conference
proceedings

20th USENIX Symposium on Operating Systems Design and Implementation (OSDI '26)

Seattle, WA, USA
July 13–15, 2026

Sponsored by
USENIX
THE ADVANCED COMPUTING SYSTEMS ASSOCIATION
In cooperation with ACM SIGOPS

USENIX
Proceedings of the 20th USENIX Symposium on Operating Systems Design and Implementation (OSDI '26) *Seattle, WA, USA* *July 13–15, 2026*

ISBN 978-1-939133-55-7

OSDI '26 Sponsors

Platinum Sponsor

amazon

Gold Sponsors

Alibaba

ByteDance

databricks

FUTUREWEI Technologies

Meta

Open Access Sponsor

جامعة الملك عبدالله للعلوم والتقنية
King Abdullah University of Science and Technology

USENIX Supporters

USENIX Patrons

Bloomberg • Google

Open Access Supporter

Google

Open Access Publishing Partner

PeerJ

USENIX Association

Proceedings of the 20th USENIX Symposium on Operating Systems Design and Implementation (OSDI '26)

July 13–15, 2026
Seattle, WA, USA

© 2026 by The USENIX Association

All Rights Reserved

This volume is published as a collective work. Rights to individual papers remain with the author or the author's employer. Permission is granted for the noncommercial reproduction of the complete work for educational or research purposes. Permission is granted to print, primarily for one person's exclusive use, a single copy of these Proceedings. USENIX acknowledges all trademarks herein.

ISBN 978-1-939133-55-7

Symposium Organizers

Program Co-Chairs
Eddie Kohler, *Harvard University*
Amar Phanishayee, *NVIDIA*

Program Committee
Reto Achermann, *The University of British Columbia*
Atul Adya, *Databricks*
Nitin Agrawal, *Google*
Irfan Ahmad, *Magnition*
Aditya Akella, *The University of Texas at Austin*
Ramnatthan Alagappan, *University of Illinois Urbana–Champaign*
Deniz Altınbüken, *Google Deepmind*
Ganesh Ananthanarayanan, *Microsoft*
Behnaz Arzani, *Microsoft Research*
Mahesh Balakrishnan, *Meta*
Oana Balmau, *McGill University*
Adam Belay, *Massachusetts Institute of Technology*
Daniel S. Berger, *Microsoft*
Ankit Bhardwaj, *Tufts University*
Ken Birman, *Cornell University*
Nathan Bronson, *OpenAI*
Anton Burtsev, *The University of Utah*
Haibo Chen, *Shanghai Jiao Tong University*
Vijay Chidambaram, *The University of Texas at Austin*
Asaf Cidon, *Columbia University*
Paolo Costa, *Microsoft Research*
Natacha Crooks, *University of California, Berkeley*
Kinan Dak Albab, *Boston University*
Murat Demirbas, *MongoDB Research*
Eric Eide, *University of Utah*
Pedro Fonseca, *Purdue University*
Xinwei (Mason) Fu, *Amazon Web Services*
Aishwarya Ganesan, *University of Illinois Urbana–Champaign*
Ada Gavrilovska, *Georgia Institute of Technology*
Kartik Gopalan, *Binghamton University*
Ronghui Gu, *Columbia University*
Andreas Haeberlen, *University of Pennsylvania and Roblox*
Dongsu Han, *Korea Advanced Institute of Science and Technology (KAIST)*
Tim Harris, *OpenAI*
Michio Honda, *University of Edinburgh*
Jon Howell, *VMware Research*
Liting Hu, *University of California, Santa Cruz*
Yu Hua, *Huazhong University of Science and Technology*
Ryan Huang, *University of Michigan*
Anand Iyer, *Georgia Institute of Technology*
Rishabh Iyer, *University of California, Berkeley*
Junchen Jiang, *The University of Chicago*
Rohan Kadekodi, *University of Washington*
Kostis Kaffes, *Columbia University*
Konstantinos Kallas, *University of California, Los Angeles*
Sudarsun Kannan, *Rutgers University*
Manos Kapritsos, *University of Michigan*
Fiodar Kazhamiaka, *Microsoft*
Daehyeok Kim, *The University of Texas at Austin*
Marios Kogias, *Imperial College London*
Dejan Kostić, *KTH Royal Institute of Technology*
Arvind Krishnamurthy, *University of Washington and Google*
Sam Kumar, *University of California, Los Angeles*
Sándor Laki, *Eötvös Loránd University*
Alberto Lerner, *Computing Flows GmbH*
Philip Levis, *Stanford University and Google*
Amit Levy, *Better Bytes and Princeton University*
Jinyang Li, *New York University*
Yu Liang, *Inria*
Hongqiang Harry Liu, *Uber*
Jing Liu, *Microsoft Research*
Jay Lorch, *Microsoft Research*
Xiaosong Ma, *Mohamed bin Zayed University of Artificial Intelligence*
Sara McAllister, *Google and University of Wisconsin–Madison*
Jayashree Mohan, *Microsoft Research*
Shuai Mu, *Stony Brook University*
Kiran-Kumar Muniswamy-Reddy, *MongoDB*
Deepak Narayanan, *NVIDIA*
Shadi Noghabi, *Google*
Pierre Olivier, *The University of Manchester*
Amy Ousterhout, *University of California, San Diego*
Aurojit Panda, *New York University*
Peter Pietzuch, *Imperial College London*
Soujanya Ponnapalli, *University of California, Berkeley*
Dan Ports, *Microsoft Research*
Vijayan Prabhakaran, *Databricks*
Andi Quinn, *University of California, Santa Cruz*
Deepti Raghavan, *Brown University*
K. K. Ramakrishnan, *University of California, Riverside*
Ramachandran Ramjee, *Microsoft Research*
Robert Ricci, *The University of Utah*
Luis Rodrigues, *INESC-ID and Instituto Superior Técnico, University of Lisbon*
Keshav Santhanam, *NVIDIA*
Malte Schwarzkopf, *Brown University*
Margo Seltzer, *The University of British Columbia*
Jiacheng Shen, *Duke Kunshan University*
Ji-Yong Shin, *Northeastern University*
Liuba Shrira, *Brandeis University*
Dimitrios Skarlatos, *Carnegie Mellon University*
Alex C. Snoeren, *University of California, San Diego*
Ryan Stutsman, *The University of Utah and Stellar Development Foundation*
Adriana Szekeres, *Microsoft Research*
Runzhou Tao, *University of Maryland*
Nikos Vasilakis, *Brown University*
Shivaram Venkataraman, *University of Wisconsin–Madison*
Marco Vieira, *University of North Carolina at Charlotte*
Chenxi Wang, *Institute of Computing Technology, Chinese Academy of Sciences*
Stephanie Wang, *University of Washington*
Yang Wang, *The Ohio State University and Meta*
Zhuang Wang, *Amazon Web Services*

Jason Waterman, *Vassar College*
Dan Williams, *Virginia Tech*
Emmett Witchel, *The University of Texas at Austin*
Kan Wu, *xAI*
Erci Xu, *Shanghai Jiao Tong University and Alibaba Cloud*
Tianyin Xu, *University of Illinois Urbana–Champaign*
Neeraja J. Yadwadkar, *The University of Texas at Austin*
Francis Y. Yan, *University of Illinois Urbana–Champaign*
Fan Yang, *Microsoft Research*
Juncheng Yang, *Harvard University*
Junfeng Yang, *Columbia University*
Ming-Chang Yang, *The Chinese University of Hong Kong*
Nickolai Zeldovich, *Massachusetts Institute of Technology*
Jian Zhang, *The Hong Kong University of Science and Technology*
Jie Zhang, *Peking University*
Yiying Zhang, *University of California, San Diego*
Yongle Zhang, *Purdue University*

Poster Session Co-Chairs
Daehyeok Kim, *The University of Texas at Austin*
Shadi Noghabi, *Google*

Mentoring Chair
Stephanie Wang, *University of Washington*

Artifact Evaluation Committee Co-Chairs
Anand Iyer, *Georgia Institute of Technology*
Sara McAllister, *Google and University of Wisconsin–Madison*

Artifact Evaluation Committee
Abhinav Agarwal, *Rubrik*
Vishal Agarwal, *NVIDIA*
Anirudha Agrawal, *Georgia Tech*
Kenan Alghythee, *University of Illinois Chicago*
Abhijeet Apsunde, *eBay Inc*
Sanjith Athlur, *Carnegie Mellon University*
Abdelrahman Baba, *University of Waterloo*
Seungsu Baek, *Korea University*
Luyao Bai, *University of Illinois Chicago*
Jongho Baik, *Korea University*
Afsara Benazir, *University of Virginia*
Ayush Bhardwaj, *Akamai Technologies*
Romil Bhardwaj, *SkyPilot*
Cláudia Brito, *INESC TEC and University of Minho*
Zechao Cai, *ETH Zurich*
Noppapon Chalermchockcharoenkit, *University of California, Berkeley, Sky Lab, and Crusoe AI*
Manas Chaudhari, *Meta*
Jiahao Chen, *Georgia Tech*
Deebakkarthi Chinnasame Rani, *University of Virginia*
Hayden Coffey, *University of Wisconsin–Madison*
Chihan Cui, *University of Wisconsin–Madison*
Ahmed Dajani, *Iowa State University*
Yi Deng, *University of Virginia*
Richi Dubey, *Georgia Institute of Technology*
Oytun Kuday Duran, *Georgia Tech*
Rohit Dwivedula, *University of Texas at Austin*
Tânia Esteves, *INESC TEC and University of Minho*
Swapnil Gandhi, *Stanford University*
Pratishtha Gaur, *University of California, San Diego*
Maryam Ghorbanvirdi, *George Washington University*
Sarmistha Sarna Gomasta, *University of Massachusetts Amherst*
Raja Gond, *Microsoft Research India*
Krishna Praneet Gudipaty, *University of Massachusetts Amherst*
Ishita Gupta, *Alai Studios*
Kiran Hombal, *University of Illinois Urbana–Champaign*
Lana Honcharuk, *Crusoe*
Fariha Tabassum Islam, *University of Wisconsin–Madison*
Priyanka Jayaswal, *Microsoft*
Dongming Jiang, *The University of Texas at Dallas*
Yiwei Jiang, *University of Wisconsin–Madison*
Yuxuan Jiang, *SambaNova Systems*
Nitin Kedia, *The University of Texas at Austin*
Tuhin Khare, *Georgia Institute of Technology*
Donghyun Kim, *The University of Texas at Dallas*
Geonwoo Kim, *The University of Texas at Austin*
Pavani Kuppili, *University of Utah*
Mahesh Lakshminarasimhan, *AMD USA*
Tingfeng Lan, *University of Virginia*
Tyler Landle, *Georgia Institute of Technology*
Kinman Lei, *Tsinghua University*
Jingzong Li, *The Hang Seng University of Hong Kong*
Lanzhe Li, *Nanjing University*
Nanqinqin Li, *Princeton University*
Zhiyue Li, *Tsinghua University*
Wei Siew Liew, *University of Utah*
Darren Lim, *Crusoe*
Chungang Lin, *Institute of Computing Technology, Chinese Academy of Sciences*
Xiaoyuan Liu, *University of California, Berkeley*
Yuxuan Liu, *University of Virginia*
Qingjie Lu, *University of Pennsylvania*
Muxi Lyu, *University of California, Berkeley*
Manvik Nanda, *University of Utah*
Harald Ng, *KTH Royal Institute of Technology*
Mary Ogunmola, *University of Utah*
Zhiyang Pan, *University of California, Berkeley*
Dimas Parikesit, *University of Virginia*
Shishir Patil, *University of California, Berkeley*
Stratos Psomadakis, *National Technical University of Athens*
Liang Qiao, *University of Science and Technology of China*
Ruoyu Qin, *Tsinghua University*
Shriram Raja, *Boston University*
Vishnu Ramadas, *University of Wisconsin–Madison*
Yeonju Ro, *University of Texas at Austin*
Giannis Roumpos, *IMDEA Software Institute*
Amit Samanta, *University of Utah*
Hetvi Shastri, *University of Massachusetts Amherst*
Hang Shen, *Northeastern University (China)*
Muhammad Shoaib, *University of Virginia*
Akshat Sinha, *Rubrik, Inc*
Eunseok Song, *Korea University*
Sheetal Sriram, *Georgia Institute of Technology*
Thanathorn Sukprasert, *University of Massachusetts Amherst*
Tong Sun, *Zhejiang University*
Mengze Tang, *University of Wisconsin–Madison*
Zirui Wang, *University of Virginia*

Li Wu, *University of Massachusetts Amherst*
Yibo Wu, *University of Washington*
Tian Xia, *University of California, Berkeley*
Zeyu Xia, *University of Virginia*
Chengpo Yan, *University of Wisconsin–Madison*
Xufeng Yang, *Xiamen University, China*
Yuanyuan Yang, *University of Washington*
Zeren Yang, *Microsoft*
Haoran Zhang, *University of Virginia*
Sen Zhang, *University of Science and Technology of China*
Zhiwei Zheng, *University of Pennsylvania*
Yanbo Zhou, *University of California, San Diego*
Tianyu Zuo, *University of Virginia*

Steering Committee
Marcos K. Aguilera, *VMware Research*
Angela Demke Brown, *University of Toronto*
Casey Henderson-Ross, *USENIX Association*
Jon Howell, *VMware Research*
Kimberly Keeton, *Google*
Jay Lorch, *Microsoft Research*
Shan Lu, *University of Chicago*
Timothy Roscoe, *ETH Zurich*
Geoff Voelker, *University of California, San Diego*
Hakim Weatherspoon, *Cornell University and Exosteller, Inc.*

Message from the OSDI '26 Program Co-Chairs

Welcome to the 20th USENIX Symposium on Operating Systems Design and Implementation (OSDI '26).

We are grateful to our reviewers for their work, to our authors for theirs, to our artifact evaluation committee (co-chaired by Anand Iyer and Sara McAllister), to the poster session chairs (Daehyeok Kim and Shadi Noghabi), to the mentorship activity chair (Stephanie Wang), to the ever-ready USENIX staff, and to all attendees. A vibrant OSDI is a three-day celebration of the best in computing systems, and this one shouldn't disappoint.

OSDI '26 received 669 reviewable submissions, more than twice any previous year. Authors came from more than 30 countries. We pre-committed to accepting at least 20% of the submissions, and we met that commitment (barely), accepting 134—around 2.5x any previous year. (Two additional submissions were accepted via revise-and-resubmit from OSDI '25.) Roughly a third of the acceptances were conditional, subject to approval via stringent shepherding; thanks to heroic efforts from both authors and the PC, all conditionally accepted papers will be presented. Only a multi-track format could accommodate substantive presentations for so much great work, so this is the first multi-track OSDI.

We accepted two kinds of submissions, Research and Operational Systems. The reviewing process for Operational Systems papers focused on design, implementation, analysis, and experience in a real-world operational context. Nineteen accepted papers are designated as Operational Systems.

Beyond Operational Systems, we changed the review process by removing the author response period, replacing revise-and-resubmit with conditional acceptance, and reducing the minimum number of reviews per submission to two. These changes were broadly successful; our attempt to encourage concise submissions was not: only three accepted papers were submitted below the page limit.

Our program committee of 117 incredible researchers, drawn from all over the world, produced 1826 penetrating reviews and more than 4000 comments. Thank you.

Now, enjoy OSDI!

Eddie Kohler, *Harvard University*
Amar Phanishayee, *NVIDIA*
OSDI '26 Program Co-Chairs

20th USENIX Symposium on Operating Systems Design and Implementation (OSDI '26)

July 13–15, 2026
Seattle, WA, USA

Monday, July 13

KV Cache and Long Context

Strata: Hierarchical Context Caching for Long Context Language Model Serving 1
Zhiqiang Xie, *Stanford University and NVIDIA;* Ziyi Xu, *Shanghai Jiao Tong University;* Mark Zhao, *University of Colorado Boulder;* Yuwei An, *Carnegie Mellon University;* Vikram Sharma Mailthody, *NVIDIA;* Scott Mahlke, *NVIDIA and University of Michigan;* Michael Garland, *NVIDIA;* Christos Kozyrakis, *NVIDIA and Stanford University*

ECHO: Efficient KV Cache Offloading with Lossless Prefetching for Serving Native Sparse Attention LLMs 17
Guangda Liu, Wenhao Chen, Chengwei Li, and Zhenyu Ning, *Shanghai Jiao Tong University;* Jing Lin and Yiwu Yao, *Huawei;* Quan Chen, Shixuan Sun, and Jieru Zhao, *Shanghai Jiao Tong University;* Minyi Guo, *Guizhou University and Shanghai Jiao Tong University*

No Buffer, No Bottleneck: Efficient Zero-Copy KV Cache Offloading for Long-Context LLMs 39
Shutian Luo and Haiying Shen, *University of Virginia*

Simple is Better: Multiplication May Be All You Need for LLM Request Scheduling 55
Dingyan Zhang, Jinbo Han, Kaixi Zhang, and Xingda Wei, *Shanghai Jiao Tong University;* Sijie Shen, Chenguang Fang, Wenyuan Yu, and Jingren Zhou, *Alibaba Group;* Rong Chen, *Shanghai Jiao Tong University*

Prism: Cost-Efficient Multi-LLM Serving via GPU Memory Ballooning 75
Shan Yu, *University of California, Los Angeles;* Yifan Qiao, *University of California, Berkeley;* Mingyuan Ma, *Harvard University;* Yangmin Li, *Carnegie Mellon University;* Shuo Yang, *University of California, Berkeley;* Xinyuan Tong, *University of Edinburgh;* Yang Wang, *Intel;* Zhiqiang Xie, *Stanford University;* Yuwei An, *Carnegie Mellon University;* Shiyi Cao, *University of California, Berkeley;* Ke Bao, *LMSYS;* Deepak Vij, Xiaoning Ding, and Yichen Wang, *ByteDance;* Qingda Lu, *Alibaba Cloud;* Zhong Wang, *Tsinghua University;* Gao Gao, *Novita AI;* Harry Xu and Junyi Shu, *University of California, Los Angeles;* Jiarong Xing, *Rice University;* Ying Sheng, *University of California, Los Angeles*

Memory Tiering and CXL

Break on Through to the Other Side: Pooling Memory Elastically with RamRyder 95
Yanbo Zhou, *University of California, San Diego;* Erci Xu, *Shanghai Jiao Tong University;* Dongjoo Seo and Adam Manzanares, *Samsung Semiconductor;* Steven Swanson, *University of California, San Diego*

MAC: Metadata Acceleration for Sustainable Performance in Big-Data Systems with CXL DRAM. 115
Dusol Lee, *Seoul National University;* Yan Sun, Houxiang Ji, Vinit Gupta, and Austin Antony Cruz, *University of Illinois Urbana–Champaign;* Inhyuk Choi, *Seoul National University;* Nam Sung Kim, *University of Illinois Urbana–Champaign;* Jihong Kim, *Seoul National University*

Finding NEMO: Nimble and Expressive Memory Observability 133
Shihang Li and Matthew Giordano, *University of Washington;* Tushar Garg, *Meta;* Rohan Kadekodi, *University of Washington;* Daniel S. Berger, *University of Washington and Microsoft Azure;* Baris Kasikci, Thomas Anderson, and Simon Peter, *University of Washington*

OBASE: Object-Based Address-Space Engineering to Improve Memory Tiering 151
Vinay Banakar, *University of Wisconsin–Madison and Google;* Suli Yang, *Google;* Kan Wu, *xAI;* Andrea C. Arpaci-Dusseau and Remzi H. Arpaci-Dusseau, *University of Wisconsin–Madison;* Kimberly Keeton, *Google*

MDK: Rethinking the data center memory reclamation problem 167
Shaurya Patel, *Google and University of British Columbia;* Suli Yang and Yawen Wang, *Google;* Kan Wu, *xAI;* Alexandra (Sasha) Fedorova, *University of British Columbia and MongoDB;* Margo Seltzer, *University of British Columbia;* Kimberly Keeton, *Google*

Sandboxing, Access Control, and Drivers

USEC: A User-Requirement-Driven Mandatory Access Control Framework for Operating Systems (Operational Systems) . . . 191
Yu Jiang, *Tsinghua University;* Wenhuan Liu, *Tsinghua University and UnionTech Software Technology Co., Ltd;* Fuchen Ma, Yuheng Shen, and Yuanliang Chen, *Tsinghua University;* Lei Zhang and He Li, *UnionTech Software Technology Co., Ltd.;* Quan Zhang and Chijin Zhou, *East China Normal University*

Mohabi: Disaggregating and Sandboxing the Firefox JavaScript Engine . . . 207
Abhishek Sharma and Anand Balaji, *The University of Texas at Austin;* Zachary Yedidia, *Stanford University;* Anthony Du and Taehyun Noh, *The University of Texas at Austin;* Iain Ireland, Jan de Mooij, and Matthew Gaudet, *Mozilla;* Tal Garfinkel, *Google;* Deian Stefan and Hovav Shacham, *University of California, San Diego;* Shravan Narayan, *The University of Texas at Austin*

Ichnaea: A Framework for Precise Tracking of Memory Objects . . . 229
Samad Haque and Sibin Mohan, *The George Washington University;* Aaron Paulos and Partha Pal, *RTX BBN Technologies*

Extracting Database Access-Control Policies From Web Applications. . . . 249
Wen Zhang, Dev Bali, and Jamison Kerney, *University of California, Berkeley;* Aurojit Panda, *NYU;* Scott Shenker, *University of California, Berkeley, and ICSI*

***iLand*: An Instruction-Level Dynamic Binary Instrumentation framework for iOS . . . 271**
Kaitao Xie, Yizhuo Wang, and Xiaolong Bai, *Alibaba Group*

LLM Training at Scale

Tessera: A Holistic Pipeline Parallelism Framework for Trillion-Parameter Heterogeneous MoE Training (Operational Systems). . . . 287
Weifang Hu, *Huazhong University of Science and Technology;* Langshi Chen, Man Yuan, Youyang Yao, Xiulong Yuan, Li Tian, Yong Li, and Wei Lin, *Alibaba Cloud;* Xuanhua Shi, *Huazhong University of Science and Technology;* Zhengping Qian and Jingren Zhou, *Alibaba Cloud*

Hetu v2: A General and Scalable Deep Learning System with Hierarchical and Heterogeneous Single Program Multiple Data Annotations. . . . 305
Haoyang Li, *Peking University;* Fangcheng Fu, *Shanghai Jiao Tong University;* Hao Ge, Sheng Lin, Xuanyu Wang, Jiawen Niu, and Yuming Zhou, *Peking University;* Xupeng Miao, *Purdue University;* Bin Cui, *Peking University and Peking University (Qingdao)*

Syncopate: Efficient Multi-GPU AI Kernels via Automatic Chunk-Centric Compute-Communication Overlap . . . 331
Xinwei Qiang, Yue Guan, and Zhengding Hu, *University of California, San Diego;* Keren Zhou, *George Mason University and OpenAI;* Yufei Ding, *University of California, San Diego, and Meta;* Adnan Aziz, *Meta*

Teaching The Old Dog New Tricks: Building Efficient Data Pipelines for Large-Scale LLM Pre-training (Operational Systems). . . . 349
Luofan Chen and Chenhan Wang, *University of Science and Technology of China and ByteDance Seed;* Weidong Zhang, Jinxin Chi, Hequan Zhang, Zanbo Wang, Chenyuan Wang, Lishu Luo, Sijin Wu, Junqi Hu, Jun Wang, and Cheng Chen, *ByteDance Seed;* Lixin Huang, Liyang Zhao, Yong Tian, and Jun Guo, *ByteDance;* Youhui Bai, *University of Science and Technology of China;* Wencong Xiao, *ByteDance Seed;* Kang Chen, *Tsinghua University;* Cheng Li, *University of Science and Technology of China and Institute of Artificial Intelligence, Hefei Comprehensive National Science Center*

Cocoon: A System Architecture for Differentially Private Training with Correlated Noises . . . 367
Donghwan Kim and Xin Gu, *The Pennsylvania State University;* Jinho Baek, Timothy Lo, Younghoon Min, Kwangsik Shin, and Jongryool Kim, *SK hynix Inc.;* Jongse Park, *Korea Advanced Institute of Science and Technology (KAIST);* Kiwan Maeng, *The Pennsylvania State University*

Testing, Debugging, and Root Causing

ValScope: Value-Semantics-Aware Metamorphic Testing for Detecting Logical Bugs in DBMSs . . . 387
Li Lin, Liehang Chen, and Rongxin Wu, *Xiamen University*

The Abstention Protocol: RCA for Clos Fabrics (Operational Systems). . . . 405
Madhava Gaikwad, *Independent;* and Deepak Pandey, *Microsoft*

When Sampling Lies: Trustworthy Performance Profiling for Flat Workloads with Blink (Operational Systems) **423**
Rishikesh Devsot, *YScope;* ChenXing Yang and Yi Fan Yu, *YScope and University of Toronto;* Prabhdeep Singh Soni, Afshin Arefi, Bryan Chan, and Reza Azimi, *Huawei Technologies Canada;* Ding Yuan, *YScope and University of Toronto*

Breaking the Reward Barrier: Accelerating Tree-of-Thought Reasoning via Speculative Exploration **437**
Shuzhang Zhong, Haochen Huang, and Shengxuan Qiu, *Peking University;* Pengfei Zuo, *ByteDance Seed;* Runsheng Wang and Meng Li, *Peking University*

Controlling Opaque-Component Effects with Semisolates and Try **453**
Evangelos Lamprou, *Brown University;* Tianyu (Ezri) Zhu, *Stevens Institute of Technology;* Di Jin and Grigoris Ntousakis, *Brown University;* Georgios Liargkovas, *Columbia University;* Calvin Eng, *Brown University;* Konstantinos Kallas, *University of California, Los Angeles;* Michael Greenberg, *Stevens Institute of Technology;* Nikos Vasilakis, *Brown University*

Kernel Scheduling and Tail Latency

SBB: Eliminating Centralized Bottlenecks in Userspace Network Runtime **473**
Kang Hu, *Peking University and Zhongguancun Laboratory;* Shuqi Dong, *Peking University;* Chuandong Li, *Peking University and Zhongguancun Laboratory;* Ran Yi, *Peking University;* Zonghao Zhang, *Peking University and Zhongguancun Laboratory;* Yiming Yao, *Peking University;* Bo An, *Zhongguancun Laboratory;* Jie Zhang, Xiaolin Wang, and Yingwei Luo, *Peking University and Zhongguancun Laboratory;* Zhenlin Wang, *Michigan Technological University;* Diyu Zhou, *Peking University*

Rakaia: Scalable In-Kernel Scheduling for TCP-Based RPCs **491**
Rui Yang, Konstantinos Prasopoulos, and Edouard Bugnion, *EPFL*

kSTEP: Characterization and Deterministic Testing of Linux CPU Scheduler Bugs **509**
Tingjia Cao, Shawn (Wanxiang) Zhong, and Caeden Whitaker, *University of Wisconsin–Madison;* Ke Han, *Purdue University;* Andrea C. Arpaci-Dusseau and Remzi H. Arpaci-Dusseau, *University of Wisconsin–Madison*

What Are You (M)Waiting For: The Hidden Cost of Idle in the Hyperscale Cloud (Operational Systems) **529**
Yun Wang, *Shanghai Jiao Tong University;* Xingguo Jia, *Alibaba Cloud;* Ben Luo and Kenan Liu, *Alibaba Group;* Shengdong Dai, *Alibaba Cloud;* Jingdong Han and Weihao Chen, *Alibaba Group;* Yicheng Gu and Xingzi Yu, *Shanghai Jiao Tong University;* Yibin Shen and Jiesheng Wu, *Alibaba Cloud;* Zhengwei Qi and Haibing Guan, *Shanghai Jiao Tong University*

Xkernel: Principled Performance Tunability of Operating System Kernels **547**
Zhongjie Chen, *Tsinghua University and Microsoft Research;* Wentao Zhang, *University of Illinois Urbana–Champaign;* Yulong Tang and Ran Shu, *Microsoft Research;* Fengyuan Ren, *Tsinghua University;* Tianyin Xu, *University of Illinois Urbana–Champaign;* Jing Liu, *Microsoft Research*

Agentic AI and LLM Operations

Murakkab: Resource-Efficient Agentic Workflow Orchestration in Cloud Platforms **567**
Gohar Irfan Chaudhry, *MIT CSAIL;* Esha Choukse, Haoran Qiu, Íñigo Goiri, and Rodrigo Fonseca, *Microsoft Azure Research;* Adam Belay, *MIT CSAIL;* Ricardo Bianchini, *Microsoft Azure*

ECO: An AI-Driven Code Efficiency Optimizer for Warehouse Scale Computers (Operational Systems) **589**
Hannah Lin and Martin Maas, *Google DeepMind;* Maximilian Roquemore, *Google;* Arman Hasanzadeh, *Google DeepMind;* Fred Lewis, Yusuf Simonson, Ameya Shringi, and Hongwen Dai, *Google;* Patrick Musau, *Google DeepMind;* Tzu-Wei Yang, *Google;* Amir Yazdanbakhsh and Deniz Altinbüken, *Google DeepMind;* Florin Papa, Maggie Nolan Edmonds, Aditya Patil, Don Schwarz, Satish Chandra, and Chris Kennelly, *Google;* Milad Hashemi, *Google DeepMind;* Parthasarathy Ranganathan, *Google*

StriaTrace: Efficient Tracing and Diagnosis for Online LLM Inference (Operational Systems) **627**
Haonan Wu, *Shanghai Jiao Tong University and Alibaba Group;* Yanqing Chen, Kun Qian, Xue Li, and Jingbo Xu, *Alibaba Group;* Erci Xu, *Shanghai Jiao Tong University;* Ennan Zhai and Wenyuan Yu, *Alibaba Group;* Guangtao Xue, *Shanghai Jiao Tong University and Shanghai Key Laboratory of Trusted Data Circulation and Governance and Web3;* Jingren Zhou, *Alibaba Group*

Diagnosing Performance Issues in Application-Defined Resources **647**
Yigong Hu and You-Liang Huang, *Boston University;* Haodong Zheng, *University of Washington and EPFL;* Yicheng Liu, *University of Washington and University of California, Los Angeles;* Dedong Xie and Baris Kasikci, *University of Washington*

Program Analysis

hS: Speculative Script Reordering at Subprocess Granularity 665
Georgios Liargkovas and Di Jin, *Brown University;* Tianyu (Ezri) Zhu and Dan Liu, *Stevens Institute of Technology;* A. Bolun Thompson, *University of California, Los Angeles;* Anirudh Narsipur, Seong-Heon Jung, and Siddhartha Prasad, *Brown University;* Diomidis Spinellis, *Athens University of Economics and Business and TU Delft;* Michael Greenberg, *Stevens Institute of Technology;* Konstantinos Kallas, *University of California, Los Angeles;* Nikos Vasilakis, *Brown University*

Incr: Faster Re-execution via Bolt-on Incrementalization 683
Yizheng Xie, Evangelos Lamprou, Jerry Xia, and Nikos Vasilakis, *Brown University*

A Compilation-based Under-Constrained Execution Engine 701
Mingjun Yin, Zhaorui Li, Ju Chen, Haochen Zeng, and Chengyu Song, *University of California, Riverside*

Aletheia: Automated Detection of Data Integrity Violations in Microservices 721
Mafalda Sofia Ferreira, João Ferreira Loff, João Garcia, and Rodrigo Rodrigues, *INESC-ID, Instituto Superior Técnico, Universidade de Lisboa*

Synchronization

Arctic: a practical lock-free adaptive radix tree 739
Newton Ni, Nicolas Garza, Jenny Stinehour, and Michael Goppert, *The University of Texas at Austin;* Michal Friedman, *ETH Zürich;* Emmett Witchel, *The University of Texas at Austin*

Efficient and Scalable Synchronization via Generalized Cache Coherence 755
Yanpeng Yu, Seung-seob Lee, Lin Zhong, and Anurag Khandelwal, *Yale University*

Shaving the Peaks: Taming Tail Latency for Managed Workloads via Disaggregated Garbage Collection 773
Hongtao Lyu, Yuhan Li, and Mingyu Wu, *Shanghai Jiao Tong University*

DeLFS: A Decentralized Log-Structured File System for Manycores 793
Taehwan Ahn, Chanhyeong Yu, Sangjin Lee, and Yongseok Son, *Chung-Ang University*

Tuesday, July 14

RL Training at Scale

Weave: Efficient Co-Scheduling for Disaggregated RL Post-Training 809
Tianyuan Wu and Lunxi Cao, *Hong Kong University of Science and Technology;* Yining Wei, *University of Illinois Urbana–Champaign;* Wei Gao, Yuheng Zhao, and Dakai An, *Hong Kong University of Science and Technology;* Shaopan Xiong, Zhiqiang Lv, Ju Huang, Siran Yang, Yinghao Yu, Jiamang Wang, and Lin Qu, *Alibaba Group;* Wei Wang, *Hong Kong University of Science and Technology*

RLinf: Flexible and Efficient Large-Scale Reinforcement Learning via Macro-to-Micro Flow Transformation . . . 829
Chao Yu, *Tsinghua University;* Yuanqing Wang, *Infinigence AI and Peking University;* Zhen Guo, Hao Lin, and Si Xu, *Infinigence AI;* Hongzhi Zang, *Tsinghua University;* Quanlu Zhang, *Infinigence AI;* Yongji Wu, *University of California, Berkeley;* Chunyang Zhu and Junhao Hu, *Infinigence AI;* Zixiao Huang, *Tsinghua University and Infinigence AI;* Mingjie Wei, *Zhongguancun Academy;* Yuqing Xie, *Tsinghua University;* Ke Yang, *Zhongguancun Academy;* Bo Dai, *Beihang University and Infinigence AI;* Zhexuan Xu and Jiakun Du, *Tsinghua University;* Xiangyuan Wang, *Peking University and Infinigence AI;* Xu Fu and Letong Shi, *Infinigence AI;* Zhihao Liu, *Zhongguancun Academy;* Kang Chen, *Peking University and Zhongguancun Academy;* Weilin Liu, *Infinigence AI;* Gang Liu, *Tsinghua University;* Boxun Li, *Infinigence AI;* Jianlei Yang, *Beihang University;* Zhi Yang, *Peking University;* Guohao Dai, *Shanghai Jiao Tong University and Infinigence AI;* Yu Wang, *Tsinghua University*

DynaRL: Flexible and Dynamic Scheduling of Large-Scale Reinforcement Learning Training 847
Yuanqing Wang, *Peking University and Infinigence AI;* Hao Lin, Junhao Hu, Chunyang Zhu, Quanlu Zhang, and Zhen Guo, *Infinigence AI;* Yuchen Zhang, *Institute of Computing Technology, Chinese Academy of Sciences and Infinigence AI;* Xu Fu and Si Xu, *Infinigence AI;* Bo Dai, *Beihang University and Infinigence AI;* Zixiao Huang, *Tsinghua University and Infinigence AI;* Chao Yu, *Tsinghua University;* Boxun Li, *Infinigence AI;* Guohao Dai, *Shanghai Jiao Tong University and Infinigence AI;* Zhi Yang, *Peking University;* Yu Wang, *Tsinghua University*

RollArt: Disaggregated Multi-Task Agentic RL Training at Scale 863
Wei Gao, Yuheng Zhao, and Tianyuan Wu, *Hong Kong University of Science and Technology;* Shaopan Xiong and Weixun Wang, *Alibaba Group;* Dakai An and Lunxi Cao, *Hong Kong University of Science and Technology;* Dilxat Muhtar, Zichen Liu, Haizhou Zhao, Ju Huang, and Siran Yang, *Alibaba Group;* Yongbin Li, *Tongyi Lab, Alibaba;* Wenbo Su, Jiamang Wang, Lin Qu, and Bo Zheng, *Alibaba Group;* Wei Wang, *Hong Kong University of Science and Technology*

Seer: Online Context Learning for Fast Synchronous LLM Reinforcement Learning. 883
Ruoyu Qin, *Moonshot AI and Tsinghua University;* Weiran He, Weixiao Huang, Yangkun Zhang, Yikai Zhao, Bo Pang, and Xinran Xu, *Moonshot AI;* Yingdi Shan, Yongwei Wu, and Mingxing Zhang, *Tsinghua University*

Disaggregated Memory Systems

Harvesting Sub-Microsecond CXL Memory Stalls with LiteSwitch. 903
Nanqinqin Li, *Princeton University;* Yuhong Zhong and Asaf Cidon, *Columbia University;* Michael J. Freedman, *Princeton University*

Duhu: Shared Disaggregated Memory for Distributed Data Processing Frameworks 921
Qiutong Men and Tao Wang, *New York University;* Jongryool Kim and Hane (Stella) Yie, *SK hynix;* Emmanuel Amaro, *Microsoft;* Marcos K. Aguilera, *NVIDIA;* Aurojit Panda, *New York University*

Blowfish: Elastic Virtual Machine Memory for Disaggregated Memory 939
Yulong Zhang, *SKLP, Institute of Computing Technology, CAS and University of Chinese Academy of Sciences;* Yilong Luo and Diyu Zhou, *SCS, Peking University, China;* Quan Chen, *Shanghai Jiao Tong University;* Quanxi Li, *SKLP, Institute of Computing Technology, CAS and University of Chinese Academy of Sciences;* Mosong Zhou, Lei Zhu, Senbo Fu, and Qian Peng, *Huawei Cloud;* Huimin Cui and Xiaobing Feng, *SKLP, Institute of Computing Technology, CAS and University of Chinese Academy of Sciences;* Tao Xie, *SCS, Peking University, China;* Chenxi Wang, *SKLP, Institute of Computing Technology, CAS and University of Chinese Academy of Sciences*

Espresso: Constructing Cost-Efficient CXL JBOF via Inter-SSD Computing Resource Sharing 957
Shushu Yi, Yuda An, Li Peng, and Xiurui Pan, *Peking University;* Qiao Li, *Mohamed bin Zayed University of Artificial Intelligence;* Jieming Yin, *Nanjing University of Posts and Telecommunications;* Guangyan Zhang, *Tsinghua University;* Wenfei Wu, *Peking University;* Chenxi Wang, *University of Chinese Academy of Sciences;* Diyu Zhou, *Peking University;* Zhenlin Wang, *Michigan Tech;* Xiaolin Wang and Yingwei Luo, *Peking University;* Ke Zhou, *Huazhong University of Science and Technology* (*HUST*)*;* Jie Zhang, *Peking University*

FORGE: Mitigating Synchronization Amplification for Memory-Disaggregated Caching Systems. 977
Zhijun Yang, Yu Hua, Ming Zhang, Menglei Chen, and Yixiao Wang, *Huazhong University of Science and Technology*

Confidential Computing

Accelerating Confidential Databases with Crypto-free Mappings 997
Wenxuan Huang, Zhanbo Wang, and Mingyu Li, *Key Laboratory of System Software, Chinese Academy of Sciences, and Institute of Software, Chinese Academy of Sciences, and University of Chinese Academy of Sciences*

Janus: Cross-World, Cooperative Nested Virtualization for Secure Containers 1015
Jiangshan Lai, *Ant Group;* Hang Huang, *Alibaba Cloud and Huazhong University of Science and Technology;* Quan Xu and Zhen Ren, *Alibaba Cloud;* Wenlong Hou, *Ant Group;* Wei Guo, *Alibaba Cloud;* Jia Rao and Hui Lu, *The University of Texas at Arlington;* Weidong Han, Jiesheng Wu, Jiang Liu, Naixuan Guan, and Yibin Shen, *Alibaba Cloud;* Feng Yu and Xu Wang, *Ant Group;* Shiqiang Zhang, *Alibaba Cloud;* Zhiheng Tao, *Ant Group;* Yisheng Xie, *Alibaba Cloud;* Song Wu and Hai Jin, *Huazhong University of Science and Technology*

Osprey: Transparent and Efficient Virtual Memory for Secure Computation 1033
Yicheng Liu, *University of California, Los Angeles;* Alice Yeh, *University of California, Berkeley;* Harry Xu, *University of California, Los Angeles;* Raluca Ada Popa, *University of California, Berkeley;* Sam Kumar, *University of California, Los Angeles*

Nested SEV: Secure and Generic SEV Support for Nested Virtualization. 1051
Kazuki Takiguchi and Kenichi Kourai, *Kyushu Institute of Technology*

µUSB: Practical and Safe USB Driver Reuse for Arm TrustZone 1067
Xuankai Zhang, Sijin Li, and Pei Meng, *University of Electronic Science and Technology of China;* Meng Wang, *CISPA Helmholtz Center for Information Security;* Yongzhao Zhang, Ting Chen, Xiaosong Zhang, and Liwei Guo, *University of Electronic Science and Technology of China*

Expert Mixture

Achieving Cloud-Grade SLOs for Local Mixture-of-Experts Inference through CPU–GPU Hybrid Design 1089
Wenxin Wang, *Tsinghua University;* Yule Hou and Yu Ji, *Xingyun Integrated Circuits Co., Ltd.;* Peng Qu and Youhui Zhang, *Tsinghua University and Beijing National Research Center for Information Science and Technology*

UEP: Portable Expert-Parallel Communication 1107
Ziming Mao, *University of California, Berkeley;* Yihan Zhang, *University of California, Davis;* Chihan Cui, *University of Wisconsin–Madison;* Zhen Huang, *AMD;* Kaichao You, *Independent Researcher;* Zhongjie Chen, *Tsinghua University;* Zhiying Xu, *Amazon Web Services;* Zhenyu Gu, *AMD;* Scott Shenker, *University of California, Berkeley, and ICSI;* Costin Raiciu, *Broadcom and Politehnica of Bucharest;* Yang Zhou, *University of California, Davis;* Ion Stoica, *University of California, Berkeley*

BatchGen: An Architecture for Scalable and Efficient Batch Inference 1125
Tairan Xu, Leyang Xue, and Zhan Lu, *University of Edinburgh;* Jinfu Deng and Hongyang Xiao, *Tencent;* Yinsicheng Jiang, Congjie He, Matej Sandor, Le Xu, and Luo Mai, *University of Edinburgh*

UCCL-Tran: An Extensible Software Transport Layer for GPU Networking 1143
Yang Zhou, *University of California, Berkeley, and University of California, Davis;* Zhongjie Chen, *Tsinghua University;* Ziming Mao, *University of California, Berkeley;* ChonLam Lao, *Harvard University;* Shuo Yang, *University of California, Berkeley;* Pravein Govindan Kannan, *IBM Research;* Xizhi Zhang, *Tsinghua University;* Jiaqi Gao, *Independent Researcher;* Yilong Zhao and Yongji Wu, *University of California, Berkeley;* Kaichao You, *Independent Researcher;* Fengyuan Ren, *Tsinghua University;* Zhiying Xu, *Amazon Web Services;* Costin Raiciu, *Broadcom and University Pollitehnica of Bucharest;* Ion Stoica, *University of California, Berkeley*

Power, Energy, and Sustainability

Hardware Lifecycle-Aware Power Planning in Commercial Hyperscale Datacenters (Operational Systems) 1167
Ruihao Li, *Meta and The University of Texas at Austin;* Leonardo Piga, Wei Su, and Carlos Torres, *Meta;* Jovan Stojkovic, *Meta and The University of Texas at Austin;* Neeraja J. Yadwadkar and Lizy K. John, *The University of Texas at Austin;* Abhishek Dhanotia, *Meta*

Kareus: Joint Reduction of Dynamic and Static Energy in Large Model Training 1185
Ruofan Wu, Jae-Won Chung, and Mosharaf Chowdhury, *University of Michigan*

SPADE: Signal-Aware DAG Scheduling and Dynamic Provisioning for Data Processing Clusters 1205
Adam Lechowicz, *University of Massachusetts Amherst;* Rohan Shenoy, *University of California, Berkeley;* Noman Bashir, *Massachusetts Institute of Technology;* Mohammad Hajiesmaili, *University of Massachusetts Amherst;* Adam Wierman, *California Institute of Technology;* Christina Delimitrou, *Massachusetts Institute of Technology*

Quota Marketplace: Dynamic Pricing for Efficient Allocation of ML Training Resources. 1223
Balasubramanian Sivan, Renato Paes Leme, Mihai Tiuca, and Ian McFarlane, *Google;* Vasilis Gkatzelis, *Google and Drexel University;* Nehal Mehta, Soheil Hassas Yeganeh, Vahab Mirrokni, and Amin Vahdat, *Google*

Consensus and Byzantine Fault Tolerance

Bodega: Localized Linearizable Reads at Anywhere Anytime via Roster Leases. 1243
Guanzhou Hu, *Amazon Web Services;* Andrea C. Arpaci-Dusseau and Remzi H. Arpaci-Dusseau, *University of Wisconsin–Madison*

Equal Opportunity: A Correctness Condition for Ordered Consensus 1283
Yunhao Zhang, *Cornell University;* Haobin Ni, *University of Washington;* Soumya Basu, *OpenReserve Holdings;* Shir Cohen, *Cornell University;* Maofan Yin, *University of California, Santa Barbara;* Lorenzo Alvisi and Robbert van Renesse, *Cornell University;* Qi Chen and Lidong Zhou, *Microsoft Research*

Jetpack: Consensus Made Generally Fast **1299**
Ze Tang, Zihao Zhang, and Weihai Shen, *Stony Brook University;* Jicheng Shi, *DatenLord;* Shuai Mu, *Stony Brook University*

Ambulance: saving BFT through racing **1325**
Neil Giridharan and Shubham Mishra, *University of California, Berkeley;* Lorenzo Alvisi, *Cornell University;* Natacha Crooks, *University of California, Berkeley;* Benjamin Marsh, *Sei Labs;* Hein Meling, *University of Stavanger;* Kartik Nayak, *Duke University;* Grzegorz Prusak, *Sei Labs*

Training Reliability and Silent Errors

SDCs in the Wild: Characterizing and Diagnosing SDC-defective GPUs in Production LLM Training (Operational Systems) **1349**
Wenxin Zheng, *Shanghai Jiao Tong University and ByteDance Seed;* Wenxiao Wang, Yun Zhang, and Mingcong Han, *ByteDance Seed;* Bin Xu, Jinyu Gu, Xingda Wei, and Haibo Chen, *Shanghai Jiao Tong University;* Zuquan Song, Gaohong Liu, Yucheng Nie, Zhe Nan, Zhuolin Zheng, Huan Yu, Shuguang Wang, Ziming Zhou, Hang Zhu, Wencong Xiao, and Xin Liu, *ByteDance Seed*

Safeguarding LLM Training at Scale: Online SDC Detection and Insights from 35 Million GPU Hours **1369**
Kinman Lei, *Tsinghua University;* Liyan Zheng, Xiang Li, Hongmin Chen, Yun Zhang, Gaohong Liu, Zuquan Song, and Zixuan Ma, *ByteDance;* Zhiyu Xue, *Tsinghua University;* Minghui Yu, Shuguang Wang, Wencong Xiao, and Haibin Lin, *ByteDance;* Yuyang Jin and Jidong Zhai, *Tsinghua University;* Bo Liu and Xin Liu, *ByteDance*

OpGuard: Bitwise Alignment for Precise and General Debugging of Production LLM Training **1385**
Ziming Zhou and Yinjie Zhao, *University of Michigan;* Hang Zhu, Wenxiao Wang, Zhihao Bai, Yun Zhang, Shuguang Wang, and Haibin Lin, *ByteDance Seed;* Peng Huang, *University of Michigan*

RobustRL: Role-based Fault Tolerance System for RL Post-Training **1407**
Zhenqian Chen and Baoquan Zhong, *Zhejiang University;* Xiang Li, *unaffiliated;* Qing Dai, Xinkui Zhao, and Miao Ye, *Zhejiang University;* Ren Cheng, *unaffiliated;* Lufei Zhang, *State Key Laboratory of Mathematical Engineering and Advanced Computing, China;* Jianwei Yin, *Zhejiang University*

Filesystems and I/O Acceleration

Oxbow: A Coordinated Architecture for Multi-component File Systems **1425**
Jongyul Kim, *University of Illinois Urbana–Champaign;* Jaehwan Lee and Inhoe Koo, *KAIST;* Peizhe Liu and Jiyuan Zhang, *University of Illinois Urbana–Champaign;* Junho Ahn, *KAIST;* Tianyin Xu, *University of Illinois Urbana–Champaign;* Youngjin Kwon, *KAIST*

Scaling the IO wall with Declarative IO **1443**
Sanjith Athlur, *Carnegie Mellon University;* Sara McAllister, *Carnegie Mellon University, University of Wisconsin, Madison, and Google;* Theo Gregersen, Timothy Kim, Yiwei Chen, Sarvesh Tandon, and Lucy Wang, *Carnegie Mellon University;* Daniel S. Berger, *Carnegie Mellon University, Microsoft Azure, and University of Washington;* Saurabh Kadekodi and Arif Merchant, *Google;* Benjamin Berg, *University of North Carolina at Chapel Hill;* Nathan Beckmann, Rashmi Vinayak, George Amvrosiadis, and Gregory R. Ganger, *Carnegie Mellon University*

Umap: Revisiting Memory-mapped I/O on Distributed File Systems for Efficient Matrix Access (Operational Systems) **1461**
Yongchao He, *unaffiliated;* Guangyan Zhang, *Tsinghua University;* Zane Cao, *ScitiX AI;* Wenfei Wu, *Peking University*

CoPilotIO: CPU as a Co-pilot for GPU I/O to Free GPU Compute **1479**
Guanyi Chen and Qi Chen, *The Hong Kong University of Science and Technology* (*Guangzhou*)*;* Shu Yin, *ShanghaiTech University;* Jian Zhang, *The Hong Kong University of Science and Technology* (*Guangzhou*)

Close to the Network

RoCE BALBOA: Service-Enhanced RDMA Offload Engine for Data Center SmartNICs **1495**
Maximilian Jakob Heer, Benjamin Ramhorst, Yu Zhu, Luhao Liu, Zhiyi Hu, Jonas Dann, and Gustavo Alonso, *ETH Zurich*

DPA-Store: An Ordered Network Data Path Key-Value Store **1513**
Frederic Schimmelpfennig, *Johannes Gutenberg-Universität Mainz;* Jan Sass, *Saarland University;* Reza Salkhordeh, *Johannes Gutenberg-Universität Mainz;* Martin Kröning and Stefan Lankes, *RWTH Aachen University;* André Brinkmann, *Saarland University*

FARLock: Asymmetric RDMA Locking Made Fair 1531
Yuehao Hu, Jiatang Zhou, Tianzheng Wang, and Keval Vora, *Simon Fraser University*

When DDIO Meets Page Coloring: Revisiting DDIO Performance with Sepia 1547
Changwoo Song, Sanghyun Kim, and Jinhyeok Oh, *Sungkyunkwan University;* Qizhe Cai, *University of Virginia;* Joonsung Kim and Jaehyun Hwang, *Sungkyunkwan University*

Graphs, ANN, and Vector Search

Disentangling Graph Dependencies for Efficient Billion-Scale GPU Vector Search 1563
Haoru Zhao, Jingkai He, Jingyao Zeng, Mingkai Dong, and Dong Du, *Shanghai Jiao Tong University*

Efficient GPU-Centric Evolving Graph Processing at Scale 1585
Yunmo Zhang and Jiacheng Huang, *City University of Hong Kong;* Xizhe Yin, *Independent Researcher;* Junqiao Qiu, *City University of Hong Kong;* Hong Xu, *The Chinese University of Hong Kong;* Chun Jason Xue, *Mohamed bin Zayed University of Artificial Intelligence*

Pluto: High-Performance, Memory-Efficient Distributed Graph Analytics Through Advanced Mirroring 1605
Ying-Wei Wu, *The University of Texas at Austin;* Christopher J. Rossbach, *The University of Texas at Austin and Microsoft;* Mattan Erez, *The University of Texas at Austin*

The Clustering Strikes Back: Building Cost-Effective and High-Performance ANNS at Scale with Helmsman (Operational Systems) 1623
Yuchen Huang and Baiteng Ma, *East China Normal University and Xiaohongshu Inc;* Yiping Sun, Yang Shi, Xiao Chen, Xiaocheng Zhong, Zhiyong Wang, and Yao Hu, *Xiaohongshu Inc;* Erci Xu, *Shanghai Jiaotong University;* Chuliang Weng, *East China Normal University*

Durable and Trustworthy Storage

WiseCode: Breaking the Scalability Barriers of Wide-Stripe Vector Codes 1641
Sijie Cai, Guangyan Zhang, and Xiao Niu, *Tsinghua University*

The LogDrive: Composable Durability for Cloud-Based Shared Logs 1663
Gardner Vickers, Lucas Bradstreet, Mahesh Balakrishnan, Prince Mahajan, David Mao, Xavier Léauté, Ismael Juma, Nikhil Bhatia, Jack Vanlightly, Prateek Jindal, Sumit Arrawatia, Andrew Grant, Dhruvil Shah, Dimitar Dimitrov, Gaurav Badoni, Shimiao Zhang, and Yang Yu, *Confluent*

Timelock Drive: Isolated Time-Based Defense for Storage Systems 1683
Jonah Rosenblum, Juechu Dong, Peter Chen, and Satish Narayanasamy, *University of Michigan*

High Fidelity Models for Large Scale Stateful Services (Operational Systems) 1699
Nouraldin Jaber, Dongyun Jin, Bernhard Kragl, Enrico Magnago, Gustavo Petri, Thorsten Tarrach, and Serdar Tasiran, *Amazon Web Services*

Virtualization and Live Migration

M3U: Scalable Kernel Memory Management for Efficient Post-copy Live Migration of High-end Virtual Machines 1715
Yizhe Xu, *Shanghai Jiao Tong University;* Yuan Tao, Zhibin Zhang, Kang Yan, and Chao Zhang, *Alibaba Cloud Computing;* Shuo Shi, Zongpu Zhang, and Xu Huan, *Shanghai Jiao Tong University;* Yibin Shen, Xudong Zheng, and Jiesheng Wu, *Alibaba Cloud Computing;* Jian Li and Haibing Guan, *Shanghai Jiao Tong University*

Compaction-Free Memory Defragmentation for Virtualization via Infinite Guest Physical Address Space 1731
Peixin Zeng, Hao Huang, Yanqi Pan, Wen Xia, Darong Yang, Jiahao Chen, and Nan Zhang, *Harbin Institute of Technology, Shenzhen*

Inside Out: A Paradigm Shift In VM Introspection 1749
Dufy Teguia, *University Grenoble Alpes, CNRS, Inria, Grenoble INP, LIG, and Orange Research;* Louis Duval, *University Grenoble Alpes, CNRS, Inria, Grenoble INP, LIG;* Teo Pisenti, *IRIT, Université de Toulouse, CNRS, Toulouse INP, UT3 Toulouse;* Kahina Lazri, *Orange Research;* Daniel Hagimont, *IRIT, Université de Toulouse, CNRS, Toulouse INP, UT3 Toulouse;* Thomas Pasquier, *University of British Columbia;* Renaud Lachaize and Alain Tchana, *University Grenoble Alpes, CNRS, Inria, Grenoble INP, LIG*

vBOIDs: Taming Chaos via Coarse-grained Scheduling Abstraction for Containers 1769
Kaesi Manakkal, *The University of Texas at Arlington;* Nathan Daughety, *Air Force Research Laboratory (AFRL);* Yu Sun, *Binghamton Unversity;* Marcus Pendleton, *Air Force Research Laboratory (AFRL);* Hui Lu, *The University of Texas at Arlington*

Wednesday, July 15

Resource-Efficient LLM Serving

Efficient LLM Serving on Commodity GPU Clusters with Data-Reduced Cross-Instance Orchestration 1787
Jiangsu Du, Hongbin Zhang, Taosheng Wei, Zhenyi Zheng, Jiazhi Jiang, Kaiyi Wu, Zhiguang Chen, and Yutong Lu, *School of Computer Science and Engineering, Sun Yat-Sen University*

Revisiting Pipeline Parallelism for LLM Serving 1803
Soonjae Hwang and Jeongseob Ahn, *Korea University*

OpenTela: Unifying Decentralized Computing Resources for Heterogeneous LLM Serving (Operational Systems). 1821
Xiaozhe Yao, *ETH Zurich;* Youhe Jiang, *University of Cambridge;* Ilia Badanin, *EPFL;* Qinghao Hu, *MIT;* Robert Matthew Smith, *ETH AI Center;* Binhang Yuan, *The Hong Kong University of Science and Technology;* Imanol Schlag, *ETH AI Center;* Eiko Yoneki, *University of Cambridge;* Ana Klimovic, *ETH Zurich*

KAIROX: Adaptive GPU-CPU Hybrid LLM Inference via Online Neuron Balancing 1839
Yapeng Jiang, *Sun Yat-sen University, Peng Cheng Laboratory, and Zhuhai Key Laboratory of Trusted Large Language Models;* Minghao Gan, *Sun Yat-sen University;* Zicong Hong, *École Polytechnique Fédérale de Lausanne;* Wuhui Chen, *Sun Yat-sen University and Peng Cheng Laboratory;* Junyuan Liang, *Sun Yat-sen University;* Yue Yu, *Peng Cheng Laboratory;* Meng Guo, *Qilu University of Technology;* Zibin Zheng, *Sun Yat-sen University*

ADAngel: Accelerating Arbitrary-Precision Quantized LLMs with Adaptive Computing Mapping 1857
Yao Liu, Wenjie Wang, Yifei Feng, Bo Peng, Jianguo Yao, and Haibing Guan, *Shanghai Jiao Tong University*

GPU Compilers and Kernels

Optimal Software Pipelining and Warp Specialization for Tensor Core GPUs. 1875
Rupanshu Soi, Rohan Yadav, Fredrik Kjolstad, and Alex Aiken, *Stanford University;* Maryam Mehri Dehnavi, Michael Garland, and Michael Bauer, *NVIDIA*

***TileLoom*: Automatic Dataflow Planning for Tile-Based Languages on Spatial Dataflow Accelerators 1891**
Wei Li, Zhenyu Bai, Heru Wang, Pranav Dangi, and Zhiqiang Zhang, *National University of Singapore;* Cheng Tan, *Arizona State University and Google;* Huiying Lan, *Lumai Ltd.;* Weng-Fai Wong and Tulika Mitra, *National University of Singapore*

MPK: A Compiler and Runtime for Mega-Kernelizing Tensor Programs 1909
Xinhao Cheng, Zhihao Zhang, Yu Zhou, and Jianan Ji, *Carnegie Mellon University;* Jinchen Jiang, *Tsinghua University;* Zepeng Zhao and Ziruo Xiao, *Carnegie Mellon University;* Zihao Ye and Yingyi Huang, *NVIDIA;* Ruihang Lai, Hongyi Jin, Bohan Hou, Mengdi Wu, Yixin Dong, and Anthony Yip, *Carnegie Mellon University;* Zihao Ye, *University of Michigan;* Songting Wang, *Carnegie Mellon University;* Wenqin Yang, *Independent Researcher;* Xupeng Miao, *Peking University;* Tianqi Chen, *Carnegie Mellon University and NVIDIA;* Zhihao Jia, *Carnegie Mellon University*

GraCE: Unlocking CUDA Graphs with Compiler Support for ML Workloads 1927
Abhishek Ghosh and Ajay Nayak, *Indian Institute of Science;* Ashish Panwar, *Microsoft Research;* Arkaprava Basu, *Indian Institute of Science*

VTC: DNN Compilation with Virtual Tensors for Data Movement Elimination. 1949
Muyan Hu, Ahan Gupta, and Jiachen Yuan, *University of Illinois Urbana–Champaign;* Vima Gupta, *Georgia Institute of Technology;* Taeksang Kim and Xin Xu, *University of Illinois Urbana–Champaign;* Janardhan Kulkarni and Ofer Dekel, *Microsoft;* Vikram Adve and Charith Mendis, *University of Illinois Urbana–Champaign*

Serverless Plus Resilience

Stop Pretending to be Busy: A Case for Serverless Paradigms in Co-located Batch Workloads (Operational Systems) 1967
Xiaohu Chai, *Tsinghua University and Ant Group;* Jianfeng Tan, Congsi Yuan, Bowen Yang, Hao Dai, Tongkai Yang, and Chao Huang, *Ant Group;* Dong Du, *Shanghai Jiao Tong University;* Yu Chen, *Quan Cheng Laboratory and Tsinghua University*

Continuation-Centric Computing with Arca 1989
Akshay Srivatsan, Yuhan Deng, Katherine Mohr, Emma Sudo, Sebastian Ingino, Francis Chua, and Keith Winstein, *Stanford University*

Rethinking Process Snapshots for Near-Warm Serverless Cold Starts 2007
Ben Holmes, Baltasar Dinis, Lana Honcharuk, and Adam Belay, *MIT CSAIL;* Joshua Fried, *University of Pennsylvania*

Distributed Speculative Execution for Resilient Cloud Applications. 2027
Tianyu Li, *MIT CSAIL;* Badrish Chandramouli and Philip A. Bernstein, *Microsoft Research;* Sam Madden, *MIT CSAIL*

TrainMover: An Interruption-Resilient Runtime for ML Training 2047
ChonLam Lao, *Harvard University and Alibaba Group;* Jiaqi Gao, Jiamin Cao, Zhipeng Zhang, Pengcheng Zhang, Jiangfei Duan, Zhilong Zheng, Yu Guan, Yichi Xu, Yong Li, and Zhengping Qian, *Alibaba Group;* Aditya Akella, *The University of Texas at Austin;* Minlan Yu, *Harvard University;* Ennan Zhai, Dennis Cai, and Jingren Zhou, *Alibaba Group*

Accelerator and Device Virtualization

MoonBright: A GPU Memory Allocator with Device-Side Page Table Materialization and Deferred TLB Coherence 2065
Yangyu Zhang, *SKLP, Institute of Computing Technology, Chinese Academy of Sciences and University of Chinese Academy of Sciences;* Lei Chen, *University of Chinese Academy of Sciences;* Chunwei Xia, *University of Leeds;* Shuaijiang Li, Shuoming Zhang, Zhicheng Li, Qianqi Sun, Jiawei Xiao, Ruiyuan Xu, Ao Chen, Guangli Li, Xiaobing Feng, Huimin Cui, Chenxi Wang, and Jiacheng Zhao, *SKLP, Institute of Computing Technology, Chinese Academy of Sciences and University of Chinese Academy of Sciences*

Nixie: Efficient, Transparent Temporal Multiplexing for Consumer GPUs 2085
Yechen Xu, Yifei Wang, Nathanael Ren, Yiran Chen, and Danyang Zhuo, *Duke University*

μShell: A Microkernel-based FPGA Shell Architecture. 2103
Jiyang Chen, Anubhav Panda, and Harshavardhan Unnibhavi, *Technical University of Munich;* Atsushi Koshiba, *Tokyo University of Science;* Pramod Bhatotia, *Technical University of Munich*

Virtualizing eBPF with Late-Binding. 2127
Jing Zhang, *Shanghai Jiao Tong University;* Xiaguannan Song, *Harbin Institute of Technology, Shenzhen;* Dong Du, Yubin Xia, Binyu Zang, and Haibo Chen, *Shanghai Jiao Tong University*

Fleet and Cluster Scheduling

DVLA: Dynamic VM Lifetime Aware Scheduling for Drifting Lifetime Distributions and Long-Lived VM Placement Debt (Operational Systems) 2149
Zhengtong Zhang, Zihan Xu, Zhidong Hu, Yanbo Shan, Fei Peng, Suhong Chen, Kaiyuan Shen, Xiangyun Kong, Handu Ding, Bing He, and Binda Ma, *Alibaba Cloud Computing*

PIMS: Fleet-wide Datacenter Maintenance with Minimal Capacity Buffer and Predictable Latency (Operational Systems). 2169
Benjamin Leonhardi, *Meta Platforms;* Evangelia Kalyvianaki, *University of Cambridge;* Yang Wang, *Meta Platforms and The Ohio State University;* Abdelrahman Adam, Agshin Nabiyev, Aleks Shirokov, Amitav Mohanty, Daniil Balenko, Elaine Zhao, and Essam Ewaisha, *Meta Platforms;* Hongbo Dong, *NexGeMM LLC;* Igor Marnat, Lev Novikov, and Min Zeng, *Meta Platforms;* Steven Shingler, *Independent Researcher;* Timofey Durakov, Wiliam de Abreu Pinho, Ben Christensen, Mayank Pundir, and Kaushik Veeraraghavan, *Meta Platforms*

Heterogeneity at Hyperscale: Characterization and Scheduling of Large Production AI Clusters at Alibaba (Operational Systems) 2187
Suyi Li, *Hong Kong University of Science and Technology;* Lingyun Yang, *Hong Kong University of Science and Technology and Alibaba Group;* Haoxuan Yu, Sheng Yao, Tianyuan Wu, Xiaoxiao Jiang, and Hanfeng Lu, *Hong Kong University of Science and Technology;* Kangjin Wang, *Alibaba Group;* Chenhao Wang, *Fudan University;* Shenglin Xu, Lun Wang, Qingyang Duan, Shenghao Liang, Xiu Lin, Meng Zhang, Wenchao Wu, Yinghao Yu, Guodong Yang, and Liping Zhang, *Alibaba Group;* Wei Wang, *Hong Kong University of Science and Technology*

MIMESYS: Generating Realistic Executable Testing Environments from Resource Usage Traces 2205
Donghyun Kim, Zichao Hu, Joydeep Biswas, Aditya Akella, and Daehyeok Kim, *The University of Texas at Austin*

A Coherent, Consistent Session on Caching

MERLIN: An Efficient Adaptive Cache Eviction Algorithm via Fine-Grained Characterization 2223
Liujia Li, Jinhao Guo, Yi Fan, and Jianyu Wu, *Peking University;* Zhenlin Wang, *Michigan Tech;* Jie Zhang, *Peking University;* Yuval Tamir, *University of California, Los Angeles;* Xiaolin Wang, Yingwei Luo, and Diyu Zhou, *Peking University*

Learning-Augmented Heuristics: Simple yet Smart, Robust and Interpretable Cache Eviction 2241
Haocheng Xia, *Harvard University and University of Illinois Urbana–Champaign;* William Nixon, *University of Chicago and Harvard University;* Bintang Dwi Marthen, *Harvard University and Institut Teknologi Bandung;* Pranav Bhandari, *Meta;* Juncheng Yang, *Harvard University*

WriteGuards: Distributed Storage Support for Strongly Consistent Caches 2261
Ziming Mao, *University of California, Berkeley;* Atul Adya and Jonathan Ellithorpe, *Databricks;* Rishabh Iyer and Matei Zaharia, *University of California, Berkeley;* Scott Shenker, *University of California, Berkeley, and ICSI;* Ion Stoica, *University of California, Berkeley*

MEGALON: Efficient Data Sharing for Partly Coherent CXL Memory 2281
Jiyu Hu, Seokjoo Cho, Landon Johnson, Kiran Hombal, and Shreesha Gopalakrishna Bhat, *University of Illinois Urbana–Champaign;* Marcos K. Aguilera, *NVIDIA;* Ramnatthan Alagappan and Aishwarya Ganesan, *University of Illinois Urbana–Champaign*

Mobile and Edge Systems

Inference in the Shadows: Taming Memory Bandwidth Contention in Mobile LLM Inference with Sereno 2299
Tong Xin, Xinrui Shi, Mingkai Dong, and Zeyu Mi, *Institute of Parallel and Distributed Systems, Shanghai Jiao Tong University*

LifeLine: An Object-Page Lifetime Alignment GC Enabling Minimal Memory Copying for Mobile Devices 2319
Jiacheng Huang and Yunmo Zhang, *City University of Hong Kong;* Qingan Li, *Wuhan University;* Junqiao Qiu, *City University of Hong Kong;* Chun Jason Xue, *Mohamed bin Zayed University of Artificial Intelligence*

Unleash All Cores: Asymmetry-aware Scalable DNN Inference on Mobile CPUs 2337
Qianlong Sang, Puyi He, Huanghuang Liang, and Yili Gong, *Wuhan University;* Chuang Hu and Xiaobo Zhou, *University of Macau;* Dazhao Cheng, *Wuhan University*

Surviving the Impossible Trinity: Revisiting CPU Scheduling Problem on Modern COTS Mobile Devices (Operational Systems) 2353
Jun Xiao, Qinhui Gu, Ligeng Chen, Lizhi Sun, Zicheng Wang, Yinggang Guo, and Lu Liu, *Honor Device Co., Ltd.;* Hao Wu, *Nanjing University;* Borui Li, *Southeast University*

Wild Cards

QTPU: Hybrid Tensor Networks for Quantum-Classical Acceleration 2367
Nathaniel Tornow, Emmanouil Giortamis, Dennis Sprokholt, Christian Mendl, and Pramod Bhatotia, *Technical University of Munich*

Acumen: A Platform for Encrypted and Accountable Collaborative Editing 2389
Ryan Cottone, *Stanford University;* Darya Kaviani, Conor Power, Will Giorza, Evelyn Koo, Natacha Crooks, and Raluca Popa, *University of California, Berkeley*

Drs.NAS: Ultra-Efficient Neural Architecture Search for Recommendation Systems 2407
Ruixuan Wang and Xun Jiao, *Villanova University*

SMARTTalk: Teaching SMART Logs to Talk to LLMs 2423
Mayur Akewar and Dongsheng Luo, *Florida International University;* Sandeep Madireddy, *Argonne National Laboratory;* Janki Bhimani, *Florida International University*

Tooling Potpourri

Svalinn: Overload Control in Large-Scale Servers with Multiple Resource Bottlenecks 2443
Bhaskar Subhash Pardeshi, Peidi Song, and Ahmed Saeed, *Georgia Institute of Technology*

PeeR: First-Class Scheduling for Latency-Critical eBPF Applications 2465
Jeremy Carin, Ben Holmes, and Weiyang Wang, *MIT CSAIL;* Ankit Bhardwaj, *Tufts University;* Manya Ghobadi, *MIT CSAIL and Systalyze*

TypeCraft: A Lightweight Data Type Profiler with High Resolution 2483
Zecheng Li, *North Carolina State University;* Xu Liu, Namhyung Kim, Blake Jones, and Alexey Alexandrov, *Google;* Jiajia Li, *North Carolina State University*

All Along the Watchtower: Achieving the Trinity of Observability in Cloud with DiTing 2501
Zhenyu Ren and Shuzhi Feng, *Alibaba Group;* Erci Xu, *Shanghai Jiaotong University;* Changsheng Niu, Haoyu Mao, Beibei Wang, Chong Gao, Zhenshan Zhang, Xinrui Yu, Jiangwei Huang, Jiesheng Wu, and Hong Tang, *Alibaba Group*

The Session About Correctness

jwmalloc: A Verified Memory Allocator for Mobile Devices 2515
Jiawei Wang, Ming Fu, Ruixian Wang, and Chao Xu, *Huawei Central Software Institute;* Jonas Oberhauser, *Huawei Central Software Institute and Huawei Hilbert Research Center;* Haibo Chen, *Huawei Central Software Institute and Shanghai Jiao Tong University*

Neuro-Symbolic Proof Generation for Scaling Systems Software Verification 2533
Baoding He, *Nanjing University;* Zenan Li, *ETH Zurich;* Wei Sun and Yuan Yao, *Nanjing University;* Taolue Chen, *Birkbeck, University of London;* Xiaoxing Ma, *Nanjing University;* Zhendong Su, *ETH Zurich*

Spain: Succinct proofs for numerical computations 2551
Zachary DeStefano, Noah Golub, Zile Huang, Julius Zhang, Sam Frank, and Michael Walfish, *NYU*

RT: Regular Types for the Streaming Shell 2583
Zekai Li, Lukas Lazarek, Evangelos Lamprou, and George Kapetanakis, *Brown University;* Konstantinos Mamouras, *Rice University;* Nikos Vasilakis, *Brown University*

Strata: Hierarchical Context Caching for Long Context Language Model Serving

Zhiqiang Xie
Stanford University & NVIDIA

Ziyi Xu
Shanghai Jiao Tong University

Mark Zhao
University of Colorado Boulder

Yuwei An
Carnegie Mellon University

Vikram Sharma Mailthody
NVIDIA

Scott Mahlke
NVIDIA & University of Michigan

Michael Garland
NVIDIA

Christos Kozyrakis
NVIDIA & Stanford University

Abstract

Long-context large language models (LLMs) enable applications that reason over hundreds of thousands to millions of tokens, but serving these workloads efficiently is challenging. Modern systems cache key-value (KV) states and rely on hierarchical context caching across GPU HBM, CPU memory, and SSDs. We show that naïve designs often become I/O-bound: fragmented KV layouts lead to small transfers that underutilize bandwidth, cache loading stalls prefill, and schedulers that ignore cache-loading latency and delay hits (concurrent requests for the same context during a cache miss) suffer severe throughput degradation.

We present Strata, a hierarchical context caching framework for long-context LLM serving. Strata introduces a GPU-assisted I/O mechanism that decouples GPU and host layouts to enable efficient large transfers, and a cache-aware scheduler that mitigates delay hits, balances batches to hide cache-loading latency, and opportunistically overlaps complementary work. Implemented as part of SGLang and deployed in production, Strata improves throughput by up to 5× over vLLM-LMCache and 3.75× over NVIDIA TensorRT-LLM, without hurting short-context performance.

1 Introduction

Large Language Models (LLMs) represent a significant advancement in machine learning, achieving remarkable proficiency in understanding and generating natural language. Their adoption is now widespread, and they are rapidly evolving towards more robust problem-solving capabilities. A prominent trend in this evolution is the expansion of context windows, allowing LLMs to parse longer input prompts. This has enabled an important set of applications that require understanding large amounts of text, including coding assistants [6], retrieval-augmented generation (RAG) [27], document analysis [49], and conversational AI agents with memory modules [40]. Leading models, including Google's Gemini series [18], Anthropic's Claude 4 [7] and the Qwen series [46], already support context windows of up to one million tokens, with expectations of two million tokens emerging soon. Other frontier models like DeepSeek-V3 [12], Llama-3.1 [3] and Llama-4 series [4] also offer substantial context lengths, typically in the range of 128K to 200K tokens.

While long contexts enable new capabilities, recomputing attention over them from scratch is prohibitively expensive. Caching previously computed key-value (KV) states offers a practical solution, as these prefixes and sources are frequently reused across applications. This technique, often referred to as context or prefix caching [5, 11, 17, 41, 53], avoids redundant prefill computation and significantly reduces response latency. However, the memory footprint of cached KV states is substantial. For example, 40 GB of GPU high-bandwidth memory (HBM) can hold only roughly 0.3M tokens for Llama-8B, which can be quickly consumed by a handful of documents or a few hundred conversation turns. The capacity constraint of GPU memory leads to frequent cache eviction, resulting in low cache hit rates and expensive recomputation. As a result, production systems adopt hierarchical caching, storing KV states in CPU memory [52], local SSDs [14], or even remote memory pools [19,43] to extend capacity and preserve context cache reuse benefits.

However, transferring large cached contexts back to the GPU introduces a major performance bottleneck. Bulk KV transfers often cause memory stall, directly inflating Time To First Token (TTFT) and degrading throughput. Figure 1 illustrates this effect: when serving the LooGLE dataset [28] with SGLang offloading KV caches to CPU memory, configured in line with standard practices reported in prior work [14], 74% of prefill time is blocked on KV transfers (the red curve), resulting in up to a 4× throughput reduction. In these cases, I/O delays rather than compute become the dominant limiting factor. We observe that this inefficiency arises from two main sources.

First, KV cache layout leads to I/O inefficiency. As context lengths grow, the volume of KV data that needs to be moved between memory tiers (e.g., CPU memory to GPU HBM) increases substantially, yet current systems achieve only a

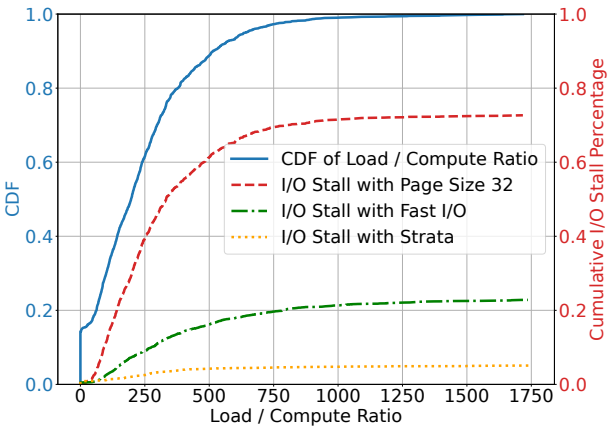


Figure 1: Benchmark profile for Qwen2.5-14B on the LooGLE dataset. The x-axis shows the Load / Compute Ratio (tokens loaded from CPU memory relative to new input tokens) per prefill batch. The right axis displays the I/O stall percentage, representing the amount of prefill execution time attributed to I/O stall. See §5.3 for full benchmark details.

fraction of the available interconnect bandwidth. As we detail in §3.1, this is because current systems adopt PagedAttention [26] to reduce GPU memory fragmentation. Paging, however, fragments KV cache across multiple non-contiguous pages. This yields many small data transfers, sometimes only a few kilobytes, that fail to saturate PCIe bandwidth. Using larger pages could mitigate fragmentation and improve I/O efficiency, but it worsens cache granularity and can reduce cache hit rate, as we demonstrate in §3.1 and §5.3.2.

Second, current schedulers are blind to cache-loading costs and delay hits. Existing systems [26, 36, 53] implicitly assume that the computation needed to prefill new tokens is sufficient to hide the latency of loading historical KV cache from slower memory tiers. However, as context lengths grow, this assumption no longer holds: cache loading time can exceed the compute time needed for prefill, leaving the system loading-bound rather than compute-bound. Figure 1 also highlights this effect. Even with our optimized I/O mechanism presented in §4.2, which removes the overhead of small page transfers (green curve), up to 24% of prefill execution time remains stalled on cache loading. Schedulers that ignore these I/O-bound characteristics generate imbalanced batches, unable to effectively hide cache-loading delays. Moreover, when serving emerging agentic workloads that query same context concurrently, we observe the *delay hit phenomenon* [8], where multiple requests for the same long context arrive while an initial cache miss is still being resolved. This reduces the effective cache hit rate, especially in high-throughput systems, and causes redundant computation of long contexts as we further detail in §3.2 and §5.3.4.

To address these bottlenecks, we propose Strata, a hierarchical context caching framework designed for long context language model serving, without performance degradation in short-context scenarios. Strata employs GPU-assisted data transfer to combat KV cache fragmentation and decouples the GPU's memory layout from that of other memory tiers, improving I/O efficiency across the hierarchy. Strata further reduces long-context overheads through cache-aware request scheduling. It mitigates delay hit, constructs balanced batches that pair sufficient prefill computation to cover I/O latency, and when cache loading stalls are unavoidable, the scheduler inserts useful complementary tasks (e.g., decoding batches) to fully utilize available compute resources. Together, these techniques ensure that scheduling remains efficient even under highly variable latency budgets.

Strata has been integrated into SGLang [53], a widely adopted open-source framework for LLM serving, and has been deployed in production environments at several leading AI companies. Our evaluation on popular long-context benchmarks, across a range of models and hardware platforms, shows that Strata improves throughput by up to 5× over vLLM-LMCache [30], a state-of-the-art open-source hierarchical context caching solution, and by up to 3.75× over NVIDIA's TensorRT-LLM, a highly optimized serving engine, without performance degradation on short-context scenarios.

2 Background

2.1 Long Context LLM Inference

LLM inference operates in two phases: *prefill* and *decode*. During prefill, the model typically processes both (i) new tokens from the user query and (ii) context tokens, drawn from sources such as documents or prior interactions. The intermediate outputs of this step, known as KV caches, are critical for efficiency because they allow subsequent tokens to attend to past tokens without recomputing all previous layers. In the decode phase, the model generates tokens autoregressively, continually reusing and extending the KV cache. Long contexts amplify the cost of both phases. A longer prefix increases prefill latency, and if cached states are evicted or unavailable, recomputing that prefix becomes increasingly expensive. During decode, each generated token reuses and extends the KV cache, whose size grows linearly with sequence length. As a result, for long-context workloads, KV cache memory footprint and data movement can become dominant bottlenecks, making efficient cache reuse, placement, and eviction critical for high-throughput, low-latency serving.

2.2 Memory Management of KV Cache

Inspired by virtual memory, PagedAttention [26] avoids reserving large contiguous blocks for KV caches by using dynamic, page-based allocation. The cache is partitioned into

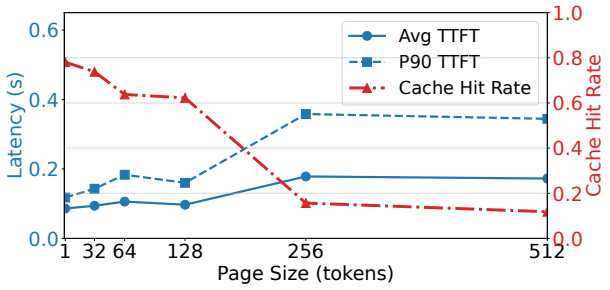


Figure 2: Large page sizes decrease cache hit rate and increase TTFT, benchmarked on H200 for Mistral-24B using the ShareGPT dataset.

small fixed-size pages that preserve logical sequence order but can be placed non-contiguously in memory, improving utilization. Typical page sizes are small — e.g., 32, 16, and 1 tokens in TensorRT-LLM, vLLM, and SGLang — and the KV states of a single token may span from tens of kilobytes to several megabytes. While such fine granularity is manageable for compute kernels, it poses efficiency challenges for data movement across memory tiers, as we will discuss in §3.1.

2.3 Context Caching in LLM Serving

Beyond intra-request reuse, systems exploit *context caching* across requests by identifying common prefixes using structures like prefix trees or hash maps [26, 43], widely adopted by providers such as OpenAI [41] and Google [17]. To extend capacity, caches are stored in slower tiers such as CPU memory [16, 22, 52], distributed memory pools [19, 29, 43], or even disk [14, 21, 30]. Recent systems, e.g., CachedAttention [14], overlap cache loading with computation on a layer-by-layer basis to minimize stalls, while asynchronously backing up newly generated caches to lower tiers.

3 Challenges of Long Context Caching

This paper addresses the challenge of managing large context caches for long-context (i.e., prefill-dominated) workloads. While this is not the only LLM scenario (i.e., short context, long generation, or single-turn workloads exist), long-context workloads constitute an important class of real-world deployments [6, 25, 28]. We next explore the systems challenges that arise in long-context workloads.

3.1 Low Bandwidth Utilization in KV Cache Transfers

The achievable throughput of an I/O subsystem is fundamentally constrained by the relationship described by Little's Law. Let λ be the arrival rate of I/O operations, C be the average number of concurrent I/O operations, and L be the average latency per operation. According to Little's Law, we have $C = \lambda \cdot L$. Let X represent the sustained data throughput (e.g., in GB/s), S be the average data size per I/O operation, the throughput can be described as $X = \lambda \cdot S$. Combining with Little's law we have $X = C \cdot S/L$ describing attainable throughput in a stable state. This equation highlights that maximizing data throughput (X) requires either high concurrency (C), large transfer sizes (S), or low latency (L).

To transfer data between CPU and GPU memory, asynchronous operations like cudaMemcpyAsync are typically used to engage the GPU's Direct Memory Access (DMA) engine [33]. However, the latency (L) in this scenario includes non-negligible CPU-GPU communication overhead and scheduling delays [20]. While concurrency (C) can be increased by launching numerous asynchronous I/O operations, practical limits arise from the available application-level parallelism (e.g., on the CPU) and the queue capacities within the GPU driver (e.g., CUDA driver) and hardware [33, 44]. Consequently, increasing the transfer size (S) often becomes the most practical lever for improving bandwidth utilization. Saturating modern high-bandwidth interconnects underscores this point; for example, achieving 75-80% of theoretical PCIe 5.0 bandwidth necessitates transfer sizes (S) in the megabyte range (i.e., 1-2MB). This principle is not limited to CPU-GPU transfers; achieving high throughput on other media like SSDs or network interfaces often demands even larger transfer sizes [9, 13, 32].

Cost of Large Pages. However, contrary to requirement of transfer efficiency, LLM inference systems favor small granularity. Smaller pages, e.g., in the range of 1-32 tokens, generally lead to better memory utilization [26] and can improve cache hit rate [34], as cache matching is performed on a per-page basis. While increasing the page size (i.e., compacting more tokens into a single continuous page and effectively increasing S) may appear to improve bandwidth utilization based on the formula, it introduces substantial performance penalties for LLM inference. To demonstrate this trade-off, we benchmarked a Mistral-24B model using the popular ShareGPT dataset, varying the KV cache page size from 1 to 512 on the SGLang framework. As shown in Figure 2, increasing the page size leads to a significant drop in the KV cache hit rate. This degradation directly results in a substantial increase in both the average and P90 TTFT, which rise by up to 2× and 2.9×, respectively, at the largest page sizes tested. We also observed similar trends in long context benchmarking scenarios as shown in Figure 10.

This preference for small pages results in very small effective transfer sizes (S) for KV cache operations, leading to severely underutilized I/O bandwidth. As illustrated in Figure 3, transferring KV cache data for 8192 tokens achieves only approximately 22% of the theoretical PCIe 5.0 bandwidth. Page size is set to 32, a value recommended in prior works on hierarchical KV cache [14, 19, 21, 52] and a maximum supported size in vLLM for CUDA GPUs [48]. This

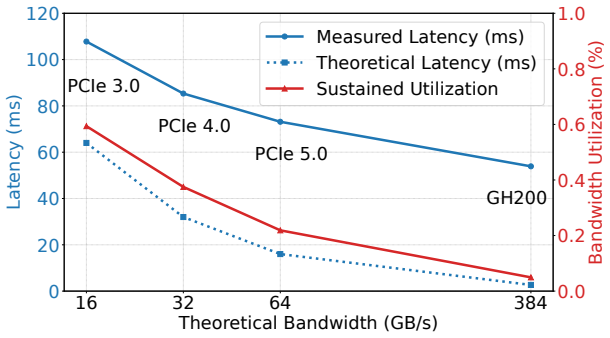


Figure 3: Latency and bandwidth utilization of loading KV caches of 8192 tokens (using page size 32) of Llama-3.1-8B from CPU to GPU on different platforms.

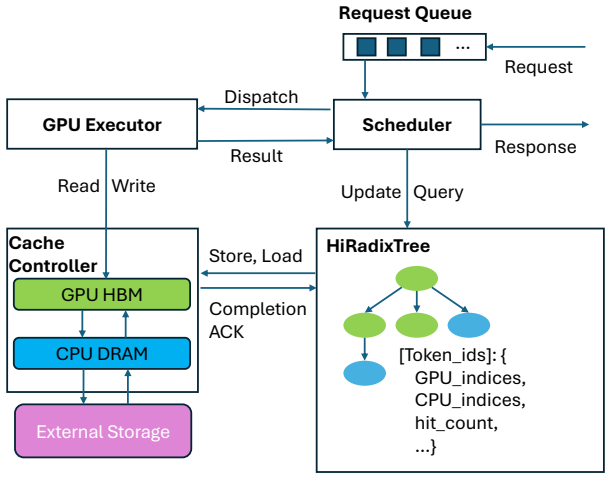


Figure 4: System Architecture of Strata

underutilization is exacerbated on platforms with even higher interconnect bandwidths, falling to as low as 5% on systems like NVIDIA's Grace-Hopper platform that replaces PCIe with NVLink and offers 6× higher peak bandwidth. This fundamental trade-off between transfer efficiency and caching benefit underscores the need for a more effective I/O mechanism that can achieve the best of both worlds.

3.2 Resource Orchestration and Delay Hits

LLM serving engines often treat GPU compute and HBM as first-class resources, since sub-tasks are usually either compute-bound (e.g., dense and attention computation in prefill) or memory-bound (e.g., decoding). With the introduction of hierarchical KV caching, this picture largely remains unchanged. Existing works either regard KV cache loading from CPU to GPU as negligible [14, 52], assuming the latency of loading cache data of layer $N+1$ via PCIe can be effectively hidden by overlapping I/O with the computation of layer N; or else opt for recomputation when the cost is justified [23]. However, when serving requests with long cached contexts but relatively few new tokens to prefill, this assumption breaks down. On one hand, the latency of bulk KV cache transfers can exceed what layer-level computation can hide; on the other hand, recomputation becomes increasingly costly as context length grows, making it an unattractive alternative. Consequently, layer-wise overlapped prefill can degrade into a PCIe bandwidth-bound task. Figure 1 illustrates this bottleneck: even with an I/O mechanism achieving 75% of theoretical PCIe bandwidth, stalls still account for up to 24% of prefill execution time.

Compounding this challenge is the *delay hit phenomenon* [8], previously studied in the networking community. It occurs when multiple requests for the same data object arrive while an initial cache miss is still being resolved, so subsequent requests are effectively treated as cache misses, even though they would have been cache hits had the first request completed. We observe an analogous effect in LLM serving: under high traffic, multiple requests that target the same context (or a prefix of it) may arrive within a short time window, which is especially common in emerging agentic workloads where concurrent attempts often reuse the same context. For example, in the Mooncake agent tool trace [43], 38% of requests share a prefix of at least 6k tokens (excluding the system prompt) with at least one other request arriving within a one second window, making delay hits highly likely in high-throughput serving systems. As illustrated in Figure 7, when such requests are scheduled to execute concurrently (either within the same batch or across data parallel instances), redundant prefill computation occurs. This problem is exacerbated by the asynchronous schedulers widely adopted in modern serving engines [47, 55], which prepare the next batch before the ongoing one completes, extending the cache-miss resolution window to the full execution time of a batch and making delay hits more likely. The impact is especially severe in long-context scenarios, where both recomputation cost and cache-miss resolution time scale unfavorably.

These observations highlight the need to re-think scheduling policies in LLM serving. Schedulers must explicitly treat CPU-GPU bandwidth as a first-class resource, balancing computation and data transfer when batching requests, more carefully overlapping I/O with compute, and mitigating delay hits to avoid redundant work while sustaining throughput.

4 Strata's Design and Implementation

4.1 Overview

Motivated by challenges discussed in §3, we built Strata, a system with two key components. The Strata *Cache Controller* (§4.2) manages the data plane elements throughout the memory hierarchy. It introduces an optimized GPU-CPU data

transfer mechanism and manages KV cache memory layouts to support efficient small page transfers as motivated in §3.1. The Strata *Scheduler* (§4.3) implements the control plane that intelligently schedules requests in a cache resource-aware manner as motivated in §3.2. It references a *HiRadixTree*, which is an extension to SGLang's RadixTree [53], effectively serving as a page table and stores metadata about each KV cache page.

Figure 4 presents the Strata architecture. When a request is submitted, it enters a request waiting queue. During the execution of the ongoing batch, the *Scheduler* continuously estimates available system resources and the resource demands of queued requests, and selects a subset to form the next batch. The *Scheduler* then sends this batch to GPU executor and initiates a KV cache loading request to the *Cache Controller*. During the execution of the prefill batch, the GPU executor synchronizes with the *Cache Controller* to ensure that the KV cache of a certain layer is available before the execution using CUDA events. Once prefill is complete, the prefilled requests are merged into a consolidated decoding batch via continuous batching [51]. Strata uses a prefill-decode (P-D) co-location design, alternating the execution of prefill and decoding batches temporally on the same GPU, and follows SGLang's practice to prioritize the execution of the prefill batch for shorter response time (TTFT) and to form a larger decoding batch for higher throughput. Finally, the *Cache Controller* actively manages the backup and eviction of any KV cache pages to lower memory hierarchies asynchronously.

4.2 Efficient KV Cache I/O

To address the limitations discussed in §3.1, inspired by established practices within the computer architecture community [33,44], Strata leverages *GPU-assisted I/O* to transfer KV cache pages between CPU and GPU memory for low-latency I/O on small, fragmented data. Specifically, instead of invoking standard *cudaMemcpyAsync* API repetitively with small data transfers, a GPU-assisted I/O job operates by launching a CUDA kernel. This kernel spawns thousands of threads. Each thread is responsible for loading a small chunk of data from a source (either GPU global memory or CPU registered pinned memory) into its local register files and then streaming this data to a destination (GPU global memory or registered CPU pinned memory).

GPU-assisted I/O offers several advantages: First, it enables **enhanced concurrency** (*C*): GPUs provide massive, cost-effective parallelism, supporting thousands of concurrent I/O operations compared to typically only tens on CPUs. Second, it is **compatible with small transfers** (efficient *S*): the granularity required for efficient GPU-assisted I/O is only 128 bytes on most architectures [39], which is sufficiently fine for single-page KV caches (kilobytes), eliminating the need to inflate page size for efficiency. Finally, it allows **flexible memory layout**: since light computation in I/O kernels is virtually free, layout transformations between GPU and CPU memory can be performed at negligible cost, enabling flexible and efficient data organization (see §4.2.1 for details).

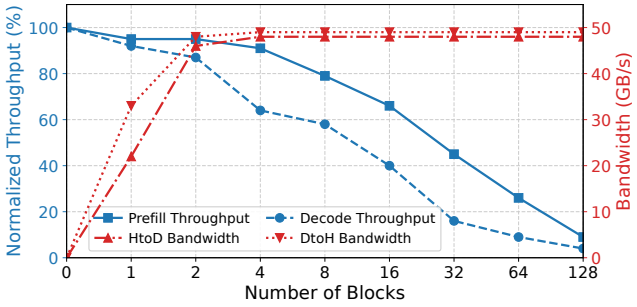


Figure 5: Performance interference vs. resources allocated to the KV-cache I/O kernel. Measurement on concurrently running Strata's I/O kernel with a prefill pass (batch of two requests with 4k input each) and a decode pass (Llama-3.1-8B, batch of 16 requests with 4k input each), respectively.

However, a challenge associated with GPU-assisted I/O, as highlighted in prior work [20], is runtime interference when co-running with other kernels. Without dedicated hardware handling the fine-grained I/O tasks, GPU threads consume valuable resources, such as register files and execution cycles, and can lead to cache pollution. Prior work [31] also demonstrated that GPU hardware schedulers often struggle to effectively manage this resource contention, potentially degrading the performance of both the I/O operations and concurrent computational kernels.

We observe that efficient data transfer does not need to monopolize the entire GPU. Strata employs a strategy of launching a small number of large CUDA blocks to incentivize the GPU's hardware scheduler to confine these I/O kernels to a small subset of Streaming Multiprocessors (SMs), as few as 1. This targeted allocation, when combined with low-level instructions to bypass the cache and thereby mitigate pollution, minimizes interference with concurrent workloads. Moreover, with the ROCm backend [2], these kernel implementations are also compatible with AMD GPUs. To balance resources for overall efficiency, we conducted microbenchmarks co-running I/O kernels with prefill and decoding kernels on an NVIDIA H200 GPU. As shown in Figure 5, using only two CUDA blocks of 1024 threads each, Strata achieves 48 GB/s transfer throughput while incurring less than 5% performance degradation on prefill and 10% on decoding. Based on these results, we select two blocks as the default quota for loading data from CPU to GPU (a critical path operation), and one block for backing up data from GPU to CPU (a non-critical path), where the bandwidth is already sufficient and overhead must be minimized. Our end-to-end evaluation confirms that this configuration sustains high I/O bandwidth while keep-

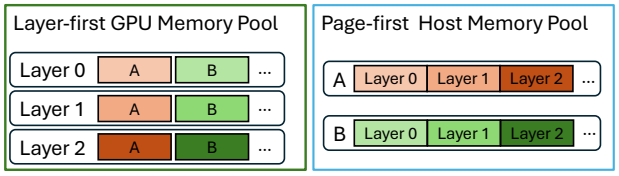


Figure 6: Layer-first vs. Page-first layouts

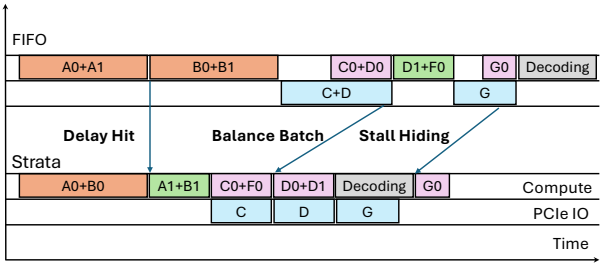


Figure 7: Scheduling Policies in Strata. Orange blocks denote prefill batches that incur a cache miss; green blocks denote cache hits in device memory; purple blocks denote cache hit in host memory; and blue blocks denote data transfer. The one decoding batch is shown in gray. X_n represents the n-th request that uses the context X.

ing overall performance impact under 5%, demonstrating that carefully tuned GPU-assisted I/O can be both efficient and non-intrusive.

4.2.1 Data Management Beyond Host Memory

When external storage is involved, the cache controller opportunistically prefetches data from storage into host memory whenever a cache hit is detected at the storage layer. The prefetch latency is overlapped with the request's queuing delay. Once the scheduler selects the request for execution, the cache controller terminates any in-flight prefetch and uses whatever cache is already available in host or GPU memory. This default *best-effort* policy is motivated by the significantly higher and less predictable latency of the storage layer relative to host-GPU transfers, for which Strata employs a synchronized, layer-wise overlapping scheme. The Strata scheduler can also be configured to hold requests undergoing prefetching until prefetch completion (a *wait-complete* policy) or until a specified timeout (a *timeout* policy), depending on the deployment scenario.

Furthermore, the data transfer inefficiency caused by fragmented memory layout, as motivated in §3.1, also extends to other storage media. In addition to small pages, LLM serving systems also favor a layer-first memory layout in the GPU memory pool (shown in Figure 6), as it aligns with the layer-wise nature of LLM computation. Concretely, the layer-first layout stores all tokens of layer 0 contiguously, then all tokens of layer 1, and so on, so that a layer's attention kernel reads its K/V as one contiguous span. This is ideal for compute but pessimal for transfer: a single logical page (one token's K/V across all layers) is scattered into L non-contiguous fragments, one per layer, each only kilobytes. A transfer-friendly page-first layout instead stores all layers of a page contiguously, yielding large contiguous blocks that saturate the interconnect, but it would force every attention kernel through an extra layer of address indirection. By leveraging GPU-assisted I/O, Strata resolves this conflict by enabling a virtually free, on-the-fly transformation between the compute-friendly and transfer-friendly layouts. To perform the layout transformation, a thread simply applies one additional arithmetic operation to its assigned offset to calculate the correct destination address. This operation has negligible overhead.

As illustrated in Figure 6, this capability decouples the layout requirements across the memory hierarchy: the GPU can maintain its computation-friendly layer-first layout, while other media, such as host memory and external storage, can adopt a page-first layout that maximizes transfer efficiency with larger, contiguous data blocks. In §5.3.5, we demonstrate how this decoupled layout strategy significantly reduces data loading time.

4.3 Cache-Aware Scheduling

As motivated in §3.2, the goal of the scheduler is to maximize caching benefit by avoiding delay hit and loading stalls. The *Scheduler* does so through three stages. First, it identifies requests that are potentially susceptible to delay hits and **defers the execution** to right after the delay hit is resolved. Second, it **formulates a balanced batch** that aims to pair loading (from host memory) with sufficient computation to hide the loading latency. Finally, in the event that batches are still loading-bound, the *Scheduler* **hides I/O stalls** by inserting useful compute inside bubbles. Note while the core scheduler of SGLang is implemented in Python, we augmented it with C++ implementation to minimize the overhead of these scheduling policies.

4.3.1 Deferral on Delay Hit

As discussed in §3.2, delay hits can cause redundant computation in two scenarios: (i) when multiple requests sharing the same cache miss are scheduled into the same batch (Figure 7), and (ii) when the execution of a request is prepared asynchronously without awareness that the corresponding context cache is still being computed. To keep track of potential delay hit, we introduce *transient nodes* in the HiRadixTree. Similar to standard nodes, they use token IDs as traversal keys, but

Algorithm 1 Balanced Batch Formation

```
1:  procedure AddBundleHit(Q, B)
2:      for each r in Q do
3:          if B.is_bundle_hit(r) then
4:              B.add(r); Q ← Q − r
5:  function BatchFormation(Q)
6:      B ← Batch(); D ← []
7:      B.add(Q.pop(0)); AddBundleHit(Q, B)
8:      while |Q| > 0 and ¬B.is_full() do
9:          r ← Q.pop(0)
10:         if B.loading_bound(r) then
11:             D.append(r)
12:         else
13:             B.add(r); AddBundleHit(Q, B)
14:     for each r in D do
15:         if B.is_full() then break
16:         B.add(r)
17:     return B
```

instead of pointing to memory indices, they carry one of two marks: `in-queue`, indicating that a request is referencing a new context, and `in-flight`, indicating that the cache for the corresponding tokens is under computation. When iterating over the request queue, Strata inserts transient nodes marked `in-queue` as needed. If a request matches existing transient nodes, it is deferred to the next scheduling round but placed at the front of the waiting queue to benefit from the soon-to-be-hot cache and minimize its impact on TTFT.

When a request proceeds to execution, its associated transient nodes are marked `in-flight`. Upon completion, the nodes are converted into standard nodes, with indices pointing to the ready context cached in memory. To prevent unnecessary deferrals, Strata uses a configurable threshold: a request is deferred only when the number of token matches on transient nodes exceeds this value. We use a default threshold of 100 matched tokens, which avoids deferring short incidental prefix matches while still capturing long-context delay hits.

4.3.2 Balanced Batch Formation

After removing candidates susceptible to delay hits, the scheduler selects requests to form the next prefill batch. In most LLM serving engines [26, 53], batch formation follows a first in, first out (FIFO) policy by default, where requests are taken in arrival order until the batch is full (either reaching a preset token limit or exhausting GPU memory). To address the loading-bound issue discussed in §3.2, Strata introduces a new batch formation mechanism that balances data loading with sufficient computation. An example is illustrated in Figure 7, a batch containing requests C0 and D0 would require loading both contexts, causing a loading stall. In contrast, forming a different batch (C, F) could get the loading of C overlapped. Another useful case is the batch (D0, D1): since they share the same context, batching them not only balances loading but also reduces GPU memory usage and on-device bandwidth pressure, further improving efficiency. We refer to this case as a *bundle hit*, the counterpart of a *delay hit*.

The procedure is detailed in Algorithm 1. Before each batch is formed, the scheduler obtains the *load* and *compute* requirements of each request using the HiRadixTree. During batch formation, as it iterates through the queue, the scheduler checks whether adding a request would reach the loading-bound limit ('loading_bound' in line 10), defined as the ratio of aggregated load to compute. When this ratio exceeds a threshold, the batch is considered loading-bound. This threshold is hardware- and model-dependent and thus can be profiled separately; in practice, Strata uses a default ratio of 100, corresponding to the point where stalls begin to appear as shown in Figure 1. If the request fits into the batch without making it loading-bound, it will be added into the batch, then the scheduler will iterate through remaining requests to preferentially add requests that bundle-hit with it. Otherwise, the request is moved to a deprioritized list. If the batch is not full till the end of the queue, the scheduler supplements it with deprioritized requests (line 16). To prevent starvation, deprioritized requests retain their original order, and each batch formation always begins with the first request in the queue.

4.3.3 Hide Loading Stalls with Bubble Filling

Even with balanced batching, some batches can still be loading-bound. The final strategy of the scheduler is *bubble filling* that overlaps loading stalls with useful computation. An example is illustrated in Figure 7, when request G0 requires a long context load, the scheduler defers computation of the prepared prefill batch and instead issues a decoding batch to the model executor to run concurrently with the context loading. This strategy complements SGLang's default prefill-first policy (as discussed in §4.1), allowing some flexibilities when choosing between prefill and decoding for improved overall utilization. Although decoding batches are also I/O-bound, they primarily saturate HBM bandwidth, whereas loading tasks saturate PCIe bandwidth. This distinction enables the two operations to overlap with minimal resource contention. It is also possible to insert a prefill batch to fill the bubble, which is more applicable to P-D disaggregated systems [54].

4.4 Cache Controller and Policy

As illustrated in Figure 4, the cache controller oversees data movement between the various memory hierarchies. When a new batch is prepared for execution, the scheduler initiates the loading of the KV cache to the GPU via HiRadixTree. Subsequently, during execution, the GPU executor consults the cache controller to ensure that the KV cache data for spe-

cific layers is available before proceeding to the execution of the subsequent layer. Upon completion of prefill or decoding requests, the scheduler initiates the storage of the KV cache to slower memory tiers, backing up specific KV cache data according to the designated cache write policy.

A single write-once policy is insufficient for hierarchical context caching because different serving workloads impose different trade-offs among write bandwidth, cache capacity, durability, and future reuse. Writing every generated KV page immediately maximizes reuse and persistence, but it can waste host memory and write bandwidth for one-off requests. Writing only at eviction minimizes write traffic, but it can add blocking work to the critical path when GPU memory is under pressure. Strata therefore exposes three write policies that correspond to three deployment regimes. The first, *write-back*, only backs up KV cache tokens when they are on the verge of eviction. This approach is suitable for resource-constrained environments as it minimizes bandwidth and capacity usage, though it can introduce additional runtime blocking. The second policy, *write-through*, initiates a backup each time new KV caches are generated and is well-suited for conversational scenarios where all input and generated content must be persistently stored. The third and default policy is *selective-write-through*. With this approach, a counter is associated with each node in the HiRadixTree and increments with each access to that node. A backup is triggered only if this counter exceeds a configurable threshold. This policy offers the most flexibility: setting the threshold to one makes it equivalent to the write-through policy, while higher values can reduce the memory footprint, particularly for scenarios like single-round question-answering tasks. The default threshold is set to 2 when write bandwidth is abundant, but it can be increased to accommodate tighter resource constraints. For all memory layers, the Least Recently Used (LRU) algorithm serves as the default eviction policy.

5 Evaluation

5.1 Methodology

Testbed. We evaluate Strata and baselines on three platforms. The *H200* platform is a node equipped with 8 NVIDIA H200 GPUs interconnected with NVLink, an Intel Sapphire Rapids CPU, and 1.6TB of DRAM. Each GPU is connected to the CPU via a PCIe 5.0 x16 link, offering up to 64 GB/s of peak bandwidth (unidirectional). The *H20-storage* is a node with 8 NVIDIA H20 GPUs and an Intel P5510 NVMe drive, providing up to 7 GB/s disk read bandwidth. The *GH200* platform is a GH200 Grace Hopper superchip [35] node, which contains one NVIDIA H100 GPU integrated with one NVIDIA Grace 64-core ARM CPU. The *GH200* system is equipped with 464 GB of LPDDR5X DRAM, providing up to 384 GB/s of memory bandwidth (unidirectional) to the CPU.

Baselines. We compare Strata with following state-of-the-art baselines. *vLLM* [26] is a popular open-source serving engine. Additionally, *vLLM-LMCache* enables hierarchical caching on vLLM using the official community extension of LMCache [30]. For our benchmarks, we used vLLM v0.8.5 and LMCache v0.2.1. The LMCache chunk size is set to 256 as default and vLLM page size was set to 32 in line with prior work [14]. *TensorRT-LLM* [36] (TRT-LLM) is an open-source serving library from NVIDIA, specialized for NVIDIA GPUs. Additionally, *TRT-LLM-HiCache* enables hierarchical caching on top of TensorRT-LLM through its automatic CPU memory offloading feature. We used TensorRT-LLM v0.17.0 in our benchmarks, with the page size also set to 32 as default. *SGLang* [53] is an open-source serving engine that delivers comparable performance to vLLM, while offering a more lightweight and customizable architecture. To enable a direct comparison to Strata, we implemented *SGLang-HiCache* which incorporates a state-of-the-art layer-wise KV cache transfer overlapping and hierarchical caching implementation using *cudaMemcpyAsync* transfers, which is in line with prior work including CachedAttention [14], Pensieve [52] and FlashGen [21]. We used SGLang v0.4.5 for all three systems. We set the page size for *SGLang* and Strata to 1 (SGLang's default), and the page size for *SGLang-HiCache* to 32 to be consistent with other hierarchical cache baselines.

Models. We utilize three popular open-source LLMs with long context capabilities, spanning small, medium, and large sizes: Llama-3.1-8B-Instruct [3] (128k context window), Qwen2.5-14B-Instruct-1M [46] (1M context window), and Llama-3.1-70B-Instruct [3] (128k context window). We use a single GPU to serve the 8B and 14B models, and 4 GPUs configured with tensor parallelism for 70B model.

Datasets. We construct workloads from three long context datasets. *LooGLE* [28] features long documents from diverse sources such as arXiv, Wikipedia, and movie/TV scripts. In our benchmarks, we use its Wikipedia portion, which provides both long and short queries paired with the documents. *NarrativeQA* [25] is an influential long-context dataset for testing models' reading comprehension capabilities, featuring even longer context examples than LooGLE. We filtered documents exceeding 128k tokens because of context window limit of the test models, and sampled 50 documents from the remainder. These two datasets mirror classic RAG use cases, in which extensive contexts are repeatedly queried by multiple users over time like question-answering systems over technical manuals [15]. *ReviewMT* [45] is a multi-agent conversation dataset, where agents simulate reviewers to converse

	LooGLE	NarrativeQA	ReviewMT	ShareGPT
avg. in	21613	54797	17708	680.9
avg. out	15.60	13.00	208.3	260.9
# contexts	105	50	100	-
# queries	2410	1461	1092	200869

Table 1: Dataset statistics.

about the quality of technical papers to make final decisions. This represents a typical agentic workflow involving long contexts and relatively longer output compared to other datasets. We also include a dataset to evaluate Strata's performance in short-context scenarios. *ShareGPT* is a popular conversational dataset comprised of a large collection of relatively short conversation histories from thousands of users, and was used in prior hierarchical KV caching studies [14,52]. Table 1 summarizes the characteristics of these datasets.

Since individual query timestamps are not available in these datasets, we simulate query arrivals using a Poisson distribution to benchmark the system using varying request rates, following prior works [21, 52]. For conversational benchmarks (ReviewMT and ShareGPT), we preserve dependencies across conversation rounds. Consistent with the methodology in Pensieve [52], we insert a 60-second "thinking time" between an LLM's response and the user's subsequent query for ShareGPT. For the long-context benchmarks, queries are randomly sampled from the dataset. In §5.3.3, we further examine the performance characteristics under different workload patterns. To avoid benchmark timeouts and unbounded client-side queuing, we cap the maximum number of in-flight requests at 128. Note this is a concurrency cap, not a fixed batch size as the actual prefill batch size is determined by each serving engine's scheduler, token budget, and available GPU memory. GPU memory is allocated according to each serving engine's default policy to ensure fairness and performance. An exception is the ShareGPT dataset, where we restrict GPU memory to approximately 500K tokens to highlight the behavior of hierarchical caching baselines. For caching configurations that utilize CPU memory, we allocate 1 TB of system DRAM as pinned memory (400 GB on GH200 due to platform limits). Disk storage is not used in most benchmarks except in §5.3.5 due to limited support in baseline systems.

5.2 End-to-end Performance Comparison

Strata is designed to improve long-context serving by reducing response latency and increasing overall throughput. Accordingly, we evaluate the system using two primary metrics: average Time To First Token (TTFT) and output token throughput. TTFT captures query response time, a key determinant of user experience, while output token throughput is a widely adopted metric for characterizing LLM serving system performance [1].

5.2.1 How does the performance of Strata compare to state-of-the-art LLM serving systems on long-context workloads?

Figure 8 presents the achieved token throughput and corresponding average TTFT for each system under varying request rates across three models and four datasets. First, we observe that *hierarchical caching is essential for high-performance long-context serving*. Across all three models, non-hierarchical caching solutions perform poorly on LooGLE. Without hierarchical caching, long-context workloads rapidly exhaust GPU memory capacity, leading to frequent cache misses during prefills. This, in turn, triggers repeated recomputation, resulting in low throughput and high latency. By contrast, systems equipped with hierarchical caching achieve substantially higher throughput and lower latency, consistently reaching approximately 95% cache hit rate by leveraging CPU memory.

Second, *Strata delivers substantial improvements over existing hierarchical caching solutions*. For Llama-8B on LooGLE, Strata achieves up to 3.2×, 2.6×, and 1.9× higher throughput at the same TTFT compared to *SGLang-HiCache*, *vLLM-LMCache*, and *TRT-LLM-HiCache*, respectively. Similar gains are observed for Qwen-14B, Strata achieves up to 3.9×, 2.1×, and 1.9× improvements; and Llama-70B, the gains reach 5×, 5×, and 3.75×. A consistent trend is seen on ReviewMT. Although longer decoding reduces the dominance of prefill time, with Llama-8B, Strata outperforms *SGLang-HiCache* by 1.7×, *vLLM-LMCache* by 2.3× and *TRT-LLM-HiCache* by 2.3×. These performance gains stem from Strata 's enhanced I/O efficiency and scheduling mechanisms, as further analyzed in §5.3.1.

5.2.2 How does Strata perform with a warm cache?

NarrativeQA presents much longer average context lengths, resulting in an extensive prefill phase that made non-hierarchical caching solutions impractical. Because this scenario presented the highest amount of memory pressure, we augmented this experiment to better understand steady-state performance characteristics (i.e., after the cache hierarchy has been filled)[1]. In this setup, we first warmed up the CPU memory by pre-computing KV caches for all contexts in the evaluation set and then flushed the KV cache on GPU memory to set the initial state. The second row of Figure 8 reports the throughput-latency curve after restarting the workload post-warmup. We report Strata, *SGLang-HiCache*, and *vLLM-LMCache*, as *TRT-LLM-HiCache* does not support pre-warming. In this setting, Strata achieves up to 2.3×, 2.6× and 2.5× throughput compared with *vLLM-LMCache* on Llama-8B, Qwen-14B and Llama-70B models respectively.

5.2.3 How does Strata perform with short-context?

Strata was explicitly designed for long-context workloads. To understand the performance of Strata on short-context workloads, the final row of Figure 8 shows the average TTFT of the baseline systems, across three models, on the ShareGPT dataset. Note that the underlying SGLang engine exhibits a slight performance disadvantage compared to the base engines

[1] NarrativeQA without pre-warming achieves similar results to the LooGLE workload. We omit results for this scenario due to space.

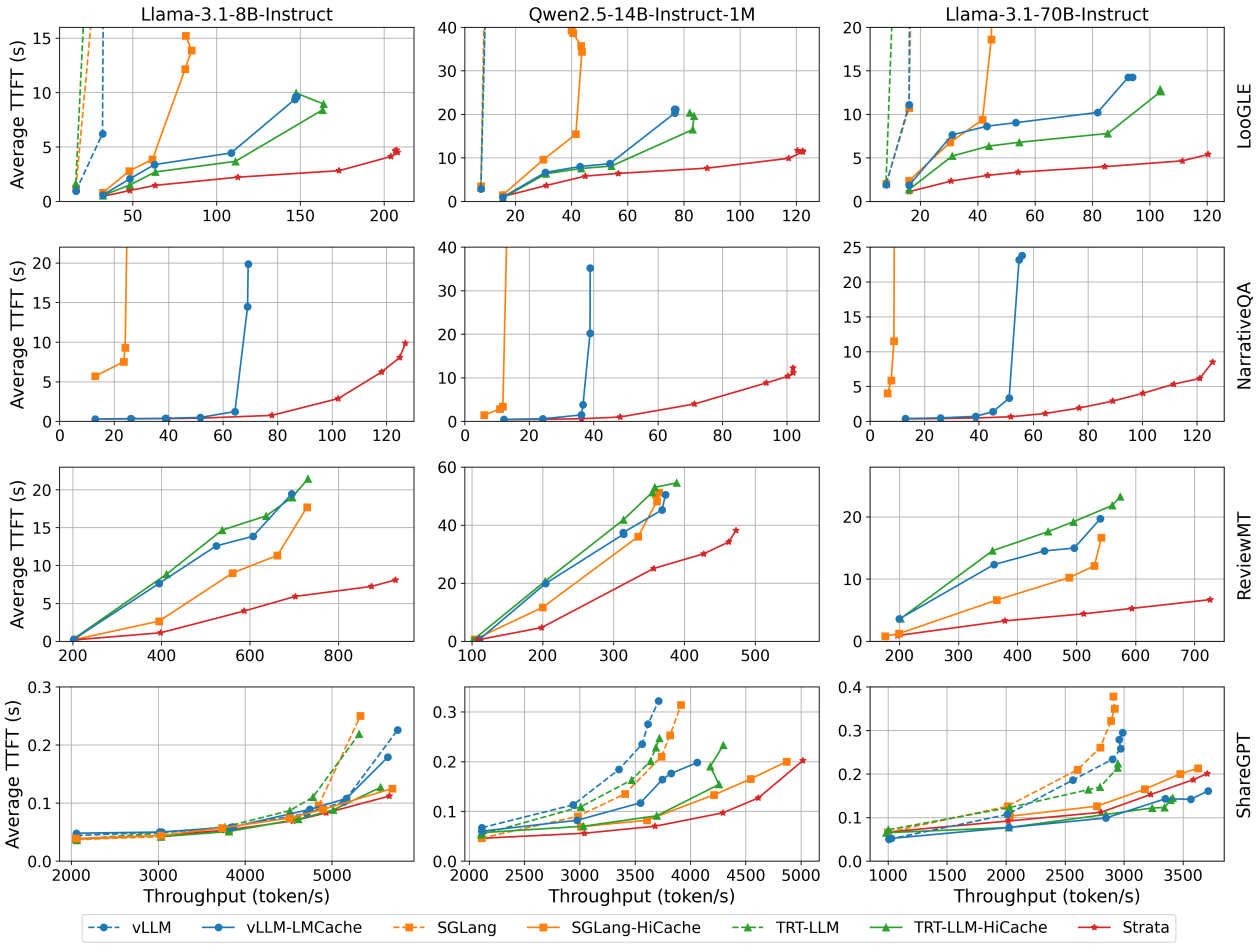


Figure 8: End-to-end benchmark performance comparison on H200.

of vLLM and TRT-LLM on the Llama-8B and -70B models due to kernel differences. Taking this into account, *Strata demonstrates comparable performance to the other state-of-the-art systems on short-context workloads*.

5.3 In-depth Performance Analysis

In this section, we conduct several breakdown analyses to better understand the performance benefits offered by Strata. Unless otherwise specified, all analyses use the LooGLE benchmark with Qwen2.5-14B model on an H200 platform.

5.3.1 How much do efficient I/O and scheduling benefit Strata?

Figure 9 presents the throughput-latency curves of Strata compared to three baselines. On top of *SGLang-HiCache*, we build and evaluate three ablated variants: *Strata-IO*, which incorporates the GPU-assisted I/O mechanism from §4.2, *Strata-Schedule-Only*, which applies the scheduling policy from §4.3, and *Strata-IO-LPM*, which integrates a longest prefix match (LPM) policy [53].

The results show that both the *Strata-scheduling* and *Strata-IO* components significantly improve the baseline hierarchical design, achieving up to 1.8× and 2.3× higher peak throughput, respectively. Each component alleviates the loading stall problem from a different perspective. Under low request rates, *Strata-scheduling* tends to deliver greater gains than *Strata-IO*, since smaller batch sizes generate lighter I/O pressure that can be more effectively mitigated by advanced scheduling. As the request rate increases, however, the I/O subsystem becomes the dominant bottleneck, making the GPU-assisted I/O mechanisms essential for sustaining high throughput.

We further compare *vLLM-LMCache* and *TRT-LLM-HiCache* directly with *Strata-IO*, since all three employ CUDA kernels to accelerate KV-cache I/O. As shown in Figure 9, their performance is comparable at low request rates, but *Strata-IO* maintains higher throughput as the request rate rises, indicating more effective mitigation of interference at scale. We also compare Strata with *Strata-IO-LPM*, which in-

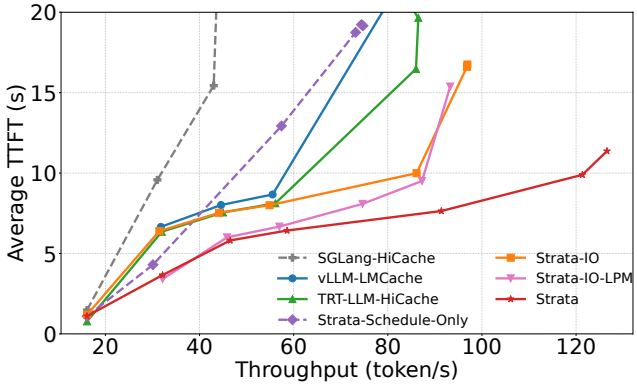


Figure 9: Breakdown of I/O and scheduling of Strata.

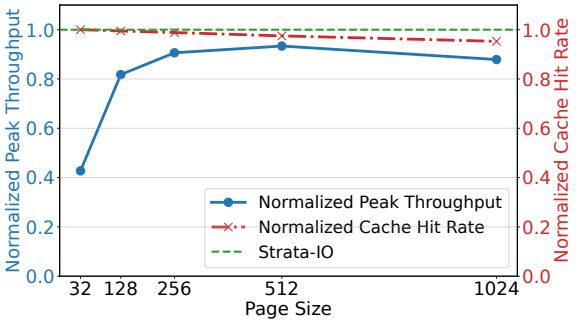


Figure 10: Performance comparison between Strata-IO and SGLang-HiCache with different page sizes.

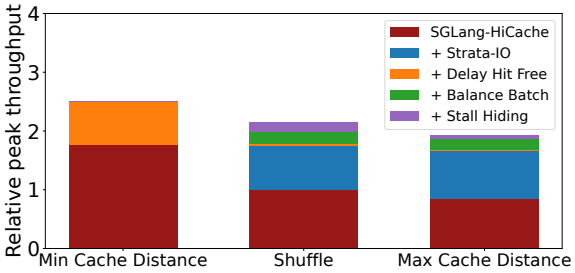


Figure 11: Breakdown of Strata's optimization attributions on different workload patterns.

creases the reuse count of on-device pages, thereby indirectly reducing host-side loading pressure and improving performance under low request rates. However, at higher request rates, it fails to sustain performance gains due to more frequent cache evictions. In contrast, Strata consistently delivers improvements because it explicitly accounts for bandwidth resources in its design.

5.3.2 Can Strata alleviate the burden of choosing a page size?

As discussed in §3.1, page size introduces inherent trade-offs among cache hit rate, I/O efficiency, and overall performance. This makes selecting the optimal page size a practical but non-trivial problem. Strata makes transfer efficiency much less sensitive to page size, reducing the need to choose large pages solely to obtain efficient transfers. To validate this claim, Figure 10 reports the peak throughput of *SGLang-HiCache* across different page sizes, normalized to *Strata-IO*. For *SGLang-HiCache*, increasing the page size initially improves throughput, since larger pages reduce loading stalls. However, beyond a certain threshold, throughput declines as cache hit rate deteriorates. Even at its best-performing setting (page size 512), *SGLang-HiCache* achieves only 93% of *Strata-IO*'s performance, primarily due to a 2.4% lower cache hit rate.

5.3.3 Can Strata adapt to varying cache distances?

Cache distance is an important property of a trace for testing cache system performance. To further augment the cache distance of our workloads, we generate two additional workloads based on the LooGLE dataset. In addition to our original (*shuffle*) workload, we create a *minimum cache distance* workload, where requests sharing the same context are lined up together, and a *maximum cache distance* workload, where queries sharing the same context are evenly distributed in the queue. As shown in Figure 11, we observe that with minimal cache distance, there is no need for hierarchical caching due to the perfect locality. In this scenario, delay hit mitigation benefits the most by improving effective cache hit rate, increasing peak throughput by 42%. However, delay hit mitigation offers no benefit in the max cache distance scenario because the high distance between similar requests naturally reduces the likelihood of the delay hit phenomenon. In contrast, the I/O efficiency mechanisms presented in §4.2 result in an improvement of 76% and 95% for the shuffle and maximum cache distance, respectively, as larger cache distances result in more cache hits to CPU DRAM. On top of that, balance batch further introduces 11% and 12% of the peak throughput improvement, respectively. Stall hiding further contributes 8% and 3% for shuffle and max cache distance, with higher benefit to the shuffle pattern due to higher variance.

5.3.4 What causes delay hit?

Beyond the request patterns discussed in §5.3.3, two system-level factors largely govern the severity of the delay hit effect: cache resolve time and system throughput (and thus the request arrival rate). As shown in Figure 12, longer cache resolve times (driven by prefill and scheduling latency) and higher throughput levels both increase the number of cache misses due to delay hits. A typical cache resolve time for a 1k-token input ranges from hundreds of milliseconds to several seconds, depending on the model and hardware. Note we study this effect using a simulated execution of the Moon-

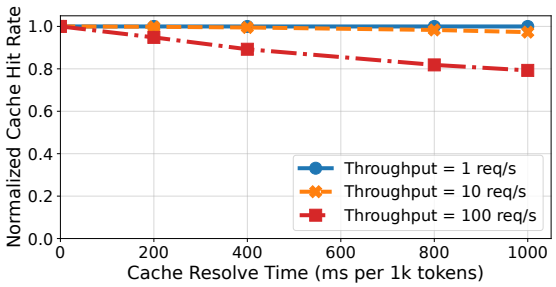


Figure 12: Delay hit impact for the Mooncake Tool-Agent trace under varying cache resolve times and throughputs. Different throughputs are achieved by scaling query timestamps.

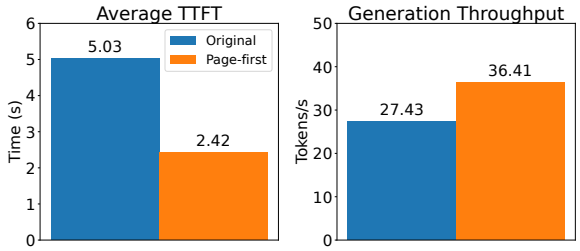


Figure 13: End-to-end performance improvement of the decoupled memory layout for DeepSeek-V3 on 8×H20 GPUs, with page size 32 and request rate 12 req/s.

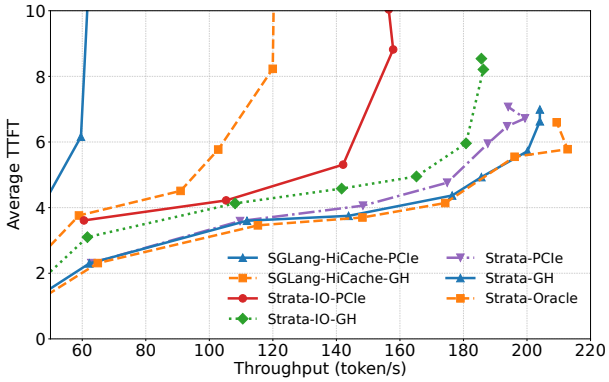


Figure 14: A zoomed-in comparison between benchmarks on PCIe-5.0 and Grace-Hopper platform.

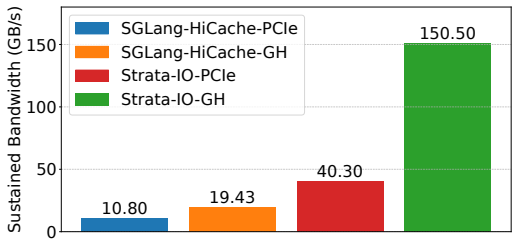


Figure 15: Sustained Bandwidth comparison.

cake trace with an effectively unlimited cache (to eliminate eviction effects), since our testbed cannot sustain sufficiently high throughput.

5.3.5 Does the decoupled memory layout benefit disk caching?

GPU-assisted I/O enables a page-first layout in host memory (and disk) without changing the GPU-resident layout (§4.2.1). Because the disks on the *H200* testbed are bandwidth-limited (<1 GiB/s), we evaluate this effect separately using the DeepSeek-V3 [12] model on the *H20-storage* platform. As shown in Figure 13, at a request rate of 12 req/s, the *page-first* layout improves average TTFT by 2.1× and throughput by 1.3× over an already reasonably large page size.

5.4 Benchmark on GH200 machine

Finally, we explored how Strata aligns with emerging hardware trends that dramatically increase the bandwidth between CPU and GPU. To do so, we benchmarked *SGLang-HiCache*, *Strata-IO* and *Strata* on both our H200 and GH200 platform. We also report a *Strata-Oracle* platform, which simulates the TTFT achieved by a system that had infinite bandwidth between the CPU and GPU.

Figure 14 reports the average TTFT achieved by *SGLang*, *Strata-IO*, and *Strata* across both of these platforms benchmarking Llama-3.1-8B models with LooGLE dataset. Figure 15 reports the averaged sustained bandwidth for the same task. We observe that, in line with our previous evaluation results, standard DMA-managed memory copy is not able to effectively utilize hardware improvement without using larger pages. While the improved bandwidth of GH200 improves latency for the SGLang baseline, only hardware improvements alone cannot outperform even Strata-IO on the H200 platform.

In contrast, using *Strata-IO* increases sustained host-GPU transfer bandwidth from 40 GB/s to 150 GB/s. However, scheduling improvements are still needed to take full advantage of platforms like Grace Hopper. While *Strata-IO-GH* achieves higher throughput with higher bandwidth mitigating the I/O stall, it still does not even outperform *Strata-PCIe*. In summary, Strata's improved I/O transfer mechanisms *with* its bandwidth-aware scheduler can take full advantage of emerging platforms like Grace Hopper, achieving comparable performance to the Oracle setting. These benchmarks reveal that while increased interconnect bandwidth is beneficial, it's often under-utilized by existing software. Ideally, leveraging new hardware capabilities, as demonstrated by *Strata-GH*'s near-oracle performance, could unlock new possibilities. Specifically, this platform shows promise for specialized, cost-effective, and high-performance long-context serving.

6 Discussion

Having demonstrated Strata's advantages, we now discuss its limitations and directions for future work.

I/O Kernel Interference. As analyzed in §4.2, Strata's GPU-assisted I/O trades a small amount of SM contention for high transfer efficiency on small, fragmented pages. CUDA 12.8 recently introduced `cudaMemcpyBatchAsync` [38], which improves transfer efficiency by batching many small copy operations into a single host-driver submission. Because it engages the GPU DMA engine, it does not compete with concurrent computation for SM resources. In the same H200 microbenchmark used for Figure 5, its throughput is lower than our GPU-assisted I/O kernel, 38 GB/s versus 48 GB/s. While this lower throughput could introduce additional stalls on the loading path, it makes the API a good fit for the less critical GPU-to-CPU backup path. Strata can therefore be configured to use the GPU-assisted I/O kernel for CPU-to-GPU loading and the batch-copy API for GPU-to-CPU backup, achieving competitive bandwidth while further reducing kernel interference. Looking forward, we plan to further reduce GPU-assisted kernel overhead and believe these results motivate more versatile on-chip I/O accelerators.

Fairness in Scheduling. Although §4.3.2 introduces a mechanism to prevent starvation, Strata's scheduler prioritizes aggregate I/O and compute efficiency and can still treat requests unevenly, risking Service-level objective (SLO) violations for individual requests. A natural extension is to make batch formation SLO- and fairness-aware when allocating resources. The current policies also focus on hiding the latency of moving data from host DRAM to HBM; an equally promising direction is to create scheduling headroom for hiding the latency of prefetching from slower storage tiers, going beyond Strata's current approach of overlapping prefetches with queuing delay.

Model Coverage. Strata targets hierarchical caching for the dense KV caches produced by standard attention. As hybrid mechanisms such as sparse and linear attention gain adoption, extending the same level of support to these models becomes increasingly important and raises distinct challenges in both memory management and scheduling-policy design.

7 Related Work

Context Caching and Sharing. Prior work has explored several mechanisms for reusing KV caches to avoid redundant computation. SGLang [53] uses a RadixTree to track shared context, while other serving engines such as vLLM [26] and Mooncake [43] employ hashing schemes that generate unique page identifiers from token IDs and prefix page hashes. LMDeploy [10] adopts a hybrid design with coarser-grained tries. Strata builds on SGLang by extending its RadixTree into a HiRadixTree. Beyond exact-prefix reuse, CacheGen [29] and CacheBlend [50] explore approximate KV cache sharing. In contrast, Strata focuses on exact caching and does not affect model accuracy. Marconi [42] adds prefix cache support for hybrid LLMs, which is complementary to Strata and a promising direction for extending our framework to hybrid models.

KV Cache Offloading. Several recent works have explored utilizing secondary memory tiers (e.g., CPU DRAM, SSDs) for KV cache storage, loading them for computation on demand. CachedAttention [14] and Pensieve [52] both adopt a layer-wise strategy to overlap KV cache loading with computation. FlashGen [21] further enhances this pipeline with re-order execution scheduling, which has been implemented in SGLang and used in our baseline settings.

Efficient Host-GPU I/O. Practitioners have long recognized that GPUs can accelerate small data movement between GPU and CPU memory [33, 44]. Strata introduces a GPU-assisted KV-cache transfer mechanism tailored to LLM serving and, to our knowledge, provides the first quantitative analysis of the trade-offs between transfer efficiency, cache hit rate, and interference overhead. GDRCopy [24] and CUDA Unified Memory [37] enable low-latency fine-grained host-GPU data transfer, but their design makes it difficult to sustain the bandwidth required for large KV-cache transfers.

Large-scale KV Cache Disaggregation. Recent works propose building large-scale disaggregated KV cache memory pools and global resource coordinators to achieve caching benefits at a larger scale. Mooncake [43] exploits using resources including CPU, DRAM, SSD and NIC to establish a disaggregated KV Cache. MemServe [19] unifies inter-request and intra-request KV cache optimizations via a global scheduler and an elastic memory pool. Strata can benefit from these designs by integrating with their KV cache transfer engines. However, Strata focuses on memory management and scheduling within single compute instances and does not inherently rely on specialized hardware, such as high-speed networking, to realize its caching benefits.

8 Conclusion

This paper presented Strata, a hierarchical context caching framework that tackles the key bottlenecks of long-context LLM serving. By combining GPU-assisted I/O to mitigate KV cache fragmentation with cache-aware scheduling that balances computation and data transfer, Strata improves utilization across diverse latency regimes. Our evaluation shows that Strata outperforms state-of-the-art systems on long-context benchmarks while preserving strong performance on short-context workloads, establishing Strata as a practical and scalable solution for efficient long-context LLM serving.

Acknowledgments

We are grateful to the anonymous reviewers for their constructive comments, which helped improve this paper. We thank Lingfan Yu for insightful discussions on Pensieve. We also thank Sicheng Pan, Tingwei Huang, Zhangheng Huang, Shuwen Wang, Teng Ma, Mingxing Zhang, Arnav Balyan, Moein Khazraee, Zhenwei Pi, Yi Zhang, Shangming Cai, Shiyang Chen, Ke Yang, and Ying Sheng from the SGLang open-source community for their valuable feedback and contributions. This research was supported in part by the Stanford Platform Lab and its affiliates. Zhiqiang Xie was supported by the NVIDIA Graduate Fellowship. We are grateful to NVIDIA and Nebius for providing computational resources.

References

[1] Artificial analysis model leaderboards. https://artificialanalysis.ai/leaderboards/models.

[2] Advanced Micro Devices, Inc. ROCm™ Software 6.4.3 Documentation. https://rocm.docs.amd.com/en/latest/, June 2025.

[3] Meta AI. Introducing llama 3.1: Our most capable models to date. https://ai.meta.com/blog/meta-llama-3-1/, July 2024.

[4] Meta AI. The llama 4 herd: The beginning of a new era of natively multimodal ai innovation. https://ai.meta.com/blog/llama-4-multimodal-intelligence/, April 2025.

[5] Anthropic. Prompt caching. https://docs.anthropic.com/en/docs/build-with-claude/prompt-caching, 2024.

[6] Anthropic. Claude code: Agentic coding assistant. https://code.claude.com/, 2025.

[7] Anthropic. Claude sonnet 4 now supports 1m tokens of context. https://claude.com/blog/1m-context, 2025.

[8] Nirav Atre, Justine Sherry, Weina Wang, and Daniel S. Berger. Caching with delayed hits. In *Proceedings of the Annual Conference of the ACM Special Interest Group on Data Communication on the Applications, Technologies, Architectures, and Protocols for Computer Communication*, SIGCOMM '20, page 495–513, New York, NY, USA, 2020. Association for Computing Machinery.

[9] Qizhe Cai, Shubham Chaudhary, Midhul Vuppalapati, Jaehyun Hwang, and Rachit Agarwal. Understanding host network stack overheads. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*, page 65–77, New York, NY, USA, 2021. Association for Computing Machinery.

[10] LMDeploy Contributors. Lmdeploy: A toolkit for compressing, deploying, and serving llm. https://github.com/InternLM/lmdeploy, 2023.

[11] DeepSeek. Prompt caching. https://api-docs.deepseek.com/guides/kv_cache.

[12] DeepSeek-AI. Deepseek-v3 technical report. https://arxiv.org/abs/2412.19437, 2025.

[13] Assaf Eisenman, Asaf Cidon, Evgenya Pergament, Or Haimovich, Ryan Stutsman, Mohammad Alizadeh, and Sachin Katti. Flashield: a hybrid key-value cache that controls flash write amplification. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 65–78, 2019.

[14] Bin Gao, Zhuomin He, Puru Sharma, Qingxuan Kang, Djordje Jevdjic, Junbo Deng, Xingkun Yang, Zhou Yu, and Pengfei Zuo. Cost-efficient large language model serving for multi-turn conversations with cachedattention. In *Proceedings of the 2024 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC'24, USA, 2024. USENIX Association.

[15] Yunfan Gao, Yun Xiong, Xinyu Gao, Kangxiang Jia, Jinliu Pan, Yuxi Bi, Yixin Dai, Jiawei Sun, Meng Wang, and Haofen Wang. Retrieval-augmented generation for large language models: A survey. *arXiv preprint arXiv:2312.10997*, 2(1), 2023.

[16] In Gim, Guojun Chen, Seung-seob Lee, Nikhil Sarda, Anurag Khandelwal, and Lin Zhong. Prompt cache: Modular attention reuse for low-latency inference. In P. Gibbons, G. Pekhimenko, and C. De Sa, editors, *Proceedings of Machine Learning and Systems*, volume 6, pages 325–338, 2024.

[17] Google. Prompt caching. https://cloud.google.com/vertex-ai/generative-ai/docs/context-cache/context-cache-overview.

[18] Google DeepMind. Gemini: Google deepmind's most capable and general ai models. https://deepmind.google/technologies/gemini/, 2025.

[19] Cunchen Hu, Heyang Huang, Junhao Hu, Jiang Xu, Xusheng Chen, Tao Xie, Chenxi Wang, Sa Wang, Yungang Bao, Ninghui Sun, and Yizhou Shan. Memserve: Context caching for disaggregated llm serving with elastic memory pool, 2024.

[20] Changho Hwang, KyoungSoo Park, Ran Shu, Xinyuan Qu, Peng Cheng, and Yongqiang Xiong. ARK: GPU-driven code execution for distributed deep learning. In

20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23), pages 87–101, Boston, MA, April 2023. USENIX Association.

[21] Jinwoo Jeong and Jeongseob Ahn. Accelerating llm serving for multi-turn dialogues with efficient resource management. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, page 1–15, New York, NY, USA, 2025. Association for Computing Machinery.

[22] Chao Jin, Zili Zhang, Xuanlin Jiang, Fangyue Liu, Shufan Liu, Xuanzhe Liu, and Xin Jin. Ragcache: Efficient knowledge caching for retrieval-augmented generation. *ACM Trans. Comput. Syst.*, 44(1), November 2025.

[23] Shuowei Jin, Xueshen Liu, Qingzhao Zhang, and Zhuoqing Mao. Compute or load kv cache? why not both? In *International Conference on Machine Learning*, pages 28031–28043. PMLR, 2025.

[24] Kawthar Shafie Khorassani, Ching-Hsiang Chu, Hari Subramoni, and Dhabaleswar K. Panda. Performance evaluation of mpi libraries on gpu-enabled openpower architectures: Early experiences. In *High Performance Computing: ISC High Performance 2019 International Workshops, Frankfurt, Germany, June 16-20, 2019, Revised Selected Papers*, page 361–378, Berlin, Heidelberg, 2019. Springer-Verlag.

[25] Tomáš Kočiský, Jonathan Schwarz, Phil Blunsom, Chris Dyer, Karl Moritz Hermann, Gábor Melis, and Edward Grefenstette. The NarrativeQA reading comprehension challenge. *Transactions of the Association for Computational Linguistics*, 6:317–328, 2018.

[26] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph Gonzalez, Hao Zhang, and Ion Stoica. Efficient memory management for large language model serving with pagedattention. In *Proceedings of the 29th Symposium on Operating Systems Principles*, page 611–626, New York, NY, USA, 2023. Association for Computing Machinery.

[27] Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen-tau Yih, Tim Rocktäschel, Sebastian Riedel, and Douwe Kiela. Retrieval-augmented generation for knowledge-intensive nlp tasks. In *Proceedings of the 34th International Conference on Neural Information Processing Systems*, Red Hook, NY, USA, 2020. Curran Associates Inc.

[28] Jiaqi Li, Mengmeng Wang, Zilong Zheng, and Muhan Zhang. LooGLE: Can long-context language models understand long contexts? In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 16304–16333, Bangkok, Thailand, 2024. Association for Computational Linguistics.

[29] Yuhan Liu, Hanchen Li, Yihua Cheng, Siddhant Ray, Yuyang Huang, Qizheng Zhang, Kuntai Du, Jiayi Yao, Shan Lu, Ganesh Ananthanarayanan, et al. Cachegen: Kv cache compression and streaming for fast large language model serving. In *Proceedings of the ACM SIGCOMM 2024 Conference*, pages 38–56, 2024.

[30] LMCache. Lmcache. https://lmcache.ai, 2025.

[31] Lingxiao Ma, Zhiqiang Xie, Zhi Yang, Jilong Xue, Youshan Miao, Wei Cui, Wenxiang Hu, Fan Yang, Lintao Zhang, and Lidong Zhou. Rammer: Enabling holistic deep learning compiler optimizations with rTasks. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 881–897. USENIX Association, November 2020.

[32] Sara McAllister, Benjamin Berg, Julian Tutuncu-Macias, Juncheng Yang, Sathya Gunasekar, Jimmy Lu, Daniel S. Berger, Nathan Beckmann, and Gregory R. Ganger. Kangaroo: Caching billions of tiny objects on flash. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, page 243–262, New York, NY, USA, 2021. Association for Computing Machinery.

[33] Seung Won Min, Vikram Sharma Mailthody, Zaid Qureshi, Jinjun Xiong, Eiman Ebrahimi, and Wen-mei Hwu. Emogi: efficient memory-access for out-of-memory graph-traversal in gpus. *Proc. VLDB Endow.*, 14(2):114–127, October 2020.

[34] NVIDIA. 5x faster time to first token with nvidia tensorrt-llm kv cache early reuse. https://developer.nvidia.com/blog/5x-faster-time-to-first-token-with-nvidia-tensorrt-llm-kv-cache-early-reuse/, 2024.

[35] NVIDIA. Nvidia gh200 grace hopper superchip. https://www.nvidia.com/en-us/data-center/grace-hopper-superchip/, 2025.

[36] NVIDIA. Nvidia tensorrt-llm. https://docs.nvidia.com/tensorrt-llm/index.html, 2025.

[37] NVIDIA Corporation. 4.1. unified memory. https://docs.nvidia.com/cuda/archive/13.1.0/, 2025.

[38] NVIDIA Corporation. CUDA Runtime API: Memory Management. https://docs.nvidia.com/cuda/archive/12.8.1/cuda-runtime-api/group__CUDART__MEMORY.html, 2025.

[39] NVIDIA Corporation. Parallel thread execution isa. https://docs.nvidia.com/cuda/parallel-thread-execution/, 2025.

[40] OpenAI. Memory and new controls for chatgpt. https://openai.com/index/memory-and-new-controls-for-chatgpt/, 2024.

[41] OpenAI. Prompt caching. https://platform.openai.com/docs/guides/prompt-caching, 2024.

[42] Rui Pan, Zhuang Wang, Zhen Jia, Can Karakus, Luca Zancato, Tri Dao, Yida Wang, and Ravi Netravali. Marconi: Prefix caching for the era of hybrid llms. *Proceedings of Machine Learning and Systems*, 7, 2025.

[43] Ruoyu Qin, Zheming Li, Weiran He, Jialei Cui, Feng Ren, Mingxing Zhang, Yongwei Wu, Weimin Zheng, and Xinran Xu. Mooncake: Trading more storage for less computation — a KVCache-centric architecture for serving LLM chatbot. In *23rd USENIX Conference on File and Storage Technologies (FAST 25)*, pages 155–170, Santa Clara, CA, February 2025. USENIX Association.

[44] Zaid Qureshi, Vikram Sharma Mailthody, Isaac Gelado, Seungwon Min, Amna Masood, Jeongmin Park, Jinjun Xiong, C. J. Newburn, Dmitri Vainbrand, I-Hsin Chung, Michael Garland, William Dally, and Wen-mei Hwu. Gpu-initiated on-demand high-throughput storage access in the bam system architecture. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, page 325–339. ACM, January 2023.

[45] Cheng Tan, Dongxin Lyu, Siyuan Li, Zhangyang Gao, Jingxuan Wei, Siqi Ma, Zicheng Liu, and Stan Z Li. Peer review as a multi-turn and long-context dialogue with role-based interactions. *arXiv preprint arXiv:2406.05688*, 2024.

[46] Qwen Team. Qwen2.5-1m: Deploy your own qwen with context length up to 1m tokens. https://qwenlm.github.io/blog/qwen2.5-1m/, January 2025.

[47] The SGLang Team. Sglang v0.4: Zero-overhead batch scheduler, cache-aware load balancer, faster structured outputs. https://lmsys.org/blog/2024-12-04-sglang-v0-4/, December 2024.

[48] vLLM Contributors. vllm configuration api reference. https://docs.vllm.ai/en/latest/api/vllm/vllm.config.html, 2025.

[49] Minzheng Wang, Longze Chen, Fu Cheng, Shengyi Liao, Xinghua Zhang, Bingli Wu, Haiyang Yu, Nan Xu, Lei Zhang, Run Luo, Yunshui Li, Min Yang, Fei Huang, and Yongbin Li. Leave no document behind: Benchmarking long-context LLMs with extended multi-doc QA. In Yaser Al-Onaizan, Mohit Bansal, and Yun-Nung Chen, editors, *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*, pages 5627–5646, Miami, Florida, USA, November 2024. Association for Computational Linguistics.

[50] Jiayi Yao, Hanchen Li, Yuhan Liu, Siddhant Ray, Yihua Cheng, Qizheng Zhang, Kuntai Du, Shan Lu, and Junchen Jiang. Cacheblend: Fast large language model serving for rag with cached knowledge fusion. In *Proceedings of the Twentieth European Conference on Computer Systems*, EuroSys '25, page 94–109, New York, NY, USA, 2025. Association for Computing Machinery.

[51] Gyeong-In Yu, Joo Seong Jeong, Geon-Woo Kim, Soojeong Kim, and Byung-Gon Chun. Orca: A distributed serving system for Transformer-Based generative models. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 521–538, Carlsbad, CA, July 2022. USENIX Association.

[52] Lingfan Yu, Jinkun Lin, and Jinyang Li. Stateful large language model serving with pensieve. In *Proceedings of the Twentieth European Conference on Computer Systems*, EuroSys '25, page 144–158, New York, NY, USA, 2025. Association for Computing Machinery.

[53] Lianmin Zheng, Liangsheng Yin, Zhiqiang Xie, Chuyue Sun, Jeff Huang, Cody Hao Yu, Shiyi Cao, Christos Kozyrakis, Ion Stoica, Joseph E. Gonzalez, Clark Barrett, and Ying Sheng. Sglang: Efficient execution of structured language model programs. In A. Globerson, L. Mackey, D. Belgrave, A. Fan, U. Paquet, J. Tomczak, and C. Zhang, editors, *Advances in Neural Information Processing Systems*, volume 37, pages 62557–62583. Curran Associates, Inc., 2024.

[54] Yinmin Zhong, Shengyu Liu, Junda Chen, Jianbo Hu, Yibo Zhu, Xuanzhe Liu, Xin Jin, and Hao Zhang. Distserve: disaggregating prefill and decoding for goodput-optimized large language model serving. In *Proceedings of the 18th USENIX Conference on Operating Systems Design and Implementation*, OSDI'24, USA, 2024. USENIX Association.

[55] Kan Zhu, Yufei Gao, Yilong Zhao, Liangyu Zhao, Gefei Zuo, Yile Gu, Dedong Xie, Tian Tang, Qinyu Xu, Zihao Ye, Keisuke Kamahori, Chien-Yu Lin, Ziren Wang, Stephanie Wang, Arvind Krishnamurthy, and Baris Kasikci. Nanoflow: towards optimal large language model serving throughput. In *Proceedings of the 19th USENIX Conference on Operating Systems Design and Implementation*, OSDI '25, USA, 2025. USENIX Association.

ARTIFACT EVALUATED
USENIX
AVAILABLE

ECHO: Efficient KV Cache Offloading with Lossless Prefetching for Serving Native Sparse Attention LLMs

Guangda Liu[1] Wenhao Chen[1] Chengwei Li[1] Zhenyu Ning[1] Jing Lin[2]
Yiwu Yao[2] Quan Chen[1] Shixuan Sun[1] Jieru Zhao[1]* Minyi Guo[3,1]

[1]*School of Computer Science, Shanghai Jiao Tong University* [2]*Huawei* [3]*Guizhou Provincial Laboratory of Big Data, College of Computer Science and Technology, Guizhou University*

Abstract

Native sparse attention has emerged as a promising approach for efficient long-context LLM inference without compromising accuracy. While it significantly reduces the attention computation and KV cache access costs, the KV cache size exhibits steeper linear growth with the context length. As a result, GPU HBM capacity becomes the bottleneck, limiting the concurrency of long-context requests and leading to poor hardware utilization and low generation throughput. We introduce ECHO, a serving system designed for native sparse-attention LLMs that employs KV cache offloading to overcome GPU HBM capacity limits. ECHO incorporates a graph-friendly cache manager that enables efficient dynamic KV cache eviction and recall entirely within GPU graphs, minimizing management overhead. Furthermore, by exploiting the numerical predictability of index scores and the sequential processing of queries, ECHO enables lossless intra-query prefetching for decoding and inter-query prefetching for prefill. By applying a fully pipelined fused GPU kernel, ECHO overlaps the recall overhead with indexer computation. Experiments show that ECHO delivers up to 2.1× higher generation throughput than state-of-the-art systems such as SGLang and vLLM under long-context workloads, while maintaining comparable latency under light load.

1 Introduction

Large language models (LLMs) have gained remarkable prominence for their ability to excel across diverse tasks and have been widely deployed in a variety of applications, such as chatbots, agents, and coding copilots [10, 24, 33]. To handle increasingly complex tasks such as multi-turn dialogue, long-horizon planning, and repository-level code understanding, the context windows of LLMs are rapidly expanding to accommodate longer inputs and maintain longer memory. Mainstream LLMs now typically support context windows of 128K tokens [15, 41], and frontier models can reach up to 1 million tokens [11, 57]. As a result, long-context inference has become a critical workload in production LLM serving.

Although extended context windows unlock powerful capabilities, processing long-context requests incurs significant efficiency issues because the cost of self-attention increases quadratically with context length. While KV cache is introduced to avoid recomputation, it does not change the quadratic complexity. Leveraging the inherent sparsity of attention computation, *sparse attention* has been extensively studied and employed to reduce the costs of attention [26, 49]. Typically, sparse attention methods first estimate the attention scores of tokens, prune less critical tokens, and then compute attention only over the top-k tokens with the highest scores.

These methods are training-free, making them simple to deploy. However, they often lead to accuracy degradation, which can impair model performance, particularly on complex tasks. To this end, recent works propose applying sparse attention during pre-training or post-training to mitigate or eliminate accuracy degradation [32, 65, 68]. For example, the recently released DeepSeek-V3.2 incorporates training-time sparse attention, employing a lightweight indexer to identify and select important tokens [13]. It shows comparable or better performance against state-of-the-art full attention models on various complex, long-context tasks [18], demonstrating the promising potential of *native sparse attention models*.

Sparse attention, whether training-free or training-time, significantly reduces the computation cost of long-context processing; however, it does not shrink the KV cache itself, and the memory footprint still grows linearly with context length. Consequently, GPU HBM capacity can be quickly exhausted even with a modest number of concurrent long-context requests. For instance, with DeepSeek-V3.2 deployed on 8 NVIDIA H200 GPUs (141 GB each), a single worker can accommodate only up to 655K tokens, allowing at most six concurrent requests of 100K tokens each. The resulting low request concurrency directly degrades hardware utilization, and the intrinsically lower arithmetic intensity of sparse attention kernels further exacerbates this under-utilization.

To overcome the limitations imposed by GPU HBM ca-

*Corresponding author.

pacity, recent works propose offloading the KV cache to host memory during inference with sparse attention, thereby alleviating GPU memory pressure and enabling higher request concurrency [6, 26]. Only KVs of selected tokens are loaded or *recalled* from the host to GPU memory for sparse attention computation. However, few KV cache offloading systems are designed for realistic serving scenarios. Most existing offloading approaches rely on inefficient cache-management strategies that introduce substantial overhead and are incompatible with graph execution such as CUDA Graphs, thereby limiting their practical applicability.

Moreover, the limited bandwidth of host-GPU interconnects such as PCIe can impose substantial overhead during KV recall operations. Several studies attempt to mitigate this overhead by pre-selecting important tokens and prefetching their KVs, exploiting inter-layer or inter-step similarity [26, 30, 62]. However, such prefetching introduces additional accuracy degradation beyond that inherent to sparse attention, undermining model reliability on complex tasks and limiting adoption in production deployments.

To address these issues, we present ECHO, a serving system for native sparse attention LLMs under long-context workloads through efficient KV cache offloading. Note that ECHO can also serve models with training-free sparse attention. Since these methods inherently incur accuracy degradation, we focus on more challenging native sparse attention LLMs with higher model performance in this work. ECHO improves serving efficiency along two axes. First, ECHO introduces a *graph-friendly cache manager* that eliminates variable-length tensors and updates cache states using fully parallel operations. This design enables all management tasks to execute entirely within the GPU graph, thereby minimizing management overhead. Second, to mitigate recall overhead without compromising accuracy, ECHO exploits the numerical predictability of index scores to break the top-k dependency, enabling *intra-query prefetching* during indexer execution in the decoding phase. For prefill, ECHO leverages the sequential processing of queries to enable *inter-query prefetching*. Both prefetching operations in ECHO are lossless, with guaranteed recall that preserves model accuracy. In addition, ECHO fuses prefetching into the indexer kernels using software pipelining, enabling prefetching to overlap with indexer computation.

In summary, this paper makes the following contributions:

- We identify the GPU memory-capacity limitations inherent in serving native sparse attention LLMs and present ECHO, an efficient long-context serving system with KV cache offloading.
- We design a graph-friendly cache manager that enables the entire decoding path to execute within a single graph, minimizing cache-management overhead.
- We reveal the numerical predictability of index scores, enabling lossless intra-query prefetching during decoding, and exploit the sequential processing of queries to enable lossless inter-query prefetching for prefill.

We implement ECHO atop the state-of-the-art serving framework SGLang [69] and kernel library DeepGEMM [12]. Experimental results show that ECHO delivers up to 2.1× generation throughput improvement over SGLang when serving DeepSeek-V3.2 under long-context workloads, while maintaining comparable latency under light load. Code is available at https://github.com/sjtu-zhao-lab/ECHO.

2 Background and Motivation

2.1 LLM Inference and KV Cache

Transformer LLMs comprise multiple Transformer layers, each with an attention module and a feed-forward network (FFN). In attention, the hidden states $\mathbf{X} \in \mathbb{R}^{L\times D}$ are projected into query, key and value tensors ($\mathbf{Q},\mathbf{K},\mathbf{V} \in \mathbb{R}^{L\times d}$), where L is the context length and D, d are the model and attention dimensions, and the output is $\text{Attn}(\mathbf{Q},\mathbf{K},\mathbf{V}) = \text{softmax}\big(\frac{\mathbf{Q}\mathbf{K}^\top}{\sqrt{d}}\big)\mathbf{V}$. To scale model capacity at low computational cost, modern LLMs augment FFNs with a Mixture-of-Experts (MoE) architecture of sparsely activated expert FFNs [9]. MoE parameters can reach hundreds of billions or even a trillion, imposing substantial GPU memory demands during inference [17, 50].

KV Cache LLMs generate tokens in an *autoregressive* manner, appending each newly produced token to the input sequence to generate the next. Mainstream LLMs adopt a *decoder-only* architecture in which a causal mask is applied to $\mathbf{Q}\mathbf{K}^\top$ during attention computation. Consequently, the $\mathbf{K}$ and $\mathbf{V}$ tensors depend only on preceding tokens and can be cached to avoid redundant computation, a mechanism known as *KV cache*. With KV cache, LLM inference proceeds in two stages: prefill and decoding. The prefill stage processes the entire input sequence, computes the KV cache, and produces the first output token. During decoding, the query vector of the most recently generated token, together with the cached KV tensors from all previous tokens, is used to compute attention for generating subsequent tokens. Prefill efficiency is typically evaluated using time-to-first-token (TTFT) and input-token throughput, whereas decoding performance is assessed using inter-token latency (ITL) and output-token throughput.

Attention Variants The original Transformer utilizes multi-head attention (MHA) with $\mathbf{Q},\mathbf{K},\mathbf{V} \in \mathbb{R}^{H\times L\times d}$, where H is the number of heads [53]. The attention of each head is computed independently, and outputs of all heads are concatenated and projected for later computation. To reduce the memory cost of KV cache and increase the arithmetic density, grouped-query attention (GQA) is adopted by many LLMs [2]. In GQA, $\mathbf{K},\mathbf{V} \in \mathbb{R}^{H_{kv}\times L\times d}$ contain fewer heads

than $\mathbf{Q}$, and a group of $G = \frac{H}{H_{kv}}$ query heads share a single KV head. While multi-query attention (MQA) further improves efficiency by setting $H_{kv} = 1$, it introduces substantial accuracy degradation, which has limited its broader adoption [47]. To strike a better balance between accuracy and efficiency, multi-head latent attention (MLA) employs low-rank compression to KV tensors [16]. Specifically, MLA first down-projects the hidden states to compressed *latent vectors*, i.e., $\mathbf{C} = \mathbf{W}^{DKV}\mathbf{X} \in \mathbb{R}^{L \times d_c}$, and then reconstructs the KV tensors for attention computation by up-projecting the latent vectors, namely $(\mathbf{K}, \mathbf{V}) = (\mathbf{W}^{UK}\mathbf{C}, \mathbf{W}^{UV}\mathbf{C})$. The up-projections can be further absorbed, and only the latent vectors $\mathbf{C}$ need to be cached during inference. Compared to GQA, MLA is both more mathematically expressive and more efficient [34], leading to its adoption in recent models [21, 50].

2.2 Sparse Attention in LLMs

The context windows of LLMs are rapidly expanding to support increasingly complex tasks, reaching 128K and even 1M tokens in state-of-the-art models [11]. To alleviate the quadratic growth of attention computation with context length, a variety of sparse-attention methods have been proposed. These approaches compute attention only over the KV tensors of the selected k tokens, reducing computational cost from $O(L^2)$ to $O(Lk)$.

Training-free Sparse Attention Most sparse attention methods are *training-free*, requiring no training or finetuning of the original models. Early methods select tokens statically, leveraging sliding window or vertical-slash patterns in attention [23, 58]. However, static sparsity typically incurs substantial accuracy degradation. To address this, dynamic methods determine sparsity patterns at inference time based on approximated attention scores (i.e., approximations of $\mathbf{QK}^\top$). For efficient score approximation, Quest [49] compresses each K block into two label vectors and estimates block importance from those labels to select important blocks for attention. Alternatively, InfiniGen [26] identifies critical channels of Q and K and approximates attention scores using only the retained channels. In general, dynamic sparse attention methods follow one of two strategies to reduce approximation cost: compressing along the context dimension or pruning along the channel dimension [6, 31, 60].

Training-time Sparse Attention To eliminate the accuracy degradation associated with training-free sparse attention, recent works explore *training-time* sparse attention. By incorporating sparsity during pre-training or post-training, model parameters are adapted to sparse-attention outputs, allowing the model to preserve its native accuracy at inference time. Sliding window attention (SWA) is one of the simplest forms of training-time sparse attention [22]. Following the block-wise paradigm, MoBA [32], NSA [65], and InfLLM-v2 [68] compress K blocks into label vectors using direct pooling or learnable MLPs.

DeepSeek Sparse Attention (DSA) Building on MLA, DSA implements training-time sparse attention by introducing a lightweight *indexer* [13]. The indexer is an MQA-like module, computing token-importance scores that are used to select the most relevant tokens for sparse MLA. Specifically, given a query token t, the importance, namely *the index score*, of token s is measured by

$$\sum_{h=1}^{H_{\text{index}}} w_{h,t}^I \,\text{ReLU}(q_{h,t}^I, k_s^I), \quad w_{h,t}^I \in \mathbb{R} \text{ and } q_{h,t}^I, k_s^I \in \mathbb{R}^{d_{\text{index}}}$$

where H_{index} is the number of indexer heads, d_{index} is the indexer dimension, $w_{h,t}^I, q_{h,t}^I$ are weights and query vector of indexer head h and token t, while k_s^I is the key vector of token s shared by all indexer heads, respectively. Tokens with the top-k highest importance are selected for sparse MLA computation. To ensure indexer efficiency, H_{index} and d_{index} are set much smaller than their counterparts in MLA. For example, in DeepSeek-V3.2, the indexer uses $H_{\text{index}} = 64$ and $d_{\text{index}} = 128$, whereas MLA employs $H = 128$ and $d_c = 512$. Moreover, DSA applies low-precision FP8 formats for q^I and k^I to further improve indexer efficiency, and the ReLU is also more efficient than the original softmax. To the best of our knowledge, DSA is currently the only training-time sparse attention method that has been validated on large scale models and realistic complex tasks [18], and it has since been adopted by other models such as the GLM-5 series [20].

2.3 KV Cache Offloading

While sparse attention reduces the computational cost of attention and the amount of KV data accessed, the total size of the KV cache still grows linearly with context length. To support extremely long contexts and large batch sizes under limited GPU HBM capacity, recent works propose offloading the KV cache to host DRAM and loading back, i.e., *recall*, only the selected KVs needed for sparse-attention computation [6, 26, 30, 31, 48, 62, 66, 72]. During inference, KV tensors for individual tokens or token blocks are dynamically evicted and recalled, enabling long-context execution within constrained GPU memory budgets. To reduce the substantial recall overhead caused by the limited bandwidth between host DRAM and GPU, InfiniGen [26] and FreeKV [30] propose to pre-select important tokens and prefetch their KVs, leveraging inter-layer and inter-step similarity, respectively.

Beyond dynamic token-level offloading, static request-level KV cache offloading has also been adopted in general LLM serving systems [8, 19, 39, 59, 64]. Unlike token-level schemes, request-level offloading loads the entire KV cache of a prior context into GPU HBM when a request arrives, and no eviction or recall occurs during inference. KVs are evicted only

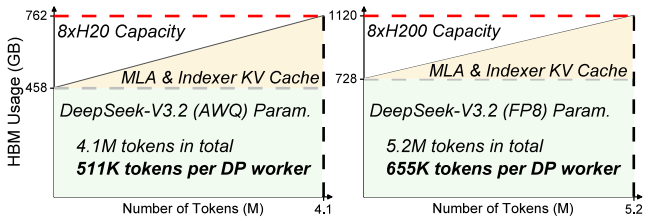


Figure 1: GPU HBM usage of DeepSeek-V3.2 (AWQ) on 8×H20 (96GB) and DeepSeek-V3.2 (FP8) on 8×H200 (141GB). Space for MLA KV cache and indexer K cache increases linearly with the number of tokens.

after the corresponding request completes. These systems focus on improving request-level hit rates to reduce TTFT.

2.4 Motivation and Challenges

Efficiently serving sparse attention models is becoming increasingly important with the emergence of native sparse attention LLMs. However, existing systems struggle to efficiently serve these models under long-context workloads, due to the following limitations and challenges.

Motivation 1: GPU HBM capacity becomes a primary bottleneck for hardware utilization and generation throughput when serving native sparse attention LLMs under long-context workloads. While sparse attention significantly reduces the computation and KV cache access costs, it increases the KV cache size due to storing additional K cache of the indexer, resulting in a steeper linear growth with the context length. Consequently, for long-context requests, GPU HBM capacity constrains the achievable batch size or request concurrency, directly limiting hardware utilization and generation throughput. This bottleneck is further exacerbated by the substantial HBM consumption of MoE parameters and the intrinsic inefficiencies of current sparse attention kernels.

Figure 1 shows the GPU HBM usage of the DeepSeek-V3.2 model as the number of tokens increases. The left plot presents the 4-bit quantized model using AWQ [29] on 8 NVIDIA H20 (96GB) GPUs, and the right plot shows the original FP8 model on 8 NVIDIA H200 (141GB) GPUs. Model parameters are sharded across GPUs using tensor parallelism (TP8). In both configurations, parameters consume more than 60% of the available GPU HBM, leaving room for only ∼4.1M and ∼5.2M tokens, respectively. Although these numbers appear sufficient, the KV cache of MLA cannot be partitioned via tensor parallelism. Instead, data parallelism (DP) is typically used for the MLA module; as a result, each worker can support only ∼511K tokens (AWQ) and ∼655K tokens. In practice, the space for KV cache is even smaller due to runtime overheads such as CUDA graph metadata. For DeepSeek-V3.2 (AWQ) running on 8×H20 GPUs with SGLang configured with TP8 and DP8 for MLA [43], we observe that each DP

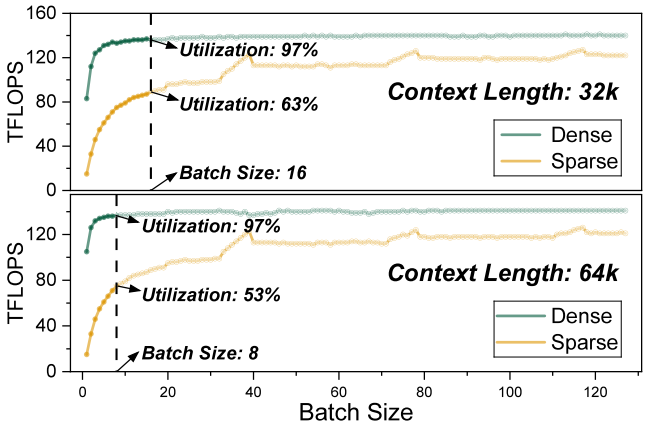


Figure 2: Hardware utilization in TFLOPS of the sparse and dense FlashMLA kernels across different batch sizes.

worker can accommodate at most 380K tokens. This implies that, for requests with ∼100K tokens, each DP worker can serve only 3–4 concurrent requests, severely limiting hardware utilization and generation throughput.

Intrinsic inefficiencies in sparse attention kernels exacerbate the under-utilization problems. Figure 2 shows the hardware utilization (TFLOPS) of sparse and dense FlashMLA [27] decoding kernels on NVIDIA H20, across varying batch sizes and context lengths. Benefiting from high arithmetic density, dense MLA kernels can fully exploit the hardware even at small batch sizes, where a batch size of 8 is adequate for 64K contexts. However, due to non-continuous memory access of the sparse attention kernel, its utilization is much lower than the dense kernel, requiring much larger batch sizes to saturate the hardware. Therefore, the limitation from HBM capacity is more severe for sparse kernels and models. Considering the full DeepSeek-V3.2 (AWQ) model deployed on 8×H20 GPUs, the capacity of 511K tokens can accommodate up to 8 requests of 64K tokens, and the corresponding kernels suffer from under-utilization issues (only 53% utilization), as marked in Figure 2.

As described in Section 2.3, KV cache offloading has been widely studied for long-context inference with limited GPU HBM. However, none of them achieves efficient long-context serving for native sparse attention models in realistic scenarios. We summarize the characteristics of representative KV cache offloading systems in Table 1. First, static request-level offloading such as Strata requires the entire KV cache of requests to be stored in GPU HBM during inference, thus cannot reduce the runtime HBM usage or improve concurrency. While ArkVale, InfiniGen, FreeKV and CLO support dynamic token-level offloading, they are not serving systems and do not support techniques such as continuous batching [63] or multi-GPU inference.

While SparseServe supports serving scenarios with dynamic offloading, its cache management is inefficient and

Table 1: Comparison of representative KV cache offloading systems.

	Strata [59]	ArkVale [6]	InfiniGen [26]	FreeKV [30]	CLO [62]	SparseServe [72]	ECHO
Category	Static	Dynamic	Dynamic	Dynamic	Dynamic	Dynamic	Dynamic
Serving Framework	✔	✘	✘	✘	✘	✔	✔
Graph Execution	✘	✘	✘	✘	✘	✘	✔
KV Prefetch	✘	✘	Lossy	Lossy	Lossy	✘	Lossless
Worker Scale	Multi-GPU	Single-GPU	Single-GPU	Single-GPU	Single-GPU	Single-GPU	Multi-GPU
Model Scale	70B Dense	7B Dense	13B Dense	14B Dense	14B Dense	8B Dense	671B MoE

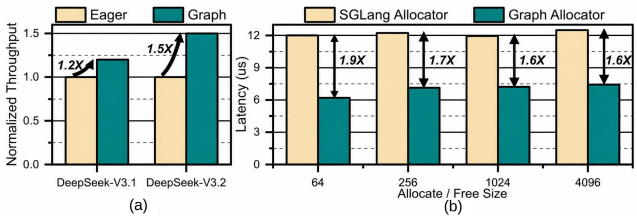


Figure 3: (a) Normalized generation throughput of DeepSeek-V3.1 (dense attention) and DeepSeek-V3.2 (sparse attention) under eager and graph execution. (b) Allocation and free latency of the SGLang allocator and our graph-friendly allocator across different sizes.

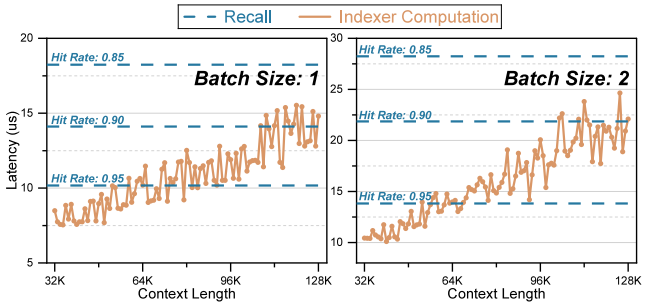


Figure 4: Latency of indexer computation across different context lengths, and latency of recall across different hit rates, profiled using DeepSeek-V3.2 on NVIDIA H20 GPU.

cannot support *graph execution*, which captures execution flow as a deterministic graph and eliminates kernel launch overhead. Graph execution is almost necessary in practical serving frameworks to eliminate significant kernel launch overhead at the decoding stage [54, 55]. The effect of graph execution is even more pronounced for native sparse attention LLMs with more operators and kernels. Figure 3(a) shows the normalized generation throughput for DeepSeek-V3.1 with dense attention and DeepSeek-V3.2 with sparse attention, using SGLang with graph execution (CUDA graph) and without (Eager). As shown, for DeepSeek-V3.1, graph execution improves throughput by 1.2×, while the improvement for DeepSeek-V3.2 is 1.5×.

Furthermore, for dynamic offloading where eviction and recall can happen at each decoding step and model layer, cache management can incur significant overhead, especially at the compute-light decoding stage. Graph execution can also reduce the management overhead, however, it requires the captured graph completely running on GPU, without any CPU control or synchronization. Cache managers and allocators in existing serving systems typically rely on dynamic tensor slicing or concatenation, which cannot be captured by graph execution and can introduce nontrivial overhead. To quantify this cost, we compare the allocation and free latency of the SGLang allocator with the graph-friendly allocator used by ECHO (Section 4), using a KV pool of 300K tokens. As shown in Figure 3(b), one allocation followed by one free takes about 12 μs with the SGLang allocator, whereas the graph-friendly allocator reduces this cost to 6–8 μs, achieving 1.6–1.9× lower latency. Moreover, the SGLang allocator relies on dynamic CPU-side control, which prevents the decoding path from being captured as a graph. Its impact therefore extends beyond the allocator's own management overhead, eliminating the graph execution benefits for end-to-end decoding shown in Figure 3(a). An alternative is piecewise graph execution [55], which runs dynamic operations eagerly while capturing the static computation into graph pieces. However, since eviction and recall occur at every layer in each decoding step, the graph would break at all layers, reintroducing frequent CPU control and kernel launches at the compute-light decoding stage. In contrast, ECHO keeps the entire decoding path within a single full graph, eliminating these per-layer overheads. All existing offloading systems in Table 1 cannot support graph-friendly cache management, incurring significant management and kernel launch overhead.

In summary, *building an efficient, practically usable dynamic KV cache offloading serving system demands graph-friendly cache management.*

Motivation 2: Recall overhead can be substantial, while the indexer computation provides an opportunity for prefetching. Compared with GPU HBM bandwidth, host–GPU interconnect bandwidth is typically one to two orders of magnitude lower. Consequently, KV recall introduces significant overhead and degrades inference efficiency, as all selected tokens' KV tensors must be transferred to the GPU before the subsequent attention computation. To mitigate

this recall overhead, InfiniGen, FreeKV, and CLO propose pre-selection and prefetching of important tokens. However, these approaches rely on inter-layer or inter-step similarity to approximate attention scores, which introduces accuracy degradation on complex tasks due to imperfect prediction.

Prefetching without compromising accuracy appears infeasible due to the sequential dependence among indexer, recall, and attention. However, a key observation that enables breaking this sequential constraint is that indexer computation has complexity $O(L^2)$, whereas recall requires only $O(Lk)$. Specifically for decoding, the indexer must access the K cache and compute index scores for all L preceding tokens, whereas recall requires loading the KV cache for at most k selected tokens, where k is a fixed constant (e.g., 2048) and L grows with context length. In practice, offloading systems maintain a cache for KV of selected tokens on GPU, thus the actual number of recalled tokens is typically less than k.

We present the latency of indexer computation across varying context lengths and the latency of MLA KV recall across varying hit rates during decoding in Figure 4, profiled on NVIDIA H20 GPU which connects with the host via 64GB/s PCIe Gen5. As the context length and hit rate increase, the latency of indexer computation can eventually exceed that of recall, for example, when the context approaches 100K tokens and the hit rate reaches 90%. Recall overhead can be further reduced when using higher-bandwidth interconnects such as NVLink or SXM, which provide 600–900 GB/s of throughput [35]. Because indexer computation and recall contend for distinct hardware resources (GPU compute versus PCIe or other interconnect bandwidth), the recall latency can, in principle, be partially or even fully overlapped through prefetching performed concurrently with indexer computation.

However, similar to other sparse attention methods, DSA selects tokens with top-k highest scores, which can only be determined after computing the index scores for all previous tokens. Consequently, the recall can only be launched after the indexer computation.

To summarize, the second challenge is that *the top-k selection hinders prefetching during indexer computation, leaving PCIe bandwidth idle and preventing the recall latency from being overlapped with indexing*.

To overcome all these challenges, we present ECHO, an efficient LLM serving system with host-side KV cache offloading. To resolve Challenge 1, ECHO incorporates a graph-friendly cache manager that uses tensor-based metadata and parallel in-graph update operations, enabling graph execution while minimizing cache management overhead. To resolve Challenge 2, ECHO introduces lossless intra-query and inter-query prefetching, leveraging numerical predictability of index scores and deeply fused pipelining kernels to enable prefetching and overlap the recall overhead.

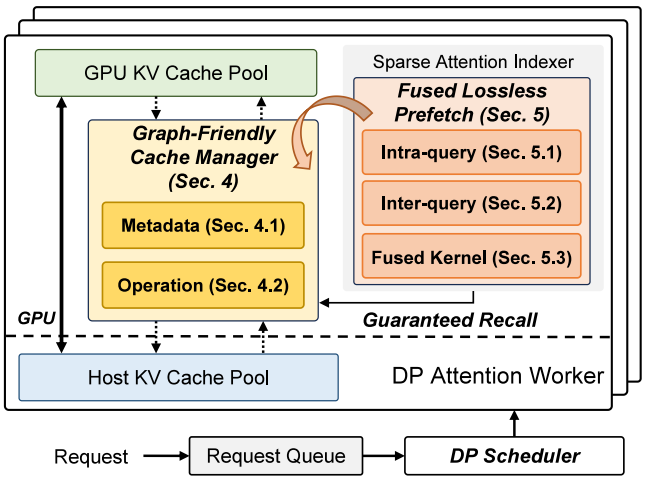


Figure 5: Overview of the ECHO design.

3 Overview of ECHO

The overview of ECHO is presented in Figure 5. As shown, each DP attention worker maintains KV cache pools on GPU and host memory, where the host pool can be much larger than the GPU pool to support more requests and higher concurrency. The DP scheduler routes requests to the selected DP worker. During inference, the K cache of sparse attention indexer is persistently stored in the GPU pool, while the generated MLA KV cache is stored in both the host and GPU pools, where the GPU pool serves as a cache for selected tokens in sparse attention.

The graph-friendly cache manager maintains metadata for the status of the host and GPU pools, which is updated upon recall or eviction. All metadata is on GPU and all update operations can be captured and replayed for graph execution in decoding, executing completely within the GPU graph and minimizing the management overhead. During indexer computation, ECHO prefetches a subset of important tokens, with prefetching fused into the indexer kernel and software-pipelined to hide its overhead. After the indexer computation, a guaranteed recall is launched for the selected tokens that are not yet in the GPU pool. The token eviction priority is then adjusted based on the index scores from the indexer.

Prefill-Decoding (PD) Disaggregation PD disaggregation is a widely applied deployment paradigm [38, 70] that eliminates interference between prefill and decoding, yielding better SLO attainment and optimization flexibility. ECHO is a general system that supports both PD disaggregation and mixed execution. While mixing prefill and decoding requests achieves better throughput in some scenarios [1, 56], we suggest deploying ECHO and other dynamic offloading systems with PD disaggregation for the following reasons. First, for prefill, the number of selected tokens in sparse attention can be substantial, as the selection of each token is independent

and the total selection is the union set of all prefill tokens. In the worst cases, a prefill request accesses KV cache of all previous tokens, which requires a huge number of tokens in the GPU pool and evicts lots of tokens from other requests, significantly degrading the overall throughput. Second, a substantial amount of KV cache needs to be transferred to the host memory during prefill. While ideally the PCIe bandwidths between host-to-GPU and GPU-to-host are independent, we observed that the transfer during prefill interferes with recall, degrading efficiency.

In the PD disaggregation setting, ECHO disables offloading on prefill instances, because prefill has limited concurrency and utilization issues as long as GPU HBM can accommodate a single long-context request (up to 128K tokens for DeepSeek-V3.2). On decoding instances, ECHO stores the KV cache transferred from prefill instances in host memory, and then launches decoding to prefetch and recall KV cache of selected tokens. This ensures that only KV cache selected during decoding is transferred to GPU, avoiding unused KV loads and maximizing HBM utilization.

4 Graph-Friendly Cache Manager

4.1 Management Metadata

Besides common metadata such as block tables in existing systems [25], ECHO maintains additional status metadata of KV cache pools for efficient dynamic eviction and recall. All metadata is stored as integer tensors on GPUs and updated in parallel to minimize overhead. Specifically, the metadata includes the following tensors:

- `GPUTokenFree`: A bitmap indicating whether each slot in the GPU pool is currently free.
- `GPUTokenPriority`: Per-slot eviction priority for the GPU pool. Tokens with lower priorities are evicted first; free slots are initialized with priority -1.
- `GPUIndicesBuffer`: A buffer holding GPU pool indices produced by the allocate and free operations.
- `GPUTokenToHost`: A mapping from each GPU pool slot to its corresponding host pool index, initialized to ∞ to denote an empty slot.
- `HostTokenToGPU`: A mapping from each host pool slot to its corresponding GPU pool index, initialized to ∞ to indicate the token is not currently cached on GPU.

All these metadata tensors are statically pre-allocated and fixed-length because graph execution does not allow dynamically created or variable-length tensors.

In existing systems, the KV cache pool is managed in a *per-model* manner, where KV slots are allocated or freed consistently across all model layers. However, dynamic eviction and recall in sparse attention produce divergent cache states across layers, as each layer selects different tokens. Therefore, ECHO manages the GPU pool *per-layer*, with metadata maintained independently for each layer. The host pool remains per-model, as it is unaffected by dynamic eviction and recall.

Storage Overhead of Metadata Let N_H and N_G denote the number of tokens in the host and GPU pools, respectively. The length of `HostTokenToGPU` is N_H, and the lengths of all other tensors are N_G. `GPUTokenFree` stores boolean elements and other tensors store 32-bit integers. Therefore, the storage cost of all metadata is $4N_H + 13N_G$ bytes. Given $N_H = 2$M and $N_G = 200$K in our setting, the metadata size is about 10MB per layer, totaling 610MB for DeepSeek-V3.2.

4.2 Parallel In-Graph Update Operations

The cache manager updates the status of KV cache pools and metadata using three basic operations: *allocate*, *free*, and *recall*. Compared to the per-model management, which requires only one update per model forward pass, the per-layer management requires updating for each model layer. To minimize management costs, ECHO introduces parallel in-graph update operations. All operations are processed efficiently and entirely in parallel on GPU, and are captured and executed in the graph during decoding, eliminating host interference and minimizing overhead.

Allocate The allocate operation is used to allocate slots from the GPU pool, which is required when (i) accommodating newly-generated KV cache and (ii) preparing slots for tokens to be recalled. It first calls *free* to ensure there are enough free slots. Allocation in existing systems introduces variable-length tensors through dynamic tensor slicing [69], making it incompatible with graph execution. ECHO designs an efficient, parallel allocation operation, addressing this limitation. As shown in Figure 6, the GPU threads read the `GPUTokenFree` in parallel, and threads with free slots launch `atomicAdd` to a global counter. The return value of `atomicAdd` is the old value of the counter, determining which threads should allocate their slots. For the example in Figure 6, slots for thread 1 and thread 5 are allocated. Correspondingly, the metadata tensors are updated by: (i) incrementing the priorities of the newly allocated tokens, and (ii) establishing a mutual mapping between the GPU slots $(1,4)$ and the host slots $(3,4)$. Finally, the indices of the allocated GPU slots are output to a buffer for later use. The allocate operation is fully parallel and processed entirely on the GPU, achieving compatibility with graph execution by avoiding the introduction of variable-length tensors.

Free The *free* operation releases GPU pool slots when the available space is not adequate, providing room for: (i) newly-generated KV cache, and (ii) the KV cache of tokens being

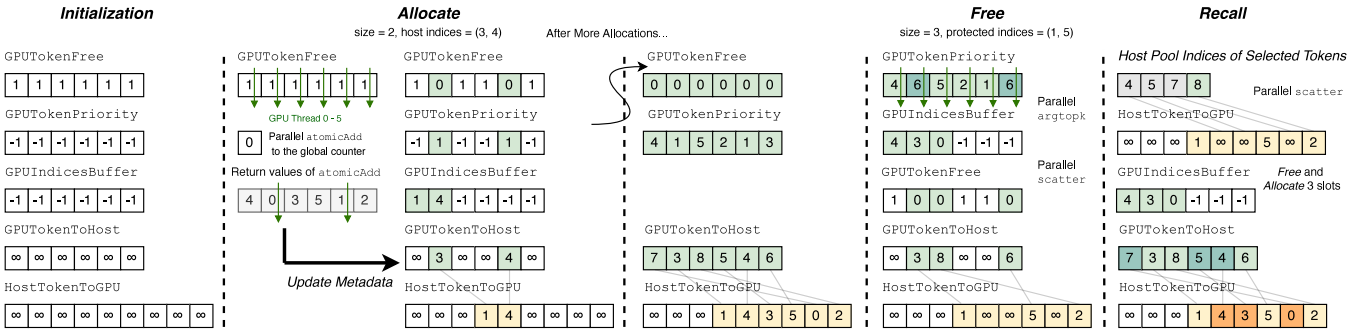


Figure 6: The metadata and update operations in the graph-friendly cache manager of ECHO.

recalled from the host pool. Tokens with the lowest priorities are evicted first, forming an LRU-like eviction policy, and eviction requires only updating the metadata. Notably, there is no need to transfer KV cache to the host pool during eviction, because the KV cache of all tokens has already been backed up to the host pool during generation.

Similar to allocation, *free* in existing systems relies on dynamic tensor concatenation, which is inefficient and not compatible with graph execution. To overcome this limitation, ECHO designs an efficient parallel *free* operation. As shown in Figure 6, it first increments the priorities of *protected* GPU slots to avoid releasing those slots. Specifically, the selected tokens of the sparse indexer are *protected* to prevent them from being evicted. Following the protection phase, a parallel `argtopk` operation is launched to obtain the indices of the GPU slots with the lowest priorities. The result indices are output to the `GPUIndicesBuffer`, thereby avoiding the creation of dynamic variable-length tensors that conflict with graph execution. A parallel `scatter` is then used to update `GPUTokenFree` and `GPUTokenToHost` using the output indices. `HostTokenToGPU` is also updated accordingly, where ∞ indicates that the corresponding token in the host pool is not present in the GPU pool. Tokens in the host pool are only freed upon request completion, which is managed by the per-model block tables.

Recall As shown in Figure 6, the cache manager obtains the host pool indices of the selected tokens from the sparse attention indexer, and uses a parallel `scatter` on `HostTokenToGPU` to identify which tokens are missing from the GPU pool. It then calls *free* and *allocate* to prepare GPU pool slots for the recalled tokens. Using the allocated GPU slot indices in `GPUIndicesBuffer` and the host pool indices of the missing tokens, the cache manager (i) updates `GPUTokenToHost` and `HostTokenToGPU` accordingly, and (ii) transfers the corresponding KV cache from the host pool to the GPU pool. The KV transfer is performed by accessing host memory directly in GPU kernels via unified virtual memory [59,72], which eliminates host-side interference, improves PCIe bandwidth utilization, and preserves graph execution.

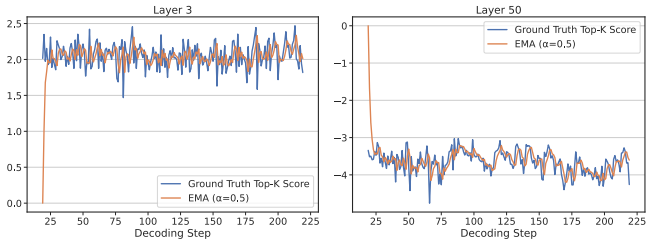


Figure 7: The 2048-th highest index scores in layers 3 and 50 of DeepSeek-V3.2 and prediction using EMA with $\alpha = 0.5$ during decoding.

4.3 Implementation

The implementation of ECHO is based on SGLang [69]. The cache manager utilizes PyTorch tensors [4] to maintain metadata, and the efficient parallel update operations are implemented with custom Triton [52] kernels. While management overhead can be further reduced through specialized CUDA kernels on NVIDIA GPUs, the Triton implementation offers better compatibility, enabling efficient dynamic offloading across a wider range of hardware backends.

5 Fused Lossless Prefetching

5.1 Intra-query Prefetching for Decoding

As discussed in Section 2.4, the top-k selection hinders prefetching during indexer computation. In contrast, top-p selection, which has been well studied in training-free sparse attention, allows a token to be selected as soon as its score exceeds a threshold p [28, 71]. Under a top-p scheme, during indexer computation, any token whose score surpasses p can be recalled immediately, enabling *intra-query prefetching*. Therefore, **prefetching becomes feasible if the top-k selection can be reformulated to top-p selection**.

A key numerical property we identify is that the k-th highest score at the current decoding step exhibits strong predictability from historical indexer scores. In Figure 7, we present the k-th ($k = 2048$) highest score of layers 3 and 50 in DeepSeek-

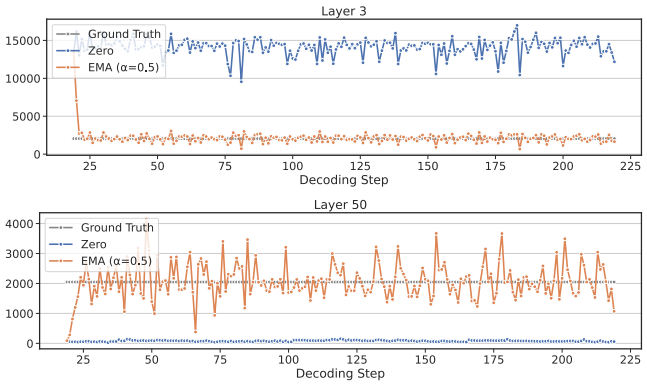


Figure 8: The number of candidate tokens for prefetching in layers 3 and 50 of DeepSeek-V3.2. The blue line uses 0 as the prediction, while the yellow line uses EMA with $\alpha = 0.5$.

V3.2 during decoding, using a 20K-token request from the ShareGPT dataset [42]. We predict the k-th highest score using exponential moving average (EMA). Specifically, let s_t denote the k-th highest score at step t. The k-th highest score at step $t+1$ is predicted with $\hat{s}_{t+1} = \alpha \cdot \hat{s}_t + (1-\alpha) \cdot s_t$, where $\alpha \in (0,1)$ is the smoothing factor. As shown in Figure 7, EMA can predict the k-th highest score of the next step accurately. Therefore, leveraging this prediction, ECHO can prefetch tokens whose scores exceed the predicted threshold during indexer computation, enabling efficient intra-query prefetching without violating top-k semantics.

In Figure 8, we present the number of tokens whose scores are larger than the predicted value, i.e., the candidate tokens for prefetching. As shown, the number of candidate tokens is close to $k = 2048$ at most decoding steps, indicating the effectiveness of our EMA prediction.

At some decoding steps, the number of candidate tokens exceeds 2048 as the predicted threshold may underestimate the true 2048-th highest score. Since prefetching too many tokens can slow down indexer computation, we set an upper bound on the number of prefetched tokens and enforce it using a global counter, as detailed in Section 5.3.

5.2 Inter-query Prefetching for Prefill

Compared to decoding, prefill natively offers the opportunity for *inter-query* prefetching. During prefill, the indexer partitions the input Q into multiple blocks and processes them sequentially. Since prefill sequences are typically long and span many Q blocks, once the scores for Q block i are computed, ECHO can prefetch the selected tokens for Q block i concurrently with the score computation for Q block $i+1$.

However, we find that obtaining the exact top-k for selection during indexer computation can significantly degrade indexer performance. Therefore, **ECHO uses an approximate top-k to obtain a subset of top-k tokens for prefetching**. Specifically, our approximate top-k builds on the radix select

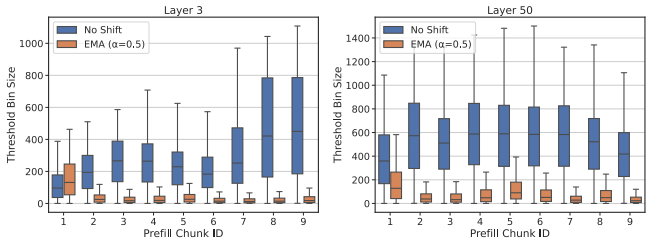


Figure 9: Threshold bin sizes of prefill chunks when computing approximate top-k with and without score shifts, profiled using DeepSeek-V3.2. The boxes and whiskers show the distribution of threshold bin sizes for tokens in the corresponding prefill chunk.

top-k algorithm [46], which constructs histograms of scores and filters higher scores in multiple rounds. Instead of applying all the filter rounds, we only build a coarse histogram and filter *by one round* to get a subset of top-k tokens with highest scores. The scores are mapped to the histogram with 256 bins based on their most significant 8 bits, and tokens within the largest bins are candidates for prefetching. Denote the size of bins as $\mathcal{S}_0, \mathcal{S}_1, ..., \mathcal{S}_{255}$. Selecting the highest j bins implies that $\mathcal{S} = \mathcal{S}_{255-j+1} + ... + \mathcal{S}_{255} \leq k$ and $\mathcal{S}_{255-j} + ... + \mathcal{S}_{255} > k$. The bin $255-j$ is referred to as the *threshold bin*, and its size determines the lower bound of the number of selected tokens, since $\mathcal{S} = \mathcal{S}_{255-j+1} + ... + \mathcal{S}_{255} > k - \mathcal{S}_{255-j}$.

Therefore, to approximate top-k as accurately as possible, we need to minimize the size of the threshold bin. Similar to intra-query prefetching, we use the EMA of the k-th highest score from the ending tokens of the previous prefill chunk to predict the threshold for the current chunk. During histogram construction, scores are shifted by subtracting the predicted k-th highest score to minimize the size of the threshold bin. As shown in Figure 9, shifting scores with the EMA prediction significantly reduces the size of the threshold bin, enhancing the accuracy of the top-k approximation.

5.3 Fused Prefetching Kernels with Pipelining

To hide overhead and benefit from prefetching, ECHO fuses prefetching operations into indexer computation kernels, leveraging *warp specialization* and *software pipelining*.

Warp Specialization Warp specialization is an advanced kernel optimization technique for modern GPUs [5]. Traditionally, GPUs follow the Single-Instruction Multiple-Thread (SIMT) programming model, where all threads within a *warp* execute the same instructions. However, modern GPUs increasingly feature specialized hardware units such as tensor cores and tensor memory accelerators (TMAs), which the SIMT model alone cannot fully exploit. Warp specialization addresses this by assigning distinct roles to different warp groups within a Cooperative Thread Array (CTA) [3]. For ex-

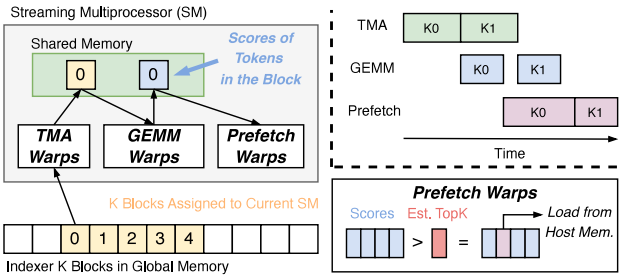


Figure 10: Data flow and timeline of the fused intra-query prefetching kernel for decoding.

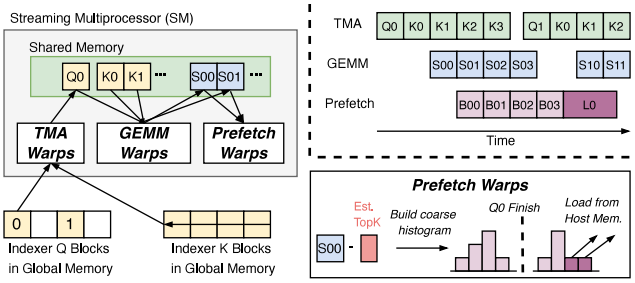


Figure 11: Data flow and timeline of the fused inter-query prefetching kernel for prefill.

ample, some warps load data from global memory via TMA, while others focus on computation with tensor cores.

Software Pipeline To further improve hardware utilization, asynchronous barriers allow the execution of different warp groups to overlap, forming a multi-stage software pipeline [5]. Software pipelining has been widely applied to overlap matrix multiplication with softmax or KV cache loading in attention computation [45, 61].

The fused prefetching kernels in ECHO are built based on the indexer computation kernels from DeepGEMM [12]. Specifically, the original kernel employs a two-stage pipeline, with producer TMA warps loading data from the global memory to shared memory, and consumer GEMM warps computing index scores using data in shared memory.

Figure 10 illustrates the fused intra-query prefetching kernel for decoding. Since the number of query tokens can be small during decoding, the indexer computation is partitioned by splitting the K cache of indexer. Each streaming multiprocessor (SM) computes index scores between a query vector and a part of indexer K cache. The query vector is persistent in shared memory, while the TMA warps iteratively load indexer K blocks from global memory to shared memory. The GEMM warps consume the K blocks, compute index scores and output to the shared memory. The scores are then consumed by the prefetch warps, which compare them with the estimated top-k score and identify tokens that can be prefetched. As shown, the kernel employs a three-stage pipeline of TMA, GEMM and prefetch. To prevent prefetching from blocking indexer computation, we allocate more pipeline stages to prefetching and use a global counter to control the number of prefetched tokens. The maximum number is proportional to context length, because indexer computation time is also proportional to context length.

The fused inter-query prefetching kernel for prefill is illustrated in Figure 11. Different from decoding, the indexer computation for prefill is partitioned by splitting on query. Each SM computes index scores between assigned Q blocks and all the corresponding K blocks. The TMA warps load Q blocks in the outer loop and K blocks in the inner loop, respectively. The GEMM warps output score blocks to shared memory, which are consumed by the prefetch warps. Scores are shifted using the estimated top-k scores as offsets, and scanned to build a histogram. Once computation of the current Q block is finished, the prefetch warps consult the histogram and load tokens with scores above the threshold. Meanwhile, TMA and GEMM warps continue computing scores for the next Q block, forming a software pipeline that overlaps computation and prefetching.

6 Evaluation

6.1 Setup

Testbed All experiments are conducted on a single node equipped with 8 NVIDIA H20 (96GB) GPUs connected to the host via 64GB/s PCIe Gen5. The node features a 224-core Intel Xeon Platinum 8480+ CPU with 1.5TB DRAM.

Baselines and Configurations We compare ECHO with the state-of-the-art LLM serving systems, including vLLM (v0.11.1) and SGLang (v0.5.4), with CUDA 12.9 and PyTorch 2.8. Both ECHO and SGLang are deployed using data parallelism (DP8) for attention and tensor parallelism (TP8) for MoE. vLLM is deployed using TP8 for both attention and MoE because vLLM does not support the DP+TP combination, and using expert parallelism (EP) for MoE incurs substantial GPU HBM costs and fails to launch on our 8×H20 node. We set the size of chunked prefill to 2048 for all the frameworks. CUDA graph during decoding is enabled for all the frameworks. The host KV cache pool in ECHO is set to 1.8M tokens, consuming about 1000GB of host memory.

Workloads Since the original FP8 DeepSeek-V3.2 cannot fit in the HBM of 8×H20, we evaluate using the 4-bit quantized DeepSeek-V3.2-Exp model [40], with the indexer and layers 0, 1, 2 and 60 left unquantized to preserve accuracy. For the long-context workload, we sample a subset of 318 requests with 80K–100K tokens from InfiniteBench [67], amounting

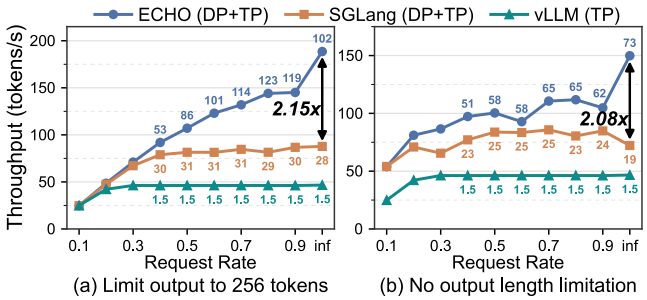


Figure 12: Generation throughput on InfiniteBench under varying request rates on 8×H20, with each system configured to use all available GPU HBM. Annotated numbers denote the effective batch size.

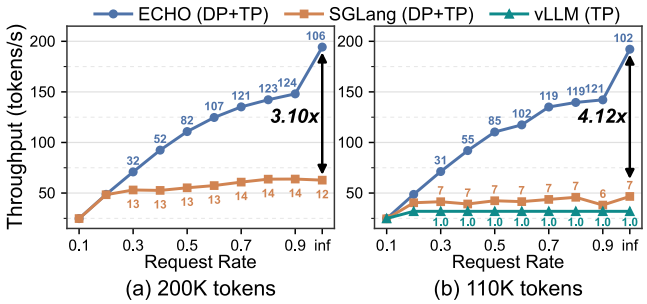


Figure 13: Generation throughput on InfiniteBench under constrained GPU HBM capacity, with output length limited to 256 tokens. Annotated numbers denote effective batch size.

to about 26M input tokens in total. For the short-context workload, we sample requests from the ShareGPT [42] dataset.

6.2 Token Generation Throughput

In this section, we evaluate the token generation throughput in a PD disaggregated setting. The 8×H20 node is launched as a decoding instance, and we pre-compute the KV cache of requests, which can be directly loaded by the decoding instance during inference. SGLang requires a pseudo prefill instance to work properly, incurring some GPU memory overhead.

Maximum Throughput We first evaluate the maximum throughput that can be achieved on the 8×H20 node by allocating all available GPU HBM. We set a fixed output length of 256 tokens for all requests to reduce variance from long-tail generations and ensure that all DP workers are saturated. As shown in Figure 12(a), ECHO achieves comparable generation throughput with SGLang and vLLM at low request rates, and scales more effectively as request rates increase. At `Inf` request rate, where all requests are sent to the serving systems at the same time, ECHO achieves up to 2.15× and 4.1× throughput improvements against SGLang and vLLM, respectively. The throughput improvement achieved by ECHO is

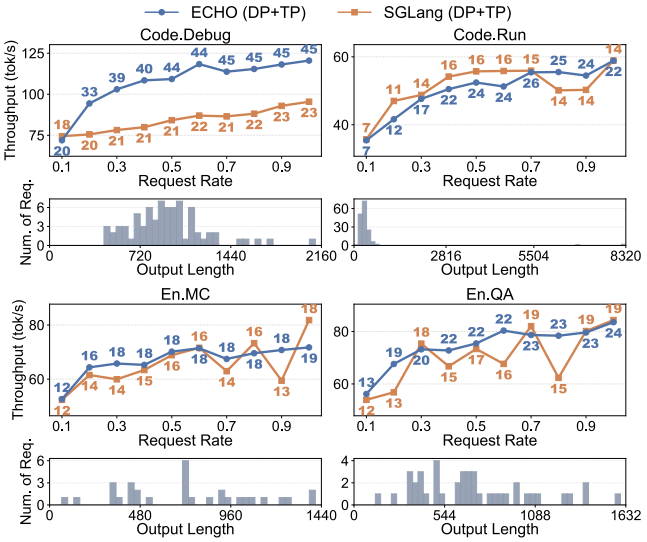


Figure 14: Per-task generation throughput on InfiniteBench without capping output length. The histograms below each task show the corresponding output-length distribution.

enabled by offloading the KV cache to the host pool, which yields larger effective batch sizes, i.e., the average number of active requests processed per decoding step, and higher hardware utilization. On the 8×H20 node, SGLang's 325K-token GPU KV pool supports only 3 to 4 concurrent InfiniteBench requests per GPU, yielding an effective batch size of about 30 across DP workers. vLLM's TP deployment further limits its total KV pool across 8 GPUs to 180K tokens due to duplicated MLA KV cache, reducing its effective batch size to about 1.5. However, vLLM partly offsets this limitation because TP deployment lowers per-step latency, so its throughput does not decrease in proportion to its effective batch size. In contrast, ECHO's host KV cache pool can accommodate 1.8M tokens and sustains much larger effective batch sizes.

Throughput without Output Length Limit Figure 12(b) reports generation throughput without imposing a fixed output-length cap. Compared with Figure 12(a), longer generations improve DP worker utilization at low request rates, increasing the throughput of both ECHO and SGLang. At higher request rates, however, long-tail generations introduce load imbalance across DP workers, leaving some workers idle and reducing the throughput of both systems. In contrast, vLLM uses pure TP and is not affected by DP load imbalance. Moreover, for long-tail requests, end-to-end throughput is dominated by inter-token latency, thus the higher ITL of ECHO narrows its throughput advantage over SGLang.

Throughput with Constrained GPU HBM Usage We further evaluate the generation throughput when the constraints on GPU HBM capacity are more severe, which can arise from

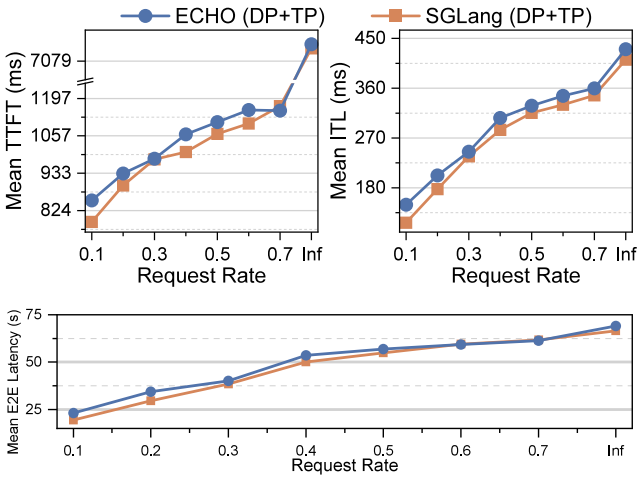


Figure 15: Mean TTFT, ITL, and end-to-end latency of ECHO and SGLang on the ShareGPT dataset.

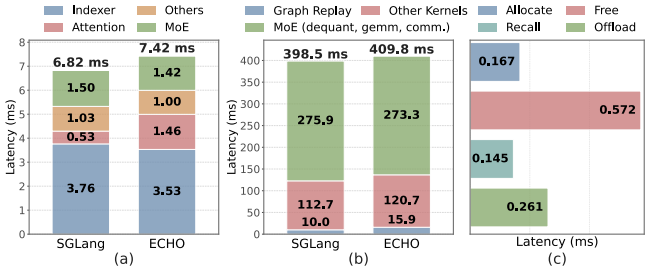


Figure 16: Latency breakdown of ECHO and SGLang. (a) Per-layer prefill latency. (b) All-layer decoding latency, decomposed into graph replay and kernel execution. (c) All-layer offloading overhead of ECHO in one decoding step, including allocate, free, recall, and KV cache offload.

larger MoE parameters. We set a fixed output length of 256 tokens to measure maximum throughput and compare with Figure 12(a). With a 200K-token GPU KV cache pool for both ECHO and SGLang, ECHO's throughput advantage increases to 3.10× at the `Inf` request rate (Figure 13(a)). When the GPU pool is further limited to 110K tokens for all three systems, SGLang and vLLM degrade substantially since their effective batch sizes are bounded by GPU pool size. In contrast, ECHO maintains high throughput by relying on host KV cache pool for a larger effective batch size, achieving up to 4.12× higher throughput than SGLang (Figure 13(b)). ECHO also slightly improves under constrained GPU HBM due to lower management overhead from the smaller GPU pool.

Per-task Throughput of InfiniteBench We further evaluate ECHO on individual InfiniteBench tasks without output length limit. As shown in Figure 14, ECHO improves average output throughput by 27.07% on Code.Debug, 2.83% on En.MC, and 7.11% on En.QA, while slightly underperforming SGLang by 1.74% on Code.Run. The gains are less consistent than in Figure 12 for two reasons. First, the per-task workloads contain relatively few requests (75, 160, 31, and 42, respectively), which limits the available concurrency and prevents them from saturating all DP workers. Second, as shown by the histograms, per-task output lengths are more skewed, causing a few long-tail requests to dominate task runtime and allowing SGLang's lower inter-token latency to offset ECHO's higher effective batch size.

6.3 Inference Latency

In this section, we evaluate inference latency in a mixed PD setting. We randomly sample 100 requests from the ShareGPT [42] dataset as a short-context, light-load workload.

The mean TTFT, ITL, and end-to-end latency of ECHO and SGLang are presented in Figure 15. Overall, ECHO maintains latency close to SGLang while enabling higher long-context throughput. The TTFT changes by at most +7.9% across request rates, mainly due to the overhead of backing up the generated KV cache to host memory during prefill. The ITL overhead is more visible because decoding performs per-layer cache management and KV recall, ranging from +2.7% to +27.8%. This overhead is largest under light load because each decoding step processes fewer concurrent requests, leaving less computation to amortize the fixed cache-management and recall costs; it drops to 2.7–7.7% for request rates of 0.3 and above. As a result, the end-to-end latency overhead is 15.9–19.2% at low request rates of 0.1–0.2, but decreases to at most 7.2% for request rates of 0.3 and above, and remains below 4.6% from 0.5 to `Inf`. The results suggest that although ECHO minimizes the overhead of dynamic KV cache offloading, it still introduces a latency tradeoff and is most beneficial for throughput-oriented long-context serving.

Latency Breakdown Figure 16 presents a detailed latency breakdown of ECHO on the ShareGPT workload under the `Inf` request rate. For prefill, ECHO increases per-layer latency from 6.82 ms to 7.42 ms, an 8.8% increase. The increase mainly appears in the attention component, reflecting the KV cache management and offload overhead introduced by ECHO. For decode, Figure 16(b) reports the all-layer latency, which ECHO increases by 2.8% over SGLang. We decompose the latency into graph replay, MoE and other kernels. MoE accounts for about 70% of the all-layer latency, including dequantization, GEMM, and AllReduce communication. ECHO adds 5.9 ms to graph replay, mainly due to CPU-side launch and metadata overhead. The offloading-specific overhead remains small, as shown in Figure 16(c), where allocation, free, recall, and KV offload take 0.167 ms, 0.572 ms, 0.145 ms, and 0.261 ms across all layers, respectively, totaling 1.15 ms, or only 0.28% of the all-layer decoding latency. Specifically, allocation and free operate on the recalled KV slots, averaging 37

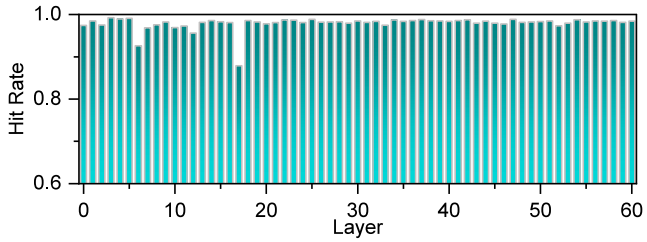


Figure 17: Per-layer GPU pool hit rates of DeepSeek-V3.2 (AWQ) on InfiniteBench, averaged across decoding steps.

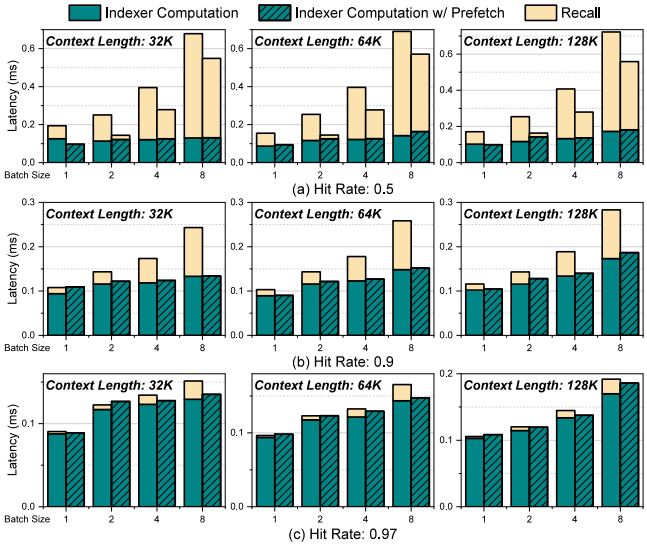


Figure 18: Latency comparison between disabling and enabling intra-query prefetching.

tokens per decoding step, while KV offload writes one newly generated token per active request, averaging 13 tokens per DP worker. These results show that graph-friendly cache management keeps dynamic offloading overhead modest, while high GPU-pool hit rates further reduce recall traffic. The remaining 6.8 ms increase mainly comes from ReduceScatter synchronization across DP workers, due to ECHO's larger concurrency and communication jitter.

6.4 In-Depth Performance Analysis

6.4.1 Hit Rates of the GPU Pool

We collect and dump the hit rates of selected tokens in the GPU pool of each layer when serving the InfiniteBench in Section 6.2, and present the hit rates averaged across decoding steps in Figure 17. As shown, the hit rates for most layers are consistently high, ranging from 0.97 to 0.99. The hit rates of layer 12 and layer 17 are at 0.95 and 0.88, respectively. The high hit rates are consistent with the observation of vertical lines in the attention pattern [23], or similar token selection of adjacent query tokens [30].

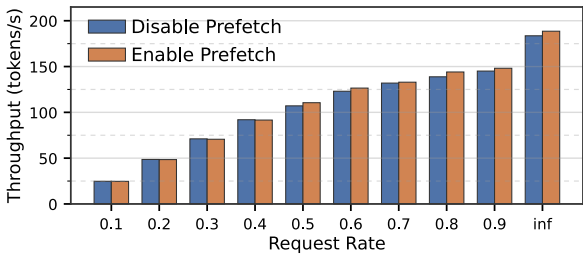


Figure 19: End-to-end generation throughput comparison between disabling and enabling intra-query prefetching.

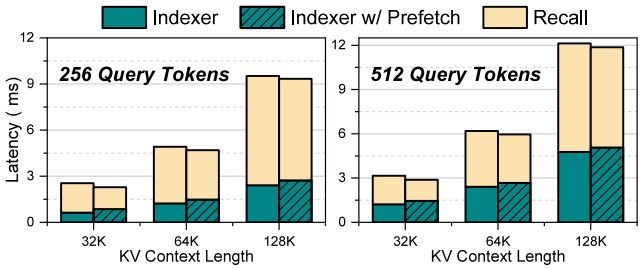


Figure 20: Latency comparison between disabling and enabling inter-query prefetching.

6.4.2 Intra-query Prefetching for Decoding

In Figure 18, we present the latency of indexer computation with and without intra-query prefetching across different context lengths, batch sizes and GPU pool hit rates, along with the latency of their corresponding guaranteed recall. As shown, intra-query prefetching achieves significant total latency reduction for hit rates of 0.5 and 0.9, up to 1.29× and 1.51×, respectively. Specifically, the recall overhead can be almost completely overlapped for hit rate 0.9. The improvements for hit rate 0.97 are modest given that most of selected tokens are already available in the GPU pool and do not require recall.

We further evaluate its end-to-end impact under the same setting as Sec. 6.2. As shown in Figure 19, enabling intra-query prefetching improves generation throughput by up to 4%. This gain is smaller than the microbenchmark improvement since the GPU pool hit rate remains high on InfiniteBench, thus only a small fraction of selected tokens need to be recalled and can benefit from prefetching. In addition, as demonstrated in Figure 16(c), MoE layers account for a large share of end-to-end decoding time, limiting the impact of reducing recall overhead. Using expert parallelism or a more efficient MoE kernel would reduce this bottleneck and make the end-to-end benefit of prefetching more pronounced.

6.4.3 Inter-query Prefetching for Prefill

In Figure 20, we present the latency of indexer computation with and without inter-query prefetching across differ-

ent KV context lengths and batch sizes, using 256 and 512 query tokens, respectively. Compared to decoding, the recall overhead during prefill is much higher because the selection for each query token is independent, resulting in up to *number of query tokens* $\times k$ tokens being selected in total. Consequently, the benefits of prefetching are limited, yielding at most a 1.1$\times$ improvement. We do not include a separate end-to-end throughput ablation for inter-query prefetching, since the PD-disaggregated evaluation in Sec. 6.2 uses precomputed KV cache. Moreover, the microbenchmark suggests that its end-to-end gain would be modest under our conservative assumptions. However, this result underestimates the potential of inter-query prefetching, as our evaluation assumes that the top-k tokens selected by different query tokens are randomly distributed, whereas prior work has observed substantial overlap across query tokens [30]. Exploiting this overlap could reduce redundant recall traffic and further improve inter-query prefetching, which we leave to future work.

7 Discussion

Generalizability to Other Sparse Attention Since DSA is currently the only publicly available native sparse attention validated on large-scale models, ECHO mainly targets DSA and MLA. Nonetheless, the graph-friendly cache manager of ECHO is compatible with both training-time and training-free sparse attention, since it addresses the general system problem of managing dynamic sparsity patterns where selected KV entries must be evicted and recalled during decoding. It is naturally suited to token-wise sparse attention such as DSA, and can also support block-wise sparse attention by changing the metadata granularity and extending the kernels with the KV-head dimension used in GQA-based models. Overall, the cache manager of ECHO should be viewed as a system substrate for dynamic sparse-attention serving rather than a design tied only to DSA, although DSA remains the most compelling target today because of its native model support and strong long-context accuracy. In contrast, intra-query and inter-query prefetching rely on properties of the DSA indexer, including a selection boundary that is numerically predictable across decoding steps and a prefill procedure that exposes a sequence of query blocks whose candidate KV entries can be identified before the exact selection finishes. Other sparse attention mechanisms can benefit from the same prefetching framework when they expose similar early-selection signals, such as threshold-based selection or multi-stage top-k selection. This also suggests an opportunity to co-design training-time or training-free sparse attention with prefetch-friendly selection mechanisms, which we leave to future work.

Throughput-Latency Tradeoff and Practical Deployment As discussed in Section 6.3, offloading introduces latency overhead from cache management and data transfer, making ECHO most attractive for throughput-oriented long-context serving. Recent offloading systems make a throughput-first tradeoff, using dual-batch execution to overlap offload overhead with computation, but may further increase the latency [7]. Although this makes offloading a throughput-first technique today, we believe its additional latency can be substantially narrowed. Specifically, cache-management overhead can be reduced through more dedicated kernel optimizations [44], while transfer overhead will benefit from faster host-GPU interconnects. For example, NVLink-C2C provides 900GB/s on Grace-based systems and is expected to reach 1.8TB/s on Vera-based systems [36]. These trends suggest that offloading can remain practical even for latency-sensitive deployments, while its main advantage today is still in improving throughput under long-context pressure.

Applicability when KV Cache is More Plentiful Although recent models reduce per-token KV-cache cost through quantization and hybrid attention [14,37,51], and newer accelerators provide 200GB or more of HBM, we believe KV-cache capacity will remain an important bottleneck and that offloading will continue to matter. On one hand, long-context inference is becoming increasingly common in agentic workloads, where context lengths of 1M tokens are already entering mainstream use, and the demand for even longer context is likely to continue growing. On the other hand, model parameters, especially MoE parameters, continue to scale toward and beyond the trillion-parameter regime, leaving less HBM available for KV cache under a fixed hardware budget. As a result, improvements in KV cache efficiency and HBM capacity are likely to be offset by simultaneous growth in context length and model size, making efficient offloading and hierarchical KV cache management a persistent system concern.

8 Conclusion

ECHO is a serving system for native sparse attention models under long-context workloads, built around efficient KV cache offloading. ECHO incorporates a graph-friendly cache manager that minimizes management overhead. To reduce recall overhead, ECHO introduces lossless intra-query prefetching based on the numerical predictability of index scores, as well as inter-query prefetching based on the sequential processing of queries. Experimental results demonstrate that ECHO delivers up to 2.1$\times$ generation throughput improvement over the state-of-the-art LLM serving system SGLang on DeepSeek-V3.2 under long-context workloads, breaking the GPU memory capacity bottleneck.

Acknowledgments

This work is sponsored by the National Natural Science Foundation of China (62472273, 62232015).

References

[1] Amey Agrawal, Nitin Kedia, Ashish Panwar, Jayashree Mohan, Nipun Kwatra, Bhargav Gulavani, Alexey Tumanov, and Ramachandran Ramjee. Taming Throughput-Latency tradeoff in LLM inference with Sarathi-Serve. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, pages 117–134, Santa Clara, CA, July 2024. USENIX Association.

[2] Joshua Ainslie, James Lee-Thorp, Michiel de Jong, Yury Zemlyanskiy, Federico Lebron, and Sumit Sanghai. GQA: Training generalized multi-query transformer models from multi-head checkpoints. In Houda Bouamor, Juan Pino, and Kalika Bali, editors, *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pages 4895–4901, Singapore, December 2023. Association for Computational Linguistics.

[3] Michael Andersch, Greg Palmer, Ronny Krashinsky, and Nick Stam. Nvidia hopper architecture in-depth. https://developer.nvidia.com/blog/nvidia-hopper-architecture-in-depth/, 2022.

[4] Jason Ansel, Edward Yang, Horace He, Natalia Gimelshein, Animesh Jain, Michael Voznesensky, Bin Bao, Peter Bell, David Berard, Evgeni Burovski, Geeta Chauhan, Anjali Chourdia, Will Constable, Alban Desmaison, Zachary DeVito, Elias Ellison, Will Feng, Jiong Gong, Michael Gschwind, Brian Hirsh, Sherlock Huang, Kshiteej Kalambarkar, Laurent Kirsch, Michael Lazos, Mario Lezcano, Yanbo Liang, Jason Liang, Yinghai Lu, C. K. Luk, Bert Maher, Yunjie Pan, Christian Puhrsch, Matthias Reso, Mark Saroufim, Marcos Yukio Siraichi, Helen Suk, Shunting Zhang, Michael Suo, Phil Tillet, Xu Zhao, Eikan Wang, Keren Zhou, Richard Zou, Xiaodong Wang, Ajit Mathews, William Wen, Gregory Chanan, Peng Wu, and Soumith Chintala. Pytorch 2: Faster machine learning through dynamic python bytecode transformation and graph compilation. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, ASPLOS '24, pages 929–947, New York, NY, USA, 2024. Association for Computing Machinery.

[5] Hongzheng Chen, Bin Fan, Alexander Collins, Bastian Hagedorn, Evghenii Gaburov, Masahiro Masuda, Matthew Brookhart, Chris Sullivan, Jason Knight, Zhiru Zhang, and Vinod Grover. Tawa: Automatic warp specialization for modern gpus with asynchronous references, 2025.

[6] Renze Chen, Zhuofeng Wang, Beiquan Cao, Tong Wu, Size Zheng, Xiuhong Li, Xuechao Wei, Shengen Yan, Meng Li, and Yun Liang. Arkvale: Efficient generative llm inference with recallable key-value eviction. In A. Globerson, L. Mackey, D. Belgrave, A. Fan, U. Paquet, J. Tomczak, and C. Zhang, editors, *Advances in Neural Information Processing Systems*, volume 37, pages 113134–113155. Curran Associates, Inc., 2024.

[7] Xinhang Chen, Chao Zhang, Jiahuan He, Wei Liu, Jianming Zhang, Wenlong Zhou, Xiao Li, Pai Zeng, Shiyong Li, Yuanpan Qian, Dong Li, and Zhaogeng Li. Ess: An offload-centric latent-cache management architecture for deepseek-v3.2-exp, 2025.

[8] Yihua Cheng, Yuhan Liu, Jiayi Yao, Yuwei An, Xiaokun Chen, Shaoting Feng, Yuyang Huang, Samuel Shen, Kuntai Du, and Junchen Jiang. Lmcache: An efficient kv cache layer for enterprise-scale llm inference, 2025.

[9] Damai Dai, Chengqi Deng, Chenggang Zhao, R. X. Xu, Huazuo Gao, Deli Chen, Jiashi Li, Wangding Zeng, Xingkai Yu, Y. Wu, Zhenda Xie, Y. K. Li, Panpan Huang, Fuli Luo, Chong Ruan, Zhifang Sui, and Wenfeng Liang. Deepseekmoe: Towards ultimate expert specialization in mixture-of-experts language models, 2024.

[10] Sumit Kumar Dam, Choong Seon Hong, Yu Qiao, and Chaoning Zhang. A complete survey on llm-based ai chatbots, 2024.

[11] Google DeepMind. Gemini 2.5: Our most intelligent ai model, March 2025.

[12] DeepSeek-AI. Deepgemm: Clean and efficient fp8 gemm kernels with fine-grained scaling, 2025.

[13] DeepSeek-AI. Deepseek-v3.2-exp: Boosting long-context efficiency with deepseek sparse attention, 2025.

[14] DeepSeek-AI. Deepseek-v4: Towards highly efficient million-token context intelligence. https://huggingface.co/deepseek-ai/DeepSeek-V4-Pro/blob/main/DeepSeek_V4.pdf, 2026. Technical report.

[15] DeepSeek-AI, Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Ruoyu Zhang, Runxin Xu, Qihao Zhu, Shirong Ma, Peiyi Wang, Xiao Bi, Xiaokang Zhang, Xingkai Yu, Yu Wu, Z. F. Wu, Zhibin Gou, Zhihong Shao, Zhuoshu Li, Ziyi Gao, Aixin Liu, Bing Xue, Bingxuan Wang, Bochao Wu, Bei Feng, Chengda Lu, Chenggang Zhao, Chengqi Deng, Chenyu Zhang, Chong Ruan, Damai Dai, Deli Chen, Dongjie Ji, Erhang Li, Fangyun Lin, Fucong Dai, Fuli Luo, Guangbo Hao, Guanting Chen, Guowei Li, H. Zhang, Han Bao, Hanwei Xu, Haocheng Wang, Honghui Ding, Huajian Xin, Huazuo Gao, Hui Qu, Hui Li, Jianzhong Guo, Jiashi Li, Jiawei Wang, Jingchang Chen, Jingyang Yuan, Junjie Qiu, Junlong Li, J. L. Cai, Jiaqi Ni, Jian Liang,

Jin Chen, Kai Dong, Kai Hu, Kaige Gao, Kang Guan, Kexin Huang, Kuai Yu, Lean Wang, Lecong Zhang, Liang Zhao, Litong Wang, Liyue Zhang, Lei Xu, Leyi Xia, Mingchuan Zhang, Minghua Zhang, Minghui Tang, Meng Li, Miaojun Wang, Mingming Li, Ning Tian, Panpan Huang, Peng Zhang, Qiancheng Wang, Qinyu Chen, Qiushi Du, Ruiqi Ge, Ruisong Zhang, Ruizhe Pan, Runji Wang, R. J. Chen, R. L. Jin, Ruyi Chen, Shanghao Lu, Shangyan Zhou, Shanhuang Chen, Shengfeng Ye, Shiyu Wang, Shuiping Yu, Shunfeng Zhou, Shuting Pan, S. S. Li, Shuang Zhou, Shaoqing Wu, Shengfeng Ye, Tao Yun, Tian Pei, Tianyu Sun, T. Wang, Wangding Zeng, Wanjia Zhao, Wen Liu, Wenfeng Liang, Wenjun Gao, Wenqin Yu, Wentao Zhang, W. L. Xiao, Wei An, Xiaodong Liu, Xiaohan Wang, Xiaokang Chen, Xiaotao Nie, Xin Cheng, Xin Liu, Xin Xie, Xingchao Liu, Xinyu Yang, Xinyuan Li, Xuecheng Su, Xuheng Lin, X. Q. Li, Xiangyue Jin, Xiaojin Shen, Xiaosha Chen, Xiaowen Sun, Xiaoxiang Wang, Xinnan Song, Xinyi Zhou, Xianzu Wang, Xinxia Shan, Y. K. Li, Y. Q. Wang, Y. X. Wei, Yang Zhang, Yanhong Xu, Yao Li, Yao Zhao, Yaofeng Sun, Yaohui Wang, Yi Yu, Yichao Zhang, Yifan Shi, Yiliang Xiong, Ying He, Yishi Piao, Yisong Wang, Yixuan Tan, Yiyang Ma, Yiyuan Liu, Yongqiang Guo, Yuan Ou, Yuduan Wang, Yue Gong, Yuheng Zou, Yujia He, Yunfan Xiong, Yuxiang Luo, Yuxiang You, Yuxuan Liu, Yuyang Zhou, Y. X. Zhu, Yanhong Xu, Yanping Huang, Yaohui Li, Yi Zheng, Yuchen Zhu, Yunxian Ma, Ying Tang, Yukun Zha, Yuting Yan, Z. Z. Ren, Zehui Ren, Zhangli Sha, Zhe Fu, Zhean Xu, Zhenda Xie, Zhengyan Zhang, Zhewen Hao, Zhicheng Ma, Zhigang Yan, Zhiyu Wu, Zihui Gu, Zijia Zhu, Zijun Liu, Zilin Li, Ziwei Xie, Ziyang Song, Zizheng Pan, Zhen Huang, Zhipeng Xu, Zhongyu Zhang, and Zhen Zhang. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning, 2025.

[16] DeepSeek-AI, Aixin Liu, Bei Feng, Bin Wang, Bingxuan Wang, Bo Liu, Chenggang Zhao, Chengqi Dengr, Chong Ruan, Damai Dai, Daya Guo, Dejian Yang, Deli Chen, Dongjie Ji, Erhang Li, Fangyun Lin, Fuli Luo, Guangbo Hao, Guanting Chen, Guowei Li, H. Zhang, Hanwei Xu, Hao Yang, Haowei Zhang, Honghui Ding, Huajian Xin, Huazuo Gao, Hui Li, Hui Qu, J. L. Cai, Jian Liang, Jianzhong Guo, Jiaqi Ni, Jiashi Li, Jin Chen, Jingyang Yuan, Junjie Qiu, Junxiao Song, Kai Dong, Kaige Gao, Kang Guan, Lean Wang, Lecong Zhang, Lei Xu, Leyi Xia, Liang Zhao, Liyue Zhang, Meng Li, Miaojun Wang, Mingchuan Zhang, Minghua Zhang, Minghui Tang, Mingming Li, Ning Tian, Panpan Huang, Peiyi Wang, Peng Zhang, Qihao Zhu, Qinyu Chen, Qiushi Du, R. J. Chen, R. L. Jin, Ruiqi Ge, Ruizhe Pan, Runxin Xu, Ruyi Chen, S. S. Li, Shanghao Lu, Shangyan Zhou, Shanhuang Chen, Shaoqing Wu, Shengfeng Ye, Shirong Ma, Shiyu Wang, Shuang Zhou, Shuiping Yu, Shunfeng Zhou, Size Zheng, T. Wang, Tian Pei, Tian Yuan, Tianyu Sun, W. L. Xiao, Wangding Zeng, Wei An, Wen Liu, Wenfeng Liang, Wenjun Gao, Wentao Zhang, X. Q. Li, Xiangyue Jin, Xianzu Wang, Xiao Bi, Xiaodong Liu, Xiaohan Wang, Xiaojin Shen, Xiaokang Chen, Xiaosha Chen, Xiaotao Nie, Xiaowen Sun, Xiaoxiang Wang, Xin Liu, Xin Xie, Xingkai Yu, Xinnan Song, Xinyi Zhou, Xinyu Yang, Xuan Lu, Xuecheng Su, Y. Wu, Y. K. Li, Y. X. Wei, Y. X. Zhu, Yanhong Xu, Yanping Huang, Yao Li, Yao Zhao, Yaofeng Sun, Yaohui Li, Yaohui Wang, Yi Zheng, Yichao Zhang, Yiliang Xiong, Yilong Zhao, Ying He, Ying Tang, Yishi Piao, Yixin Dong, Yixuan Tan, Yiyuan Liu, Yongji Wang, Yongqiang Guo, Yuchen Zhu, Yuduan Wang, Yuheng Zou, Yukun Zha, Yunxian Ma, Yuting Yan, Yuxiang You, Yuxuan Liu, Z. Z. Ren, Zehui Ren, Zhangli Sha, Zhe Fu, Zhen Huang, Zhen Zhang, Zhenda Xie, Zhewen Hao, Zhihong Shao, Zhiniu Wen, Zhipeng Xu, Zhongyu Zhang, Zhuoshu Li, Zihan Wang, Zihui Gu, Zilin Li, and Ziwei Xie. Deepseek-v2: A strong, economical, and efficient mixture-of-experts language model, 2024.

[17] DeepSeek-AI, Aixin Liu, Bei Feng, Bing Xue, Bingxuan Wang, Bochao Wu, Chengda Lu, Chenggang Zhao, Chengqi Deng, Chenyu Zhang, Chong Ruan, Damai Dai, Daya Guo, Dejian Yang, Deli Chen, Dongjie Ji, Erhang Li, Fangyun Lin, Fucong Dai, Fuli Luo, Guangbo Hao, Guanting Chen, Guowei Li, H. Zhang, Han Bao, Hanwei Xu, Haocheng Wang, Haowei Zhang, Honghui Ding, Huajian Xin, Huazuo Gao, Hui Li, Hui Qu, J. L. Cai, Jian Liang, Jianzhong Guo, Jiaqi Ni, Jiashi Li, Jiawei Wang, Jin Chen, Jingchang Chen, Jingyang Yuan, Junjie Qiu, Junlong Li, Junxiao Song, Kai Dong, Kai Hu, Kaige Gao, Kang Guan, Kexin Huang, Kuai Yu, Lean Wang, Lecong Zhang, Lei Xu, Leyi Xia, Liang Zhao, Litong Wang, Liyue Zhang, Meng Li, Miaojun Wang, Mingchuan Zhang, Minghua Zhang, Minghui Tang, Mingming Li, Ning Tian, Panpan Huang, Peiyi Wang, Peng Zhang, Qiancheng Wang, Qihao Zhu, Qinyu Chen, Qiushi Du, R. J. Chen, R. L. Jin, Ruiqi Ge, Ruisong Zhang, Ruizhe Pan, Runji Wang, Runxin Xu, Ruoyu Zhang, Ruyi Chen, S. S. Li, Shanghao Lu, Shangyan Zhou, Shanhuang Chen, Shaoqing Wu, Shengfeng Ye, Shengfeng Ye, Shirong Ma, Shiyu Wang, Shuang Zhou, Shuiping Yu, Shunfeng Zhou, Shuting Pan, T. Wang, Tao Yun, Tian Pei, Tianyu Sun, W. L. Xiao, Wangding Zeng, Wanjia Zhao, Wei An, Wen Liu, Wenfeng Liang, Wenjun Gao, Wenqin Yu, Wentao Zhang, X. Q. Li, Xiangyue Jin, Xianzu Wang, Xiao Bi, Xiaodong Liu, Xiaohan Wang, Xiaojin Shen, Xiaokang Chen, Xiaokang Zhang, Xiaosha Chen, Xiaotao Nie, Xiaowen Sun, Xiaoxiang Wang, Xin Cheng, Xin Liu, Xin Xie, Xingchao Liu, Xingkai Yu, Xinnan Song, Xinxia Shan, Xinyi Zhou, Xinyu Yang,

Xinyuan Li, Xuecheng Su, Xuheng Lin, Y. K. Li, Y. Q. Wang, Y. X. Wei, Y. X. Zhu, Yang Zhang, Yanhong Xu, Yanhong Xu, Yanping Huang, Yao Li, Yao Zhao, Yaofeng Sun, Yaohui Li, Yaohui Wang, Yi Yu, Yi Zheng, Yichao Zhang, Yifan Shi, Yiliang Xiong, Ying He, Ying Tang, Yishi Piao, Yisong Wang, Yixuan Tan, Yiyang Ma, Yiyuan Liu, Yongqiang Guo, Yu Wu, Yuan Ou, Yuchen Zhu, Yuduan Wang, Yue Gong, Yuheng Zou, Yujia He, Yukun Zha, Yunfan Xiong, Yunxian Ma, Yuting Yan, Yuxiang Luo, Yuxiang You, Yuxuan Liu, Yuyang Zhou, Z. F. Wu, Z. Z. Ren, Zehui Ren, Zhangli Sha, Zhe Fu, Zhean Xu, Zhen Huang, Zhen Zhang, Zhenda Xie, Zhengyan Zhang, Zhewen Hao, Zhibin Gou, Zhicheng Ma, Zhigang Yan, Zhihong Shao, Zhipeng Xu, Zhiyu Wu, Zhongyu Zhang, Zhuoshu Li, Zihui Gu, Zijia Zhu, Zijun Liu, Zilin Li, Ziwei Xie, Ziyang Song, Ziyi Gao, and Zizheng Pan. Deepseek-v3 technical report, 2025.

[18] DeepSeek-AI, Aixin Liu, Aoxue Mei, Bangcai Lin, Bing Xue, Bingxuan Wang, Bingzheng Xu, Bochao Wu, Bowei Zhang, Chaofan Lin, Chen Dong, Chengda Lu, Chenggang Zhao, Chengqi Deng, Chenhao Xu, Chong Ruan, Damai Dai, Daya Guo, Dejian Yang, Deli Chen, Erhang Li, Fangqi Zhou, Fangyun Lin, Fucong Dai, Guangbo Hao, Guanting Chen, Guowei Li, H. Zhang, Hanwei Xu, Hao Li, Haofen Liang, Haoran Wei, Haowei Zhang, Haowen Luo, Haozhe Ji, Honghui Ding, Hongxuan Tang, Huanqi Cao, Huazuo Gao, Hui Qu, Hui Zeng, Jialiang Huang, Jiashi Li, Jiaxin Xu, Jiewen Hu, Jingchang Chen, Jingting Xiang, Jingyang Yuan, Jingyuan Cheng, Jinhua Zhu, Jun Ran, Junguang Jiang, Junjie Qiu, Junlong Li, Junxiao Song, Kai Dong, Kaige Gao, Kang Guan, Kexin Huang, Kexing Zhou, Kezhao Huang, Kuai Yu, Lean Wang, Lecong Zhang, Lei Wang, Liang Zhao, Liangsheng Yin, Lihua Guo, Lingxiao Luo, Linwang Ma, Litong Wang, Liyue Zhang, M. S. Di, M. Y Xu, Mingchuan Zhang, Minghua Zhang, Minghui Tang, Mingxu Zhou, Panpan Huang, Peixin Cong, Peiyi Wang, Qiancheng Wang, Qihao Zhu, Qingyang Li, Qinyu Chen, Qiushi Du, Ruiling Xu, Ruiqi Ge, Ruisong Zhang, Ruizhe Pan, Runji Wang, Runqiu Yin, Runxin Xu, Ruomeng Shen, Ruoyu Zhang, S. H. Liu, Shanghao Lu, Shangyan Zhou, Shanhuang Chen, Shaofei Cai, Shaoyuan Chen, Shengding Hu, Shengyu Liu, Shiqiang Hu, Shirong Ma, Shiyu Wang, Shuiping Yu, Shunfeng Zhou, Shuting Pan, Songyang Zhou, Tao Ni, Tao Yun, Tian Pei, Tian Ye, Tianyuan Yue, Wangding Zeng, Wen Liu, Wenfeng Liang, Wenjie Pang, Wenjing Luo, Wenjun Gao, Wentao Zhang, Xi Gao, Xiangwen Wang, Xiao Bi, Xiaodong Liu, Xiaohan Wang, Xiaokang Chen, Xiaokang Zhang, Xiaotao Nie, Xin Cheng, Xin Liu, Xin Xie, Xingchao Liu, Xingkai Yu, Xingyou Li, Xinyu Yang, Xinyuan Li, Xu Chen, Xuecheng Su, Xuehai Pan, Xuheng Lin, Xuwei Fu, Y. Q. Wang, Yang Zhang, Yanhong Xu, Yanru Ma, Yao Li, Yao Li, Yao Zhao, Yaofeng Sun, Yaohui Wang, Yi Qian, Yi Yu, Yichao Zhang, Yifan Ding, Yifan Shi, Yiliang Xiong, Ying He, Ying Zhou, Yinmin Zhong, Yishi Piao, Yisong Wang, Yixiao Chen, Yixuan Tan, Yixuan Wei, Yiyang Ma, Yiyuan Liu, Yonglun Yang, Yongqiang Guo, Yongtong Wu, Yu Wu, Yuan Cheng, Yuan Ou, Yuanfan Xu, Yuduan Wang, Yue Gong, Yuhan Wu, Yuheng Zou, Yukun Li, Yunfan Xiong, Yuxiang Luo, Yuxiang You, Yuxuan Liu, Yuyang Zhou, Z. F. Wu, Z. Z. Ren, Zehua Zhao, Zehui Ren, Zhangli Sha, Zhe Fu, Zhean Xu, Zhenda Xie, Zhengyan Zhang, Zhewen Hao, Zhibin Gou, Zhicheng Ma, Zhigang Yan, Zhihong Shao, Zhixian Huang, Zhiyu Wu, Zhuoshu Li, Zhuping Zhang, Zian Xu, Zihao Wang, Zihui Gu, Zijia Zhu, Zilin Li, Zipeng Zhang, Ziwei Xie, Ziyi Gao, Zizheng Pan, Zongqing Yao, Bei Feng, Hui Li, J. L. Cai, Jiaqi Ni, Lei Xu, Meng Li, Ning Tian, R. J. Chen, R. L. Jin, S. S. Li, Shuang Zhou, Tianyu Sun, X. Q. Li, Xiangyue Jin, Xiaojin Shen, Xiaosha Chen, Xinnan Song, Xinyi Zhou, Y. X. Zhu, Yanping Huang, Yaohui Li, Yi Zheng, Yuchen Zhu, Yunxian Ma, Zhen Huang, Zhipeng Xu, Zhongyu Zhang, Dongjie Ji, Jian Liang, Jianzhong Guo, Jin Chen, Leyi Xia, Miaojun Wang, Mingming Li, Peng Zhang, Ruyi Chen, Shangmian Sun, Shaoqing Wu, Shengfeng Ye, T. Wang, W. L. Xiao, Wei An, Xianzu Wang, Xiaowen Sun, Xiaoxiang Wang, Ying Tang, Yukun Zha, Zekai Zhang, Zhe Ju, Zhen Zhang, and Zihua Qu. Deepseek-v3.2: Pushing the frontier of open large language models, 2025.

[19] Bin Gao, Zhuomin He, Puru Sharma, Qingxuan Kang, Djordje Jevdjic, Junbo Deng, Xingkun Yang, Zhou Yu, and Pengfei Zuo. Cost-Efficient large language model serving for multi-turn conversations with CachedAttention. In *2024 USENIX Annual Technical Conference (USENIX ATC 24)*, pages 111–126, Santa Clara, CA, July 2024. USENIX Association.

[20] GLM-5-Team. Glm-5: from vibe coding to agentic engineering, 2026.

[21] Shengding Hu, Yuge Tu, Xu Han, Chaoqun He, Ganqu Cui, Xiang Long, Zhi Zheng, Yewei Fang, Yuxiang Huang, Weilin Zhao, et al. Minicpm: Unveiling the potential of small language models with scalable training strategies. *arXiv preprint arXiv:2404.06395*, 2024.

[22] Albert Q. Jiang, Alexandre Sablayrolles, Arthur Mensch, Chris Bamford, Devendra Singh Chaplot, Diego de las Casas, Florian Bressand, Gianna Lengyel, Guillaume Lample, Lucile Saulnier, Lélio Renard Lavaud, Marie-Anne Lachaux, Pierre Stock, Teven Le Scao, Thibaut Lavril, Thomas Wang, Timothée Lacroix, and William El Sayed. Mistral 7b, 2023.

[23] Huiqiang Jiang, Yucheng Li, Chengruidong Zhang, Qianhui Wu, Xufang Luo, Surin Ahn, Zhenhua Han, Amir H. Abdi, Dongsheng Li, Chin-Yew Lin, Yuqing Yang, and Lili Qiu. Minference 1.0: Accelerating prefilling for long-context llms via dynamic sparse attention, 2024.

[24] Juyong Jiang, Fan Wang, Jiasi Shen, Sungju Kim, and Sunghun Kim. A survey on large language models for code generation, 2024.

[25] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph Gonzalez, Hao Zhang, and Ion Stoica. Efficient memory management for large language model serving with pagedattention. In *Proceedings of the 29th Symposium on Operating Systems Principles*, SOSP '23, page 611–626, New York, NY, USA, 2023. Association for Computing Machinery.

[26] Wonbeom Lee, Jungi Lee, Junghwan Seo, and Jaewoong Sim. InfiniGen: Efficient generative inference of large language models with dynamic KV cache management. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, pages 155–172, Santa Clara, CA, July 2024. USENIX Association.

[27] Jiashi Li and Shengyu Liu. Flashmla: Efficient multi-head latent attention kernels. https://github.com/deepseek-ai/FlashMLA, 2025.

[28] Chaofan Lin, Jiaming Tang, Shuo Yang, Hanshuo Wang, Tian Tang, Boyu Tian, Ion Stoica, Song Han, and Mingyu Gao. Twilight: Adaptive attention sparsity with hierarchical top-p pruning, 2025.

[29] Ji Lin, Jiaming Tang, Haotian Tang, Shang Yang, Wei-Ming Chen, Wei-Chen Wang, Guangxuan Xiao, Xingyu Dang, Chuang Gan, and Song Han. Awq: Activation-aware weight quantization for llm compression and acceleration, 2024.

[30] Guangda Liu, Chengwei Li, Zhenyu Ning, Minyi Guo, and Jieru Zhao. Freekv: Boosting kv cache retrieval for efficient llm inference, 2025.

[31] Guangda Liu, Chengwei Li, Jieru Zhao, Chenqi Zhang, and Minyi Guo. Clusterkv: Manipulating llm kv cache in semantic space for recallable compression, 2024.

[32] Enzhe Lu, Zhejun Jiang, Jingyuan Liu, Yulun Du, Tao Jiang, Chao Hong, Shaowei Liu, Weiran He, Enming Yuan, Yuzhi Wang, Zhiqi Huang, Huan Yuan, Suting Xu, Xinran Xu, Guokun Lai, Yanru Chen, Huabin Zheng, Junjie Yan, Jianlin Su, Yuxin Wu, Neo Y. Zhang, Zhilin Yang, Xinyu Zhou, Mingxing Zhang, and Jiezhong Qiu. Moba: Mixture of block attention for long-context llms, 2025.

[33] Junyu Luo, Weizhi Zhang, Ye Yuan, Yusheng Zhao, Junwei Yang, Yiyang Gu, Bohan Wu, Binqi Chen, Ziyue Qiao, Qingqing Long, Rongcheng Tu, Xiao Luo, Wei Ju, Zhiping Xiao, Yifan Wang, Meng Xiao, Chenwu Liu, Jingyang Yuan, Shichang Zhang, Yiqiao Jin, Fan Zhang, Xian Wu, Hanqing Zhao, Dacheng Tao, Philip S. Yu, and Ming Zhang. Large language model agent: A survey on methodology, applications and challenges, 2025.

[34] Fanxu Meng, Pingzhi Tang, Xiaojuan Tang, Zengwei Yao, Xing Sun, and Muhan Zhang. Transmla: Multi-head latent attention is all you need, 2025.

[35] NVIDIA. Datasheet of nvidia h100 gpu, 2025.

[36] NVIDIA. Nvidia vera cpu delivers high performance, bandwidth, and efficiency for ai factories. https://developer.nvidia.com/blog/nvidia-vera-cpu-delivers-high-performance-bandwidth-and-efficiency-for-ai-factories/, 2026. NVIDIA Technical Blog.

[37] OpenAI. gpt-oss-120b & gpt-oss-20b model card, 2025.

[38] Pratyush Patel, Esha Choukse, Chaojie Zhang, Aashaka Shah, Inigo Goiri, Saeed Maleki, and Ricardo Bianchini. Splitwise: Efficient generative llm inference using phase splitting. In *ISCA*, June 2024.

[39] Ruoyu Qin, Zheming Li, Weiran He, Jialei Cui, Feng Ren, Mingxing Zhang, Yongwei Wu, Weimin Zheng, and Xinran Xu. Mooncake: Trading more storage for less computation — a KVCache-centric architecture for serving LLM chatbot. In *23rd USENIX Conference on File and Storage Technologies (FAST 25)*, pages 155–170, Santa Clara, CA, February 2025. USENIX Association.

[40] QuantTrio. Deepseek-v3.2-exp-awq. https://huggingface.co/QuantTrio/DeepSeek-V3.2-Exp-AWQ, 2025.

[41] Qwen, :, An Yang, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chengyuan Li, Dayiheng Liu, Fei Huang, Haoran Wei, Huan Lin, Jian Yang, Jianhong Tu, Jianwei Zhang, Jianxin Yang, Jiaxi Yang, Jingren Zhou, Junyang Lin, Kai Dang, Keming Lu, Keqin Bao, Kexin Yang, Le Yu, Mei Li, Mingfeng Xue, Pei Zhang, Qin Zhu, Rui Men, Runji Lin, Tianhao Li, Tianyi Tang, Tingyu Xia, Xingzhang Ren, Xuancheng Ren, Yang Fan, Yang Su, Yichang Zhang, Yu Wan, Yuqiong Liu, Zeyu Cui, Zhenru Zhang, and Zihan Qiu. Qwen2.5 technical report, 2025.

[42] RyokoAI. Sharegpt52k. https://huggingface.co/datasets/RyokoAI/ShareGPT52K, 2023.

[43] SGLang Team. Sglang documentation: Deepseek v3.2 usage, 2025. Accessed: 2025-12-04.

[44] SGLang Team. Hisparse: Turbocharging sparse attention with hierarchical memory. https://www.lmsys.org/blog/2026-04-10-sglang-hisparse/, 2026. Blog post.

[45] Jay Shah, Ganesh Bikshandi, Ying Zhang, Vijay Thakkar, Pradeep Ramani, and Tri Dao. Flashattention-3: Fast and accurate attention with asynchrony and low-precision, 2024.

[46] Anil Shanbhag, Holger Pirk, and Samuel Madden. Efficient top-k query processing on massively parallel hardware. In *Proceedings of the 2018 International Conference on Management of Data*, SIGMOD '18, page 1557–1570, New York, NY, USA, 2018. Association for Computing Machinery.

[47] Noam Shazeer. Fast transformer decoding: One write-head is all you need, 2019.

[48] Hanshi Sun, Li-Wen Chang, Wenlei Bao, Size Zheng, Ningxin Zheng, Xin Liu, Harry Dong, Yuejie Chi, and Beidi Chen. Shadowkv: Kv cache in shadows for high-throughput long-context llm inference, 2025.

[49] Jiaming Tang, Yilong Zhao, Kan Zhu, Guangxuan Xiao, Baris Kasikci, and Song Han. Quest: Query-aware sparsity for efficient long-context llm inference. *ICML*, 2024.

[50] Kimi Team, Yifan Bai, Yiping Bao, Guanduo Chen, Jiahao Chen, Ningxin Chen, Ruijue Chen, Yanru Chen, Yuankun Chen, Yutian Chen, Zhuofu Chen, Jialei Cui, Hao Ding, Mengnan Dong, Angang Du, Chenzhuang Du, Dikang Du, Yulun Du, Yu Fan, Yichen Feng, Kelin Fu, Bofei Gao, Hongcheng Gao, Peizhong Gao, Tong Gao, Xinran Gu, Longyu Guan, Haiqing Guo, Jianhang Guo, Hao Hu, Xiaoru Hao, Tianhong He, Weiran He, Wenyang He, Chao Hong, Yangyang Hu, Zhenxing Hu, Weixiao Huang, Zhiqi Huang, Zihao Huang, Tao Jiang, Zhejun Jiang, Xinyi Jin, Yongsheng Kang, Guokun Lai, Cheng Li, Fang Li, Haoyang Li, Ming Li, Wentao Li, Yanhao Li, Yiwei Li, Zhaowei Li, Zheming Li, Hongzhan Lin, Xiaohan Lin, Zongyu Lin, Chengyin Liu, Chenyu Liu, Hongzhang Liu, Jingyuan Liu, Junqi Liu, Liang Liu, Shaowei Liu, T. Y. Liu, Tianwei Liu, Weizhou Liu, Yangyang Liu, Yibo Liu, Yiping Liu, Yue Liu, Zhengying Liu, Enzhe Lu, Lijun Lu, Shengling Ma, Xinyu Ma, Yingwei Ma, Shaoguang Mao, Jie Mei, Xin Men, Yibo Miao, Siyuan Pan, Yebo Peng, Ruoyu Qin, Bowen Qu, Zeyu Shang, Lidong Shi, Shengyuan Shi, Feifan Song, Jianlin Su, Zhengyuan Su, Xinjie Sun, Flood Sung, Heyi Tang, Jiawen Tao, Qifeng Teng, Chensi Wang, Dinglu Wang, Feng Wang, Haiming Wang, Jianzhou Wang, Jiaxing Wang, Jinhong Wang, Shengjie Wang, Shuyi Wang, Yao Wang, Yejie Wang, Yiqin Wang, Yuxin Wang, Yuzhi Wang, Zhaoji Wang, Zhengtao Wang, Zhexu Wang, Chu Wei, Qianqian Wei, Wenhao Wu, Xingzhe Wu, Yuxin Wu, Chenjun Xiao, Xiaotong Xie, Weimin Xiong, Boyu Xu, Jing Xu, Jinjing Xu, L. H. Xu, Lin Xu, Suting Xu, Weixin Xu, Xinran Xu, Yangchuan Xu, Ziyao Xu, Junjie Yan, Yuzi Yan, Xiaofei Yang, Ying Yang, Zhen Yang, Zhilin Yang, Zonghan Yang, Haotian Yao, Xingcheng Yao, Wenjie Ye, Zhuorui Ye, Bohong Yin, Longhui Yu, Enming Yuan, Hongbang Yuan, Mengjie Yuan, Haobing Zhan, Dehao Zhang, Hao Zhang, Wanlu Zhang, Xiaobin Zhang, Yangkun Zhang, Yizhi Zhang, Yongting Zhang, Yu Zhang, Yutao Zhang, Yutong Zhang, Zheng Zhang, Haotian Zhao, Yikai Zhao, Huabin Zheng, Shaojie Zheng, Jianren Zhou, Xinyu Zhou, Zaida Zhou, Zhen Zhu, Weiyu Zhuang, and Xinxing Zu. Kimi k2: Open agentic intelligence, 2025.

[51] Qwen Team. Qwen3.5: Accelerating productivity with native multimodal agents, February 2026.

[52] Philippe Tillet, H. T. Kung, and David Cox. Triton: an intermediate language and compiler for tiled neural network computations. In *Proceedings of the 3rd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*, MAPL 2019, page 10–19, New York, NY, USA, 2019. Association for Computing Machinery.

[53] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need, 2023.

[54] vLLM Team. vllm documentation: Acl graphs, 2025.

[55] vLLM Team. vllm documentation: Cuda graphs, 2025.

[56] Chao Wang, Pengfei Zuo, Zhangyu Chen, Yunkai Liang, Zhou Yu, and Ming-Chang Yang. Prefill-decode aggregation or disaggregation? unifying both for goodput-optimized llm serving, 2025.

[57] xAI. Grok 3 beta — the age of reasoning agents, February 2025.

[58] Guangxuan Xiao, Yuandong Tian, Beidi Chen, Song Han, and Mike Lewis. Efficient streaming language models with attention sinks. *ICLR*, 2024.

[59] Zhiqiang Xie, Ziyi Xu, Mark Zhao, Yuwei An, Vikram Sharma Mailthody, Scott Mahlke, Michael Garland, and Christos Kozyrakis. Strata: Hierarchical context caching for long context language model serving, 2025.

[60] Shuo Yang, Ying Sheng, Joseph E. Gonzalez, Ion Stoica, and Lianmin Zheng. Post-training sparse attention with double sparsity, 2024.

[61] Zihao Ye, Lequn Chen, Ruihang Lai, Wuwei Lin, Yineng Zhang, Stephanie Wang, Tianqi Chen, Baris Kasikci, Vinod Grover, Arvind Krishnamurthy, and Luis Ceze. Flashinfer: Efficient and customizable attention engine for llm inference serving, 2025.

[62] Jiawei Yi, Ping Gong, Youhui Bai, Jiaqi Ruan, Shengnan Wang, Pengcheng Wang, Haibo Wang, Weiguang Wang, Xia Zhu, Feng Wu, and Cheng Li. Clo: Efficient llm inference system with cpu-light kvcache offloading via algorithm-system co-design, 2025.

[63] Gyeong-In Yu, Joo Seong Jeong, Geon-Woo Kim, Soojeong Kim, and Byung-Gon Chun. Orca: A distributed serving system for Transformer-Based generative models. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 521–538, Carlsbad, CA, July 2022. USENIX Association.

[64] Lingfan Yu, Jinkun Lin, and Jinyang Li. Stateful large language model serving with pensieve. In *Proceedings of the Twentieth European Conference on Computer Systems*, EuroSys '25, page 144–158. ACM, March 2025.

[65] Jingyang Yuan, Huazuo Gao, Damai Dai, Junyu Luo, Liang Zhao, Zhengyan Zhang, Zhenda Xie, Y. X. Wei, Lean Wang, Zhiping Xiao, Yuqing Wang, Chong Ruan, Ming Zhang, Wenfeng Liang, and Wangding Zeng. Native sparse attention: Hardware-aligned and natively trainable sparse attention, 2025.

[66] Hailin Zhang, Xiaodong Ji, Yilin Chen, Fangcheng Fu, Xupeng Miao, Xiaonan Nie, Weipeng Chen, and Bin Cui. Pqcache: Product quantization-based kvcache for long context llm inference, 2025.

[67] Xinrong Zhang, Yingfa Chen, Shengding Hu, Zihang Xu, Junhao Chen, Moo Khai Hao, Xu Han, Zhen Leng Thai, Shuo Wang, Zhiyuan Liu, and Maosong Sun. ∞bench: Extending long context evaluation beyond 100k tokens, 2024.

[68] Weilin Zhao, Zihan Zhou, Zhou Su, Chaojun Xiao, Yuxuan Li, Yanghao Li, Yudi Zhang, Weilun Zhao, Zhen Li, Yuxiang Huang, Ao Sun, Xu Han, and Zhiyuan Liu. Infllm-v2: Dense-sparse switchable attention for seamless short-to-long adaptation, 2025.

[69] Lianmin Zheng, Liangsheng Yin, Zhiqiang Xie, Chuyue Sun, Jeff Huang, Cody Hao Yu, Shiyi Cao, Christos Kozyrakis, Ion Stoica, Joseph E. Gonzalez, Clark Barrett, and Ying Sheng. Sglang: Efficient execution of structured language model programs, 2024.

[70] Yinmin Zhong, Shengyu Liu, Junda Chen, Jianbo Hu, Yibo Zhu, Xuanzhe Liu, Xin Jin, and Hao Zhang. DistServe: Disaggregating prefill and decoding for goodput-optimized large language model serving. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, pages 193–210, Santa Clara, CA, July 2024. USENIX Association.

[71] Qihui Zhou, Peiqi Yin, Pengfei Zuo, and James Cheng. Progressive sparse attention: Algorithm and system co-design for efficient attention in llm serving, 2025.

[72] Qihui Zhou, Peiqi Yin, Pengfei Zuo, and James Cheng. Sparseserve: Unlocking parallelism for dynamic sparse attention in long-context llm serving, 2025.

A Artifact Appendix

Abstract

This artifact provides the source code, pre-built environment, scripts, and datasets to reproduce the main results of ECHO, a KV cache offloading system with lossless prefetching for serving native sparse attention LLMs, implemented on top of SGLang. It supports running ECHO and the SGLang baseline on DeepSeek-V3.2 and reproducing the end-to-end throughput, end-to-end latency, fused prefetch kernel, and top-k analysis results reported in the paper.

Scope

The artifact validates the central claims of the paper, namely that ECHO improves generation throughput over the SGLang baseline under long-context workloads, both when all GPU HBM is allocated to the KV cache and under constrained GPU pool sizes, while preserving competitive latency. Specifically, it reproduces: end-to-end generation throughput (Figures 12–13), per-task throughput on InfiniteBench (Figure 14), end-to-end latency on ShareGPT (Figure 15), the fused intra-/inter-query prefetch kernel microbenchmarks (Figures 18–19), and the top-k analysis (Figures 7–9).

Contents

The artifact is distributed in two complementary forms:

- `echo-v0.tar.zst` — a pre-built Docker image with all dependencies (including compiled `sglang` and `DeepGEMM`) and the InfiniteBench test dataset, recommended for reproducing the end-to-end experiments.
- `ECHO.tar.gz` — a lightweight source-only package (`sglang` and `DeepGEMM` not pre-compiled) convenient for browsing the code.

Both forms include the end-to-end experiment scripts, the fused-kernel microbenchmark, and the top-k analysis pipeline (detailed in the experiment workflow below). Two configurations are compared throughout: `no-offload` (the SGLang baseline) and `offload` (ECHO).

Hosting

The artifact is hosted as a Zenodo record at https://zenodo.org/records/20376436, which contains both packages above, this guide, and the integrity checksum for the Docker image.

Requirements

A single server with 8 NVIDIA H20 GPUs (96 GB HBM each), 1.5 TB CPU memory, and GPUs connected to the host via PCIe Gen5. The software environment requires `docker` with GPU support and `zstd`; all dependencies are otherwise bundled in the Docker image. A large local disk (≥400 GB) is needed to store the downloaded DeepSeek-V3.2 (W4 AWQ) model.

Experiment Workflow

The full procedure is summarized below; all commands except setup are run *inside* the container. Each end-to-end script runs the SGLang baseline (`no-offload`) followed by ECHO (`offload`) on the same deployment, sweeping request rates and writing one log per rate to its output directory. By default the scripts sweep ten request rates (0.1–0.9 and `inf`); passing `--request-rates 0.8,0.9,inf` yields a faster run.

1. **Set up the environment.** Load and launch the Docker image, then download the W4 AWQ DeepSeek-V3.2 weights and activate the prepared environment (all later commands run inside the container):

```
zstd -dc echo-v0.tar.zst | docker load
docker run --privileged --gpus=all -it \
  -v /a_large_disk:/data1 --network host \
  --shm-size=128g --name echo-container echo:v0
modelscope download --local_dir \
  /data1/dsv32awq QuantTrio/DeepSeek-V3.2-AWQ
source /workspace/ECHO/.venv/bin/activate
```

2. **End-to-end throughput and latency (Figures 12–15).** From `/workspace/ECHO/sglang`, run the four sweeps below: full-HBM throughput (Figure 12), constrained-pool throughput (Figure 13), per-task throughput (Figure 14), and ShareGPT latency (Figure 15). Results are written under `infini-318/`, `infini-task/`, and `sharegpt-100/`, each split into `no-offload-<MMDD>/` and `offload-<MMDD>/`:

```
cd /workspace/ECHO/sglang
bash e2e_test_pd_infini.sh
bash e2e_test_pd_infini.sh \
  --no-offload-max-total-tokens 200_000 \
  --offload-nsa-dev-cache-size 200_000
bash e2e_test_pd_infini_task.sh
bash e2e_test_mix_sharegpt.sh
```

3. **Fused prefetch kernels (Figures 18–19).** From `/workspace/ECHO/DeepGEMM`, run the kernel microbenchmarks for intra- and inter-query prefetch:

```
cd /workspace/ECHO/DeepGEMM
python test_fuse.py decode_prefetch
python test_fuse.py prefill_prefetch
```

4. **Top-k analysis (Figures 7–9).** Regenerate the analysis figures from the bundled logits tensors; final PDFs appear under `echo/pdfs/`:

```
cd /workspace/ECHO/top-k-figures
pip install -r requirements.txt
./run_pipeline.sh
```

If a server fails to start, inspect the per-phase logs (e.g., `decode.log`, `router.log`) in the working directory, and run `scripts/killall_sglang.sh` to clear any leftover processes between runs.

ARTIFACT EVALUATED
USENIX
AVAILABLE

No Buffer, No Bottleneck: Efficient Zero-Copy KV Cache Offloading for Long-Context LLMs

Shutian Luo Haiying Shen
Department of Computer Science, University of Virginia

Abstract

Long-context large language models (LLMs) store computed attention *key* and *value* (KV) matrices from previous decoding steps to avoid full-sequence recomputation. While this reuse reduces computation, the KV cache grows linearly with sequence length, exerting significant pressure on GPU memory capacity. Existing offloading stores KV blocks in CPU memory but requires extra GPU buffers, wasting GPU memory and doubling CPU–GPU transfers, which creates capacity and bandwidth bottlenecks.

We present DirectKV, to our knowledge, the first zero-copy KV cache offloading system for modern heterogeneous CPU–GPU platforms such as NVIDIA GH200/GB200 with high-bandwidth NVLink-C2C interconnects. By leveraging NVLink-C2C instead of PCIe, DirectKV makes CPU memory a practical extension of GPU KV-cache capacity during long-context inference. DirectKV eliminates the GPU staging buffer (no buffer) by enabling GPU kernels to directly access CPU-resident KV cache, and substantially reduces the CPU–GPU data-movement bottleneck (no bottleneck) through CPU-memory-aware CUDA kernels tailored for heterogeneous CPU–GPU platforms. It further fuses KV generation with attention computation into a single CUDA kernel and employs warp-level pipelining to overlap KV fetching, computation, and write-back, thereby hiding stalls caused by CPU memory access. On GH200, DirectKV reduces CPU–GPU transfer volume by up to 50%, cuts GPU memory usage by 43%, and improves end-to-end performance by up to 1.2× compared to existing solutions.

1 Introduction

Large language models (LLMs) with long-context capabilities underpin applications such as multi-turn dialogue, in-context learning, and long-horizon code generation [18, 35, 50, 51]. To sustain throughput, modern inference engines maintain a *key–value (KV) cache* that stores previously computed *K*/*V* tensors [37]. During attention, each new query Q must attend to all prior K and V entries; caching avoids recomputation and reduces compute cost from quadratic to linear in sequence length. However, this efficiency comes at the expense of memory: KV cache size grows linearly with both sequence length and model dimension. For billion-parameter LLMs with 128K tokens, the cache alone can demand hundreds of gigabytes [28], far exceeding high-bandwidth GPU memory (HBM) capacity.

To address this issue, existing systems explore several CPU offloading strategies. Swap-based approaches [46, 53–55] place KV blocks in CPU memory (or SSD) and use large GPU staging buffers to overlap transfers with computation. Other systems store KV in remote or disaggregated memory and fetch it on demand [39, 42], but still follow a swap-like model in which storage and computation are separated: KV must be staged into GPU memory before attention. Crucially, all of these designs still stage KV cache into GPU memory along the critical path and depend on extra GPU-resident buffers during attention computation, because their attention kernels assume that all operands (including KV) reside in HBM. A third line of work partially offloads attention computation and KV cache to CPUs [27, 34, 41, 57], or moves the entire attention computation onto CPUs to avoid repeated PCIe transfers [31], but in doing so sacrifices the full compute capability of the GPU.

In short, none of these systems strikes the ideal balance of *using CPU memory as low-cost KV storage without additional HBM-resident KV buffers while fully exploiting GPU compute for high-performance attention*. They remain fundamentally limited by the large bandwidth gap between CPU–GPU interconnects (40–60 GB/s over PCIe) and HBM (3–4 TB/s), which leaves GPU compute units underutilized and exacerbates the GPU memory wall [30].

Recent CPU–GPU superchips such as NVIDIA GH200 and GB200 [14, 15] provide up to 900 GB/s bidirectional bandwidth via NVLink-C2C—7× higher than PCIe Gen5 [16]. This creates new opportunities for efficient KV cache offloading. However, existing systems such as Pie [53] still depend on staging buffers in HBM, consuming scarce GPU memory

and preventing full exploitation of this capability.

A more promising direction is *zero-copy*, where GPU kernels directly access CPU-resident KV tensors [2, 4]. Zero-copy eliminates staging buffers and reduces HBM usage, but naive application performs poorly: matrix multiplication kernels are tuned for HBM-resident tensors, so repeatedly fetching operands from CPU memory exposes the bandwidth gap and further lowers GPU L2 hit rate. Our measurements show that naïve zero-copy can be over 20× slower on PCIe and still 2× slower on NVLink-C2C (see § 2.3).

Our key observation is that *shared memory (SMEM)* within each streaming multiprocessor (SM) can be leveraged to make zero-copy practical. Acting as a fast on-chip buffer, SMEM enables flexible tiling strategies that restructure data access in matrix multiplication. By tiling through SMEM, we shift bandwidth pressure from the CPU–GPU interconnect to HBM, fully exploiting NVLink-C2C while maintaining GPU compute efficiency. This makes near-HBM performance achievable without staging buffers, moving closer to the ideal design: *CPU memory for cheap storage, GPU for high-performance computation*.

Building on this insight, we propose DirectKV, to our knowledge the first efficient zero-copy KV cache offloading system for NVIDIA GH200 Superchips. DirectKV targets long-context, latency-sensitive decoding workloads that require large effective batch sizes under tight GPU memory budgets. To relieve GPU memory pressure and fully exploit host DRAM, DirectKV centers on a *CPU-memory-aware tiling strategy* for matrix multiplication that deliberately shifts bandwidth demand from the CPU–GPU interconnect to HBM, using shared memory to stream CPU-resident tensors efficiently and reuse them across tiles. This design allows *zero-copy access to CPU-resident KV* to approach HBM-resident performance without relying on GPU staging buffers.

Beyond optimized tiling, DirectKV integrates two complementary techniques: (i) fine-grained warp-level pipelining that overlaps data transfer with computation, and (ii) kernel fusion of KV projection and attention that keeps *K*/*V* tensors in SMEM rather than round-tripping them through CPU memory. Together, these optimizations eliminate redundant KV transfers, hide interconnect latency, conserve HBM capacity, and deliver high throughput.

We build DirectKV[1] on FlashAttention-3 [44], but fundamentally extend it from GPU-resident execution to heterogeneous CPU–GPU settings on GH200, where interconnect bottlenecks emerge as the key challenge. Compared to prior offloading approaches [34, 46, 53], DirectKV reduces CPU–GPU transfer volume by up to 50%, lowers GPU memory usage by 43%, and improves end-to-end performance by up to 1.2×, all while maintaining full attention accuracy.
In summary, this paper makes the following contributions:

- *A new design for KV cache offloading.* We propose *DirectKV*, to our knowledge the first zero-copy offloading system for NVIDIA superchips. DirectKV eliminates staging buffers (no buffer) and substantially reduces the dominant CPU–GPU data-movement bottleneck (no bottleneck).

- *Kernel–memory co-design.* We identify why naïve zero-copy underperforms—repeated CPU-memory fetches and poor L2 locality—and introduce three optimizations: (i) CPU-memory-aware tiling to shift the bottleneck to HBM, (ii) warp-level pipelining to overlap transfer and compute, and (iii) kernel fusion to eliminate redundant KV refetching.

[1] Source code: https://github.com/shutianluo/DirectKV.

2 Background and Motivations

2.1 Attention Module in LLM Inference

LLM inference consists of multiple Transformer blocks, each containing an attention mechanism followed by a feed-forward network [28, 50, 51, 56]. The input to a Transformer block is a tensor $X \in R^{N\times D}$, where N is the number of tokens and D is the model dimension. The attention module projects X into three matrices: query (Q), key (K), and value (V), using separate weight matrices W^Q, W^K, and W^V:

$$Q,K,V = XW^Q,\ XW^K,\ XW^V, \quad W^Q,W^K,W^V \in R^{D\times d_k}.$$

For multi-head attention [45], Q, K, and V are reshaped into $R^{H\times N\times d}$, where H is the number of heads and $d = D/H$ is the per-head dimension. A common variant is *grouped-query attention (GQA)* [20], where multiple query heads share the same key and value heads to improve efficiency.

For LLM models that incorporate rotary positional embeddings (e.g., LLaMA [28, 50]), the queries and keys are further augmented with *rotary transformations*:

$$\tilde{Q},\tilde{K} = \mathrm{RoPE}(Q,K),$$

where RoPE applies position-dependent rotations in the query/key space to encode relative positional information. In contrast, other models (e.g., OPT [56]) omit this step and rely on alternative positional encodings.

Finally, for each head, the *scaled dot-product attention* is computed as:

$$\mathrm{Attention}(Q,K,V) = \mathrm{softmax}\left(\frac{QK^\top}{\sqrt{d}}\right)V.$$

LLM inference proceeds in two phases [59]: the prefill phase, where the full input sequence is processed in parallel to initialize the KV cache, and the decode phase, where tokens are generated autoregressively, each new query attending to all cached K/V.

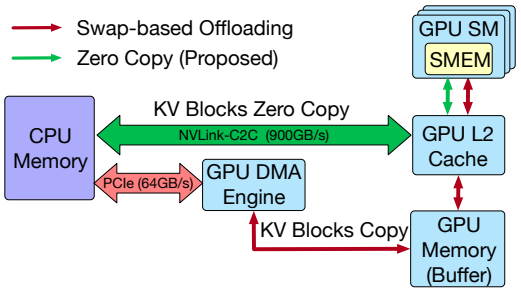
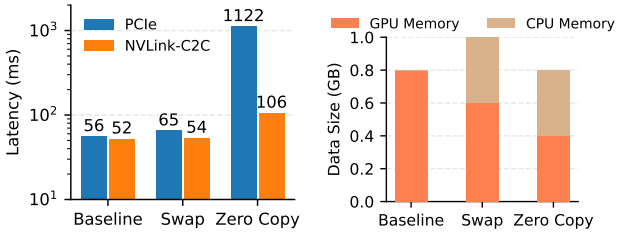


Figure 1: KV cache offloading via swapping and zero copy.



(a) Latency comparison. (b) Memory usage.

Figure 2: Comparison among three mechanisms for matrix multiplication.

2.2 KV Cache Offloading

During autoregressive inference, the KV cache stores K and V tensors from prior decoding steps, reducing attention cost from quadratic to linear in sequence length. Yet the cache grows *linearly* with context length and model dimension, quickly exhausting GPU memory. Existing systems offload KV blocks to CPU memory through HBM staging buffers.

2.2.1 Swap-based KV Cache Offloading

As shown by the red path in Fig. 1, KV blocks are stored in CPU memory and are copied to GPU memory before use via a GPU copy engine [13]. To reduce stalls, systems overlap these transfers with computation. However, PCIe bandwidth (64 GB/s) is far lower than HBM bandwidth (4 TB/s), making data movement a bottleneck. To hide transfer latency, a large GPU buffer is required, wasting valuable GPU memory. Even then, poor prefetch coordination can still cause GPU stalls. Moreover, swap-based designs perform layer-by-layer swap-in and swap-out of KV blocks, doubling interconnect traffic and further constraining scalability.

Swap-based offloading can prevent CUDA Graphs from being replayed unchanged. CUDA Graphs reduce per-token launch overhead by capturing and replaying a fixed sequence of GPU operations. However, swap-based KV offloading introduces explicit `cudaMemcpyAsync` [17] operations for KV swap-in/out, whose addresses vary across decoding steps as different KV blocks are selected. These dynamic transfer addresses can force graph updates or re-capture, reducing the benefits of CUDA Graphs in practical serving systems.

2.2.2 Zero-Copy KV Cache Offloading

As shown by the green path in Fig. 1, zero-copy KV cache offloading [4] allows GPU kernels to directly access CPU-resident KV blocks. This eliminates explicit transfers and HBM staging buffers. Once KV blocks are device-accessible, the attention kernel can fetch them directly and use asynchronous tiled data movement to bring KV tiles into SMEM for computation.

Compared with swap-based offloading, zero-copy is better suited for CUDA Graph execution. Because KV blocks are loaded inside the attention kernel rather than through explicit memory-copy operations, dynamic KV-block selection changes only the kernel inputs or internal address resolution, not the captured graph topology. This allows the graph to be replayed without per-step updates or re-capture.

The effectiveness of zero-copy primarily depends on CPU–GPU bandwidth rather than access latency. KV-cache access in long-context attention is naturally coarse-grained: each SM fetches KV tiles of around 100 KB, aligned with its SMEM capacity, and more than 100 SMs can fetch such tiles concurrently. This creates MB-scale aggregate data movement from CPU memory to the GPU. In this regime, sustained CPU–GPU bandwidth becomes the dominant bottleneck. Recent CPU–GPU platforms such as NVIDIA GH200 make this direction practical with NVLink-C2C, which provides up to 900 GB/s bidirectional bandwidth and allows CPU memory to act as an effective extension of GPU KV-cache capacity.

2.3 Challenges in KV Cache Offloading

Even with NVLink-C2C, a significant *bandwidth gap* remains between the CPU–GPU interconnect (900 GB/s bidirectional) and HBM bandwidth (4 TB/s). In the GEMM case study, repeated operand reuse across the $O(n^3)$ computation amplifies remote-memory traffic. In attention, a similar effect appears when CPU-resident KV tiles are repeatedly fetched across query, head, and batch dimensions.

To illustrate this challenge, we conduct a case study comparing swap-based and zero-copy KV cache offloading using a matrix multiplication benchmark, $C = A \times B$, where A, B, and C are square matrices of size $n \times n$ ($n = 10240$, each 400 MB). We evaluate three configurations: (i) *Baseline*, where both A and B reside entirely in GPU memory; (ii) *Swap*, where a 200 MB GPU buffer is allocated to temporarily store portions of B transferred from CPU memory via PCIe or NVLink-C2C during multiplication; and (iii) *Zero-Copy*, where B resides entirely in CPU memory and is accessed directly by the GPU over PCIe or NVLink-C2C without intermediate buffering. Experiments are conducted on two hardware platforms: an NVIDIA H100 connected to the host via PCIe and an NVIDIA GH200, where an H100 GPU is tightly coupled to a Grace

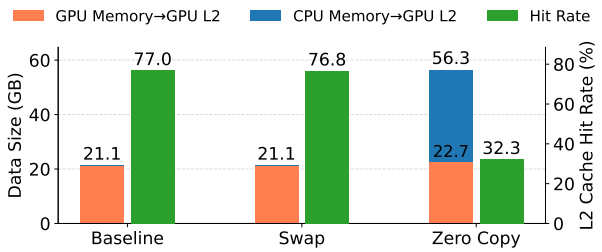


Figure 3: Data fetching and L2 cache hit rate.

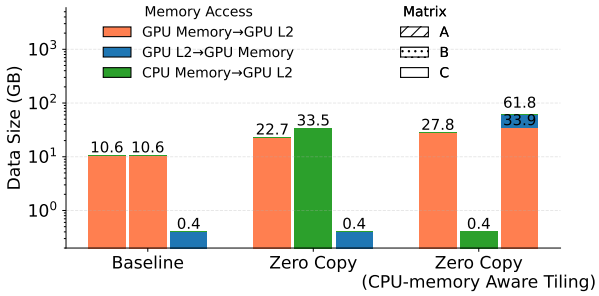


Figure 4: Data movement breakdown across three matrix multiplication strategies.

CPU via NVLink-C2C.

Zero-Copy saves GPU memory but still incurs performance penalties. As shown in Fig. 2(a), the *Swap* configuration adds less than 10% overhead on PCIe and about 5% on NVLink-C2C due to asynchronous data transfer. In contrast, *Zero-Copy* incurs over a 20× slowdown on PCIe (1122 ms vs. 56 ms) and a 2× slowdown on NVLink-C2C (106 ms vs. 52 ms). This penalty arises because *Zero-Copy* repeatedly fetches B from CPU memory throughout the $O(n^3)$ computation, fully exposing the bandwidth gap between CPU–GPU interconnects and HBM. While costly in performance, *Zero-Copy* eliminates the need for a large GPU buffer, reducing memory usage by 50% compared to *Baseline* and by 33% compared to *Swap*, as shown in Fig. 2(b).

Zero-Copy further reduces GPU L2 cache efficiency. To quantify this effect, we measure data traffic and L2 hit rate on GH200 (Fig. 3). *Baseline* and *Swap* perform similarly since both A and B reside in GPU memory, achieving high L2 hit rates (∼77%) and 21.1 GB of HBM-to-L2 traffic—well above the matrix size due to repeated accesses in the $O(n^3)$ computation. In contrast, *Zero-Copy* suffers frequent stalls because CPU–GPU transfers (900 GB/s, 450 GB/s per direction) are much slower than HBM-to-L2 transfers (4 TB/s), increasing CPU-to-GPU traffic to 22.7 GB while reducing the L2 hit rate to 32.3%—a 58% drop compared to *Baseline*/*Swap*. This shows that reduced cache locality, in addition to limited bandwidth, compounds the performance gap of naïve zero-copy offloading.

2.4 Opportunity: Kernel-memory Co-Design

GPU kernels support flexible data access patterns, enabling customized CUDA implementations that can optimize *zero-copy* KV cache access from CPU memory. This flexibility creates opportunities to mitigate the challenges discussed above. In particular, we focus on two key techniques — *tiling* and *warp-level pipelining* — to improve data locality, reduce redundant transfers, and better exploit the available CPU–GPU bandwidth.

2.4.1 Tiling Technology for Asymmetric Bandwidth

To better understand optimization opportunities, we revisit matrix multiplication with a focus on *tiling technology*. Modern GPUs have much smaller cache capacity than global memory but provide high-bandwidth shared memory within each streaming multiprocessor (SM). CUDA kernels can explicitly control shared memory usage, enabling computation to be organized in *tiles*: small submatrices of A and B are fetched from global memory into shared memory, multiplied to produce a partial tile of C, and the partial results are accumulated in shared memory before writing the final output to global memory. This tiling mechanism reduces global memory accesses and provides flexibility to redesign the computation pattern for cases where one input matrix resides in CPU memory.

Figure 4 shows the baseline access pattern. In *Baseline* (both A and B in GPU memory), the bottleneck is fetching A and B from GPU memory to L2 cache, each at 10.6 GB, while writing only 0.4 GB of C back to GPU memory. CUTLASS-based matrix multiplication iterates over tiles of A and B independently to compute each tile of C, with no data dependency between tiles. However, in *Zero-Copy*, where B resides in CPU memory, the asymmetric bandwidth between NVLink-C2C (900 GB/s, 450 GB/s per direction) and HBM (4 TB/s) causes the data fetched from CPU memory to be amplified by repeated accesses, reaching 33.5 GB—over 3× the original tile size—making CPU-to-GPU transfers the bottleneck.

To address this, we develop a *CPU-memory-aware tiling* strategy for matrix multiplication (details in § 5.1). The key idea is to shift bandwidth pressure away from the CPU–GPU interconnect onto on-device HBM. Instead of iterating over tiles of A and B, our optimized kernel iterates over tiles of A and C, while keeping the current B tile resident in SM shared memory. This reduces CPU-to-GPU traffic for B from 33.5 GB to just 0.4 GB, directly attacking the main bottleneck, since HBM bandwidth is roughly an order of magnitude higher than CPU–GPU interconnect bandwidth. The trade-off is that C must now be read from and written to HBM more frequently for accumulation, increasing HBM traffic by about 2× (from 33.5 GB over the CPU–GPU interconnect to 61.8 GB + 33.9 GB in HBM). However, this extra load can be absorbed by HBM's higher bandwidth while substantially relieving the CPU–GPU bottleneck. As shown in Fig. 5, this optimization reduces latency from 106 ms to 54 ms (a 49% performance improvement) and increases the L2 cache hit

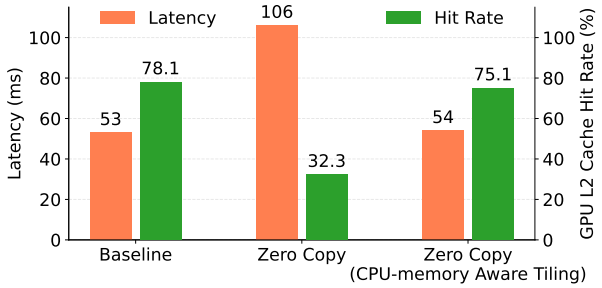
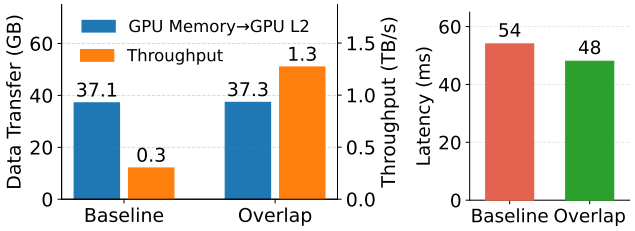
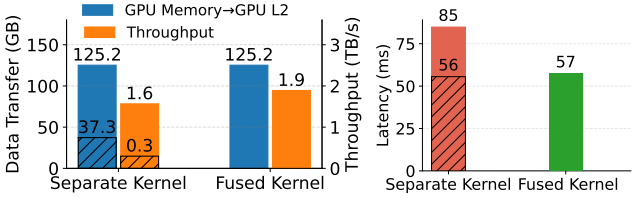


Figure 5: Performance and GPU L2 cache hit rate across three matrix multiplication strategies.



(a) Data fetching & HBM throughput. (b) Performance.

Figure 6: Data fetching and performance under baseline (no overlap) vs. warp-level overlap.



(a) Data fetching & HBM throughput. (b) Performance.

Figure 7: Data fetching and performance under separate kernel vs. fused kernel. Hatched bars (bottom) denote the first kernel.

rate from 32.3% to 75.1%.

Takeaway: By leveraging tiling, we can design a CPU-memory-aware *Zero-Copy* scheme that accommodates asymmetric bandwidth. Iterating over GPU-resident matrices rather than CPU-resident ones shifts bandwidth demand to HBM, fully exploiting its capacity and improving L2 cache efficiency for significantly better performance.

2.4.2 Warp-Level Pipelining for Communication and Computation Overlapping

Modern GPUs such as NVIDIA's Hopper architecture (H100/GH200) support multiple warp groups within a thread block that can execute distinct tasks concurrently. This capability enables *warp-level pipelining*, where different warps overlap communication (data fetching) and computation (matrix multiplication) to hide memory latency (§ 5).

To quantify its impact, we compare matrix multiplication with and without pipeline fetching. In the baseline (no overlap), the kernel fetches one tile at a time, synchronizes all threads, performs computation, and then repeats. In contrast, pipeline fetching allows the SM to prefetch the next tile while simultaneously computing on the current tile. As shown in Fig. 6, both approaches transfer nearly identical data volumes. However, warp-level pipelining sustains 4.3× higher HBM throughput (1.3 TB/s vs. 0.3 TB/s) and reduces latency by 11% (48 ms vs. 54 ms), demonstrating effective communication–computation overlap.

Takeaway: Warp-level pipelining effectively hides memory latency by overlapping data transfer and computation, yielding significantly higher throughput and lower latency without increasing data movement.

2.4.3 Fused Kernel for High Throughput

In conventional implementations, projection and attention score computation are launched as separate CUDA kernels. This separation forces the K and V tensors generated by the projection kernel to be written back to CPU memory (to update the KV cache) and then re-read by the attention kernel — creating redundant CPU–GPU transfers and exacerbating the bandwidth gap between NVLink-C2C (900GB/s aggregate, 450GB/s per direction) and HBM (4 TB/s per direction). This motivates us to *fuse* the projection and attention score computation into a single CUDA kernel to improve the throughput of data transfer (details in § 5.3 and § 5.4).

We evaluate this idea using a simplified case study. Specifically, the first kernel generates K via a matrix multiplication $K = XW_k$ and the second kernel consumes K for another matrix multiplication $P = QK^\top$. Rather than separate it into different kernels, we fuse them into one kernel using two stages. The first stage fetches X and W_k from HBM, generates K in registers, and writes the newly generated K back to CPU memory for KV-cache updates. The second stage immediately consumes K for $QK^\top$ score computation. The tensor X, W_k and Q reside in the SMEM. This design breaks the strict read–write dependency in traditional *Zero-Copy*, minimizes stalls, and increases effective throughput utilization.

As shown in Fig.7(a), the separate-kernel design transfers a similar amount of data as the fused kernel but sustains 15.8% lower HBM throughput due to redundant writes of K back to CPU memory and the subsequent re-fetching in the next kernel. In contrast, the fused kernel eliminates these redundant transfers, achieving 1.9TB/s throughput. This directly translates to performance gains: latency drops from 85ms to 57 ms, a 49% speedup as shown in Fig.7(b), by improving memory locality and bandwidth utilization.

Takeaway: Kernel fusion eliminates redundant memory transfers and improves data locality, further delivering higher throughput and performance.

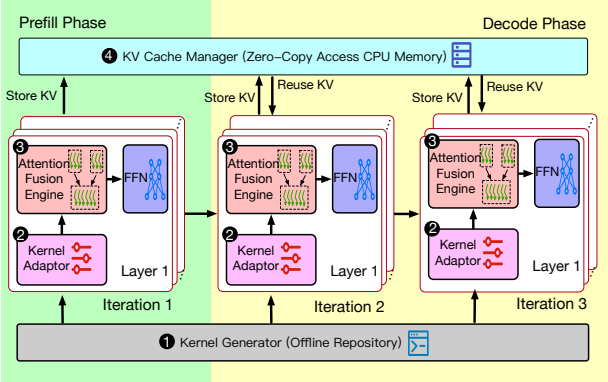


Figure 8: The system architecture of DirectKV.

3 Overview of DirectKV Architecture

This section outlines the overall architecture of DirectKV, a KV cache decoding system for LLM serving by reducing GPU memory pressure and improving runtime efficiency. DirectKV integrates *zero-copy KV cache offloading* with *fused projection-attention kernels*, dynamically adapting to input precision and execution phases (prefill or decode) as shown in Fig. 8. Both phases utilize a shared kernel execution framework that includes the following core components:

The *Kernel Generator* ❶ compiles a rich set of CUDA kernel candidates offline, specialized for different tensor precisions (e.g., FP16, BF16), tiling configurations, and execution modes (prefill vs. decode). It serves as an offline repository, ensuring that optimized kernels are pre-built and ready for runtime dispatch.

The *Kernel Adaptor* ❷ dynamically selects an appropriate CUDA kernel at runtime based on tensor precision (e.g., FP16, BF16), head dimension, and current execution phase (multi-token prefill vs. single-token decode). It allows the system to flexibly dispatch to optimized implementations without manual intervention or recompilation.

The *Attention Fusion Engine* ❸ fuses K/V projection and attention computation into a single CUDA launch to eliminate redundant KV cache transfers. It leverages shared memory within the SM and warp-level programming to overlap KV cache transfer with attention score computation.

The *KV Cache Manager* ❹ stores the key and value tensors in host-pinned memory using zero-copy access. Rather than using explicit cudaMemcpyAsync to move KV tensors between host and device, we utilize pinned memory to allow direct GPU access to host-resident tensors within SM. This design avoids redundant transfers and supports long-sequence inference without increasing GPU memory usage.

4 Kernel Generation and Adaptation

Typically, CUDA requires kernel specialization for input precision and tiling to fully exploit hardware, so multiple kernels must be compiled offline and selected at runtime. Accordingly, DirectKV introduces a *Kernel Generator* to generate candidate kernels offline and *Kernel Adaptor* a runtime component that chooses the most efficient kernel online.

4.1 SMEM Partitioning and Reuse

In DirectKV, SMEM serves as the on-chip buffer for matrix multiplications in the attention module. Its limited capacity is a primary hardware constraint that directly shapes kernel design. To balance projection and attention workloads, the SMEM within each SM is logically divided into two regions:

1. **Projection buffers:** used to store tiles of the input X and projection weights W_k and W_v during KV projection.
2. **Attention buffers:** used to store intermediate K, V, the query Q, and the output O during attention computation.

To alleviate SMEM pressure, DirectKV overlaps the usage of these two regions. Specifically, K reuses the buffer originally allocated for W_k, and V reuses the buffer allocated for W_v. Phase-specific optimizations further reduce demand: in the *decode* phase, O is written directly from registers to HBM (see § 5.3), while in the *prefill* phase, O reuses the buffer allocated for W_v in shared memory (see § 5.4). This partition-and-reuse strategy maximizes SMEM utilization and enables larger tiles under fixed hardware limits.

By default, DirectKV adopts two pipeline stages to overlap computation and communication at the warp level, which doubles the buffering requirement. Since SMEM is carved from the unified L1/SMEM pool (e.g., 256 KB on GH200) [7], we conservatively reserve a fraction α (80% by default) for SMEM and leave the remainder to L1 cache. The SMEM allocation must therefore satisfy:

$$\alpha \cdot P \geq 3 \cdot m \cdot \text{size(T)} \cdot (Dim \cdot N), \tag{1}$$

where P is the total L1/SMEM pool size, the factor 3 accounts for buffering three tensors in projection and attention, m is the number of pipeline stages (2 by default), T is the tensor data type with size(T) denoting its element size in bytes, Dim is the KV head dimension, and N is the tile size.

Given a specific configuration `<T, Dim, N>`, we compute the maximum feasible N that satisfies this constraint. These valid configurations are collected into a candidate set that guides kernel pre-building.

4.2 Offline Kernel Pre-Building

From this candidate set, the *Kernel Generator* pre-compiles fused kernels (details in § 5) using C++ template instantiations. Listing 1 illustrates the prefill kernel; the decode kernel

Listing 1: Offline Kernel Pre-building using C++ Templates.

```
template <typename T, int Dim, int N>
__global__ void prefill_kernel(
    T* X, const T* Wq, const T* Wk,
    const T* Wv, T* Q, T* O, int seq_len);

// Offline instantiations
template __global__ void prefill_kernel<int, 128,
     64>(...);
template __global__ void prefill_kernel<float, 64,
     64>(...);
```

is constructed analogously but specialized for its execution phase. Template syntax allows kernels to be specialized at compile time with respect to `<T, Dim, N>`, ensuring each variant is fully optimized by the compiler while avoiding runtime code generation overhead. This approach also keeps the codebase maintainable, as the same kernel logic can be reused across multiple configurations. All such instantiations are compiled offline into a candidate pool of fused kernels.

4.3 Online Kernel Choosing

At runtime, the *Kernel Adaptor* selects the most suitable kernel for the current phase (*prefill* or *decode*) by matching the requested `<T, Dim, N>` configuration to the closest available instantiation in the candidate pool. This process is lightweight, as it only requires substituting the template parameters with the pre-compiled kernels. As a result, DirectKV can dynamically adapt to workload characteristics while guaranteeing that each kernel execution leverages a carefully tuned, hardware-efficient implementation.

5 Kernel–Memory Co-Design Attention

We design the CUDA kernel to fully exploit *tiling* and *warp-level programming*, adapting the data access pattern to efficiently support zero-copy KV cache offloading. In this section, we detail how these techniques enable a high-performance fused kernel tailored for tightly coupled CPU–GPU superchip architectures.

5.1 CPU-Memory Aware Matrix Multiplication

To exploit HBM bandwidth and relieve pressure on the CPU–GPU interconnect, our key idea is to shift data-fetching overhead from host to device memory. This requires rethinking matrix multiplication access patterns, which conventionally assume all operands reside in GPU memory. Tiling techniques [5, 26] offer a natural solution: by decomposing large matrices into smaller submatrices (tiles) that fit within SMEM, CUDA kernels can flexibly control data movement, maximize reuse and minimize redundant accesses. Building on this, we design a CPU-memory aware tiling strategy that reuses CPU-fetched tiles efficiently while leveraging HBM's bandwidth for repeated accesses.

Algorithm 1: Tiled data-fetching patterns for matrix multiplication (Native vs. CPU-aware).

1 **Procedure** `NativeTiling` $(A \in R^{M\times K}, B \in R^{K\times N}, C \in R^{M\times N})$:
2 Initialize $C_{i,j}$;
3 **for** $k \leftarrow 1$ **to** T_K **do**
4 Load $A_{i,k}$; /* HBM → SMEM */
5 Load $B_{k,j}$; /* HBM → SMEM */
6 $C_{i,j} \leftarrow C_{i,j} + A_{i,k} \times B_{k,j}$; /* reuse stationary C */
7 Store $C_{i,j}$; /* registers/SMEM → HBM */
8 **Procedure** `CpuAwareTiling` $(A \in R^{M\times K}, B \in R^{K\times N}, C \in R^{M\times N})$:
9 Load $B_{k,j}$; /* CPU Memory → SMEM (zero copy) */
10 **for** $i \leftarrow 1$ **to** T_M **do**
11 Load $A_{i,k}$; /* HBM → SMEM */
12 Load $C_{i,j}$; /* HBM → SMEM */
13 $C_{i,j} \leftarrow C_{i,j} + A_{i,k} \times B_{k,j}$; /* reuse stationary B */
14 Store $C_{i,j}$; /* registers/SMEM → HBM */

We use a matrix multiplication kernel $C = A \times B$ to illustrate the distinction between conventional tiling and our CPU-aware tiling as shown in Algorithm 1. The key difference lies in which operand is reused and, consequently, where the bandwidth pressure is concentrated within the memory hierarchy. In *NativeTiling* (Step 1), both A and B tiles are repeatedly fetched from GPU HBM into shared memory, while C remains stationary in registers or SMEM until the final write-back. This design is efficient when all operands reside in GPU memory but requires frequent HBM accesses for both A and B. In particular, each multiplication step reloads B tiles for every (i, j, k) triple, so the data movement on B scales with the $O(n^3)$ computation. In contrast, *CpuAwareTiling* (Step 8) adapts to zero-copy offloading by treating B as stationary: each B tile is fetched once per (k, j) pair and reused across all i, reducing the traffic from $O(n^3)$ to $O(n^2)$. The trade-off is that C must be reloaded and updated from HBM at every step, increasing its traffic cost from $2 \cdot O(n^3)$ in *NativeTiling* to $3 \cdot O(n^3)$ in *CpuAwareTiling*. This design shifts bandwidth demand away from the constrained CPU–GPU interconnect and onto HBM, where the additional overhead can be efficiently absorbed.

This trade-off directly influences the design of attention score computation. Since SMEM is insufficient to hold all tensors, there are two possible iteration strategies: (i) iterate over Q for each (K,V) pair, or (ii) iterate over (K,V) for a given Q. In our design, we adopt the first strategy during the *prefill* phase (§ 5.3), where the KV cache resides in CPU memory and is expensive to reload, making it preferable to iterate over Q stored in HBM. This scheme corresponds to *CpuAwareTiling*. In contrast, during the *decode* phase (§ 5.4), the length of Q is only a single token, so we instead iterate over (K,V) to avoid repeatedly reading intermediate output

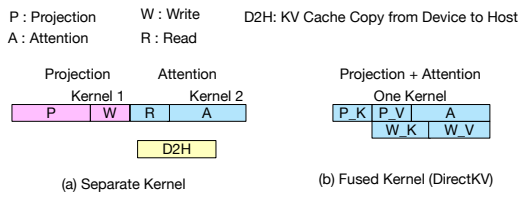


Figure 9: KV Projection and attention fusing.

scores. This scheme corresponds to *NativeTiling*.

5.2 Warp-Level Parallelism for Overlapping

On NVIDIA's Hopper architecture [22], SM threads can be divided into multiple warp groups, with each group independently handling tasks such as computation or communication. To maximize efficiency in the fused kernel, we employ *warp-level pipelining* for both projection and attention: while one set of warps fetches the next tiles from memory, others concurrently compute on the current tiles. This overlap hides memory latency and sustains high throughput, akin to asynchronous data copy in swap-based KV offloading where host–device transfers overlap with computation. On Hopper GPUs, this mechanism is further reinforced by the *Tensor Memory Accelerator (TMA)* [10], a hardware copy engine within each SM that supports asynchronous data movement and frees compute warps to focus on arithmetic.

5.3 Fused Kernel for Prefill Phase

Rather than launching separate kernels for each projection followed by a standalone attention kernel, our fused approach executes the entire sequence in a single pass (Fig. 9). The key insight is that the generated K and V tiles are retained in SMEM and immediately consumed by the subsequent attention computation, eliminating redundant writes to and reads from CPU memory.

Algorithm 2 outlines the design of the fused prefill kernel. The kernel initializes three specialized warp groups—*producer*, *consumer*, and *storer* (Step 3). The producer warps fetch tiles from HBM, the consumer warps perform the computation, and the storer warps handle writing intermediate results (e.g., K/V) back to CPU memory when required. This division enables overlap between data movement and computation.

The projection stage is decomposed into two sub-operations that generate K and V tiles, respectively. For each sub-operation, we employ pipeline parallelism across warp groups. For example, during K tile generation, while consumer warps compute on the current tile (Step 7), producer warps simultaneously prefetch the next tile from HBM (Step 6), thereby overlapping computation with data movement. Once the projection of K and V for a given tile is complete, the storer warps transfer the results to CPU memory, while the consumer warps proceed with attention computation

Algorithm 2: Fused Prefill Kernel

Input: Input embeddings $X \in R^{N\times D}$, projection weights $W_k, W_v \in R^{D\times d}$, query $Q \in R^{S\times d}$, head dimension d, sequence length S, tile sizes B_{kv} and B_q for KV cache and query, respectively.

1 Initialize output tensor $O \in R^{S\times d}$
2 Initialize streaming-softmax state $(\ell=0,\ m=-\infty)$
3 Initialize three Warp Groups including Producer, Consumer and Storer
4 **for** $j \leftarrow 1$ **to** $T_s = \left\lceil \frac{S}{B_{\text{kv}}} \right\rceil$ **do**
  /* KV Projection Stage */
5 **for** $i \leftarrow 1$ **to** $T_d = \left\lceil \frac{D}{d} \right\rceil$ **do**
6 **Producer**: Load X_{ij}, W_i^k from HBM
7 **Consumer**: $K_j = X_{ij} W_i^k + K_j$
8 **if** `use_rope` **then**
9 $K_j \leftarrow RoPE(K_j)$
10 **Storer**: Zero-copy write K_j to CPU Memory
11 **for** $i \leftarrow 1$ **to** $T_d = \left\lceil \frac{D}{d} \right\rceil$ **do**
12 **Producer**: Load X_{ij}, W_i^v from HBM
13 **Consumer**: $V_j = X_{ij} W_i^v + V_j$
14 **Storer**: Zero-copy write V_j to CPU Memory
  /* Attention Stage */
15 **for** $i \leftarrow 1$ **to** $T_s = \left\lceil \frac{S}{B_q} \right\rceil$ **do**
16 **Producer**: Load Q_i, O_i, m_i, ℓ_i from HBM ;
17 **Consumer**:
18 $S_{ij} \leftarrow Q_i K_j^\top$
19 $(m_i, \ell_i, O_i) \leftarrow StreamSoftmax(m, \ell_i, O_i, S_{ij}, V_j)$
20 Write O_i, m_i back to HBM

on previously generated tiles. After generating the K tiles, we optionally apply rotary position embeddings (RoPE) [28, 48] (Step 8). Because RoPE relies only on precomputed sine and cosine functions determined by token positions, each K tile can be transformed independently, enabling efficient parallelization across SMs without inter-tile dependencies.

For attention score computation, we adopt the strategy of iterating over Q (Step 15), which is precomputed before launching the fused attention kernel and resides in HBM. For each (K,V) tile, this design allows SMs to process different tiles independently, thereby maximizing parallelism. A trade-off of this design is that intermediate outputs O_i and auxiliary statistics m_i, ℓ_i must be repeatedly fetched for accumulation and streaming softmax updates [26, 36, 40] (Step 19). These updates are standard in long-sequence attention to ensure numerical stability and are retained without modification. Importantly, the additional HBM traffic introduced by these updates remains modest compared to CPU–GPU transfers, and thus does not become the dominant bottleneck. To avoid write–

read dependencies on O_i across SMs, we assign different attention heads and batches to different SMs. Since the product of the number of heads (e.g., 8) and batch size (e.g., 32) typically exceeds the number of SMs (e.g., 132 on GH200), this scheduling ensures that no two SMs simultaneously operate on the same head–batch pair with different (K,V) tiles. Finally, similar to the projection stage, we employ pipeline parallelism: producer warps prefetch the next Q tiles from HBM (Step 16), while consumer warps concurrently perform computations on the current tiles (Step 18), thereby overlapping data movement with computation.

5.4 Fused Kernel for Decode Phase

In contrast to the prefill phase, where avoiding repeated KV cache fetching from CPU memory dominates the cost, the decode phase reuses the cached K,V heavily while computing projections and attention scores for only a single newly generated query token. To fully utilize parallelism in this setting, we adopt the complementary strategy of iterating over all cached (K,V) given the single query q_t. Since only one token is processed, each (K,V) pair needs to be traversed just once, avoiding additional CPU–GPU memory transfers. Furthermore, this design eliminates redundant reloads of the output attention scores: o_t can be retained in registers and reused across iterations, rather than repeatedly fetched from HBM. This optimization minimizes overhead while preserving high throughput as illustrated in Algorithm 3.

The fused decode kernel requires only two warp groups, *producer* and *consumer*, since only a single newly generated (K,V) pair is written back to CPU memory. In this case, a dedicated storer group is unnecessary, and the consumer warps handle the write-back after completing the new (K,V) projection. Specifically, the producer first fetches the query q_t from HBM (Step 2) to prepare for subsequent iterations. During the attention stage, the producer loads (K,V) tiles from CPU memory into SMEM via zero-copy, while the consumer performs computation over these tiles. Pipeline parallelism is employed to overlap communication and computation: while the producer fetches the next tile (Step 5), the consumer concurrently computes on the current tile (Step 6), thereby sustaining throughput and reducing stalls.

6 Efficient KV Cache Management

The KV cache grows linearly with sequence length and quickly exceeds GPU capacity. To address this, we introduce the *KV Cache Manager*, which offloads KV tensors to CPU memory and enables direct GPU access via zero-copy.

The manager allocates CPU buffers with `cudaHostAlloc` as pinned memory [9], making host memory page-locked and directly accessible to the GPU without staging. Once written, KV tensors remain in pinned buffers and are reused across iterations without recomputation or redundant transfers.

Algorithm 3: Fused Decode Kernel (token-by-token)

Input : New token $x_t \in R^{1\times d_{\text{model}}}$, weights W_k, W_v, head dim d, cached KV length S_{kv}, flags `use_rope`, tile sizes B_{kv} for KV cache.

1 Initialize output tensor o_t
2 Initialize streaming-softmax state $(\ell=0,\ m=-\infty)$
3 **Producer**: Load q_t from HBM
/* Attention over cached KV */
4 **for** $j \leftarrow 1$ **to** $T_s = \left\lceil \frac{S_{kv}}{B_{\text{kv}}} \right\rceil$ **do**
5 **Producer**: zero-copy load cached tiles (K_j, V_j) from CPU memory
6 **Consumer**:
7 $S_j \leftarrow q_t K_j^\top$
8 $(m,\ell,o_t) \leftarrow StreamSoftmax(m,\ell,o_t,S_j,V_j)$
/* Project current token's KV */
9 **for** $i \leftarrow 1$ **to** $T_d = \lceil \frac{D}{d} \rceil$ **do**
10 **Producer**: Load X_i, W_i^k, W_i^v from HBM
11 **Consumer**: $k_t = X_i W_i^k + k_t,\ v_t = X_i W_i^v + v_t$
12 **if** `use_rope` **then**
13 **Consumer**: $k_t \leftarrow RoPE(k_t)$
14 **Consumer**: Append (k_t, v_t) to KV cache in CPU memory
/* Attention with new KV */
15 **Consumer**:
16 $s_t \leftarrow q_t k_t^\top$
17 $(m,\ell,o_t) \leftarrow StreamSoftmax(m,\ell,o_t,s_t,v_t)$
18 Write o_t back to HBM

During *prefill*, the manager stores the initial K,V entries from each layer in CPU memory. During *decode*, the GPU reuses these cached entries directly, while only appending new K,V for the latest token.

Integrated with the Kernel Adaptor and Attention Fusion Engine (Fig. 8), the KV Cache Manager reduces interconnect traffic, relieves GPU memory pressure, and enables scalable long-context generation beyond GPU-only caching.

7 Implementation and Evaluation

7.1 System Implementation

Prototype. We implement *DirectKV* in CUDA 12.4, building on NVIDIA's CUTLASS library for matrix multiplication [12] and extending FlashAttention-3 [44]. The implementation integrates three key techniques: (i) CPU-aware matrix multiplication, (ii) warp-level pipelining, and (iii) kernel fusion of projection and attention. In total, our prototype contributes roughly 5,300 lines of CUDA/C++ code on top of FlashAttention-3, with additional Python bindings to enable integration into PyTorch 2.2. For kernel-level analysis,

we use NVIDIA Nsight Compute to report hardware metrics, including L2 cache hit rate and memory throughput, in order to quantify efficiency gains.

KV cache management. DirectKV includes a lightweight KV cache manager (❹ in Fig. 8) to support direct attention over CPU-resident KV blocks. The manager allocates KV buffers in pinned host memory using `cudaHostAlloc` [9]. Since these buffers are page-locked and GPU-accessible, DirectKV kernels can access CPU-resident KV blocks through device-visible pointers without staging them in GPU memory. Once generated, KV tensors remain in the pinned buffers and are reused across decoding iterations, avoiding recomputation and redundant CPU–GPU transfers.

7.2 Evaluation Setup

Server Configurations. We evaluate DirectKV on an NVIDIA GH200 Grace–Hopper Superchip with a Hopper GPU, 96 GB of HBM3, and LPDDR5X CPU memory [15]. The system runs CUDA 12.4 and PyTorch 2.3, and our custom kernels are implemented using CUTLASS 3.0+ with Hopper support via CuTe. For baseline comparison, we use the H100 (PCIe Gen5) to quantify the benefits of NVLink-C2C (§ 7.4.3), since both GH200 and H100 employ the same H100 GPU architecture.

LLM models. We evaluate Llama-3.1-8B [11] and OPT [56] models with 13B and 30B parameters.

Workloads. Following prior work on LLM serving [37], we evaluate on real-world datasets including ShareGPT [6, 8] (user-shared ChatGPT conversations) and Alpaca [49, 52] (instruction-following data generated by GPT-3.5). Since request timestamps are not provided, we generate arrivals as a Poisson process parameterized by the request rate [37].

Our main experiments use request rates up to 30 req/s and context lengths from 1k to 32K tokens, representing a stable operating region where systems can be compared under reasonable latency. We further add stress-case workloads with higher request rates and longer context lengths to evaluate behavior under saturation and memory pressure. These workloads show when baseline systems run out of GPU memory or experience severe latency degradation from KV-cache movement and staging overheads. We also vary batch size to cover both latency-critical interactive serving and throughput-oriented batch scenarios.

Baseline schemes: We compare DirectKV against three KV cache offloading systems:

- **SGLang** [58] is a high-performance serving framework for large language and multimodal models. We use SGLang as an HBM-resident baseline, where the KV cache is stored in GPU memory.
- **Pie** [53] uses a large GPU staging buffer to swap KV blocks from CPU memory to GPU via NVLink-C2C, overlapping transfers with compute.

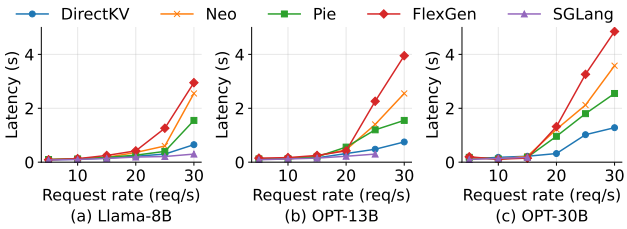


Figure 10: Per-token latency under varying request rate for different models.

- **FlexGen** [46] uses GPU, CPU memory, and disk with a cost-model search and 4-bit compression to reduce memory footprint and enable LLM inference under tight GPU memory constraints.
- **Neo** [34] offloads part of attention computation and KV cache to CPUs, using asymmetric GPU–CPU pipelining and load-aware scheduling to increase throughput.

7.3 End-to-end Performance

7.3.1 Per-token Latency

We evaluate three models under varying request rates, as shown in Fig. 10. SGLang achieves the lowest latency when the model and KV cache fit entirely in GPU memory, as it serves both parameters and KV cache from HBM without offloading overhead. However, this design is constrained by GPU memory capacity: SGLang supports all request rates for Llama-3.1-8B, but runs out of memory at the highest request rate for OPT-13B and beyond low request rates for OPT-30B. DirectKV achieves the lowest latency among offloading-based systems and remains close to SGLang when SGLang is feasible. On Llama-3.1-8B, DirectKV is slightly slower than SGLang but substantially faster than Neo, Pie, and FlexGen, reducing latency at 30 req/s from 1.55–2.95s to 0.75s. For OPT-13B, DirectKV remains competitive with SGLang at moderate loads and continues to support 30 req/s, where SGLang OOMs; at this load, DirectKV achieves 0.75s latency, compared with 1.55–3.95s for other offloading systems. For OPT-30B, SGLang only supports low request rates, while DirectKV scales to 30 req/s and maintains much lower latency than Neo, Pie, and FlexGen under high load.

These results show that SGLang provides the best performance when sufficient HBM is available, whereas DirectKV offers a better performance–capacity tradeoff. By using zero-copy KV offloading, DirectKV avoids staging buffers and redundant CPU–GPU copies, reducing offloading overhead and enabling graceful scaling when GPU-resident systems exceed HBM capacity.

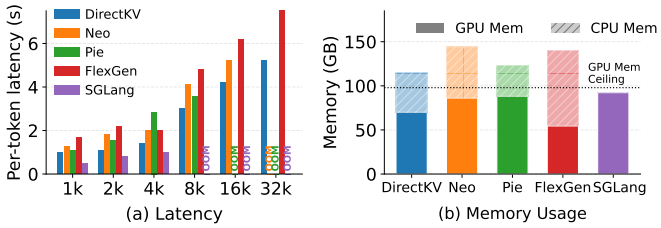


Figure 11: Performance and memory usage under different sequence length.

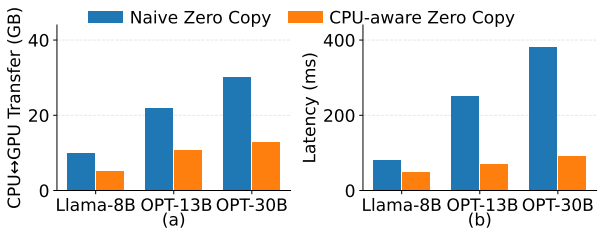


Figure 12: Data fetching and latency per token comparison between naive zero-copy and CPU-aware zero-copy.

7.3.2 Impact of Context Length on Latency

We evaluate per-token latency under different sequence lengths, as shown in Fig. 11(a).DirectKV consistently achieves the lowest latency among systems that support each context length, with a 1.2× average speedup. Its advantage increases as the sequence length grows: at 16k tokens, DirectKV is about 1.3× faster than Neo and Pie and about 1.7× faster than FlexGen. At 32k tokens, DirectKV remains efficient, while Neo, Pie, and SGLang run out of memory and FlexGen incurs much higher latency. SGLang is fast when all data fits in GPU memory, but fails at longer contexts due to GPU memory pressure. FlexGen supports longer contexts but suffers from explicit data movement and multi-tier offloading overhead. By keeping KV cache in CPU memory and directly accessing it through zero-copy execution, DirectKV reduces GPU memory usage while sustaining low latency for long-context serving.

7.3.3 GPU and CPU Memory Savings

We evaluate average GPU and CPU memory usage under varying sequence lengths across different LLM models on GH200, which has 96GB of GPU memory as shown in Fig.11(b). SGLang keeps both model parameters and KV cache in GPU memory, consuming 92GB on average and approaching the GPU memory limit as the sequence length grows. Although Neo, Pie, and FlexGen offload data to CPU memory, they still consume substantial GPU memory due to additional runtime buffers for KV-cache movement, using 86GB, 88GB, and 74 GB, respectively. In contrast, DirectKV uses only 47GB of GPU memory, saving 35GB on average compared with these offloading-based systems, which corresponds to a 43% reduction in GPU memory usage. This saving comes from storing the KV cache in CPU-pinned memory and allowing the attention kernel to directly consume CPU-resident KV blocks. As a result, DirectKV stays well below the 96 GB GPU memory ceiling and avoids SGLang's OOM behavior while achieving lower latency than other offloading-based systems.

7.4 Component-Level Evaluation

7.4.1 Benefits of CPU-Memory-Aware Matrix Multiplication

Naïve zero-copy attention performs matrix multiplications directly on KV blocks fetched from CPU memory, leading to repeated transfers of the same operands and excessive communication overhead. We evaluate our CPU-aware zero-copy mechanism, which reorganizes data access so that each KV block fetched from CPU memory is maximally reused in GPU registers and HBM before eviction. This design shifts the dominant pressure from the CPU-GPU interconnect to the GPU-side memory hierarchy, where HBM bandwidth is much higher and can better absorb the additional traffic. As shown in Fig. 12(a), CPU-aware zero-copy reduces CPU–GPU transfer volume by up to 50% compared with naive zero-copy. This reduction translates into up to 70% lower inference latency across LLaMA-8B, OPT-13B, and OPT-30B, as shown in Fig. 12(b). Overall, CPU-aware zero copy consistently improves end-to-end throughput by reducing communication cost and fully exploiting GPU compute.

7.4.2 Advantage of Fused Kernel

In existing offloading approaches K, and V projections are launched as separate kernels, followed by an attention kernel. This separation forces repeated refetching of KV blocks from CPU memory, inflating interconnect traffic and underutilizing GPU bandwidth. Our fused kernel addresses this inefficiency by combining QKV projection and attention into a single pass. Projected tensors remain in shared memory and are immediately consumed by attention, eliminating redundant global memory accesses. As shown in Fig. 13, the fused design sustains up to 3.5× higher HBM throughput and delivers 2.5–3.0× lower latency across Llama-8B, OPT-13B, and OPT-30B. These results demonstrate that kernel–memory co-design is essential to making zero-copy offloading practical and efficient on heterogeneous CPU–GPU platforms.

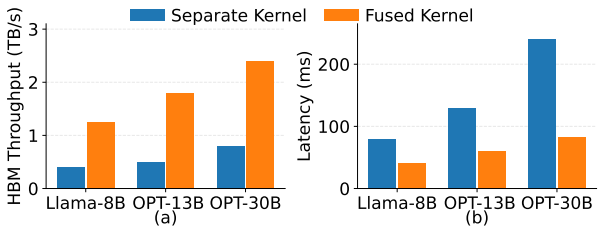


Figure 13: HBM throughput and latency per token comparison between separate kernel and fused kernel.

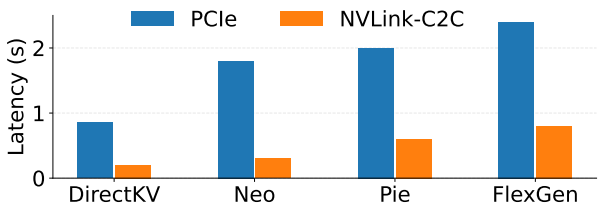


Figure 14: Latency per-token under 512 sequence length when using PCIe and NVLink-C2C.

7.4.3 Impact of High-bandwidth Interconnection

The efficiency of zero-copy offloading is fundamentally limited by the bandwidth of the CPU–GPU interconnect. With PCIe, KV blocks must traverse a lower-bandwidth channel, which can quickly become the system bottleneck and cause GPUs to stall while waiting for data. In contrast, NVLink-C2C provides much higher bandwidth, allowing DirectKV to sustain higher HBM utilization and better overlap communication with computation. Thus, on PCIe, DirectKV should be viewed primarily as a capacity-extension mechanism: it eliminates GPU staging buffers, but its performance advantage remains bounded by CPU–GPU interconnect bandwidth.

As shown in Fig. 14, DirectKV consistently achieves the lowest latency on both interconnects, with the gap especially pronounced under NVLink-C2C. Relative to PCIe, NVLink-C2C reduces attention latency by up to 4.2× and improves throughput stability under long contexts. Competing systems such as Neo, Pie, and FlexGen remain bottlenecked by staging overheads, yielding only modest improvements under NVLink-C2C.

8 Discussion

8.1 Integration with Existing Serving Systems

DirectKV is intended to complement, rather than replace, full-featured LLM serving systems such as vLLM and SGLang. These systems already provide mature mechanisms for continuous batching, prefix reuse, eviction, and hierarchical KV caching [23, 24, 32, 37, 58]. Such mechanisms determine how requests are scheduled, which KV blocks are reused or shared, and when blocks are retained, evicted, or promoted across memory tiers. DirectKV addresses a separate concern: how to execute attention efficiently once the relevant KV blocks reside in CPU memory.

This separation makes DirectKV naturally compatible with existing serving stacks. DirectKV does not change scheduling, batching, or KV-management policies; instead, it provides an efficient execution path for CPU-resident KV blocks. A serving system can integrate DirectKV by extending its KV allocator to support CPU-pinned storage for offloaded blocks, while leaving the scheduler, model execution graph, and frontend API unchanged. Therefore, DirectKV can directly benefit existing KV-cache optimizations that already place KV blocks in CPU memory, including prefix caching, eviction, and hierarchical KV management, by avoiding the copy back to HBM before attention.

8.2 Hardware Applicability

DirectKV is best suited to CPU–GPU platforms with high-bandwidth interconnects, such as NVLink-C2C on GH200/GB200. Its coarse-grained access pattern—∼100 KB KV blocks accessed concurrently by over 100 SMs—creates MB-scale aggregate transfers, making performance primarily limited by sustained CPU–GPU bandwidth rather than individual remote-load latency. Thus, on PCIe-only platforms, DirectKV may still reduce GPU memory usage, but throughput benefits are often limited by interconnect bandwidth.

Importantly, this bandwidth-oriented design does not rely on platform-specific coherence mechanisms. DirectKV uses zero-copy pinned memory rather than GH200's unified page table (UPT). Although UPT provides a coherent CPU–GPU virtual address space, DirectKV does not require coherence because KV blocks are not concurrently updated by the CPU. In our setting, zero-copy achieves comparable NVLink-C2C bandwidth while relying only on standard pinned host memory and device-visible pointers, making DirectKV applicable beyond UPT-specific platforms.

8.3 Scalability and Distributed Serving

DirectKV is primarily a node-local heterogeneous-memory optimization, but it composes with common LLM-serving parallelism strategies and distributed KV layouts. Under tensor parallelism, each GPU maintains its own attention-head shard and CPU-pinned KV pool; under pipeline parallelism, KV pools partition naturally by layer. When KV cache is partitioned across servers, each server can apply DirectKV to its local CPU-resident KV blocks, while the distributed runtime still handles query and partial-output communication. Thus, DirectKV improves per-server KV capacity, but does not eliminate inter-server communication costs.

DirectKV also benefits disaggregated prefill–decode serving because CPU-pinned KV blocks can be transferred via

RDMA without GPU staging. Its main limitations are context-parallel scale-out and host-memory pressure: coordinating remotely owned KV blocks or integrating with a multi-tier KV hierarchy would require additional runtime support, and zero-copy benefits diminish once accesses fall back to pageable or disk-backed tiers. We leave these extensions to future work.

9 Related Work

Efficient LLM Inference. A rich line of work has optimized transformer inference for latency and throughput. Systems such as TensorRT-LLM [3], vLLM [37], and DeepSpeed-Inference [21] accelerate execution through operator fusion, batching, and kernel optimizations. DistServe [59] disaggregates prefill and decode to reduce interference, while TurboTransformers [29] and HuggingFace [1] provide widely adopted runtimes. Sarathi-Serve [19] dynamically allocates prefill/decode resources, ZeRO-Offload [43] shifts optimizer states to CPU memory, PowerInfer [47] exploits activation sparsity, SGLang [58] optimizes multi-turn interaction, and FlashDecoding++ [33] provides low-latency decoding kernels. While effective within GPU memory constraints, none address KV cache offloading across CPU–GPU memory, the focus of DirectKV.

KV Cache Offloading. Many recent systems propose offloading the KV cache from GPU to CPU memory to support longer contexts. FlexGen [46] formulates tensor placement as a linear programming problem across GPU, CPU, and disk, using compression to increase throughput under resource constraints. NEO [34] employs asymmetric GPU–CPU pipelining and load-aware scheduling to balance computation and memory usage, achieving higher throughput under batch-serving settings. Pie [53] adopts a swap-based design, staging KV blocks in large GPU buffers transferred over NVLink-C2C. Infinite-LLM [38] orchestrates all available GPU and CPU memories across the data center to store the KV cache, but applications may experience degraded performance when data resides in CPU or remote memory. FastDecode [31] offloads the entire attention computation onto CPUs to avoid repeated PCIe transfers, but is constrained by limited CPU throughput. HeteGen [57] adopts heterogeneous parallelism with asynchronous overlap to mitigate I/O bottlenecks and reduce inference latency. While effective, these approaches either incur staging overhead, depend on software-level scheduling, or expose applications to unpredictable performance when CPU memory is involved. DirectKV instead eliminates staging buffers through a kernel–memory co-design, making zero-copy practical for attention computation.

GPU Kernels Optimization for Attention. FlashAttention-1 and 2 [25,25] optimize GPU kernel execution by tiling data in HBM, substantially reducing memory traffic and improving throughput. FlashAttention-3 [44] further improves performance on Hopper GPUs with advanced pipelining strategies. FlashInfer [54] targets heterogeneous KV-cache layouts with specialized CUDA kernels, leveraging block-sparse and composable formats to enhance memory efficiency and minimize redundancy. However, all of these optimizations assume that KV caches reside entirely in GPU memory and are not designed for zero-copy access to CPU-resident KV without HBM-resident buffers. When applied naively in a zero-copy setting (i.e., directly pointing K/V to host memory), these kernels suffer severe performance degradation because they do not account for the large bandwidth gap between CPU–GPU interconnects and HBM.

In contrast, DirectKV adapts FlashAttention-3-style kernels to heterogeneous CPU–GPU settings in which KV caches are stored in CPU memory, and mitigates the interconnect bottleneck through a kernel–memory co-design.

10 Conclusion

We proposed DirectKV, an efficient zero-copy KV cache offloading system for heterogeneous CPU–GPU architectures. Naïve zero-copy exacerbates the bandwidth gap between CPU–GPU interconnects and HBM, severely limiting performance. DirectKV addresses this by redesigning matrix multiplication access patterns to shift the bottleneck to HBM, while further improving efficiency through warp-level pipelining that overlaps communication with computation and fused kernels that eliminate redundant KV refetching. Evaluated on NVIDIA GH200, DirectKV reduces GPU memory usage by 43% and improves end-to-end performance by up to 1.2× over state-of-the-art systems.

Acknowledgements

We sincerely thank the anonymous reviewers and the shepherd, for their valuable feedback and suggestions. This work was supported in part by U.S. NSF grants NSF-2421782, NSF-2350425, NSF-2319988, and NSF-2206522; Microsoft Research Faculty Fellowship 8300751; an Amazon Research Award; AWS Cloud Credits for Research; and cloud computing credits provided by Lambda, Voltage Park and Vultr.

References

[1] Hugging face accelerate. https://huggingface.co/docs/accelerate/index, 2022.

[2] Nvidia pinned memory. https://docs.nvidia.com/cuda/cuda-c-programming-guide/#page-locked-host-memory, 2022.

[3] Nvidia tensorrt-llm. https://github.com/NVIDIA/TensorRT-LLM, 2022.

[4] Nvidia zero copy memory. https://docs.nvidia.com/cuda/cuda-c-programming-guide/#zero-copy-memory, 2022.

[5] Cuda tiling. https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#programming-interface, 2023.

[6] Huggingface dataset. https://huggingface.co/datasets, 2023.

[7] Nvidia l1/smem. https://developer.nvidia.com/blog/nvidia-hopper-architecture-in-depth/, 2023.

[8] Sharegpt. https://sharegpt.com/, 2023.

[9] Cuda memory management. https://docs.nvidia.com/cuda/cuda-runtime-api/group__CUDART__MEMORY.html, 2025.

[10] Hopper tensor memory accelerator. https://developer.nvidia.com/blog/nvidia-hopper-architecture-in-depth/, 2025.

[11] Meta llama-3-8b. https://huggingface.co/meta-llama/Meta-Llama-3-8B, 2025.

[12] Nvidia cutlass. https://github.com/NVIDIA/cutlass, 2025.

[13] Nvidia direct memory access engine tensor. https://docs.nvidia.com/gpudirect-storage/design-guide/index.html, 2025.

[14] Nvidia gb200. https://www.nvidia.com/en-us/data-center/gb200-nvl72, 2025.

[15] Nvidia gh200. https://www.nvidia.com/en-us/data-center/grace-hopper-superchip/, 2025.

[16] Nvidia h100. https://www.nvidia.com/en-us/data-center/h100/, 2025.

[17] Nvidia cuda copy operation. https://docs.nvidia.com/cuda/parallel-thread-execution/, 2026.

[18] Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, et al. Gpt-4 technical report. *arXiv preprint arXiv:2303.08774*, 2023.

[19] Amey Agrawal, Nitin Kedia, Ashish Panwar, Jayashree Mohan, Nipun Kwatra, Bhargav Gulavani, Alexey Tumanov, and Ramachandran Ramjee. Taming {Throughput-Latency} tradeoff in {LLM} inference with {Sarathi-Serve}. In *Proceedings of OSDI*, 2024.

[20] Joshua Ainslie, James Lee-Thorp, Michiel De Jong, Yury Zemlyanskiy, Federico Lebrón, and Sumit Sanghai. Gqa: Training generalized multi-query transformer models from multi-head checkpoints. *arXiv preprint arXiv:2305.13245*, 2023.

[21] Reza Yazdani Aminabadi, Samyam Rajbhandari, Ammar Ahmad Awan, Cheng Li, Du Li, Elton Zheng, Olatunji Ruwase, Shaden Smith, Minjia Zhang, Jeff Rasley, et al. Deepspeed-inference: enabling efficient inference of transformer models at unprecedented scale. In *Proceedings of IEEE SC*, 2022.

[22] Hopper GPU Architecture. https://docs.nvidia.com/cuda/hopper-tuning-guide/index.html, 2025.

[23] Automatic Prefix Caching. https://docs.vllm.ai/en/v0.18.0/features/automatic_prefix_caching/, 2026.

[24] Yihua Cheng, Yuhan Liu, Jiayi Yao, Yuwei An, Xiaokun Chen, Shaoting Feng, Yuyang Huang, Samuel Shen, Kuntai Du, and Junchen Jiang. Lmcache: An efficient kv cache layer for enterprise-scale llm inference. *arXiv preprint arXiv:2510.09665*, 2025.

[25] Tri Dao. Flashattention-2: Faster attention with better parallelism and work partitioning. 2024.

[26] Tri Dao, Dan Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. Flashattention: Fast and memory-efficient exact attention with io-awareness. 2022.

[27] Weishu Deng, Yujie Yang, Peiran Du, Lingfeng Xiang, Zhen Lin, Chen Zhong, Song Jiang, Hui Lu, and Jia Rao. Hgca: Hybrid gpu-cpu attention for long context llm inference. *arXiv preprint arXiv:2507.03153*, 2025.

[28] Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Amy Yang, Angela Fan, et al. The llama 3 herd of models. *arXiv e-prints*, 2024.

[29] Jiarui Fang, Yang Yu, Chengduo Zhao, and Jie Zhou. Turbotransformers: an efficient gpu serving system for transformer models. In *Proceedings of PPoPP*, 2021.

[30] Amir Gholami, Zhewei Yao, Sehoon Kim, Coleman Hooper, Michael W Mahoney, and Kurt Keutzer. Ai and memory wall. *IEEE Micro*, 2024.

[31] Jiaao He and Jidong Zhai. Fastdecode: High-throughput gpu-efficient llm serving using heterogeneous pipelines. *arXiv preprint arXiv:2403.11421*, 2024.

[32] Hierarchical KV Caching (HiCache). https://docs.sglang.io/docs/advanced_features/hicache, 2026.

[33] Ke Hong, Guohao Dai, Jiaming Xu, Qiuli Mao, Xiuhong Li, Jun Liu, Kangdi Chen, Yuhan Dong, and Yu Wang. Flashdecoding++: Faster large language model inference with asynchronization, flat gemm optimization, and heuristics. *Proceedings of MLSys*, 2024.

[34] Xuanlin Jiang, Yang Zhou, Shiyi Cao, Ion Stoica, and Minlan Yu. Neo: Saving gpu memory crisis with cpu offloading for online llm inference. 2025.

[35] Jared Kaplan, Sam McCandlish, Tom Henighan, Tom B Brown, Benjamin Chess, Rewon Child, Scott Gray, Alec Radford, Jeffrey Wu, and Dario Amodei. Scaling laws for neural language models. *arXiv preprint arXiv:2001.08361*, 2020.

[36] Nikita Kitaev, Łukasz Kaiser, and Anselm Levskaya. Reformer: The efficient transformer. 2020.

[37] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph Gonzalez, Hao Zhang, and Ion Stoica. Efficient memory management for large language model serving with pagedattention. In *Proceedings of SOSP*, 2023.

[38] Bin Lin, Chen Zhang, Tao Peng, Hanyu Zhao, Wencong Xiao, Minmin Sun, Anmin Liu, Zhipeng Zhang, Lanbo Li, Xiafei Qiu, et al. Infinite-llm: Efficient llm service for long context with distattention and distributed kvcache. *arXiv preprint arXiv:2401.02669*, 2024.

[39] Yuhan Liu, Hanchen Li, Yihua Cheng, Siddhant Ray, Yuyang Huang, Qizheng Zhang, Kuntai Du, Jiayi Yao, Shan Lu, Ganesh Ananthanarayanan, et al. Cachegen: Kv cache compression and streaming for fast large language model serving. In *Proceedings of SIGCOMM*, 2024.

[40] Maxim Milakov and Natalia Gimelshein. Online normalizer calculation for softmax. *arXiv preprint arXiv:1805.02867*, 2018.

[41] Daon Park and Bernhard Egger. Improving throughput-oriented llm inference with cpu computations. In *Proceedings of PACT*, 2024.

[42] Ruoyu Qin, Zheming Li, Weiran He, Jialei Cui, Feng Ren, Mingxing Zhang, Yongwei Wu, Weimin Zheng, and Xinran Xu. Mooncake: Trading more storage for less computation—a {KVCache-centric} architecture for serving {LLM} chatbot. In *Proceedings of FAST*, pages 155–170, 2025.

[43] Jie Ren, Samyam Rajbhandari, Reza Yazdani Aminabadi, Olatunji Ruwase, Shuangyan Yang, Minjia Zhang, Dong Li, and Yuxiong He. {Zero-offload}: Democratizing {billion-scale} model training. In *Proceedings of ATC*, 2021.

[44] Jay Shah, Ganesh Bikshandi, Ying Zhang, Vijay Thakkar, Pradeep Ramani, and Tri Dao. Flashattention-3: Fast and accurate attention with asynchrony and low-precision. 2024.

[45] Noam Shazeer. Fast transformer decoding: One write-head is all you need. *arXiv preprint arXiv:1911.02150*, 2019.

[46] Ying Sheng, Lianmin Zheng, Binhang Yuan, Zhuohan Li, Max Ryabinin, Beidi Chen, Percy Liang, Christopher Ré, Ion Stoica, and Ce Zhang. Flexgen: High-throughput generative inference of large language models with a single gpu. In *Proceedings of ICML*, 2023.

[47] Yixin Song, Zeyu Mi, Haotong Xie, and Haibo Chen. Powerinfer: Fast large language model serving with a consumer-grade gpu. In *Proceedings of SOSP*, 2024.

[48] Jianlin Su, Murtadha Ahmed, Yu Lu, Shengfeng Pan, Wen Bo, and Yunfeng Liu. Roformer: Enhanced transformer with rotary position embedding. *Neurocomputing*, 2024.

[49] Rohan Taori, Ishaan Gulrajani, Tianyi Zhang, Yann Dubois, Xuechen Li, Carlos Guestrin, Percy Liang, and Tatsunori B. Hashimoto. Stanford alpaca: An instruction-following llama model. https://github.com/tatsu-lab/stanford_alpaca, 2023.

[50] Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, et al. Llama: Open and efficient foundation language models. *arXiv preprint arXiv:2302.13971*, 2023.

[51] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. *Advances in neural information processing systems*, 30, 2017.

[52] Yizhong Wang, Yeganeh Kordi, Swaroop Mishra, Alisa Liu, Noah A. Smith, Daniel Khashabi, and Hannaneh Hajishirzi. Self-instruct: Aligning language model with self generated instructions, 2022.

[53] Yi Xu, Ziming Mao, Xiangxi Mo, Shu Liu, and Ion Stoica. Pie: Pooling cpu memory for llm inference. *arXiv preprint arXiv:2411.09317*, 2024.

[54] Zihao Ye, Lequn Chen, Ruihang Lai, Wuwei Lin, Yineng Zhang, Stephanie Wang, Tianqi Chen, Baris Kasikci, Vinod Grover, Arvind Krishnamurthy, et al. Flashinfer: Efficient and customizable attention engine for llm inference serving. 2025.

[55] Chengye Yu, Tianyu Wang, Zili Shao, Linjie Zhu, Xu Zhou, and Song Jiang. Twinpilots: A new computing paradigm for gpu-cpu parallel llm inference. In *Proceedings of Systor*, 2024.

[56] Susan Zhang, Stephen Roller, Naman Goyal, Mikel Artetxe, Moya Chen, Shuohui Chen, Christopher Dewan, Mona Diab, Xian Li, Xi Victoria Lin, et al. Opt: Open pre-trained transformer language models. *arXiv preprint arXiv:2205.01068*, 2022.

[57] Xuanlei Zhao, Bin Jia, Haotian Zhou, Ziming Liu, Shenggan Cheng, and Yang You. Hetegen: Efficient heterogeneous parallel inference for large language models on resource-constrained devices. In *Proceedings of MLSys*, 2024.

[58] Lianmin Zheng, Liangsheng Yin, Zhiqiang Xie, Chuyue Livia Sun, Jeff Huang, Cody Hao Yu, Shiyi Cao, Christos Kozyrakis, Ion Stoica, Joseph E Gonzalez, et al. Sglang: Efficient execution of structured language model programs. 2024.

[59] Yinmin Zhong, Shengyu Liu, Junda Chen, Jianbo Hu, Yibo Zhu, Xuanzhe Liu, Xin Jin, and Hao Zhang. {DistServe}: Disaggregating prefill and decoding for goodput-optimized large language model serving. In *Proceedings of OSDI*, 2024.

ARTIFACT EVALUATED
USENIX
AVAILABLE

Simple is Better: Multiplication May Be All You Need for LLM Request Scheduling

Dingyan Zhang[†1], Jinbo Han[†,‡1], Kaixi Zhang[†,‡1], Xingda Wei[✉1], Sijie Shen[2], Chenguang Fang[2], Wenyuan Yu[2], Jingren Zhou[2], Rong Chen[1]

[1]Institute of Parallel and Distributed Systems, Shanghai Jiao Tong University [2]Alibaba Group

Abstract

High-quality LLM request scheduling requires meeting two key objectives: ensuring the routed instance has KV$ to accelerate request execution, and ensuring that the workload is balanced across instances. Achieving both objectives is challenging because pursuing one may compromise the other. Current approaches use various combinators (e.g., linear combinations) to compute a scheduling score that combines indicators for the two objectives. These approaches are complex: they either require significant workload-specific hyperparameter tuning or model-hardware-aware simulator development, yet could still lead to suboptimal performance.

In this paper, we show that using a simple multiplication of two carefully chosen indicators—one KV$-aware (new prefill tokens if routed to an instance) and one load-balancing-aware (current batch size of the instance)—as the scheduling score (LMETRIC[*]) can achieve both objectives simultaneously without any hyperparameter tuning. The key idea is that the simply multiplied score considers both objectives in a manner similar to a linear combination, but the original hyperparameters cancel out during comparison, so no tuning is needed to find the best parameters. The two indicators are chosen based on our analysis of LLM characteristics. Our extensive experiments show that this simple approach can reduce TTFT by 92% and 39%, and TPOT by 24% and 51%, compared to vLLM-v1 and an in-production scheduler on real-world workloads covering chatbots and coding agents. We also derive the mathematical conditions under which multiplication may fail, and find that such conditions are extremely rare in practice and can be detected (and mitigated) beforehand.

LMETRIC has been deployed in production and canary release confirms its effectiveness.

1 Introduction

This paper studies how to efficiently route LLM requests to a cluster of serving instances—the minimal LLM engine deployment unit. Serving LLMs has become a key building block in modern society, and LLM providers typically deploy clusters of instances for serving, where each cluster contains a *global scheduler* that routes incoming requests to the instances it manages [40, 50, 51, 24, 35, 6, 45]. Upon receiving a request, the instance generates result tokens in two phases: The prefill (P) phase generates the first result token, and its serving quality is measured by time-to-first-token (TTFT). The decode (D) phase then generates the remaining tokens in a streaming manner, and its serving quality is measured by time-per-output-token (TPOT).

Providing an effective scheduling policy is crucial for cluster-level LLM serving. This is because, similar to traditional request routing [17, 54, 22], better request placement significantly reduces overall latency (lower TTFT and TPOT) due to factors such as improved load balancing across instances. Low-latency serving is especially critical for current interactive applications such as ChatGPT [37] and copilots [21], as it is essential to meeting user expectations [9, 55, 20]. Moreover, recent agentic workloads consume tokens rapidly through interactive programs [8, 23, 36].

Achieving a good LLM-specific scheduling policy is non-trivial: First, considering only load balancing across instances—which is adopted by a recent state-of-the-art serving system vLLM [45] and traditional request routing—is insufficient. This is because the computation required to process each request differs across instances due to KV$, the intermediate context of the processed tokens (§4.2). Specifically, if an incoming request's (partial) input tokens hit the KV$ cached on an instance, the instance can skip generating the corresponding KV$ for the hit tokens, thereby accelerating the subsequent prefill and decode phases. However, incorporating only KV$-aware indicators into scheduling decisions (e.g., the KV$ hit ratio if routing a request to an instance) is also insufficient, because it biases requests towards instances with KV$ hits and hurts load balancing across instances (§4.3).

To balance the two objectives—KV$-awareness and load balancing—three combination strategies exist.

First, the linear combination strategy (i.e., weighted sum) [29, 35] combines the indicators for each objective—per-instance metrics that score each objective if the request were routed to that instance, e.g., the KV$ hit ratio for KV$-awareness and the current batch size for load balancing—into a single score for scheduling (§4.4). It is a popular choice adopted by current works and one of the world's leading LLM service providers (Alibaba BAILIAN). However, linear combination requires complex workload-specific hyperparameter tuning to achieve both objectives. Moreover, a statically tuned hyperparameter is suboptimal because workloads may change

[*]Our system name LMETRIC stands for **L**arge **M**odel metric, paying homage to Lyapunov's stability theory and Markov's queueing theory.

[†]Most work done when intern at Alibaba Group.

[‡]Jinbo Han and Kaixi Zhang contributed equally and significantly to this work.

[✉]Xingda Wei is the corresponding author (wxdwfc@sjtu.edu.cn).

dynamically (§4.4).

Second, the filter-based strategy first filters out instances that are suspected to suffer from imbalanced workloads, and then selects the instance with the most KV$ hits among the remaining instances [3]. It still requires non-trivial workload-specific tuning to determine the filtering threshold. Worse still, its scheduling is biased towards load balancing and thus cannot fully utilize the KV$ (§4.5).

Finally, the simulation-based strategy [40, 24, 56, 13] first uses a simulator to predict the expected latency of routing a request to each instance, and then takes the latency as the scheduling score. The simulator estimates the latency based on its current indicators, e.g., stored KV$ and request load in progress (queued requests), so the latency score can be viewed as a high-order combination of the KV$ and load-balance indicators to achieve the best of both worlds. However, the effectiveness of the strategy relies heavily on the accuracy of the simulator, which requires complex per-model, per-hardware, and per-deployment development. Otherwise, an inaccurate simulation can lead to poor scheduling performance (§4.6). Even with an accurate simulator, it may still fail to achieve performance comparable to other candidates in certain cases.

In this paper, we show that using the multiplication of one indicator for KV$-awareness and one indicator for load balancing as the scheduling score can effectively combine the two objectives without complex hyperparameter tuning or any simulator. The key idea is to replace the addition operation in a linear combination with multiplication. The resulting score preserves a trend similar to that of a linear combination, and the hyperparameters cancel out during score comparisons among instances, so no tuning is required (§5).

Making this simple method work well in practice requires care. First, we found that careful indicator selection can further improve the method's effectiveness (§5.1). For example, using the number of queued new prefill tokens when routing a request to an instance, considering KV$ hits, as the KV$-aware indicator is better than using the KV$ hit ratio. Second, we mathematically derive the approximate conditions under which multiplication may fail. Based on the formulated conditions, we found that they are extremely rare in practice (§5.2)—occurring under extreme KV$ skewness that compromises load balance. Hence, we further design a two-phase approach to detect and mitigate it. Upon detection, we can fall back to a load-balancing-only policy.

We have compared our method with state-of-the-art systems, including vLLM [45], ai-Dynamo [35], llm-d [24], and the one used in Alibaba BAILIAN [15] on real LLM serving workloads covering chatbots, API calling, and coding agents (§6). On an H20 cluster with up to 16 GPUs, evaluation on popular models covering both dense and MoE architectures confirms the benefits of our approach. LMETRIC has been deployed in production at BAILIAN on hundreds of GPUs, and performance observed from a canary release confirms its effectiveness.

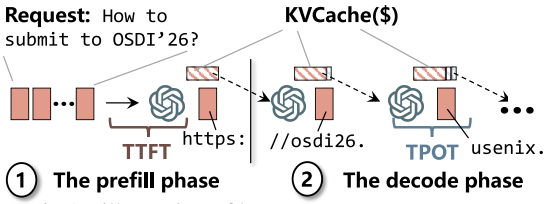


Figure 1: An illustration of how an LLM generates output tokens, and the two performance metrics: time-to-first-token (TTFT) and time-per-output-token (TPOT).

Contributions. This paper makes three contributions:

- The first systematic study of how to efficiently schedule LLM serving requests in a cluster (§4).
- The first multiplicative combination for efficient LLM request scheduling (§5).
- Extensive analysis and evaluation that confirm the effectiveness of the multiplicative approach (§5.1, §5.2, §6).

Our code is open-sourced at https://github.com/blitz-serving/blitz-router, and all our traces can be found at https://github.com/alibaba-edu/qwen-bailian-usagetraces-anon.

Discussion: PD-colocation vs. PD-disaggregation. We focus on PD-colocated serving in this paper, where both prefill and decode requests are served on the same instance. While there also exist deployments where prefill and decode requests are served on different instances (PD-disaggregation) [38, 55, 26], PD-colocated serving is still widely adopted in practice [50] because it is easier to maintain (no instance role management), does not rely on fast networking between instances, and yields better performance under certain conditions [31, 30]. We discuss how our observations and solutions apply to PD-disaggregation in §7.

2 LLM Serving and Scheduling

Serving requests with an LLM (Figure 1). LLMs generate tokens auto-regressively in two steps: ① In the *prefill* phase, the input tokens are fed into the model to produce the first output token. The LLM then enters the *decode* phase (②) to generate subsequent tokens one-by-one, each conditioned on all prior input and generated tokens. To accelerate decode, the *processed context* is materialized as tensors in GPU memory, termed the *key-value cache* (KV$). Consequently, decode computation is far smaller than prefill, since the KV$ for input tokens is already available and not regenerated.

Two key serving-quality metrics are *time-to-first-token (TTFT)* and *time-per-output-token (TPOT)*: TTFT determines the service's responsiveness to user requests, while TPOT affects both subsequent responsiveness and overall request completion time.

Serving instance and KV$ cache (Figure 2). An instance is the minimum unit that serves requests and hosts one complete copy of the LLM's parameters, potentially spanning multiple

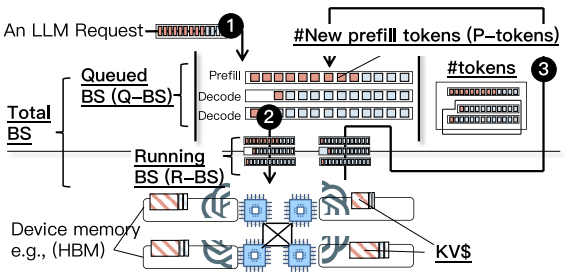


Figure 2: The system view of how an LLM serving instance handles requests and representative direct system **indicators** that can be collected by the global scheduler. The detailed meaning of each indicator will be described at first use. **BS** is the abbreviation for batch size.

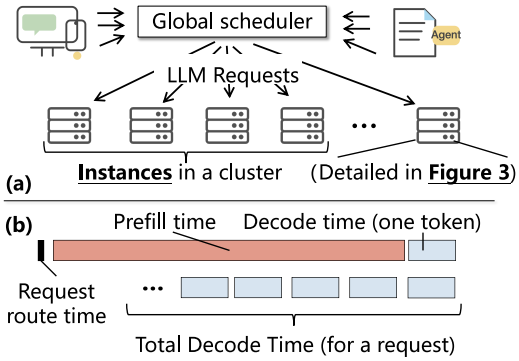


Figure 3: (a) System view of an LLM serving cluster, and (b) Comparison of per-request serving time and routing cost.

GPUs if the model exceeds a single GPU's memory. The serving flow is the same in either case: an incoming request is pushed into a queue (prefill or decode) on the instance (❶). Once the GPU(s) become available, queued requests are batched and executed efficiently with *chunked prefill* [2] (❷). The instance then re-enqueues any request that needs further decoding (❸).

Serving is *stateful*: a request's KV$ is cached in GPU or CPU memory even after generation finishes (KV$ cache [40, 19, 47]). If a future request routed to the instance shares a prefix with a cached KV$, the instance can skip computing the matched prefix tokens, significantly reducing computation cost. In Figure 2, blue tokens skip computation on a KV$ hit.

Request scheduling in a cluster (Figure 3). To handle large request volumes, providers deploy clusters of instances, each with a dedicated *global scheduler* that routes requests to its managed instances, as shown in (a). The scheduler runs a *scheduling policy* to pick a destination instance per request—our focus. Providers may deploy multiple clusters of the same model for reliability and geo-affinity, but inter-cluster routing is out of the scope of this paper.

All existing scheduling policies follow a three-step process: the router first (optionally) *filters* instances, then *scores* the remainder by a preference order, and routes the request to the best-scoring one. Scores derive from per-instance indicators, described in §4.

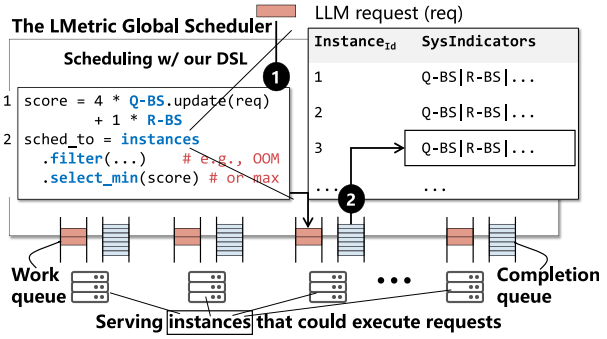


Figure 4: The system architecture of our indicator factory and its programming model for scheduling policies.

3 The Analysis Framework

To analyze different scheduling policies in an apples-to-apples manner, we implement a flexible LLM scheduling analysis framework. Two drivers shape its design. (1) Existing policies are buried in concrete open- and closed-source serving-system implementations, making apples-to-apples comparison hard. For example, AIBrix's [3] Go reimplementation of vLLM's policy runs 6.2 × faster than vLLM's Python version [45] due to a later-confirmed performance bug [46]; our Rust implementation is a further 1.2 × faster than AIBrix on the same policy. (2) Though complex, all existing policies boil down to computing scheduling scores from per-instance indicators. A unified indicator factory thus lets developers implement and explore new policies in a few lines of code (described below).

Indicator factory. Our framework is a standalone inference router written in Rust that works with any LLM serving engine; Rust keeps it efficient and robust. Its core component, shown in Figure 4, is an *indicator factory* that automatically collects and, when needed, computes the indicators each scheduling policy requires. For scalability, the factory piggybacks indicator collection on instance responses: the router maintains a long-lived connection to each engine instance and, on every response, extracts the required indicators from the response header and updates the factory.

Programming model. Our framework provides a simple API to implement different scheduling policies. The API lets developers express scheduling as a function over the factory's symbolic per-instance indicators in a few lines. On top of the score function, we further provide primitives to filter instances and to select the instance with the minimum (or maximum) score. Line 1 of Figure 4 shows the policy adopted by vLLM [45] using our framework: the score is a weighted sum of `Q-BS` (queued batch size—the number of queued requests in an instance's queue) and `R-BS` (the number of running requests within batch). On a new request (line 4), the router retrieves and computes the indicator values from the factory in parallel, derives a score for every instance, and routes the request to the instance with the minimum score.

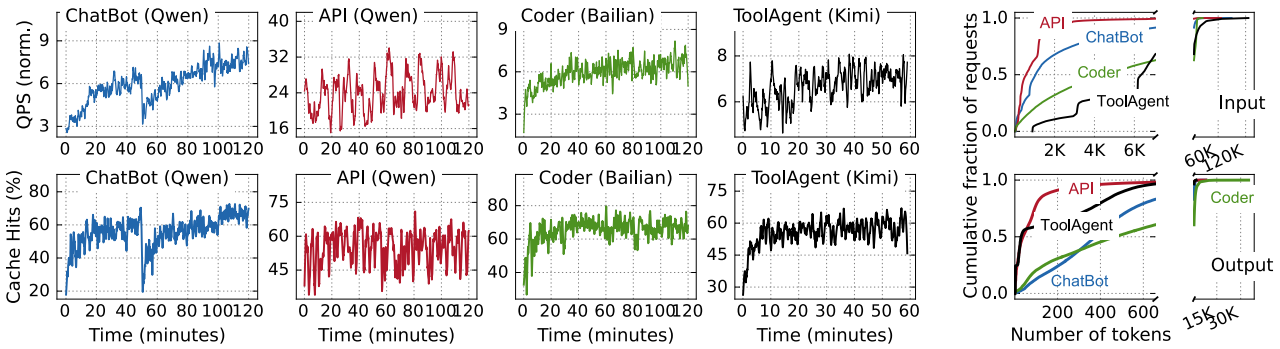


Figure 5: Our studied traces that cover major scenarios in powering LLM services.

4 Characterizing LLM Request Scheduling

4.1 Characterization Methodology

Testbed and instance used. Unless otherwise specified, all experiments run on a testbed with 16 NVIDIA H20 GPUs—hardware similar to that used for hosting LLM services at BAILIAN. Each GPU has 96 GB HBM, sufficient for common models available on the market [50]. The router runs on a high-end CPU server with 160 Intel Xeon cores and 1 TB of DRAM. Our instance runs vLLM-v1 (vLLM) [45]—the state-of-the-art LLM serving engine with the latest optimizations, such as chunked prefill [2] and fast GPU kernels [18].

Models. We chose LLM models representative of different architectures and popular market choices, including dense (Qwen2-7B) and mixture-of-experts (MoE) models (Qwen3-30B) [50].

Workloads. We analyze real-world LLM serving traces, both open-source and collected from BAILIAN, covering common LLM applications: **ChatBot (Qwen)** and **Agent (Qwen)** [11] are open-sourced Alibaba Cloud traces that collect requests from a chatbot service similar to ChatGPT and an LLM API calling agent service [8, 23, 36], respectively. **Coder** collects requests issued by coding agent services to a dedicated cluster in BAILIAN on a single day in November 2025, and **ToolAgent (Kimi)** [34] is another open-sourced trace from Kimi that collects requests from an agent service. All traces except ToolAgent (Kimi) are collected from a single cluster routed by one global router; the source of ToolAgent (Kimi) does not specify whether it is collected the same way.

Our analyzed workloads are representative: they span broad application scenarios, and each selected trace preserves the essential characteristics for evaluating LLM scheduling policies. Specifically, all requests in our traces contain the (hashed) content and request-issuance timestamp, which are critical for evaluating the impact of KV$-aware scheduling on global scheduling (see §4.2). Other popular datasets like AzureLLM-Trace [10] or BurstGPT [49] do not provide such content. Note that with traces containing hashed content, we can still replay them with behavior that exactly matches the original [48]. To facilitate future research, our high-performance trace replayer is open-sourced [12].

Figure 5 visualizes key features of our evaluated traces, including their request arrival rates, input and output token numbers, and the KV$ hit rates assuming an infinite KV$ space. The request arrival rate is normalized due to confidentiality considerations required for the **Coder** trace. Across all traces, over a given serving interval (e.g., 1 hour), the request arrival and KV$ hit rates are relatively stable, with a few short-term fluctuations. Input and output token numbers vary across traces but are typically modest except for a few outliers.

Trace scaling. Since the traces are collected from clusters at different scales than our testbed, we scale them according to our testbed capability similar to prior work [33, 7, 25, 53, 42]. Unless otherwise specified, we scale the average request arrival rate to half of the maximum rate of our testbed obtained via offline profiling. This approximates the serving configurations in BAILIAN because when the arrival rate approaches serving capacity, BAILIAN commonly reroutes requests to another underloaded cluster or simply rejects them in non-critical cases (e.g., ChatApp) [40]. Otherwise, the service-level objectives (SLOs) of many requests cannot be met due to queuing [39]. Our end-to-end analysis in §6 further measures the impact of different request arrival rates on scheduling performance, and shows consistent results.

4.2 Load-balancing Alone is Insufficient for LLM

A starting case: the vLLM policy [45]. Our study starts with the default global scheduling policy adopted by vLLM [45]—a popular open-source LLM serving engine widely used in both industry and academia. Figure 6 (a) shows its scheduling method, which uses the batch size of each instance as the indicator for the routing score. It is a variant of the classic load-balancing-centric join-the-shortest-queue (JSQ) policy with an extension for LLM: at each instance, the batch size includes both the requests running on the instance (R-BS) and those queued in the instance's queue (Q-BS).

Retrofitting vLLM with KV$-awareness (BAILIAN). While JSQ tries to balance the workload across instances, it is unaware of the KV$ state of the incoming requests. This

(a) vLLM

```
req = receive()
## one score per instance
score = 4 * Q-BS + 1 * R-BS
sched_to = instances
           .select_min(score)
req.forward(sched_to)
```

(b) vLLM + KV$-awareness (e.g., Bailian-like)

```
req = receive()
kv_hit = KV$.match(req)
score  = λ * (1-kv_hit) + (1-λ) * norm(BS)
sched_to = instances
           .select_min(score)
req.forward(sched_to)
```

Range [0,1]

Figure 6: (a) The scheduling score of vLLM and (b) adding KV$-awareness to its score for LLM scheduling. Note that the linear combination of two indicators has only one degree of freedom (the weight λ).

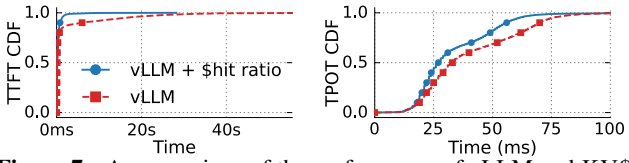


Figure 7: A comparison of the performance of vLLM and KV$-aware scheduling on ChatBot Trace with Qwen3-30B model. Other workloads and models are similar.

matters because routing requests to instances with a higher KV$ hit rate substantially reduces request latency. Figure 6 (b) shows a simple extension of vLLM adopted by BAILIAN and others [35, 6] to make request scheduling KV$-aware: it adds a KV$ indicator to the score function—an estimate of the KV$ hit ratio if the request is routed to that instance. Here, KV is the symbolic value representing the per-instance KV$ hash map. There are two points to note here: (1) the load balance indicator (batch size) needs to be normalized to $[0, 1]$ to match the scale of the hit ratio because otherwise they cannot simply be added together; (2) the analysis in this section assumes the linear combination coefficients (λ) are fixed, and the next section discusses the rationale for setting them.

As shown in Figure 7, adding KV$-awareness to a load-balancing-only policy improves the average TTFT by 84% and the average TPOT by 17%, which is as expected because it increases the KV$ hit ratio as profiled in Figure 8. Interestingly, although the improved KV$ hit ratio—at first glance—is only beneficial to the prefill, we found it also reduces the decode time because it reduces the computation required for each instance, allowing the instance to dedicate more GPU time to the decode phase.

4.3 KV$-awareness vs. Load balancing: The Trade-off

Although adding KV$-awareness is intuitively beneficial (§4.2), realizing this benefit in practice is non-trivial because the KV$ objective may interfere with load balancing. With a linear combination, the priority of each objective is controlled by the weight assigned to each indicator (λ in Figure 6 (b)): a larger weight on the KV$ component prioritizes routing

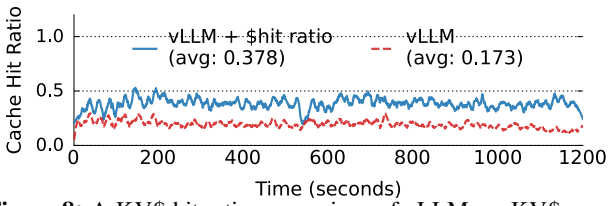


Figure 8: A KV$ hit ratio comparison of vLLM vs. KV$-aware scheduling on ChatBot Trace with Qwen3-30B model. Other workloads and models are similar.

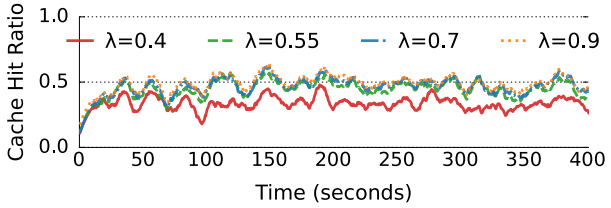


Figure 9: A comparison of the KV$ hit ratio by changing the weight of KV$-awareness in the policy described in Figure 6 (b) on ChatBot Trace with Qwen3-30B model.

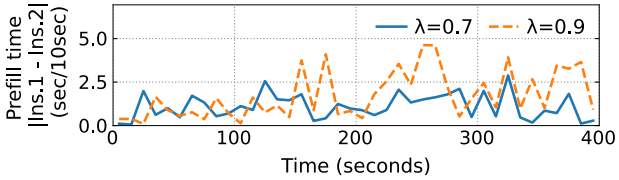


Figure 10: A profile of the workload imbalance between two instances when running with two different weights in a linear combination on the ChatBot Trace using the Qwen3-30B model. The reported metric is the absolute served prefill time in each 10-second window between the two instances (Inst.).

requests to instances with higher hit ratios even when other instances are much less loaded, leading to load imbalance.

Figure 11 illustrates this trade-off, reporting the overall TTFT and TPOT by sweeping weights. We can see that when increasing the weight from 0.4 to 0.9, the TTFT first gradually decreases and then increases, except for the API trace, which is less affected due to its short input length. The increased KV$ hit ratio explains this: as shown in Figure 9, on the ChatBot trace (other traces are similar) the hit ratio rises accordingly with the increased weight.

Despite the increased KV$ hit ratio, there is a knee point in the weight (e.g., 0.7 for ChatBot), beyond which the overall latency starts to increase due to load imbalance across instances. Figure 10 profiles this imbalance, plotting the prefill work assigned to two instances under different weights (0.7 vs. 0.9) over the same 400-second burst period on the ChatBot trace. The prefill time—seconds spent on prefill within each 10-second window—measures the workload per instance: since most tokens are generated during decode, an instance dominated by prefill produces fewer tokens than others. For each setup, we select the two instances (out of 16) with the highest standard deviation of prefill time. We observe that under $\lambda = 0.9$ the average prefill time differs significantly between the two instances (3.57s vs. 2.17s), while $\lambda = 0.7$ yields

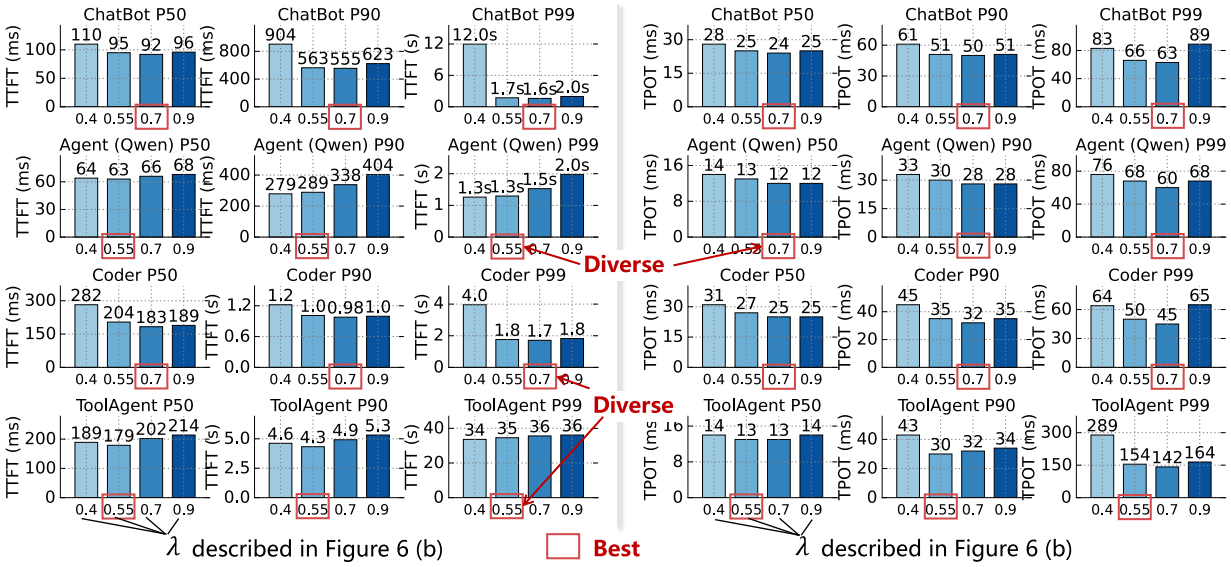


Figure 11: An analysis of how the performance varies with different hyperparameters on four traces with a linear-combination-based method. The model used is Qwen3-30B.

similar values (3.43s vs. 3.40s).

4.4 The Case of Linear Combination

Based on the trade-offs explored previously, using a linear combination to achieve both KV$-awareness and load balancing leads to two issues:

Cons #1. Requires workload-specific hyperparameter tuning. The importance of KV$ and its impact on load imbalance are workload-dependent. For example, Figure 11 presents the tuning results for different traces when running a Qwen3-30B model. We can see that the evaluated optimal weight for each workload varies: in ChatBot the optimal weight is 0.7, while in API it is 0.55, even though both workloads have a similar KV$ access pattern. Note that we cannot afford to sweep all possible configurations, as replaying each trace on the testbed consumes a substantial amount of GPU time. As a result, the linear combination requires workload-specific hyperparameter tuning that is non-trivial in practice due to the diversity of workloads [47].

Cons #2. Sub-optimal performance. During the evaluation in §6, we found that a statically tuned weight cannot always achieve competitive performance compared to other baselines. We hypothesize that this is because the optimal weight may vary over time. Intuitively, if the GPUs are idle, we need to prioritize KV$ hits with a larger KV$ weight. On the other hand, if prioritizing KV$ results in an imbalance, we need to reduce the weight to improve load balancing. However, to the best of our knowledge, all existing work uses a fixed tuned weight for the entire serving duration. While it is possible to design strategies to adaptively tune the weight over time, doing so would add system complexity.

4.5 The Case of Filter-based Combination

Methodology. Figure 13 shows how a typical filter-based combination works for integrating KV$-awareness and load balancing in LLM scheduling systems such as AIBrix [3]. First, the router checks whether the current cluster has an imbalanced load—i.e., the range between the maximum and minimum batch sizes across instances (`BS.max() - BS.min()`) exceeds a threshold (line 3). If so, the router abandons KV$-awareness and simply routes the request to the instance with the smallest batch size for load balancing (lines 4–5). Otherwise, the router uses the KV$ hit ratio as an indicator to route requests to instances (lines 6–9) for KV$-awareness.

Cons #1. Still requires workload-specific hyperparameter tuning. Similar to linear combination, filter-based methods also require hyperparameter tuning because the threshold for determining load imbalance (`Range`) is workload-dependent. As shown in Figure 12, the optimal threshold of a typical filter-based method [3] varies across workloads: for example, in Coder, increasing the threshold from 4 to 16 improves the P50 TTFT and TPOT by 44% and 15%, respectively. On the other hand, for the API trace, 16 is a better choice than 4.

Cons #2. Sub-optimal performance. Besides hyperparameter tuning, filter-based combination is slower than linear combination with properly tuned weights, as shown in Figure 12, because it biases towards load balancing and may forgo the benefits of KV$-awareness. Specifically, when a load imbalance is detected, it completely ignores KV$-awareness, even though routing requests to instances with a higher KV$ hit ratio could still be beneficial if the amount of reduced computation is significant (as it helps reduce the load). With linear combination, this is possible as long as the weight assigned to the KV$ hit ratio is not too small. This is not possible in

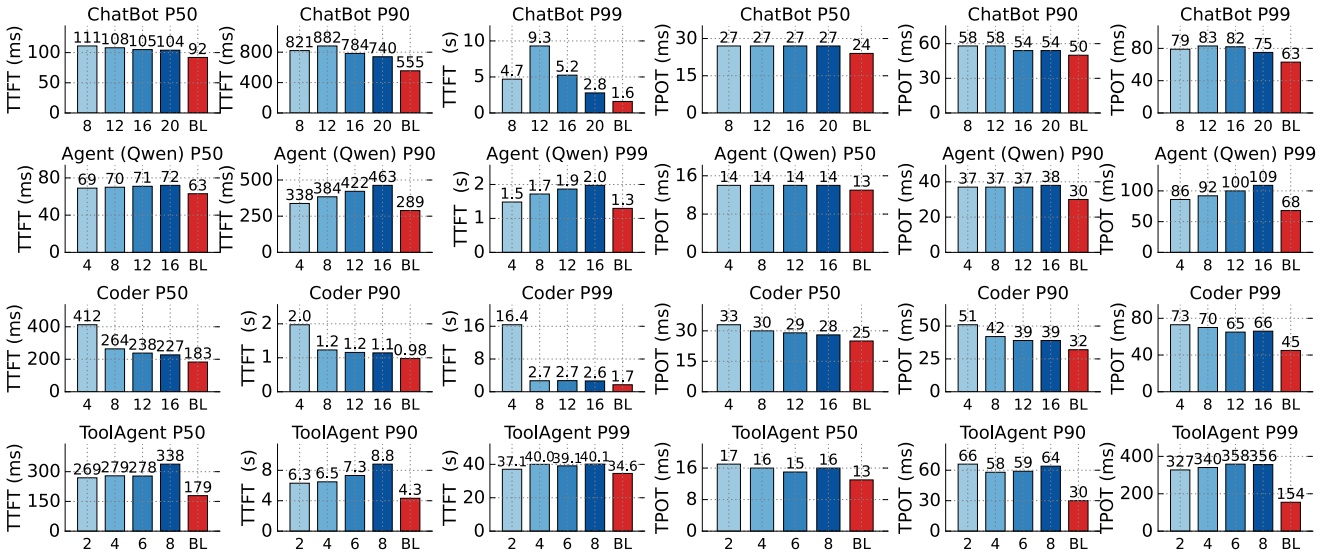


Figure 12: An analysis of how performance varies with different hyperparameters on four traces using a filter-based combination method. **BL** denotes the linear-combination-based method for comparison, tuned with the best hyperparameter. The numbers (2,4,6,8) are the range described in Figure 13. The model used is Qwen3-30B.

```
1  req = receive()                                  Filter-based
2  kv_hit = KV$.match(req)                            method
3  if BS.max()- BS.min() > Range:                (e.g., Aibrix-KV$)
       # Load balance
4      sched_to = instances
                  .select_min(BS)
5  else: # KV$-awareness
6      sched_to = instances
                  .select_max(kv_hit)
                  .select_min(BS)
7  req.forward(sched_to)
```

Figure 13: The pseudocode of filter-based combination of KV$-awareness and load balancing in LLM scheduling, simplified from `prefix-cache` policy of AIBrix [3].

```
1  req = receive()                                  Latency–Based
2  TTFT = Simulator_model.sim(req, BS, KV$, ...)
3  sched_to = instances.select_min(TTFT)
4  req.forward(sched_to)
```

Figure 14: The pseudocode of simulation-based method for combining KV$-awareness and load balancing in LLM scheduling.

filter-based methods.

4.6 The Case of Simulation-based Combination

Methodology. Finally, simulation-based methods [40, 24, 56, 13] use the simulated serving time of routing a request to a specific instance as the routing score. This is based on the observation that the execution of LLM requests is relatively deterministic—each step is a single forward pass—so it can be simulated accurately. Figure 14 shows the pseudocode of a typical simulation-based method [24]: after receiving a request, the router first estimates the TTFT of routing the request to each instance via a simulator (e.g., VIDUR [1]) (line 2), and then routes the request to the instance with the lowest estimated TTFT (lines 3–4). Note that simulation-based methods typically estimate the TTFT instead of the end-to-end latency because the end-to-end latency depends on the number of output tokens, which is unpredictable.

Simulation-based methods can be viewed as a higher-order combination of indicators that achieve KV$-awareness and load balancing. This is because the simulator must use the KV$ state and batch size of each instance as input features to accurately simulate the TTFT. Here, we chose a simulation-based implementation similar to llm-d [24] but with a retrofitted simulator from VIDUR [1]—the state-of-the-art LLM instance simulator. Our retrofitting consists of two parts: (1) we extend the simulator to consider KV$-aware execution by modeling the prefill phase with cache hits, and (2) we re-implement it in Rust and enable parallel simulation to scale the online simulation to multiple instances. Without KV$-awareness, the simulation-based method does not outperform its counterparts in our setting, similar to the observations we made in §4.2. Without Rust, the original Python-based implementation has considerable scheduling latency and would incur substantial online scheduling overhead. We also have several optimizations to scale the simulator to hundreds of instances. We will leave the details to another paper as it is not the focus of this work.

We studied a state-of-the-art method and found that it outperforms the linear-combination-based method in certain traces, as also shown in Figure 14. Despite the improved performance, simulation-based methods still have two issues due to their complexity:

Cons #1. Development complexity due to per-model implementation and per-hardware tuning. The performance of simulation-based methods depends on the accuracy of the simulator, which is non-trivial to achieve in practice because we need to consider both the model architecture and

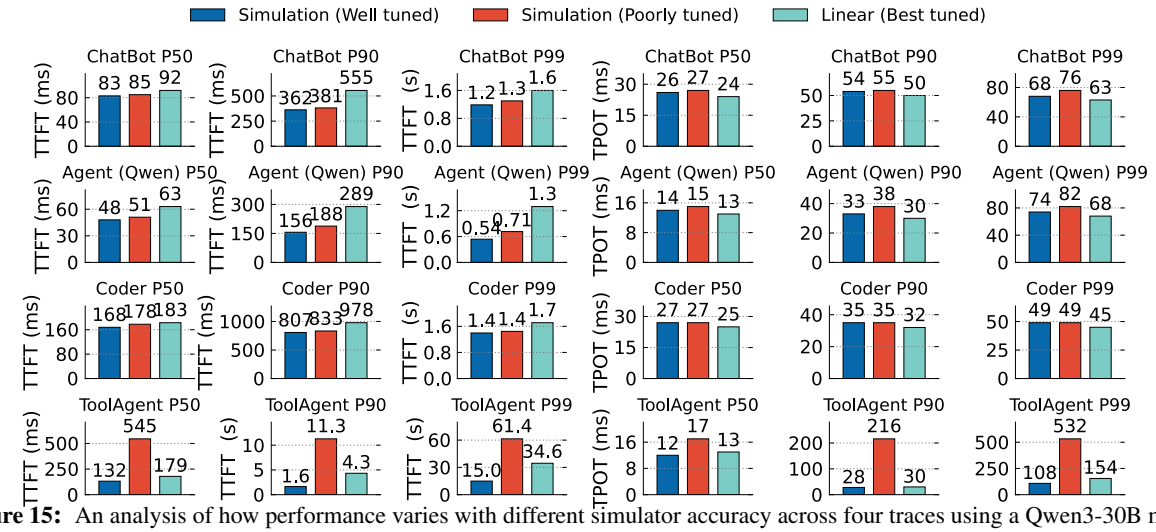


Figure 15: An analysis of how performance varies with different simulator accuracy across four traces using a Qwen3-30B model.

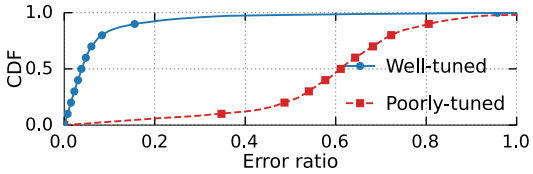


Figure 16: A comparison of a well-tuned simulator vs. a non-tuned one on ChatBot Trace with Qwen3-30B model.

the hardware characteristics. To quantify the impact of simulator accuracy on scheduling performance, Figure 15 presents the performance of using well-tuned vs. non-tuned simulators on four traces when serving a Qwen3-30B model. The poorly tuned simulator is one originally used for another model—Qwen2-7B, while a well-tuned simulator is the one implemented specifically for Qwen3-30B. Figure 16 measures the TTFT deviation when using two simulators to serve a Qwen3-30B model compared with using vLLM. We can see that a well-tuned simulator achieves much higher accuracy than an untuned one. With a more accurate simulator, the TTFT and TPOT tail latency improve by 75.6% and 79.7%, respectively.

While it is possible to develop per-model simulators for high scheduling performance, doing so incurs non-trivial development complexity because the simulations highly depend on the model architecture—which is evolving rapidly with new modules (e.g., linear attention [41] and Engram [14]). Meanwhile, for the same model, the hyperparameters of the simulator need to be tuned according to the hardware characteristics, leading to further engineering efforts.

Cons #2. Still sub-optimal performance. Simulation-based methods still suffer from sub-optimal performance, especially for TPOT. As shown in Figure 15, on the ToolAgent trace, the TPOT tail latency is 71.1% slower than that of the best linear-combination-based method. We hypothesize that this is because mispredictions of the simulator lead to load imbalance. For example, in Figure 16, we can see that even with

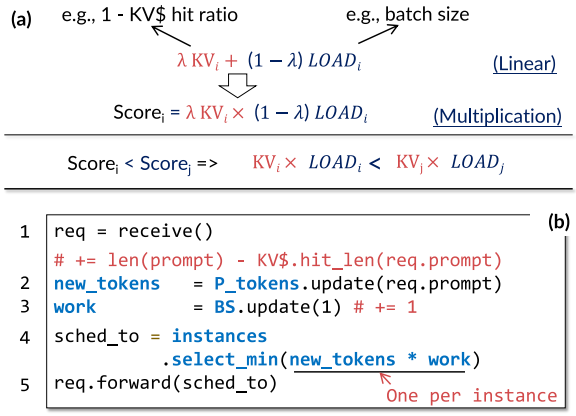


Figure 17: (a) An illustration of how multiplication combines two indicators without the hyperparameters needed in linear combination, and (b) the pseudocode of our scheduling method.

a well-tuned simulator, there are still about 10% of requests with more than 20% prediction error. Such errors mainly come from two sources: request reordering at the vLLM API server, and inaccuracies in latency prediction.

5 Simple Multiplication May Be All You Need

The methodology (Figure 17). Our method is simple: we only need to multiply two carefully chosen indicators to compute the scheduling score, one for KV$-awareness and the other for load balancing, and then route the request to the instance with the minimal score. The basic idea is based on the observation that, if a linear combination of two indicators works, then multiplication can also work, with the benefit of avoiding hyperparameter tuning, as shown in (a). The two indicators—**P-token** (the number of new prefill tokens if the request is routed to an instance, considering KV$ hits) and **BS** (the batch size of the instance)—are chosen based on our analysis in §5.1.

To see why routing to the instance with the minimal (P-

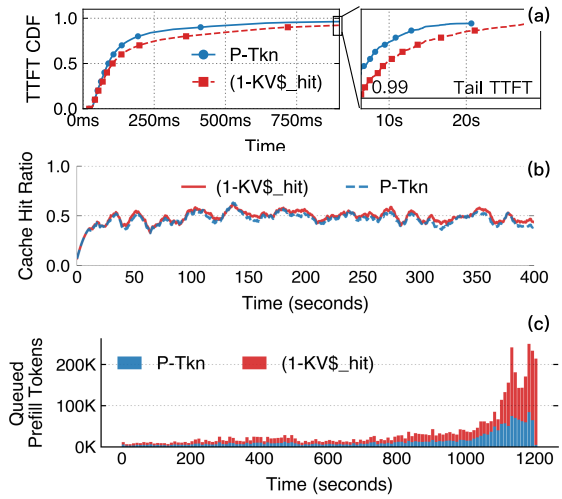


Figure 18: (a) A comparison of using new prefill tokens (**P-token**, shown as "P-Tkn" in the figure) vs. 1-KV$ hit ratio (1-KV$_{hit}$) as the KV$-awareness indicator ($A$) in $A \times BS$ scheduling, (b) the KV$ hit ratio analysis, and (c) the queued prefill tokens analysis. The analysis is done on the Qwen3-30B model with the ChatBot(Qwen) trace. Note that the bottom graph is not a stacked graph; rather, the bars overlap each other.

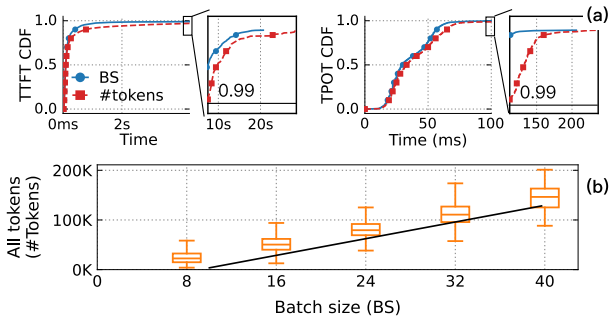


Figure 19: (a) A comparison of using batch size (BS) vs. total tokens (#Tokens) as the load-balancing indicator (B) in P-token $\times B$ scheduling. (b) The profiled relationship between batch size and total tokens. The analysis is done on the Qwen3-30B model with the ChatBot(Qwen) trace.

token $\times$ BS) score considers both objectives well, consider two instances i and j: if routing the request to instance i results in more KV$ hits than routing it to instance j, then instance i will have a lower **P-token** value unless there are many queued prefill requests in instance i (indicating work imbalance). Meanwhile, the **BS** captures the decode workload of each instance, so if instance i has a significantly larger batch size than instance j, the multiplication will likely favor instance j for load balancing.

5.1 The Choice of the Indicators

KV$-awareness indicator (P-token). Besides **P-token**, another natural choice for the KV$-awareness indicator is the **1-KV$ hit ratio**, i.e., the KV$ hit ratio if the request is routed to an instance. It is adopted by works such as Preble [44] and AIGW [6]. Note that we subtract from one because a higher KV$ hit ratio should yield a lower score to align with the new prefill tokens indicator. We do not consider TTFT because it requires a simulator, which is not always applicable.

Our empirical analysis shows that using **P-token** yields better performance than using **1-KV$ hit ratio**. Figure 18 (a) shows that using **P-token** results in a 14.4% lower P50 TTFT and a 42.8% lower P95 TTFT compared to using **1-KV$ hit ratio**. For a fair comparison, we fix the load-balancing indicator to **BS** in both cases. Due to space limitations, we only report the results for one workload here; however, the trend is consistent across all evaluated workloads.

To understand the cause of this difference, Figure 18 further breaks down the KV$ hit ratios of different methods in (b) and the load balancing status in (c). We can see that the two methods achieve similar KV$ hit ratios, so they are equally KV$-aware. The key difference is that using **P-token** achieves better load balancing, as it additionally considers the queued prefill tokens in each instance. As a result, when making scheduling decisions, the router bypasses instances with many queued prefill requests, even if they have a high KV$ hit ratio.

Load-balancing indicator (BS). Besides **BS**, another common choice is the total tokens (**#Tokens**) on each instance, as adopted by works such as ai-Dynamo [35] and AIGW [6]. The rationale is that the total computation cost of a request is proportional to the number of tokens in its context. However, we found that using **BS** yields better performance, as shown in Figure 19 (a), for two reasons. First, workloads can be categorized into prefill and decode loads, where the former is considered in our **P-token** indicator. Thus, we only need an indicator for the decode workload, which is exactly what **BS** captures (note that we have also tried using decode batch size, and the results are similar since the **BS** is dominated by the decode requests). Moreover, **BS** is a better indicator for the (decode) work assigned to an instance because the decode time is more stable across batch sizes. Specifically, a larger batch size leads to a longer decode time, but more decode tokens do not necessarily mean decode is slower when the batch is small, thanks to the KV$ [55]. This is illustrated in Figure 19 (b), where we profile the relationship between batch size and total tokens while serving a ChatBot(Qwen) workload with Qwen3-30B.

5.2 Benign and Failure Cases Analysis of the Multiplicative Scheduling Score

Overview. At a high level, as long as there is no load imbalance, i.e., one instance is overloaded while others are idle, routing requests to instances with high KV$ hit rates (i.e., low **P-token**) is always beneficial. Thus, the multiplication fails when a set of instances is about to be overloaded but their increase in **BS** cannot offset the decrease in **P-token** from high KV$ hit rates, so requests keep being routed there and cause the imbalance. This can happen under extreme KV$

skewness: when some prefixes are repeatedly accessed but only cached on a small set of instances—which we term KV$ *hotspots*. We found that such hotspots are rare in practice—at least not present in any of our evaluated traces. More importantly, as their patterns are clear, we can design a detector to catch them beforehand.

To analyze the failure cases of our multiplication method, we derive the condition under which KV$ hotspots occur and the increased batch size cannot offset the high KV$ hit rate given a workload pattern. For each workload, we first group requests by their KV$ prefixes and partition instances according to their KV$ ownership. Next, for each request class, we analyze the relationship between the workload pattern, the P-token indicator given the KV$ hit rate, and the batch size indicator.

Prelude: request grouping and instance partitioning. First, we partition all requests into a set of classes $\mathcal{C}$, where each $c \in \mathcal{C}$ corresponds to a group of requests that share the same KV$ prefix. In practice, a class roughly matches an application or a user: their requests share the same system prompt and often a similar conversation history [47]. For a fixed class c, we consider an accumulation time window $(t, t+window)$ and denote by x the fraction of all requests arriving at the cluster that belong to class c within this window; the remaining fraction is $\overline{x} = 1 - x$.

For each class c, we partition the cluster into two sets of instances: M and $\overline{M}$. M contains instances whose cache currently holds the prefix of class c, i.e., with KV$ hits, while $\overline{M}$ contains all other instances.

Impact of a request class on batch size. Let QPS be the total query rate of the cluster, and let t be the arrival time of the suspected class-c requests that may lead to imbalance. We denote the average batch size per instance in a balanced state prior to t as BS_0. We also denote the expected batch sizes of instances in M and $\overline{M}$ by BS_t and $\overline{BS}_t$ during the period $(t, t+window)$. We assume the extreme case in which all class-c requests are routed to M due to high KV$ hits, because routing any class-c request to an instance in $\overline{M}$ would cache the prefix there, expand M, and thereby dissipate the hotspot. Under this worst-case assumption, we establish the following approximate expression for the ratio between a potentially overloaded instance's batch size and a non-overloaded one's:

$$\frac{BS_t}{\overline{BS}_t} = \frac{BS_0 + (x \cdot \text{QPS})\,/|M| \cdot t}{BS_0 + (\overline{x} \cdot \text{QPS})\,/|\overline{M}| \cdot t}. \tag{1}$$

The main term $(x \cdot \text{QPS})\,/|M| \cdot t$ corresponds to the number of requests of class c routed to the instances that cache its prefix, while $(\overline{x} \cdot \text{QPS})\,/|\overline{M}| \cdot t$ is the number of all other requests routed to the remaining instances.

Analysis: when multiplication suffices, and is this common? As long as the batch size of the suspected hotspot instances is not larger than that of the others, it is beneficial to route requests to these instances, since doing so can exploit the KV$ hits without incurring load imbalance. This is precisely what our multiplicative method is designed for. The remaining question is whether such cases are common in practice. Fortunately, Equation 1 reveals that the batch-size skewness is governed by two ratios: the class popularity $x/\overline{x}$ and the cache coverage $|M|/|\overline{M}|$. We can therefore empirically analyze the prevalence of such cases by tracking these two ratios at runtime, sampling the request classes with the highest KV$ hits within each one-minute window.

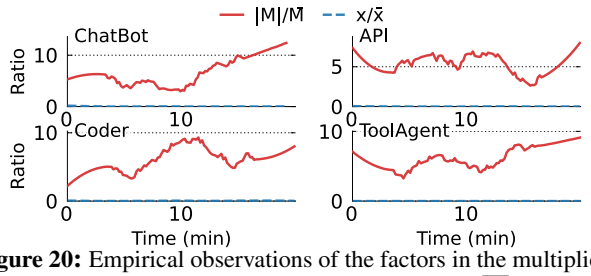


Figure 20: Empirical observations of the factors in the multiplicative score across different traces. If $x/\overline{x} \leq |M|/|\overline{M}|$, no KV$ hotspot can cause load imbalance, so our multiplicative method can effectively balance the load with KV$-awareness.

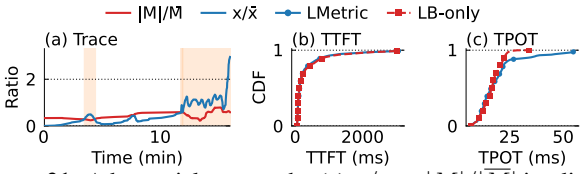


Figure 21: Adversarial case study: (a) $x/\overline{x} > |M|/|\overline{M}|$ implies a hotspot that could cause imbalance under LMETRIC, (b–c) performance comparison with a load-balance-only solution.

Figure 20 shows that for all traces (representative of LLM serving and studied in this work), our multiplication remains in its applicable regime, because the expected batch size of the suspected hotspot instances is not larger than that of the others. Concretely, every sampled class satisfies Equation 2:

$$\frac{x}{\overline{x}} \leq \frac{|M|}{|\overline{M}|} \tag{2}$$

which says the class's relative popularity $(x/\overline{x})$ never exceeds the relative cache coverage it enjoys $(|M|/|\overline{M}|)$; i.e., the prefix is cached on enough instances to absorb its share of the arrivals. This implies that even when every class-c request lands on M, no hit instance accumulates a larger batch size than a non-hit one, i.e., transforming and then substituting Equation 2 into Equation 1, we get $BS_t \leq \overline{BS}_t$.

Analysis: when multiplication fails. Based on the previous analysis, if Equation 2 does not hold, i.e., $x/\overline{x} > |M|/|\overline{M}|$, then a KV$ hotspot can cause load imbalance that breaks our multiplicative method. Though we have not observed such cases in any of our evaluated traces, we used the pattern of KV$ hotspots to examine a subset of the production workloads in BAILIAN and found one, as shown in the orange-shaded windows in Figure 21 (a). The cluster serves a thinking

workload with bursts of long requests sharing a common prefix, visible around minute 11 of the run. In this case, the batch indicator cannot outperform the KV$ indicator, and thus the multiplicative score continuously routes requests to a few instances with high KV$ hits, leading to load imbalance and performance degradation. Figure 21 (b–c) shows that LMETRIC cannot outperform a load-balancing-only solution (i.e., vLLM) during this period.

Retrofit: a two-phase detector for KV$ hotspots. The boundary condition in Equation 2 yields a clear failure-case detector based purely on the workload pattern and KV$ states, which we run alongside each scheduling decision to catch potential hotspots. For each request class, we monitor the two ratios $x/\overline{x}$ and $|M|/|\overline{M}|$ in real time; when Equation 2 is violated, we raise an alarm and intervene by filtering out the suspected instances (M) from the routing targets (e.g., the orange-shaded windows in Figure 21 (a)). To bound the monitoring overhead, we only track requests with the highest KV$ hit rates.

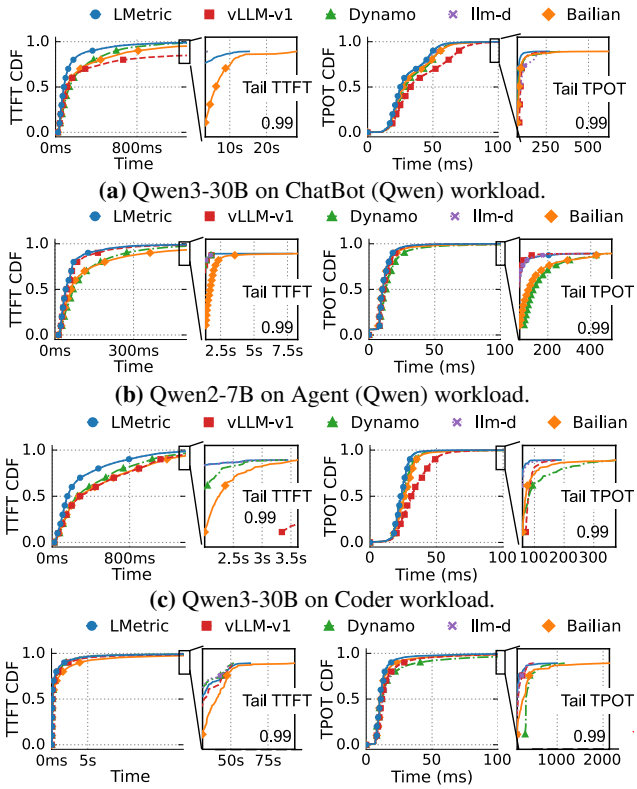
However, Equation 2 is only a necessary (not sufficient) condition for a hotspot, because it was derived under the worst-case assumption that all class-c requests are routed to M. In reality, even when Equation 2 is violated, it may still be beneficial to route requests to the suspected instances with high KV$ hits, as long as we do not route too many. We therefore augment the detector with a second phase: after the first phase raises an alarm, we track each subsequent class request and filter out M from scheduling only when $2|M|$ consecutive requests would receive a smaller multiplicative score on a hotspot instance $m \in M$ than on a non-hotspot one $m' \in \overline{M}$: i.e., P-token$_m \times BS_m \leq$ P-token$_{m'} \times BS_{m'}$.

6 End-to-end Evaluation

6.1 Comparison with Production Schedulers

Baselines. We compare LMETRIC against popular production schedulers and the current scheduler used in BAILIAN. As certain baselines show low router throughput in their default implementations, e.g., vLLM exhibits low processing throughput with its Python-based router [46], we re-implement their policies within our highly optimized Rust router framework described in §3. For an apples-to-apples comparison, we evaluate all policies under our framework, and we have carefully verified that our re-implementations are no slower than their original implementations. The detailed descriptions of the baselines are as follows:

- **BAILIAN** is the production scheduler used in BAILIAN's LLM serving system. It adopts a linear-combination-based approach similar to the one shown in Figure 6 (b). We have carefully tuned its hyperparameters for each workload to achieve the best performance.
- **vLLM [45]** is a widely used LLM serving system that adopts a load-balancing-only design (see Figure 6 (a)).



(a) Qwen3-30B on ChatBot (Qwen) workload.

(b) Qwen2-7B on Agent (Qwen) workload.

(c) Qwen3-30B on Coder workload.

(d) Qwen3-30B on Agent (Kimi) workload.

Figure 22: End-to-end TTFT and TPOT CDFs of LMETRIC and baselines on four workloads.

- **Dynamo [35]** is a popular serving framework released by NVIDIA. It also adopts a linear-combination-based approach but with a different choice of indicators than BAILIAN's. The two indicators chosen are the number of prefill tokens (the same as our P-token) for KV$-awareness and the total tokens in the instance for load balancing. The router routes requests to the instance with the minimal regulated and weighted sum of the two indicators. Similar to BAILIAN, we also tune its hyperparameters for each workload to achieve the best performance.
- **llm-d [24]** adopts a latency-based scheduling policy: it estimates the TTFT and routes requests to the instance with the lowest TTFT using the simulation-based approach described in §4.6.

Overall performance. Figure 22 shows the end-to-end TTFT and TPOT CDFs of LMETRIC and the baselines on four traces. All experiments are conducted on a 16-GPU testbed with 16 instances, and each trace is scaled to half of the maximum load that the testbed can handle. Due to space constraints, we report results for a representative subset: the Qwen3 MoE model on ChatBot, Coder, and Agent workloads, and the Qwen2 model on the Agent (Qwen) workload. We observe consistent performance trends across all model–trace combinations.

LMETRIC outperforms all baselines across all traces. On

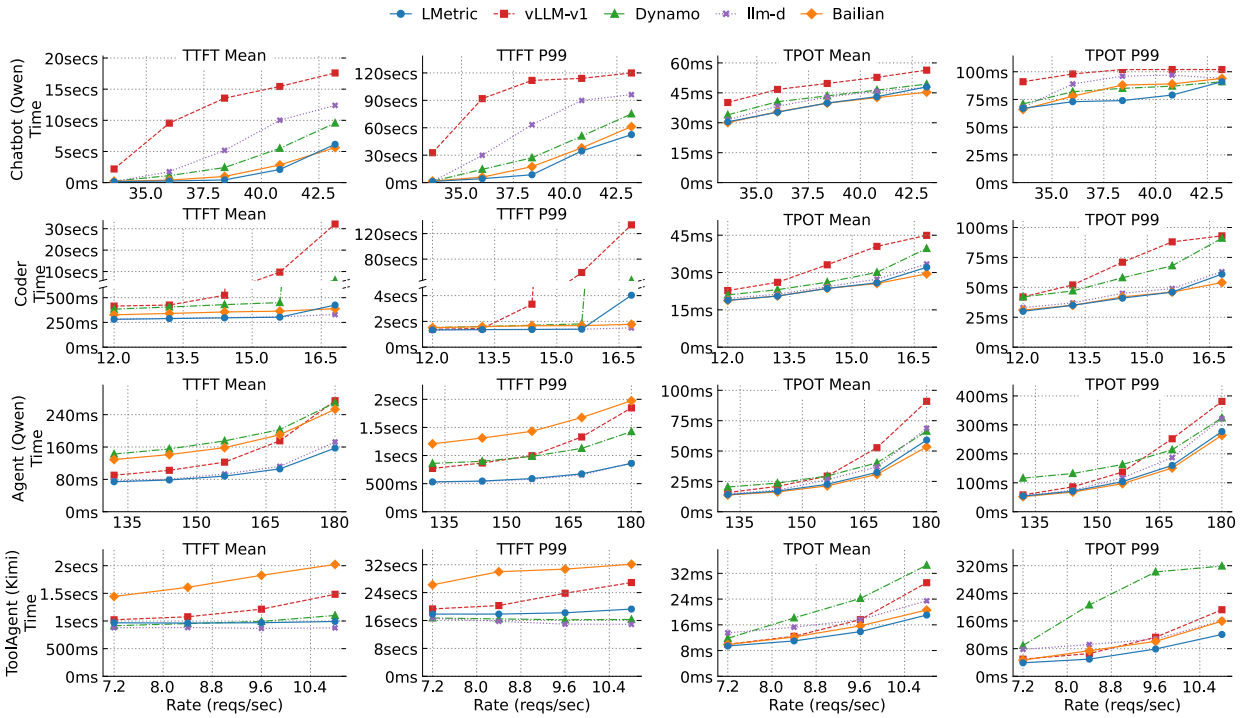


Figure 23: End-to-end performance under different request rates. Except for the second row, which uses a Qwen2-7B model, all other rows use a Qwen3-30B model.

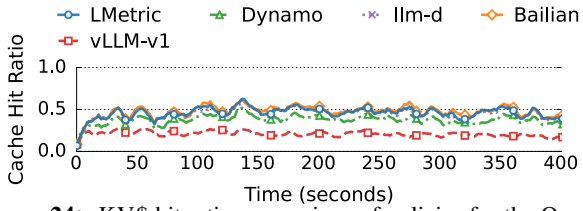


Figure 24: KV$ hit ratio comparison of policies for the Qwen3-30B model on the ChatBot (Qwen) workload.

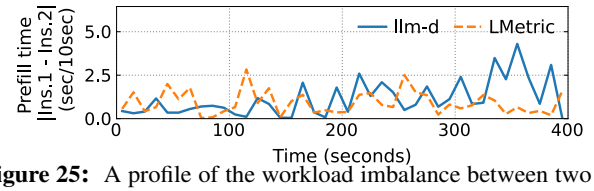


Figure 25: A profile of the workload imbalance between two instances under LMETRIC and llm-d while serving a ChatBot (Qwen) workload with the Qwen3-30B model. The reported metric is the absolute served prefill time in each 10-second window between the two instances (Inst.).

the ChatBot workload, it reduces the mean TTFT by 92% and the mean TPOT by 24% compared to vLLM, and it reduces the P99 TPOT by 13% compared to llm-d—the second-best policy—with a similar TTFT. The improved performance mainly comes from being KV$-aware without sacrificing load balancing. To examine LMETRIC's behavior, Figure 24 plots the KV$ hit ratio for different systems on the ChatBot workload. We can see that LMETRIC consistently achieves a KV$ hit ratio comparable to that of other KV$-aware policies, and its ratio is significantly higher than that of the KV$-unaware policy (vLLM). Meanwhile, Figure 25 further analyzes the imbalance following the approach used in §4.3. For brevity, we compare LMETRIC only with llm-d—the second-best policy on the ChatBot trace. We can see that LMETRIC achieves a better-balanced load than llm-d.

Performance under different request rates. Figure 23 further shows how LMETRIC performs under different request arrival rates. The results are largely consistent with those under a fixed request rate setting: LMETRIC outperforms all baselines across different traces and request rates, except for the ToolAgent trace, where LMETRIC exhibits a slightly higher (10%) mean TTFT than llm-d but achieves a 30% lower TPOT. This may be because a simulation-based approach can better estimate the prefill workload than our simple P-token indicator. Nevertheless, LMETRIC still achieves the lowest TPOT without relying on a complex simulator, as it considers both KV$ management and (decode) load balancing. The performance gap between baselines widens as the request rate increases, because under high load, a more balanced and KV$-aware scheduling strategy can yield higher overall system throughput, leading to faster consumption of queued requests.

6.2 Comparison with Recent Research Schedulers

Baselines. We further compare LMETRIC with two state-of-the-art research schedulers: Preble [44] and PolyServe [56]. As in the previous section, both baselines run in our router framework (§3): (1) Preble's open-source release runs on a different router stack from ours, (2) PolyServe is not

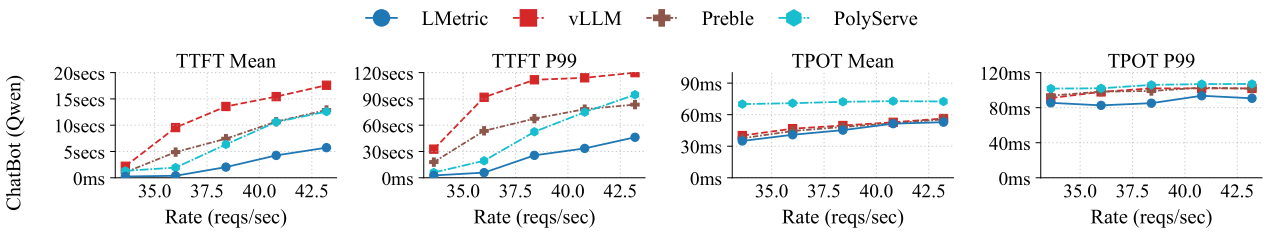


Figure 26: End-to-end performance of LMETRIC alongside two recent research baselines (Preble and PolyServe) under the same setting as Figure 23, with vLLM included as a reference. Model: Qwen3-30B on the ChatBot (Qwen) workload.

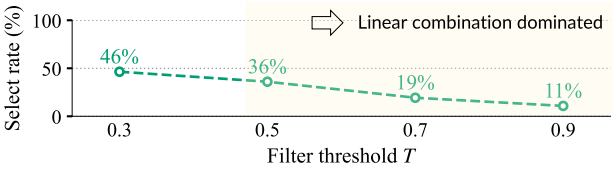


Figure 27: Preble's KV$-aware branch selection rate under different filter thresholds T.

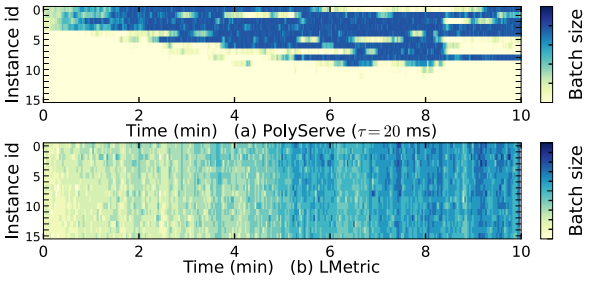


Figure 28: Running batch size across 16 instances over a 10-minute window, under PolyServe and LMETRIC. Workload: ChatBot (Qwen) trace at 18.75 reqs/sec on Qwen3-30B instances.

open-sourced, and re-implementing both in our framework enables an apples-to-apples comparison. We have carefully tuned different implementations and their hyperparameters to achieve the best performance, as detailed in §A.

- **Preble [44]** adopts a hybrid filter-based (§4.5) and linear-combination (§4.4) scheme. It first filters instances with high KV$ hit ratios based on a threshold ($T = 0.5$ tuned in §A.1). If the filter returns any instance, it routes to the one with the highest hit ratio; otherwise, it routes the request to the instance with the highest linear-combined score over KV$ and load indicators.
- **PolyServe [56]** is a simulation-based scheduler (§4.6) that optimizes a different objective from the other baselines and LMETRIC: meet the SLO while creating a load gradient across instances that facilitates auto-scaling. Using the simulator's predicted TTFT and TPOT, PolyServe routes each request to the most loaded instance whose predicted latency still meets the SLO bounds SLO_{TTFT} and SLO_{TPOT}; when no instance is feasible, it falls back to the instance with the lowest predicted TPOT.

Overall performance under different request rates. Figure 26 reports mean and P99 TTFT/TPOT for LMETRIC and the baselines on ChatBot (Qwen) with Qwen3-30B. Other workloads share similar results.

Compared with Preble, LMETRIC reduces mean TTFT by 56% and mean TPOT by 8%, and it reduces P99 TTFT by 45% and P99 TPOT by 16%. Although Preble is faster than vLLM thanks to KV$-awareness, it performs similarly to the linear-combination baselines that LMETRIC outperforms, because it falls back to the linear-combination branch most of the time (see Figure 27). Note that lowering the threshold to reduce the linear-combination branch selection rate does not necessarily improve Preble's performance, as it biases the scheduling towards KV$ and thus sacrifices load balancing.

Compared with PolyServe, LMETRIC achieves lower mean and P99 TTFT and TPOT across all request rates in Figure 26. The gap reflects PolyServe's design objective: instead of balancing load, PolyServe creates a load gradient across instances so that idle ones can be released by auto-scaling. Concentrating load this way raises instance utilization but degrades per-request latency. Figure 28 illustrates this trade-off: PolyServe loads instances 0–8 and leaves 9–15 idle, while LMETRIC spreads the same aggregate workload across all 16 instances.

6.3 In-production Evaluation

Deployment overview and methodology to validate performance improvements. LMETRIC now serves production traffic at BAILIAN as the scheduler for a Qwen3.5-27B cluster, our first full-scale deployment. To measure the impact on service quality, we report the observed performance from a canary release. On one day in May 2026, we split the traffic that previously fed one production cluster across two clusters: 1/3 of the traffic went to a cluster running LMETRIC, and the remaining 2/3 to a cluster running BAILIAN's prior scheduling method. To ensure a fair comparison, we sized the clusters so that each received the same reqs/GPU. The LMETRIC cluster contained several hundred GPUs; the detailed cluster setup is confidential and thus omitted.

Performance. LMETRIC reduces both TTFT and TPOT in production: Figure 29 compares the two clusters using screenshots of BAILIAN's internal performance dashboard. The snapshots show that, compared with BAILIAN's prior scheduler, mean TTFT and mean TPOT drop by 39% and 51% respectively. The results imply that LMETRIC can serve

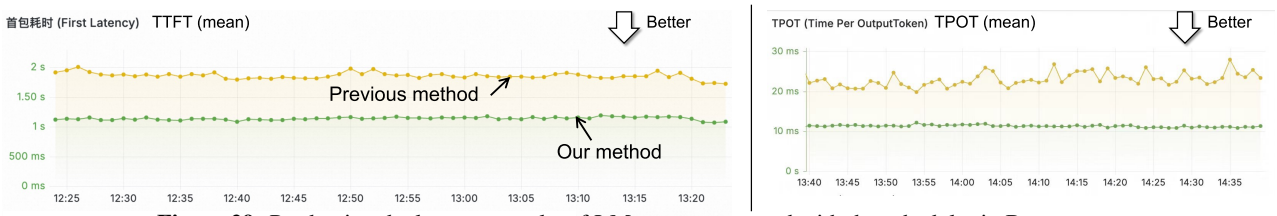


Figure 29: Production deployment results of LMETRIC compared with the scheduler in BAILIAN.

the same workload with fewer GPUs under the same SLO.

7 Discussion

Scheduling under PD-disaggregation. Scheduling under PD-disaggregation is different (and we argue simpler) than our targeted PD-colocation, for two reasons. First, routing decode requests primarily needs to consider load balancing. Existing systems [35, 3, 24, 40] already handle this effectively with simple indicators like batch size [3] or total tokens [35]. Second, routing prefill requests can address KV$-awareness and prefill load balancing with a unified indicator—the number of new prefill tokens after cache hits. Using this indicator as a scheduling score naturally combines both objectives without explicit hyperparameter tuning [3, 35], and we find it effective in our analysis. On the other hand, PD-disaggregation does introduce new scheduling challenges, such as managing prefill and decode clusters with different capacities, which we leave to future work.

Scheduling under heterogeneous deployments. In real-world production, model services are heterogeneous in both models and GPU types, while LMETRIC targets scheduling for a single model under homogeneous GPUs. Nevertheless, LMETRIC still works in this setting because providers, to cope with this complexity, logically partition their deployment into multiple clusters, each serving a single model on a single GPU type. The global scheduler inside each cluster therefore sees a homogeneous pool of instances, where LMETRIC applies.

KVCache sharing. Some serving clusters support KV$ sharing: an instance can fetch the requested KV$ from another instance through RDMA when it does not hold the cache locally [40]. This mechanism reduces the cost of scheduling a request to an instance without KV$, but it does not remove the benefits of KV$-aware scheduling provided by LMETRIC because local KV$ hits are still cheaper than remote KV$ fetches in two ways. First, a local hit avoids a remote transfer on the TTFT path. Second, it avoids keeping the same KV$ on two instances, which saves memory and reduces cache pressure.

8 Related Work

LLM requests global scheduling. LMETRIC continues the line of research on scheduling LLM requests in a cluster [29, 27, 44, 40, 56, 24, 52]. To the best of our knowledge, all these methods target both KV$-awareness and load balancing by combining indicators for each objective through the three approaches discussed in §4. LMETRIC reuses these indicators but combines them with a simple yet efficient multiplication combinator.

LLM requests scheduling within an instance. These works are orthogonal to LMETRIC's global scheduling: they optimize request execution within an instance. Sarathi-Serve [2] introduces chunked prefill, which splits long prefill requests into smaller chunks to reduce stalls for co-located decode requests. VTC [43] adopts token-based admission control to achieve fairness. FairBatching [32] uses a linear-time analytical model to prioritize prefill versus decode tokens and dynamically form batches. A good global scheduler can further improve these local schedulers, e.g., by reducing overloads that are difficult to handle locally.

Optimizing LLM serving. Beyond scheduling, LLM serving systems improve performance by increasing KV$ hit rates [40, 47], adding model-serving elasticity [53, 50], and improving GPU execution efficiency [29, 28, 18, 16], just to name a few. These techniques coexist with scheduling optimizations.

9 Conclusion

We contribute the first multiplicative combinator for high-quality LLM request scheduling, achieving both KV$-awareness and load balancing in a hyperparameter-free manner. Evaluations on real-world workloads covering chatbots and agents confirm the benefits of our approach, and our method has been deployed in production with confirmed effectiveness.

10 Acknowledgment

We sincerely thank the OSDI'26 reviewers for their insightful comments and Neeraja J. Yadwadkar for shepherding our paper. We also thank Jinyu Gu for valuable feedback and for suggesting the classic system term "simple is better" for our paper. This work was supported in part by the National Natural Science Foundation of China (No. 62572302), the Fundamental and Interdisciplinary Disciplines Breakthrough Plan of the Ministry of Education of China (No. JYB2025XDXM122), and the Alibaba Innovative Research Program.

References

[1] AGRAWAL, A., KEDIA, N., MOHAN, J., PANWAR, A., KWATRA, N., GULAVANI, B. S., RAMJEE, R., AND TUMANOV, A. VIDUR: A large-scale simulation framework for LLM inference. In *Proceedings of the Seventh Annual Conference on Machine Learning and Systems, MLSys 2024, Santa Clara, CA, USA, May 13-16, 2024* (2024), P. B. Gibbons, G. Pekhimenko, and C. D. Sa, Eds., mlsys.org.

[2] AGRAWAL, A., KEDIA, N., PANWAR, A., MOHAN, J., KWATRA, N., GULAVANI, B., TUMANOV, A., AND RAMJEE, R. Taming Throughput-Latency tradeoff in LLM inference with Sarathi-Serve. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)* (Santa Clara, CA, July 2024), USENIX Association, pp. 117–134.

[3] Aibrix. https://github.com/vllm-project/aibrix, 2025.

[4] v0.4.1: fatal error: concurrent map writes in prefix cache eviction. https://github.com/vllm-project/aibrix/issues/1580, 2025.

[5] AIBrix `prefix_cache_preble.go`. https://github.com/vllm-project/aibrix/blob/main/pkg/plugins/gateway/algorithms/prefix_cache_preble.go, 2026.

[6] Aigw. https://github.com/aigw-project/aigw, 2025.

[7] ALI, A., PINCIROLI, R., YAN, F., AND SMIRNI, E. Optimizing inference serving on serverless platforms. *Proc. VLDB Endow. 15*, 10 (2022), 2071–2084.

[8] ANTHROPIC. Claude api. https://www.anthropic.com/api, 2025.

[9] ARAPAKIS, I., BAI, X., AND CAMBAZOGLU, B. B. Impact of response latency on user behavior in web search. In *The 37th International ACM SIGIR Conference on Research and Development in Information Retrieval, SIGIR '14, Gold Coast , QLD, Australia - July 06 - 11, 2014* (2014), S. Geva, A. Trotman, P. Bruza, C. L. A. Clarke, and K. Järvelin, Eds., ACM, pp. 103–112.

[10] AZURE. Azure llm inference traces. https://github.com/Azure/AzurePublicDataset/blob/master/AzureLLMInferenceDataset2024.md, 2024.

[11] Qwen-bailian anonymous dataset. https://github.com/alibaba-edu/qwen-bailian-usagetraces-anon, 2025.

[12] BLITZ SERVING. Trace replayer. https://github.com/blitz-serving/trace-replayer.

[13] CHEN, S., JIA, Z., KHAN, S., KRISHNAMURTHY, A., AND GIBBONS, P. B. Slos-serve: Optimized serving of multi-slo llms. *CoRR abs/2504.08784* (2025).

[14] CHENG, X., ZENG, W., DAI, D., CHEN, Q., WANG, B., XIE, Z., HUANG, K., YU, X., HAO, Z., LI, Y., ZHANG, H., ZHANG, H., ZHAO, D., AND LIANG, W. Conditional memory via scalable lookup: A new axis of sparsity for large language models, 2026.

[15] CLOUD, A. Alibaba cloud bailian. https://www.aliyun.com/product/bailian, 2026.

[16] DAO, T. Flashattention-2: Faster attention with better parallelism and work partitioning. In *The Twelfth International Conference on Learning Representations, ICLR 2024, Vienna, Austria, May 7-11, 2024* (2024), OpenReview.net.

[17] DELIMITROU, C., AND KOZYRAKIS, C. Quasar: resource-efficient and qos-aware cluster management. In *Architectural Support for Programming Languages and Operating Systems, ASPLOS 2014, Salt Lake City, UT, USA, March 1-5, 2014* (2014), R. Balasubramonian, A. Davis, and S. V. Adve, Eds., ACM, pp. 127–144.

[18] FlashInfer: Kernel Library for LLM Serving. https://github.com/flashinfer-ai/flashinfer, 2025.

[19] GAO, B., HE, Z., SHARMA, P., KANG, Q., JEVDJIC, D., DENG, J., YANG, X., YU, Z., AND ZUO, P. Cost-Efficient large language model serving for multi-turn conversations with CachedAttention. In *2024 USENIX Annual Technical Conference (USENIX ATC 24)* (Santa Clara, CA, July 2024), USENIX Association, pp. 111–126.

[20] GIGASPACES. Amazon found every 100ms of latency cost them 1% in sales. https://www.gigaspaces.com/blog/amazon-found-every-100ms-of-latency-cost-them-1-in-sales, 2024.

[21] GITHUB. Accelerate your development speed with copilot. https://copilot.github.com, 2024.

[22] GOG, I., SCHWARZKOPF, M., GLEAVE, A., WATSON, R. N. M., AND HAND, S. Firmament: Fast, centralized cluster scheduling at scale. In *12th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2016, Savannah, GA, USA, November 2-4, 2016* (2016), K. Keeton and T. Roscoe, Eds., USENIX Association, pp. 99–115.

[23] GOOGLE. Gemini api. https://ai.google.dev/api, 2025.

[24] GOOGLE. llm-d. https://github.com/llm-d/llm-d, 2025.

[25] GUJARATI, A., KARIMI, R., ALZAYAT, S., HAO, W., KAUFMANN, A., VIGFUSSON, Y., AND MACE, J. Serving dnns like clockwork: Performance predictability from the bottom up. In *14th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2020, Virtual Event, November 4-6, 2020* (2020), USENIX Association, pp. 443–462.

[26] HU, C., HUANG, H., XU, L., CHEN, X., XU, J., CHEN, S., FENG, H., WANG, C., WANG, S., BAO, Y., SUN, N., AND SHAN, Y. Inference without interference: Disaggregate LLM inference for mixed downstream workloads. *CoRR abs/2401.11181* (2024).

[27] HU, X., ZENG, T., YUAN, X., SONG, L., ZHANG, G., AND HE, B. Bestserve: Serving strategies with optimal goodput in collocation and disaggregation architectures. *CoRR abs/2506.05871* (2025).

[28] KAMATH, A. K., PRABHU, R., MOHAN, J., PETER, S., RAMJEE, R., AND PANWAR, A. Pod-attention: Unlocking full prefill-decode overlap for faster LLM inference. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating*

Systems, Volume 2, ASPLOS 2025, Rotterdam, Netherlands, 30 March 2025 - 3 April 2025 (2025), L. Eeckhout, G. Smaragdakis, K. Liang, A. Sampson, M. A. Kim, and C. J. Rossbach, Eds., ACM, pp. 897–912.

[29] KWON, W., LI, Z., ZHUANG, S., SHENG, Y., ZHENG, L., YU, C. H., GONZALEZ, J., ZHANG, H., AND STOICA, I. Efficient memory management for large language model serving with pagedattention. In *Proceedings of the 29th Symposium on Operating Systems Principles, SOSP 2023, Koblenz, Germany, October 23-26, 2023* (2023), J. Flinn, M. I. Seltzer, P. Druschel, A. Kaufmann, and J. Mace, Eds., ACM, pp. 611–626.

[30] LAI, R., LIU, H., LU, C., LIU, Z., CAO, S., SHAO, S., ZHANG, Y., MAI, L., AND USTIUGOV, D. Tokenscale: Timely and accurate autoscaling for disaggregated LLM serving with token velocity. *CoRR abs/2512.03416* (2025).

[31] LI, J., ZHU, Y., LEE, E. K., AND NAHRSTEDT, K. Revisiting disaggregated large language model serving for performance and energy implications. *CoRR abs/2601.08833* (2026).

[32] LYU, H., LIU, B., WU, M., AND CHEN, H. Fairbatching: Fairness-aware batch formation for LLM inference. *CoRR abs/2510.14392* (2025).

[33] MIAO, X., SHI, C., DUAN, J., XI, X., LIN, D., CUI, B., AND JIA, Z. Spotserve: Serving generative large language models on preemptible instances. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2, ASPLOS 2024, La Jolla, CA, USA, 27 April 2024- 1 May 2024* (2024), R. Gupta, N. B. Abu-Ghazaleh, M. Musuvathi, and D. Tsafrir, Eds., ACM, pp. 1112–1127.

[34] Mooncake trace. https://github.com/kvcache-ai/Mooncake/blob/main/FAST25-release/traces/toolagent_trace.jsonl, 2025.

[35] NVIDIA. ai-dynamo. https://github.com/ai-dynamo/dynamo, 2025.

[36] OPENAI. Openai developer platform. https://platform.openai.com/docs/overview.

[37] OPENAI. Chatgpt. https://chatgpt.com, 2025.

[38] PATEL, P., CHOUKSE, E., ZHANG, C., SHAH, A., GOIRI, Í., MALEKI, S., AND BIANCHINI, R. Splitwise: Efficient generative LLM inference using phase splitting. In *51st ACM/IEEE Annual International Symposium on Computer Architecture, ISCA 2024, Buenos Aires, Argentina, June 29 - July 3, 2024* (2024), IEEE, pp. 118–132.

[39] Pollaczek–khinchine formula. https://en.wikipedia.org/wiki/PollaczekâĂŞKhinchine_formula#cite_note-2, 2025.

[40] QIN, R., LI, Z., HE, W., CUI, J., REN, F., ZHANG, M., WU, Y., ZHENG, W., AND XU, X. Mooncake: Trading more storage for less computation — a KVCache-centric architecture for serving LLM chatbot. In *23rd USENIX Conference on File and Storage Technologies (FAST 25)* (Santa Clara, CA, Feb. 2025), USENIX Association, pp. 155–170.

[41] Qwen3-next. https://qwen.ai/blog?id=4074cca80393150c248e508aa62983f9cb7d27cd&from=research.latest-advancements-list, 2026.

[42] SAJAL, S. M., ZHU, T., URGAONKAR, B., AND SEN, S. Traceupscaler: Upscaling traces to evaluate systems at high load. In *Proceedings of the Nineteenth European Conference on Computer Systems, EuroSys 2024, Athens, Greece, April 22-25, 2024* (2024), ACM, pp. 942–961.

[43] SHENG, Y., CAO, S., LI, D., ZHU, B., LI, Z., ZHUO, D., GONZALEZ, J. E., AND STOICA, I. Fairness in serving large language models. In *18th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2024, Santa Clara, CA, USA, July 10-12, 2024* (2024), A. Gavrilovska and D. B. Terry, Eds., USENIX Association, pp. 965–988.

[44] SRIVATSA, V., HE, Z., ABHYANKAR, R., LI, D., AND ZHANG, Y. Preble: Efficient distributed prompt scheduling for LLM serving. In *The Thirteenth International Conference on Learning Representations, ICLR 2025, Singapore, April 24-28, 2025* (2025), OpenReview.net.

[45] vllm v0.12.0 release. https://github.com/vllm-project/vllm/releases/tag/v0.12.0, 2025.

[46] [bugfix]: Avoid unnecessary coordination for non-moe data parallel. https://github.com/vllm-project/vllm/issues/24461, 2026.

[47] WANG, J., HAN, J., WEI, X., SHEN, S., ZHANG, D., FANG, C., CHEN, R., YU, W., AND CHEN, H. Kvcache cache in the wild: characterizing and optimizing kvcache cache at a large cloud provider. In *Proceedings of the 2025 USENIX Conference on Usenix Annual Technical Conference* (USA, 2025), USENIX ATC '25, USENIX Association.

[48] WANG, J., HAN, J., WEI, X., SHEN, S., ZHANG, D., FANG, C., CHEN, R., YU, W., AND CHEN, H. Kvcache cache in the wild: Characterizing and optimizing kvcache cache at a large cloud provider. In *2025 USENIX Annual Technical Conference (USENIX ATC 25)* (July 2025), USENIX Association.

[49] WANG, Y., CHEN, Y., LI, Z., KANG, X., FANG, Y., ZHOU, Y., ZHENG, Y., TANG, Z., HE, X., GUO, R., ET AL. Burstgpt: A real-world workload dataset to optimize llm serving systems. In *Proceedings of the 31st ACM SIGKDD Conference on Knowledge Discovery and Data Mining V. 2* (2025), pp. 5831–5841.

[50] XIANG, Y., LI, X., QIAN, K., YANG, Y., ZHU, D., YU, W., ZHAI, E., LIU, X., JIN, X., AND ZHOU, J. Aegaeon: Effective GPU pooling for concurrent LLM serving on the market. In *Proceedings of the ACM SIGOPS 31st Symposium on Operating Systems Principles, SOSP 2025, Lotte Hotel World, Seoul, Republic of Korea, October 13-16, 2025* (2025), Y. Won, Y. Kwon, D. Yuan, and R. Isaacs, Eds., ACM, pp. 1030–1045.

[51] XIANG, Y., LI, X., QIAN, K., YU, W., ZHAI, E., AND JIN, X. Servegen: Workload characterization and generation of large language model serving in production. *CoRR abs/2505.09999* (2025).

[52] YUAN, Y., ZUO, P., WANG, B., CHEN, Z., TAN, Z., AND YU, Z. Dualmap: Enabling both cache affinity and load balancing for distributed LLM serving. *CoRR abs/2602.06502* (2026).

[53] ZHANG, D., WANG, H., LIU, Y., WEI, X., SHAN, Y., CHEN, R., AND CHEN, H. Blitzscale: Fast and live large model autoscaling with O(1) host caching. In *19th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2025, Boston, MA, USA, July 7-9, 2025* (2025), L. Zhou and Y. Zhou, Eds., USENIX Association, pp. 275–293.

[54] ZHANG, X., TUNE, E., HAGMANN, R., JNAGAL, R., GOKHALE, V., AND WILKES, J. Cpi^2: CPU performance isolation for shared compute clusters. In *Eighth Eurosys Conference 2013, EuroSys '13, Prague, Czech Republic, April 14-17, 2013* (2013), Z. Hanzálek, H. Härtig, M. Castro, and M. F. Kaashoek, Eds., ACM, pp. 379–391.

[55] ZHONG, Y., LIU, S., CHEN, J., HU, J., ZHU, Y., LIU, X., JIN, X., AND ZHANG, H. Distserve: Disaggregating prefill and decoding for goodput-optimized large language model serving. In *18th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2024, Santa Clara, CA, USA, July 10-12, 2024* (2024), A. Gavrilovska and D. B. Terry, Eds., USENIX Association, pp. 193–210.

[56] ZHU, K., SHI, H., XU, L., SHAN, J., KRISHNAMURTHY, A., KASIKCI, B., AND XIE, L. Polyserve: Efficient multi-slo serving at scale. *CoRR abs/2507.17769* (2025).

A Appendix

A.1 More on Preble

```
1  req = receive()
2  kv_hit = KV$.match(req)
3  if kv_hit > T:
       # KV$-awareness
4      sched_to = instances
                  .select_max(kv_hit)
                  .select_min(P_tokens)
5  else: # Linear fallback
6      sched_to = instances
                  .select_min(∑3min α * P_tokens
                                  + β * BS)
7  req.forward(sched_to)
```

Figure 30: The pseudocode of Preble's method.

Preble method in detail. Preble [44] adopts two of the combination methods we identify in a hybrid form: a KV$-aware filter (§4.5) on top of a linear-combination fallback (§4.4). Figure 30 shows the pseudocode using our scheduling language. After receiving a request, Preble first filters the instances with high KV$ hits—i.e., instances whose cached prefix covers more than a threshold T of the prompt (line 3). Among the instances tied for the highest KV$ hit ratio, it routes the request to the one with the least prefill load (line 4). Otherwise, it uses a linear-combination score over all instances and selects the one with the smallest score (line 6).

More specifically, Preble adopts the following scoring function for the fallback path:

$$\arg\min_i \sum_{r \in W_i} \left(\mathrm{PT}_r + \mathrm{DT}_r\right),$$

a sum of per-request prefill time PT_r and decode time DT_r over recent requests W_i routed to instance i. Preble realizes PT_r by assigning a pre-determined per-token prefill cost to each newly prefilled token of r, and DT_r by assigning a pre-determined per-request decode cost to r itself, both aggregated per instance over a 3-minute sliding window. In its implementation [5], Preble derives these two costs from exact indicators exported by the engine, see below.

$$\arg\min_i \left(\alpha \cdot \sum_{\text{window}} \text{P-token}_i \;+\; \beta \cdot \sum_{\text{window}} \mathrm{BS}_i\right).$$

The fallback score of Preble is therefore a variant of the linear combination analyzed in §4.4.

Setup and tuning. We re-implement Preble inside LMETRIC's router framework for two reasons: (1) to enable an apples-to-apples comparison with our method, and (2) the open-source reference does not sustain our request rate, an issue tracked upstream [4].

We carefully tune the configurations of Preble to ensure a fair comparison. Preble has three knobs to tune[1]: the filter threshold T and the linear-combination coefficients. Tuning the combination space is expensive, so we focus on T and fix the linear-combination coefficients α and β using the profiling method described in Preble's paper. Surprisingly, we find T has little impact on performance (see Figure 31), and Preble's default ($T = 0.5$) is already optimal on our traces. As a result, all experiments in §6.2 use the published default $T = 0.5$.

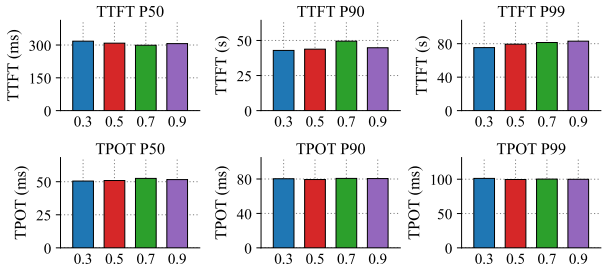


Figure 31: Performance of Preble as the filter threshold T varies on the ChatBot (Qwen) trace; model: Qwen3-30B.

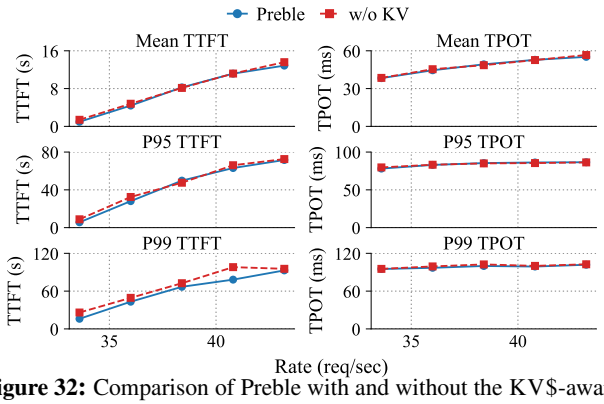


Figure 32: Comparison of Preble with and without the KV$-aware filter.

Comparison of Preble with and without the KV$-aware filter. To characterize how much of Preble's behavior comes from its KV$-aware filter on top of the linear-combination fallback (§4.4), we compare the default ($T = 0.5$) against $T = 1$, which disables the filter and routes purely by the fallback score. Figure 32 reports the comparison on the ChatBot (Qwen) trace with 16 Qwen3-30B instances. We can see that the filter yields a measurable improvement, but the bulk of Preble's performance on this trace is dominated by its linear-combination component, consistent with the T sweep in Figure 31 and our analysis in §6.2 (Figure 27).

A.2 More on PolyServe

PolyServe method in detail. PolyServe [56] is a simulator-based scheduler that optimizes a different objective: meet the SLO while creating a load gradient across instances that facilitates auto-scaling. Figure 33 shows its scheduler in our scheduling language. On each request, PolyServe first calls the simulator of §4.6 to predict per-instance TTFT and TPOT under the new request, conditioned on per-instance batch size

[1] The fallback score actually depends only on the ratio α/β, so the two coefficients carry one degree of freedom. Nevertheless, Preble's implementation exposes α and β as separate parameters, so we tune both to match it.

```
1  req = receive()
2  TTFT,TPOT = Simulator.predict(
                 req, BS, KV$, ...
               )
3  if ∀ instance, TTFT > SLO_TTFT or TPOT > SLO_TPOT:
       # Load-balancing branch
       sched_to = instances.select_min(TPOT)
5  else: # Utilization branch
6      sched_to = instances.filter(
                    TTFT ≤ SLO_TTFT and TPOT ≤ SLO_TPOT
                  ).select_max(TPOT)
7  req.forward(sched_to)
```

Figure 33: The pseudocode of PolyServe's simulation-based filter scheduler.

BS and KV$ footprint (line 2). It then branches on the simulator's output. If any instance meets the SLO bounds SLO_{TTFT} and SLO_{TPOT} (line 5), PolyServe takes the utilization branch: filter to feasible instances and route to the one with the highest predicted TPOT, i.e., the most loaded feasible instance (line 6). Otherwise (line 3), PolyServe falls back to load balancing and routes to the instance with the lowest predicted TPOT (line 4).

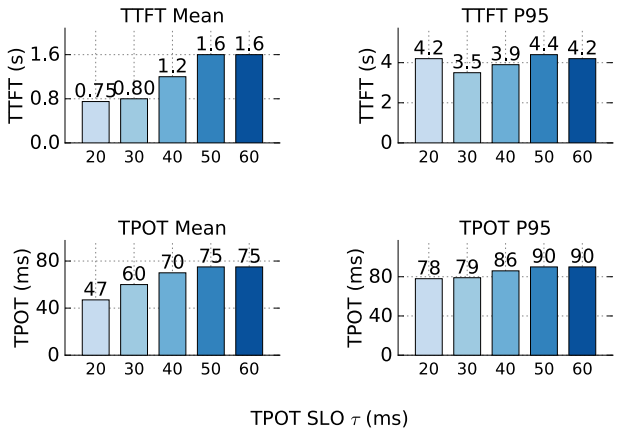


Figure 34: PolyServe end-to-end TTFT and TPOT under different TPOT-SLO thresholds τ on the ChatBot (Qwen) trace at 35.0 reqs/sec on the same 16 Qwen3-30B instances as §6.2.

Setup and tuning. PolyServe's scheduling is parameterized by the SLO bounds SLO_{TTFT} and SLO_{TPOT}, so its performance depends on the configured SLO. Following PolyServe's paper [56], we tune the SLO and adopt the best-performing setting in §6.2. Figure 34 reports the tuning of τ (denoting SLO_{TPOT}) on the ChatBot (Qwen) trace, with SLO_{TTFT} held fixed; we adopt $\tau = 20$ ms. We show only this τ sweep because SLO_{TTFT} had little impact on end-to-end performance on our traces.

ARTIFACT EVALUATED
USENIX
AVAILABLE

Prism: Cost-Efficient Multi-LLM Serving via GPU Memory Ballooning

Shan Yu[1], *Yifan Qiao*[2], *Mingyuan Ma*[3], *Yangmin Li*[4], *Shuo Yang*[2], *Xinyuan Tong*[5], *Yang Wang*[6], *Zhiqiang Xie*[7], *Yuwei An*[4], *Shiyi Cao*[2], *Ke Bao*[8], *Deepak Vij*[9], *Xiaoning Ding*[9], *Yichen Wang*[9], *Qingda Lu*[10], *Zhong Wang*[11], *Gao Gao*[12], *Harry Xu*[1*], *Junyi Shu*[1*], *Jiarong Xing*[13*], *Ying Sheng*[1*]

[1]*UCLA* [2]*UC Berkeley* [3]*Harvard University* [4]*CMU* [5]*University of Edinburgh* [6]*Intel* [7]*Stanford University*
[8]*LMSYS* [9]*ByteDance* [10]*Alibaba Cloud* [11]*Tsinghua University* [12]*Novita AI* [13]*Rice University*

Abstract

Inference providers must maintain availability for many LLMs, including low-volume but essential models, making resource efficiency increasingly important as token prices fall. Analysis of production traces reveals a dynamic *bursty-group pattern* in which sets of models become active together and shift over time; existing space- and time-sharing approaches lack principled mechanisms to adapt to this variability, forcing trade-offs between SLO adherence and efficiency. We observe that elastic memory allocation can unify spatial and temporal sharing. Based on this insight, we have developed Prism, a memory-centric LLM co-serving framework that applies *memory ballooning* to reclaim memory across models and support both forms of sharing under a single scheme. Prism's balloon driver, referred to as `kvcached`, has been open-sourced at `https://github.com/ovg-project/kvcached`, and deployed in production environments across 10K+ GPUs.

1 Introduction

Serving LLMs is costly for providers such as Hugging Face and Alibaba Cloud, which must host thousands of base and fine-tuned models, many with low request volumes yet mandatory availability [43, 53, 54]. As token prices fall and models vary widely in size and workload patterns, reducing inference cost while preserving performance has become a central objective. The core challenge, however, is not hardware expense but chronic under-utilization: to satisfy strict SLOs, industry practice often dedicates a *model-parallel GPU group* (*i.e.*, one or more GPUs jointly serving a single model instance), guaranteeing immediate responsiveness but wasting substantial capacity. This strategy is particularly inefficient for models with bursty or sparse traffic, and production analyses show that GPU duty cycles commonly fall below 30% [53, 54].

To improve utilization, prior work has explored various GPU sharing mechanisms at different granularities. Because auto-regressive LLM inference is intrinsically memory-bound, tail latency is governed primarily by memory stalls (*e.g.*, page faults, PCIe migrations). Moreover, compute sharing is already well studied—both via hardware mechanisms such as NVIDIA MIG [29] and vGPU [30], and via software solutions [10, 12, 13, 27, 42, 62]. In contrast, effective memory sharing remains comparatively underexplored. We therefore center this work on advancing effective GPU memory sharing.

The work of memory sharing for LLM serving falls into two categories: space sharing and time sharing. Space sharing (*e.g.*, MuxServe [11]) colocates multiple models on the same GPU to utilize unused memory and compute cycles, which is effective for smaller, low-traffic models that are continuously accessed. Time sharing (*e.g.*, QLM [33] and Aegaeon [53]) swaps models in and out of GPU memory, which is ideal for handling sporadic requests to models as the resources can be freed up when models are idle. While these strategies are effective for specific cases, our analysis of a broad range of production serving traces demonstrates that they fail to adapt to more complex and realistic usage scenarios.

In particular, we have conducted a comprehensive analysis of production traces drawn from major inference providers (*i.e.*, Hyperbolic and Novita AI) as well as independent evaluation platforms (*i.e.*, Chatbot Arena). These traces encompass 11 days to 16 months of traffic across 58 heterogeneous models. Our results indicate that real-world workloads display diverse, rapidly changing patterns that elude any single, fixed sharing mechanism.

Key observations. At the model level, deployed models show a *bursty-group* behavior: when a burst occurs, requests concentrate on a particular subset of models, and that subset shifts over time, similar to an application's dynamically changing working set. This pattern is driven by modern compound AI systems and agentic pipelines, which depend on a central reasoning LLM together with a number of fine-tuned or LoRA models for specialized agent tasks. At the request level, detailed inspection of the traces confirms that requests are extremely *dynamic and volatile* and fluctuate rapidly and unpredictably, limiting the value of existing GPU-sharing tech-

* Corresponding authors.

niques. Space sharing, alone, fails to adapt to longer idle periods; for instance, in the Novita AI trace, where models are idle for over 70% of the time, space sharing creates significant memory fragmentation, leaving 50% of GPU memory occupied but unused. In contrast, time sharing, alone, struggles with rapid workload fluctuations. The significant latency overhead of swapping (*e.g.*, several seconds) causes severe thrashing during traffic bursts, resulting in a spike of requests violating their latency SLOs due to queuing delays.

Maximizing resource utilization while preserving SLO attainment requires integrating both strategies: space sharing for colocating low-traffic models and time sharing for burst handling and idle-resource reclamation. However, mingling these approaches is far from trivial due to the fundamentally different challenges they must address—the flexibility of memory balancing for space sharing vs. the efficiency of model swapping for time sharing. One could attempt to switch spatial and temporal sharing strategies dynamically; yet, co-locating models with conflicting sharing needs creates resource contention. Specifically, a model requiring time-sharing typically relies on preempting resources to maximize burst performance, which directly contradicts the persistent resource availability required by a model colocated through space-sharing. Without a mechanism to fluidly mediate these conflicting operational modes, the system is forced into rigid configurations that cannot adapt to dynamic load changes.

Insight: a memory-centric view to unify space- and time-sharing. Our central observation is that *GPU memory is the unifying bottleneck, and each sharing strategy is concerned with a different portion of that memory*: time-sharing focuses on efficiently swapping in *LLM weights*, whereas space-sharing determines how to scale *KV-cache* capacity among concurrently served models. This perspective parallels *memory ballooning* in virtualization, where a hypervisor dynamically reclaims memory from guest VMs and reallocates it where needed. In the context of LLM co-serving, an analogous "balloon driver" can reclaim memory from idle models to swap in the weights of new models (supporting time-sharing) or from models with low request rates by shrinking their KV-cache reservations (supporting space-sharing). Treating GPU memory as an elastic resource allows the system to fluidly shift between sharing modes and even enable both simultaneously (*i.e.*, apply time-sharing to only a subset of colocated models), ensuring active models obtain the memory required to satisfy their SLOs while maximizing cluster-level efficiency.

Prism. We developed Prism, a new memory-centric GPU sharing system for multi-LLM serving. At its core, Prism implements the concept of memory ballooning for LLMs to enable *demand-aware, cross-model* memory sharing, a capability missing in existing systems but critical for unifying spatial and temporal strategies. Prism ensures that models can promptly expand their memory footprint to meet latency SLOs, while unused memory is efficiently harvested from others to maximize overall cluster utilization. The design of Prism addresses two key challenges.

Challenge 1: How to enable flexible memory ballooning for LLMs? Today's LLM serving engines (*e.g.*, SGLang and vLLM) are designed for single-model serving and adopt *per-model static* memory allocation. While techniques like `PagedAttention` manage memory efficiently within a model, they operate at the application level and cannot harvest memory across model boundaries.

Flexible memory ballooning requires a mechanism to dynamically resize the physical memory footprint of a model at runtime without disrupting its execution. Prism tackles this through a model balloon driver for serving engines (§5). The model balloon driver decouples virtual and physical GPU memory: serving engines reserve a large virtual address space during initialization, but physical memory is allocated and mapped only on demand. This design enables transparent, fine-grained memory redistribution at a 2 MB granularity with millisecond-level overhead, supporting diverse model architectures with zero code change to attention kernels.

Challenge 2: How to place models and schedule requests to maximize SLO attainment? While the balloon driver provides the mechanism for elasticity, maximizing SLO attainment requires intelligent policies to decide *when* and *where* memory should be allocated. Without proper coordination, contention for shared memory can lead to thrashing and performance degradation. Prism solves this through decoupled model placement and request scheduling. First, the fast-changing bursty groups (§3.1) require a placement algorithm that can quickly decide where each model should run. Prism tackles this with a global demand-complementary placement algorithm that minimizes the KV pressure ratio (KVPR), balancing model memory demand with GPU capacity (§6.1). Second, volatile request arrivals and diverse SLO targets require estimating each request's time slack so the memory manager can steer memory toward latency-sensitive tasks. Prism achieves this via a local slack-aware arbitration algorithm, which leverages a GPU-level request queue to prioritize requests based on their slack (§6.2).

Results. We implemented Prism on top of SGLang [64], a widely used LLM serving engine. We evaluated Prism comprehensively with two production traces, and with **58 representative LLMs of varying architectures and sizes, on GPU clusters with up to 32 H100 GPUs.** Our results show that Prism achieves up to 3.3× higher time-to-first-token (TTFT) SLO attainment and 2× higher time-per-output-token (TPOT) SLO attainment given the same number of GPUs. When targeting the same level of SLO attainment, Prism achieves up to over 2× cost reduction or 3.5× more requests compared to the state-of-the-art.

Prism's balloon driver, referred to as `kvcached`, has been open-sourced and actively maintained at https://github.com/ovg-project/kvcached. `kvcached` has been adopted by many industry users to support production work-

Trace name	Service provider	# models	Time span
Hyperbolic	Hyperbolic [19]	24	4 months
Novita	Novita AI [3]	16	1 month
Arena-Battle	Chatbot Arena [9]	129	16 months
Arena-Chat	Chatbot Arena [9]	84	11 days

Table 1: Production trace summary.

loads in clusters of more than 10K GPUs. We report usage analyses from two companies at the end of §7.

2 Background

SLOs of Online LLM Serving. In online LLM serving, unlike offline batch processing, the system must interact with users or downstream applications in real-time. Consequently, providers enforce strict SLOs to guarantee the quality of experience. Two key latency metrics define these objectives. First, *Time-To-First-Token* (TTFT) measures the latency to process the input prompt and generate the first token, which is a crucial metric for interactive applications, such as chatbots. Second, *Time-Per-Output-Token* (TPOT) measures the average inter-token latency during the autoregressive generation, which is essential to ensure that text generation is fluid and matches human reading speeds.

Memory footprint of LLM serving. LLM inference is inherently memory-bound, with GPU memory dominated by model weights and KV cache.

Model weights. LLM weights are massive; for example, a 70B-parameter model requires roughly 140 GB of GPU memory in FP16. Under long-tail workloads [53], idle models waste GPU memory; prior work [13, 53] mitigates this with model eviction and accelerated weight loading.

KV cache and PagedAttention. The KV cache stores intermediate Key (K) and Value (V) tensors for attention layers to avoid redundant computation. Unlike weights, the KV cache is dynamic, growing with sequence length and batch size.

To improve GPU memory efficiency, mainstream LLM serving systems adopt PagedAttention [21], which splits each request's KV cache into fixed-size blocks that can be stored non-contiguously and reused across requests. Frameworks such as SGLang [64] and vLLM [21] implement this design by reserving large PyTorch tensors as a KV block pool and mapping each request's logical blocks to physical offsets in the pool. For each of the L attention layers, a K tensor and a V tensor of shape (B, T, H, D) are allocated, where B is the number of blocks, T is the tokens per block, H is the number of attention heads, and D is the head dimension. The values of L, H, and D vary across model architectures (*e.g.*, Llama-3-8B $(32, 8, 128)$ vs. GPT-OSS-120B $(36, 8, 64)$), resulting in different KV block sizes.

3 Production Trace Analysis

Prior studies have primarily profiled single-model LLM services [48, 50], but the cross-model dynamics that arise in multi-LLM serving remain largely under-examined. As a result, evaluations in recent multi-LLM serving work often rely on synthetic or simplified workloads, such as Poisson/Gamma arrivals [11, 33, 53], temporally rescaled traces from non-LLM domains [13, 14], or dataset-driven replays [39].

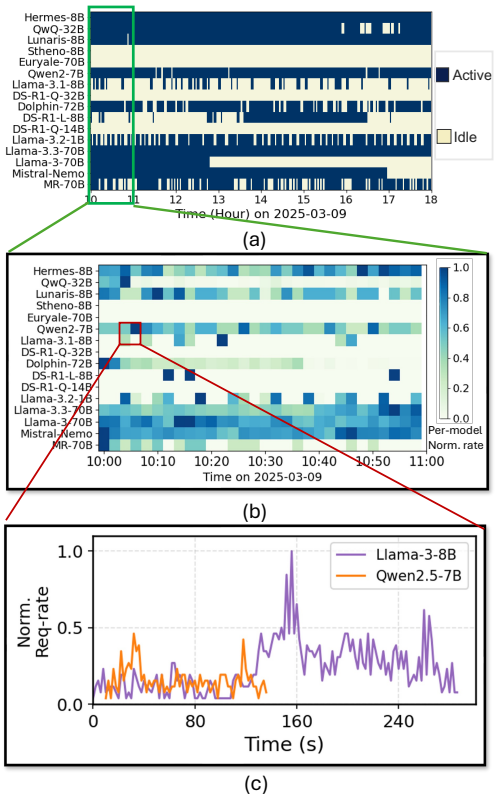


Figure 1: Combined view of model-request dynamics; the x-axes denote time. (a) Variation of concurrently active model groups over a six-hour window (8am–4pm): the y-axis lists all models from the Novita trace; each cell represents a three-minute interval, with dark/-light shading indicating whether the model is active. (b) Zoomed-in view of volatile request patterns from 11am–1pm, showing per-model normalized request rates, where darker colors indicate higher rates. (c) A further zoom-in over a 5-minute window from (b), comparing two models on a shared normalization.

To address these gaps, we conduct a detailed characterization of production workloads from major inference providers (*i.e.*, Hyperbolic and Novita AI) as well as independent evaluation platforms (*i.e.*, Chatbot Arena), explicitly capturing the fine-grained, time-varying dynamics of multi-LLM traffic. The four traces span 24–129 deployed models over weeks to months of traffic, summarized in Table 1. In this section, we utilize the traces from Novita AI as a representati4ve example to illustrate our observations and report cross-trace statistics. We include a detailed CDF analysis in Appendix A.1.

3.1 Model Level: Shifting Bursty Groups

We begin by examining the dynamics of concurrently active models (*i.e.*, those processing at least one request) from the

Novita trace over a six-hour span in Figure 1(a), where a horizontal slice of a model represents its active/idle activities over time, and a vertical slice at a point of time reveals which subset of the models is active at the point. Our analysis reveals two properties that are common for production workloads: *bursty groups* and *heterogeneous access patterns*.

Vertical behavior: bursty groups. Vertically, deployed models demonstrate a *bursty-group* behavior: models receive requests in short, irregular bursts separated by long and variable idle intervals. Because these bursts occur asynchronously across models, only a small subset is active at any moment, and this subset changes rapidly over time. We refer to this transient set as the *bursty group*, analogous to how an application's *working set* changes during its execution—there are common models that appear in almost every group but most models experience sporadic requests and hence appear only in a small number of groups. Across the four traces, 23%–50% of models are active concurrently on average, and the active set changes 54–766 times per hour. Although the total number of deployed LLMs can be large, only the bursty group drives actual demand. Efficient GPU utilization thus requires adapting resources to this shifting model group, rather than allocating resources statically to all models.

Horizontal behavior: Heterogeneous activation patterns. Horizontally, the workload exhibits significant heterogeneity across models in activation patterns. Some models, such as `Llama-3.3-70B`, sustain long stretches of continuous activity, while others (like `Distilled-Deepseek-R1-Qwen-14B(DS-R1-Q-14B)`) activate only in sparse, short-lived instances with only a small group of applications/users using it. This heterogeneity is largely driven by the diverse roles that models play in modern compound AI systems and agentic pipelines. Large general-purpose models (*e.g.*, `Llama-3.3-70B`) often act as central reasoning or planning components and remain active throughout user interactions, leading to sustained request streams. In contrast, smaller distilled models (*e.g.*, `DS-R1-Q-14B`) are typically invoked as auxiliary components—such as for tool use or verification—and are triggered only at specific pipeline stages, resulting in intermittent, bursty activations.

Aggregating requests across many applications and users yields a mixed access pattern in which some models remain persistently active, others appear only sporadically, and even the same model may exhibit different behaviors over time. These differences have direct implications for resource management: long-running models benefit from sustained colocation, whereas short and scattered bursts favor opportunistic execution. Consequently, multi-LLM serving systems must adapt to the heterogeneous activation behavior of each model, rather than assuming uniform access patterns.

Implication: Inefficiency of static partitioning. The existence of shifting *bursty groups* fundamentally challenges any static strategies. Since only a small, changing subset of models drives demand at any given moment, statically partitioning GPU memory across all models (to ensure availability) results in severe resource fragmentation. In the Novita AI trace, for example, models are idle for over 70% of the time on average. Under a static memory-partitioning regime, the memory reserved for these idle models is wasted, preventing active models in the current bursty group from utilizing that capacity to scale up and meet their SLOs.

3.2 Request Level: Dynamic and Volatile

We next zoom in to a shorter period of time to examine the fine-grained dynamics of individual models. Figure 1(b) depicts the request rates for the models over a two-hour window, normalized within each model so that low-volume bursts remain visible alongside high-volume ones.

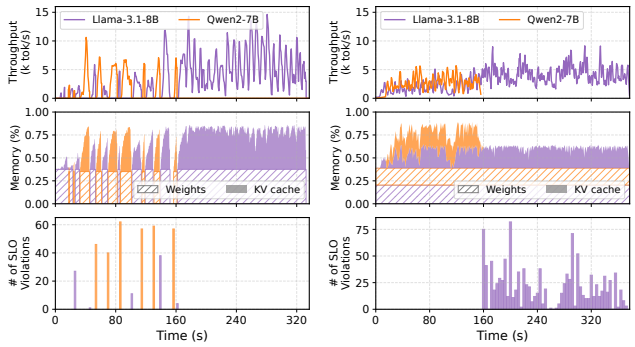
Extreme volatility and pattern shifts. As shown in Figure 1(b), request streams shift patterns frequently. Even models that appear continuously active in coarser views actually exhibit rapid oscillations between low-demand periods and short, intense bursts (*e.g.*, `Qwen2-7B`). Similarly, models that are only sporadically invoked display burst sizes and inter-arrival times that vary significantly over time (*e.g.*, `Llama-3.1-8B`).

Abrupt pattern transitions. Furthermore, these workload patterns often undergo rapid, unforeseen transitions. Figure 1(c) provides a granular, 5-minute zoom-in view of request arrivals for two representative models. This window reveals a stark shift in interaction dynamics: a phase of moderate, interleaved activity—where both models contend for resources simultaneously—abruptly gives way to a concentrated burst from a single model. Such sudden pattern shifts occur without discernible triggers, implying that the system cannot rely on historical stability or long-term pattern recognition; instead, it must possess the agility to react to instantaneous changes. Such transitions are pervasive: across our traces, many models exhibit a request rate coefficient of variation above 1 and 40–100 idle intervals per hour.

Implication: Instability of pure time sharing. The extreme *volatility* and rapid pattern transitions undermine pure time-sharing strategies. As seen in Figure 1(c), request streams frequently transition into phases of interleaved activity where multiple models are active simultaneously. In these scenarios, a time-sharing system relying on swapping must constantly evict and reload weights to serve alternating requests. This leads to model thrashing, where the latency overhead of loading weights dominates execution time, causing immediate SLO violations. This volatility necessitates the stability of space sharing, where working sets remain resident to handle interleaved demand without swapping penalties.

3.3 Applying Existing Approaches

To understand the implications of these abrupt pattern transitions, we ran two representative state-of-the-art strategies,



(a) Pure time sharing. (b) Pure space sharing.

Figure 2: Performance of pure time- (a) and space-sharing (b) for the trace in Figure 1(c). Pure time sharing in (a) causes model switching when both models have requests, while pure space sharing in (b) causes queuing delay during requests spike.

time sharing (QLM [33]) and *space sharing* (static partition [1, 29]), on the trace segment shown in Figure 1(c). Figure 2 illustrates the resulting memory usage and cumulative SLO violations for each approach.

Failure of time sharing. Time sharing manages resources by swapping models in and out of GPU memory. This approach struggles during the initial phase of *interleaved activity*. As both models receive frequent, overlapping requests, the system is forced to repeatedly evict and reload weights to serve alternating inputs. As shown in Figure 2a, this leads to severe *model thrashing*: the GPU spends more time on PCIe transfers and engine re-initialization than on computation. As such, request queues grow rapidly, resulting in a spike of SLO violations even before the workload intensity peaks.

Failure of space sharing. Space sharing avoids swapping by partitioning GPU memory between models. While this provides stability during the interleaved phase, it fails catastrophically when the workload transitions to the burst phase, as shown in Figure 2b. Although one model becomes idle, its model weights remain locked, preventing the active model which is now experiencing a demand surge from utilizing the full GPU capacity. This rigid isolation creates artificial resource scarcity: the active model is starved of KV cache memory while adjacent memory sits idle. The result is a throughput cap that causes significant queuing delays and SLO violations during the burst.

3.4 Takeaway: A Hybrid Approach

Our analysis demonstrates that production multi-LLM workloads exhibit conflicting requirements that no single existing strategy can satisfy:

- **Bursty groups benefit from time-sharing capabilities:** At the model level, the active set shifts over time. Pure space sharing fails here because it rigidly locks memory for idle models, causing fragmentation and preventing active models from scaling. The system must be able to evict idle resources to maximize utilization.
- **Volatile requests benefit from space-sharing capabilities:** At the request level, demand fluctuates rapidly and often interleaves across models. Pure time sharing fails here because swapping causes severe thrashing during these volatile periods. The system must be able to keep active models colocated in memory to provide low-latency access.

Motivation for Prism. These two patterns capture the workload at coarse- and fine-grained levels, respectively. Because they reflect different facets of the same underlying behavior, they cannot be addressed in isolation, necessitating a hybrid approach: the system must fluidly shift between the strategies at runtime. It needs the *elasticity* to act like a time-sharing system when reclaiming memory from the idle models, while simultaneously acting like a space-sharing system to co-serve the models in an active bursty group without causing SLO violations. This calls for a design that manages GPU memory as a dynamic, shared resource rather than static partitions.

4 Overview

Prism is a memory-efficient GPU sharing system for cost-effective multi-LLM serving. Figure 3 shows its system architecture and design overview. Prism receives inference requests at its frontend, which routes them to the corresponding LLMs for processing. To improve cost efficiency, Prism serves LLMs with flexible combinations of space and time sharing. For instance, high-rate models may occupy GPUs exclusively, while multiple low-rate models can be colocated on a single GPU or a model-parallel GPU group. In Prism, a GPU group represents the strict scheduling boundary—a set of GPUs tightly coupled to jointly serve one model instance. Prism's global scheduler treats each of these groups as a distinct, indivisible resource unit. Idle models are temporarily evicted to CPU DRAM and reactivated when new requests arrive. Prism continuously adjusts its sharing strategies based on runtime workload to maximize overall SLO attainment.

Prism achieves this with two key designs for flexible, demand-aware cross-model memory sharing and coordination. First, it introduces a balloon driver, referred to as `kvcached` (§5) that enables flexible and efficient memory sharing between models, forming the foundation for rapid adaptation to workload variations and policy changes. Second, Prism optimizes the overall resource allocation through a memory-centric control plane that uses a combination of (1) load-aware model placement (§6.1) that maximizes memory headroom for GPUs to facilitate ballooning and (2) slack-aware request arbitration that prioritizes requests on each GPU for maximizing SLO attainment (§6.2).

Relation with autoscaling. Prism is orthogonal to and can be seamlessly integrated with autoscaling [4,23,40,59]. Prism focuses on efficiently sharing GPUs between models (replicas)

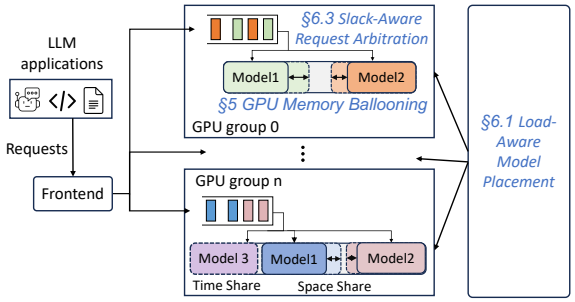


Figure 3: The system architecture and design overview of Prism.

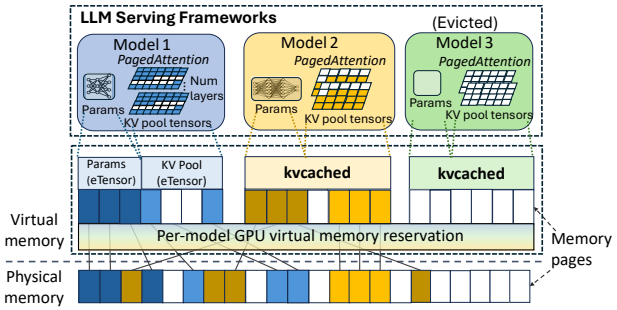


Figure 4: Memory ballooning for effective multi-LLM serving.

while autoscaling automatically scales up/down the number of model replicas.

5 GPU Memory Ballooning for Multi-LLMs

This section describes how Prism enables flexible cross-model memory sharing with our balloon driver, providing a foundation for adapting policies to dynamic workloads.

5.1 Requirements and Challenges

Mainstream LLM serving engines [21, 64] adopt block-based KV token management (*e.g.*, PagedAttention [21]). While this design effectively reduces *request-level* memory fragmentation *within a single* LLM, it requires the application (serving engine) to manage the GPU physical memory directly, by pre-allocating a large portion of GPU memory as the KV block pool. This memory thus remains occupied even when there is no active request, and the KV blocks within the tensor are unused. This prevents the redistribution of unused memory across different LLMs, limiting the ability to support flexible time- and space-sharing among multiple LLMs. This leads to four key requirements for Prism.

R1: Fast memory reallocation between model weights and KV cache. When a model is swapped out, the memory released by its weights should be quickly repurposed by other models as KV cache. Similarly, when an idle model is reactivated upon new requests, the running models must promptly release unused KV cache memory for the new model's weights and KV cache. However, this is hard with statically reserved KV pool tensors, whose physical GPU memory cannot be partially released. A workaround is to copy active token blocks into a smaller tensor and then destroy the old one. However, this not only requires waiting for the slow copy to complete, but also demands enough GPU memory to hold both tensors during the transient period.

R2: KV cache sharing across diverse model architectures. For models colocated on a GPU, their KV cache memory should be flexibly shared; *e.g.*, unused KV cache memory from one LLM can be easily reused by others. However, LLMs often have diverse KV cache layouts, resulting in different tensor shapes, *e.g.*, different head dimensions and number of attention heads. Therefore, simply using a pool of unified KV tensors across models is infeasible: different token sizes lead to misaligned tensor shapes, and different layer counts cause inconsistent numbers of tensors.

R3: Minimal redistribution overhead and memory fragmentation. Since memory redistribution may occur frequently, it must incur low overhead while minimizing fragmentation to improve utilization. This rules out a strawman design that splits the reserved KV pool tensor into multiple smaller segments so that free segments from one model can be reused by another. This is because choosing the segment size is inherently challenging: too many small segments increase second-level indexing overhead, while too few large segments cause severe fragmentation.

R4: Transparent to existing serving frameworks. The new mechanism should remain fully compatible with existing serving frameworks, where the KV block pool is allocated as large tensors, and require no changes to attention kernels. The previously mentioned segment-based method violates this requirement, as it splits the tensor into segments, which necessitates extensive kernel modifications.

Summary. Multi-LLM serving must efficiently cope with heterogeneous memory demands (*e.g.*, different model weights and KV token blocks), each with distinct semantics (*e.g.*, head dimensions and layer counts) and lifecycles. Existing single-LLM systems manage GPU memory entirely at the application level, imposing their own semantics on top of a fixed GPU allocation. This approach works well for a single model, which can efficiently manage its memory in isolation. In multi-LLM settings, however, such isolation becomes a barrier, as memory needs to be shared and reallocated across models dynamically to accommodate bursty group behavior and workload volatility.

5.2 Solution: GPU Memory Ballooning

To meet all the above requirements, our key insight is that memory management should be *pushed down to the GPU runtime level.* At this level, the system can treat all models uniformly, allowing GPU memory to be transparently redistributed across models, while applications manage only the memory allocated to them with their own semantics.

Building on this idea, Prism introduces a *balloon driver*, named `kvcached`, which sits as a shim between inference engines and GPU memory. Open sourced at https://github.com/ovg-project/kvcached, `kvcached` manages the entire GPU physical memory for LLM serving, including both model weights and the KV cache pool. Meanwhile, `kvcached` reserves a large contiguous virtual address space for each engine, which presents it to the engine as if it were dedicated GPU memory. On top of this reserved space, `kvcached` translates application-level semantics (*e.g.*, weights, KV caches, and intermediate buffers) into allocations of physical GPU memory pages. Physical pages are created only on demand and mapped to virtual addresses lazily, allowing memory to expand and shrink as workloads change, as illustrated in Figure 4.

Specifically, `kvcached` meets all the requirements in §5.1 through the following designs.

D1: Unified model weights and KV cache management. `kvcached` achieves fast memory redistribution between model weights and KV cache through unified virtual and physical memory management. Since `kvcached`-managed memory is agnostic to application semantics, both weights and KV cache can be seamlessly reallocated across engines. Moreover, `kvcached` dynamically adjusts physical memory limits: when an evicted model is reactivated, it shrinks the limits of other models on the same GPU, bounding their allocations and immediately freeing space for the new model. The opposite happens when a model is evicted.

D2: Automatic token block mapping. To support models with different layer counts and token sizes, `kvcached` employs an internal KV cache manager to map token blocks onto underlying virtual and physical GPU pages. Prism assigns token blocks to available slots within a virtual page, or allocates a new physical page when necessary. To prevent size conflicts and ensure portability across architectures, the KV cache manager further segregates token blocks from different models onto distinct memory pages.

D3: Overhead and fragmentation optimizations. `kvcached` minimizes memory redistribution overhead and fragmentation through three key optimizations. First, mainstream serving frameworks maintain separate K and V tensors per layer, requiring $2L$ page allocations across all layers each time. To reduce this overhead, `kvcached` reorganizes the memory layout in virtual space, so that all layers' K and V vectors of a token are stored in contiguous virtual space, requiring only one batch allocation for all pages ($2L\times$ speedup). Second, `kvcached` uses a pre-allocation thread to asynchronously prepare a small buffer of GPU pages. Engines draw new pages from this buffer first, while released pages are returned to it and only physically freed if the buffer exceeds its limit or memory must be reclaimed for a new model. Finally, to reduce fragmentation, `kvcached` uses 2MB memory pages, and prioritizes using partially filled pages.

D4: Elastic tensor abstraction. `kvcached` ensures transparency with existing LLM serving frameworks by introducing an *elastic tensor (eTensor)* abstraction, implemented via PyTorch's extension interface to abstract away the details of physical page allocation from the serving engine. Elastic tensors can be used exactly like regular PyTorch tensors, requiring no modifications to attention kernels and remaining fully compatible with CUDA graph optimizations commonly used in LLM serving frameworks.

5.3 Fast Model Loading

Model swapping speed directly impacts the flexibility of GPU sharing. High latency can hinder the timely swapping of models with strict SLOs, limiting policy adaptability. While deactivation is straightforward, *i.e.*, terminating the engine and releasing all memory, reactivation is more complicated, which involves: (1) initializing a new serving engine and reserving a new virtual address space for KV cache pool; and (2) loading the model weights from CPU DRAM. If done naïvely, this process can take tens of seconds—far exceeding the TTFT SLOs of online LLM inference, which are often within a few seconds or less.

Reusable engine pools. The root cause of (1) lies in the tight coupling between the engine and the model it serves. In current systems, an engine shares the same lifecycle as its model—when a model is evicted, its engine is also terminated, along with its virtual address space. As a result, every model activation incurs the full cost of engine initialization.

Prism eliminates this overhead by decoupling the lifecycles of the engines and models. Specifically, it maintains an engine pool on each GPU, where engines are pre-initialized with virtual address space and distributed contexts. Upon model activation, Prism selects an available engine from the pool and starts model loading directly. When a model is evicted, its physical memory is released, but its engine with virtual address space is returned to the engine pool for future reuse. However, an engine cannot directly reuse previously reserved virtual memory space to serve a new model. This is because current inference engines perform index-based token access, which depends on a model-specific memory layout that is incompatible with models of different architectures, *e.g.*, different numbers of layers or token sizes. To address this, Prism introduces a KV cache virtual memory manager to manage the pre-reserved virtual memory spaces in the engine pool. When a new model is loaded, the manager dynamically aligns the reserved virtual space to match the memory layout required by the new model (one-time effort), and then creates a new `kvcached` based on the aligned memory spaces. The `kvcached` can then correctly and efficiently locate the virtual memory page where each token resides during inference.

Parallel model weight loading. The time spent on (2) model weight loading is heavily influenced by the utilization of the CPU–GPU interconnect bandwidth. We found that loading models naïvely via the `cudaMemcpyAsync` API

to a GPU fails to saturate the interconnect bandwidth, even when invoked from multiple threads. This may be due to all `cudaMemcpyAsync` operations targeting the same GPU executing serially, limited by the CUDA driver and hardware. Prism overcomes this bottleneck by chunking model weights into smaller segments, loading them in parallel across multiple GPUs on the same node, and then aggregating them to the target GPU via high-speed NVLink interconnects. This parallelized strategy significantly accelerates model loading. To minimize interference with running workloads on GPUs, Prism partitions model weights at the granularity of individual weight tensors and loads them in a streaming fashion. As a result, each GPU only needs to maintain a small buffer (*e.g.*, 30MB), minimizing possible memory contention.

6 A Memory-Centric Control Plane

The memory ballooning mechanism in §5 provides the elasticity needed to unify spatial and temporal sharing, but leveraging it effectively requires policies that adapt to evolving bursty groups and volatile request patterns. Our goal is to improve memory efficiency and maximize SLO attainment for TTFT and TPOT through coordinated memory sharing. Because TTFT and TPOT can interfere [65], we prioritize TTFT, whose prompt length is known, while noting that the resulting improvements in resource scheduling also benefit TPOT by reducing memory-induced preemptions.

Challenges. Unifying time and space sharing complicates scheduling because decisions must jointly span models, requests, and tokens, all of which shape GPU memory usage and latency. These layers are tightly coupled: model residency sets memory availability, request concurrency determines consumption, and token execution affects both utilization and SLOs. Optimizing any one dimension in isolation—e.g., maximizing colocation or strictly protecting SLOs—can destabilize the others. Consequently, time- and space-sharing choices form a large, interdependent decision space where placement, memory pressure, and request scheduling continually interact in non-trivial ways.

A natural formulation of this problem is a joint optimization over model placement, request scheduling, and memory allocation (*e.g.*, via an ILP), but this quickly becomes intractable. The placement space alone is enormous: with M models and N GPUs, there are N^M possible assignments even before considering migration decisions. Request scheduling is likewise combinatorial, as differing prompt lengths, arrival times, SLOs, output lengths, and engine states yield a factorial explosion in possible execution orders. Moreover, placement and scheduling are tightly coupled—migrating even one model changes which GPUs can serve which requests and how KV caches evolve, thereby altering future scheduling decisions. All of this must be decided online, with the scheduler observing only recent arrivals while lacking knowledge of future bursts, output lengths, or concurrency.

Insight. Our key insight is that *this complexity simplifies when managed through a memory-centric lens*. Models, requests, and tokens influence system behavior primarily through how they consume and contend for GPU memory, while SLO attainment is directly governed by the availability of that memory at execution time. By making memory the central optimization target, we unify multiple interacting objectives, utilization efficiency, latency control, and stability, under a single principle: controlling memory contention. This reduces a multi-metric, multi-level scheduling problem into a tractable design centered on memory management.

Memory contention manifests at multiple scopes: cluster-level model activity drives coarse-grained memory demand, while fine-grained request dynamics determine SLO outcomes. No single control loop can manage both effectively, motivating a memory-centric hierarchical scheduler: a macro, cluster-level plane that shapes memory pressure via model placement, and a micro, GPU-level plane that allocates memory per request to satisfy heterogeneous SLOs. Coordinating these layers enables Prism to maintain high utilization and robust SLO attainment.

6.1 Load-Aware Model Placement

The objective of the global model placement strategy is to place models across GPUs to maximize the headroom available for memory ballooning. If too many models from the same active bursty group are colocated, they will simultaneously demand memory for KV-cache expansion, leading to severe contention and out-of-memory errors.

To prevent this, Prism employs a load-aware placement strategy. Because predicting exact memory usage is difficult due to unknown output lengths of LLM inference requests, we utilize a robust heuristic referred to as the KV Pressure Ratio (KVPR). KVPR quantifies the intensity of memory contention on a GPU by comparing the aggregate urgency of memory demand against the remaining capacity, calculated as $\frac{w_token_rate}{shared_kv}$, where $w_token_rate = \frac{token_rate * token_size}{SLO}$ represents the SLO-weighted token memory usage rate of a model, indicating its memory demand per unit time. By counting both input tokens from newly admitted requests and decode tokens produced by running requests per unit time, $token_rate$ captures the full KV-cache growth rate and helps KVPR accurately reflect GPU memory pressure. We use the TPOT SLO for SLO because autoregressive decoding dominates generation latency and is especially sensitive to KV-cache memory contention. $shared_kv$ is the memory available for the KV cache on a GPU. A high KVPR indicates a high-pressure GPU where memory ballooning is likely to be stifled.

Algorithm 1 realizes our model placement strategy. Prism prioritizes the most aggressive memory consumers by sorting models in descending order of their SLO-weighted token usage rates w_token_rate, ensuring that high-demand workloads are allocated resources first (Lines 1–3). For each model, the scheduler identifies the GPU that minimizes the resulting

KVPR. This step inherently enforces complementarity: by targeting the device with the lowest existing pressure, Prism effectively colocates high-demand models with low-demand ones, thereby maximizing the capacity available for the high-demand model to balloon its memory usage (Lines 5–8). If the selected GPU differs from the model's current GPU, Prism migrates the model accordingly. However, this migration incurs overhead from engine switching and model weight loading. To avoid unnecessary migrations with marginal benefit, Prism compares the KVPR of the best and current GPUs, and proceeds only if the improvement exceeds a threshold τ (Line 8). Finally, Prism assigns the model to its selected GPU and updates the corresponding GPU states (Lines 9–12).

Analysis. This greedy approach approximates the optimal solution by bounding the maximum KVPR across the cluster, ensuring that no single GPU becomes a disproportionate bottleneck that stifles memory sharing. A detailed formal analysis is provided in Appendix A.2. The algorithm also seamlessly supports large models utilizing Tensor Parallelism (TP). By treating each TP partition as a distinct scheduling unit with anti-affinity constraints, Prism ensures they are placed on separate GPUs, thereby aggregating memory capacity across devices to satisfy massive KV demands.

Model eviction and activation. The placement strategy evicts a model if it remains idle and GPU resources become constrained for other models. Prism performs eviction based on an empirical threshold, which can be determined by analyzing the idle interval distribution; a detailed sensitivity analysis is provided in Appendix A.4. When the model receives new requests, Prism immediately reactivates it by placing it on the GPU with the lowest KV pressure ratio, drawing from a pre-warmed engine pool with reusable distributed contexts and buffers to avoid re-initialization, and using parallel weight loading to keep cold-start latency within TTFT budgets (§5.3).

Model migration. Model migration is designed to preserve TTFT during placement changes. Rather than stopping request processing while the destination instance is prepared, Prism keeps the source instance active and continues serving requests until the target is ready. This allows migration latency to be overlapped with ongoing execution, so requests experience only the short switch-over delay. Therefore, Prism's effectiveness does not depend on fast interconnects. Practically, when the source and target GPUs are connected via NVLink, Prism uses NVLink as an optional shortcut for migrating model weights and resident KV caches. In non-NVLink environments, Prism leverages GPUDirect RDMA [31] if available, or falls back to standard eviction and reactivation, which takes at most a few seconds for <70B models.

6.2 Slack-Aware Request Arbitration

While global placement balances demand, memory contention arises locally when resident models process requests concurrently. Under unified sharing, strict isolation causes fragmentation, whereas uncontrolled sharing risks starvation, with urgent requests blocked by long-running ones. Models on the same GPU may compete for KV cache memory, and without coordination a high-rate, relaxed-SLO model can monopolize memory and degrade SLO attainment for stricter workloads.

A naïve approach is to cap each model's memory usage, but choosing appropriate limits is difficult under dynamic workloads and heterogeneous SLOs: conservative caps throttle throughput, while generous ones starve other models. Adapting limits at runtime helps but cannot take effect immediately, since memory must be freed by completing in-flight requests—a process that can take seconds depending on request length and load. This coordination challenge arises because each engine maintains its own queue and greedily schedules requests as memory becomes available.

To resolve this, Prism employs a slack-aware request arbitration strategy. Instead of maintaining separate queues per model, Prism uses a shared per-GPU request queue that arbitrates access to each GPU's physical memory pool to coordinate the memory usage for heterogeneous SLO requests. Our goal is to prioritize requests that are most critical for maximizing SLO attainment. Prism goes beyond simple deadline prioritization by leveraging the exact time slack of each request, which is defined as the buffer between a request's deadline and its required execution time.

With both the deadline (d_r) and execution cost (e_r) known, Prism transforms the scheduling challenge into a classic deadline problem. We adopt the Moore-Hodgson algorithm [25], which minimizes the number of deadline misses. As shown in Algorithm 2, given a set of requests R, Prism first sorts them in ascending order of their prefill completion deadlines (Line

Algorithm 1 Model Placement to Maximize Memory Headroom.

Require: Number of GPUs N, GPU memory capacity C, migration threshold τ, and M models. Each model m_j has: token rate t_j, weight w_j, token size tz_j, current device index g_j, and latency SLO s_j.

Ensure: Assign each model to a GPU to balance the resource demand and remaining memory capacity.

1: Sort models by $\frac{t_j * tz_j}{s_j}$ in descending order. Denote the sorted sequence as $m_1, m_2, \ldots, m_M$.
2: **for** $i = 1$ to N **do**
3: $\quad shared_kv_i \leftarrow C$; $w_token_rate_i \leftarrow 0$
4: **for** $k = 1$ to M **do**
5: $\quad$ /* find the GPU *best_idx* that minimizes KVPR */
6: $\quad best_r,\ best_idx \leftarrow (\min, \text{argmin}) \frac{w_token_rate_i}{shared_kv_i}$
7: $\quad current_r \leftarrow \frac{w_token_rate_{g_k}}{shared_kv_{g_k}}$
8: $\quad best_gpu \leftarrow \begin{cases} best_idx, & \text{if } current_r - best_r > \tau \\ g_k, & \text{otherwise} \end{cases}$
9: $\quad$ Assign model m_k to $best_gpu$
10: $\quad w_token_rate_{best_gpu} \leftarrow w_token_rate_{best_gpu} + \frac{r_k}{s_k}$
11: $\quad shared_kv_{best_gpu} \leftarrow shared_kv_{best_gpu} - w_k$
12: **return** Model-to-GPU placement

Algorithm 2 GPU-Local Request Scheduling

Require: A set of n requests $R = \{r_1, r_2, \ldots, r_n\}$. Each request r_i has: a prompt length p_i, a chunked-prefill speed c_i determined by the model serves it, a TTFT SLO s_i, and an arrival time a_i.

Ensure: A subset of requests $S \subseteq R$ that can be executed in order to maximize TTFT SLO attainment.

1: Sort R in ascending order of deadlines $d_i = a_i + s_i$: $r_1, r_2, \ldots, r_n$ such that $d_1 \leq d_2 \leq \cdots \leq d_n$.
2: Initialize $S \leftarrow \emptyset$, *current_time* $\leftarrow$ Timer.time()
3: **for** $k = 1$ to n **do**
4: Let $r \leftarrow r_k$, $e_r \leftarrow \frac{p_r}{c_r}$
5: Append r to S
6: Update *current_time* $\leftarrow$ *current_time* $+ e_r$.
7: **if** *current_time* $> a_r + s_r$ **then**
8: /* pop the request with longest execution time */
9: Let $r_{\max} \leftarrow \arg\max_{r' \in S} \frac{p_{r'}}{c_{r'}}$
10: Remove $r_{\max}$ from S
11: Update *current_time* $\leftarrow$ *current_time* $- \frac{p_{r_{\max}}}{c_{r_{\max}}}$
12: **return** S

1). Then, for each request in the sorted list, Prism appends it to the schedule list S and checks whether it can finish within its TTFT SLO (Lines 3-7). If the most recently added request cannot meet its deadline, Prism evicts the request in S with the longest execution time (Lines 9–11). It then moves to the next request and continues this process until evaluating all requests. Finally, Prism dispatches the accepted requests in S following their order in the schedule.

Analysis. Prism focuses arbitration on TTFT attainment because it gates when a request can start; once admitted, a request runs through decoding under its TPOT SLO, which is governed by memory headroom rather than queue ordering. Long-running decode requests are preempted only when memory is severely constrained, so a single heavy request cannot stall many newly admitted ones, matching the batch scheduling policies of SGLang and vLLM [21, 64].

Request scheduling ensures optimal TTFT attainment when chunked-prefill has prefill running at each inference step. This is because the prefill time e_r of a request r can be estimated as $e_r = \frac{p_r}{c_r}$, allowing us to compute the prefill completion time of any request r_i in a sequence using $d_{r_i} = a_{r_i} + \sum_{i=1}^{n} \frac{p_{r_i}}{c_{r_i}}$, where n is the number of requests (including r_i) waiting for processing. With this information, we can follow the proof of the original Moore-Hodgson algorithm [8] to prove the optimality of our scheduling algorithm. Our admission control ensures prefill runs at each inference step (so no starvation) by admitting a proper number of requests to each engine and preempting long decoding requests.

7 Evaluation

We implemented a prototype of Prism with ∼10,400 lines of Python and 774 lines of C++ code. As the serving backend, we used SGLang [64], a widely adopted open-source inference engine, and extended it with our elastic memory manager library. The library is built on CUDA VMM APIs [28] and exposes standard KV cache allocation APIs via Python bindings, requiring only 22 lines of changes to integrate with SGLang.

On the frontend, we used a Redis queue [38] to cache incoming requests from all clients. Prism's local scheduler dispatches these requests to the serving engines of corresponding models based on Algorithm 2. For tensor-parallel models, the GPU-local scheduler runs only on the first rank, and the resulting scheduling decisions are broadcast to all other ranks to ensure consistency. Prism's global scheduler operates as a separate Python process, collecting execution metrics from each engine—such as request rates and queue status. It makes scheduling decisions (*e.g.*, model evictions and activations) and communicates them to the engines using ZeroMQ [58].

	Mean E2E (s)	P95 E2E (s)	Req Tput (r/s)	Token Tput (t/s)
MuxServe	7.40	18.85	7.97	3363.53
MuxServe++	5.25	12.09	7.98	3353.09
	Mean TTFT (s)	P95 TTFT (s)	Mean TPOT (ms)	P95 TPOT (ms)
MuxServe	1.47	11.04	21.21	31.95
MuxServe++	0.089	0.320	18.82	33.97

Table 2: Performance comparison of MuxServe and MuxServe++

7.1 Experimental Setup

Testbed. We conducted our experiments on a cluster of four nodes, each equipped with eight NVIDIA H100-80G GPUs interconnected via 600GB/s NVLink. These nodes communicate through a 100Gbps Ethernet network. Each node features two 52-core Intel Xeon Platinum 8480+ CPUs, 1.7 TB of DRAM, and a PCIe Gen5 x16 interface. All nodes run Ubuntu 22.04 with CUDA Toolkit 12.4.

Baselines. We compared Prism against four baselines that share GPUs to serve multi-LLMs: (1) *Static partition (S-Partition)*; (2) *MuxServe++*; (3) *QLM [33]*; and (4) ServerlessLLM [13]. Note that the original MuxServe is built on vLLM and supports only Llama-2 models. We ported it to SGLang and generalized its memory mechanism with our kvcached to support heterogeneous models, referred to as MuxServe++. We evaluated the performance of MuxServe++ and MuxServe using three Llama-3.1-8B models under different request rates over a 10-minute period: 199 requests/min, 262 requests/min, and 22 requests/min. All experiments were conducted under the same and consistent conditions. The results are shown in Table 2. As we can see, MuxServe++ achieves comparable or even better performance.

Note that general GPU-sharing systems target application-agnostic workloads and cannot be directly compared: they focus on single-GPU sharing rather than cluster-level GPU sharing, and are not designed to optimize LLM serving, which requires both TTFT and TPOT adherence (§8).

Traces and models. We used two real-world traces, Hyperbolic [19] and Arena-Chat [9]. For each trace, we sampled a representative set of models, including both popular and

Model size	1B-3B	4B-8B	9B-30B	31B-70B
#LLMs	43	8	3	4

Table 3: Models used in our evaluation.

long-tail models, ensuring their workload characteristics (*e.g.*, popularity distribution and idle patterns) align with the observations in §3. To simulate various scenarios, we scaled the traces by multiplying the number of requests by a factor of N, increasing the load while preserving the original traffic patterns—a common way used in prior work [11, 22]. In total, we evaluated 58 LLMs as detailed in Table 3. The large-scale experiments (§7.4) use all models, while other evaluations (§7.2—§7.3) select subsets tailored to specific goals.

Metrics. Our primary performance metrics are TTFT and TPOT attainment. To establish SLOs for each model, we first ran its workload on dedicated GPUs to measure its P95 TTFT and TPOT latencies. This process produced TTFT SLOs ranging from 0.04s to 0.13s and TPOT SLOs from 5.2ms to 50.9ms. We then scaled these base values by a factor to evaluate system performance under varying latency requirements, following an approach consistent with prior work [11, 22, 36]. We also reported aggregated throughput. To account for model idle periods, throughput considers the actual time (excluding idle time) spent serving them.

7.2 End-to-End Performance

We start with the end-to-end performance of Prism under varying request rates, SLO requirements, and GPU supplies.

SLO attainment vs. request rate. We first evaluated various inference loads by serving eight models on two shared GPUs. As shown in Figure 5 (first row), Prism consistently outperforms all baselines by maintaining a significantly higher TTFT SLO attainment. On the Hyperbolic trace, Prism supports up to 2.3× and 3.5× more requests than MuxServe++ and static partitioning, respectively, while still achieving 99% SLO attainment. On the Arena-Chat trace, it handles over 3× more requests than all baselines. The gains stem from KVPR-driven placement, which keeps per-GPU memory pressure low so active models retain headroom under bursty arrivals. MuxServe++'s TTFT SLO attainment drops quickly with higher request rates because it cannot evict idle models or relocate models across GPUs, leading to memory contention and degraded performance.

QLM time-shares GPUs by packing pending requests into groups and dispatching each group to a GPU under an SLO-aware stochastic policy, swapping models and preempting unfinished requests whenever consecutive groups target different models. The frequent swapping makes QLM's TTFT attainment worse than static partitioning.

ServerlessLLM serves models in a time-sharing manner for serverless workloads: inactive models are unloaded, and newly arriving requests wait until the scheduler reactivates the required model on the server with the fastest startup, selected based on checkpoint locality. Because the full cold-start process still dominates request latency, ServerlessLLM achieves the worst TTFT SLO attainment.

Prism maintains high TPOT attainment through its demand-aware scheduler, which balances workloads to reduce memory contention. Although the scheduler explicitly targets TTFT, TPOT benefits as a side effect: TPOT is degraded mainly by oversized batches and memory-induced preemptions, both of which subside once KVPR-balanced placement caps per-GPU memory contention. QLM achieves lower TPOT because it over-batches under high load, increasing per-iteration latency, while ServerlessLLM performs even worse by allowing unbounded batch sizes that further raise P99 TPOT. Both MuxServe++ and static partitioning experience severe memory contention at high request rates, causing frequent preemptions that substantially degrade TPOT.

SLO attainment vs. SLO requirements. The middle row of Figure 5 shows the attainment across different SLO targets. As SLO scales up, Prism quickly achieves 99% TTFT and TPOT attainment. In contrast, no baseline achieves 99% TTFT attainment even on the largest SLO scale in this experiment, and their attainment rates do not improve significantly as the scale increases. Among the baselines, MuxServe++ achieves the best TTFT performance, reaching 84.79% and 67.22% attainment on the largest SLO scale for the two traces. The gap to Prism is primarily due to their inflexibility to adapt sharing policies dynamically. The TPOT attainment of all systems increases rapidly as the SLO scale grows because TPOT is less sensitive to memory contention. We also observe that the Hyperbolic trace requires higher SLO scales due to more bursty and heavier request patterns.

SLO attainment vs. available GPUs. Finally, we evaluate performance when provisioning more GPUs. We selected 18 models from Table 3, representing a mix of popular and tail models with diverse load variability. To fully test the flexibility of our scheduling strategy, we included models of varying sizes from 1B to 8B, all of which fit within a single 80GB GPU. This setup enables a wide range of model-sharing combinations across GPUs. As shown in Figure 5 (last row), Prism achieves 99% TTFT and TPOT attainment using only four and five GPUs on the two traces, respectively, demonstrating its effectiveness in improving cost-efficiency while maintaining performance. In comparison, all baselines fail to reach 99% TTFT attainment even with eight GPUs, and only a few baselines achieve 99% TPOT attainment when seven or eight GPUs are provisioned.

7.3 Performance Analysis

Next, we provide a detailed performance breakdown to analyze the effectiveness of each design in Prism and how each contributes to its strong end-to-end performance.

Flexible cross-model memory sharing. We first evaluated the benefits of our flexible cross-model memory sharing by comparing with static partition using a simplified two-model trace extracted from Arena-Chat, shown in Figure 6 (first

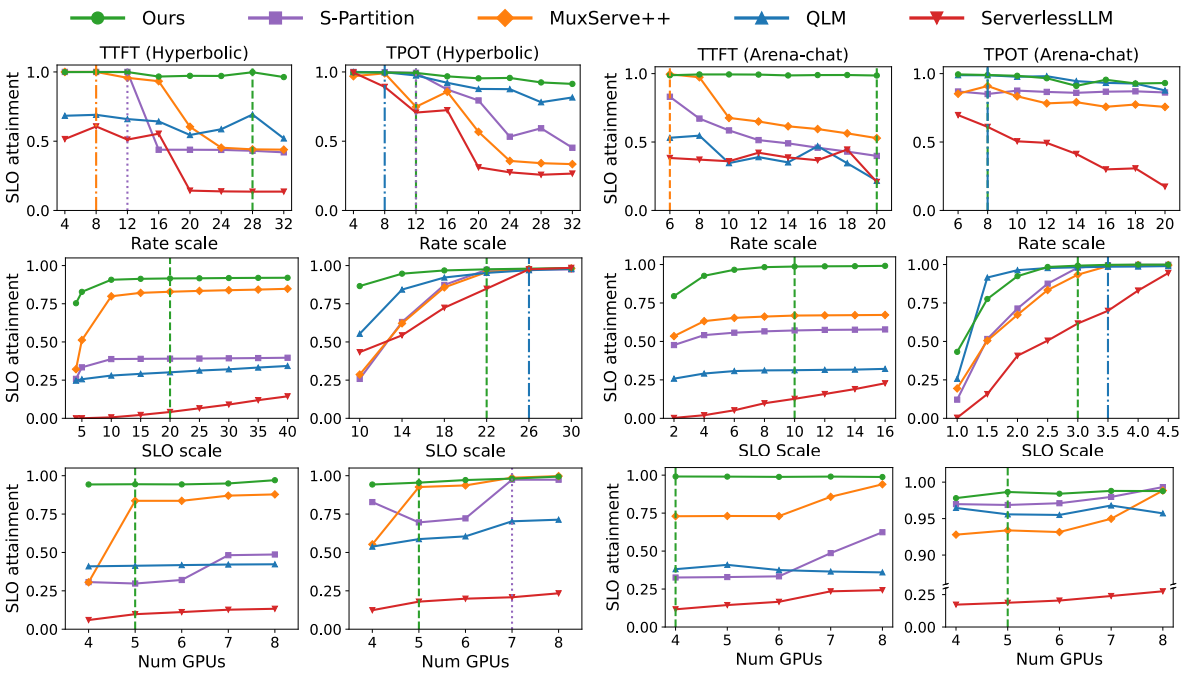


Figure 5: End-to-end performance comparison on SLO attainments under varying scales of rates, SLOs, and numbers of available GPUs. The dotted vertical lines mark where the system reaches 99% TTFT or TPOT attainment.

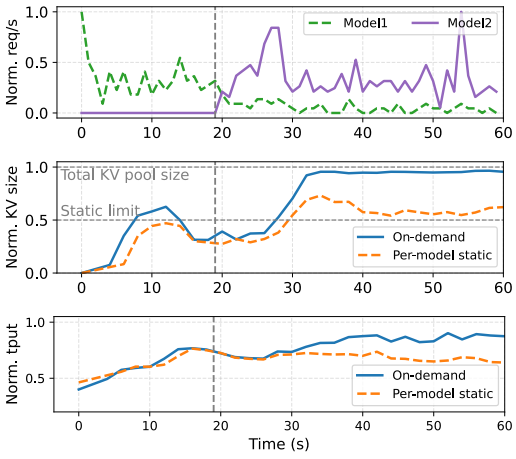
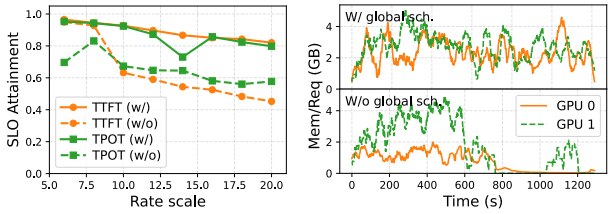


Figure 6: Benefits of cross-model memory coordination. The first figure shows the request rates. The last two figures shows the total KV memory size and the throughout of the two models.

row). We present the normalized total KV cache usage and aggregated throughput for both methods in the last two rows of Figure 6. As we can see, Prism's on-demand memory allocation allows it to use more memory for KV cache, particularly after the 20th second, when Model1 experiences low demand while Model2 faces a surge in request rates. The larger KV cache memory enables Prism to achieve higher throughput, as shown in the last row. In contrast, under static partitioning, even when Model1 underutilizes its memory, Model2 cannot leverage the unused memory due to the static allocation boundary.

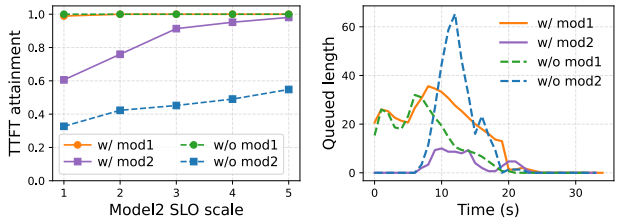


(a) Attainment with rate scales (b) GPU load status

Figure 7: Effectiveness of global model placement scheduling.

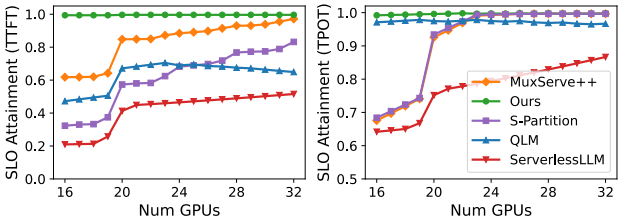
Model placement. Next, we evaluated the benefits of our global scheduler that conducts model placement. In this experiment, we used two GPUs to serve eight models from Arena-Chat. Figure 7a presents the TTFT and TPOT attainment with the global scheduler enabled or disabled. The results show that enabling the global scheduler significantly improves both TTFT and TPOT attainment. To provide further insights, we plot the average KV cache memory available per request as it arrives on each GPU. With the global scheduler enabled, the load is more evenly balanced across the two GPUs, allowing each request to access more KV cache memory on average. In contrast, without the global placement scheduler, the load is imbalanced: GPU1 shows more available memory during the first 600 seconds, while the near-zero availability between the 800th and 1000th seconds indicates that GPU1 is idle while GPU0 is overloaded.

Request arbitration. To evaluate the GPU-local scheduler that prioritizes requests, we use two models: we fix the SLO scale of Model1 to eight and vary the SLO scales of Model2

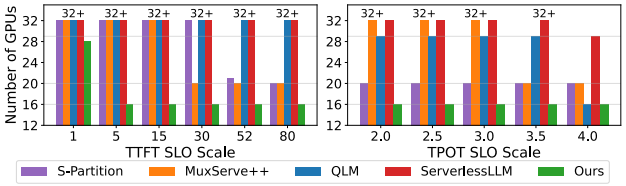


(a) Attainment with SLO scales (b) Queue length with time

Figure 8: Effectiveness of GPU local request scheduling.



(a) SLO attainment with cluster size scaling



(b) Number of GPUs needed for 99% SLO attainments

Figure 9: SLO attainment and cost saving at large scales.

to evaluate the priority-based admission control in the GPU-local scheduling. Figure 8a shows the TTFT attainment as we vary the SLO scale of Model2. Model1 consistently maintains high attainment, and enabling our GPU-local scheduling improves the SLO attainment of Model2 by more than 40%. To dive deeper, we plot the queue length of each model in Figure 8b of one experiment run. From the queue lengths, we clearly observe that when the local scheduler is enabled, the system prioritizes Model2's requests, which are shorter but have stricter SLO requirements.

7.4 Large-Scale Evaluation

Finally, we evaluated how effectively Prism in reducing the cost of multi-LLM serving at scale. We served all 58 models listed in Table 3, following common TP practices for large models: TP=4 for 32B models [46,47] and TP=4 or 8 for 70B models [2, 24], utilizing 32 GPUs in total. We sampled 58 models from the Arena-Chat trace and the Hyperbolic trace.

SLO attainments vs. number of GPUs. Figure 9a shows TTFT and TPOT attainment with an increased number of GPUs. Prism outperforms all baselines, achieving nearly 99% TTFT attainment with just 16 GPUs, while MuxServe++ requires 32 GPUs to reach similar performance, and other methods require even more. As the number of GPUs increases, all baselines improve in TTFT and TPOT attainment

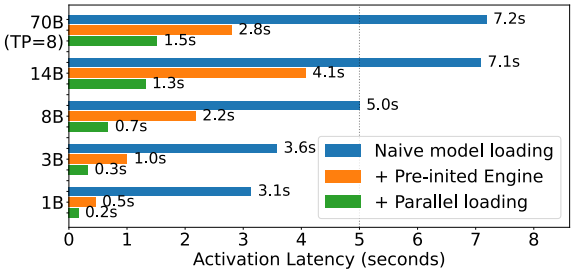
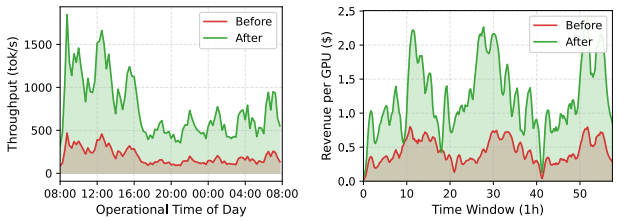


Figure 10: The activation time for models with different sizes. Data is measured on H100 GPUs.



(a) Company A (b) Company B

Figure 11: Production results from two companies, showing (a) throughput and (b) revenue per GPU before and after using Prism.

except QLM. We find this to be related to its suboptimal scheduling algorithm. QLM assigns requests to GPUs without considering which models are already on GPUs; when a queue is empty, it picks the first available GPU, often triggering unnecessary swaps. With more GPUs, this leads to more idle devices and frequent swapping. Its inefficient swapping, which requires engine restart [37], adds significant latency, causing queued requests to miss their SLOs.

Cost saving. Figure 9b shows the number of GPUs required to achieve 99% SLO attainments at different SLO scales. If a system fails to achieve 99% attainment with all 32 GPUs, we denote its GPU requirement as "32+". Prism achieves 99% TTFT SLO attainments with only 16 GPUs when SLO scale is 5 and TPOT SLO scale is 2.0. MuxServe++ needs 20 GPUs to get 99% TTFT SLO attainments with SLO scale $\geq$ 30, while static partition needs even more GPUs or a higher SLO scale. For TPOT, static partition is the best across all baselines, requiring 20 GPUs, while QLM and MuxServe++ need at least 29 GPUs when TPOT SLO scale $\leq$ 3.0.

7.5 System Overhead

Model activation latency. We also measured model activation latency from pageable CPU memory for models ranging from 1B to 70B parameters (see §5.3 for more details). Our optimizations significantly reduce activation latency. Prism loads small models (1B~8B) within 0.7s, a medium-size model (14B) in just 1.3s, and large models (>70B) in 1.5s (Figure 10). These results show that Prism can promptly activate evicted models upon receiving new requests.

Activation and migration frequency. Over a 10-minute window of the eight-models-on-two-GPUs run, Prism issued

2 idle-driven activations and 3 inter-GPU migrations. Migrations stay off the TTFT critical path: weights and in-flight KV state transfer over NVLink within tens of milliseconds (*e.g.*, ∼20 ms for an 8B model) while the source instance keeps serving. Activations complete within 2 s, well inside the multi-second TTFT SLOs typical of online serving.

Elastic memory overhead. We evaluated Prism's elastic memory overhead in the worst case, where the request rates are constant, leaving no opportunities for dynamic sharing. With two colocated Llama-3.2-3B models on an A100-40G GPU, Prism incurs only 3 ms (4 %) TTFT and 4 ms (13 %) TPOT overhead at a high load of 32 req/s and only 2 ms (3.5 %) TTFT and 3 ms (7 %) TPOT overhead at a high load of 28 req/s (see §A.3), compared to static partitioning.

7.6 Production Workloads

As of December 2025, Prism has been deployed in multiple organizations serving LLM workloads. Figure 11 summarizes deployments at a large tech company (Company A) using Prism on an internal GPU cluster, and a commercial LLM inference provider (Company B) serving external customers. To isolate Prism's contribution from workload variation, both deployments use shadow workload replay: the same online request stream is mirrored to two identical clusters, one running Prism and the other not, so that the model mix and traffic patterns are held constant across the two arms.

Figure 11a shows Company A's per-GPU token throughput before and after adopting Prism. Company A serves diverse applications using fine-tuned and off-the-shelf models (3B–70B) on a shared GPU pool. With Prism dynamically adjusting model placement and sharing—without changes to existing inference engines—Company A achieves consistently higher throughput (3.89× on average) over several weeks, with no SLO violations and unchanged tail latency. Figure 11b presents results from Company B, which optimizes revenue per GPU under highly variable traffic, computed as the prefill and decode tokens generated in the window priced at the provider's published per-token rates, normalized by GPU count. Prior to Prism, fragmentation and overprovisioning constrained revenue; after deployment, Prism improves utilization while meeting all SLOs, raising revenue per GPU by 2.86× by converting idle capacity into billable tokens.

8 Related Work

SLO-aware LLM scheduling. Llumnix [45], SLOs-Serve [6], ExeGPT [32], SAGESERVE [20], DistServe [65], and MELL [35] explore SLO-aware scheduling to improve LLM inference performance. However, they primarily focus on request scheduling or resource allocation for single-model serving, while Prism enables dynamic cross-model memory harvesting for efficient multi-model serving.

Memory management in LLM serving. vAttention [34] and vTensor [56] use CUDA virtual memory APIs [28] to decouple physical and virtual memory allocation. However, their purpose is to simplify programming and improve kernel efficiency for single-LLM serving. In addition, they require re-implementing a large portion of the current inference engine stack, while our method preserves compatibility with the widely used PagedAttention [21] mechanism.

GPU sharing techniques. GPU sharing has been extensively studied [5, 7, 11, 13, 16–18, 22, 26, 33, 44, 49, 51–53, 55, 57, 60, 61, 63], but largely at the OS or runtime layer for application-agnostic, single-GPU workloads. Compute-centric designs time-slice between latency-critical and best-effort co-tenants [12, 52] or add preemptive scheduling across heterogeneous accelerators [42], whereas Prism targets memory sharing across uniformly latency-critical LLMs spanning multiple GPUs. The closest memory-centric work, MSched [41], schedules GPU–CPU paging *after* oversubscription, while Prism proactively prevents oversubscription via coordinated placement. Existing multi-LLM solutions remain fragmented: MuxServe [11] supports spatial sharing but pins models to devices even when idle, while Aegaeon [53] performs temporal sharing via prefill–decode disaggregation, requiring weight duplication and struggling with diverse, dynamic access patterns. Prism unifies spatial and temporal sharing through elastic GPU memory management.

9 Conclusion

This paper introduces Prism, a multi-LLM serving system that improves cost efficiency while maximizing SLO attainment via GPU sharing. Prism achieves this by enabling flexible memory ballooning, and employing a two-level scheduling algorithm to make efficient use of GPU memory.

Acknowledgments

We thank the anonymous reviewers for their comments and are particularly grateful to our shepherd Xiaosong Ma for her valuable feedback. We also thank Ion Stoica, Matei Zaharia, Jiangfei Duan, Chenxi Wang, Shi Liu, Zhenting Zhu, and Yicheng Liu for their discussions and insights. We are grateful to Hyperbolic, Novita AI, Chatbot Arena, NVIDIA, and the SGLang open-source community for production traces and hardware support. This work was supported by an Amazon AI Fellowship (awarded to Shan Yu), the National Science Foundation (CNS-1763172, CNS-2007737, CNS-2006437, CNS-2106838, CNS-2147909, CNS-2128653, CNS-2301343, CNS-2330831, CNS-2403254, IIS-2546642), the Alibaba Research Intern Program, and gifts to Sky Computing Lab from industry partners (Accenture, AMD, Anyscale, Cisco, Google, IBM, Intel, Intesa Sanpaolo, Lambda, Lightspeed, Mibura, Microsoft, NVIDIA, Samsung SDS, and SAP). Junyi Shu, Jiarong Xing, Harry Xu, and Ying Sheng are the corresponding authors.

References

[1] N. Agarwal. Implementing Fractional GPUs in Kubernetes with Aliyun Scheduler. https://huggingface.co/blog/NileshInfer/implementing-fractional-gpus-in-kubernetes, 2024. Accessed: May, 2025.

[2] M. AI. Llama 3.3-70B-Instruct. https://huggingface.co/meta-llama/Llama-3.3-70B-Instruct, 2024. Accessed: May, 2025.

[3] N. AI. Novita AI homepage. https://novita.ai/, 2025. Accessed: May, 2025.

[4] ai-dynamo. dynamo: A datacenter-scale distributed inference serving framework. https://github.com/ai-dynamo/dynamo, 2025. Accessed: 2025-08-18.

[5] G. Chen, Y. Zhao, X. Shen, and H. Zhou. Effisha: A software framework for enabling effficient preemptive scheduling of gpu. In *Proceedings of the 22nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '17, page 3–16, New York, NY, USA, 2017. Association for Computing Machinery.

[6] S. Chen, Z. Jia, S. Khan, A. Krishnamurthy, and P. B. Gibbons. SLOs-Serve: Optimized Serving of Multi-SLO LLMs. *arXiv preprint arXiv:2504.08784*, 2025.

[7] W. Chen, Z. Mo, H. Xu, K. Ye, and C. Xu. Interference-aware multiplexing for deep learning in gpu clusters: A middleware approach. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '23, New York, NY, USA, 2023. Association for Computing Machinery.

[8] J. Cheriyan, R. Ravi, and M. Skutella. A Simple Proof of the Moore-Hodgson Algorithm for Minimizing the Number of Late Jobs. *Operations Research Letters*, 49(6):842–843, 2021.

[9] W.-L. Chiang, L. Zheng, Y. Sheng, A. N. Angelopoulos, T. Li, D. Li, H. Zhang, B. Zhu, M. Jordan, J. E. Gonzalez, and I. Stoica. Chatbot Arena: An Open Platform for Evaluating LLMs by Human Preference, 2024.

[10] P. H. Coppock, B. Zhang, E. H. Solomon, V. Kypriotis, L. Yang, B. Sharma, D. Schatzberg, T. C. Mowry, and D. Skarlatos. Lithos: An operating system for efficient machine learning on gpus. In *Proceedings of the ACM SIGOPS 31st Symposium on Operating Systems Principles*, SOSP '25, page 1–17, New York, NY, USA, 2025. Association for Computing Machinery.

[11] J. Duan, R. Lu, H. Duanmu, X. Li, X. Zhang, D. Lin, I. Stoica, and H. Zhang. MuxServe: Flexible Multiplexing for Efficient Multiple LLM Serving. *arXiv preprint arXiv:2404.02015*, 2024.

[12] R. Fan, T. Ren, M. Xie, S. Gao, J. Shu, and Y. Lu. Gpreempt: Gpu preemptive scheduling made general and efficient. In *Proceedings of the 2025 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC '25, USA, 2025. USENIX Association.

[13] Y. Fu, L. Xue, Y. Huang, A.-O. Brabete, D. Ustiugov, Y. Patel, and L. Mai. ServerlessLLM:Low-Latency Serverless Inference for Large Language Models. In *USENIX OSDI*, pages 135–153, 2024.

[14] S. Gao, Q. Wang, S. Zeng, Y. Lu, and J. Shu. Weaver: efficient multi-llm serving with attention offloading. In *ATC*, USENIX ATC '25, USA, 2025. USENIX Association.

[15] R. L. Graham. Bounds on Multiprocessing Timing Anomalies. *SIAM Journal on Applied Mathematics*, 17(2):416–429, 1969.

[16] L. Han, Z. Zhou, and Z. Li. Pantheon: Preemptible Multi-DNN Inference on Mobile Edge GPUs. In *Proceedings of the 22nd Annual International Conference on Mobile Systems, Applications and Services*, pages 465–478, 2024.

[17] M. Han, H. Zhang, R. Chen, and H. Chen. Microsecond-Scale Preemption for Concurrent GPU-Accelerated DNN Inferences. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 539–558, 2022.

[18] M. Han, H. Zhang, R. Chen, and H. Chen. Microsecond-scale preemption for concurrent GPU-accelerated DNN inferences. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 539–558, Carlsbad, CA, July 2022. USENIX Association.

[19] Hyperbolic. Hyperbolic: The Open Access AI Cloud. Hyperbolic Provides Affordable GPU Access and Inference Services for Those at the Edges of AI. https://hyperbolic.xyz/, 2024.

[20] S. Jaiswal, K. Jain, Y. Simmhan, A. Parayil, A. Mallick, R. Wang, R. S. Amant, C. Bansal, V. Rühle, A. Kulkarni, et al. Serving Models, Fast and Slow: Optimizing Heterogeneous LLM Inferencing Workloads at Scale. *arXiv preprint arXiv:2502.14617*, 2025.

[21] W. Kwon, Z. Li, S. Zhuang, Y. Sheng, L. Zheng, C. H. Yu, J. Gonzalez, H. Zhang, and I. Stoica. Efficient Memory Management for Large Language Model Serving with PagedAttention. In J. Flinn, M. I. Seltzer, P. Druschel, A. Kaufmann, and J. Mace, editors, *Proceedings of the 29th Symposium on Operating Systems Principles, SOSP 2023*, pages 611–626. ACM, 2023.

[22] Z. Li, L. Zheng, Y. Zhong, V. Liu, Y. Sheng, X. Jin, Y. Huang, Z. Chen, H. Zhang, J. E. Gonzalez, and I. Stoica. AlpaServe: Statistical Multiplexing with Model Parallelism for Deep Learning Serving. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*, pages 663–679, Boston, MA, July 2023. USENIX Association.

[23] llm-d. llm-d: A kubernetes-native high-performance distributed llm inference framework. https://github.com/llm-d, 2025. Accessed: 2025-08-18.

[24] Meta. Introducing Llama 3.1: Our Most Capable Models to Date. https://ai.meta.com/blog/meta-llama-3-1/, 2024.

[25] J. M. Moore. An N Job, One Machine Sequencing Algorithm for Minimizing the Number of Late Jobs. *Management Science*, 15(1):102–109, 1968.

[26] K. K. Ng, H. M. Demoulin, and V. Liu. Paella: Low-Latency Model Serving with Software-Defined GPU Scheduling. In *Proceedings of the 29th Symposium on Operating Systems Principles*, pages 595–610, 2023.

[27] NVIDIA. NVIDIA Multi-Process Service. https://docs.nvidia.com/deploy/mps/index.html, 2024. Accessed: August, 2025.

[28] NVIDIA. CUDA Toolkit Documentation: Virtual Memory Management. https://docs.nvidia.com/cuda/cuda-driver-api/group__CUDA__VA.html, 2025. Accessed: May, 2025.

[29] NVIDIA. NVIDIA Multi-Instance GPU. https://www.nvidia.com/en-us/technologies/multi-instance-gpu/, 2025. Accessed: May, 2025.

[30] NVIDIA. Unlock Next Level Performance with Virtual GPUs. https://www.nvidia.com/en-us/data-center/virtual-solutions/, 2025. Accessed: August, 2025.

[31] NVIDIA. NVIDIA GPUDirect RDMA. https://network.nvidia.com/products/GPUDirect-RDMA/, 2026.

[32] H. Oh, K. Kim, J. Kim, S. Kim, J. Lee, D.-s. Chang, and J. Seo. Exegpt: Constraint-Aware Resource Scheduling for LLM Inference. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, pages 369–384, 2024.

[33] A. Patke, D. Reddy, S. Jha, H. Qiu, C. Pinto, C. Narayanaswami, Z. Kalbarczyk, and R. Iyer. Queue Management for SLO-Oriented Large Language Model Serving. In *Proceedings of the 2024 ACM Symposium on Cloud Computing*, SoCC '24, page 18–35, New York, NY, USA, 2024. Association for Computing Machinery.

[34] R. Prabhu, A. Nayak, J. Mohan, R. Ramjee, and A. Panwar. vAttention: Dynamic Memory Management for Serving LLMs without PagedAttention, 2024.

[35] L. Qianli, H. Zicong, C. Fahao, L. Peng, and G. Song. Mell: Memory-Efficient Large Language Model Serving via Multi-GPU KV Cache Management. *arXiv preprint arXiv:2501.06709*, 2025.

[36] R. Qin, Z. Li, W. He, J. Cui, F. Ren, M. Zhang, Y. Wu, W. Zheng, and X. Xu. Mooncake: Trading More Storage for Less Computation—A KVCache-Centric Architecture for Serving LLM Chatbot. In *23rd USENIX Conference on File and Storage Technologies (FAST 25)*, pages 155–170, 2025.

[37] QLM Project. QLM: Quantum Language Model Project. https://github.com/QLM-project/QLM/blob/eea5b622e2c4c6abd705876880f50014c4d9d0d1/qlm/endpoints/endpoint.py#L57, 2024. Accessed: May, 2025.

[38] Redis. Redis List documentation. https://redis.io/docs/latest/develop/data-types/lists/, 2025. Accessed: May, 2025.

[39] RyokoAI. Sharegpt dataset. https://sharegpt.com, 2024.

[40] sgl project. Ome: A kubernetes operator for enterprise-grade management and serving of llms, 2025.

[41] W. Shen, Y. Chen, R. Chen, and H. Chen. Msched: Gpu multitasking via proactive memory scheduling. *arXiv preprint arXiv:2512.24637*, 2025.

[42] W. Shen, M. Han, J. Liu, R. Chen, and H. Chen. Xsched: preemptive scheduling for diverse xpus. In *Proceedings of the 19th USENIX Conference on Operating Systems Design and Implementation*, OSDI '25, USA, 2025. USENIX Association.

[43] Y. Sheng, S. Cao, D. Li, C. Hooper, N. Lee, S. Yang, C. Chou, B. Zhu, L. Zheng, K. Keutzer, J. E. Gonzalez, and I. Stoica. SLoRA: Scalable Serving of Thousands of LoRA Adapters. In *Proceedings of Machine Learning and Systems*, volume 6, pages 296–311, 2024.

[44] F. Strati, X. Ma, and A. Klimovic. Orion: Interference-Aware, Fine-Grained GPU Sharing for ML Applications. In *Proceedings of the Nineteenth European Conference on Computer Systems*, pages 1075–1092, 2024.

[45] B. Sun, Z. Huang, H. Zhao, W. Xiao, X. Zhang, Y. Li, and W. Lin. Llumnix: Dynamic Scheduling for Large

Language Model Serving. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, pages 173–191, 2024.

[46] Q. Team. Qwen2.5-32b. https://huggingface.co/Qwen/Qwen2.5-32B, 2024. Accessed: May, 2025.

[47] Q. Team. Qwen2.5: A Party of Foundation Models. https://qwenlm.github.io/blog/qwen2.5/, 2024. Accessed: May, 2025.

[48] Y. Wang, Y. Chen, Z. Li, Z. Tang, R. Guo, X. Wang, Q. Wang, A. C. Zhou, and X. Chu. Towards efficient and reliable llm serving: A real-world workload study, 2024.

[49] Y. Wang, C. Liu, D. Wong, and H. Kim. Gcaps: Gpu context-aware preemptive priority-based scheduling for real-time tasks, 2024.

[50] Wang, Jiahao and Han, Jinbo and Wei, Xingda and Shen, Sijie and Zhang, Dingyan and Fang, Chenguang and Chen, Rong and Yu, Wenyuan and Chen, Haibo. Kvcache cache in the wild: Characterizing and optimizing kvcache cache at a large cloud provider. In *2025 USENIX Annual Technical Conference (USENIX ATC 25)*, pages 465–482. USENIX Association, July 2025.

[51] Q. Weng, L. Yang, Y. Yu, W. Wang, X. Tang, G. Yang, and L. Zhang. Beware of fragmentation: Scheduling GPU-Sharing workloads with fragmentation gradient descent. In *2023 USENIX Annual Technical Conference (USENIX ATC 23)*, pages 995–1008, Boston, MA, July 2023. USENIX Association.

[52] B. Wu, Z. Zhang, Z. Bai, X. Liu, and X. Jin. Transparent GPU sharing in container clouds for deep learning workloads. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 69–85, Boston, MA, April 2023. USENIX Association.

[53] Y. Xiang, X. Li, K. Qian, Y. Yang, D. Zhu, W. Yu, E. Zhai, X. Liu, X. Jin, and J. Zhou. Aegaeon: Effective gpu pooling for concurrent llm serving on the market. In *SOSP*, SOSP '25, page 1030–1045, New York, NY, USA, 2025. Association for Computing Machinery.

[54] Y. Xiang, X. Li, K. Qian, W. Yu, E. Zhai, and X. Jin. ServeGen: Workload Characterization and Generation of Large Language Model Serving in Production. *arXiv preprint arXiv:2505.09999*, 2025.

[55] W. Xiao, S. Ren, Y. Li, Y. Zhang, P. Hou, Z. Li, Y. Feng, W. Lin, and Y. Jia. AntMan: Dynamic scaling on GPU clusters for deep learning. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 533–548. USENIX Association, November 2020.

[56] J. Xu, R. Zhang, C. Guo, W. Hu, Z. Liu, F. Wu, Y. Feng, S. Sun, C. Shao, Y. Guo, et al. vTensor: Flexible Virtual Tensor Management for Efficient LLM Serving. *arXiv preprint arXiv:2407.15309*, 2024.

[57] P. Yu and M. Chowdhury. Salus: Fine-grained gpu sharing primitives for deep learning applications, 2019.

[58] ZeroMQ. ZeroMQ Website. https://zeromq.org/, 2025. Accessed: May, 2025.

[59] D. Zhang, H. Wang, Y. Liu, X. Wei, Y. Shan, R. Chen, and H. Chen. Fast and Live Model Auto Scaling with O (1) Host Caching. *arXiv preprint arXiv:2412.17246*, 2024.

[60] S. Zhang, Q. Chen, W. Cui, H. Zhao, C. Xue, Z. Zheng, W. Lin, and M. Guo. Improving GPU Sharing Performance through Adaptive Bubbleless Spatial-Temporal Sharing. In *Proceedings of the Twentieth European Conference on Computer Systems*, pages 573–588, 2025.

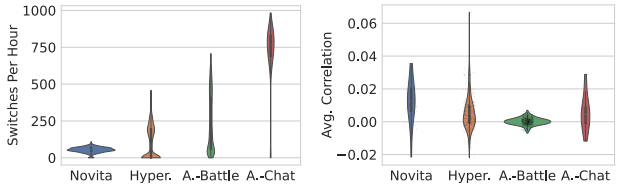
[61] Y. Zhang, H. Yu, C. Han, C. Wang, B. Lu, Y. Li, Z. Jiang, Y. Li, X. Chu, and H. Li. SGDRC: Software-Defined Dynamic Resource Control for Concurrent DNN Inference on NVIDIA GPUs. In *Proceedings of the 30th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming*, pages 267–281, 2025.

[62] W. Zhao, A. Jayarajan, and G. Pekhimenko. Tally: Non-intrusive performance isolation for concurrent deep learning workloads. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1*, ASPLOS '25, page 1052–1068, New York, NY, USA, 2025. Association for Computing Machinery.

[63] Y. Zhao, X. Liu, S. Liu, X. Li, Y. Zhu, G. Huang, X. Liu, and X. Jin. Muxflow: Efficient and safe gpu sharing in large-scale production deep learning clusters, 2023.

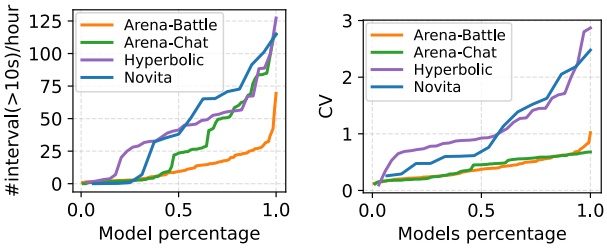
[64] L. Zheng, L. Yin, Z. Xie, J. Huang, C. Sun, C. H. Yu, S. Cao, C. Kozyrakis, I. Stoica, J. E. Gonzalez, C. Barrett, and Y. Sheng. SGLang: Efficient Execution of Structured Language Model Programs, 2023.

[65] Y. Zhong, S. Liu, J. Chen, J. Hu, Y. Zhu, X. Liu, X. Jin, and H. Zhang. DistServe: Disaggregating prefill and decoding for goodput-optimized large language model serving. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, pages 193–210, 2024.



(a) Bursty-group shifts. (b) Request rate correlation.

Figure 12: Request patten shifts.



(a) Number of idle intervals per hour. (b) CV of request rate per minute.

Figure 13: Volatile requests.

A Appendix

A.1 Analysis of All production traces

Overview. To understand the workload of multi-LLM serving, we analyze four production traces in detail. These traces are collected from representative service providers as summarized in Table 1. The first two traces are from Hyperbolic [19] and Novita AI [3], two popular LLM inference service providers. They offer inference APIs for a variety of foundation models and also support user-deployed, fine-tuned models. The last two traces are from Chatbot Arena [9], a widely used open-source platform for LLM evaluation. It compares model responses via human preference voting (Arena-Battle) and also provides interfaces for real-time conversations with various models (Arena-Chat).

Bursty-group behaviors. The bursty groups in the production traces shift rapidly over time, and this behavior can be quantified through model switches. As shown in Figure 12a, we plot the number of model switches per hour for each trace. To compute this metric, we treat a model as active if it has received at least one request in the past two minutes. A model switch is counted whenever the set of active models changes. Across the four traces, Novita exhibits the fewest switches, yet still averages 54 switches per hour, meaning the active set changes almost once per minute. In contrast, Arena-Chat, which contains the largest and most diverse collection of models, experiences an average of 766 switches per hour, reflecting that its active set changes every few seconds.

Unpredictability. The request rate at a given time is also highly unpredictable in our traces. Figure 12b quantifies predictability by measuring, for each model, the Pearson correlation between its request-rate time series on consecutive days. Across all traces, the correlations cluster near zero, showing that a model's traffic at a given time of day provides virtually no information about its traffic at the same time on the next day.

Volatile requests. Figure 13a further shows that models frequently alternate between activity and idleness: many models in the Hyperbolic and Novita traces experience more than 40–100 idle intervals per hour (>10 s each), making static reservations wasteful. Complementing this, Figure 13b reports the coefficient of variation (CV), calculated as the standard deviation divided by the mean (σ/μ), of requests per minute for each model. Both Hyperbolic and Novita traces contain many models with CV > 1. The Chatbot Arena traces have lower request rates, resulting in smaller CVs overall, but many models still exhibit CVs greater than 0.5. Together, the high frequency of idle intervals and the substantial variability captured by CV indicate that all traces exhibit strong, persistent volatility.

A.2 Analysis of Algorithm 1

A.2.1 KVPR Bound Analysis

The global model placement algorithm ensures that the maximum KV pressure ratio (KVPR) across all GPUs is bounded by the maximum KVPR in the optimal placement. We give the following analysis.

Let $KVPR_{OPT}$ be the minimum possible maximum KVPR achievable by any optimal placement. Let $KVPR_{max}$ be the maximum KVPR produced by Algorithm 1. We want to show $KVPR_{max}$ is bounded by $KVPR_{OPT}$.

Bottleneck Analysis: Focus on the GPU, denoted as g_{max}, that achieves the highest KVPR ($KVPR_{max}$) given by Algorithm 1's placement. Let m_k be the *last* model assigned to this GPU g_{max}. Let W_{before} and S_{before} represent the total SLO-weighted request rate and shared KV memory on g_{max} *just before* model m_k was assigned. The final state on this GPU is $KVPR_{max} = (W_{before} + d_k)/(S_{before} - w_k)$, where d_k is the SLO-weighted request rate (r_k/s_k) and w_k is the memory weight of model m_k. Similar to Graham [15], this proof aims to demonstrate that both the state *before* m_k was added and the contribution *of* m_k are bounded relative to $KVPR_{OPT}$. Specifically, it seeks to establish two conceptual bounds:

- **Bound 1 (Related to state before m_k):** The KVPR on g_{max} just before m_k's assignment, $\frac{W_{before}}{S_{before}}$, was the minimum among all GPUs at that moment due to the algorithm's greedy choice. This minimum KVPR is typically related to the average "pressure" across the system, which, in turn, is argued to be no larger than the optimal maximum pressure. This suggests the inequality: $W_{before}/S_{before} \leq KVPR_{OPT}$
- **Bound 2 (Related to model m_k):** The "pressure" exerted by the critical model m_k must be handled by the optimal

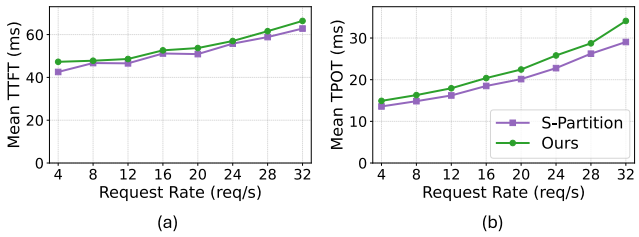


Figure 14: Mean latency comparison.

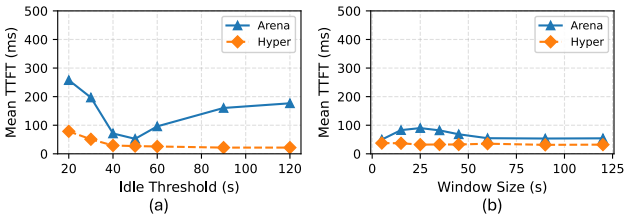


Figure 15: Sensitivity to hyperparameters.

solution. A fundamental lower bound on the optimal solution is the maximum pressure any single model would exert if placed alone on an otherwise empty GPU, *i.e.*, $KVPR_{OPT} \geq d_k/(C - w_k)$.

The final step involves integrating these insights to bound $KVPR_{max} = \frac{W_{before}+d_k}{S_{before}-w_k}$. Following Graham's proof [15], we substitute these into the numerator and get $KVPR_{max} \leq KVPR_{OPT} \cdot (1 + \frac{C}{S_{gmax}-w_k})$.

A.2.2 TP Support

The model placement algorithm in Algorithm 1 seamlessly integrates models utilizing Tensor Parallelism (TP). We conceptualize a TP model requiring tp_size GPUs as being composed of tp_size distinct parts. For scheduling purposes, we create tp_size entries in the sorted model list for such a model, assigning each entry $\frac{1}{tp_size}$ of the original weight and request rate. A beneficial property emerges from this decomposition: since these entries have identical $\frac{r_k}{s_k}$ values, they remain adjacent after sorting. This adjacency increases the likelihood that, as the algorithm iterates, these parts are initially assigned to different GPUs due to rising KVPRs. To ensure the distribution, if assigning a TP part to the GPU with the minimum KVPR would result in collocating it with another part of the same original model, we instead assign it to the GPU exhibiting the second-lowest KVPR. Through this decomposition strategy and modified assignment rule, our algorithm effectively considers and manages the placement of TP models alongside single-GPU models.

A.3 Elastic Memory Overhead

To stress-test the worst case, we evaluate the system under a constant request rate without idle periods, eliminating opportunities for memory coordination through either time- or space-sharing. In this setting, static partitioning provides the strongest baseline, as it divides memory according to the steady request rate. Figure 14 presents a comparison between our system and static partitioning when serving two Llama-3.2-3B models on a single A100-40G GPU. In this experiment, the global scheduler does not alter placements and the GPU-local scheduler does not reorder requests, since all requests have equal priority. The only additional cost arises from the elastic memory manager, which dynamically maps and unmaps pages, unlike static partitioning that relies on pre-allocation. With optimizations such as buffer preallocation and contiguous layouts, this overhead remains modest: mean TTFT and TPOT increase by only 3–5% as request rates scale, demonstrating the efficiency of our system even under worst-case conditions.

A.4 Sensitivity Analysis

We analyze the sensitivity of Prism to two key hyperparameters: the model eviction idle threshold and the load monitoring window size with Hyperbolic and Chatbot Arena traces. The results are shown in Figure 15.

Idle Eviction Threshold. Figure 15(a) reports the mean TTFT as we vary the idle threshold, which dictates how long a model must remain inactive before being evicted. The results reveal a clear convex trade-off. When the threshold is too short (*e.g.*, < 40 s), the system becomes overly aggressive, evicting models during short inter-arrival gaps. This leads to instability and thrashing, where the system must frequently pay the high penalty of model reactivation for subsequent requests, thereby inflating TTFT. Conversely, when the threshold is too long (*e.g.*, $>$ 80s), idle models hoard GPU memory that could otherwise be harvested. This resource locking prevents the scheduler from placing new active models or expanding the KV cache for concurrent requests, resulting in resource starvation and increased queuing delays. Empirically, a threshold of approximately 45 seconds achieves the optimal balance between minimizing reactivation overhead and maximizing resource reclamation.

Load Monitoring Window Size. Figure 15(b) examines the impact of the sliding window size used to calculate the moving average of token rates for the KVPR. This parameter controls the sensitivity of the placement algorithm: a small window makes the scheduler sensitive to transient bursts, while a large window focuses on long-term trends. The results demonstrate that Prism is generally robust to variations in window size. We observe that a window size of approximately 60 seconds provides a stable estimation of memory pressure, effectively smoothing out short-term noise while remaining responsive enough to shift resources during sustained workload changes.

ARTIFACT EVALUATED
USENIX
AVAILABLE

Break on Through to the Other Side: Pooling Memory Elastically with RAMRYDER

Yanbo Zhou[†], Erci Xu[‡*], Dongjoo Seo[§], Adam Manzanares[§], Steven Swanson[†]
[†]*UC San Diego* [‡]*Shanghai Jiao Tong University* [§]*Samsung Semiconductor*

Abstract

Cloud vendors offer diverse infrastructure services by flexibly allocating resources, such as compute, storage, and networking, across virtual machines. However, memory allocation is still less flexible: the vendors allocate memory capacity in a fixed ratio to virtual CPUs and provide no mechanism for allocating memory bandwidth. Recent data show that this can lead to underutilization of both capacity and bandwidth when an application's demands diverge or change over time. To address these challenges, we propose RAMRYDER, a software-defined elastic memory system for cloud virtual machines that allows the system to allocate memory bandwidth and capacity (mostly) independently. RAMRYDER controls the mapping between memory pages in the guest OS and the underlying memory channels, providing performance isolation between virtual machines and allowing dynamic changes to bandwidth and capacity allocation. RAMRYDER improves average capacity and bandwidth utilization by 28.6% and 43.2%, respectively, across the cluster while delivering performance comparable to the best case with exclusive access.

1 Introduction

Cloud service providers provide low-cost and flexible compute resources by supporting the flexible allocation of resources across virtual machines (VMs). Although existing virtualization mechanisms allow for flexible allocation of CPU [7, 9, 81], disk [13, 30, 31], and networking [14, 29] resources, cloud vendors often provide CPU and memory resources in fixed ratios. This can lead to underutilized memory resources in the cloud. Indeed, recent reports show that up to 45% of memory capacity is untouched for half of the VMs in Azure's data center [70].

Previous studies have addressed this problem by allowing more flexible allocation of memory capacity [70] or enabling memory capacity harvesting [40, 70, 85], but large cluster traces show that memory bandwidth is even more severely underutilized, leaving substantial bandwidth stranded in cloud data centers. We find that 90% of servers exhibit average bandwidth utilization below 44.5%. Even peak bandwidth utilization stays below 82.2% for 90% of servers.

The underlying causes of these problems are *spatial over-provisioning* and *temporal over-provisioning*. First, spatial over-provisioning arises because users with strict performance requirements often want exclusive hardware access to avoid bandwidth contention and meet performance quality of service (QoS) targets. A typical practice in multi-tenant cloud environments is to subscribe to at least half of a physical server (i.e., one CPU socket), which guarantees exclusive access to hardware resources [22] and thus avoids performance interference from co-located VMs. Second, temporal over-provisioning arises because users appear to over-provision memory capacity, but in practice they are reserving memory bandwidth to sustain the applications' peak loads as the memory resource is allocated at VM creation and remains fixed throughout the VM lifespan.

Furthermore, the underutilized bandwidth is not necessarily correlated with underutilized capacity. Rather, they exhibit different spatial and temporal patterns. Ideally, cloud providers could allocate memory capacity and bandwidth independently to match the varying demands of co-located VMs, thereby improving overall memory utilization. However, existing approaches, including memory harvesting [40, 85, 104], memory disaggregation via RDMA [2, 8, 25, 44, 45, 62, 66, 68, 86, 90, 95, 105] or CXL [3, 24, 41, 42, 46, 50, 51, 70, 114, 117, 120], only focus on memory capacity utilization and do not alleviate either spatial or temporal over-provisioning for memory bandwidth. Moreover, although CPU vendors have introduced hardware throttling mechanisms [20, 54] by inserting delays between the L2 and the last-level cache (LLC), previous studies [37, 84, 97, 110, 115, 118] show that these techniques cannot precisely allocate memory bandwidth or mitigate interference.

To address the challenges of memory management, we propose RAMRYDER, a software-defined elastic memory system for cloud VMs. RAMRYDER comprises three key techniques. First, RAMRYDER enables bandwidth allocation by assigning memory channels[1] to VMs and taking control of the mapping between memory pages in the guest OS and underlying memory channels. This also provides robust performance isolation between VMs on different channels. Second, RAMRYDER enables the system to allocate memory bandwidth and capacity (mostly) independently from CXL devices. To achieve this, RAMRYDER controls the number of CXL channels mapped to the VM's physical memory and how pages are interleaved

*Corresponding author: Erci Xu (jostep90@gmail.com).

[1] In this paper, a DIMM channel denotes a DDR channel attached to the CPU and populated with DIMMs, while a CXL channel denotes a DDR channel inside a CXL memory device. We use memory channel as a generic term for both DIMM and CXL channels.

across DIMM and CXL channels. Finally, RAMRYDER allows dynamic changes to bandwidth and capacity allocation by adjusting channel allocation and page-to-channel mapping at runtime.

RAMRYDER[2] runs on unmodified commodity servers by exposing memory channels to software via BIOS/UEFI reconfiguration and memory region detection, and implementing a user-space resource manager for channel management, a hypervisor for channel virtualization, and a guest kernel for channel abstraction and page-to-channel allocation policies.

We evaluate RAMRYDER on an AMD EPYC Zen 5 server with DDR5 DIMMs and CXL 2.0 memory devices. The results show that RAMRYDER enables VMs to achieve bandwidth and latency close to those of exclusive hardware access, and reduces the tail latency impact of a noisy neighbor by up to 42.2% on shared hardware. Moreover, RAMRYDER improves average capacity and bandwidth utilization by up to 28.6% and 43.2%, respectively, across the cluster.

The rest of the paper is organized as follows. §2 introduces relevant technologies. §3 analyzes memory bandwidth underutilization and the limitations of existing approaches. §4 and §5 present the design and implementation of RAMRYDER, respectively. §6 evaluates RAMRYDER. §7 discusses RAMRYDER's limitations and target scenarios, §8 surveys related work, and §9 concludes the paper.

2 Background

RAMRYDER targets memory systems for cloud VMs and relies on microarchitectural details of the memory subsystem. This section provides background on relevant technologies.

2.1 Memory Provisioning in the Cloud

Cloud vendors offer elastic infrastructure services (e.g., compute [7,9,81], storage [13,15,31,80], and networking [14,29]), but per-instance memory allocation remains static throughout the instance's lifetime. To accommodate diverse user requirements, cloud vendors design several types of physical servers with different memory-to-CPU ratios as instance types [10, 16, 17] and split each server into several units as instance sizes. For example, AWS defines its compute-optimized instances [10] with a 2:1 memory-to-vCPU ratio (i.e., 2 GB of memory per vCPU), its general-purpose instances [16] with a 4:1 ratio, and its memory-optimized instances [17] with an 8:1 ratio. The largest memory-optimized instance provides 192 vCPUs and 1536 GB of memory [12] while the smallest one provides 1 vCPU and 8 GB of memory. Tenants must select one of the instance types and then an instance size based on estimates of the workloads.

2.2 Memory Organization Inside the Server

A server's memory consists of several hardware components, and DRAM mapping determines how physical memory is mapped to them. This subsection briefly introduces the memory subsystem and DRAM mapping relevant to RAMRYDER.

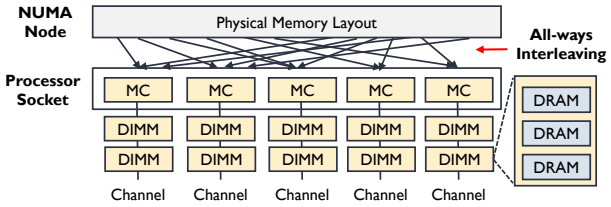


Figure 1: Server memory subsystem (§2.2). *Memory controllers (MCs) and channels are integrated into each processor socket. The physical memory of a NUMA node is interleaved across all channels of the socket, leading memory accesses to share and compete for the aggregate channel bandwidth.*

The server memory subsystem consists of memory controllers, memory channels, DIMMs, and DRAM. As shown in Figure 1, memory controllers and memory channels are integrated into the processor socket, and the DIMMs are installed in slots connected to each channel. From the OS perspective, the DIMMs attached to a processor socket appear as the physical memory of a distinct NUMA node.

DRAM mapping defines how physical memory addresses are mapped to underlying hardware components, determining how much parallelism across memory channels is utilized when accessing a given sequence of pages and consequently affecting memory bandwidth. Unlike virtual-to-physical address mapping, which is controlled by the OS via page tables and can change at runtime, DRAM mapping is set by hardware registers at boot time through BIOS/UEFI settings [19,28,48] and remains fixed after boot. Modern servers interleave sequential physical addresses across all memory channels in a socket (i.e., all-ways interleaving shown in Figure 1) at cache-line granularity (i.e., 64 B [75]) or larger (e.g., 256 B to 4 KB [19,28,48]) to maximize memory bandwidth.

2.3 Memory Expansion Beyond the Server

CXL [34] enables DRAM expansion over PCIe. DRAM on CXL is also subject to DRAM mapping, but with additional complexity. CXL devices contain internal DDR channels [102] and are exposed as physical memory in CPU-less zNUMA [70] nodes in servers. CXL devices can be interleaved at granularities from 256 B to 16 KB [59,96]. Each CXL component (e.g., a root port or CXL device) includes Host-managed Device Memory (HDM) registers that define the interleaving granularity [59,96] and can be configured via BIOS/UEFI and CXL configuration tools [32].

3 Motivation

Memory bandwidth in data centers is severely underutilized, and this underutilization is orthogonal to the well-documented underutilization of memory capacity. Bandwidth underutilization has several causes, yet existing techniques for managing capacity or throttling bandwidth do not address the problem.

[2]The source code of RAMRYDER is publicly available at https://github.com/ramryder-project/ramryder

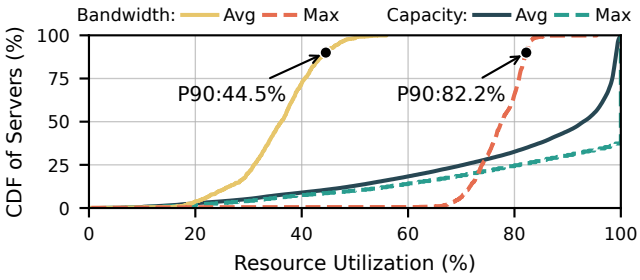


Figure 2: Resource utilization across servers in the cloud (§3.1). *Memory bandwidth is more severely underutilized than memory capacity: 90% of servers in the cloud use no more than 44.5% of available bandwidth on average and no more than 82.2% at peak.*

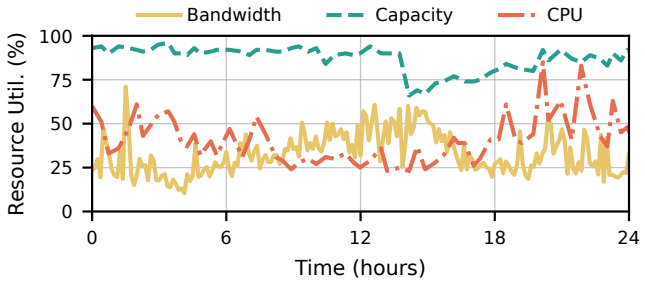


Figure 3: Resource utilization of a single server over one day (§3.1). *A closer look at individual servers reveals decoupled resource utilization trends: memory capacity, memory bandwidth, and CPU utilization can vary over time and in different ways.*

3.1 Bandwidth is Underutilized yet Understudied

Memory bandwidth is more severely underutilized than memory capacity yet remains understudied. To quantify this underutilization, we analyze cluster traces[3] from a major cloud vendor [6] and summarize the following two key observations.

Severely stranded bandwidth. Memory bandwidth underutilization leaves substantial bandwidth stranded in cloud data centers (Figure 2): 90% of servers exhibit average bandwidth utilization below 44.5%; even peak bandwidth utilization stays below 82.2% for 90% of servers. In contrast, 55.4% of servers exhibit both average and maximum capacity utilization above 90%, although 18.3% of servers still have average capacity utilization below 60%. Therefore, optimizing memory resource efficiency should consider both memory capacity and bandwidth.

Decoupled resource utilization. Memory capacity and bandwidth utilization vary over time and in different ways (Figure 3): memory capacity and bandwidth utilization on a physical server are not correlated with each other or with CPU utilization. Therefore, independently managing memory capacity and bandwidth can potentially consolidate complementary workloads and improve overall memory resource efficiency.

3.2 Root Causes of Underutilization

Memory resource underutilization can be attributed to both spatial and temporal over-provisioning.

Spatial over-provisioning. Users with strict performance QoS often want exclusive access to dedicated hardware resources. A typical practice in multi-tenant cloud environments is over-provisioning, such as subscribing to at least half of a physical server (i.e., one CPU socket). This guarantees exclusive access to hardware resources [22], including the LLC and the memory channels described in §2.2, and thus avoids performance interference from co-located VMs. However, this approach achieves isolation by purchasing extra memory and CPU capacity, which is inefficient for both customers and vendors.

Consider a dual-socket server with 96 cores, 384 GB memory capacity, and 600 GB/s memory bandwidth per socket. User A runs a latency-sensitive service (e.g., Redis [89], Memcached [79]) that requires 48 cores, 256 GB capacity, and 50 GB/s bandwidth, while user B runs a bandwidth-intensive job (e.g., STREAM [78], graph processing [23]) requiring 48 cores, 96 GB capacity, and 500 GB/s bandwidth. In principle, these complementary workloads could consolidate on a single socket without over-provisioning. In practice, however, meeting latency targets and bandwidth guarantees forces user A to occupy an entire socket, while user B must take the other full socket to secure sufficient bandwidth.

Temporal over-provisioning. The memory resource is allocated at VM creation and remains fixed throughout the VM lifespan. This forces users to provision memory resources to accommodate the applications' peak loads, resulting in underutilization at off-peak times. In this case, users appear to over-provision memory capacity, but in practice they are reserving memory bandwidth to sustain the applications' peak loads. As shown in Figure 3, despite the high demand for capacity, its utilization remained below 80% for four consecutive hours during the day due to over-provisioning for peak load. Across the full trace dataset (§3.1), more than 30% of servers show long periods (over one hour) of off-peak capacity usage and over 90% of servers show long periods of off-peak bandwidth usage.

3.3 The Limitations of Existing Approaches

Modern systems provide a rich array of memory management features in hardware and software, but none of them effectively addresses the bandwidth utilization problem. This subsection discusses existing approaches and their limitations.

Memory pooling. Prior work utilizes CXL to pool memory resources to improve memory utilization [3, 24, 41, 42, 46, 50, 51, 70, 114, 117, 120]. However, these approaches mainly manage memory capacity rather than bandwidth, and thus do not eliminate spatial over-provisioning for performance QoS or temporal over-provisioning for peak loads.

[3] The traces span thousands of physical nodes running diverse public-cloud workloads, including latency-sensitive services (e.g., e-commerce) and bandwidth-intensive jobs (e.g., MapReduce, Spark, and ML training).

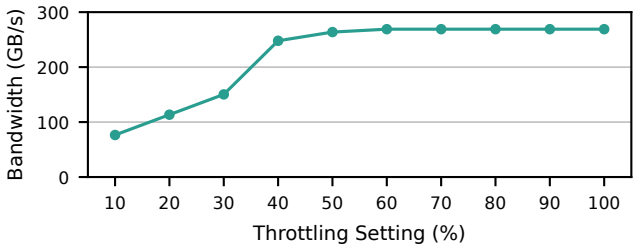


Figure 4: Memory bandwidth with different hardware throttling settings (§3.3). *Hardware throttling cannot precisely or linearly control memory bandwidth. It provides noticeable bandwidth control only at very low throttling settings, where the injected delay is long.*

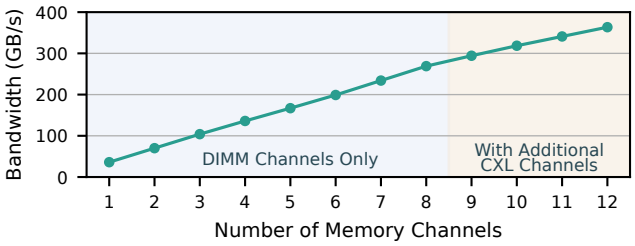


Figure 5: Memory bandwidth with different memory channel configurations (§4). *Bandwidth scales nearly linearly with available DIMM channels. CXL channels provide additional bandwidth and follow the same scaling trend.*

Hardware throttling. CPU vendors provide hardware throttling mechanisms to allocate memory bandwidth and reduce interference among co-located VMs (e.g., Intel MBA [54], AMD QoS [20]). These techniques insert delays between the L2 cache and the LLC to throttle bandwidth usage. However, delay-based throttling cannot provide precise bandwidth control (see Figure 4) and thus fails to effectively mitigate interference caused by co-located VMs (see more experimental results in §6.1 and §6.2). Previous studies have also reported this limitation [37, 84, 97, 110, 115, 118]. Furthermore, inserting extra delays wastes CPU cycles by stalling cores [39].

Software throttling. To control and isolate memory bandwidth, several software-based throttling mechanisms have been proposed. Canvas [105] isolates memory bandwidth by scheduling network packets. Spirit [67] controls memory bandwidth by limiting network transmission. However, these techniques target RDMA-based remote memory and cannot control memory bandwidth for cache-coherent memory (i.e., DIMMs and CXL devices) because, after pages are mapped, memory accesses are issued by the CPU via load/store instructions without software involvement on the access path.

Indirect approaches. Since there is no direct bandwidth allocation mechanism [26], several indirect approaches have been proposed, including limiting the number of cores [39, 74, 115], reducing core frequency [52, 84], and restricting LLC ways [26, 83]. However, these techniques are impractical in cloud environments, as they compromise other key VM metrics (e.g., CPU cores and frequency) in exchange for memory bandwidth control or are only effective for LLC-sensitive applications with LLC-related controls [26, 83].

4 RAMRYDER

RAMRYDER is a software-defined elastic memory system for cloud VMs, providing flexible capacity and bandwidth allocation with robust performance isolation. Its key insight is to manage memory at channel granularity, motivated by two observations. First, bandwidth scales with available memory channels, including both DIMM and CXL channels (Figure 5). Second, congestion stems from channel sharing caused by hardware-controlled interleaving. Therefore, RAMRYDER

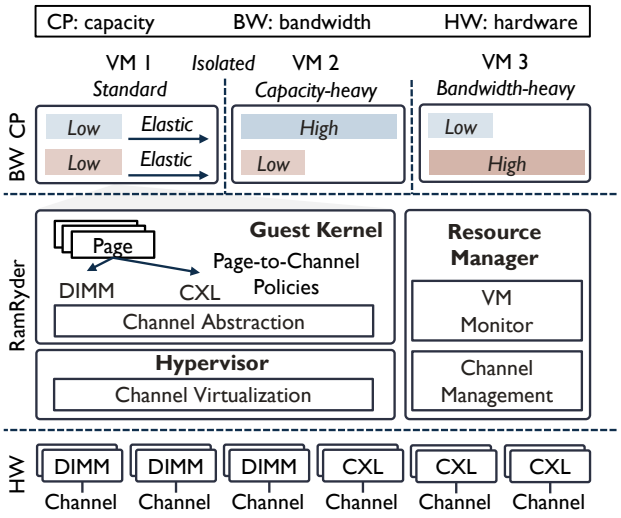
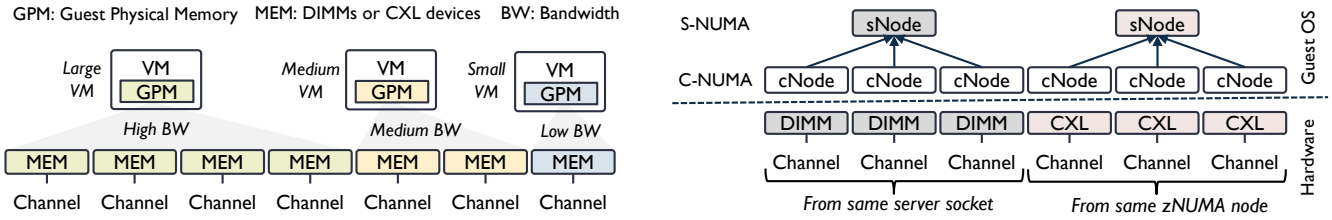


Figure 6: Overview of system architecture (§4). *RamRyder enables memory channel allocation in software to achieve elastic capacity and bandwidth allocation with performance isolation. The runtime includes a user-space resource manager for channel management, a hypervisor for channel virtualization, and a guest kernel for channel abstraction and page-to-channel allocation policies.*

takes the role of hardware and directly manages channel allocation and page-to-channel mapping in software.

Goals. To address the challenges of memory resource efficiency in cloud data centers (§3), RAMRYDER targets three goals: **Goal 1:** enable bandwidth allocation and performance isolation on commodity servers without hardware modifications; **Goal 2:** extend capacity and bandwidth independently; and **Goal 3:** adjust capacity and bandwidth dynamically.

Architecture. RAMRYDER manages memory channels and the devices attached to each channel in software (Figure 6). It statically allocates DIMMs to provide guaranteed capacity and bandwidth, and uses CXL devices to provide elastic best-effort capacity and bandwidth that VMs can independently subscribe to and dynamically scale on demand. The RAMRYDER runtime consists of a resource manager in the host user space, a hypervisor, and a modified guest kernel. The resource manager is responsible for per-channel capacity



(**a**) Channel Allocation

(**b**) Channel Abstraction

Figure 7: Overview of memory channel allocation and abstraction (§4.1). *To allocate memory channels to VMs, the hypervisor maps the VM's guest physical memory to memory devices on the target channels (**a**). Memory on each channel is then exposed as a C-NUMA node (i.e., cNode) in the guest OS, and multiple cNodes are organized as an S-NUMA node (i.e., sNode) by the guest OS (**b**).*

management and channel allocation, the hypervisor enables channel virtualization, and the guest kernel provides channel abstraction and page-to-channel allocation policies in the OS.

Key techniques. To achieve Goal 1, RAMRYDER allocates memory bandwidth by allocating memory channels and controls how pages in the guest OS map to underlying channels. This also provides robust performance isolation since VMs on different DIMM channels are unlikely to interfere with each other (§4.1). To realize Goal 2, RAMRYDER introduces a channel selection policy that determines how many channels to allocate based on bandwidth and capacity demands, and a cross-tier page allocation policy that interleaves pages across DIMM and CXL channels to use additional CXL bandwidth (§4.2). Finally, to achieve Goal 3, RAMRYDER introduces a channel hot-plug/unplug mechanism to allocate and reclaim bandwidth at runtime. It also leverages lazy migration to redistribute previously allocated pages, allowing these pages to utilize the bandwidth of newly attached channels (§4.3).

4.1 Enabling Bandwidth Allocation and Isolation

To achieve bandwidth allocation and performance isolation, RAMRYDER needs to control how memory pages in a VM map to underlying memory channels. This requires four steps. First, RAMRYDER enables channel allocation in software to override hardware-controlled channel interleaving. Next, RAMRYDER isolates LLCs across VMs to avoid cache-level contention that reduces the benefits of channel allocation. Then, RAMRYDER introduces a channel abstraction in the guest OS to expose this hardware component while remaining compatible with existing kernel mechanisms. Finally, RAMRYDER introduces a new page allocation policy to fully utilize the allocated channels within each VM.

Channel allocation. To realize memory channel allocation for VMs, RAMRYDER overrides the hardware-controlled interleaving (§2.2) by directly managing the memory devices attached to each channel, including DIMMs and CXL devices. To allocate channels to a VM, the RAMRYDER hypervisor maps the VM's guest physical memory to the memory devices attached to the target channels (Figure 7(a)). §5.1 explains how RAMRYDER identifies the DIMMs and CXL devices attached to each channel on commodity servers via BIOS/UEFI.

For locally attached DIMMs, RAMRYDER allocates channels in proportion to the VM's memory capacity. For example, a VM that subscribes to 40% of the memory capacity receives 40% of the dedicated channels, which provide 40% of the available bandwidth. Moreover, using this channel allocation mechanism, RAMRYDER can independently allocate additional capacity and bandwidth from CXL devices (§4.2).

LLC isolation. Channel allocation alone cannot achieve full performance isolation since the LLC can introduce cache-level contention across VMs. Therefore, RAMRYDER further isolates LLCs across VMs by pinning each VM's vCPUs on separate chiplets [38], each containing one core complex (CCX) with its own LLC [21].

Channel abstraction. After channels are allocated to a VM, the guest OS needs to recognize the new hardware component while remaining compatible with existing kernel mechanisms. RAMRYDER abstracts memory channels as C-NUMA (i.e., Channel-NUMA) nodes and uses S-NUMA (i.e., Server-NUMA) nodes to preserve the original server-level NUMA topology (e.g., server sockets and zNUMA nodes for CXL [70]). This channel abstraction allows the guest kernel to leverage existing NUMA primitives for page-to-channel allocation while preserving server-level NUMA optimizations.

Specifically, the RAMRYDER hypervisor virtualizes the guest physical memory mapped to each memory channel as a distinct C-NUMA node (cNode) and exposes its physical topology (e.g., socket/zNUMA and channel) to the RAMRYDER guest kernel via the ACPI table [58], which describes hardware topology. The guest kernel then detects the topology of each cNode and constructs S-NUMA nodes (sNodes) by grouping cNodes from the same socket/zNUMA domain (Figure 7(b)). The kernel also maintains a mapping between sNodes and cNodes and only exposes sNodes to users.

Page allocation policy. To better utilize allocated channels within a VM, the guest OS needs to decide how to allocate memory pages across those channels. The RAMRYDER guest kernel uses a channel-interleaving policy to maximize performance. Specifically, when allocating memory pages for applications, the RAMRYDER guest kernel performs page allo-

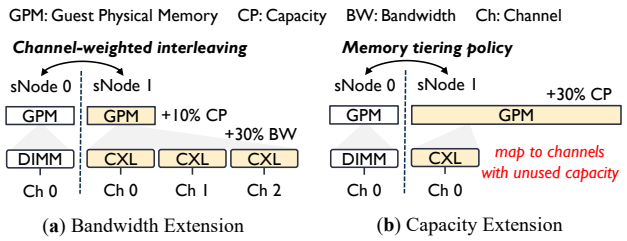


Figure 8: Example of bandwidth and capacity extensions (§4.2). *For bandwidth extension (**a**), RamRyder maps guest memory across multiple CXL channels and leverages a channel-weighted interleaving policy to utilize additional CXL bandwidth. For capacity extension (**b**), RamRyder maps guest memory to channels with unused capacity and leverages a tiering policy to utilize CXL capacity.*

cation in two steps and applies a different policy at each step. It first selects an sNode based on the server-level NUMA-aware policy and then chooses a cNode within that sNode by interleaving pages equally across all cNodes. This channel interleaving allows applications to utilize all available memory channels in parallel within an sNode and achieves performance comparable to hardware-controlled interleaving, which operates at granularities ranging from cache-line size to page size [19, 28, 48, 75].

4.2 Extending Capacity and Bandwidth

RAMRYDER uses CXL devices to provide additional memory capacity and bandwidth. To extend these resources independently, RAMRYDER first uses a channel selection policy to determine how many channels to allocate based on capacity and bandwidth demands. Then, because CXL memory appears as a separate memory tier (i.e., a remote S-NUMA node in the guest OS), RAMRYDER introduces channel-weighted interleaving as a cross-tier page allocation policy for utilizing additional CXL bandwidth.

Channel selection. Given a certain amount of guest physical memory (GPM), RAMRYDER can control the available bandwidth by selecting the number of channels the memory will span. RAMRYDER maps GPM across multiple CXL channels when a VM requires additional bandwidth but only limited capacity (Figure 8(a)). In contrast, RAMRYDER maps GPM to fewer CXL channels when a VM needs additional capacity without proportional bandwidth demand (Figure 8(b)).

With this mapping, capacity and bandwidth utilization within a single CXL channel may become imbalanced. To mitigate this imbalance without sacrificing performance QoS, RAMRYDER pairs VMs with complementary memory demands and co-locates them on the same CXL channels. For instance, if one VM consumes an additional 10% capacity and 60% bandwidth (i.e., occupying 60% of the CXL channels), RAMRYDER allocates the remaining 50% capacity on those channels to another VM that requires less than 50% of CXL capacity with little additional bandwidth demand.

Channel-weighted interleaving. After allocating CXL memory and exposing it as a separate tier, the RAMRYDER guest kernel applies a traditional memory tiering policy [77], which migrates hot pages to DIMMs and keeps cold pages in CXL devices, to utilize additional CXL capacity and introduces channel-weighted interleaving as a cross-tier page allocation policy for utilizing additional CXL bandwidth (Figure 8).

The channel-weighted interleaving policy maximizes bandwidth across tiers by using a weighted ratio between S-NUMA nodes (sNodes) based on each sNode's maximum bandwidth. The maximum bandwidth of a sNode is determined by its number of channels and the bandwidth provided per channel.

For example, consider a VM with three DIMM channels in sNode-0 and one CXL channel in sNode-1, where each DIMM channel provides 36 GB/s and each CXL channel provides 27 GB/s of bandwidth. The guest kernel applies an 8:3 weighted ratio between sNode-0 and sNode-1 (derived from 36×3 and 27×1) by allocating 8 pages from sNode-0 followed by 3 pages from sNode-1, and so forth. Within each sNode, the guest kernel retains channel interleaving equally across its C-NUMA nodes, as described in §4.1.

4.3 Adjusting Capacity and Bandwidth at Runtime

To dynamically adjust memory capacity and bandwidth at runtime, RAMRYDER uses different mechanisms for the two resources. For capacity, RAMRYDER reuses existing Linux memory statistics and the hot-plug/unplug mechanism in the guest OS. For bandwidth, RAMRYDER monitors VM bandwidth usage and introduces a channel hot-plug/unplug mechanism, along with page redistribution, to adjust VM bandwidth.

Capacity monitoring and allocation. RAMRYDER monitors each VM's memory usage by reading Linux's statistics in `meminfo` via a domain socket and hot-plugs memory chunks from CXL devices into the guest when usage exceeds a high threshold (e.g., 80%) or hot-unplugs them when usage falls below a low threshold (e.g., 40%) for consecutive seconds.

Bandwidth monitoring. RAMRYDER leverages processor performance counters to monitor VM bandwidth usage and detect bandwidth bottlenecks and waste for bandwidth adjustment. Every second, RAMRYDER measures the aggregate bandwidth across all cores assigned to each VM. When a VM's bandwidth consumption exceeds the high threshold (e.g., 80%), the RAMRYDER resource manager allocates additional bandwidth (i.e., CXL channels) to the VM. Conversely, if bandwidth consumption stays below the low threshold (e.g., 40%) for consecutive seconds, the RAMRYDER resource manager reclaims bandwidth from that VM.

Channel hot-plug/unplug. To dynamically adjust memory bandwidth without altering capacity, RAMRYDER needs to change the channels allocated to the VM at runtime. To enable this capability, RAMRYDER introduces a channel hot-plug/unplug mechanism. This requires four steps. First, RAMRYDER allocates extra guest physical memory mapped to an

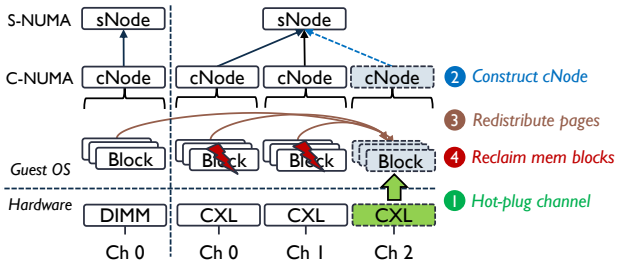


Figure 9: Workflow of adaptive bandwidth adjustment (§4.3). *Adjusting bandwidth at runtime first adds memory channels, then constructs new C-NUMA nodes (cNodes), redistributes pages across cNodes, and finally reclaims memory blocks to retain the same capacity. These four steps are color-coded, with matching icons and text labels indicating the corresponding operations.*

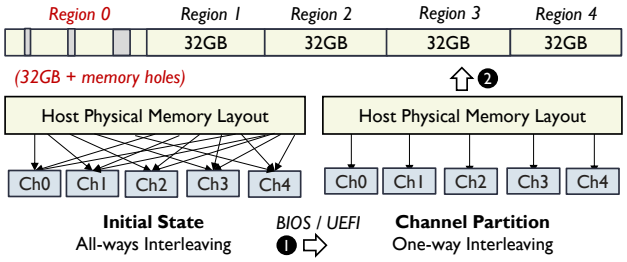


Figure 10: Process of memory topology provisioning (§5.1). *RamRyder gains software control over per-channel memory by first partitioning memory channels via BIOS/UEFI and then detecting per-channel regions from the memory layout. These regions are then reserved as DAX devices for software management.*

additional CXL channel. If the VM previously had a total capacity X across N channels, the new channel is mapped to a new guest physical memory region of size $X/(N+1)$. Next, the RamRyder hypervisor hot-plugs the additional guest physical memory into the guest OS (1, Figure 9). The RamRyder guest kernel then constructs a cNode and adds it to an existing sNode (2, Figure 9), or creates a new one if no sNode exists. Then, RamRyder reclaims the same amount of capacity $X/(N+1)$ from the VM's previously allocated channels by hot-unplugging memory from existing cNodes under the same sNode (4, Figure 9). Since the target reclaim size is $X/(N+1)$ and the VM previously spanned N channels, each cNode should return $X/(N+1)/N$ capacity to keep capacity balanced across all cNodes under a sNode. Through these steps, the VM gains additional bandwidth while maintaining the same total memory capacity.

Note that this procedure enables the VM to immediately utilize additional bandwidth if the workload is allocation-intensive. However, if pages are already allocated on previous channels, RamRyder performs page redistribution (3, Figure 9) between step 2 and 4 above to balance pages evenly across channels. We explain this redistribution process in detail below. Similarly, during channel hot-unplug, RamRyder applies the same procedure in reverse to reclaim bandwidth when the workload demand decreases, including page redistribution across fewer channels when necessary.

Page redistribution. To utilize the newly attached channels, RamRyder redistributes existing pages so that their page-to-channel placement is realigned with the original channel-weighted interleaving (§4.2) and page allocation (§4.1) policies. This redistribution is performed by triggering page faults followed by lazy migration. After hot-plugging new channels, the RamRyder guest kernel scans the page table entries of mapped pages and clears their present bits, which triggers page faults when these pages are subsequently accessed by applications. Upon a page fault, RamRyder recalculates the target sNode and its corresponding cNode, and marks misplaced pages for migration to the new cNode.

5 Implementation

RamRyder is implemented on unmodified commodity servers with a memory topology provisioning toolchain and runtime software. The toolchain reconfigures servers to expose the underlying memory topology required to run RamRyder, while the runtime software implements RamRyder's key components as an end-to-end system.

5.1 Memory Topology Provisioning

To realize RamRyder on commodity servers, we need to gain control over raw memory devices, including DIMMs and CXL devices, under each memory channel in software. To achieve this, we implement a memory topology provisioning toolchain that involves the following operations.

Partitioning memory channels. Since the physical address space is interleaved across all DIMM channels by default, the first step to gain control over the DIMMs under each channel is to partition DIMM channels. We partition DIMM channels primarily by disabling channel interleaving via BIOS/UEFI during system boot (1, Figure 10). After partitioning DIMM channels, the physical address space is linearly mapped to underlying channels (see channel partition in Figure 10).

Detecting memory regions. The second step is to detect each DIMM channel's memory region. As the physical address space is linearly mapped to underlying channels after partitioning, we calculate each DIMM channel's memory region based on the capacity of the DIMMs populated in that channel (2, Figure 10). In practice, the host OS may reserve holes in the memory layout [60]. RamRyder adjusts its address calculations by adding offsets to skip these holes when determining each channel's region (see region 0 in Figure 10).

Reserving memory regions. The final step is to reserve memory under each DIMM channel in user space. We keep 10 GB for the host OS to ensure it has sufficient memory to operate and reserve the remaining region under each channel as a separate DAX device using the `memmap` kernel parameter.

CXL device reservation. For CXL devices, current PCIe Gen 5 CXL devices have one DDR5 channel internally [102].

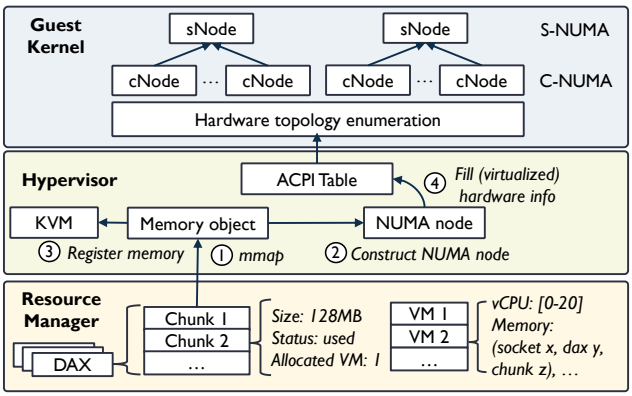


Figure 11: Implementation details. (§5.2). *The RamRyder runtime consists of a user-space resource manager, a hypervisor, and a guest kernel. The resource manager splits per-channel DAX devices into chunks; the hypervisor virtualizes allocated chunks from the same device as a NUMA node and exposes the hardware topology via the ACPI table; and the guest kernel constructs cNodes and sNodes from the exposed hardware topology.*

Therefore, we only need to reserve the whole CXL device, which represents DRAM under one CXL channel. We reserve each CXL device as a separate DAX device in user space via BIOS/UEFI and the CXL configuration tool [32].

5.2 Runtime Software

RAMRYDER's runtime (Figure 11) consists of a resource manager (RM) built from scratch, a hypervisor built on top of QEMU, and an extended guest kernel based on Linux.

Resource manager. The RAMRYDER RM is a user-space daemon that manages memory and monitors VMs. The RM splits each DAX device into 128 MB chunks as the minimal allocation units for VMs, aligned with the kernel's hot-plug block size. Each chunk is uniquely identified by `(socket_id, dax_id, chunk_id)`. The RM also exposes an RPC interface over domain sockets for QEMU to request memory during VM boot. In addition, it uses Linux `perf_event` APIs [61] to collect per-core performance counters and aggregate them into per-VM bandwidth utilization, and interacts with the QEMU Guest Agent [35] to obtain per-VM capacity usage.

Hypervisor. The RAMRYDER hypervisor is QEMU with four major modifications. First, it organizes memory chunks from different DAX devices (i.e., memory channels) into distinct NUMA nodes and exposes the physical topology (i.e., socket and DAX indices) to the guest kernel via reserved domains in the ACPI table [58]. Second, it extends the QEMU Guest Agent to support memory-usage queries used by the RM. Third, it integrates an RPC client that allows QEMU to request memory from the RM during VM launch. Finally, it adds support for attaching and detaching channels at runtime.

Guest kernel. The RAMRYDER guest kernel extends Linux with four major modifications. First, it extends the memory probing process to retrieve physical topology information of C-NUMA nodes from the ACPI table during system boot. Second, it implements the S-NUMA and C-NUMA abstractions within the existing NUMA framework and establishes metadata mappings between them. Third, it implements a new page-mapping policy in the memory-policy layer to maximize performance under these abstractions. Finally, it extends the kernel's memory hot-plug/unplug mechanism to construct new S-NUMA and C-NUMA nodes and trigger page redistribution upon C-NUMA hot-plug/unplug events.

6 Evaluation

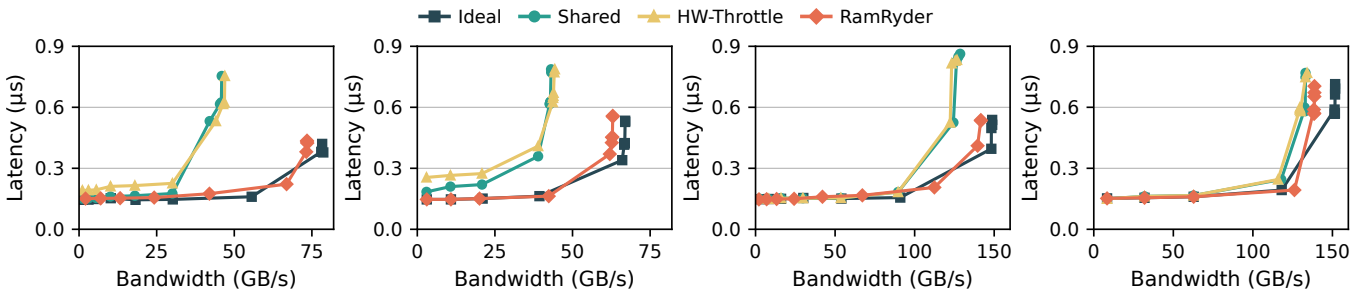
Our evaluation aims to answer the following questions:

- What benefits does RAMRYDER provide for VMs in terms of performance and isolation (§6.1)?
- Can RAMRYDER provide additional memory capacity or bandwidth while maintaining QoS for applications with asymmetric memory demands (§6.2)?
- Can RAMRYDER improve memory utilization across both capacity and bandwidth in cloud data centers (§6.3)?
- What overheads does RAMRYDER introduce, and how sensitive is RAMRYDER to highly dynamic workloads (§6.4)?

Testbed. Our server has an AMD EPYC Zen 5 processor with 128 logical cores (SMT enabled) and 12 DDR5 DIMM channels per socket. The cores are organized into 8 CPU dies, each with a 32 MB L3 cache and 16 logical cores. To match cloud compute-optimized instances (i.e., high compute with minimal memory), we use 8 DIMM channels in one socket, each populated with a 32 GB DDR5 DIMM. We create four VMs (details below), each using the same vCPU-to-memory ratio as AWS and Alibaba compute-optimized instances [5, 11]. The server also includes four 256 GB Samsung CXL 2.0 memory devices and runs Debian with Linux 6.15.

VM setup. We create four VMs to model multiple tenants. VM 1 and VM 2 each have 16 vCPUs and 32 GB of memory, while VM 3 and VM 4 each have 48 vCPUs and 96 GB of memory, following AWS and Alibaba compute-optimized instances [5, 11]. Each VM's vCPUs are pinned to logical cores on separate core complexes (CCXs) [21], giving each VM dedicated LLCs. For RAMRYDER, we initially allocate one DIMM channel each to VM 1 and VM 2 and three DIMM channels each to VM 3 and VM 4, keeping memory bandwidth allocation proportional to capacity and CPU cores.

Compared approaches. We compare RAMRYDER against three approaches: *Ideal*, where VMs have exclusive access to hardware resources without co-located VMs; *Shared*, the default configuration where multiple VMs co-locate on a server; and *HW-Throttle*, which throttles memory bandwidth on top of *Shared* to mitigate co-located VM interference using Intel MBA [54] and AMD QoS [20]. For *HW-Throttle*, bandwidth is throttled in proportion to allocated capacity and CPU cores (e.g., a VM allocated 60% of the capacity and CPU cores is throttled to 60% of the maximum bandwidth).



(a) VM 1 (16 vCPUs): Read (b) VM 2 (16 vCPUs): 3:1 R/W (c) VM 3 (48 vCPUs): Read (d) VM 4 (48 vCPUs): 3:1 R/W

Figure 12: Latency and bandwidth under co-located VMs (§6.1). *VMs with fewer vCPUs (VM 1 and VM 2) are more vulnerable to interference from co-located VMs with more vCPUs (VM 3 and VM 4). RamRyder mitigates contention and delivers performance close to the ideal baseline with exclusive hardware access.*

6.1 Microbenchmarks

RAMRYDER guarantees performance QoS in multi-tenant environments through bandwidth allocation and performance isolation, without dedicated hardware resources. This subsection uses microbenchmarks to quantify this benefit.

Configuration. We use Intel MLC [53] to generate memory-intensive workloads, with read-only traffic on VM 1 (16 vCPUs) and VM 3 (48 vCPUs), and a 3:1 read/write mix on VM 2 (16 vCPUs) and VM 4 (48 vCPUs). Each VM runs the benchmark on all available cores with a 100 MB buffer and a 64 B stride per core, ensuring the working set exceeds the CPU cache and accesses come from main memory. We run all VMs concurrently and measure memory bandwidth and average memory access latency.

Reads. Under read-only workloads (Figure 12(a) and (c)), both *Shared* and *HW-Throttle* suffer noticeable performance degradation, while RAMRYDER mitigates contention and delivers performance close to *Ideal*. In *Shared*, VM 1 experiences the most severe slowdowns: its memory access latency increases by up to 78.5%, and its maximum bandwidth drops by up to 41.2% compared to *Ideal*. This degradation arises because all VMs share the same memory channels, and VMs with more CPU cores can contend for more bandwidth, resulting in greater interference to VMs with fewer cores.

HW-Throttle, however, fails to mitigate contention and instead increases latency for the small VM with fewer cores (see the small gap between the green and yellow lines under 25 GB/s bandwidth in Figure 12(a)). This occurs because inserting extra delays cannot precisely control bandwidth and further wastes CPU cycles by stalling cores [39]. This observation aligns with prior work showing that *HW-Throttle* is ineffective for high-bandwidth workloads and only effective for low-bandwidth workloads when throttled to very low levels by inserting many delays [37, 84, 97, 115].

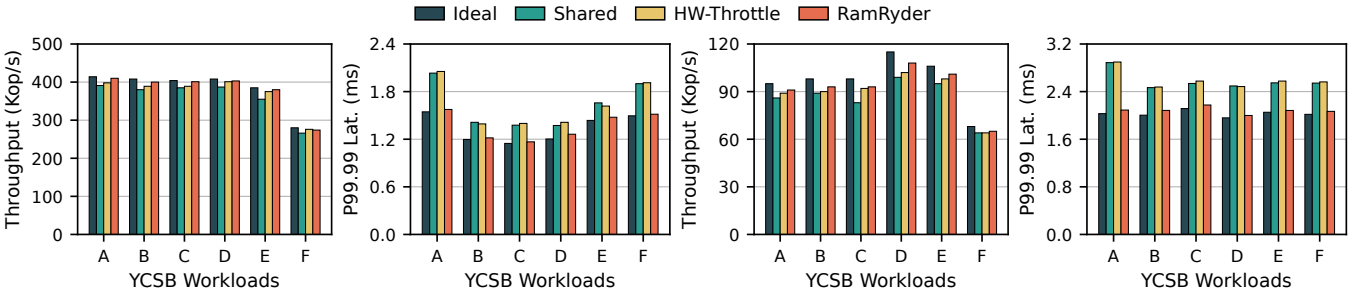
In RAMRYDER, VMs achieve bandwidth close to *Ideal* through channel allocation and isolation, which precisely control bandwidth and physically isolate VMs to avoid contention. For latency, VM 1 differs from *Ideal* by less than 5%, and its latency is up to 42.7% lower than that in *Shared*. A small gap to *Ideal* still exists because the VM in *Ideal* can use more available channels at the same bandwidth level, enabling greater parallelism and thus lower latency. Although RAMRYDER limits the number of channels, it effectively mitigates contention from co-located VMs through physical isolation.

Mixed reads/writes. Under mixed workloads (Figure 12(b) and (d)), the overall trends remain consistent with those under read-only workloads. However, VM 2 shows about 15.3% lower bandwidth than under read-only workloads for both *Ideal* and RAMRYDER. In addition, VM 4 experiences less impact under both *Shared* and *Ideal* than under read-only workloads. These differences arise from the varying capabilities of memory channels to handle reads versus writes. §6.4 analyzes the scalability of memory channels under different read/write ratios in detail.

6.2 Application Benchmarks

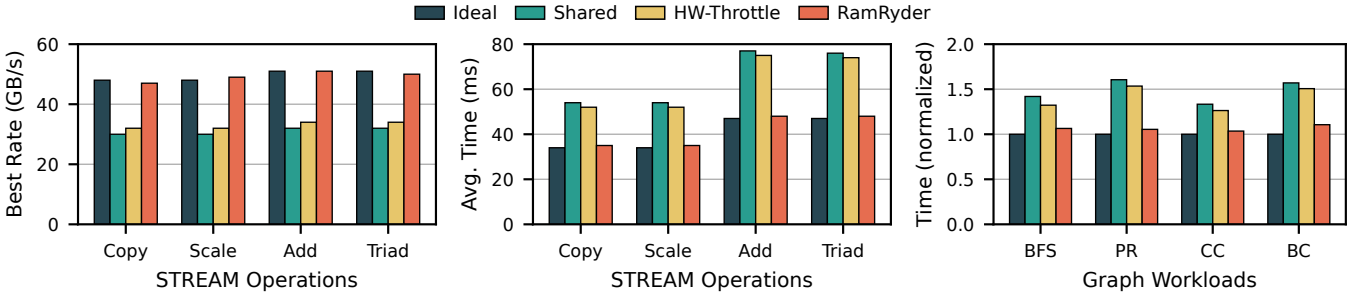
RAMRYDER enables independent memory capacity and bandwidth extension with CXL devices while maintaining performance QoS, without subscribing to extra memory resources (i.e., paying for additional capacity to obtain more bandwidth, or for additional bandwidth to get more capacity). This subsection uses application benchmarks to quantify this benefit.

Configuration. We evaluate two scenarios: (1) mixed workloads where capacity-demanding and latency-sensitive applications co-locate with bandwidth-intensive applications, and (2) all bandwidth-intensive applications. In the mixed scenario, we allocate an additional 100 GB of CXL memory to each of VM 1 and VM 2 and run Memcached and Redis with YCSB [33], respectively. In the bandwidth-only scenario, we allocate 10 GB of CXL memory and map it to one CXL channel for each of VM 1 and VM 2 and run STREAM and graph processing, respectively. Because VMs with fewer CPU cores are more susceptible to interference (§6.1), we allocate one CXL channel with 10 GB of CXL memory to each of VM 3 and VM 4 and generate peak mixed read/write traffic with Intel MLC to stress-test both scenarios. For *Ideal*, *Shared*, and *HW-Throttle*, we apply the existing Linux kernel



(a) Throughput of Memcached (b) Tail Latency of Memcached (c) Throughput of Redis (d) Tail Latency of Redis

Figure 13: Performance of Memcached and Redis under YCSB workloads with co-located VMs (§6.2). *RamRyder mitigates channel contention through channel isolation and thus achieves throughput and tail latency close to the Ideal baseline, while Shared and HW-Throttle suffer higher latency and lower throughput due to channel contention.*



(a) Throughput of STREAM (b) Execution Time of STREAM (c) Execution Time of Graph Processing

Figure 14: Performance of STREAM and graph-processing workloads with co-located VMs (§6.2). *RamRyder mitigates channel contention through channel isolation and thus achieves throughput and execution time close to the Ideal baseline, while Shared and HW-Throttle suffer lower throughput and higher execution time due to channel contention.*

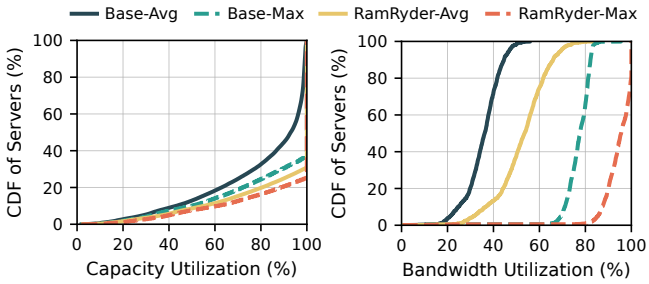
tiering [77] policy to utilize CXL memory for all four VMs.

Memcached. We populate 60 million key-value pairs (16 B keys and 1 KB values), and each YCSB workload issues 30 million operations. In terms of throughput (Figure 13(a)), all approaches achieve similar performance across YCSB workloads because memory bandwidth utilization remains low during execution, so throughput is not significantly affected by co-located VMs. However, both *Shared* and *HW-Throttle* exhibit higher tail latency (Figure 13(b)), particularly under YCSB-A (31.4% higher) and YCSB-F (27.2% higher). We observe that most memory accesses are served from local DIMMs, while CXL memory only stores cold pages with low access frequency. Therefore, the latency degradation primarily stems from contention on DIMM channels. In contrast, RAMRYDER mitigates contention from the bandwidth-intensive workloads and achieves latency close to *Ideal* by allocating exclusive DIMM channels to each VM.

Redis. We use the same configuration as in the Memcached experiment (60 million key-value pairs and 30 million YCSB operations per workload). The throughput (Figure 13(c)) and latency (Figure 13(d)) trends are similar to those seen in the Memcached results. We further observe that both *Ideal* and RAMRYDER achieve slightly higher throughput than *Shared* and *HW-Throttle*, particularly under YCSB-D (RAMRYDER improves throughput by 9% and *Ideal* by 16.2%). For latency, *Shared* and *HW-Throttle* incur up to 42.7% higher latency in the worst case (YCSB-A), while RAMRYDER remains within a 5% latency increase relative to *Ideal*.

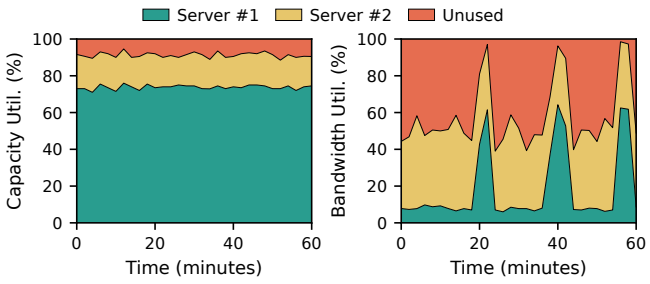
STREAM. We configure each STREAM workload (Copy, Scale, Add, and Triad) with an array size of 50 million elements to ensure that the working set far exceeds the LLC capacity, forcing most accesses to be served from main memory. In *Shared*, throughput (Figure 14(a)) is 37.3% lower and execution time (Figure 14(b)) is 58.8% higher compared to *Ideal* due to contention from co-located VMs. In *HW-Throttle*, both throughput and execution time remain similar to *Shared* (only about a 7% improvement for both metrics). By contrast, RAMRYDER achieves performance comparable to *Ideal*, with both throughput and execution time within 5% across all STREAM workloads, by allocating exclusive DIMM and CXL channels to the bandwidth-intensive VMs, thereby mitigating contention.

Graph processing. We generate graphs with 67 million nodes and 1.3 billion undirected edges and run four workloads: Breadth-First Search (BFS), PageRank (PR), Connected Components (CC), and Betweenness Centrality (BC).



(a) Utilization of Capacity (b) Utilization of Bandwidth

Figure 15: Cluster-level resource utilization before and after co-locating compatible workloads from cloud traces (§6.3). *RamRyder improves both capacity and bandwidth utilization by co-locating workloads with complementary demands within hardware limits.*



(a) Utilization of Capacity (b) Utilization of Bandwidth

Figure 16: Server-level resource utilization when replaying cloud workloads from a matched server pair (§6.3). *The two servers have complementary demands, allowing RamRyder to improve utilization while meeting both capacity and bandwidth demands.*

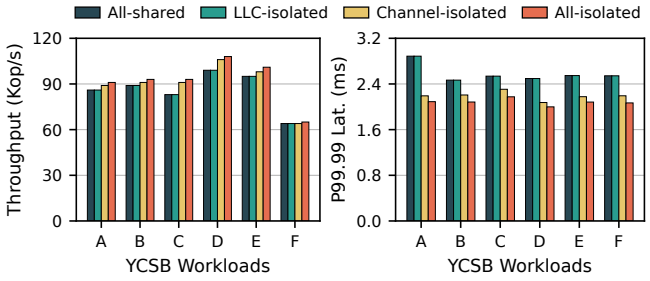
Because these workloads report only execution time and exhibit large differences in absolute runtime, we normalize all results to *Ideal* (Figure 14(c)). In *Shared*, execution time is 41.4% higher than *Ideal* due to contention from co-located bandwidth-intensive VMs. In *HW-Throttle*, execution time improves slightly but still shows a substantial gap compared to *Ideal*. By contrast, RAMRYDER reduces execution time by 25.2% relative to *Shared* and remains within 5% of *Ideal*. These benefits arise from RAMRYDER's channel allocation and isolation, which eliminate contention and ensure QoS.

6.3 Cloud Workloads

RAMRYDER improves both capacity and bandwidth utilization in cloud data centers by consolidating complementary workloads on the same server while guaranteeing performance QoS. This subsection quantifies these benefits at both the cluster and server levels by emulating cloud workloads using the cluster traces [6] described in §3.

Cluster-level utilization. We analyze the memory traces to identify server pairs whose combined capacity and bandwidth usage at each timestamp remains within hardware limits (i.e., under 100%), and then calculate the resulting cluster-level utilization. At the cluster level, RAMRYDER improves average and maximum capacity utilization at P30 by 28.6% and 22.1% (Figure 15(a)), and average and maximum bandwidth utilization at P90 by 43.2% and 26.1% (Figure 15(b)).

Server-level utilization. We replay the memory requests of selected server pairs using a custom workload generator that reproduces their timestamped capacity and bandwidth usage, and run these workloads on the previously configured VM 3 and VM 4. As shown in Figure 16, server 1 requires high capacity (over 70%) with low bandwidth (averaging 10.1%) and occasional bursts, whereas server 2 uses low capacity (below 20%) but high bandwidth (averaging 38.4%). RAMRYDER accommodates these complementary demands and dynamically allocates memory bandwidth when workload demands change, improving utilization while meeting each workload's requirements.



(a) Throughput (b) Tail Latency

Figure 17: Attribution of performance isolation with Redis under YCSB workloads (§6.4). *Channel isolation contributes more to reducing interference than LLC isolation, and combining both isolation mechanisms achieves the best performance.*

6.4 Overhead and Sensitivity Analysis

RAMRYDER achieves its performance and utilization benefits with limited overhead. This subsection quantifies the overhead of RAMRYDER's internal mechanisms and its sensitivity to highly dynamic workloads.

Attribution of performance isolation. We conduct an ablation study to quantify the contributions of LLC and channel isolation, using the same VM configurations and workloads as in §6.2. We compare the all-shared, LLC-isolated, channel-isolated, and all-isolated setups, using Redis on VM 2 as a representative workload. As shown in Figure 17, channel sharing causes more severe interference than LLC sharing. Therefore, LLC isolation alone provides limited benefit. Channel isolation achieves much stronger isolation, although it still leaves residual LLC contention. With both LLC and channel isolation enabled, RAMRYDER achieves the best performance.

Scalability of memory channels. To evaluate scalability, we launch a VM with 128 vCPUs and initially one DIMM channel. We use Intel MLC to measure peak bandwidth under different read/write ratios while incrementally adding DIMM and then CXL channels. As shown in Figure 18, bandwidth scales linearly with the number of DIMM channels for both

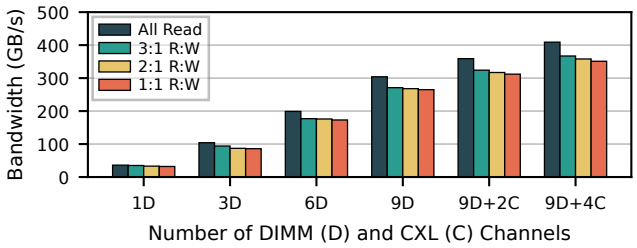
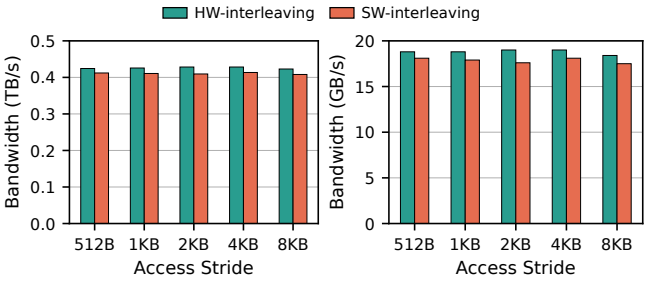


Figure 18: Scalability of software-defined memory channels (§6.4). *Memory bandwidth scales nearly linearly with DIMM channels under both read-only and mixed workloads; additional CXL channels further increase bandwidth.*



(a) Bandwidth with 128 threads (b) Bandwidth with 1 thread

Figure 19: Overhead of software-defined interleaving under different access strides (§6.4). *Software-defined interleaving delivers bandwidth close to hardware-controlled interleaving with low overhead across both 128-thread and single-thread workloads.*

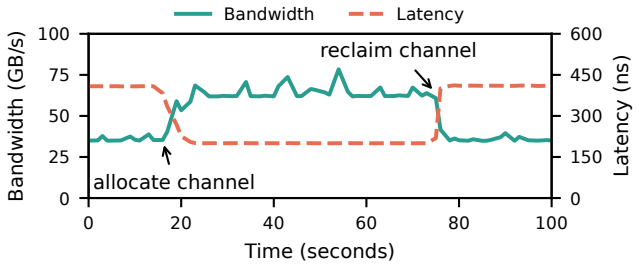


Figure 20: Bandwidth and latency during dynamic channel allocation and reclamation (§6.4). *Since channel changes require page redistribution, RamRyder leverages lazy migration to redistribute pages, avoiding large latency spikes despite the migration delay.*

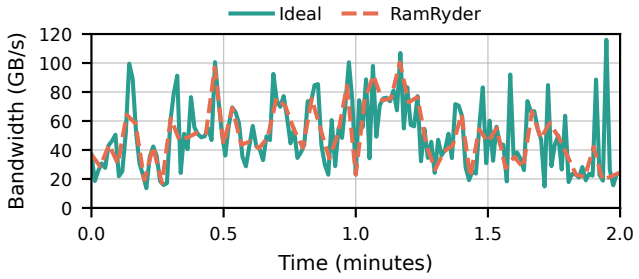


Figure 21: Bandwidth adjustment under highly dynamic workloads (§6.4). *RamRyder adapts to sustained bandwidth changes and approaches the over-provisioned ideal baseline, while transient bursts may be missed due to monitoring granularity and page-redistribution delays.*

read-only and mixed workloads. When CXL channels are added, the scaling slope decreases slightly because CXL channels provide lower bandwidth than DIMM channels.

Overhead of software-defined interleaving. To quantify this overhead, we launch a VM with 128 vCPUs and all DIMM channels, then use STREAM to measure peak bandwidth with multiple threads and a single thread under different stride sizes. With 128 threads, RAMRYDER achieves peak bandwidth comparable to hardware interleaving, with an average overhead of 3.6% and a maximum of 4.4% across stride sizes (Figure 19(a)). With a single thread, RAMRYDER introduces a higher average overhead of 5.1%, with a more visible gap at intermediate stride sizes, especially the 2 KB stride, where overhead reaches 7.4% (Figure 19(b)). This is likely because hardware interleaving benefits from finer interleaving granularity, which better exploits row buffer locality.

Overhead of dynamic bandwidth allocation. Dynamic bandwidth allocation involves channel hot-plug/unplug and page redistribution. Channel hot-plug/unplug takes only a few microseconds and is negligible; redistribution dominates the cost and depends on the number of allocated pages and migration rate. To measure this overhead, we run a read-only workload on a VM with one DIMM channel using Intel MLC, then attach a CXL channel, and later reclaim it (Figure 20). The workload uses 10 GB of memory and initially reaches 38 GB/s. After adding a CXL channel, bandwidth rises to 68 GB/s over 2.2 seconds, corresponding to a 1.82 GB/s migration rate while migrating about 40% of pages. Reclaiming the channel completes in 1.1 seconds because RAMRYDER performs lazy migration on allocation but immediate migration on reclamation. In both operations, latency remains smooth without noticeable spikes, indicating that redistribution incurs low overhead despite lasting a few seconds.

Sensitivity to highly dynamic demand. We evaluate RAMRYDER with a highly dynamic workload and compare it with an ideal baseline that over-provisions all channels. We observe two expected differences from this over-provisioned baseline (Figure 21). First, RAMRYDER may miss extremely short bursts and cannot provide instantaneous bandwidth for such transient spikes. Second, for other bandwidth changes, RAMRYDER takes time to increase bandwidth and therefore slightly lags behind the ideal baseline. These gaps come from two sources. First, RAMRYDER needs at least 1 second to collect performance-counter information, so very short bursts can finish before the next decision point. Second, increasing bandwidth requires page redistribution, so newly attached channels contribute bandwidth gradually rather than immediately. Overall, RAMRYDER transitions smoothly and catches up to sustained demand increases, making it more suitable for VMs with gradually increasing bandwidth demand.

7 Discussion

Our prototype and evaluation focus on the feasibility of elastic capacity and bandwidth allocation by managing memory channels in software. While the results are encouraging, we acknowledge several limitations. This section discusses approaches to address them and describes target scenarios.

Guest kernel modifications. RAMRYDER requires guest kernel modifications, which complicates deployment in the cloud. This design, however, could enable finer-grained isolation: the guest kernel can isolate memory channels across co-located jobs within a VM, while these jobs remain opaque to the host. We therefore expect memory channel abstractions to become a standard OS interface for both guest and host kernels, eliminating the need for guest kernel modifications.

Reserving bandwidth beforehand. RAMRYDER initially allocates DIMM channels based on each VM's memory bandwidth demand estimated from its proportion of allocated vCPUs, following the same policy used for local storage in the cloud [4, 18]. If this estimate deviates significantly from the VM's actual demand, RAMRYDER can also reallocate DIMM channels dynamically to avoid bandwidth shortages and resource waste.

Efficiency of dynamic bandwidth allocation. RAMRYDER requires several seconds to redistribute pages after channels are hot-plugged. This dynamic bandwidth allocation capability is suitable for VMs with gradually increasing bandwidth demand. For future optimization, RAMRYDER could leverage learning-based policies to predict bandwidth demand and trigger channel hot-plugging and page redistribution in advance.

Channel sharing for small VMs. Modern processors support up to 12 DIMM channels per socket, but this granularity remains too coarse for small VMs whose demand is below a full channel. Thus, RAMRYDER can place multiple small VMs on the same channel while isolating them from large VMs with many vCPUs, a major source of interference (§6.1). This also fits hardware throttling, which is effective mainly when the bandwidth limit is set very low [84, 115], making it a potential mechanism to control small VMs sharing a channel.

Granularity of CXL channels. The CXL memory devices in our testbed use a single DDR5 channel; thus, the entire device can be reserved as DAX to represent one channel. Future PCIe 6.0/7.0 CXL devices will require multiple DDR channels for higher bandwidth [102], but their internal interleaving policies remain unclear. Ideally, vendors should expose these policies as configurable options, allowing the OS to control the mapping from physical memory to internal DDR channels.

Applicability to multi-host CXL memory pools. RAMRYDER is currently designed and implemented for a single host server, but can be extended to a CXL memory pool shared by multiple hosts. In such scenarios, one host can act as the master node running the RAMRYDER resource manager, while others communicate with it via the RAMRYDER RPC interface (§5.2) to request memory capacity and bandwidth.

8 Related Work

Memory interference optimization. Prior work reduces memory interference via architectural techniques such as LLC partitioning [43, 57, 76, 87, 91, 92] and channel partitioning [73, 82, 116], which require hardware changes. System-level approaches include Canvas [105] for far-memory isolation and DRAM mapping aliases [47] for OS-guided partitioning or interleaving. RAMRYDER is inspired by both architectural and system-level designs to physically isolate channels across VMs to reduce interference.

Memory bandwidth allocation. Existing bandwidth allocation relies on indirect controls, including throttling between L2 and LLC [37, 55, 110, 115], CPU-level mechanisms [52, 84, 115], LLC partitioning with throttling [26, 83, 118], and CPU scheduling [39, 74]. These techniques are indirect, often workload-specific, and may compromise other metrics. RAMRYDER allocates bandwidth directly by allocating channels.

Memory bandwidth extension with CXL. Recent work explores using CXL devices to extend memory bandwidth. Many studies quantify the bandwidth benefits of CXL and examine page interleaving across memory tiers [27, 56, 71, 93, 100, 101, 106, 107]. Other efforts leverage CXL bandwidth for databases [49], PCIe device pooling [120], and LLM inference [63, 111, 113]. RAMRYDER extends this direction by enabling independent bandwidth allocation on CXL devices.

Memory capacity extension with CXL. CXL-based capacity extension spans memory tiering within a single server [1, 36, 64, 65, 69, 72, 77, 88, 94, 98, 99, 103, 108, 109, 112, 119] and memory disaggregation across multiple servers via CXL switches [3, 24, 41, 42, 46, 50, 51, 70, 114, 117]. RAMRYDER focuses on the single-server setting and can further extend to switch-based multi-host settings.

9 Conclusion

Memory bandwidth is severely underutilized in cloud data centers due to spatial and temporal memory over-provisioning, which stems from inflexible memory allocation and the lack of bandwidth allocation and performance isolation. This paper proposes RAMRYDER, a software-defined elastic memory system for cloud VMs. RAMRYDER takes the role of hardware and directly manages memory channel allocation and mapping in software to allocate bandwidth and capacity independently. This also enables performance isolation and allows resource allocation to change dynamically at runtime. RAMRYDER therefore improves both memory capacity and bandwidth utilization while guaranteeing performance QoS.

Acknowledgments

We thank our shepherd and the OSDI'26 anonymous reviewers for their insightful feedback, and Fan Ni for early discussions and suggestions. This work was supported in part by ACE, one of the seven centers in JUMP 2.0, a Semiconductor Research Corporation (SRC) program sponsored by DARPA.

References

[1] Neha Agarwal and Thomas F Wenisch. Thermostat: Application-transparent page management for two-tiered main memory. In *Proceedings of the 22nd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2017.

[2] Marcos K Aguilera, Nadav Amit, Irina Calciu, Xavier Deguillard, Jayneel Gandhi, Pratap Subrahmanyam, Lalith Suresh, Kiran Tati, Rajesh Venkatasubramanian, and Michael Wei. Remote memory in the age of fast networks. In *Proceedings of the 8th Symposium on Cloud Computing (SoCC)*, 2017.

[3] Minseon Ahn, Thomas Willhalm, Norman May, Donghun Lee, Suprasad Mutalik Desai, Daniel Booss, Jungmin Kim, Navneet Singh, Daniel Ritter, and Oliver Rebholz. An examination of CXL memory use cases for in-memory database management systems using sap hana. *Proceedings of the VLDB Endowment*, 2024.

[4] Alibaba. Block Storage performance, 2025. https://www.alibabacloud.com/help/en/ecs/user-guide/block-storage-performance.

[5] Alibaba. c9ae, compute-optimized instance family, 2025. https://www.alibabacloud.com/help/en/ecs/user-guide/overview-of-instance-families#c9ae.

[6] Alibaba. Cluster Trace Program, 2025. https://github.com/alibaba/clusterdata.

[7] Alibaba. What are the benefits of burstable instances, 2025. https://www.alibabacloud.com/help/en/ecs/user-guide/benefits-1.

[8] Emmanuel Amaro, Christopher Branner-Augmon, Zhihong Luo, Amy Ousterhout, Marcos K Aguilera, Aurojit Panda, Sylvia Ratnasamy, and Scott Shenker. Can far memory improve job throughput? In *Proceedings of the 15th European Conference on Computer Systems (EuroSys)*, 2020.

[9] Amazon. Burstable performance instances, 2025. https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/burstable-performance-instances.html.

[10] Amazon. Compute optimized Amazon EC2 instance types, 2025. https://aws.amazon.com/ec2/instance-types/compute-optimized/.

[11] Amazon. Compute optimized Amazon EC2 instance types, 2025. https://aws.amazon.com/ec2/instance-types/compute-optimized/.

[12] Amazon. EC2 R8g Instances, 2025. https://aws.amazon.com/ec2/instance-types/r8g/.

[13] Amazon. Elastic Block Store, 2025. https://aws.amazon.com/ebs/.

[14] Amazon. Elastic Fabric Adapter, 2025. https://aws.amazon.com/hpc/efa/.

[15] Amazon. Elastic File System, 2025. https://aws.amazon.com/efs/.

[16] Amazon. General purpose Amazon EC2 instance types, 2025. https://aws.amazon.com/ec2/instance-types/general-purpose/.

[17] Amazon. Memory optimized Amazon EC2 instance types, 2025. https://aws.amazon.com/ec2/instance-types/memory-optimized/.

[18] Amazon. Specifications for Amazon EC2 storage optimized instances, 2025. https://docs.aws.amazon.com/ec2/latest/instancetypes/so.html.

[19] AMD. Memory Population Guidelines for AMD EPYC 7003 Series Processors, 2021. https://www.amd.com/content/dam/amd/en/documents/epyc-technical-docs/other/56873_0_80_PUB.pdf.

[20] AMD. Platform Quality of Service (QoS)., 2022. https://www.amd.com/content/dam/amd/en/documents/processor-tech-docs/other/56375_1_03_PUB.pdf.

[21] AMD. 5th Generation AMD EPYC Processors, 2025. https://www.amd.com/en/products/processors/server/epyc/9005-series.html.

[22] AWS. Side-channel protections in the broader EC2 service, 2025. https://docs.aws.amazon.com/pdfs/whitepapers/latest/security-design-of-aws-nitro-system/security-design-of-aws-nitro-system.pdf#side-channel-protections-in-the-broader-ec2-service.

[23] Scott Beamer, Krste Asanović, and David Patterson. The GAP benchmark suite. *arXiv preprint arXiv:1508.03619*, 2015.

[24] Daniel S Berger, Yuhong Zhong, Fiodar Kazhamiaka, Pantea Zardoshti, Shuwei Teng, Mark D Hill, and Rodrigo Fonseca. Octopus: Scalable low-cost CXL memory pooling. *arXiv preprint arXiv:2501.09020*, 2025.

[25] Irina Calciu, M Talha Imran, Ivan Puddu, Sanidhya Kashyap, Hasan Al Maruf, Onur Mutlu, and Aasheesh Kolli. Rethinking software runtimes for disaggregated memory. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2021.

[26] Shuang Chen, Christina Delimitrou, and José F Martínez. Parties: QoS-aware resource partitioning for multiple interactive services. In *Proceedings of the 24th International Conference on Architectural*

Support for Programming Languages and Operating Systems (ASPLOS), 2019.

[27] Albert Cho, Anish Saxena, Moinuddin Qureshi, and Alexandros Daglis. COAXIAL: A CXL-centric memory system for scalable servers. In *Proceedings of the 2024 International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2024.

[28] Cisco. UCS Server BIOS Tokens in Intersight Managed Mode, 2025. https://www.cisco.com/c/en/us/td/docs/unified_computing/ucs/Intersight/IMM_BIOS_Tokens_Guide/b_IMM_Server_BIOS_Tokens_Guide/m-memory.html.

[29] Alibaba Cloud. Elastic RDMA, 2025. https://www.alibabacloud.com/help/en/ecs/user-guide/elastic-rdma-erdma/.

[30] Alibaba Cloud. Overview of Block Storage, 2025. https://www.alibabacloud.com/help/en/ecs/user-guide/elastic-block-storage-devices.

[31] Google Cloud. High-performance block storage for any use case, 2025. https://cloud.google.com/products/block-storage?hl=en.

[32] CXL community. CXL Memory Resource Kit (CMRK), 2025. https://github.com/cxl-reskit/cxl-reskit.

[33] Brian F Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud computing (SoCC)*, 2010.

[34] Debendra Das Sharma, Robert Blankenship, and Daniel Berger. An introduction to the compute express link (CXL) interconnect. *ACM Computing Surveys*, 2024.

[35] QEMU Project Developers. QEMU Guest Agent (QGA), 2025. https://qemu-project.gitlab.io/qemu/interop/qemu-ga.html.

[36] Padmapriya Duraisamy, Wei Xu, Scott Hare, Ravi Rajwar, David Culler, Zhiyi Xu, Jianing Fan, Christopher Kennelly, Bill McCloskey, Danijela Mijailovic, et al. Towards an adaptable systems architecture for memory tiering at warehouse-scale. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2023.

[37] Giorgio Farina, Gautam Gala, Marcello Cinque, and Gerhard Fohler. Assessing Intel's memory bandwidth allocation for resource limitation in real-time systems. In *Proceedings of the 25th IEEE International Symposium On Real-Time Distributed Computing (ISORC)*, 2022.

[38] Alessandro Fogli, Bo Zhao, Peter Pietzuch, and Jana Giceva. CHARM: Chiplet Heterogeneity-Aware Runtime Mapping System. In *Proceedings of the 21st European Conference on Computer Systems (EuroSys)*, 2026.

[39] Joshua Fried, Zhenyuan Ruan, Amy Ousterhout, and Adam Belay. Caladan: Mitigating interference at microsecond timescales. In *Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2020.

[40] Alexander Fuerst, Stanko Novaković, Íñigo Goiri, Gohar Irfan Chaudhry, Prateek Sharma, Kapil Arya, Kevin Broas, Eugene Bak, Mehmet Iyigun, and Ricardo Bianchini. Memory-harvesting VMs in cloud platforms. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2022.

[41] Donghyun Gouk, Miryeong Kwon, Hanyeoreum Bae, Sangwon Lee, and Myoungsoo Jung. Memory pooling with CXL. *IEEE Micro*, 2023.

[42] Donghyun Gouk, Sangwon Lee, Miryeong Kwon, and Myoungsoo Jung. Direct access, High-Performance memory disaggregation with DirectCXL. In *Proceedings of the 2022 USENIX Annual Technical Conference (ATC)*, 2022.

[43] Sriram Govindan, Jie Liu, Aman Kansal, and Anand Sivasubramaniam. Cuanta: quantifying effects of shared on-chip resource interference for consolidated virtual machines. In *Proceedings of the 2nd ACM Symposium on Cloud Computing (SoCC)*, 2011.

[44] Juncheng Gu, Youngmoon Lee, Yiwen Zhang, Mosharaf Chowdhury, and Kang G Shin. Efficient memory disaggregation with Infiniswap. In *Proceedings of the 14th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2017.

[45] Zhiyuan Guo, Yizhou Shan, Xuhao Luo, Yutong Huang, and Yiying Zhang. Clio: A hardware-software co-designed disaggregated memory system. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2022.

[46] Minho Ha, Junhee Ryu, Jungmin Choi, Kwangjin Ko, Sunwoong Kim, Sungwoo Hyun, Donguk Moon, Byungil Koh, Hokyoon Lee, Myoungseo Kim, et al. Dynamic capacity service for improving CXL pooled memory efficiency. *IEEE Micro*, 2023.

[47] Marius Hillenbrand, Mathias Gottschlag, Jens Kehne, and Frank Bellosa. Multiple physical mappings: Dynamic DRAM channel sharing and partitioning. In *Proceedings of the 8th Asia-Pacific Workshop on Systems (APSys)*, 2017.

[48] HPE. UEFI System Utilities User Guide for HPE ProLiant Gen10, Proliant Gen10 Plus Servers, and HPE Synergy, 2021. https://itpfdoc.hitachi.co.jp/manuals/ha8000v/hard/Gen10/UEFI/30-293E3364-104_en.pdf.

[49] Wentao Huang, Mo Sha, Mian Lu, Yuqiang Chen, Bingsheng He, and Kian-Lee Tan. Bandwidth Expansion via CXL: A Pathway to Accelerating In-Memory Analytical Processing. *Proceedings of the VLDB Endowment. ISSN*, 2024.

[50] Yibo Huang, Haowei Chen, Newton Ni, Vijay Chidambaram, Dixin Tang, Emmett Witchel, Zhiting Zhu, and Zhipeng Jia. Tigon: A distributed database for a CXL pod. In *Proceedings of the 19th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2025.

[51] Yibo Huang, Newton Ni, Vijay Chidambaram, Emmett Witchel, and Dixin Tang. Pasha: An efficient, scalable database architecture for CXL pods. In *Proceedings of the Conference on Innovative Data Systems Research (CIDR)*, 2025.

[52] Satoshi Imamura and Eiji Yoshida. FairHym: Improving inter-process fairness on hybrid memory systems. In *Proceedings of the 9th Non-Volatile Memory Systems and Applications Symposium (NVMSA)*. IEEE, 2020.

[53] Intel. Memory Latency Checker (MLC)., 2024. https://www.intel.com/content/www/us/en/developer/articles/tool/intelr-memory-latency-checker.html.

[54] Intel. Memory Bandwidth Allocation (MBA), 2025. https://www.intel.com/content/www/us/en/developer/articles/technical/introduction-to-memory-bandwidth-allocation.html.

[55] Ravi Iyer, Li Zhao, Fei Guo, Ramesh Illikkal, Srihari Makineni, Don Newell, Yan Solihin, Lisa Hsu, and Steve Reinhardt. QoS policies and architecture for cache/memory in CMP platforms. *ACM SIGMETRICS Performance Evaluation Review*, 2007.

[56] Divya Kiran Kadiyala and Alexandros Daglis. Pushing the Memory Bandwidth Wall with CXL-enabled Idle I/O Bandwidth Harvesting. *arXiv preprint arXiv:2511.12349*, 2025.

[57] Harshad Kasture and Daniel Sanchez. Ubik: Efficient cache sharing with strict QoS for latency-critical workloads. In *Proceedings of the 19th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2014.

[58] The kernel development community. ACPI Tables, 2025. https://docs.kernel.org/driver-api/cxl/platform/acpi.html.

[59] The kernel development community. CXL Multi-Level Interleave, 2025. https://docs.kernel.org/driver-api/cxl/linux/example-configurations/multi-interleave.html.

[60] The kernel development community. Physical Memory Model, 2025. https://docs.kernel.org/mm/memory-model.html.

[61] Michael Kerrisk. perf_event_open - set up performance monitoring, 2025. https://man7.org/linux/man-pages/man2/perf_event_open.2.html.

[62] Anurag Khandelwal, Yupeng Tang, Rachit Agarwal, Aditya Akella, and Ion Stoica. Jiffy: Elastic far-memory for stateful serverless analytics. In *Proceedings of the 17th European Conference on Computer Systems (EuroSys)*, 2022.

[63] Dowon Kim, MinJae Lee, Janghyeon Kim, Hyuck-Sung Kwon, Hyeonggyu Jeong, Sang-Soo Park, Minyong Yoon, Si-Dong Roh, Yongsuk Kwon, Jinin So, et al. Scalable Processing-Near-Memory for 1M-Token LLM Inference: CXL-Enabled KV-Cache Management Beyond GPU Limits. *arXiv preprint arXiv:2511.00321*, 2025.

[64] Jonghyeon Kim, Wonkyo Choe, and Jeongseob Ahn. Exploring the design space of page management for multi-tiered memory systems. In *Proceedings of the 2021 USENIX Annual Technical Conference (ATC)*, 2021.

[65] Andres Lagar-Cavilla, Junwhan Ahn, Suleiman Souhlal, Neha Agarwal, Radoslaw Burny, Shakeel Butt, Jichuan Chang, Ashwin Chaugule, Nan Deng, Junaid Shahid, et al. Software-defined far memory in warehouse-scale computers. In *Proceedings of the 24th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2019.

[66] Seung-seob Lee, Jachym Putta, Ziming Mao, and Anurag Khandelwal. Spirit: Fair allocation of interdependent resources in remote memory systems. In *Proceedings of the ACM SIGOPS 31st Symposium on Operating Systems Principles (SOSP)*, 2025.

[67] Seung-seob Lee, Jachym Putta, Ziming Mao, and Anurag Khandelwal. Spirit: Fair Allocation of Interdependent Resources in Remote Memory Systems.

In *Proceedings of the ACM SIGOPS 31st Symposium on Operating Systems Principles (SOSP)*, 2025.

[68] Seung-seob Lee, Yanpeng Yu, Yupeng Tang, Anurag Khandelwal, Lin Zhong, and Abhishek Bhattacharjee. MIND: In-network memory management for disaggregated data centers. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles (SOSP)*, 2021.

[69] Taehyung Lee, Sumit Kumar Monga, Changwoo Min, and Young Ik Eom. Memtis: Efficient memory tiering with dynamic page classification and page size determination. In *Proceedings of the ACM SIGOPS 29th Symposium on Operating Systems Principles (SOSP)*, 2023.

[70] Huaicheng Li, Daniel S. Berger, Lisa Hsu, Daniel Ernst, Pantea Zardoshti, Stanko Novakovic, Monish Shah, Samir Rajadnya, Scott Lee, Ishwar Agarwal, et al. Pond: CXL-based memory pooling systems for cloud platforms. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2023.

[71] Jinshu Liu, Hamid Hadian, Yuyue Wang, Daniel S Berger, Marie Nguyen, Xun Jian, Sam H Noh, and Huaicheng Li. Systematic CXL memory characterization and performance analysis at scale. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2025.

[72] Jinshu Liu, Hamid Hadian, Hanchen Xu, and Huaicheng Li. Tiered Memory Management Beyond Hotness. In *Proceedings of the 19th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2025.

[73] Lei Liu, Zehan Cui, Yong Li, Yungang Bao, Mingyu Chen, and Chengyong Wu. BPM/BPM+ Software-based dynamic memory partitioning mechanisms for mitigating DRAM bank-/channel-level interferences in multicore systems. *ACM Transactions on Architecture and Code Optimization (TACO)*, 2014.

[74] David Lo, Liqun Cheng, Rama Govindaraju, Parthasarathy Ranganathan, and Christos Kozyrakis. Heracles: Improving resource efficiency at scale. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture (ISCA)*, 2015.

[75] Kevin Loughlin, Jonah Rosenblum, Stefan Saroiu, Alec Wolman, Dimitrios Skarlatos, and Baris Kasikci. Siloz: Leveraging DRAM isolation domains to prevent inter-VM rowhammer. In *Proceedings of the ACM SIGOPS 29th Symposium on Operating Systems Principles (SOSP)*, 2023.

[76] Jason Mars, Lingjia Tang, Kevin Skadron, Mary Lou Soffa, and Robert Hundt. Increasing utilization in modern warehouse-scale computers using bubble-up. *IEEE Micro*, 2012.

[77] Hasan Al Maruf, Hao Wang, Abhishek Dhanotia, Johannes Weiner, Niket Agarwal, Pallab Bhattacharya, Chris Petersen, Mosharaf Chowdhury, Shobhit Kanaujia, and Prakash Chauhan. TPP: Transparent page placement for CXL-enabled tiered-memory. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2023.

[78] John D. McCalpin. STREAM: Sustainable memory bandwidth in high performance computers., 2006. http://www.cs.virginia.edu/stream/.

[79] Memcached. A distributed memory object caching system., 2018. https://memcached.org/.

[80] Microsoft. Azure Elastic SAN, 2025. https://azure.microsoft.com/en-us/products/storage/elastic-san/.

[81] Microsoft. B family general purpose VM size series, 2025. https://learn.microsoft.com/en-us/azure/virtual-machines/sizes/general-purpose/b-family.

[82] Sai Prashanth Muralidhara, Lavanya Subramanian, Onur Mutlu, Mahmut Kandemir, and Thomas Moscibroda. Reducing memory interference in multicore systems via application-aware memory channel partitioning. In *Proceedings of the 44th annual IEEE/ACM international symposium on microarchitecture (MICRO)*, 2011.

[83] Jinsu Park, Seongbeom Park, and Woongki Baek. CoPart: Coordinated partitioning of last-level cache and memory bandwidth for fairness-aware workload consolidation on commodity servers. In *Proceedings of the 14th European Conference on Computer Systems (EuroSys)*, 2019.

[84] Jinsu Park, Seongbeom Park, Myeonggyun Han, Jihoon Hyun, and Woongki Baek. HyPart: A hybrid technique for practical memory bandwidth partitioning on commodity servers. In *Proceedings of the 27th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2018.

[85] Yifan Qiao, Zhenyuan Ruan, Haoran Ma, Adam Belay, Miryung Kim, and Harry Xu. Harvesting idle memory for application-managed soft state with Midas. In *Proceedings of the 21st USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2024.

[86] Yifan Qiao, Chenxi Wang, Zhenyuan Ruan, Adam Belay, Qingda Lu, Yiying Zhang, Miryung Kim, and Guo-

qing Harry Xu. Hermit: Low-latency, high-throughput, and transparent remote memory via feedback-directed asynchrony. In *Proceedings of the 20th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2023.

[87] Moinuddin K Qureshi and Yale N Patt. Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches. In *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2006.

[88] Amanda Raybuck, Tim Stamler, Wei Zhang, Mattan Erez, and Simon Peter. HeMem: Scalable tiered memory management for big data applications and real nvm. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles (SOSP)*, 2021.

[89] Redis. The preferred, fastest, and most feature-rich cache, data structure server, and document and vector query engine., 2025. https://memcached.org/.

[90] Zhenyuan Ruan, Malte Schwarzkopf, Marcos K Aguilera, and Adam Belay. AIFM: High-performance, application-integrated far memory. In *Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2020.

[91] Daniel Sanchez and Christos Kozyrakis. The ZCache: Decoupling ways and associativity. In *Proceedings of the 43rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2010.

[92] Daniel Sanchez and Christos Kozyrakis. Vantage: Scalable and efficient fine-grain cache partitioning. In *Proceedings of the 38th annual international symposium on Computer architecture (ISCA)*, 2011.

[93] Rohit Sehgal, Vishal Tanna, Vinicius Petrucci, and Anil Godbole. Optimizing system memory bandwidth with Micron CXL memory expansion modules on Intel Xeon 6 processors. *arXiv preprint arXiv:2412.12491*, 2024.

[94] Sai Sha, Chuandong Li, Yingwei Luo, Xiaolin Wang, and Zhenlin Wang. vTMM: Tiered memory management for virtual machines. In *Proceedings of the 18th European Conference on Computer Systems (EuroSys)*, pages 283–297, 2023.

[95] Yizhou Shan, Yutong Huang, Yilun Chen, and Yiying Zhang. LegoOS: A disseminated, distributed OS for hardware resource disaggregation. In *Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2018.

[96] Debendra Das Sharma, Mahesh Wagh, Vincent Hache, and Mahesh Natu. Compute Express Link (CXL): An Open Industry Standard for Composable Computing, 2023. https://computeexpresslink.org/wp-content/uploads/2023/12/CXL_FMS-2023-Tutorial_FINAL.pdf.

[97] Parul Sohal, Michael Bechtel, Renato Mancuso, Heechul Yun, and Orran Krieger. A closer look at Intel resource director technology (RDT). In *Proceedings of the 30th International Conference on Real-Time Networks and Systems (RTNS)*, 2022.

[98] Kevin Song, Jiacheng Yang, Zixuan Wang, Jishen Zhao, Sihang Liu, and Gennady Pekhimenko. HybridTier: an adaptive and lightweight CXL-memory tiering system. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2025.

[99] Yan Sun, Jongyul Kim, Zeduo Yu, Jiyuan Zhang, Siyuan Chai, Michael Jaemin Kim, Hwayong Nam, Jaehyun Park, Eojin Na, Yifan Yuan, et al. M5: Mastering Page Migration and Memory Management for CXL-based Tiered Memory Systems. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2025.

[100] Yan Sun, Yifan Yuan, Zeduo Yu, Reese Kuper, Chihun Song, Jinghan Huang, Houxiang Ji, Siddharth Agarwal, Jiaqi Lou, Ipoom Jeong, et al. Demystifying CXL memory with genuine CXL-ready systems and devices. In *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2023.

[101] Yupeng Tang, Ping Zhou, Wenhui Zhang, Henry Hu, Qirui Yang, Hao Xiang, Tongping Liu, Jiaxin Shan, Ruoyun Huang, Cheng Zhao, et al. Exploring performance and cost optimization with ASIC-based CXL memory. In *Proceedings of the 19th European Conference on Computer Systems (EuroSys)*, 2024.

[102] TrendForce. Samsung Unveils CXL Roadmap: CMM-D 2.0 Samples Ready, 3.1 Targeted for Year-End, 2025. https://www.trendforce.com/news/2025/10/17/news-samsung-unveils-cxl-roadmap-cmm-d-2-0-samples-ready-3-1-targeted-for-year-end/.

[103] Midhul Vuppalapati and Rachit Agarwal. Tiered Memory Management: Access Latency is the Key! In *Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles (SOSP)*, 2024.

[104] Carl A Waldspurger. Memory resource management in VMware ESX server. *ACM SIGOPS Operating Systems Review*, 2002.

[105] Chenxi Wang, Yifan Qiao, Haoran Ma, Shi Liu, Wenguang Chen, Ravi Netravali, Miryung Kim, and Guoqing Harry Xu. Canvas: Isolated and adaptive swapping for multi-applications on remote memory. In

Proceedings of the 20th USENIX Symposium on Networked Systems Design and Implementation (NSDI), 2023.

[106] Xi Wang, Jie Liu, Jianbo Wu, Shuangyan Yang, Jie Ren, Bhanu Shankar, and Dong Li. Performance Characterization of CXL Memory and Its Use Cases. In *Proceedings of the 2025 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2025.

[107] Zixuan Wang, Suyash Mahar, Luyi Li, Jangseon Park, Jinpyo Kim, Theodore Michailidis, Yue Pan, Tajana Rosing, Dean Tullsen, Steven Swanson, et al. The Hitchhiker's Guide to Programming and Optimizing CXL-Based Heterogeneous Systems. *arXiv preprint arXiv:2411.02814*, 2024.

[108] Johannes Weiner, Niket Agarwal, Dan Schatzberg, Leon Yang, Hao Wang, Blaise Sanouillet, Bikash Sharma, Tejun Heo, Mayank Jain, Chunqiang Tang, et al. TMO: Transparent memory offloading in datacenters. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2022.

[109] Lingfeng Xiang, Zhen Lin, Weishu Deng, Hui Lu, Jia Rao, Yifan Yuan, and Ren Wang. Nomad:Non-Exclusive memory tiering via transactional page migration. In *Proceedings of the 18th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2024.

[110] Yaocheng Xiang, Chencheng Ye, Xiaolin Wang, Yingwei Luo, and Zhenlin Wang. EMBA: Efficient memory bandwidth allocation to improve performance on Intel commodity processor. In *Proceedings of the 48th International Conference on Parallel Processing (ICPP)*, 2019.

[111] Rui Xie, Asad Ul Haq, Linsen Ma, Yunhua Fang, Zirak Burzin Engineer, Liu Liu, and Tong Zhang. Amplifying Effective CXL Memory Bandwidth for LLM Inference via Transparent Near-Data Processing. *arXiv preprint arXiv:2509.03377*, 2025.

[112] Zi Yan, Daniel Lustig, David Nellans, and Abhishek Bhattacharjee. Nimble page management for tiered memory systems. In *Proceedings of the 24th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2019.

[113] Xinjun Yang, Qingda Hu, Junru Li, Feifei Li, Yuqi Zhou, Yicong Zhu, Qiuru Lin, Jian Dai, Yang Kong, Jiayu Zhang, et al. Beluga: A CXL-Based Memory Architecture for Scalable and Efficient LLM KVCache Management. *arXiv preprint arXiv:2511.20172*, 2025.

[114] Xinjun Yang, Yingqiang Zhang, Hao Chen, Feifei Li, Gerry Fan, Yang Kong, Bo Wang, Jing Fang, Yuhui Wang, Tao Huang, et al. Unlocking the Potential of CXL for Disaggregated Memory in Cloud-Native Databases. In *Companion of the 2025 International Conference on Management of Data*, 2025.

[115] Jifei Yi, Benchao Dong, Mingkai Dong, Ruizhe Tong, and Haibo Chen. MT^2: Memory Bandwidth Regulation on Hybrid NVM/DRAM Platforms. In *Proceedings of the 20th USENIX Conference on File and Storage Technologies (FAST)*, 2022.

[116] Heechul Yun, Renato Mancuso, Zheng-Pei Wu, and Rodolfo Pellizzoni. PALLOC: DRAM bank-aware memory allocator for performance isolation on multicore platforms. In *Proceedings of the 19th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE, 2014.

[117] Mingxing Zhang, Teng Ma, Jinqi Hua, Zheng Liu, Kang Chen, Ning Ding, Fan Du, Jinlei Jiang, Tao Ma, and Yongwei Wu. Partial failure resilient memory management system for (CXL-based) distributed shared memory. In *Proceedings of the ACM SIGOPS 29th Symposium on Operating Systems Principles (SOSP)*, 2023.

[118] Ying Zhang, Jian Chen, Xiaowei Jiang, Qiang Liu, Ian M Steiner, Andrew J Herdrich, Kevin Shu, Ripan Das, Long Cui, and Litrin Jiang. LIBRA: Clearing the cloud through dynamic memory bandwidth management. In *Proceedings of the 2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, 2021.

[119] Yuhong Zhong, Daniel S Berger, Carl Waldspurger, Ryan Wee, Ishwar Agarwal, Rajat Agarwal, Frank Hady, Karthik Kumar, Mark D Hill, Mosharaf Chowdhury, et al. Managing Memory Tiers with CXL in Virtualized Environments. In *Proceedings of the 18th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2024.

[120] Yuhong Zhong, Daniel S Berger, Pantea Zardoshti, Enrique Saurez, Jacob Nelson, Dan RK Ports, Antonis Psistakis, Joshua Fried, and Asaf Cidon. Oasis: Pooling PCIe devices over CXL to boost utilization. In *Proceedings of the ACM SIGOPS 31st Symposium on Operating Systems Principles (SOSP)*, 2025.

MAC: Metadata Acceleration for Sustainable Performance in Big-Data Systems with CXL DRAM

Dusol Lee[1], Yan Sun[2], Houxiang Ji[2], Vinit Gupta[2], Austin Antony Cruz[2], Inhyuk Choi[1], Nam Sung Kim[2], Jihong Kim[1]

[1]*Seoul National University* [2]*University of Illinois Urbana-Champaign*

Abstract

Compute Express Link (CXL) DRAM has emerged as a promising solution to address the capacity constraints of conventional systems using DDR DRAM. By decoupling memory expansion from the DDR interface generation imposed by the host CPU's memory controller, CXL DRAM allows cost-effective scaling of system memory capacity. However, as memory capacity grows, memory management metadata can become too large to fit entirely in DDR DRAM, necessitating that part or all of it to be placed in CXL DRAM. Moreover, since the OS views CXL DRAM as a CPU-less remote node, the host CPU manages metadata in CXL DRAM, thereby increasing metadata management latency. We find that this overhead significantly reduces memory reclamation efficiency and causes considerable increases in the tail latency of latency-sensitive applications. In this work, we investigate the performance impact of metadata placement in CXL DRAM and propose MAC, a near-memory processing (NMP) solution that accelerates memory-intensive components of metadata management directly within CXL DRAM to improve memory reclamation efficiency. Compared to conventional OS-based memory reclamation, MAC reduces application tail latency by up to 98%.

1 Introduction

CXL DRAM has emerged as a promising solution to address the memory capacity limitations of applications whose datasets exceed available memory, such as large-scale analytics and database workloads [58, 7, 61]. The CXL interface enables significantly larger memory capacity with fewer platform-level constraints compared to traditional DDR DIMMs. Although CXL DRAM has approximately 2.4× higher access latency than DDR DRAM [34, 80], it offers substantial scalability benefits. Servers can overcome the traditional 'memory capacity wall' and provision memory at a lower cost. Furthermore, by equipping servers with CXL DRAM, memory provisioning costs for the servers can be reduced by more than half compared to DDR DRAM-based systems with equivalent total capacity—for example, in configurations where the DDR-to-CXL DRAM ratio is 1:4 [9, 60, 59, 18].

However, such servers introduce new challenges in memory management. The operating system (OS) maintains a page descriptor [49] for each physical memory frame to keep track of the state of each page (where a "page" refers to a logical unit that is mapped to a physical frame in virtual memory systems), including metrics such as the recency of access and the reference count. The OS also uses the Xarray data structure [52] to index file-backed pages in the page cache [84]. The size of these metadata structures scales proportionally as a function of total system memory and file sizes, and may become too large to fit entirely in DDR DRAM of a server, which provides low latency but limited capacity under cost constraints. For example, in a server with a DDR DRAM to CXL DRAM capacity ratio of 1:4, the total size of memory management metadata can reach up to 24% of the DDR DRAM capacity. If the ratio increases to 1:8, this capacity overhead can approach 40%.

When these metadata structures spill over into slow CXL DRAM due to insufficient space in DDR DRAM, we uncover a deterioration in the rate of memory reclamation, which degrades application performance. Specifically, modern operating systems rely on a background memory reclaimer (*e.g.*, kswapd [38] in Linux) to free pages under memory pressure. However, when the background memory reclaimer has to access metadata in slow CXL DRAM, it cannot free pages quickly enough. This forces user applications to trigger a foreground memory reclaimer. Because the foreground memory reclaimer lies on the critical path of application I/O [68, 54, 82, 83], its accesses to slow CXL DRAM significantly increase the tail latency of applications.

To observe the impact on foreground reclamation on application tail latency, we conducted experiments where the latency of metadata access—specifically for Xarray and page descriptors—was modulated during both background and foreground memory reclamation. When metadata access latency increased by approximately 2.4×, we observed that the 99.99th-percentile (p99.99) tail latencies increased by about 2.7×, primarily due to a 6.5× increase in the number of direct reclamation events on the critical path of application

execution. These findings suggest that placing metadata in high-latency CXL DRAM can severely deteriorate application responsiveness—especially for latency-sensitive workloads that demand tail latency guarantees within just a few milliseconds (*e.g.*, 1–10 ms at p99.99 [75, 31, 78, 1, 90]).

To address these challenges, we propose MAC (Memory Reclamation Acceleration for CXL), a near-memory processing (NMP)-based memory reclamation acceleration solution designed for systems deploying large-capacity CXL DRAM in this paper. Specifically, we make the following key contributions.

Contribution-1: Uncovering the capacity and performance bottlenecks of kernel metadata in CXL memory systems. We uncover that metadata structures—such as page descriptors and Xarray nodes—can impose significant overhead on both memory capacity and application performance. We further show that their simple and repetitive access patterns make them well-suited for offloading to near-memory processing units within modern CXL devices in big-data systems.

Contribution-2: Co-designing OS and hardware for memory reclamation acceleration. MAC accelerates memory reclamation by offloading two main operations in the page reclaim path to an NMP accelerator within CXL DRAM: the traversal of victim page descriptors and Xarray walk. To enable efficient access, metadata is placed directly in CXL DRAM, which shares the same physical address space as the host CPU. MAC leverages the Linux kernel's direct address mapping mechanism [44] to allow the NMP accelerator to access metadata without the need for host CPU's involvement or additional address mapping tables, enabling fast metadata traversal. Furthermore, MAC places a host-accessible shared buffer inside the CXL DRAM region, enabling low-latency communication between the host CPU and the accelerator, thereby minimizing data transfer overhead.

For each offloading request, the CPU first compiles a batch of operations and writes the arguments to the shared memory buffer. The CPU then issues a MMIO [36] write to the NMP, upon which the NMP will fetch the arguments asynchronously with CXL.cache requests. Upon receiving the input arguments, the NMP processes the batch with multiple operation units. Each operation unit fetches the data with CXL.cache link, processes it with its internal state machine, and repeats until reaching the result. Finally, the NMP writes the returned data to the shared buffer and signal the CPU for completion.

Contribution-3: Prototyping MAC with a commercial CXL device for a demonstration. MAC was prototyped with an Intel Agilex 7 I-series FPGA [24] to accelerate requests offloaded from the host CPU. Since current commodity hardware does not support the FPGA-initiated fast cache invalidation feature (BIsnp[1]), which is needed for fast coherency between FPGA-modified data and the host CPU cache, we replace the BIsnp feature with a CXL.cache operation that performs coherent write to CXL memory. In this mode, memory requests are always sent to the host CPU's root complex for coherence processing and perform any cache invalidation if necessary. We anticipate that performance will remain similar once BIsnp is supported in future generations of CPUs.

[1] In the host-biased mode [5] supported by the current commodity

2 Background

2.1 CXL DRAM

Compute Express Link (CXL) [8] is a cache-coherent interconnect protocol built upon the PCI Express (PCIe) interface. CXL-attached memory, such as DDR DRAM, operates over this interface and leverages the physical and electrical characteristics of PCIe. Recent high-capacity, high-bandwidth CXL DRAM modules typically occupy a full PCIe ×16 slot and can support large memory capacities—for example, up to 4 TiB per device [77, 10, 76].

To communicate with such memory devices, the host CPU must support the CXL protocol stack [70]. Modern server-class processors, such as Intel 6th-generation Xeon Scalable Processor [25], provide native support for CXL 2.0 and allocate a portion of their PCIe lanes accordingly. For instance, a single-socket server offers up to 64 CXL lanes out of 136 total PCIe lanes, supporting data transfer rates up to 32 GT/s per lane [23].

One notable advantage of CXL DRAM modules is that they decouple memory technologies from the host CPU's integrated memory controller [80, 4, 72, 29]. Since the DRAM interface is internal to the CXL device itself, the memory type (*e.g.*, DDR4, DDR5) is no longer constrained by what the CPU natively supports. This abstraction enables greater flexibility in memory configurations and allows vendors to integrate cost-effective or higher-density DRAM technologies regardless of CPU generations.

Consequently, systems can provision larger memory pools at lower cost by relying more heavily on CXL DRAM, rather than expanding DDR DRAM through conventional DIMM slots. Due to these advantages, hyperscalers can reduce server memory costs by more than 50%, when using CXL DRAM based on recycled DIMMs from retired servers, compared to DIMM-only configurations for the same memory capacity [9, 60, 59, 18].

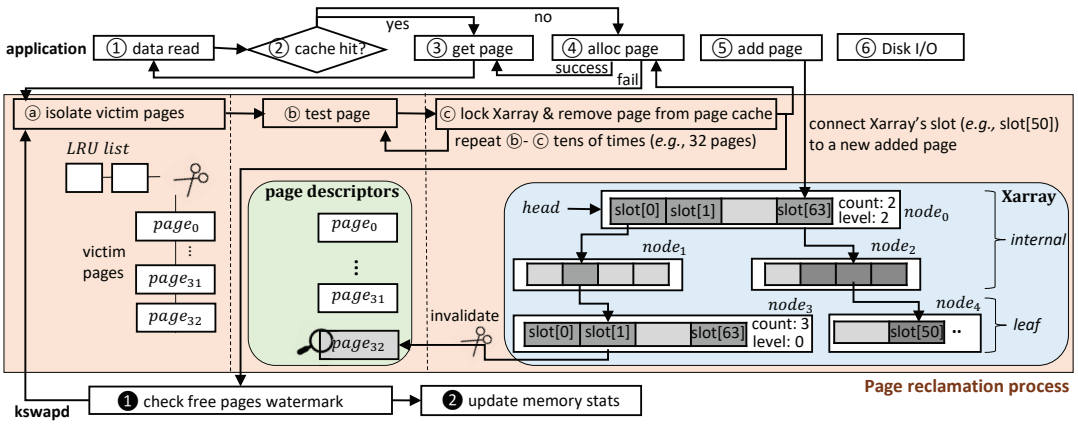


Figure 1: Page reclamation and metadata access.

2.2 Linux Memory Management

Modern Linux manages physical memory by dividing it into fixed-size page frames, typically 4 KiB in size. In Linux, the page cache subsystem associates each page frame with a corresponding 64-byte page descriptor [49], which maintains metadata such as access frequency, mapping information, and file-related properties. These descriptors are used by the Linux kernel to track page status during accesses and evictions. The total memory footprint of page descriptors is 1.6% of the system's physical memory, assuming 4 KiB pages and a 64-byte descriptor size, and must reside in system memory to ensure fast lookup and updates.

When an application performs file I/O operations with offset within a given file, the accessed file data are loaded into memory, and the kernel instantiates page descriptors for each 4 KB chunk of that file. To efficiently locate a page descriptor within the page cache using a file offset, Linux employs the Xarray data structure [52] as an index. Each file is associated with its own Xarray, and each Xarray node is 584 bytes, containing up to 64 entries that point to page descriptors. As the size of the file-backed memory working set grows, and as system memory scales up, the aggregate size of the Xarray nodes increases accordingly. These metadata structures are heavily accessed during memory reclamation.

When the host CPU runs data-intensive workloads, high memory demands can increase memory pressure. This causes the number of free pages to drop below the high-watermark threshold, prompting the Linux kernel to initiate page reclamation and free memory. To manage this process, the kernel spawns a kernel thread (kswapd) per NUMA node [48], with each thread bound to a dedicated CPU core to perform background reclamation. However, in scenarios where applications perform I/O operations across tens of CPU cores, the rate of memory allocation can outpace that of memory reclamation. Once free pages drop below the low-watermark threshold during execution of an application, the CPU core suspends its application process to carry out page reclamation, which is known as foreground reclamation [47].

Figure 1 illustrates how each type of metadata is accessed under both background and foreground reclamation scenarios. For example, when an application attempts to read file data (①), the kernel first checks whether the corresponding data page resides in the page cache (②). If the page exists (cache hit), it is returned directly to the application (③); otherwise (cache miss), a free page is allocated (④). If the system's free page count exceeds the low watermark, the newly allocated page is inserted into the page cache (⑤), and disk I/O is performed to fetch the data (⑥). Conversely, when the allocation step (④) observes the free page count below the low watermark, memory reclamation runs in the application thread context (foreground reclamation). First, candidate pages are isolated from the system's page list (*e.g.*, LRU list) (ⓐ). Next, each page descriptor is inspected to evaluate its eviction suitability—verifying attributes such as validity, clean/dirty state, recency of access, reference count status, and other relevant flags (ⓑ). The kernel then traverses the Xarray structure down to the leaf node to remove the corresponding page index within the file from the page cache (ⓒ)—for example, invalidating the pointer in slot[1] of $node_3$, as shown in the Figure 1. To invalidate the pointer, the kernel stores a special shadow value [57] into the Xarray slot[1]. This action invalidates the original pointer and means that the corresponding page is no longer present in the page cache. In a single reclamation cycle (ⓐ-ⓒ), this process is repeated tens to hundreds of times depending on memory pressure.

In contrast, background reclamation is handled by the kernel thread kswapd, which is triggered under memory pressure. This thread executes a loop: as long as the free page count remains below the high watermark (❶), it repeat-

FPGA, when the FPGA needs to modify the data in its attached CXL DRAM, it must send the memory request to the host CPU's root complex for coherence processing, relying on the host CPU to access the CXL DRAM. This involvement of the host CPU leads to sub-optimal performance.

edly performs the reclamation process (ⓐ-ⓒ) and maintains memory state (*e.g.*, managing LRU lists and free/used page statistics) (❷).

3 Impact of Deploying Large-Capacity CXL DRAM

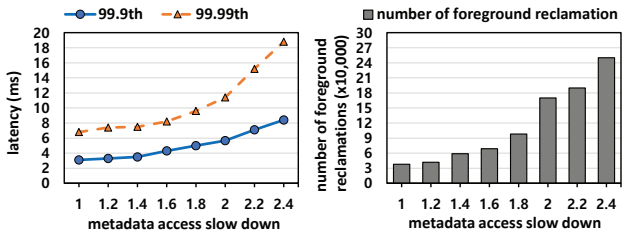
In this section, we examine the impact of deploying large-capacity CXL DRAM on system performance, focusing on the capacity and performance overheads of managing kernel metadata. We show that metadata placement in CXL DRAM can significantly affect both memory utilization and the efficiency of memory reclamation.

3.1 Overhead of Managing Metadata in CXL-Enabled Memory Systems

Capacity overhead. In large-memory systems, the metadata for memory management can occupy a substantial portion of the physical memory, making it expensive to keep entirely within the host DDR DRAM. Consider a workload accessing a single 1.8 TiB file on a system with 120 GiB memory. Accessing the entire file requires indexing up to 483 million pages (1.8 TiB / 4 KiB), which, given 64 entries per Xarray node, corresponds to a maximum of 7.5 million leaf nodes and an Xarray footprint of approximately 4.1 GiB (= 7.5 millions × 584 B), assuming sufficient memory to hold the entire Xarray. Furthermore, because each file maintains its own Xarray comprising both internal and leaf nodes, the total Xarray footprint grows with the number of files. Similarly, the page descriptor consumes around 2 GiB of memory space (= 120 GiB / 4 KiB × 64 B per descriptor).

For example, in our evaluation using the YCSB RocksDB [71] read-intensive workload [88] with a 1.8 TiB database and 120 GiB system memory comprised of 24 GiB DDR DRAM and real 96 GiB CXL DRAM, the Xarray metadata footprint reached 3.7 GiB. In total, metadata, including 2 GiB for page descriptors and 3.7 GiB for Xarray nodes, consumed approximately 5.7 GiB of system memory. Similarly, under a PostgreSQL [65] TPC-B workload [64] with a 1 TiB dataset, the Xarray metadata was 3.3 GiB and page descriptors accounted for 2 GiB, collectively occupying around 5.3 GiB of the 120 GiB memory.

The problem is, in environments where the capacity of DDR DRAM is relatively smaller than the CXL DRAM, such metadata is too large to be fully stored within CXL DRAM. If CXL DRAM capacity is four times that of the DDR DRAM capacity, the total size of metadata can consume 24% of the DDR DRAM capacity (= 2 GiB + 3.7 GiB) / (120 GiB / 5). As the proportion of CXL DRAM increases, the relative share of metadata within DDR DRAM also grows. For instance, if CXL DRAM is eight times larger than DDR DRAM, metadata alone can occupy approximately 40% of DDR DRAM capacity.



(a) (b)

Figure 2: Impact of metadata access slowdown on application performance: (a) tail latency of application and (b) foreground reclamation frequency when running RocksDB with YCSB, 10M Read Ops with 100 threads, 1.8 TiB DB, 120 GiB DDR DRAM, and 32 CPU cores.

To avoid excessive consumption of DDR DRAM by page descriptors when installing memory devices like CXL DRAM, recent Linux kernels support placing these descriptors on the device itself [42]. Additionally, when allocating Xarray nodes, the kernel employs a fallback mechanism [43]: if no DDR DRAM is available, the allocation automatically shifts to CXL DRAM, preventing contention with application data in DDR DRAM and potential Out-Of-Memory (OOM) events.

Performance cost. Placing metadata in CXL DRAM, nevertheless, can adversely impacts the performance of latency-sensitive applications. The page reclamation process traverses the Xarray and page descriptors repeatedly to remove tens or even hundreds of pages depending on the degree of memory pressure (as shown in the ⓑ-ⓒ of the Figure 1). Each time, the host CPU must access metadata residing in CXL DRAM, incurring the extra latency associated with remote memory accesses.

When the efficiency of metadata access by the Linux kernel's background memory reclaimer (kswapd) is reduced due to slow remote memory accesses, the memory reclamation rate declines. This, in turn, forces applications to perform foreground reclamation directly, which can increase application's tail latency considerably. To observe the effect of metadata access latency on application tail latency, we intentionally introduced delays in accessing metadata structures (specifically, the Xarray and page descriptors) during background reclamation. As shown in Figure 2(a), when metadata access latency increases by 2.4×, the 99.9th and 99.99th-percentile (p99.9 and p99.99) latencies increase by approximately 2.6× and 2.8×, respectively. The underlying reason, as shown in the Figure 2(b), is that the frequency of foreground reclamation, which lies on the critical path of the application I/O, increases by 6.5×. This demonstrates that placing memory metadata in CXL DRAM can substantially degrade application performance. Considering that latency-

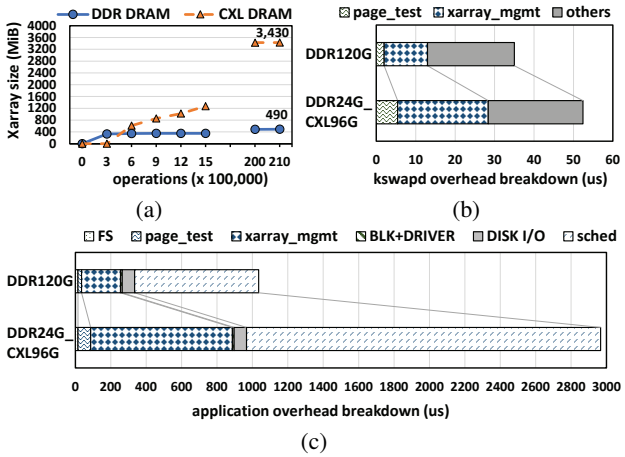


Figure 3: Impact of Xarray management on (a) per-node memory capacity, (b) kswapd performance, and (c) application performance overheads in a system equipped with large capacity CXL DRAM.

sensitive hyperscale applications often require p99.99 read latencies within just a few milliseconds [75, 31, 78, 1, 90], the observed slowdown can pose a significant challenge.

To verify the occurrence of metadata placement in CXL DRAM and the resulting tail-latency effects, we evaluated YCSB workload on RocksDB under realistic application conditions. As shown in Figure 3(a), at the start of execution (up to 400 thousand operations), the kernel allocates Xarray nodes in DDR DRAM to enable fast metadata access. However, as the free pages in DDR DRAM runs out, Xarray nodes begin to compete with application data for the free pages, causing them to spill over into CXL DRAM. During the later stages of execution, most Xarray nodes end up residing in CXL DRAM. This makes kswapd access remote memory while traversing the Xarray, the effect of which is shown in Figure 3(b). As a result, the Xarray access latency significantly increases. Furthermore, because page descriptors are also placed in CXL DRAM, page test time increases by 2.2×. Overall, these effects reduce kswapd 's page reclamation efficiency by 42%. This, in turn, triggers foreground reclamation, which causes the application to experience frequent tail latency spikes, as depicted in Figure 2(a).

Figure 3(c) presents a breakdown of I/O overhead for application threads experiencing foreground reclamation. In systems using both DDR DRAM and CXL DRAM, page descriptor traversal overhead (page_test in Figure 3(c)) during direct reclamation is 3.6× higher than in all-DDR systems, and Xarray traversal overhead (xarray_mgmt in Figure 3(c)) is 3.9× higher. These differences exceed the 2.4× higher access latency of CXL DRAM compared to DDR DRAM, indicating a deeper inefficiency in page reclamation. One main reason is that I/O operations often request data sizes larger than the 4 KiB page, which requires allocating multiple pages. Each page allocation (④ in the Figure 1) triggers a foreground reclamation if free pages are unavailable, substantially increasing I/O latency. Furthermore, these reclamation events consume the CPU core for long periods, allowing other threads to preempt the offending core via scheduling [56, 50], which causes additional long latency spikes (sched in Figure 3(c)). As a result, Figure 3(c) shows that systems with CXL DRAM exhibit 2.8× more extreme latency spikes compared to those using DDR DRAM alone.

3.2 Suitability of Offloading Kernel Metadata Management to CXL DRAM

The kernel's page reclamation process involves simple yet repetitive operations. First, as shown in ⓑ of Figure 1, during page descriptor traversal, the kernel iteratively evaluates each victim page descriptor's flag bits (*e.g.*, valid, referenced, active, clean bits) using basic bitmask operations to determine whether pages are eligible for reclamation. Likewise, during Xarray traversal—as shown in ⓒ of Figure 1—the kernel descends from the Xarray head to a leaf node and replaces the pointer to the page descriptor in the target slot with a specific long integer "shadow [57]" value. This descent process uses simple arithmetic: based on the page offset in the file, the kernel selects the target slot (slot[0] of the $node_0$ in the Figure 1) in the current node's slot array and computes the next node's virtual address via pointer subtraction and shift. Overall, these operations are composed of basic bitmask checks, arithmetic, and shift operations. These operations are repeated for each batch multiple pages (at least 32), which constitutes the minimum unit of a single kernel's page reclamation cycle.

Moreover, both page descriptors and Xarray structures reside within the kernel's direct-mapped virtual address space [44]. Consequently, translating a kernel virtual address to its corresponding physical address requires no page table walk and can be accomplished using a simple arithmetic operation (*e.g.*, via the ___pa() [39] macro). This lightweight translation enables an in-DRAM accelerator attached to CXL DRAM to efficiently access and process metadata without additional mapping logic. Since a single instance of kswapd handles each memory node (both DDR DRAM and CXL DRAM) reclamation, the compute resources provided by modern CXL DRAM controllers (*e.g.*, 16-core Arm processors [73]) are sufficient to effectively manage these tasks.

Additionally, for pages that require extra work beyond the Xarray walk, we can significantly reduce page-reclamation time by letting the host CPU process those tasks in parallel. For instance, file-backed pages mapped via mmap must be unmapped from user PTEs before reclaim; the kernel performs this via reverse mapping (rmap [37]). Since rmap unmapping is control-flow–heavy and can proceed in parallel with the Xarray walk, the host CPU can execute it concurrently with the CXL DRAM-side walk to improve overall reclamation efficiency.

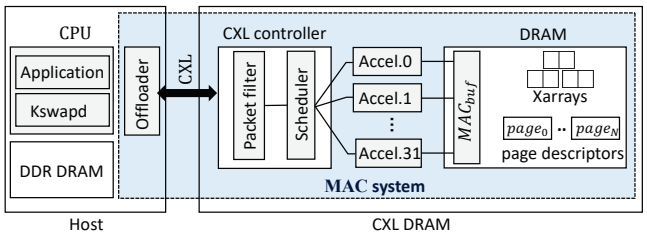


Figure 4: An overview of MAC system.

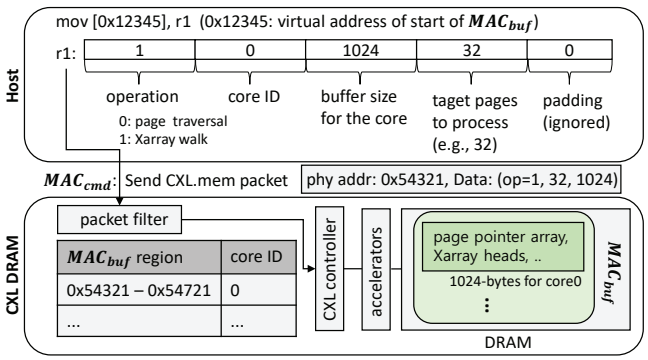


Figure 5: NMP launch process in the MAC system (the virtual address 0x12345 of $\texttt{MAC}_{\text{buf}}$ corresponds to the physical address 0x54321 and is translated by the host.).

4 Design

4.1 Overview

To ensure stable performance for latency-sensitive applications through efficient and swift memory reclamation, we propose Metadata Acceleration in CXL DRAM (MAC). By offloading metadata processing during page reclamation into CXL DRAM, MAC enables rapid memory recovery and effectively eliminates frequent, long tail-latency events.

As shown in the Figure 4, the MAC system enables both applications and the kswapd —the primary triggers of page reclamation—to offload metadata processing (page descriptor traversal and Xarray walks) to CXL DRAM. Accelerators in CXL DRAM then perform near-data processing (NMP) to manage metadata efficiently. The offloading workflow consists of two stages: (1) metadata information (including target page descriptors and head pointers of the Xarrays) is delivered to CXL DRAM, and (2) the accelerators traverse the metadata in parallel and execute the necessary operations. In the MAC system, both page descriptors and Xarray metadata are intentionally allocated to the capacity-rich, low-contention CXL DRAM rather than the more limited DDR DRAM. This placement enables the accelerators to access metadata within CXL DRAM at high speed and with low contention, greatly enhancing metadata-processing efficiency. This design aligns with recent kernel mechanisms [42] that enable metadata placement directly into CXL DRAM, reducing DDR DRAM pressure.

4.2 Metadata Acceleration Launch in CXL DRAM

To offload metadata access operations in page reclamation (traversing page descriptors to identify reclaimable pages and traversing Xarray to remove pages from the page cache), each task must support NMP launch capability. The host must deliver metadata information such as pointer addresses to target page descriptors and Xarray structures. This delivery must be lightweight and efficient.

One could modify the CXL.mem protocol to encode this data directly, but doing so would require hardware modifications to commodity CPU processors—an impractical approach in general-purpose environments [17, 69]. Instead, we introduce a memory-mapped [17] based command mechanism ($\texttt{MAC}_{\text{cmd}}$) and a shared buffer ($\texttt{MAC}_{\text{buf}}$) for communication.

First, private memory in the CXL DRAM ($\texttt{MAC}_{\text{buf}}$ in the Figure 5) is reserved to deliver NMP information. The $\texttt{MAC}_{\text{buf}}$ contains the information required by the accelerators to perform NMP operations within CXL DRAM. The information includes head addresses of Xarray structures to be walked, file page indices targeted for removal from the Xarray, shadow values to be stored in the Xarray, and arrays of addresses to page descriptors. As shown in Figure 5, $\texttt{MAC}_{\text{buf}}$ allocates a dedicated work buffer per host CPU core at system boot.

Second, each host core issues $\texttt{MAC}_{\text{cmd}}$ —a memory-mapped command using the standard CXL.mem protocol—to initiate NMP. As shown in Figure 5, we pre-register the work-buffer address ranges of each host core ($\texttt{MAC}_{\text{buf}}$ regions) in a packet filter. Whenever NMP is needed, the host core writes its $\texttt{MAC}_{\text{cmd}}$ to CXL DRAM. The $\texttt{MAC}_{\text{cmd}}$ is not a new CXL protocol—it is simply a standard memory write operation using the existing CXL.mem request. The packet filter recognizes the memory writes within the registered address range as NMP commands, parses the command, and triggers metadata processing with the accelerator. The $\texttt{MAC}_{\text{cmd}}$ encodes the type of operation (whether traversal of page descriptors or the Xarrays), the requesting host core's ID, and the size of the $\texttt{MAC}_{\text{buf}}$ containing data required for the NMP. Although multiple processes (*e.g.*, application threads, kswapds) may be scheduled per CPU core on the host, Linux disables core preemption during page reclamation. As a result, the process that initiates the NMP remains bound to a single core until reclamation completes. Consequently, allocating one $\texttt{MAC}_{\text{buf}}$ per core (rather than per process) is sufficient to support all concurrent reclamation activities.

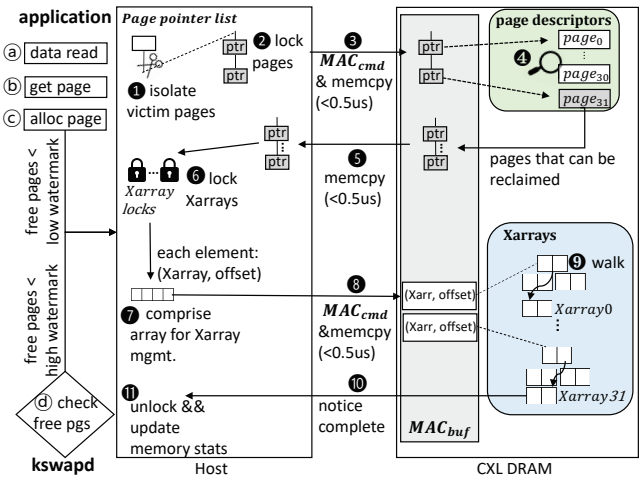


Figure 6: An operational overview of MAC.

4.3 Metadata Acceleration (MAC) in Action

When page reclamation occurs on the host—whether triggered by foreground or background activity—our design leverages near-data processing (NMP) in CXL DRAM to accelerate the process. Figure 6 illustrates the communication and acceleration between the host CPU and CXL DRAM under each reclamation scenario.

Background reclamation acceleration . Each memory node (*e.g.*, host node and CXL DRAM node) runs a dedicated kswapd thread that continuously monitors system free memory and triggers page reclamation when free space drops below the high-watermark (ⓓ in Figure 6). At this point, kswapd first isolates a subset of candidate page pointers (*e.g.*, 32 entries) from a page-pointer list (such as the LRU list) (❶). Next, kswapd acquires locks on these pages (❷), collects their physical addresses into an array, delivers it into the CXL DRAM buffer $\texttt{MAC}_{\text{buf}}$, and issues a $\texttt{MAC}_{\text{cmd}}$ (❸). Accelerators within CXL DRAM then access each page descriptor via the supplied addresses and inspect their flags across all pages (❹). During this inspection, pages are classified into reclaimable (clean file pages) and non-reclaimable (*e.g.*, dirty, invalid, active, or recently referenced). Once the acceleration completes, the host CPU retrieves the results into host memory via memcpy (❺). It then proceeds to evict the reclaimable pages from the page cache by first holding locks on the relevant Xarray structures (❻). The CPU constructs an array of pairs containing each Xarray 's head pointer and the target page index (❼), delivers these into $\texttt{MAC}_{\text{buf}}$, and issues another $\texttt{MAC}_{\text{cmd}}$ (❽). The accelerator uses this information to traverse each Xarray and invalidate the specified pages, repeating the process for the entire set of pages to be removed (❾). Finally, once all invalidations have been performed, the host CPU releases all Xarray locks and updates memory and Xarray statistics (❿-⓫).

Foreground reclamation acceleration . Foreground reclamation is triggered during an application's I/O when free pages drops below the low-watermark. As illustrated in Figure 6, during I/O an application thread may allocate new pages from the kernel (ⓐ–ⓒ). If the number of free pages falls below the low-watermark, the page reclamation routine (❶–⓫) is repeatedly invoked in the kernel. Moreover, when an application allocates multiple pages—such as for I/O units larger than 4 KiB—the page reclamation routine (❶–⓫) may be invoked repeatedly for each individual page allocation attempt (ⓒ).

4.4 Metadata Synchronization between Host and CXL DRAM

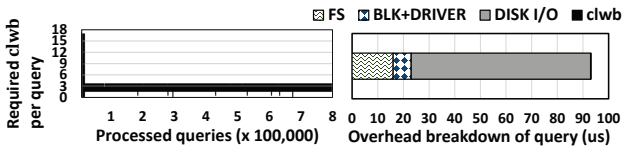
Xarray updates on page insertions and evictions modify internal nodes, requiring synchronization between the host CPU cache and CXL DRAM. For example, when a new page is inserted (⑤ in Figure 1), the node data in the Xarray is updated to point to the newly inserted page (*e.g.*, slot[50] of $node_4$ in Figure 1). To ensure accelerators in CXL DRAM observe up-to-date Xarray content, the host issues clwb to write back the modified cacheline, which costs about 30 ns in our experiments[2]. In contrast, when a page is evicted from the page cache (ⓒ in Figure 1) by the CXL DRAM, the pointer to that page must be invalidated (*e.g.*, slot[1] of $node_3$ in Figure 1). This requires invalidating the corresponding cacheline in the host CPU cache. Starting with CXL 3.x, the memory device supports an invalidate protocol (BIsnp [6]) that allows the device to trigger cacheline invalidation directly, and this is expected to incur a latency of around 500 ns[3]

Accordingly, we integrated a delay model into the MAC system, which enforces that either a clwb instruction or the BIsnp protocol is invoked after the Xarray is updated by the host CPU, in order to guarantee data consistency between the host CPU and CXL DRAM.

However, if such synchronization occurs too frequently, it may degrade application performance. To assess this, we measured clwb/BIsnp rates in practice by counting Xarray-node modifications during insertions and evictions while running RocksDB with YCSB. First, we measured the number of clwb instructions issued per query and the corresponding overhead. As shown in Figure 7(a), the number of clwb operations is high during the early phase of the workload due to the creation of Xarray structures, as almost internal and leaf nodes are allocated at this stage. Over time, however, the number stabilizes to about 1–3 clwb instructions per query. This corresponds to the insertion of 1-3 pages fetched from SSD due to cache misses, each of which requires updating

[2]Because newly inserted pages are excluded from reclamation—and are therefore untouched during Xarray walks in CXL DRAM—we use asynchronous clwb rather than a synchronous clwb

[3]Since no commodity device-biased CXL product provides cache coherency today, we estimated BIsnp overhead using host-biased mode: on our Intel Agilex 7 FPGA, a single host–device coherence handshake measured about 500 ns.



(a) (b)

Figure 7: clwb overhead breakdown: (a) clwb operations required on cache-misses during application execution and (b) application performance when writeback cachelines.

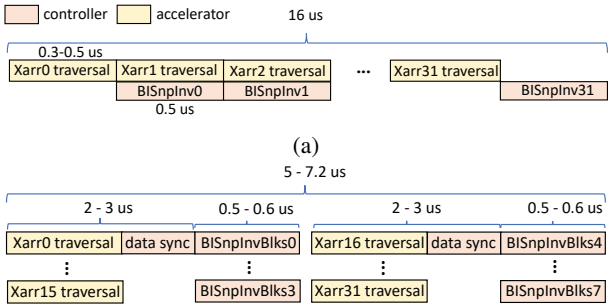


Figure 8: Cooperative parallel page reclamation within the CXL DRAM: (a) overlapping BIsnp and Xarray walk and (b) parallel Xarray walk and bulk invalidation.

the slot array in a leaf node of the Xarray. The overhead of these operations is relatively small: as shown in Figure 7(b), two Xarray operations incur 60 ns (2 × 30 ns) on average, accounting for only 0.06% of the overall query lifetime.

Second, in the case of page deletions, each reclaimed page requires a single cacheline back invalidation. Since pages are reclaimed in batches of 32, the accelerators in CXL DRAM perform 32 Xarray walks via NMP, which takes approximately 12.8 (0.4 us × 32) us on average as shown in Figure 8(a). If BIsnp operations are performed serially after each Xarray walk, they introduce an additional overhead of 16 us (0.5 us × 32), which is longer than the Xarray walks. We address this BIsnp-induced overhead using the techniques described in the following section 4.5.

On the other hand, for page-descriptor traversal, the small subset of descriptors that participate in reclamation (*e.g.*, 32 pages) are not modified by the CXL DRAM; thus no back-invalidation is required. Furthermore, clwb is required only at step ❸ in Figure 6. Issuing clwb in batches (*e.g.*, for 32 pages) pipelines the writebacks asynchronously, and because victim pages are unlikely to have been recently accessed—and thus rarely reside in the CPU cache—the overhead is negligible.

4.5 Cooperative Parallel Page Reclamation

Parallel processing within CXL DRAM . Unlike the read-only descriptor traversal, when offloading Xarray walk, the kernel must write a "shadow" value into the leaf slot of the Xarray (*e.g.*, slot[1] of $node_3$ in Figure 1). This write requires maintaining coherence between host CPU caches and CXL DRAM. Since one shadow is stored per Xarray walk, on average, a single host CPU cacheline must be invalidated via Back Invalidation Snoop message (BIsnp [6]). If BIsnp messages are sent serially for each Xarray walk, their overhead can decrease page reclamation efficiency. To hide this BIsnp overhead, we parallelize the process: as shown in Figure 8(a), the CXL DRAM accelerator and controller perform Xarray walk and the back-invalidation (BIsnpInv in the Figure 8(a)) concurrently.

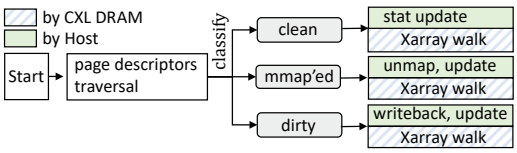


Figure 9: Cooperative page reclamation between Host CPU and CXL DRAM.

Moreover, by exploiting accelerator parallelism, we further reduce total Xarray walk time. Because no structural changes (node allocation or deallocation in the Xarray) occur during the walk, we can process multiple traversals in parallel—up to the number of accelerators or ARM cores, as in Figure 8(b). After batching the Xarray walks, the device issues BIsnp invalidations; these transactions can be pipelined, allowing multiple invalidations to be in flight and overlapping the 500 ns coherence-handshake latency. In addition, CXL 3.x provides block invalidation (BIsnpInvBlks) that covers up to four consecutive cache lines per message, reducing the number of protocol messages by up to 4×. However, when multiple accelerators execute tasks in parallel, the controller must track each accelerator's completion status, because accelerators finish at different times and there are atomically updated shared variables for each accelerator (*e.g.*, the number of accelerators completed). To enable this, shared variables are maintained between the controller and the accelerators to monitor task progress and ensure synchronization. The overhead of this execution and data synchronization introduces an additional delay of approximately 1 us; consequently, as shown in Figure 8(b), executing 16 Xarray walks takes about 2–3 us. As a result, as shown in Figure 8(b), parallelism enables a further 55% reduction in Xarray walk overhead.

Cooperation between host and CXL DRAM . When the host CPU offloads Xarray walks to CXL DRAM, it must hold the corresponding Xarray locks (as illustrated in ❻ of Figure 6) and prohibit CPU preemption until the offloaded operation completes to prevent deadlocks on the held Xarray locks. We exploit this idle interval by parallelizing reclamation: CXL DRAM performs metadata traversals while the host executes the remaining steps.

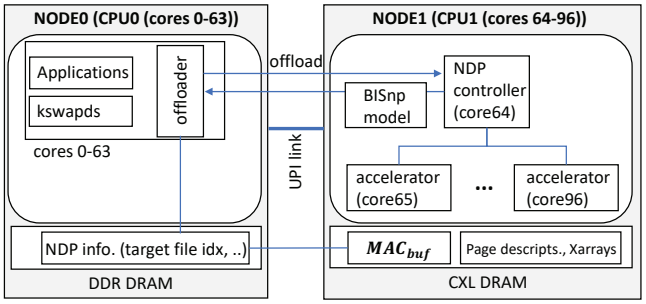


Figure 10: MAC emulation environment.

Table 1: Emulation system configurations.

Component	Specification
OS	Ubuntu 24.04.1 LTS (Linux kernel v6.14)
Storage	Enterprise NVMe SSD 8 TiB and 4 TiB
CPU	Intel Xeon 6710E (2 CPU sockets, each with 64 cores and 96 MiB LLC per CPU)
Socket 0 Socket 1	2 × 32 GiB DDR5-6400 (2, 4, or 8) × 32 GiB DDR5-6400
Local / remote memory access	118 ns / 270 ns

As shown in Figure 9, once page descriptors have been traversed, pages are classified as clean, dirty, or mapped. Clean pages can be immediately removed from the page cache, whereas mapped pages must first be unmapped from page tables before eviction. Dirty pages require a writeback operation before they can be reclaimed. Because writeback incurs costly I/O (via the block layer and device drivers), the kernel prioritizes reclaiming clean pages [81, 2, 50]; nevertheless, under write-excessive workloads (*e.g.*, almost of the pages are dirty), dirty pages can still be reclaimed. To maximize parallelism and hide unmap/writeback costs, we employ a classification-based scheme: while CXL DRAM walks the Xarray, the host CPU concurrently either unmaps mapped pages or initiates writeback for dirty pages, and updates memory and Xarray statistics, as shown in Figure 9.

5 Evaluation

5.1 Setup

We evaluate the effectiveness of our MAC system through two complementary implementations. First, we built a software-based emulation using a dual-NUMA-node server to model CXL-attached NMP behavior. Second, we developed a hardware prototype on FPGA to demonstrate performance gains from accelerations. Our evaluation focuses on comparing tail-latency and analysis, with real-world datacenter workloads.

Emulated MAC **system.** We integrate MAC into Linux kernel 6.14; configuration parameters are in Table 1. As shown in Figure 10, the application and kswapd execute on NODE0 (64 cores), whereas NODE1 emulates a CXL device with 33 cores (one controller and 32 accelerators). To enable metadata processing within CXL DRAM, we allocate NODE1's page-descriptor array via memory hotplug [45] and place Xarray-related slabs on NODE1 using NUMA-controlled allocation [11]. As shown in Figure 10, an offloader lets the application or kswapd on NODE0 send NMP information to NODE1 via memcpy. We implement a BIsnp model to emulate protocol behavior and delay for coherence during metadata updates. Finally, to minimize cache effects in the emulation environment, we maintained the LLC cache size of NODE0 at 96 MiB and restricted the LLC cache size of NODE1 to the minimum allocatable unit of 8 MiB using Intel Cache Allocation Technology (Intel CAT) [26].

Evaluated systems. The evaluated systems are as follows:

- Baseline: Our baseline is the patched Linux 6.14 kernel that optimizes reclaim cycles and reduces tail latency [74]. Xarray nodes are preferentially allocated in CPU-local DDR DRAM, and fall back to CXL DRAM under memory pressure; page descriptors are split by node—NODE0 in NODE0 DRAM, NODE1 in NODE1 DRAM. This placement is a practical approach that efficiently leverages metadata locality and capacity in CXL DRAM systems.

- Baseline-P: Unlike the Baseline, we modified the kernel to pin metadata in CPU-local DDR DRAM by allocating metadata in DDR DRAM even when free space is scarce. Specifically, we implement kernel support for dynamic allocation of Xarray nodes by introducing a flag (GFP_THISNODE) that enforces allocation on the DDR DRAM node when allocating metadata [46]. However, attempting to allocate metadata under extreme memory pressure (*i.e.*, out-of-memory conditions) may lead to abnormal application termination [33]. To mitigate this issue, when an OOM condition is imminent, metadata allocation is redirected to CXL DRAM [51].

- MAC-S: MAC-S (corresponding to Figure 8(a)) accelerates metadata management using a single controller and four accelerators. The controller schedules offloaded requests (*e.g.*, 32 Xarray walks); two accelerators serve two background kswapds and two serve foreground reclamations. Page descriptors for DDR DRAM are allocated in DDR DRAM and traversed by the host CPU. Page descriptors for CXL DRAM and all Xarrays reside in CXL DRAM, with their traversal offloaded to a accelerator.

- MAC-P: MAC-P (corresponding to Figure 8(b)) accelerates metadata management by exploiting the maximum available parallelism of CXL DRAM-resident accelerators, using one controller and thirty-two accelerators. Offloaded tasks run in parallel; after each batch, the controller aggregates results and applies a modeled BIsnpInvBlks delay computed by the BIsnp delay model.

Hardware prototype for MAC system. We prototype MAC on FPGA to quantify the reduction in page descriptor traversal and Xarray-walk time from hardware offload versus host-only traversal in CXL DRAM. At initialization, the host allocates a host buffer and passes its physical address to the accelerator; this buffer carries inputs for Xarray walk and page-descriptor traversal, and later the results. Offloading proceeds as follows: (1) the host writes inputs and triggers the accelerator via an MMIO write; (2) the accelerator fetches inputs with CXL.cache reads and distributes them to per-operation units whose state machines mirror the software logic; (3) the units issue device-biased memory requests that do not require updates[4], but the final Xarray update is performed via a host-biased write, thereby keeping the update coherent with the CPU cache; (4) and the host is notified upon completion. To preserve correctness, when the host updates the Xarray, it flushes the modified cache lines to the device so the accelerator does not operate on stale data.

5.2 Workloads

We conducted evaluations across a diverse set of database workloads using systems that are widely adopted in practice:

- **RocksDB with YCSB workloads:** We deployed RocksDB [71] alongside YCSB [88] to evaluate behaviors under both read-intensive and write-intensive workloads. For this purpose, we generated two RocksDB databases (2.5 TiB and 2.0 TiB).
- **PostgreSQL with OLTP workload:** Using PostgreSQL [65]—an open-source, object-relational database management system—we evaluated transaction workload with the OLTP benchmark [64]. We created a 2.0 TiB database for the evaluation.
- **Neo4j with LDBC SNB workloads:** We evaluate Neo4j [15], a graph database, using the LDBC SNB benchmark tool [16]. We use two types of queries: Interactive Short (IS) workload and Interactive Complex (IC) workload [12]. To this end, we import a 1 TiB dataset (SF: 1,000) [14] into the database.
- **LMDB with GET workload:** We evaluate LMDB [55], a database that performs file I/O using mmap [41], with the ioarena [27] benchmark. We created a 1.7 TiB database.

5.3 Results

All reported results are collected after a warm-up phase, during which both DDR DRAM and CXL DRAM are fully populated. Logs used for CDF and box-plot analysis are flushed to a separate SSD to avoid interfering with application performance.

RocksDB with YCSB workloads. We first evaluate a YCSB read-only workload on a 2.5 TiB RocksDB database. To assess the impact of metadata spillover into CXL DRAM on page-reclamation efficiency, we vary the DDR:CXL capacity ratio by fixing host memory at 64 GiB and setting CXL to 64/128/256 GiB (1:1/1:2/1:4); for each configuration, the thread count is tuned to highest throughput. Figure 11 presents the latency distribution in the form of a CDF. Across all ratios, MAC-S and MAC-P exhibit lower tail latencies than Baseline. For example, at a DDR DRAM to CXL DRAM ratio of 1:2, MAC-S reduces the p99.99 latency by 97% compared to Baseline, while MAC-P achieves a 98% reduction. These improvements are closely tied to the page reclamation rate.

Table 2: Xarray distribution across DDR DRAM and CXL DRAM under Baseline at the end of the benchmark.

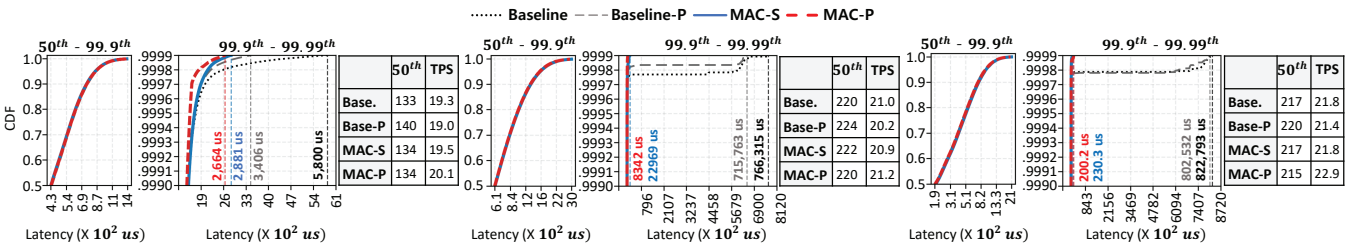
DDR-CXL Configuration	DDR	CXL	Total
64G-64G	2,047 MiB	2,378 MiB	4,425 MiB
64G-128G	2,030 MiB	3,451 MiB	5,481 MiB
64G-256G	2,101 MiB	4,420 MiB	6,521 MiB

Table 2 shows Baseline suffers substantial Xarray spillover to CXL (*e.g.*, 68% at 1:4), throttling kswapd. In contrast, MAC-S and MAC-P place metadata in CXL DRAM and accelerate its management via NMP, thereby maximizing the efficiency of kswapd. For example, compared to Baseline, MAC-S and MAC-P reduce Xarray walk overhead by 25% and 80%, respectively, and reduce page descriptor traversal overhead by 58%. As a result, the MAC-P increases free-page generation by 36%, despite incurring BIsnp (sync.) and host–device communication (comm.) overheads, as shown in Figure 12(b). Consequently, as shown in Figure 12(a), MAC-P reduces foreground reclamations by 66% at 1:2 compared to Baseline.

Compared to Baseline-P, both MAC-S and MAC-P also exhibit lower tail latency distributions. As shown in Figure 11(a), MAC-S and MAC-P reduce the p99.99 latency by 15% and 22%, respectively, relative to Baseline-P. In Baseline-P, most metadata resides in DDR DRAM, which reduces page reclamation overhead by 22% compared to Baseline, as shown in Figure 12(b). Despite this benefit, Baseline-P still suffers from latency spikes due to the high overhead of slab memory management [40].

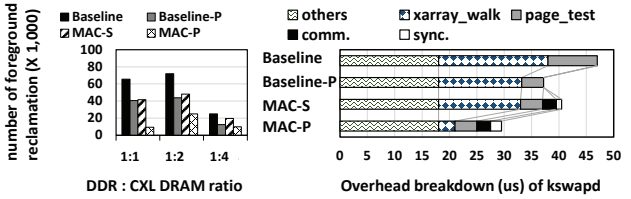
When an application fetches data from SSD into memory, Xarray nodes are dynamically allocated from pages in slab memory pools [40], where each NUMA node (DDR DRAM or CXL DRAM) maintains its own pool. However, under Baseline-P, which allocates Xarray only on DDR DRAM, severe memory pressure significantly increases the overhead of managing the DDR DRAM node's slab memory pool for metadata allocation. Specifically, application data and kernel metadata contend heavily for DDR DRAM space, as metadata allocations are restricted to the DDR DRAM despite the presence of separate slab pools in CXL DRAM. We observed that while slab allocation latency [46] ranges from 2–4 μs in Baseline, MAC-S, and MAC-P, it increases to 10–600 μs in Baseline-P.

[4] We issue device-biased reads to bypass the CXL home agent's coherence checks, thereby accelerating each read.



(a) (b) (c)

Figure 11: Latency CDFs and TPS (×1,000) across memory configurations for the RocksDB read-only workload: (a) DDR:CXL = 1:1 (90 threads), (b) DDR:CXL = 1:2 (200 threads), and (c) DDR:CXL = 1:4 (250 threads).



(a) (b)

Figure 12: kswapd reclamation overhead and resulting foreground reclamations: (a) foreground reclamation frequency and (b) kswapd for CXL DRAM overhead breakdown when running RocksDB with YCSB read-only workload under DDR:CXL = 1:2 configuration.

Consequently, as shown in Figure 11, the increased slab allocation cost and resulting contention on shared resources cause Baseline-P to exhibit tail latencies comparable to Baseline.

In addition to the slab management overhead, the preferential allocation of large metadata to DDR DRAM forces application data to be allocated and served from CXL DRAM, thereby degrading application throughput. We observe that Baseline-P achieves lower TPS (transactions per second) than the other schemes. For example, as shown in Figure 11(c), Baseline-P delivers 6% lower TPS than MAC-P, which experiences the least memory pressure on DDR DRAM and achieves higher utilization of application data in DDR DRAM.

We next evaluate a YCSB workload with a 5:5 read-to-update (read-modify-write) ratio on a 2.0 TiB RocksDB database. Updates trigger memory-intensive compactions; when compactions stall, the tail latency of writes and reads increases sharply [86]. As shown in Figure 13, across all memory configurations our MAC system delivers stable performance. For example, for read operations, as shown in Figure 13(a), MAC-P reduces the p99.9 latency by 24% and the p99.99 latency by an average of 27% compared to Baseline. For update operations, as shown in Figure 13(d), MAC-P reduces p99.99 latency by 52% compared to Baseline and by 48% compared to Baseline-P. Notably, by alleviating DDR DRAM memory pressure caused by large metadata, MAC-P improves throughput by 10% over Baseline and 17% over Baseline-P. This result suggests that, for update workloads, higher utilization of fast DDR DRAM by application data leads to improved throughput.

PostgreSQL with OLTP. We evaluate an OLTP workload (SELECT-only) on a 2.0 TiB PostgreSQL database using pgbench [64]. We configure PostgreSQL 's internal buffer (shared_buffers [66]) to 25% of system memory following PostgreSQL 's guidance [66], and for each memory setting, we increase the thread count to achieve the highest queries-per-second. As shown in Figure 14, our MAC-S and MAC-P deliver stable performance across all memory configurations compared to Baseline. For example, in a configuration with a DDR DRAM to CXL DRAM ratio of 1:2, MAC-P reduces the number of queries that experience foreground reclamation by 88% (Figure 14(e)), which in turn lowers the p99.99 latency by 92% (Figure 14(b)) and maintains stable latency during execution.

Moreover, as shown in Figure 14(d), MAC-S and MAC-P improve TPS by up to 5% across all memory configurations over Baseline and Baseline-P by more efficiently utilizing DDR DRAM.

Neo4j with LDBC SNB workloads. Graph processing in Neo4j can incur significant disk I/O under memory pressure when the dataset exceeds available memory, which in turn triggers frequent memory reclamation. We first evaluate our system using an IS workload that concurrently executes seven Interactive Short Queries (IS) [12], which exhibit OLTP characteristics. As shown in Figure 15(a), both MAC-S and MAC-P achieve lower tail latencies across all queries by enabling more efficient page reclamation under memory pressure. For example, for SQ6, MAC-P reduces p99.9 latency by 21% and p99.99 latency by 82% compared to Baseline, and further reduces p99.9 and p99.99 latencies by 11% and 24%, respectively, compared to Baseline-P.

Moreover, MAC-S and MAC-P achieve shorter overall benchmark runtimes than Baseline-P. As shown in Figure 15(b), they reduce the total runtime by 7% compared to Baseline-P, driven by faster metadata allocation as well as higher application data fetch and processing within DDR

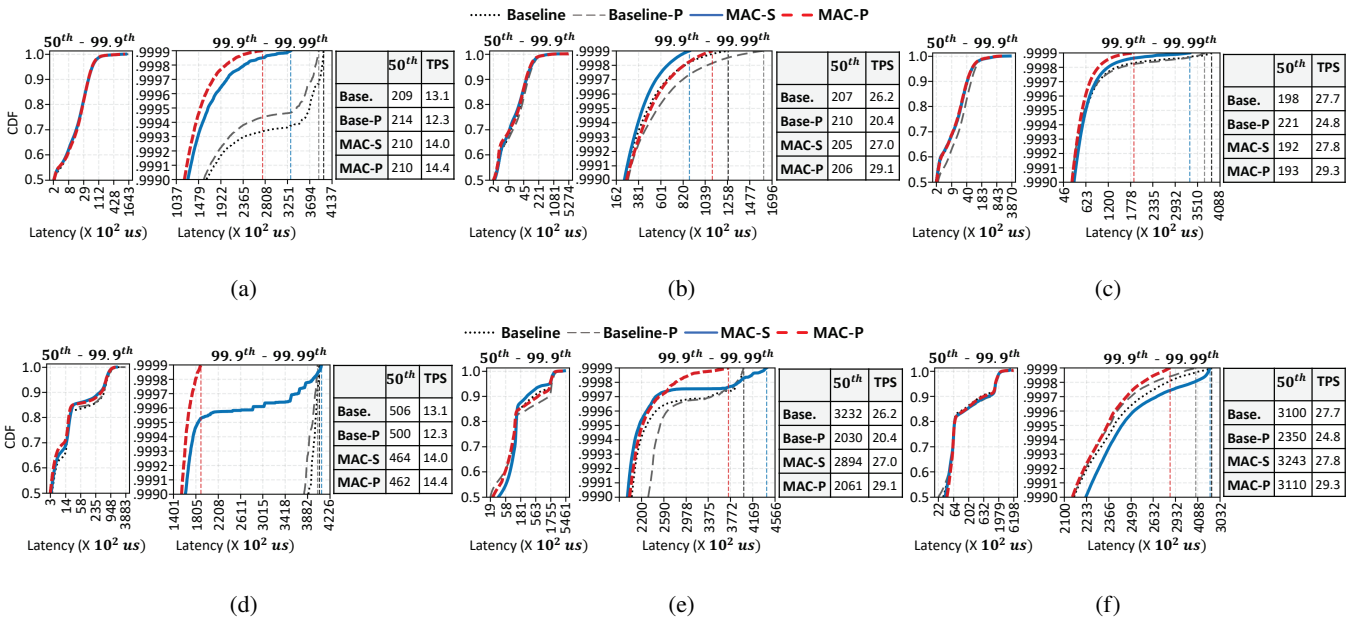


Figure 13: Latency CDFs and TPS (×1,000) across memory configurations for the RocksDB 50/50 read–update workload: (a) DDR:CXL = 1:1 (READ, 100 threads), (b) DDR:CXL = 1:2 (READ, 230 threads), (c) DDR:CXL = 1:4 (READ, 250 threads), (d) DDR:CXL = 1:1 (UPDATE, 100 threads), (e) DDR:CXL = 1:2 (UPDATE, 230 threads), and (f) DDR:CXL = 1:4 (UPDATE, 250 threads).

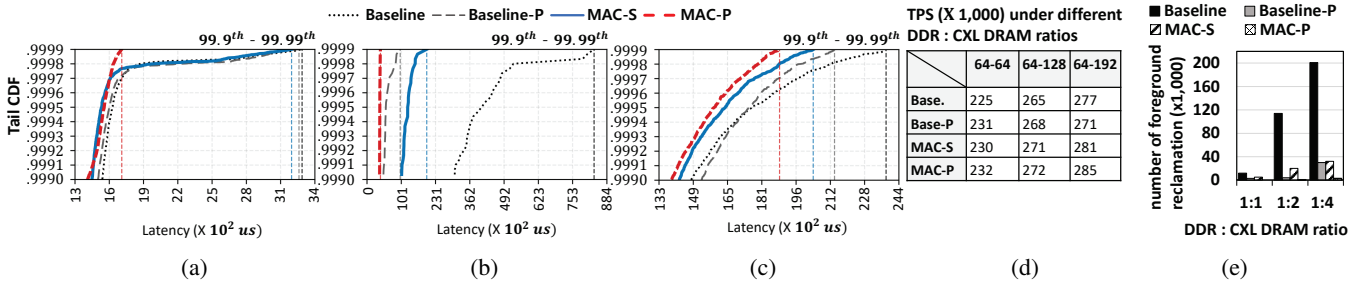


Figure 14: Latency CDFs, TPS, and foreground reclamation frequency across memory configurations for the OLTP workload with PostgreSQL: (a) DDR:CXL = 1:1 (100 threads), (b) DDR:CXL = 1:2 (160 threads), (c) DDR:CXL = 1:4 (250 threads), (d) TPS comparisons, and (e) foreground reclamation frequency.

DRAM.

Next, we evaluate our system using a hybrid workload that concurrently executes seven Interactive Short (IS) queries and eleven Interactive Complex (IC) queries [12] which exhibit OLAP characteristics. As shown in Figure 16(a), our system achieves lower p99.9 and p99.99 tail latencies for most queries. However, for some long-running, computation-bound queries (e.g., CQ1 and CQ2), the improvement is smaller. Nevertheless, in terms of overall runtime, both MAC-S and MAC-P still outperform Baseline-P, achieving an 11% reduction in total runtime, as shown in Figure 16(b).

LMDB with GET operations. We evaluate the GET workload on a 1.7 TiB LMDB database using ioarena [27]. Since LMDB [55] accesses the database via mmap [41], we assessed how the cooperative page reclamation scheme in Fig-

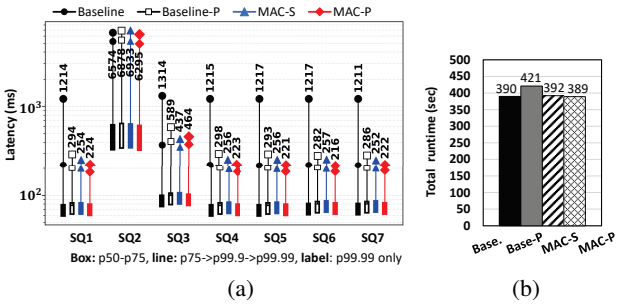
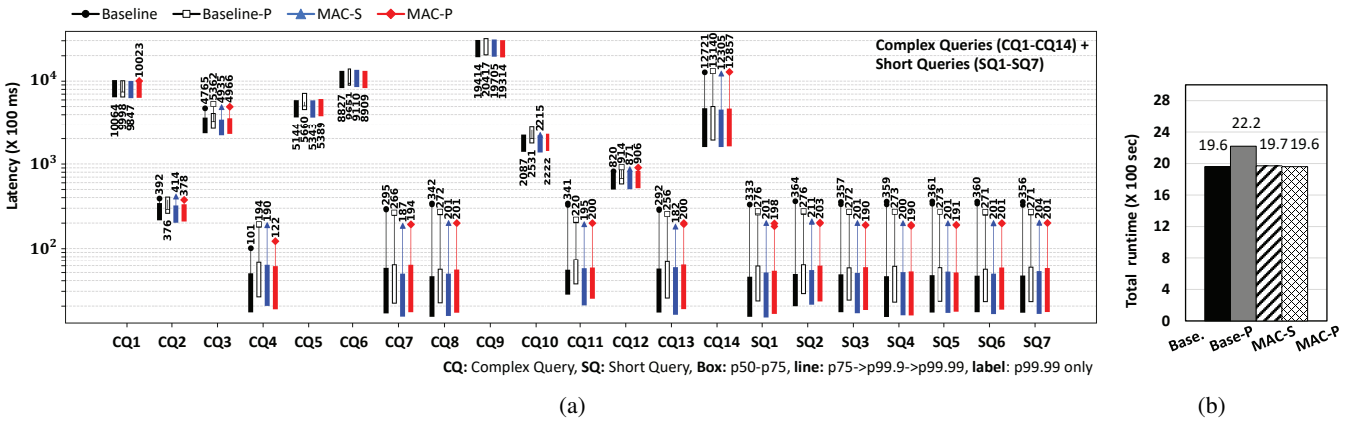
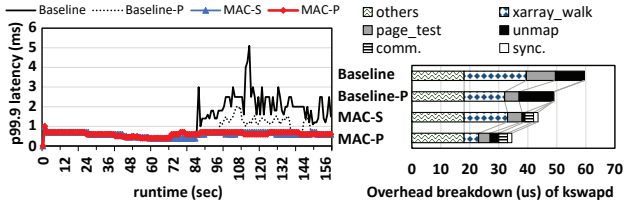


Figure 15: Latency distributions and total runtime under a 1:2 DDR DRAM:CXL DRAM configuration: (a) box plots for the Interactive Short (IS) workload (400K ops with 200 threads), (b) runtime across schemes.



(a) (b)

Figure 16: Latency distributions and total runtime for the combination of the Interactive Short (IS) and Interactive Complex (IC) workload (13K ops with 200 threads) under a 1:2 DDR:CXL DRAM configuration: (a) box plots, (b) runtime across schemes.



(a) (b)

Figure 17: Tail latency (updated every 1 s) under DDR:CXL = 1:2 configuration for the GET benchmark with LMDB (100 threads): (a) p99.9 and (b) kswapd overhead breakdown.

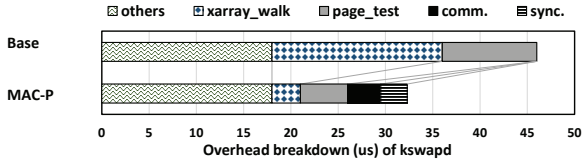


Figure 18: FPGA-based validation of page reclamation acceleration.

ure 9 affects performance. As shown in Figure 17(a), MAC-S and MAC-P reduce p99.9 latency by 62% and 35% on average over Baseline and Baseline-P, respectively, while maintaining stable performance.

This performance improvement stems from the fact that our MAC processes the unmapping of reclaimed mmap'ed pages from the process page table in parallel with the Xarray walk, as shown in Figure 9. As a result, as shown in Figure 17(b), in MAC-S and MAC-P the unmap operation is overlapped with the Xarray walk time, reducing page reclamation time by 26% and 42%, respectively, compared to Baseline. Compared to Baseline-P, MAC-S and MAC-P reduce reclamation time by 10% and 31%, respectively.

Results with a prototype system.

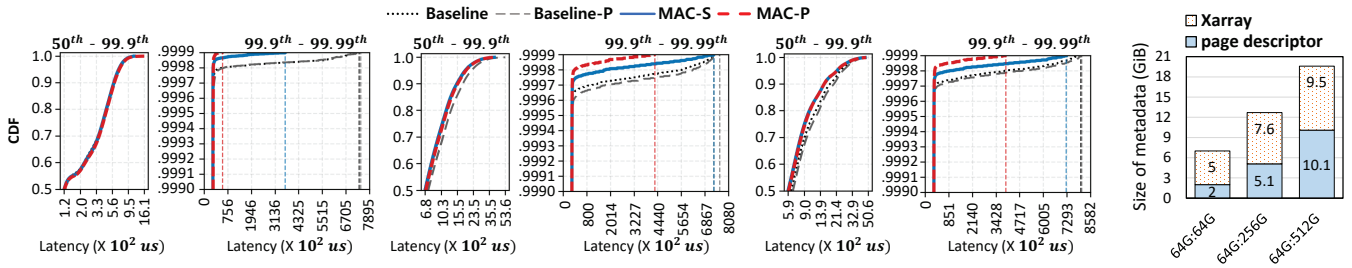
To validate the fidelity of our emulation results, we measured the end-to-end latency of kswapd 's page reclamation on real FPGA hardware. To do so, we synthesized the Linux kernel's Xarray walk function [53] and page-descriptor traversal logic [50] into an FPGA RTL design. We perform our FPGA evaluation on a server described in Table 3.

Table 3: Prototype system configurations.

Component	Specification
OS	Ubuntu 24.04.1 LTS (Linux kernel v6.11)
DDR DRAM size	128 GiB
CPU	Intel Xeon 6538Y+ (2 CPU sockets, each with 32 cores and 60 MiB LLC per CPU) [22]
Socket 0 & 1	2 × 64 GiB DDR5-4800
CXL device On-board DRAM	Intel Agilex 7 I-series FPGA [24] 2 x 8 GiB DDR4-2666
FPGA utilization	ALM: 233k(26%), Registers: 480k(26%), M20K: 2123(16%) [20, 19]

We benchmark the overhead of the page-reclamation process invoked by kswapd, using synthetic page descriptors and Xarray structures representative of file-sizes and tree depths observed in RocksDB and PostgreSQL (e.g., Xarray depth of 3 levels).

As shown in Figure 18, our MAC technique reduces the Xarray-walk overhead by 82% and the page descriptor traversal overhead by 48% by enabling pipelined and parallel execution of accelerators within CXL DRAM. While offloading work to CXL DRAM introduces additional costs—the time for the host CPU to transfer required information and trigger MMIO operations (comm. in Figure 18), and the overhead of notifying the host CPU of completion via polling and returning the computed values, along with cache-coherency overhead (sync. in Figure 18)—MAC still decreases the end-to-end runtime of kswapd page reclamation by 30%. This improvement is comparable to the performance of MAC-P in Figure 12(b), validating the results of the emulated architecture.



(a) (b) (c) (d)

Figure 19: Latency CDFs and metadata size across memory configurations for the 3.5 TiB RocksDB read-only workload: (a) DDR:CXL = 1:1 (60 threads), (b) DDR:CXL = 1:4 (250 threads), (c) DDR:CXL = 1:8 (300 threads), and (d) metadata size.

6 Related Work

We organize related work into two areas: (1) metadata overhead reduction and (2) near-memory processing (NMP).

Metadata management. Several studies [33, 13, 68, 63, 35, 87, 67] have improved system and application performance by reducing the overheads of kernel metadata management. For example, Radiant [33] tackles the performance degradation caused by long address translation when page tables spill over to high-latency non-volatile main memories [21] by migrating page tables. Hermit [68] mitigates tail-latency issues caused by remote memory accesses during application-page swapping in RDMA-connected clusters using multiple swap threads, but focuses on in-memory applications and does not consider the overheads of metadata structures. In contrast, our MAC system is distinguished in that there has been no prior characterization of the size or performance overhead of existing kernels' memory management metadata in large-capacity, CPU-less CXL DRAM environments.

NMP with CXL DRAM. Many works [79, 28, 32, 62, 85, 89, 30] have improved application performance by leveraging CXL DRAM–based near-memory processing (NMP) to overcome performance bottlenecks. For example, [85] accelerates query processing with in-memory column scans and B^+-tree operations, while [79] performs NMP-based memory profiling on CXL DRAM to reduce the host-CPU burden, improving both migration accuracy and performance. In contrast, our MAC system offloads Linux's page-reclamation path to CXL DRAM, showcasing a complementary NMP use case that targets OS-level memory-management overheads.

7 Discussion

While vendors typically consider DDR DRAM to CXL DRAM ratios ranging from 1:1 to 1:4, future systems may employ more aggressive configurations such as 1:8. As shown in Figure 19(d), under a 1:8 ratio, metadata occupies approximately 20 GiB, corresponding to 31% of DDR DRAM capacity. Consequently, Baseline and Baseline-P, which place metadata in DDR DRAM, experience 5-30% higher 50th-90th percentile latencies than MAC-S and MAC-P due to increased contention for scarce DDR DRAM, as shown in Figure 19(c). These results show that, as CXL DRAM capacity scales, preserving DDR DRAM for application data becomes increasingly important for latency-sensitive workloads [3].

Although we focus on CXL memory expanders, the same considerations apply to future CXL 3.0 memory-pooling systems [34] with multiple CXL DRAM devices. In this case, nodes belonging to the same Xarray should preferably reside in the same CXL DRAM device and migrate together when reassigned. Moreover, applications spanning multiple CXL DRAM devices could exploit device-level parallelism for metadata processing.

8 Conclusion

We have presented MAC, a near-memory processing approach that co-designs the OS and a CXL-resident accelerator to offload the dominant metadata operations in the memory reclaim path—Xarray [52] walks and page-descriptor [49] traversals. To our knowledge, this is the first paper to address the capacity and performance costs of kernel metadata in CXL DRAM systems from a sustainability perspective and to resolve them via NMP-based reclamation offloading. Compared to a modern Linux kernel–based system equipped with CXL DRAM, MAC reduces the p99.99 latency by up to 98%, thereby ensuring sustained, stable application performance.

Acknowledgments

This paper was result of the research project supported by SK hynix Inc. This work was supported in part by PRISM, one of the seven centers in JUMP 2.0, a Semiconductor Research Corporation (SRC) program sponsored by DARPA. This work was partly supported by Institute of Information & Communications Technology Planning & Evaluation (IITP) grant funded by the Korea government (MSIT) (RS-2025-02214654 and RS-2024-00456287). The ICT at Seoul National University provided research facilities for this study.

References

[1] Marc Brooker, Mike Danilov, Chris Greenwood, and Phil Piwonka. On-demand container loading in {AWS} lambda. In *Proceedings of the USENIX Annual Technical Conference (ATC'23)*, pages 315–328, 2023.

[2] Chao Yang. The Page Cache and Page Writeback. https://github.com/firmianay/Life-long-Learner/blob/master/linux-kernel-development/chapter-16.md, 2022.

[3] Albert Cho, Anish Saxena, Moinuddin Qureshi, and Alexandros Daglis. Coaxial: A cxl-centric memory system for scalable servers. In *SC24: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–15. IEEE, 2024.

[4] COMPUTE EXPRESS LINK CONSORTIUM. DRAM Resource Scalability Enabled by CXL. https://computeexpresslink.org/blog/dram-resource-scalability-enabled-by-cxl-1071/, 2023.

[5] COMPUTE EXPRESS LINK CONSORTIUM. Compute Express Link (CXL). https://computeexpresslink.org/wp-content/uploads/2024/02/CXL-2.0-Specification.pdf, 2024.

[6] COMPUTE EXPRESS LINK CONSORTIUM. Compute express link (cxl) specification. https://computeexpresslink.org/wp-content/uploads/2024/02/CXL-3.0-Specification.pdf, 2024.

[7] COMPUTE EXPRESS LINK CONSORTIUM. Opportunities and challenges for compute express link (cxl). https://computeexpresslink.org/wp-content/uploads/2024/11/CR-CXL-101_FINAL.pdf, 2024.

[8] COMPUTE EXPRESS LINK CONSORTIUM. About CXL. https://computeexpresslink.org/about-cxl/, 2025.

[9] COMPUTE EXPRESS LINK CONSORTIUM. Advantages of cxl memory sharing for emerging applications. https://computeexpresslink.org/wp-content/uploads/2025/06/CXL_Q2-2025-Webinar_FINAL.pdf, 2025.

[10] COMPUTE EXPRESS LINK CONSORTIUM. FMS 2025 CXL Demos. https://computeexpresslink.org/fms-2025-cxl-demos/, 2025.

[11] Debian. Allocate an object on the specified node. https://manpages.debian.org/testing/linux-manual-4.9/kmem_cache_alloc_node.9, 2017.

[12] Orri Erling, Alex Averbuch, Josep Larriba-Pey, Hassan Chafi, Andrey Gubichev, Arnau Prat, Minh-Duc Pham, and Peter Boncz. The LDBC social network benchmark: Interactive workload. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD'15)*, pages 619–630, 2015.

[13] Bin Gao, Qingxuan Kang, Hao-Wei Tee, Kyle Timothy Ng Chu, Alireza Sanaee, and Djordje Jevdjic. Scalable and effective page-table and tlb management on numa systems. In *Proceedings of the USENIX Annual Technical Conference (ATC'24)*, pages 445–461, 2024.

[14] Graph Data Council. LDBC Social Network Benchmark (LDBC SNB). https://ldbcouncil.org/benchmarks/snb/datasets/, 2022.

[15] Graph Data Council. Neo4j: Graphs for Everyone. https://github.com/neo4j/neo4j, 2024.

[16] Graph Data Council. LDBC_SNB_INTERACTIVE. https://github.com/ldbc, 2026.

[17] Hyungkyu Ham, Jeongmin Hong, Geonwoo Park, Yunseon Shin, Okkyun Woo, Wonhyuk Yang, Jinhoon Bae, Eunhyeok Park, Hyojin Sung, Euicheol Lim, and Gwangsun Kim. Low-overhead general-purpose near-data processing in cxl memory expanders. In *Proceedings of the IEEE/ACM International Symposium on Microarchitecture (MICRO'24)*, pages 594–611. IEEE, 2024.

[18] Reece Hayden and Paul Schell. Opportunities and challenges for compute express link (cxl). *ABI Research*, 2024.

[19] Intel Corporation. Adaptive Logic Module (ALM) Definition. https://www.intel.com/content/www/us/en/programmable/quartushelp/17.0/reference/glossary/def_alm.htm, 2017.

[20] Intel Corporation. M20K memory block Definition. https://www.intel.com/content/www/us/en/programmable/quartushelp/17.0/reference/glossary/def_m20k.htm, 2017.

[21] Intel Corporation. Intel Optane Persistent Memory. https://www.intel.com/content/www/us/en/products/docs/memory-storage/optane-persistent-memory/optane-dc-persistent-memory-brief.html, 2020.

[22] Intel Corporation. Intel Xeon Gold 6538Y+ Processor. https://www.intel.com/content/www/us/en/products/sku/237563/intel-xeon-gold-6538y-processor-60m-cache-2-20-ghz/specifications.html, 2023.

[23] Intel Corporation. The Intel Xeon 6 Product Family. https://www.intel.com/content/dam/www/central-libraries/us/en/documents/2024-05/intel-xeon-6-product-brief.pdf, May 2024.

[24] Intel Corporation. Agilex 7 FPGA I-Series Development Kit (2x R-Tile and 1x F-Tile). https://www.intel.com/content/www/us/en/products/details/fpga/development-kits/agilex/agi027.html, 2025.

[25] Intel Corporation. Intel Xeon 6 Processors. https://www.intel.com/content/www/us/en/products/details/processors/xeon.html, 2025.

[26] Intel Corporation. Introduction to Cache Allocation Technology. https://www.intel.com/content/www/us/en/developer/articles/technical/introduction-to-cache-allocation-technology.html, 2025.

[27] IOarena. IOarena. https://github.com/pmwkaa/ioarena, 2022.

[28] Junhyeok Jang, Hanjin Choi, Hanyeoreum Bae, Seungjun Lee, Miryeong Kwon, and Myoungsoo Jung. {CXL-ANNS}:{Software-Hardware} collaborative memory disaggregation and computation for {Billion-Scale} approximate nearest neighbor search. In *Proceedings of the USENIX Annual Technical Conference (ATC'23)*, pages 585–600, 2023.

[29] Houxiang Ji, Srikar Vanavasam, Yang Zhou, Qirong Xia, Jinghan Huang, Yifan Yuan, Ren Wang, Pekon Gupta, Bhushan Chitlur, Ipoom Jeong, et al. Demystifying a cxl type-2 device: A heterogeneous cooperative computing perspective. In *Proceedings of the 57th IEEE/ACM International Symposium on Microarchitecture (MICRO'24)*, 2024.

[30] Houxiang Ji, Yifan Yuan, Yang Zhou, Ipoom Jeong, Ren Wang, Saksham Agarwal, and Nam Sung Kim. Re-architecting end-host networking with cxl: Coherence, memory, and offloading. In *Proceedings of the 58th IEEE/ACM International Symposium on Microarchitecture (MICRO'25)*, 2025.

[31] Saehoon Kim, Yuxiong He, Seung-won Hwang, Sameh Elnikety, and Seungjin Choi. Delayed-dynamic-selective (dds) prediction for reducing extreme tail latency in web search. In *Proceedings of the ACM International Conference on Web Search and Data Mining (WSDM'15)*, pages 7–16, 2015.

[32] Seoyoung Ko, Hyunjeong Shim, Wanju Doh, Sungmin Yun, Jinin So, Yongsuk Kwon, Sang-Soo Park, Si-Dong Roh, Minyong Yoon, Taeksang Song, et al. Cosmos: A cxl-based full in-memory system for approximate nearest neighbor search. *IEEE Computer Architecture Letters*, 24(1):173–176, 2025.

[33] Sandeep Kumar, Aravinda Prasad, Smruti R Sarangi, and Sreenivas Subramoney. Radiant: efficient page table management for tiered memory systems. In *Proceedings of the ACM SIGPLAN International Symposium on Memory Management (ISMM'21)*, pages 66–79, 2021.

[34] Huaicheng Li, Daniel S Berger, Lisa Hsu, Daniel Ernst, Pantea Zardoshti, Stanko Novakovic, Monish Shah, Samir Rajadnya, Scott Lee, Ishwar Agarwal, et al. Pond: Cxl-based memory pooling systems for cloud platforms. In *Proceedings of the ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'23)*, pages 574–587, 2023.

[35] Zhiyue Li and Guangyan Zhang. {StreamCache}: Revisiting page cache for file scanning on fast storage devices. In *Proceedings of the USENIX Annual Technical Conference (ATC'24)*, pages 1119–1134, 2024.

[36] Linus. How to access I/O mapped memory from within device drivers. https://www.kernel.org/doc/html/v5.9/core-api/bus-virt-phys-mapping.html.

[37] Linux Kernel. Reverse mapping. https://www.kernel.org/doc/gorman/html/understand/understand006.html, 2003.

[38] Linux Kernel. Page Frame Reclamation. https://www.kernel.org/doc/gorman/html/understand/understand013.html, 2004.

[39] Linux Kernel. Mapping Physical to Virtual Kernel Addresses. https://www.kernel.org/doc/gorman/html/understand/understand006.html, 2005.

[40] Linux Kernel. Slab Allocator. https://www.kernel.org/doc/gorman/html/understand/understand011.html, 2005.

[41] Linux Kernel. Memory Mapping. https://linux-kernel-labs.github.io/refs/heads/master/labs/memory_mapping.html, 2020.

[42] Linux Kernel. Memory hot(un)plug. https://www.kernel.org/doc/html/v6.6/admin-guide/mm/memory-hotplug.html, March 2021.

[43] Linux Kernel. What is numa? https://www.kernel.org/doc/html/v6.6/mm/numa.html?highlight=numa+node, 2021.

[44] Linux Kernel. Memory Management. https://www.kernel.org/doc/html/next/x86/x86_64/mm.html, 2023.

[45] Linux Kernel. Memory Map. https://docs.kernel.org/driver-api/cxl/linux/memory-hotplug.html, 2023.

[46] Linux Kernel. filemap_add_folio. https://elixir.bootlin.com/linux/v6.14.6/source/mm/filemap.c#L859, 2025.

[47] Linux Kernel. Memory Allocation Guide. https://www.kernel.org/doc/html/v4.20/core-api/memory-allocation.html, 2025.

[48] Linux Kernel. Physical Memory. https://www.kernel.org/doc/html/next/mm/physical_memory.html, 2025.

[49] Linux Kernel. Physical Memory Model. https://docs.kernel.org/mm/memory-model.html, 2025.

[50] Linux Kernel. shrink_folio_list function. https://elixir.bootlin.com/linux/v6.14.6/source/mm/vmscan.c#L1082, 2025.

[51] Linux Kernel. xa_nomem. https://elixir.bootlin.com/linux/v6.14.6/source/lib/xarray.c#L377, 2025.

[52] Linux Kernel. Xarray. https://docs.kernel.org/core-api/xarray.html, 2025.

[53] Linux Kernel. xas_alloc. https://elixir.bootlin.com/linux/v6.14.6/source/lib/xarray.c#L769, 2025.

[54] Bo Liu, Kaihao Bai, Pu Pang, Quan Chen, Yaoxuan Li, Zhuo Song, Baolin Wang, and Minyi Guo. Pmr: Priority memory reclaim to improve the performance of latency-critical services. In *Proceedings of the IEEE International Conference on Parallel and Distributed Systems (ICPADS'23)*, pages 2452–2459, 2023.

[55] LMDB. LMDB: Lightning Memory-Mapped Database from Symas. https://github.com/LMDB, 2025.

[56] LWN. RCU, cond_resched(), and performance regressions. https://lwn.net/Articles/603252/, 2014.

[57] Matthew Wilcox. Xarray. https://www.kernel.org/doc/html/v6.6/core-api/xarray.html, 2023.

[58] MemVerge. At MemCon, MemVerge Demonstrates How Compute Express Link (CXL) Memory Expansion Will Close the Gap Between CPU and Memory Performance. https://www.prnewswire.com/news-releases/at-memcon-memverge-demonstrates-how-compute-express-link-cxl-memory-expansion-will-close-the-gap-between-cpu-and-memory-performance-301783895.html?utm_source=chatgpt.com, 2023.

[59] MemVerge. 50% less cost and 300% more capacity with cxl. https://memverge.com/cxl-use-case-slash-memory-costs-and-expand-capacity/, 2024.

[60] MemVerge. The case for cxl memory expansion. https://files.futurememorystorage.com/proceedings/2024/20240806_CXLT-102-1_Tian.pdf, 2024.

[61] Micron. Micron's perspective on impact of cxl on dram bit growth rate. https://assets.micron.com/adobe/assets/urn:aaid:aem:b2e25f63-85a2-44c9-b46f-717830deefa5/original/as/cxl-impact-dram-bit-growth-white-paper.pdf, 2023.

[62] Sang-Soo Park, KyungSoo Kim, Jinin So, Jin Jung, Jonggeon Lee, Kyoungwan Woo, Nayeon Kim, Younghyun Lee, Hyungyo Kim, Yongsuk Kwon, et al. An lpddr-based cxl-pnm platform for tco-efficient inference of transformer-based large language models. In *Proceedings of the IEEE International Symposium on High-Performance Computer Architecture (HPCA'24)*, pages 970–982. IEEE, 2024.

[63] Kiet Tuan Pham, Seokjoo Cho, Sangjin Lee, Lan Anh Nguyen, Hyeongi Yeo, Ipoom Jeong, Sungjin Lee, Nam Sung Kim, and Yongseok Son. Scalecache: A scalable page cache for multiple solid-state drives. In *Proceedings of the European Conference on Computer Systems (EuroSys'25)*, pages 641–656, 2024.

[64] PostgreSQL. PostgreSQL Client Applications. https://www.postgresql.org/docs/current/pgbench.html, 2025.

[65] PostgreSQL. PostgreSQL: The World's Most Advanced Open Source Relational Database. https://www.postgresql.org/, 2025.

[66] PostgreSQL. Server Configuration. https://www.postgresql.org/docs/current/runtime-config-resource.html, 2025.

[67] Yingjin Qian, Marc-André Vef, Patrick Farrell, Andreas Dilger, Xi Li, Shuichi Ihara, Yinjin Fu, Wei Xue, and André Brinkmann. Combining buffered {I/O} and direct {I/O} in distributed file systems. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST'24)*, pages 17–33, 2024.

[68] Yifan Qiao, Chenxi Wang, Zhenyuan Ruan, Adam Belay, Qingda Lu, Yiying Zhang, Miryung Kim, and Guoqing Harry Xu. Hermit:{Low-Latency},{High-Throughput}, and transparent remote memory via {Feedback-Directed} asynchrony. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI'23)*, pages 181–198, 2023.

[69] Ragesh Thottathil. Cxl verification using portable stimulus. https://dvcon-proceedings.org/wp-content/uploads/1070.pdf, 2023.

[70] Rambus. Compute Express Link (CXL): All you need to know. https://www.rambus.com/blogs/compute-express-link/#protocols, 2024.

[71] RocksDB. RocksDB. https://rocksdb.org/, 2025.

[72] Samsung Electronics Co. Expanding the Limits of Memory Bandwidth and Density: Samsung's CXL Memory Expander. https://semiconductor.samsung.com/news-events/tech-blog/expanding-the-limits-of-memory-bandwidth-and-density-samsungs-cxl-dram-memory-expander/, 2022.

[73] ServTheHome. 16 Arm Cores Marvell Structera-A. https://www.servethehome.com/this-cxl-memory-controller-has-16-arm-cores-marvell-structera-a/, 2024.

[74] ServTheHome. Limit IRQ holdoff latency impact. https://lore.kernel.org/lkml/235e470402ee86a6bf94cfe73543b0c0f5a42071.camel@gmx.de/, 2024.

[75] L Shalev, H. Ayoub, Bshara a N., Y. Fatael, O. Golan, A. llany, A. Levin, Z. Machulsky, K. Milczewski, M. Olson, V. Priescu, S. Rajagopal, and A. Saidi. The tail at amazon web services scale. *IEEE Micro*, 44(5):23–29, 2024.

[76] SMART Modular Technologies. FSMART Modular Technologies Introduces New Family of CXL Add-in Cards for Memory Expansion in High Performance Servers. https://www.smartm.com/media/press-releases/SMART_Modular_Technologies_Introduces_New_Family_of_CXL_Add-in_Cards_for_Memory_Expansion_in_High_Performance_Servers, 2025.

[77] SMART Modular Technologies. SMART's CXA-8F2W. https://www.smartm.com/product/cxl-aic-cxa-8f2w, 2025.

[78] SQL Server Central. Understanding Cosmos DB. https://www.sqlservercentral.com/blogs/understanding-cosmos-db, 2018.

[79] Yan Sun, Jongyul Kim, Zeduo Yu, Jiyuan Zhang, Siyuan Chai, Michael Jaemin Kim, Hwayong Nam, Jaehyun Park, Eojin Na, Yifan Yuan, et al. M5: Mastering page migration and memory management for cxl-based tiered memory systems. In *Proceedings of the ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'25), Volume 2*, pages 604–621, 2025.

[80] Yan Sun, Yifan Yuan, Zeduo Yu, Reese Kuper, Chihun Song, Jinghan Huang, Houxiang Ji, Siddharth Agarwal, Jiaqi Lou, Ipoom Jeong, et al. Demystifying cxl memory with genuine cxl-ready systems and devices. In *Proceedings of the Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'23)*, pages 105–121, 2023.

[81] SUSE Labs. Linux memory management. https://lpc.events/event/11/contributions/896/attachments/793/1493/slides-r2.pdf, 2022.

[82] SUSE Labs. Tuning the memory management subsystem. https://documentation.suse.com/sles/15-SP6/html/SLES-all/cha-tuning-memory.html, 2024.

[83] The Kernel Development Community. Memory Allocation Guide. https://www.kernel.org/doc/html/v6.14-rc2/core-api/memory-allocation.html, 2023.

[84] The Linux Kernel. The memory management in linux. https://docs.kernel.org/admin-guide/mm/concepts.html, 2025.

[85] Marcel Weisgut, Daniel Ritter, Pınar Tözün, Lawrence Benson, and Tilmann Rabl. Cxl memory performance for in-memory data processing. *VLDB Endowment*, 18(9):3119–3133, 2025.

[86] Giorgos Xanthakis, Antonios Katsarakis, Giorgos Saloustros, and Angelos Bilas. vlsm: Low tail latency and i/o amplification in lsm-based kv stores. *arXiv preprint arXiv:2407.15581*, 2024.

[87] Lingfeng Xiang, Zhen Lin, Weishu Deng, Hui Lu, Jia Rao, Yifan Yuan, and Ren Wang. Nomad:{Non-Exclusive} memory tiering via transactional page migration. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI'24)*, pages 19–35, 2024.

[88] YCSB. Cloud Serving Benchmark (YCSB). https://github.com/brianfrankcooper/YCSB/, 2025.

[89] Sungmin Yun, Hwayong Nam, Kwanhee Kyung, Jaehyun Park, Byeongho Kim, Yongsuk Kwon, Eojin Lee, and Jung Ho Ahn. Clay: Cxl-based scalable ndp architecture accelerating embedding layers. In *Proceedings of the ACM International Conference on Supercomputing (ICS'24)*, pages 338–351, 2024.

[90] Timothy Zhu. *Meeting tail latency SLOs in shared networked storage*. PhD thesis, Google, 2015.

ARTIFACT EVALUATED
USENIX
AVAILABLE

Finding NEMO: Nimble and Expressive Memory Observability

Shihang Li[1,*] Matthew Giordano[1,*] Tushar Garg[2,†] Rohan Kadekodi[1]
Daniel S. Berger[1,3] Baris Kasikci[1] Thomas Anderson[1] Simon Peter[1]
[1]*University of Washington* [2]*Meta* [3]*Microsoft Azure*

Abstract

Modern datacenter servers increasingly deploy heterogeneous, multi-tier memory hierarchies. For these new architectures, OSes depend on measurements of memory usage to make intelligent placement and control decisions. However, existing hardware and software mechanisms for tracking memory usage on these systems require difficult tradeoffs between coverage, timeliness, granularity, flexibility, and overhead.

We present NEMO, a nimble and expressive hardware memory telemetry engine for server memory controllers (MCs) that gives OS subsystems policy-specific views of memory behavior. NEMO enables flexible telemetry rules that filter memory operations, map accesses to counters, and apply simple updates to per-counter state. We prototype NEMO on an FPGA-based CXL-attached memory expander. Evaluating three diverse use cases, we show that NEMO provides higher-fidelity signals at substantially lower CPU overhead across a range of state-of-the-art memory management systems: it speeds up HeMem's reaction to hot-set changes by 5×, accelerates THP splitting in MEMTIS by 10.4×, and detects noisy neighbors in Linux with 350× lower overhead. These telemetry improvements yield up to 1.7× higher throughput and 23% lower latency across key-value stores and databases.

1 Introduction

Modern datacenter server memory hierarchies have become increasingly complex with cache-coherent multi-socket DRAM, accelerator unified virtual memory [19], and Compute Express Link (CXL)-attached memory expanders [23, 43, 56] that may themselves implement internal tiering [71], compression [15], and pooling [39, 72]. A cache miss or uncached streamed memory operation can experience very different latency and bandwidth depending on where and how it is served [62]. At the same time, memory has become an expensive server component requiring careful management, accounting for 40-50% of server costs [43, 57].

To use these various types of memory effectively, OSes and runtimes need to actively manage and control the placement and use of memory [17, 35, 36, 43, 48, 63, 64, 67, 71, 73]. CXL is now deployed in major cloud platforms [23], where multi-tenancy demands flexible and evolving memory policies: some tenants require hot-set tracking, others need to identify sparsely accessed regions inside huge pages, and operators must attribute bandwidth to catch noisy neighbors. Early operational experience confirms that the lack of efficient CXL runtime telemetry limits the ability of cloud providers to attribute slowdowns to individual tenants and mitigate interference [7]. Small changes in placement or migration decisions can produce large shifts in application performance and interference between co-located tenants [36, 62, 64, 67]. Finally, memory management must incur low overhead to leave more resources available to end users [49, 60].

Existing mechanisms for gathering memory usage information pose awkward tradeoffs to the OS developer. Page-level access bit tracking is flexible, but can be expensive when tracking usage across large memories and multiple memory tiers [48]. Hardware cache miss sampling can reduce overheads, but comes at the cost of reduced precision [2, 68]. Hardware counters on CPUs and memory controllers typically aggregate events at the level of channels, banks, or sockets, revealing overall pressure but not which virtual pages, objects, or tenants are responsible [46, 53]. In response, recent CXL/OCP specifications [15] and research systems [55, 73] have proposed fixed-function hotness tracking and bandwidth statistics, but these provide only rigid, design-time semantics that cannot adapt as datacenter policies and memory hierarchies evolve (§2.2).

This paper proposes NEMO, a hardware-software co-designed memory telemetry engine that addresses these problems. On the hardware side, NEMO is a lightweight extension to memory controllers (MCs): each MC is augmented with a small, off-data-path telemetry pipeline and a modest SRAM budget for telemetry state. NEMO pipelines let the OS define telemetry rules that filter relevant memory operations, map access sites via a small MC-resident table to telemetry state, and apply simple update operations to per-counter state. On the software side, NEMO exposes an OS-facing programming model and Linux driver that gives the OS flexi-

*Equal contribution.
†Work done at Microsoft.

ble, policy-driven control over what telemetry to track, while preserving high fidelity and full coverage. OS kernel subsystems can install and evolve their own sets of rules via a common controller-side engine. The controller executes these rules on each memory operation to maintain policy-specific state over OS-chosen granularities and aggregates (e.g., page ranges, tenants, or allocation sites), as opposed to predefined aggregates offered by fixed-function hardware counters.

Designing NEMO raises three concrete challenges. First, MCs see requests at very high rates, forcing us to represent rich policies within a small, carefully managed SRAM state budget. Second, for flexibility, NEMO must support observations at multiple software-defined spatial granularities concurrently. Third, the OS must configure and read out NEMO state efficiently, without turning the memory controller into a new CPU bottleneck. We will show how NEMO's match-update pipeline, translation scheme, and driver interface address these constraints while preserving full coverage.

We prototype NEMO in FPGA on a CXL-attached memory expander development kit [3] and integrate it with existing OS memory management mechanisms, anchoring our design in the CXL Type-3 deployment model that cloud providers are already rolling out today [23]. The prototype monitors every CXL memory operation and exports compact aggregates to the kernel, enabling near-real-time views of hot sets and sparse access patterns. An implementation in server integrated memory controllers (IMCs) would further reduce overheads by providing fast-tier coverage. Recent Xeon processors, for example, already support delivering memory responses to additional consumers without affecting cache performance (e.g., Direct2Core [26]).

NEMO sits at a practical vantage point in the memory hierarchy. MCs already host QoS and monitoring features, making them a natural place to host a small, flexible telemetry engine. In contrast, cache-centric designs such as täkō [51] require intrusive changes to core microarchitecture, making them hard to deploy in existing processors, while in-DIMM approaches offer only very limited logic and vendor-specific control paths, making rich telemetry difficult. Because the NEMO pipeline runs alongside, rather than on, the memory datapath, it does not add latency to cache hits or DRAM/CXL accesses; it only observes request metadata and maintains aggregates in local SRAM. Compared to existing telemetry approaches, this hardware offload raises fidelity and reduces CPU overhead, allowing the OS to reason about complex, multi-tier memory hierarchies using precise, up-to-date signals within a tight compute and SRAM budget.

We make four contributions:

1. We characterize the challenges of memory observability in heterogeneous, multi-tier server memory systems and study the tension between fidelity, flexibility, and resource constraints (§2).
2. We present NEMO, a lightweight MC-resident memory telemetry engine that exposes filtering, translation, and aggregation abstractions, enabling OS subsystems to install telemetry rules that express a wide range of policies within realistic compute and SRAM budgets (§3).
3. We build a working NEMO prototype[1] on a CXL-attached FPGA memory expander development kit and integrate it with existing OS memory management mechanisms in Linux, HeMem [48], and MEMTIS [36] (§4).
4. We evaluate NEMO on three diverse use-cases, showing that NEMO improves policy effectiveness and CPU efficiency over existing telemetry mechanisms (§5). Across these cases, NEMO-based policies improve tail latency, throughput, and isolation compared to state-of-the-art baselines that rely on sampled PMU data or coarse per-core counters, while consuming a modest fraction of the CPU cycles used by host-only telemetry. In particular, the NEMO telemetry information provides 2-10× faster identification of hot (§5.1) and skewed (§5.2) pages than alternative approaches with similar CPU overhead, and provides bandwidth tracking within 0.1% of ground truth with ~350× lower overhead (§5.3), resulting in up to 1.7× throughput improvement with a 23% latency reduction across FlexKVS [31], FASTER KV [14], and Silo [59].

2 Background

We characterize memory observability into five dimensions that directly affect OS policy performance in the datacenter environment:

1. *Coverage*: the fraction of memory accesses that are directly observed.
2. *Timeliness*: the delay between a change in memory-access behavior and when telemetry reflects that change.
3. *Granularity*: the smallest unit to which memory accesses can be attributed (e.g., cache line, page, region, or tenant).
4. *Flexibility*: the degree to which the OS can define which accesses are tracked and how they are aggregated.
5. *Overhead*: the system resources consumed to collect, process, and export telemetry.

We use these dimensions to survey existing memory telemetry mechanisms. Software techniques provide flexibility but achieve limited coverage and timeliness at high CPU cost (§2.1), while hardware offloads offer efficient, high-coverage signals only at coarse granularities and with fixed semantics chosen before tape-out (§2.2). Prior to NEMO, no existing mechanism simultaneously delivers on all five.

2.1 Software Telemetry: Flexible but Costly

Many memory management policies must infer access behavior indirectly using software-based aggregation built on hardware signals. These approaches provide flexibility, but

[1] NEMO is open source at https://github.com/vic-lsh/nemo.

incur high CPU overhead while achieving limited coverage and timeliness.

One of the simplest techniques is *soft page faults*: the OS deliberately unmaps pages (or marks them read-only) and infers access patterns from the resulting faults [1,5,8,33,43,61]. This mechanism can, in principle, capture every access. However, fault handling has high CPU overhead in practice, and so systems can only apply it to a small subset of memory [43], limiting coverage. Granularity is also limited to hardware-defined page sizes.

To reduce overhead, systems often rely on *page-table scanning*. The OS periodically walks page tables and inspects accessed and dirty bits to collect per-page read and write activity [1,17,29,42,67]. This eliminates the direct cost of faults but introduces two limitations. First, it sacrifices timeliness: accesses are only observed at each scan interval. For example, production systems limit the scan frequency to every 30 seconds [17]. Second, it lacks coverage and granularity: each page yields only one bit of information per interval, providing no notion of access intensity within the interval or at a granularity other than the page. Increasing scan frequency improves timeliness and coverage, but quickly drives CPU overhead to impractical levels [39,48,71]. CPU overhead can be decreased by mapping at a coarser page granularity, but this decreases fidelity.

A third approach combines hardware sampling with software aggregation. For example, Intel Processor Event-Based Sampling (PEBS) [27] supports sampling cache misses and lets the OS reconstruct access patterns from the sampled access locations [17, 36, 48]. This improves granularity but reintroduces a tradeoff between coverage and timeliness on one hand, and CPU overhead on the other. This occurs in two ways: 1) A higher sampling rate improves coverage but increases CPU cost for sample processing. 2) A higher sample collection frequency improves timeliness but also increases CPU cost due to increased interrupt rates. Prior work [73] shows that sampling 10% of all cache misses leads to an application slowdown of more than 50%. In practice, systems sample at a much lower rate to limit CPU overhead [17, 48]. Being Performance Monitoring Unit (PMU)-based, PEBS also misses important DRAM traffic (cf. §2.2). Finally, PEBS does not support profiling VM guests from the host [66], making it non-viable for public cloud operators.

Across all three techniques, the same pattern emerges: flexibility is achieved via software processing. However, fidelity is limited by how much CPU overhead the system is willing to tolerate, forcing the OS to trade coverage, timeliness, and granularity against overhead. As a result, even state-of-the-art profilers that carefully manage sampling and multiplexing still pay non-trivial CPU overheads and cannot run continuously at high rates without perturbing the workload [37].

2.2 Hardware Offloads: Efficient but Rigid

Modern CPUs and memory controllers (MCs) expose hardware counters such as Intel's Memory Bandwidth Monitoring (MBM) [46] and integrated MC statistics [27]. These provide low-overhead, timely, and high-coverage telemetry, but only at the coarse granularities and semantics chosen by hardware designers. For example, Intel's PMU tracks per-core LLC misses and socket-level MC DRAM activity, missing important traffic such as prefetches, non-temporal accesses, or the ability to attribute memory activity to a core or process. Intel's MBM hardware counters can attribute memory bandwidth to processes, but they have known accuracy issues [53] and does not offer options such as filtering reads or writes. Research approaches [55, 73] go further by tracking hot pages or cache lines in hardware, but they fix telemetry configuration, granularity, and coverage (e.g., top-5 hot pages and cache lines [55]) at hardware design time. As a result, hardware telemetry performs well for the specific use cases that it was built for, but sacrifices flexibility. When policy needs to evolve, the OS is forced to fall back to software aggregation.

3 NEMO: Nimble and Expressive

NEMO is a hardware-software co-designed memory telemetry engine that gives the OS a flexible, high-fidelity view of server memory traffic at the memory controller. On the hardware side, NEMO is a modest change: each MC is augmented with a small, off-data-path processing pipeline, coupled with a small SRAM state array, to enable tracking memory operations without adding latency. On the software side, the OS programs these pipelines via a narrow interface and retrieves pre-aggregated state, keeping CPU overhead low. NEMO's goal is to provide high coverage, timely updates, fine-grained attribution, and OS-defined metrics, while incurring low CPU and MC overhead. We now describe the constraints this setting imposes and the abstractions NEMO provides to remain both expressive and practical.

Challenge: enabling flexibility with full coverage. Full coverage limits per-request computation: telemetry processing must keep up with controller bandwidth, ruling out designs with unbounded or data-dependent processing latency. It also restricts where telemetry state can reside: updates cannot add memory traffic or interfere with application access, yet every request may modify state. Storing telemetry in DRAM would require two extra accesses per update and severely reduce bandwidth, so all telemetry must fit within a small SRAM near the controller, requiring careful allocation across granularities and regions to support diverse OS needs. Lastly, modern servers interleave physical memory across MCs at fine granularity; to maintain full coverage, NEMO must track telemetries across MCs and provide OS-level aggregation. As a result, NEMO must create nimble abstractions that sat-

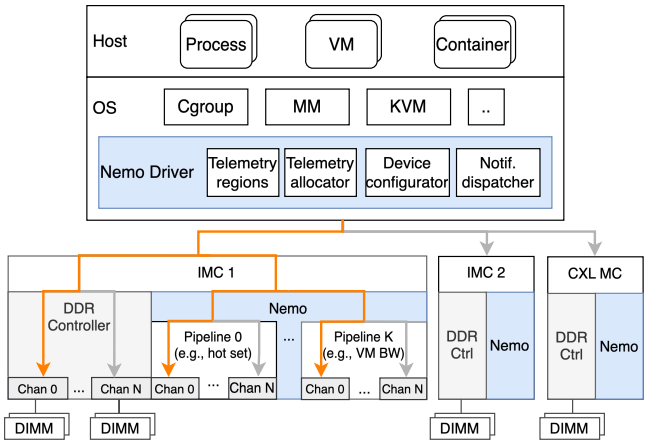


Figure 1: NEMO augments each MC with a programmable telemetry engine. Each memory request is broadcast to all NEMO pipelines (monitoring only), while the memory operation itself is routed to a single channel (one possible route shown in orange). The NEMO driver orchestrates NEMO instances to track telemetries requested by OS subsystems.

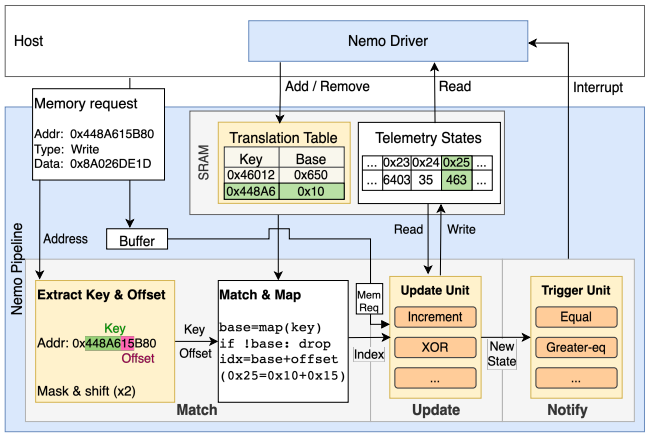


Figure 2: NEMO pipeline and host interactions. Each pipeline implements a fixed match-update-notify structure; highlighted tiles are configured per telemetry.

isfy line-rate, SRAM, and aggregation constraints while still supporting diverse telemetry use cases.

System overview. Figure 1 shows NEMO's architecture. Each memory controller is augmented with a small telemetry engine and exposes it to the OS through a driver. Each engine contains a number of telemetry pipelines, each of which can be configured for a different telemetry. As memory requests arrive, NEMO taps their headers off the data path and feeds them to all telemetry pipelines simultaneously, while the original requests are routed to DRAM. Each NEMO pipeline is capable of snooping all memory channels on the MC. The OS specifies and reads back specific telemetries via the NEMO driver. The driver programs per-controller pipelines, manages telemetry state in SRAM, and returns aggregated views to OS subsystems. For important signals, the OS installs predicates that let NEMO raise interrupts. Memory management policies then act on these high-fidelity signals.

Running example. Consider a telemetry that tracks accesses to selected 2 MiB pages (hugepages). The OS asks NEMO to maintain one counter per tracked page. When a memory request for a tracked page reaches the memory controller, NEMO increments that page's counter. The OS periodically reads these counters to estimate page hotness. This is the simplest use of NEMO's pipeline. The same structure also supports different filters, granularities, update operations, and notification predicates.

3.1 NEMO Telemetry Pipeline

To maintain full coverage with predictable latency, NEMO leverages OS-configurable fixed-function pipelines inside each memory controller. As shown in Figure 2, NEMO telemetry pipelines follow a match-update-notify structure, inspired by the reconfigurable match table architecture [11] used in programmable network switches and interface cards. Each pipeline applies this structure to all memory channels. To carry out this work, each pipeline owns (1) a *translation table* that maps each incoming memory request via its physical address to a telemetry state index and (2) a set of *telemetry states* that hold its per-channel telemetry counters. First, a pipeline matches all or part of the physical memory address to produce an index into the telemetry state. If there is a match, the pipeline's update stage updates the selected telemetry state using a configured operator and, optionally, any part of the memory request as operand. Finally, the pipeline's notify stage may optionally trigger an interrupt to notify the host of notable state updates, using a comparison operator. For page hotness tracking, the match stage maps the request to the specific counter for its 2 MiB page, and the update stage increments that counter. The OS periodically reads the counter array for placement decisions.

NEMO restricts per-request telemetry updates to associative and commutative operations. This is for scalability within and across controllers. Commodity servers often need multiple MC instances (e.g., one or more per socket or SNC partition [24], as well as CXL memory controllers behind the PCIe root complex). Physical memory access is interleaved across controllers at fine granularity for performance. Within one MC, a telemetry may itself be implemented using multiple NEMO pipelines, and each channel within a pipeline owns some telemetry state SRAM banks. Associative and commutative updates let each NEMO instance combine concurrent updates locally and let the OS merge per-MC states in any order to reconstruct a global view. Each pipeline stage is replicated per memory channel in order to maintain full bandwidth within a pipeline.

We now describe each telemetry pipeline stage and the host interface for reading telemetry state.

Match stage. The match stage maps each memory request to the corresponding telemetry state index. The main design challenge is that the mapping must be flexible enough to support OS-defined granularities and semantics, yet simple enough to compute at line rate.

To address this challenge, NEMO adopts a mask-shift-add design that uses simple, fixed-latency arithmetic and a compact translation table to support flexible aggregations (e.g., from cache line to multi-GiB regions, from per-page to per-tenant). At a high level, the match stage matches a request to a telemetry state index, identifying the state to update in the next stage. For page-hotness tracking, NEMO derives a key from the request's physical address by masking and shifting away the offset within a 2 MiB page. The key indexes the translation table, which returns the counter index for that page. If the key is missing or invalid, NEMO drops the request.

The translation table is itself stored in banked SRAM and gives flexibility in indexing the telemetry state SRAM. In general, NEMO derives the key using two OS-specified parameters: (1) a bitmask that selects the relevant address range, and (2) a right shift that sets the primary region size. These two parameters let keys identify *primary regions* (e.g., a 2 MiB or 4 GiB region), which must have power-of-two sizes and be aligned to their size boundaries.

Keys index the translation table, which returns the *base indices*. Keys may map one-to-one to states (e.g., one counter per page) or many-to-one (e.g., all pages in a tenant share a telemetry state).

For simple telemetries such as per-page hotness, the base index is the final state index. For telemetries that subdivide a primary region, the translation table's output—the base index—is added to an *offset* to produce the final state index. With offsets, a memory request matches on a sub-region within the request's primary region. The OS supplies a secondary mask and right shift that determine sub-region size. NEMO takes the designated masked bits of the physical address, shifts them, and obtains the sub-region offset, which is added to the base index to get the final state index. When no subdivision is needed, the OS sets the secondary mask to zero. Figure 2's Match Stage illustrates the full base-index-plus-offset arithmetic. This simple arithmetic lets the OS trade spatial fidelity for coverage: the same SRAM budget can either track a small region at fine granularity or maintain coarse-grained counters over large regions.

Lastly, different pipelines can filter and translate at different granularities. Pipelines can be configured to drop reads or writes, giving more flexibility to the OS.

Update stage. The update stage takes the telemetry state index and the memory request as input and performs a read-modify-write on the state selected by the match stage, using the update operator chosen for the telemetry (Table 1). The hotness telemetry uses addition: each request adds one to the selected counter. Operators use request fields as operands.

To store telemetry states while maintaining the bandwidth requirements of a memory controller, NEMO leverages banked SRAM. Each channel in each pipeline can be configured by the OS to exclusively own some number of banks. This ensures that bank conflicts never arise and avoids SRAM stranding. SRAM accesses do not stall the NEMO pipeline. Instead, we use value forwarding to protect against data hazards when updating the same SRAM telemetry state consecutively (cf. §4).

Telemetry state access. Each controller advertises a *telemetry region*: a physically contiguous range whose contents are backed by MC SRAM. The NEMO driver reserves these regions in the OS kernel so they are never used as anonymous memory, and knows to use this region to read telemetry states.

On a memory load from the telemetry region, the controller bypasses DRAM and the telemetry pipeline and instead returns the telemetry state from SRAM. To support efficient state resetting, NEMO allows the driver to configure an optional reset operation (any update operator in Table 1 or reset-to-constant) that is executed immediately after returning the telemetry data. Reset-on-read makes each OS poll return per-page access counts for the latest placement interval.

Notify stage. Polling is sufficient for many policies, including page-hotness tracking. For urgent signals, the notify stage allows each pipeline to specify a simple predicate on updated state (e.g., comparison against a threshold). After an update, the notify logic evaluates the predicate and raises an interrupt if it holds. Each controller only sees its slice of traffic, but fine-grained interleaving makes per-controller patterns similar; alerts thus act as hints that the OS should re-check telemetries across controllers.

3.2 NEMO Control Plane

The NEMO driver is the control plane for all interactions with NEMO controllers. It installs telemetries, allocates telemetry state, manages translation entries, exposes telemetry regions, and dispatches notifications.

Telemetry installation. To install a telemetry, NEMO exposes a constrained per-request programming model. Each NEMO *telemetry* is installed by specifying:

1. *Memory request filter*: R/W type and address range;
2. *Translation rule*: maps addresses to telemetry states;
3. A *telemetry update operation* on the selected state;
4. *Notify predicate*: optional notification trigger;
5. *Telemetry read side-effect*: optional state reset.

When an OS service submits a telemetry specification to the NEMO driver, the driver first validates the telemetry, then allocates available pipelines on each relevant MC.

Category	Config	Options
Match	Filter	Request type (R/W/All)
	Base index	Bit mask & shift
	Offset	Bit mask & shift
Update	Update op	One of: +, -, >>, <<, XOR; specify operand.
Notify	Predicate	One of: ==, >=, >, <, <=; specify operand.
Read telemetry	Side effect	One of the update ops, or reset to a configured value.

Table 1: Configuration options in NEMO telemetry rules.

Telemetry state allocation and translation. The NEMO driver uses a slab allocator to allocate the appropriate number of pipelines and telemetry state banks, and then programs the translation table to map physical memory ranges to allocated telemetry state indices. For any allocated state, the OS can dynamically update which regions map to it by adding or removing translation entries. This is helpful for aligning the translation table with OS telemetry semantics (e.g., mapping only a process's pages to telemetry states). The driver validates translation entries by only mapping addresses within the physical memory range that a controller serves. It also tracks how a telemetry's states are allocated across multiple controllers to ensure correct aggregation on telemetry reads.

Reading telemetry. At boot, the NEMO driver discovers each controller's telemetry region, maps it, and removes it from general use. The driver exposes an API to aggregate a telemetry's states across controllers, applying the optionally configured read-side effect. For most metrics, OS services may simply periodically read telemetry state from controllers (polling).

When reading telemetry, the driver reconstructs a global view by reducing per-MC states using the update operator's natural combine rule (e.g., sum per-MC values for addition, XOR for bitwise-XOR, and multiplication for right bit-shifts). Because all update operators are associative and commutative, the driver can compute a logical telemetry state by folding per-MC states.

Notification dispatch. OS services can also install predicates that let NEMO raise interrupts. When a NEMO controller raises an interrupt, it tags the corresponding pipeline. The driver resolves this to a telemetry and invokes its registered callback, delivering alerts to the OS subsystem.

3.3 Putting It Together

We now extend the hotness example with offsets, producing a concrete telemetry for access skew within selected hugepages.

Tracking within-page skew. To measure access skew within selected 2 MiB hugepages, NEMO maintains one counter per 4 KiB basepage. A hugepage-splitting policy can use these counters to find pages whose accesses concentrate in only a few basepages.

```
1  nemo_telemetry_t t = nemo_register_telemetry((nemo_spec_t)
2  {
3    .filter      = {.type=R|W},
4    .translation = {
5      // Track hugepages across all physical memory
6      .primary   = {.mask=NEMO_ALL_MEM, .shift=HP_SHIFT},
7      // Offset: basepage number within the hugepage
8      .sub       = {.mask=HP_BP_MASK,   .shift=BP_SHIFT},
9    },
10   .update      = {.op=NEMO_OP_ADD,    .operand=1},
11   .notify      = {.op=NEMO_OP_NOOP},
12   .read_effect = {.op=NEMO_OP_RESET, .value=0},
13 });
14
15 // Set translation table to track new hugepages
16 void reprogram(uint64_t hp_paddr[], int npages) {
17   for (int i = 0; i < npages; i++)
18     nemo_set_translation(t,
19         /*key=*/        hp_paddr[i] >> HP_SHIFT,
20         /*region_idx=*/ i);
21 }
```

Listing 1: NEMO telemetry for basepage skewness within selected hugepages. The primary region is a 2 MiB hugepage; the sub-region is a 4 KiB basepage.

Listing 1 shows this telemetry. Lines 2–13 install the static telemetry rule `t`: match both reads and writes, compute a hugepage key from the physical address, compute a basepage offset within that hugepage, increment the selected counter, and reset counters on read. Resetting on read makes each poll return basepage access counts for only the last interval.

The OS then chooses which hugepages to track by calling `reprogram()`, which installs a translation entry for `t` per tracked hugepage with `nemo_set_translation(t, key, region_idx)` (lines 15–21). The *key* is the hugepage frame number (`paddr >> HP_SHIFT`), and the *region index* assigns this hugepage a distinct base index in the telemetry state array; the driver picks base indices so different hugepages' 512-counter ranges do not overlap. As before, the basepage number within the hugepage forms the offset that NEMO adds to the base index to select the final per-basepage counter.

Listing 1 is one point in NEMO's configuration space. Translation entries are dynamic: the OS can reprogram any entry to time-multiplex its slots across more regions than fit in SRAM. The per-basepage skewness telemetry in §5.2 uses this to track skewness across all of memory. Translation can be one-to-many, as in this example, where one hugepage maps to 512 basepage counters. Translation can also be many-to-one: the per-tenant bandwidth telemetry in §5.3 maps all of a tenant's pages to a single state. Other telemetries vary request filters, state update operations, read-side effects, and notification predicates within the same match-update-notify structure.

What NEMO can and cannot express. NEMO's per-request semantics—an associative and commutative single-step update to one memory location, optionally followed by a fixed-cycle predicate—cover the broad class of memory observability tasks that aggregate per-access events into per-region or per-tenant counters, including all three use cases in §5. NEMO does not support telemetries that need multiple arithmetic steps or multiple locations per request, unless they can be done by different pipelines operating independently and later combined. For example, computing the variance of a region's access counts must maintain both a running sum and a running sum of squares of the differences from the running sum. Likewise, exact top-K tracking depends on history beyond a single SRAM-resident counter. §6.3 discusses extensions that lift these restrictions.

Summary. NEMO simultaneously provides high coverage, low overhead, flexibility, granularity, and timeliness for memory telemetry. Its off–data-path pipeline and SRAM-resident state avoid perturbing MC latency or bandwidth, while the fixed-function pipeline sustains full coverage at line rate with predictable latency. The pipeline's configurable stages offer flexibility, and the translation table supports use-case-specific aggregations at OS-defined granularities. Finally, the driver's memory-mapped telemetry region and optional alerts offer low-overhead state retrieval and timely reactions. Together, these mechanisms give the OS a flexible, high-fidelity telemetry mechanism that fits within the throughput and resource constraints of modern memory controllers.

4 Implementation

NEMO is prototyped on an Altera Agilex 7 FPGA I-Series Development Kit with CXL 2.0 Type-3 support in hard IP [3]. The FPGA is connected to our host machine via PCIe 5.0 x16, and has 16GiB of onboard DDR4 DRAM. It features 913k logic modules (ALMs), 3.7M ALM registers, and 13k M20K SRAM blocks, summing to 32MiB of usable SRAM blocks. Our design consists of the CXL IP, a DDR4 memory controller, and our NEMO logic, running at 400 MHz with resource usage described in Table 2. The host machine runs an Intel Xeon Gold 6430 CPU, with 32 physical cores and 64 hyperthreads. It contains 2MiB of L2 cache per core and 60MiB of shared LLC cache, and 256GiB DDR5 memory. The CXL-attached DRAM is substantially slower than host DRAM. Using Intel MLC [28], we measure host DRAM at 114.1 ns load-to-use latency and 114.9 GiB/s bandwidth, while the CXL DRAM exhibits 380.3 ns latency and 16.4 GiB/s bandwidth, i.e., roughly 3.3× higher latency and 7× lower bandwidth.

NEMO pipeline. Our FPGA has 2 memory channels corresponding to the parity of a cache line address. Thus, each pipeline must support the match-update protocol on two channels in parallel. Since each channel can issue one read or one write on each FPGA cycle, maintaining an initiation interval of 1 on each channel is crucial to achieving full coverage.

	Full Bitstream	NEMO
Logic Utilization	17%	4.8%
Logic Registers	9.3%	1.7%
M20Ks (SRAM)	14%	4.4%

Table 2: FPGA resource utilization. Full Bitstream refers to the utilization of the entire design, including CXL and DRAM controller logic, whereas NEMO refers to the NEMO logic overhead.

Match stage. The translation table is implemented as a dual-port SRAM, with one port used to read and one used to modify the table. The translation table is currently replicated across each channel, though a true dual-port design could eliminate this extra duplicate. The translation table key is extracted from the memory operation's address via a configurable subtraction and shift sequence. As a prefilter, we also implement range bounds before key lookup. Once the telemetry state base is retrieved, an offset computed from the address is added to get the final telemetry state index.

Update stage. Each channel in the pipeline features its own telemetry state SRAM, implemented via the M20K block RAM primitives on our board. In order to maintain coverage and keep up with memory controller bandwidth, it is crucial that we implement read-modify-write hazard protection without stalling the pipeline. The SRAM primitives offered on the board provide 1-cycle latency reads and writes, meaning that a 1-cycle update operation can use data forwarding as a hazard solution. All of our update operations fit this 1-cycle latency requirement, so forwarding is sufficient to avoid data hazards. If a future update operation takes more than 1 cycle to compute, we can double the clock speed and halve the initiation interval to ensure full line-rate. If that is not an option, we could further split the N existing channels into $N \times M$ virtual channels using a round robin load balancing strategy per physical channel, allowing an initiation interval of M cycles to perform update operations at the cost of M times the SRAM.

Notify stage. Notifications are not currently implemented, but are simulated by polling counters every 1 ms on a core until a counter threshold is met.

Telemetry State Access. We modify the provided memory controller instantiation to advertise a 32GiB address range instead of its 16GiB DDR capacity. The prototype intercepts requests to the upper half of the address space for telemetry state retrieval. The driver is responsible for ensuring that this address range is only used by the driver to read telemetry.

To extract telemetry, NEMO interposes on memory requests and intercepts those that belong to the state retrieval address range. Since we expose telemetry as a standard memory region rather than an MMIO region, the driver must be careful to invalidate old cached telemetry information. However, we chose this as it improves performance over MMIO, giving us prefetchable, 512 bits/cycle/channel of telemetry output, matching DRAM throughput. Our implementation does not yet fully pipeline telemetry state access.

To configure each pipeline, each pipeline exposes control registers to the host via MMIO, allowing the NEMO device driver to configure all aspects of its functionality. This operates at 64 bit/cycle data width. We chose MMIO because these operations are typically not performance-critical and simplify the implementation by not having to intercept writes.

Resource budget. Each pipeline carries an 8,192-entry translation table (~14 KiB at 14 bits/entry) for the match stage and a per-channel $8{,}192 \times 64$-bit state array (128 KiB across the FPGA's 2 channels) for the update stage, totaling ~150 KiB. Across the full bitstream, NEMO uses 4.4% of the FPGA's M20K SRAM and 4.8% of its logic (cf. Table 2); the largest deployment we evaluate, MEMTIS-NEMO (§5.2), uses 8 such pipelines for ~1.2 MiB of telemetry SRAM. The headroom can support larger tables, deeper state, or more pipelines.

5 Evaluation

We evaluate NEMO by integrating it into three OS responsibilities that rely on memory telemetry: detecting application hot-set changes for tiering (§5.1), identifying transparent hugepage (THP) split candidates (§5.2), and detecting and throttling memory-bandwidth noisy neighbors (§5.3). Compared to state-of-the-art systems in each setting, NEMO delivers ~5× faster adaptation to hot set changes, accelerates THP splitting by 10.4×, and detects noisy neighbors in Linux with 350× lower overhead. These improvements yield up to 1.7× higher throughput and 23% lower latency across key-value stores and databases—core building blocks in modern clouds that are often deployed as large, memory-resident services that are both latency and cost-sensitive. Their access patterns stress exactly the phenomena our mechanisms target, making them representative workloads to evaluate.

For each use case, we describe a NEMO telemetry, the OS integration, and measure the resulting improvements in performance, reaction time, and CPU overhead compared to state-of-the-art telemetry approaches. Together, these results show that NEMO's match-update-notify pipelines with OS-controlled translation can address diverse memory telemetry challenges across a wide range of memory management systems and applications.

```
nemo_telemetry_t t = nemo_register_telemetry((nemo_spec_t)
{
  .filter      = {.type=R|W},
  .translation = {
    .primary   = {.mask=NEMO_ALL_MEM, .shift=HP_SHIFT},
    // no sub-region: one counter per hugepage
  },
  .update      = {.op=NEMO_OP_ADD,   .operand=1},
  .notify      = {.op=NEMO_OP_NOOP},
  .read_effect = {.op=NEMO_OP_RESET, .value=0},
});

// Map each tracked hugepage to its own counter slot
for (i = 0; i < n_hp; i++)
  nemo_set_translation(t,
      /*key=*/        hp_paddr[i] >> HP_SHIFT,
      /*region_idx=*/ i);
```

Listing 2: NEMO telemetry for per-hugepage hot-set tracking. One counter per tracked hugepage; no sub-region offset.

5.1 Hot Set Phase Change Recovery

Detecting application hot set changes is crucial for application performance in tiered, memory-capacity-constrained environments. Existing systems rely on sampled memory telemetry, forcing a tradeoff between sample rate and CPU overhead; in contrast, NEMO captures a complete view of memory accesses, reacts to hot set changes 5× faster, and improves application throughput by up to 1.69×, all while maintaining similar CPU overhead.

NEMO integration. We integrate NEMO telemetry into HeMem [48], a userspace memory tiering system. HeMem uses Intel Processor Event-Based Sampling (PEBS) to track page hotness by sampling LLC misses at a 0.02% rate. Each second, HeMem processes new samples by locating the corresponding page struct and incrementing its access counter, before migrating hot data to the fast tier and cold data to the slow tier.

With NEMO, HeMem sets up NEMO to update a per-hugepage counter on *every* LLC miss. This is shown in Listing 2. The counter is reset on read so each poll returns counts for the last interval. HeMem then scans the telemetry array to update each page struct's access counter by the NEMO-reported count.

Evaluation setup. We compare HeMem with NEMO telemetry (HeMem-NEMO) against HeMem with PEBS (HeMem-PEBS). As discussed in §4, NEMO only observes the slow memory tier, so both setups still use PEBS to sample the fast tier. To make hotness scores comparable across tiers, we scale NEMO access counts by the PEBS sampling rate and otherwise leave HeMem's tiering policy unchanged.

We first evaluate HeMem-NEMO and HeMem-PEBS using FlexKVS [31], an in-memory key-value store. We run FlexKVS with 6 threads, 32B keys, 1024B values, and 1:2 fast:slow memory ratio (24 GiB total); 20% set of keys form a hot set which are accessed 90% of the time. To stress time-

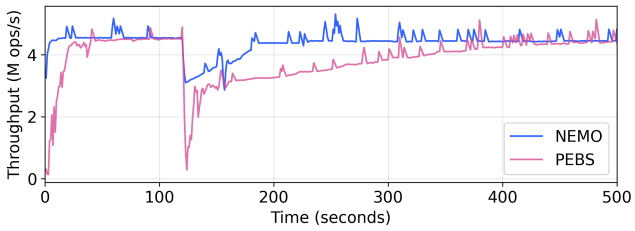


Figure 3: FlexKVS timeline with hot set phase change, featuring HeMem's default PEBS sample rate of 0.02%. NEMO is able to recover to steady state 5× faster than PEBS.

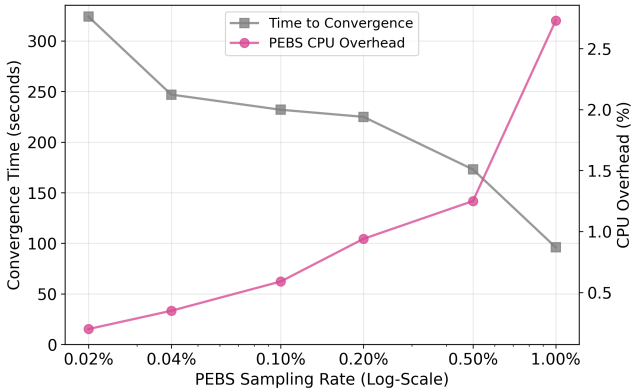


Figure 4: Hot set migration time and CPU overhead with different PEBS sampling rates in HeMem. NEMO takes 67 seconds to converge with 0.89% CPU overhead.

liness of hot-set detection, we engineer a complete hot-set shift: the initially hot keys become cold and a disjoint set of keys become hot. Since FlexKVS's working set spans both fast and slow memory tiers, HeMem migrates pages between tiers based on its hotness estimates. We track FlexKVS performance over time and compare how quickly each variant converges after the hot-set shift.

We then study object-size sensitivity using FASTER KV [14], a concurrent key-value store and cache used in Azure Durable Functions [12]. We choose the number of items and value size so that each experiment fills both fast and slow memory: total memory used (keys, values, metadata) is 30 GiB with a 1:1 fast:slow ratio. FASTER KV runs with 16 client threads, each pinned to its own core, issuing operations according to YCSB workload B [16] (95% reads, 5% writes). We measure aggregate throughput over 5 minutes, reporting the median per-second throughput after a 5-minute warmup. Keys follow a Zipf distribution tuned to approximate an 80/20 rule (80% of accesses target the hottest 20% of keys), mimicking production KV stores [6].

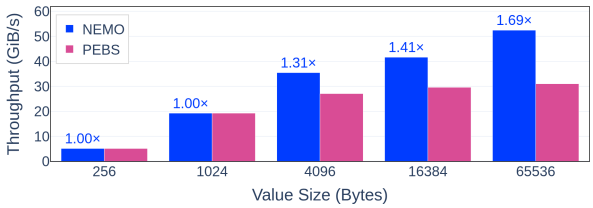


Figure 5: FASTER KV throughput across various value sizes. HeMem-NEMO identifies the hot set with larger value sizes, which PEBS is unable to do because of the prefetcher. There is no hot set with smaller value sizes.

5.1.1 Results

Hot set churn. Figure 3 shows FlexKVS throughput when the entire hot set shifts two minutes into the run. HeMem-NEMO recovers within 67 s, bounded by HeMem's migration bandwidth. HeMem-PEBS takes 324 s at HeMem's default 0.02% sampling rate. With PEBS, time to convergence depends strongly on the sampling rate (Figure 4): low rates reduce CPU cost but delay detecting the new hot set, while high rates improve timeliness at the expense of substantial sampling overhead. Raising the sampling rate beyond 1% is not feasible because it degrades application performance, as shown by prior work [73]. Due to this tradeoff, the gap between HeMem-NEMO and HeMem-PEBS cannot be closed, even in the limit. NEMO eliminates the tradeoff by offloading observation to the memory controller, providing full-fidelity hotness signals at 0.89% CPU overhead.

Value size sensitivity. Figure 5 shows FASTER KV throughput as we vary value size. For small values, HeMem-NEMO and HeMem-PEBS perform similarly: many objects fit in each huge page, so accesses are effectively uniform at page granularity and there is little page-level hot-set signal to exploit. As value sizes grow, HeMem-NEMO performance improves while HeMem-PEBS stagnates. Accesses to larger values become increasingly prefetch-friendly. These prefetches generate DRAM traffic but do not produce PEBS samples [27]. PEBS misses several types of traffic, including prefetches and non-temporal accesses; in contrast, NEMO captures all memory traffic.

With larger values with NEMO, a few hot values dominate each page. NEMO's per-hugepage counters accurately capture this skew, allowing HeMem-NEMO to keep the working set in the fast tier. Overall, NEMO enables higher throughput for skewed, large-value workloads without tuning sample rates or incurring higher CPU overhead.

5.2 THP Page Split Candidate Selection

We next evaluate NEMO on a tiering system that supports transparent huge page splitting. Linux transparent huge

page system (THP) [58] automatically coalesces base pages into 2 MiB huge pages to reduce TLB pressure and page-management overheads. However, THP can cause *hugepage bloat* [35,36,38]: workloads that touch only a few base pages within each hugepage still pin the entire 2 MiB in the fast tier. In tiered memory, where fast-tier capacity is scarce, splitting a *skewed* hugepage, whose accesses are concentrated on a small subset of base pages, allows hot base pages to remain in the fast tier while demoting cold base pages to the slow tier, freeing space for other hot data. We evaluate the benefits of using NEMO to reduce hugepage bloat in a tiered memory system: NEMO finds 2× more page-split candidates with up to 10.4× reduction in detection time, with up to 1.13× better application performance.

NEMO integration. We integrate NEMO into MEMTIS [36], a memory tiering system that modifies Linux v5.15 to support hugepage splitting. Our integration uses the telemetry described in Section 3.3 to simultaneously track hugepage access counts and skewness. Instead of processing PEBS samples, MEMTIS-NEMO reads hugepage access counters directly from NEMO and applies its original policy: MEMTIS ranks pages by access count and classifies the highest-ranking pages that fit in the fast tier as hot. In our prototype, NEMO sees only CXL-attached memory, so we retain MEMTIS's original PEBS-based mechanism for the local DRAM tier.

Because NEMO's SRAM capacity limits the number of basepage counters, we time-multiplex skewness tracking across hugepages. Our telemetry instantiates eight NEMO pipelines: one tracks hugepage access counts and seven each track 16 hugepages at basepage granularity, allowing skewness tracking for 112 hugepages simultaneously. MEMTIS-NEMO extends MEMTIS's mechanism of grouping hot and cold pages with Linux's active/inactive LRU lists. Every 500ms, MEMTIS-NEMO reuses MEMTIS's PEBS sampling thread to program NEMO to track the next 112 hugepages from one of the two lists, using `reprogram()` from Listing 1. Scanning the inactive list is necessary because a hugepage that looks cold overall can still contain hot basepages; consecutive scans alternate between the active and inactive lists. The basepage counters then feed MEMTIS's skewness score. A full sweep takes $(|\text{active}| + |\text{inactive}|)/112 \times 500\,\text{ms}$; §6.1 analyzes how this time-multiplexing approach scales with memory size.

Evaluation setup. To stress fast-tier memory efficiency, we vary the fast-to-slow memory capacity ratio from 1:2 to 1:8. As the fast tier becomes more constrained, the penalty of not splitting skewed hugepages increases, since promoting an unsplit skewed hugepage consumes a disproportionately large share of fast-tier capacity. Our question is whether MEMTIS-NEMO's benefit grows as the fast tier shrinks, because NEMO can identify skewed pages to split more accurately and quickly.

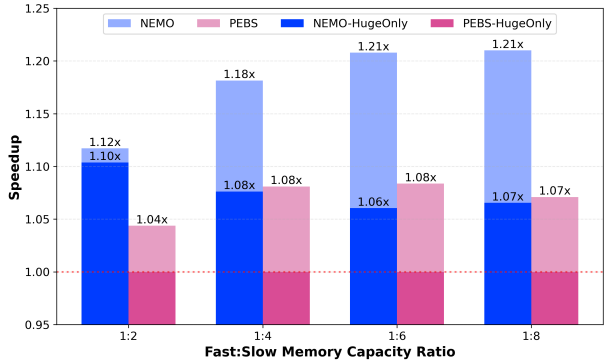
Our prototype can only observe telemetry over CXL memory, so memory allocated to the fast tier and never migrated would not be visible to NEMO. To give NEMO an opportunity to collect telemetry on all pages, we cap the application's memory usage at 16 GiB (the CXL capacity in our prototype) and modify MEMTIS-NEMO to allocate from CXL first and pin memory on CXL for 180 seconds. This is a conservative setup for MEMTIS-NEMO: MEMTIS-PEBS allocates from the fast tier by default and begins migrating immediately, giving it a 180-second head start. Our experiment uses Silo [59], an in-memory database engine. Following prior work [36], we run the YCSB-C workload [16] with a Zipfian distribution, 90M keys, and key and value sizes of 64 B and 100 B, respectively. We confirm that Linux THP allocates over 99% of Silo's memory on huge pages. We run Silo for 1,500 seconds on 20 CPU cores and report performance after a 500-second warmup period, allowing application performance to stabilize after migration and page splitting.

5.2.1 Results

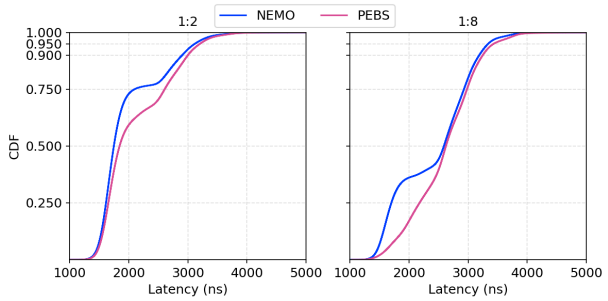
Silo performance. Figure 6a shows Silo's throughput speedup when using MEMTIS-NEMO. We also report the performance with page-splitting disabled (labeled NEMO-HugeOnly) to isolate the benefits of NEMO's skewness telemetry from the benefits of its hugepage access count telemetry. The figure is normalized to MEMTIS-PEBS with page-splitting disabled (i.e., only uses transparent hugepages).

When fast tier memory is abundant (1:2), NEMO shows little benefit in page splitting, because NEMO's high fidelity telemetry already identifies better page promotion candidates than PEBS due to the lack of sampling. The gap between NEMO and NEMO-HugeOnly widens as fast tier memory becomes smaller (1:4 to 1:8), showing that NEMO's page splitting telemetry helps MEMTIS use fast tier memory more efficiently by splitting and promoting hot basepages. MEMTIS-NEMO outperforms the PEBS-based baseline across all fast-to-slow memory ratios, with larger performance improvements as fast tier memory becomes more constrained. This is because NEMO identifies more than 2× the number of skewed pages than PEBS, allowing MEMTIS to pack more hot basepages into the fast tier. At the 1:8 memory ratio, Silo on MEMTIS-NEMO outperforms Silo on MEMTIS-PEBS by 13%.

Figure 6b shows that Silo benefits from better page placement decisions in MEMTIS-NEMO. As fast tier capacity becomes smaller (1:2 vs. 1:8), both MEMTIS-PEBS and MEMTIS-NEMO see higher latencies due to more accesses to slow tier memory. However, because NEMO identifies pages to split more effectively, MEMTIS can use the fast tier more efficiently by packing more hot basepages into the fast tier. This lowers Silo's latency across all percentiles compared to the PEBS baseline.



(a)



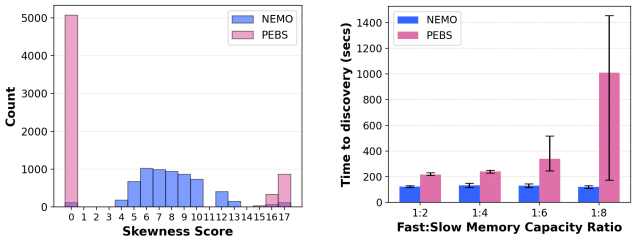
(b)

Figure 6: Silo performance under MEMTIS-NEMO and MEMTIS-PEBS across fast:slow memory ratios (1:2 to 1:8). Top: throughput speedup normalized to MEMTIS-PEBS without page splitting; NEMO-HugeOnly and PEBS-HugeOnly disable page splitting. Bottom: Silo latency CDF under 1:2 and 1:8 ratios.

Accuracy of page-split telemetry. Why is MEMTIS-NEMO's page-split telemetry more effective than MEMTIS-PEBS's sampling-based approach? MEMTIS ranks a huge page's skewness using a score computed from its base-page access counts: the more uneven the accesses, the higher the skewness score; if none of a huge page's base pages are hot, its skewness score is 0.

Figure 7a shows the distribution of skewness scores for all huge pages in Silo under the 1:8 fast-to-slow ratio, measured after each system reaches steady state. The distribution derived from PEBS is bimodal: most huge pages appear non-skewed (score = 0), while about 18% appear highly skewed (score $\geq$ 16). In contrast, NEMO yields a smoother distribution with far fewer zero-score pages and a broad spread of non-zero skewness values.

The PEBS distribution is distorted by sampling and cooling. MEMTIS periodically "cools" access counts by aging out old samples so that pages that no longer receive samples become cold, a standard technique in sample-based systems [36, 48]. Short cooling intervals improve responsiveness but give cold pages too few samples per interval, causing most hugepages



(a) (b)

Figure 7: Left: distribution of per-hugepage skewness scores under 1:8 memory ratio. Right: median time to identify 95% of page-split candidates over 5 runs; error bars show min/max.

to collapse to a zero skewness score; longer intervals allow counts to accumulate but hurt responsiveness to changes in access patterns, degrading application performance.

NEMO avoids this cooling trade-off. Since NEMO captures every cache miss to a tracked huge page, the policy can compute skewness directly from the per-basepage counts collected over a brief window (500 ms in our setup). The resulting telemetry reflects current access patterns without needing to age out stale observations. With NEMO, on this workload, skewness values span a wide range. NEMO enables MEMTIS to distinguish moderately from highly skewed pages and select splits more precisely.

Timeliness of finding page-split candidates. We now compare how quickly MEMTIS-PEBS and MEMTIS-NEMO discover split candidates. Using the same setup as the end-to-end experiment, Figure 7b reports the median time to identify 95% of all page-split candidates over five runs. We see that MEMTIS-NEMO's detection time is essentially independent of fast-tier size, because the number of candidates it can examine per unit time is tied to total physical memory, not to how much resides in the fast tier. In contrast, PEBS slows down and becomes more variable as the fast tier shrinks (e.g., 1:6 and 1:8 ratios). PEBS can quickly detect the most skewed hugepages (within roughly 250 seconds) but requires many more samples to classify moderately skewed pages. MEMTIS's dynamic PEBS sample rate adjustment, aimed at maintaining CPU overhead below 3%, produces large variance observed in sample rates for moderately hot pages in the 1:8 configuration. In contrast, since NEMO counts every access to tracked pages, it can compute accurate skewness for all candidates within about 150 seconds, yielding up to 10.4$\times$ detection speedup from PEBS.

CPU overhead. MEMTIS-NEMO incurs 3% overhead across all fast-to-slow memory ratios: 1.67% from NEMO processing and the rest from hot-tier PEBS sample gathering. MEMTIS-PEBS incurs 2.89% overhead.

```
nemo_telemetry_t t = nemo_register_telemetry((nemo_spec_t)
{
  .filter      = {.type=R|W},
  .translation = {
    .primary   = {.mask=NEMO_ALL_MEM, .shift=HP_SHIFT},
    // no sub-region: many hugepages -> one tenant state
  },
  .update      = {.op=NEMO_OP_ADD,   .operand=1},
  .notify      = {.op=NEMO_OP_GE,    .operand=BW_CAP},
  .read_effect = {.op=NEMO_OP_RESET, .value=0},
});

// Map every hugepage of tenant i to tenant i's state
for (i = 0; i < n_tenants; i++)
  for (j = 0; j < tenant[i].n_hp; j++)
    nemo_set_translation(t,
        /*key=*/        tenant[i].hp_paddr[j] >> HP_SHIFT,
        /*region_idx=*/ i);
```

Listing 3: NEMO telemetry for per-tenant bandwidth estimation. Many hugepages map to one tenant state. The notify predicate triggers when the threshold is exceeded.

5.3 Noisy Neighbor Detection

We also evaluate NEMO on its ability to accurately detect noisy neighbor applications. We show that NEMO detects noisy neighbors with two orders of magnitude lower CPU overhead compared to the state-of-the-art.

Memory performance isolation remains challenging in virtualized environments. Tenants expect stable memory latency and bandwidth, but operators cannot observe VM-internal behavior. As a result, a background task such as garbage collection can temporarily saturate DRAM bandwidth and violate the SLOs of a colocated latency-sensitive service [18]. Achieving isolation first requires detecting when a tenant consumes excessive memory bandwidth—a *noisy neighbor*. Timely detection is crucial: any delay directly delays mitigation and exposes other tenants to performance loss. Existing approaches built on hardware performance counters suffer from imprecise measurements, limited programmability, and high CPU overhead (§2.2). In contrast, NEMO can be programmed to attribute exact DRAM throughput to each tenant, enabling precise and timely noisy-neighbor detection [54].

NEMO uses the translation table to map all huge pages of a tenant to a single telemetry state. On each memory access, hardware increments this state, yielding a per-tenant count of DRAM accesses. The OS polls each counter every Δt milliseconds to compute bandwidth. When a tenant allocates or frees huge pages, the OS updates the translation table to preserve correct attribution. The administrator chooses Δt based on isolation needs; it can range from milliseconds to minutes. Finally, the OS installs a notification condition that raises an interrupt when a tenant's counter exceeds its bandwidth allocation. Listing 3 shows the telemetry: many-to-one translation maps every hugepage of a tenant to that tenant's state, the update operator sums accesses, and a $\geq$ predicate raises an interrupt on bandwidth-cap violation.

NEMO integration. We integrate NEMO with Linux cgroups to throttle noisy neighbors in software. A userspace QoS controller, modeled after node-level Kubernetes agents [34], monitors each tenant's NEMO bandwidth counter and adjusts its cgroup CPU quota to enforce a per-tenant bandwidth cap. We use cgroups as the actuation mechanism because they are ubiquitous in modern OSes and cloud platforms; we avoid Intel Memory Bandwidth Allocation (MBA) [22], whose throttling behavior is inaccurate in our measurements and prior work [53, 65]. The controller implements a simple feedback loop: every epoch it reads per-tenant counters, maintains an exponentially weighted moving average of bandwidth, and proportionally increases or decreases CPU quota based on the gap to the target bandwidth, using a learning rate to dampen adjustments, similar to prior regulators [18, 65, 70]. To reduce detection latency and CPU overhead, it also relies on NEMO notifications to react immediately when a tenant exceeds its allocation. Our prototype currently tracks bandwidth only for CXL-attached memory, so we restrict allocations to CXL. Finally, we use emulated NEMO notifications as detailed in §4.

Evaluation setup. As a baseline, we use Intel Memory Bandwidth Monitoring (MBM) counters via the PQoS library [30]; PMU-based schemes are less suitable due to inaccuracy and limited virtualization support (§2.1). In both cases, a userspace controller periodically polls per-tenant bandwidth counters and adjusts Linux cgroup CPU quotas to enforce a per-tenant bandwidth cap, varying the polling interval to explore the tradeoff between CPU overhead and reaction time. Our workload consists of a latency-sensitive victim, FlexKVS [31], and a single-core synthetic noisy neighbor that uses non-temporal stores to saturate memory bandwidth. The noisy neighbor alternates every 0.5 s between high-bandwidth execution and sleep, and is configured with a target cap of 1 GiB/s (~5% of our prototype's peak bandwidth).

5.3.1 Results

Bandwidth telemetry accuracy. We first run the noisy neighbor in isolation and compare bandwidth reported by MBM and NEMO against an application-level estimate. We approximate the application's true DRAM bandwidth by summing its non-temporal (NT) store volume. Across three 30-second runs, both NEMO and MBM closely match this NT-derived ground truth, with reported bandwidths within 0.1%.

Noisy neighbor throttling efficacy. Next, we co-locate the victim and noisy neighbor, throttling the noisy neighbor using either NEMO or MBM telemetry, and compare against two baselines: the victim alone and the victim with an unthrottled noisy neighbor. Figure 8 shows the resulting FlexKVS latency CDFs. With an unthrottled noisy neighbor, FlexKVS p99 latency increases by ~22%. With throttling based on NEMO

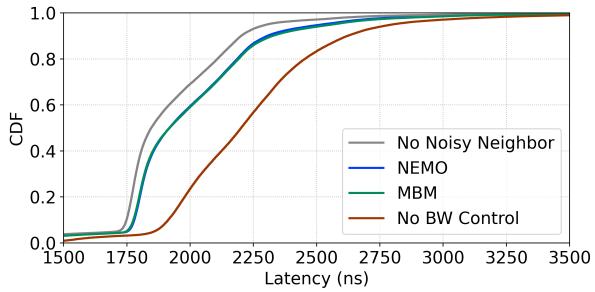


Figure 8: FlexKVS latency CDF using different telemetries for noisy neighbor detection and throttling. NEMO and MBM show equivalent detection capabilities when MBM uses a 1000Hz polling frequency (consuming 30%+ CPU overhead).

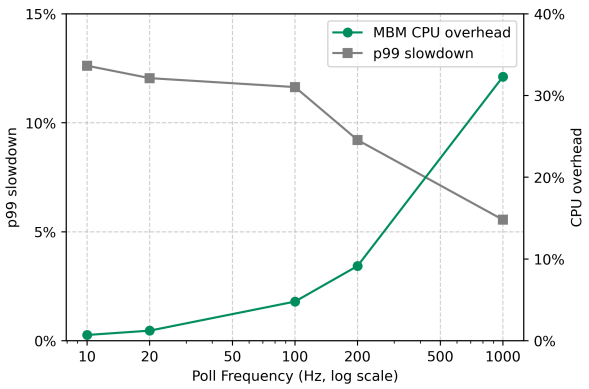


Figure 9: MBM p99 slowdown and CPU overhead vs. polling frequency.

and MBM, the p99 increase shrinks to 5.2% and 5.5%, respectively. In both cases, efficacy is ultimately limited by cgroups' actuation latency (~10 ms).

CPU overhead. Polling more frequently improves reaction time but increases overhead, so for MBM we vary the polling interval from 1 ms to 100 ms. Figure 9 shows the trade-off. At a 100 ms polling interval, MBM consumes only 0.7% of a core but reacts slowly, leading to a 13% increase in p99 latency. At a 1 ms interval, MBM reduces the p99 increase to 5.5% but raises CPU overhead to 32%. In contrast, NEMO achieves similar throttling effectiveness with only 0.09% CPU overhead. NEMO's notification mechanism triggers immediate reaction when a tenant exceeds its bandwidth allocation; the controller clears NEMO counters at a 50 ms interval in our setup.

6 Discussion

The preceding sections show that MC-resident telemetry can provide OS policies with timely, low-overhead memory signals. We now discuss three implications of the design: telemetry scalability with SRAM-backed state, the threat model under bare-metal and virtualized deployments, and how chained pipelines can extend NEMO to support more use cases.

6.1 Telemetry Scalability

The granularity of NEMO telemetries scales with on-chip SRAM, depending on the number of translation table entries and the amount of telemetry state. Many use cases rely primarily on one or the other resource, balancing SRAM utilization. Coarse telemetries (per-tenant bandwidth, §5.3) use many translation entries mapping into few states. Fine-grained telemetries (per-basepage skewness, §5.2) use fewer translation entries but many states per region.

Telemetry time multiplexing. When regions of interest exceed the resident SRAM budget, the OS can time-multiplex telemetries by reprogramming translation table entries between scans. Below we compute the *sweep time* T_{sweep}, the time to visit every telemetry region once under uniform coverage (the worst case; in practice, skewed access patterns and hot-region prioritization yield shorter effective sweeps). With P pipelines, S states of width W at G-byte granularity, and scan period τ, sweeping an M-byte region takes

$$T_{\text{sweep}} = \frac{M \times \tau}{P \times S \times G}.$$

Substituting $|\text{SRAM}| \approx P \times S \times W$ to utilize all available SRAM, this becomes

$$T_{\text{sweep}} = \frac{M \times \tau \times W}{|\text{SRAM}| \times G}.$$

Scaling implications. The formula has two scaling implications. First, the DRAM (M) to SRAM ($|\text{SRAM}|$) ratio sets the area-vs-timeliness trade-off. Second, counter width W is a knob orthogonal to coverage. At a fixed SRAM budget, halving W doubles the number of states and proportionally halves T_{sweep}. In our MEMTIS-NEMO prototype ($P=7$, $S=8{,}192$, $G=4$ KiB, $\tau=500$ ms), 1.4 MiB of telemetry SRAM covers 16 GiB DRAM at $T_{\text{sweep}} \approx 37$ s. At constant ratio, a 1 TiB tier needs ~90 MiB, on the same order as on-die SRAM for datacenter SoCs [4, 25]. Unlike CPU caches, NEMO uses SRAM in less area-constrained sites like the uncore (for IMCs) or controller ASICs (for CXL MCs). The prototype's $W=8$ B is conservative for our 500 ms scan; 1–4 B counters would shrink either the SRAM or T_{sweep} by 2–8×. Better encoding lowers the bit width W required per counter; for example, a saturating overflow bit for very hot data is sufficient to distinguish very hot counters, with remaining bits spent discriminating among warm and less-warm counters.

6.2 Security

NEMO's trusted computing base consists of the MC hardware and the NEMO driver in the OS or hypervisor kernel. We assume that telemetries are configured by a privileged operator. The NEMO driver validates each translation entry against the MC's physical address ranges, reusing the kernel's page-ownership checks to decide which pages each telemetry covers. In a virtualized setting, a paravirtualized interface lets a guest OS request telemetries over its memory. The hypervisor validates each request against the guest's allocated ranges and never exposes another guest's pages.

The hardware path adds no observable timing channel. Under both bare-metal and virtualization, the MC observes only request metadata (address and R/W type) for OS-registered pages, and the match-update pipeline runs with fixed overhead off the data path, so memory access latency is independent of NEMO's state. Side channels outside the MC (cache and DRAM timing) are unaffected.

6.3 Supporting Complex Telemetries

NEMO's pipeline performs one bounded update per request, keeping hardware simple and at line rate but ruling out telemetries needing cross-state reads or multi-step computation. A natural extension chains pipelines so one's output feeds the next (analogous to multi-stage match-action in RMT [11] switches) to express richer telemetries while preserving fixed-cycle stages. For example, approximate frequency sketches such as count-min [69] help when per-region tracking exceeds the SRAM budget: each chained pipeline holds one of the sketch's k counter arrays, and in its match stage a single-cycle hash (e.g., multiply-shift) projects each address onto that array, a small extension to the existing mask-shift key derivation. Querying a region's frequency hashes its address k times and takes the minimum across arrays. Each chained pipeline retains bounded per-request work at line rate.

7 Related Work

Hardware memory telemetry. Recent works have explored gathering telemetry directly from the memory controller [40, 47, 50, 52, 55, 73], employing different techniques (e.g., count-min sketches [55, 73] or Majority Element Algorithm [47]) for different use cases (e.g., tiered memory [55,73], disaggregated memory [40]) and memory technologies (e.g., 3D-stacked [40, 55]). While these designs demonstrate the value of in-memory telemetry, they lack flexibility because the telemetry is fixed at hardware design time. In contrast, NEMO enables the OS to define and install custom telemetry to MCs, achieving both flexibility and coverage.

OS-inferred memory telemetry. The OS can infer memory access patterns from hardware-generated signals, such as page access tracking via soft page faults [1, 5, 8, 33, 43, 61], page table scanning [1,17,29,42,67] or sampling-based approaches using PMUs [17, 36, 48]. These approaches provide flexibility but necessarily are limited because of high CPU overhead. NEMO offers an alternative by allowing the OS to install use-case-specific telemetries to memory controllers, achieving flexibility and high coverage with low CPU overhead.

Memory programmability. Prior works have explored introducing programmability to memory controllers. These works can alter the memory request (address or data), the request schedule, or DDR commands to improve application performance [9, 13], energy efficiency [9], fault tolerance, QoS [41], or for prototype exploration [21]. Beyond the memory controller, täkō [51] proposes a polymorphic cache hierarchy that executes software callbacks to transform data or alter traversal paths. Relative to these proposals, NEMO is simpler. It performs passive monitoring off the MC data path, making it lightweight and deployable.

Line-rate programmability. For network protocol stacks, the Reconfigurable Match-Action Table (RMT) architecture [11] and the P4 programming model [10] enable line-rate configurable dataplane actions. Operators often use these for telemetry [20, 32, 44, 45]. Inspired by these works, NEMO occupies a distinct design point: HW/SW codesign to provide high-fidelity telemetry on MCs for diverse OS use cases.

8 Conclusion

Server memory hierarchies demand high-fidelity, low-overhead telemetry, but existing techniques are often heavyweight, coarse, noisy, or fixed-function. NEMO addresses this gap by augmenting memory controllers with a small, flexible, off-data-path match-update-notify pipeline and per-controller SRAM state. Together with an OS-facing programming model and driver, OS kernel subsystems can define what to track and at which granularity. Using our CXL-based FPGA prototype, we show that this model improves the efficiency, accuracy, and responsiveness of tiering, huge-page splitting, and noisy-neighbor detection, outperforming state-of-the-art baselines within practical compute and SRAM budgets.

Acknowledgements. We thank our shepherd and the anonymous reviewers for their valuable feedback on the paper and artifact. This work is supported in part by the NSF CAREER Award 2333885; NSF grants 2105868, 2212580, and 2450086; the NSF Graduate Research Fellowship; PRISM, one of the seven centers in JUMP 2.0, a Semiconductor Research Corporation (SRC) program sponsored by DARPA; the University of Washington Center for the Future of Cloud Infrastructure (FOCI); as well as generous donations from NVIDIA, AMD, Intel, and Arm.

References

[1] AGARWAL, N., AND WENISCH, T. F. Thermostat: Application-transparent page management for two-tiered main memory. *SIGPLAN Not. 52*, 4 (Apr. 2017), 631–644.

[2] AKIYAMA, S., AND HIROFUCHI, T. Quantitative evaluation of intel PEBS overhead for online system-noise analysis. In *Proceedings of the 7th International Workshop on Runtime and Operating Systems for Supercomputers (ROSS '17)* (2017), ACM, pp. 1–8.

[3] ALTERA. Agilex™ 7 fpga i-series development kit (2× r-tile and 1× f-tile). https://www.altera.com/products/devkit/po-3012/agilex-7-fpga-i-series-development-kit-2x-r-tile-and-1x-f-tile, 2024.

[4] AMD. 4th gen AMD EPYC processor architecture. https://www.amd.com/content/dam/amd/en/documents/products/epyc/4th-gen-epyc-processor-architecture-white-paper.pdf, 2022.

[5] ARCANGELI, A. Autonuma. https://blog.linuxplumbersconf.org/2012/wp-content/uploads/2012/09/2012-lpc-virt-autonuma-arcangeli.pdf, 2012.

[6] BERG, B., BERGER, D. S., MCALLISTER, S., GROSOF, I., GUNASEKAR, S., LU, J., UHLAR, M., CARRIG, J., BECKMANN, N., HARCHOL-BALTER, M., AND GANGER, G. R. The CacheLib caching engine: Design and experiences at scale. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)* (Nov. 2020), USENIX Association, pp. 753–768.

[7] BERGER, D. S., KUMAR, K., VUPPALAPATI, M., DOUGLAS, C., SATHRE, J., ROBINSON, I., TANDON, P., AND HILL, M. D. CXL in Cloud Practice: Practical Lessons for Incrementally Scaling Deployment . *IEEE Transactions on Computers 75*, 04 (Apr. 2026), 1234–1246.

[8] BERGMAN, S., FALDU, P., GROT, B., VILANOVA, L., AND SILBERSTEIN, M. Reconsidering os memory optimizations in the presence of disaggregated memory. In *Proceedings of the 2022 ACM SIGPLAN International Symposium on Memory Management* (New York, NY, USA, 2022), ISMM 2022, Association for Computing Machinery, p. 1–14.

[9] BOJNORDI, M. N., AND IPEK, E. A programmable memory controller for the ddrx interfacing standards. *ACM Trans. Comput. Syst. 31*, 4 (Dec. 2013).

[10] BOSSHART, P., DALY, D., GIBB, G., IZZARD, M., MCKEOWN, N., REXFORD, J., SCHLESINGER, C., TALAYCO, D., VAHDAT, A., VARGHESE, G., AND WALKER, D. P4: programming protocol-independent packet processors. *SIGCOMM Comput. Commun. Rev. 44*, 3 (July 2014), 87–95.

[11] BOSSHART, P., GIBB, G., KIM, H.-S., VARGHESE, G., MCKEOWN, N., IZZARD, M., MUJICA, F., AND HOROWITZ, M. Forwarding metamorphosis: fast programmable match-action processing in hardware for sdn. *SIGCOMM Comput. Commun. Rev. 43*, 4 (Aug. 2013), 99–110.

[12] BURCKHARDT, S., CHANDRAMOULI, B., GILLUM, C., JUSTO, D., KALLAS, K., MCMAHON, C., MEIKLEJOHN, C. S., AND ZHU, X. Netherite: Efficient execution of serverless workflows. *Proc. VLDB Endow. 15*, 8 (2022), 1591–1604.

[13] CARTER, J., HSIEH, W., STOLLER, L., SWANSON, M., ZHANG, L., BRUNVAND, E., DAVIS, A., KUO, C.-C., KURAMKOTE, R., PARKER, M., SCHAELICKE, L., AND TATEYAMA, T. Impulse: building a smarter memory controller. In *Proceedings Fifth International Symposium on High-Performance Computer Architecture* (1999), pp. 70–79.

[14] CHANDRAMOULI, B., PRASAAD, G., KOSSMANN, D., LEVANDOSKI, J., HUNTER, J., AND BARNETT, M. Faster: A concurrent key-value store with in-place updates. In *Proceedings of the 2018 International Conference on Management of Data* (New York, NY, USA, 2018), SIGMOD '18, Association for Computing Machinery, p. 275–290.

[15] CHAUHAN, P., PETERSEN, C., MORRIS, B., AND GLISSE, J. Hyperscale tiered memory expander specification – for compute express link® (cxl®) revision 1. https://www.opencompute.org/documents/hyperscale-tiered-memory-expander-specification-for-compute-express-link-cxl-1-pdf, 2023.

[16] COOPER, B. F., SILBERSTEIN, A., TAM, E., RAMAKRISHNAN, R., AND SEARS, R. Benchmarking cloud serving systems with ycsb. In *Proceedings of the 1st ACM Symposium on Cloud Computing* (New York, NY, USA, 2010), SoCC '10, Association for Computing Machinery, p. 143–154.

[17] DURAISAMY, P., XU, W., HARE, S., RAJWAR, R., CULLER, D., XU, Z., FAN, J., KENNELLY, C., MCCLOSKEY, B., MIJAILOVIC, D., MORRIS, B., MUKHERJEE, C., REN, J., THELEN, G., TURNER, P., VILLAVIEJA, C., RANGANATHAN, P., AND VAHDAT, A. Towards an adaptable systems architecture for memory tiering at warehouse-scale. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3* (New York, NY, USA, 2023), ASPLOS 2023, Association for Computing Machinery, p. 727–741.

[18] FRIED, J., RUAN, Z., OUSTERHOUT, A., AND BELAY, A. Caladan: Mitigating interference at microsecond timescales. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)* (Nov. 2020), USENIX Association, pp. 281–297.

[19] GANGULY, D., ZHANG, Z., YANG, J., AND MELHEM, R. Interplay between hardware prefetcher and page eviction policy in CPU–GPU unified virtual memory. In *Proceedings of the 46th International Symposium on Computer Architecture (ISCA)* (2019).

[20] GUPTA, A., HARRISON, R., CANINI, M., FEAMSTER, N., REXFORD, J., AND WILLINGER, W. Sonata: query-driven streaming network telemetry. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication* (New York, NY, USA, 2018), SIGCOMM '18, Association for Computing Machinery, p. 357–371.

[21] HASSAN, H., VIJAYKUMAR, N., KHAN, S., GHOSE, S., CHANG, K., PEKHIMENKO, G., LEE, D., ERGIN, O., AND MUTLU, O. Softmc: A flexible and practical open-source infrastructure for enabling experimental dram studies. In *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)* (2017), pp. 241–252.

[22] HERDRICH, A. J., CORNU, M. D., AND ABBASI, K. M. Introduction to memory bandwidth allocation. https://www.intel.com/content/www/us/en/developer/articles/technical/introduction-to-memory-bandwidth-allocation.html, 2019.

[23] HYLANDER, E. G. Azure delivers the first cloud vm with intel xeon 6 and cxl memory - now in private preview, Nov. 2025. https://techcommunity.microsoft.com/blog/SAPApplications/azure-delivers-the-first-cloud-vm-with-intel-xeon-6-and-cxl-memory---now-in-priv/4470067.

[24] INTEL. Technical overview of the 4th gen intel® xeon® scalable processor family. https://www.intel.com/content/www/us/en/developer/articles/technical/fourth-generation-xeon-scalable-family-overview.html, 2022.

[25] INTEL. Intel® xeon® 6 product brief (Granite Rapids). https://www.intel.com/content/www/us/en/products/docs/xeon-6-product-brief.html, 2024.

[26] INTEL CORPORATION. *Intel® Xeon® Processor Scalable Memory Family Uncore Performance Monitoring Reference Manual*, 2017. See Chapter 2.6: Intel UPI Link Layer Performance Monitoring. https://www.intel.com/content/www/us/en/content-details/671389/intel-xeon-processor-scalable-memory-family-uncore-performance-monitoring-reference-manual.html.

[27] INTEL CORPORATION. *Intel 64 and IA-32 Architectures Software Developer's Manual, Volume 3: System Programming Guide*, 2023. See Chapter 18: Performance Monitoring. https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html.

[28] INTEL CORPORATION. Intel® Memory Latency Checker. https://www.intel.com/content/www/us/en/developer/articles/tool/intelr-memory-latency-checker.html, 2025.

[29] KANNAN, S., GAVRILOVSKA, A., GUPTA, V., AND SCHWAN, K. Heteroos: Os design for heterogeneous memory management in datacenter. *SIGARCH Comput. Archit. News 45*, 2 (June 2017), 521–534.

[30] KANTECKI, T., CORNU, M. D., HETHERINGTON, A., ALEKSINSKI, M., ANDRALOJC, W., AND BOCZKOWSKI, A. pqos — intel(r) resource director technology / amd pqos monitoring and control tool, Apr. 2022. man page, version from Debian testing (intel-cmt-cat 25.04-1).

[31] KAUFMANN, A., PETER, S., SHARMA, N. K., ANDERSON, T., AND KRISHNAMURTHY, A. High performance packet processing with flexnic. *SIGPLAN Not. 51*, 4 (Mar. 2016), 67–81.

[32] KIM, C., SIVARAMAN, A., KATTA, N. P. K., BAS, A., DIXIT, A. A., AND WOBKER, L. J. In-band network telemetry via programmable dataplanes. In *Proceedings of the 2015 ACM SIGCOMM Conference Posters and Demos* (2015), SIGCOMM Posters and Demos '15, Association for Computing Machinery.

[33] KIM, J., CHOE, W., AND AHN, J. Exploring the design space of page management for Multi-Tiered memory systems. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)* (July 2021), USENIX Association, pp. 715–728.

[34] Koordinator: Qos-based scheduling for efficient orchestration of microservices, ai, and big data workloads on kubernetes. https://koordinator.sh/, 2025.

[35] KWON, Y., YU, H., PETER, S., ROSSBACH, C. J., AND WITCHEL, E. Coordinated and efficient huge page management with ingens. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation* (USA, 2016), OSDI'16, USENIX Association, p. 705–721.

[36] LEE, T., MONGA, S. K., MIN, C., AND EOM, Y. I. Memtis: Efficient memory tiering with dynamic page classification and page size determination. In *Proceedings of the 29th Symposium on Operating Systems Principles* (New York, NY, USA, 2023), SOSP '23, Association for Computing Machinery, p. 17–34.

[37] LI, A., SUDVARG, M., LI, Z., BARUAH, S., GILL, C., AND ZHANG, N. Tintin: a unified hardware performance profiling infrastructure to uncover and manage uncertainty. In *Proceedings of the 19th USENIX Conference on Operating Systems Design and Implementation* (USA, 2025), OSDI '25, USENIX Association.

[38] LI, C., SHA, S., ZENG, Y., YANG, X., LUO, Y., WANG, X., WANG, Z., AND ZHOU, D. Taming hot bloat under virtualization with hugescope. In *Proceedings of the 2024 USENIX Conference on Usenix Annual Technical Conference* (USA, 2024), USENIX ATC'24, USENIX Association.

[39] LI, H., BERGER, D. S., NOVAKOVIC, S., HSU, L., ERNST, D., ZARDOSHTI, P., SHAH, M., RAJADNYA, S., LEE, S., AGARWAL, I., HILL, M. D., FONTOURA, M., AND BIANCHINI, R. Pond: CXL-based memory pooling systems for cloud platforms. In *Proceedings of the 28th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '23)* (2023), ACM, pp. 574–589.

[40] LI, H., LIU, K., LIANG, T., LI, Z., LU, T., YUAN, H., XIA, Y., BAO, Y., CHEN, M., AND SHAN, Y. Hopp: Hardware-software co-designed page prefetching for disaggregated memory. In *2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA)* (2023), pp. 1168–1181.

[41] MA, J., SUI, X., SUN, N., LI, Y., YU, Z., HUANG, B., XU, T., YAO, Z., CHEN, Y., WANG, H., ZHANG, L., AND BAO, Y. Supporting differentiated services in computers via programmable architecture for resourcing-on-demand (pard). *SIGPLAN Not. 50*, 4 (Mar. 2015), 131–143.

[42] MARUF, A., GHOSH, A., BHIMANI, J., CAMPELLO, D., RUDOFF, A., AND RANGASWAMI, R. Multi-clock: Dynamic tiering for hybrid memory systems. In *2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA)* (2022), pp. 925–937.

[43] MARUF, H. A., WANG, H., DHANOTIA, A., WEINER, J., AGARWAL, N., BHATTACHARYA, P., PETERSEN, C., CHOWDHURY, M., KANAUJIA, S., AND CHAUHAN, P. Tpp: Transparent page placement for cxl-enabled tiered-memory. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3* (New York, NY, USA, 2023), ASPLOS 2023, Association for Computing Machinery, p. 742–755.

[44] MCKEOWN, N., ANDERSON, T., BALAKRISHNAN, H., PARULKAR, G., PETERSON, L., REXFORD, J., SHENKER, S., AND TURNER, J. Openflow: enabling innovation in campus networks. *SIGCOMM Comput. Commun. Rev. 38*, 2 (Mar. 2008), 69–74.

[45] NARAYANA, S., SIVARAMAN, A., NATHAN, V., GOYAL, P., ARUN, V., ALIZADEH, M., JEYAKUMAR, V., AND KIM, C. Language-directed hardware design for network performance monitoring. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication* (New York, NY, USA, 2017), SIGCOMM '17, Association for Computing Machinery, p. 85–98.

[46] NGUYEN, K. T. Introduction to memory bandwidth monitoring in the intel® xeon® processor E5 v4 family. https://www.intel.com/content/www/us/en/developer/articles/technical/introduction-to-memory-bandwidth-monitoring.html, 2016. Intel Technical Article.

[47] PRODROMOU, A., MESWANI, M., JAYASENA, N., LOH, G., AND TULLSEN, D. M. Mempod: A clustered architecture for efficient and scalable migration in flat address space multi-level memories. In *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)* (2017), pp. 433–444.

[48] RAYBUCK, A., STAMLER, T., ZHANG, W., EREZ, M., AND PETER, S. Hemem: Scalable tiered memory management for big data applications and real nvm. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles* (New York, NY, USA, 2021), SOSP '21, Association for Computing Machinery, p. 392–407.

[49] REN, G., TUNE, E., MOSELEY, T., SHI, Y., RUS, S., AND HUNDT, R. Google-wide profiling: A continuous profiling infrastructure for data centers. *IEEE Micro 30*, 4 (July 2010), 65–79.

[50] RYOO, J. H., MESWANI, M. R., PRODROMOU, A., AND JOHN, L. K. Silc-fm: Subblocked interleaved cache-like flat memory organization. In *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)* (2017), pp. 349–360.

[51] SCHWEDOCK, B. C., YOOVIDHYA, P., SEIBERT, J., AND BECKMANN, N. täkō: a polymorphic cache hierarchy for general-purpose optimization of data movement. In *Proceedings of the 49th Annual International Symposium on Computer Architecture* (New York, NY, USA, 2022), ISCA '22, Association for Computing Machinery, p. 42–58.

[52] SIM, J., ALAMELDEEN, A. R., CHISHTI, Z., WILKERSON, C., AND KIM, H. Transparent hardware management of stacked dram as part of memory. In *2014 47th Annual IEEE/ACM International Symposium on Microarchitecture* (2014), pp. 13–24.

[53] SOHAL, P., BECHTEL, M., MANCUSO, R., YUN, H., AND KRIEGER, O. A closer look at intel resource director technology (RDT). In *30th International Conference on Real-Time Networks and Systems (RTNS '22)* (2022), ACM, pp. 95–106.

[54] SOHAL, P., TABISH, R., DREPPER, U., AND MANCUSO, R. E-warp: A system-wide framework for memory bandwidth profiling and management. In *2020 IEEE Real-Time Systems Symposium (RTSS)* (2020), pp. 345–357.

[55] SUN, Y., KIM, J., YU, Z., ZHANG, J., CHAI, S., KIM, M. J., NAM, H., PARK, J., NA, E., YUAN, Y., WANG, R., AHN, J. H., XU, T., AND KIM, N. S. *M5: Mastering Page Migration and Memory Management for CXL-based Tiered Memory Systems*. Association for Computing Machinery, New York, NY, USA, 2025, p. 604–621.

[56] SUN, Y., YUAN, Y., YU, Z., KUPER, R., SONG, C., HUANG, J., JI, H., AGARWAL, S., LOU, J., JEONG, I., WANG, R., AHN, J. H., XU, T., AND KIM, N. S. Demystifying cxl memory with genuine cxl-ready systems and devices. In *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture* (New York, NY, USA, 2023), MICRO '23, Association for Computing Machinery, p. 105–121.

[57] THE NEXT PLATFORM. Cxl and gen-z iron out a coherent interconnect strategy. https://www.nextplatform.com/connect/2020/04/03/cxl-and-gen-z-iron-out-a-coherent-interconnect-strategy/1644243, 2020.

[58] Transparent hugepage support. https://docs.kernel.org/mm/transhuge.html.

[59] TU, S., ZHENG, W., KOHLER, E., LISKOV, B., AND MADDEN, S. Speedy transactions in multicore in-memory databases. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles* (New York, NY, USA, 2013), SOSP '13, Association for Computing Machinery, p. 18–32.

[60] VERMA, A., PEDROSA, L., KORUPOLU, M., OPPENHEIMER, D., TUNE, E., AND WILKES, J. Large-scale cluster management at google with borg. In *Proceedings of the Tenth European Conference on Computer Systems* (New York, NY, USA, 2015), EuroSys '15, Association for Computing Machinery.

[61] VERMA, V. Tiering-0.8. https://git.kernel.org/pub/scm/linux/kernel/git/vishal/tiering.git/log/?h=tiering-0.8, 2024.

[62] VUPPALAPATI, M., AND AGARWAL, R. Tiered memory management: Access latency is the key! In *Proceedings of the 30th ACM Symposium on Operating Systems Principles (SOSP '24)* (2024), ACM, pp. 79–94.

[63] WEINER, J., AGARWAL, N., SCHATZBERG, D., YANG, L., WANG, H., SANOUILLET, B., SHARMA, B., HEO, T., JAIN, M., TANG, C., AND SKARLATOS, D. Tmo: transparent memory offloading in datacenters. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems* (New York, NY, USA, 2022), ASPLOS '22, Association for Computing Machinery, p. 609–621.

[64] XIANG, L., LIN, Z., DENG, W., LU, H., RAO, J., YUAN, Y., AND WANG, R. Nomad: Non-Exclusive memory tiering via transactional page migration. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)* (Santa Clara, CA, July 2024), USENIX Association, pp. 19–35.

[65] XIANG, Y., YE, C., WANG, X., LUO, Y., AND WANG, Z. Emba: Efficient memory bandwidth allocation to improve performance on intel commodity processor. In *Proceedings of the 48th International Conference on Parallel Processing* (New York, NY, USA, 2019), ICPP '19, Association for Computing Machinery.

[66] XU, L. Kvm: x86/pmu: Prevent any host user from enabling pebs for profiling guest. https://lore.kernel.org/all/6c4bd247-1f81-4b43-9e21-012f831d26b8@linux.intel.com/T/, 11 2023. Linux Kernel Mailing List post, message ID <6c4bd247-1f81-4b43-9e21-012f831d26b8@linux.intel.com>.

[67] YAN, Z., LUSTIG, D., NELLANS, D. W., AND BHATTACHARJEE, A. Nimble page management for tiered memory systems. In *Proceedings of the 24th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '19)* (2019), ACM, pp. 331–345.

[68] YI, J., DONG, B., DONG, M., AND CHEN, H. On the precision of precise event based sampling. In *Proceedings of the 11th ACM SIGOPS Asia-Pacific Workshop on Systems (APSys '20)* (2020), ACM, pp. 89–96.

[69] YU, M., JOSE, L., AND MIAO, R. Software defined traffic measurement with opensketch. In *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation* (USA, 2013), nsdi'13, USENIX Association, p. 29–42.

[70] YUN, H., YAO, G., PELLIZZONI, R., CACCAMO, M., AND SHA, L. Memguard: Memory bandwidth reservation system for efficient performance isolation in multi-core platforms. In *2013 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)* (2013), pp. 55–64.

[71] ZHONG, Y., BERGER, D. S., WALDSPURGER, C., WEE, R., AGARWAL, I., AGARWAL, R., HADY, F., KUMAR, K., HILL, M. D., CHOWDHURY, M., AND CIDON, A. Managing memory tiers with CXL in virtualized environments. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)* (Santa Clara, CA, July 2024), USENIX Association, pp. 37–56.

[72] ZHONG, Y., KAZHAMIAKA, F., ZARDOSHTI, P., TENG, S., FONSECA, R., HILL, M. D., AND BERGER, D. S. Octopus: Enhancing CXL memory pods via sparse topology. In *23rd USENIX Symposium on Networked Systems Design and Implementation (NSDI 26)* (Renton, WA, May 2026), USENIX Association, pp. 1303–1322.

[73] ZHOU, Z., CHEN, Y., ZHANG, T., WANG, Y., SHU, R., XU, S., CHENG, P., QU, L., XIONG, Y., ZHANG, J., AND SUN, G. NeoMem: Hardware/Software Co-Design for CXL-Native Memory Tiering . In *2024 57th IEEE/ACM International Symposium on Microarchitecture (MICRO)* (Los Alamitos, CA, USA, Nov. 2024), IEEE Computer Society, pp. 1518–1531.

OBASE: Object-Based Address-Space Engineering to Improve Memory Tiering

Vinay Banakar[1, 3], Suli Yang[3], Kan Wu[2*], Andrea C. Arpaci-Dusseau[1],
Remzi H. Arpaci-Dusseau[1], Kimberly Keeton[3]

[1]University of Wisconsin-Madison [2]xAI [3]Google

Abstract

Hardware and OS mechanisms for memory tiering are widely deployed, yet datacenters still overprovision DRAM. The root cause is *hotness fragmentation*: allocators place objects by size rather than access pattern, so hot and cold objects become interleaved within the same pages. A single hot object marks its page as active, trapping surrounding cold data in expensive DRAM. Our analysis of Google datacenter workloads shows that up to 97% of the bytes in active pages are cold and unreclaimable. We propose *address-space engineering*: dynamically reorganizing virtual memory so that hot objects cluster into uniformly hot pages and cold objects into uniformly cold pages. We present *OBASE*, a compiler-runtime system for unmanaged languages that serves as an object-aware *frontend* for page-aware OS *backends*. *OBASE* tracks accesses via lightweight pointer instrumentation and migrates objects at runtime using a lock-free protocol that is safe under concurrency. By reorganizing the address space, *OBASE* enables unmodified backends (kswapd, TMO, TPP, Memtis) to tier memory effectively. Across ten concurrent data structures, six backends, and production traces from Meta and Twitter, *OBASE* improves page utilization by 2–4× and reduces memory footprint by up to 70%, with only 2–5% overhead.

1 Introduction

Memory capacity, especially DRAM, has become the dominant cost driver in modern data centers, often accounting for 50% of server capital expenses [1, 49, 59]. Memory tiering promises a solution: by placing cold, less accessed data in cheaper, slower tiers—such as compressed memory, SSDs, or CXL-attached disaggregated memory [3, 4, 6, 11, 42, 43]—one could significantly reduce the expensive DRAM capacity required, thereby lowering costs [29, 44, 49, 56, 60, 62].

In principle, this approach shows great promise, primarily because in hyperscale workloads, only a small volume of memory is hot; the vast majority remains cold and un-accessed for extended periods. As shown in Figure 1, across six different Google workloads [13], only 1.7%–21.3% of bytes are accessed during the trace duration. Indeed, four of the six workloads access less than 3% of bytes, suggesting that we could reclaim 80% to 98% of DRAM for secondary tiers with minimal impact on application performance.

*Work done at Google

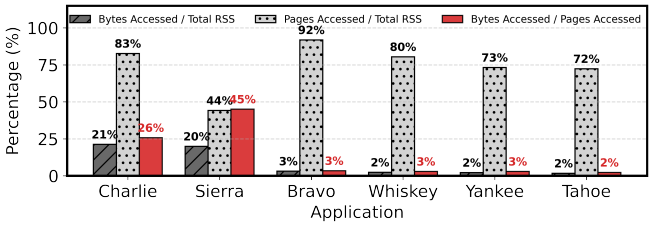


Figure 1. The granularity gap in memory access of Google workloads. Dark Grey: The percentage of total memory *bytes* actually accessed. Light Grey: The percentage of total memory *pages* accessed. Red: The page utilization (calculated as Bytes Accessed / Pages Accessed, see §2.2 for more details).

In practice, however, realizable gains often fall short of this potential. Google reported offloading only 20% of infrequently accessed data to a compressed memory tier [43], while Meta achieved similarly modest savings of 20–32% [62].

The crux of this gap lies in a mismatch between how applications organize data and how the OS manages memory: applications access data at *object* granularity, which varies in size, while the OS manages memory at fixed-size *page* granularity (4KB, 2MB, or 1GB) [20]. Modern allocators place objects on pages with no regard for how these objects will be accessed in the future, often placing frequently accessed objects and rarely touched objects intermingled on the same page. As a result, a few hot bytes (e.g., a single object) would make a whole page active from the OS's perspective, trapping the whole page in expensive DRAM. One can clearly see this phenomenon in Figure 1, where even though only 3.2% of *bytes* are accessed in a workload Bravo, 91.8% of the *pages* are accessed, making them unfit candidates for reclamation. Similar trends can be observed across all six workloads. We term this phenomenon **hotness fragmentation** and identify it as the root cause of inefficient tiering at hyperscalers. We quantify this fragmentation using the *page utilization* metric.

To combat hotness fragmentation, we introduce **address space engineering**, which *dynamically* reorganizes objects within the address space so that objects with similar access intensities are grouped together. By segregating hot and cold objects, address space engineering promises to close the gap between theoretical and realizable memory savings. It creates a memory layout with uniformly cold pages (or hot) that are significantly more amenable to tiering.

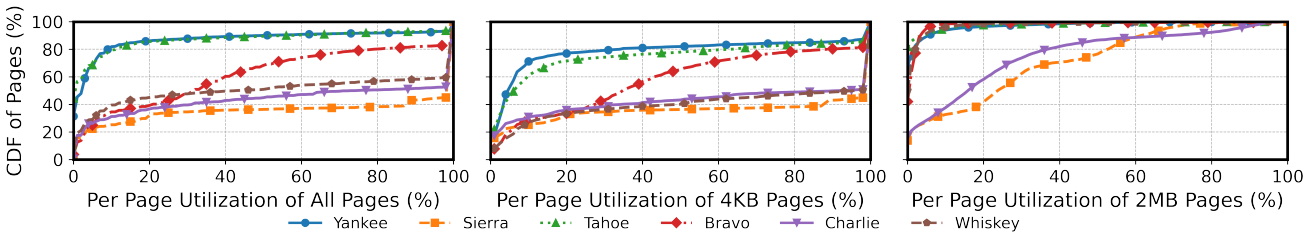


Figure 2. Low Per-Page Utilization in Google Data Center Workloads. CDFs of page utilization for six applications, shown for all pages combined (left) and separately for 4KB pages (center) and 2MB pages (right).

Crucially, by focusing on the address space *layout*, we decouple memory tiering into two orthogonal concerns, allowing independent innovation within each: the *layout* (frontend) problem, which organizes objects to yield high-quality page candidates for reclamation; and the *reclamation* (backend) problem, which focuses on migration policies and mechanisms across different tiers. This separation allows us to reuse existing page-based tiering infrastructures—both swap-based [43, 62] and byte-addressable [44, 49, 56]—and leverage future improvements in reclamation mechanisms. It also ensures that our layout optimization techniques remain applicable to future, currently non-existent memory tiers (e.g., CXL 3.0 fabrics or NVM). Further, it significantly lowers the adoption barrier for hyperscalers: rather than requiring changes to their existing tiering backends, address space engineering provides a superior memory layout that makes existing backends work more effectively.

In this paper, we present ***O*bject-*B*ased *A*ddress-*S*pace *E*ngineering (*OBASE*)**, a compiler-runtime system that engineers the address space for *unmanaged* languages like C/C++. *OBASE* acts as an intelligent *frontend* for memory organization: it manages pointer-based data structures, transparently profiles object access using lightweight instrumentation to determine access intensity, and employs a novel lock-free protocol to migrate objects safely, clustering hot and cold objects together. As a result, *OBASE* prepares the memory layout for any OS *backend*, enabling them to work more effectively. *OBASE* preserves a familiar programming model: developers annotate which pointer fields may relocate, and the compiler routes their accesses through guides. This does require giving up two assumptions (stable object addresses and pointer arithmetic over managed objects) and applies to pointer-based structures rather than arbitrary code; we make this boundary explicit in §7.

We evaluate *OBASE* with ten concurrent pointer-based data structures and six state-of-the-art reclamation and tiering backends [17, 44, 49, 60, 62]. Using key-value store workloads (YCSB) and production traces (Meta, Twitter), we demonstrate that *OBASE* improves page utilization (by 2–4×) and achieves higher memory savings (up to 70%) at the same performance overhead, or conversely, achieves equivalent savings with negligible performance impact. For tiered-memory configurations, *OBASE* achieves the same throughput with half the DRAM capacity.

2 Case for Dynamic Object Reorganization

In this section, we show that hotness fragmentation is both severe and unavoidable under static object placement. We introduce a metric to quantify fragmentation, analyze data center traces from Google [13], and demonstrate that object hotness changes continuously over time. Finally, we explain why unmanaged languages make dynamic reorganization especially challenging, motivating the design of *OBASE*.

2.1 Page Utilization: A Metric

The OS marks a page as "active" if it receives at least one access during a time window T. We define *per-page utilization* as the fraction of a touched page's capacity that is actually accessed. Let $P(T)$ be the set of touched pages during T, and let $U(p,T)$ be the number of unique bytes accessed within page p during T. We then define the aggregate *page utilization* as:

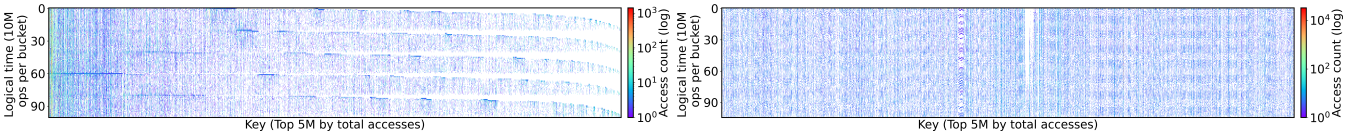
$$\text{Page Utilization}(T) = \frac{\sum_{p \in P(T)} U(p,T)}{\sum_{p \in P(T)} \text{Size}(p)}$$

Low utilization means a page appears hot to the OS even when most of its bytes are cold. Previous work [21] demonstrated low page utilization for open-source databases (e.g., Redis, MongoDB) where 75–90% of accessed pages utilized less than 15% of their capacity for YCSB workloads.

2.2 Fragmentation in Data Center Workloads

To understand hotness fragmentation across a broader range of applications, we analyze memory access traces from six data center workloads at Google [13], collected using DynamoRIO [12]. We process each unique access, annotate pages as 4KB or 2MB using `/proc/self/smaps`, and count unique 64B cache lines touched per page. Based on instruction counts, core counts, and typical warehouse-scale CPU characteristics [41], we estimate the traces capture up to ~30s of steady-state execution.

Figure 2 shows the cumulative distribution of per-page utilization for all workloads, ordered as: (a) aggregated across both 4KiB and 2MiB page sizes, (b) 4KB pages only, and (c) 2MB pages only.



(a) Meta KV Trace. White horizontal bands indicate coordinated quiet periods where access drops across many keys. These bands reveal phased workload behavior.

(b) Twitter Trace. A sparse, scattered pattern indicates sporadic hotness. A few keys on the far left remain consistently hot, but the majority exhibit bursty access with long idle gaps.

Figure 3. Temporal Evolution of Key Hotness. Heatmaps showing access frequency (log scale) for the top 5M keys over logical time buckets (10M operations each). If hotness were static, we would observe continuous vertical bands on the left side of each plot. Instead, we see shifting phases (Meta) and intermittent bursts (Twitter), demonstrating that the hot working set evolves continuously.

Aggregated view. Aggregated across both page sizes, all six workloads show populations of poorly-utilized pages. Yankee and Tahoe are the most fragmented: median utilization is around 3%, meaning half of their touched pages waste over 95% of capacity on cold data. Bravo follows with a median near 35%. The remaining workloads have higher overall utilization but still carry long left tails: roughly 25–30% of pages in Sierra, Charlie, and Whiskey use less than 20% of their capacity. Page-based reclamation cannot distinguish these poorly-utilized pages from well-utilized ones without object-level information.

4KB pages. Fragmentation is visible even at the smallest page size: for Tahoe and Yankee ~80% of pages fall below 20% utilization, for Bravo 60% fall below 40%, and Charlie, Whiskey, and Sierra have 35–40% of pages below 40%. Hotness fragmentation is thus inherent to access patterns, not merely an artifact of huge pages.

2MB pages. Fragmentation worsens dramatically for huge pages. Tahoe, Bravo, and Yankee are extreme: 85–90% of their huge pages utilize under 10% of their 2MB capacity, and even Sierra has 40% below 20%. Each such page, consuming 512× a base page, holds over 1.8MB of cold data alongside a few accessed cache lines.

Finding 1: *Active pages are mostly cold: 70–90% of bytes in pages the OS considers hot receive no accesses.*

2.3 Object Hotness Changes Over Time

One potential solution for hotness fragmentation is to segregate hot and cold objects at allocation time, using static hints or profiling [40]. This strategy assumes that hotness is both identifiable at allocation time and relatively stable afterward.

First, identifying hotness at allocation time is challenging because the same code path allocates objects with vastly different lifecycles. For example, in Redis and Memcached, a single `SET` handler allocates memory for all incoming records, yet one record might be a session token accessed every millisecond, while another is a user profile never touched again. Static analysis cannot distinguish these cases at allocation time. Second, this approach assumes that object hotness is relatively stable: objects identified as hot when allocated will remain hot throughout their lifetimes.

We show that these assumptions do not hold. We analyze object-level traces from Meta [22] and Twitter [65], which record operations in logical time (10M operations per bucket). Figure 3 visualizes access intensity for the top 5 million keys (ranked by total accesses) over 100 such buckets. If hotness were static, the most popular keys (left) would exhibit continuous vertical bands. Instead, both traces show significant churn: bursts of activity followed by long idle gaps.

Meta: Phased hotness. The Meta workload shows coordinated phases where many keys become inactive at once, visible as white horizontal bands. Even keys that are repeatedly accessed overall alternate between active bursts and extended dormancy. A page packed with currently-hot objects eventually becomes a page dominated by cold objects as access patterns shift.

Twitter: Sporadic hotness. The Twitter workload shows a sparse pattern. A small slice of keys remains consistently hot, but the majority exhibit brief access bursts separated by long idle gaps, even among the top 5 million keys.

To quantify churn, we measure reuse-distance variability (75th to 25th percentile of number of operations between accesses to the same key). For mid-sized objects (64B–4KB), representing 94% of Meta keys and 98.2% of Twitter keys, 75% of keys have a reuse spread exceeding 5×, and 65% exceed 30×. Thus, access gaps for the majority of keys fluctuate by more than an order of magnitude.

Finding 2: *Hotness is transient; object hotness is neither knowable at allocation time nor stable over time. As a result, one-time placement cannot prevent hotness fragmentation, and dynamic object migration based on changing hotness is required.*

2.4 Object Mobility in Unmanaged Languages

Moving objects dynamically presents varying degrees of difficulty depending on the language runtime. Managed runtimes, such as the JVM and Go, can relocate objects during garbage collection by updating references atomically, making mobility relatively straightforward. In unmanaged languages like C++, however, programs assume that an object's address is stable [23], rendering dynamic object migration significantly

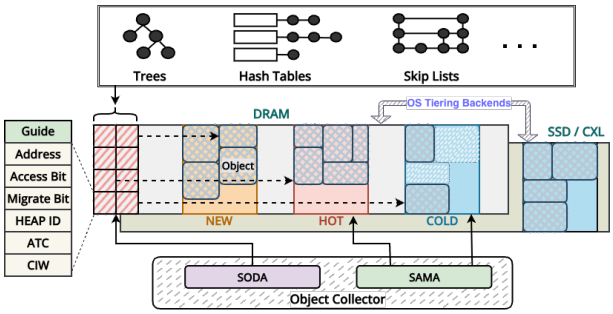


Figure 4. *OBASE* Overview. *OBASE* acts as a frontend that organizes the virtual address space into hot and cold regions. The Object Collector monitors access via Guides and SODA, migrates objects using SAMA, and presents a re-organized address space to the OS backend.

more challenging. Thus, the challenge is not merely to reorganize the layout dynamically, but to do so without breaking program correctness under aliasing and concurrency.

n this paper, we demonstrate how to enable object mobility in *unmanaged* languages specifically because of this challenge. We show that address-space engineering is achievable in C++, where stable addresses are the default assumption, by routing accesses through an indirection rather than preserving the address itself; techniques proven in this setting generalize naturally to managed runtimes.

To make the problem tractable in an unmanaged context, we focus on **pointer-based data structures**, where elements are accessed indirectly through pointers. These structures are ubiquitous in memory-intensive applications (Table 2). Rather than attempting to intercept arbitrary raw-pointer dereferences, which is intractable in C++, we route accesses to relocatable objects through an explicit indirection object (a *guide*, §3.2) that the application uses in place of a raw pointer. When an object moves, the system redirects the guide to its new location transparently. This mechanism must operate safely in highly concurrent environments, without resorting to coarse-grained locks or stop-the-world pauses that would negate the performance benefits of an improved memory layout.

In § 3, we show how *OBASE* provides this transparency and concurrency within the C++ execution model, enabling dynamic reorganization.

3 *Object-Based Address-Space Engineering*

Our goal is to engineer an application's address space so that page-based backends can efficiently reclaim or tier memory. *OBASE* achieves this by reshaping object placement: cluster hot objects into dense regions to increase page utilization, and segregate cold objects into separate regions to expose large pools of reclaimable memory.

OBASE operates in environments where the operating system manages memory in page-sized units and may migrate pages across tiers [20, 42, 43, 58]. The design does not assume a particular tiering policy or memory hierarchy; it only assumes that backends observe per-page activity. By presenting backends with regions that are uniformly hot or cold, *OBASE* allows existing mechanisms—from traditional page reclaim (kswapd, zswap) to tiering engines (such as TMO, TPP, Memtis, and HeMem)—to make better decisions without becoming object-aware [43, 44, 49, 56, 62].

Achieving object-level placement with page-level backends poses four challenges: **(C1) object mobility**—C++ ties object identity to its address, so relocation must be transparent and safe under aliasing; **(C2) low-overhead tracking**—hotness classification must run in the common path without degrading performance; **(C3) dynamic adaptation**—hotness shifts continuously (§2.3), so the layout must evolve; and **(C4) safe concurrent migration**—objects must move without global pauses even while threads hold pointers to them.

3.1 System Overview

OBASE's design rests on separating the *layout* problem (organizing objects so pages contain uniformly hot or cold data) from the *tiering* problem (deciding which pages to evict and where to put them). We call systems addressing the layout problem *frontends* and those addressing reclamation *backends*: existing tiering systems like Kswapd, TPP, Memtis, and TMO are backends that assume the layout is given. *OBASE* is a frontend that takes responsibility for layout so any backend can operate on uniformly hot or cold pages.

The following subsections describe how *OBASE* enables object mobility (§3.2), tracks accesses efficiently (§3.3), organizes the address space (§3.4), and migrates objects safely under concurrency (§3.5).

Figure 4 illustrates the architecture. *OBASE* runs in a continuous control loop. All dereferences of managed objects pass through a lightweight *guide* abstraction, which records whether an object was accessed recently. A background *Object Collector (OC)*, periodically processes this metadata to classify objects by access activity and decide whether they should reside in the NEW, HOT, or COLD heaps. Based on this classification, the OC reorganizes the address space by migrating objects between heaps using a safe, lock-free protocol based on *Active Thread Counts (ATC)*. Finally, *OBASE* exposes these organized regions to the OS through a *Spatially-Aware Memory Allocator (SAMA)*, which lays out each heap as a contiguous virtual range, enabling coarse-grained OS hints. This control loop ensures that the virtual address space continuously adapts to the application's shifting working set while presenting page-based backends with clear hot and cold targets.

3.2 Object Mobility in Unmanaged Languages

OBASE decouples an object's logical identity from its location using a lightweight *guide* abstraction. A guide carries the current location of the object as well as additional metadata needed by *OBASE*. Developers interact with objects by using guides rather than raw pointers. When a guide is dereferenced, *OBASE* resolves it to the object's current address and

records that the object was accessed (described in §3.3). The indirection layer is the mechanism that makes later relocation transparent; code that previously operated on pointers continues to operate on guides.

Developers choose which pointers can participate in relocation by marking them with annotations (e.g., the pointers to keys and values in hash-table buckets or the record pointers in B+ trees). In practice this annotation is small: for the ten data structures in our evaluation (Table 2), each managed type carries one to three relocatable pointer fields (typically the structure's authoritative child) while container algorithms and client code are untouched. Guides are enforced through three compiler passes (detailed in §4.7). A type-level pass identifies annotated pointers and rewrites their dereferences to invoke the guide. An instrumentation pass injects hooks at access sites so *OBASE* can observe uses without modifying application logic. A validation pass ensures that managed objects are not used in unsupported ways (e.g., pointer arithmetic over nodes or assumptions about physical contiguity). These passes allow developers to adopt *OBASE* incrementally, starting with the portions of a codebase where object residency matters most.

The division of labor is deliberately narrow: the *developer* marks a small set of pointer fields and guarantees that no unannotated raw pointer aliases a managed object; the *compiler* performs everything else: rewriting declarations and dereferences, and inserting the tracking hooks. An annotated guide carries unique ownership of its object, analogous to `std::unique_ptr`: *OBASE* updates that single guide when relocating, and does not track hidden aliases, so structures that share a node through multiple pointers (e.g., graphs, doubly-linked lists) are out of scope. The validation pass rejects pointer arithmetic and physical contiguity assumptions over managed objects rather than silently miscompiling them. A raw pointer obtained from a guide dereference (e.g., held in a register) remains valid for the duration of the enclosing public operation, because that operation keeps the object's active-thread count above zero and thereby vetoes any concurrent migration (§3.5); callers must not cache such pointers beyond the operation.

Two further programming-model restrictions keep these invariants intact under concurrent migration. First, the `Guide<T>` type appears only inside the data structure's implementation, where Pass 2 rewrites the developer-marked pointer fields (§4.7); it does not appear in calling code. Second, guides do not cross the data structure's public boundary as parameters or return values; they are held as internal fields, and callers interact through primitive-typed APIs (e.g., `HashMap::get(int idx)` in Figure 7). Together these ensure that every guide dereference is reachable from a public-method entry whose TAG participates in the migration protocol.

3.3 Low-Overhead Access Tracking

To classify objects by temperature, *OBASE* must observe which objects are accessed over time, but tracking must be cheap enough to run continuously. Existing mechanisms fall short at both ends of the spectrum. Hardware page table access bits operate at page granularity and cannot distinguish a few hot cache lines from megabytes of cold data on the same page [8, 43]. Software profilers such as DynamoRIO and LLVM's memory profiling provide fine-grained information but impose prohibitive overheads for always-on deployment [12, 14]. Hardware sampling via PEBS offers lower overhead but provides statistical coverage rather than precise per-object tracking [7].

Instead, *OBASE* embeds access tracking directly into guide pointer dereferences, yielding object-level information without significant overhead. A guide overloads the dereference operator so that each access records that the underlying object was used. Modern 64-bit architectures reserve high-order address bits in canonical user-space pointers, so *OBASE* stores a small amount of per-object metadata in these unused bits [16]. Inline metadata avoids external side tables and ensures that recording access is part of normal pointer use.

Metadata is updated on every dereference using a single atomic read–modify–write (RMW): an `accessed` bit is set if it has not already been set, thereby skipping subsequent updates to avoid unnecessary cache-coherence traffic for frequently touched objects. This design keeps the common path small, and the resulting per-access cost is comparable to a cache hit. The metadata also holds state used by the runtime to classify objects or coordinate relocation, without requiring separate allocations or indirection.

To let the runtime observe all managed objects without walking application-specific pointer graphs, *OBASE* maintains a Sparse Object Data Activity (SODA) bitmap (§4.3) over the process heap. SODA records which virtual regions contain managed objects and enables the Object Collector (OC) to discover objects by scanning these regions rather than interpreting container internals.

3.4 Dynamically Engineering the Layout

The access signals from §3.3 feed into a layout policy that continuously reorganizes the virtual address space. OBASE groups objects by observed temperature so that page-based backends see large, uniform regions rather than interleaved hot and cold data. Three logical temperature heaps capture this temperature: NEW holds freshly allocated objects whose access pattern is not yet clear, HOT holds the current working set, and COLD holds objects inactive for multiple scan periods. Each heap occupies a dedicated virtual address range, so an object's address directly encodes its temperature.

Objects move between heaps as access intensity changes (Figure 5): inactive HOT objects are demoted to COLD, and COLD objects that become active again are promoted back.

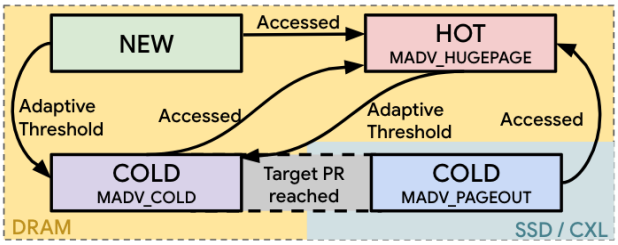


Figure 5. Object Migration State Diagram. Objects transition between heaps based on access intensity. The Object Collector promotes accessed objects to HOT and demotes inactive objects to COLD, allowing SAMA to apply different `madvise` policies to each region if required.

This continuous reclassification tracks the workload's evolving working set rather than its allocation history.

To realize these logical heaps in the virtual address space, *OBASE* employs a Spatially-Aware Memory Allocator (SAMA) that reserves a contiguous virtual address range for each heap and sub-allocates objects within that range. Contiguity is a deliberate design choice that allows coarse-grained OS hints to be applied to whole heaps rather than individual pages, exposing large pools of cold memory to page-based tiering systems. The Object Collector (OC) periodically scans SODA to classify each managed object. Objects touched since the last scan are candidates for promotion; objects that remain untouched accumulate evidence of coldness.

Reacting to a single inactive window would make classification sensitive to transient bursts. *OBASE* therefore tracks a per-object Consecutive Inactive Window(CIW) counter: each scan window without an access increments CIW; any access resets it to zero. Objects whose CIW exceeds a cold threshold become eligible for demotion to COLD; objects in COLD that are accessed rejoin HOT. This hysteresis ensures that only sustained inactivity triggers migration to COLD.

The cold threshold C_t governs how long an object must remain inactive before migration. Too aggressive and COLD objects are frequently re-accessed; too conservative and reclaimable memory lingers in HOT. *OBASE* adapts C_t using a promotion-rate target. We define the promotion rate (PR) as the fraction of the working set drawn from COLD heap in each scan window:

$$PR = \frac{\text{unique COLD pages accessed}}{\text{working set size}} \times \frac{60}{\text{scan interval (s)}}$$

Crucially, *OBASE* measures COLD-heap accesses regardless of where those pages physically reside—it cannot determine which tier a page occupies and does not attempt to. If the observed rate exceeds a target, C_t increases by one window; if below, C_t decreases. This additive adjustment converges to a workload-specific regime. The goal is not to minimize COLD-heap accesses, but to ensure that pages classified as COLD are *safe targets* for any backend policy. §4 details initialization and tuning.

By design, *OBASE* is decoupled from any specific backend. The base system reorganizes the address space but issues no reclamation hints. This separation is intentional: the value of address-space-engineering lies in making pages uniformly hot or cold, which improves the precision of any page-based mechanism. Backends such as kswapd, TMO, TPP, and Memtis observe per-page activity through their existing interfaces (PTE-A bits, PEBS samples, or PSI signals) and naturally make better decisions when COLD pages contain only cold objects.

Optionally, *OBASE* can issue proactive hints to accelerate reclamation. In this hinted mode, once the promotion rate stabilizes below the target, *OBASE* marks COLD pages with `MADV_COLD` or `MADV_PAGEOUT` to signal that they are safe to reclaim. Similarly, SAMA may request huge pages for HOT (`MADV_HUGEPAGE`) to selectively improve TLB coverage over dense hot data. These hints are strictly advisory and complement rather than replace backend policies.

3.5 Safe Concurrent Migration

Reorganizing the address space requires relocating objects while application threads may hold pointers to them. In managed languages, garbage collectors solve this problem using load barriers or stop-the-world pauses. C++ offers neither: there is no runtime to intercept pointer loads, and production systems cannot tolerate pauses. OBASE must therefore migrate objects without blocking threads, while guaranteeing that every dereference sees a valid location. As described in §3.2, all dereferences pass through guides, so relocation reduces to atomically updating a single pointer-sized word. Callers never follow stale pointers and do not need explicit synchronization.

Lightweight Scoping. An object can be migrated only when no thread is actively using it. Classic approaches—stop-the-world pauses or per-access load barriers—are unsuitable for C++ systems code [30, 50, 63]. Instead,*OBASE* introduces a per-object *Active Thread Count* (ATC) that tracks how many public data structure operations have observed a guide during a migration window. The ATC captures the notion of active use: if a thread begins an operation that may dereference an object, ATC is incremented once for that scope; when the operation completes, the ATC is decremented. Migration is permitted only when the ATC reaches zero, indicating that no thread is currently executing an operation that could read or modify the object.

The ATC must be updated efficiently. Rather than incrementing the ATC on every pointer dereference—which would impose unacceptable overhead—*OBASE* scopes tracking to public API boundaries. When a thread enters a data-structure operation (e.g., get, insert), it registers with a Thread-local Active scope guard (TAG); when the operation completes, the guard decrements the ATC for all objects touched. This design reflects how C++ programmers reason about pointer validity: pointers are valid for the duration of the operation that

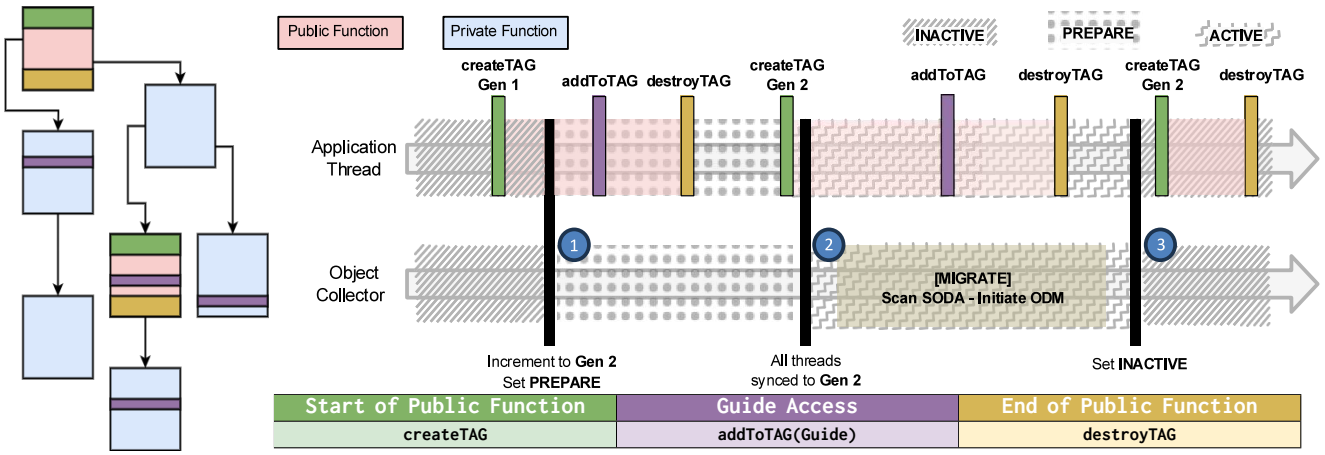


Figure 6. Application Thread Execution Flow And Interaction With OC The left side shows the instrumented call graph, code is inserted in three scenarios as shown. Public functions create and destroy Thread-local Active scope Guards (TAGs), maintaining nesting levels and registering the thread in the Thread Activity Index (TAI). Guides increment active reference counts only if added to the TAG. Reference counts are decremented only when the outermost public function exits. Active Thread Count (ATC) is enabled only during `PREPARE` and `ACTIVE` states.

obtained them, not indefinitely. Compiler instrumentation (detailed in §4.2) inserts the necessary hooks automatically.

Active Windows via Epochs. Always-on ATC tracking would impose overhead even when no migration is in progress. *OBASE* therefore activates ATC only during periodic migration epochs. Outside these windows, guide dereferences record only access activity (§3.3) with no ATC overhead. When the OC initiates migration, it coordinates an epoch transition that ensures all active threads enable ATC tracking before any object is moved. This epoch-based activation confines the synchronization cost to brief, infrequent windows.

Optimistic Migration. Given ATC=0, *OBASE* moves objects using an optimistic protocol inspired by optimistic concurrency control. The Object Collector copies the object to its target heap, then attempts to atomically swing the guide to the new address. If any thread accesses the object during the copy, the commit fails and the move is abandoned—the object remains in place, and the thread sees valid data. This optimistic approach has two key properties: (1) threads never block on migration, and (2) concurrent access safely vetoes relocation rather than observing inconsistent state. §4.5 details the CAS-based protocol.

Safety and Non-Blocking Progress. Safety follows from a single invariant: the guide is updated atomically, and migration aborts on any conflicting access. No thread can ever follow a stale pointer. Progress is non-blocking: application threads never wait on the collector, and the collector performs bounded work per object before committing or abandoning. Frequently accessed objects naturally resist migration (their ATC rarely reaches zero), while cold objects eventually move. The result is an address space that reshapes continuously without stop-the-world pauses, while preserving familiar pointer semantics.

4 Implementation

This section details *OBASE*'s concrete realization.* We cover the guide metadata encoding (§4.1), scope-guard data structures (§4.2), SODA bitmap layout (§4.3), controller parameters (§4.4), the epoch-based migration protocol (§4.5), kernel reclamation optimizations (§4.6), and the compiler passes that automate guide management (§4.7).

4.1 Guide Metadata Encoding and Heap Allocation

Each guide uses the 48 bits required for canonical x86-64 user-space addresses and repurposes the upper 16 bits for metadata: 7 bits encode ATC (supporting up to 128 concurrent threads per object), 5 bits track CIW (up to 32 windows, or 60+ minutes at the default 120s interval), 2 bits identify the current heap (NEW, HOT, COLD), and 2 bits store the access and migration-lock flags. As 128-bit addressing becomes more prevalent [5], implementations can expand these fields without changing the guide abstraction. We implement the three heaps using a Spatially-Aware Memory Allocator (SAMA) built on jemalloc's extent management. SAMA reserves large `mmap` regions for each heap and allocates objects within them, returning unused extents to the OS as objects migrate or are freed.

4.2 Efficient Scope Guard Tracking

Figure 6 illustrates the instrumented call graph. The compiler inserts three hooks: `createTAG` (green) at public API entry, `addToTAG(guide)` (purple) at each guide dereference, and `destroyTAG` (yellow) at public API exit. Public functions may

OBASE is open-sourced at https://github.com/WiscADSL/obase

call private helpers that also dereference guides; the TAG tracks all such accesses but only decrements the ATC when the *outermost* public function returns. This nesting-aware design ensures that the ATC remains positive throughout the entire operation, even when internal helpers are inlined or called multiple times.

We implement TAGs using a `BaseDeltaPtrSet` that exploits pointer locality: it encodes pointers as a base address plus 32-bit deltas, grouping up to 16 nearby pointers within two cache lines. Insertion is O(1) when the pointer falls within an existing group; otherwise a new group is created in O(log G) time, where G is the number of groups. Since pointers within a single operation cluster tightly (e.g., keys and values in the same bucket, nodes along a tree path), most insertions hit existing groups. Across the ten data structures in our evaluation, the median number of unique guides per operation ranges from 3 (hash tables) to 12 (B+Tree traversals), keeping per-operation TAG overhead under 100 ns.

4.3 SODA Bitmap Structure and Object Discovery

The Sparse Object Data Activity (SODA) bitmap uses a two-level structure to cover the heap address space without allocating storage for empty regions. SODA divides the address space into coarse-grained blocks; each block contains 64-bit words where individual bits indicate guide presence at fixed-size slots. Blocks are allocated lazily and reclaimed when empty. Because SODA tracks guide pointers rather than object addresses, it remains valid across relocations; the guide's address does not change when the object moves. Memory overhead is one bit per potential guide slot plus block-level bookkeeping. The OC scans SODA at a configurable interval (120 s by default), chosen to align with cold-memory detection thresholds in warehouse-scale tiering systems [29, 43].

4.4 Cold Threshold Controller

OBASE initializes C_t to three scan windows; at the default 120 s interval, this is approximately six minutes of inactivity before demotion, consistent with the five-minute rule for tiered storage [19, 32, 33]. The target promotion rate of 1% is based on budgets used in production compressed-memory and CXL tiering deployments [29, 43]. C_t is bounded between 1 and 32 windows. After each scan window, an additive-increase/additive-decrease (AIAD) loop (Algorithm 1) compares the observed promotion rate PR_{actual} (§3.4) against the target and adjusts C_t accordingly, so the threshold settles into a workload-specific regime without large swings.

4.5 Epoch Protocol and Optimistic Migration

As shown in Figure 6 the OC coordinates migration through three states (`INACTIVE`, `PREPARE`, and `ACTIVE`) using a global epoch counter and a Thread Activity Index (TAI) - a per-thread slot that lets the OC verify, without blocking threads, that every active thread has observed the new epoch before migration begins.

Algorithm 1 Cold-Threshold Controller (per scan window)

$$PR_{actual} \leftarrow \frac{\text{unique COLD pages accessed}}{\text{working-set size}} \times \frac{60}{\text{scanInterval}}$$

if $PR_{actual} > PR_{target}$ **then** ▷ demotions revisited too often
  $C_t \leftarrow \min(C_t + 1, 32)$ ▷ demote more conservatively
else if $PR_{actual} < PR_{target}$ **then** ▷ slack in the budget
  $C_t \leftarrow \max(C_t - 1, 1)$ ▷ reclaim more aggressively
end if

Time	Actor	Action	ATC	Lock	Outcome
(a) Concurrent dereference aborts migration:					
t_0	OC	read guide	0	0	eligible
t_1	OC	CAS: set lock	0	1	copying begins
t_2	Thread	dereference	1	0	lock cleared
t_3	OC	CAS: commit	*mismatch*		**aborted**
(b) No interference, migration succeeds:					
t_0	OC	read guide	0	0	eligible
t_1	OC	CAS: set lock	0	1	copying begins
t_2	OC	CAS: commit	0	0	**success**

Table 1. Race between migration and concurrent access. (a) A dereference at t_2 modifies the guide, causing the OC's commit CAS to fail. (b) When no thread intervenes, both CAS operations succeed and the object moves.

Epoch Transitions. In the `INACTIVE` state, the ATC updates are disabled; dereferences only set access bits. When the OC initiates migration, it increments the epoch counter and enters `PREPARE` state 1. Threads entering public methods record the current epoch in the TAI (a small array indexed by thread-ID hash); exits clear their slot. The OC repeatedly scans the TAI; once every non-empty slot reflects the new epoch 2, all active threads have enabled ATC tracking, and the OC enters the `ACTIVE` state . After migration completes, the OC returns to `INACTIVE` state 3. Crucially, threads never block; they simply record epoch participation, while the OC performs convergence checks.

Optimistic Data Migration. Within `ACTIVE` states, the OC migrates objects using an Optimistic Data Migration (ODM) protocol (Algorithm 2), similar in spirit to Optimistic Concurrency Control (OCC), where a job is performed first and verified later, essentially acting as a weak transaction. For a candidate with ATC=0 and migration-lock clear, the OC: (1) atomically sets the migration-lock bit on the guide slot; (2) allocates space in the target heap via SAMA and copies the object; (3) constructs a new guide (new address, heap ID, reset CIW, cleared lock) and publishes it with a single commit CAS, the same atomic both installs the new address and releases the lock. If the commit CAS fails, a concurrent dereference has changed the guide and the migration is abandoned (the OC explicitly clears the lock it set; see Algorithm 2).

Race Resolution. Table 1 illustrates a race between migration and access. Every dereference performs an atomic

Algorithm 2 Optimistic Data Migration (ODM)

```
1:  procedure MigrateObject(ptr, targetHeap)                    ▷
    ptr: address of guide slot
2:      guide ← AtomicLoad(ptr)                                 ▷ Load object guide
3:      if RefCount(guide) > 0 then
4:          return false                                        ▷ Object in use
5:      end if
6:      srcAddr ← ExtractAddr(guide)
7:      srcHeap ← ExtractHeapType(guide)                        ▷ Get current heap type
8:      SetMigrationLock(ptr)                                   ▷ CAS; mark as migrating
9:      size ← GetSize(srcAddr)
10:     dstAddr ← samalloc(size,targetHeap)
11:     if dstAddr = null then
12:         ClearMigrationLock(ptr)
13:         return false                                        ▷ Allocation failed
14:     end if
15:     Copy(dstAddr,srcAddr,size)                              ▷ Copy data
16:     newGuide ← CreateGuide(guide,dstAddr,targetHeap)        ▷
    Inherits lock from guide metadata
17:     ClearMigrationLock(newGuide)                            ▷
    Local clear of inherited lock; commit via CAS below
18:     success ← AtomicCAS(ptr,guide,newGuide)
19:     if success then
20:         Free(srcAddr,srcHeap)                               ▷ Release old memory
21:         return true
22:     else
23:         Free(dstAddr,targetHeap)                            ▷ Rollback
24:         ClearMigrationLock(ptr)                             ▷ CAS; release lock we set
25:         return false                                        ▷ Migration failed, guide changed
26:     end if
27: end procedure
```

CAS that sets the access bit and *clears* the migration-lock bit. If a thread accesses object x while the OC is copying it (t_2), the guide changes, the commit CAS fails (t_3), and x remains at its original address. The thread always sees valid data; concurrent access supersedes migration.

4.6 Kernel Page Reclamation Optimization

When objects migrate to the COLD heap and the Object Collector issues `MADV_PAGEOUT`, Linux's page reclamation path becomes the bottleneck for efficient memory tiering. The default path in `shrink_folio_list()` processes each page individually: it clears the page table entry (PTE), issues a TLB flush or shootdown that triggers an inter-processor interrupt (IPI) to every core, and submits the page to the block I/O layer. This fine-grained approach creates substantial overhead when reclaiming large regions, as the cumulative cost of per-page TLB invalidations and IPIs degrades performance even for threads accessing unrelated hot data.

We modify `shrink_folio_list()` to batch these operations: it aggregates pages (up to a full PMD spanning 512 base pages), clearing each PTE and marking it for pageout, but defers TLB invalidation until a complete batch is prepared, then issues one flush over the entire range and submits all pages together for I/O. By amortizing TLB shootdowns and I/O submissions, this reduces IPIs by more than 99% when demoting 10 GiB of memory versus the unmodified kernel. Beyond local tiering, batched invalidation also addresses a scalability bottleneck in RDMA-based far-memory systems [35, 48, 57], where per-page shootdowns cause IPI storms whose latencies grow super-linearly with thread count; aggregating across hundreds of pages lets the reclamation path scale for both CXL and RDMA backends.

```
1  void HashMap::get(int idx) {
2      createTAG();                    // public entry
3      Guide<char>& v = buckets[idx]; // HashMap member
4      addToTAG(&v);                   // before deref
5      if (*v > 37) { destroyTAG(); return; }
6      addToTAG(&v);  *v = 42;
7      addToTAG(&v);
8      std::cout << "Value: " << *v << std::endl;
9      destroyTAG();                   // public exit
10 }
```

Figure 7. Compiler transformation. The developer marks one pointer field of `HashMap`; the pass rewrites its type from `char*` to `Guide<char>`, brackets the public method with `createTAG`/`destroyTAG`, and inserts `addToTAG` before each guide dereference. The `buckets` array is unmarked and stays in memory, so indexing itneeds no guard; only the guide dereference (`*v`) does. Callers invoke `HashMap::get` with a primitive key (no `Guide` crosses the public-method boundary), so caller code neither holds nor dereferences a `Guide`. ATC increments inside `addToTAG` only when the TAG state is not `INACTIVE`.

4.7 Compiler Passes for Guide Management

OBASE uses three complementary compiler passes, implemented on Clang and LLVM, to convert raw pointers to guides and insert the TAG/ATC instrumentation, confining developer effort to marking which pointer fields are managed. **Visibility extraction.** A Clang frontend pass ensures TAGs are created only at public boundaries by using a `RecursiveASTVisitor` to find public methods that serve as data-structure entry points (e.g., `get`, `set`, `delete`) and records their visibility for later stages. **Pointer-to-guide conversion.** A second Clang pass uses the rewriter to convert the developer-listed pointer declarations and usages into guides, giving precise control over which objects *OBASE* manages. **IR instrumentation.** The third pass runs on the LLVM IR. A fixed-point analysis first identifies the set of functions that directly use guides (via conversions, destructor calls, operator overloads, or assignments), propagates this set through the call graph to find functions that touch guides indirectly, and combines it with the visibility data from Pass 1 to classify each function. It then inserts `createTAG`/`destroyTAG` at the entry and exit of public functions that touch guides, and `addToTAG` before each guide dereference in all transformed functions (public and private). This restricts TAG creation to public boundaries while still tracking every access throughout the call stack (Figure 7).

Structure	Concurrency Control	Used In
HashTable Harris [36]	Lock-free algorithm	NGINX
HashTable Pugh [54]	Fine-grained r/w lock	Redis, Memcached
HashTable Java CHM [15]	Segmented bucket locks	Linux kernel, HAProxy
SkipList Coarse	Global lock	LevelDB/RocksDB
SkipList Fraser [31]	Lock-free algorithm	Redis Sorted Sets
SkipList Herlihy [37]	Optimistic fine-grained	Cassandra, CockroachDB
B+Tree Coarse	Global lock	SAP HANA
B+Tree OCC	OCC w/ epoch reclaim	VoltDB index
MassTree [47]	OCC + RCU	MICA, Silo
Adaptive Radix Tree [45]	Fine-grained r/w lock	DuckDB, PostgreSQL

Table 2. Concurrent data structures evaluated with *OBASE*. These structures span lock-free, fine-grained, and coarse-grained concurrency mechanisms, demonstrating *OBASE*'s compatibility with diverse synchronization approaches.

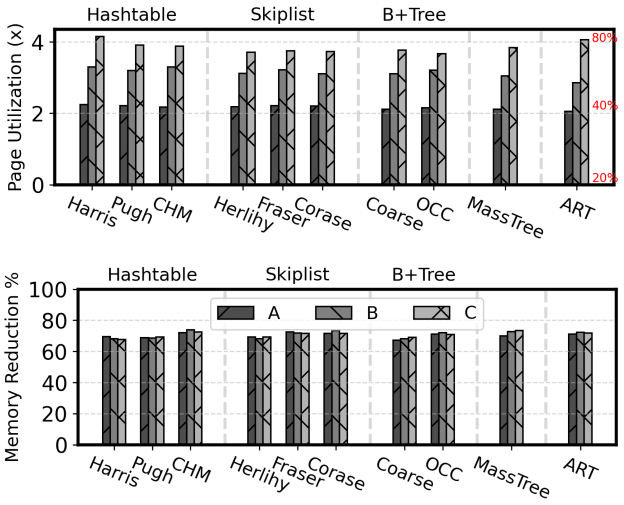


Figure 8. *OBASE* effectiveness (YCSB, 10M keys). Top: Page utilization improvement relative to the baseline allocator; *OBASE* increases utilization by 2–4× across workloads and data structures. Bottom: RSS reduction after *OBASE* pages out the COLD heap (via Kswapd). Memory footprint shrinks by 65–72%.

5 Evaluation

We evaluate *OBASE* along four dimensions:

E1: Effectiveness. Does *OBASE* reduce hotness fragmentation and memory footprint across different data structures and workloads? (§5.2)

E2: Backend synergy. How much does *OBASE* improve the effectiveness of existing page-based reclamation and tiering backends? (§5.3)

E3: Overhead and scalability. What runtime overheads do tracking and migration introduce, and how do they scale with thread count? (§5.4)

E4: Dynamic behavior. How does *OBASE* adapt to changing hotsets in long-running, real-world workloads? (§5.5)

5.1 Experimental Setup

All experiments run on an Intel Xeon Gold 5218 (16 cores, SMT disabled) server configured in `performance` governor mode with Ubuntu 22.04 and Linux kernel 6.12. The memory subsystem comprises two tiers: a fast tier of 2×16 GB DDR4 DRAM modules at 2400 MHz, and a slow tier of 4×128 GB Intel Optane DC Persistent Memory 100 modules at 2666 MHz. All six memory devices occupy distinct channels to avoid interface. We configure Optane PMEM in `Memory Mode` and expose it as a separate NUMA node, creating a two-tier hierarchy representative of emerging CXL-attached memory deployments [46, 49]. The Optane tier provides approximately 2.5× higher access latency than local DRAM [64], consistent with first-generation CXL memory characteristics [58]. For reclamation experiments requiring swap, we use a 512 GB Intel P4800X Optane SSD.

CrestDB testbed. We implemented CrestDB, a concurrent in-memory key-value store, to evaluate *OBASE* across diverse data structures and concurrency mechanisms. CrestDB integrates ten high-performance data structures spanning the concurrency-control spectrum (Table 2), from lock-free algorithms to global locks. Many structures are borrowed from ASCYLIB [34], which provides production grade implementations. This diversity demonstrates that *OBASE*'s compiler instrumentation and migration protocol are compatible with a wide range of synchronization approaches without structure-specific modifications. All data structures maintain guide pointers to key and value objects; CrestDB deep-copies inserted data, ensuring *OBASE* manages the authoritative copy rather than application-held aliases. Unless otherwise noted, CrestDB runs with six server threads and six client threads.

Workloads. We use the YCSB benchmark suite with Zipfian-distributed keys to model skewed access patterns typical of production workloads. To evaluate real-world adaptivity, we replay production traces from Meta (CacheLib [22], DBbench [24]) and Twitter (Cluster 7, Cluster 23) [65].

Controller parameters. Unless otherwise noted, the Object Collector scans every 120 s, targets a 1% promotion rate, and adjusts C_t by ±1 window per scan within $1 \le C_t \le 32$ (§3.4); the conservative 1% target follows production compressed-memory deployments [43, 62]. The optimal rate is hardware-dependent (faster tiers tolerate higher cold-access fractions), and hardware-aware tuning is future work. §5.5 examines C_t dynamics over time.

5.2 Effective Address-Space-Engineering (E1)

We first evaluate whether *OBASE* achieves its core objective: reducing hotness fragmentation and converting unreclaimable cold data into reclaimable pages. We run CrestDB with all ten data structures under YCSB workloads A, B, and C with Zipfian keys. We load 10M keys with 30-byte keys and 1024-byte values, creating a 13 GiB dataset. Unless stated otherwise, all results in this section are measured after *OBASE* has converged (promotion rate below the 1% target).

Page utilization. Figure 8(a) reports page utilization before and after *OBASE* reorganizes objects into NEW, HOT,

and COLD heaps. Initially, the data structures exhibit 18–20% utilization when measured over 120 s windows: most pages contain only a handful of accessed cache lines, reflecting the hotness fragmentation observed in production traces (§2).

After three scan intervals, the Object Collector classifies objects based on guide access bits and migrates them into HOT or COLD heaps. Compared to the baseline, *OBASE* improves page utilization by 2× for workload A (50% writes), approximately 3× for workload B (5% writes), and up to 4× for the read-only workload C. Across data structures, absolute utilization after convergence ranges from roughly 40% (workload A) to 80% (workload C).

The variation across workloads reflects how the NEW and HOT heaps interact. Workload C has no updates: once objects are classified as hot, they remain in the HOT heap and no new objects are allocated there. Nearly all accessed bytes end up densely packed in a small number of HOT pages, and utilization approaches 80%. In workload B, occasional updates allocate fresh values in NEW, so the working set splits between NEW and HOT and overall utilization stabilizes around 60–70%. Workload A performs frequent updates, continuously injecting new objects into NEW. Utilization still roughly doubles, but cannot reach read-only levels because a larger fraction of hot objects reside in NEW during their initial epochs.

The consistency of the improvement across ten structurally different data structures shows that *OBASE*'s benefits derive from object-temperature clustering rather than data-structure-specific layout optimizations.

Memory footprint. Higher page utilization translates directly into reclaimable cold memory. Once the promotion rate falls below the 1% target—typically after 3–4 scan intervals (6–8 minutes) for YCSB—*OBASE* proactively issues `madvise(MADV_PAGEOUT)` on the COLD heap. Figure 8(b) shows the resulting RSS reduction relative to a baseline without reclamation.

Across all data structures and workloads, *OBASE* reduces RSS by 65–72%. For workload B with 10M keys, the baseline uses 12.4 GiB; after *OBASE* converges and pages out COLD, RSS drops to 3.5–4.0 GiB. Because COLD pages contain almost exclusively inactive objects, proactive paging does not cause swap-in storms or noticeable throughput degradation (we quantify overheads in §5.4).

Takeaway #1: *OBASE improves page utilization across all data structures as it tracks object hotness without the semantic knowledge of each structure. This enables uniform hotness fragmentation reduction across diverse concurrency mechanisms.*

5.3 Backend Synergy (E2)

The memory savings demonstrated in §5.2 are valuable only if OS tiering backends can exploit them without degrading performance. We now show that *OBASE* enables existing reclamation and tiering systems to achieve aggressive memory savings while preserving throughput.

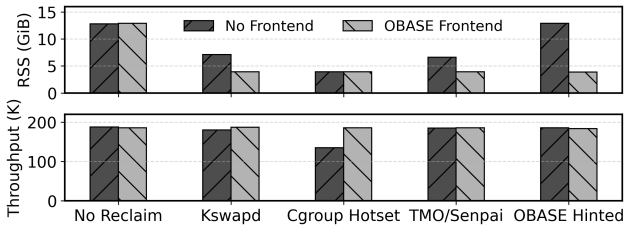


Figure 9. *OBASE* with reclamation backends (YCSB-C, MassTree). Top: RSS after convergence. Bottom: throughput. Without *OBASE*, backends face a trade-off between memory savings and performance. With *OBASE* (hatched bars), all backends achieve near-optimal memory savings with minimal throughput loss.

5.3.1 Paging-based Reclamation Backends. We run CrestDB with MassTree under YCSB-C in a memory-constrained configuration: the workload has a 13 GiB footprint but an active working set of approximately 4 GiB. We compare four reclamation strategies, each with and without *OBASE* as a frontend:

- **Kswapd:** Linux's background reclaimer, triggered by memory pressure from a co-located process.
- **Cgroup:** Memory limit set to working-set size (4 GiB), forcing aggressive reclamation.
- **TMO:** Meta's PSI-based proactive reclaimer [62].
- ***OBASE* Hinted:** Proactive `MADV_PAGEOUT` on the COLD heap after convergence.

Throughout the evaluation, *OBASE* denotes the frontend alone, address-space reorganization with proactive paging disabled, so combinations like *OBASE*+TMO isolate the benefit of reorganization from any hinting. *OBASE* Hinted additionally issues `MADV_PAGEOUT` on the COLD heap.

Figure 9 reveals a fundamental trade-off that *OBASE* resolves. **Without *OBASE***, backends must choose between memory efficiency and performance. Kswapd reduces RSS from 13 GiB to 7 GiB (1.8×) with no throughput loss, but leaves 3 GiB of cold data trapped in mixed-temperature pages due to its page level view through PTE scans. Cgroup reaches 4 GiB (3.2×), but throughput collapses by 38% as the kernel inevitably evicts hot objects. TMO achieves a 6.5 GiB RSS (2×) with no throughput loss, but cannot reclaim further because PSI signals page-level pressure regardless of object-level coldness.

With *OBASE*, all backends reach 4 GiB RSS—matching the most aggressive policy—with no throughput degradation. Kswapd now reclaims COLD pages preferentially. Cgroup no longer thrashes because evicted pages contain genuinely cold objects. TMO's PSI probes no longer encounter mixed-temperature pages that resist reclamation. *OBASE* Hinted achieves the same result proactively, without relying on any backend policy.

Takeaway #2: *OBASE achieves aggressive reclamation while preserving application performance.*

5.3.2 Tiering Backends. To demonstrate the memory tiering benefits of reduced hotness fragmentation, we evaluate

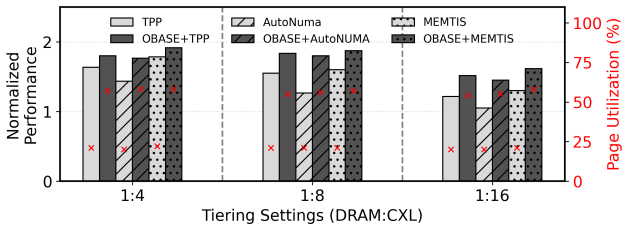


Figure 10. *OBASE* with tiering backends (YCSB-B, MassTree, 50M keys). Throughput normalized to CXL-only baseline (higher is better). Without *OBASE*, performance degrades as DRAM shrinks because the hot set cannot fit. With *OBASE*, the hot set compacts, enabling stable performance even at 1:16.

OBASE with three page based migration systems: TPP [49], AutoNUMA [60], and Memtis [44].

Setup. We load CrestDB (MassTree) with 50M keys (30-byte keys, 1024-byte values), a 67 GiB dataset. We vary the DRAM:CXL ratio across three configurations: 1:4 (14.8 GiB DRAM), 1:8 (7.4 GiB DRAM), and 1:16 (3.9 GiB DRAM), with the remaining capacity on Optane PMEM. As in [44], performance is normalized to a baseline where all data resides on the slow tier; values above 1.0× indicate speedup from effective DRAM use.

The hot-set mismatch. Without *OBASE*, the working set spans 16.3 GiB of pages at 21% utilization— slightly larger than the 1:4 DRAM budget of 14.8 GiB. No configuration can keep all accessed data in DRAM. With *OBASE*, the same logical working set compacts to 6.33 GiB at 57% utilization, fitting comfortably at 1:4 and 1:8 ratios.

TPP. TPP uses hysteresis-based promotion and proactive demotion to manage DRAM headroom. Figure 10 shows TPP achieves 1.65× speedup at 1:4, degrading to 1.25× at 1:16 as the DRAM budget shrinks below the fragmented hot set. With *OBASE*, TPP reaches 1.85× at 1:4 and retains 1.45× at 1:16—a 16% improvement at the most constrained ratio.

AutoNUMA. AutoNUMA promotes any accessed remote page immediately, without hysteresis, causing thrashing when the hot set exceeds DRAM capacity. Without *OBASE*, AutoNUMA underperforms TPP by 15–20%, achieving only 1.05× at 1:16—barely better than CXL-only. With *OBASE*, AutoNUMA matches TPP-alone performance: at 1:8, *OBASE*+AutoNUMA (1.6×) exceeds TPP alone (1.55×). When pages are uniformly hot or cold, even naive promotion decisions become correct.

Memtis. Memtis uses hardware sampling (PEBS) to identify hot pages, achieving the best baseline results: 1.8× at 1:4 and 1.55× at 1:16. With *OBASE*, Memtis improves to 1.95× and 1.7×, respectively. The gains are smaller (3–10% vs. 12–29% for TPP/AutoNUMA) because Memtis already captures much of the page-level signal—but even the most sophisticated page-level policy benefits from object-level reorganization.

Because *OBASE* reduces the effective hot set by 2.5×, ratio **1:X** performs comparably to baseline at **1:(X/2)**. For example, *OBASE* + TPP at 1:16 reaches 1.45×, within ~6% of TPP alone at 1:8 (1.55×). For operators, this means the same performance with half the DRAM, or double the effective capacity of existing tiered deployments.

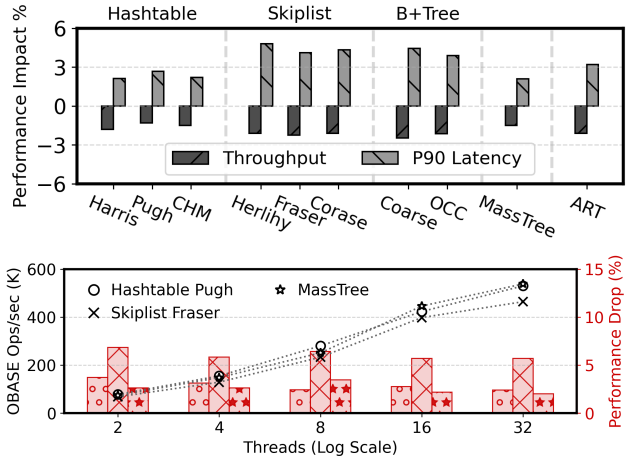


Figure 11. *OBASE* overhead and scalability (YCSB, 10M keys). Top: Throughput and p90 latency overhead across data structures. Overhead ranges from 1.5–5% depending on structure. Bottom: Scalability from 2 to 32 threads. Bars show absolute throughput; markers show overhead relative to baseline. Overhead remains bounded at 1–8% with no upward trend.

Takeaway #3: *OBASE compacts the hot set so it fits in smaller DRAM budgets, making page-based tiering backends effective.*

5.4 Overhead and Scalability (E3)

Given the memory savings and backend improvements demonstrated above, we now quantify *OBASE*'s runtime costs.

5.4.1 Steady-State Overhead. Figure 11(a) reports throughput and p90 latency overhead when no reclamation or tiering backend is active, normalized to an uninstrumented baseline.

On average, *OBASE* reduces throughput by 2.5% and increases p90 latency by 5%. Hash tables see the smallest impact (1.5–3% throughput drop), while skiplists, B+Trees, and ART experience 3–5% overhead. This variation correlates with the number of guides touched per operation: hash table lookups dereference few nodes, whereas tree traversals visit more nodes and incur proportionally more tracking overhead.

The overhead has two main sources: (1) a tagged-pointer read-modify-write on each guide dereference (4–5 ns, comparable to an L1 cache hit), and (2) TAG/ATC bookkeeping during `ACTIVE` migration epochs. The Object Collector runs in a dedicated thread and consumes less than 1% of CPU time.

5.4.2 Thread Scalability. A natural concern is whether *OBASE*'s atomic operations become contention bottlenecks at higher thread counts. We measure scalability by varying CrestDB server threads from 2 to 32 on three representative data structures spanning different synchronization mechanisms: Hashtable Pugh (fine-grained locking), Skiplist Fraser (lock-free), and MassTree (OCC with epoch reclamation).

Figure 11(b) shows that throughput scales and overhead remains bounded at 1–8% regardless of thread count. Critically, overhead does not increase with concurrency: all three data structures exhibit similar overheads at 2 and 32 threads.

This scalability follows from *OBASE*'s design: guide metadata updates target per-object state (no cross-thread contention), the test-and-set optimization skips redundant writes for hot objects, TAGs are thread-local, and ATC increments occur only during brief `ACTIVE` epochs. The 2–5% overhead is measured against a DRAM-only baseline with no memory pressure—an idealized scenario that production systems rarely enjoy. In the tiered-memory environments where *OBASE* is designed to operate, the comparison reverses: as §5.3 showed, backends *without OBASE* suffer 10–38% throughput loss from poor page-selection decisions, while backends *with OBASE* match DRAM-only performance.

Takeaway #4: *OBASE imposes 1.5–5% overhead across data structures and stays within 8% from 2 to 32 threads, a modest cost relative to the backend improvements in §5.2–5.3.*

5.5 Real World Traces

Synthetic workloads exercise controlled forms of skew, but production systems exhibit substantially more complex behavior: shifting hotsets, mixed read/write/delete ratios, and locality patterns that evolve over hours. We therefore evaluate *OBASE* on four real-world traces to assess whether its fragmentation-reduction benefits generalize beyond YCSB and whether the feedback controller adapts stably to long-term changes in access patterns.

We evaluate four traces that span a range of access patterns:

- **Meta CacheLib** [22]: Read-heavy (83% GET) with gradually shifting popular keys.
- **DBench Mixgraph** [24]: Models Meta's ZippyDB with key-range locality across 30 prefixes. Read-heavy (85% GET, 14% PUT, 1% SEEK).
- **Twitter Cluster 7** [65]: High skew (α=1.07) with small, concentrated working set of reads and writes.
- **Twitter Cluster 23** [65]: Write-heavy (31% SET, 30% INCR) with low skew (α=0.274) and deletes.

These traces cover the spectrum from highly skewed (Cluster 7) to nearly uniform (Cluster 23), and from read-dominated (CacheLib) to write-heavy (Cluster 23). We replay each trace on CrestDB with ART and measure memory reduction and page utilization improvement.

Page utilization (E1). Page utilization improves by 1.8–3.4× across traces. Cluster 23 shows the highest gain (3.4×) because its low skew disperses accesses across many keys, yielding very low baseline utilization. Cluster 7's high skew naturally concentrates accesses, so the baseline is already reasonable and the relative improvement is smaller (1.8×).

Memory reduction (E1, E2 & E4). Figure 12 shows that *OBASE* Hinted reduces RSS by 36–58% compared to no reclamation. Cluster 7 achieves the largest reduction (58%) because its high skew concentrates the working set into fewer hot objects; DBench shows the smallest (36%) because key-range locality spreads accesses more uniformly. Adding *OBASE* to TMO provides 15–30% additional savings relative to TMO-alone, demonstrating that *OBASE* and TMO are complementary: TMO identifies reclaimable pages, while *OBASE* ensures those pages contain uniformly cold data.

Figure 12. Production trace results. Memory reduction (left axis): *OBASE* Hinted vs. no-reclaim, and *OBASE*+TMO vs. TMO-alone. Page utilization improvement (right axis, hatched) from address-space reorganization.

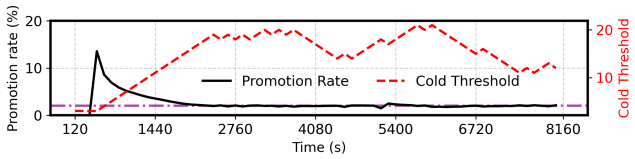


Figure 13. Cold threshold adaptation (Meta CacheLib). Promotion rate (black) and cold threshold C_t (red) over time. The controller automatically adjusts C_t to maintain the 1% target (purple).

Adaptive cold threshold (E4). Figure 13 demonstrates *OBASE*'s ability to adapt to workload dynamics over 2.3 hours of the Meta CacheLib trace. At startup, the initial C_t=3 causes premature demotions, spiking the promotion rate to 14%. The controller increments C_t each scan interval; within 25 minutes, C_t reaches 18 and the promotion rate drops below the 1% target. A high promotion rate during convergence *does not indicate degraded performance*: COLD-heap pages remain in DRAM until a backend explicitly pages them out, and the promotion rate reflects workload behavior rather than backend decisions (§3.4).

Around t=5400 s, a workload shift causes a brief spike, as previously cold keys become active; the controller raises C_t and the system re-converges as the new hotset stabilizes. Throughout the trace, C_t varies between 10–20 windows while maintaining the promotion rate near the target, demonstrating that *OBASE* tracks workload evolution.

Takeaway #5: *OBASE delivers consistent memory savings across real world workloads with diverse characteristics and dynamically adapts to shifting access patterns.*

6 Related Work

Object-Level Management. AIFM [57] and MIRA [35] manage memory at object granularity for disaggregated far memory, bypassing OS page management for specialized user-space runtimes; AIFM builds on Shenango's green threads and kernel-bypass networking, with explicit `DerefScope` guards around far-memory accesses. *OBASE* is complementary: it

acts as a *frontend* that reorganizes the address space for unmodified OS *backends*, optimizing *what* to tier rather than the latency of remote access, so we do not compare directly. The `DerefScope` contrast is instructive: it guards each individual far-memory dereference so the runtime can fault the object in, whereas *OBASE*'s TAG/ATC scopes bracket a whole public operation as a quiescence check that permits migration only when no thread is active, closer to epoch-based reclamation than to per-access load barriers.

Other systems indirect through pointers for different ends. Midas [55] harvests idle memory for application-managed *soft state*, dropping recomputable data under pressure, while Atlas [26] accelerates far-memory applications with a hybrid paging/runtime data plane; both optimize access to *remote* memory, whereas *OBASE* reorganizes *local* resident objects by observed access intensity so page-based backends see uniformly hot or cold pages. Alaska [61] uses handle-based indirection for heap compaction but does not track object hotness or organize memory by access intensity. ObjecTier [53] proposes a non-invasive object-consolidation framework with the same high-level vision as *OBASE*, motivated by simulations and they defer implementation to future work.

Allocation-Time Placement. Static placement approaches [27, 28, 40, 51, 66] make decisions at allocation time based on hints or profiling. However, object hotness is neither knowable at allocation nor stable over time(§2). *OBASE* tracks access patterns at runtime, enabling runtime migration.

Page-Level Tiering. TPP [49], Memtis [44], HawkEye [52], and TMO [62] all operate at page granularity. Even efforts to shrink the tiering granularity (e.g., Memtis's dynamic 4KB classification [44]) are bounded by the page floor, since subpage objects of differing hotness still share a 4KB page (§2). *OBASE* works below this floor by reorganizing at object granularity, and is designed to improve these backends.

Garbage Collection and Object Relocation. GCs have long relocated objects to improve locality: generational collectors cluster recent allocations, profile-guided reordering [39] groups hot objects during copying, and the Bookmarking Collector [38] avoids paging during collection. *OBASE* differs in relocation safety: where ZGC [63] and Shenandoah [30] use load barriers to redirect accesses while threads hold stale pointers, *OBASE* adopts a *quiescence-based* approach inspired by epoch-based reclamation [25, 31] and RCU [50], relocating only when an object's active-thread count reaches zero and aborting on any concurrent access via CAS failure. It also differs in purpose: GCs relocate to reclaim memory or improve cache locality, whereas *OBASE* relocates to improve *page utilization* so that tiering mechanisms see uniform pages.

7 Limitations and Applicability

OBASE targets a specific class of code, pointer-based concurrent data structures in unmanaged languages, rather than aiming for universal applicability. Its requirements bound where it applies, which we make explicit so users can judge fit.

- **No pointer stability.** Relocation invalidates raw pointers cached across operations (e.g., addresses retained by Abseil maps or STL iterators). Callers must re-resolve through the guide, as pointer-unstable containers already require [10].
- **Single ownership.** A guide asserts exclusive ownership like `std::unique_ptr`, and migration updates only that guide. Multiple guides aliasing one object (shared graph or doubly-linked nodes) are unsupported. CrestDB enforces this by deep-copying on insert.
- **No pointer arithmetic.** Guides do not support `+`/`[]` across object boundaries, so contiguous arrays, matrices, and packed columnar data are not supported in *OBASE*; a validation pass rejects such uses.
- **Language support.** The design relies on dereference-operator overloading, which is not supported by Go or Java.
- **Annotation.** Developers mark a small set of pointer fields, so adoption is not fully transparent; the compiler handles the rest (§4.7).
- **Hardware composability.** The guide encoding reuses high-order address bits, which can contend with Intel LAM [2], ARM TBI [9], or HWASAN [18] if those features claim the same bits; wider addressing [5] relaxes this.

We target unmanaged languages, as most latency-sensitive datacenter code is C++ or Rust; the mechanisms generalize to managed runtimes.

8 Conclusion

OBASE shows that memory tiering improves when applications cooperate with the OS rather than bypass it. By reorganizing virtual address space so that page boundaries align with object temperature, *OBASE* makes existing backends more effective without modifying them. The approach has limitations: it requires developer annotation of relocatable pointers and applies only to pointer-based structures. However, the principle of address-space engineering extends beyond tiering. The same techniques could improve generational garbage collection (grouping by access intensity rather than age), reduce false sharing of pages across NUMA nodes (separating objects by access pattern), and strengthen isolation (grouping by trust level). An application's address-space layout is itself a channel to the OS [20]: by grouping objects of similar temperature, *OBASE* turns layout into an object-level signal that unmodified page-based tiering can act on, with no new interface between the two.

Acknowledgments

We thank David Culler, Abhinav Sharma, Lilian Tsai, Qian Ge, Teresa Johnson, Sujay Yadalam, colleagues in SystemsResearch@Google and the ADvanced Systems Laboratory (ADSL), the anonymous reviewers, and our shepherd for valuable feedback. This work was supported by gifts from Google.

References

[1] 2020. CXL And Gen-Z Iron Out A Coherent Interconnect Strategy. https://www.nextplatform.com/2020/04/03/cxl-and-gen-z-iron-out-a-coherentinterconnect-strategy/

[2] 2021. Enable Intel LAM in Linux. https://lwn.net/Articles/902094/

[3] 2022. Compute Express Link. https://www.computeexpresslink.org/.

[4] 2022. Intel Optane DC PMM. https://www.intel.com/content/www/us/en/products/docs/memory-storage/optane-persistent-memory/overview.html

[5] 2022. Road to 128-bit Linux. https://lwn.net/Articles/908026/

[6] 2022. Samsung Memory-semantic ssd. https://news.samsung.com/global/samsung-electronics-unveils-far-reaching-next-generation-memory-solutions-at-flash-memory-summit-2022

[7] 2024. Advanced profiling topics. PEBS and LBR. https://easyperf.net/blog/2018/06/08/Advanced-profiling-topics-PEBS-and-LBR

[8] 2024. DAMON: Data Access Monitor. https://sjp38.github.io/post/damon/

[9] 2024. Top-Byte-Ignore (TBI) ARM. https://en.wikichip.org/wiki/arm/tbi

[10] 2025. Abseil Pointer Instability. https://abseil.io/docs/cpp/guides/container#fn:pointer-stability

[11] 2025. Azure delivers cloud VM with Intel Xeon 6 and CXL memory. https://techcommunity.microsoft.com/blog/sapapplications/azure-delivers-the-first-cloud-vm-with-intel-xeon-6-and-cxl-memory---now-in-priv/4470067

[12] 2025. DynamoRIO: Dynamic Binary Instrumentation Framework. https://dynamorio.org/

[13] 2025. Google Workload Traces Version 2. https://console.cloud.google.com/storage/browser/external-traces-v2.

[14] 2025. LLVM-dev RFC:Sanitizer-based Heap Profiler. https://lists.llvm.org/pipermail/llvm-dev/2020-June/142744.html.

[15] 2025. Overview of Package util.concurrent. https://gee.cs.oswego.edu/dl/classes/EDU/oswego/cs/dl/util/concurrent/intro.html

[16] 2025. Tagged Pointers. https://en.wikipedia.org/wiki/Tagged_pointer.

[17] 2025. VM Linux Kernel Doc. https://docs.kernel.org/admin-guide/sysctl/vm.html.

[18] 2026. Hardware-Assisted AddressSanitizer Design Documentation. https://clang.llvm.org/docs/HardwareAssistedAddressSanitizerDesign.html.

[19] Raja Appuswamy, Goetz Graefe, Renata Borovica-Gajic, and Anastasia Ailamaki. 2019. The five-minute rule 30 years later and its impact on the storage hierarchy. *Commun. ACM* 62, 11 (Oct. 2019), 114–120. https://doi.org/10.1145/3318163

[20] Remzi H. Arpaci-Dusseau and Andrea C. Arpaci-Dusseau. 2023. *Operating Systems: Three Easy Pieces* (1.10 ed.). Arpaci-Dusseau Books.

[21] Vinay Banakar, Suli Yang, Kan Wu, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, and Kimberly Keeton. 2025. Tidying Up the Address Space. In *Proceedings of the 3rd Workshop on Disruptive Memory Systems (DIMES '25)*. 63–72. https://doi.org/10.1145/3764862.3768179

[22] Benjamin Berg, Daniel S. Berger, Sara McAllister, Isaac Grosof, Sathya Gunasekar, Jimmy Lu, Michael Uhlar, Jim Carrig, Nathan Beckmann, Mor Harchol-Balter, and Gregory R. Ganger. 2020. The CacheLib Caching Engine: Design and Experiences at Scale. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. 753–768. https://www.usenix.org/conference/osdi20/presentation/berg

[23] Stephen M. Blackburn, Perry Cheng, and Kathryn S. McKinley. 2004. Myths and realities: the performance impact of garbage collection. *SIGMETRICS Perform. Eval. Rev.* 32, 1 (jun 2004), 25–36. https://doi.org/10.1145/1012888.1005693

[24] Zhichao Cao, Siying Dong, Sagar Vemuri, and David H.C. Du. 2020. Characterizing, Modeling, and Benchmarking RocksDB Key-Value Workloads at Facebook. In *18th USENIX Conference on File and Storage Technologies (FAST 20)*. 209–223. https://www.usenix.org/conference/fast20/presentation/cao-zhichao

[25] Badrish Chandramouli, Guna Prasaad, Donald Kossmann, Justin Levandoski, James Hunter, and Mike Barnett. 2018. FASTER: A Concurrent Key-Value Store with In-Place Updates. In *Proceedings of the 2018 International Conference on Management of Data (SIGMOD '18)*. 275–290. https://doi.org/10.1145/3183713.3196898

[26] Lei Chen, Shi Liu, Chenxi Wang, Haoran Ma, Yifan Qiao, Zhe Wang, Chenggang Wu, Youyou Lu, Xiaobing Feng, Huimin Cui, Shan Lu, and Harry Xu. 2024. A Tale of Two Paths: Toward a Hybrid Data Plane for Efficient Far-Memory Applications. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*. 77–95. https://www.usenix.org/conference/osdi24/presentation/chen-lei

[27] Yu Chen, Ivy B. Peng, Zhen Peng, Xu Liu, and Bin Ren. 2020. ATMem: adaptive data placement in graph applications on heterogeneous memories. In *Proceedings of the 18th ACM/IEEE International Symposium on Code Generation and Optimization (CGO 2020)*. 293–304. https://doi.org/10.1145/3368826.3377922

[28] Subramanya R. Dulloor, Amitabha Roy, Zheguang Zhao, Narayanan Sundaram, Nadathur Satish, Rajesh Sankaran, Jeff Jackson, and Karsten Schwan. 2016. Data tiering in heterogeneous memory systems. In *Proceedings of the Eleventh European Conference on Computer Systems (EuroSys '16)*. Article 15, 16 pages. https://doi.org/10.1145/2901318.2901344

[29] Padmapriya Duraisamy, Wei Xu, Scott Hare, Ravi Rajwar, David Culler, Zhiyi Xu, Jianing Fan, Christopher Kennelly, Bill McCloskey, Danijela Mijailovic, Brian Morris, Chiranjit Mukherjee, Jingliang Ren, Greg Thelen, Paul Turner, Carlos Villavieja, Parthasarathy Ranganathan, and Amin Vahdat. 2023. Towards an Adaptable Systems Architecture for Memory Tiering at Warehouse-Scale. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3 (ASPLOS 2023)*. 727–741. https://doi.org/10.1145/3582016.3582031

[30] Christine H. Flood, Roman Kennke, Andrew Dinn, Andrew Haley, and Roland Westrelin. 2016. Shenandoah: An open-source concurrent compacting garbage collector for OpenJDK. In *Proceedings of the 13th International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools (PPPJ '16)*. Article 13, 9 pages. https://doi.org/10.1145/2972206.2972210

[31] Keir Fraser. 2003. Practical lock-freedom. https://api.semanticscholar.org/CorpusID:11933396

[32] Jim Gray and Goetz Graefe. 1997. The five-minute rule ten years later, and other computer storage rules of thumb. *SIGMOD Rec.* 26, 4 (Dec. 1997), 63–68. https://doi.org/10.1145/271074.271094

[33] Jim Gray and Franco Putzolu. 1987. The 5 minute rule for trading memory for disc accesses and the 10 byte rule for trading memory for CPU time. In *Proceedings of the 1987 ACM SIGMOD International Conference on Management of Data (SIGMOD '87)*. 395–398. https://doi.org/10.1145/38713.38755

[34] Rachid Guerraoui and Vasileios Trigonakis. 2016. Optimistic concurrency with OPTIK. *SIGPLAN Not.* 51, 8, Article 18 (Feb. 2016), 12 pages. https://doi.org/10.1145/3016078.2851146

[35] Zhiyuan Guo, Zijian He, and Yiying Zhang. 2023. Mira: A Program-Behavior-Guided Far Memory System. In *Proceedings of the 29th Symposium on Operating Systems Principles (SOSP '23)*. 692–708. https://doi.org/10.1145/3600006.3613157

[36] Timothy L. Harris. 2001. A Pragmatic Implementation of Non-blocking Linked-Lists. In *Proceedings of the 15th International Conference on Distributed Computing (DISC '01)*. 300–314.

[37] Maurice Herlihy, Yossi Lev, Victor Luchangco, and Nir Shavit. 2007. A simple optimistic skiplist algorithm *(SIROCCO'07)*. 124–138.

[38] Matthew Hertz, Yi Feng, and Emery D. Berger. 2005. Garbage collection without paging. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '05)*. 143–153. https://doi.org/10.1145/1065010.1065028

[39] Xianglong Huang, Stephen M. Blackburn, Kathryn S. McKinley, J Eliot B. Moss, Zhenlin Wang, and Perry Cheng. 2004. The garbage

collection advantage: improving program locality. *SIGPLAN Not.* 39, 10 (Oct. 2004), 69–80. https://doi.org/10.1145/1035292.1028983

[40] Teresa Johnson, Snehasish Kumar, , and David Li. 2021. RFC: IR metadata format for MemProf. https://groups.google.com/g/llvm-dev/c/aWHsdMxKAfE/m/WtEmRqyhAgAJ

[41] Svilen Kanev, Juan Pablo Darago, Kim Hazelwood, Parthasarathy Ranganathan, Tipp Moseley, Gu-Yeon Wei, and David Brooks. 2015. Profiling a warehouse-scale computer. *SIGARCH Comput. Archit. News* 43, 3S (June 2015), 158–169. https://doi.org/10.1145/2872887.2750392

[42] Sungjoon Koh, Junhyeok Jang, Changrim Lee, Miryeong Kwon, Jie Zhang, and Myoungsoo Jung. 2019. Faster than Flash: An In-Depth Study of System Challenges for Emerging Ultra-Low Latency SSDs . In *2019 IEEE International Symposium on Workload Characterization (IISWC)*. 216–227. https://doi.org/10.1109/IISWC47752.2019.9042009

[43] H. Andrés Lagar-Cavilla, Junwhan Ahn, Suleiman Souhlal, Neha Agarwal, Radoslaw Burny, Shakeel Butt, Jichuan Chang, Ashwin Chaugule, Nan Deng, Junaid Shahid, Greg Thelen, Kamil Adam Yurtsever, Yu Zhao, and Parthasarathy Ranganathan. 2019. Software-Defined Far Memory in Warehouse-Scale Computers.. In *ASPLOS*, Iris Bahar, Maurice Herlihy, Emmett Witchel, and Alvin R. Lebeck (Eds.). 317–330.

[44] Taehyung Lee, Sumit Kumar Monga, Changwoo Min, and Young Ik Eom. 2023. MEMTIS: Efficient Memory Tiering with Dynamic Page Classification and Page Size Determination. In *Proceedings of the 29th Symposium on Operating Systems Principles (SOSP '23)*. 17–34. https://doi.org/10.1145/3600006.3613167

[45] Viktor Leis, Alfons Kemper, and Thomas Neumann. 2013. The adaptive radix tree: ARTful indexing for main-memory databases. In *2013 IEEE 29th International Conference on Data Engineering (ICDE)*. 38–49. https://doi.org/10.1109/ICDE.2013.6544812

[46] Huaicheng Li, Daniel S. Berger, Lisa Hsu, Daniel Ernst, Pantea Zardoshti, Stanko Novakovic, Monish Shah, Samir Rajadnya, Scott Lee, Ishwar Agarwal, Mark D. Hill, Marcus Fontoura, and Ricardo Bianchini. 2023. Pond: CXL-Based Memory Pooling Systems for Cloud Platforms. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2 (ASPLOS 2023)*. 574–587. https://doi.org/10.1145/3575693.3578835

[47] Yandong Mao, Eddie Kohler, and Robert Tappan Morris. 2012. Cache craftiness for fast multicore key-value storage. In *Proceedings of the 7th ACM European Conference on Computer Systems (EuroSys '12)*. 183–196. https://doi.org/10.1145/2168836.2168855

[48] Hasan Al Maruf and Mosharaf Chowdhury. 2020. Effectively Prefetching Remote Memory with Leap. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*. 843–857. https://www.usenix.org/conference/atc20/presentation/al-maruf

[49] Hasan Al Maruf, Hao Wang, Abhishek Dhanotia, Johannes Weiner, Niket Agarwal, Pallab Bhattacharya, Chris Petersen, Mosharaf Chowdhury, Shobhit Kanaujia, and Prakash Chauhan. 2022. TPP: Transparent Page Placement for CXL-Enabled Tiered Memory. arXiv:arXiv:2206.02878

[50] Paul Mckenney and JOHN SLINGWINE. 1998. Read-copy update: Using execution history to solve concurrency problems. *Parallel and Distributed Computing and Systems* (01 1998).

[51] Svetozar Miucin and Alexandra Fedorova. 2018. Data-driven spatial locality. In *Proceedings of the International Symposium on Memory Systems (MEMSYS '18)*. 243–253. https://doi.org/10.1145/3240302.3240417

[52] Ashish Panwar, Sorav Bansal, and K. Gopinath. 2019. HawkEye: Efficient Fine-grained OS Support for Huge Pages. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '19)*. 347–360. https://doi.org/10.1145/3297858.3304064

[53] Vinita Pawar, Ankit Bhardwaj, and Ryan Stutsman. 2025. ObjecTier: Non-Invasively Boosting Memory Tiering Performance. In *Companion of the 16th ACM/SPEC International Conference on Performance Engineering (ICPE '25)*. 180–186. https://doi.org/10.1145/3680256.3721319

[54] William Pugh. 1990. *Concurrent maintenance of skip lists.* Technical Report. USA.

[55] Yifan Qiao, Zhenyuan Ruan, Haoran Ma, Adam Belay, Miryung Kim, and Harry Xu. 2024. Harvesting Idle Memory for Application-managed Soft State with Midas. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*. 1247–1265. https://www.usenix.org/conference/nsdi24/presentation/qiao

[56] Amanda Raybuck, Tim Stamler, Wei Zhang, Mattan Erez, and Simon Peter. 2021. HeMem: Scalable Tiered Memory Management for Big Data Applications and Real NVM. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles (SOSP '21)*. 392–407. https://doi.org/10.1145/3477132.3483550

[57] Zhenyuan Ruan, Malte Schwarzkopf, Marcos K. Aguilera, and Adam Belay. 2020. AIFM: High-Performance, Application-Integrated Far Memory. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. 315–332. https://www.usenix.org/conference/osdi20/presentation/ruan

[58] Yan Sun, Yifan Yuan, Zeduo Yu, Reese Kuper, Chihun Song, Jinghan Huang, Houxiang Ji, Siddharth Agarwal, Jiaqi Lou, Ipoom Jeong, Ren Wang, Jung Ho Ahn, Tianyin Xu, and Nam Sung Kim. 2023. Demystifying CXL Memory with Genuine CXL-Ready Systems and Devices. In *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO '23)*. 105–121. https://doi.org/10.1145/3613424.3614256

[59] Muhammad Tirmazi, Adam Barker, Nan Deng, Md E. Haque, Zhijing Gene Qin, Steven Hand, Mor Harchol-Balter, and John Wilkes. 2020. Borg: the next generation. In *Proceedings of the Fifteenth European Conference on Computer Systems (EuroSys '20)*. Article 30, 14 pages. https://doi.org/10.1145/3342195.3387517

[60] Rik van Riel and Vinod Chegu. 2014. Automatic NUMA balancing.Red Hat Summit.

[61] Nick Wanninger, Tommy McMichen, Simone Campanoni, and Peter Dinda. 2024. Getting a Handle on Unmanaged Memory. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3 (ASPLOS '24)*. 448–463. https://doi.org/10.1145/3620666.3651326

[62] Johannes Weiner, Niket Agarwal, Dan Schatzberg, Leon Yang, Hao Wang, Blaise Sanouillet, Bikash Sharma, Tejun Heo, Mayank Jain, Chunqiang Tang, and Dimitrios Skarlatos. 2022. TMO: Transparent Memory Offloading in Datacenters. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '22)*. 609–621. https://doi.org/10.1145/3503222.3507731

[63] Albert Mingkun Yang and Tobias Wrigstad. 2022. Deep Dive into ZGC: A Modern Garbage Collector in OpenJDK. *ACM Trans. Program. Lang. Syst.* 44, 4, Article 22 (Sept. 2022), 34 pages. https://doi.org/10.1145/3538532

[64] Jian Yang, Juno Kim, Morteza Hoseinzadeh, Joseph Izraelevitz, and Steve Swanson. 2020. An Empirical Guide to the Behavior and Use of Scalable Persistent Memory. In *18th USENIX Conference on File and Storage Technologies (FAST 20)*. 169–182. https://www.usenix.org/conference/fast20/presentation/yang

[65] Juncheng Yang, Yao Yue, and K. V. Rashmi. 2020. A large scale analysis of hundreds of in-memory cache clusters at Twitter. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. 191–208. https://www.usenix.org/conference/osdi20/presentation/yang

[66] Zhuangzhuang Zhou, Vaibhav Gogte, Nilay Vaish, Chris Kennelly, Patrick Xia, Svilen Kanev, Tipp Moseley, Christina Delimitrou, and Parthasarathy Ranganathan. 2024. Characterizing a Memory Allocator at Warehouse Scale. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3 (ASPLOS '24)*. 192–206. https://doi.org/10.1145/3620666.3651350

MDK: Rethinking the data center memory reclamation problem

Shaurya Patel[1,2] Suli Yang[1] Yawen Wang[1] Kan Wu[3]
Alexandra (Sasha) Fedorova[2,4] Margo Seltzer[2] Kimberly Keeton[1]

[1]*Google* [2]*University of British Columbia* [3]*xAI* [4]*MongoDB*

Abstract

The traditional memory management problem maximizes application performance when constrained by a fixed-size memory. Today's data centers face a different problem: their goal is to maximize the number of jobs on a server without violating performance Service Level Objectives (SLOs). Since a key constraint for placing additional jobs is memory, data center systems proactively reclaim memory from running jobs to create space for new jobs. This difference fundamentally flips the optimization problem that memory management policies need to address.

Designing practical policies requires a set of tools: 1) an optimal policy that provides a bound on what any policy can achieve, 2) metrics to compare policies, and 3) efficient techniques for evaluating potential policies. However, we find that foundational tools from the traditional setting, such as the optimal policy OPT, Miss Ratio Curves (MRCs), and efficient ways to generate MRCs, do not apply in this new setting. The data center setting demands a new set of tools.

We present the Memory Designer's Kit, *MDK*, a framework for designing and evaluating data center memory management policies. *MDK* includes an offline provably optimal policy; *Memory Performance Curves (MPCs)*, which show how memory savings vary when constrained by performance; and an efficient technique that is up to 208× faster than simulation for producing MPCs. We demonstrate *MDK*'s utility by developing three data center policies that improve average memory savings by up to 10% relative to a state-of-the-art policy.

1 Introduction

Memory management, especially policies for effective eviction of pages from caches, has long been a cornerstone of operating systems [4, 8, 9, 15, 19, 32, 33, 35, 47, 48]. These policies focus on managing the memory of a single computer (node) and can be framed as an optimization problem with the goal of maximizing application performance while constrained by the node's limited memory capacity. Page replacement policies, such as LRU, evict pages in reaction to memory pressure (i.e., when the memory is full and new pages need to be allocated). In this setting, application performance is tracked using a *performance proxy*, the cache miss rate, which is defined as the ratio of the number of cache misses to the total number of memory accesses, measured over the application's lifetime. Replacement policies minimize the miss ratio under the assumption that doing so will maximize application performance.

Optimizing memory usage in *data centers* has become increasingly important, due to the high cost of DRAM [14, 21, 34, 38, 45, 63]. In a data center, each server operates as part of a cluster, and jobs can be placed on or migrated between servers. The goal is to run as many jobs as possible on each server, while satisfying application performance targets [5, 6]. Maximizing server utilization allows data center operators to reduce their total cost of ownership (TCO). Data center operators deploy *reclamation policies* that proactively evict cold pages from memory before facing memory pressure, in order to reduce DRAM usage and make space for scheduling additional jobs. Evicted pages are either demoted to a cheaper byte-addressable memory tier over CXL [1], or swapped to disks [66] or to compressed memory [37].

Like traditional reactive *replacement policies*, proactive data center *reclamation policies* track performance using performance proxies; the main differences are how the proxies are measured and the choice of specific metrics. A replacement policy measures the overall miss ratio across an application's lifetime, whereas a reclamation policy measures a performance proxy in discrete time windows. This windowed approach is better aligned with data center application performance SLOs, which are enforced in a statistical manner over time intervals. Reclamation performance proxy metrics are defined at the OS level to provide consistency across the diverse set of data center workloads. Some modern examples of performance proxy metrics used for reclamation include Google's *promotion rate*, defined as the ratio of faulting pages to the number of unique pages accessed within a time interval [37], and Meta's *Pressure Stall Information (PSI)*, defined as the percentage of compute potential that is unproductive

over a time window due to resource stalls [66].

These differences in the data center environment create a new optimization problem that flips the traditional goal and constraint: the *goal* is to maximize memory savings whenever an application can relinquish memory to allow more jobs to be run, and the *constraint* is to maintain target performance proxies over all time windows to comply with application SLOs. We focus on maximizing average memory savings, subject to a target promotion rate; we discuss the broader problem space in §2.

Designing a new policy requires several tools, as shown in Figure 1. Traditional replacement policy development benefits from an established set of tools, developed over decades. The optimal policy OPT [47] is used to understand policy performance headroom, the best any policy can perform. Policies are compared using Miss Ratio Curves (MRCs) [47], which plot the miss ratio as a function of cache size. Numerous algorithms have been developed to efficiently generate MRCs (*e.g.,* [47,57,64]) based on the inclusion property [47], which makes it possible to calculate miss ratios for multiple cache sizes in a single iteration over a trace. Unfortunately, these well-established tools for evaluating replacement policies do not apply to the data center reclamation problem. As we demonstrate in §2, OPT [47], the well-known optimal policy that minimizes miss ratio, violates time-window performance proxy targets, making it non-optimal for the data center reclamation problem. Further, since the data center reclamation problem formulation flips the optimization goal and constraint, and uses a diverse set of metrics [6,37,62,66] other than the miss ratio, MRCs and their generation techniques are not applicable.

We present the **Memory Designer's Kit**, ***MDK***, an offline framework for designing new data center reclamation policies. *MDK* consists of four components: 1) the **Memory Performance Curve (MPC)**: the data center equivalent of conventional MRCs, which plots a memory optimization goal (*e.g.,* average memory savings) as a function of a performance proxy (*e.g.,* promotion rate). 2) the **Optimal Performance Proxy (OPP) policy**: an offline, provably optimal policy for maximizing average memory savings subject to a performance proxy, the promotion rate constraint. It can be generalized to maximizing average memory savings subject to any performance constraint that is computable on a trace of accesses. 3) **theoretical properties**: properties that relate a reclamation policy's performance proxy and memory savings as a function of different policy settings. Analogous to the inclusion property in the traditional setting, these properties, *eviction decisions* and *eviction times*, make it possible to efficiently calculate MPCs. 4) an **efficient MPC generation framework**: a method that lets us efficiently generate MPCs for multiple performance proxy values in a single pass. *MDK*'s MPC generation is linear in time complexity, 12.5× to 208× faster than our simulator depending on the number of page access events in a trace.

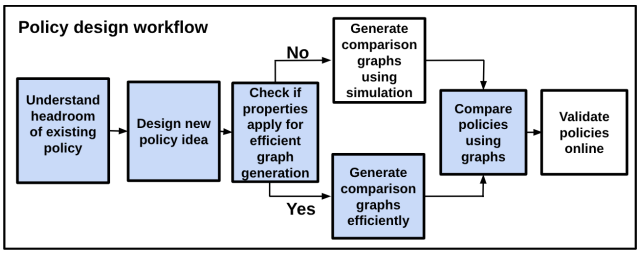


Figure 1: Typical workflow for designing new replacement and reclamation policies. *MDK* contributions are required for multiple reclamation policy development steps (blue squares).

We demonstrate how *MDK* can be used to analyze and develop policies for the data center reclamation problem. First, we examine the age-based reclamation policy (AGE, an approximation of the working set policy [17]) used in g-swap [37] and evaluate its effectiveness in the data center setting. We compare it to *MDK*'s optimal policy, OPP, using MPCs and show that a significant gap exists between the two, indicating substantial room for improvement.

Second, we use *MDK* to develop three policies. Two are history-based heuristics: the single-parameter **P**rior **A**ge with **W**ait (PAW) and the two-parameter **P**rior **A**ge and **C**urrent **E**lapsed (PACE). The third is a learned policy that imitates OPP. Our heuristic policies reclaim a page as soon as it's accessed rather than waiting for the page to cool off (as done in AGE). Using *MDK*, we analyze the performance of these policies, compared to AGE, on offline traces. PAW, PACE, and AGE have tunable parameters dependent on trace characteristics. We measure their maximum potential performance achieved by tuning their parameters using complete evaluation trace information. PAW generates slightly better average memory savings than AGE at lower promotion rates for workloads that exhibit repeating access patterns (four out of eight benchmarks). We validate PAW's performance by implementing it on Linux and confirm that PAW achieves 4% higher memory savings than AGE for a graph processing workload without any performance loss. However, for workloads without any repeating patterns, AGE performs significantly better. Given a representative trace, our second policy PACE performs as well as AGE for any workload by design, and improves performance by 10% for a subset of workloads. Evaluating policy tuning in *MDK* with training and testing phases, or tuning the policies online, are valuable topics of future research.

MDK is generalizable beyond the problem we study in this paper. We selected two metrics out of a set of many possible metrics used in data centers [20,39,66,75]. *MDK* is designed to support different memory saving targets and performance proxies, and we describe the additional work required to apply *MDK* to an even broader context. The rest of the paper is organized as follows: First, we discuss the requirements for data center memory reclamation and motivate the new problem formulation (§2). Then we introduce *MDK*, a toolkit for evaluating policies under the new problem formulation (§3)

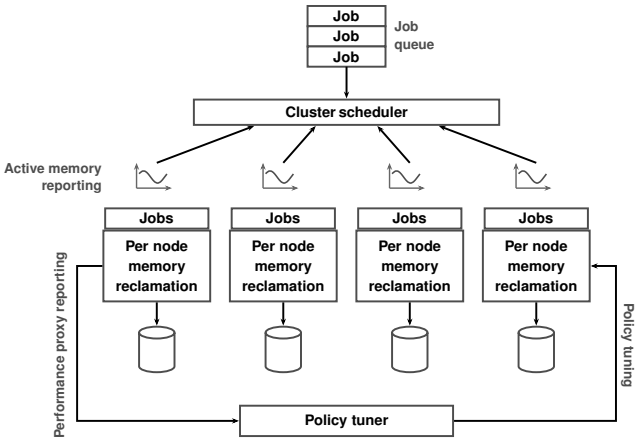


Figure 2: Data center system architecture. Reclamation policies run on individual nodes and proactively reclaim application memory and swap it to disk or compressed memory.

and discuss our implementation (§4). We outline our evaluation methodology and demonstrate the design workflow for developing new policies using *MDK* (§5). Finally, we review related work (§6) and conclude (§7).

2 Data center memory reclamation

We begin by describing the architecture of existing data center reclamation systems, the selection of memory savings targets, and performance constraint metrics in existing systems. We then identify the requirements and intuition for what constitutes a good metric. Finally, we motivate why designing new reclamation policies demands new tools, illustrating this with specific metrics. Because the space of reasonable metrics is vast, we focus our discussion on the metrics used throughout the rest of the paper, mentioning others as areas for further research.

2.1 Data center reclamation architecture

Data center operators use memory reclamation policies to reduce applications' memory usage, so they can run more jobs on servers, thus maximizing server utilization and minimizing TCO [37, 39, 66]. Figure 2 shows a typical data center architecture, which includes a complex ecosystem of feedback loops that operate across multiple timescales and influence the design of memory reclamation. Cluster schedulers manage and place jobs on nodes [62] and periodically track resource usage to predict future demands and place new jobs [6, 54].

Memory reclamation policies, operating on individual nodes, decide *when* to reclaim pages, *which pages* to reclaim, and *how many* to reclaim. They proactively free up DRAM by moving cold application data to slower and cheaper storage, such as compressed memory [37], and SSDs [66]. Recent systems [20, 39, 43, 75] also use byte-addressable CXL [1] memory pools to store cold data.

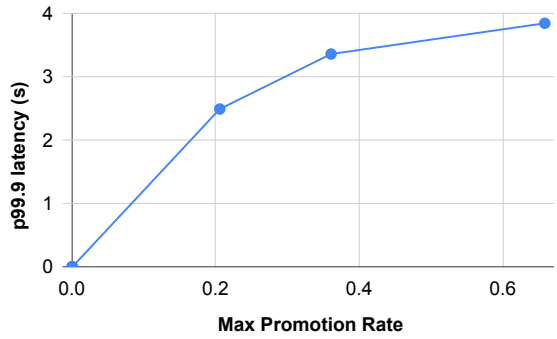


Figure 3: Promotion rate versus tail latency for Cassandra running YCSB. Promotion rate increases as tail latency across windows increases.

Reclamation policies use a *proactive operating mode* in which they reclaim memory alongside application execution, using a *tuner* that determines how many pages to reclaim. The tuner determines the number of pages to reclaim based on the observed performance degradation of the application. It measures performance degradation in discrete time windows over seconds or minutes, which allows the system to maximize memory savings while satisfying application SLOs. Applications evaluate their SLOs (e.g., query rate or tail latency) over discrete periods, not over their entire execution; therefore, satisfying application SLOs requires that a system promptly measure, report, and respond to degradation. For instance, TMO's Senpai [66] measures the performance for ten-second windows, while g-swap [37] measures two-minute windows. If performance suffers, the system reclaims memory less aggressively; if performance is acceptable, it reclaims memory more aggressively.

A policy's *control parameter* determines if a page should be reclaimed. For example, g-swap [37] uses page age, reclaiming every page that is unused after a time T, while TMO uses a page's position in LRU ordering to determine which pages to reclaim.

Although the reclamation system design dictates the overall optimization problem, system designers must still decide meaningful ways to measure performance degradation and memory savings.

2.2 SLO-aligned performance proxy metrics

Systems track application performance degradation using *performance proxies* that the operating system (not the application) can easily evaluate. This is analogous to what systems have historically done, i.e., identify the page to evict by minimizing the number of page faults. In either setting, a performance proxy should: 1) be easy for the OS to monitor (despite hardware heterogeneity), 2) be meaningful for the diverse application mix hosted in the data center, and 3) correlate with application performance metrics commonly used in data centers.

Traditional performance proxy metrics such as cache miss rate and page fault rate are not a good match for data centers. Cache miss rate (defined as total misses divided by total accesses) cannot be accurately measured in the data center. Miss rate can be computed in the OS as the number of page faults divided by the number of page accesses, which requires observing every page access, but typical OS telemetry doesn't provide exact counts of repeated accesses to pages already in memory. Page fault rate (the total number of page faults normalized by time) is used by earlier work [15, 55], since it can be easily measured by the OS. However, as shown by TMO [66], page fault rate isn't a good data center metric: it doesn't account for hardware heterogeneity, and it fails to account for the total number of application memory accesses in a time interval.

Because traditional proxies lack applicability, reclamation system designers have developed their own. g-swap [37] uses the *promotion rate per time window*, which Google's deployment defines as the number of non-compulsory page faults normalized by the number of unique pages accessed in that window [37]. g-swap monitors this metric every two minutes and ensures it does not exceed a threshold [37]. This metric proves especially effective if page faults have a relatively uniform cost; we found that it tracks application performance well. Figure 3 illustrates that as Cassandra's (described in §5) tail latency increases, it also leads to a higher promotion rate. Additionally, checking page table entry (PTE) access bits reliably measures the number of unique pages accessed across different hardware platforms without introducing CPU caching or security concerns.

TMO uses *Pressure Stall Information* (PSI) [66]. PSI measures the time tasks spend waiting for page faults, thus measuring the impact of each page fault. Page faults have different performance costs on hardware with different performance characteristics (e.g., compressed memory or SSD); promotion rate might not capture this difference. PSI addresses this memory tier heterogeneity and provides a more direct indication of performance degradation due to memory contention; however, because PSI is a function of system load, it is harder to use in offline settings over memory access traces.

Recent byte-addressable systems use still other proxies [20, 46] such as *Secondary Tier Access Ratio* (STAR). STAR calculates the total accesses directed to a secondary tier (e.g., CXL memory) relative to the total number of accesses. STAR is a good match for byte-addressable memory because data can be accessed directly without promotion. Ultimately, the system architecture and hardware heterogeneity determine the proxy choice.

2.3 Memory savings optimization targets

Optimizing for a *memory savings target* should enable running more jobs on a node, which decouples memory reclamation and cluster scheduling. Higher level feedback loops guide the design of the memory savings target: a cluster scheduler can leverage memory savings only if those savings last long enough to allow newly scheduled applications to amortize the overhead of job placement. For example, Google data centers use a threshold of at least five minutes [54].

In addition to increasing overall efficiency, optimization targets might encode other goals such as minimizing page migration traffic between memory tiers. For example, if a page is reclaimed and migrated to a slower memory tier and then accessed again a short time later, it produces unnecessary page migrations, which harm application performance.

An effective metric must therefore lead to meaningful savings that increase utilization, while satisfying other goals such as keeping page migration activity low. TMO and g-swap maximize the average memory savings of an application over time. Another metric worth exploring is *tail memory usage*, which measures the $95^{th}\%ile$ (or higher) memory usage of a node (i.e., machine memory capacity minus memory savings) and can serve as an effective indicator of future peak memory usage [6], which is helpful for capacity planning [12].

2.4 The data center reclamation problem

The data center reclamation problem space offers many possible optimization problems, defined by the choice of memory savings metric and performance proxy. Having determined the desirable characteristics of these metrics, we focus on one specific problem formulation to understand how to design reclamation policies and to demonstrate the use of *MDK*.

We adopt average memory savings as the memory savings target, as is currently done in several existing systems [37, 66]. We adopt g-swap's promotion rate per time window as the performance proxy. We choose promotion rate because it's easier to compute on offline traces than PSI, making it a practical choice for understanding how to design reclamation policies.

The specific problem we explore is: **maximize average memory savings subject to the constraint that the promotion rate (measured every time window T_{proxy}) does not exceed a specified threshold (the target promotion rate)**. Here, T_{proxy} defines the time scale for monitoring and reacting to performance violations.

2.5 The need for a new design toolkit

Unfortunately, foundational tools designed for replacement policy design workflow shown in Figure 1, don't translate directly to the reclamation problem.

First, policies deemed optimal for the traditional problem are suboptimal for data center reclamation. Consider Mattson's OPT [47], the optimal policy for minimizing miss ratio for a cache of a fixed size M, and VMIN [50], the optimal policy for minimizing miss ratio in a variably-sized cache of average size M. OPT evicts the resident page that will be

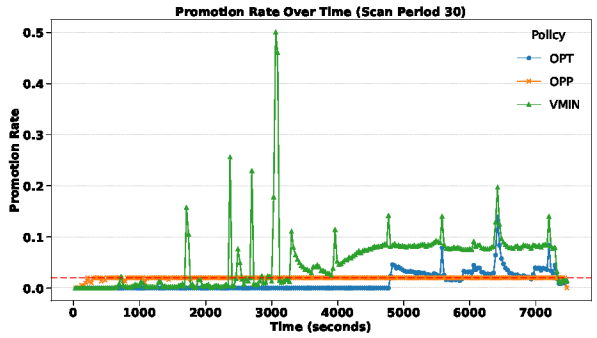


Figure 4: Promotion rate over time for a Cassandra server running YCSB with 50% read and 50% update operations. The red line depicts the target promotion rate. Both OPT and VMIN violate the target promotion rate.

accessed farthest into the future. VMIN proactively reclaims a page immediately after access, if its next access is greater than a given preset parameter T_F. Figure 4 shows how the promotion rate varies over time for a Cassandra workload (described in §5), with the target promotion rate set to 2% (shown as a horizontal red line). We see that both OPT and VMIN violate the target promotion rate when measured over discrete time intervals, while Optimal Performance Proxy (OPP, the optimal policy we define in §3.2) satisfies this constraint throughout.

Mattson's MRC construction algorithm [47] allows researchers to compare policies in the traditional setting. It relies on a theoretical property called the *the inclusion property* that states that a cache of size $C_1 > C_2$, implies C_1 always contains all elements contained in C_2. Therefore, if a memory access causes a miss in C_1, it also causes a miss in C_2. Researchers define the largest cache size that experiences a miss as the critical capacity C_t; caches larger than C_t will not experience the miss. Mattson's algorithm uses this insight to compute the critical cache capacity for each access in a trace. This allows the algorithm to compute Miss Ratio Curves (MRCs) in log-linear time complexity. This approach falls short for reclamation policies, because they do not use the same control parameters, *i.e.,* cache size C. Additionally, the memory size used by the application is implicit for replacement policies, but not for reclamation policies, where it changes as we reclaim memory over time, requiring us to compute the memory size alongside the performance proxy. Clearly, new approaches are needed for developing practical reclamation policies.

3 *MDK*: a toolkit for the new problem

We present the **Memory Designer's Kit** (*MDK*), the data center memory reclamation analog to the policy development tools and techniques used for traditional cache replacement. *MDK* consists of four components: 1) the *Memory Performance Curve (MPC)* (§3.1): the data center equivalent of conventional MRCs to illustrate policy tradeoffs. 2) the *Optimal Performance Proxy (OPP) policy* (§3.2): an offline, provably optimal policy for maximizing average memory savings subject to a promotion rate constraint. 3) *theoretical properties* (§3.3): properties that relate a reclamation policy's promotion rate and memory savings as a function of different policy settings. 4) an *efficient MPC generation framework* (§ 3.4): a method that leverages the theoretical properties to efficiently generate MPCs for multiple performance proxy values in a single pass. We also discuss *MDK*'s generality (§3.5).

3.1 Memory Performance Curves (MPCs)

Memory Performance Curves (MPCs), shown in Figure 7 and Figure 8, plot how the average memory savings (on the y-axis) changes as a function of the target promotion rate (on the x-axis). Since both metrics are dependent values driven by the policy control parameter, R_t, (described in §3.3), their relationship is not surjective, meaning that the achievable memory savings do not map to the entire spectrum of performance values. We present MPCs as scatter plots because drawing lines between points would falsely imply that these intermediate performance proxy values can be achieved when, in fact, they cannot. Just like MRCs, MPCs help us understand how to trade performance for memory savings under a given policy and answer questions such as: if we can tolerate a certain level of performance degradation (e.g., a 2% promotion rate instead of 1%), what additional memory savings can we achieve? MPCs also provide a consistent way to compare the performance of different policies: under the same performance constraint (e.g., a 1% target promotion rate), which policy achieves greater memory savings? §5 demonstrates using MPCs for these purposes.

3.2 Optimal Performance Proxy (OPP) policy

A provably optimal policy is an important tool for developing practical policies for a given problem, even if it requires future knowledge and is impractical to implement. It allows system designers to compare policies to the optimal to determine how much performance might be gained by designing a better practical policy. It often inspires more practical policies; for example, S3FIFO [2] used OPT to conclude that, at smaller cache sizes, most objects are accessed only once. For learning-based policies, it also serves as ground truth for training [29, 40, 42, 44, 52, 56, 58–60, 65, 68, 71, 76].

We present an optimal policy for the data center reclamation problem. We begin by illustrating the intuition behind an optimal policy for this setting (§3.2.1) and then formally present the Optimal Performance Proxy (OPP) policy and discuss key insights (§3.2.2).

T	Pages accessed	Pages cached			Pages evicted		
		OPT	VMIN	OPP	OPT	VMIN	OPP
1	a,b,c	a,b,c	a,b,c	a,b,c		a,b	a,c
2	d,e,f	d,e,f	c,d,e,f	b,d,e,f	a,b,c	e	e
3	d	d,e,f	c,d,f	b,d,f		d	d
4	c,f	c,e,f	c,f	b,c,f	d	c,f	c,f
5	a,b	a,b,c	a,b	a,b			

Table 1: Non-optimality of OPT and VMIN. Both OPT (cache size 3) and VMIN (forward reuse distance 3) evict pages that violate future promotion rates. Page promotions for previously accessed pages, i.e., *page faults*, appear in red. A violation of the target promotion rate occurs in a period where the ratio of page faults to accesses is greater than 50%.

3.2.1 Non-optimality: an example

To understand the key requirements of an optimal policy for data center reclamation, it's helpful to first examine why traditional optimal replacement policies such as OPT and VMIN fail. Table 1 provides an example: for a given access trace, it shows accessed, cached, and evicted[1] pages for OPT, VMIN, and OPP and highlights (in red) violations of the target promotion rate (set at 50%). A violation occurs in a time period when the promotion rate (the number of page faults for previously accessed pages relative to the total number of pages accessed) is greater than the target.

All three policies begin by caching a, b, and c. During time $T2$, OPT for a cache of size 3 evicts a, b, and c, to make room for d, e, and f. No eviction happens during $T3$ as the only accessed page (d) is already cached. During $T4$, OPT evicts d, which will never be accessed again, to accommodate c. Finally, during $T5$, accesses to both a and b produce cache misses, leading to a promotion rate of 100%, which exceeds the 50% target and constitutes a violation.

VMIN with a forward reuse distance of 3 (i.e., pages not accessed during the next 3 windows should be reclaimed) has the same initial cache state as OPT. However, at the end of $T1$, VMIN retains c, which is used again in $T4$ for a reuse distance of only 2, but evicts a and b proactively, because they are not used again until $T5$ for a reuse distance of 3. VMIN's cache state at the end of $T2$ contains four pages, as VMIN does not impose a fixed cache size. During $T3$, VMIN retains c and f, but evicts e as it will never be accessed again. During $T4$, only d is evicted, because it is never accessed again. However, during $T5$, accesses to both a and b cause misses, resulting in a 100% promotion rate, violating the constraint.

As we can see, both OPT and VMIN perform reclamation without considering *the future promotion rate*. They may be optimal in terms of the *total* promotions, but not the *rate in time windows*. As a result, they may cause promotions to be clustered in a single time window rather than spread out, which may violate SLOs.

Key insight: An optimal policy whose goal is to maintain a target promotion *rate* should treat *the number of promotion events relative to the number of unique elements accessed in a given time window* as a constraint during reclamation.

[1]We use the terms *reclamation* and *eviction* interchangeably, but prefer *eviction* when contrasting with prior work.

3.2.2 Optimal Performance Proxy (OPP) policy

OPP is a two-pass algorithm that takes a trace of page-level accesses and produces a set of page reclamation decisions for each time window, defined by a parameter *window_size*. During the first pass, it determines the set of pages accessed in each time window. During the second pass, at the end of each window, OPP reclaims all pages, p, that will not generate a page fault that violates the target promotion rate in the future window in which they are accessed.

More formally, given a trace of accesses A at times T to a set of all uniformly-sized pages P, $A \subseteq T \times P$, where each access is represented as a tuple $\langle t, p \rangle$ with $t \in T$ and $p \in P$. Given $W = \{1, 2, \ldots, N_w\}$ a set of measurement windows, each of a uniform time duration D, OPP maintains $U = \{U_w \mid w \in W\}$, the count of (unique) pages accessed in each window w, calculated during the first pass of the trace, and $F = \{F_w \mid w \in W\}$, a set of F_w, the running count of promotions (page faults) in each window w. $\forall \langle t, p \rangle \in A$, OPP will reclaim p, accessed at time t with known future access t_b in window B, where the window index B is computed as t_b/D, as long as the promotion rate generated by p's next access will not exceed the target promotion rate (i.e., $F_B/U_B \leq$ target promotion rate). If p is reclaimed, then the number of future promotions (F_B), is incremented.

Let us now return to the example in Table 1. Starting with the same initial state as VMIN and OPT, OPP reclaims a at the end of interval $T1$, but not b (unlike VMIN, which reclaims both), as both a and b are accessed during $T5$, and reclaiming them both will violate the target promotion rate of 50%. For the same reason, OPP avoids reclaiming b in future intervals, but OPP reclaims c at $T1$ as c's next access in $T4$ does not violate the target promotion rate.

Key insight: *OPP reclaims page as soon as possible, such that the future accesses of reclaimed pages are spread across time windows instead of clustered together. This approach allows it to meet the performance constraint for promotion rate while maximizing average memory savings.* We present the proof of OPP's optimality in Appendix C.

3.3 *MDK* theoretical properties

We next explore the properties of reclamation policies, with the goal of identifying properties analogous to the inclusion property in the traditional setting that, when present, will enable efficient MPC production.

Consider policies with one control parameter, R_t. The tuple $\langle e, T \rangle$ is an *eviction tuple* indicating that page e is evicted at time T. Let $S_{R_t} = (\langle e_1, T_1 \rangle, \langle e_2, T_2 \rangle, \ldots \langle e_n, T_n \rangle)$ be the sequence of all eviction tuples for a given policy

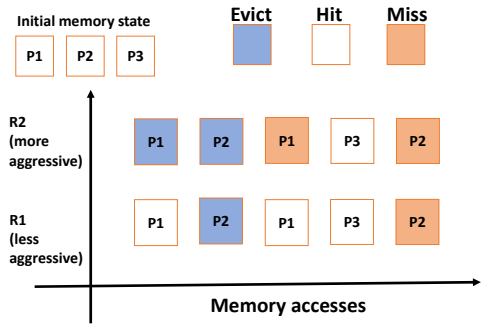


Figure 5: Decisions of a hypothetical reclamation policy that follows both *eviction decisions* and *eviction times* properties. Blue blocks depict a page reclamation after that page is accessed. The more aggressive control parameter, R2, always evicts everything that R1 evicts, at the same time.

configured with parameter R_t. The eviction sequence $E_{R_t} = (e_1, e_2, \ldots, e_n)$ is an ordered sequence of *evictions* extracted from S_{R_t} without timestamps.

Definition 1: A policy configured with parameter R_2 is *more aggressive* than R_1 if $|E_{R2}| \geq |E_{R1}|$, i.e., a policy with a larger number of evictions is more aggressive.

Definition 2: The *critical parameter,* R_C is the least aggressive parameter that causes an eviction, i.e., if R_C is the critical parameter for an eviction e, then e is an element of E_{R_C}, and for all configurations R_x: $e \in E_{R_x} \iff |E_{R_x}| \geq |E_{R_C}|$
This is analogous to the inclusion property's critical capacity, where R_C is a critical parameter if it is the least aggressive parameter for which an eviction happens.

Property: ***Eviction decisions***. *If* $|E_{R2}| \geq |E_{R1}|$ *then* $E_{R1} \preceq E_{R2}$, *i.e.,* E_{R1} is a subsequence of E_{R2}.

The *eviction decision* property states that if parameter R_2 is more aggressive than parameter R_1 for a policy, then every eviction made with R_1 is also made with R_2. Figure 5 illustrates this for a hypothetical policy where R2, which is more aggressive than R1, evicts every page evicted by R1. Note that this property applies to E_{R_t} and not S_{R_t}, which means that it does not impose a restriction that caches contain the same elements at all times T, only that their eviction sequences are ordered subsequences.

Property: ***Eviction times***. *If* $|S_{R2}| \geq |S_{R1}|$ *then* $S_{R1} \preceq S_{R2}$.

The *eviction times* property states that if a policy can be configured with two parameters R_2 and R_1, and R_2 is more aggressive than R_1, then every eviction tuple with R_1 also occurs with R_2, i.e., *evictions happen at the same time*. The policy in Figure 5 illustrates this: for both control parameters R1 and R2, P2 is evicted at the same time. The *eviction times* property can be satisfied only if *eviction decisions* is satisfied.

These properties enable efficient MPC generation. If a policy follows both the properties, then, for any given page fault, we need to track only promotion and savings caused by this fault for the critical parameter, as values for all more aggressive parameters must experience this fault and can be computed using the critical parameters' value.

We can prove that VMIN adheres to the *eviction decisions* and *eviction times* properties. VMIN takes a reuse distance T_F as input. If an element's next-access distance exceeds T_F (determined using future access knowledge), it is reclaimed immediately after access. If a reuse distance parameter T_F induces a page fault for a memory access, all reuse distance parameters smaller than T_F must also induce page faults, as the next-access distance must exceed them as well. Further, since a page is reclaimed immediately after being accessed, all parameters that cause a page to be reclaimed, must do so at the same time. Thus, VMIN follows both properties. We omit proofs for other policies implemented in *MDK*, but they can be derived by construction.

3.4 Efficient MPC generation

The *MDK* MPC generation framework provides the building blocks to efficiently construct MPCs for various reclamation policies that adhere to the *MDK* properties. Efficient MPC generation is important, as simulating policies can take a long time.

```
def generate_mpc(A):
    savings    = [0] * GetParamSpaceSize()
    promotions = [0] * GetParamSpaceSize()

    for t, p in A:
        R_C = GetCriticalParam(t, p)
        promotions[R_C] += 1
        access_savings = GetMemorySavings(t, p)
        savings[R_C] += access_savings

    cumulative_promotions = 0
    cumulative_savings    = 0

    # Iterate through parameters sorted by their aggression
        level order
    param_order = GetParamAggressionOrder()
    for R_t in param_order:
        cumulative_promotions += promotions[R_t]
        cumulative_savings    += AccumulateSavings(savings[
            R_t])

        print(R_t, cumulative_promotions, cumulative_savings
            )
```

Listing 1: Efficient MPC generation template. Users need to manually implement five policy-specific functions.

3.4.1 Understanding the MPC template

Listing 1 presents a high-level approach for constructing MPCs for policies that operate with a single control parameter. It takes as input a trace of page accesses and their access times and requires that the user provide five policy-specific functions. We first explain the core ideas of the approach and

```
GetCriticalParam(page_id, curr_time) {
  return (curr_time - prev_time[page_id])
}
```

```
GetMemorySavings(page_id, curr_time) {
  return (curr_time - prev_time[page_id] ) * page_sz
}
```

```
AccumulateSavings(savings) {
  return savings
}
```

Figure 6: Policy specific function examples for MPC generation of the VMIN policy. *GetCriticalParam* computes the critical parameter, which is the least aggressive future reuse distance for which a page access causes a page fault. *GetMemorySavings* returns the time-multiplied savings associated with each page, assuming it was migrated back into DRAM for the current access. *AccumulateSavings* iteratively computes the memory savings associated with all control parameters of a policy.

then discuss how to use this implementation by detailing the policy-specific functions required for VMIN [50].

The generator leverages the fact that all memory accesses cause a page fault under a sufficiently aggressive control parameter setting. For each access, we compute the critical parameter R_C (line 6), the least aggressive setting where that access causes a page fault. In traditional systems, this is the largest cache size for a miss: any cache size smaller than this critical size will also experience a miss if the policy follows the inclusion property. Similarly, in our context, any parameter that is more aggressive than R_C will also experience a given page fault if the policy follows the *eviction decisions* property. Computing R_C depends on the policy and the control parameter: g-swap uses page age threshold, VMIN uses future reuse distance, etc. Thus, we implement a policy-specific *GetCriticalParam* function.

Page faults create memory savings for the time a page is removed from memory. The key idea for efficient memory-saving computation of all parameters is that the generator tracks savings from reclaiming page P only for the critical parameter R_C (line 9). If the policy follows the *eviction times* property, more aggressive settings will save the same amount, because they will *reclaim the page at the same time as* R_C. We easily calculate savings: if a page is reclaimed at T_1 and faulted in at T_2 (for the current access), the saving is $(T_2 - T_1) \times page_size$. The policy-specific function *GetMemorySavings* determines T_1, as different policies reclaim pages at different times. For example, g-swap reclaims once a duration of age has elapsed since the page was last accessed, while VMIN reclaims immediately after a page is accessed.

The algorithm's last step builds the full MPC. It recursively calculates promotion rate and average memory savings for all possible policy parameters. For the sake of exposition, we simplify promotion rate calculation here, but our algorithm computes it per time window. First, we determine the order of parameter iteration from least aggressive to most aggressive. The order of aggressiveness is also policy specific, e.g., largest age to smallest for g-swap, smallest to largest promotion rate for OPP. Assuming policies follow *eviction decisions*, we cumulatively add page faults to more aggressive settings (line 17) to get the promotion rate for all parameters. *AccumulateSavings* recursively computes average memory savings. For policies following *eviction times*, we add savings from less aggressive parameters to more aggressive ones. This generates the entire MPC.

3.4.2 Using the generator with policy specific functions

We used *MDK* to implement several policies; we include the policy-specific functions for these policies in Appendix A. Here we illustrate how to use *MDK*'s MPC generator using VMIN [50] in Figure 6. Recall that VMIN reclaims any page whose next access occurs more than than T_F time period in the future.

GetParamSpace and *GetParamAggressionOrder* (omitted from Figure 6) define the template's invariant for building the MPC from least to most aggressive parameters. The least aggressive parameter for VMIN is the largest reuse distance in a trace, and the most aggressive value is 1, so we order these parameters from high to low.

GetCriticalParam takes the page ID and access time of the page access and calculates the critical parameter R_C. For VMIN, R_C is simply the current page's reuse distance, i.e., the largest possible T_F setting for which this page would have been reclaimed by the policy when the page was previously accessed. Because VMIN reclaims on access, the currently accessed page would cause a page fault only if VMIN immediately reclaimed this page after its last access. Suppose the current time is T_2 and the previous time the page was accessed was T_1. At T_1, the policy finds the next access, which occurs at T_2, and reclaims the page for any $T_F < (T_2 - T_1)$, with R_C being $T_2 - T_1$. In the generator, we can directly compute R_C at T_2 by looking at the last time the page was accessed. Therefore, we do not need to maintain future data. As shown in Figure 6, we compute this by subtracting the last access time from the current one, T_C. Because VMIN follows *eviction decisions*, any $T_F < T_C$ will also experience this page fault, and the template counts faults for all $T < T_C$ to construct the MPC.

GetMemorySavings takes the same input as *GetCriticalParam* and calculates the memory saved by a reclamation, when a promotion happens. The value will be determined by the time between when a page is reclaimed and the time of its next access. In VMIN, pages are reclaimed immediately after access, so the savings are the number of windows between a page's reclaimed access and its next access.

AccumulateSavings adds up memory savings for all settings as the MPC is constructed. For policies such as VMIN that follow *eviction times*, we simply add the savings of less aggressive settings to more aggressive ones as we go through the settings in the template, as shown in Figure 6. If a policy does not follow *eviction times* (e.g., the AGE policy), this

function lets designers create custom recursive calculations.

3.5 Generality of *MDK*

We present *MDK* in the context of one specific problem, characterized by the metrics *average memory savings* and *promotion rate*. However, as discussed in §2, there is a large problem space for data center memory reclamation, with different choices of memory savings targets and SLO-aligned performance proxies. The tools provided by *MDK* generalize well to this broader problem space, enabling researchers to explore beyond the specific problem presented here: (1) An MPC, by definition, accommodates any memory-saving metric and performance proxy, making it applicable to any problem in the space. (2) The optimal policy will vary by problem formulation; however, the development of OPP provides key insights into designing optimal policies for other problems: the performance constraint should always be treated as a first-class consideration, regardless of which proxy defines the constraint. (3) The *eviction decisions* and *eviction times* properties allow us to categorize policies with one control parameter, R_t, based on whether they satisfy these properties, regardless of the specific problem they address. (4) The MPC generation template works with any single parameter policy that satisfies the *eviction decisions* property, regardless of the specific problem formulation. With the PACE generator (Appendix B), we demonstrate that it is possible to apply our properties to two-parameter policies as well; generalizing them to multiple parameters is future work. (5) We focus on metrics that can be computed using page access traces. Runtime metrics such as PSI require an analytical model that can approximate PSI values offline; developing and incorporating such a model in *MDK* is future work.

4 Implementation

MDK consists of a collection of C++ tools and libraries to simulate data-center memory reclamation policies and generate MPCs efficiently.

Trace collection and format: We collect page access traces on Linux using a kernel thread similar to *kstaled* from g-swap [37]. Applications run with all their data in DRAM and hugepages disabled, and the kernel thread periodically (every 30 seconds in our experiments) walks the page table, writes a trace record for each page accessed, and clears the access bits. The trace record contains the page ID and an access period counter value. The performance metric that we use in the paper, promotion rate, is evaluated on a per-application basis as described in g-swap; we focus on collecting per-application traces instead of traces for applications competing against each other. This periodic algorithm does not capture multiple accesses to a page in the same period. This omission is problematic for frequency-focused policies or performance metrics. However promotion rate is resilient to this omission as long as a page is not faulted multiple times within a time window (which is unlikely).

Policy simulation: Simulation provides ground truth for evaluating the accuracy of MPCs generated using *MDK*, but existing simulators do not implement variable-sized-memory policies, so we implemented our own, similar to libcachesim [72]. Our simulation framework takes as input the traces described in the previous section and simulates both fixed- and variable-sized-memory eviction algorithms. At the end of each time period, the simulator selects pages to reclaim according to the policy.

We simulate eight different policies: two fixed-sized-memory policies (LRU and OPT) and six variable-sized-memory policies (VMIN, AGE, PAW, PACE, L-OPP, and OPP). LRU was faithfully implemented based on prior work, and OPT follows the algorithm described by Mattson et al. [47]. VMIN was implemented as described by Prieve et al. [50], and our AGE policy follows the description from Lagar-Cavilla et al. [37]. Finally, we implemented new policies PAW, PACE, and L-OPP that are described in §5.4. We implemented OPP as described in §3.2.2.

Efficient MPC construction: Our MPC construction library takes as input a trace and a policy and outputs a collection of performance proxy, memory saving pairs that together express an MPC. We implemented MPC generators for AGE, OPP, VMIN, PAW, and PACE as case studies. Single-parameter policies were implemented in *MDK* in fewer than 87 lines of code. All single-parameter policies follow the *eviction decisions property*, and all but the AGE policy follow the *eviction times property*. Policies that follow the *eviction times* property can use the same AccumulateSavings function. However, the GetCriticalParam and GetMemorySavings functions require policy-specific data structures and computation, such as the one described for VMIN in §3.4.2. The framework supports only integer control parameters, so we currently generate only integer promotion rates for OPP. For PACE, our new two-parameter policy (§5.4), we implemented a specialized MPC generator described in Appendix B.

5 Evaluation

We evaluate the individual components of *MDK* and demonstrate *MDK*'s use in Figure 1's policy design workflow. To evaluate *MDK*'s components, we compare the accuracy and speed of its MPC generation relative to simulation and compare its OPP policy to VMIN. To illustrate *MDK*'s use in designing new policies, we show how it can be used to understand the headroom of existing policies, inspire the creation of new policy ideas, and efficiently generate MPCs to compare policy alternatives.

Table 2: Workloads used to evaluate *MDK*. Execution time (in seconds) for *MDK*'s MPC generator and simulation for the OPP policy, which is the slowest. *MDK* generates the plot for all parameters; for simulation, we conduct separate simulations for 10 control parameters to populate the MPC.

System	Workload	Memory	Trace duration	*MDK* (s)	Simulation (s)	Speedup
Cassandra	YCSB	17.48G	120 minutes	88.4	9752	110×
Memcached	Twitter KV	15.06G	120 minutes	432.4	24165	55×
GraphX PageRank	PageRank on Twitter dataset	10.2G	28 minutes	36.8	2253	61×
NGINX Media streaming	Video serving	820M	11 minutes	0.4	5	12.5×
TaoBench	Caching	160G	45 minutes	583.2	48240	82×
DjangoBench	Web serving	1.56G	51 minutes	6.7	160	24×
FeedSim	Object aggregation/ranking	56G	22.5 minutes	52.9	11050	208×
MediaWiki	Web serving	3.9G	15.5 minutes	8.2	260	32×

5.1 Methodology

We evaluate *MDK* using page access traces generated from eight workloads in the CloudSuite [49] and DCPerf [61] benchmark suites, as listed in Table 2. All workloads were configured with the default parameters specified in the CloudSuite Docker images and DCPerf documentation. We collected all our page traces and ran our end-to-end case study on Linux kernel version 5.10 on a machine with an Intel Xeon CPU E5-2696 with 256 GB of RAM. We used a 64-core AMD EPYC 7B13 machine with 128GB of RAM for benchmarking *MDK*'s simulator and MPC generator.

5.2 *MDK* MPC generation

We evaluate the accuracy and speedup of *MDK*'s MPC generation relative to that of simulation for the workloads in Table 2.

MPC generation accuracy: *MDK* includes MPC generation algorithms for policies enabled by our theoretical properties. To validate the accuracy of *MDK*'s MPC generation, we compare *MDK* MPCs to MPCs created via simulation. We confirm that the simulator and MPC produce similar average memory savings and promotion rates for all policies. For one-parameter policies, we compared ten different parameter settings, and for the two-parameter policy PACE, we compared 15 different parameter settings. The mean absolute errors of all MPCs are within 1% of the simulation (due to rounding inconsistencies).

MPC generation time: Our goal in pursuing fast MPC generation is to enable rapid evaluation of new policies and/or parameter settings for those policies, rather than relying on simulation or in-kernel implementations. Table 2 shows the runtime for generating MPCs using *MDK* versus simulation for the OPP policy, the slowest policy in both the *MDK* framework and our simulator. OPP MPC construction takes twice the time of both VMIN and AGE, as OPP needs to process the trace twice, first to gather the number of pages accessed in each window and second to compute reclamation decisions. The time that we report for simulation is for simulating ten different parameter settings, run sequentially. The simulation time is dependent on the number of page access events in each trace. Memcached and TaoBench have the highest number of page access events, leading to the slowest simulation time. *MDK*'s MPC generation algorithms outperform simulation by 12.5× to 208×. *MDK*'s MPC generation is faster due to the linear time complexity of the MPC generation algorithms compared to the quadratic time complexity of simulation. The simulation time can be reduced by running simulations in parallel; however, the simulator would still be limited by its time complexity.

5.3 Comparing optimal policies

Figure 4 shows the promotion rate over time for Cassandra. Recall that VMIN and OPT violate the promotion rate. OPP maintains a target promotion rate for the duration of the entire workload, due to it considering both the accesses per time interval and the reclamations. Although VMIN and OPT maximize system throughput and can lead to better average memory savings when the performance proxy is measured over the entire execution of a program [18], they make suboptimal decisions that lead to promotion rate violations in time windows, likely leading to application SLO violations.

5.4 Using *MDK*

We show how we use the policy development workflow (Figure 1) for new policies. First, we use *MDK* to understand AGE's performance compared to OPP. Then, leveraging insights from OPP, we describe how we design two heuristic policies and a learned policy. We then describe the generation of their MPCs and discuss our policies' performance compared to AGE. Finally, we validate one of our heuristic policies on Linux.

5.4.1 Understanding headroom:

Figure 7 shows the average memory savings as a function of the promotion rate for our workloads. We expect that memory savings should increase as the promotion rate constraint increases (i.e., policy becomes more aggressive). MPCs tell us what parameters of a policy perform best under an operating

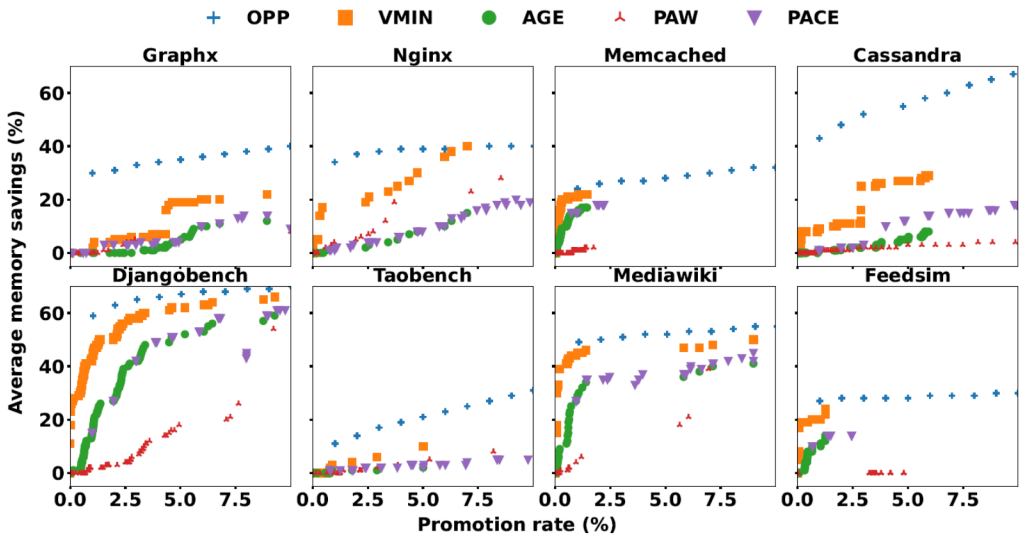


Figure 7: MPC for various policies supported by *MDK*. We show results only at lower, more practical, promotion rates. OPP and VMIN are offline optimal policies, while AGE is a practical one. PriorAgeWithWait (PAW) is a new practical heuristic policy that we introduced. As expected, OPP always has the best average memory savings for any promotion rate. PAW can perform better than AGE for workloads with predictable access pattern, while AGE performs better than PAW for unpredictable workloads. PACE, a new two-parameter policy that combines PAW and AGE, always performs as well as AGE and has the potential to outperform AGE for workloads such as Cassandra.

regime. We use our MPC results to answer questions about operating regimes, *i.e.,* "what is the best memory saving that can be achieved under a certain promotion rate?". Although not directly visible in our MPCs, the generation algorithm also helps answer "what is the control parameter setting that we should use to achieve a certain promotion rate?" as shown in Listing 1 (line 20). Our MPCs show that for multiple applications, tuning practical policies cannot achieve certain operating regimes. This can be observed in Memcached, where policies do not have any parameters that lead to a promotion rate higher than 2.5%. Collections of vertical points (often called "cliffs" for MRCs) are desirable, as they represent an increase in memory savings for a small increase in promotion rate (e.g., the AGE and VMIN policies for Memcached and MediaWiki). In other words, these vertical patterns indicate that slightly more aggressive policies improve average memory savings nearly "for free". In contrast, horizontal plateaus are undesirable: they represent cases in which increasing the promotion rate constraint does not increase memory savings, as can be observed in the case of Mediawiki. In all cases, the MPCs quantify the policy operating range by showing how a given promotion rate impacts average memory savings, enabling the selection of a reasonable target promotion rate. MPCs also provide insights about what promotion rate must be tolerated to achieve a desired average memory savings.

We compare the performance of the optimal policies, comparing OPP to VMIN. We omit OPT from discussion, because it is a fixed-sized memory policy that VMIN consistently outperforms. Understanding the performance of the optimal policy for a given setting is important, as it provides an upper bound on what can be achieved by a practical implementation. As expected, OPP (blue plus) consistently has the highest average memory savings. Varying the promotion rate constraint for workloads such as Cassandra and TaoBench can lead to better average memory savings, as indicated by the slope of the MPC. For Cassandra, OPP achieves 40% average memory savings under a 1% promotion rate, while VMIN cannot achieve the same savings even under a 10% promotion rate. For workloads such as FeedSim and MediaWiki, the average memory savings for OPP is relatively flat, indicating that it is not possible to increase savings via more aggressive reclamation (i.e., by tolerating further performance degradation). This provides insight into the performance ceiling for each workload, given different promotion rates.

Now that we understand optimal performance, we examine the performance of AGE policy that approximates Google's g-swap [37] (green circles in Figure 7) compared to OPP. AGE is a simple policy that reclaims a page if a certain time period has elapsed since its last access, and it hasn't been accessed since. This mandated waiting period prevents unnecessary reclamations, but it also prevents AGE from reclaiming pages quickly, foregoing savings. As a result, compared to OPP, the AGE policy does significantly worse. This suggests that there is ample headroom to develop a better policy. For many workloads (such as Memcached and DjangoBench), the best memory savings that the AGE policy achieves under different operating regimes scale similarly to VMIN; this is expected, since page fault characteristics are theoretically identical for

these policies [18], while savings are not. Further, the AGE policy is often conservative, as shown in the GraphX workload; the AGE policy never exceeds ∼10% average memory savings, even on the most aggressive settings, as it waits for a minimum number of time windows before reclaiming pages, making it impossible to reap the full potential savings.

5.4.2 Designing and evaluating new policies

We use the insights revealed in Figure 7 to propose policies that bridge the gap between AGE and OPP. The steps for designing a policy shown in Figure 1 are: coming up with a new policy idea, generating its MPC, which entails understanding whether *MDK* properties apply, and if they do, implementing their MPC generators in *MDK*, and comparing the new policies' performance to AGE. We describe the steps per policy for three new policies.

Prior Age with Wait (PAW)

Design: First, we develop a simple heuristic policy that reclaims pages after access, similar to OPP, by using historical reuse distances. This policy, **P**rior **A**ge with **W**ait (PAW), reclaims a page if its prior age, the distance between a page's last two accesses, is greater than a threshold P, and at least one minute has elapsed since its last access. Waiting one minute addresses potential issues for frequently used pages: if a page is recently accessed, it is likely to be accessed again, so the policy waits briefly before reclaiming it to increase confidence in the reclamation.

Generating MPC: PAW follows the *eviction decisions* and *eviction times* properties, so we implemented its generator in *MDK*. Writing the policy-specific functions took fewer than 65 LOC. For the *GetCriticalParam* function, we reused the AGE policy's implementation, as the only difference is looking at the current reuse distance versus the previous one. The other two memory functions are identical to VMIN's, as both policies follow the *eviction times* property.

Comparing performance to AGE: Figure 7 illustrates the MPC for comparing AGE with PAW. PAW outperforms AGE on workloads with predictable access patterns: it saves up to 10% more memory than AGE on GraphX, NGINX, and TaoBench. However, for workloads such as Memcached and FeedSim, where past behavior is not a good indicator of future accesses, AGE's conservative approach is better. As our results clearly show, for certain workloads where reuse distances are not a good indication for reclaiming a page, AGE outperforms PAW significantly.

Prior Age and Current Elapsed (PACE)

```
def PACE(page, P, A):
    alpha = get_prior_age(page)     # Last reuse distance
    beta  = get_current_age(page)   # Time elapsed since last
    active reference
    evict = False
    if alpha > P:
        evict = True
    if beta > A:
        evict = True
    return Evict
```

Listing 2: The PACE two parameter policy. PACE evicts pages immediately if their reuse distance is greater than a control parameter P. If the reuse distance is small, then PACE waits until time A has elapsed, similar to AGE. to reclaim a page.

Design: The **P**rior **A**ge and **C**urrent **E**lapsed (PACE), detailed in Listing 2, is a simple two-parameter policy that combines the aggressiveness of PAW with the conservative nature of AGE. PACE directly addresses the shortcomings of PAW for workloads where reuse distances are not enough to make good decisions. For each page, PACE checks if its reuse distance, *i.e.,* the difference between its prior two accesses, is greater than a prior age threshold P; if so, it evicts the page immediately. Unlike PAW, PACE does not wait for one minute before reclaiming a page. If the reuse distance is not greater than P, then PACE waits a time period A after a page's last access before reclaiming it. PACE has two control parameters, P and A. Tuning P allows PACE to evict pages immediately upon access, similar to PAW, and for pages where P is not a good indication for eviction, PACE reclaims a page after it is not accessed for A time intervals, similar to g-swap's current AGE policy.

It is trivial to see that under the correct (P,A) setting PACE must always perform as well as, if not better than AGE. If P is set to infinity, no page would ever satisfy the prior age condition; therefore PACE can always fall back to AGE. The challenging part is determining the best parameter combination for a given application. Simulating all parameters for PACE is prohibitively expensive due to many possible (P,A) values; our traces have more than 10,000 unique (P,A) combinations. Additionally, there is no intuition that allows us to guess good (P,A) combinations.

Generating MPC: We present a novel specialized algorithm for efficiently generating the MPC for the PACE policy, rooted in our theoretical properties. Although we cannot compare the aggressiveness of two arbitrary parameter combinations (P_1,A_1) and (P_2,A_2), because these parameter combinations don't follow the *eviction decisions* and *eviction times* properties, P and A individually follow these properties, implying that both $E_{(P+1,A)}$ and $E_{(P,A+1)}$ would be subsequences of $E_{(P,A)}$. This fact allows us to generate the MPC for PACE efficiently using the dynamic programming solution to the suffix sum problem. We describe the algorithm that we

implemented in about 300 loc, in detail in Appendix B.

Comparing performance to AGE: In Figure 7, we report the top three performing configurations for PACE for each 1% promotion rate increments, such as 1-2%, 2-3%. The results show the maximum potential benefit (an upper bound) when the optimal parameters are chosen on the same trace used for evaluation. As expected, PACE's best configurations always perform as well as AGE's best configurations. For most workloads, PACE obtains modest memory savings improvement of 1-4% across different promotion rates. For workloads such as Cassandra and GraphX, PACE 's best configurations outperform AGE by 8-10%; PACE increases savings by evicting pages earlier. Since AGE, PAW, and PACE have parameters dependent on application characteristics, a tuner would need to observe application characteristics online and then set *P* or *A* parameters for all three policies. Note that with more tunable parameters, the results are more susceptible to overfitting when using the exact same trace for tuning and evaluation; tuning without a priori trace knowledge for unseen traces is left for future work.

Learned OPP (L-OPP)

Design: OPP demonstrates significant headroom over AGE, however it requires future information. Inspired by prior work on learned policies in the traditional problem [29, 42, 59], we investigate whether a learned policy that imitates OPP, *L-OPP*, can improve upon AGE without needing future information. Our intent is to illustrate that OPP can be used more directly to derive a new policy, rather than to create the perfect learned policy. We frame page reclamation as a binary classification problem: given a page access and its re-access history, the model predicts whether to reclaim the page. We train a gradient boosted tree model for this task, using the open-source YDF library [24] with default hyperparameters. To build the training dataset, we first use periodic page table access bit scans to collect page access traces for the DCPerf benchmarks [61]. Then, we use our simulator to run OPP on these traces with a target promotion rate of 2% and log every reclaim/not reclaim decision along with the historical reuse distances of each page. Our goal is that L-OPP will achieve the same promotion rate. We then train a separate model for each workload on the first 80% of the dataset, using six past reuse distances for each page as features. We use the last 20% of the dataset for validation. The final L-OPP policy uses AGE to reclaim first-use pages (i.e., those that have no reuse features); for all other pages, it queries the trained model at the end of the time interval in which they have been accessed.

Generating MPCs: We implement L-OPP in our simulator, because it does not obey the *eviction decisions* and *eviction times* properties, because it doesn't have any tunable parameters. We incorporate the C++ implementation [24] of L-OPP as a new policy in our simulator to generate MPCs. For simulator testing, we use a completely different trace than the one used for training; we collect this trace by running access bit scans on a second execution of the same workload.

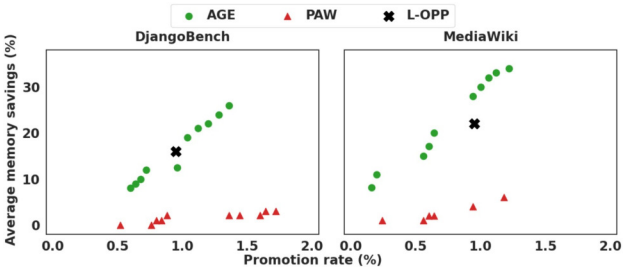


Figure 8: MPC for AGE and our learned policy, L-OPP.

Comparing policies to L-OPP: Figure 8 compares AGE, PAW, and L-OPP for the DCPerf benchmarks. L-OPP has better average memory savings than PAW for both workloads and is modestly better than AGE for DjangoBench. L-OPP is conservative and the number of per-window reclamations it makes are small, so we need to have longer workloads such as DjangoBench to accumulate savings over time to show benefit. (Similar observations have been reported for conservative non-learned policies [66].) L-OPP maintains a 1% promotion rate constraint, demonstrating that the learned policy maintains the promotion rate for which it was trained. For TaoBench and FeedSim (results omitted), L-OPP's low offline precision results in a high promotion rate, showcasing that high precision models are important to meet the promotion rate constraint. Although these initial results suggest the need for feature engineering and training at different target promotion rates to improve L-OPP's performance (a topic for future work), L-OPP demonstrates OPP's utility for policy creation, in addition to headroom analysis.

5.4.3 Validating policy end-to-end in Linux:

We evaluate *MDK*'s findings by implementing PAW and AGE in Linux. Both policies reclaim pages every 30 seconds: AGE reclaims pages that are unused for 10 minutes, and PAW reclaims pages with reuse distance greater than 2.5 minutes. We use an SSD as the swap backend and turn off hugepages. Table 3 shows the average memory consumption and swap usage of both policies for running the Page Rank benchmark on GraphX. We choose parameter settings to operate under a 4% promotion rate threshold based on our offline MPC results.

As predicted by our MPC, PAW saved 4% more memory without compromising performance. The savings are as expected, because AGE is cautious and reclaims late, while PAW reclaims pages shortly after access, increasing the average memory savings as shown in our results. The promotion rate of both policies was around 1.5%, significantly lower than the 4% threshold we expected. For promotion rate to match

the simulation and our framework, policies need to observe page accesses at similar times. However, both policies ran faster than our offline trace, and the promotion rate is directly dependent on the runtime of the application. The fact that the policies run in less time than our original trace shifts the windows in which reclamations occur, changing the observed promotion rate. There are several avenues of future work to explore to analyze and mitigate this effect: (1) Implementing *MDK* MPC generators in a self-tuning system similar to Senpai [66], would allow for accurate estimation of promotion rate while applications are running, (2) Running an analytical analysis that develops confidence bounds of the offline MPC results for online use.

Table 3: Policy Comparison on Linux for GraphX. We measure the mean and standard deviation of memory and swap usage.

Policy	Swap (GB)	Memory Usage (GB)	Time taken (mins)
PAW	0.59 ± 0.75	9.11 ± 0.84	16.9
AGE	0.23 ± 0.29	9.52 ± 0.27	17.1

6 Related work

MDK builds upon concepts from prior work on optimal algorithms, variable-sized cache policies, approaches for generating MRCs, and memory reclamation metrics.

Optimality: Several prior works focus on optimal policies for fixed-sized memories for uniform object sizes [8, 47], non-uniform object sizes [10], variable-sized memories [50], and for non-uniform memory access cost [30]. These policies do not consider performance constraints that the data center context requires. Optimal policies have been adapted to various domains including databases [13, 31, 69], cache-oblivious algorithms [22], tiered memory systems [73], and confidential applications [36]. These prior works on fixed-sized memories could be extended to the data center formulation.

Several works use optimal algorithms to train ML models [29, 40, 42, 44, 52, 56, 58–60, 65, 68, 71, 76]. We explore training a learned policy L-OPP using OPP, that demonstrates OPP's utility for policy creation.

Variable-sized policies: Working set [17] provided a theory to model applications by their memory behavior. Replacement policies inspired by this model optimize for miss ratio without a fixed-sized memory constraint [18] and generate better average memory savings than OPT while using statistical future knowledge [41]. These works are complementary to our work. Prior work explored adapting cache size based on performance metrics [3, 25], and modern application frameworks also exploit variable-sized allocation by forcing applications to shrink their internal cache footprints under pressure [23, 51]; our work focuses on variable-sized policies instead of adapting the size.

Miss ratio curves: *MDK* extends the work of Mattson et al. [47] and later work on MRC construction for algorithms other than LRU [7, 18], by introducing new theoretical properties that apply to a larger set of policies. Additional related work uses approximation to speed up Mattson's approach, including Shards [64], KOSMO [57], AET [28], and Counter Stacks [67]. *MDK* produces exact MPCs, and we leave approximate concepts for *MDK* algorithms as future work. Several systems use online MRC generation to guide cache size allocation (e.g., Memshare [11] and OSCA [74]) and tune policies (e.g., Cliffhanger [16, 27, 77]). We leave using *MDK* MPCs to tune data center systems for future work.

Memory reclamation metrics: Prior work used metrics other than miss ratio, such as fault rate [15, 55], pressure stall information (PSI) [66], promotion rate failures [70], frequency of access [53], amortized offcore latency [43], and throughput [26]. *MDK* is designed to be extensible in order to accommodate a diverse set of metrics.

7 Conclusion

Data center memory reclamation has a goal of maximizing memory savings to allow more jobs to be scheduled on servers, while maintaining application performance targets. This formulation leads to a different optimization problem than the traditional one, and the tools required to address this problem are lacking. We introduced *MDK*, an offline framework for data center policy exploration that includes a new optimal policy OPP, MPCs to evaluate policy performance, theoretical properties, and efficient algorithms to construct MPCs. Using *MDK*, we show that existing policies leave ample room for improvement. We develop new practical and learned policies by leveraging OPP insights and evaluate these policies using *MDK*, demonstrating *MDK*'s utility and the potential to improve upon existing policies. Ultimately, we hope researchers will use *MDK* to develop better memory reclamation policies for the data center problem.

Acknowledgements

The authors thank David Culler, Kathryn McKinley, John Wilkes, Hank Levy, our shepherd, and the anonymous reviewers of OSDI'26 for their feedback on the draft. The authors thank Vinay Banakar, Rajath Shashidhara, and Anil Yelam for their insights and help in collecting the page access traces required for this work.

References

[1] Homepage - Compute Express Link — computeexpresslink.org. https://computeexpresslink.org/. [Accessed 11-04-2025].

[2] FIFO queues are all you need for cache eviction. In *Proceedings of the Twenty-ninth Symposium on Operating Systems Principles*, SOSP '23, pages 130–149. Association for Computing Machinery, 2023.

[3] Raphael Alonso and Andrew W. Appel. An advisor for flexible working sets. In *Proceedings of the 1990 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '90, page 153–162. Association for Computing Machinery, 1990.

[4] Sorav Bansal and Dharmendra S. Modha. CAR: Clock with adaptive replacement. In *3rd USENIX Conference on File and Storage Technologies (FAST 04)*, San Francisco, CA, March 2004. USENIX Association.

[5] Luiz André Barroso, Jimmy Clidaras, and Urs Hölzle. *The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines, Second Edition*. 2013.

[6] Noman Bashir, Nan Deng, Krzysztof Rzadca, David Irwin, Sree Kodak, and Rohit Jnagal. Take it to the limit: peak prediction-driven resource overcommitment in datacenters. In *Proceedings of the Sixteenth European Conference on Computer Systems*, EuroSys '21, page 556–573. Association for Computing Machinery, 2021.

[7] Nathan Beckmann and Daniel Sanchez. Modeling cache performance beyond LRU. In *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 225–236, 2016.

[8] L. A. Belady. A study of replacement algorithms for a virtual-storage computer. *IBM Systems Journal*, 5(2):78–101, 1966.

[9] A. Bensoussan, C. T. Clingen, and R. C. Daley. The multics virtual memory. In *Proceedings of the Second Symposium on Operating Systems Principles*, SOSP '69, page 30–42, New York, NY, USA, 1969. Association for Computing Machinery.

[10] Daniel S. Berger, Nathan Beckmann, and Mor Harchol-Balter. Practical bounds on optimal caching with variable object sizes. *Proc. ACM Meas. Anal. Comput. Syst.*, 2(2), June 2018.

[11] Daniel Byrne, Nilufer Onder, and Zhenlin Wang. mpart: miss-ratio curve guided partitioning in key-value stores. *ISMM 2018*, page 84–95, June 2018.

[12] Marcus Carvalho, Walfredo Cirne, Francisco Brasileiro, and John Wilkes. Long-term slos for reclaimed cloud computing resources. In *Proceedings of the ACM Symposium on Cloud Computing*, SOCC '14, page 1–13. Association for Computing Machinery, 2014.

[13] Audrey Cheng, David Chu, Terrance Li, Jason Chan, Natacha Crooks, Joseph M. Hellerstein, Ion Stoica, and Xiangyao Yu. Take out the TraChe: Maximizing (tra)nsactional ca(che) hit rate. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*, pages 419–439, July 2023.

[14] Jeongku Choi. Memory prices surge up to 90% from Q4 2025. https://counterpointresearch.com/en/insights/Memory-Prices-Surge-Up-to-90-From-Q4-2025, February 2026.

[15] Wesley W. Chu and Holger Opderbeck. The page fault frequency replacement algorithm. In *Proceedings of the December 5-7, 1972, Fall Joint Computer Conference, Part I*, AFIPS '72 (Fall, part I), page 597–609, 1972.

[16] Asaf Cidon, Assaf Eisenman, Mohammad Alizadeh, and Sachin Katti. Cliffhanger: Scaling performance cliffs in web memory caches. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*, pages 379–392, Santa Clara, CA, March 2016. USENIX Association.

[17] Peter J. Denning. The working set model for program behavior. *Commun. ACM*, 11(5):323–333, May 1968.

[18] Peter J. Denning. Working set analytics. *ACM Comput. Surv.*, 53(6), February 2021.

[19] Peter J. Denning and Kevin C. Kahn. A study of program locality and lifetime functions. *SIGOPS Oper. Syst. Rev.*, 9(5):207–216, November 1975.

[20] Padmapriya Duraisamy, Wei Xu, Scott Hare, Ravi Rajwar, David Culler, Zhiyi Xu, Jianing Fan, Christopher Kennelly, Bill McCloskey, Danijela Mijailovic, Brian Morris, Chiranjit Mukherjee, Jingliang Ren, Greg Thelen, Paul Turner, Carlos Villavieja, Parthasarathy Ranganathan, and Amin Vahdat. Towards an adaptable systems architecture for memory tiering at warehouse-scale. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, ASPLOS 2023, page 727–741. Association for Computing Machinery, 2023.

[21] Jacob Fox. Looks like memory prices are set to keep increasing over 2026 before falling. https://finance.yahoo.com/news/looks-memory-prices-set-keep-152000968.html, December 2025.

[22] M. Frigo, C.E. Leiserson, H. Prokop, and S. Ramachandran. Cache-oblivious algorithms. In *40th Annual Symposium on Foundations of Computer Science (Cat. No.99CB37039)*, pages 285–297, 1999.

[23] Megan Frisella, Shirley Loayza Sanchez, and Malte Schwarzkopf. Towards increased datacenter efficiency with soft memory. In *Proceedings of the 19th Workshop on Hot Topics in Operating Systems*, HotOS '23, page 127–134, New York, NY, USA, 2023. Association for Computing Machinery.

[24] Google. Ydf: Yggdrasil decision forests. https://ydf.readthedocs.io/en/stable, 2025.

[25] Anupam Gupta, Ravishankar Krishnaswamy, Amit Kumar, and Debmalya Panigrahi. *Elastic Caching*, pages 143–156. Proceedings of the 2019 Annual ACM-SIAM Symposium on Discrete Algorithms (SODA).

[26] Xiangpeng Hao, Xinjing Zhou, Xiangyao Yu, and Michael Stonebraker. Towards buffer management with tiered main memory. *Proc. ACM Manag. Data*, 2(1), March 2024.

[27] Taekyung Heo, Yang Wang, Wei Cui, Jaehyuk Huh, and Lintao Zhang. Adaptive page migration policy with huge pages in tiered memory systems. *IEEE Transactions on Computers*, 71(1):53–68, 2022.

[28] Xiameng Hu, Xiaolin Wang, Lan Zhou, Yingwei Luo, Zhenlin Wang, Chen Ding, and Chencheng Ye. Fast miss ratio curve modeling for storage cache. *ACM Trans. Storage*, 14(2), April 2018.

[29] Akanksha Jain and Calvin Lin. Back to the future: Leveraging Belady's algorithm for improved cache replacement. In *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, pages 78–89, 2016.

[30] J. Jeong and M. Dubois. Cache replacement algorithms with nonuniform miss costs. *IEEE Transactions on Computers*, 55(4):353–365, 2006.

[31] Zhaoxuan Ji, Zhongle Xie, Yuncheng Wu, and Meihui Zhang. LBSC: A Cost-Aware Caching Framework for Cloud Databases . In *2024 IEEE 40th International Conference on Data Engineering (ICDE)*, pages 4911–4924, Los Alamitos, CA, USA, May 2024. IEEE Computer Society.

[32] Song Jiang and Xiaodong Zhang. Lirs: an efficient low inter-reference recency set replacement policy to improve buffer cache performance. *SIGMETRICS Perform. Eval. Rev.*, 30(1):31–42, June 2002.

[33] Theodore Johnson and Dennis Shasha. 2q: A low overhead high performance buffer management replacement algorithm. In *Proceedings of the 20th International Conference on Very Large Data Bases*, VLDB '94, page 439–450, San Francisco, CA, USA, 1994. Morgan Kaufmann Publishers Inc.

[34] Uksong Kang, Hak-Soo Yu, Churoo Park, Hongzhong Zheng, John Halbert, Kuljit Bains, and Joo Sun Choi. Co-architecting controllers and dram to enhance dram process scaling. In *The Memory Forum*, 2014.

[35] T. Kilburn, D. B. G. Edwards, M. J. Lanigan, and F. H. Sumner. One-level storage system. *IRE Transactions on Electronic Computers*, EC-11(2):223–235, 1962.

[36] Sam Kumar, David E. Culler, and Raluca Ada Popa. MAGE: Nearly zero-cost virtual memory for secure computation. In *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*, pages 367–385. USENIX Association, July 2021.

[37] Andres Lagar-Cavilla, Junwhan Ahn, Suleiman Souhlal, Neha Agarwal, Radoslaw Burny, Shakeel Butt, Jichuan Chang, Ashwin Chaugule, Nan Deng, Junaid Shahid, Greg Thelen, Kamil Adam Yurtsever, Yu Zhao, and Parthasarathy Ranganathan. Software-defined far memory in warehouse-scale computers. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '19, page 317–330. Association for Computing Machinery, 2019.

[38] Seok-Hee Lee. Technology scaling challenges and opportunities of memory devices. In *2016 IEEE International Electron Devices Meeting (IEDM)*, pages 1–1. IEEE, 2016.

[39] Huaicheng Li, Daniel S. Berger, Lisa Hsu, Daniel Ernst, Pantea Zardoshti, Stanko Novakovic, Monish Shah, Samir Rajadnya, Scott Lee, Ishwar Agarwal, Mark D. Hill, Marcus Fontoura, and Ricardo Bianchini. Pond: Cxl-based memory pooling systems for cloud platforms. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, ASPLOS 2023, page 574–587. Association for Computing Machinery, 2023.

[40] Pengcheng Li and Yongbin Gu. Learning forward reuse distance, 2020.

[41] Pengcheng Li, Colin Pronovost, William Wilson, Benjamin Tait, Jie Zhou, Chen Ding, and John Criswell. Beating opt with statistical clairvoyance and variable size caching. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '19, page 243–256. Association for Computing Machinery, 2019.

[42] Evan Zheran Liu, Milad Hashemi, Kevin Swersky, Parthasarathy Ranganathan, and Junwhan Ahn. An imitation learning approach for cache replacement. *CoRR*, abs/2006.16239, 2020.

[43] Jinshu Liu, Hamid Hadian, Hanchen Xu, and Huaicheng Li. Tiered memory management beyond hotness. In *19th USENIX Symposium on Operating Systems Design and Implementation (OSDI 25)*, pages 731–747. USENIX Association, July 2025.

[44] Ke Liu, Kan Wu, Hua Wang, Ke Zhou, Ji Zhang, and Cong Li. Slap: An adaptive, learned admission policy for content delivery network caching. In *2023 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 457–467, 2023.

[45] Tobias Mann. DRAM prices expected to double in Q1 as AI ambitions push memory fabs to their limit. https://www.theregister.com/2026/02/02/dram_prices_expected_to_double/, February 2026.

[46] Hasan Al Maruf, Hao Wang, Abhishek Dhanotia, Johannes Weiner, Niket Agarwal, Pallab Bhattacharya, Chris Petersen, Mosharaf Chowdhury, Shobhit Kanaujia, and Prakash Chauhan. Tpp: Transparent page placement for cxl-enabled tiered-memory. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, ASPLOS 2023, page 742–755. Association for Computing Machinery, 2023.

[47] R.L. Mattson, J. Gecsei, D. R. Slutz, and I. L. Traiger. Evaluation techniques for storage hierarchies. *IBM Systems Journal*, 9(2):78–117, 1970.

[48] Nimrod Megiddo and Dharmendra S. Modha. ARC: A Self-Tuning, low overhead replacement cache. In *2nd USENIX Conference on File and Storage Technologies (FAST 03)*, San Francisco, CA, March 2003. USENIX Association.

[49] Tapti Palit, Yongming Shen, and Michael Ferdman. Demystifying cloud benchmarking. In *2016 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 122–132, April 2016.

[50] Barton G. Prieve and R. S. Fabry. Min—an optimal variable-space page replacement algorithm. *Commun. ACM*, 19(5):295–297, May 1976.

[51] Yifan Qiao, Zhenyuan Ruan, Haoran Ma, Adam Belay, Miryung Kim, and Harry Xu. Harvesting idle memory for application-managed soft state with midas. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*, pages 1247–1265, Santa Clara, CA, April 2024. USENIX Association.

[52] Shuting Qiu, Qilin Fan, Xiuhua Li, Xu Zhang, Geyong Min, and Yongqiang Lyu. Oa-cache: Oracle approximation-based cache replacement at the network edge. *IEEE Transactions on Network and Service Management*, 20(3):3177–3189, 2023.

[53] Jie Ren, Dong Xu, Junhee Ryu, Kwangsik Shin, Daewoo Kim, and Dong Li. Mtm: Rethinking memory profiling and migration for multi-tiered large memory. In *Proceedings of the Nineteenth European Conference on Computer Systems*, EuroSys '24, page 803–817. Association for Computing Machinery, 2024.

[54] Krzysztof Rzadca, Paweł Findeisen, Jacek Świderski, Przemyslaw Zych, Przemyslaw Broniek, Jarek Kusmierek, Paweł Krzysztof Nowak, Beata Strack, Piotr Witusowski, Steven Hand, and John Wilkes. Autopilot: Workload autoscaling at google scale. In *Proceedings of the Fifteenth European Conference on Computer Systems*, 2020.

[55] E. Sadeh. An analysis of the performance of the page fault frequency (pff) replacement algorithm. In *Proceedings of the Fifth ACM Symposium on Operating Systems Principles*, SOSP '75, page 6–13, New York, NY, USA, 1975. Association for Computing Machinery.

[56] Ishan Shah, Akanksha Jain, and Calvin Lin. Effective mimicry of Belady's min policy. In *2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 558–572, 2022.

[57] Kia Shakiba, Sari Sultan, and Michael Stumm. Kosmo: Efficient online miss ratio curve generation for eviction policy evaluation. In *22nd USENIX Conference on File and Storage Technologies (FAST 24)*, pages 89–105, Santa Clara, CA, February 2024. USENIX Association.

[58] Zhan Shi, Xiangru Huang, Akanksha Jain, and Calvin Lin. Applying deep learning to the cache replacement problem. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '52, page 413–425. Association for Computing Machinery, 2019.

[59] Zhenyu Song, Daniel S. Berger, Kai Li, and Wyatt Lloyd. Learning relaxed Belady for content distribution network caching. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 529–544, Santa Clara, CA, February 2020. USENIX Association.

[60] Zhenyu Song, Kevin Chen, ikhil Sarda, Deniz Altınbüken, Eugene Brevdo, Jimmy Coleman, Xiao Ju, Pawel Jurczyk, Richard Schooler, and Ramki Gummadi. HALP: Heuristic aided learned preference eviction policy for YouTube content delivery network. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 1149–1163, Boston, MA, April 2023. USENIX Association.

[61] Wei Su, Abhishek Dhanotia, Carlos Torres, Jayneel Gandhi, Neha Gholkar, Shobhit Kanaujia, Maxim Naumov, Kalyan Subramanian, Valentin Andrei, Yifan Yuan, and Chunqiang Tang. Dcperf: An open-source, battle-tested performance benchmark suite for datacenter workloads. In *Proceedings of the 52nd Annual International Symposium on Computer Architecture*, ISCA '25, page 1717–1730. Association for Computing Machinery, 2025.

[62] Muhammad Tirmazi, Adam Barker, Nan Deng, Md Ehtesam Haque, Zhijing Gene Qin, Steven Hand, Mor Harchol-Balter, and John Wilkes. Borg: the next generation. In *EuroSys'20*, Heraklion, Crete, 2020.

[63] TrendForce. DRAM Industry Analysis–1Q26. https://www.trendforce.com/research/download/RP260305JB, March 2026.

[64] Carl A. Waldspurger, Nohhyun Park, Alexander Garthwaite, and Irfan Ahmad. Efficient MRC construction with SHARDS. In *13th USENIX Conference on File and Storage Technologies (FAST 15)*, pages 95–110, Santa Clara, CA, February 2015. USENIX Association.

[65] Peng Wang, Hong Jiang, Yu Liu, Zhelong Zhao, Ke Zhou, and Zhihai Huang. Beyond Belady to attain a seemingly unattainable byte miss ratio for content delivery networks. *IEEE Transactions on Parallel and Distributed Systems*, 35(11):1949–1963, 2024.

[66] Johannes Weiner, Niket Agarwal, Dan Schatzberg, Leon Yang, Hao Wang, Blaise Sanouillet, Bikash Sharma, Tejun Heo, Mayank Jain, Chunqiang Tang, and Dimitrios Skarlatos. Tmo: transparent memory offloading in datacenters. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '22, page 609–621. Association for Computing Machinery, 2022.

[67] Jake Wires, Stephen Ingram, Zachary Drudi, Nicholas J. A. Harvey, and Andrew Warfield. Characterizing storage workloads with counter stacks. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 335–349, Broomfield, CO, October 2014. USENIX Association.

[68] Daniel Lin-Kit Wong, Hao Wu, Carson Molder, Sathya Gunasekar, Jimmy Lu, Snehal Khandkar, Abhinav Sharma, Daniel S. Berger, Nathan Beckmann, and Gregory R. Ganger. Baleen: ML admission & prefetching for flash caches. In *22nd USENIX Conference on File and Storage Technologies (FAST 24)*, pages 347–371, Santa Clara, CA, February 2024. USENIX Association.

[69] Junyi Xie, Jun Yang, and Yuguo Chen. On joining and caching stochastic streams. In *Proceedings of the 2005 ACM SIGMOD International Conference on Management of Data*, SIGMOD '05, page 359–370. Association for Computing Machinery, 2005.

[70] Dong Xu, Junhee Ryu, Kwangsik Shin, Pengfei Su, and Dong Li. FlexMem: Adaptive page profiling and migration for tiered memory. In *2024 USENIX Annual Technical Conference (USENIX ATC 24)*, pages 817–833, Santa Clara, CA, July 2024. USENIX Association.

[71] Juncheng Yang, Ziming Mao, Yao Yue, and K. V. Rashmi. GL-Cache: Group-level learning for efficient and high-performance caching. In *21st USENIX Conference on File and Storage Technologies (FAST 23)*, pages 115–134, Santa Clara, CA, February 2023. USENIX Association.

[72] Juncheng Yang, Yao Yue, and K. V. Rashmi. A large-scale analysis of hundreds of in-memory cache clusters at twitter. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 191–208. USENIX Association, 2020.

[73] Lei Zhang, Reza Karimi, Irfan Ahmad, and Ymir Vigfusson. Optimal data placement for heterogeneous cache, memory, and storage systems. *Proc. ACM Meas. Anal. Comput. Syst.*, 4(1), May 2020.

[74] Yu Zhang, Ping Huang, Ke Zhou, Hua Wang, Jianying Hu, Yongguang Ji, and Bin Cheng. OSCA: An Online-Model based cache allocation scheme in cloud block storage systems. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pages 785–798. USENIX Association, July 2020.

[75] Yuhong Zhong, Daniel S. Berger, Carl Waldspurger, Ryan Wee, Ishwar Agarwal, Rajat Agarwal, Frank Hady, Karthik Kumar, Mark D. Hill, Mosharaf Chowdhury, and Asaf Cidon. Managing memory tiers with CXL in virtualized environments. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, pages 37–56, Santa Clara, CA, July 2024. USENIX Association.

[76] Giulio Zhou and Martin Maas. Learning on distributed traces for data center storage systems. In *4th Conference on Machine Learning and Systems (MLSys 2021)*, 2021.

[77] Pin Zhou, Vivek Pandey, Jagadeesan Sundaresan, Anand Raghuraman, Yuanyuan Zhou, and Sanjeev Kumar. Dynamic tracking of page miss ratio curve for memory management. In *Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XI, page 177–188. Association for Computing Machinery, 2004.

A MPC constructors for OPP and AGE policies

We discuss the MPC constructors for AGE and OPP for exposition.

MPC construction for AGE

AGE policy only adheres to the eviction decisions *and not the* eviction times *property.*

The AGE policy accepts a parameter R_A representing the threshold age of a page. Any page not used for R_A time intervals is reclaimed. This policy follows the eviction times property as any age that is less than R_A (i.e. more aggressive) would also reclaim the same page, thus incurring the same fault. However, this policy doesn't follow the eviction times property, as the more aggressive age would reclaim the page earlier than a less aggressive one. The page would reach that age sooner, which means that the eviction times property can't be satisfied. The critical parameter R_c computation (shown in Table 4 for AGE eviction policy is similar to VMIN's. The key difference lies in the parameter: AGE policy waits R_A time intervals for eviction, making the critical parameter one less than the reuse distance. However, it's memory savings function differs significantly from that of VMIN's. The memory savings is always the size of the element because the critical parameter (the age) of AGE is always 1 less than the reuse distance. For example, if an element is accessed at times 10 and 20, the critical age parameter would be 9, and the page would be evicted at time 19 for one time unit by the critical parameter. When accumulating memory savings, the less aggressive parameter's savings must be adjusted to account for the earlier evictions (as the *eviction times* property is violated). This adjustment involves adding the difference between the two age parameters (assumed to be 1 here) multiplied by the number of pages evicted by the less aggressive parameter, as these pages would have been evicted one time unit earlier by the more aggressive policy.

MPC construction for OPP

OPP adheres to the eviction decisions and the eviction times property.

The OPP policy that we presented follows both the theoretical properties we introduced. OPP accepts a target promotion rate, to maintain in each time window. It evicts elements as soon as they are accessed similar to VMIN adhering to the *eviction times* property. An aggressive target promotion rate will always evict the pages from the less aggressive promotion rate first. For example, all the evictions under a target promotion rate of 1% would also happen under a target promotion rate of 2% as all the pages that cause page faults under 1% setting would also be included in the 2% setting.

In the case of OPP, the size of the integer parameter space is 100, the max possible target promotion rate assuming integer promotion rates. Floating promotion rates can be configured using the *GetParamSpace* and *GetParamAggressionOrder* functions. The parameter traversal order is increasing as the least aggressive parameter (target promotion rate) is 1% and the most aggressive is 100%. This is the opposite case of VMIN and the AGE policy.

OPP MPC construction requires additional data structures. It requires maintaining the future accesses for a given page and the number of pages accessed in all windows. We construct this by first pre-processing the trace in linear time and computing the number of unique pages accessed in each time window and maintaining a per-page vector of windows in which it is accessed. Once the trace is pre-processed, the template computes the MPC. OPP's *GetCriticalParameter* function first updates the promotion rate that the page's *next* access will cause, by computing the expected number of future faults divided by the total accesses in the future period. The reason to do that is identifying the promotion rate associated with a memory access by looking at the past is not feasible efficiently, so we must update future promotion rates before the access occurs. The critical promotion rate R_C then is simply read from the same data structure that was written on the page's previous access.

OPP's memory savings is the same as VMIN as it follows the *eviction times* property, this simplifies it's memory savings computation.

B MPC construction for PACE

Listing 3 details the MPC constructor for the PACE policy. Our algorithm takes as input a page access trace, and outputs the memory savings and promotion rate associated with every (P,A) combination. The core logic relies on using suffix sum for computing the MPC. The algorithm first computes $\text{Raw}_{(p,a)}$, counting the number of pages with exact reuse distance p and elapsed age a. We then compute a Co-Age threshold matrix, denoted as $C_{(p,a)}$, that counts pages that would be evicted for reuse distance greater than P **and** elapsed time greater than A as $C_{(p,a)} = \text{Raw}_{(p,a)} + C_{(p+1,a)} + C_{(p,a+1)} - C_{(p+1,a+1)}$. This counts instances where pages meet or exceed thresholds p and a simultaneously; a summation of the pages that exactly have a reuse distance and elapsed time of P and A and all the pages that satisfy less aggressive parameters, *i.e.,*$(P+1,A)$ and $(P,A+1)$, subtracting the evictions due to $(P+1,A+1)$, the intersection of the two settings to avoid double counting the same evictions. The correctness of this recursion lies in the fact that P and A follow *eviction decisions* and *eviction times* individually, allowing us to use the well-known dynamic programming solution to the suffix sum problem for solving this recursion efficiently.

PACE evicts pages if they satisfy the reuse distance P **or** the current idle limit A. We use the Co-Age threshold matrix

Table 4: Policy details for *MDK* MPC generator that allows for easy implementation of MPC algorithms

Policies (LOC)	**GetParamSpace**	**GetParamAggressionOrder**	**GetCriticalParam (page_id, time)**	**GetMemorySavings (page_id, time)**	**AccumulateSavings (savings)**
LRU (53)	max_cache_size	$\{c \in N \mid \text{max_c} \geq c \geq 1\}$	return stack.index_of(page_id)	N/A	N/A
VMIN (40)	max_distance	$\{d \in N \mid \text{max_d} \geq d \geq 1\}$	return time - prev_access[page_id]	return (time - prev_access[page_id]) $\times$*Page_size*	return savings
AGE (52)	max_distance	$\{d \in N \mid \text{max_d} \geq d \geq 1\}$	return (time - prev_access[page_id]) - 5 min	return Page_size	return savings + Page_size $\times$(*npages_reclaimed*)
OPP (87)	100	$\{r \in N \mid 1 \leq r \leq 100\}$	curr_PR = saved_PR[page_id] j = next_access(page_id) F(j)++ saved_PR[page_id] = F(j)/W(j) return curr_PR	return (time - prev_access[page_id]) $\times$*Page_size*	return savings
PriorAgeWithWait (PAW) (62)	max_distance	$\{d \in N \mid \text{max_d} \geq d \geq 1\}$	interval = prev_access[page_id] - prev_reaccess[page_id] return interval < 1 min ? 0 : interval	return (time - prev_access[page_id]) $\times$*Page_size*	return savings

C to compute the final promotion rate, $Promotion_rate_{(P,A)}$ as $Promotion_rate_{(P,A)} = C_{(P,1)} + C_{(0,A)} - C_{(P,A)}$. Here, $C_{(P,1)}$ computes promotions arising due to pages that have a reuse distance greater than P and are going to elapse at least one time interval after their eviction, *i.e.,* their age would be at least 1 on reaccess. $C_{(0,A)}$ captures age-based promotions due to pages that would have an elapsed time greater than A. Finally, subtracting the intersection $C_{(P,A)}$ corrects the overlapping promotions that both conditions would satisfy.

The memory savings $Savings_{(P,A)}$ computation uses the same logic as the promotion rate. However, it requires an adjustment. Pages would be evicted at different times based on the condition they satisfy. Suppose a page with reuse distance $\alpha = T_2 - T_1$ accessed at time T_1, with next access at T_2. When the page is refaulted at T_2, we need to evaluate the savings generated. If $\alpha > P$, then the page is removed from memory immediately at T_1, and the savings generated by this eviction would be $(T_2 - T_1) \times page_size$. If $\alpha < P$, then the page has to stay idle for a time period A before it is evicted; in this case, the savings generated would be $(T_2 - T_1 - A) \times page_size$. The savings need to be adjusted for this waiting period. To handle this, along with $Raw_{(p,a)}$, we also compute $Gap_{(p,a)}$, which computes the maximum time a page stays out of memory for every two accesses to it; in our example, it is $(T_2 - T_1) \times page_size$. The Co-Age Threshold matrix for memory savings is computed as $C_sv_{(p,a)} = \text{Gap}_{(p,a)} + C_sv_{(p+1,a)} + C_sv_{(p,a+1)} - C_sv_{(p+1,a+1)}$. For the final savings, we subtract the adjustment required to accurately count the savings for the AGE policy using, $Savings_{(P,A)} = C_sv_{(P,1)} + C_sv_{(0,A)} - C_sv_{(P,A)} - (num_pages_age \times A)$, we achieve this by keeping track of the number of pages evicted due to the AGE condition in our algorithm.

C Proof of Optimality for OPP

We first focus on a trace collected by reading accessed bits similar to existing data center systems, where all accesses within a time window W_n are assumed to have occurred at the exact same time, $t_{start}(W_n)$, the start time of the window. The kernel can only observe accesses to pages by reading the access bit at regular time intervals, only allowing it to assign that time interval as the time of access when the tracing happens.

We prove the optimality of the OPP algorithm by contradiction using a "first difference" argument. An optimal schedule maximizes total memory savings while adhering to all constraints.

Definition of Memory Saving: Let the saving from a single eviction of page p at time t_{evict} be defined as:

$$s(p, t_{\text{evict}}) = \left\{ t_{\text{next}}(p) - t_{\text{evict}} \right\}$$

where $t_{\text{next}}(p)$ is the time of the next access to page p. The

Listing 3: MPC generator for the PACE policy

```
def pace_mpc_generator(trace, scan_period, page_size):
    min_t, max_t = min(t for pid, t in trace), max(t for pid
        , t in trace)
    num_windows = (max_t - min_t) / scan_period
    dim = num_windows + 1
    Raw, Gap = [[[]]], [[[]]] # 3d array dimensions (dim,
        dim, num windows)
    prev_w, prev_prev_w = {}, {}
    for page_id, t_now in trace:
        w = (t_now - min_t) / scan_period
        if prev_w.get(page_id, -1) != -1:
            p = prev_w[page_id] - prev_prev_w.get(page_id,
                prev_w[page_id])
            a = w - prev_w[page_id] - 1
            Raw[min(p, dim-1), min(a, dim-1), w] += 1
            Gap[min(p, dim-1), min(a, dim-1), w] += (w -
                prev_w[page_id]) * page_size
        prev_prev_w[page_id], prev_w[page_id] = prev_w.get(
            page_id, -1), w
    C, C_sv = [[[]]], [[[]]] # 3d array dimensions (dim, dim
        , windows)
    for p in range(dim - 1, -1, -1):
        for a in range(dim - 1, -1, -1):
            for w in range(num_windows):
                C[p, a, w] = Raw[p, a, w] + C[p+1, a, w] + C
                    [p, a+1, w] - C[p+1, a+1, w]
                C_sv[p, a, w] = Gap[p, a, w] + C_sv[p+1, a,
                    w] + C_sv[p, a+1, w] - C_sv[p+1, a+1, w
                    ]
    results = []
    for P in range(1, dim):
        for A in range(1, dim):
            total_promotions, total_savings = 0, 0
            for w in range(num_windows):
                total_promotions += C[P, 1, w] + C[0, A, w]
                    - C[P, A, w]
                num_pages_age = C[0, A, w] - C[P, A, w]
                total_savings += C_sv[P, 1, w] + C_sv[0, A,
                    w] - C_sv[P, A, w] - (num_pages_age * A
                     * page_size)
            results.append(Result(P, A, total_savings,
                total_promotions))
    return results
```

total memory saving for a schedule S is the sum over all its evictions assuming uniform page size:

$$\text{Saving}(S) = \sum_{(p,t_{\text{evict}})\in S} s(p,t_{\text{evict}})$$

Let S_{OPP} be the schedule from the OPP algorithm. Assume there exists a hypothetical optimal schedule S_{opt} with $\text{Saving}(S_{\text{opt}}) > \text{Saving}(S_{\text{OPP}})$. This implies the schedules must differ.

Let's consider the first access event in the trace, (P,t_p), where the decisions of the two schedules diverge as both schedules cannot be identical if $\text{Saving}(S_{\text{opt}}) > \text{Saving}(S_{\text{OPP}})$. Both schedules were identical for all accesses prior to t_p, their internal states (e.g., fault counts for future windows) are identical at this point. One theoretical way for them to differ is if S_{opt} evicts P at t_p, but S_{OPP} does not. However, this is impossible. If S_{opt} evicts P, the eviction must be valid, i.e., the fault slot in the future window W_n is available and this eviction doesn't violate the promotion rate. Because the states are identical, the eviction would also be valid for S_{OPP}. The OPP algorithm is maximally greedy by definition: it **always** performs an eviction if it is valid. Therefore, S_{OPP} would also have evicted P, which contradicts our premise that the schedules differ.

Thus, the first point of divergence can only be the other way around:

$$(P,t_p) \in S_{\text{OPP}} \quad \text{and} \quad (P,t_p) \notin S_{\text{opt}}$$

For OPP to evict P, the eviction must be valid, contributing a saving of $s(P,t_p) > 0$. Let the next access to P be in window W_n, which means this eviction consumes one fault from W_n's budget.

Now, consider why S_{opt} would not evict P. There are two possibilities for how S_{opt} uses the fault budget in window W_n:

Case A: S_{opt} never uses the fault slot in W_n that P's eviction would have taken. In this case, S_{opt} simply forgoes the savings $s(P,t_p)$ by not reclaiming this page. We could construct a new schedule $S' = S_{\text{opt}} \cup \{(P,t_p)\}$. This schedule is valid because the required fault slot in W_n was available in S_{opt}. The savings of this new schedule would be $\text{Saving}(S') = \text{Saving}(S_{\text{opt}}) + s(P,t_p)$, which is strictly greater than the savings of S_{opt}. This contradicts the assumption that S_{opt} is optimal.

Case B: S_{opt} uses the same fault slot in W_n for another page, Q. For S_{opt} to use the fault slot on a different page, it must evict a page Q at some time t_q, where Q's next access is also in window W_n. Since we are at the **first** point of difference at time t_p, it must be that $t_p \le t_q$. Let's construct a new schedule S' from S_{opt} by swapping the decision for Q with the decision for P:

$$S' = (S_{\text{opt}} \setminus \{(Q,t_q)\}) \cup \{(P,t_p)\}$$

This new schedule S' is valid. It uses the same fault slot in W_n that S_{opt} used, just for a different page.

$$\text{Saving}(S') = \text{Saving}(S_{\text{opt}}) - s(Q,t_q) + s(P,t_p)$$

The saving for each eviction is $s(p,t_{\text{evict}}) = t_{\text{next}}(p) - t_{\text{evict}}$. Under our assumption, both P and Q are re-accessed at the exact same time, so $t_{\text{next}}(P) = t_{\text{next}}(Q) = t_{start}(W_n)$.

Therefore, the difference in savings becomes:

$$s(P,t_p) - s(Q,t_q) = (t_{start}(W_n) - t_p) - (t_{start}(W_n) - t_q) = t_q - t_p$$

Since we established that $t_p \le t_q$, it follows that $t_q - t_p \ge 0$. Thus, $\text{Saving}(S') \ge \text{Saving}(S_{\text{opt}})$. We have transformed S_{opt} into a schedule that agrees with OPP's first decision without decreasing the total savings. By repeating this argument for all points of difference, we can show that $\text{Saving}(S_{\text{OPP}}) \ge \text{Saving}(S_{\text{opt}})$, which contradicts our initial assumption. Therefore, under this setting, OPP is optimal.

D Investigating OPP optimality for traces with exact timestamps

Now, we relax the assumption and allow accesses to be spread within a window given a trace with exact time stamps for each memory access. The below counter-example demonstrates that OPP is not optimal in this case.

Parameters:

- Window Size (W): 100 time units. ($W_1 : [1,100], W_2 : [101,200]$)
- Target Promotion Rate (ρ_{target}): 0.5 (50%)
- Durable Threshold (T_{durable}): 50

Access Trace (T):

$$T = [(A, t = 10), (B, t = 20), (A, t = 110), (B, t = 190)]$$

There are two accesses in W_2, so its fault budget is $0.5 \times 2 = 1$.

OPP's Sub-Optimal Schedule:

1. At $t = 10$, OPP considers evicting page A. The next access is at $t = 110$. The eviction is valid ($110 - 10 = 100 \geq T_{\text{durable}}$) and a fault slot is available in W_2. OPP evicts A and uses the only fault slot for W_2. The saving is $110 - 10 = 100$.

2. At $t = 20$, OPP considers evicting page B. While the eviction is valid ($190 - 20 = 170 \geq T_{\text{durable}}$), the fault budget for W_2 is now full. OPP cannot evict B.

The total saving for the OPP schedule is **100**.

Optimal schedule for exact trace: An optimal algorithm would compare the potential savings. Evicting A yields a saving of 100, while evicting B yields a saving of 170. Knowing it only has one fault slot, the optimal schedule would forgo evicting A to save the slot for the more profitable eviction of B. The total saving would be **170**.

Since $170 > 100$, OPP is not optimal for this scenario.

Proof of Bounded Gap for OPP schedule

We prove that the total gap between the savings of ideal S_{opt} for this scenario and S_{OPP} is bounded.

Let $E_n(S)$ be the set of pages evicted by schedule S that fault in window W_n, and let F_n be the fault budget for that window. The number of such evictions is bounded, so $|E_n(S)| \leq F_n$. Because OPP is maximally greedy, it will always make at least as many valid evictions for a given window as any other valid schedule. Thus, $|E_n(S_{\text{OPP}})| \geq |E_n(S_{\text{opt}})|$. Let $m_n = |E_n(S_{\text{OPP}})|$ and $k_n = |E_n(S_{\text{opt}})|$, where $m_n \geq k_n$.

The total savings gap is the sum of the gaps from each fault window:

$$\text{Gap}_{\text{total}} = \text{Saving}(S_{\text{opt}}) - \text{Saving}(S_{\text{OPP}}) =$$

$$\sum_{n=1}^{N_{\text{windows}}} \left(\sum_{Q_i \in E_n(S_{\text{opt}})} s(Q_i) - \sum_{P_j \in E_n(S_{\text{OPP}})} s(P_j) \right)$$

Let's analyze the gap for a single window n. Since $m_n \geq k_n$ and all savings $s(P_j)$ are non-negative, we can find an upper bound by dropping the extra $m_n - k_n$ terms from OPR's savings:

$$\text{Gap}_n \leq \sum_{i=1}^{k_n} s(Q_i) - \sum_{j=1}^{k_n} s(P_j)$$

where $\{P_j\}$ represents the first k_n pages chosen by OPP for window n (i.e., those with the earliest access times).

$$\text{Gap}_n \leq \left(\sum_{i=1}^{k_n} t_{\text{next}}(Q_i) - \sum_{j=1}^{k_n} t_{\text{next}}(P_j) \right) +$$

$$\left(\sum_{j=1}^{k_n} t_{\text{evict}}(P_j) - \sum_{i=1}^{k_n} t_{\text{evict}}(Q_i) \right)$$

Now we bound the two parenthesized terms.

1. Bounding the t_{evict} term: The set $\{P_j\}$ contains the k_n valid evictions with the earliest possible eviction times. The set $\{Q_i\}$ is another set of k_n valid evictions. By definition, the sum of the earliest eviction times must be less than or equal to the sum for any other set of the same size. Therefore, $\sum t_{\text{evict}}(P_j) \leq \sum t_{\text{evict}}(Q_i)$, which implies the second term is non-positive and can be dropped for an upper bound:

$$\text{Gap}_n \leq \sum_{i=1}^{k_n} t_{\text{next}}(Q_i) - \sum_{j=1}^{k_n} t_{\text{next}}(P_j)$$

2. Bounding the t_{next} term: For any page X in these sets, $t_{\text{next}}(X)$ must fall within window W_n. Let $t_{start}(W_n)$ be the start time of this window. We can establish firm bounds: $t_{start}(W_n) \leq t_{\text{next}}(X) < t_{start}(W_n) + W$. To maximize the gap, we maximize the sum for Q_i and minimize the sum for P_j:

$$\sum t_{\text{next}}(Q_i) < \sum_{i=1}^{k_n} (t_{start}(W_n) + W) = k_n \cdot (t_{start}(W_n) + W)$$

$$\sum t_{\text{next}}(P_j) \geq \sum_{j=1}^{k_n} t_{start}(W_n) = k_n \cdot t_{start}(W_n)$$

Substituting these into our inequality for Gap_n:

$$\text{Gap}_n < k_n \cdot (t_{start}(W_n) + W) - k_n \cdot t_{start}(W_n) = k_n \cdot W$$

Since $k_n = |E_n(S_{\text{opt}})| \leq F_n$, we can conclude that $\text{Gap}_n < F_n \cdot W$. The gap for a single window is strictly bounded by its fault budget multiplied by the window size.

3. Bounding the Total Gap: The total gap is the sum of the per-window gaps. Let $F_{\text{total}} = \sum F_n$ be the sum of all fault budgets over the entire trace.

$$\text{Gap}_{\text{total}} = \sum_{n=1}^{N_{\text{windows}}} \text{Gap}_n <$$

$$\sum_{n=1}^{N_{\text{windows}}} (F_n \cdot W) = W \cdot \sum_{n=1}^{N_{\text{windows}}} F_n = F_{\text{total}} \cdot W$$

Thus, the total savings gap is strictly bounded by the total fault budget multiplied by the window size. This proves the algorithm is near-optimal.

USEC: A User-Requirement-Driven Mandatory Access Control Framework for Operating Systems (Operational Systems)

Yu Jiang[1], Wenhuan Liu[1,2], Fuchen Ma[1*], Yuheng Shen[1], Yuanliang Chen[1*]
Lei Zhang[2], He Li[2], Quan Zhang[3], Chijin Zhou[3]

[1]*School of Software, Tsinghua University, KLISS, BNRist, Beijing, China*
[2]*UnionTech Software Technology Co., Ltd, Beijing, China*
[3]*East China Normal University, Shanghai, China*

Abstract

Fine-grained access control over kernel resources is essential for containing compromised applications and protecting modern operating systems. However, mainstream mandatory access control (MAC) mechanisms such as SELinux are notoriously hard to configure, incur non-trivial performance overhead, and often break compatibility. In practice, many enterprise Linux deployments disable SELinux by default due to its complexity, configuration burden, and compatibility issues.

In this paper, we present `USEC`, a new kernel access-control framework co-designed with security vendors to make strong MAC practical at scale. `USEC` introduces: (1) simpler configuration via resource-centric policy templates and semantic resource classes; (2) a demand-driven enforcement path with decision caching that reduces kernel overhead; and (3) binary-compatible LSM interfaces for process lifecycle, file I/O, and socket events that coexist with existing modules for compatibility. We implement `USEC` as a Linux security extension and evaluate it in terms of configuration simplicity, runtime overhead, and compatibility. Under the same security requirements,`USEC` policies contain up to 10× fewer lines of policy code than SELinux, while reducing runtime overhead by 3.4%–17.1% relative to SELinux across representative server and desktop workloads. `USEC` has been adopted by over 210 security vendors, including QiAnXin, 360, and NSFOCUS. As of early 2025, it has been deployed on more than 8,000,000 enterprise endpoints in production. These results demonstrate that `USEC` can provide strong, configurable kernel protection that is both efficient and widely deployable.

1 Introduction

Operating systems rely on the kernel to mediate all accesses from user processes to kernel-managed resources such as files, devices, IPC endpoints, and network sockets. Fine-grained control of these kernel resources is crucial for containing compromised applications and enforcing least privilege in practice. Modern Linux therefore uses kernel security frameworks [22] and mandatory access control (MAC) policies to decide, on each system call, whether an operation should be allowed, denied, or audited.

However, mainstream kernel MAC frameworks such as SELinux [12] are notoriously hard to deploy in practice. For example, several Oracle enterprise products either explicitly do not support SELinux or fail when it is enabled [8, 9]: the Oracle HSM client and Oracle Database documentation note that 'Security-Enhanced Linux (SELinux) is not supported on Oracle Automatic Storage Management Cluster File System (Oracle ACFS) file systems' [7].

Generally, existing MAC frameworks are hard to use due to three major limitations: **First, policy configuration is complex:** administrators must write and maintain thousands of low-level rules tied to kernel resources and operations, making policies difficult to understand, audit, and evolve as software stacks change. **Second, the overhead is relatively high:** fine-grained checks on frequent file, IPC, and socket operations introduce non-trivial runtime cost, so operators often disable strict policies on latency-sensitive or high-throughput workloads. **Third, compatibility is fragile:** strict MAC policies frequently break legacy or proprietary applications, and diagnosing such failures is painful, which discourages enabling these mechanisms by default in production systems. However, addressing these three limitations remains difficult for the following reasons.

The first challenge lies in designing a practical configuration mechanism for MAC policies, as resource types and application behaviors are complex and continuously evolving. Existing frameworks such as SELinux expose a very low-level policy interface: administrators must assign security labels to subjects and objects and then express access control in terms of type pairs and operations. In realistic deployments, this quickly leads to thousands of types and rules spread across multiple policy modules, making it hard to understand the global effect of a change, to review policies, or to evolve them as the software stack changes.

The second challenge is to reduce MAC enforcement over-

*Fuchen Ma and Yuanliang Chen are the corresponding authors.

head without weakening control over the resources that actually need protection. Current frameworks such as SELinux install a large number of hooks along system-call and kernel-subsystem paths (files, processes, IPC, networking) and invoke the MAC engine on almost every access. For example, prior work [26] reports about an 87% slowdown on Linux v5.3 under a much more MAC-intensive setting. However, because policies are written under the assumption that all these hooks are present, selectively removing or coalescing checks without creating security blind spots is extremely difficult, forcing operators to choose between strong but expensive enforcement and weaker ad-hoc configurations.

The third challenge is how to enforce vendor-specific MAC policies without breaking the installation and execution of current applications. Current MAC frameworks, such as SELinux, typically control access at the file path and label type levels. Distributions and products therefore ship default policies that restrict writes to many system directories; however, real-world applications often need to create or modify files in various locations during installation and updates. In practice, it is very hard to decide in advance which resources must be strictly protected and which can remain flexible across vendors, so minor errors in MAC policy design can easily lead to compatibility issues for end users.

In this paper, we present `USEC`, a new MAC framework for Linux that addresses these three challenges. (1) To simplify configuration, `USEC` designs a resource-centric policy abstraction. Instead of requiring administrators to enumerate domain–type pairs and maintain thousands of allow rules, `USEC` anchors every policy rule directly at the resource being protected, and expresses which principals may read, write, or execute it. For example, on UOS V25, the deployed USEC policy uses only 11 policy modules and a 949 KB policy file, whereas the corresponding SELinux policy contains 320 modules and a 2.1 MB policy file. USEC also reduces the number of `file_contexts` rules from 5,428 to 1,577 and the number of `homedirs` rules from 408 to 65. (2) To reduce enforcement overhead, `USEC` adopts demand-driven security enforcement: vendors declare only the few security-critical capabilities they need, and `USEC` enables hooks only along those paths, letting other operations bypass the MAC engine. (3) To improve compatibility, `USEC` adds a small set of binary-compatible LSM hooks for process lifetime, file I/O auditing, and socket events, providing the necessary observability while coexisting with existing kernel security modules.

To evaluate whether `USEC` can achieve these goals, we implement it as a Linux security extension and compare it against SELinux and other MAC frameworks under multiple server and desktop operating-system workloads. Under equivalent security requirements, `USEC` policies contain up to 10× fewer rules than other frameworks, while consistently incurring lower runtime overhead than SELinux across all three evaluated benchmarks. To further evaluate its real-world effectiveness, we build a UAPP ecosystem [20] to collaborate with more than 210 security vendors such as QiAnXin and 360, and promote `USEC` among them. By early 2025, it has been deployed on more than 8,000,000 enterprise endpoints [19], including financial institutions and critical-infrastructure clients in the energy and transportation sectors.

Overall, this paper makes the following contributions:

- We design the `USEC` MAC framework with a resource-centric policy model, demand-driven security enforcement, and compatibility security interface.These techniques ensure secure access to explicitly protected kernel resources while supporting highly available, online policy updates.
- We implement `USEC` as a Linux security extension, and more than 210 security vendors adopt it. By early 2025, we have deployed over 8,000,000 endpoints with `USEC`.
- We evaluate `USEC` against SELinux and other frameworks. Under equivalent security requirements, `USEC` policies contain 10× fewer lines of policy code than SELinux and consistently incur lower (3.4%–17.1%) runtime overhead.

2 Background

2.1 Mandatory Access Control

Mandatory Access Control (MAC) is a security mechanism that operates within the operating system and enforces access to system resources [2, 10, 23]. In MAC, both subjects (e.g., processes or threads) and objects (e.g., files, ports, memory segments) are assigned security attributes or labels, and access requests are evaluated by comparing these labels against a centrally defined policy. The principal workflow typically involves: (1) labeling of subjects and objects, (2) a policy that defines permitted interactions based on labels, (3) runtime enforcement where each access request is checked against the policy, and (4) auditing or logging of access decisions for verification and compliance. Because the control is not at the user side, MAC systems offer isolation and can reduce the risk of misuse by users or compromised processes.

On Linux, MAC is typically implemented through the Linux Security Modules (LSM) framework [22]. The kernel exposes security hooks for system calls, file operations, and inter-process communication. And MAC modules such as SELinux [12] and AppArmor [1] register callbacks on these hooks to consult their policies before an operation proceeds.

2.2 How MAC Works in Practice

In a general MAC implementation, as shown in Figure 1, a user-space program issues a system call such as `open()` or `connect()` to access some resource. The call enters the kernel, which first resolves the concrete object being accessed (e.g., kernel file). Before the kernel actually accesses the target object, the MAC logic runs. In detail, the kernel exposes security hooks at critical points, and MAC modules register

callbacks on them. When a hook is invoked, the MAC subsystem looks up the security labels of the calling process and the target object. These labels describe which application or domain a process belongs to, or how sensitive or trusted a file is. The MAC policy engine then determines whether the requested operation is permitted for this label and action combination. Because these checks run inside the kernel's trusted MAC layer, user-space code cannot bypass them. If the policy allows, the system call proceeds to the resources; if it denies, the kernel aborts the call and returns an error.

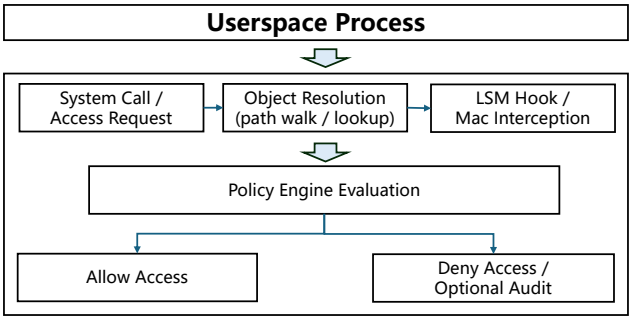


Figure 1: Basic runtime workflow of MAC enforcement

3 Motivating Examples

3.1 MAC Configuration in Practice

Consider a security vendor that wants to ship an endpoint protection agent for Linux servers. The vendor's policy goal is like: (1) only the agent process may read and write files under `/opt/agent/data/`, (2) system administrators may install or upgrade the agent binary, but cannot tamper with its runtime data, and (3) a separate audit operator may only read logs under `/var/log/agent/`. In addition, the customer requires a three-way separation of duties among the system administrator, security administrator, and audit administrator.

Expressing this policy in SELinux quickly becomes nontrivial. The vendor must first introduce several new types (e.g., `agent_t`, `agent_exec_t`, `agent_data_t`, `agent_log_t`) and attributes, add corresponding `file_context` rules so that different installation paths on different distributions are labeled correctly, and then write `allow` rules that tie these types to existing domain types such as `sshd_t`, `init_t`, and distribution-specific admin domains. To enforce separation of duties, the policy must also define SELinux users and roles (`sysadmin_u`, `secadmin_u`, `audit_u`), map Unix accounts to these users, and encode role-specific constraints over the `_u` field using low-level `constraint` statements that govern who is allowed to change which labels on which objects.

In a realistic deployment, this configuration is hard to get right and even harder to maintain. Small changes to the underlying Linux distribution (e.g., moving the agent's data directory, renaming an admin domain, or adding a new log location) may require coordinated edits across multiple SELinux modules and constraint blocks. A missing or overly restrictive rule leads to silent denials that are difficult for non-experts to diagnose; an overly permissive rule weakens the intended separation of duties and may only be discovered during an audit. As a result, vendors either ship large, complex policy modules that few customers are willing to enable, or avoid using SELinux's full model and fall back to ad-hoc scripts and DAC-based conventions.

Requirements: To be usable in real deployments, a MAC mechanism should expose a configuration model that matches how administrators and vendors reason about security, rather than forcing them to work directly with low-level kernel types and constraints. Concretely, policies should be expressed in terms of applications, roles, and a small number of explicitly protected resource groups (e.g., "agent data," "logs," "keys"), rather than thousands of distribution-specific label types. The mechanism should provide built-in templates for common patterns (such as three-way separation of duties between system, security, and audit administrators), allow policies to be updated incrementally without editing multiple interdependent modules, and offer automated checks or tooling to catch conflicts and missing rules before deployment. Such a configuration model would let vendors and operators describe high-level intent concisely, while the MAC framework is responsible for translating that intent into concrete enforcement on the target kernel.

3.2 Overhead under MAC Enforcement

The overhead of MAC enforcement is a significant concern, particularly in high-performance computing environments. A primary example of this is the performance degradation caused by SELinux. While offering robust security, SELinux achieves this by instrumenting the kernel with a large number of hooks at various points in the system call path.

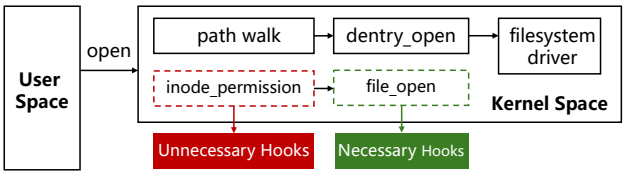


Figure 2: Illustration of how SELinux hooks along a simple file-open path.

For instance, as shown in Figure 2, even a simple file-open request must pass through several MAC check points after the `open()` syscall enters the kernel and before the filesystem is actually accessed. In a simple "allow-all" or coarse-grained policy, only the final decision point (green box) is effectively needed, while the earlier checks (red boxes) never change the outcome but are still executed on every call. Zhang et al. [26] show that on Linux 5.3, merely enabling such hooks

can increase the latency of `open()` by up to 87% compared to a DAC-only kernel, meaning that much of the overhead comes from traversing generic MAC hooks without any corresponding security benefit.

Requirements: This highlights a critical requirement for a practical MAC framework: the ability to enforce security policies with minimal overhead. A suitable design should derive, from the declared protection scope, the precise kernel paths on which hooks are actually required. By retaining mediation only on those capability-relevant paths, it becomes possible to significantly reduce performance overhead without weakening protection for the resources explicitly placed under MAC control. Such an approach avoids burdening the system with unnecessary security checks, thereby improving performance while preserving strong protection where it is needed.

3.3 MAC-Induced Compatibility Failures

Beyond configuration complexity and runtime overhead, a third motivation comes from frequent compatibility failures when MAC is enabled in real systems. In practice, many enterprise applications assume DAC-style permissions and break once SELinux starts enforcing path- and type-based policies: installers fail to create or modify files under protected directories, backup or security suites lose access to devices, and camera or container services refuse to start until SELinux is disabled or put into permissive mode [3, 4, 25].

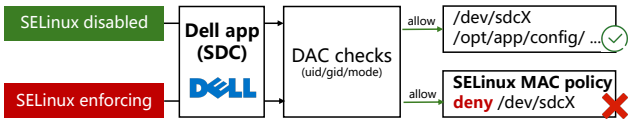


Figure 3: Illustration of a Dell application SDC failed to load correctly with SELinux enabled.

A concrete example comes from Dell's Storage Data Collector (SDC) software [4]. As shown in Figure 3, when SELinux is configured in enforcing mode, SDC fails to start and certain management operations cannot be performed, even though all traditional UNIX permissions are correctly set. The root cause is that SELinux's type-based policies block SDC from accessing specific device files and system directories that the application assumes are usable during startup and runtime. From the administrator's perspective, the system passes DAC checks but the MAC layer silently denies these accesses, causing the service to fail until SELinux is switched to permissive mode or disabled entirely. This illustrates how, once MAC enforcement is tied to fine-grained path and type rules, it becomes easy for a seemingly reasonable policy to break legitimate software. It's hard to predict which applications will stop working when SELinux is turned on.

Requirements: To avoid breaking legitimate software, a practical MAC mechanism should provide strong protection only where it is explicitly needed, while remaining non-intrusive elsewhere. Concretely, enforcement on a given resource should be *opt-in*: critical files, devices, and sockets that require protection are explicitly bound to MAC policies, while other resources fall back to DAC or audit-only behavior rather than being silently blocked. The MAC layer should also support a learning or shadow-enforcement mode that observes real deployments, reports would-be denials, and lets vendors refine their policies before switching to strict enforcement. This is particularly important in practical deployments, where vendors often need to harden only a small set of high-value resources, yet cannot afford widespread disruption to installation, update, or normal execution paths elsewhere in the system. Such a compatibility-aware design allows operators to protect the resources they explicitly care about, while keeping the rest of the software stack largely unaffected.

4 USEC Design

4.1 Threat Model and Protection Scope

`USEC` follows an *explicit protection model*: rather than mediating all kernel objects, it enforces strong protection only for resources covered by the declared capability set.

Formally, let O denote the set of kernel objects visible to the system, such as files, devices, IPC endpoints, and sockets, and let $\mathcal{C}$ denote the set of capabilities declared by the deployment. Each capability $c \in \mathcal{C}$ corresponds to a protected resource set $O_c \subseteq O$ and a set of security-sensitive operations $\mathcal{A}_c$ that may affect those resources. `USEC` guarantees that accesses to $\bigcup_{c\in\mathcal{C}} O_c$ through operations in $\bigcup_{c\in\mathcal{C}} \mathcal{A}_c$ are mediated by the retained hook set derived from $\mathcal{C}$.

We consider an attacker who can execute unprivileged or application-level code and invoke normal kernel interfaces, including file operations, IPC, device access, and networking paths, in an attempt to access, modify, disable, or indirectly reach protected resources. We assume kernel integrity, correct operation of the LSM framework, and trustworthiness of the small management TCB, including `usecd`, `dbus-daemon`, `udevd`, and the audit infrastructure. Accordingly, `USEC` is designed to stop unauthorized accesses to declared critical resources, but it does not attempt to automatically infer missing security-critical resources; ensuring that all relevant resources are included in $\mathcal{C}$ remains a deployment responsibility.

This scope is important for interpreting `USEC`'s guarantees. The goal is not complete mediation over all kernel objects, but a closed protection boundary for explicitly declared high-value resources. For protected file objects, `USEC` enforces protection on kernel object state rather than user-visible names whenever possible. As a result, alias-based or indirect accesses, such as hard links, symbolic links, rename, or file-descriptor passing, are mediated through capability-relevant hooks rather than trusted merely because they use different names. More broadly, the resource-centric design reduces

the attack surface of policy configuration and common proxy paths by directly protecting the resources and management interfaces that matter, while leaving undeclared resources outside the strong-protection scope.

4.2 Resource-Centric Policy Model

Traditional kernel MAC frameworks such as SELinux [12] and AppArmor [1] are fundamentally *process/domain-centric*: administrators typically start from process domains (subject types) and enumerate which object types they may access. This model works well for system-wide hardening, but it is a poor fit for security vendors who typically care about protecting a small set of high-value resources against 'everyone else'. Expressing such intent with process-centric rules often means touching many domains and types, and it is difficult to ensure that all untrusted processes are covered.

To address these issues, `USEC` adopts a *resource-centric* configuration model. Instead of starting from processes and asking "what may this process access?", we start from resources and ask "who may touch this particular resource, and how?". For example, to secure files, each rule is anchored at a file path and lists which users or applications may read, write, or execute that file. `USEC` extends this idea from files to other kernel resources: a policy rule is always attached to a *resource descriptor* (e.g., a file path prefix, a device node, a mount point, or a socket endpoint), and specifies the set of principals and operations that are allowed on that resource. Principals are represented by `USEC`'s identity bitmap, so that a rule can simply say "allow processes with bits `AGENT` or `SECADMIN` to write this resource", while all other processes are implicitly denied.

Concretely, the overall design of the resource-centric policy model in `USEC` is shown in Figure 4. Each JSON entry is anchored at a specific resource and records its metadata, resource-specific attributes (`attrs`), and the set of operations that a policy may grant on this resource. These JSON policies are then compiled into an identity-bitmap layout and a compact rule table, which the kernel enforces through LSM hooks. For each access, `USEC` builds a resource descriptor and looks it up in the rule table. Only if a matching rule exists does `USEC` perform the bitmap and operation checks.

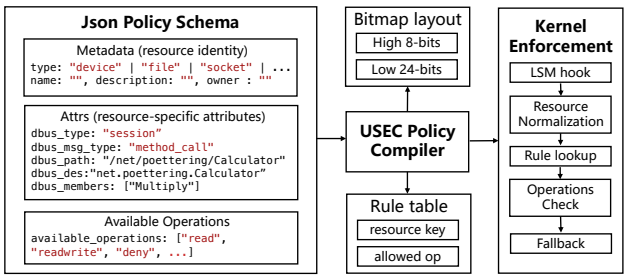


Figure 4: Overall design of resource-centric config policy.

A key property of this design is that `USEC` performs *matching only on resources that have been explicitly declared as protected*. At enforcement time, LSM hooks extract a canonical descriptor for the accessed resource (e.g., the absolute file path, device identifier, or socket tuple) and look it up in the `USEC` rule tables. If there is at least one matching rule, `USEC` applies the corresponding bitmap and operation checks to decide allow or deny. If there is no matching rule, the resource is treated as unprotected by `USEC` and the access decision falls back to the underlying DAC (and optional audit), with no additional policy lookup. In other words, resources that vendors never mention in JSON incur effectively zero `USEC` overhead and are not accidentally constrained by `USEC` policies.

This resource-centric, demand-driven design has two practical benefits. First, it aligns configuration with how vendors and administrators naturally think: they only need to write rules for the relatively small set of sensitive resources they care about, rather than modifying many process domains or global type lattices. Second, it ensures that enabling `USEC` does not impose a blanket policy cost on the entire system: only accesses that touch protected resources pay the price of policy matching, while all other accesses remain as cheap and compatible as in a DAC-only system. The next subsection describes how these declared resource-oriented rules are translated into the capability set that drives retained-hook resolution and selective enforcement in the kernel. At compile time, the declared resource-oriented rules are translated into the capability set that drives retained-hook resolution and selective enforcement.

4.3 Demand-Driven MAC Enforcement

A central design goal in `USEC` is to decouple mandatory access control (MAC) enforcement from the conventional assumption that all kernel objects and all control paths must always be mediated. Traditional MAC systems such as SELinux rely on broad, always-on mediation: they install a dense set of hooks across file-system, process, IPC, and networking paths, and route relevant accesses into a common policy engine. While this design provides broad coverage, it also incurs unnecessary cost in the deployments we target, where many accesses unrelated to the declared protection goal still pay MAC dispatch, cache lookup, and synchronization overhead.

Formally, given a declared capability set $\mathcal{C}$, `USEC` does not attempt to mediate all kernel objects in O. Instead, it enforces protection only for the protected resource sets $\{O_c\}_{c\in\mathcal{C}}$ and operation sets $\{\mathcal{A}_c\}_{c\in\mathcal{C}}$ induced by the active capabilities. Intuitively, O_c captures *what* must be protected, while $\mathcal{A}_c$ captures *how* those resources may be accessed or modified. `USEC` then derives a retained hook set that mediates accesses to $\bigcup_{c\in\mathcal{C}} O_c$ through $\bigcup_{c\in\mathcal{C}} \mathcal{A}_c$, while bypassing hooks that are irrelevant to the declared capability set. Figure 5 shows the basic enforcement pipeline.

Capability-to-Hook Resolution: Given the active capability set $\mathcal{C}$, `USEC` resolves each capability $c \in \mathcal{C}$ through

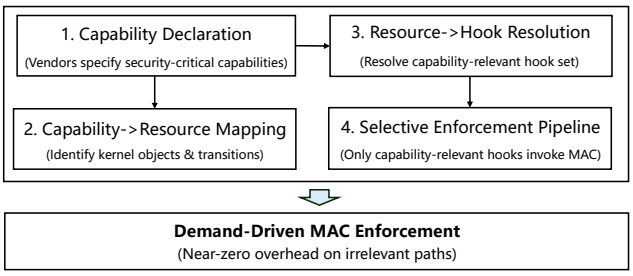


Figure 5: The Basic Pipeline of USEC's MAC Enforcement.

a compiler-driven translation pipeline rather than by manually selecting hooks. Concretely, policies are written in a compact JSON format and compiled into (i) TE permissions for the policy engine and (ii) a hook set for selective enforcement. For each declared capability, the policy compiler queries a capability dictionary that maps the capability to its corresponding TE permissions and dependent mediation hooks. For example, `FILE_READ` expands to permissions such as `read`, `open`, and `getattr`, and to hooks such as `file_open`, `inode_permission`, `mmap_file`, `file_read`, and `file_close`; similarly, `SOCK_CONNECT` maps to `socket_connect`, and `PROC_FORK` maps to `task_create`. Thus, USEC resolves a high-level capability into both the policy semantics and the concrete kernel mediation points required to enforce it.

After expanding all capabilities, USEC computes the retained-hook set by taking the union of the hook sets associated with all declared capabilities. This union is materialized as a compact global bitmap, where each bit corresponds to one hook in the target kernel's hook inventory. At runtime, only hooks whose bits are set are allowed to enter the policy path; all other hooks return immediately. This bitmap-based formulation makes the enforcement boundary explicit: hook pruning is not an ad hoc optimization, but the direct result of capability expansion and hook aggregation. A hook can be elided only if it does not belong to the hook set required by any declared capability.

The mapping is kernel-aware rather than universal. The same high-level capability may be realized through different mediation points depending on the kernel version and enabled subsystems. USEC therefore maintains a per-kernel hook inventory and checks version compatibility between the compiler and the kernel before materializing the bitmap. If the exported kernel hook version does not match the compiler's expected version, policy compilation is rejected rather than silently using a stale mapping. In this way, correctness comes from capability-dictionary expansion, kernel-specific hook binding, and version matching, while representative workloads are used only to validate deployment coverage and identify redundant hot-path hooks after the required enforcement surface has already been established.

Selective Enforcement Pipeline: At runtime, USEC replaces the monolithic always-on MAC path with a selective enforcement pipeline. Each installed hook first performs a constant-time membership test against the retained-hook bitmap. If the hook is not enabled for the active capability set, it returns immediately without entering the MAC engine. Otherwise, USEC invokes `check_usec_perm()`, which consults the permission cache and policy database to complete normal policy evaluation. Thus, USEC preserves the familiar hook-based mediation structure, but avoids dispatching irrelevant accesses into the policy engine.

Table 1: An example of capability-to-hook resolution in USEC.

Capability	Protected Operations	Retained Hooks
File-write-related integrity protection	write, mmap write, metadata update, link/rename	`file_open`, `file_permission`, `mmap_file`, `inode_setattr`, `inode_link`, `inode_rename`

This design differs from SELinux's uniform enforcement model, where a broad hook set remains active regardless of whether a deployment actually protects the corresponding resources. In USEC, enforcement cost scales with the declared protection scope. A deployment that only protects sensitive files and process attributes need not pay for mediation on unrelated network or device paths, while a deployment that enables socket or device control retains precisely those hooks.

Table 1 illustrates a concrete capability-to-hook resolution example. Consider a deployment that protects the integrity of sensitive files through file-write-related capabilities. During compilation, USEC expands the declared file capabilities into the operations that may modify file contents or metadata, or otherwise change the authority over the same object, including direct writes, memory-mapped writes, metadata updates, and link/rename operations. It then resolves these operations to the concrete hooks that mediate them on the target kernel. For example, `file_open` and `file_permission` cover ordinary write paths, `mmap_file` covers memory-mapped access, `inode_setattr` covers metadata changes, and `inode_link` and `inode_rename` cover aliasing-related updates. Hooks outside these paths, such as `task_kill` or `socket_sendmsg`, are not retained because they do not affect the declared file-integrity capability. This example illustrates that the retained-hook set is derived from capability expansion rather than manually selected, and that enforcement remains closed over the relevant access paths without paying the system-wide cost of full mediation.

Operational Benefits: Demand-driven enforcement provides three practical benefits. First, the enforcement surface grows with the declared security intent rather than with the full kernel mediation surface. Second, it gives operators a principled way to avoid the system-wide cost of full mediation without manually disabling checks in an ad hoc manner. Third, it supports heterogeneous deployments: when the pro-

tected capability set changes, when USEC is ported to a new kernel family, or when new capabilities are introduced, USEC simply re-instantiates the capability-to-hook mapping and recomputes the retained-hook bitmap for the target environment. This makes USEC particularly suitable for production deployments where usability and performance matter as much as access-control strength.

4.4 Compatibility-Oriented Security Interface

Beyond configuration complexity and runtime overhead, a key goal of USEC is to avoid the compatibility problems that often appear when SELinux is enabled in real deployments. Therefore, USEC needs to meet two compatibility requirements. First, enforcement is resource centric and opt in, with a permissive mode that lets operators validate policies without triggering SELinux style breakage. Second, the implementation is fully stackable with SELinux and other LSMs, keeping policy and state separate so that enabling USEC does not interfere with existing MAC modules.

Preventing Hook-Level Incompatibility with SELinux. In conventional deployments, SELinux and other in-kernel security modules are wired directly into LSM hook sites. Once these hooks are compiled in, third-party security logic can only be added by patching and rebuilding the kernel, and any mismatch in hook prototypes or call order may cause subtle breakage. In practice, enabling SELinux often changes which hooks run and in what order, so applications that relied on a particular hook behavior or on a vendor kernel patch may start to fail as soon as SELinux is turned on.

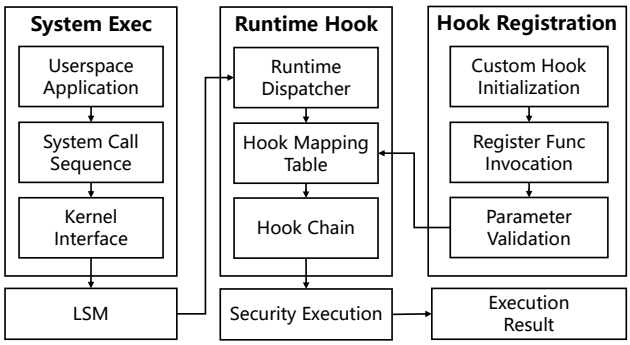


Figure 6: Workflow of USEC's Security Interface.

As shown in Figure 6, USEC inserts a security interface between normal system execution and vendor code. Once a userspace application issues a system call, which traverses the kernel interface and reaches the LSM hook. Instead of invoking third-party modules directly, the LSM hands control to USEC's runtime dispatcher, which consults a hook mapping table and, if a matching entry exists, runs the corresponding hook chain to produce a security decision. When no mapping exists, the dispatcher returns immediately and the LSM continues with its default behavior, so enabling USEC does not alter how SELinux or other in-kernel modules see the call. Also, to maintain the mapping table, vendors initialize their custom hooks and call a registration function in USEC, which validates the argument list, return type, and function pointer against the original LSM prototype before inserting any entry into the mapping table. Hooks that fail validation are rejected and never affect runtime execution.

Compatibility with SELinux and other MACs. Beyond improving compatibility for individual applications, USEC is designed to run alongside existing MAC modules, such as SELinux. At the LSM level, USEC follows the upstream stacking model and uses the standard blob-based security domains for tasks, inodes, and files. It tracks its own state in `struct usec_state`, which points to an independent usec_policy, upolicydb, usec_uavc, and usidtab. This mirrors SELinux's separation of `selinux_state`, policydb, selinux_avc, and sidtab, but all data structures and permission-checking routines (e.g., `check_usec_perm()`) operate on USEC's own state. Loading or updating USEC policies therefore does not affect SELinux's policy database or AVC cache, and the reverse is also true.

SELinux	USEC
`struct selinux_state`	`struct usec_state`
`struct selinux_avc`	`struct usec_uavc`
`struct policydb`	`struct upolicydb`
`struct selinux_policy`	`struct usec_policy`
`struct sidtab`	`struct usidtab`
`struct av_decision`	`struct uav_decision`
`selinux_avc_init()`	`usec_avc_init()`
`avc_has_perm_noaudit()`	`check_usec_perm_noaudit()`
`avc_has_perm()`	`check_usec_perm()`

Table 2: USEC's policy/AVC layer mirrors SELinux with a separate set of data structures and interfaces.

Table 2 summarizes this mapping relations between SELinux and USEC at the policy and AVC layer. Each SELinux structure or interface that manages policy state, SIDs, and cached decisions has a corresponding USEC structure that serves the same role but is kept separate in memory, which makes coexistence possible without entangling internal state.

For hooks that are already fully stackable under the blobs mechanism, USEC simply registers as another LSM, where other MACs, including SELinux in enforcing mode, run as usual. For legacy mount-related hooks that still share fields such as `fs_context.security`, USEC avoids conflicts by not storing its options in that field; instead, it records its mount security options in a private in-kernel list keyed by the caller's task, and applies them when the mount completes. This allows USEC to coexist with SELinux even on mount paths.

Finally, USEC reuses the SELinux policy model and on-disk format while keeping its runtime policy state separate. Its policy layer ports the relevant SELinux code into renamed structures (for example `upolicydb` and `usidtab`) and trims features that fall outside USEC's narrower scope. Labels follow

SELinux conventions as well, which means systems that already deploy SELinux can introduce USEC without changing the on-disk labeling scheme, and the same extended attributes remain meaningful to both modules. Taken together, these choices let operators incrementally deploy USEC on top of existing SELinux configurations, using USEC's opt-in rules and permissive mode to harden selected resources while preserving compatibility with existing SELinux policies.

5 Implementation

USEC is implemented as a Linux security extension that integrates with the LSM framework and provides a lightweight, capability-driven access control path. It consists of five core components. (1) User space provides policy management via usecd and libusec, which generate and load capability policies. (2) usecfs is a kernel interface that transfers policies and configuration data into the kernel at runtime. (3) LSM hooks serve as mediation points for kernel operations such as file access, process control, and socket communication. (4) UAVC (USEC-specific AVC Access Vector Cache) caches recent authorization decisions to avoid repeated policy lookups on hot paths. (5) the policy engine evaluates access requests against capability policies stored in the kernel.

Enforcement Path. At runtime, each LSM hook first checks whether it belongs to the retained hook set derived from the active capabilities. If not, it returns immediately through the bypass path. Otherwise, the request is forwarded to the UAVC cache; on a cache miss, the policy engine evaluates the request and updates the cache. This design preserves the standard hook-based mediation model while avoiding unnecessary policy invocations on irrelevant paths.

Integration with Linux. USEC is implemented as a loadable LSM module and is compatible with LSM stacking. It reuses existing kernel security infrastructure, including security blobs for labeling and hook dispatch mechanisms, and does not require intrusive changes to core kernel subsystems. Policies are loaded dynamically from user space via usecfs, allowing deployment without modifying application binaries.

Automation vs. Manual Effort. Most of USEC's operation is automated. Policy loading, hook dispatch, cache lookup, and enforcement are handled entirely within the system. The only manual component is the maintenance of capability-to-hook bindings for different kernel versions and configurations. This process is lightweight and localized, as it updates only the mappings required for affected capabilities rather than the entire enforcement pipeline.

Implementation Size. USEC adds 19,223 lines of kernel code, including 18,765 lines of new implementation and 458 lines of conditionally compiled code under #ifdef CONFIG_SECURITY_USEC. The user-space components add 63,189 lines of code in total, including 37,787 lines in permission-manager, 5,333 lines in usecpolicy, 17,524 lines in libusec, and 2,545 lines in libsemanage. Overall, USEC consists of 82,412 lines of new code across kernel and user space.

6 Evaluation

6.1 Policy Complexity Reduction

In this section, we present a comparative case study demonstrating how USEC reduces the complexity of expressing access control policies compared to existing Linux MAC systems. We use two representative scenarios: 1) camera device permission control and 2) ATSPI/DBus interface management, to evaluate expressiveness, extensibility, and maintainability.

6.1.1 Camera Access Control

To experimentally evaluate policy complexity, we compare how SELinux, AppArmor, and USEC implement the same task: restricting third-party applications from accessing the system camera device. Although the enforcement goal is identical, the mechanisms differ fundamentally, resulting in different policy sizes and maintenance costs.

```
attribute deepin_app_domain;
attribute uos_controlled_resource_type;
attribute uos_camera_resource_type;

allow deepin_app_domain { file_type -
       uos_controlled_resource_type }:
       chr_file all_chr_file_perms;

interface(`uos_camera_resource_type_set',`
       typeattribute $1 uos_camera_resource_type;')

tunable_policy(`allow_app_access_camera',`
       allow app_t uos_camera_resource_type:chr_file {
       getattr ioctl map open read };')
```

Figure 7: SELinux attribute and allow-rule construction

In SELinux, the camera device ultimately corresponds to the kernel label v4l_device_t. However, SELinux cannot directly express the concept of a 'camera resource'. Instead, the policy writer must distribute this concept across multiple policy subsystems. In our reproduced configuration, three new attributes (deepin_app_domain, uos_controlled_resource_type, and uos_camera_resource_type) must be created before any access control rule can be written. Additional attributes such as deepin_app_private_file_type are required because SELinux decomposes resources into file-system classes and type categories rather than treating them as logical entities. Once these attributes exist, access control is implemented procedurally. A representative fragment from our SELinux experiment is shown in Figure 7.

To complete the policy, the camera device label v4l_device_t must be registered inside the kernel module, and this interface must be exposed to third-party modules

through a dedicated `.if` file. These fragments occupy multiple files (`uos_perm_control.te`, `uos_perm_control.if`, `devices.te`). Even the minimal working configuration exceeds **300 lines** of SELinux policy code. AppArmor is simpler than SELinux in this case because it can constrain access using pathname-oriented rules rather than type attributes and inter-module bindings. However, it still does not directly capture the higher-level notion of a protected "camera resource" and remains tied to concrete device nodes and profile rules.

In `USEC`, we repeat the same experiment using the resource-centric model. Instead of decomposing the concept of a camera across multiple subsystems, `USEC` treats the camera as a first-class resource described declaratively in a single JSON object as shown in Figure 8.

```
{  "name": "camera",
   "type": "object",
   "subtype": "device",
   "attrs": { "devtype": "camera" },
   "available_operations": ["read",
   "readwrite", "deny"],
   "version": "1.0"
}
```

(a) USEC resource definition for a camera

```
{  "policies": [{
   "subject" : { "name":"com.example.app" },
   "objects": [{
       "object": "camera",
       "operations": ["read"]
   }]
 }]
}
```

(b) USEC policy granting read access

Figure 8: USEC's configuration for camera resource control.

These two JSON documents, together fewer than 20 lines, capture the entire access control logic. No attribute propagation, no macro definition, no kernel-module modification, and no policy recompilation are required. The fundamental reason for this reduction is that `USEC` is *resource-centric*. A resource is expressed directly at the level developers reason about (e.g., 'camera device'), while SELinux requires mapping this concept into filesystem classes, device types, type attributes, Boolean tunables, and module interfaces, and AppArmor still relies on concrete pathname-based profile rules. As a result, although all three systems can enforce the same high-level restriction, the SELinux implementation spans hundreds of lines across several policy modules, AppArmor remains bound to low-level device naming, whereas the `USEC` configuration is fully contained in a small declarative specification. This experiment demonstrates that `USEC` achieves the same enforcement outcomes at a fraction of the configuration complexity, providing better maintainability and extensibility.

6.1.2 ATSPI/DBus Interface Control

To further evaluate expressiveness and policy complexity, we study a second scenario involving the AT-SPI accessibility framework. AT-SPI exposes dozens of DBus interfaces such as `EditableText`, `Action`, `Component`, and `DeviceEventController`, each containing numerous methods including both read-oriented queries (e.g., `GetExtents`, `CopyText`) and security-sensitive write operations (e.g., `InsertText`, `GenerateKeyboardEvent`).

```
attribute uos_controlled_resource_type;
attribute uos_atspi_write_type;

# associate a DBus interface with a custom controlled type
typeattribute atspi_EditText_t uos_atspi_write_type;

# allow or deny DBus method calls via dbus class rules
allow app_t uos_atspi_write_type:dbus { send_msg };
```

Figure 9: SELinux-style DBus mediation for AT-SPI methods

In SELinux, implementing such control is challenging because SELinux has no direct notion of 'DBus interface' or 'DBus method'. Instead, DBus mediation must be reconstructed indirectly through a combination of types, attributes, and allow rules governing the `dbus` class. Each method to be restricted must ultimately be represented through the DBus message type, the object path, the interface string, and possibly custom DBus type attributes. This causes the semantics of a single AT-SPI interface to be fragmented across multiple policy constructs that must be coordinated manually. The figure 9 below shows a representative SELinux fragment.

```
{ "name": "ATSPI-WRITE",
  "type": "object",
  "subtype": "dbus",
  "attrs": {
    "bus_type": "accessibility",
    "dbus_msg_type": "method_call",
    "dbus_interface": "org.a11y.atspi.EditableText",
    "dbus_members": ["InsertText", "CutText", "DeleteText"],
    "operations": ["read", "readwrite", "deny"]
  }
}
```

(a) USEC definition of AT-SPI write interfaces

```
{ "policies": [{
  "subject": { "name": "dbus-send" },
  "objects": [{
    "object": "ATSPI-WRITE",
    "operations": ["read"]
  }]
}]}
```

(b) USEC granting read-only AT-SPI access

Figure 10: USEC's configuration for AT-SPI interface control.

Extending this pattern to dozens of AT-SPI methods requires repeating the same steps: creating new types or attributes, binding them to the correct interface strings, and ensuring that the correct DBus object paths are labeled. The resulting policy spans multiple files and hundreds of lines. Again, this complexity arises not from the difficulty of the se-

curity requirement, but from SELinux's type-centric architecture: DBus interfaces must be decomposed and re-assembled manually using low-level primitives. AppArmor is also limited here: although it is lighter-weight than SELinux, it does not naturally express method-level DBus semantics as first-class policy objects, and fine-grained interface control still requires indirect, string-oriented rule construction.

In contrast, `USEC` models DBus interfaces directly as first-class resources. A single JSON object describes the bus type, message type, interface name, member list, and allowed operations. The semantics remain intact without any intermediate type constructions. The JSON fragment in Figure 10 defines all write-oriented AT-SPI methods as a controlled object.

The experiment demonstrates that `USEC` enforces fine-grained method-level control over DBus interfaces without requiring SELinux-style decomposition into types, attributes, macros, or inter-module bindings, and without AppArmor-style indirect string-oriented policy construction. In scenarios like AT-SPI, where interfaces evolve frequently and clients dynamically issue method calls, `USEC`'s configuration provides better simplicity and maintainability.

6.2 Evaluation on Overhead

To evaluate the runtime overhead introduced by `USEC`'s MAC enforcement, we use UnixBench 5.1.3 [18] as a general-purpose system benchmark, and further supplement it with two workloads that exercise security-relevant kernel paths more directly: Filebench, a filesystem-intensive benchmark suite, and Nginx, a network-facing web-server workload. Compared with UnixBench, these two workloads stress the VFS, inode, file-permission, socket, and IPC paths that are central to MAC enforcement, and therefore provide a more meaningful view of MAC overhead in practice. The experiment environment is a HONOR NBLK-WAX9X notebook with an AMD Ryzen 7 3700U processor (4 cores / 8 logical processors). The operating system is UOS Desktop 20 Professional, running Linux kernel version 4.19.0-amd64-desktop, and the main memory is 16GB DDR4. The storage device is a 512 GB Toshiba KXG60ZNV512G SSD. All experiments are repeated 10 times, and the tables report the average, standard deviation, and volatility.

We evaluate four different tools: (1) SELinux, representing a traditional full-coverage MAC framework where all LSM hooks remain active; (2) USEC, our lightweight capability-driven MAC module; (3) AppArmor, another widely deployed MAC system with pathname-based mediation; and (4) LSM Disabled, where all security modules are removed, serving as the zero-overhead baseline. To ensure a fair comparison, we configure USEC, SELinux, and AppArmor to enforce same protection goals for each evaluated workload. In particular, for a given benchmark, all MAC systems are configured to protect the same class of resources and to mediate the same high-level operations required by the workload; the only difference is how each system realizes that protection internally. USEC uses its capability-driven policy model and retained-hook bitmap, whereas SELinux and AppArmor use their native policy mechanisms. The LSM-disabled baseline removes MAC enforcement entirely and therefore differs only in that no access-control checks are performed.

UnixBench as a general-purpose baseline. We retain UnixBench as a broad system benchmark, but use it only as a general-purpose reference rather than the main evidence for MAC overhead. As shown in Table 3, based on 10 runs, SELinux lowers the average System Benchmarks Index Score from 3,993.15 in the LSM-disabled baseline to 3,681.53, corresponding to a 7.80% slowdown, while USEC and AppArmor incur 2.96% and 3.21% overhead, respectively. This result is expected because several UnixBench components are CPU-bound and exercise few MAC-relevant kernel paths, making them relatively insensitive to mediation overhead.

UnixBench reveals that CPU-bound components such as Dhrystone and Whetstone show only minor variation across configurations, indicating that these workloads spend most of their time in user mode and trigger relatively few MAC-relevant kernel checks. By contrast, filesystem- and IPC-related components show a clearer separation, where SELinux imposes higher overhead than the lighter-weight configurations. This suggests that broad system benchmarks are useful for establishing a baseline, but they can understate the cost of always-on MAC mediation because not all subtests heavily exercise security-sensitive kernel paths.

Filebench: filesystem-intensive overhead. We next evaluate filesystem-intensive workloads using Filebench. These workloads repeatedly exercise file creation, open, read/write, metadata update, namespace traversal, and fsync operations, all of which frequently traverse the hooks targeted by MAC enforcement. Table 4 reports the results on five representative Filebench workloads.

Overall, USEC remains closer to the LSM-disabled baseline than SELinux across the evaluated Filebench workloads. On *webserver*, USEC achieves 88,908.51 ops/s, compared with 91,029.30 ops/s in the baseline, while SELinux achieves 84,359.23 ops/s. On *fileserver*, USEC reaches 56,510.22 ops/s, again closer to the baseline (58,267.98 ops/s) than SELinux (53,391.44 ops/s). On the fsync-intensive *varmail* workload, USEC achieves 27,000.82 ops/s, which is also closer to the baseline (27,643.02 ops/s) than either SELinux (24,334.93 ops/s) or AppArmor (25,715.50 ops/s). These results indicate that USEC generally reduces the performance cost of MAC mediation on common filesystem-intensive paths, even though these operations frequently traverse security-relevant hooks in the VFS and inode layers. At the same time, the Filebench results also show that filesystem-intensive workloads can be affected by factors beyond the MAC mechanism itself. For example, on *random write* and *single stream read*, the relative ordering of USEC and AppArmor is less uniform than on the other workloads, and the LSM-disabled baseline is

Table 3: Selected UnixBench results comparing SELinux, USEC, AppArmor, and the LSM-disabled baseline. Each configuration was run for 10 trials. For each metric, we report the average, standard deviation, and volatility (volatility = standard deviation / average). Higher throughput or composite index values indicate better performance. The bottom row shows the relative performance degradation compared to the LSM-disabled configuration.

Test Item	SELinux			USEC			AppArmor			LSM Disabled		
	Avg.	Stdev.	Volatility	Avg.	Stdev.	Volatility	Avg.	Stdev.	Volatility	Avg.	Stdev.	Volatility
Dhrystone 2 (register variables)	94,902,690.67	1,796,536.09	1.89%	107,993,964.91	4,122,540.63	3.82%	89,417,776.41	135,610.69	0.15%	98,745,652.18	2,982,540.68	3.02%
Double-Precision Whetstone	19,989.02	1,037.92	5.19%	20,965.18	271.35	2.94%	19,674.11	15.22	0.08%	20,781.12	354.07	1.70%
Pipe Throughput	3,558,673.88	9,755.85	0.27%	3,998,685.01	139,447.98	3.49%	4,351,265.53	137,575.96	3.16%	4,645,714.83	5,472.74	0.12%
File Copy (1024 buf / 2000 blocks)	1,356,816.90	4,418.47	0.33%	1,370,818.44	26,770.23	1.95%	1,423,522.21	18,348.76	1.29%	1,457,369.49	6,000.70	0.41%
System Call Overhead	5,975,768.33	5,950.58	0.10%	6,224,448.09	52,963.26	0.85%	6,271,104.63	327,866.08	5.23%	6,341,157.01	10,767.52	0.17%
Process Creation	39,528.68	143.65	0.36%	40,275.54	158.28	0.39%	40,349.44	170.82	0.42%	40,360.75	110.25	0.27%
Shell Scripts	2,871.43	11.97	0.42%	2,980.05	13.95	0.47%	2,926.74	114.94	3.93%	3,004.96	15.09	0.50%
System Benchmarks Index Score	3,681.53	4.45	0.12%	3,875.07	200.36	5.17%	3,864.81	66.14	1.71%	3,993.15	164.12	4.11%
Performance Degradation vs Baseline (LSM Disabled)	7.80%			2.96%			3.21%			0%		

not always far above the MAC configurations. This suggests that filesystem-internal behavior, caching, synchronization, and I/O scheduling effects can have a substantial influence on throughput in these workloads. When aggregating the five Filebench workloads, SELinux incurs a 10.86% performance degradation relative to the LSM-disabled baseline, whereas USEC incurs 6.87%, and AppArmor incurs 9.51%. Overall, these results show that `USEC` substantially narrows the performance gap compared with a traditional full-coverage MAC design on filesystem-intensive workloads.

Nginx: network-facing overhead. We further evaluate a web-server workload using Nginx. This benchmark stresses socket creation, connection handling, request processing, and message transmission, which together exercise the socket-, file-, and IPC-related paths often involved in MAC checks for server applications. As shown in Table 5, USEC achieves 92,185.08 req/s on average, corresponding to 95.68% of the LSM-disabled baseline and incurring only 4.32% throughput overhead. In contrast, SELinux achieves 84,748.05 req/s, corresponding to 87.96% of the baseline, while AppArmor achieves 81,774.77 req/s, corresponding to 84.87% of the baseline and incurring 15.13% throughput overhead.

The larger gap observed here, compared with UnixBench, suggests that realistic server workloads are much more effective at exposing the runtime cost of MAC enforcement because they repeatedly trigger socket-, file-, and IPC-related checks along the request-processing path. The latency results show a similar trend. USEC remains relatively close to the baseline in both median and tail latency, with a P50 latency of 3.569 ms and a P99 latency of 12.140 ms, whereas SELinux increases P50 latency to 3.710 ms and P99 latency to 12.990 ms. AppArmor shows the largest throughput loss and also exhibits higher latency than USEC across the key percentiles, with a P75 latency of 8.333 ms, a P90 latency of 9.526 ms, and a P99 latency of 12.120 ms. This means that the benefit of lightweight mediation is not limited to average throughput, but also extends to request latency behavior, which is particularly important for interactive services. In other words, USEC not only preserves near-baseline throughput, but also avoids the more visible latency inflation observed under heavier mediation mechanisms.he pronounced long-tail latency inflation observed under heavier mediation mechanisms.

Overall, the evaluation shows that traditional full-coverage MAC enforcement incurs limited slowdown on broad CPU-oriented benchmarks, but can introduce more visible overhead once workloads repeatedly traverse filesystem and network paths that are central to MAC enforcement. In contrast, `USEC` remains consistently closer to the LSM-disabled baseline across both Filebench and Nginx, while substantially outperforming SELinux and showing more stable behavior than AppArmor under high-concurrency web serving.

This also clarifies the apparent gap between the 87% slowdown cited in our motivation and the smaller overheads observed in our own evaluation. The 87% number comes from prior work [26], which evaluates SELinux under a much more MAC-intensive setting on Linux v5.3, and should therefore be interpreted as a workload-specific stress-case result rather than a general statement about SELinux overhead in most environment scenarios. Our experiment results show the same qualitative trend across workloads: SELinux incurs only a 7.8% slowdown on the broad, CPU-heavy UnixBench suite, but this grows to 12.04% on the more security-sensitive Nginx workload reported above, where MAC-relevant kernel paths are exercised much more frequently. At the same time, even Nginx remains a realistic service workload rather than an extreme stress case, since substantial execution time is still spent in ordinary request processing and protocol handling outside MAC-dominated paths. In this sense, the 87% result reflects a specific upper-end MAC-intensive scenario, whereas our Filebench and Nginx results capture more representative practical deployments.

6.3 Evaluation on Compatibility

To evaluate compatibility, we focus on two scenarios. First, we consider applications that fail to deploy under SELinux, and show that `USEC` can protect selected resources without breaking them. Second, we start from an existing SELinux policy known to work in production and demonstrate that `USEC` can remain compatible with such policy. We also discuss AppArmor where relevant: although it is often less intrusive than SELinux, it does not provide the same migration path

Table 4: Filebench results comparing SELinux, USEC, AppArmor, and the LSM-disabled baseline. Each configuration was run for 10 trials. For each metric, we report the average, standard deviation, and volatility (volatility = standard deviation / average). Higher throughput values indicate better performance. The bottom row shows the relative performance degradation compared to the LSM-disabled baseline.

Test Item	SELinux			USEC			AppArmor			LSM Disabled		
	Avg.	Stdev.	Volatility	Avg.	Stdev.	Volatility	Avg.	Stdev.	Volatility	Avg.	Stdev.	Volatility
Webserver (ops/s)	84,359.23	7,119.02	8.44%	88,908.51	8,806.86	9.91%	85,161.08	1,969.54	2.31%	91,029.30	11,305.62	12.42%
Fileserver (ops/s)	53,391.44	7,820.46	14.65%	56,510.22	1,550.88	2.74%	55,733.03	1,139.10	2.04%	58,267.98	4,239.23	7.28%
Varmail (ops/s)	24,334.93	301.05	1.24%	27,000.82	1,030.29	3.82%	25,715.50	926.48	3.60%	27,643.02	1,923.49	6.96%
Random Write (ops/s)	52,735.83	5,179.47	9.82%	45,609.51	2,392.82	5.25%	53,506.79	701.12	1.31%	60,780.45	8,376.83	13.78%
Single Stream Read (ops/s)	4,176.73	76,12	1.82%	4,739.79	121,72	2.57%	3,962.66	51.00	1.29%	4,822.51	140,23	2.91%
Performance Degradation vs Baseline (LSM Disabled)	10.86%			6.87%			9.51%			0%		

Table 5: Nginx results comparing SELinux, USEC, AppArmor, and the LSM-disabled baseline. Each configuration was run for 10 trials. For each metric, we report the average, standard deviation, and volatility (volatility = standard deviation / average). Higher throughput and lower latency indicate better performance.

Test Item	SELinux			USEC			AppArmor			LSM Disabled		
	Avg.	Stdev.	Volatility	Avg.	Stdev.	Volatility	Avg.	Stdev.	Volatility	Avg.	Stdev.	Volatility
Request Throughput (req/s)	84,748.05	252.99	0.30%	92,185.08	216.87	0.24%	81,774.77	250.14	0.31%	96,350.20	174.49	0.18%
P50 Latency (ms)	3.710	0.0067	0.18%	3.569	0.0032	0.09%	3.581	0.0032	0.09%	3.501	0.0032	0.09%
P75 Latency (ms)	5.949	0.0292	0.49%	5.316	0.0327	0.62%	8.333	0.0356	0.43%	4.988	0.0301	0.60%
P90 Latency (ms)	7.706	0.0117	0.15%	7.524	0.0126	0.17%	9.526	0.0151	0.16%	7.440	0.0141	0.19%
P99 Latency (ms)	12.990	0.1595	1.23%	12.140	0.1350	1.11%	12.120	0.1549	1.28%	11.820	0.0789	0.67%
Performance Degradation vs Baseline (LSM Disabled)	12.04%			4.32%			15.13%			0%		

from existing SELinux policies nor the same resource-centric abstraction used by `USEC`.

Case 1: Mitigating SELinux-induced failure. Many applications are built assuming pure DAC semantics [11, 14]. When SELinux's path- and type-based rules are enabled without a well-tuned policy, installers may fail to update their own files, and services may lose access to devices or D-Bus endpoints. AppArmor is often more permissive in practice because it relies on pathname-oriented profiles and may therefore avoid some SELinux-specific deployment failures. However, this does not eliminate the underlying policy-engineering burden when administrators still need to align application identifiers, resources, and interface permissions. In this case, `USEC` uses templates and D-Bus-based configuration to derive SELinux-compatible rules from a compact app identifier, protecting private files and D-Bus interfaces while avoiding failures common with hand-written SELinux policies.

```
1  policy_template_t g_policy_template[MAX_ID_TYPE] = {
2      [APPID_TYPE] = {
3      .policy_template = "type deepin_app_@n_t;\n"
4          "deepin app domain_set(deepin_app_@n_t)\n"
5          "domain unkillable(deepin_app_@n_t)\n"
6          "type deepin_app_@n_file_t;\n"
7          "deepin_app_private_file_type_set(deepin_app_@n_file_t)\n"
8          "allow deepin_app_@n_t deepin_app_@n_file_t:dir
9          _file_class_set ~{ relabelfrom relabelto };\n"
10         "deepin_access_all_sensitive_resource(deepin_app_@n)\n\n",
11     .template_escape_cnt = 0
12     },
13 }
```

Figure 11: A template that turns an app identifier into a SELinux-compatible policy fragment.

As shown in Figure 11, we first declare the global template table and select the `APPID_TYPE` entry(lines 1–2), and encode the SELinux rules with the placeholder `@n` (lines 3–10). Instead of manually writing all rules for every program, the administrator only supplies an app identifier (e.g., the D-Bus name) over `USEC`'s management D-Bus; the policy generator substitutes `@n` with that identifier, emits SELinux-compatible rules, and loads them into `upolicydb` and `usidtab`. For the `net.poettering.Calculator` test service, we bind its identifier to the `APPID_TYPE` template and describe the calculator object and subjects (`d-feet`, `busctl`, `dbus-send`) in JSON, choosing operations such as `readwrite` or `deny`. This keeps the enforcement semantics aligned with SELinux while removing most manual policy editing that previously caused SELinux-induced application failures. Compared with AppArmor, the benefit is different: rather than relying on simpler pathname matching alone, `USEC` allows the administrator to start from a compact resource-oriented description while still preserving SELinux-compatible policy semantics.

Case 2: Matching SELinux protection and compatibility. To evaluate whether `USEC` can reuse existing SELinux policies without losing protection or compatibility, we install the upstream SELinux default policy and load a subset of its rules into `USEC`. This case is primarily a comparison against SELinux, since AppArmor does not share SELinux's policy representation, label model, or binary policy format, and therefore does not offer an analogous policy-migration path.

In detail, we migrate a SELinux policy into `USEC`'s policy engine, as shown in Figure 12. First, we place `USEC` in permissive mode via the system security manager (`DisableSecurity`), so that policy changes do not disrupt running applications (lines 1-6). Next, we copy the compiled SELinux policy `usec_test.pp` from

```
# Put USEC in permissive mode during migration
$ busctl call \
org.deepin.PermissionManager \
/org/deepin/PermissionManager/PolicyManager \
org.deepin.PermissionManager.PolicyManager \
DisableSecurity

# Copy the SELinux policy fragment into USEC 's policy set
$ cp /etc/selinux/targeted/policy/usec_test.pp \
/etc/usec/policy/usec_test.pp

# Load policy into USEC
$ usec-policy-load /etc/usec/policy/usec_test.pp

# Re-enable USEC enforcement via the system security manager
$ busctl call \
org.deepin.PermissionManager \
/org/deepin/PermissionManager/PolicyManager \
org.deepin.PermissionManager.PolicyManager \
EnableSecurity
```

Figure 12: Migrating a SELinux policy fragment into USEC.

/etc/selinux/targeted/policy/ into USEC's policy directory under /etc/usec/policy/, and load it with usec-policy-load (lines 8-13). At load time, USEC parses the SELinux binary policy and populates its own upolicydb, usidtab, and usec_uavc, which mirror SELinux's policydb, sidtab, and AVC but live in a separate usec_state. Finally, we re-enable enforcement with the EnableSecurity call, so subsequent accesses are checked by USEC (lines 15-20).

USEC reuses the SELinux policy format and label syntax, so policy fills upolicydb and usidtab with the same rules and security identifiers that SELinux has, while on-disk labels stay unchanged. At the same time, this state lives in usec_state and is wired to LSM hooks through the blobs mechanism, so USEC runs alongside SELinux without sharing internal data. For objects covered by the imported policy, both modules see the same subject–object pairs and reach the same allow or deny result; for other objects, USEC returns early, and the request is handled by DAC and any SELinux rules. In practice, loading a SELinux policy into USEC preserves SELinux-level protection on protected resources and does not introduce extra denials when the two MACs are stacked. This compatibility is complementary to AppArmor: AppArmor provides lightweight confinement, but does not support direct reuse of existing SELinux policies as USEC does.

7 Lessons Learned

Lesson 1: Hook retention must be instantiated for the target kernel and software stack. Although USEC's policy model is conceptually portable, practical hook pruning is not a one-time universal mapping. Our initial prototype assumed that a single generic capability-to-hook mapping would be sufficient across deployments. In practice, this assumption proved too coarse. The same high-level capability may be realized through different mediation points depending on the kernel version, enabled subsystems, and userspace software stack. As a result, a retained hook set that is correct and efficient on one product may be suboptimal, or even require additional bindings, on another.

This led us to treat capability-to-hook resolution as a compiler-driven, kernel-aware construction step. In USEC, a declared policy is first translated into capabilities, each capability is then expanded through a capability dictionary into TE permissions and dependent hooks, and the final retained-hook set is materialized as a compact bitmap for the target kernel. This means that hook retention is derived systematically from the declared security intent rather than from ad hoc profiling or manual pruning. To keep this process sound across environments, the compiler and the kernel must also agree on the hook inventory; when the exported hook version changes, the mapping must be re-instantiated rather than silently reused. Representative workloads still matter, but for narrower purposes than we initially expected. First, they validate that the retained set covers the control paths exercised in the intended deployment. Second, they help identify redundant mediation points on hot paths once the required capability coverage has already been established. In other words, profiling improves performance, but it does not determine which security-relevant transitions may remain unchecked.

This distinction proved important in deployment. On some desktop builds, the dominant retained hooks were concentrated in file-system paths such as file_open and inode_permission. On appliance-style systems with heavier network or device activity, a larger fraction of the retained cost came from socket- and device-related hooks. The lesson is not that USEC should keep only "hot" hooks; rather, it must first retain all hooks required by the declared capabilities on the target kernel, and only then optimize away redundant mediation points that unnecessarily burden hot paths.

Lesson 2: Deployability depends on how small the enforcement surface can be made in practice. Although USEC is motivated by strong access-control semantics, our industrial deployments repeatedly showed that deployability depends at least as much on efficiency as on policy expressiveness. System integrators rarely reject a security mechanism because they disagree with its protection goal; they reject it because the runtime cost is visible on latency-critical paths or because the operational workflow becomes too complex.

This was especially evident during early rollouts. Even after replacing SELinux-style broad mediation with capability-driven selective enforcement, the initial retained-hook set was not always performance-optimal for deployed products. Real workloads exposed concentration points that were only weakly reflected in synthetic testing. On desktop systems, reducing redundant checks on frequent file-system paths improved responsiveness. On appliance-style systems, the main costs often shifted to networking and device paths, where a retained set that worked well elsewhere could still introduce noticeable latency under sustained packet or I/O activity. These experiences reinforced a practical rule: correctness

must be established first through capability expansion and kernel-aware binding, but deployability depends on how aggressively the implementation removes unnecessary enforcement from the paths that dominate real workloads. More broadly, this experience shaped USEC's overall design philosophy. A deployable MAC mechanism should not optimize for theoretical generality alone. Instead, it should expose a narrow and understandable security intent, compile that intent into a compact retained-hook set for the target kernel and software stack, and then tune the resulting enforcement surface against representative workloads for the product line. For us, the combination of explicit capability scope, kernel-aware capability-to-hook resolution, bitmap-based selective enforcement, and workload-guided refinement was essential to making USEC acceptable in production.

8 Related Work

8.1 MAC on Operating Systems

Mandatory Access Control has been widely adopted in operating systems, especially via the Linux Security Modules (LSM) framework. Among popular implementations, SELinux [12] offers a rich MAC model with type enforcement, role-based access control, and optionally multilevel security. It labels both subjects and objects and enforces kernel-level decisions, providing strong isolation and expressive policies, but it is also burdened by substantial runtime overhead and policy-management complexity. In contrast, AppArmor [1] uses pathname-based profiles rather than full label+type enforcement, making configuration and deployment simpler, but its expressiveness and isolation guarantees are weaker. Smack [17] is tailored for simplicity and embedded/IoT use cases: trading off many of SELinux's advanced features in favor of reduced complexity and overhead, yet empirical studies indicate that in default settings Smack (and similar lightweight systems) may not prevent attacks as effectively as SELinux. Finally, TOMOYO [16] uses an invocation-history domain model and pathname-based policies, while Yama [5] adds supplementary MAC-like restrictions, both reduce overhead and administrative burden but at the cost of weaker isolation.

USEC fits into a familiar trade-off between expressiveness, overhead, and usability. SELinux offers rich policies at higher complexity and runtime cost, while AppArmor, Smack, and TOMOYO favor simpler, lower-overhead configurations with weaker guarantees. Prior work shows SELinux can block all tested attack stages in embedded firmware, where these lighter systems fail under default settings. USEC builds on this lineage, targeting the low-overhead corner while preserving mandatory enforcement.

8.2 MAC in Cloud/Virtualised Systems

Early work on MAC in virtualised environments focuses on enforcing isolation at the hypervisor or VM level. sHype [13] extends the Xen hypervisor (and later the IBM POWER hypervisor) with a MAC architecture that controls VM–VM and VM–resource interactions using configurable policies such as simple type enforcement and Chinese Wall, with an emphasis on low overhead and strong VM isolation. Xen Security Modules (XSM) [24] provide a general MAC framework inside Xen; its FLASK module [15] brings SELinux-style fine-grained MAC to domains, event channels, and I/O resources, implementing flexible policies over hypercalls and hypervisor objects. Building on these foundations, Win et al. [21] combine hypervisor-level MAC with virtual machine introspection, enforcing STE/CWE-style policies on resource accesses in cloud environments. Shamon [6] generalises sHype into a shared reference monitor that uses TPM-based attestation and security VMs to enforce distributed MAC policies across coalitions of VMs and hosts.

USEC targets a different point in the stack by running inside the guest kernel and protecting in-guest files, devices, and sockets. Like sHype and XSM/FLASK, it is label-based, but enforcement is resource-centric and opt-in, so only named resources trigger MAC checks, and others follow DAC and any in-guest MAC, such as SELinux. Unlike Shamon and VMI-based schemes that use separate security VMs, USEC keeps enforcement local and offers a permissive shadow mode to refine policies before enabling strict enforcement.

9 Conclusion

In this work, we presented USEC, a practical mandatory access control framework co-designed with security vendors to make strong kernel protection both usable and deployable at scale. By introducing a resource-centric policy abstraction, USEC lets administrators express high-level intent directly over the small set of security-critical resources they care about. USEC further reduces runtime costs through demand-driven enforcement, and improves compatibility by preserving DAC semantics for all resources not explicitly opted into protection. USEC can provide SELinux-level protection with significantly fewer rules, substantially lower overhead, and far fewer compatibility issues across real server and desktop workloads. USEC has already been adopted by over 210 commercial security vendors and deployed on over 8,000,000 endpoints, showing that strong, configurable kernel MAC enforcement can be both efficient and practical in modern Linux systems.

Acknowledgements

We would like to express our appreciation to the anonymous reviewers for their valuable and constructive feedback, as well as to the shepherd for the guidance in improving the paper. This research is sponsored in part by the NSFC Program (No. U2441238, 62525207).

References

[1] AppArmor. Apparmor: Linux kernel security module. https://apparmor.net/, 2025. Accessed at December 09, 2025.

[2] Sven Bugiel, Stephan Heuser, and Ahmad-Reza Sadeghi. Flexible and fine-grained mandatory access control on android for diverse security and privacy policies. In *Proceedings of the 22nd USENIX Conference on Security*, SEC'13, page 131–146, USA, 2013. USENIX Association.

[3] danwalsh. Selinux should and does block access to docker socket. https://danwalsh.livejournal.com/78373.html, 2025. Accessed at December 09, 2025.

[4] DELL. Linux sdcs with selinux set to enforcing. https://www.dell.com/support/kbdoc/en-us/000199106/linux-sdcs-with-selinux-set-to-enforcing?lwp=rt, 2024. Accessed at November 18, 2025.

[5] The kernel development community. Yama. https://www.kernel.org/doc/html/latest/admin-guide/LSM/Yama.html, 2025. Accessed at November 18, 2025.

[6] Jonathan M. McCune, Stefan Berger, Ramón Cáceres, Trent Jaeger, and Reiner Sailer. Shamon: A system for distributed mandatory access control. In *Proceedings of the Annual Computer Security Applications Conference (ACSAC)*, pages 23–32, 2006.

[7] Oracle. Oracle advanced cluster file system administrator's guide. https://docs.oracle.com/en/database/oracle/oracle-database/26/acfsg/overview-acfs-advm.html, 2025. Accessed at December 09, 2025.

[8] Oracle. Oracle database] installation guide, security enhanced linux (selinux) is not supported on oracle acfs file systems. https://docs.oracle.com/cd/E11882_01/install.112/e47689.pdf/, 2025. Accessed at December 09, 2025.

[9] Oracle. Release notes for oracle linux 7.2. https://docs.oracle.com/en/operating-systems/oracle-linux/7/relnotes7.2/relnotes-KnownIssues.html#ol7-fixed-known-issues, 2025. Accessed at December 09, 2025.

[10] Sylvia Osborn. Mandatory access control and role-based access control revisited. In *Proceedings of the Second ACM Workshop on Role-Based Access Control*, RBAC '97, page 31–40, New York, NY, USA, 1997. Association for Computing Machinery.

[11] Sylvia Osborn, Ravi Sandhu, and Qamar Munawer. Configuring role-based access control to enforce mandatory and discretionary access control policies. *ACM Trans. Inf. Syst. Secur.*, 3(2):85–106, May 2000.

[12] N Peter Loscocco. Integrating flexible support for security policies into the linux operating system [c]. In *Proceedings of the FREENIX Track: USENIX Annual Technical Conference*, 2001.

[13] Reiner Sailer, Trent Jaeger, Enriquillo Valdez, Ramón Cáceres, Ronald Perez, Stefan Berger, John Linwood Griffin, and Leendert van Doorn. Building a MAC-based security architecture for the Xen open-source hypervisor. In *Proceedings of the 21st Annual Computer Security Applications Conference (ACSAC)*, pages 276–285. IEEE Computer Society, 2005.

[14] Ravi Sandhu and Qamar Munawer. How to do discretionary access control using roles. In *Proceedings of the Third ACM Workshop on Role-Based Access Control*, RBAC '98, page 47–54, New York, NY, USA, 1998. Association for Computing Machinery.

[15] Ray Spencer, Stephen Smalley, Peter Loscocco, Mike Hibler, Dave Andersen, and Jay Lepreau. The flask security architecture: System support for diverse security policies. In *8th USENIX Security Symposium (USENIX Security 99)*, Washington, D.C., August 1999. USENIX Association.

[16] The Linux Kernel Documentation Project. *TOMOYO — The Linux Kernel Documentation*. The Linux Kernel Organization, 2019. Linux kernel.

[17] The Linux Kernel Documentation Project. *Smack — The Simplified Mandatory Access Control Kernel*. The Linux Kernel Organization, 2022. Linux kernel.

[18] UnixBench. Unixbench system microbenchmarks. https://www.usenix.org/legacy/publications/library/proceedings/usenix01/freenix01/full_papers/loscocco/loscocco_html/node15.html, 2025. Accessed at December 09, 2025.

[19] UOS. 8 million installed capacity achieved, tongxin software launches five-year plan for 'billion level scale'. https://www.uniontech.com/news-info/2367.html, 2024. Accessed at December 09, 2025.

[20] UOS. Uapp introduction. https://ecology.chinauos.com/adaptidentification/uapp/index.html, 2025. Accessed at December 09, 2025.

[21] Thu Yein Win, Huaglory Tianfield, and Quentin Mair. Virtualization security combining mandatory access control and virtual machine introspection. In *Proceedings of the 2014 IEEE/ACM 7th International Conference on Utility and Cloud Computing (UCC)*, pages 1004–1009. IEEE, 2014.

[22] Chris Wright, Crispin Cowan, Stephen Smalley, James Morris, and Greg Kroah-Hartman. Linux security modules: General security support for the linux kernel. In *11th USENIX Security Symposium (USENIX Security 02)*, San Francisco, CA, August 2002. USENIX Association.

[23] Glenn Wurster. Security mechanisms and policy for mandatory access control in computer systems. 2010.

[24] Xen Project. Xen security modules: XSM-FLASK. https://wiki.xenproject.org/wiki/Xen_Security_Modules_%3A_XSM-FLASK, 2014. Accessed: 2025-12-09.

[25] Xie XiuQi. The impact of selinx and audit security mechanisms on os performance. https://gitee.com/openeuler/release-management/issues/I7FCHU, 2023. Accessed at December 09, 2025.

[26] Wenhui Zhang, Peng Liu, and Trent Jaeger. Analyzing the overhead of file protection by linux security modules. In *Proceedings of the 2021 ACM Asia Conference on Computer and Communications Security*, ASIA CCS '21, page 393–406, New York, NY, USA, 2021. Association for Computing Machinery.

Mohabi: Disaggregating and Sandboxing the Firefox JavaScript Engine

Abhishek Sharma
UT Austin

Anand Balaji
UT Austin

Zachary Yedidia
Stanford University

Anthony Du
UT Austin

Taehyun Noh
UT Austin

Iain Ireland
Mozilla

Jan de Mooij
Mozilla

Matthew Gaudet
Mozilla

Tal Garfinkel
Google

Deian Stefan
UC San Diego

Hovav Shacham
UC San Diego

Shravan Narayan
UT Austin

Abstract

We present Mohabi—a modern Firefox browser with a securely sandboxed JavaScript engine. Mohabi leverages software-based fault isolation (SFI) to ensure JavaScript engine bugs cannot affect the rest of the browser. To achieve this, we disaggregated the JavaScript engine (SpiderMonkey) from the rest of the browser, and developed techniques that leverage the type system and automatic code generation to make this complex transformation—that spans numerous data structures and deeply intertwined control flow across tens of thousands of functions—safe and tractable with reasonable engineering effort.

We then sandboxed SpiderMonkey using an optimized SFI toolchain we developed to meet the unique challenges of JavaScript engines, such as efficient support for large memory footprints. Mohabi only incurs modest overheads on common benchmarks—24.82% on JetStream and 24.43% on Speedometer. Mohabi is the most ambitious case study in retrofitting in-process sandboxing in a large system to date, and our x86-64 SFI toolchain is the fastest to date, imposing overheads between 5.9%–6.6% in SPEC 2017.

1 Introduction

Modern JavaScript engines are a seemingly inexhaustible source of critical memory-safety vulnerabilities [73]. In the past decade, these have become a major vector for in-the-wild attacks on browsers [58,90,130]. This problem is fundamental to the design of modern JavaScript engines, and there are several reasons for this.

First, production JavaScript engines like SpiderMonkey, V8 and JavaScriptCore rely on an intricate system of code generation and execution engines including interpreters, just-in-time compilers, and profile-based code optimizers. JavaScript engines dynamically juggle these different engines to achieve an optimal balance of startup latency and efficiency. This makes them incredibly complex (§2).

Next, their inputs—JavaScript, WebAssembly, and various Web APIs—are quite complex; as a result, the compiler and optimization stages have a very large attack surface. Further, this attack surface is not static, and continues to change and grow as Web standards evolve.

The final and perhaps most damning element that makes this situation intractable is that memory safety bugs in JavaScript engines are often not directly in the compilers or interpreters, but rather in runtime-generated code. For example, while JavaScript requires all array accesses to be bounds checked, JavaScript JITs produce code without these bounds checks when range analysis proves they can be safely elided. However, bugs in the range analysis [18, 19, 40] result in memory-unsafe code that can perform out-of-bounds array accesses. Consequently, our most powerful tool for preventing memory safety bugs—language-level type systems, e.g., in Rust—cannot address this problem.

Operating system and browser vendors have tacitly acknowledged that this is not a solvable problem in the near term, and introduced new modes that enhance security by disabling the JIT. In 2022 Apple introduced Lockdown Mode—an optional feature across their mobile, desktop, and wearable platforms—that disables the JIT in Safari [27]. Microsoft introduced Enhanced Security Mode [98] that similarly disables the JIT, and Android recently announced Advanced Protection Mode that "disables the JavaScript optimizer" [25].

This solution comes at a high cost. For example, in SpiderMonkey, JITs offer a 3.5×–7× speedup [52] for JavaScript. Even worse, disabling JITs doesn't offer a complete solution. For example, 5 of the 12 exploits in Chrome's JavaScript engine in 2025 would not be prevented by disabling the JIT [62], while a recent study found 23 remote-code execution vulnerabilities in the interpreter used in Edge's enhanced-security mode [135]. Browsers need a better solution.

Our solution. To address this challenge, we present Mohabi[1], the first sound sandbox relying on software-based fault isolation (SFI [134]) applied to JavaScript (JS) engines in a modern web browser.

[1] Mohabi stands for Monkey Habitat—a pleasant, but secure enclosure for the SpiderMonkey JavaScript engine used by the Firefox browser.

Mohabi sandboxes SpiderMonkey JS engine instances in the Mozilla Firefox browser. Thus, even if the JS engine is compromised, it is confined, and cannot arbitrarily corrupt memory in the rest of the browser or invoke disallowed system calls to escalate privileges.

In contrast, prior academic work [26, 106, 107] on JIT sandboxing (§9) has only sandboxed standalone JS engines in isolation, while industry-developed sandboxes (e.g., V8's Ubercage [70] and Safari's JITCage [37]) have so far been unsound and frequently bypassed (§7.4).

To build Mohabi, we have developed two new capabilities. First, we needed tools to allow disaggregating SpiderMonkey from Firefox, i.e., to separate out the JS engine, and then securely and efficiently re-enable control and data flow with minimal engineering effort. Second, we needed an SFI toolchain that could meet the challenges of sandboxing SpiderMonkey in production with low overhead—something that no existing SFI toolchain could do.

To enable disaggregation, we developed a set of C++ types and tools that allow us to securely express common patterns for sharing and control flow. For example, split-allocation types (§4.2) enable us to shard an object across separated (sandboxed and unsandboxed) allocators—based on what needs to be shared, and what needs to remain isolated. We also observed that modern browsers rely heavily on automatic code generation [5,15,17,39] and wrapper types [63,100,104] to integrate components together; we hook into and modify these tools to simplify disaggregation. Without these tricks, disaggregating SpiderMonkey would be a massive engineering effort—the JS engine itself supports well over 2,000 API calls, with an order of magnitude more call sites into the engine scattered over Firefox's considerable codebase. Once disaggregated, we needed a way to efficiently sandbox SpiderMonkey instances in memory—as the high memory and context switch costs of process sandboxing would have been a non-starter.

To address this challenge of scalable SFI, we developed MH-LFI. MH-LFI's key differentiating features are (1) its efficient support for large sandboxes—sandboxes with memory pools greater than 4 GiB, and (2) its safe handling of JIT-compiled code. The former is a necessity for modern browser features like ArrayBuffers and WebAssembly instances[2], while the latter is needed for all modern JS engines. MH-LFI works by rewriting assembly files to insert SFI-style guard checks into binaries, and is (mostly) independent from the choice of C/C++ compilers used to compile the browser—only a small amount of MH-LFI is implemented with compiler changes.

MH-LFI provides a separate validator to check that all sandboxed code has been correctly instrumented. This allows us to ensure that SFI instrumentation added by the JIT and AOT compilers is correct, removing the compilers from the trusted computing base (TCB). We also found that the validator was essential to catching edge cases of missing SFI checks that would otherwise have been overlooked (§6.3).

Mohabi's performance & security. We evaluate Mohabi on Speedometer [11]—the industry-standard full browser benchmark that tests both JavaScript and overall browser performance—and observe overheads of 24.43%. We break down the individual sources of overheads in §7.1. We also evaluate Mohabi on a pure JavaScript benchmark (JetStream [9]) to compare to prior work that sandboxed standalone JS engines [26,106,107]—and observe an overhead of 24.82%. We compare MH-LFI to Native Client (NaCl)—the SFI toolchain used in prior work sandboxing JS engines [26] and show how MH-LFI outperforms NaCl in §7.2. All modified AOT and JIT compilers pass all tests in the LLVM and SpiderMonkey codebases respectively, and based on our analysis, MH-LFI is the fastest x86-64 SFI toolchain to date.

In §7.3, we analyze how Mohabi defends against commonly used JS engine exploitation techniques and highlight some JS engine bugs that Mohabi would have mitigated—specifically, recent bugs reported by Anthropic's Mythos LLM [81] several months *after* Mohabi was developed. Additionally, we contrast Mohabi with Ubercage (V8's partial JavaScript sandbox) to highlight the differences between the two systems in §7.4. We analyze a subset of prior Ubercage sandbox bypasses and show how Mohabi prevents them.

Artifacts. All source code, prototypes, and benchmarks from this paper are available at `https://github.com/UT-Security/jitsbx-root`.

2 Why JavaScript Engine Security is Hard

Modern JavaScript (JS) engines have an extremely high bar for performance—they must begin executing code instantly to avoid introducing latency during page loads, while also efficiently executing modern websites that can be as complex as full-fledged desktop applications using JIT compilation.

To achieve this, they rely on a range of complex JITs and interpreters. This complexity has led to numerous vulnerabilities [69]. SpiderMonkey—the JS engine in Firefox that we sandbox in Mohabi—consists of 9 different code execution-engines [101]. JavaScript alone consists of a C++ bytecode interpreter, an optimized interpreter, along with non-optimizing and optimizing JIT compilers. These compilers are organized as tiers—at each tier the compiler takes longer to start executing, but applies more optimizations for greater efficiency. Additionally, there are interpreters and JITs for WebAssembly (Wasm) and regular expressions [83]. Other JS engines such as V8 (Chrome) and JavaScriptCore (Safari) use similar tiered architectures, although their internals differ.

[2]While WebAssembly code is already sandboxed [80], browsers' implementations are very buggy as they (1) reuse compiler infrastructure from their buggy JS engines, and (2) execute WebAssembly code that is very intertwined with JavaScript. We thus treat WebAssembly as effectively unsandboxed.

JS engines use these different *tiers* by dynamically changing compilation strategies at runtime, switching execution tiers based on how often a piece of code is executed. SpiderMonkey supports this with the help of *on-stack replacement*, allowing code to *tier-up* in the middle of execution—switching from being interpreted to executed as JITted code.

SpiderMonkey also supports switching from executing a JITted version of JavaScript code to interpreting it, through *bailouts*. This is necessary because the optimizing tiers employ *speculative optimization* [113], meaning that they optimize code based on type information collected during runtime—which is required for dynamically typed languages like JavaScript. This information is used along with other optimizations to specialize code, for instance, by eliding security checks (e.g., array bounds checks), if they are deemed redundant. The specialized code includes checks to ensure that assumptions hold on inputs to functions; if inputs are found to not be of the expected types, the result is a *bailout* from optimized tiers to the interpreter.

This complex architecture means that bugs in any of the above components—e.g., the JavaScript interpreters [69], the JavaScript compilers [20, 40, 66], the WebAssembly runtime [108]—result in vulnerabilities affecting the entire JS engine [76] and browser. These vulnerabilities are a common part of real-world exploit chains [58, 90] where JavaScript engine vulnerabilities are used to obtain initial code execution in the browser's renderer. This is then chained with privilege escalation vulnerabilities to take over a victim's machine.

Browsers have historically resorted to process-based isolation to address security concerns. The multi-process architecture [35] separated the browser into a privileged parent process and content processes. Site isolation [102, 119] extended this further by ensuring that tabs belonging to different top-level sites never share a content process. However, applying the same approach here—separating the JS engine—is a non-starter. The JS engine is tightly coupled with the DOM engine, event loop, and other browser subsystems, with extremely frequent interactions. Introducing an additional process boundary would require IPC for each of these interactions, resulting in prohibitive overhead.

In this paper, we present Mohabi—a sound, in-process sandbox for JS engines that can address this problem.

3 Mohabi: Securing the JavaScript Engine

Mohabi secures the JS engine by placing it in a sandbox implemented using software-based fault isolation (SFI). SFI works by isolating all memory (code and data) used by the JS engine in a dedicated portion of the process' address space called the *sandbox memory*, and restricting all memory-access and control-flow operations to this memory. Consequently, memory corruption resulting from JS engine bugs is restricted to sandbox memory and cannot affect the rest of the browser.

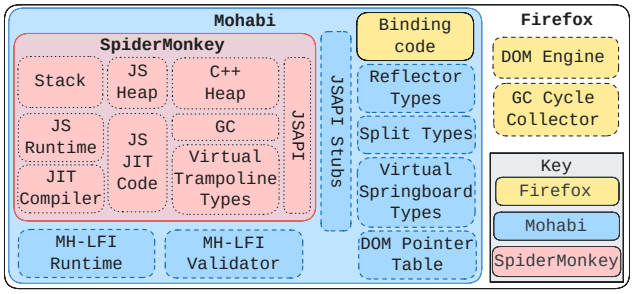


Figure 1: **Mohabi's design.** *Mohabi disaggregates SpiderMonkey from Firefox and places it in a high-performance MH-LFI sandbox. Disaggregation and secure interactions are mediated by custom types such as Reflector types. System-calls are checked by MH-LFI's runtime and code pages are checked by MH-LFI's validator to preserve security.*

Mohabi relies on our new SFI toolchain, MH-LFI (§5), that scales to a JS engine's unique requirements. MH-LFI is implemented in two parts: (1) an external rewriter that modifies the binaries produced by C/C++ compilers to ensure they follow SFI rules, and (2) a modified JIT compilation pipeline to ensure that all JITted code produced follows SFI rules. To exclude these compilers from Mohabi's TCB, MH-LFI comes with a binary validator that checks that all executable code pages within the sandbox comply with SFI rules.

Mohabi's design (Figure 1) is based on several observations about modern browsers that allow JS engine sandboxing without significant performance or engineering costs.

Browsers employ site isolation. Browsers implement site isolation [102, 119] to ensure different top-level sites are rendered in separate processes. This model ensures that a site's process does not contain any secrets about other sites visited by the browser. Consequently, Mohabi sandboxes JS engines in the site's process using write-only sandboxing—it does not restrict any reads of data. This is sufficient to prevent a compromised JS engine from escalating privileges.

Browser JITs share a compiler backend. The various JIT compilers used by JS engines produce native code through a shared architecture-specific emission backend [8, 12, 14]. Additionally, these backends tend to be stable over long periods of time; e.g., SpiderMonkey's backend last received a major update 10 years ago [2]. Mohabi uses this shared backend as a choke point for enforcing SFI—enabling instrumentation regardless of the specific JIT optimizations or the IRs used.

Browsers auto-generate glue code. All browsers auto-generate the glue code layer [1, 5, 17] between the JS engine and DOM—the primary browser component that interacts with the JS engine. Mohabi augments this code generation as a location to include safety checks on data output by the JS engine APIs—a necessary step to prevent confused deputy attacks on the DOM engine by a compromised JS engine.

Mohabi leverages these insights along with the new MH-

LFI SFI tool to sandbox the SpiderMonkey engine used by the Firefox browser. We note that Mohabi's design is general and can be applied to any modern browser.

4 Disaggregating SpiderMonkey from Firefox

The first step in Mohabi is to separate or "disaggregate" SpiderMonkey's code from Firefox code as sandboxing restrictions should only be applied to the former. This involves disaggregating control flow and data structures.

We *disaggregate control flow* by first isolating control flow, and then explicitly permitting safe control flow from SpiderMonkey to Firefox, e.g., for functions that are explicitly passed to the JS engine as callbacks. This is challenging as control flow can get arbitrarily nested. We then *disaggregate data structures* between Firefox and SpiderMonkey by placing only data that must be shared in the sandbox's memory.

Fortunately, we found this task was tractable across the large and complex Firefox codebase due to the presence of a handful of repeated patterns in how control flow and data sharing occur. We developed tools to handle these patterns, and thus enable safe and sometimes automated disaggregation.

These tools include split-allocation types (§4.2), which enable selective sharing of object fields across trust boundaries, as well as Reflector (§4.3), Virtual Trampoline, and Virtual Springboard types (§4.5), which enable safe and structured control flow between browser and sandbox. Additionally, we developed a stub library generator to wrap the SpiderMonkey interface (§4.4), which exposes 2,250 unique functions, and is used at an order of magnitude more call sites across Firefox.

We next explain our process of disaggregating SpiderMonkey, i.e., separating it from the browser and then safely reintegrating it back into the browser using the above tools.

4.1 Breaking out the JavaScript Engine

Firefox uses SpiderMonkey by directly linking it with browser code. We began by modifying the browser build process to compile SpiderMonkey separately using our Ahead-of-Time MH-LFI SFI toolchain (§5). Additionally, since SpiderMonkey's JITs compile code within the sandbox, we modify them to produce code conforming to our SFI scheme (§6).

To ensure SpiderMonkey does not share any data structures or symbols with the rest of Firefox, it is compiled as a standalone binary with its own private libc. This library must be explicitly loaded and assigned a memory region it can use—the sandbox memory. We refer to the produced library as the sandboxed SpiderMonkey instance—having all its control flow and stores restricted to a sandbox memory region.

Next, we modify each Firefox process that uses the JS engine to create a per-process sandbox instance. These processes include: the browser's UI (managed using JavaScript), each Content Process (handles JavaScript from the web), and the Proxy Auto-Config Process (configures the browser's proxy settings using JavaScript). When a sandboxed SpiderMonkey instance is loaded, our SFI validator (see §5.4) confirms that it is sandboxed correctly, then instantiates the sandbox instance. All threads spawned by the process (e.g., Web Workers in the Content Process) use the same sandbox instance, but receive their own per-thread sandbox context for stack, thread-local storage and registers.

```
template <typename T>
struct Rooted {
  Rooted<T>** stack; // pointer to stack top
  Rooted<T>* prev; // previous root in stack
  T ptr; // pointer to GC'd JS object

  explicit Rooted(RootingContext* cx) : ptr(...) {
      ...
    this->prev = ...; // point to prev root
    *this->stack = this;
  }
  ~Rooted() { ... }
};
```

Figure 2: **The Rooted<T> Type.** *One of the types used for JS object references in the browser that need to be registered with the GC. We implement this in Mohabi as a split-allocation type, with select fields moved into the sandbox.*

4.2 Safe Garbage Collector Support

SpiderMonkey provides a moving garbage collector (GC) that moves JavaScript (JS) object allocations to optimize tracing and reduce fragmentation on the heap. Moving a JS object requires the GC to update all references to the object to the new location; this includes any references from Firefox code.

SpiderMonkey tracks such references by requiring Firefox to use custom types like `Rooted<T>` in Figure 2 to declare references to JS objects. On construction, `Rooted<T>` automatically adds itself as a node in a LIFO linked-list of GC roots. SpiderMonkey uses this LIFO linked-list to trace the active roots when a GC is triggered.

However, disaggregating SpiderMonkey means that the GC can no longer directly update such references to JS objects that come from Firefox code. To restore functionality, we must allow the GC to modify `Rooted` references when object migration or heap compaction occurs. Concretely, this means that the GC should be permitted to modify the `ptr` field from Figure 2, but not other fields in the object such as the LIFO link-list. To allow secure updates to the `ptr` field, we introduced split-allocation types.

Split-allocation types. These types split a single logical object across two locations. Some fields are stored in sandbox memory, to enable updates by sandboxed code—other fields are allocated outside of the sandbox—to protect their integrity. This separation and management of multiple allocations for different fields is hidden from the user of the type.

A split version of `Rooted<T>` is one use of split-allocation

```
1  // this is called when 'window.location' is modified
2  static bool
3  set_location(JSContext* cx_, JS::Handle<JSObject*> obj,
4               ... , JSJitSetterCallArgs args) {
5    BindingCallContext cx(cx_, "Window.location setter");
6    ...
7    JS::Rooted<JSObject*> targetObj(cx, &v.toObject());
8    return JS_SetProperty(cx, targetObj, "href", args[0]);
9  }
10 // defines set_location as a JSObject property setter
11 static const JSJitInfo location_setterinfo = {
12   { set_location }, ... ,JSJitInfo::Setter, ...};
13 static const JSPropertySpec sUnforgeableAttributes_spec[]
14   = { JSPropertySpec::nativeAccessors(
15           "location", JSPROP_ENUMERATE | JSPROP_PERMANENT,
16           ..., &location_setterinfo), ...};
```

Figure 3: **The DOM-JS Engine binding layer.** *All production browsers use automatic code generation to create a binding layer between the JS engine and the DOM. Here the code shows a setter function on the* `window` *DOM reflector object.*

types. Other GC types such as `Heap<T>`—which captures long-term references from Firefox's heap to SpiderMonkey objects—were also migrated to split-allocation types. Overall, these types were applied to thousands of sites in the browser.

4.3 Supporting Firefox's DOM reflectors

The Document Object Model (DOM) specification and associated web platform specifications [3, 6] list APIs that JS engines can use to interact with the browser. Beyond HTML manipulation, these APIs support Web Workers (background threads), Web Storage (key/value storage), and Geolocation. These APIs are exposed to the JS runtime with the help of DOM reflectors, which are projections of DOM functionality as JS objects.

For example, the `window` object in Figure 3 is a DOM reflector that has several attributes—one of which is `location`. When executing JS code that sets the `window.location` property, SpiderMonkey invokes the `set_location` function at line 3. This function forwards the write to the `window` object's `href` property (line 8), requiring a nested invocation to the JS engine at the `JS_SetProperty` call.

However, disaggregation would mean that such callbacks cannot directly be invoked by SpiderMonkey, as the code is outside the sandbox memory (in Firefox's code). Mohabi instead requires SpiderMonkey to register all such callback functions as legitimate callback targets via a provided API. When registered functions are invoked as callbacks, Mohabi permits this control-flow securely through a trusted routine called a trampoline (described in §4.4).

We modify Firefox to explicitly register each callback function with the sandbox runtime at the time of creating a DOM reflector object. An instance of this is when defining the setter function (line 12) for the `location` property (line 15) before wrapping the browser's `window` object in a DOM reflector.

To apply this modification automatically across multiple data types, we introduced sandboxed reflector types.

Sandboxed reflector types. These types build on the observation that all browsers use API specifications written in WebIDL (an interface definition language) to auto-generate binding code [1, 5, 17, 39] to set up DOM reflector types. We modify this code generator to be sandbox-aware. For each DOM reflector type that it previously generated, we instead generate a sandboxed reflector type. When the first reflector object instance corresponding to a sandboxed reflector type is created, the browser functions exposed to JS are automatically registered as callbacks with the sandbox.

We needed a systematic solution to this because this approach applies not just to the `Window` reflector type but for the over 1,075 other reflector types automatically generated from WebIDL. These reflector types could potentially register up to 15,700 callbacks—if JavaScript code causes an instance of every single DOM reflector type to be created at least once.

4.4 Safe JS Engine Calls from the Browser

The browser can make function calls into the JS engine, for example, to modify the properties of a DOM reflector object as discussed in §4.3. These functions are exposed to Firefox through SpiderMonkey's library interface (the *JSAPI*). By building and linking SpiderMonkey separately from the browser, we have blocked direct calls into this code. We now focus on securely permitting these function calls.

Mohabi requires that all control-flow transfers from Firefox into the sandboxed JS engine use a piece of trusted code called the springboard. Springboards perform a safe context-switch on the way into the sandbox—they are responsible for switching stacks, saving callee-saved registers and setting the contents of reserved registers used by the SFI scheme (§5.1). Trampolines do the inverse for control-flow transfers in the other direction, when SpiderMonkey code returns to the browser, for example, through a callback registered with the sandbox runtime (§4.3).

To automatically re-allow safe JSAPI calls in Firefox via springboards, we build a stub-library generator.

Stub-Library Generator. The generated stub-library helps us account for the now dynamic location of the sandbox instance being invoked. This is done with the help of (1) an indirect-jump table with an entry for the resolved sandbox address of each JSAPI symbol and (2) a unique springboard routine corresponding to each JSAPI symbol.

Linking the browser to the SpiderMonkey stub-library statically resolves all JSAPI symbol references to a springboard. When the MH-LFI runtime instantiates the SpiderMonkey sandbox instance, it populates the indirect-jump table in the stub-library, allowing calls from Firefox to the JS engine.

The support provided by our automatically generated stub library is essential as the JSAPI exposes over 2,250 functions and Firefox contains an order of magnitude more unique

references to them, making explicit modification at each call site in the code impractical.

4.5 Interactions with the Browser Event Queue

JavaScript provides support for programming with asynchronous tasks using Promise objects. For example, the Fetch API, which is the modern JS interface to make HTTP requests, returns a Promise object. The `Promise.then` method allows a user to specify continuation code, which is scheduled for execution once the fetch request completes.

```
const promise = fetch("https://example.org/products.json");
promise.then((response) => ..., (error) => ...);
```

Support for scheduling these asynchronous jobs is provided by the browser rather than the JS engine. Concretely, the browser provides a `JobQueue` object (shown below), allowing SpiderMonkey to schedule jobs on the browser's job queue:

```
class JobQueue {
  // Enqueue a resolution job for promise
  virtual bool enqueuePromiseJob(JSContext* cx,
    JS::HandleObject promise, JS::HandleObject job,
    ...) = 0;
  ...
};
```

However, this is a problem when separating SpiderMonkey from Firefox. As this abstract class is implemented within the browser, the virtual functions are located outside the sandbox and must be explicitly registered as allowed callbacks (similar to the DOM reflectors in §4.3) to be callable by the JS engine. However, since vtable entries are automatically populated by the C++ compiler, there is no easy way to do this.

To address this, we introduce virtual trampoline and virtual springboard types, which help mediate dynamic dispatch across the sandbox boundary in both directions.

Virtual Trampoline Types. These are types within the sandbox that inherit from another underlying type, then override its virtual method implementations to redirect the call through a function pointer table instead. The user of a virtual trampoline type is expected to initialize the function pointer table with valid callback functions (§4.3).

Virtual Springboard Types. These are types outside the sandbox that help transfer control-flow into the sandbox. They expose the same interface as the underlying sandbox type, and act as wrappers. Each virtual method implementation forwards the call through a corresponding function exposed by the sandbox; this performs the actual virtual dispatch on the underlying instance within the sandbox.

In addition to job scheduling for Promises, we use virtual trampoline and springboard types in SpiderMonkey's implementation of JavaScript Proxy Objects, External DOM Strings and GC tracing, among others. In total, we expose these types for around 20 classes in the JSAPI.

```
1  // Binding code before tainting
2  bool get_width(..., JS::Handle<JSObject*> obj,
3           void* void_self, ...) {
4    ...
5    auto* self = static_cast<mozilla::dom::ImageData*>
6                                (void_self); }
7
8  // Binding code after tainting
9  bool get_width(..., JS::Handle<JSObject*> obj,
10   MC::OpaquePointer<void*> void_self, ...) {
11   ...
12   auto* self = MC::dom::DOMPointerTable::sanitize
13             <mozilla::dom::ImageData*>(void_self); }
```

Figure 4: **Adding Sanitization Checks to the Boundary.** *Firefox's binding code auto-generation is modified to insert sanitization checks before using the DOM pointer retrieved from the sandbox. This example shows the generated binding code for returning the width of an image.*

4.6 Type Safety at the Binding Layer

Secure disaggregation requires sanitizing any data received by Firefox that comes from SpiderMonkey. This is needed to prevent the sandboxed code from compromising Firefox via confused-deputy attacks [45, 82, 104]. We illustrate this using DOM reflectors as an example.

When DOM reflectors are passed to JavaScript, a pointer to the underlying C++ DOM object that it reflects is stored within it. When JS engines invoke a permitted DOM reflector callback (§4.3), this pointer is extracted and used to call methods on the underlying DOM object. However, this pointer could have been corrupted by compromised JS code to induce memory-safety errors in Firefox when the reflector is used.

To address this, we introduce pointer tables.

DOM Pointer Table. This is a table where each entry stores a valid DOM pointer, along with the type of the object. To make sure DOM code remains safe, we augment binding code generation to sanitize the incoming pointer, by making sure it exists within the table and is of the right type. Figure 4 illustrates one such example—the code at line 5 unsafely uses the pointer, whereas the augmented code at line 12 sanitizes the pointer before using it. To support this, we modify DOM reflector initialization to add an entry to the pointer table. As multiple reflector objects may point to the same underlying DOM object, each table entry also maintains a reference count, which is decremented on DOM reflector finalization.

While this case study serves as an example of how we can automate boundary checks, a more complete handling of boundary checks requires more work, and is discussed in §8.

5 Sandboxing SpiderMonkey Code

We now discuss how Mohabi can sandbox a disaggregated SpiderMonkey engine from Firefox using software-based fault isolation (SFI) [134]. SFI modifies compilers to insert guards

prior to all memory accesses and control-flow instructions to restrict them to the sandbox memory; additionally, all system calls are rewritten to function calls on a trusted runtime that blocks calls which may compromise the sandbox [47].

In order to scale to a production system like SpiderMonkey, an SFI toolchain must satisfy the following requirements:

- **No artificial memory limits.** Almost all SFI toolchains [53, 80,97,105,123,139] impose a limit of 4 GiB on the sandbox memory for efficient sandboxing. This is entirely impractical for browser features like ArrayBuffers, WebAssembly, etc. The rare SFI toolchains that don't impose this limit today (e.g., Wasm-64 toolchains [121]) lose efficiency and have impractically high performance overheads [126].
- **Support concurrent JIT code generation.** The SFI runtime must allow safe additions, removals, and modifications to sandbox code—even under concurrent execution by multiple threads[3]. This is necessary to support sandboxing of the JIT compilers used by JS engines.
- **Support binary validation.** While the JIT compiler itself is sandboxed, safely running code produced by it requires the SFI scheme to support binary validation [134]—to ensure that all SFI checks are present in the generated code. Binary validation has the added benefit of moving a buggy AOT SFI compiler [89, 99] out of the TCB.
- **Use the native ABI.** SFI tools such as NaCl [140] and Wasm [80] use sandbox-relative pointers (offsets) rather than native pointers—requiring translation between these representations at the sandbox boundary. While improving a sandbox's memory footprint [124], this significantly complicates retrofitting [104].
- **No reliance on hardware extensions.** While allowing efficient sandboxing [133, 137], browser users have a variety of different hardware and operating systems [4], without consistent support for such features.
- **No compiler "hard forks".** SFI compilers are often implemented as bespoke forks of GCC/clang [54, 123, 140]. This imposes a long-term maintenance burden and often means SFI tools are stuck as forks of outdated compiler versions.

We designed MH-LFI (building on the LFI [139] SFI compiler) to meet all these requirements—requirements that no existing SFI toolchain satisfies. MH-LFI's SFI scheme and supporting runtime are described in §5.1 and §5.2.

Even beyond these requirements, large applications like SpiderMonkey contain code patterns that add slowdowns not observed in standard benchmarks where SFI tools are tested. We describe several new optimizations that we needed to develop to address this in §5.3. Finally, we describe how MH-LFI can meet the above requirements when implemented as a simple assembly rewriter operating on the output of compilers (§5.4)—allowing Firefox to continue to support multiple compiler toolchains and frequently upgrade versions.

[3] Support for concurrent execution and generation/modification of code is needed as browsers' JS engines support threads via Web Workers.

Original code	Sandboxed equivalent (a "macroinstruction")

`addq %rdi, (%rax)`	`movq %rax, %r11`
	`andq %r15, %r11`
	`addq %rdi, (%r14, %r11)`

`movq %rdi, (%rax)`	`pext %r15, %rax, %r11` `movq %rdi, (%r14, %r11)`

`jmpq *%rdi`	`andq %r15, %rdi`
	`andq $0xfffffffffffffffe0, %rdi`
	`addq %r14, %rdi`
	`jmpq *%rdi`

Table 1: **Rewrites needed to conform to MH-LFI's data access and control-flow policies.** *Instructions on the left are rewritten into a* `macroinstruction`*—a collection of instructions that must all reside within the same bundle (to guarantee their execution as a unit). Note that the* `movq` *example preserves flags by using* `pext` *rather than* `and`*.*

MH-LFI allows us to sandbox SpiderMonkey's ahead-of-time code written in C/C++. We defer the discussion of adapting MH-LFI's scheme to SpiderMonkey's JITs to §6.

5.1 MH-LFI's SFI Scheme

MH-LFI's rewriter enforces a sandboxing scheme reminiscent of prior SFI tools (e.g., Native Client's x86-64 implementation [123]), but designed for sandboxes with large memory pools and augmented with additional optimizations (§5.3).

MH-LFI assumes that sandbox memories are contiguous, have a size that is a power of two, and are also aligned to the same power of 2. Within the sandbox memory, all pages are either non-writable for code or non-executable for data (enforced via `mmap` protections), with additional special handling for JITs (§6.2) which require writable *and* executable pages.

Like prior classic SFI schemes, MH-LFI comes with a binary validator that analyzes executable machine code to determine whether such code is guaranteed to maintain the sandbox's security guarantees. We now describe these requirements along with the associated rewrites that must be applied to compiler output to allow it to pass this validator.

MH-LFI's rewriter applies three security policies: the *data access policy*, which ensures that all memory writes are restricted to the sandbox region; the *control-flow policy*, which ensures that all control flow stays within the sandbox region (except to specific entry points in host code); and the *system-call policy*, which ensures that system-call instructions are rewritten to function calls in the trusted runtime. The rewriter uses three reserved registers—`r14` (sandbox base), `r15` (sandbox mask) and `r11` (scratch)—to enforce these policies, as illustrated by the examples in Table 1.

Data access policy. Memory writes are prefixed with guard instructions that (1) Compute the original effective destination address into the `scratch` register; (2) Mask out the top bits of the `scratch` register to restrict the address to a specified size—256 GiB for Mohabi; and (3) Add the 64-bit sandbox

base address to produce the final address that is used for the write. As Mohabi's sandbox memory is an aligned, contiguous allocation of 256 GiB (§5.2), this transformation guarantees that memory writes to the stack or heap are within the sandbox memory. MH-LFI optionally allows memory reads to be restricted in the same way, but this is unnecessary for Mohabi when site isolation [102, 119] is present.

Stack operations that operate directly on the stack pointer register, `%rsp` (e.g., push, pop, etc.) are handled by ensuring that the `%rsp` register always points to sandbox memory—for performance, this is enforced any time `%rsp` is updated [134] (rather than prior to each stack operation). As Mohabi's sandbox memory is surrounded by guard regions (§5.2)—unmapped memory pages—sequences of stack push/pops will fault if they go past the ends of sandbox memory.

Control-flow policy. To enforce that control transfers remain within the sandbox region, we use a similar mask sequence before indirect jumps. However, we must additionally enforce that indirect jumps do not target the middle of another instruction, as this may execute alternate instruction streams. To prevent this, we use bundling [97].

Bundling requires that individual instructions and macroinstructions (e.g., the sandbox-equivalent transformation of instructions shown in the second column of Table 1)—be contained within a single 32-byte bundle. If a (macro) instruction spans a bundle boundary, the rewriter injects padding using no-op (`NOP`) instructions to move the instruction into the next bundle. The rewriter ensures that the control-flow mask zeroes the bottom 5 bits of the control-flow targets prior to jumps—forcing all jumps to land on the start of bundles.

Return instructions (`ret`) are transformed to a `pop-mask-jump` sequence so that we can mask the address in a register—directly masking this on the stack would allow a concurrent thread to modify a return address after it has been guarded.

System-call policy. Allowing arbitrary system calls can allow sandboxed code to effectively break out of the sandbox [47]. Thus, similar to prior work, system calls are rewritten into function calls to a trusted runtime (§5.2) which checks if the system call can be safely permitted [84, 88, 122].

5.2 MH-LFI Runtime

MH-LFI also provides a runtime for Firefox to safely interact with the sandboxed SpiderMonkey instance. Such runtimes are a standard part of sandboxing toolchains [16, 84, 88, 96, 122, 123, 139, 140], so we only offer a brief description here. The MH-LFI runtime offers the following features.

The runtime provides binary (ELF) loaders and sandbox memory initialization routines that allocate a contiguous memory space, copy the binary code into this space, and initialize a new stack for the sandboxed code in order to support code execution in a sandboxed context.

The runtime provides *springboards* and *trampolines*—assembly routines that allow safe transitions into and out of sandboxed code. These routines switch the location of the program stack (sandboxed code requires stacks inside the sandbox memory), enforce caller/callee-save register expectations to preserve security [91], and set reserved registers (e.g., `r14` sandbox base) to expected values prior to transferring control-flow. MH-LFI's springboards and trampolines are standard and resemble those used in prior SFI tools [123, 140].

Finally, the runtime interposes on all system calls made by sandboxed code (with help from the rewriter (§5.1)), and permits only system calls that don't compromise the sandbox. These include the system calls used to manage executable code pages (§6.2)—to allow safe addition, removal and modification of JIT-compiled code within the sandbox.

5.3 Optimizing MH-LFI

In addition to implementing known SFI optimizations [105], we implemented several novel optimizations in MH-LFI. The first optimization is implemented in an LLVM pass, while the other two are implemented directly in the rewriter.

Prefix Padding. Bundles require inserting `NOP` instructions to prevent instructions from crossing bundle boundaries. Such `NOP`s may increase frontend stalls in the CPU pipeline. As a remedy, we replace `NOP`s by distributing their padding length across other non-`NOP` instructions within the bundle using x86 instruction prefixes [10]. We do this by adding one or more `cs` segment prefixes to instructions that immediately precede a `NOP` within the same bundle, limiting the total number of prefixes to 5, to avoid CPU decoding overheads.

Thread Local Storage (TLS) Operations. On Linux, TLS is supported using the `FS` segment base register. In the context of SFI, since the `FS` segment register is already used by the host, sandbox code must access TLS in a different way. Our original implementation of MH-LFI followed prior work, locating TLS using a call to the sandbox runtime—while reasonably performant for most domains, we found that this was extremely slow for browsers. We optimized these TLS operations by leveraging the observation from [105]—to make use of the unused `GS` segment register on x86-64.

Flags Preserving Masking. MH-LFI uses bitwise `and` instructions to ensure that memory accesses remain in the sandbox memory. However, the x86-64 `and` instruction modifies the processor flags; in cases where this is an issue for subsequent instructions, we use the `pext` instruction instead, which does not modify the flags. Since `pext` is slow compared to `and`, we only use it when the assembly rewriter determines that the flags must be preserved across the memory access.

5.4 Implementing MH-LFI

MH-LFI is implemented on top of LFI [139] and is a standalone assembly rewriter operating on the output of (mostly) standard clang/LLVM compilation with specific flags and a minor patch to clang/LLVM to support optimizations.

The rewriter. The rewriter consumes object files output by compilers and transforms the GNU assembly into the bundled macroinstructions described previously. These new object files can then be linked and used as normal. The rewriter relies on minor changes to the compiler that we discuss next.

Compiler patch. Since Firefox uses clang/LLVM as its compiler by default, we focus on this toolchain (although our approach also works with GCC). MH-LFI relies on three minor changes to the compiler, which we expose under a new compiler target (target triple `x86_64_mohabi-linux-musl`). First, MH-LFI's scheme utilizes guard instructions that require a `scratch` register for intermediate computations, a `base` register and a `mask` register (§5.1). Since we have to transform code after register allocation, we instruct the compiler to leave these registers unused during compilation. Second, we enable the use of instruction alignment directives—an off-by-default feature supported by LLVM—to align instructions into bundles. Third, we add an LLVM backend pass that optimizes assembly for our SFI scheme (§5.3). These changes account for only ≈ 600 lines in clang/LLVM. As a proof point of easy maintenance, we began the project in LLVM 19, but shifted to LLVM 22 partway, with less than 3 days of work.

Validating the SpiderMonkey binary. Similar to other SFI toolchains such as Native Client (NaCl) [140], MH-LFI includes a binary validator to ensure that all code within sandboxed binaries follows the control-flow and data-access policies specified in §5.1. Our validator, however, supports a large-sandbox model and uses a modern decoder (Fadec [55]) to enhance performance (making it faster than NaCl's validator (Table 3)). The verifier is written as a C program that checks all requirements of the SFI scheme, similar to prior work [89, 99, 140].

6 Sandboxing the JIT

SpiderMonkey uses a complex hierarchy of JIT compilers to speed up execution (§2). We need to modify these JITs so that the generated code is securely sandboxed. Concretely, this takes two steps: (1) modifying JIT compilers to generate MH-LFI-compliant code, and (2) using new secure APIs to manage modification of code pages—a unique challenge for JITs as this is very performance-sensitive code. We discuss this next followed by examples of easy-to-miss edge cases.

6.1 Modifying JIT Compilers

SpiderMonkey uses an abstraction layer called MacroAssembler (MASM) for all dynamically generated code. All of its JIT compilers emit code as sequences of MASM operations, each of which gets lowered to native code by an architecture-specific backend. We modify MASM's x86-64 backend to enforce MH-LFI's data access and control-flow SFI policies.

Implementing the SFI scheme. Like we did for AOT code, we reserve registers `r14` and `r15` for SFI by marking them as unallocatable in the MASM register allocator. We modify MASM operations that perform memory-access or indirect control-flow to emit the appropriate masked macroinstructions, and modify MASM's x86-64 instruction assembler to insert `NOP`s to prevent (macro) instructions from crossing bundle boundaries. Finally, we identify all valid indirect targets in the code emitted by each of the JIT compilers and ensure that they are bundle aligned. A difference compared to the AOT case is that we don't reserve `r11` since SpiderMonkey's JITs used this as an internal scratch register; we opted instead to share the scratch register (for JIT and SFI) to reduce the register pressure on this code.

Disentangling JIT-compiled Code and Data. As an optimization, JITs emit data such as floating point constants and relocation tables—used to trace and update references to garbage collected data in the JIT-compiled code—in executable memory. Unfortunately, this results in JIT-compiled code containing code-gadgets (random constants that are interpreted as instructions) that violate the SFI scheme. To prevent this, we modify MASM to emit constants in 32-byte bundles where constants are placed in bytes 1 to 31, and byte 0 is fixed as the byte for an x86 `HLT` instruction, meaning any attempt to jump here would fault. Where possible, we also used a more direct approach of moving data outside code sections.

6.2 Managing JIT Code Pages

The sandboxed JIT compiler must update executable code pages to both add new JIT code and modify existing JIT code (for instance, to perform bailouts (§2)). However, this is a major performance challenge for an SFI tool, because the safe way to update code pages in a multi-threaded environment is a prohibitively slow three-step process: (1) the code page must be made read-write and the JIT compiler modifies the part of the code page it wants, (2) the code page must be made read-only and the validator must check the *entire* code page as any part of the previously writable page could have been tampered with by a concurrent thread, and (3) the code page must be made read-exec after the validator has finished. Any deviation from this recipe would introduce opportunities for a malicious concurrent thread to execute unvalidated code or tamper with code page contents to remove SFI checks.

Our solution: Dual Mapping. To avoid the overheads of having to reverify entire code pages any time code is updated, Mohabi uses an alternate approach called Dual Mapping. Here, MH-LFI's runtime uses its system-call interposition (§5.2) to support writing to sandboxed code pages in two steps: First, when an executable page is made writable, the runtime creates a shadow virtual page with read-write permissions to the same underlying physical page outside the sandbox. Second, MH-LFI offers a trusted-runtime call to safely and efficiently perform updates to this shadow mapping. This trusted call

takes the following steps: (1) it makes the code page mapped in the sandbox read-only, (2) it updates the externally mapped writable page with the new code and validates *only the new code*, (3) it restores the code page in the sandbox to executable. This approach is much faster as it reduces page permission updates from 3 to 2, and requires only validating new code. The latter optimization makes a huge difference in practice—safe code updates went from doubling our overheads to adding negligible overheads in our evaluations. While dual-mapping with multiple processes was explored by prior work [128] and prior production browsers [56], we see that MH-LFI's in-process isolation makes this much more practical.

6.3 Why Validating SFI Code Matters

Using a binary validator, beyond being necessary for secure JIT compilation, also helped uncover cases of missed SFI checks in the JIT in early prototypes. We discuss this below.

Case study 1: Zero-padding code pages. Wasm and JavaScript JIT compilers do not always fill the entirety of allocated executable memory with code. The unused space at the end of an allocation is filled with the byte `0x00`, which decodes to the `addb %al,(%rax)` instruction, violating the memory isolation properties of MH-LFI. Upon discovery, we modified the trusted-runtime call used to manage JIT code pages (§6.2) to instead pad inserted code with the `0xcc` byte—a safe interrupt instruction on x86.

Case study 2: Missing Indirect Jump Masks. We modified MASM to mask the target of all indirect jumps. However, in certain instances, the JIT compiler directly uses the low-level assembler to emit indirect jump instructions, which are unmasked and therefore insecure. The validator found these errors, allowing us to fix this.

Case study 3: Clobbering of Reserved Registers. The Wasm compiler generates code that may trigger traps as a form of error handling. To support recovery from the trap, Wasm saves any register it uses when the trap occurs and restores it upon resuming. However, since Wasm code is not trusted, it should not be permitted to restore reserved registers like the sandbox base `r14`. The validator flagged these errors, and we made changes to ensure that these registers were never modified.

7 Evaluation

In this section, we evaluate Mohabi's performance and security. Mohabi's performance is evaluated on a machine with an Intel Raptor Lake i9-13900K, 32 GiB RAM, Ubuntu 24.04 on Linux 6.14.0. For browser and JS benchmarks, we use Firefox ESR-115 (with and without Mohabi) pinned to two isolated CPU cores whose frequency is fixed at 2.2 GHz with hyper-threading disabled. SPEC benchmarks run isolated on one CPU core fixed at 2.2 GHz with hyper-threading disabled.

Benchmark	Firefox	Mohabi
JetStream 2.2	86.27 ± 0.71	64.86 ± 1.10 (24.82%)
Speedometer 3.1	6.81 ± 0.06	5.14 ± 0.03 (24.43%)

Table 2: **Comparison of Firefox and Mohabi scores on the JetStream 2.2 and Speedometer 3.1 benchmarks.** *Results characterize the median score across 15 runs of vanilla Firefox and Firefox with Mohabi on each benchmark. For both benchmarks, a higher score is better.*

7.1 Performance of Mohabi

We evaluated Mohabi on the industry-standard browser benchmark, Speedometer 3.1 [11]. Speedometer measures end-to-end browser performance using simulated user interactions with web applications—modeling real-world web use.

While our focus is on full-browser performance, prior academic work [26, 106, 107] only sandboxed standalone JS engines and thus relied on JavaScript and Wasm benchmark suites like JetStream 2.2 [9]. We also evaluate this to enable a more direct comparison with prior work.

Finally, we isolate the cost of each element of the sandbox: memory masks, instruction bundling, forward-edge control-flow checks, backward-edge control-flow checks, system call mediation, trampolines and springboards. As a reference point, we also show the browser performance when disabling the JIT—to estimate overheads of the hardened no-JIT modes as used in Edge, Safari, and Chrome.

Results. Mohabi's Speedometer and JetStream results are shown in Table 2. We see total overheads of 24.43% and 24.82% respectively. While not negligible, these are extremely modest compared to partial security wins of disabling the JIT altogether. Mohabi also imposed no noticeable slowdown in routine use on sites like YouTube and Reddit.

While a full apples-to-apples comparison with prior work is impossible as JS engines and benchmark suites differ and continually evolve, we nevertheless look at the rough trends (Figure 8). Compared to prior work, Mohabi is not as expensive as previous attempts at a fully sandboxed JavaScript engine, e.g., NaClJIT [26], but more expensive than unsound SFI mitigations used by production browsers like Ubercage [70].

We show the performance overheads of the individual components of Mohabi in Figure 5. Broadly, backward-edge control-flow checks are the dominant source of overhead. Other costs are smaller, but they still add up. However, as noted by prior work [26], the costs of the individual components do not add up linearly, as each component has an impact on others when enabled together.

7.2 Performance of MH-LFI

To evaluate MH-LFI, we compare its performance with the Native Client (NaCl) SFI toolchain [123]. We chose this for two reasons. First, this was the toolchain used by Na-

Fwd-CFI	Bwd-CFI	Bundles	Store-Masks	Syscalls	Mohabi	No-JIT
6%	13%	5%	6%	3%	24%	53%

Figure 5: **Mohabi decrease in Speedometer score by SFI component.** *We observe that each security check from SFI adds some overheads, with the largest overheads coming from backward-edge control-flow protections. For reference, we also show the overheads of disabling JS JITs.*

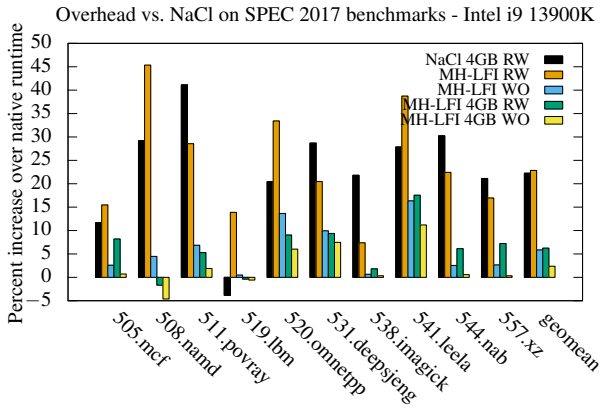


Figure 6: **SFI overheads on SPEC 2017 benchmarks that are compatible with NaCl.** *We observe that MH-LFI* 4 GiB *outperforms NaCl (6.6% vs. 22.3%) on the compatible sandbox configuration.*

ClJIT [26]—the closest prior work to this paper. Second, despite newer standards for sandboxing such as WebAssembly [80], prior work has shown that these toolchains incur significant overheads [87, 104] (in part, because they use a simple platform-agnostic intermediate representation). NaCl still remains the fastest x86-64 SFI baseline available today.

To ensure a fair comparison, we run MH-LFI both as used by Firefox, but also in a configuration that matches NaCl—a 4 GiB sandbox that guards read and write operations. NaCl does not support larger sandboxes and no longer supports write-only sandboxes. Since MH-LFI and NaCl are built on different baseline compiler versions (clang 22 and clang 3.7.0 respectively), we measure overheads normalized with respect to their own baseline clang versions for fairness and report these on the subset of SPEC 2017 that NaCl supports.

Beyond this, we report the benefit of Mohabi's optimizations (§5.3), the speed of Mohabi's validator, and overheads of Mohabi on the full SPEC 2017 benchmark in the Appendix.

Results. Figure 6 shows overheads on SPEC 2017. Comparing 4 GiB sandboxes, we see that MH-LFI imposes 6.6% overhead vs NaCl's 22.3%. For a large-memory, write-only sandboxing as we use in Mohabi, we see 5.9% overhead. The Appendix shows that Mohabi optimizations were able to speed up Mohabi by 6%–8% (Figure 9) and Mohabi's validator outperforms NaCl's validator (Table 3).

7.3 Security Analysis of Mohabi

Ideally, the security of systems like Mohabi can only be evaluated with extended production use. As a basic assessment, we discuss how Mohabi blocks common JS engine exploit techniques, give examples of mitigatable JS engine bugs discovered after its implementation, and finally discuss its TCB.

Common JS attack techniques. We discuss how Mohabi blocks common JS attack techniques used in exploits—both in-the-wild and in competitions like Pwn2Own.

Code pointer hijacking: Exploits can corrupt control flow by attacking code-pointers on the forward edge [115] or the backward edge [48]. MH-LFI's CFI scheme ensures that jump and return targets remain within the sandboxed region, and that they are bundle-aligned, as described in §5.1. MH-LFI is also not vulnerable to control-flow bending [43]-style attacks, as it does not rely on a fine-grained CFI to provide guarantees.

Overwriting Executable Memory: Some exploits take advantage of the fact that not all browsers implement *Write Xor Execute* for JIT executable memory [41, 69]. Mohabi prevents this by construction with the dual-mapping scheme (§6.2).

JIT spray attacks: Exploits may try to fill code pages with shellcode/gadgets by either leveraging existing JITted code or by crafting code that uses floating point constants that encode gadgets [115, 120]. The MH-LFI validator prevents such attacks by making sure that JITted code cannot contain gadgets that bypass SFI guarantees.

Corruption through concurrent threads: Exploits may leverage the presence of JS engine threads (in the form of Web Workers [7]), to use one thread to overwrite code in the JIT compilation buffer of another thread [77, 78]. MH-LFI prevents this by ensuring that the runtime always checks compiled code using the binary validator before it is copied into an executable page. TOCTOU issues with validation are prevented through the dual mapping scheme (§6.2).

Unsafe syscalls: Exploits may invoke syscalls as part of a privilege escalation [42, 49, 129]. The MH-LFI runtime prevents this as it intercepts all syscalls from within the sandbox, and allows only a safe subset of syscalls with checked parameters.

Evaluating new JS bugs. In April 2026 (after Mohabi's prototype was complete), Firefox shipped their largest batch of security fixes in monthly updates [65], to fix bugs discovered through Anthropic's Mythos LLM including several bugs in the SpiderMonkey JS engine [81]. We examined a few of these bugs and saw cases of memory corruption in the JIT compiler [21], GC [24], inline caches [22], and the JS runtime [23]. Mohabi would mitigate all these bugs by confining the memory corruption to within the sandbox's memory. Attacks leveraging these bugs would only be possible via confused-deputy attacks on missing boundary checks (consistent with the threat model of Mohabi).

TCB. The unsandboxed SpiderMonkey engine contains ≈ 784,000 lines of code (≈ 50% from the JIT compiler and

Wasm runtime)—all of which are a part of its TCB; further, any JITted code emitted at runtime is also part of the TCB. In contrast, Mohabi has a relatively small TCB consisting of the MH-LFI runtime, MH-LFI validator and data sanitizers at the sandbox boundary. The MH-LFI runtime is implemented in 6,200 lines of C. The MH-LFI validator relies on the Fadec [55] x86 decoder which uses a 1,894 line x86-64 instruction encoding lookup table generated at build time and 1,276 lines of C. The validator itself consists of 991 lines of C. SFI runtimes and validators have been shown to be formally verifiable by prior work [88, 89, 99], and could enable further TCB reduction. Our data sanitization layer only implements a limited set of boundary checks at this time and does not cover the full JSAPI interface; it currently consists of 650 lines of C++ definitions for various wrapper types and 80 lines of C++ to maintain the DOM reflector table.

7.4 Comparison with Ubercage

We contrast Mohabi with Ubercage. Although Ubercage is only a partial sandbox, it is still the most significant deployment of SFI techniques for JS engines currently in use.

Ubercage Design. Ubercage takes a fundamentally different approach to sandboxing vs. Mohabi. Rather than choosing a sandbox boundary and performing disaggregation all at once, it is building one incrementally and at a different layer than Mohabi—between the JS heap (and by extension, the Wasm heap), and the rest of the JS engine. While this has greatly simplified its deployment as performance overhead is very low ($\approx$ 1% [71]), its security guarantees are weaker.

Ubercage works by requiring JITted code to include SFI-style masks to restrict memory operations to a 1 TiB sandbox[4], using a code-pointer table to restrict forward-edge control flow to safe locations, and an external-pointer table to secure access to sensitive data outside the sandbox. Any AOT-compiled JS runtime code relies on a type system to prevent confused deputy attacks, with different types used to access sandboxed pointers, code-pointer tables, and external-pointer tables. However, Ubercage being a partial SFI scheme, does not protect data on the stack (including return addresses), and does not employ a binary validator, which is critical given the size of its TCB as discussed below. Ubercage treats the JS engine's AOT code (the GC, the JIT compiler, etc.), as well as JIT-compiled code, as trusted and within its TCB.

Ubercage has proposals to address some of this—a binary validator [75] (similar to Mohabi's) that ensures JIT-compiled code can never violate sandboxing guarantees, and prototypes for hardware-based isolation schemes (for hardware that supports it) to eliminate sandbox bypasses [72]. Like Mohabi, Ubercage must also apply sanitization checks to data originating from the sandbox; for Ubercage, this is any data from the JS heap. Importantly, given the size and complexity of Ubercage's chosen boundary, we believe that it is much more challenging to manually secure in comparison to the boundary chosen by Mohabi (which can leverage the auto-generated nature of DOM-JS glue code §4.3).

Ubercage Bypasses. Ubercage's phased rollout over multiple years has resulted in over 150 bugs to date [13], originating from both rigorous testing [34, 74] and real-world attacks [92]. While the number of such bugs has trended down as development progressed and bug-bounty programs helped identify missed checks, bugs still arise as a direct consequence of Ubercage's current design and choice of complex sandbox boundary. We highlight some of these bugs[5] in Figure 7 according to the JS engine component that was targeted, and contrast this with how such bugs would be mitigated by Mohabi. We analyze a few of these in detail below.

Case study 1: Bug 350292240 [93]. This is a vulnerability in Wasm function signature validation, caused by attackers being able to corrupt type metadata located in the sandbox. Wasm type signatures rely on runtime type checks to guarantee correctness, but these checks operate on sandboxed data, making them vulnerable to corruption. Unlike Mohabi, Ubercage considers the JIT compiler trusted, meaning that it must validate data coming from the untrusted JS heap. Ubercage misses a sanitization check here, allowing an attacker to generate buggy JIT code that accesses memory outside the sandbox.

Case study 2: Bug 338381304 [109]. This bug displays how attackers can corrupt the number of expected parameters in a JS function's stack frame, resulting in a miscomputation of stack size, and subsequent improper stack access. The bug is a result of Ubercage storing a trusted field for parameter count on the sandboxed heap, and not sanitizing it on use. This field is referenced when attempting to de-optimize a JIT-compiled function, to read the number of stack parameters it uses. When this value is corrupted, the stack frame is shrunk by the wrong size, and subsequent accesses use unintended values, causing type confusion. In contrast, Mohabi's design means that the entire C++ runtime is untrusted, ensuring that such mistakes are constrained to within the sandbox.

Case study 3: Bug 462217236 [116]. This bug describes how attackers can induce a use-after-free in the host as V8's GC relies on in-sandbox data to determine liveness of an object outside the sandbox. Specifically, the trusted JS runtime creates an object called a dispatch entry as part of the trusted stack-unwinding support for (`asm.js`) exceptions. In some execution paths, the only strong reference to the dispatch entry is from an object inside the sandbox. If the untrusted JS code uses in-sandbox memory corruption to remove the reference to this dispatch entry, the dispatch entry can get cleaned up by the GC. In this scenario, any subsequent exceptions thrown by JS code would be handled by the trusted runtime and use

[4] In practice, Ubercage uses a nested 4 GiB sandbox within a 1 TiB region for JavaScript objects, but this does not change its security guarantees.

[5] Note that we deliberately exclude bugs that emerged due to Ubercage's phased rollout over multiple years (i.e., bugs targeting unimplemented parts of Ubercage), and focus on bugs that arose due to Ubercage's design.

	Ubercage	Mohabi
JIT Compiler	Ubercage places V8's JIT compilers outside the sandbox, and the JITs are trusted to generate correctly sandboxed code. Multiple bugs were caused by the JIT missing masks for stores [92, 94] or missing sanitization checks on input coming from the sandbox [93].	Mohabi sandboxes SpiderMonkey's JIT compilers and any code generated/modified by JITs is checked by the binary validator prior to receiving executable page permissions (§6.3).
JS Runtime	Ubercage places V8's JS interpreter and runtime outside the sandbox. Multiple bugs [95, 109, 131, 132] have been found in these components due to missing sanitization checks on input coming from the sandbox.	Mohabi sandboxes SpiderMonkey's C++ runtime, checks it using the binary validator during sandbox instantiation (§5.4).
Garbage Collector (GC)	Ubercage places V8 GC outside the Ubercage sandbox boundary. Multiple use-after-free (UAF) vulnerabilities [116–118] were caused by the GC relying on untrusted sandbox data to determine liveness, resulting in premature memory reclamation.	Mohabi places SpiderMonkey's GC primarily inside the Mohabi sandbox. However, a portion of the GC responsible for reclaiming cyclic references [31] across the JS engine and DOM is outside the sandbox boundary. Mohabi utilizes split-allocation types (§4.2) to harden these interactions; sanitizing all data flow across this boundary is left for future work.
JS-DOM Boundary	Ubercage's sandbox boundary is fundamentally internal to the JS engine, and does not expose a security surface to the DOM, *assuming* the internal sandbox boundary does not have bugs.	Mohabi's design places the sandbox boundary outside the JS engine, allowing it to use the auto-generated glue layer between the DOM and JS engine to insert sanitization checks as shown in the example with DOM reflectors (§4.3). Mohabi's hypothesis is that this is a more securable and maintainable boundary for a JS engine sandbox, given the smaller attack surface.

Figure 7: **Comparison of the Ubercage and Mohabi sandbox boundaries.** *Ubercage's sandbox contains only the JavaScript heap-data memory region leaving the V8 JIT compilers, JavaScript runtime, and GC as trusted code outside the sandbox. In contrast, Mohabi places the entire SpiderMonkey engine in the sandbox. To guarantee security, both Ubercage and Mohabi must sanitize untrusted data crossing their respective sandbox boundaries. However, Mohabi presents a smaller attack surface, which we illustrate using examples of past Ubercage sandbox bypasses at the boundary of each component.*

the stale dispatch entry pointer from the stack. This UaF can result in an attacker obtaining control over the stack pointer.

8 Discussion and Future Work

Securing the JS Engine boundary. An important next step for Mohabi is to build a system that can check that all required boundary security checks have been added. While Mohabi illustrates how to secure this boundary with case studies such as the DOM reflector objects (§4.3), building and guaranteeing full coverage remains an open challenge. This is because the boundary between the browser and JS engine was not originally engineered with JS engine isolation in mind—it exposes low-level interfaces and makes pervasive use of shared data structures that are implicitly trusted by the browser. While we believe this next step is tractable (e.g., by automatically generating sanitizations in the glue layer), redesigning this boundary to be narrower can greatly simplify this task.

Taking Mohabi beyond x86-64. We focus on x86-64 in this paper as it dominates Firefox deployments on desktop devices [4]. However, RISC architectures such as ARM64 and RISC-V have gained usage on other platforms. Prior work [139] has shown that RISC ISAs lend themselves to efficient SFI schemes due to the availability of more general purpose registers and a fixed-length instruction encoding. However, as in the case of x86-64, prior SFI tools have mostly targeted 4 GiB sandboxes for ahead-of-time compiled code.

Optimizing SFI with Hardware Features. Modern CPUs provide hardware extensions [28, 29, 85, 86] for control-flow integrity and memory isolation that can be used by SFI schemes to reduce overheads [44, 133]. However, limited hardware availability and inconsistent operating system support make it impractical to rely entirely on these features alone. Systems that use JIT compilation, such as JS engines, can—in principle—partially overcome this by dynamically using the available hardware support for just-in-time code, while relying on software enforcement for ahead-of-time code.

9 Related Work

Prior work [26, 57, 70, 106, 107] has tried to secure JS engines with SFI. We analyze these attempts on two aspects: (1) *Full browser support*—does the design handle the complexities of integration with JS engines in production browsers? (2) *SFI completeness*—is the SFI scheme sound and does it come with a binary validator to minimize the TCB? We summarize our findings in Figure 8 and give more details in Figure 10 in the Appendix.

Broadly, academic work has made good progress on adapting sandboxing tools for JS engines; however, these cannot be applied to production browsers as they do not address the challenges discussed in §1. Industry efforts, in contrast, have optimized overhead by selectively applying some SFI techniques; however, being partial schemes, they are frequently bypassed. Thus, the challenge of a full sandbox for JS engines in a production-scale browser has not been tackled yet.

NaClJIT. NaClJIT [26] first demonstrated sandboxing JS engines by sandboxing the V8 engine. They compiled V8's runtime code using the Native Client (NaCl) SFI compiler [123, 140] and modified the V8 JavaScript JIT to only emit code that followed NaCl's SFI rules, along with an SFI validator to ensure that no checks were missing. While NaClJIT demonstrated the changes needed in SFI compilers to support sandboxing JIT engines, they found that their approach had high overheads of 51%–60% due to (1) the added

	NaClJIT [26]	RockJIT [106]	Ubercage [70]	JITCage [37]	Mohabi
Full browser support	No (Shell/JS engine)	No (Shell/JS engine)	Yes	Yes	Yes
Completeness of SFI scheme (with validator)	Yes	Yes	Partial guarantees (ad-hoc checks, bytecode validator, future binary validator [75])	No	Yes
Performance overhead[a]	51% (V8 bench, Sunspider)	14% (Octane2)	1% (JetStream, Speedometer)	Not reported	24.82% (JetStream), 24.43% (Speedometer)

[a]Different browsers, versions, benchmarks (Sunspider, JetStream, V8) used. This column is included only so readers can get a rough idea.

Figure 8: **Prior efforts to sandbox JavaScript engines.** *Academic efforts have focused on sandboxing JavaScript engines in shells/standalone JavaScript engines. Efforts from the industry, in production browsers, have focused on partial security mitigations. Mohabi, to our knowledge, is the first effort that overcomes all of these downsides.*

SFI runtime checks, (2) the longer compilation times of JITs that enforce SFI, and (3) time needed to run the SFI validator.

RockJIT. RockJIT [106] enforced a different security invariant—fine-grained control-flow integrity in the V8 JS engine. However, their approach required ensuring *memory write*s from the JS engine did not corrupt the data structures maintaining valid CFI targets. They also included a binary validator for this. The resulting system only imposed a 14% overhead on JavaScript engine performance. However, RockJIT (like NaClJIT) operated on the pre-2015 (i.e., single JIT), standalone V8 JS engine rather than in a browser, and thus did not have to address support for complex tiering or disaggregation. It also imposed a 4 GiB memory limit, making it inapplicable to present-day browsers.

Ubercage. In 2021, Chrome implemented partial SFI-style restrictions in Chrome's V8 JavaScript engine in the Ubercage [70] project. It represents the largest use of these techniques in a production JS engine today. We contrast Mohabi and Ubercage in detail in §7.4. Ubercage continues to evolve, and includes plans to leverage hardware support [64] (e.g., Intel MPK) to provide more robust security (as guaranteed by Mohabi using SFI) and incorporate a binary validator to check correctness [75] (similar to Mohabi). However, it is not clear that this incremental approach to adopting SFI will ultimately result in sandboxes that cannot be bypassed (§7.4).

JITCage. Safari's JavaScriptCore (JSC) employs a few SFI-style restrictions to stop specific attacks that leverage type confusions [69]. For example, attackers try to forge fake pointers to ArrayBuffers in JavaScript as it allows them to access large regions of memory. JSC combats this by placing some sensitive objects like ArrayBuffers in dedicated memory regions called Gigacages [57], with guarded memory accesses to these regions. JSC also relies on hardware features like SPRR [110, 125] to enforce *Write Xor Execute* protections, and Pointer Authentication Codes (PAC) [30] and recently Memory Tagging [114] for best-effort [36, 68] control-flow and memory-safety protections. However, these mitigations are partial, not directly attempting to constrain the effects of vulnerabilities. Attackers have been able to show that Safari's protections are all bypassable [90, 130] in multiple real-world exploit chains. [37, 38] offer more details on these mitigations.

Other hardening approaches. Formal verification has been used to prove the correctness of specific JS JIT optimizations [40, 127], or used to build (simpler) verified JS JITs [32, 33, 79, 103]—however more work is needed to scale these techniques to production browsers. NoJITSu [107] separates data types in JS engines into a handful of pools for which write-permissions can be selectively enabled as needed (leveraging Intel MPK for efficiency). Similar to Gigacage, V8 uses the hardened PartitionAlloc [46] to isolate sensitive objects such as ArrayBuffers [61]. JSC also previously used a best-effort [67, 136] StructureID randomization to prevent attackers from being able to forge fake objects in the heap. JS engines have historically used *Write Xor Execute* memory for some code-overwrite protections [50]; however, browsers have started to remove these for performance [51] and are opting to instead use hardware features such as SPRR and MPK where available [59, 60, 111]. JS engines may also implement constant blinding [112] to mitigate JIT spray attacks.

10 Conclusion

Browser vendors today are scrambling to fix the firehose of bugs found in JavaScript engines including those recently found by LLMs like Anthropic's Mythos. What's clear from this chaos is that modern JavaScript engines are notoriously complex systems that, while fast, are a seemingly inexhaustible source of memory-safety vulnerabilities. This is not fundamental though: Mohabi shows that browsers can isolate JavaScript engines using in-process sandboxing with modest performance overhead to eliminate the large JavaScript engine security surface that browsers are exposed to today.

Acknowledgments. We thank the reviewers for their insightful feedback, Samuel Groß for feedback on the Ubercage section of this work. This work was supported in part by NSF grants #2120696, #2120642, #2327337, #2146755, #2155235, #2327336, and gifts from Mozilla, Qualcomm, and Stanford FDCI and IOG Research Hub.

References

[1] Blink-V8 bindings generator. https://source.chromium.org/chromium/chromium/src/+/main:third_party/blink/renderer/bindings/scripts/bind_gen/README.md.

[2] Clean-up: Move MacroAssemblerSpecific::call to the MacroAssembler. https://bugzilla.mozilla.org/show_bug.cgi?id=1178770.

[3] DOM specification. https://dom.spec.whatwg.org/.

[4] Firefox Public Data Report. https://data.firefox.com/dashboard/hardware.

[5] Firefox-SpiderMonkey bindings generator. https://searchfox.org/firefox-main/source/dom/bindings/Codegen.py.

[6] HTML specification. https://html.spec.whatwg.org/.

[7] HTML specification for web workers. https://html.spec.whatwg.org/multipage/#toc-workers.

[8] JavaScriptCore x86-64 MacroAssembler backend. https://github.com/WebKit/WebKit/blob/main/Source/JavaScriptCore/assembler/MacroAssemblerX86_64.h.

[9] JetStream JavaScript and WebAssembly benchmark. https://browserbench.org/JetStream2.2/.

[10] Optimizing subroutines in assembly language: An optimization guide for x86 platforms. https://www.agner.org/optimize/optimizing_assembly.pdf.

[11] Speedometer browser benchmark. https://browserbench.org/Speedometer3.1/.

[12] SpiderMonkey x86-64 MacroAssembler backend. https://searchfox.org/firefox-main/source/js/src/jit/x64/MacroAssembler-x64.h.

[13] V8 sandbox bug tracker. https://issues.chromium.org/issues?q=hotlistid:4802478%20type:vulnerability%20-status:(duplicate%20%7C%20inactive%20%7C%20infeasible%20%7C%20intended_behavior%20%7C%20not_reproducible%20%7C%20obsolete).

[14] V8 x86-64 MacroAssembler backend. https://github.com/v8/v8/blob/main/src/codegen/x64/macro-assembler-x64.h.

[15] Web IDL specification. https://webidl.spec.whatwg.org/#.

[16] WebAssembly system interface. https://wasi.dev.

[17] WebKit-JSCore bindings generator. https://github.com/WebKit/WebKit/blob/main/Source/WebCore/bindings/scripts/generate-bindings.pl.

[18] V8: incorrect type information on Math.expm1. https://project-zero.issues.chromium.org/issues/42450781, 2018.

[19] Exploiting the Math.expm1 typing bug in V8. https://abiondo.me/2019/01/02/exploiting-math-expm1-v8/, 2019.

[20] CVE-2024-29943. https://bugzilla.mozilla.org/show_bug.cgi?id=1886849, 2024.

[21] Bug 2029801. https://github.com/mozilla-firefox/firefox/commit/7c09940954fe5cbc235efcdf2f1ebeaf607ef6d8, 2026.

[22] Bug 2024916 - pass arguments array object through variant. https://github.com/mozilla-firefox/firefox/commit/dc1d45f126a06166709ad659f1b862795484c75a, 2026.

[23] Bug 2029735 - handle utf8 vs latin1 comparisons correctly in utf8equalschars. https://github.com/mozilla-firefox/firefox/commit/1ce59f141a1fe03e880e1a7ba723206b5b3126ba, 2026.

[24] Bug 2029754 - check chunk to be decommitted is still in the empty chunks list. https://github.com/mozilla-firefox/firefox/commit/13cc6379c01647796db0dc6868d7bc47b6b0a349, 2026.

[25] Android. Advanced Protection Mode. https://support.google.com/android/answer/16339980.

[26] Jason Ansel, Petr Marchenko, Úlfar Erlingsson, Elijah Taylor, Brad Chen, Derek L. Schuff, David Sehr, Cliff L. Biffle, and Bennet Yee. Language-independent sandboxing of Just-in-Time compilation and self-modifying code. In *PLDI*. ACM, 2011.

[27] Apple. Lockdown mode. https://support.apple.com/en-us/105120.

[28] The Guarded Control Stack. https://developer.arm.com/documentation/ddi0487/mb/-Part-D-The-AArch64-System-Level-Architecture/-Chapter-D11-The-Guarded-Control-Stack.

[29] Pointer Authentication on ARM. https://learn.arm.com/learning-paths/servers-and-cloud-computing/pac/pac.

[30] Brandon Azad. Examining Pointer Authentication on the iPhone XS. https://googleprojectzero.blogspot.com/2019/02/examining-pointer-authentication-on.html, 2019.

[31] David F Bacon and Vadakkedathu T Rajan. Concurrent cycle collection in reference counted systems. In *European Conference on Object-Oriented Programming*, pages 207–235. Springer, 2001.

[32] Aurèle Barrière, Sandrine Blazy, Olivier Flückiger, David Pichardie, and Jan Vitek. Formally verified speculation and deoptimization in a JIT compiler. In *POPL*. ACM, 2021.

[33] Aurèle Barrière, Sandrine Blazy, and David Pichardie. Formally verified native code generation in an effectful JIT: Turning the CompCert backend into a formally verified JIT compiler. In *POPL*. ACM, 2023.

[34] Nils Bars, Lukas Bernhard, Moritz Schloegel, and Thorsten Holz. Empirical security analysis of software-based fault isolation through controlled fault injection. In *Proceedings of the 2025 ACM SIGSAC Conference on Computer and Communications Security*, pages 2639–2652, 2025.

[35] Adam Barth, Collin Jackson, Charles Reis, TGC Team, et al. The security architecture of the Chromium browser. In *Technical report*. Stanford University, 2008.

[36] Ian Beer. Blasting past Webp - an analysis of the NSO BLASTPASS iMessage exploit. https://googleprojectzero.blogspot.com/2025/03/blasting-past-webp.html, 2025.

[37] Eloi Benoist-Vanderbeken and Fabien Perigaud. Wen eta jb? a 2 million dollars problem. https://www.sstic.org/media/SSTIC2019/SSTIC-actes/WEN_ETA_JB/SSTIC2019-Article-WEN_ETA_JB-benoist-vanderbeken_perigaud.pdf, 2019.

[38] Eloi Benoist-Vanderbeken and Fabien Perigaud. An apple a day keeps the exploiter away. https://www.sstic.org/media/SSTIC2022/SSTIC-actes/an_apple_a_day/SSTIC2022-Article-an_apple_a_day-benoist-vanderbeken_perigaud.pdf, 2022.

[39] Fraser Brown, Shravan Narayan, Riad S Wahby, Dawson Engler, Ranjit Jhala, and Deian Stefan. Finding and preventing bugs in JavaScript bindings. In *2017 IEEE Symposium on Security and Privacy (SP)*, pages 559–578. IEEE, 2017.

[40] Fraser Brown, John Renner, Andres Noetzli, Sorin Lerner, Hovav Shacham, and Deian Stefan. Towards a verified range analysis for JavaScript JITs. In *PLDI*. ACM, 2020.

[41] Amy Burnett, Patrick Biernat, and Markus Gaasedelen. Weaponization of a JavaScriptCore vulnerability. https://blog.ret2.io/2018/07/11/pwn2own-2018-jsc-exploit/, 2018.

[42] Lee Campbell. Exploiting NVMAP to escape the Chrome sandbox - CVE-2014-5332. https://projectzero.google/2015/01/exploiting-nvmap-to-escape-chrome.html, 2015.

[43] Nicholas Carlini, Antonio Barresi, Mathias Payer, David Wagner, and Thomas R Gross. Control-Flow bending: On the effectiveness of Control-Flow integrity. In *24th USENIX Security Symposium (USENIX Security 15)*, pages 161–176, 2015.

[44] Xiangdong Chen, Zhaofeng Li, Tirth Jain, Vikram Narayanan, and Anton Burtsev. Limitations and opportunities of modern hardware isolation mechanisms. In *2024 USENIX Annual Technical Conference (USENIX ATC 24)*, pages 349–368, 2024.

[45] Yi Chien, Vlad-Andrei Bădoiu, Yudi Yang, Yuqian Huo, Kelly Kaoudis, Hugo Lefeuvre, Pierre Olivier, and Nathan Dautenhahn. Civscope: Analyzing potential memory corruption bugs in compartment interfaces. In *Proceedings of the 1st Workshop on Kernel Isolation, Safety and Verification*, KISV '23, page 33–40, New York, NY, USA, 2023. Association for Computing Machinery.

[46] Chromium. Efficient and safe allocations everywhere! https://blog.chromium.org/2021/04/efficient-and-safe-allocations-everywhere.html.

[47] Emma Connor, Tyler McDaniel, Jared M Smith, and Max Schuchard. PKU pitfalls: Attacks on PKU-based memory isolation systems. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 1409–1426, 2020.

[48] Jack Dates. 32 bits, 32 gigs, 1 click... https://blog.ret2.io/2021/06/02/pwn2own-2021-jsc-exploit/, 2021.

[49] Jack Dates. Exploiting Intel graphics kernel extensions on macOS. https://blog.ret2.io/2022/06/29/pwn2own-2021-safari-sandbox-intel-graphics-exploit/, 2022.

[50] Jan de Mooij. WX̂ JIT-code enabled in Firefox. https://jandemooij.nl/blog/wx-jit-code-enabled-in-firefox/, December 2019.

[51] Jan de Mooij. Consider disabling code memory protection in the content process. https://bugzilla.mozilla.org/show_bug.cgi?id=1835876, May 2023.

[52] Jan de Mooij, Matthew Gaudet, Iain Ireland, Nathan Henderson, and J Nelson Amaral. CacheIr: The benefits of a structured representation for inline caches. In *Proceedings of the 20th ACM SIGPLAN International Conference on Managed Programming Languages and Runtimes*, pages 34–46, 2023.

[53] Liang Deng, Qingkai Zeng, and Yao Liu. ISboxing: An instruction substitution based data sandboxing for x86 untrusted libraries. In *IFIP International Information Security and Privacy Conference*, pages 386–400. Springer, 2015.

[54] Alan Donovan, Robert Muth, Brad Chen, and David Sehr. PNaCl: Portable native client executables. *Google White Paper*, 2010.

[55] Alexis Engelke. Fadec — fast decoder for x86-32 and x86-64 and encoder for x86-64. https://github.com/aengelke/fadec.

[56] Ivan Fratric. Bypassing mitigations by attacking JIT server in Microsoft Edge. https://raw.githubusercontent.com/google/p0tools/master/JITServer/JIT-Server-whitepaper.pdf#page=13.

[57] Gigacage. https://phakeobj.netlify.app/posts/gigacage/, 2019.

[58] Sergei Glazunov. Project zero - in-the-wild series: Chrome exploits. https://googleprojectzero.blogspot.com/2021/01/in-wild-series-chrome-exploits.html, January 2021.

[59] Google Chrome. enable pkey protections for js heap. https://chromium-review.googlesource.com/c/v8/v8/+/4793851.

[60] Google Chrome. Support fast WX̂ permission switching on apple silicon. https://chromium-review.googlesource.com/c/v8/v8/+/3579303.

[61] Google Chrome. Use PA for ArrayBufferAllocator when malloc is PA. https://chromium-review.googlesource.com/c/v8/v8/+/6651069.

[62] Google Chrome. V8 Exploit Tracker. https://docs.google.com/document/d/1njn2dd5_6PB7oZGTmkmoihYnVcJEgRwEFxhHnGoptLk. Accessed: Nov 2025.

[63] Google Chrome. Use-after-freedom: MiraclePtr. https://security.googleblog.com/2022/09/use-after-freedom-miracleptr.html, September 2022.

[64] Google Chrome. V8 hardware support. https://docs.google.com/document/d/12MsaG6BYRB-jQWNkZiuM3bY8X2B2cAsCMLLdgErvK4c, February 2024.

[65] Brian Grinstead, Christian Holler, and Frederik Braun. Behind the scenes hardening Firefox with Claude Mythos Preview. https://hacks.mozilla.org/2026/05/behind-the-scenes-hardening-firefox/, May 2026.

[66] Samuel Groß. JITSploitation i: A JIT bug. https://googleprojectzero.blogspot.com/2020/09/jitsploitation-one.html, 2020.

[67] Samuel Groß. JITSploitation ii: Getting read/write. https://googleprojectzero.blogspot.com/2020/09/jitsploitation-two.html, 2020.

[68] Samuel Groß. PAC and JIT hardening bypass. https://project-zero.issues.chromium.org/issues/42451144/, 2020.

[69] Samuel Groß. Attacking JavaScript engines - a case study of JavaScriptCore and CVE-2016-4622. https://phrack.org/issues/70/3, 2021.

[70] Samuel Groß. V8 sandbox - high-level design doc. https://docs.google.com/document/d/1FM4fQmIhEqPG8uGp5o9A-mnPB5BOeScZYpkHjo0KKA8, July 2021.

[71] Samuel Groß. The V8 sandbox. https://v8.dev/blog/sandbox, April 2024.

[72] Samuel Groß. Experiment with hardware support for the V8 sandbox. https://issues.chromium.org/issues/350324877, May 2025.

[73] Samuel Groß. JS engine security in 2025: New bugs, new defenses. https://powerofcommunity.net/2025/slide/s-92443.pdf, November 2025.

[74] Samuel Groß. V8: pkey-based sandbox fuzzer. https://chromium-review.googlesource.com/c/v8/v8/+/7580844, April 2026.

[75] Samuel Groß. The V8 sandbox from compiler correctness to runtime containment. https://popl26.sigplan.org/details/prisc-2026-papers/1/The-V8-Sandbox-From-Compiler-Correctness-to-Runtime-Containment, January 2026.

[76] Samuel Groß and Amy Burnett. Attacking JavaScript engines in 2022. https://saelo.github.io/presentations/offensivecon_22_attacking_javascript_engines.pdf, 2022.

[77] Qihoo 360 Guang Gong of Alpha Team. Security: race condition lead to many fatal Error D in WebAssembly.validate. https://issues.chromium.org/issues/40089891, 2017.

[78] Qihoo 360 Guang Gong of Alpha Team. V8 type confusion in web assembly. https://issues.chromium.org/issues/40088842, 2017.

[79] Shu-yu Guo and Jens Palsberg. The essence of compiling with traces. In *POPL*. ACM, 2011.

[80] Andreas Haas, Andreas Rossberg, Derek L Schuff, Ben L Titzer, Michael Holman, Dan Gohman, Luke Wagner, Alon Zakai, and JF Bastien. Bringing the web up to speed with WebAssembly. *Communications of the ACM*, 61(12):107–115, 2018.

[81] Bobby Holley. The zero-days are numbered. https://blog.mozilla.org/en/privacy-security/ai-security-zero-day-vulnerabilities/, April 2026.

[82] Hong Hu, Zheng Leong Chua, Zhenkai Liang, and Prateek Saxena. Identifying arbitrary memory access vulnerabilities in privilege-separated software. In *European Symposium on Research in Computer Security*, pages 312–331. Springer, 2015.

[83] Iain Ireland. A new regexp engine in SpiderMonkey. https://hacks.mozilla.org/2020/06/a-new-regexp-engine-in-spidermonkey/.

[84] Bumjin Im, Fangfei Yang, Chia-Che Tsai, Michael LeMay, Anjo Vahldiek-Oberwagner, and Nathan Dautenhahn. The endokernel: Fast, secure, and programmable subprocess virtualization. *arXiv preprint arXiv:2108.03705*, 2021.

[85] Intel® 64 and IA-32 architectures software developer's manual, 2020.

[86] Intel Corporation. A technical look at Intel® Control-flow Enforcement Technology. https://www.intel.com/content/www/us/en/developer/articles/technical/technical-look-control-flow-enforcement-technology.html, 2020.

[87] Abhinav Jangda, Bobby Powers, Emery D Berger, and Arjun Guha. Not so fast: Analyzing the performance of WebAssembly vs. native code. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 107–120, 2019.

[88] Evan Johnson, Evan Laufer, Zijie Zhao, Dan Gohman, Shravan Narayan, Stefan Savage, Deian Stefan, and Fraser Brown. WaVe: a verifiably secure WebAssembly sandboxing runtime. In *2023 IEEE Symposium on Security and Privacy (SP)*, pages 2940–2955. IEEE, 2023.

[89] Evan Johnson, David Thien, Yousef Alhessi, Shravan Narayan, Fraser Brown, Sorin Lerner, Tyler McMullen, Stefan Savage, and Deian Stefan. Доверя́й, но проверя́й: SFI safety for native-compiled Wasm. In *NDSS*. Internet Society, 2021.

[90] Kaspersky. Operation triangulation. https://securelist.com/operation-triangulation-catching-wild-triangle/110916/, October 2023.

[91] Matthew Kolosick, Shravan Narayan, Evan Johnson, Conrad Watt, Michael LeMay, Deepak Garg, Ranjit Jhala, and Deian Stefan. Isolation without taxation: near-zero-cost transitions for WebAssembly and SFI. *Proceedings of the ACM on Programming Languages*, 6(POPL):1–30, 2022.

[92] Clement Lecigne and Benoît Sevens of Google Threat Analysis Group. [0-day] V8 sandbox bypass via turbofan. https://issues.chromium.org/issues/420637585, 2025.

[93] Seunghyun Lee. V8 sandbox bypass: Aar/w via generic function table call-indirect rtt check bypass. https://issuetracker.google.com/issues/350292240, 2025.

[94] Seunghyun Lee. V8 sandbox bypass: Aaw via clobbered i32 high word on return value in liftoff. https://issuetracker.google.com/issues/421403261, 2025.

[95] Seunghyun Lee. V8 sandbox bypass: Control flow hijack via switch-case over corrupted messagetemplate enum value. https://issuetracker.google.com/issues/390816209, 2025.

[96] Hugo Lefeuvre, Nathan Dautenhahn, David Chisnall, and Pierre Olivier. Sok: Software compartmentalization. In *2025 IEEE Symposium on Security and Privacy (SP)*, pages 3107–3126. IEEE, 2025.

[97] Stephen McCamant and Greg Morrisett. Evaluating SFI for a CISC architecture. In *USENIX Security Symposium*, volume 10, pages 209–224, 2006.

[98] Microsoft. Enhanced Security Mode. https://www.microsoft.com/en-us/edge/features/enhanced-security-mode.

[99] Greg Morrisett, Gang Tan, Joseph Tassarotti, Jean-Baptiste Tristan, and Edward Gan. RockSalt: better, faster, stronger SFI for the x86. In *Proceedings of the 33rd ACM SIGPLAN conference on Programming Language Design and Implementation*, pages 395–404, 2012.

[100] Mozilla. GC rooting guide. https://developer.mozilla.org.cach3.com/en-US/docs/SpiderMonkey/GC_Rooting_Guide.

[101] Mozilla. SpiderMonkey — Firefox source docs. https://firefox-source-docs.mozilla.org/js/index.html.

[102] Project Fission. https://wiki.mozilla.org/Project_Fission, 2019.

[103] Magnus O. Myreen. Verified just-in-time compiler on x86. In *POPL*. ACM, 2010.

[104] Shravan Narayan, Craig Disselkoen, Tal Garfinkel, Nathan Froyd, Eric Rahm, Sorin Lerner, Hovav Shacham, and Deian Stefan. Retrofitting fine grain isolation in the firefox renderer. In *Proceedings of the 29th USENIX Conference on Security Symposium*. USENIX Association, 2020.

[105] Shravan Narayan, Tal Garfinkel, Evan Johnson, Zachary Yedidia, Yingchen Wang, Andrew Brown, Anjo Vahldiek-Oberwagner, Michael LeMay, Wenyong Huang, Xin Wang, et al. Segue & colorguard: Optimizing SFI performance and scalability on modern architectures. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1*, pages 987–1002, 2025.

[106] Ben Niu and Gang Tan. RockJIT: Securing Just-in-Time compilation using modular control-flow integrity. In *CCS*. ACM, 2014.

[107] Taemin Park, Karel Dhondt, David Gens, Yeoul Na, Stijn Volckaert, and Michael Franz. NoJITsu: Locking down JavaScript engines. In *NDSS*. Internet Society, 2020.

[108] Manfred Paul. CVE-2024-2887: A Pwn2Own winning bug in Google Chrome. https://www.zerodayinitiative.com/blog/2024/5/2/cve-2024-2887-a-pwn2own-winning-bug-in-google-chrome, 2024.

[109] pazdnikov2005@gmail.com. V8 sandbox bypass: stack corruption due to parameter count mismatch. https://issuetracker.google.com/issues/338381304, 2024.

[110] Sven Peter. Apple silicon hardware secrets: Sprr and guarded exception levels (gxf). https://blog.svenpeter.dev/posts/m1_sprr_gxf/, 2021.

[111] Nicolas B. Pierron. Firefox pkey-enabled prototype. https://github.com/nbp/gecko-dev/tree/bugzilla_1886557_prototype, 2024.

[112] Nicolas B. Pierron. Make JIT spraying implausible. https://bugzilla.mozilla.org/show_bug.cgi?id=1886557, 2024.

[113] Filip Pizlo. Speculation in JavaScriptCore. https://webkit.org/blog/10308/speculation-in-javascriptcore/, 2020.

[114] Marcus Plutowski. [libpas] implement primary support for MTE. https://github.com/WebKit/WebKit/pull/50687, 2025.

[115] Vignesh S Rao. Writeup for CVE-2019-11707. https://vigneshsrao.github.io/posts/writeup/, 2019.

[116] Krishna Ravishankar. V8 sandbox bypass: Aaw/pc control via dispatch entry uaf during instantiateasmjs by hijacking start. https://issues.chromium.org/issues/462217236, 2025.

[117] Krishna Ravishankar. V8 sandbox bypass: Aaw/pc control via jsdispatchentry uaf. https://issues.chromium.org/issues/443772809, 2025.

[118] Krishna Ravishankar. V8 sandbox bypass: Referencing non-shared heap data across isolates leads to uaf -> aaw/pc control. https://issues.chromium.org/issues/444865195, 2025.

[119] Charles Reis, Alexander Moshchuk, and Nasko Oskov. Site isolation: process separation for web sites within the browser. In *Proceedings of the 28th USENIX Conference on Security Symposium*. USENIX Association, 2019.

[120] ESET Research. Romcom exploits Firefox and windows zero days in the wild. https://www.welivesecurity.com/en/eset-research/romcom-exploits-firefox-and-windows-zero-days-in-the-wild/, 2024.

[121] Andreas Rossberg. Memory64 proposal for WebAssembly. https://github.com/WebAssembly/memory64, 2020.

[122] David Schrammel, Samuel Weiser, Richard Sadek, and Stefan Mangard. Jenny: Securing syscalls for PKU-based memory isolation systems. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 936–952, 2022.

[123] David Sehr, Robert Muth, Cliff Biffle, Victor Khimenko, Egor Pasko, Karl Schimpf, Bennet Yee, and Brad Chen. Adapting software fault isolation to contemporary CPU architectures. In *SEC*. USENIX, 2010.

[124] Igor Sheludko and Santiago Aboy Solanes. Pointer compression in V8. https://v8.dev/blog/pointer-compression, March 2020.

[125] Siguza. APRR. https://blog.siguza.net/APRR/, 2019.

[126] Ben Smith. Wasm memory64: Bounds-checking strategies. https://github.com/WebAssembly/memory64/issues/3, April 2020.

[127] Naomi Smith, Abhishek Sharma, John Renner, David Thien, Fraser Brown, Hovav Shacham, Ranjit Jhala, and Deian Stefan. Icarus: Trustworthy Just-in-Time compilers with symbolic meta-execution. In *Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles*, pages 473–487, 2024.

[128] Chengyu Song, Chao Zhang, Tielei Wang, Wenke Lee, and David Melski. Exploiting and protecting dynamic code generation. In *NDSS*, 2015.

[129] Maddie Stone. Bad binder: Android in-the-wild exploit. https://projectzero.google/2019/11/bad-binder-android-in-wild-exploit.html, 2019.

[130] Maddie Stone. 2022 0-day in-the-wild exploitation...so far. https://googleprojectzero.blogspot.com/2022/06/2022-0-day-in-wild-exploitationso-far.html, 2022.

[131] v8sbxfuzz. V8 sandbox bypass: UB in WebAssemblyMemoryGrow because AddressType is constructed from on-heap data. https://issuetracker.google.com/issues/390453039, 2025.

[132] v8sbxfuzz. V8 sandbox bypass: UB in WebAssemblyTableGet because AddressType is constructed from on-heap data. https://issuetracker.google.com/issues/390441099, 2025.

[133] Anjo Vahldiek-Oberwagner, Eslam Elnikety, Nuno O Duarte, Michael Sammler, Peter Druschel, and Deepak Garg. ERIM: Secure, efficient in-process isolation with protection keys (MPK). In *28th USENIX Security Symposium (USENIX Security 19)*, pages 1221–1238, 2019.

[134] Robert Wahbe, Steven Lucco, Thomas E Anderson, and Susan L Graham. Efficient software-based fault isolation. In *Proceedings of the fourteenth ACM symposium on Operating systems principles*, pages 203–216, 1993.

[135] Nan Wang and Chen Ziling. Enhanced Insecurity Mode: 23 RCEs in Edge's "Safe" WebAssembly interpreter. https://www.offensivecon.org/speakers/2026/nan-wang-and-ziling-chen.html, 2026.

[136] Yong Wang. Thinking outside the JIT compiler: Understanding and bypassing StructureID randomization with generic and old-school methods. https://i.blackhat.com/eu-19/Thursday/eu-19-Wang-Thinking-Outside-The-JIT-Compiler-Understanding-And-Bypassing-StructureID-Randomization-With-Generic-And-Old-School-Methods.pdf.

[137] Robert NM Watson, Jonathan Woodruff, Peter G Neumann, Simon W Moore, Jonathan Anderson, David Chisnall, Nirav Dave, Brooks Davis, Khilan Gudka, Ben Laurie, et al. Cheri: A hybrid capability-system architecture for scalable software compartmentalization. In *2015 IEEE Symposium on Security and Privacy*, pages 20–37. IEEE, 2015.

[138] Jeff Xu. Introduce mseal. https://lwn.net/Articles/960465/, 2024.

[139] Zachary Yedidia. Lightweight Fault Isolation: Practical, efficient, and secure software sandboxing. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, pages 649–665, 2024.

[140] Bennet Yee, David Sehr, Gregory Dardyk, J Bradley Chen, Robert Muth, Tavis Ormandy, Shiki Okasaka, Neha Narula, and Nicholas Fullagar. Native client: A sandbox for portable, untrusted x86 native code. *Communications of the ACM*, 53(1):91–99, 2010.

Appendix

We include some additional performance and security evaluations beyond what is in the paper below.

We report the speed of Mohabi's validator in Table 3, the benefit of Mohabi's optimizations and overheads of Mohabi on the full SPEC 2017 benchmark in Figure 9—the body of the paper only reports performance on the NaCl supported subset of benchmarks (§7.2).

Results. Broadly, we see that Mohabi's validator outperforms NaCl's validator. While this is in part due to careful optimizations we included, we attribute this speedup primarily because we rely on a modern high-performance x86-64 decoder [55]. We also see that Mohabi optimizations were able to speed up Mohabi by 6%–8%.

Benchmark	MH-LFI 4 GiB	MH-LFI Large	NaCl
deepsjeng_r	126.70	100.70	82.77
imagick_r	148.61	119.08	95.08
lbm_r	118.93	98.32	79.26
leela_r	147.03	122.81	86.07
mcf_r	112.05	90.50	72.78
nab_r	123.70	98.93	83.29
namd_r	154.33	127.79	89.38
omnetpp_r	167.81	145.00	101.57
parest_r	161.15	130.44	95.50
povray_r	157.40	133.75	83.07
xz_r	126.37	106.35	78.10
Average	140.37	115.79	86.08

Table 3: Validator throughput comparison (MiB/s)

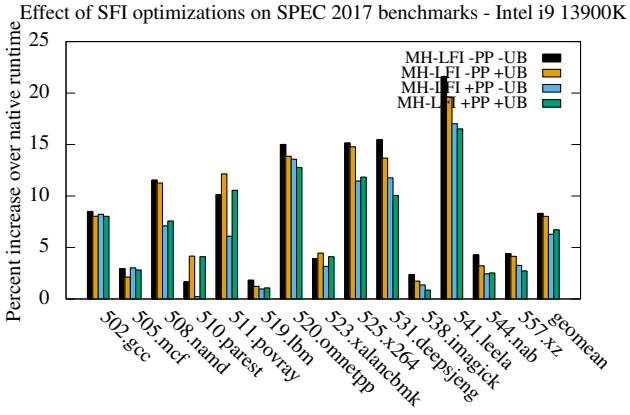


Figure 9: Breakdown of benefits from SFI optimizations. PP is prefix padding and UB is unaligned branches. Unaligned branches are primarily a binary size optimization as they have little-to-no positive effect for runtime performance.

We also provide a more detailed comparison of prior work and existing industry efforts to secure JS engines using SFI-style techniques in Figure 10.

	NaClJIT [26]	RockJIT [106]	NoJITsu [107]	Ubercage [70]	JITCage (Gigacage + PAC + SPRR) [37]	Mohabi
Isolates	pre-2015 V8 JS engine	pre-2015 V8 JS engine	SpiderMonkey JIT-compiled code	V8 JS engine	JSC JS engine	SpiderMonkey JS engine
Runs in full browser	N (Shell/JS engine)	N (Shell/JS engine)	N (Shell/JS engine)	Y	Y	Y
Hardware Independent (MPK, PAC, etc.)	Y	Y	N	Partial (CFI relies on CET/PAC)[a]	N (Unnecessary)[b]	Y
Mem Guards in JS RT (C++)	Y	Y	Y	Partial (for JS/Wasm objects only)	N (for sensitive objects)	Y
Mem Guards in JS JIT-compiled code	Y	Y	Partial (non-primitive JS objects only)	Y	N (for sensitive objects)	Y
Tainted data checks	N	N	N	Partial (in progress)	N	Partial (Future work)
SFI binary validator	Y	Y	N	N (Future work [75])	N	Y
Prevents heap OOB writes (throughout engine)	Y (NaCl)	Y	Partial (MPK for JS engine)[c]	Partial (for JS/Wasm objects only)	N (for sensitive objects)	Y (MH-LFI)
Prevents heap OOB reads (throughout engine)	Y (NaCl)	N[d]	Partial (MPK for JS engine)	Partial (for JS/Wasm objects only)	N (for sensitive objects)	N (Unnecessary)
Forward-edge control-flow protections	Y (NaCl)	Y (Fine-grained CFI)	N	Partial - code pointer tables, hardware features (IBT/PAC)	Partial (Bypassable PAC)	Y (MH-LFI)
Backward-edge control-flow protections	Y (NaCl)	Y (Bypassable CFI [43])	N	Partial - hardware features (Shadow Stack/PAC)	Y (Bypassable PAC)	Y (MH-LFI)
Memory limit	4 GiB	4 GiB	None	1024 GiB	60 GiB-64 GiB	256 GiB (Any chosen limit)
Unusable Virtual Memory due to Guard Pages	80 GiB	4 GiB	None	32 GiB	0 GiB-4 GiB	4 GiB
Safely handles W & X pages	Y	Y	N (Unnecessary)	Y (mseal [138])	Y (SPRR [110])	Y
JS benchmark overhead[e]	51% (V8 bench, Sunspider)	14% (Octane2)	5% (Longspider)	1% (JetStream, Speedometer)	Not reported	24.82% (JetStream), 24.43% (Speedometer)

[a] Ubercage relies on a few hardware features such as PAC and CET for completeness of their SFI scheme.
[b] JITCage is designed solely for Apple machines running ARM64, so it is not expected to be backwards compatible
[c] Protections exist only within the JS engine.
[d] Pre-2015 V8 did not have site isolation and had in-process secrets.
[e] Different browsers, versions, benchmarks (Sunspider, JetStream, V8) used. This column is included only so readers can get a rough idea.

Figure 10: **Prior efforts to sandbox JavaScript engines.** *Academic efforts have focused on sandboxing JavaScript engines in shells/standalone JavaScript engines. Efforts from the industry, in production browsers, have focused on partial security mitigations. Mohabi, to our knowledge, is the first effort that overcomes all of these downsides. For the industry efforts, we note that this represents the state of defenses at the time of the paper, and these may continue to evolve.*

Ichnaea: A Framework for Precise Tracking of Memory Objects

Samad Haque[†], Sibin Mohan[†], Aaron Paulos[‡] *and* Partha Pal[‡]
[†]*The George Washington University,* [‡]*RTX BBN Technologies*
{sam.ulhaque, sibin.mohan}@gwu.edu, {aaron.paulos, partha.pal}@rtx.com

Abstract

Tracing memory objects (*who* accessed[1] *what* object and *when*) is often important for understanding the runtime behavior of modern software. This type of rich per-access metadata can aid in debugging, tracing, forensics and other tasks. Collecting this information is non-trivial since it will be either be incomplete or requires heavy instrumentation and/or hardware support and likely adds significant runtime overheads (*e.g.,* Intel Pin or Valgrind slow programs down by $10-100x$).

We present `Ichnaea`[2], a purpose-built, precise and complete framework based on memory protection keys (MPK) that delivers context-rich object events at *very low cost* to the application. `Ichnaea` is dormant until one of the objects of interest (ObjOfInterest) is read or written to — at which point it logs any access attempts and changes to the ObjOfInterest along with rich context information ("*who is attempting access?*", "*what changes, if any, are being applied?*") before returning control to the application. In general `Ichnaea` reduces the tracing overheads by $10-60x$ when compared to the widely used framework Intel Pin , while still capturing *precise*, per-access information needed to diagnose memory vulnerabilities, performance hot-spots and subtle concurrency errors.

1 Introduction

> "*Every move you make, every bond you break, every step you take, I'll be watching you.*"
> — Police, Every Breath You Take (1983).

Modern software written in low-level languages (*e.g.,* C/C++) continues to grow in both size and complexity while relying on manual memory management, exposing raw pointers and permitting (unrestricted) indirect control flow. For instance, `nginx` [41], a widely deployed web server application written in C now exceeds 160 KLoC[3]. As codebases expand, maintenance becomes increasingly difficult, particularly when reasoning about subtle control-flow anomalies and memory-safety violations that stem from unintended or rogue *object accesses* [1]. Often control flow is not expressed explicitly but is data-driven — function pointers, callback tables and state-carrying objects determine which code runs next (*e.g.,* `nginx`, `postgresql` [44], *etc.*). For example `ngx_http_request_t` is a central, state-carrying object that guides request control-flow across nginx's event-driven phase pipeline. Tracing *accesses* to such objects can be quite challenging as they are hard to find due to the large number of *control and data-flow indirections*. These issues are especially exacerbated in multi-threaded codebases where in order to understand parallel object access patterns, one is either forced to serialize applications [28] or deal with incomplete information [15]. It is difficult to make headway in any of the above without tracing critical data or control flow objects with high degrees of indirection — all of which requires the ability to track *pointers*, including *function pointers* and *who (*`function`, `thread_id`, `user_id` *) is reading from/writing to them* and *when*.

Hence, there is a need for more *precise tracing* of object accesses (*e.g.,* reads/writes) in memory. Existing approaches for object tracing fall into two broad classes: *(a)* static and *(b)* dynamic analysis. Static techniques, *e.g.,* symbolic execution [6, 23] or compile-time instrumentation [10] struggle in practice, especially for large codebases. Symbolic execution suffers from path explosion (*e.g.,* [24, 25]), while precise compile-time instrumentation [38] struggles with undecidable problems like the difficulty in statically locating all access sites, especially under pointer aliasing and indirect control flow (see section 2.1.1). Figure 1 depicts a simple program where memory accesses can either be missed or over-approximated by a static analyzer — the object, `GLOBAL`, is accessed via complex control flows (function pointers based on user input) and data indirection (pointer to dynamically allocated object, passed as function argument). Such patterns

[1] Read or write
[2] "Ichnaea" is the Greek Goddess of tracing and tracking
[3] Measured using *cloc* [2] on `nginx` [35] commit 6ed1188.

```
int GLOBAL; % ObjectOfInterest

typedef void (*func_ptr)(struct ctrl);
func_ptr funcArr[] = { &foo, &bar, ... };

int main(int argc, char **argv) {
    struct ctrl A { int * reference , ... };
    A.reference = &GLOBAL;
    int idx = atoi(argv[1]);
    funcArr[idx](A);
}
```

```
void foo(struct ctrl prm) {
    printf("%d\n", *(prm.reference));
    ...
}

void bar(struct ctrl prm) {
    *(prm.reference) = 10;
    ...
}
```

Figure 1: An example of object access through indirect control flow manipulation. Such access are hard to find statically *e.g.,* knowing that the object `GLOBAL` is written to by the function `bar()`

are common in software, often leading to subtle memory-safety violations, *e.g.,* CVE-2021-23017 [1], where a single out-of-bounds write in the request resolver (triggered through intricate pointer arithmetic) corrupted a heap object and enabled denial-of-service (DOS) and remote code-execution (RCE) attacks. Such off-by-one errors, especially when memory (*e.g.,* object reference) is shared across multiple components in a multi-threaded program, remain pervasive and are notoriously difficult to detect using conventional debugging or static analysis techniques.

Dynamic techniques can avoid many of these issues since the information about the actual object/function is available at access time. These methods often introduce high overheads as they usually involve tracking almost *all loads/stores* using *(i)* compile-time instrumentation [19] or *(ii)* runtime instrumentation (*e.g.,* Pin [30], DynamoRIO [8]) — all of which slow down execution by 10–100×, making them unsuitable for long-running applications, test-suites, benchmarks or even for fuzzing-type methods [39]. Tracking accesses via the backing memory (*e.g.,* hardware debug registers [18] or virtual-memory mechanisms [28]) reduces overheads but is fundamentally constrained as such techniques either support only a small number of objects or suffer from false sharing [45], causing missed accesses in multi-threaded environments (see section 2) thus making the approach *lossy*.

Hence, current dynamic approaches are either prohibitively slow or fundamentally lossy, making them unsuitable in practice. More importantly, none provide **precise** (tracing singular objects instead of blocks of memory or all loads/stores), **complete**[4] (tracing accesses from application, external libraries and the kernel) and **lossless** (logging all accesses without misses) tracing of memory objects in one package. We found no tool that meets all the above requirements in one package.

We present `Ichnaea`, a tracing framework that uses the virtual-memory mechanism along with Memory Protection Keys [43] (MPKs) for *precise*, *lossless* and *complete* memory object tracing of programs written in low-level, manually managed languages such as C/C++ using compile-time annotations. While using compile-time annotations might reduce the scope of use since most binaries don't contain enough debug information to be used in order to generate high quality traces, we believe that such annotations are a reasonable requirement since most analyses that require precise tracing also require source code. `Ichnaea` works by first disabling access to the page that holds our 'Object of Interest' (ObjOfInterest) using virtual page protection. This means that any access to the object triggers a page fault. A pagefault handler routine is then used to *(i)* first allow the access *(ii)* log the "access context" and *(iii)* lock the object back again so that future accesses can still be caught — hence "tracing" the object. With traditional `mprotect` based page protection (*e.g.,* used by `Dthreads` [28]), changing the page access permissions is too slow (2µs+) and global (*for all threads*) leading to significant slowdowns and false sharing problems. In `Ichnaea` we use MPKs, to change the access permission of pages, which is very fast (50ns) and local to the thread (see section 2.2). This enables us to *precisely* and *efficiently* track when an object is accessed (*i.e.,* read from or written to) even in multi-threaded programs while completely alleviating the aforementioned sources of loss. `Ichnaea` *significantly reduces tracing overheads* — for instance, $10-60x$ reduction in overheads for SPECInt [40] workloads compared with SoTA tools like Intel Pin (see section 5.3). We can also *attribute accesses to their origins* and even provide *accurate timestamp information* — that can then be examined offline for further forensic analyses and debugging of complex (multi-threaded) programs .

Hence, our main contributions are:

1. **A *precise* and *lossless* object-tracing model** [Section 2.1]. We identify fundamental sources of loss in practical dynamic analysis techniques — *viz.,* coarse access-control granularity, global page permissions, syscall windows, atomic writes and false sharing. We then formalize requirements for complete object-level read/write tracing in both, single and multi-threaded C/C++ programs.
2. **`Ichnaea`: A new MPK-based tracing framework** [Section 3]. To the best of our knowledge, `Ichnaea` is the first object-tracing system to use Memory Protection

[4] we claim to be complete in userspace, not kernel. See Section 6, *para* 2.

Keys to provide efficient *per-thread* access control. This design eliminates all of the aforementioned dominant sources of trace losses, making it possible to analyze object accesses patterns in complex (multi-threaded) programs.

3. **Low overhead pagefault handling** [section 5.6.2]. We developed a low-overhead pagefault handling path (section 4.1) where we show that lightweight `pkey_set` operations *reduce pagefault handler overheads* by as much as 25*x* compared to traditional `mprotect`-based methods (see section 2.1.3)! This allows for more efficient accesses and improved precision while logging traces.

Our practical and deployable object-tracing framework[5] (section 3.2) integrates seamlessly with C/C++ programs, with *minimal* and *non-invasive* annotations to the source and supports real-world, multi-threaded workloads.

`Ichnaea` enables analyses that were previously impractical such as debugging rogue object accesses [1] in large codebases, recovering precise object-access patterns in multi-threaded systems [28], performing post-incident forensics and understanding accesses to security-critical objects to devise finer-grained access-control policies for compartmentalization [26]. Prior approaches either incur prohibitive slowdowns (slowing down test suites, benchmarks and fuzzers to a crawl) or produce incomplete traces, leading to inaccurate or misleading analyses. `Ichnaea` is a perfect tool for tracing a sizable number of objects with minimal slowdown, tracing all objects however comes with caveats (see section 6).

We next discuss some relevant background material.

2 Background

2.1 Tracing Memory Objects

There are two broad ways to trace objects in memory (hereby referred to as ObjOfInterest): static analysis [10] and dynamic analysis [7].

2.1.1 Static analysis

Static analysis does not produce complete or sound [29] traces as doing so would require precise symbolic executions [6, 23] which is impractical for most codebases since it suffers from problems like *path explosion* [24]. Compile-time nondeterminism further exacerbates this limitation since it is impossible to predict some aspects of the execution *(e.g., user input)* resulting in over-approximated results. Another idea is to statically find all sites where ObjOfInterests are modified and inject instrumentation that logs the accesses dynamically (a mix of static and dynamic analysis). However, for any given ObjOfInterest (heap-allocated or global) it is generally *undecidable* to identify *all* potential access sites, particularly in the presence of indirect memory references, pointer arithmetic, aliasing and data-dependent control flow. An example of this is shown in Figure 1 where the object `GLOBAL` is modified by a chain of indirections. There are pruning based approaches [47] that can reduce the number of instrumented `loads` and `stores` by removing paths guaranteed to not touch an ObjOfInterest. However, for complex programs *e.g.,* `nginx` this is impractical as the various control flow indirections lead to an over-approximation of accesses sites — resulting in no reduction in the amount of instrumentation anyway. Further, static analysis-based instrumentation necessitates *compiling every library statically into the program* which might be impractical when library source code is either hard to instrument (*e.g., libc*) or unavailable. As a result, there is a high likelihood that static analyses will miss many read/write operations.

2.1.2 Dynamic analysis

Dynamic analysis solves many of the above problems by *instrumenting* all *load and store* instructions in the program at runtime, so that no access goes untraced. Dynamic analysis *requires significant instrumentation* (usually a call to a trampoline function when ObjOfInterest is accessed) that can be either at *compile-time* [19, 38], *runtime* (*e.g.,* Dynamic Binary Instrumentation tools like Intel Pin [30], DynamoRIO [8] and others [34]) or by *binary rewriting* [11]. Accesses from kernel or dynamically loaded libraries are still not covered as that'd require kernel or dynamic library instrumentation which is often impractical due to the complexity of the kernel and dynamic loaders. However, dynamic analysis suffers from following problems:

Performance Impact: The issue with tracking loads and stores is that the instrumentation cannot be precisely injected at all object access sites since finding all such locations is know to be undecidable. Thus, instrumentation based dynamic techniques pay the penalty of instrumenting most loads and stores, slowing down the program by as much as *50-100x*. Some tools claim to have better performance (*e.g.,* 2x in case of ASAN [38]), but these claims are for simpler tasks (*e.g.,* bounds checking) instead of *precise* and *complete object tracing*. This slowdown makes it impractical to exercise long running test-suites, benchmarks, fuzzing, *etc.* that are crucial for simulating runtime behavior of big complex programs.

Limited Coverage: Some of the dynamic analysis techniques (that use compile-time and binary-rewriting [33] methods) do not cover accesses in *shared libraries* unless they're baked in with the original program. Sources for shared libraries are not always available or easily instrumentable, reducing trace quality. Furthermore, these tools do not *trace kernel reads*[6]*/writes* for an object.

General tools: most of the tools mentioned above are general-purpose tracing tools and do not specialize in precise tracing of memory objects, thus needing some further work to adapt them as such.

[5] Open-sourced at https://ichnaeatracer.github.io/

[6] `Ichnaea` does not claim to completely trace kernel reads, see section 6

2.1.3 Tracing Backing Memory — Dynamic Analysis

A better way (again via dynamic analysis methods) is to *trace writes to the backing memory*, either by using hardware debug registers [17] or virtual memory [31] protection. Hardware debug registers are limited in number and size (usually 4 in x86 with a max capacity of few bytes) making them impractical for tracing big, or a multitude of objects. The idea behind using virtual memory protection-based tracing is that the access permissions of the memory page that is backing an ObjOfInterest can be modified to disallow reads/writes. Any thread attempting to access the object would run into a pagefault (segfault). The pagefault handler can be customized to let the access proceed — by temporarily unlocking the page using `mprotect` [31], *i.e.,* making it read/write (R/W) — and logging the access. This means that once access to a object (the backing page) has been restricted (via the `mprotect` call), no active instrumentation is needed to trace the objects. Thus the *program pays virtually no cost as long as the object isn't accessed*. This is a more efficient approach than watching every load and store since there's virtually no overhead for user-space tracing until an ObjOfInterest is accessed. In contrast, monitoring every load/store adds constant overhead even when the loads and stores are not made to ObjOfInterests.

Tools like Dthreads [28] and others [3, 20–22] use this technique to trace memory accesses though none match our purpose exactly which is to precisely trace objects in memory with rich metadata. However, in the context of precise and complete tracing of memory objects, this technique suffers from various challenges:

1. **Interrupt Handler Overhead:** when a restricted access is attempted, the CPU calls the kernel pagefault handler that then calls a custom handler — a costly ordeal taking up to 2.2µs (see stage 1 in Table 5) for each attempted access. Next, the custom handler must *(a)* unlock, *(b)* emulate the faulting instruction, *(c)* log the access and *(d)* lock the page back. This unlock-relock cycle requires two `mprotect` calls (2µs each, 4µs total; see appendix A.2), making it the primary bottleneck. Further, tracing of reads becomes quite expensive as programs naturally tend to read more than write, adding significant performance impacts to objects that reside on the same page as an ObjOfInterest. Naively isolating all ObjOfInterests on their own dedicated pages can solve this issue but, it significantly increases memory usage.

2. **Trace Losses:** one of the biggest problems faced by pagefault based tracing is the issue of losses in multi-threaded systems. The reasons being the coarse granularity of virtual memory protection (access control can only be applied on page sized blocks) and the fact that traditional page permissions apply across *all* threads and not just one. Trace losses can happen due to any of the following cases in multi-threaded systems:

   Syscall Losses: to cover kernel writes to traced objects (not reads, see section 6), the least non-invasive idea (*i.e.,* without modifying the kernel) is to wrap all *libc* covered syscalls (*e.g.,* `read(), readv()`) with a wrapper using the LD_PRELOAD mechanism [14]. The wrapper's job is to *unlock and lock all/one traced objects* before and after a syscall, which is extremely expensive. More importantly, *either one or all traced objects are unlocked during this time* and *any* thread can simply write to any traced objects without causing a pagefault *i.e.,* without being traced. This could potentially result in *significant trace losses* since syscalls can take an arbitrary amount of time (up to a few milliseconds) while memory accesses generally take less than 100 nanoseconds. This means that in a multi-threaded program with frequent trace object accesses, the possible lower bound for losses could be *hundreds of missed traces per syscall*.
   False Sharing Loss: in cases where multiple objects share the same backing page *e.g.,* a traced object sharing a page either with another traced or un-traced object. One thread could unlock the whole page (for a brief window) to access one object, while also leaving other objects on the same page unintentionally unlocked. Writes to these objects will be lost during this window.
   Atomic objects: since atomic objects don't need locks, two threads can try to access an object at the same time. This means that while one thread has temporarily enabled access to a traced atomic object (unlocked the object), another thread could access the same object without triggering the pagefault. This is because the time window for handing the pagefault in the first thread (the object is unlocked during a part of this window) can be very large (see *R/W window* in Table 5) in comparison to how long it takes to do complete atomic write.

3. **False Sharing:** since virtual memory protection works at page granularity, if any other objects (that aren't being traced) share the same page with a traced object then they will also trigger a pagefault. These faults can be filtered out by the pagefault handler (by checking the exact faulting address). However, a pagefault and its handling is expensive (see stages 1,2 in Table 5) and must now include, the cost of two additional expensive, `mprotect` calls. Hence, incurring too many pagefaults will *significantly reduce performance*. This can be solved by allocating a dedicated page for every object. However, as mentioned earlier the memory overhead of doing this may become impractical.

2.2 Memory Protection Keys

Memory Protection Keys (MPKs) [36] extend the traditional virtual-memory model by *decoupling access rights from page-table entries*. One or more pages can be tagged with a key of which there are 16 in total. The reason for

multiple keys is to have a larger number of distinct access control domains. A domain could be a secure crypto library which needs its own key to isolate it's pages from other parts the code and similarly other libraries and modules can have their own keys to isolate their data and stack thus improving security by memory isolation. Page table entries only know which key is associated with each page. Each processor core maintains a local register that specifies the access permissions for pages tagged with any of the 16 keys. A thread can therefore revoke access (*e.g.,* remove write access to all pages tagged with key *k*) without modifying page tables or triggering TLB flushes — overheads unavoidable with classic page-protection mechanisms. By default, all pages are associated with key 0 and all threads have access to pages tagged with key 0 but these can be changed during runtime to provide isolation. MPKs enable fast, user-level domain switching and fine-grained intra-process isolation. For our system, this prevents cross-thread interference since granting access in one thread does not affect others, thereby avoiding false-sharing-induced trace losses. MPKs are supported on both Intel [36] and ARM architectures [5] and are used for memory isolation, *e.g.,* page table hardening [9].

3 Design

To solve the challenges mentioned in Section 2, we introduce `Ichnaea`, a tracing framework designed for **precise**, **lossless** and **complete** memory object tracing for programs written in low-level manually managed languages *viz.,* C/C++. `Ichnaea` works by using Memory Protection Keys [43] (MPKs) that provide `mprotect`-like access control without requiring page table changes, needing only a CPU-local register rewrite to switch protection domains. This enables per-thread access control completely *alleviating all losses* (see section 2.1.3) while *halving interrupt handling cost*, leading to significant reduction in tracing overheads. Further, traced global objects can be put in a dedicated section by simply adding a macro (see appendix A.1) before its initialization. This ensures that other global ObjOfInterests do not trigger unnecessary segfaults. Heap objects can be isolated efficiently (section 4.5). `Ichnaea` is realized as a shared library, `libichnaea.so`, that is loaded into the target process (at load time) to trace ObjOfInterests. This design alleviates the need of linking the program with our library or adding special compiler passes, both of which require modifying complex build systems. `Ichnaea` exposes a simple API call and some helper macros (see appendix A.12) that can be added to the program just by adding a header file. This file contains a *weak reference* to our API call — thus, it does not require any build-system changes or additional linking steps.

3.1 Design Principles

We believe a tracing mechanism must follow these principles:

1. Low startup overhead: startup overhead is important as many benchmarks launch repeated short-lived instances and fuzzers could execute thousands of runs per second. Even a few extra milliseconds per launch can balloon into hours over the life of a campaign.

2. Minimal cost until ObjOfInterest is accessed: most executions may never access the objects we care about (ObjOfInterests). The tracer must remain effectively dormant until a monitored object is accessed.

3. Low implementation complexity: a smaller code base is easier to audit, maintain and port. Avoiding heavyweight features such as a JIT compiler or kernel modules not just reduces the chance of side effects but also startup overheads.

4. Ease of use: users should be able to trace objects with minimal API call. Simple usage patterns encourage adoption.

5. Rich contextual information: when an access to an ObjOfInterest occurs, the tracer must record more than just the address and value from each running thread — it should also capture the instruction pointer, thread ID, stack trace, data and an accurate timestamp. This context turns raw events into helpful debugging data.

3.2 System Overview

Figure 2 illustrates the overall workflow of `Ichnaea`. The framework is divided into three stages:

1. **Compile-time:** `Ichnaea` employs lightweight compile-time annotations to register ObjOfInterests. Developers *explicitly* identify objects by invoking our API call:

   ```
   ichnaea_register_obj(addr,size,name,type);
   ```

   This call records the address, size and type information of the object to be tracked during runtime. We also add special compiler directives (line #2, Listing A.12) so that stores to these objects are not optimized away by the compiler. Once annotated, the workload is compiled without *any modification to its build system*. Wrong annotations (*e.g.,* object size) may result in incorrect traces but *all the correctly annotated objects will be traced without any loss*.

2. **Runtime:** The call to register an ObjOfInterest constructs metadata tables that create a map of registered ObjOfInterests, their address and backing pages. Next, it disables read/write (or both) access to the object's backing page. Future calls to register other objects (at any point in time) do the exact same steps. Since access to the backing pages of ObjOfInterests is disabled, any future attempt by the program (or its child processes or external libraries) to access any of the pages belonging to ObjOfInterests triggers a segmentation fault (`SIGSEGV`) *i.e.,* the segfault happens at the moment when the page access is initiated. A *custom signal handler* within `libichnaea.so` intercepts this fault and performs the following actions:

   - checks if faulting address is a registered ObjOfInterest (sets flag `isTraced`),

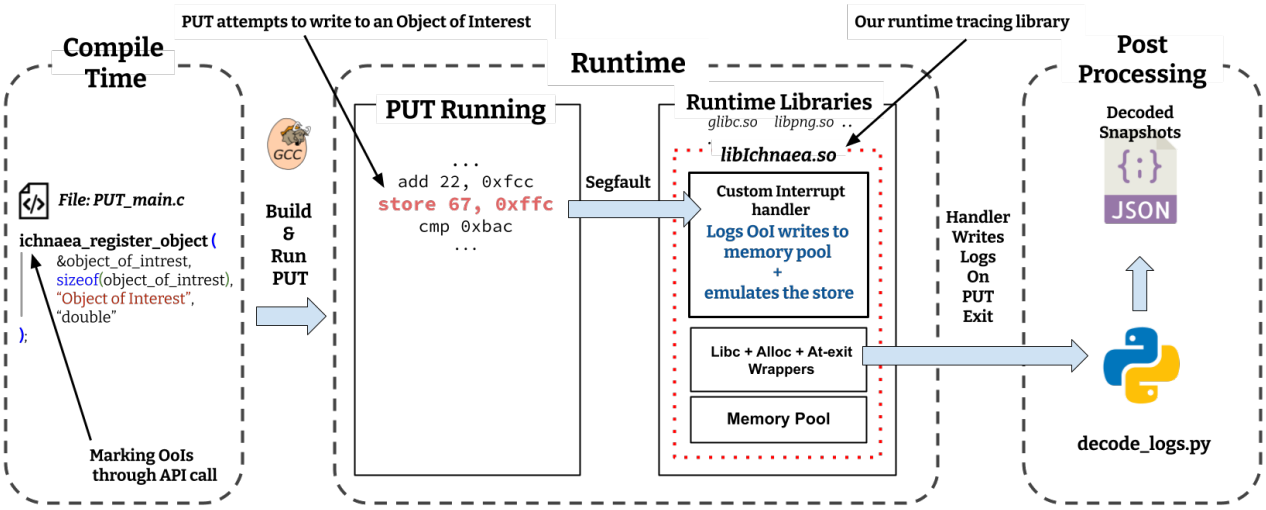


Figure 2: System Overview of `Ichnaea`. The dashed lines represent the tracing phase. The red dotted line box contains all the components that form `Ichnaea`'s runtime library

- enables access to the underlying page
- emulates the instruction that tried to access the ObjOfInterest (so that the program runs correctly)
- `iff (isTraced)`:
  - it logs the attempted memory access with metadata (*i.e.,* execution stack, thread id) to a dedicated trace buffer
- resets the pkey to disable access again
- returns so the program can proceed normally

This selective interception strategy ensures that only accesses to designated objects incur overheads, while *all other program execution proceeds without interruption*. Hence, `Ichnaea` remains *toggled off* until an ObjOfInterest is accessed. To trace accesses made by syscalls, we wrap all `libc` syscalls. These wrappers inspect the syscall arguments to find if an ObjOfInterest is being accessed. If so, they log the access after the syscall finishes. These wrappers remain inactive till the first call to register an object is made (see Section section 4.3 for more details.)

3. **Post-processing:** upon program termination, event logs stored in trace buffer are serialized into a standardized `JSON` format. This structured representation aids downstream analysis, enabling *(i)* investigation of fine-grained access patterns, *(ii)* correlation of memory activity with program phases and *(iii)* integrating traces into profiling or visualization pipelines with minimal effort. For each traced object, `Ichnaea` provides data *e.g.,* the accessor's call stack, process_id, thread_id, access type (R/W), timestamp and hash *etc.* See Appendix A.4 for more details.

4 Implementation

The `Ichnaea` prototype was implemented using less than two thousand lines of C code and *required no kernel modifications.* We have open-sourced our implementation online[7].

4.1 Segfault driven tracing

Figure 3 shows sequence of operations in `Ichnaea`. The framework uses *pagefaults* (also known as *segfaults*) to *trace object access*. A segmentation fault is an interrupt raised by the memory management unit (MMU) when a program accesses a page without the right permissions that are set at runtime. At any point during runtime (ideally at the start of program, *e.g.,* `main()`), one or more ObjOfInterests (heap or static allocations) are "registered" for tracing. The registration API then invokes `pkey_mprotect` syscall to tag the backing virtual pages that hold those objects with a `pkey`. This `pkey` is assigned by the kernel and we use it to tag all pages that are to be traced. The current running thread then disables access to any pages that are tagged with our `pkey`. Hence, any subsequent access (the first box in Fig 3) to these pages triggers a segfault (`SIGSEGV`) — the second box in Fig 3. The resulting `SIGSEGV` is intercepted by our *custom signal handler*. The handler then *(i)* enables access to all pages for that thread only *(ii)* emulates the faulting instruction *(iii)* logs the access context and data and finally, *(iv)* returns to normal execution. Without emulation, we would have had to run the faulting instruction again after the handler unlocks all ObjOfInterests. This means that we would have to lock the ObjOfInterests right after running the faulting instruction in order to avoid trace losses. This necessitates adding a breakpoint (`INT3` in-

[7] https://ichnaeatracer.github.io/

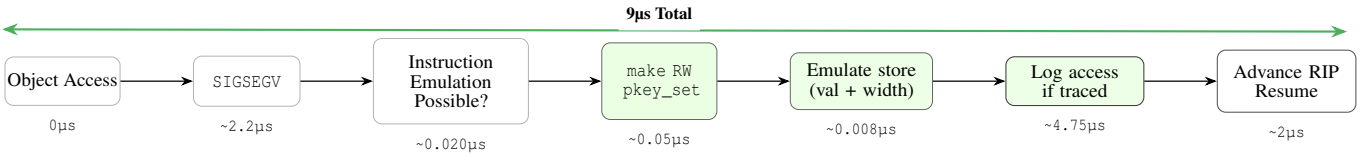


Figure 3: Fault handling sequence in the interrupt handler. Boxes are numbered from right to left starting from 1. Box colors indicate page protection status (Green: Unlocked, Gray: Locked).

struction) right after the faulting instruction. The new trap instruction will run the segfault handler again to lock the ObjOfInterests back. Thus, causing one more round of segfault and handler execution. Furthermore, adding breakpoints in user code during runtime may cause other threads to hit the same breakpoint (*e.g.,* while running a shared `memcpy` on a separate piece of data) causing additional complications.

Also note that emulating all the instructions on x86 is huge engineering effort. Thus for the instructions we do not emulate yet *e.g.,* floating point writes and other complex instructions, the handler takes the *slow path*. The slow path ensures that the non-emulated instruction is executed directly (on the CPU) instead, thus ensuring correct program behavior. However it's slower in comparison to emulation (see appendix A.9 for details). Slow path is an engineering effort as emulating each instruction requires understanding and dissection — while a challenge, it is easy to overcome in production. We will explore this in future work.

Detailed Breakdown of the segfault Handler If the segfaulting instruction can be emulated (third box in Fig 3) and the faulting address corresponds to an ObjOfInterest or happens to fall on the page backing an ObjOfInterest, we implement the following steps in the interrupt handler,

1. *Temporary unlock:* (fourth box in Fig 3) access to all pages marked with the tracer's `pkey` (`Ichnaea` uses only one out of 16 pkeys, to lock all objects) is enabled via `pkey_set()` for that thread. Other threads will still fault if they try to access any ObjOfInterest, including the one unlocked for the current thread, hence alleviating the issue of trace losses.

2. *Emulated store:* (fifth box in Fig 3) the value and size of the access requested by the faulting instruction are extracted from instruction operands by inspecting CPU register states. Access is then emulated to restore original program behavior by reading/writing the appropriate number of bytes at the *effective address*.

3. *Logging:* (sixth box in Fig 3) if the access target was an ObjOfInterest, it is recorded (`thread_id`, IP, call stack, data) in a dedicated trace memory pool.

4. *Advance execution:* (seventh box in Fig 3) after instruction emulation, the instruction pointer (`RIP`) is advanced to the next instruction in order to avoid another *segfault*. Returning from the handler auto-restores key permissions to the pre-interrupt (locked) state so there is no need to re-lock pages.

5. *Exit:* at ProgramUndrTest's exit, `Ichnaea` writes all snapshot data from the memory pool to disk.

All the above mentioned steps are done in `async-signal` safe manner, for instance, by using dedicated local copies of every `libc` function used. Heap allocations (for `Ichnaea`'s own use) are carried out using custom allocators that invoke `mmap` [32]. For each ObjOfInterest access, we log:

(*a*) **who** accessed it, (thread and process id via the `gettid()/getpid()`),

(*b*) **where** it occurred, (instruction pointer `RIP` and call stack via `libunwind()`)

(*c*) **when** it occurred (timestamp via `clock_gettime()`, plus a global access counter) and finally

(*d*) **what** changed (before/after dumps for writes).

All the logs are written to a dedicated memory pool to avoid the overhead of file I/O during runtime. The data is written to disk at program exit.

4.2 Concurrent and Cross-library Accesses

Unlike traditional `mprotect` based techniques, `Ichnaea` only allows ObjOfInterest *access within the current thread*, so other threads will still segfault while accessing other ObjOfInterests that either happen to be on the same page or use atomic operations. This allows our framework to seamlessly support unmodified, highly concurrent applications such as PostgreSQL [44] and Nginx [41].

Since `Ichnaea` works by removing access permissions to the backing page of an ObjOfInterest, any access from a library (*e.g.,* `snprintf` in `libc`, *etc.*) will result in a pagefault that in turn, will log the access. This alleviates the need to statically recompile all the dynamic runtime libraries which might be required for compiler based instrumentation techniques that instrument every load and store.

4.3 Kernel Accesses

A complication arises when the kernel accesses a user-space buffer via syscalls *e.g.,* `read()`, `recvmsg()`, `getcwd()`.

Because these accesses occur in kernel mode, the kernel never triggers a user-space segfault that Ichnaea relies on; instead, the kernel detects the invalid access internally, skips the syscall and returns an error such as EFAULT, indicating that an invalid buffer was encountered. A naïve remedy is to retry the syscall after unlocking the faulting page, but this fails because syscalls are stateful (many of them advance file offsets, consume descriptor buffers or deliver signals), so re-execution can alter program semantics.

We therefore wrap *libc functions*[8]. The wrapper first enables accesses to all ObjOfInterests *(for the current thread)*, inspects syscall arguments to identify which pointers (or I/O vectors) intersect with traced objects to know what's being accessed before forwarding the syscall to the kernel. After the call returns, the wrapper restores the original protection. We further use hashes of ObjOfInterest states to detect changes for syscalls that may be called but don't end up writing to any ObjOfInterests *e.g.,* a `read()` call on an empty file buffer. Details and an example of our wrapper can be found in appendix A.6.

Parsing Syscall arguments, though conceptually simple, is difficult in practice due to:

- variable-length aggregates: Interfaces, *e.g.,* `readv()`, `recvmsg()` *etc.* pass an array of `iovecs` whose length is unknown until runtime, requiring a loop inside the wrapper. This is a huge engineering effort and results in some runtime overhead.
- inline syscalls and virtual dynamic shared object (VDSO) shortcuts: Some performance critical libraries issue syscalls directly via `syscall` or the VDSO page, bypassing libc hooks unless caught with `seccomp` [42] style tracing or binary rewriting [37].

These issues make a full in-kernel solution attractive, but kernel tracing requires elevated privileges and introduces maintenance cost across releases and patches. In contrast, our user-space wrapper keeps Ichnaea self-contained while still covering a significant number of syscalls that perform kernel-to-user copies, *viz.,* `read()`, `write()`, `recv*()` and `ioctl()`, along with output pointers and miscellaneous queries such as `getcwd()` and `getdents64()`. Syscalls for which we cannot find the object addresses (*e.g.,* `ioctl, readv`), we do the following: *(a)* for writes, we keep a hash of last states of traced objects and compare post-syscall state to determine if any change has been made *(b)* and for ObjOfInterest reads, if we can't resolve the read targets by parsing the syscall arguments, we miss those traces. However, this is an engineering limitation, with enough effort a wrapper can potentially dissect all syscall targets. Empty writes (*e.g.,* `read()` on an empty file) are also detected but empty reads (object passed to the syscall for reading but not read by the kernel) are not detected.

[8] Currently, we wrap 20 such functions list on website

4.4 Invocation

First, users need to identify the ObjOfInterests (heap or global) that they wish to trace, *e.g.,* structs such as `ngx_http_core_module` in `nginx` that govern critical aspects of request handling. In a large codebase like `nginx`, determining which components read or modify such structures is difficult to do manually or via static analysis, especially when accesses occur through deep chains of indirection — a common pattern in `nginx`. Similar difficulties arise when tracking unintended accesses to a buffer or diagnosing asynchronous accesses that manifest within otherwise synchronous systems. Listing 4 in appendix A.10 shows the invocation sequence for Ichnaea. Integrating Ichnaea into an existing project requires two minor changes to program source:

1. **Build-time change:** Add the header *tracer.h* and insert our API call to `ichnaea_register_object()` for every object that should be traced. No additional build-system edits are necessary. An *optional* flag `-rdynamic` can be added while building to improve the quality of callstack symbols.
2. **Run-time activation:** Launch the program with the tracer library pre-loaded as follows:
   `LD_PRELOAD=libIchnaea.so ./myBinary`
   The `LD_PRELOAD` can be set for the whole environment as Ichnaea stays dormant until a program initializes it.
3. **Post run data decoding:** At program exit, Ichnaea dumps the binary object data snapshots and related context information in a directory. We provide a Python script to convert binary snapshot data given the structure of objects. A sample of the `JSON` file with traced data from an `nginx` run can be found in Listing 1.

4.5 Tracing Dynamic Heap Allocations

Ichnaea traces dynamically allocated ObjOfInterests without requiring users to manually annotate every `malloc` site. It also isolates ObjOfInterests from unrelated objects to avoid unintended segfaults. To do so, Ichnaea wraps `malloc`/`calloc`/`realloc` to provide two additional capabilities:

(i) *automatic detection* of ObjOfInterest allocations

(ii) *page-isolated slab allocation* for each ObjOfInterest

Annotation and Automatic Detection: Pointer objects (global or heap handles) that hold references to heap buffers are registered via our API call (see section 4.4). In addition, the initialization sites of these pointers are marked with a lightweight compiler annotation (listing 2, Line 2 in appendix A.5). This annotation prevents compilers from optimizing away the store instructions needed to automatically detect the allocation.

At runtime, the interposed allocator (*e.g.,* `malloc`) inspects its caller and identifies the instruction storing the return value;

the destination of this store is compared against registered ObjOfInterest handles to determine whether the allocation belongs to an ObjOfInterest. We validated this mechanism using both `gcc` and `clang`. This allows `Ichnaea` to trace dynamically allocated objects *without requiring users to manually locate allocation sites* in large codebases.

ObjOfInterest Isolation: If the allocation targets an ObjOfInterest, the wrapper allocates a page-aligned, page-sized slab using `memalign()` and returns that pointer; ordinary allocations are forwarded to the native allocator. Because each on-heap ObjOfInterest resides on a private page, unrelated accesses cannot spuriously trigger `SIGSEGV`. Even when tracing say a 1000 objects, the memory overhead would only be around 8 megabytes plus the cost of the `mmap` syscall made by allocators. All of these in our view are acceptable since this analysis is done offline. Appendix A.7 discusses possible refinements. Custom allocators can also be instrumented similarly using a macro (see appendix A.12).

5 Evaluation

We evaluate `Ichnaea` against two tools, *(i)* Intel Pin (SoTA) *(ii)* `mprotect` based tracer. The workloads we used are described in Table 1. **Intel Pin** was chosen since it provides *complete* traces (with the exception of syscalls) without any build system modification and is faster [30] than other complete tracing tools like DynamoRIO [8] or Valgrind [34]. **The `mprotect` based tracer** was chosen as it represents tools like Dthreads [28] that use `mprotect` to trace object accesses. For a fair comparison, we simply changed `Ichnaea`'s implementation to use `mprotect` instead of MPKs (details in appendix A.3). Thus the `mprotect` based tracer gets all benefits and optimizations of `Ichnaea` minus the MPKs. Compiler injected instrumentation based techniques were not selected as even they also instrument almost every load and store and require adding complicated compiler instrumentation passes.

The workloads are part of the *SPECInt* benchmark which represents real world programs (*i.e., gcc, xz, perlbench*) and have exhaustive benchmarks and test suites that ensure that the frequency of object accesses represent real world scenario.

As part of our evaluation, we intend to explore the following research questions:

- **R1**: Can the state of the art for object tracing be improved in terms of Performance? See sections 5.3.1 and 5.7
- **R2**: Can the quality of tracing be improved including completeness, precision and coverage? See section 3.
- **R3**: Can usability/adaptability of object tracing be improved? Section 3.

5.1 Evaluation Setup, Workloads and Metrics

The evaluation platform is running Ubuntu 24.04.2, kernel 6.14.0. Experiments were executed on Intel(R) Xeon(R) Silver 4316 CPU @ 2.30GHz with 64GiB of memory.

Workload	Description	Version
SPEC CPU2017	SPEC CPU2017 Integer Speed Testsuite	1.1.9
Postgres	Highly parallel workload with built-in testsuite	17.5
Synthetic Program	To compare fuzzing overheads	N/A

Table 1: Description of different workloads and their attributes

Table 1 describes the workloads used in our evaluation with version numbers. We selected SPEC CPU 2017 [40] because it is the de-facto industry benchmark for CPU performance and micro architectural studies containing real work benchmarks. Its workloads are compute intensive rather than I/O-bound, so they saturate the memory hierarchy with millions of loads and stores — exactly the traffic pattern we need to stress an object tracer. SPEC CPU is also what Intel Pin used in their evaluation. The *exchange2* from SPEC wasn't included as it's written in Fortran. We ran 5 copies of each workload for this experiment. We also test a highly parallel workload, postgres [44] to highlight `Ichnaea`'s performance under high-concurrency scenarios. We ran `postgres` test suites 3 times for each tracing tool. As seen in Table 2, the main metric we capture is execution times for various test suites under diverse conditions *e.g.,* high-frequency accesses, parallel accesses, low-frequency accesses and runtime with and without traced objects. While running fuzzing experiments, we capture the number of paths covered (*i.e.,* exercised by the fuzzer), the number of paths successfully traced (accesses recorded by tracer) and the runtime overhead relative to native execution.

Table 2: Metrics captured across test suites and fuzzing experiments.

Category	Metric Captured
Execution Time (Test Suites)	High-frequency accesses Parallel accesses Low-frequency accesses Runtime with traced objects Runtime without traced objects
Realistic Use cases (Fuzzing)	Paths covered (i.e., exercised by the fuzzer) Objects traced (ObjOfInterest accesses captured by instrumentation) Total fuzzer runtime

5.2 Object Selection Rationale

We selected `global` objects that were either *(i)* statically allocated at load time or *(ii)* acted as pointers to heap regions that would be allocated later. We don't trace stack allocated objects. For external libraries, it was often hard to locate such

objects directly in the source tree so we relied on the projects' test-suites and benchmark drivers to discover global objects that are actually touched under realistic workloads.

We selected objects with varying access frequencies to highlight their effects on tracing overhead. For example, in SPECInt's *x265* (a HEVC/H.265 video encoder), we traced `cli_input_t input`—an infrequently accessed file pointer; while in *gcc* we traced `input_location`, which experiences frequent runtime accesses. This range of frequencies ensures we capture diverse tracing behaviors across variables. More details are available on our website.[9]

5.3 Comparison with Intel Pin (SoTA)

We compared `Ichnaea` with Intel Pin (state of the art for dynamic tracing tools) using metrics such as performance, features and ease of use. To adapt Intel Pin to mimic `Ichnaea`'s behavior, we devised the following workflow:

1. Compile the ProgramUndrTest with static addresses so ObjOfInterest addresses don't change at runtime
2. Dump all object addresses from the binary to a database
3. Filter out ObjOfInterest addresses from the database
4. Create a pintool [30] aware of ObjOfInterest addresses
5. At runtime, log every store instruction (address + value) whose target address is in ObjOfInterest database
6. For each ObjOfInterest access, fetch context info (*pid, tid, callstack*) through ptrace API

These steps create a pintool closely that mimics `Ichnaea`'s behavior. To trace dynamically allocated heap objects, we manually added hooks to all the allocation sites. These hooks inform the pintool of new allocations to ObjOfInterest.

5.3.1 Performance Comparison with Intel Pin

Figure 4 compares the runtime overhead (lower is better) incurred by `Ichnaea`, *Intel Pin* and a `mprotect` based tracer for SPECInt workloads. The x-axis lists all the application under the SPECWorkloads. The y-axis shows median runtime normalized to un-instrumented baseline (log scale), where a value of 136 means a 36% slowdown. For each benchmark, the green dotted line is the un-instrumented baseline, first bar (orange) is `Ichnaea` runtime, the second bar (yellow) is `mprotect` based tracer and third bar (blue) is Intel Pin runtime. For details on the kind of traced objects see Section 5.2.

Figure 4 shows that in most cases `Ichnaea` significantly outperforms `Intel Pin` — by anywhere from 12 to 60 times. This addresses research question **R1** from Section 5 as we significantly improve performance when compared to SoTA. We also address **R2** since these improvements to tracing techniques mean more efficient analysis pipelines meaning complex software is better understood and thus more resilient, correct and secure.

The relatively low overhead of `Ichnaea` stems from its lightweight design: a single `LD_PRELOAD`-ed shared library

[9] https://ichnaeatracer.github.io/

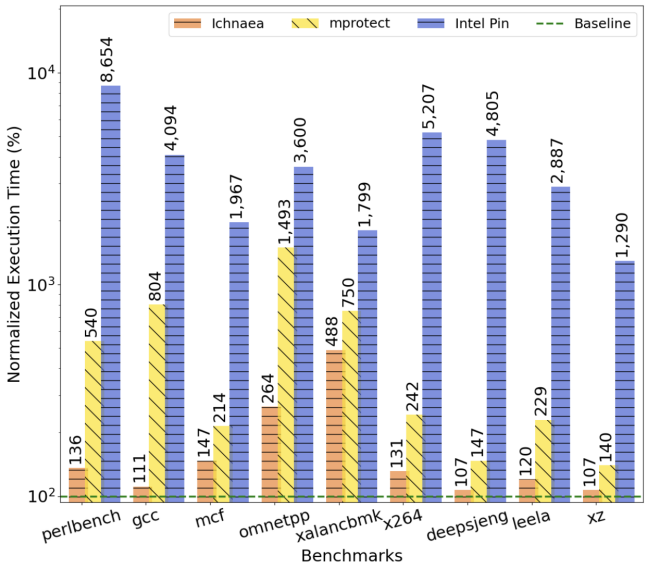


Figure 4: Comparisons of runtimes: `Ichnaea` vs Intel-Pin vs `mprotect` Instrumented for SPECIntSpeed Benchmarks normalized to un-instrumented runtime (red line) [Log Scale]

imposing itself *only on targeted events*. By comparison, Intel Pin's overhead originates from its dynamic binary instrumentation where *all instructions* are analyzed to filter out relevant events such as load and store. This evaluation of every instruction along with heavy components (*e.g.,* code caches, JIT compiler) are responsible for IntelPins' slowdown.

As seen in Figure 4, while `Ichnaea` typically increases runtime by only 1-3× over native execution, its impact on *xalan* is significantly higher (4.8×). This is attributable to the workload's extensive use of global variables that are frequently updated during runtime. Because `Ichnaea` traces several of these hot-path globals, many unrelated objects that reside on the same virtual memory pages and therefore incur the same page-protection and instrumentation overhead. This collateral impact amplifies the runtime cost, resulting in a relatively larger slowdown compared to typical workloads.There are ways to avoid part of these overheads which are described in section 3. However, we did not use any optimizations (*e.g.,* placing globals objects in `.rodata` to cut collateral segfaults) to highlight the worst case performance for the tool. Such optimizations will only improve `Ichnaea`'s performance. Although the overheads are high, they're much lesser than Intel Pin. Further, this kind of analysis is usually done offline hence making our overhead acceptable.

The general variance in `Ichnaea`'s performance (1 to 2×) across benchmarks is largely attributable to the characteristics of the traced objects. Some reside outside the "'hot path"' and incur minimal cost, while others lie directly in high-frequency execution paths, amplifying overheads.

The overhead introduced by Intel Pin during traced runs ranges from 12× to 86×. This variation depends on program

characteristics such as size (larger programs require more instrumentation points) and complexity rather than solely on object access frequency or location. Additionally, applications that perform many syscalls show relatively lower overhead, because Pin does not trace syscalls, limiting visibility into such accesses.

5.3.2 Comparison with `mprotect` tracer

`Ichnaea` outperforms traditional `mprotect` based tracing anywhere from 1.3—7×. This is due to the lower interrupt handling cost as `mprotect` needs to modify the page table entries. In a multi threaded program, page table entry change is sent to all the cores causing TLB flushes thus significantly increasing the cost at runtime as the number of thread increases.

5.4 Performance comparison with Postgres

In table 3 we present the results comparing `Ichnaea` and `Intel Pin` using a highly parallel workload — `Postgres` [44] — using its built-in regression test suite. For these experiments, we traced *five global structures* responsible for maintaining internal state during runtime. The table shows Real, User and Sys times captured in seconds for an uninstrumented binary, `Ichnaea` instrumented and `Intel Pin` instrumented runs of `Postgres`'s test suite.

The table shows that `Ichnaea` runs *26 times faster* than `Intel Pin`. These results highlight `Ichnaea`'s ability to maintain significantly lower runtime overheads even in multi-threaded, state-intensive workloads. The *Sys* time (time spent in kernel mode) consumed by `Ichnaea` is higher (11.2s) than Intel pin (10.4) as interrupts (triggered at object access) are first sent to the kernel, leading to longer kernel code execution. The *User* time (time spent in userspace) is higher for Intel Pin because of it's high overheads. Details about the test environment and workload setup are provided in Table 1. This also addresses research question **R1** as this demonstrates that we can make tracing highly-parallel workloads more efficient.

Table 3: Postgres performance (average of 3 runs) on experimental setup

Configuration	Real (s)	User (s)	Sys (s)
Un-instrumented	17	4.9	4.3
`Ichnaea`	**35**	**15.1**	**11.2**
Intel Pin	944	38.7	10.4

5.5 Object Scaling and Isolation Overheads

We analyze how `Ichnaea`'s overhead scales with the number of traced objects using a synthetic workload of 100, 8 byte objects (50 global, 50 heap-allocated). All objects are accessed 10000 times (split equally between reads/writes), regardless of whether they are traced, keeping the access pattern identical across configurations. Runtimes are normalized to the 2-object baseline as using configurations with no traced

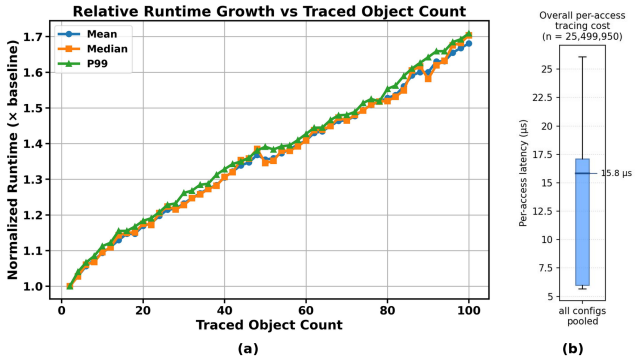


Figure 5: (a) Runtime of `Ichnaea` as traced object count grows from 2 to 100, normalized to the 2-object baseline (20 runs per configuration, 10K accesses per run, evenly split between reads and writes). Mean (blue circles), median (orange squares) and P99 (green triangles) are shown. (b) Box plot of pooled per-object access overhead in microseconds.

objects (*i.e.,* vanilla untraced program) was skewing the results. This is because the untraced version runs 10 times faster as the workload here represents the *absolute worst case* for `Ichnaea`. A value of 1.35 indicates 35% overhead over the baseline.

As Figure 5 (a) shows the overhead grows in a linear fashion — each additional 20 objects add approximately 8–10% runtime. Further, the proximity of the mean, median and tail shows the stability of `Ichnaea`'s overheads even under heavy load. At first glance, one might expect each new traced object to add a full ∼9μs overhead (∼9μs is around 8900% overhead in comparison to a un traced object access that takes 50ns), but this is not what we observe. Since all 50 globals easily fit on a single page (400 bytes vs 4096 bytes page size), locking any one of them faults all global accesses regardless — adding more traced globals therefore increases only logging and stack-unwind costs (less than 5μs each), not fault rate. This would be the case for most applications as a page-sized set of globals is sufficient for most use cases. Traced heap objects each occupy an isolated page and thus contribute genuine "new faults", making them the significant driver of the overhead. The remaining overhead comes from per-access metadata writes, stack unwinds and logging.

Figure 5 (b) shows the pooled box plot of the per object access latency in microseconds. The plot is combined across all runs as the range and median of the box plots for any number of traced objects were almost exactly the same (all medians differed by less than 1% from each other) which means the *object access overhead is consistent as the number of traced objects grow*. In this the median is around 15 μs, since the current experiments rapidly swaps between objects which adds extra cache overheads.

This workload represents a worst case scenario: *every object* is on the hot path and *every access* is traced, unlike real

Table 4: Heap isolation overhead at N=10,000 ObjOfInterests (median of 100 iterations). RSS is resident set size which represents the actual physical memory used by the program.

	Packed	Isolated	Ratio
Total RSS (Memory used)	4.2 MiB	80.3 MiB	19×
Per-object RSS	0.42 KiB	8.03 KiB	—
Per-alloc time	728 ns	5.5 μs	7.6×

programs where traced objects account for a small fraction of total execution. This is reflected in our SPECInt and PostgreSQL results (section 5.3), where overheads are substantially lower. Optimizations such as isolating hot-path globals on a dedicated page would further reduce the overheads seen here.

Memory Overhead of Heap Isolation

We quantify the worst-case cost of `Ichnaea`'s heap-isolation strategy (section 4.5) with many small ObjOfInterests exercised with a microbenchmark that performs 10,000 allocations under two configurations: *packed* (`malloc`) and *isolated* (`posix_memalign` to a full page, exactly mirroring `Ichnaea`'s instrumented allocator). Sizes are drawn from a seeded distribution skewed toward small objects (∼55% in 8–64 B), matching the ObjOfInterest mix in nginx and Postgres. Every allocated object is accessed and kernel optimizations such as THP [27] are disabled to prevent 2 MiB rounding. We exclude the tracer library so the comparison isolates `Ichnaea`'s allocation strategy itself; `Ichnaea`'s allocator interposer adds a further ∼500–600 ns/alloc, independent of object size.

Table 4 shows the cost of using paged isolated slabs for ObjOfInterest allocation. RSS here stands for resident set size which is the true amount of physical ram in use by the program in contrast to virtual memory which could be higher. The heading row shows *packed* (normal allocation where objects are packed together) vs *isolated* (page aligned and page isolated allocations) and the last column shows the ratio between the two. The first row shows the total RSS for both packed and isolated configurations. For *packed*, since objects sized were random and small , the total size of 10,000 objects comes to 4.2MiB. For *isolated* section, however, since each of those 10,000 objects occupy almost two full pages, the cost is ∼80 MiBs (10,000 × 8 × 1024). The second row shows memory overhead per object which is ∼8 KiB/object, twice the 4 KiB lower bound due to kernel bookkeeping (fresh VMAs and page-table entries) when page-aligned requests bypass the `glibc` arena. The ratio here is skipped as sizes of packed allocations vary. The overhead is bounded and linear in N, independent of object size. The third row shows the per allocation time overhead for both configurations, a 7.6× increase is due to the fact that the `glibc` allocators usually have to make a special kernel request for each isolated object allocation. Page-per-object isolation is therefore the right default as the overhead and side effects (`glibc` allocates critical data structures on heap causing problems if those structs segfault) of not isolating could be much higher. Furthermore, such analyses are usually done offline where these overheads are acceptable. For campaigns tracking $\gtrsim 10^3$–10^4 small heap objects, the suballocator sketched in appendix A.7 that packs multiple ObjOfInterests into a shared isolated region that reduces both memory amplification and overhead per allocation. This could be an avenue for future improvements.

5.6 Detailed Analysis of Overheads

We identify 2 sources of overheads introduced by `Ichnaea`:

1. **Runtime Library Overhead**: even when no ObjOfInterests are accessed, `Ichnaea` incurs a one-time startup cost from loading the `LD_PRELOAD`-ed runtime library, plus a continuous cost from its interceptors on allocators and libc functions, which add function-call and context-switching overhead on every invocation.
2. **Object Access Overhead**: the additional latency incurred for each access operation on an ObjOfInterest.

5.6.1 Runtime Library Overhead

We measure the runtime of `Ichnaea` without tracing any objects to measure the overhead added by the `alloc` and `libc` wrappers alone. We do this by running the SPECInt benchmark suite with `Ichnaea`'s runtime library (`libIchnaea`) loaded but without tracing any objects. By design, the runtime library only interposes on allocators and syscalls after the first ObjOfInterest registration. For this experiment, however, we explicitly enabled all interceptions without requiring API calls to measure their overheads in the absence of tracing.

Figure 6 compares SPECint runtimes with and without traced objects under `Ichnaea`. The y-axis shows median runtime normalized to the un-instrumented baseline, where a value of 488 indicates a 388% slowdown. For each benchmark, the dotted green line is the baseline, the yellow bar is `Ichnaea` loaded but with no traced objects and the orange bar is `Ichnaea` with traced objects (for details on selected objects, see Section 5.2 for details).

The experiments show that our runtime library adds overheads even without any traced objects as all allocator functions and select syscalls are still intercepted. The percentage variability in the overheads of un-instrumented vs un-traced runs is due the frequency and the nature of syscalls used in the different workloads. Workloads that tend to use syscalls (*e.g.,* `read()/recvfrom` *etc.*) or heap allocators (*e.g.,* `malloc`) frequently incur higher overhead even when there are no traced objects. But the relative overhead is still low as syscalls are usually way more expensive than a wrapper indirection.

5.6.2 Object Access Overhead

To quantify the cost of writing to ObjOfInterests, we measured the CPU time for each of the event sequence from →attempted

Table 5: Breakdown of fault handling overheads in microseconds (2.4 million writes). Fast path only.

Stage	Name	Description	Fast Path CPU time (μs)		
			Median	99th	Top 1%
1	Handler launch	Time from access to SIGSEGV handler entry	2.2	3.0	6.0
2	`SIGSEGV` Handler	Time from SIGSEGV handler start to return	6.1	16.4	18.2
–	**Total time**	Attempted access to normal exec (end to end)	**9.0**	**19.5**	**21.9**
Metric	**Related Metrics**				
1	`mprotect()` call	Linux `mprotect()` (avg of R/W & RO updates)	2.0	4.5	5.9
2	R/W Window	Untraced window (R/W→AD transition)	4.0	8.8	11.2
3	`pkey_set()` call	Cost of `pkey_set()`	0.05	0.08	0.1

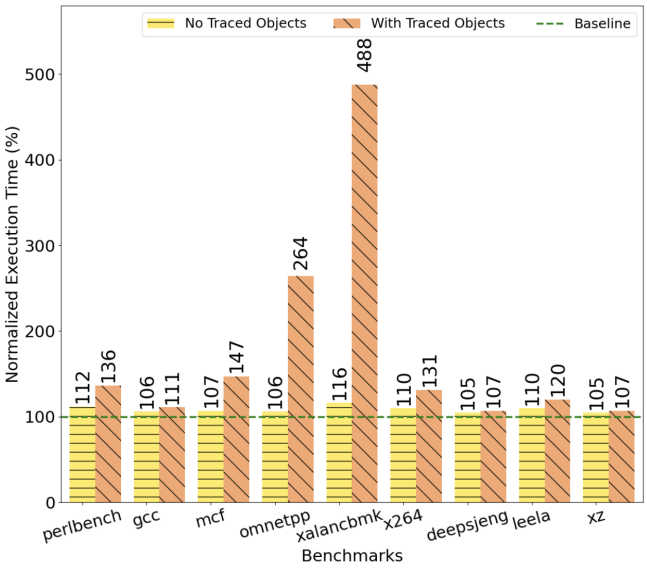


Figure 6: workload runtime comparisons between a `Ichnaea` run *with* and *without* any traced objects with SPECInt suite. Green line is un-instrumented runtime.

ObjOfInterest access →`SIGSEGV` handling →resumption of normal execution. Table 5 reports the CPU times for each stage of the handler (see Section 4.1 and Figure 3). Each row table shows the median, 99th percentile and mean of the top 1% of samples, with times in microseconds. Results are based on 2.4 million writes generated by running a custom write-intensive workload 200 times (1200 writes per run), with writes evenly split between global and heap objects to capture diverse access patterns. We only trace writes as they are more expensive to trace than reads as the written data must be stored in the trace memory pool but reads do not.

5.7 Realistic Use Case

We now explore a realistic use case — fuzzing — used to improve coverage in dynamic analysis use-cases. Table 6 summarizes the performance of *AFL++* [12] fuzzing our custom fuzzing workload[10] under three configurations: native execution (no instrumentation), with `Ichnaea` and with Intel Pin. X/Y means, X out of Y. Did not finish (DNF) means that even after 12 hours, the fuzzer didn't converge to a single Object access, so we stopped the experiment. The synthetic fuzzing target was designed to be memory access intensive with multiple branches, 20% (17) of which led to an ObjOfInterest access. All experiments were run with the same seed for fair comparison. Here, *covered* indicates that the path containing the ObjOfInterest was exercised by the fuzzer, while *traced* indicates that a ObjOfInterest access was successfully captured by the tracer. Time indicates the time it took for the fuzzer to exercise all the sites with ObjOfInterests.

Table 6: Fuzzing configurations: coverage, tracing, throughput and runtime.

Config	Covered ObjOfInterests	Traced ObjOfInterests	Time
Only fuzzer	17/17	**0**/17	4 min
Fuzzer + `Ichnaea`	17/17	**17**/17	25 min
Fuzzer + Intel Pin	**0**/17	**0**/17	DNF

In the experiments, the native fuzzer exercised all paths containing ObjOfInterests within 5 minutes, *without tracing any of them*. With `Ichnaea`, the same coverage was achieved in *25 minutes* and *all 17 accesses to ObjOfInterests were captured*. In contrast, Intel Pin failed to reach any ObjOfInterest paths even after 12 hours. These results show that while `Ichnaea` imposes overheads compared to the native configuration, it enables fuzzing assisted dynamic object tracing and makes it practical. Intel Pin, however, suffers from prohibitive slowdown and integration complexity, making it infeasible for the use case. Adapting Intel Pin was only possible through *AFL-pin* [46], a third-party wrapper that forwards AFL's fork-server and coverage feedback through Pin's instrumentation. This addresses research question `R2`, showing `Ichnaea` provides a

[10] See our website for details

feasible path to fuzzing assisted dynamic object tracing.

6 Discussion

Stack Objects are difficult to trace since locking their backing page would trigger a flood of pagefaults, as most program activity happens on the stack. Padding with variable length arrays [16] (VLAs) to push the ObjOfInterest onto its own page is one workaround, but compiler-driven memory reordering can defeat the isolation.

Tracing Kernel Reads requires detailed parsing of all syscall interfaces to determine which memory regions are being accessed. This is a huge engineering effort. However this is not a fundamental challenge.

Exercising the Program for Dynamic Analysis completely and in a sound manner is an undecidable problem. However for our purpose test suites, benchmarks and fuzzing can generally yield acceptable results.

Engineering limitations, `Ichnaea` has known limitations *e.g.,* bugs, un-optimized code, avoidable 6 μs interrupt handling time (see table 5, stage 2) due to expensive `libunwind` calls, incomplete instruction emulation (see appendix A.9) and `libc` syscall coverage limited to those exercised by our workloads. None are fundamental to our approach and can be addressed with additional engineering effort.

Tracing All objects Since `Ichnaea` is designed to trace a small, predefined subset of the total number of objects, the number of such targets `Ichnaea` can track, while maintaining the low overheads, is highly variable. If we trace all objects, `Ichnaea` may incur overheads higher than some of the existing tools *as every access is now an interrupt* each of which gets quite expensive. The exact point where instrumentation-based approaches overtake `Ichnaea` is dependent on multiple factors such the number of traced objects and their average access frequency.

7 Related Work

Compiler Based Instrumentation techniques *e.g.,* ASAN [38], Gleipnir [19] rely on instrumenting every load and store which effectively has the same overhead since after every load and store, the program needs to check all memory operands of the instruction against all ObjOfInterests, if there is a match, a trampoline call to log accesses and collect metadata. One can also reduce the number of instructions that need instrumentation by pruning it out from routines that can be deterministically said to not modify any ObjOfInterest. However, these pruning techniques fail or suffer from path explosion in complex codebases like postgresSql and nginx that heavily rely on indirect control flow etc. Furthermore, the complexity of adding all these compiler passes to first instrument and then prune is non-trivial and significantly discourages usage. All dynamically loaded libraries need to be instrumented, pruned and compiled into the main binary. We did not find any compiler based tool that was complete and did this with justifiable amount of effort.

Dthreads [28] is tool that enforces deterministic multi-threading for C/C++ program. It uses virtual memory protection technique to keep a check on shared data access. However, to fix the problem of false sharing, they have to split a multi-threaded program into a multiple programs sharing data which may not be idea or even feasible for many applications. Other solutions [3, 20–22] using similar techniques to tracing various things using virtual memory protections also suffer from loses described in section 2.1.3.

Hardware debug registers in X86 [17], ARM [4] also provide a fast and efficient way of tracing often used in debuggers like gdb watchpoints [13] and in [18] to isolate and monitor certain memory regions that contain secret keys. However, there's only a few per-thread word-sized watch points that trap on every access, making them too scarce, fine-grained and context-local to trace multiple (more than 4) arbitrary-sized objects. In gdb, when the system runs out of the few hardware debug registers, it switches to single stepping mode (basically looking at every load and store). Further, combining tools like gdb with test-suites, benchmark and fuzzers may require significant amount of effort.

Acknowledgments

We would like to thank our shepherd, Junfeng Yang (Columbia University), and the anonymous reviewers for helping us improve the paper. We wish to thank Prof. Gabriel Parmer and Gustavo Londono for discussions and feedback. This material is based upon work supported in part by the United States Air Force and DARPA under Contract No. FA8750-24-C-B031, the National Science Foundation (NSF) CPS CAREER Award 2246937, the Department of Energy (DoE) contract 49680-1-CCNS22589F and the George Washington University. Any findings, opinions, recommendations or conclusions expressed in the paper are those of the authors and do not necessarily reflect the views of sponsors.

References

[1] Cve-2021-23017: Nginx resolver vulnerability. https://nvd.nist.gov/vuln/detail/CVE-2021-23017, 2021. Accessed: 2025-11-20.

[2] AlDanial. cloc: Count lines of code. https://github.com/AlDanial/cloc, 2025. Accessed: 2025-02-12.

[3] A. W. Appel, J. R. Ellis, and K. Li. Real-time concurrent collection on stock multiprocessors. In Proceedings of the ACM SIGPLAN 1988 Conference on Programming Language Design and Implementation, PLDI '88, page

11–20, New York, NY, USA, 1988. Association for Computing Machinery.

[4] Arm Ltd. Non-invasive debug — arm debug architecture. https://developer.arm.com/documentation/ddi0406/b/Debug-Architecture/Introduction-to-the-ARM-Debug-Architecture/About-the-ARM-Debug-architecture/Non-invasive-debug, 2008. ARM Architecture Reference Manual, DDI 0406B. Accessed: 2025-11-21.

[5] ARM Ltd. ARM Architecture Reference Manual, ARMv8, for the ARMv8-A Architecture Profile, 2022. Includes ARMv8.5-A Memory Protection Keys for EL0.

[6] Roberto Baldoni, Emilio Coppa, Daniele Cono D'elia, Camil Demetrescu, and Irene Finocchi. A survey of symbolic execution techniques. ACM Comput. Surv., 51(3), May 2018.

[7] Thoms Ball. The concept of dynamic analysis. ACM SIGSOFT Software Engineering Notes, 24(6):216–234, 1999.

[8] D. Bruening, T. Garnett, and S. Amarasinghe. An infrastructure for adaptive dynamic optimization. In International Symposium on Code Generation and Optimization, 2003. CGO 2003., pages 265–275, 2003.

[9] Jonathan Corbet. Page-table hardening with memory protection keys. Linux Weekly News (LWN.net), 2025. Accessed: 2025-11-23.

[10] Patrick Cousot and Radhia Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages, POPL '77, page 238–252, New York, NY, USA, 1977. Association for Computing Machinery.

[11] Dyninst Project. Dyninst: Binary instrumentation and analysis framework. https://github.com/dyninst/dyninst. Accessed: 2025-11-21.

[12] Andrea Fioraldi, Dominik Maier, Heiko Eißfeldt, and Marc Heuse. AFL++ : Combining incremental steps of fuzzing research. In 14th USENIX Workshop on Offensive Technologies (WOOT 20). USENIX Association, August 2020.

[13] Free Software Foundation. Debugging with GDB: The GNU Source-Level Debugger, Version 5.1.1. GNU Project, 2002. Section: Setting Watchpoints. Accessed: 2025-11-21.

[14] Free Software Foundation. The LD_PRELOAD Environment Variable — The GNU C Library. GNU Project, 2024. Section: Dynamic Loader Environment Variables. Accessed: 2025-11-21.

[15] Damian Giebas and Rafał Wojszczyk. Detection of concurrency errors in multithreaded applications based on static source code analysis. IEEE Access, 9:61298–61323, 2021.

[16] GNU Project. Gcc online documentation: Variable length. https://gcc.gnu.org/onlinedocs/gcc/Variable-Length.html. Accessed: 2025-02-12.

[17] Intel Corporation. Intel 64 and ia-32 architectures software developer's manual, section 18.2: Debug registers. https://www.intel.com/content/dam/support/us/en/documents/processors/pentium4/sb/253669.pdf, 2023. Document Number 253669, Accessed: 2025-11-21.

[18] Jinsoo Jang and Brent Byunghoon Kang. In-process memory isolation using hardware watchpoint. In Proceedings of the 56th Annual Design Automation Conference 2019, DAC '19, New York, NY, USA, 2019. Association for Computing Machinery.

[19] Tomislav Janjusic and Krishna Kavi. Gleipnir: A memory profiling and tracing tool. ACM SIGARCH Computer Architecture News, 41(4):8–12, 2013.

[20] Sheetal V. Kakkad and Paul R. Wilson. Address translation strategies in the texas persistent store. In Proceedings of the 5th Conference on USENIX Conference on Object-Oriented Technologies & Systems - Volume 5, COOTS'99, page 8, USA, 1999. USENIX Association.

[21] Roman Kashitsyn. Ic internals: Orthogonal persistence. 2022-04-28. Accessed 2025-07-22.

[22] Haim Kermany and Erez Petrank. The compressor: concurrent, incremental, and parallel compaction. SIGPLAN Not., 41(6):354–363, June 2006.

[23] James C. King. Symbolic execution and program testing. Commun. ACM, 19(7):385–394, July 1976.

[24] Saparya Krishnamoorthy, Michael S. Hsiao, and Loganathan Lingappan. Tackling the path explosion problem in symbolic execution-driven test generation for programs. In 2010 19th IEEE Asian Test Symposium, pages 59–64, 2010.

[25] Saparya Krishnamoorthy, Michael S. Hsiao, and Loganathan Lingappan. Tackling the path explosion problem in symbolic execution-driven test generation for programs. In 2010 19th IEEE Asian Test Symposium, pages 59–64, 2010.

[26] Hugo Lefeuvre, Nathan Dautenhahn, David Chisnall, and Pierre Olivier. Sok: Software compartmentalization. In 2025 IEEE Symposium on Security and Privacy (SP), pages 3107–3126, 2025.

[27] Linux Kernel Documentation. Transparent hugepage support. https://docs.kernel.org/admin-guide/mm/transhuge.html, 2026. Accessed: 2026-05-21.

[28] Tongping Liu, Charlie Curtsinger, and Emery D Berger. Dthreads: efficient deterministic multithreading. In Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles, pages 327–336, 2011.

[29] Benjamin Livshits, Manu Sridharan, Yannis Smaragdakis, Ondřej Lhoták, J Nelson Amaral, Bor-Yuh Evan Chang, Samuel Z Guyer, Uday P Khedker, Anders Møller, and Dimitrios Vardoulakis. In defense of soundiness: A manifesto. Communications of the ACM, 58(2):44–46, 2015.

[30] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. Acm sigplan notices, 40(6):190–200, 2005.

[31] Michael Kerrisk. mprotect(2) — Set Protection on a Region of Memory. man7.org Linux Manual Pages, 2024. Accessed: 2025-11-21.

[32] Michael Kerrisk (Ed.). mmap(2) — memory mapping system call. https://man7.org/linux/man-pages/man2/mmap.2.html, 2025. Accessed: 2025-12-10.

[33] Albert R Myers. A binary instrumentation tool suite for capturing and compressing traces for multithreaded software. 2014.

[34] Nicholas Nethercote and Julian Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. ACM Sigplan notices, 42(6):89–100, 2007.

[35] NGINX, Inc. Nginx: High-performance web server and reverse proxy. https://github.com/nginx/nginx, 2002–2025. Accessed: 2025-02-12.

[36] Oleksii Oleksenko, Dmitrii Kuvaiskii, Pramod Bhatotia, Pascal Felber, and Christof Fetzer. Intel MPX Explained: A Cross-layer Analysis of the Intel MPX System Stack. Proceedings of the ACM on Measurement and Analysis of Computing Systems, 2018.

[37] Eric Schulte, Michael D. Brown, and Vlad Folts. A broad comparative evaluation of x86-64 binary rewriters. In Proceedings of the 15th Workshop on Cyber Security Experimentation and Test, CSET 2022, page 129–144. ACM, August 2022.

[38] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitry Vyukov. Addresssanitizer: A fast address sanity checker. In USENIX ATC 2012, 2012.

[39] The SQLite project. How SQLite Is Tested, 1.0 edition, 2025-05-31. Accessed 2025-07-22.

[40] Standard Performance Evaluation Corporation (SPEC). Spec cpu
textregistered 2017 benchmark. https://www.spec.org/cpu2017/, 2025. Accessed August 18, 2025.

[41] Igor Sysoev et al. Nginx: The high-performance web server and reverse proxy. Nginx official project.

[42] The Linux Kernel Documentation. Seccomp bpf (secure computing with filters), 2025. Accessed: 2025-08-17.

[43] The Linux Kernel Project. Linux Kernel Memory Protection Keys (pkeys), 2024. Accessed: 2025-11-21.

[44] The PostgreSQL Global Development Group. Postgresql: The world's most advanced open source relational database, 1996-2025. Accessed: 2025-08-17; active open-source development and community.

[45] J. Torrellas, H.S. Lam, and J.L. Hennessy. False sharing and spatial locality in multiprocessor caches. IEEE Transactions on Computers, 43(6):651–663, 1994.

[46] vanhauser-thc. afl-pin: run afl with pintool. https://github.com/vanhauser-thc/afl-pin.git, 2025. Accessed August 19, 2025; GitHub repository under AGPL-3.0 license, 66 stars, 11 forks.

[47] V. Vipindeep and Pankaj Jalote. Efficient static analysis with path pruning using coverage data. SIGSOFT Softw. Eng. Notes, 30(4):1–6, May 2005.

A Appendix

A.1 Access control granularity

Since modern MMUs enforce protection at *page* granularity, Ichnaea must change access rights for an entire 4 KiB page even when tracing single 4-byte object. For global ObjOfInterests, unrelated objects sharing the same page will also fault on accesses. Ichnaea handles such collateral SIGSEGVs like traced objects but skips logging, limiting overhead to a handler entry-exit cycle. Workarounds exist but trade usability for precision. Dedicated ELF sections aligned to page boundaries avoid interference but require linker scripts; we provide a lightweight macro (ICHNAEA_ISOLATE_GLOBAL) to isolate globals dedicated sections (see Line #1 in Listing 2). Alternatives include wrapping variables in a struct/union, which is portable but intrusive or object renaming, where a global ObjOfInterest is aliased within a page-aligned structure —– effective but Clang-specific. Given these trade-offs, the default strategy is to tolerate rare collateral faults and rely on the handler's fast path to keep overhead low.

A.2 Cost of a *mprotect()* call

Changing page permissions with mprotect [31] is inherently expensive due to several micro-architectural and OS-level effects. First, each permission change requires a transition into the kernel, incurring a syscall and context-switch overhead. Second, updating page permissions forces the kernel to modify page-table entries, which triggers page-table walks and invalidates cached TLB translations. On multicore systems, the kernel must broadcast inter-processor interrupts (IPIs) to flush or synchronize TLB entries across all cores that may have accessed the page, introducing huge additional latency. Collectively, the syscall cost, page-table manipulation and cross-core TLB shootdowns make mprotect() orders of magnitude slow. A mprotect call on a single threaded application roughly costs 2.0µs (see *Related Metrics* in Table 5) which is a significant amount of time, this cost is further multiplied by the number of threads running in the program.

A.3 mprotect based object tracing

Traditional way of access control for memory uses the mprotect. Although it is lossy and slow, this technique is still faster than Intel Pin. To realize this, we modified Ichnaea to use mprotect instead of pkey_set() call to lock and unlock the ObjOfInterests. No further change was required.

A.4 Output Format of Trace

Listing 1 shows a sample of decoded object trace generated by Ichnaea running an Nginx with 50 traced objects. This snapshot shows the runtime state of an ObjOfInterest named *http_core*. It contains details such as the object's type *(ngx_module_t)*, size in memory and the number of snapshots taken so far. Each snapshot then captures the values of its fields (*e.g.,* indices, version, signature, context pointers and names), including both raw values (like numeric IDs or memory addresses) and any resolved symbolic information (such as function names).

Listing 1: Sample of a decoded run in json

```
"http_core": {
 "type":"ngx_module_t",
 "size": 200,
 "total_snapshots": 4,
 "snapshots": {
    "0": {
     "is_read" : false,
     "call_stack" : [......, main+0x55 (nginx) ],
     "pid" : 115200,
     "tid" : 177234,
     "data": {
       "ngx_uint_t␣ctx_index": 18446744073709551615,
       "ngx_uint_t␣index": 18446744073709551615,
       "char␣*␣name": 0,
       "ngx_uint_t␣spare0": [
           "0␣x0",
            {
               "resolved_fn_name␣":"Null"
            }
        ],
       "ngx_uint_t␣spare1": 0,
       "ngx_uint_t␣version": 1024000,
       "const␣char␣*␣signature": 103329564877648,
       "void␣*␣ctx": [
           "0␣x5dfa4a04e0e0",
            {
               "resolved_fn_name␣":"Unknown"
            }
        ],.......
```

A.5 Annotating Custom Allocators

Listing 2 shows how Nginx's custom allocator can be instrumented with Ichnaea 's macros. The ICHNAEA_ISOLATE_GLOBAL and ICHNAEA_MARK_PTR directives identify global pointers to be traced, while ICHNAEA_MARK_ALLOC_CHK tags allocator calls so that newly created objects are automatically tracked at runtime. This lightweight annotation ensures that heap-allocated ObjOfInterests can be identified and traced by the tracer.

Listing 2: Sample of Nginx's custom allocator instrumented with our macro

```
void * heap_obj_ptr ICHNAEA_ISOLATE_GLOBAL;
ICHNAEA_MARK_PTR(heap_obj_ptr);

ngx_pool_t *
ngx_create_pool(size_t size, ngx_log_t *log)
{
    ICHNAEA_MARK_ALLOC_CHK(size , ngx_pool_t * )
    ngx_pool_t  *p;
    ...........
    return p;
}
```

A.6 Example of LibC Wrapper

Listing 3 illustrates how a standard libc *read* call is wrapped to trace kernel writes. Before invoking the real read, the wrapper temporarily unlocks only those pages that contain ObjOfInterests, ensuring that kernel writes to traced objects can proceed safely. Once the syscall completes, the same pages are re-locked to restore protection. This mechanism guarantees that updates originating from the kernel are observed without disrupting normal I/O behavior.

Listing 3: Sample read libc call wrapper for tracing kernel writes

```
ssize_t read(int fd, void *buf, size_t nbytes)  {
  void * pg_aligned_addr_of_buffer =
    (void *)(((uintptr_t)buf) & ~(PAGE_SIZE - 1));

  wrapper_objsnf_unlock_all_objs_or_one(
    pg_aligned_addr_of_buffer
    );

  size_t rtn =
    wrapper_objsnf_real_read(fd, buf, nbytes);

  wrapper_objsnf_lock_all_objs_or_none(
    pg_aligned_addr_of_buffer
    );

  return rtn;
}
```

A.7 Improving Isolated heap allocations

Isolating heap objects is done by allocating an individual page to each dynamically allocated *ObjOfInterest* even if they are as small as a few bytes. This however wastes memory capacity and in case of workloads that allocate heap *ObjOfInterests* frequently, also requires the actual *libc allocators* to frequently request new pages from the kernel that in turn reduces performance by adding frequent *syscall* overheads. This can be solved by keeping track of what parts of isolated memory that was allocated for an *ObjOfInterest* is unused and use it for allocating future *ObjOfInterests* as well. This is possible with some engineering effort as doing this is equivalent to writing another *malloc* for *ObjOfInterests*. However we do not implement this in our prototype. Details on how we handle heap objects can be found in Section 4.5.

A.8 Feature Comparison with intel Pin

While raw performance is one of the major advantages of our approach, Table 7 highlights how `Ichnaea` also differs in feature coverage and usability. `Ichnaea` uniquely supports tracing kernel writes, a capability Pin doesn't support[11]. Both Pin and our framework can trace global and heap objects. Both Intel Pin and `Ichnaea` don't support tracing of stack variables. However, with significant engineering effort and compiler tricks it is possible for both cases. Neither tool requires manual discovery of write sites. Both Intel Pin and `Ichnaea` can identify dynamic allocations of ObjOfInterests. However, Pin does not support dynamic allocation out of the box.

Finally, with respect to setup and startup, both frameworks are simple to initialize in principle, but Pin's startup phase incurs high cost compared to the lightweight startup of `Ichnaea`. Together, these comparisons emphasize that our contributions go beyond speed alone — they also expand what can be traced and reduce the practical burden on users while providing rich context info. This further answers **R1** since making tracing easily usable and efficient increases its adoption in software pipelines thus improving software reliability.

Table 7: Feature comparison between Intel Pin and `Ichnaea`.

Feature	**Intel Pin**	**`Ichnaea`**
Supports Global / Heap / Stack	Yes / Yes / No	Yes / Yes / No
Needs manual write-site discovery?	No	No
Needs manual alloc-site discovery?	No	No
Lossy?	No	No
Runtime cost	High	Very Low
Supports Fuzzing	No	Yes[12]
Kernel writes tracing	No	Yes
Setup cost	Moderate	Low
Startup cost	High	Low

A.9 Fallback strategy to instruction emulation

Due to complexity, emulating all x86 instruction is challenging and requires considerable engineering effort. When `Ichnaea` encounters such instructions, it employs a slower fallback path by planting a breakpoint at the instruction immediately following the faulting one and then allowing the original instruction to execute on the CPU after making the ObjOfInterest accessible. This approach introduces a subtle hazard in multi-threaded workloads: the injected breakpoint resides in the process's shared `.text` section, making it visible to all threads. If another thread happens to be executing the same code path while operating on unrelated data it will still hit the breakpoint and be diverted into the handler, potentially corrupting execution despite object independence. Although this scenario requires both code-path overlap and precise timing between the writing thread and other concurrent threads and is therefore unlikely and hard to measure, it cannot be ruled out even when each thread operates on disjoint objects since the code segment itself is inherently

[11] Tracing kernel writes is possible in pin by mimicking our technique, however, due to pin's design it will incur significant overhead thus reducing usability even further

[12] `Ichnaea` supports fuzzing with, see Section: 5.7

Listing 4: Invocation Example of Ichnaea

```
#include <stdio.h>
......
// ------ Inclusion of our header file ---------
// The header file defines weak refs to our main API
// calls to signal runtime definition
#include <.../ichnaea.h>
....
double object_of_intrest;

int main () {
    // ------ API Usage Example ---------

    // Runtime check to verify availability of API
    if (ichnaea_register_object) {
        // Main API Interface
        ichnaea_register_object(
              &object_of_intrest,
            sizeof(object_of_intrest),
            "Object␣of␣Intrest",
            "double"
        ); ......
    }}
// ------ Invocation example ---------

Compile the program under test
user@machine:~$ make -j`nproc`

Run the program under test using:
~$ LD_PRELOAD=libIchnaea.so ./my_binary ..
```

shared. However, we can get rid of this problem given enough engineering effort.

A.10 Invocation

A sample of `Ichnaea`'s invocation procedure is shown in listing 4

A.11 Slow Path (Breakpoint-Based Handling).

If decoding or emulation fails (*e.g.,* atomic instructions, SIMD stores), the handler falls back to a *"slow"* path. The slow path can be eliminated with engineering effort as it causes some unconventional problems. The slow path does the following:*i.e.,*

1. *instruction analysis:* the faulting instruction is disassembled to determine its length

2. *breakpoint insertion:* a software interrupt (also called breakpoint or INT3) is injected immediately *after* the instruction; the existing byte of memory that the interrupt replaced is saved in a per-thread interrupt context

3. *temporary unlock:* the page containing the object is marked RW

4. *resume execution:* the faulting instruction *re-executes* and completes the write natively. Execution then continues to the next instruction which is the software interrupt we inserted in step (2).

The subsequent interrupt `SIGTRAP` is again handled by our custom interrupt handler . The handler then does the following:

1. *Relock:* the object's page is returned to RO.

2. *Restore instruction:* the original byte replaced by the breakpoint location is restored from interrupt context.

3. *Logging:* if the write target is a traced object, the store is logged into the memory pool.

4. *Resume execution:* the instruction pointer is set to the original restored instruction.

5. *Exit:* at ProgramUndrTest's exit, `Ichnaea` gathers all snapshot data collected in the memory pool and writes it to disk.

A.12 Handling Custom Allocators.

The mechanism assumes that the application invokes the *glibc* allocator directly (*e.g., malloc calloc/realloc*). However, if a workload introduces its own layer on top of *libc allocs*, (*e.g.,* a `custom_malloc()` that wraps `malloc()`), `Ichnaea` only sees the intermediate function and cannot match the request to a traced variable. In such cases, developers must annotate the helper itself using our macro. Listing 2 in appendix shows an example with `nginx`, where the macro `ICHNAEA_MARK_ALLOC` is applied to custom allocators. The macro takes the requested size and type cast as arguments and if the allocation is for a traced object, it overrides the custom allocator to return a private, isolated page. If the user needs to modify the allocated memory in their custom allocator, we also provide a macro that returns a boolean that can be used to write custom allocation logic for ObjOfInterest allocation. Our original mechanism is easily adaptable to custom allocators using simple macros (see Listing 2 for details).

ARTIFACT EVALUATED
USENIX
AVAILABLE

Extracting Database Access-Control Policies From Web Applications

Wen Zhang* *UC Berkeley*
Dev Bali *UC Berkeley*
Jamison Kerney *UC Berkeley*
Aurojit Panda *NYU*
Scott Shenker *UC Berkeley and ICSI*

Abstract

To safeguard sensitive user data, web developers typically rely on implicit access-control policies, which they implement using access checks and query filters. This ad hoc approach is error-prone as these scattered checks and filters are easy to misplace or misspecify, and the lack of an explicit policy precludes external access-control enforcement. More critically, it is difficult for humans to discern what policy is embedded in application code (i.e., what data the application may access)—an issue that worsens as development teams evolve.

This paper tackles *policy extraction*: the task of extracting the access-control policy embedded in an application by summarizing its data queries. An extracted policy, once vetted for errors, can stand alone as a specification for the application's data access, and can be enforced to ensure compliance as code changes over time. We introduce Ote, a policy extractor for Ruby on Rails web applications. Ote uses concolic execution to explore execution paths through the application, generating traces of SQL queries and conditions that trigger them. It then merges and simplifies these traces into a final policy that aligns with the observed behaviors. We applied Ote to three real-world applications and compared extracted policies to handwritten ones, revealing several errors in the latter.

1 Introduction

Protecting sensitive data from unauthorized access is a critical concern for today's web applications. Therefore, when building web applications, developers must determine what *access-control policy* the application should enact—for example, a university might want a policy that ensures a student's grades are visible only to the student and their instructors.

In today's applications, access-control policies are embedded in application code. Furthermore, in most cases they are spread across several functions and in the filter predicates of multiple database queries. This practice is error-prone: Missing or misspecified access checks have previously led to sensitive-data exposure [10, 36, 46, 47, 57, 89]. But more fundamentally, because the policy is never stated explicitly, it is difficult for anyone other than the application's developer to understand *what* policy is embedded in the code. Worse, as time passes, even the application development team is unlikely to remember the policy, and is unlikely to be able to reconstruct it from application code. While there have been research frameworks that require explicit policy specification [2, 6, 11, 105, 106], we aim to address legacy applications rather than requiring them to be rewritten in such frameworks.

This paper tackles the task of *policy extraction*: extracting a web application's implicitly embedded access-control policy by summarizing its possible data accesses. A human then reviews the extracted policy to better understand the application's data accesses, ensuring they are within the bounds of intended data revelations. If not, the application likely has an access-check bug to be fixed. Once reviewed, the policy can stand alone as a specification for the application's data accesses and can optionally be enforced using an enforcer [56, 58, 107] to ensure continued compliance.

We present an approach for extracting policies from legacy web applications. Our approach begins by exploring execution paths through application code using concolic execution [35, 83], producing transcripts that record the conditions under which SQL queries are issued (§ 4). These transcripts are then merged and simplified to derive a policy that allows each recorded query to be issued under its conditions (§ 5).

A key challenge here is scalability: Web-backend code is often complex with many branches, making exhaustive path exploration infeasible. But we empirically observe that the logic governing query issuance typically relies on only a small set of simple operations (§ 4.1). We thus tailor concolic execution to track only those operations, reducing the path space (and implementation effort). We further prune exploration using an LLM-based relevance judge (§ 4.6) that identifies and ignores branches unrelated to data access, reducing exploration time from potentially days to mere hours.

We implemented this approach in Ote, a policy-extraction tool for web applications written in Ruby on Rails. We

*Now at Google.

then applied Ote to three real-world applications, two of which we had previously written policies for by hand [107]. When we compared the extracted policies to the handwritten ones (§ 7.6), we identified several formerly unknown errors in the latter—including a few overly permissive views that reveal sensitive data to unauthorized users. This underscores the difficulty of understanding access-control logic in complex legacy applications and the utility of policy extraction in aiding this understanding. A further review of the extracted policies uncovered a subtle bug we had inadvertently introduced into application code that silently disabled an access check. These findings show that extracting and then inspecting policies results in a clearer understanding of what the program does—and should do—than current practice provides.

Due to its use of concolic execution and an LLM relevance judge, Ote cannot guarantee that the extracted policy covers all possible queries or captures every condition under which queries are issued (§ 3.2). Ote also needs the user to specify certain input constraints (§ 3.1) and to review the relevance judge's results (§ 4.6), supports only a subset of SQL (§ 3.2), and could use a better user interface for policy auditing. Even so, our evaluation (§ 7) demonstrates that for real-world applications, Ote extracts policies that are immediately useful, identifying several errors in handwritten policies and application code. These results show that Ote already provides concrete, practical value and represents a meaningful step toward improving data security in existing web applications.

2 Motivation and Background

2.1 Challenges of Policy Creation

We were prompted to tackle the policy-extraction problem by our earlier experience hand-crafting policies for existing web applications. A few years ago, while investigating externally enforced access control for web applications,[1] we took open-source applications, chose several representative URL endpoints, and tried our best to write down policies allowing the data accesses needed for their intended function [107, § 8.1].

This process was extremely tedious and time-consuming. We reviewed documentation, inspected database schemas, experimented with the applications using sample data, and read their source code. Using this information and common sense, we formulated policies that we thought would allow the endpoints to function correctly while protecting sensitive data. Despite our best efforts, we often discovered errors in our drafts, including an omission that would have leaked sensitive data—one that we only later discovered by chance.

Our struggle was not unique: Policy creation is known as a major challenge across security domains including role-based [69, § 6.5], attribute-based [27, p. 39], and relationship-based [20] access control, securing cloud resources [81], and sandboxing syscalls [71]. While solutions have been proposed for some domains (§ 8), generating database policies for web applications has remained an open problem that is uniquely challenging, due to both the dynamism of web applications and the fine granularity of the policies required.

[1] By "externally enforced", we mean having an enforcer program mediate the application's database queries, preventing accesses deemed unauthorized according to an access-control policy [49, 56, 58, 107].

Listing 1 A handler that displays a course's grade sheet.

```
def view_grade_sheet(db, session, req):
  role = db.sql(
    "SELECT * FROM roles "
    "WHERE user_id = ? AND course_id = ?",
    session["user_id"], req["course_id"])
  if role is None: raise Http404
  if not role.is_instructor: raise Http403
  all_grades = db.sql(
    "SELECT * FROM grades "
    "WHERE course_id = ?", role.course_id)
  return format_html(all_grades, ...)
```

Listing 2 An example policy for the handler in Listing 1.

(V_1)
```
SELECT * FROM roles
WHERE user_id = MyUserId
```
A user can view their role (if any) in any course.

(V_2)
```
SELECT grades.* FROM roles, grades
WHERE roles.user_id = MyUserId
  AND roles.is_instructor
  AND grades.course_id = roles.course_id
```
An instructor for a course can view all grades.

2.2 Policy as SQL View Definitions

Before delving into how Ote extracts policies, we first describe how our access-control policies are specified.

We target web applications that store data in a relational database; when a user visits a page, the application issues SQL queries on the user's behalf and renders the page using the query results. In this context, a classic way to specify access-control policies is to use a list of *view definitions* [62, 78, 79], which are SQL `SELECT` statements—parameterized by session parameters like the current user ID—that define information in the database that a user is allowed to access.

Under a view-based policy, a query is allowed only if it can be fully answered using the views. This criterion extends to a program that (conditionally) issues multiple queries, by treating the program as "one big query" that returns the results of its constituent queries. These notions can be made precise based on *query determinacy* [65], but in the interest of space, we omit the formal definitions and offer an example instead.[2]

Example 2.1. Suppose a course-management site has a web request handler that displays a course's grade sheet (Listing 1).

[2] We refer interested readers to the literature for both theoretical [3, 50, 65, 109] and practical [14, 15, 78, 107] treatments of this subject.

It ensures that the user is an instructor before fetching grades.

Note that the handler has access to both *session parameters* (user ID) and *request parameters* (course ID). Session parameters are trusted (e.g., set by an authentication mechanism), and may appear in the policy and dictate the extent of allowed data access. Request parameters are untrusted (e.g., parsed from an HTTP request) and must not appear in the policy.

A policy for this handler might look like Listing 2, where `MyUserId` denotes the user-ID session parameter. Notably, it does not reference the course-ID request parameter, instead allowing the handler's queries *for any course ID*. This policy precisely captures the information the handler can query. ◀

Like prior work in database access control [4, 14, 15, 49, 56, 78, 86, 107], we focus on extracting policies for database *reads* (`SELECT`s) only. Similar techniques can be used to extract conditions for other operations, although our policy language (§ 3.2) and algorithms (§ 5) would need to be extended.

3 Overview

Given an application, the ideal policy extractor would produce a view-based access-control policy that satisfies:

Completeness Allows all queries the application can issue.

Tightness Reveals as little information as possible subject to completeness.

Conciseness Has a short representation in SQL.

For example, given Listing 1 (but written in a real web framework), we would like to extract the policy in Listing 2.

Policy extraction is challenging, and Ote is not guaranteed to meet all three goals (see § 3.2). Nevertheless, we show in § 7 that Ote produces policies that are useful in practice.

3.1 Workflow

Before discussing how we approach these goals, we first describe Ote's workflow from a user's perspective (Figure 1).

Suppose a user wants to extract a policy from a web application. We assume that the application is written in a supported framework (Ruby on Rails) and that the user is familiar with the application's functionality.

The user starts by declaring the handlers to analyze and providing the application's *database constraints*. Ote supports two forms of database constraints, which cover all the constraints we encountered:

1. *A set of columns in a table is unique.* For example, a uniqueness constraint on `roles(user_id, course_id)` states that a user has at most one role in a course.
2. *A query Q_1's result is contained in a query Q_2's result.* Such constraints are commonly used to express foreign-key invariants: e.g., every `assignments.course_id` must appear as some `courses.id`.

These constraints describe the application's valid database states; Ote uses them to keep exploration within those states during concolic execution (§ 4.4) and to simplify the generated policy (§ 5.2). To reduce the user's burden, Ote provides a tool that can automatically generate most required constraints from a Ruby on Rails application's schema and models (§§ 6 and 7.1.3).

The user then packages the application into a Docker container and invokes Ote, which:

1. Explores execution paths through the handlers via concolic execution, producing *transcripts* that record the branches taken and the SQL queries issued (§ 4);
2. For each individual handler, analyzes the transcripts to generate a preliminary policy allowing each query to be issued under its recorded conditions (§§ 5.1 to 5.3);
3. Merges the individual policies and prunes any redundant views, producing a final policy (§ 5.4).

Next, the user inspects the generated policy, using their domain knowledge of the application's privacy requirements:

- A policy that is too broad can indicate an access-control bug. The user can investigate the queries that yielded a too-broad view using inputs logged by concolic execution (§ 6) and modify the application if appropriate.
- A policy that is too tight can be *broadened* to permit more intended data accesses. This may simplify the policy by removing filters or by replacing multiple views with a single one. Ote assists with policy broadening: The user adds a broader view to the policy, either before or after view pruning, and then invokes pruning (Step 3) to automatically remove now-redundant views (§ 7.7).

Once satisfied, the user can optionally enforce the policy to ensure the application's current and future compliance.

3.2 Assumptions, Scope, and Utility

Queries At its core, Ote supports project-select-join (PSJ) queries in set semantics. These are queries that (1) always return distinct rows, and (2) have the form:

```
SELECT [DISTINCT] col1, col2, ...
FROM tbl1, tbl2, ... WHERE ...
```
(PSJ)

Also supported are common queries that Ote can mechanically rewrite into this form, such as queries with inner joins or of the form `SELECT 1 FROM tbl1, tbl2, ... WHERE ... LIMIT 1`.

We found that the distinct-rows assumption does not limit utility—we have never encountered queries that may return duplicate rows in our evaluation. But applications do issue queries more complex than PSJ; this is handled differently in different stages: The concolic-execution driver precisely models richer SQL features (§ 4.4), but view generation and pruning must approximate complex queries using PSJ (§§ 5.3 and 7.3). We plan to extend our prototype to support more complex queries.

Policies Ote generates policies consisting of PSJ views, which are expressive enough for the applications we studied.

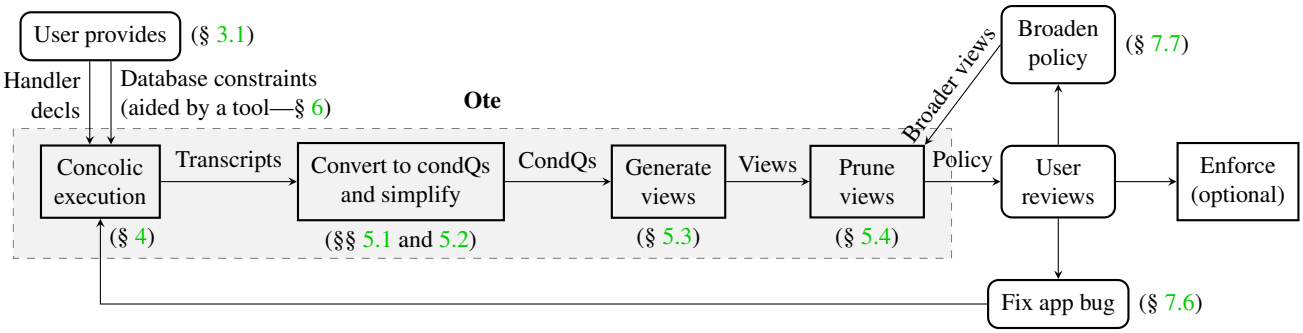


Figure 1: Policy extraction workflow. "CondQs" stands for conditioned queries (§ 5.1).

A notable limitation is that PSJ views cannot generally express negations (e.g., "Q_1 is allowed only if Q_2 returns no rows"). Negations complicate approximating a query's information content [97, §2.2] and can slow down enforcement [107, §6], so we defer handling negations to future work.

Non-guarantees Ote *does not guarantee* completeness or tightness: It may generate a policy that disallows a query issued by the application, or one that can be tightened while allowing the same application queries. This is because:

- Concolic execution can miss execution paths as the input space explored is bounded.
- Uninstrumented operations in the *query-issuing core* can lead to incomplete path conditions (§ 4.1).
- A pruning optimization uses an LLM-based relevance judge, which can make mistakes (§ 4.6).
- Approximating complex queries using PSJ can make a policy more, or less, restrictive than ideal (§ 7.3).

In general, complete-and-tight policy extraction for Turing-complete code is impossible [77]. But in practice, Ote can produce policies more accurate than handwritten ones (§ 7.6).

Similarly, Ote *does not guarantee* that it produces the most concise SQL representation, but its simplification and pruning steps (§ 5) make it feasible to inspect the policy (§ 7.6).

Application assumptions Ote uses a modified Ruby interpreter and Ruby on Rails framework for concolic execution (§ 4.5), and so it supports only applications written in Rails. However, our approach generalizes to other languages.

Ote's effectiveness depends on a "simple query-issuing core" assumption (§ 4.1). In short, we assume that the application's SQL query issuance depends only on *simple expressions*—ones that use only operations instrumented by Ote's Ruby interpreter. Most notably, Ote does not instrument string formatting (doing so would complicate SMT solving), and so it requires the application to issue SQL queries in parameterized form rather than splicing parameters into query strings within Ruby. Fortunately, idiomatic Rails code already issues parameterized queries by default, and the few places in our evaluation where this is not the case were easy to fix (§ 7.1).

4 Exploring Executions

Ote begins by exploring paths through application code via concolic execution. We open this section with some empirical observations about typical web applications that motivated our use of concolic execution.

4.1 Observation: Simple Query-issuing Cores

Given a web application's codebase, consider all program statements that issue SQL queries. Informally, imagine the backward program slice [100] from these statements. This slice, which we call the *query-issuing core*, is the part relevant to policy extraction: It consists of the program components on which query issuance has a control- or data-dependence. Thus, it includes code that computes query parameters or determines whether later queries are issued, but excludes code used only for localization, formatting, HTML/JSON generation, logging, etc. In Listing 1, for instance, the two queries (Lines 2 and 8) are in the query-issuing core, as are the emptiness and instructor checks (Lines 6 and 7) that determine whether the second query is issued; the final HTML-rendering line (Line 11) is outside the core.

While the codebase as a whole can be complex, it has been observed [24, 84] that the query-issuing core of a typical web application is often *simple*. We confirm this observation—for the applications we studied, the core consists mostly of:

1. Conditionals that test if a query's result set is empty [84, § 2] or check basic conditions on primitive values (e.g., checking for equality or nullity) [85, § 6.2];
2. Loops over a query's result set [84, § 2] with no loop-carried dependencies [24, § 4]; and
3. Trivial data-flow to query statements—e.g., passing a value returned by one query to another. (Given that the application issues parameterized SQL queries, no query-string formatting operation appears in the data flow.)

Our goal is therefore to analyze this simple query-issuing core without spending resources on the surrounding codebase. This is challenging because the query-issuing core is not syntactically isolated. For instance, in a typical Ruby on

Rails application, SQL queries may be issued from controller actions, view templates, view helpers, and external libraries, each of which can also contain substantial code unrelated to query issuance. To address this challenge, we use concolic execution, which allows Ote to analyze the query-issuing core without syntactically isolating it.

4.2 Concolic Execution: What and Why

In concolic execution [35, 83], a program is executed repeatedly using concrete inputs that have symbolic variables attached. As the program runs, its state is tracked both concretely and symbolically. When the program branches on a symbolic condition, the condition and its outcome are recorded. This produces a conjunction of constraints—the *path conditions*—that led execution down a path. The conjuncts are then negated using a solver to generate new inputs (up to a bound) that will steer execution down other paths.

We chose concolic execution because it offers a "pay-as-you-go" model for symbolic tracking: We selectively instrument the operations that might appear in the query-issuing core, and the rest will simply execute concretely by default. This strategy reduces not only our instrumentation effort, but also the number of paths explored—a branch on an uninstrumented condition will not lead to a new path.

Note that if an application's query-issuing core is *not simple*—so it contains uninstrumented operations—then Ote may miss certain queries or conditions, leading to a policy that is either broader or tighter than ideal. We discuss this case, with examples and mitigations, in § 4.7.

4.3 System Architecture

Ote has a *driver* that generates inputs and concurrent *executors* that run application code on each input. The driver tracks explored paths in a prefix tree; for every prefix, it negates the last condition and invokes an SMT solver to generate a new input. (The driver keeps the prefix tree in memory and generates inputs sequentially, but both can be relaxed to improve performance.) It then sends the input to an executor, which runs a handler using an instrumented Ruby interpreter and Ruby on Rails framework (§ 4.5) and sends back a *transcript* capturing the path conditions and queries issued (§ 4.4). Exploration terminates when all prefixes have been visited.

4.4 Symbolic Modeling and Input Generation

Concolic execution requires symbolically modeling the handler's inputs, consisting of the database and session/request parameters. To ensure termination, the input space must be bounded. Following prior work [23], we model the database as tables containing a bounded number of symbolic rows (our prototype uses a bound of 2). The driver asserts the application's database constraints (§ 3.1), so it explores only valid database states. Inspired by UrFlow's loop analysis [24, § 4.2], we also restrict the input space so that each query returns at most one row. For simplicity, we will describe our algorithms under this assumption, even though they can be extended to handle queries returning multiple rows.

Under this modeling, a transcript is a sequence of operations performed by the application with two types of records:

1. $\textsc{Query}_i(sql, params, isEmpty)$, meaning the i^{th} query issued was the query *sql* with parameters *params*, and the result set was empty if *isEmpty* is true.[3] If not empty, a symbol r_i is introduced to represent the result row.
2. $\textsc{Branch}(cond, outcome)$, meaning the condition *cond* was branched on, and the *outcome* (either true or false) branch was taken. The condition can reference session and request parameters as well as columns returned by previous queries (e.g., "$r_5.author_id = MyUserId$").

Listing 3 shows an example transcript from a run of the handler in Listing 1, when the user is an instructor.

Our SMT encoding represents bounded database tables using conditional tables [38] and uses the theory of integers to model all database values [37], including timestamps and strings. Each nullable value is accompanied by a boolean indicating if it is `NULL`. This simple encoding naturally supports equality and arithmetic operations but not string operations, a limitation that can be lifted using a string solver [16, 53].

Similar to prior work [37], we encode a subset of SQL into SMT on bounded symbolic tables. Our encoding supports left- and inner-joins and count- and sum-aggregations. One notable unsupported feature is ordering, which we have not needed under the assumption that each query returns at most one row.[4] We implement several optimizations for input generation: reusing Z3 AST objects, calling the solver incrementally, and caching conflicts (resulting in infeasible paths) using unsat cores [82].

4.5 Instrumentation and Tracking

To maintain symbolic state, we modified the JRuby interpreter [42] to add an optional "symbolic expression" field to each Ruby object. For each class that we want represented symbolically, we implement a `with_sym` method that returns a clone with a symbolic expression attached, and amend methods that we want instrumented to attach symbolic expressions to their results. Unmodified methods simply return an object with no expression attached, concretizing the result as desired.

We implemented symbolic representations of ten classes (`String`, `Fixnum`, etc.) covering all symbolic inputs in our evaluation, and instrumented simple operations (equality, null-check, etc.) that appear in the query-issuing core. We ensured that instances of a singleton object (`true`, `false`, `nil`) with

[3] Ote assumes that the application always inspects the result set of a query by branching on its emptiness, and does not track it as a separate BRANCH.

[4] Ordering cannot be ignored when a `LIMIT` clause can truncate a result set with multiple distinct rows; we encountered no such queries in our evaluation.

Listing 3 A transcript from a run of the handler in Listing 1, when the user is an instructor for the course.

1. QUERY$_1$(`SELECT * FROM roles WHERE user_id = ? AND course_id = ?`, $\langle$`MyUserId`, `CourseId`$\rangle$, $isEmpty = false$)
2. BRANCH($r_1.is_instructor, outcome = true$)
3. QUERY$_2$(`SELECT * FROM grades WHERE course_id = ?`, $\langle r_1.course_id \rangle, isEmpty = false$)

different symbolic expressions are treated as equal, and that a mutating method clears an object's symbolic expression by default.

To track queries and branches, we added a library for maintaining transcript records, exposing methods `record_query` and `record_branch` (§ 4.4). We modify the Rails database layer to call `record_query` after every query, and modify JRuby's `isTrue` and `isFalse` methods (which evaluate an object's truthiness and falsiness) to call `record_branch` with the object's symbolic expression if it has one. The `record` methods also save the stack trace at which the query or branch is evaluated, for use in relevance-based pruning (§ 4.6).

4.6 Skipping Irrelevant Branches

Concolic execution suffers from *path explosion*: The number of feasible paths can grow exponentially with the number of branches. Ote is no exception: Without further optimization, five web handlers in our evaluation (§ 7.3) failed to finish after ten hours and would likely have required days. Thus, we must aggressively reduce the paths explored, but without missing any query-issuing behavior.

Irrelevant conditionals Even with selective instrumentation (§ 4.2), many tracked branches lie outside the query-issuing core.[5] Consider the following example:

Listing 4 An irrelevant conditional from Autolab.

```
<% if submission.version == 0 then %>
  <font size=-2>Unofficial</font>
<% else %>
  <%= submission.version %>
<% end %>
```

The highlighted conditional affects only HTML generation and has no bearing on data access; negating such conditionals produces new paths that reveal no new query behavior. Skipping these *irrelevant* conditionals can reduce the exploration space dramatically, since each may cut the path count in half.

Identifying irrelevance using an LLM Statically identifying irrelevant conditionals is hard in Ruby on Rails [88] due to dynamic language features and runtime template compilation [72, § 6.2]. However, many irrelevant branches follow recognizable patterns—e.g., conditionals that issue no queries and mutate no state. Rather than manually encoding such patterns as heuristics, we draw on recent successes in using LLMs to complement software analysis [51, 63, 90, 95] and delegate the identification of irrelevant conditionals to a *relevance judge* implemented using an LLM-based coding agent.

[5] Even if the core is simple, not every simple condition is in the core!

Listing 5 An abridged relevance-judge output for the conditional in Listing 4, produced by `gpt-5` in 24 seconds (§ 7.5).

```
IRRELEVANT

The conditional `submission.version == 0` [...]
    only controls presentation text:
- If true, it renders the literal "Unofficial".
- If false, it renders the numeric `submission.
    version`.
[...]
```

When Ote encounters a conditional, it creates a prompt containing a definition of irrelevance, the conditional, and the stack trace at which the conditional was evaluated. It sends the prompt to a coding agent, which autonomously inspects source files and answers "relevant", "irrelevant", or "unsure" (treated as relevant), plus an explanation. The prompt is generic and contains no framework- or application-specific details. Ote records the verdict and explanation for later inspection; see Listing 5 for an example.

Because the relevance judge can take minutes to render a verdict (§ 7.5), Ote calls it asynchronously. Exploration carries on, assuming the conditional is relevant; if the judge later designates a conditional as irrelevant, the driver retroactively records it as such and takes it into account in the future.

Ote avoids calling the relevance judge when (1) the conditional's designation is known (e.g., "Q returns non-empty" is relevant if Q's result is used in another query), or (2) the conditional is *vacuous*—i.e., its negation is infeasible given the relevant conditionals that precede it. Because a conditional's vacuousness may depend on the relevance of its predecessors, when Ote marks a conditional as irrelevant, it recomputes the vacuousness of any formerly-vacuous later conditionals.

Pruning Using the judge's verdicts, the driver avoids negating irrelevant conjuncts in path conditions, skips paths whose subsequence of *relevant* conditions is already covered, and drops irrelevant conditions from the final transcript.

Accuracy Because the relevance judge uses an LLM, it cannot guarantee accuracy. To defend against mistakes, Ote exposes the verdicts and explanations for human review, and al-

lows the human to guide the judge by adding inline comments prefixed with "RELEVANCE-HINT", which the LLM agent is instructed to consider. In our evaluation (§ 7.1), we added hints primarily to compensate for the coding agent's limited understanding of Rails and external libraries. With these hints, we verified the judge's irrelevance verdicts to be accurate.

4.7 When the Core Is Not Simple

The simple query-issuing core assumption (§ 4.1) is not true for all practical applications. It can be violated in two ways.

Complex control-flow conditions This case arises when a query's issuance depends on a complex expression. For example, diaspora crashes if a photo's `url` is not a valid URL, thus preventing later queries; the URL-validity condition relies on regex-matching operations, which Ote does not instrument.

Suppose a query Q is issued conditioned on an expression C with some uninstrumented operations. This may cause Ote to generate a policy that is either broader or tighter than ideal:

1. If Ote happens to generate an input that satisfies C (this is not guaranteed because the driver is unaware of C), then Q will appear as a conditioned query *without* C in its condition, leading to a policy broader than ideal.
2. If Ote never generates an input that satisfies C, then Q may not appear in any transcript, leading to a policy that may not allow Q—which is tighter than ideal.

In practice, Case 1 has not been a problem for us because (1) the complex condition C typically reflects business logic and has no privacy implications (§ 7.7), and (2) query Q is likely also issued elsewhere under a simpler condition. Case 2 is problematic for validations (e.g., URL validity); we handle this by manually setting the input using database constraints (e.g., setting URLs to always be `http://foo.com`).

If an uninstrumented operation turns out important for an application domain, we would update Ote's Ruby instrumentation, SMT encoding, and SQL generation to support it.

Complex data flow This case arises when the transcript contains an expression derived from a symbolic value through uninstrumented operations—a problematic scenario because the symbolic value is concretized. For example, the conditional `if uid < uid * uid` may insert a BRANCH record for $uid < 16$ into the transcript, as multiplication is not instrumented. In such cases, the driver might fail to terminate because a "new path condition" emerges for every value of `uid`. Even if we cut off the exploration, the generated policy would be overly strict and verbose, littered with concrete filters like $uid < 16$.

To detect such harmful concretizations, the driver reports each new constant and new query it encounters (there should be a bounded number of these). This mechanism alerted us to a few non-parameterized SQL queries (§ 7.1), where symbolic values were formatted into the query string. We encountered no concretization issues in our final evaluation, and we plan to implement more precise detection of such "partially concrete" conditions via heavier-weight data-flow tracking.

5 Generating a Policy

After exploration, Ote merges and simplifies the transcripts and creates a preliminary set of views for each handler. It then gathers the views for all handlers explored and removes redundancy by leveraging an existing enforcement tool, Blockaid [107]. We now delve into this policy-generation process.

5.1 Preprocessing Into Conditioned Queries

As a first step, Ote processes the transcripts for each handler into a set of *conditioned queries*. A conditioned query is a tuple $\langle sql, params, conditions \rangle$, where *conditions* is a list of prior QUERY and BRANCH records.[6] It associates each query with the conditions under which it is issued; one conditioned query is generated for each query issued in each transcript. As Ote does not currently support negations in policies (§ 3.2), it drops from *conditions* any QUERY record with $isEmpty = true$ (i.e., a condition that a prior query returns empty).

5.2 Simplifying Conditioned Queries

For each handler, Ote simplifies the set of conditioned queries by removing redundancy. At a high level, the simplifications remove conditions that do not affect whether a query is issued.

For example, suppose two conditioned queries have the same SQL and parameters, but one is guarded by $C \wedge b$ and the other by $C \wedge \neg b$. Since the query is issued regardless of the branch outcome, Ote replaces the pair with a single conditioned query guarded only by C. Similarly, a conditioned query guarded by C subsumes one guarded by $C \wedge d$ with the same SQL and parameters. Database constraints expose another source of redundancy, allowing Ote to remove conditions that are guaranteed to hold, such as a prior lookup that must succeed because of a foreign-key dependency.

Ote performs these and other simplifications by following the steps shown in Algorithm 1:

- Remove BRANCHes that *must* be taken due to an input constraint or a prior condition (Line 2);
- Unify variables that are constrained to be equal by query filters (Line 3);
- Remove identical QUERY records from each conditioned query's *conditions* (Line 4);
- Drop *vacuous* QUERY records—for queries that *must* return a row due to, e.g., a foreign-key dependency—whose result is not subsequently referenced (Line 7);

[6] A prior QUERY represents the *condition* that a query Q returned empty or not; the *action* of Q's issuance is captured by Q's own conditioned queries.

Algorithm 1 Simplifying a set of conditioned queries (§ 5.2).

```
1:  for all conditioned query do
2:      remove vacuous branches
3:      propagate equalities
4:      remove duplicate queries
5:  repeat
6:      for all conditioned query do
7:          remove vacuous-and-unused query records
8:      repeat merge branches until convergence
9:  until convergence
10: remove subsumed
```

- Merge pairs of conditioned queries that differ only in the outcome of a single BRANCH record (Line 8)—the query is issued no matter which way the branch goes;
- Remove a conditioned query if another exists with the same *sql* and *params* but only a subset of the *conditions* (Line 10)—the latter subsumes the former.

Each step is parallelized across cores. Due to the large number of conditioned queries, we designed these steps to favor efficiency over optimality. Any missed opportunities for simplification will be caught by the final pruning step (§ 5.4).

5.3 Generating SQL View Definitions

Ote now generates one SQL view per conditioned query. The view should reveal the same information as the conditioned query's path if its conditions are met, and no information otherwise. For this, Ote uses an iterative algorithm that "conjoins" each condition record onto an accumulated query definition A. It maintains the invariant that query A:

- Returns empty if any previous condition is violated;
- Returns the Cartesian product of all prior queries' results (i.e., concatenations of one-row-per-query) otherwise.

Query A serves two purposes: It captures the branching conditions, and it exposes query results to be referenced by later records. Lastly, the algorithm conjoins the final query onto A.

Before fully specifying the view-generation algorithm, let us walk through an example.

Example 5.1. Consider Listing 3. We shall generate the view for the conditioned query associated with QUERY_2.

The algorithm starts with the query A that returns the empty tuple. After the first record (QUERY_1), A is updated to A_1:

```
SELECT * FROM roles
WHERE user_id = MyUserId
  AND course_id = CourseId
```

(We use CamelCase for request and session parameters.)

After the second record (the BRANCH), A is updated to A_2:

```
SELECT * FROM roles
WHERE user_id = MyUserId
  AND course_id = CourseId
  AND is_instructor
```

Algorithm 2 View generation from conditioned query (§ 5.3).

```
1:  procedure GENERATESQLVIEW(cq)
2:      A ← {⟨⟩}    ▷ Constant query returning empty tuple
3:      M ← {} ▷ Maps query result column to A's column
4:      for all cond ∈ cq.conds do
5:          if cond is BRANCH(θ, outcome = true) then
6:              A ← σ_{θ[M]}(A)
7:          else if cond is BRANCH(θ, outcome = false) then
8:              A ← σ_{¬θ[M]}(A)
9:          else if cond is QUERY_ℓ(sql, params) then
10:             Q_ℓ ← SQLTORA(sql, params)
11:               ▷ Converts SQL query to relational algebra
12:             (A, M) ← CONJOINQUERY(Q_ℓ, A, M)
13:     Q_{ℓ+1} ← SQLTORA(cq.sql, cq.params)
14:     (A, M) ← CONJOINQUERY(Q_{ℓ+1}, A, M)
15:     return RATOSQL(A)
16: procedure CONJOINQUERY(Q_ℓ, A, M)
17:     let Q_ℓ = π_{j_1,...,j_m} σ_θ(S_1 × S_2 × ···)    ▷ Normal form
18:     n ← arity(A)                ▷ Number of columns in A
19:     θ' ← θ[k ↦ k + n][M]                ▷ ∀ column index k
20:     A ← π_{1,...,n,n+j_1,...,n+j_m} σ_{θ'}(A × S_1 × S_2 × ···)
21:     M ← M ∪ {r_ℓ.i ↦ i + n | 1 ≤ i ≤ m}
22:     return (A, M)
```

Observe that A indeed returns the same rows as QUERY_1 if the branch condition holds, and an empty result otherwise.

Then, QUERY_2 is conjoined onto A, resulting in view V:

```
SELECT roles.*, grades.* FROM roles, grades
WHERE roles.user_id = MyUserId
  AND roles.course_id = CourseId
  AND roles.is_instructor
  AND grades.course_id = roles.course_id
```

Note that QUERY_2's sole parameter, $r_1.\textit{course_id}$, has been replaced by the `course_id` column exposed by A_2. A final step remains to remove the `CourseId` parameter, which we will discuss at the end of this subsection. ◀

A generated view can be thought of as a natural generalization of the conditioned query to cases where a SQL query can return multiple rows: The view allows a query to be issued for every combination of rows returned by the prior queries, as long as the branching conditions are met. This would allow the program to have loops over result sets in the form we assume in the simple query-issuing core (§ 4.1).

We specify the view-generation procedure in Algorithm 2, which uses relational algebra notation (under the unnamed perspective) [1, § 3.2] for conciseness. The algorithm works for PSJ queries in normal form [1, Prop. 4.4.2]. It does not currently handle general joins or aggregations; when these arise, we approximate them using supported constructs (§ 7.3).

Algorithm 2 essentially follows the steps illustrated by Example 5.1. It maintains a mapping $\mathcal{M}$ from result columns of prior queries to columns in A (e.g., mapping $r_1.\textit{course_id}$

to the third column of A). It then uses this mapping to resolve references to results from prior queries.

Removing request parameters Recall from Example 2.1 that request parameters like `CourseId` must not appear in view definitions. So strictly speaking, Example 5.1 has produced not one view, but a set of views:[7]

$$\{V[\texttt{CourseId} \mapsto x] : x \in \text{dom}(\texttt{roles.course_id})\}$$

To collapse this set into one view, we use the following fact.

Fact 5.2 (Informal)**.** *Let* $V[\texttt{X}]$ *be a view definition of the form:*

```
SELECT col_1, col_2, ... FROM tbl_1, tbl_2, ...
WHERE col_j = X AND f
```

where $\texttt{col}_j$ *is a non-nullable column,* X *is a request parameter, and* f *does not refer to* X*. Then the set of views* $\{V[\texttt{X} \mapsto x] : x \in \text{dom}(\texttt{col}_j)\}$ *reveals the same information as the view:*

```
SELECT col_j, col_1, col_2, ...
FROM tbl_1, tbl_2, ... WHERE f
```

While this fact applies only to queries of a specific form, it already covers all cases encountered in our evaluation. We leave a theoretical study of the general case to future work.

Example 5.1 (continuing from p. 8)**.** Starting from view V, Ote removes the condition `roles.course_id = CourseId`, but refrains from adding `roles.course_id` to the `SELECT` statement because it is redundant with the already-present `grades.course_id`. This brings us to the final view $V^\star$:

```
SELECT * FROM roles, grades
WHERE roles.user_id = MyUserId
  AND roles.is_instructor
  AND grades.course_id = roles.course_id
```

The view $V^\star$ differs from the handwritten view V_2 only by retaining extra columns. However, conditioned on the existence of V_1 (which Ote would have generated from another conditioned query), $V^\star$ reveals the same information as V_2 and so is equally tight. For simplicity, Ote outputs $V^\star$ without removing extraneous columns, since we find $V^\star$ just as concise and readable as V_2. ◀

Outputting SQL Ote outputs SQL views in the form:

```
SELECT ... FROM tbl1, tbl2, ... WHERE ...
```

similar to $V^\star$ above. It uses standard query-optimization passes to make the query shorter and easier to read.

5.4 Pruning Views via Enforcement

Lastly, Ote minimizes the set of views for each handler—producing a subset that reveals the same information—and then takes their union and minimizes it again. To minimize a set $\mathbf{V}$ of views, Ote goes through each view $V \in \mathbf{V}$ and checks whether the information revealed by V is already contained in that revealed by $\mathbf{V} \setminus \{V\}$; if so, it removes V. Heuristically, Ote sorts the views by the number of joins in decreasing order, so that longer views have a chance of being removed first.

It remains to check information containment. This is the same problem as policy enforcement: checking whether issuing V *as a query* is allowed under the policy $\mathbf{V} \setminus \{V\}$. To tackle this, we repurpose an existing enforcement tool, Blockaid [107], which we extended with a command-line interface to be invoked by Ote. Finally, Ote outputs the minimized set of views for human review.

[7] We let $V[\texttt{X} \mapsto x]$ denote the view definition V with each occurrence of X replaced by x, and $\text{dom}(\texttt{col})$ denote the set of valid values for column `col`.

6 Implementation and Practical Aspects

Driver and Policy Generator Ote's concolic-execution driver and policy generator are implemented in Scala 3. The driver uses Apache Calcite [13] to convert SQL queries into relational algebra, and invokes Z3 using its Java binding [28]. It communicates with executors using Protobuf messages, and saves program inputs and transcripts to compressed JSON files. The policy generator uses Calcite's query analyses and optimizations for conditioned-query simplification (§ 5.2), and uses Scala's parallel collections [74] for parallelization.

For the relevance judge (§ 4.6), Ote invokes parallel instances of the Codex CLI [70] coding agent using the non-interactive `exec` subcommand. Ote parses the agent's textual output for the verdict and logs the details for human review, treating any "unsure" verdict or malformed output as relevant.

Executors Ote's library uses RSpec [80] to invoke handlers ("controller actions") with inputs from the driver. At startup, the executor clears the database. For each input, it begins a database transaction, populates the database, installs symbolic request parameters by patching the `params` hash, and sets two symbolic session parameters: the user ID and the current time. It then invokes the handler and rolls back the transaction.

We disable Rails's fragment and low-level caching to expose queries issued only on cache misses. As an optimization, we also disable Rails's query cache, which introduces branches that do not affect what data is fetched. We execute handlers on MySQL backed by an in-memory `tmpfs`, and configure string columns to use a case-sensitive collation [108] as our SMT encoding (§ 4.4) does not support case insensitivity.

Generating database constraints To help the user write down an application's database constraints (§ 3.1), Ote provides a tool that generates common types of constraints for a Rails application, by inspecting the database schema's SQL constraints and the Active Record models' validators, associations, and inheritance hierarchy. The user may then supplement the list with constraints not covered by the tool (§ 7.1),

Table 1: Number of database constraints (§ 7.1.3).

	Application Logic		For Ote	
	Generated	Manual	Manual	**Total**
diaspora	347	63	2	412
Autolab	185	42	4	231
Odin	116	9	1	126

a task we hope to automate using LLMs in the future.

Tracing a view back to the application To help users understand why a view was generated, Ote records with each view the ID of an execution from which the view is derived. Using this ID, the user can recover the corresponding input and re-run the execution—optionally under a debugger—to inspect the program state when a query was issued.

7 Evaluation

We applied Ote to three existing applications:

1. diaspora [30], a social network with over 850 k users;
2. Autolab [22], a platform for managing course assignments used at over 20 schools; and
3. The Odin Project (Odin) [92], a site where over a million users take web development classes and share their work.

We chose open-source Ruby on Rails applications with nontrivial access-control logic, while staying within the scope of the Ote prototype. Within this space, diaspora and Autolab were natural choices because we had written policies for them by hand before [107, § 8.1], allowing us to compare those policies with the extracted ones. Odin provides a complementary case: We had never worked with it before, so our experience applying Ote to Odin was not biased by prior knowledge.

The key takeaways from this evaluation are:

- Ote can extract a policy within a few hours (§ 7.4).
- The extracted policies avoided several errors present in the handwritten policies and revealed an access-check bug in the application code (§ 7.6).
- Applying Ote required limited manual setup and review effort (which we summarize in § 7.8).

7.1 Application Setup

All applications run on MySQL in our experiments. Their database schemas are nontrivial: diaspora has 50 tables and 387 total columns, Autolab has 26 tables and 269 total columns, and Odin has 17 tables and 152 total columns.

7.1.1 Code changes

We used versions of diaspora (v0.7.14) and Autolab (v2.7.0) that had previously been modified to work with Blockaid. To summarize the most relevant modifications:

- We had modified a few code locations to issue parameterized queries. (Most queries were already parameterized, but some needed a rewrite—see below for an example.)
- We had modified the applications to fetch sensitive data only if it affects output. (These modifications allow for finer-grained policies but are *not required by Ote*; we kept them for an apples-to-apples policy comparison.)
- We had rewritten one query to an equivalent form supported by Ote. (This modification is *not fundamentally required*—it could have been implemented in Ote itself.)

In addition, we modified seven lines in Autolab to save query results in variables for reuse. This simple refactor helped the relevance judge identify more irrelevant conditionals by clarifying that accessing the variables does not trigger a query.

For Odin, we made three changes (over commit `f6762f0`):

- We modified two locations to issue parameterized SQL queries: e.g., from `where('expires >= ?', Time.now)` to `where(expires: Time.now..)`, because the former inserts `Time.now` into the query string within Ruby, resulting in non-parameterized SQL.
- We deleted one line to prevent an aggregation from appearing in the *conditions* portion of conditioned queries (currently unsupported by Ote); this change preserved application behavior.

7.1.2 Relevance hints

We added seven `RELEVANCE-HINT` comments (§ 4.6): one to diaspora and six to Autolab. (Odin had few enough branches that the relevance-judge optimization was not needed—see Table 2.) Two of the hints clarify how an external library affects queries; three note that an already-loaded expression does not trigger a query; one notes that an expression *may* issue a query; and one encodes the assumption that an Autolab scoring function issues no queries.

All of these hints, except the last, compensate for limitations in the LLM-based coding agent's reasoning. We expect these hints to become unnecessary with improvements in the agent's capabilities, or in how it is used: e.g., providing the agent with library documentation, details on Rails's ORM, and runtime information about where queries originate.

7.1.3 Database constraints

Table 1 summarizes the database constraints supplied to Ote. Over 80 % were auto-generated (§ 6). The rest were added manually: Some were transcribed from application logic, while those in the "For Ote" column were introduced to scope the extracted policy or reduce exploration time.

Of the manually written application-logic constraints, many captured application-level invariants—e.g., two diaspora posts cannot be reshares of each other—and could not be auto-generated. For fields with constraints either unsupported by

Table 2: Statistics and performance. "✂" marks runs that used the relevance judge (§ 4.6). Under Statistics, "#Cond. Queries" shows the number of conditioned queries before and after simplification (§ 5.2); "#SQL Views" shows the number of views after per- and cross-handler pruning (§ 5.4). Under Running Time, "Simplify" stands for conditioned-query simplification and view generation (§§ 5.2 and 5.3), "Prune" for per-handler view-pruning, "Final Prune" for cross-handler view-pruning (§ 5.4), and "Total" for the total sequential running time.

	Statistics				Running Time				
Handler	#Paths	#Cond. Queries	#SQL Views		Explore	Simplify	Prune	Final Prune	**Total**
diaspora								36 min	**4.7 h**
`People#stream` ✂	163 832	189 412 → 188	148	→ 134	1.1 h	8 min	35 min		
`Posts#show` ✂	73 489	47 256 → 208	87		32 min	1 min	20 min		
`People#show` ✂	27 480	1318 → 35	27		15 min	33 s	4 min		
`Notifications#index` ✂	32 907	1018 → 75	54		16 min	38 s	9 min		
`Conversations#index`	57 534	93 368 → 134	39		14 min	4 min	12 min		
`Comments#index`	3607	41 → 23	17		2 min	24 s	3 min		
Autolab								29 min	**3.9 h**
`Assessments#show` ✂	217 543	391 621 → 1592	138	→ 144	1.9 h	28 min	32 min		
`Assessments#gradesheet`	15 681	9020 → 46	35		6 min	1 min	6 min		
`Assessments#index`	7839	9993 → 86	27		4 min	50 s	7 min		
`Submissions#download`	79	53 → 21	18		1 min	21 s	3 min		
`Courses#index`	36	17 → 8	5		2 min	21 s	50 s		
`Metrics#getNumPending`	11	14 → 8	7		57 s	20 s	59 s		
The Odin Project								8 min	**59 min**
`Lessons#show`	48 670	49 459 → 152	46	→ 24	13 min	3 min	13 min		
`Courses#show`	4462	3133 → 37	25		3 min	30 s	4 min		
`ProjectSubmissions#index`	3052	2615 → 39	17		3 min	31 s	3 min		
`Users#show`	2003	2059 → 14	9		2 min	29 s	1 min		
`Paths#index`	28	26 → 6	4		2 min	19 s	38 s		
`Sitemap#index`	28	13 → 4	3		1 min	20 s	27 s		

our SMT encoding (e.g., URL validity) or related to the external environment (e.g., an Autolab course name must correspond to a directory on disk), we constrained them to a fixed set of valid values after creating any required external state.

We wrote seven additional constraints specifically for policy extraction. To save exploration time, we required nullable string fields to be non-empty (as the two cases are typically treated identically), limited the Autolab analysis to non-admin users (as admin policies are trivial to write), and fixed the diaspora user's language to English. In Autolab, we disabled scheduler actions (which are not governed by user-specific policies) and disallowed zero score-penalties because our prototype does not track float operations (this preserves application features as a `0.0` penalty is equivalent to no penalty).

7.2 Experiment Setup

We ran experiments on Google Compute Engine using a `c3-standard-176` instance. The driver launched 48 parallel executors and used Z3 v4.11.2 for SMT solving. It invoked Codex CLI (v0.58.0) using the `gpt-5` model and medium reasoning effort, with a maximum parallelism of 16, a timeout of 5 min, and at most three retries after timeout.For view pruning, it invoked Blockaid with a timeout of 5 s. Executors ran a modified version of JRuby v9.3.13.0 atop OpenJDK 21.

We ran path exploration on each handler three times; Table 2 shows the data from the runs with median exploration time. To save on LLM calls, we enabled relevance-based pruning (§ 4.6) only for those handlers whose unpruned exploration exceeded 15 min (marked with ✂ in Table 2).

7.3 Paths, Conditioned Queries, and Views

Table 2 shows that while the number of explored paths can be large, Ote is able to reduce the number of views to between 24 and 144—a manageable number for human review.

For view generation, we had to approximate some SQL features unsupported by either our Ote prototype or Blockaid:

- We rewrite a `LEFT JOIN` into an equivalent `INNER JOIN` if possible. Otherwise, we split it into an `INNER JOIN` (for rows that match) and a `SELECT` (for rows that don't).
- We rewrite a `SELECT COUNT(*) FROM tbl` in the query portion into `SELECT id FROM tbl`, and omit aggregations in the condition portion.[8]
- We omit date-timestamp comparisons and replace "`Now + 1 second`" with "`Now`".

[8]The latter did not end up affecting the final policies because every aggregation present in the condition was redundant.

Table 3: Relevance-judge verdicts and running times. "Rel." denotes relevant; "Irrel." denotes irrelevant (§ 7.5).

Handler	Counts		Duration (min)
	Rel.	Irrel.	(mean ± std)
diaspora			
`People#stream`	31	242	1.3 ± 0.4
`Posts#show`	13	171	1.4 ± 0.4
`People#show`	16	86	1.6 ± 0.4
`Notifications#index`	21	82	1.9 ± 0.7
Autolab			
`Assessments#show`	46	54	1.0 ± 0.3

These approximations are "lossy": They can broaden or tighten the policy and must be applied with human discretion.

For Autolab, we applied a simple broadening immediately after view generation (Figure 1). The `Assessments#show` handler queries the `score_adjustments` table under many complex conditions, producing a large number of views. Since the table contains only non-sensitive metadata, we added a single broad view: `SELECT * FROM score_adjustments`, which caused view pruning to eliminate all narrower views involving the table. Table 2 reports results with this broadening applied. We give a more involved example of broadening in § 7.7.

7.4 Performance

Table 2 (right) shows the running times for each phase of policy extraction. Overall, end-to-end extraction completes within five hours at most. When we ran path exploration on the handlers marked with ✂ *without* the relevance-judge optimization (§ 4.6), all timed out after ten hours (not shown in the table) and, by our estimate, would have required *many days* to finish. This demonstrates that relevance-based pruning is essential for exploring real web-application code at scale.

7.5 Calls to the Relevance Judge

Table 3 reports relevance-judge verdicts (it never returned "unsure") and call-duration statistics. We manually reviewed every irrelevant verdict to confirm its accuracy. Although there were over 600 irrelevant verdicts, they arose from a limited set of code locations (e.g., in diaspora, a profile-picture helper invoked along many paths), making grouped auditing more straightforward. We note that the call durations include LLM reasoning time and serving latency, both outside our control.

7.6 Findings From the Extracted Policies

We now compare the extracted policies with the handwritten diaspora and Autolab policies from our prior work [107, § A]. For Odin, where no handwritten policy was available, we manually inspected the extracted policy and found it accurate for the handlers analyzed.

Table 4: View count in extracted vs handwritten policies. Extracted policies contain more views because they often preserve conditions that handwritten policies relax (§ 7.6).

	diaspora	Autolab	Odin
Extracted policy	134	144	24
Handwritten policy	66	37	–

Expectations We expect extracted policies to be tighter (more restrictive) than handwritten ones: Ote aims to produce the tightest possible policy, whereas humans often relax non-privacy-critical conditions and allow accesses beyond what the application requires (§ 7.7). Thus, extracted policies are generally longer because they encode conditions omitted from handwritten policies (Table 4). In our evaluation, this extra detail remained manageable for human review.

Where handwritten policies reveal too much But not all relaxations by the human policy-writer are benign. When we compared the extracted Autolab policy against the handwritten one, we found that the latter granted course assistants access to five types of records in a "disabled" course, when the application's logic states that only *instructors* should have access. Such an erroneous policy, if enforced, would allow a future code change to leak sensitive data to course assistants.

Where handwritten policies reveal too little Unexpectedly, there is also information revealed by the extracted policy but not by the handwritten one. To investigate, we used Blockaid to check whether each extracted view is allowed as a query under the handwritten policy. We found that:

- The handwritten diaspora policy failed to reveal the pod of a "remote" person and data for `MentionedInPost` and `MentionedInComment` notifications.
- The handwritten Autolab policy overlooked granting instructors access to all attachments in their courses.

Enforcing a policy with such oversights would disrupt application functionality by denying legitimate data accesses.

A defective access check While reviewing the extracted Autolab policy, we noticed that none of the submissions-related views checked the `assessments.exam` column. This was suspicious because Autolab is supposed to prohibit students from downloading prior exam submissions.

It turned out that we had introduced a bug years earlier when adapting Autolab for access control. When configuring the `lazy_column` gem [55] to defer loading sensitive fields, we mistakenly listed the column using its query method name

`exam?` instead of its actual name `exam`. This caused the gem to override `exam?` to always return `nil` (which is falsy), silently disabling the exam-check logic. So a correct-looking access check was rendered a no-op due to a misuse of an external library elsewhere in the codebase—a subtle bug that we discovered only after reviewing the policy extracted by Ote.

7.7 Broadening the Extracted Policy

Sometimes, an extracted policy includes many combinations of conditions under which data can be accessed, reflecting business logic rather than privacy concerns. For example, the extracted diaspora policy has 39 views (out of 134) that reveal a profile under various conditions: if it belongs to the current user, or to the author of a public post, etc. But based on our understanding, a diaspora user's profile is intended to be public, except for a few columns guarded by the `public_details` flag. This means we can simplify the policy by *broadening* it.

We can broaden the policy with the help of Ote's view pruning (§ 5.4). First, we write down four simpler, broader views to capture the relaxed conditions for accessing profiles:

1. ```
   SELECT id, first_name, ... FROM profiles
   ```
   (For all profiles, some columns are always visible...)
2. ```
   SELECT * FROM tags, taggings
   WHERE tags.id = taggings.tag_id
     AND taggings.taggable_type = 'Profile';
   ```
   (...and so are the profile taggings.)
3. ```
   SELECT * FROM profiles
   WHERE public_details = TRUE
   ```
   (All columns are visible if the profile's "public details" flag is set...)
4. ```
   SELECT * FROM profiles, people
   WHERE profiles.person_id = people.id
     AND people.owner_id = MyUserId
   ```
   (...or if the profile belongs to the current user.)

We add these to the policy and re-run view-pruning, which removes 36 of the 39 profile-related views as redundant. This process spares the human from having to reason about view subsumption, which can be tedious and tricky. (For example, the remaining three views, which pertain to the current user's contacts, are *not* subsumed by the ones that we added!)

7.8 Manual Effort and Expertise Required

The steps involved in applying Ote required varying degrees of SQL, Rails, and application knowledge. Before extraction:

- The *code changes* (§ 7.1.1) were mostly compatibility edits that required general Rails and SQL familiarity. The only application-specific changes—fetching sensitive data only when it affects output—were inherited from our prior work and are not required by Ote.
- Writing *relevance hints* (§ 7.1.2) required Rails knowledge to determine whether particular expressions or library calls could issue queries.
- Writing manual *database constraints* (§ 7.1.3) required familiarity with the application's data model, including its schema, Rails model associations and validators, and invariants encoded only in application logic.

After extraction:

- *Query approximations* (§ 7.3) required SQL knowledge to assess their effect on the policy, and application familiarity to decide whether they were appropriate.
- *Policy review* (§ 7.6) required application and SQL knowledge to explain policy differences as policy mistakes, application bugs, or benign discrepancies.
- *Policy broadening* (§ 7.7) required application familiarity to identify non-privacy-relevant distinctions, and SQL knowledge to write the broader views.

8 Related Work

Symbolic execution Symbolic execution [45] is a classic path-exploration technique that maintains symbolic program state. We decided not to use it because implementing it for an interpreted language is challenging (more so than for lower-level languages [21, 61, 87]) due to the dynamic features and functionality implemented outside the language [18, § 2.2].

Bug finding Many tools use symbolic or concolic execution to find bugs in web applications [9, 23, 99], typically aiming to trigger certain statements or states. Policy extraction requires not only reaching the query-issuing statements, but also gathering the conditions under which they are reached. Other systems infer security-relevant behavior from code to discover violations. AutoISES [91] infers expected security checks around OS operations and reports omissions; iHunter [54] finds privacy violations in iOS SDKs by recovering privacy-sensitive data flows; and Derailer [66] and Space [67] check for security bugs in web applications by validating data exposures against human input or a catalog. Such tools use inferred behavior as evidence of bugs, whereas Ote produces an access-control policy for review and enforcement.

Syscall filtering Abhaya [71] synthesizes Seccomp-bpf and Pledge policies by statically analyzing a program's syscall behavior and finding a tight policy expressible in the target sandbox language. SysPart [75], C2C [33], and Sysfilter [29] similarly harden binaries by inferring the set of syscalls a program may invoke and restricting the allowed syscalls to this set. These systems are conceptually similar to Ote in that they infer policies from programs. However, they typically rely on static analysis (sometimes complemented by dynamic observations) and reason about a smaller policy space: syscall identities and, in some systems, predicates over syscall arguments. In contrast, Ote targets a dynamic scripting language and extracts expressive relational database policies, including the conditions under which SQL queries are issued.

Instrumentation strategies Some systems track symbolic values in dynamic languages by performing instrumentation *within* the target language. They represent symbolic objects through proxying [17, 98] or inheritance [12], and track path conditions using Boolean-conversion hooks [12, 17] or debug tracing [98, § 4.1.2]. These approaches avoid the need for a custom interpreter and support standard environments [8]. In contrast, Ote performs *offline* analysis, allowing us to modify the interpreter for better transparency [26] and performance.

Learning models of applications Konure infers models of database-backed applications [84, 85] by generating targeted inputs to probe the application as a black-box. This work informed our formulation of the simple query-issuing core assumption (§ 4.1). However, black-box probing cannot effectively recover conditions of query issuance [85, § 6.2], whereas Ote can extract them directly via instrumentation.

Policy mining Policy-mining systems share our goal of generating access-control rules that capture existing practices. Most take existing access-control lists [19, 102, 103], operation logs [32, 40, 60], or human interactions [39], and produce role- [32, 60, 102], attribute- [40, 103, 104], or relationship-based [19, 39] rules, often via statistical techniques. Unlike these systems, Ote does not require live data and instead analyzes the application, which provides the visibility needed to produce fine-grained policies. AutoArmor [52] is closer to Ote in that it analyzes application code to generate access-control policies, but it targets inter-service calls in microservice applications whereas Ote extracts database access-control policies from web applications.

9 Discussion and Future Work

Ote enables better data security in existing web applications: by extracting the access-control policy embedded in application code, it makes that policy explicit, allowing an admin to understand and ultimately enforce the intended policy.

Although Ote is semi-automated and provides no formal guarantees, these traits are shared by most policy-assistance tools. RBAC and ABAC policy-mining tools (§ 8) routinely depend on human-curated inputs [31], require humans in the loop [59, § 7.2], and offer no guarantees as they operate on partial [25, 44, 60] or noisy [7, 64, 101] data using statistical methods. Yet these tools have consistently proven effective in helping admins develop practical policies [41, 59, 68, 94, 96]. These broader successes, together with our own, give us confidence that Ote can provide similar—if not greater—value, particularly because Ote generates far finer-grained policies than typical RBAC or ABAC rules.

We conclude with a few avenues for future work:

Automation bias and complacency A policy extractor may create a false sense of security when reviewers accept a plausible-looking policy without sufficient scrutiny. This risk is an instance of *automation bias and complacency*: Reviewers may over-rely on computer output and abdicate their decision-making responsibility [73]. Future work should investigate these effects in the context of policy extraction and adapt established mitigations to this setting [34].

Policy comprehension SQL is precise and familiar, but SQL views can become verbose when they encode many application-level conditions. A natural next step is a concise policy domain-specific language (DSL) that preserves SQL-view semantics while making policies easier to read, compare, and audit. Such a DSL should draw on query-language usability studies [5, 76] and ORM abstractions [93]; verified lifting techniques [43, 48] could translate generated SQL views into this DSL while preserving their meaning.

Coding agents The rapid improvement of coding agents raises a natural question: Could an agent simply read source code and extract a policy directly? In principle, yes—human experts can do this, albeit slowly and imperfectly, and a capable enough agent might automate it. But the goal is not merely to produce a plausible policy; it is to produce one whose provenance a human can audit and understand.

Ote has limitations: it applies only under stated assumptions and relies on certain heuristics. But these limitations are explicit. Ote decomposes policy extraction into well-defined subproblems grounded in concrete traces, derives policies through explicit transformations, and produces intermediate artifacts that help reviewers trace a policy fragment to its source.[9] An end-to-end agent may also make mistakes, but those mistakes can be harder to localize because they arise from an opaque reasoning process with unclear failure modes.

Rather than replacing Ote's structure, agents could operate within it: modeling complex application logic, flagging suspicious views, suggesting policy broadenings, etc. This way, agents are used for well-scoped subtasks whose outputs are attached to explicit artifacts, so that increased automation does not come at the cost of auditability.

Acknowledgments

We thank the anonymous reviewers and members of the UC Berkeley NetSys Lab and Sky Computing Lab for their feedback. This work is supported in part by NSF grant 2145471 and by gifts from Accenture, Algorithmic SuperIntelligence Labs, Amazon, AMD, Anyscale, Broadcom, cmpnd, Google, IBM, Intel, Intesa Sanpaolo, Lambda, Lightspeed, NVIDIA, Samsung SDS, and SAP.

[9] Even when Ote uses an LLM-based relevance judge, it records the judge's verdicts and explanations for audit.

References

[1] Serge Abiteboul, Richard Hull, and Victor Vianu. *Foundations of Databases*. Addison-Wesley, 1995.

[2] Justus Adam, Carolyn Zech, Livia Zhu, Sreshtaa Rajesh, Nathan Harbison, Mithi Jethwa, Will Crichton, Shriram Krishnamurthi, and Malte Schwarzkopf. Paralegal: Practical static analysis for privacy bugs. In Lidong Zhou and Yuanyuan Zhou, editors, *19th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2025, Boston, MA, USA, July 7-9, 2025*, pages 957–978. USENIX Association, 2025.

[3] Foto Afrati, Rada Chirkova, and H. V. Jagadish. *Answering Queries Using Views*. Morgan & Claypool Publishers, 2nd edition, 2019.

[4] Rakesh Agrawal, Paul Bird, Tyrone Grandison, Jerry Kiernan, Scott Logan, and Walid Rjaibi. Extending relational database systems to automatically enforce privacy policies. In Karl Aberer, Michael J. Franklin, and Shojiro Nishio, editors, *Proceedings of the 21st International Conference on Data Engineering, ICDE 2005, 5-8 April 2005, Tokyo, Japan*, pages 1013–1022. IEEE Computer Society, 2005.

[5] Alireza Ahadi, Julia Coleman Prior, Vahid Behbood, and Raymond Lister. A quantitative study of the relative difficulty for novices of writing seven different types of SQL queries. In *Proceedings of the 2015 ACM Conference on Innovation and Technology in Computer Science Education, ITiCS 2015*. ACM, 2015.

[6] Kinan Dak Albab, Artem Agvanian, Allen Aby, Corinn Tiffany, Alexander Portland, Sarah Ridley, and Malte Schwarzkopf. Sesame: Practical end-to-end privacy compliance with policy containers and privacy regions. In Emmett Witchel, Christopher J. Rossbach, Andrea C. Arpaci-Dusseau, and Kimberly Keeton, editors, *Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles, SOSP 2024, Austin, TX, USA, November 4-6, 2024*, pages 709–725. ACM, 2024.

[7] Manar Alohaly, Hassan Takabi, and Eduardo Blanco. A deep learning approach for extracting attributes of ABAC policies. In Elisa Bertino, Dan Lin, and Jorge Lobo, editors, *Proceedings of the 23rd ACM Symposium on Access Control Models and Technologies, SACMAT 2018, Indianapolis, IN, USA, June 13-15, 2018*, pages 137–148. ACM, 2018.

[8] Kalev Alpernas, Aurojit Panda, Leonid Ryzhyk, and Mooly Sagiv. Cloud-scale runtime verification of serverless applications. In Carlo Curino, Georgia Koutrika, and Ravi Netravali, editors, *SoCC '21: ACM Symposium on Cloud Computing, Seattle, WA, USA, November 1 - 4, 2021*, pages 92–107. ACM, 2021.

[9] Shay Artzi, Adam Kiezun, Julian Dolby, Frank Tip, Danny Dig, Amit M. Paradkar, and Michael D. Ernst. Finding bugs in dynamic web applications. In Barbara G. Ryder and Andreas Zeller, editors, *Proceedings of the ACM/SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2008, Seattle, WA, USA, July 20-24, 2008*, pages 261–272. ACM, 2008.

[10] Warwick Ashford. Facebook photo leak flaw raises security concerns, March 2015. https://www.computerweekly.com/news/2240242708/Facebook-photo-leak-flaw-raises-security-concerns.

[11] Thomas H. Austin, Jean Yang, Cormac Flanagan, and Armando Solar-Lezama. Faceted execution of policy-agnostic programs. In Prasad Naldurg and Nikhil Swamy, editors, *Proceedings of the 2013 ACM SIGPLAN Workshop on Programming Languages and Analysis for Security, PLAS 2013, Seattle, WA, USA, June 20, 2013*, pages 15–26. ACM, 2013.

[12] Thomas Ball and Jakub Daniel. Deconstructing dynamic symbolic execution. In Maximilian Irlbeck, Doron A. Peled, and Alexander Pretschner, editors, *Dependable Software Systems Engineering*, volume 40 of *NATO Science for Peace and Security Series, D: Information and Communication Security*, pages 26–41. IOS Press, 2015.

[13] Edmon Begoli, Jesús Camacho-Rodríguez, Julian Hyde, Michael J. Mior, and Daniel Lemire. Apache Calcite: A foundational framework for optimized query processing over heterogeneous data sources. In Gautam Das, Christopher M. Jermaine, and Philip A. Bernstein, editors, *Proceedings of the 2018 International Conference on Management of Data, SIGMOD Conference 2018, Houston, TX, USA, June 10-15, 2018*, pages 221–230. ACM, 2018.

[14] Gabriel Bender, Lucja Kot, and Johannes Gehrke. Explainable security for relational databases. In Curtis E. Dyreson, Feifei Li, and M. Tamer Özsu, editors, *International Conference on Management of Data, SIGMOD 2014, Snowbird, UT, USA, June 22-27, 2014*, pages 1411–1422. ACM, 2014.

[15] Gabriel Bender, Lucja Kot, Johannes Gehrke, and Christoph Koch. Fine-grained disclosure control for app ecosystems. In Kenneth A. Ross, Divesh Srivastava, and Dimitris Papadias, editors, *Proceedings of*

the ACM SIGMOD International Conference on Management of Data, SIGMOD 2013, New York, NY, USA, June 22-27, 2013, pages 869–880. ACM, 2013.

[16] Murphy Berzish, Vijay Ganesh, and Yunhui Zheng. Z3str3: A string solver with theory-aware heuristics. In Daryl Stewart and Georg Weissenbacher, editors, *2017 Formal Methods in Computer Aided Design, FMCAD 2017, Vienna, Austria, October 2-6, 2017*, pages 55–59. IEEE, 2017.

[17] Alessandro Bruni, Tim Disney, and Cormac Flanagan. A peer architecture for lightweight symbolic execution, February 2011. Retrieved April 4, 2024 from https://hoheinzollern.wordpress.com/wp-content/uploads/2008/04/seer1.pdf.

[18] Stefan Bucur, Johannes Kinder, and George Candea. Prototyping symbolic execution engines for interpreted languages. In Rajeev Balasubramonian, Al Davis, and Sarita V. Adve, editors, *Architectural Support for Programming Languages and Operating Systems, ASPLOS 2014, Salt Lake City, UT, USA, March 1-5, 2014*, pages 239–254. ACM, 2014.

[19] Thang Bui and Scott D. Stoller. A decision tree learning approach for mining relationship-based access control policies. In Jorge Lobo, Scott D. Stoller, and Peng Liu, editors, *Proceedings of the 25th ACM Symposium on Access Control Models and Technologies, SACMAT 2020, Barcelona, Spain, June 10-12, 2020*, pages 167–178. ACM, 2020.

[20] Thang Bui, Scott D. Stoller, and Hieu Le. Efficient and extensible policy mining for relationship-based access control. In Florian Kerschbaum, Atefeh Mashatan, Jianwei Niu, and Adam J. Lee, editors, *Proceedings of the 24th ACM Symposium on Access Control Models and Technologies, SACMAT 2019, Toronto, ON, Canada, June 03-06, 2019*, pages 161–172. ACM, 2019.

[21] Cristian Cadar, Daniel Dunbar, and Dawson R. Engler. KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs. In Richard Draves and Robbert van Renesse, editors, *8th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2008, December 8-10, 2008, San Diego, California, USA, Proceedings*, pages 209–224. USENIX Association, 2008.

[22] Autolab project. https://autolabproject.com/.

[23] Avik Chaudhuri and Jeffrey S. Foster. Symbolic security analysis of Ruby-on-Rails web applications. In Ehab Al-Shaer, Angelos D. Keromytis, and Vitaly Shmatikov, editors, *Proceedings of the 17th ACM Conference on Computer and Communications Security, CCS 2010, Chicago, Illinois, USA, October 4-8, 2010*, pages 585–594. ACM, 2010.

[24] Adam Chlipala. Static checking of dynamically-varying security policies in database-backed applications. In Remzi H. Arpaci-Dusseau and Brad Chen, editors, *9th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2010, October 4-6, 2010, Vancouver, BC, Canada, Proceedings*, pages 105–118. USENIX Association, 2010.

[25] Carlos Cotrini, Thilo Weghorn, and David A. Basin. Mining ABAC rules from sparse logs. In *2018 IEEE European Symposium on Security and Privacy, EuroS&P 2018, London, United Kingdom, April 24-26, 2018*, pages 31–46. IEEE, 2018.

[26] Limitations – crosshair 0.0.54 documentation. Retrieved April 16, 2024 from https://crosshair.readthedocs.io/en/latest/limitations.html.

[27] Saptarshi Das, Barsha Mitra, Vijayalakshmi Atluri, Jaideep Vaidya, and Shamik Sural. Policy engineering in RBAC and ABAC. In Pierangela Samarati, Indrajit Ray, and Indrakshi Ray, editors, *From Database to Cyber Security - Essays Dedicated to Sushil Jajodia on the Occasion of His 70th Birthday*, volume 11170 of *Lecture Notes in Computer Science*, pages 24–54. Springer, 2018.

[28] Leonardo de Moura. Z3 for Java, 2012. Retrieved October 6, 2024 from https://leodemoura.github.io/blog/2012/12/10/z3-for-java.html.

[29] Nicholas DeMarinis, Kent Williams-King, Di Jin, Rodrigo Fonseca, and Vasileios P. Kemerlis. sysfilter: Automated system call filtering for commodity software. In Manuel Egele and Leyla Bilge, editors, *23rd International Symposium on Research in Attacks, Intrusions and Defenses, RAID 2020, San Sebastian, Spain, October 14-15, 2020*, pages 459–474. USENIX Association, 2020.

[30] The diaspora* project. https://diasporafoundation.org/.

[31] Mario Frank, Joachim M. Buhmann, and David A. Basin. Role mining with probabilistic models. *ACM Trans. Inf. Syst. Secur.*, 15(4):15:1–15:28, 2013.

[32] Nurit Gal-Oz, Yaron Gonen, Ran Yahalom, Ehud Gudes, Boris Rozenberg, and Erez Shmueli. Mining roles from web application usage patterns. In Steven Furnell, Costas Lambrinoudakis, and Günther Pernul, editors, *Trust, Privacy and Security in Digital Business - 8th International Conference, TrustBus 2011,*

Toulouse, France, August 29 - September 2, 2011. Proceedings, volume 6863 of *Lecture Notes in Computer Science*, pages 125–137. Springer, 2011.

[33] Seyedhamed Ghavamnia, Tapti Palit, and Michalis Polychronakis. C2C: fine-grained configuration-driven system call filtering. In Heng Yin, Angelos Stavrou, Cas Cremers, and Elaine Shi, editors, *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security, CCS 2022, Los Angeles, CA, USA, November 7-11, 2022*, pages 1243–1257. ACM, 2022.

[34] Kate Goddard, Abdul Roudsari, and Jeremy C Wyatt. Automation bias: a systematic review of frequency, effect mediators, and mitigators. *Journal of the American Medical Informatics Association*, 19(1):121–127, 2012.

[35] Patrice Godefroid, Nils Klarlund, and Koushik Sen. DART: directed automated random testing. In Vivek Sarkar and Mary W. Hall, editors, *Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation, Chicago, IL, USA, June 12-15, 2005*, pages 213–223. ACM, 2005.

[36] Matthew Green. Twitter post: Piazza offers anonymous posting, but does not hide each user's total number of posts, October 2017. https://twitter.com/matthew_d_green/status/925053953330634753.

[37] Yang He, Pinhan Zhao, Xinyu Wang, and Yuepeng Wang. VeriEQL: Bounded equivalence verification for complex SQL queries with integrity constraints. *Proc. ACM Program. Lang.*, 8(OOPSLA1):1071–1099, 2024.

[38] Tomasz Imielinski and Witold Lipski Jr. Incomplete information in relational databases. *J. ACM*, 31(4):761–791, 1984.

[39] Padmavathi Iyer and Amir Masoumzadeh. Towards automated learning of access control policies enforced by web applications. In Silvio Ranise, Roberto Carbone, and Daniel Takabi, editors, *Proceedings of the 28th ACM Symposium on Access Control Models and Technologies, SACMAT 2023, Trento, Italy, June 7-9, 2023*, pages 163–168. ACM, 2023.

[40] Padmavathi Iyer and Amirreza Masoumzadeh. Mining positive and negative attribute-based access control policy rules. In Elisa Bertino, Dan Lin, and Jorge Lobo, editors, *Proceedings of the 23rd ACM Symposium on Access Control Models and Technologies, SACMAT 2018, Indianapolis, IN, USA, June 13-15, 2018*, pages 161–172. ACM, 2018.

[41] Sakuna Harinda Jayasundara, Nalin Asanka Gamagedara Arachchilage, and Giovanni Russello. Sok: Access control policy generation from high-level natural language requirements. *ACM Comput. Surv.*, 57(4):102:1–102:37, 2025.

[42] JRuby – the Ruby programming language on the JVM. https://www.jruby.org.

[43] Shoaib Kamil, Alvin Cheung, Shachar Itzhaky, and Armando Solar-Lezama. Verified lifting of stencil computations. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2016*. ACM, 2016.

[44] Leila Karimi, Maryam Aldairi, James Joshi, and Mai Abdelhakim. An automatic attribute-based access control policy extraction from access logs. *IEEE Trans. Dependable Secur. Comput.*, 19(4):2304–2317, 2022.

[45] James C. King. Symbolic execution and program testing. *Commun. ACM*, 19(7):385–394, 1976.

[46] Eddie Kohler. Hide review rounds from paper authors • kohler/hotcrp@5d53abc, March 2013. https://github.com/kohler/hotcrp/commit/5d53ab.

[47] Eddie Kohler. Download PC review assignments obeys paper administrators • kohler/hotcrp@80ff966, March 2015. https://github.com/kohler/hotcrp/commit/80ff96.

[48] Shadaj Laddad, Conor Power, Mae Milano, Alvin Cheung, and Joseph M. Hellerstein. Katara: synthesizing crdts with verified lifting. *Proc. ACM Program. Lang.*, 6(OOPSLA2), 2022.

[49] Kristen LeFevre, Rakesh Agrawal, Vuk Ercegovac, Raghu Ramakrishnan, Yirong Xu, and David J. DeWitt. Limiting disclosure in Hippocratic Databases. In Mario A. Nascimento, M. Tamer Özsu, Donald Kossmann, Renée J. Miller, José A. Blakeley, and K. Bernhard Schiefer, editors, *(e)Proceedings of the Thirtieth International Conference on Very Large Data Bases, VLDB 2004, Toronto, Canada, August 31 - September 3 2004*, pages 108–119. Morgan Kaufmann, 2004.

[50] Alon Y. Levy, Alberto O. Mendelzon, and Yehoshua Sagiv. Answering queries using views (extended abstract). In *Proceedings of the Fourteenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, PODS '95, page 95–104, New York, NY, USA, 1995. Association for Computing Machinery.

[51] Haonan Li, Yu Hao, Yizhuo Zhai, and Zhiyun Qian. Enhancing static analysis for practical bug detection: An LLM-integrated approach. *Proc. ACM Program. Lang.*, 8(OOPSLA1):474–499, 2024.

[52] Xing Li, Yan Chen, Zhiqiang Lin, Xiao Wang, and Jim Hao Chen. Automatic policy generation for inter-service access control of microservices. In Michael D. Bailey and Rachel Greenstadt, editors, *30th USENIX Security Symposium, USENIX Security 2021, August 11-13, 2021*, pages 3971–3988. USENIX Association, 2021.

[53] Tianyi Liang, Andrew Reynolds, Cesare Tinelli, Clark W. Barrett, and Morgan Deters. A DPLL(T) theory solver for a theory of strings and regular expressions. In Armin Biere and Roderick Bloem, editors, *Computer Aided Verification - 26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18-22, 2014. Proceedings*, volume 8559 of *Lecture Notes in Computer Science*, pages 646–662. Springer, 2014.

[54] Dexin Liu, Yue Xiao, Chaoqi Zhang, Kaitao Xie, Xiaolong Bai, Shikun Zhang, and Luyi Xing. ihunter: Hunting privacy violations at scale in the software supply chain on ios. In Davide Balzarotti and Wenyuan Xu, editors, *33rd USENIX Security Symposium, USENIX Security 2024, Philadelphia, PA, USA, August 14-16, 2024*. USENIX Association, 2024.

[55] Jorge Manrubia. jorgemanrubia/lazy_columns: Rails plugin that adds support for lazy-loading columns in Active Record models, 2015. https://github.com/jorgemanrubia/lazy_columns.

[56] Alana Marzoev, Lara Timbó Araújo, Malte Schwarzkopf, Samyukta Yagati, Eddie Kohler, Robert Tappan Morris, M. Frans Kaashoek, and Sam Madden. Towards multiverse databases. In *Proceedings of the Workshop on Hot Topics in Operating Systems, HotOS 2019, Bertinoro, Italy, May 13-15, 2019*, pages 88–95. ACM, 2019.

[57] Mark Maunder. Vulnerability in WordPress Core: Bypass any password protected post. CVSS score: 7.5 (High), June 2016. https://www.wordfence.com/blog/2016/06/wordpress-core-vulnerability-bypass-password-protected-posts/.

[58] Aastha Mehta, Eslam Elnikety, Katura Harvey, Deepak Garg, and Peter Druschel. Qapla: Policy compliance for database-backed systems. In Engin Kirda and Thomas Ristenpart, editors, *26th USENIX Security Symposium, USENIX Security 2017, Vancouver, BC, Canada, August 16-18, 2017*, pages 1463–1479. USENIX Association, 2017.

[59] Barsha Mitra, Shamik Sural, Jaideep Vaidya, and Vijayalakshmi Atluri. A survey of role mining. *ACM Comput. Surv.*, 48(4):50:1–50:37, 2016.

[60] Ian M. Molloy, Youngja Park, and Suresh Chari. Generative models for access control policies: applications to role mining over logs with attribution. In Vijay Atluri, Jaideep Vaidya, Axel Kern, and Murat Kantarcioglu, editors, *17th ACM Symposium on Access Control Models and Technologies, SACMAT '12, Newark, NJ, USA - June 20 - 22, 2012*, pages 45–56. ACM, 2012.

[61] Mark Mossberg, Felipe Manzano, Eric Hennenfent, Alex Groce, Gustavo Grieco, Josselin Feist, Trent Brunson, and Artem Dinaburg. Manticore: A user-friendly symbolic execution framework for binaries and smart contracts. In *34th IEEE/ACM International Conference on Automated Software Engineering, ASE 2019, San Diego, CA, USA, November 11-15, 2019*, pages 1186–1189. IEEE, 2019.

[62] Amihai Motro. An access authorization model for relational databases based on algebraic manipulation of view definitions. In *Proceedings of the Fifth International Conference on Data Engineering, February 6-10, 1989, Los Angeles, California, USA*, pages 339–347. IEEE Computer Society, 1989.

[63] Amy Munson, Juanita Gomez, and Alvaro A. Cárdenas. With a little help from my (LLM) friends: Enhancing static analysis with LLMs to detect software vulnerabilities. In *IEEE/ACM International Workshop on Large Language Models for Code, LLM4Code@ICSE 2025, Ottawa, ON, Canada, May 3, 2025*, pages 25–32. IEEE, 2025.

[64] Masoud Narouei and Hassan Takabi. Towards an automatic top-down role engineering approach using natural language processing techniques. In Edgar R. Weippl, Florian Kerschbaum, and Adam J. Lee, editors, *Proceedings of the 20th ACM Symposium on Access Control Models and Technologies, Vienna, Austria, June 1-3, 2015*, pages 157–160. ACM, 2015.

[65] Alan Nash, Luc Segoufin, and Victor Vianu. Views and queries: Determinacy and rewriting. *ACM Trans. Database Syst.*, 35(3):21:1–21:41, 2010.

[66] Joseph P. Near and Daniel Jackson. Derailer: interactive security analysis for web applications. In Ivica Crnkovic, Marsha Chechik, and Paul Grünbacher, editors, *ACM/IEEE International Conference on Automated Software Engineering, ASE '14, Vasteras, Sweden - September 15 - 19, 2014*, pages 587–598. ACM, 2014.

[67] Joseph P. Near and Daniel Jackson. Finding security bugs in web applications using a catalog of access control patterns. In Laura K. Dillon, Willem Visser, and Laurie A. Williams, editors, *Proceedings of the*

38th International Conference on Software Engineering, ICSE 2016, Austin, TX, USA, May 14-22, 2016, pages 947–958. ACM, 2016.

[68] Mohammad Nur Nobi, Maanak Gupta, Lopamudra Praharaj, Mahmoud Abdelsalam, Ram Krishnan, and Ravi S. Sandhu. Machine learning in access control: A taxonomy and survey. *CoRR*, abs/2207.01739, 2022.

[69] Alan C. O'Connor and Ross J. Loomis. 2010 Economic Analysis of Role-Based Access Control. Technical report, National Institute of Standards and Technology, December 2010.

[70] OpenAI. Codex CLI, 2025. Retrieved December 7, 2025 from https://developers.openai.com/codex/cli/.

[71] Shankara Pailoor, Xinyu Wang, Hovav Shacham, and Isil Dillig. Automated policy synthesis for system call sandboxing. *Proc. ACM Program. Lang.*, 4(OOPSLA):135:1–135:26, 2020.

[72] Paolo Perrotta. *Metaprogramming Ruby 2*. Pragmatic Bookshelf, 2014.

[73] Amanda Potasznik. ABCs: Differentiating algorithmic bias, automation bias, and automation complacency. In *2023 IEEE International Symposium on Ethics in Engineering, Science, and Technology (ETHICS)*, pages 1–5, 2023.

[74] Aleksandar Prokopec and Heather Miller. Overview | Parallel Collections | Scala Documentation. Retrieved April 16, 2024 from https://docs.scala-lang.org/overviews/parallel-collections/overview.html.

[75] Vidya Lakshmi Rajagopalan, Konstantinos Kleftogiorgos, Enes Göktas, Jun Xu, and Georgios Portokalidis. SysPart: Automated temporal system call filtering for binaries. In Weizhi Meng, Christian Damsgaard Jensen, Cas Cremers, and Engin Kirda, editors, *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security, CCS 2023, Copenhagen, Denmark, November 26-30, 2023*, pages 1979–1993. ACM, 2023.

[76] Phyllis Reisner. Human factors studies of database query languages: A survey and assessment. *ACM Comput. Surv.*, 13(1), 1981.

[77] H. G. Rice. Classes of recursively enumerable sets and their decision problems. *Transactions of the American Mathematical Society*, 74(2):358–366, 1953.

[78] Shariq Rizvi, Alberto O. Mendelzon, S. Sudarshan, and Prasan Roy. Extending query rewriting techniques for fine-grained access control. In Gerhard Weikum, Arnd Christian König, and Stefan Deßloch, editors, *Proceedings of the ACM SIGMOD International Conference on Management of Data, Paris, France, June 13-18, 2004*, pages 551–562. ACM, 2004.

[79] Arnon Rosenthal and Edward Sciore. View security as the basis for data warehouse security. In Manfred A. Jeusfeld, Hua Shu, Martin Staudt, and Gottfried Vossen, editors, *Proceedings of the Second Intl. Workshop on Design and Management of Data Warehouses, DMDW 2000, Stockholm, Sweden, June 5-6, 2000*, volume 28 of *CEUR Workshop Proceedings*, page 8. CEUR-WS.org, 2000.

[80] RSpec: Behaviour driven development for Ruby. Retrieved April 16, 2024 from https://rspec.info/.

[81] Neha Rungta. A billion SMT queries a day (invited paper). In Sharon Shoham and Yakir Vizel, editors, *Computer Aided Verification - 34th International Conference, CAV 2022, Haifa, Israel, August 7-10, 2022, Proceedings, Part I*, volume 13371 of *Lecture Notes in Computer Science*, pages 3–18. Springer, 2022.

[82] Daniel Schwartz-Narbonne, Martin Schäf, Dejan Jovanovic, Philipp Rümmer, and Thomas Wies. Conflict-directed graph coverage. In Klaus Havelund, Gerard J. Holzmann, and Rajeev Joshi, editors, *NASA Formal Methods - 7th International Symposium, NFM 2015, Pasadena, CA, USA, April 27-29, 2015, Proceedings*, volume 9058 of *Lecture Notes in Computer Science*, pages 327–342. Springer, 2015.

[83] Koushik Sen, Darko Marinov, and Gul Agha. CUTE: a concolic unit testing engine for C. In Michel Wermelinger and Harald C. Gall, editors, *Proceedings of the 10th European Software Engineering Conference held jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2005, Lisbon, Portugal, September 5-9, 2005*, pages 263–272. ACM, 2005.

[84] Jiasi Shen and Martin C. Rinard. Using active learning to synthesize models of applications that access databases. In Kathryn S. McKinley and Kathleen Fisher, editors, *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2019, Phoenix, AZ, USA, June 22-26, 2019*, pages 269–285. ACM, 2019.

[85] Jiasi Shen and Martin C. Rinard. Active learning for inference and regeneration of applications that access databases. *ACM Trans. Program. Lang. Syst.*, 42(4):18:1–18:119, 2021.

[86] Jie Shi, Hong Zhu, Ge Fu, and Tao Jiang. On the soundness property for SQL queries of fine-grained access control in DBMSs. In *ICIS*, pages 469–474. IEEE Computer Society, 2009.

[87] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Andrew Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Krügel, and Giovanni Vigna. SOK: (state of) the art of war: Offensive techniques in binary analysis. In *IEEE Symposium on Security and Privacy, SP 2016, San Jose, CA, USA, May 22-26, 2016*, pages 138–157. IEEE Computer Society, 2016.

[88] P.J.M. Smit. A study on static taint analysis techniques in Ruby. Thesis, University of Twente, Enschede, June 2025.

[89] Ben Stock. Search leaks hidden tags • Issue #135 • kohler/hotcrp, June 2018. https://github.com/kohler/hotcrp/issues/135.

[90] Bogdan Alexandru Stoica, Utsav Sethi, Yiming Su, Cyrus Zhou, Shan Lu, Jonathan Mace, Madanlal Musuvathi, and Suman Nath. If at first you don't succeed, try, try, again...? Insights and LLM-informed tooling for detecting retry bugs in software systems. In Emmett Witchel, Christopher J. Rossbach, Andrea C. Arpaci-Dusseau, and Kimberly Keeton, editors, *Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles, SOSP 2024, Austin, TX, USA, November 4-6, 2024*, pages 63–78. ACM, 2024.

[91] Lin Tan, Xiaolan Zhang, Xiao Ma, Weiwei Xiong, and Yuanyuan Zhou. Autoises: Automatically inferring security specification and detecting violations. In Paul C. van Oorschot, editor, *Proceedings of the 17th USENIX Security Symposium, July 28-August 1, 2008, San Jose, CA, USA*, pages 379–394. USENIX Association, 2008.

[92] The Odin Project. https://www.theodinproject.com/.

[93] Alexandre Torres, Renata Galante, Marcelo Soares Pimenta, and Alexandre Jonatan B. Martins. Twenty years of object-relational mapping: A survey on patterns, solutions, and their implications on application design. *Inf. Softw. Technol.*, 82, 2017.

[94] Ivan Tsarynny. Hell yeah, i want an automated content security policy!, June 2022. Retrieved December 2, 2025 from https://www.feroot.com/blog/want-automated-content-security-policy/.

[95] Haoxin Tu, Seongmin Lee, Yuxian Li, Peng Chen, Lingxiao Jiang, and Marcel Böhme. Large language model-driven concolic execution for highly structured test input generation. *CoRR*, abs/2504.17542, 2025.

[96] Tufin. Securing network segments and optimizing permissive rules with the automatic policy generator, 2015. Retrieved December 2, 2025 from https://lp.tufin.com/rs/769-ICF-145/images/APG_Whitepaper.pdf.

[97] Qihua Wang, Ting Yu, Ninghui Li, Jorge Lobo, Elisa Bertino, Keith Irwin, and Ji-Won Byun. On the correctness criteria of fine-grained access control in relational databases. In *VLDB*, pages 555–566. ACM, 2007.

[98] Ruowen Wang, Peng Ning, Tao Xie, and Quan Chen. MetaSymploit: Day-one defense against script-based attacks with security-enhanced symbolic analysis. In Samuel T. King, editor, *Proceedings of the 22nd USENIX Security Symposium, Washington, DC, USA, August 14-16, 2013*, pages 65–80. USENIX Association, 2013.

[99] Gary Wassermann, Dachuan Yu, Ajay Chander, Dinakar Dhurjati, Hiroshi Inamura, and Zhendong Su. Dynamic test input generation for web applications. In Barbara G. Ryder and Andreas Zeller, editors, *Proceedings of the ACM/SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2008, Seattle, WA, USA, July 20-24, 2008*, pages 249–260. ACM, 2008.

[100] Mark D. Weiser. Program slicing. *IEEE Trans. Software Eng.*, 10(4):352–357, 1984.

[101] Yutang Xia, Shengfang Zhai, Qinting Wang, Huiting Hou, Zhonghai Wu, and Qingni Shen. Automated extraction of ABAC policies from natural-language documents in healthcare systems. In Donald A. Adjeroh, Qi Long, Xinghua Mindy Shi, Fei Guo, Xiaohua Hu, Srinivas Aluru, Giri Narasimhan, Jianxin Wang, Mingon Kang, Ananda Mondal, and Jin Liu, editors, *IEEE International Conference on Bioinformatics and Biomedicine, BIBM 2022, Las Vegas, NV, USA, December 6-8, 2022*, pages 1289–1296. IEEE, 2022.

[102] Zhongyuan Xu and Scott D. Stoller. Algorithms for mining meaningful roles. In Vijay Atluri, Jaideep Vaidya, Axel Kern, and Murat Kantarcioglu, editors, *17th ACM Symposium on Access Control Models and Technologies, SACMAT '12, Newark, NJ, USA - June 20 - 22, 2012*, pages 57–66. ACM, 2012.

[103] Zhongyuan Xu and Scott D. Stoller. Mining attribute-based access control policies. *CoRR*, abs/1306.2401, 2013.

[104] Zhongyuan Xu and Scott D. Stoller. Mining attribute-based access control policies from logs. In Vijay Atluri and Günther Pernul, editors, *Data and Applications Security and Privacy XXVIII - 28th Annual IFIP WG*

11.3 Working Conference, DBSec 2014, Vienna, Austria, July 14-16, 2014. Proceedings, volume 8566 of *Lecture Notes in Computer Science*, pages 276–291. Springer, 2014.

[105] Jean Yang, Travis Hance, Thomas H. Austin, Armando Solar-Lezama, Cormac Flanagan, and Stephen Chong. Precise, dynamic information flow for database-backed applications. In Chandra Krintz and Emery D. Berger, editors, *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2016, Santa Barbara, CA, USA, June 13-17, 2016*, pages 631–647. ACM, 2016.

[106] Jean Yang, Kuat Yessenov, and Armando Solar-Lezama. A language for automatically enforcing privacy policies. In John Field and Michael Hicks, editors, *Proceedings of the 39th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2012, Philadelphia, Pennsylvania, USA, January 22-28, 2012*, pages 85–96. ACM, 2012.

[107] Wen Zhang, Eric Sheng, Michael Alan Chang, Aurojit Panda, Mooly Sagiv, and Scott Shenker. Blockaid: Data access policy enforcement for web applications. In Marcos K. Aguilera and Hakim Weatherspoon, editors, *16th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2022, Carlsbad, CA, USA, July 11-13, 2022*, pages 701–718. USENIX Association, 2022.

[108] Xing Zhang. MySQL 8.0.1: Accent and case sensitive collations for utf8mb4, 2017. Retrieved Nov 7, 2024 from https://dev.mysql.com/blog-archive/mysql-8-0-1-accent-and-case-sensitive-collations-for-utf8mb4/.

[109] Zheng Zhang and Alberto O. Mendelzon. Authorization views and conditional query containment. In *ICDT*, volume 3363 of *Lecture Notes in Computer Science*, pages 259–273. Springer, 2005.

Table 5: Artifact hosting locations.

Content	Location	Branch / release
Artifact README	https://github.com/ote-project/artifact-eval	main branch
Ote implementation		
Core implementation (§§ 4 and 5)	https://github.com/ote-project/concolic_driver	main branch
Modified JRuby (§ 4.5)	https://github.com/ote-project/jruby	9.3.13.0-ote branch
Modified Blockaid (§ 5.4)	https://github.com/ote-project/blockaid	standalone-checking branch
Applications (§ 7.1)		
diaspora	https://github.com/ote-project/diaspora	ote-v0.7.14.0 branch
Autolab	https://github.com/ote-project/Autolab	ote-v2.7.0 branch
The Odin Project	https://github.com/ote-project/theodinproject	ote-ruby2.6 branch
Experiments		
Virtual-machine image	https://github.com/ote-project/artifact-eval	ae release tag (assets)
Experiment scripts	https://github.com/ote-project/ae-scripts	main branch
Application config for Ote (§ 3.1)	https://github.com/ote-project/ae-app-config	main branch
Handwritten policy baseline (§ 7.6)	https://github.com/ote-project/app-policies	ote branch

A Artifact Appendix

Abstract

This artifact provides the Ote implementation, including the concolic-execution driver and supporting modifications to JRuby and Blockaid, along with the modified applications and setup needed to reproduce the evaluation.

Scope

The artifact reproduces Tables 2 to 4 in a PDF and writes the extracted policies to separate output files.

Contents

The artifact includes the source repositories for Ote's concolic-execution driver (§ 4.3), the JRuby interpreter modified for symbolic tracking (§ 4.5), Blockaid modified to support view pruning (§ 5.4), and the three modified applications used in the evaluation (§ 7.1). It also provides a prebuilt virtual-machine image with the scripts and configuration files needed to reproduce the evaluation.

Ote uses an LLM-based relevance judge implemented with OpenAI's Codex CLI (§ 7.2). To save resources, the artifact includes a "mock judge" that, instead of calling an LLM, returns a verdict after an configured delay. By default, the artifact's experiment script uses the mock judge with a 90-second per-call delay, approximating the real judge's latency.

Hosting

The artifact is hosted under the ote-project GitHub organization (https://github.com/ote-project). Table 5 lists the hosting location for each component.

Requirements

The artifact was tested on Google Cloud using a Compute Engine c3-standard-176 instance. Importing the VM image requires access to a Google Cloud Storage bucket for uploading the image before import. Running Ote with the real Codex-based relevance judge requires an OpenAI account.

iLand: An Instruction-Level Dynamic Binary Instrumentation framework for iOS

Kaitao Xie, Yizhuo Wang, and Xiaolong Bai*
Alibaba Group
{easylyou.xkt,xrosheart926,bxl1989}@gmail.com

Abstract

Dynamic binary instrumentation (DBI) enables monitoring and modification of executing programs and forms the foundation for a range of program analysis and security testing. On iOS, however, no DBI is available for non-jailbroken devices. Existing approaches on other platforms (e.g., Android) rely on JIT compilation, which is prohibited by the iOS sandbox. The limited CPU and memory resources of mobile devices further constrain the practical deployment of DBI.

We propose *iLand*, a novel instruction-level DBI framework for iOS. Instead of JIT compilation, it translates instructions into predefined micro-operations and interprets using precompiled atomic execution units. To reduce CPU and memory overhead, it employs application-only emulation: only the app's code is interpreted, while system libraries run natively. We implement *iLand* as a standard sandboxed iOS app capable of emulating other apps. It preserves the emulated apps' original functionality and user experience such as dynamic UI rendering, real-time interaction, live video streaming.

Based on this DBI framework, we further implemented an instruction-level dynamic tracing tool and used it to study policy-violation behaviors on 60 top-ranked App Store apps. We found that 13 (21%) apps are still invoking private APIs, of which 2 invoke APIs explicitly prohibited by Apple. Our analysis further revealed the new and stealthy methods employed by apps to evade Apple's App Review. In particular, in 15 (25%) of the apps, we observed a new way to collect sensitive information by direct invocation of the SVC instruction.

1 Introduction

Dynamic binary analysis is widely used in software testing and security research [10, 25, 31, 50, 52, 59]. On iOS, one of the most popular mobile operating systems, such techniques are particularly valuable because App Store apps are mostly closed-source. However, Apple's sandbox—including mandatory code signing and runtime hardening—imposes significant limitations: the dynamic code generation and loading of unsigned binaries are not allowed.

As a consequence, dynamic analysis on iOS is primarily based on two approaches: code injection on jailbroken devices [15, 45] and app repackaging on non-jailbroken devices [6, 51]. Code injection in jailbroken conditions exploits kernel-level vulnerabilities to bypass the sandbox. However, publicly available jailbreaks support only up to iOS 16, incompatible with the recent iOS versions. App repackaging involves unpacking the app bundle, injecting custom libraries, modifying the main binary's load commands, and then resigning and repackaging the app. The limitation is that it typically instruments apps at the API-level rather than instruction-level. Both methods either rely on a jailbroken environment or compromise the integrity of the app binary, where apps may detect and behave differently from their original functionalities.

The requirements to adhere to the system sandbox while maintaining transparency lead us to explore DBI techniques. DBI techniques enable the insertion of probes and user-supplied analysis routines into running programs. Conventional DBI frameworks, such as Valgrind [46] and Dyninst [8], have demonstrated utility and effectiveness across numerous application domains [1,5,18,19,21,24,28,35,37,44,49,54,56]. However, they are not applicable to iOS currently due to two major challenges. First, the sandbox forbids dynamic code generation: DBI cannot allocate RWX (readable, writable and executable) memory pages. Second, the limited CPU and memory resources of mobile devices constrain practical deployment. For instance, iPhone 15 and iOS 17 were both released in 2023, where system libraries consume 3.3 GB memory and device has only 6 GB RAM. All system libraries are collected into Dyld Shared Cache (DSC), mapped and shared by all app processes. This means a DBI framework has to allocate a large region for guest app's DSC while retaining its own DSC mapping, which can quickly exhaust memory and trigger Jetsam [3] (iOS's out-of-memory killer).

In this paper, we present *iLand*, a dynamic binary instrumentation framework for iOS that monitors and alters the execution of guest apps at the instruction-level. To avoid

*Xiaolong Bai is the corresponding author.

JIT-compiled code, *iLand* adopts a translation and interpretation approach: it translates instructions into predefined micro-operations with equivalent semantics, and interprets these using precompiled atomic execution units. To fit the memory constraints, *iLand* employs *application-only emulation*: it interprets only the app's own code while allowing library code to run natively, avoiding the need to interpret the large DSC while making use of native libraries for CPU-intensive tasks such as video codecs and cryptographic operations.

Application-only emulation introduces a challenge absent from conventional DBI: native library code may return to or call back into interpreted app code (N→I transitions), yet the library code is immutable and the iOS sandbox prohibits generating executable code at runtime, making it fundamentally difficult to recapture control. To address this, we introduce a *stateless control-flow manager* that constructs trampolines via shared memory pages and `pc-relative` instructions, redirecting N→I transitions back to the interpreter without per-call state or RWX memory. Building on this, we employ a layered approach combining static analysis, runtime interception, and signal-based fallback to capture all N→I transitions.

We implement *iLand* as a standard sandboxed app, able to emulate guest apps. To evaluate performance, our tests on the SPEC CPU 2017 benchmark reveal that while *iLand* incurs higher overhead compared to JIT-based DBI, it achieves a 10x speedup over the qemu-tci interpreter. To evaluate real-world compatibility, we run 64 top-ranked apps from the US App Store under *iLand* and manually test their main functionality (e.g., login, browsing, search, video). Of these, 49 (77%) worked correctly, 11 showed partial functionality loss (e.g., app detected anomalies of environment and refused login); and 4 terminated abnormally (e.g., app's watchdog timeouts triggered termination). Overall, 60 apps reached a usable state with working UI interactions.

Built on *iLand*, we implement an instruction-level dynamic tracing tool that records control-flow transfers of interest, including indirect calls and SVC invocations for 60 top-ranked apps in the compatibility test. We found that 13 (21%) apps are still invoking private APIs, which are not allowed according to App Store Guidelines. Like prior works [15, 17], we treat undocumented APIs as private, yet Apple neither publishes an authoritative list of private APIs nor specifies exactly which calls trigger rejection. To validate our finding, we developed a test app invoking such APIs and upload it to TestFlight, Apple's pre-release distribution platform. We found 2 invocation are explicitly prohibit by TestFlight automated review. Our analysis further revealed the new and stealthy methods employed by current apps to evade Apple's App Review, including runtime symbol resolution, direct SVC invocations, `dlopen` of generated binaries and constructing return-oriented programming (ROP) chains. Notably, 15 (25%) apps we observed employ a new way of collecting sensitive information via direct SVC invocation. These findings demonstrate that *iLand* enables effective dynamic analysis of iOS apps.

In summary, this paper makes the following contributions:

- We present *iLand*, an instruction-level DBI framework for iOS, and propose application-only emulation.
- We propose a stateless control-flow manager to handle frequent boundary crossings.
- We implement *iLand* as a standard sandboxed iOS app, apply it to 64 top-ranked iOS apps and manually test main functionality. 60 of them reached a usable state.
- We build tracing tool on top of *iLand* and study the 60 apps. We found 13 (21%) apps invoked private APIs, 15 (25%) apps employed SVC invocation to collect system information, and we revealed four new stealthy methods that current apps used to evade Apple's App Review.

2 Background and Motivation

In this section, we first introduce the iOS sandbox and jailbreaking. We then present the fundamentals of DBI and the challenges in porting it to iOS. Finally, we provide real-world motivating examples discovered by *iLand* that highlight the importance of dynamic analysis in iOS security research.

2.1 iOS Sandbox and Jailbreak

The iOS sandbox, including code signing and runtime hardening, restricts apps' access to system resources and user data, thus strengthening platform security. Code signing is mandatory on iOS: the app's main Mach-O binary is signed with a developer certificate, establishing identity. The signature is verified at runtime by Apple Mobile File Integrity (AMFI for short) [41], a built-in kernel extension. To ensure code integrity, AMFI detects unauthorized code execution (unsigned, modified or injected code), and enforces termination (SIGKILL). Entitlements, embedded in the code signature, declare capabilities; a provisioning profile issued by Apple authorizes a specific set of entitlements for a signed app. However, certain entitlements that grant access outside the app sandbox are restricted to Apple-signed system apps.

Runtime hardening protects runtime integrity by blocking JIT execution and unsigned-library loading. Specifically, apps need the *com.apple.security.cs.allow-jit* entitlement to use RWX (readable, writable, and executable) memory and *com.apple.security.cs.disable-library-validation* to load unsigned libraries. Both entitlements are private to Apple, so third-party apps cannot declare them or generate code at runtime.

While sandbox strengthens iOS security, it constrains dynamic analysis of App Store apps. To conduct runtime analysis, prior works [30, 38, 53, 58] resort to jailbreak technique, which exploits vulnerabilities in older iOS releases to gain arbitrary kernel execution, modify kernel memory, and bypass sandbox restrictions. We argue that jailbreak-based analysis is unsustainable: i) jailbreaks are version-specific and rarely track the latest releases; ii) operating in a sandbox-bypassed

Table 1: Popular publicly available jailbreak tools.

iOS Version	Public Jailbreak Tools
14.8	uncOver [47]
15.4.1	fugu15 [32]
16.6.1	dopamine [23]
16.7.12	palera1n [11]
17.0	
18.0	no available tool
26.0	

environment affects app behavior—apps may detect jailbreaking and behave differently—undermining result validity; and iii) Apple's rapid patching and hardening have made public jailbreaks increasingly scarce.

Table 1 summarizes publicly available jailbreak tools: current advanced jailbreaks target up to iOS 16 (released in 2022), with no working jailbreak for iOS 17 or later. We therefore advocate a jailbreak-independent dynamic analysis framework.

2.2 Porting DBI to iOS

DBI inserts extra code into programs to monitor and alter their runtime state. Pin [43], DynamoRIO [9], Valgrind [46] and Dyninst [8] are widely used DBI frameworks, providing instruction-level instrumentation across architectures and operating systems. Although Dyninst supports Arm64 and Valgrind supports macOS, none currently support non-jailbroken iOS. Figure 1 shows a typical DBI architecture: the `JIT Compiler` decodes app binaries and compiles basic blocks at runtime, rewriting control-flow transfers so the DBI can mediate every change. Translated blocks are placed in `Code Cache` for execution. On control-flow events (e.g., function calls) or analysis-relevant conditions, the `Context Switcher` decides whether to compile or dispatch the next block, invoke analysis routines, or hand control to the OS. This design fundamentally relies on RWX memory for the `Code Cache`, which iOS prohibits, making conventional DBI hard to port.

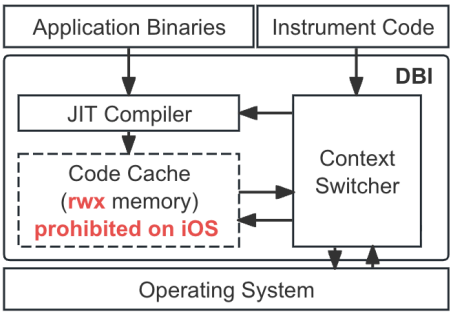


Figure 1: The typical DBI architecture. Code Cache, a key component, requires RWX memory that is prohibited on iOS.

CPU emulation executes programs by decoding and interpreting instructions to faithfully reproduce state transitions. The interpreter-based approaches avoid RWX memory while enabling instruction-level instrumentation. Popular frameworks include QEMU [7] (interpreter mode) and Bochs [40], but they currently do not support iOS.

Table 2: The RAM size and Dyld Shared Cache (DSC) footprint in recent iPhone and iOS version.

iPhone	iOS	Released	RAM	DSC	
				Whole	Code
13	15	2021	4 GB	2.3 GB	1.7 GB
14	16	2022	6 GB	2.7 GB	2.0 GB
15	17	2023	6 GB	3.3 GB	2.4 GB
16	18	2024	8 GB	3.9 GB	2.8 GB
17	26	2025	8 GB	4.6 GB	3.4 GB

However, a full-system interpreter is impractical on iOS due to the limited available memory. Table 2 summarizes usable physical memory and DSC footprint on recent iOS devices. iOS aggregates all system libraries into a single DSC, which is mapped and shared by every app process. On a 6 GB device, the DSC exceeds 3 GB; interpreting both the DSC and app binaries would quickly exhaust memory.

The practical solution is *application-only emulation*: interpret only the app's own code while letting system libraries execute natively. This avoids the memory problem and preserves native performance for CPU-intensive library routines. However, application-only emulation introduces a new challenge that does not arise in conventional DBI:

Bidirectional control-flow transitions without RWX memory. In conventional JIT-based DBI, the framework controls all code execution, so transitions between app and library code are handled implicitly. In application-only emulation, execution alternates between interpreted app code and native library code. Native-to-interpreter (N→I) transitions—where native libraries return to or call back into app code—are fundamentally difficult: native library code is immutable, and the iOS sandbox prohibits generating executable code at runtime. Moreover, N→I transitions arise through diverse mechanisms with distinct calling conventions. Correctly and efficiently recapturing control across all these paths is the central technical challenge that *iLand* addresses.

Accurate and efficient interpretation. The Arm64 instruction set is extensive, with over 60 general mnemonics and more than 380 SIMD mnemonics for float and vector computations, resulting in a large engineering effort. In addition, interpretation overhead directly affects usability; excessive latency triggers UI timeouts and renders apps unusable.

2.3 Motivating Examples of App vetting

Under Apple's App Store Guidelines, apps that use private APIs are not approved for App Store distribution. In our motivating examples, we focus on reject APIs—a subset of private APIs that are reliably flagged by Apple's TestFlight automated review. Such API usage is expected to be rejected during formal App Store review. To ensure typicality, the following cases, identified by *iLand*, are drawn from four top-ranked

free apps among the 60 evaluated, show the stealthy methods to evade App Review and demonstrate *iLand*'s runtime capabilities to reliably detect potentially risky behaviors:

Indirect call. `SecTaskCopyValueForEntitlement` is a reject API on iOS. *iLand* observed that the App_a (top 40) constructs the symbol name at runtime by concatenating `SecTaskCopyVa` and `lueForEntitlement` into a temporary NSString, resolves it via `dlsym`, and then invokes it. This deliberate indirection may reduce the recall of static detectors.

System call. `_dyld_get_shared_cache_range` is a reject API; equivalent functionality can be available via the `SYS_shared_region_check_np` syscall. *iLand* observed that App_b (top 50) reaches hand-written inline assembly at runtime, issuing SVC instruction to invoke this syscall. Such syscall-based invocation patterns are worth vetting, as they may complicate detection of private-API usage.

Indirect jumps via ROP chains. Prior work [55] had shown that return-oriented programming (ROP) can be used to evade App Review by directly manipulating registers to reorder control flow absent from the original binary. *iLand* found that App_c (top 10) searched the DSC for a `SVC RET` gadget—for example, within `mach_msg_trap`, a three-instruction syscall wrapper whose second and third instructions is the desired gadget—and then crafts the stack and the link register to synthesize call sites.

Loading of generated binaries. On iOS, `dlopen` of a valid signed binary is permitted, but generating and loading code at runtime (e.g., a downloaded binary) is considered improper by Apple. *iLand* found that App_d (top 60) issues SVC to invoke `SYS_open` and `SYS_write`, generates a binary to disk, and calls `dlopen`. Our manual analysis indicates the generated binary lacks a valid code signature; `dlopen` would succeed only on sandbox-bypassed devices. We hypothesize this pattern is used to detect jailbreaks. Although the intent may not be evil, such behavior may pose risks to users.

3 Overview

We design *iLand*, an instruction-level dynamic binary instrumentation framework for iOS. Figure 2 depicts the architecture of *iLand*, which comprises four components:

- **The translator** converts Arm64 instructions into the intermediate representation (IR), and inserts analysis code via custom plugins.
- **The loader** loads guest app binaries and the generated IR at appropriate memory locations and protections.
- **The interpreter** interprets the generated IR based on pre-compiled execution units located on the RX memory and monitors guest app control-flow transfers.
- **The Virtual Environment Manager** handles interactions between the guest app and the system, preserving transparency for the guest app.

iLand fundamentally diverges from JIT-based DBI in two respects. First, it replaces JIT compilation with a translation-and-interpretation pipeline: the translator converts app code into an IR of predefined micro-operations, and the interpreter executes these using precompiled execution units residing in RX memory. The original code and IR reside in read-only memory, complying with the iOS sandbox. Translation is performed ahead-of-time (AOT) so that pre-translated code is prepared for repeated use.

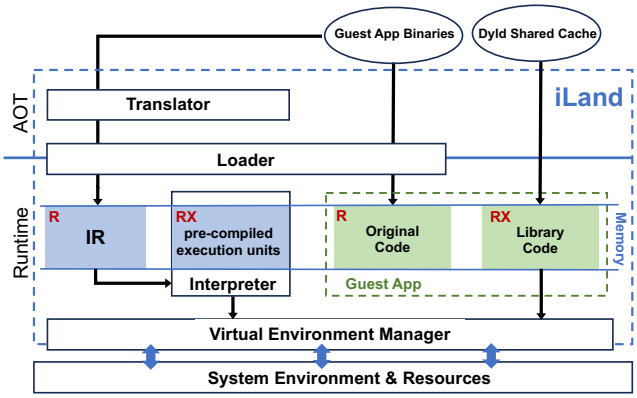


Figure 2: Architecture of *iLand*.

Second, and more significantly, *iLand* adopts application-only emulation, which introduces a control-flow management problem absent from conventional DBI (§ 6.3). Since library code runs natively, the framework must recapture control on every N→I transition. *iLand* introduces a *stateless control-flow manager* (§ 6.2) that constructs trampolines via shared memory pages, requiring no RWX memory and no per-call state. Building on this, a layered strategy captures diverse N→I paths—including C++ virtual methods, Objective-C (ObjC in short) dynamic dispatch, Swift witness tables, ObjC blocks, and stack unwinding.

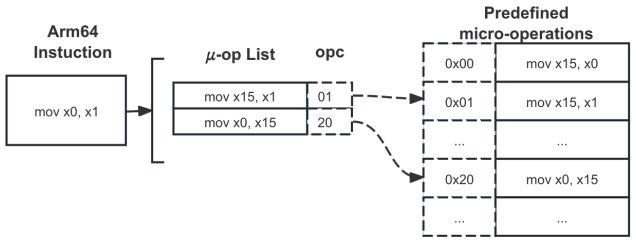
4 Translator

The translator converts native instructions into an IR. We first describe the IR design and the set of reserved registers, and then summarize the translation process.

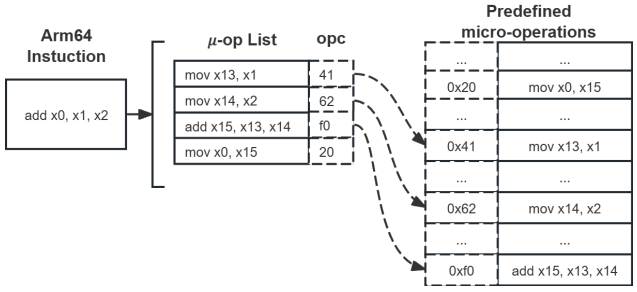
4.1 IR

Each IR in *iLand* IR comprises a 16-bit opcode and optional operands. The opcode indexes a predefined micro-operation that implements semantics equivalent to the corresponding native instruction. These operations, called μ-op, are precompiled execution units in the interpreter's code segment. A native instruction is typically decomposed into three μ-ops using dedicated preserved registers:

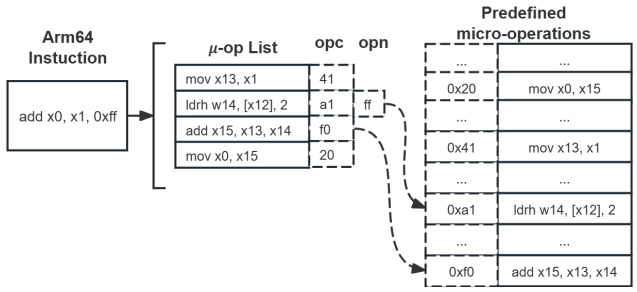
1. μMOV_{src}: Transfers source registers to the fixed preserved source register $X_{\mu src}$.
2. μOP: Performs the original instruction operation and stores the result in the fixed preserved destination register $X_{\mu dst}$, e.g., μADD and μLDR.



(a) Register transfer translation



(b) Register-register addition translation



(c) Register-immediate addition translation

Figure 3: Translation case studies. Registers x12-x15 are preserved by *iLand*. Guest accesses to these registers trigger safe translation to alternative operations.

3. μMOV_{dst} (conditional): Transfers results from $X_{\mu dst}$ to destination registers if required.

This design ensures semantic preservation while maintaining μ-op complexity at $O(n)+O(m)$, where n is the instruction mnemonic count and m is the register quantity. By separating register transfer logic from operations, we avoid the combinatorial explosion of $O(n \cdot m^x)$ (where x represents the maximum register operands per instruction), achieving a practical trade-off between performance and space efficiency. For instructions with immediate operand, we embed the immediate values in IR operand field, and replace the μMOV_{src} with μLDR_{opnd}, which loads the operand value into $X_{\mu src}$. We take `MOV` and `ADD` instructions as examples to demonstrate the relationship between the Arm64 instruction and μ-op.

`MOV` instruction. Figure 3a demonstrates the translation of the register transfer instruction, i.e., `MOV`. *iLand* decomposes general register moves into two μ-ops: first moving the source register to preserved register x15, then transferring from x15 to the destination register. This approach reduces 1,024 potential Arm64 move instructions to 64 reusable units.

`ADD` instruction. Figures 3b and 3c illustrate translations of arithmetic `ADD` instruction. As per Arm64 specifications, register-register and register-immediate additions is distinct handling. The second case employs register transfer plus a unified μADD (`add x15, x13, x14`), and the third case demonstrates immediate loading through operand field encoding (0xff). *iLand* uses variable-length operand encoding to avoid additional μ-op. This strategy enables multiple instruction variants to share common μ-ops, minimizing code footprint while maintaining full architectural coverage.

With this design, *iLand*'s μ-ops form a carefully curated set of Arm64 instruction units (implemented by 6,709 execution units $\leq$128KB), capable of representing all possible combinations of instruction mnemonics and operands.

4.2 Preserved Registers

We adopt a selective register preservation strategy in μ-ops. Only designated subset of registers (x10-x15) is preserved for emulation, while guest apps directly access the remaining registers. Any guest access to preserved registers is transparently redirected to per-thread register shadows in thread-local storage (TLS). This design reduces the conventional register-switching overhead, as context preservation requires saving or restoring only the designated registers, thus efficiently supporting frequent switching during application-only emulation.

The preservation strategy optimizes for performance considerations: i) shadow memory access latency grows with the number of reserved registers, ii) register pressure minimization requires avoiding frequently-used registers in iOS runtime. Through empirical analysis of Arm64 register usage patterns in Mach-O binaries compiled with Apple's toolchain, we observe:

- x0-x7: primary function parameter passing and return
- x8: large structure returns (AAPCS64 compliance)
- x16: syscall number carrier and long-jump usage in `__stub` sections
- x17: `objc_msgSend` dispatch register
- x20: swift method context pointer (`self` pointer)
- x20-x28: caller-saved registers
- x29-x31: frame pointer (FP), link register (LR), and stack pointer (SP)

This observations motivates our selection of x10-x15 as preserved registers, achieving dual objectives: i) Minimizing conflicts with iOS runtime conventions through unused register selection, ii) Enabling efficient SIMD operations (e.g., `LD1 x10-x13, [x15]`) through contiguous register usage The chosen registers demonstrate negligible utilization in sampled iOS applications while maintaining compatibility with ARMv8.4 vector instruction requirements.

4.3 Translation Process

During translation, guest code segments are mapped as read-only. Our static translation engine directly processes instruc-

tions as raw 4-byte quantities:

1. Linearly fetch instructions, leveraging Arm64's fixed-width instruction encoding
2. Parse the instruction type and operands
3. Select corresponding μMOV_{src} (or μLDR_{opnd}), μOP, and μMOV_{dst} accordingly
4. Encode the IR using the μ-ops's indexes in interpreter and optional operands
5. Flow-insensitive optimization preserving original instruction ordering

The translation output is a position-independent data file (*ir* file) containing generated IRs with embedded metadata for runtime interpretation. The loader maps the *ir* file into read-only memory at runtime. It contains 2 major sections as:

$Section_{\mathbf{IR}}$: This section contains the translated IRs. The IRs are linearly arranged as their represented instructions in the original executable file's code segment.

$Table_{\mathbf{IR_offset}}$: During runtime, the interpreter needs to map original instruction addresses to their corresponding IR instances. Because IR length varies (depending on operand presence), IR positions within $Section_{\mathrm{IR}}$ cannot be deterministically computed. *iLand* therefore builds $Table_{\mathrm{IR_offset}}$, an array that records the byte offset of each IR instance. Each 4-byte entry (matching fixed instruction width) corresponds to one native instruction in the original code segment. The benefits of $Table_{\mathrm{IR_offset}}$ include: i) Maintain O(1) lookup complexity through direct offset calculation; ii) Preserve architecture-specific alignment; iii) Enable position-independent IR storage through base-plus-offset addressing.

5 Loader

The loader is the first component to run in the runtime environment, establishing memory layout at guest emulation startup. It performs two major functions: i) loading standard Mach-O files and the generated *ir* files, ii) implementing initialization routines to support ObjC runtime and exception handling.

Mach-O and IR files loading. The loader builds on the open-source dyld [2] from Apple, which loads Mach-O binaries with valid code signatures and identities. While inheriting dyld's core functionality, it introduces three enhancements:

- Specified address memory allocation. The loader implements a `dlopen` to load targets at specific addresses, enabling deterministic memory management.
- Non-executable code segment support. The loader maintains strict read-only permissions for the loaded code.
- Interpreted initialization execution. To complement initialization, the loader forwards the initialization routines (`load` methods of ObjC, `_init_offsets` of C and `__mod_init_func` of C++) from Non-Executable segments to the interpreter in the standard order.

In addition to these three enhancements, to preserve semantics under address space layout randomization (ASLR) while loading *ir* files, the loader performs IR operand patching by iterating through IRs that are sensitive to base addresses.

Initialization routines implementation. The ObjC runtime feature and exception handling both rely on the default dyld's initialization capabilities, including: i) ObjC metadata synchronization, where dyld parses ObjC metadata and integrates them into the `libobjc` data structures; and ii) dyld function pointer fixups, where dyld resolves pointers in libraries that point to dyld implementations for initialization.

To address ObjC metadata synchronization, the loader parses the DSC's local symbol table to locate the private function `notifyObjCMapped` in `libobjc`. By invoking this function, the loader registers ObjC metadata within loader-managed Mach-O files.

Because the DSC is initialized before any *iLand* code runs, the loader needs to locate function pointers initialized by dyld and redirect them to the loader's own implementations. For instance, the loader redirects private function `_dyld_find_unwind_sections` in `libunwind` to its own implementation, which retrieves unwind table information from correct loader-managed Mach-O files, thereby correctly processing exception of guest and directing control flow to appropriate guest's catch blocks.

6 Interpreter

The interpreter executes the initialization routines produced by the loader and then proceeds to `main` in guest's main Mach-O binary. Throughout this process, the interpreter manages all control flow transfers.

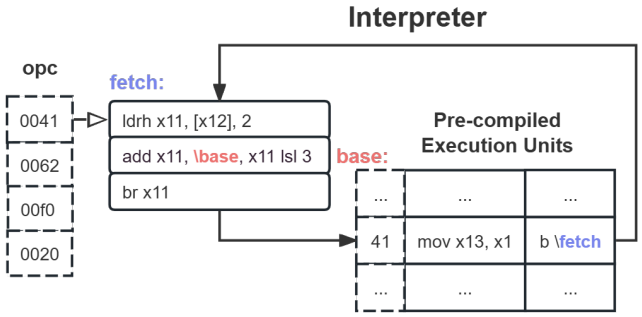


Figure 4: μ-op Interpretation Workflow. Blue markers denote instruction-fetch boundaries, red indicates base address mapping for first-instruction anchoring.

6.1 Interpretation Engine

As mentioned in § 4.1, each opcode indexes a precompiled execution unit, i.e., implementation of μ-op. For efficiency, we set opcode as the offset of the corresponding unit in interpreter's precompiled execution section. The interpreter's core engine employs a straightforward μ-op dispatch-loop: it iteratively fetches the opcode and jumps to the unit indexed by the opcode. In this loop, it uses a preserved register (`X12`)

pointing to the IR (within the mapped *ir* file). To form a tight interpreting loop, each execution unit performs three functions: i) performing the semantics specific to μ-op definition, ii) advancing the `X12` pointer to the next IR if operands are present, and iii) jumping back to the dispatch-loop header.

With this design, the dispatch-loop header requires only three instructions (Figure 4):

- Load opcode pointed at `X12`
- Add opcode with the base of the execution units
- Branch to the unit

The entire dispatch-loop, including the dispatching header and all the execution units, is implemented in hand-written assembly with precise constraints on preserved registers.

6.2 Stateless Control Flow Management

During emulation, target programs inevitably require access to program counter (PC) values through `ADR` instructions or branch instructions (e.g., `B` or `BR`). For direct branches, we resolve the branch target using $Table_{\text{IR_offset}}$ and encode the target IR locations into IR operands. Indirect branches are handled by extracting target register values, followed by IR address lookup via $Table_{\text{IR_offset}}$. In short, direct branch targets are resolved at translation time, whereas indirect branch targets are resolved at runtime.

Although *iLand* manages all control-flow transfers between IR blocks, certain execution mode transitions remain outside its direct control. Transfers occur between interpreting mode (I-state) and native executing mode (N-state). A primary I→N transition occurs when the guest invokes system library functions under application-only emulation. Correspondingly, the reverse N→I transition occurs when native functions return or invoke callbacks passed as function pointer parameters (e.g., `pthread_create()` parameter of `start_routine`). I→N transitions are handled by restoring the preserved registers only; no additional invocation state must be saved.

N →I transitions are more complex because RWX memory is forbidden. To enable stateless transitions, we implement a *trampoline* mechanism. Every guest instructions is associated with a trampoline in $Section_{tpl}$. For a given guest code address ($Addr_{\text{x}}$) and its trampoline ($Tramp_{\text{x}}$), transitioning from N to $Addr_{\text{x}}$ in the guest program is semantically equivalent to transitioning from N to $Tramp_{\text{x}}$ in *iLand*'s interpreter.

$Section_{tpl}$ achieves dynamic adaptation via shared pages, which combines immutable code and writable data while exploiting `pc-relative` instructions. Each trampoline page set consists of three code pages and one data page (Figure 5). Within the `TPL Code Page`, each $Tramp_{\text{x}}$ consists of two instructions:

- `stp x15, lr, [sp, #-0x10]`
- `bl L_stub_page`

$Tramp_{\text{x}}$ preserves two registers at the stack's top, then branches to `TPL Stub Page`. Code in `TPL Stub Page` reconstructs $Addr_{\text{x}}$ from LR together with data in the `TPL Data Page`. It then branches to unified entry point to start interpreter's loop at $Addr_{\text{x}}$. Notably, this process requires no additional state and does not overwrite any registers.

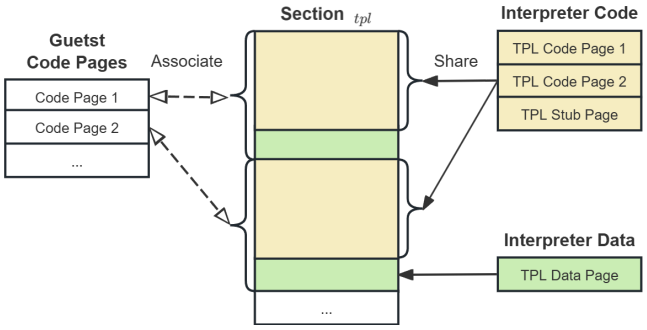


Figure 5: Trampoline memory layout. Memory pages with identical color are shared with same pages.

Each trampoline in $Section_{tpl}$ has a fixed length, enabling efficient address conversion between $Addr_{\text{x}}$ and $Tramp_{\text{x}}$ via bitwise operations rather than expensive table lookups.

We take the system function `pthread_create` as an example to demonstrate how *trampolines* are used in the N→I transition. Before native call to `pthread_create`, the interpreter identifies the native return address ($Addr_{\text{ret}}$) and the `start_routine` parameter ($Addr_{\text{start_routine}}$). It then replaces $Addr_{\text{ret}}$ and $Addr_{\text{start_routine}}$ with their trampoline addresses, i.e., $Tramp_{\text{ret}}$ and $Tramp_{\text{start_routine}}$. The substitution ensures N→I transition as:

- $Tramp_{\text{ret}}$ lets the caller thread resumes IR interpretation at IR_offset_{ret} after `pthread_create` returns
- $Tramp_{\text{start_routine}}$ lets the created thread begins IR execution at $IR_offset_{\text{start_routine}}$

The trampoline mechanism both preserves register state and ensures faithful entry into the interpreter's loop, exhibiting stateless characteristics and complying with diverse calling conventions. Notably, pages in $Section_{tpl}$ are shared and reused, thus i) they consume little additional physical memory despite $Section_{tpl}$'s large virtual size; and ii) they comply with the iOS sandbox because the code is always immutable.

6.3 Application-Only Emulation

Due to limited resource on mobile device, *iLand* focuses on emulating the app's own code, and allows system library functions to execute natively. This strategy requires *iLand* to regain control on N→I. Besides the return address and function-pointer parameters, other N→I scenarios that application-only emulation must handle include:

- **C++ virtual methods** of the guest app's C++ object lead to N→I transitions, e.g., objects' virtual destructors are invoked in `delete`.
- **ObjC message**, the dynamic dispatch to invoke ObjC methods defined in the guest app, leads to N→I transitions. In addition, ObjC runtime operations, including class creation and method swizzling, may also cause

N→I transitions.

- **ObjC block**, similar to function pointers, is the callback mechanism defined in ObjC. When calling system functions, apps pass `Block` parameters to receive callbacks.
- **Swift function pointers**, generated during compilation for reflection, e.g., witness table of Swift type, can behave like the C++ virtual methods invocation.
- **Stack unwinding**, used to implement exception handling (e.g. *try-catch*), transfers control to catch code block defined in the app, leading to N →I transition.

iLand utilizes a layered approach to ensure that N→I transitions are successfully captured and replaced by trampoline.

Initialization optimization driven by static analysis. As mentioned in Section 5, *iLand* employs a loader resolving dependencies and symbols. This specialized loader enables early-stage analysis before the guest app starts, including three optimizations for N→I transitions:

- Address fixups, where code pointers within data segments are identified via ASLR relocation information. Unidentified pointers fall through to the signal-based fallback, affecting only performance, not correctness.
- Symbol resolution, where specialized handling is implemented for `extern` decorated global function pointers in system libraries.
- ObjC pre-initialization, where all ObjC method function pointers are replaced with trampoline wrappers prior to class registration.

Runtime interception. *iLand* intercepts invocations on external C/C++/ObjC/Swift functions. As with our handling of `pthead_create`, during interception *iLand* replaces return addresses, function pointer parameters and ObjC block arguments with their corresponding trampolines. Additionally, return address modification requires special handling for stack unwinding. *iLand* addresses this by instrumenting private function pointers in `libunwind` to recognize trampoline points, enabling correct backtraces.

Signal handling with replicated memory layout. *iLand* preserves the guest app original memory layout, including binary code segments. If *iLand* fails to capture an N→I transition with a trampoline, control transfers into the original read-only code region, causing a `SIGSEGV` or `SIGBUS` generated by the kernel. *iLand* uses POSIX signal handling to intercept these exceptions and regain control.

However, this method incurs significant performance overhead, resulting in operations up to a thousand times slower than native execution. Consequently, this approach is used only as a fallback when previous mechanisms fail.

Lazy optimization during signal handling. *iLand* implements a lazy optimization strategy in its general signal handling. On each signal, *iLand* performs analysis:

- Resolve C++ vtable pointers using the object `this` pointer (the `X0` register) and vtable structure metadata
- Identify ObjC hybrid callback patterns by leveraging the ObjC object layout conventions

This approach incurs a one-time overhead per signal occurrence while maintaining optimal performance for subsequent invocations, and is applied primarily to C++ virtual method and ObjC block callback scenarios.

Interpreting libraries. In edge cases where libraries frequently transfer back to the guest app and *iLand* may fail to capture the transition, *iLand* continues to interpret the libraries to reduce the enormous overhead caused by frequent signal handling. A prominent example is `libswiftcore`, the Swift runtime library. Swift objects carry variable-length metadata structures containing multiple compressed function pointers (e.g., witness tables), whose diversity makes it impractical to statically enumerate all callback targets.

7 Virtual Environment Manager

The manager serves as the primary interface between the *iLand* and iOS, handling interactions between the guest app and the system. Utilizing the control-flow monitor capabilities provided by the interpreter, the manager intercepts and forwards library calls and syscalls, ensuring correct access to app and system resources while remaining transparent to the guest app. Specifically, the manager virtualizes a sandboxed filesystem and provides an in-process isolated memory.

File system virtualization. The manager implements a sandboxed file system that is visible only to the guest app and supports standard file operations. Each app's sandbox comprises a data container for app data storage and a bundle container for the app's code and resources. The guest app can reuse *iLand*'s data container since *iLand* does not store data in the container. As for the bundle container, *iLand* redirects the bundle path of the app to a clean folder. Specifically, *iLand* places the guest app's bundle in a private app group path and hijacks the bundle path by intercepting all file-related library calls and system calls.

In-process isolated memory. Leveraging the loader's capability, *iLand* loads the guest app into a designated memory region. The manager takes over the app's memory access, making this memory appear as the app's isolated memory. Specifically, iOS apps can access memory via three mechanisms: i) library APIs (e.g., `mach_vm_read`); ii) SVC invocations (e.g., Mach syscall); and iii) memory access instructions (e.g., `LDR`). The manager enforces memory isolation for cases (i) and (ii) by intercepting memory-access library calls and syscalls. Because *iLand*'s own memory is coalesced into a continuous region that lies at a lower virtual address than the guest-viewed region, with a randomized gap, guest apps generally do not probe unavailable address via raw load/store instructions (doing so typically triggers `SIGSEGV` termination). Accordingly, it is safe to omit case (iii). Although we have extended execution units of memory-access instructions, this feature is disabled by default for performance reasons.

8 Implementation

We implement *iLand* with more than 300 KLOC of C++/ObjC code, including 6,709 hand-written assembly execution units (each typically 1–3 assembly instructions), cross-compiled into an iOS dynamic framework.

Empirical optimization. Considering *iLand*'s sensitivity to performance, we made deliberate trade-offs in the translator and interpreter implementation. By moderately increasing the IR count, IR operand length, and memory footprint, we achieved relative performance improvements:

- Immediate value optimization. For instructions with immediate operands (e.g., 0-128), we introduced specialized μ-ops that directly assign values to registers instead of fetching them from IR operand. This increases IR count but eliminates one memory read operation.
- High-Frequency register operation acceleration. To reduce register spilling overhead, we add dedicated μ-ops that operate directly on the accessed registers. A notable example is for stack pushing instruction `STP X29,X30,[SP,#-0x10]!` frequently seen at function prologues. Adding a specialized μ-op avoids two register move cycles by directly handling `X29` and `X30`.
- Trampoline precomputation. Discussed in § 6.2, *iLand* replaces return addresses with trampolines during system library invocation. By extending `BL/BLR` instruction operands to encode precomputed trampolines into operands, we optimize the N→I transition. This reduces a multi-memory-access computation into a single memory read, increasing memory usage but reducing latency.

Prototype iOS app. Based on *iLand*, we implement a standard sandboxed iOS app. Although apps' main binaries are uniformly encrypted after distributed on App Store, researchers can get decrypted versions by tools (e.g., `trollde crypt` [57]) or public Internet (e.g., `decrypt.day` [14]). Our prototype processes decrypted packages:

Translator (§ 4) first performs ahead-of-time (AOT) translation on all binaries (including the main binary and embedded frameworks) int the IPA package. This batch translation generates *ir* files which are stored in the private app group path of our prototype app, together with the unpacked IPA contents. After translation, the stored artifacts are used in further interpretation. At startup, the loader (§ 5) coordinates the dynamic loading of *ir* files and the corresponding native binaries. The interpreter (§ 6) executes the μ-ops and manage control-flow transfers. The virtual environment manager (§ 7) transparently redirects sandbox resource access.

9 Evaluation

In this section, we perform a systematic assessment of *iLand*, focusing on four key dimensions: efficiency, capability, compatibility, and usability.

Table 3: Experimental setup. QEMU-jit represents the standard user-space emulator, while QEMU-tci denotes a built with the TCI (tiny code interpreter) backend.

Hardware	OS	Test Suite	Tool
iPhoneX Apple A11 Bionic 2.4GHz,3GB MEM	iOS 16.7.7	SPEC CPU ®2017	native Valgrind *iLand*
Alibaba Cloud ECS Yitian710 [42] 2.7GHz,4GB MEM	Linux 5.10.134 AArch64	SPEC CPU ®2017	native QEMU-jit QEMU-tci
iPhone14 Pro Apple A16 Bionic 3.4GHz,6GB MEM	iOS 17.3.1	AppStore Top Rank iOS Apps	*iLand*

9.1 Experimental Setup

We set up three groups of experiments to evaluate *iLand*'s performance (Groups 1 and 2) and real-world compatibility, usability (Group 3). The hardwares, OS, test suites, and tools used for comparison in each group are detailed in Table 3.

Group 1 setup. Group 1 is designed to assess the performance of *iLand* against other tools on the iPhone. Among existing tools, only Valgrind supports Apple OS. However, it natively supports only macOS, and its extended version for iOS([15] in 2015) lacks 64-bit support. Therefore, we ported the latest Valgrind (v3.24.0) to iOS, enabling DBI for command-line programs. As Valgrind relies on JIT-compiled code, we conducted the comparative experiments on a jailbroken device, specifically using an iPhone X running iOS 16.7.7, which supports Palera1n [11](v2.0) jailbreak. As for the test suite, we selected the SPEC CPU benchmark, which, although lacking an official iOS port, supports macOS Arm64 architectures. We performed cross-compilation of its v1.1.7 release on an Apple M1 Pro device to support iOS. Out of the original 19 benchmarks evaluating integer and floating-point performance, we excluded 11: 8 due to incompatibility with the iOS compilation toolchain (written in Fortran), and 3 failing to execute natively on the iOS platform.

Group 2 setup. Since *iLand* is designed to operate on iPhones where RWX memory is restricted, we use Group 2 to demonstrate its interpretation efficiency. Given the lack of comparable tools on iOS, we selected `qemu-tci`, which operates based on interpretation. Considering the platform differences, we ran both on their respective platforms and compared their slowdown factors relative to their native executions. We used the same SPEC CPU benchmark as in Group 1, and measured the performance slowdown of `qemu-tci` running on Elastic Cloud Server (ECS). Then we compare these results to the slowdown observed with *iLand* on the iPhone.

Group 3 setup. To demonstrate real-world compatibility and usability, we select an iPhone 14 Pro running iOS 17.3.1, for which no public jailbreak is currently available, and apply *iLand* to 60 top-ranked iOS apps, obtained from the US App Store in April 2025.

Table 4: Performance on SPEC CPU®2017. Numbers of Column 3-8 are the execution time in seconds.

Suite Name	Description	Group 1: on iPhoneX			Group 2: on Yitian710		
		native	Valgrind	*iLand*	native	QEMU-jit	QEMU-tci
perlbench_s	Perl interpreter	10.2	99.4	875.3	7.0	96.1	1981.0
gcc_s	GNU C compiler	19.6	228.0	973.6	16.6	147.8	2850.8
mcf_s	Route planning	60.4	272.6	785.4	40.1	230.1	3431.4
xalancbmk_s	XML to HTML conversion	34.2	415.6	3228.2	27.5	535.5	8622.4
x264_s	Video compression	12.5	111.0	1244.2	13.4	153.7	4512.3
xz_s	General data compression	23.2	61.8	663.7	16.8	66.1	1450.9
imagick_s	Image manipulation	21.0	353.8	911.8	16.0	613.4	10612.4
nab_s	Molecular dynamics	6.4	46.1	206.7	5.0	114.4	2033.0

9.2 Performance Efficiency

We conduct evaluation on iPhone X with `SPEC CPU®2017`. Given that the resource constraints of mobile devices, we employed the reduced-scale dataset `train` rather than `ref` as input throughout our evaluations. Adequate time intervals were implemented between consecutive test suites to cool down, preventing CPU throttling that may compromise benchmark accuracy. During testing, both of Valgrind and *iLand* run without any instrumentation. The execution times for native execution, Valgrind, qemu-jit, qemu-tci, and *iLand* in Groups 1 and 2 are listed in Table 4.

The results of Group 1 reveal that, compared to native execution, Valgrind and *iLand* exhibited a 5-10x and 15-90x increase in execution time, respectively. *iLand* translates each Arm64 instruction into several micro-operations, which results in a substantial increase in the number of instructions. In contrast, Valgrind natively run generated instructions and enabling a more lightweight instruction expansion. The performance gap between *iLand* and Valgrind (5-10x) is expected: Valgrind executes JIT-compiled native instructions, whereas *iLand* interprets micro-operations without RWX memory. Despite this gap, *iLand* remains practical for interactive use.

To contextualize *iLand*'s interpretation efficiency, Group 2 compares it against `qemu-tci`, another interpreter-based system. Because `qemu-tci` does not run on iOS and *iLand* does not run on Linux, we compare their respective slowdown factors relative to native execution on each platform, rather than absolute times. The results show that `qemu-tci` incurs 100-600x overhead over native execution, while *iLand* achieves 15-90x—a 5-15x advantage, especially on floating-point benchmarks. This advantage stems from *iLand*'s tight interpreting loop, which reduce dispatch overhead compared to `qemu-tci`'s generic tiny-code interpreter.

9.3 Instrumentation Capabilities

We categorize instrumentation capabilities along three dimensions: instruction-level, library call (libcall), and system call (syscall) instrumentation. Instruction-level instrumentation represents the most comprehensive capability, enabling arbitrary code insertion at any location without length constraints. Libcall and syscall instrumentation represent the ability to intercept standard library calls and hook supervisor calls (typically via `SVC`), respectively. Additionally, we consider maintaining instrumentation transparency to be an essential requirement, which involves evaluating whether the user-instrumented code preserves its original behavior.

We analyzed and compared the capabilities of *iLand* and existing popular instrumentation tools, as listed in Table 5. Only DBI tools and QEMU can achieve the three dimensions of instrumentation. Although QEMU shares equivalent instrumentation capabilities with *iLand*, it cannot support closed-source iOS. Compared to the other two DBI tools, only *iLand* can operate without relying on JIT and support iOS. Valgrind, even under jailbroken conditions, is limited to supporting GUI applications with basic functionality due to memory constraints, while Pin does not support Arm64.

Compared with Static Binary Instrumentation (SBI) tools, *iLand* exhibits technical superiority by code insertion at any instruction without constraints, whereas SBI tools suffer from instrumentation limitations dependent on both target instruction locations and surrounding code patterns. Notably, E9Patch encounters failures when processing specific instruction sequences, while ARMore demonstrates failures for binaries exceeding 256MB in size.

Compared to Frida and Cydia Substrate, which are widely used in iOS ecosystem, only *iLand* can insert arbitrary code at any position on the instruction-level. Substrate, designed to operate on jailbroken iOS devices, utilizes inline hooking, which requires space for several instructions to establish control flow hijacking. Frida's repackaging-based code injection mechanism offers limited analytical capacity, restricted to basic libcall hooks (via Global Offset Table modification), while lacking support for syscall interception.

9.4 Real-world Compatibility

We manually assess the compatibility of *iLand* across 64 apps from the US App Store's Top Free Apps list, downloaded in April 2025. We decrypted them by `trolldecrypt` [57] tool and emulated them using *iLand*, then manually tested whether their core functionality remained usable.

We list the tested apps in Appendix A.1. The apps cover 16

Table 5: Capabilities of DBA tools. ✓ denotes features that are deliberately designed and built-in supported. ○ indicates partial implementation with limitations or conditional constraints. × represents unsupported characteristics.

Type	Tool	Instruction Level Instrument	Library Call Intercept	System Call Intercept	Arm64 Support	JIT-free	iOS Adaptation	Instrument Transparency
DBI	*iLand*	✓	✓	✓	✓	✓	✓	✓
	Valgrind [46]	✓	✓	✓	✓	×	○	○
	Pin [43]	✓	✓	✓	×	×	×	○
SBI	E9Patch [16]	○	✓	✓	×	✓	×	×
	ARMore [6]	○	✓	✓	✓	✓	×	×
Other	QEMU [7]	✓	✓	✓	✓	✓	×	✓
	Frida [22]	×	✓	×	✓	✓	✓	×
	Substrate [12]	×	✓	✓	✓	×	✓	×

categories, with binary sizes broadly distributed between 40 and 400 MB. Of the 64 apps, 49 complete their core functionality under *iLand*—including login, browsing, search, video, and common app-specific features. These results indicate that *iLand* exhibits good compatibility. Notably, `Gmail`, a popular mail app, works correctly when logging in and browsing emails. `Youtube`, a well-known video app, plays video smoothly without lag or stutter, and other widely used social media apps, such as `X`, `Threads` and `Discord`, also support logging in and browsing real-time posts. Qualitatively, during manual testing the 49 working apps felt responsive: UI interactions (scrolling, tapping, typing) proceeded without perceptible delay, because UI rendering and media decoding are handled by native system libraries. These cases demonstrate the effectiveness of application-only emulation.

As for the other 11 apps that exhibited functional anomalies and 4 apps that self-terminated, our manual analysis indicated that these issues were mainly caused by their detection of *iLand*'s monitoring. A typical case involved an app attempting to delete files under its own bundle directory. Such operations are prohibited and failed on real iOS devices, but succeed within *iLand*'s virtualized environment, resulting in behavioral differences in subsequent executions. In theory, monitoring can be extended by monitor and instrumentation capabilities to further hide these traces, but how to find and hide the traces is beyond the scope of this paper.

9.5 Usability of App Vetting

Apple's App Review process substantially reduces the prevalence of malicious apps. However, prior work [15, 17] shows that policy-violating behavior can be embedded in complex app logic and may bypass review. We analyzed 60 top-ranked apps in the compatibility test and implemented a dynamic tracing plugin on top of *iLand* to collect four complementary types of evidence indicative of private-API use:

- Static SVC instructions. We count SVC occurrences in apps' binaries. Given that syscalls have convenient C wrappers, resorting to hand-written assembly is rare.
- Runtime syscall execution. We record executed SVC instructions and syscall function call, including the arguments passed to each syscall.
- Runtime dlsym usage. We log arguments passed to `dlsym` at runtime. Since the standard practice is to include headers and rely on link-time symbol resolution (simpler and faster), `dlsym` usage is rare.
- Indirect jumps to non-imported symbols. Using DBI's control-flow monitoring, we track all transfers into the DSC. We filter out transfers to imported symbols and known callbacks, and analyze the remaining targets.

Results. Across the 60 apps we tested, 24 embedded SVC instructions in their binaries, totaling 2,914 occurrences. Table 6 enumerates syscalls whose C-level wrappers are private APIs. Additionally, 15 apps invoke filesystem-related syscalls (e.g., SYS_lstat64, SYS_unlink) to access system directories.

Leveraging *iLand*'s control-flow monitoring, we traced all occurrences of transfers into non-imported symbols in DSC. We then resolved targets using the DSC local symbol table and filtered out 1,078 reasonable calls (e.g., Swift metadata accessor or ObjC callback). We manually verified the remaining targets and identified 13 apps that invoke private APIs. We list these APIs in Table 7. Among these, `SecTaskCopyValueForEntitlement` and `iokit_user_client_trap` are reject APIs; apps should not use them.

Discussion. We classify the motives for the private-API uses we found into three categories:

(A) Device fingerprinting [39] via system information that can be used to uniquely identify a device. Such data enables cross-account and cross-app tracking, which may raise privacy risks. For example, `statfs64` can reveal characteristics of the root file system that can be combined into fingerprinting features.

(B) Environment attestation and anti-tampering. Apps detect jailbreaking or repackaging by verifying code or signature integrity; for instance, the `SYS_csops` syscall can retrieve process's code-signing information; Apps may also search for injected code by enumerating memory with `vm_region`, or compare results from semantically equivalent APIs (e.g., syscalls versus libc wrappers) to detect tampering of API results.

Table 6: Uses of syscall detected by iLand in 60 top-ranked iOS apps.

Syscall Number	Corresponding C wrapper (Private API)	Functionality	Apps	Potential intent categories
SYS_csops	SecTaskCopyValueForEntitlement SecTaskCopyTeamIdentifier SecTaskCopySigningIdentifier	Fetch the process's code-signing status and entitlements	1	B
SYS_proc_info	proc_regionfilename proc_pidinfo proc_listpidspath	Fetch the process's path inforamtion	1	B
SYS_ptrace	ptrace	Prevent LLDB attaching	1	B
SYS_shared_region _check_np	_dyld_get_shared_cache_range	Fetch the DSC mapped start address and length	1	B
SYS_statfs64 SYS_statfs SYS_getfsstat64	statfs64 statfs getfsstat64	Get filesystem unique information	9	A
SYS___mac_syscall	__mac_syscall	Check if the process is in sandbox	2	B

Table 7: Uses of private APIs detected by iLand.

Indirect call to non-imported function	Functionality	Apps	Potential intent categories
iokit_user_client_trap	Interact to IOKit	1	B
SecTaskCreate- FromSelf SecTaskCopyValue- ForEntitlement	Get entitlements	1	B
task_dyld_process_- info_notify_get	Get loaded dylib	1	B
ubrk_* e.g., ubrk_next	Use lowlevel unicode processor	1	C
uregex_* e.g., uregex_start	Use lowlevel regex engine	2	C
vm_region_64 vm_region_recurse_64	Get memory mapping	5	B
ptrace	Prevent LLDB attaching	2	B
__sel_registerName	Register ObjC selector in another way	1	B
__syscall	System call in another way	2	B

(C) Performance or capability gains via low-level, undocumented APIs, aiming for faster execution or access to functionality that not publicly exposed.

We assess categories (A) and (B) as potential privacy risk for users, whereas category (C) can introduce stability risks for the app itself across OS updates.

We find that several private APIs can be invoked via direct system calls, yet syscall-based usage appears to receive insufficient attention during app review. For example, `SYS_csops` retrieves code-signing and entitlement information. The three `SecTaskCopy*` APIs (Table 6) that offer similar functionality are reject APIs; calls to those C APIs are rejected by TestFlight's automated review, but the syscall-based invocation bypassed the review. Beyond this, we observed 15 apps covertly reading system files via `SYS_open` and `SYS_read`—for instance, a system file in path */private/var/containers/...* accidentally contains device fingerprinting, which poses privacy risk. The syscall route is rather stealthy: i) syscall numbers are passed in registers and are therefore observable at runtime, complicating static analysis; and ii) whether a syscall is suspicious depends on its arguments—for example, `SYS_open` on sandboxed paths is benign, whereas attempts to open system directories are worth further scrutiny. *iLand* can reliably monitor such covert syscall activity.

We further performed reverse engineering to analyze how these apps evade App Review—for example, some construct control flow via ROP. We detail these techniques in § 2.3.

Limitation. Our case studies did not further analyze ObjC private-API usage, although *iLand* can instrument ObjC message sends and capture argument values. As with other dynamic analyses, achieving full path coverage is challenging. To mitigate this in our experiments, we manually completed account registration and login flows to drive deeper app logic. Following prior work [15], forced execution can improve coverage; *iLand* may support this in future work.

10 Related Work

Dynamic binary instrumentation (DBI) has many mature technologies. DynamoRIO [9], Intel Pin [43], Dyninst [8] and Valgrind [46], the most popular general DBI tools, achieve efficient instruction-level instrumentation through JIT-compiled code. To further improve traditional DBI, researchers proposed multiple optimization approaches, including translation optimization [13, 27, 34, 60], multi-thread efficiency improvement [29, 33, 48], and LLVM backend integration [19, 44, 59]. Although researchers have extended DBI capabilities to diverse platforms [4, 21, 26, 54], the closed-source iOS, running on limited-resource devices, carries a barrier to porting.

Several instrumentation solutions have been developed for iOS, such as Cydia Substrate [12], Dobby [36] and Fishhook [20]. These solutions require code injection on jailbreaking devices, overwriting the original instructions to jump to the inserted code when implementing inline-hook instrumentation, which is not transparent and is limited by instruction space, preventing instrumentation at arbitrary locations. As a result, there is a lack of a DBI solution capable of performing arbitrary instruction instrumentation on iOS.

11 Conclusion

In this paper, we present *iLand*, an instruction-level dynamic binary instrumentation framework for iOS. *iLand* translates and interprets binary instructions using its IR and atomic execution units without RWX memory. We propose application-only emulation, where system libraries run natively without interpreting, addressing mobile device memory and performance constraints. We implement the prototype of *iLand* as a standard iOS app and evaluate it on 64 top-ranked iOS apps from the App Store. In our compatibility study, 49 apps completed their core functionality under emulation, and 60 apps reached a sufficiently usable state for our app-vetting study. We also conducted an app-vetting study on these 60 apps and found 13 apps are still invoking private APIs. Our further analysis revealed new and stealthy methods to evade App Review, and found a new way of collecting sensitive information via direct SVC instruction invocation.

Acknowledgments

We thank the OSDI reviewers and our shepherd for their insightful feedback and guidance, which significantly improved this paper. We are also grateful to Alibaba Group for its support of this work and its research incentive policies.

References

[1] Anand, K., Smithson, M., Elwazeer, K., Kotha, A., Gruen, J., Giles, N., and Barua, R. A compiler-level intermediate representation based binary analysis and rewriting system. In *Proceedings of the 8th ACM European Conference on Computer Systems* (New York, NY, USA, 2013), EuroSys '13, Association for Computing Machinery, p. 295–308.

[2] Apple Inc. dyld: The dynamic linker, 2025. https://github.com/apple-oss-distributions/dyld, accessed April 2025.

[3] Apple Inc. Identifying high-memory use with Jetsam event reports, 2025. https://developer.apple.com/documentation/xcode/identifying-high-memory-use-with-jetsam-event-reports, accessed April 2025.

[4] Baiocchi, J., Childers, B. R., Davidson, J. W., and Hiser, J. Enabling dynamic binary translation in embedded systems with scratchpad memory. *ACM Trans. Embed. Comput. Syst. 11*, 4 (2012), 89:1–89:33.

[5] Bala, V., Duesterwald, E., and Banerjia, S. Dynamo: a transparent dynamic optimization system. In *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation* (New York, NY, USA, 2000), PLDI '00, Association for Computing Machinery, p. 1–12.

[6] Bartolomeo, L. D., Moghaddas, H., and Payer, M. ARMore: Pushing love back into binaries. In *32nd USENIX Security Symposium (USENIX Security 23)* (Anaheim, CA, Aug. 2023), USENIX Association, pp. 6311–6328.

[7] Bellard, F. QEMU, a fast and portable dynamic translator. In *2005 USENIX Annual Technical Conference (USENIX ATC 05)* (2005), USENIX Association.

[8] Bernat, A. R., and Miller, B. P. Anywhere, any-time binary instrumentation. In *Proceedings of the 10th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools* (New York, NY, USA, 2011), PASTE '11, Association for Computing Machinery, p. 9–16.

[9] Bruening, D. *Efficient, transparent, and comprehensive runtime code manipulation*. PhD thesis, Massachusetts Institute of Technology, Cambridge, MA, USA, 2004. https://dspace.mit.edu/entities/publication/b63c9921-1f04-4cd2-bb8b-75c5d2f6df3b.

[10] Cao, M., Hou, X., Wang, T., Qu, H., Zhou, Y., Bai, X., and Wang, F. Different is good: Detecting the use of uninitialized variables through differential replay. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security* (New York, NY, USA, 2019), CCS '19, Association for Computing Machinery, p. 1883–1897.

[11] Chan, N. Jailbreak for iphone, ipad, macbooks, and appletv's for versions 15 and higher, 2025. https://palera.in, accessed April 2025.

[12] Cydia. Cydia Substrate: The powerful code modification platform behind Cydia, 2025. https://www.cydiasubstrate.com, accessed April 2025.

[13] D'Antras, A., Gorgovan, C., Garside, J., and Luján, M. Low overhead dynamic binary translation on arm. *SIGPLAN Not. 52*, 6 (June 2017), 333–346.

[14] Decrypt IPA Store. Decrypt IPA Store: Free decrypted IPAs for educational and research purposes, 2025. https://decrypt.day, accessed April 2025.

[15] Deng, Z., Saltaformaggio, B., Zhang, X., and Xu, D. iris: Vetting private api abuse in ios applications. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security* (New York, NY, USA, 2015), CCS '15, Association for Computing Machinery, p. 44–56.

[16] Duck, G. J., Gao, X., and Roychoudhury, A. Binary rewriting without control flow recovery. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation* (New York, NY, USA, 2020), PLDI 2020, Association for Computing Machinery, p. 151–163.

[17] Egele, M., Kruegel, C., Kirda, E., and Vigna, G. PiOS: Detecting privacy leaks in iOS applications. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)* (2011), The Internet Society.

[18] Enck, W., Gilbert, P., Han, S., Tendulkar, V., Chun, B.-G., Cox, L. P., Jung, J., McDaniel, P., and Sheth, A. N. TaintDroid: An information-flow tracking system for realtime privacy monitoring on smartphones. *ACM Transactions on Computer Systems 32*, 2 (2014), 1–29.

[19] Engelke, A., and Schulz, M. Instrew: leveraging LLVM for high performance dynamic binary instrumentation. In *VEE '20: 16th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments, virtual event [Lausanne, Switzerland], March 17, 2020* (2020), S. Nagarakatte, A. Baumann, and B. Kasikci, Eds., ACM, pp. 172–184.

[20] FACEBOOK. fishhook: A library that enables dynamically rebinding symbols in Mach-O binaries running on iOS, 2025. https://github.com/facebook/fishhook, accessed April 2025.

[21] FEINER, P., BROWN, A. D., AND GOEL, A. Comprehensive kernel instrumentation via dynamic binary translation. In *Proceedings of the 17th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2012, London, UK, March 3-7, 2012* (2012), T. Harris and M. L. Scott, Eds., ACM, pp. 135–146.

[22] FRIDA. Frida: Dynamic instrumentation toolkit for developers, reverse engineers, and security researchers, 2025. https://frida.re, accessed April 2025.

[23] FRÖDER, L. A jailbreak for arm64 (A8–A11) and arm64e (A12–A16, M1–M2), 2025. https://ellekit.space/dopamine/, accessed April 2025.

[24] GAO, D., LIN, H., LI, Z., HUANG, C., LIU, Y., QIAN, F., GONG, L., AND XU, T. Trinity: High-performance mobile emulation through graphics projection. In *16th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2022, Carlsbad, CA, USA, July 11-13, 2022* (2022), M. K. Aguilera and H. Weatherspoon, Eds., USENIX Association, pp. 285–301.

[25] GOSAIN, A., AND SHARMA, G. A survey of dynamic program analysis techniques and tools. In *Proceedings of the 3rd International Conference on Frontiers of Intelligent Computing: Theory and Applications (FICTA) 2014 - Volume 1, Bhubaneswar, Odisa, India, 14-15 November 2014* (2014), S. C. Satapathy, B. N. Biswal, S. K. Udgata, and J. K. Mandal, Eds., vol. 327 of *Advances in Intelligent Systems and Computing*, Springer, pp. 113–122.

[26] GUHA, A., HAZELWOOD, K. M., AND SOFFA, M. L. Memory optimization of dynamic binary translators for embedded systems. *ACM Trans. Archit. Code Optim. 9*, 3 (2012), 22:1–22:29.

[27] HAWKINS, B., DEMSKY, B., BRUENING, D., AND ZHAO, Q. Optimizing binary translation of dynamically generated code. In *2015 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)* (2015), pp. 68–78.

[28] HAZELWOOD, K., AND KLAUSER, A. A dynamic binary instrumentation engine for the arm architecture. In *Proceedings of the 2006 International Conference on Compilers, Architecture and Synthesis for Embedded Systems* (New York, NY, USA, 2006), CASES '06, Association for Computing Machinery, p. 261–270.

[29] HAZELWOOD, K. M., LUECK, G., AND COHN, R. Scalable support for multithreaded applications on dynamic binary instrumentation systems. In *Proceedings of the 8th International Symposium on Memory Management, ISMM 2009, Dublin, Ireland, June 19-20, 2009* (2009), H. Kolodner and G. L. S. Jr., Eds., ACM, pp. 20–29.

[30] HEID, K., ANDRAE, V., AND HEIDER, J. Towards detecting device fingerprinting on iOS with API function hooking. In *Proceedings of the 2023 European Interdisciplinary Cybersecurity Conference* (New York, NY, USA, 2023), EICC '23, Association for Computing Machinery, p. 78–84.

[31] HENDERSON, A., YAN, L. K., HU, X., PRAKASH, A., YIN, H., AND MCCAMANT, S. Decaf: A platform-neutral whole-system dynamic binary analysis platform. *IEEE Transactions on Software Engineering 43*, 2 (2017), 164–184.

[32] HENZE, L. Fugu15: A semi-untethered permasigned jailbreak for iOS 15, 2025. https://github.com/pinauten/Fugu15, accessed April 2025.

[33] HONG, D., WU, J., YEW, P., HSU, W., HSU, C., LIU, P., WANG, C., AND CHUNG, Y. Efficient and retargetable dynamic binary translation on multicores. *IEEE Trans. Parallel Distributed Syst. 25*, 3 (2014), 622–632.

[34] JIA, N., YANG, C., WANG, J., TONG, D., AND WANG, K. SPIRE: improving dynamic binary translation through spc-indexed indirect branch redirecting. In *ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (co-located with ASPLOS 2013), VEE '13, Houston, TX, USA, March 16-17, 2013* (2013), S. Muir, G. Heiser, and S. M. Blackburn, Eds., ACM, pp. 1–12.

[35] JIA, X., ZHANG, C., SU, P., YANG, Y., HUANG, H., AND FENG, D. Towards efficient heap overflow discovery. In *26th USENIX Security Symposium (USENIX Security 17)* (Vancouver, BC, Aug. 2017), USENIX Association, pp. 989–1006.

[36] JMPEWS. Dobby: A lightweight, multi-platform, multi-architecture hook framework, 2025. https://github.com/jmpews/Dobby, accessed April 2025.

[37] KEDIA, P., AND BANSAL, S. Fast dynamic binary translation for the kernel. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles* (New York, NY, USA, 2013), SOSP '13, Association for Computing Machinery, p. 101–115.

[38] KELLNER, A., HORLBOGE, M., RIECK, K., AND WRESSNEGGER, C. False sense of security: A study on the effectivity of jailbreak detection in banking apps. In *2019 IEEE European Symposium on Security and Privacy (EuroS&P)* (2019), pp. 1–14.

[39] KOLLNIG, K., SHUBA, A., VAN KLEEK, M., BINNS, R., AND SHADBOLT, N. Goodbye tracking? impact of ios app tracking transparency and privacy labels. In *Proceedings of the 2022 ACM Conference on Fairness, Accountability, and Transparency* (New York, NY, USA, 2022), FAccT '22, Association for Computing Machinery, p. 508–520.

[40] LAWTON, K. P. Bochs: A portable PC emulator for Unix/X. *Linux Journal 1996*, 29es (1996), 7–es. https://www.linuxjournal.com/article/1310.

[41] LEVIN, J. *MacOS and iOS Internals, Volume II: Kernel Mode*. Technologeeks Press, 2019. https://www.newosxbook.com/.

[42] LOGHIN, D. Are ARM cloud servers ready for database workloads? an experimental study. *IEEE Transactions on Cloud Computing 12*, 3 (2024), 818–829.

[43] LUK, C.-K., COHN, R., MUTH, R., PATIL, H., KLAUSER, A., LOWNEY, G., WALLACE, S., REDDI, V. J., AND HAZELWOOD, K. Pin: building customized program analysis tools with dynamic instrumentation. PLDI '05, Association for Computing Machinery, p. 190–200.

[44] LYU, Y., HONG, D., WU, T., WU, J., HSU, W., LIU, P., AND YEW, P. DBILL: an efficient and retargetable dynamic binary instrumentation framework using llvm backend. In *10th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments, VEE '14, Salt Lake City, UT, USA, March 01 - 02, 2014* (2014), M. Hirzel, E. Petrank, and D. Tsafrir, Eds., ACM, pp. 141–152.

[45] MULLINER, C., OBERHEIDE, J., ROBERTSON, W., AND KIRDA, E. PatchDroid: Scalable third-party security patches for Android devices. In *Proceedings of the 29th Annual Computer Security Applications Conference* (New York, NY, USA, 2013), ACSAC '13, Association for Computing Machinery, p. 259–268.

[46] NETHERCOTE, N., AND SEWARD, J. Valgrind: a framework for heavy-weight dynamic binary instrumentation. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation* (New York, NY, USA, 2007), PLDI '07, Association for Computing Machinery, p. 89–100.

[47] PWN20WND. The most advanced jailbreak tool for iOS 11.0–14.8, 2025. https://unc0ver.dev/, accessed April 2025.

[48] ROBSON, D., AND STRAZDINS, P. E. Parallelisation of the valgrind dynamic binary instrumentation framework. In *IEEE International Symposium on Parallel and Distributed Processing with Applications, ISPA 2008, Sydney, NSW, Australia, December 10-12, 2008* (2008), IEEE Computer Society, pp. 113–121.

[49] SANG, Q., WANG, Y., LIU, Y., JIA, X., BAO, T., AND SU, P. Airtaint: Making dynamic taint analysis faster and easier. In *IEEE Symposium on Security and Privacy, SP 2024, San Francisco, CA, USA, May 19-23, 2024* (2024), IEEE, pp. 3998–4014.

[50] SHAN, Z., GUO, H., AND PANG, J. BTMD: A framework of binary translation based malcode detector. In *2012 International Conference on Cyber-Enabled Distributed Computing and Knowledge Discovery, CyberC 2012, Sanya, China, October 10-12, 2012* (2012), IEEE Computer Society, pp. 39–43.

[51] SOLIMEO, A., CAPACCI, L., TAINO, S., AND MONTANARI, R. MADIOS: dynamic app vulnerability analysis in non-jailbroken devices. In *Proceedings of the Second Italian Conference on Cyber Security, Milan, Italy, February 6th - to - 9th, 2018* (2018), E. Ferrari, M. Baldi, and R. Baldoni, Eds., vol. 2058 of *CEUR Workshop Proceedings*, CEUR-WS.org.

[52] SPREITZENBARTH, M., SCHRECK, T., ECHTLER, F., ARP, D., AND HOFFMANN, J. Mobile-Sandbox: combining static and dynamic analysis with machine-learning techniques. *International Journal of Information Security 14* (2015), 141–153.

[53] TANG, Z., TANG, K., XUE, M., TIAN, Y., CHEN, S., IKRAM, M., WANG, T., AND ZHU, H. iOS, your OS, everybody's OS: Vetting and analyzing network services of iOS applications. In *29th USENIX Security Symposium (USENIX Security 20)* (Aug. 2020), USENIX Association, pp. 2415–2432.

[54] VILLA, O., STEPHENSON, M., NELLANS, D., AND KECKLER, S. W. Nvbit: A dynamic binary instrumentation framework for nvidia gpus. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture* (New York, NY, USA, 2019), MICRO '52, Association for Computing Machinery, p. 372–383.

[55] WANG, T., LU, K., LU, L., CHUNG, S., AND LEE, W. Jekyll on iOS: When benign apps become evil. In *22nd USENIX Security Symposium (USENIX Security 13)* (2013), USENIX Association, pp. 559–572.

[56] WANG, Z., LI, J., WU, C., YANG, D., WANG, Z., HSU, W.-C., LI, B., AND GUAN, Y. Hspt: Practical implementation and efficient management of embedded shadow page tables for cross-isa system virtual machines. In *Proceedings of the 11th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments* (New York, NY, USA, 2015), VEE '15, Association for Computing Machinery, p. 53–64.

[57] WH1TE4EVER. TrollDecryptor: Decrypt iOS apps for TrollStore, 2023. https://github.com/wh1te4ever/TrollDecryptor, accessed April 2025.

[58] WU, S., LI, J., ZHOU, H., FANG, Y., ZHAO, K., WANG, H., QIAN, C., AND LUO, X. CydiOS: A model-based testing framework for iOS apps. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis* (New York, NY, USA, 2023), ISSTA 2023, Association for Computing Machinery, p. 1–13.

[59] YADAVALLI, S. B., AND SMITH, A. Raising binaries to llvm ir with mctoll (wip paper). In *Proceedings of the 20th ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, and Tools for Embedded Systems* (New York, NY, USA, 2019), LCTES 2019, Association for Computing Machinery, p. 213–218.

[60] ZHANG, X., GAO, X., GUO, Q., HUANG, J., LIU, H., AND MENG, X. VBIW: optimizing indirect branch in dynamic binary translation. In *10th IEEE International Conference on High Performance Computing and Communications & 2013 IEEE International Conference on Embedded and Ubiquitous Computing, HPCC/EUC 2013, Zhangjiajie, China, November 13-15, 2013* (2013), IEEE, pp. 1456–1462.

A Appendix

The table lists the 64 evaluated apps by decreasing binary size and all were drawn from the App Store's Top Free list in April 2025 on iOS 16.7.7.

Table A.1: Compatibility evaluation. `Binaries` represents the total size of the app's binaries. `Eval.` delineates the outcomes, where ✓, ○ and × indicate full, partial, no functionality, respectively.

App Name	Chart	Binaries (MB)	Eval.	Test Behaviors
CapCut	Photo & Video	429.9	✓	View
T-Life	Lifestyle	370.1	✓	View
Sam's Club	Shopping	352.7	✓	View
Walmart	Shopping	351.1	✓	View
Paypal	Finance	322.6	✓	Login
Facebook	Social Network	301.1	×	Abort
Instagram	Photo & Video	289.1	○	Login Denied
Gmail	Productivity	287.4	✓	Login; Email
LinkedIn	Business	284.1	✓	Login; View
GroupMe	Social Network	282.1	○	Login Denied
DoorDash	Food & Drink	269.0	✓	Login; View
Snapchat	Photo & Video	266.9	○	Login Denied
X	News	259.9	✓	Login; View
Reddit	News	245.8	✓	Login; View
Duolingo	Education	238.7	○	Homepage Hang
Cash App	Finance	221.0	○	Homepage Hang
Airbnb	Travel	220.8	✓	Login; View
McDonald's	Food & Drink	216.5	✓	Login; View
Zoom Workplace	Business	211.1	✓	Login; Meeting
Pinterest	Lifestyle	209.5	✓	Login; View
Whatnot	Shopping	207.7	○	Login Denied
Instacart	Food & Drink	205.5	✓	View
Google	Utilities	197.9	✓	Search; Video
Spotify	Music	187.9	✓	Login; View
Youtube	Photo & Video	177.1	✓	Search; Video
GameChanger	Sports	174.7	○	Login Hang
Twitch	Photo & Video	160.0	✓	Login; Video
Credit Karma	Finance	153.8	○	Homepage Hang
Google Docs	Productivity	151.6	✓	Login; View
Google Slides	Productivity	146.0	✓	Login; View
Affirm	Shopping	140.4	✓	View
Klarna	Shopping	139.6	×	Abort
Messenger	Social Network	134.1	○	Login Denied
TurboTax	Finance	130.8	○	Login Hang
Macy's	Shopping	130.6	✓	View
Google Chrome	Utilities	125.9	×	Abort
Progressive	Finance	117.1	✓	View
ReelShort	Entertainment	115.8	✓	Video
DramaBox	Entertainment	108.1	✓	Video
Masters 2025	Sports	107.8	✓	View
Indeed Jobs	Business	104.5	✓	Login; View
Gauth	Education	99.9	✓	View
Scoopz	Entertainment	96.5	✓	Video
MLB	Sports	92.1	✓	Login; View
Chime	Finance	91.7	✓	View
Remini	Photo & Video	88.7	✓	View
Cleanup	Utilities	88.1	✓	View
Starbucks	Food & Drink	84.8	✓	View
Tubi	Entertainment	84.6	✓	Login; Video
Audible	Books	84.1	✓	View; Listen
Best Buy	Shopping	82.3	✓	View
Google Gemini	Productivity	77.3	✓	Login; Talk
Amazon	Shopping	74.1	○	Homepage Hang
Taco Bell	Food & Drink	65.5	×	Abort
Threads	Social Network	65.0	✓	Login; View
Shop	Shopping	58.9	✓	Login; View
Grok	Productivity	54.2	✓	Talk
Substack	News	53.3	✓	Login; View
Temu	Shopping	49.4	✓	Login; View
TooGoodToGo	Food & Drink	48.4	✓	View
Canva	Photo & Video	47.3	✓	Login; View
Discord	Social Network	46.6	✓	Login; View
Roku	Entertainment	38.5	✓	View
Aritzia	Shopping	37.5	✓	Login; View

Tessera: A Holistic Pipeline Parallelism Framework for Trillion-Parameter Heterogeneous MoE Training (Operational Systems)

Weifang Hu* SCTS/BDTS, HUST
Langshi Chen* Alibaba Cloud
Man Yuan* Alibaba Cloud
Youyang Yao Alibaba Cloud
Xiulong Yuan Alibaba Cloud
Li Tian Alibaba Cloud
Yong Li Alibaba Cloud
Wei Lin Alibaba Cloud
Xuanhua Shi† SCTS/BDTS, HUST
Zhengping Qian† Alibaba Cloud
Jingren Zhou Alibaba Cloud

Abstract

Training trillion-parameter models relies on pipeline parallelism to scale across thousands of accelerators. However, architectures exemplified by the Qwen family are evolving from uniform Transformer blocks into heterogeneous compositions that combine sparse Mixture-of-Experts (MoE) with different attention variants. This heterogeneity breaks the uniformity assumption behind existing pipeline systems: partitions chosen from serial layer costs become imbalanced after communication is overlapped with computation, because different layer combinations hide different fractions of their communication. Furthermore, at the scale of 10K+ GPUs, MoE routing variation adds a runtime source of stalls, creating transient idle slots that static plans cannot anticipate.

We present Tessera, a framework that co-optimizes partitioning and overlap scheduling for heterogeneous pipeline training, and adapts to runtime dynamics. Tessera introduces: (1) an *overlap scheduler* that synthesizes fine-grained interleavings for each layer combination; (2) an *overlap-aware partitioner* that uses profiled post-overlap cost to select partitions balanced for parallel execution; and (3) a *dynamic bubble optimizer* that fills routing-induced idle slots with movable tasks at runtime. Deployed on production clusters for pre-training Qwen3 and Qwen3-Next, Tessera improves throughput by 20%–33% over our production baseline across five workloads at scales from 4,096 to 12,288 GPUs, reaching 39% MFU on a trillion-parameter model. In controlled experiments against Megatron-Core MoE with public recipes, Tessera achieves up to 1.24× higher MFU.

1 Introduction

Foundation models are evolving from uniform Transformer stacks into heterogeneous compositions that mix hybrid attention variants with sparse Mixture-of-Experts (MoE) blocks [3, 24, 37]. The uniform overlap and partitioning policies behind current pipeline systems no longer transfer cleanly. In training the **Qwen3** and **Qwen3-Next** model families on production clusters exceeding 10,000 GPUs, we find that structural heterogeneity, most visible in Qwen3-Next, breaks the assumptions these systems rely on.

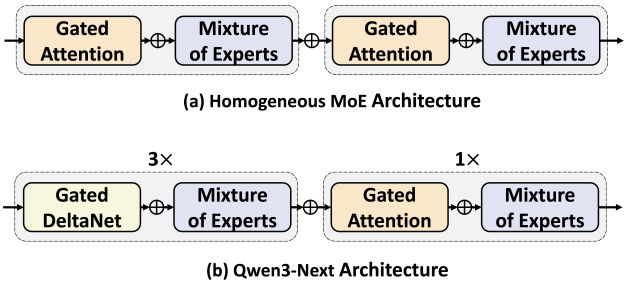


Figure 1: **Heterogeneity in Qwen3-Next Architecture.** (a) standard Transformer layer; (b) Qwen3-Next interleaves Gated DeltaNet and standard attention layers.

The cost this heterogeneity exposes lies in how communication is overlapped with computation. MoE layers require heavy All-to-All (A2A) communication for token dispatch and combination [8, 11, 26]. Pipeline schedules can hide this cost by overlapping A2A transfers with computation from other in-flight microbatches. For uniform architectures (Fig. 1(a)), every stage has a similar compute-to-communication ratio, so a single overlap strategy applied uniformly suffices.

Qwen3-Next (Fig. 1(b)) breaks this assumption. It interleaves linear-attention layers (Gated DeltaNet) with standard softmax-attention layers, each followed by a sparse MoE block. These layer types have different compute profiles, especially at long sequence lengths. When such diverse layers are grouped into pipeline stages, different combinations present different compute-communication timelines. A fixed overlap strategy applies one interleaving to all; it fits some combinations but leaves more communication exposed in others. In our measurements, the resulting overlap gain differs by 3×.

*Weifang Hu, Langshi Chen, and Man Yuan contributed equally. This work was done while Weifang Hu was an intern at Alibaba Cloud.
†Corresponding authors. Emails: xhshi@hust.edu.cn, zhengping.qzp@alibaba-inc.com

This variation creates a cyclic dependency between partitioning and overlap scheduling. The partition determines which layer combinations co-execute and thus which overlaps are possible. Choosing a good partition, however, requires accurately estimating each combination's post-overlap cost, which requires scheduling it and measuring the resulting execution on hardware. A partition balanced for serial execution cost can become the pipeline bottleneck after overlap, because stages that look equal in serial time hide vastly different fractions of their communication. Existing pipeline systems sidestep this coupling by assuming uniform overlap efficiency—an assumption that holds for homogeneous models but fails under heterogeneity. In our experience, domain experts then spend weeks hand-tuning this interplay for each new model architecture.

Even after static co-optimization addresses the predictable structural coupling, MoE routing stochasticity leaves residual runtime bubbles: the number of tokens dispatched to each expert fluctuates across iterations, creating ephemeral idle slots that a static schedule cannot reliably anticipate or exploit.

We present Tessera, a pipeline parallelism framework that co-optimizes overlap scheduling and partitioning, and adapts to runtime dynamics. Designed as a plug-in to the host training framework, it requires no invasive modifications to the existing training loop. Within a given pipeline schedule (e.g., interleaved 1F1B), Tessera synthesizes a specialized overlap schedule for each layer combination and uses its measured post-overlap cost to guide partition selection. At runtime, a *Dynamic Bubble Optimizer* monitors routing metadata to predict idle slots and fills them with deferred work. The design has three parts:

- **Overlap Scheduling.** Tessera observes that overlap efficiency depends on the specific layer combination, not just aggregate arithmetic cost. It replaces fixed strategies by synthesizing and profiling a specialized overlap schedule for each combination.
- **Overlap-Aware Partitioning.** Using profiled post-overlap costs, Tessera selects a partition that balances measured post-overlap cost rather than serial arithmetic, breaking the cyclic dependency via a profile-then-partition sequence.
- **Dynamic Bubble Optimization.** At runtime, Tessera predicts routing-induced idle slots from token-distribution metadata and fills them with deferred tasks, recovering throughput that static planning leaves unexploited.

We implemented Tessera atop Alibaba's internal Megatron-LM and deployed it on production clusters with over 10,000 NVIDIA Hopper GPUs for pre-training the Qwen3 and Qwen3-Next families. Compared to our production-optimized internal baseline, Tessera improves throughput by 20%–33% across five workloads at scales from 4,096 to 12,288 GPUs,

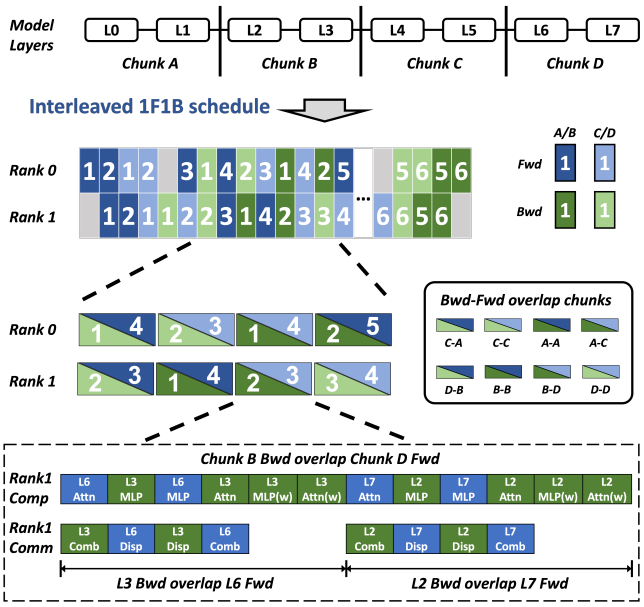


Figure 2: **Mechanism of Inter-Microbatch Overlap.** The bottom panel illustrates how the backward pass of Chunk *B* is overlapped with the forward pass of Chunk *D* by interleaving decomposed operations.

with peak MFU reaching 39% on a trillion-parameter run. In controlled experiments on a 256-GPU cluster against Megatron-Core MoE with publicly available per-model recipes, Tessera achieves up to 1.24× higher MFU. Tessera has been in continuous production use since April 2025.

2 Background and Motivation

2.1 Pipeline Execution and Overlap Pairs

This section defines the pipeline execution model that underlies the rest of the paper. Whereas Tensor Parallelism [22] splits operators within a layer and Context Parallelism [18] partitions the sequence dimension across devices, **Pipeline Parallelism (PP) [7]** partitions the model *across* layers into groups called *chunks*. Each chunk is assigned to a pipeline stage mapped to a specific device topology. To reduce the idle time (bubble) inherent in sequential stage execution, each training batch is further split into *microbatches* that can be pipelined through stages concurrently. A schedule governs how microbatches are dispatched. The widely adopted 1F1B schedule [22] alternates one forward and one backward microbatch per stage to bound memory while keeping the pipeline busy. Its *interleaved* variant further reduces bubble by assigning each physical rank multiple "virtual" stages, each corresponding to a distinct model chunk.

Inter-microbatch overlap. In an interleaved schedule, adjacent operations on a rank can come from different chunks and microbatches, allowing one chunk's communication to

overlap another's computation. We refer to communication-computation overlap across different microbatches as *inter-microbatch overlap*. As shown in Fig. 2, chunks from different microbatches alternate on the same rank, and the A2A communication of one chunk's backward pass (e.g., Chunk *B*) can execute concurrently with the computation of a subsequent chunk's forward pass (e.g., Chunk *D*). We call two chunk operations that share such a concurrent execution window an *overlap pair*. An overlap pair includes the execution direction of each chunk, such as backward-on-*B* overlapped with forward-on-*D* in Fig. 2. Thus, pair types such as `B-D` and `D-B` are not merely unordered chunk sets; they can represent different directional overlap cases. For each overlap pair, the relevant performance metric is not the serial sum of its constituent operations, but the *post-overlap cost*: the resulting makespan of the co-scheduled overlap pair.

The uniformity assumption. Traditional PP overlap strategies rely on a strong implicit assumption: the model consists of uniform "Transformer bricks" with similar arithmetic intensity and communication patterns [33]. Under this assumption, all overlap pairs are structurally identical, so a hand-crafted fixed overlap template combined with layer-count balancing naturally yields both effective overlap and a load-balanced pipeline. Next-generation architectures break this uniformity.

2.2 Heterogeneous MoE Architectures

To scale Mixture-of-Experts (MoE) models, experts are partitioned and distributed across multiple devices [11]. This distribution necessitates All-to-All (A2A) communication for token dispatch and combination. At scales beyond several thousand GPUs, A2A latency constitutes a substantial fraction of iteration time, making it the primary target for overlap.

Prior works like Comet [39] employ *intra-microbatch overlap*, fusing communication with computation from the *same* microbatch. However, for heterogeneous models this approach forces kernel fragmentation by slicing the sequence dimension, reducing arithmetic intensity (detailed in §5). Inter-microbatch overlap avoids this fragmentation but shifts the hard problem: given a pipeline stage partition, which overlap pairs are induced, and how should their sub-operations interleave? Exploiting this form of overlap requires decomposing layers into fine-grained schedulable units so that communication and computation from different chunks can be interleaved. We refer to each such unit as a *task* (e.g., `Dispatch`, `MLP`, `Combine`).

Qwen3-Next: a concrete case of pair diversity. Qwen3-Next adopts a hybrid attention design in which every four consecutive layers follow a 3:1 pattern: three Gated DeltaNet (GDN) linear-attention layers followed by one full softmax-attention layer. Each layer is followed by a sparse MoE feed-forward block. Under production sequence lengths, this operator mix produces up to 10× compute-time asymmetry between adjacent attention operations. As shown in Fig. 3, when these

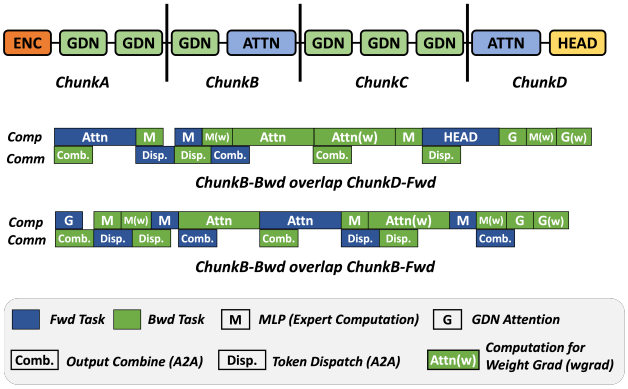


Figure 3: **Combinatorial Explosion of Overlap Patterns.** The Qwen3-Next architecture produces asymmetric chunks, resulting in diverse overlap pairings that require structurally different fine-grained plans. For clarity, MoE blocks following the GDN and standard attention are not shown.

asymmetric chunks are assigned to virtual stages, they form overlap pairs with diverse compute-communication profiles. Each letter (A, B, C, D) denotes a chunk type defined by its attention/MoE composition. Even a simple 2-stage pipeline produces up to eight distinct pair types (e.g., `B-D`, `D-B`), each requiring a different fine-grained interleaving. A single overlap template cannot serve this diversity.

2.3 The Co-Optimization Problem

Existing systems optimize pipeline training as a waterfall: first partition by serial layer cost, then apply a fixed overlap template, then execute the resulting static plan. Heterogeneous MoE violates each step of this pipeline. We identify three compounding failures: the overlap scheduling space is combinatorial (§2.3.1), the partition objective is misleading (§2.3.2), and runtime variation leaves residual idle slots that static planning cannot predict (§2.3.3). The first two form a static planning cycle that requires joint optimization. The third motivates runtime adaptation.

2.3.1 Combinatorial Overlap Space

Template-based schedulers cannot handle the pair diversity created by heterogeneous chunk types. In a homogeneous model, every overlap pair is identical (e.g., `A-A`), and a single interleaving strategy works universally. In Qwen3-Next, even a single partition of the minimal 2-stage case produces up to eight distinct pair types, each requiring a unique fine-grained schedule: a computation-intensive chunk might successfully hide a communication-intensive chunk's A2A, whereas two communication-heavy chunks may result in significant exposed latency. Existing systems rely on rigid templates (e.g., always aligning "forward combine" with "backward attention"). With asymmetric chunks, the optimal interleaving de-

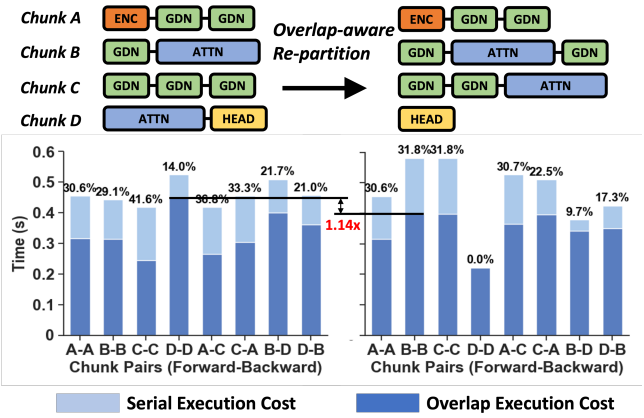


Figure 4: **Impact of Variable Overlap Efficiency.** Variation in overlap gains under two partition plans. Data collected from training 8 layers of the Qwen3-Next-80B model with sequence length 256K on 128 GPUs. Left: A partition balanced for serial execution becomes imbalanced in post-overlap execution. Right: Tessera's overlap-aware partitioner finds a plan that balances the post-overlap cost.

pends on the specific hardware resource usage of the constituent chunks. Manually engineering schedules for this combinatorial space is impractical.

2.3.2 Variable Overlap Efficiency

Standard partitioners sum the sequential execution time of layers and balance this sum across stages. This strategy implicitly assumes that *overlap efficiency*, the fraction of communication hidden, is constant across the model.

In heterogeneous MoE models, overlap efficiency varies drastically. Fig. 4 (Left) presents profiling data from a balanced partition for the model architecture illustrated in Fig. 3. The chunk pair `C-C` achieves a 41.6% reduction in latency via overlap, while pair `D-D` achieves only 14.0% because both sides are computation-dominated with little A2A communication to hide. A partition balanced for serial execution can thus manifest severe imbalances during parallel execution.

An optimal partitioner must be *overlap-aware*: it should deliberately create serially "imbalanced" assignments if those pairings have complementary compute/communication profiles that maximize overlap. As shown in Fig. 4 (Right), the serial-cost-balanced partition has a 1.14× higher bottleneck post-overlap cost than the overlap-aware partition.

Moreover, the dependency is bidirectional. The overlap schedule determines how much communication each pair hides, which in turn determines which partition minimizes the bottleneck. A partition that appears serially imbalanced may yield the lowest post-overlap cost when its pairings have complementary profiles. Partition quality therefore cannot be evaluated without first scheduling and measuring each candidate pair. Partitioning and overlap scheduling are thus cyclically dependent.

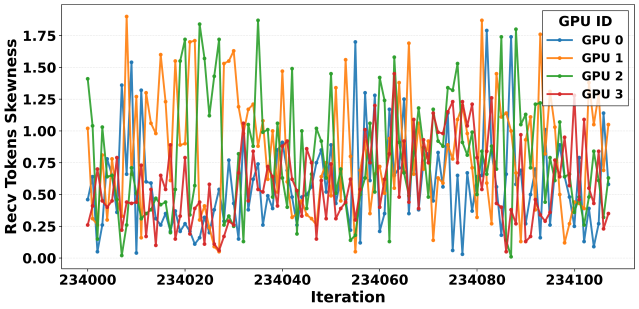


Figure 5: **Runtime Token Load Variation.** The distribution of received tokens per device varies across iterations, creating dynamic imbalance that static planning cannot address.

2.3.3 Runtime Stochasticity

Even an optimal static plan is vulnerable to runtime stochasticity. The number of tokens routed to specific experts fluctuates across iterations, causing the duration of MoE compute kernels to vary. In Qwen3-Next, this effect is amplified by fine-grained sparse routing: each token activates only 10 of 512 experts, so each expert receives a smaller expected share of the token pool. This lower routing coverage increases the relative variation of per-expert and per-rank token counts across iterations, as reflected in the per-device load-skewness fluctuations shown in Fig. 5. These fluctuations create ephemeral idle slots that no static schedule can predict.

The key observation that makes runtime optimization feasible is that not all rank-local work lies on the microbatch critical path. We call the critical-path tasks that define stage latency the *backbone tasks*. The remaining timing-flexible tasks (e.g., weight-gradient computation, gradient reduction) that can fill transient gaps are called *movable tasks*. Deferring movable tasks and injecting them into routing-induced idle slots can recover throughput that static planning leaves unexploited.

Summary. Partitioning and overlap scheduling are cyclically dependent: partition boundaries determine which overlap pairs form, while overlap efficiency determines which partition is optimal. Runtime routing variation adds a dynamic dimension that no static solution can address. Together, these challenges render stratified optimization pipelines inadequate for heterogeneous MoE training.

3 Tessera Design

To resolve the partitioning–overlap cycle (§2.3.2), Tessera fixes the high-level pipeline schedule (e.g., interleaved 1F1B) and co-optimizes partitioning and overlap scheduling within it. A static planner produces an execution plan (§3.1), and a plan-agnostic execution engine (§4) interprets it at runtime,

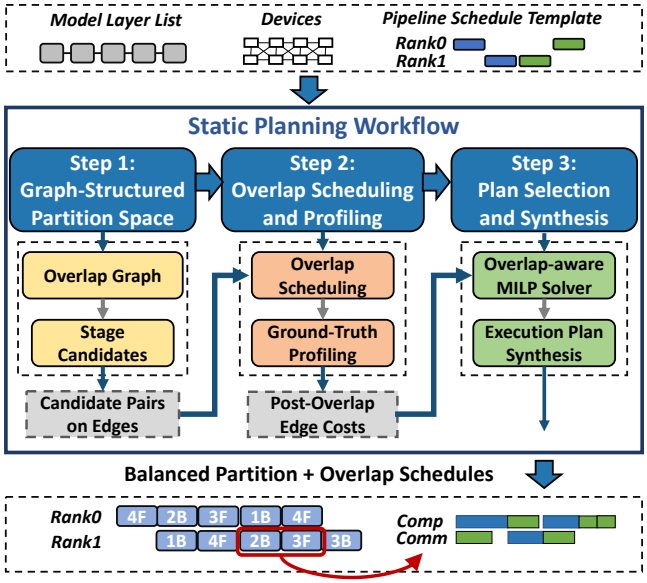


Figure 6: Overlap-aware static planning workflow.

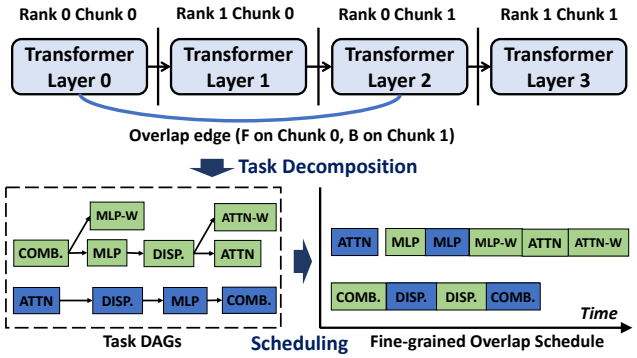


Figure 7: Decomposing an overlap edge into task DAGs and synthesizing a fine-grained overlap schedule. The figure highlights one forward–backward overlap edge on Rank 0, omitting other pass-direction edges. Each stage contains a single layer for clarity.

decoupling scheduling logic from the host framework. To handle stochasticity that the static plan cannot predict, a *Dynamic Bubble Optimizer* fills routing-induced idle slots with eligible movable tasks, without changing partition boundaries or the planned order of backbone tasks (§3.2).

3.1 Static Planning

We refer to the fixed high-level pipeline schedule (e.g., interleaved 1F1B) as the *pipeline schedule template*. Given this template, Tessera resolves the static partitioning–overlap cycle through a *candidate-generation-and-profiling* workflow. Fig. 6 summarizes this workflow. Step 1 constructs an overlap graph from the template and attaches bounded stage candidates to each virtual stage (§3.1.1). Step 2 instantiates each candidate pair on an overlap edge as a concrete overlap pair, synthesizes its fine-grained overlap schedule, and profiles its post-overlap cost on a reference device group (§3.1.2). Step 3 selects one candidate per stage via MILP to minimize the bottleneck post-overlap cost, then materializes the selected partition and per-pair schedules into the final execution plan (§3.1.3).

3.1.1 Step 1: Overlap Graph and Candidate Space

The pipeline schedule template determines a set of virtual stages and which of them may execute microbatch operations concurrently on the same rank. These relationships define an *overlap graph* $G=(\mathcal{S},\mathcal{E})$. Each node $s \in \mathcal{S}$ is a virtual pipeline stage under the template. Tessera adds an *overlap edge* $e=(s,t) \in \mathcal{E}$ for each overlap opportunity that the template creates on a pipeline rank, where endpoints s and t may be the same virtual stage. Each edge records the pass directions of the two endpoint operations, so the same endpoint pair may yield multiple edges, e.g., forward-on-s with backward-on-t and backward-on-s with forward-on-t. A self-loop denotes an overlap opportunity between operations of the same virtual stage from different microbatches. With the graph topology fixed by the template, the layers each node carries remain to be selected.

Tessera starts from a serial-cost-balanced baseline partition as the initial assignment of layers to virtual stages. To explore alternative partitions, Tessera generates a candidate set C_s for each stage s. A *stage candidate* $c \in C_s$ specifies one possible model chunk for virtual stage s: a contiguous layer range generated by perturbing boundaries around the baseline. For any candidate pair $(c \in C_s, d \in C_t)$ on an overlap edge $e=(s,t)$, the post-overlap cost lies between $\max(T_c, T_d)$ (perfect overlap) and T_c+T_d (zero overlap), where T_c and T_d denote the serial costs of candidates c and d respectively. Tessera restricts candidates to a bounded neighborhood around the baseline to keep overlap scheduling and profiling tractable.

Together, the overlap graph and stage candidates define the search space for joint optimization. Selecting one candidate per node determines a concrete pipeline partition. Each edge poses a scheduling subproblem: for a given pair of endpoint candidates, Tessera must synthesize a fine-grained overlap schedule and measure its post-overlap cost. The profiled edge costs then feed the partition solver that minimizes the bottleneck post-overlap cost across the graph.

3.1.2 Step 2: Overlap Scheduling and Profiling

Once candidates c and d are assigned to the endpoint stages of edge e, the layer ranges, pass directions, and device topology are fully determined, specifying a concrete overlap pair. For each such pair, Tessera synthesizes a fine-grained overlap schedule and measures the post-overlap cost $T_{e,c,d}$.

Algorithm 1 Event-driven scheduling for overlap-pair task DAGs

Input: task DAGs D_1, D_2 (overlap pair)
Output: Overlap schedule S for tasks in $D_1 \cup D_2$

Ready $\leftarrow$ initial tasks with no unsatisfied dependencies
Running $\leftarrow \emptyset$; *Pending* $\leftarrow \emptyset$; *Time* $\leftarrow 0$
while *Ready* $\cup$ *Running* $\neq \emptyset$ **do**
  for resource $r \in \{Comp, Comm\}$ where r is idle **do**
    $\tau \leftarrow$ SELECT(*Ready*$[r]$) ▷ backbone-first gap-fit selection
    if $\tau = \perp$ **then continue**
    if ISMOVABLE(τ) **and** *Running* $\neq \emptyset$ **and** SHOULDDEFER(τ, *Time*) **then**
      move τ from *Ready*$[r]$ to *Pending*$[r]$ ▷ defer to protect backbone completion
    else
      remove τ from *Ready*
      SCHEDULE(τ, *Time*)
  Time $\leftarrow$ next task completion event
  Update *Ready* with newly unblocked tasks in D_1, D_2
  ▷ Backfill deferred movable tasks
for $\tau \in$ *Pending* in reverse readiness order **do**
  BACKFILL(τ, S) ▷ best-fit residual slot or tail

Tessera decomposes each overlap pair into two task DAGs, one per chunk, as illustrated in Fig. 7. Each task is an atomic scheduling unit with a duration, resource type, and dependency constraints. These attributes determine when a task becomes ready, which resource it can use, and whether it fits the current gap.

Given these DAGs, Tessera uses an event-driven list scheduler (Algorithm 1) to minimize the pair's makespan on two resources (Comp, Comm). At each event, SELECT applies two heuristics. (i) *Backbone-first alignment*: backbone tasks take precedence, and a gap-fit rule picks the task whose duration is closest to the remaining window on the complementary resource. (ii) *Conditional deferral*: movable tasks are placed when they fit the current gap without extending the pair's makespan. SHOULDDEFER defers a movable task that would extend the pair's makespan. Once backbone placement is complete, deferred movable tasks are backfilled into residual slots within their lifetimes using a best-fit rule; tasks with no suitable slot are placed at the tail.

Tessera executes the synthesized schedule on a *reference device group*, a dedicated device set configured with the same TP/EP topology as the target rank, to measure *ground-truth post-overlap cost*. Physical profiling is necessary because real hardware interference causes the true overlapped cost to deviate from analytical estimates (§5). To bound overhead, Tessera profiles each distinct overlap pair only once: when multiple edge-candidate pairs yield identical chunk specifications, pass directions, and device topologies, their profiles are shared. Results are cached by device-mesh class and chunk specification for reuse across ranks and replicas with matching GPU type and interconnect pattern.

3.1.3 Step 3: Overlap-Aware Partition Selection

With all post-overlap costs $T_{e,c,d}$ profiled, partition selection reduces to a graph labeling problem. Pipeline iteration time depends on the full critical path through the fixed schedule. Under the fixed template, Tessera targets the steady-state overlap opportunities that recur across microbatches: a selected edge with high profiled post-overlap cost would repeatedly expose a straggler. Tessera therefore uses the maximum selected post-overlap edge cost as a surrogate objective for partition selection. Formally, binary variable $y_{s,c}$ selects candidate c for node s; $z_{e,c,d}$ activates the candidate pair selected on edge $e{=}(s,t)$. Each node selects exactly one candidate, and each edge variable is consistent with its endpoint selections:

$$
\begin{aligned}
\min \quad & T_{\text{dom}} \\
\text{s.t.} \quad & \sum_{c \in C_s} y_{s,c} = 1 && \forall s \in \mathcal{S}, \\
& \sum_{d \in C_t} z_{e,c,d} = y_{s,c} && \forall e{=}(s,t) \in \mathcal{E},\, c \in C_s, \\
& \sum_{c \in C_s} z_{e,c,d} = y_{t,d} && \forall e{=}(s,t) \in \mathcal{E},\, d \in C_t, \\
& T_{\text{dom}} \geq \textstyle\sum_{c,d} T_{e,c,d}\, z_{e,c,d} && \forall e \in \mathcal{E}, \\
& y_{s,c},\ z_{e,c,d} \in \{0,1\},
\end{aligned}
\tag{1}
$$

$T_{e,c,d}$ is the profiled *post-overlap* cost of candidate pair (c,d) on edge e. The bottleneck constraint charges each overlap edge by the cost of its selected pair. The full solver additionally enforces contiguous non-overlapping layer ranges, device-topology compatibility, and per-rank memory capacity constraints.

Complexity and Profile-Guided Pruning. By decoupling the search into stage-level variables and linking adjacent stages via consistency constraints, this MILP formulation reduces the binary variable count from $O(N \cdot K^{vp})$ in rank-level enumeration to $O(N \cdot vp \cdot K^2)$, where N is the number of pipeline ranks, vp is the number of virtual stages per rank, and K is the candidate count per stage; bottleneck constraints scale as $|\mathcal{E}|$, one per overlap edge. To further accelerate the solver, Tessera applies safe profile-guided pruning. The bottleneck post-overlap cost of the baseline partition, T_{base}, serves as an upper bound on T_{dom}: any candidate pair whose profiled cost exceeds T_{base} is discarded, since selecting it cannot improve upon the baseline. If a stage candidate c loses all valid pairs on its incident edges, it is removed from the search space. In practice, this significantly reduces the MILP size while preserving all solutions with $T_{\text{dom}} \leq T_{\text{base}}$.

Plan Synthesis. The selected node labels on the same physical rank compose that rank's assignment. The solver's output is materialized into the final execution plan by instantiating the pipeline schedule template and substituting each edge with the fine-grained overlap schedule synthesized in Step 2.

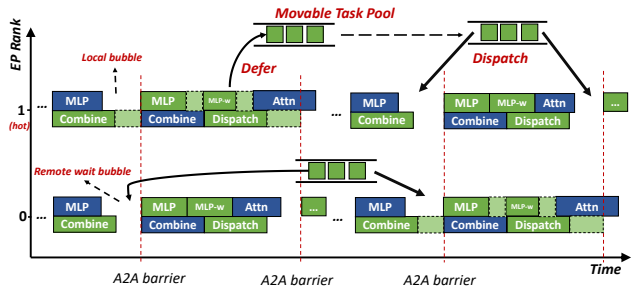


Figure 8: Dynamic scheduling of movable tasks into bubbles.

3.2 Dynamic Bubble Optimization

Static planning assigns each movable task a default execution position within the overlap schedule. At runtime, MoE routing variation changes task durations, shifting the size and location of idle slots in ways that no static plan can predict. The *Dynamic Bubble Optimizer* may defer eligible movable tasks and dispatch them into predicted idle slots, recovering throughput that static planning leaves unexploited.

3.2.1 Bubble Prediction and Slot Sizing

Runtime variability is primarily driven by token routing decisions. To detect routing-induced bubbles before they occur, Tessera reuses routing metadata that becomes available before expert dispatch. Within each EP group, ranks exchange only per-expert token counts via a lightweight collective that piggybacks on the existing MoE dispatch path. Because the exchange is scoped to the EP group and carries scalar counts rather than tensor payloads, it introduces no cluster-wide synchronization. An imbalance monitoring mechanism uses these counts to estimate the workload assigned to each rank. Deviations from the profiled static schedule predict where and how large idle slots will form.

Tessera pre-annotates target slot locations during static planning. Whether a slot materializes, and how large it is, is determined at runtime. As illustrated in Fig. 8, load imbalance typically prolongs specific operations (e.g., MLP on hot experts), inducing local idle time and cross-device wait states.

To capture these routing-induced gaps, Tessera locates structural boundaries directly after MoE computation tasks within the overlap schedule (§3.1.2). The two chunks in an overlap pair can originate from different microbatches with independent token distributions; their latencies can shift in opposite directions, so the idle window at these boundaries fluctuates widely across iterations. Pre-annotating these locations provides Tessera with effective insertion points for movable tasks.

Target Slot Sizing. To ensure that task injection does not stall the training pipeline, Tessera must estimate slot sizes asynchronously. Token distribution is resolved during the forward dispatch, so load information becomes available several steps before the target slot. The sizing calculation thus overlaps with useful computation. Using the observed token distribution and the offline-profiled static schedule, Tessera estimates the fillable idle window b_β at each pre-annotated target slot β. Non-positive windows are skipped.

Algorithm 2 Dynamic Bubble-Filling

Input: Token distribution $\mathbf{T}$, offline profile $\mathcal{P}$, movable task pool Q

▷ Retrieve pre-annotated target slots
$\mathcal{B} \leftarrow$ GETTARGETSLOTS() ▷ From static plan
▷ Fill predicted slots from movable task pool
for each target slot $\beta \in \mathcal{B}$ **do**
  $b_\beta \leftarrow$ ESTIMATEBUBBLESIZE($\mathbf{T}, \mathcal{P}, \beta$) ▷ Asynchronous
  if $b_\beta > 0$ **then**
    $r \leftarrow \beta$.rank, $t_{\text{start}} \leftarrow \beta$.time
    $\mathcal{C} \leftarrow \emptyset$
    for each task $\sigma \in Q$ **do**
      if σ is ready on r **and** σ.duration $\le b_\beta$ **and** $t_{\text{start}} +$ σ.duration $\le \sigma$.deadline **then**
        $s \leftarrow$ COMPUTESCORE(σ, b_β)
        $\mathcal{C} \leftarrow \mathcal{C} \cup \{(\sigma, s)\}$
    if $\mathcal{C} \neq \emptyset$ **then**
      $\sigma^* \leftarrow \arg\max_{(\sigma,s)\in\mathcal{C}} s$
      SCHEDULE(σ^*, on r, start at t_{start})
      Remove σ^* from Q

3.2.2 Movable Task Injection

Movable Task Pool. Tessera fills predicted slots from a pool of movable tasks whose deadline slack (the margin between readiness and deadline minus execution time) is positive. It organizes these tasks into priority queues ordered by deadline urgency. For example, `Wgrad` is enqueued as soon as its preceding backward computation completes, and is assigned a hard deadline corresponding to the iteration end. We observe that this pool is typically well-provisioned in production, as `Wgrad` tasks naturally accumulate off the critical path during load imbalance events. The pool is bounded by a configurable per-GPU capacity and the available memory headroom. Once either bound is reached, newly ready movable tasks are kept at their original positions rather than deferred.

Guaranteed Execution. If a movable task's deadline slack approaches zero without a suitable target slot being found, the runtime preemptively injects it into the main stream, prioritizing correctness over optimization.

Bubble-Filling Policy. With the fillable window b_β resolved asynchronously, Tessera triggers the injection logic immediately prior to launching the overlapped tasks. The dynamic scheduler employs a greedy heuristic (Algorithm 2) to scan the pool and select the feasible task σ with the highest score $S(\sigma, b_\beta)$, which prioritizes tight duration fit and small deadline slack. If no suitable task is found, the slot remains empty. Because the pool search space is restricted and each b_β is computed asynchronously before its target

slot, the combined overhead of slot sizing and task selection is negligible ($< 10\,\mu$s). Furthermore, by triggering this logic asynchronously ahead of the sparsely pre-annotated target slots rather than after every task, this mechanism minimizes the interference of dynamic scheduling with the continuous pipeline execution.

4 Implementation

Tessera is implemented as a standalone library (11,000 lines of Python and 2,000 lines of C++) integrated with Megatron-LM [22] (as a host framework).

Plan-Agnostic Execution Engine. To execute the diverse overlap patterns required by asymmetric overlap pairs without invasive modifications to the host framework's imperative loops, Tessera implements a *Plan-Agnostic Execution Engine* in C++. This engine decouples schedule definition from execution by interpreting the static plan as a specification rather than hard-coded control flow. It coordinates execution via a lock-free finite-state machine (FSM). The main Forward and Backward Torch threads drive their respective tasks, pausing at designated yield points to synchronize with the engine's global state and achieve the planned inter-microbatch overlap. Concurrently, an additional background thread processes movable tasks registered with the engine, such as `Wgrad`, according to the execution plan. This architecture enables the runtime to execute diverse pipeline execution plans generated by the static planner without relying on hard-coded pipeline schedule control flow.

Customizing Task Boundaries and Overlap Schedules. Instead of requiring developers to refactor MoE model definitions into isolated task functions, Tessera exposes task boundaries through lightweight execution probes inserted in the existing forward and backward code. The `advance(TaskName)` API marks host-side probe points where the execution engine may block or release the current forward/backward execution thread, slicing the continuous execution into named tasks without moving tensor computation out of the host framework. For movable tasks that lie off the backbone, such as `Wgrad`, Tessera uses `register_task()` to expose the callable after its dependencies become available. Listing 1 illustrates this pattern inside a schedulable MoE operator.

By default, the system automatically constructs overlap schedules from these tasks using its built-in scheduler (§3.1.2). Thanks to Tessera's *Plan-Agnostic Execution Engine*, users can alternatively specify custom interleaving strategies via a YAML configuration file. As shown in the second half of Listing 1, this interface allows experts to define fine-grained, device-specific execution sequences for interleaved streams—enabling domain-specific overlap optimizations when default heuristics are insufficient.

User Interface. Tessera exposes a declarative API that separates *schedule control* from the *tensor computation* managed by the host framework. Users invoke `build_execution_plan` to trigger the Static Planner, which ingests the model structure, applies task decomposition rules, and synthesizes the execution plan. `run` then accepts this plan and hands control to the Pipeline Execution Engine. To ensure correctness within complex training loops, Tessera utilizes standardized hooks (e.g., `pre_fwd`, `post_fwd`, `post_iter`) to inject framework-specific logic (e.g., loss aggregation, gradient reduction) into the execution flow, keeping such logic localized in hooks.

Listing 1: Customizing task boundaries and overlap schedules.

```
# Schematic: Tessera probe API usage (simplified)
class SchedulableMoEFunction(torch.autograd.Function):
    ...
    @staticmethod
    def forward(ctx, hidden_states, expert_weights):
        ...
        tessera.scheduler.advance("F_DISP")
        dispatched_tokens = dispatch(hidden_states)
        tessera.scheduler.advance("F_MLP")
        expert_outputs = compute_experts(dispatched_tokens, ...)
        ...
    @staticmethod
    def backward(ctx, grad_outputs):
        ...
        tessera.scheduler.advance("B_COMBINE")
        grad_combined = combine_logic_bwd(grad_outputs)
        tessera.scheduler.advance("B_MLP")
        grad_inputs = compute_activation_grad(grad_combined, ...)
        tessera.scheduler.register_task(
            task_name="B_WGRAD",
            func=compute_weight_grad,
            args=(grad_combined, hidden_states)
        )
        ...
# User-specified overlap schedule (overlap.yaml)
overlap_plans:
  - chunk_pair:
      fwd_chunk: [Layer0, Layer1]
      bwd_chunk: [Layer0, Layer1]
    pattern:
      - B_COMB:L1           # -- overlap: bwd L1 comm
      - F_ATTN:L0           # --  with fwd L0 compute
      - F_DISP:L0
      - B_MLP:L1
      - F_COMB:L0
      - B_WGRAD:L1
      - ...
```

5 Engineering Experience and Discussion

We deployed Tessera in the production pre-training of the Qwen3 and Qwen3-Next model families, scaling beyond 10,000 NVIDIA Hopper GPUs. Scaling to this magnitude exposes fracture points where theoretical scheduling abstractions break down under physical realities. In this section, we discuss the operational challenges encountered and the engineering solutions adopted to sustain high efficiency at scale.

Intra-microbatch vs. Inter-microbatch Overlap. We evaluated *intra-microbatch* overlap (partitioning input tokens to hide MoE dispatch) against Tessera's *inter-microbatch* approach. While intra-microbatch overlap offers theoretical latency hiding, we found it operationally fragile at scale. Par-

titioning along the sequence dimension fragments GEMM operations, reducing arithmetic intensity below the threshold where compute throughput dominates. Aggressive kernel fusion (e.g., Comet [39]) can recover efficiency, but in our production stack the resulting coupling between MoE kernels and specific communication backends complicates independent backend upgrades. In contrast, Tessera's inter-microbatch approach maintains full GEMM computational intensity and backend modularity.

Theoretical Overlap vs. Silicon Reality. Static overlap schedulers estimate an overlap pair's makespan from the task DAG, using task durations, resource types, and dependencies to determine the expected computation–communication overlap. This theoretical estimate is useful for synthesizing candidate schedules, but large-scale production deployments reveal a small, non-uniform gap between the idealized schedule and silicon execution [4, 10]. Across Qwen3-Next pre-training workloads, the theoretical cost underestimates measured overlapped execution in most cases, with an average deviation of around 5%. One major source is SM contention: EP communication kernels reserve ∼20 SMs for All-to-All dispatch and combine, causing a stable 10–20% slowdown when they overlap with Attention or MoE-MLP kernels. We also observe that each task is a coarse scheduling unit rather than a single kernel (e.g., a communication-dominant task may contain small compute kernels), which can leave residual interference, or even serialization, that prevents the idealized schedule from achieving perfect overlap.

Granularity of Cost Profiling. Chunk-pair profiling is affordable for our production workloads: even for trillion-scale models, profiling completes within one hour because repeated model structure bounds the number of distinct overlap pairs. We also explored a lower-cost alternative, *primitive-level profiling*, which profiles reusable overlap patterns between pairs of task primitives and composes the measured interactions to estimate a full chunk-pair timeline. Compared with the purely theoretical estimate above, primitive-level profiling reduces average error by capturing stable pairwise effects such as SM contention. However, its residual error remains uneven across overlap pairs, reaching up to 15% in tail cases. This tail behavior is problematic for partition selection because the MILP relies on relative cost comparisons among candidate stage assignments; a few misestimated overlap pairs can change the selected bottleneck edge or partition boundary. The residual gap arises from cross-primitive effects, including relay overlaps across consecutive kernels, cumulative execution state in caches and communication libraries, and host-side launch overhead. We therefore use chunk-pair profiling as the default, while keeping primitive-level profiling as a lower-cost fallback.

Infrastructure Jitter. Scaling beyond 10,000 GPUs reveals that infrastructure noise is a persistent source of throughput instability. Network congestion and switch contention cause stochastic variation in pipeline communication latency, both across iterations and across ranks. While the Static Planner addresses structural heterogeneity, these transient fluctuations require runtime adaptation. We have explored extending DBO to absorb infrastructure jitter by reusing its bubble-filling mechanism. Leveraging the observation that jitter often exhibits short-term temporal correlation, a history-based predictor (moving average of the last 10 iterations) estimates upcoming bubble sizes, and per-rank injection fills these transient gaps with locally pooled movable tasks. This approach is architecturally feasible since infrastructure-induced bubbles enter the same prediction path as routing-induced ones. Systematic validation of this extension under controlled jitter conditions remains future work.

Memory Pressure Trade-off. While the *Dynamic Bubble Optimizer* improves throughput by delaying movable tasks to fill idle pipeline slots, this strategy introduces a subtle memory trade-off: delaying a `Wgrad` task prolongs the liveness of its associated activation and gradient tensors, increasing peak memory consumption. In our production runs, aggressive bubble filling occasionally triggered out-of-memory (OOM) errors on memory-constrained workloads. In practice, we rely on the bounded movable-task pool described in §3.2.2, configuring its per-GPU cap from observed memory headroom. This cap limits how many deferred tasks can keep activations and gradients live at once. When memory headroom is tight, Tessera can also represent memory-management operations, including activation offloading and recomputation, as *movable tasks* and schedule them into suitable pipeline bubbles.

Decoupling System Constraints via Padding. A major friction point in large-scale training is the conflict between hyperparameter tuning and system constraints. The standard interleaved 1F1B schedule [22] theoretically mandates that the number of microbatches (M) per iteration must be perfectly divisible by the pipeline parallel size (K). In production, however, M is dictated by the global batch size required for model convergence, while K is constrained by device memory limits. Forcing $M\%K = 0$ often requires suboptimal batch size adjustments that degrade model quality. Tessera resolves this by decoupling these constraints via *microbatch padding*. We inject `shadow` actions, null operations that skip computation but perform necessary pipeline communication (sending pseudo-tensors), to satisfy structural dependencies without altering the mathematical equivalence of the batch. Shadow actions are excluded from loss accumulation, gradient scaling, and optimizer accounting. Leveraging Tessera's configuration interface, this padding is achieved by simply specifying a compliant plan file, avoiding invasive code changes. Empirically, this mechanism introduces negligible overhead (∼0.5%) while maximizing configuration flexibility.

Table 1: **Specification of Qwen Models in Experimentation.** Model Scale: Small <100B, Medium 100–500B, Large 500B–1T, Trillion >1T.

Models	TopK / #Experts	Model Scale	Publicly Available
Qwen3-235B	8 / 128	Medium	Yes
Qwen3-Next-80B	10 / 512	Small	Yes
Qwen3-L	8 / 128	Large	No
Qwen3-XL	8 / 128	Trillion	No
Qwen3-Next-M	10 / 512	Medium	No
Qwen3-Next-L	10 / 512	Trillion	No
Qwen3-Next-XL	10 / 512	Trillion	No

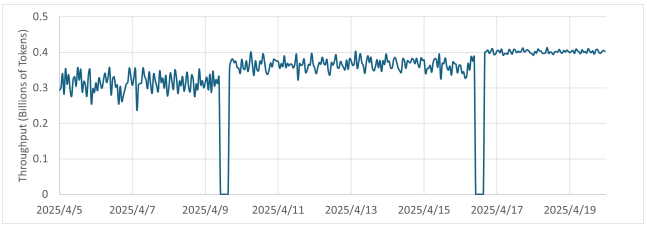


Figure 9: **Phased Impact of Tessera Deployment.** A 2-week production trace (April 5 – April 19, 2025) from the 8,192-GPU Qwen3-XL training job. The first step-change (April 9) marks the deployment of Tessera's *Static Planner*, which resolves structural bottlenecks. The second jump (April 16) marks the activation of the *Dynamic Bubble Optimizer*, which absorbs runtime stochasticity to stabilize throughput at peak levels.

6 Evaluations

6.1 Experimental Setup

We conduct experiments on Alibaba's production cluster, comprising NVIDIA Hopper GPU servers (each equipped with 8 GPUs) interconnected via a high-performance RoCE network. Our production evaluation focuses on pre-training the **Qwen3** and **Qwen3-Next** model families. For the controlled open-source comparison (§6.3), we additionally evaluate DeepSeek-V3 and Nemotron-3 Super. All experiments use a mixture of proprietary internal datasets and open-source datasets. We measure system performance based on Model FLOPs Utilization (MFU), end-to-end throughput, and planning overhead. For the primary production results in Table 2, each workload is run with and without Tessera under the same cluster configuration and hyperparameters.

Baselines. We benchmark Tessera against two baselines that serve complementary evaluation purposes: (1) **Internal Optimized Baseline (§6.2):** For production-scale evaluation on clusters of up to 12,288 GPUs, we compare against Alibaba's internal Megatron-LM fork, the production training stack used for the same Qwen3-family workloads before enabling Tessera. This baseline already includes interleaved 1F1B scheduling, native communication overlap, and production-tuned FLOPs-balanced stage partitioning for the same cluster configuration. This setup isolates the incremental benefit of Tessera's optimizations on top of an optimized production stack. (2) **Megatron-Core MoE (v0.16.1) [36]:** For controlled, reproducible experiments on a 256-GPU cluster, we compare against Megatron-Core MoE using public per-model recipes from Megatron-MoE-ModelZoo (1ca7da9) [35] and Megatron-Bridge v0.4.1 [23]. These recipes enable Megatron-Core's built-in MoE optimizations, including EP communication overlap.

6.2 End-to-End Production Performance

We first present primary results on the massive-scale cluster. As standard open-source frameworks require significant customization at this scale, we benchmark against the *Internal Optimized Baseline* defined in §6.1—a production-hardened Megatron-LM tuned by domain experts. Note that, Tessera's planning scope does not grow with the full cluster size: Tessera generates one execution plan for a model-parallel device mesh and replicates it across data-parallel ranks, so scaling increases the number of plan instances rather than the planning problem size. DBO likewise operates within each EP group with no cross-replica coordination.

Performance Consistency. Table 2 summarizes the gains. Tessera consistently improves MFU by 20%–33% across all workloads. The **standard MoE (Qwen3 family)** repeats the same Attention+MoE block across layers, achieving high baseline MFU (29.7%–32.0%) due to strong arithmetic intensity; the trillion-parameter Qwen3-XL further improves baseline MFU by amortizing communication overheads. For these models, the primary gains come from mechanisms that do not require layer-type diversity: fine-grained inter-microbatch overlap hides A2A communication, while DBO fills routing- and pipeline-induced bubbles. The overlap-aware partitioner still contributes because production partitions include non-uniform boundary chunks (e.g., embedding/output-head chunks), and boundary shifts can change the post-overlap costs of the resulting stage pairs. The **Qwen3-Next family** interleaves GDN and full softmax-attention layers, producing structurally heterogeneous chunks. It employs an extremely sparse MoE architecture (TopK=10/512). As newly deployed models lacking mature full-stack optimizations, such as kernel fusion or communication coalescing, their baseline MFU is significantly lower (15.9%–19.6%), reflecting reduced compute intensity and higher system integration overhead under extreme sparsity. Despite this, Tessera delivers 20.0%–32.8% MFU gains, demonstrating robustness across model scales, cluster sizes, and deployment maturity levels.

A Phased Deployment. To disentangle the contributions of Tessera's components, Fig. 9 illustrates a 2-week "hot up-

Table 2: **Production Performance Gains across Diverse Workloads.** Tessera demonstrates consistent MFU improvements across controlled production-scale pre-training workloads, spanning model scales, MoE sparsities, and cluster sizes from 4,096 to 12,288 GPUs.

Workload Task	TopK / #Experts	Model Scale	Cluster Scale	Baseline MFU	Tessera MFU	Speedup
Qwen3-L	8 / 128	Large	8,192 GPUs	29.7%	**36.3%**	+22.0%
Qwen3-XL	8 / 128	Trillion	8,192 GPUs	32.0%	**39.0%**	+21.8%
Qwen3-Next-M	10 / 512	Medium	4,096 GPUs	16.7%	**20.0%**	+20.0%
Qwen3-Next-L	10 / 512	Trillion	8,192 GPUs	19.6%	**24.0%**	+22.5%
Qwen3-Next-XL	10 / 512	Trillion	12,288 GPUs	15.9%	**21.1%**	+32.8%

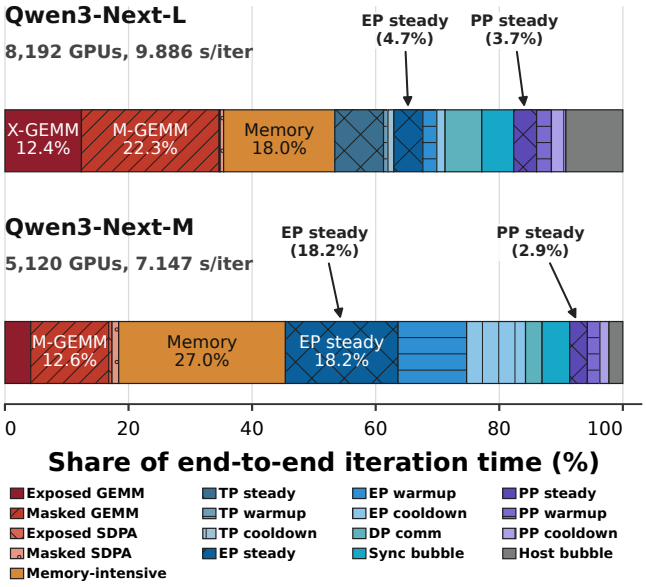


Figure 10: **Iteration-time breakdown for two production Qwen3-Next workloads.** Steady, warmup, and cooldown refer to pipeline phases.

grade" of the Qwen3-XL training. **Phase 1 (Static Planner):** On April 9, replacing the baseline with Tessera's static plan yielded an immediate ∼13% throughput boost. Decomposition reveals this gain came from (1) *Load Balancing* (∼9%), where our overlap-aware partition balanced post-overlap stage costs, and (2) *Latency Hiding*, masking the All-to-All communication overhead. **Phase 2 (Dynamic Optimizer):** On April 16, enabling dynamic bubble filling elevated MFU to 39.0%. As evidenced by the flattened trace, dynamic task injection reduced production throughput variability after DBO activation.

Production Case Study. To understand when Tessera's overlap gains are most effective and where residual bottlenecks remain, we further decompose iteration time for two Qwen3-Next production workloads in Fig. 10. In both cases, Tessera's overlap-aware partitioning compresses steady-phase PP bubbles to roughly 3% of iteration time, confirming effective load balancing across pipeline stages. Yet the two workloads exhibit sharply different overlap efficiency. On the trillion-scale Qwen3-Next-L (8,192 GPUs), Tessera compresses exposed EP communication in the pipeline steady phase to 4.7% of iteration time. Including warmup and cooldown, 73% of EP communication is hidden behind computation, leaving exposed EP at only 8.3% of iteration time. On a separate 5,120-GPU Qwen3-Next-M run (distinct from the 4,096-GPU production result in Table 2), however, steady-phase exposed EP alone reaches 18.2% of iteration time. Including warmup and cooldown, only 26% of EP communication is hidden, with total exposed EP reaching 38.9% of iteration time. The root cause is compute intensity. Qwen3-Next-M has fewer parameters per expert, producing shorter GEMM durations relative to A2A transfer time and leaving each overlap pair with insufficient computation to hide the A2A latency. This contrast shows that Tessera's overlap gains scale with the workload's compute-to-communication ratio.

Gap to Ideal Execution. Even in the favorable regime (the top bar of Fig. 10), 27% of EP communication remains exposed (819 ms), over 40% of which originates from warmup and cooldown phases. Beyond exposed EP, PP bubbles contribute another 8.5% (835 ms), split between residual stage imbalance (366 ms) and structural warmup/cooldown cost (469 ms) inherent to interleaved 1F1B. Together, exposed EP communication and pipeline idle time account for 17% of iteration time, which we target in ongoing optimization.

Bitwise Equivalence. Tessera modifies only *when* and *where* tasks execute on the device timeline; the computation graph, operator semantics, and reduction order are unchanged. Gradient equivalence therefore holds by construction. We verify this empirically on a trillion-scale Qwen3-Next workload by comparing runs with and without Tessera's optimizations under deterministic mode. The two runs produce bit-identical loss trajectories. This confirms that Tessera's scheduling transformations introduce zero numerical divergence, and the system has since sustained months of production pre-training without observed quality regression.

6.3 Comparison with State-of-the-Art

To complement the production evaluation against our internal Megatron baseline (§6.2), we compare Tessera against **Megatron-Core MoE** [36] on a 256-GPU cluster. We evaluate three open-source MoE models: (i) **Qwen3-235B** [37],

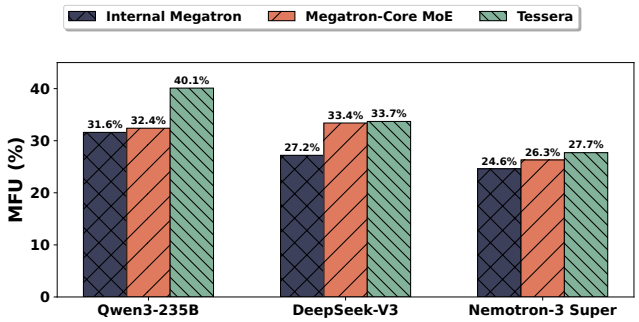


Figure 11: MFU comparison among Internal Megatron, Megatron-Core MoE, and Tessera on a 256-GPU cluster. Tessera remains competitive with publicly available Megatron-Core MoE recipes on the non-Qwen workloads while consistently improving over the internal Megatron baseline across all three models.

a standard Transformer MoE (128 experts, top-8, sequence length 4K); (ii) **DeepSeek-V3** [3], which combines a dense prefix, compressed MLA attention, an MTP head, and a sparse MoE (256 experts, top-8, sequence length 4K); and (iii) **Nemotron-3 Super** [24], a hybrid Mamba/Attention model with latent MoE and MTP layers (512 experts, top-22, sequence length 8K). For Megatron-Core MoE, we follow the per-model recipes from the public Megatron-MoE-ModelZoo [35] and Megatron-Bridge [23].

As shown in Fig. 11, Tessera improves MFU over the internal Megatron baseline by 1.27×, 1.24×, and 1.13× on Qwen3-235B, DeepSeek-V3, and Nemotron-3 Super, respectively, showing that Tessera's optimizations generalize beyond the Qwen3 family to architecturally diverse MoE workloads. Against the recipe-tuned Megatron-Core MoE baseline, Tessera improves MFU by 1.24× on Qwen3-235B and achieves comparable MFU on non-Qwen models.

We attribute Tessera's gains to three mechanisms: (i) the overlap scheduler and DBO jointly mask exposed EP communication; (ii) the overlap-aware partitioner reduces stage-level load imbalance; and (iii) DBO overlaps deferred `Wgrad` with PP send/receive, shrinking the pipeline bubble. On Qwen3-235B, Megatron-Core MoE's A2A overlap removes most exposed EP communication but leaves PP wait intact. Relative to the internal Megatron baseline, Tessera cuts exposed EP communication by ∼75% and PP wait by ∼90%. On DeepSeek-V3, both Tessera and Megatron-Core MoE's hand-tuned recipe mask over 70% of EP communication and yield balanced PP stages; Tessera's DBO further overlaps deferred `Wgrad` with PP send/receive, reaching slightly higher end-to-end MFU. On Nemotron-3 Super, Megatron-Core MoE's A2A-overlap implementation relies on a hand-crafted Transformer-layer decomposition and currently handles at most one MTP layer. EP communication outside this supported pattern therefore remains unoverlapped. Tessera reduces exposed EP communication by 40% and the pipeline bubble by 30%, yielding higher end-to-end MFU than Megatron-Core MoE.

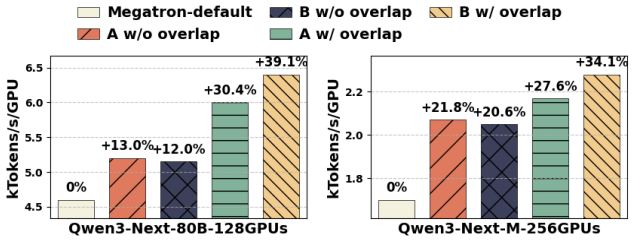


Figure 12: Impact of co-optimizing partitioning and overlap scheduling.

6.4 Co-optimizing Partitioning and Overlap

We evaluate the impact of co-optimizing overlap scheduling and partitioning through two experiments: training Qwen3-Next-80B on 128 GPUs and Qwen3-Next-M on 256 GPUs. Both setups utilize an interleaved 1F1B schedule with a micro-batch size of 1 per GPU, a pipeline-parallel size of 4 with an interleave size of 2, and an expert-parallel size of 32. To support a sequence length of 64K, we enable Context Parallelism. For a controlled comparison of static execution plans, we employ a mock router that uniformly distributes tokens and disable DBO, isolating static plan efficiency from dynamic runtime variance.

We establish Megatron-LM's native uniform partition as the baseline. Against this, we compare two alternative strategies, each evaluated with and without Tessera's fine-grained overlap scheduling: (1) **Partition A**: a latency-guided partition balanced by serial execution time; and (2) **Partition B**: Tessera's overlap-aware partition. As shown in Fig. 12, the two partitions exhibit markedly different sensitivities to inter-microbatch overlap. When overlap scheduling is disabled, Partition A achieves slightly higher end-to-end throughput than Partition B, as indicated by the A w/o overlap and B w/o overlap bars. However, once overlap scheduling is enabled, the relative advantage shifts: Partition B attains significantly greater overlap gains and ultimately delivers the highest end-to-end throughput (B w/ overlap vs. A w/ overlap). These results confirm that partitioning decisions must be co-optimized with overlap scheduling rather than optimized in isolation.

6.5 Overlap Scheduling

We evaluate overlap scheduling on the candidate overlap pairs enumerated by Static Planning, before partition selection. For Qwen3-Next-M at sequence length 4K, the EP32 and EP8 regimes produce 690 and 536 scheduling instances. We compare four strategies: FIFO schedules tasks in topological order without priority or alignment; +ALIGN adds backbone-priority and gap-fit alignment (§3.1.2); +DEFER adds conditional deferral of movable `Wgrad` tasks; and ILP solves

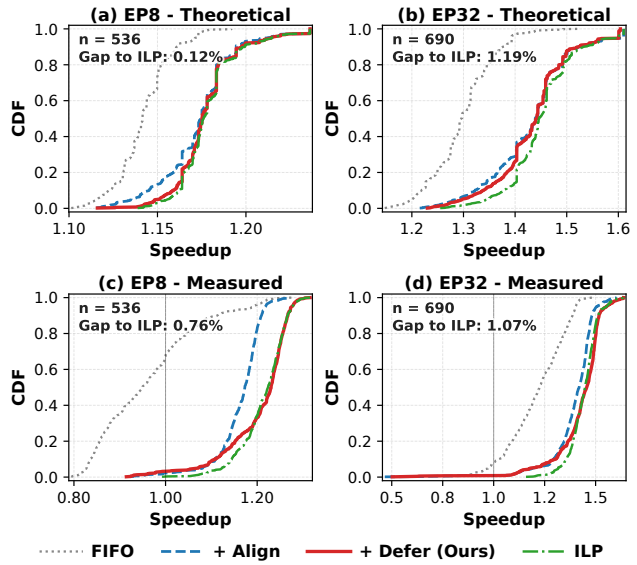


Figure 13: CDF of per-instance speedup over serial execution for four overlap schedulers. Top row: analytical makespan; bottom row: measured post-overlap cost on hardware. +DEFER (red) closely tracks the ILP reference (green) across the full distribution at both EP widths.

the resource-constrained DAG with CBC [5] under a 300-s per-instance budget.

As shown in Fig. 13, EP32 exposes larger overlap gains than EP8 because EP communication accounts for a greater share of each pair's serial cost. In both cases, the measured CDFs closely follow the analytical predictions, with the offset consistent with the theory-to-silicon gap discussed in §5. The curves also show that +ALIGN alone captures most of the gain over FIFO. +DEFER's mean gain over +ALIGN is modest, but some instances gain over 10%. Taken together, the two heuristics close to within 1.19%/0.12% of ILP on EP32/EP8 in analytical makespan, and 1.07%/0.76% on measured post-overlap cost.

Quality vs. solving-time trade-off. Solving all instances with CBC takes hours to days. Per-instance solve time is long-tailed: many hit the 300-s budget without a proven optimum. Tessera's event-driven scheduler finishes in under a minute, staying within $\sim 1\%$ of ILP on profiled post-overlap cost. We therefore adopt the heuristic as the production default.

6.6 Dynamic Bubble Optimization

We evaluate DBO's speedup, overhead, and memory cost on Qwen3-Next-M (256 GPUs, 4K sequence length). To isolate DBO's contribution from static planning decisions, we run DBO on top of two static plans. The *Baseline Partition* balances serial FLOPs across pipeline stages; the *Overlap-Aware Partition* is selected by Tessera's partitioner using profiled post-overlap costs. Following the bounded-pool policy in §3.2.2, we cap the per-GPU movable-task pool at 8. For each plan we report three configurations: (i) *static*, DBO disabled; (ii) *w/ DBO*, full DBO active; and (iii) *always-keep*, which runs the full DBO monitoring, slot-sizing, and injection-decision paths but keeps every movable task at its original position instead of deferring it. The always-keep row isolates the steady-state overhead of DBO without the benefit of task reordering.

Table 3: Performance and peak-memory cost of Tessera's Dynamic Bubble Optimizer (DBO) on Qwen3-Next-M. Peak mem. denotes the maximum allocated memory normalized by the per-GPU memory budget.

Metric	Baseline Partition	Overlap-Aware Partition
Iter-time (s), static	8.28	6.98
Iter-time (s), w/ DBO	7.83	6.67
Iter-time (s), always-keep	8.34	7.06
Peak mem., static	69.3%	67.4%
Peak mem., w/ DBO	72.9%	70.0%

Table 3 summarizes the results. DBO cuts iteration time by 5.4% on Baseline and 4.4% on Overlap-Aware. The larger gain on Baseline is expected: serial-balanced partitions leave larger PP wait, which DBO overlaps with deferred `Wgrad` tasks; overlap-aware partitioning already shrinks this wait during static planning. Always-keep adds about 1% over the corresponding static run, so the monitor and injection paths are lightweight on their own. On the memory side, DBO increases peak memory by 2–4 percentage points of the per-GPU memory budget, since deferred movable tasks prolong the liveness of their input activations and gradients. This increase is consistent with the bounded per-GPU movable-task pool. The capacity can be raised for more bubble-filling, or lowered to reduce peak memory.

Production-Scale Validation. Two production deployments confirm DBO's effectiveness beyond the 256-GPU ablation. On the 8,192-GPU Qwen3-XL job (§6.2, Fig. 9), end-to-end MFU reached 39.0% after DBO activation. On a 6,144-GPU Qwen3-L run (distinct from Table 2), DBO alone reduced PP bubble time by 641 ms, yielding a 3.4% throughput gain. For production-scale overhead, DBO runs independently within each EP group, so scaling to more GPUs increases the number of independent instances without adding cross-group coordination.

6.7 Planning Overhead Analysis

Tessera's static planning is offline and consists of profiling distinct overlap pairs and solving the partitioning MILP. We evaluate on Qwen3-Next-M and Qwen3-Next-L under an 8,192-GPU target configuration with PP8-C2 and PP8-C4, denoting 2 or 4 chunks per pipeline rank under pipeline parallel

size 8. Profiling runs on a 64-GPU reference device group. Across sequence lengths from 4K to 128K, repeated model structure bounds the distinct overlap-pair count to 237–327 under PP8-C4 and 575–637 under PP8-C2. Measured profiling time grows with sequence length and peaks at about 3,050 seconds at 128K for the trillion-scale PP8-C2 case, still below one hour.

Table 4: **MILP problem scale and solve time.** Pruning removes candidate pairs whose post-overlap cost exceeds T_{base}.

Configuration	#Decision Variables	#After Pruning	Solve Time
M-PP8-C2	9,358	1,782	0.34s
L-PP8-C2	23,180	6,993	1.51s
M-PP8-C4	16,192	5,505	0.82s
L-PP8-C4	35,904	12,207	4.22s

Table 4 reports the MILP scale and solve time. The edge-based formulation introduces stage-selection and edge-pair activation variables, scaling as $O(S{\cdot}K + E{\cdot}K^2)$ and avoiding K^{vp} rank-level enumeration. Profile-guided pruning (§3.1.3) removes all candidate pairs whose post-overlap cost exceeds T_{base}, reducing the effective problem size by roughly 3–5×. All tested configurations are solved within five seconds on a 64-core server.

7 Related Work

Pipeline Model Parallelism. Pipeline parallelism was established by GPipe [7], PipeDream [21], and Interleaved 1F1B [22]. It remains a core scaling primitive in recent large-scale MoE training systems and workloads [30, 32, 37]. Specialized variants target bidirectional execution (Chimera [13]), bubble minimization (Zero Bubble [25]), sequence-level partitioning (Seq1F1B [27]), activation checkpointing (Mario [20]), and memory efficiency (MEPipe [29]). These systems largely rely on static templates that lack the expressivity for the misaligned overlap patterns of heterogeneous MoE layers; Tessera instead decomposes operations into fine-grained task graphs for flexible interleavings.

Communication-Computation Overlap. Flux [1] and Comet [39] employ *intra-microbatch overlap*, fusing MoE communication with computation, but often require invasive kernel fusion that hinders modularity. For *inter-microbatch overlap*, DualPipe [3] orchestrates microbatch execution to hide communication, while WeiPipe [14] overlaps weight transfers. Inter-microbatch approaches still rely on manually designed or workload-specific schedules. Tessera automates inter-microbatch overlap discovery through global planning and task decomposition.

Automated Model Partitioning. Symbolic partitioners such as Alpa [40], Tessel [15], nnScaler [16], and Colossal-AI [12] use analytical cost models, while Aceso [17], AdaPipe [28], Koala [31], and GraphPipe [9] use profiling. Mist [41] and PipeWeaver [34] begin to account for overlap effects. However, most frameworks still optimize partitioning separately from overlap scheduling, balancing serial costs rather than post-overlap execution. Tessera profiles post-overlap cost directly and partitions for parallel execution.

Dynamic MoE Load Balancing. Runtime imbalance in MoE has driven two directions: token/expert redistribution via replication or linear programming (FasterMoE [6], Janus [19], PopFetcher [38], LPLB [2]), and system-level adaptation via dynamic kernel tuning (Tutel [8], DeepSpeed-MoE [26]). Tessera is complementary: rather than trying to perfectly balance stochastic token loads, it accepts routing variation and absorbs the resulting pipeline bubbles by dynamically scheduling movable tasks, such as gradient reduction and offloading, into idle slots.

8 Conclusion

Training trillion-parameter heterogeneous MoE models exposes a cyclic dependency between pipeline partitioning and communication overlap that traditional stratified optimization cannot resolve. Tessera breaks this cycle by profiling post-overlap cost for each layer combination to co-optimize partitioning and overlap scheduling, while a Dynamic Bubble Optimizer fills routing-induced idle slots with movable tasks at runtime. Deployed on production clusters for pre-training Qwen families, Tessera improves throughput by 20%–33% across five workloads at scales from 4,096 to 12,288 GPUs, reaching 39% MFU on a trillion-parameter model and up to 1.24× higher MFU than Megatron-Core MoE in controlled experiments. Tessera demonstrates that a unified, profile-driven co-optimization approach systematically outperforms manual heuristics, providing a practical foundation for training next-generation heterogeneous AI models.

Acknowledgments

We thank our shepherd and the anonymous reviewers for their careful reading and constructive feedback. We would like to express our gratitude to the following individuals for their contributions to this work. For algorithmic insights, we are especially grateful to Bo Zheng, Rui Men, Yuqiong Liu, and Hao Ge, whose ideas and discussions have significantly shaped this work. For systems development and engineering, we thank Linlang Jiang, Zhixiang Ruan, Hongqing Chen, Yunfei Mao, Siyu Wang, Xing Wang, and Chang Si, who have continuously advanced the underlying system that made this work possible. Weifang Hu and Xuanhua Shi were supported in part by the National Key Research and Development Program of China (No. 2024YFB4505202), The Major Program (JD) of Hubei Province (No. 2023BAA024), and Hubei Provincial Natural Science Foundation of China (No. 2026AFA002).

References

[1] Li-Wen Chang, Wenlei Bao, Qi Hou, Chengquan Jiang, Ningxin Zheng, Yinmin Zhong, Xuanrun Zhang, Zuquan Song, Ziheng Jiang, Haibin Lin, Xin Jin, and Xin Liu. FLUX: Fast Software-based Communication Overlap On GPUs Through Kernel Fusion, 2024.

[2] DeepSeek-AI. Linear-Programming-Based Load Balancer (LPLB). https://github.com/deepseek-ai/LPLB, 2025.

[3] DeepSeek-AI, Aixin Liu, Bei Feng, Bing Xue, Bingxuan Wang, Bochao Wu, Chengda Lu, Chenggang Zhao, Chengqi Deng, Chenyu Zhang, Chong Ruan, Damai Dai, Daya Guo, Dejian Yang, Deli Chen, Dongjie Ji, Erhang Li, Fangyun Lin, Fucong Dai, Fuli Luo, Guangbo Hao, Guanting Chen, Guowei Li, H. Zhang, Han Bao, Hanwei Xu, Haocheng Wang, Haowei Zhang, Honghui Ding, Huajian Xin, Huazuo Gao, Hui Li, Hui Qu, J. L. Cai, Jian Liang, Jianzhong Guo, Jiaqi Ni, Jiashi Li, Jiawei Wang, Jin Chen, Jingchang Chen, Jingyang Yuan, Junjie Qiu, Junlong Li, Junxiao Song, Kai Dong, Kai Hu, Kaige Gao, Kang Guan, Kexin Huang, Kuai Yu, Lean Wang, Lecong Zhang, Lei Xu, Leyi Xia, Liang Zhao, Litong Wang, Liyue Zhang, Meng Li, Miaojun Wang, Mingchuan Zhang, Minghua Zhang, Minghui Tang, Mingming Li, Ning Tian, Panpan Huang, Peiyi Wang, Peng Zhang, Qiancheng Wang, Qihao Zhu, Qinyu Chen, Qiushi Du, R. J. Chen, R. L. Jin, Ruiqi Ge, Ruisong Zhang, Ruizhe Pan, Runji Wang, Runxin Xu, Ruoyu Zhang, Ruyi Chen, S. S. Li, Shanghao Lu, Shangyan Zhou, Shanhuang Chen, Shaoqing Wu, Shengfeng Ye, Shengfeng Ye, Shirong Ma, Shiyu Wang, Shuang Zhou, Shuiping Yu, Shunfeng Zhou, Shuting Pan, T. Wang, Tao Yun, Tian Pei, Tianyu Sun, W. L. Xiao, Wangding Zeng, Wanjia Zhao, Wei An, Wen Liu, Wenfeng Liang, Wenjun Gao, Wenqin Yu, Wentao Zhang, X. Q. Li, Xiangyue Jin, Xianzu Wang, Xiao Bi, Xiaodong Liu, Xiaohan Wang, Xiaojin Shen, Xiaokang Chen, Xiaokang Zhang, Xiaosha Chen, Xiaotao Nie, Xiaowen Sun, Xiaoxiang Wang, Xin Cheng, Xin Liu, Xin Xie, Xingchao Liu, Xingkai Yu, Xinnan Song, Xinxia Shan, Xinyi Zhou, Xinyu Yang, Xinyuan Li, Xuecheng Su, Xuheng Lin, Y. K. Li, Y. Q. Wang, Y. X. Wei, Y. X. Zhu, Yang Zhang, Yanhong Xu, Yanhong Xu, Yanping Huang, Yao Li, Yao Zhao, Yaofeng Sun, Yaohui Li, Yaohui Wang, Yi Yu, Yi Zheng, Yichao Zhang, Yifan Shi, Yiliang Xiong, Ying He, Ying Tang, Yishi Piao, Yisong Wang, Yixuan Tan, Yiyang Ma, Yiyuan Liu, Yongqiang Guo, Yu Wu, Yuan Ou, Yuchen Zhu, Yuduan Wang, Yue Gong, Yuheng Zou, Yujia He, Yukun Zha, Yunfan Xiong, Yunxian Ma, Yuting Yan, Yuxiang Luo, Yuxiang You, Yuxuan Liu, Yuyang Zhou, Z. F. Wu, Z. Z. Ren, Zehui Ren, Zhangli Sha, Zhe Fu, Zhean Xu, Zhen Huang, Zhen Zhang, Zhenda Xie, Zhengyan Zhang, Zhewen Hao, Zhibin Gou, Zhicheng Ma, Zhigang Yan, Zhihong Shao, Zhipeng Xu, Zhiyu Wu, Zhongyu Zhang, Zhuoshu Li, Zihui Gu, Zijia Zhu, Zijun Liu, Zilin Li, Ziwei Xie, Ziyang Song, Ziyi Gao, and Zizheng Pan. DeepSeek-V3 Technical Report, 2025.

[4] Paul Elvinger, Foteini Strati, Natalie Enright Jerger, and Ana Klimovic. Understanding GPU Resource Interference One Level Deeper, 2025.

[5] John Forrest and Robin Lougee-Heimer. CBC User Guide. INFORMS, 2005.

[6] Jiaao He, Jidong Zhai, Tiago Antunes, Haojie Wang, Fuwen Luo, Shangfeng Shi, and Qin Li. FasterMoE: Modeling and Optimizing Training of Large-Scale Dynamic Pre-Trained Models. In *Proceedings of the 27th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '22)*, pages 120–134, 2022.

[7] Yanping Huang, Youlong Cheng, Ankur Bapna, Orhan Firat, Mia Xu Chen, Dehao Chen, HyoukJoong Lee, Jiquan Ngiam, Quoc V. Le, Yonghui Wu, and Zhifeng Chen. GPipe: Efficient Training of Giant Neural Networks Using Pipeline Parallelism. In *Proceedings of the 33rd International Conference on Neural Information Processing Systems (NeurIPS '19)*, pages 103–112, 2019.

[8] Changho Hwang, Wei Cui, Yifan Xiong, Ziyue Yang, Ze Liu, Han Hu, Zilong Wang, Rafael Salas, Jithin Jose, Prabhat Ram, Joe Chau, Peng Cheng, Fan Yang, Mao Yang, and Yongqiang Xiong. Tutel: Adaptive Mixture-of-Experts at Scale. In *Proceedings of the 6th Conference on Machine Learning and Systems (MLSys '23)*, 2023.

[9] Byungsoo Jeon, Mengdi Wu, Shiyi Cao, Sunghyun Kim, Sunghyun Park, Neeraj Aggarwal, Colin Unger, Daiyaan Arfeen, Peiyuan Liao, Xupeng Miao, Mohammad Alizadeh, Gregory R. Ganger, Tianqi Chen, and Zhihao Jia. GraphPipe: Improving Performance and Scalability of DNN Training with Graph Pipeline Parallelism. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '25)*, pages 557–571, 2025.

[10] Seonho Lee, Jihwan Oh, Junkyum Kim, Seokjin Go, Jongse Park, and Divya Mahajan. Characterizing Compute-Communication Overlap in GPU-Accelerated Distributed Deep Learning: Performance and Power Implications, 2025.

[11] Dmitry Lepikhin, HyoukJoong Lee, Yuanzhong Xu, Dehao Chen, Orhan Firat, Yanping Huang, Maxim Krikun,

Noam Shazeer, and Zhifeng Chen. GShard: Scaling Giant Models with Conditional Computation and Automatic Sharding, 2020.

[12] Shenggui Li, Hongxin Liu, Zhengda Bian, Jiarui Fang, Haichen Huang, Yuliang Liu, Boxiang Wang, and Yang You. Colossal-AI: A Unified Deep Learning System For Large-Scale Parallel Training. In *Proceedings of the 52nd International Conference on Parallel Processing (ICPP '23)*, pages 766–775, 2023.

[13] Shigang Li and Torsten Hoefler. Chimera: Efficiently Training Large-Scale Neural Networks with Bidirectional Pipelines. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC '21)*, 2021.

[14] Junfeng Lin, Ziming Liu, Yang You, Jun Wang, Weihao Zhang, and Rong Zhao. WeiPipe: Weight Pipeline Parallelism for Communication-Effective Long-Context Large Model Training. In *Proceedings of the 30th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming (PPoPP '25)*, pages 225–238, 2025.

[15] Zhiqi Lin, Youshan Miao, Guanbin Xu, Cheng Li, Olli Saarikivi, Saeed Maleki, and Fan Yang. Tessel: Boosting Distributed Execution of Large DNN Models via Flexible Schedule Search. In *Proceedings of the IEEE International Symposium on High-Performance Computer Architecture (HPCA '24)*, pages 803–816, 2024.

[16] Zhiqi Lin, Youshan Miao, Quanlu Zhang, Fan Yang, Yi Zhu, Cheng Li, Saeed Maleki, Xu Cao, Ning Shang, Yilei Yang, Weijiang Xu, Mao Yang, Lintao Zhang, and Lidong Zhou. nnScaler: Constraint-Guided Parallelization Plan Generation for Deep Learning Training. In *Proceedings of the 18th USENIX Symposium on Operating Systems Design and Implementation (OSDI '24)*, pages 347–363, 2024.

[17] Guodong Liu, Youshan Miao, Zhiqi Lin, Xiaoxiang Shi, Saeed Maleki, Fan Yang, Yungang Bao, and Sa Wang. Aceso: Efficient Parallel DNN Training through Iterative Bottleneck Alleviation. In *Proceedings of the 19th European Conference on Computer Systems (EuroSys '24)*, pages 163–181, 2024.

[18] Hao Liu, Matei Zaharia, and Pieter Abbeel. Ring Attention with Blockwise Transformers for Near-Infinite Context. In *Proceedings of the 12th International Conference on Learning Representations (ICLR '24)*, 2024.

[19] Juncai Liu, Jessie Hui Wang, and Yimin Jiang. Janus: A Unified Distributed Training Framework for Sparse Mixture-of-Experts Models. In *Proceedings of the ACM SIGCOMM 2023 Conference (SIGCOMM '23)*, pages 486–498, 2023.

[20] Weijian Liu, Mingzhen Li, Guangming Tan, and Weile Jia. Mario: Near Zero-cost Activation Checkpointing in Pipeline Parallelism. In *Proceedings of the 30th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming (PPoPP '25)*, pages 197–211, 2025.

[21] Deepak Narayanan, Aaron Harlap, Amar Phanishayee, Vivek Seshadri, Nikhil R. Devanur, Gregory R. Ganger, Phillip B. Gibbons, and Matei Zaharia. PipeDream: Generalized Pipeline Parallelism for DNN Training. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP '19)*, pages 1–15, 2019.

[22] Deepak Narayanan, Mohammad Shoeybi, Jared Casper, Patrick LeGresley, Mostofa Patwary, Vijay Korthikanti, Dmitri Vainbrand, Prethvi Kashinkunti, Julie Bernauer, Bryan Catanzaro, Amar Phanishayee, and Matei Zaharia. Efficient Large-Scale Language Model Training on GPU Clusters Using Megatron-LM. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC '21)*, pages 1–15, 2021.

[23] NVIDIA. Megatron-Bridge: Enabling model support for Megatron-Core. https://github.com/NVIDIA-NeMo/Megatron-Bridge, 2025.

[24] NVIDIA. Nemotron 3 Super: Open, efficient mixture-of-experts hybrid Mamba-Transformer model for agentic reasoning. https://research.nvidia.com/labs/nemotron/Nemotron-3-Super/, 2026.

[25] Penghui Qi, Xinyi Wan, Guangxing Huang, and Min Lin. Zero Bubble (Almost) Pipeline Parallelism. In *Proceedings of the 12th International Conference on Learning Representations (ICLR '24)*, 2024.

[26] Samyam Rajbhandari, Conglong Li, Zhewei Yao, Minjia Zhang, Reza Yazdani Aminabadi, Ammar Ahmad Awan, Jeff Rasley, and Yuxiong He. DeepSpeed-MoE: Advancing Mixture-of-Experts Inference and Training to Power Next-Generation AI Scale. In *Proceedings of the 39th International Conference on Machine Learning (ICML '22)*, pages 18332–18346, 2022.

[27] Ao Sun, Weilin Zhao, Xu Han, Cheng Yang, Xinrong Zhang, Zhiyuan Liu, Chuan Shi, and Maosong Sun. Seq1F1B: Efficient Sequence-Level Pipeline Parallelism for Large Language Model Training, 2024.

[28] Zhenbo Sun, Huanqi Cao, Yuanwei Wang, Guanyu Feng, Shengqi Chen, Haojie Wang, and Wenguang Chen. AdaPipe: Optimizing Pipeline Parallelism with Adaptive Recomputation and Partitioning. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '24)*, pages 86–100, 2024.

[29] Zhenbo Sun, Shengqi Chen, Yuanwei Wang, Jian Sha, Guanyu Feng, and Wenguang Chen. MEPipe: Democratizing LLM Training with Memory-Efficient Slice-Level Pipeline Scheduling on Cost-Effective Accelerators. In *Proceedings of the 20th European Conference on Computer Systems (EuroSys '25)*, pages 1263–1278, 2025.

[30] Yehui Tang, Yichun Yin, Yaoyuan Wang, Hang Zhou, Yu Pan, Wei Guo, Ziyang Zhang, Miao Rang, Fangcheng Liu, Naifu Zhang, Binghan Li, Yonghan Dong, Xiaojun Meng, Yasheng Wang, Dong Li, Yin Li, Dandan Tu, Can Chen, Youliang Yan, Fisher Yu, Ruiming Tang, Yunhe Wang, Botian Huang, Bo Wang, Boxiao Liu, Changzheng Zhang, Da Kuang, Fei Liu, Gang Huang, Jiansheng Wei, Jiarui Qin, Jie Ran, Jinpeng Li, Jun Zhao, Liang Dai, Lin Li, Liqun Deng, Peifeng Qin, Pengyuan Zeng, Qiang Gu, Shaohua Tang, Shengjun Cheng, Tao Gao, Tao Yu, Tianshu Li, Tianyu Bi, Wei He, Weikai Mao, Wenyong Huang, Wulong Liu, Xiabing Li, Xianzhi Yu, Xueyu Wu, Xu He, Yangkai Du, Yan Xu, Ye Tian, Yimeng Wu, Yongbing Huang, Yong Tian, Yong Zhu, Yue Li, Yufei Wang, Yuhang Gai, Yujun Li, Yu Luo, Yunsheng Ni, Yusen Sun, Zelin Chen, Zhe Liu, Zhicheng Liu, Zhipeng Tu, Zilin Ding, and Zongyuan Zhan. Pangu Ultra MoE: How to Train Your Big MoE on Ascend NPUs, 2025.

[31] Yu Tang, Lujia Yin, Qiao Li, Hongyu Zhu, Hengjie Li, Xingcheng Zhang, Linbo Qiao, Dongsheng Li, and Jiaxin Li. Koala: Efficient Pipeline Training through Automated Schedule Searching on Domain-Specific Language. *ACM Trans. Archit. Code Optim.*, 22(2), 2025.

[32] Kimi Team, Yifan Bai, Yiping Bao, Guanduo Chen, Jiahao Chen, Ningxin Chen, Ruijue Chen, Yanru Chen, Yuankun Chen, Yutian Chen, Zhuofu Chen, Jialei Cui, Hao Ding, Mengnan Dong, Angang Du, Chenzhuang Du, Dikang Du, Yulun Du, Yu Fan, Yichen Feng, Kelin Fu, Bofei Gao, Hongcheng Gao, Peizhong Gao, Tong Gao, Xinran Gu, Longyu Guan, Haiqing Guo, Jianhang Guo, Hao Hu, Xiaoru Hao, Tianhong He, Weiran He, Wenyang He, Chao Hong, Yangyang Hu, Zhenxing Hu, Weixiao Huang, Zhiqi Huang, Zihao Huang, Tao Jiang, Zhejun Jiang, Xinyi Jin, Yongsheng Kang, Guokun Lai, Cheng Li, Fang Li, Haoyang Li, Ming Li, Wentao Li, Yanhao Li, Yiwei Li, Zhaowei Li, Zheming Li, Hongzhan Lin, Xiaohan Lin, Zongyu Lin, Chengyin Liu, Chenyu Liu, Hongzhang Liu, Jingyuan Liu, Junqi Liu, Liang Liu, Shaowei Liu, T. Y. Liu, Tianwei Liu, Weizhou Liu, Yangyang Liu, Yibo Liu, Yiping Liu, Yue Liu, Zhengying Liu, Enzhe Lu, Lijun Lu, Shengling Ma, Xinyu Ma, Yingwei Ma, Shaoguang Mao, Jie Mei, Xin Men, Yibo Miao, Siyuan Pan, Yebo Peng, Ruoyu Qin, Bowen Qu, Zeyu Shang, Lidong Shi, Shengyuan Shi, Feifan Song, Jianlin Su, Zhengyuan Su, Xinjie Sun, Flood Sung, Heyi Tang, Jiawen Tao, Qifeng Teng, Chensi Wang, Dinglu Wang, Feng Wang, Haiming Wang, Jianzhou Wang, Jiaxing Wang, Jinhong Wang, Shengjie Wang, Shuyi Wang, Yao Wang, Yejie Wang, Yiqin Wang, Yuxin Wang, Yuzhi Wang, Zhaoji Wang, Zhengtao Wang, Zhexu Wang, Chu Wei, Qianqian Wei, Wenhao Wu, Xingzhe Wu, Yuxin Wu, Chenjun Xiao, Xiaotong Xie, Weimin Xiong, Boyu Xu, Jing Xu, Jinjing Xu, L. H. Xu, Lin Xu, Suting Xu, Weixin Xu, Xinran Xu, Yangchuan Xu, Ziyao Xu, Junjie Yan, Yuzi Yan, Xiaofei Yang, Ying Yang, Zhen Yang, Zhilin Yang, Zonghan Yang, Haotian Yao, Xingcheng Yao, Wenjie Ye, Zhuorui Ye, Bohong Yin, Longhui Yu, Enming Yuan, Hongbang Yuan, Mengjie Yuan, Haobing Zhan, Dehao Zhang, Hao Zhang, Wanlu Zhang, Xiaobin Zhang, Yangkun Zhang, Yizhi Zhang, Yongting Zhang, Yu Zhang, Yutao Zhang, Yutong Zhang, Zheng Zhang, Haotian Zhao, Yikai Zhao, Huabin Zheng, Shaojie Zheng, Jianren Zhou, Xinyu Zhou, Zaida Zhou, Zhen Zhu, Weiyu Zhuang, and Xinxing Zu. Kimi K2: Open Agentic Intelligence, 2025.

[33] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is All You Need. In *Proceedings of the 30th International Conference on Neural Information Processing Systems (NIPS '17)*, pages 5998–6008, 2017.

[34] Zhenliang Xue, Hanpeng Hu, Xing Chen, Yimin Jiang, Yixin Song, Zeyu Mi, Yibo Zhu, Daxin Jiang, Yubin Xia, and Haibo Chen. PipeWeaver: Addressing Data Dynamicity in Large Multimodal Model Training with Dynamic Interleaved Pipeline, 2025.

[35] Zijie Yan et al. Megatron-MoE-ModelZoo: Best practices for training MoE models with Megatron-Core. https://github.com/yanring/Megatron-MoE-ModelZoo, 2025.

[36] Zijie Yan et al. Scalable training of mixture-of-experts models with Megatron Core, 2026.

[37] An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengen Huang, Chenxu Lv, Chujie Zheng, Dayiheng Liu, Fan Zhou, Fei Huang, Feng Hu, Hao Ge, Haoran Wei, Huan Lin, Jialong Tang, Jian Yang, Jianhong Tu, Jianwei Zhang, Jianxin Yang, Jiaxi Yang, Jing Zhou, Jingren Zhou, Junyang Lin, Kai Dang, Keqin Bao, Kexin Yang, Le Yu, Lianghao Deng, Mei Li, Mingfeng Xue, Mingze Li, Pei Zhang, Peng Wang, Qin Zhu, Rui Men, Ruize Gao, Shixuan Liu, Shuang Luo, Tianhao Li, Tianyi Tang, Wenbiao Yin, Xingzhang Ren, Xinyu Wang, Xinyu Zhang, Xuancheng Ren, Yang Fan, Yang Su, Yichang Zhang, Yinger Zhang, Yu Wan, Yuqiong Liu, Zekun

Wang, Zeyu Cui, Zhenru Zhang, Zhipeng Zhou, and Zihan Qiu. Qwen3 Technical Report, 2025.

[38] Junyi Zhang, Chuanhu Ma, Xiong Wang, Yuntao Nie, Yuqing Li, Yuedong Xu, Xiaofei Liao, Bo Li, and Hai Jin. PopFetcher: Towards Accelerated Mixture-of-Experts Training Via Popularity Based Expert-Wise Prefetch. In *Proceedings of the 2025 USENIX Annual Technical Conference (ATC '25)*, pages 1053–1069, 2025.

[39] Shulai Zhang, Ningxin Zheng, Haibin Lin, Ziheng Jiang, Wenlei Bao, Chengquan Jiang, Qi Hou, Weihao Cui, Size Zheng, Li-Wen Chang, Quan Chen, and Xin Liu. Comet: Fine-grained Computation-communication Overlapping for Mixture-of-Experts, 2025.

[40] Lianmin Zheng, Zhuohan Li, Hao Zhang, Yonghao Zhuang, Zhifeng Chen, Yanping Huang, Yida Wang, Yuanzhong Xu, Danyang Zhuo, Eric P. Xing, Joseph E. Gonzalez, and Ion Stoica. Alpa: Automating Inter- and Intra-Operator Parallelism for Distributed Deep Learning. In *Proceedings of the 16th USENIX Symposium on Operating Systems Design and Implementation (OSDI '22)*, pages 559–578, 2022.

[41] Zhanda Zhu, Christina Giannoula, Muralidhar Andoorveedu, Qidong Su, Karttikeya Mangalam, Bojian Zheng, and Gennady Pekhimenko. Mist: Efficient Distributed Training of Large Language Models via Memory-Parallelism Co-Optimization. In *Proceedings of the 20th European Conference on Computer Systems (EuroSys '25)*, pages 1298–1316, 2025.

ARTIFACT EVALUATED
USENIX
AVAILABLE

Hetu v2: A General and Scalable Deep Learning System with Hierarchical and Heterogeneous Single Program Multiple Data Annotations

Haoyang Li[§] Fangcheng Fu[†] Hao Ge[§] Sheng Lin[§]
Xuanyu Wang[§] Jiawen Niu[§] Yuming Zhou[§] Xupeng Miao[§] Bin Cui[§‡]

[§]*School of Computer Science & Beijing Key Laboratory of Software and Hardware Cooperative Artificial Intelligence Systems, Peking University* [†]*School of Artificial Intelligence, Shanghai Jiao Tong University*
[‡]*Institute of Computational Social Science, Peking University (Qingdao)*

Abstract

The Single-Program Multiple-Data (SPMD) paradigm provides a unified abstraction to annotate various parallel dimensions in distributed deep learning (DL) training. With SPMD, users can write training programs from the viewpoint of a single device, and the system will automatically deduce the tensor sharding and communication patterns. However, with the recent development in large-scale DL models, distributed training exhibits *spatial* and *temporal* workload heterogeneity, arising from both device disparities (e.g., mixed hardware, failures) and data variations (e.g., uneven sequence lengths). Such heterogeneity violates SPMD's assumption of symmetric workload partitioning, which restricts its ability to express and optimize heterogeneous parallel strategies effectively.

To address this, we propose HSPMD within the Hetu v2 system to achieve general and scalable DL training. HSPMD extends SPMD's *declarative annotations* to support asymmetric sharding and composes standard communication primitives for hierarchical communication, all while retaining the simplicity of a single-device programming model. HSPMD handles *spatial* heterogeneity through progressive graph specialization, enabling device-specific execution logic, and addresses *temporal* heterogeneity via dynamic graph switching. Evaluations on (a) heterogeneous devices, (b) unstable devices, and (c) mixed-length data scenarios show that HSPMD matches or outperforms specialized systems, providing a flexible and efficient solution for modern distributed DL training. Code is available: https://github.com/PKU-DAIR/Hetu.

1 Introduction

The rapid advancement of large-scale deep learning (DL) models, particularly large language models (LLMs) [17, 34, 83, 85, 89, 93], has pushed the boundaries of distributed training systems. Models like GPT [55], Gemini [64], and DeepSeek [15] routinely span billions of parameters, requiring a large number of devices working in concert for efficient training.

To meet this demand, parallel strategies have evolved from singular approaches to sophisticated hybrid schemes combining data [45, 91], sequence [5, 28, 47], tensor [37, 51, 60], and pipeline [27, 35, 50, 57] parallelism. Within this landscape, the Single-Program Multiple-Data (SPMD) paradigm has emerged as a dominant abstraction due to its simplicity and scalability. By augmenting a single program with *declarative annotations*, SPMD enables automatic parallelization while abstracting away low-level deployment details. For example, systems such as GSPMD [86] and Alpa [92] annotate tensors with sharding semantics like *Split* and *Duplicate*, while DeepSpeed [28, 58] and Megatron [37, 51, 60] annotate model layers with distributed variants (e.g., extending Linear into ColumnParallelLinear and RowParallelLinear). This approach decouples parallel strategy specification from the DL model, facilitating more scalable distributed training.

At the core of this declarative SPMD paradigm lies a fundamental premise: the training workload should be uniformly partitioned and distributed. This is because SPMD primitives are built to symmetrically shard tensors or layers across devices, following the assumptions that **(i)** hardware resources are homogeneous and stable, and **(ii)** training burdens from all input data are equivalent. However, as DL training continues to scale, heterogeneous workloads have become increasingly prevalent. In practice, both hardware devices and input data introduce heterogeneity, as summarized in Table 1, leaving the standard SPMD design poorly suited to such scenarios.

Multifaceted heterogeneity. (i) From the device perspective, heterogeneous and unstable devices are common in real-world cluster/cloud environments. On the one hand, due to the GPU shortage problem [62, 88], modern MLaaS platforms typically host mixed GPU generations [30, 79], and the scarcity of high-end GPUs (e.g., A100, H100) has led cloud providers to leverage heterogeneous GPUs with divergent capabilities (e.g., computation FLOPS, memory capacities, network bandwidth) for training [25, 63]. On the other hand, device instability like GPU/node failures [30] occurs frequently. For example, Llama 3's training underwent 419 unexpected interruptions over 54 days, with 148 attributable to faulty GPUs [18]. **(ii)** From the data perspective, the symmetric design of SPMD is a good fit when all input data are associated with equiva-

Table 1: Example heterogeneous scenarios arising from device and data aspects, along with their characteristics and representative specialized SPMD training systems addressing each scenario.

Scenarios	Characteristics		Representatives
	Spatial	Temporal	
Heterogeneous Devices	✓		HexiScale [87] Sailor [63], etc.
Unstable Devices	✓	✓	Oobleck [29], ReCycle [21], etc.
Mixed-length Data	✓	✓	HotSPa [23], WLB-LLM [78], etc.

lent workloads (e.g., preprocessed tabular datasets). However, to achieve general intelligence, modern DL models are often trained with unstructured, raw data, which demonstrates inherent length variations (e.g., varying text lengths [38], diverse image resolutions [16], differing video/audio durations [10]).

In short, these pervasive forms of heterogeneity fundamentally break the symmetry assumption of the standard SPMD paradigm and give rise to an urgent demand for asymmetric execution to effectively tackle heterogeneity.

Current solutions and limitations. As illustrated in Figure 1 (left), one alternative is the Multiple-Program Multiple-Data (MPMD) paradigm, which uses distinct programs to encode different execution logic. However, MPMD often suffers from limited scalability and user-friendliness, as discussed in §2.3. Consequently, most distributed DL training systems addressing heterogeneity still adhere to the SPMD paradigm. As summarized in Table 1, a wide array of recent studies have extended standard SPMD to specific heterogeneous scenarios, including: heterogeneous devices [31,41,61,63,71,84,87,90], unstable devices (i.e., elastic training) [3, 21, 29, 69], and mixed-length data [7,22,23,33,76,78]. A common approach among them is to integrate custom schedulers (e.g., HexiScale's heterogeneous pipeline scheduler [87], Oobleck's elastic pipeline scheduler [29]) to enable asymmetric execution behaviors beyond standard SPMD, as shown in Figure 1 (middle). However, although these schedulers address heterogeneity in their respective scenarios, they are largely scenario-specific and built into the system with tight coupling to particular workloads. This design limits their flexibility and prevents them from serving as a general-purpose solution.

Our solution and contributions. Compared with prior case-by-case solutions, we propose HSPMD (Hierarchical and Heterogeneous SPMD) within Hetu v2, a system that addresses multifaceted heterogeneity from a more general and fundamental perspective. **(i) Primitive-level extensions.** As shown in Figure 1 (right), instead of layering extensive scheduler-level efforts, HSPMD pushes extensibility down to the primitive level of *declarative annotations*. Specifically, we design the HSPMD sharding annotations (§4) and hierarchical communication resolution (§5), which enable asymmetric sharding and communication beyond SPMD's inherent symmetry. While preserving SPMD's core principle of separating

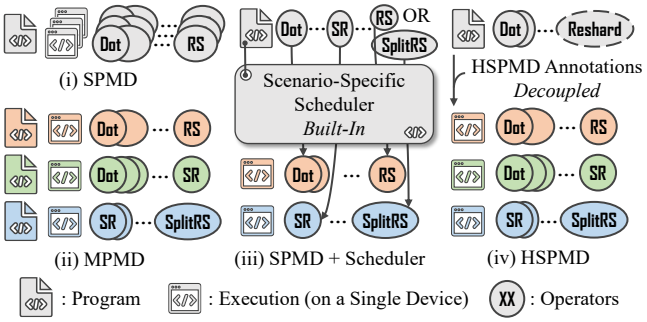


Figure 1: Comparison of SPMD, MPMD, scheduler-enhanced SPMD variants, and HSPMD. While SPMD imposes strict symmetry, the others enable asymmetric execution to tackle heterogeneity.

the parallel strategy from a single programming view of the DL model, HSPMD's primitives enable asymmetric execution natively, without the need for crafting scenario-specific schedulers. **(ii) Characteristic-driven abstractions**. Departing from prior work that categorizes solutions by application scenarios (Table 1), we revisit and disentangle heterogeneity through its intrinsic characteristics: *spatial* and *temporal* (detailed in §2.3). *Spatial* heterogeneity is due to the imbalanced workload and necessitates spatially heterogeneous parallel strategies (Figure 2), while *temporal* heterogeneity arises from dynamic workload changes and requires temporal reconfiguration across strategies (Figure 3). Accordingly, we introduce graph specialization (§6) and graph switching (§7), respectively. While these abstractions are scenario-agnostic individually, they serve as modular building blocks that can be composed into scenario-specific solutions, allowing HSPMD to generalize beyond single-purpose SPMD variants.

In summary, this paper makes the following contributions.

- We introduce primitive-level extensions to break SPMD's inherent symmetry, directly bridging single-device programming with heterogeneous parallelization.
- We present a general system design methodology that builds upward from fundamental heterogeneity characteristics, rather than starting from specific heterogeneous scenarios and customizing downward as in prior specialized systems.
- We evaluate HSPMD across diverse training scenarios, covering (a) heterogeneous devices, (b) unstable devices, and (c) mixed-length data. Empirical results show that HSPMD is at least on par with, and in most cases surpasses, state-of-the-art SPMD systems and specialized SPMD variants.

2 Background and Motivations

2.1 Sharding and Communication in Training

As model sizes increase, the computational and memory resources of a single device become insufficient to support training, necessitating the adoption of distributed training. This introduces two critical challenges: **(i)** how to effectively manage the **sharding** of different training components across devices; and **(ii)** how to efficiently coordinate their **communi-**

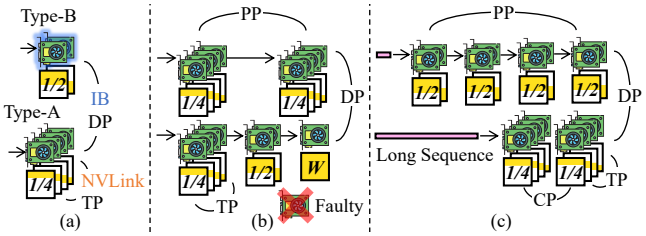


Figure 2: Use spatially heterogeneous parallel strategies to tackle spatial heterogeneity in scenario (a)-(c). "DP, CP, TP, PP" represent data, context, tensor, and pipeline parallelism, respectively. And scenarios (a), (b), and (c) refer to heterogeneous devices, unstable devices, and mixed-length data, respectively.

cation. To address these challenges, a variety of parallelisms and distributed training techniques have been proposed.

Data sharding. Data parallelism (DP) [45, 91] shards the input data along the batch dimension, enabling different model replicas to process data simultaneously and synchronize gradients via all-reduce. For long sequences, techniques like sequence or context parallelism (SP [28] or CP [5, 47]) can be applied to further shard the sequence dimension. These require additional communication (e.g., all-to-all or ring-based send-receive) to ensure correctness of the attention [73].

Model sharding. Pipeline parallelism (PP) [24, 27, 35, 50, 57] shards model weights across layers, where activations are communicated between pipeline stages via send-receive. Besides, tensor parallelism (TP) [37, 51, 60] shards model weights along the hidden size dimension, requiring frequent all-gather and reduce-scatter on activations at each layer.

Optimizer states sharding. Optimizer states consume significant memory. ZeRO [58] fully shards them across devices, necessitating all-gather and reduce-scatter operations, while some other methods [8, 9, 43] use finer-grained sharding, balancing storage redundancy and communication costs.

These techniques are often combined [42, 46, 48, 74, 75, 77], creating complex sharding and communication patterns that challenge system-level expression and handling at scale.

2.2 SPMD Training Systems

DL training can be modeled as a directed acyclic graph (DAG), where nodes represent operations and edges denote data dependencies. Tensors are the fundamental units flowing through the graph, carrying numerical values and metadata (e.g., shape, type). Operators transform input tensors into outputs (e.g., dot, attention). Following this, SPMD has emerged as a powerful paradigm (e.g., GSPMD [86], Alpa [92], Unity [72], DTensor [67, 68]). As shown in Figure 4 (left), by specifying the DAG with *declarative annotations*, SPMD decouples the programming from the complexities of parallelization, enabling automatic derivation and accommodation for diverse sharding and communication patterns.

Limitations. Despite its advantages, the SPMD paradigm imposes strict symmetry constraints. First, existing SPMD

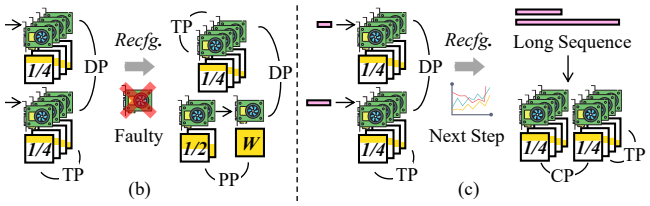


Figure 3: For scenarios (b) and (c), beyond the spatial heterogeneity shown in Figure 2, heterogeneity may also evolve over time, demanding temporal reconfiguration between parallel strategies. (Scenario (a) is omitted because it exhibits only static spatial heterogeneity.)

annotations are confined to the uniform partitioning of tensors across regular device meshes. Additionally, SPMD-style collective communication primitives, such as all-reduce, all-gather, and reduce-scatter, require symmetric participation from devices. These two constraints make it challenging for SPMD to handle **asymmetric sharding** and **communication**.

2.3 Other Paradigms and Our Motivations

As mentioned earlier, both devices and data can induce heterogeneity, leading to scenarios such as: (a) heterogeneous devices, (b) unstable devices, and (c) mixed-length data.

MPMD vs. SPMD. Due to its symmetry constraints, SPMD is ill-suited for these diverse heterogeneous scenarios. As shown in Figure 1 (left), one alternative is the MPMD paradigm (e.g., Pathways [4], Ray [49]). However, MPMD is more suited for multi-task settings (e.g., RL for LLMs [44, 59]), where heterogeneity manifests across tasks. By contrast, in single-task training with intra-task heterogeneity, MPMD typically requires generating and managing thousands of distinct programs across a large cluster, incurring substantial compilation overheads [39] and leading to a poor user experience. As a result, the scalability and simplicity of the single-program paradigm in SPMD remain highly desirable at large scale.

Scheduler-enhanced SPMD variants. Consequently, many systems continue to build upon SPMD but integrate a high-level scheduler to enable heterogeneous execution logic, as shown in Figure 1 (middle). While maintaining a single programming model, this approach requires workarounds to bypass SPMD's inherent symmetry, which introduces extensive control and branching logic within the scheduler. More critically, these schedulers are built-in and scenario-specific, making such SPMD variants only applicable to a single scenario.

Motivations. This motivates a more principled approach beyond scheduler-level customization. Specifically, we ask: can SPMD itself provide intrinsic support for asymmetry? A natural direction is to extend its *declarative annotations*. Compared with built-in schedulers, this approach offers better decoupling, natively expresses asymmetric parallelization, and provides a unified substrate across diverse scenarios.

Moreover, prior work treats each heterogeneous scenario as an isolated case, requiring dedicated system mechanisms and ad hoc extensions, which is fundamentally unscalable. This

motivates us to seek a more essential characterization that cuts across diverse scenarios. Specifically, we distill two fundamental dimensions underlying multifaceted heterogeneity: *spatial* heterogeneity and *temporal* heterogeneity (Table 1).

Spatial heterogeneity refers to workload imbalance at a given moment, calling for spatially heterogeneous parallel strategies. For example (Figure 2): (a) for heterogeneous devices, one can adopt a higher TP for memory-constrained GPUs and a lower TP for others; (b) for unstable devices, heterogeneous TP grouping and PP composition allow the remaining GPUs (e.g., 15 GPUs) to be fully utilized when one GPU fails; (c) for mixed-length data, GPUs can be partitioned into subgroups optimized for distinct sequence lengths (e.g., employing larger TP or CP for long sequences).

Temporal heterogeneity reflects workload variation over time, necessitating dynamic reconfiguration of parallel strategies. For example (Figure 3): (b) for unstable devices, when a GPU/node becomes unavailable, the overall parallel strategy must be reconfigured; (c) for mixed-length data, when the distribution of sequence lengths shifts across training steps (e.g., a step with very long sequences appears), reconfiguring to a strategy tailored for long sequences is essential to ensure better performance or to avoid out-of-memory (OOM) errors.

To this end, we introduce HSPMD within Hetu v2. **(i)** At the primitive level, as illustrated in Figure 4 (right), HSPMD extends SPMD to natively support asymmetric sharding (§4) and communication (§5), while preserving single-program semantics and collective operations. **(ii)** At the abstraction level, to address *spatial* and *temporal* heterogeneity, we propose graph specialization (§6) and graph switching (§7), enabling heterogeneous deployment and dynamic reconfiguration, respectively. Together, these innovations establish HSPMD as a general system for multifaceted heterogeneity.

3 HSPMD Design and Overview

In this section, we provide a walkthrough of HSPMD's core design and workflow. As depicted in Figure 5 (left), following prior work [32, 45, 92], HSPMD adopts a graph-based representation. Users provide a single *defined graph* consisting of three types of operators: **(i)** Leaf operators (e.g., placeholders, parameters) that produce data, model parameters, or optimizer states; **(ii)** `Reshard` operators, introduced by HSPMD as abstract markers to indicate potential resharding of intermediate results (e.g., gathering activations, resharding half-precision parameters); **(iii)** All other operators that encode the model's computation logic. An *annotation plan*, either user-specified or generated by an external planner, is then associated with the *defined graph*. This plan assigns output sharding annotations (§4) to **(i)** leaf and **(ii)** `Reshard` operators, defining the initial sharding layout and resharding behavior of critical tensors, thereby collectively specifying the parallel strategy.

Given the *defined graph* and the *annotation plan*, HSPMD deduces the output sharding of **(iii)** all other operators and resolves how each `Reshard` operator can be concretely achieved by communication operators (e.g., reduce-scatter, all-gather) automatically (§5).[1] Additionally, HSPMD will transparently handle the deployment and execution of parallel strategies, allowing users to write a single program by simply providing the *defined graph* (and optionally the *annotation plan*), without engaging in manual orchestration of distributed training.

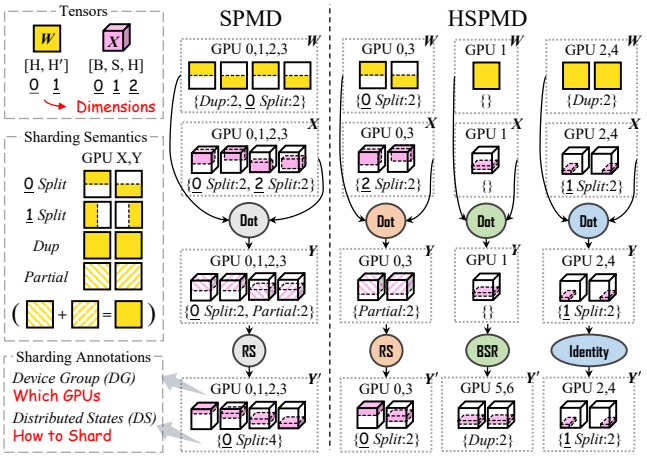


Figure 4: Left: Standard SPMD-style sharding annotations (e.g., DTensor [67, 68]) and their deduction when employing DP and TP across GPU 0-3. They can only express symmetric sharding and communication. Right: An example of HSPMD expressing asymmetry: TP between GPU 0,3 and GPU 5,6; PP between GPU 1 and GPU 5,6; CP between GPU 2,4; and DP across them. "`RS`, `BSR`" represents reduce-scatter and batched-send-receive (§5.3), respectively.

HSPMD leverages two abstractions to tackle heterogeneity (Figure 5, middle). Firstly, to address *spatial* heterogeneity, where devices may require divergent execution logic, we introduce graph specialization (§6). Once the *defined graph* and *annotation plan* are in place, each device specializes its local operators, generating a device-specific *executable graph* for deployment and execution. Secondly, to address *temporal* heterogeneity, where the parallel strategy needs to be reconfigured, we introduce graph switching (§7). When a new *annotation plan* (e.g., changes in parameter sharding) is introduced, HSPMD facilitates seamless switching between parallel strategies by online resharding the necessary weights between *executable graphs*, without requiring a restart.

As illustrated in Figure 5 (middle and right), these two abstractions, graph specialization and switching, are modular building blocks that can be flexibly combined to accommodate diverse heterogeneous scenarios. **(a) Heterogeneous devices** (*spatial* heterogeneity only): An expert user or scenario-specific planner provides an *annotation plan* specifying a heterogeneous parallel strategy. *Executable graphs* are specialized once offline (line 1) and remain fixed throughout training (line 9). **(b) Unstable devices** (*spatial* + unknown

[1] Not all communication needs to be explicitly represented via `Reshard`. For example, CP involves ring-based send–receive within attention [47]. We provide a fused `ParallelAttn` operator to encapsulate such implicit communication, keeping it transparent to the annotation system.

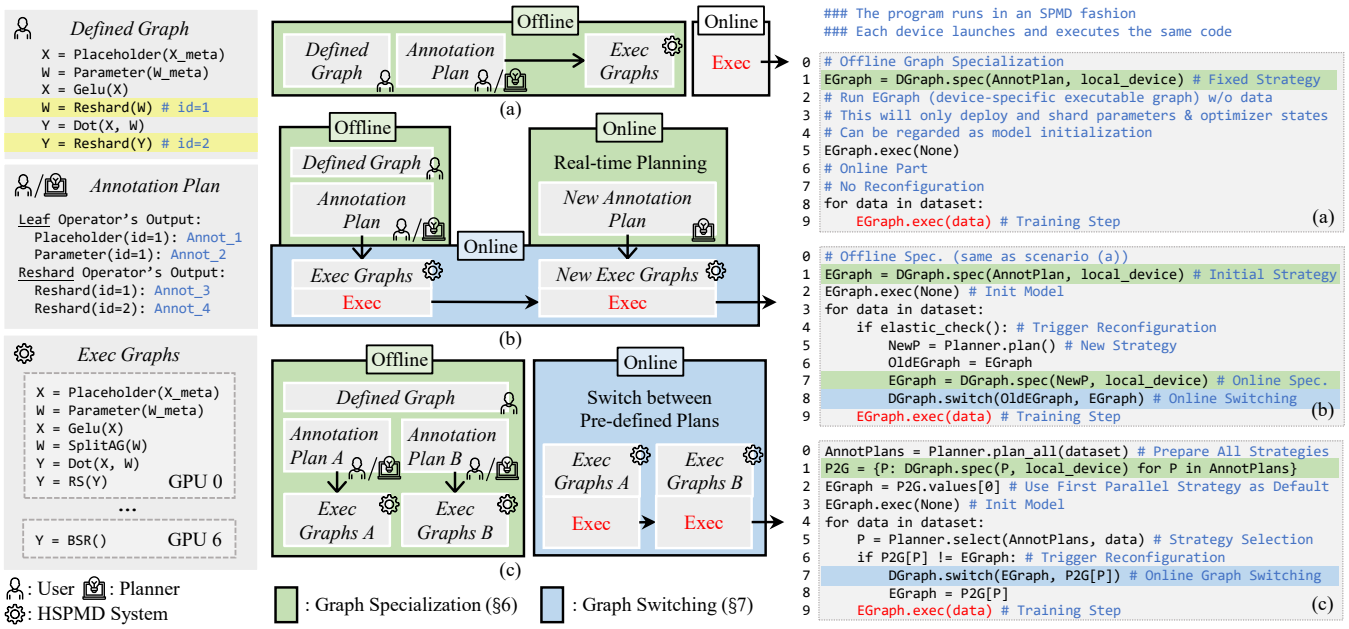


Figure 5: HSPMD's workflow and code snippets (DL operators align with Figure 4). Our system relies on two abstractions: graph specialization (§6), which generates device-specific *executable graphs* from a *defined graph* (provided by users) and an *annotation plan* (provided by users or planners); and graph switching (§7), which enables switching between arbitrary *executable graphs*. These two abstractions act as building blocks that can be flexibly applied to: (a) heterogeneous devices, (b) unstable devices, and (c) mixed-length data.

temporal heterogeneity): The initial setup mirrors case (a). Upon triggering a reconfiguration (e.g., due to changes in device availability), an automatic planner generates a new *annotation plan* (line 5), and the system specializes new *executable graphs* at runtime (line 7). Graph switching then seamlessly transitions model and optimizer states (line 8), ensuring uninterrupted training (line 9). **(c) Mixed-length data** (*spatial* + predictable *temporal* heterogeneity): Users or planners prepare multiple *annotation plans* (line 0), each optimized for a particular sequence length distribution, with corresponding *executable graphs* pre-specialized offline (line 1). During training, the system dynamically selects and switches to the optimal plan when input distribution shifts (line 5-8).

Although scenario-specific planners are needed (e.g., for *annotation plan* generation and selection), HSPMD is designed as a general framework to address diverse forms of heterogeneity, rather than to optimize automatic planning for a single scenario. Our focus is on introducing system core designs, while keeping the planner modular and easily replaceable. In the following, we first detail the sharding annotations (§4) and communication resolution (§5), which facilitate asymmetric sharding and communication, respectively. We then show how HSPMD addresses *spatial* heterogeneity with graph specialization (§6) and *temporal* heterogeneity via graph switching (§7). Finally, we briefly introduce HSPMD's planners (§8).

4 Sharding Annotations

This section introduces HSPMD's sharding annotations, which hierarchically extend SPMD-style annotations through a bottom-up approach to support asymmetric sharding. Compared to crafting high-level specialized schedulers on top of SPMD, this low-level extension enables SPMD itself to express asymmetry and generalize to diverse scenarios.

4.1 Basic SPMD-style Annotations

Following existing SPMD-style *declarative annotations* such as GSPMD [86] and DTensor [67, 68], we associate each tensor with two attributes: **(i) *Device Group (DG)***, which specifies the devices where the tensor resides; **(ii) *Distributed States (DS)***, which defines how the tensor is sharded across these devices. While some critical tensors (e.g., data, parameters, `Reshard` outputs) have their *DG* and *DS* explicitly specified by the *annotation plan*, others are deduced automatically.

As shown in Figure 4 (left), the *DG* is represented as an ordered list of GPU indices, and the *DS* is an ordered dictionary where the key *dim* corresponds to a logical distributed dimension (a virtual axis for sharding), and its corresponding value represents the number of shards along *dim*. Following previous works, there are three sharding semantics: **(i) *Split*** ($dim \geq 0$), where the tensor is uniformly split along its physical shape dimension *dim*; **(ii) *Duplicate*** ($dim = -1$), where the tensor is fully replicated; **(iii) *Partial*** ($dim = -2$), where the tensor's values are partially stored. These annotations come with inherent deduction rules. For instance in Figure 4 (left), given the annotations of W and X, the annotation of their `Dot` result can be derived: Y retains the same *DG* as the inputs, while its *DS* transforms the original *split* into *partial*.

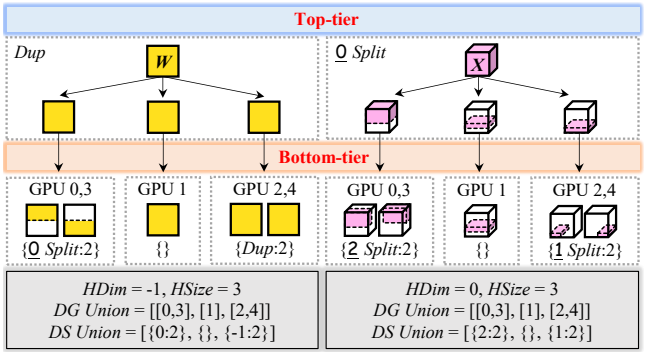


Figure 6: HSPMD's sharding annotations (aligned with Figure 4). The tensor is first sharded along the *HDim* dimension into *HSize sharding subgroups*. The devices in each *sharding subgroup* are given by the corresponding *DG* within the *DG Union*. And each *sharding subgroup* applies a *DS* from the *DS Union*. In this example, both W and X contain 3 *sharding subgroups* (*HSize* = 3).

4.2 Hierarchical & Heterogeneous Extensions

Building on basic SPMD annotations, we introduce hierarchical constructs to support asymmetric sharding, enabling heterogeneous execution logic as shown in Figure 4 (right).

(I) *Device Group Union (DG Union)* and *Distributed States Union (DS Union).* As illustrated in Figure 6, we generalize the single *DG* and *DS* associated with each tensor to multiple *DG* partitions and *DS* configurations, forming *DG Union* and *DS Union*. Each *DS* configuration in the union corresponds to a *DG* partition in the union, describing sharding within that particular device subgroup, referred to as a *sharding subgroup*.[2] We term the sharding within each subgroup as **bottom-tier sharding**. This approach enables the simultaneous representation of multiple sharding patterns.

(II) *Heterogeneous Dimension (HDim)* and *Heterogeneous Size (HSize).* To model relationships between *sharding subgroups* in a union, we further introduce *Heterogeneous Dimension (HDim)* and *Heterogeneous Size (HSize)*. As illustrated in Figure 6, *HSize* specifies the number of distinct *sharding subgroups*, while *HDim* describes the sharding across them. For instance, $HDim = 0$ for tensor X implies splitting the tensor's first dimension across subgroups, while $HDim = -1$ for tensor W indicates full replication across subgroups. We refer to this as the **top-tier sharding**.[3]

4.3 Discussions

HSPMD's annotations can be viewed as a sharding extension layered atop standard SPMD annotations. Although this extension introduces additional system complexity, much of it remains hidden from the users' view. For one thing, as introduced in §3, users only need to define a logical computation graph (*defined graph*) and optionally provide a small set of annotations (*annotation plan*), the rest can be automatically deduced by our system. For another, HSPMD's annotation deduction can be decomposed into multiple standard SPMD deductions within each *sharding subgroup*, thereby simplifying the deduction procedure (this will be detailed in §6.1).

Readers might wonder whether it is more preferable to consider a more granular multi-tier hierarchy, or a simpler scheme that only shards data unevenly across devices. We discuss these two alternatives below.

- *Why two-tier, not multi-tier:* Introducing a multi-tier hierarchy is unnecessary. *HDim* and *HSize* at the top tier already suffice to express complex asymmetric shardings while keeping the added complexity minimal. Moreover, the two-tier design matches how compute resources are organized in practice: a node or node group is internally load-balanced under SPMD, while heterogeneity exists across nodes or node groups. Adding further tiers would enlarge the planning space and deduction complexity without yielding additional expressiveness for today's cluster deployment.
- *Why not simply shard data unevenly:* Sharding data alone, without a fundamental change at the annotation level, is insufficient. **(i)** For *temporal* heterogeneity, it is ineffective: when a GPU fails (Figure 3(b)), the TP/PP degrees must be reconfigured, which cannot be handled by data re-sharding alone. **(ii)** For *spatial* heterogeneity, it is suboptimal: under low inter-cluster bandwidth, heterogeneous DP requires exchanging full gradients, whereas heterogeneous PP only transfers activations for a single layer, with much smaller volume. Capturing such PP strategies requires annotation-level support beyond simple data sharding.

Once tensor annotations are specified by the *annotation plan* or deduced by the system, HSPMD will determine the exact communication pattern for each `Reshard` operator and will instantiate it with device-specific communication operators that can be executed directly. The next section (§5) will describe how the asymmetric communication operators (e.g., `BSR` in Figure 4,5) are derived from the given annotations.

5 Hierarchical Communication Resolution

As depicted in Figure 7, to reshard from a source annotation to a destination one, we develop a comprehensive classification procedure to determine the appropriate communication operators. Aligning with the concepts of bottom-tier and top-tier sharding, we classify communication into two categories: **bottom-tier communication** and **top-tier communication**. Bottom-tier communication operates independently within each *sharding subgroup*, determined solely by changes in bottom-tier sharding (§5.1). In contrast, top-tier communication requires interactions across *sharding subgroups* and is influenced by the entire annotation hierarchy (§5.2).

During this procedure, we prioritize the use of collective communication operators, decomposing asymmetric communication into multiple symmetric collective communications

[2] *Sharding subgroups* must consist of mutually exclusive device subsets.

[3] The shape of each shard can vary and is determined at runtime, which is decoupled from the static annotation system (as elaborated in §6.3).

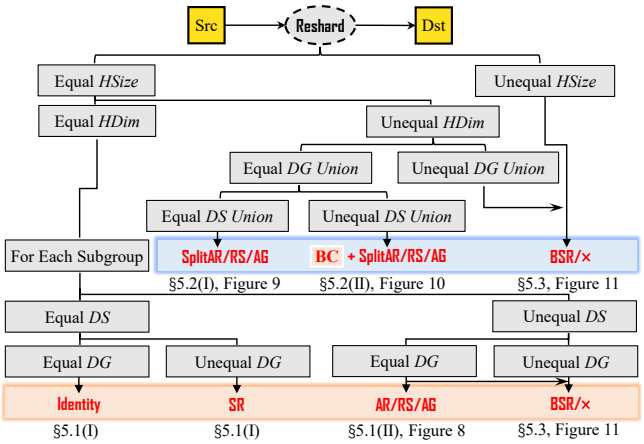


Figure 7: HSPMD's communication resolution. Orange denotes bottom-tier communication, executed independently within *sharding subgroups*; blue indicates top-tier communication, involving interactions between *sharding subgroups*. "SR": send-receive; "AR, RS, AG": all-reduce, reduce-scatter, all-gather; "BSR": batched-send-receive; "×": unsupported; "BC": bottom-tier communication.

when possible. For one thing, collective communication libraries (CCLs) have been extensively optimized [13, 26, 40, 54], often achieving higher performance than hand-crafted kernels. For another, modern GPU cluster network topologies (e.g., Multi-rail [1, 2], Clos [80]) are typically organized into multiple tiers [19], which aligns well with our hierarchical design. Specifically, **(i)** bottom-tier communication within *sharding subgroups* is usually operated on homogeneous devices (intra-machine or inter-machine with uniform links, typically high bandwidth) and can exploit fast symmetric collective primitives, whereas **(ii)** top-tier communication handles potentially heterogeneous inter-subgroup links (e.g., IB/TCP mixtures, often slower). When collective communication is infeasible, we develop a batched-send-receive (BSR) operator (§5.3) to accomplish sophisticated communications.

5.1 Bottom-tier Communication

When the source and destination annotations share the same *HSize* and *HDim*, it implies that the top-tier sharding remains unchanged. Consequently, all *sharding subgroups* can individually perform communication, typically reducing to standard SPMD resolution. As illustrated in Figure 7, for each *sharding subgroup*, we consider the following cases:

(I) Subgroup's source *DS* equals destination *DS*. We then determine whether the *DG* changes from the source to the destination, in which case each corresponding device executes either an Identity or send-receive (SR) operator to directly transfer the local shard.

(II) Subgroup's source *DS* does not equal destination *DS*. If the source and destination share the same *DG*, the device composition of the subgroup is unchanged, and only intra-subgroup resharding is required. As shown in Figure 8, we analyze the *DS* to select the appropriate collective operator,

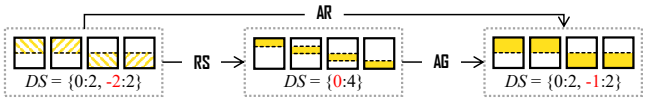


Figure 8: Bottom-tier collective communication.

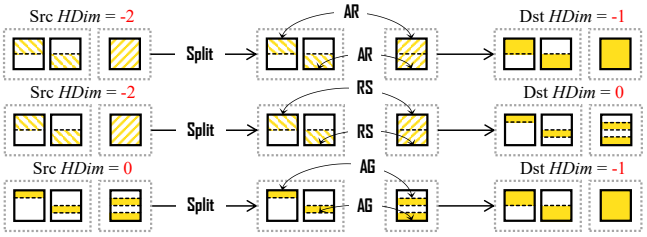


Figure 9: Top-tier collective communication. In scenarios where only *HDim* is altered, each shard can be split to the finest granularity, followed by multiple collective communication operators.

e.g., all-reduce (AR), reduce-scatter (RS), or all-gather (AG).

If these collective communication conditions are unmet, or if the *DG* differs, the system will instead employ batched-send-receive (BSR). Its mechanism will be detailed in §5.3.

5.2 Top-tier Communication

Next, we examine the scenario where the *HSize* of the source matches that of the destination, but their *HDim* values differ. If the *DG Union* of the source is identical to that of the destination (i.e., every *DG* in the union is equivalent), the transformation between the source and the destination can still be efficiently realized using collective communication operators. We analyze the following two cases:

(I) Source *DS Union* equals destination *DS Union*. In this case, only the *HDim* changes between the source and destination. As depicted in Figure 9, the transformation can be achieved through three distinct operators: split-all-reduce (SplitAR), split-reduce-scatter (SplitRS), and split-all-gather (SplitAG). The process starts by identifying the finest-grained slices of all *sharding subgroups*. Subsequently, based on the *HDim* change, collective communication is performed across these *sharding subgroups* for each slice, while maintaining the bottom-tier sharding within each subgroup.

(II) Source *DS Union* does not equal destination *DS Union*. As illustrated in Figure 10, in this case we first use bottom-tier communication (e.g., RS, Identity) to align the *DS Union*, reducing the problem to (I). We then apply top-tier communication (e.g., SplitAG) to align the *HDim*.

For scenarios where the *DG Unions* of the source and the destination are not identical, or where the *HSize* values differ, we once again resort to the BSR operator (§5.3).

5.3 Batched-Send-Receive Mechanism

When it does not involve *Partial*, any complex resharding can be decomposed into multiple send and receive operators, referred to as batched-send-receive (BSR), which is particularly useful for handling complex communication patterns that cannot be supported by collective communications.

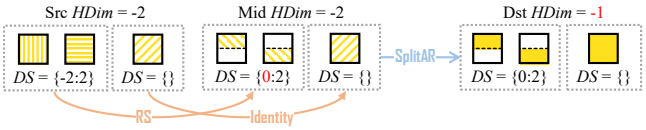


Figure 10: Top-tier collective communication succeeded by bottom-tier communication. In scenarios where both *HDim* and *DS Union* are modified, we first align the *DS* within each *sharding subgroup*, followed by aligning the *HDim* across different *sharding subgroups*.

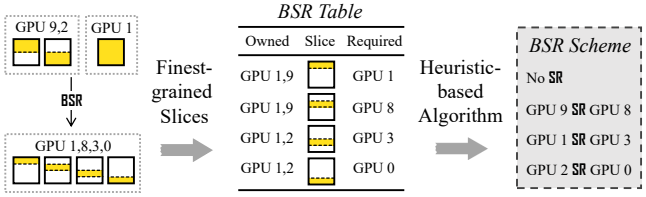


Figure 11: Batched-Send-Receive (BSR) mechanism. Based on the annotations of the source and destination, we construct a *BSR table* and subsequently generate the *BSR scheme* for communication.

As depicted in Figure 11 (left), we begin by identifying the finest-grained slices, based on which we construct a *BSR table* that maps each slice to its owner devices and the devices that require it. Subsequently, a *BSR scheme* is generated to describe how the slices are transmitted. Particularly, as illustrated in Figure 11 (right), we generate the *BSR scheme* by sequentially scanning the *BSR table* and applying the following heuristics: **(i)** If the slice is already owned, we perform a direct copy without communication; **(ii)** We prefer links with higher bandwidth (e.g., GPU 9 sending to GPU 8 via NVLink); **(iii)** If bandwidths are equal, we balance the cumulative send load (e.g., GPU 1 and GPU 2 send alternately).

Discussions. Since each slice may have multiple senders (owner devices) and receivers (required devices), deriving the optimal *BSR scheme* necessitates solving a Generalized Assignment Problem (GAP) [81], which is NP-hard. Our heuristic approach reduces the solving complexity to $O(pq)$, where p is the number of entries in the *BSR table* and q is the maximum number of receivers per entry. Here, p corresponds to the partition granularity of the finest-grained slices (e.g., $p = 4$ in Figure 11) and does not increase with cluster scale, since scaling is typically achieved by replication along the DP dimension rather than finer per-tensor partitioning. As a result, the solver remains tractable even at a large scale.

Our solver models only P2P bandwidth, which may be insufficient under highly heterogeneous network topologies. In such cases, a single P2P transfer may traverse multiple physical links, and the true bottleneck can reside in a specific link along the path. A more fine-grained model that captures per-link characteristics could better expose such intra-path bottlenecks. However, in practice, the BSR overhead is already small relative to computation (see §10), suggesting that further optimizing the solver would yield limited benefits. Overall, though our heuristic-based algorithm may not achieve the optimal, it is more practical and easier to implement, with strong performance and low overhead (see §10).

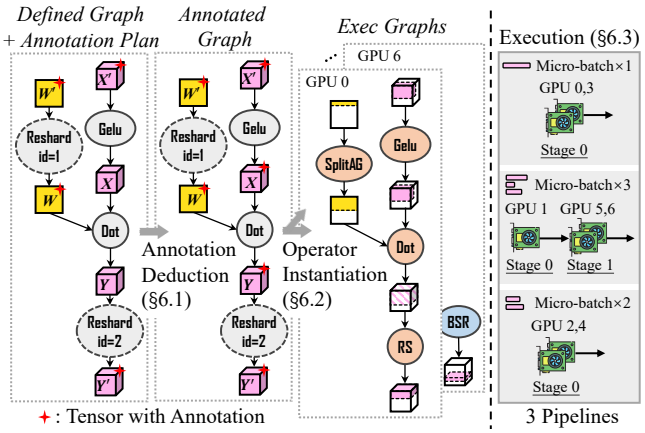


Figure 12: HSPMD 's graph specialization. We systematically deduce and parse annotations from a unified *defined graph*, transforming it into device-specific execution logic to support heterogeneous parallelization. The annotations remain consistent with Figure 4.

Last but not least, BSR is infeasible when *Partial* is involved. However, in practice, *Partial* tensors are typically intermediate results associated with collective communication, which can be handled by other top-tier or bottom-tier communication mechanisms. Given that complex resharding for *Partial* tensors is generally unnecessary, we omit such scenarios, yet our work can be easily extended by integrating BSR with all-reduce (when source involves *Partial*) or numeric division (when destination involves *Partial*).

6 Progressive Graph Specialization

In this section, we detail the abstraction of graph specialization that tackles *spatial* heterogeneity. As shown in Figure 12, the process starts from a *defined graph* with initial annotations specified by the *annotation plan* (i.e., the output sharding of leaf and Reshard operators). We then propagate these annotations to all remaining tensors, resulting in a fully *annotated graph* (§6.1). Next, we resolve the concrete communication of Reshard (as detailed in §5) and instantiate device-specific operators to generate distinct *executable graphs* for each device (§6.2), which are then deployed for pipelined execution (§6.3). Starting from a single-program abstraction, this specialization process incrementally derives divergent execution logics across devices, enabling the expression of heterogeneous parallel strategies that address *spatial* heterogeneity.

6.1 Annotation Deduction

As shown in Figure 12, we start by deducing the unspecified annotations (e.g., the output tensor X of Gelu and the output tensor Y of Dot). For simple unary operators like Gelu, the input tensor's annotation is straightforwardly propagated to the output. However, more complex operators, such as Dot, require a more intricate deduction process. Below, we outline the general methodology of annotation deduction.

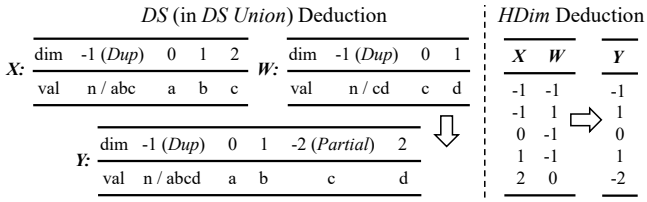


Figure 13: The deduction rules of *DS Union* and *HDim* for a 3D × 2D `Dot` operator. *n* is the number of GPUs in the *sharding subgroup* and *a-c* are values along different sharding dimensions.

Deduction of *DG Union* and *HSize*. For any standard operator, the output tensors inherit the same *DG Union* and *HSize* as the inputs. This requirement stems from a semantic constraint enforced by HSPMD: all inputs and outputs of an operator must have matching *DG Union* and *HSize*, ensuring that the operator executes locally on its assigned device, consuming inputs to produce outputs without implicit cross-device resharding. For multi-input operators with mismatched *DG Union* or *HSize*, users must explicitly insert `Reshard` operators before certain inputs to resolve the discrepancy.

Deduction of *DS Union* and *HDim*. The deduction of *DS Union* and *HDim* depends on the specific characteristics of each operator and is governed by the rules of distributed computation. To illustrate this process, we analyze the `Dot` operator, which takes a 3D tensor X and a 2D tensor W as inputs, as a concrete example. In the *DS Union* deduction process (Figure 13, left), since the *DG Union* and *HSize* are aligned, the deduction simplifies to sequentially deriving the *DS* for each *sharding subgroup*. This mirrors the SPMD deduction, where sharding values for X, W, and Y are derived using straightforward, rule-based logic. As for *HDim* deduction, top-tier sharding is essentially a simplified 1D sharding, following a similar rule-based logic. For example, as shown in Figure 13 (right), if X has a *HDim* of 0 and W has a *HDim* of -1, the output tensor Y retains a *HDim* of 0, preserving the heterogeneous top-tier *Split* after the `Dot` operator.

Discussions. We support annotation deduction for a wide range of operators. As shown in Table 2, for the vast majority, the annotation is propagated from inputs to outputs without modification. Deviations from this default behavior are limited: only `Reshard` may alter *DG Union* and *HSize*, while changes to *DS Union* and *HDim* are confined to a few specific operators (e.g., `Dot`) that inherently transform sharding patterns and thus employ custom deduction logic. This design, where most operators reuse a common propagation procedure, allows HSPMD's deduction to be efficient and scalable.

6.2 Operator Instantiation

Upon obtaining the annotations for all tensors, we instantiate a unique *executable graph* for each device. This process primarily consists of two key steps.

Non-Local operator removal. First, operators whose inputs and outputs do not involve the local device in their *DG Union* are excluded from its *executable graph*. For instance in Figure 12, since no tensors prior to Y' are placed on GPU 6, all operators except `Reshard` (id=2) are removed. This pruning step is essential for enabling pipeline execution (§6.3).

Table 2: Operator annotation deduction behaviors.

Operators	Modified Annotation Attributes
`Reshard`	*DG Union*, *HSize*, *DS Union*, *HDim*
`Dot`, `Sum`, `Reshape`, etc.	*DS Union*, *HDim*
All other operators	—

Reshard instantiation. For all remaining `Reshard` operators, we apply the hierarchical communication resolution (§5) to instantiate them with the exact communication operators.

(I) Bottom-tier communication. For bottom-tier communication, each *sharding subgroup* independently determines its communication operator and instantiates it accordingly: In Figure 12, `Reshard` (id=2) is classified as bottom-tier, leading GPU 0 and GPU 6 (belonging to different *sharding subgroups*) to instantiate it with `RS` and `BSR`, respectively.

(II) Top-tier communication. For operators classified as top-tier, all GPUs in the *DG Union* apply the same instantiation logic: In Figure 12, `Reshard` (id=1) is identified as `SplitAG` and uniformly instantiated on GPU 0–4.

6.3 Execution

At runtime, operators in the *executable graphs* are not invoked directly. Instead, they are organized into pre-/post-processing steps and forward/backward passes, which are repeatedly scheduled in a pipelined fashion. HSPMD supports various scheduling strategies (e.g., GPipe [27], 1F1B [50]) and allows pipelines to independently process different micro-batch counts and sizes (Figure 12, right). While the *annotation plan* defines the sharding patterns and indicates the `Reshard` behaviors (Figure 5, line 1), the concrete shapes of the input data and the intermediate tensors (e.g., tensors to be communicated) are determined dynamically on each device at runtime (Figure 5, line 9), independent of the static *executable graphs*. This decoupling between graph specialization and runtime execution enables flexible pipeline scheduling and supports variable-length inputs. Due to the space constraint, we refer interested readers to Appendix A.2 for more details.

6.4 Discussions

Below we summarize the data structures maintained per device (i.e., GPU). Following the SPMD paradigm, HSPMD runs one process per device, and each process independently maintains its own copy of: **(i)** a device-agnostic *defined graph* of the user program, **(ii)** a device-agnostic *annotation plan*, **(iii)** the corresponding device-specific *executable graph* derived via graph specialization, **(iv)** operator metadata, including the *BSR tables* for all `BSR` operators in its *executable graph*.

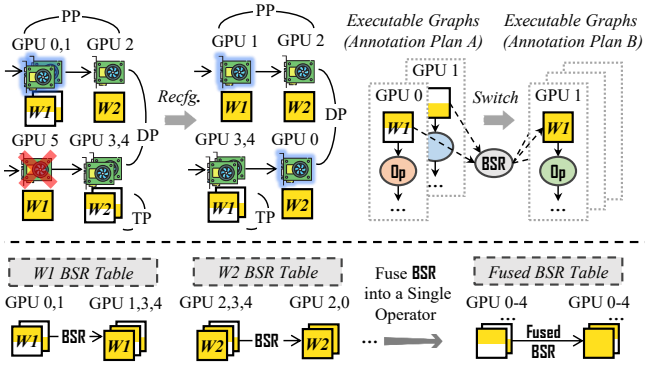


Figure 14: Top: Online reconfiguration is achieved by transforming weight annotations using `BSR` across *executable graphs*. Bottom: Fusing `BSR` into a single operator further enhances efficiency.

Among these, **(i)** and **(ii)** are identical in content across processes, while **(iii)** and **(iv)** are unique to each process. Since these structures encode only parallelization and execution logic (excluding model parameters and activations), each process keeps them in the CPU DRAM of its host. In practice, they occupy only a few MiB per CPU host.

7 Dynamic Graph Switching

HSPMD's graph specialization abstraction addresses *spatial* heterogeneity by generating device-specific parallel execution logic given an *annotation plan*. However, *temporal* heterogeneity, which evolves over time, demands dynamic reconfiguration of the overall parallelization. This necessitates switching from one *annotation plan* to another (Figure 5(b),(c)).

As illustrated in Figure 14 (top), to enable fast reconfiguration, we avoid reloading from checkpoints under the new sharding format. Instead, we reshard the weights from the old *executable graphs* by leveraging the high interconnect bandwidth among devices. Since weight annotations do not involve *Partial*, each weight can be directly resharded using the `BSR` operator (§5.3) to bridge the old and new plans.

To further optimize communication, rather than executing a separate `BSR` operator for each weight, we consolidate all *BSR tables* into a fused one (Figure 14, bottom), so that the corresponding *fused BSR scheme* can balance the communication load across all GPUs. Moreover, we fuse multiple send-receive operators between the same GPU pair into a single operator, significantly reducing kernel launch overhead. By executing this `Fused BSR` operator, the system can therefore efficiently transition to the new *annotation plan* and resume execution without restart or checkpoint reloading.[4]

Discussions. A natural concern is whether maintaining the data structures of different parallel strategies (i.e., multiple plans and graphs) incurs prohibitive overhead as the system scales to large and heterogeneous clusters. The answer is no.

[4]If a slice is missing from the fused *BSR table* (i.e., no GPU currently owns that slice), reconfiguration falls back to checkpoint reloading.

A key observation is that the number of strategies is independent of cluster scale or heterogeneity. For unstable devices, only a single strategy is maintained at any time, as the previous one is discarded after each reconfiguration. For mixed-length data, we pre-generate a set of strategies, each optimized for a specific maximum sequence length (e.g., one for batches with maximum length up to 16K and another up to 32K), and select the appropriate strategy at runtime based on the actual maximum sequence length of each batch. As a result, the number of strategies scales only with the granularity of the maximum-sequence-length partitioning, rather than with the cluster itself. Consequently, the overall maintenance overhead remains bounded regardless of cluster scale or heterogeneity.

8 HSPMD's Planner

This section presents the key implementation aspects of HSPMD's planner, which automatically generates *annotation plans* and thus frees users from constructing them by hand. In HSPMD, the planner is treated as an external component that can be readily replaced by alternative implementations.

Cost model and plan generation. Following existing works on automatic parallelism [48, 72, 92], our default implementation adopts a profiling-based approach, building a *cost model* for each operator on each hardware resource: for example, a compute operator (e.g., `Attention`) on a specific device (e.g., H20 or H800 GPU), or a communication operator (e.g., `AR`, `BSR`) on a given interconnect (e.g., NVLink or IB). We profile representative configurations (e.g., model sizes, tensor parallel degrees, input lengths) and fit cost models. To evaluate a plan, the planner decomposes its dataflow into operators and aggregates their predicted costs. Based on these estimates, it generates optimized *annotation plans* using ILP, MINLP, or dynamic programming, implemented with PuLP [65] and Pyomo [6]. Details are deferred to Appendix A.1.

Overhead of the planner. An *annotation plan* is a JSON dictionary with negligible storage cost. The planning consists of an offline phase (pre-generation) and an online phase (generation or selection). As shown in Figure 5, the offline phase runs before the training loop and is therefore off the critical path, while the online phase depends on the scenario: **(a)** heterogeneous devices require no online planning, **(b)** unstable devices trigger only a single re-generation upon cluster changes; **(c)** mixed-length data uses pre-generated plans with lightweight online selection. The overhead is therefore well controlled, with a detailed breakdown provided in Appendix A.1.

Generalizability to evolving models and clusters. The *cost model* is operator-centric rather than configuration-specific, so it generalizes naturally across model architectures and cluster settings. Model structural properties (e.g., layer count) and cluster factors (e.g., device count and topology) therefore do not affect it. Re-profiling is only required when new operators or new hardware resources are introduced (e.g., a new

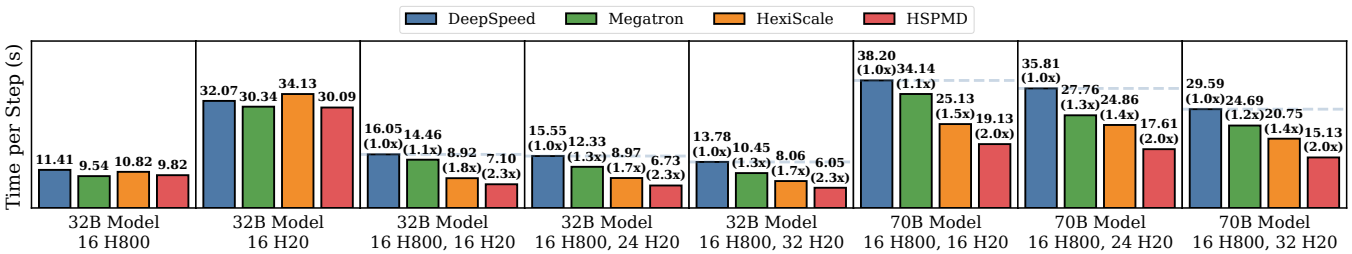


Figure 15: Scenario (a): Training performance across different device configurations and model size.

compute operator, a new GPU type, or a new interconnect).

Sensitivity to annotation plan quality. End-to-end performance is directly influenced by the quality of the *annotation plan*. Nevertheless, a clear lower bound is always guaranteed: in the extreme case, the planner can fall back to a fully homogeneous strategy, under which HSPMD reduces to standard SPMD execution. Since our goal is to establish HSPMD as a general and flexible framework for heterogeneous execution rather than to optimize scenario-specific planners, improving planner optimality is left to future work.

9 Evaluations

In this section, we evaluate HSPMD under three representative scenarios: (a) heterogeneous devices, (b) unstable devices, and (c) mixed-length data (the workflow of each is depicted in §3). Our evaluation aims to answer two central questions:

- How does HSPMD outperform standard SPMD in the presence of diverse *spatial* and *temporal* heterogeneity?
- Why does extending low-level *declarative annotations*, as in HSPMD, provide greater effectiveness than relying on high-level schedulers, as in other SPMD variants?

Baselines. As shown in Table 3, we select two categories of baselines. The first category includes standard SPMD training systems: **(i)** DeepSpeed [28, 58], which supports DP (with ZeRO-series) and SP; **(ii)** Megatron [37, 51, 60], which supports DP (with ZeRO-1), TP, PP, and CP. The second category consists of scenario-specific systems that enhance SPMD with dedicated schedulers: **(iii)** HexiScale [87], a framework built on top of Megatron, employing a heterogeneous GPipe [27] scheduler to support varying pipeline stages and TP degrees across stages; **(iv)** Oobleck [29], an elastic training framework that maintains fault tolerance with pre-defined pipeline templates, and features a scheduler for dynamic reconfiguration by merging and borrowing resources between pipeline templates; **(v)** HotSPa [23], which utilizes a hot-switchable scheduler to reconfigure between several pre-defined homogeneous strategies (DP, TP, PP) to handle sequences of varying lengths, without incurring the overhead of cold-start.

Experimental setup. Our testbed consists of 16 H800 and 32 H20 GPUs (detailed in Appendix C.1). Given that LLMs represent the most significant DL workload today, and that existing baselines predominantly focus on LLMs, our evaluation centers on them as well. We adopt the widely-used Llama

Table 3: Baselines: Two scenario-agnostic SPMD systems, and three scheduler-enhanced SPMD variants designed for specific scenarios.

Systems	Scenarios	Paradigms
(i) DeepSpeed **(ii)** Megatron	(a)-(c)	Standard SPMD
(iii) HexiScale	Only (a)	+ Hetero. Pipeline Scheduler
(iv) Oobleck	Only (b)	+ Elastic Pipeline Scheduler
(v) HotSPa	Only (c)	+ Hot-Switchable Scheduler

architecture [70] in various model sizes, with a default context length of 4K and a global batch size of 64, measuring per-step training time. All baselines are carefully tuned to their optimal performance. For HSPMD, we employ profiling-based planners (detailed in Appendix A.1) to generate and select the *annotation plan*. Strategies are provided in Appendix C.

(a) Heterogeneous devices. We first evaluate performance under static *spatial* heterogeneity through experiments on heterogeneous devices. As shown in Figure 15, on homogeneous devices, all systems perform similarly, confirming that performance differences do not stem from engineering artifacts. On heterogeneous devices, however, HSPMD consistently outperforms the baselines. DeepSpeed and Megatron are constrained by symmetric sharding, which hinders workload balance on devices with varying capabilities. HexiScale, on the other hand, has two key limitations: **(i)** its built-in scheduler tightly couples the heterogeneous parallel strategy specification with execution, making it difficult to flexibly support complex pipeline scheduling schemes like 1F1B; and **(ii)** its scheduler lacks support for generalized asymmetric communication, relying instead on coarse-grained broadcasts. In contrast, HSPMD leverages *declarative annotations*, which decouple heterogeneous specifications from automatic pipeline execution (§6.3) and enable hierarchical communication constructs (§5), thus facilitating flexible pipeline scheduling and efficient communication, resulting in improved performance.

(b) Unstable devices. We next evaluate elastic training performance under *spatial* and unknown *temporal* heterogeneity. Notably, DeepSpeed and Megatron employ a checkpoint-and-restart approach upon reconfiguration and use optimal training strategies under each configuration. Both HSPMD and Oobleck [29] disable ZeRO-1 [58] to retain DP-dimension redundancy for recovery without checkpoint-and-restart. This design is common across fault-tolerant training systems (in-

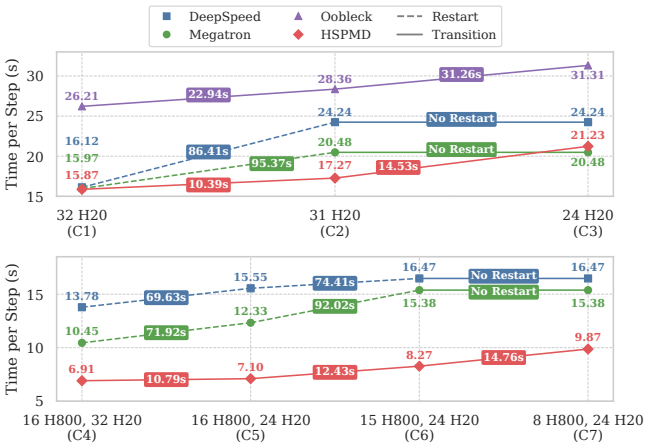


Figure 16: Scenario (b): Elastic training traces on homogeneous and heterogeneous clusters. We annotate the per-step training time for each configuration, along with the reconfiguration overhead.

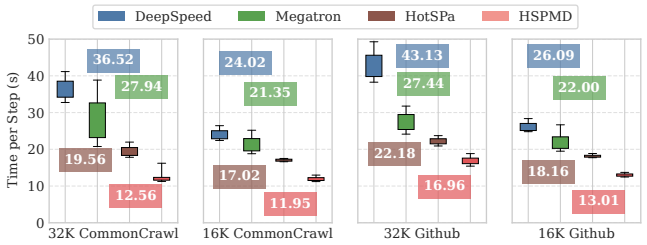


Figure 17: Scenario (c): Training performance with mixed-length data using different context lengths and datasets on 32 H20 GPUs. Box plots show time distributions with mean values annotated.

cluding other recent works like Recycle [21]). Thus, both our generalized system and the specialized baseline inherently trade some training performance for fault tolerance. As shown in Figure 16, we conduct experiments training a 32B model using two traces, each incorporating GPU and node failures.

Upon a GPU failure, DeepSpeed and Megatron are constrained by their symmetric sharding and must discard the entire node (*spatial*). In contrast, HSPMD can leverage all remaining GPUs, leading to improved performance (e.g., C2). Moreover, their significant restart overhead further makes them less suitable for elastic environments (*temporal*).

On the other hand, Oobleck enables restart-free reconfiguration but underperforms compared to other systems because its elastic pipeline scheduler confines fault tolerance strategies to predefined pipeline templates. By comparison, HSPMD employs tensor-level *declarative annotations* rather than rigid pipeline-level templates to specify parallelization, which enables exploration of a much broader strategy space. Furthermore, while Oobleck's elastic scheduler only supports naïve model broadcasting during reconfiguration, HSPMD 's *declarative annotations* allow more general per-tensor send-receive analysis to form a more balanced `Fused BSR` (as evaluated in §10), significantly reducing reconfiguration overhead.

(c) Mixed-length data. Finally, we evaluate the mixed-length data scenario to assess performance under *spatial* and predictable *temporal* heterogeneity. We train a 32B model for 100 steps on CommonCrawl and GitHub datasets using 32 H20 GPUs with a batch size of 200K tokens, testing different context lengths (32K and 16K). As baselines, DeepSpeed and Megatron pack mixed-length sequences into the fixed context window, truncating any excess [38]. HotSPa also employs data packing and additionally switches between pre-defined strategies for different length intervals. Results are shown in Figure 17, while Figure 18 illustrates a specific case: HSPMD pre-generates two heterogeneous strategies—Strategy A, which is slower but handles longer sequences, and Strategy B, optimized for shorter ones. HSPMD dynamically selects and switches between them across training steps (as depicted in Figure 5(c)). All reported per-step timings include strategy reconfiguration overhead for both HotSPa and HSPMD.

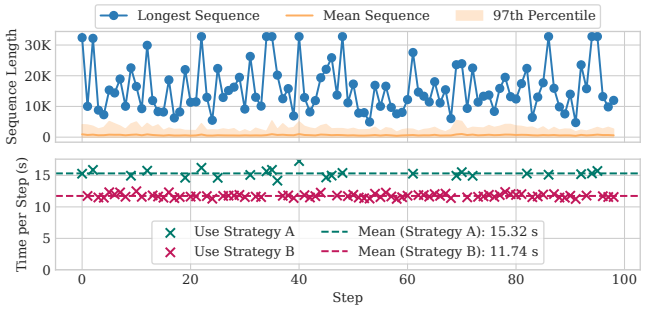


Figure 18: Sequence length variation and strategies employed by HSPMD in the "32K CommonCrawl" case. HSPMD dynamically switches between heterogeneous strategies (A and B) across different steps to optimize performance when sequence lengths shift.

DeepSpeed and Megatron, constrained by the symmetric nature of SPMD to a fixed homogeneous strategy, perform poorly because most sequences are much shorter than the context limit (97% < 8K in Figure 18), making long-context-oriented strategies inefficient for packed short sequences. While HotSPa's hot-switchable scheduler allows dynamic adjustment to length-tailored parallel strategies in the temporal dimension, it remains constrained by the symmetry of SPMD and cannot support spatially heterogeneous parallel strategies. This leads to lower performance compared to HSPMD, which addresses both *spatial* and *temporal* heterogeneity through a unified abstraction of extended *declarative annotations*.

Summary. Our evaluation across diverse scenarios reveals two findings: **(i)** standard SPMD is inadequate for tackling heterogeneity, and **(ii)** scheduler-enhanced SPMD variants can address specific scenarios but have intrinsic limitations. First, compared to primitive-level extensions, addressing heterogeneity at the scheduler level is coarse-grained, which constrains the design space of heterogeneous strategies and the flexibility of strategy reconfiguration (e.g., HexiScale cannot support flexible pipeline schedulers, Oobleck relies on fixed pipeline templates, and HotSPa uses rigid homogeneous strategies), thereby limiting their performance. More importantly, these approaches work around SPMD's symmetry by

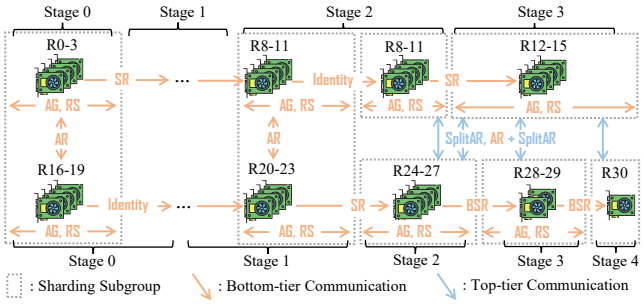


Figure 19: Parallel strategy employed in C2. "R" represents rank.

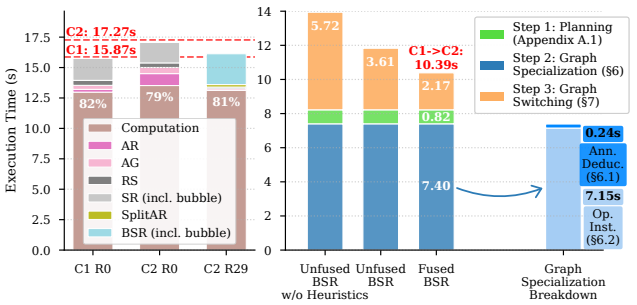


Figure 20: Left: Time per step breakdown for homogeneous (C1) and heterogeneous (C2) parallel strategies. Right: Reconfiguration overhead from C1 to C2 with different `BSR` algorithms.

building a top-down system stack tailored to particular scenarios, making them effective only in isolated cases. In contrast, HSPMD introduces low-level extensions on *declarative annotations*, directly bridging the gap between SPMD's symmetric nature and the asymmetric demands of heterogeneous environments. This enables a versatile, bottom-up system stack that provides general solutions across diverse scenarios.

10 Case Study

In this section, we provide an in-depth analysis of HSPMD's components, using the C2 configuration and the C1-to-C2 reconfiguration process (in Figure 16) as key examples.

Strategy deployment and communication resolution. Figure 19 shows HSPMD 's deployment and communication patterns under C2 (31 H20 GPUs), forming two independent pipelines. Most stages use 4 GPUs, except the last two in the second pipeline, which use 2 and 1 GPU(s). Within each stage, TP is applied via `AG` and `RS` (§5.1(II)). And inter-stage communication within a pipeline employs `SR` (§5.1(I)) or `BSR` (§5.3). For cross-pipeline gradient synchronization, `AR` (§5.1(II)), `SplitAR` (§5.2(I)), and subgroup-specific `AR` following `SplitAR` (§5.2(II)) are used. Besides, we also provide the loss curves of C1 and C2 in Appendix B, which further confirm that the heterogeneous strategy and the introduced communication operators do not affect convergence.

Strategy execution time breakdown. We then analyze the per-step computation and communication time for each rank in C2. Figure 20 (left) contrasts ranks 0 and 29, which follow asymmetric execution logic, against rank 0 in the homogeneous C1 as a reference. The results reveal that: **(i)** the heterogeneous strategy achieves balanced workload distribution across ranks; and **(ii)** similar to the homogeneous strategy, computation remains the dominant runtime component. The additional overhead introduced by `SplitAR` and `BSR` is minimal, indicating that HSPMD's asymmetric communication does not incur significant performance degradation.

Strategy reconfiguration overhead. We eventually evaluate the overhead of reconfiguring from C1 to C2, which consists of three sequential phases (Figure 5(b)): planning (Appendix A.1), graph specialization (§6), and graph switching (§7). As shown in Figure 20 (right), the planner generates

Table 4: Distribution of C1-to-C2 communication volume under different `BSR` algorithms. We show the total communication volume each rank sends via NVLink and InfiniBand (IB).

Format: *Sender Rank ID: NVLink Vol. (MB) | IB Vol. (MB)*

Unfused BSR w/o Heuristics			
R4: 0|748	R5: 0|747	R6: 0|747	R7: 0|747
R8: 0|1495	R9: 0|1495	R10: 0|1495	R11: 0|1495
R12: 0|5517	R13: 0|11496	R14: 0|8222	R15: 0|11496
Fused BSR			
R8: 0|747	R9: 0|747	R10: 0|747	R11: 0|747
R15: 0|11496	R20: 748|0	R21: 748|0	R22: 748|0
R23: 748|0	R24: 0|748	R25: 0|748	R26: 0|748
R27: 0|748	R28: 5517|0	R29: 11496|0	R30: 8222|0

the *annotation plan* with minimal latency. Graph specialization then follows, dominated by operator instantiation (§6.2), which involves adjusting the graph and creating new CCL communication groups but typically completes within 10s, while the cost of annotation deduction (§6.1) is negligible.

As for graph switching, we compare three `BSR` algorithms: **(i)** A baseline without heuristics (using minimal rank IDs for broadcasting); **(ii)** Non-fused per-tensor `BSR` with heuristics; **(iii)** Our `fused BSR`. Results show that our approach achieves the lowest end-to-end switching overhead despite maintaining the same total communication volume. As shown in Table 4, this improvement stems from evenly distributed traffic across ranks and better utilization of high-bandwidth NVLink.

11 Discussions and Future Work

In this section, we discuss two additional aspects of HSPMD: **(i)** its support for Mixture-of-Experts (MoE) models under expert imbalance, and **(ii)** its interoperability with existing SPMD frameworks. Both represent promising directions enabled by HSPMD's design, which we leave to future work.

Support for MoE and expert imbalance. MoE models are typically trained with expert parallelism (EP) [39], which shards experts across devices, but they often suffer from load imbalance due to skewed token routing. A representative mitigation is EPLB [14], which replicates heavily loaded experts to distribute the workload. HSPMD naturally supports such strategies through its two-tier sharding annotations: at the

top tier, HSPMD applies *Split* on the hidden-size dimension to split across different experts, and at the bottom tier, it uses different *Dup* degrees to replicate different loaded experts. When expert workloads shift substantially during training, HSPMD can reconfigure the overall *annotation plan* via graph switching, without requiring any further extension.

Interoperability with existing SPMD frameworks. Any prior annotation-based SPMD framework (e.g., Alpa [92], GSPMD [86], and DTensor [67, 68]) can be extended to support HSPMD's annotations by adding one additional tier on top of its existing SPMD annotations, which amounts to introducing two extra attributes (*HDim* and *HSize*) and promoting the original annotation structure to a list of such structures. On top of this, HSPMD's communication resolution, graph specialization, and graph switching mechanisms can be plugged in to enable heterogeneous parallelization, while the framework's existing parallelization primitives (e.g., operator-level sharding rules, automatic annotation deduction, and its standard parallelism implementations such as DP, TP, PP, and CP) can be reused as-is. By contrast, frameworks that realize parallelism by specifying particular layers, such as Megatron [60] and DeepSpeed [58], are harder to build HSPMD on top of, since they do not expose a unified, operator-level annotation surface and instead rely on ad-hoc, per-layer specifications (e.g., `ColumnParallelLinear`).

12 Related Work

12.1 Tackling Workload Heterogeneity

Recent works have tackled workload heterogeneity in large-scale DL training through scenario-specific optimizations.

(a) Heterogeneous devices. HAP [90] uses a sharding ratio to handle uneven TP partitioning. Whale [31] partitions workloads by splitting the global graph into heterogeneous task graphs. AMP [41] and HetHub [84] support heterogeneous layer partitioning in PP, while Metis [71] extends Alpa [92] for heterogeneous strategy search. HexiScale [87] enables more fine-grained heterogeneous 3D parallelism, and Sailor [63] further optimizes the search cost for such strategies.

(b) Unstable devices. Varuna [3] uses checkpoint restarts and adjusts DP and PP to find optimal strategies after failures. Bamboo [69] achieves fault tolerance via redundant storage on adjacent pipeline stages, Oobleck [29] employs multiple pipeline templates for fault tolerance and strategy tuning, and ReCycle [21] rebalances the workload from failed pipelines.

(c) Mixed-length data. FlexSP [76] uses dynamic SP for varied sequences, while ByteScale [22] and DCP [33] adopt dynamic CP. HotSPa [23] assigns homogeneous strategies to different length intervals. WLB-LLM [78] addresses load imbalance specifically in CP and PP, while Zeppelin [7] achieves finer-grained control over attention and NIC imbalance.

In contrast to these specialized solutions, HSPMD introduces a unified abstraction via low-level *declarative annotations* instead of high-level schedulers. This design offers greater generality and could serve as a foundational substrate for addressing diverse heterogeneity. Furthermore, prior works have proposed planning algorithms tailored to targeted scenarios, which can be seamlessly integrated with our work by encoding their strategies through our annotations.

12.2 SPMD and MPMD Training Systems

Beyond the SPMD training systems discussed above, another line of work adopts the MPMD paradigm, such as Pathways [4], JaxPP [82], and Piper [20]. One motivation behind their MPMD design is the observation that pipeline parallelism (PP) is not inherently SPMD, as different devices naturally execute different model stages. Nevertheless, SPMD remains the dominant paradigm in mainstream training systems for its scalability and simplicity (see §2.3), and PP is typically encoded in an SPMD style, where devices share a global program but diverge at runtime. For example, GSPMD [86] treats PP as layer-wise sharding, while Megatron [60] employs a light scheduler to execute subgraphs. HexiScale [87] and Oobleck [29] further introduce more advanced schedulers for heterogeneous execution. These approaches effectively encode PP within SPMD. HSPMD takes the same stance: rather than switching to MPMD, it extends SPMD with *declarative annotations* for asymmetry, preserving single-program simplicity while enabling the heterogeneous execution of various parallelism schemes including PP (see §6.3).

13 Conclusion

We present HSPMD, a novel system that extends the SPMD paradigm to support heterogeneous parallelization and dynamic reconfiguration, addressing both *spatial* and *temporal* heterogeneity. Through low-level sharding annotations and hierarchical communication, HSPMD bridges the gap between SPMD's inherent symmetric constraints and the emerging demands of asymmetric execution in large-scale DL training. Extensive evaluation across (a) heterogeneous devices, (b) unstable devices, and (c) mixed-length data demonstrates HSPMD's effectiveness and broad applicability.

Acknowledgments

This work is supported by National Natural Science Foundation of China (U23B2048, 62402011), Fundamental and Interdisciplinary Disciplines Breakthrough Plan of the Ministry of Education of China (JYB2025XDXM108), and High-performance Computing Platform of Peking University. Fangcheng Fu and Bin Cui are the corresponding authors.

References

[1] Y. Ajima, T. Kawashima, T. Okamoto, N. Shida, K. Hirai, T. Shimizu, S. Hiramoto, Y. Ikeda, T. Yoshikawa, K. Uchida, and T. Inoue. The tofu interconnect d. In *IEEE International Conference on Cluster Computing (CLUSTER)*, pages 646–654. IEEE, 2018.

[2] Argonne Leadership Computing Facility. Thetagpu, 2024.

[3] Sanjith Athlur, Nitika Saran, Muthian Sivathanu, Ramachandran Ramjee, and Nipun Kwatra. Varuna: scalable, low-cost training of massive deep learning models. In *Proceedings of the Seventeenth European Conference on Computer Systems*, EuroSys '22, page 472–487, New York, NY, USA, 2022. Association for Computing Machinery.

[4] Paul Barham, Aakanksha Chowdhery, Jeff Dean, Sanjay Ghemawat, Steven Hand, Daniel Hurt, Michael Isard, Hyeontaek Lim, Ruoming Pang, Sudip Roy, Brennan Saeta, Parker Schuh, Ryan Sepassi, Laurent Shafey, Chandu Thekkath, and Yonghui Wu. Pathways: Asynchronous distributed dataflow for ml. In D. Marculescu, Y. Chi, and C. Wu, editors, *Proceedings of Machine Learning and Systems*, volume 4, pages 430–449, 2022.

[5] William Brandon, Aniruddha Nrusimha, Kevin Qian, Zachary Ankner, Tian Jin, Zhiye Song, and Jonathan Ragan-Kelley. Striped attention: Faster ring attention for causal transformers, 2023.

[6] Michael L Bynum, Gabriel A Hackebeil, William E Hart, Carl D Laird, Bethany L Nicholson, John D Siirola, Jean-Paul Watson, David L Woodruff, et al. *Pyomo-optimization modeling in python*, volume 67. Springer, 2021.

[7] Chang Chen, Tiancheng Chen, Jiangfei Duan, Qianchao Zhu, Zerui Wang, Qinghao Hu, Peng Sun, Xiuhong Li, Chao Yang, and Torsten Hoefler. Zeppelin: Balancing variable-length workloads in data parallel large model training, 2025.

[8] Qiaoling Chen, Diandian Gu, Guoteng Wang, Xun Chen, YingTong Xiong, Ting Huang, Qinghao Hu, Xin Jin, Yonggang Wen, Tianwei Zhang, and Peng Sun. Internevo: Efficient long-sequence large language model training via hybrid parallelism and redundant sharding, 2024.

[9] Qiaoling Chen, Qinghao Hu, Guoteng Wang, Yingtong Xiong, Ting Huang, Xun Chen, Yang Gao, Hang Yan, Yonggang Wen, Tianwei Zhang, and Peng Sun. Amsp: Reducing communication overhead of zero for efficient llm training, 2024.

[10] Rohan Choudhury, Guanglei Zhu, Sihan Liu, Koichiro Niinuma, Kris M. Kitani, and Laszlo Attila Jeni. Don't look twice: Faster video transformers with run-length tokenization. In *The Thirty-eighth Annual Conference on Neural Information Processing Systems*, 2024.

[11] Tri Dao. Flashattention-2: Faster attention with better parallelism and work partitioning, 2023.

[12] Tri Dao, Daniel Y. Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. Flashattention: Fast and memory-efficient exact attention with io-awareness. In *Advances in Neural Information Processing Systems (NeurIPS)*.

[13] Daniele De Sensi, Tommaso Bonato, David Saam, and Torsten Hoefler. Swing: short-cutting rings for higher bandwidth allreduce. In *Proceedings of the 21st USENIX Symposium on Networked Systems Design and Implementation*, NSDI'24, USA, 2024. USENIX Association.

[14] DeepSeek-AI. EPLB: Expert parallelism load balancer. https://github.com/deepseek-ai/EPLB, 2025.

[15] DeepSeek-AI, Aixin Liu, Bei Feng, Bing Xue, et al. Deepseek-v3 technical report, 2025.

[16] Mostafa Dehghani, Basil Mustafa, Josip Djolonga, Jonathan Heek, Matthias Minderer, Mathilde Caron, Andreas Peter Steiner, Joan Puigcerver, Robert Geirhos, Ibrahim Alabdulmohsin, Avital Oliver, Piotr Padlewski, Alexey A. Gritsenko, Mario Lucic, and Neil Houlsby. Patch n' pack: Navit, a vision transformer for any aspect ratio and resolution. In *Thirty-seventh Conference on Neural Information Processing Systems*, 2023.

[17] Fei Du, Xin-Jian Ma, Jing-Ru Yang, Yi Liu, Chao-Ran Luo, Xue-Bin Wang, Hai-Ou Jiang, and Xiang Jing. A survey of llm datasets: From autoregressive model to ai chatbot. *Journal of Computer Science and Technology*, 39(3):542–566, 2024.

[18] Abhimanyu Dubey, Abhinav Jauhri, et al. The llama 3 herd of models, 2024.

[19] Enfabrica. Scaling to 100k+ gpu ai clusters using flat 2-tier network designs, 2024.

[20] Megan Frisella, Arvin Oentoro, Xiangyu Gao, Gilbert Bernstein, and Stephanie Wang. Piper: Towards flexible pipeline parallelism for pytorch. In *Proceedings of the 4th Workshop on Practical Adoption Challenges of ML for Systems*, PACMI '25, page 1–6, New York, NY, USA, 2025. Association for Computing Machinery.

[21] Swapnil Gandhi, Mark Zhao, Athinagoras Skiadopoulos, and Christos Kozyrakis. Recycle: Resilient training of large dnns using pipeline adaptation. In *Proceedings*

of the ACM SIGOPS 30th Symposium on Operating Systems Principles, SOSP '24, page 211–228, New York, NY, USA, 2024. Association for Computing Machinery.

[22] Hao Ge, Junda Feng, Qi Huang, Fangcheng Fu, Xiaonan Nie, Lei Zuo, Haibin Lin, Bin Cui, and Xin Liu. Bytescale: Communication-efficient scaling of llm training with a 2048k context length on 16384 gpus. In *Proceedings of the ACM SIGCOMM 2025 Conference*, SIGCOMM '25, page 963–978, New York, NY, USA, 2025. Association for Computing Machinery.

[23] Hao Ge, Fangcheng Fu, Haoyang Li, Xuanyu Wang, Sheng Lin, Yujie Wang, Xiaonan Nie, Hailin Zhang, Xupeng Miao, and Bin Cui. Enabling parallelism hot switching for efficient training of large language models. In *Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles*, SOSP '24, page 178–194, New York, NY, USA, 2024. Association for Computing Machinery.

[24] Lei Guan, Dong-Sheng Li, Ji-Ye Liang, Wen-Jian Wang, Ke-Shi Ge, and Xi-Cheng Lu. Advances of pipeline model parallelism for deep learning training: An overview. *Journal of Computer Science and Technology*, 39(3):567–584, 2024.

[25] Runsheng Benson Guo, Utkarsh Anand, Arthur Chen, and Khuzaima Daudjee. Cephalo: Harnessing heterogeneous gpu clusters for training transformer models. In *Proceedings of the 39th ACM International Conference on Supercomputing*, ICS '25, page 368–383, New York, NY, USA, 2025. Association for Computing Machinery.

[26] Mert Hidayetoglu, Simon Garcia de Gonzalo, Elliott Slaughter, Pinku Surana, Wen mei Hwu, William Gropp, and Alex Aiken. Hiccl: A hierarchical collective communication library, 2024.

[27] Yanping Huang, Youlong Cheng, Ankur Bapna, Orhan Firat, Mia Xu Chen, Dehao Chen, HyoukJoong Lee, Jiquan Ngiam, Quoc V. Le, Yonghui Wu, and Zhifeng Chen. *GPipe: efficient training of giant neural networks using pipeline parallelism*. Curran Associates Inc., Red Hook, NY, USA, 2019.

[28] Sam Ade Jacobs, Masahiro Tanaka, Chengming Zhang, Minjia Zhang, Shuaiwen Leon Song, Samyam Rajbhandari, and Yuxiong He. Deepspeed ulysses: System optimizations for enabling training of extreme long sequence transformer models, 2023.

[29] Insu Jang, Zhenning Yang, Zhen Zhang, Xin Jin, and Mosharaf Chowdhury. Oobleck: Resilient distributed training of large models using pipeline templates. In *Proceedings of the 29th Symposium on Operating Systems Principles*, SOSP '23, page 382–395, New York, NY, USA, 2023. Association for Computing Machinery.

[30] Myeongjae Jeon, Shivaram Venkataraman, Amar Phanishayee, Junjie Qian, Wencong Xiao, and Fan Yang. Analysis of Large-Scale Multi-Tenant GPU clusters for DNN training workloads. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 947–960, Renton, WA, July 2019. USENIX Association.

[31] Xianyan Jia, Le Jiang, Ang Wang, Wencong Xiao, Ziji Shi, Jie Zhang, Xinyuan Li, Langshi Chen, Yong Li, Zhen Zheng, Xiaoyong Liu, and Wei Lin. Whale: Efficient giant model training over heterogeneous GPUs. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, pages 673–688, Carlsbad, CA, July 2022. USENIX Association.

[32] Zhihao Jia, Matei Zaharia, and Alex Aiken. Beyond data and model parallelism for deep neural networks. In A. Talwalkar, V. Smith, and M. Zaharia, editors, *Proceedings of Machine Learning and Systems*, volume 1, pages 1–13, 2019.

[33] Chenyu Jiang, Zhenkun Cai, Ye Tian, Zhen Jia, Yida Wang, and Chuan Wu. Dcp: Addressing input dynamism in long-context training via dynamic context parallelism. In *Proceedings of the ACM SIGOPS 31st Symposium on Operating Systems Principles*, SOSP '25, page 221–236, New York, NY, USA, 2025. Association for Computing Machinery.

[34] Jared Kaplan, Sam McCandlish, Tom Henighan, Tom B. Brown, Benjamin Chess, Rewon Child, Scott Gray, Alec Radford, Jeffrey Wu, and Dario Amodei. Scaling laws for neural language models, 2020.

[35] Taebum Kim, Hyoungjoo Kim, Gyeong-In Yu, and Byung-Gon Chun. Bpipe: memory-balanced pipeline parallelism for training large language models. In *Proceedings of the 40th International Conference on Machine Learning*, ICML'23. JMLR.org, 2023.

[36] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. In *3rd International Conference on Learning Representations (ICLR)*, 2015.

[37] Vijay Korthikanti, Jared Casper, Sangkug Lym, Lawrence McAfee, Michael Andersch, Mohammad Shoeybi, and Bryan Catanzaro. Reducing activation recomputation in large transformer models, 2022.

[38] Mario Michael Krell, Matej Kosec, Sergio P. Perez, and Andrew Fitzgibbon. Efficient sequence packing without cross-contamination: Accelerating large language models without impacting performance, 2022.

[39] Dmitry Lepikhin, HyoukJoong Lee, Yuanzhong Xu, Dehao Chen, Orhan Firat, Yanping Huang, Maxim Krikun,

Noam Shazeer, and Zhifeng Chen. Gshard: Scaling giant models with conditional computation and automatic sharding, 2020.

[40] Baojia Li, Xiaoliang Wang, Jingzhu Wang, Yifan Liu, Yuanyuan Gong, Hao Lu, Weizhen Dang, Weifeng Zhang, Xiaojie Huang, Mingzhuo Chen, Jie Chen, Chunzhi He, Yadong Liu, Xiaoyuan Hu, Chen Liu, Xuefeng Ji, Yinben Xia, Xiang Li, Zekun He, Yachen Wang, and Xianneng Zou. Tccl: Co-optimizing collective communication and traffic routing for gpu-centric clusters. In *Proceedings of the 2024 SIGCOMM Workshop on Networks for AI Computing*, NAIC '24, page 48–53, New York, NY, USA, 2024. Association for Computing Machinery.

[41] Dacheng Li, Hongyi Wang, Eric Xing, and Hao Zhang. Amp: automatically finding model parallel strategies with heterogeneity awareness. In *Proceedings of the 36th International Conference on Neural Information Processing Systems*, NIPS '22, Red Hook, NY, USA, 2022. Curran Associates Inc.

[42] Haoyang Li, Fangcheng Fu, Hao Ge, Sheng Lin, Xuanyu Wang, Jiawen Niu, Yujie Wang, Hailin Zhang, Xiaonan Nie, and Bin Cui. Malleus: Straggler-resilient hybrid parallel training of large-scale models via malleable data and model parallelization. *Proc. ACM Manag. Data*, 3(3), June 2025.

[43] Haoyang Li, Fangcheng Fu, Sheng Lin, Hao Ge, Xuanyu Wang, Jiawen Niu, Jinbao Xue, Yangyu Tao, Di Wang, Jie Jiang, and Bin Cui. Hydraulis: Balancing large transformer model training via co-designing parallel strategies and data assignment. *Proc. ACM Manag. Data*, 3(6), December 2025.

[44] Haoyang Li, Sheng Lin, Fangcheng Fu, Yuming Zhou, Xiaodong Ji, Yanfeng Zhao, Lefeng Wang, Jie Jiang, and Bin Cui. Unleashing efficient asynchronous rl post-training via staleness-constrained rollout coordination, 2026.

[45] Shen Li, Yanli Zhao, Rohan Varma, Omkar Salpekar, Pieter Noordhuis, Teng Li, Adam Paszke, Jeff Smith, Brian Vaughan, Pritam Damania, and Soumith Chintala. Pytorch distributed: Experiences on accelerating data parallel training, 2020.

[46] Sheng Lin, Fangcheng Fu, Haoyang Li, Hao Ge, Xuanyu Wang, Jiawen Niu, Yaofeng Tu, and Bin Cui. Lobra: Multi-tenant fine-tuning over heterogeneous data. *Proceedings of the VLDB Endowment*, 18(8):2616–2625, April 2025.

[47] Hao Liu, Matei Zaharia, and Pieter Abbeel. Ringattention with blockwise transformers for near-infinite context. In *The Twelfth International Conference on Learning Representations*, 2024.

[48] Xupeng Miao, Yujie Wang, Youhe Jiang, Chunan Shi, Xiaonan Nie, Hailin Zhang, and Bin Cui. Galvatron: Efficient transformer training over multiple gpus using automatic parallelism. *Proc. VLDB Endow.*, 16(3):470–479, November 2022.

[49] Philipp Moritz, Robert Nishihara, Stephanie Wang, Alexey Tumanov, Richard Liaw, Eric Liang, Melih Elibol, Zongheng Yang, William Paul, Michael I. Jordan, and Ion Stoica. Ray: a distributed framework for emerging ai applications. In *Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation*, OSDI'18, page 561–577, USA, 2018. USENIX Association.

[50] Deepak Narayanan, Aaron Harlap, Amar Phanishayee, Vivek Seshadri, Nikhil R. Devanur, Gregory R. Ganger, Phillip B. Gibbons, and Matei Zaharia. Pipedream: generalized pipeline parallelism for dnn training. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP '19, page 1–15, New York, NY, USA, 2019. Association for Computing Machinery.

[51] Deepak Narayanan, Mohammad Shoeybi, Jared Casper, Patrick LeGresley, Mostofa Patwary, Vijay Anand Korthikanti, Dmitri Vainbrand, Prethvi Kashinkunti, Julie Bernauer, Bryan Catanzaro, Amar Phanishayee, and Matei Zaharia. Efficient large-scale language model training on gpu clusters using megatron-lm, 2021.

[52] NVIDIA. cublas, 2025.

[53] NVIDIA. cutlass, 2025.

[54] NVIDIA. Nvidia collective communications library (nccl), 2025.

[55] OpenAI, Josh Achiam, Steven Adler, et al. Gpt-4 technical report, 2024.

[56] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Köpf, Edward Z. Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library. In *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, December 8-14, 2019, Vancouver, BC, Canada*, pages 8024–8035, 2019.

[57] Penghui Qi, Xinyi Wan, Guangxing Huang, and Min Lin. Zero bubble pipeline parallelism, 2023.

[58] Samyam Rajbhandari, Jeff Rasley, Olatunji Ruwase, and Yuxiong He. Zero: Memory optimizations toward training trillion parameter models, 2020.

[59] Guangming Sheng, Chi Zhang, Zilingfeng Ye, Xibin Wu, Wang Zhang, Ru Zhang, Yanghua Peng, Haibin Lin, and Chuan Wu. Hybridflow: A flexible and efficient rlhf framework. In *Proceedings of the Twentieth European Conference on Computer Systems*, EuroSys '25, page 1279–1297, New York, NY, USA, 2025. Association for Computing Machinery.

[60] Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. Megatron-lm: Training multi-billion parameter language models using model parallelism, 2020.

[61] Linghao Song, Jiachen Mao, Youwei Zhuo, Xuehai Qian, Hai Li, and Yiran Chen. Hypar: Towards hybrid parallelism for deep learning accelerator array. In *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 56–68, 2019.

[62] Foteini Strati, Paul Elvinger, Tolga Kerimoglu, and Ana Klimovic. Ml training with cloud gpu shortages: Is cross-region the answer? In *Proceedings of the 4th Workshop on Machine Learning and Systems*, EuroMLSys '24, page 107–116, New York, NY, USA, 2024. Association for Computing Machinery.

[63] Foteini Strati, Zhendong Zhang, George Manos, Ixeia Sánchez Périz, Qinghao Hu, Tiancheng Chen, Berk Buzcu, Song Han, Pamela Delgado, and Ana Klimovic. Sailor: Automating distributed training over dynamic, heterogeneous, and geo-distributed clusters. In *Proceedings of the ACM SIGOPS 31st Symposium on Operating Systems Principles*, SOSP '25, page 204–220, New York, NY, USA, 2025. Association for Computing Machinery.

[64] Gemini Team, Rohan Anil, Sebastian Borgeaud, Jean-Baptiste Alayrac, et al. Gemini: A family of highly capable multimodal models, 2024.

[65] The PuLP Team. Optimization with pulp, 2009.

[66] The Pybind Team. pybind, 2025.

[67] The PyTorch Team. Distributedtensor – pytorch documentation, 2025. PyTorch stable documentation (v2.5).

[68] The TensorFlow Team. Dtensor overview – tensorflow documentation, 2025. TensorFlow Documentation (v2.16).

[69] John Thorpe, Pengzhan Zhao, Jonathan Eyolfson, Yifan Qiao, Zhihao Jia, Minjia Zhang, Ravi Netravali, and Guoqing Harry Xu. Bamboo: Making preemptible instances resilient for affordable training of large DNNs. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 497–513, Boston, MA, April 2023. USENIX Association.

[70] Hugo Touvron, Louis Martin, Kevin Stone, et al. Llama 2: Open foundation and fine-tuned chat models, 2023.

[71] Taegeon Um, Byungsoo Oh, Minyoung Kang, Woo-Yeon Lee, Goeun Kim, Dongseob Kim, Youngtaek Kim, Mohd Muzzammil, and Myeongjae Jeon. Metis: Fast automatic distributed training on heterogeneous GPUs. In *2024 USENIX Annual Technical Conference (USENIX ATC 24)*, pages 563–578, Santa Clara, CA, July 2024. USENIX Association.

[72] Colin Unger, Zhihao Jia, Wei Wu, Sina Lin, Mandeep Baines, Carlos Efrain Quintero Narvaez, Vinay Ramakrishnaiah, Nirmal Prajapati, Pat McCormick, Jamaludin Mohd-Yusof, Xi Luo, Dheevatsa Mudigere, Jongsoo Park, Misha Smelyanskiy, and Alex Aiken. Unity: Accelerating DNN training through joint optimization of algebraic transformations and parallelization. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 267–284, Carlsbad, CA, July 2022. USENIX Association.

[73] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Advances in Neural Information Processing Systems (NeurIPS)*, pages 5998–6008, 2017.

[74] Xuanyu Wang, Fangcheng Fu, Haoyang Li, Hao Ge, Sheng Lin, Jiawen Niu, and Bin Cui. Elastor: Elastic and efficient model partitioning and checkpointing for fault-tolerant distributed training. In *Proceedings of the 31st ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming*, PPoPP '26, page 398–412, New York, NY, USA, 2026. Association for Computing Machinery.

[75] Yujie Wang, Youhe Jiang, Xupeng Miao, Fangcheng Fu, Shenhan Zhu, Xiaonan Nie, Yaofeng Tu, and Bin Cui. Improving automatic parallel training via balanced memory workload optimization. *IEEE Transactions on Knowledge and Data Engineering*, 36(8):3906–3920, August 2024.

[76] Yujie Wang, Shiju Wang, Shenhan Zhu, Fangcheng Fu, Xinyi Liu, Xuefeng Xiao, Huixia Li, Jiashi Li, Faming Wu, and Bin Cui. Flexsp: Accelerating large language model training via flexible sequence parallelism. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, ASPLOS '25, page 421–436, New York, NY, USA, 2025. Association for Computing Machinery.

[77] Yujie Wang, Shenhan Zhu, Fangcheng Fu, Xupeng Miao, Jie Zhang, Juan Zhu, Fan Hong, Yong Li, and Bin Cui. Spindle: Efficient distributed training of multi-task large models via wavefront scheduling. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2025)*, pages 1139–1155. ACM, 2025.

[78] Zheng Wang, Anna Cai, Xinfeng Xie, Zaifeng Pan, Yue Guan, Weiwei Chu, Jie Wang, Shikai Li, Jianyu Huang, Chris Cai, Yuchen Hao, and Yufei Ding. Wlb-llm: workload-balanced 4d parallelism for large language model training. In *Proceedings of the 19th USENIX Conference on Operating Systems Design and Implementation*, OSDI '25, USA, 2025. USENIX Association.

[79] Qizhen Weng, Wencong Xiao, Yinghao Yu, Wei Wang, Cheng Wang, Jian He, Yong Li, Liping Zhang, Wei Lin, and Yu Ding. MLaaS in the wild: Workload analysis and scheduling in Large-Scale heterogeneous GPU clusters. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, pages 945–960, Renton, WA, April 2022. USENIX Association.

[80] Wikipedia. Clos network, 2025.

[81] Wikipedia. Generalized assignment problem, 2025.

[82] Anxhelo Xhebraj, Sean Lee, Hanfeng Chen, and Vinod Grover. Scaling deep learning training with mpmd pipeline parallelism, 2024.

[83] Zhiheng Xi, Wenxiang Chen, Xin Guo, Wei He, Yiwen Ding, Boyang Hong, Ming Zhang, Junzhe Wang, Senjie Jin, Enyu Zhou, Rui Zheng, Xiaoran Fan, Xiao Wang, Limao Xiong, Yuhao Zhou, Weiran Wang, Changhao Jiang, Yicheng Zou, Xiangyang Liu, Zhangyue Yin, Shihan Dou, Rongxiang Weng, Wenjuan Qin, Yongyan Zheng, Xipeng Qiu, Xuanjing Huang, Qi Zhang, and Tao Gui. The rise and potential of large language model based agents: a survey. *Science China Information Sciences*, 68(2):121101, Jan 2025.

[84] Si Xu, Zixiao Huang, Yan Zeng, Shengen Yan, Xuefei Ning, Quanlu Zhang, Haolin Ye, Sipei Gu, Chunsheng Shui, Zhezheng Lin, Hao Zhang, Sheng Wang, Guohao Dai, and Yu Wang. Hethub: A distributed training system with heterogeneous cluster for large-scale models, 2024.

[85] Weikai Xu, Chengrui Huang, Shen Gao, and Shuo Shang. Llm-based agents for tool learning: A survey. *Data Science and Engineering*, Jun 2025.

[86] Yuanzhong Xu, HyoukJoong Lee, Dehao Chen, Blake Hechtman, Yanping Huang, Rahul Joshi, Maxim Krikun, Dmitry Lepikhin, Andy Ly, Marcello Maggioni, Ruoming Pang, Noam Shazeer, Shibo Wang, Tao Wang, Yonghui Wu, and Zhifeng Chen. Gspmd: General and scalable parallelization for ml computation graphs, 2021.

[87] Ran Yan, Youhe Jiang, Xiaonan Nie, Fangcheng Fu, Bin Cui, and Binhang Yuan. Hexiscale: Accommodating large language model training over heterogeneous environment, 2025.

[88] Zongheng Yang, Zhanghao Wu, Michael Luo, Wei-Lin Chiang, Romil Bhardwaj, Woosuk Kwon, Siyuan Zhuang, Frank Sifei Luan, Gautam Mittal, Scott Shenker, and Ion Stoica. SkyPilot: An intercloud broker for sky computing. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 437–455, Boston, MA, April 2023. USENIX Association.

[89] Huangzhao Zhang, Kechi Zhang, Zhuo Li, Jia Li, Yongmin Li, Yunfei Zhao, Yuqi Zhu, Fang Liu, Ge Li, and Zhi Jin. Deep learning for code generation: a survey. *Science China Information Sciences*, 67(9):191101, Aug 2024.

[90] Shiwei Zhang, Lansong Diao, Chuan Wu, Zongyan Cao, Siyu Wang, and Wei Lin. Hap: Spmd dnn training on heterogeneous gpu clusters with automated program synthesis. In *Proceedings of the Nineteenth European Conference on Computer Systems*, EuroSys '24, page 524–541, New York, NY, USA, 2024. Association for Computing Machinery.

[91] Yanli Zhao, Andrew Gu, Rohan Varma, Liang Luo, Chien-Chin Huang, Min Xu, Less Wright, Hamid Shojanazeri, Myle Ott, Sam Shleifer, Alban Desmaison, Can Balioglu, Pritam Damania, Bernard Nguyen, Geeta Chauhan, Yuchen Hao, Ajit Mathews, and Shen Li. Pytorch fsdp: Experiences on scaling fully sharded data parallel, 2023.

[92] Lianmin Zheng, Zhuohan Li, Hao Zhang, Yonghao Zhuang, Joseph E. Gonzalez, and Ion Stoica. Alpa: Automating inter- and intra-operator parallelism for distributed deep learning. In *Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 559–578, 2022.

[93] Xuanhe Zhou, Zhaoyan Sun, and Guoliang Li. Db-gpt: Large language model meets database. *Data Science and Engineering*, 9(1):102–111, Mar 2024.

A Implementation Details

HSPMD is designed to accommodate diverse forms of heterogeneity, which requires additional computation graph representations and dynamic adjustments. However, extracting computation graph information in PyTorch [56] is non-trivial. To address this, we have developed a prototype framework comprising 87.9K lines of C++ code and 16.9K lines of CUDA code, supporting graph-based distributed DL training. The framework implements all major parallelism paradigms, including data parallelism (DP), tensor parallelism (TP), pipeline parallelism (PP), and context parallelism (CP), as well as complementary techniques such as ZeRO-1, offloading, activation recomputation, and mixed-precision training.

Within the total 87.9K lines of C++, 16.9K lines implement the core logic of HSPMD, covering sharding annotation, communication resolution, graph specialization, and graph switching. An additional 14.7K lines of C++, largely consisting of pybind [66] bindings, serve as glue code that exposes HSPMD APIs to Python. Together with 12.5K lines of Python, these APIs allow users to define tensors, operators, and training workflows, enabling seamless deployment and reconfiguration of heterogeneous strategies.

Our prototype system is optimized for large language models (LLMs) training, with collective communication primitives implemented via NCCL [54] and computation kernels accelerated using libraries such as FlashAttention [11, 12], cuBLAS [52], and cutlass [53]. When evaluated with homogeneous parallel strategies, HSPMD achieves performance comparable to Megatron [37, 51, 60] (as evaluated in §9(a)). While our current implementation and experiments focus on LLMs—due to their extreme model sizes and demand for massive GPU clusters—the proposed designs are broadly applicable to other deep learning workloads.

Below, we provide additional details on the implementation that are not covered in the main text.

A.1 Scenario-specific Planning

In this section, we present our approach for automatically generating and selecting *annotation plans* (§3) for the three representative scenarios in §9. The primary focus of this paper is on the underlying system design rather than on optimizing the planning process. To that end, the planner is implemented as a modular component that users can readily replace or extend—for example, by incorporating algorithms from prior scenario-specific work. The implementation described here represents just one of many possible planning algorithms, provided to demonstrate HSPMD 's flexibility in supporting diverse application settings.

Our approach starts by profiling execution time and memory usage for homogeneous parallelism configurations on a single model unit (e.g., an `Attention` operator), given the target model and workload specifications (i.e., context length and batch size). Based on these measurements, we develop a regression-based *cost model* to estimate per-device execution time and memory usage for any heterogeneous parallel strategy expressible in HSPMD. This is achieved by analyzing the dataflow and aggregating results from different *sharding subgroups*, each of which employs homogeneous parallelism and can be estimated individually. Below, we detail the algorithms used in each scenario.

(a) Heterogeneous devices. Using our *cost model*, we formulate a two-level optimization problem. The objective is to minimize the maximum estimated execution time across all devices while satisfying memory constraints:

- **Pipeline configuration** (first-level): We determine the number of pipelines (§6.3) along with their parallel methods, stage counts, and device-to-stage mappings. This is formulated as a Mixed-Integer Nonlinear Programming (MINLP) problem.
- **Layer & micro-batch assignment** (second-level): We assign model layers to different pipeline stages and calculate optimal micro-batch configurations (i.e., micro-batch size and count) for each pipeline, solved as multiple Integer Linear Programming (ILP) problems.

The combined solution yields a unique *annotation plan* for deployment.

While the ILPs can be solved efficiently, the MINLP remains computationally expensive. Instead of solving the original MINLP directly, we adopt a heuristic-guided greedy search to prune suboptimal solutions upfront, thereby shrinking the strategy space and reducing the solving overhead while preserving solution quality.

(b) Unstable devices. Our elastic training approach begins by generating an optimal initial *annotation plan* using the same planner as in (a). When reconfiguration is required, profiling is unnecessary since the model and workload remain unchanged. Moreover, we intelligently prune the MINLP search space using intermediate results from the initial planning phase. This optimization enables the first-level problem to be solved significantly faster during reconfigurations, while the second-level ILPs (remaining identical to the initial solution) continue to be resolved with minimal overhead.

(c) Mixed-length data. For datasets with variable sequence lengths, our methodology begins by analyzing the dataset distribution to obtain sequence length statistics. Based on these statistics, we employ a dynamic programming algorithm that jointly determines both the optimal sequence length intervals for partitioning variable-length sequences and the corresponding parallel method for each interval. Different pipelines, each tailored to a specific interval and parallel method, are then combined into a single heterogeneous parallel strategy. Recognizing that per-batch sequence length distributions may deviate from global dataset statistics, we prepare multiple parallel strategies optimized for different maximum sequence

Table 5: Workflow and time breakdown of different scenarios. Step 1 (planning) is automated by our planner module (users may optionally provide the *annotation plan* manually), while Step 2 (graph specialization) and Step 3 (graph switching) are handled by HSPMD. The planner generates the *annotation plan* (offline/online) and may select the optimal plan if multiple candidates exist (online). *Executable graphs* are specialized upon receiving the *annotation plan* (offline/online), and different *executable graphs* can be switched on-the-fly (online). Note that all online overheads are included in §9's end-to-end results.

Scenario	**Step 1: Planning**	**Step 2: Graph Specialization (§6)**	**Step 3: Graph Switching (§7)**
Heterogeneous Devices	**Offline:** Single *annotation plan* generated before training (6.4s-18.2s).	**Offline:** Executed once before training (6.8s-8.3s).	—
Unstable Devices	**Offline:** Initial *annotation plan* generated before training (9.1s-18.2s). **Online:** New *annotation plan* generated whenever device availability changes (0.05s-1.2s).	**Online:** Executed each time a new *annotation plan* is generated (6.2s-7.9s).	**Online:** Triggered after new *executable graphs* are specialized (2.2s-5.3s, switch both FP32 parameters & optimizer states).
Mixed-length Data	**Offline:** Multiple *annotation plans* generated before training (24.1s-26.8s). **Online:** *Annotation plan* selection for next training steps (1.5s-2.1s, overlapped with the current step's training).	**Offline:** Executed for each *annotation plan* before training (14.8s-16.2s).	**Online:** Triggered between training steps if different *annotation plans* are favored (0.8s-1.3s, switch only BF16 parameters).

lengths. During execution, for each incoming batch, our system leverages the *cost model* again to distribute sequences among pipelines and dynamically select the strategy with the lowest estimated execution time. This data assignment and strategy selection process runs concurrently on CPU resources, carefully orchestrated to overlap with GPU training. By parallelizing the planning of upcoming steps (e.g., preparing the next 10 steps during the current step's execution) across available CPU cores, our implementation effectively amortizes and hides the planner's runtime overhead.

Across all scenarios, we leverage PuLP [65] for solving ILPs and Pyomo [6] for solving MINLPs. As detailed in Table 5, we provide overhead measurements including planning, graph specialization (§6), and graph switching (§7) for each scenario, accompanied by thorough workflow descriptions. The results indicate that our planners either introduce minimal overhead or run offline, ensuring negligible impact on online training.

A.2 Execution

In this section, we provide the details not covered in §6.3 on how to recognize heterogeneous pipelines and coordinate their execution, as well as how to support dynamic input shapes during execution without relying on the sharding annotation system.

Pipeline construction and scheduling. HSPMD supports various scheduling schemes, including GPipe [27] and 1F1B [50], while enabling independent pipelines to process micro-batches with different sizes and quantities. This functionality is achieved through two key steps: **(i)** determining the pipeline structure based on current sharing annotations (i.e., identifying device-to-stage and device-to-pipeline mappings); and **(ii)** coordinating device execution according to the selected pipeline scheduler (e.g., 1F1B).

As illustrated in Figure 21 (left), after *executable graphs* are specialized (§6), we further partition each device's *executable graph* into four types of *executable subgraphs*:

- **Pre-processing** (*Pre*): Includes operators such as precision conversion (e.g., FP32 to BF16) for mixed-precision training, and parameter gathering (e.g., `AG`, `SplitAG`) for ZeRO [58] sharding.
- **Post-processing** (*Post*): Includes operators for gradient accumulation (e.g., `Sum`), gradient synchronization (e.g., `RS`, `SplitRS`), and optimizer updates (e.g., Adam [36]).
- **Forward** (*F*): Includes operators for the DL forward pass

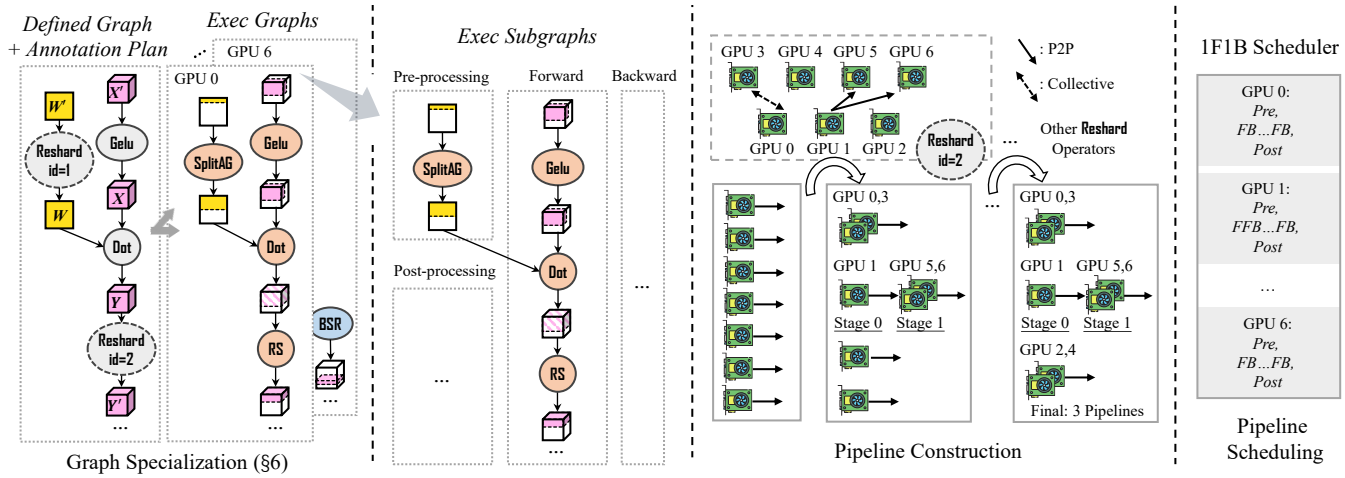


Figure 21: Pipeline construction and scheduling. The *executable graph* associated with each device is divided into multiple *executable subgraphs*. Once the pipelines are constructed, each device is assigned a schedule that dictates the execution order of its *executable subgraphs*. This figure aligns with Figure 12.

(e.g., `Dot`, `AttentionFwd`).

- **Backward** (*B*): Includes operators for the DL backward pass (e.g., `Dot`, `AttentionBwd`).

These *executable subgraphs* serve as the scheduling units for subsequent execution given the pipeline scheduler.

Below, we describe the process of constructing pipelines, whereby each device identifies its stage number and pipeline number, ensuring that all devices are aware of their roles during pipeline scheduling. We design an algorithm for constructing pipelines in a step-by-step manner. As illustrated in Figure 21 (middle), we initially construct a separate pipeline for each device. Subsequently, we sequentially analyze all `Reshard` operators involved in the forward and backward *executable subgraphs*. In Figure 21, we identify that `Reshard` (id=1) is executed only once at the beginning, which belongs to the pre-processing *executable subgraph* and does not participate in forward or backward passes, so the first communication operator actually analyzed is `Reshard` (id=2). For this communication operator, we decompose its internal communication into two categories: collective communication and peer-to-peer (P2P) communication. All devices involved in collective communication are merged into the same pipeline, while devices involved in P2P communication are concatenated into subsequent stages of the pipeline. For example, in Figure 21 (middle), after processing `Reshard` (id=2), the pipelines containing GPU 0 and GPU 3 are merged into one, while GPU 5 and GPU 6 are appended to the pipeline containing GPU 1 as the next stage. Through iterative application of this process, all pipelines are eventually fully constructed.

After pipeline construction, the system orchestrates execution according to the pipeline structure during runtime. Each device is assigned a schedule that defines the execution order of its *executable subgraphs*. For instance, in Figure 21 (right), when using the 1F1B scheduler, GPU 0 (located at the final stage) follows the schedule $Pre, FB \ldots FB, Post$, whereas GPU 1 (on the first stage) executes according to $Pre, FFB \ldots FB, Post$. Here, *Pre* and *Post* denote the pre- and post-processing *executable subgraphs*, respectively, and *F* and *B* represent forward and backward *executable subgraphs*, respectively.

Symbolic shape extension. During execution, HSPMD dynamically adapts to variations in input tensor shapes across different pipelines (e.g., to facilitate load balancing, data packing, and other runtime optimizations). While static sharding annotations define the high-level sharding pattern, the concrete shapes of individual shards are resolved at runtime.

To support this dynamic behavior, HSPMD extends tensor metadata with symbolic variables, which represent unknown dimensions of the shape (e.g., **S** for sequence length). These variables are propagated through tensor operations on each device while maintaining dimensional constraints. For instance, splitting a tensor along its sequence dimension would generate the constraint $\mathbf{S}' = \mathbf{S}/2$, preserving algebraic relationships between the symbols of the two tensors. The system resolves these symbols to concrete arithmetic values only when actual input tensors are provided at runtime to each device. This design allows different ranks to process dynamically shaped data. Additionally, HSPMD enforces both compile-time and runtime checks to detect invalid symbol usage. For example, defining $\mathbf{S}' = \mathbf{S}/2$ without satisfying the constraint $\mathbf{S} \equiv 0 \pmod 2$ is illegal and may lead to shape mismatches during cross-rank communication.

B Training Convergence

In this section, we present a comparative analysis of training loss under two configurations (C1 and C2, totally aligned with

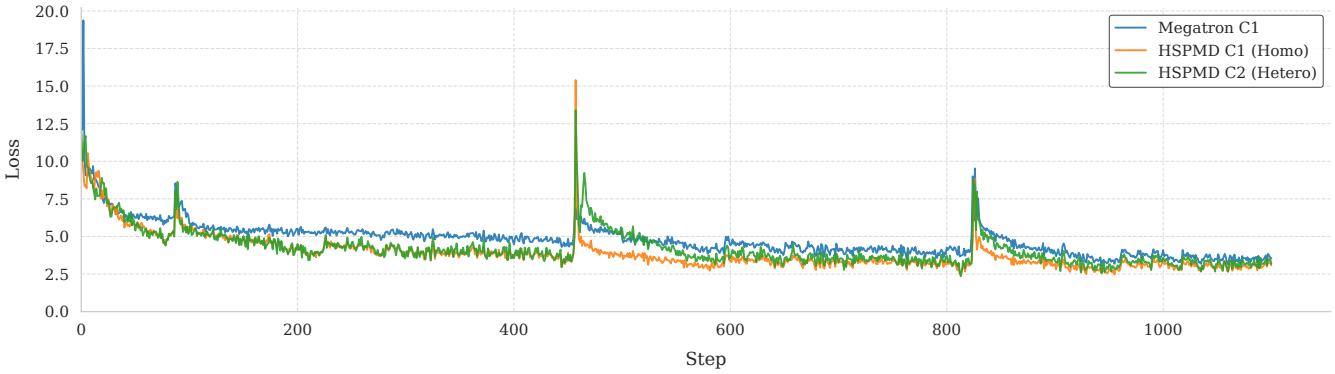


Figure 22: Training loss comparison.

§10). In both configurations, a 32B Llama model is trained using HSPMD, yet they differ in GPU resources and parallel strategies:

- C1: The model is trained on 32 H20 GPUs under a homogeneous parallel strategy.
- C2: The model is trained on 31 H20 GPUs under a heterogeneous parallel strategy.

Comprehensive descriptions of these parallel strategies are documented in Appendix C.3, while the full training setup is summarized in Table 6.

Table 6: Training setup.

Category	Parameter	Value
Model Configuration	**Number of Layers**	60
	Hidden Size	6,656
	Attention Heads	64
	FFN Hidden Size	17,920
	Vocabulary Size	50,304
	Query Groups	64
	Activation Function	SwiGLU
	Normalization	RMSNorm
	Position Encoding	RoPE
	Dropout	None
Hyperparameters	**Precision**	BF16
	Optimizer	Adam
	Learning Rate	1×10^{-4}
	Warm-up Steps	0
	Scheduler	Constant
	Weight Decay	0
	Adam β_1	0.9
	Adam β_2	0.999
	Adam ε	1×10^{-8}
	Gradient Clipping	None
	Dataset	Github
	Epoch	1
	Context Length	4,096
	Batch Size	64

We also train a model using Megatron, employing the same homogeneous parallel strategy and training setup as in C1. As shown in Figure 22, the results show that HSPMD achieves good convergence under both homogeneous and heterogeneous settings. In the homogeneous setting (C1), the convergence behavior closely matches that of the Megatron baseline, with minor differences in absolute loss values attributable to framework-level implementation variations. Besides, the heterogeneous setting (C2) exhibits convergence nearly identical to C1, with only marginal fluctuations observed during training irregularities. These findings demonstrate that the introduction of asymmetric sharding and diverse asymmetric communication patterns does not compromise training convergence.

C More Experimental Details

This section provides a comprehensive analysis of HSPMD's heterogeneous parallel strategies across diverse scenarios and compares them against baseline methods. While these strategies (formulated as *annotation plans*) are automatically generated by scenario-specific planners (Appendix A.1), users also have the flexibility to define customized strategies by manually supplying their own *annotation plan*.

C.1 Cluster Setup

Table 7 presents the configuration of our computing cluster. The H800 GPU offers higher computational power but lower intra-node (NVLink) communication bandwidth, whereas the H20 GPU provides lower computational performance but higher communication bandwidth. Additionally, the two GPUs differ slightly in memory capacity, posing challenges for heterogeneous parallel computing on this setup.

C.2 Scenario (a): Heterogeneous Devices

Table 8 summarizes the optimal parallel strategies for DeepSpeed and Megatron, the two baselines, across different model

Table 7: Heterogeneous GPU cluster.

GPUs	Memory	BF16 Tensor Core	NVLink
16 H800	81559 MB	990 TFLOPS	400 GB/s
32 H20	97871 MB	148 TFLOPS	900 GB/s

sizes and heterogeneous device setups. These strategies were derived through an exhaustive search process, systematically evaluating various combinations of parallelisms and training optimizations to determine the most efficient ones.

Table 8: Optimal parallel strategies for DeepSpeed and Megatron. "DP" refers to the data parallel degree, "TP" to the tensor parallel degree, "PP" to the pipeline parallel degree, and "SP" to the sequence parallel (Ulysses-SP) degree. "AC" signifies the use of activation checkpointing, while "bs" denotes micro-batch size. All DeepSpeed configurations employ ZeRO-3 optimization, whereas Megatron implementations utilize ZeRO-1.

Model Size & GPUs	DeepSpeed	Megatron
32B, 16 H800	DP8SP2AC, bs2	TP4PP4, bs1
32B, 16 H20	DP8SP2AC, bs2	TP4PP4, bs1
32B, 16 H800, 16 H20	DP16SP2AC, bs2	DP2TP4PP4, bs2
32B, 16 H800, 24 H20	DP20SP2AC, bs4	DP2TP4PP5, bs2
32B, 16 H800, 32 H20	DP24SP2AC, bs1	DP4TP4PP3, bs2
70B, 16 H800, 16 H20	DP16SP2AC, bs1	TP8PP4, bs1
70B, 16 H800, 24 H20	DP20SP2AC, bs2	TP8PP5, bs1
70B, 16 H800, 32 H20	DP24SP2AC, bs1	TP8PP6, bs1

Next, as shown in Table 9, we sequentially present the optimal strategies for HSPMD across different heterogeneous device setups. These strategies are given by our static planner (as introduced in Appendix A.1). Each cell represents a tensor parallel group, detailing the assigned rank IDs (denoted as "R") and the corresponding layer IDs (denoted as "L"). Yellow cells correspond to H800 GPUs, while white ones represent H20 GPUs. The leftmost column indicates the micro-batch size (denoted as "bs") and the number of micro-batches (represented by the preceding coefficient) for each pipeline. Between pipelines, data parallelism is utilized to synchronize gradients.

Notably, across all strategies, we utilize ZeRO-1 to partition FP32 parameters and optimizer states across data parallel replicas. This memory optimization enables the adoption of more memory-intensive parallel strategies (i.e., those with higher data parallel degrees). While homogeneous parallel strategies use standard `AG` (all-gather) for parameter collection and `RS` (reduce-scatter) for gradient synchronization, our heterogeneous approach requires specialized operators. Specifically, when tensor parallel degrees differ across data parallel groups, the inserted `Reshard` operators (responsible for parameter collection and gradient synchronization) are instantiated with `SplitAG` and `SplitRS` operators (detailed in §5.2). This adaptation facilitates the asymmetric communication required when integrating ZeRO-1 with heterogeneous pipelines.

Table 9: Optimal parallel strategies for HSPMD on different heterogeneous clusters. R0-15 are H800 GPUs and R16-47 are H20 GPUs.

32B, 16 H800, 16 H20				
32 × bs1 (4 Stages)	R16-19 L0-6	R20-23 L7-13	R0-3 L14-36	R4-7 L37-59
32 × bs1 (4 Stages)	R24-27 L0-6	R28-31 L7-13	R8-11 L14-36	R12-15 L37-59

32B, 16 H800, 24 H20				
32 × bs1 (5 Stages)	R16-19 L0-5	R20-23 L6-11	R24-27 L12-17	R0-3 L18-38
	R4-7 L39-59			
32 × bs1 (5 Stages)	R28-31 L0-5	R32-35 L6-11	R36-39 L12-17	R8-11 L18-38
	R12-15 L39-59			

32B, 16 H800, 32 H20			
16 × bs1 (3 Stages)	R16-19 L0-10	R20-23 L11-21	R0-3 L22-59
16 × bs1 (3 Stages)	R24-27 L0-10	R28-31 L11-21	R4-7 L22-59
16 × bs1 (3 Stages)	R32-35 L0-10	R36-39 L11-21	R8-11 L22-59
16 × bs1 (3 Stages)	R40-43 L0-10	R44-47 L11-21	R12-15 L22-59

70B, 16 H800, 16 H20				
64 × bs1 (4 Stages)	R16-23 L0-10	R24-31 L11-21	R0-7 L22-50	R8-15 L51-79

70B, 16 H800, 24 H20				
64 × bs1 (5 Stages)	R16-23 L0-9	R24-31 L10-19	R32-39 L20-29	R0-7 L30-54
	R8-15 L55-79			

70B, 16 H800, 32 H20			
32 × bs1 (3 Stages)	R16-23 L0-16	R24-31 L17-33	R0-7 L34-79
32 × bs1 (3 Stages)	R32-39 L0-16	R40-47 L17-33	R8-15 L34-79

C.3 Scenario (b): Unstable Devices

Table 10 presents the optimal strategies of DeepSpeed and Megatron after each reconfiguration in the unstable devices scenario. Notably, GPU failures occurred during the reconfigurations from C1 to C2 and from C5 to C6. However, due to the system's inability to support non-uniform sharding, they failed to utilize the remaining seven functional GPUs in the machine. Consequently, the entire machine was treated as

failed, which is why the strategy for C2 matches C3, and C6's strategy aligns with C7's.

Table 10: Optimal parallel strategies for DeepSpeed and Megatron. "DP" refers to the data parallel degree, "TP" to the tensor parallel degree, "PP" to the pipeline parallel degree, and "SP" to the sequence parallel (Ulysses-SP) degree. "AC" signifies the use of activation checkpointing, while "bs" denotes micro-batch size. All DeepSpeed configurations employ ZeRO-3 optimization, whereas Megatron implementations utilize ZeRO-1.

Configuration	DeepSpeed	Megatron
C1: 32 H20	DP16SP2AC, bs2	DP2TP4PP4, bs2
C2: 31 H20	DP12SP2AC, bs2	TP4PP6, bs1
C3: 24 H20	DP12SP2AC, bs2	TP4PP6, bs1
C4: 16 H800, 32 H20	DP24SP2AC, bs1	DP4TP4PP3, bs2
C5: 16 H800, 24 H20	DP20SP2AC, bs2	TP8PP5, bs1
C6: 15 H800, 24 H20	DP16SP2AC, bs2	DP2TP4PP4, bs2
C7: 8 H800, 24 H20	DP16SP2AC, bs2	DP2TP4PP4, bs2

Upon changes in GPU availability, we invoke our online planner (Appendix A.1) to determine the optimal parallel strategy. Table 11 reports the strategies adopted by HSPMD on homogeneous clusters (C1–C3). In our setup, we deploy two pipelines and disable ZeRO-1 to ensure fault isolation. Failures in one pipeline do not cause permanent loss of model weights, eliminating the need for restarting. This setting is consistent with Oobleck, which also disables ZeRO-1 and achieves fault tolerance through data-parallel replication, making our comparison fair. Notably, the same design is followed by many elastic training systems (e.g., Recycle [21]).

Table 11: Parallel strategies for HSPMD during elastic training on homogeneous clusters.

C1: 32 H20				
16 × bs2 (4 Stages)	R0-3 L0-14	R4-7 L15-29	R8-11 L30-44	R12-15 L45-59
16 × bs2 (4 Stages)	R16-19 L0-14	R20-23 L15-29	R24-27 L30-44	R28-31 L45-59

C2: 31 H20				
33 × bs1 (4 Stages)	R0-3 L0-14	R4-7 L15-29	R8-11 L30-44	R12-15 L45-59
31 × bs1 (5 Stages)	R16-19 L0-15	R20-23 L16-31	R24-27 L32-47	R28-29 L48-55
	R30 L56-59			

C3: 24 H20			
32 × bs1 (3 Stages)	R0-3 L0-19	R4-7 L20-39	R8-11 L40-59
32 × bs1 (3 Stages)	R12-15 L0-19	R16-19 L20-39	R20-23 L40-59

In Table 12, we detail the strategies (C4–C7) adopted by HSPMD on heterogeneous clusters, with yellow and white cells representing H800 and H20 GPUs, respectively. Following the homogeneous cluster approach, we disable ZeRO-1 and configure two pipelines to enable fault tolerance.

However, disabling ZeRO-1 increases memory overhead, precluding the use of the optimal parallel strategy. For instance, the strategy in C4: 16 H800, 32 H20 (with a data parallel degree of 2) differs from that in Table 9 for standard heterogeneous clusters (32B, 16 H800, 32 H20, with a data parallel degree of 4). Consequently, training performance degrades by approximately 15%, increasing the per-step training time from 6.05s to 6.91s (shown in Figure 15 and Figure 16).

Despite this performance degradation, HSPMD's heterogeneous sharding effectively balances workloads, still maintaining superior performance compared to the homogeneous parallel strategies used in DeepSpeed and Megatron (shown in Figure 16).

Table 12: Parallel strategies for HSPMD during elastic training on heterogeneous clusters. R0-15 are H800 GPUs and R16-47 are H20 GPUs.

C4: 16 H800, 32 H20				
32 × bs1 (6 Stages)	R16-19 L0-4	R20-23 L5-10	R24-27 L11-16	R28-31 L17-22
	R0-3 L23-40	R4-7 L41-59		
32 × bs1 (6 Stages)	R32-35 L0-4	R36-39 L5-10	R40-43 L11-16	R44-47 L17-22
	R8-11 L23-40	R12-15 L41-59		

C5: 16 H800, 24 H20				
32 × bs1 (5 Stages)	R16-19 L0-5	R20-23 L6-11	R24-27 L12-17	R0-3 L18-38
	R4-7 L39-59			
32 × bs1 (5 Stages)	R28-31 L0-5	R32-35 L6-11	R36-39 L12-17	R8-11 L18-38
	R12-15 L39-59			

C6: 15 H800, 24 H20				
33 × bs1 (5 Stages)	R16-19 L0-5	R20-23 L6-11	R24-27 L12-17	R0-3 L18-38
	R4-7 L39-59			
31 × bs1 (6 Stages)	R28-31 L0-5	R32-35 L6-11	R36-39 L12-17	R8-11 L18-39
	R12-13 L40-52	R14 L53-59		

C7: 8 H800, 24 H20				
32 × bs1 (4 Stages)	R16-19 L0-8	R20-23 L9-18	R24-27 L19-28	R0-3 L29-59
32 × bs1 (4 Stages)	R28-31 L0-8	R32-35 L9-18	R36-39 L19-28	R4-7 L29-59

C.4 Scenario (c): Mixed-length Data

Table 13 presents the optimal parallel training strategies for the mixed-length data scenario using DeepSpeed and Megatron, with context lengths of 32K and 16K (using the 32B model on 32 H20 GPUs). Due to the substantially longer context lengths, these frameworks must adopt larger sequence/-context or tensor parallelism. However, in practice, only a small fraction of sequences per training step actually reach the maximum context length. As a result, most shorter sequences are processed inefficiently, missing opportunities for more optimized training strategies.

Table 13: Optimal parallel strategies for DeepSpeed and Megatron. "DP" refers to the data parallel degree, "TP" to the tensor parallel degree, "PP" to the pipeline parallel degree, "CP" to the context parallel degree, and "SP" to the sequence parallel (Ulysses-SP) degree. "AC" signifies the use of activation checkpointing, while "bs" denotes micro-batch size. All DeepSpeed configurations employ ZeRO-3 optimization, whereas Megatron implementations utilize ZeRO-1.

Context Length	DeepSpeed	Megatron
32K	DP4SP8AC, bs1	DP2TP8CP2, bs1
16K	DP8SP4AC, bs1	TP8PP4, bs1

Table 14 presents the parallel strategies adopted by HotSPa across different sequence length intervals. By partitioning sequences into buckets based on length and processing them sequentially within a training step (with gradient accumulation before weight updates), this approach maintains mathematical equivalence to standard training while enabling length-adaptive parallel strategies.

Although this method incurs frequent switching overhead within the step, such costs are substantially outweighed by the benefits of parallel strategies optimization, ultimately resulting in performance improvements.

Table 14: Optimal parallel strategies for HotSPa. "DP" refers to the data parallel degree, "TP" to the tensor parallel degree, and "PP" to the pipeline parallel degree. "bs" denotes micro-batch size. All configurations employ ZeRO-1 optimization.

Context Length	Interval-based Parallel Strategies
32K	16K-32K: DP2TP16, bs1 4K-16K: DP2TP8PP2, bs1 0-4K: DP4TP4PP2, bs1
16K	4K-16K: DP2TP8PP2, bs1 0-4K: DP4TP4PP2, bs1

Tables 15 and 16 further present the heterogeneous strategies adopted by HSPMD for context lengths of 32K and 16K, respectively. In Table 15, Strategy A and Strategy B correspond to those illustrated in Figure 18.

For HSPMD, steps with varying maximum sequence lengths are processed using either Strategy A or B, while sequences of different lengths within a step are distributed across multiple pipelines to balance the load based on the *cost model* we designed. Once all data within a step has been processed across the pipelines, gradients are synchronized to perform the step update. This approach further improves performance compared to HotSPa by eliminating frequent strategy switching within a step and more effectively addressing workload imbalance through the use of heterogeneous strategies, rather than homogeneous ones.

Table 15: Heterogeneous parallel strategies for HSPMD with 32K context length, where the strategy is dynamically selected based on the maximum sequence length (MaxSeqLen) at each processing step.

Strategy A: MaxSeqLen ∈ (16K, 32K]	
Long Sequence Pipeline-1 (1 Stages)	R0-15 L0-59
Short Sequence Pipeline-1 (1 Stages)	R16-19 L0-59
... (1 Stages)	... L0-59
Short Sequence Pipeline-4 (1 Stages)	R28-31 L0-59

Strategy B: MaxSeqLen ∈ (0K, 16K]		
Long Sequence Pipeline-1 (1 Stages)	R0-7 L0-59	
Short Sequence Pipeline-1 (2 Stages)	R8-11 L0-29	R12-15 L30-59
... (2 Stages)	... L0-29	... L30-59
Short Sequence Pipeline-3 (2 Stages)	R24-27 L0-29	R28-31 L30-59

Table 16: Heterogeneous parallel strategies for HSPMD with 16K context length, where the strategy is dynamically selected based on the maximum sequence length (MaxSeqLen) at each processing step.

Strategy A: MaxSeqLen ∈ (4K, 16K]		
Long Sequence Pipeline-1 (1 Stages)	R0-7 L0-59	
Short Sequence Pipeline-1 (2 Stages)	R8-11 L0-29	R12-15 L30-59
... (2 Stages)	... L0-29	... L30-59
Short Sequence Pipeline-3 (2 Stages)	R24-27 L0-29	R28-31 L30-59

Strategy B: MaxSeqLen ∈ (0K, 4K]
DP4TP4PP2

ARTIFACT EVALUATED
USENIX
AVAILABLE

Syncopate: Efficient Multi-GPU AI Kernels via Automatic Chunk-Centric Compute-Communication Overlap

Xinwei Qiang[1,*], Yue Guan[1,*], Zhengding Hu[1], Keren Zhou[3,4], Yufei Ding[1,2], Adnan Aziz[2]
[1]*University of California, San Diego,* [2]*Meta,* [3]*George Mason University,* [4]*OpenAI*
[1]*{x1qiang, yueguan, zhh068, yufeiding}@ucsd.edu*
[2]*{adnanaziz}@meta.com*
[3]*{kzhou6}@gmu.edu*

Abstract

Communication has become a first-order bottleneck in large-scale GPU workloads, and existing distributed compilers address it mainly by overlapping whole compute and communication kernels at the stream level. This coarse granularity incurs extra kernel launches, forces device-wide synchronizations at kernel boundaries, and leaves substantial slack when the slowest tile or kernel stretches the communication tail. We present Syncopate, a compiler and runtime that enable automatic fine-grained overlap around a single fused compute kernel. Syncopate introduces a communication chunk abstraction that decouples communication granularity from kernel structure and backend mechanisms, allowing chunk-level plans to be ported from existing distributed compilers, written directly by users, or instantiated from reusable templates. Given a local Triton kernel and a chunk schedule, Syncopate performs transformations to align computation with chunk availability. Implemented as a source-to-source compiler on Triton, Syncopate delivers an average end-to-end speedup of 1.3× and up to 4.7× on multi-GPU workloads. Our code is open-sourced at https://github.com/tie-pilot-qxw/syncopate.

1 Introduction

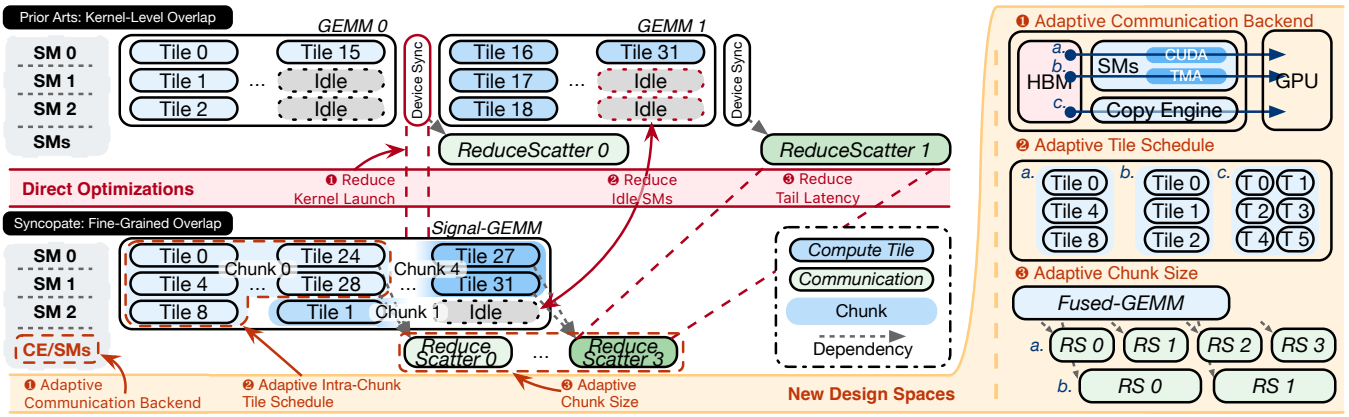
Communication has become a first-order bottleneck for training and serving large neural networks on multi-GPU systems. Even with high-bandwidth interconnects such as NVLink [23] and NVSwitch, collective operations like AllGather, ReduceScatter, and All-to-All frequently dominate end-to-end latency for tensor-parallel feed-forward layers and attention layers. To hide this cost, recent systems and distributed compilers aggressively search for schedules that overlap computation and communication at the kernel level. Given a computation graph and a device mesh, these compilers select parallelization strategies, insert communication kernels, and assign compute and communication kernels to streams so that multiple kernels can run concurrently. This kernel-level overlap [4, 10, 31, 36, 38, 47] has become the default mechanism for improving utilization in distributed settings.

However, kernel-level overlap is fundamentally insufficient for fully hiding communication latency. As illustrated in Fig. 1, this kernel-level scheduling forces a device-wide synchronization at every kernel boundary and incurs extra launch and sync overhead for each communication phase (❶). Moreover, by splitting computation into multiple shorter kernels, the work within each launch is further partitioned into waves of compute tiles on each SM, increasing the fraction of time SMs sit idle; even when part of the data needed by later kernels is already available, tiles in the current wave must wait for the slowest tile to finish (❷). Finally, the coarse granularity of kernel-level overlap leaves a long segment of communication at the end of the timeline that receives little or no overlap (❸).

This motivates us to overlap computation and communication at a finer intra-kernel granularity. Such fine-grained overlap opens up new design space for improving end-to-end efficiency. As shown by the yellow region in Fig. 1, Syncopate launches communication directly from within the fused kernel, rather than delegating to external communication libraries, giving the compiler explicit control over which hardware backend to use for each transfer (copy engine, tensor memory accelerator, or load/store on CUDA cores) as shown with ❶. This also allows us to explore much smaller communication granularities without incurring additional kernel-launch overhead, and to tune the chunk size that best balances link throughput against synchronization cost (❷). Finally, because tiles and communication are orchestrated inside a single kernel, we can reshape the intra-kernel tile schedule to track communication progress while still preserving locality in the register, shared-memory, and cache hierarchy (❸).

We present Syncopate, a compiler that turns the vision of fine-grained overlap for distributed kernel generation into a practical and general system. At the heart of Syncopate is the notion of a communication chunk, which represents a logical block of data associated with a particular communication operation and the tiles that produce or consume it. This abstrac-

*Xinwei and Yue contributed equally.



(a) Comparison of compute–communication overlap at different levels

(b) New design space for fine–grained overlap

Figure 1: Motivating example of Syncopate. Red numbers show the direct improvements gained by fine-grained overlap over kernel-level overlap, while orange numbers show the additional improvements from the new design space enabled by Syncopate.

tion is motivated by a key observation: fine-grained overlap requires a communication granularity that flexibly matches how tiles generate data and how communication backends consume it, rather than assuming that tile-level or full-kernel granularity is always appropriate. By making chunks explicit, Syncopate can represent a wide range of overlap patterns and enable the compiler to reason about when each chunk becomes available, which tiles produce or use it, and how these dependencies interact with fused multi-stage tensor programs and potentially irregular collectives. This abstraction exposes a small set of principled knobs, including chunk size, backend choice, and tile order, that define the design space later explored by our autotuner.

Building on this abstraction, Syncopate implements a source-to-source compiler and runtime that transform standard Triton [34] kernels and a high-level chunk-based communication plan into fused distributed kernels capable of fine-grained compute–communication overlap. The compiler restructures the kernel's tile execution to follow communication progress and selects appropriate backends to realize each chunk transfer, while a lightweight runtime executes these transfers and integrates seamlessly with PyTorch distributed [2] so that Syncopate kernels can replace standard operators with only minimal changes to user code. Syncopate further performs inter- and intra-chunk autotuning, adjusting chunk sizes, backend choices, SM allocations, and tile schedules, to consistently achieve high performance across diverse operators. This implementation strategy makes the abstract chunk-based model concrete, enabling rapid prototyping of new overlap policies while remaining compatible with real distributed workloads and production communication stacks.

In summary, this work makes the following contributions:

- We introduce a chunk-based abstraction that decouples high-level overlap intent from low-level implementation, enabling fine-grained compute-communication overlap inside distributed kernels.
- We design and implement a compiler pipeline and runtime that takes annotated local kernels and chunk-level communication plans, then generates efficient fused distributed kernels with inter- and intra-chunk autotuning.
- We evaluate Syncopate on a diverse set of distributed workloads and observe an average speedup of 1.3× on common operators, with improvements reaching up to 4.7× in the best cases.

2 Background and Related Works

2.1 Distributed Compilers

As model sizes grew, tensor compilers [5, 6, 9, 29, 40, 43] incorporated basic multi-GPU support [1, 16, 27, 30, 37, 41], typically by composing single-GPU kernels with predefined collective primitives. Systems like Alpa [44], Mercury [13], and other recent distributed compilers extend this idea by searching over communication patterns and schedules at the level of whole kernels: given a parallelization strategy, they first construct a communication plan containing AllGather, ReduceScatter, or All-to-All operators, and then explore different mappings of compute and communication kernels to streams in order to maximize kernel-level overlap (Table 1). This design has made distributed training far more accessible, but it also bakes in a rigid abstraction boundary: communication is planned as a sequence of full-kernel collectives whose launch and completion times are the basic units of scheduling. As a result, all these distributed compilers focus their search on the communication plan at the kernel level and are fundamentally blind to finer-grained opportunities inside kernels, such as overlapping per-shard or per-tile communication with computation, reusing remote data across tiles, or exploiting topology-aware pipelining within a fused kernel. Once a kernel is chosen and its associated collective is placed, the compiler can only treat it as an atomic black box, leaving

Table 1: Comparison of projects on distributed operations.

Projects	Granularity	Compute	Communication	Schedule	Performance
Automatic Approaches					
Alpa [44]	Kernel	Auto	Auto	Template	✔
Mercury [13]	Kernel	Auto	Auto	Auto	✔✔
Manual Implementations					
Flux [3]	Tile	Manual	Manual	Manual	✔✔
AsyncTP [38]	Tile	Manual	Manual	Manual	✔✔
FlashOverlap [14]	Chunk	Manual	Manual	Manual	✔✔✔
Domain Specific Languages					
ThunderKittens [32,33]	Tile	Manual	Manual	Manual	✔✔✔
TritonDistributed [45,46]	Chunk	Manual	Manual	Manual	✔✔✔
Syncopate	Chunk	Auto	Auto	Template	✔✔✔

significant intra-kernel overlap potential untapped even under an "optimal" kernel-level schedule.

2.2 Manual Kernel Designs

A complementary line of work pushes beyond kernel-level overlap and instead hand-crafts fine-grained pipelines that interleave communication and computation at the level of tiles, tokens, or shards. Flux [3] fuses GEMM with collectives at tile granularity and over-decomposes work to maximize overlap for both training and inference, while Comet [42] targets MoE with a shared-tensor abstraction and NVSHMEM-backed buffers to overlap token-wise communication with tile-wise compute at production scale. FlashOverlap [14] uses lightweight readiness signaling and layout reordering to trigger overlap with standard NCCL collectives without modifying existing compute kernels, and systems like TritonDistributed [45,46] introduce tile-centric or OpenSHMEM-style primitives to Triton [34] that make it easier to author overlapped kernels (e.g., fused AllGather/GEMM or GEMM/ReduceScatter) in a domain-specific language.

Despite these advances, all of these systems fundamentally rely on manual, operator-specific engineering: experts must design fused kernels, choose tiling and buffering schemes, and reason about synchronization and communication protocols for each new model architecture or hardware platform. The resulting implementations are highly optimized but difficult to generalize or retarget, and they do not provide a general compiler abstraction for expressing and reusing fine-grained overlap patterns across operators. Emerging fine-grained DSLs and primitives greatly lower the barrier to writing overlapped kernels, but they still place the burden of discovering effective overlap strategies, encoding dependency structure, and validating correctness squarely on the programmer. In practice, users still need to write or modify DSL-level communication and computation code, explicitly manage signal/wait, buffer ownership, and memory ordering, and revalidate these low-level details for each operator and communication pattern.

2.3 Communication Backends

Modern GPU systems [19] expose several mechanisms for moving tensors across devices, each with distinct performance and programmability trade-offs (Tbl. 2). Copy Engines saturate NVLink [23] at 400 GB/s per direction on H100 and run independently of the SMs. Therefore, they do not consume compute resources and are ideal when communication can be decoupled from computation. However, they are typically driven by host APIs and can only transfer contiguous data, so high-dimensional strided tensors must be decomposed into many smaller transfers, each requiring a separate launch costing around 2-3μs, which can significantly reduce the effective bandwidth as each transfer time is also very short.

Table 2: Comparison of various GPU communication mechanisms.

	Hardware	Programming	Collective	Bandwidth
Copy Engine	Copy Engine	Host Launch	✗	✔✔✔
TMA	SM	Async. Instruction	✗	✔✔
Load/Store	SM	Sync. Instruction	✔	✔

Tensor Memory Accelerator (TMA) [19] paths achieve high bandwidth using dedicated asynchronous tensor-copy hardware, and can achieve a throughput of 300+ GB/s with only about 16 SMs issuing TMA instructions. This makes TMA attractive for overlapping structured tensor movement with computation within a node. At the same time, TMA must be launched by SM threads and does not currently support inter-node communication or in-network (switch-based) collective reduction, which limits its applicability to intra-node, point-to-point patterns.

Finally, plain load/store-based communication, often combined with registers and shared memory, attains slightly lower peak bandwidth than copy engines or TMA [33] but is significantly more flexible. These mechanisms integrate naturally with switch-based collective reduction (NVSHARP [20]), enabling fine-grained per-shard communication and in-network reductions. The downside is that they consume SM resources and are synchronous from the issuing warp's perspective, so they are harder to pipeline and require careful scheduling to hide communication latency.

3 Motivation

To fully exploit modern GPUs, distributed training systems must overlap computation and communication. Prior work mainly relies on coarse, kernel-level partitioning of computation kernels, which leaves substantial performance on the table. In this section, we revisit the motivating example in Figure 1 using detailed microbenchmarks (Figure 2), and show three key insights:

> ★***Insight 1: Limitations of Kernel-Level Overlap.*** *Kernel-level overlap is fundamentally limited by SM under-utilization and kernel-launch overheads.*

Figure 2(a) reports SM utilization for different GEMM

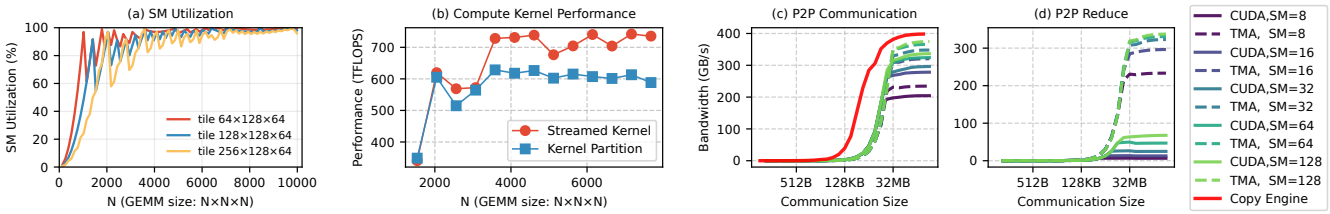


Figure 2: Motivation experiment results. (a) SM utilization under different GEMM sizes and tile sizes. (b) Performance comparison between a streamed GEMM kernel and a kernel-partitioned baseline. (c,d) Bandwidth of different communication backends under varying message sizes.

sizes under several commonly used tile-size configurations. Large GEMMs provide enough tile waves to saturate the SMs across all configurations. As the GEMM size decreases, fewer tile waves are generated, and the partially filled last wave dominates a larger portion of execution, causing SM utilization to drop due to wave quantization. Partitioning a GEMM into many sub-kernels forces each launch to operate on a smaller shape, pushing execution into exactly this low-utilization regime. This effect directly limits the benefit of kernel-level partition-based overlap (corresponding to ❷ in Figure 1), since overlapping many small kernels simply wastes SM capacity.

Figure 2(b) compares the end-to-end performance of (i) a baseline that partitions GEMM into multiple small kernels for overlap and (ii) a streamed GEMM kernel that internally pipelines tiles. Although both variants execute the same arithmetic operations, the kernel-partitioned baseline incurs substantial performance loss due to extra kernel launches (❶) and the SM under-utilization observed in Figure 2(a). In contrast, the streamed kernel maintains high utilization by exposing fine-grained compute tiles within a single launch, enabling overlap without fragmenting the workload. These results show that simply launching more kernels is not an effective path toward overlap; we need mechanisms that expose intra-kernel concurrency while preserving efficient GPU execution.

> ★***Insight 2: Granularity and Backend Effects.*** *Communication efficiency varies sharply with transfer granularity and backend selection.*

Communication efficiency varies sharply with transfer granularity and backend behavior. Figures 2(c) and 2(d) show the achieved bandwidth of different communication backends as we vary the transfer size and the number of SMs. Each backend shows distinct scaling behavior: some reach peak bandwidth at moderate transfer sizes, while others require larger transfers or more SMs to reach their full potential. Moreover, different backends support different communication patterns, such as point-to-point transfers versus reductions.

Taken together, these granularity and backend effects imply that the optimal configuration depends jointly on (i) the compute tile size, which determines the cadence at which results become available, (ii) the communication transfer size, which trades off latency and bandwidth, and (iii) the choice of communication backend. Coarse-grained kernel-level overlap cannot flexibly coordinate these parameters, since compute and communication are implemented as separate kernels with rigid interfaces. Intra-kernel overlap, by coordinating computation and communication at the same time, can align tile production with backend-specific sweet spots and select transfer granularities that sustain high utilization across SMs and copy engines.

> ★***Insight 3: A Unified Unit for Intra-Kernel Overlap.*** *Effective intra-kernel overlap therefore requires a communication unit with tunable granularity and a stable interface across communication backends.*

The combined granularity and backend effects indicate that effective overlap requires a communication unit whose size can match both the rate at which tiles produce data and the efficiency points of different communication backends. At the same time, this unit must provide a stable boundary between the high-level communication schedule and its backend-specific realization, so that schedules need not be rewritten for each backend. We therefore introduce a chunk abstraction that offers both tunable intra-kernel granularity and a unified interface for mapping communication onto diverse backends.

4 Overview

Syncopate is a compiler and runtime framework that turns locally written Triton kernels into distributed, fine-grained overlapped kernels. Rather than asking programmers to manually fuse communication and computation, Syncopate takes an existing local kernel and a high-level distribution specification as input, and automatically synthesizes an intra-kernel schedule that interleaves tile-wise communication with computation according to the available communication backends.

Input and User Interface. On the compute side, Syncopate consumes unmodified local Triton kernels annotated with

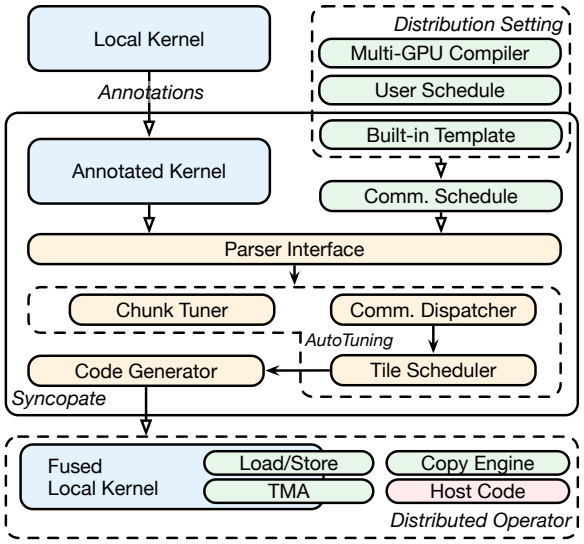


Figure 3: System overview of Syncopate.

Listing 1: Annotated Local Triton Kernel API.

```
@triton.jit
def kernel_gemm(a_ptr, b_ptr, ...):
    start_pid = tl.program_id(axis=0)
    # @sy.axis_count M block=BLOCK_SIZE_M
    num_pid_m = tl.cdiv(M, BLOCK_SIZE_M)
    # @sy.tile_id persistent
    tile_id = start_pid - NUM_SMS
    ...
    a_desc = tl.make_tensor_descriptor(a_ptr, ...)
    ...
    for _ in range(0, k_tiles * tiles_per_SM):
        tile_id += NUM_SMS
        # @sy.dispatch begin
        # @sy.pid_map M=pid_m N=pid_n
        pid_m, pid_n = get_pid_mn(tile_id,
                                  num_pid_m, ...)
        # @sy.dispatch end
        offs_am = pid_m * BLOCK_SIZE_M
        offs_bn = pid_n * BLOCK_SIZE_N
        offs_k = ki * BLOCK_SIZE_K
        a = a_desc.load([offs_am, offs_k])
        b = b_desc.load([offs_bn, offs_k])
        accumulator = tl.dot(a, b.T, accumulator)
```

Listing 2: Communication Schedule Example.

```
def all_gather_1d_swizzle(shape,dtype,axis,rank,...):
    plan = DevicePlan(dev=rank)
    plan.tensors_involved[buf] = (torch.Size(shape))
    local = shard(rank)
    plan.local_regions.setdefault(buf,[]).append(local)
    for i in range(mesh):
        peer = (i + rank) % mesh  # 1D swizzle
        if peer == rank: continue
        r = shard(peer)
        plan.add_op(Transfer(
                        op=TransferOp.PULL,
                        dst_buf=buf, dst_region=r,
                        src_buf=buf, src_region=r,
                        peer=peer, shard_idx=peer,
                    ...))
    return plan
```

lightweight scheduling metadata (Listing 1). Programmers write kernels as if they were running on a single device, using standard Triton primitives for indexing, tiling, and tensor descriptors. Lightweight Syncopate annotations (e.g., axis counts, tile identifiers, and dispatch regions) are required to identify the logical tiles and iteration structure but do not change the kernel's semantics. On the distribution side, Syncopate uses a communication plan that encodes the desired global data movement pattern and device topology (Listing 2). This plan is expressed using a small API defined by the users or imported directly from higher-level compilers searching parallel schedule.

To end users, Syncopate exposes a small set of APIs for (1) registering local kernels with their annotations, (2) constructing or selecting predefined communication plans (such as 1D/2D AllGather or ReduceScatter swizzles), and (3) compiling these into executable distributed kernels. High-level frameworks can wrap these APIs so that model authors only specify tensor partitioning and desired collectives, while Syncopate automatically generates the corresponding overlapped kernels. This separation of concerns allows experts to encode distribution and communication strategies once, while ordinary users invoke the resulting distributed operators through familiar, library-style calls.

Syncopate Architecture and Output. Given a local kernel and its communication plan, Syncopate lowers them into a unified dependence representation over tiles, shards, and communication operations. The compiler then searches for a fine-grained schedule that maps tiles to devices, assigns communication operations to appropriate backends (e.g., copy engine, TMA, or load/store), and inserts the necessary synchronization to respect both compute and communication dependencies. The output preserves the original numerical semantics while tightly pipelining asynchronous communication and computation around a single fused compute kernel. From the user's perspective, the generated operator can be invoked with the same signature as the original local kernel, plus standard distributed-runtime arguments (e.g., rank, world size, mesh), and integrated into existing training or inference code without further changes.

5 Syncopate: Fine-Grained Overlap Compiler

Syncopate automatically transforms locally written kernels into fine-grained overlapped distributed kernels by aligning the execution of local computation tiles with a global communication schedule. At a high level, we introduce a *chunk-centric* compilation pipeline that treats communication and computation symmetrically: communication is described as transfers of logical chunks, while computation is expressed as tiles that consume and produce these chunks. The compiler then derives dependencies between chunks and tiles,

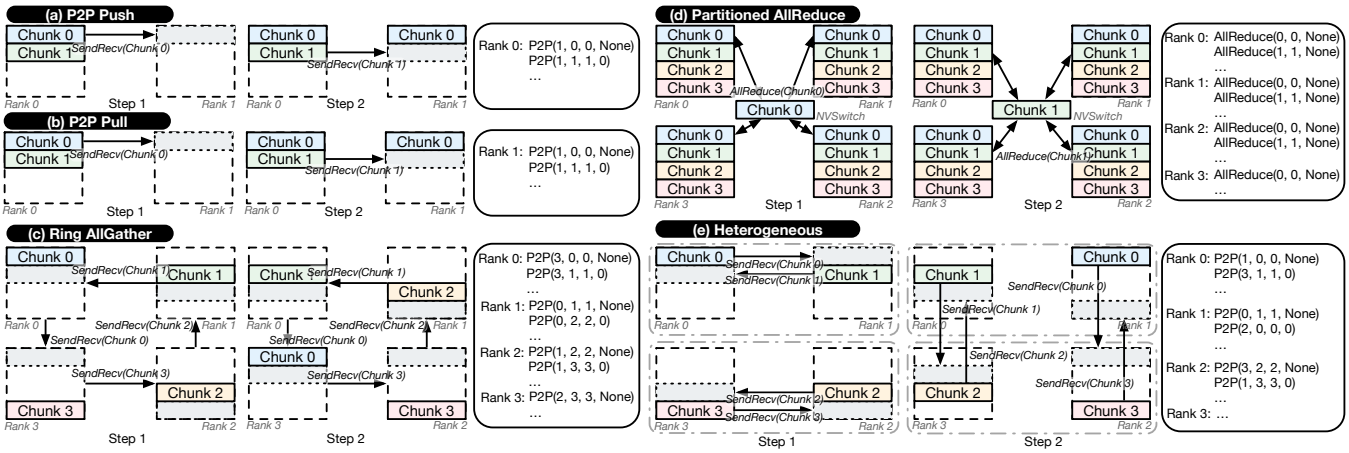


Figure 4: Communication schedule abstraction. (a) and (b) illustrate the same point-to-point exchange expressed as push and pull variants, respectively. (c) shows a ring-based AllGather pattern. (d) represents a partition-based AllReduce schedule. (e) depicts a heterogeneous swizzled AllGather pattern that pipelines communication across multiple hierarchy levels.

rewrites the tile scheduler to follow the communication order, inserts the necessary synchronization, and finally explores several implementation choices to generate high-performance overlapped code.

5.1 Communication Schedule Abstraction

We first define a communication-side abstraction that captures how data moves across devices independently of any particular local kernel implementation. This abstraction is built around the notion of a *chunk*, an intermediate layout between the global logical tensor and the local computation tiles. By operating at this intermediate granularity, the abstraction is expressive enough to describe a wide range of distributed schedules while remaining compatible with both partition-based and loop-based IRs in existing distributed compilers.

Definition. A *chunk* is a logical block of data that is communicated as a unit. Each chunk contains one or more tiles, where a tile is the basic unit of computation in the local kernel. Importantly, the chunk size in the communication schedule specifies *logical* transfers; the same logical chunk may later be implemented using different physical communication patterns or backends during lowering.

Conceptually, a chunk can be represented as: `chunk = Chunk(sizes=[...] , layout=..., tensor=...)`. Based on this abstraction, we define communication operators over chunks. We consider two primary classes of operators: point-to-point (P2P) transfers and collective communications.

- **P2P transfer** is represented as `P2P(src_rank, dst_rank, src_chunk, dst_chunk, dependency)`. This operator moves a chunk from a source rank to a destination rank, optionally guarded by a dependency on other chunks or operations. Note that for a pair of P2P operations on the source and destination ranks, we only include the operation on one side. If the P2P operation is defined on the source side, it represents a push operation; otherwise, it represents a pull operation. This will lead to different implementation choices during lowering.
- **Collective communication** is represented as `Collective(collective_type, src_chunk, dst_chunk, ranks, dependency)`. This operator applies a collective operation (e.g., AllGather, ReduceScatter) over a set of ranks on a given chunk with explicit dependency control. When explicitly defined as collective operations, the compiler can leverage the optimized collective implementations provided by the communication backends.

For both operator types, the `dependency` field encodes any ordering constraints that must be respected between communication operations. Specifically, it is represented as a `(rank, index)` tuple, indicating that the current operation cannot start until the specified operation on the given rank has completed. This allows us to express complex communication patterns, such as ring exchanges or multi-stage collectives, by chaining dependencies between chunks.

Upon this abstraction, a *communication schedule* is defined as a sequence of chunk-level communication operations with their associated dependencies on each rank as `schedule := [rank:Int, operations:List[CommOp]]:List`. Since there is no restriction on the operation for each rank, the communication schedule can express heterogeneous communication patterns where different ranks perform different operations on different chunks at different times.

Expressiveness. Despite its simplicity, the chunk abstraction is sufficiently expressive to capture a wide range of communication schedules covering all the overlap patterns used in practice (Fig. 4). To elaborate, (a) and (b) show the same P2P communication between two ranks expressed as push and pull operations, respectively, demonstrating the flexibility of

pull/push semantics. (c) illustrates a ring-based AllGather pattern, which is a common pattern for asynchronous distributed operators [18], where each rank sends and receives chunks in a pipelined manner, with dependencies ensuring the correct order of operations. (d) depicts a partition-based collective AllReduce pattern, where each rank contributes a chunk to the collective operation and performs the accumulation over the communication fabric. This pattern is often used in partition-based distributed compilers for kernel-level overlap. Finally, (e) shows a complex heterogeneous swizzled AllGather pattern advancing (c). By utilizing the port abstraction, each rank processes communication at different mesh hierarchy levels, enabling fine-grained pipelining and overlap across multiple dimensions. With this abstraction, different collective patterns, pipelined P2P exchanges, and hybrid schemes that combine intra-node and inter-node communication can all be written as sequences of chunk-level P2P and collective operators with explicit dependencies. Because chunks are defined in terms of logical tensor regions rather than concrete buffers, the same schedule can be reused across different kernels and tensor shapes, and later specialized by the compiler.

Lowering from Higher-Level Compiler IRs. Communication schedules in this abstraction can either be defined manually or automatically derived from existing distributed compiler IRs. In the manual case, users construct chunk objects and communication operators directly using our API as shown in Listing 2. In most cases, manual schedules are parameterized communication algorithms, not fully unrolled per-world-size plans: ring-based algorithms, for example, can be expressed with loops over ranks, chunks, or mesh levels, and Syncopate specializes them to the concrete world size, topology, and chunk configuration. Syncopate also provides predefined templates such as 1D/2D AllGather or ReduceScatter swizzles for the common communication patterns. The user can instantiate these templates with different chunk sizes, mesh topologies, communication axes, and pipeline stages to generate reusable communication schedules.

When integrating with a higher-level distributed compiler, Syncopate provides frontends for both partition-based and loop-based IRs. For partition-based IRs, we analyze the global data partitioning and the implied communication pattern between partitions to infer chunk sizes, participating ranks, and the corresponding P2P or collective operators. For loop-based IRs, we traverse loop nests, identify communication points, and group the communicated regions into chunks according to the chosen granularity. In both cases, the result is a uniform chunk-level schedule that decouples the high-level communication intent from any particular implementation (Listing 3). Specifically, the collective operators can be directly inserted ("direct") into our communication plan, or they can be further lowered to P2P communication operators using our templates ("template") or using some collective synthesis algorithms ("synth") such as TACOS [39].

Listing 3: Lowering from Higher-Level IR.

```
def emit_steps(steps, mesh, path="template"):
    comm = CommPlan()
    for step in steps:
        if step.is_p2p(): # push/pull/local_copy
            for rank in mesh:
                emit_p2p(comm.plans[rank], step, rank)
        else:  # collective
            for rank in mesh:
            # Path in ["direct", "synth", "template"]
                emit_collective(path, comm.plans[rank],
  step, rank)
    return comm

def lower_partition_ir(part_ir, axis_info, mesh, path="
 template"):
    steps = []
    for tensor in part_ir.tensors:
        layout = part_ir.placement[tensor]
        meta = axis_info[tensor]
        steps.extend(parse_partition_to_steps(tensor,
 layout, meta))
    return emit_steps(steps, mesh, path)

def lower_loop_ir(loop_ir, mesh, path="template"):
    steps = []
    for node in walk(loop_ir):
        steps.extend(parse_comm_intents(node))
    return emit_steps(steps, mesh, path)
```

5.2 Chunk-based Code Generation

Given a chunk-level communication schedule, the next step is to reorganize local computation so that tiles are executed in an order that aligns with the arrival and consumption of chunks. Intuitively, we want the kernel to compute exactly those tiles whose data has already been communicated, and to defer tiles whose input chunks are still in flight. This *compute chunk scheduling* bridges the gap between the communication abstraction and the tile-level structure of the original local kernel.

Compute Kernel Annotations. To make tile-level scheduling explicit, users provide lightweight annotations on the local computation kernel using Syncopate's API. These annotations do not change the numerical semantics of the kernel; instead, they expose its tiling structure and iteration order to the compiler. Although expressed as Python comments, they follow a structured directive format analogous to OpenMP [25] pragmas, allowing the compiler to reliably parse and verify them. Concretely, we require three pieces of information:

- **Tile size:** the logical shape of each tile along the relevant dimensions (e.g., GEMM blocks), which allows us to map tiles to chunks.
- **Tile index identifier:** a program variable (or tuple of variables) that uniquely identifies the tile being processed in a given iteration.
- **Tile scheduler:** the loop or control structure that advances the tile index and determines the order in which tiles are visited.

These annotations can often be derived from existing index-

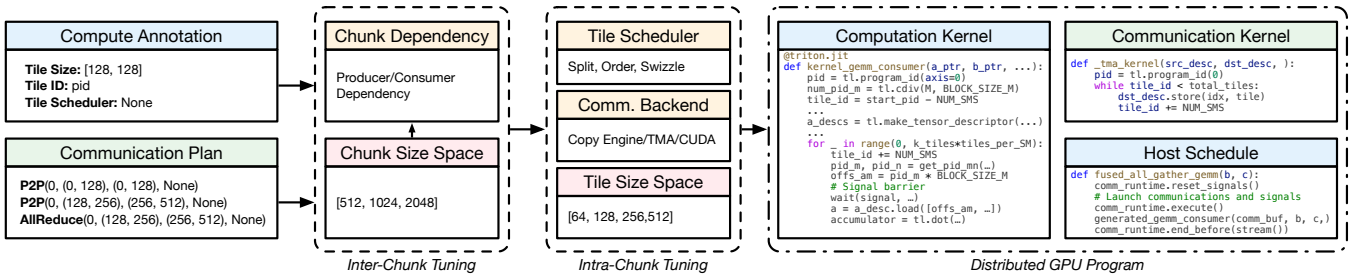


Figure 5: Compilation pipeline. In this example, we show communication using specialized SM as an independent kernel synchronized with signals. It can also be a fused kernel depending on the communication backend.

ing expressions and loop bounds with minor code changes, as illustrated in the overview section. These annotations are compiler metadata. They only expose the tile space, the tile identifier, and the mapping from a tile id to logical tensor-axis coordinates. Syncopate uses this information to relate the local Triton tile scheduler to the chunk-level schedule.

Dependency Parsing. With both the communication schedule and the annotated compute kernel in hand, Syncopate constructs a dependence graph over chunks and tiles according to the user-specified dependency intent in the plan, including how communication chunks depend on each other and how communication axes map to computation axes. For each chunk, we track its producer(s) and consumer(s), as well as any explicit ordering constraints encoded in the communication schedule (e.g., pipeline stages). For each tile, we determine which chunks it reads and writes based on its tile index, tensor layout, and the chunk-to-compute-axis mapping.

From this graph, the compiler identifies the minimal set of synchronization points needed to respect all data dependencies. Concretely, we insert wait operations in the kernel so that a tile that consumes a given chunk cannot start until the corresponding communication operator has completed. This synchronization can be implemented using different mechanisms depending on the chosen backend, but is always derived from the same chunk-level dependency structure.

Communication Code Generation. Once the communication schedule and tile scheduler have been aligned, Syncopate lowers the abstract chunk-level plan into concrete communication code. As illustrated in Fig. 7, the same logical schedule can be realized by several backends that differ in how they move chunks and how they allocate SM resources. Concretely, we support five realizations: (1) using the dedicated copy engine, (2) using TMA on a specialized SM, (3) using TMA on a co-located SM, (4) using operator-instruction load/store on a specialized SM, and (5) using operator-instruction load/store on a co-located SM. In all cases, tiles are produced on one side and consumed by operations on the other side, but the mechanism for signaling readiness and the division of compute versus communication work across SMs differ.

For each operator, Syncopate first builds a dependency graph over tiles and communication steps from the chunk schedule, then lowers this graph into backend-specific code that enforces all dependencies by construction. When targeting the copy engine or a specialized SM, the compiler emits global-memory signals and kernel launches so that communication progresses asynchronously relative to the main compute tiles. When targeting co-located SM backends, it instead generates shared-memory barriers and index bookkeeping to coordinate communication and computation within the same SM. Because all five realizations share the same logical schedule but expose different latency/bandwidth and resource tradeoffs, they form a search space for the autotuner: Syncopate automatically generates all valid implementations, measures their end-to-end performance, and selects the best-performing backend for each operator and hardware configuration.

Tile-Scheduler Swizzling. As visualized in Fig. 6, the communication plan and the original computation kernel typically induce different layouts over the global tensor: communication groups tiles into chunks based on where data needs to move, while the kernel groups tiles into waves based on its own traversal order. Prior work reconciles this mismatch by explicitly reordering data between communication and computation, paying extra global-memory traffic and synchronization. In contrast, Syncopate keeps the communicated chunks in-place and instead *swizzles* the tile scheduler at the intra-kernel level. We reorder the sequence of waves so that each chunk is consumed as soon as it arrives, and apply an intra-chunk swizzle that visits tiles in an order that preserves locality within the chunk. This chunk-based tile schedule aligns compute with communication progress without additional reordering kernels, enabling fine-grained overlap purely through scheduling. This transformation has two levels: at the chunk level, Syncopate reorders chunks to follow the communication schedule and consume chunks as they become ready; within each chunk, it applies a finer-grained tile scheduler, using the local kernel's original tile traversal by default while allowing alternative axis-based traversals when needed.

5.3 Communication-Centric Auto-Tuning

The chunk abstraction also provides a natural space for communication-centric auto-tuning. Because chunks sit exactly at the boundary between the global communication

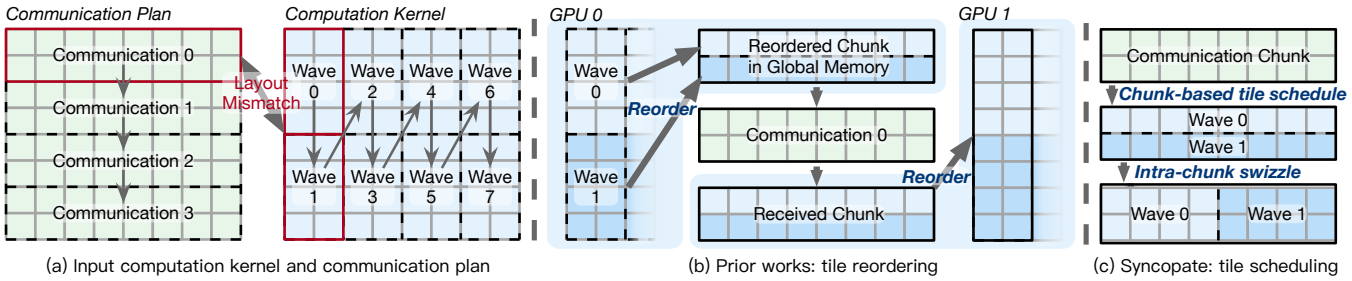


Figure 6: Tile scheduler transformation. (a) Computation and communication naturally follow different tile/chunk layouts, creating misalignment. (b) Prior approaches resolve this by inserting explicit data reordering between the two paths. (c) Syncopate instead rewrites the tile schedule to follow chunk order and applies intra-chunk swizzles for locality, aligning compute with communication progress without extra data movement.

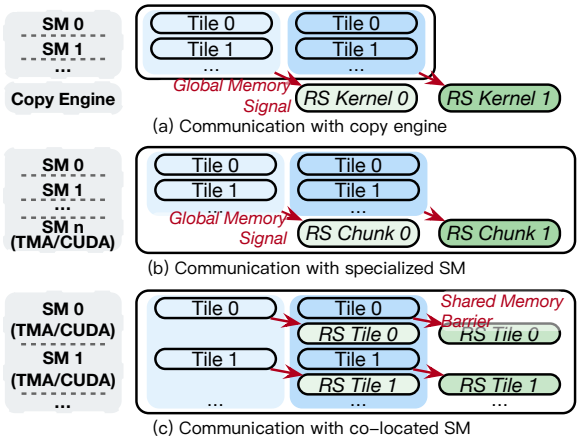


Figure 7: Communication backend selection. (a) Communication issued by the copy engine with global-memory signaling. (b) Dedicated SMs drive transfer. (c) Communication is co-located with compute on the same SM.

schedule and the local tile scheduler, changing chunk-level parameters simultaneously reshapes how data moves across ranks and how computation is ordered within each kernel. Rather than only tuning conventional kernel parameters (e.g., block sizes), Syncopate exposes a higher-level search space whose knobs directly control overlap and resource sharing. Given a local kernel and a chunk-level communication schedule, Syncopate uses an enumerate-and-measure tuning pass. The per-candidate compilation overhead is low because Syncopate performs lightweight source-to-source rewriting on Triton kernels and relies on the existing Triton JIT backend for code generation.

At the *inter-chunk* level, we tune the chunk size, shape, and split factor for each logical transfer. Larger chunks tend to achieve higher effective bandwidth on copy engines and TMA but reduce the granularity of overlap, while smaller chunks enable more fine-grained pipelining at the cost of higher per-chunk overhead and more synchronization. Different operators and model sizes favor different trade-off points: communication-heavy A2A-GEMM and GEMM-AR, for example, benefit from intermediate split factors that balance bandwidth and overlap, as confirmed by our sensitivity study in §6. The tuner searches this space under hardware-specific constraints (e.g., minimum efficient transfer size for copy engines and TMA alignment rules) and prunes configurations that would violate these hardware limits.

At the *intra-chunk* level, we tune both the computation tile configuration and how each chunk is realized by a communication backend. Given a fixed logical schedule, the compiler can instantiate each transfer using any of the backends in Fig. 7 (copy engine, intra- or inter-SM TMA, or CUDA load-/store on specialized or co-located SMs) and can vary the number of SMs assigned to communication when applicable. Some schedules benefit from using TMA for intra-node tensor movement and load/store-based communication for small, reduction-heavy shards, while others favor copy engines for large bulk transfers with minimal SM involvement. In parallel, the autotuner explores different tile sizes and intra-tile orders that better align compute waves with the chosen chunk layout, improving locality and avoiding long communication tails.

Crucially, all of these decisions operate on top of the same chunk-level dependence graph. Changing the backend, SM allocation, or tile order never requires re-deriving the global communication plan; instead, Syncopate reuses the existing schedule and regenerates backend-specific code that enforces the same dependencies. This separation of logical schedule from physical realization is what makes the search space both rich and manageable: as our ablation results show, reasonable but suboptimal settings can easily leave more than a factor of two in performance, while the tuned configuration found by our communication-centric autotuner consistently coincides with the most balanced point between computation, communication, and hardware utilization.

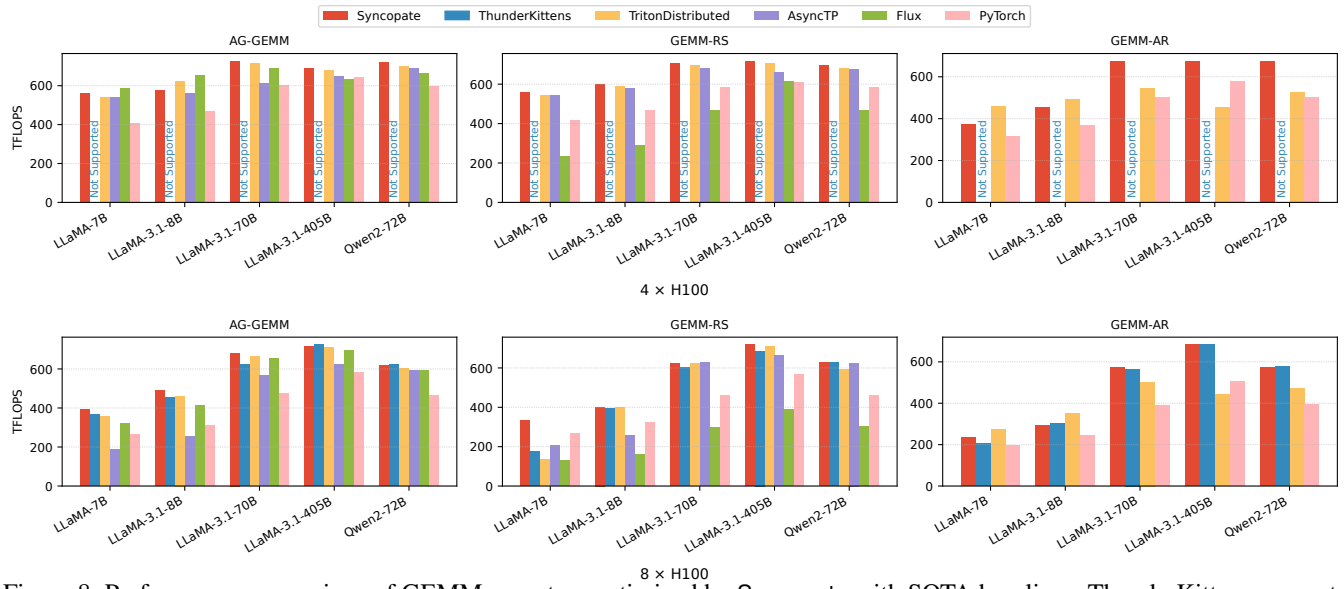


Figure 8: Performance comparison of GEMM operators optimized by Syncopate with SOTA baselines. ThunderKittens supports only 8 GPUs; 4-GPU is unsupported. When both settings are unsupported, the bar is omitted.

6 Evaluation

6.1 Experimental Setup

Testbed. We evaluate Syncopate on a server with 8 NVIDIA H100 GPUs connected via NVLink with an aggregate bandwidth of 900 GB/s, which is a representative single-node setting used in prior multi-GPU kernel studies [3, 13, 33, 46]. Unless otherwise stated, all measurements are taken on a single node using all 8 GPUs; in later experiments, we vary the number of active devices to study scalability and portability. Syncopate is implemented with CUDA v12.9, NVSHMEM v3.3.9, and PyTorch v2.7, and we run all baselines on the same software stack to ensure a fair comparison.

Workloads. We use Syncopate to optimize representative multi-GPU operators that dominate the cost of modern LLM workloads: general matrix multiplication (GEMM) and attention [7, 8, 28, 35]. For GEMM, we benchmark three distributed variants: AllGather–GEMM (AG-GEMM) and GEMM–ReduceScatter (GEMM-RS), and GEMM–AllReduce (GEMM-AR), which appear in tensor-parallel [31] or sequence parallel [17] feed-forward network (FFN) layers. For attention, we evaluate both head-parallel (HP) [15] and sequence-parallel (SP) schedules, including the overlapped RingAttention (Ring-Attn) [18] variant.

Operator shapes are derived from the FFN layers and attention layers of open-source Llama-3 [11] and Qwen [26] models, covering a range of hidden dimensions, head counts, and parallelism configurations that are typical of large-scale LLM deployments. For attention, we sweep over multiple sequence lengths to reflect common short- and long-context use cases under different distribution strategies. Overall, this workload suite exercises Syncopate across both regular GEMM-heavy and more irregular attention patterns.

Baselines. To assess the effectiveness of operators generated and tuned by Syncopate, we compare against both state-of-the-art manually engineered kernels and fully automatic compiler-based approaches. As manual baselines, we include built-in operators from fine-grained overlap DSLs such as ThunderKittens [32, 33] and TritonDistributed [45, 46], as well as highly optimized implementations including AsyncTP [38], Flux [3], and Triton kernels paired with NCCL [24] collectives.

To isolate the benefit of Syncopate's automatic fine-grained overlap, we further compare against automatic operators produced by existing distributed compiler frameworks, including Domino [36], Alpa [44], and Mercury [13]. For these comparisons, we transform the communication schedules found by each compiler into our chunk-level representation and reuse the same high-level plans in Syncopate, so that any performance difference reflects our intra-kernel overlap and backend-selection mechanisms rather than differences in global parallelization strategy.

Overlap Granularity. AsyncTP, Flux, TritonDistributed, and ThunderKittens perform fine-grained intra-kernel overlap by explicitly coordinating communication and computation inside specialized kernels or DSL programs. Domino, Mercury, and the PyTorch RingAttention baseline use inter-kernel overlap, while the remaining baselines execute communication through separate library collectives without overlap.

Base Triton Kernels. Our GEMM local kernels are off-the-shelf Triton GEMM kernels with only Syncopate annotations added. For attention, we start from standard Triton attention kernels and make minor modifications to support split-KV ex-

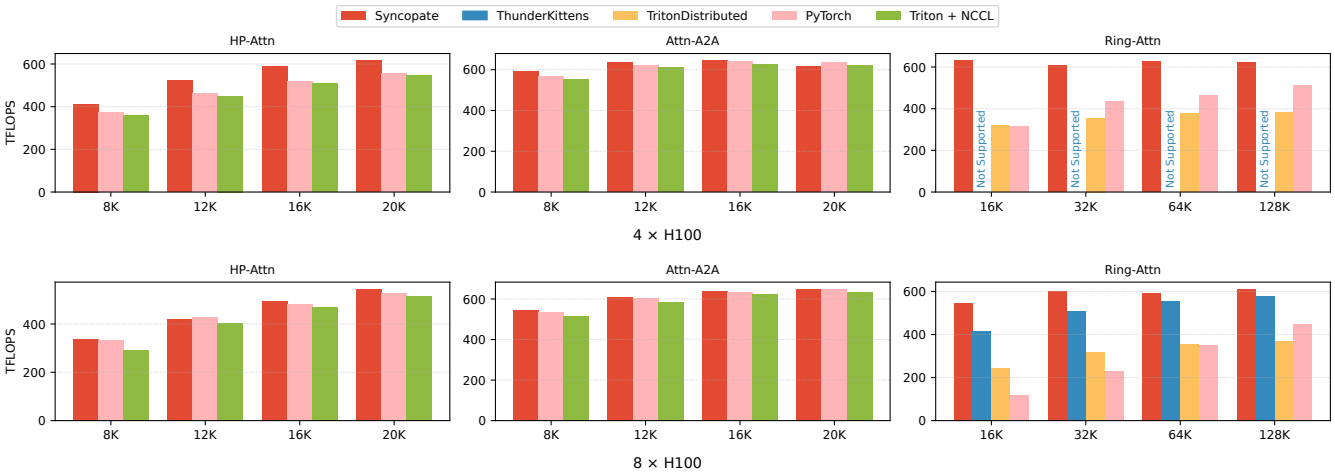


Figure 9: Performance comparison of operators optimized by Syncopate with SOTA baselines.

ecution, which is not exposed by the original kernels. In both cases, the kernels remain local compute kernels; Syncopate generates the communication logic, synchronization, backend selection, and tile-schedule transformations.

6.2 Performance Benchmark

Operator Results. Across all evaluated settings, Syncopate generates automatically optimized operators that are competitive with carefully hand-engineered baselines, while showing even larger gains over fully automatic distributed compilers. As summarized in Fig. 8 and Fig. 9, Syncopate sustains high TFLOPS on both GEMM and attention operators under multiple communication patterns (AG-GEMM, GEMM-RS, GEMM-AR, HP/SP attention, and Ring-Attn) and model configurations derived from Llama-3 and Qwen.

In Fig. 8, Syncopate is at or near the best-performing curve in almost every GEMM configuration. On common, heavily optimized AG-GEMM and GEMM-RS cases where manual kernels such as ThunderKittens, TritonDistributed, AsyncTP, and Flux already sit close to the hardware limits, Syncopate essentially matches their peak throughput on both 4- and 8-GPU settings, achieving on average 99.8% of the best baseline on 4 GPUs and 104% on 8 GPUs. For GEMM-AR, Syncopate is marginally below TritonDistributed on 7B/8B shapes, yet it scales more effectively and becomes the top-performing kernel on larger model configurations. These results indicate that our generic compilation pipeline can recover the same highly tuned overlap patterns that experts design by hand. The small advantage over the strongest baseline in some 8-GPU cases comes from the chunk-level abstraction. By grouping multiple tiles under a communication chunk, Syncopate reduces the number of fine-grained readiness checks and synchronization points while still preserving overlap.

In Fig. 9, we observe a similar trend for attention operators. Under standard HP attention with moderate sequence lengths, Syncopate closely tracks the best manual implementations, confirming that the chunk-based abstraction does not sacrifice performance even for workloads that already benefit from hand-optimized kernels. As the problem becomes harder, moving to Ring-Attn, longer sequences, and 8-GPU runs, the gap widens in favor of Syncopate. Our compiler maintains high TFLOPS while baseline kernels degrade more rapidly, since it can reshape chunks, rebalance compute and communication, and choose different backends as the sequence length and parallelism strategy change. Notably, on the most communication-intensive Ring-Attn settings, Syncopate delivers the best performance despite not being tailored specifically for this operator.

Integration Results. We further evaluate how Syncopate composes with existing automatic distributed compilers that search for the communication schedule among devices, using their communication plans as our inputs. As summarized in Fig. 10, for each of Domino, Alpa, and Mercury, we keep the original parallelization strategy and the searched communication schedule fixed, convert that schedule into our chunk-level representation, and let Syncopate generate the fine-grained overlapped kernels. Across both GEMM and attention workloads on 4- and 8-H100 configurations, integrating Syncopate consistently reduces end-to-end operator latency compared to the native implementations shipped with these systems, showing that chunk-based intra-kernel overlap exposes an additional optimization dimension on top of their global parallelization decisions.

This experiment also illustrates that Syncopate can be cleanly integrated with both partition-based and loop-based compiler stacks. Domino and Alpa operate on partitioned IRs and expose their communication plans as sequences of collectives between logical tensor partitions, while Mercury is built around a loop-centric IR for ring [18] and double-ring attention [12]. Because our interface only requires a well-defined, implementation-agnostic communication schedule and lightweight annotations on the local kernels, these systems can plug into Syncopate with minimal source modifications. This

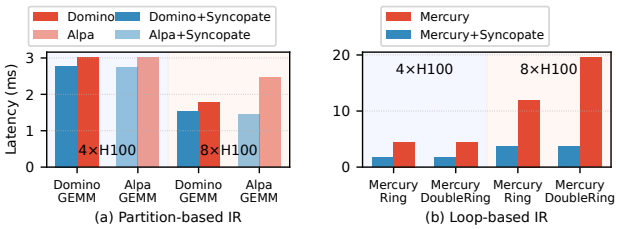


Figure 10: Evaluation of lowering partition-based and loop-based IRs generated by higher-level distributed compilers into Syncopate.

compatibility allows practitioners to reuse mature distributed compilers for global partitioning, while relying on Syncopate to automatically realize high-performance, fine-grained overlap within each generated operator.

6.3 Ablation and Sensitivity Studies

We next evaluate the effect of Syncopate's chunk-based code generation and communication-centric auto-tuning components through a series of ablation and sensitivity analyses in Fig. 11. Each subplot corresponds to one of the key design choices introduced in §5: communication backend selection and SM allocation, chunk size (split factor), and intra-tile scheduling.

Communication Backend and SM Tuning. Fig. 11(a) and (c) study how different realizations of the same logical communication schedule perform under the backends described in § 5.2, and how tuning the number of active SMs further refines this choice. In (a), for GEMM-RS and AG-GEMM, copy-engine and intra-SM TMA backends achieve the highest TFLOPS, while purely CUDA load/store realizations saturate at much lower throughput. The gap between the best and worst backend for the same logical schedule is comparable to the gap between our final implementation and several baselines in Fig. 8, indicating that backend selection alone can determine whether overlap is effective. This confirms that the ability to instantiate the same chunk schedule with different backends is crucial: no single mechanism dominates across operators, and picking a suboptimal backend can leave more than half of the available performance on the table. In (c), for a fixed backend, varying the number of SMs devoted to communication reveals a clear sweet spot where computation and communication are balanced. For both 405B and 70B GEMMs, allocating too few SMs underutilizes the link bandwidth, whereas allocating too many starves the main kernel. The optimal SM count also shifts with model size, which matches the design of our backend code generation: the autotuner treats SM allocation as a first-class knob and automatically selects a near-optimal point for each operator/hardware pair instead of relying on a single hard-coded ratio.

Chunk Size Tuning. Fig. 11(b) varies the number of chunks (split factor) used to realize the same high-level schedule for A2A-GEMM and GEMM-AR. Smaller split factors correspond to larger chunks with higher single-transfer efficiency but fewer overlap opportunities, while larger split factors increase overlap at the cost of per-chunk overhead. The curves exhibit a clear non-monotonic trend, with performance peaking at an intermediate split (e.g., 2–3 splits, about 128MB for GEMM-AR) and degrading when chunks become either too coarse or too fine. Notably, naive choices that are convenient to implement by hand (such as a single large chunk or splitting once per rank) sit far from the optimum. This behavior directly reflects the trade-off discussed in our chunk-based code generation: practitioners cannot reliably pick a single "good" chunk size by hand, whereas Syncopate searches this space automatically and selects the configuration that best matches the operator's compute/communication balance.

Intra-Tile Scheduling. Finally, Fig. 11(d) explores different intra-tile scheduling strategies for a representative GEMM configuration, varying the order in which tiles within each chunk are visited. Each point corresponds to a valid schedule with tile size on M, N, K dimensions and the pipeline stages that preserve program semantics but change locality and load balance. To represent different search candidates, we calculate the consumed shared memory size and plot the performance of these valid schedules. The wide spread in TFLOPS shows that tile order alone can introduce more than a 2 × performance difference, reinforcing the importance of the tile-scheduler transformation in § 5.2. High-performing schedules cluster around orders that align tile waves with the communication chunk order introduced in our chunk-based code generation, whereas poorly performing ones repeatedly revisit tiles in a way that destroys locality or creates long tails of unfinished work. By generating and evaluating these schedules automatically, Syncopate converges on tile orders that co-optimize cache reuse and SM utilization, without requiring users to reason manually about low-level tiling and swizzling policies.

Taken together, these studies demonstrate that Syncopate's chunk abstraction is not only expressive but also forms a practical search space: the same logical schedule can be realized via multiple backends, chunk sizes, SM allocations, and tile orders, and the differences between reasonable but non-optimal choices and the tuned configuration are often comparable to, or larger than, the gaps between systems in our main benchmarks. The auto-tuning framework in § 5.3 systematically explores this space using the code generation mechanisms of § 5.2, turning what would otherwise be a brittle, hand-tuned process into a robust, compiler-driven optimization.

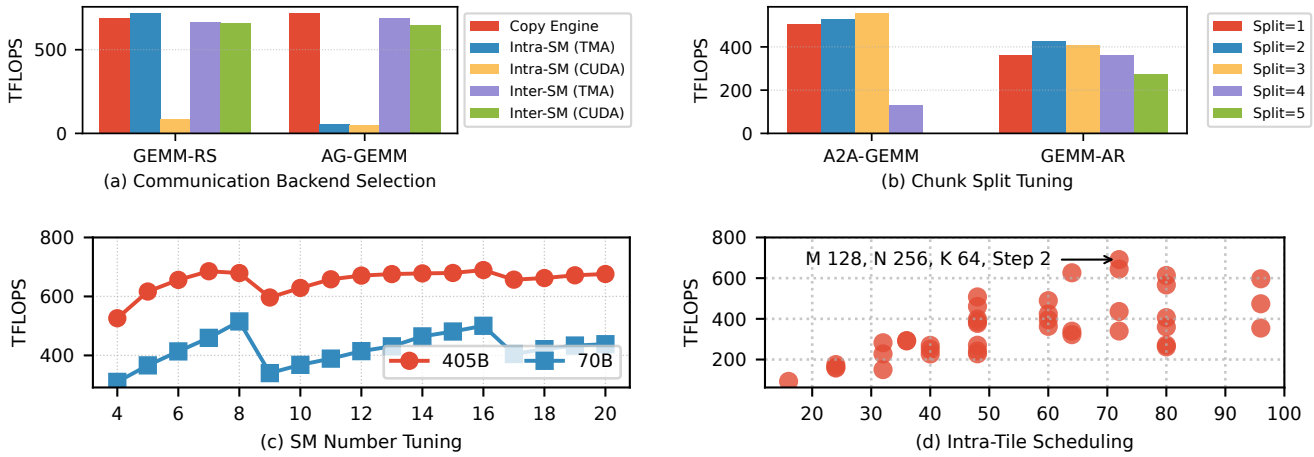


Figure 11: Ablation and sensitivity studies of Syncopate's auto-tuning design space.

7 Discussion and Future Work

Extending to other DSLs. Syncopate is implemented as a Triton source-to-source compiler, but the core abstraction only requires a tile-centric frontend that exposes tile shapes, tile identifiers, tile traversal order, and tile memory regions. Thus, DSLs such as CuTeDSL [21] or cuTile [22] could be supported with an additional frontend/lowering layer, as long as kernels are written in a tiled style. The chunk schedules, dependency construction, and backend-selection logic would remain reusable.

Scope of compute-memory overlap. Syncopate targets fine-grained overlap between tiled computation and explicit inter-GPU communication. Overlapping independent compute-bound and memory-bound kernels, as in NanoFlow-style optimizations [47], is largely orthogonal and can often be handled by stream-level scheduling. Such optimizations could be modeled by Syncopate only when the stages must be coordinated as chunked producer-consumer relationships with tile-level dependencies.

Dynamically shaped workloads. Syncopate currently specializes kernels to a fixed operator shape or a small set of common shape buckets. For variable sequence lengths, the chunk schedule can usually be re-instantiated with the new shape parameters; when chunk ranges are stored as device-side metadata, a lightweight update kernel can refresh these values before launching the generated kernel. A full dynamic-shape runtime that automatically manages shape buckets and metadata updates is left to future work.

Multi-node settings. Syncopate's chunk abstraction naturally extends to multi-node settings, since larger chunks are also a suitable unit for bandwidth-efficient NIC transfers. The main extension is not to generalize tiles or chunks, but to enrich the communication plan with a channel dimension that distinguishes intra-node and inter-node paths. Different channels can progress concurrently, allowing NVLink transfers and NIC transfers to be scheduled and overlapped independently. The compiler can then assign chunks to channels according to the topology and use different backend realizations for each channel. Supporting this setting requires additional backend and runtime support for inter-node communication, which we leave to future work.

Why a compiler abstraction? Syncopate is not meant to replace expert-written kernels for a fixed operator and hardware target. Its goal is to avoid re-encoding the same fine-grained overlap logic, signal/wait protocol, and backend-specific communication code for each schedule or parallel configuration. The chunk abstraction lets users or higher-level compilers specify the logical communication policy once, while Syncopate lowers it to concrete dependencies, synchronization, backend choices, and tile schedules.

8 Conclusion

Syncopate bridges the abstraction gap between communication planning and fine-grained overlapping by introducing a chunk-based interface that unifies communication plans with tiled GPU computation. By automatically aligning tile schedules with communication progress and selecting among heterogeneous backends, Syncopate turns intra-kernel overlap into a general compiler capability rather than a hand-engineered optimization. The framework integrates cleanly with existing distributed compilers, complementing their global parallelization strategies with backend-aware, fine-grained scheduling. This decoupled design provides a foundation for systems that co-schedule computation and communication and accommodate diverse communication backends.

References

[1] Sami Alabed, Daniel Belov, Bart Chrzaszcz, Juliana Franco, Dominik Grewe, Dougal Maclaurin, James Molloy, Tom Natan, Tamara Norman, Xiaoyue Pan, et al. Partir: Composing spmd partitioning strategies for machine learning. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1*, pages 794–810, 2025.

[2] Jason Ansel, Edward Yang, Horace He, Natalia Gimelshein, Animesh Jain, Michael Voznesensky, Bin Bao, Peter Bell, David Berard, Evgeni Burovski, Geeta Chauhan, Anjali Chourdia, Will Constable, Alban Desmaison, Zachary DeVito, Elias Ellison, Will Feng, Jiong Gong, Michael Gschwind, Brian Hirsh, Sherlock Huang, Kshiteej Kalambarkar, Laurent Kirsch, Michael Lazos, Mario Lezcano, Yanbo Liang, Jason Liang, Yinghai Lu, C. K. Luk, Bert Maher, Yunjie Pan, Christian Puhrsch, Matthias Reso, Mark Saroufim, Marcos Yukio Siraichi, Helen Suk, Shunting Zhang, Michael Suo, Phil Tillet, Xu Zhao, Eikan Wang, Keren Zhou, Richard Zou, Xiaodong Wang, Ajit Mathews, William Wen, Gregory Chanan, Peng Wu, and Soumith Chintala. Pytorch 2: Faster machine learning through dynamic python bytecode transformation and graph compilation. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, ASPLOS '24, page 929–947, New York, NY, USA, 2024. Association for Computing Machinery.

[3] Li-Wen Chang, Wenlei Bao, Qi Hou, Chengquan Jiang, Ningxin Zheng, Yinmin Zhong, Xuanrun Zhang, Zuquan Song, Chengji Yao, Ziheng Jiang, et al. Flux: fast software-based communication overlap on gpus through kernel fusion. *arXiv preprint arXiv:2406.06858*, 2024.

[4] Chang Chen, Xiuhong Li, Qianchao Zhu, Jiangfei Duan, Peng Sun, Xingcheng Zhang, and Chao Yang. Centauri: Enabling efficient scheduling for communication-computation overlap in large model training via communication partitioning. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, pages 178–191, 2024.

[5] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, et al. {TVM}: An automated {End-to-End} optimizing compiler for deep learning. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 578–594, 2018.

[6] Tianqi Chen, Lianmin Zheng, Eddie Yan, Ziheng Jiang, Thierry Moreau, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. Learning to optimize tensor programs. In *Proceedings of the 32nd International Conference on Neural Information Processing Systems*, NIPS'18, page 3393–3404, Red Hook, NY, USA, 2018. Curran Associates Inc.

[7] Tri Dao. Flashattention-2: Faster attention with better parallelism and work partitioning. *arXiv preprint arXiv:2307.08691*, 2023.

[8] Tri Dao, Dan Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. Flashattention: Fast and memory-efficient exact attention with io-awareness. *Advances in neural information processing systems*, 35:16344–16359, 2022.

[9] Siyuan Feng, Bohan Hou, Hongyi Jin, Wuwei Lin, Junru Shao, Ruihang Lai, Zihao Ye, Lianmin Zheng, Cody Hao Yu, Yong Yu, et al. Tensorir: An abstraction for automatic tensorized program optimization. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, pages 804–817, 2023.

[10] Raja Gond, Nipun Kwatra, and Ramachandran Ramjee. Tokenweave: Efficient compute-communication overlap for distributed llm inference, 2025.

[11] Aaron Grattafiori, Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Alex Vaughan, et al. The llama 3 herd of models. *arXiv preprint arXiv:2407.21783*, 2024.

[12] Diandian Gu, Peng Sun, Qinghao Hu, Ting Huang, Xun Chen, Yingtong Xiong, Guoteng Wang, Qiaoling Chen, Shangchun Zhao, Jiarui Fang, et al. Loongtrain: Efficient training of long-sequence llms with head-context parallelism. *arXiv preprint arXiv:2406.18485*, 2024.

[13] Yue Guan, Xinwei Qiang, Zaifeng Pan, Daniels Johnson, Yuanwei Fang, Keren Zhou, Yuke Wang, Wanlu Li, Yufei Ding, and Adnan Aziz. Mercury: Unlocking multi-gpu operator optimization for llms via remote memory scheduling. In *Proceedings of the ACM SIGOPS 31st Symposium on Operating Systems Principles*, SOSP '25, page 1046–1061, New York, NY, USA, 2025. Association for Computing Machinery.

[14] Ke Hong, Xiuhong Li, Minxu Liu, Qiuli Mao, Tianqi Wu, Zixiao Huang, Lufang Chen, Zhong Wang, Yichong Zhang, Zhenhua Zhu, et al. Flashoverlap: A lightweight design for efficiently overlapping communication and computation. *arXiv preprint arXiv:2504.19519*, 2025.

[15] Sam Ade Jacobs, Masahiro Tanaka, Chengming Zhang, Minjia Zhang, Shuaiwen Leon Song, Samyam Rajbhandari, and Yuxiong He. Deepspeed ulysses: System optimizations for enabling training of extreme long sequence transformer models. *arXiv preprint arXiv:2309.14509*, 2023.

[16] Abhinav Jangda, Jun Huang, Guodong Liu, Amir Hossein Nodehi Sabet, Saeed Maleki, Youshan Miao, Madanlal Musuvathi, Todd Mytkowicz, and Olli Saarikivi. Breaking the computation and communication abstraction barrier in distributed machine learning workloads. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 402–416, 2022.

[17] Vijay Anand Korthikanti, Jared Casper, Sangkug Lym, Lawrence McAfee, Michael Andersch, Mohammad Shoeybi, and Bryan Catanzaro. Reducing activation recomputation in large transformer models. *Proceedings of Machine Learning and Systems*, 5:341–353, 2023.

[18] Hao Liu, Matei Zaharia, and Pieter Abbeel. Ringattention with blockwise transformers for near-infinite context. In *The Twelfth International Conference on Learning Representations*.

[19] NVIDIA. NVIDIA H100 Tensor Core GPU Architecture. Technical report, NVIDIA, mar 2022. White paper.

[20] NVIDIA. NVIDIA Scalable Hierarchical Aggregation and Reduction Protocol (SHARP) Rev 3.0.0, 2024. Accessed: 2025-02-10.

[21] NVIDIA. CuTe DSL. https://docs.nvidia.com/cutlass/latest/media/docs/pythonDSL/cute_dsl.html, 2026. Accessed: 2026-04-18.

[22] NVIDIA. cuTile Python. https://docs.nvidia.com/cuda/cutile-python/, 2026. Accessed: 2026-04-18.

[23] NVIDIA Corporation. Nvidia nvlink high-speed interconnect: Application performance. Technical report, NVIDIA Corporation, 2015. Accessed: 2025-04-16.

[24] NVIDIA Corporation. *NVIDIA Collective Communications Library (NCCL)*, 2025. Version 2.26.2.

[25] OpenMP Architecture Review Board. Openmp application programming interface. https://www.openmp.org/specifications/, 2023.

[26] Qwen, :, An Yang, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chengyuan Li, Dayiheng Liu, Fei Huang, Haoran Wei, Huan Lin, Jian Yang, Jianhong Tu, Jianwei Zhang, Jianxin Yang, Jiaxi Yang, Jingren Zhou, Junyang Lin, Kai Dang, Keming Lu, Keqin Bao, Kexin Yang, Le Yu, Mei Li, Mingfeng Xue, Pei Zhang, Qin Zhu, Rui Men, Runji Lin, Tianhao Li, Tianyi Tang, Tingyu Xia, Xingzhang Ren, Xuancheng Ren, Yang Fan, Yang Su, Yichang Zhang, Yu Wan, Yuqiong Liu, Zeyu Cui, Zhenru Zhang, and Zihan Qiu. Qwen2.5 technical report, 2025.

[27] Keshav Santhanam, Siddharth Krishna, Ryota Tomioka, Tim Harris, and Matei Zaharia. Distir: An intermediate representation and simulator for efficient neural network distribution. *arXiv preprint arXiv:2111.05426*, 2021.

[28] Jay Shah, Ganesh Bikshandi, Ying Zhang, Vijay Thakkar, Pradeep Ramani, and Tri Dao. Flashattention-3: Fast and accurate attention with asynchrony and low-precision. *Advances in Neural Information Processing Systems*, 37:68658–68685, 2024.

[29] Junru Shao, Xiyou Zhou, Siyuan Feng, Bohan Hou, Ruihang Lai, Hongyi Jin, Wuwei Lin, Masahiro Masuda, Cody Hao Yu, and Tianqi Chen. Tensor program optimization with probabilistic programs. *Advances in Neural Information Processing Systems*, 35:35783–35796, 2022.

[30] Noam Shazeer, Youlong Cheng, Niki Parmar, Dustin Tran, Ashish Vaswani, Penporn Koanantakool, Peter Hawkins, HyoukJoong Lee, Mingsheng Hong, Cliff Young, Ryan Sepassi, and Blake Hechtman. Mesh-tensorflow: Deep learning for supercomputers, 2018.

[31] Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. Megatron-lm: Training multi-billion parameter language models using model parallelism. *arXiv preprint arXiv:1909.08053*, 2019.

[32] Benjamin F. Spector, Simran Arora, Aaryan Singhal, Daniel Y. Fu, and Christopher Ré. Thunderkittens: Simple, fast, and adorable ai kernels, 2024.

[33] Stuart H. Sul, Simran Arora, Benjamin F. Spector, and Christopher Ré. Parallelkittens: Systematic and practical simplification of multi-gpu ai kernels, 2025.

[34] Philippe Tillet, H. T. Kung, and David Cox. Triton: an intermediate language and compiler for tiled neural network computations. In *Proceedings of the 3rd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*, MAPL 2019, page 10–19, New York, NY, USA, 2019. Association for Computing Machinery.

[35] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. *Advances in neural information processing systems*, 30, 2017.

[36] Guanhua Wang, Chengming Zhang, Zheyu Shen, Ang Li, and Olatunji Ruwase. Domino: Eliminating communication in llm training via generic tensor slicing and overlapping, 2024.

[37] Haoran Wang, Lei Wang, Haobo Xu, Ying Wang, Yuming Li, and Yinhe Han. Primepar: Efficient spatial-temporal tensor partitioning for large transformer model training. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, ASPLOS '24, page 801–817, New York, NY, USA, 2024. Association for Computing Machinery.

[38] Shibo Wang, Jinliang Wei, Amit Sabne, Andy Davis, Berkin Ilbeyi, Blake Hechtman, Dehao Chen, Karthik Srinivasa Murthy, Marcello Maggioni, Qiao Zhang, Sameer Kumar, Tongfei Guo, Yuanzhong Xu, and Zongwei Zhou. Overlap communication with dependent computation via decomposition in large deep learning models. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1*, ASPLOS 2023, page 93–106, New York, NY, USA, 2022. Association for Computing Machinery.

[39] William Won, Midhilesh Elavazhagan, Sudarshan Srinivasan, Swati Gupta, and Tushar Krishna. Tacos: Topology-aware collective algorithm synthesizer for distributed machine learning. In *Proceedings of the 2024 57th IEEE/ACM International Symposium on Microarchitecture*, MICRO '24, page 856–870. IEEE Press, 2024.

[40] Mengdi Wu, Xinhao Cheng, Shengyu Liu, Chunan Shi, Jianan Ji, Man Kit Ao, Praveen Velliengiri, Xupeng Miao, Oded Padon, and Zhihao Jia. Mirage: A {Multi-Level} superoptimizer for tensor programs. In *19th USENIX Symposium on Operating Systems Design and Implementation (OSDI 25)*, pages 21–38, 2025.

[41] Yuanzhong Xu, HyoukJoong Lee, Dehao Chen, Blake Hechtman, Yanping Huang, Rahul Joshi, Maxim Krikun, Dmitry Lepikhin, Andy Ly, Marcello Maggioni, Ruoming Pang, Noam Shazeer, Shibo Wang, Tao Wang, Yonghui Wu, and Zhifeng Chen. Gspmd: General and scalable parallelization for ml computation graphs, 2021.

[42] Shulai Zhang, Ningxin Zheng, Haibin Lin, Ziheng Jiang, Wenlei Bao, Chengquan Jiang, Qi Hou, Weihao Cui, Size Zheng, Li-Wen Chang, et al. Comet: Fine-grained computation-communication overlapping for mixture-of-experts. *arXiv preprint arXiv:2502.19811*, 2025.

[43] Lianmin Zheng, Chengfan Jia, Minmin Sun, Zhao Wu, Cody Hao Yu, Ameer Haj-Ali, Yida Wang, Jun Yang, Danyang Zhuo, Koushik Sen, Joseph E. Gonzalez, and Ion Stoica. Ansor: generating high-performance tensor programs for deep learning. In *Proceedings of the 14th USENIX Conference on Operating Systems Design and Implementation*, OSDI'20, USA, 2020. USENIX Association.

[44] Lianmin Zheng, Zhuohan Li, Hao Zhang, Yonghao Zhuang, Zhifeng Chen, Yanping Huang, Yida Wang, Yuanzhong Xu, Danyang Zhuo, Eric P Xing, et al. Alpa: Automating inter-and {Intra-Operator} parallelism for distributed deep learning. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 559–578, 2022.

[45] Size Zheng, Wenlei Bao, Qi Hou, Xuegui Zheng, Jin Fang, Chenhui Huang, Tianqi Li, Haojie Duanmu, Renze Chen, Ruifan Xu, et al. Triton-distributed: Programming overlapping kernels on distributed ai systems with the triton compiler. *arXiv preprint arXiv:2504.19442*, 2025.

[46] Size Zheng, Jin Fang, Xuegui Zheng, Qi Hou, Wenlei Bao, Ningxin Zheng, Ziheng Jiang, Dongyang Wang, Jianxi Ye, Haibin Lin, et al. Tilelink: Generating efficient compute-communication overlapping kernels using tile-centric primitives. *arXiv preprint arXiv:2503.20313*, 2025.

[47] Kan Zhu, Yufei Gao, Yilong Zhao, Liangyu Zhao, Gefei Zuo, Yile Gu, Dedong Xie, Tian Tang, Qinyu Xu, Zihao Ye, Keisuke Kamahori, Chien-Yu Lin, Ziren Wang, Stephanie Wang, Arvind Krishnamurthy, and Baris Kasikci. Nanoflow: Towards optimal large language model serving throughput, 2025.

A Artifact Appendix

Abstract

The Syncopate artifact contains the source code for the chunk-centric compiler and runtime described in this paper, together with unit tests, end-to-end multi-GPU tests, experiment entry points, and a Dockerfile for the GPU software environment. A short CPU-only workflow validates the core communication-plan representation, signal planning, lowering logic, and compiler passes. On a multi-GPU NVIDIA Hopper-class system, the artifact also supports correctness and performance evaluation of generated GEMM and attention operators.

Scope

The artifact can be used to validate three central claims of the paper: (1) chunk-level communication plans can express reusable communication schedules and lower them to per-rank execution plans; (2) Syncopate can transform local tiled kernels into correct fine-grained overlapped multi-GPU kernels; and (3) the generated operators can be benchmarked across communication backends and scheduling choices. The CPU-only tests validate the compiler abstractions and passes, while the GPU tests validate generated-kernel correctness against unfused references and report execution time.

Reproducing all absolute performance numbers and comparisons in the paper requires a 4- or 8-GPU Hopper server with NVLink/NVSwitch connectivity and the external baseline implementations. Several external baselines are not vendored in the artifact; the repository documents this limitation and points to their upstream repositories.

Contents

The repository is organized as follows:

- `syncopate/communication/`: communication-plan descriptors, signal planning, code generation, and runtime support;
- `syncopate/computation/`: GEMM and attention kernel templates and the source-to-source transformation pass;
- `syncopate/interface/`: lowering from communication plans to per-rank schedules and tile schedules;
- `tests/`: CPU-only compiler tests and end-to-end multi-GPU correctness/performance tests;
- `experiments/`: figure-oriented experiment entry points for the operator, attention, compiler-integration, and ablation results; and
- `utils/`: the distributed launcher, communication microbenchmarks, and helper scripts.

Hosting

The artifact is publicly available under the MIT license at `https://github.com/tie-pilot-qxw/syncopate`. The version evaluated for OSDI '26 Artifact Evaluation is commit `ce2b13e496eb6629ab42aac2aec1d9e31e084ba6`. Evaluators and readers should check out this commit to obtain the fixed artifact version.

Requirements

The CPU-only validation requires Linux, Python 3.10 or newer, PyTorch, and `pytest`; it typically completes in under five minutes. Full generated-kernel and performance experiments require Linux, Docker with the NVIDIA Container Toolkit, and a single node with at least four NVLink/NVSwitch-connected NVIDIA Hopper-class GPUs; some experiments require eight GPUs.

Teaching The Old Dog New Tricks: Building Efficient Data Pipelines for Large-Scale LLM Pre-training (Operational Systems)

Luofan Chen[1,2*†] Chenhan Wang[1,2*†] Weidong Zhang[2‡] Jinxin Chi[2] Hequan Zhang[2] Zanbo Wang[2]
Chenyuan Wang[2] Lishu Luo[2] Sijin Wu[2] Junqi Hu[2] Jun Wang[2] Cheng Chen[2]
Lixin Huang[3] Liyang Zhao[3] Yong Tian[3] Jun Guo[3] Youhui Bai[1‡]
Wencong Xiao[2] Kang Chen[4] Cheng Li[1,5]

[1]*University of Science and Technology of China* [2]*ByteDance Seed*
[3]*ByteDance* [4]*Tsinghua University*
[5]*Institute of Artificial Intelligence, Hefei Comprehensive National Science Center*

Abstract

Data pipelines play a critical role in the performance of large-scale pre-training jobs running on thousands of GPUs. In this work, we present a comprehensive quantitative analysis of data access patterns from production workloads and reveal three previously underreported bottlenecks. First, cross-datacenter (cross-DC) traffic emerges as a major source of latency when evaluating in-training models using remote checkpoints. Second, checkpoint loading during startup phases frequently suffers from I/O contention that delays job initialization. Third, data transformation during loading becomes a significant and CPU-intensive bottleneck for multimodal models. Guided by these findings, we introduce three optimizations: global-namespace-based predictive checkpoint replication, proactive hot-file replication, and offloading data transformation to storage-tier CPU resources. Crucially, we demonstrate that these optimizations are not system-specific but address fundamental architectural mismatches in the LLM era. They are broadly applicable to both legacy and modern storage systems, offering a high-return path to upgrade infrastructure with minimal engineering intrusion. Together, these techniques reduce wasted GPU hours per evaluation from 16,800 to 4,000, shorten checkpoint loading time at each training start by 40.8%, and reduce training stalls caused by data loading by 63.2%.

1 Introduction

The training of foundation models has shifted the center of gravity in systems infrastructure [1, 7, 13, 18, 25, 56]. Training models with over 100 billion parameters now requires the coordination of thousands of GPUs across massive clusters for extended periods. The systems community has aggressively optimized compute scheduling [15, 16, 24, 34, 40, 46, 59, 60] and network collectives [10, 20, 41, 55] to maximize Model FLOPs Utilization (MFU) [6]. However, the data pipelines responsible for loading, transforming, and delivering exabytes of training data have received significantly less scrutiny.

This oversight is critical because the data pipeline is no longer a passive utility. It is a primary performance determinant. In this paper, we present a comprehensive analysis of data access patterns from production pre-training workloads. Our environment manages tasks ranging from billion-parameter text models to trillion-parameter multimodal models. We rely on the Hadoop Distributed File System (HDFS) [47] as our storage backbone due to its cost-effectiveness and capacity to hold exabytes of data.

We analyze 30 thousand training job traces collected over a 90-day period to characterize the interaction between modern pre-training jobs and traditional storage architectures. Our analysis reveals three distinct bottlenecks that arise when general-purpose storage systems confront the extreme scale of large model training.

First, we identify a Cross-DC Latency Trap. To ensure model quality, production training runs rely on *companion evaluation*, an out-of-band validation pipeline that periodically loads recent checkpoints and runs benchmark suites in parallel with training. Because evaluation jobs have different scheduling and hardware requirements from training, they often run in remote datacenters. We find that fetching checkpoint shards across the Wide Area Network is bound by the Round-Trip Time of thousands of small tensor reads rather than available bandwidth. This latency dominance results in severe delays and wastes millions of GPU hours.

Second, we observe an Initialization I/O Storm. Large-scale training involves frequent restarts due to debugging or failures [4, 51, 57]. Job initialization triggers a synchronous surge of read operations where thousands of workers contend for a small set of hot files, such as global metadata and shared embedding parameters. This contention saturates storage nodes and creates tail latencies that block the entire cluster.

Third, we encounter a Transformation Wall in multimodal training. Unlike text-only models, multimodal training involves CPU-intensive transformations like video and image

*Equal contribution.
†Work done during internships at ByteDance Seed.
‡Corresponding authors.

decoding. We show that host CPUs on training nodes cannot keep pace with GPU consumption. This shifts the bottleneck from storage I/O to local compute and causes significant pipeline stalls.

We argue that these bottlenecks persist because storage systems operate *reactively* while training workloads are inherently deterministic. To address this, we introduce three software-defined optimizations that expose application-level patterns to the storage layer.

- **Predictive Checkpoint Replication.** We leverage the regular schedule of evaluation jobs to replicate checkpoints and prioritize urgent feedback signals. This reduces companion evaluation I/O time by **76.1%**.
- **Proactive Hotspot Prediction.** We implement a cooperative mechanism where the training framework signals hot files to the storage system before execution. The storage layer proactively replicates these files to diffuse the I/O storm. This reduces checkpoint loading time by **40.8%**.
- **Storage-Side Transformation Offloading.** We utilize underutilized CPUs on storage nodes to perform data processing. This Just-in-Time mechanism pipelines transformation with training and reduces data loading stalls by **63.2%**.

Our work demonstrates that legacy Big Data systems like HDFS can meet the rigorous demands of exabyte-scale large model training without requiring a complete architectural replacement. Our contributions are as follows:

1. We characterize pre-training data access patterns using production traces and identify three key bottlenecks that have long been overlooked: cross-DC latency traps for companion evaluation, synchronous initialization I/O storms, and the multimodal transformation wall.
2. We propose software-defined optimizations leveraging workload determinism, including predictive checkpoint replication to mask latency, proactive hot-file replication to diffuse contention, and storage-side offloading for CPU-intensive transformations. These optimizations substantially reduce wasted training time and improve overall efficiency.
3. We demonstrate that these optimizations serve as a generalizable framework for addressing architectural mismatches in the LLM era. By bridging the gap between infrastructure and training, we provide a practical blueprint for evolving general storage to meet LLM demands. We validate these findings in a production environment and plan to release the anonymized traces to support future research.

2 Background and Motivation

2.1 Pre-training Data Pipeline

We present the data pipeline architecture that supports pre-training jobs on thousands to tens of thousands of GPUs in our production environment. We choose HDFS [47] as our storage infrastructure, because it was the existing, cost-effective infrastructure capable of holding exabytes of data. As shown in Figure 1, the pipeline consists of two primary components: a storage tier and a dataloader.

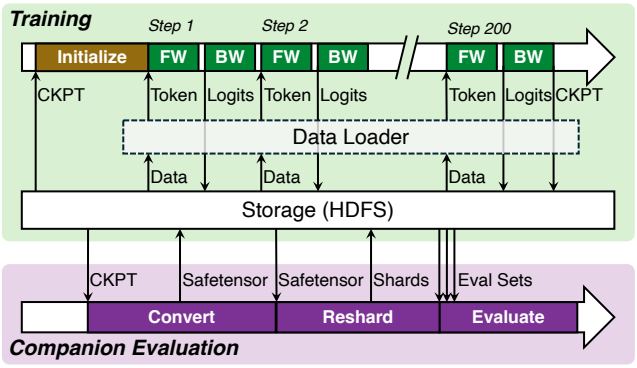


Figure 1: Data Pipeline in Pre-training

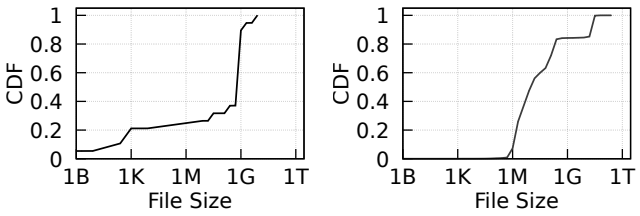
The storage tier persists all artifacts required during pre-training, including training datasets, model checkpoints, and logits. Training datasets encompass heterogeneous modalities such as text, images, and videos. Checkpoints contain metadata together with sharded tensors for model parameters and optimizer states [21] and serve as the authoritative record of training progress. Logits capture the raw, unnormalized prediction vectors emitted after each forward pass, which are typically not used during training but are essential for debugging and monitoring model behavior.

The dataloader sits between storage and the training framework. It transforms raw data into the formats expected by the model and coordinates data movement through CPU memory onto GPUs. The transformation operations include computationally intensive steps such as video and image decoding, spatial resizing for vision inputs, tokenization and sequence packing for text data [30, 33]. The dataloader thus acts as both a transformation engine and a staging mechanism that prepares and pipelines data efficiently to the accelerator stack.

Pre-training jobs interact with the data pipeline through two I/O paths. They issue direct reads and writes to the storage tier when initializing from checkpoints, persisting checkpoints, and emitting logits. In parallel, they obtain training data through the dataloader, which performs storage reads and executes sample transformation. Together, these interactions define the steady state and bursty I/O patterns imposed on the storage subsystem.

We characterize pre-training execution as consisting of three phases: **initialization**, **iterative training**, and **companion evaluation**, each with different data access behaviors.

During the **initialization phase**, the training framework issues blocking operations on the critical path. Specifically, it loads the full model checkpoint, including parameters and optimizer states, from the storage tier. These tensors are distributed or replicated across thousands of *ranks*, i.e., the



(a) T-L Dataset (b) MM-L Dataset

Figure 2: The file size distribution in datasets used in T-L and MM-L traces.

worker processes participating in distributed training that typically own one GPU each, according to the adopted parallelization strategy [40, 46, 59].

After initialization, the job enters the **iterative training phase**, which executes a sequence of *training steps*. We define a step as the processing of a full batch of data, culminating in a single model parameter update after the necessary forward and backward passes. During this phase, storage access proceeds largely asynchronously with GPU computation. The dataloader continuously prefetches and transforms data for future steps, while the training framework writes logits after every forward pass and periodically persists checkpoints to storage. These activities generate sustained read and write traffic throughout the training run.

The **companion evaluation phase** runs periodically in parallel with iterative training on separate evaluation clusters, as evaluation usually imposes different scheduling and resource needs compared to training. Since training metrics such as loss are often insufficient to detect issues such as quality degradation or model collapse, companion evaluation provides an external measurement of progress by assessing the model on a recent checkpoint and on data that is distinct from the training set (i.e., real-world benchmarks [14, 26, 27]). Each evaluation cycle loads the latest checkpoint from storage, merges the sharded tensors into unified formats, and then loads both the merged checkpoint and benchmark suites, before executing inference and writing the results back to storage. Although logically separate from training, the results of companion evaluation can feed back to the training control plane and trigger a rollback.

2.2 Data in Pre-training

2.2.1 Traces

To investigate I/O access patterns, we select a set of five representative training tasks from a pool of 30 thousand traces collected over a 90-day period. As detailed in Table 1, these tasks exemplify state-of-the-art production workloads, accounting for 70% of total GPU hours while capturing diversity in:

- **Modality:** We cover mainstream training domains spanning text-only and multimodal workloads.
- **Scale:** The tasks range from small (billions-parameter) to large (trillion-parameter) models, utilizing training cluster sizes between approximately 4K and 20K GPUs.
- **Training strategy:** We incorporate diverse distributed strategies, including FSDP [59], FSDP2 [37], and Megatron [46], which constitute the dominant paradigms in our training environment.

To characterize these workloads, we employ a multi-layered tracing approach that spans both the training framework and the storage layer. At the storage layer, we collect per-operation records from HDFS client and server endpoints, including the access path, operation type (e.g., `open`, `read`, `pread`, `write`, and `close`), request size, start and end timestamps, and observed latency. At the training layer, we use PyTorch Profiler [38] and manual instrumentation to record the start and end times of major phases, including checkpoint loading and saving, dataloader fetch and transformation, forward and backward computation, logits saving, and companion-evaluation merge and resharding. We then join these records using timestamps to distinguish raw storage service time from framework-visible end-to-end latency, which is necessary for interpreting the data loading measurements in Table 2. We will release the traces once anonymized.

2.2.2 Data Types

Based on our sampled traces, we classify the data flowing through the pipeline into three primary types: datasets, checkpoints, and logits. Table 2 quantifies the distinct I/O patterns, including average I/O sizes per training host, average I/O latency, and I/O frequency, exhibited by each type across our production workloads.

Dataset: During each training step, the dataloader retrieves part of the dataset from storage, transforms each sample into model-ready tensors through operations like decoding, cropping and tokenization, and feeds the result to the GPU for computation. The modality of the model significantly influences the characteristics of the dataset. For instance, in the text-based T-L training task, the dataset occupies approximately 20 TB across 20K files. In contrast, the multimodal MM-L task employs a vastly larger dataset containing diverse file types, with a total size of about 6 PB and a file count exceeding 3 million. Figure 2 illustrates the file size distribution for these datasets. Table 2 reports the end-to-end data loading latency, which includes both the retrieval and the transformation steps; we further break this down in Section 5.

Checkpoint data: A training checkpoint consists of model parameters, optimizer states, dataloader states, and miscellaneous metadata. At regular intervals (e.g., every 100 or 500 steps), this checkpoint data is persisted to the underlying storage system to ensure fault tolerance [29, 50]. We detail the checkpoint sizes for the sampled traces in Table 1. The trace-level view in Table 2 further shows per-host checkpoint read

Table 1: Model training strategies, types, and checkpoint sizes of the representative trials sampled for this study. Each trial runs on a production cluster with thousands of GPUs.

Trace	Strategy	Type	Checkpoint Size
Text Small (T-S)	FSDP2	Text	≈1 - 5 TB
Text Large (T-L)	Megatron	Text	≈5 - 10 TB
MultiModal Small (MM-S)	FSDP	Multimodal	≈1 - 5 TB
Multimodal Large (MM-L)	FSDP2	Multimodal	≈1 - 5 TB
Multimodal Omni (MM-O)	FSDP	Multimodal	≈5 - 10 TB

Table 2: Average I/O size, latency, and interval statistics. Values represent the average size per training host. Each metric reports the end-to-end data pipeline latency observed by the training framework. The symbol '–' indicates data is unavailable because the feature was disabled during model training.

	Read checkpoint		Data loading			Save checkpoint			Save logits			Step
Trace	**Size (GB)**	**Lat (s)**	**Size (GB)**	**Lat (s)**	**Interval (step)**	**Size (GB)**	**Lat (s)**	**Interval (step)**	**Size (GB)**	**Lat (s)**	**Interval (step)**	**Time (s)**
T-S	688	137.58	0.01	5.77	1	11.21	56.27	500	–	–	–	14.03
T-L	606.57	109.79	0.009	5.51	1	7.03	25.74	200	1.99	2.96	1	8.27
MM-S	351.27	300.33*	0.05	8.64	1	1.40	106.83	200	–	–	–	19.20
MM-L	373.81	139.25	0.02	5.29	1	5.84	68.31	200	0.18	2.34	1	18.17
MM-O	1133.63	106.83	0.01	6.16	1	13.93	128.605	200	–	–	–	14.39

* Denotes that this data is obtained before predictive hotspot prediction (Section 4)

and save sizes, which differ substantially due to deduplicated writes; Section 4 explains the source of this asymmetry in detail.

Logits data: Logits represent the raw, unnormalized prediction vectors output by the final layer on each rank. Depending on the model configuration, this results in approximately 30 MB to 300 MB of data. Unlike checkpoints, which are required for fault tolerance, logits are mainly persisted for debugging and monitoring. Developers optionally enable logits saving during phases such as early training, configuration updates, or divergence diagnosis, while stable steady-state runs often disable it to conserve I/O bandwidth. When enabled, logits are generated after every forward pass and persisted at every step.

2.3 Emerging Storage Challenges

During pre-training, distinct data types are accessed at specific phases of the lifecycle, often generating I/O operations that consume a significant portion of the total timeline. In this paper, we characterize the unique I/O bottlenecks inherent to each distinct phase and present our production mechanisms to address them.

1. Cross-DC I/O in companion evaluation (§3). Companion evaluation is critical for validating pre-training models, but is bottlenecked by the need to fetch checkpoints from remote (cross-DC) training clusters due to scheduling and hardware constraints. This creates significant I/O challenges: a "small-IO latency trap" where fetching thousands of small tensors is bound by WAN RTT rather than bandwidth, and severe bandwidth contention from concurrent background tasks. To mitigate these issues, we introduce Predictive Checkpoint Replication, which leverages the regular evaluation interval to pipeline and batch checkpoint replication in advance, and Signal-Driven Prioritization, which allocates bandwidth to urgent evaluations triggered by training anomalies.

2. The initialization I/O storm (§4). The initialization phase is bottlenecked by an "I/O storm" where synchronous checkpoint loading triggers massive data contention on a small set of hot files (e.g., global metadata and replicated parameters). This contention saturates storage nodes, creating straggler reads that block the entire cluster and waste significant GPU cycles. To resolve this, we introduce Proactive Hotspot Prediction, a cooperative mechanism where the training framework explicitly signals deterministic access patterns to the storage layer. This allows the system to pre-replicate hot files before the job starts, diffusing contention and converting unpredictable tail latency into constant-time access.

3. The transformation wall (§5). For multimodal (e.g., MM-L) pre-training, data loading shifts from being I/O-bound to compute-bound due to the high cost of transforming video and image data (e.g., decoding, cropping). This "transformation wall" creates severe straggler issues where slow host-side processing forces GPUs to idle, wasting significant compute resources during the iterative training phase. Offline transformation is infeasible due to extreme storage amplification

($50\times \sim 100\times$) and inflexibility to hyperparameter changes. To resolve this, we introduce Storage-Side Transformation Offloading. By leveraging the predictable ordering provided by our deterministic dataloader and underutilized CPUs on storage nodes, we push compute-intensive transformations down to the storage tier. This "Just-in-Time" processing pipeline overlaps transformation with training and reduces both latency and stall time.

3 Cross-DC Companion Evaluation

Training large-scale foundation models with hundreds of billions of parameters is a costly and extended process that involves thousands of GPUs running for weeks or months. In this environment, detecting training issues early is essential. Although monitoring training loss is common practice [48, 56], our production experience shows that in-loop signals such as loss and gradient norms are not reliable indicators of model health. A model may show stable training loss while already exhibiting forgetting or collapse on downstream tasks. This gap motivates the use of *companion evaluation*, a continuous, out-of-band validation pipeline that assesses model performance in parallel with training. In our deployment, companion evaluation serves as a critical safeguard that helps prevent the training system from progressing on an already diverging model state.

In our production environment, companion evaluations execute on separate clusters that are often located in different datacenters. This design is driven by two primary constraints. First, pre-training jobs demand strict synchronous execution across thousands of GPUs, relying on gang scheduling [12] to maintain lockstep progress; consequently, inserting evaluation tasks into the same cluster would fragment scheduling and degrade training throughput. On the other hand, while pre-training necessitates the latest accelerator generation with high-speed interconnects to minimize the step time, evaluation workloads are throughput-oriented and can run effectively on older, more cost-efficient hardware. These evaluation clusters are typically deployed wherever power and space are available, often resulting in physical separation from training resources.

3.1 Companion Evaluation Workflow

The bottom part of Figure 1 illustrates the end-to-end workflow of companion evaluation. Each evaluation run transforms a distributed training checkpoint [8] into an inference-ready model and then executes benchmark workloads. The workflow proceeds in three stages.

First, *checkpoint merging* converts distributed checkpoint shards into inference-ready modality submodels. During training, the training framework writes checkpoints as many distributed shards optimized for parallel writes on large training clusters [50]. For evaluation, however, these shards must be converted into a standard inference format such as `safetensors` [42]. The evaluation cluster therefore retrieves the relevant parameter shards from remote storage and performs merging separately for each modality: it reconstructs the complete parameter tensors, discards optimizer states and other tensors that are unnecessary for inference, and writes each merged modality submodel to local storage as `safetensors` files.

Table 3: Data read/write sizes and time consumption for the MM-L model. The checkpoint merging phase dominates I/O time due to cross-DC traffic.

Stage	Read (GB)	Write (GB)	Total (Time)	I/O (Time)
Merging	**2631***	**1742**	**2h 03m**	**2h 03m**
Reshard	**872**	**1435**	**18m**	**18m**
Evaluation	**8954**	0	**1h 55m**	**4m**

* Denotes Cross-DC Traffic

Next, *model resharding* reorganizes the merged checkpoint to match the device topology and parallelism configuration of the evaluation cluster. This reshaping ensures that inference can be executed efficiently across the available devices.

Finally, the *model evaluation* stage uses a client-server architecture. The server component loads the resharded model and initializes the HTTP serving endpoint, while the client simultaneously executes inference across benchmark datasets to quantitatively assess model quality. The system writes the resulting metrics to storage, and the training control system subsequently ingests them.

This workflow requires substantial data movement and local processing, and its efficiency is directly shaped by the network and storage behavior observed during checkpoint merging. Table 3 shows the data sizes and time distribution of these stages.

3.2 Checkpoint Merging Bottlenecks

Our study spans a 30-day observation window covering 19 pre-training tasks across models ranging from small to large parameter counts. During this period, the system executed 3,589 companion evaluations, which contributed to operational efficiency by identifying 156 critical model regressions.

The value of this mechanism depends on timely feedback. When evaluation results are delayed, training continues to advance on a divergent model state, which both wastes training GPU time and makes recovery harder.

The root of this delay lies in data I/O. For a representative multimodal large model (MM-L), Table 3 shows that a single evaluation cycle can exceed four hours, with data I/O accounting for approximately 56.6% of the end-to-end time. The cost of this I/O bottleneck is substantial: across the 156

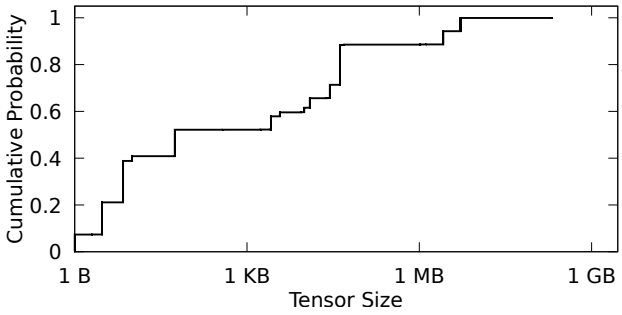


Figure 3: CDF of tensor sizes in a typical checkpoint. The prevalence of small tensors exacerbates the latency penalty of cross-DC reads.

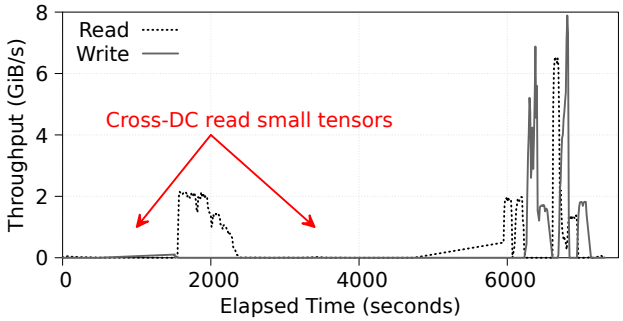


Figure 4: Throughput during Checkpoint Read. The performance is limited by the high RTT of cross-DC links when fetching small tensors.

regressions in our dataset, I/O-induced delays wasted approximately 2.6 million GPU hours, offsetting nearly half of the 5.5 million training GPU hours that the evaluation system would otherwise have saved.

Of this long data I/O time, we further find that 84.8% is consumed by the checkpoint merging phase, which must read approximately 2.6 TB of data over the Wide Area Network. Unlike later stages that operate entirely on local storage, checkpoint merging requires retrieving tensors across datacenters, which makes it the principal source of evaluation delay. Specifically, we identify two primary causes of the delay:

Small-I/O latency trap. Transformer-based models consist of thousands of disjoint tensors, particularly small *LayerNorm* parameters and *Biases*. As illustrated in Figure 3, over 60% of the tensors in the MM-L checkpoint are smaller than 16 KB. Because the per-modality merge described in Section 3.1 retrieves only the parameter tensors needed for each submodel, the resulting reads are issued at tensor granularity rather than as bulk transfers. Even when the merge server fetches tensors concurrently, each request still requires several cross-DC WAN round trips. This fragmentation interacts poorly with the underlying distributed file system, which is optimized for coarse-grained 128-MB blocks. This mismatch produces two effects:

- **Read amplification.** Fetching a 4 KB tensor often requires reading a significantly larger block on HDFS data nodes, incurring about 1.5× read amplification.
- **RTT-bound throughput.** On a high-latency WAN link (typical 100 ms RTT), the throughput for these thousands of small tensor reads becomes bound by latency rather than bandwidth. This manifests as the "sawtooth" throughput pattern in Figure 4, where effective utilization drops to a fraction of the available link capacity (i.e., 60 Gbps).

Bandwidth contention. Even for large tensors, cross-DC bandwidth remains a scarce resource contended for by hundreds of concurrent tasks (208 on average), including data migration, log aggregation, and parallel evaluations. Figure 5 illustrates this contention; the average available bandwidth per task fluctuates significantly and often degrades to less than 1 GB/s. A naive "First-Come, First-Served" policy fails in this context because it treats latency-critical evaluations indistinguishably from background archival jobs.

3.3 Training-Aware I/O Optimization

To address the cross-DC I/O bottlenecks identified earlier, we apply two training-aware optimizations that require no dedicated network hardware.

Predictive checkpoint replication. Companion evaluations follow a predictable interval, typically every one thousand training steps. This regularity allows us to initiate pipelined checkpoint replication as soon as the training system begins saving a checkpoint at the scheduled interval. Rather than letting the merge script issue thousands of tensor-granularity WAN reads, the storage system batches the required checkpoint shards into large contiguous transfers and caches them in the evaluation cluster's storage tier. A lightweight server-side namespace service, NNProxy, then directs evaluation jobs to read from these local cached shards when they are available. This converts many Wide Area Network reads into local reads and removes both cross-DC RTT and small-read fragmentation from most scheduled evaluations.

Signal-driven prioritization. Not all evaluations have equal urgency. While periodic evaluations follow a fixed schedule, others are triggered by training anomalies such as loss spikes or abnormal gradient behavior. These events require faster feedback. We introduce a simple priority signal that combines model scale, task importance, and anomaly severity. The storage scheduler uses this signal to preempt background traffic and allocate network bandwidth to urgent evaluations, reducing delay when fast diagnosis is needed.

3.4 Evaluation

By applying the above optimizations, we convert previously unmanaged I/O contention into predictable resource alloca-

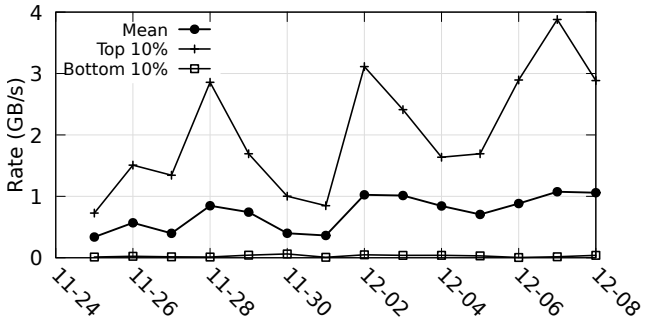


Figure 5: The average cross-DC data transmission rate in our production environment. The bandwidth is highly contended and bursty.

tion. These training-aware mechanisms reduce average checkpoint merging latency by 76.1%, with improvements reaching 89.3% for the T-S model and 70.8% for the MM-L model. They also cut I/O-induced compute waste per regression from 16,800 GPU hours to 4,000 GPU hours. In total, these optimizations recover nearly 2 million GPU hours across our observation window.

3.5 Future Directions

To further improve performance across geographically distributed clusters, we plan to implement vertical co-design between training frameworks and storage layers. Two directions are especially promising.

Training informed storage control. Training frameworks should expose signals such as evaluation interval and convergence indicators to the storage control plane. These signals would allow the storage system to anticipate latency-sensitive operations, allocate resources accordingly, and avoid interference from background traffic.

I/O-Aware checkpoint formats. Standard distributed checkpoint layouts are optimized almost exclusively for high-throughput, parallel writes during training, which inherently introduces the structural fragmentation that plagues selective cross-DC reads. A promising direction is the design of layout-agnostic or multi-layout checkpoint formats. By restructuring how tensors are grouped, such as packing fragmented parameters (e.g., LayerNorm layers and biases) into contiguous, storage-block-aligned extents while segregating volatile optimizer states, the framework could natively support zero-overhead, selective WAN streaming without requiring extensive server-side merging.

4 Initialization I/O Storm

Large-scale model training is characterized not by continuous execution, but by frequent interruptions and restarts; these arise from hardware failures, automated system maintenance, and, most notably, intensive debugging cycles during model development [18, 51]. Initialization is a fully synchronous barrier because training cannot resume until every rank has reloaded its share of the latest checkpoint from storage. When thousands of ranks restart in lockstep, they issue concurrent reads against a small set of hot files at the same instant, turning the storage tier into the bottleneck of the start phase.

Table 4: Breakdown of training startup latency.

Stage	Operation	Time (s)
Job Init.	Cluster Startup	23
	Dist. Context Init.	130
Model Const.	Model Construction	118
Ckpt. Recovery	Coordination	76
	Downloading from HDFS	**300**
	Parameter Loading	104
Total		**751**

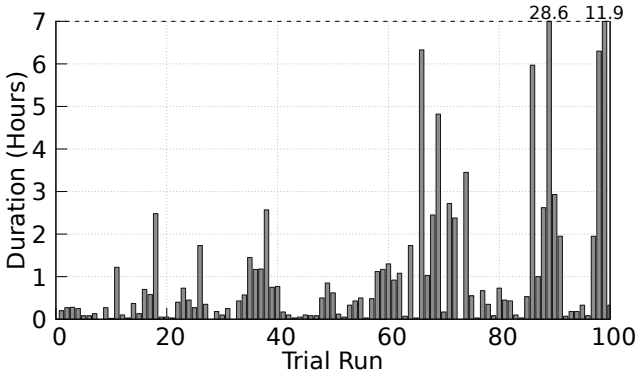
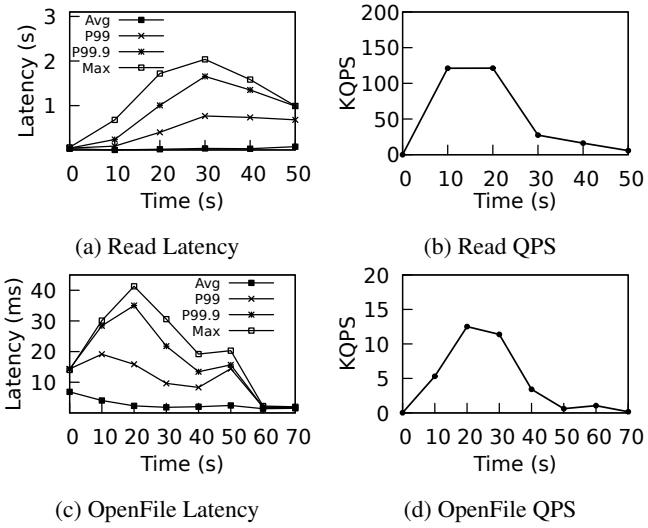


Figure 6: Duration of continuous training sessions between restarts in the MM-S trace. Short bars indicate frequent restarts, while long bars indicate longer uninterrupted training periods.

4.1 Synchronized Loading Bottleneck

In our production environment, we observe that job initialization is a significant overhead. The breakdown of training startup latency in the MM-S trace, as shown in Table 4, reveals that blocking I/O operations during file download are the primary bottleneck, consuming 39.95% of the total startup duration.

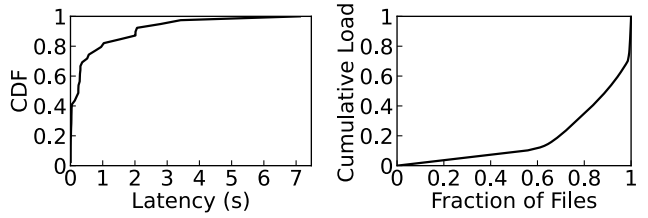
The operational impact of this latency is magnified by the bursty nature of job restarts. Figure 6 depicts the restart distribution from our MM-S trace. Each bar corresponds to one restart interval, with the y-axis showing the elapsed training time before the next restart. We observe distinct clusters of activity where developers repeatedly restart jobs for code updates or hyperparameter tuning. For instance, within a single



(a) Read Latency (b) Read QPS

(c) OpenFile Latency (d) OpenFile QPS

Figure 7: Latency and throughput (QPS) for Read and OpenFile operations under the default 3-replica policy.



(a) Access latency CDF. (b) Cumulative load distribution.

Figure 8: Characterization of I/O contention. (a) Latency distribution for file downloads. (b) Load distribution relative to the fraction of files.

111-minute debugging window, a job underwent 4 consecutive restarts. With an average downloading cost of 5 minutes per run, this I/O overhead accounts for 18.02% of the total wall-clock time, severely throttling developer velocity and wasting costly GPU cycles. Extrapolating to our cluster-wide operation, the startup delay would result in a loss of more than one million GPU hours annually.

4.2 The Anatomy of an I/O Storm

To pinpoint the root cause of slow checkpoint loading, we conduct a controlled experiment scaling to 2,048 GPUs. We instrument the storage client to decompose the latency of every I/O request.

Contrary to common assumptions that metadata operations (e.g., `open`, `getattr`) are the bottleneck in distributed file systems [23], our analysis reveals that *data contention* is the primary culprit. As illustrated in Figure 7, `open` latencies remain low and stable. However, `read` operations exhibit extreme tail latencies that correlate perfectly with QPS spikes.

Figure 8a plots the CDF of download latency. The distribution exhibits a pronounced long tail, indicating that while the majority of downloads complete quickly, a small fraction of tasks suffer from significantly high latency. Our breakdown shows that these few straggler reads dominate the completion time, accounting for 67.97% of the total wait.

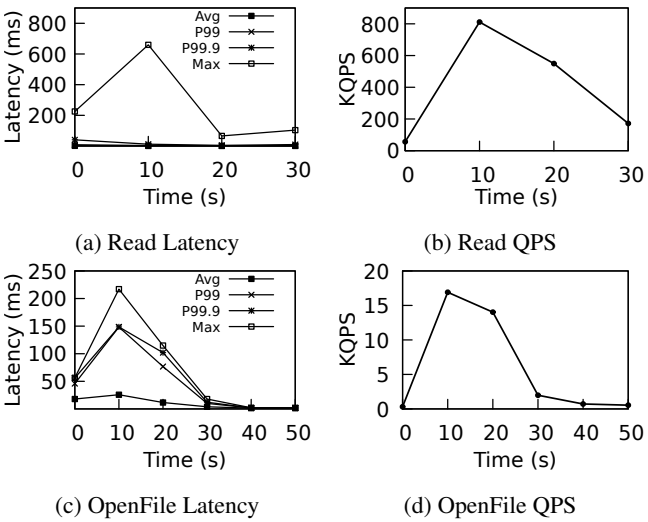
By calculating the peak QPS for each file within 0.5-second windows, we observe a heavy-tailed distribution in Figure 8b. The cumulative load represents the contribution to the aggregate peak QPS (i.e., the sum of all individual file peaks). Specifically, the top 5% of files account for 38.8% of this total pressure. These hotspots are typically caused by two sources: global metadata files (e.g., `.metadata` or `common_states`) accessed by all ranks, and more significantly, access skew resulting from redundancy elimination.

This skew is a byproduct of current checkpointing systems [8, 35, 50, 52], which prioritize write throughput and storage efficiency by employing *parallel saving* coupled with *redundancy elimination*. While effective for writing, this approach causes loading asymmetry. We illustrate this issue using the Embeddings and MoE layers. For Embeddings (which are replicated parameters), the system persists only a single copy to minimize overhead; consequently, restoring this layer triggers a one-to-many access storm where hundreds of ranks concurrently contend for a solitary file. Under the standard 3-replica policy, the limited replicas are overwhelmed by such concurrent requests, causing latencies to increase sharply. In contrast, MoE experts (which are sharded parameters) are saved as distinct files corresponding to their parallel groups. This allows subsets of ranks to access different files concurrently, effectively distributing the load and avoiding the single-point bottlenecks observed in the embedding layer.

4.3 Proactive Hotspot Prediction

Facing this challenge, traditional storage systems employ reactive replication [3, 54]: they detect heat via runtime monitoring and trigger replication when thresholds are breached. In the context of training startup, this approach is fundamentally flawed for two reasons. First, the contention window is transient (often lasting only tens of seconds). By the time the system detects the hotspot and creates new replicas, the loading phase is already over. Second, initiating replication traffic during the I/O storm competes for the already saturated network bandwidth, exacerbating the bottleneck rather than relieving it. To address these limitations, mitigation must be proactive, leveraging the training framework's information to prepare the storage layer before the storm begins.

We propose a cooperative mechanism in which the training framework explicitly informs the storage system about upcoming access patterns via a `SetReplicationHints` interface. Since training configurations (parallelism strategy, world size) are known at job submission, the set of hot files is deterministic. The framework identifies two categories of



(a) Read Latency (b) Read QPS

(c) OpenFile Latency (d) OpenFile QPS

Figure 9: Latency and throughput (QPS) for Read and OpenFile operations under the default 128-replica policy.

data: global metadata, where the target replication factor is set proportional to the job's world size N, and replicated tensors, where access demands are aggregated based on tensor-to-rank mappings to determine the expected concurrency degree ω_f for each file f. Upon receiving these hints, typically immediately after a checkpoint is written or before a job is scheduled, the storage system proactively replicates the identified hot files using a heuristic policy:

$$R_{target}(f) = \lceil \frac{\omega_f}{C_{load}} \rceil \quad (1)$$

where C_{load} represents the safe concurrent load capacity of a single replica. To prevent storage bloat, these ephemeral replicas are tagged with a Time-To-Live (TTL) and automatically reclaimed after the startup phase concludes.

4.4 Evaluation and Discussion

We evaluate proactive replica expansion on the 2,048-GPU training job. Based on the job scale and the predicted checkpoint access pattern, the policy increases the replication factor of hot metadata and replicated-parameter files to 128 before checkpoint recovery begins.

As shown in Figure 9, replica expansion improves checkpoint loading in two ways. First, the peak aggregate Read QPS increases, indicating that compute nodes spend less time blocked on storage and can issue requests more aggressively. Second, and more importantly, the average read latency decreases because the hottest files are no longer served by only a small number of replicas. By eliminating straggler reads, the total checkpoint loading time decreases from 38.48 s to 22.78 s, a 40.8% improvement.

While the absolute saving in this controlled 2,048-GPU experiment is on the order of seconds, the operational impact grows with training scale. Without intervention, synchronized checkpoint recovery creates increasingly severe contention as more ranks restart together. By preparing replicas before the recovery phase, proactive replica expansion turns an unpredictable, scale-dependent storage bottleneck into a more stable loading process. Based on these results, we have codified replication guidelines for our production environment: 64 replicas for 10k-GPU clusters and 128 replicas for 20k-GPU clusters.

The benefit of replica expansion depends on how the training parallelism strategy materializes model state during checkpointing. In the most favorable case, fully sharded data parallelism already spreads parameter shards across ranks and files. The remaining imbalance usually comes from a small set of shared structures, such as checkpoint metadata, common states, or other non-sharded tensors. Expanding only these structures is often enough to make the recovery load well balanced across storage replicas, with little storage overhead. The common production case is less ideal but still highly targeted: each rank typically saves a small number of tensors, often a single tensor, so most files have clear ownership while replicated tensors and global states still create a few hot files. Replica expansion is effective here because the system only needs to expand this small hot subset instead of increasing the replication factor of the entire checkpoint. A less favorable case arises when a replicated tensor is packed together with much larger sharded parameter shards in the same checkpoint file. Since the replication decision is made at file granularity, expanding the file for the shared tensor also replicates the unrelated sharded data.

4.5 Future Directions

To further improve checkpoint recovery at larger training scales, future systems could better align checkpoint file layout and recovery-time fanout with replica expansion.

Deduplication-aware checkpoint packing. A natural extension is to consider recovery-time replication when partitioning tensor saving tasks. Tensors that are aggressively deduplicated, and therefore likely to be read by many ranks during recovery, should be placed in dedicated files or co-packed with tensors that require similar replica expansion. This layout keeps file-level replica decisions aligned with tensor-level demand: expanding a hot file increases capacity for shared reads without also copying large unrelated shards.

Communication-assisted checkpoint loading. The loading plan of checkpoint recovery already indicates which ranks share the same checkpoint shards, providing the information needed to organize communication-assisted fanout. Depending on deployment scale, rank locality, and framework-integration constraints, future systems could serve a hot shard through additional storage replicas, a scoped broadcast path, or a hybrid of both.

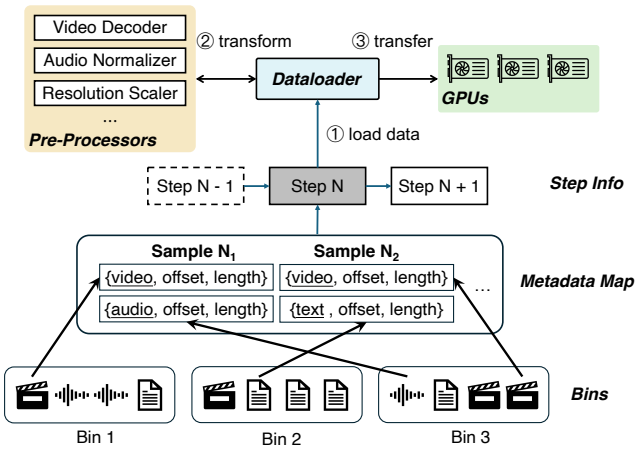


Figure 10: Structure of the on-disk format and the data loading pipeline for the deterministic dataloader.

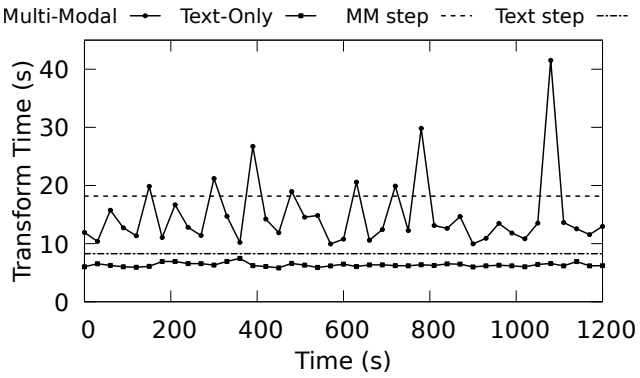


Figure 11: Maximum dataloader latency: Text (T-L) vs. Multimodal (MM-L). MM-L suffers from extreme tail latency due to heterogeneous transformation costs.

5 MM-L Transformation Wall

In conventional text-only model training, the data loading process is effectively hidden behind GPU computation. Data reads and lightweight tokenization finish well before each step starts, leaving the training loop compute-bound. However, as training moves from text-only models to multimodal models, the dataloader must serve not only text but also heterogeneous data such as audio, video, and images. As a result, data loading for multimodal training can no longer be fully overlapped with GPU execution, causing the input pipeline to exceed the compute time by a wide margin and become the dominant source of training stalls.

5.1 Deterministic Dataloader

Multimodal training in our environment uses a deterministic dataloader architecture. Unlike standard loaders such as the PyTorch DataLoader [39] or streaming dataloaders [32] that incur runtime overhead from dynamic shuffle computation and complex state management, our approach utilizes an offline-generated global execution plan. This guarantees strict deterministic ordering across thousands of workers and achieves global random shuffling.

Figure 10 illustrates the three data formats used to organize and serve raw training data. Under the hood, the system relies on *consolidated binary bins* where raw multimodal data including audio, video, and concatenated image and text is stored as large contiguous binary files in the global storage system. We choose a bin size of 20 GB to balance file management overhead against the cost of merging bins, providing precise byte range access without the read amplification common in columnar formats like Parquet [49]. To enable fast lookup, we employ *metadata maps* that serve as a logical-to-physical translation layer, converting sample identifiers into byte offsets and lengths within the binary bins. Our production datasets contain roughly ten thousand map files totaling about 0.7 TB. Finally, we utilize *per-step info files* to manage ordering. Since each training epoch consumes all samples in a shuffled order, we generate a file listing the exact sequence of sample identifiers to be read. This allows shuffling to be performed entirely by permuting index entries offline, achieving a new global order without touching or rewriting the underlying physical data.

During training, the dataloader performs the following sequence to prepare data for the next training step:

- **Metadata resolution.** The dataloader begins by reading the per-step info file to get the sample identifiers required for the upcoming batch. It then consults the metadata maps to translate these identifiers into byte offsets and lengths within the consolidated binary bins.
- **Data retrieval.** Using the resolved offsets, the dataloader issues `read` or `pread` system calls to fetch the corresponding byte ranges from the binary bins. This stage performs random access into the underlying files to retrieve only the specific samples needed for the current batch.
- **Sample transformation.** Once the raw bytes are retrieved, the dataloader applies modality-specific transformations. For instance, video samples will go through a series of operations such as video decoding, frame extraction, and cropping, followed by conversion to tensors, while image samples will be decoded, resized, and transformed into tensors. These operations are executed on the host CPU to avoid the large memory overhead of GPU-based transformation and to keep GPU resources focused on model computation. The resulting tensors are staged for transfer to the GPU at the start of the next step.

Table 5: Breakdown of Data Loading Latency in MM-L. Transformation dominates the pipeline, and high variance (Max vs. Avg) indicates severe straggler issues.

	Read (s)		**Process (s)**	**Other (s)**	**Total (s)**
Metric	Metadata	Data	Transform	Overhead	Time
Avg	0.007	0.013	5.05	0.279	5.35
Max	0.506	0.266	41.52	0.428	42.72

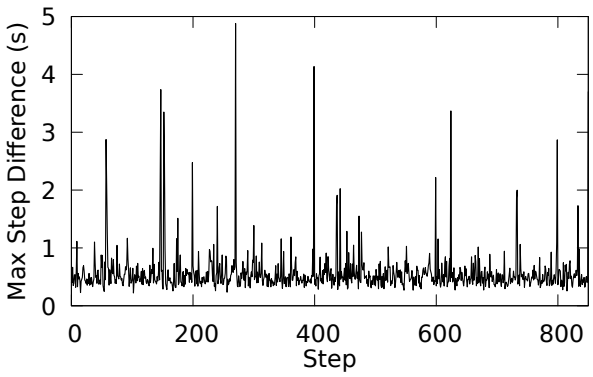


Figure 12: The time difference in training step time across different machines.

5.2 Transformation Stalls

Table 5 decomposes the data loading latency into four phases in the MM-L training trace: metadata resolution, data retrieval, transformation, and runtime overheads such as timing and threading. Transformation overwhelmingly dominates the critical path, accounting for 94.4% of the total data loading time (5.05 s of 5.35 s). I/O retrieval itself is negligible, averaging only 13.6 ms. This shows that MM-L workloads are compute-bound: limited CPU capacity, rather than storage bandwidth, drives up the time required for transformation.

The heterogeneous nature of multimodal data, mixing short text with high-resolution images and long video clips, introduces substantial variability in transformation cost. Figure 11 compares the maximum data loading latency across hosts for T-L and MM-L traces. While T-L workloads show tightly clustered latencies, MM-L workloads exhibit a pronounced long tail. For instance, the slowest host can take 42.72 s in our sampled period, where the average data loading time for that step is 5.35 s. This produces a 37.4 s gap in that single step.

This variability is driven by sample-length variance in multimodal datasets rather than by a transient storage issue. To preserve training coverage, MM-L batches include both lightweight samples and long video clips whose transformation cost scales with duration, codec complexity, and the number of sampled frames. As GPU step time continues to shrink, this host-side long tail becomes increasingly difficult to hide behind model computation. In our sampled phase, a host processing a 161.9 MB sample requires 41.5 seconds to finish the transformation, causing the time variation. In contrast, other hosts process an average of only 20 MB and complete the transformation in 5.05 seconds. In extreme cases, a machine assigned a 500 MB H.264 video of about 10 minutes requires over 2.18 minutes for decoding and frame extraction, even with `decord` using hardware-accelerated decoding and four concurrent decoding threads. Such stragglers force thousands of GPUs to idle, waiting for them to complete. Figure 12 illustrates the variance in training step duration. We find that for a single step, the execution time difference across hosts can reach up to 5 seconds, forcing other nodes to wait for the straggler. This temporal skew directly translates to a waste of over 10,000 GPU hours per day.

5.3 Storage-Side Transformation Offload

Addressing this bottleneck requires reducing the cost of the transformation phase. While performing offline transformation (i.e., decoding and transforming raw samples into ready-to-train tensors before training) seems like a straightforward solution, it is not viable at the production scale for two fundamental reasons. First, *storage amplification* makes storing decoded data prohibitive. High-resolution images and videos are typically archived in space-efficient formats such as JPEG [19] and H.265 [17]. Decoding them into float tensors causes extreme storage expansion: video footprints increase by more than 40×, while images exhibit a smaller but still substantial expansion of several tens of times. At the petabyte scale, the decoded representation would demand storage in the exabyte range, merely shifting the bottleneck from online CPU transformation to raw storage capacity. Second, offline transformation suffers from inherent *inflexibility*. The required transformations depend heavily on model hyperparameters such as crop size, frame count, and resolution, all of which vary across pre-training runs. Any alteration to the model architecture or data augmentation policy necessitates a complete regeneration of the pre-transformed dataset. This process consumes millions of GPU hours and significantly hampers model development velocity.

Instead, we exploit a resource that is already paid for: *storage nodes in high-throughput clusters often have underutilized CPU capacity*. While training hosts are saturated by model computation and media processing, storage nodes are typically bottlenecked by network or disk bandwidth, leaving significant compute capacity idle.

This opportunity arises from a fundamental I/O-compute mismatch in our cluster. We observe that while training nodes are Compute/Memory saturated, storage nodes are typically bounded by HDD seek latency or Network Interface Card (NIC) limits. Our telemetry indicates that storage node CPUs often sit at 20–30% utilization waiting for I/O interrupts, a resource gap we exploit to hide the format conversion cost.

We redesign the storage layer to act not just as a data repository, but as a *Disaggregated Pre-processing Engine*. This design is fundamentally enabled by the strict determinism of our deterministic dataloader (Section 5.1). Since the data consumption sequence is fixed in the global index, the storage system can accurately predict future data access patterns, allowing it to perform transformation using its idle cores. We implement a *Pushdown Transformation Engine* with three core mechanisms:

Schedule synchronization. Upon training initialization, the training framework shares the dataset identifier and step progress with the storage system. Storage nodes, having access to the per-step info files, autonomously resolve the sequence of upcoming data blocks. This eliminates the need for the training client to issue millions of data and metadata operation RPCs during the critical path.

Just-in-Time (JIT) transformation. Storage nodes maintain a *Consumer Queue* for the training job. They speculatively read raw binary data and execute the registered transformation graph such as decoding, random cropping, and normalization in the background. This pipelining ensures that while the GPU computes Step N, the storage layer is already preparing tensors for Step $N+1$. For video samples, the storage-side engine also applies frame sampling before data is sent back to the training cluster. This keeps the network payload bounded despite the temporary expansion caused by decoding and prevents format inflation from becoming a new tail-latency source.

Load-aware fallback. To prevent saturating the storage node, we implement dynamic backpressure to guarantee storage stability, which is a hard constraint in our shared production environment. If a storage node's CPU usage exceeds a safety threshold (e.g., 80%), it aborts the transformation and returns raw bytes to the training dataloader. The training client detects the data format (Tensor vs. Binary) and falls back to local processing. This hybrid approach allows us to absorb transformation spikes using the aggregate compute power of the storage fleet while protecting the storage control plane from overload.

5.4 Evaluation

This architecture yields significant improvements in our production environment, validating the effectiveness of moving compute to the storage tier. Specifically, by offloading heavy lifting to the storage pool, the P99 data loading latency drops by 85.7%, effectively mitigating the straggler effect. In terms of training efficiency, stall time due to transformation stragglers is reduced by 63.2%, translating to a relative 10.8% improvement in MFU. Finally, for resource utilization, we observe a 94% reduction in data loading CPU usage on training hosts, practically eliminating the host-side bottleneck.

5.5 Future Directions

Storage-side transformation offload depends on co-located compute headroom in the storage tier. This assumption holds in our environment because the same storage clusters support large-scale data processing and often have idle CPU cycles while waiting on disks or network. It may not hold for cloud object stores or lean storage appliances that expose little programmable compute near the data [2, 28]. In such cases, other techniques are also possible to improve the transformation performance and reduce the training slowdown:

Processed-sample caching: Caching preprocessed samples [22, 30, 58], as previous work suggests, can complement storage-side offload when training repeatedly revisits data. In large-scale pre-training, which is the focus of this paper, globally shuffled samples are rarely revisited within the useful lifetime of a cache, so processed-data caching provides little hit-rate benefit while adding consistency and capacity pressure. In regimes with repeated sample access, such as reinforcement learning or repeated fine-tuning over a smaller corpus, caching becomes effective and can store the transformed outputs produced by the storage-side engine.

Predictive transform-aware batching: Since the deterministic dataloader knows the exact sample sequence offline, a lightweight cost model can be used to predict each sample's transformation cost in advance, based on metadata such as raw sample size, codec, resolution, and clip duration. With these estimates, batches can then be composed to balance transformation cost across hosts, mixing heavy and light samples so that no single host is assigned a disproportionate share of expensive clips.

6 What If?

Building an efficient data pipeline for exabyte-scale pre-training necessitates navigating trade-offs between storage cost, compute efficiency, and engineering complexity. Along our operational journey, we critically evaluated several alternatives that we eventually discarded as impractical at scale. We structure this discussion as a series of "What If" questions.

Q1: What if we used Peer-to-Peer (P2P) distribution for model loading?

In Section 4, we address the initialization storm by proactively replicating data on storage nodes. While P2P distribution (e.g., NCCL [10] and UCX [45]) theoretically scales linearly with cluster size, we strictly prioritize a hierarchical client-server model to mitigate multi-tenancy interference and tail latency sensitivity.

First, large GPU clusters concurrently execute multiple training tasks. P2P's unstructured, all-to-all traffic pattern induces network congestion, saturating buffers and disrupting the latency-sensitive NCCL collectives of concurrent jobs. Second, training initialization functions as a synchronous barrier constrained by the slowest node. P2P performance re-

mains inherently probabilistic; a single slow peer—suffering from throttling or local congestion—delays the startup of the entire gang. In contrast, our hierarchical replication guarantees a deterministic SLA with confined north-south traffic, effectively isolating the cluster from cross-tenant interference and probabilistic tail latencies.

Q2: What if we deploy a dedicated transformation cluster as an alternative to storage-side processing?

In Section 5, we address multimodal transformation overhead by offloading compute to storage nodes. An alternative is a dedicated sample transformation cluster, a fleet of CPU-rich machines that decode and transform data before forwarding tensors to training hosts. This architecture, exemplified by tf.data service [33] and Ray Data [31], decouples data preparation from training. We reject this design due to significant network amplification and operational overhead.

First, decoded tensors dwarf their compressed sources. Video decompression inflates data volume by 50–100×; transferring these tensors over the datacenter network shifts the bottleneck from CPU to bandwidth and saturates the shared fabric. Second, a standalone cluster introduces provisioning complexity: capacity must be sized for peak multimodal demand yet remains idle during text-only workloads. In contrast, our storage-side offloading exploits spare CPU cycles already provisioned on storage nodes, avoiding network amplification and eliminating the need for a dedicated resource pool.

Q3: What if we migrate to a specialized AI-native storage system as a modern alternative to HDFS?

Recent works propose specialized AI storage systems (e.g., 3FS, AIStore) offering superior small-file performance and kernel-bypass networking. We retain and optimize a legacy Big Data system over a greenfield solution primarily due to data gravity and operational continuity.

HDFS serves as our unified data lake, hosting exabytes of data across diverse business lines beyond AI training. Migrating this volume to a new system is operationally infeasible and cost-prohibitive. Moreover, AI-native systems often require specific hardware configurations (e.g., all-NVMe) or custom clients that break compatibility with the broader ecosystem (e.g., Spark-based cleaning pipelines). Our work demonstrates that by exposing application-level patterns, legacy systems like HDFS can meet strict LLM training requirements without wholesale replacement. This software-defined approach enables iterative infrastructure evolution while maintaining service stability.

7 Related Work

Storage infrastructure for model training. Traditional Parallel File Systems (PFS) such as Lustre [44] and GPFS [43] provide high checkpointing bandwidth but degrade under the massive metadata pressure of multimodal datasets. Conversely, disaggregated object stores like S3 [2] and Azure Blob [28] offer scalability but incur high latency and limited random access throughput. While emerging AI-native systems (e.g., 3FS [11], AIStore [9]) address these limitations via kernel-bypass networking and specialized caching, they often require greenfield deployments. As detailed in Section 6, migrating exabytes of legacy data to specialized stacks is operationally infeasible. Our work demonstrates that by exposing application-level determinism, general-purpose Big Data systems (HDFS [47]) can meet strict training SLAs without fundamental architectural replacement.

Data loading and global coordination. Prior research optimizes data loading through domain-specific caching (e.g., Quiver [22] and SiloD [58]) or coordinated fetching (e.g., CoorDL [30]). These systems are primarily reactive, depending on runtime hit rates. In contrast, our deterministic dataloader (Section 5.1) exploits the determinism of LLM training to convert random access into sequential streams. Regarding data formats, solutions like TFRecord [33], WebDataset [53], and Lance [36] mitigate random seeks by aggregating samples but complicate global shuffling. Our index-based approach ensures global randomness without physical data movement. Finally, while works like CrossPipe [5] optimize pipeline schedules, we address the storage logistics of companion evaluation (Section 3) by leveraging namespace-level predictive checkpoint replication to mask cross-DC latency.

8 Conclusion

We identify critical storage bottlenecks in large-scale pre-training, including cross-DC latency during companion evaluation, initialization storms, and multimodal transformation overhead. By leveraging workload determinism, we implement software-defined optimizations such as predictive checkpoint replication, proactive replication, and storage-side offloading to convert the storage tier into a proactive partner. These mechanisms significantly reduce companion evaluation latency, startup time, and data loading stalls. Our work demonstrates that legacy systems like HDFS can effectively support exabyte-scale training through vertical co-design.

Acknowledgments

We are grateful to the anonymous reviewers and shepherd for their constructive feedback. We also thank our colleagues from the storage, training infrastructure, and pre-training teams for their support in deploying and evaluating the techniques described in this paper. This work is supported in part by the National Natural Science Foundation of China under Grant No.: U25B2020 and 62572454. Weidong Zhang and Youhui Bai are the corresponding authors.

References

[1] Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, et al. GPT-4 technical report. *arXiv preprint arXiv:2303.08774*, 2023.

[2] Amazon. Amazon S3 - cloud object storage - AWS. https://aws.amazon.com/s3/, June 2026.

[3] Ganesh Ananthanarayanan, Sameer Agarwal, Srikanth Kandula, Albert Greenberg, Ion Stoica, Duke Harlan, and Ed Harris. Scarlett: Coping with skewed content popularity in mapreduce clusters. In *Proceedings of the Sixth Conference on Computer Systems*, EuroSys '11, pages 287–300, New York, NY, USA, April 2011. Association for Computing Machinery.

[4] BigScience Workshop. BigScience/train/tr11-176b-ml/chronicles.md. https://github.com/bigscience-workshop/bigscience/blob/master/train/tr11-176B-ml/chronicles.md, June 2026.

[5] Tiancheng Chen, Ales Kubicek, Langwen Huang, and Torsten Hoefler. CrossPipe: Towards optimal pipeline schedules for cross-datacenter training. In *Proceedings of the 2025 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC '25, pages 1089–1108, USA, July 2025. USENIX Association.

[6] Aakanksha Chowdhery, Sharan Narang, Jacob Devlin, Maarten Bosma, Gaurav Mishra, Adam Roberts, Paul Barham, Hyung Won Chung, Charles Sutton, Sebastian Gehrmann, Parker Schuh, Kensen Shi, Sasha Tsvyashchenko, Joshua Maynez, Abhishek Rao, Parker Barnes, Yi Tay, Noam Shazeer, Vinodkumar Prabhakaran, Emily Reif, Nan Du, Ben Hutchinson, Reiner Pope, James Bradbury, Jacob Austin, Michael Isard, Guy Gur-Ari, et al. PaLM: Scaling language modeling with pathways. *Journal of Machine Learning Research*, 24(240):1–113, 2023.

[7] Gheorghe Comanici, Eric Bieber, Mike Schaekermann, Ice Pasupat, Noveen Sachdeva, Inderjit Dhillon, Marcel Blistein, Ori Ram, Dan Zhang, Evan Rosen, et al. Gemini 2.5: Pushing the frontier with advanced reasoning, multimodality, long context, and next generation agentic capabilities. *arXiv preprint arXiv:2507.06261*, 2025.

[8] PyTorch Contributors. Getting started with distributed checkpoint (DCP). https://docs.pytorch.org/tutorials/recipes/distributed_checkpoint_recipe.html, June 2026.

[9] NVIDIA Corporation. NVIDIA/AIStore. https://github.com/NVIDIA/aistore, June 2026.

[10] NVIDIA Corporation. NVIDIA/NCCL. https://github.com/NVIDIA/nccl, June 2026.

[11] DeepSeek-AI. DeepSeek-AI/3FS: A high-performance distributed file system designed to address the challenges of AI training and inference workloads. https://github.com/deepseek-ai/3FS, June 2026.

[12] Dror G. Feitelson and Larry Rudolph. Gang scheduling performance benefits for fine-grain synchronization. *Journal of Parallel and Distributed Computing*, 16(4):306–318, December 1992.

[13] Aaron Grattafiori, Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Alex Vaughan, et al. The Llama 3 herd of models. *arXiv preprint arXiv:2407.21783*, 2024.

[14] Wenyi Hong, Yean Cheng, Zhuoyi Yang, Weihan Wang, Lefan Wang, Xiaotao Gu, Shiyu Huang, Yuxiao Dong, and Jie Tang. MotionBench: Benchmarking and improving fine-grained video motion understanding for vision language models, 2025.

[15] Yanping Huang, Youlong Cheng, Ankur Bapna, Orhan Firat, Dehao Chen, Mia Chen, HyoukJoong Lee, Jiquan Ngiam, Quoc V Le, Yonghui Wu, et al. GPipe: Efficient training of giant neural networks using pipeline parallelism. *Advances in neural information processing systems*, 32, 2019.

[16] Changho Hwang, Wei Cui, Yifan Xiong, Ziyue Yang, Ze Liu, Han Hu, Zilong Wang, Rafael Salas, Jithin Jose, Prabhat Ram, HoYuen Chau, Peng Cheng, Fan Yang, Mao Yang, and Yongqiang Xiong. Tutel: Adaptive mixture-of-experts at scale. In D. Song, M. Carbin, and T. Chen, editors, *Proceedings of Machine Learning and Systems*, volume 5, pages 269–287. Curan, 2023.

[17] ITU-T. H.265: High efficiency video coding. https://www.itu.int/rec/T-REC-H.265, June 2026.

[18] Ziheng Jiang, Haibin Lin, Yinmin Zhong, Qi Huang, Yangrui Chen, Zhi Zhang, Yanghua Peng, Xiang Li, Cong Xie, Shibiao Nong, et al. MegaScale: Scaling large language model training to more than 10,000 GPUs. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*, pages 745–760, 2024.

[19] Joint Photographic Experts Group. JPEG. https://jpeg.org/index.html, June 2026.

[20] Heehoon Kim, Junyeol Ryu, and Jaejin Lee. TCCL: Discovering better communication paths for PCIe GPU clusters. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, volume 3

of *ASPLOS '24*, pages 999–1015, New York, NY, USA, April 2024. Association for Computing Machinery.

[21] Diederik P Kingma. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[22] Abhishek Vijaya Kumar and Muthian Sivathanu. Quiver: An informed storage cache for deep learning. In *18th USENIX Conference on File and Storage Technologies (FAST 20)*, pages 283–296. USENIX Association, February 2020.

[23] Jiahao Li, Biao Cao, Jielong Jian, Cheng Li, Sen Han, Yiduo Wang, Yufei Wu, Kang Chen, Zhihui Yin, Qiushi Chen, et al. Mantle: Efficient hierarchical metadata management for cloud object storage services. In *Proceedings of the ACM SIGOPS 31st Symposium on Operating Systems Principles*, pages 928–943, 2025.

[24] Zhiqi Lin, Youshan Miao, Quanlu Zhang, Fan Yang, Yi Zhu, Cheng Li, Saeed Maleki, Xu Cao, Ning Shang, Yilei Yang, et al. nnScaler: Constraint-guided parallelization plan generation for deep learning training. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, pages 347–363, 2024.

[25] Aixin Liu, Bei Feng, Bing Xue, Bingxuan Wang, Bochao Wu, Chengda Lu, Chenggang Zhao, Chengqi Deng, Chenyu Zhang, Chong Ruan, et al. DeepSeek-V3 technical report. *arXiv preprint arXiv:2412.19437*, 2024.

[26] Quanfeng Lu, Wenqi Shao, Zitao Liu, Fanqing Meng, Boxuan Li, Botong Chen, Siyuan Huang, Kaipeng Zhang, Yu Qiao, and Ping Luo. Gui odyssey: A comprehensive dataset for cross-app gui navigation on mobile devices. *arXiv preprint arXiv:2406.08451*, 2024.

[27] Kaijing Ma, Xinrun Du, Yunran Wang, Haoran Zhang, Zhoufutu Wen, Xingwei Qu, Jian Yang, Jiaheng Liu, Minghao Liu, Xiang Yue, Wenhao Huang, and Ge Zhang. KOR-Bench: Benchmarking language models on knowledge-orthogonal reasoning tasks, 2025.

[28] Microsoft. Azure Blob Storage | Microsoft Azure. https://azure.microsoft.com/en-us/products/storage/blobs, June 2026.

[29] Jayashree Mohan, Amar Phanishayee, and Vijay Chidambaram. CheckFreq: Frequent, fine-grained DNN checkpointing. In *19th USENIX Conference on File and Storage Technologies (FAST 21)*, pages 203–216, 2021.

[30] Jayashree Mohan, Amar Phanishayee, Ashish Raniwala, and Vijay Chidambaram. Analyzing and mitigating data stalls in dnn training. *Proc. VLDB Endow.*, 14(5):771–784, January 2021.

[31] Philipp Moritz, Robert Nishihara, Stephanie Wang, Alexey Tumanov, Richard Liaw, Eric Liang, Melih Elibol, Zongheng Yang, William Paul, Michael I Jordan, et al. Ray: A distributed framework for emerging AI applications. In *13th USENIX symposium on operating systems design and implementation (OSDI 18)*, pages 561–577, 2018.

[32] MosaicML. MosaicML/streaming: A data streaming library for efficient neural network training. https://github.com/mosaicml/streaming, June 2026.

[33] Derek G. Murray, Jiří Šimša, Ana Klimovic, and Ihor Indyk. tf.data: A machine learning data processing framework. *Proceedings of the VLDB Endowment*, 14(12):2945–2958, July 2021.

[34] Deepak Narayanan, Aaron Harlap, Amar Phanishayee, Vivek Seshadri, Nikhil R Devanur, Gregory R Ganger, Phillip B Gibbons, and Matei Zaharia. Pipedream: Generalized pipeline parallelism for dnn training. In *Proceedings of the 27th ACM symposium on operating systems principles*, pages 1–15, 2019.

[35] NVIDIA. Dist_checkpointing package — Megatron Core. https://docs.nvidia.com/megatron-core/developer-guide/latest/api-guide/core/dist_checkpointing.html, June 2026.

[36] Weston Pace, Chang She, Lei Xu, Will Jones, Albert Lockett, Jun Wang, and Raunak Shah. Lance: Efficient random access in columnar storage through adaptive structural encodings, April 2025.

[37] PyTorch. Getting started with fully sharded data parallel (FSDP2) — PyTorch tutorials 2.9.0+cu128 documentation. https://docs.pytorch.org/tutorials/intermediate/FSDP_tutorial.html, June 2026.

[38] PyTorch. PyTorch profiler — PyTorch tutorials 2.9.0+cu128 documentation. https://docs.pytorch.org/tutorials/recipes/recipes/profiler_recipe.html, June 2026.

[39] PyTorch. torch.utils.data — PyTorch 2.9 documentation. https://docs.pytorch.org/docs/stable/data.html, June 2026.

[40] Samyam Rajbhandari, Jeff Rasley, Olatunji Ruwase, and Yuxiong He. ZeRO: Memory optimizations toward training trillion parameter models. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–16. IEEE, 2020.

[41] Zhenghang Ren, Yuxuan Li, Zilong Wang, Xinyang Huang, Wenxue Li, Kaiqiang Xu, Xudong Liao, Yijun Sun, Bowen Liu, Han Tian, Junxue Zhang, Mingfei Wang, Zhizhen Zhong, Guyue Liu, Ying Zhang, and

Kai Chen. Enabling efficient GPU communication over multiple NICs with FuseLink. In *19th USENIX Symposium on Operating Systems Design and Implementation (OSDI 25)*, pages 91–108, 2025.

[42] Safetensors Contributors. safetensors: Simple, safe way to store and distribute tensors. https://github.com/safetensors/safetensors, June 2026.

[43] Frank Schmuck and Roger Haskin. GPFS: A shared-disk file system for large computing clusters. In *Proceedings of the 1st USENIX Conference on File and Storage Technologies*, FAST'02, page 16, USA, January 2002. USENIX Association.

[44] Philip Schwan et al. Lustre: Building a file system for 1000-node clusters. In *Proceedings of the 2003 Linux symposium*, volume 2003, pages 380–386, 2003.

[45] Pavel Shamis, Manjunath Gorentla Venkata, M Graham Lopez, Matthew B Baker, Oscar Hernandez, Yossi Itigin, Mike Dubman, Gilad Shainer, Richard L Graham, Liran Liss, et al. UCX: an open source framework for HPC network APIs and beyond. In *2015 IEEE 23rd Annual Symposium on High-Performance Interconnects*, pages 40–43. IEEE, 2015.

[46] Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. Megatron-LM: Training multi-billion parameter language models using model parallelism, March 2020.

[47] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. The Hadoop distributed file system. In *2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*, pages 1–10, May 2010.

[48] Kimi Team, Yifan Bai, Yiping Bao, Guanduo Chen, Jiahao Chen, Ningxin Chen, Ruijue Chen, Yanru Chen, Yuankun Chen, Yutian Chen, et al. Kimi K2: Open agentic intelligence. *arXiv preprint arXiv:2507.20534*, 2025.

[49] Deepak Vohra. Apache parquet. In Deepak Vohra, editor, *Practical Hadoop Ecosystem: A Definitive Guide to Hadoop-Related Frameworks and Tools*, pages 325–335. Apress, Berkeley, CA, 2016.

[50] Borui Wan, Mingji Han, Yiyao Sheng, Yanghua Peng, Haibin Lin, Mofan Zhang, Zhichao Lai, Menghan Yu, Junda Zhang, Zuquan Song, Xin Liu, and Chuan Wu. ByteCheckpoint: A unified checkpointing system for large foundation model development. In *22nd USENIX Symposium on Networked Systems Design and Implementation (NSDI 25)*, pages 559–578, 2025.

[51] Borui Wan, Gaohong Liu, Zuquan Song, Jun Wang, Yun Zhang, Guangming Sheng, Shuguang Wang, Houmin Wei, Chenyuan Wang, Weiqiang Lou, et al. Robust LLM training infrastructure at ByteDance. In *Proceedings of the ACM SIGOPS 31st Symposium on Operating Systems Principles*, pages 186–203, 2025.

[52] Guanhua Wang, Olatunji Ruwase, Bing Xie, and Yuxiong He. FastPersist: Accelerating model checkpointing in deep learning. *arXiv preprint arXiv:2406.13768*, 2024.

[53] webdataset. WebDataset/WebDataset. https://github.com/webdataset/webdataset, June 2026.

[54] Ouri Wolfson, Sushil Jajodia, and Yixiu Huang. An adaptive data replication algorithm. *ACM Trans. Database Syst.*, 22(2):255–314, June 1997.

[55] Guanbin Xu, Zhihao Le, Yinhe Chen, Zhiqi Lin, Zewen Jin, Youshan Miao, and Cheng Li. AutoCCL: Automated collective communication tuning for accelerating distributed and parallel DNN training. In *22nd USENIX Symposium on Networked Systems Design and Implementation (NSDI 25)*, pages 667–683, 2025.

[56] An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengen Huang, Chenxu Lv, et al. Qwen3 technical report. *arXiv preprint arXiv:2505.09388*, 2025.

[57] Susan Zhang, Stephen Roller, Naman Goyal, Mikel Artetxe, Moya Chen, Shuohui Chen, Christopher Dewan, Mona Diab, Xian Li, Xi Victoria Lin, Todor Mihaylov, Myle Ott, Sam Shleifer, Kurt Shuster, Daniel Simig, Punit Singh Koura, Anjali Sridhar, Tianlu Wang, and Luke Zettlemoyer. OPT: Open pre-trained transformer language models, 2022.

[58] Hanyu Zhao, Zhenhua Han, Zhi Yang, Quanlu Zhang, Mingxia Li, Fan Yang, Qianxi Zhang, Binyang Li, Yuqing Yang, Lili Qiu, et al. SiloD: A co-design of caching and scheduling for deep learning clusters. In *Proceedings of the Eighteenth European Conference on Computer Systems*, pages 883–898, 2023.

[59] Yanli Zhao, Andrew Gu, Rohan Varma, Liang Luo, Chien-Chin Huang, Min Xu, Less Wright, Hamid Shojanazeri, Myle Ott, Sam Shleifer, Alban Desmaison, Can Balioglu, Pritam Damania, Bernard Nguyen, Geeta Chauhan, Yuchen Hao, Ajit Mathews, and Shen Li. PyTorch FSDP: Experiences on scaling fully sharded data parallel, September 2023.

[60] Lianmin Zheng, Zhuohan Li, Hao Zhang, Yonghao Zhuang, Zhifeng Chen, Yanping Huang, Yida Wang, Yuanzhong Xu, Danyang Zhuo, Eric P Xing, et al. Alpa:

Automating inter-and intra-operator parallelism for distributed deep learning. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 559–578, 2022.

ARTIFACT EVALUATED
USENIX
AVAILABLE

Cocoon: A System Architecture for Differentially Private Training with Correlated Noises

Donghwan Kim[1], Xin Gu[1], Jinho Baek[2], Timothy Lo[2], Younghoon Min[2], Kwangsik Shin[2], Jongryool Kim[2], Jongse Park[3], Kiwan Maeng[1]

[1]*The Pennsylvania State University*
[2]*SK hynix Inc.* [3]*KAIST*

Abstract

Machine learning (ML) models memorize and leak training data, causing serious privacy issues to data owners. Training algorithms with differential privacy (DP) have been gaining attention as a solution. However, these algorithms add noise at each training iteration and degrade accuracy, limiting their real-world adoption. To improve accuracy, a new family of approaches adds carefully designed *correlated noises*, so that noises cancel out each other across iterations. We performed an extensive characterization study of these new mechanisms and show they incur non-negligible overheads when the model is relatively large or uses large embedding tables compared to the hardware capacity. Motivated by the analysis, we propose Cocoon, a framework for efficient training with correlated noises. Cocoon stores and processes the large noise history across CPU, GPU, and memory extension module, introduces optimizations for sparse embedding tables, and leverages to-be-commercialized near-memory processing (NMP) devices. On a real system with an FPGA-based NMP device prototype, Cocoon improves the performance by 1.23–10.82×.

1 Introduction

Machine learning (ML) models are vulnerable to memorization and leakage of their training data—sometimes by accident, or when model is subjected to an adversarial attack. This poses a serious *privacy risk* because ML models are often trained with sensitive user data (*e.g.*, medical data [108], audio [66] or keyboard inputs [82,113], personal chat logs [103], *etc.*). The threat is not hypothetical but imminent: a recent study demonstrated the feasibility of attacking the popular ChatGPT [68] to extract its training data that contains sensitive information. Similar attacks have been shown to be possible for other models [9, 10, 28, 50]. The privacy risks can deter user participation in data collection and training, degrading the quality of real-world ML-based services [60].

Differential privacy (DP) [25] is one of the most principled and widely adopted approaches for mitigating such privacy risks. DP theoretically bounds the information leakage of private data through its statistics. In the context of ML training, this translates to a mathematical bound on the probability that an adversary can successfully extract information of the training data from the trained model [36,37,39,49,70]. DP is not a premature research concept; already deployed in several real-world applications, including the US Census [3], Apple's device analytics [86,91], and Google Chrome's behavior telemetry [26]. The use of DP for ML training is also gaining increased attention. For instance, Google publicly announced its use of DP for training all of their current and future smart keyboard (Gboard) models [101,113], and also recently released VaultGemma [89], an LLM that was trained with DP from scratch. Microsoft has also limitedly deployed a DP-trained model on their SwiftKey keyboard [2].

Most DP training algorithms add Gaussian noise to the gradient at each training iteration to provide a theoretical privacy guarantee [1, 21]. This Gaussian noise accumulates over training and significantly degrades the model accuracy, which is the primary barrier to DP's widespread adoption. To counteract the accuracy issue, a recent line of works [13–16,48,63–65,79,109] proposed to use noises that are ***correlated*** across iterations, allowing noises added in later iterations can partially cancel out earlier noises. *Correlated noise mechanisms* have demonstrated both theoretical [13,55] and empirical [13–16,48,63–65,79,109] benefits over simply adding Gaussian noise. The mechanism has already been successfully deployed in real-world products [101,113], and adopted into popular ML privacy libraries [32,33]. Their rapid adoption send a strong signal of their importance.

Despite their growing popularity, the system implications of correlated noise have not been thoroughly studied. Existing literature mostly focused on accuracy rather than system efficiency, training relatively small models with a large number of TPUs/GPUs (*e.g.*, using 1024 machines to train a billion-parameter model [63]). In these *resource-abundant* environments, noise-related overheads are small. However, our study reveals that correlated noise incurs significant system overheads, including a memory footprint that exceeds the pa-

rameter size by over an order of magnitude, and causing up to 14.49× runtime slowdown. Such costs are particularly severe for *economical* setups with fewer TPUs/GPUs, which is a configuration frequently studied to enable large model training on small and mid-sized entities [5, 83–85, 110]. This overhead arises because generating correlated noises involve storing and processing noise history from past iterations, which can scale up to hundreds of gigabytes or even terabytes. It becomes especially problematic for (1) models with large embedding tables and (2) large-scale, billion-parameter models (*e.g.*, LLMs).

We introduce *Cocoon*, a framework for efficient at-scale DP training with correlated noises. Cocoon splits the large noise history between CPU, GPU, and (if present) additional memory extension module, and processes them in a distributed fashion to maximize performance. To mitigate managing noise history for large embedding tables, Cocoon precomputes and stores the noise for embedding tables in a compact format by exploiting their gradient sparsity (Section 4.2). Furthermore, when part of the noise history is offloaded to a memory extension module, we show that leveraging emerging near-memory processing (NMP) solutions [46, 87] can greatly improve performance and reduce the cost of deployment (Section 4.3). Our evaluation on a real system, with an FPGA-based industry prototype NMP device, showed 1.23–10.82× speedup on various models and training scenarios. We summarize our contributions:

- We present an extensive system characterization study of emerging DP training methods that use correlated noises. Our analysis reveals that, on an economical setup where the number of GPUs is moderate, correlated noise generation can introduce non-negligible slowdown.
- We introduce Cocoon, a highly-optimized, PyTorch-based DP training library that uses correlated noises. Cocoon splits and processes noise history on CPU, GPU, and (if present) emerging NMP devices.
- Cocoon introduces a noise pre-computing and coalescing optimization that accelerates training with large embedding tables by 2.33–10.82×. With to-be-commonly-deployed NMP devices, Cocoon can achieve 1.23–2.32× speedup over the baseline when training large models, which we demonstrate with a real industry prototype.

2 Background and Motivation

2.1 Differentially Private Training Algorithms

Differential privacy. Differential privacy [25] (DP) provides a robust guarantee that the outcome of a randomized mechanism does not change significantly when a single entry in the input dataset changes. When applied to ML training, DP ensures that the final trained model does not depend significantly on a single sample (or user [11, 19]) in the training corpus, which effectively limits an adversary's capabilities. For example, DP parameters can theoretically bound the attack success rate for adversaries attempting to infer whether a user participated during training [49, 70] or trying to directly reconstruct the original training data [36, 37, 39] from a DP-trained model. A model trained with DP can be used for multiple inferences over time, and it will not leak additional information beyond the privacy cost paid upfront during training. This is in contrast with inference-time DP techniques [31, 54, 94, 97], which leak more information about the corpus with more inferences [80, 97].

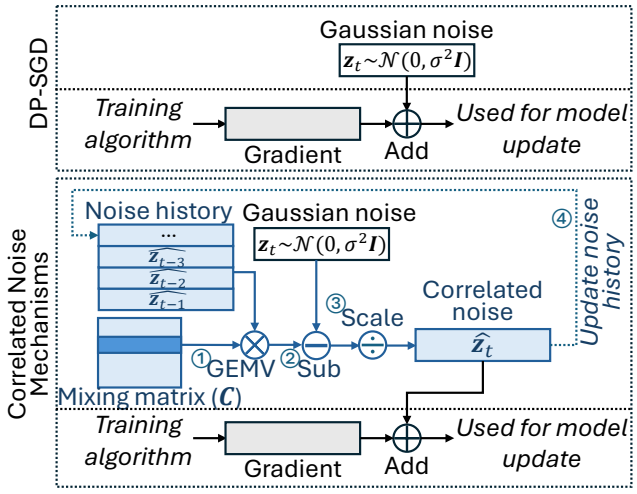


Figure 1: Independent Gaussian vs. correlated noise.

Training with Gaussian noise. The most common method to train a model with DP is using DP-SGD [1] or its variants. DP-SGD introduces three key modifications to regular SGD. First, the batch in each iteration is formed through random sampling with replacement. Second, unlike SGD, which calculates an average gradient across the batch, DP-SGD first calculates per-sample gradients, scales (clips) each one individually, and then averages the scaled gradients. Finally, an independently sampled Gaussian noise is added to the averaged gradient before it is applied to update the model [1]. While these steps provide stronger privacy guarantees, DP-SGD significantly compromises the trained model's accuracy.

Several follow-up works have introduced optimizations to improve the model accuracy over the DP-SGD algorithm. Most of this research still relies on *independent Gaussian noise* and focuses on optimizing other components of DP-SGD (e.g., sampling [20], clipping [7, 67], hyperparameter [21], feature engineering [96], post-processing after noise addition [34, 112], optimizer [12, 29, 92, 93, 102], and architecture [21, 74]). These optimizations still suffer from accuracy degradation due to Gaussian noise. Figure 1 (top) highlights the independent Gaussian noise added to each gradient in DP-SGD.

Correlated noise mechanisms. Instead of adding independent Gaussian noise, a recent line of works [13–16, 48, 63–65, 79] proposed to add noise that is *correlated* across iterations. When generating correlated noises at iteration t, these mechanisms mix noises that were used in the previous $\hat{b}-1$ iterations (*i.e.*, noise history), where $\hat{b}$ is called the *band size*.

Mathematically, correlated noises are generated as follows. For a model with m trainable parameters trained through n training iterations, let $\mathbf{z}_t \in \mathbb{R}^m$ be a Gaussian noise sampled at iteration t. Each noise is as large as the trainable parameters (m), and training a large model requires using a noise that is as large. For a specific *mixing matrix* $\mathbf{C} \in \mathbb{R}^{n \times n}$, the correlated noise at iteration t, $\hat{\mathbf{z}}_t \in \mathbb{R}^m$, can be calculated as:

$$\hat{\mathbf{z}}_t = (\mathbf{z}_t - \sum_{\tau=1}^{\min(t,\hat{b}-1)} \mathbf{C}[t,t-\tau]\hat{\mathbf{z}}_{t-\tau})/\mathbf{C}[t,t]. \quad (1)$$

Figure 1 (bottom) visualizes how $\hat{\mathbf{z}}_t$ is calculated. First, a ① *weighted average* of the $\hat{b}-1$ previous noises is performed. The result is ② subtracted from a newly-sampled Gaussian noise ($\mathbf{z}_t$) and ③ properly rescaled, which becomes the new correlated noise ($\hat{\mathbf{z}}_t$). As in DP-SGD, $\hat{\mathbf{z}}_t$ is added to the gradient, and the noisy gradient is used to update the model. At the same time, ④ $\hat{\mathbf{z}}_t$ is saved to the noise history, so that it can be used to generate future noises.

The weighting/rescaling factors at t-th iteration are defined by the t-th row of the mixing matrix $\mathbf{C}$, and the weighted averaging can be done through a *matrix-vector multiplication (GEMV)* between the stacked noise history (matrix of size $(\hat{b}-1) \times m$) and the t-th row of $\mathbf{C}$ (each row is of size n but only has $\hat{b}-1$ nonzero elements). $\mathbf{C}$ should be carefully designed to guarantee DP, and different prior works developed different designs of $\mathbf{C}$ [15, 48, 65]. When $\hat{b}=1$ and $\mathbf{C}=\mathbf{I}$ (an identity matrix), this reduces to independent Gaussian noise.

Generality and importance of correlated noise. As Figure 1 shows, correlated noise can almost serve as a drop-in replacement for independent Gaussian noise[1]. Thus, making the correlated noise generation efficient is likely to benefit a wide range of DP training algorithms that currently use independent Gaussian noise. Several recent works have demonstrated that correlated noise has theoretical [13, 55] and empirical [13–16, 48, 63–65, 79, 109] benefits over independent Gaussian noise. Google recently adopted correlated noise in its production (Gboard training [101, 113]) and has been exploring using it for larger models [63, 79], which strongly signals its practical importance. A recent 212-page report [79] summarizes the latest efforts around correlated noise. Its generality and importance motivate our study.

[1]When using correlated noise, it puts a restriction on what data sampling method can be used, but other parts of the training do not need to change.

2.2 Training of Embedding Tables

An embedding table is a trainable data structure that converts categorical features into a dense vector. It is commonly used in deep learning recommendation models (DLRMs) or large language models (LLMs). Especially in DLRMs, embedding tables are very tall and dominate the model size [38, 62, 71]. During each training iteration, only rows (entries) of the table corresponding to the present feature values of the input are used, which leads to several unique behaviors. First, the training speed grows sub-linearly with its size compared to other models, because only a tiny subset is used even when the entire table is large. Second, unused entries in each iteration have zero gradients. Still, DP training requires noise addition to these zero gradients for privacy [62, 72]. Third, only the entries accessed in each iteration contribute to the gradient calculation at that iteration. As we will show later, the first characteristic leads to a unique overhead when using correlated noises, and the latter two will be leveraged by Cocoon's optimizations (Section 4.2).

2.3 CXL and Near-memory Processing (NMP)

Compute express link (CXL) is an open industry interconnect standard based on PCIe, which allows high-speed loads and stores for memory expansion modules plugged into the PCIe slot. Memory expansion module connected through CXL, or *CXL memory*, has emerged as a way to expand memory capacity. We study the use of such CXL memory to expand the device memory capacity when the noise history is too large to be hosted on main memory.

While CXL memory provides larger memory, data access through CXL is much slower than the main memory of the CPU. Several recent works [35, 51, 53, 78, 88, 107] have proposed to put additional compute units in the CXL controller to reduce data movement through PCIe, and some have made it into (near-)real products [46, 87]. In Section 4.3, we show how such emerging *near-memory processing (NMP) CXL memory* products can accelerate Cocoon.

3 Characterization of Correlated Noise

To study the system overheads of correlated noise, we trained various models with independent Gaussian noise and correlated noise. We used a correlated noise mechanism called BandMF [15]. Our study can be generalized to other mechanisms, because other mechanisms only differ in the derivation of mixing matrix $\mathbf{C}$ (Section 2.1), and are computationally equivalent. Our study highlights that correlated noise mechanisms experience non-negligible memory (Section 3.1) and compute (Section 3.2) overheads.

For the study, we trained popular models from prior DP training literature on a dual-socket Intel Xeon Gold 6330 CPU with 256GB DRAM and 8x NVIDIA A5000 GPUs, which

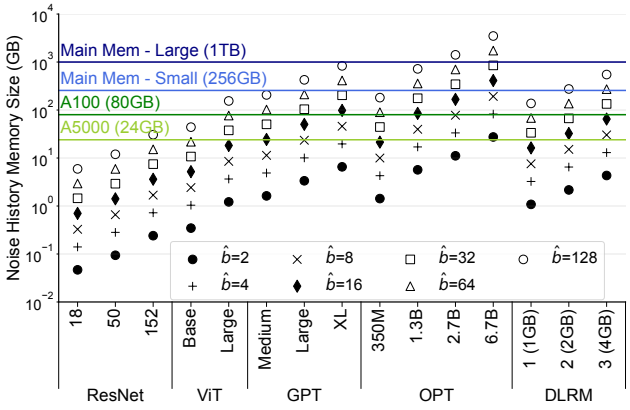


Figure 2: Noise history size of various ML models and $\hat{b}$.

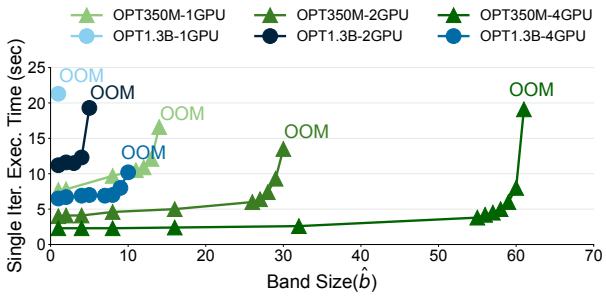


Figure 3: Training time of OPT [111] on 1–4 A5000 GPUs.

represents an economical setup of a small/mid-sized entity. The models we trained include: ResNet [41], ViT [24], LLMs (GPT [81] and OPT [111]), and DLRM [71]. Measurements were done on our custom DP training code with correlated noise support, which we built as part of our Cocoon library (Section 4). We only highlight the most interesting results.

3.1 Memory Overhead

Generating correlated noise involves storing $\hat{b}-1$ past noises, each of which is as large as the number of trainable parameters (m). This introduces several capacity and performance issues.

Capacity issue. Figure 2 summarizes the memory footprint of the noise history for various models and $\hat{b}$, along with common GPU memory and main memory (CPU DRAM) sizes. The footprint frequently exceeds the GPU/CPU memory, and the noise history must be offloaded to main memory or secondary memory (*e.g.*, CXL memory or SSD) to run the training at all in these cases. Performance implications of these fallbacks are discussed in Section 3.2. Some works [63, 79] found such offloading to be unnecessary, but this was because they had enough GPUs/TPUs, unlike our economical setup.

Takeaway 1: The noise history can become larger than the aggregate GPU/TPU memory or even the main memory.

Performance issue. Even when the entire noise history fits into the GPU, the performance can degrade if too little memory is left for training. Figure 3 shows the training latency of two OPT models [111], where different lines correspond to different models and GPU counts. It can be seen that the training time increases with $\hat{b}$ for all setups, until an out-of-memory (OOM) error is triggered. This is because if too little memory is left, training must use smaller microbatches, underutilizing the GPU [98].

Takeaway 2: Even when storing the entire noise on the GPU is possible, it may degrade the training performance.

Why not re-generate noises? Instead of storing past noises, prior work [48] considered only storing the seed and re-generating noises on every iteration. However, doing so requires re-generating all the noises from the beginning $(\hat{\mathbf{z}}_1, \ldots, \hat{\mathbf{z}}_t)$ and not just the past $\hat{b}-1$ noises, because each noise generation recursively requires its past $\hat{b}-1$ noises. This incurs $O(n^2)$ overhead for n training iterations, which becomes too large unless n is very small. Another prior work [79] similarly observed that this approach scales poorly with n.

3.2 Compute Overhead

The weighted averaging of prior noises (GEMV between the noise history and the mixing vector) also incurs a computational overhead. We mainly consider economical setups where part of the noise history is stored in main/secondary memory.

For the parts hosted outside the GPU, we studied two options for processing them: (1) *GPU-GEMV* performs GEMV only on the GPU. While GEMV is fast on a GPU, this design requires additional data transfer from the main/secondary memory to GPU through the slow PCIe bus. (2) *CPU-GEMV* performs GEMV on the CPU for the subset of the noise history stored in main/secondary memory, and only sends the result to the GPU. While CPU's GEMV is slower, this design enjoys the higher bandwidth between the CPU and main memory, compared to the PCIe bus. Also, CPU-side GEMV can happen *in parallel* with GPU-side training and can be partially or completely hidden.

Overheads of DLRM. Correlated noises incur a unique overhead on DLRMs due to their large embedding tables (Section 2.2). Figure 4 shows the training time of three DLRMs with different embedding dimensions (DLRM-1/2/3). We only characterized a single-GPU setup because the single-iteration latency was too small (near 100ms) to make data parallel training effective. GPU-GEMV's latency consists of the GPU-side training and GEMV ("Train (GPU)"), and data transfer from the main memory ("Transfer (Main Mem)"). CPU-GEMV's latency is governed by the slower of the two parallel tasks shown in side-by-side bars: the GPU-side training, and the CPU-side GEMV ("GEMV (CPU)") plus the

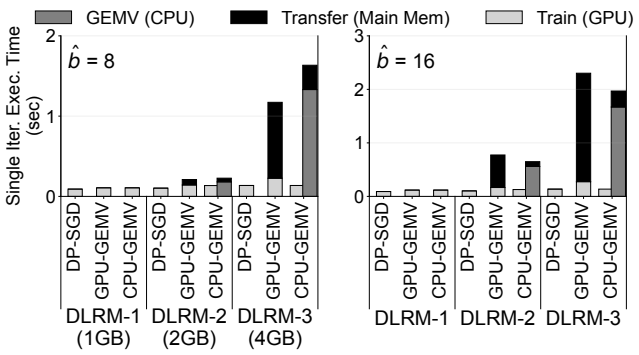


Figure 4: Training time breakdown for DLRM.

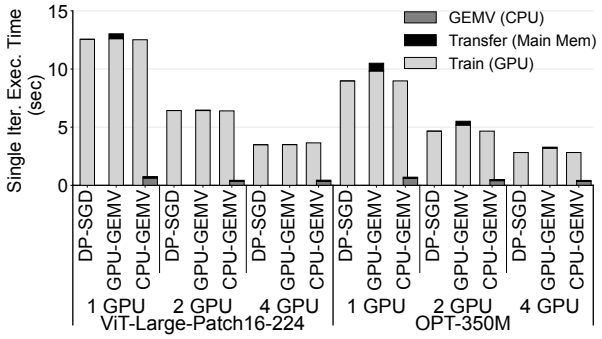


Figure 5: Training time breakdown when the noise history entirely fits into main memory.

transfer of the GEMV result to the GPU ("Transfer (Main Mem)").

In general, GPU-GEMV is better when the model and/or $\hat{b}$ is small, and CPU-GEMV outperforms when they are larger. However, except for uninteresting cases where the entire noise history fits into GPU (DLRM-1), both baselines still incurred non-negligible slowdown. Even the better baseline between the two incurred **2.03–8.62**× ($\hat{b}$=8) and **6.28–14.49**× ($\hat{b}$=16) slowdown. This was due to the noise-related overheads growing much more linearly with m compared to the training time, which grows sub-linearly with m (Section 2.2).

Takeaway 3: For DLRM, correlated noises incur significant slowdown compared to using independent noises.

Overheads of non-DLRM. Models other than DLRM experienced similar trends: the exact type mattered less, and their absolute size (m) mattered more. Specifically, the behavior depended on whether the noise history overflowed the main memory and involved secondary memory.

Figure 5 shows the result from training ViT-L and OPT-350M with $\hat{b}$=16. In these setups, the entire noise fits into the main memory. The slowdown was only 0.6–18.2% for GPU-GEMV, and there was negligible slowdown for CPU-GEMV. Other small models (CNNs, ViTs, and sub-billion LLMs) showed similar behaviors, which we omit.

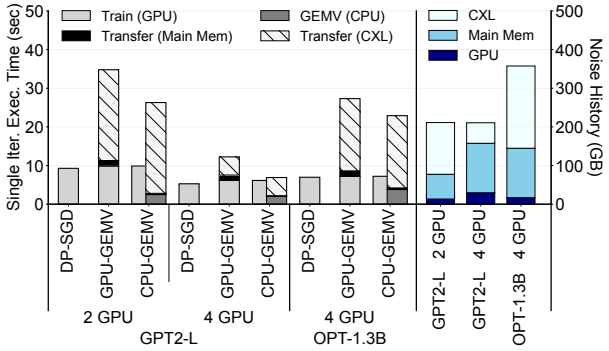


Figure 6: Training time breakdown with noise history partially stored in CXL memory.

When the model size and $\hat{b}$ get larger, noise history must be partially offloaded to secondary memory which incurs a non-negligible slowdown. We only focused on CXL memory due to its relatively better read speed. Slowdown will be higher for slower alternatives like SSD. Figure 6 shows the training time breakdown (left), along with the information on where different portions of the noise history are stored (right). When training GPT2-L with 2 GPUs, 63% of the noise history was placed in CXL memory, leading to **2.83–3.75**× slowdown. When more GPUs are added (GPT2-L with 4 GPUs), some noises could move from CXL memory to GPU memory, improving the slowdown to 1.30–2.31×. With a larger OPT-1.3B, the slowdown again increases to **3.28–3.91**×.

Takeaway 4: For non-DLRM models, the overhead is small if the noise history can fit into main memory. When the noise history overflows main memory, data traffic to secondary memory adds significant latency.

When CPU is highly utilized. We additionally note that CPU-GEMV may incur a larger slowdown when the CPU suffers from resource contention. As an illustration, we trained OPT-350M with $\hat{b} = 64$ on 4 GPUs, and varied the number of cores used by the CPU-side GEMV. We observed that the training starts to slow down if not enough CPU cores can be dedicated to GEMV. With only 4–7% of the cores, the training slowed down by 1.52–2.77×.

Takeaway 5: CPU-GEMV may incur additional overhead when the CPU is highly congested.

4 The Cocoon System

We introduce Cocoon, a framework for efficient DP training with correlated noise. Cocoon stores and processes large noise history across GPU, CPU, and CXL memory device with NMP to maximize performance on an economical setup (Figure 7). For large embedding tables of DLRM, Cocoon provides a dedicated optimization that pre-computes and stores correlated noises in a coalesced format (Section 4.2). When

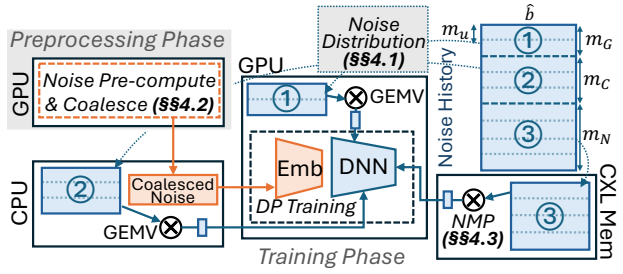


Figure 7: Overview of Cocoon. Cocoon distributes the large noise history across ① GPU, ② CPU, and ③ CXL memory, and performs GEMV near each memory unit to minimize data movement (Section 4.1). Cocoon introduces a preprocessing phase for large embedding tables (Section 4.2), and GEMV inside the CXL memory is done through NMP (Section 4.3).

the CXL memory with NMP exists, Cocoon offloads part of the GEMV to the NMP device to further improve performance (Section 4.3). Cocoon is built on top of Amazon's fastDP [8] and tested on a node with multiple GPUs and an industry prototype CXL NMP device. Cocoon can leverage other NMP devices as long as they support GEMV [46, 87].

Threat model. DP training methods provide guarantees against an adversary who can access (1) the final model and (2) all the intermediate gradients. Cocoon assumes the same adversary except for its embedding table optimization (Section 4.2), where it assumes a weaker-but-practical adversary who can access only the final model and not the intermediate gradients. Assuming an attacker with full knowledge about the intermediate gradients is an artifact of DP-SGD's underlying math and is often considered excessively strong [70]. The weaker-but-practical adversary we assume for the embedding table optimization reflects the most common real-world threat, such as attackers attempting to extract training data from ML services via APIs [68] or through open-sourced weights [9, 10], where intermediate gradients are unavailable. Many other works adopt this same practical adversary model [17, 27, 62, 69, 70, 72, 99].

4.1 Distributing Noise with Profiling

Cocoon splits the $\hat{b} \times m$ noise history in its parameter (m) dimension and distributes slices of size $\hat{b} \times m_G$, $\hat{b} \times m_C$, and $\hat{b} \times m_N$ to GPU, CPU, and NMP device, respectively ($m_G + m_C + m_N = m$). The goal is to minimize the overall training latency while ensuring none of the devices experience an out-of-memory error. Cocoon achieves this through simple profiling, whose cost can be amortized across multiple training iterations.

The goal is solve the following optimization problem:

$$\min_{m_G, m_C, m_N} \quad \max\Big(\mathrm{Lat}_{\text{train-}G} + \mathrm{Lat}_{\text{gemv-}G}(m_G, \hat{b}), \mathrm{Lat}_{\text{gemv-}C}(m_C, \hat{b}), \mathrm{Lat}_{\text{gemv-}N}(m_N, \hat{b})\Big)$$

$$\text{s.t.} \quad m_G + m_C + m_N = m,$$
$$Mem_{\text{total-}G} \geq Mem_{\text{train-}G} + \mathrm{Size}(m_G \times \hat{b}) + Buf,$$
$$Mem_{\text{total-}C} \geq Mem_{\text{train-}C} + \mathrm{Size}(m_C \times \hat{b}),$$
$$Mem_{\text{total-}N} \geq \mathrm{Size}(m_N \times \hat{b}),$$

where $\mathrm{Lat}_{\text{train-}G}$ refers to the single training iteration latency on the GPU, $\mathrm{Lat}_{\text{gemv-}X}(m_X, \hat{b}), X \in \{G, C, N\}$ is the GEMV latency on GPU (G), CPU (C), and NMP (N) with the given noise history, $Mem_{\text{total-}X}, X \in \{G, C, N\}$ refers to each device's total memory capacity, and $Mem_{\text{train-}X}, X \in \{G, C\}$ is the peak memory usage of GPU and CPU for training (NMP is not used for training). Buf is a space reserved in GPU for receiving the GEMV result from CPU and NMP plus avoiding out-of-memory (OOM) due to noise in the estimation, and is a design parameter that depends on the implementation. In short, Cocoon minimizes the overall latency, which is bottlenecked by the slowest among the three devices (GPU training + GEMV, CPU GEMV, and NMP GEMV), while ensuring no out-of-memory.

Finding an optimal split requires accurately estimating each term. $\mathrm{Size}(m_X \times \hat{b})$ and $Mem_{\text{total-}X}$ can be known exactly offline, and $\mathrm{Lat}_{\text{train-}G}$ and $Mem_{\text{train-}X}$ can be upper-bounded by running a few training iterations. $\mathrm{Lat}_{\text{gemv-}X}$ cannot be easily estimated because the GEMV latency is not necessarily linear with the matrix size, especially when the matrix is small. We simplify the optimization goal by defining a minimum unit of allocation, m_u, that perfectly divides m, and choose m_G, m_C, and m_N to be a multiple of m_u. This simplifies the GEMV latency estimation because if m_u is large enough, GEMV becomes memory-bottlenecked and almost linearly scales with the number of m_u processed in each device, *i.e.*, $\mathrm{Lat}_{\text{gemv-}X}(m_X, \hat{b}) \approx \frac{m_X}{m_u} \times \mathrm{Lat}_{\text{gemv-}X}(m_u, \hat{b})$. Measuring $\mathrm{Lat}_{\text{gemv-}X}(m_u, \hat{b})$ for $X \in \{G, C, N\}$ once and plugging the approximation back in the optimization objective gives us a linear objective that can be solved to get m_G, m_C, and m_N. Our implementation also receives GEMV results from CPU and NMP devices to the GPU in m_u-granularity, which means that Buf should be at least $2m_u$ to temporarily hold GEMV results sent from CPU/NMP.

As an example, consider training OPT-1.3B with $\hat{b}$=64 on 4×A5000 GPUs (24GB each, $Mem_{\text{total-}G}$=96GB across 4 GPUs), 128GB DRAM ($Mem_{\text{total-}C}$=128GB), and a 256GB NMP device ($Mem_{\text{total-}N}$=256GB). The noise of the $m \approx$1.4B model is sharded across the GPUs, CPU, and NMP memory. Training takes about 72GB across 4 GPUs ($Mem_{\text{train-}G}$=72GB) and 22GB of CPU DRAM

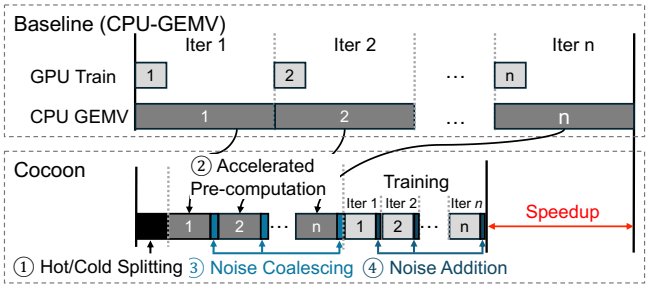


Figure 8: Cocoon's embedding table optimization. Cocoon replaces the per-iteration CPU-GEMV bottleneck (top) with accelerated noise pre-computation (bottom).

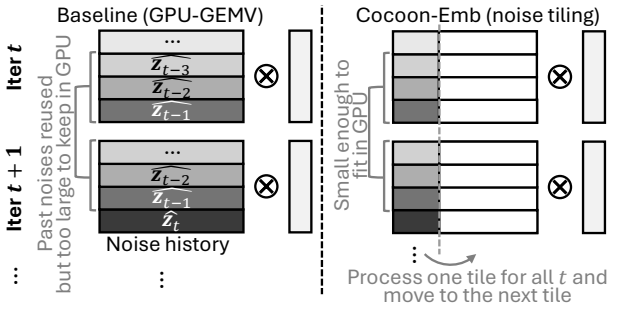


Figure 9: Cocoon's noise tiling for embedding tables. Cocoon (right) tiles the noise history to fit within GPU memory during pre-computation, avoiding the memory spilling required by the baseline (left) when data exceeds GPU capacity.

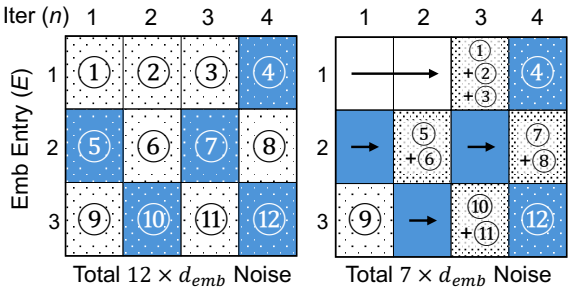


Figure 10: Noise coalescing (d_{emb}: embedding dimension). Cocoon (right) reduces storage overhead (12 to 7 units) by aggregating noise until an entry is accessed (filled blue), replacing the per-iteration baseline (left).

($Mem_{\text{train-}C}$=22GB). If we heuristically set m_G=m_u and Buf=1.5GB per GPU, solving the above optimization problem gives m_u=m_G ≈71M, meaning that m_u divides the parameter m into 20 equal slices. GPUs hold one slice across 4 GPUs, CPU holds 5, and NMP can hold the remaining 14. Varying Buf had little effect on the overall performance, as the GPU-side overhead is mostly due to training, which is independent of Buf. Thus, a reasonable value that would not cause OOM suffices.

4.2 Optimizations for Embedding Tables

Figure 8 summarizes Cocoon's optimization for embedding tables. For embedding tables, Cocoon ① splits the table entries by access frequency into either hot or cold, ② efficiently pre-computes all the correlated noises to be used for the cold entries, ③ coalesces and stores the noises in a compact format, and ④ runs training using the pre-computed noises.

Noise pre-computing. Instead of performing GEMV on each iteration (Figure 8, top), Cocoon *pre-computes* correlated noises for all the future iterations of the embedding tables before the actual training (Figure 8, bottom). The pre-computed results cannot be reused across jobs for privacy, so pre-computing must be done efficiently to not bottleneck the entire training. While pre-computing performs the same amount of GEMV as the baselines, it can be done much faster due to two benefits. Compared to CPU-GEMV, pre-computing can use the faster GPU, which idles before the training starts. Compared to GPU-GEMV, Cocoon maximizes data reuse inside the GPU and minimizes data transfer through PCIe with *noise tiling*.

Figure 9 explains noise tiling. Between consecutive iterations, the most recently updated $\hat{b}-2$ out of $\hat{b}-1$ rows of the noise history are reused (Figure 9, left). However, the reused data ($(\hat{b}-2)\times m$) is often too large to be kept inside the GPU, and GPU-GEMV must spill it to main memory between iterations. Instead, Cocoon splits the noise history into smaller tiles and performs noise pre-computing for each tile, while choosing the tile size so that the reused data always fits inside the GPU (Figure 9, right). After generating all future noises from one tile, Cocoon moves on to the next tile. Noise tiling is only possible during pre-computing, where we can choose to compute noises for all future iterations for one tile before computing the next tile. GPU-GEMV cannot benefit from noise tiling, as it must immediately compute the entire noise (*i.e.*, for all the tiles) before proceeding to the next iteration.

Noise coalescing. Without any optimization, the size of the entire pre-computed noises to be used over n training iterations is $m \times n$, which is too large to store. Cocoon uses a technique called *noise coalescing* to solve this issue. As discussed in Section 2.2, only a few entries in embedding tables are used at each iteration, and unused entries do not contribute to that iteration's gradient. Thus, we do not need to accurately update (*i.e.*, add proper noise) all the entries in every iteration. Instead, it is sufficient to add an equivalent, *aggregated* or *coalesced* noises, as long as they are added before an entry is accessed.

Figure 10 shows a toy example of an embedding table with three entries trained over four iterations. Colored boxes indicate in which iteration each entry is accessed, and dotted

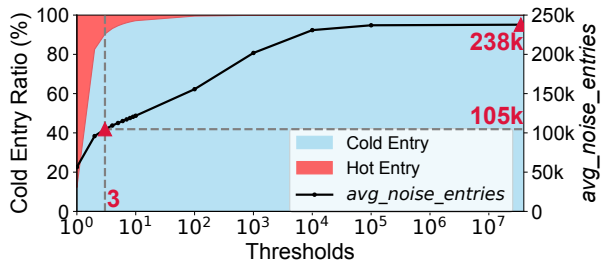


Figure 11: Hot/Cold threshold analysis. Lowering the threshold to 3 reduces *avg_noise_entries* (memory overhead) by 2.3× compared to not using hot/cold splitting.

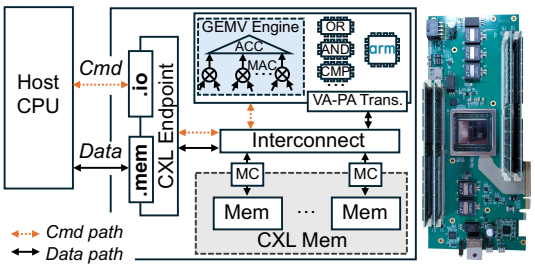


Figure 12: NMP prototype. FPGA-based CXL board featuring an integrated GEMV engine for near-memory acceleration.

numbered boxes indicate when noises are added to each entry's gradient. For example, entry 1 is only accessed in the 4th iteration. Without coalescing (Figure 10, left), noises must be added to all the entries in all the iterations. Instead, noise coalescing (Figure 10, right) only adds an equivalent, aggregated noise right before each entry is accessed or training ends. For example, no noise is added to entry 1 in iterations 1–2, and an equivalent noise (①+②+③) is added at the end of iteration 3. During pre-computing, Cocoon merges and only stores the aggregated noise (*e.g.*, stores ①+②+③ instead of storing three noises separately). In our toy example, only 7 (instead of 12) aggregated noises need to be stored. The benefit is much larger for real models.

Implementing noise coalescing requires knowing exactly when each entry will be accessed during training. This can be known by using a random batch sampler with the same random seed both during pre-computing and training. Cocoon stores the coalesced noise (Figure 10, right) in a compressed sparse column (CSC) format. Cocoon does not pre-compute noises for the rest of the model (*e.g.*, MLP layers) and simply uses GPU-GEMV/CPU-GEMV as they are small.

Hot/cold splitting. The size of the coalesced noise is directly proportional to *avg_noise_entries*, the average number of entries that need noise to be added in each iteration (*avg_noise_entries*=$\frac{7}{4}$ in our example), which is correlated with the average access frequency of each entry. Typically, most entries are scarcely accessed, but a few "hot" entries [71] drive up *avg_noise_entries*. To reduce *avg_noise_entries*, Cocoon classifies each entry as either "hot" or "cold" and only pre-computes and coalesces noise for cold entries. Hot entries, just like MLP layers, rely on CPU-GEMV/GPU-GEMV. As there are usually only a few hot entries [71], the additional overhead is moderate. We use a simple threshold to classify entries as hot or cold based on their access frequency.

Figure 11 illustrates the relationship between the threshold and the *avg_noise_entries* for Criteo Kaggle [47] dataset. The dataset has 39 million samples, and the model used for this dataset has 33 million unique embedding table entries. A lower threshold labels more entries as hot. For example, using 3 as a threshold labels 7% of the entries as hot, lowering *avg_noise_entries* from 238K to 105K (2.3× memory reduction), compared to not using hot/cold splitting. We empirically choose the threshold to balance the memory overhead and additional GEMV overhead.

4.3 Leveraging Near-Memory Processing

When the noise history is too large and must be partially stored in CXL memory, Cocoon leverages a near-memory processing (NMP) device if available to greatly reduce the data movement overhead. In particular, we demonstrate the benefit by adopting an FPGA-based industry prototype CXL memory device with NMP capabilities, provided through our industry partner. The benefit of adopting an NMP device will become larger for cases where the CPU is heavily utilized.

NMP device overview. Figure 12 shows the prototype device from our industry partner. The prototype is implemented as an add-in card (AIC)-type custom board that integrates a CXL controller and an NMP engine into a Xilinx Versal (VP1502) FPGA (Figure 12, right). The board is equipped with DDR4 mounted in DIMM slots. The device receives commands from the CPU via CXL.io and data via CXL.mem, and can act either as normal CXL memory or perform GEMV with data in its memory. The hardware is equipped with MAC and ACC (accumulation) hardware IP, which Cocoon uses to run the desired GEMV. The prototype additionally contains the complete set of basic logic blocks required to compose SQL and ML operators, which Cocoon did not use. The hardware performs a simple virtual-to-physical address translation through saving and looking up memory offset for each matrix and using it to locate them in CXL memory. When there are multiple jobs running on the host, their commands are queued and processed in a first-come-first-served fashion.

Integration to Cocoon. Figure 13 illustrates how correlated noise generation is partially offloaded to the NMP device. The $\hat{b}-1$ past noises have the size of a $(\hat{b}-1)\times m$ noise history matrix. Noise used at step t is stored at the $(t \pmod{(\hat{b}-1)})$-th row, updating the rows in a circular manner (*i.e.,* storing

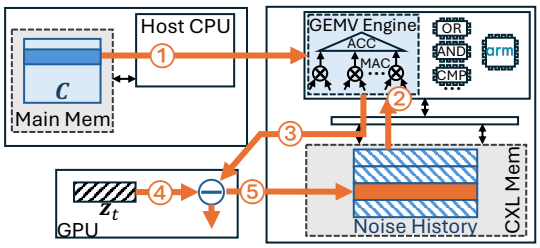


Figure 13: Correlated noise generation with NMP. CPU offloads GEMV to NMP (①-③) with Gaussian noise generation (④) to produce correlated noise that updates the noise history and model (⑤).

noise history in a ring buffer). At each iteration, the CPU ① passes an appropriate mixing vector to the NMP device, ② initiates GEMV between the mixing vector and the noise history, and ③ the GEMV result sent to GPU. ④ the GPU generates Gaussian noise (noise generation can be done by others as well for perfect parallelism, but we use the GPU due to its high throughput). Finally, ⑤ the generated noise updates the noise history and model. These process can be done in parallel while the GPU performs training. Cocoon pre-normalizes the mixing vector ($\mathbf{C}[t,t-\tau]$ in Equation 1) and the Gaussian noise ($\mathbf{z}_t$) by the (t,t)-th entry of $\mathbf{C}$ prior to GEMV to avoid later scaling. As the noise history table is updated in a circular fashion, the mixing vector must also be properly reordered, which is done statically before training.

Cocoon sends the mixing vector once to the NMP device, stores it in an internal buffer, and reuses it m times. For reasonably large models, this amortizes the vector transfer cost. The GEMV operation is implemented with MAC and ACC (accumulation) hardware IP, and the memory bandwidth is maximized through memory-channel interleaving. The prototype device we used achieved 47.9GB/s peak GEMV throughput. The performance may improve in the future with faster DRAM technologies (*e.g.*, DDR5). While the current implementation detail is specific to the prototype we used, we believe similar integration will be possible with other NMP devices [53, 78, 88], as long as they allow GEMV. Coherence is controlled by flushing at the end of every iteration.

5 Evaluation

5.1 Experimental Setup

Hardware. Most of our evaluation was done on a dual-socket Intel Xeon Gold 6330 CPU with 256GB DRAM and 8 NVIDIA RTX A5000 GPUs. When needed, we ran additional experiments on a more powerful dual-socket AMD EPYC 7763 CPU server with 1TB of DRAM and 8 NVIDIA A100 (80GB) GPUs. We additionally used the industry prototype NMP device. Unless noted otherwise, DLRMs were trained on a single GPU without NMP, and LLMs were trained on four GPUs with NMP. CPU-side GEMV used Intel MKL (Intel) and OpenBLAS (AMD) in Pytorch (v.2.4.0).

Due to its early-stage engineering, our NMP prototype exhibited a suboptimal memcpy throughput of 5–7GB/s. This is significantly lower than what a typical CXL memory can achieve (32GB/s for gen5 8x). To *emulate* the end-to-end performance of a mature system, we separately measured the NMP device's overhead and analytically scaled the memcpy overhead assuming a 22GB/s memcpy throughput, which was measured from a similar in-house CXL memory. During training, we run a separate process that adds the estimated NMP device overhead to the training latency, while the rest of the CPU/GPU runs training end-to-end. Although the end-to-end performance is emulated when NMP is involved, except for the memcpy performance, all the overheads were from real measurements. When NMP is not involved, our results are from direct end-to-end measurement. We assume all resources (CPU cores, PCIe bandwidth, CXL memory capacity, *etc.*) are divided evenly across GPUs.

Datasets/models. For DLRM, we used the Criteo Kaggle dataset [47] and the architecture from [71]. We additionally generated synthetic datasets to study the impact of varying the number of embedding entries and data skewness. Synthetic datasets were generated by first ensuring all embedding entries are accessed at least once, and generating the remainder such that the entry accesses follow a Zipfian distribution with a varying α. For non-DLRMs, we used ImageNet [22]-sized dummy data and E2E dataset [73], although the datasets do not impact performance, and architectures from TorchVision/HuggingFace.

Hyperparameters. The training batch size B and band size $\hat{b}$ crucially influence the correlated noise overheads. We used the following values from the literature: $B = 1024$ for vision and language models [6, 8], $B = 65536$ for DLRMs [20], and $\hat{b}$=2–256 [15, 29, 64]. The impact of these hyperparameters is additionally evaluated in the sensitivity studies.

5.2 Performance Improvements: DLRM

5.2.1 End-to-End Training Time

Figure 14 compares the DLRM training time of Cocoon with the baselines. All the bars are normalized to the training time of DP-SGD. For $\hat{b} > 8$, Cocoon consistently outperforms the baselines. Compared to the better baseline, Cocoon improves the overall training time by **2.46–4.87**$\times$ for $\hat{b} > 8$. The speedup generally increases with $\hat{b}$ (2.46$\times$ for $\hat{b} = 16$ and 4.87$\times$ for $\hat{b} = 64$). The breakdown shows that pre-computing dominates the Cocoon-Emb latency.

When $\hat{b} < 8$, the entire noise history fits into the GPU. For these *trivial cases*, the training time of both baselines becomes identical (we only show GPU-GEMV) and close to DP-SGD, and our optimization simply adds unnecessary overheads.

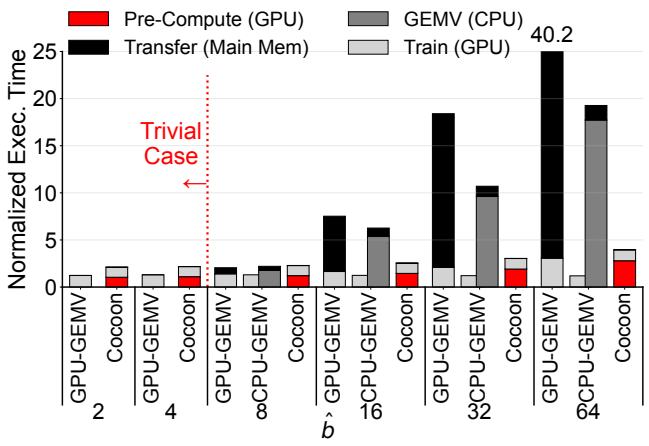


Figure 14: Normalized training time of Cocoon with DLRM. When the entire noise history fits into GPU memory (trivial case; $\hat{b}$=2–4), our optimization can be turned off.

Such trivial cases can be easily detected by comparing the noise history size with the GPU memory capacity, and our optimization can be turned off.

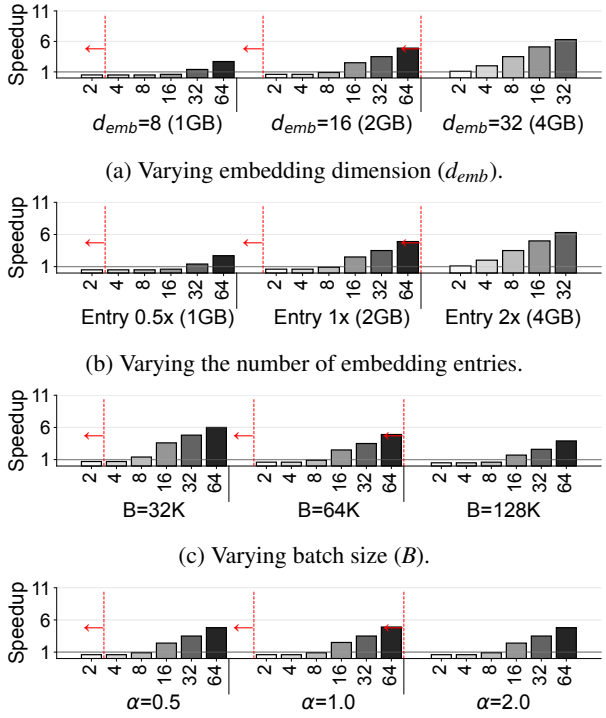
5.2.2 Sensitivity Study

Figure 15 shows the speedup of Cocoon over the better baseline between CPU-GEMV and GPU-GEMV, while varying different dimensions of the model and dataset. Again, bars located at the left side of the red vertical line are trivial cases, and our embedding table optimizations can be turned off.

Model size. Figures 15a and 15b show the speedup of Cocoon while varying the model size, adjusting the embedding dimension (d_{emb}) or the number of embedding entries. The speedup improves with the model size: if we compare the bars at $\hat{b} = 32$, the speedup improves from 3.51× to **6.27-6.35**× when the size is doubled, and reduced to 1.37× when halved. This is because larger models must offload more noise history to main memory and penalize the baselines more severely. Cocoon will become more effective as models grow.

Batch size. Figure 15c shows that the speedup decreases with an increasing batch size. When considering $\hat{b} = 32$, the peak speedup increased from 3.51× with $B = 64K$ to **4.79**× with $B = 32K$, and reduced to 2.57× with $B = 128K$. This is because the correlated noise generation overhead (which Cocoon optimizes) stays the same regardless of the batch size, while the training latency (which Cocoon cannot optimize) becomes larger with bigger B. While not shown, Doubling the entries accessed per sample (*i.e.*, pooling factor [71]) has nearly the same effect as doubling the batch size.

Skewness. Figure 15d shows the speedup of Cocoon when the access frequency of each embedding entry experiences different skewness. The skewness was controlled by varying α of the Zipfian distribution of our synthetic dataset. Interestingly, the skewness had only a minor effect on training time.



(a) Varying embedding dimension (d_{emb}).

(b) Varying the number of embedding entries.

(c) Varying batch size (B).

(d) Varying skewness (larger Zipf α means more skewed).

Figure 15: Speedup of Cocoon under various models and datasets. Numbers below each bar indicates $\hat{b}$.

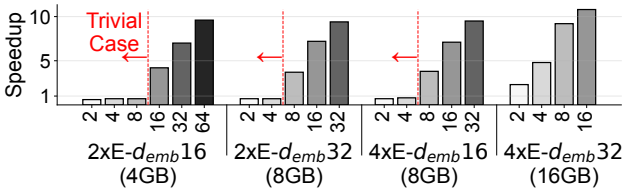


Figure 16: Speedup of Cocoon with various model sizes on an A100 GPU. Numbers below each bar indicates $\hat{b}$.

We will later show that skewness is a critical factor in the memory footprint in Section 5.3.

Hardware. Figure 16 shows the speedup of Cocoon on more powerful A100 GPUs. As A100 has more memory, we used larger models (4–16GB) with a larger number of embedding entries (2–4× larger, denoted as 2×E/4×E) and bigger embedding dimensions (d_{emb}=16–32). Cocoon achieved a speedup of **2.33–10.82**× for non-trivial cases, which is generally larger than the results on the A5000. Cocoon's speedup is larger on A100 because its computation (GPU-side GEMV and training) is much faster than on the A5000, but overheads related to correlated noises (GPU-main memory data transfer, CPU-side GEMV), which Cocoon optimizes, are similar.

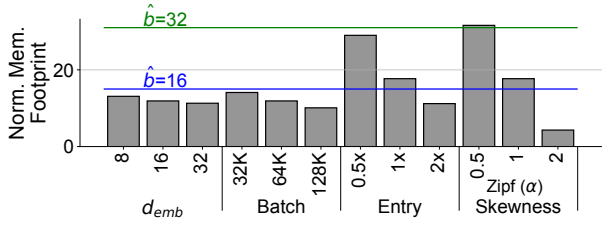


Figure 17: Memory footprint of coalesced noise normalized by the model size. Memory footprint of a noise history without pre-computing for different $\hat{b}$ are in horizontal lines.

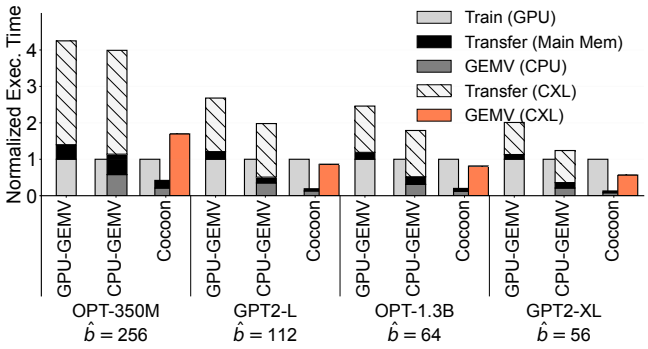


Figure 18: End-to-end normalized training time of Cocoon +NMP and the baselines. Model sizes are in an ascending order. $\hat{b}$ is chosen, so that over 200GB of the noise history is offloaded to CXL memory.

5.3 Memory Overhead: DLRM

5.3.1 Overall Memory Overhead

Unlike CPU-GEMV and GPU-GEMV, which incur a fixed $O(\hat{b}m)$ memory overhead, the overhead of Cocoon with its embedding table optimization depends on the effectiveness of its noise coalescing. In the worst case, Cocoon must hold all the pre-computed noises ($O(nm)$ with n iterations), which can be much larger than that of CPU-GEMV/GPU-GEMV (usually, $n \gg \hat{b}$). However, the actual memory overhead is much less thanks to noise coalescing.

Figure 17 evaluates the memory footprint of the coalesced noise of Cocoon while varying the embedding dimension (d_{emb}), batch size, number of embedding entries, and entry access distribution skewness. The bars are normalized to the model size m. For this figure, we used $n = 1800$ (three epochs using Criteo Kaggle [47] dataset with $B = 64K$), so the worst-case overhead is $1800\times m$. However, the actual memory overhead is only 4.3–31.6×, which is *less* than the memory overhead of the baselines in many cases (shown in horizontal lines for $\hat{b}$=16 and $\hat{b}$=32). The memory overhead of Cocoon is independent of $\hat{b}$ and only depends on the entry access pattern, while the overheads of the baselines grow linearly with $\hat{b}$.

5.3.2 Sensitivity Study

Figure 17 shows how different models and datasets affect the efficacy of noise coalescing. It can be seen that the efficacy decreases with reducing d_{emb} and batch size, but the effect is small. Conversely, decreasing the number of embedding entries and using datasets with less skewed patterns significantly increases the memory overhead. This meets our expectation because noise coalescing works better when batched samples are mostly accessing the same entries, which leads to lower *avg_noise_entries*. Decreasing the number of entries has a similar effect to reduced skewness, because the accesses are hashed into the remaining entries.

Table 1: Hardware cost and power estimate (GPT2-XL, $\hat{b}$=64).

1 HW Unit: 4×NIVIDA A5000 (4×$2K), 1× Intel Xeon Gold 6330 (1×$2.1K), 8×16GB DDR4 3200MHz (8×$0.15K)			
	GPU-Only	CPU-GEMV	**Cocoon**
HW Unit	10	3	1 + NMP [78]
Cost ($)	113K	33.9K	18.3K
Peak Power (KW)	11.6	3.5	1.2
Normalized Throughput	1	0.33	0.13

5.4 Performance Improvement with NMP

5.4.1 End-to-End Training Time

Figure 18 plots the training time and breakdown for the baselines and Cocoon with the NMP hardware prototype, when the models are large enough to involve CXL memory. Cocoon has three bars side-by-side, indicating the GPU-side training, CPU-side GEMV, and the GEMV happening inside the NMP device, all happening in parallel. $\hat{b}$ values are chosen for each model to ensure over 200GB of the noise history is offloaded to CXL memory, to avoid trivial, uninteresting setups.

Figure 18 shows that Cocoon consistently outperforms the baselines, achieving **1.23–2.32**× speedup compared to the better baseline. Cocoon achieves high speedup by eliminating the large data transfer overhead between the CXL memory and CPU/GPU ("Transfer (CXL)"), while incurring a moderate GEMV overhead inside the CXL controller ("GEMV (CXL)"). When the GEMV overhead of Cocoon is less than the training time and can be completely hidden (GPT2-L, OPT-1.3B, GPT2-XL), training becomes the critical path. Our noise distribution strategy minimizes the overall latency, trying to put the minimum noise history on the bottlenecked device.

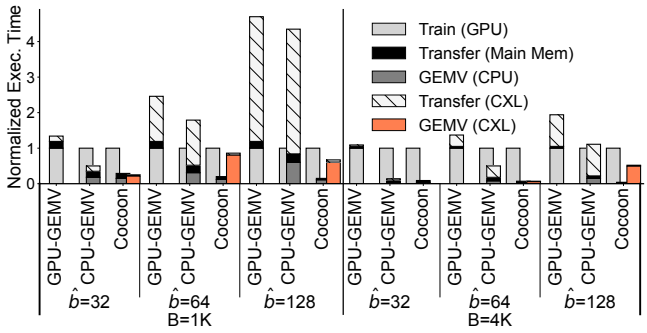


Figure 19: End-to-end normalized training time of Cocoon +NMP and the baselines with OPT-1.3B. $\hat{b} = 128$ number assumes a hypothetical CXL device with an expanded memory capacity, as it does not fit into our current CXL prototype.

5.4.2 Power and Cost

Table 1 compares the estimated hardware cost and peak power of Cocoon with the NMP device against two alternative setups (GPU-Only, CPU-GEMV), which do not involve CXL memory and instead scale out the GPU (GPU-Only) or CPU (CPU-GEMV) to hold the large noise history. GPU-Only represents prior works' setups [63, 79] where enough GPUs hold the entire noise history. We assume GPT2-XL training with $\hat{b}$=64 (∼413GB of noise history). As there can be many different ways to scale out, we simplify the estimation by considering our current single-socket configuration (4xA5000 + 1xIntel Xeon Gold + 128GB memory) as one hardware unit, and estimating how many such units are needed to support noise generation. As our prototype NMP device does not have a public power/cost number, we use numbers from prior work that used a similar device [78]. The device from [78] is more powerful than our prototype, so the power/cost numbers are likely to be an *upper bound*. We do not model the extra cost (*e.g.*, network, power supply) of scaling out beyond a single node. All of these simplistic estimations *favor other baselines*, and Cocoon's actual benefit is likely to be higher.

Table 1 shows that Cocoon can realize training with much less power and hardware cost. This is because instead of using expensive GPU/CPU memory, Cocoon can use CXL NMP memory to partially host and process noise history, which is expected to be much cheaper and lower power than CPU/GPU [78]. However, as the throughput also increases with the hardware units, the baselines also achieve better throughput. If you compare the ratio between the normalized throughput numbers and the cost/power, Cocoon is similar to CPU-GEMV, while being much more cost/power-efficient than GPU-Only. This is because GPU-GEMV introduces too many GPUs, and GPUs can become underutilized. The result indicates that scaling out CPU/GPU is better when high throughput is the main goal, while Cocoon with NMP can be useful if one has a limit on the power/cost budget.

5.4.3 Sensitivity Study

Model size. Figure 18 also shows that smaller models (models to the left) achieve more speedup than larger models (models to the right), when the noise history size inside the CXL memory is similar. This is because the training time ultimately becomes the major bottleneck and limits achievable speedup in larger models.

Band size. Figure 19 plots the end-to-end training time and breakdown for OPT-1.3B while varying $\hat{b}$. When using $\hat{b} = 128$, the noise history is too large to fit into our CXL memory, and we cannot currently run it on our CXL prototype. For $\hat{b} = 128$, we provide an analytical projection assuming a hypothetical future device with expanded memory capacity, but otherwise identical characteristics (GEMV throughput, interconnect bandwidth, *etc.*). This additional datapoint shows how the idea can scale to larger CXL memory devices in the future to support even larger models and band sizes. When $\hat{b}$ is small (*e.g.*, $\hat{b}$=32), Cocoon and CPU-GEMV both perform similarly to DP-SGD, because the training time dominates and all the other overheads are hidden. With larger $\hat{b}$, both the baseline and Cocoon start to incur slowdown, and Cocoon outperforms the best baseline by 2.26×.

Batch size. Figure 19 shows that the speedup decreases when we increase the batch size from B=1K to B=4K. This is because the training time increases with larger batch sizes, and the rest of the overheads can be hidden behind this increased training time. With B=4K, Cocoon showed speedup over the best baseline only when $\hat{b}$=64; otherwise, both Cocoon and the baseline performed on-par. The result indicates that Cocoon's benefit will decrease when using larger batches, but would still show speedup when $\hat{b}$ is large enough.

5.5 Performance Improvement: DLRM+NMP

Our current evaluation only covers the use of NMP with LLMs, not DLRMs. This is because our NMP prototype's limited GEMV throughput was insufficient to benefit DLRM, whose single-iteration latency was on the order of hundreds of milliseconds. Currently, all the GEMV for DLRM (during both pre-compute and actual training) is done on the GPU to operate near this timescale, as detailed in Section 4.2. However, DLRM can also benefit from NMP if its GEMV throughput improves in the future. For example, [78] claimed that building an NMP device with TB/s of internal bandwidth is possible. If such a high bandwidth is indeed possible, we can use the NMP device to accelerate pre-computing of the noise history for the embedding tables. If we assume the throughput from [78], our analytical projection shows that DLRM training would be accelerated by an additional **2.4**×.

6 Related Work

DP-SGD for large models. In the earlier days of DP training research, it was thought that DP training only works well for small models [21]. Recently, many studies have shown that DP training can work well for larger foundation models [4, 7, 8, 11, 19, 40, 43, 61, 64, 98, 104, 105, 112] and DLRMs [18, 20, 23, 30, 62, 72] as well. Google recently released an LLM that was fully trained from scratch with DP [89].

System optimizations for DP training. Several works studied how DP-SGD can be made faster through optimizing the software [6, 8, 58, 95] and hardware [75]. These works mainly focused on efficiently calculating the per-example gradient and are orthogonal to this work. There are also works that studied how to accelerate DP-SGD for DLRMs [30, 62, 72]. LazyDP [62], the closest to our work, also leveraged the fact that one can defer adding noise until an entry is accessed and add an equivalent, aggregated noise. However, [62] relies on the fact that the sum of independent Gaussians is also a Gaussian, and it cannot be extended to (non-independent) correlated noises. Our optimization for DLRM is entirely different from LazyDP. The others [30, 72] modify the DP algorithm itself, affecting privacy and accuracy.

Near/in-memory processing. Near-memory processing (NMP) runs memory-intensive workloads closer to memory. Prior works explored running compute inside the CXL controller (*e.g.*, for LLM inference [35, 78], vector database search [53, 88], and DLRM inference [51, 107]), DIMM (*e.g.*, for DLRM [52, 56, 77] and database operations [52]), network switches [44, 45], and SSD controllers [90, 100]. Processing-in-memory (PIM) embeds compute logic directly in the memory hardware to enjoy even higher bandwidth [42, 57, 59, 76, 106]. Cocoon leverages NMP on a CXL controller for correlated noise generation.

6.1 Applicability to Large-scale Training

While our evaluation focuses on single-node configurations (economical setups), the idea of Cocoon naturally extends to multiple nodes, because correlated noise generation of each model dimension is independent (noise generation for different dimensions can be easily parallelized across nodes). However, if additional nodes provide sufficient aggregate CPU/GPU memory, the relative benefit of the CXL NMP device diminishes. We leave the exploration of these scaled-out setups as future work.

7 Conclusion

DP training with correlated noise is an emerging technique whose system implications have yet to be thoroughly studied. We conducted a systematic study of the new technique and found several major bottlenecks when applied to DLRMs and billion-parameter models on an economical setup. We introduced Cocoon, a framework for efficient DP training with correlated noise. When baseline approaches fail to deliver competitive performance, hardware/software designs of Cocoon can deliver 1.23–10.82× speedup.

Acknowledgments

This work was partly supported by the U.S. National Science Foundation under award No. CNS-2349610 and CCF-2529883. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

References

[1] Martin Abadi, Andy Chu, Ian Goodfellow, H Brendan McMahan, Ilya Mironov, Kunal Talwar, and Li Zhang. Deep learning with differential privacy. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2016.

[2] Abdelrahman Abouelenin, Mohamed Abdelrehim, Raffy Fahim, Amr Hendy, and Mohamed Afify. Privacy-preserving transformers: SwiftKey's differential privacy implementation. In *arxiv.org*, 2025.

[3] John M Abowd. The US Census Bureau adopts differential privacy. In *Proceedings of the ACM SIGKDD international conference on knowledge discovery & data mining*, 2018.

[4] Sebastian Rodriguez Beltran, Marlon Tobaben, Joonas Jälkö, Niki Loppi, and Antti Honkela. Towards efficient and scalable training of differentially private deep learning. In *arxiv.org*, 2024.

[5] Zhiqi Bu, Justin Chiu, Ruixuan Liu, Yu-Xiang Wang, Sheng Zha, and George Karypis. Zero redundancy distributed learning with differential privacy. In *ICLR Workshop on Pitfalls of limited data and computation for Trustworthy ML*, 2023.

[6] Zhiqi Bu, Jialin Mao, and Shiyun Xu. Scalable and efficient training of large convolutional neural networks with differential privacy. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2022.

[7] Zhiqi Bu, Yu-Xiang Wang, Sheng Zha, and George Karypis. Automatic Clipping: Differentially private deep learning made easier and stronger. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2023.

[8] Zhiqi Bu, Yu-Xiang Wang, Sheng Zha, and George Karypis. Differentially private optimization on large

model at small cost. In *Proceedings of the International Conference on Machine Learning (ICML)*, 2023.

[9] Nicholas Carlini, Florian Tramer, Eric Wallace, Matthew Jagielski, Ariel Herbert-Voss, Katherine Lee, Adam Roberts, Tom Brown, Dawn Song, Ulfar Erlingsson, Alina Oprea, and Colin Raffel. Extracting training data from large language models. In *Proceedings of the USENIX Security Symposium*, 2021.

[10] Nicolas Carlini, Jamie Hayes, Milad Nasr, Matthew Jagielski, Vikash Sehwag, Florian Tramer, Borja Balle, Daphne Ippolito, and Eric Wallace. Extracting training data from diffusion models. In *Proceedings of the USENIX Security Symposium*, 2023.

[11] Zachary Charles, Arun Ganesh, Ryan McKenna, H. Brendan McMahan, Nicole Mitchell, Krishna Pillutla, and Keith Rush. Fine-tuning large language models with user-level differential privacy. In *arxiv.org*, 2024.

[12] Jay Chooi, Kevin Cong, Russell Li, and Lillian Sun. DP-AdamW: Investigating decoupled weight decay and bias correction in private deep learning. In *arxiv.org*, 2025.

[13] Christopher A Choquette-Choo, Krishnamurthy Dvijotham, Krishna Pillutla, Arun Ganesh, Thomas Steinke, and Abhradeep Thakurta. Correlated noise provably beats independent noise for differentially private learning. In *arxiv.org*, 2023.

[14] Christopher A. Choquette-Choo, Arun Ganesh, Saminul Haque, Thomas Steinke, and Abhradeep Guha Thakurta. Near-exact privacy amplification for matrix mechanisms. In *Proceedings of the International Conference on Learning Representations (ICLR)*, 2025.

[15] Christopher A Choquette-Choo, Arun Ganesh, Ryan McKenna, H Brendan McMahan, John Rush, Abhradeep Guha Thakurta, and Zheng Xu. (Amplified) Banded Matrix Factorization: A unified approach to private training. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2023.

[16] Christopher A. Choquette-Choo, H. Brendan McMahan, Keith Rush, and Abhradeep Thakurta. Multi-epoch matrix factorization mechanisms for private machine learning. In *Proceedings of the International Conference on Machine Learning (ICML)*, 2023.

[17] Christopher A. Choquette-Choo, Florian Tramèr, Nicholas Carlini, and Nicolas Papernot. Label-only membership inference attacks. In *Proceedings of the International Conference on Machine Learning (ICML)*, 2021.

[18] Lynn Chua, Qiliang Cui, Badih Ghazi, Charlie Harrison, Pritish Kamath, Walid Krichene, Ravi Kumar, Pasin Manurangsi, Nicolas Mayoraz, Hema Venkata Krishna Giri Narra, Steffen Rendle, Amer Sinha, Avinash V. Varadarajan, and Chiyuan Zhang. Training differentially private ad prediction models with semi-sensitive features. In *Proceedings of the Workshop on Data Mining for Online Advertising (AdKDD)*, 2024.

[19] Lynn Chua, Badih Ghazi, Yangsibo Huang, Pritish Kamath, Ravi Kumar, Daogao Liu, Pasin Manurangsi, Amer Sinha, and Chiyuan Zhang. Mind the privacy unit! User-level differential privacy for language model fine-tuning. In *Conference on Language Modeling (COLM)*, 2024.

[20] Lynn Chua, Badih Ghazi, Pritish Kamath, Ravi Kumar, Pasin Manurangsi, Amer Sinha, and Chiyuan Zhang. Scalable dp-sgd: Shuffling vs. poisson subsampling. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2024.

[21] Soham De, Leonard Berrada, Jamie Hayes, Samuel L Smith, and Borja Balle. Unlocking high-accuracy differentially private image classification through scale. In *arxiv.org*, 2022.

[22] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. ImageNet: A large-scale hierarchical image database. In *Proceedings of the Conference on Computer Vision and Pattern Recognition (CVPR)*, 2009.

[23] Carson Denison, Badih Ghazi, Pritish Kamath, Ravi Kumar, Pasin Manurangsi, Krishna Giri Narra, Amer Sinha, Avinash V. Varadarajan, and Chiyuan Zhang. Private ad modeling with DP-SGD. In *Proceedings of the Workshop on Data Mining for Online Advertising (AdKDD)*, 2023.

[24] Alexey Dosovitskiy, Lucas Beyer, Alexander Kolesnikov, Dirk Weissenborn, Xiaohua Zhai, Thomas Unterthiner, Mostafa Dehghani, Matthias Minderer, Georg Heigold, Sylvain Gelly, Jakob Uszkoreit, and Neil Houlsby. An image is worth 16x16 words:Transformers for image recognition at scale. In *Proceedings of the International Conference on Learning Representations (ICLR)*, 2020.

[25] Cynthia Dwork. Differential privacy. In *International Colloquium on Automata, Languages, and Programming (ICALP)*, 2006.

[26] Úlfar Erlingsson, Vasyl Pihur, and Aleksandra Korolova. Rappor: Randomized aggregatable privacy-preserving ordinal response. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2014.

[27] Vitaly Feldman, Ilya Mironov, Kunal Talwar, and Abhradeep Thakurta. Privacy Amplification by Iteration. In *IEEE Annual Symposium on Foundations of Computer Science (FOCS)*, 2018.

[28] Matt Fredrikson, Somesh Jha, and Thomas Ristenpart. Model inversion attacks that exploit confidence information and basic countermeasures. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2015.

[29] Arun Ganesh, Brendan McMahan, and Abhradeep Thakurta. On design principles for private adaptive optimizers. In *arxiv.org*, 2025.

[30] Badih Ghazi, Yangsibo Huang, Pritish Kamath, Ravi Kumar, Pasin Manurangsi, Amer Sinha, and Chiyuan Zhang. Sparsity-preserving differentially private training of large embedding models. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2023.

[31] Antonio Ginart, Laurens van der Maaten, James Zou, and Chuan Guo. Submix: Practical private prediction for large-scale language models. In *arxiv.org*, 2022.

[32] Google. TensorFlow Privacy. 2025.

[33] Filip Granqvist, Congzheng Song, Áine Cahill, Rogier van Dalen, Martin Pelikan, Yi Sheng Chan, Xiaojun Feng, Natarajan Krishnaswami, Vojta Jina, and Mona Chitnis. pfl-research: simulation framework for accelerating research in private federated learning. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2025.

[34] Xin Gu, Gautam Kamath, and Zhiwei Steven Wu. Choosing public datasets for private machine learning via gradient subspace distance. In *IEEE Conference on Secure and Trustworthy Machine Learning (SaTML)*, 2025.

[35] Yufeng Gu, Alireza Khadem, Sumanth Umesh, Ning Liang, Xavier Servot, Onur Mutlu, Ravi Iyer, and Reetuparna Das. PIM is all you need: A CXL-enabled GPU-free system for large language model inference. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operation Systems (ASPLOS)*, 2025.

[36] Chuan Guo, Brian Karrer, Kamalika Chaudhuri, and Laurens Van der Maaten. Bounding training data reconstruction in private (deep) learning. In *Proceedings of the International Conference on Machine Learning (ICML)*, 2022.

[37] Chuan Guo, Alexandre Sablayrolles, and Maziar Sanjabi. Analyzing privacy leakage in machine learning via multiple hypothesis testing: A lesson from fano. In *Proceedings of the International Conference on Machine Learning (ICML)*, 2023.

[38] Udit Gupta, Samuel Hsia, Vikram Saraph, Xiaodong Wang, Brandon Reagen, Gu-Yeon Wei, Hsien-Hsin S. Lee, David Brooks, and Carole-Jean Wu. DeepRecSys: A system for optimizing end-to-end at-scale neural recommendation inference. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, 2020.

[39] Jamie Hayes, Borja Balle, and Saeed Mahloujifar. Bounding training data reconstruction in DP-SGD. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2023.

[40] Jiyan He, Xuechen Li, Da Yu, Huishuai Zhang, Janardhan Kulkarni, Yin Tat Lee, Arturs Backurs, Nenghai Yu, and Jiang Bian. Exploring the limits of differentially private deep learning with group-wise clipping. In *Proceedings of the International Conference on Learning Representations (ICLR)*, 2023.

[41] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep Residual Learning for Image Recognition. In *Proceedings of the Conference on Computer Vision and Pattern Recognition (CVPR)*, 2016.

[42] Yintao He, Haiyu Mao, Christina Giannoula, Mohammad Sadrosadati, Juan Gómez-Luna, Huawei Li, Xiaowei Li, Ying Wang, and Onur Mutlu. PAPI: Exploiting dynamic parallelism in large language model decoding with a processing-in-memory-enabled computing system. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operation Systems (ASPLOS)*, 2025.

[43] Yuzheng Hu, Fan Wu, Ruicheng Xian, Yuhang Liu, Lydia Zakynthinou, Pritish Kamath, Chiyuan Zhang, and David A. Forsyth. Empirical privacy variance. In *Proceedings of the International Conference on Machine Learning (ICML)*, 2025.

[44] Wenqin Huangfu, Krishna T Malladi, Andrew Chang, and Yuan Xie. Beacon: Scalable near-data-processing accelerators for genome analysis near memory pool with the cxl support. In *Proceedings of the International Symposium on Microarchitecture (MICRO)*, 2022.

[45] Pingyi Huo, Anusha Devulapally, Hasan Al Maruf, Minseo Park, Krishnakumar Nair, Meena Arunachalam, Gulsum Gudukbay Akbulut, Mahmut Taylan Kandemir, and Vijaykrishnan Narayanan. Pifs-rec: Process-in-fabric-switch for large-scale recommendation system inferences. In *Proceedings of the International Symposium on Microarchitecture (MICRO)*, 2024.

[46] SK Hynix. CMM-Ax. https://www.linkedin.com/posts/sk-hynix_sk-hynix-cmm-ax-activity-7262426491148222465-sAbH/, 2024.

[47] Kaggle. Criteo Display Advertising Challenge, 2014.

[48] Peter Kairouz, Brendan McMahan, Shuang Song, Om Thakkar, Abhradeep Thakurta, and Zheng Xu. Practical and private (deep) learning without sampling or shuffling. In *Proceedings of the International Conference on Machine Learning (ICML)*, 2021.

[49] Peter Kairouz, Sewoong Oh, and Pramod Viswanath. The composition theorem for differential privacy. In *Proceedings of the International Conference on Machine Learning (ICML)*, 2015.

[50] Sanjay Kariyappa, Chuan Guo, Kiwan Maeng, Wenjie Xiong, G. Edward Suh, Moinuddin K. Qureshi, and Hsien-Hsin S. Lee. Cocktail party attack: Breaking aggregation-based privacy in federated learning using independent component analysis. In *Proceedings of the International Conference on Machine Learning (ICML)*, 2023.

[51] Liu Ke, Udit Gupta, Carole-Jean Wu, Benjamin Youngjae Cho, Mark Hempstead, Brandon Reagen, Xuan Zhang, David Brooks, Vikas Chandra, Utku Diril, Amin Firoozshahian, Kim Hazelwood, Bill Jia, Hsien-Hsin S. Lee, Meng Li, Bert Maher, Dheevatsa Mudigere, Maxim Naumov, Martin Schatz, Mikhail Smelyanskiy, and Xiaodong Wang. Recnmp: Accelerating personalized recommendation with near-memory processing. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, 2020.

[52] Liu Ke, Xuan Zhang, Jinin So, Jong-Geon Lee, Shin-Haeng Kang, Sukhan Lee, Songyi Han, YeonGon Cho, Jin Hyun Kim, Yongsuk Kwon, KyungSoo Kim, Jin Jung, Ilkwon Yun, Sung Joo Park, Hyunsun Park, Joonho Song, Jeonghyeon Cho, Kyomin Sohn, Nam Sung Kim, and Hsien-Hsin S. Lee. Near-memory processing in action: Accelerating personalized recommendation with axdimm. *IEEE Micro*, 2022.

[53] Seoyoung Ko, Hyunjeong Shim, Wanju Doh, Sungmin Yun, Jinin So, Yongsuk Kwon, Sang-Soo Park, Si-Dong Roh, Minyong Yoon, Taeksang Song, and Jung Ho Ahn. COSMOS: A CXL-based full in-memory system for approximate nearest neighbor search. In *IEEE Computer Architecture Letters*, 2025.

[54] Tatsuki Koga, Ruihan Wu, Zhiyuan Zhang, and Kamalika Chaudhuri. Privacy-preserving retrieval-augmented generation with differential privacy. In *arxiv.org*, 2025.

[55] Anastasiia Koloskova, Ryan McKenna, Zachary Charles, John Rush, and H Brendan McMahan. Gradient descent with linearly correlated noise: Theory and applications to differential privacy. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2023.

[56] Youngeun Kwon, Yunjae Lee, and Minsoo Rhu. TensorDIMM: A practical near-memory processing architecture for embeddings and tensor operations in deep learning. In *Proceedings of the International Symposium on Microarchitecture (MICRO)*, 2019.

[57] Hyojung Lee, Daehyeon Baek, Jimyoung Son, Jieun Choi, Kihyo Moon, and Minsung Jang. PAISE: PIM-accelerated inference scheduling engine for transformer-based LLM. In *Proceedings of the International Symposium on High-Performance Computer Architecture (HPCA)*, 2025.

[58] Jaewoo Lee and Daniel Kifer. Scaling up differentially private deep learning with fast per-example gradient clipping. *Proceedings on Privacy Enhancing Technologies*, 2021.

[59] Sukhan Lee, Shin-haeng Kang, Jaehoon Lee, Hyeonsu Kim, Eojin Lee, Seungwoo Seo, Hosang Yoon, Seungwon Lee, Kyounghwan Lim, Hyunsung Shin, Jinhyun Kim, O Seongil, Anand Iyer, David Wang, Kyomin Sohn, and Nam Sung Kim. Hardware architecture and software stack for PIM based on commercial DRAM technology : Industrial product. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, 2021.

[60] Kif Leswing. Facebook says Apple iOS privacy change will result in $10 billion revenue hit this year. https://www.cnbc.com/2022/02/02/facebook-says-apple-ios-privacy-change-will-cost-10-billion-this-year.html, 2022.

[61] Xuechen Li, Florian Tramèr, Percy Liang, and Tatsunori Hashimoto. Large language models can be strong differentially private learners. In *Proceedings of the International Conference on Learning Representations (ICLR)*, 2022.

[62] Juntaek Lim, Youngeun Kwon, Ranggi Hwang, Kiwan Maeng, Edward Suh, and Minsoo Rhu. LazyDP: Co-designing algorithm-software for scalable training of differentially private recommendation models. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operation Systems (ASPLOS)*, 2024.

[63] Ryan McKenna. Scaling up the banded matrix factorization mechanism for differentially private ML. In *arxiv.org*, 2024.

[64] Ryan McKenna, Yangsibo Huang, Amer Sinha, Borja Balle, Zachary Charles, Christopher A. Choquette-Choo, Badih Ghazi, George Kaissis, Ravi Kumar, Ruibo Liu, Da Yu, and Chiyuan Zhang. Scaling laws for differentially private language models. In *arxiv.org*, 2025.

[65] H. Brendan McMahan, Zheng Xu, and Yanxiang Zhang. A hassle-free algorithm for private learning in practice: Don't use tree aggregation, use BLTs. In *arxiv.org*, 2024.

[66] Assaf Hurwitz Michaely, Xuedong Zhang, Gabor Simko, Carolina Parada, and Petar Aleksic. Keyword spotting for Google assistant using contextual speech recognition. In *IEEE Automatic Speech Recognition and Understanding Workshop (ASRU)*, 2017.

[67] Milad Nasr, Saeed Mahloujifar, Xinyu Tang, Prateek Mittal, and Amir Houmansadr. Effectively using public data in privacy preserving machine learning. In *Proceedings of the International Conference on Machine Learning (ICML)*, 2023.

[68] Milad Nasr, Javier Rando, Nicholas Carlini, Jonathan Hayase, Matthew Jagielski, A Feder Cooper, Daphne Ippolito, Christopher A Choquette-Choo, Florian Tramèr, and Katherine Lee. Scalable extraction of training data from aligned, production language models. In *Proceedings of the International Conference on Learning Representations (ICLR)*, 2025.

[69] Milad Nasr, Reza Shokri, and Amir Houmansadr. Comprehensive privacy analysis of deep learning: Passive and active white-box inference attacks against centralized and federated learning. In *IEEE Symposium on Security and Privacy (SP)*, 2019.

[70] Milad Nasr, Shuang Songi, Abhradeep Thakurta, Nicolas Papernot, and Nicholas Carlin. Adversary instantiation: Lower bounds for differentially private machine learning. In *IEEE Symposium on Security and Privacy (SP)*, 2021.

[71] Maxim Naumov, Dheevatsa Mudigere, Hao-Jun Michael Shi, Jianyu Huang, Narayanan Sundaraman, Jongsoo Park, Xiaodong Wang, Udit Gupta, Carole-Jean Wu, Alisson G. Azzolini, Dmytro Dzhulgakov, Andrey Mallevich, Ilia Cherniavskii, Yinghai Lu, Raghuraman Krishnamoorthi, Ansha Yu, Volodymyr Kondratenko, Stephanie Pereira, Xianjie Chen, Wenlin Chen, Vijay Rao, Bill Jia, Liang Xiong, and Misha Smelyanskiy. Deep learning recommendation model for personalization and recommendation systems. In *arxiv.org*, 2019.

[72] Lin Ning, Steve Chien, Shuang Song, Mei Chen, Yunqi Xue, and Devora Berlowitz. EANA: Reducing privacy risk on large-scale recommendation models. In *Proceedings of the ACM Conference on Recommender Systems (RecSys)*, 2022.

[73] Jekaterina Novikova, Ondřej Dušek, and Verena Rieser. The E2E dataset: New challenges for end-to-end generation. In *Proceedings of the 18th Annual SIGdial Meeting on Discourse and Dialogue*, 2017.

[74] Nicolas Papernot, Abhradeep Thakurta, Shuang Song, Steve Chien, and Úlfar Erlingsson. Tempered sigmoid activations for deep learning with differential privacy. In *Proceedings of the AAAI Conference on Artificial Intelligence*, 2021.

[75] Beomsik Park, Ranggi Hwang, Dongho Yoon, Yoonhyuk Choi, and Minsoo Rhu. DiVa: An accelerator for differentially private machine learning. In *Proceedings of the International Symposium on Microarchitecture (MICRO)*, 2022.

[76] Jaehyun Park, Jaewan Choi, Kwanhee Kyung, Michael Jaemin Kim, Yongsuk Kwon, Nam Sung Kim, and Jung Ho Ahn. AttAcc! unleashing the power of pim for batched transformer-based generative model inference. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operation Systems (ASPLOS)*, 2024.

[77] Jaehyun Park, Byeongho Kim, Sungmin Yun, Eojin Lee, Minsoo Rhu, and Jung Ho Ahn. TRiM: Enhancing processor-memory interfaces with scalable tensor reduction in memory. In *Proceedings of the International Symposium on Microarchitecture (MICRO)*, 2021.

[78] Sang-Soo Park, KyungSoo Kim, Jinin So, Jin Jung, Jonggeon Lee, Kyoungwan Woo, Nayeon Kim, Younghyun Lee, Hyungyo Kim, Yongsuk Kwon, Jinhyun Kim, Jieun Lee, YeonGon Cho, Yongmin Tai, Jeonghyeon Cho, Hoyoung Song, Jung Ho Ahn, and Nam Sung Kim. An LPDDR-based CXL-PNM platform for TCO-efficient inference of transformer-based large language models. In *Proceedings of the International Symposium on High-Performance Computer Architecture (HPCA)*, 2024.

[79] Krishna Pillutla, Jalaj Upadhyay, Christopher A. Choquette-Choo, Krishnamurthy Dvijotham, Arun Ganesh, Monika Henzinger, Jonathan Katz, Ryan McKenna, H. Brendan McMahan, Keith Rush, Thomas Steinke, and Abhradeep Thakurta. Correlated noise mechanisms for differentially private learning. In *arxiv.org*, 2025.

[80] Natalia Ponomareva, Sergei Vassilvitskii, Zheng Xu, Brendan McMahan, Alexey Kurakin, and Chiyaun Zhang. How to DP-fy ML: A practical tutorial to machine learning with differential privacy. In *Proceedings of the 29th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*, 2023.

[81] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. Language models are unsupervised multitask learners. *OpenAI blog*, 2019.

[82] Rahul Raguram, Andrew M. White, Yi Xu, Jan-Michael Frahm, Pierre Georgel, and Fabian Monrose. On the privacy risks of virtual keyboards: Automatic reconstruction of typed input from compromising reflections. *IEEE Transactions on Dependable and Secure Computing*, 2013.

[83] Samyam Rajbhandari, Jeff Rasley, Olatunji Ruwase, and Yuxiong He. ZeRO: Memory optimizations toward training trillion parameter models. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2020.

[84] Samyam Rajbhandari, Olatunji Ruwase, Jeff Rasley, Shaden Smith, and Yuxiong He. ZeRO-Infinity: Breaking the GPU memory wall for extreme scale deep learning. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2021.

[85] Jie Ren, Samyam Rajbhandari, Reza Yazdani Aminabadi, Olatunji Ruwase, Shuangyan Yang, Minjia Zhang, Dong Li, and Yuxiong He. ZeRO-Offload: Democratizing billion-scale model training. In *USENIX Annual Technical Conference (USENIX ATC)*, 2021.

[86] Apple Machine Learning Research. Understanding aggregate trends for Apple intelligence using differential privacy. https://machinelearning.apple.com/research/differential-privacy-aggregate-trends, 2025.

[87] Samsung. Near memory solutions for the AI era. https://semiconductor.samsung.com/news-events/tech-blog/near-memory-solutions-for-the-ai-era/, 2023.

[88] Joonseop Sim, Soohong Ahn, Taeyoung Ahn, Seungyong Lee, Myunghyun Rhee, Jooyoung Kim, Kwangsik Shin, Donguk Moon, Euiseok Kim, and Kyoung Park. Computational CXL-memory solution for accelerating memory-intensive applications. *IEEE Computer Architecture Letters*, 2022.

[89] Amer Sinha, Thomas Mesnard, Ryan McKenna, Daogao Liu, Christopher A. Choquette-Choo, Yangsibo Huang, Da Yu, George Kaissis, Zachary Charles, Ruibo Liu, Lynn Chua, Pritish Kamath, Pasin Manurangsi, Steve He, Chiyuan Zhang, Badih Ghazi, Borja De Balle Pigem, Prem Eruvbetine, Tris Warkentin, Armand Joulin, and Ravi Kumar. VaultGemma: A differentially private Gemma model. In *arxiv.org*, 2025.

[90] Xuan Sun, Hu Wan, Qiao Li, Chia-Lin Yang, Tei-Wei Kuo, and Chun Jason Xue. RM-SSD: In-storage computing for large-scale recommendation inference. In *Proceedings of the International Symposium on High-Performance Computer Architecture (HPCA)*, 2022.

[91] Jun Tang, Aleksandra Korolova, Xiaolong Bai, Xueqiang Wang, and Xiaofeng Wang. Privacy loss in Apple's implementation of differential privacy on MacOS 10.12. In *arxiv.org*, 2017.

[92] Qiaoyue Tang and Mathias Lécuyer. DP-Adam: Correcting DP bias in Adam's second moment estimation. In *arxiv.org*, 2023.

[93] Qiaoyue Tang, Frederick Shpilevskiy, and Mathias Lécuyer. DP-AdamBC: Your DP-Adam is actually DP-SGD (unless you apply bias correction). In *Proceedings of the AAAI Conference on Artificial Intelligence*, 2024.

[94] Xinyu Tang, Richard Shin, Huseyin A Inan, Andre Manoel, Fatemehsadat Mireshghallah, Zinan Lin, Sivakanth Gopi, Janardhan Kulkarni, and Robert Sim. Privacy-preserving in-context learning with differentially private few-shot generation. In *Proceedings of the International Conference on Learning Representations (ICLR)*, 2024.

[95] David Tastuggine and Ilya Mironov. Introducing Opacus: A high-speed library for training pytorch models with differential privacy. https://ai.meta.com/blog/introducing-opacus-a-high-speed-library-for-training-pytorch-models-with-differential-privacy/, 2020.

[96] Florian Tramer and Dan Boneh. Differentially private learning needs better features (or much more data). In *arxiv.org*, 2020.

[97] Laurens van der Maaten and Awni Hannun. The trade-offs of private prediction. In *arxiv.org*, 2020.

[98] Liangyu Wang, Junxiao Wang, Jie Ren, Zihang Xiang, David E. Keyes, and Di Wang. FlashDP: Memory-efficient and high-throughput DP-SGD training for large language models. In *NeurIPS 2024 Workshop on Adaptive Foundation Models*, 2024.

[99] Lauren Watson, Chuan Guo, Graham Cormode, and Alexandre Sablayrolles. On the importance of difficulty calibration in membership inference attacks. In *Proceedings of the International Conference on Learning Representations (ICLR)*, 2022.

[100] Mark Wilkening, Udit Gupta, Samuel Hsia, Caroline Trippel, Carole-Jean Wu, David Brooks, and Gu-Yeon Wei. RecSSD: Near data processing for solid state drive based recommendation inference. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operation Systems (ASPLOS)*, 2021.

[101] Zheng Xu, Yanxiang Zhang, Galen Andrew, Christopher A. Choquette-Choo, Peter Kairouz, H. Brendan McMahan, Jesse Rosenstock, and Yuanbo Zhang. Federated learning of Gboard language models with differential privacy. In *The Annual Meeting of the Association for Computational Linguistics: Industry Track (ACL)*, 2023.

[102] Ruoxing Yang. DP-Adam-AC: Privacy-preserving fine-tuning of localizable language models using Adam optimization with adaptive clipping. In *arxiv.org*, 2025.

[103] Da Yu, Peter Kairouz, Sewoong Oh, and Zheng Xu. Privacy-preserving instructions for aligning large language models. In *Proceedings of the International Conference on Machine Learning (ICML)*, 2024.

[104] Da Yu, Saurabh Naik, Arturs Backurs, Sivakanth Gopi, Huseyin A. Inan, Gautam Kamath, Janardhan Kulkarni, Yin Tat Lee, Andre Manoel, Lukas Wutschitz, Sergey Yekhanin, and Huishuai Zhang. Differentially private fine-tuning of language models. In *Proceedings of the International Conference on Learning Representations (ICLR)*, 2022.

[105] Yaodong Yu, Maziar Sanjabi, Yi Ma, Kamalika Chaudhuri, and Chuan Guo. ViP: A differentially private foundation model for computer vision. In *Proceedings of the International Conference on Machine Learning (ICML)*, 2024.

[106] Sungmin Yun, Kwanhee Kyung, Juhwan Cho, Jaewan Choi, Jongmin Kim, Byeongho Kim, Sukhan Lee, Kyomin Sohn, and Jung Ho Ahn. Duplex: A device for large language models with mixture of experts, grouped query attention, and continuous batching. In *Proceedings of the International Symposium on Microarchitecture (MICRO)*, 2024.

[107] Sungmin Yun, Hwayong Nam, Kwanhee Kyung, Jaehyun Park, Byeongho Kim, Yongsuk Kwon, Eojin Lee, and Jung Ho Ahn. CLAY: CXL-based scalable NDP architecture accelerating embedding layers. In *Proceedings of the International Conference on Supercomputing (ICS)*, 2024.

[108] Angela Zhang, Lei Xing, James Zou, and Joseph C Wu. Shifting machine learning for healthcare from development to deployment and from models to data. *Nature Biomedical Engineering*, 2022.

[109] Jiaojiao Zhang, Linglingzhi Zhu, Dominik Fay, and Mikael Johansson. Locally differentially private online federated learning with correlated noise. In *IEEE Transactions Signal Processing*, 2025.

[110] Liang Zhang, Bingcong Li, Kiran Koshy Thekumparampil, Sewoong Oh, and Niao He. DPZero: private fine-tuning of language models without backpropagation. In *Proceedings of the International Conference on Machine Learning (ICML)*, 2024.

[111] Susan Zhang, Stephen Roller, Naman Goyal, Mikel Artetxe, Moya Chen, Shuohui Chen, Christopher Dewan, Mona Diab, Xian Li, Xi Victoria Lin, Todor Mihaylov, Myle Ott, Sam Shleifer, Kurt Shuster, Daniel Simig, Punit Singh Koura, Anjali Sridhar, Tianlu Wang, and Luke Zettlemoyer. OPT: Open pre-trained transformer language models. In *arxiv.org*, 2022.

[112] Xinwei Zhang, Zhiqi Bu, Borja Balle, Mingyi Hong, Meisam Razaviyayn, and Vahab Mirrokni. DiSK: Differentially private optimizer with simplified kalman filter for noise reduction. In *Proceedings of the International Conference on Learning Representations (ICLR)*, 2025.

[113] Yuanbo Zhang, Daniel Ramage, Zheng Xu, Yanxiang Zhang, Shumin Zhai, and Peter Kairouz. Private federated learning in Gboard. In *arxiv.org*, 2023.

A Artifact Appendix

Abstract

Our artifact is available on GitHub. The repository contains our implementation of Cocoon, a system for differentially private training with correlated noise, along with scripts to reproduce the experiments in this paper. This codebase extends `fastDP` with noise distribution, generation, CPU offload pipelines, and noise preprocessing for embedding.

Scope

This artifact allows readers to validate the performance claims in the paper. Figures 3, 5, 14, 15 and 18 are scripted runs, and all other figures can be reproduced with appropriate changes. The getting-started script checks correctness and throughput on a single GPU in less than 30 minutes.

Contents

The repository consists of five directories: `fastDP/` contains the core library including the engine and noise worker. `examples/` holds experiment scripts for LLM, DLRM, and image classification. `benchmark/` provides a getting-started script with correctness checks and throughput benchmarks. `pfl/` contains BandMF matrix solvers from pfl-research. `train_utils/` provides shared training utilities.

Hosting

The artifact is available on GitHub (https://github.com/psu-paws/Cocoon), `osdi26` branch. The `main` branch of the GitHub repository will be actively updated.

Requirements

All main experiments were run on a machine with two sockets of 4× NVIDIA RTX A5000 24GB GPUs and an Intel Xeon Gold 6330 CPU with 128 GB DRAM. Parameters must be tuned to match the machine's topology and memory configuration.

ARTIFACT EVALUATED
USENIX
AVAILABLE

ValScope: Value-Semantics-Aware Metamorphic Testing for Detecting Logical Bugs in DBMSs

Li Lin
School of Informatics
Xiamen University
linli1210@stu.xmu.edu.cn

Liehang Chen
School of Informatics
Xiamen University
2320415112@qq.com

Rongxin Wu
School of Informatics
Xiamen University
wurongxin@xmu.edu.cn

Abstract

Database Management Systems (DBMSs) are crucial for data processing in many large-scale applications. However, detecting logical bugs in DBMSs remains challenging, as defining what constitutes a correct query result is inherently difficult. Metamorphic testing (MT) addresses this issue by checking relations between systematically transformed queries. However, existing MT approaches mainly rely on *equivalent* or *set-semantic* relations, and thus fail to detect subtle bugs that preserve the result set while corrupting value semantics, such as faulty aggregation, ordering, or numeric computation.

In this paper, we propose a unified *SQL query approximation* model that integrates set-semantic and value-semantic reasoning. Beyond result set inclusion or equivalence, our model captures how value-level changes affect query correctness. Based on this model, we develop ValScope, which generates and mutates SQL queries using predefined mutators and performs approximation propagation analysis to reason about global semantic effects. We evaluate ValScope on 6 widely used DBMSs and uncover 67 unique logical bugs, many of which were missed by prior approaches. The results show that ValScope substantially broadens the spectrum of detectable logical bugs beyond existing MT techniques.

1 Introduction

Database Management Systems (DBMSs) play a critical role in applications such as online banking and e-commerce [14, 44]. Like other large-scale systems, DBMSs involve complex code logic and a wide range of functionalities, making them prone to bugs [4, 19, 22]. Among these, logical bugs are particularly critical as they silently lead to incorrect query results in DBMSs [23, 34]. To detect logical bugs, existing approaches [12, 17, 21, 34–36, 38–40] generate SQL queries to test DBMSs and check whether the produced results follow the expectations. One of the fundamental technical challenges in this process is to define what constitutes a correct result for a given query, which is a classical problem in software testing—known as the test oracle problem [18]. To address the test oracle problem, many approaches have been proposed, including differential testing [8, 37], oracle-guided synthesis [36] and metamorphic testing (MT) [12, 17, 21, 34–36, 38–40]. Differential testing cannot reveal bugs that manifest only in a single DBMS implementation and is often inapplicable when queries contain dialect-specific features [17]. Similarly, oracle-guided synthesis provides limited coverage because it reasons about only one pivot row at a time, causing it to miss many logical bugs [17]. These limitations have led to the increasing adoption of MT, which overcomes the oracle challenge by establishing expected semantic relations between pairs of systematically transformed queries [17, 27].

Although MT has become the most effective paradigm for detecting logical bugs in DBMSs, existing MT approaches [12, 17, 21, 34–36, 38–40] still struggle to capture many subtle semantic inconsistencies. Current MT approaches can be broadly divided into two categories: those based on **equivalent metamorphic relations** and those based on **approximation metamorphic relations**. The first category, represented by NoREC [34] and TLP [35], defines the MR as strict output equivalence between the original and transformed queries. NoREC transforms an optimizable query into a non-optimizable form to verify that both forms produce identical results, while TLP partitions query predicates into multiple subqueries whose combined outputs are expected to equal the original result. While these equivalence-based frameworks have proven effective in detecting optimizer and predicate-related inconsistencies, they remain insufficient to expose many deeply hidden bugs [17]. This limitation arises because, under a constrained mutation space, the pair of equivalent queries often still share the same buggy operators or functions, ultimately producing identical yet incorrect results [17]. Consequently, such approaches are restricted to surface-level correctness and fail to uncover deeper semantic flaws that preserve structural equivalence yet violate relational semantics internally.

The second category, exemplified by Pinolo [17], generalizes this paradigm through **set-semantic approximation**.

```
CREATE TABLE t1 (c1 INT , c4 INT);
INSERT INTO t1 (c1,  c4) VALUES (3584,  36);
CREATE UNIQUE INDEX idx_t1_pk_c4 ON t1 (c1, c4);

-- Original Query
SELECT t1.c1 AS c1, SUM(DISTINCT t1.c4*2) AS col_1, DENSE_RANK()
OVER (ORDER BY 1 = 1) AS col_1_2 FROM t1 AS t1 WHERE (NOT
REGEXP_LIKE(t1.c1, '[0-9]{3}-[0-9]{2}-[0-9]{4}'))  GROUP BY t1.c1;
--result:   3584 |     72 |         1

-- Mutated Query
SELECT t1.c1 AS c1, SUM(DISTINCT t1.c4) AS col_1, DENSE_RANK()
OVER (ORDER BY 1 = 1) AS col_1_2 FROM t1 AS t1 WHERE (NOT
REGEXP_LIKE(t1.c1, '[0-9]{3}-[0-9]{2}-[0-9]{4}')) GROUP BY t1.c1;
--result:   3584 |     36 |         1          --(expected)
--result:   3584 |     NULL |   1          --(actual)
```

Figure 1: **An example of a logical bug detected by ValScope in MySQL (Bug #119321).**

Instead of requiring exact equality, Pinolo relaxes query predicates to generate over- and under-approximate variants and checks whether the resulting sets satisfy inclusion or containment relations. This relaxation significantly expands the detectable bug space, allowing the detection of subtle semantic inconsistencies that equivalence-based approaches often overlook. However, set-semantic approximation reasons about inclusion or containment relations among result tuples, rather than how tuple values are computed. Operations such as aggregation or arithmetic collapse a (multi-)set of input values into scalar results, whose correctness depends on numeric computation rather than tuple membership. When such value-level computation is corrupted but the underlying tuple set remains unchanged, the resulting outputs can still satisfy the same set relations. Consequently, inconsistencies that arise from value semantics fall outside the expressive scope of set-semantic approximation.

Figure 1 shows a real bug from MySQL (Bug #119321) [2], where the DBMS incorrectly handles `SUM(DISTINCT)` during aggregation. The original query computes `SUM(DISTINCT t1.c4 * 2)` and returns `72`. After applying a semantics-preserving mutation by removing the arithmetic multiplication and evaluating `SUM(DISTINCT t1.c4)` directly, the expected result should be `36`, since both queries operate on the same tuple set and the distinct value remains unchanged. However, MySQL wrongly returns `NULL`, indicating a silent value-level corruption in numeric computation and aggregation. However, set-level approximation approaches Pinolo fail to detect this bug. Although Pinolo's predicate relaxation effectively captures inconsistencies caused by set expansion or contraction, its checking mechanism focuses solely on tuple-set containment. Since this bug arises entirely within the value semantics of aggregation—while the tuple set remains unchanged—no containment relation is violated, causing the error to fall outside Pinolo's detection capability.

A significant portion of logical bugs manifest not through changes to the result set, but through subtle corruption of value semantics in the computed output. To effectively detect these hidden inconsistencies, we introduce the notion of **value-semantic approximation**, which models value-level behavior through the monotonic direction and magnitude of expected changes. This perspective complements set-semantic approximation: when a mutation preserves the tuple set but alters an aggregation or arithmetic expression, the result should follow a predictable monotonic relationship. By seamlessly integrating set-semantic and value-semantic approximation, we establish **a unified SQL query approximation model** capable of reasoning about both structural and value semantics. This unified model allows us to predict how local mutations propagate through the AST and how the final result should change. Any violation of these predicted approximation relations reveals a semantic inconsistency—precisely the class of deeply hidden logical bugs that existing MT frameworks fundamentally cannot capture.

Based on the proposed SQL query approximation model, we design a novel approach, ValScope which detects logical bugs in DBMSs by jointly reasoning about both set-semantic and value-semantic approximation. ValScope follows a generate–mutate–verify paradigm. It first generates syntactically valid and semantically diverse SQL queries and then systematically mutates them according to the defined approximation mutators. For each pair of original and mutated queries, ValScope performs approximation propagation analysis to determine how local semantic mutations (e.g., changing `MAX` to `MIN` or relaxing predicates) affect global query behavior along the SQL abstract syntax tree (AST). Finally, both queries are executed on the target DBMS, and their outputs are compared against the predicted approximation relation. Any violation of the expected set- or value-level relationship indicates a potential logical bug.

We implement our approach as a practical DBMS testing tool and evaluate it on 6 widely-used and extensively-tested DBMSs, MySQL, MariaDB, OceanBase, Percona, PolarDB and TiDB. In total , ValScope find 67 logical bugs, including 23 in MySQL, 10 in MariaDB, 7 in OceanBase, 10 in Percona, 12 in PolarDB and 5 in TiDB. Among these bugs, 57 are confirmed. These results demonstrate the effectiveness of ValScope in finding logical bugs in DBMSs. Overall, we make the following contributions:

- We propose a unified model, SQL query approximation, that combines set-semantic and value-semantic reasoning to detect logical bugs in DBMSs.
- We implement a novel MT framework, ValScope, which systematically generates, mutates, and verifies SQL queries based on the proposed approximation relations, effectively identifying logical bugs in DBMSs.
- We evaluate ValScope on 6 real-world DBMSs. In total, we found 67 logical bugs. To further facilitate research on DBMS testing, we open-source the tool at https://github.com/linli1724647576/ValScope

2 Background

2.1 Database Management Systems and SQL

Database Management Systems. Database Management Systems (DBMSs) are fundamental to modern software ecosystems, offering systematic mechanisms for storing, organizing, and accessing large volumes of structured data. This work focuses on *relational DBMSs*, which manage data according to the relational model [11]. In such systems, data is organized into tables, each containing tuples (records) that represent real-world entities.

SQL. Structured Query Language (SQL) [9] is the standard interface for interacting with relational DBMSs. It provides a unified syntax for defining, manipulating, and querying data, and can be broadly classified into four categories: Data Definition Language (DDL), Data Manipulation Language (DML), Data Query Language (DQL), and Data Control Language (DCL). Among them, DQL, represented primarily by the `SELECT` statement, forms the core of data retrieval in relational DBMSs. The `SELECT` statement supports rich semantics including filtering, grouping, aggregation, and joining across multiple relations, making it the most fundamental yet semantically complex component of SQL. In this work, we focus on detecting logical bugs in DQL, as they constitute the majority of real-world query workloads and are crucial to ensuring the correctness and reliability of DBMS query processing.

2.2 Logical Bugs in DBMSs

Logical Bugs. Logical bugs are one of the most critical types of bugs in DBMSs, silently causing incorrect query results without triggering system crashes [17, 27]. Unlike crash bugs that exhibit obvious failures, logical bugs corrupt query outputs in subtle ways, posing serious risks to data integrity and application reliability.

Existing Logical Bug Detection Approaches. Existing research on detecting logical bugs mainly relies on automated testing techniques. However, designing an effective testing framework is challenging due to the **test oracle problem**—it is difficult to determine the correct result of a complex SQL query for comparison. To address this issue, researchers have explored three main categories of approaches [17]. The first is differential testing [8, 37], which executes the same query across multiple DBMSs and compares their outputs; inconsistencies reveal potential logic flaws, but dialect differences and heterogeneous semantics often limit its applicability. The second, oracle-guided synthesis [36], generates queries expected to return a specific pivot row and reports an error when the row is missing, but it only captures localized issues and fails to expose deeper semantic inconsistencies. The third and most prominent category is metamorphic testing, which establishes expected semantic relations between pairs of systematically transformed queries. This approach enables *oracle-free* verification, offering better scalability and generality. We will provide an in-depth analysis in the following section.

2.3 Metamorphic Testing

In recent years, metamorphic testing (MT) has become the most effective and widely adopted approach for detecting logical bugs in DBMSs [17, 27]. The core idea of MT is to construct multiple SQL statements whose results are expected to satisfy a specific relation, known as a metamorphic relation (MR). When the actual query results violate this relation, it indicates that at least one query triggers a logical bug in the tested DBMS.

Equivalent MR. Traditional MT, such as NoREC [34] and TLP [35], define the metamorphic relation as strict equivalence between the outputs of the original and transformed queries. Although this equivalence-based strategy effectively bypasses the lack of ground truth, it constrains the search space of test cases: the transformations typically preserve all operators, functions, and predicates of the original query, merely changing structural forms. As a result, such approaches are limited to verifying surface-level correctness and often fail to reveal deeper semantic errors—particularly those that do not produce directly inconsistent result sets but still violate relational semantics internally [17].

Approximation MR. To overcome the over-restrictive nature of equivalence-based testing, PINOLO [17] introduces the concept of set-semantic approximation relations, relaxing the equivalence assumption. By expanding or constraining query predicates, it generates over- and under-approximate queries and judges correctness through inclusion or containment relations between result sets. This relaxation enables the detection of a broader spectrum of logical bugs, uncovering deeper semantic inconsistencies that equivalence-based approaches often miss [17].

However, approximation at the set level still overlooks subtle semantic shifts that occur at the value level, such as incorrect aggregations, ordering, or numeric deviations that preserve the same tuple set but alter the meaning of the result. Building on this insight, our work extends MT from set semantics to value semantics. We propose a new class of value-semantic approximation relations that reason about the direction and magnitude of result changes while maintaining structural consistency. By integrating set-semantic approximation with value-semantic approximation, our framework establishes a unified, multi-dimensional criterion for logical bug detection—enabling the discovery of more nuanced and deeply hidden semantic faults that existing MT frameworks cannot capture.

3 SQL Query Approximation

In this section, we introduce a comprehensive notion of SQL Query Approximation, which unifies two complementary per-

spectives of query behavior: the **set-semantic** and **value-semantic** dimensions. The set-semantic dimension characterizes differences in the returned tuple sets, while the value-semantic dimension captures monotonic variations in the computed or aggregated values over those tuples. Together, these two dimensions form a unified framework for expressing and reasoning about semantic consistency, enabling the construction of more expressive test oracles that can detect both set-level and value-level inconsistencies in DBMS behavior.

3.1 Set-Semantic Approximation

We first formalize the notion of approximation at the set level. This relation captures inclusion or containment among query result sets.

Definition 3.1 (Set-Semantic Approximation Relation)**.** Given a database D, let q_1 and q_2 be two SQL queries whose result sets are $R(q_1,D)$ and $R(q_2,D)$, respectively. We say that q_1 is the *set-level under-approximation* of q_2 over D, denoted by $q_1 \preceq^s_D q_2$, if and only if:

$$R(q_1,D) \subseteq R(q_2,D)$$

Conversely, q_1 is the *set-level over-approximation* of q_2 over D, denoted by $q_1 \succeq^s_D q_2$, if and only if:

$$R(q_1,D) \supseteq R(q_2,D)$$

Here, $R(q,D)$ represents the multi-set returned by evaluating query q on database D, and $\subseteq$ and $\supseteq$ denote inclusion and containment relations between two multi-sets.

Intuitively, the set-semantic approximation forms a partial order over queries: $q_1 \preceq^s_D q_2$ means that q_1 produces a narrower or more restrictive result than q_2, while $q_1 \succeq^s_D q_2$ means that q_1 yields a broader or less restrictive result. These two relations are inverses of each other and together define the lattice of set-level approximations.

Example 3.1. Consider a database $D = \{t_1\}$, where $t_1(c_1) = \{-1,0,1\}$. Now consider the following queries:

$$q_1 : \textbf{SELECT } c_1 \textbf{ FROM } t_1 \textbf{ WHERE } c_1 \leq 0$$

$$q_2 : \textbf{SELECT } c_1 \textbf{ FROM } t_1 \textbf{ WHERE TRUE}$$

The corresponding query results are:

$$R(q_1,D) = \{-1,0\}, \quad R(q_2,D) = \{-1,0,1\}$$

Clearly,

$$R(q_1,D) \subseteq R(q_2,D)$$

which establishes the set-semantic approximation relation:

$$q_1 \preceq^s_D q_2$$

Intuitively, q_1 is a restricted version of q_2; relaxing the selection predicate expands the result set and induces a containment relation between their outputs.

3.2 Value-Semantic Approximation

To overcome the limitation of purely set-level inclusion, we extend the approximation relation from the set-level to the value-level. Unlike the set-level relation that focuses on tuple inclusion, the value-semantic relation captures monotonic changes in target columns—columns whose values depend on value-level computation semantics (e.g., aggregation, arithmetic, or other scalar transformations), rather than on tuple membership. This allows the framework to detect logical bugs where queries return identical tuples but diverge in their value semantics.

Definition 3.2 (Value-Semantic Approximation Relation)**.** Given a database D, let q_1 and q_2 be two SQL queries whose result sets are $R(q_1,D)$ and $R(q_2,D)$, respectively. Let $C_t \subseteq \text{Cols}(R(q_1,D)) \cap \text{Cols}(R(q_2,D))$ denote the target columns whose values will be compared. Let G denote the grouping or ordering basis, determined as follows:

If the query contains a `GROUP BY` clause, G corresponds to the group-by keys. Otherwise, G represents a deterministic ordering over non-target columns (e.g., primary key or lexicographic ordering of attributes) to align tuples for comparison.

We say that q_1 is the *value-level over-approximation* of q_2 over D, denoted by $q_1 \succeq^v_D q_2$, if and only if:

$$\forall g \in G^*, \forall c \in C_t,\ V_{q_1}(g,c) \geq V_{q_2}(g,c)$$

where G^* is the set of all comparable tuple groups under G, and $V_q(g,c)$ denotes the value of column c in group g (or tuple position) produced by query q.

Conversely, q_1 is the *value-level under-approximation* of q_2, denoted by $q_1 \preceq^v_D q_2$, if and only if:

$$\forall g \in G^*, \forall c \in C_t,\ V_{q_1}(g,c) \leq V_{q_2}(g,c)$$

This definition unifies two cases: group-wise comparison for aggregation queries, and order-aligned comparison for non-aggregated results. The comparison operators $\geq$ and $\leq$ are interpreted under the total order induced by the SQL data type of column c, following standard SQL comparison semantics.

Intuitively, the *set-semantic approximation* ($\preceq^s_D$) describes inclusion of tuples, while the *value-semantic approximation* ($\preceq^v_D$) reflects monotonicity among the values of corresponding tuples.

Example 3.2. Consider a table $t_1(c_1,c_2)$ as follows:

$t_1 =$

c_2	c_1
A	10
A	20
B	5
B	7

Let the following two queries be defined:

$$q_1 : \textbf{SELECT } c_2,\ \text{MAX}(c_1) \textbf{ FROM } t_1 \textbf{ GROUP BY } c_2$$

q_2 : **SELECT** c_2, MIN(c_1) **FROM** t_1 **GROUP BY** c_2

The results are:

$R(q_1,D) =$

c_2	MAX(c_1)
A	20
B	7

$R(q_2,D) =$

c_2	MIN(c_1)
A	10
B	5

Under the grouping basis $G = \{c_2\}$ and target column $C_t = \{c_1\}$, we have for each $g \in G^* = \{\text{A}, \text{B}\}$:

$$V_{q_1}(g, c_1) \geq V_{q_2}(g, c_1)$$

Hence, $q_1 \succeq^v_D q_2$. Intuitively, both queries return identical group sets (thus $q_1 \equiv^s_D q_2$), but differ monotonically in their value semantics: the aggregated value of q_1 in each group is no smaller than that of q_2.

3.3 Approximation Propagation

The approximation relations introduced in the previous section capture the semantic correspondence between two complete SQL queries by comparing their result sets or value outputs. However, in practical DBMS testing, a mutation usually affects only a local part of the query—for instance, a predicate, an operator, or an aggregation function—rather than the entire query. To describe and reason about such fine-grained changes, we first define the concept of an *approximate mutator*, which serves as the smallest structural unit that can induce a local approximation within a query.

Definition 3.3 (Approximate Mutator)**.** Let q denote an SQL query represented as an abstract syntax tree (AST) and $n \in q$ a node of this tree. An *approximate mutator* μ is an operator that transforms n into a semantically related variant n' such that the two nodes satisfy an approximation relation $n \preceq^\alpha_D n'$ or $n \succeq^\alpha_D n'$, where $\alpha \in \{s, v\}$ indicates the set-semantic or value-semantic level. Each mutator μ is characterized by:

- **Scope:** the AST node type it applies to, such as predicate, arithmetic expression, aggregation function, or subquery.
- **Transformation rule:** the syntactic or semantic change performed on the node (e.g., <→<=, `MAX`→`MIN`, adding a conjunct condition).
- **Approximation type:** the type of approximation induced by the mutation, determined by both the direction and the semantic level of the change. The direction can be either *under-approximation* or *over-approximation*, while the semantic level can be either *set-semantic* approximation or value-semantic approximation.

We denote the application of a mutator as $n' = \mu(n)$, and the resulting local approximation relation as $n \preceq^\alpha_D n'$ (or its inverse).

An approximate mutator represents the atomic semantic deviation within a query that can propagate through its parent operators and affect the overall query result. To understand how such a local change influences the final query output, we extend the discussion from the semantic level of full-query comparison to the structural level of SQL. Specifically, we define the concept of *approximation propagation*, which describes how a local approximation relation established at one node of the query's AST can be transmitted through its parent operators and clauses, thereby determining how a single mutation impacts the overall approximation behavior of the query.

Definition 3.4 (Approximation Propagation)**.** Let D be a database, and let n_1, n_2 denote two semantically comparable nodes (e.g., subqueries, predicates, or expressions) in the SQL AST. These nodes may correspond directly to a mutation site introduced by an approximate mutator ($n_2 = \mu(n_1)$ for some mutator μ), or they may represent ancestor nodes whose semantics are indirectly affected by the propagation of such a local mutation. We use the unified notation $n_1 \preceq^\alpha_D n_2$ to represent an *approximation relation* of type $\alpha \in \{s, v\}$ where s and v correspond to the set-semantic and value-semantic levels, respectively. The relations defined in §3.1 and §3.2 describe query-level approximations between complete queries. In contrast, approximation propagation extends these relations to the structural level, capturing how local approximations between AST nodes can influence or induce approximations at higher layers of the query.

At the higher layers of the query, operators (such as aggregation functions or filtering conditions) are applied to the nodes and affect the overall query's approximation behavior. Formally, each operator op is characterized by two semantic properties. First, a *mapping* ($\alpha_{\text{in}} \to \alpha_{\text{out}}$) specifies how op transforms the approximation level between the set dimension ($\alpha = s$) and the value dimension ($\alpha = v$). Second, a *direction indicator* $\sigma(op) \in \{+1, -1\}$ specifies whether op preserves or reverses the approximation order. Specifically, $\sigma(op) = +1$ indicates that the operator is *order-preserving*, i.e., it maintains the direction of approximation , whereas $\sigma(op) = -1$ indicates that the operator is *order-reversing*, i.e., it flips the approximation direction . Based on these two attributes, the propagation of a local relation $n_1 \preceq^\alpha_D n_2$ through an operator op can be organized into four canonical forms:

- **(Set → Set)**: Let two nodes n_1 and n_2 satisfy a set-semantic approximation $n_1 \preceq^s_D n_2$. When both nodes are uniformly embedded under a set-level operator op_s , the resulting parent nodes $op_s(n_1)$ and $op_s(n_2)$ satisfy:

  $$op_s(n_1) \preceq^{s,\sigma(op_s)}_D op_s(n_2).$$

  Here, $\sigma(op_s) = +1$ for monotone set operators such as `EXISTS` that preserve tuple inclusion, and $\sigma(op_s) = -1$ for antitone operators like `NOT` or `NOT EXISTS` that reverse tuple inclusion.

- **(Set → Value)**: Given $n_1 \preceq_D^{s} n_2$, let f be an aggregation or set-to-value mapping operator. Applying f to both nodes induces a value-semantic approximation:

$$f(n_1) \ \preceq_D^{v,\sigma(f)} \ f(n_2).$$

  Here, $\sigma(f) = +1$ for aggregation functions whose outputs grow monotonically with the input set (e.g., `MAX`, `SUM`, `COUNT`), and $\sigma(f) = -1$ for antitone ones (e.g., `MIN`).

- **(Value → Value)**: Let two value-producing nodes satisfy $n_1 \preceq_D^{v} n_2$. For a scalar operator op_v applied to both nodes, the resulting values satisfy:

$$op_v(n_1) \ \preceq_D^{v,\sigma(op_v)} \ op_v(n_2).$$

  Here, $\sigma(op_v) = +1$ if op_v is order-preserving with respect to the value-semantic approximation (e.g., `x + c` with $c \geq 0$), and $\sigma(op_v) = -1$ if op_v reverses the order (e.g., negation).

- **(Value → Set)**: Let two value expressions satisfy $n_1 \preceq_D^{v} n_2$. If both expressions are consumed by a predicate or filtering operator op_s, the induced result sets satisfy:

$$op_s(n_1) \ \preceq_D^{s,\sigma(op_s)} \ op_s(n_2).$$

  Here, $\sigma(op_s) = +1$ for monotone-increasing predicates (e.g., `x > c`, where larger values of `x` make the condition more likely to hold and thus expand the result set), and $\sigma(op_s) = -1$ for monotone-decreasing ones (e.g., `x < c`, where larger values of `x` make the condition less likely to hold, causing the result set to shrink).

Remark. In this definition, n_1 and n_2 are not restricted to complete queries. They can represent corresponding subqueries, expressions, or predicates within a single query or across two query variants. The relation $\preceq_D^{\sigma}$ thus captures how a local semantic approximation propagates through SQL operators according to their monotonic behavior, bridging the value- and set-level semantics within the same unified framework.

Intuitively, the propagation mechanism provides the semantic bridge between *tuple-level inclusion* and *value-level monotonicity*. Set-level approximations can trigger value changes through monotone o perators, while value level changes can, in turn, alter the query result set when the affected values participate in predicates. This bidirectional propagation enables comprehensive reasoning over multi-layer SQL dependencies.

Example 3.3 (Set → Value Propagation)**.** Consider two queries over a table $t_1(c_1, c_2)$:

$$q_1 : \textbf{SELECT MAX}(c_1) \ \textbf{FROM} \ t_1 \ \textbf{WHERE} \ c_2 < 100$$

$$q_2 : \textbf{SELECT MAX}(c_1) \ \textbf{FROM} \ t_1 \ \textbf{WHERE} \ c_2 < 200$$

In the query structure, let n_1 and n_2 denote the `WHERE` clause nodes of q_1 and q_2, respectively. The condition $c_2 < 100$ in n_1 is stricter than $c_2 < 200$ in n_2, so the rows selected by n_1 form a subset of those selected by n_2. The parent node of these filters is the aggregation operator `MAX`, which is monotone increasing: when more rows are included, the maximum value of c_1 can only increase or remain the same. As a result, the difference at the set level (fewer or more tuples) propagates upward to a difference at the value level (smaller or larger aggregated value).

Intuitively, n_1 and n_2 illustrate how a local change in the filter condition at the set level can influence the aggregated result value, demonstrating the propagation from Set to Value.

Example 3.4 (Value → Set Propagation)**.** Consider two semantically related queries over a table $t_1(c_1)$:

$$q_1 : \textbf{SELECT} * \ \textbf{FROM} \ (\textbf{SELECT MAX}(c_1) \ \textbf{AS} \ x \ \textbf{FROM} \ t_1) \ \textbf{AS} \ \textit{subq} \ \textbf{WHERE} \ x > 100$$

$$q_2 : \textbf{SELECT} * \ \textbf{FROM} \ (\textbf{SELECT MIN}(c_1) \ \textbf{AS} \ x \ \textbf{FROM} \ t_1) \ \textbf{AS} \ \textit{subq} \ \textbf{WHERE} \ x > 100$$

The two queries differ only in the inner aggregation. Let n_1 and n_2 denote the aggregation nodes `MAX`(c_1) and `MIN`(c_1), respectively. Changing `MAX` to `MIN` decreases the derived value x. Since the outer predicate `x > 100` is monotone increasing in x, smaller x values make the condition harder to satisfy, resulting in fewer output tuples. Consequently, the result of q_2 becomes a subset of q_1, showing a typical Value → Set propagation.

These propagation behaviors connect the two approximation dimensions, allowing a single mutation at any AST node (e.g., `MAX`→`MIN`) to yield predictable, analyzable effects on both result structure and result values. The unified propagation model forms the semantic foundation of our framework.

4 Approach

Based on the SQL Query Approximation model proposed in Section 3, we present a novel approach ValScope for detecting logical bugs in DBMSs. Our approach leverages both set-semantic and value-semantic approximations to identify discrepancies in query results that traditional methods often overlook. In the following, we will provide detailed explanations of our approach.

4.1 Overview

We illustrate the overall workflow of our approach in Figure 2. The entire process follows a generate–mutate–verify paradigm designed to uncover logic bugs in DBMSs. In the pre-processing phase, we randomly populate multiple tables in a test database, following standard DBMS random testing practices [36]. This randomized setup provides a diverse and

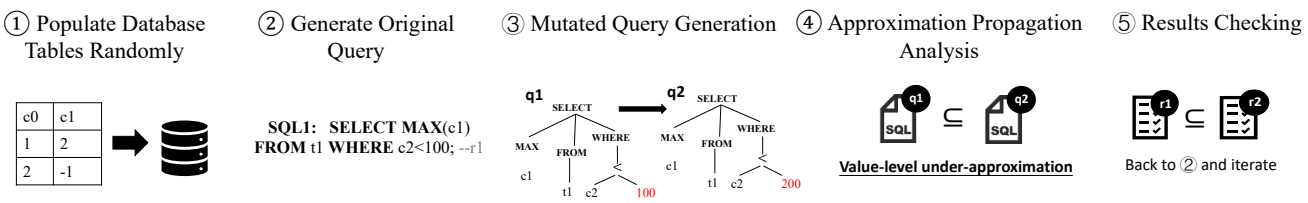


Figure 2: **Overview of our approach.**

unbiased data distribution for subsequent query evaluations. Next, our system generates a syntactically valid SQL query that serves as the original query. It then parses the query and traverses its AST to identify which grammatical constructs can be safely mutated. Based on the SQL query approximation models defined in Section 3, the system automatically synthesizes several approximate queries by mutating the original queries. After the mutated queries are constructed, the framework performs approximation propagation analysis to reason how local semantic changes at the mutated node propagate through the SQL AST. This analysis establishes the global query-level approximation relation (either at the set level or the value level) between the original and mutated queries. Finally, both queries are executed on the tested DBMS instance, and their outputs are compared against the predicted approximation relation. Any violation of this expected relation indicates a potential logical inconsistency in the DBMS.

In the following sections, we introduce each core component in detail, including the construction of the original query (Section 4.2), the design of approximate mutators (Section 4.3), the propagation algorithm (Section 4.4), and the result checking procedure (Section 4.5).

4.2 Construction of the Original Query

To support diverse SQL structures in testing, our framework constructs original queries based on a compact yet expressive SQL grammar, as illustrated in Figure 3. This grammar extends the baseline adopted in PINOLO [17] by incorporating a richer set of syntactic components, enabling the generation of more varied queries. Compared with PINOLO, which mainly focuses on basic `SELECT-FROM-WHERE` statements, our implementation supports a broader range of SQL features, including comprehensive `JOIN` types and conditions, `WITH`-clause based common table expressions (CTEs), and row-level locking modes such as `FOR UPDATE`. It also introduces extended functionalities such as `GROUP BY` with `HAVING` filters and `ORDER BY` clauses, nested function calls with type casting and `CASE` expressions. These extensions collectively allow our system to generate queries that better reflect the syntactic and semantic richness of real-world DBMS workloads.

A key difference between our design and PINOLO lies in the handling of aggregation-related query constructs. PINOLO is built upon set-level approximation relations defined over

Statement	*S*	::=	*Q* | *Q* FOR *LM* | WITH *WI* (, *WI*)* *Q* [FOR *LM*]
Query	*Q*	::=	*QC* [*OB*] [*L*] | *Q SO Q*
SelectCore	*QC*	::=	SELECT [DISTINCT] *SL FC* [*WH*] [*GB*]
SelectList	*SL*	::=	*VE* [AS *id*] (, *VE* [AS *id*])*
FromClause	*FC*	::=	FROM *TR* (, *TR*)*
TableRef	*TR*	::=	*id* [AS *id*] | (*Q*) AS *id*
		|	*TR JTYP* JOIN *TR* [ON *BE*]
JoinType	*JTYP*	::=	INNER | LEFT OUTER | RIGHT OUTER | CROSS
Where	*WH*	::=	WHERE *BE*
GroupBy	*GB*	::=	GROUP BY *CR* (, *CR*)* [HAVING *BE*]
OrderBy	*OB*	::=	ORDER BY *VE* [ASC | DESC] (, *VE* [ASC | DESC])*
Limit	*L*	::=	LIMIT *int*
SetOp	*SO*	::=	UNION | UNION ALL | INTERSECT | EXCEPT
BoolExpr	*BE*	::=	*VE COP VE* | NOT *BE* | *BE* AND *BE* | *BE* OR *BE*
		|	EXISTS (*Q*) | *VE* IS [NOT] NULL
		|	*VE* [NOT] IN (*Q* | *VE* (, *VE*)*)
CompOp	*COP*	::=	= | <> | < | > | <= | >= | LIKE | BETWEEN
ValueExpr	*VE*	::=	CASE WHEN *BE* THEN *VE* ELSE *VE* END | (*Q*)
		|	| *id* | *FCALL* | *CONST*
FuncCall	*FCALL*	::=	*id* ([*VE* (, *VE*)*])
LockMode	*LM*	::=	UPDATE | SHARE | NO KEY UPDATE | KEY SHARE
WithItem	*WI*	::=	*id* AS (*Q*)
ColumnRef	*CR*	::=	*id* [. *id*]
Const	*CONST*	::=	integer, float, string, date, time, timestamp, binary, TRUE, FALSE, NULL

Figure 3: **Compact SQL syntax supported by our approach.**

tuple-level query results, which fundamentally limits its applicability to queries whose outputs preserve monotonic inclusion. As a result, queries involving aggregation—such as those with `GROUP BY` or `HAVING` clauses—are not supported, since aggregation collapses sets of tuples into computed values and breaks the set-level monotonicity assumed by its model. By contrast, our approach introduces value-semantic approximation relations (Section 3) that explicitly capture how aggregated and computed values evolve under localized semantic perturbations, thereby enabling systematic testing of queries with aggregation and grouping.

4.3 Approximate Mutators

To instantiate the SQL Query Approximation model proposed in Section 3, we design a collection of *approximate mutators* that introduce controlled semantic deviations into an SQL query. Each mutator corresponds to a localized modification applied to a specific node of the query's AST. The design goal is to cover both set-level and value-level aspects of SQL semantics, enabling the discovery of a broad range of logical bugs in DBMSs. We have designed a total of 26 mutators,

Table 1: **Representative Approximate Mutators.** Transforming **C1** into **C2** achieves the over-approximation; the inverse achieves the under-approximation.

Type	C1	C2
Set-Semantic Mutators		

Relation	`r1 UNION r2`
	`A EXCEPT B`
	`A INTERSECT B`
		`r1 UNION ALL r2`
		`A`
		`A UNION B`

Predicate	`WHERE cond`
	`col LIKE '_abc%'`
	`a < b`
	`INNER JOIN`
		`WHERE TRUE`
		`col LIKE '%abc%'`
		`a <= b`
		`LEFT JOIN`

Value-Semantic Mutators		

Aggregation	`COUNT(DISTINCT c)`
	`AVG(c)`
	`STDDEV_SAMP(c)`
		`COUNT(c)`
		`MAX(c)`
		`VAR_SAMP(c) (c>1)`

Expression	`expr` `expr`	`expr + k (k > 0)` `expr * k (k has the same sign as expr)`

including 17 set-semantic mutators and 9 value-semantic mutators. Due to space limitations, we only present some representative mutators in Table 1. The complete set of mutators can be accessed in our Github repository [1]. Now we provide more explanations on the approximate mutators.

- **Mutating Relations:** These mutators change how data sets are combined or excluded. For example, replacing `UNION` with `UNION ALL` allows duplicates, while replacing `INTERSECT` with `UNION` alters set inclusion behavior. These mutations affect the breadth of the result set.
- **Mutating Predicates:** These mutators modify the query conditions. For instance, changing a `WHERE` condition to `TRUE` includes all rows, while `FALSE` excludes all. Modifying comparison operators (e.g., < to <=) broadens the results.
- **Mutating Aggregations:** These mutators alter aggregation behavior. For example, switching `COUNT(DISTINCT c)` to `COUNT(c)` affects duplicate handling, and replacing `AVG(c)` with `MAX(c)` changes the nature of the result.
- **Mutating Expressions:** These mutators modify arithmetic or logical expressions, such as adding or multiplying by a constant. These changes impact the computed values and affect the final query output.

4.4 Approximation Propagation Analysis

In this section, we further develop an executable algorithm to determine how local semantic changes propagate through the SQL AST. The main purpose of this algorithm is to formalize the top-down reasoning process introduced in Definition 3.4 into a systematic, bottom-up propagation procedure that connects local node mutations with their global semantic consequences at the query level.

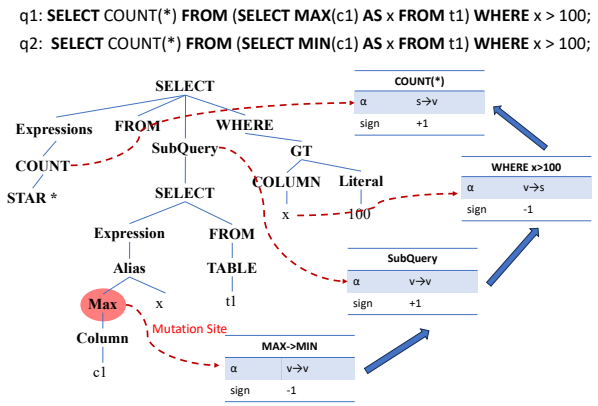


Figure 4: **Example of how the proposed algorithm propagates local semantic changes across the SQL AST.**

Algorithm 1 presents the overall propagation process. Initially (Line 1–3), the algorithm receives the mutated node information $node_info$ and the complete AST of the query. It first identifies the mutated node n_{mut} and constructs its ancestor chain from the mutation site to the query root, represented as list $= [n_{mut}, \ldots, n_{root}]$. This structure enables the algorithm to traverse each parent operator sequentially and reason about how the mutation propagates upward. The algorithm then initializes the local semantic level of the mutation (Line 5–7): if the mutated node involves predicates or subqueries, it starts at the set level ($\alpha(n_{mut}) = s$); otherwise, for expressions or aggregations, it starts at the value level ($\alpha(n_{mut}) = v$). A direction accumulator *sign* is also initialized to $+1$ to indicate that propagation initially preserves directionality. In the propagation stage (Line 9–17), the algorithm iteratively traverses each parent node of n_{mut} in a bottom-up manner. Before propagating, the algorithm invokes $\textsc{DependsOn}(n_i, n_{i-1})$ to check whether the parent node semantically depends on the mutated child node—by referencing its output column (for value-level nodes) or embedding it as a subquery or relation (for set-level nodes). Only dependent nodes are considered for propagation. For each parent operator op_i (Line 9), it consults Table 2 to retrieve the corresponding semantic mapping $(\alpha_{in} \rightarrow \alpha_{out})$ and its monotonic direction $\sigma(op_i) \in \{+1, -1\}$ (Line 13). The semantic level α is updated according to the operator's input-output mapping (e.g., Set→Value for aggregation or Value→Set for predicate filters), while the cumulative direction *sign* is updated multiplicatively ($sign \leftarrow sign \cdot \sigma(op_i)$), preserving or reversing the relation based on operator polarity (Line 14–16). This recursive process effectively tracks the path of semantic transformation from the mutation site to the query output. Finally, at the root node (Lines 19–25), the algorithm materializes the final query-level approximation relation between the original and mutated queries, returning either a set-level or a value-level approximation, depending on the semantic level of the root and the accumulated propagation direction.

Algorithm 1 Approximation Propagation across SQL AST

Input: Mutated node info $node_info$, SQL AST AST
Output: Final query-level approximation between original and mutated queries

// Step 1: Initialization
Identify mutated node n_{mut}.
Build ancestor chain list $= [n_{mut}, \ldots, n_{root}]$.
// Step 2: Local relation at mutation site
Decide initial level $\alpha(n_{mut}) \in \{s, v\}$ by node type.
Set local relation $\mathcal{R}(n_{mut}) \leftarrow (\preceq^{\alpha(n_{mut})})$.
Initialize direction accumulator $sign \leftarrow +1$.
// Step 3: Bottom-up propagation
for each parent node n_i in $list$ (from child to root) **do**
    Determine operator type op_i at n_i.
    if not DEPENDSON(n_i, n_{i-1}) **continue**
    end if
    Lookup rule of op_i in Table 2 to get $(\alpha_{in} \to \alpha_{out}, \sigma(op_i))$.
    Update level: $\alpha(n_{i+1}) \leftarrow \alpha_{out}$.
    Update direction: $sign \leftarrow sign \cdot \sigma(op_i)$.
    Propagate symbolically: $\mathcal{R}(n_{i+1}) \leftarrow \mathcal{R}(n_i)$ with $sign$ applied.
end for
// Step 4: Materialize root-level relation
if $\alpha(n_{root}) = s$
    return $q_1 \preceq^s_D q_2$ with sign $+$,
    or $q_1 \succeq^s_D q_2$ with sign $-$.
else
    return $q_1 \preceq^v_D q_2$ with sign $+$,
    or $q_1 \succeq^v_D q_2$ with sign $-$.
end if

function DEPENDSON(parent, child)
    if $\alpha(child) = v$
        return parent.exprs contains column or alias from child
    else if $\alpha(child) = s$
        return parent.references(child) as subquery or table
    else
        return false
    end if
end function

As illustrated in Figure 4, the mutation occurs at the aggregation node where `MAX(c1)` is replaced by `MIN(c1)`, initializing a value-level relation ($\alpha = v$) with reversed monotonicity ($\sigma = -1$). Following Algorithm 1, the algorithm then traverses its ancestor chain list $= [n_{\text{MAX}}, n_{\text{subq}}, n_{\text{WHERE}}, n_{\text{COUNT}}]$ to evaluate how this local change propagates upward through the query structure. When the propagation reaches the subquery node (`SELECT ... AS x`), the semantics are lifted from the value level to the set level ($\alpha : v \to s$, $\sigma = +1$). According to Table 2, a subquery appearing in the `FROM` clause acts as a data source that preserves tuple inclusion—expanding the underlying relation results in a superset of tuples, thus maintaining positive monotonicity. The subsequent `WHERE x>100` clause applies a filtering operator that preserves the set level but reverses direction ($\alpha : s \to s$, $\sigma = -1$). Finally, the outer `COUNT(*)` operator aggregates the resulting set back to a value-level ($\alpha : s \to v$, $\sigma = +1$). The cumulative sign is therefore $(-1)\times(+1)\times(-1)\times(+1) = +1$, yielding a final value-level approximation $q_1 \preceq^v_D q_2$, which indicates that the output of q_1 forms a value-level under-approximation of q_2.

4.5 Results Checking

The primary purpose of results checking is to verify whether the outputs of the original and mutated SQL queries align with the predicted approximation relations, as defined in the previous section. Unlike traditional methods that rely on strict equivalence or set inclusion checks, which often miss subtle semantic differences, our approach incorporates both set-semantic and value-semantic approximation relations. This allows us to detect a wider range of logical inconsistencies in DBMS query results. In our approach, after executing both queries, we compare their results based on the expected approximation relation (set or value). Specifically, for set-semantic approximation, we check if the result set of the mutated query is a subset or superset of the original query's result set, as predicted by the approximation relation. For value-semantic approximation, we compare the values in the target columns of both queries, ensuring that they satisfy the expected monotonic relationship (either increasing or decreasing). A violation of this relation signals a logical inconsistency in the DBMS's query processing.

5 Implementation

We implemented our approach VALSCOPE as a DBMS testing system, which was written in Python with 13,455 lines of code. The following describes the important implementation details.

Database Population. We adopt the approach PINOLO [17] to generate random database instances from scratch, while also covering most of the data types as specified in the SQL documentation. Unlike existing methods of random database instance generation, VALSCOPE also takes into account type-specific behaviors and boundary cases. For each column, values are generated according to the constraints of the respective data type to ensure validity. This helps prevent common runtime errors such as `INSERT` failures caused by out-of-range or incorrectly formatted values. To increase the likelihood of triggering type-related errors, we also introduce boundary values, including extremely large or small numbers, as well as text with mixed case. Additionally, to test specific arithmetic operations and function-related mutation operators, we explicitly specify the sign (positive or negative) for certain columns.

Test Case Generation and Parsing. Similar to SQL generators like SQLSMITH [5] and GO-RANDGEN [32], we implement VALSCOPE from scratch to generate original queries. VALSCOPE ensures the consistency of data dependencies,

Table 2: **Operator types and semantic propagation rules.**

Operator Type	Semantic Mapping	Monotonic Direction $\sigma(op)$	Example and Interpretation
Aggregation Functions	Set → Value	$+1$: `MAX`, `SUM`, `COUNT` -1: `MIN`	Expanding the input set increases the aggregated result for `MAX`/`SUM`/`COUNT`, or decreases it for `MIN`.
Predicate / Filter	Value → Set	$+1$: $x > c$, $x \geq c$ -1: $x < c$, $x \leq c$	Increasing the compared value makes $x > c$ easier (result set expands) and $x < c$ harder (result set shrinks).
Logical Operator	Set → Set	$+1$: `EXISTS`, `OR`, `AND` -1: `NOT`, `NOT EXISTS`	`OR`/`EXISTS`/`AND` are monotone increasing w.r.t. each input (union/existsintersection); `NOT`/`NOT EXISTS` flip inclusion.
Arithmetic / Expression	Value → Value	Depends on operator sign	Linear arithmetic preserves or reverses direction: `x + k` ($+1$ if $k > 0$), `x * k` ($+1$ if $k > 0$, -1 if $k < 0$), and `-x` flips monotonicity.
Join / Projection / Set Op	Set → Set	$+1$: `JOIN`, `SELECT`, `UNION`, `INTERSECT` mixed: `EXCEPT` (*left* $+1$ / *right* -1)	Structural operators propagate tuple inclusion: `JOIN`, basic `SELECT`, `UNION`, `INTERSECT` are monotone increasing; `EXCEPT` increases with the left input but decreases with the right input.
Subquery / CTE	Context-dependent	$+1$: as `FROM`/CTE source inherits parent: scalar subquery	Monotonicity depends on usage context: (1) In the `FROM` clause (e.g., `SELECT * FROM (SELECT a FROM t)`), it acts as a data source and preserves inclusion ($+1$). (2) As a scalar subquery (e.g., `WHERE x > (SELECT AVG(y) FROM t)`), it inherits the outer predicate's direction ($> \Rightarrow +1$, $< \Rightarrow -1$). (3) For a CTE (e.g., `WITH cte AS (...) SELECT * FROM cte`), it behaves similarly to a `FROM` subquery, typically monotonic ($+1$).

function dependencies, and expression dependencies in query generation. It carefully selects tables and columns based on their usage patterns, applies the correct functions with compatible argument types, and maintains type compatibility across operations, ensuring semantic correctness. This allows ValScope to achieve higher semantic accuracy than existing generators such as SQLSmith and Go-randgen, particularly in handling complex queries with multiple nested subqueries. To apply the approximate mutators to original queries, we use SQLGlot [28], which accepts the same context-free grammar used in seed query generation, to generate ASTs of original queries for mutation.

Bug Report Post-processing. ValScope evaluates queries in the tested DBMS to obtain query results. It is important to note that inconsistent query results often arise during our testing process. To avoid repetitive bug reports and make the test results easier to understand, we first use SQLess [26] to simplify the bugs and pinpoint their root causes. Specifically, SQLess applies a delta-debugging [41] reduction by iteratively removing clauses, expressions, or subqueries from the original query pair, while repeatedly re-executing the reduced queries to check whether the violation still persists. Additionally, we follow the approach from Pinolo [17] to remove duplicates. Concretely, we associate each simplified query pair with the earliest bug-involved version of the tested DBMS in which the violation can be reproduced, and group query pairs that share the same bug-involved version as one bug. Finally, we manually inspect the post-processed results to ensure that each reported bug is minimized and deduplicated before submitting a clear bug report to developers for confirmation.

Supported DBMSs and Adaptation. ValScope is primarily designed around the MySQL syntax, ensuring robust support for MySQL-based databases, such as TiDB and OceanBase. The architecture of ValScope is modular and decoupled, which facilitates easy adaptation to different DBMS dialects. To integrate support for additional DBMS dialects, users only need to modify the relevant components to accommodate the specific syntax, data types, and operations of the target DBMS. This adaptation is similar to extending SQLSmith [5] and can be achieved with just a few hundred lines of code. A more streamlined approach is to leverage the latest extension tool, QTRAN [27], which automates the process of adapting metamorphic-oracle-based logical bug detection techniques for multi-DBMS dialect support.

6 Evaluation

Our evaluation aims to answer the following questions to demonstrate the effectiveness of ValScope:

Q1 Can ValScope find real logical bugs in widely used and extensively tested DBMSs? (Section 6.2)

Q2 Can ValScope find logical bugs that are missed by existing approaches? (Section 6.3)

6.1 Experimental Setup

Tested DBMS. We evaluate ValScope on 6 widely used DBMSs, as summarized in Table 3. These DBMSs were chosen based on their popularity and extensive use in real-world applications. MySQL, MariaDB, TiDB, and OceanBase are widely used, with MySQL ranking second in the DB-Engines ranking, while MariaDB ranks 13th, and TiDB and OceanBase are also prominent players. Percona, a derivative of MySQL, and PolarDB, a cloud-native database, were selected to represent a broader spectrum of DBMSs. All of these DBMSs have undergone extensive testing for both logical [12, 17, 21, 23, 27, 34–36, 38, 40, 42] and crash

bugs [5, 19, 32, 43]. Although finding new bugs in such widely tested systems is challenging, it serves to validate the effectiveness of VALSCOPE. In our experiments, VALSCOPE tests the latest versions of each DBMS. When updates are made to any DBMS, new tests are initiated on the updated versions. The whole experiment takes about one month.

Table 3: **The demographics of the DBMSs under test**

DBMS	Version	GitHub Stars	DB-Engine	First Release
MySQL [29]	9.4.0	11.9K	2	1995
MariaDB [15]	12.0.2	6.7K	13	2009
OceanBase [30]	5.7.25	9.8K	99	2010
Percona [31]	8.0.43-34	1.2K	122	2008
PolarDB [10]	8.0.32	1.6K	105	2017
TiDB [33]	8.0.11	39.4K	85	2016

Environment. We conduct the experiments on one server with 104-cores Intel(R) Xeon(R) Gold 6230R CPU @2.10GHz and 500 GB memory. The server operates under the Ubuntu 20.04 OS, with the 5.4.0-135-generic version of the Linux kernel.

6.2 Bug Detection

Table 4 summarizes the status of the logical bugs found by VALSCOPE for each DBMS. In total, we reported 67 unique logical bugs across six DBMSs, including 23 in MySQL, 10 in MariaDB, 7 in OceanBase, 10 in Percona, 12 in PolarDB, and 5 in TiDB. Notably, all of these bugs were previously unknown, highlighting the effectiveness of VALSCOPE in uncovering hidden logical inconsistencies. Of these, 57 bugs have been confirmed, and 10 are still pending verification.

Bug Importance. Although anecdotal, developer feedback is an important indicator of an approach's effectiveness and the found bugs' importance. Two DBMS companies reached out to us about our testing efforts; both were interested in how we found the bugs, and one invited us to help them find more bugs. For example, a developer from PolarDB commented: "Outstanding work, this tool looks very useful, and we will fully support any technical requirements you may need." Additionally, we received positive feedback on public bug tracking platforms. Among the confirmed bugs, one in MySQL was labeled as a serious bug, indicating it is a high-impact and difficult-to-resolve issue; furthermore, in MariaDB, 10 bugs were labeled as "Major", which represents the highest severity level in the system.

Bug Diversity. We further analyzed the characteristics of the queries that triggered the logical bugs. Among the 57 confirmed bugs, 45 involve subqueries, 17 are related to aggregation functions, and 6 stem from numeric computations within expressions. In addition, 19 bugs are associated with join operations, 22 with `GROUP BY` and `HAVING` clauses. Since a single query may contain multiple constructs, these categories are not mutually exclusive; rather, they show that VALSCOPE can uncover logical bugs arising from rich combinations of subqueries, aggregations, arithmetic expressions, joins, and grouping clauses, instead of being confined to a narrow class of query templates.

Table 4: **Status of logical bugs found by VALSCOPE**

DBMS	Reported	Confirmed	Pending
MySQL	23	23	0
MariaDB	10	10	0
OceanBase	7	6	1
Percona	10	10	0
PolarDB	12	7	5
TiDB	5	1	4
Total	**67**	**57**	**10**

Bug Detected via Approximation Relations. All reported bugs were uncovered through our SQL query approximation framework. Among all confirmed bugs, 43 ultimately violate value-semantic approximation, while 32 require reasoning about cross-dimension propagation between set and value semantics. These results demonstrate the effectiveness of VALSCOPE in jointly modeling set-semantic and value-semantic, enabling the detection of subtle logical bugs that are beyond the reach of prior MT techniques based solely on equivalence or set semantic approximation.

Bug Triggering Rules. To better understand the effectiveness of different mutators, we analyze which mutators are involved in triggering the confirmed logical bugs found by VALSCOPE. Figure 5 shows the top-10 mutators ranked by the number of confirmed bugs they trigger. We observe that the distribution is skewed rather than uniform: a small number of mutators contribute to a large portion of the confirmed bugs. In particular, rule `R1` (removing `DISTINCT`) and `R18` (transforming `COUNT(DISTINCT c)` into `COUNT(c)`) are the most effective, indicating that duplicate handling and aggregation semantics are common sources of confirmed logical bugs in DBMSs. Several other mutators, such as predicate relaxation (`R5`) and join transformation (`R17`), also consistently trigger bugs. At the same time, the fact that multiple mutators contribute to bug discovery suggests that the bug space is diverse, and incorporating a rich set of mutators is necessary to achieve broad coverage. This observation further suggests that extending the mutator set could potentially uncover additional bugs. Due to space limitations, we provide a complete mapping between confirmed bugs and their triggering mutation rules, along with detailed frequency statistics, in our Github repository [25].

Query Generation Efficiency. We evaluate query generation efficiency by measuring the number of test cases generated per second, where each test case consists of an original query and its corresponding mutated query. VALSCOPE achieves query generation throughput comparable to existing methods [5, 13, 32]. Specifically, the generation rates are 2000 queries/sec for MySQL, 1923 for MariaDB, 2000 for OceanBase, 1960 for Percona, 1960 for PolarDB, and 1960 for TiDB.

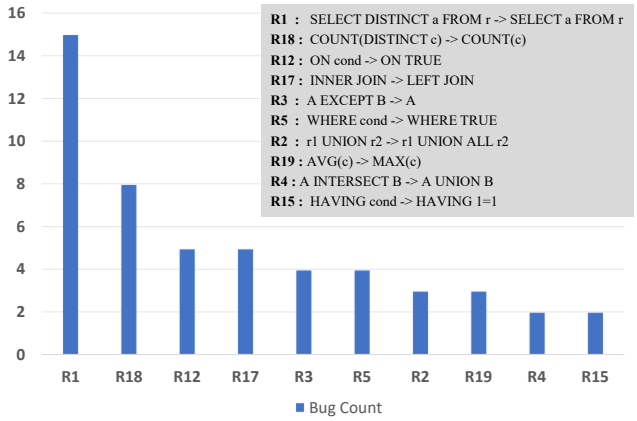


Figure 5: **Top-10 mutators ranked by the number of 57 confirmed logical bugs they trigger.**

In addition to competitive throughput, ValScope achieves high execution success rates across most DBMSs. The execution success rates are 93% for MySQL, 88% for MariaDB, 87% for OceanBase, 91% for Percona, 72% for PolarDB, and 87% for TiDB. These results indicate that ValScope can generate structurally complex queries with nested constructs and inter-query dependencies, while preserving data consistency, functional dependencies, and expression-level correctness. Although the testing campaign lasted for nearly one month, the majority of logical bugs were uncovered at an early stage, with 61 out of the 67 reported bugs (over 90%) detected within the first 24 hours. This observation demonstrates that ValScope is effective at rapidly exposing potential logical flaws, making it well suited for efficient and practical DBMS testing.

Case Study. Figure 6 shows a logical bug in MariaDB [3] found by ValScope, where inconsistent metadata propagation across query branches leads to incorrect result representation. The original query uses `MAX(1)` in the first branch of a `UNION ALL`, while the mutated query replaces it with `AVG(1)`. Under the given database state, both expressions evaluate to the same logical value 1, so the expected result remains `(1), (2025-10-17)`. However, the mutated query instead returns `(0x312E...)` and `(0x32303235...)`—the outputs are silently converted into binary-encoded hex strings. The root cause lies in the implementation of the `AVG()` aggregate function, where the result collation is not properly initialized. Unlike `MAX()` and `MIN()`, which explicitly inherit type and collation metadata from their input via `set_collation`, the `Item_sum_avg` class omits this step, leaving its metadata incomplete. During `UNION ALL` type unification, this missing collation forces the engine to fall back to a binary supertype, leading to incorrect encoding of all result values. The fix restores consistency by adding the missing metadata initialization in `AVG()`, ensuring correct type resolution and human-readable outputs across query branches.

```
-- Original Query
(SELECT MAX(1) AS col FROM t1) UNION ALL
(SELECT CURDATE() AS col);
--result: (1) ,
         (2025-10-17)

-- Mutated Query
(SELECT AVG(1) AS col FROM t1) UNION ALL
(SELECT CURDATE() AS col);

--result: (1) ,                          --- (expected)
         (2025-10-17)
--result: (0x312E3030303030303030) ,     --- (actual)
         (0x323032352D31302D3137)
```

(a) Bug Manifestation: Metadata Erosion and Collation Dissonance in Aggregate-to-Union Projection

```
if (init_sum_func_check(thd))
  return TRUE;

collation= DTCollation_numeric();
decimals=0;
set_maybe_null(sum_func() != COUNT_FUNC);
```

(b) Fix : Integral Restoration of Metadata Attributes in Sum-item Resolution

Figure 6: **MariaDB Bug #37888: unexpected type changing after changing MAX to AVG.**

6.3 Comparative Study

To demonstrate that ValScope detects logical bugs missed by existing methods, we compare ValScope with state-of-the-art DBMS testing approaches from two perspectives: testing tools and timing of bug detection.

Tool Comparison. At the tool level, we selected six state-of-the-art logical bug detection approaches for comparison: PQS [36], NoREC [34], TLP [35], DQP [7], Pinolo [17] and EET [21]. Since the implementation of TQS [40] is not publicly available, and it belongs to the same class of approaches as DQP [7], we include only DQP as a representative for comparison. We run each tool on the latest version of each supported DBMS, with a testing duration of 24 hours. Since the baseline methods do not support all DBMSs tested by ValScope our comparison is limited to the DBMSs commonly supported by ValScope and each baseline method.

Table 5 shows that ValScope outperforms the other tools in detecting logical bugs across all DBMSs, especially in MySQL. Specifically, ValScope found 28,10,20,36,40 and 23 more bugs than PQS, NoREC, TLP, DQP, Pinolo and EET. Note that the increments are calculated based only on the DBMSs commonly supported by both approaches. The other approaches were less effective in our experiments because we tested the latest versions of the DBMSs. The advantage of ValScope mainly comes from its unified reasoning over both set-semantic and value-semantic approximation. In

Table 5: **Number of logical bugs triggered in 24 hours.**

DBMS	PQS	NoREC	TLP	DQP	Pinolo	EET	ValScope
MySQL	0	-	4	0	2	3	21
MariaDB	-	0	-	0	1	-	10
OceanBase	0	-	-	-	0	-	7
Percona	-	-	-	-	-	-	8
PolarDB	-	-	-	-	-	-	10
TiDB	-	-	2	0	0	0	5
Total	0	0	6	0	3	3	61
Increment	28	10	20	36	40	23	

Note."-" indicates that the tool does not support the DBMS. "Increment" is computed on the commonly supported DBMSs between the baseline tool and ValScope.

contrast, prior approaches are limited to equivalence-based checking, set-level approximation, or narrower oracle assumptions, and therefore cannot effectively capture value-level corruptions or cross-dimension semantic violations. We further study whether existing oracles can detect the logical bugs found by ValScope. For each bug-triggering query found by ValScope, we manually construct the corresponding transformed queries required by each baseline oracle and check whether the bug can still be exposed under that testing pattern. Among the 61 logical bugs found by ValScope within 24 hours, Pinolo can detect 12, TLP can detect 4, NoREC can detect 2, and DQP can detect 6, while ValScope uniquely detects 48 of them (78.9%). This result shows that a substantial portion of the bugs found by ValScope are beyond the detection scope of existing oracles, further demonstrating that ValScope complements prior techniques and is particularly effective at exposing logical bugs involving value-level and cross-dimension semantic inconsistencies.

Temporal Comparison. We also present a comparative study from a temporal perspective following EET [21], aiming to evaluate whether ValScope can uncover logical bugs missed by prior approaches. We analyze the earliest bug-involved version of each of the 57 confirmed logical bugs discovered by ValScope, and examine whether these versions predate the publication of existing approaches [7, 12, 17, 21, 34–36, 38–40]. This enables us to assess the effectiveness of ValScope in identifying long-standing issues overlooked by earlier techniques.

As shown in Table 6, among the 57 confirmed bugs, **48** bugs originated before 2020, indicating that all post-2020 approaches (e.g., PQS [36], TLP [35], NoREC [34]) failed to detect these long-standing issues in their extensive evaluations. In addition, **5** bugs were already present between 2020 and 2023, yet were not identified by testing approaches introduced after 2023, such as EET [21], Pinolo [17], TQS [40] and DQP [7]. These findings show that existing approaches indeed missed a substantial number of long-lived logical bugs. These missed bugs can be attributed to fundamental limitations of existing approaches. First, equivalence-based techniques such as TLP [35] apply coarse-grained query-level mutations that reuse the same faulty operators, causing both queries to return identical—yet incorrect—results. Second, approximation-based methods such as Pinolo [17] rely solely on set semantics and therefore cannot detect value-level corruptions that occur without modifying the tuple set. However, ValScope jointly reasons about set- and value-semantic behaviors and performs fine-grained mutations, enabling it to expose subtle errors that all prior approaches fundamentally overlook.

Furthermore, we investigate the longest latency of the confirmed bugs for each DBMS. As reported in Table 6, ValScope identifies at least one bug with a latency exceeding three years in every tested DBMS, with the longest latency reaching **20 years**. These results demonstrate that ValScope is highly effective at discovering long-standing, previously unknown logical bugs across diverse DBMSs.

Table 6: **Latency of the confirmed logical bugs found by ValScope**

DBMS	Found	Bug-involved year		Longest latency
		< 2023	< 2020	
MySQL	23	19	18	20 years
MariaDB	10	10	9	13 years
OceanBase	6	6	3	5 years
Percona	10	10	10	12 years
PolarDB	7	7	7	7 years
TiDB	1	1	1	3 years
Total	**57**	**53**	**48**	**20 years**

7 Discussion

False Positives. False positives in our approach mainly stem from spec-dependent behaviors rather than true logical bugs, including floating-point imprecision, `NULL` semantics, and vendor-specific implementations. For floating-point computations (e.g., in `SUM` or `AVG`), minor numerical deviations may arise due to rounding, execution order, or precision differences; we therefore adopt a tolerance-based comparison, where two values a and b are considered consistent if $|a-b| \le \varepsilon_{\text{abs}}$ or $|a-b|/\max(1,|a|,|b|) \le \varepsilon_{\text{rel}}$. For `NULL` semantics, discrepancies may result from SQL's three-valued logic and special aggregation behavior (e.g., `COUNT(*)` vs. `COUNT(col)`); we address this via `NULL`-aware comparison rules that treat `NULL` as a distinct semantic category and avoid comparing it as a regular scalar. Finally, certain inconsistencies originate from vendor-specific behaviors (e.g., implicit type conversions or function semantics), for which we perform manual validation. These strategies effectively reduce spurious violations, and in practice we did not observe persistent false positives among the confirmed bugs.

Limitations. Our approach relies on value-semantic monotonicity to reason about expected result changes, which limits its applicability to certain classes of SQL queries. In particular, non-monotonic functions (e.g., `SIN`, `ABS`, `ROUND`, or complex `CASE`-based expressions) do not exhibit a consistent

directional behavior, making it difficult to derive reliable approximation relations. For such functions, small input changes may lead to unpredictable output variations, preventing effective value-level reasoning. In addition, queries involving non-deterministic functions (e.g., `RAND()`, `NOW()`) may produce different results across executions, violating the stability assumption required by our checking mechanism. While non-monotonic but deterministic functions are difficult to reason about directly at the function level, they can still be treated as an atomic expression `exp` and tested through monotonic outer transformations such as `exp + 2` or `exp * k`, which still induce predictable value-level changes. In contrast, queries involving non-deterministic functions are currently out of scope, since their results may vary across executions and thus violate the stability assumption required by our checking mechanism.

8 Related Work

Metamorphic Testing for Logical Bug Detection in DBMSs. A variety of MT approaches have been proposed to detect logical bugs in DBMSs [7, 12, 17, 20, 21, 34–36, 38–40]. For example, NoREC [34] transforms an optimizable query into a non-optimizable form and detects semantic inconsistencies by comparing their outputs. TLP [35] divides query predicates into multiple subqueries and verifies that the union of their results is equivalent to the original query. PQS [36] constructs queries expected to retrieve a specific pivot row, while DQE [38] detects logical bugs by comparing whether different SQL queries with the same predicate access the same rows in the database. PINOLO [17] generalizes this paradigm through set-level approximation, where predicates are relaxed or restricted to generate over- and under-approximate queries, and correctness is judged by inclusion or containment relations between result sets.

As discussed in Section 1 and Section 2, existing MT approaches are primarily limited to equivalence-based testing or set-level approximations. In contrast, our work extends MT by jointly modeling set- and value-semantic approximation, enabling the detection of logical bugs that remain undetectable under prior frameworks.

DBMS Test-Case Generation. A variety of approaches have been proposed for generating diverse test cases for DBMSs, with the aim of improving coverage and revealing potential bugs. These techniques [5, 16, 19, 27, 43] typically focus on generating valid and varied SQL queries, but may not specifically target logical bugs. SQLSMITH [5] is a grammar-based DBMS fuzzer that embeds SQL grammar rules to generate complex SQL queries. It uses a random walk approach to explore the SQL syntax and generate a wide range of queries. SQUIRREL [43] is a mutation-based DBMS fuzzer which introduces an intermediate representation for SQL queries and models dependencies between SQL statements, enabling the generation of queries that contain multiple SQL operations. GRIFFIN [16] uses a grammar-free mutation approach, where it mutates SQL queries based on DBMS state information encapsulated in a metadata graph. QTRAN [27] is a LLM-based approach that can automatically translate test cases from other DBMSs. DYNSQL [19] takes a dynamic approach by interacting with the DBMS to capture the latest state information, allowing for the incremental generation of valid and complex queries. These techniques aim to prevent semantic errors and improve the diversity of generated queries. In addition to general query generation, some approaches have been specifically designed to aid in bug detection. SQLRIGHT [24] leverages code coverage feedback to enhance test-case generation. QPG [6] takes a different approach by recording the query plans covered during DBMS testing and prioritizing the mutation of queries that trigger new query plans.

These general query generation approaches complement VALSCOPE. While approaches like SQLSMITH, SQUIRREL, and GRIFFIN focus on generating diverse queries, VALSCOPE can help identify logical bugs hidden in complex or rarely executed query logic. Conversely, the test cases generated by these methods can provide VALSCOPE with high-quality and varied queries, expanding its ability to uncover bugs. Together, these approaches offer a comprehensive strategy for DBMS testing, improving both query coverage and the detection of logical bugs.

9 Conclusion

In this paper, we propose a unified SQL query approximation model that integrates set-semantic and value-semantic reasoning for detecting logical bugs in DBMSs. Based on this model, we develop VALSCOPE, a metamorphic testing framework that mutates SQL queries and propagates semantic approximations to uncover deep logical inconsistencies. Our evaluation on 6 widely used DBMSs discovers 67 logical bugs, 57 of which have been confirmed at the time of submission. We believe the effectiveness of VALSCOPE can inspire more follow-up research on DBMS testing.

Acknowledgements

We thank the anonymous reviewers and shepherd for their valuable and insightful feedback. We also thank the DBMS developers for triaging and fixing our reported bugs. This research was supported by the Natural Science Foundation of China (Grant No. 62272400), the Fujian Provincial Natural Science Foundation of China (Grant No. 2025J010002), and Xinmiao Project of ZheJiang Provincial Applied Basic Research Program (Grant No. 2026XMZD032). Li Lin began this work while affiliated with Xiamen University. Now he is a Ph.D. student at Zhejiang University. Rongxin Wu is the corresponding author and works as a member of Fujian Engineering Research Center of High-Performance Intelligent Computing Systems in Xiamen University.

References

[1] Valscope mutation documentation. https://github.com/linli1724647576/ValScope/blob/main/mutation.md. Accessed Dec 2025.

[2] Mysql bug #119321: Unexpected null after changing sum to sum(x * 2). https://bugs.mysql.com/bug.php?id=119321, 2025. Accessed Dec 2025.

[3] Mdev-37888: Unexpected type changing after changing avg to max. https://jira.mariadb.org/browse/MDEV-37888, 2026. Accessed Mar 2026.

[4] Atul Adya. Weak consistency: a generalized theory and optimistic implementations for distributed transactions. 1999.

[5] Anse1. Sqlsmith: A random sql query generator. https://github.com/anse1/sqlsmith, 2020. Accessed Dec 2025.

[6] Jinsheng Ba and Manuel Rigger. Testing database engines via query plan guidance. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE 23)*, pages 2060–2071. IEEE, 2023.

[7] Jinsheng Ba and Manuel Rigger. Keep it simple: Testing databases via differential query plans. *Proceedings of the ACM on Management of Data*, 2(3):1–26, 2024.

[8] Hardik Bati, Leo Giakoumakis, Steve Herbert, and Aleksandras Surna. A genetic approach for random testing of database systems. In *Proceedings of the 33rd international conference on Very large data bases*, pages 1243–1251, 2007.

[9] Donald D Chamberlin and Raymond F Boyce. Sequel: A structured english query language. In *Proceedings of the 1974 ACM SIGFIDET (now SIGMOD) workshop on Data description, access and control*, pages 249–264, 1974.

[10] Alibaba Cloud. Polardb product overview. https://help.aliyun.com/zh/polardb/product-overview/, 2023. Accessed Dec 2025.

[11] Edgar F Codd. A relational model of data for large shared data banks. *Communications of the ACM*, 13(6):377–387, 1970.

[12] Wenqian Deng, Jie Liang, Zhiyong Wu, Jingzhou Fu, and Yu Jiang. Detecting logic bugs in dbmss via equivalent data construction. *Proceedings of the ACM on Management of Data*, 3(6):1–25, 2025.

[13] SQLancer developers. Sqlancer: Automated testing to find logic and performance bugs in database systems. https://github.com/sqlancer/sqlancer, 2025. Accessed Dec 2025.

[14] Daniela Florescu, Alon Levy, and Alberto Mendelzon. Database techniques for the world-wide web: A survey. *ACM Sigmod Record*, 27(3):59–74, 1998.

[15] MariaDB Foundation. Mariadb, 2023. Accessed Dec 2025.

[16] Jingzhou Fu, Jie Liang, Zhiyong Wu, Mingzhe Wang, and Yu Jiang. Griffin: Grammar-free dbms fuzzing. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering (ASE 22)*, pages 1–12, 2022.

[17] Zongyin Hao, Quanfeng Huang, Chengpeng Wang, Jianfeng Wang, Yushan Zhang, Rongxin Wu, and Charles Zhang. Pinolo: Detecting logical bugs in database management systems with approximate query synthesis. In *2023 USENIX Annual Technical Conference (USENIX ATC 23)*, pages 345–358, 2023.

[18] William E Howden. Theoretical and empirical studies of program testing. *IEEE Transactions on Software Engineering*, (4):293–298, 2006.

[19] Zu-Ming Jiang, Jia-Ju Bai, and Zhendong Su. {DynSQL}: Stateful fuzzing for database management systems with complex and valid {SQL} query generation. In *32nd USENIX Security Symposium (USENIX Security 23)*, pages 4949–4965, 2023.

[20] Zu-Ming Jiang, Si Liu, Manuel Rigger, and Zhendong Su. Detecting transactional bugs in database engines via {graph-based} oracle construction. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*, pages 397–417, 2023.

[21] Zu-Ming Jiang and Zhendong Su. Detecting logic bugs in database engines via equivalent expression transformation. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, pages 821–835, 2024.

[22] Jinho Jung, Hong Hu, Joy Arulraj, Taesoo Kim, and Woonhak Kang. Apollo: Automatic detection and diagnosis of performance regressions in database systems. *Proceedings of the VLDB Endowment*, 13(1):57–70, 2019.

[23] Matteo Kamm, Manuel Rigger, Chengyu Zhang, and Zhendong Su. Testing graph database engines via query partitioning. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 23)*, pages 140–149, 2023.

[24] Yu Liang, Song Liu, and Hong Hu. Detecting logical bugs of {DBMS} with coverage-based guidance. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 4309–4326, 2022.

[25] Li Lin. Valscope: Trigger bug rules. https://github.com/linli1724647576/ValScope/blob/main/trigger%20bug%20rules.md, 2026. Accessed Mar 2026.

[26] Li Lin, Zongyin Hao, Chengpeng Wang, Zhuangda Wang, Rongxin Wu, and Gang Fan. Sqless: Dialect-agnostic sql query simplification. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 24)*, pages 743–754, 2024.

[27] Li Lin, Qinglin Zhu, Hongqiao Chen, Zhuangda Wang, Rongxin Wu, and Xiaoheng Xie. Qtran: Extending metamorphic-oracle based logical bug detection techniques for multiple-dbms dialect support. In *Proceedings of the 34nd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 25)*, pages 731–752, 2025.

[28] Toby Mao. Sqlglot: A no-dependency sql parser, transpiler, optimizer, and engine, 2024. Accessed Dec 2025.

[29] MySQL. Mysql, 2023. Accessed Dec 2025.

[30] OceanBase. Oceanbase open source, 2023. Accessed Dec 2025.

[31] Percona. Percona - open source database software and services, 2023. Accessed Dec 2025.

[32] PingCap. go-randgen, 2022. Accessed Dec 2025.

[33] PingCAP. Tidb, 2023. Accessed Dec 2025.

[34] Manuel Rigger and Zhendong Su. Detecting optimization bugs in database engines via non-optimizing reference engine construction. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 1140–1152, 2020.

[35] Manuel Rigger and Zhendong Su. Finding bugs in database systems via query partitioning. *Proceedings of the ACM on Programming Languages*, 4(OOPSLA):1–30, 2020.

[36] Manuel Rigger and Zhendong Su. Testing database engines via pivoted query synthesis. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 667–682, 2020.

[37] Donald R Slutz. Massive stochastic testing of sql. In *VLDB*, volume 98, pages 618–622, 1998.

[38] Jiansen Song, Wensheng Dou, Ziyu Cui, Qianwang Dai, Wei Wang, Jun Wei, Hua Zhong, and Tao Huang. Testing database systems via differential query execution. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE 23)*, pages 2072–2084. IEEE, 2023.

[39] Jiansen Song, Wensheng Dou, Yu Gao, Ziyu Cui, Yingying Zheng, Dong Wang, Wei Wang, Jun Wei, and Tao Huang. Detecting metadata-related logic bugs in database systems via raw database construction. *Proceedings of the VLDB Endowment*, 17(8):1884–1897, 2024.

[40] Xiu Tang, Sai Wu, Dongxiang Zhang, Feifei Li, and Gang Chen. Detecting logic bugs of join optimizations in dbms. *Proceedings of the ACM on Management of Data*, 1(1):1–26, 2023.

[41] Andreas Zeller. *Why programs fail: a guide to systematic debugging*. Morgan Kaufmann, 2009.

[42] Chi Zhang and Manuel Rigger. Constant optimization driven database system testing. *Proceedings of the ACM on Management of Data*, 3(1):1–24, 2025.

[43] Rui Zhong, Yongheng Chen, Hong Hu, Hangfan Zhang, Wenke Lee, and Dinghao Wu. Squirrel: Testing database management systems with language validity and coverage feedback. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, pages 955–970, 2020.

[44] Xuanhe Zhou, Chengliang Chai, Guoliang Li, and Ji Sun. Database meets artificial intelligence: A survey. *IEEE Transactions on Knowledge and Data Engineering*, 34(3):1096–1116, 2020.

A Artifact Appendix

Abstract

The artifact for VALSCOPE packages the implementation of the SQL generation, seed-query extraction, and mutation-based checking framework described in the paper, together with runnable scripts for six DBMS dialects. It enables users to execute end-to-end testing campaigns, inspect generated SQL workloads and mutation logs, and reproduce representative logical bug reports produced by the system.

Scope

The artifact supports validation of the paper's core systems claims: VALSCOPE combines set-semantic and value-semantic metamorphic testing in one framework, works across multiple MySQL-family DBMSs, and can generate concrete bug evidence including schema setup, original queries, mutated queries, and mismatch or failure logs. The artifact is intended for reproducing the testing workflow and representative bug-finding results rather than re-running every 24-hour campaign reported in the full evaluation.

Contents

The artifact repository contains the VALSCOPE source code, dependency manifest, dialect-specific execution scripts, example generated SQL outputs, mutation logs, and documentation. The main entry points are `main.py` for the core workflow and `scripts/<dialect>/run.sh` for automated evaluation on MySQL, MariaDB, Percona, TiDB, OceanBase, and PolarDB. Generated schemas, queries, and seed queries are written to `generated_sql/`; execution logs are written to `logs/`; and mutation mismatch records are stored under `invalid_mutation/`.

Hosting

The artifact is hosted at https://github.com/linli1724647576/ValScope-AE. The evaluated version is the public `master` branch at commit `ee89ac36840252063e951c5cbb61e430970f5241` (published on May 21, 2026).

Requirements

The artifact requires Python 3.8 or newer and the Python packages listed in `requirements.txt`. The automated evaluation scripts additionally require Docker and a POSIX-compatible shell environment (for example Linux, macOS, or WSL on Windows). The artifact has been prepared and tested with the following DBMS versions: MySQL 9.4.0, MariaDB 12.0.2, Percona Server 8.0.43-34, TiDB 8.0.11, OceanBase 5.7.25, and PolarDB X-Cluster 8.0.32.

Getting Started

Install dependencies with `pip install -r requirements.txt`. A typical short run can be launched with `bash scripts/mysql/run.sh -mutator-type both -run-hours 0.5`. Each script starts the target DBMS, waits for readiness, bootstraps a schema, runs SQL generation and seed-query extraction, and then invokes the value and/or set mutators. Evaluators can inspect the resulting logs and mutation records to verify bug evidence, including mismatching query results and DBMS failures.

The Abstention Protocol: RCA for Clos Fabrics (Operational Systems)

Madhava Gaikwad*
Independent

Deepak Pandey
Microsoft

Abstract

Root cause analysis (RCA) in large datacenter networks is challenging because telemetry is noisy, partial, and asynchronous. Score-based approaches degrade under these conditions, often yielding unstable or incorrect attributions.

We present CORESEC, a production RCA system that replaces weighted fusion with a PAM-style abstention algebra. Telemetry agents are composed using control flags that yield deterministic decisions and explicit abstention when evidence is ambiguous. CoreSec combines this algebra with topology-aware configurations that capture failure surfaces across Clos fabrics and converge monotonically as evidence accumulates.

Deployed at hyperscale, CoreSec provides stable and explainable RCA behavior across diverse environments without retuning. Our experience shows that structured composition with abstention forms a practical foundation for automated RCA in real-world cloud networks.

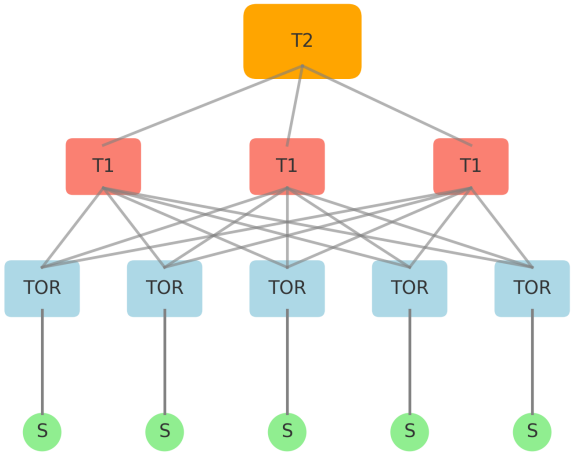


Figure 1: Three-tier Clos topology. Servers connect to TORs, TORs to T1 aggregation switches, and T1s to T2 spines.

1 Introduction

Large cloud networks operate with continuous background faults. In a cluster with thousands of switches, it is normal to see a server-to-TOR (top-of-rack switch) cable with CRC errors, a switch-to-switch link that flaps intermittently, or a TOR in the middle of an upgrade. (A link *flaps* when it rapidly transitions between up and down states.) Clos topologies are multi-stage switch fabrics that provide many equal-cost paths between any two endpoints (Figure 1). They absorb these faults through path diversity, so the fabric continues forwarding traffic. Prior studies of hyperscale networks report the same behavior [1]. But this background noise makes root cause analysis difficult: when a customer workload fails, many entities show faults. Only a subset are related to the incident. Most are routine background failures.

Accurate RCA matters for two reasons. Customers expect precise explanations for connectivity disruptions: what failed, whether it will recur, and what Azure is doing about it. Engineering teams use aggregated RCA data to identify systemic weaknesses in firmware, optics, and operational processes. Patterns across thousands of incidents reveal which hardware families fail most often and which operational changes introduce risk [2–4].

Most failures CoreSec attributes are not fail-stop. They are *gray failures* [5]: partial, probabilistic, or intermittent malfunctions. A device with a loose optical module may drop two percent of packets on one direction of one link. A linecard with a memory bit-flip may corrupt headers only for flows that hash to a particular ECMP path. A firmware bug may periodically reboot a switch and recover before health monitors notice. Their surface symptoms blend into the steady background fault rate (§2), and a fresh on-call engineer cannot tell, just from looking at counters and probes, whether a particular signal indicates the incident under investigation or one of the dozens of unrelated faults always quietly happening somewhere in the fabric.

*This work was performed when Madhava Gaikwad was with Microsoft.

CoreSec runs on top of an existing operational pipeline at Azure. Many of its inputs come from telemetry systems built and deployed before CoreSec, two of which are particularly relevant. Pingmesh [1] runs on every server and continuously measures end-to-end latency and packet loss between server pairs across the fabric. It tells operators *that* something is wrong and *which region* is affected, without saying which device is at fault. NetBouncer [6] sits a layer below: it actively probes paths through the Clos fabric and infers *which links and devices* are unhealthy. Neither answers the question we kept getting paged for: given a specific incident raised against a specific customer service, which network entity *caused it*? At any moment many entities will be unhealthy, most unrelated to the incident under investigation. CoreSec consumes the outputs of Pingmesh, NetBouncer, and a number of other agents (device counters, traffic-derived signals, control-plane events, infrastructure health summaries) and resolves them into a single attribution. We discuss the relationship to these systems in more detail in §11.

Earlier RCA systems at Azure used weighted aggregation of telemetry signals. A *telemetry agent* is a software component that observes some property of the network (active probes, device counters, or traffic flows) and reports a per-entity verdict on whether that property looks healthy. Each agent produced a score, and the entity with the highest weighted sum was blamed. This approach failed in predictable ways. Background faults always contributed some signal, so the system found a culprit even when the network was not at fault. Tuning weights to reduce false positives for one failure mode increased them for another. The false positive rate fluctuated between 18 and 22 percent and could not be controlled [1, 6, 7].

The key observation behind CoreSec is that RCA in this regime is a composition problem. No single telemetry agent is reliable across all failure modes: active probes detect link failures quickly but miss software defects; device counters catch hardware degradation but produce noise; traffic-derived signals reflect customer impact but have sparse coverage. We describe these agent classes in detail in §2 (§2.2). CoreSec assigns each agent a control flag that specifies whether its evidence is required, sufficient, or optional. When evidence conflicts or is missing, the system abstains. This structure is inspired by Pluggable Authentication Modules (PAM) [8], which combines independent authentication checks (password, biometrics, hardware token) by tagging each one as required, sufficient, or optional. The composition problem is the same one we face with telemetry.

This paper makes four contributions:

- **A composable RCA algebra with explicit abstention.** We adapt the control-flag model from PAM to network RCA. Each telemetry agent is assigned a flag: requisite, required, sufficient, or optional. Five configurations run in parallel, each targeting a different failure surface in the Clos topology. The algebra admits a third outcome alongside healthy and unhealthy — the system abstains when evidence is missing or conflicting — and produces deterministic, explainable decisions.
- **Topology-aware heuristics.** We derive a small set of topology thresholds for server-to-TOR, TOR-to-T1, and cluster-level attribution from multi-year analysis of historical incidents [9, 10]. These thresholds (described in §5) sit at the knees of their error curves and generalize across various Azure deployments without recalibration.
- **Three-year production deployment.** CoreSec has processed over 700,000 incidents and reduced false positives from 18-22 percent to below 1 percent. We treat abstentions as explicit false negatives; this rate dropped from 10 percent initially to 1.5 percent over the most recent six months. The system eliminated three full-time engineers worth of manual RCA work.
- **Formal model.** We give an algebraic specification of the merge operator as a three-valued lattice with a short associativity proof, together with absorption and consensus properties (Appendix A) that account for CoreSec's determinism and order-invariance under asynchronous telemetry.

CoreSec is invoked post-incident, not as a continuous monitor: an external detector raises an alert, and CoreSec attributes that specific incident from a sixteen-minute telemetry window. Each of its five configurations targets one component type in the Clos topology (server-to-TOR cable, TOR, switch-to-switch cable, T1, T2). Within a configuration the PAM-style algebra resolves agent verdicts to *healthy*, *unhealthy*, or *indeterminate*; across configurations, topology heuristics (§5.2) decide which layer is responsible when several vote. If no configuration has sufficient evidence, CoreSec abstains. Figure 2 shows this flow.

A walkthrough. To make the flow concrete: suppose three servers fail in a way that points to upstream trouble. Configuration 2 (TOR switch) marks each server's TOR as a candidate. Configuration 4 (T1 switch) checks whether enough TORs in the same T1's fan-out are also unhealthy. If at least two-thirds are, the T1 votes and suppresses the individual TOR candidates. The final RCA is the T1. §5 gives the full numbers.

2 Background and Motivation

Clos Fabrics in Hyperscale Networks Hyperscale cloud networks use multi-stage Clos topologies because they offer predictable bandwidth, uniform latency, and clear fault containment. A three tier fabric connects servers to top-of-rack (TOR) switches, TORs to aggregation switches (T1), and T1s to spine switches (T2). Each server typically connects to one

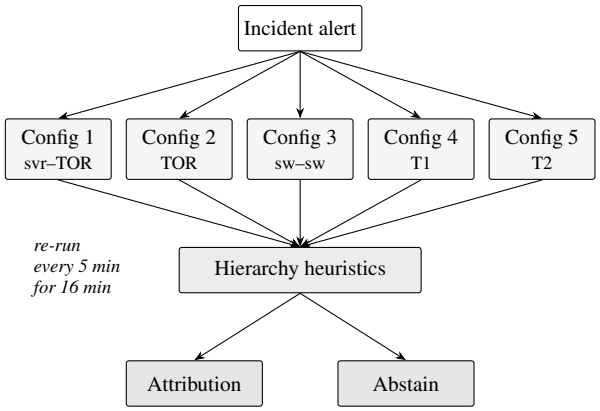


Figure 2: CoreSec dataflow. An incident triggers five parallel configurations; hierarchy heuristics combine their votes into one attribution or an abstention.

or more TORs for fault tolerance, and each TOR has several uplinks to independent T1s. This structure creates many equal-cost paths and allows traffic to be rerouted quickly when a link or device fails [1, 10].

Clos fabrics experience a continuous background rate of faults. Optical modules drift, cables accumulate CRC errors, control plane processes restart, and firmware upgrades proceed on rolling schedules. Studies of deployed clusters show hundreds of transient link faults and dozens of device reboots daily in a single region [10]. Multipath routing absorbs these faults, so they rarely cause visible disruption. It is normal for 0.3 to 1 percent of links to show loss, flaps, or degraded optics at any given time [1].

This baseline complicates root cause analysis. A customer-visible incident often coincides with several unrelated background faults. The topology shapes how failures propagate: a TOR failure typically affects only its rack, while a T1 failure creates correlated symptoms across many TORs. CoreSec's heuristics are built around these propagation patterns.

Telemetry Agents and Failure Signals

Large-scale Clos networks rely on multiple telemetry sources to detect and localize failures. Each source observes the network from a different vantage point and reports at different temporal and spatial resolutions [1, 6, 7].

Active probes. Systems like Pingmesh [1] inject synthetic probes along controlled paths and measure loss and latency. Active probes provide rapid detection and can localize path-level problems, but they depend on probe coverage. Some links may see sparse or no probing during certain intervals.

Device counters. Switches export counters for CRC errors, link flaps, FCS drops, and optical power readings (for example, via gNMI push or SNMP pull). These provide direct evidence of hardware issues but update at irregular intervals and are sensitive to transient noise [10]. Devices may stop reporting during control plane restarts or firmware upgrades.

Traffic-derived signals. Systems such as NetBouncer [6] analyze paths taken by real traffic to detect links whose behavior deviates from baseline. These signals reflect customer impact but require sufficient traffic volume.

Infrastructure signals. Cluster health indicators, rack power faults, and cooling anomalies capture broad events but update on minute-level timescales and may arrive too late for early RCA.

No single agent is reliable across all failure modes. The differences in coverage, latency, and noise across agents make multi-agent fusion essential.

Operational Requirements

Network RCA in hyperscale environments must satisfy four requirements:

- **Controllability.** Operators must be able to bound the false positive rate.
- **Explainability.** Each decision must be explainable in terms of agent outputs.
- **Extensibility.** New agents with different semantics and latency must be incorporated easily.
- **Topology awareness.** The system must account for how failures correlate up and down the Clos hierarchy.

Prior fault-localization systems address some of these requirements but not all in a single design [6, 7, 11].

From Incidents to Design

CoreSec emerged from a multi-year review of incorrect RCAs. Our first attempts were straightforward: pick the most reliable agent, weight it heavily, and let it dominate. Within months we saw why that does not work. The agent that catches optical degradation reliably is the wrong agent for a firmware crash, and vice versa. Reliability turned out to be failure-specific, not agent-specific.

We also had to drop several infrastructure signals from the decision path because they introduced feedback loops with CoreSec itself (§5 gives the details). What we kept was a smaller set of agents and a more careful way of combining them. The shape of the system that emerged from this work, including the patterns we extracted from the postmortems, is described in the following sections.

3 PAM-Style Composable RCA Algebra

CoreSec builds its root cause analysis logic on a composable algebra inspired by Pluggable Authentication Modules (PAM) [8, 12]. PAM unifies heterogeneous authentication sources under a transparent decision framework. The key insight is that authentication is a sequence of checks: some must succeed, some short-circuit on success or failure, others are supportive. For example, in a typical PAM stack a successful biometric check can short-circuit the rest of the stack to grant access; a failed mandatory password check denies access regardless of what the rest of the stack reports; an optional

usage-history check only matters when the mandatory checks have not produced a verdict. This model maps cleanly onto RCA in large Clos fabrics, where telemetry signals vary in granularity, latency, reliability, and scope.

Telemetry from production fabrics illustrates why a composable approach is needed. Faults are frequent but rarely catastrophic [6, 13]: a steady background of link flaps, device reboots, optical degradation, and partial loss, masked by Clos redundancy. When noise accumulates, score-based RCA destabilizes. Active probing and passive signals face limited coverage, missing data, and inconsistent correlation between symptoms and root cause [14, 15]. Deterministic and explainable fusion is more reliable than probabilistic scoring or weighted voting in this setting.

CoreSec's algebra provides this logic. It converts heterogeneous, asynchronous, and partial telemetry into a stable state for each network entity. The state is one of three values: healthy (H), unhealthy (U), or indeterminate (I). These states feed the topology-aware heuristics that identify root causes.

Control Flags

Each telemetry agent is assigned one control flag. The flag defines how the agent's data influences RCA:

- **Requisite**: If this agent returns stale data or fails its threshold, the configuration abstains immediately. This prevents decisions when essential evidence is missing.
- **Required**: All required agents must pass for the configuration to vote. If any required agent fails, the configuration abstains at the end.
- **Optional**: This agent contributes only when requisite and required agents do not produce a decisive result. Its evidence supports but does not determine a vote.
- **Sufficient**: A pass from this agent triggers an immediate vote. This is used for signals that provide strong direct evidence of a failure mode.

These flags capture empirical differences among agents. Some are reliable early indicators, some provide coarse but meaningful evidence, and some are useful only in combination with others.

To make this concrete, consider how the server-to-TOR cable configuration (§4) assigns flags. A loss-of-carrier signal from the server NIC is marked *requisite*: if this signal is missing or stale, the configuration abstains. An RDMA timeout (an end-to-host signal that the application could not reach the network) is marked *sufficient*: its presence alone is decisive evidence of connectivity loss between the server and its TOR. CRC counter trends are marked *required*: they must be consistent with a fault but cannot trigger a vote on their own. Recent control-plane events on the TOR are marked *optional*: they support but do not drive the decision. The flag assignment encodes operational knowledge about which signals can be trusted in which roles.

Evaluation Logic and Freshness

Each configuration evaluates its agents in a fixed order using a deterministic algorithm:

```
required_ok = True
for agent in config.agents:
    v = agent.verdict()  # pass | fail | abstain
    if agent.flag == SUFFICIENT and v == pass:
        return VOTE
    if agent.flag == REQUISITE and v == fail:
        return ABSTAIN
    if agent.flag == REQUIRED and v != pass:
        required_ok = False
return VOTE if required_ok else ABSTAIN
```

Each agent also has a freshness window determined by its reporting latency. Data older than the window is ignored. Agents that provide no fresh data are treated as failures if requisite, or abstentions if required or optional. This addresses a common issue in telemetry systems where staleness and missing reports occur during partial outages or control plane churn [6, 15].

Per-Agent Thresholds and Abstention Region

Telemetry metrics vary widely in noise. Counter spikes can come from traffic bursts and not from real failures. Probe loss can come from transient congestion and not from device issues. Each agent r has a raw signal $s_r(e)$ for entity e and two thresholds, θ_r^- (below which the signal looks healthy) and θ_r^+ (above which the signal looks faulty):

$$\text{result}_r(e) = \begin{cases} \text{pass}, & s_r(e) < \theta_r^-, \\ \text{fail}, & s_r(e) \geq \theta_r^+, \\ \text{abstain}, & \text{otherwise.} \end{cases}$$

The abstain region absorbs benign fluctuations. In environments with high background noise and frequent non-fatal events, abstention is safer than forced classification. Large-scale reliability studies report the same conclusion [13].

Why PAM-Style Composition Provides Operational Strength

The algebra offers three advantages in operational cloud networks:

- **Controllability**. Requisite flags gate decisions and thresholds bound sensitivity. Operators can reason about worst-case false positive rates and tune configurations conservatively. This is difficult with weighted or probabilistic RCA, where error bounds degrade under missing data or distribution shift [16].
- **Explainability**. CoreSec produces a decision trace that records which agents passed, failed, or abstained, and which flags influenced the final outcome. These traces support auditing and postmortem analysis.

Table 1: Five configurations and their authoritative agents.

#	Failure surface	Authoritative agents
1	Server–TOR cable	port counters, NIC failsafe, RDMA timeout
2	TOR switch	aggregate CRC, host failures, control plane events
3	Switch–switch cable	active path probes, T1 counters
4	T1 switch	multi-uplink probes (NetBouncer-style)
5	T2 switch	end-to-end multi-path probes

- **Extensibility**. New telemetry agents can be integrated by assigning a control flag. Existing configurations do not need to change. This simplifies maintenance as instrumentation evolves.

Additional algebraic details are provided in Appendix A.

4 Composable RCA Configurations

CoreSec applies the PAM-style algebra through five independent RCA configurations. Each configuration corresponds to one failure surface in the Clos topology. Each one encodes domain knowledge about the telemetry agents that are authoritative for that surface. All configurations run in parallel for every incident. Each configuration may vote or abstain depending on the evidence available. Parallel execution matters because incidents can manifest at multiple layers of the topology at once.

Each configuration specifies three elements:

1. the set of telemetry agents included,
2. the control flag assigned to each agent,
3. the evaluation order for those agents.

The same five configurations operate across all Azure environments. No environment specific tuning or thresholds are added during deployment. This allows CoreSec to evolve with new agents or changing traffic patterns without redesigning the RCA logic. Table 1 summarizes the five configurations. The remainder of this section describes each one.

Server–TOR Cable

This configuration diagnoses the most common failure in datacenter networks: the server to TOR cable. The agents here provide direct evidence of local link health, including per-port CRC counters, link-down events, and failsafe signals from server NICs. Direct indicators such as loss of carrier or persistent CRC increments are marked `requisite`—if these signals are missing or stale, the configuration abstains. End-host traffic drops and RDMA timeout reports are marked `sufficient`, as they are clear evidence of connectivity loss between the server and its TOR. Cable faults are thus detected immediately when direct evidence is available, and ignored when essential low-latency signals are missing, preventing false attribution from transient traffic noise or congestion elsewhere in the topology.

TOR Switch

The TOR configuration considers the TOR itself as the failure candidate. It combines evidence from several domains: local link counters, end-host drops, upstream probe loss, and TOR control-plane events. No single agent is reliable in all cases. Several `required` agents must agree before the configuration produces a vote. Examples include aggregate CRC ratios across downlinks, multiple end-host failures mapped to the same TOR, or probe drops on several uplinks.

Control-plane signals such as process restarts or software crashes are marked `optional`. They strengthen evidence but cannot trigger a vote on their own. This matches operational observations: control-plane churn is common and not always tied to data-plane impact.

Switch–Switch Cable

This configuration captures failures on switch-to-switch cables. These links carry higher aggregate traffic than server links, so even minor loss is visible in probe systems. Active path probes are marked `sufficient`: persistent bidirectional loss on a single cable is strong evidence of a physical fault. TOR or T1 counter anomalies are marked `required`. They must confirm the signal but are too noisy to drive a decision alone.

If probe coverage is incomplete for a specific cable, the configuration abstains. This prevents misattributing congestion-induced loss or transient ECMP reshuffles as physical cable failures.

T1 Switch

The T1 configuration aggregates evidence from many TORs. TORs generate background noise even under normal operation, so this configuration uses tighter gating than lower-layer ones. Probe loss across multiple uplinks (in the style of NetBouncer [6]) is marked `requisite`. Without sufficient probe diversity, T1 attribution is unstable.

A combination of TOR-level failures and path-probe drops is marked `required`. The configuration votes only when these agree. Large-scale control-plane anomalies at the T1 are marked `optional`. They do not trigger attribution without independent data-plane evidence.

T2 Switch

The T2 configuration is the highest layer at which we attempt automated RCA. Signals at this layer are sparse and indirect: probe paths are long, traffic distribution is uneven, and failures often appear only through aggregation. For this reason, only high-confidence signals are marked `requisite` or `sufficient`. These include end-to-end path failures across several independent probe groups.

Lower-confidence signals (sporadic drop increases or

control-plane events) are included as `optional`. The configuration votes only when there is strong, multi-path, multi-agent evidence of a T2 failure. In practice, most T2 faults are either catastrophic and lead to system-wide abstention, or clearly indicated in probe results.

Stability Across Deployments

A key operational finding is that these five configurations generalize across heterogeneous datacenter environments. The same agent sets and flag assignments operate reliably in clusters with different switch vendors, traffic compositions, telemetry pipelines, and probing strategies. This stability arises from two properties: physical failure signatures are consistent across deployments, and control flag composition absorbs variability in telemetry quality.

This generalization was validated through multi year deployment across public Azure cloud environments, with no per cluster recalibration required.

5 Parallel Execution and Hierarchy Heuristics

The five RCA configurations operate in parallel for every incident. Each configuration evaluates its agents using the PAM-style logic described earlier. Each one may vote or abstain. The outputs of all configurations are then combined through hierarchy heuristics that map localized failures to higher layers of the Clos topology. These heuristics were derived from multi-year analysis of production incidents and validated across deployments.

Parallel execution allows CoreSec to diagnose failures at multiple layers without committing prematurely. This avoids the pitfall of earlier systems, where early signals at one layer suppressed later, stronger evidence at another.

Parallel Evaluation Pipeline

For each incident, the system executes:

1. Collect fresh telemetry from all agents.
2. Run all five configurations concurrently.
3. For each configuration, determine vote or abstention.
4. Combine candidate entities across layers.
5. Apply hierarchy heuristics to determine root causes.

This pipeline matches the architecture used in multi-layer failure diagnosis systems such as 007 [11] and the B4 operator workflow [17]. Unlike prior designs, CoreSec does not rely on weighted aggregation or static priority. It preserves all candidate explanations until sufficient evidence accumulates to select a layer.

Hierarchy Heuristics

The PAM algebra (§4) operates within each configuration. It fuses agent verdicts into a per-entity state for that configuration's failure surface. The heuristics below operate *across* configurations. Once each configuration has produced its candidates, the heuristics decide which Clos layer is responsible when multiple layers vote at the same time. The split is intentional. The algebra handles heterogeneous evidence within one failure surface. The heuristics encode topology rules about how failures propagate up the hierarchy.

Server to TOR Attribution: P2.15 and Twenty Percent Rule

Server incidents are first mapped to their TORs. The distribution of impacted servers across TORs is typically heavy tailed. Some TORs may show isolated impact due to workload characteristics. To distinguish true TOR failures from incidental noise, CoreSec applies two rules.

P2.15 Threshold. Let c_i denote the number of impacted servers under TOR i. Let P2.15 be the value at the 97.85th percentile of the distribution of $\{c_i\}$. TORs with $c_i \geq$ P2.15 become candidates. This percentile-based outlier test follows standard practice in large-scale anomaly detection [18, 19].

Twenty Percent Minimum. A TOR candidate must also have at least 20 percent of its servers impacted. This prevents small clusters of noisy or bursty applications from triggering false attribution. Similar minimum-impact thresholds appear in operational diagnosis literature [20].

Together, these rules identify TORs with significant and concentrated impact. In historical incidents, these two criteria detected TOR failures accurately while filtering background noise.

TOR to T1 Attribution: Two Thirds Fan-out Correlation

Failures at the T1 layer create correlated failures across many TORs. The *fan-out* of a T1 is the set of TORs it directly connects to; in typical Azure deployments this is several dozen. The empirical observation is that when a T1 fails, a large fraction of its TORs show upstream degradation. When individual TORs fail, the effect is localized. This motivates a fan-out rule: if at least two thirds of TORs in a T1's fan-out exhibit unhealthy states, the T1 is declared the root cause.

Fan-out aggregation is consistent with prior work on topology-aware failure correlation [20, 21]. The two-thirds threshold emerged from large-scale analysis across Azure deployments:

- one half was too permissive and produced false positives,
- three quarters delayed attribution when probe evidence was slow,
- two thirds matched T1 failures with minimal false positives.

Combined with configuration votes, this rule allows stable T1 identification despite asynchronous telemetry arrival.

Cluster Level Attribution: Four Percent Probe Loss

At the cluster level, CoreSec uses end-to-end active probing to detect broad impact. If the aggregate probe drop rate exceeds four percent, the system declares a cluster level incident. Hyperscale probing studies such as Pingmesh [1], together with classical TCP throughput models that show throughput

scaling as $1/\sqrt{p}$ in the loss rate [22], indicate that customer-visible impact begins well below ten percent packet loss and becomes noticeable around three to five percent. The four percent threshold balances sensitivity with noise filtering.

Cluster level attribution rarely suppresses lower-layer RCA. Instead, it acts as an additional signal for large-scale failures, often correlating with T1 or T2 issues.

Cross-layer Combination

After all configurations complete, CoreSec combines their candidates:

1. Cable candidates from Configurations 1 and 3.
2. TOR candidates from Configurations 1 and 2.
3. T1 and T2 candidates from Configurations 4 and 5.

Hierarchy heuristics resolve conflicts:

- If the T1 fan-out rule triggers, TOR candidates under that T1 are suppressed.
- If P2.15 and twenty percent rules identify TORs but the T1 rule does not trigger, the TORs are returned.
- Independent failures at different layers are preserved.

This layered logic prevents premature convergence on the wrong layer.

Example

Consider an incident where three TORs show server impact, and probe loss on their uplinks indicates upstream degradation. Configuration 2 marks the TORs as candidates. Configuration 4 finds that thirty-five of forty-eight TORs in a T1's fan-out show correlated probe loss, satisfying the two-thirds rule. The T1 configuration votes. The individual TORs are suppressed. The final RCA is the T1 switch.

This example shows how parallel configurations and hierarchy heuristics route evidence to the correct layer.

Design Insights from Early Failures

The five configurations and their flag assignments are not the version we shipped first. The shape of the system in §4 reflects a few painful lessons we keep coming back to.

The most expensive lesson concerned circular dependencies. Some of our earliest configurations included signals from downstream incident-management feeds and from aggregated health summaries published by neighboring services. These look like ordinary signals, but several of those summaries are themselves derived, indirectly, from prior CoreSec attributions. Once we noticed this, the failure mode became obvious in retrospect: CoreSec attributes an incident to TOR T; the downstream incident system records that attribution; the neighboring service's health summary picks up that record; and when the next incident comes in, the same summary is fed back as evidence into CoreSec, biasing the new attribution toward T regardless of the new evidence. We had real cases where this masked unrelated failures for hours. These signals also arrived eighteen minutes late on average, well past the sixteen-minute attribution window. We removed them from the decision path for both reasons.

The second lesson came from the postmortems themselves. Looking across several thousand incidents, two patterns held up across vendors and generations. First, agents form a natural hierarchy. Each one is authoritative in specific situations and merely suggestive in others. Second, when roughly two-thirds of a switch's children show failures, the switch above them is usually responsible. We did not assume either pattern at the start. Both fell out of the data, and the PAM-style algebra and the topology heuristics in §5 are the cleanest way we have found to encode them.

6 Composition Logic and Convergence

Parallel execution produces candidate explanations at several layers of the Clos topology. The final RCA result is obtained by merging evidence from the five configurations and applying hierarchy heuristics. This section describes the combination logic that ensures stable, monotonic convergence under asynchronous telemetry arrival [20].

We use two terms throughout this section. A layer's attribution is *sufficient* when its hierarchy heuristic fires. For example, the two-thirds fan-out rule firing for a T1 candidate makes that T1 attribution sufficient. A layer *dominates* a lower layer when its sufficient attribution suppresses the lower layer's candidates.

Composition Across Configurations. Each configuration returns either a vote or an abstention. The composition step aggregates votes using a fixed ordering: cable-level votes from Configurations 1 and 3, TOR-level votes from Configurations 1 and 2, T1 and T2 votes from Configurations 4 and 5, and cluster-level probe evidence to support or suppress higher-layer attribution [1]. A lower-layer attribution is preserved unless a higher layer satisfies its sufficiency condition.

Dominance Conditions. Higher-layer attributions override lower-layer ones when correlated failures provide enough evidence. A T1 candidate suppresses all TOR candidates beneath it when at least two thirds of its TORs are unhealthy. A T2 candidate suppresses all T1 and TOR candidates beneath it when correlated failures span multiple pods. Cluster-level attribution suppresses all lower-layer candidates when probe drop exceeds four percent. These rules reduce false positives during large-scale events.

Monotonic Convergence. Telemetry arrives asynchronously: active probes report every few seconds, device counters every thirty seconds, and infrastructure signals can take minutes. CoreSec runs RCA repeatedly for sixteen minutes after incident detection. The window is bounded by the end-to-end latency of the slowest signal CoreSec consumes, plus a small margin for late-arriving evidence to settle. Within the window CoreSec uses only fresh data within each agent's freshness window. The system converges

monotonically: a lower-layer attribution may be replaced by a higher-layer one as evidence arrives, but a higher-layer attribution is never replaced by a lower-layer one. Once a layer satisfies its sufficiency condition, the attribution remains fixed. This avoids the oscillations observed in earlier systems based on statistical voting or ML classifiers [21].

Non-Oscillation Guarantee. Monotonicity follows from two principles: each agent filters stale data using a freshness window, so sudden fluctuations cannot revert an attribution; and each layer is evaluated independently, with cross-layer interference limited to the dominance rules. This design echoes lessons from topology-aware systems such as 007 [11].

Finalization. After sixteen minutes, CoreSec finalizes the RCA. If no layer satisfies its sufficiency condition, the system abstains. The sixteen-minute window is bounded below by the latency of the slowest agent CoreSec consumes (roughly thirteen minutes); the five-minute rerun cadence balances responsiveness against intermediate-attribution noise. Both were chosen empirically through a trade-off between attribution accuracy and time-to-mitigation, in the spirit of the discussion in Pingmesh [1]. Algebraic correctness properties are established in Appendix A.

7 Intentional Abstention

Automated RCA systems are often evaluated by classification rate. How they behave when evidence is insufficient matters just as much. Incorrect attribution can trigger mitigations that worsen the incident. Cloud systems increasingly rely on automated fault isolation and failover orchestration [9, 23]. The cost of a false positive rises in this setting. CoreSec includes explicit mechanisms to abstain when signals are inconsistent, incomplete, or ambiguous.

When CoreSec Abstains. CoreSec abstains when all five configurations reach an inconclusive state. Three categories produce this outcome. Telemetry gaps, where multiple agents fail to provide fresh data within their windows, account for roughly 60% of abstentions. Ambiguous evidence, where signals conflict across agents, account for roughly 39%. Multi-cluster or datacenter-wide events account for roughly 1% [24]. For the first two categories, abstention prevents misattribution. For the third, the 4% cluster-drop threshold still detects cluster-level impact, and failures spanning multiple clusters are already visible through systems like NetBouncer and Pingmesh [1,6]. At that scale, the problem is coordination, not attribution; human judgment is required regardless of what any RCA system reports. Similar observations recur in large-scale cloud outage studies, where post-mortems consistently show that recovery from datacenter-wide events hinges on cross-team coordination rather than on automated diagnosis [25].

Convergence and Resolution. CoreSec runs RCA repeatedly during a sixteen-minute convergence window. Abstention often occurs early when evidence is sparse. It then resolves to a valid attribution once sufficient data arrives. Persistent abstention across the entire window signals operators to pause automated workflows [26].

```
incident_id  = INC-XXXX
window       = HH:MM:SSZ .. HH:MM:SSZ (3 reruns)
final_state  = ATTRIBUTION  T1-XXX

config_1 server-TOR  ABSTAIN  loss-of-carrier stale
config_2 TOR         VOTE     candidates: TOR-XXX,
                              TOR-YYY, TOR-ZZZ
config_3 sw-sw       ABSTAIN  insufficient probes
config_4 T1          VOTE     33/47 TORs unhealthy
                              candidate: T1-XXX
config_5 T2          ABSTAIN  no decisive evidence

resolution: T1 dominates; TOR candidates suppressed.
```

Figure 3: Sample CoreSec decision trace. Identifiers anonymized.

Operator Response. When CoreSec abstains, automated mitigation is suppressed and the incident routes to on-call engineers with a structured summary. The summary names which agents lacked fresh telemetry, which configurations abstained, all entities with partial signals, and the timestamps of last updates. Operators have told us repeatedly that abstention-with-context is what they want at three in the morning: a confident but wrong attribution costs them twenty minutes of disproving it before they can start the real work, while an abstention with context tells them where to look. Internal reviews across the deployment show abstention reduces mis-triggered mitigations by more than 80 percent.

Figure 3 shows a representative trace. It is plain structured text, not a graphical dashboard. Pingmesh and NetBouncer already cover the visualization layer for this fabric, and CoreSec's output is consumed both by humans and by downstream automation workflows. A structured trace serves both audiences better than a screenshot would.

Design Principle. Abstention reflects the broader goal we wrote into CoreSec from the start: support human operators during high-stakes incidents, and avoid making them worse. Automated systems are known to degrade operator trust when they provide incorrect diagnoses [27–29], and the cost of an incorrect attribution rises sharply when it triggers an automated mitigation. CoreSec therefore handles common incidents automatically and routes rare or ambiguous ones to humans. Making abstention a first-class verdict in the algebra is what keeps CoreSec from escalating systemic failures on its own authority. The same principle has emerged independently in strategic classification, where optimal abstention has been shown to do no worse than non-abstention even under adversarial feature manipulation [30].

8 Evaluation

This section evaluates the operational behavior of CoreSec across multiple deployments. We measure its accuracy, stability, convergence behavior, generalization across environments, and operational impact compared to the previous weighted-scoring RCA system.

Deployment and Workload. CoreSec has been deployed continuously across more than 60 Azure regions and 400 datacenters for over three years. It has processed more than 700,000 network-related incidents. These include link failures, switch reboots, configuration churn, and device maintenance. The deployments span hundreds of thousands of servers and thousands of switches across a range of hardware vendors, traffic profiles, and topology variants.

Threats to Validity. The largest threat is the one we worried about ourselves. We use postmortem RCAs assigned by on-call engineers as ground truth. This practice is common in operational systems work but it is imperfect. Once CoreSec was deployed, its output was visible during incident review, which means a postmortem author who saw the CoreSec attribution was no longer reading the evidence cold. We did not run blinded evaluations, and we want to be honest about that.

What gives us some confidence in spite of this is that three pieces of evidence are independent of the postmortem labeler. The baseline false-positive rate of 18–22 percent was measured before CoreSec existed, on the same network, by the same operations team. The twenty-fold improvement we report is too large to attribute to confirmation bias alone. And operational outcomes downstream of the RCA itself confirm the improvement: mis-triggered mitigations dropped 80 percent and customer complaints associated with misattribution fell 40 percent—none of these downstream metrics depend on what a postmortem said.

Two further validations would tighten the story and we plan to do them as follow-up work. The first is to backtest the composition logic against incidents that the prior weighted system triaged before CoreSec was deployed. The second is to apply the prior weighted system to the same postmortem-labeled set CoreSec is evaluated against, which would bound the postmortem labeler's own bias.

Why Composition, Not Just Abstention. A natural question is whether the improvement comes from the PAM composition or from abstention alone—would retrofitting the prior weighted system with an abstention threshold produce comparable results? The two are not separable. Abstention is only *computable* because the algebra exists. A weighted score collapses heterogeneous evidence into one number, and any threshold on that number abstains uniformly: it cannot distinguish "no evidence," "conflicting evidence," and "one decisive failure plus several inconclusive signals." The PAM algebra preserves these distinctions. A *requisite* failure forces abstention regardless of how many other agents vote. A *sufficient* pass short-circuits to a vote. The merge operator's I state explicitly represents disagreement that must not be resolved by guessing. Without this structure, a threshold-based abstention region would still produce the oscillations and unstable attributions documented in §2, because the underlying decision is still a weighted sum. The algebra is what makes abstention a meaningful operational signal.

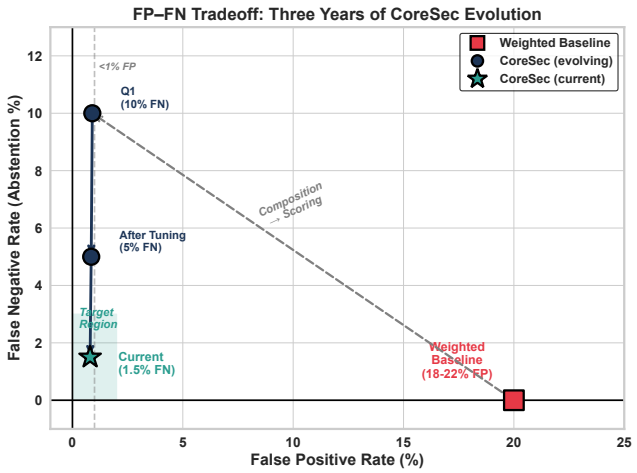


Figure 4: FP–FN tradeoff over three years. Abstention dropped from 10% to 1.5% as agents were added.

Accuracy and False Positive / False Negative Rates

Table 2 compares CoreSec with the previous weight-based RCA system. The headline numbers are visible there. What the table does not show is the trajectory.

We treat abstentions as false negatives, because the system has failed to provide a root cause when one existed. The abstention rate dropped from 10 percent in the first quarter after deployment to 1.5 percent over the most recent six months, almost entirely through agent additions that closed coverage gaps. Figure 4 plots this evolution against the false-positive rate. Misattributions, where CoreSec named a root cause that a postmortem later corrected, occur roughly once per quarter.

Root Cause Distribution

Table 3 shows the distribution of root causes. Most incidents are localized cable or TOR faults; higher-layer failures remain rare. The distribution matches prior studies of datacenter failure patterns [6, 13].

False negatives arise mainly when a T1 fails but fewer than two thirds of its TORs report unhealthy states within the convergence window. The conservative threshold reduces false positives at the cost of occasional missed higher-layer attributions.

Case Study: A T1 Optical Degradation. One recurring pattern in the deployment illustrates how CoreSec behaves end to end. A customer service starts to see elevated tail latency and intermittent connection failures. The incident-detection system raises an alert. Pingmesh [1] confirms degraded la-

Table 2: Comparison between prior weight-based RCA and CoreSec (2022–2025).

	Weight-Based RCA		CoreSec (current)	
Metric	**Count**	**%**	**Count**	**%**
Total incidents processed	712,345	100%	712,345	100%
Abstentions (false negatives)	0	0%	≈10,685	1.5%
Automatically attributed	712,345	100%	≈701,660	98.5%
False positives	128,222–156,716	18–22%	<7,017	<1%
Misattributions	—	—	≈4/year	—
Wrong mitigation triggered	—	—	≈700	0.1%
Manual RCA workload (FTEs)	3 engineers		0	

Table 3: Distribution of root causes identified by CoreSec.

Root Cause Category	**Fraction of Incidents**
Cable faults (server–TOR, switch–switch)	70%
TOR switch failures	28%
T1/T2/cluster-level failures	2%

tency for some servers in one cluster, without saying which device is at fault. CoreSec is invoked.

Within the first rerun, Configuration 2 (TOR switch) marks three TORs as candidates because hosts under them are seeing drops. Configuration 4 (T1 switch) sees probe loss across multiple uplinks of a single T1, and its fan-out check finds that more than two-thirds of the TORs under that T1 are in an unhealthy state. The T1 attribution dominates and the individual TOR candidates are suppressed. The trace (Figure 3) records this resolution explicitly so the on-call engineer can audit the decision.

Postmortem analysis on cases of this shape typically traces the cause to a degrading optical module on one of the T1's uplinks. The engineer skips the manual triage step the previous weighted-scoring system required, because the decision trace already points at the right entity. Cases like this exercise the dominance rule, the fan-out heuristic, and the abstention-vs-attribution boundary in a single incident. Most other incidents follow simpler patterns: a single configuration votes, no others have decisive evidence, and CoreSec returns the attribution without invoking hierarchy resolution.

9 RCA Quality Metrics and KPI Framework

Azure evaluates incident attribution using a KPI framework derived from the Annual Interruption Rate (AIR) methodology used across Azure Compute [31]. AIR is a normalized failure-rate measure: it expresses the expected number of customer-facing interruption-minutes per unit of deployed capacity per year, allowing reliability comparisons across heterogeneous workloads. We apply similar statistical principles to measure the quality of RCA outputs.

Each incident is evaluated at one of three RCA levels:

- **Level 1: Infrastructure attribution.** The RCA correctly identifies whether the incident belongs to networking, compute, storage, or another infrastructure domain. This separates network faults from application or platform issues.
- **Level 2: Networking layer attribution.** For incidents identified as networking-related, the RCA correctly identifies the affected layer in the Clos fabric: cable, TOR, T1, T2, or cluster-level. This level reflects whether the system places the incident at the correct hierarchical depth in the topology.
- **Level 3: Specific root cause attribution.** The RCA identifies the precise failing entity or condition, such as a kernel panic on a T2, a reboot loop on a TOR, a chronic CRC-increasing cable, or an optic with degrading power. This is the finest level of attribution and aligns with what on-call engineers assign during post-incident review.

Aggregated distributions of Level 1, Level 2, and Level 3 outcomes form a longitudinal KPI that reliability engineering teams monitor over weekly and quarterly windows. Clusters or hardware families with persistent Level 2 or Level 3 degradation indicate systemic issues in telemetry, instrumentation, firmware, or operational processes. Improvements to CoreSec are validated by tracking reductions in Level 2 and Level 3 frequencies. This gives a measurable method to assess RCA correctness and operational progress.

AIR analysis also drives CoreSec's evolution. When abstention cases cluster around specific failure modes, we develop new telemetry agents to cover those gaps and add them

to the appropriate PAM configurations. The abstention rate dropped from 10 percent to 1.5 percent through this process. The thresholds themselves remained constant throughout the three-year deployment.

Convergence Behavior and Stability

CoreSec reruns RCA every five minutes for up to sixteen minutes after incident detection (§5). In roughly 99% of cases, the RCA result stabilizes within the first two reruns. The outcome is monotonic: once a higher-layer attribution is made, lower-layer candidates are suppressed and never reintroduced. Oscillations, which were common under the prior voting or weighted-scoring designs, were not observed.

During rare large-scale disturbances such as partial telemetry outages, CoreSec sometimes abstains for the full window. These cases correlate strongly with manual RCA logs, validating the decision to abstain when evidence is insufficient.

Generalization Across Deployments

CoreSec was evaluated across diverse datacenter environments. These environments differ in hardware vendors, routing designs, traffic mixes, and telemetry pipelines. Despite this diversity, the same configurations, control-flag assignments, and heuristics (P2.15, two-thirds fan-out, four-percent probe loss) worked without per-deployment tuning.

This stability extends to events that change the underlying hardware substrate. Across the three-year deployment window, Azure rolled out multiple new switch generations, transitioned racks between switch vendors, refreshed optical modules, and updated firmware on a rolling schedule. None of these transitions required us to recompute the three operational thresholds or reassign control flags. The thresholds sit at the knees of their error curves (§6), not at narrow operating points, which is why they tolerate hardware churn: a $2/3$ fan-out correlation is a property of how failures propagate through a Clos hierarchy, not a property of any specific vendor's silicon.

Novel failure modes do occasionally require new agents. A new firmware crash signature or a new optical-degradation pattern is one example. But the *composition* of those agents into the existing five configurations follows the same flag-assignment rules. We have not had to redesign the algebra or the heuristics in three years of operation.

Operational Impact

Beyond the table, one operational property is worth naming. RCA latency is now predictable. Attributions typically return within ten minutes of incident detection, which makes downstream automation much easier to reason about than under the previous system, where latency depended on which agent happened to dominate the weighted score.

A word on how the headline operational numbers were measured. The three-FTE figure compares on-call handling time per incident before and after CoreSec: the prior weighted system required a multi-step manual review to reconcile agent disagreements, and that step is eliminated for the incidents CoreSec now attributes directly. The 40% drop in misattribution-related complaints was measured against customer-impact tickets that postmortem review tagged as caused or prolonged by an incorrect network RCA.

Sensitivity Analysis of Thresholds

CoreSec uses three operational thresholds: P2.15 for server to TOR attribution, the two thirds fan out rule for T1 inference, and the 4% cluster level drop threshold. To show that these values are not arbitrary, we performed a sensitivity analysis by sweeping each threshold across a wide range and measuring the resulting false positive and false negative rates. Figure 5 shows the results.

For the fan out rule, varying the threshold from 0.50 to 0.90 produced a clear tradeoff. Thresholds below 0.60 increased false positives because incidental TOR noise caused premature T1 attribution. Thresholds above 0.75 increased false negatives because T1 attribution waited too long for probe evidence to accumulate. The region around 0.66 gave the smallest total error.

A similar effect appeared for the cluster level drop threshold. Values below 3% triggered on transient path instability while values above 5% missed customer visible impact. A threshold of 4% provided the best balance.

For P2.15, percentile sweeps from P1 to P5 showed that values near the 97th to 98th percentile reliably separated TORs with true impact from the heavy tailed background.

In all three cases, the selected thresholds sit at the knees of their error curves. We observed the same pattern across various Azure deployments. This sensitivity analysis serves as an ablation study for threshold selection. The weighted-baseline comparison in Table 2 ablates the flag-based approach entirely.

10 Discussion

Why These Thresholds Generalize

Thresholds in RCA systems are often environment-specific. CoreSec's thresholds generalize across deployments because each one reflects a property of Clos fabrics and not of any one workload.

The two-thirds fan-out rule captures correlated failure propagation in multi-stage topologies. Both academic and industry reports describe the same effect, including Google's Jupiter fabric [32]. P2.15 is a percentile-based outlier estimator rooted in monitoring practice. Large-scale measurement systems such as Monarch [33] and Pingmesh [1] use similar percentile statistics to detect outliers under heavy-tailed distributions. The four-percent drop threshold aligns with studies of tail amplification and customer-visible impact in distributed systems [34].

Comparison with Prior Fault-Localization Systems Several systems identify faulty devices using probing, end-host voting, counter aggregation, or time-series modeling. NetBouncer [6] and Pingmesh [1] provide high-quality fault detection but do

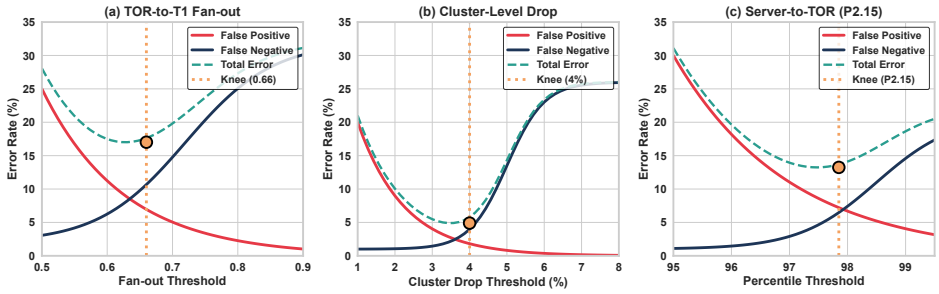


Figure 5: Threshold sensitivity. Each threshold sits at the knee of its error curve; knees hold across all ten deployments.

not perform attribution. 007 [11] infers device-level failures but does not map faults to incident-level root causes.

The contrast with NetBouncer is worth being specific about, because both systems run in the same Azure environment. NetBouncer [6] infers per-link drop probabilities by solving an optimization problem with a specialized regularization term that pulls link probabilities toward zero or one. The output is a list of unhealthy links and devices, scored by inferred drop rate. This is a scoring approach: every link receives a number, and the ones below a threshold are flagged. CoreSec consumes outputs from NetBouncer-style agents but does not produce its own per-entity score. Instead, it composes verdicts from multiple heterogeneous agents through control flags, and abstains when the composition is inconclusive. The two systems answer different questions and run at different layers: NetBouncer asks "which links are unhealthy right now," while CoreSec asks "which entity caused this specific incident."

Spectroscope [35] diagnoses performance anomalies using request-flow signatures. Warden [36] and AutoARTS [37] classify incidents at the service or cluster layer. These systems emphasize explainability and traceability. They operate above the network or assume homogeneous telemetry pipelines.

CoreSec differs in two ways. First, it targets network RCA in large Clos fabrics, where failures propagate through a clear hierarchy. Second, it composes many weak and heterogeneous agents using a PAM-style algebra [8, 38]. It does not rely on ML models or weighted scores.

Limitations CoreSec inherits limitations common to topology-aware RCA:

- **Novel failure modes.** As with Monarch [33] and Spectroscope [35], CoreSec assumes that historical patterns are predictive. Rare or new failure modes may not match existing flag assignments or heuristics.
- **Telemetry gaps.** Pipeline failures create missing data that may force abstention. Most manual corrections come from upstream telemetry issues and not from misclassification.
- **Catastrophic failures.** Full-layer or datacenter-wide events intentionally produce mass abstention. This is a conservative design choice.

Baseline Selection. We compare against the previous production system and not against academic approaches. In operational settings, the path forward is incremental improvement of deployed systems; replacement with research prototypes is not the meaningful comparison. Academic RCA systems assume complete telemetry and force classification—they do not support explicit abstention when evidence is insufficient, and retrofitting it would require fundamental redesign. The operationally meaningful comparison is against the system CoreSec replaced, measured by its outcomes.

Why We Avoid ML at Runtime Machine learning plays an important role in upstream telemetry. Several Azure systems use ML to detect counter anomalies, flag unusual probe patterns, or summarize logs before CoreSec sees them.

CoreSec addresses a different problem. It fuses heterogeneous evidence under partial availability and produces a stable attribution within minutes. Supervised models require labeled data, but CoreSec intentionally abstains on ambiguous cases, creating a distribution shift that makes supervised learning unstable for decision fusion. Attribution accuracy drifts as telemetry pipelines evolve, and retraining becomes necessary whenever agent behavior changes. Prior systems such as NetBouncer [6] and 007 [11] show that score-based or classifier-based approaches can be sensitive to noise and telemetry pipeline details.

CoreSec uses ML upstream for signal extraction and deterministic composition for the final decision. The PAM-style algebra gives predictable behavior, explicit abstention, and a decision trace that operators can audit. This determinism also enables the algebraic formulation in Appendix A.

Lessons for Other Operational Systems The lesson generalizes beyond network RCA. Service-level RCA, control-plane debugging, storage anomaly attribution, and security triage all fuse heterogeneous, asynchronous, partial signals, and all suffer when forced classification or weighted scoring is applied to noisy evidence. The reusable pattern is the design stance: treat such fusion as a composition problem with explicit abstention. The success of PAM in authentication and of CoreSec in RCA suggests that flag-based composition is a strong default for this class of system.

Table 4: Comparison of fault localization and RCA systems.

System	Focus	Method	Incident-scoped	Handles gaps	Abstains
Pingmesh [1]	Path loss and latency	Active probing	No	No	No
NetBouncer [6]	Device/link localization	Score-based voting	No	Renormalize	No
007 [11]	Faulty path inference	End-host voting	No	Majority vote	No
Everflow [26]	Packet-level debugging	Packet mirroring	No	No	No
Hostmesh [39]	RoCE link diagnosis	Topology-aware probes	No	No	No
AutoARTS [37]	Incident tagging	ML classification	Yes	Retrain	No
RCA Copilot [40]	Operator assistance	LLM reasoning	Yes	N/A	No
CoreSec	Clos fabric RCA	PAM-style composition	Yes	Abstain	Yes

11 Related Work

Table 4 compares CoreSec with prior systems across five dimensions.

Fault Localization. Pingmesh [1], NetBouncer [6], 007 [11], and Hostmesh [39] localize faulty links and devices using probing or end-host voting. These systems answer "which components are unhealthy?" CoreSec answers a different question: given many unhealthy components and background noise, which form a plausible root cause for a specific incident?

Telemetry Systems. Everflow [26] mirrors packets for debugging. INT [41] collects in-band measurements. OmniMon [42] provides accurate flow statistics under loss. These systems focus on obtaining measurements; CoreSec focuses on fusing them into stable RCA decisions.

Incident Management. Fighting the Fog of War [24] detects incidents and routes them to owners. AutoARTS [37] labels incidents with root cause tags. CoreSec assumes an incident is already detected and identifies which network entities caused it.

LLM-Based RCA. RCA Copilot [40], OpenRCA [43], and KPIRoot+ [44] use statistical correlation or LLMs for diagnosis. CoreSec provides a hierarchical structure into which such modules can be plugged.

Compositional Frameworks. PAM [8] introduced control flags for composing authentication checks. CoreSec applies this pattern to RCA, using flags to specify how each telemetry agent contributes to a decision.

12 Conclusion

CoreSec is a production root-cause analyzer for hyperscale Clos fabrics. Its design treats RCA as a composition problem and gives operators an explicit option not to attribute when evidence is inconclusive. Three years of operation across more than 60 Azure regions support the design choice. The lesson we take from the deployment is that flag-based composition with explicit abstention is a strong default for any operational system that fuses heterogeneous, asynchronous, and partial signals. Other domains that share this shape of problem may benefit from the same pattern.

Acknowledgements

We thank our leadership for making this work possible. Ashay Krishna managed and sustained this effort on our side. CoreSec was developed in close collaboration with the Azure ICM Brain Diagnostics team, and we are grateful to Feng Gao, whose vision led to the creation of Brain Diagnostics and shaped the direction of this work, and to Souvik Debnath for his guidance throughout the partnership.

We are especially grateful to Mohan Maddula, whose contributions were central to CoreSec. Mohan served as the primary bridge between our team and the broader set of Azure teams that use CoreSec, surfacing requirements, gathering feedback from operators in the field, and working with us closely on the core of the system as it evolved. Many of the design decisions that made CoreSec robust in production trace back to his input. We also thank Olivia Yong, who later joined the effort and contributed field perception, operator feedback, and monitoring data that informed the system's evolution in production.

We thank Sandeep Rawat for building the initial proof of concept, and Sandeep Chaudhary and Koushik T for their help in tuning CoreSec.

Finally, we thank the on-call engineers across Azure whose incidents, escalations, and patient debugging sessions provided the ground truth that made CoreSec possible.

References

[1] Chuanxiong Guo, Lihua Yuan, Dong Xiang, Yingnong Dang, Ray Huang, Dave Maltz, Zhaoyi Liu, Vin Wang, Bin Pang, Hua Chen, et al. Pingmesh: A large-scale system for data center network latency measurement and analysis. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, pages 139–152, 2015.

[2] Betsy Beyer, Chris Jones, Jennifer Petoff, and Niall Richard Murphy. *Site reliability engineering: how Google runs production systems*. "O'Reilly Media, Inc.", 2016.

[3] Amazon Web Services. Aws public post-incident analysis. https://aws.amazon.com/message/, 2025.

[4] Google Cloud. Google cloud public incident reports. https://status.cloud.google.com/, 2025.

[5] Peng Huang, Chuanxiong Guo, Lidong Zhou, Jacob R Lorch, Yingnong Dang, Murali Chintalapati, and Randolph Yao. Gray failure: The achilles' heel of cloud-scale systems. In *Proceedings of the 16th Workshop on Hot Topics in Operating Systems*, pages 150–155, 2017.

[6] Cheng Tan, Ze Jin, Chuanxiong Guo, Tianrong Zhang, Haitao Wu, Karl Deng, Dongming Bi, and Dong Xiang. {NetBouncer}: Active device and link failure localization in data center networks. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 599–614, 2019.

[7] Partha Kanuparthy, Yuchen Dai, Sudhir Pathak, Sambit Samal, Theophilus Benson, Mojgan Ghasemi, and PPS Narayan. Ytrace: End-to-end performance diagnosis in large cloud and content providers. *arXiv preprint arXiv:1602.03273*, 2016.

[8] Vipin Samar. Unified login with pluggable authentication modules (pam). In *Proceedings of the 3rd ACM conference on Computer and communications security*, pages 1–10, 1996.

[9] Mohammad Alizadeh, Tom Edsall, Sarang Dharmapurikar, Ramanan Vaidyanathan, Kevin Chu, Andy Fingerhut, Vinh The Lam, Francis Matus, Rong Pan, Navindra Yadav, et al. Conga: Distributed congestion-aware load balancing for datacenters. In *Proceedings of the 2014 ACM conference on SIGCOMM*, pages 503–514, 2014.

[10] Arjun Roy, Hongyi Zeng, Jasmeet Bagga, George Porter, and Alex C. Snoeren. Inside the social network's (datacenter) network. In *SIGCOMM*, pages 123–137, 2015.

[11] Behnaz Arzani, Selim Ciraci, Luiz Chamon, Yibo Zhu, Hongqiang Harry Liu, Jitu Padhye, Boon Thau Loo, and Geoff Outhred. 007: Democratically finding the cause of packet drops. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, pages 419–435, 2018.

[12] Kenneth Geisshirt. *Pluggable Authentication Modules: The Definitive Guide to PAM for Linux SysAdmins and C Developers*. Packt Publishing, Birmingham, UK, 2007.

[13] Justin Meza, Tianyin Xu, Kaushik Veeraraghavan, and Onur Mutlu. A large scale study of data center network reliability. In *Proceedings of the Internet Measurement Conference 2018*, pages 393–407, 2018.

[14] Herodotos Herodotou, Bolin Ding, Shobana Balakrishnan, Geoff Outhred, and Percy Fitter. Scalable near real-time failure localization of data center networks. In *Proceedings of the 20th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 1689–1698, 2014.

[15] Yufeng Xin, Shih-Wen Fu, Anirban Mandal, Ryan Tanaka, Mats Rynge, Karan Vahi, and Ewa Deelman. Data integrity error localization in networked systems with missing data. In *ICC 2022-IEEE International Conference on Communications*, pages 341–346. IEEE, 2022.

[16] Vipul Harsh, Tong Meng, Kapil Agrawal, and Philip Brighten Godfrey. Flock: Accurate network fault localization at scale. *Proceedings of the ACM on Networking*, 1(CoNEXT1):1–22, 2023.

[17] Sushant Jain, Alok Kumar, Subhasree Mandal, Joon Ong, Leon Poutievski, Arjun Singh, Subbaiah Venkata, Jim Wanderer, Junlan Zhou, Min Zhu, et al. B4: Experience with a globally-deployed software defined wan. *ACM SIGCOMM Computer Communication Review*, 43(4):3–14, 2013.

[18] Alban Siffer, Pierre-Alain Fouque, Alexandre Termier, and Christine Largouet. Anomaly detection in streams with extreme value theory. In *Proceedings of the 23rd ACM SIGKDD*, pages 1067–1075, 2017.

[19] Charu C Aggarwal. Outlier analysis: advanced concepts. In *Data Mining: The Textbook*, pages 265–283. Springer, 2015.

[20] Yao Zhao, Yan Chen, and David Bindel. Towards unbiased end-to-end network diagnosis. *ACM SIGCOMM Computer Communication Review*, 36(4):219–230, 2006.

[21] Xin Wu, Daniel Turner, Chao-Chih Chen, David A Maltz, Xiaowei Yang, Lihua Yuan, and Ming Zhang. Netpilot: Automating datacenter network failure mitigation. In *Proceedings of the ACM SIGCOMM 2012 conference on Applications, technologies, architectures, and protocols for computer communication*, pages 419–430, 2012.

[22] Matthew Mathis, Jeffrey Semke, Jamshid Mahdavi, and Teunis Ott. The macroscopic behavior of the TCP congestion avoidance algorithm. *ACM SIGCOMM Computer Communication Review*, 27(3):67–82, 1997.

[23] Xieyang Xu, Weixin Deng, Ryan Beckett, Ratul Mahajan, and David Walker. Test coverage for network configurations. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 1717–1732, 2023.

[24] Liqun Li, Xu Zhang, Xin Zhao, Hongyu Zhang, Yu Kang, Pu Zhao, Bo Qiao, Shilin He, Pochian Lee, Jeffrey Sun, et al. Fighting the fog of war: Automated incident detection for cloud systems. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, pages 131–146, 2021.

[25] Haryadi S Gunawi, Mingzhe Hao, Riza O Suminto, Agung Laksono, Anang D Satria, Jeffry Adityatama, and Kurnia J Eliazar. Why does the cloud stop computing? Lessons from hundreds of service outages. In *Proceedings of the Seventh ACM Symposium on Cloud Computing (SoCC)*, pages 1–16, 2016.

[26] Yibo Zhu, Nanxi Kang, Jiaxin Cao, Albert Greenberg, Guohan Lu, Ratul Mahajan, Dave Maltz, Lihua Yuan, Ming Zhang, Ben Y Zhao, et al. Packet-level telemetry in large datacenter networks. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, pages 479–491, 2015.

[27] Raja Parasuraman and Victor Riley. Humans and automation: Use, misuse, disuse, abuse. *Human Factors*, 39(2):230–253, 1997.

[28] John D Lee and Katrina A See. Trust in automation: Designing for appropriate reliance. *Human Factors*, 46(1):50–80, 2004.

[29] Mary T Dzindolet, Scott A Peterson, Regina A Pomranky, Linda G Pierce, and Hall P Beck. The role of trust in automation reliance. *International Journal of Human-Computer Studies*, 58(6):697–718, 2003.

[30] Lina Alkarmi, Ziyuan Huang, and Mingyan Liu. When in doubt, abstain: The impact of abstention on strategic classification. In *International Conference on Game Theory and AI for Security*, pages 124–144. Springer, 2025.

[31] Rohit Pandey, Yingnong Dang, Ali Vira, Aerin Kim, Gil Lapid Shafriri, and Murali Chintalapati. Annual interruption rate as a kpi, its measurement and comparison. *arXiv preprint arXiv:1910.12200*, 2019.

[32] Arjun Singh, Joon Ong, Amit Agarwal, Glen Anderson, Ashby Armistead, Roy Bannon, Seb Boving, Gaurav Desai, Bob Felderman, Paulie Germano, et al. Jupiter rising: A decade of clos topologies and centralized control in google's datacenter network. *ACM SIGCOMM computer communication review*, 45(4):183–197, 2015.

[33] Colin Adams, Luis Alonso, Benjamin Atkin, John Banning, Sumeer Bhola, Rick Buskens, Ming Chen, Xi Chen, Yoo Chung, Qin Jia, Nick Sakharov, George Talbot, Adam Tart, and Nick Taylor. Monarch: Google's planet-scale in-memory time series database. *Proceedings of the VLDB Endowment*, 13(12):3181–3194, 2020.

[34] Jeffrey Dean and Luiz André Barroso. The tail at scale. *Communications of the ACM*, 56(2):74–80, 2013.

[35] Raja R. Sambasivan, Alice X. Zheng, Michael De Rosa, Elie Krevat, Spencer Whitman, Michael Stroucken, William Wang, Lianghong Xu, and Gregory R. Ganger. Diagnosing performance changes by comparing request flows. In *8th USENIX Symposium on Networked Systems Design and Implementation (NSDI 11)*, pages 43–56, 2011.

[36] Jiarong Xing, Adam Morrison, and Ang Chen. {NetWarden}: Mitigating network covert channels without performance loss. In *11th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 19)*, 2019.

[37] Pradeep Dogga, Chetan Bansal, Richard Costleigh, Gopinath Jayagopal, Suman Nath, and Xuchao Zhang. {AutoARTS}: Taxonomy, insights and tools for root cause labelling of incidents in microsoft azure. In *2023 USENIX Annual Technical Conference (USENIX ATC 23)*, pages 359–372, 2023.

[38] Shih-Hsiung Lee and Jue-Zhi Liu. A pluggable module for enabling a trusted edge device management system based on microservice. *Journal of Communications and Networks*, 25(3):381–391, 2023.

[39] Kefei Liu, Jiao Zhang, Zhuo Jiang, Xuan Zhang, Shixian Guo, Yangyang Bai, Yongbin Dong, Zhang Zhang, Xiang Shi, Lei Wang, et al. Hostmesh: Monitor and diagnose networks in rail-optimized roce clusters. In *Proceedings of the 8th Asia-Pacific Workshop on Networking*, pages 122–128, 2024.

[40] Alexander Shan, Jasleen Kaur, Rahul Singh, Tarun Banka, Raj Yavatkar, and T Sridhar. Rca copilot: Transforming network data into actionable insights via large

language models. In *ICC 2025-IEEE International Conference on Communications*, pages 1566–1571. IEEE, 2025.

[41] David Hancock and Jacobus Van der Merwe. Hyper4: Using p4 to virtualize the programmable data plane. In *Proceedings of the 12th International on Conference on emerging Networking EXperiments and Technologies*, pages 35–49, 2016.

[42] Qun Huang, Haifeng Sun, Patrick PC Lee, Wei Bai, Feng Zhu, and Yungang Bao. Omnimon: Re-architecting network telemetry with resource efficiency and full accuracy. In *Proceedings of the Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication*, pages 404–421, 2020.

[43] Junjielong Xu, Qinan Zhang, Zhiqing Zhong, Shilin He, Chaoyun Zhang, Qingwei Lin, Dan Pei, Pinjia He, Dongmei Zhang, and Qi Zhang. Openrca: Can large language models locate the root cause of software failures? In *The Thirteenth International Conference on Learning Representations*, 2025.

[44] Wenwei Gu, Renyi Zhong, Guangba Yu, Xinying Sun, Jinyang Liu, Yintong Huo, Zhuangbin Chen, Jianping Zhang, Jiazhen Gu, Yongqiang Yang, et al. Kpiroot+: An efficient integrated framework for anomaly detection and root cause analysis in large-scale cloud systems. *Empirical Software Engineering*, 31(2):28, 2026.

A Algebraic View of the CoreSec Merge Rule

This appendix gives a compact formalization of the multi-agent merge rule used by CoreSec. The algebra is small. It provides determinism, associativity, and order-invariance. These properties are needed for correctness when telemetry arrives at different cadences. PAM's flag semantics are specified procedurally in the original paper [8], the X/Open Single Sign-on specification, and the manuals of operating systems that implement PAM [12]. We are not aware of a prior algebraic treatment. The formalization here is scoped to CoreSec's correctness arguments and does not attempt to characterize PAM in general.

Decision Alphabet

Each agent r produces for each entity e one of three symbols:

- $\mathbf{H_r(e)}$: fresh evidence indicates the entity is healthy,
- $\mathbf{U_r(e)}$: fresh evidence indicates the entity is unhealthy,
- $\mathbf{A_r(e)}$: the agent abstains due to insufficient information.

CoreSec reduces these to a three-state lattice:

$$\text{State}(e) \in \{H, U, I\},$$

where I denotes an indeterminate (unresolved) state. In what follows we drop the agent index r and the entity argument e; the merge operator $\oplus$ defined below acts on the merged states $\{H, U, I\}$ directly.

Merge Operator

For $x, y \in \{H, U, I\}$, define the merge operator $\oplus$ as:

$$x \oplus y = \begin{cases} U & \text{if } x = U \text{ or } y = U, \\ H & \text{if } x = H \text{ and } y = H, \\ I & \text{otherwise.} \end{cases}$$

Interpretation:

- A single decisive failure forces U.
- A healthy state requires unanimous evidence.
- Any disagreement yields I.

This captures CoreSec's principle. One piece of solid failure evidence matters. Healthy requires agreement. Conflict means wait.

Associativity

The operator $\oplus$ is associative:

$$(x \oplus y) \oplus z = x \oplus (y \oplus z) \qquad \forall x, y, z \in \{H, U, I\}.$$

The proof is a simple case analysis:

- If any of x, y, z equals U, both sides reduce to U.
- If all three equal H, both sides reduce to H.
- Otherwise there is disagreement, and both sides reduce to I.

Associativity ensures that ordering of agents does not affect outcomes. It also allows correct streaming updates when some telemetry arrives earlier than others.

Identities and Absorption

Two structural properties explain the stability of CoreSec's merge rule.

Failure absorbs everything.

$$U \oplus x = U \quad \forall x.$$

This guarantees that a single trustworthy failure signal suffices to classify an entity as unhealthy.

Healthy requires consensus.

$$H \oplus H = H, \qquad H \oplus x = I \text{ for } x \neq H.$$

This prevents an entity from being declared healthy when evidence is partial or conflicting.

Abstain is neutral.

Abstentions are removed at the agent layer before merging and do not influence $\oplus$. They neither enforce H nor override U and do not suppress I when evidence conflicts. Silence does not distort RCA decisions.

Relation to Short-Circuiting Evaluation

The operator $\oplus$ resembles classical short-circuit decision stacks. A decisive failure short-circuits to U. Clean consensus yields H. Mixed evidence produces I. CoreSec does not import the full control-flow semantics of systems like PAM, but the analogy helps explain the stability of the merge rule under partial and asynchronous information.

Operational Consequences

This small algebra guarantees:

- **Determinism**: repeated evaluation produces identical results.
- **Convergence**: as telemetry becomes fresh, states move monotonically from I to H or U.
- **Safety**: a single false healthy report cannot mask a real failure.
- **Order-invariance**: proofs hold for streaming, parallel, or out-of-order agent evaluation.
- **Implementation simplicity**: merge is a two-line function in production code.

These algebraic properties are the foundation for CoreSec's correctness. They explain why the system generalizes across heterogeneous telemetry pipelines and multiple cloud deployments.

ARTIFACT EVALUATED
USENIX
AVAILABLE

When Sampling Lies: Trustworthy Performance Profiling for Flat Workloads with Blink (Operational Systems)

Rishikesh Devsot[†] ChenXing Yang[†‡] Yi Fan Yu[†‡] Prabhdeep Singh Soni[§]
Afshin Arefi[§] Bryan Chan[§] Reza Azimi[§] Ding Yuan[†‡]

YScope[†] *University of Toronto*[‡] *Huawei Technologies Canada*[§]

Abstract

Performance optimization of mobile systems is critical for achieving responsive user experiences. However, accurately measuring the effects of such optimizations is challenging. We found that real-world mobile workloads exhibit a flat profile, invoking thousands of short-lived routines that contribute evenly to overall performance with no dominant bottlenecks. Sampling-based profilers such as perf struggle in this setting: skid, shadow effects, and incomplete function coverage can yield systematically incorrect results, not just high variance.

We present Blink, a lightweight instrumentation framework that provides robust coverage for short-lived routines while maintaining low overhead. Blink inserts instrumentation into functions, enabling precise measurements. We show that Blink achieves 99.999% accuracy while incurring 1% overhead. Deployed in a variety of use cases at Huawei, Blink offers a practical and reliable alternative for mobile compiler performance analysis.

1 Introduction

Effective performance optimization starts with trustworthy profiling. Compiler engineers routinely rely on profilers to attribute performance changes to specific functions, libraries, or code regions, particularly when evaluating the impact of fine-grained optimizations. Accuracy is therefore non-negotiable: a profiler with high error rates or limited coverage (e.g., failing to cover the optimization target) is quickly dismissed; worse, a profiler that systematically misattributes performance effects can waste substantial time chasing non-issues.

Within Huawei's HarmonyOS [10] systems compiler team, we design compiler optimizations that reduce CPU cycles, dynamic instructions, and cache misses across diverse workloads. Improvements of even 1% are regarded as meaningful, because they can accumulate into a noticeable increase in responsiveness or battery life.

In the past, we have relied on perf [2], the de facto standard on Linux-based systems, as our profiling tool. Its sampling-based design, which leverages hardware Performance Monitoring Units (PMUs), is widely used to attribute performance changes to specific code regions.

While perf is effective in analyzing performance bottlenecks (e.g., a long-running function), this paper demonstrates that, for an important class of real-world workloads—namely flat profiles with thousands of short-lived functions, the primary workload pattern we face—perf's sampling results can be not merely noisy, but systematically misleading.

We report extensive real experiences that reveal a number of issues with perf on flat profiles. First, when we developed an optimization that leveraged ARMv8.1's Large System Extension (LSE), which should reduce the number of instructions executed, perf instead consistently reported an increase in instruction count. This led engineers to spend weeks investigating before discovering perf's measurements were simply wrong. The root cause is the interplay of skid and the shadow effect. Skid occurs because modern processors cannot deliver PMU interrupts precisely at the instruction that triggered the counter overflow; instead, samples are attributed to downstream instructions. While skid alone introduces random noise, the problem becomes severe when combined with shadow effects: long-latency instructions can block the reorder buffer, effectively shrinking the skid window and biasing samples toward them. In the LSE optimization, we replaced multiple short-latency instructions with a single long-latency atomic instruction, triggering a combined skid-shadow effect in perf. As a result, perf reported consistent yet completely false measurements. While skid and shadow effects have been studied by prior works [5, 8, 9, 13, 20, 22, 29, 31, 34, 37–39], we provide real-world experiences highlighting their impact.

We also experienced two other fundamental limitations of sampling-based profilers. First, sampling provides poor coverage for short-lived functions: even at the highest sampling rates, perf only captures 62% of the functions. Second, high-frequency sampling introduces significant perturbation, including cache pollution (increasing L1 data cache misses by up to 46x) and scheduler distortion.

Motivated by these observations, we present Blink, a lightweight tracing-based profiling framework designed to be trustworthy for flat-profile workloads and short-duration code regions. Rather than relying on interrupts, Blink instruments code to read PMU counters directly at user-defined program points (e.g., function entry and exit). This design eliminates skid and shadow effects by construction and provides near-complete coverage of executed functions.

A key challenge in tracing systems is overhead. Blink addresses this with several techniques. It uses self-patching binary rewriting to dynamically enable and disable instrumentation, ensuring that disabled tracing only incurs the cost of a single jump instruction. It minimizes register pressure and cache coherence traffic by using per-thread data structures and hand-written assembly.

Blink is used regularly by the compiler team during development, and has been integrated into the performance testing framework. Our evaluation shows that Blink achieves over 99.999% accuracy when measuring instruction counts against known ground truth, and introduces negligible user-visible overhead: Blink increases frame drops—the most critical performance metric on smartphones—by less than 1%.

We present several case studies that highlight Blink's practical impact. Even for workloads where perf can produce accurate flat-profile measurements, it often requires a large number of repeated runs (e.g., more than 50) to obtain stable results. In contrast, Blink delivers reliable measurements in just one or two runs. This fast turnaround is crucial for enabling ML-based autotuning of compiler flags, where the training process depends on high-frequency trials and therefore demands a profiling tool to provide rapid feedback. Engineers have also used Blink to diagnose long-standing frame rendering anomalies that were previously impossible to analyze with perf due to their short-lived nature.

2 Experience with Perf

Huawei compiler engineers heavily relied on perf to assess the impact of their optimizations, specifically using it in sampling mode (`record` option). [1] This allows them to attribute low-level performance shifts to specific code regions (modules, libraries, and functions), as optimizations often affect only these localized areas. All our observations and results concerning perf are from its use in sampling mode.

Unfortunately, through painful lessons the compiler engineers have learned that perf's output cannot be trusted due to its inaccuracy in our workload. This is a terrifying reality because we have spent months chasing non-issues–trying to understand why an optimization appeared to fail, only to discover that perf's results were misleading. Worse, a poorly designed optimization that actually degrades performance could mistakenly be deployed to production. Next, we will describe several cases showing how perf's unreliable results have complicated engineers' development process.

2.1 Experimental Setup

Our experiments were conducted on a Huawei Mate 60 Pro smartphone running OpenHarmony OS version 0324. It uses Huawei HiSilicon KIRIN 9000S SoC, which has a total of 8 CPU cores: four "big" (more powerful) cores: one 2.62 GHz Cortex-A710 and three 2.15 GHz Cortex-A710, and four "little" (less powerful) 1.53 GHz Cortex-A510 cores. The phone has 12 GB 3200 MHz RAM. To measure the performance change of compiler optimizations, the compiler team uses a test suite that mimics real-life user interaction through automated clicks and swipes in representative applications.

The Large Picture View in the Camera app (C/LPV) test is prioritized by developers within this suite. This test involves a series of swipes to browse pictures, and its results are more consistent run-to-run because it does not rely on an internet connection. Some of our experiments were run only on the C/LPV test, while others were run on the entire test suite.

The primary optimization target is the system's frame rendering process, `render_service`. Rendering performance is one of the most critical concerns for smartphones because (1) it is one of the most resource-demanding tasks, and (2) it directly influences user experience, as the rendering frame rate reflects the visual smoothness [36]. Unless otherwise specified, all results in this paper are from profiling `render_service`, and we report the average of 10 runs.

2.2 Skid and Shadow Effect on LSEO

We describe a case where misleading output from perf led to weeks of wasted engineering effort pursuing a non-issue. The problem arose after compiler engineers implemented Large System Extension Optimization (LSEO), an optimization designed to reduce instructions for atomic increment/decrement operations, as explained in Listing 1. [2]

To confirm LSEO's effectiveness, the engineers enabled it for a single library, `librender`, and used perf to profile its dynamic instruction counts. However, this supposedly quick sanity check turned into a lengthy investigation: perf *consistently* reports a 6%, or 90 million, *increase* in instruction count for `librender` post-LSEO. Despite the percentage fluctuating, the trend—an instruction increase—was always reported across different runs.

Trusting this contradictory result, the engineers engaged in an extensive "bug hunt", including (1) manually examining `librender`'s binary instruction sequences to verify correct optimizations were applied (we wrote static binary analysis

[1] At Huawei we use *Hiperf*, which is a fork of Linux perf adapted for OpenHarmony. We believe, however, that all the findings in this paper generally apply to Linux perf as they are fundamentally rooted in its design.

[2] In addition to reduced instructions, other benefits of the optimization include (1) no branch prediction cost, (2) less cache coherence traffic and contentions, (3) better forward progress and scalability, and (4) simpler code.

Listing 1 Two versions of ARM instruction sequences for atomically decrementing a value. **Lines 1-6 are before LSEO**: `ldaxr` (Load-Acquire Exclusive) loads the value stored at memory address in `x8` into `x9`, and sets up an exclusive monitor on that address. `sub` decrements the value in `x9` by 1, storing the result in `x10`. `stlxr` (Store-Release Exclusive) attempts to *conditionally* store `x10` back to `[x8]`. The store succeeds only if the exclusive monitor is still valid (i.e., no one else has written to `[x8]` since the `ldaxr`). `w11` indicates whether the store was successful or not. `cbnz` then checks the value of `w11`, and branches back to `retry` if it is non-zero (i.e., store failed). This sequence repeats until the store is successful. **Lines 8-10 are after LSEO**. ARM's Large System Extensions (LSE), introduced in ARMv8.1-A, added an atomic add instruction `ldaddal`. Now we can implement the same atomic decrement logic by using only two instructions, without the need for a retry loop.

```
1  // before lse optimization
2  retry:
3      ldaxr x9, [x8]
4      sub x10, x9, #1
5      stlxr w11, x10, [x8]
6      cbnz w11, goto retry // retry if failed
7
8  // after lse optimization
9      mov x9, #-1
10     ldaddal x9, x8, [x8] // atomic add
```

and found a total of 4,006 static instances of atomic decrement); (2) verifying that the change only affected the atomic decrement; (3) confirming that LSEO was only applied to `librender` and no other libraries (by only swapping the library binary file and flashing the phone to the same state beforehand); and (4) brainstorming extensively on any possible side effects that could increase the instruction count. Only after these steps, combined with the observation that perf's non-sampling mode [3] indeed reported a 13M (.4%) decrease in total instruction count *for the entire process*, we concluded that perf's sampling result was inaccurate.

The root cause is the combined effect of skid and shadow effect. Perf implements sampling by configuring the Performance Monitoring Unit (PMU) counter to overflow at a user-specified frequency (default 4kHz). A PMU counter overflow triggers an interrupt, allowing perf to collect profiling information within the interrupt handler.

Skid refers to the imprecision between the instruction that actually causes the PMU counter to overflow and the Instruction Pointer (IP) recorded when the interrupt is handled. ARM-compatible processors cannot deliver the interrupt precisely at the triggering instruction [8]. Instead, the interrupt is delivered a few instructions later, so the recorded IP often corresponds to a nearby downstream instruction.

If skid were the only factor, it would introduce only random noise. For a consistent skid window (e.g., 100 instructions), the sampled IP would be, on average, 100 instructions after the event's source. With a large sample size, this noise should not lead to an incorrect statistical conclusion. However, when skid is compounded with *shadow effect*, the result could be *consistently wrong*.

The *shadow effect* occurs because long-latency instructions remain in the processor's in-flight instruction buffer (e.g., the reorder buffer) for extended periods [8]. Perf samples the instruction at the head of this buffer (the next one to retire). Since instructions retire in program order, a long-latency instruction at the head blocks all subsequent instructions from retiring, effectively shortening the skid window and significantly biasing the sampled IP toward long-latency instructions.

This is the case for the LSEO. The atomic add instruction `ldaddal` in the LSEO code is a long-latency instruction because it must gain exclusive ownership of the cache line during the entirety of the operation, often implementing the same retry mechanism as underlying micro-operations.

Figure 1 shows this result. It plots the instruction count measured by perf for the top 8 libraries in the `render_service` process. Here we run perf at its default 4kHz sampling rate. [4] Notably, while `librender` reported 30.9M increase in instruction count post-LSEO, two *unchanged* libraries–`librender_server` and `ld-musl`–reported a combined decrease of 38.7M. Analysis of the sample call stacks confirms that functions in `librender_server` and `ld-musl` call functions in `librender`. The observed change is attributed to a shadow effect from `ldaddal`: prior to LSEO, instructions frequently "skidded" from `librender` into `librender_server` and `ld-musl` at function return. After LSEO, samples that previously skidded past `librender` are now captured at `ldaddal`.

Skid effect can be almost entirely eliminated by CPUs that support *precise event delivery* [8]. Examples include Intel Precise Event-Based Sampling (PEBS) [21] and AMD Instruction-Based Sampling (IBS) [14]. They record the exact IP that caused a PMU counter to overflow.

ARM introduced an optional Statistical Profiling Extension (SPE) [28] starting with v8.2 to mitigate skid. Unlike interrupt-driven sampling, SPE periodically samples and records microarchitectural states. However, the HiSilicon Kirin 9000S cores currently lack support for SPE.

Crucially, even with SPE, our specific issue remains unresolved. This is because SPE does not support sampling based on instruction count (i.e., one sample every N retired instructions) [1]. Implementing instruction count sampling would be costly, as it requires complex and potentially disruptive deep hooks into the pipeline. Instead, SPE sampling is based on micro-operations [1]. Consequently, a single, long-running

[3] perf stat measures hardware and software performance counters over the entire duration of a program

[4] Note that a higher sampling rate on perf does not address the skid and shadow effect.

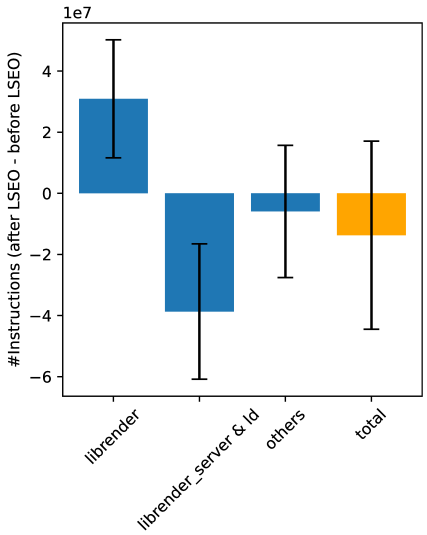


Figure 1: Difference in mean dynamic instruction count (y-axis) after and before LSEO for `librender`, the combination of `librender_server` and `ld-musl`, other libraries, and total for the entire process. Total for the entire process is computed with perf stat, whereas the others are computed using perf record (sampling). Shown in error bars is the geometric sum of the 2 standard deviations.

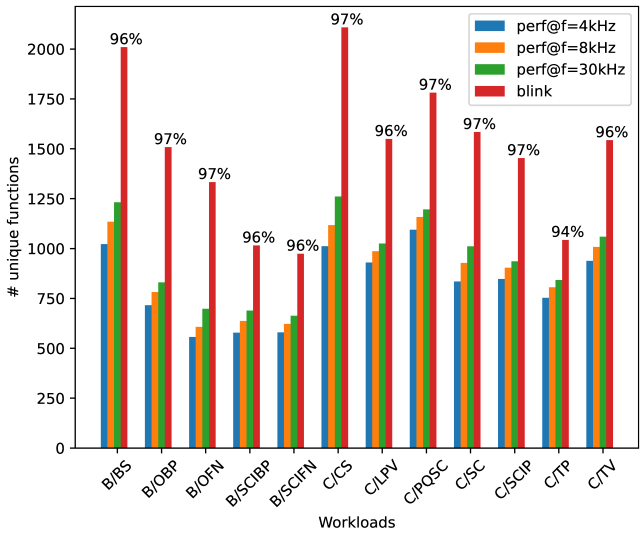


Figure 2: Unique functions covered by perf record at different sampling frequencies and Blink. Blink's coverage is reported on top of each workload.

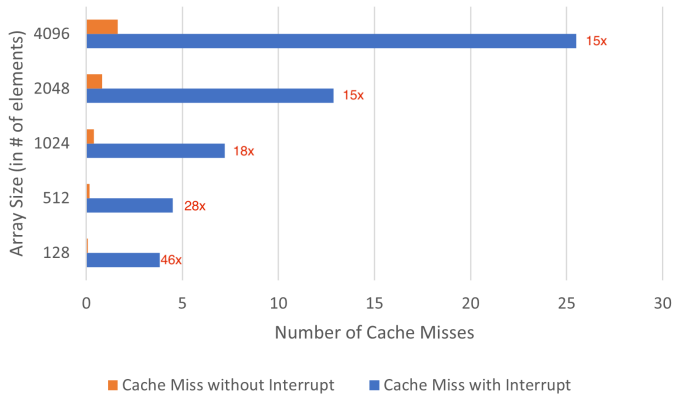


Figure 3: Cache misses of one `read_random()` invocation with and without timer interrupts.

instruction that involves many micro-operations will be sampled more frequently, resulting in a similar issue. In contrast, Intel PEBS and AMD IBS would resolve our issue because they support instruction count sampling.

Precise instruction count can also be obtained with Intel's Last Branch Record (LBR), which records all taken branch addresses. Unfortunately, our phones do not have this hardware support. In addition, this approach is limited to a buffer size. More importantly, LBR can only provide instruction counts, but engineers are often interested in other PMU events.

2.3 Low Coverage and Disturbance

Perf's sampling inherently limits the coverage. Figure 2 illustrates the coverage of unique functions across our 12-test suite. We tested various sampling rates for perf, ranging from 4kHz up to 30kHz.

At 4kHz, perf can only capture 54% of the functions on average. Even at 30kHz, coverage is still limited to 62%. While this is acceptable when profiling for performance bottlenecks, it poses a major obstacle when analyzing a flat profile or profiling functions with very short execution times, which is the case for the rendering workload where the execution time is often equally split across thousands of different short-lived functions. In contrast, Blink achieves a coverage of 96%.

Sampling at a higher frequency is often impractical. We ran an experiment where we increased the sampling rate from 4 kHz to 40 kHz. While we expected a 10x increase in the number of samples, we only observed about a 7x increase. This is because perf monitors its CPU usage and dynamically adjusts the actual sampling frequency to remain below a certain threshold (25% by default).

Furthermore, a high sampling rate can also significantly perturb performance. This can cause some frame rendering to take longer, missing deadlines and causing UI jank. We also observed that the increased CPU demand from the higher sampling frequency can cause threads typically scheduled on a little core to compete for a time slice on a big core. This shift completely alters the scheduling dynamics and, consequently, the entire system's performance characteristics.

2.4 Cache Pollution

Perf's sampling mechanism—interrupts—could introduce perturbation by polluting processor caches. We designed an experiment using a function `read_random()` that reads from an array randomly. Each element is cache line aligned (64 bytes). We ensured the array elements were resident in the L1 data cache before the function invocation. We read randomly because, for sequential reads, the hardware prefetcher would eliminate the cache misses.

Directly using perf sampling was not feasible as it utilizes the same registers needed to measure L1-Dcache misses. To simulate the effect of a sampling interrupt, we register a 4 kHz timer, which triggers hardware interrupts that invoke a signal handler. When the signal is received, the handler invokes `read_random()` to read the array one time, and we record the number of cache misses. `read_random()` is invoked a total of X times where X is the number of timer interrupts. In comparison, for the baseline (without interrupt), we invoke `read_random()` X times and compute the average cache misses for each invocation.

Figure 3 shows that, across different array sizes, the interrupts led to a dramatic increase in L1 D-cache misses, ranging from 15x to 46x. Such cache misses can alter the timing of loads and stores, potentially perturbing sensitive workloads. For instance, in the pre-optimized LSE snippet in Listing 1, sampling interrupts can increase load/store latency on the sampled CPU. Consequently, a CPU not sampled by perf is more likely to win the contention, thereby increasing the number of retry attempts on the sampled CPU.

3 Blink: A Trustworthy Tracing Tool

We design and implement Blink, an efficient and *trustworthy* performance profiler. Blink is a tracing-based profiler: it injects code snippets that read PMU counters before and after the code block being profiled, and records the counter values in a memory buffer that is later flushed to a trace file. By relying on tracing instead of interrupt-based sampling, it avoids the skid and shadow effects, and achieves full function coverage as users can trace any code region.

By default, Blink traces every function by inserting paired trace points at their entry and exit. It is also general: it can be used to trace any arbitrary code blocks. Developers may provide a list of functions to exclude from tracing.

Using Blink involves three steps: (1) Compile-time instrumentation: Target binary is compiled with appropriate flags to enable instrumentation at function entry and exit. (2) Execution and data collection: Each time an instrumented function is executed, Blink records a pair of tuples—`<tpID, PMU value>`—one at entry and one at exit, where `tpID` is a unique tracepoint ID. (3) Post-processing: The trace files are analyzed to compute the difference between the entry and exit PMU readings and to produce per-function summaries. Blink also generates flamegraphs [19], a feature used by engineers.

Blink can operate in two modes: sampling or always-on. In sampling mode, after N samples are collected within a thread for a particular function, the instrumented tracing code will self-disable its tracing via a carefully designed binary rewriting mechanism. A control thread will then periodically re-enable the tracing.

Next, we explain the design and implementation details.

3.1 Lightweight Tracing

We carefully designed our tracing logic to ensure its high efficiency. Crucially, because users expect zero overhead when tracing is disabled, we design a self-binary-rewriting mechanism to enable and disable tracing. We further optimize the tracing overhead by minimizing the use of shared variables and synchronization, and by using hand-written assembly to tightly control register usage.

An important concept in Blink is a *trace point*–a static code location to be instrumented. For each traced function, the trace points include its entry block and all its exit blocks (a function may have multiple exit blocks). Blink assigns a unique ID to each function and to each of its trace points.

Listing 2 shows the instruction sequence embedded at each trace point. It saves the required registers to the stack before calling the `tracing` function that performs the actual PMU reads. Specifically, lines 3-5 save registers `x0`, `x1`, `x8`, `x16`, `x29` and `x30`. This is much less than the full caller-saved set (`x0-x15, x29, x30`) specified by the AArch64 ABI. This is safe because these are the only registers used by `tracing`.

Lines 7-10 store the two arguments to be passed into `tracing`, namely `fn` and `tpID`, into `x0` and `x1`. Lines 12-14 store the address of `tracing` to `x16` before branching to it, effectively calling `tracing` with `fn` and `tpID` as arguments. Lines 16-18 restore the registers after `tracing` returns.

The first `nop` instruction is the target for binary rewriting. To disable tracing, we overwrite this instruction to a jump (i.e., an unconditional branch) to skip the following 13 instructions (`b +13`). As a result, when tracing is disabled, the only added overhead is a single jump instruction—there is no need to check for a flag and execute a conditional branch.

Listing 3 shows `tracing` logic (shown in C but implemented in assembly). It takes two arguments: a pointer to a structure `FData` and `tpID`, the trace point ID. Each function has a `FData` structure that stores its function ID `fID`.

We first obtain a pointer `tls` to the thread-local storage (TLS). The TLS region stores an array of counters (`tls->counters`), one for each function, and the buffer to store the profiling information (`tls->buf`).

Lines 5-11 in Listing 3 disable the tracing at the current trace point if its counter exceeds $2N$. Specifically, we obtain the IP of the `nop` instruction shown in Listing 2 in the caller, which is located at 11 instructions above the return address,

Listing 2 Instructions inlined at each trace point.

```
nop  // if disabled, rewritten to b +13

stp x0, x1, [sp,-16]!      // save args
stp  x8, x16 [sp,-16]!     // save scratch/x16
stp x29,x30,[sp,-16]!      // save FP/LR

adrp x0, fn:HI
add  x0, x0, fn:LO      // fn arg0
mov  w1, tpID:LO
movk  w1, tpID:HI LSL 16    // tpID arg1

adrp x16, tracing:HI
add  x16, x16, tracing:LO
blr  x16                   // call tracing

ldp x29,x30,[sp],#16       // restore FP/LR
ldp x8, x16,[sp],#16
ldp x0, x1, [sp],#16       // restore args
```

Listing 3 The function body of `tracing` that is invoked to collect PMU counters.

```
void tracing(FData* fn, int tpID){
  int fID = fn->fID;
  TLS* tls = emuTLS(); // thread-local storage
  // Disabling logic
  if (tls->counters[fID] >= 2*N || fn->disabled == true) {
    asm("mov %0, lr" : "=r"(ret)); // return addr
    *(ret - 11*4) = BRANCH_13; // rewrite with "b +13"
    tls->counter[fID] = 0 // reset
    fn->disabled=true;
    return;
  }

  asm("mrs %0," PMU_REGISTER :"=r"(pmu));
  record(tls->buf, tpID, pmu); // write to buf
  tls->counters[fID]++; // increment counter
  if(isfull(tls->buf))
     flush(tls->buf);  // flush
}
```

and rewrite it to an unconditional branch `b +13`, effectively skipping the compiler-injected snippet in Listing 2.

Lines 13-18 read the PMU counter value, store a tuple `<tpID, pmu>` into the per-thread buffer, and increment `tls->counters[fID]`. If the buffer is full, it will flush it to persistent storage.

The reason we compare the counter value to $2N$ instead of N is that trace points for a function are executed in pairs (i.e., the entry and exit of a function). Both share the same `FData`, hence the counter of this function `tls->counters[fID]` gets incremented twice per invocation. Consequently, each function invocation contributes two entries into the buffer. During post-processing, we pair these records and compute their difference to determine the performance of the function.

The reason that we are differentiating `fID` and `tpID` is that they represent two (subtle) distinct identities in the instrumentation. `fID`, stored in `FData`, uniquely identifies a function, whereas `tpID` uniquely identifies a trace point. A single function may have multiple trace points (i.e., multiple exit blocks), yet all of them correspond to the same `fID`. `fID` allows us to properly count the total number of function invocations, while `tpID` enables us to distinguish between entry and exit events during post-processing. This distinction becomes essential in cases such as recursion, where multiple nested invocations of the same function are active simultaneously; having `tpID` ensures we pair the correct entry and exit points.

This design minimizes variable sharing across threads, avoiding the need for synchronization and reducing cache-coherence traffic on shared variable read/write, which can be as costly as synchronization itself [12]. Both the buffer and the counters are stored per-thread. The only variable shared across threads in Listing 3 is `fn->disabled`, which indicates whether tracing is disabled for that function. Because multiple threads could be running this function, the `FData` is shared. However, this `disabled` flag is updated rarely, so the resulting cross-thread coherence overhead is negligible.

Enabling Tracing

Tracing is re-enabled periodically by a control thread waking up every T ms. It goes through every function's `FData` and examines the `disabled` flag. If a function's flag is set, the control thread enables tracing by rewriting the first instruction in Listing 2 from `b +13` back to `nop`. The address of this instruction is stored in the FData for each tracing function. If a function has n trace points, its FData contains n corresponding addresses. The control thread iterates over all these addresses and restores the initial instruction at each trace point, thereby re-enabling tracing for the entire function.

As a result of this design, for every thread Blink will collect at most N samples for every function for every T ms interval. This ensures high coverage on those rarely executed functions, yet the cap N keeps the tracing overhead bounded within each interval, regardless of how frequently a function executes. This design achieves a practical balance between limiting overhead and maintaining broad sampling coverage.

Blink has a feature to enable or disable tracing signals. Initially, all traces are disabled, and the control thread is only created when a signal is delivered. This feature is useful in practice because it allows developers to skip tracing the process startup phase.

3.2 Addressing Skid

Because modern superscalar CPUs execute instructions out of order, the PMU counter read instruction can be reordered. This can cause a similar skid effect. Blink supports an optional feature that adds an instruction fence (`ISB`) before every PMU read to enforce correct order. In practice, engineers only use this feature when measuring dynamic instruction count.

In addition, the injected instrumentation itself introduces a systematic measurement offset: instrumentation adds additional instructions that are inevitably captured in the measurement itself. This affects instruction count measurement. By manually counting the included instructions, we know this offset is 44 instructions, and we correct it in our post-processing.

3.3 Tree versus Self

The difference of the PMU counters between the entry and exit of a function would include the performance of all its callees (and their entire call chains). We refer to such measurements as the "tree" measurement. In contrast, a "self" measurement refers to only the function without any callees.

Blink supports both measurements. In theory, supporting "self" is straightforward because post-processing reconstructs callee relationships. By tracking entry and exit events, we can determine which functions are active at any point in time and subtract the PMU contributions of each callee from its caller, thereby isolating the caller's own ("self") cost.

In practice, however, not every function is instrumented. For any uninstrumented function, we would need to instrument immediately before and after its call instruction (`bl`) to subtract its contribution. For this feature Blink also allows developers to specify a list of call sites to exclude from instrumentation. This is used to skip a small number of very short helper functions that are invoked frequently and would otherwise cause a substantial slowdown if every call site were instrumented. (These short helper functions are not part of the system libraries we aim to trace and therefore are not included in our tree-based measurement.)

3.4 Other Implementation Details

Handling context switches and interrupts. The PMU counter on a CPU core increases monotonically, independent of context switches or interrupts. Therefore, when a thread is switched out, Blink must save its current PMU value and later restore it when the thread is switched back in. This save/restore logic is already handled by the kernel's perf subsystem in the same code path where the kernel saves and restores general-purpose registers. To leverage this capability, Blink makes a one-time `perf_event` system call at the first trace point executed by each thread.

Compiler integration. We cannot implement Blink's instrumentation as a compiler pass, because later optimization passes can modify the instrumented instructions. For example, instrumentation code might be reordered past the target; instrumentation might also cause functions that would otherwise be inlined to be no longer inlined because of the added instructions, hence the instrumented version may exhibit different performance behavior compared to the uninstrumented version.

To make sure the binaries are still comparable after instrumentation, we inject a placeholder at the Machine Intermediate Representation (MIR) stage, the last stage before IR becomes assembly. Naturally, MIR compilation passes execute after most major optimization passes. When MIR is lowered into assembly in the later stages, we replace the placeholder with actual assembly instructions.

Flush control. The per-thread buffer size is configurable. The default size is 10,000 records (i.e., `<tpID, PMU value>` tuples), which is determined experimentally such that we don't flush excessively, around once every 100ms. Each tuple is 16 bytes long, hence the buffer size is 160KB per thread.

Emulated TLS. For systems with native TLS support, a pointer to the Thread Control Block (TCB) is accessible using a system register (TPIDR_EL0 in ARMv8). Supporting this feature requires support from the linker, threading system libraries, and the architecture (providing a thread-specific system register).

Huawei's OpenHarmony currently does not support native Thread Local Storage (TLS). So we have to use emulated TLS (`emuTLS`) to obtain a pointer to a thread-local buffer. Emulated TLS is much slower than reading a register because it relies on POSIX thread API calls, specifically `pthread_get_specific` and `pthread_set_specific`, to emulate thread-local storage by using thread-specific keys to locate thread-specific data. In `tracing`, we made sure that `emuTLS` is only called once by reserving a register to save its return value.

Determining the PMU register. Unlike perf, Blink is designed to read the PMU counter values in the user space. However, `perf_event` does not specify which PMU counter it will use to record the event for user space programs, because in perf reading the PMU counter is done by the kernel module. On Huawei Mate phones, which provide eight PMU counters, we empirically found that the first five are reserved for kernel profiling. Blink therefore defaults to use PMU counter index 5 (`PMU_REGISTER` at line 11 of Listing 3 is a macro defined as `PMEVCNTR5`). For configurations where this counter is also reserved, we provide a script to detect the available PMU counters; users can then pass the appropriate index to Blink.

4 Evaluation

We aim to answer the following questions in our evaluation:

- *Correctness*: Can Blink profiles be trusted? How accurate are the PMU reads at function entry and exit?
- *Performance*: What is the performance overhead of Blink? Does Blink's overhead affect user-perceived metrics such as frame drops?
- *Utility*: How do developers use Blink? Does Blink's profiling data lead to more insights compared to perf?

Our experimental methodology follows the setup described in Section 2.1. Unless otherwise specified, we run Blink in

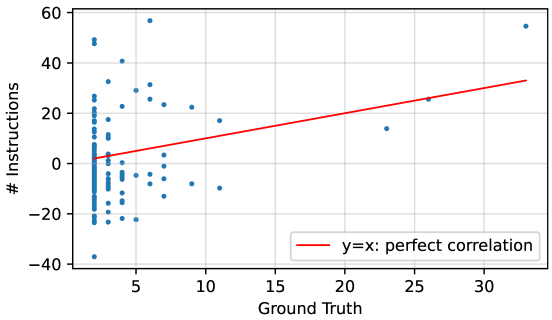


Figure 4: Per-function DIs: Blink *without* `ISB` vs. ground truth. Each point represents a unique function. Only 157 of the 1,112 static functions were invoked, so the figure contains 157 points.

sampling mode with $N = 30$ and $T = 400$ms, meaning that for every 400ms, each function running in each thread will collect at most 30 samples. We enable `ISB` and apply offset correction only when using Blink to measure dynamic instruction count; for all other PMU events we remove `ISB` and do not apply any post-processing correction.

4.1 Correctness Evaluation

To assess Blink's correctness, we compare Blink's reports against ground truth. Among the various PMU events, we use the **dynamic instructions (DIs)** event for comparison.

We selected 1,112 functions in `librender` that contain neither branches nor function calls. For such straight-line functions, DIs executed are known exactly. We use Blink to measure the DIs only for these functions via an include-list. We then ran the Camera large-picture-view workload `C/LPV` 100 times, and collected a combined total of approximately 177 million function invocations for the functions we selected. Among them, only *568 (less than .001%)* measurements showed any deviation from the ground truth. In all remaining records, Blink reported the exact instruction count as the ground truth, achieving an accuracy of $> 99.999\%$. These rare mismatches are likely due to infrequent counter glitches that occur when threads are interrupted.

Accuracy without `ISB`. Although engineers typically insert `ISB`s when measuring DIs, they often use Blink without `ISB`s when collecting other PMU events. Therefore, we evaluate Blink's accuracy in the absence of `ISB`s.

Figure 4 shows Blink's DI measurement without `ISB`s. The x-axis shows the ground truth instruction count of each function, while the y-axis shows the DI reported by Blink. Perfect accuracy corresponds to points lying on the red diagonal.

Across all functions, the mean error is -3.12 instructions, with a 95% confidence interval of $[-5.38, -0.86]$. This indicates that Blink's DI measurements without `ISB`s under-report the instruction counts by only a few instructions on average.

Jank	Baseline	Blink
Jank0	183.9 (65%)	186.8 (64%)
Jank1	68.5 (24%)	79.3 (27%)
Jank2	14.1 (5%)	14.5 (5%)
Jank3	14.1 (5%)	10.1 (3%)
Jank4	0.63 (0.2%)	0.38 (0.1%)
Jank5	1.75 (0.6%)	0.75 (0.3%)
Total Frames	283.0 (100%)	291.8 (100%)

Table 1: **Average jank breakdown for the gallery swipe workload.** Jank0 indicates the frame was rendered on time (higher is better); Jank$N$ ($N > 0$) represents N consecutive frames were delayed (lower is better). Higher Jank levels correspond to worse user experience.

4.2 Overhead and Perturbation

A profiling tool should not introduce overhead that causes user-perceived slowdown or perturbs application behavior. To evaluate this, we measure frame delays, also known as "jank". Table 1 reports the distribution of jank levels over 10 runs of `C/LPV`. We compare two configurations: the **Baseline**, which runs the workload without any instrumentation, and **Blink** configured *with* `ISB` enabled.

The jank distributions are similar in both configurations. Blink introduces only a 1% decrease in Jank 0 events (i.e., timely rendering), which is well within the acceptable margin (typically $< 2\%$). Interestingly, we also observe that some higher-level janks (e.g., multi-frame delays) shift to lower levels (e.g., single-frame delays) when running with Blink. This effect is likely due to workload fluctuations rather than a real performance improvement.

Micro-benchmark. We designed a simple micro-benchmark where we call a function 10 million times, and use Blink to trace it (without `ISB`). Each trace point executes an average of 140.9 instructions, and takes 38.7 cycles.

Memory overhead. We measure memory overhead using the peak resident-set high-watermark of `render_service`. We compared the Blink-instrumented runs against the uninstrumented baseline using the default buffer size (100k elements). When evaluated over 12 test workloads, the mean, median, and maximum overhead are 417.9 KB, 1,339.6 KB, and 5,666 KB, respectively. This overhead is dominated by the per-thread buffer allocation.

4.3 Use Cases and Experiences

Blink is available for all Huawei developers, but primarily used by the compiler engineers, who use it on flat-profile workloads to analyze short-lived functions. Blink has also been integrated into the performance testing framework so engineers can easily use Blink to debug any performance

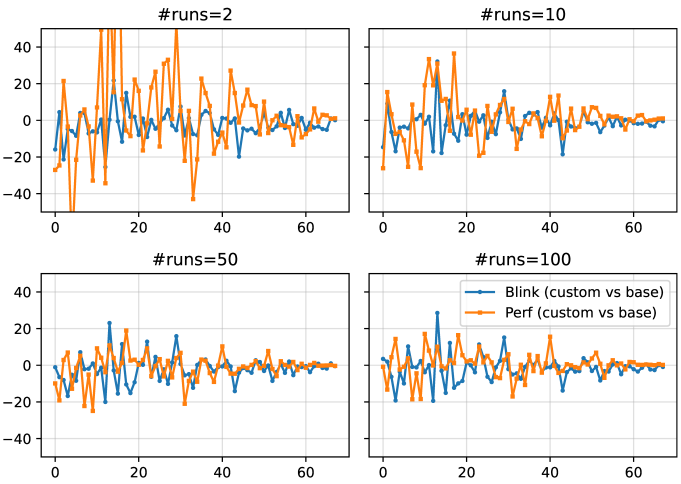


Figure 5: Comparison between perf and Blink. The x-axis lists functions (sorted by total cycle count), and the y-axis shows the percentage change in cycles when tuning from the `base` to `custom` compiler settings.

regressions. Next, we present four use cases where engineers find Blink particularly helpful.

4.3.1 Auto-tuning

One of the benefits of Blink is that it can collect trustworthy measurements with a small number of runs. Auto-tuning is such a use case, where we programmatically tune compiler flags, such as function inline threshold, in order to optimize the performance of a workload. These compiler adjustments can cause subtle and sometimes unpredictable performance shifts, so the tuning process must be guided and validated by reliable per-function profiles. The compiler team aims to train a machine learning model to automate these tuning decisions, which requires stable, fine-grained per-function data. Unfortunately, perf record is not suitable for this purpose due to its high variance across runs.

Figure 5 compares perf with Blink on 80 performance-critical functions in `librender` affected by a set of flags. The 'base' configuration uses the default compilation flags, whereas the 'custom' configuration modifies their values. [5] We measure per-function % cycle difference using both tools and repeat the workload between 2 and 100 times.

Perf exhibits substantial variance when only a few runs are available, stabilizing only after 50 or more repetitions. In contrast, Blink's results are stable after two runs.

This stability is critical for ML-based auto-tuning. The search space for compiler flags is large, and ML training requires evaluating many flag configurations. If each evaluation required 50+ repetitions (as with perf), the total cost would be prohibitive. With Blink, a workload generally needs to be run at most twice per trial, dramatically reducing training time and enabling exploration of a much larger configuration space.

[5] These flags are -inline-threshold, -intra-scc-cost-multiplier, -earlycse-mssa-optimization, -jump-threading-threshold, and -max-speculation-depth.

4.3.2 Frame Jank Investigation

Here we describe how engineers successfully triaged a long-standing performance bug using Blink. The `C/LPV` workload occasionally exhibited a frame that took significantly longer to render at the beginning of a swipe gesture.

We first provide some background on how frame rendering works on modern smartphones. A smartphone display refreshes at a fixed rate. On the Mate 60 Pro, the highest refresh rate is 120 Hz, meaning the screen renders a new frame every 8.33 ms. The display subsystem exposes this refresh rate via hardware VSync events. These events drive the system's software VSync callbacks, which are delivered to the foreground app to trigger frame rendering for the next display cycle. The app sends this frame to the system's rendering service to render it on the display. If the entire procedure takes longer than 8.33ms, the rendering is delayed; in severe cases several frames will be dropped, causing visible stutter.

Debugging with perf. Before Blink was available, engineers tried to debug this issue with perf. However, perf's sampling granularity is insufficient when rendering a frame only takes about 8.33ms. The highest sampling rate we can practically get from perf is between 30kHz and 40kHz—even if we set it to 100kHz, the maximum allowed sampling rate, it internally throttles due to overhead control. Even at 40kHz, we can only collect a few hundred samples per rendering window. When we compared the samples of a delayed frame with a healthy frame, the profiles looked completely different, with each function only sampled once or twice. We are essentially comparing two flat profiles that look very different. We concluded that profiling with perf is not suitable for short-duration tasks such as frame rendering.

Debugging with Blink. For this case, we used Blink to instrument functions in the rendering path and measure their cycle count. Tracing was enabled and disabled directly within the software VSync handler, giving us a clean trace for each frame-rendering interval. We further produced a differential flame graph [18] by comparing the healthy and delayed renderings. This analysis revealed a 10% increase in the execution time of the `execute` function in `libArk`. `libArk` is Huawei's managed-code runtime (supporting JavaScript, Java, and ArkTS), and `execute` is its main entry point. This clearly indicates that the slowdown lies in the application or in the libArk runtime, rather than in the system's rendering pipeline.

Unfortunately, Blink cannot trace the callees of `execute`, because they are JIT-generated code fragments produced by a library with a specialized compilation pipeline and non-standard ABI. Such code cannot be linked against static li-

	Avg. DIs	# Samples	Total DIs
Before LSEO	352.8	2,250	793,774
After LSEO	349.8	2,280	797,541

Table 2: Averaged DIs (dynamic instructions) per invocation of `update` before and after LSEO.

braries, while Blink's `tracing` function must be statically linked for instrumentation. As a result, callees such as `stub.an` (JIT-compiled JavaScript fragments) cannot be instrumented.

Nevertheless, Blink's findings were sufficient to triage the issue, and the bug was assigned to the Camera/Gallery app team and the libArk runtime team.

4.3.3 LSEO

Once Blink became available, one of the first cases engineers used it for was to confirm that LSEO indeed reduced instruction count. Specifically, we focus on the `update` function, which statically contains 26 LSEO opportunities, the most among all functions in the `librender` library. Interestingly, perf also reports the largest change for this function, claiming a 20% *increase* in DI after applying LSEO. This makes `update` an ideal case for evaluating LSEO's real effect and for contrasting perf's misleading measurement.

Table 2 presents the averaged DI for `update`, measured by Blink with `ISB` enabled. According to Blink, LSEO yields an average *reduction* of 3 instructions per invocation.

4.3.4 Test Suite Integration

We integrated Blink into Huawei's phone performance testing framework. Next we report results from the 12 tests in 2 applications (Camera and Browser).

As shown in Figure 2, a major advantage of Blink is its high function coverage: Blink is able to cover 96% of unique functions, whereas perf covers at most 62%. This is particularly valuable for compiler engineers, as optimization targets often lie in short-lived functions that appear flat or sparsely sampled under perf. Blink cannot cover all functions because some helper routines and stubs are generated by the compiler at link time, which occurs after the MIR stage where Blink inserts instrumentation.

Next we compare total cycle measurements from Blink and perf for all functions covered by both tools. For this experiment, Blink's tracing was configured to be always on.

Figure 6 shows the comparison of Blink and perf. Each data point represents a unique function covered by both tools. Its x- and y-axis coordinates are the measurements from perf and Blink, respectively. If the two measurements were identical, the point would lie on the red $y = x$ line. Data points above the perfect correlation are when Blink reports higher cycle count than perf, whereas those below are when Blink's measurements are lower.

Browser (B)			
Work	Slope	Y-int.	R^2
BS	1.66	1.75e5	0.9995
OBP	1.54	3.04e5	0.9991
OFN	1.53	-1.98e6	0.9971
SCIBP	1.74	-3.78e5	0.9986
SCIFN	1.32	8.11e3	0.9993

Camera (C)			
Work	Slope	Y-int.	R^2
CS	1.37	3.32e5	0.9998
LPV	1.25	6.14e5	0.9999
PQSC	1.08	9.01e5	0.9997
SC	1.19	7.17e5	0.9997
SCIP	1.23	-3.30e4	0.9999
TP	1.23	3.05e5	0.9999
TV	1.11	3.24e6	0.9941

Table 3: Details of linear regression results

Table 3 provides detailed linear regression statistics. Slopes closer to 1 indicate greater similarity between perf and Blink.

Because the true cycle counts are unknown, we cannot determine which tool is strictly accurate. However, several observations are notable.

First, for functions with higher total cycle counts, the two tools tend to converge. The deviations from the $y = x$ line primarily occur for functions with small cycle counts. Given that perf's sampling design indicates that its accuracy is higher for long functions but lower for short functions, whereas Blink's tracing design should not be affected by function length, this result likely indicates that Blink's result is correct.

Second, the normalized standard deviation (NSD) results also show that Blink's result is more reliable than perf's. A positive NSD difference (i.e., yellow color) indicates perf has higher noise than Blink, whereas a negative value (i.e., blue) indicates the opposite. In 8 out of the 12 tests, perf demonstrated a higher standard deviation than Blink (as indicated by the NSD spectrum going to a higher positive value than the negative). There is only one test, B/OFN (Browser/Open-First-News), [6] where Blink showed higher noise than perf.

Finally, as shown in 9 of the 12 tests in Table 3, the y-intercept is positive, meaning that the regression line lies above the perfect-correlation line, indicating that Blink reports higher cycle counts than perf, which is expected because Blink's instrumentation introduces a fixed overhead of 38.7 cycles per collected tracing point, and we do not subtract this overhead during post-processing. Across all workloads, the R^2 values are high, indicating excellent fit quality.

5 Related Work

The problems associated with Event-Based Sampling (EBS)–including sampling skid and shadowing effect–have been extensively discussed in prior works [5, 8, 9, 13, 20, 22, 29, 31, 34, 37–39]. While these works provided valuable theoretical analysis, we describe experiences from real-world workloads to show the significance of these issues.

[6] This workload opens the first news article on m.baidu.com, and its fluctuations are affected by Wi-Fi quality

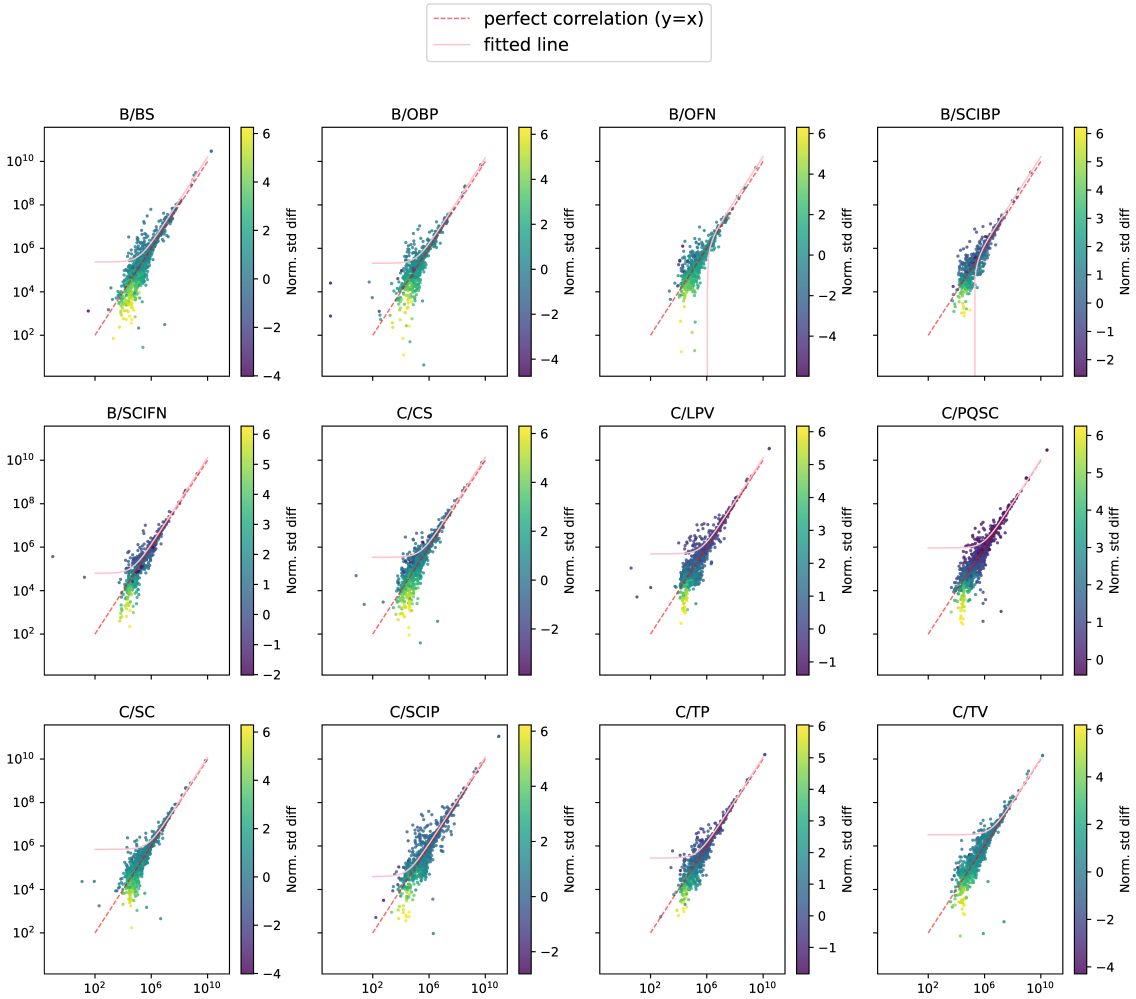


Figure 6: Log-log plot comparing total cycle count for perf (X-axis) and Blink (Y-axis). Linear regression in pink; perfect correlation line $y = x$ shown in red (dotted). Coloring represents the Normalized Standard Deviation difference based on 10 runs, $(std_{\text{perf}} - std_{\text{Blink}})/(0.5(avg_{\text{perf}} + avg_{\text{Blink}}))$. More yellow indicates that perf has higher standard deviation.

Some [5, 23, 26, 35, 37] enhance the accuracy of EBS by mathematically modeling the error due to skid and shadowing effects. For example, Xu et al. [37] quantify the value of skid in number of cycles and then use that to fix the sampled instruction distribution. These are orthogonal to Blink as both tackle the problem of EBS inaccuracy.

Many tracing profilers leverage compiler instrumentation. Compared to Blink, these approaches are often much simpler as they only support measuring execution time (rather than PMU events) and lack fine-grained sampling control. XRay [7] uses Clang/LLVM to insert instrumentation. While it also uses binary rewriting to disable or enable tracing, the mechanism is simpler than Blink's: it patches a no-op to a call to the tracing function, which takes no arguments because its tracing logic is minimal. Blink needs to pass in call-site-specific information as arguments to `tracing`, which requires a more sophisticated binary rewriting strategy. Moreover, XRay incurs 20%-40% overhead.

Hubble [24] is a function tracing tool for Android applications written in managed languages (Java and Kotlin). It records a time stamp and a function pointer at the entry and exit of every function. Compared to Blink, Hubble is much simpler—it is always-on and does not support PMU events, self-patching, or tracing native functions. However, Hubble is more efficient and lightweight than Blink and it is used on users' phones in production.

BOLT [30] is a binary optimizer to relocate functions after PGO (Profile-Guided Optimization) and LTO (link-time optimization) are performed. The authors argued that a sampling-based approach has low overhead and doesn't require recompilation, hence it should be preferred over tracing. However, we show a number of issues with sampling-based approach

in flat-profile workloads.

There are a number of other function call profilers. `gprof` [17] collects timing information through sampling and call graph information through tracing. GProfNG [27] extends `gprof` to work with PMU counters. However, GProfNG uses sampling instead of instrumentation. `strace` [3] and `ftrace` [33] allow users to profile system calls and kernel behavior.

There are a number of tracing tools designed to profile a large distributed cluster, including Project5 [4], MagPie [6], X-Trace [16], Dapper [32], Fay [15], ÜberTrace [11], and Pivot Tracing [25]. In comparison, Blink focuses on low-level profiling for smartphone workloads.

6 Limitations

Blink has several limitations. First, it is currently only used in testing and development environments, not on end-users' devices. A production-grade profiler would require far more comprehensive overhead evaluation to demonstrate that its *worst-case* overhead remains within 1–2% [24]. Further, Blink's self-patching mechanism requires access to a rooted and unlocked smartphone, which end-users do not have access to. Second, although Blink achieves 96% coverage across all functions—substantially higher than perf—a small subset of functions cannot be instrumented because they follow different ABIs. Third, Blink currently targets ARM64. Porting Blink to other architectures requires rewriting its hand-written ARM64 tracing instructions (Listing 2 and Listing 3) for other ISAs, which takes non-trivial effort. Finally, Blink's design is closely tied to instrumenting compiled binaries. As such, it cannot be used on interpreted and JIT-compiled languages. However, there exist other tools, like Hubble [24], that modify the runtime to trace managed languages.

7 Concluding Remarks

This paper first demonstrates that sampling-based profilers such as perf can produce systematically misleading results for flat-profile workloads with short-lived functions. Through real-world smartphone rendering workloads, we show how skid, shadow effects, limited coverage, and measurement perturbation can invalidate profiling results, leading engineers to chase non-issues or misjudge optimizations.

We present Blink, a lightweight tracing-based profiler that provides accurate and high-coverage PMU measurements with minimal overhead. Overall, we find that Blink is well-suited for profiling functions with low total hardware event counts, which constitute a large portion of Huawei's system libraries. These same functions are precisely the ones that sampling-based profilers struggle to capture reliably. Blink offers a practical solution for reliably evaluating compiler optimizations in environments where even small performance gains matter. Blink is available at https://github.com/Huawei-CPLLab/Blink.

8 Acknowledgements

We thank our reviewers and the shepherd for their insightful reviews and feedback. We also thank Henry Kao, Steven Huang, and all the other members of Huawei's BiSheng Mobile CPU team for their invaluable support in resolving technical challenges during the implementation of Blink. This research was supported by a contract between Huawei and YScope.

References

[1] perf-arm-spe(1) - Linux manual page. https://man7.org/linux/man-pages/man1/perf-arm-spe.1.html.

[2] perf(1) - Linux manual page. https://man7.org/linux/man-pages/man1/perf.1.html.

[3] strace(1) - Linux manual page. https://man7.org/linux/man-pages/man1/strace.1.html.

[4] Marcos K. Aguilera, Jeffrey C. Mogul, Janet L. Wiener, Patrick Reynolds, and Athicha Muthitacharoen. Performance Debugging for Distributed Systems of Black Boxes. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP '03)*, pages 74–89, Bolton Landing, NY, October 2003. ACM.

[5] Subho S. Banerjee, Saurabh Jha, Zbigniew Kalbarczyk, and Ravishankar K. Iyer. BayesPerf: minimizing performance monitoring errors using Bayesian statistics. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '21)*, page 832–844, Virtual, USA, 2021. ACM.

[6] Paul Barham, Austin Donnelly, Rebecca Isaacs, and Richard Mortier. Using Magpie for Request Extraction and Workload Modelling. In *Proceedings of the 6th USENIX Symposium on Operating Systems Design and Implementation (OSDI '04)*, pages 259–272, San Francisco, CA, December 2004. USENIX Association.

[7] Dean Michael Berris, Alistair Veitch, Nevin Heintze, Eric Anderson, and Ning Wang. XRay: A Function Call Tracing System. Technical report, 2016. A white paper on XRay, a function call tracing system developed at Google.

[8] Dehao Chen, Neil Vachharajani, Robert Hundt, Xinliang Li, Stephane Eranian, Wenguang Chen, and Weimin Zheng. Taming Hardware Event Samples for Precise and Versatile Feedback Directed Optimizations. *IEEE Transactions on Computers*, 62(2):376–389, 2013.

[9] Dehao Chen, Neil Vachharajani, Robert Hundt, Shih-wei Liao, Vinodha Ramasamy, Paul Yuan, Wenguang Chen, and Weimin Zheng. Taming hardware event samples for FDO compilation. In *Proceedings of the 8th Annual IEEE/ACM International Symposium on Code Generation and Optimization (CGO '10)*, page 42–52, Toronto, ON, Canada, 2010. ACM.

[10] Haibo Chen, Xie Miao, Ning Jia, Nan Wang, Yu Li, Nian Liu, Yutao Liu, Fei Wang, Qiang Huang, Kun Li, Hongyang Yang, Hui Wang, Jie Yin, Yu Peng, and Fengwei Xu. Microkernel Goes General: Performance and Compatibility in the HongMeng Production Microkernel. In *Proceedings of the 18th USENIX Symposium on Operating Systems Design and Implementation (OSDI '24)*, pages 465–485, Santa Clara, CA, July 2024. USENIX Association.

[11] Michael Chow, David Meisner, Jason Flinn, Daniel Peek, and Thomas F. Wenisch. The Mystery Machine: End-to-end Performance Analysis of Large-scale Internet Services. In *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI '14)*, pages 217–231, Broomfield, CO, October 2014. USENIX Association.

[12] Tudor David, Rachid Guerraoui, and Vasileios Trigonakis. Everything you always wanted to know about synchronization but were afraid to ask. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP '13)*, pages 33–48, Farmington, PA, 2013. ACM.

[13] Maria Dimakopoulou, Stéphane Eranian, Nectarios Koziris, and Nicholas Bambos. Reliable and Efficient Performance Monitoring in Linux. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC '16)*, pages 396–408, Salt Lake City, UT, 2016. IEEE.

[14] Paul J. Drongowski. Instruction-Based Sampling: A New Performance Analysis Technique for AMD Family 10h Processors. White paper, Advanced Micro Devices, Inc., Boston Design Center, November 2007. https://www.amd.com/content/dam/amd/en/documents/archived-tech-docs/white-papers/AMD_IBS_paper_EN.pdf.

[15] Úlfar Erlingsson, Marcus Peinado, Simon Peter, and Mihai Budiu. Fay: Extensible Distributed Tracing from Kernels to Clusters. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP '11)*, pages 311–326, Cascais, Portugal, October 2011. ACM.

[16] Rodrigo Fonseca, George Porter, Randy H. Katz, Scott Shenker, and Ion Stoica. X-trace: A Pervasive Network Tracing Framework. In *Proceedings of the 4th USENIX Symposium on Networked Systems Design and Implementation (NSDI '07)*, pages 271–284, Cambridge, MA, April 2007. USENIX Association.

[17] Susan L. Graham, Peter B. Kessler, and Marshall K. McKusick. gprof: a Call Graph Execution Profiler. In *Proceedings of the ACM SIGPLAN '82 Symposium on Compiler Construction*, pages 120–126, Boston, MA, 1982. ACM.

[18] Brendan Gregg. Differential Flame Graphs, November 2014. Online article and tool description. http://www.brendangregg.com/blog/2014-11-09/differential-flame-graphs.html.

[19] Brendan Gregg. The flame graph. *Commun. ACM*, 59(6):48–57, May 2016.

[20] Yuxin Guo, Alex W. Chadwick, Márton Erdős, Utpal Bora, Ilias Vougioukas, Giacomo Gabrielli, and Timothy M. Jones. OptiWISE: Combining Sampling and Instrumentation for Granular CPI Analysis. In *Proceedings of the 2024 IEEE/ACM International Symposium on Code Generation and Optimization (CGO '24)*, pages 373–385, Edinburgh, UK, 2024. IEEE/ACM.

[21] Intel Corporation. Precise Events — Intel VTune Profiler User Guide. User guide, Intel Corporation, 2022. https://www.intel.com/content/www/us/en/docs/vtune-profiler/user-guide/2023-0/precise-events.html.

[22] W. Korn, P.J. Teller, and G. Castillo. Just how accurate are performance counters? In *Proceedings of the 2001 IEEE International Performance, Computing, and Communications Conference (IPCCC '01)*, pages 303–310, Phoenix, AZ, 2001. IEEE.

[23] Roy Levin, Ilan Newman, and Gadi Haber. Complementing Missing and Inaccurate Profiling Using a Minimum Cost Circulation Algorithm. In *Proceedings of the 2008 International Conference on High Performance Embedded Architectures and Compilers (HiPEAC '08)*, pages 291–304, Göteborg, Sweden, 2008. Springer Berlin Heidelberg.

[24] Yu Luo, Kirk Rodrigues, Cuiqin Li, Feng Zhang, Lijin Jiang, Bing Xia, David Lion, and Ding Yuan. Hubble: Performance Debugging with In-Production, Just-In-Time Method Tracing on Android. In *Proceedings of the 16th USENIX Symposium on Operating Systems Design and Implementation (OSDI '22)*, pages 787–803, Carlsbad, CA, July 2022. USENIX Association.

[25] Jonathan Mace, Ryan Roelke, and Rodrigo Fonseca. Pivot Tracing: Dynamic Causal Monitoring for Distributed Systems. In *Proceedings of the 25th ACM Symposium on Operating Systems Principles (SOSP '15)*, pages 378–393, Monterey, CA, October 2015. ACM.

[26] Wiplove Mathur and Jeanine Cook. Toward Accurate Performance Evaluation using Hardware Counters. In *Proceedings of the ITEA Modeling and Simulation Workshop (ITEA '03)*, pages 23–32, Las Cruces, NM, January 2003. ITEA.

[27] Vladimir Mezentsev. [patch] gprofng: A new GNU Profiler, August 2021. https://sourceware.org/pipermail/binutils/2021-August/117665.html.

[28] Jumana Mundichipparakkal, Rui Chang, Yibo Cai, Darin Greene, Michael Williams, and Tushar Chouhan. Arm Statistical Profiling Extension: Performance Analysis Methodology. White paper, Arm Ltd., 2023. https://developer.arm.com/community/arm-community-blogs/b/architectures-and-processors-blog/posts/statistical-profile-extension.

[29] Andrzej Nowak, Ahmad Yasin, Avi Mendelson, and Willy Zwaenepoel. Establishing a base of trust with performance counters for enterprise workloads. In *Proceedings of the 2015 USENIX Annual Technical Conference (ATC '15)*, pages 541–548, Santa Clara, CA, 2015. USENIX Association.

[30] Maksim Panchenko, Rafael Auler, Bill Nell, and Guilherme Ottoni. BOLT: A Practical Binary Optimizer for Data Centers and Beyond. In *Proceedings of the 2019 IEEE/ACM International Symposium on Code Generation and Optimization (CGO '19)*, pages 2–14, Washington, DC, 2019. IEEE/ACM.

[31] Muhammad Aditya Sasongko, Milind Chabbi, Paul H J Kelly, and Didem Unat. Precise Event Sampling on AMD Versus Intel: Quantitative and Qualitative Comparison. *IEEE Transactions on Parallel and Distributed Systems*, 34(5):1594–1608, 2023.

[32] Benjamin H. Sigelman, Luiz André Barroso, Mike Burrows, Pat Stephenson, Manoj Plakal, Donald Beaver, Saul Jaspan, and Chandan Shanbhag. Dapper, a Large-Scale Distributed Systems Tracing Infrastructure. Technical report, Google, Inc., April 2010.

[33] Steven Rostedt. *ftrace - Function Tracer*, July 2017. https://www.kernel.org/doc/Documentation/trace/ftrace.txt.

[34] Vincent M. Weaver, Dan Terpstra, and Shirley Moore. Non-determinism and overcount on modern hardware performance counter implementations. In *Proceedings of the 2013 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS '13)*, pages 215–224, Austin, TX, 2013. IEEE.

[35] Bo Wu, Mingzhou Zhou, Xipeng Shen, Yaoqing Gao, Raul Silvera, and Graham Yiu. Simple profile rectifications go a long way. In *Proceedings of the 27th European Conference on Object-Oriented Programming (ECOOP '13)*, pages 654–678, Montpellier, France, 2013. Springer-Verlag.

[36] Yuanpei Wu, Dong Du, Chao Xu, Yubin Xia, Yang Yu, Ming Fu, Binyu Zang, and Haibo Chen. OS Rendering Service Made Parallel with Out-of-Order Execution and In-Order Commit. In *Proceedings of the 19th USENIX Symposium on Operating Systems Design and Implementation (OSDI '25)*, pages 601–618, Boston, MA, July 2025. USENIX Association.

[37] Hao Xu, Qingsen Wang, Shuang Song, Lizy Kurian John, and Xu Liu. Can we trust profiling results? understanding and fixing the inaccuracy in modern profilers. In *Proceedings of the ACM International Conference on Supercomputing (ICS '19)*, page 284–295, Phoenix, AZ, 2019. ACM.

[38] Jifei Yi, Benchao Dong, Mingkai Dong, and Haibo Chen. On the precision of precise event based sampling. In *Proceedings of the 11th ACM SIGOPS Asia-Pacific Workshop on Systems (APSys '20)*, page 98–105, Tsukuba, Japan, 2020. ACM.

[39] D. Zaparanuks, M. Jovic, and M. Hauswirth. Accuracy of performance counter measurements. In *Proceedings of the 2009 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS '09)*, pages 23 –32, Boston, MA, April 2009. IEEE.

ARTIFACT EVALUATED
USENIX
AVAILABLE

Breaking the Reward Barrier: Accelerating Tree-of-Thought Reasoning via Speculative Exploration

Shuzhang Zhong[1,2], Haochen Huang[1,2], Shengxuan Qiu[1,2], Pengfei Zuo*[3], Runsheng Wang[2], and Meng Li*[1,2]

[1]Institute for Artificial Intelligence, Peking University
[2]School of Integrated Circuits, Peking University
[3]ByteDance Seed

Abstract

Tree-of-Thought (ToT) reasoning structures Large Language Model (LLM) inference as a tree-based search, demonstrating strong potential for solving complex mathematical and programming tasks. However, its efficiency is constrained by the *reward dependency barrier*—a synchronization bottleneck caused by sequential reward-guided exploration that limits search parallelism and introduces substantial latency. Prior system optimizations, mainly designed for linear Chain-of-Thought (CoT) reasoning, cannot address these challenges, leaving the efficiency of ToT underexplored.

To enhance ToT reasoning efficiency, we observe that the reasoning paths can be explored speculatively to break the reward synchronization barrier. Therefore, in this paper, we propose SPEX and introduce three key techniques: (i) intra-query speculative path selection to predict and expand high-potential branches of ToT, (ii) inter-query budget allocation to balance speculative resource allocation across queries dynamically, and (iii) adaptive early termination to prune deep and redundant branches for a skewed search tree.

We implement SPEX on top of the SGLang framework and evaluate it across diverse ToT algorithms and LLMs. Extensive experiments show that SPEX achieves $1.2 \sim 3\times$ speedup for different ToT reasoning algorithms. Moreover, SPEX synergizes with token-level speculative decoding, achieving cumulative speedups of up to $4.1\times$. Ablation studies further confirm the contributions of each technique. Overall, SPEX represents a significant step toward efficient and scalable ToT reasoning, unlocking the parallelism required for high-performance inference-time scaling for LLMs.

1 Introduction

The recent success of Large Language Models (LLMs) has been driven by their ability to scale with inference-time compute budgets [3, 8, 26, 47, 50–52, 69]. Two main approaches have been constructed, including Chain-of-Thought (CoT) [13, 41, 59, 72] and Tree-of-Thought (ToT) [2, 11, 16, 19, 55, 66]. CoT prompts LLMs to reason over a coherent sequence of tokens (denoted as "thought") that serve as intermediate steps toward problem solving. ToT further generalizes CoT by maintaining a search tree and allowing LLMs to explore different branches of the tree adaptively, as in Figure 1 (a). To guide exploration, ToT often relies on a reward signal that evaluates existing reasoning branches and decides the next course of action, e.g., expanding existing branches or backtracking. Compared to CoT, ToT enables LLMs to perform complex reasoning and greatly enhance their problem-solving capabilities.

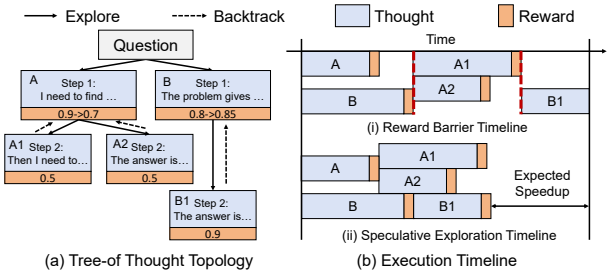


Figure 1: Illustration of ToT and Reward Barrier.

Prior ToT algorithms employ distinct exploration strategies driven by how reward signals guide the search. They can be categorized into two paradigms that impose different dependency patterns on the system [64]. **Breadth-First Search (BFS)** maintains a set of the most promising thoughts at each step, utilizing intermediate reward feedback to concurrently prioritize and expand branches at the current search frontier. In contrast, algorithms like MCTS [68] incorporate **Depth-First Search (DFS)**. The reward of all the nodes of a reasoning branch is iteratively updated via tree-search value propagation from leaf nodes. The search policy then leverages these refined rewards to revisit and expand prior nodes, enforcing a sequential dependency on previous iterations.

* Corresponding authors: Meng Li (meng.li@pku.edu.cn), Pengfei Zuo (pfzuo.cs@gmail.com)

While ToT enhances reasoning capabilities and accuracy, it comes with increased latency overhead [10, 24, 30]. Many studies have explored strategies for efficient reasoning. On the algorithmic side, techniques like TrimR focus on reducing the length of thoughts for better reasoning efficiency [7, 9, 17, 24, 32, 35, 36, 45, 57, 70]. On the system side, serving frameworks such as vLLM [27] and SGLang [71] achieve significant efficiency improvement with PagedAttention and RadixAttention. However, these system optimizations are primarily designed for linear CoT reasoning and overlook the unique characteristics of ToT reasoning [29].

In this paper, we identify the critical performance bottleneck in ToT reasoning as the Reward Dependency Barrier as illustrated in Figure 1(b). In Breadth–First Search, the barrier imposes a synchronization overhead: expansion decision is gated by the aggregation of reward values of all concurrent thoughts. High variance in decoding lengths creates stragglers, forcing shorter branches to idle until the longest one completes. For Depth-First Search, the barrier imposes a sequential constraint: each new traversal must wait for the previous rollout outcome and tree-value update before deciding whether to continue or backtrack, enforcing a serialized execution flow. Consequently, both scenarios prevent the system from sustaining dense batch processing. This inefficiency limits parameter reuse and KV cache sharing, significantly degrading arithmetic intensity and shifting the workload to a memory-bound regime.

To break the barrier, we propose **SPEX** to enhance parallelism in ToT reasoning. By proactively generating subsequent thoughts for promising branches without waiting for the synchronization of reward values, SPEX allows the system to overlap the generation of future thoughts with the completion of current thoughts. It addresses three key challenges in speculative exploration: (i) How to accurately predict which branches to expand speculatively, (ii) How to efficiently allocate computational resources across queries with varying demands and reuse opportunities, and (iii) How to manage skewed tree structures, where a few disproportionately deep thought branches dominate overall inference latency and are difficult to explore speculatively.

To address these challenges, we identify three key insights that shape the design of SPEX. First, reward stability across iterations enables reliable prediction of high-potential branches, even in DFS. Second, shared computation opportunities, such as KV cache reuse and parameter sharing, can enhance speculative efficiency and resource utilization. Finally, in skewed tree structures, most correct solutions emerge at shallow depths, making it possible to terminate the deep branches early.

Motivated by these insights, SPEX introduces three key techniques: (i) **Intra-query speculative branch selection**, which leverages reward stability in DFS and score-based allocation in BFS to identify high-likelihood branches for speculative expansion. (ii) **Inter-query budget allocation**, which dynamically distributes speculative resources across queries based on predicted utility, KV cache reuse potential, and system constraints, ensuring balanced and efficient utilization. (iii) **Adaptive early termination**, which halts exploration in skewed trees once sufficient confidence is achieved, avoiding unnecessary computation in deep branches.

Finally, to unify speculative exploration across diverse ToT reasoning algorithms, we design SPEX as **a general producer–consumer execution framework** that integrates the three techniques. The framework decouples branch expansion (producer role) from control logic (consumer role), enabling concurrent progress on both primary and speculative branches while adapting to system resource conditions. Our contributions can be summarized as follows:

- We identify the reward dependency barrier as the principal bottleneck in ToT reasoning.
- We propose SPEX, a speculative exploration framework that breaks this barrier by proactively and speculatively exploring promising branches.
- We design three complementary techniques, i.e., intra-query speculative selection, inter-query budget allocation, and early termination strategy, to improve the effectiveness of speculation.
- We implement the **SPEX** on top of SGLang and demonstrate up to 3× speedup over prior-art ToT reasoning algorithms without compromising accuracy. SPEX is available at https://github.com/PKU-SEC-Lab/SPEX.

2 Background

2.1 Tree-of-Thought Reasoning

Recent research on inference scaling laws has shown that LLM performance improves with increased inference-time compute budget [3, 8, 20, 26, 42, 43, 50–52]. Chain-of-Thought (CoT) extends reasoning depth along a single trajectory, Tree-of-Thought (ToT) further expands reasoning breadth by exploring multiple solution paths in parallel [64].

Structurally, ToT frames reasoning as a search over a tree of discrete steps. In this framework, each step constitutes a coherent segment of thought, serving as the atomic unit of expansion. Accordingly, a node represents the partial solution state reached by applying a step, and a sequential trajectory of these states from the question to a final answer forms a reasoning path [60].

The search is guided by reward signals, derived either from a dedicated reward model [60] or through the reasoning model's intrinsic self-evaluation [18]. As illustrated in Figure 2, ToT algorithms can be categorized by their expansion and traversal strategies into BFS and DFS.

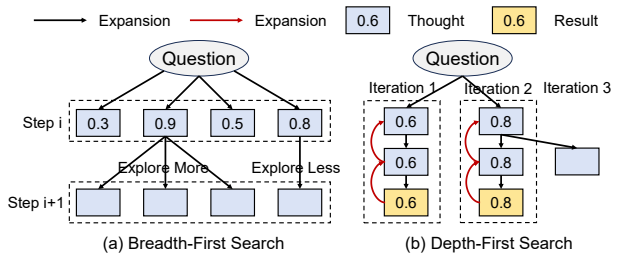


Figure 2: Classification of different ToT algorithms.

2.1.1 Breadth-First Search

Breadth-First Search maintains a collection of the most promising nodes at each step, optimizing the search frontier by utilizing intermediate feedback to concurrently prioritize and extend these branches [64]. A representative algorithm is REBASE [60] illustrated in Figure 2(a), which utilizes process rewards to guide this search. It employs a softmax-based mechanism to dynamically allocate expansion budgets. Formally, given a total sampling budget B_i at depth i, the expansion width W_j for a node n_j is calculated as:

$$W_j = \text{Round}\left(B_i \frac{\exp(R(n_j)/T_b)}{\sum_k \exp(R(n_k)/T_b)}\right)$$

where $R(N_j)$ denotes the reward score of node n_j, T_b is the balance temperature controlling the exploration-exploitation trade-off, and the summation index k iterates over all candidate nodes at the current depth i.

2.1.2 Depth-First Search

Depth-First Search explores the most promising path until a terminal state is reached, and then leverages updated value estimates to backtrack and revisit high-potential prior states for alternative exploration [64]. Thus, the next traversal is chosen only after the current rollout updates the tree state. Monte Carlo Tree Search (MCTS) is a representative algorithm for this category as illustrated in Figure 2(b) [5, 46, 56, 62]. It balances the trade-off between exploration and exploitation using the Upper Confidence Bound (UCB) metric to select the next action a^*:

$$a^* = \arg\max_a \left[Q(s,a) + c \cdot \sqrt{\frac{\log N(s)}{N(s,a)}}\right]$$

where $Q(s,a)$ represents the estimated value of taking action a in state s, $N(s)$ and $N(s,a)$ denote the visit counts for the state and the specific action, respectively, and c is the exploration constant. By updating these statistics after each traversal, the algorithm retrospectively adjusts its search focus. However, this iterative dependency typically enforces a sequential execution pattern, severely limiting parallelism.

Notably, DFS can operate in isolation or coexist with BFS. Pure DFS, such as RSTAR-MCTS [46], generates a single continuous chain per iteration, receiving a reward signal only upon reaching the final outcome. In contrast, REST-MCTS [68] adopts a hybrid strategy where each iteration functions as a step-wise BFS: at every depth, it expands multiple candidate nodes and utilizes immediate process rewards to select the single best child for the subsequent step.

2.2 Existing Optimizations for ToT Reasoning

Existing system optimizations for LLM inference are highly effective for single-path CoT reasoning, but few are specifically designed to support the tree-structured search process of ToT reasoning. This fundamental mismatch leaves current serving infrastructures unable to fully exploit the potential of multi-path search.

On the system side, a series of serving frameworks have substantially improved throughput and latency for linear decoding workloads. vLLM [27] introduces PageAttention, which decouples key-value caching from attention computation. SGLang [71] implements RadixAttention to reuse intermediate KV-cache memory across requests. FlashAttention and FlashDecoding utilize the online softmax to reduce memory usage [15]. FlashInfer integrates the above optimizations of attention into a unified block-sparse framework [65]. Together, these efforts demonstrate the effectiveness of system-level optimization for general LLM decoding.

On the algorithmic side, efficient reasoning methods primarily target token efficiency in CoT. Approaches reduce the number of generated tokens or adaptively adjust reasoning length based on task complexity [7, 9, 17, 35, 36, 57, 70]. ETS [23] goes beyond token-level analysis by providing a system-oriented perspective, showing that in memory-bound scenarios, KV cache access dominates inference costs. By optimizing KV reuse and access patterns, ETS connects algorithm design with system efficiency.

Despite these advances, existing methods share a fundamental limitation: they are tailored for linear CoT reasoning. While CoT-style optimizations leverage batching and prefix sharing across requests, they do not accommodate the irregular, tree-shaped exploration inherent to ToT reasoning.

3 Motivation and Challenge

Unlike CoT, ToT requires concurrent branch expansion, dynamic allocation of computational budgets, and synchronization based on reward signals. These characteristics introduce new system-level bottlenecks that current inference systems are not designed to handle.

In this section, we first present our key observation that the reward dependency barrier constitutes the primary bottleneck limiting ToT reasoning efficiency (§3.1). We then provide a quantitative analysis of how this barrier constrains system

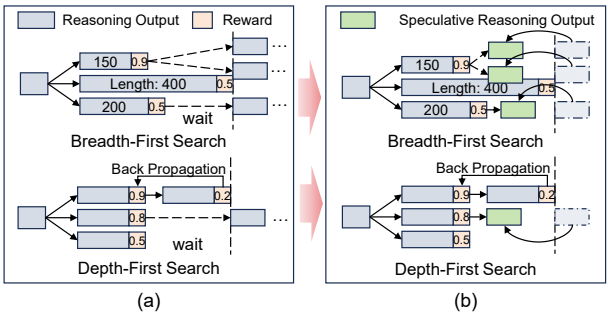


Figure 3: Example of (a) reward barrier and (b) our proposed speculative exploration.

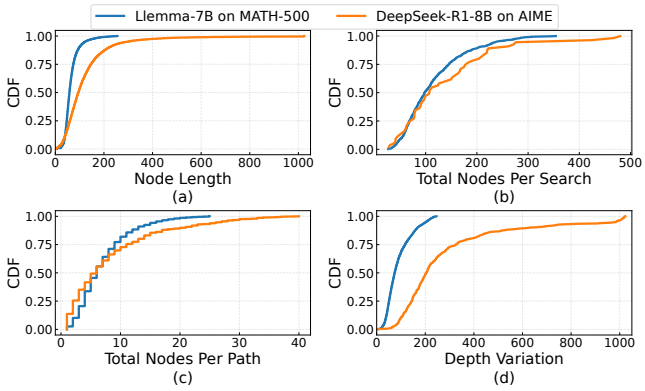


Figure 4: Cumulative Distribution Function (CDF) of reasoning characteristics for Llemma-7B [1] and DeepSeek-R1-8B. (a) Node Lengths; (b) Total Nodes Per Search; (c) Total Nodes Per Path; (d) Variation of Node Lengths at the same depth.

performance (§3.2), followed by our motivation: speculative branch exploration as a means to improve efficiency (§3.3). Finally, we discuss the challenges to perform speculative exploration in ToT reasoning (§3.4).

3.1 Observation on Reward Dependency Barrier

We observe that the reward dependency barrier is the principal performance bottleneck in ToT reasoning: At each stage of the search, the system must pause to await feedback from the reward model before determining the next action. The manifestation of this barrier varies depending on the underlying algorithmic paradigm and arises from two main sources.

1) Intrinsic Sequentiality in Depth-First Search. In DFS algorithm, the search advances along a single reasoning path, with progress contingent on receiving the reward signal. Only after obtaining this feedback can the system decide whether to backtrack and which alternative branch to explore next. This results in inherently sequential execution and severely limits opportunities for parallelism, as depicted in Figure 3(a).

2) Synchronization Bottlenecks in Breadth-First Search. In contrast, BFS algorithm can explore multiple candidate branches in parallel at a given search depth. However, its overall progress remains gated by synchronization points at each level: the system must aggregate reward signals across all branches before determining the expansion budget for each node at the next depth, as shown in Figure 3(a). This introduces critical synchronization overheads that limit parallelism, as faster branches are forced to idle until the global aggregation completes.

To quantify the reward barrier, we analyze the cumulative distribution of node lengths and their variation across search depths. We conduct this profiling on two representative workloads: Llemma-7B on MATH-500 and DeepSeek-R1-8B on the challenging AIME benchmark. Here, a node length is the number of tokens generated for one reasoning step. As shown in Figure 4(a), DFS typically involve a large number of sequential steps, with each step generating 50-100 tokens. Figure 4(b) and 4(c) further illustrate that the total number of nodes per search and the number of nodes along each path are both substantial, indicating that MCTS-style algorithms require traversing many nodes in series. Given that these methods often require hundreds of such steps, this results in significant cumulative latency. For BFS algorithm, Figure 4(d) quantifies the synchronization overhead by plotting the variation in node lengths at the same depth. The slow saturation of the CDF curves reveals a pronounced long-tail effect: while many branches finish early, the system is forced to idle for the slowest "straggler" branches to complete. This high variance, particularly evident in the DeepSeek workload, exacerbates resource underutilization at synchronization barriers.

Overall, this analysis highlights that both algorithmic paradigms are fundamentally constrained by the reward barrier: DFS methods suffer from high sequential latency due to deep search paths, while BFS methods are limited by stragglers and synchronization overhead.

3.2 System-Level Performance Analysis

To quantitatively illustrate how the reward dependency barrier limits system efficiency, we analyze the performance of ToT using the Roofline model. Two primary contributors dominate memory operations during inference: (i) model weight accesses and (ii) KV cache accesses. Both are significantly affected when parallelism is constrained, leading to repeated memory operations and inefficient resource utilization.

When batch sizes are small due to limited parallelism, the system must repeatedly load model weights for each reasoning step. Similarly, KV cache accesses become inefficient due to the lack of shared prefixes across reasoning paths. Without sufficient KV sharing, the system repeatedly reloads overlapping or identical context segments across different branches,

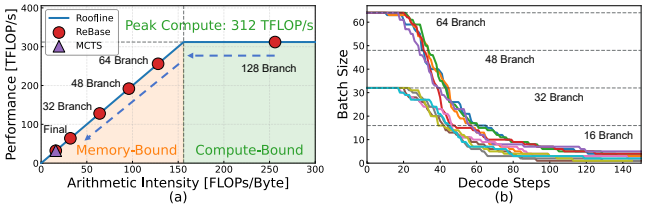


Figure 5: (a) Illustrative roofline analysis; (b) Intensity degradation over decode steps.

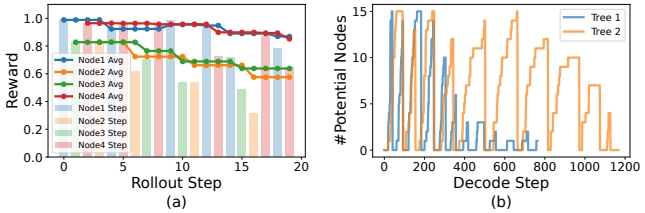


Figure 6: (a) Stability of MCTS node values across rollout steps. These estimates change as later rollouts update ancestor nodes. (b) Fluctuations in the number of potential speculative nodes for two queries at different decoding steps in REBASE.

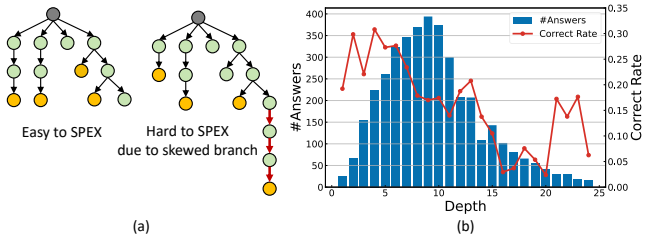


Figure 7: (a) SPEX struggles with skewed trees under the REBASE algorithm due to dominant deep branches. (b) Answer count and correct rate of different reasoning depths in skewed deep trees (depth > 10).

increasing memory bandwidth consumption. This repeated loading not only adds latency but also prevents tree-based optimizations like RadixAttention.

As shown in Figure 5(a), these repeated memory operations result in low arithmetic intensity (FLOPs/Byte), making memory access the primary bottleneck. DFS is inherently confined to the memory-bound region due to their strict sequentiality. While BFS starts with higher parallelism, it suffers from batch attrition. As illustrated in Figure 5(b), the early completion of shorter branches degrades the effective batch size, dynamically shifting the workload from compute-bound into memory-bound regime.

3.3 Motivation

While the reward barrier limits inter-branch concurrency, significant latent parallelism exists within the reasoning tree. By **speculatively exploring confident branches** before reward feedback arrives, we can break the reward barrier, reduce synchronization delays, and reuse shared prefixes.

This motivates **SPEX**, a speculative exploration framework that exploits potential parallelism to overcome sequential bottlenecks in ToT reasoning. Specifically, SPEX proactively and speculatively explores promising candidate branches based on predictive signals, as illustrated in Figure 3(b). This removes the blocking dependency on reward feedback, allowing the expansion process to continue without idling.

3.4 Challenges

However, directly applying speculative exploration is insufficient, as it faces the following challenges.

Challenge 1: Speculating Inaccuracy on Future-Needed Branches. Speculating on future-needed branches is straightforward in BFS, where candidate nodes are confined to the current depth. However, for DFS algorithms like MCTS, every node in the tree may be revisited, making it difficult to predict which branches will be required. This complexity increases the difficulty of efficiently allocating speculative computation to accelerate future exploration.

Challenge 2: Complex Speculative Budget Allocation across Multiple Queries. In multi-query scenarios, distributing speculative resources becomes more complex due to variations in the number of potential nodes per query and their probabilities of future relevance. As illustrated in Figure 6(b), the number of nodes available for speculative execution fluctuates significantly between queries. Additionally, these nodes may vary in their likelihood of being utilized, depending on the accuracy of predictions for each query. This variability, combined with limited system resources, requires dynamic allocation strategies that prioritize queries based on their search state, predicted utility, and overall system capacity to optimize throughput and responsiveness.

Challenge 3: Inefficient Resource Allocation under Skewed Tree Structures. For highly skewed trees, SPEX struggles to achieve significant acceleration, as shown in Figure 7(a). In such cases, the majority of computational resources are consumed by disproportionately deep branches, which are challenging to predict and explore speculatively. This reduces parallelism, increases latency from sequential exploration, and limits efficient resource allocation, particularly when dealing with long dominant branches.

4 The SPEX Design

This section presents the design of SPEX, a speculative exploration framework that breaks the reward dependency barrier

by increasing parallelism for ToT reasoning.

4.1 Overview

The key idea of SPEX is to speculatively explore confident branches before reward feedback arrives, as presented in §3.3. To address the challenges identified in speculative exploration (§3.4), SPEX leverages three core techniques, each motivated by key insights observed in ToT reasoning.

Insight 1: Reward Stability Enables Prediction. As shown in Figure 6(a), node value estimates in MCTS exhibit high stability across iterations, with reward values changing minimally as rollouts progress. A later rollout updates the value estimates and visit counts of its ancestor nodes, so a node's estimate may evolve over time. This is because each expansion step updates only one node's statistics, and rewards, averaged over multiple rollouts, evolve slowly while UCB variations are primarily influenced by visit counts. This stability allows reasonably accurate predictions of node behavior, facilitating speculative execution or pruning strategies to reduce sequential bottlenecks.

Based on the insight, we propose *an efficient intra-query speculative branch selection scheme*, which leverages reward stability in DFS and score-based allocation in BFS to identify high-likelihood branches for speculation (addressing Challenge 1).

Insight 2: Shared Computation Shapes Allocation. Efficient allocation of speculative resources requires evaluating the potential benefits of each query. The primary gains from speculative exploration arise from two forms of shared computation: (i) parameter loading reuse, where speculative branches share model weights with the primary branch, and (ii) KV cache reuse, where branches with overlapping prefixes reuse cached attention states. These benefits are influenced by factors such as the prediction accuracy of speculative branches and the proportion of the KV cache that can be reused. By quantifying these elements, the system can guide resource distribution to maximize the overall utility of speculation while minimizing waste on low-impact queries.

Based on the insight, we propose *an efficient inter-query budget allocation scheme*, which dynamically distributes speculative resources across queries based on predicted utility, KV cache reuse potential, and system constraints, ensuring balanced and efficient utilization (addressing Challenge 2).

Insight 3: Answer Depth Bias Favors Shallower Exploration. As illustrated in Figure 7(b), even for skewed deep trees, most answers are generated at shallower depths, where the correct rate is also higher. This indicates that early-stage exploration typically produces both more frequent and more accurate solutions, suggesting that deep exploration may often be unnecessary under majority vote [31] strategies.

Based on the insight, we propose *an adaptive early termination strategy*, which halts exploration in skewed trees once sufficient confidence is achieved, avoiding unnecessary computation in deep branches (addressing Challenge 3).

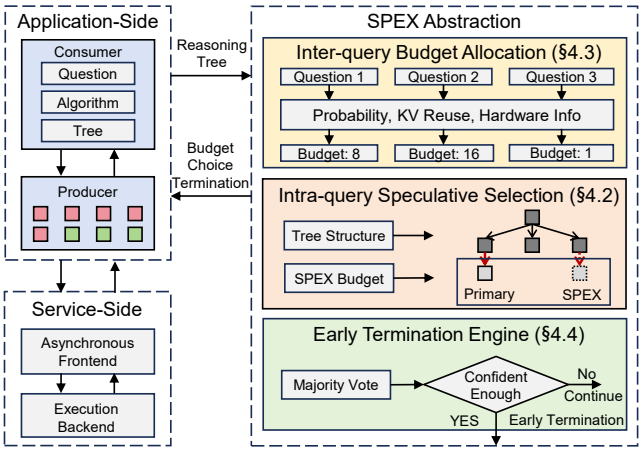


Figure 8: The architectural overview of SPEX.

Together, these three techniques define the SPEX workflow, as illustrated in Figure 8. First, given multiple queries, SPEX applies *the inter-query budget allocation scheme* to allocate speculative resources based on query probability, KV cache reuse potential, and hardware capabilities (§4.3). Next, for each query, SPEX employs *the intra-query speculative selection scheme* to predict and expand the most promising future branches for speculative exploration (§4.2). Finally, to handle skewed trees, SPEX integrates *the adaptive early termination strategy*, which halts exploration and returns answers when the system has gathered sufficient confidence (§4.4).

Finally, to unify speculative exploration across diverse ToT reasoning algorithms, we design SPEX as *a general producer–consumer execution framework* (§4.5) that integrates the three techniques.

4.2 Intra-query Speculative Selection

To improve efficiency within a single query, SPEX integrates speculative exploration into the node selection process, employing two complementary strategies: one tailored for DFS and the other for BFS algorithms.

1) Speculation for Depth-First Search. As shown in Insight 1, reward estimates of individual nodes in MCTS exhibit high stability across iterations, enabling accurate forecasting of future node selections without invoking the reward value.

SPEX leverages this property by *simulating* the next k primary selections under the current tree state. Here, simulation means applying the same UCB selection rule on a temporary tree state, using current value estimates, visit counts, and active speculation records, without invoking LLM generation or reward evaluation. After each simulated selection, the algorithm updates visit counts and UCB values before proceeding to the next simulated step, ensuring that subse-

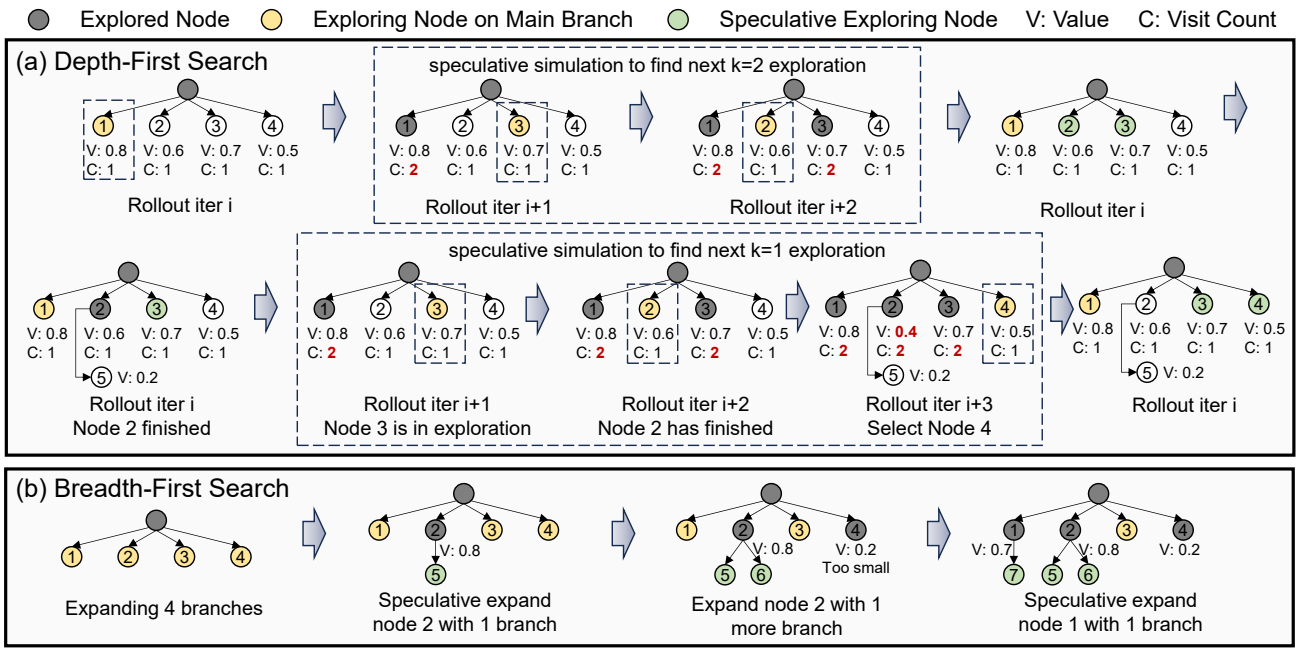


Figure 9: SPEX for (a) DFS and (b) BFS. (a) SPEX simulates the next k iterations, updating visit counts (C) and values (V) after each simulated choice, and skipping nodes already under or completed in speculation. (b) SPEX allocates speculative branches via softmax over reward scores, prioritizing high-value nodes to improve reuse and reduce stragglers.

quent predictions are conditioned on prior choices. It does not assume rewards are identical along a path; it relies on short-horizon stability of node value estimates. If the selected node is already under speculative expansion, it is skipped. If speculation for that node has been completed, the stored speculative reward is incorporated into the UCB update to refine the accuracy of future forecasts. This process focuses speculative computation on unoccupied, high-likelihood branches while reusing prior results to improve prediction accuracy.

Algorithm 1 presents the speculative selection procedure, and Figure 9(a) illustrates its execution. SPEX predicts the next UCB-based choices, skips nodes already under speculation, and integrates completed speculative results into value updates. As a result, the system keeps producers busy with reusable work while minimizing redundant expansions.

2) Speculation for Breadth-First Search. In BFS, speculative exploration targets the inefficiency caused by the long-tail variance in decoding lengths. Since shorter branches finish significantly earlier than the longest ones, they create valuable idle slots at the reward barrier. SPEX leverages this opportunity by proactively expanding these completed short nodes while the system waits for the stragglers.

As illustrated in Figure 9(b), the speculative budget matches the count of completed nodes. To distribute this budget, SPEX applies the expansion policy of the underlying BFS algorithm to these finished nodes. For instance, with REBASE, SPEX employs its reward-based softmax to determine the expansion width. By strictly adhering to this baseline policy, SPEX allocates more speculative children to higher-scoring nodes (e.g., node 2). This consistency ensures that speculative choices mirror the algorithm's intended search path, thereby maximizing prediction accuracy.

Algorithm 1 Speculation for ToT-DFS.

Input: Search tree *tree*, number of speculative branches k
Output: Ordered list of speculative nodes to expand

```
selected_nodes ← [ ]
for t ← 1 to k do
    node ← SimulateNext(tree)
    while node ∈ active_expansions or node ∈ completed_speculations do
        if node ∈ completed_speculations then
            UpdateUCBWithReward(node.reward)
        end if
        node ← SimulateNext(tree)
    end while
    selected_nodes.append(node)
    IncrementVisitCount(node)
    UpdateUCB(tree)
end for
return selected_nodes
```

Mis-speculation Handling. SPEX adopts distinct strategies for handling mis-speculation. For DFS, SPEX operates without explicit detection mechanisms. Since DFS explores paths iteratively, a prediction not selected in the current round

remains a valid candidate for future rollouts. In contrast, for BFS, SPEX implements strict verification. It detects misspeculation by cross-referencing predicted nodes with the actual expansion frontier. If a speculated node is not selected by the current frontier, SPEX immediately terminates the corresponding branch to prevent resource wastage.

4.3 Inter-query Budget Allocation

In multi-query serving, SPEX allocates speculative execution across queries to maximize overall throughput while respecting hardware limits and each query's parallel capacity.

We first determine the global speculative budget k_{total} using a roofline analysis. Achievable throughput is bounded by the intersection of the compute and memory ceilings; the corresponding batch concurrency is taken as k_{total}. This ensures that speculation moves execution toward the compute-bound regime without oversaturating on-chip memory bandwidth or degrading arithmetic intensity.

The acceleration of speculation arises from two primary forms of reuse: (i) *parameter reuse*, where speculative branches amortize model parameters with the primary branch, and (ii) *KV cache reuse*, where branches sharing long prefixes with the primary path reuse KV states in attention computation. Live nodes keep their prefix KV states in the serving cache when resident; a speculative child only needs to generate KV states after its fork point. Let S_w denote the model parameter size, and $S_{KV}(q)$ the amount of KV state in query q that can be reused by its speculative branches.

For each query q, we estimate its speculative utility as

$$score_q = C_q \times P_q \times (S_w + S_{KV}(q)), \tag{1}$$

where C_q is the exploitable parallel capacity within the query and P_q is the predicted hit rate of speculative branches. Concretely, C_q is the number of eligible speculative branches that query q can issue without duplicating active work.

The allocation of the global budget follows a softmax weighting over these scores:

$$k_q = \min\left(C_q,\ \left\lfloor k_{total} \cdot \frac{e^{\tau score_q}}{\sum_{q'} e^{\tau score_{q'}}} \right\rfloor\right), \tag{2}$$

where τ controls allocation sharpness. This strategy assigns more speculative capacity to queries with higher expected benefit while avoiding overspeculation beyond each query's intrinsic parallelism.

4.4 Early Termination for Skewed Trees

In highly skewed trees, SPEX faces significant challenges in achieving acceleration due to the disproportionate computational cost of exploring deep branches. However, as noted in Insight 3, most answers are generated at shallower depths, where the correctness rate is higher. This suggests that exhaustive exploration of deep branches is often unnecessary, especially when reliable solutions can already be inferred from earlier exploration stages.

To address this, we propose an *adaptive early termination strategy* that halts exploration once sufficient evidence has been gathered to confidently determine a result. This termination applies to the actual ToT search, not only to speculative branches. Specifically, during exploration, SPEX monitors the total number of generated answers n and the confidence scores of the top two hypotheses. The confidence scores are defined as:

$$\text{conf}_{1\text{st}} = \sum_{i \in 1\text{st}} w_i, \quad \text{conf}_{2\text{nd}} = \sum_{i \in 2\text{nd}} w_i, \tag{3}$$

where $\text{conf}_{1\text{st}}$ and $\text{conf}_{2\text{nd}}$ represent the total rewards belonging to the top 1 and top 2 answers, respectively.

Exploration terminates early if it satisfies the condition:

$$n \geq t \quad \text{and} \quad \text{conf}_{1\text{st}} - \text{conf}_{2\text{nd}} > \alpha \cdot \text{avg}(\text{w}_{2\text{nd}}), \tag{4}$$

where t is the minimum number of answers required, α is a scaling factor controlling the margin of confidence, and $\text{avg}(\text{w}_{2\text{nd}})$ is the average weight of answers belonging to the second-best hypothesis. In the experiments, α is set to 0.5. This ensures that exploration avoids unnecessary computation in deep branches when high-confidence solutions can already be determined from shallower exploration. Thus, any potential accuracy tradeoff stems from pruning these deep branches before full exploration.

4.5 General Producer–Consumer Framework

To unify speculative exploration across different ToT algorithms, we design SPEX as *a general producer–consumer execution framework*. It decouples branch expansion (producer role) from control logic (consumer role), enabling concurrent processing of both primary and speculative branches while dynamically adapting to system conditions.

Specifically, multiple producers execute explorations in parallel. A producer may be assigned to (i) a node along the *primary branch*—the branch that the underlying search algorithm $\mathcal{A}$ is currently committed to exploring—or (ii) a node from *speculative branches* predicted to have high utility. The centralized consumer loop monitors the completion of these expansions through a shared completion queue.

When a completed node belongs to the primary branch, the consumer immediately appends it to the search tree and triggers the next expansion step dictated by $\mathcal{A}$. Simultaneously, the consumer evaluates whether idle producers can be utilized for additional speculative expansions, thereby overlapping useful computation with reward model latency. The scheduler also favors speculative nodes with reusable prefixes, reducing extra KV-cache footprint. If the completed node originates from a speculative branch, it is appended to the speculative subtree. The system again inspects available resources and,

Algorithm 2 SPEX Producer-Consumer Framework

Input: Initial state s_0, search algorithm type $\mathcal{A}$
Output: Final state s_n

```
tree ← Tree structure with root s_0
request_queue, completion_queue ← []
Launch multiple ProducerWorker() instances
request_queue.put(s_0)
while search is not complete do
    (node, child) ← await completion_queue
    if tree.is_on_primary_branch (node, A) then
        tree.add_node(child)
        tree.expand(A)
    else
        tree.add_spec_node(child)
    end if
    while has_idle_producers() do
        spec_node ← tree.select_spec()
        request_queue.put(spec_node)
    end while
end while
Return tree.s_n

procedure ProducerWorker
    while not Terminate do
        node ← await request_queue
        new_child ← expand_node(node)
        completion_queue.put(node, new_child)
    end while
end procedure
```

if capacity permits, issues further speculative expansions to keep producer threads saturated.

Algorithm 2 outlines the execution flow. The design achieves three benefits. **1) Non-blocking Execution:** Primary branch progress is never stalled by speculative work. **2) Opportunistic Parallelism:** Idle resources are opportunistically exploited for parallel speculative exploration. **3) Algorithm-agnostic Unification:** The framework remains agnostic to the underlying reasoning algorithm, supporting different paradigms within a unified scheduling architecture.

5 Implementation

We implemented the **SPEX system** on top of SGLang [71], introducing several key modifications and components.

On the server side, we modified the SGLang front-end implementation. The original system was designed to handle batch requests that return results collectively. To enable finer-grained scheduling, we re-engineered the system to support asynchronous responses, allowing individual requests to return results independently.

On the application side, we utilized Python's coroutine framework to implement the coordinated operation of a **Producer-Consumer** model, which manages interactions with SGLang. All three core components of the SPEX system were implemented on the application side, ensuring seamless integration and efficient task execution.

Table 1: Configurations of evaluated algorithms. The configuration values denote the target number of generated answers.

Algorithm	Type	Configuration
REST-MCTS [68]	DFS+BFS	5, 10
RSTAR-MCTS [46]	DFS	5, 10
REBASE [60]	BFS	8, 16
ETS [23]	BFS	8, 16

6 Experimental Evaluation

6.1 Experimental Setup

Models. We evaluated SPEX across four models: Llemma-7B, Llemma-34B [1], DeepSeek-R1-Distill-Qwen-8B [20], and Qwen3-30B-A3B [53]. The Llemma suite was selected to ensure comparability with established ToT baselines, as these models are used in those evaluations [23, 60]. We employ its dedicated reward model for evaluation. Conversely, DeepSeek and Qwen were included to validate generalization to state-of-the-art models, where we derived reward signals directly from intrinsic log probabilities to serve as confidence proxies [18].

Hardware. We conducted experiments across different hardware setups to evaluate SPEX's performance under various scenarios. Specifically, we tested Llemma-7B and Deepseek-8B on NVIDIA A6000 GPUs, while Llemma-34B and Qwen3-30B-A3B were evaluated on NVIDIA A100 GPUs. These setups allowed us to analyze and compare performance across diverse compute environments.

Datasets. To ensure a fair comparison, we matched the dataset difficulty to the capabilities of each model. For the Llemma family, we utilized GSM8K [14] and MATH-500 [21, 34] following the experiment setup of REBASE and ETS, as more extreme challenges like AIME remain intractable for Llemma even with ToT. In contrast, for the state-of-the-art models (DeepSeek and Qwen), simpler tasks like MATH-500 can be easily solved without search. Therefore, we utilized high-difficulty benchmarks, including AIME 2024 [39], AIME 2025 [40], BRUMO [4], and HMMT-Feb25 [22] to demonstrate the performance of our framework.

Algorithms. As listed in Table 1, we evaluated SPEX on four representative ToT algorithms: REST-MCTS, RSTAR-MCTS, REBASE and ETS. Each algorithm was tested under two configurations, which represent the number of answers to generate; specifically, this corresponds to the rollout count for DFS methods and the initial search width for BFS methods. Here batch size denotes concurrent ToT queries, not individual

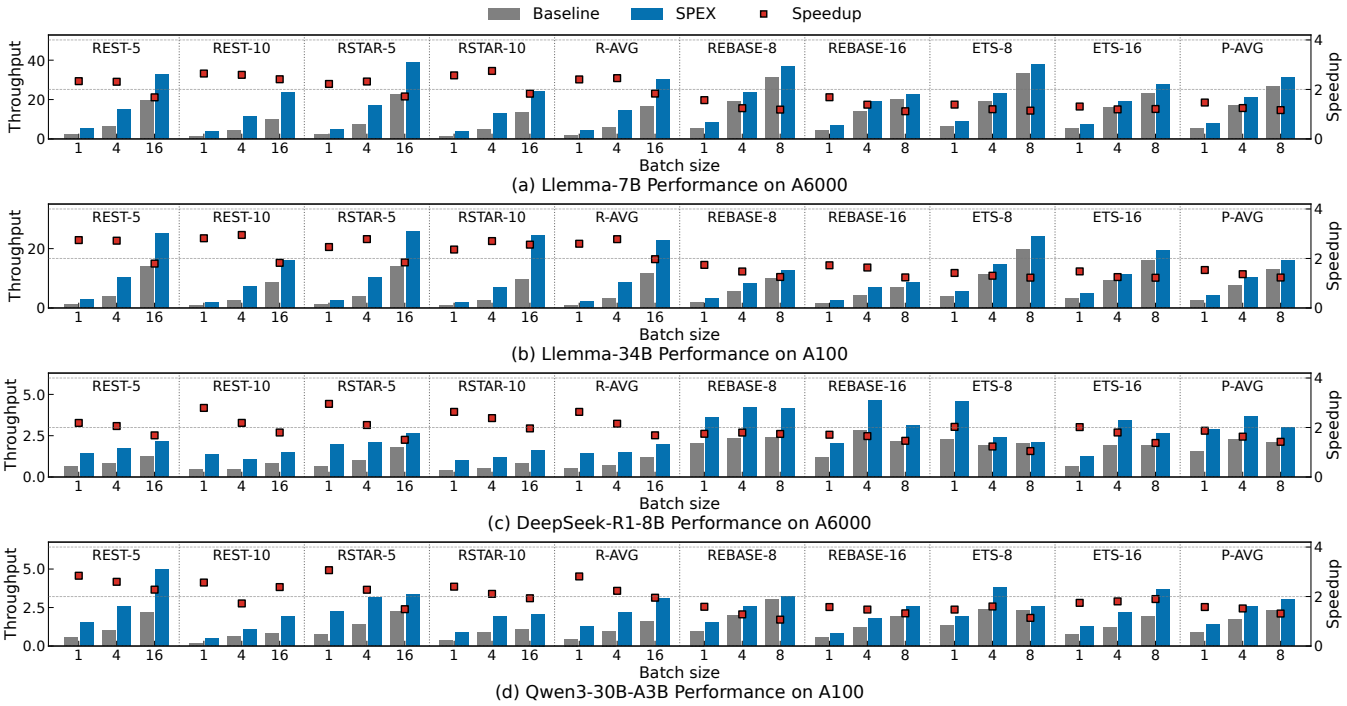


Figure 10: The speedup and throughput (finished questions per minute) for different ToT reasoning tasks.

branch-generation requests. For example, one REBASE-16 query already contains 16 requests, so BS=8 can expose up to 128 such requests before SPEX adds speculative branches.

6.2 End-to-End Performance

Figure 10 presents the throughput and speedup of SPEX compared to baseline across various algorithms, configurations, and batch sizes. The results demonstrate consistent performance improvements with SPEX under diverse settings. SPEX targets query throughput; extra speculative branches may slightly increase per-token latency, as analyzed in § 6.6.

On average, SPEX achieves a speedup of 1.8–3× for DFS and 1.2–1.9× for BFS. The greater acceleration for DFS stems from their inherently limited branch-level parallelism, where SPEX introduces more opportunities for optimization. In contrast, algorithms without DFS like REBASE already exhibit significant parallelism, leaving less room for further improvement. Among DFS algorithms, REST-MCTS achieves slightly higher speedup compared to RSTAR-MCTS, as the BFS creates more opportunities for SPEX optimization.

Impact of Batch Size: The speedup is particularly notable at smaller batch sizes. These batch sizes should be interpreted at the ToT-query level, not as the number of individual decoding sequences. For DFS algorithms, SPEX can achieve up to 3× acceleration at small batch sizes, and for BFS-only algorithms, the speedup reaches up to 1.7×. As the batch size increases, the system transitions from being memory-bound to compute-bound, reducing SPEX's relative benefits.

Impact of Configuration: DFS algorithms demonstrate higher speedup when generating more answers, as this increases the number of potential nodes. Conversely, for BFS-only algorithms, generating more answers results in lower speedup. This is because algorithms like REBASE expand branches proportional to the required answers, transitioning to a compute-bound regime.

6.3 Accuracy Evaluation

We report pass@1 accuracy and keep the same decoding settings, including sampling temperature, top-p, and token limits, for all methods. Table 2 demonstrates the accuracy evaluation across pure CoT, the ToT baseline, and SPEX. The results show that stepwise ToT provides a clear accuracy lift over pure CoT, and SPEX preserves this lift while improving throughput. The results indicate that early termination introduced by SPEX does not negatively impact accuracy. Any accuracy difference mainly comes from early termination changing the explored search space. For most configurations, SPEX can achieve slightly higher accuracy than the ToT baseline. This gain stems from our adaptive early termination technique. Deep, skewed branches tend to exhibit lower correctness. By terminating these branches early upon reaching high confidence, SPEX prevents low-quality tails from skewing the

Table 2: Accuracy evaluation.

Model	Dataset	Method	REST	RSTAR	REBASE	ETS	AVG
Config.			5 / 10	5 / 10	8 / 16	8 / 16	
Llemma -7B	MATH -500	CoT	26.8	26.8	26.8	26.8	26.8
		ToT	35.2 / 37.4	**35.6 / 37.0**	**43.4** / 45.2	40.6 / 43.8	40.3
		SPEX	**36.8 / 38.8**	33.8 / 36.4	40.8 / **46.4**	**42.2 / 46.0**	**41.5**
	GSM8K	CoT	63.8	63.8	63.8	63.8	63.8
		ToT	75.8 / 77.1	74.6 / 76.4	83.5 / 86.7	82.1 / 83.4	80.7
		SPEX	**77.3 / 78.1**	**74.9 / 76.8**	**84.6 / 87.0**	**85.0 / 86.7**	**81.8**
Llemma -34B	MATH -500	CoT	29.0	29.0	29.0	29.0	29.0
		ToT	**41.2 / 40.0**	**37.2** / 38.4	43.4 / **48.0**	41.0 / **45.6**	42.3
		SPEX	38.2 / **40.8**	34.8 / **38.8**	**46.0** / 46.6	**44.9 / 45.4**	**42.9**
	GSM8K	CoT	75.1	75.1	75.1	75.1	75.1
		ToT	79.9 / 80.9	77.7 / 80.1	87.3 / 88.0	**86.7 / 87.3**	83.9
		SPEX	**80.0 / 81.7**	**79.1 / 80.3**	**87.6 / 88.0**	86.5 / **87.9**	**84.2**
Deepseek	AIME24	CoT	53.3	53.3	53.3	53.3	53.3
		ToT	73.3 / **73.3**	66.7 / 70.0	**80.0** / 76.7	73.3 / 73.3	73.3
		SPEX	**76.7 / 70.0**	**66.7 / 73.3**	76.7 / **76.7**	**76.7 / 73.3**	**73.8**
	AIME25	CoT	56.7	56.7	56.7	56.7	56.7
		ToT	70.0 / **73.3**	70.0 / 70.0	66.7 / 63.3	60.0 / 60.0	66.7
		SPEX	**70.0 / 70.0**	**70.0 / 73.3**	**66.7 / 70.0**	**63.3 / 66.7**	**68.8**
	BRUMO	CoT	60.0	60.0	60.0	60.0	60.0
		ToT	70.0 / **70.0**	73.3 / 73.3	70.0 / 73.3	63.3 / 63.3	69.6
		SPEX	**70.0 / 66.7**	**76.7 / 76.7**	**76.7 / 73.3**	**70.0 / 70.0**	**72.5**
	HMMT	CoT	46.7	46.7	46.7	46.7	46.7
		ToT	60.0 / 60.0	60.0 / 60.0	**60.0 / 60.0**	53.3 / 60.0	59.2
		SPEX	**60.0 / 60.0**	**60.0 / 63.3**	56.7 / 56.7	**60.0 / 60.0**	**59.6**
Qwen	AIME24	CoT	53.3	53.3	53.3	53.3	53.3
		ToT	70.0 / 73.3	70.0 / 73.3	**80.0 / 76.7**	73.3 / 73.3	73.8
		SPEX	**73.3 / 80.0**	**80.0 / 80.0**	76.7 / **76.7**	**76.7 / 80.0**	**77.9**
	AIME25	CoT	60.0	60.0	60.0	60.0	60.0
		ToT	**70.0 / 66.7**	66.7 / 66.7	60.0 / 63.3	63.3 / **60.0**	64.6
		SPEX	63.3 / **70.0**	**66.7 / 66.7**	**70.0 / 70.0**	**63.3 / 56.7**	**65.8**
	BRUMO	CoT	66.7	66.7	66.7	66.7	66.7
		ToT	**73.3 / 76.7**	76.7 / 73.3	**73.3 / 76.7**	70.0 / 73.3	74.2
		SPEX	70.0 / **76.7**	**80.0 / 73.3**	70.0 / **76.7**	**76.7 / 76.7**	**74.6**
	HMMT	CoT	53.3	53.3	53.3	53.3	53.3
		ToT	60.0 / 60.0	56.7 / 60.0	63.3 / 56.7	56.7 / **63.3**	59.6
		SPEX	**63.3 / 60.0**	**60.0 / 60.0**	**63.3 / 60.0**	**60.0 / 60.0**	**60.8**

majority vote.

6.4 Ablation Study

To evaluate the contribution of each key technique in SPEX, we conducted an ablation study as shown in Figure 11. Three techniques were analyzed: T1 (Intra-query Speculative Selection), T2 (Inter-query Budget Allocation), and T3 (Early Termination).

T1 demonstrates significant speedup by allowing speculative execution within queries to exploit parallelism, especially for smaller batch sizes. T2 becomes increasingly critical as batch size grows, ensuring efficient allocation of computational resources when the system transitions from memory-bound to compute-bound. T3 provides a stable speedup of around 1.2× across all configurations by identifying and terminating low-potential branches early.

The combination of all three techniques yields the highest overall performance, with notable improvements in both small-batch and large-batch scenarios.

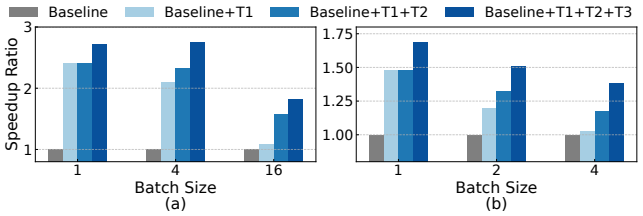


Figure 11: Ablation study of three techniques in SPEX for (a) RSTAR-10 and (b) REBASE-16. T1 represents Intra-query Speculative Selection, T2 represents Inter-query Budget Allocation and T3 represents Early Termination.

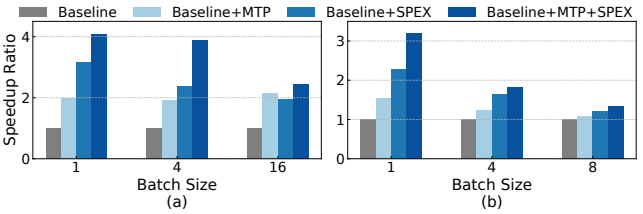


Figure 12: Orthogonality analysis of SPEX and MTP for (a) RSTAR-10 and (b) REBASE-8.

6.5 Compatibility with Speculative Decoding

To demonstrate that SPEX is orthogonal to token-level optimizations, we integrated it with the Multi-Token Prediction (MTP) module of DeepSeek-R1-8B. Since the performance of speculative decoding is highly sensitive to the token tree, we empirically tuned the token tree size based on the batch size. For RSTAR-10, which inherently has a smaller effective batch size due to sequential dependency, we maintained a token tree size of 16 across all settings. For REBASE-8, we adopted an adaptive configuration: a tree size of 16 for batch size (BS) 1, scaled down to 8 for BS=4, and 4 for BS=8, to mitigate compute contention.

Figure 12 illustrates the speedup breakdown. The results confirm that SPEX and MTP target distinct bottlenecks and can be effectively composed. The combination (**Baseline+MTP+SPEX**) consistently yields the highest throughput. Notably, for RSTAR-10 at BS=1, combining SPEX with MTP boosts the speedup from $\sim 2.0\times$ (MTP only) and $\sim 3.1\times$ (SPEX only) to a remarkably $\sim 4.1\times$. As the batch size increases, the system transitions from memory-bound to compute-bound. Consequently, the marginal gain from MTP diminishes. However, SPEX maintains robust acceleration even when token-level speculation saturates.

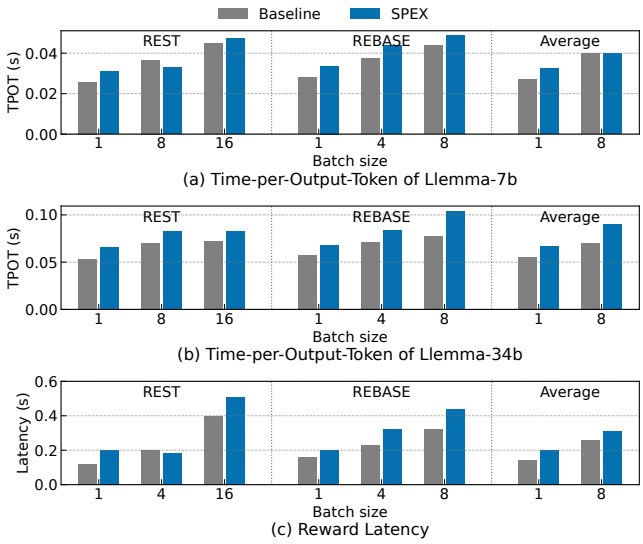


Figure 13: Overhead analysis of SPEX.

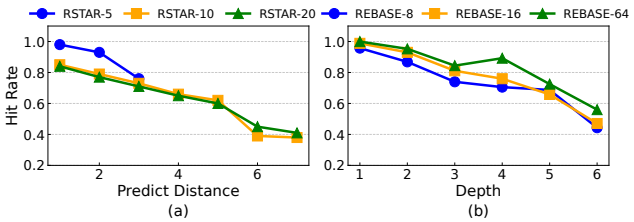


Figure 14: Prediction accuracy evaluation. (a) Hit rate of RSTAR-MCTS for predicting rollouts across different distances. (b) Hit rate of REBASE at varying depths.

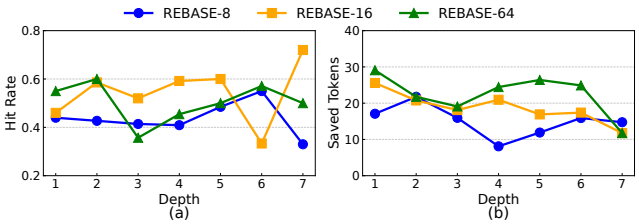


Figure 15: (a) Probability of speculative explorations reaching the critical path for REBASE configurations. (b) Tokens already generated speculatively before the critical path reaches.

6.6 Overhead analysis

As SPEX increases the number of parallel branches within each query, it introduces additional overhead that affects both the reasoning model's Time-per-Output-Token (TPOT) and the reward model's evaluation latency, particularly when the system approaches a compute-bound regime. Figure 13 illustrates these effects.

For the reasoning model's output speed, while SPEX raises the risk of pushing the system into an over-compute-bound state, the observed overhead remains controlled within 15%. This indicates that SPEX effectively balances the trade-off between batch size scaling and computational efficiency.

For the reward model's evaluation latency, the impact of SPEX is minimal. The average delay introduced by SPEX is less than 0.1 seconds, which is negligible compared to the reasoning model's overall processing time. This demonstrates that SPEX's design introduces only minor latency overhead for reward evaluation, ensuring it does not become a bottleneck in the reasoning pipeline.

6.7 Prediction Accuracy

Figure 14 evaluates the prediction accuracy of SPEX. In (a), the hit rate of RSTAR-MCTS decreases as the prediction distance increases, illustrating the challenge of maintaining accuracy in DFS algorithms when predicting further rollouts. In (b), the hit rate of REBASE decreases with increasing depth, as shallow layers contain more low-scoring nodes that are easier to identify, while deeper layers consist of nodes with closer scores, making differentiation more difficult. Therefore, we prefer to allocate SPEX resources evenly for DFS algorithms and prioritize shallow layers for BFS algorithms.

For DFS algorithms, prediction accuracy directly translates into speedup. However, for BFS-only algorithms like REBASE, only speculative expansions along the critical path lead to acceleration. As shown in Figure 15(a), the probability of speculative expansions reaching the critical path ranges from 40% to 60% across different depths. In Figure 15(b), the average number of tokens pre-generated before the critical path reaches them is approximately 20 per depth, highlighting the importance of accurate targeting for efficient performance.

7 Related Work

We categorize related work into three lines and position SPEX with respect to each.

Speculative Decoding. Speculative decoding is an effective method for accelerating memory-bound LLM inference. It uses a smaller draft model to predict future tokens, which are then verified by the main model [6, 28, 61]. Recent advancements have extended this technique to accelerate reasoning models [33, 44, 49, 63]. These techniques operate along a *single linear trajectory*, speculating on the next-token sequence of one reasoning path. In contrast, SPEX speculates over the *tree structure* of reasoning, predicting which branches will be needed before reward feedback is available. As we show in our evaluation, tree-level speculative exploration in SPEX is complementary to token-level speculative decoding and can be composed with it to yield cumulative speedups.

Speculative Tree Search and Speculative Execution. Speculation has also been applied to MCTS and other search procedures. Prior work on speculative MCTS [12, 25] targets AlphaZero-style game search, where the bottleneck is the cost of reward evaluation. These systems speculate on future rollouts to amortize network computation, assuming a compute-bound regime and relatively cheap tree traversal. Our setting is fundamentally different: ToT reasoning with LLMs is predominantly memory-bandwidth-bound due to autoregressive decoding and KV-cache access. SPEX is designed specifically for this regime: it breaks the reward dependency barrier for both Depth-First and Breadth-First Search, incorporates KV-cache reuse and parameter reuse into its scheduling decisions, and coordinates speculation across concurrent queries. More broadly, hardware-level speculative execution mechanisms (e.g., branch prediction, value prediction, and speculation for FSM parallelization [48]) motivate using prediction to unlock latent parallelism; SPEX instantiates this principle at the ToT algorithm and system level for LLM reasoning workloads.

Efficient Reasoning Algorithms. To improve reasoning efficiency, various algorithmic approaches focus on reducing the number of generated tokens. These methods include early exit [10, 17, 32, 67], CoT compression [7, 9, 24, 35, 57, 70], and adaptive reasoning [36–38, 45, 54, 58], which dynamically shorten or adjust reasoning paths based on task complexity or model confidence. These methods primarily optimize the *amount* of test-time compute. SPEX is orthogonal: given a target ToT algorithm and its compute budget, we focus on *how* to schedule and speculate over tree expansions to break the reward dependency barrier and improve hardware efficiency without changing the underlying reasoning algorithm.

8 Conclusion

We propose SPEX, a speculative exploration framework that breaks the reward dependency barrier in Tree-of-Thought reasoning. By introducing intra-query speculative branch selection, inter-query budget allocation, and adaptive early termination, SPEX significantly enhances parallelism and reduces latency. Comprehensive evaluations demonstrate up to 3× speedup for DFS and 1.7× speedup for BFS, all while preserving accuracy. SPEX provides an effective and scalable solution for optimizing ToT reasoning, bridging the gap between algorithmic innovation and system-level performance.

Acknowledgements

This work was supported in part by the National Key Research and Development Program under Grant 2024YFB4505004, in part by NSFC under Grant 62495102, Grant 92464104, and Grant 62341407, in part by Beijing Municipal Science and Technology Program under Grant Z241100004224015, in part by 111 Project under Grant B18001.

References

[1] Zhangir Azerbayev, Hailey Schoelkopf, Keiran Paster, Marco Dos Santos, Stephen McAleer, Albert Q Jiang, Jia Deng, Stella Biderman, and Sean Welleck. Llemma: An open language model for mathematics. *arXiv preprint arXiv:2310.10631*, 2023.

[2] Edward Beeching, Lewis Tunstall, and Sasha Rush. Scaling test-time compute with open models. 2024. Hugging Face Technical Report.

[3] Bradley Brown, Jordan Juravsky, Ryan Ehrlich, Ronald Clark, Quoc V Le, Christopher Ré, and Azalia Mirhoseini. Large language monkeys: Scaling inference compute with repeated sampling. *arXiv preprint arXiv:2407.21787*, 2024.

[4] BRUMO. brown university math olympiad 2025. https://www.brumo.org/, 2025.

[5] Guillaume Chaslot, Sander Bakkes, Istvan Szita, and Pieter Spronck. Monte-carlo tree search: A new framework for game ai. In *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, volume 4, pages 216–217, 2008.

[6] Charlie Chen, Sebastian Borgeaud, Geoffrey Irving, Jean-Baptiste Lespiau, Laurent Sifre, and John Jumper. Accelerating large language model decoding with speculative sampling. *arXiv preprint arXiv:2302.01318*, 2023.

[7] Hongzhan Chen, Siyue Wu, Xiaojun Quan, Rui Wang, Ming Yan, and Ji Zhang. Mcc-kd: Multi-cot consistent knowledge distillation. *arXiv preprint arXiv:2310.14747*, 2023.

[8] Lingjiao Chen, Jared Quincy Davis, Boris Hanin, Peter Bailis, Ion Stoica, Matei A Zaharia, and James Y Zou. Are more llm calls all you need? towards the scaling properties of compound ai systems. *Advances in Neural Information Processing Systems*, 37:45767–45790, 2024.

[9] Xiao Chen, Sihang Zhou, Ke Liang, Xiaoyu Sun, and Xinwang Liu. Skip-thinking: Chunk-wise chain-of-thought distillation enable smaller language models to reason better and faster. *arXiv preprint arXiv:2505.18642*, 2025.

[10] Xingyu Chen, Jiahao Xu, Tian Liang, Zhiwei He, Jianhui Pang, Dian Yu, Linfeng Song, Qiuzhi Liu, Mengfei Zhou, Zhuosheng Zhang, et al. Do not think that much for 2+ 3=? on the overthinking of o1-like llms. *arXiv preprint arXiv:2412.21187*, 2024.

[11] Pengyu Cheng, Yong Dai, Tianhao Hu, Han Xu, Zhisong Zhang, Lei Han, Nan Du, and Xiaolong Li. Self-playing adversarial language game enhances llm reasoning. *Advances in Neural Information Processing Systems*, 37:126515–126543, 2024.

[12] Scott Cheng, Mahmut Taylan Kandemir, and Ding-Yong Hong. Speculative monte-carlo tree search. In *Advances in Neural Information Processing Systems*, pages 88664–88683, 2024.

[13] Zheng Chu, Jingchang Chen, Qianglong Chen, Weijiang Yu, Tao He, Haotian Wang, Weihua Peng, Ming Liu, Bing Qin, and Ting Liu. Navigate through enigmatic labyrinth a survey of chain of thought reasoning: Advances, frontiers and future. *arXiv preprint arXiv:2309.15402*, 2023.

[14] Karl Cobbe, Vineet Kosaraju, Mohammad Bavarian, Mark Chen, Heewoo Jun, Lukasz Kaiser, Matthias Plappert, Jerry Tworek, Jacob Hilton, Reiichiro Nakano, et al. Training verifiers to solve math word problems. *arXiv preprint arXiv:2110.14168*, 2021.

[15] Tri Dao, Dan Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. Flashattention: Fast and memory-efficient exact attention with io-awareness. *Advances in neural information processing systems*, 35:16344–16359, 2022.

[16] Xidong Feng, Ziyu Wan, Muning Wen, Stephen Marcus McAleer, Ying Wen, Weinan Zhang, and Jun Wang. Alphazero-like tree-search can guide large language model decoding and training. *arXiv preprint arXiv:2309.17179*, 2023.

[17] Yichao Fu, Junda Chen, Siqi Zhu, Zheyu Fu, Zhongdongming Dai, Yonghao Zhuang, Yian Ma, Aurick Qiao, Tajana Rosing, Ion Stoica, et al. Efficiently scaling llm reasoning with certaindex. *arXiv preprint arXiv:2412.20993*, 2024.

[18] Yichao Fu, Xuewei Wang, Yuandong Tian, and Jiawei Zhao. Deep think with confidence, 2025.

[19] Zitian Gao, Boye Niu, Xuzheng He, Haotian Xu, Hongzhang Liu, Aiwei Liu, Xuming Hu, and Lijie Wen. Interpretable contrastive monte carlo tree search reasoning. *arXiv preprint arXiv:2410.01707*, 2024.

[20] Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Ruoyu Zhang, Runxin Xu, Qihao Zhu, Shirong Ma, Peiyi Wang, Xiao Bi, et al. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning. *arXiv preprint arXiv:2501.12948*, 2025.

[21] Dan Hendrycks, Collin Burns, Saurav Kadavath, Akul Arora, Steven Basart, Eric Tang, Dawn Song, and Jacob Steinhardt. Measuring mathematical problem solving with the math dataset. *arXiv preprint arXiv:2103.03874*, 2021.

[22] HMMT Organization. Harvard-MIT Mathematics Tournament (HMMT) February 2025. https://www.hmmt.org, 2025.

[23] Coleman Hooper, Sehoon Kim, Suhong Moon, Kerem Dilmen, Monishwaran Maheswaran, Nicholas Lee, Michael W Mahoney, Sophia Shao, Kurt Keutzer, and Amir Gholami. Ets: Efficient tree search for inference-time scaling. *arXiv preprint arXiv:2502.13575*, 2025.

[24] Bairu Hou, Yang Zhang, Jiabao Ji, Yujian Liu, Kaizhi Qian, Jacob Andreas, and Shiyu Chang. Thinkprune: Pruning long chain-of-thought of llms via reinforcement learning. *arXiv preprint arXiv:2504.01296*, 2025.

[25] Juhwan Kim, Byeongmin Kang, and Hyungmin Cho. Specmcts: Accelerating monte carlo tree search using speculative tree traversal. *IEEE Access*, 9:142195–142205, 2021.

[26] Takeshi Kojima, Shixiang Shane Gu, Machel Reid, Yutaka Matsuo, and Yusuke Iwasawa. Large language models are zero-shot reasoners. *Advances in neural information processing systems*, 35:22199–22213, 2022.

[27] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph Gonzalez, Hao Zhang, and Ion Stoica. Efficient memory management for large language model serving with pagedattention. In *Proceedings of the 29th symposium on operating systems principles*, pages 611–626, 2023.

[28] Yaniv Leviathan, Matan Kalman, and Yossi Matias. Fast inference from transformers via speculative decoding. In *International Conference on Machine Learning*, pages 19274–19286. PMLR, 2023.

[29] Jinhao Li, Jiaming Xu, Shan Huang, Yonghua Chen, Wen Li, Jun Liu, Yaoxiu Lian, Jiayi Pan, Li Ding, Hao Zhou, et al. Large language model inference acceleration: A comprehensive hardware perspective. *arXiv preprint arXiv:2410.04466*, 2024.

[30] Sixu Li, Yuzhou Chen, Chaojian Li, Yonggan Fu, Zheng Wang, Zhongzhi Yu, Haoran You, Zhifan Ye, Wei Zhou, Yongan Zhang, and Yingyan (Celine) Lin. Orches: Orchestrated test-time-compute-based llm reasoning on collaborative gpu-pim heterogeneous system. In *Proceedings of the 58th IEEE/ACM International Symposium on Microarchitecture*, MICRO '25, page 476–489, New York, NY, USA, 2025. Association for Computing Machinery.

[31] Yifei Li, Zeqi Lin, Shizhuo Zhang, Qiang Fu, Bei Chen, Jian-Guang Lou, and Weizhu Chen. Making language models better reasoners with step-aware verifier. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 5315–5333, 2023.

[32] Yiwei Li, Peiwen Yuan, Shaoxiong Feng, Boyuan Pan, Xinglin Wang, Bin Sun, Heda Wang, and Kan Li. Escape sky-high cost: Early-stopping self-consistency for multi-step reasoning. *arXiv preprint arXiv:2401.10480*, 2024.

[33] Baohao Liao, Yuhui Xu, Hanze Dong, Junnan Li, Christof Monz, Silvio Savarese, Doyen Sahoo, and Caiming Xiong. Reward-guided speculative decoding for efficient llm reasoning. *arXiv preprint arXiv:2501.19324*, 2025.

[34] Hunter Lightman, Vineet Kosaraju, Yuri Burda, Harrison Edwards, Bowen Baker, Teddy Lee, Jan Leike, John Schulman, Ilya Sutskever, and Karl Cobbe. Let's verify step by step. In *The Twelfth International Conference on Learning Representations*, 2023.

[35] Weizhe Lin, Xing Li, Zhiyuan Yang, Xiaojin Fu, Hui-Ling Zhen, Yaoyuan Wang, Xianzhi Yu, Wulong Liu, Xiaosong Li, and Mingxuan Yuan. Trimr: Verifier-based training-free thinking compression for efficient test-time scaling. *arXiv preprint arXiv:2505.17155*, 2025.

[36] Runze Liu, Junqi Gao, Jian Zhao, Kaiyan Zhang, Xiu Li, Biqing Qi, Wanli Ouyang, and Bowen Zhou. Can 1b llm surpass 405b llm? rethinking compute-optimal test-time scaling. *arXiv preprint arXiv:2502.06703*, 2025.

[37] Jinghui Lu, Haiyang Yu, Siliang Xu, Shiwei Ran, Guozhi Tang, Siqi Wang, Bin Shan, Teng Fu, Hao Feng, Jingqun Tang, et al. Prolonged reasoning is not all you need: Certainty-based adaptive routing for efficient llm/mllm reasoning. *arXiv preprint arXiv:2505.15154*, 2025.

[38] Feng Luo, Yu-Neng Chuang, Guanchu Wang, Hoang Anh Duy Le, Shaochen Zhong, Hongyi Liu, Jiayi Yuan, Yang Sui, Vladimir Braverman, Vipin Chaudhary, et al. Autol2s: Auto long-short reasoning for efficient large language models. *arXiv preprint arXiv:2505.22662*, 2025.

[39] Mathematical Association of America. American Invitational Mathematics Examination (AIME) 2024. https://www.maa.org/math-competitions/aime, 2024.

[40] Mathematical Association of America. American Invitational Mathematics Examination (AIME) 2025. https://www.maa.org/math-competitions/aime, 2025.

[41] Xuefei Ning, Zinan Lin, Zixuan Zhou, Zifu Wang, Huazhong Yang, and Yu Wang. Skeleton-of-thought: Prompting llms for efficient parallel generation. *arXiv preprint arXiv:2307.15337*, 2023.

[42] OpenAI. Learning to reason with llms. Technical report, OpenAI, 2024. Technical Report.

[43] OpenAI. OpenAI O3-mini System Card. Technical report, OpenAI, January 2025. Technical Report.

[44] Rui Pan, Yinwei Dai, Zhihao Zhang, Gabriele Oliaro, Zhihao Jia, and Ravi Netravali. Specreason: Fast and accurate inference-time compute via speculative reasoning. *arXiv preprint arXiv:2504.07891*, 2025.

[45] Penghui Qi, Zichen Liu, Tianyu Pang, Chao Du, Wee Sun Lee, and Min Lin. Optimizing anytime reasoning via budget relative policy optimization. *arXiv preprint arXiv:2505.13438*, 2025.

[46] Zhenting Qi, Mingyuan Ma, Jiahang Xu, Li Lyna Zhang, Fan Yang, and Mao Yang. Mutual reasoning makes smaller llms stronger problem-solvers. *arXiv preprint arXiv:2408.06195*, 2024.

[47] Jiahao Qiu, Yifu Lu, Yifan Zeng, Jiacheng Guo, Jiayi Geng, Huazheng Wang, Kaixuan Huang, Yue Wu, and Mengdi Wang. Treebon: Enhancing inference-time alignment with speculative tree-search and best-of-n sampling. *arXiv preprint arXiv:2410.16033*, 2024.

[48] Junqiao Qiu, Xiaofan Sun, Amir Hossein Nodehi Sabet, and Zhijia Zhao. Scalable fsm parallelization via path fusion and higher-order speculation. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '21, page 887–901, New York, NY, USA, 2021. Association for Computing Machinery.

[49] Junhan Shi, Yijia Zhu, Zhenning Shi, Dan Zhao, Qing Li, and Yong Jiang. Speccot: Accelerating chain-of-thought reasoning through speculative exploration. In *ES-FoMo III: 3rd Workshop on Efficient Systems for Foundation Models*.

[50] Charlie Snell, Jaehoon Lee, Kelvin Xu, and Aviral Kumar. Scaling llm test-time compute optimally can be more effective than scaling model parameters. *arXiv preprint arXiv:2408.03314*, 2024.

[51] NovaSky Team. Sky-t1: Train your own o1 preview model within $450, 2025.

[52] Qwen Team. Qwq: Reflect deeply on the boundaries of the unknown. *Hugging Face*, 2024.

[53] Qwen Team. Qwen3 technical report, 2025.

[54] Xu Wan, Wei Wang, Wenyue Xu, Wotao Yin, Jie Song, and Mingyang Sun. Adapthink: Adaptive thinking preferences for reasoning language model. *arXiv preprint arXiv:2506.18237*, 2025.

[55] Chaojie Wang, Yanchen Deng, Zhiyi Lyu, Liang Zeng, Jujie He, Shuicheng Yan, and Bo An. Q*: Improving multi-step reasoning for llms with deliberative planning. *arXiv preprint arXiv:2406.14283*, 2024.

[56] Jun Wang, Meng Fang, Ziyu Wan, Muning Wen, Jiachen Zhu, Anjie Liu, Ziqin Gong, Yan Song, Lei Chen, Lionel M Ni, et al. Openr: An open source framework for advanced reasoning with large language models. *arXiv preprint arXiv:2410.09671*, 2024.

[57] Yibo Wang, Li Shen, Huanjin Yao, Tiansheng Huang, Rui Liu, Naiqiang Tan, Jiaxing Huang, Kai Zhang, and Dacheng Tao. R1-compress: Long chain-of-thought compression via chunk compression and search. *arXiv preprint arXiv:2505.16838*, 2025.

[58] Yunhao Wang, Yuhao Zhang, Tinghao Yu, Can Xu, Feng Zhang, and Fengzong Lian. Adaptive deep reasoning: Triggering deep thinking when needed. *arXiv preprint arXiv:2505.20101*, 2025.

[59] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, Denny Zhou, et al. Chain-of-thought prompting elicits reasoning in large language models. *Advances in neural information processing systems*, 35:24824–24837, 2022.

[60] Yangzhen Wu, Zhiqing Sun, Shanda Li, Sean Welleck, and Yiming Yang. Inference scaling laws: An empirical analysis of compute-optimal inference for llm problem-solving. In *The Thirteenth International Conference on Learning Representations*, 2025.

[61] Heming Xia, Zhe Yang, Qingxiu Dong, Peiyi Wang, Yongqi Li, Tao Ge, Tianyu Liu, Wenjie Li, and Zhifang Sui. Unlocking efficiency in large language model inference: A comprehensive survey of speculative decoding. *arXiv preprint arXiv:2401.07851*, 2024.

[62] Yuxi Xie, Anirudh Goyal, Wenyue Zheng, Min-Yen Kan, Timothy P Lillicrap, Kenji Kawaguchi, and Michael Shieh. Monte carlo tree search boosts reasoning via iterative preference learning. *arXiv preprint arXiv:2405.00451*, 2024.

[63] Wang Yang, Xiang Yue, Vipin Chaudhary, and Xiaotian Han. Speculative thinking: Enhancing small-model reasoning with large model guidance at inference time. *arXiv preprint arXiv:2504.12329*, 2025.

[64] Shunyu Yao, Dian Yu, Jeffrey Zhao, Izhak Shafran, Tom Griffiths, Yuan Cao, and Karthik Narasimhan. Tree of thoughts: Deliberate problem solving with large language models. *Advances in neural information processing systems*, 36:11809–11822, 2023.

[65] Zihao Ye, Lequn Chen, Ruihang Lai, Wuwei Lin, Yineng Zhang, Stephanie Wang, Tianqi Chen, Baris Kasikci, Vinod Grover, Arvind Krishnamurthy, et al. Flashinfer: Efficient and customizable attention engine for llm inference serving. *arXiv preprint arXiv:2501.01005*, 2025.

[66] Lifan Yuan, Ganqu Cui, Hanbin Wang, Ning Ding, Xingyao Wang, Jia Deng, Boji Shan, Huimin Chen, Ruobing Xie, Yankai Lin, et al. Advancing llm reasoning generalists with preference trees. *arXiv preprint arXiv:2404.02078*, 2024.

[67] Linan Yue, Yichao Du, Yizhi Wang, Weibo Gao, Fangzhou Yao, Li Wang, Ye Liu, Ziyu Xu, Qi Liu, Shimin Di, et al. Don't overthink it: A survey of efficient r1-style large reasoning models. *arXiv preprint arXiv:2508.02120*, 2025.

[68] Dan Zhang, Sining Zhoubian, Ziniu Hu, Yisong Yue, Yuxiao Dong, and Jie Tang. Rest-mcts*: Llm self-training via process reward guided tree search. *Advances in Neural Information Processing Systems*, 37:64735–64772, 2024.

[69] Qin Zhang, Yangbin Yu, QIANG FU, Deheng Ye, et al. More agents is all you need. *Transactions on Machine Learning Research*.

[70] Shangziqi Zhao, Jiahao Yuan, Guisong Yang, and Usman Naseem. Can pruning improve reasoning? revisiting long-cot compression with capability in mind for better reasoning. *arXiv preprint arXiv:2505.14582*, 2025.

[71] Lianmin Zheng, Liangsheng Yin, Zhiqiang Xie, Chuyue Livia Sun, Jeff Huang, Cody Hao Yu, Shiyi Cao, Christos Kozyrakis, Ion Stoica, Joseph E Gonzalez, et al. Sglang: Efficient execution of structured language model programs. *Advances in neural information processing systems*, 37:62557–62583, 2024.

[72] Anni Zou, Zhuosheng Zhang, Hai Zhao, and Xiangru Tang. Generalizable chain-of-thought prompting in mixed-task scenarios with large language models. *arXiv preprint arXiv:2310.06692*, 2023.

ARTIFACT EVALUATED
USENIX
AVAILABLE

Controlling Opaque-Component Effects with Semisolates and Try

https://binpa.sh/try

Evangelos Lamprou*
Brown University

Tianyu (Ezri) Zhu*
Stevens Institute of Technology

Di Jin
Brown University

Grigoris Ntousakis
Brown University

Georgios Liargkovas
Columbia University

Calvin Eng
Brown University

Konstantinos Kallas
UCLA

Michael Greenberg
Stevens Institute of Technology

Nikos Vasilakis
Brown University

Abstract

Many developers and systems today rely on opaque software components. When executing, these components affect each other and the broader environment in which they execute. Some of these effects are expected and desired; others not so. This paper introduces semisolates, an abstraction and corresponding subsystem for controlling and manipulating the effects of opaque components. Available as an unprivileged, higher-order, language-agnostic command, `try` interposes on a component's execution to automatically capture and control its effects. Effect control includes introspection, optional application, effect stacking, and further manipulation—all driven by several real-world case studies. Today `try` is used in research and production applications across several organizations, mediating potentially undesired effects, maintaining full compatibility with real-world components, and incurring a modest performance overhead well within each case's acceptable levels.

1 Introduction

Modern software systems contain many intercommunicating components, including various subsystems, programs, commands, and packages [11, 44, 45, 93]. These components are often available in diverse programming languages, interact opaquely with the broader environment, and are developed by developers of varying skill and care. Even quick-and-dirty data-processing scripts pull together many polyglot components [44, 71] and simple update scripts or AI agents invoke other commands to complete their jobs [45, 93].

Unfortunately, the execution of these components—during *e.g.*, exploration or development—results in unpredictable and often-irreversible effects. These effects are a direct and necessary byproduct of the computation encoded by a component: executing `npm install` downloads and installs new software packages, directly altering several files and directories [1]. At times, effects are unintentional: naively running a `find -exec` as suggested by an LLM may result in deleting critical user files [52]. These and other cases (§2) hint at the necessity to control these effects during component execution.

Figure 1: Applying `try`. Execute a command in its own `try` semisolate to control its effects: inspect, selectively apply, revert, partially hide (all shown) or otherwise manipulate them—all within the *current* environment rather than a new isolated container.

This paper presents a new abstraction, the *semisolate* (§3), and corresponding implementation, `try` (§4), for controlling the effects of such opaque components. Semisolates offer the ability to execute a component in the current environment via an unprivileged, language-agnostic, higher-order command that allows its users to inspect and manipulate a completely unmodified component's externally observable behavior. But rather than focusing on the behavior encoded in standard streams, easily controllable by the usual pipe-and-filter patterns, `try` focuses on *side* effects such as filesystem modifications. And contrary to various forms of containment and virtualization [10, 54, 68], which create a new environment that fully isolates all effects, `try` allows the executing components to access the current environment in which they execute and enables effect manipulation directly in that environment; the term *semisolate* (from *semi-*, meaning "partially," and *isolate*) reflects this partial rather than complete isolation.

More specifically, `try` allows introspecting, selectively applying, stacking, and further manipulating a component's effects to the filesystem and the broader environment. To automatically capture these effects, `try` limits the component's view of the filesystem: it first sets up a *semisolate* to create a view of the filesystem and environment that retains partial isolation; it then executes the component inside the semisolate, collecting effects into a writable temporary directory; finally, either during its (streaming) execution or upon completion, the semisolate allows manipulating the collected effects by

*Equal contribution.

staging them, reverting them or applying them to the underlying filesystem—either wholly or partially (Fig. 1).

This opaque effect control is applicable broadly (§2): it can be used to inspect the effects of risky or cryptic LLM suggestions before applying them [26]; track the dependency graph for systems composed of opaque components, useful for parallelization or reordering [65]; provide a `--dry-run` capability, found today in some commands, that is general and command-agnostic [89]; and support specification inference for opaque components [45]. These and other examples drive `try`'s design requirements: effect control that allows hiding state, stacking effects, and partially or fully applying or rolling them back. No current abstraction and corresponding subsystem offers granular effect control with `try`'s wealth of features.

Applying `try` to real-world cases (§5) indicates that `try` controls all relevant component effects, maintains behavior equivalence with non-semisolated execution, and incurs a modest performance overhead relative to the needs of each use case. Semisolates and `try` offer a distinct set of features from, and better performance than, container systems such as Docker, OrbStack, and Podman [54], built using similar components arranged quite differently.

In summary, this paper makes several contributions: it

- identifies the shared needs of several case studies for which today's effect control is insufficient (§2);
- presents a novel effect-control abstraction, a semisolate, that meets the needs of these case studies (§3);
- describes the implementation of `try`, a new effect-control subsystem built for Linux environments (§4);
- characterizes `try` by applying it to several real-world components drawn from the earlier case studies (§5).

It also discusses `try`'s limitations and the experiences of organizations using `try` (§6), as well as prior related work (§7). Appendix A contains all LLM prompts used in the evaluation, Appendix B collects several testimonials on `try`'s potential impact, and Appendix C describes the artifact accompanying this paper. The `try` subsystem is open-source, MIT-licensed software, publicly available on GitHub [92].

2 Examples and Goals

To understand the need for controlling component effects, this section describes several real-world case studies (§2.1). Combined, these case studies highlight several shared requirements for the design of a flexible effect-control abstraction and corresponding subsystem (§2.2).

2.1 Motivating Examples

Risky or cryptic LLM suggestions: Recent agentic systems have incorporated large language models (LLMs), allowing them to act on behalf of the developer by executing arbitrary commands inside the host system [2, 3, 7]. Consider using an LLM to generate a script that "cleans up its containing directory"; it generates `delete.sh` containing the following:

```
rm -fr "$(cd "${0%/*}" && echo $PWD)"/*
```

Unfortunately, when called from some paths, the runtime expansion `${0%/*}` will expand to the script's path, `cd` will fail, and `rm` will delete everything user-writable. Prefixing this line with `try` indicates the deletion of all user-writable directories and offers the option to discard these changes.

```
try sh delete.sh
  /home (deleted)
  /etc (deleted)
  ...
  Apply these changes? [y/i/N]
```

A developer or system calling `try` can decide whether to inspect these changes and choose to discard them, or inspect them individually and choose which ones to apply.

Dependency tracking: Speculative execution reordering offers significant speedups [65], but ensuring it respects read-after-write dependencies across opaque components is particularly challenging. Consider the following program that compiles three and links two `C` files:

```
gcc -c a.c -o a.o; gcc b.c -o a
gcc -c c.c -o c.o; gcc a.o c.o -o b
```

The second call to `gcc` does not depend on the first, thus it can be executed in parallel. Unfortunately, detecting this opportunity automatically at runtime requires detecting effects and postponing their application to the broader environment until they are confirmed to not violate read-after-write dependencies—at times discarding these effects to effectively roll back command misspeculation. Crucially, subsequent invocations must start their execution with the illusion that prior effects have been applied—necessitating chaining reversible effect application across component invocations.

Prefixing `gcc` with `try` while explicitly asking it to stack sets of effects solves both problems:

```
try -t trace1.log -N env1 gcc -c c.c -o c.o
try -t trace2.log -N env2 -L env1 gcc a.o c.o -o b
```

The `-N` option instructs `try` to store, instead of directly applying, `gcc`'s effects for later use with a configurable name. The `-t` flag enables system-call tracing to capture ordered filesystem effects. Next, `-L` injects `gcc`'s uncommitted effects from `env1` into its own view, making `c.o` visible to `try`'s second invocation.

Third-party library risks: Public third-party libraries amplify developer capabilities, but risk introducing undesired effects [15, 56, 63]. Consider the popular `node-ipc` library [72], invoked as part of a larger codebase:

```
npm test node-ipc
```

Earlier versions of `node-ipc` included undesired effects modifying critical system files [18, 86]. Prefixing `npm test` with `try` highlights files about to be modified without making any modifications prior to approval:

```
try npm test node-ipc
  /etc/passwd (modified)
  /bin/bash (modified)
  /sbin/init (modified)
  ...
  Commit these changes? [y/i/N]
```

The output of `try` shows all such effects, allowing a developer to review any unexpected effects and, if necessary, selectively apply only the relevant ones. It is also possible to filter or aggregate over the set of effects to make review easier (§6).

Dry-run execution: Some commands offer specially configured dry-run modes (e.g., `--dry-run`, `--fake`, `--no-commit`, `--print`, `--just-print`, `--simulate`, `-n`, `-s`) that show the effects of simulating a command's execution [67]. Consider an invocation about to clean temporary files in the current directory using `find`:

```
find . -name '*tmp*' -delete
```

Unfortunately, `find` does not offer a built-in dry-run functionality. Prefixing the command with `try -n` instead executes the command in a semisolate, collecting and presenting its actual effects to the user while discarding them.

```
try -n find . -name '*tmp*' -delete
  ./tmp_build/ (deleted)
  ./tmp.log (deleted)
```

Using `try`, a user sees the concrete effects that the command's execution would have produced, even if it does not implement built-in dry-run functionality.

Partial-specification mining: Automated acceleration of programs that use opaque components [38, 71] depends on partial component specifications—*e.g.*, the order in which a component reads its input streams [34]. Generating such specifications automatically requires exploring a command's invocation space while carefully collecting the effects of different invocations. Consider the following invocation of `grep`:

```
grep -oE [a-zA-Z]+ one.txt two.txt
```

Unfortunately, understanding that `grep` first reads `one.txt` and *then* reads `two.txt` requires a more detailed view of its effects. Prefixing `grep` with `try -t` offers a detailed, ordered log of `grep`'s read and write effects:

```
try -t trace.log grep -oE [a-zA-Z]+ one.txt two.txt
```

The resulting `trace.log` file contains information that can then be compiled into a partial specification for `grep`.

```
r /usr/bin/grep
r /usr/lib/libc.so.6
r one.txt
r two.txt
w stdout
```

This order of effects extracted by `try -t` reveals invariants (*e.g.*, `one.txt` preceding `two.txt`) that need to be maintained when optimizing programs that invoke `grep`.

2.2 Desiderata for the Missing Abstraction

The aforementioned case studies illustrate an underlying common need to abstract away effect control. A suitable effect-control abstraction has several core requirements: (1, **I**) *effect introspection*, *i.e.*, the ability to detect, collect, present, and inspect a component's side effects in the same environment directly executing the component; (2, **A**) *selective application*, *i.e.*, the ability to decide whether to apply effects to the broader environment or roll them back; (3, **S**) *effect stacking*, allowing commands to execute in the context of earlier effects that have not been applied yet, potentially rolling back multiple stacks of effects; (4, **M**) *further effect manipulation* beyond direct application or discard, including partial application of only some effects, effect recording and delayed propagation, and selective hiding of the underlying environment. These requirements typically need to be met in the context of the current executing environment—*not* a virtualized or contained environment fully isolated from the current environment.

Apart from these core requirements, the abstraction's applicability and reach would further benefit from an implementation that combines streaming support, allowing its users to manipulate the effects of long-running components as they are generated; configurable effect handlers for tuning the granularity of effects; and runtime performance overheads that do not affect interactive use. These features are not essential to the core abstraction—*i.e.*, a batch-only, non-configurable, slow implementation would still fit all of §2.1's case studies—but they broaden its applicability and reach.

Other characteristics such as complete mediation against arbitrary adversarial behavior are out of scope: the abstraction aims at the inconvenient or undesired effects of everyday mistakes or accidents—not actively malicious, `try`-aware components trying to bypass its underlying mechanisms [49].

3 Abstracting Away Effect Control

To meet the aforementioned desiderata (§2.2) and abstract away effect control, `try` offers a language-agnostic and configurable abstraction that launches one or more components in a semisolate. A semisolate offers a constrained and configurable (Tab. 1) private view of the real host environment—*i.e.*, not a fresh environment in a container or virtual machine. In the example below, `sh` runs in the exact same environment as

Table 1: Control configuration and `try` flags. Controlling effects with `try` supports significant configurability during semisolate creation, execution, and conclusion—including core options (in black, *Cf.*§3) and peripheral features and subsystems (faded, *Cf.*§6).

Semisolate stage	Configuration	Flag
Creation	Stack over prior effects	`-L`
	Name current effect group	`-N`
	Ignore specific paths	`-i`
Execution	Disable the network	`-x`
	Create filesystem trace	`-t`
	Collect effect delta	`--diff`
Conclusion	Apply all effects	`-y`
	Discard all effects	`-n`
	Inspect effects	`-e`
	Include, exclude specific paths	`-I`, `-E`
	Summarize effects	`-s`
	Offer friendlier summary	`-h`

`curl`, but within a private view that controls, manipulates, and potentially discards the application of effects to the broader environment:

```
curl -sL https://sh.rustup.rs | try sh
```

This private view can be configured to, for example, completely mirror the global filesystem, partially mirror it while hiding subtrees, or correspond to one or more layers of as-yet-unapplied effects from previous `try` executions.

Semisolates: The abstraction of a semisolate provides the illusion of direct access to its outer environment and supports flexible effect manipulation. Semisolates collect the effects of opaque components in this private view and expose them to their callers for further manipulation. Using semisolates, `try` allows its callers to (1) selectively apply or discard these effects, with options to view partial earlier state; (2) collect the effects for possible application later; and (3) stack and expose as-yet-unapplied effects across invocations. This design allows the effect-mediated component to execute in partial isolation, without affecting the global filesystem until the caller explicitly consents.

To achieve effect control, a semisolate operates in three configurable stages: (1) *creation*: the semisolate sets up the appropriate view for the executing component to ensure appropriate effect mediation; (2) *execution*: the semisolate launches the target component within a private view, collecting relevant effects; and (3) *conclusion*: the semisolate optionally and selectively applies recorded effects into the main environment.

All three stages typically operate on the current directory and are highly configurable to support the nuances of various real-world uses (§2). The next few paragraphs detail such tuning, both at the level of individual stages and the current `try` flags summarized in Tab. 1, after first establishing which effects are controllable and their level of control.

Which effects are controllable: Unix and its descendants offer many well-studied, fine-grained tools for the pipe-and-filter abstractions—for program *output*. There are only coarser, less well-studied tools to help manage program *effects*. Analogizing the Unix environment to a single executable, existing abstractions focus on function inputs and outputs (Unix: standard streams, exit status), but not the heap (Unix: the filesystem). Semisolates offer a new, fine-grained tool for effect management. They add abstractions for manipulating effects around a child command—for example, the initial filesystem view, the final filesystem view, and the filesystem access trace. They can optionally be configured to support additional system, network, and stream effects as part of a unified abstraction.

Effects are made available to the semisolate caller as a combination of (1) filesystem paths or stream identifiers, and (2) optional metadata such as effect type and success. Each path can be a valid path pointing to an existing file or directory; it can also be a *non*-existent path that might lead to access failure—still useful when a command relies on a file's absence to exhibit the intended behavior. The effect type includes information such as read-only, write-only, and read-write: a read-write effect is, for example, `open`ing a file with any flag other than `O_RDONLY`; a read-only effect includes operations that only interrogate a file's metadata through, for example, `stat`ing a file; a write effect includes modifying filesystem metadata through, for example, `chmod`ing or `unlink`ing a file. Each recorded effect instance is accompanied by metadata that indicates whether an effect succeeded; or—if it failed—the reason for its failure.

By default, `try` manipulates effects across entire process subtrees. For example, if a `try`-controlled process launches a subprocess that writes to the filesystem, then `try` will include those write effects in the group of effects directly available for manipulation. A parent process can also use `try` to wrap and programmatically control child effects.

Semisolate creation: Upon creation, a semisolate configures the private view of the executing component—to allow both stacking the effects of different semisolates and prevent unintended effects (including reads), useful for hiding sensitive information from the executing component. Configuring this initial view translates to setting up a virtual-filesystem environment before any modifications to the filesystem.

Upon creation, semisolates may configure effect stacking and hiding. Stacking configures the component's initial view to import the effects produced from a previous semisolate. Selective hiding masks certain parts of the underlying filesystem from the executing component, given a specification that describes a filesystem subtree. These are implemented by `try`'s interface as `-L`, which accepts the name of a previous semisolate, itself produced by a previous call to `try` with the `-N` flag; and `-i`, which takes as input a regular expression that matches filesystem paths to be hidden from the component's view. The latter is semantically equivalent to stacking the

effects of `try -n N rm dir1 dir2...` over `try -L N`—but `-i` is more efficient as it directly sets up exclusions without first running an additional (potentially expensive) command.

Semisolate execution: During execution, semisolates additionally configure the classes of effects to interpose on and the corresponding interposition granularity—for example, whether to include system-call tracing. The current classes of effects include filesystem effects, network effects, signals, and interprocess streams.

The current interface of `try` supports: `-t`, which makes `try` produce a filesystem access trace, containing an ordered list of effects; and `-x`, which prevents the component from accessing the network. Apart from the effect path, this detailed view consists of the mode and the return code, which can be inspected to infer negative effects like failed accesses.

Semisolate conclusion: Upon conclusion, semisolates offer the ability to decide which effects to apply and which to discard, eventually generating the final filesystem view. They allow selective effect application at multiple granularities, from binary all-or-nothing to per-effect application, while also supporting path-oriented filtering. They can also hold individual effects and allow inspecting them directly using conventional Unix abstractions.

These options can be configured when a semisolate concludes. The current interface supports inspecting individual effects (`-e`) and, based on predefined caller policies, programmatically filtering these changes (`-E` and `-I`). It also supports applying, discarding, or mixing them (`-y`, `-n`, and `-e`). And `-N` stores effects in a named effect directory, potentially used earlier during creation with `-L`.

4 The Try Subsystem

This section describes how `try` implements the aforementioned abstraction (§3) for programs executing in Linux userspace. It first outlines `try`'s use of union filesystems, namespaces, and system-call tracing before diving into the details of semisolate creation (§4.1), execution (§4.2), and conclusion (§4.3), closing with a detailed walkthrough of multiple `try` invocations put together (§4.4).

Union filesystems: To achieve effect control, `try` needs to interpose on the component's view of the filesystem. This interposition is responsible for manipulating the underlying filesystem view, creating a partially private view that allows accessing the underlying contents while capturing and containing effects. This private view is implemented as a union filesystem, configured appropriately to either mirror the underlying filesystem, partially mirror it with sensitive paths hidden, or correspond to the as-yet-uncommitted state of a previously executed run of `try`. To contain effects, `try` uses the configured union filesystem to record filesystem changes into a temporary directory—without affecting the *host environment* until effects are applied.

To instantiate the concept of effect *layers*, `try` leverages OverlayFS—a specific implementation of a union filesystem part of the Linux kernel [24]. Union filesystems are a class of filesystems that allow multiple directories to be presented as a single unified directory without modifying the underlying kernel structures [70]. A union filesystem layers an upper directory (*upperdir*) and one or more lower directories (*lowerdirs*) into a single merged view—without merging their underlying data.

In the merged view, the content at a specific path `p` is determined by searching for `p` in the upperdir, followed by all lowerdirs in order until `p` is found. The first match is presented in the merged view; if `p` exists in multiple layers, content from the higher layer takes precedence.

Changes to the merged view are forwarded to the upperdir and persist inside it, even after unmounting the union filesystem directory. Changing a file at `p` in the merged view places the new version of the file at `p` in the upperdir. Removing `p` places a *whiteout file* at the corresponding upperdir location.

Namespaces: To execute privileged actions—while executing in an unprivileged manner—and interpose on the execution context of the component, `try` leverages Linux process namespaces. Namespaces isolate resources such as the network, pseudo-filesystems like `procfs`, permissions, and signals from the rest of the system, offering (groups of) processes the illusion of exclusive ownership of those resources—enabling the creation of a private (and controllable) view of the underlying system state. For example, the PID namespace assigns new PIDs to processes inside the namespace: processes in the namespace can only see and interact with each other using the PIDs assigned in the namespace.

Process-related interactions such as signaling between processes and adjustment of scheduling priorities are contained to the namespace. Components in `try` are only partly able to interact with ambient resources and objects such as processes, users, or network devices—with `try` interposing on all accesses to the broader system.

System-call tracing: To observe the order of a component's effects, and go beyond effects that modify the filesystem, `try` leverages system-call tracing—interposing between the operating system kernel and user-space components to monitor and record their interactions. Tracing commands such as `strace` on Linux offer visibility into exact calls performed by a process, their order, and their arguments, as well as the signals received by the process. Combined, they offer a wealth of information about the process—including filesystem operations, process management, inter-process communication, and network access. By selectively enabling tracing when necessary, `try` extracts ordered, finer-grained information about component effects beyond filesystem modifications.

Algorithm 1 Semisolate creation. Key steps in setting up a new semisolate, before running a component;

Require: *path*
1: **if** *path* is not provided **then**
2: `    path ← mktemp -d`
3: **end if** ▷ Set up directories
4: **if** *path*'s filesystem is overlayfs **then**
5: `    mount -t tmpfs tmpfs path`
6: **end if**
7: **for all** *dir* **in** *topLevelDirs* **do**
8: `    mkdir -p path/{upperdir,workdir,tmp}/dir`
9: **end for**
10: `unshare ...` ▷ Prepare namespaces
11: **for all** *dir* **in** *topLevelDirs* **do**
12: `    mount -t overlayfs ...path/tmp/dir`
13: **if** overlay failed **then**
14: `        mergerfs path/tmp/dir...`
15: `        mount -t overlayfs...path/tmp/dir`
16: **end if**
17: **end for**
18: `mount path/tmp/dev/...` ▷ Mount devices
19: `unshare ...` ▷ Enter environment

4.1 Semisolate creation

Each semisolate uses its own special writable directory to record effects—the `upperdir`—and a second temporary directory—the `workdir`—used by the union filesystem for internal bookkeeping (Alg. 1, lines 1–5). Each of these directories can be (1) a new, appropriately prefixed temporary directory created with `mktemp`; (2) a new directory whose name is provided by the `try` caller; or (3) an existing directory, potentially used in previous `try` invocations.

OverlayFS does not support certain filesystems to host the upperdir or the `workdir`. One such filesystem is OverlayFS itself, a common scenario when using `try` within Docker. In this case, `try` mounts an additional `tmpfs` over the semisolate directory (Alg. 1, line 5). This ensures that all of `try`'s side effects prior to the conclusion stage are confined within the semisolate directory.

Setting up mounting options: To bypass OverlayFS's restriction of overlapping the lower and upper directory, `try` creates an individual union filesystem mount for each top-level directory—*e.g.*, `/bin`, `/usr`, and `/home`—instead of using the root directory `/` as the union filesystem's lower directory.

Specifically, `try` first creates directories `upperdir`, `workdir`, and `tmproot` (Alg. 1, lines 5–12). In these top-level directories, it then creates a child directory for each top-level directory in the filesystem, such as `/etc`, `/usr`, and `/home`. Additional lower directories already containing effects are added to the mount options during this step—layered on top of the corresponding union filesystem merged views. The `tmproot` directory will become the final merged view presented to the executing component (§4.2).

Once inside the new namespace, `try` iterates over the mount options constructed in the previous step and mounts the union filesystem (Alg. 1, lines 12–14). It mounts the `/proc`, `/dev`, and `/devpts` pseudo-filesystems within the `tmproot` directory and symlinks all special files through which the component interacts with its standard input and output streams—*i.e.*, `/proc/self/fd/{0,1,2}` and `/dev/{stdin,stdout,stderr,fd,pts}`. Using bind-mounts [23], it duplicates device files such as `zero`, `null`, and `random`, into the `tmproot`'s `/dev` directory.

To implement file hiding, `try` creates an additional directory called `excldir`. This `excldir` directory is mounted as an intermediate layer right above the lower directory and contains whiteout files corresponding to files that will be hidden from the merged view.

Submount permissions: Unfortunately, filesystem submounts within any OverlayFS layer cannot be unmounted by the current user: mount points inside layers are ignored because these layers correspond to overlapping filesystem fragments. Consider an initial filesystem *A* that contains paths `/x`, `/x/y`, and `/x/y/z`, and a new filesystem *B* mounted at `/x/y`, effectively hiding `/x/y/z` as contents at that mount point by *B*. If `/x` is specified as an OverlayFS lower directory, it will include only the contents of *A* and ignore the submount to *B*—thus including `/x/y/z` in the merged view.

As `try` aims to limit access to information otherwise inaccessible without its use, it prevents unprivileged access to directories with submounts as lower or upper layers. Although the fresh mount namespace grants `try` the capability to create new overlay mounts, among other root-like capabilities, it is forbidden to unmount existing submounts.

To address this problem, `try` uses an additional union filesystem to "flatten" all content within the directory into a single filesystem (Alg. 1, line 14) when submounts are present. The union filesystem used by `try` for this purpose is mergerfs [62], which combines the entire directory, including its submounts, into a single, unified filesystem view. This contrasts with OverlayFS, which does not support submounts. By creating this flattened view of the directory, submounts are effectively eliminated, allowing OverlayFS to use the directory as a `lowerdir` without issues.

4.2 Semisolate execution

After preparing a semisolate, `try` launches the effect-controlled component in its own dedicated namespace and optionally traces its execution.

Execution namespacing: The semisolate executes components within a new user, PID, mount, and network namespace (Alg. 1, line 10). There are two distinct reasons for which `try` uses namespaces: (1) namespaces offer controlled isolation from the broader environment, restricting the process's view so that all effects—together with the filesys-

tem view shaped by the semisolate (§4.1)—are contained, (2) namespaces create the illusion of administrative privileges, allowing a semisolate to mount `/proc` and the union filesystem itself, which would otherwise require elevated privileges. The capability for user namespaces requires setting the `unprivileged_userns_clone` kernel flag, enabled by default in most modern Linux distributions and depended on by container runtimes like Podman [30]. This avoids the need to implement `try` as a `setuid` binary requiring elevated privileges to execute.

Finally, `try` enters a second new namespace (Alg. 1, line 19) to isolate the component's process tree from `try`'s own process, setting the component's root directory to `tmproot` using `unshare`, and finally executes the component under optional system-call tracing.

Execution tracing: To extract additional information about filesystem effects, `try` can interpose on relevant system-call invocations using `strace`. Rather than tracing all filesystem-related system calls when they occur, `try` focuses on collecting relevant information upon `open`—as this is typically a necessary step before subsequent operations on the file descriptor such as reads, writes, random accesses, or direct memory mapping. This approach lowers `try`'s performance overhead, as it avoids tracing reads, writes, and other latency-critical data manipulation operations, omitting further tracing during subsequent use of the file descriptor.

To improve its reach while still avoiding finer-grained tracing, `try` additionally traces other system calls that take paths as arguments—*e.g.*, `open`, `mkdir`, and `unlink`. To trap only this preconfigured set of system calls, `try` uses Seccomp-BPF [42] with a predefined filter—but allows users to pass additional filters if they need to track additional effects, including operations on already-opened file descriptors.

Non-existent paths: Non-existent or missing paths pose a challenge, as ignoring their accesses risks introducing effect holes: accessing non-existent paths generates an observable effect—*e.g.*, negative return value from `open`. Therefore, `try` records such accesses as negative dependencies. Such negative dependencies are useful for recording effects of searching dynamic library paths and configuration files.

4.3 Semisolate conclusion

Semisolates conclude their execution by going through all the effects in the `upperdir` and either applying predefined policies or logging effects to one of the standard streams. As all changes are captured in the effect directory, applying these effects amounts to carefully comparing the captured effects to the underlying filesystem and environment.

To apply filesystem-related effects, `try` recursively processes the upper directory in a preorder fashion, moving files out of the semisolate and into the real filesystem by handling directories before their contents. For regular files in the upper directory, `try` replaces the original file with the new file. For directories, it either (1) replaces the original file with the new directory, if a regular file exists at the directory's original path; (2) replaces the original file or directory with the new file, if the directory has the extended attribute `user.overlay.opaque`, indicating that directories in lower layers have been shadowed by new definitions in the merged view; or (3) replaces the original directory with the new directory. For symlinks, it removes any existing file at the original path and recreates the symlink with the same target as the one in the upper directory. If `try` encounters a whiteout file—a character device with 0 major and minor numbers—in the upperdir, it deletes the original file or directory at that path.

If the semisolate directory resides on the same filesystem as the target location, `rename` is used to avoid unnecessary disk copying. Otherwise, `try` falls back to the host's `mv` utility.

4.4 Putting it all together

Fig. 2 illustrates the internals of four `try` invocations.

The first invocation wraps p_1, eliding all network access inside the semisolate. It prepares an overlay layer with a whiteout mask at path π, and mounts it on top of the host filesystem's lowerdir, ignoring the underlying π in the filesystem. It then enters the new namespace, and launches p_1 inside it. While p_1 is running, `try` captures five effects in the upperdir, which it later processes to summarize the order of effects (`-t`): (1) reading α hits the lowerdir, which forwards the call to the underlying filesystem, (2) writing to β hits the upperdir, copying β there, (3) reading β hits the upperdir, which sees the latest write, (4) reading path π, masked in the overlay layer, thus failing and resulting in `try` recording a negative dependency; and deleting path δ, which places a mask file at δ in the upperdir, effectively hiding it from the merged view. The sequence of collected effects (α_R, β_W, β_R, π_R, δ_D) is fed to `stdout` but the effects themselves are discarded alongside the semisolate (`-n`).

Next, `try` wraps p_2, creating a new namespace and mounting the host filesystem inside a fresh overlay filesystem lowerdir, now without any path masking or network isolation. The three component effects it captures are: (1) writing β, with identical effects as before, (2) writing ψ, and (3) deleting δ—both operations captured in the upperdir. This time, `try` stores the semisolate (and by extension, its effects) inside a named directory on the host filesystem (`-N` σ), after filtering out excluded effects (`-E` ψ) by iterating over the upperdir and deleting them.

Next `try` wraps p_3 while stacking the effects of the latest `try` invocation (`-L` σ). It captures: (1) writing α, which brings it to the upperdir, (2) reading β, which hits the stacked lowerdir from semisolate σ, and (3) writing π—recorded in the upperdir. The effects stored in the semisolate are the two writes to α and π, and the delete of δ inherited from σ.

Finally, `try` commits the effects (`-y`). It iterates from bottom to top, first applying the effects from the stacked semiso-

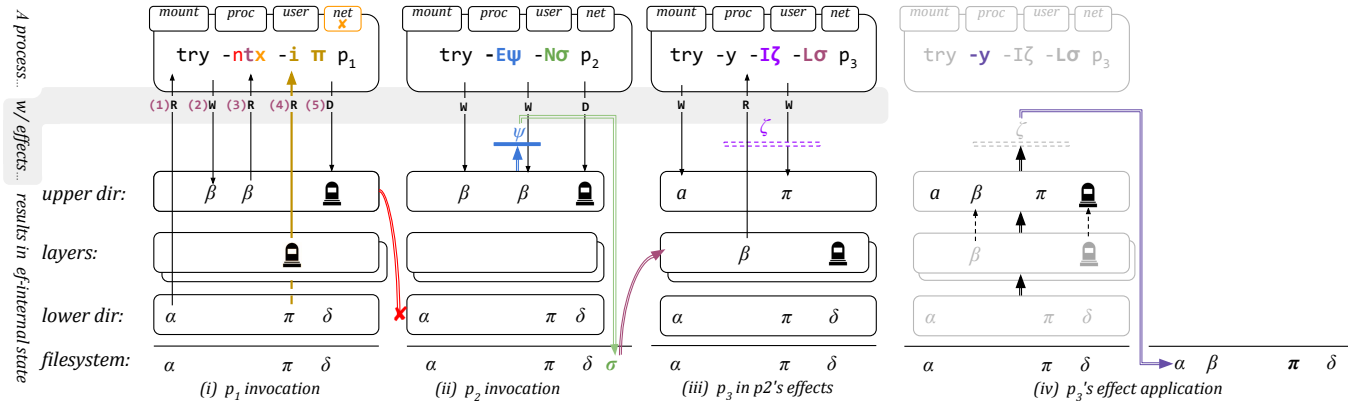


Figure 2: Detailed walkthrough of the internals of multiple `try` invocations. A detailed walkthrough of these invocations is presented in §4.4. From top to bottom: `try`'s invocation, component effects, `try` internal state, and host filesystem state. From left to right: different `try` invocations on different components on the same host filesystem. Single arrows indicate effects; double arrows indicate effect discard, storage, stacking, and application; and certain colors correspond to certain `try` invocation options.

late σ, followed by the current semisolate's effects, applying each one based on its type. It focuses on paths in ζ (`-I ζ`), *i.e.*, the writes to β and π, which are applied to the host system.

This approach implements the semisolate abstraction, allowing for effect control on arbitrary components without requiring any invasive instrumentation or modification.

5 Evaluation

This section reports on the results of applying `try` in several instances of each case-study application presented earlier (§2). Key dimensions of each study's characterization (§5.1–5.5) include (1) `try`'s ability to *control* all relevant effects; (2) `try`'s behavioral *equivalence* to semisolate-free execution when effects are finally applied to the host environment; and (3) `try`'s runtime *performance* relative to semisolate-free execution and full isolation (§2.2). A series of microbenchmarks zoom into `try`'s sources of overhead (§5.6).

Control focuses on confirming that `try` correctly identifies **`I`**, applies or discards **`A`**, selects **`S`**, or manipulates **`M`** all relevant effects during a component's execution in a semisolate.

Behavioral equivalence focuses on confirming two aspects of `try`: (1) the in-semisolate behavior during execution, meaning that the computation in the semisolate executes as expected and produces results identical to the non-semisolated execution; and (2) the eventual state of the filesystem, meaning that the filesystem state is indistinguishable from `try`-free execution. The former confirms that `try` does not affect the internal computation, including its direct effects; and the latter confirms that it correctly applies the manipulated effects.

Performance characterization compares runtime without isolation, with `try`, and with Docker v28.4.0 configured with full isolation; performing container (not image) creation, input copy-in, execution, container teardown, and copy-out. Experiments use an environment with 64 GB of RAM and an 8-core 3.70 GHz Intel Xeon W-2145. Software used is the Linux kernel v6.1 on Debian GNU/Linux 13 (trixie), GNU coreutils v9.7, Node.js v18.19.0, and Python v3.13.5.

Benchmark results are summarized in Table 2. In terms of effect control, `try` mediates desired effects, propagates them appropriately, and hides sensitive environment state. In terms of behavioral equivalence, `try` behaves equivalently to `try`-free execution, confirmed both via manual inspection and automated checksum comparison between the entire filesystem tree after executing each benchmark with and without `try`. In terms of performance, `try` incurs a 1.0×–8.3× performance overhead over vanilla, `try`-free execution and enjoys a 1.1×–225.7× performance improvement over containment in Docker. We do not provide a geomean or other statistical summary for overhead over vanilla execution or speedup over Docker, as each case study requires different semisolate configurations and corresponding `try` flags to control effects—and thus incur different overheads.

We discuss the benchmarks grouped by use case below.

5.1 Risky or cryptic LLM suggestions

These benchmarks consist of five scripts generated by ChatGPT for tasks distilled from a survey of LLM-mediated workflows, spanning conventional chat interfaces and autonomous agents such as Codex and OpenClaw [82] (prompts included in Appendix §A): (1) **`crawl`**, updates the timestamps of 10,000 files in a directory, using `find`, `exec`, and `touch` [90]; (2) **`fresh`**, compressing 10,000 10KB files that exceed a certain size, using `find`, `exec`, and `gzip` [35]; (3) **`archive`**, searching for files with a `.txt` extension in 100 directories of 10,000 files each, using `find` [37]; (4) **`logs`**, checking for errors in a 10-million-line file using `grep` [40]; (5) **`order`**, sorting a 10-million-line file using `sort` [6]. Benchmarks were executed with `try -y` and `try -n`.

Table 2: Benchmark summary. The benchmark applications for the five use cases evaluated. The **Control** column indicates `try`'s effect control between effect introspection (**I**), application and discard (**A**), stacking (**S**), and manipulation (**M**). The **Equivalence** columns indicate whether the output (O) and side effects (SE) match those of vanilla execution. The **Performance** columns show the **execution time ratio** relative to the baseline (Docker or Vanilla), and whether `try` comparatively has a speedup (↑) or slowdown (↓). The **Description** column describes each benchmark's purpose and `try`'s invocation. The **Source** column provides references to the original benchmark sources.

Use case	Name	Control		Equivalence O	Equivalence SE	Performance vs. Docker	Performance vs. Vanilla	Description	Source
Risky or cryptic LLM suggestions (§5.1)	**crawl**	✓	I A M	✓	✓	1.4×↑	1.4×↓	`try` LLM-generated shell pipelines	[6, 35, 37, 40, 90]
	fresh	✓	I A M	✓	✓	1.3×↑	1.8×↓		
	archive	✓	I A M	✓	✓	225.7×↑	1.3×↓		
	logs	✓	I A M	✓	✓	58.2×↑	1.9×↓		
	order	✓	I A M	✓	✓	2.0×↑	1.1×↓		
Dependency tracking (§5.2)	**covid**	✓	I A S M	✓	✓	1.5×↑	1.0×↓	`try -t -L` Script with dependency tracking	[13, 19, 41, 94, 95]
	nlp	✓	I A S M	✓	✓	3.4×↑	1.2×↓		
	spell	✓	I A S M	✓	✓	2.3×↑	1.0×↓		
	unixfun	✓	I A S M	✓	✓	18.7×↑	1.7×↓		
	ci-cd	✓	I A S M	✓	✓	1.1×↑	1.0×↓		
Third-party library risks (§5.3)	**LinOTP**	✓	I A M	✓	✓	2.0×↑	1.2×↓	`try -i` Git repos with risky pre-commit hooks	[8, 28, 64, 66, 73]
	frogmouth	✓	I A M	✓	✓	4.2×↑	1.0×↓		
	kibble	✓	I A M	✓	✓	1.4×↑	1.2×↓		
	okteto	✓	I A M	✓	✓	1.7×↑	1.3×↓		
	uv-metrics	✓	I A M	✓	✓	3.1×↑	1.0×↓		
Dry-run execution (§5.4)	**git**	✓	I A	✓	✓	3.2×↑	69.8×↓	`try -n` Dry-run command invocations	[27, 31, 32, 75, 96]
	patch	✓	I A	✓	✓	3.0×↑	82.6×↓		
	rename	✓	I A	✓	✓	2.8×↑	4.9×↓		
	rsync	✓	I A	✓	✓	1.2×↑	7.8×↓		
	stow	✓	I A	✓	✓	2.8×↑	6.3×↓		
Partial-specification mining (§5.5)	**cp**	✓	I A S	✓	✓	2.2×↑	8.3×↓	`try -t` Standard Unix utilities	[38, 71]
	ls	✓	I A S	✓	✓	1.9×↑	3.7×↓		
	rm	✓	I A S	✓	✓	1.6×↑	7.1×↓		
	sed	✓	I A S	✓	✓	1.9×↑	8.1×↓		
	xargs	✓	I A S	✓	✓	1.6×↑	1.7×↓		

Control: All computations executing in semisolates configured with `try -n` mediated all effects that were applied to the filesystem by `try`-free executions.

Equivalence: All computations executing in semisolates configured with `try -y` produce correct results; and their effects on the filesystem are identical to `try`-free executions.

Performance: When compared to vanilla execution, `try` introduces runtime overheads 1.1×–1.9×. Variations come from the complexity of effects interposed during execution. For example, in **`order`**, `try` achieves the lowest overhead (1.1×) as it interposes on a single write effect, writing the sorted file to disk. The **`crawl`**, **`fresh`**, and **`archive`** tasks exhibit higher overheads (up to 1.9×) because they operate on many files from the host system (use of `find`), which results in more upperdir copy-ups. Relative to Docker-contained execution, `try` demonstrates substantial performance improvements across all five benchmarks, achieving performance improvements of 1.3×–225.7×. Two benchmarks incur outlier speedups: the **`archive`** benchmark only interacts with its standard output stream (`find`'s output) which is effectively a symlink to the same file descriptor in the host environment, meaning that `try` boils down to vanilla execution, paying only the flat cost of semisolate creation, while Docker still pays the full container setup and teardown costs. Similarly, the **`logs`** benchmark reads from and writes to a single large file, meaning that again `try` pays only a single creation and copy-up cost, while Docker with full isolation needs to pay the copy-in and copy-out costs as well.

5.2 Dependency tracking

These benchmarks consist of several real-world shell programs from the Koala suite [44] with many and complex runtime dependencies between their components. (1) the **`covid`** benchmark, which processes bus schedule data collected during COVID-19 to generate summary statistics [94]; (2) the **`nlp`** benchmark, which performs natural language processing tasks on a text corpus [41]; (3) the **`spell`** benchmark, which

checks spelling in a large text file [13]; (4) the **unixfun** benchmark, which solves challenges from Unix 50th anniversary game [95]; and (5) the **ci-cd** benchmark, which compiles several C source files [19]. These programs contain a diverse set of commands that include GNU Coreutils, POSIX, and third-party commands. To stack effects across all component invocations without applying them on the host environment (§2.1), `try` was configured with `-L` and with `-t` to extract dependencies.

Control: All computations executed with `try -t -n -L` mediated all effects that were applied to the filesystem by `try`-free executions. Manual inspection and multiple re-executions confirm that executing these benchmarks with `try -t -L` identifies all true dependencies between components.

Equivalence: Tracking the dependencies discovered by `try -t` and later applying them with `try -y -L` produces correct results. Their effects on the filesystem are identical to the ones produced by `try`-free executions.

Performance: When compared to vanilla execution, `try` introduces absolute overheads 1.0×–1.7×, which is the flat cost of setting up the semisolate environment for each tracked command. However, while the semisolate setup cost is flat, tracing overheads scale with the number of effects a component produces. This is less evident in statically compiled components, such as commands from GNU coreutils, which are tightly packed binaries with very minimal dependencies (often just `libc`), but evident with components relying on interpreted runtimes (*e.g.*, a custom `lowercase.py` utility in **nlp**), which need to resolve their entire dependency tree during execution. The performance of `try` significantly outperforms fully-isolated execution across all five dependency tracking benchmarks, demonstrating speedups 1.1×–18.7×.

5.3 Third-party library risks

These benchmarks consist of `git` repositories with pre-commit hooks that introduce undesirable effects: (1) **LinOTP** [64] multi-factor authenticator; (2) the **frogmouth** [28] markdown reader; (3) Apache **kibble** [8] aggregating and visualizing software data; (4) the **okteto** [66] container orchestrator; and (5) **uv-metrics** [73], a machine-learning metrics reporter. The pre-commit hooks perform several operations on the filesystem, communicate with the network, and include a code-injection attack that leads to data exfiltration. They were prefixed with `try -i`, excluding the `/etc/passwd` file from the semisolate.

Control: All computations executed with `try -i` mediated across all effects present in the `try`-free executions, except `/etc/passwd`. This effect originated from the pre-commit hook at `.git/hooks`, which attempted to append `/etc/passwd` to `README.md`.

Equivalence: All computations executed with `try -i` produce correct results, correctly committing the intended changes, and the resulting filesystem state remains identical to that of `try`-free execution. Across both the vanilla pre-commit hook workloads and the five code-injection vulnerabilities evaluated, `try` preserves correctness.

Performance: When compared to vanilla execution, `try` has a performance impact of up to a 1.3×. The performance of `try` is comparable to vanilla execution for the **frogmouth** and **uv-metrics**, while maintaining low overheads for **LinOTP** (1.2×), **okteto** (1.3×), and **kibble** (1.2×), whose pre-commit hooks perform short-running computations, comparable with `try`'s semisolate setup overhead. It outperforms Docker-contained execution across all five benchmarks, achieving speedups 1.4×–4.2×. With blocking enabled, `try` reduces the effect surface and thus computation performed by the injected pre-commit hooks. This leads to `try` improving the runtime of the **frogmouth** and **kibble** workloads by 0.8× and 0.9× respectively.

5.4 Dry-run execution

These benchmarks consist of commands with built-in dry-run functionality: (1) **git** [31], using `git clean`; (2) **patch** [32]; (3) **rename** [96]; (4) **rsync** [75]; and (5) **stow** [27]. Each workload was executed twice: once with dry-run enabled, without `try`; once with `try -n`, without dry-run.

Control: All computations executed with `try -n` except two mediated on and reported the exact same effects as the corresponding `--dry-run` executions. One of the two exceptions was **rename**: instead of reporting `a -> b`, `try` reports equivalent `a (deleted) b (created)` pairs.

The second exception is more interesting: when combined with the `--log-file` flag, **rsync**'s dry-run mode omits effects in the produced log file [74], whereas `try` captures and reports all produced effects.

Equivalence: All computations executed with `try -n` correctly suppress all effects to the host environment and present the same output and side effects as the dry-run executions (excluding **rename**'s representation difference). Invocations executed with `try -y` correctly apply all effects to the host environment, equivalent to `try`-free execution.

Performance: When compared to dry-run execution `try` demonstrates overheads of 4.9×–82.6×. These overheads are expected since each command's native dry-run mode avoids performing any filesystem action and is implemented as a separate path that does not perform any effectful computation (average time 0.03s), whereas `try` executes the real computation and then discards its effects (average time 0.26s). Compared to full isolation in Docker, `try` improves performance across all five benchmarks between 1.2×–3.2×.

5.5 Partial-specification mining

These benchmarks evaluate `try`'s capability to support automatic specification mining for shell commands, focusing on determining the order of their read and write effects to their standard streams or the filesystem. These properties correspond to the ones required by PaSh [38] and POSH [71] to parallelize or distribute shell scripts. The commands evaluated in this section are: (1) **`ls`**, a command that lists directory contents; (2) **`cp`**, a command that copies files and directories; (3) **`rm`**, a command that removes filesystem entries; (4) **`sed`**, a stream editor that filters and transforms text; and (5) **`xargs`**, a higher-order command that executes command lines from standard input. The specification miner explores 11, 100, 40, 1408, and 3 configurations for these commands respectively. For **`xargs`**, it uses the `rm`, `ls`, and `cat` commands as arguments. All workloads were executed using `try -t`.

Control: All computations executed with `try -t` mediate all observed effects, capturing every read and write operation performed by the evaluated commands.

Equivalence: All computations executed with `try -t` produce correct results, and the resulting filesystem state remains identical to that of `try`-free execution. Across all commands evaluated, the inferred specifications match or extend PaSh's handwritten specifications. In three cases, `try` enables the miner to detect effects omitted from the original PaSh specifications. For example, although **`cp`** is marked as side-effectful in the original specifications, its behavior in these evaluation scenarios is effectively pure, as it modifies only files explicitly listed in its arguments.

Performance: Since the configuration miner needs to explore a large number of command invocations to infer their specifications, `try`'s overheads compound significantly. Compared to vanilla execution, `try` increases execution time 1.7×–8.3×, caused by `try`'s flat cost of setting up the semisolate environment and its tracing overhead. Vanilla execution is not meaningful in this use case, as the specification miner needs to observe effects to infer specifications. The performance of `try` improves relative to Docker-contained execution across all five mining benchmarks, achieving speedups 1.6×–2.2×. Benefits from `try`'s lighter semisolate setup and teardown times compound over the many invocations required by the specification miner (here, 1562 in total).

5.6 Microbenchmarks

The goal of the set of microbenchmarks is to analyze `try`'s sources of overhead. These benchmarks consist of three workloads: (1) **`echo`**, a Rust function that creates zero files, invoked 100 times; (2) **`files`**, a Rust function that creates 10,000 small files of size 16 bytes each; and (3) **`big_file`**, a Rust function that creates five large files of size 1GB each. All workloads were executed using `try -y` and `try -n`, yielding complete control and equivalence, thus we focus on performance breakdown (Fig. 3).

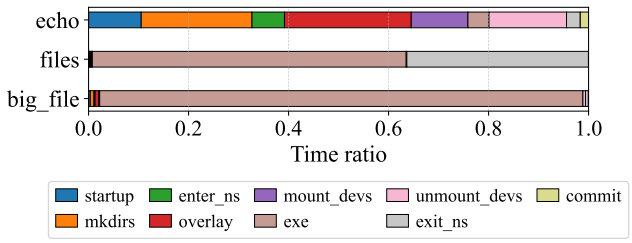


Figure 3: The performance breakdown of `try`. The performance of `try` in three microbenchmarks and its overhead breakdown across different execution stages.

The performance analysis breaks down the `try` runtime into nine stages: (i) *startup* parses command-line arguments and enumerates top-level directories; (ii) *mkdirs* provisions the upper, work, and temporary directories; (iii) *enter_ns* creates a fresh namespace; (iv) *overlay* establishes the OverlayFS mount points; (v) *mount_devs* bind-mounts selected device files; (vi) *exe* executes the target process; (vii) *unmount_devs* unmounts the device files; (viii) *exit_ns* exits the namespace, triggering the automatic unmount of remaining OverlayFS layers; and (ix) *commit* moves output files from the upper directory to the host filesystem. For the baseline **`echo`** command, which completes in 165ms, the overhead is caused by filesystem and namespace operations. Creating the OverlayFS (*overlay*) accounts for 25% (42ms), followed by directory creation (*mkdirs*) at 22% (37ms). Device management also creates measurable overhead, with *unmount_devs* taking 16% (26ms) and *mount_devs* taking 11% (19ms). In longer-running benchmarks, such as **`big_file`** (9.2s) and **`files`** (11s), these setup and teardown costs remain roughly constant, making the relative overhead comparable to the actual *exe* time. However, **`files`** presents an outlier in the teardown phase as *exit_ns* takes 6.3s. This latency is caused at semisolate conclusion because `try` needs to iterate over the many files created in the OverlayFS `upperdir`.

6 Discussion, Limitations & Experiences

Limitations: Current `try` limitations primarily impact compatibility in specific scenarios: restrictions and altered application behavior due to its use of user namespaces, challenges in handling pseudo-filesystems and physical devices, and inability to fully disable certain isolation mechanisms.

Due to its use of user namespaces, `try`-semisolated processes cannot interact with other users. Even when a process appears to be root in the namespace, it cannot switch UID or operate on files owned by other users. Components that rely on creating or switching users may behave unexpectedly.

Processes inside semisolates bypass writable permissions without needing to change the file permissions (a special

feature of UID 0 in Linux). This may affect components that rely on write permissions or ones running without elevated privileges, such as `npm`, when the user ID is 0.

As `try` requires user and PID namespaces for unprivileged OverlayFS mounting, it creates a non-configurable isolation barrier for IPC mechanisms such as signals, complicating interprocess coordination when such isolation is not desired.

External modules: Two additional modules augment `try`. For namespace-local root-like capabilities without host root privileges, `try` maps the current effective UID and GID to the superuser inside the new user namespace. Because `unshare` cannot map multiple UID or GID ranges, traversing directories owned by another accessible GID can trigger `EOVERFLOW`; `try-gidmapper` addresses this case, with security implications that go beyond `try`'s current scope.

As the number of effects can overwhelm a user, an external `try` module summarizes effects in a human-friendlier form. This module, called `try-summarize`, queries an opaque language model with the output from `try` and additional instructions requesting (1) a non-expert summary of the effects, starting from the most important ones; (2) information about whether these effects touch sensitive directories. Both prompts offer functionality `try`'s users could directly implement, but proved useful during recent user interactions with `try`.

Experiences: Colleagues and students using `try` have shared several notable experiences (see Appendix B for testimonials unrelated to the `try` authors). The `try` subsystem is also used in several research projects. An out-of-order execution engine called hS [48] and a bolt-on incrementalization system called Incr [99] use `try` to build a dependency graph between opaque components during execution and revoke their effects when appropriate. They achieve speedups of up to 9.3× and 373.3× over vanilla execution, respectively.

Users often discard effects when the wrapped component wrote outside the expected filesystem subtree, *e.g.*, creating auxiliary files in the home directory, or when its set of effects was larger than expected, *e.g.*, `chmod +x *` affecting all files in the current directory. In several cases, they discarded the effects of mistyped `pip` and `cpanm` invocations. A colleague, after running a training job under `try`, inspected the produced checkpoint files and selectively committed only the highest-performing one. A teaching assistant used `try` to shield against unintended effects from student submissions. Several students have used `try` to interpose on agentic workflows, wrapping their session with `try` and inspecting all produced effects before committing them. Even when the agent performed non-filesystem accesses (*e.g.*, access to webpages, interactions with email and messaging APIs), which often were read-only network effects, `try` remained useful. It allowed students to mediate all resulting local filesystem effects, critical when the agent performed an incorrect action based on misleading information it fetched from the web.

It is possible for users to arrive at an inconsistent set of updates when selectively applying or reordering effects. For example, the effect of creating file `f` in directory `d` depends on the effect of creating directory `d`. When `try` applies an inconsistent update, it issues a warning similar to standard filesystem manipulation commands (*e.g.*, `mv`). Since the semisolate is stored on disk, users are able to re-apply a different subset of effects or reorder semisolates differently—a capability that hS depends on [48].

7 Related Work

Containment and virtualization: There has been a significant amount of work on full isolation, including in containers [9, 54, 84, 87] and virtual machines [10, 12, 16, 53, 68]. These primitives create an isolated environment that is different from the host environment they execute in, completely avoiding sharing of any sort. Semisolates instead run a program in a simulacrum of the current environment—as if on the same environment, but with significantly lower risk—and allow for further and selective manipulation of its effects.

Software isolation and confinement: Earlier work on isolation and confinement [4, 36, 39, 69, 76, 78], focused on controlling and, if necessary, protecting against harmful component effects. These approaches offer structures at the operating system level, often not supporting unmodified Linux environments. In contrast, `try` leverages subsystems available in modern operating systems—requiring no modifications.

Tools such as bubblewrap [17] similarly rely on Linux kernel mechanisms to create isolated execution environments for unmodified programs. However, bubblewrap requires configuring the environment view available to the wrapped process ahead-of-execution, does not support layering, and either blocks effects during execution—potentially leaving the component at an inconsistent state—or reflects effects immediately to the host environment. In contrast, `try` requires no configuration (aside from optional path ignoring), supports layering unapplied effects across executions, and allows users to inspect and selectively apply effects *after* execution.

Other systems achieve confinement via system call interposition [14, 43]. For example, MBOX traces and interposes on system calls to inform the user of unintended or undesired effects. Such interposition meets only some of `try`'s requirements; it does not, for example, support stacking or further manipulation. Semisolates and `try` take a different approach, offering several more features—and compatibility across several modern environments.

Language-based wrapping and proxying: Semisolates bear resemblance to earlier research that attempts to control effects and capabilities using dynamic effect wrapping [5, 21, 51, 55, 57, 58]. In contrast to `try`, these approaches are not language-agnostic, as they are tightly integrated into a language and its runtime environment [5, 20, 21, 56], and often require developer effort through code modifications, or isolation policies [50, 57, 58, 61, 91]. Semisolates offer language-agnostic

effect control on unmodified programs.

Capabilities and sandboxing: Semisolates inherit ideas from early capability and sandboxing work [46,47,59,60,83]. Software fault isolation [97] rewrites object code to prevent execution reaching addresses outside a domain. Rather than completely restricting naming, `try` dynamically shadows names defined outside the compartment's context—a technique isomorphic to a combination of capability revocation followed by attenuation; but contrary to earlier efforts, `try`'s shadowing works with completely unmodified binaries.

Filtering and access prevention: Another class of sandboxing tools is based on software fault isolation [97], including NaCl [81], WASM [33], and RLBox [63]. These systems focus on intraprocess memory-level isolation. Instead, `try` supports effect control (not isolation) outside the process—including effects such as file modifications.

Other systems control effects by preventing file accesses [77, 98, 100], terminating processes that access files outside an allowed set, or filtering dangerous or unnecessary system calls [22, 29, 42, 85]. These approaches prevent effects completely—possibly breaking the application if it needs to perform the effects to complete its execution.

Transactional filesystems: Earlier research on transactional filesystems [25, 79, 80, 88] has developed support for atomically committing or reverting changes to multiple files in a filesystem. In contrast to `try`, which allows calling programs to inspect and apply some of the effects—or further manipulate them at will.

8 Conclusion

Software systems consist of many components, written in many languages, and interacting with each other and the environment. This paper showcases the need for and feasibility of building a new abstraction, the semisolate, for controlling potentially undesired effects, maintaining full compatibility with real-world components, and incurring a performance overhead that is aligned with several practical applications.

Acknowledgements

We thank the anonymous OSDI'26 reviewers and our shepherd for their feedback; the OSDI'26 Artifact Evaluation reviewers for their time; and the Brown CS2952R (Fall'24) participants for their input on several iterations of this paper. This material is based upon research supported by NSF awards CCF-2525351, CCF-2525352, CCF-2525353, CNS-2247687, CNS-2312346, and CNS-2312347, DARPA contract no. HR001124C0486, an Amazon Research Award (Fall 2024), a Google ML-and-Systems Junior Faculty Award, a seed grant from Brown University's Data Science Institute, and a BrownCS Faculty Innovation Award.

References

[1] eslint-scope vulnerability. https://security.snyk.io/vuln/SNYK-JS-ESLINTSCOPE-11120, 2024. Accessed: 2024-09-29.

[2] Claude code. https://claude.com/product/claude-code, 2025. Accessed: 2025-12-10.

[3] Codex: Lightweight coding agent that runs in your terminal. https://github.com/openai/codex, 2025. Accessed: 2025-08-29.

[4] Mike Accetta, Robert Baron, William Bolosky, David Golub, Richard Rashid, Avadis Tevanian, and Michael Young. Mach: A New Kernel Foundation for UNIX Development. In *USENIX Technical Conference*, 1986.

[5] Pieter Agten, Steven Van Acker, Yoran Brondsema, Phu H Phung, Lieven Desmet, and Frank Piessens. JSand: Complete Client-Side Sandboxing of Third-Party JavaScript without Browser Modifications. In *Annual Computer Security Applications Conference (ACSAC)*, pages 1–10, 2012.

[6] Ali Sajid. How to sort a file in-place? https://stackoverflow.com/questions/29244351/how-to-sort-a-file-in-place, 2015. Accessed: 2025-12-04.

[7] Cursor / Anysphere. Cursor: The best way to code with ai. https://cursor.com/, 2025. Accessed: 2025-09-25.

[8] Apache Kibble Developers. Apache kibble. https://github.com/apache/kibble.

[9] Sergei Arnautov, Bohdan Trach, Franz Gregor, Thomas Knauth, Andre Martin, Christian Priebe, Joshua Lind, Divya Muthukumaran, Dan O'Keeffe, Mark L. Stillwell, David Goltzsche, Dave Eyers, Rüdiger Kapitza, Peter Pietzuch, and Christof Fetzer. SCONE: Secure Linux containers with Intel SGX. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 689–703, Savannah, GA, November 2016. USENIX Association.

[10] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. *ACM SIGOPS Operating Systems Review*, 37(5):164–177, 2003.

[11] Len Bass, Paul Clements, and Rick Kazman. *Software architecture in practice*. Addison-Wesley Professional, 2021.

[12] Muli Ben-Yehuda, Michael D. Day, Zvi Dubitzky, Michael Factor, Nadav Har'El, Abel Gordon, Anthony Liguori, Orit Wasserman, and Ben-Ami Yassour. The turtles project: Design and implementation of nested virtualization. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Vancouver, BC, October 2010. USENIX Association.

[13] Bentley, Jon and Knuth, Don and McIlroy, Doug. Programming pearls: a literate program. *CACM*, 29(6):471–483, June 1986.

[14] Antonio Bianchi, Yanick Fratantonio, Christopher Kruegel, and Giovanni Vigna. Njas: Sandboxing unmodified applications in non-rooted devices running stock android. In *Proceedings of the 5th Annual ACM CCS Workshop on Security and Privacy in Smartphones and Mobile Devices*, pages 27–38, 2015.

[15] Andrea Bittau, Petr Marchenko, Mark Handley, and Brad Karp. Wedge: Splitting applications into reduced-privilege compartments. In *Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation*, NSDI'08, pages 309–322, Berkeley, CA, USA, 2008. USENIX Association.

[16] Xiaoxin Chen, Tal Garfinkel, E. Christopher Lewis, Pratap Subrahmanyam, Carl A. Waldspurger, Dan Boneh, Jeffrey Dwoskin, and Dan R.K. Ports. Overshadow: a virtualization-based approach to retrofitting protection in commodity operating systems. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XIII, page 2–13, New York, NY, USA, 2008. Association for Computing Machinery.

[17] Bubblewrap Contributors. Bubblewrap. https://github.com/containers/bubblewrap. Accessed: 2025-09-25.

[18] MITRE Corporation. Cve-2022-23812: Vulnerability details. https://www.cve.org/CVERecord?id=CVE-2022-23812, 2025. Accessed: 2025-01-02.

[19] Charlie Curtsinger and Daniel W. Barowy. Riker: Always-Correct and fast incremental builds from simple specifications. In *USENIX Annual Technical Conference (ATC)*, pages 885–898, Carlsbad, CA, July 2022. USENIX Association.

[20] Tom Van Cutsem. Membranes in JavaScript. https://tvcutsem.github.io/js-membranes, 2012. Accessed: 2020-03-16.

[21] Willem De Groef, Fabio Massacci, and Frank Piessens. NodeSentry: Least-privilege Library Integration for Server-Side JavaScript. In *Annual Computer Security Applications Conference (ACSAC)*, pages 446–455, 2014.

[22] Nicholas DeMarinis, Kent Williams-King, Di Jin, Rodrigo Fonseca, and Vasileios P Kemerlis. Sysfilter: Automated system call filtering for commodity software. In *International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*, pages 459–474, 2020.

[23] Linux Kernel Documentation. mount(2) — Linux manual page, 2024. Available at https://man7.org/linux/man-pages/man2/mount.2.html.

[24] Linux Kernel Documentation. Overlayfs - Linux kernel documentation, 2024. Available at https://www.kernel.org/doc/html/latest/filesystems/overlayfs.html.

[25] Robert Escriva and Emin Gun Sirer. The design and implementation of the warp transactional filesystem. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 469–483, Santa Clara, CA, March 2016. USENIX Association.

[26] fegome90-cmd. Cursor ai executes destructive command during development session. https://forum.cursor.com/t/cursor-ai-executes-destructive-command-rm-rf-during-development-session/129401, 2025. Accessed: 2025-09-24.

[27] Free Software Foundation. GNU Stow Manual. https://www.gnu.org/software/stow/manual/. Accessed: 2026-06-06.

[28] Frogmouth Developers. Frogmouth. https://github.com/Textualize/frogmouth.

[29] Alexander J Gaidis, Vaggelis Atlidakis, and Vasileios P Kemerlis. SysXCHG: Refining privilege with adaptive system call filters. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*, pages 1964–1978, 2023.

[30] Holger Gantikow, Steffen Walter, and Christoph Reich. Rootless containers with podman for hpc. In *High Performance Computing: ISC High Performance 2020 International Workshops, Frankfurt, Germany, June 21–25, 2020, Revised Selected Papers*, page 343–354, Berlin, Heidelberg, 2020. Springer-Verlag.

[31] Git Project. Git - git-clean Documentation. https://git-scm.com/docs/git-clean. Accessed: 2026-06-06.

[32] GNU patch contributors. patch(1) — Linux Manual Page. https://man7.org/linux/man-pages/man1/patch.1.html. Accessed: 2026-06-06.

[33] Andreas Haas, Andreas Rossberg, Derek L Schuff, Ben L Titzer, Michael Holman, Dan Gohman, Luke Wagner, Alon Zakai, and JF Bastien. Bringing the web up to speed with WebAssembly. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 185–200, 2017.

[34] Shivam Handa, Konstantinos Kallas, Nikos Vasilakis, and Martin C. Rinard. An order-aware dataflow model for parallel Unix pipelines. *Proc. ACM Program. Lang.*, 5(ICFP), August 2021.

[35] IAspireToBeGladOS. How to compress files larger than a certain size in a directory? https://stackoverflow.com/questions/42097461/how-to-compress-files-larger-than-a-certain-size-in-a-directory, 2017. Accessed: 2025-12-04.

[36] Sotiris Ioannidis, Steven M. Bellovin, and Jonathan M. Smith. Sub-operating systems: A new approach to application security. In *Proceedings of the 10th Workshop on ACM SIGOPS European Workshop*, EW 10, pages 108–115, New York, NY, USA, 2002. ACM.

[37] john. How can I recursively find all files in current and subfolders based on wildcard matching? https://stackoverflow.com/questions/5905054/how-can-i-recursively-find-all-files-in-current-and-subfolders-based-on-wildcard, 2011. Accessed: 2025-12-04.

[38] Konstantinos Kallas, Tammam Mustafa, Jan Bielak, Dimitris Karnikis, Thurston H.Y. Dang, Michael Greenberg, and Nikos Vasilakis. Practically correct, Just-in-Time shell script parallelization. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 769–785, Carlsbad, CA, July 2022. USENIX Association.

[39] Poul-Henning Kamp and Robert NM Watson. Jails: Confining the omnipotent root. In *Proceedings of the 2nd International SANE Conference*, volume 43, page 116, 2000.

[40] KCD. Regex with grep. https://stackoverflow.com/questions/66579161/regex-with-grep, 2021. Accessed: 2025-12-04.

[41] Kenneth Ward Church. Unix for Poets. https://web.stanford.edu/class/cs124/kwc-unix-for-poets.pdf, 1994.

[42] The Linux Kernel. Seccomp bpf (secure computing with filters). https://www.kernel.org/doc/html/latest/userspace-api/seccomp_filter.html, 2023.

[43] Taesoo Kim and Nickolai Zeldovich. Practical and effective sandboxing for non-root users. In *USENIX Annual Technical Conference (ATC)*, pages 139–144, 2013.

[44] Evangelos Lamprou, Ethan Williams, Georgios Kaoukis, Zhuoxuan Zhang, Michael Greenberg, Konstantinos Kallas, Lukas Lazarek, and Nikos Vasilakis. The Koala benchmarks for the shell: Characterization and implications. In *USENIX Annual Technical Conference (ATC)*, pages 449–64, Boston, MA, July 2025. USENIX Association.

[45] Lukas Lazarek, Seong-Heon Jung, Evangelos Lamprou, Zekai Li, Anirudh Narsipur, Eric Zhao, Michael Greenberg, Konstantinos Kallas, Konstantinos Mamouras, and Nikos Vasilakis. From ahead-of- to just-in-time and back again: Static analysis for unix shell programs. In *2025 Workshop on Hot Topics in Operating Systems*, HotOS '25, page 88–95, New York, NY, USA, 2025. Association for Computing Machinery.

[46] R. Levin, E. Cohen, W. Corwin, F. Pollack, and W. Wulf. Policy/mechanism separation in hydra. In *Proceedings of the Fifth ACM Symposium on Operating Systems Principles*, SOSP '75, pages 132–140, New York, NY, USA, 1975. ACM.

[47] H. M. Levy. *Capability Based Computer Systems*. Digital Press, 1984.

[48] Georgios Liargkovas, Di Jin, Tianyu (Ezri) Zhu, Dan Liu, A. Bolun Thompson, Anirudh Narsipur, Seong-Heon Jung, Siddhartha Prasad, Diomidis Spinellis, Michael Greenberg, Konstantinos Kallas, and Nikos Vasilakis. hS: Speculative script reordering at subprocess granularity. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. USENIX Association, 2026.

[49] Xin Lin, Lingguang Lei, Yuewu Wang, Jiwu Jing, Kun Sun, and Quan Zhou. A measurement study on Linux container security: Attacks and countermeasures. In *Proceedings of the 34th Annual Computer Security Applications Conference*, ACSAC '18, page 418–429, New York, NY, USA, 2018. Association for Computing Machinery.

[50] Sergio Maffeis, John C Mitchell, and Ankur Taly. Object capabilities and isolation of untrusted web applications. In *2010 IEEE Symposium on Security and Privacy*, pages 125–140. IEEE, 2010.

[51] Jonas Magazinius, Daniel Hedin, and Andrei Sabelfeld. Architectures for Inlining Security Monitors in Web applications. In *International Symposium on Engineering*

Secure Software and Systems (ESSoS), pages 141–160, 2014.

[52] Vahid Majdinasab, Michael Joshua Bishop, Shawn Rasheed, Arghavan Moradidakhel, Amjed Tahir, and Foutse Khomh. Assessing the security of github copilot's generated code - a targeted replication study. In *IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 435–444, 2024.

[53] Jeanna Neefe Matthews, Wenjin Hu, Madhujith Hapuarachchi, Todd Deshane, Demetrios Dimatos, Gary Hamilton, Michael McCabe, and James Owens. Quantifying the performance isolation properties of virtualization systems. In *Proceedings of the 2007 Workshop on Experimental Computer Science*, ExpCS '07, page 6–es, New York, NY, USA, 2007. Association for Computing Machinery.

[54] Dirk Merkel. Docker: lightweight linux containers for consistent development and deployment. *Linux Journal*, 2014(239):2, 2014.

[55] Leo A Meyerovich and Benjamin Livshits. Conscript: Specifying and enforcing fine-grained security policies for JavaScript in the browser. In *2010 IEEE Symposium on Security and Privacy*, pages 481–496. IEEE, 2010.

[56] James Mickens. Pivot: Fast, synchronous mashup isolation using generator chains. In *2014 IEEE Symposium on Security and Privacy*, pages 261–275. IEEE, 2014.

[57] Mark S Miller, Mike Samuel, Ben Laurie, Ihab Awad, and Mike Stay. Caja: Safe active content in sanitized JavaScript. *Google white paper*, 2009.

[58] Mark S Miller, Tom Van Cutsem, and Bill Tulloh. Distributed electronic rights in javascript. In *European Symposium on Programming*, pages 1–20. Springer, 2013.

[59] Mark S Miller, Ka-Ping Yee, Jonathan Shapiro, et al. Capability myths demolished. Technical report, Technical Report SRL2003-02, Johns Hopkins University Systems Research Laboratory, 2003. http://www. erights. org/elib/capability/duals, 2003.

[60] Mark Samuel Miller. *Robust Composition: Towards a Unified Approach to Access Control and Concurrency Control*. PhD thesis, Baltimore, MD, USA, 2006. AAI3245526.

[61] Scott Moore, Christos Dimoulas, Dan King, and Stephen Chong. Shill: a secure shell scripting language. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, OSDI'14, page 183–199, USA, 2014. USENIX Association.

[62] Antonio SJ Musumeci. mergerfs: a featureful union filesystem. https://github.com/trapexit/mergerfs, 2012–2025. Accessed: 2025-01-14.

[63] Shravan Narayan, Craig Disselkoen, Tal Garfinkel, Nathan Froyd, Eric Rahm, Sorin Lerner, Hovav Shacham, and Deian Stefan. Retrofitting fine grain isolation in the Firefox renderer. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 699–716, 2020.

[64] netgo software GmbH. LinOTP. https://github.com/LinOTP/LinOTP.

[65] Edmund B. Nightingale, Peter M. Chen, and Jason Flinn. Speculative execution in a distributed file system. *ACM SIGOPS Operating Systems Review*, 39(5):191–205, 2005.

[66] Okteto. Okteto - Develop your applications directly in your Kubernetes Cluster. https://github.com/okteto/okteto, 2025. Accessed: 2025-12-04.

[67] Senthilkumar Palani. How to use `-dry-run` flag in linux commands: Avoid mistakes before they happen. https://ostechnix.com/linux-dry-run-flag-guide-beginners/, February 2026. Accessed: 2026-06-09.

[68] Simon Peter, Jialin Li, Irene Zhang, Dan R. K. Ports, Doug Woos, Arvind Krishnamurthy, Thomas Anderson, and Timothy Roscoe. Arrakis: The operating system is the control plane. *ACM Trans. Comput. Syst.*, 33(4):11:1–11:30, November 2015.

[69] NSA Peter Loscocco. Integrating flexible support for security policies into the Linux operating system. In *Proceedings of the FREENIX Track:... USENIX Annual Technical Conference*, page 29. The Association, 2001.

[70] David Quigley, Josef Sipek, Charles P Wright, and Erez Zadok. Unionfs: User-and community-oriented development of a unification filesystem. In *Proceedings of the 2006 Linux Symposium*, volume 2, pages 349–362, 2006.

[71] Raghavan, Deepti and Fouladi, Sadjad and Levis, Philip and Zaharia, Matei. POSH: a data-aware shell. In *Proceedings of the 2020 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC'20, USA, 2020. USENIX Association.

[72] RIAEvangelist. node-ipc: Inter-process communication for node.js. https://github.com/RIAEvangelist/node-ipc, 2025. Accessed: 2025-01-02.

[73] Sam Ritchie. UV Metrics: Metric reporting and experiment management for ml workflows., 2020.

[74] RsyncProject. Issue #582: Rsync GitHub Issue. https://github.com/RsyncProject/rsync/issues/582. Accessed: 2026-05-27.

[75] RsyncProject. rsync(1) Manpage. https://rsync.samba.org/ftp/rsync/rsync.1.html. Accessed: 2026-06-06.

[76] J. M. Rushby. Design and verification of secure systems. In *Proceedings of the Eighth ACM Symposium on Operating Systems Principles*, SOSP '81, pages 12–21, New York, NY, USA, 1981. ACM.

[77] Mickaël Salaün. Landlock lsm: toward unprivileged sandboxing. *Linux Security Summit*, 2017.

[78] Jerome H Saltzer. Protection and the control of information sharing in multics. *Communications of the ACM*, 17(7):388–402, 1974.

[79] Frank Schmuck and Jim Wylie. Experience with transactions in quicksilver. In *Proceedings of the thirteenth ACM symposium on Operating systems principles*, pages 239–253, 1991.

[80] Russell Sears and Eric Brewer. Stasis: Flexible transactional storage. In *Proceedings of the 7th symposium on Operating systems design and implementation*, pages 29–44, 2006.

[81] David Sehr, Robert Muth, Cliff Biffle, Victor Khimenko, Egor Pasko, Karl Schimpf, Bennet Yee, and Brad Chen. Adapting software fault isolation to contemporary CPU architectures. In *19th USENIX Security Symposium (USENIX Security 10)*, 2010.

[82] Natalie Shapira, Chris Wendler, Avery Yen, Gabriele Sarti, Koyena Pal, Olivia Floody, Adam Belfki, Alex Loftus, Aditya Ratan Jannali, Nikhil Prakash, Jasmine Cui, Giordano Rogers, Jannik Brinkmann, Can Rager, Amir Zur, Michael Ripa, Aruna Sankaranarayanan, David Atkinson, Rohit Gandikota, Jaden Fiotto-Kaufman, EunJeong Hwang, Hadas Orgad, P Sam Sahil, Negev Taglicht, Tomer Shabtay, Atai Ambus, Nitay Alon, Shiri Oron, Ayelet Gordon-Tapiero, Yotam Kaplan, Vered Shwartz, Tamar Rott Shaham, Christoph Riedl, Reuth Mirsky, Maarten Sap, David Manheim, Tomer Ullman, and David Bau. Agents of chaos. https://arxiv.org/abs/2602.20021, 2026.

[83] Jonathan S Shapiro, Jonathan M Smith, and David J Farber. *EROS: a fast capability system*, volume 33. ACM, 1999.

[84] Zhiming Shen, Zhen Sun, Gur-Eyal Sela, Eugene Bagdasaryan, Christina Delimitrou, Robbert Van Renesse, and Hakim Weatherspoon. X-containers: Breaking down barriers to improve performance and isolation of cloud-native containers. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '19, page 121–135, New York, NY, USA, 2019. Association for Computing Machinery.

[85] Dimitrios Skarlatos, Qingrong Chen, Jianyan Chen, Tianyin Xu, and Josep Torrellas. Draco: Architectural and operating system support for system call security. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 42–57. IEEE, 2020.

[86] Snyk. Snyk vulnerability: Snyk-js-nodeipc-2426370. https://security.snyk.io/vuln/SNYK-JS-NODEIPC-2426370, 2025. Accessed: 2025-01-02.

[87] Stephen Soltesz, Herbert Potzl, Marc E. Fiuczynski, Andy Bavier, and Larry Peterson. Container-based operating system virtualization: a scalable, high-performance alternative to hypervisors. In *European Conference on Computer Systems (EuroSys)*, EuroSys '07, page 275–287, New York, NY, USA, 2007. Association for Computing Machinery.

[88] Richard P Spillane, Sachin Gaikwad, Manjunath Chinni, Erez Zadok, and Charles P Wright. Enabling transactional file access via lightweight kernel extensions. In *FAST*, volume 9, pages 29–42, 2009.

[89] Patrick Stevens. In praise of `-dry-run`. https://www.gresearch.com/news/in-praise-of-dry-run/, May 2021. Accessed: 2026-06-09.

[90] Teekin. Unix shell script: Update timestamp on all subdirectories and sub-files, including those with spaces. https://stackoverflow.com/questions/3718645/unix-shell-script-update-timestamp-on-all-sub-directories-and-sub-files-includ, 2010. Accessed: 2025-12-04.

[91] Jeff Terrace, Stephen R Beard, and Naga Praveen Kumar Katta. JavaScript in JavaScript (js. js): sandboxing third-party scripts. In *Presented as part of the 3rd USENIX Conference on Web Application Development (WebApps 12)*, pages 95–100, 2012.

[92] The try authors and contributors. Try: Controlling side effects with semisolates. https://github.com/binpash/try, 2025. Accessed: 2025-09-25.

[93] Lillian Tsai and Eugene Bagdasarian. Contextual agent security: A policy for every purpose. In *Proceedings of the 2025 Workshop on Hot Topics in Operating Systems*, HotOS '25, page 8–17, New York, NY, USA, 2025. Association for Computing Machinery.

[94] Tsaliki, Eleftheria and Spinellis, Diomedes. The Real Numbers for Athens Buses. https://insidestory.gr/article/noymera-leoforeia-athinas, 2020.

[95] Unix Game. The Unix Game - 50 Challenges to Master the Command Line. https://unixgame.io/unix50, 2024. Accessed: 2025-10-19.

[96] util-linux contributors. rename(1) — Linux Manual Page. https://man7.org/linux/man-pages/man1/rename.1.html. Accessed: 2026-06-06.

[97] Robert Wahbe, Steven Lucco, Thomas E Anderson, and Susan L Graham. Efficient software-based fault isolation. In *Proceedings of the fourteenth ACM symposium on Operating systems principles*, pages 203–216, 1993.

[98] Chris Wright, Crispin Cowan, James Morris, Stephen Smalley, and Greg Kroah-Hartman. Linux security module framework. In *Ottawa Linux Symposium*, volume 8032, pages 6–16, 2002.

[99] Yizheng Xie, Evangelos Lamprou, Jerry Xia, and Nikos Vasilakis. Incr: Faster re-execution via bolt-on incrementalization. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. USENIX Association, 2026.

[100] Wenhui Zhang, Peng Liu, and Trent Jaeger. Analyzing the overhead of file protection by Linux security modules. In *Proceedings of the 2021 ACM Asia Conference on Computer and Communications Security*, pages 393–406, 2021.

A LLM-Generated Scripts

This appendix lists the exact prompts used to generate the LLM-based benchmarks in §5.1, along with the scripts returned by ChatGPT.

`crawl`: Find & Touch

PROMPT

Write a simple Bash script that searches a directory (for example ./data) and runs a command on every file it finds. Use find with -exec and include comments explaining what the script does. The command should just touch each file.

GENERATED SCRIPT

```
directory="./data"
find "$directory" -type f -exec touch "{}" \;
```

`fresh`: Compress Files Larger Than 1 KB

PROMPT

Write a Bash script that finds all files larger than 1 KB in a directory (e.g., ./data) and compresses them using gzip. Use find with the -size filter and an -exec command. Include comments.

GENERATED SCRIPT

```
directory="./data"
find "$directory" -type f -size +1k \
  -exec sh -c 'gzip "{}"' \;
```

`archive`: Find All .txt Files

PROMPT

Give me a command to find all .txt files in the current directory and its subdirectories.

GENERATED SCRIPT

```
find . -name "*.txt"
```

`logs`: Extract Error Lines From Log

PROMPT

Give me a command to extract only ERROR, WARNING, or CRITICAL lines from a log file and save them to a new file. Make sure the command doesn't fail even if nothing is matched.

GENERATED SCRIPT

```
grep -E "ERROR|WARNING|CRITICAL" file.log \
  > filtered_errors.log || true
```

`order`: Sort a File

PROMPT

Give me a command to sort the contents of book.txt and save the result to another file.

GENERATED SCRIPT

```
sort book.txt > sortedBook.txt
```

B Testimonials

This appendix includes testimonials from users who have tried `try` and shared their feedback on social media platforms. They offer a snapshot of real-world experiences and community reception of the tool. The feedback was primarily gathered from platforms such as personal blogs and Hacker News.

Looks nifty—basically it's the "let me try that in a container first" except on your live system with no setup to get it going.

— **@mikepurvis**, *Hacker News*

This is a much needed software.

— **@johnsbullings**, *Hacker News*

Very interesting! This feels like something that should have always existed.

— **@Benjamin Oakes**, *benjaminoakes.com*

This is cool I probably will use this.

— **@jagtstronaut**, *Hacker News*

I love that their demo is to finally give me a dry run command for pip.

— **@gdevenyi**, *Hacker News*

The script is mounting an overlay filesystem for each root directory on the current system, then looking at what has been added as a new layer... Pure genius.

— **@MichaelMoser123**, *Hacker News*

I can see this being incredibly useful.

— **river**, *lobste.rs*

`try` is clever.

— **Jim Fowler**, *x.com*

I look forward to it helping me prevent programs polluting my laptop.

— **bfiedlera**, *lobste.rs*

This tool can bring a level of control that is typically only available in databases and VCSs, and make it a commodity.

— **@klabb3**, *Hacker News*

C Software Artifact

Abstract

The software artifact accompanying this paper consists of the `try` subsystem, its documentation, a test suite, and the scripts used to reproduce the evaluation in §5. It supports three goals: inspecting the implementation and its accompanying materials, exercising the system through its test suite, and reproducing the paper's main evaluation results, including the summary across the five use cases and the microbenchmark breakdown.

Scope

The artifact covers the claims made by the paper's implementation and evaluation sections. In particular, it includes: (1) the `try` subsystem and the semisolate abstraction described in §3 and §4; (2) the five use cases and supporting benchmarks from §2 and §5; and (3) the automation used to reproduce all key results in §5.

Contents

The artifact contains the following components:

- the `try` subsystem itself;
- documentation, including a top-level README, a manual page, and a short tutorial;
- a regression test suite covering `try`'s core functionality;
- the five evaluation workloads and their supporting utilities; and
- scripts for running the evaluation and generating all key result artifacts.

Hosting

The artifact is publicly available both as a GitHub repository and as a Zenodo archival snapshot. The GitHub version is hosted at `https://github.com/binpash/try/tree/osdi26-ae`, and the archival version is hosted at `https://zenodo.org/records/19444649`.

ARTIFACT EVALUATED
USENIX
AVAILABLE

SBB: Eliminating Centralized Bottlenecks in Userspace Network Runtime

Kang Hu[†,‡] Shuqi Dong[†] Chuandong Li[†,‡] Ran Yi[†] Zonghao Zhang[†,‡] Yiming Yao[†]
Bo An[‡] Jie Zhang[†,‡] Xiaolin Wang[†,‡] Yingwei Luo[†,‡] Zhenlin Wang[*] Diyu Zhou[†]
[†] *Peking University*, [‡] *Zhongguancun Laboratory*, [*] *Michigan Technological University*

Abstract

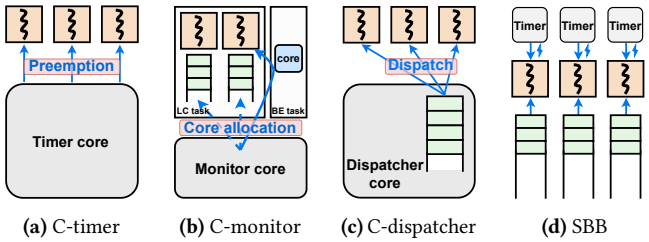
To achieve high throughput, low latency, and high CPU efficiency, userspace network runtimes must perform three types of scheduling: 1) *request preemption* to minimize tail latency, 2) *CPU allocation* among services to avoid wasting CPU cycles when the load is low, and 3) *request load balancing* across worker cores to achieve work conservation. However, prior designs rely on centralized components, which inevitably become scalability bottlenecks as the number of CPU workers increases, limiting system performance scaling.

This paper presents SBB, a purely decentralized userspace network runtime that simultaneously delivers high performance, high CPU efficiency, and high scalability by advancing in both system mechanism and scheduling policy. For system mechanism, SBB leverages the emerging User Interrupt mechanism in a novel way, delivering two types of device interrupts to the userspace runtime: 1) user-level timer interrupts for request preemption, and 2) user-level NIC interrupts for packet arrival to perform CPU allocation. For scheduling policy, SBB introduces a two-level algorithm, which marries flow migration (for persistent imbalance) with task stealing (for temporary imbalance), challenging the conventional wisdom that centralized load balancing outperforms decentralized approaches. Evaluation shows that SBB achieves 1.7× to 5.2× higher throughput when scaling to 48 cores, while meeting the same tail latency target.

1 Introduction

Modern high-speed NICs and links have enabled datacenters to enjoy high bandwidth, on the scale of hundreds of Gbps [1, 3, 8, 12]. On the other hand, CPU processing speeds in datacenters struggle to keep pace with the performance of NIC hardware [66, 82]. It is thus foreseeable that a single CPU core, or even a dozen cores, will be insufficient to handle the massive volume of requests in datacenters in a timely manner [52, 58, 69]. As a result, network runtimes must achieve great multicore scalability.

As tail latencies enter the microsecond realm, modern operating systems such as Linux fail to meet high-performance networking demands, which has motivated researchers to build userspace network runtimes that bypass the kernel and leverage fine-grained scheduling to achieve μs-scale request processing [16, 24, 52, 55, 59, 83, 87]. Besides high throughput and low latency, network runtimes also need to support high CPU efficiency, as even a small efficiency loss can significantly increase cost at scale [33, 67]. To meet these demands, network runtimes must perform three types of scheduling: 1) *request preemption* to prevent long requests from causing head-of-line blocking of short ones [43, 54], 2) *CPU allocation* among co-located applications to avoid wasting CPU cycles when the load is low [60, 74], and 3) *request load balancing* across worker cores to achieve work conservation [71, 76].



(a) C-timer **(b)** C-monitor **(c)** C-dispatcher **(d)** SBB

Figure 1: Comparison of userspace network runtime architectures. The centralized timer, monitor and dispatcher will all become bottlenecks as the number of workers increases (§2.3.2). SBB removes all centralized components and achieves great scalability.

Unfortunately, to perform the scheduling, existing works rely on standalone centralized entities, leading to scalability bottlenecks. For request preemption, a common design [43, 51, 54, 63] is to use a centralized timer to periodically send some specific signals to all workers (Figure 1a). For CPU allocation, many works [38, 74] deploy a centralized monitor that polls each worker's NIC queue, detecting packet backlog and making CPU core reallocation decisions (Figure 1b). For request load balancing, a common method [51, 53, 54, 74] is to rely on a centralized dispatcher to distribute requests to worker cores (Figure 1c). When scaling to more workers, these centralized entities will become the bottleneck (§2.3.2).

While this scalability limitation is widely recognized [43, 70], existing works fall short due to i) unvalidated assumptions and the lack of ii) scalable system mechanisms and iii) effective decentralized scheduling policies. To wit, conven-

tional wisdom is that one can scale centralized entities with multiple cores [54, 85], each handling a subset of requests or worker cores. Our measurement (Figure 3) challenges it, showing that using more cores achieves worse load balancing, and does not help with request preemption and CPU allocation. In addition, existing system mechanisms lack a μs-scale, decentralized approach to notify worker cores of scheduling events, like time quanta expiration or new packet arrivals, making a centralized timer or monitor the only option. Finally, for scheduling policies, centralized load balancing is chosen as it better approximates a shared FCFS queue, the ideal policy proven by queuing theory [81], but it overlooks the constraint of scalability.

This paper presents SBB[1], a highly scalable μs-scale network runtime that eliminates all centralized scheduling entities with even better scheduling quality than prior designs. SBB's design advances in both system mechanism and scheduling policy.

For the **system mechanism**, we find that centralized request preemption and CPU allocation stem from a lack of different types of hardware support. The former lacks an efficient timer for detecting and preempting long requests. For the latter, interestingly, the root cause is the use of polling to handle incoming packets. Polling prevents a timely detection of packet arrivals when multiple applications are co-located. Thus, a centralized entity is a must to monitor incoming packets and decide if each network application is runnable to perform CPU allocation.

To provide system mechanisms for scalable scheduling, SBB leverages the emerging User Interrupt (UINTR) feature [10, 17]. With UINTR, SBB delivers two types of device interrupts to userspace, a capability not documented in Intel hardware manuals [9]. SBB delivers per-core timer interrupts to the network runtime, thereby avoiding core count as a scalability bottleneck for request preemption. In addition, SBB delivers NIC interrupts to userspace, eliminating a centralized entity to detect packet arrivals, thereby enabling scalable CPU allocation.

For the **scheduling policy**, our main observation is that existing decentralized load balancing policies fail to consider both causes of imbalance, explaining their poor performance. To wit, existing designs [38, 71, 76] use a combination of NIC steering hardware (e.g., RSS [15]) and task stealing. The former maps a network flow to a worker core, while the latter enables idle cores to steal requests from busy ones. However, the flow-to-core steering in NIC is susceptible to both temporary (*e.g.*, a burst of requests in a single flow) and persistent load imbalance (*e.g.*, some cores are assigned too many flows). Task stealing is only effective for temporary imbalance, causing excessive cache coherence traffic for persistent imbalance, and hence, poor performance.

[1]The national railway company of Switzerland. While both countries enjoy efficient rail networks, Switzerland's is known for being highly decentralized, avoiding centralized busy hubs such as Shinjuku in Tokyo.

Thus, SBB introduces a two-level policy that marries task stealing with flow migration into μs-scale network runtime. For temporary imbalance, SBB uses an enhanced task stealing technique to balance at packet-level, while to handle persistent imbalance, SBB leverages flow migration, which programs the NIC to migrate flows away from overloaded cores, thereby addressing imbalance at flow-level. We show, with both simulation (as a platform-independent setup) and real-world experiments, that the scheduling quality of this two-level policy closely matches that of a centralized one, proving that centralized schemes are not inherently better. With it, SBB achieves scalable request load balancing.

We integrate SBB with DPDK [7] to benefit from its wide deployment. We evaluate SBB with synthetic workloads and real-world applications, showing that SBB scales well and achieves an SLO-compliant throughput increase of 1.7× to 5.2× over prior work.

In summary, this paper makes the following contributions:

- **Analysis.** We analyze and reveal the root causes of scheduling scalability bottlenecks in existing μs-scale network runtimes.
- **Mechanisms and Policies.** For scalable scheduling, we propose 1) the use of UINTR to deliver userspace timer and NIC interrupts for request preemption and CPU allocation, and 2) a two-level request load balancing policy to mitigate comprehensive imbalance problems.
- **SBB.** We implement our mechanisms and policies in SBB[2], a high-performance userspace network runtime that supports all three types of scheduling in a decentralized and scalable architecture.

2 Background and Motivation

This section identifies the three key scheduling types in userspace network runtimes (§2.1), presents an introduction to User Interrupt (§2.2), and introduces how prior work uses centralized entities to perform the scheduling (§2.3).

2.1 Scheduling in Network Runtimes

Modern network runtimes must meet three requirements: a) high throughput, b) low median and tail latency, and c) high CPU efficiency. High throughput and low median latency exploit the performance of modern NICs. Tail latency matters because network requests often fan out across multiple components, and thus the end-to-end latency is determined by the slowest response [33, 35]. For the last requirement, even marginal CPU efficiency gains can yield substantial cost savings in datacenters [67, 73, 74, 79].

To meet these requirements, userspace network runtimes perform three types of scheduling: 1) request preemption, 2) CPU allocation, and 3) request load balancing.

Request preemption happens during request processing to avoid head-of-line blocking, thereby reducing tail latency. Head-of-line blocking occurs because heavy-tailed network

[2]The artifact of SBB is at https://github.com/TELOS-syslab/SBB-Artifact.

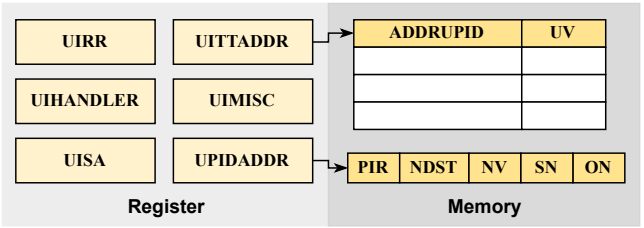


Figure 2: Key hardware components of User Interrupt.

services involve skewed workloads, where a small portion of requests take much longer to process than others [13, 14, 78]. To minimize tail latency, a network runtime needs to preempt long requests to prioritize short ones [81]. Furthermore, since network requests need to be processed at microsecond scales, the overhead of preemption and task switching must be minimal, making μs-scale preemption an essential feature for userspace network runtimes [54].

CPU allocation is necessary to maximize CPU efficiency, *i.e.*, the fraction of CPU cycles spent performing useful work. It is common practice to co-locate multiple applications on a single server, including latency-critical (LC) and best-effort (BE) applications [21, 50, 84]. Thus, the runtime needs to allocate an appropriate share of CPU time to each application and dynamically adjust allocations as load changes: it must reallocate idle cores from LC applications to BE tasks, and be able to promptly reassign cores back to LC applications to handle load bursts [60].

Request load balancing is performed to assign each incoming request to a processing core. The goal is to ensure that no cores are idle while requests are pending. It tries to mitigate the imbalance issue between workers: some workers are too overloaded to handle assigned requests, while some idle workers have no assigned requests.

2.2 User Interrupt

User Interrupt (UINTR) [9, 10, 17] is a new hardware feature introduced with Intel Sapphire Rapids CPUs, which allows processes to send and receive interrupts directly in user space without trapping into the kernel. By bypassing the kernel, UINTR avoids overheads such as switching address spaces or privilege levels, thereby offering high performance.

Main hardware resources. Figure 2 illustrates the key hardware resources for User Interrupt, which consist of registers and memory structures. Each core is equipped with six model-specific registers (MSRs): 1) *UIRR*: a bitmap in which each set bit indicates a pending interrupt, 2) *UIHANDLER*: stores the linear address of the user-interrupt handler, 3) *UISA*: controls stack pointer adjustment before delivering a user interrupt, 4) *UITTADDR*: holds the memory address of the User Interrupt Target Table (UITT), 5) *UIMISC*: contains UITTSZ (the highest index of a valid entry in the UITT) and UINV (the interrupt vector to be delivered to user space), and 6) *UPIDADDR*: stores the memory address of the User Posted Interrupt Descriptor (UPID).

In terms of memory resources, each thread has a UPID and a UITT. *UITT* is a table of entries, each storing UV (the user interrupt vector) and ADDRUPID (the address of the target UPID). *UPID* is a memory region storing the interrupt state (*e.g.*, the pending flag (*PIR*) and vector (*NV*)), so that both hardware and the OS can manage user-level interrupts.

User Interrupt sending. Intel provides a new instruction, *SENDUIPI*, to send UIPIs, which takes an index into the UITT as its parameter. Upon execution, the hardware first locates the entry in the UITT using the provided index. It then retrieves the ADDRUPID field to locate the target UPID, updates the PIR field according to the UV to post the interrupt, and finally delivers the interrupt to the target core.

User Interrupt handling. When a UINTR is signaled to a core, the receiving CPU proceeds through three sequential stages: 1) Identification: The CPU first checks whether the incoming interrupt vector matches UINV. If not, the interrupt is processed as a conventional kernel-managed interrupt; otherwise, it proceeds to the next stage. 2) Notification Processing: Once identified as a UINTR, the CPU copies the bitmap of pending interrupt requests from the PIR field of the associated UPID into UIRR, clears the PIR field, and proceeds to deliver each interrupt corresponding to a set bit in the UIRR. 3) Delivery: When the CPU subsequently enters user mode, and the UIRR indicates pending interrupts, the CPU transfers control to the user-level interrupt handler whose address is stored in the UIHANDLER register.

	Request load balancing	Request preemption	CPU allocation
IX [24], ZygOS [76], Arrakis [75]	✓	✗	✗
Shinjuku [54], Concord [51]	✓̸	✓̸	✗
Perséphone [34]	✓̸	✗	✗
TQ [70]	✓̸	✓	✗
Libpreemptible * [63]	✓̸	✓̸	✗
Aspen-KB * [43]	✓	✓̸	✗
Shenango [74]	✓̸	✗	✓̸
Caladan [38]	✓	✗	✓̸
Junction * [39], Vessel * [65]	✓	✗	✓̸
Skyloft * [53]	✓̸	✓	✓̸
SBB*	✓	✓	✓

Table 1: A comparison of various userspace network runtimes. SBB is the only one that supports all three types of scheduling without any centralized entities. ✗: Not supported. ✓̸: Supported with a centralized entity. ✓: Supported with decentralized design. * means this system leverages User Interrupt.

2.3 Related Work

The importance of μs-scale networking has motivated a rich body of prior work, summarized in Table 1. We draw two conclusions from the table: 1) Most prior work does not support all three types of scheduling, failing to be a fully functional network runtime; and 2) all prior work uses centralized entities for at least one scheduling type, hindering the whole

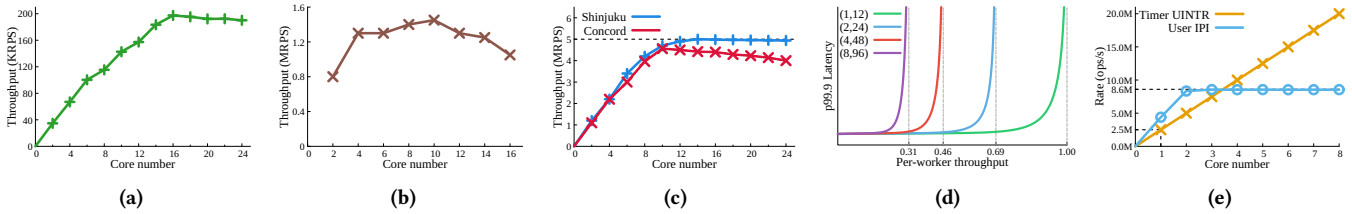


Figure 3: The centralized entities incur poor scalability. (a)–(c) show the scalability bottlenecks of centralized entities individually for request preemption, CPU allocation, and request load balancing. (d) shows that using multiple centralized entities for request load balancing causes imbalance again, in which (n, m) means n dispatchers and m workers. (e) shows the scalability problem of using user IPIs.

system's scalability.

We next introduce how prior work performs the three types of scheduling (§2.3.1), and analyze the limits of centralized entities (§2.3.2) in detail.

2.3.1 How Prior Work Performs Scheduling

Request preemption is often performed by a centralized timer that sends preemption signals to workers when time quantum expires. Although the specific preemption mechanisms differ, they all follow a similar architecture: a dedicated centralized component manages the preemption of a group of workers, which are responsible for actual request processing. Existing works use advanced preemption mechanisms to minimize the overhead. For example, Shinjuku [54] uses virtualization-hardware-assisted IPIs, Concord [51] employs compiler instrumentation, while Libpreemptible [63] and Aspen-KB [43] utilize Userspace IPIs. Despite this, all these advanced mechanisms still rely on a centralized entity.

CPU allocation uniformly uses a centralized entity when present. Some prior systems give up supporting multi-application co-location (see Table 1)—an approach that is increasingly unacceptable given the growing demand for CPU efficiency in datacenters. Other systems, such as Shenango [74] and Caladan [38], introduce a centralized component to dynamically manage CPU core allocation. They employ a dedicated monitor that continuously monitors queue backlog on NICs and decides whether cores need to be reallocated to LC tasks to handle load bursts.

Request load balancing can be 1) centralized or 2) decentralized. The userspace network runtimes at an early stage usually adopt decentralized architecture, only relying on Receive Side Scaling (RSS) [15], a modern NIC hardware mechanism to steer incoming packets to cores. This is achieved by mapping each packet to a flow (based on, *e.g.*, the source and destination IP addresses and ports, and the protocol number), and then assigning each flow to a core.

However, the flow steering hardware suffers from load imbalance, as different cores may be allocated skewed flows. Recent works [38, 43, 76] alleviate the hardware drawbacks with task stealing, but suffer from performance issues [54, 71]. By contrast, more recent systems [51, 53, 54, 70, 74] choose to use a dedicated dispatcher thread to poll the single NIC queue, handle network protocol processing, and send the packet to an idle worker.

2.3.2 The Scalability Bottlenecks

Although the three types of centralized components, *i.e.*, timer, monitor, and dispatcher, could support each scheduling type to some extent, they unavoidably become the bottleneck of the whole system, as the number of workers increases. In order to analyze each type of centralized component in a targeted manner, we select representative systems and conduct focused tests in the corresponding scenarios, showing their impact on scalability.

Scalability bottleneck 1: centralized timer. We select Shinjuku [54] as a representative system that implements request preemption, which uses one centralized timer to record the processing time of each worker, and when the processing time exceeds the time quanta, sends an IPI to interrupt it, forcing it to switch to the next packet.

To investigate Shinjuku's scalability problem, we conduct a heavy-tailed workload test with an increasing number of workers. As shown in Figure 3a, when the number of workers exceeds a threshold, the throughput of the system stops increasing and even decreases. Further analysis reveals that the timer becomes the bottleneck once the number of workers exceeds 16: when many workers need preemption simultaneously, the timer cannot send IPIs to all of them during the configured preemption interval (*e.g.*, 5μs). Previous studies have also shown the centralized timer's scalability limitation when using many workers [43, 70].

Scalability bottleneck 2: centralized monitor. Caladan [38] is state-of-the-art network runtime that supports multi-application co-location, and acts as a basis for subsequent works [39, 65, 71]. It introduces a component called iokernel, which acts as a centralized monitor, checking whether any LC task thread has accumulated packets in its NIC queue every scheduling interval (typically 10μs). If so, it preempts a core from a BE task and reassigns it to the LC task. If the backlog remains, iokernel repeats the process in the next interval, continuing until the load burst subsides.

To evaluate iokernel's ability to reallocate cores, we modify Caladan's load generator following on-off network patterns, which are common in the real world [25, 26, 41]. The gen-

erator alternates between two phases every 1 ms: Phase 1 produces a specified load, and Phase 2 stops sending packets. Other testbed configurations are the same as §6.3.

We set the SLO to a p99.9 latency below 50μs and measure the maximum load Caladan could sustain without SLO violations for different worker counts. As shown in Figure 3b, Caladan exhibits poor scalability. The reason lies in the iokernel: as the load and number of cores increase, it fails to reallocate enough cores to LC tasks in a timely manner when packets arrive in bursts, which easily causes SLO violations.

Scalability bottleneck 3: centralized dispatcher. To evaluate the centralized dispatcher's ability to perform load balancing, we select Shinjuku and Concord [51] to exclusively run Fixed(1) workload (see Table 3), to exclude the need for request preemption and CPU allocation. Figure 3c shows that the overall throughput has an upper limit of 5MRPS. A straightforward calculation validates this issue: even though Shinjuku employs various optimizations (such as batching and simplifying protocol logic) to largely reduce the packet's processing overhead for dispatcher (200ns measured on our machine on average), since all the packets are processed by dispatcher, the maximum packet processing rate of the system is capped at $1/200$ ns $= 5$ MRPS. It is analogous to a river whose overall flow remains constrained by the narrowest upstream section, no matter how much the downstream channel is widened.

Multiple dispatchers do not help. The scalability bottleneck of centralized dispatchers is widely admitted [43, 54, 71], and a common assumption is that a viable solution is to deploy multiple dispatchers, each responsible for dispatching a subset of packets [54, 85].

The experiment results refute this belief. Figure 3d shows that multiple dispatchers suffer from load imbalance again. We constrain each dispatcher to manage only 12 workers in a group. As the number of groups increases, the throughput on average decreases notably.

User IPIs do not scale. Although UINTR is a relatively new technology, it has already been adopted in several systems, primarily for sending userspace inter-processor interrupts (UIPIs) [39, 43, 63, 65].

However, although UIPI provides higher performance, it has a limited sending rate regardless of the number of cores used, as shown in Figure 3e. Although we increase the number of senders that use dedicated cores sending user IPIs, the rate of interrupts still has an upper limit, which is attributed to one NUMA's constraint. This result means the IPI operations still cannot scale with the workers, even when adding more cores to serve as the timer or monitor.

3 Enabling Decentralized Network Runtime

Based on the following analysis and experiments, this section concludes the design goals (§3.1) and presents the key design ideas behind SBB, enabling it to eliminate centralized entities for all three types of scheduling without sacrificing performance benefits (§3.2 and §3.3).

3.1 Design Goals

SBB aims to build a purely decentralized userspace network runtime, identifying the following as design goals:

High Scalability. Since multi-core collaboration is essential in datacenters to ensure that CPU processing keeps pace with the rapid evolution of NIC hardware, SBB's foremost goal is to construct a network runtime with strong scalability.

High Performance. SBB's scalability across multiple cores must not come at the cost of performance with fewer cores. Regardless of the number of workers, SBB should achieve better performance than existing systems, supporting higher throughput and lower tail latency across various workloads.

High Efficiency. To improve CPU utilization and save energy, SBB must support the co-location of LC applications with other applications on the same cores, saving idle CPU cycles without decreasing LC applications' performance.

3.2 Decentralized Preemption and CPU Allocation

Through further analysis, we next discuss, for request preemption and CPU allocation, (1) why prior work adopts a centralized design and (2) why SBB can leverage User Interrupt to eliminate it.

Request Preemption. For request preemption, prior work forgoes a decentralized design due to a lack of hardware support. Conventional interrupts trap to the kernel, thus being unsuitable for kernel-bypass runtimes. While Dune [23] enables direct access to privileged hardware features, it does not include timer interrupts, which the OS uses to regain CPU control. Hence, the only viable solution is to use virtualization-assisted IPIs or user IPIs [43, 54, 63].

CPU Allocation. This demand is fundamentally caused by the inherent conflict between polling and multi-application co-location [47]. While polling offers lower overhead, it carries an unavoidable drawback: without interrupts, an application can learn about packet arrivals only when it is actively running on a core and polling. Once descheduled, the application has no means of receiving notification of new packets until the scheduler assigns it a core again, which typically takes millisecond intervals, unacceptable for microsecond-scale network requests [65]. Facing this challenge, prior work requires an entity to monitor and perform CPU allocation.

Our approach. Our observation is that UINTR can eliminate the centralized entity for both scheduling types. To enable request preemption, SBB delivers per-core timer interrupts to userspace (§4.3), without an upper limit on preemption rates (Figure 3e). For CPU allocation, SBB forgoes polling and delivers NIC UINTR, enabling the worker core to be aware of packet arrivals promptly without a centralized entity (§4.2).

Our use of UINTR departs from most prior work. As discussed in §2.3.2, prior work focuses on UIPI as a faster alternative to IPIs for centralized preemption. Rather, SBB delivers NIC and timer interrupts to userspace, enabling a decentral-

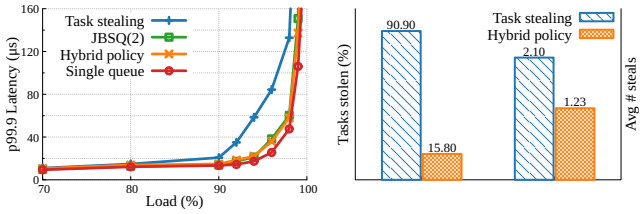


Figure 4: Simulation results of different load balancing policies.

ized design. A traditional concern about our design is that interrupts may be too expensive for μs-scale networking [22]. However, the breakdown experiments show that both NIC and timer UINTR are high-performance mechanisms (§6.5).

3.3 Scalable and Effective Request Load Balancing

Contrary to the conventional wisdom that centralized design performs better, we find that with proper scheduling policy, decentralized request load balancing achieves quality matching centralized schemes. We demonstrate this using a prior simulation system [71]. Keeping most setups unchanged, we add the simulation of NIC RSS mechanism and group the packets into flows.

We evaluate four designs. 1) **`Single queue`** represents the optimal case proven by queuing theory [81], where an idle worker always fetches requests from a shared queue, thus achieving perfect load balancing. However, this design is impractical on real systems, due to contention on the shared queue. 2) **`JBSQ(2)`** is the closest approximation [51] to `Single queue`, and thus the best centralized design. In this case, each worker owns a two-slot private queue, and a centralized entity dispatches requests from the shared queue to the shortest worker queue. 3) **`Task stealing`** is the prior best decentralized design, making idle workers steal requests from other workers. The stealing overhead is set to 200ns, following the prior work's configuration. 4) **`Hybrid policy`** is the two-level load balancing policy used by SBB.

Figure 4a shows that `task stealing` is still far from `JBSQ(2)`, and Figure 4b shows why. We record two metrics when the load is 95%: 1) **`Tasks stolen`** represents the percentage of completed tasks that are stolen at least once. 2) **`Average number of steals`** counts the average number of steal events among all the stolen tasks. It shows that task stealing performs badly due to too many stealing events, which cause cache misses. After collecting the information of those stolen packets, we are surprised to see that most packets belong to only a few flows, which means due to the randomness of RSS, some cores are assigned much fewer flows than other cores, making the former continue stealing packets in a flow from the latter.

Our approach. The analysis shows that the poor performance of a decentralized design is not inherent. Our main observation is that the flow-to-core mapping in NIC steering

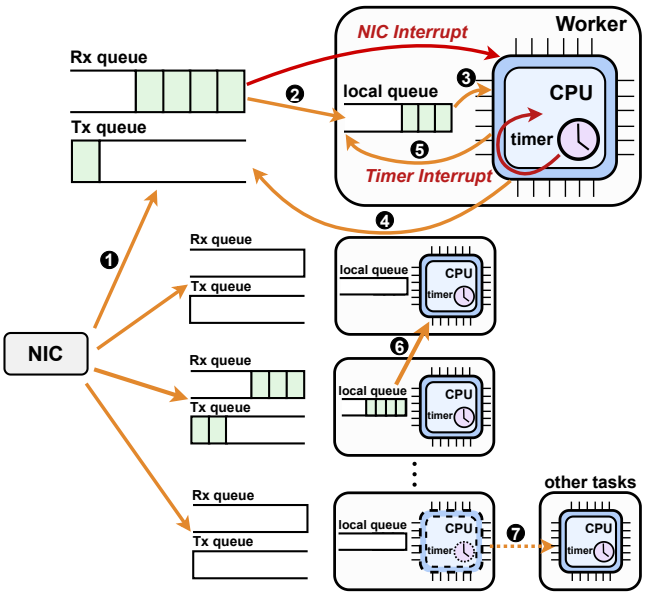


Figure 5: Overall architecture of SBB.

hardware causes both 1) temporary load imbalance, where a flow experiences a burst of requests, and 2) persistent load imbalance, where some cores are assigned too many flows. Task stealing enables a worker experiencing a sudden burst of incoming packets to offload packets to others, which is only suitable for the former, not the latter, where it suffers from continuous and excessive cache coherence overhead and thus underperforms.

Therefore, we propose a two-level decentralized load balancing design, marrying task stealing with flow migration to handle comprehensive load imbalance. The simulation shows that this hybrid policy largely reduces unnecessary stealing (Figure 4b) and achieves nearly the same performance as `JBSQ(2)` (Figure 4a).

4 Design

4.1 An Overview of SBB

Figure 5 illustrates the overall architecture of SBB. Instead of employing any centralized components, SBB builds a purely decentralized network runtime. Each worker thread is bound to a CPU core and maintains its own local queue and local timer. The NIC provides each worker thread with a dedicated set of Rx and Tx queues. As a result, from a global perspective, every component belongs exclusively to one worker thread—no shared components exist across multiple worker threads.

The workflow of SBB proceeds as follows: ❶ When a packet arrives at the NIC, it is assigned to one Rx queue via RSS (Receive Side Scaling) [15]. ❷ The Rx queue triggers a user-level NIC interrupt, which is handled by the worker thread in user space. ❸ The worker thread moves packets from its Rx queue into its local queue and then processes them. ❹ After processing a packet, the worker thread con-

structs a response and transmits it via its Tx queue. ❺ A local timer tracks the processing time of each packet, and triggers a timer interrupt if a packet's processing exceeds the predefined quanta. The worker's CPU then saves the packet's state, requeues it, and proceeds to the next packet. When a worker thread finds its Rx queue empty and has no packets to process, it ❻ performs task stealing or flow migration to balance the load; if still failing to obtain new packets, it ❼ yields the core to other co-located applications to increase CPU efficiency.

This purely decentralized architecture provides inherent advantages for scalability. All workers can directly send and receive packets through the NIC, enabling fully parallel dataplane processing without centralized dispatching. In the control plane, the removal of all centralized components eliminates the need for any central entity to issue control signals—including those for request preemption and CPU allocation. By eliminating centralized components (*i.e.*, timer, monitor and dispatcher) used by prior work (§2.3.2), SBB allows the system's overall throughput to scale nearly linearly with the number of workers.

However, this model also introduces key challenges: SBB still needs to perform three types of scheduling to provide high throughput, low latency, and high efficiency. The following sections describe how SBB provides each of these scheduling functions.

4.2 Timely CPU Allocation

Choosing interrupt over polling. As demonstrated in §2.3.2, polling has an inherent conflict with multi-application co-location, while Caladan's centralized monitor cannot solve this with great scalability.

Our observation is that interrupt is more suitable than polling for multi-application co-location. We build a prototype for test, which leverages DPDK's interrupt-driven packet reception [7]. The results indicate that while its tail latency is higher, the system shows promising scalability under the given conditions (Figure 3b). When a new packet arrives, the interrupt mechanism promptly wakes up the LC task even if it is not currently running on the core, enabling timely request processing without relying on external control components.

However, conventional interrupt mechanism involves kernel crossing, incurring higher overhead than polling [42, 62], which makes it unsuitable for userspace network runtimes. Therefore, we leverage a new hardware feature, User Interrupt (UINTR), to achieve timely CPU allocation. To the best of our knowledge, SBB is the first to adopt UINTR as a replacement for polling for NIC notification mechanisms.

Delivering device user interrupt. Although UINTR has been adopted in several recent works, most utilize it solely for UIPIs [19, 39, 43, 45, 63, 65]; no prior work has employed UINTR as a notification mechanism for NICs. This is because the current user-interrupt hardware is not designed to handle interrupts generated from devices, *e.g.*, NICs. As described in §2.2, the CPU can only proceed to the interrupt handler in the third step if two earlier conditions are met: 1) The interrupt vector from the NIC must match UINV, and 2) the PIR field must be non-zero.

Meeting the first condition is relatively straightforward: during initialization, the interrupt vector of each NIC queue can be obtained, and UINV can be set accordingly. However, satisfying the second condition is challenging because even if a bit in PIR is set initially, it will be cleared during the first UINTR handling cycle. To continuously receive NIC-generated user interrupts, the PIR must be rewritten after each interrupt is processed. The UINTR hardware does not provide a low-overhead mechanism for userspace programs to achieve this: either issuing a SENDUIPI to oneself, which would generate an IPI, or trapping into the kernel to modify PIR, thus forfeiting the performance benefit of kernel bypass.

To overcome this limitation at low cost, some systems have introduced software workarounds [53, 61]. We use the approach proposed by Aeolia [61]: during initialization, a new system call maps UPID into the address space of the user program. This enables the application to rewrite PIR after handling each NIC-generated user interrupt, thereby re-arming the next UINTR.

Reallocating cores via interrupt. We replace polling with UINTR as the NIC notification method, achieving a low-overhead mechanism that is suitable for CPU sharing. When a packet arrives at the NIC, the corresponding queue raises an interrupt to its assigned core. If the LC task is currently running on that core, the interrupt is delivered directly to the registered handler in user space, introducing minimal overhead compared to polling (see §6.5). If the LC task is not running, which is the case where polling cannot handle efficiently, the interrupt vector will not match UINV, causing the interrupt to be processed as a traditional kernel interrupt. During initialization, SBB registers a custom kernel handler for each NIC queue, which triggers when packets arrive while the associated LC task is inactive. This handler wakes up the LC task and invokes the scheduler, allowing the task to regain CPU resources and process the request promptly.

Compared to polling-based sharing approach, adopting UINTR fundamentally resolves the conflict between polling and CPU sharing. Furthermore, in contrast to Caladan's centralized and passive CPU scheduling, SBB enables each core to actively and promptly switch among multiple applications without relying on external components, demonstrating both high performance and strong scalability under load bursts.

Avoiding interrupt storm. One natural concern about interrupts is the interrupt storm, *i.e.*, packets arriving at high rates may trigger interrupts so frequently that interrupt handling consumes excessive CPU cycles and influences other packets' processing [48, 56].

SBB avoids this classic problem by leveraging NIC's auto-mask capability (introduced in §4.4), which is similar to in-

terrupt moderation [20]. As shown in Figure 6, SBB only unmasks NIC interrupt and receives packets at step 1; during other steps, SBB keeps NIC interrupt masked, to prevent it from influencing these packets' handling. Evaluation demonstrates that this approach can effectively mitigate the interrupt storm issue at high loads (Figure 8a).

4.3 Decentralized Request Preemption

Delivering user interrupt for LAPIC timer. Since a centralized entity that performs request preemption unavoidably becomes the bottleneck (§2.3.2), a decentralized preemption mechanism is needed, leveraging per-CPU capabilities to self-preempt. We observe that modern CPUs are equipped with a single LAPIC timer, which can be configured in various timing modes to trigger interrupts on the local CPU at preset intervals. This approach 1) does not depend on external components, thus offering good scalability [43, 54, 63], as well as 2) offers high precision that compiler instrumentation techniques cannot achieve [43, 51, 70].

However, the LAPIC timer has a critical drawback: its traditional preemption approach (Linux's *setitimer*) involves multiple switches between user and kernel space, resulting in an overall overhead of approximately 3μs, which is higher than several existing mechanisms (see Table 4). To integrate the timer interrupt mechanism into μs-scale network runtimes, it is essential to reduce the overhead.

Fortunately, we find that UINTR can be used to handle timer interrupts directly in user space, similar to NIC interrupts. During initialization, each worker thread configures its LAPIC timer to use the same interrupt vector as the NIC interrupt, which is also the same as UINV value. Therefore, when the timer interrupt is triggered, the CPU jumps directly to the user-level handler, completely bypassing the kernel.

Request preemption workflow. SBB performs request preemption as follows: before each LC task thread processes a request, the worker arms a one-shot timer interrupt that fires after a specified time quantum. If the current request finishes within the quantum, the LC task thread resets the timer for the next request. Otherwise, this means a long request is being processed, and a userspace timer interrupt is triggered. The worker preempts the current request, saves its execution context to the local request queue, and schedules the next request for processing.

Programming the timer interrupt costs only 50 cycles on our machine, incurring negligible overhead. This is because it only requires writing to two registers (APIC_LVTT and APIC_TMICT), both of which have been mapped into the userspace memory during initialization. As shown in §6.5, timer UINTR achieves both low latency and high scalability.

4.4 Combining Two User Interrupts

One challenge in SBB is to enable the coexistence of timer UINTR and NIC UINTR, which is not solved by prior work such as Skyloft [53] and Aeolia [61]. This challenge exists because when the CPU jumps to the user interrupt handler,

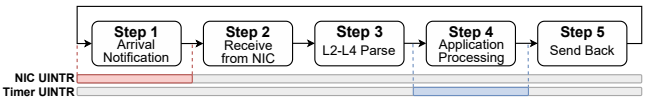


Figure 6: SBB's request processing flow.

there is a need to distinguish the interrupt source, as the CPU must handle timer and NIC UINTR differently. However, Intel does not provide support for this need. For a given CPU, there is only one set of UINTR register resources (*e.g.*, UIHANDLER, UINV). Regardless of the interrupt source, all UINTRs jump to the same handler address, making it impossible to distinguish their sources directly. Moreover, control functions such as *stui* and *ctui* can only enable or disable all UINTRs indiscriminately, which does not meet the need to disable one while keeping the other active. Therefore, enabling two device UINTRs to coexist—each performing its function without conflict—becomes a key problem that SBB must solve. To the best of our knowledge, SBB is the first work to attempt the coexistence of two UINTRs.

SBB's solution is based on two following observations. First, most modern NICs, *e.g.*, Intel E810 [2] and Mellanox ConnectX-5 [6], automatically disable interrupts after triggering them until the user program explicitly re-arms it. Second, the two types of UINTRs can be active at non-overlapping periods: NIC UINTR is used to notify the user thread of packet arrival, while timer UINTR remains active during application-layer operations. Hence, by carefully partitioning the packet processing stages, SBB can distinguish the interrupt source.

Based on these observations, we refine the packet handling flow in each worker, as shown in Figure 6. Following the Linux interrupt handling model, we divide network request processing into *top half* and *bottom half*. When a packet arrives at the NIC queue and triggers a NIC UINTR, the interrupt handler simply sets a pending flag and returns immediately. The subsequent steps, including dequeuing the packet from the NIC, protocol-stack parsing, application-layer processing, and sending back the response, are carried out by the worker thread.

By partitioning the packet processing flow at fine granularity, SBB ensures that the two UINTRs are never enabled simultaneously. SBB re-enables NIC interrupts only after completing the last batch of requests, excluding long requests that were interrupted and re-queued. As for the period during which timer UINTR should be active, we note that the variation among requests lies mainly in the application-layer processing; the durations of other stages are relatively consistent. Therefore, timer UINTR needs to be active only during application-layer execution to address head-of-line blocking.

This design involves three important advantages: first, keeping NIC UINTR disabled at other stages eliminates the interrupt storm issue when facing high RPC rates (§4.2); second, programming the timer UINTR only during application-layer execution avoids unnecessary interrupts; finally, since each

stage only has one possible interrupt source, this allows SBB to directly identify and control the corresponding UINTR, enabling two types of UINTR to coexist.

4.5 Two-Level Load Balancing Policy

4.5.1 Flow Migration

As shown in §3.3, task stealing is not suitable for handling persistent load imbalance. SBB leverages flow director functionality, which is commonly available in commodity NICs [2, 4, 6], to overcome this challenge. The NIC driver can install rules that steer packets matching specific flow characteristics (*e.g.*, 5-tuples) directly to designated receive queues, while unmatched packets continue to be distributed via RSS. The cost of installing rules is approximately 10 μs, making it suitable for infrequent use in μs-scale network runtimes.

Using this mechanism, when a worker becomes persistently overloaded due to hosting too many or overly heavy flows, SBB can migrate an entire flow to an idle worker. This enables faster and more effective balancing than packet-level stealing. By combining task stealing and flow migration, SBB employs a two-level hybrid load balancing policy that efficiently handles both temporary and persistent imbalance.

In detail, workers have two roles: victims and stealers. When a worker finds its Rx queue empty, it will become a stealer and try to obtain load from other workers, namely victims. Their behavior is as follows.

From the victim's perspective, each worker can be in one of three states: *Light*, *Busy*, or *Overloaded*. After each round of packet reception from the NIC, a worker updates its state based on whether the number of received packets exceeds or falls below predefined watermarks. To avoid transient fluctuations caused by bursty traffic, state transitions are only triggered after three consecutive rounds that satisfy the watermark condition.

Initially, all workers start in the *Light* state. When a worker receives packets above the *LIGHT_TO_BUSY_THRESHOLD* for three consecutive rounds, it transitions to *Busy* state, indicating a temporary burst in load. If a *Busy* worker continues to exceed the *BUSY_TO_OVERLOADED_THRESHOLD* for three consecutive rounds, this indicates persistent overload that cannot be alleviated by task stealing, and the worker transitions to the *Overloaded* state. Upon entering this state, the worker extracts and publishes the flow identifier of the next packet it processes, enabling other workers to migrate that flow. Conversely, if the load falls below low watermarks for three consecutive rounds, the worker transitions back to *Busy* or *Light* accordingly.

On the other hand, from the stealer's perspective, when a worker finds its NIC receive queue empty, it scans other workers to steal tasks. For each candidate worker, the stealer first checks its state. If the worker is *Light*, it likely has sufficient capacity to handle its own load, and stealing is unlikely to be beneficial, so it is skipped. If the worker is *Busy*, this indicates temporary overload, and packet-level task stealing is sufficient. The stealer attempts to access the victim's queue and steal half of the pending requests for processing. If the worker is *Overloaded*, this indicates sustained overload and flow migration is preferred. In this case, the stealer retrieves the published flow information and installs a NIC rule to redirect the corresponding flow to its own queue.

4.5.2 Enhanced Task Stealing

Although SBB introduces flow migration, we find that task stealing remains necessary. If only flow migration is used, even if we directly control flows to be perfectly the same across workers in terms of number and size—ensuring equal load over longer time scales—performance still degrades. This is because the bursty nature of network traffic leads to short-term load imbalance that flow migration alone cannot address. In summary, task stealing and flow migration are complementary: the former is effective for handling temporary imbalance, while the latter addresses persistent imbalance. Only by combining both can SBB robustly handle diverse network load conditions.

Therefore, in addition to introducing flow migration, SBB also incorporates several optimizations to task stealing. We first describe the baseline task stealing workflow, and then present the optimizations applied in SBB.

Baseline task stealing workflow. Each worker receives incoming packets from the NIC and enqueues them into a request queue. Since DPDK achieves optimal performance when receiving packets in a batch of 64, SBB sets the request queue capacity to 64 accordingly. Workers process requests in FIFO order. A worker polls the NIC for the next batch of packets only after its local request queue becomes empty, except for long requests that may be preempted and re-enqueued. Each request queue is protected by a lock.

Optimizations to task stealing. Building on this baseline design, SBB introduces the following optimizations to improve performance:

O1: Owner-exclusive block. SBB reserves the first 16 entries of each request queue as an owner-exclusive block. Requests in this region are invisible to stealers and therefore do not require locking. When the number of incoming packets in a round is below 16, all requests reside within this owner-exclusive block and are processed entirely by the owner worker, avoiding unnecessary stealing at low load. When the number of incoming packets exceeds 16, only the excess requests are placed in the stealer-visible region and may be stolen. This design significantly reduces lock contention and cache coherence overhead under low load.

O2: Avoid multi-stealing per task. A common inefficiency in task stealing systems is multi-stealing, where a request stolen by one worker is subsequently stolen again by another, as shown in Figure 4b. To eliminate this overhead, SBB ensures that once a stealer acquires requests, it processes them directly instead of enqueuing them again

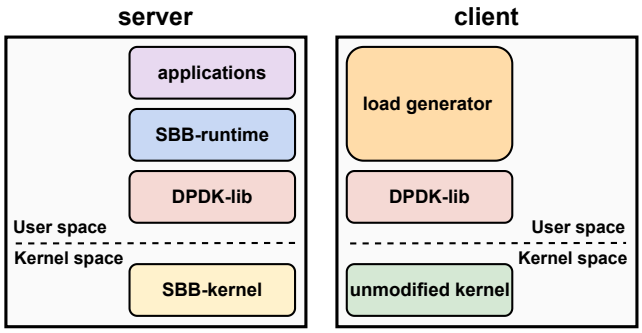


Figure 7: Implementation of server and client.

into its own queue. This prevents repeated stealing of the same request and improves overall efficiency.

O3: Batch dequeuing and stealing. Since each access to the request queue requires locking, SBB amortizes this cost via batching. The owner dequeues a batch of requests (*e.g.*, 16) per lock acquisition and processes them sequentially. Similarly, a stealer acquires the lock once and steals half of the currently available requests in the queue. This reduces the frequency of lock operations and improves cache locality.

O4: Use of try-lock. When attempting to acquire a victim's queue lock, the stealer uses a non-blocking try-lock. If the attempt fails, the stealer does not spin but immediately proceeds to the next candidate worker. This avoids unnecessary contention and wasted CPU cycles.

O5: Victim selection strategy. SBB adopts a locality-aware victim selection policy. A stealer begins scanning from the victim it successfully stole from in the previous round, as that worker is likely to remain in a *Busy* or *Overloaded* state. It then continues scanning other workers in a round-robin fashion. To further reduce contention among multiple stealers, SBB initializes each worker with a different starting point, thereby minimizing the likelihood that multiple stealers target the same victim queue simultaneously.

5 Implementation

SBB's implementation mainly consists of two parts: a kernel patch and a network runtime (Figure 7). The kernel patch, based on Linux kernel 6.12.20, comprises 2,095 lines of code (LOC) and primarily adds support for both NIC and timer UINTR. The network runtime is built on DPDK v25.07.0 [7], to benefit from its wide deployment. It contains 4,343 LOC, implementing all the designs presented in this paper. In addition, it includes a lightweight TCP/UDP network stack used in much prior work [51, 54, 70]. Although the protocol stack lacks some key features (*e.g.*, congestion control), we use it for a fair comparison.

Similar to most prior designs [38, 43, 51, 53, 74], SBB requires application rewrites to integrate with its runtime. SBB defines a runtime interface layer, as shown in Table 2. An application can be ported to the SBB runtime by implementing these three hook functions.

Function Hooks	Description
init_application()	Initialize application's data and modules.
process_request()	Process the request at application layer.
build_response()	Build the payload of application response.

Table 2: The hooks defined by the runtime interface.

For evaluation, we also implement a full-featured client packet generator. Built on unmodified kernel and DPDK, this client can produce workloads with configurable request duration distributions, packet arrival patterns, and transmission rates, consisting of 2,067 LOC.

6 Evaluation

We evaluate SBB to answer the following questions:

1. How does SBB perform with dedicated cores across various workloads? (§6.2)
2. How does SBB behave under multi-application co-location? (§6.3)
3. How well does SBB scale with an increasing number of worker cores? (§6.4)
4. What is the performance impact of each design component in SBB? (§6.5)

6.1 Methodology

Testbed. Our testbed follows the RFC 2544 [29] setup with two directly connected machines: a server running SBB or the baseline systems, and a client generating the load. The server is equipped with two NUMA nodes, each containing a 64-core Intel Xeon Platinum 8592 processor operating at 1.9 GHz. We configure the server following prior work [38]: TurboBoost, CPU idle states, CPU frequency scaling, and transparent hugepages are disabled. The client machine is equipped with a single-socket Intel Xeon 6780E processor, containing 144 physical cores running at 3.0 GHz.

For the NICs, both server and client are equipped with a 100 Gbps Intel E810 NIC [2]. We found that Caladan suffers from performance degradation when using the Intel NIC, so we run Caladan with ConnectX-5 [6] to achieve the best performance, following its guidance [5].

As for the kernel configuration, SBB runs on Ubuntu 24.04 with its custom kernel; while for baseline evaluations, we follow each specific demand: Shinjuku and Concord run on Ubuntu 18.04 with kernel 4.4.0; Caladan and TQ run on Ubuntu 22.04 with kernel 5.15.0; Skyloft runs on Ubuntu 24.04 with its custom kernel. As SBB and baselines are userspace network runtimes, we believe that this does not change the conclusions we reach.

Systems evaluated. Since SBB integrates three types of scheduling, we conduct experiments to evaluate each scheduling type individually. As most prior work does not perform all these scheduling (Table 1), we compare with each

Workloads	Request	Runtime (μs)	Ratio
Fixed(1)	N/A	1	100%
High Bimodal	Short	1	50%
	Long	100	50%
Extreme Bimodal	Short	0.5	99.5%
	Long	500	0.5%
RocksDB-lt	GET	1.25	100%
RocksDB-ht	GET	1.25	50%
	SCAN	613	50%

Table 3: The list of some evaluated workloads.

only for the problem it targets. Since request load balancing is the basis of network runtime, it is evaluated in all experiments. For request preemption, we select Shinjuku [54], Concord [51], and TQ [70] to evaluate SBB's preemption mechanism. For CPU allocation, we compare with Skyloft [53], Caladan [38], and its optimized version, Caladan-DL, which trades off tail latency against CPU efficiency using a Delay Range policy [71].

Workload. To make a fair comparison with each baseline, we follow these prior works to select workloads. Overall, these workloads either run on dedicated cores (§6.2) or share cores with other tasks (§6.3), to evaluate SBB's ability to perform request preemption and CPU allocation, respectively.

To evaluate request preemption, we follow prior work [43, 51, 70] to use one synthetic and one real-world application. The synthetic workload is a server application that spins for the duration specified by each request, which allows us to evaluate SBB under a range of service time distributions. Specifically, we run the Fixed(1), High-Bimodal, and Extreme-Bimodal workloads. Fixed(1) is a low-dispersion synthetic workload with a fixed 1 μs service time for all requests [51, 54]. High-Bimodal is modeled after YCSB's Workload A, which has a balanced read/update mix [31]. Extreme-Bimodal is modeled after Facebook's USR workload, which is dominated by short GET requests with a small fraction of much longer requests [18].

For real-world evaluation, we use RocksDB [14], a popular and widely deployed key-value store developed by Facebook [37]. We build a custom network server atop the RocksDB storage backend. We disable logging and keep all data fit in memory to better stress networking.

To evaluate CPU allocation, we follow Caladan's configuration: Memcached [11] serves as the latency-critical (LC) task, and the swaptions benchmark from PARSEC [27] serves as the best-effort (BE) task. Memcached is an in-memory key-value store with light-tailed workload patterns, while swaptions application computes portfolio prices using Monte Carlo simulations.

6.2 Handling Various Kinds of Loads

Table 3 summarizes the distribution of request duration for various workloads. Unless otherwise specified, all systems use 16 threads, each on a dedicated core.

We first compare the systems under a low-dispersion workload: Fixed(1). In such a scenario, no head-of-line blocking occurs; hence, preemption provides no benefits. Figure 8a shows the results. SBB sustains 9.7MRPS under the 99.9th-percentile slowdown SLO of 50×, achieving over 90% higher throughput than TQ, Concord, and Shinjuku. This improvement mainly stems from SBB's decentralized load balancing policy: when faced with high rates of requests, there is no centralized dispatcher becoming the bottleneck of the system.

We evaluate SBB's ability to perform request preemption with two synthetic high-dispersion workloads, with Figure 8b and Figure 8c showing the results. For the High-Bimodal workload, SBB reaches 260KRPS, outperforming baselines by more than 30%. Under the Extreme-Bimodal workload, SBB achieves more than 40% increase in throughput. These gains are attributed to SBB's decentralized, low-overhead preemption mechanism, demonstrating that even with a modest number of cores and without scalability in mind, SBB delivers superior performance.

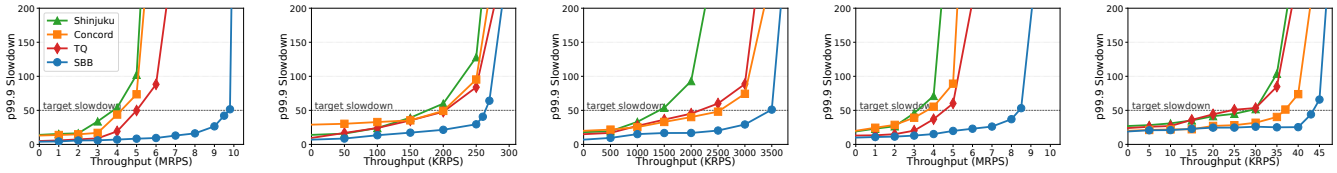
To evaluate SBB's performance on real-world workloads, we run RocksDB with both light-tailed (Figure 8d) and heavy-tailed patterns (Figure 8e). At the same SLO target, SBB supports 20% to 80% higher throughput over prior work. Notably, these comparisons use a small number of workers for all systems; thanks to its better scalability, SBB will exhibit even larger performance gains as the number of workers increases.

Summary. SBB provides high performance with dedicated cores across various workloads, due to its load balancing policy and request preemption mechanism.

6.3 Sharing CPU Resources

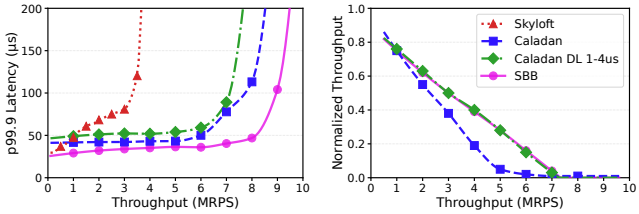
To demonstrate that SBB can improve CPU utilization by yielding CPU resources to BE tasks while LC tasks have low load, we compare it with Skyloft, Caladan and Caladan-DL. We use 16 CPU cores, where Memcached and swaptions are each launched with 16 threads and bound to dedicated cores. For Skyloft and Caladan, one more dedicated core is required for the iokernel. In contrast, SBB has no centralized component; each core runs both a Memcached thread and a swaptions thread. Clients send packets following a Poisson distribution [72]. When a packet arrives at the NIC, the Memcached thread is scheduled to run and yields the CPU core to the swaptions thread during idle periods.

Figure 9a shows the results of the LC task. Skyloft performs worst for the same reason as Shinjuku: the centralized dispatcher limits the overall throughput. Unlike Skyloft, by allowing each worker to receive packets directly from the NIC instead of relying on a centralized dispatcher, both SBB and Caladan surpass the 5MRPS limit. Under an SLO of p99.9 latency less than 100μs, SBB achieves throughput improvements of 28% and 15% over the two Caladan variants, respectively. We attribute this to its proactive scheduling



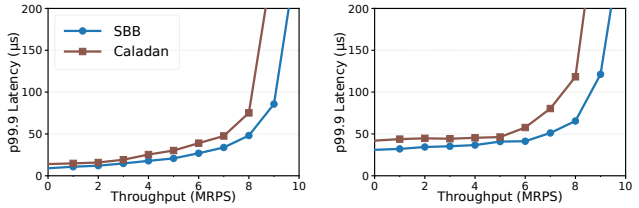
(a) Fixed(1) **(b)** High Bimodal **(c)** Extreme Bimodal **(d)** RocksDB-lt **(e)** RocksDB-ht

Figure 8: Performance comparison between Shinjuku, Concord, TQ, and SBB under various workloads.



(a) Tail latency for Memcached. **(b)** Throughput of swaptions.

Figure 9: Performance comparison for two applications.



(a) Tail latency for Fixed(1). **(b)** Tail latency for Memcached.

Figure 10: Co-location of multiple LC tasks.

mechanism: when a packet arrives while a BE task is running, SBB uses interrupt-based notification to immediately trigger a process switch. This approach yields lower tail latency and higher load tolerance compared to Caladan's passive strategy, which reassigns cores only after detecting packet delays.

Figure 9b illustrates the throughput of the BE task, excluding Skyloft, which runs Memcached on dedicated cores. All three systems yield CPU resources to the BE task under low load, with BE utilization decreasing as load increases. Compared to the original Caladan, Caladan-DL increases tolerance to packet delays, thereby improving CPU efficiency at the cost of higher tail latency. SBB, on the other hand, achieves the lowest tail latency while delivering CPU efficiency comparable to that of Caladan-DL, demonstrating its strength in sharing CPU resources across applications.

SBB also supports other co-location patterns, *e.g.*, multiple LC tasks co-location. We run a Memcached instance and a synthetic application on a set of cores, and use the client to generate a mix of Memcached and synthetic requests. Figure 10 shows SBB's ability to perform CPU allocation when running multiple LC tasks.

Summary. Using NIC UINTR to perform CPU allocation, SBB achieves both lower latency and higher efficiency when co-locating multiple applications.

6.4 Scaling to More Workers

By leveraging User Interrupt and two-level load balancing policy, SBB effectively performs three types of scheduling, outperforming prior work. A further and more important question is: how does SBB scale? As NIC hardware advances, processing network requests at line rate requires an increasing number of worker cores, placing greater demands on the scalability of the network runtime.

We demonstrate SBB's strong scalability through the following experiments that individually evaluate SBB's scalability for three types of scheduling (Figure 11).

First, we evaluate Fixed(1) with varying numbers of workers, using the same configuration as in Figure 8a. Under the same SLO, the achievable throughput for 16, 32, and 48 workers is approximately 9.5, 19, and 26.5MRPS, respectively (Figure 11a). For reference, we also plot ideal linear scalability, corresponding to 2× and 3× the throughput of the 16-worker case. The results show that when scaling from 16 to 32 workers, throughput nearly doubles, indicating great scalability. Scaling from 16 to 48 workers yields 2.8× higher throughput, although the growth rate slightly declines. We attribute this sublinear scaling mainly to overhead from load balancing when using too many cores, which we will discuss in §7. Despite this minor limitation, we expect SBB's throughput to continue increasing up to at least 64 cores. In contrast, Shinjuku, Concord and TQ show almost no improvements when increasing from 16 to 32 workers, due to the bottleneck of the centralized dispatcher.

Second, we use the High Bimodal and Extreme Bimodal workload to measure the maximum SLO-satisfying throughput across different worker counts. As shown in Figure 11b and 11c, when increasing from 16 to 32 workers, the throughput scales from 260 to 520KRPS, and from 3.4 to 6.8MRPS, demonstrating the perfect scalability for performing request preemption. With 48 workers, we observe a pattern similar to that of Fixed(1): though the gain is slightly below ideal linear scaling, throughput still increases by more than 35% compared to the 32-worker case. This confirms that the timer UINTR mechanism introduces no scalability issues and effectively resolves head-of-line blocking.

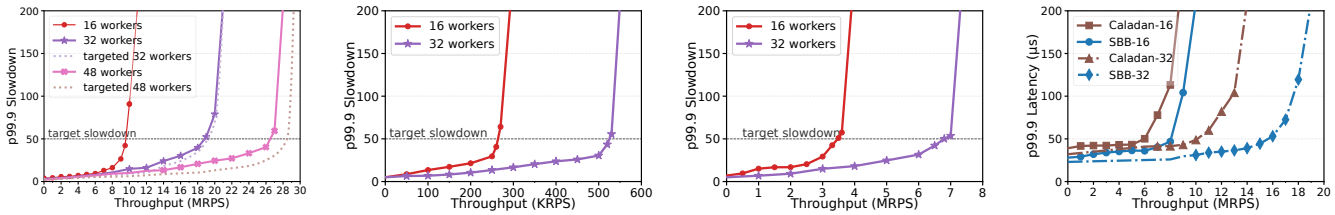


Figure 11: Scalability evaluation for various workloads. (a), (bc) and (d) are used to individually evaluate SBB's scalability for performing request load balancing, request preemption and CPU allocation.

Finally, we run experiments with both Memcached and swaptions co-located on varying worker counts to evaluate how well SBB scales while improving CPU efficiency. The setup resembles that of Figure 9. For a fair comparison focused on low tail latency, we use the original Caladan rather than Caladan-DL. Results in Figure 11d show that, under a p99.9 latency SLO of 50μs, SBB's throughput scales from 8 to 16MRPS, demonstrating excellent scalability. In contrast, Caladan scales only from 6 to 10.2MRPS. In fact, at 32 workers, Caladan nearly reaches its scaling limit: we find adding more workers slightly reduces its SLO-compliant throughput, consistent with prior observations [65]. This is because, beyond a certain number of workers, the centralized iokernel becomes a bottleneck when scanning NIC queues and assigning cores, ultimately limiting scalability.

Summary. SBB achieves good scalability for all three types of scheduling, across various workloads.

6.5 Breakdown of Each Design Component

We now evaluate the individual components of SBB using microbenchmarks.

NIC UINTR. For an actively running application thread, the fastest notification mechanism for packet arrival is continuous polling of the NIC queue, which introduces nearly negligible latency. In contrast, interrupt-based mechanisms, no matter whether they bypass the kernel or not, involve at least interrupt handler processing and thus incur additional latency. A natural concern is that interrupt latency may impose substantial overhead compared to polling.

To assess this, we design the following microbenchmark: we implement a network program that simulates processing a 1μs request while keeping all other logic identical, and vary the packet notification mechanism among polling, UINTR, and the conventional interrupt mechanism provided by DPDK. We timestamp each stage of the process and collect the results on the client side. Figure 12 presents the breakdown of round-trip latency.

The results show that on two Intel E810 100 Gbps NICs, the end-to-end propagation delay over the data link is 9μs, packet parsing in the network stack takes about 0.3μs, and the application layer consumes the specified 1μs. Compared with polling, UINTR adds an extra 0.45μs in notification latency

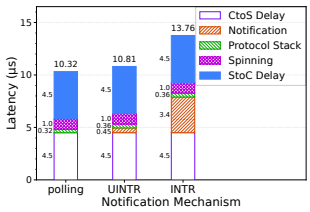


Figure 12: Breakdown of round-trip latency.

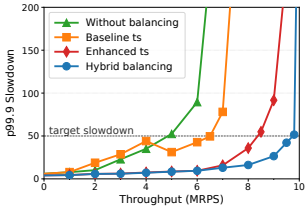


Figure 13: Different load balancing configurations.

and increases network stack overhead by 0.04μs—which we attribute to cache effects caused by the interrupt. Thus, the total additional overhead of UINTR over polling is 0.49μs, accounting for 4.7% of the end-to-end latency. We also repeat the measurement on a ConnectX-5 NIC, where the baseline link propagation delay is 4.7μs; in this case, UINTR increases the end-to-end latency by 8.1%. Given that UINTR directly enables CPU sharing and its overhead remains modest, we consider it acceptable, especially for SLOs that typically allow a slowdown factor of up to 50×.

We also compare conventional interrupts with UINTR. The notification overhead of a conventional interrupt is 3.4μs, significantly higher than UINTR's 0.45μs. This is because conventional interrupts traverse the kernel and incur scheduling overhead; the interrupt mechanism itself constitutes only a small fraction of the total cost. In conclusion, adopting NIC UINTR as the packet-notification mechanism fundamentally enables network runtimes to perform CPU allocation, while incurring low additional latency.

Timer UINTR. Table 4 compares the overhead of several common preemption mechanisms used in userspace network runtimes. Timer UINTR exhibits notably lower overhead than signals, IPI, and UIPI, allowing it to employ a smaller preemption quantum and further mitigate head-of-line blocking. Although the compiler instrumentation technique achieves lower overhead by polling, it suffers from the imprecise preemption problem [43]. More importantly, timer UINTR offers a unique architectural advantage: it enables each worker to preempt itself, eliminating the centralized component that other methods rely on.

Preemption Mechanism	System	Latency (cycles)
Signal-based timer	Linux	7697
Custom IPI	Shinjuku [54]	1878
User IPI	Aspen-KB [43]	1381
Timer UINTR	SBB	688

Table 4: Latency comparison of preemption mechanisms.

Load balancing policy. To evaluate SBB's two-level load balancing policy, we run the Fixed(1) test with various balancing strategies. Keeping all other configurations the same, we modify the balancing policy as follows: 1) **`Without balancing`**: each worker only processes requests in its own Rx queue assigned by RSS; 2) **`Baseline ts`**: a basic task stealing implementation similar to Shenango's; 3) **`Enhanced ts`**: the improved task stealing scheme described in §4.5.2; 4) **`Hybrid balancing`**: the two-level load balancing policy used by SBB.

Without any balancing policy, SBB reaches its SLO limit much earlier because, under RSS's random packet-to-queue assignment, system capacity is constrained by the most heavily loaded worker. With basic task stealing, SBB achieves higher throughput, but an interesting phenomenon is that it suffers from even higher tail latency at lower loads. Through further analysis, we find that this is due to too much unnecessary task stealing, which provides no benefit but instead introduces synchronization overhead. Based on the analysis, we enhance task stealing with several optimizations (§4.5.2), which mainly reduce unnecessary stealings and synchronization overhead, and increase the throughput by 30%. Nevertheless, when the load is too high, there are still too many stealings incurring overhead, most of which come from only a few flows. SBB's two-level load balancing policy effectively mitigates this problem. By addressing temporal imbalance with task stealing and persistent imbalance with flow migration, SBB achieves notably higher throughput, which demonstrates the necessity of the combination of task stealing and flow migration.

Summary. NIC and timer User Interrupt can function with low overhead, and the hybrid policy can perform load balancing effectively.

7 Discussion

Hardware-offloaded scheduling. To process RPCs at microsecond-scale latencies, besides those works running on commodity hardware (Table 1), there are also some recent works leveraging NIC-CPU co-design [32, 46, 49, 64, 77, 85]. They use the on-board FPGA or ARM of SmartNICs to offload the scheduling, such as implementing JBSQ policy for request load balancing [49, 57].

The hardware-offloaded approach does have some exclusive advantages, as the programmable NIC has a global view and higher processing speed. However, they also have some challenges: some hardware is able to compare only a limited number of registers in one pass, which means the performance also degrades when scaling to more workers [57]. More importantly, these approaches rely on specialized hardware modification, which precludes wide adoption in the near future [51]. In comparison, SBB uses only a software implementation and relies on commodity hardware functions (*i.e.*, User Interrupt, RSS, flow director), suitable for both datacenters and VMs in public clouds.

SBB's scalability limitation. Although SBB eliminates all centralized components and exhibits much better scalability than prior work, it is not perfectly scalable beyond 48 cores (Figure 11a). This is attributed to the synchronization overhead of inter-core communication, *i.e.*, task stealing and flow migration. Although SBB leverages user interrupt to enable each core to perform request preemption and CPU allocation independently, and introduces the two-level load balancing policy to largely reduce task stealing frequency, the overhead still exists, and will enlarge when scaling to more cores.

We identify three potential directions to achieve better scalability. First, we can offload scheduling to SmartNICs, as discussed earlier. Second, we can continue to improve task stealing algorithm to satisfy the commutativity rule [30], which enables better scalability. Although there is already much work improving task stealing [28, 36, 40, 44, 80, 86], no algorithms follow the commutativity rule. Third, we may leverage some commodity hardware features, similar to using UINTR to perform request preemption and CPU allocation. We notice that Dynamic Load Balancer (DLB) is a new feature in Intel Xeon Scalable Processor, which is designed to mitigate the load imbalance problem [68]. Although it could not be used trivially in SBB's purely decentralized architecture, we view leveraging this to perform load balancing with better scalability as promising future work.

8 Conclusion

This paper presents SBB, a purely decentralized userspace network runtime. Designed with scalability as its foremost objective, SBB performs request preemption, CPU allocation, and request load balancing to provide high performance and efficiency, without any centralized components as potential bottlenecks. Our evaluation demonstrates that SBB outperforms baseline systems by 1.7× to 5.2× across a variety of scenarios and workloads.

9 Acknowledgment

We thank the anonymous shepherd and reviewers for their insightful feedback. This work was supported by the National Cyber Security-National Science and Technology Major Project (under Grant No: 2025ZD1503200), the National Science Foundation of China (No.62372011, 92582118, 62032001), and Zhongguancun Laboratory. Diyu Zhou is the corresponding author.

References

[1] Cornelis Unveils CN6000 800 Gbps Ethernet SuperNIC for AI and HPC at Scale. https://www.cornelis.com/stories/cornelis-unveils-cn6000-800-gbps-ethernet-supernic-for-ai-and-hpc-at-scale.

[2] Intel® Ethernet Controller E810 Datasheet. https://www.intel.com/content/www/us/en/content-details/613875/intel-ethernet-controller-e810-datasheet.html.

[3] AMD Pensando™ Pollara 400 AI NIC: Transforming Scale-Out Networks for the Modern Data Center. https://www.amd.com/en/products/network-interface-cards/pensando.html.

[4] Broadcom P2200G Dual-Port 200GbE PCIe Ethernet NIC. https://www.broadcom.com/products/ethernet-connectivity/network-adapters/p2200g.

[5] Caladan Source: runtime/net/core.c. https://github.com/shenango/caladan/blob/73107415ed007fdcd8e54074c6d14cb58370422a/runtime/net/core.c#L854.

[6] NVIDIA Mellanox ConnectX-5 Ethernet Adapter. https://www.nvidia.com/en-us/networking/ethernet-adapters/connectx-5/.

[7] DPDK: Data Plane Development Kit. https://dpdk.org.

[8] Intel Ethernet: Driving Optimized, Secure, and Scalable Performance from Data Center to Edge, . https://www.intel.com/content/www/us/en/products/details/ethernet.html.

[9] Intel® 64 and IA-32 Architectures Software Developer's Manual, . https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html.

[10] UINTR Linux Kernel. https://github.com/intel/uintr-linux-kernel.

[11] Memcached - A Distributed Memory Object Caching System. https://memcached.org.

[12] NVIDIA Launches BlueField-4: The Processor Powering the Operating System of AI Factories. https://blogs.nvidia.com/blog/bluefield-4-ai-factory/.

[13] Redis. https://redis.io/.

[14] RocksDB. http://rocksdb.org/.

[15] Introduction to Receive Side Scaling. https://docs.microsoft.com/en-us/windows-hardware/drivers/network/introduction-to-receive-side-scaling.

[16] Seastar – High-Performance Service-Application Framework. https://github.com/scylladb/seastar/.

[17] x86 User Interrupts Support. https://lwn.net/Articles/869140/.

[18] B. Atikoglu, Y. Xu, E. Frachtenberg, S. Jiang, and M. Paleczny. Workload Analysis of a Large-Scale Key-Value Store. In *Proceedings of the 40th ACM International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS)*, London, UK, June 2012.

[19] B. Aydogmus, L. Guo, D. Zuberi, T. Garginkel, D. Tullsen, A. Ousterhout, and K. Taram. Extended User Interrupts (xUI): Fast and Flexible Notification Without Polling. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, New York, NY, Mar. 2025.

[20] L. A. Barroso, K. Gharachorloo, R. McNamara, A. Nowatzyk, S. Qadeer, B. Sano, S. Smith, R. Stets, and B. Verghese. Eliminating Receive Livelock in an Interrupt-Driven Kernel. In *Proceedings of the ACM Transactions on Computer Systems*, New York, NY, Aug. 1997.

[21] L. A. Barroso, J. Clidaras, and U. Hölzle. *The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines*. Synthesis Lectures on Computer Architecture. Morgan & Claypool Publishers, 2013.

[22] L. A. Barroso, M. Marty, D. Patterson, and P. Ranganathan. Attack of the Killer Microseconds. *Communications of the ACM*, 2017.

[23] A. Belay, A. Bittau, A. J. Mashtizadeh, D. Terei, D. Mazières, and C. Kozyrakis. Dune: Safe User-Level Access to Privileged CPU Features. In *Proceedings of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Hollywood, CA, Oct. 2012.

[24] A. Belay, G. Prekas, A. Klimovic, S. Grossman, C. Kozyrakis, and E. Bugnion. IX: A Protected Dataplane Operating System for High Throughput and Low Latency. In *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Broomfield, CO, Oct. 2014.

[25] T. Benson, A. Anand, A. Akella, and M. Zhang. Understanding Data Center Traffic Characteristics. In *Proceedings of the 1st ACM Workshop on Research on Enterprise Networking (WREN)*, Barcelona, Spain, Aug. 2009.

[26] T. Benson, A. Akella, and D. A. Maltz. Network Traffic Characteristics of Data Centers in the Wild. In *Proceedings of the 10th ACM Internet Measurement Conference (IMC)*, Melbourne, Australia, Nov. 2010.

[27] C. Bienia. *Benchmarking Modern Multiprocessors*. PhD thesis, Princeton University, 2011.

[28] R. D. Blumofe and C. E. Leiserson. Scheduling Multithreaded Computations by Work Stealing. In *Proceedings of the 35th IEEE Symposium on Foundations of Computer Science (FOCS)*, Santa Fe, NM, Nov. 1994.

[29] S. Bradner and J. McQuaid. Benchmarking Methodology for Network Interconnect Devices. Technical Report RFC 2544. https://www.rfc-editor.org/rfc/rfc2544.txt.

[30] A. T. Clements, M. F. Kaashoek, E. Kohler, R. T. Morris, and N. Zeldovich. The Scalable Commutativity Rule: Designing Scalable Software for Multicore Processors. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP)*, Farmington, PA, Nov. 2013.

[31] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking Cloud Serving Systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing (SoCC)*, Indianapolis, IN, June 2010.

[32] A. Daglis, M. Sutherland, and B. Falsafi. RPCValet: NI-Driven Tail-Aware Balancing of μs-Scale RPCs. In *Proceedings of the 24th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Providence, RI, Apr. 2019.

[33] J. Dean and L. A. Barroso. The Tail at Scale. *Communications of the ACM*, 2013.

[34] H. M. Demoulin, J. Fried, I. Pedisich, M. Kogias, B. M. Loo, L. T. X. Phan, and I. Zhang. When Idling Is Ideal: Optimizing Tail-Latency for Heavy-Tailed Datacenter Workloads with Perséphone. In *Proceedings of the 28th ACM Symposium on Operating Systems Principles (SOSP)*, Koblenz, Germany, Oct. 2021.

[35] D. Didona and W. Zwaenepoel. Size-Aware Sharding for Improving Tail Latencies in In-Memory Key-Value Stores. In *Proceedings of the 16th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, Boston, MA, Feb. 2019.

[36] X. Ding, K. Wang, P. B. Gibbons, and X. Zhang. BWS: Balanced Work Stealing for Time-Sharing Multicores. In *Proceedings of the 7th European Conference on Computer Systems (EuroSys)*, Bern, Switzerland, Apr. 2012.

[37] S. Dong, A. Kryczka, Y. Jin, and M. Stumm. RocksDB: Evolution of Development Priorities in a Key-value Store Serving Large-scale Applications. *ACM Transactions on Storage*, 2021.

[38] J. Fried, Z. Ruan, A. Ousterhout, and A. Belay. Caladan: Mitigating Interference at Microsecond Timescales. In *Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Virtual, Oct. 2020.

[39] J. Fried, G. I. Chaudhry, E. Saurez, E. Choukse, Í. Goiri, S. Elnikety, R. Fonseca, and A. Belay. Making Kernel Bypass Practical for the Cloud with Junction. In *Proceedings of the 21st USENIX Symposium*

on Networked Systems Design and Implementation (NSDI), Santa Clara, CA, Apr. 2024.

[40] M. Frigo, C. E. Leiserson, and K. H. Randall. The Implementation of the Cilk-5 Multithreaded Language. In *Proceedings of the 1998 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, Montreal, Canada, June 1998.

[41] S. Ghorbani, Z. Yang, P. B. Godfrey, Y. Ganjali, and A. Firoozshahian. DRILL: Micro Load Balancing for Low-Latency Data Center Networks. In *Proceedings of the 2017 ACM Special Interest Group on Data Communication Conference (SIGCOMM)*, Los Angeles, CA, Aug. 2017.

[42] H. Golestani, A. Mirhosseini, and T. F. Wenisch. Software Data Planes: You Can't Always Spin to Win. In *Proceedings of the 10th ACM Symposium on Cloud Computing (SoCC)*, Santa Cruz, CA, Nov. 2019.

[43] L. Guo, D. Zuberi, T. Garfinkel, and A. Ousterhout. The Benefits and Limitations of User Interrupts for Preemptive Userspace Scheduling. In *Proceedings of the 22nd USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, Philadelphia, PA, Apr. 2025.

[44] A. Handleman, K. Singer, T. B. Schardl, and I.-T. A. Lee. Towards Zero Spawn Overhead: Work Stealing Without Deques. In *Proceedings of the 30th ACM Symposium on Principles and Practice of Parallel Programming (PPOPP)*, Las Vegas, NV, Mar. 2025.

[45] K. Huang, J. Zhou, Z. Zhao, D. Xie, and T. Wang. Low-Latency Transaction Scheduling via Userspace Interrupts: Why Wait or Yield When You Can Preempt? *Proceedings of the ACM on Management of Data*, 2025.

[46] J. T. Humphries, K. Kaffes, D. Mazières, and C. Kozyrakis. Mind the Gap: A Case for Informed Request Scheduling at the NIC. In *Proceedings of the 18th ACM Workshop on Hot Topics in Networks (HotNets)*, Princeton, NJ, Nov. 2019.

[47] J. Hwang, M. Vuppalapati, S. Peter, and R. Agarwal. Rearchitecting Linux Storage Stack for μs Latency and High Throughput. In *Proceedings of the 15th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Virtual, Oct. 2021.

[48] M. G. Iatrou, A. G. Voyiatzis, and D. N. Serpanos. Network Stack Optimization for Improved IPsec Performance on Linux. In *Proceedings of the International Conference on Security and Cryptography (SECRYPT)*, Milan, Italy, July 2009. SciTePress.

[49] S. Ibanez, A. Mallery, S. Arslan, T. Jepsen, M. Shahbaz, C. Kim, and N. McKeown. The nanoPU: A Nanosecond Network Stack for Datacenters. In *Proceedings of the 15th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Virtual, Oct. 2021.

[50] C. Iorgulescu, R. Azimi, Y. Kwon, S. Elnikety, M. Syamala, V. Narasayya, H. Herodotou, P. Tomita, A. Chen, J. Zhang, and J. Wang. PerfIso: Performance Isolation for Commercial Latency-Sensitive Services. In *Proceedings of the 2018 USENIX Annual Technical Conference (ATC)*, Boston, MA, July 2018.

[51] R. Iyer, M. Unal, M. Kogias, and G. Candea. Achieving Microsecond-Scale Tail Latency Efficiently with Approximate Optimal Scheduling. In *Proceedings of the 29th ACM Symposium on Operating Systems Principles (SOSP)*, Koblenz, Germany, Oct. 2023.

[52] E. Jeong, S. Woo, M. Jamshed, H. Jeong, S. Ihm, D. Han, and K. Park. mTCP: A Highly Scalable User-Level TCP Stack for Multicore Systems. In *Proceedings of the 11th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, Seattle, WA, Mar. 2014.

[53] Y. Jia, K. Tian, Y. You, Y. Chen, and K. Chen. Skyloft: A General High-Efficient Scheduling Framework in User Space. In *Proceedings of the 30th ACM Symposium on Operating Systems Principles (SOSP)*, Austin, TX, Nov. 2024.

[54] K. Kaffes, T. Chong, J. T. Humphries, A. Belay, D. Maziéres, and C. Kozyrakis. Shinjuku: Preemptive Scheduling for μsecond-scale Tail Latency. In *Proceedings of the 16th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, Boston, MA, Feb. 2019.

[55] R. Kapoor, G. Porter, M. Tewari, G. M. Voelker, and A. Vahdat. Chronos: Predictable Low Latency for Data Center Applications. In *Proceedings of the 3rd ACM Symposium on Cloud Computing (SoCC)*, San Jose, CA, Oct. 2012.

[56] H. Kim, S. Wang, and R. Rajkumar. Responsive and Enforced Interrupt Handling for Real-Time System Virtualization. In *Proceedings of the 21st IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, Hong Kong, China, Aug. 2015.

[57] M. Kogias, G. Prekas, A. Ghosn, J. Fietz, and E. Bugnion. R2P2: Making RPCs First-Class Datacenter Citizens. In *Proceedings of the 2019 USENIX Annual Technical Conference (ATC)*, Renton, WA, July 2019.

[58] A. Kumar, R. Katkam, P. Chaudhary, P. Naik, and M. Vutukuru. AppSteer: Framework for Improving Multicore Scalability of Network Functions via Application-Aware Packet Steering. In *Proceedings of the 24th IEEE International Symposium on Cluster, Cloud and Internet Computing (CCGrid)*, Philadelphia, PA, May 2024.

[59] J. Lei, K. Zhao, D. Hou, and F. Zhou. Space Network Emulation System Based on a User-Space Network Stack. *ZTE Communications*, 2025.

[60] J. Leverich and C. Kozyrakis. Reconciling High Server Utilization and Sub-Millisecond Quality-of-Service. In *Proceedings of the 9th European Conference on Computer Systems (EuroSys)*, Amsterdam, Netherlands, Apr. 2014.

[61] C. Li, R. Yi, Z. Zhang, J. Liu, C. Min, J. Zhang, Y. Luo, X. Wang, Z. Wang, and D. Zhou. Aeolia: A Fast and Secure Userspace Interrupt-Based Storage Stack. In *Proceedings of the 31st ACM Symposium on Operating Systems Principles (SOSP)*, New York, NY, Oct. 2025.

[62] J. Li, N. K. Sharma, D. R. K. Ports, and S. D. Gribble. Tales of the Tail: Hardware, OS, and Application-level Sources of Tail Latency. In *Proceedings of the 5th ACM Symposium on Cloud Computing (SoCC)*, Seattle, WA, Nov. 2014.

[63] Y. Li, N. Lazarev, D. Koufaty, T. Yin, A. Anderson, Z. Zhang, G. E. Suh, K. Kaffes, and C. Delimitrou. LibPreemptible: Enabling Fast, Adaptive, and Hardware-Assisted User-Space Scheduling. In *Proceedings of the 30th IEEE Symposium on High Performance Computer Architecture (HPCA)*, Edinburgh, UK, Mar. 2024.

[64] J. Lin, A. Cardoza, T. Khan, Y. Ro, B. E. Stephens, H. Wassel, and A. Akella. RingLeader: Efficiently Offloading Intra-Server Orchestration to NICs. In *Proceedings of the 20th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, Boston, MA, Apr. 2023.

[65] J. Lin, Y. Chen, S. Gao, and Y. Lu. Fast Core Scheduling with Userspace Process Abstraction. In *Proceedings of the 30th ACM Symposium on Operating Systems Principles (SOSP)*, Austin, TX, Nov. 2024.

[66] B. Liu, X. Huang, Q. Li, Z. Huang, Y. Sun, W. Li, J. Zhang, P. Yin, and K. Chen. CEIO: A Cache-Efficient Network I/O Architecture for NIC-CPU Data Paths. In *Proceedings of the 2025 ACM Special Interest Group on Data Communication Conference (SIGCOMM)*, Coimbra, Portugal, Sept. 2025.

[67] D. Lo, L. Cheng, R. Govindaraju, P. Ranganathan, and C. Kozyrakis. Heracles: Improving Resource Efficiency at Scale. In *Proceedings of the 42nd ACM/IEEE International Symposium on Computer Architecture (ISCA)*, Portland, OR, June 2015.

[68] J. Lou, S. Vanavasam, Y. Yuan, R. Wang, and N. S. Kim. Dynamic Load Balancer in Intel Xeon Scalable Processor: Performance Analyses, Enhancements, and Guidelines. In *Proceedings of the 52nd ACM/IEEE International Symposium on Computer Architecture (ISCA)*, Tokyo, Japan, June 2025.

[69] H. Luo, S. Hu, W. Wang, Y. Tang, and J. Zhou. Research on Multi-Core Processor Analysis for WCET Estimation. *ZTE Communications*, 2024.

[70] Z. Luo, S. Son, D. Bali, E. Amaro, A. Ousterhout, S. Ratnasamy, and S. Shenker. Efficient Microsecond-Scale Blind Scheduling with Tiny Quanta. In *Proceedings of the 29th ACM International Conference on Ar-*

chitectural Support for Programming Languages and Operating Systems (ASPLOS), La Jolla, CA, Apr. 2024.

[71] S. McClure, A. Ousterhout, S. Shenker, and S. Ratnasamy. Efficient Scheduling Policies for Microsecond-Scale Tasks. In *Proceedings of the 19th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, Renton, WA, Apr. 2022.

[72] D. Meisner, C. M. Sadler, L. A. Barroso, W.-D. Weber, and T. F. Wenisch. Power Management of Online Data-Intensive Services. In *Proceedings of the 38th ACM/IEEE International Symposium on Computer Architecture (ISCA)*, San Jose, CA, June 2011.

[73] A. Ousterhout, A. Belay, and I. Zhang. Just In Time Delivery: Leveraging Operating Systems Knowledge for Better Datacenter Congestion Control. In *Proceedings of the 11th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud)*, Renton, WA, July 2019.

[74] A. Ousterhout, J. Fried, J. Behrens, A. Belay, and H. Balakrishnan. Shenango: Achieving High CPU Efficiency for Latency-Sensitive Datacenter Workloads. In *Proceedings of the 16th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, Boston, MA, Feb. 2019.

[75] S. Peter, J. Li, I. Zhang, D. R. K. Ports, D. Woos, A. Krishnamurthy, T. Anderson, and T. Roscoe. Arrakis: The Operating System Is the Control Plane. In *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Broomfield, CO, Oct. 2014.

[76] G. Prekas, M. Kogias, and E. Bugnion. ZygOS: Achieving Low Tail Latency for Microsecond-Scale Networked Tasks. In *Proceedings of the 26th ACM Symposium on Operating Systems Principles (SOSP)*, Shanghai, China, Oct. 2017.

[77] M. Sutherland, S. Gupta, B. Falsafi, V. Marathe, D. Pnevmatikatos, and A. Daglis. The NEBULA RPC-Optimized Architecture. In *Proceedings of the 47th ACM/IEEE International Symposium on Computer Architecture (ISCA)*, Virtual, May 2020.

[78] S. Tu, W. Zheng, E. Kohler, B. Liskov, and S. Madden. Speedy Transactions in Multicore In-Memory Databases. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP)*, Farmington, PA, Nov. 2013.

[79] A. Verma, L. Pedrosa, M. Korupolu, D. Oppenheimer, E. Tune, and J. Wilkes. Large-Scale Cluster Management at Google with Borg. In *Proceedings of the 10th European Conference on Computer Systems (EuroSys)*, Bordeaux, France, Apr. 2015.

[80] J. Wang, B. Trach, M. Fu, D. Behrens, J. Schwender, Y. Liu, J. Lei, V. Vafeiadis, H. Härtig, and H. Chen. BWoS: Formally Verified Block-Based Work Stealing for Parallel Processing. In *Proceedings of the 17th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Boston, MA, July 2023.

[81] A. Wierman and B. Zwart. Is Tail-Optimal Scheduling Possible? *Operations Research*, 2012.

[82] Q. Xu, S. Miano, X. Gao, T. Wang, A. Murugadass, S. Zhang, A. Sivaraman, G. Antichi, and S. Narayana. State-Compute Replication: Parallelizing High-Speed Stateful Packet Processing. In *Proceedings of the 22nd USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, Philadelphia, PA, Apr. 2025.

[83] I. Zhang, A. Raybuck, P. Patel, K. Olynyk, J. Nelson, O. S. Navarro Leija, A. Martinez, J. Liu, A. Kornfeld Simpson, S. Jayakar, P. H. Penna, M. Demoulin, P. Choudhury, and A. Badam. The Demikernel Datapath OS Architecture for Microsecond-Scale Datacenter Systems. In *Proceedings of the 28th ACM Symposium on Operating Systems Principles (SOSP)*, Koblenz, Germany, Oct. 2021.

[84] X. Zhang, E. Tune, R. Hagmann, R. Jnagal, V. Gokhale, and J. Wilkes. CPI^2: CPU Performance Isolation for Shared Compute Clusters. In *Proceedings of the 8th European Conference on Computer Systems (EuroSys)*, Prague, Czech Republic, Apr. 2013.

[85] J. Zhao, I. Uwizeyimana, K. Ganesan, M. C. Jeffrey, and N. E. Jerger. ALTOCUMULUS: Scalable Scheduling for Nanosecond-Scale Remote Procedure Calls. In *Proceedings of the 55th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, Chicago, IL, Oct. 2022.

[86] S. Zhao, H. Gu, and A. J. Mashtizadeh. SKQ: Event Scheduling for Optimizing Tail Latency in a Traditional OS Kernel. In *Proceedings of the 2021 USENIX Annual Technical Conference (ATC)*, Virtual, July 2021.

[87] L. Zhu, Y. Shen, E. Xu, B. Shi, T. Fu, S. Ma, S. Chen, Z. Wang, H. Wu, X. Liao, Z. Yang, Z. Chen, W. Lin, Y. Hou, R. Liu, C. Shi, J. Zhu, and J. Wu. Deploying User-Space TCP at Cloud Scale with Luna. In *Proceedings of the 2023 USENIX Annual Technical Conference (ATC)*, Boston, MA, July 2023.

ARTIFACT EVALUATED
USENIX
AVAILABLE

Rakaia: Scalable In-Kernel Scheduling For TCP-Based RPCs

Rui Yang
EPFL, Switzerland

Konstantinos Prasopoulos
EPFL, Switzerland

Edouard Bugnion
EPFL, Switzerland

Abstract

Delivering RPCs with high throughput and low latency demands work-conserving scheduling across many CPU cores and eliminating head-of-line (HOL) blocking across all messages. By exposing per-connection byte streams rather than messages to userspace, the POSIX TCP API inherently induces HOL blocking both within and across connections. To mitigate HOL blocking, RPC frameworks such as gRPC must reconstruct message semantics in userspace through additional abstractions including dedicated I/O threads, work queues, and worker thread pools, introducing significant context switching and synchronization overheads.

This paper presents Rakaia, a framework that hides all TCP-level abstractions from userspace and exposes a purely message-oriented API. By performing message parsing and work-conserving scheduling directly in the kernel's TCP receive path, *at the earliest possible point*, Rakaia efficiently eliminates HOL blocking and avoids the heavy userspace machinery imposed by stream-based APIs.

We implemented Rakaia as a Linux kernel module with support for kTLS. Rakaia is compatible with the kernel's TCP stack and existing RPC protocols. We also adapted gRPC to use Rakaia's API. Our evaluation shows Rakaia: *(i)* consistently eliminates HOL blocking across a wide range of connection counts; *(ii)* achieves up to 5× higher throughput-under-SLO than KCM, Linux's current in-kernel message API over TCP; *(iii)* improves gRPC-Go's throughput-under-SLO by up to 1.56×, and gRPC-C++'s by up to 2.69×; and *(iv)* improves the throughput-under-SLO for real-world applications including Silo running TPC-C and OpenTelemetry Collector by 1.39× and 1.42×, respectively.

1 Introduction

Remote Procedure Calls (RPCs) [6] underpin nearly all communication in modern datacenters [5, 21, 41, 48, 52, 65]. A single cloud application may consist of hundreds of microservices deployed across thousands of containers, all communicating via RPCs. This centrality of RPCs has motivated extensive research in transport protocols, middleboxes, dataplanes, and frameworks [1, 2, 4, 7, 10, 11, 19, 27, 34, 35, 40, 51, 60, 61], with one goal: minimizing tail latency to meet strict service-level objectives (SLOs) [3, 13].

Cloud applications commonly multiplex RPCs over TCP, which remains the de-facto transport in production. Yet TCP's stream-oriented abstraction is a poor fit for message-oriented RPCs [57]. When exposed to userspace through the POSIX API, the kernel's TCP/IP stack delivers unstructured byte-streams, with no notion of message boundaries. As a result, RPCs sharing the same TCP stream are processed sequentially by default, causing HOL blocking whenever a slow RPC delays subsequent ones. Moreover, uneven request arrivals across connections can severely imbalance work across cores.

Existing RPC frameworks attempt to compensate for this mismatch entirely in userspace. To enable concurrent, out-of-order RPC processing on top of a sequential TCP stream, frameworks like gRPC must reconstruct message semantics using additional userspace abstractions: dedicated I/O threads to demultiplex byte streams and reassemble messages, work queues to hand messages across threads, and worker thread pools to execute RPCs. This layered, interaction-intensive pipeline introduces significant context switching, communication, and synchronization overheads. In practice, these overheads limit scalability: for example, increasing the number of TCP connections to a 20-hardware-thread gRPC-Go server from 80 to 5,000 increases the number of goroutines by 16×, and reduces throughput-under-SLO by 11% (§5.3). This poor scaling persists even though the Linux kernel's TCP stack can easily handle millions of concurrent TCP connections.

A rich body of prior work addresses this problem by avoiding TCP altogether. These systems do so either by introducing a custom message-oriented transport (*e.g.,* Homa) [8, 51, 56, 60] or relying on userspace network stacks (*e.g.,* eRPC) [29, 32, 35, 40]. By abandoning TCP's stream abstraction, they eliminate HOL blocking at its source and can deliver excellent RPC performance. While effective, these systems are difficult to deploy. They often require custom clients, non-

standard infrastructure, or dedicated hardware, which limits their practicality in today's multi-tenant cloud environments.

We therefore raise the following question: *Can one achieve high-performance RPCs free of HOL blocking, without the complex userspace machinery or deployability tradeoffs?*

We answer this question with **Rakaia**, an in-kernel architecture that eliminates HOL blocking for RPC frameworks, while preserving TCP wire compatibility. Our two key insights are: *(i)* message parsing and scheduling should occur in the same `softirq` context that processes incoming TCP packets. Since the kernel is already executing the TCP receive path on every packet arrival, adding message extraction and scheduling logic at this point incurs minimal overhead, and avoids building a separate userspace message-processing pipeline with its own threads, queues, and synchronization; *(ii)* messages must be decoupled from their underlying TCP connections to allow global scheduling. Exposing a connection-agnostic, message-oriented API to userspace enables true work conservation and concurrent RPC processing across all connections.

Kernel Connection Multiplexor [37] (KCM), the first upstream Linux attempt to expose a message API over TCP, embraces the first insight but violates the second: messages remain tied to their TCP connection and are scheduled locally. As a result, it falls short for production settings: it suffers from inter-connection HOL blocking, scales poorly, and is not suitable for high-performance RPCs.

Following both insights, Rakaia provides high-performance, work-conserving scheduling by introducing three mechanisms: *(i)* Rakaia attaches a message parser to each TCP connection to extract discrete messages directly from their incoming streams, avoiding application-level demultiplexing; *(ii)* Rakaia introduces a logically centralized, message-granular request queue that decouples message scheduling from individual connections, eliminating HOL blocking both within and across connections; and *(iii)* Rakaia uses a lightweight delegation mechanism on the transmission path to avoid contention when multiple threads try to send messages via a single TCP socket, efficiently integrating with the TCP/IP stack.

Our implementation of Rakaia on Linux kernel v6.8 targets easy deployments and upstream integration: the core functionality of Rakaia is implemented as a standalone, dynamically loadable Linux kernel module of ∼3000 LoC. Rakaia does not require any core kernel changes, except for the registration of a new socket type (∼60 LoC). Rakaia supports TLS-encrypted traffic by leveraging kernel TLS (kTLS) [68].

Our evaluation shows that Rakaia delivers high-throughput RPCs with low tail latency by eliminating HOL blocking. Its performance matches the theoretical expectations and remains consistent across a wide range of workloads, including coarse-grained (100µs) and fine-grained (20µs) tasks; varying number of connections (20, 80 and 5,000); with and without TLS encryption. To assess practical impact, we modify both gRPC-Go and gRPC-C++ to use the Rakaia API and compare them against the standard POSIX-based implementations.

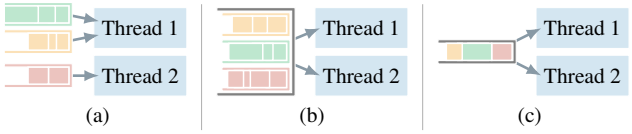


Figure 1: Three queuing models for two application threads: (a) connection-partitioned FIFO (**CP**), (b) connection-shared FIFO (**CS**), and (c) message-shared FIFO (**MS**)

Rakaia-enhanced gRPC-Go improves throughput-under-SLO by up to 1.56×, while maintaining stable performance as the number of connections scales. Rakaia also improves gRPC-C++ throughput-under-SLO by up to 2.69× across the evaluated APIs. Finally, Rakaia improves realistic gRPC workloads, increasing throughput-under-SLO by up to 1.39× for Silo [67], an in-memory database, and by 1.42× for OpenTelemetry Collector [54].

2 Motivation and prior work

2.1 The case for layering on top of TCP

The most popular RPC protocols [5, 21, 48, 65] almost universally rely on TCP for transport and TLS for security. TLS and PKI offer end-to-end confidentiality and integrity, while TCP remains the protocol of choice for its ubiquitous kernel support and its reliability in deployment.

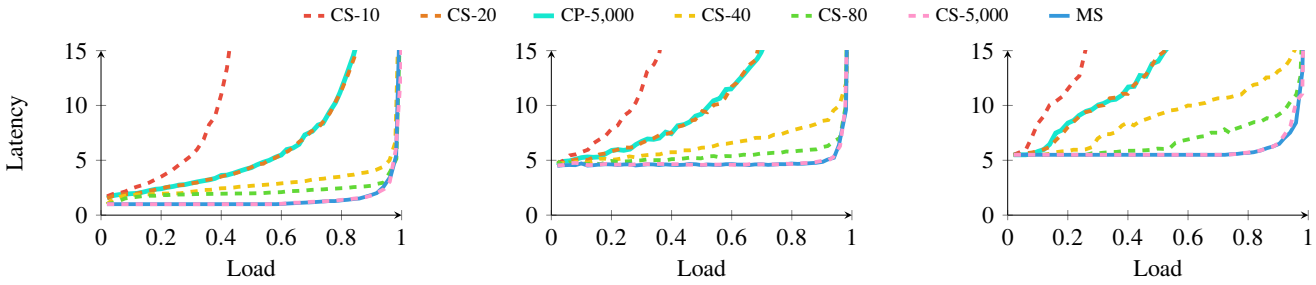
Despite decades of optimizations in the Linux TCP/IP stack [62, 64, 66], the nature of TCP continues to hinder RPC frameworks. The connection-oriented POSIX TCP API introduces two forms of head-of-line (HOL) blocking: *(i)* inter-connection HOL blocking when traffic is unevenly distributed across TCP connections; *(ii)* intra-connection HOL blocking, where a slow RPC delays all subsequent ones on the same connection. Both forms of HOL blocking lead to work imbalance, degrading tail latency and throughput.

These issues are inherent to the connection abstraction of TCP. To mitigate them, userlevel frameworks commonly bring additional layers of abstraction to break TCP's sequential stream semantics and enable concurrent RPC processing within and across connections, a process we detail in §2.3.

2.2 A quick queuing-theoretical detour

To understand the implications of delivering RPCs over TCP streams, we simulate the following queuing-theoretical models of an RPC server (Figure 1).

Connection-partitioned FIFO (CP): the server statically partitions connections across threads, leveraging RSS [62]. This approach minimizes synchronization overheads among threads. However, since connections are statically assigned to threads, this model suffers from both inter- and intra-connection HOL blocking and is not work-conserving.



(a) Deterministic (b) Exponential (c) Bimodal

Figure 2: Simulation results for the 99^{th} percentile tail latency for three service time distributions on a 20-core server.

Connection-shared FIFO (CS): the server places all authenticated connections into a single pool. Any idle thread can poll any connection in this pool, which in turn requires application-level synchronization. This improves load balancing across threads, but the connection remains the unit of ownership during request processing. At any time, only one thread can process messages from a given connection, even if other threads are idle. While CS eliminates inter-connection HOL blocking, it still suffers from intra-connection HOL blocking. Therefore, it is also not work-conserving.

Message-shared FIFO (MS): the server explicitly extracts requests from connections, enqueues them in a shared queue, and assigns requests to idle threads for processing. This model makes scheduling independent of connection ownership, allowing any idle worker to process the next available message. It eliminates both inter- and intra-connection HOL blocking at the cost of an intermediate extraction and queuing stage, typically implemented by a dedicated thread or thread pool.

We use a discrete-event simulator to model an open-loop workload generator with a Poisson arrival process and zero system overheads. This lets us isolate the queuing effects of each scheduling model from implementation overheads such as context switching. Generated requests are distributed across a varying number of connections. Figure 2 illustrates the impact on 99^{th} percentile tail latency, for three well-known service distributions (see §5.1 for details), and for different numbers of simulated client connections (*e.g.,* CS-40 denotes the CS model with 40 connections). The results show that *(i)* the connection-partitioned (CP) model fails to balance load, even with a large number of connections (CP-5,000); *(ii)* the connection-shared (CS) model suffers from intra-connection HOL blocking; *(iii)* yet, the severity of HOL blocking in the CS model depends strongly on the number of connections, *e.g.,* CS-5,000 nearly matches the performance of **MS**.

We derive two insights that inform our system design: *(i)* eliminating both intra- and inter-connection HOL blocking through a shared message queue (MS) reduces tail latency; and yet, *(ii)* larger connection-to-core ratios (*e.g.,* CS-5,000) reduce the impact of HOL blocking. Therefore, the throughput overheads of a centralized MS model implementation must remain low for the solution to be acceptable.

2.3 gRPC and HTTP/2

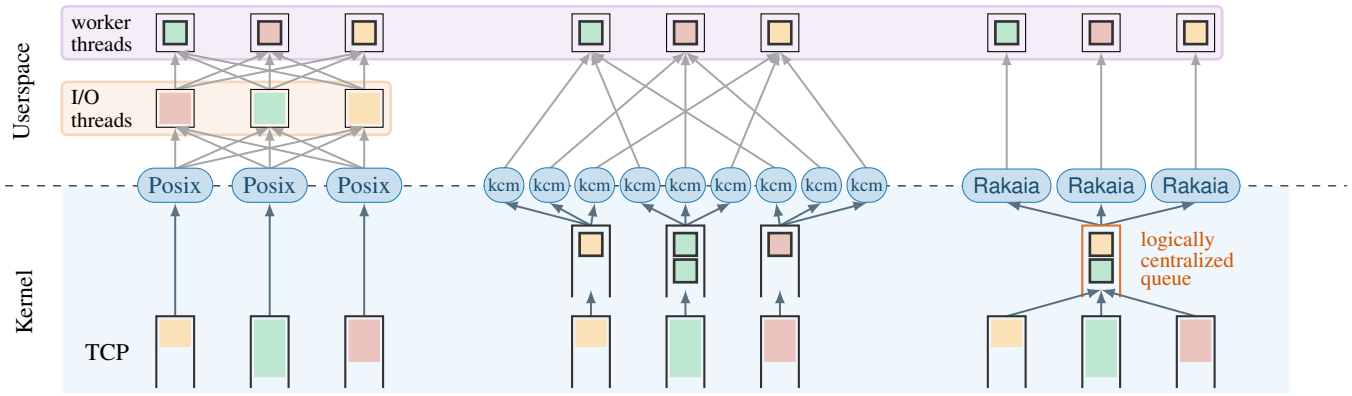
Figure 3(a) depicts the high-level architecture of gRPC, a popular userspace RPC stack that approximates the **MS** model. To achieve this, gRPC uses dedicated I/O threads to read TCP streams, reassemble RPC messages, and enqueue them into work queues, from which worker threads dequeue and execute RPCs. This design effectively decouples messages from their originating connections and distributes them to worker threads in a global FIFO manner, providing the logically centralized scheduling assumed in the **MS** model. gRPC is additionally built on top of HTTP/2 [5], which allows concurrent RPCs to interleave on a single connection, and supports out-of-order replies, thereby removing protocol-level HOL blocking.

However, achieving this concurrent RPC processing entirely in userspace is costly. gRPC must reconstruct message semantics through a layered pipeline of I/O threads, work queues, and worker thread pools. This introduces substantial inter-thread communication, synchronization, and context-switching overhead. Even the most performant gRPC implementation, Go-based gRPC [24], which relies on lightweight goroutines to mitigate these costs, experiences scalability challenges: as the number of goroutines grows, throughput eventually suffers (§5.3).

2.4 KCM – the previous in-kernel attempt

Shown in Figure 3(b), KCM [37] is the first in-kernel attempt to expose a message API on top of TCP. It relies on `strparser`, a Linux kernel facility for detecting message boundaries in TCP streams, to parse messages directly in the kernel. For each TCP connection, KCM then creates a separate KCM socket for every application thread and delivers each parsed message to one of these sockets. Replies are routed back along the same path to the underlying TCP connection.

While this avoids the inter-thread communication required by userspace frameworks like gRPC, it falls short of providing a true **MS** model. Scheduling remains fundamentally connection-driven rather than message-driven: each TCP socket independently selects among its own KCM sockets with no global view across connections or threads. Consequently, KCM suffers from inter-connection HOL blocking,



(a) gRPC (b) KCM (c) Rakaia

Figure 3: Three RPC-oriented architectures on top of TCP that aim to eliminate HOL blocking (receive side).

where idle threads cannot pull work from busy connections.

KCM scales poorly. With T threads and C connections, it creates $T \times C$ KCM sockets, a quadratic blow-up that quickly becomes impractical for high-connection services [26].

Combined with its poor performance (§5.2) and current lack of support for kTLS or userspace TLS libraries (*e.g.*, `openssl`), KCM is not considered a viable foundation for high-performance RPCs.

To summarize, high-performance RPCs built on top of TCP require a framework that approximates the **MS** model with minimal overhead. Existing userspace frameworks incur substantial overheads due to their threading and I/O models. While KCM is the only prior in-kernel attempt to expose message boundaries over TCP, it fundamentally retains the connection as the primary scheduling abstraction, making message-level work conservation impossible. In the next section, we describe how our design, Rakaia (Figure 3(c)), addresses these limitations by enabling low-overhead, message-oriented scheduling directly in the kernel.

3 The Design of Rakaia

Rakaia's design is driven by two key insights. First, message parsing and scheduling can be performed directly in the kernel's TCP receive path, in the same `softirq` context that processes incoming packets. Since the kernel is already executing to handle each packet, delineating messages and making scheduling decisions at this point adds minimal additional overhead. Second, decoupling messages from their TCP connections as early as in the kernel enables message-level scheduling and the exposure of a purely message-oriented API to applications. Together, these insights allow Rakaia to approximate the **MS** model with low overhead (through a logically centralized scheduler), while remaining fully compatible with the unmodified Linux TCP/IP stack.

At a high level, we set the following goals for Rakaia:

- **Message-level work conservation:** Rakaia schedules at the level of messages instead of per connection, ensuring no core stays idle while work exists.
- **Performance:** Rakaia outperforms existing TCP-based RPC frameworks in terms of throughput and tail latency.
- **Scalability:** Rakaia scales with an increasing number of TCP connections.
- **Compatibility:** Rakaia is fully compatible with the unmodified Linux TCP/IP stack and transport-level security mechanisms like kTLS. Any updates or optimizations in the underlying stack are automatically inherited by Rakaia.

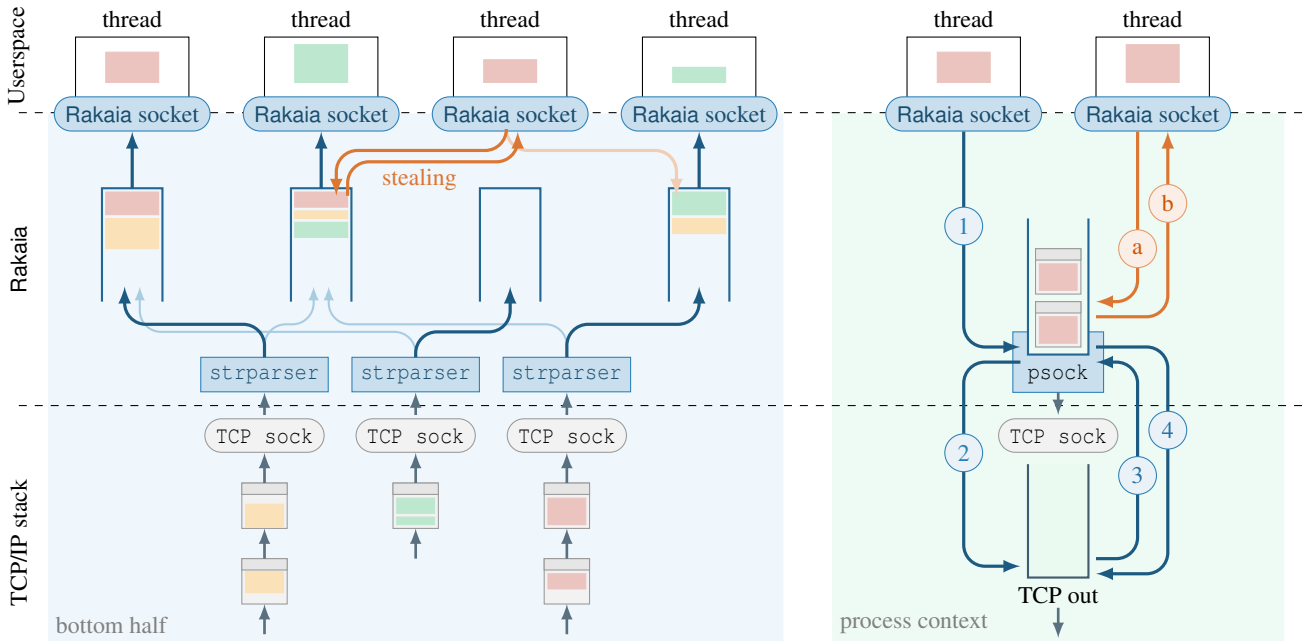
3.1 Overview

Figure 4 shows the high-level design of Rakaia, a lightweight in-kernel layer for message processing, demultiplexing, and scheduling atop the TCP/IP stack. Rakaia exposes a message-granular API to userspace (§3.2) and is fully compatible with TLS (§3.5). Rakaia operates in three layers:

Hook with the lower TCP/IP stack: Rakaia leaves the Linux TCP/IP stack unmodified and leverages `strparser`, which also underpins KCM, to extract message boundaries directly from the TCP byte stream. We extend `strparser` beyond length-delimited protocols to support HTTP/2. A protocol-specific parser is attached to each TCP socket and incrementally reconstructs complete messages from the receive queue before exposing them to the upper layers.

The scheduling layer: Once a message is parsed, it enters Rakaia's logically centralized but physically distributed scheduler, which maintains one FIFO queue per core. Any message may be assigned to any queue: Rakaia uses a power-of-two-choice (P2C) policy [50] to sample two queues and place the message on the shorter one, balancing load with constant overhead. Workers dequeue from their local queue first and invoke stealing if it is empty, preserving work-conserving scheduling without the scalability bottleneck of a global queue.

Message reception and transmission: Rakaia exposes a single message-oriented socket per application thread. On reception, the Rakaia socket delivers complete messages from



(a) Receiving Path

(b) Transmission Path

Figure 4: Overview of Rakaia's design. In the receiving path, solid arrows ( ) show the chosen execution path, while faded arrows ( ) indicate unselected candidates under power-of-two choices. In the transmission path, (1) - (4) shows a thread sending messages along the fast path, while (a) - (b) illustrates a case where message sending is delegated.

the scheduler to userspace. After the application processes messages, this socket transmits the responses over their associated TCP connections. When multiple threads attempt to send over the same TCP connection concurrently, Rakaia resolves contention using a lightweight delegation mechanism. After completing a transmission, a worker immediately fetches the next message or transitions to idle.

3.2 Rakaia API

It is well known that POSIX was not designed for scalability or message-oriented communication [12]. Further, RPC frameworks for TCP typically rely on dedicated I/O threads to demultiplex traffic, thereby introducing significant scheduling, queuing and synchronization overheads.

Rakaia simplifies this architecture by exposing a message-oriented, connection-oblivious socket API. Each worker thread communicates with the kernel through a single socket that directly sends and receives discrete messages over any TCP connection. Because all transport state remains in the TCP stack, the Rakaia socket maintains no per-connection bookkeeping. This design removes the need for I/O threads and provides a lightweight and scalable RPC interface.

At the API level, Rakaia was initially inspired by Homa/Linux, a Linux kernel implementation of the Homa transport protocol [51, 56]. Both Rakaia and Homa/Linux expose a single socket for sending and receiving discrete messages to and from arbitrary peers, a common practice in message-oriented communication, where applications exchange discrete messages with remote endpoints, *e.g.,* the POSIX UDP socket semantics. Unlike UDP and Homa, whose sockets directly communicate with remote peers, Rakaia's socket abstraction is built entirely atop TCP and serves only as a kernel-facing interface. It therefore does not maintain any remote-peer state, making it a lightweight layer over the underlying TCP stack.

Table 1 shows the APIs provided to the applications by Rakaia. We introduce a new protocol family `AF_RAKAIA` for the Rakaia socket. Each worker thread creates a Rakaia socket using the standard `socket` system call with `AF_RAKAIA`, and a custom protocol identifier `RAKAIAPROTO_CONNECTED`. Since each application thread only needs a single Rakaia socket to handle all TCP traffic, this design reduces resource consumption and simplifies file descriptor management. For instance, when using `epoll`, a thread needs to monitor only one file descriptor instead of one for each individual TCP connection.

Rakaia introduces a single new control operation: `rakaia_attach`, exposed as an `ioctl` on the Rakaia socket. The `rakaia_attach` call associates an existing TCP connection with the Rakaia framework and selects which built-in message parser should be used via `proto_id`. Rakaia currently provides parsers for `Memcached` and `HTTP/2`-based protocols (*e.g.,* gRPC). It can easily be extended to support additional ULP protocols. Once a connection is attached, Rakaia

Table 1: The API provided by Rakaia for applications.

Function	Description
`socket(AF_RAKAIA, SOCK_DGRAM, RAKAIAPROTO_CONNECTED)`	Creates a Rakaia socket
`close(int rakaia_fd)`	Closes the Rakaia socket
`rakaia_attach(int rakaia_fd, int csock, int proto_id)`	Attaches a TCP socket to Rakaia and selects the built-in message parser (e.g., Memcached or gRPC).
`recvmsg(int rakaia_fd, struct msghdr *msg, int flags)`	Receives a complete message from the Rakaia socket
`sendmsg(int rakaia_fd, const struct msghdr *msg, int flags)`	Sends a complete message through the Rakaia socket

begins to reconstruct incoming messages for that connection and makes parsed messages available to worker threads. Although the file descriptor of a Rakaia socket is passed to the kernel when invoking `rakaia_attach`, it serves merely as an entry point and does not create a persistent association between that Rakaia socket and the attached TCP connection. Message reception and transmission continue to use the standard `recvmsg` and `sendmsg` calls on the Rakaia socket.

3.3 Receive Path

Figure 4(a) shows the three main tasks of the receiving path: *(i)* parsing TCP byte streams into discrete messages, *(ii)* completely decoupling messages from their original connection, and *(iii)* work-conserving scheduling across Rakaia sockets.

Connection setup and message parsing: When a TCP connection is registered via the `rakaia_attach` system call, Rakaia initializes a `rakaia_psock` data structure to wrap the target TCP socket. This wrapper serves two primary purposes. First, it holds a new `strparser` instance for message parsing. Second, it stores the necessary fields to support the transmission path, which we describe in detail in §3.4.

On the RX path, Rakaia reconstructs complete application messages directly from the TCP byte stream. As illustrated in Figure 5, message parsing differs between supported protocols. For simple length-delimited protocols like Memcached, parsing only consists of inspecting a fixed-size header and extracting the encoded payload length. For Memcached, Rakaia uses the binary protocol's `opaque` header field to match responses to requests [49,58]. In HTTP/2-based protocols, each RPC is mapped to a single HTTP/2 stream, which in turn consists of one or more frames. The parser reads each frame's header, extracting its stream ID, length, type, and flags. Each complete frame is then appended to a per-stream buffer maintained in a hash table, which enables Rakaia to assemble messages that span multiple frames. When a frame carrying the `END_STREAM` flag arrives, the buffered frames for that stream are coalesced into a complete message and passed to the upper layer. Note that control frames such as `PING` or `WINDOW_UPDATE` are handled entirely in-kernel, allowing immediate flow control updates and acknowledgments.

Message insertion and origin tracking: Once a message has been reconstructed, Rakaia embeds a pointer to the corresponding `rakaia_psock` into the first `skb` of the message before enqueuing it into the central message queue. When a Rakaia socket dequeues this message and copies it to userspace, it extracts and caches the `rakaia_psock` pointer internally. This allows the subsequent `sendmsg()` call on the same Rakaia socket to access the correct underlying TCP connection, without revealing connection information to userspace.

Work-conserving scheduling: Rakaia presents a logically centralized message queue to applications while physically distributing scheduling across per-socket receive queues. Each Rakaia socket maintains a local FIFO queue and primarily serves messages from this queue. When a new message becomes ready, Rakaia assigns it to a socket using P2C load balancing: it randomly samples two candidate sockets and enqueues the message into the one with the shorter local queue. If the selected socket's queue is empty, indicating that at least one worker is idle, the message is instead delivered immediately via direct handoff. When a socket exhausts its local queue, it attempts to steal work from another active socket. To do so efficiently, Rakaia maintains a bitmap that tracks non-empty queues and again applies P2C to select a victim among them. If no work exists anywhere in the system, the socket enters idle state until new messages arrive. This design preserves work conservation across all load regimes. Under light load, messages are dispatched immediately via direct handoff. Under heavy load, queues naturally build up and sockets dynamically rebalance load through work stealing, approximating a centralized queue without its scalability bottleneck. Moreover, because Rakaia uses P2C for message-level distribution, its load balance is insensitive to the number of TCP connections: even with a small number of connections, P2C ensures tightly bounded queue lengths and maintains near-uniform load across sockets [50].

3.4 Transmit Path

The transmission path of Rakaia consists of two major stages: *(i)* copying the message into the kernel, and *(ii)* transmitting it over TCP with minimal lock contention.

Independent message copying: On `sendmsg`, Rakaia copies the message (and its associated TCP socket identifiers) into

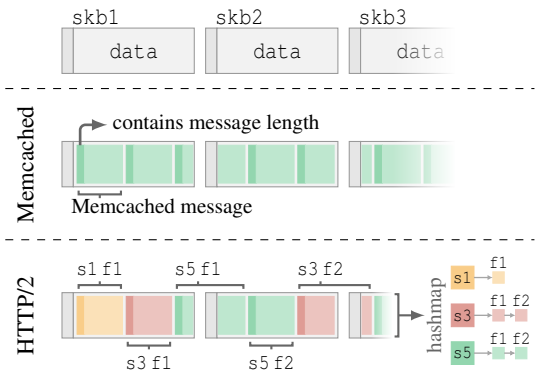


Figure 5: Parsing Memcached and HTTP/2. `s1`, `s3`, and `s5` refer to stream with ID 1, 3, and 5. `f1` and `f2` refer to the first and second frames of a stream.

the kernel. Rakaia copies each message independently into its newly allocated `skb`. A natural alternative is to batch messages destined for the same TCP socket into a shared sending queue so they can be packed into a single `skb`. While this may reduce `skb` allocation overhead, it introduces extra latency, substantial synchronization complexity, and difficult corner cases. In particular, under memory pressure, partial copy failures make it hard to free an `skb` that contains multiple messages. Our design avoids these pitfalls by copying each message independently, ensuring that `skb`s can be created and freed cleanly even in failure paths.

Direct or delegated transmission: Before message copying, Rakaia retrieves the cached `rakaia_psock` pointer (recorded by `recvmsg`) to identify the corresponding TCP socket. At this point, Rakaia does not directly enqueue messages into the TCP socket's `sk_write_queue`, since doing so requires acquiring the TCP socket lock, a mutex-like lock that would serialize all concurrent senders and put some to sleep.

Rakaia therefore employs a two-phase delegation mechanism for transmission, illustrated in Figure 4(b) and detailed in Algorithm 1. For each TCP socket, the `rakaia_psock` wrapper structure maintains a lightweight virtual sending queue (`virt_queue`) for temporary message accumulation. Access to that queue is shown as ① in Figure 4(b). This queue is merely a staging buffer: it accumulates messages without touching TCP's socket lock. Rakaia checks a boolean flag within the `rakaia_psock` to determine whether another thread is currently handling message transmission. If the flag is set, the sender returns to userspace immediately without further action (ⓑ). Otherwise, it sets the flag and becomes the designated transmitter: it acquires the TCP socket lock, moves all staged messages from `virt_queue` into `sk_write_queue`, and initiates message transmission via the standard TCP stack by calling `tcp_push` (②).

To avoid leaving behind messages that are concurrently enqueued by other threads to `virt_queue`, the designated transmitter repeatedly rechecks `virt_queue` (③-④) and flushes any newly arrived messages before relinquishing the TCP socket lock. When the queue is observed empty, it clears the flag and then releases the TCP socket lock, allowing future senders to proceed.

Algorithm 1 Delegation mechanism in the transmission path

```
1:  function SendMessage(psock, msg)
2:      set csk ← psock.sk                    ▷ associated TCP socket
3:      Enqueue(psock.virt_sndbuf, msg)
4:      if psock.tx_in_use is true then
5:          return
6:      end if
7:      set psock.tx_in_use ← true
8:      lock_sock(csk)
9:      while true do
10:         if psock.virt_sndbuf is empty then
11:             set psock.tx_in_use ← false
12:             break
13:         end if
14:         RequeueAll(psock.virt_sndbuf, csk.sndbuf)
15:         sk_flush_backlog(csk)             ▷ process rx backlog
16:         ...                               ▷ Omitted TCP stack logic
17:         tcp_push(csk)                     ▷ send out messages
18:     end while
19:     release_sock(csk)
20: end function
```

3.5 Composing with TLS

Rakaia's in-kernel message parsing relies on visibility into plaintext TCP streams, and the design described so far assumes unencrypted traffic. At first glance, this may suggest that Rakaia cannot handle encrypted traffic. However, Linux provides built-in kernel TLS (kTLS), which enables in-kernel TLS encryption and decryption. In kTLS, the TLS handshake is performed in userspace, typically right after accepting a TCP connection. Once the handshake completes, applications share the session key with the kernel, offloading TLS record processing to kTLS. This mechanism allows Rakaia to operate over TLS connections, preserving end-to-end confidentiality.

Figure 6 shows how Rakaia integrates with kTLS. In the receiving path, when a TCP socket receives encrypted TLS traffic, kTLS intercepts the data and parses it into discrete TLS records. These records are handed directly to the `strparser` attached to the TCP socket. The `strparser` then queries the kTLS decryption module to obtain the plaintext bytes and invokes its parsing program to extract application-level messages. Processing then proceeds as in §3.3.

The transmission side is simpler. Rakaia hands plaintext messages to the kTLS encryption module, which encrypts and transmits them via the corresponding TCP socket.

Note that kTLS commonly performs decryption in process context, typically when `recvmsg` is called. Although kTLS

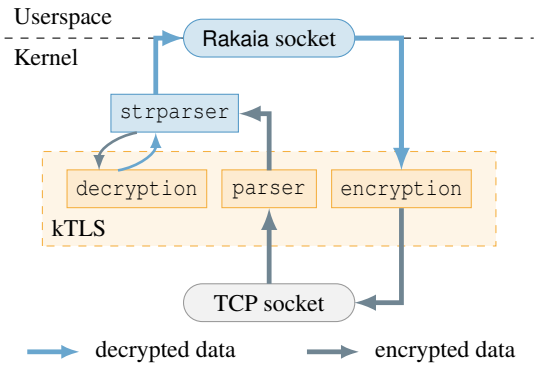


Figure 6: Rakaia running atop kTLS

exposes an interface for decryption in `softirq`, its current implementation is not robust and suffers from contention. To avoid this, Rakaia invokes the kTLS decryption module from a Linux workqueue [44], ensuring that decryption runs in an appropriate context. We plan to patch kTLS to support reliable `softirq` decryption, enabling tighter integration with Rakaia.

4 Implementation

We implemented Rakaia as a dynamically loadable kernel module on Linux version `v6.8`, consisting of approximately 3,000 lines of C code. In addition, we provide a kernel patch of around 60 lines to register the new Rakaia socket type. This patch makes minimal modifications to the existing kernel codebase and does not alter the TCP stack semantics.

Extending `strparser`: We extend `strparser` to support HTTP/2 stream parsing. In the default `strparser`, when a single message spans multiple skbs, the parser links them by using the head skb's `frag_list` and the `next` pointers of subsequent skbs. HTTP/2, however, introduces an additional layer of structure: messages are formed from multiple *frames*, and frames belonging to the same stream may arrive in separate skbs. To correctly assemble full HTTP/2 streams, Rakaia retains the original linking mechanism for individual frames and then uses reserved space in `skb->cb` to maintain per-stream linkage. This produces a single skb chain per stream, which the kernel can deliver as a complete logical message.

Offloading HTTP/2 transport logic: Supporting gRPC requires more than extending `strparser` to extract HTTP/2 messages. HTTP/2 introduces a substantial control plane (*e.g.,* flow control, PING frames) that would normally be handled in userspace. To fully eliminate transport-layer work from the application runtime, Rakaia offloads these mechanisms into the kernel as well. During parsing, Rakaia identifies HTTP/2 control frames and processes them directly (*e.g.,* applying and generating window updates), rather than delivering them to userspace. This not only simplifies the userspace logic, but also makes responding to control events far cheaper, since no syscalls or context switches are needed. Rakaia supports all HTTP/2 control-plane features used by gRPC.

Custom TCP path until `tcp_push`: While we keep the semantics of the TCP/IP stack intact, we cannot simply invoke the normal `tcp_sendmsg` in the transmission path after copying messages into the kernel. This is because `tcp_sendmsg` contains the logic for TCP socket lock acquisition, `skb` allocation, and data copying, all of which are unnecessary for Rakaia. To allow efficient transmission, we move the TCP stack logic such as sequence number initialization and rx backlog processing directly into `rakaia_sendmsg`. Afterwards, we call the existing function `tcp_push`, which is provided by the kernel for pushing `skb`s onto the network.

5 Evaluation

We evaluate Rakaia's performance and compare it against a range of RPC frameworks to answer the following questions:

- Does Rakaia behave according to the theoretical expectations of the **MS** model across various service-time regimes and levels of connection scaling? (§5.2)
- Does Rakaia bring performance gains to gRPC? (§5.3)
- Can Rakaia integrate efficiently with TLS, in comparison with other approaches? (§5.4)
- Does Rakaia deliver benefits for real-world RPC workloads beyond synthetic benchmarks? (§5.5)

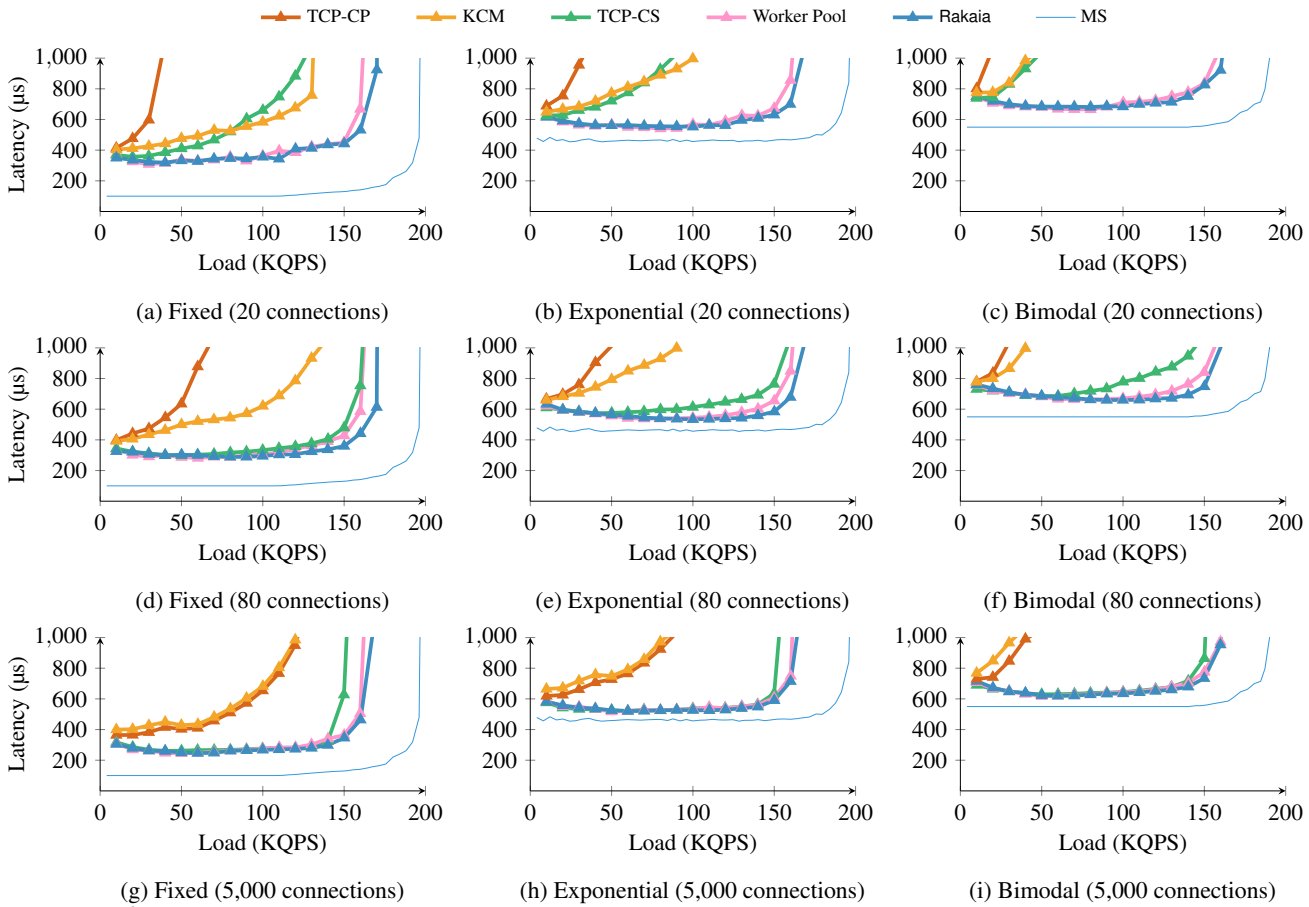
5.1 Experimental setup and methodology

Hardware: All experiments are conducted using Cloudlab [16] xl170 nodes. Each machine is equipped with one Intel Xeon E5-2640v4 CPU (10 cores, 2.4 GHz). Hyperthreading is enabled. The machines run an Ubuntu LTS 24.04 distribution with the Linux kernel version `6.8`.

In all experiments, the server is set up to use 20 hyperthreads. For most experiments, this implies a 20-way multithreaded application. For gRPC-Go, the server is set up with 20 worker goroutines for both gRPC-POSIX and gRPC-Rakaia. All experiments are compute-bound.

Experimental setup: In all experiments, we report the 99^{th} percentile tail latency (y-axis) versus the achieved load in RPCs per second (x-axis), both measured at the client when responses are received. We use an extended version of Lancet [39] as the load generator. Lancet first opens all connections at once, then issues requests according to a Poisson process in an open-loop fashion. Lancet supports multiple RPC protocols, including Memcached [48] and gRPC, with optional TLS support.

Synthetic distributions: For our synthetic microbenchmarks, clients send requests, encoded using a specified RPC protocol, to the server that executes them. The service times for these



(a) Fixed (20 connections) (b) Exponential (20 connections) (c) Bimodal (20 connections)

(d) Fixed (80 connections) (e) Exponential (80 connections) (f) Bimodal (80 connections)

(g) Fixed (5,000 connections) (h) Exponential (5,000 connections) (i) Bimodal (5,000 connections)

Figure 7: 99^{th} percentile latency as a function of throughput for 100µs tasks. Top, middle, and bottom rows correspond to 20, 80, and 5,000 client connections, respectively.

requests follow one of three well-known distributions from the literature [14, 32, 38, 42, 45–47, 61]:

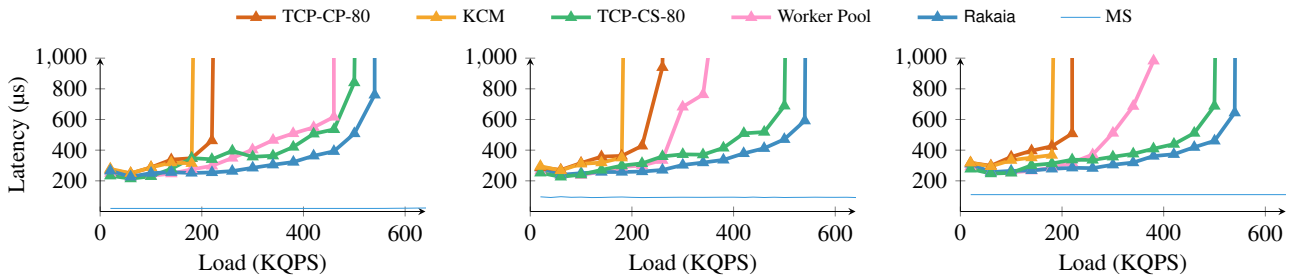
- deterministic: $P[X = \bar{S}] = 1$
- exponential with mean service time $\bar{S}$
- bimodal: $P[X = \bar{S}/2] = .9$; $P[X = 5.5 \times \bar{S}] = .1$

Comparison: We compare the following systems:

1. **TCP-CP**, a simple lock-free `libevent`-based program where connections are split among threads (as directed by RSS [62]). Each thread both processes the requests and sends back the replies, ensuring natural in-order delivery.
2. **TCP-CS**, a simple connection-sharing, `libevent`-based program where threads temporarily lock access to a TCP socket to process requests and then return replies in-order.
3. **Worker Pool**, a userspace design that implements the **MS** model. It uses a small number of I/O threads and a pool of worker threads. The I/O threads block on a shared `epoll` instance and reconstruct messages from TCP streams. Once a complete request is received, the I/O thread enqueues it into a shared lock-free ring buffer, from which a worker thread dequeues it for processing. After processing the request, the worker attaches the response to the corresponding connection and marks the connection as having pending output. If the worker can acquire the I/O ownership of the connection, it flushes the response directly. Otherwise, it arms the connection with `EPOLLOUT` so an I/O thread can flush it once the socket is writable.
4. **KCM**, as available natively in the Linux kernel. We equip KCM with the same parsing program for extracting RPC messages and delivering them to userspace.
5. **gRPC-POSIX**, the official implementations. We evaluate both gRPC-Go (v1.75.0) and gRPC-C++ (v1.78.1).
6. **Rakaia**, our contribution, implemented as a kernel module.
7. **gRPC-Rakaia**, our contribution, modified gRPC-Go and gRPC-C++ implementations running on top of Rakaia.

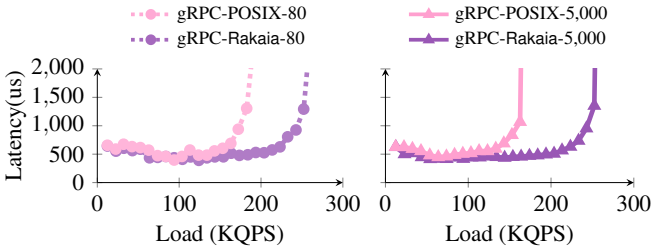
5.2 Rakaia follows the MS model

We conduct the first set of experiments with varying TCP connections to the server, TLS disabled, and an open-loop load generator issuing as many outstanding requests per connection as needed. We measure the latency of replies at clients. For the Worker Pool, we sweep the number of I/O threads and

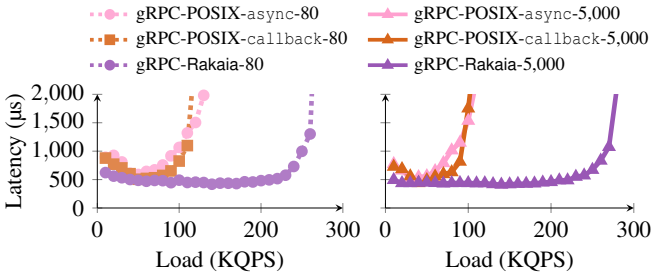


(a) Fixed (80 connections). (b) Exponential (80 connections). (c) Bimodal (80 connections).

Figure 8: 99^{th} percentile latency as a function of throughput for 20µs tasks with 80 client connections.



(a) gRPC-Go



(b) gRPC-C++

Figure 9: 99^{th} percentile latency according to throughput for gRPC-POSIX and gRPC-Rakaia (Exponential ($\bar{S} = 20$µs)).

Table 2: CPU time breakdown for gRPC-POSIX and gRPC-Rakaia, at 170 KQPS on a server with 20 hardware threads.

Config	# goroutines	%I	%K	%U	%sched	%mstack
gRPC-POSIX-80	1,077	7.5	33.0	54.6	7.6	2.1
gRPC-POSIX-5,000	16,946	4.1	32.2	60.0	5.5	12.3
gRPC-Rakaia-80	29	22.9	33.2	38.0	0.8	0.1
gRPC-Rakaia-5,000	33	23.5	34.0	37.3	1.0	0.1

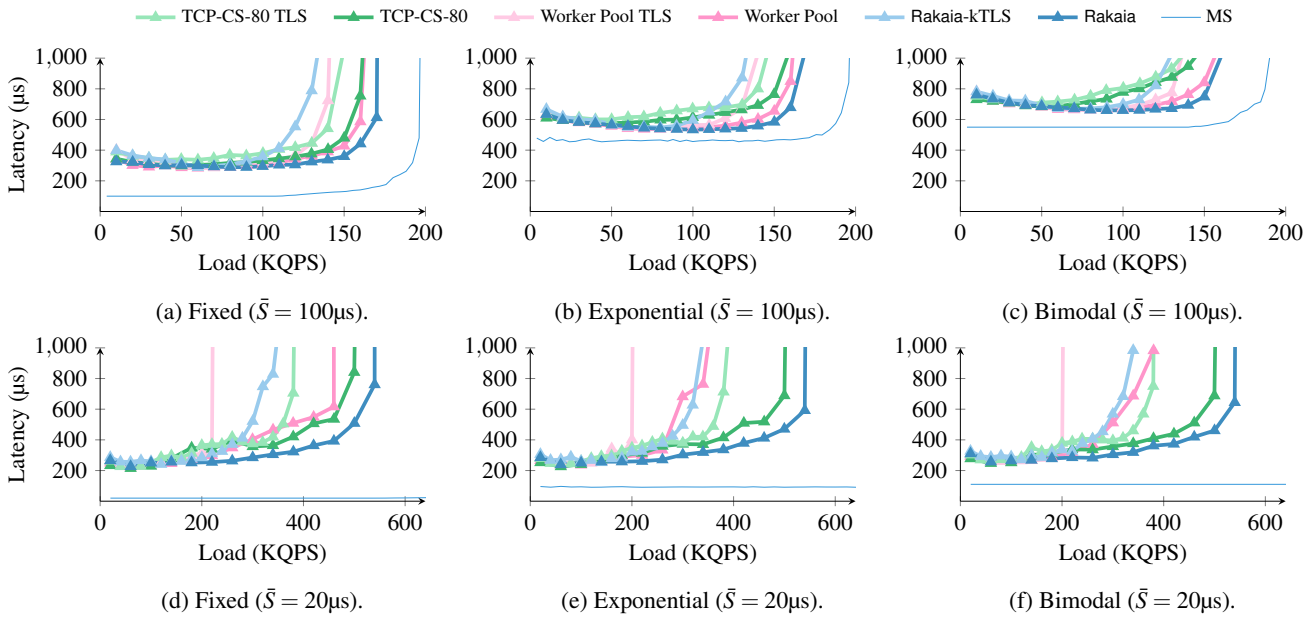
worker threads and use the best-performing configuration for each service time.

Figure 7 shows the 99^{th} percentile latency for 100µs service time across three different numbers of client connections (20, 80, and 5,000), following the three distributions described in §5.1. For reference, we plot the theoretical **MS** curve for an $M/G/20$ queue, scaled by the service time and assuming no CPU or network overheads.

Connection Scaling: Figure 7 shows that Rakaia remains close to the **MS** model across all tested connection counts. With only 20 connections, Rakaia sustains about 160 KQPS throughput across the three service-time distributions, whereas TCP-CS reaches only about 40 KQPS in the bimodal case. Thus, when intra-connection HOL blocking is most severe, Rakaia delivers roughly 4× the throughput of TCP-CS. The Worker Pool tracks Rakaia more closely than other baselines and exhibits the same overall scaling trend, showing that a userspace design can approximate **MS**-like behavior, at the 100µs scale.

As the number of connections increases to 80 and then 5,000, TCP-CS improves substantially because the requests are spread across more connections, reducing intra-connection HOL blocking. This trend is in line with the simulations in Figure 2. Nevertheless, even with 5,000 connections, Rakaia still consistently outperforms TCP-CS by 7%, though they are expected to behave almost identically in theory. Likewise, Rakaia also retains a small advantage over the Worker Pool. In both cases, the additional performance comes from Rakaia's in-kernel message scheduling, which avoids the userspace locking and coordination overheads among application threads, required by both TCP-CS and the Worker Pool. All other approaches (KCM and TCP-CP) are not competitive because they suffer from HOL blocking, *e.g.,* KCM only achieves around 30 KQPS in the bimodal case, 19% of Rakaia's throughput.

Shorter tasks: Figure 8 evaluates 20µs tasks with 80 client connections. At this granularity, system overheads account for a much larger fraction of end-to-end latency, making implementation costs more visible. The Worker Pool is particularly sensitive and suffers from significant performance degradation. We tune the I/O-thread and worker-thread configuration for the fixed service-time case, where it performs competitively. However, even with the same configuration, its performance degrades under the exponential and bimodal distributions, where higher service-time variability causes earlier tail-latency growth. KCM introduces high system overheads and performs worse than the vanilla TCP-CP and TCP-CS models, despite its in-kernel message parsing advantages. In contrast, Rakaia has low system overheads, and consistently maintains its advantages over TCP-CS.



(a) Fixed ($\bar{S} = 100\mu s$). (b) Exponential ($\bar{S} = 100\mu s$). (c) Bimodal ($\bar{S} = 100\mu s$).

(d) Fixed ($\bar{S} = 20\mu s$). (e) Exponential ($\bar{S} = 20\mu s$). (f) Bimodal ($\bar{S} = 20\mu s$).

Figure 10: 99^{th} percentile latency as a function of throughput with TLS enabled for 100µs and 20µs tasks, respectively.

5.3 gRPC over Rakaia

Figure 9 shows that Rakaia improves gRPC's performance across a range of connection counts, for both the Go and C++ gRPC stacks.

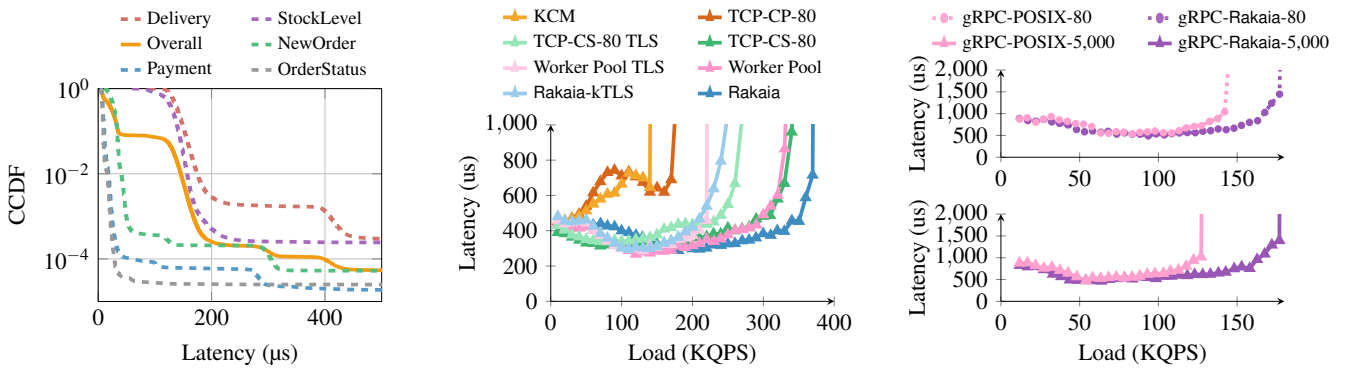
In the gRPC-Go case, with 80 connections, gRPC-Rakaia already sustains 39% higher load than gRPC-POSIX by avoiding the userspace I/O goroutines and inter-goroutine synchronization on the receive path. The benefit becomes even more pronounced as the number of connections increases. At 5,000 connections, gRPC-POSIX sustains 11% lower throughput than in the 80-connection case, suggesting that its per-connection goroutine structure begins to impose measurable scheduling overhead at scale. In contrast, gRPC-Rakaia maintains nearly identical throughput from 80 to 5,000 connections. As a result, its advantage over gRPC-POSIX grows to 1.56× at 5,000 connections.

Table 2 shows the CPU time breakdown for gRPC-POSIX and gRPC-Rakaia when serving 170 KQPS. **%I**, **%K**, and **%U** denote idle, kernel, and userspace time, respectively. **%sched** captures time spent in the Go runtime scheduler, while **%mstack** accounts for time spent on Go stack growth and management (*i.e.,* `morestack`). With POSIX sockets, gRPC relies on a large number of goroutines to emulate the **MS** model, scaling from 1,077 goroutines at 80 connections to around 17,000 at 5,000 connections. This explosion in concurrency significantly increases runtime overhead, reflected in high scheduler activity (6-8%) and substantial Go stack management overhead (up to 12.3%). In contrast, gRPC-Rakaia requires only a few dozen goroutines (20 workers plus a small number of runtime goroutines) regardless of connection numbers, as message multiplexing and scheduling are handled in the kernel. As a result, scheduler and stack management overhead becomes negligible. At 5,000 TCP connections, the shift from userspace to the kernel increases kernel time by only 1.8 pp, while reducing userspace CPU time by 22.7 pp.

Figure 9(b) repeats the experiment with gRPC-C++, comparing the `async` [22] and `callback` [23] APIs against gRPC-Rakaia. We omit gRPC-C++'s `sync` API because it runs RPC handlers on a dynamically growing thread pool [25]. Under high concurrency, this pool can grow with the number of active RPCs, leading to poor performance. Note that gRPC-Rakaia exposes the same user-facing API as `sync`, allowing existing applications to use it as a drop-in replacement. Unlike the standard `sync` implementation, it does not inherit the scalability bottleneck of a dynamically growing thread pool. The results show that gRPC-Rakaia improves gRPC-C++ throughput by 2× and 2.33× for the `async` and `callback` APIs, respectively, at 80 connections. At 5,000 connections, the corresponding improvements are 2.69× and 2.67×, respectively. This improvement is even more pronounced than in the gRPC-Go case, because gRPC-C++'s threading model incurs higher overheads than gRPC-Go's goroutine-based design.

5.4 The impact of TLS

Figure 10 compares the performance of Rakaia, TCP-CS and the Worker Pool, with and without TLS enabled. We omit *(i)* KCM, as it does not work out-of-the-box with kTLS on recent kernels, and *(ii)* gRPC-Rakaia, as Go's standard library does not yet support kTLS. Support for kTLS in gRPC-Rakaia will become available once the recently accepted proposal to integrate kTLS into Go [20] is implemented. Enabling kTLS has a measurable impact in terms of throughput for Rakaia. This is due to the use of the Linux workqueue to perform



(a) CCDF of task execution time

(b) 99^{th} percentile latency vs. throughput

(c) 99^{th} percentile latency vs. throughput

Figure 11: Silo running TPC-C benchmark

decryption on the receive path (see §3.5), which becomes a bottleneck. This limitation is not fundamental: once kTLS supports safe decryption in `softirq` context, this bottleneck can be removed.

For 100µs tasks, Rakaia-kTLS reaches about 120–130 KQPS, close to TCP-CS-TLS and the Worker Pool TLS at about 130–140 KQPS. For 20µs tasks, TLS overhead becomes more prominent. Rakaia-kTLS still reaches about 320–340 KQPS, reasonably close to TCP-CS-TLS at about 380 KQPS, while the Worker Pool TLS drops to only about 200–220 KQPS. This drop occurs because the I/O-thread configuration for the Worker Pool is tuned for the unencrypted case. With TLS enabled, the same I/O threads must handle higher per-request processing cost, so they saturate early and cannot feed workers fast enough. With Rakaia, I/O processing scales automatically through RSS, removing this responsibility from userspace.

5.5 Real-world applications

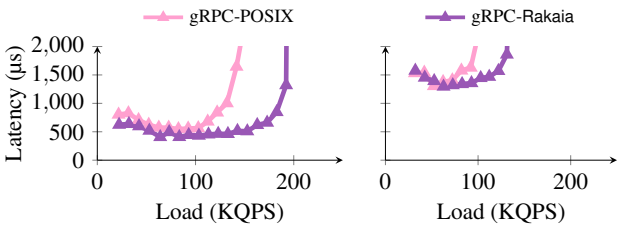
The impact on TPC-C query latency: To assess the benefits of Rakaia in the real world, we use Silo [67], an in-memory database optimized for multicore scalability. Using a methodology similar to [61], we run a networked server layer on top of Silo, which generates transactions from the TPC-C benchmark upon receiving new requests. Silo's garbage collection is disabled as it introduces large variability in 99^{th} percentile latency. Additionally, we did not implement marshalling for SQL queries and responses.

Figure 11(a) shows the multi-modal service time CCDF of the 5 TPC-C query tasks measured without queuing or network delays. The workload is substantially more complex than the synthetic distributions we used up to this point. Figure 11(b) plots tail latency as a function of achieved throughput for Rakaia (w/ and w/o TLS), TCP-CS-80 and the Worker Pool (w/ and w/o TLS), and the TCP-CP-80 and KCM baselines (w/o TLS). It shows that Rakaia's systematic elimination of HOL blocking delivers the highest throughput-under-SLO. TCP-CP-80 and KCM still collapse early because of their inferior scheduling policies. While TCP-CS-80 and the Worker Pool come much closer to Rakaia, they both still show earlier tail-latency growth as load increases. At 500µs tail latency, Rakaia sustains 350 KQPS, compared with 309 KQPS for TCP-CS-80 and 299 KQPS for the Worker Pool. This remaining gap indicates that, even when intra-connection HOL blocking is reduced, Rakaia still benefits from in-kernel message scheduling rather than paying the userspace coordination overheads required by TCP-CS and the Worker Pool.

Figure 11(c) compares Silo served through gRPC-Go over gRPC-POSIX and gRPC-Rakaia under 80 and 5,000 connections. gRPC-Rakaia improves upon gRPC-POSIX's throughput by 1.25× and 1.39×, respectively. This confirms that Rakaia's benefits extend to real-world RPC workloads, even when layered beneath a full gRPC stack.

OpenTelemetry: We also evaluate OpenTelemetry Collector [54], a widely used telemetry service for receiving, processing, and exporting traces, metrics, and logs in cloud deployments. In our benchmark, clients export traces to the Collector using the OpenTelemetry Protocol (OTLP) over gRPC-Go. This exercises the same gRPC-Go receive path as our earlier gRPC experiments, while replacing synthetic RPC payloads with realistic telemetry messages.

We evaluate two Collector configurations. The first uses a no-op exporter, which isolates the cost of processing OTLP requests without introducing backend storage or export traffic. The second uses a Jaeger-backed configuration [30], where the Collector batches incoming spans and exports them asynchronously to a Jaeger backend running on a separate machine. Following the OpenTelemetry testbed-style workload, each frontend RPC carries 10 spans, and the Collector uses the default batch size of 8192 spans with a 200 ms timeout. For the Jaeger configuration, we reserve two exporter consumers for asynchronous backend export and run gRPC-Rakaia workers on the remaining cores. Figure 12(a) shows that, with the no-op exporter, gRPC-Rakaia improves throughput-under-SLO by 1.35× over gRPC-POSIX at a 2 ms 99^{th} percentile tail-



(a) No-op (b) With Jaeger

Figure 12: 99^{th} percentile latency according to throughput for OpenTelemetry Collector with POSIX and Rakaia transports.

latency SLO, sustaining 192 KQPS compared with 142 KQPS. Because the no-op exporter removes backend work, this result isolates the benefit of accelerating the Collector's server-side gRPC. Figure 12(b) shows that this benefit persists with the Jaeger-backed configuration, where frontend RPC handling competes with batching, protobuf serialization, OTLP/gRPC export, and Jaeger ingestion. The backend export path remains an unmodified gRPC client path and is not accelerated by Rakaia. Even so, gRPC-Rakaia improves frontend throughput-under-SLO by 1.42×, sustaining 131 KQPS compared with 92 KQPS for gRPC-POSIX. These results show that Rakaia improves the performance of realistic gRPC server frontends, in addition to synthetic RPC handlers.

6 Discussion

Support for other gRPC implementations: Rakaia currently supports gRPC-Go (the most performant of the official gRPC implementations [24]) and gRPC-C++ out of the box. Extending Rakaia support of gRPC to other languages (*e.g.,* Java, Rust) is straightforward and would similarly remove the need for dedicated I/O threads and transport logic in userspace.

Compatibility with stack optimizations: Rakaia is compatible with standard kernel and NIC offloading mechanisms (*e.g.,* TSO/GSO, GRO), as it operates atop the TCP stack. It also supports TLS offloading, such as kTLS and NIC-based TLS, since decryption occurs in the kernel before messages reach Rakaia. Rakaia is, however, incompatible with SmartNICs that fully offload the TCP stack, as it relies on the kernel TCP pipeline to extract and queue messages. Finally, `io_uring` is compatible with Rakaia's `recvmsg`/`sendmsg` interface and could amortize syscall overhead.

Limitations inherited from TCP: Rakaia removes message-level HOL blocking caused by multiplexing RPCs over a TCP stream. However, because it leaves the TCP/IP stack unmodified, Rakaia necessarily inherits TCP's transport-level limitations. First, since TCP provides in-order delivery, a lost or reordered segment can stall progress on the affected connection until the missing data is retransmitted. Second, Rakaia inherits TCP's sender-driven congestion control, which is known to perform poorly under incast [8, 28, 60] compared to receiver-driven transports such as Homa [51, 56]. These tradeoffs are the inherent cost of preserving TCP wire compatibility and support for unmodified clients.

Complex scheduling policies and other RPC protocols: While Rakaia uses a logically centralized queue with run-to-completion workers, it can support richer queuing models (*e.g.,* priority queues). We leave this to future work. Rakaia can be easily extended to support other RPC protocols, provided they embed each message's length in their headers. Current parsers reside directly in the module, but additional ones can be implemented as in-kernel extensions or via eBPF.

kTLS: Linux's current implementation of kTLS does not correctly handle concurrent decryptions within a `softirq` context; Rakaia therefore falls back to a workqueue [44], incurring scheduler overhead. We plan to patch kTLS to support safe, scalable `softirq` decryption.

7 Related Work

Eliminating HOL blocking of TCP: Built atop HTTP/2 [5], gRPC [21] multiplexes multiple logical streams over a single TCP connection. While this design eliminates both inter- and intra-connection HOL blocking, it requires complex user-space orchestration. Rakaia instead performs message parsing, demultiplexing, and scheduling entirely in the kernel. Supporting HTTP/2 and other length-delimited message protocols, Rakaia enables efficient out-of-order delivery over TCP and integrates seamlessly with gRPC.

Addressing HOL blocking beyond TCP: Prior work also addresses HOL blocking by moving beyond TCP's byte-stream abstraction. One class of systems layers atop userspace UDP stacks: Shinjuku [32] and Concord [29] employ dedicated dispatchers and preemptive scheduling to achieve work conservation and low tail latency; Machnet [63] similarly bypasses the kernel via DPDK but as a networking sidecar, exposing a message-passing API over shared memory. QUIC [41], a reliable transport built over UDP, eliminates TCP's transport-level HOL blocking by multiplexing independent streams over a single connection. Apart from UDP-based approaches, recent custom transports [8, 51, 56, 60] treat messages as the fundamental unit of transfer and enable receiver-driven scheduling. While delivering excellent performance, these systems require either modified clients, custom network stacks, or dedicated infrastructure, limiting compatibility with existing TCP-based applications and hindering incremental deployment. In contrast, Rakaia preserves TCP/TLS wire compatibility, enabling easy deployment with unmodified clients across both WAN-facing and datacenter services.

Dataplane TCP stacks: Research dataplane TCP stacks often rely on DPDK [15] to bypass the Linux network stack for higher throughput and lower latency [4, 17, 31, 35, 36, 43, 59,

61, 63, 69]. Among them, IX [4], Arrakis [59], mTCP [31], and MICA [43] rely on NIC RSS to statically assign TCP flows to cores and are therefore not work-conserving. ZygOS [61], Shenango [55], and Caladan [18] address load imbalance through work stealing. However, they still follow the connection-shared queuing model and therefore suffer from intra-connection HOL blocking. Junction [17] aims to make kernel bypass practical for cloud operators by using advanced NIC features to enable high-density deployments of unmodified applications. While these systems support TCP, they replace the Linux TCP stack with custom userspace implementations. Rakaia instead runs atop the unmodified Linux TCP/IP stack, preserving its mature ecosystem.

Linux and BSD network stack optimizations: Because of its centrality, the Linux network stack is constantly being optimized [62, 64, 66]. In this context, KCM [37] provides message-granularity scheduling in the kernel, but suffers from several design flaws and fails to scale with increasing numbers of connections. Outside the kernel tree, Minion [53] enables out-of-order delivery of TCP segments and offers a datagram abstraction, thereby mitigating HOL blocking caused by packet loss or in-order delivery. Unlike Rakaia's logically centralized message queue, Minion exposes connection semantics to the application. Syrup [33] enables user-defined scheduling for network traffic using eBPF hooks but it is limited to UDP traffic. MegaPipe [26] uses per-core channels and batched system calls to minimize contention and improve throughput for short-lived connections and small messages, but remains a connection-oriented design. SKQ [70] extends FreeBSD's Kqueue to perform event scheduling but still suffers from HOL blocking as it schedules at the granularity of TCP connections. NetChannel [9] is an end-to-end transport redesign that multiplexes virtual sockets over underlying transport channels (*e.g.,* TCP connections) to scale network processing across cores. Like Rakaia, it decouples the application-facing socket abstraction from underlying channels. Unlike Rakaia, it requires virtual socket support at both endpoints, while Rakaia is purely server-side and works with unmodified TCP clients. Moreover, because NetChannel still binds packet delivery to the virtual socket chosen at connection setup, it cannot provide Rakaia-style work-conserving scheduling across sockets.

8 Conclusion

We presented Rakaia, a new in-kernel architecture for message-oriented scheduling atop the unmodified TCP/IP stack, eliminating both inter- and intra-connection head-of-line blocking. Rakaia provides a connection-oblivious API with a logically centralized message queue for applications and uses in-kernel transmit delegation to reduce contention across shared TCP connections. Rakaia is compatible with existing RPC protocols and supports transport-layer security out of the box. By combining connection-oblivious message scheduling with in-kernel transmit delegation, Rakaia outperforms KCM by 5× and improves gRPC-Go and gRPC-C++ throughput by up to 1.56× and 2.69×, respectively. Rakaia advances beyond Linux's current in-kernel solution, KCM, and is a practical replacement.

Acknowledgements

We thank Sanidhya Kashyap, Marios Kogias, Boris Pismenny, and the anonymous reviewers for their valuable feedback. We also thank Rüdiger Birkner, Charly Castes, Neelu Kalani, and Lyu Tao for their helpful comments. We thank Mihai Indreias, Marin Philippe, and Julien Ray for their contributions to Rakaia at various stages of the project. This work was funded by the Swiss State Secretariat for Education, Research, and Innovation (SERI) under the SwissChips initiative.

References

[1] Mohammad Alizadeh, Albert G. Greenberg, David A. Maltz, Jitendra Padhye, Parveen Patel, Balaji Prabhakar, Sudipta Sengupta, and Murari Sridharan. Data center TCP (DCTCP). In *Proceedings of the ACM SIGCOMM 2010 Conference*, pages 63–74, 2010.

[2] Mohammad Alizadeh, Shuang Yang, Milad Sharif, Sachin Katti, Nick McKeown, Balaji Prabhakar, and Scott Shenker. pFabric: minimal near-optimal datacenter transport. In *Proceedings of the ACM SIGCOMM 2013 Conference*, pages 435–446, 2013.

[3] Luiz André Barroso, Jimmy Clidaras, and Urs Hölzle. *The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines, Second Edition*. Synthesis Lectures on Computer Architecture. Morgan & Claypool Publishers, 2013.

[4] Adam Belay, George Prekas, Mia Primorac, Ana Klimovic, Samuel Grossman, Christos Kozyrakis, and Edouard Bugnion. The IX Operating System: Combining Low Latency, High Throughput, and Efficiency in a Protected Dataplane. *ACM Trans. Comput. Syst.*, 34(4):11:1–11:39, 2017.

[5] M. Belshe, R. Peon, and M. Thomson. Hypertext Transfer Protocol Version 2 (HTTP/2). RFC 7540 (Proposed Standard), May 2015.

[6] Andrew Birrell and Bruce Jay Nelson. Implementing Remote Procedure Calls. *ACM Trans. Comput. Syst.*, 2(1):39–59, 1984.

[7] Anat Bremler-Barr, David Hay, Idan Moyal, and Liron Schiff. Load balancing memcached traffic using soft-

ware defined networking. In *Proceedings of the 2017 IFIP Networking Conference*, pages 1–9, 2017.

[8] Qizhe Cai, Mina Tahmasbi Arashloo, and Rachit Agarwal. dcPIM: near-optimal proactive datacenter transport. In *Proceedings of the ACM SIGCOMM 2022 Conference*, pages 53–65, 2022.

[9] Qizhe Cai, Midhul Vuppalapati, Jaehyun Hwang, Christos Kozyrakis, and Rachit Agarwal. Towards μs tail latency and terabit ethernet: disaggregating the host network stack. In *Proceedings of the ACM SIGCOMM 2022 Conference*, pages 767–779, 2022.

[10] Jingrong Chen, Yongji Wu, Shihan Lin, Yechen Xu, Xinhao Kong, Tom Anderson, Matthew Lentz, Xiaowei Yang, and Danyang Zhuo. Remote Procedure Call as a Managed System Service. In *Proceedings of the 20th Symposium on Networked Systems Design and Implementation (NSDI)*, pages 141–159, 2023.

[11] Eyal Cidon, Sean Choi, Sachin Katti, and Nick McKeown. AppSwitch: Application-layer Load Balancing within a Software Switch. In *Proceedings of the 1st Asia-Pacific Workshop on Networking (APNet)*, pages 64–70, 2017.

[12] Austin T. Clements, M. Frans Kaashoek, Nickolai Zeldovich, Robert T. Morris, and Eddie Kohler. The Scalable Commutativity Rule: Designing Scalable Software for Multicore Processors. *ACM Trans. Comput. Syst.*, 32(4):10:1–10:47, 2015.

[13] Jeffrey Dean and Luiz André Barroso. The tail at scale. *Commun. ACM*, 56(2):74–80, 2013.

[14] Henri Maxime Demoulin, Joshua Fried, Isaac Pedisich, Marios Kogias, Boon Thau Loo, Linh Thi Xuan Phan, and Irene Zhang. When Idling is Ideal: Optimizing Tail-Latency for Heavy-Tailed Datacenter Workloads with Perséphone. In *Proceedings of the 28th ACM Symposium on Operating Systems Principles (SOSP)*, pages 621–637, 2021.

[15] Data plane development kit. http://www.dpdk.org/.

[16] Dmitry Duplyakin, Robert Ricci, Aleksander Maricq, Gary Wong, Jonathon Duerig, Eric Eide, Leigh Stoller, Mike Hibler, David Johnson, Kirk Webb, Aditya Akella, Kuang-Ching Wang, Glenn Ricart, Larry Landweber, Chip Elliott, Michael Zink, Emmanuel Cecchet, Snigdhaswin Kar, and Prabodh Mishra. The Design and Operation of CloudLab. In *Proceedings of the 2019 USENIX Annual Technical Conference (ATC)*, pages 1–14, 2019.

[17] Joshua Fried, Gohar Irfan Chaudhry, Enrique Saurez, Esha Choukse, Íñigo Goiri, Sameh Elnikety, Rodrigo Fonseca, and Adam Belay. Making Kernel Bypass Practical for the Cloud with Junction. In *Proceedings of the 21st Symposium on Networked Systems Design and Implementation (NSDI)*, pages 55–73, 2024.

[18] Joshua Fried, Zhenyuan Ruan, Amy Ousterhout, and Adam Belay. Caladan: Mitigating Interference at Microsecond Timescales. In *Proceedings of the 14th Symposium on Operating System Design and Implementation (OSDI)*, pages 281–297, 2020.

[19] Peter Xiang Gao, Akshay Narayan, Gautam Kumar, Rachit Agarwal, Sylvia Ratnasamy, and Scott Shenker. pHost: distributed near-optimal datacenter transport over commodity network fabric. In *Proceedings of the 2015 ACM Conference on Emerging Networking Experiments and Technology (CoNEXT)*, pages 1:1–1:12, 2015.

[20] crypto/tls: support kernel-provided tls. https://en.wikipedia.org/wiki/TCP_Offload_Engine.

[21] grpc. https://grpc.io/.

[22] grpc c++ asynchronous api. https://grpc.io/docs/languages/cpp/async/.

[23] grpc c++ callback api. https://grpc.io/docs/languages/cpp/callback/.

[24] grpc performance board. https://grafana-dot-grpc-testing.appspot.com/?orgId=1. Accessed on 02 Dec 2025.

[25] grpc performance best practices. https://grpc.io/docs/guides/performance/.

[26] Sangjin Han, Scott Marshall, Byung-Gon Chun, and Sylvia Ratnasamy. MegaPipe: A New Programming Interface for Scalable Network I/O. In *Proceedings of the 10th Symposium on Operating System Design and Implementation (OSDI)*, pages 135–148, 2012.

[27] Mark Handley, Costin Raiciu, Alexandru Agache, Andrei Voinescu, Andrew W. Moore, Gianni Antichi, and Marcin Wójcik. Re-architecting datacenter networks and stacks for low latency and high performance. In *Proceedings of the ACM SIGCOMM 2017 Conference*, pages 29–42, 2017.

[28] Torsten Hoefler, Karen Schramm, Eric Spada, Keith D. Underwood, Cedell Alexander, Bob Alverson, Paul Bottorff, Adrian M. Caulfield, Mark Handley, Cathy Huang, Costin Raiciu, Abdul Kabbani, Eugene Opsasnick, Rong Pan, Adee Ran, and Rip Sohan. Ultra Ethernet's Design Principles and Architectural Innovations. *CoRR*, abs/2508.08906, 2025.

[29] Rishabh R. Iyer, Musa Unal, Marios Kogias, and George Candea. Achieving Microsecond-Scale Tail Latency Efficiently with Approximate Optimal Scheduling. In *Proceedings of the 29th ACM Symposium on Operating Systems Principles (SOSP)*, pages 466–481, 2023.

[30] Jaeger. https://www.jaegertracing.io/.

[31] Eunyoung Jeong, Shinae Woo, Muhammad Asim Jamshed, Haewon Jeong, Sunghwan Ihm, Dongsu Han, and KyoungSoo Park. mTCP: a Highly Scalable User-level TCP Stack for Multicore Systems. In *Proceedings of the 11th Symposium on Networked Systems Design and Implementation (NSDI)*, pages 489–502, 2014.

[32] Kostis Kaffes, Timothy Chong, Jack Tigar Humphries, Adam Belay, David Mazières, and Christos Kozyrakis. Shinjuku: Preemptive Scheduling for μsecond-scale Tail Latency. In *Proceedings of the 16th Symposium on Networked Systems Design and Implementation (NSDI)*, pages 345–360, 2019.

[33] Kostis Kaffes, Jack Tigar Humphries, David Mazières, and Christos Kozyrakis. Syrup: User-Defined Scheduling Across the Stack. In *Proceedings of the 28th ACM Symposium on Operating Systems Principles (SOSP)*, pages 605–620, 2021.

[34] Anuj Kalia, Michael Kaminsky, and David G. Andersen. FaSST: Fast, Scalable and Simple Distributed Transactions with Two-Sided (RDMA) Datagram RPCs. In *Proceedings of the 12th Symposium on Operating System Design and Implementation (OSDI)*, pages 185–201, 2016.

[35] Anuj Kalia, Michael Kaminsky, and David G. Andersen. Datacenter RPCs can be General and Fast. In *Proceedings of the 16th Symposium on Networked Systems Design and Implementation (NSDI)*, pages 1–16, 2019.

[36] Antoine Kaufmann, Tim Stamler, Simon Peter, Naveen Kr. Sharma, Arvind Krishnamurthy, and Thomas E. Anderson. TAS: TCP Acceleration as an OS Service. In *Proceedings of the 2019 EuroSys Conference*, pages 24:1–24:16, 2019.

[37] Kernel connection multiplexor. https://docs.kernel.org/networking/kcm.html.

[38] Marios Kogias and Edouard Bugnion. HovercRaft: achieving scalability and fault-tolerance for microsecond-scale datacenter services. In *Proceedings of the 2020 EuroSys Conference*, pages 25:1–25:17, 2020.

[39] Marios Kogias, Stephen Mallon, and Edouard Bugnion. Lancet: A self-correcting Latency Measuring Tool. In *Proceedings of the 2019 USENIX Annual Technical Conference (ATC)*, pages 881–896, 2019.

[40] Marios Kogias, George Prekas, Adrien Ghosn, Jonas Fietz, and Edouard Bugnion. R2P2: Making RPCs first-class datacenter citizens. In *Proceedings of the 2019 USENIX Annual Technical Conference (ATC)*, pages 863–880, 2019.

[41] Adam Langley, Alistair Riddoch, Alyssa Wilk, Antonio Vicente, Charles Krasic, Dan Zhang, Fan Yang, Fedor Kouranov, Ian Swett, Janardhan R. Iyengar, Jeff Bailey, Jeremy Dorfman, Jim Roskind, Joanna Kulik, Patrik Westin, Raman Tenneti, Robbie Shade, Ryan Hamilton, Victor Vasiliev, Wan-Teh Chang, and Zhongyi Shi. The QUIC Transport Protocol: Design and Internet-Scale Deployment. In *Proceedings of the ACM SIGCOMM 2017 Conference*, pages 183–196, 2017.

[42] Yueying Li, Nikita Lazarev, David Koufaty, Tenny Yin, Andy Anderson, Zhiru Zhang, G. Edward Suh, Kostis Kaffes, and Christina Delimitrou. LibPreemptible: Enabling Fast, Adaptive, and Hardware-Assisted User-Space Scheduling. In *Proceedings of the 30th IEEE Symposium on High-Performance Computer Architecture (HPCA)*, pages 922–936, 2024.

[43] Hyeontaek Lim, Dongsu Han, David G. Andersen, and Michael Kaminsky. MICA: A Holistic Approach to Fast In-Memory Key-Value Storage. In *Proceedings of the 11th Symposium on Networked Systems Design and Implementation (NSDI)*, pages 429–444, 2014.

[44] Linux workqueue. https://docs.kernel.org/core-api/workqueue.html.

[45] Yi Lu, Qiaomin Xie, Gabriel Kliot, Alan Geller, James R. Larus, and Albert G. Greenberg. Join-Idle-Queue: A novel load balancing algorithm for dynamically scalable web services. *Perform. Evaluation*, 68(11):1056–1071, 2011.

[46] Zhihong Luo, Sam Son, Dev Bali, Emmanuel Amaro, Amy Ousterhout, Sylvia Ratnasamy, and Scott Shenker. Efficient Microsecond-scale Blind Scheduling with Tiny Quanta. In *Proceedings of the 29th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-XXIX)*, pages 305–319, 2024.

[47] Sarah McClure, Amy Ousterhout, Scott Shenker, and Sylvia Ratnasamy. Efficient Scheduling Policies for Microsecond-Scale Tasks. In *Proceedings of the 19th Symposium on Networked Systems Design and Implementation (NSDI)*, pages 1–18, 2022.

[48] Memcached. https://memcached.org/.

[49] Memcached binary protocol. https://docs.memcached.org/protocols/binary/.

[50] Michael Mitzenmacher. The Power of Two Choices in Randomized Load Balancing. *IEEE Trans. Parallel Distributed Syst.*, 12(10):1094–1104, 2001.

[51] Behnam Montazeri, Yilong Li, Mohammad Alizadeh, and John K. Ousterhout. Homa: a receiver-driven low-latency transport protocol using network priorities. In *Proceedings of the ACM SIGCOMM 2018 Conference*, pages 221–235, 2018.

[52] NGINX. Nginx reverse proxy, 2023. https://docs.nginx.com/nginx/admin-guide/web-server/reverse-proxy [Accessed: (06/2023)].

[53] Michael F. Nowlan, Nabin Tiwari, Janardhan R. Iyengar, Syed Obaid Amin, and Bryan Ford. Fitting Square Pegs Through Round Pipes: Unordered Delivery Wire-Compatible with TCP and TLS. In *Proceedings of the 9th Symposium on Networked Systems Design and Implementation (NSDI)*, pages 383–398, 2012.

[54] Opentelemetry collector. https://github.com/open-telemetry/opentelemetry-collector.

[55] Amy Ousterhout, Joshua Fried, Jonathan Behrens, Adam Belay, and Hari Balakrishnan. Shenango: Achieving High CPU Efficiency for Latency-sensitive Datacenter Workloads. In *Proceedings of the 16th Symposium on Networked Systems Design and Implementation (NSDI)*, pages 361–378, 2019.

[56] John K. Ousterhout. A Linux Kernel Implementation of the Homa Transport Protocol. In *Proceedings of the 2021 USENIX Annual Technical Conference (ATC)*, pages 99–115, 2021.

[57] John K. Ousterhout. It's Time to Replace TCP in the Datacenter. *CoRR*, abs/2210.00714, 2022.

[58] Packetbeat memcache fields. https://www.elastic.co/docs/reference/beats/packetbeat/exported-fields-memcache.

[59] Simon Peter, Jialin Li, Irene Zhang, Dan R. K. Ports, Doug Woos, Arvind Krishnamurthy, Thomas E. Anderson, and Timothy Roscoe. Arrakis: The Operating System Is the Control Plane. *ACM Trans. Comput. Syst.*, 33(4):11:1–11:30, 2016.

[60] Konstantinos Prasopoulos, Ryan Kosta, Edouard Bugnion, and Marios Kogias. SIRD: A Sender-Informed, Receiver-Driven Datacenter Transport Protocol. In *Proceedings of the 22nd Symposium on Networked Systems Design and Implementation (NSDI)*, pages 451–471, 2025.

[61] George Prekas, Marios Kogias, and Edouard Bugnion. ZygOS: Achieving Low Tail Latency for Microsecond-scale Networked Tasks. In *Proceedings of the 26th ACM Symposium on Operating Systems Principles (SOSP)*, pages 325–341, 2017.

[62] Microsoft corp. receive side scaling. http://msdn.microsoft.com/library/windows/hardware/ff556942.aspx.

[63] Alireza Sanaee, Vahab Jabrayilov, Ilias Marinos, Anuj Kalia, Divyanshu Saxena, Prateesh Goyal, Kostis Kaffes, and Gianni Antichi. Fast Userspace Networking for the Rest of Us. *CoRR*, abs/2502.09281, 2025.

[64] Scaling in the linux networking stack. https://www.kernel.org/doc/Documentation/networking/scaling.txt.

[65] Apache thrift. https://thrift.apache.org/.

[66] Tcp offload engine. https://en.wikipedia.org/wiki/TCP_Offload_Engine.

[67] Stephen Tu, Wenting Zheng, Eddie Kohler, Barbara Liskov, and Samuel Madden. Speedy transactions in multicore in-memory databases. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP)*, pages 18–32, 2013.

[68] David Watson. Crypto kernel tls socket. https://lwn.net/Articles/665602/, 2015.

[69] Irene Zhang, Amanda Raybuck, Pratyush Patel, Kirk Olynyk, Jacob Nelson, Omar S. Navarro Leija, Ashlie Martinez, Jing Liu, Anna Kornfeld Simpson, Sujay Jayakar, Pedro Henrique Penna, Max Demoulin, Piali Choudhury, and Anirudh Badam. The Demikernel Datapath OS Architecture for Microsecond-scale Datacenter Systems. In *Proceedings of the 28th ACM Symposium on Operating Systems Principles (SOSP)*, pages 195–211, 2021.

[70] Siyao Zhao, Haoyu Gu, and Ali José Mashtizadeh. SKQ: Event Scheduling for Optimizing Tail Latency in a Traditional OS Kernel. In *Proceedings of the 2021 USENIX Annual Technical Conference (ATC)*, pages 759–772, 2021.

ARTIFACT EVALUATED
USENIX
AVAILABLE

kSTEP: Characterization and Deterministic Testing of Linux CPU Scheduler Bugs

Tingjia Cao[†] Shawn (Wanxiang) Zhong[†*] Caeden Whitaker[†] Ke Han[§]
Andrea C. Arpaci-Dusseau[†] Remzi H. Arpaci-Dusseau[†]
† University of Wisconsin-Madison § Purdue University

Abstract

We present an in-depth study of CPU scheduler bugs, covering both functional violations and misalignments between implementation and policy. Our study shows that these bugs are hard to observe and even harder to trigger. To close this gap, we introduce kSTEP. kSTEP gives testers fine-grained control over scheduler-invoking events and runs them deterministically on isolated CPUs. kSTEP produces noise-free, repeatable traces that expose subtle behavior. We then build a coverage-guided fuzzer on top of kSTEP for automated CPU scheduler testing. We demonstrate kSTEP's effectiveness by reproducing seven real-world scheduler bugs and uncovering four new ones. kSTEP is publicly available at https://github.com/kstep-dev/kstep.

1 Introduction

Linux CPU scheduling is critical to application performance and service quality [2,24,49,55,77,82,85,93–97]. Real-world services inevitably experience severe performance drops due to scheduler bugs. CoreOS (a container-optimized OS [84]) reported that a scheduler bug caused a 50% performance loss in web applications [12]; another bug caused system hangs on 30% of nodes in a production Hadoop cluster every week [87].

CPU schedulers have complex correctness expectations. Prior work focuses on *functionality bugs* [8,43–45,67,107], but schedulers must also keep their implementation aligned with intended policies (*e.g.,* policies specify when and where a task runs while balancing desirable properties like fairness, efficiency, and migration costs). We call violations of this alignment *policy bugs*. When they occur, schedulers still make functional choices, but compromise service quality. Policy bugs have especially low observability; to external users, there is no clear signal confirming a bug has occurred (*e.g.,* a fairness drop may be a bug or an intentional trade-off).

Despite its importance and complexity, the scheduler remains poorly tested and poorly understood. First, existing scheduler tests are surprisingly inadequate. Developers mostly rely on long-running workloads to catch performance regressions [10,34,36,40,69,74,88], rather than using systematic tests targeting specific code paths and corner cases. Furthermore, we lack a quantitative understanding of the real scheduler bugs that have accumulated over the years. This raises the question: *What types of bugs exist in modern schedulers? Why are they so difficult to test? How can testing be improved?*

Characterization study (§3). To answer these questions, we conduct an in-depth characterization study of modern Linux scheduler bugs. We examined 232 bug-fix commits since 2020, uncovering how these bugs are triggered, what causes them, and what they break. Our study reveals twelve key findings, organized into three main categories.

First, scheduler bugs are important. Their consequences are non-negligible: 15% lead to fatal failures; many others break strict guarantees, directly undermining functional correctness; and even when functionality holds, policy bugs compromise important properties (*e.g.,* balance, fairness, locality, energy efficiency) that services depend on. The community's response confirms this importance: fixes are carefully backported, showing a sustained commitment to reliability.

Second, scheduler bugs are hard to find. The first issue is that they suffer from poor observability: only 27% manifest as clear panics or warnings, while the vast majority (73%) are silent functional faults or policy misalignments that leave no warnings or sufficient triggering traces. Most root causes (75%) are subtle semantic faults, such as state or logic errors. These arise from extensive functionality requirements and accumulated heuristics, and uncovering them necessitates deep domain knowledge. Moreover, triggering conditions are complex: many bugs require exploration of a large space of scheduler attributes, and some (28%) further depend on kernel events or specific CPU attributes.

Third, current practice does little to compensate. Many bugs (45%) stay hidden for years, sometimes for over a decade. Most policy bugs are still caught by manual review. These bugs are rarely reproduced: most have no clear trigger sequence, and only 22% of user-submitted fixes include developer validation. Preventive actions are uncommon as well: patches rarely add tests, warnings, or tracepoints, leaving the subsystem prone to regressions. It is surprising that such a fragile, review-driven process remains the state of the art.

Together, these findings show that scheduler bugs are important, deeply hidden, and systematically underserved by today's testing practices, highlighting the need for a systematic approach to scheduler testing.

Controlled and deterministic testing with kSTEP (§§4-5). We then propose kSTEP[1] for *controlled* and *deterministic* CPU-scheduler testing. kSTEP offers two key capabilities.

[*] Major contributor to the design and implementation of kSTEP.

[1] kSTEP: **K**ernel **S**cheduler **T**est and **E**valuation **P**latform

First, controlling the timing of scheduler-invoking events (*i.e.,* external events that invoke the scheduler to make decisions or change its state) is crucial for triggering scheduler bugs. Almost all scheduler bugs are time-sensitive, as scheduler state evolves continuously with frequent timer ticks. Tiny shifts in event timing can alter subsequent state transitions and scheduling decisions. Yet, traditional userspace tests lack such precise control. To achieve controlled testing, kSTEP provides a rich set of events for userspace tasks, kernel activities, CPU attributes, and explicit scheduler ticks. Testers specify the sequence of events in a *driver program*, and a kSTEP kernel module dispatches them in the specified order. Thus, testers can precisely drive scheduler state changes through kSTEP events (*e.g.,* testers can reach a target scheduler state with controlled ticks and insert an event at that moment). Scheduler testing is similar but orthogonal to controlled concurrency testing that focuses on discovering concurrency bugs by manipulating thread interleavings [29, 33, 37, 38, 59, 72, 78, 79]. These frameworks use their own schedulers instead of exercising the existing Linux scheduler code.

Second, kSTEP ensures repeatable, noise-free scheduler execution. Many scheduler bugs, especially policy bugs, can only be diagnosed by inspecting the internal state. Without determinism and isolation, it is difficult to tell if trace differences are due to bugs or normal variability. To achieve deterministic scheduler execution, kSTEP: mocks clock sources; isolates test CPUs from noisy activities; and resets the initial scheduler state before the driver program runs. Existing record-and-replay techniques for userspace programs [14, 15, 25–27, 66, 75, 80, 81, 91] and virtual machines [4, 31, 90, 106] also ensure determinism on the same input. However, these methods are not applicable to scheduler testing as they record noise (*i.e.,* not stable [104]), making it hard to compare across similar inputs or different kernels.

These properties provide a foundation for advanced scheduler testing tools. We build the kSTEP tracer, which records stable execution paths for CPU scheduler, helping developers infer event sequences to reproduce bugs. We further build the kSTEP fuzzer to explore scheduler behaviors automatically: it uses clean coverage as feedback, deterministically replays prefixes of interesting tests, and mutates subsequent events.

Evaluation with a case study (§6). Finally, we evaluate kSTEP through a detailed case study of seven sampled scheduler bugs. Despite the difficulty of triggering and observing these bugs in practice, kSTEP enables simple, effective testing. kSTEP makes it feasible to trigger every sampled bug with at most 20 events instead of hours-long benchmarks; kSTEP makes observation deterministic, producing stable, noise-free traces that expose buggy behaviors with measurable impact. We also uncovered four previously unknown bugs (*i.e.,* two by manually written tests and two by the automated kSTEP fuzzer). These results show that kSTEP enables concise, deterministic tests for scheduler bugs that would otherwise be especially challenging to trigger or inspect in practice.

2 Background and Motivation

2.1 Linux CPU Scheduler

The scheduler decides which runnable thread each CPU executes next. Linux implements multiple scheduling policies through scheduling classes, each encapsulating its own policy. CPU scheduling is critical for quality of service. For example, cloud platforms rely on the Linux scheduler to determine who runs where and when; containers and microVMs depend on its policies for proportional CPU time sharing and quota enforcement [1, 2, 52, 89]. Over decades, the kernel has accumulated complicated scheduling policies [16, 24, 69], including early designs like Round Robin in 2000, the Completely Fair Scheduler (CFS) in 2007 [77], Deadline (DL) scheduling in 2014 [94], and later NUMA-aware [93], capacity-aware [96], and energy-aware [97] refinements. Recent years have brought even faster evolution, including Earliest Eligible Virtual Deadline First (EEVDF) [108] and the extensible userspace scheduling framework `sched_ext` [95].

2.2 Scheduler Bugs Are Not Well Understood

CPU schedulers have two tiers of correctness expectations: they must preserve functional correctness and keep the implementation aligned with policy. Accordingly, we classify scheduler bugs into *functionality bugs* and *policy bugs*.

Functionality bugs. The *first tier of correctness expectations* for the scheduler is functional correctness. Functionality specifies what must hold for the system to be valid: services remain available and all strict guarantees are respected. For example, the scheduler must always run tasks only on allowed CPUs, prevent starvation of runnable tasks, and avoid crashes.

Prior work has extensively examined functionality bugs. Fail-stop, fail-slow [8, 43, 71, 92], and gray failures [45] all expose strong external signals (*i.e.,* crashes, timeouts, or unacceptable performance drops). Prior work on silent semantic bugs [67, 68, 70] studies non-fatal functionality bugs. They are quieter, but still surface through unambiguously incorrect results or inconsistent states (*e.g.,* missing replicas, stale reads, or lost data). Thus, people can confidently identify bug occurrences by validating outputs or persisted states.

Policy bugs. The *second tier of correctness* for schedulers is to keep their implementation aligned with their policies. Policy guides how to choose among many functional actions to deliver "good" service (*e.g.,* the scheduler aims to deliver fairness, cache locality, and efficiency).

We refer to a *policy bug* as an implementation defect where the scheduler is functional, but its implementation misaligns with the intended policy. When policy bugs occur, service quality can degrade unpredictably: workloads may experience unfairness, performance drops, short-term work conservation violations, or high scheduling cost. Policy bugs matter as service quality is central to modern systems [9, 18, 28, 102, 103]. We cannot assume these service quality drops are less harmful than rare crashes: at scale, small but persistent inefficiencies translate into continuous waste in capacity and energy.

```
int wake_affine_idle(int this_cpu, ..., int sync) {
  struct rq *rq = cpu_rq(this_cpu);     // curr runqueue
  if (sync &&                           // sync wakeup
-     (rq->nr_running == 1))            // single runnable
+     (rq->nr_running - cfs_h_nr_delayed(rq) == 1))
    return this_cpu;                    // stay on core
}
```

Figure 1: Excerpt of a policy bug in waker CPU selection: aa3ee4f (*sched/fair: Fixup wake_up_sync() vs delayed dequeue*).

Unlike well-studied functionality bugs [8,43,67,68], policy bugs cannot be confidently identified by external operators. System calls succeed, tasks complete, and the system appears "healthy." Their symptoms are performance anomalies, but these are not sufficient evidence of a bug, since intentional policy trade-offs may also degrade performance for certain workloads. As a result, it is difficult to construct principled oracles for policy bugs without internal state inspection.

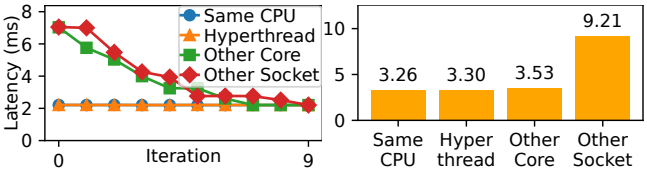
Prior work on performance bugs [48, 60] focuses on implementation inefficiencies that slow the system without affecting its outputs. Policy bugs can impact performance in two ways. Some, similar to traditional performance bugs, slow down the scheduler itself without changing its output. Most of them, however, alter scheduling decisions, thereby impacting application performance.

Bug #1: A real-world policy bug. Figure 1 shows a policy bug in CFS (aa3ee4f). During a synchronous wakeup, the waker is about to sleep. In this case, the wakee should run on the waker's CPU if the waker is the only runnable task. This ensures the local CPU retains its warm cache and boosted frequency. The code checks for this condition by testing whether `nr_running` == 1. However, EEVDF's delayed-dequeue feature [108,109] breaks this rule. Sleepers that overshoot their slice stay enqueued to burn off debt, still counting to `nr_running`. These "ghost" sleepers make the runqueue look busy, so the wakee is pushed to another CPU, needlessly sacrificing a warm cache and boosted frequency.

Figure 2 shows the performance impact. In Figure 2.1 (CPU-bound), the waker wakes the child, which then runs 10 identical busy-loop iterations. Selecting the waker's core keeps latency low as it is already frequency-boosted; a remote core makes the first iteration 3× slower and requires ~40ms to recover. In Figure 2.2 (cache-bound), accessing parent-touched data takes 3× longer on a remote socket than the local CPU. Such slowdowns matter: prior work shows it costs 1% of revenue per additional 100 ms tail latency [3, 28].

When a user sees a wakee run on a remote CPU, they cannot assume this bug occurred. The scheduler may legitimately prefer a remote CPU (*e.g.,* with higher capacity to fit the wakee). Thus, an external observer cannot tell whether the behavior reflects a bug or an intended policy design. Exposing it requires inspecting internal scheduler state and decisions. In §6.1, we show how kSTEP exposes this buggy behavior.

Our contribution. Despite their impact on service quality and complexity, scheduler bugs (both functionality violations and misalignment between implementation and policy) remain understudied. To narrow this gap, we conduct a comprehensive characterization study (§3) and detailed case studies (§6).



(1) CPU-bound child latency. **(2)** Cache-bound child latency.

Figure 2: Impact of sync-wake misplacement.

2.3 Scheduler Bugs Are Difficult to Test

In this section, we examine current approaches to scheduler correctness, their limitations, and challenges in testing.

Current efforts toward scheduler correctness. We review existing approaches and demonstrate why they are inadequate.

• *Testing.* Existing scheduler tests leave most behavior untouched. In the Linux Test Project [53], the CFS category consists of only three tests for bandwidth throttling, starvation, and hackbench [88], despite 30k lines of code (LoC) in CFS. In practice, existing tests focus primarily on basic operations such as thread creation and CPU affinity, along with a handful of stress tests using synthetic workloads.

• *Benchmark.* Performance benchmarks are often used to evaluate schedulers [10, 34, 36, 40, 69, 74, 88]. However, this approach is neither reliable nor complete. First, benchmarks may show performance drops but cannot expose incorrect behaviors without a reference result. Second, their results are often noisy, making bugs hard to diagnose. Third, benchmarks lack the control needed to reliably reproduce bugs.

• *Simulation.* LinSched [19] is a Linux scheduler simulator, allowing developers to write and execute tests outside the kernel. Its development highlights the need for scheduler testing tools. However, it lacks fidelity as a simulator: it omits many kernel interactions and does not support symmetric multiprocessing (SMP). Furthermore, due to high maintenance costs, it has not been maintained since Linux v3.3.

• *Verification.* Existing work has produced verified schedulers [51, 58] and proofs for specific Linux scheduling functions [56, 57]. These efforts either rely on overly simplified scheduling policies or cover only small code sections, and thus cannot capture the full range of Linux's scheduler policies.

Challenges for scheduler testing. Tests consist of event sequences designed to trigger failures; people inspect test outputs to determine whether a bug occurred [105]. In a scheduler context, event sequences (fork, wakeup, quota updates, *etc.*) invoke scheduling operators (enqueue, dequeue, reschedule, *etc.*) and generate diverse behaviors. However, schedulers are not tested extensively in practice using the standard approach.

• *Challenge 1: Scheduler behavior is highly time sensitive.* Testing the scheduler demands more than simply replaying a sequence of events: precise control over the timing of every event is essential. The scheduler's internal state evolves with every tick, so even small differences in when events occur can result in significantly different scheduling outcomes. For

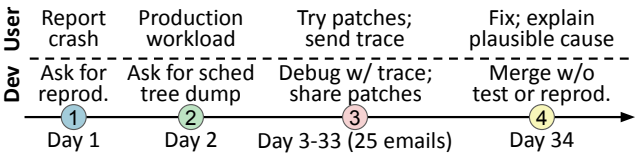


Figure 3: Timeline of a production scheduler bug (bbce3de): weeks of remote debugging without a confirmed reproducer. No test was added to prevent future regressions.

instance, Bug #1 is triggered only if a kthread's synchronous wakeup happens during a narrow, transient window when the waker's runqueue contains more queued than runnable tasks.

• *Challenge 2: Userspace events are insufficient.* Current tests focus on specifying userspace events to trigger the scheduler, overlooking many crucial events that originate from the kernel or hardware. Other kernel subsystems call scheduler-exported methods or create kernel threads; the hardware timer triggers ticks, invoking load balancing and task switches. Controlling these events is challenging from userspace. In addition, scheduler decisions depend on CPU attributes (*e.g.,* topology, capability, frequency). Hardware diversity is growing: from asymmetric cores to multi-level topologies (*e.g.,* chiplet [7], sub-NUMA clustering [46]). Testing across this diversity is impractical. Physical hardware is costly, and virtualization platforms do not emulate all variations. Thus, many hardware-specific bugs are surfaced only in production (*e.g.,* ee6e44d).

• *Challenge 3: Identify bugs from noisy traces.* When a bug is triggered, it is also difficult to observe. The scheduler is highly nondeterministic, so traces differ across runs, making it difficult to tell whether differences come from bugs or from normal variability. Modern kernels add noise with interrupts, RCU callbacks, workqueue activity, and clock behavior. Long-running benchmarks worsen the problem: a bug may surface occasionally, buried among unrelated event traces.

Bug #2: Month-long debugging without a reproducer. Figure 3 shows a recent case [23]. ① A user reported a null-pointer dereference, and the developer asked whether it was reproducible. ② The user could only describe the symptoms (`pick_task_fair` crashes more often) and could not reproduce it outside production. ③ Tedious rounds of remote debugging followed: the user dumped scheduler trees, the developer studied them and suggested patches, and the cycle repeated (25 emails in total). Both sides mentioned trying to reproduce the bug "to crank up the iteration speed," but without success. ④ Eventually, the user assembled a complex failure path and proposed a patch, which was merged later (bbce3de). The bug was not reproduced; the fix was never validated against a test. The patch was approved as developers thought it made sense. In contrast, a 20-event reproducer with kSTEP can reproduce it deterministically (§6.1).

Our contribution. To address these challenges, we introduce kSTEP (§4) and the testing tools built on top of kSTEP (§5). Users write driver programs to specify an event sequence; kSTEP executes them deterministically in an isolated environment with unmodified Linux kernels.

	Type	Responsibilities	#Bugs
Framework	Core	Skeleton and orchestration among sched classes	108
	Topology	Builds the hierarchy of scheduling domains	10
	Account	CPU time accounting and cgroup CPU statistics	9
Classes	Fair	Fair scheduling via vruntime and load balancing	106
	RT	Fixed-priority real-time scheduling	9
	Deadline	Earliest deadline wins scheduling	25

Table 1: Logical components. The per-component bug counts exceed the total because one commit can affect multiple components.

3 Characterization Study

In this section, we present key findings from our scheduler bug analysis. Do some components contain more bugs (§3.2)? How long do bugs exist in the kernel (§3.2)? How hard are they to observe (§3.3)? What consequences do they cause (§3.4)? How are they found and handled (§3.5)? Why do they occur (§3.6)? What triggers them (§3.7)? Overall, our findings show that the CPU scheduler is hard to test and that current practice falls short. Many bugs are hard to observe and even harder to trigger, pushing developers towards code inspection and long-running benchmarks instead. These results also highlight opportunities for more effective scheduler testing.

3.1 Methodology and Limitations

In this section, we give a brief description of how we collect bugs and discuss the limitations of our methodology.

Methodology. We collected all bug-fix commits in the Linux CPU scheduler from mainline [61] and stable branches [42] since 2020. We scanned all commits modifying files in `kernel/sched/`, excluding `sched_ext`, and classified a commit as a bug fix if it contained a *Fixes:<SHA>* trailer or indicated a defect (*e.g.,* using terms such as *fix*, *regression*, *inversion*, *stall/hang*, *violate*, *revert*). We excluded commits that added features, optimized design, maintained code, or modified comments. After filtering, we obtained a study set of 232 scheduler bugs, each investigated and classified by one of the authors.

Limitations. As with all characterization studies, there is an inherent risk that our findings may not be representative.

• *Representativeness of the selected bugs.* First, our study covers bug-fix commits, and it may miss real-world scheduler bugs that were encountered by users but never reported or fixed. Second, our study should be interpreted as characterizing bugs in the Linux scheduler rather than scheduler implementations in general. Third, our bug-filtering criteria rely on signals such as *Fixes:<SHA>* trailers and bug-indicating keywords. We use conservative criteria to reduce false positives, but this may miss bug-fix commits without these signals.

• *Possible observer error.* Although we manually examined each collected patch, some classification decisions require judgment and may be subject to researcher bias.

3.2 Overview of Scheduler Bugs

The Linux CPU scheduler has grown from 15k LoC to 50k LoC in 15 years, turning a once-small subsystem into one of the largest subsystems in Linux. Where does the code live, and how do scheduler bugs and fixes move through releases?

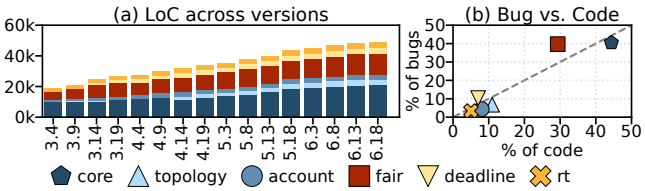


Figure 4: Code size and bug counts of different components.

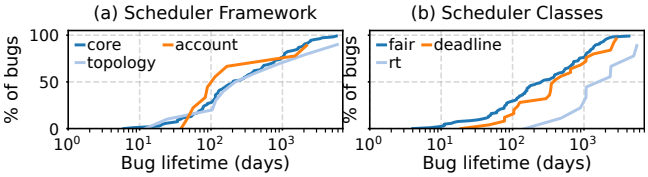


Figure 5: Distribution of bug lifetimes across different components.

Logical components in the CPU scheduler. We divide the CPU scheduler into 2 components with 6 logical subcomponents (Table 1). The scheduler does two things: the *scheduler framework* provides the core mechanisms and *scheduler classes* implement specific policies. The framework has three parts. *Core* defines the policy-agnostic skeleton: it runs *schedule()* for task switching, manages task states, and exposes exported functions used by syscalls and other subsystems (cgroups, power, *etc.*). *Topology* maps hardware layout into scheduling domains for load balancing and CPU selection. *Accounting* tracks and reports runtime statistics to users, forming the observability layer. The scheduler classes then plug into this framework. *Fair* implements the default CFS policy, *RT* implements fixed-priority real-time scheduling, and *Deadline* implements earliest-deadline-first scheduling, ensuring tasks receive X runtime every Y period, as configured.

Bug distribution across components. Table 1 lists the bug counts by component. Figure 4a shows how each component grows across kernel versions, and Figure 4b compares each component's share of code to its share of bugs.

• *Scheduler framework.* The *Core* component accounts for nearly half of all bugs and about 40% of the scheduler's code. Although *Core* is mostly policy-agnostic, it orchestrates the entire scheduling pipeline and interacts with user workloads, other kernel subsystems, and hardware signals. This infrastructure is complex and thus a major source of failures.

• *Scheduler classes.* Among the classes, *Fair* grows fastest and contains more bugs relative to its size. CFS accumulates a large number of heuristics over the years (*e.g.,* NUMA balancing, EEVDF, energy-aware hooks) with little structural cleanup. For example, *select_idle_sibling()* is the final step in CPU selection, deciding whether to use the candidate CPU or its nearby idle sibling. Despite its simple task, it spans 72 LoC and 33 predicates with heuristics accumulated over years.

Finding 1: *The scheduler framework accounts for nearly half of bugs, due to extensive functionality requirements; the Fair scheduler has accumulated heuristics and is bug-prone.*

Lifetime. Figure 5 shows how long bugs stay in mainline Linux before being fixed. Roughly 45% remain for over a year, and 5% survive for more than a decade. These bugs often stay hidden because they require specific hardware topologies or unusual workloads that standard benchmarks never hit. Many also compromise scheduler properties like fairness or work conservation, which are soft targets with no ground-truth curves to compare against. The presence of multi-year bugs stresses how hard it is to trigger and expose scheduler bugs.

Finding 2: *45% of bugs hidden for years, 5% over a decade.*

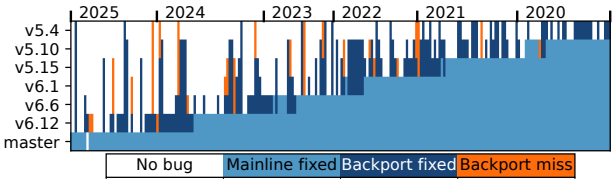


Figure 6: Backported versions for 232 bugs. Columns represent bugs, rows represent versions, and cells represent backport status.

	Type	Description
	Crash/Hang	Kernel becomes unusable or makes no progress
Functionality	Sched Attr	Violation of sched attribute guarantees (*e.g.,* CPU affinity)
	Trace	The scheduler's trace records incorrect values
	Task State	Invalid state (*e.g.,* runnable task missing from all queues)
	Hotplug	CPUs fail to online/offline (*e.g.,* task remains on offline CPU)
	Starvation	Starvation occurs in starvation-free scheduler classes
	Return Val	Functions exposed to other subsystems return wrong values
	Config	Feature-config violation (*e.g.,* enabled feature not used)
Policy	Balance	Leaving some CPUs overloaded while others idle
	Fairness	Some tasks get more than their fair share of CPU
	Freq Ctrl	Miscoordination with frequency/utilization control
	Locality	Reduces cache efficiency or degrades NUMA locality
	Energy Eff	Degrades energy efficiency (*e.g.,* waking too many CPUs)
	Sched Cost	Increases scheduling overhead or cost in hot paths
	Class Prio	Wrong order across classes (*e.g.,* DL lock holder not boosted)
	Benign	Little impact (*e.g.,* warning triggered but corrected instantly)

Table 2: Categories of bug observability and consequences.

Backport. Figure 6 shows how scheduler bug fixes propagate from mainline Linux to Long-Term Support (LTS) branches. Each cell shows how a version handles a fix: (1) the bug is absent, (2) the fix is present before release, (3) the fix is correctly backported, or (4) the fix should have been backported but is missing. We find that once a scheduler fix lands in mainline, it is propagated to all affected versions in most cases, even when those branches differ significantly in age and code.

Finding 3: *The community shows a sustained commitment to scheduler reliability: most bugs are backported properly.*

3.3 Bug Observability

As discussed in §2.2, violations of functionality and policy have different observability. In this section, we aim to examine the observability of scheduler bugs.

Categories. We first classify each bug by observability; each category exhibits distinct symptoms, as listed in Table 2.

• *Functionality: Crash/Hang*. Bugs can fail-stop the kernel with a panic, oops, hard lockup, or an unrecoverable hang. Users notice them immediately as the service is unavailable.

• *Functionality: Non-fatal*. These bugs violate strict func-

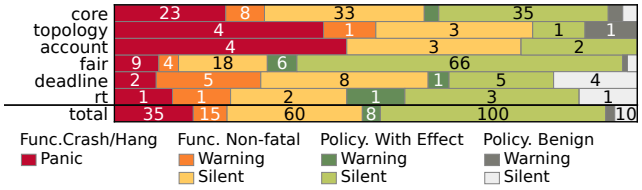


Figure 7: Observability and diagnostic logs of bugs. Bugs can span components, so component-wise counts can exceed the total.

tional guarantees (*e.g.,* affinity not honored, tasks failing to wake). While these bugs are less immediately visible, they can still be detected through explicit checks by users.

• *Policy: With Effect*. These bugs preserve functional semantics, but the implementation does not align with the intended design. They can compromise soft properties the scheduler wants to achieve, as shown in Table 2. External symptoms alone are insufficient to distinguish a policy bug from an intentional design trade-off. Determining whether a bug occurs requires examining the internal scheduler state.

• *Policy: Benign*. Policy bugs whose effects are minimal or quickly self-correcting. They have no obvious impact and are usually observable only through code review or log warnings.

Bug observability. Figure 7 shows the observability of each bug. 15% of scheduler bugs cause highly visible failures such as crashes or hangs. Silent functional violations (26%) and misalignment between policy and implementation (47%) dominate, a higher share of silent bugs than prior work [67].

Policy bugs are especially common in *Fair*: 70% of its bugs fall into this category. The reason is that CFS offers proportional sharing in a best-effort manner while balancing different goals (fairness, locality, energy, *etc.*). Policy bugs occur when these heuristics drift from the intended design. Other components also contribute to policy bugs. The *Core* layer maintains policy-critical state (runqueue statistics, task attributes, *etc.*); *Topology* defines scheduling domains to guide balancing; *RT* and *Deadline*, despite their originally functional semantics, have accumulated heuristics (*e.g.,* energy-aware scheduling) and contain a number of policy bugs.

How much do logs reveal? Figure 7 also marks whether a bug triggers any kernel panics or warnings. Scheduler warnings have narrow coverage: they guard only coarse invariants (*e.g.,* avoiding double enqueues) rather than checking whether each scheduling decision is valid or policy-consistent. As a result, 73% of bugs leave no warnings in the logs. In addition, for nearly all bugs in our study, the critical scheduler events are not recorded by default tracepoints. While powerful tracing tools exist [17, 54, 62, 69, 99, 101], fully tracing scheduler activity is expensive and disturbs time-sensitive decisions, making it impractical for production workloads. Thus, the bug-triggering event sequences are often absent in traces.

Finding 4: *Silent functionality and policy bugs dominate (73%) the scheduler bugs. Logs and traces are insufficient to reveal them or record their triggering events.*

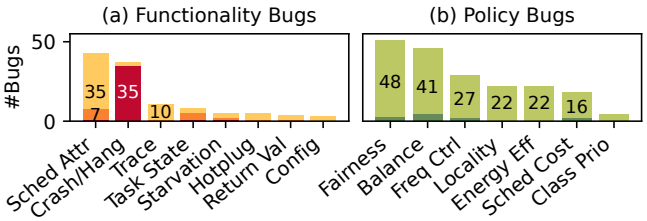


Figure 8: Bug consequences. Consequence definitions in Table 2. Colors follow Figure 7. Counts exceed 232 via consequence overlap.

3.4 Bug Consequences

In this section, we examine how scheduler bugs affect the system and applications using the categories in Table 2.

Functionality violations. Figure 8a shows that while crashes and hangs are common (35 cases), most functional bugs are silent violations. Breaking strict interface semantics is the most common scheduler failure (42 cases). The scheduler exposes many interfaces for configuring task-, cgroup-, and system-level attributes. Some have strict specifications (*e.g.,* allowed CPU sets, the CPU time a Deadline task must receive within its window). Violating them constitutes a functionality bug. Others express non-strict goals (*e.g.,* proportional share for CFS tasks, short-term work conservation) that the scheduler attempts but does not guarantee at all times; falling short of these goals does not indicate a functionality bug. The remaining cases come from breaking other strict guarantees in the scheduler skeleton. Table 2 shows a detailed breakdown.

Finding 5: *Most functionality bugs result from violating strict requirements of the scheduler-attribute interface.*

Service quality compromises. Policy bugs do not violate functional guarantees. Their symptoms are compromises of soft expectations (Table 2). For example, CFS shares express a soft expectation about the proportion of CPU time a task should receive relative to its peers. Such a share is often traded off against other goals, such as migration costs. We examine bugs by the soft expectations they compromise (Figure 8b).

Balance and fairness dominate because they sit at the center of many trade-offs. Balance can conflict with locality, energy efficiency, and scheduling cost. Fairness competes with frequency control, energy efficiency, and scheduling cost. To balance these trade-offs, the scheduler relies on numerous heuristics, making it particularly fragile. Hardware-related policies (locality, freq control, and energy-aware decisions) are frequently violated (73 cases). Modern hardware diversity forces the scheduler to grow many architecture-specific rules, making these paths especially bug-prone. Moreover, some bugs make the scheduler inefficient by slowing decision-making, increasing migration rates, or causing redundant calculations.

Finding 6: *Balance and fairness are the most fragile properties; hardware-aware policies are also hard to get right.*

3.5 Bug Discovery and Mitigation in Practice

Despite the scheduler's critical role, bug handling remains largely ad hoc. This section examines how scheduler bugs are

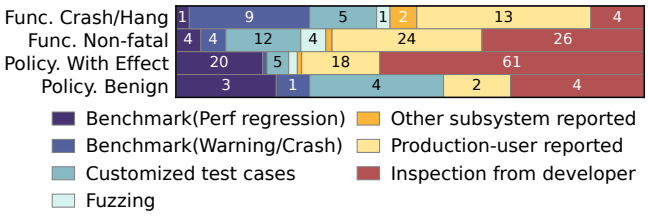


Figure 9: Discovery methods for scheduler bugs.

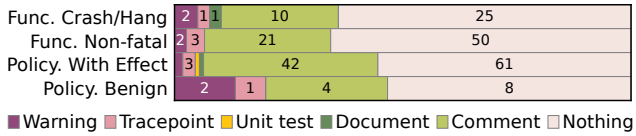


Figure 10: Prevention actions taken after fixing scheduler bugs. Prevention actions can overlap, so the total can exceed 232 bugs.

discovered, reproduced, and prevented in practice.

How are bugs discovered? Figure 9 shows how bugs in each observability level first surface: via tests (benchmark regressions, customized test cases, fuzzing tools), user reports (from either other kernel subsystems or users), or code review. Fatal errors are easiest to see, so tests and users catch 89% of them. Non-fatal functionality bugs keep the system running, but users and tests still find many of them (65%) because they can observe clear semantic violations. In contrast, users and tests only find 47% of policy bugs due to low observability. The system remains functional while service quality quietly drifts, and users cannot tell whether a performance dip is a policy bug or an intended trade-off. The remaining policy bugs are found through code review; a fragile process makes policy violations easy to miss.

> ***Finding 7:*** *Policy bugs rely heavily on code review to uncover; far fewer are reported by users or exposed by tests.*

Lack of details to reproduce bugs. Bugs are rarely reproduced during the scheduler bug-fixing process. Figure 9 shows that 61% of runtime-exposed bugs are observed by regressions or alerts from long-running workloads, with no precise triggering sequence mentioned from report to fix. More surprisingly, developers reproduce the bug and validate the patch for only 22% of user-submitted patches. Instead, patch approval relies on code inspection and the fact that "the symptom went away". The scarcity of reproducers reflects the scheduler's poor testability. §2.3 discusses the challenges that make it difficult to construct stable, repeatable bug-triggering sequences. Without reliable reproducers, validation is difficult; there are no tests to prevent reintroduction of the bug.

> ***Finding 8:*** *Bugs are rarely reproduced: 61% of runtime-exposed bugs lack a trigger sequence; only 22% of user-reported patches are explicitly reproduced and validated.*

Lack of preventive action. Mature systems typically require bug fixes to include regression tests [5, 83], which CI runs to ensure the same bug does not recur. Figure 10 shows a very different pattern: most fixes are "one-shot" patches with no preventive action. The dominant outcome is *Nothing*. The most common follow-up is a code comment, which is helpful for clarity but provides no enforcement. Unit tests and selftests are almost never added; runtime checks are rare. Once a scheduler bug is patched, little is done to prevent similar bugs from recurring (*e.g.,* `db6cc3f`). It depends on the vigilance of developers rather than regression tests.

> ***Finding 9:*** *Common practices rarely prevent bug recurrence, as fixes rarely add unit tests, warnings, or tracepoints.*

Category	Description
State	Incorrect updates or missed updates of scheduler state (*e.g.,* wrong statistics, corrupted runnable lists or rb-trees).
Logic	Scheduling decisions made from state are undesirable.
Generic	Generic semantic error (*e.g.,* wrong types, overflows, typos)
Concurrency	Race conditions or violations of locking protocols
Memory	Unsafe mem ops (*e.g.,* leak, null ptr, buffer overflow)

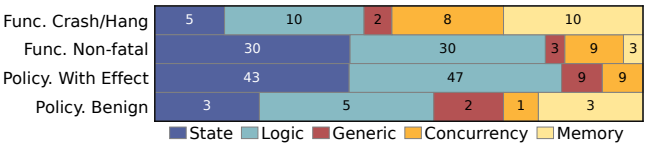


Figure 11: Categories and distribution of bug root causes.

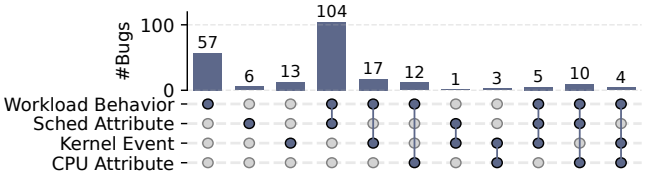


Figure 12: Triggering conditions for scheduler bugs. Each column shows a group; dots mark the combination of trigger types required.

3.6 Root Cause

Understanding why bugs arise is essential. Different root causes call for different ways to detect, expose, and fix them, so a clear taxonomy helps both developers and tool builders. We classify bugs into five categories listed in Figure 11. Overall, *state* and *logic* faults make up the vast majority (75%) of bugs. These semantic errors often demand scheduler-domain knowledge to understand, detect, and fix. General root causes common to all systems, including *concurrency* and *memory* bugs, account for 12% and 7% of cases, respectively; they are more likely to lead to crashes or hangs and are therefore easier to observe. Generic coding mistakes, such as integer overflows or typos, are relatively rare. These bugs can also lead to different levels of observability and can sometimes be difficult to expose.

> ***Finding 10:*** *75% of scheduler bugs stem from incorrect state updates or wrong logic, and likely require domain expertise to detect and resolve. These bugs often remain silent.*

3.7 Triggering Conditions

There are opportunities to improve testing by examining the types of input events required to trigger bugs. We analyze each bug's triggering events, with results shown in Figure 12.

Workload behavior. Almost all bugs (90%) require specific

userspace workload behaviors like task count, arrival patterns, CPU-usage patterns, synchronization, or sleep/wake timing.

Scheduler attributes. For 54% of the bugs, the trigger depends on how scheduler attributes are configured. Linux exposes rich interfaces for expressing performance requirements at the task, cgroup, and system levels. Among the studied bugs, 24%, 16%, and 4% rely on per-task, cgroup-level, and system-wide attributes, respectively; 8% appear only when multiple classes of tasks interact. From a testing perspective, these attributes form a rich input space, and effective testing must systematically explore these attribute combinations.

> ***Finding 11:*** *Exploring the search space for different scheduler attributes is important for 54% of bugs.*

Kernel events. Some triggers are events raised by other Linux subsystems: workqueue kthread pools, filesystem kthread activities, CPU hotplug, freezer operations, and so forth. These account for about 19% of bugs (10% hotplug, 4% kthread behavior, 5% other). These events invoke the scheduler through a large set of exported functions, and are difficult to orchestrate from userspace (*i.e.,* one cannot control the timing of filesystem kthread wakeups by issuing file operations).

Special CPU attributes. Most bugs (88%) can be triggered on a multicore machine with SMT. The rest require specialized attributes: 5% on NUMA systems, 5% on asymmetric cores, and 2% on others (*e.g.,* 4-way SMT, CPU-less NUMA nodes, or some private IBM hardware). These cases show that some failures only occur on CPUs with specific attributes that typical test environments rarely cover.

> ***Finding 12:*** *28% of bugs cannot be triggered by specific userspace behavior alone; tests must explore kernel events and varied hardware topologies at the same time.*

4 kSTEP: Kernel Scheduler Test Platform

To address the challenges of scheduler testability (§2.3), we present kSTEP[2], a framework for *controlled* and *deterministic* execution of the Linux CPU scheduler in an isolated environment. In this section, we describe the design and implementation of kSTEP. In §5, we demonstrate how kSTEP enables the development of testing tools for the Linux scheduler, such as an execution-path tracer and a coverage-guided fuzzer. We design kSTEP around the following principles:

Precise timing. Scheduler bugs are often highly sensitive to timing, posing a significant challenge for testing and reproduction (Challenge 1). kSTEP provides fine-grained control over event timing, enabling precise triggering of bugs.

Rich control. Another challenge is insufficient control over the scheduler state from userspace (Challenge 2). kSTEP exposes events for userspace tasks, kernel activities, CPU attributes, and ticks, enabling control over scheduler behaviors.

Noise-free determinism. Noisy and nondeterministic scheduler behaviors make it difficult to detect and diagnose bugs (Challenge 3). kSTEP guarantees that the same input always produces the same clean scheduler trace, enabling reliable bug reproduction and precise comparisons of scheduler behaviors.

[2]In ASCII, kSTEP is a bit flip away from KSTEP (Kernel Scheduler: Two Easy Pieces), which in turn is a single bit flip away from OSTEP [9].

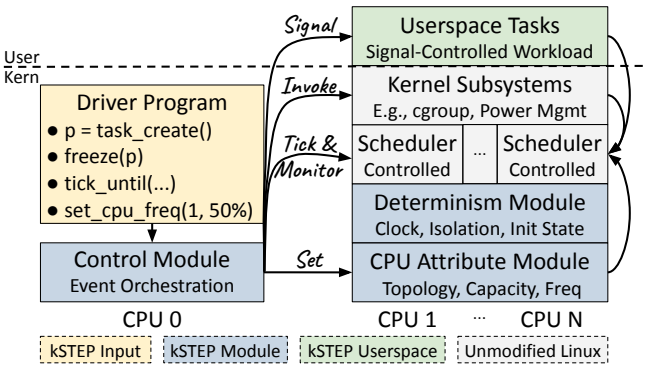


Figure 13: Architecture of kSTEP.

Fidelity. To ensure that the insights gained through kSTEP are directly applicable to real systems, kSTEP guarantees that its traces reflect real-world scheduler behavior.

Portability. To enable reliable reproduction and analysis of scheduling behaviors across different kernel versions, kSTEP does not require modification to the kernel source and is compatible with a broad range of kernel versions.

4.1 kSTEP Overview

Figure 13 illustrates the architecture of kSTEP. The input to kSTEP is a driver program that specifies a sequence of events to execute. The left side of the figure shows a dedicated CPU responsible for running both the driver program and the control module; the right side shows a controlled environment with the remaining CPUs deterministically executing the events as specified by the control module.

Controlled execution (§4.2). The driver program runs in kernel space and uses the control module to dispatch the events to the controlled environment. The events include not only actions to control processes but also kernel events (*e.g.,* kthread) and CPU attributes (*e.g.,* topology), which are otherwise impossible to control from userspace. One special event is the explicit scheduler tick. By issuing ticks manually, kSTEP advances logical time and triggers the scheduler exactly when desired. Explicit tick control enables kSTEP to order events relative to scheduler execution, allowing precise control. To preserve fidelity, the driver program cannot directly modify scheduler state; it reaches desired states only through events to exercise normal kernel code paths.

Deterministic execution (§4.3). On the remaining CPUs, userspace runs only the processes explicitly created by the control module, with nothing else permitted. These processes do not encode any workload-specific logic of their own; instead, their execution is orchestrated by the control module via signals, eliminating variability due to unpredictable execution speeds and making program behavior deterministic.

On the kernel side, the scheduler runs unmodified, but with an additional determinism module to eliminate all sources of

```
// Setup tasks: t1 runnable, t2: paused, t3 runnable
t1, t2, t3 = task_create()×3; task_wakeup({t1, t3});
// Setup cgroup hierarchy: g0: {g1: {t1}, g2: {t2, t3}}
cgroup_create({"/g0", "g0/g1", "g0/g2"});
cgroup_move_task("/g0/g1", {t1});
cgroup_move_task("/g0/g2", {t2, t3});
cgroup_set_weight({"/g0/g1", "/g0/g2"}, 15);
// Prepare scheduler state
tick_until(/* t1 eligible, but g0 and g1 ineligible */);
cgroup_move_task("/", {t1, t3});
// Trigger bug
cgroup_destroy("/g0/g1");       // Bug: overflow g0's vlag
cgroup_set_weight("/g0", 106); // Bug: overflow vruntime
// Observe task starvation
task_wakeup(t2); tick_repeat(200);
```

Figure 14: Driver pseudocode to reproduce Bug #2 in Figure 3. We discuss the driver program in more detail in Table 4.

Type	Events
Task	task_{create(), pause(p), wakeup(p), pin(p, cpumask), fork(p), set_sched(p, sched), set_prio(p, prio), freeze(p)}
Kern	cgroup_{create(g), move_task(g, p), set_weight(g, w), pin(g, cpumask), destroy(g)}, kthread_{create(), pause(p), wakeup(p)}, and any exported kernel function
Tick	tick(), tick_repeat(n), tick_until(pred)
CPU	set_freq(cpu, val), set_capacity(cpu, val), set_topo(topo)

Table 3: Selected events provided by kSTEP. In the parameters, p is a task, g is a cgroup name, and pred is a predicate function.

nondeterminism affecting scheduler execution. It controls the clock values observed by the scheduler, suppresses unintended kernel activities (*e.g.,* interrupts), and resets the initial scheduler state that may otherwise vary with the boot sequence.

Limitation. While kSTEP supports concurrent and parallel execution of tasks on multiple CPUs, the sequence of events generated by the driver program is serialized. As a result, kSTEP cannot directly expose concurrency bugs within the scheduler. As noted in §3.6, these bugs are uncommon (12%).

4.2 Controlled Execution

kSTEP operates based on a driver program that defines a sequence of events to be executed. Controlled execution means the scheduler advances exclusively through driver-issued events. The driver is a C function compiled into a kSTEP control module; it calls kSTEP's event API to orchestrate tasks, inject ticks, and configure CPU attributes on the controlled CPUs (CPUs 1–N) in the desired order. Figure 14 shows an example driver that builds a cgroup hierarchy, advances ticks until a target scheduler state is reached, then restructures the cgroups to trigger a vruntime overflow bug. Table 3 lists selected events in kSTEP.

Userspace tasks. kSTEP provides a set of events to create and control userspace tasks. For example, a driver can create tasks, pause and wake them at chosen points, pin them to CPUs, and adjust their scheduler class or priority. Most task events send signals to the userspace program, exercising the usual userspace-to-kernel code paths. When no signal is pending, the userspace program simply runs a busy loop to stay runnable. A few events, such as freezing a task, are handled by kernel-side helpers.

Kernel subsystems. kSTEP includes a set of functions that interact with other kernel subsystems, for example, to create cgroups or kthreads. Since drivers run inside a kernel module, they can also call any exported kernel function. This direct access allows fine-grained control over kernel events.

Scheduler ticks. Scheduler state evolves with every timer tick, so time-sensitive bugs require controlling when ticks occur relative to other events. kSTEP therefore provides explicit tick events to trigger scheduler execution at chosen points. It also provides a "tick until" primitive: the driver supplies a predicate, and kSTEP injects ticks until the predicate is satisfied, enabling precise waits for transient scheduler states before issuing the next event.

CPU attributes. Scheduler decisions depend on hardware properties such as topology, CPU capacity, and frequency, but these configurations are hard to control on real machines or in virtualized environments [13]. kSTEP exposes these CPU attributes as events, allowing a driver to emulate diverse hardware setups and observe how they affect scheduler decisions.

4.3 Deterministic Execution

kSTEP incorporates a determinism module to make driver execution reproducible across runs on the same kernel. Unlike traditional record-and-replay approaches that attempt to control all sources of nondeterminism [4, 14, 15, 25–27, 31, 66, 75, 80, 81, 90, 91, 106], we identify three key factors that contribute to scheduler nondeterminism: clock sources, noisy activities, and initial scheduler state.

Mock clock sources. kSTEP mocks two clock sources to ensure deterministic execution: `sched_clock`, which governs general scheduling operations (*e.g.,* time slice), and `jiffies`, which tracks timing for the next load balancing event. After each scheduler tick finishes on all controlled CPUs, both clocks are advanced, guaranteeing that time-related behaviors are fully reproducible.

Isolate noisy activities. To eliminate the noise introduced by system activities, we isolate the controlled CPUs. First, we implement a custom, minimal userspace `init` program (PID 1) to ensure that there is no other unintended process running on the controlled CPUs. We also address sources of kernel-level nondeterminism by explicitly redirecting all interrupts, RCU callbacks, and workqueue workers to CPU 0. This isolation is essential to achieving noise-free scheduler execution by ensuring the only events affecting the scheduler are those deliberately controlled by kSTEP.

Reset scheduler state. Because the kernel boot sequence is inherently nondeterministic, before driver execution we reset scheduler state to a known initial state, including zeroing runqueue load estimates, resetting per-task `vruntime` values, and clearing load-balancing cost counters on each scheduler domain. This ensures each driver run starts from the same scheduler state, independent of boot-time noise.

4.4 Implementation

The kSTEP source code is available at https://github.com/kstep-dev/kstep. The kernel module is about 1.5k LoC, complemented by a lightweight userspace program for `init` and controlled userspace tasks in about 150 LoC.

Portability. We continuously test kSTEP against long-term support kernels from the past five years (v5.15, v6.1, v6.6, v6.12, and v6.18), as well as the latest release (v7.0, at the time of writing). kSTEP is fully compatible with both `x86_64` and `arm64`, and we confirm that it produces consistent results across both architectures. As the kernel evolves, maintaining kSTEP requires little effort.

Workflow. We provide scripts to download and build a specified kernel version, then run kSTEP on the unmodified kernel in QEMU [13]. Once the kernel boots, our custom `init` loads the kernel module and executes a specified driver program. The driver emits structured JSON output to a QEMU serial console, which is used in our analysis and visualization scripts, as well as high-level tools such as the tracer and fuzzer in §5.

Driver execution. A driver defines a `name`, a `setup()` function for initialization (*e.g.,* construct test objects), a `run()` function to issue the event sequence, and optional callbacks on scheduler events (*e.g.,* on tick or load-balance). After `setup()`, kSTEP cancels the tick timer on controlled CPUs, mocks `sched_clock` and `jiffies`, and resets scheduler state. kSTEP then executes `run()` to control tasks, invoke kernel events, inject ticks, and configure CPU attributes. To implement a tick event, kSTEP advances the mocked clocks, sends IPIs [65] to invoke the tick function on each controlled CPU, and waits until all controlled CPUs finish the tick and drain pending scheduler softirqs [64]. This ensures that all CPUs reach a consistent state before the next event.

Kernel integration. kSTEP uses ftrace [98] to trace selected scheduler functions, such as load-balancing and task-group allocation paths, allowing kSTEP to observe scheduler-internal events without modifying kernel source. To call private kernel functions, kSTEP locates their address with kallsyms [63].

5 Scheduler Testing with kSTEP

kSTEP's determinism and control over scheduling events unlock a wide range of testing and analysis capabilities for kernel developers and researchers. In this section, we implement an execution-path tracer (§5.1) and a coverage-guided fuzzer (§5.2) on top of kSTEP. We then discuss additional opportunities made possible by kSTEP (§5.3).

5.1 Deterministic Scheduler Execution Tracer

Based on kSTEP, we implement a deterministic execution-path tracer that records scheduler code paths each driver event triggers. These noise-free traces help developers reproduce bugs and inspect the scheduler paths that lead to a behavior.

Event-aware tracing. For each kSTEP event, the tracer reports program counters (PCs) separately for each participating task's execution path. For example, when task A wakes task B, the tracer records the waker path executed by A and the wakee path executed by B separately. This lets users inspect each individual scheduler execution path separately and understand how each path affects scheduler state. Because kSTEP replay is deterministic, the same test generates the same trace across runs. Developers can therefore refine event sequences against stable evidence, making elusive bugs easier to reproduce than in the original nondeterministic environment.

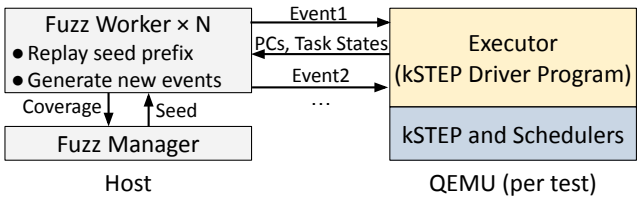


Figure 15: Overview of kSTEP fuzzer.

Implementation. We instrument the Linux kernel using LLVM SanitizerCoverage [100], which inserts PC-tracing callbacks on scheduler control-flow edges. We then symbolize PCs into source-code locations using `addr2line` [41]. To avoid noise, kSTEP records PCs mainly on the isolated test CPUs (CPUs 1–N). On CPU 0, tracing is active only while executing kSTEP events (*e.g.,* waking up a testing thread). This targeted approach ensures traces reflect test-induced scheduler behavior rather than unrelated kernel activity.

5.2 Coverage-Guided Scheduler Fuzzer

kSTEP enables noise-free coverage collection and exact event replay, making it a strong foundation for scheduler fuzzing [76]. We build a coverage-guided fuzzer on top of kSTEP to automatically explore scheduler states.

Overview. Figure 15 illustrates the kSTEP fuzzer. A host-side *manager* coordinates multiple *workers*, each running tests in a fresh QEMU instance. Each worker sends generated events to an in-guest *executor* (*i.e.,* a kSTEP driver program), which applies them to the Linux scheduler in deterministic order.

Interactive input generation. The fuzzer tracks task states (*i.e.,* running, runnable, blocked, frozen) to generate valid next events. For example, it only generates a `fork` event from a running task. After each event, the worker waits for completion and observes the updated task states. This interactive approach substantially reduces invalid inputs.

Coverage collection. The executor uses LLVM SanitizerCoverage [100] to log the program counters reached by each event. The worker converts these PCs into control-flow edge coverage and sends the result to the manager for corpus management [35]. As in §5.1, kSTEP filters PCs to avoid unrelated kernel activity. Recording coverage per event allows the fuzzer manager to pinpoint exactly which event triggers new scheduler behavior. Without kSTEP, unrelated kernel activity would pollute coverage, making fuzzing feedback unreliable.

Replay and mutation. Our mutation strategy builds on kSTEP's determinism to resume from interesting intermediate scheduler states and explore nearby behaviors. The manager marks events that yielded new coverage as split points; to

mutate a prior test, a worker replays the test up to a split point, then generates a short fresh event sequence from that point. Without deterministic replay, the same prefix could reach different states, making mutation-based fuzzing unreliable.

5.3 Potential Use Cases

In this section, we discuss additional use cases we believe can be supported by kSTEP's capabilities in the future.

Scheduler testing agents. kSTEP is well-suited for LLM-driven agents to automate scheduler testing. Such an agent can generate kSTEP driver programs to reproduce existing bugs or systematically explore scheduler behavior for new issues, testing each candidate sequence against a target kernel in seconds and iteratively improving based on results. This process depends on kSTEP's determinism: the deterministic execution trace gives the agent repeatable evidence, guiding targeted refinement of the event sequence.

Anomaly detection. kSTEP's deterministic scheduler execution produces repeatable traces that can be reliably analyzed. This provides a foundation for the development of advanced oracles to detect abnormal scheduler behavior automatically.

Policy evaluation. kSTEP 's ability to replay identical workloads deterministically enables precise comparisons across kernel versions or policy configurations. A tool built on this capability can quantify how a patch shifts scheduling decisions, expose subtle regressions, or validate that a new policy respects intended invariants.

6 kSTEP Evaluation with Case Study

Our study shows that scheduler bugs are hard to trigger and expose in practice. Thus, we propose kSTEP to execute a sequence of events deterministically, making the scheduler testable and analyzable. In this section, we evaluate kSTEP and ask the following research questions:

- §6.1: How hard is it to find kSTEP tests to trigger bugs?
- §6.2: How well can kSTEP's deterministic and noise-free trace help observe the buggy behavior and bug impacts?
- §6.3: How can kSTEP help uncover new scheduler bugs?

Case-study bug set. We sampled 7 bugs that span different components, observability, and triggers. Table 4 lists their descriptions and types (policy or functionality bug). Most bug reports lack details on how to reproduce them. These cases provide a diverse evaluation set for kSTEP.

Experiment setup. We conduct experiments on a machine with 2 Intel Xeon Silver CPUs (10 cores each) running Ubuntu 24.04. For each bug, we check out the repository immediately preceding the fix commit. We then run kSTEP on QEMU to execute our custom driver program that triggers the bug. Next, we check out the fix commit and re-run the program. Each run launches a new QEMU and shuts it down upon completion.

6.1 Trigger Bugs with kSTEP

In Table 4, we report the number of kSTEP events and LoC for each reproducer to evaluate how complex real scheduler bugs look under kSTEP 's event model. Long sequences would make kSTEP impractical. Reproducers for our studied bugs are compact: at most 20 steps and 47 LoC. Also, most studied bugs require at most five tasks to trigger, except for one case that stresses the scheduler with 20k tasks. Notably, most studied bugs were never reproduced by developers (*e.g.,* Bug #2 took a month to fix without a confirmed reproduction); kSTEP reduces them to short event sequences that run in seconds.

We also list the event types used in each reproducer. The set is diverse: task events that create, wake, pause, or freeze tasks; kernel events that control cgroups or kthreads; and CPU-frequency changes. In addition, three bugs require ticking until a specific state to hit a narrow transient window. This diversity matches our findings in §3.

Bug #1: Trigger a kernel event in a time window. This is the same bug in §2.2 (Figures 1 and 2): it only occurs when a kthread calls a sync wakeup when it is the only runnable one and the runqueue has `nr_queued` > `nr_running`.

The reproducer creates a kthread as the waker (*i.e.,* sync wakeup is an operation for kthreads only) and a userspace task that later becomes a "ghost" sleeper. Then, we tick until the userspace task becomes ineligible (*i.e.,* its runtime exceeds its share). We block it here so it remains delayed-dequeued under EEVDF policy and causes `nr_queued` > `nr_running`. With the paused task as a ghost sleeper and no other runnable tasks, both triggering conditions are satisfied. At this moment, we ask the waker to call sync wakeup. The same reproducer later showed that the upstream fix for Bug #1 was incomplete; we report the remaining bug separately as Bug #8 in §6.3.

This example shows how kSTEP lets testers reliably execute an event at a transient state. The tester advances the scheduler with "tick until," stops at the precise state, and then issues the target event at the same logical timestamp. Thus, the event is guaranteed to observe the expected transient condition, making hard-to-trigger bugs reliably reproducible.

Bug #2: cgroup hierarchy. This is the same bug in Figure 3, which remained unreproduced after a month of attempts. It requires a cgroup hierarchy and issuing an event at a transient state. Figure 14 shows the kSTEP reproducer in pseudocode.

To reproduce this bug, the driver program builds a cgroup hierarchy and places Task 1 (runnable) in cgroup 1, and Task 2 (paused) and Task 3 (runnable) in cgroup 2. The first precondition requires that cgroup 0's and cgroup 1's scheduling entities be delayed-dequeued. To reach this state, we tick the scheduler into a transient state where Task 1 is eligible but cgroup 1 and cgroup 0 are not. We then move Task 1 and Task 3 back to the root cgroup; since both tasks are currently eligible, the scheduler immediately dequeues their entities, while cgroup 0's and cgroup 1's entities remain delayed-dequeued to burn off their lag. This satisfies the first precondition.

Next, we destroy cgroup 1 so that cgroup 1's and cgroup 0's entities fully dequeue. We then reweight cgroup 0 so that the bug overflows its vruntime to a huge positive value. Finally, we wake up Task 2 and observe that Task 2 is starved because CFS avoids running cgroup 0 due to its inflated vruntime.

	ID	Description	Step	LoC	Task	CPU	T2T	Event Types	Impact
Studied Bugs	**#1** Policy	aa3ee4f: Sync-wakeup wakee not placed on waker's CPU when a delay-dequeued task exists	7	47	3	2	2.13	**Task**: create, pause, wakeup **Kernel**: kthread_{create, wakeup} **Tick until**: !eligible(tsk)	**Performance:** ~3× initial slowdown of the wakee
	#2 Func.	bbce3de: Starvation or crash due to vruntime overflow when a cgroup is dequeued under specific conditions	20	38	3	1	8.68	**Task**: create, pause, wakeup **Kernel**: cgroup_{create, set_weight} **Tick until**: eligible(tsk) && !eligible(grp)	**Correctness:** Kernel panic or tasks in the cgroup starved
	#3 Func.	cd9626e: Delay-dequeued task can still wake up after being frozen	5	28	3	1	0.02	**Task**: create, pause, wakeup, freeze **Tick until**: !eligible(tsk)	**Correctness:** Fails to capture a consistent process snapshot
	#4 Policy	6d7e478: Excessive rebalancing from multiple CPUs in the same sched group	6	28	5	2C4T	0.06	**Task**: create, pin	**Performance:** ~1 μs slowdown recurs roughly every 130 ms
	#5 Policy	17e3e88: util_avg misaccounted when the last real-time task blocks	5	26	2	1	0.49	**Task**: create, pause, set_prio **CPU**: set_freq	**Performance:** ~3× slowdown of a thread, recovers about 0.5 s later
	#6 Policy	2feab24: Pinning many tasks to one CPU makes load balancing very slow	2	21	20k	2	–	**Task**: create, pin	**Performance:** 2% CPU time is wasted for the useless rebalance
	#7 Policy	5068d84: Reweighting a running task misupdates min_vruntime	6	20	3	2	0.01	**Task**: create, set_prio **Kernel**: cgroup_{create, pin}	Benign
New Bugs	**#8** Policy	Sync-wakeup wakee not placed on the waker's CPU despite an idle sibling	7	47	3	2	0.04	**Task**: create, pause, wakeup **Kernel**: kthread_{create, wakeup} **Tick until**: !eligible(tsk)	**Performance:** ~3× initial slowdown of the wakee
	#9 Policy	Selecting the "equalize idle CPUs" balancing mode but failing to equalize idle CPUs across groups	22	44	4	4	0.27	**Task**: create, wakeup, pin **CPU**: set_capacity, set_topo	**Performance:** 1 CPU remains idle throughout execution despite pending runnable tasks
	#10 Policy	Wrong overload label on sched group prevents migration to an idle CPU	24	30	6	4	5.32	**Task**: create, wakeup, pin **CPU**: set_topo	**Performance:** 1 CPU stays idle despite pending runnable tasks
	#11 Policy	CPU's estimated utilization fluctuates sharply with an almost fully busy task	27	78	1	1	0.13	**Task**: create, wakeup, pause, set_prio **CPU**: set_capacity	**Performance:** CPU frequency may fail to ramp up

Table 4: Studied and newly found bugs. Bugs #1–#7 are studied bugs. Bugs #8 and #9 are new bugs we found using manually written kSTEP drivers. Bugs #10 and #11 are new bugs found by the kSTEP fuzzer. 2C4T: 2 cores and 4 threads. T2T: time to trigger by fuzzer in hours.

This bug has intricate triggering conditions, yet our reproducer needs only 20 events because kSTEP is able to stop the scheduler at the desired but transient state (*i.e.,* the task is eligible but its cgroup is not) with only one event, enabling reliable reproduction of bugs involving deep hierarchies.

Automated bug triggering. We use the kSTEP fuzzer (§5.2) to search for triggering event sequences. For each studied bug, we encode an invariant capturing the expected behavior; for instance, bug #4 is captured by: within a scheduling group, at most one designated core invokes the load-balancing algorithm [69]. We run the fuzzer for 24 hours, then manually inspect each violation to determine whether it reproduces the bug, is a false positive, or exposes a new anomaly.

Table 4 reports triggering times in hours. Most bugs are triggered within one hour, showing that kSTEP's deterministic, isolated execution lets the fuzzer reach buggy scheduler states effectively. Bug #2 is the hardest: it took developers 30 days to fix without a reproducer, while the fuzzer needs over 8 hours because the bug requires a condition (*i.e.,* a task must be eligible while the scheduling entities of its ancestors are not), which is reached only after exhaustive exploration. Bug #6 remains out of reach: it requires 20k threads, beyond our fuzzing scale. The fuzzer also uncovers two new bugs, Bug #10 and Bug #11, described in §6.3.

Summary. These cases demonstrate that, despite the complex triggering conditions of scheduler bugs, kSTEP enables testers to reproduce them with *simple, concise* test cases. kSTEP also enables effective automated fuzz testing of the scheduler, both to trigger the studied bugs and to uncover new anomalies.

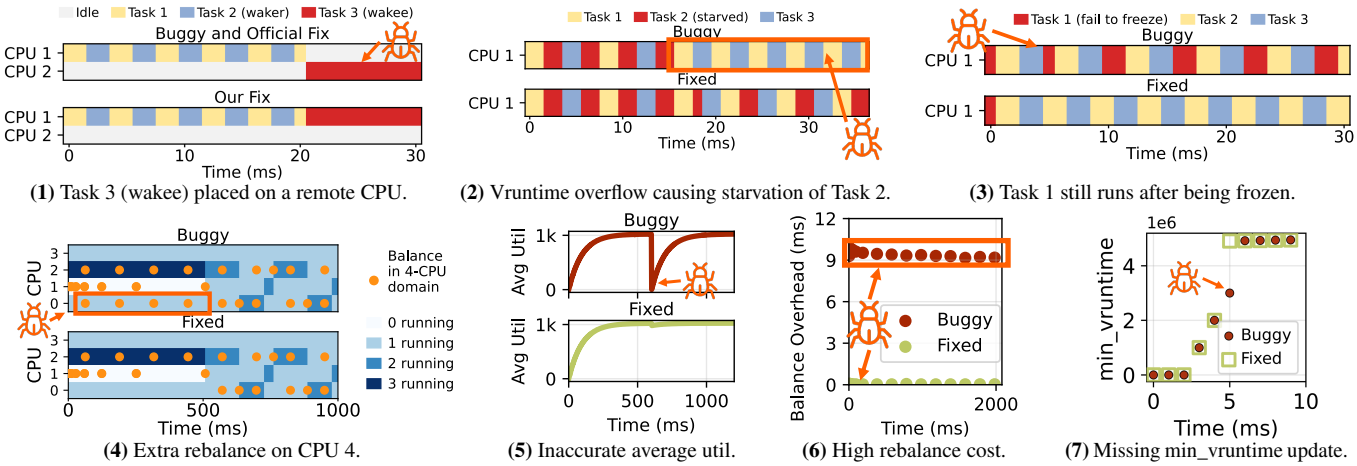
6.2 Observe Bugs with kSTEP

The output of kSTEP is a trace of the scheduler's internal states. Compared with uncontrolled systems, kSTEP traces are noise-free and deterministic. In this section, we visualize the trace generated by the reproducers of our studied bugs in Figure 16 and summarize their impact in Table 4. We first show how these traces help observe buggy behavior and support impact analysis through detailed case studies. We also evaluate the determinism of kSTEP by verifying that traces generated prior to the bug trigger are identical across runs.

Bug #1. Figure 16.1 shows a kthread calling `sync_wakeup` on CPU 1. In the buggy version, the wakee is placed on CPU 2, though CPU 1 (where the waker runs) is immediately idle; in the official fixed version (aa3ee4f), it still selects CPU 2, as the fix is incomplete. We provide a more detailed explanation in §6.3. With our patch, the wakee selects CPU 1, preserving locality. Figure 2.1 quantifies the potential impact: CPU 2 can have a low frequency as it is idle; the wakee can be 3× slower and recover after tens of milliseconds.

Bug #2. In Figure 16.2, once the bug is triggered, the vruntime of Task 2's cgroup entity overflows to a huge 64-bit value, making it effectively ineligible to run until peers catch up, so it is starved almost indefinitely. In the fixed trace, Task 2 runs normally, sharing CPU time with others.

Bug #3. In Figure 16.3, kSTEP sends a freeze to Task 1 when it is sleeping but remains on the runqueue (because of the delayed dequeue feature). In the buggy version, the task is not frozen successfully and later wakes up to run on its CPU again; in the fixed version, it remains frozen and cannot be



(1) Task 3 (wakee) placed on a remote CPU. **(2)** Vruntime overflow causing starvation of Task 2. **(3)** Task 1 still runs after being frozen.

(4) Extra rebalance on CPU 4. **(5)** Inaccurate average util. **(6)** High rebalance cost. **(7)** Missing min_vruntime update.

Figure 16: Visualization of the buggy behavior for each bug reproduced by kSTEP. In all figures, the *x*-axis shows logical time. The *y*-axis shows the CPU ID for (1)-(4); average utilization for (5); wall-time rebalance latency for (6); and minimum vruntime for (7). The colored segments show the running task on each CPU in (1)-(3), and the number of runnable tasks on each CPU in (4).

woken again. This harms correctness; applications relying on the freezer to snapshot a process can be affected.

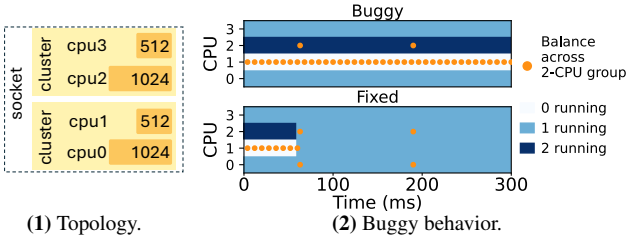
Bug #4. In Figure 16.4, we create and pin 1, 0, 3, and 1 tasks on CPUs 4–7, respectively, and examine which CPUs invoke rebalance across the 4-CPU sched domain. Ideally, only one CPU per sched group should do so; CPUs 4–5 and CPUs 6–7 form two groups. In the buggy version, both CPUs 4 and 5 attempt balancing, causing redundant calculations. In the fixed version, only one CPU per group invokes domain balancing. The impact is small (under 1 microsecond CPU time wasted per 130 ms) but can still matter for latency-critical applications. Once we remove CPU affinity for the tasks and allow migrations, the bug no longer triggers, and both versions revert to one balancing CPU per group.

Bug #5. Figure 16.5 shows that when the RT task sleeps and a CFS task takes over, the buggy scheduler drops the CPU's `util_avg` sharply and recovers after ~500 ms. A `util_avg`-driven frequency governor may lower CPU frequency and affect the task's latency (Figure 2.1). Also, the busy CPU may appear lightly loaded and attempt to pull tasks.

Bug #6. Figure 16.6 shows that pinning 20k tasks to one CPU makes load balancing on other CPUs expensive: these rebalance attempts take around 2.5 ms to finish. The fix adds a hard cap on the number of tasks to inspect, restoring normal cost (1 μs). In our setup, this bug wastes ~4 ms of CPU time every ~200 ms (the interval between affected rebalances).

Bug #7. This bug did not update the `min_vruntime` of a runqueue when reweighting a cgroup. In Figure 16.7, when triggered, the buggy version's `min_vruntime` is higher. However, this one-tick error has almost no impact on applications.

Determinism. In Figures 16.1-16.3, we see identical task switches before the trigger and clear divergence after it. In Figure 16.4, every balance attempt and task migration occurs at identical timestamps except for CPU 4. In Figures 16.5 and 16.7, buggy and fixed curves evolve identically up to the trigger. These results demonstrate kSTEP's determinism.



(1) Topology. **(2)** Buggy behavior.

Figure 17: In Bug #9, CPU 1 is idle under the specified topology.

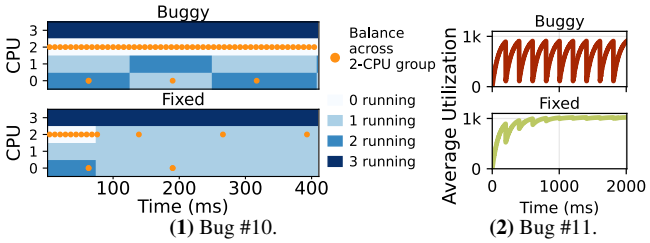
Summary. kSTEP enables deterministic and noise-free observation of scheduler bugs. It enables precise side-by-side comparison of buggy and fixed runs. The observation shows that most scheduler bugs cause noticeable disruptions across different aspects of system behavior.

6.3 New Bugs Discovered and Fixed

We also use kSTEP to uncover four new bugs in the latest Linux kernel, which we have reported to the developers. Bugs #8 and #9 are found using manually written kSTEP drivers. Bugs #10 and #11 are found by the kSTEP fuzzer.

Bug #8: Bug #1 patch does not completely fix it. Bug #8 is the remaining bug we exposed when validating Bug #1's upstream fix with our triggering program: the child still selects a remote CPU as shown in Figure 16.1. The root cause is CFS's two-phase wakeup. Phase 1 chooses a candidate CPU via heuristics such as `wake_affine_idle`; after the original fix, this phase correctly returns the local CPU. Phase 2 then decides whether to use the candidate CPU or a nearby idle CPU via `select_idle_sibling`, which still uses `nr_running` instead of the effective runnable count that excludes delayed-dequeue tasks. Thus, the patch fixed the misuse in `wake_affine_idle` but left the same issue in `select_idle_sibling`. After correcting both sites, the child selects the about-to-idle local CPU (Figure 16.1).

Bug #9: Work-conserving violation on a specific topology. Bug #9 occurs on a 2-cluster topology of varying CPU capacities (Figure 17.1). We pin 1, 0, 2, and 1 tasks on CPUs 0–3, re-



(1) Bug #10. (2) Bug #11.

Figure 18: Buggy behavior of Bug #10 and Bug #11.

spectively, then relax the affinity of tasks on CPU 2 to CPUs 1–2. The idle CPU 1 should quickly pull a task, but instead remains idle while CPU 2 stays overloaded (Figure 17.2).

The root cause is a wrong calculation. To equalize idle CPUs across the local and busiest groups, the scheduler computes the imbalance as: (local.idle_cpus - busiest.idle_cpus) / 2. In our workload, the local group has one idle CPU, and the busiest group has no idle CPU. The formula computes an imbalance of 0, so the scheduler performs no migration. This contradicts the intended policy to balance idle CPUs. To fix this bug, when the local group has one more idle CPU than the busiest group and the busiest group is overloaded, we set the imbalance to 1 instead of 0. Figure 17.2 shows that, with our fix, idle CPU 1 quickly pulls a task from busy CPU 2.

Bug #10: Work-conserving violation due to a wrong group label. We found Bug #10 on a 2-cluster topology of 4 CPUs. We pin 3 tasks to CPU 3 and 3 tasks to CPUs 0–1, then relax the affinity of the latter to CPUs 0–3. CPU 2 is idle and should quickly pull a task because there are 3 tasks eligible to run on it. However, Figure 18.1 shows CPU 2 stays idle indefinitely.

The root cause is a mislabeling: CPUs 2–3 form a cluster where CPU 3 is overloaded and CPU 2 is idle, so the cluster should be labeled *imbalanced*. Instead, the scheduler labels it *overloaded*, causing the load-balancing logic to conclude no migration is needed. With the fix, CPU 2 promptly pulls a task from CPU 1 (Figure 18.1).

Bug #11: utilization avg plunges immediately after sleep. We place a real-time task on a low-capacity CPU that runs for 200 ticks and sleeps for 1 tick, keeping the CPU nearly fully utilized. Its estimated utilization should thus stay high and stable, but Figure 18.2 shows it drops sharply when the task sleeps. The cause is that the scheduler's idle-time accounting mishandles low-capacity CPUs, overestimating how long the CPU was idle. The fix scales the idle-time estimate by the CPU capacity, keeping the estimate high for the busy task.

Summary. These results show that kSTEP uncovers new scheduler bugs with simple tests and provides a foundation for future regression and automated testing.

7 Related Work

CPU scheduler correctness. Many benchmarks have been built to evaluate schedulers: Hackbench [88] stresses message passing; Schbench [74] simulates web servers; Adrestia [34] exercises load balancing; Rt-app [10] emulates mobile tasks; and Cyclictest [40] measures wakeup latency. Another line of work focuses on verification. Ipanema [58] reimplements schedulers in a domain-specific language; Lawall *et al.* verify individual policies [56, 57]. Others simulate schedulers in userspace: LinSched and Apple's XNU scheduler test [6].

Controlled testing. Controlled testing has long been used to expose concurrency bugs [29, 33, 37, 38, 59, 72, 78, 79]. Fray [59] is a platform for performing push-button concurrency testing of JVM programs. Prior tools, such as Chess [79] and delay-bounding [33], search for bugs by bounding and exploring user-level interleavings. Controlled concurrency testing is orthogonal to kSTEP because these systems use their own schedulers to enforce interleaving rather than exercising Linux scheduler code paths.

Record and replay. Application-level record-and-replay systems record all nondeterministic inputs and internal sources of nondeterminism, such as concurrency [14, 15, 21, 25–27, 66, 75, 80, 81, 91]. Whole-machine record-and-replay techniques record every nondeterministic event along the VM boundary, allowing its entire execution to be replayed exactly [4, 22, 30–32, 86, 90, 106]. These techniques can replay previously recorded bugs but offer little help in triggering time-sensitive ones. kSTEP further lets users control scheduler state with its events, making such bugs significantly easier to expose. Also, kSTEP does not control whole-machine nondeterminism; it targets deterministic scheduler execution.

CPU scheduling policies. Many scheduling policies target different objectives [11, 20, 39, 47, 50, 55, 73, 82, 85]. We do not evaluate how these policies perform for particular workloads; instead, we check whether the implementation aligns with the intended design of the CPU scheduler.

8 Conclusion

In this paper, we characterize CPU scheduler bugs and find that they are important, yet hard to observe and trigger. We introduce kSTEP to precisely control scheduler behavior through deterministic tests, producing noise-free, repeatable traces. With kSTEP, we are able to reproduce existing bugs and uncover new ones, showing its promise as a foundation for systematic scheduler testing.

Acknowledgments

We thank our shepherd, the reviewers of OSDI '26, and students in ADSL for their valuable feedback. We are grateful to Suyan Qu and Junxuan Liao from ADSL for their careful proofreading before the submission deadline. This work was supported by NSF grant CNS-2402859 and the taxpayers of Wisconsin and the USA. Caeden Whitaker was supported by the NSF Graduate Research Fellowship Program under Grant No. 2137424, and the Graduate School and the Office of the Vice Chancellor for Research at UW-Madison through the Wisconsin Alumni Research Foundation. Any opinions, findings, conclusions, and recommendations expressed do not necessarily reflect the views of NSF or any other institutions.

References

[1] Docker Project. https://www.docker.io/, 2015.

[2] cgroups - Linux Control Groups. https://man7.org/linux/man-pages/man7/cgroups.7.html, May 2023.

[3] Akamai Technologies, Inc. Akamai online retail performance report: Milliseconds are critical — web performance analytics show even 100-millisecond delays can impact customer engagement and online revenue. Press Release, Akamai Technologies, Inc., 2017. April 18, 2017. Cambridge, MA, USA. Available online: https://www.akamai.com/newsroom/press-release/akamai-releases-spring-2017-state-of-online-retail-performance-report.

[4] Antithesis Operations LLC. Antithesis: autonomous software testing. https://antithesis.com/.

[5] Apache Cassandra Project. Testing — apache cassandra development. https://cassandra.apache.org/_/development/testing.html, 2025. Accessed: 2025-09-16.

[6] Apple Inc. Apple xnu kernel scheduler test harness. https://github.com/apple-oss-distributions/xnu/tree/main/tests/sched/sched_test_harness, 2025.

[7] ARM. Multi-chip modules or modular design chiplets for custom silicon. https://www.arm.com/markets/chiplets, 2025.

[8] Remzi. Arpaci-Dusseau and Andrea Arpaci-Dusseau. Fail-stutter fault tolerance. In *Proceedings Eighth Workshop on Hot Topics in Operating Systems*, pages 33–38, 2001.

[9] Remzi Arpaci-Dusseau and Andrea Arpaci-Dusseau. *Operating Systems: Three Easy Pieces*. 2018.

[10] Giacomo Bagnoli and Vincent Guittot. Rt-app - real-time application suite. https://github.com/scheduler-tools/rt-app, 2010.

[11] Adam Belay, George Prekas, Ana Klimovic, Samuel Grossman, Christos Kozyrakis, and Edouard Bugnion. IX: a protected dataplane operating system for high throughput and low latency. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 49–65, 2014.

[12] Dharma Bellamkonda. Issue 2623: CFS scheduler bug throttles highly threaded i/o blocked applications in kubernetes. https://github.com/coreos/bugs/issues/2623, 2019.

[13] Fabrice Bellard. QEMU, a fast and portable dynamic translator. In *USENIX Annual Technical Conference (ATC)*, 2005.

[14] Tom Bergan, Owen Anderson, Joseph Devietti, Luis Ceze, and Dan Grossman. Coredet: A compiler and runtime system for deterministic multithreaded execution. In *Proceedings of the fifteenth International Conference on Architectural support for programming languages and operating systems*, pages 53–64, 2010.

[15] Emery D Berger, Ting Yang, Tongping Liu, and Gene Novark. Grace: Safe multithreaded programming for C/C++. In *Proceedings of the 24th ACM SIGPLAN conference on Object oriented programming systems languages and applications*, pages 81–96, 2009.

[16] Justinien Bouron, Sebastien Chevalley, Baptiste Lepers, Willy Zwaenepoel, Redha Gouicem, Julia Lawall, Gilles Muller, and Julien Sopena. The battle of the schedulers: FreeBSD ULE vs. Linux CFS. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 85–96, Boston, MA, July 2018. USENIX Association.

[17] bpftrace contributors. bpftrace. https://github.com/bpftrace/bpftrace, 2025. Accessed: 2025-12-08.

[18] Nathan Bronson, Zach Amsden, George Cabrera, Prasad Chakka, Peter Dimov, Hui Ding, Jack Ferris, Anthony Giardullo, Sachin Kulkarni, Harry Li, Mark Marchukov, Dmitri Petrov, Lovro Puzar, Yee Jiun Song, and Venkat Venkataramani. Tao: Facebook's distributed data store for the social graph. In *Proceedings of the 2013 USENIX Conference on Annual Technical Conference*, USENIX ATC'13, page 49–60, USA, 2013. USENIX Association.

[19] John M Calandrino, Dan P Baumberger, Tong Li, Jessica C Young, and Scott Hahn. Linsched: The Linux scheduler simulator. In *PDCCS*, pages 171–176, 2008.

[20] Tingjia Cao, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, and Tyler Caraza-Harter. Making serverless Pay-For-Use a reality with leopard. In *22nd USENIX Symposium on Networked Systems Design and Implementation (NSDI 25)*, pages 189–204, Philadelphia, PA, April 2025. USENIX Association.

[21] Yufei Chen and Haibo Chen. Scalable deterministic replay in a parallel full-system emulator. *SIGPLAN Not.*, 48(8):207–218, February 2013.

[22] Yufei Chen and Haibo Chen. Scalable deterministic replay in a parallel full-system emulator. In *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '13,

page 207–218, New York, NY, USA, 2013. Association for Computing Machinery.

[23] Pat Cody. [patch] sched/fair: Add null pointer check to pick_next_entity(). https://lore.kernel.org/all/20250320205310.779888-1-pat@patcody.io/T/#u, 2025.

[24] Jonathan Corbet. The long road to lazy preemption. *LWN.net*, oct 2024. Accessed: 2025-09-14.

[25] Heming Cui. Stable deterministic multithreading through schedule memoization. In *9th USENIX Symposium on Operating Systems Design and Implementation (OSDI 10)*, 2010.

[26] Heming Cui, Jiri Simsa, Yi-Hong Lin, Hao Li, Ben Blum, Xinan Xu, Junfeng Yang, Garth A Gibson, and Randal E Bryant. Parrot: A practical runtime for deterministic, stable, and reliable threads. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 388–405, 2013.

[27] Heming Cui, Jingyue Wu, John Gallagher, Huayang Guo, and Junfeng Yang. Efficient deterministic multithreading through schedule relaxation. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, pages 337–351, 2011.

[28] Jeffrey Dean and Luiz André Barroso. The tail at scale. *Commun. ACM*, 56(2):74–80, February 2013.

[29] Ankush Desai, Shaz Qadeer, and Sanjit A. Seshia. Systematic testing of asynchronous reactive systems. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2015, page 73–83, New York, NY, USA, 2015. Association for Computing Machinery.

[30] Brendan Dolan-Gavitt, Josh Hodosh, Patrick Hulin, Tim Leek, and Ryan Whelan. Repeatable reverse engineering with panda. In *Proceedings of the 5th Program Protection and Reverse Engineering Workshop*, PPREW-5, New York, NY, USA, 2015. Association for Computing Machinery.

[31] George W. Dunlap, Samuel T. King, Sukru Cinar, Murtaza A. Basrai, and Peter M. Chen. Revirt: enabling intrusion analysis through virtual-machine logging and replay. *SIGOPS Oper. Syst. Rev.*, 36(SI):211–224, December 2003.

[32] George W. Dunlap, Dominic G. Lucchetti, Michael A. Fetterman, and Peter M. Chen. Execution replay of multiprocessor virtual machines. In *Proceedings of the Fourth ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE '08, page 121–130, New York, NY, USA, 2008. Association for Computing Machinery.

[33] Michael Emmi, Shaz Qadeer, and Zvonimir Rakamarić. Delay-bounded scheduling. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '11, page 411–422, New York, NY, USA, 2011. Association for Computing Machinery.

[34] Matt Fleming. Adrestia - scheduler load balancer microbenchmark suite. https://github.com/mfleming/adrestia, 2016.

[35] Matt Fleming. syzkaller - kernel fuzzer. https://github.com/google/syzkaller, 2016.

[36] Matt Fleming. A survey of scheduler benchmarks. https://lwn.net/Articles/725238/, 2017.

[37] Pedro Fonseca, Cheng Li, and Rodrigo Rodrigues. Finding complex concurrency bugs in large multithreaded applications. In *Proceedings of the Sixth Conference on Computer Systems*, EuroSys '11, page 215–228, New York, NY, USA, 2011. Association for Computing Machinery.

[38] Pedro Fonseca, Rodrigo Rodrigues, and Björn B. Brandenburg. SKI: Exposing kernel concurrency bugs through systematic schedule exploration. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 415–431, Broomfield, CO, October 2014. USENIX Association.

[39] Joshua Fried, Zhenyuan Ruan, Amy Ousterhout, and Adam Belay. Caladan: Mitigating interference at microsecond timescales. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 281–297, 2020.

[40] Thomas Gleixner, Clark Williams, and John Kacur. cyclictest - clock_nanosleep latency detection. https://wiki.linuxfoundation.org/realtime/documentation/howto/tools/cyclictest, 2016.

[41] GNU Binutils. addr2line – convert addresses into file names and line numbers. https://sourceware.org/binutils/docs/binutils/addr2line.html. Accessed: 2026-06-07.

[42] Greg Kroah-Hartman and contributors. Linux kernel stable tree. https://github.com/gregkh/linux, 2025. Accessed: 2025-12-08.

[43] Haryadi S. Gunawi, Riza O. Suminto, Russell Sears, Casey Golliher, Swaminathan Sundararaman, Xing Lin, Tim Emami, Weiguang Sheng, Nematollah Bidokhti, Caitie McCaffrey, Deepthi Srinivasan, Biswaranjan Panda, Andrew Baptist, Gary Grider, Parks M. Fields, Kevin Harms, Robert B. Ross, Andree Jacobson,

Robert Ricci, Kirk Webb, Peter Alvaro, H. Birali Runesha, Mingzhe Hao, and Huaicheng Li. Fail-slow at scale: Evidence of hardware performance faults in large production systems. *ACM Trans. Storage*, 14(3), October 2018.

[44] Ke Han, Sruthi P C, Yayu Wang, Yaoxu Song, Bishal Basak Papan, Junwen Yang, Pedro Fonseca, and Yongle Zhang. UpFuzz: Detecting data format incompatibility bugs during distributed storage system upgrade. In *23rd USENIX Symposium on Networked Systems Design and Implementation (NSDI 26)*, pages 1225–1242, Renton, WA, May 2026. USENIX Association.

[45] Peng Huang, Chuanxiong Guo, Lidong Zhou, Jacob R. Lorch, Yingnong Dang, Murali Chintalapati, and Randolph Yao. Gray failure: The Achilles' heel of cloud-scale systems. In *Proceedings of the 16th Workshop on Hot Topics in Operating Systems*, HotOS '17, pages 150–155, New York, NY, USA, May 2017. ACM.

[46] Intel. Intel xeon processor scalable family technical overview. https://www.intel.com/content/www/us/en/developer/articles/technical/xeon-processor-scalable-family-technical-overview.html, 2025.

[47] Călin Iorgulescu, Reza Azimi, Youngjin Kwon, Sameh Elnikety, Manoj Syamala, Vivek Narasayya, Herodotos Herodotou, Paulo Tomita, Alex Chen, Jack Zhang, et al. PerfIso: Performance isolation for commercial Latency-Sensitive services. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 519–532, 2018.

[48] Guoliang Jin, Linhai Song, Xiaoming Shi, Joel Scherpelz, and Shan Lu. Understanding and detecting real-world performance bugs. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '12, page 77–88, New York, NY, USA, 2012. Association for Computing Machinery.

[49] Kostis Kaffes, Jack Tigar Humphries, David Mazières, and Christos Kozyrakis. Syrup: User-defined scheduling across the stack. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, SOSP '21, page 605–620, New York, NY, USA, 2021. Association for Computing Machinery.

[50] Antoine Kaufmann, Tim Stamler, Simon Peter, Naveen Kr Sharma, Arvind Krishnamurthy, and Thomas Anderson. Tas: Tcp acceleration as an os service. In *Proceedings of the Fourteenth EuroSys Conference 2019*, pages 1–16, 2019.

[51] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. sel4: formal verification of an os kernel. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, SOSP '09, page 207–220, New York, NY, USA, 2009. Association for Computing Machinery.

[52] Kubernetes. Pod Resource Management. https://kubernetes.io/docs/concepts/configuration/manage-resources-containers/, August 2014.

[53] Paul Larson. Testing Linux with the Linux test project. In *Ottawa Linux Symposium*, volume 108, 2002.

[54] Julia Lawall, Harsh Chhaya-Shailesh, Jean-Pierre Lozi, and Gilles Muller. Graphing tools for scheduler tracing. https://inria.hal.science/hal-04001993, 2023. Accessed: 2025-12-08.

[55] Julia Lawall, Himadri Chhaya-Shailesh, Jean-Pierre Lozi, Baptiste Lepers, Willy Zwaenepoel, and Gilles Muller. Os scheduling with nest: Keeping tasks close together on warm cores. In *Proceedings of the Seventeenth European Conference on Computer Systems*, pages 368–383, 2022.

[56] Julia Lawall, Keisuke Nishimura, and Jean-Pierre Lozi. Should we balance? towards formal verification of the Linux kernel scheduler. In *Static Analysis: 31st International Symposium, SAS 2024, Pasadena, CA, USA, October 20–22, 2024, Proceedings*, page 194–215, Berlin, Heidelberg, 2024. Springer-Verlag.

[57] Julia Lawall, Keisuke Nishimura, and Jean-Pierre Lozi. Understanding Linux kernel code through formal verification: A case study of the task-scheduler function select_idle_core. In *Proceedings of the Workshop Dedicated to Olivier Danvy on the Occasion of His 64th Birthday*, pages 94–105, 2025.

[58] Baptiste Lepers, Redha Gouicem, Damien Carver, Jean-Pierre Lozi, Nicolas Palix, Maria-Virginia Aponte, Willy Zwaenepoel, Julien Sopena, Julia Lawall, and Gilles Muller. Provable multicore schedulers with ipanema: application to work conservation. EuroSys '20, 2020.

[59] Ao Li, Byeongjee Kang, Vasudev Vikram, Isabella Laybourn, Samvid Dharanikota, Shrey Tiwari, and Rohan Padhye. Fray: An efficient general-purpose concurrency testing platform for the jvm. *Proceedings of the ACM on Programming Languages*, 9(OOPSLA2):4035–4063, 2025.

[60] Jiaxin Li, Yiming Zhang, Shan Lu, Haryadi S. Gunawi, Xiaohui Gu, Feng Huang, and Dongsheng Li. Performance bug analysis and detection for distributed storage and computing systems. *ACM Trans. Storage*, 19(3), June 2023.

[61] Linus Torvalds and contributors. Linux kernel source tree. https://github.com/torvalds/linux, 2025. Accessed: 2025-12-08.

[62] Linux Kernel contributors. include/trace/events/sched.h Linux kernel v6.17.8 source. https://elixir.bootlin.com/linux/v6.17.8/source/include/trace/events/sched.h, 2025. Accessed: 2025-12-08.

[63] Linux Kernel developers. Module kallsyms support – kernel/module/kallsyms.c. https://elixir.bootlin.com/linux/v7.0.11/source/kernel/module/kallsyms.c. Accessed: 2026-06-08.

[64] Linux Kernel developers. SCHED_SOFTIRQ – Linux kernel softirq. https://elixir.bootlin.com/linux/latest/source/include/linux/interrupt.h. Accessed: 2025-12-08.

[65] Linux Kernel developers. smp_call_function_single – Linux kernel SMP API. https://elixir.bootlin.com/linux/latest/source/kernel/smp.c. Accessed: 2025-12-08.

[66] Tongping Liu, Charlie Curtsinger, and Emery D Berger. Dthreads: efficient deterministic multithreading. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, pages 327–336, 2011.

[67] Chang Lou, Yuzhuo Jing, and Peng Huang. Demystifying and checking silent semantic violations in large distributed systems. In *Proceedings of the 16th USENIX Symposium on Operating Systems Design and Implementation*, OSDI '22, pages 91–107, Carlsbad, CA, USA, July 2022. USENIX Association.

[68] Chang Lou, Dimas Shidqi Parikesit, Yujin Huang, Zhewen Yang, Senapati Diwangkara, Yuzhuo Jing, Achmad Imam Kistijantoro, Ding Yuan, Suman Nath, and Peng Huang. Deriving semantic checkers from tests to detect silent failures in production distributed systems. In *Proceedings of the 19th USENIX Symposium on Operating Systems Design and Implementation*, OSDI '25, Boston, MA, USA, July 2025. USENIX Association.

[69] Jean-Pierre Lozi, Baptiste Lepers, Justin Funston, Fabien Gaud, Vivien Quéma, and Alexandra Fedorova. The Linux scheduler: a decade of wasted cores. In *Proceedings of the Eleventh European Conference on Computer Systems*, pages 1–16, 2016.

[70] Lanyue Lu, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, and Shan Lu. A study of Linux file system evolution. In *11th USENIX Conference on File and Storage Technologies (FAST 13)*, pages 31–44, San Jose, CA, February 2013. USENIX Association.

[71] Ruiming Lu, Yunchi Lu, Yuxuan Jiang, Guangtao Xue, and Peng Huang. One-size-fits-none: Understanding and enhancing slow-fault tolerance in modern distributed systems. In *Proceedings of the 22nd USENIX Symposium on Networked Systems Design and Implementation*, NSDI '25, pages 359–378. USENIX Association, April 2025.

[72] Shan Lu, Soyeon Park, Eunsoo Seo, and Yuanyuan Zhou. Learning from mistakes: a comprehensive study on real world concurrency bug characteristics. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XIII, page 329–339, New York, NY, USA, 2008. Association for Computing Machinery.

[73] Michael Marty, Marc de Kruijf, Jacob Adriaens, Christopher Alfeld, Sean Bauer, Carlo Contavalli, Michael Dalton, Nandita Dukkipati, William C Evans, Steve Gribble, et al. Snap: A microkernel approach to host networking. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, pages 399–413, 2019.

[74] Chris Mason. schbench - scheduler benchmark. https://git.kernel.org/pub/scm/linux/kernel/git/mason/schbench.git/, 2016.

[75] Timothy Merrifield, Sepideh Roghanchi, Joseph Devietti, and Jakob Eriksson. Lazy determinism for faster deterministic multithreading. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 879–891, 2019.

[76] Barton P. Miller, Lars Fredriksen, and Bryan So. An empirical study of the reliability of unix utilities. *Communications of the ACM*, 33(12):32–44, December 1990.

[77] Ingo Molnar. CFS scheduler. https://docs.kernel.org/scheduler/sched-design-CFS.html, 2007.

[78] Suvam Mukherjee, Pantazis Deligiannis, Arpita Biswas, and Akash Lal. Learning-based controlled concurrency testing. *Proc. ACM Program. Lang.*, 4(OOPSLA), November 2020.

[79] Madanlal Musuvathi, Shaz Qadeer, Thomas Ball, Gerard Basler, Piramanayagam Arumuga Nainar, and Iulian Neamtiu. Finding and reproducing heisenbugs in concurrent programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI'08, page 267–280, USA, 2008. USENIX Association.

[80] Robert O'Callahan, Chris Jones, Nathan Froyd, Kyle Huey, Albert Noll, and Nimrod Partush. Engineering record and replay for deployability. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, pages 377–389, Santa Clara, CA, July 2017. USENIX Association.

[81] Marek Olszewski, Jason Ansel, and Saman Amarasinghe. Kendo: efficient deterministic multithreading in software. In *Proceedings of the 14th international conference on Architectural support for programming languages and operating systems*, pages 97–108, 2009.

[82] Amy Ousterhout, Joshua Fried, Jonathan Behrens, Adam Belay, and Hari Balakrishnan. Shenango: Achieving high CPU efficiency for latency-sensitive datacenter workloads. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 361–378, 2019.

[83] PostgreSQL Global Development Group. Committing checklist. https://wiki.postgresql.org/wiki/Committing_checklist, 2025. Accessed: 2025-09-16.

[84] The Fedora Project. Container optimized os. https://www.fedoraproject.org/coreos, 2025.

[85] Henry Qin, Qian Li, Jacqueline Speiser, Peter Kraft, and John Ousterhout. Arachne:Core-Aware thread management. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 145–160, 2018.

[86] Shiru Ren, Le Tan, Chunqi Li, Zhen Xiao, and Weijia Song. Samsara: Efficient deterministic replay in multiprocessor environments with hardware virtualization extensions. In *2016 USENIX Annual Technical Conference (USENIX ATC 16)*, pages 551–564, Denver, CO, June 2016. USENIX Association.

[87] Matthew Ruffell. sched: Prevent CPU lockups when task groups take longer than the period. https://bugs.launchpad.net/bugs/1836971, 2019.

[88] Rusty Russell, Yanmin Zhang, Ingo Molnar, and David Sommerseth. hackbench - scheduler benchmark/stress test. https://wiki.linuxfoundation.org/realtime/documentation/howto/tools/hackbench, 2010.

[89] Prateek Sharma, Lucas Chaufournier, Prashant Shenoy, and Y. C. Tay. Containers and virtual machines at scale: A comparative study. In *Proceedings of the 17th International Middleware Conference*, Middleware '16, New York, NY, USA, 2016. Association for Computing Machinery.

[90] MXVMJ Sheldon and Ganesh Venkitachalam Boris Weissman. Retrace: Collecting execution trace with virtual machine deterministic replay. In *Proceedings of the Third Annual Workshop on Modeling, Benchmarking and Simulation (MoBS 2007)*, 2007.

[91] Sudarshan M Srinivasan, Srikanth Kandula, Christopher R Andrews, Yuanyuan Zhou, et al. Flashback: A lightweight extension for rollback and deterministic replay for software debugging. In *USENIX annual technical conference, general track*, pages 29–44. Boston, MA, USA, 2004.

[92] Xudong Sun, Runxiang Cheng, Jianyan Chen, Elaine Ang, Owolabi Legunsen, and Tianyin Xu. Testing configuration changes in context to prevent production failures. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 735–751. USENIX Association, November 2020.

[93] The kernel development community. Numa. The Linux Kernel documentation (v5.6). https://www.kernel.org/doc/html/v5.6/vm/numa.html, 2020. Accessed: 2025-09-14.

[94] The kernel development community. Deadline scheduling (sched_deadline). The Linux Kernel documentation. https://docs.kernel.org/scheduler/sched-deadline.html, 2025. Accessed: 2025-09-14.

[95] The kernel development community. Extensible scheduler class. The Linux Kernel documentation. https://docs.kernel.org/scheduler/sched-ext.html, 2025. Accessed: 2025-09-14.

[96] The kernel development community. Scheduler capacity. The Linux Kernel documentation. https://docs.kernel.org/scheduler/sched-capacity.html, 2025. Accessed: 2025-09-14.

[97] The kernel development community. Scheduler energy awareness. The Linux Kernel documentation. https://docs.kernel.org/scheduler/sched-energy.html, 2025. Accessed: 2025-09-14.

[98] The Linux Kernel documentation contributors. ftrace - Function Tracer — Linux Kernel documentation. https://docs.kernel.org/trace/ftrace.html. Accessed: 2026-06-08.

[99] The Linux Kernel documentation contributors. Tracepoints — Linux Kernel documentation. https://www.kernel.org/doc/html/v5.4/trace/tracepoints.html, 2020. Accessed: 2025-12-08.

[100] The LLVM Project. Sanitizercoverage. https://clang.llvm.org/docs/SanitizerCoverage.html, 2025.

[101] Traceshark contributors. Traceshark. https://github.com/cunctator/traceshark, 2025. Accessed: 2025-12-08.

[102] Alexandre Verbitski, Anurag Gupta, Debanjan Saha, Murali Brahmadesam, Kamal Gupta, Raman Mittal, Sailesh Krishnamurthy, Sandor Maurice, Tengiz Kharatishvili, and Xiaofeng Bao. Amazon aurora: Design considerations for high throughput cloud-native relational databases. In *Proceedings of the 2017 ACM International Conference on Management of Data*, SIGMOD '17, page 1041–1052, New York, NY, USA, 2017. Association for Computing Machinery.

[103] Abhishek Verma, Luis Pedrosa, Madhukar Korupolu, David Oppenheimer, Eric Tune, and John Wilkes. Large-scale cluster management at google with borg. In *Proceedings of the Tenth European Conference on Computer Systems*, EuroSys '15, New York, NY, USA, 2015. Association for Computing Machinery.

[104] Junfeng Yang, Heming Cui, Jingyue Wu, Yang Tang, and Gang Hu. Making parallel programs reliable with stable multithreading. *Communications of the ACM*, 57(3):58–69, 2014.

[105] Ding Yuan, Yu Luo, Xin Zhuang, Guilherme Renna Rodrigues, Xu Zhao, Yongle Zhang, Pranay U. Jain, and Michael Stumm. Simple testing can prevent most critical failures: An analysis of production failures in distributed Data-Intensive systems. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 249–265, Broomfield, CO, October 2014. USENIX Association.

[106] Tianren Zhang, Sishuai Gong, and Pedro Fonseca. Krr: efficient and scalable kernel record replay. In *Proceedings of the 19th USENIX Conference on Operating Systems Design and Implementation*, OSDI '25, USA, 2025. USENIX Association.

[107] Shawn Wanxiang Zhong, Jing Liu, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. Revealing the unstable foundations of ebpf-based kernel extensions. In *Proceedings of the Twentieth European Conference on Computer Systems*, EuroSys '25, page 21–41, New York, NY, USA, 2025. Association for Computing Machinery.

[108] Peter Zijlstra. Eevdf scheduler. https://docs.kernel.org/scheduler/sched-eevdf.html, 2024.

[109] Peter Zijlstra. [patch 17/24] sched/fair: Implement delayed dequeue. https://lore.kernel.org/all/20240727105030.226163742@infradead.org/, 2024.

What Are You (M)Waiting For: The Hidden Cost of Idle in the Hyperscale Cloud (Operational Systems)

Yun Wang[♠], Xingguo Jia[♢], Ben Luo[♢], Kenan Liu[♢], Shengdong Dai[♢], Jingdong Han[♢], Weihao Chen[♢], Yicheng Gu[♠], Xingzi Yu[♠], Yibin Shen[♢], Jiesheng Wu[♢], Zhengwei Qi[♠]*, Haibing Guan[♠]

[♠]*Shanghai Jiao Tong University* [♢]*Alibaba Group*

Abstract

Oversubscription is central to large-scale clouds: multiplexing virtual CPUs (vCPUs) over fewer physical CPUs (pCPUs) improves utilization and sales density, but meeting strict latency Service Level Objectives (SLOs) requires precise control over idle behavior. In 1:1 settings, idle-passthrough—especially `mwait`-passthrough—works well: by allowing guests to initiate hardware idle transitions directly, it eliminates idle-induced Virtual Machine (VM) exits and achieves near–bare-metal latency.

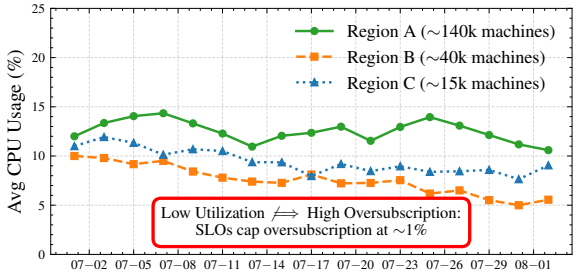
In production oversubscribed environments, however, passthrough breaks down. Because the hypervisor cannot observe `mwait` idleness, a vCPU never yields its pCPU, causing idle vCPU to monopolize cores and driving up contention, steal time ratios, live migrations, and SLO alarms across regions. Controlled experiments reproduce these production symptoms: **even an idle vCPU executing `mwait` can raise colocated tail latency by up to 3×.**

We present `mwait`-sched, a virtualization-aware redesign of `mwait` handling that reconciles bare-metal-like idle latency with predictable pCPU multiplexing. `mwait`-sched integrates deterministic timer-based emulation, fine-grained idle-interval classification, and a scalable multi-address `mwait`-proxy that restores idle visibility without frequent VM exits. Across nine representative workloads, it reduces P99 (**99$^{\text{th}}$** percentile) latency by 30–50% and reduces steal ratio by 30–40%. At hyperscale, across globally distributed production regions comprising ∼ 3.2M pCPUs, it reduces high-contention steal events by over 80%, cuts daily live migrations by 30–50%, and raises oversubscription ratio from 1.0% to 20.3%, effectively adding ∼ 600,000 vCPUs of sellable capacity.

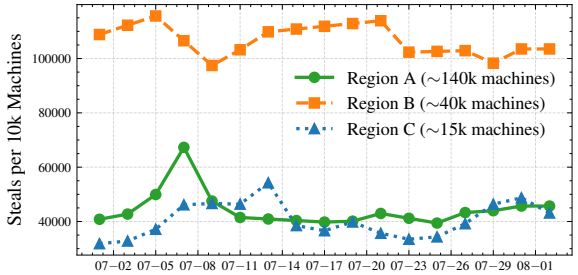
1 Introduction

Oversubscription has become a common and effective strategy in large-scale cloud platforms, allowing providers to increase resource utilization and improve overall fleet efficiency [9, 15, 16, 22, 29, 36, 38, 45, 54]. By multiplexing virtual CPUs (vCPUs) over a smaller pool of physical CPUs (pCPUs), Cloud Service Providers (CSPs) can significantly raise oversubscription ratios without impacting most workloads.

At the same time, strict latency Service Level Objectives (SLOs) remain central to cloud service guarantees. For latency-sensitive applications, even small scheduling delays can translate into SLO violations [5, 32, 40, 51]. The x86 `mwait` instruction is the standard low-latency idle path on modern CPUs: paired with `monitor`, it places a core into a



(a) Average CPU utilization across three production regions.



(b) Steal events across the region, where a VM's steal time ratio exceeds 5% in a 3-second sampling interval.

Figure 1: Cluster-level traces from three production regions. Although utilization is low, oversubscription is capped at ∼1% because higher selling would trigger contention and violate latency SLOs.

*Corresponding author.

low-power C-state and wakes on a write to a monitored memory location, avoiding the IPI round-trip that interrupt-driven idle paths like `hlt` require. In non-oversubscribed (1:1) settings, `mwait`-passthrough offers an appealing solution (details in §3): by allowing the guest to directly drive hardware idle transitions, it eliminates Virtual Machine (VM) exits on idle instructions and substantially lowers idle-exit latency, thereby improving tail performance [24].

However, when applied to oversubscribed environments, this same mechanism introduces critical problems. Instead of enabling efficient multiplexing, `mwait`-passthrough disrupts pCPU sharing and leads to escalating contention signals across production regions.

To understand contention in our production environment, we analyzed three regions representing large, medium, and small deployments. Figure 1a shows VM-level CPU utilization across these clusters. Despite uniformly low utilization, oversubscription remains limited to only about **1%**, because selling additional capacity would amplify contention and violate latency SLOs. At million-core scale, each additional percentage point of oversubscription yields roughly 10,000 vCPUs of additional sellable capacity.

To quantify contention, we use the steal-time ratio, the fraction of time a vCPU is ready to run but cannot obtain pCPU resources, and treat 5% as the operational threshold for noticeable interference [34].

Figure 1b shows that, even under similarly low utilization, regions differ by more than an order of magnitude in the fraction of VMs exceeding this threshold. Unlike utilization, rising steal-time ratios directly indicate performance degradation. This divergence demonstrates that utilization is decoupled from interference, making it an unreliable signal for safe vCPU placement.

A deeper investigation revealed the root cause. Despite its excellent latency characteristics, `mwait`-passthrough is fundamentally incompatible with pCPU sharing because its **native semantics do not translate into virtualization.** On bare metal, `mwait` is a true blocking primitive: the processor enters a deep C-state and relinquishes execution until a microarchitectural wake event occurs. In virtualization, however, this hardware-level blocking cannot be observed or mediated by the hypervisor. As a result, a passthrough vCPU never performs a schedulable yield. From the host scheduler's viewpoint, it remains continuously runnable and is therefore accounted as 100% busy. This semantic gap—between hardware-defined idleness and software-visible runnability—causes passthrough vCPUs to monopolize their pCPUs under oversubscription, amplifying contention, delaying colocated vCPUs, and destabilizing latency-sensitive workloads. This leaves us with a simple goal: keep `mwait`'s low wake-up latency, but make idle vCPUs visible to the host scheduler when pCPUs are shared. We introduce `mwait`-sched to meet this goal through three mechanisms:

1. **Profile-guided timer-based `mwait` emulation.** A hypervisor timer regains control from idle vCPUs at slice intervals tuned per workload from PMU counters (short slices of 20–50 µs for I/O-heavy services, longer for CPU-bound), restoring idle visibility to the host scheduler without per-instruction trap-and-emulate cost.

2. **Bimodal-idle vCPU aggregation.** `mwait` idle durations follow a bimodal distribution—transient (μs-scale spinlock backoff) versus stable (ms-scale quiescence); `mwait`-sched aggregates only stable-idle vCPUs onto shared pCPUs, raising density without making transient-idle interference worse.

3. **Multi-address `mwait`-proxy.** A hypervisor-side linked list of monitored addresses, scanned on every hypervisor entry, replaces per-vCPU timers when several vCPUs share a pCPU; this variant scales beyond two-vCPU colocation and is deployed on the burst-instance fleet.

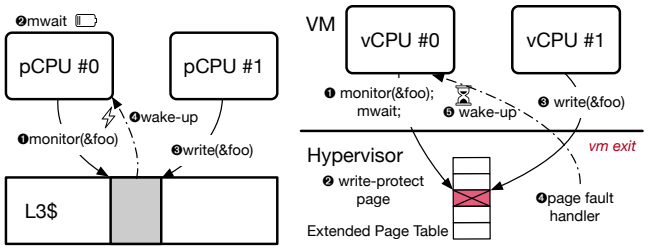
We deployed `mwait`-sched in the production fleet of a leading global cloud provider, covering $\sim 3.2M$ pCPU cores. In production, it reduces average daily alarms by 61.5%. In targeted workloads that match our main deployment scenarios, it reduces latency by 60% on average. To our knowledge, this is the first work to systematically study and optimize idle instructions in large-scale cloud environments.

2 Background

2.1 vCPU Scheduling

Colocation for Latency-intensive Workloads. Cloud workloads exhibit substantial variability in CPU demand, prompting cloud providers to increase pCPU utilization by multiplexing multiple vCPUs on fewer physical cores. Prior studies examine how to pack heterogeneous tasks while balancing contention, latency, and utilization [4, 9, 14, 21, 31]. A long line of work explores colocating latency-critical (LC) and best-effort tasks to increase aggregate efficiency [13, 17, 29, 33, 48]. In production, however, even colocating idle vCPUs with LC workloads can violate SLOs: idle vCPUs create unpredictable wake-up interference that breaks LC latency. Consequently, CSPs adopt *exclusive core allocation* for LC workloads, avoiding any form of colocation.

A key mechanism enabling safe oversubscription is the use of idle instructions as cross-layer scheduling signals. Unlike bare-metal systems, where idle instructions serve solely as power-management hints, virtualized environments reinterpret them as opportunities for vCPU descheduling: the hypervisor treats a guest idle event as an explicit yield, reclaiming pCPU cycles for runnable vCPUs [5, 33, 54]. Thus, accurate idle detection is fundamental to high-density multiplexing.



(a) Bare-metal `mwait` (b) Virtualized `mwait`

Figure 2: Bare-metal vs. virtualized `mwait`. Hardware monitoring enables fast, cache-line–granular wake-up. Virtualization must use coarse, high-overhead mechanisms such as page-protected emulation.

2.2 Idle Instructions

Modern CPUs provide `hlt` and `mwait` to enter idle states with varying power and latency characteristics [41, 50]. `hlt` halts execution until an interrupt arrives, whereas `mwait` waits on monitored memory events and enables deeper C-states with lower exit latency [23, 55]. These properties make `mwait` an appealing candidate for latency-sensitive workloads.

However, `mwait` semantics fundamentally conflict with virtualization. The instruction depends on microarchitectural monitoring and cache-line–granularity wake-up signals that software-based hypervisors cannot observe or emulate [42]. Virtualized clouds rely on idle events to detect quiescent vCPUs and reassign their pCPUs; `hlt` supports this by triggering a VM exit and allowing the hypervisor to suspend the vCPU. But these VM exits impose substantial overhead that inflates tail latency [6, 25, 28].

`mwait` creates even deeper challenges. Upstream KVM, lacking hardware visibility, treats intercepted `monitor`/`mwait` as no-ops, forcing the vCPU to spin and preventing the hypervisor from recognizing idleness [26, 27]. Passthrough allows the guest to execute native `mwait` and enter hardware C-states, but because the vCPU thread remains bound to the pCPU, guest idleness becomes host-level busywait, causing large and unpredictable delays for colocated vCPUs [47]. In short, both no-op emulation and passthrough destroy the scheduling semantics required for efficient pCPU sharing.

Virtualizing `mwait`: Monitoring vs. Software Simulation. Figure 2 illustrates the semantic gap. On bare metal, `monitor` tracks writes to a cache line and wakes the CPU through a microarchitectural event. In virtualization, issuing `monitor`/`mwait` causes a VM exit. The hypervisor must either poll the monitored address—wasting pCPU cycles—or rely on page-protected emulation, where writes trigger page faults or EPT (Extended Page Tables) violations [18]. This emulation incurs thousands of cycles and operates at page granularity, far coarser than `mwait`'s intended semantics.

Host vs. Guest Semantic Gap. Passthrough further exposes a scheduling mismatch: the guest expects `mwait` to place the CPU in a deep C-state, while the host scheduler expects runnable vCPU threads. When the pCPU enters deep idle, the host loses visibility into the vCPU's status, preventing safe time-slice reallocation and introducing long wake latencies that affect other tenants [47]. These effects directly conflict with oversubscription and predictable tail-latency guarantees.

Security Considerations. Allowing unregulated passthrough of `mwait` in multi-tenant settings introduces denial-of-service and timing-side-channel risks [52]. A malicious VM could force pCPUs into deep C-states, delaying colocated workloads and amplifying latency variance.

Existing Mechanisms and Limitations. Prior mechanisms attempt to reconcile `mwait` with virtualization, but all introduce severe trade-offs:

1. **NOP Simulation**: Trap-and-resume converts guest idle loops into rapid VM exits, preserving wake-up latency but imposing high exit overhead.

2. **`mwait`-Passthrough (non-oversubscribe)**: Allows native C-state transitions but prevents the hypervisor from detecting idleness, making oversubscription unsafe.

3. **`hlt`-Passthrough**: Avoids the trap on `hlt` itself, but every wake-up still arrives as an IPI that traps for emulation and injection, so the hypervisor pays an exit-and-reentry per idle-active transition.

4. **Paravirtualization**: Requires intrusive guest changes and is unsuitable for public clouds.

Across these methods, none preserve low-latency `mwait` semantics while enabling scalable, high-density vCPU aggregation. Existing approaches either abandon hardware behavior, incur prohibitive overheads, or break core-sharing safety.

Summary. Efficiently supporting `mwait` in production-scale virtualized environments remains a challenging and high-impact problem. The instruction's reliance on microarchitectural memory events is fundamentally misaligned with software-based hypervisor control, and both existing emulation and passthrough fail to provide accurate idle detection or safe pCPU sharing. Overcoming this semantic conflict is essential for enabling low-latency workloads and high oversubscription in modern multi-tenant clouds.

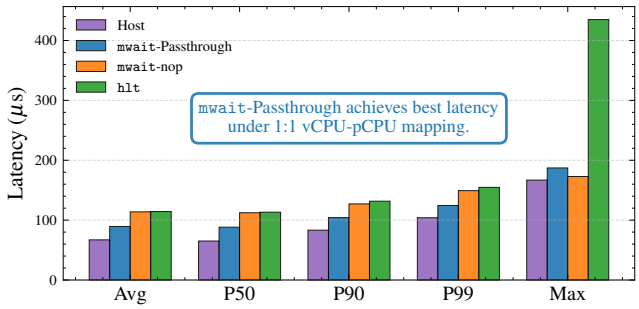


Figure 3: YCSB request latency under four execution configurations (Redis). Among all VM setups, the VM with `mwait` passthrough achieves the lowest latency, while the `hlt`-based VM and the VM using trapped `mwait` both suffer higher latency due to the overhead of trap-and-emulation on idle instructions.

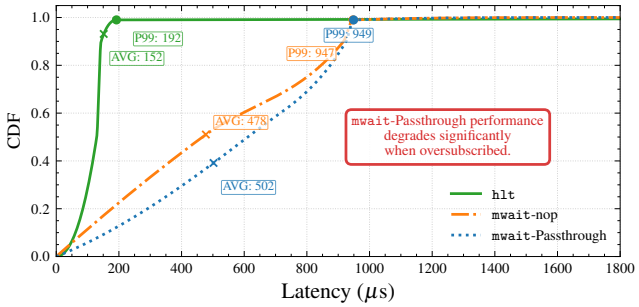


Figure 4: YCSB latency CDF on a latency-critical VM colocated on the same pCPU with an otherwise-idle VM. Three curves vary the idle instruction the colocated VM uses: `hlt`, `mwait`-nop (KVM treats `mwait` as a no-op), and `mwait`-passthrough. Both `mwait` variants degrade the latency-critical VM substantially relative to `hlt`, even though the colocated VM is doing no work, because under both the idle vCPU never yields the pCPU back to the latency-critical workload.

3 Characterization

3.1 Symptoms: Latency Benefits but Severe Colocation Interference

We conducted YCSB (Yahoo! Cloud Serving Benchmark [8]) experiments with four different configurations: (1) Bare-metal, where the system runs directly on physical hardware without any virtualization; (2) VM with `hlt`, where the virtual machine (VM) uses the `hlt` instruction to enter an idle state; (3) VM with `mwait`, where the VM uses the `mwait` instruction to enter a low-power state; and (4) VM with `mwait` passthrough, where `mwait` is passed directly to the guest without any intervention. In configurations 2 and 3, both `hlt` and `mwait` are privileged instructions, so they trigger a **trap and emulation** process. As a result, both setups involve VM exits when the instructions are executed.

Table 1: Comparison of vm exits (exits/s)

	mwait-passthrough	**mwait-nop**	**hlt**
Total	9,953	1,835,258	6,161
Idle-induced	0	1,826,602	1,773

As shown in Figure 3, we observed the impact of different configurations on average read and write latencies, as well as P99 (**99th** percentile) latencies. The Bare-metal configuration, representing the host system, provides the highest performance and serves as the ideal performance for latency.

Among the virtualized configurations, the `mwait`-passthrough VM delivers the lowest latency: its average read and write latencies are roughly 20% lower than those of both the `hlt`-based VM and the `mwait`-nop VM. The improvement also appears in the tail, where passthrough achieves lower P99 latency.

The key difference is that `mwait`-passthrough avoids all idle-induced exits, while `mwait`-nop triggers more than 1.8M and `hlt` incurs 1,773 (Table 1). By allowing the guest to change the hardware idle state without involving the hypervisor, passthrough eliminates the trap–and–resume cycles that dominate exit overhead. This directly accounts for both the roughly 20% reduction in average latency and the improvement in P99 latency.

While `mwait`-passthrough allows the guest to directly issue hardware `mwait`, it becomes problematic once the host CPU is shared. To quantify this effect, we colocate a latency-critical VM with an otherwise-idle VM on the same pCPU and vary the idle instruction the colocated VM uses: `hlt` (the standard, hypervisor-trapped path), `mwait`-nop (KVM's current treatment of guest `mwait`), and `mwait`-passthrough. We then run a YCSB latency test on the loaded VM. As shown in Figure 4, both `mwait` variants cause substantial performance degradation relative to `hlt`, even though the second VM is completely idle. This result highlights a fundamental problem: once a vCPU executing `mwait` occupies a shared pCPU without yielding, any co-located vCPUs can experience severe and unpredictable latency inflation.

3.2 Root Cause: Missing Idle-Exit Signals under Passthrough

Although `mwait`-passthrough substantially reduces VM exits, it also prevents an idle vCPU from yielding control back to the hypervisor. This limitation becomes harmful under oversubscription—the common case in production—where multiple vCPUs share a single pCPU. In such settings, enabling `mwait` passthrough can unintentionally amplify contention and increase latency.

As shown in Figure 5, when there is no oversubscription and each vCPU is pinned to its own pCPU, `mwait`-

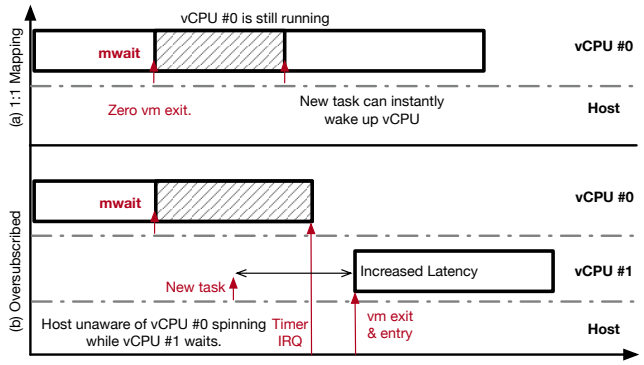


Figure 5: Behavior of `mwait` under passthrough in non-oversubscribed (top) and oversubscribed (bottom) settings. Passthrough preserves low latency when each vCPU owns a dedicated pCPU, but prevents timely yielding under oversubscription, inflating latency for colocated vCPUs.

passthrough behaves benignly. Even if the guest enters an idle loop, no other vCPU competes for the core, and the vCPU will simply resume execution without incurring additional VM exits. This is why passthrough delivers the best latency in 1:1 vCPU–pCPU environments.

The behavior changes fundamentally under oversubscription. Consider the scenario in which vCPU 0 executes `mwait` intending to yield. Because passthrough treats `mwait` as a hardware instruction, the hypervisor receives no signal that the vCPU has become idle. Consequently, the pCPU continues to be "owned" by vCPU 0—even though it is idle—until an unrelated event such as a timer interrupt triggers a scheduling opportunity. Meanwhile, vCPU 1, which is runnable and waiting on the same pCPU, experiences inflated queuing delays. The resulting latency inflation is especially severe for latency-critical workloads colocated with a passthrough-`mwait` VM.

In contrast, if `mwait` is trapped, the hypervisor can immediately reassign the pCPU to vCPU 1, restoring expected scheduling behavior and significantly improving tail latency. This divergence in behavior makes it necessary to handle idle instructions differently depending on whether the system is oversubscribed. In short, while passthrough is ideal for 1:1 mappings, it is detrimental in shared-core settings where timely yielding is essential for maintaining low latency.

4 `mwait`-sched Design

`mwait`-sched is designed to make `mwait` usable in oversubscribed, multi-tenant cloud environments, where naïve passthrough causes uncontrolled contention and existing trap-and-emulate approaches fail to preserve low-latency behavior. The scheduler must protect latency-sensitive workloads, support high consolidation to improve resource utilization, and mitigate interference under colocation. To meet these requirements, we design `mwait`-sched as a hypervisor-level mechanism that preserves the low-exit semantics of `mwait` while giving the hypervisor sufficient visibility to safely multiplex vCPUs on shared pCPUs. We have deployed `mwait`-sched across production regions, where it operates at cloud scale and serves diverse tenant workloads. The remainder of this section describes its core components: profile-guided `mwait` emulation, contention-aware wake-up, `mwait`-based vCPU aggregation, and a scalable multi-address `mwait`-proxy for high-density consolidation.

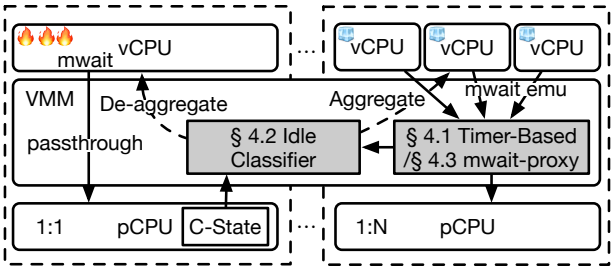


Figure 6: Overview of `mwait`-sched. Active vCPUs run 1:1 with passthrough `mwait` (left); the §4.2 idle classifier aggregates stable-idle vCPUs onto a shared 1:*N* pCPU (right), where `mwait` is emulated by either the §4.1 hypervisor timer or the §4.3 multi-address linked-list proxy.

Figure 6 illustrates the core idea of `mwait`-sched. The hypervisor handles vCPUs in two modes based on how often they idle. Active vCPUs run 1:1 with passthrough `mwait`, which lets the core enter a hardware C-state directly and gives the lowest exit latency. Once a vCPU's idle pattern is classified as *stable* by idle classifier in §4.2, it is aggregated onto a shared 1:*N* pCPU where `mwait` is emulated either by §4.1's periodic hypervisor timer (used for low-density colocation, e.g., dedicated instances at $\leq$1:2) or by §4.3's multi-address linked-list proxy (used at higher density on burst instances, typically $\sim$1:4). When a previously stable vCPU resumes activity, the classifier de-aggregates it back to a dedicated pCPU. Together these mechanisms close the visibility gap of `mwait` under virtualization without sacrificing the low-latency idle path for active workloads.

4.1 Profile-guided `mwait` emulation

As §3 shows, `mwait` passthrough works well only when a vCPU exclusively owns a pCPU, while enabling `mwait` exits is necessary under colocation to avoid uncontrolled contention. A natural first step is therefore to enable passthrough only for exclusive vCPUs. However, this immediately raises a key challenge: how to emulate `mwait` correctly when multiple vCPUs share a pCPU.

As discussed in §2.2, the special semantics of `mwait` make simple trap-and-simulate infeasible. The hypervisor cannot reliably observe the memory writes that should wake a sleeping vCPU, and therefore cannot emulate the hardware monitor-

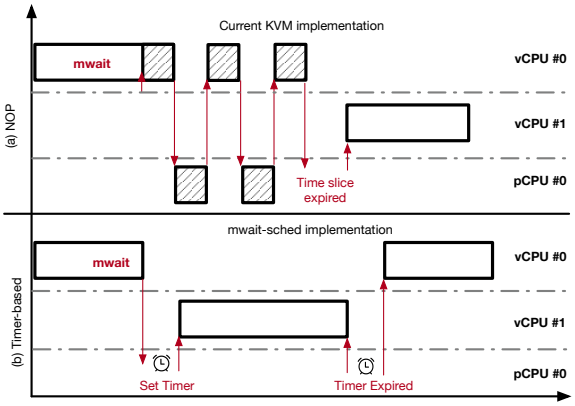


Figure 7: Profile-Guided `mwait` emulation allows vCPUs to yield in a controlled interval, preventing pCPU monopolization and reducing contention under colocation.

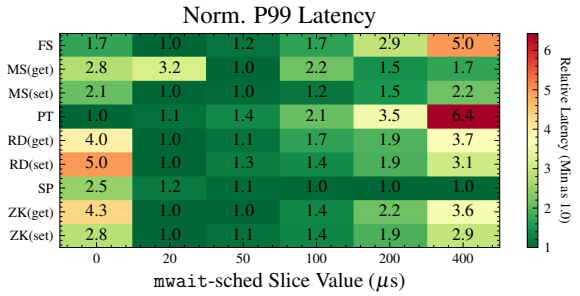


Figure 8: Normalized P99 latency across nine representative workloads (definitions in Table 3) as we vary the `mwait`-sched slice length. Each row is normalized to the minimum P99 observed for that workload (greener is better). All measurements were collected under a 2 vCPU : 1 pCPU colocation setting. Short slices (0–50 µs) favor I/O- and synchronization-heavy services, whereas CPU-bound compute prefers longer slices.

ing behavior. To address this, we introduce a periodic timer that forces control back to the vCPU at fixed intervals. At each timer tick, the vCPU checks whether its wake-up conditions are met, allowing us to provide deterministic `mwait`-like behavior even when vCPUs are colocated.

Figure 7 illustrates the effect of KVM's current no-op implementation. When `mwait` is treated as a NOP, a vCPU never yields and effectively monopolizes the pCPU time slice, causing severe interference for colocated vCPUs.

In `mwait`-sched, we replace the NOP with a timer-based emulation. The timer guarantees that a vCPU regains control within a bounded time window, while allowing the scheduler to allocate the pCPU to other vCPUs between ticks. The length of this window determines the tradeoff between latency and throughput: short slices allow latency-sensitive workloads to regain execution quickly, while overly short slices introduce excessive VM switching overhead and waste CPU cycles in context switching.

To balance this tradeoff, a number of approaches could in principle be adopted—ranging from offline profiling with per-VM tuning, to online machine-learning classifiers, to fine-grained instruction-level instrumentation. However, these techniques introduce nontrivial complexity, operational burden, or deployment risks at hyperscale. For production stability, we adopt a simple and robust solution: a profile-guided `mwait` emulation.

As shown in Figure 8, most latency-sensitive, I/O-heavy workloads (FS, MySQL, Redis, ZooKeeper, and PT) achieve their lowest P99 latency when the `mwait`-sched slice is short, typically in the 20–50 µs range: for example, Redis get/set improve by roughly 4–5× when moving from a 0 µs slice to 20 µs. In contrast, the CPU-intensive `super-pi` benchmark benefits from longer slices, with P99 latency steadily decreasing by about 2.5× as we extend the slice from 0 µs to 400 µs. The mechanism is direct: I/O- and synchronization-heavy workloads idle frequently and are sensitive to wake-up latency, so a shorter slice that resumes them quickly when the wake event arrives reduces queuing delay; `super-pi` rarely idles, so any short-slice timer overhead pays for nothing and just wastes cycles.

eBPF traces of representative vCPUs make this concrete: an I/O-bound vCPU enters `mwait` only ∼19 times/sec but stays idle ∼53 ms each time (∼96% of episodes longer than 1 ms), while a CPU-bound vCPU enters ∼6,900 times/sec with each idle averaging only ∼110 µs (∼95% under 200 µs). The I/O-bound vCPU spends most of its time idle waiting for a request, so each wake aligns with a real I/O or synchronization event and a short slice cuts directly into the request tail. The CPU-bound vCPU's idles are short scheduler-sync gaps with no external wake to align with, so a short slice just adds a timer interrupt per gap without any latency upside.

Because we cannot inspect applications inside customer VMs, we infer workload type by sampling PMU (Performance Monitor Unit [20]) counters and using the ratio of IOPS (Input/Output Operations Per Second) to vCPU utilization as a simple, deployable indicator: high ratios correlate with latency-sensitive workloads, while low ratios correspond to CPU-intensive jobs. Based on this signal, `mwait`-sched dynamically steers such workloads toward shorter or longer slices, balancing latency and throughput across diverse tenants.

As shown in Figure 9, the guest-reported VM utilization stays nearly flat (around 6–9%) across all slice lengths, but the host CPU utilization jumps from about 8–10% at 400 µs to over 50–60% at 0–20 µs. These extra host cycles are spent on more frequent timer interrupts, VM exits, and rescheduling rather than useful work, effectively increasing the cost of idleness and capping the workload's maximum sustainable throughput. Thus, blindly choosing the smallest slice is counterproductive: `mwait`-sched must trade off tail-latency gains against the additional host CPU consumption of overly

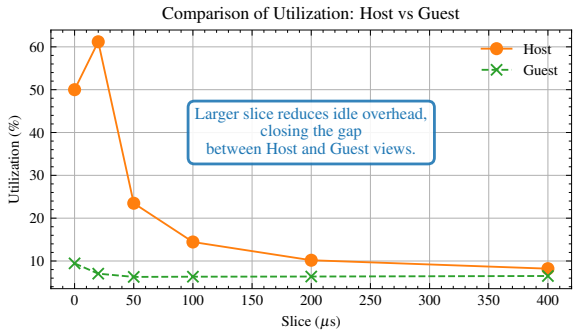


Figure 9: Average host/guest CPU utilization of the nine representative workloads under different `mwait`-sched slice lengths (2 vCPU : 1 pCPU). Short slices inflate host usage due to timer and scheduling overhead, while guest-reported utilization remains almost unchanged.

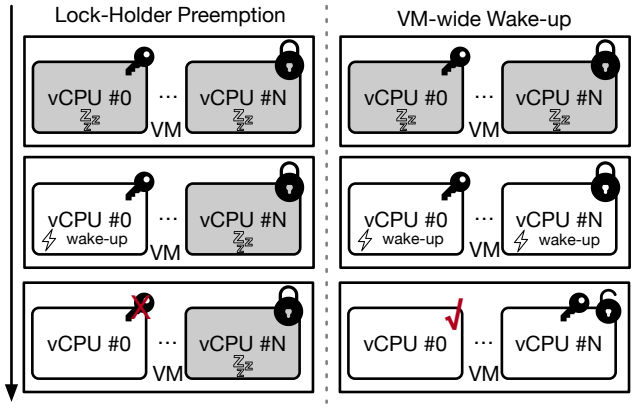


Figure 10: Naïve `mwait` emulation wakes only the waiter, leaving the lock holder descheduled and causing lock-holder preemption. `mwait`-sched instead performs VM-wide wake-ups, ensuring lock holders and waiters are co-scheduled to restore forward progress under oversubscription.

fine-grained slices.

Avoiding Lock-Holder Preemption in `mwait` Emulation. A naïve implementation of profile-guided `mwait` emulation wakes only the vCPU that issued the simulated `mwait` exit. However, this approach suffers from the classic *lock-holder preemption* problem: the awakened vCPU often depends on another vCPU—typically the lock holder or a synchronization partner—that has been descheduled by the hypervisor. As a result, the awakened vCPU immediately stalls on the same contended memory location, wasting its entire time slice and amplifying tail latency. In oversubscribed environments, where such dependency chains are common, this can lead to severe head-of-line blocking and cascading delays across the VM.

To address this issue, we treat `mwait`-driven wake-ups as VM-local synchronization events and wake *all* runnable vCPUs belonging to the same VM. When the periodic timer re-evaluates simulated `mwait` states and determines that a vCPU should be woken, we activate not only the vCPU that originally issued the simulated `mwait` but also other vCPUs in the same VM that may lie on its critical path. This optimization addresses the classic *lock-holder preemption* problem: by waking all runnable vCPUs that may participate in the upcoming synchronization step—including lock holders, I/O threads, and application threads likely to release or propagate the event—we ensure immediate forward progress and avoid pathological latency spikes under oversubscription.

The overhead of this design is low in practice: VMs typically have only a small number of runnable vCPUs near idle boundaries, and waking them merely marks the vCPUs as eligible for scheduling rather than forcing context switches. Figure 11 confirms this even in the worst case where all 48 vCPUs of a VM are woken at once. The hypervisor cost to mark all 48 runnable stays at $\sim$20 µs across oversubscription ratios $R \in \{1.0, 1.1, 1.2, 1.5\}$, since the work scales with the vCPU count, not the level of oversubscription. The time until every vCPU is actually on-CPU climbs to $\sim$240 µs at R=1.5, but that gap is just the queue waiting for a pCPU to free up; any wake mechanism inherits it under oversubscription. In contrast, the latency benefits are substantial. By aligning wake-up semantics with VM-level synchronization behavior, the scheduler ensures that lock holders and waiters are co-scheduled, reducing wake-up jitter and improving end-to-end performance under oversubscription.

Figure 10 illustrates the difference between naïve single-vCPU wake-up and our VM-wide wake-up strategy. In the naïve case, the waiter is awakened while the lock holder remains descheduled, causing the waiter to spin unproductively until the next scheduling opportunity. In contrast, our approach wakes both the waiter and the lock holder, restoring forward progress immediately.

4.2 `mwait`-based vCPU Aggregation

A central insight of `mwait`-sched is that `mwait` provides a fine-grained, guest-generated signal of short-term idleness that can be formalized and exploited for safe vCPU aggregation. To distinguish meaningful slack from brief synchronization stalls, `mwait`-sched maintains a lightweight profile of idle intervals. Let Δt_i denote the duration of the i-th idle episode, measured as the time between two consecutive `mwait` entries or between `mwait` entry and the next activity-indicating timer tick. Empirically, idle durations form a bimodal distribution: one cluster corresponding to microsecond-scale *transient idle* caused by spinlock backoff or retry loops, and another corresponding to millisecond-scale *stable idle* produced by genuine quiescence such as thread sleep or queue emptiness. Figure 12 confirms this on production hosts: in eBPF-sampled traces, 99.6% of `mwait` episodes on busy vCPUs finish within 200 µs, while

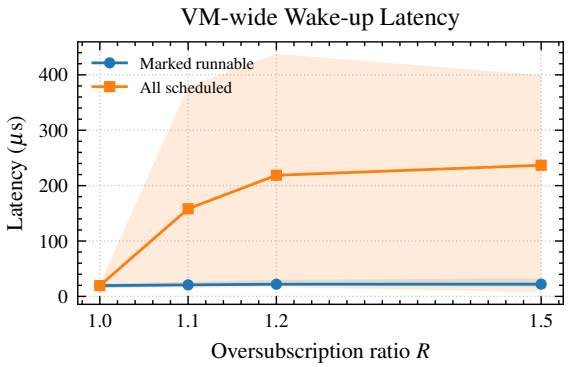


Figure 11: VM-wide wake-up latency on a 48-vCPU VM at oversubscription ratio R (vCPUs per pCPU). *Marked runnable* is the hypervisor's cost to mark all 48 vCPUs eligible to run; *All scheduled* is the time until every vCPU has been picked up by a pCPU. Shaded bands span per-trial min/max over a 30 s window; markers show the mean.

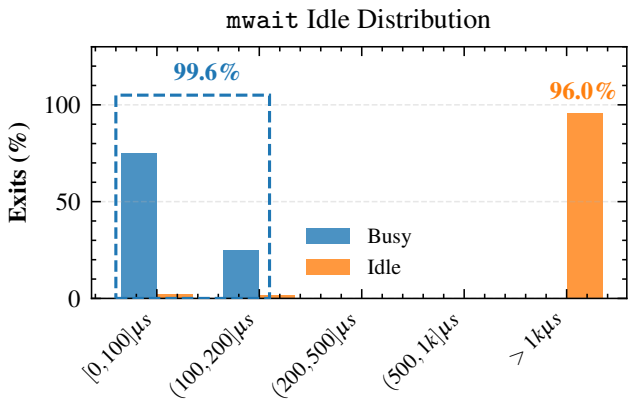


Figure 12: Distribution of `mwait` idle durations sampled with eBPF on randomly chosen production hosts. 99.6% of episodes on busy vCPUs end within 200 µs; 96.0% of episodes on idle vCPUs last beyond 1 ms. The two populations are essentially disjoint, justifying the binary stable/transient classifier.

96.0% of episodes on idle vCPUs last beyond 1 ms. Almost nothing lands between 200 µs and 1 ms, so a single threshold cleanly separates the two populations.

We formalize this distinction using a classifier $C(\Delta t_i)$:

$$C(\Delta t_i) = \begin{cases} \text{transient}, & \Delta t_i < \theta \text{ or } \Delta t_i \in \text{HighVar}(W), \\ \text{stable}, & \Delta t_i \geq \theta \text{ and } \Delta t_i \in \text{LowVar}(W), \end{cases}$$

where θ is a latency-safe aggregation threshold, and $\text{Var}(W)$ measures the variance of idle intervals over a sliding window $W = \{\Delta t_{i-k}, \ldots, \Delta t_i\}$. Transient idle thus corresponds to short, bursty intervals with high variance, indicating imminent forward progress and making aggregation unsafe. Stable idle corresponds to long, low-variance intervals, signaling that the vCPU is quiescent and eligible for aggregation.

Only vCPUs for which $C(\Delta t_i) = \text{stable}$ are considered for packing. When spare capacity is available, the scheduler aggregates these stable-idle vCPUs onto shared pCPUs, increasing consolidation density while avoiding interference with active workloads.

To preserve latency guarantees, aggregation is tightly coupled with deterministic de-aggregation. When a vCPU exhibits transient idleness, exiting `mwait` quickly, it is immediately separated from its siblings and restored to dedicated execution. This rapid de-aggregation prevents bursty workloads from incurring queuing delays while still allowing deep consolidation during idle phases.

Compared to traditional consolidation mechanisms that rely on coarse reactive signals such as CPU utilization or run-queue length, `mwait`-based aggregation is predictive: the guest explicitly informs the hypervisor of current idleness, enabling fine-grained packing opportunities without incurring unnecessary interference. This design improves consolidation density, supports higher oversubscription ratios, and reduces pCPU fragmentation while keeping tail latency close to the non-overcommitted baseline.

Architectural C-state reporting does not solve this visibility problem. Although x86 processors expose MWAIT-hint-driven C-state residency fields through power-management registers [19], these counters are too coarse and asynchronous for scheduling. They can show that a core spent time in an idle state, but they do not provide the timely per-vCPU yield signal needed to multiplex pCPUs safely.

4.3 Multi-Address `mwait`-Proxy

The profile-guided, timer-based approach works well when only two vCPUs share a pCPU, but it fails to scale as more vCPUs are colocated. Assigning a dedicated timer to each vCPU triggers excessive VM switches, consuming substantial pCPU cycles in context switching. Moreover, for workloads such as database systems that rely heavily on bare-metal `mwait` on the `NEED_RESCHED` flag to bypass IPIs [46], timer-based emulation becomes fundamentally inadequate: the guest must wait until the next timer tick, even when the `NEED_RESCHED` flag is already set.

To support high vCPU aggregation while avoiding these delays, we introduce a multi-address, linked-list–based `mwait`-proxy design that tracks multiple watched locations and promptly boosts these workloads without relying on periodic timers. Each time a vCPU executes `mwait` and exits, we insert its monitored address into a shared linked list maintained by the hypervisor. Whenever control returns to the hypervisor, we scan this list and check all monitored addresses to determine whether any vCPU should be woken up. Unlike the timer-based approach, this method allows multiple vCPUs to

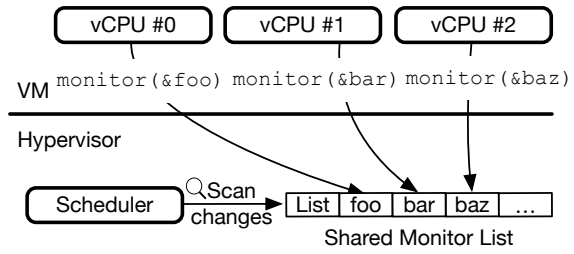


Figure 13: `mwait`-proxy replaces per-vCPU timer polling with a shared multi-address monitor. By tracking all `mwait` addresses in the hypervisor, it supports many colocated vCPUs without incurring excessive VM exits.

share a pCPU efficiently without installing multiple periodic timers, eliminating unnecessary VM switches and reducing wasted cycles.

We believe this multi-address proxy is the most practical way to emulate multi-address `mwait` using today's x86 hardware. Supporting true multi-address monitoring would likely require ISA support—such as a new vectorized monitor instruction—to allow the hardware to track multiple addresses natively.

Wake-up correctness. Native x86 `mwait` wakes on any store to the monitored address, not on a value change, so a same-value write or a write-then-revert can in principle defeat a value-comparison check. `mwait`-sched is unaffected: the timer just delays the vCPU between checks, and the wake check runs at every tick, so no event in the steady state goes unchecked. The question applies to `mwait`-proxy, which checks the monitored value (not the coherence event) when it scans the watch list. We accept this simplification because, in current Linux guests, only the idle path uses `mwait`, and its wake signal is the kernel's `need_resched` flag going from 0 to 1. Only the idle vCPU itself clears the flag, so a same-value write by another vCPU does not occur in practice; even if it did, native `mwait` would just wake the vCPU, re-check, and re-enter `mwait`—indistinguishable from our skipping the spurious wake. No kernel logic we surveyed depends on observing every store as a discrete pulse: a software flag that is set and then cleared between observation points is already an inherently missable event, so guests do not build protocols around it. Moreover, KVM's upstream `mwait`-nop has the same limitation: with `mwait` NOPed, the guest's idle loop polls the wake flag itself, and a same-value write or write-then-revert between two polls is equally missed. `mwait`-proxy therefore introduces no regression relative to current production behavior.

Security. Pure `mwait`-passthrough lets an idle vCPU monopolize a shared pCPU, which §2.2 flagged as a denial-of-service vector. Both `mwait`-sched and `mwait`-proxy close this gap: a malicious or buggy guest cannot withhold the pCPU beyond one timer slice (`mwait`-sched) or until the next hypervisor entry (`mwait`-proxy), so the worst-case interference one tenant can impose on its colocated neighbors is small and predictable. The hypervisor stays the trust boundary, and these paths use only state already available from VM exits, so they do not add any new attacker capability.

Table 2: Environment Configuration

Type	Hardware / Software Configuration
Host	Intel Xeon CPU Platinum 8269CY 52C/104T @ 2.50GHz (2 sockets) 12 × 16GB DRAM DDR4 2666 MT/s Linux 4.9.168 QEMU 8.2.2
Guest VM	4 vCPUs, 8GB RAM Linux 5.10.134 NIC: dual-queue virtio-net ×1 10Gb/s max throughput, 190,000 PPS
Production	37 regions, 92,000 nodes Same generation as host

Table 3: Representative Workloads Used in the `mwait`-sched Evaluation

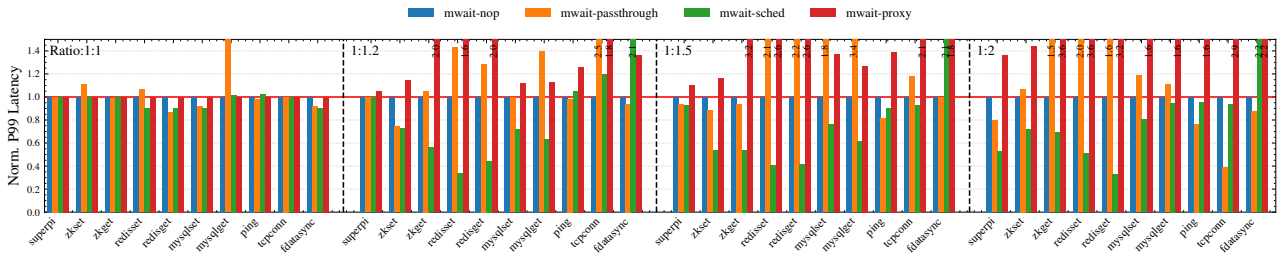
Workload	**Type**	**Description**
FS [2]	Disk I/O	File I/O with `fdatasync`
MS [3]	Disk I/O	MySQL: Random get/set
PT	Network	Ping Gateway
RD [35]	Network	Redis: Random get/set
SP [12]	Computation	Calculates π
TC	Network	TCP connection loop
ZK [1]	Network	ZooKeeper service

5 Evaluation

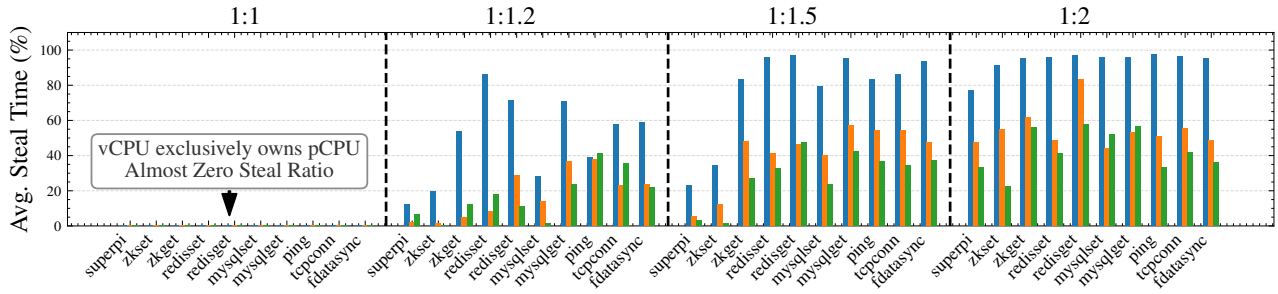
We evaluate `mwait`-sched through a comprehensive set of experiments. First, we examine end-to-end latency and preemption behavior across diverse workloads to characterize its scheduling impact. Second, we assess the effectiveness of `mwait`-proxy under high oversubscription ratios and quantify its ability to sustain performance at scale. Third, we analyze multi-region deployment data to demonstrate the improvements `mwait`-sched delivers in real production environments.

Experimental Setup. We evaluate `mwait`-sched through controlled experiments and production deployments. Our study compares four configurations:

- **`mwait`-nop (Baseline)**: Upstream KVM implements `mwait` as a no-op, causing the vCPU to remain runnable



(a) **Normalized P99 Latency (Lower is better).** Values are normalized to the `mwait-nop` baseline at the same ratio. `mwait`-sched effectively mitigates tail latency degradation under high oversubscription (1:1.5/1:2), particularly for latency-sensitive workloads like Redis and ZooKeeper. `mwait`-proxy is benign at 1:1 and scales smoothly through the low-density regime, with the synchronization-heavy workloads (RD, ZK Get, TC) climbing fastest—consistent with the higher-density picture in Figure 15.



(b) **vCPU Steal Time Analysis (Lower is better).** The breakdown of steal time across different oversubscription ratios. `mwait`-sched significantly suppresses steal time in oversubscribed scenarios (e.g., 1:2), explaining the latency improvements observed in (a). The 1:1 base case shows negligible contention.

Figure 14: **Performance and Interference Analysis.** Comparison of `mwait-nop`, `mwait-passthrough`, and the proposed `mwait`-sched and `mwait`-proxy mechanisms. Simple benchmarks such as Ping remain stable, while `mwait`-sched reduces preemption overhead for synchronization-heavy workloads.

and always accounted as 100% busy. This reflects the default behavior in today's virtualized clouds.

- **`mwait-passthrough`**: `mwait` executes directly without triggering VM exits, offering the lowest exit latency but preventing pCPU sharing because the vCPU never yields to the hypervisor.

- **`mwait-sched`**: Our timer-based emulation of `mwait`, which exposes idle signals to the hypervisor while avoiding the high overhead of trap-and-emulation. This enables predictable pCPU sharing under oversubscription.

- **`mwait-proxy`**: A linked-list–based multi-address monitor that extends `mwait`-sched to track many vCPUs concurrently, improving aggregation density under high oversubscription.

All configurations are shown in Table 2. Experiments constrain guest VMs to a single NUMA node to avoid cross-node interference. Our production evaluation spans multiple regions and heterogeneous workloads, enabling characterization across diverse usage patterns and deployment scales.

Workload. Due to privacy and isolation requirements, our microbenchmarks were conducted in a controlled testbed rather than on production VMs. The testbed evaluation spans 24 representative workload categories and 220 *scenarios*. A scenario pairs a workload category with a traffic profile, including request mix, rate, and key/payload distribution. These scenarios are derived from production traffic patterns, user reports, and performance incidents. Table 3 lists the subset we highlight in figures throughout this paper. In each experiment round, multiple VMs were randomly assigned scenarios and executed concurrently to capture contention, preemption behavior, and end-to-end latency under oversubscription.

5.1 Stand-Alone Classifier Evaluation

§4.1 uses the IOPS-to-utilization ratio from PMU counters to choose short or long timer slices. We test this classifier on PMU telemetry from 1,000 production VMs whose workload type is known from the VM image. The two classes are well separated: I/O-bound VMs have a median ratio of 840, while CPU-bound VMs have a median ratio of 20, a 42× gap. We set $T = 100$ between the two groups. This gives a 0.21% overall misclassification rate: 0.07% of I/O-bound VMs are

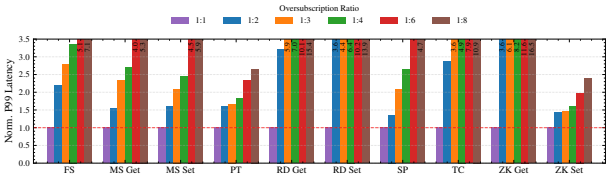


Figure 15: Normalized P99 latency of `mwait`-proxy across ten workloads (FS, MS Get/Set, PT, RD Get/Set, SP, TC, ZK Get/Set) at oversubscription ratios from 1:1 to 1:8. The y-axis is capped at 3.5×; bars over the cap show their value as a label. 1:4 is the burst tier's typical production density and 1:6 its peak; 1:8 is included as a stress point beyond production.

classified as CPU-bound, and 0.36% of CPU-bound VMs are classified as I/O-bound. The harmful error is the first one because it can hurt tail latency; the second mainly wastes a few timer ticks. The result is not sensitive to the exact threshold: any $T \in [50, 150]$ keeps misclassification below 0.7%.

5.2 End-to-End Tail Latency

Mitigating Contention. Figure 14a shows the impact of our mechanism on tail latency under different oversubscription ratios. When contention is low (1:1 mapping), all configurations behave similarly. As the oversubscription ratio increases, however, `mwait`-sched scales more effectively. At high oversubscription levels, it reduces the P99 latency of latency-sensitive workloads such as ZooKeeper and Redis by 30–50% compared to the `mwait`-nop baseline. This improvement aligns with the steal-time analysis in Figure 14b. Under `mwait`-nop, vCPUs suffer severe starvation—steal time exceeds 90% for synchronization-heavy workloads—indicating frequent lock-holder preemption. In contrast, `mwait`-sched leverages the semantic gap to convert unproductive spin-wait loops into cooperative yields, reducing steal time to below 60% and allowing critical sections to make forward progress even under heavy interference.

Two outliers need explanation. First, `mwait`-passthrough spikes for `mysqlset` at 1:1 because the pCPU enters deep C-states between requests, and waking from them adds latency to each MySQL write. Second, `fdatasync` stays high under `mwait`-sched at every ratio because the timer can add up to one slice of delay per fsync, and `fdatasync` issues many fsyncs back-to-back.

5.3 `mwait`-Proxy

Figure 15 shows normalized P99 latency for ten representative workloads at oversubscription ratios 1:1, 1:2, 1:3, 1:4, 1:6, and 1:8. The 1:2–1:4 range covers the densities used in production, with 1:4 the burst tier's typical density and 1:6 its peak; 1:8 is a stress point beyond production. At 1:2 most workloads stay within ∼2.2× (FS 2.2×, MS 1.6×, PT 1.6×, SP 1.4×), and the synchronization-heavy ones (TC, RD Get/Set, ZK Get) sit at 2.9–3.7×. At 1:4 the workloads split further: CPU- and I/O-bound ones scale moderately (1.8–3.4×), while synchronization-heavy ones reach TC 4.9×, RD Set 6.4×, RD Get 7.0×, and ZK Get 8.2×. ZK Set, dominated by write batching rather than per-request synchronization, stays close to baseline (1.4–1.6×). The 5–8× inflation on the synchronization-heavy group fits within the burst tier's variance budget, which is what makes `mwait`-proxy a viable production scheme at this density.

Beyond 1:4 the proxy's monitor-list traversal cost dominates: each entry into the hypervisor scans a longer list, and per-vCPU wake events compete for service. At 1:6 (burst peak) RD Get/Set and ZK Get reach 10–12× and TC reaches 7.9×, while CPU- and I/O-bound jobs sit at 2–5×. At 1:8 the worst cases land at 11–16× (TC 11×, RD Set 14×, RD Get 15×, ZK Get 16×), exposing the scaling limit of software multi-address emulation and motivating the hardware-level vectorized monitor support discussed in §6.

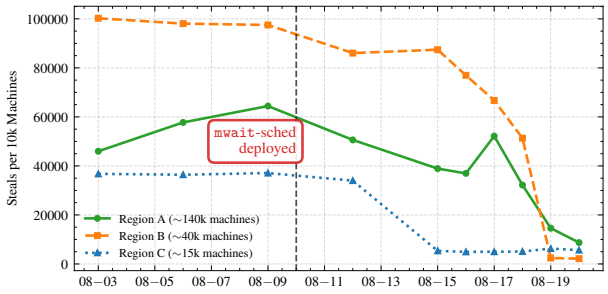
5.4 Production Rollout

Our production fleet has two instance classes with different SLO requirements, and we pick `mwait`-sched vs. `mwait`-proxy by class at deployment rather than auto-switching per VM. *Dedicated* instances have strict latency SLOs and stay at ≤1:2 oversubscription, since putting more than two vCPUs on a pCPU easily causes SLO violations at this tier; this fleet runs `mwait`-sched, which is what we report below. *Burst* instances tolerate higher latency variance and typically run at ∼1:4 (peaking at 1:6), where `mwait`-proxy handles the longer monitor list. The numbers in this section therefore come from the dedicated fleet under `mwait`-sched.

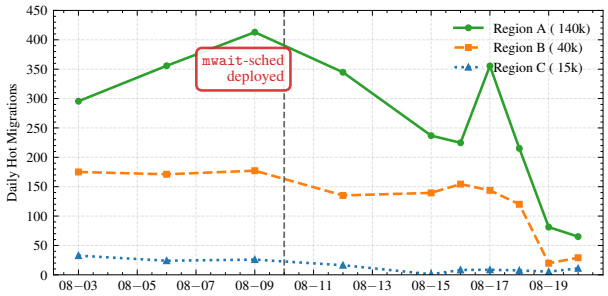
We selected three representative regions with distinct machine scales and baseline contention levels to evaluate the cross-region effects of `mwait`-sched. Before deployment, the frequency of high-contention steal events (steal ratio > 5%) differed significantly across regions: Region B was under the most pressure (around 95,000–100,000 steals per 10k machines); Region A was moderate (rising from 45,000 to 65,000); and Region C was flatter (around 36,000–37,000).

As shown in Figure 16a, all three regions experience a substantial drop after the rollout of `mwait`-sched. Within about ten days, Region B drops by ∼97% (from a ∼100,000 peak to under 5,000), Region A by ∼85% (from a ∼65,000 peak to under 10,000), and Region C by ∼86% (from ∼37,000 to ∼5,000). Despite their disparate starting points, the post-deployment curves converge to a significantly lower and tightly clustered band, indicating that `mwait`-sched consistently suppresses oversubscription-induced contention across heterogeneous environments.

Hot migration activity provides an independent validation of this effect. When enough VMs exceed the 5% steal time



(a) Normalized steal events.



(b) Daily live migrations.

Figure 16: Cross-region production signals before and after `mwait`-sched rollout. (a) Steal events drop sharply and converge to a consistently low band across heterogeneous regions. (b) Hot migrations also decline immediately after deployment, independently confirming that reduced contention improves VM stability and mitigates scheduler-driven relocations.

threshold, the platform initiates VM hot migrations to avoid SLO violations. As shown in Figure 16b, all three regions exhibit a clear decline in daily hot migrations immediately following deployment. Since hot migrations are triggered primarily by severe CPU contention and imbalance, their consistent decline corroborates the steal-ratio reductions and highlights the improved runtime stability brought by `mwait`-sched.

Figure 17 shows the global effects of deploying `mwait`-sched across multiple production regions covering 3.2M physical CPU cores. After rollout, the oversubscription ratio increased from 1.0% to 20.3%, effectively adding roughly 600,000 vCPUs of sellable capacity. Despite this substantial increase in utilization, system reliability improved: the normalized daily alarm rate dropped from 512 to 197 per 10k machines across regions. These results demonstrate that precise idle-state handling allows the platform to safely operate at much higher density while simultaneously reducing QoS-related incidents at hyperscale.

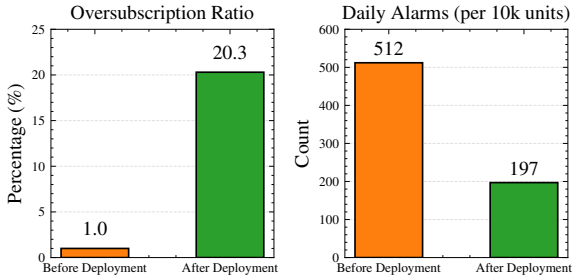


Figure 17: Global deployment impact of `mwait`-sched across multiple regions covering 3.2M physical CPU cores. The rollout increased oversubscription ratio (1.0%→20.3%) while reducing daily alarms (512→197 per 10k machines), demonstrating higher utilization without compromising stability.

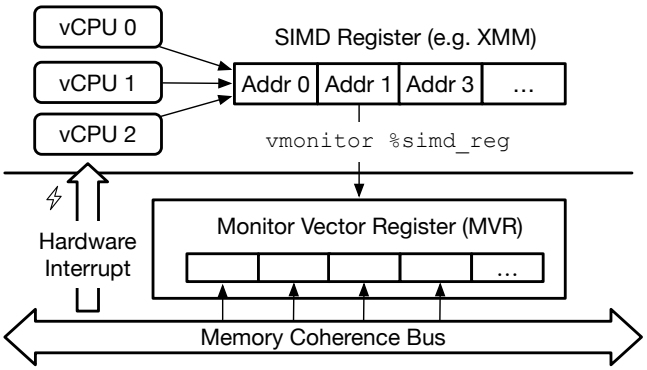


Figure 18: Conceptual design of a vectorized `vmonitor` instruction. Software supplies a SIMD-packed list of addresses, which hardware loads into a Monitor Vector Register (MVR) and checks in parallel against coherence-bus writes.

6 Recommendations for Future Hardware

Based on our experience deploying `mwait` at scale in virtualized clusters, we believe future ISAs (including x86, Arm, and RISC-V) should treat idle instructions not merely as power-management hints, but as first-class scheduling primitives. Below we outline several directions for hardware designers; these proposals are conceptual and intended to guide future architectural support rather than describe implemented mechanisms.

Multi-address monitoring. Current `monitor/mwait` instructions bind a thread to a single watched location, yet virtualized workloads often need to observe the quiescent state of many vCPUs simultaneously (e.g., runnable flags or per-thread wake tokens). We recommend exploring vectorized forms of the instruction that accept a small array of virtual addresses and a validity mask, allowing software to arm or update a set of watchpoints with a single operation.

Conceptually, such an extension could take the form of

a new vmonitor interface (Figure 18) in which software supplies a vector of addresses using SIMD registers. Hardware would load these addresses into a Monitor Vector Register (MVR) and compare coherence-bus writes against all entries in parallel, triggering an interrupt whenever any monitored location changes. This generalizes mwait from a single-address primitive into a hardware-level multi-way event monitor—akin to providing select/epoll semantics within the CPU. While we do not implement this design, it illustrates a promising direction for reducing polling overhead, eliminating unnecessary context switches, and enabling a host thread to efficiently wait on many vCPU idle signals. Such support would make multi-address mwait semantics practical for virtualization and allow denser vCPU aggregation in oversubscribed cloud deployments.

Virtualization-aware semantics. Future ISAs should also define how idle instructions interact with hypervisors explicitly: which state transitions are visible to the guest, which are owned by the host, and which events must be delivered as exits. A "virtualizable wait" variant, for example, could guarantee that entering the wait state always notifies the hypervisor, while wakeups may be signaled either through a watched-memory write or via a host-injected event. Clear semantics of this kind would enable safe pCPU sharing while still providing the low exit latencies expected by modern guests.

7 Related Work

7.1 Idle in Virtualization

Idle instructions are a fundamental but often overlooked component of virtualized execution. Prior measurements show that idleness can dominate context-switch overhead: Liu et al. [28] observe that idle periods account for up to 66.7% of context-switch costs in GaaS (GPU as a Service) clouds, and similar effects appear in nested virtualization where idle loops amplify VM–VMM interaction overheads [25]. Most existing work identifies VM idleness via VM introspection [10, 11, 41], inferring quiescence by sampling guest states or events, but such approaches are infeasible at hyperscale due to overhead, isolation concerns, and operational complexity. To our knowledge, mwait-sched is the first production-scale system that directly leverages hardware idle instructions as a scheduling signal, avoiding introspection while enabling sub-microsecond reactivity.

7.2 Oversubscription Optimization

A large body of work aims to improve vCPU colocation under oversubscription [5, 37, 44, 48, 53, 54]. Hardware-assisted techniques isolate interference through cache partitioning or memory-bandwidth control [29], and in-hardware request schedulers prioritize latency-critical VMs [7, 39, 43, 49]. Coordination-based designs further reduce double scheduling (e.g., UFO [33]) or exploit fine-grained time windows to identify transient opportunities for packing (e.g., SmartHarvest [48]). A separate line of work targets high CPU efficiency for latency-sensitive workloads through user-space scheduling and core reallocation, most notably Shenango [30]. These designs deliver impressive efficiency on dedicated workloads but require application-level integration with their runtime, which is incompatible with our public-cloud setting where guests run unmodified images. However, two limitations persist in public clouds: QoS signals are generally unavailable, and even nominally idle VMs can introduce interference because the hypervisor cannot estimate the stability of idle periods. mwait-sched addresses both challenges by exploiting the semantics of idle instructions: the guest-generated idle signal reveals short-term quiescence without application-level hints, allowing the hypervisor to dynamically adjust colocation decisions. By aggregating vCPUs with stable idle patterns and separating those exhibiting only transient waits, mwait-sched achieves a practical balance between reducing exit latency and improving CPU utilization.

8 Conclusion

mwait-passthrough provides low wake-up latency, but hides idle vCPUs from the host scheduler and prevents safe pCPU sharing. mwait-sched restores this visibility through timer-based emulation and idle-interval classification, while mwait-proxy supports denser colocation. This lets idle vCPUs yield their cores while preserving the low-latency behavior needed by guests. Across nine workloads, mwait-sched reduces P99 latency by 30–50% and steal ratio by 30–40%. Its deployment across $\sim$ 3.2M pCPUs reduced high-contention steal events by over 80% and raised the oversubscription ratio from 1.0% to 20.3%.

Acknowledgments

We thank our shepherd and the anonymous reviewers for their insightful comments that improved the quality of this paper. We also thank our colleagues for their feedback during the design and deployment. Their engineering support was essential to the production rollout and large-scale evaluation. This work was partially supported by the Ministry of Industry and Information Technology 2024 Cloud Operating System Project, the National Natural Science Foundation of China (No. 62572307), the Shanghai Key Laboratory of Scalable Computing and Systems, and the Alibaba AIR program.

References

[1] Apache ZooKeeper. https://zookeeper.apache.org, October 2024. [Online; accessed 11. Dec. 2025].

[2] fdatasync(2) — Arch manual pages. https://man.archlinux.org/man/fdatasync.2.en, August 2025. [Online; accessed 11. Dec. 2025].

[3] MySQL. https://www.mysql.com, August 2025. [Online; accessed 11. Dec. 2025].

[4] Omid Alipourfard, Hongqiang Harry Liu, Jianshu Chen, Shivaram Venkataraman, Minlan Yu, and Ming Zhang. Cherrypick: Adaptively unearthing the best cloud configurations for big data analytics. In Aditya Akella and Jon Howell, editors, *14th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2017, Boston, MA, USA, March 27-29, 2017*, pages 469–482. USENIX Association, 2017.

[5] Pradeep Ambati, Iñigo Goiri, Felipe Vieira Frujeri, Alper Gun, Ke Wang, Brian Dolan, Brian Corell, Sekhar Pasupuleti, Thomas Moscibroda, Sameh Elnikety, Marcus Fontoura, and Ricardo Bianchini. Providing slos for resource-harvesting vms in cloud platforms. In *14th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2020, Virtual Event, November 4-6, 2020*, pages 735–751. USENIX Association, 2020.

[6] Adam Belay, George Prekas, Ana Klimovic, Samuel Grossman, Christos Kozyrakis, and Edouard Bugnion. IX: A protected dataplane operating system for high throughput and low latency. In Jason Flinn and Hank Levy, editors, *11th USENIX Symposium on Operating Systems Design and Implementation, OSDI '14, Broomfield, CO, USA, October 6-8, 2014*, pages 49–65. USENIX Association, 2014.

[7] Jongwook Chung, Yuhwan Ro, Joonsung Kim, Jaehyung Ahn, Jangwoo Kim, John Kim, Jae W. Lee, and Jung Ho Ahn. Enforcing last-level cache partitioning through memory virtual channels. In *28th International Conference on Parallel Architectures and Compilation Techniques, PACT 2019, Seattle, WA, USA, September 23-26, 2019*, pages 97–109. IEEE, 2019.

[8] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with YCSB. In Joseph M. Hellerstein, Surajit Chaudhuri, and Mendel Rosenblum, editors, *Proceedings of the 1st ACM Symposium on Cloud Computing, SoCC 2010, Indianapolis, Indiana, USA, June 10-11, 2010*, pages 143–154. ACM, 2010.

[9] Eli Cortez, Anand Bonde, Alexandre Muzio, Mark Russinovich, Marcus Fontoura, and Ricardo Bianchini. Resource central: Understanding and predicting workloads for improved resource management in large cloud platforms. In *Proceedings of the 26th Symposium on Operating Systems Principles, Shanghai, China, October 28-31, 2017*, pages 153–167. ACM, 2017.

[10] Thomas Dangl, Benjamin Taubmann, and Hans P. Reiser. Rapidvmi: Fast and multi-core aware active virtual machine introspection. In Delphine Reinhardt and Tilo Müller, editors, *ARES 2021: The 16th International Conference on Availability, Reliability and Security, Vienna, Austria, August 17-20, 2021*, pages 19:1–19:10. ACM, 2021.

[11] Brendan Dolan-Gavitt, Tim Leek, Michael Zhivich, Jonathon T. Giffin, and Wenke Lee. Virtuoso: Narrowing the semantic gap in virtual machine introspection. In *32nd IEEE Symposium on Security and Privacy, SP 2011, 22-25 May 2011, Berkeley, California, USA*, pages 297–312. IEEE Computer Society, 2011.

[12] Fibonacci43. SuperPI. https://github.com/Fibonacci43/SuperPI, August 2025. [Online; accessed 11. Dec. 2025].

[13] Alexander Fuerst, Stanko Novakovic, Iñigo Goiri, Gohar Irfan Chaudhry, Prateek Sharma, Kapil Arya, Kevin Broas, Eugene Bak, Mehmet Iyigun, and Ricardo Bianchini. Memory-harvesting vms in cloud platforms. In Babak Falsafi, Michael Ferdman, Shan Lu, and Thomas F. Wenisch, editors, *ASPLOS '22: 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Lausanne, Switzerland, 28 February 2022 - 4 March 2022*, pages 583–594. ACM, 2022.

[14] Robert Grandl, Mosharaf Chowdhury, Aditya Akella, and Ganesh Ananthanarayanan. Altruistic scheduling in multi-resource clusters. In Kimberly Keeton and Timothy Roscoe, editors, *12th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2016, Savannah, GA, USA, November 2-4, 2016*, pages 65–80. USENIX Association, 2016.

[15] Jing Guo, Zihao Chang, Sa Wang, Haiyang Ding, Yihui Feng, Liang Mao, and Yungang Bao. Who limits the resource efficiency of my datacenter: an analysis of alibaba datacenter traces. In *Proceedings of the International Symposium on Quality of Service, IWQoS 2019, Phoenix, AZ, USA, June 24-25, 2019*, pages 39:1–39:10. ACM, 2019.

[16] Benjamin Hindman, Andy Konwinski, Matei Zaharia, Ali Ghodsi, Anthony D. Joseph, Randy H. Katz, Scott Shenker, and Ion Stoica. Mesos: A platform for fine-grained resource sharing in the data center. In David G. Andersen and Sylvia Ratnasamy, editors, *Proceedings*

of the 8th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2011, Boston, MA, USA, March 30 - April 1, 2011. USENIX Association, 2011.

[17] Jack Tigar Humphries, Neel Natu, Ashwin Chaugule, Ofir Weisse, Barret Rhoden, Josh Don, Luigi Rizzo, Oleg Rombakh, Paul Turner, and Christos Kozyrakis. ghost: Fast & flexible user-space delegation of linux scheduling. In Robbert van Renesse and Nickolai Zeldovich, editors, *SOSP '21: ACM SIGOPS 28th Symposium on Operating Systems Principles, Virtual Event / Koblenz, Germany, October 26-29, 2021*, pages 588–604. ACM, 2021.

[18] Intel Corporation. 5-level paging and 5-level ept white paper. White Paper 335252-002, Revision 1.1, Intel Corporation, May 2017. Revised and published online May 24, 2018.

[19] Intel Corporation. *Intel 64 and IA-32 Architectures Software Developer's Manual, Volume 3A: System Programming Guide*, 2024.

[20] Intel Corporation. Perfmon events. https://perfmon-events.intel.com/, 2025. Performance monitoring events reference for Intel processors.

[21] Sangeetha Abdu Jyothi, Carlo Curino, Ishai Menache, Shravan Matthur Narayanamurthy, Alexey Tumanov, Jonathan Yaniv, Ruslan Mavlyutov, Iñigo Goiri, Subru Krishnan, Janardhan Kulkarni, and Sriram Rao. Morpheus: Towards automated slos for enterprise clusters. In Kimberly Keeton and Timothy Roscoe, editors, *12th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2016, Savannah, GA, USA, November 2-4, 2016*, pages 117–134. USENIX Association, 2016.

[22] Alok Gautam Kumbhare, Reza Azimi, Ioannis Manousakis, Anand Bonde, Felipe Vieira Frujeri, Nithish Mahalingam, Pulkit A. Misra, Seyyed Ahmad Javadi, Bianca Schroeder, Marcus Fontoura, and Ricardo Bianchini. Prediction-based power oversubscription in cloud platforms. In Irina Calciu and Geoff Kuenning, editors, *Proceedings of the 2021 USENIX Annual Technical Conference, USENIX ATC 2021, July 14-16, 2021*, pages 473–487. USENIX Association, 2021.

[23] Michael Larabel. Amd updates linux patches for lowering idle exit latency. *Phoronix*, May 2022. "AMD engineer began posting Linux kernel patches so the kernel prefers the MWAIT instruction over HALT".

[24] Wanpeng Li. Towards a more scalable kvm hypervisor. In *KVM Forum 2018*, Edinburgh, Scotland, 2018. The Linux Foundation. Presentation slides and PDF available online.

[25] Jin Tack Lim and Jason Nieh. Optimizing nested virtualization performance using direct virtual hardware. In James R. Larus, Luis Ceze, and Karin Strauss, editors, *ASPLOS '20: Architectural Support for Programming Languages and Operating Systems, Lausanne, Switzerland, March 16-20, 2020*, pages 557–574. ACM, 2020.

[26] Linux Kernel Developers. KVM's Default "MWAIT as NOP" Emulation Behavior. https://github.com/torvalds/linux/blob/416f99c3b16f582a3fc6d64a1f77f39d94b76de5/arch/x86/kvm/x86.c.

[27] Linux Kernel Developers. Linux KVM x86 MONITOR/MWAIT Emulation using NOP. https://github.com/torvalds/linux/blob/416f99c3b16f582a3fc6d64a1f77f39d94b76de5/arch/x86/kvm/x86.c#L2244. Linux kernel source code, commit 416f99c3b16f582a3fc6d64a1f77f39d94b76de5, accessed December 6, 2025.

[28] Ming Liu, Tao Li, Neo Jia, Andy Currid, and Vladimir Troy. Understanding the virtualization "tax" of scale-out pass-through gpus in gaas clouds: An empirical study. In *21st IEEE International Symposium on High Performance Computer Architecture, HPCA 2015, Burlingame, CA, USA, February 7-11, 2015*, pages 259–270. IEEE Computer Society, 2015.

[29] David Lo, Liqun Cheng, Rama K. Govindaraju, Parthasarathy Ranganathan, and Christos Kozyrakis. Heracles: improving resource efficiency at scale. In Deborah T. Marr and David H. Albonesi, editors, *Proceedings of the 42nd Annual International Symposium on Computer Architecture, Portland, OR, USA, June 13-17, 2015*, pages 450–462. ACM, 2015.

[30] Amy Ousterhout, Joshua Fried, Jonathan Behrens, Adam Belay, and Hari Balakrishnan. Shenango: Achieving high CPU efficiency for latency-sensitive datacenter workloads. In Jay R. Lorch and Minlan Yu, editors, *16th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2019, Boston, MA, February 26-28, 2019*, pages 361–378. USENIX Association, 2019.

[31] Kay Ousterhout, Patrick Wendell, Matei Zaharia, and Ion Stoica. Sparrow: distributed, low latency scheduling. In Michael Kaminsky and Mike Dahlin, editors, *ACM SIGOPS 24th Symposium on Operating Systems Principles, SOSP '13, Farmington, PA, USA, November 3-6, 2013*, pages 69–84. ACM, 2013.

[32] Archit Patke, Dhemath Reddy, Saurabh Jha, Haoran Qiu, Christian Pinto, Chandra Narayanaswami, Zbigniew Kalbarczyk, and Ravishankar K. Iyer. Queue management for slo-oriented large language model serving. In *Proceedings of the 2024 ACM Symposium on Cloud Computing, SoCC 2024, Redmond, WA, USA, November 20-22, 2024*, pages 18–35. ACM, 2024.

[33] Yajuan Peng, Shuang Chen, Yi Zhao, and Zhibin Yu. UFO: the ultimate qos-aware core management for virtualized and oversubscribed public clouds. In Laurent Vanbever and Irene Zhang, editors, *21st USENIX Symposium on Networked Systems Design and Implementation, NSDI 2024, Santa Clara, CA, April 15-17, 2024*. USENIX Association, 2024.

[34] Red Hat, Inc. *Steal Time Accounting*. Red Hat, Inc., 2015. In *Red Hat Enterprise Linux 7 Virtualization Deployment and Administration Guide*, Chapter 8.3.

[35] Redis. Redis - The Real-time Data Platform. *Redis*, August 2025. [Online; accessed 11. Dec. 2025].

[36] Benjamin Reidys, Pantea Zardoshti, Íñigo Goiri, Celine Irvene, Daniel S. Berger, Haoran Ma, Kapil Arya, Eli Cortez, Taylor Stark, Eugene Bak, Mehmet Iyigun, Stanko Novakovic, Lisa Hsu, Karel Trueba, Abhisek Pan, Chetan Bansal, Saravan Rajmohan, Jian Huang, and Ricardo Bianchini. Coach: Exploiting temporal patterns for all-resource oversubscription in cloud platforms. In Lieven Eeckhout, Georgios Smaragdakis, Kaitai Liang, Adrian Sampson, Martha A. Kim, and Christopher J. Rossbach, editors, *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1, ASPLOS 2025, Rotterdam, The Netherlands, 30 March 2025 - 3 April 2025*, pages 164–181. ACM, 2025.

[37] Ghazal Sadeghian, Mohamed Elsakhawy, Mohanna Shahrad, Joe Hattori, and Mohammad Shahrad. Unfaasener: Latency and cost aware offloading of functions from serverless platforms. In Julia Lawall and Dan Williams, editors, *Proceedings of the 2023 USENIX Annual Technical Conference, USENIX ATC 2023, Boston, MA, USA, July 10-12, 2023*, pages 879–896. USENIX Association, 2023.

[38] Varun Sakalkar, Vasileios Kontorinis, David Landhuis, Shaohong Li, Darren De Ronde, Thomas Blooming, Anand Ramesh, James Kennedy, Christopher Malone, Jimmy Clidaras, and Parthasarathy Ranganathan. Data center power oversubscription with a medium voltage power plane and priority-aware capping. In James R. Larus, Luis Ceze, and Karin Strauss, editors, *ASPLOS '20: Architectural Support for Programming Languages and Operating Systems, Lausanne, Switzerland, March 16-20, 2020*, pages 497–511. ACM, 2020.

[39] Mohammad Shahrad, Sameh Elnikety, and Ricardo Bianchini. Provisioning differentiated last-level cache allocations to vms in public clouds. In Carlo Curino, Georgia Koutrika, and Ravi Netravali, editors, *SoCC '21: ACM Symposium on Cloud Computing, Seattle, WA, USA, November 1 - 4, 2021*, pages 319–334. ACM, 2021.

[40] Sudipta Saha Shubha and Haiying Shen. Adainf: Data drift adaptive scheduling for accurate and slo-guaranteed multiple-model inference serving at edge servers. In Henning Schulzrinne, Vishal Misra, Eddie Kohler, and David A. Maltz, editors, *Proceedings of the ACM SIGCOMM 2023 Conference, ACM SIGCOMM 2023, New York, NY, USA, 10-14 September 2023*, pages 473–485. ACM, 2023.

[41] Rayman Preet Singh, Tim Brecht, and Srinivasan Keshav. Towards VM consolidation using a hierarchy of idle states. In Ada Gavrilovska, Angela Demke Brown, and Bjarne Steensgaard, editors, *Proceedings of the 11th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments, Istanbul, Turkey, March 14-15, 2015*, pages 107–119. ACM, 2015.

[42] Gabriel L. Somlo. Handling of guest-mode monitor and mwait. https://www.contrib.andrew.cmu.edu/~somlo/OSXKVM/mwait.html, 2014. Last updated: Feb. 05, 2014.

[43] Jovan Stojkovic, Chunao Liu, Muhammad Shahbaz, and Josep Torrellas. Hardharvest: Hardware-supported core harvesting for microservices. In *Proceedings of the 52nd Annual International Symposium on Computer Architecture, ISCA 2025, Tokyo, Japan, June 21-25, 2025*, pages 708–722. ACM, 2025.

[44] Amoghavarsha Suresh and Anshul Gandhi. Servermore: Opportunistic execution of serverless functions in the cloud. In Carlo Curino, Georgia Koutrika, and Ravi Netravali, editors, *SoCC '21: ACM Symposium on Cloud Computing, Seattle, WA, USA, November 1 - 4, 2021*, pages 570–584. ACM, 2021.

[45] Chunqiang Tang, Kenny Yu, Kaushik Veeraraghavan, Jonathan Kaldor, Scott Michelson, Thawan Kooburat, Aravind Anbudurai, Matthew Clark, Kabir Gogia, Long Cheng, Ben Christensen, Alex Gartrell, Maxim Khutornenko, Sachin Kulkarni, Marcin Pawlowski, Tuomas Pelkonen, Andre Rodrigues, Rounak Tibrewal, Vaishnavi Venkatesan, and Peter Zhang. Twine: A unified cluster management system for shared infrastructure. In *14th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2020, Virtual Event, November 4-6, 2020*, pages 787–803. USENIX Association, 2020.

[46] Marcelo Tosatti. [PATCH] sched: introduce configurable delay before entering idle. *Linux Kernel Mailing List (LKML)*, May 2019. Message posted on May 7, 2019. URL: https://lkml.org/lkml/2019/5/7/1080.

[47] Michael S. Tsirkin. Kvm: introduce flag to allow guest control of host cpu power state. QEMU-Devel Mailing List, June 2018.

[48] Yawen Wang, Kapil Arya, Marios Kogias, Manohar Vanga, Aditya Bhandari, Neeraja J. Yadwadkar, Siddhartha Sen, Sameh Elnikety, Christos Kozyrakis, and Ricardo Bianchini. Smartharvest: harvesting idle cpus safely and efficiently in the cloud. In Antonio Barbalace, Pramod Bhatotia, Lorenzo Alvisi, and Cristian Cadar, editors, *EuroSys '21: Sixteenth European Conference on Computer Systems, Online Event, United Kingdom, April 26-28, 2021*, pages 1–16. ACM, 2021.

[49] Jifei Yi, Benchao Dong, Mingkai Dong, Ruizhe Tong, and Haibo Chen. Mtˆ2: Memory bandwidth regulation on hybrid NVM/DRAM platforms. In Dean Hildebrand and Donald E. Porter, editors, *20th USENIX Conference on File and Storage Technologies, FAST 2022, Santa Clara, CA, USA, February 22-24, 2022*, pages 199–216. USENIX Association, 2022.

[50] Xin Zhan, Reza Azimi, Svilen Kanev, David M. Brooks, and Sherief Reda. CARB: A c-state power management arbiter for latency-critical workloads. *IEEE Comput. Archit. Lett.*, 16(1):6–9, 2017.

[51] Chengliang Zhang, Minchen Yu, Wei Wang, and Feng Yan. Mark: Exploiting cloud services for cost-effective, slo-aware machine learning inference serving. In Dahlia Malkhi and Dan Tsafrir, editors, *Proceedings of the 2019 USENIX Annual Technical Conference, USENIX ATC 2019, Renton, WA, USA, July 10-12, 2019*, pages 1049–1062. USENIX Association, 2019.

[52] Ruiyi Zhang, Taehyun Kim, Daniel Weber, and Michael Schwarz. (M)WAIT for it: Bridging the gap between microarchitectural and architectural side channels. In Joseph A. Calandrino and Carmela Troncoso, editors, *32nd USENIX Security Symposium, USENIX Security 2023, Anaheim, CA, USA, August 9-11, 2023*, pages 7267–7284. USENIX Association, 2023.

[53] Yanqi Zhang, Iñigo Goiri, Gohar Irfan Chaudhry, Rodrigo Fonseca, Sameh Elnikety, Christina Delimitrou, and Ricardo Bianchini. Faster and cheaper serverless computing on harvested resources. In Robbert van Renesse and Nickolai Zeldovich, editors, *SOSP '21: ACM SIGOPS 28th Symposium on Operating Systems Principles, Virtual Event / Koblenz, Germany, October 26-29, 2021*, pages 724–739. ACM, 2021.

[54] Yunqi Zhang, George Prekas, Giovanni Matteo Fumarola, Marcus Fontoura, Iñigo Goiri, and Ricardo Bianchini. History-based harvesting of spare cycles and storage in large-scale datacenters. In Kimberly Keeton and Timothy Roscoe, editors, *12th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2016, Savannah, GA, USA, November 2-4, 2016*, pages 755–770. USENIX Association, 2016.

[55] Hang Zhu, Kostis Kaffes, Zixu Chen, Zhenming Liu, Christos Kozyrakis, Ion Stoica, and Xin Jin. Racksched: A microsecond-scale scheduler for rack-scale computers. In *14th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2020, Virtual Event, November 4-6, 2020*, pages 1225–1240. USENIX Association, 2020.

ARTIFACT EVALUATED
USENIX
AVAILABLE

Xkernel: Principled Performance Tunability of Operating System Kernels

Zhongjie Chen[†§], Wentao Zhang[‡], Yulong Tang[§],
Ran Shu[§], Fengyuan Ren[†], Tianyin Xu[‡], Jing Liu[§]

[§]*Microsoft Research*, [‡]*University of Illinois Urbana-Champaign*, [†]*Tsinghua University*

Abstract

The Linux kernel is permeated with constant values that are critical to system performance. Many of these constants, referred to as *perf-consts*, are magic numbers with brittle assumptions on hardware and workloads. Unfortunately, there is no capability of *in-situ* tuning of perf-const values on deployed kernels. This paper rethinks OS performance tunability. We present Xkernel, a system that offers a safe, efficient, and programmable interface for *in-situ* tuning of any perf-consts directly on a running kernel. Xkernel transforms any perf-const into a tunable knob on demand using a novel approach called Scoped Indirect Execution (SIE). SIE captures precise binary boundaries where a perf-const enters system state and redirects control to synthesized instructions that update the state as if new values were used. Xkernel goes beyond version atomicity when updating perf-consts to guarantee side-effect safety, a property notably absent in existing kernel update mechanisms. Case studies on various OS subsystems demonstrate significant performance benefits of tuning perf-consts which is made possible by Xkernel.

1 Introduction

Modern operating systems like Linux are permeated with constant values that shape system performance. These constants, referred to as *performance-critical constants* or *perf-consts*, appear in various source-code forms (e.g., macros, literals, and static integers); they govern thresholds, time intervals, batch sizes, scaling factors, etc. Whether to balance latency and throughput or to match batching behavior to device parallelism, perf-consts embed design trade-offs and workload/hardware semantics directly into kernel behavior, forming the kernel's implicit performance policy.

Perf-consts are not tunable in deployed systems without recompiling and rebooting the OS kernels. Unfortunately, their values are often "*arbitrarily chosen* [40]" by developers based on brittle heuristics, limited testing, or assumptions on dated hardware, which "*just happen(ed) to work well* [38]." However, static magic numbers can hardly serve dynamic workloads or diverse hardware configurations, especially emerging ones that significantly deviate from the time those values were chosen (see [14,16,20,39,56,75,76]).

Ideally, perf-consts should be decided at runtime, dynamically adapting to workload patterns, hardware characteristics, and service-level objectives. In practice, the benefits of tuning perf-consts are substantial. In one of our case studies (§2.1), tuning a perf-const yields 50× throughput improvement. Yet such benefits are completely missed as modern OSes provide no mechanism for tuning perf-consts.

Today, tuning a perf-const typically involves converting it into a runtime knob via interfaces such as `sysctl` [77] and `sysfs` [54], or changing its value in kernel source and updating the deployed kernel through live patching [1,7,10,22,23]. However, neither satisfies the needs of performance tuning. The former (`sysctl`/`sysfs`) is limited to a small subset of predefined constants with fixed granularity, often system-wide, and provides no safety guarantees—the correctness of an update depends on manual reasoning. The latter needs to recompile kernel code and apply binary diffs, incurring minutes-level delays, which is fundamentally incompatible with fast policy adaptation and online tuning.

In this paper, we advocate for *principled OS performance tunability*—a general machinery that enables safe, fast value updates for *in-situ* performance tuning of *any* perf-consts in deployed OS kernels, *without* kernel recompilation or rebooting. Such tunability must support expressive, programmable policies, ensure correctness and safety, as well as allow millisecond-scale value updates.

Achieving this goal is challenging. First, the system must precisely identify all instructions that consume the original constant, in the presence of sophisticated compiler transformations such as constant folding and strength reduction. Any missed or incorrectly replaced instructions may leave unsafe remnants in the running kernel. Second, the system must generate the instructions based on the new value, as well as the tuning policies, without recompilation. Finally, the kernel execution may have already produced side effects on runtime state, and any update must not cause conflicts on them.

We introduce *Scoped Indirect Execution (SIE)*, a novel mechanism that addresses the aforementioned challenges. Our key insight is that a constant—unlike arbitrary code—has structural semantics: its influence enters the machine state (registers or memory) at a specific point and manifests through a small instruction sequence, enabling safe, in-situ tuning of the perf-const without recompilation or rebooting. For a perf-const, the point where its value enters registers or memory, is *well-scoped*: it can be identified by static analysis and be represented by a symbolic state expression agnostic to compiler optimizations; moreover, it is small enough (typically several instructions) to analyze for side effects.

SIE leverages this structure to identify the precise binary region where the perf-const affects runtime machine state termed a *critical span*, by deriving the symbolic relationship between the perf-const and the affected registers or memory. Within this span, SIE inserts a set of *indirections*: small code snippets tied to specific kernel address locations. When execution reaches these locations, the indirections update the machine state to reflect the new value as per policy. Once execution leaves the critical span, the resulting state matches what would have occurred had the perf-const been changed directly in the binary. The kernel binary remains untouched; all modification occurs indirectly and locally.

To ensure safety, SIE analyzes how effects from the critical span propagate to derive a second region, the *safety span*, which encapsulates all consumption of constant-dependent state. At runtime, indirections are enabled only when the execution is outside the safety span, ensuring that updates never occur while the original effects are still in flight.

We build Xkernel atop Linux as the first system to realize principled performance tunability for OS kernels. Xkernel implements SIE, reducing update latency from minutes to milliseconds and supporting programmable tuning policies written in eBPF. Xkernel does not change the kernel source or require reboots. It seamlessly integrates with deployed kernels as its implementation depends on stable kernel features like Kprobe, kernel modules, and eBPF.

We evaluate Xkernel across core Linux subsystems, including CPU scheduling, memory management, storage, and network. Xkernel unlocks previously inaccessible tuning opportunities, achieving up to 50× microbenchmark improvement and boosting real-world application performance, e.g., 1.2× throughput in RocksDB and 81% latency reduction in NGINX. Xkernel enables new capabilities: online exploration of design trade-offs, adaptation to hardware and workload patterns, control of OS-internal maintenance behavior, and coordinated tuning across multiple perf-consts.

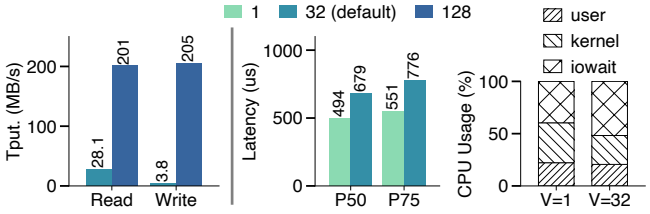
We extensively evaluate Xkernel on 140 perf-consts (comparable in scale to the 145 performance knobs exposed by `sysctl`). We show that SIE applies broadly, introduces negligible runtime overhead (a few hundred cycles per perf-const update), and achieves millisecond-scale policy updates.

In summary, this paper makes the following contributions:

- Principled OS performance tunability that exposes the unexplored performance benefits of pervasive perf-consts;
- Scoped Indirect Execution (SIE), a new approach that enables safe, fast performance tuning of any perf-consts with programmable policies;
- Xkernel, a practical implementation on Linux, which can directly benefit deployed kernels;
- Case studies and extensive evaluation demonstrating new opportunities and inspiring new tuning techniques;
- Xkernel is open-sourced at `https://github.com/xkernel-org/Xkernel/`.

Table 1: Performance regimes of perf-consts.

Category	Explanation and Examples
Threshold	Triggering a behavior change as a limit or boundary `#define MAX_SOFTIRQ_RESTART 10`
Interval	Controlling deferred or periodic actions `#define IPVS_SYNC_SEND_DELAY (HZ/50)`
Batch Size	Work processed together per operation to amortize cost `#define BLK_MAX_REQUEST_COUNT 32`
Scaling Factor	A multiplier that adjusts the magnitude or intensity `delta *= 4;`



(a) FIO on HDD (b) RocksDB on NVMe SSD

Figure 1: Performance benefits of changing the value of a perf-const `BLK_MAX_REQUEST_COUNT` based on hardware devices and workloads. The default value is 32.

2 Background and Motivation

2.1 Perf-Consts in Linux

A perf-const is a fixed numeric value used by kernel code to control the *magnitude* of OS behavior, without altering correctness or existence of that behavior. Table 1 shows four common types. Perf-consts are pervasive in Linux, appearing in every kernel subsystem. Typical perf-const representations include macros, literal immediates, and `static const` variables. Perf-consts govern core trade-offs (e.g., latency vs. throughput, responsiveness vs. utilization); they also shape how workloads interact with hardware. There are strong needs to customize their values based on application SLOs, workload behavior, and device characteristics.

A motivating example. We take `BLK_MAX_REQUEST_COUNT`, a perf-const in Linux's storage subsystem, as an example. Introduced in 2011 [40], this perf-const controls block I/O's plug behavior [4] by delaying request submission to merge adjacent requests and reduce device contention. Once buffered requests reach the threshold specified by this perf-const, they are flushed to the device. The original commit shows the value was arbitrary: "16 *works efficiently to reduce lock contention...* 32 *also works in my tests.*" The choice implicitly reflects then-current hardware—CPU speed, storage performance, and lock overhead. A decade later, in 2021, the value was raised to 32 [6], justified by observed benefits on NVMe devices. However, with the wide variety of hardware (HDD, SSD, and NVMe devices) and workload access patterns, a single magic number is inevitably suboptimal.

Figure 1(a) shows the performance benefits of tuning the perf-const. Running a sequential FIO workload [5] with intra-segment shuffling (4 KB requests randomly shuffled

within each 128 KB segment) on a 7200-RPM SAS HDD, the default value (32) causes the plug to flush too frequently, missing many opportunities to merge adjacent requests. Increasing the value to 128 allows most requests to be merged, maximizing sequential disk access. This improves read and write performance by 7× and 54×, respectively.

By contrast, a large value of `BLK_MAX_REQUEST_COUNT` may not always benefit NVMe SSDs—sometimes request merging has little benefit but adds overhead. We deploy RocksDB on a 256GB Toshiba XG3 NVMe SSD, and run the *multiread-random* workload from DBbench [21] with 32GB dataset (16B keys; 2048B values). We use RocksDB's `io_uring`-backed MultiGet API for asynchronous parallelism and use Direct I/O for data transfer between the storage device and user memory. As shown in Figure 1(b), reducing `BLK_MAX_REQUEST_COUNT` from 32 to 1 reduces CPU time spent on I/O wait by 12%, yielding an end-to-end 1.2× throughput improvement, while reducing P50 and P75 latency by 1.37× and 1.41×, respectively.

2.2 Limitations of Existing Mechanisms

Despite the strong benefits of tuning perf-consts, modern OSes like Linux provide little support or interface—perf-consts are hardwired into kernel binary once compiled and cannot be changed at runtime once deployed in production.

Existing mechanisms are inflexible and unsafe. One way to tune a perf-const is to convert a perf-const into a runtime variable and expose it through kernel interfaces like `sysctl` [77], `sysfs` [54] or system calls. However, this is not a general mechanism—it requires modifying source code on a per perf-const basis and results in rigid, narrow interfaces.

One fundamental difficulty is to predefine a complete set of perf-consts *a priori* before deployment; only a very small subset of perf-consts are currently exposed. Linux interfaces like `sysctl` and `sysfs` are treated as kernel ABIs and therefore prioritize stability over flexibility [44]. For example, our analysis shows that `sysctl` knobs change slowly—among 145 `sysctl` knobs, 96 of them have remained unchanged since 2005. Moreover, decisions to expose perf-consts as `sysctl` knobs have been largely *ad hoc*, driven by developers' preference and experience (Appendix A).

Besides the difficulty of upstreaming, source-code conversion is fundamentally inflexible: changing the granularity of a tuning policy from global (the `sysctl` default) to per-process or per-cgroup requires modifying source code again, which in turn requires recompilation and rebooting.

In addition, `sysctl` and `sysfs` conversions are known to risk safety [12, 72]. In our study, 20 of 145 conversions led to bugs, largely due to concurrency: for example, the `sysctl` setter writes to a shared global variable from a separate context while core kernel logic reads it concurrently, easily causing races in practice. Our inspection shows that 43 `sysctl` knobs are potentially buggy and lead to races or inconsistent states (see Appendix A).

Kernel live patching offers no rescue. A relevant mechanism is Kernel Live Patching (KLP) [1, 50, 60, 67, 74] which enables patching deployed kernels without rebooting. KLP modifies kernel source, recompiles it, and applies the resulting binary diff to replace selected regions of the kernel image. However, KLP is not suited for perf-consts tuning.

First, KLP is too slow. It takes minutes per patch. Updating a perf-const value would require recompiling the kernel (with the new value) and patching the binary diff. By the time a patch is applied, the workload may have changed.

Second, KLP uses functions as the patching unit. When tuning spans multiple functions (e.g., due to compiler inlining or multiple affected perf-consts), finding safe quiescent points becomes difficult and fragile [13]. KLP offers no side-effect safety and struggles to support consistent states under multi-threading [60, 67].

2.3 Our Goal: Principled OS Tunability

Our goal is to address the limitations of existing mechanisms and enable safe and fast tuning of any perf-consts in a running OS kernel with *principled OS tunability*:

Transparent in-situ tuning on deployed systems. Tuning a perf-const should not require the slow process of recompilation, redeployment, or rebooting the OS kernel.

Programmability and flexible granularity. Effective tuning requires programmable policies that can encode sophisticated algorithms and heuristics through fine-grained control (e.g., TCP flow awareness [80]). Policies should apply at flexible granularities (e.g., selected cgroups, selected devices, or their combinations) instead of fixed ones.

Out-of-box tuning for all perf-consts. A general tuning mechanism must support *any* perf-consts in the kernel and be fully compatible with standard OS kernel distributions.

Millisecond-scale policy updates with low overhead. Performance tuning must be fast and not cause interference on the target OS. We aim at *millisecond*-scale policy updates.

System safety during tuning. Tuning must be safe at runtime and must not result in inconsistent system states. Arbitrarily changing a perf-const value at runtime can be unsafe.

2.4 Challenges

Principled tunability introduces new challenges, stemming from the requirement to avoid recompilation and rebooting.

Precisely locating instructions for the perf-const. The compiler materializes each perf-const into instructions in the kernel binary. However, compiler optimizations can fold constants, merge lines, or reorder nearby instructions. A missed instruction leaves part of the original value in the system state; a misidentified one may overwrite unrelated computation and corrupt the system state.

Generating instructions for new values and policies. When users express new values and policies in a high-level language, we must generate the required instructions directly. KLP systems achieve this by recompiling the ker-

nel, which introduces minutes of delay. This is challenging because the generated code must interact correctly with existing kernel instructions and avoid conflicts such as register clobbering near the original sites.

Side-effects of the original value. The original value may have already propagated, producing side effects that kernel execution still depends on. A reboot clears these effects but is incompatible with our goal. We therefore must identify and detect when these side effects have dissipated so the new perf-const value can be applied safely at runtime.

3 Xkernel

Xkernel is the first system to provide a safe, fast, and expressive interface for *in-situ* tuning of any perf-const on a deployed Linux kernel. We present the design of Xkernel, beginning with *Scoped Indirect Execution* (SIE), a novel mechanism for principled tunability that addresses the challenges outlined above (§2.4). We then describe the design principles and the key techniques that realize SIE and provide programmability in Xkernel.

3.1 Scoped Indirect Execution (SIE)

The insight is that perf-consts exhibit intrinsic instruction-level semantics. For a perf-const, the point where its value enters the machine state (registers/memory) forms a *well-scoped region*. This region—a *critical span*—has three properties: (1) it can be localized by static analysis, (2) it corresponds to a *symbolic state expression* that captures the relation between the machine state and the perf-const, (3) the region is *small*, involving only a few instructions, which keeps correctness reasoning narrow and makes safe runtime updates tractable.

SIE identifies the critical span based on the symbolic state expression and safely redirects execution, at runtime, to a short JIT-compiled instruction sequence. These instructions execute next to the original code and, when combined, produce the same effects as if the perf-const were changed directly. The original binary remains intact; all modifications occur indirectly, within scoped regions.

SIE ensures side-effect safety by defining a second scope, *safe span*, within which the effects of the perf-const are encapsulated. Transitioning occurs only when execution is outside this span, converting a temporal coordination problem into a spatial one that can be analyzed statically.

SIE in Xkernel. Xkernel implements SIE on Linux. Figure 2 gives an overview of Xkernel and its user-facing workflow. Xkernel performs *offline* static analysis on kernel code to understand the symbolic state expression, critical span (CS), and safe span (SS) of each perf-const, and maintains them in a global *scope table*. The offline analysis is a one-time effort. The scope table only needs to be updated when the deployed kernel is updated.

To tune a perf-const, users (or agents) specify the perf-const in source code (by source file, line number, and token

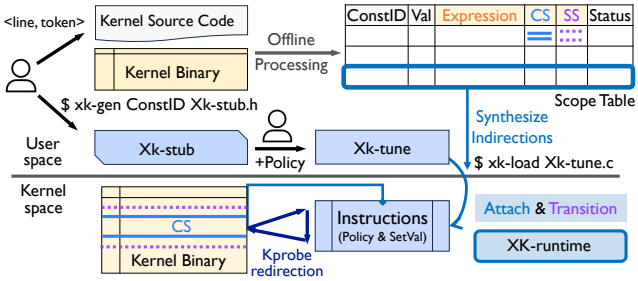


Figure 2: Overview of Xkernel.

index) and implement tuning policies in eBPF, called *Xk-tune*. An Xk-tune can be loaded at any time in the OS. A lightweight Xk-runtime handles attachment and transitions.

3.2 Design Principles

Separating value updates from tuning policies. Xkernel separates mechanism from policy: SIE provides a general mechanism for safe in-situ value updates, while programmability is delegated to safe kernel extensions (e.g., eBPF).

Synthesizing state update code, not patch instructions. Directly creating replacement instructions without recompilation is untenable. Instead, Xkernel synthesizes code that updates the system state described by symbolic state expressions of the perf-const, turning instruction replacement into a general state-update function.

Reusable static processing for fast policy update. Xkernel incurs a one-time offline cost per perf-const for kernel build, symbolic execution, and analysis. The resulting artifact (scope table) enables millisecond-scale *in-situ* updates and is reusable across all future values and policies. In contrast, live patching or source-code modification requires kernel rebuilding and rebooting for every change.

Decoupling version atomicity and side-effect safety. Xkernel ensures version atomicity with correctness guaranteed by symbolic state expressions, when transitioning from the original value to a new one. Xkernel also offers side-effect safety based on safe spans that encapsulate all effects of the original value. The safety of transition is supported for both per-thread and multi-threading.

Encapsulated kernel writes; free reads. To allow user-written policy programs (Xk-tunes) to update perf-const values, Xkernel provides a simple, safe API. Writes to kernel state are protected and only done by Xk-runtime.

3.3 Capturing Instruction-level Effect of Perf-Consts

Conceptually, Xkernel must replace the effect of instructions tied to the original perf-const value with the updated effect based on the new value, while preserving other runtime state. A key challenge is to precisely capture the effect, as reversing kernel binaries to original source code is difficult and costly. Our insight is that the state affected by perf-consts is tractable to capture. A perf-const manifests as a numeric value chosen by the compiler and becomes part of runtime state by instructions that produce a local numeric effect. De-

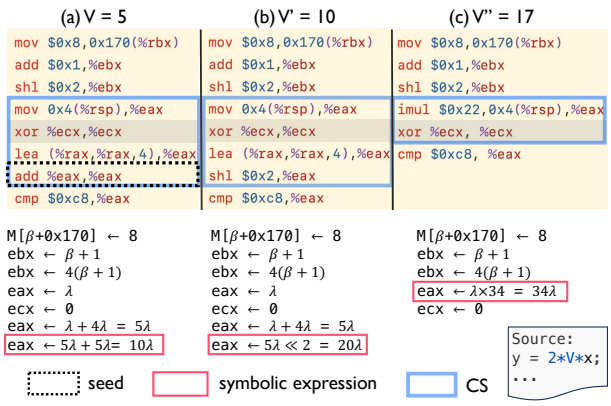


Figure 3: Symbolic expression derived from a perf-const (V)'s seed instructions and its critical span. (a) and (b) show instructions from binaries rebuilt with $V' = 10$ and $V'' = 17$. The seed is obtained by diffing (a) and (b). The IV values in the symbolic expression are 10, 20, and 34.

spite compiler optimizations that obscure its representation, we find that *symbolic state expression* provides a clean, precise way to express how a perf-const affects runtime state.

3.3.1 How Does a Perf-Const Affect Runtime State?

If a value is a constant and ultimately affects system state, there must be a well-defined point where it enters machine state. This boundary reflects the compiler's decision to materialize the constant and is determined statically. Since a constant carries no state before its introduction (unlike a variable), this entry point is identifiable in principle and is the semantic boundary we must capture.

A key challenge is to handle complications due to compiler optimizations. A perf-const may be folded, propagated, or merged with other expressions. Backend optimizations may further rewrite it for strength reduction or interleave unrelated instructions. As shown in Figure 3(a), the instruction sequence for a source-level perf-const of value 5 contains no immediate value 5 in the binary. Recompiling the same source with values 10 and 17 produces distinct instruction sequences (e.g., transforming `add` into `shl`, and reducing three instructions to one). Thus, the perf-const often appears in transformed form, denoted as IV, not its source value V.

Despite these transformations, IV can only influence machine state by becoming part of register or memory states, denoted as R/M. All effects must arise through instructions expressible in arithmetic and bitwise operations. Even source-level logical expressions reduce to comparisons such as "`cmp eax, edx`", which update condition flags as the arithmetic result of "`eax - edx`". Thus, regardless of source syntax, the effect of a perf-const in the binary always reduces to a mathematical relation in the form: $R/M \leftarrow f(R/M, IV)$.

Once expressed symbolically, this relation becomes explicit. As shown in Figure 3, the same source code with IV equal to 10, 20, and 34 produces drastically different binary code, yet their symbolic state expressions all reduce to `eax` $\leftarrow IV \times$ `eax`. Intuitively, compiler transformations preserve the numeric semantics of the constant, and those semantics are directly reflected in this symbolic form.

We use symbolic execution to derive a *symbolic state expression* that precisely captures how a perf-const affects runtime state. Symbolic execution naturally isolates relevant instructions from noise introduced by optimizations and yields the mathematical relationship between IV and architectural state R/M. Resolving IV back to V yields the final expression relating V to runtime machine state.

3.3.2 Deriving Symbolic State Expressions

The first issue is locating the instructions that consume a perf-const. Debug information (e.g., line number) is too coarse-grained—one source line can be compiled into hundreds of instructions, only a small subset of which uses the perf-const. Xkernel therefore begins by identifying seed instructions through binary differencing: we modify the perf-const at the source level and rebuild the kernel, then use the diff to drive symbolic execution. We run symbolic execution on both binaries and derive the transformed value (IV).

Finding seeds. Seed instructions are those that differ when the perf-const value changes in source code. Xkernel finds seeds by assigning the perf-const a magic value that differs from the original value and rebuilding the kernel binary. For example, when $V' = 10$, the seed instruction is `add eax eax`. In principle, instructions in the binary diff are the seeds.

From seeds to symbolic state expressions. Symbolic execution expands from the seeds to derive a converged symbolic state expression that reflects their effect. We exhaustively explore instructions backward and forward until the symbolic state reaches a fixed point while still incorporating the seed effect. The two symbolic executions must converge to the same symbolic form. In theory, extracting IV from a multivariate symbolic expression may require multivariate coefficient matching when multiple symbolic variables (registers or memory locations) appear. However, perf-consts act as simple knobs, and we did not observe such cases in our experience and our extensive evaluation.

A remaining complication is solver canonicalization: symbolic engines rewrite expressions in simplified mathematical forms (e.g., `shl 2` becomes $\times 4$). This matters when the source uses shift operations like ‹‹ or ››. Xkernel explores multiple symbolic execution branches and discards those that cannot preserve a linear relationship with V.

Mapping transformed value IV to the original value V. To recover the relation between IV and V, Xkernel attempts to fit a linear mapping of the form $IV = a \cdot V + b$. We compile additional binaries using another modified value V'', extract the resulting (V, IV) pairs, and solve for a and b via interpolation. When the recovered mapping is non-linear (e.g., $V \times V$), Xkernel reports the case, allowing users to tune the derived $V \times V$ rather than V. However, we never encountered such a case in our evaluation.

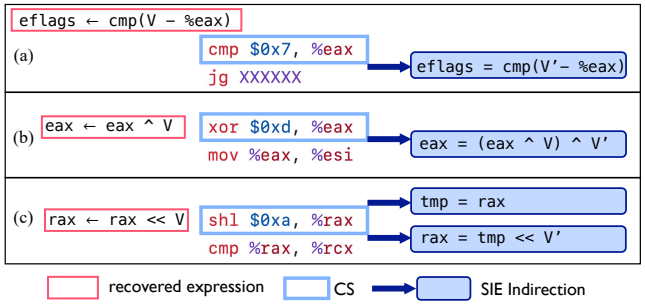


Figure 4: Synthesized SIE indirections for various CSes. An SIE indirection contains pairs of {location, update}. Locations are shown by the blue arrows.

3.4 Synthesizing Indirections

The recovered expression precisely captures how the original value affects runtime state; the next step is to generate instructions using new value. Xkernel turns the instruction replacement problem into a state-update problem. It synthesizes code that overwrites machine state affected by the original execution. The synthesized indirection is in the form of {location, update} pairs; when the specified kernel location is reached, the update code ensures that the effects of the system states is equivalent to that as if the new value V' of the perf-const were used in the kernel source. We introduce a binary representation for each symbolic state expression termed *critical span* and present our synthesis algorithm.

3.4.1 Critical Span

A critical span represents one occurrence of a perf-const at the binary level. Concretely, a CS is a single-entry, single-exit instruction sequence that begins at the first instruction contributing to the perf-const's symbolic state expression and ends either when the resulting state is first consumed or when the basic block ends.

Invariant. Let σ_{in} denote any machine state, CS_v denote the critical span executed with value v, and I denote the synthesized instructions that update v. A value update from v to v' is correct when executing the critical span is observationally equivalent to executing the span with the new value:

$$\forall \sigma_{\text{in}} : \ \text{Exec}\big(\sigma_{\text{in}}, CS_v \circ I\big) \ \equiv \ \text{Exec}\big(\sigma_{\text{in}}, CS_{v'}\big)$$

Here, "$\equiv$" denotes equivalence of externally visible state at the exit of the CS.

Construction. CSes can be automatically constructed for a given perf-const from the recovered symbolic state expression. Xkernel conducts forward and backward slicing to find the start instruction that first consumes the value to the last instruction that can recover the symbolic state expression. A CS is specified as a pair of [start, end] binary addresses.

3.4.2 Synthesis Algorithm

The indirection is synthesized based on the symbolic expression, as exemplified by Figure 4. If R/M affected by the perf-const are not overwritten within the CS, the indirection consists of a single code insertion located after the last instruction in the CS, computing the result using the new value V' and overwriting R/M (Figure 4a).

If R/M are overwritten within the CS, the update depends on whether the overwriting instructions are reversible. For reversible arithmetic or logical operations (Figure 4b), Xkernel synthesizes inverse logic as part of the update.

If the modification cannot be reversed (e.g., due to irreversible arithmetic, bit masking, or information loss), Xkernel generates dual-location: the original value is captured before modification and restored after the CS (Figure 4c).

Note that SIE does not skip the original CS; all instructions in the CS still execute to preserve state equivalence, including instructions that are in the CS but do not depend on the perf-const, e.g., `xor` instructions in Figure 3(a). By construction of the symbolic state expression, SIE updates only the states related to the perf-consts.

SIE relies on the Kprobe mechanism [48] to redirect execution to JITed update code. Because each trigger incurs overhead and different probe locations have different costs, Xkernel adjusts the placement within a CS, using the symbolic state expression to ensure correctness (details in §4).

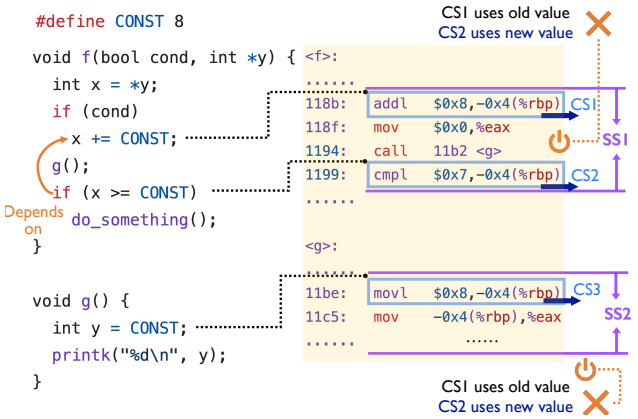


Figure 5: An example of a perf-const's CSes and SSes. Binary representation: SS1[118b, 1199], SS2[11be, 11c8].

3.5 Safe Transition

A symbolic state expression and critical span (CS) of a perf-const enables runtime updates of its value. However, it does not guarantee a *safe* transition of system states from the original value to the new. Figure 5 gives examples where unsafe transition of `CONST` could introduce inconsistent system states. If SIE is enabled when the execution is between CS1 and CS2, the system may execute CS1 with the old value and CS2 with the new. The program point right before `g()`'s return is also unsafe. Although CS3 has no dependency on CS1 or CS2, its call stack lies between them.

The core of safe transition is a well-defined transition scope that reflects the required safety guarantees; transitions are allowed only when execution is outside that scope. We formalize this scope as a safety span (SS), which captures the required safety guarantees and their binary representa-

tion (e.g., SS1 and SS2 in Figure 5). Safety span provides a concrete foundation for safe transition in Xkernel.

Xkernel enforces an update on a perf-const to happen *after* all the instructions that consume its value finish their execution, i.e., the lifetimes of all data objects derived from the perf-consts have expired upon exiting its safe spans. We call this property *side-effect safety*.

3.5.1 Safe Span

A safe span (SS) represents the execution unit in which *transitive dependencies* of a perf-const's critical span (CS) are encapsulated (see Figure 5). An SS includes not only instructions in the CS, but also all subsequent instructions that consume values derived from the perf-const. Concretely, an SS is a single-entry, multi-exit program slice constructed to satisfy a *confinement* property—any instruction that consumes objects that have dependencies with the perf-const's specific version must execute within the SS. So, once execution leaves the SS, no thread retains a dependency on the old value, and switching to a new value is considered safe.

Safety Invariant. Let T denote the set of concerned threads. A transition is safe only when the system is in a state where no thread is active in the SS as per program counters (PCs):

$$\forall t \in T, PC(t) \notin \bigcup SS \Longrightarrow \text{transition}(v \rightarrow v') \text{ is safe.}$$

Xkernel primarily focuses on data dependencies and the SSes are in the form of thin slices [66]. Our empirical analysis shows that data dependencies are sufficient in capturing the required safety of updating most perf-consts. The SS analysis is designed to be pluggable for future extensions, e.g., to incorporate control-flow analysis if needed.

Construction. A safe span of a perf-const can be automatically constructed from its critical span (CS) using forward thin slicing [66]. Any instruction that is transitively flow dependent on the perf-const would be included in the SS. Note that the dependency analysis is inter-procedural, because the perf-const's value can be propagated across function boundaries. If two SSes (constructed from different CSes) overlap, they are merged. An SS is specified as a pair of [start, end1/end2/...] binary address intervals in the top-level frame once the analysis terminates.

Safety properties. As SSes of a perf-const encapsulate the side effects of its value, Xkernel enables *side-effect safety* when updating perf-consts.

Existing Kernel live patching (KLP) mechanisms [1, 60, 67] offer per-thread *version atomicity*—a thread must execute either the old or new code, not their mixture [50]. KLP does so in the unit of functions, assuming every function having a well-defined semantic and being self-contained. In Xkernel, version atomicity is guaranteed through CSes. No existing KLP mechanism provides side-effect safety, as side effects of arbitrary function code are difficult to define.

Xkernel also guarantees multi-threading safety beyond per-thread safety by enforcing *global consistency*—all participating threads must be outside the SSes. In effect, the per-thread property must hold for all threads.

3.5.2 Transition Mechanisms

Xkernel ensures per-thread and multi-threading safety by monitoring the execution at the transition points. The monitors are implemented as kprobes [48], inserted at the entry and exit of each SS. These kprobes are used only during the transition. Once the transition is done, they are removed. The same transitioning mechanism is used for rollback and for updating the value of a perf-const.

Per-thread safety. Unlike KLP that implements timer-based polling, Xkernel detects safe transition points proactively by monitoring execution at the SS boundary. When execution reaches an entry kprobe, Xkernel checks if this SS is deep in the call stack of other SSes (e.g., SS2 in Figure 5) by stack inspection. If so, execution continues and the check is retried upon entering another SS.

Global consistency. Xkernel also supports global consistency for multi-threading safety. Unlike Kpatch [60] that uses a dedicated thread to repeatedly invoke `stop_machine` to opportunistically check if all threads happen to be safe for transition. Xkernel introduces a new mechanism termed *self-convergent transition*. The idea is to use efficient global reference counts maintained by entry and exit kprobes of each SS to track safety. Xkernel invokes `stop_machine` only once to initialize the reference count by scanning all running threads and counting the active SS entry boundaries in their stacks. Afterward, participating threads naturally self-converge when crossing SS boundaries and update the reference count. When the global reference count is zero, no thread resides in any SS, and the transition is safe.

Liveness and timeout. Xkernel may fail to find a safe transition point for a perf-const, causing liveness issues. This can happen when the instrumented path is not executed after the update request, or when one or more threads remain inside an SS and prevent convergence. In either case, Xkernel reports the transition as failed after a configurable timeout.

3.6 Programmable Policy Plane

Xkernel provides simple APIs to support expressive tuning policies for each perf-const. To ensure safety, Xkernel disallows users from directly updating kernel states. Any user policy code (Xk-tune) must follow the stubs auto-generated by an Xkernel tool (`xk-gen`) and passed to an in-kernel Xk-runtime. Xk-tune is written in eBPF style and has eBPF observability. Xk-runtime then compiles Xk-tune into an eBPF kprobe which includes SIE-based kernel-state updates.

Usage model and APIs. To tune a perf-const, the user first uses a command-line tool `xk-gen` to generate an Xk-tune stub which includes the unique ID of the perf-const and a header file that declares Xkernel APIs. The user is expected to implement the policy in the stub using the APIs. An Xk-tune follows the eBPF event-driven model; it is invoked when the perf-const is used at runtime. One perf-const may have mul-

```c
/* handler of xkernel context (xk_ctx) */
typedef const struct xk_ctx * xk_handle_t;
/* User probe definition */
#define XK_TUNE(unique_name, perfconst_id, args...)
/* The set and transition API */
long xk_set(xk_handle_t xk_ctx, u64 val);
bool xk_transition_done(xk_handle_t xk_ctx);
```

Figure 6: Xkernel API.

```c
XK_TUNE(tcp_hystart, "net/ipv4/tcp_cubic.c:L349:3:0") {
    // 1. Safety guard (mandatory)
    if (!xk_transition_done(xk_ctx)) return 0;
    // 2. User policy logic
    //    - Obtain the current network flow's RTT
    struct sock *sk = (struct sock *)PT_REGS_PARM1(ctx);
    struct bictcp *ca = inet_csk_ca(sk);
    u32 cur_rtt = BPF_CORE_READ(ca, curr_rtt);
    //    - Set the value to 1 only when RTT > 80ms
    if (cur_rtt >= 80000) xk_set(xk_ctx, 1);
    return 0;
} /* my_policy.bpf.c */
```

Figure 7: An example of Xk-tune. User-written policy code is highlighted; the rest is from the auto-generated stub.

tiple Xk-tunes according to the number of CS.

Figure 6 shows Xkernel APIs. Two core APIs are `xk_set` to update a perf-const value and `xk_transition_done` to check if the transition is completed. An Xk-tune can read kernel state and invoke `xk_set` to update the perf-const. Figure 7 shows an example of Xk-tune which is used for the case study on a perf-const of TCP CUBIC presented in §5.

Supporting kernel-state updates. eBPF programs are restricted to read-only access to kernel memory to prevent arbitrary writes. To support controlled updates, Xkernel exposes one single kernel-state update API as a BPF kfunc [45]. Kfuncs enable extended functionality through kernel modules. The Xkernel kfunc, `sie_write_kernel`, can be invoked only by Xk-tune (see §4). The kfunc calls the SIE indirections generated for the target perf-const to modify registers in `pt_regs` or kernel memory. The `pt_regs` holds the CPU register state at the moment the probe is triggered and provides a direct reference to the kernel context. Linux already exposes this state to tracing infrastructures (e.g., Kprobe and ftrace), and it is part of a stable tracing ABI [49]. Our kfunc extends this from read access to write access.

Transpiling Xk-tune into eBPF. Each Xk-tune is transpiled into an eBPF kprobe (`BPF_KPROBE`), corresponding to a CS for the perf-const. Figure 8 shows the source-code form of the eBPF code of the Xk-tune in Figure 7. Xk-runtime uses the scope table to locate the SIE indirections for the perf-const and wraps the update logic into generated functions, e.g., `impl_sie_logic_cs1` in Figure 8. A pointer to this function is stored in the `xk_ctx`, effectively implementing `xk_set` for that perf-const. In each generated Kprobe handler, the user-written policy function (e.g., `__user_policy_tcp_hystart`) is invoked. Hence, the policy code has the kernel context (via `ctx`) and the implementation of SIE indirections is provided as embedded function pointers. The safety of Xk-tune is checked by the eBPF verifier.

```c
// 1. Helper: SIE Indirection
static __always_inline void impl_sie_logic_cs1(
    struct pt_regs *regs, u64 val) {
    // Recovered symbolic state expression:
    //   ebx = (regs->bx) >> 3   (original value: 3)
    u64 new_bx = ((regs->bx) << 3) >> val;
    // Writing back to pt_regs using the kfunc
    sie_write_kernel(&regs->bx, sizeof(regs->bx), &new_bx);
}
// 2. Kprobe attachment: CS address (function+offset)
SEC("kprobe/tcp_cubic_hystart_check+0x4F")
int BPF_KPROBE(impl_cs_1) {
    // Wrap raw context into safe xk_ctx
    struct xk_ctx xk_ctx = {
        .regs = ctx,
        .set_fn = &impl_sie_logic_cs1,
    };
    // Call user policy (inlined)
    return __user_policy_tcp_hystart(&xk_ctx, ctx);
} /* my_policy.internal.bpf.c */
```

Figure 8: eBPF kprobe (in a source form) transpiled from the Xk-tune shown in Figure 7.

Transaction semantics. Xkernel supports tuning multiple perf-consts by bundling Xk-tunes in a single file. All Xk-tunes in the file form an atomic transaction—their tuning logic is loaded or unloaded together. Each perf-const may belong to at most one active transaction. The scope table tracks each perf-const status. If any perf-const is in an active transaction, it cannot be tuned in a new transaction.

4 Implementation

We implement Xkernel with ∼1.7K lines of kernel C code for Xk-runtime. Xkernel's toolchain is implemented in about 11K lines of Python, with ∼5K lines for CS and SS analysis via symbolic execution.

CS and SS analysis. We implement a symbolic execution engine specific to CS analysis. Upon capturing the CS, it synthesizes all potential SIE indirections ({location, update}). We implement the SS analysis on kernel bitcode using an LLVM pass. By correlating CS and SS with the `.section` metadata in the binary diff, we resolve the precise target function and offset for Kprobe attachment.

Minimizing Kprobe overhead. Linux Kprobe may have high overhead if breakpoint or single-stepping traps are used [32]. Linux saves the overhead through boosting (skipping single-step traps when post-handlers are absent) and jump-optimization (replacing breakpoints with direct jumps). Xkernel exploits these optimizations to minimize runtime overhead. To enable boosting, Xkernel replaces post-handlers with pre-handlers on the next instruction (`pc+1`). This assumes the next instruction is not a jump destination; this is almost always the case because Xkernel targets the precise point where V enters runtime states. Xkernel maximizes jump-optimizations by strategically attaching kprobes. We implement a new optimization that handles the case when immediate values of perf-consts are directly used in conditional jumps, which Linux Kprobe does not handle. We describe the optimization in Appendix B.

Table 2: Summary of case studies that demonstrate Xkernel's features and capabilities.

Case #	Perf-const	Subsystem	Default	Performance Tuning Regime
Case-1	BLK_MAX_REQUEST_COUNT	Storage (Block layer)	32	Adapting to hardware devices and workload patterns (see §2.1)
Case-2	MAX_SOFTIRQ_RESTART	CPU (Interrupt)	10	Choosing tradeoffs between tail latency and CPU utilization
Case-3	SHRINK_BATCH	Memory (Reclamation)	128	Controlling kernel internal behavior (e.g., maintenance)
Case-4	NR_MAX_BATCHED_MIGRATION	Memory (Page migration)	512	Tuning and reasoning with kernel and hardware observability
Case-5	HYSTART_DELAY_[MAX, MIN, factor]	Network (TCP CUBIC)	[16, 4, 3]	Collective tuning of interdependent perf-consts

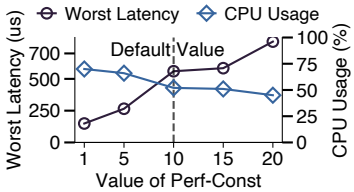


Figure 9: Cost-benefit tradeoff that can be tuned by changing values of MAX_SOFTIRQ_RESTART.

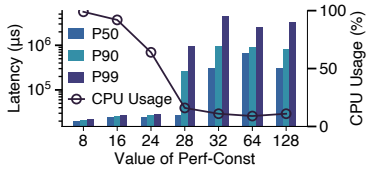


Figure 10: Tuning SHRINK_BATCH can significantly reduce write latency of the workload.

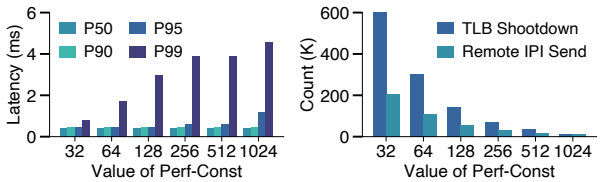


Figure 11: Tuning NR_MAX_BATCHED_MIGRATION to reduce tail latency, which requires understanding TLB shootdown behavior.

Xk-runtime. The runtime is implemented as a kernel module (xk-sie.ko). In addition to transpiling Xk-tunes to eBPF kprobes and loading them, it registers BPF Kfuncs and manages transitions. sie_write_kernel is registered for SIE internal use, and xk_transition_done is exposed to users to check transition status. It tracks completion via global refcounts maintained in Xk-runtime (multi-threading) or BPF_MAP_TYPE_TASK_STORAGE (per-thread). The kernel module attaches monitoring kprobes at SSes and detaches them with the help of a background monitor kthread. For multi-threading safety, it employs stop_machine to initialize refcounts via stack traversal before activating kprobes.

Xkernel tools. We implement a series of userspace tools. xk-build generates the scope table from the source patch. xk-gen specifies the ConstID in the scope table and outputs the Xk-stub. xk-load invokes Xk-runtime to transpile the xk-tune code with standard BPF tool-chains, install resulting BPF kprobes and ensure safe transition. Finally, xk-unload performs a symmetric, ordered teardown.

5 Case Studies

We present case studies that tune perf-consts across kernel subsystems, provide a glimpse at the broader policy space enabled by Xkernel, and discuss the resulting implications for tunability and complexity.

5.1 Performance Benefits and Tuning Rationales

Table 2 summarizes the case studies; we then discuss their performance benefits and tuning rationales.

Case-1: Adapting to hardware and workload patterns. Xkernel enables users to tailor perf-const values according to specific hardware properties and workload patterns. We revisit the perf-const BLK_MAX_REQUEST_COUNT presented in §2.1. Users can use a single Xk-tune program to simultaneously set the value to 128 for the sequential FIO workload on the HDD and to 1 for the RocksDB random workload on the NVMe SSD, obtaining the performance benefits shown in Figure 1. Meanwhile, other workloads can continue using the default value.

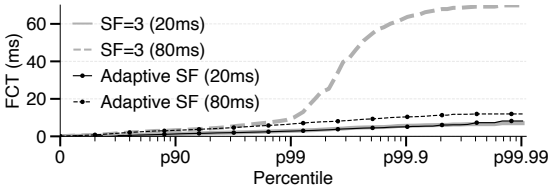


Figure 12: Flow completion time (FCT) of different scaling factors (perf-consts) under different RTTs running NGINX.

Case-2: Choosing performance tradeoffs. Xkernel enables users to choose points along performance tradeoff curves to better satisfy SLOs; such tradeoffs are widespread in OS kernels, with examples such as latency vs. throughput and resource utilization vs. latency.

We show how Xkernel enables balancing the tradeoff between CPU efficiency and tail latency in the CPU (interrupt) subsystem. The perf-const, MAX_SOFTIRQ_RESTART limits how many times the software interrupt (softirq) handler can be rerun in one iteration before yielding, preventing indefinite softirq processing. Increasing its value improves CPU efficiency but may increase tail latency of running tasks. Therefore, it has major impacts on latency-critical workloads, especially when they are colocated with throughput-oriented workloads [57].

We run a latency-critical workload W_l using cyclictest [63] and a throughput-oriented workload W_t on a 4-node cluster with 25 Gbps networking, varying the perf-const values.

Figure 9 shows a clear tradeoff between tail latency of W_l and CPU utilization. The original value (10) provides a reasonable balance with 52% CPU utilization and a worst-case latency of 560 us; achieving the optimal tail latency of 149 us requires paying a penalty of 22% CPU utilization.

Case-3: Controlling kernel internal behavior. Oftentimes, application performance is affected by the kernel's internal behavior such as memory reclamation, which is hard

to control externally. With Xkernel, kernel *internal* behavior can be aligned with the application's access patterns.

We demonstrate this capability using `SHRINK_BATCH`, a perf-const for Linux shrinkers, i.e., kernel callbacks that reclaim slab objects under memory pressure. It controls how many entries a shrinker checks when it scans a slab's LRU list for memory reclamation. `SHRINK_BATCH` was introduced before 2005 [73] and its value (128) has remained unchanged since then [46]. Linux implements 43 shrinkers for different slab objects (zswap entries, inode, dentry, etc). We focus on the zswap-shrinker for the zswap subsystem [52] that is widely used in datacenters [30, 37].

We run a workload that writes data blocks sequentially into a large anonymous mmaped region and periodically overwrites written blocks, which resembles data analytics with a memory budget (constantly triggering zswap). The periodic reuse exposes a classic working-set effect. When the batch size is small, fewer pages are reclaimed and swapped, increasing the chance to reuse data in memory. In contrast, an aggressive batch size may trigger thrashing.

As shown in Figure 10, the original value of 128, despite the relatively low CPU utilization (11%), incurs a large latency penalty. In fact, any values above 24 cause thrashing with unnecessary disk I/O, slowing the user workload. In contrast, smaller values ($\leqslant$24) provide much lower latency.

Case-4: Tuning and reasoning with observability. Written in eBPF, policy code in Xkernel has observability on kernel and hardware metrics and behavior. These internal metrics are valuable for reasoning about performance, confirming tuning effectiveness, and being incorporated directly into tuning policies to decide values.

We use `NR_MAX_BATCHED_MIGRATION`, a perf-const that specifies the batch size of pages migrated across NUMA nodes [58], to demonstrate this feature. The default value (512) was introduced in 2023 [79] and has never been updated. This perf-const controls the amortized costs of TLB shootdowns and page migration. Larger batches amortize per-page migration cost but defer TLB shootdowns, which could increase stall time for concurrent threads accessing pages under migration. Online experiments with Xkernel help us understand the relationship between local memory access and TLB shootdowns.

We run a workload with one group of threads repeatedly touching hot data on a NUMA node, while another group continuously migrates hot pages to that node to shift the hot region over time. This workload mimics memory-intensive applications with moving working sets.

We vary `NR_MAX_BATCHED_MIGRATION` values and measure the tradeoff between latency of memory access to hot pages during migration and TLB shootdowns, as shown in Figure 11. A small value can effectively reduce tail latency by prioritizing memory access over migration, with a penalty of increased TLB shootdowns. Tuning this perf-const values requires observability of TLB behavior.

Case-5: Collective tuning of interdependent perf-consts. With Xkernel, users can jointly tune multiple perf-consts that collectively decide fine-grained behavior with transaction semantics (§3.6).

We demonstrate this capability by tuning the behavior of TCP CUBIC's HyStart delay-based congestion window. When a connection is in slow start, TCP CUBIC checks whether the current RTT exceeds a delay threshold computed from three perf-consts (two defined as macros and one used as an operand in a shift operation). If the measured delay exceeds this threshold, CUBIC assumes congestion and exits slow start. The optimal threshold is highly dependent on RTT. With high RTTs, the current perf-consts are overly sensitive and trigger premature slow-start exit; with low RTTs, they are too conservative and delay exiting slow start. These perf-consts were introduced in 2008 [26] and changed three times [17, 18, 25]. Prior work reported the importance of tuning them [27], but they remain constants.

We first ran a microbenchmark to understand scaling factors (SFs) under different RTTs. To provide sufficient room for the scaling factor to take effect under high RTT, we change `HYSTART_DELAY_MAX` from 16*ms* to 32*ms*. We find that SF=1 significantly reduces tail latency for slow flows, but increases tail latency for fast flows compared to SF=3 (default). Based on this observation, we implement a selective tuning policy that dynamically adjusts the SF of TCP Cubic for flows with long RTT (Figure 7).

We show performance gains of tuning these perf-consts collectively on NGINX, deployed as a web server for photo and video content [55]. We run a mixed workload of concurrent 20*ms* and 80*ms* flows, representing fast and slow connections. As shown in Figure 12, Xkernel achieves a significant FCT reduction (81% at P99.99) for long-RTT flows while maintaining performance parity for short-RTT flows.

5.2 A Glimpse into Xkernel-Enabled Policies

Xkernel enables powerful tuning policies and creates opportunities for policy innovation. We briefly discuss three directions that hint at a broader unexplored policy space.

Flexible granularity. One major benefit of Xkernel is its flexible granularity in applying tuning policies. Unlike hard-coded granularities, such as processes, cgroups, or devices, Xkernel allows users to define tuning granularity using conditions unforeseen before binary deployment.

As demonstrated by the cases in Table 2, Xkernel allows values to be set at the granularity of a given application (potentially spanning multiple processes or threads), a specific device type, or a set of network flows. Users can update these granularities by changing the condition under which `xk_set` is applied in a new Xk-tune. The zswap shrinker (Case-3) is an interesting example: 43 shrinkers across subsystems share the same perf-const as their batch size, yet Xkernel allows customizing batch sizes for different shrinkers at runtime by checking the caller's name (see Appendix D).

```
XK_TUNE(...) {
    // Get the pid upon invocating this kernel code path
    u64 pid = bpf_get_current_pid_tgid() & 0xFFFFFFFF;
    // Leverage hints from the application process
    int *hint = bpf_map_lookup_elem(&hint_map, &pid);
    if (hint) {  xk_set(xk_ctx, 1);  }
}
```

Figure 13: An example of an application-informed policy. The application communicates the hint through an eBPF map (`hint_map`), and the Xk-tune uses this hint to apply the tuning value to selected threads.

```
/* Track consecutive failed attempts to merge adjacent
   blocks and store the count in fail_cnt for use as a
   tuning heuristic */
SEC("kretprobe/blk_attempt_plug_merge")
int BPF_KRETPROBE(blk_attempt_plug_merge_ret, long ret) {
    int *fail_cnt = bpf_task_storage_get(&fail_cnt_map,
        bpf_get_current_task_btf(), ...);
    if (ret == 0 && fail_cnt) (*fail_cnt)++;
    else *fail_cnt = 0;
}
/* Retrieve the heuristic and tune accordingly */
XK_TUNE(blk_add_rq_to_plug, "block/blk.h:L312:32:0") {
    int *fail_cnt = bpf_task_storage_get(&fail_cnt_map,
        bpf_get_current_task_btf(), ...);
    // Heuristic: treat the workload as random when many
    // (16, an ad-hoc threshold) consecutive merges fail
    if (fail_cnt && *fail_cnt >= 16) xk_set(xk_ctx, 1);
}
```

Figure 14: An Xk-tune program using ad-hoc workload heuristics. It instruments `blk_attempt_plug_merge` with a kretprobe to track historical merge failures.

Application-informed policies. Xkernel enables a class of application-informed tuning policies that better coordinate with application-level semantics and knowledge of workload patterns. This feature relies on eBPF's mature user-kernel communication. Figure 13 illustrates one such collaboration, where a workload-pattern hint is passed via an eBPF map. For example, a RocksDB application can use such hints to distinguish foreground threads from compaction threads.

Ad-hoc tuning heuristics. Xkernel enables ad-hoc tuning heuristics that are easier to identify once workloads and deployment scenarios are known, without requiring such heuristics to be baked into the source code. Figure 14 illustrates one such heuristic for Case-1, where kernel-internal observability metrics are used to infer whether the workload is sequential or random. Moreover, because an Xk-tune runs on each execution, Xkernel supports adaptive policies, such as linear or exponential changes to the tuning value.

5.3 Tunability and Complexity

Choosing values for perf-consts faces inherent complexity: the best value depends on the workload, hardware, and policy objective in a given deployment. Existing practice hides this complexity behind fixed compile-time choices, but this can leave significant performance opportunities fundamentally unattainable. In contrast, Xkernel makes perf-consts tunable and programmable in a unified framework, exposing the broad and powerful policy space discussed above.

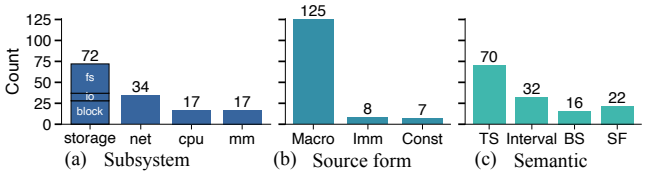


Figure 15: Characteristics of the evaluated perf-consts. The categories are specified in Table 1.

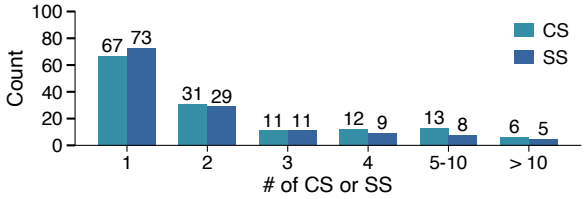


Figure 16: Number of CS and SS per perf-const.

Such exposure does not create new complexity; rather, it allows users, for the first time, to tackle this complexity in a principled manner. Exposing only a subset of perf-consts is insufficient when multiple constants jointly affect system behavior; one untunable perf-const can limit the effectiveness of others. We believe managing the full perf-consts set with Xkernel opens a promising research direction.

6 Evaluation

To evaluate the generality, safety, and efficiency of Xkernel, we collected 140 perf-consts from four Linux subsystems. These perf-consts cover common semantics and source-code forms, as shown in Figure 15. Note that the evaluated perf-consts are a small subset of all perf-consts in Linux, though the number is comparable to that of `sysctl` performance-related knobs. Evaluations run on a machine with 128 GB RAM, a 28-core Intel Xeon Gold 5512U CPU (2.1 GHz), and two 800 GB NVMe-Gen4 SSDs on CloudLab [19].

Xkernel supports all but one (99.3%) perf-consts. The one Xkernel fails to support is due to limitations of Kprobe (see Appendix C for more details).

We show that: (1) SIE incurs negligible runtime overhead; (2) Xkernel supports millisecond-scale policy updates and safe transitions; and (3) the cost of offline, one-time static analysis is about 18 minutes per perf-const.

6.1 Characteristics of Critical and Safe Spans

We first present the characteristics of critical spans (CSes) and safe spans (SSes) of the evaluated perf-consts. The information helps interpret our experiment results. As shown in Figure 16, the 140 perf-consts have 367 CSes in total. Most constants are highly localized: 48% map to a single CS, and 86% map to fewer than five. A long tail also exists: in 4% of cases, a constant maps to more than ten CSes; `DEF_PRIORITY` and `NFS4_POLL_RETRY_MAX` have the highest counts (16 and 17) due to aggressive inlining of their enclosing callers.

Out of the 367 CSes, 82 have symbolic values IV that differ from the perf-const value V in the source representation (§3.3.2). Xkernel correctly recovers the symbolic relations

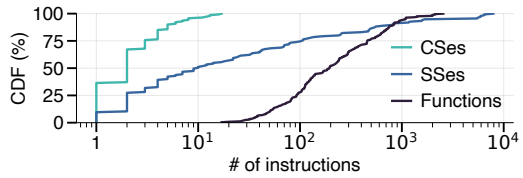


Figure 17: Size of CSes, SSes, and Functions. CS size is measured precisely. SS size is approximated by summing the instructions in the span, including the callees. For functions, we count only the instructions in the function body (excluding callees), so the value is an underestimate.

for all cases. Only three CSes require dual-location indirections to handle irreversible updates (§3.4.2).

We obtained 300 SSes in total from the 367 CSes. As shown in Figure 16, 26 of the perf-consts exhibit CSes that have data dependencies, resulting in a smaller number of SSes compared to their number of CSes.

Figure 17 compares the sizes of CSes, SSes, and kernel functions. CSes are small, mostly containing a single instruction; only 12 contain two, and one each at lengths 3, 4, 5, and 7. This confirms that the point where a perf-const first affects runtime state is simple and compact. The median SS contains only 10 instructions, showing that safe transitions are usually confined to a small scope. SS size, however, has a long tail significantly larger than CS size, reflecting the inherent complexity of kernel data dependencies. The largest SS in our dataset is about 8K instructions. SSes make these dependencies explicit. Functions expand the scope unnecessarily and are much larger than SS except in extreme cases, but with weaker safety guarantee than SSes.

6.2 SIE Overhead

We evaluate SIE overhead along three dimensions: the per-trigger CPU cost, the frequency at which SIE kprobes fire, and the number of SIE kprobes enabled at the same time.

Per-trigger cost (CPU cycles). An SIE invocation includes the underlying kprobe cost plus SIE-specific work: checking transition state (O1), executing indirection (O2), and, when needed, reading kernel state (O3) or accessing BPF maps (O4) for user-space policy interaction. We attach SIE kprobes at different offsets inside a kernel function (`vfs_write`) to compare jump-optimized and INT3-based probes. We use a single-byte write workload with an intentionally short kernel path bottlenecked at `vfs_write`. Table 3 shows that the kprobe mechanism dominates: an empty jump-optimized kprobe adds 168 cycles (for 84.2% of evaluated CSes), whereas an empty INT3-based kprobe adds 1765 cycles. O1–O4 add little additional cost.

Trigger frequency. We evaluate the impact of trigger frequency using a controlled benchmark. We implement a multithreaded workload that uses `io_uring` and performs asynchronous one-byte writes to `/dev/null` with an SIE kprobe attached to `io_issue_sqe`. We vary both the offered IOPS (to control the trigger rate of SIE) and the amount of per-operation computation. Figure 18 shows the slowdown at

Operation	Cycles/Op	Overhead(%)	Cases #
Baseline	946		
Empty BPF Kprobe (JMP)	1114	+18%	
Xkernel (JMP+O1)	1146	**+21%**	
Xkernel (JMP+O1+O2)	1167	**+22%**	309
Xkernel (JMP+O1+O2+O3)	1186	**+25%**	(84.2)%
Xkernel (JMP+O1+O2+O3+O4)	1187	**+25%**	
Empty BPF Kprobe (INT3)	2711	+187%	
Xkernel (INT3+O1)	2752	**+191%**	
Xkernel (INT3+O1+O2)	2801	**+196%**	58
Xkernel (INT3+O1+O2+O3)	2795	**+195%**	(15.8)%
Xkernel (INT3+O1+O2+O3+O4)	2799	**+196%**	

Table 3: CPU cycles breakdown of one SIE invocation. O1: Transition state check; O2: Indirection's execution; O3: Kernel Memory Read; O4: BPF MAP Access

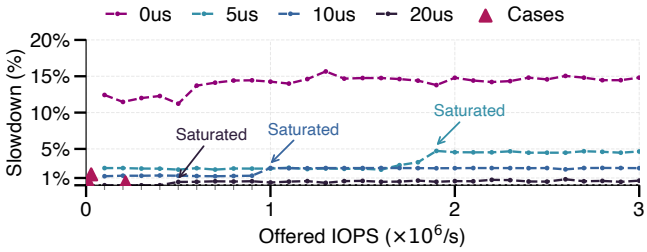
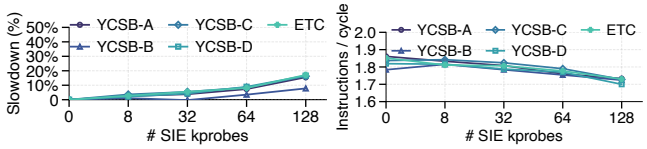


Figure 18: Slowdown of median latency. The trigger rate of SIE kprobes is equal to the offered IOPS before saturation and remains unchanged afterwards.



(a) Throughput slowdown (b) Instructions per cycle (IPC)

Figure 19: Redis slowdown with multiple SIE kprobes.

different trigger rates; we also annotate the data points from case studies (§5) by triangles.

When the workload performs essentially no work (0μs) in each operation, the slowdown caused by SIE is 15%; the slowdown drops to 5% and 2% when each operation includes 5μs and 10μs of computation, respectively. Once per-operation processing reaches 20μs, the overhead falls below 1%. The slowdown curve remains relatively flat as trigger rate increases, indicating that the overhead scales predictably even under high execution frequency of SIE. These results show that Xkernel introduces negligible overhead, even when triggered millions of times per second. In practice, applications tend to execute their own logic for more than 20μs, making the overhead even more negligible.

Multiple SIE kprobes. We evaluate how the per-trigger cost accumulates when many tunable constants are active under a real application. We run Redis [61] with YCSB [9] and Facebook ETC workloads [3], and vary the number of enabled SIE kprobes; each SIE kprobe is placed on the critical path. Figure 19 reports throughput and instructions per cycle (IPC). With 32 SIE kprobes, throughput drops by at most 4%. Even with 128 SIE kprobes, throughput slowdown

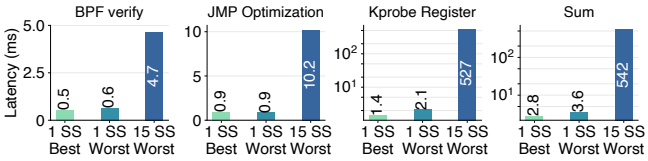


Figure 20: Policy-update time. Best case: only one kprobe is required per CS. Worst case: two additional kprobes are needed to ensure global consistency.

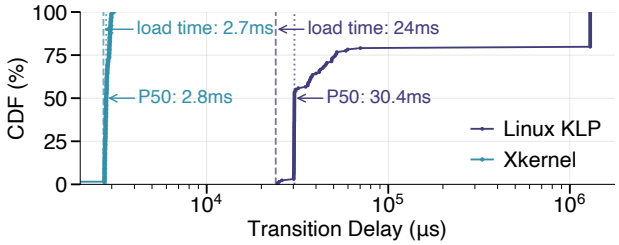


Figure 21: Transition time of Xkernel using CS versus Linux KLP. The target function has 898 instructions.

is 7–14% across workloads, while IPC drops by about 6%. These results show that Xkernel is practical to tune many perf-consts with modest overhead.

Finally, a tuning should be enabled when its benefit outweighs the pure cost of SIE invocations; in our case studies, the benefits usually far outweigh this overhead.

6.3 Policy-update and Transition Time

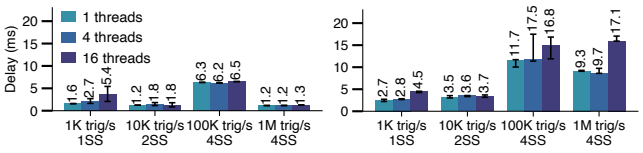
We measure the time to load a policy program (Xk-tune) and the time to transit from the original value to a new one.

Policy-update time. Figure 20 shows the breakdown of loading an Xk-tune program, including BPF verification, jump-optimization, and Kprobe registration. The main cost is Kprobe registration, and thus the number of kprobes has a major impact on policy load time. Even in the case where the perf-const has 15 SSes, Xkernel bounds the policy-update time to 542 milliseconds, meeting our design goal.

Transition time. Xkernel provides the near-optimal transition time for per-thread version atomicity and side-effect safety, and is scalable with many concurrent threads.

Per-thread version atomicity. We compare Xkernel with Linux KLP [50] on version atomicity; both support version atomicity. We tune a perf-const which controls TCP backlog size [51] and measure transition time with an `iperf3` [71] workload of 128 flows (threads). Figure 21 shows the CDF of the *end-to-end* latency, including the policy-update time (2.7 *ms* for Xkernel and 24 *ms* for KLP modulo 7-minute patch-generation time). The median latency for KLP and Xkernel is 2.8ms and 30.4ms, respectively. This shows that CS is a much more efficient transition unit than function.

Per-thread side-effect safety. Figure 22a shows transition time of four representative cases with different triggering rates of Xk-tunes and the number of threads that load Xk-tunes. We do not compare with KLP which does not support side-effect safety. In all cases, the transition time is less than 10 milliseconds. A per-thread transition completes as soon



(a) Per-thread (b) Global consistency

Figure 22: Transition time of side-effect safety.

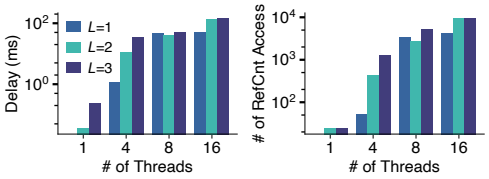


Figure 23: The impact of SS size with global consistency. L is the length of function calls to reach the CS of a perf-const; it is used to control the SS size.

as the first kprobe fires upon entering the first SS. We see no clear correlation of the triggering rate and the number of the SSes with the transition time—all configurations trigger at least once per millisecond and complete the transition immediately once the entry kprobe fires.

Global consistency. Figure 22b shows transition time with global consistency on side-effect safety with multiple threads. The time increases compared to per-thread safety. We do not compare with Kpatch [60] because its global-consistency mode was later deprecated [59] and it does not support side-effect safety.

To understand the time in more depth, we run a controlled experiment using the microbenchmark in §6.2, varying the SS size (controlled by L levels of call stacks) and the number of threads. Figure 23 shows that transition time increases with higher concurrency (fewer safe points) and larger SS sizes. The number of reference-count updates shows the same pattern: a larger number of updates indicates a lower likelihood of leaving the SS. Overall, the transition time is low (144*ms*) even under high concurrency (16 threads).

6.4 Offline Static Analysis Time

Xkernel incurs a one-time cost per perf-const for static analysis; after that, the perf-const can be tuned online anytime. This offline process consists of two kernel compilations, symbolic execution to recover symbolic state expressions and CSes, and analysis to construct SSes. On average, the process takes 18 minutes per perf-const. The analysis of different perf-consts can be done in parallel using spot instances. Compilation takes about seven minutes (two compilation runs executed in parallel across 56 threads). The remaining time is dominated by SS construction, which varies significantly based on the dependency structure. On average 11 $\pm$ 20 minutes and up to 124 for the most complex case.

Overall, Xkernel's offline cost is linear in the number of perf-consts in the kernel source code. For N constants, constructing the scope table requires $2N$ recompilations. In contrast, KLP or source-level recompilation requires one binary per target value; with M possible values per constant, all

combinations can require up to M^N builds. The scope table is reusable across machines running the same kernel binary, allowing cost amortization across a deployment.

7 Discussion

Applicable constants of Xkernel. Xkernel prefers safety over completeness, while still supporting a broad range of perf-consts. Xkernel does not target constants which can directly change kernel memory layout (e.g., array sizes or struct padding), as such changes can introduce dangerous pointer arithmetic and are challenging to make safe. Xkernel rejects the constants it cannot handle upon detection. The layout changes can be reliably detected by reading debug information such as DWARF (e.g., `pahole`). Xkernel can also detect and reject dead-code elimination cases where the offline analysis finds the constant absent from one binary.

Concurrency safety. A CS is a very short instruction sequence (Figure 17) that materializes a constant into runtime state; it typically contains no lock operations, blocking primitives, or memory barriers. An XK-tune only changes this materialization, so it preserves the original concurrency semantics and does not introduce the risk of races from breaking locks. The kprobe mechanism also provides a solid foundation, as probe insertion is atomic.

Choosing a good value. To maximize flexibility, Xkernel does *not* impose any built-in bounds on new values (except for architectural constraints such as register width limits). Such bounds belong to the performance semantics of each perf-const, not the SIE mechanism. Users can encode range checks, device-specific constraints, specifications defined by RFCs, and any other value checks in XK-tune. Xkernel treats them as policy, keeping mechanism and policy separate.

The focus of this work is the mechanism and interface to realize principled OS tunability (§2.3), not the policies on deciding the optimal values of each perf-const. We expect the new opportunities and flexibility enabled by Xkernel to inspire many new tuning techniques (e.g., with AI [41]).

Maintenance of offline results. The scope table is tied to the exact kernel source, compiler version, and build configurations to match the running binary. Xkernel assumes that the compiler version and build configurations of the running kernel are available. This holds because (1) they are inherently known for customized kernels, and (2) for out-of-the-box kernels, they can be retrieved from official sources.

Out-of-tree modules and drivers. Xkernel is not limited to in-tree kernel code. SIE also supports loadable kernel modules and vendor drivers with available source code. Each scope table entry is tied to a specific module or driver version and must be regenerated after it is rebuilt or updated.

8 Related Work

OS and system performance tuning. Tuning has long been essential for OS performance [2, 24, 64]. Existing practice, however, relies on the limited set of knobs exposed through `sysctl` and `sysfs`, leaving substantial performance potential unexplored. Kernel configuration systems (e.g., Kconfig [47]) customize the OS but are designed for feature selection [36, 65, 68, 69], not performance tuning, since they operate purely at compile time. Recent techniques using AI agents show promise in improving search strategies [41, 42], but they are limited to a fundamentally constrained tuning space. We view Xkernel as a new foundation to expand this space and advance tuning techniques. Our techniques and principled tunability can potentially benefit many other systems that actively explore tuning policies [34, 35, 43, 70].

Programmable kernel extensions. Kernel extensibility has long been a central topic in OS research [15]. This area has seen renewed momentum with the rise of eBPF, which has been used to customize kernel behavior across many subsystems [11, 29, 31, 33, 78, 81, 83]. These efforts share our motivation of adapting kernel behavior to diverse workloads and hardware. Our work focuses on perf-consts and principled tunability. While Xkernel uses eBPF as its policy interface, its core contribution is SIE, which enables principled in-situ tuning. TCP-BPF [8] provides a similar capability by allowing TCP parameters to be tuned through an eBPF program. However, it is limited to the TCP subsystem and requires substantial engineering effort to restructure kernel code paths. In contrast, Xkernel is general to any perf-consts.

OS and program live update. SIE shares attributes with live-update systems, but is designed specifically for perf-consts—users (or agents) specify a value and Xkernel synthesizes the update instructions. OS live update has been widely studied [7, 10, 22, 23] and has led to products such as Ksplice [1], Kpatch [60], and KLP [50]. As discussed, these systems treat functions as the unit of version atomicity, making them a poor match for perf-consts. User-space live-update systems also operate at function-level unit [28, 62]. A recent effort, PlugSched [53] provides live updates for CPU scheduling but depends on policy-specific understanding of kernel scheduler data structures to reconstruct state. In contrast, Xkernel provides a general and principled mechanism without relying on specific kernel data structures.

9 Conclusion

We presented Xkernel which uses Scoped Indirect Execution (SIE), the first mechanism to achieve principled OS tunability and enable safe, fast in-situ tuning of *any* perf-consts in OS kernels. Xkernel offers a new path for OS performance and opens untapped performance opportunities.

Acknowledgement

We thank our shepherd and the anonymous reviewers for their insightful comments, and Shawn Zhong for his help at an early stage. This work is supported by National Natural Science Foundation of China (NSFC) under Grant No. 62132007 and No. 62221003, and by NSF CNS-2145295.

References

[1] ARNOLD, J., AND KAASHOEK, M. F. Ksplice: Automatic Rebootless Kernel Updates. In *Proceedings of the EuroSys Conference (EuroSys '09)* (April 2009).

[2] ARPACI-DUSSEAU, A. C., ARPACI-DUSSEAU, R. H., BURNETT, N. C., DENEHY, T. E., ENGLE, T. J., GUNAWI, H. S., NUGENT, J. A., AND POPOVICI, F. I. Transforming policies into mechanisms with infokernel. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP '03)* (October 2003).

[3] ATIKOGLU, B., XU, Y., FRACHTENBERG, E., JIANG, S., AND PALECZNY, M. Workload analysis of a large-scale key-value store. In *Proceedings of the 2012 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS '12)* (London, United Kingdom, June 2012).

[4] AXBOE, J. Explicit block device plugging. https://lwn.net/Articles/438256/.

[5] AXBOE, J. Fio – flexible I/O tester. https://github.com/axboe/fio.

[6] AXBOE, J. Linux kernel commit – block: bump max plugged deferred size from 16 to 32. https://github.com/torvalds/linux/commit/ba0ffdd8ce48ad7f7e85191cd29f9674caca3745.

[7] BAUMANN, A., HEISER, G., APPAVOO, J., DA SILVA, D., KRIEGER, O., WISNIEWSKI, R. W., AND KERR, J. Providing dynamic update in an operating system. In *Proceedings of the USENIX Annual Technical Conference (USENIX '05)* (April 2005).

[8] BRAKMO, L. TCP-BPF: Programmatically tuning TCP behavior through BPF. In *The Technical Conference on Linux Networking (NetDev 2.2)* (November 2017).

[9] BRIAN FRANK COOPER. Yahoo! Cloud Serving Benchmark. https://github.com/brianfrankcooper/YCSB, 2010.

[10] CHEN, H., CHEN, R., ZHANG, F., ZANG, B., AND YEW, P.-C. Live updating operating systems using virtualization. In *Proceedings of the 2nd International Conference on Virtual Execution Environments (VEE '06)* (June 2006).

[11] CHEN, Z., MENG, Q., LAO, C., LIU, Y., REN, F., YU, M., AND ZHOU, Y. eTran: Extensible kernel transport with eBPF. In *Proceedings of the 22nd USENIX Symposium on Networked Systems Design and Implementation (NSDI '25)* (April 2025).

[12] CORBET, J. 5.19 fixes tags and the commits they fix. https://lwn.net/Articles/902938/.

[13] CORBET, J. A rough patch for live patching. https://lwn.net/Articles/634649/.

[14] DALTON, A., YU, C., AND KIM, J. Linux kernel commit – f2fs: increase the limit for reserve_root. https://github.com/torvalds/linux/commit/da35fe96d12d15779f3cb74929b7ed03941cf983.

[15] DRUSCHEL, P., PAI, V. S., AND ZWAENEPOEL, W. Extensible kernels are leading OS research astray. In *The Sixth Workshop on Hot Topics in Operating Systems (HotOS VI)* (May 1997).

[16] DUKKIPATI, N., REFICE, T., CHENG, Y., CHU, J., HERBERT, T., AGARWAL, A., JAIN, A., AND SUTIN, N. An argument for increasing TCP's initial congestion window. *ACM SIGCOMM Computer Communication Review 40*, 3 (June 2010), 26–33.

[17] DUMAZET, E., AND MILLER, D. S. Linux kernel commit – tcp_cubic: refine Hystart delay threshold. https://github.com/torvalds/linux/commit/42eef7a0bb0989cd50d74e673422ff98a0ce4d7b.

[18] DUMAZET, E., AND MILLER, D. S. Linux kernel commit – tcp_cubic: switch bictcp_clock() to usec resolution. https://github.com/torvalds/linux/commit/cff04e2da308c522f654237b45dd64248fe8d1fa.

[19] DUPLYAKIN, D., RICCI, R., MARICQ, A., WONG, G., DUERIG, J., EIDE, E., STOLLER, L., HIBLER, M., JOHNSON, D., WEBB, K., AKELLA, A., WANG, K., RICART, G., LANDWEBER, L., ELLIOTT, C., ZINK, M., CECCHET, E., KAR, S., AND MISHRA, P. The design and operation of CloudLab. In *Proceedings of the USENIX Annual Technical Conference (USENIX '19)* (Renton, WA, July 2019).

[20] DUYCK, A., DUMAZET, E., AND MILLER, D. S. Linux kernel commit – net: allow gso_max_size to exceed 65536. https://github.com/torvalds/linux/commit/7c4e983c4f3cf94fcd879730c6caa877e0768a4d.

[21] FACEBOOK. RocksDB. http://rocksdb.org/.

[22] GIUFFRIDA, C., KUIJSTEN, A., AND TANENBAUM, A. S. Safe and automatic live update for operating systems. In *Proceedings of the 18th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '13)* (March 2013).

[23] GOULLON, H., ISLE, R., AND LÖHR, K.-P. Dynamic restructuring in an experimental operating system. *IEEE Transactions on Software Engineering SE-4*, 4 (July 1978), 298–307.

[24] GREGG, B. *Systems performance: enterprise and the cloud*. Pearson Education, 2014.

[25] HA, S. Linux kernel commit – tcp_cubic: make the delay threshold of HyStart less sensitive. https://github.com/torvalds/linux/commit/2b4636a5f8ca547000f6aba24ec1c58f31f4a91d.

[26] HA, S., AND MILLER, D. S. Linux kernel commit – [TCP] CUBIC v2.3. https://github.com/torvalds/linux/commit/ae27e98a51526595837ab7498b23d6478a198960.

[27] HA, S., AND RHEE, I. Hybrid slow start for high-bandwidth and long-distance networks. In *Proceedings of the Sixth International Workshop on Protocols for FAST Long-Distance Networks (PFLDnet '08)* (March 2008).

[28] HAYDEN, C. M., SMITH, E. K., DENCHEV, M., HICKS, M., AND FOSTER, J. S. Kitsune: efficient, general-purpose

dynamic software updating for C. In *ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA '12)* (October 2012).

[29] HEO, T. sched_ext schedulers and tools. https://github.com/sched-ext/scx.

[30] HEO, T., SCHATZBERG, D., NEWELL, A., LIU, S., DHAKSHINAMURTHY, S., NARAYANAN, I., BACIK, J., MASON, C., TANG, C., AND SKARLATOS, D. IOCost: Block IO Control for Containers in Datacenters. In *Proceedings of the 27th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '22)* (February 2022).

[31] HØILAND-JØRGENSEN, T., BROUER, J. D., BORKMANN, D., FASTABEND, J., HERBERT, T., AHERN, D., AND MILLER, D. The eXpress data path: fast programmable packet processing in the operating system kernel. In *Proceedings of the 14th International Conference on Emerging Networking EXperiments and Technologies (CoNEXT '18)* (December 2018).

[32] JIA, J., LE, M., AHMED, S., WILLIAMS, D., JAMJOOM, H., AND XU, T. Fast (Trapless) Kernel Probes Everywhere. In *Proceedings of the USENIX Annual Technical Conference (USENIX '24)* (Santa Clara, CA, July 2024).

[33] KAFFES, K., HUMPHRIES, J. T., MAZIÈRES, D., AND KOZYRAKIS, C. Syrup: User-defined scheduling across the stack. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP '21)* (October 2021).

[34] KANELLIS, K., DING, C., KROTH, B., MÜLLER, A., CURINO, C., AND VENKATARAMAN, S. LlamaTune: sample-efficient DBMS configuration tuning. *arXiv preprint arXiv:2203.05128* (2022).

[35] KANELLIS, K., YADALAM, S., VENKATARAMAN, S., AND SWIFT, M. Striking the right chord: Parameter tuning in memory tiering systems. In *Proceedings of the 3rd Workshop on Disruptive Memory Systems (DIMES '25)* (October 2025).

[36] KUO, H.-C., CHEN, J., MOHAN, S., AND XU, T. Set the configuration for the heart of the OS: On the practicality of operating system kernel debloating. In *Abstracts of the 2020 SIGMETRICS/Performance Joint International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS '20)* (June 2020).

[37] LAGAR-CAVILLA, A., AHN, J., SOUHLAL, S., AGARWAL, N., BURNY, R., BUTT, S., CHANG, J., CHAUGULE, A., DENG, N., SHAHID, J., THELEN, G., YURTSEVER, K. A., ZHAO, Y., AND RANGANATHAN, P. Software-Defined Far Memory in Warehouse-Scale Computers. In *Proceedings of the 24th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '19)* (April 2019).

[38] LI, S. Linux kernel source comment: "it just happens to work well, that's all.". https://elixir.bootlin.com/linux/v6.14/source/mm/swap_state.c#L587.

[39] LI, S., AND AXBOE, J. Linux kernel commit – blk-throttle: choose a small throtl_slice for SSD. https://github.com/torvalds/linux/commit/d61fcfa4bb18992dc8e171996808e1034dc643bb.

[40] LI, S., AND AXBOE, J. Linux kernel commit – block: avoid building too big plug list. https://github.com/torvalds/linux/commit/55c022bbddb2c056b5dff1bd1b1758d31b6d64c9.

[41] LIARGKOVAS, G., JABRAYILOV, V., FRANKE, H., AND KAFFES, K. An Expert in Residence: LLM Agents for Always-On Operating System Tuning. In *Workshop on ML for Systems at NeurIPS* (December 2025).

[42] LIARGKOVAS, G., SODHI, P. S., AND KAFFES, K. Set it and forget it: Zero-mod ML magic for Linux tuning. In *Proceedings of the 4th Workshop on Practical Adoption Challenges of ML for Systems (PACMI '25)* (October 2025).

[43] LIN, Q., ZHANG, Z., THAKKAR, V., SUN, Z., ZHENG, M., AND CAO, Z. StorageXTuner: An LLM agent-driven automatic tuning framework for heterogeneous storage systems. *arXiv preprint arXiv:2510.25017* (2025).

[44] LINUX DEVELOPERS. ABI stable symbols. https://www.kernel.org/doc/html/v6.14/admin-guide/abi-stable.html.

[45] LINUX DEVELOPERS. BPF kernel functions (kfuncs). https://www.kernel.org/doc/html/v6.14/bpf/kfuncs.html.

[46] LINUX DEVELOPERS. Git blame of SHRINK_BATCH: fixed value since 2005. https://github.com/torvalds/linux/blame/e538a582097878c536c68002c79722e4a037c080/mm/vmscan.c#L832.

[47] LINUX DEVELOPERS. Kconfig language. https://www.kernel.org/doc/html/v6.14/kbuild/kconfig-language.html.

[48] LINUX DEVELOPERS. Kernel Probes (Kprobes). https://www.kernel.org/doc/html/v6.14/trace/kprobes.html.

[49] LINUX DEVELOPERS. Kprobes preserve kernel-state context. https://elixir.bootlin.com/linux/v6.14/source/tools/lib/bpf/bpf_tracing.h#L813.

[50] LINUX DEVELOPERS. Livepatch. https://www.kernel.org/doc/html/v6.14/livepatch/livepatch.html.

[51] LINUX DEVELOPERS. Perf-const in tcp_sendmsg_locked. https://elixir.bootlin.com/linux/v6.14/source/net/ipv4/tcp.c#L1162.

[52] LINUX DEVELOPERS. zswap. https://www.kernel.org/doc/html/v6.14/admin-guide/mm/zswap.html.

[53] MA, T., CHEN, S., WU, Y., DENG, E., SONG, Z., CHEN, Q., AND GUO, M. Efficient scheduler live update for linux kernel with modularization. In *Proceedings of the 28th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '23)* (Vancouver, Canada, March 2023).

[54] MOCHEL, P., AND MURPHY, M. The filesystem for exporting kernel objects. https://www.kernel.org/doc/html/v6.14/filesystems/sysfs.html, 2003.

[55] MURALIDHAR, S., LLOYD, W., ROY, S., HILL, C., LIN, E., LIU, W., PAN, S., SHANKAR, S., SIVAKUMAR, V., TANG, L., AND KUMAR, S. f4: Facebook's warm BLOB storage system. In *Proceedings of the 11th Symposium on Operating Systems Design and Implementation (OSDI '14)* (October 2014).

[56] PEDRONI, P., VALENTE, P., AND AXBOE, J. Linux kernel commit – block, bfq: boost throughput by extending queue-merging times. https://github.com/torvalds/linux/commit/7812472f973047a886e4ed9a91d98d6627dd746f.

[57] PREKAS, G., KOGIAS, M., AND BUGNION, E. ZygOS: Achieving low tail latency for microsecond-scale networked tasks. In *Proceedings of the 26th ACM Symposium on Operating Systems Principles (SOSP '17)* (October 2017).

[58] RAYHAN, Y., AND AREF, W. G. Revisiting page migration for main-memory database systems. *arXiv preprint arXiv:2503.17685* (2025).

[59] RED HAT. Discussions on kpatch bugs and limitations. https://github.com/dynup/kpatch/pull/1355.

[60] RED HAT. kpatch: dynamic kernel patching. https://github.com/dynup/kpatch.

[61] REDIS. Redis. http://redis.io/.

[62] ROMMEL, F., DIETRICH, C., FRIESEL, D., KÖPPEN, M., BORCHERT, C., MÜLLER, M., SPINCZYK, O., AND LOHMANN, D. From Global to Local Quiescence: Wait-Free Code Patching of Multi-Threaded Processes. In *Proceedings of the 14th USENIX Conference on Operating Systems Design and Implementation (OSDI '20)* (November 2020).

[63] RT-TESTS DEVELOPER TEAM. Cyclictest. https://wiki.linuxfoundation.org/realtime/documentation/howto/tools/cyclictest/start.

[64] SALTZER, J., AND KAASHOEK, M. F. *Principles of Computer System Design: an Introduction*. Morgan Kaufmann, 2009.

[65] SHE, S., LOTUFO, R., BERGER, T., WĄSOWSKI, A., AND CZARNECKI, K. Reverse engineering feature models. In *Proceedings of the 33rd International Conference on Software Engineering (ICSE '11)* (May 2011).

[66] SRIDHARAN, M., FINK, S. J., AND BODÍK, R. Thin Slicing. In *Proceedings of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation (PLDI'07)* (June 2007).

[67] SUSE. kgraft: Live patching of the Linux kernel. https://events.static.linuxfound.org/sites/events/files/slides/kGraft.pdf.

[68] TARTLER, R., DIETRICH, C., SINCERO, J., SCHRÖDER-PREIKSCHAT, W., AND LOHMANN, D. Static analysis of variability in system software: The 90,000 `#ifdefs` issue. In *Proceedings of the USENIX Annual Technical Conference (USENIX '14)* (June 2014).

[69] TARTLER, R., LOHMANN, D., SINCERO, J., AND SCHRÖDER-PREIKSCHAT, W. Feature consistency in compile-time–configurable system software: Facing the Linux 10,000 feature problem. In *Proceedings of the EuroSys Conference (EuroSys '11)* (April 2011).

[70] THAKKAR, V., SUKUMAR, M., DAI, J., SINGH, K., AND CAO, Z. Can modern LLMs tune and configure LSM-based key-value stores? In *16th ACM Workshop on Hot Topics in Storage and File Systems (HotStorage '24)* (July 2024).

[71] THE IPERF3 AUTHORS. iperf3: A TCP, UDP, and SCTP network bandwidth measurement tool. https://github.com/esnet/iperf.

[72] THE VULERT TEAM. Data race vulnerability in Linux kernel: sysctl access issue. https://vulert.com/vuln-db/debian-11-linux-289447.

[73] TORVALDS, L. Linux-2.6.12-rc2: Let it rip. https://github.com/torvalds/linux/commit/1da177e4c3f41524e886b7f1b8a0c1fc7321cac2, 2005.

[74] TUXCARE. KernelCare: Rebootless security patching with zero downtime. https://tuxcare.com/enterprise-live-patching-services/kernelcare-enterprise/.

[75] VALENTE, P., AND AXBOE, J. Linux kernel commit – block, bfq: reduce threshold for detecting command queueing. https://github.com/torvalds/linux/commit/a3c92560324bd616deaecb6842b2a0337a80ad8b.

[76] VALENTE, P., AND AXBOE, J. Linux kernel commit – block, bfq: reduce write overcharge. https://github.com/torvalds/linux/commit/d5801088a7bd210dd8fd7add04745e35f0f6ea72.

[77] VAN RIEL, R. Documentation for /proc/sys. https://www.kernel.org/doc/html/v6.14/admin-guide/sysctl/index.html, 1999.

[78] YELAM, A., WU, K., GUO, Z., YANG, S., SHASHIDHARA, R., XU, W., NOVAKOVIC, S., SNOEREN, A. C., AND KEETON, K. PageFlex: flexible and efficient user-space delegation of Linux paging policies with eBPF. In *Proceedings of the USENIX Annual Technical Conference (USENIX '24)* (Santa Clara, CA, July 2024).

[79] YING, H. Linux kernel commit – migrate_pages: restrict number of pages to migrate in batch. https://github.com/torvalds/linux/commit/42012e0436d44aeb2e68f11a28ddd0ad3f38b61f.

[80] ZHANG, H. tcp: Set pingpong threshold via sysctl. https://github.com/torvalds/linux/commit/562b1fdf061bff9394ccd884456ed1173c224fdc.

[81] ZHONG, Y., LI, H., WU, Y. J., ZARKADAS, I., TAO, J., MESTERHAZY, E., MAKRIS, M., YANG, J., TAI, A., STUTSMAN, R., AND CIDON, A. XRP: In-Kernel storage functions with eBPF. In *Proceedings of the 16th USENIX Conference on Operating Systems Design and Implementation (OSDI '22)* (July 2022).

[82] ZIJLSTRA, P., DICKINS, H., MORTON, A., AND TORVALDS, L. Linux kernel commit – mm: per device dirty threshold. https://github.com/torvalds/linux/commit/04fbfdc14e5f48463820d6b9807daa5e9c92c51f.

[83] ZUSSMAN, T., ZARKADAS, I., CARIN, J., CHENG, A., FRANKE, H., PFEFFERLE, J., AND CIDON, A. cache_ext: Customizing the page cache with eBPF. In *Proceedings of the 30th ACM Symposium on Operating Systems Principles (SOSP '25)* (October 2025).

A Sysctl Performance Knobs

We did a pilot study of `sysctl` knobs on Ubuntu 24.04 LTS which uses Linux v6.14. The number of `sysctl` knobs may vary slightly across distributions. We found 702 `sysctl` knobs in total (deduplicated due to NIC name prefix) and inspected all of them. Among them, 145 `sysctl` knobs are related to performance tuning. The others are for debugging, observability, and feature toggles.

We study the evolution of these 145 `sysctl` knobs, and summarize the results in Figure 24. We observe that the `sysctl` knobs evolve slowly. Among the 145 `sysctl` knobs, 96 have not changed since 2005 (earlier commit history is unavailable). Only 49 knobs were indeed evolved from constants—perf-consts studied in this paper. Among them, 19 (columns 1–19 in Figure 24) were historically converted from a perf-const. The other 30 were extended from system-wide values to per-namespace (cgroup) values. Reading the commit messages and the related discussions, it is clear that converting a perf-const into a `sysctl` knob is very slow due to the conservative upstream practice in Linux. For example, the TCP ping-pong threshold was set to 1 originally, raised to 3 in 2019, reverted to 1 in 2022, and only recently exposed as a `sysctl` in response to application demands (e.g., SQL workloads) in 2023 [80]. Therefore, waiting for a perf-const to be converted to a `sysctl` knob may be unwise.

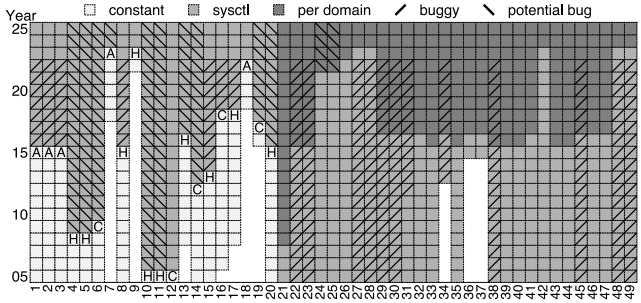


Figure 24: Evolution of `sysctl` knobs. Each box shows the status of a knob. constant: a fixed perf-const in the source code; `sysctl`: exposed as a `sysctl` knob. per-domain: the `sysctl` knob is made per namespace. buggy: time between the introduction of a bug and the commit that fixes it. potential bug: a likely bug identified by our inspection. A/H/C: reason for the change (A: application-driven, H: hardware-driven, C: exposing control of an inherent trade-off).

Note that converting a perf-const into a `sysctl` knob is non-trivial and error-prone. As shown in Figure 24, 20 knobs required bug fixes after their conversion, taking an average of 12 years to resolve. Most issues stem from race conditions introduced by the new global variable. We check the bug pattern of all the 149 `sysctl` knobs, and identify 43 additional knobs with likely unresolved bugs. Basically, converting a perf-const into a writable interface adds complexity: external writers (i.e., callers of the interface) can update the value while kernel threads may still read the old one. Fine-grained control adds further complexity. For example, changing the dirty-ratio knob to operate per device introduced a deadlock because the `sysctl` configuration path was not aware of the new semantics [82]. Xkernel addresses these problems by safely updating perf-const values through SIE (see §7).

B Optimization for Conditional Branches

When the immediate value of a perf-const is used in a conditional branch (e.g., Figure 4(a)), the location target of SIE is the corresponding `jmp` instruction. However, a conditional jump cannot be jump-optimized by Kprobe [32].

For such cases, we implement a new optimization that allows jump-optimized kprobes. The key idea is to modify R/M in the `cmp` in advance (at a position that jump-optimization is possible), and restore it along the control flow via extra jump-optimized kprobes, synchronized by a task-local flag, as shown in Figure 25. In this way, the same effect as synthesizing an SIE indirection to modify `eflags` can be achieved by proactively adjusting R/M used in the conditional jump. For example, in Figure 4(a), changing the immediate value from 7 to 4 is functionally equivalent to adding 3 to `eax` before the comparison. More generally, if we want to replace V with V', we can adjust the R/M operand by $\Delta V = V - V'$ in advance, and later restore its original value along all relevant control-flow paths. This optimization increases the fraction of jump-optimized kprobes from 66.6% to 88.3% in our evaluation.

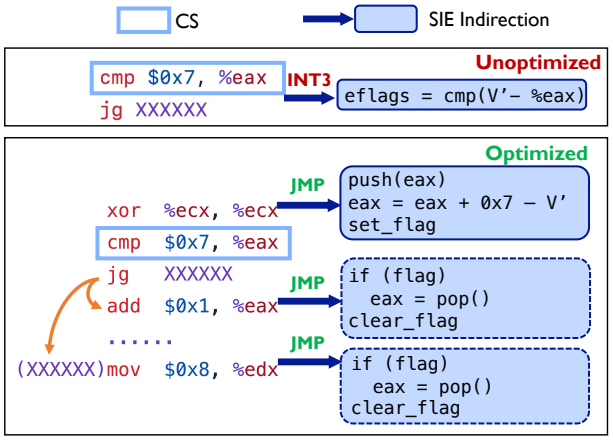


Figure 25: Optimization for conditional branches. The `cmp` instruction is undesired because its length is less than 5 bytes, which prevents Kprobe jump-optimization. Xkernel attempts to attach the probe to a preceding instruction to enable Kprobe jump-optimization.

To implement this optimization, Xkernel inserts two extra kprobes at the jump target and the subsequent instruction. These kprobes query the task-local storage and restore the operand. To ensure restoration occurs only when control flow originates from the CS, and not from unrelated execution paths, we employ a task-local flag. This flag is set upon entry to the first kprobe and cleared when the two extra kprobes trigger. If the first kprobe and either of these two kprobes fail to enable jump-optimization, we fall back to the

```
XK_TUNE(do_shrink_slab, "mm/shrinker.c:L381:128:0") {
    // 1. Safety Guard (Mandatory)
    if (!xk_transition_done(xk_ctx)) return 0;
    struct shrinker *s = (struct shrinker *)
        PT_REGS_PARM2(ctx);
    char name[32];
    if (bpf_probe_read_kernel(name, sizeof(name),
        &s->name) < 0) return 0;
    // 2. Tune SHRINK_BATCH only for zswap
    if (bpf_strncmp(name, 12, "zswap-shrink") == 0)
        xk_set(xk_ctx, 64);
    return 0;
} /* SHRINK_BATCH.bpf.c */
```

Figure 26: An Xk-tune for zswap shrinker; it identifies zswap shrinkers by checking the name.

```
#define MAX_ROCKSDB_THREADS 16
struct { // Filled by RocksDB threads
    __uint(type, BPF_MAP_TYPE_HASH);
    __uint(max_entries, MAX_ROCKSDB_THREADS);
    __type(key, u32);
    __type(value, int);
} hint_map SEC(".maps");
XK_TUNE(blk_add_rq_to_plug, "block/blk.h:L312:32:0") {
    // 1. Safety Guard (Mandatory)
    if (!xk_transition_done(xk_ctx)) return 0;
    // Get the pid upon invocating this kernel code path
    u64 pid = bpf_get_current_pid_tgid() & 0xFFFFFFFF;
    // 2. Leverage hints from the application
    int *hint = bpf_map_lookup_elem(&hint_map, &pid);
    if (hint) xk_set(xk_ctx, 1);
    return 0;
} /* BLK_MAX_REQUEST_COUNT.bpf.c */
```

Figure 27: An application-informed policy for RocksDB. RocksDB characterizes threads with random-read patterns and exposes their IDs through a BPF map. Xk-tune leverages this hint to enable per-thread tuning.

original SIE location. When successful, this optimization replaces one INT3-based kprobe with two jump-optimized kprobes. Despite introducing extra kprobes, this approach is beneficial: a jump-optimized kprobe is an order of magnitude cheaper than an INT3-based kprobe.

C Kprobe Limitation

In our evaluation (§6), Xkernel failed to handle one perf-const, where the target kernel functions could not be attached. The perf-const, `SEND_MAX_EXTENT_REFS`, is located in `fs/btrfs/send.c` within the `check_extent_items()` function. In Linux v6.14, the symbol `check_extent_item()` has multiple definitions (e.g., appearing simultaneously in the kernel core and a loadable module). When registering a kprobe using the syntax `[MOD:]SYM[+offs]`, the kernel's `trace_kprobe` mechanism resolves the symbol name. However, to prevent ambiguity, the kernel rejects the registration if multiple matches are found, causing the kprobe creation to fail with `EINVAL` or `-EADDRNOTAVAIL`.

D Policy Code with Xkernel

We present three Xk-tune programs that illustrate the policies enabled by Xkernel (§5.2): object-level tuning granularity, application-informed policy, and ad-hoc heuristics. These examples show how users express such policies in Xkernel.

Figure 26 shows an Xk-tune program that customizes perf-consts for different objects. Written in eBPF, Xk-tunes can incorporate many features to customize tuning granularity. The perf-const is `SHRINK_BATCH` (Case-3 in §5.1). Figure 27 shows an Xk-tune program for an application-informed policy for a RocksDB application. The perf-const is `BLK_MAX_REQUEST_COUNT` (Case-1 in §5.1). Figure 28 shows an Xk-tune program that coordinates with other kprobes and uses ad-hoc workload heuristics. The perf-const is `BLK_MAX_REQUEST_COUNT` (Case-1 in §5.1).

```
#define MERGE_FAIL_THRESHOLD 16
struct { // Filled by blk_attempt_plug_merge()
    __uint(type, BPF_MAP_TYPE_TASK_STORAGE);
    __uint(map_flags, BPF_F_NO_PREALLOC);
    __type(key, int);
    __type(value, int);
} hint_map SEC(".maps");
/* Track consecutive failed attempts to merge adjacent
   blocks and store the count in fail_cnt for use as a
   tuning heuristic */
SEC("kretprobe/blk_attempt_plug_merge")
int BPF_KRETPROBE(blk_attempt_plug_merge_ret, long ret) {
    struct task_struct *task = bpf_get_current_task_btf();
    int *fail_cnt = bpf_task_storage_get(&hint_map,
        task, NULL, BPF_LOCAL_STORAGE_GET_F_CREATE);
    if (!fail_cnt) return 0;

    if (ret == 0) (*fail_cnt)++;
    else *fail_cnt = 0;

    return 0;
}
XK_TUNE(blk_add_rq_to_plug, "block/blk.h:L312:32:0") {
    // 1. Safety guard (mandatory)
    if (!xk_transition_done(xk_ctx)) return 0;
    struct task_struct *task = bpf_get_current_task_btf();
    // 2. Leverage hints from blk_attempt_plug_merge()
    // Heuristic: treat the workload as random when many
    // (16, an ad-hoc threshold) consecutive merges fail
    int *fail_cnt = bpf_task_storage_get(&hint_map,
        task, NULL, 0);
    if (fail_cnt && *fail_cnt >= MERGE_FAIL_THRESHOLD)
        xk_set(xk_ctx, 1);
    return 0;
} /* BLK_MAX_REQUEST_COUNT.bpf.c */
```

Figure 28: An Xk-tune program using ad-hoc workload heuristics for RocksDB. The Xk-tune instruments `blk_attempt_plug_merge` with a kretprobe to track historical merge failures. Upon detecting a high failure rate, the Xk-tune adjusts the threshold to a smaller value (1).

E Xkernel Tool Commands

We show a few commands that use Xkernel tools (§4).

```
xk-build /usr/src/linux tune.patch # > ConstID
xk-gen ConstID xk-stub.h
xk-load [global/task/imm] xk-tune.c
xk-unload ConstID/all
```

Murakkab: Resource-Efficient Agentic Workflow Orchestration in Cloud Platforms

Gohar Irfan Chaudhry Esha Choukse[†] Haoran Qiu[†] Íñigo Goiri[†]
Rodrigo Fonseca[†] Adam Belay Ricardo Bianchini[‡]
MIT CSAIL [†]*Microsoft Azure Research* [‡]*Microsoft Azure*

Abstract

Agentic workflows commonly coordinate multiple models and tools with complex control logic. They are quickly becoming the dominant paradigm for AI applications. However, serving them remains inefficient with today's frameworks. The key problem is that they expose workflows as opaque sequences of model and tool calls that tightly couple agent logic with model and hardware choices. This prevents systems from holistically reasoning about trade-offs across accuracy, latency, energy, and cost. This leads to resource waste and degraded service-level objectives (SLOs).

We present Murakkab, a resource-efficient serving system for agentic workflows. Murakkab introduces a declarative abstraction that decouples workflow specification from execution configuration. A profile-guided optimizer and adaptive runtime jointly manage the full stack: orchestrating workflow components, mapping them to models and hardware, and dynamically reconfiguring execution to satisfy user-defined SLOs. By exposing the internal structure of agentic workflows, Murakkab enables cross-layer optimization that existing frameworks and cloud schedulers cannot achieve.

Our evaluation on diverse workflows shows that, compared to state-of-the-art systems, Murakkab reduces GPU usage by up to 2.8×, energy consumption by 3.7×, and cost by 4.3× while maintaining SLOs.

1 Introduction

The rise of complex, multi-step tasks in domains like scientific discovery, education, and health analytics has driven the emergence of *agentic workflows* [16, 22, 35, 41]. These workflows integrate large language models (LLMs) and multimodal variants with external, specialized tools such as search engines, code interpreters, and databases, allowing them to reason and act dynamically [71, 82, 88]. Modern cloud platforms [7, 8, 19] (*e.g.,* Azure AI Foundry) are rapidly evolving to support these agentic workflows, providing a unified platform to develop, orchestrate, and run them on managed model deployments and compute infrastructure.

Motivation. Despite this foundational infrastructure, deploying and operating complex agentic workflows at scale is fundamentally challenging. Today's systems follow an *imperative*, *siloed* paradigm where developers manually configure and compose a sequence of agents and tools (*e.g.,* using frameworks like LangGraph [46] or LlamaIndex [53]), while the cloud providers handle model deployments (with autoscaling) that treat LLM inference requests from the agents no different from standard LLM API calls (*e.g.,* ChatGPT). This approach suffers from a critical architectural limitation where optimization is performed in strict silos:

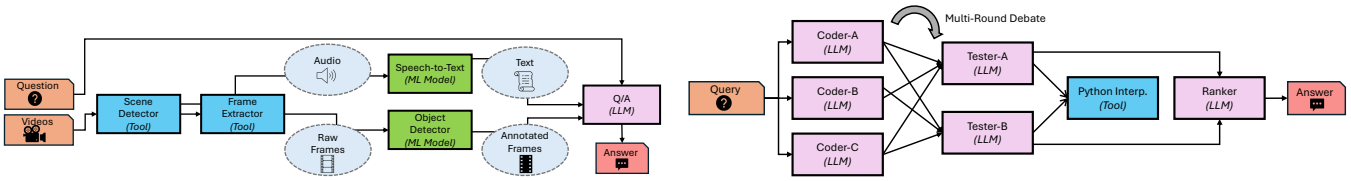
- *Workflow-level*: What specific agents to include and how to structure the workflow topology.
- *Agent-level*: Tuning specific parameters (*e.g.,* what model to use per-agent, or what frame sampling frequency to set for video analysis etc.).
- *Hardware-level*: Resource provisioning (*e.g.,* accelerator type and quantity, parallelism configuration etc.).

Since these layers are optimized independently, the system cannot reason about global, *end-to-end objectives* such as minimizing cost while meeting a per-query accuracy SLO or maximizing throughput under a latency constraint. Consequently, deployed workflows are rigid, often suffer from poor resource utilization, and require laborious manual reconfiguration across all layers to handle any necessary adaptation (*e.g.,* a sudden increase in traffic or a model update).

This inefficiency gap points to a missing architectural principle in current cloud platforms: unified, end-to-end orchestration that spans the agentic logic, the model serving layer, and the underlying infrastructure. We propose a paradigm shift, moving from the current siloed deployment to a *declarative*, *holistic* optimization approach, closing the gap for running agentic workflows in cloud platforms like Azure AI Foundry [8] or Google Vertex AI [19]. The core idea is to introduce a control plane that treats the entire agentic workflow: the sequence of LLM calls, tool executions, and data dependencies, as a single, optimizable computational graph running on a cloud platform's model deployments.

Our Work. We present Murakkab[1] [12], a resource-efficient system for serving multi-tenant agentic workflows, built on two principles: (1) declarative workflow specification and, (2) adaptive, SLO-aware runtime system design.

[1] An Urdu/Persian word (مُرَکَّب) meaning "compound" (referring in this context to agentic workflows composed of many pieces).



(a) Video Q/A workflow [87]: multi-modal with tools handling different modalities before feeding into an LLM for answering the query.

(b) Code generation workflow [25, 55]: text-only with an LLM Debate structure to write, test and execute code.

Figure 1: Two agentic workflows with different characteristics and components.

Murakkab lets developers focus on application logic–similar to serverless–while the runtime manages workflow orchestration, model selection, and resource provisioning to meet SLOs efficiently. This is critical in emerging environments where a single entity manages both models/agents and infrastructure. By unifying workflow configuration, model selection, and resource management, Murakkab achieves higher efficiency than today's fragmented systems, where layers operate in isolation.

Developers describe workflows with a declarative specification as *logical* tasks and dependencies, decoupled from model, tool, or hardware choices. This separation enables Murakkab to integrate workflow orchestration with resource management, dynamically reconfiguring workflow parameters (*e.g.,* models or tools) and hardware configurations as needed, based on offline profiles and online monitoring to optimize quality, latency, energy, and cost objectives.

This design enables Murakkab to efficiently explore the configuration space of workflow-, agent- and hardware-level knobs while meeting user-defined SLOs. Unlike existing deployments that treat agentic workflows as opaque, Murakkab leverages end-to-end visibility to coordinate scheduling, colocation, and multiplexing across multi-tenant workflows.

We evaluate Murakkab on representative workflows (video question answering, code generation and math problem solving) using production-scale traces. Murakkab achieves up to 2.8× lower GPU usage, 3.7× less energy, and 4.3× lower cost than LangGraph, a state-of-the-art agentic workflow orchestration, while preserving quality and latency SLOs.

Summary. This paper makes the following contributions:

- A declarative programming model and adaptive runtime unifying orchestration with resource management.
- Profile-guided optimization and workflow-aware scheduling for multi-tenant workloads.
- Evaluation showing significant efficiency gains without workflow quality or execution latency SLO violations.

2 Background and Motivation

2.1 Agentic Workflows

We define an ***agent*** as a composable unit that autonomously makes decisions, invokes tools, and participates in workflows, either independently or in collaboration with other agents [11, 20]. Each agent is typically powered by: (1) a *model* (*e.g.,* an LLM) for reasoning and language understanding, (2) a set of *instructions* specifying its goals, behavior, and constraints, and (3) *tools* for retrieving knowledge or taking actions. When multiple agents (each with specialized roles) interact and collaborate to achieve complex goals, we call the resulting process an ***agentic workflow***.

Serving agentic workflows differs fundamentally from single-model inference. Such workflows require coordinated interaction and data exchange among multiple agents to produce a final response. These agents often span diverse use cases, modalities (*e.g.,* text, audio, images etc.), execution times, and software/hardware requirements. Composing effective workflows, thus, demands a nuanced understanding of the trade-offs between task accuracy, request-serving latency, energy consumption, and hardware cost [43].

2.2 Example Workflows

We consider three widely used, representative workflows [66]: *Video Q/A*, *Code Generation*, and *Math Q/A* which exhibit distinct characteristics. For the rest of the paper, we focus on the first two, as shown in Figure 1, and include evaluation of the third one in Section A.1. A wide range of other workflows can be orchestrated using a similar approach.

Video Q/A. Video Q/A workflows are deployed for interactive query processing [74], security footage analysis [69], and many other use-cases [75, 80]. We construct a comprehensive agentic workflow to capture these diverse use-cases [2, 87]. This multi-modal workflow answers textual queries about input videos (Figure 1a). Several agents in this workflow collaborate to produce the final result:

1. *Scene Detector* to separate the input videos into distinct scenes that ensure better capture of key-frames and extract the audio clip corresponding to each scene.
2. *Frame Extractor* to sample frames from each scene.
3. *Speech-to-Text* for audio transcription.
4. *Object Detector* to annotate frames with objects of interest that are related to the query.
5. *Multi-modal LLM* (or LMM) to answer the user query given the processed frames and audio transcript.

The resulting agentic workflow can be modeled as a di-

rected acyclic graph (DAG), where nodes represent individual agents (*e.g.,* Speech-to-Text transcription) and edges capture data flow and execution dependencies among agents (*e.g.,* passing frames from Frame Extractor to Object Detector).

Code Generation. An important use case for agentic workflows is coding and software engineering [4, 28, 33, 58] or generating code on-the-fly for accomplishing tasks in a workflow [83]. We present a representative text-only workflow that translates natural language descriptions into executable Python code (Figure 1b). It adopts the *LLM Debate* framework [25, 55], where coder agents propose candidate solutions and tester agents generate tests and execute them using a Python interpreter. Each agent plays a unique role (*e.g.,* algorithm developer, unit tester) and may employ the same or different LLMs. The agents engage in iterative rounds of debate, aiming to reach consensus or terminating after a predefined number of rounds. The final output is selected as the highest-voted solution, determined by an LLM based on both the proposed candidates and the original query.

Dynamic Coding Pipeline. A complementary class of coding workflows [4, 28] constructs the execution workflow *dynamically* for each request, rather than relying on a fixed multi-agent structure. We build a representative pipeline inspired by production code assistants (*e.g.,* Claude Code [4]) that translates natural-language tasks into executable Python programs while adapting its execution structure to each request (problem prompt). The pipeline comprises three LLM roles: (1) a *writer* that generates candidate solutions, (2) an optional *reviewer* that critiques candidates using test outcomes, and (3) an optional *orchestrator* that selects the writer specialization and compute budget per problem. Writers can adopt specialized roles such as *algorithmic* or *I/O-oriented*, with reasoning efforts of *detailed* or *concise*, each emphasizing a different solution strategy tailed to the given problem prompt.

Each outer iteration launches a configurable number of parallel writer calls whose outputs are ranked against the problem's public tests, after which the top-performing candidate is selected. If all public tests pass and the reviewer is disabled, execution terminates early. Otherwise, failing test traces (and optionally reviewer feedback) are fed back as structured guidance for the next iteration. The reviewer may run on every iteration or only as a fallback when tests fail, while still preserving early termination.

Unlike the static code-generation workflow, the execution structure of this pipeline is determined on a per-request basis. In addition, each sub-workflow exposes configurable knobs such as number of review rounds, writer candidates, iteration budget, reviewer usage, and model assignment. Together, these decisions jointly define a large space of possible execution strategies, making the workflow a challenging testbed to navigate across heterogeneous operating points with diverse performance-cost tradeoffs.

```python
# == Workflow nodes (coupled app logic + configs) ==
scene_detection = Tool(
    fn=SceneDetector(),    # Implemented earlier by the
      developer.
    key=AWS_KEY, resources={"CPUs": 32}
)
frame_extractor = Tool(
    fn=FrameExtractor(),   # Implemented earlier by the
      developer.
    key=AWS_KEY, params={"num_frames": 15}, resources={"CPUs": 32}
)
speech_to_text = MLModel(
    name="Whisper", key=OPENAI_API_KEY, resources={"PTUs": 50}
)
object_detection = MLModel(
    name="CLIP", key=AZURE_KEY, resources={"CPUs": 128}
)
question_answer = LLM(
    name="Llama-3.2", key=DATABRICKS_API_KEY,
    params={"batch": 256}, resources={"GPUs": 8, "Type": "H100"},
    system_prompt="You are an agent that can understand videos.",
    user_prompt="Answer the given question about the video."
)
# == Query and data ==
ques    = "What is the name of the person wearing the red dress?"
videos  = ["road_trip.mp4"]
# == Workflow (ordering of nodes and data flow) ==
scenes, audio  = scene_detection(videos)
frames         = frame_extractor(scenes)
transcript     = speech_to_text(audio)
obj_frames     = object_detection(frames)
answer         = question_answer(ques, transcript, obj_frames)
```

Listing 1: Simplified video Q/A workflow definition. It shows tightly coupled application logic and execution details typical of existing frameworks.

2.3 Rigid and Imperative Definitions

Current state-of-the-practice for agentic workflow development are frameworks such as LangGraph [46], LangChain [45], LlamaIndex [53], AutoGen [6] and CrewAI [21]. They follow an ***imperative*** paradigm: developers define workflow components (*i.e.,* agents), their execution logic, interactions, and numerous ***configuration parameters***.

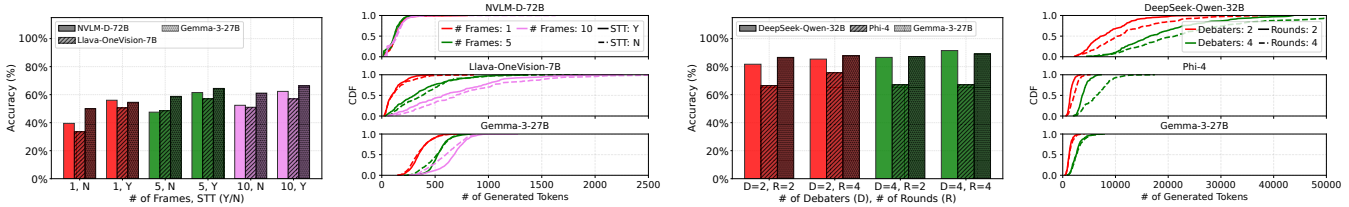
For example, in a video Q/A workflow (Listing 1), the developer specifies parameters for `FrameExtractor` (number of frames per scene), enables options like speech-to-text or object detection, and defines the ***execution mechanism***, or model, for each component (*e.g.,* `SceneDetector` in OpenCV, Whisper [65] for transcription, Llama-3.2 [36] for reasoning).

Developers must also make ***resource allocation*** decisions, such as assigning CPUs/GPUs for self-hosted or rented VMs, or choosing service tiers (*e.g.,* provisioned throughput units or PTU [10]) for managed services.

This tightly couples configuration choices with the workflow DAG, making deployed workflows *rigid*. Any change requires manual updates and redeployment across the stack.

2.4 Fragmented Stack and Goals

Agent developers create and deploy workflows that expose endpoints callable by end users. Entities involved in developing, using, and executing agentic workflow (*i.e.,* developers, end users, and providers) prioritize different goals. *Developers*



(a) Video Q/A accuracy. Varying frames, enable STT, and LMM.

(b) Video Q/A tokens generated by varying the knobs.

(c) Code Gen. accuracy. Varying debaters, rounds, and LLM.

(d) Code Gen. tokens generated by varying the knobs.

Figure 2: Workflow accuracy under different configurations and the token generation load on the respective models.

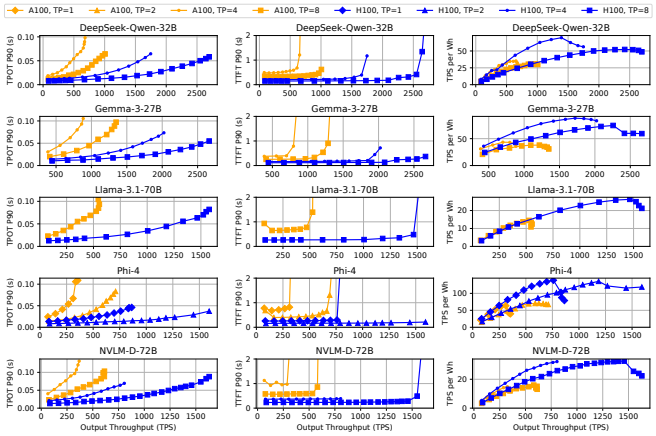


Figure 3: Model performance under different hardware and parallelism configurations.

value response quality, latency, or both. *End-users* additionally care about execution cost. *Providers* prioritize resource efficiency and cost. However, the decision-making burden for workflow configuration today largely falls on developers, who must navigate an overwhelming design space and deployment knobs without holistic system visibility. Ultimately, these decisions impact all involved entities in the agentic workflow life-cycle. These configurations include:

- ***Workflow-level configurations***: *e.g.,* whether to include a Speech-to-Text Transcript agent.
- ***Agent-level configurations***: *e.g.,* how many frames to extract in Frame Extractor, which LLM to use for Q/A.
- ***Hardware-level configurations***: *e.g.,* CPU *vs.* GPU and parallelism degree for each model.

2.5 Characterization and Motivation

Poor Resource Efficiency. Frameworks for agentic workflows (*e.g.,* LangGraph [46] and LlamaIndex [53]) place the burden of workflow configuration on developers. Most developers are neither systems nor ML experts, and cannot reason about accelerator choice, model parallelism, or model selection. Configurations are therefore often arbitrary and inefficient, leading to poor performance and high cost. Even experienced developers, lacking insight into user priorities, default to maximizing accuracy at the expense of efficiency.

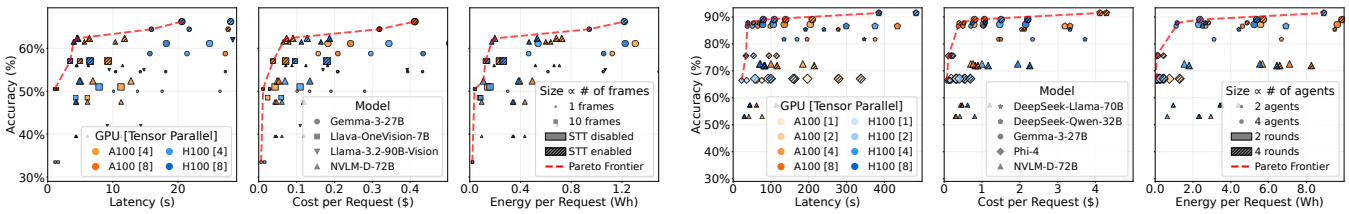
From the cloud provider's perspective, these workflows are opaque. The cloud platform has little visibility into workflow components (*e.g.,* models) or interactions (*e.g.,* task sequences and data flow). Tightly coupled application logic and execution details prevent the platform from reconfiguring workflows to improve resource efficiency while meeting SLOs on accuracy or latency. The result is resource underutilization, increased costs, and a larger energy footprint (often passed on to users through higher prices).

> **Insight 1:** *Cloud platforms lack visibility into workflow internals* (e.g., *tasks, requirements), preventing end-to-end optimization. Meanwhile, developers lack control or insight into system-level resource behavior.*

Multi-level Trade-offs. Configurations spanning across workflow level, agent level, and hardware layer introduce complex trade-offs over accuracy, latency, energy, and cost.

Workflow- and Agent-level knobs. Figure 2 shows a subset of configurations for the video Q/A and code generation workflows and their response accuracy. For video Q/A, Figure 2a shows that both workflow-level knobs (*e.g.,* whether to enable speech-to-text) and agent-level knobs (*e.g.,* number of frames extracted) significantly affect accuracy. Figure 2b shows the tokens generated for each configuration. Higher token generation is associated with higher latency, cost, and GPU load. For example, using Gemma-3-27B [81] with 10 frames and STT enabled achieves the highest video Q/A accuracy (66.2%) but also generates the most tokens compared to other configurations of the same model. Token counts vary widely across requests, from 250 to nearly 1000, often with a heavy tail.

For the code generation workflow, accuracy is sensitive to the number of debaters and rounds in LLM Debate (Figure 2c). Some configurations, such as DeepSeek-Qwen-32B [24] (a reasoning model), achieve the highest accuracy but at a much higher token generation cost (Figure 2d). At the median, it generates ≈20,000 tokens versus ≈2,500 for Gemma-3-27B under the same workflow configuration. Yet token counts alone do not capture the full end-to-end workflow characteristics, such as latency, cost, or energy.



(a) Video Q/A workflow.

(b) Code generation workflow.

Figure 4: Large space of workflow configurations along a subset of knobs and metrics.

Hardware-level knobs. Different models in a workflow have varying compute/memory requirements. Their interaction with the hardware determines whether a workflow can meet user SLOs while satisfying cloud platform constraints, like the number of GPUs allocated. Figure 3 shows models running on different hardware configurations (*e.g.,* GPU types and parallelism). We evaluate time to first token (TTFT) and time per output token (TPOT) under varying load, and measure energy efficiency as throughput (in tokens per second or TPS) per Wh. The results reveal a rich trade-off space, where configurations offer different balances of throughput, latency, and energy efficiency.

Insight 2: *Configuration knobs introduce fundamental trade-offs: workflow- and agent-level knobs affect accuracy* vs. *latency/cost, while additionally, hardware-level knobs drive cost* vs. *performance trade-offs, making it challenging to optimize workflows across multiple objectives.*

High-Dimensional Configuration Space. The configuration space for the end-to-end agentic workflow can grow exponentially with the configuration choices of the individual agents that make up the workflow. Figures 4a and 4b show that even modest agentic workflows like video Q/A or code generation, with only a small set of knobs, produce explosive configuration spaces to choose from. Each point in these Pareto frontiers represents a valid end-to-end configuration derived from varying model selection, hardware accelerator assignments, and resource scaling (with model parallelism) strategies. Navigating this configuration space manually is infeasible given that: (1) objectives vary across users (*e.g.,* low-latency *vs.* low-cost *vs.* high accuracy), (2) the optimal configuration depends on dynamic runtime context (*e.g.,* user traffic, model availability, cloud resource availability), and (3) model and hardware choices are rapidly evolving.

Insight 3: *Configuration complexity of each workflow grows combinatorially with the number of agents and their parameters at each level: ($O(\#WorkflowKnobs \times \#AgentKnobs \times \#HardwareKnobs)$).*

3 Murakkab Design

In light of this characterization, we propose Murakkab [12], a system that: (1) has visibility into workflow requirements, (2) holistically controls the entire workflow development and execution stack, and (3) enables automated configuration.

Instead of rigid, imperative workflows hand-tuned by developers, Murakkab supports *declarative* specifications that *decouple* high-level intent from low-level configuration. Developers focus on application logic, while the system *dynamically configures* workflows to adapt to evolving runtime conditions. *Developers* define workflow structure and task dependencies, *end-users* specify SLOs (*e.g.,* quality, latency, cost), and the *platform* dynamically optimizes execution for resource efficiency (*e.g.,* energy, cost) in a multi-tenant environment. Thus, workflows can automatically adapt to new models, varying load, or changing resources (without code rewrites or redeployments).

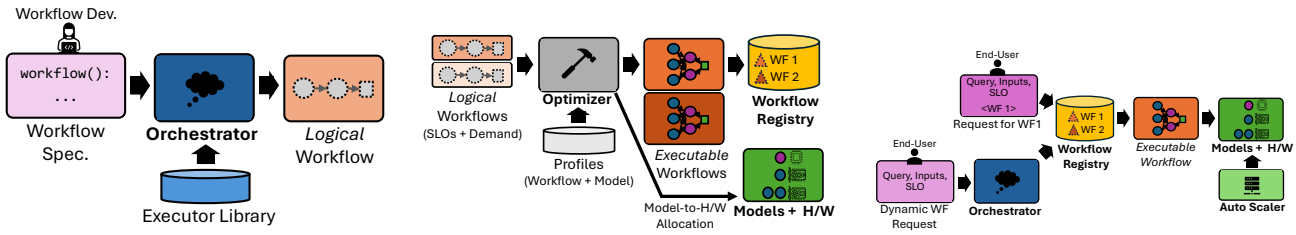
3.1 Workflow Life-Cycle

Murakkab cohesively manages the end-to-end life-cycle, unlike existing systems that fragment these among different entities. There are three main phases in an agentic workflow's life-cycle, as shown in Figure 5: (1) Workflow Development (Figure 5a), (2) Workflow Optimization (Figure 5b), and (3) Workflow Execution (Figure 5c). We summarize these in Table 1 and cover each of them in more detail below.

3.2 Development

Murakkab adopts a declarative paradigm for workflow specification that decouples *application logic* from low-level *execution details*. Workflow developers may specify high-level tasks, and optionally, the data flow between them, without the need to provide any workflow configurations (*e.g.,* resource allocation or model selection details).

Executor Library. Murakkab is designed to interoperate with existing model and tool ecosystems. It supports LLMs and traditional machine learning (ML) models from repositories such as Hugging Face [26], as well as tools from open-source libraries and platforms, including OpenAI Agents SDK [67], Google Vertex AI Agent Garden [34], NVIDIA NeMo Agent Toolkits [61], and Microsoft Azure AI Foundry Tools [9]. In Murakkab, an *executor* is a functional unit within a workflow



(a) Development phase. (b) Optimization and deployment phase. (c) Execution phase.

Figure 5: Murakkab manages end-to-end workflow life-cycle: from development to optimized deployment and execution.

Decision	Frequency	Scope	Notes
Phase 1: Workflow Development (Orchestrator)			
Workflow DAG structure	Once at onboarding	Workflow	Per-query for dynamic requests
Executor assignment per DAG node	Once at onboarding	Workflow	Refined per SLO tier by optimizer
Phase 2: Workflow Optimization (Optimizer)			
Final model/tool per executor	Per epoch	(Workflow, SLO)	
GPU type and parallelism	Per epoch	(Workflow, SLO)	Implicit via model-profile choice
Workflow-level knobs	Per epoch	(Workflow, SLO)	
Per-model instance count (n_m)	Per epoch	Model	Sized for projected peak load
Phase 3: Workflow Execution			
Reactive scale-out / scale-in	Continuous	Model	Auto-Scaler
Batch composition	Continuous	Instance	
Per-request routing and dispatch	Per request	Request	

Table 1: Decisions made by Murakkab across the workflow life-cycle.

```
# == Sub-tasks in the workflow ==
scene_detect   = "Given a list of videos, identify scenes in each."
frame_extract  = "Given a list of scenes, extract frames."
stt            = "Given a list of scenes, convert audio to text."
q_a            = "Answer the query given some context."
# == Workflow description (sub-tasks and data flow) ==
def workflow(query, videos):
    scenes      = scene_detect(videos)
    frames      = frame_extract(scenes)
    transcript  = stt(scenes)
    answer      = q_a(query, [frames, transcript])
    return answer
# == Execution with example request ==
query   = "What is the name of the person wearing the red dress?"
videos  = ["road_trip.mp4"]
result  = run(workflow(query, videos), slo=LOW_LATENCY)
```

Listing 2: Murakkab's declarative workflow specification of the video Q/A abstracts away configuration details, letting developers focus on application logic.

that can take one of three forms:

1. ***LLM***: specialized LLM configurations (fine-tuning, few-shot learning, or even just domain-specific prompting);
2. ***Structured compositions***: aggregations of models, *e.g.,* a self-reflection [73] or an LLM-Debate pattern [55] built from multiple LLMs;
3. ***Tool***: utility modules for AI workflows to take actions with (*e.g.,* OpenCV frame extractor for video processing, web-search, file-search, computer-use, or any third-party tools that follow the MCP specification [70]). Traditional ML models are also included as tools (*e.g.,* CNN-based image classifiers or Word2Vec sentiment analyzers).

Attributes. Each model or tool in the library exposes three attributes: (1) a textual description, (2) an interface specification, and (3) a key-value list of configurable parameters. For example, the frame extraction tool exposes the knobs: `F` (number of frames to extract) and `cores` (number of CPU cores to run on). The LLM Debate composition exposes the knobs: `D` (number of debaters), `R` (number of rounds), and `model` (which LLM to use). The *orchestrator* uses these descriptions and interfaces to rank and assign executors (models or tools) for workflow tasks, deferring parameter configuration to a later optimization phase (Section 3.3).

A key design choice in Murakkab is mapping a broad range of unknown tasks to executors that are built from a finite, known set of models and tools in the library. If none is found, Murakkab prompts the developer to onboard a suitable one.

Workflow Specification. We allow tasks to be expressed in *natural language* with varying levels of control over how the task may be executed, in contrast to imperative paradigms dis-

cussed earlier. Recent agentic-workflow development frameworks (*e.g.,* DSPy [77]) are moving in this direction. Murakkab takes this a step further and completely decouples configuration details from the workflow specification.

Declarative Specification. Developers can manually decompose tasks into sub-tasks and define data flow between them, providing precise control over workflow steps. Listing 2 shows the developer specifies sub-tasks (*e.g.,* scene detection, speech-to-text transcription, frame extraction) and their dependencies, ensuring sequential execution to achieve the desired outcome, such as answering a question about a video. Configuration details (*e.g.,* which LLM to use, number of frames to extract, resource allocation) are omitted from the specification. However, Murakkab does not restrict developers from specifying any execution preferences (*e.g.,* particular LLM choice or hardware constraint), which are then incorporated into the optimization process as constraints.

Interface. Once deployed, workflows are exposed to end users via dedicated REST endpoints. Each request may include an SLO, such as accuracy, latency, or cost tier. Murakkab uses these SLOs to configure workflows per request and optimize resource efficiency across tasks.

Workflow Orchestrator. The *workflow orchestrator* transforms a *declarative workflow* specification into a *logical workflow*. It interprets the specification, parses tasks and sub-tasks, and maps each to an appropriate executor from Murakkab's library. At the core is an LLM with tool-calling capabilities [71], which receives a list of available executors and their interfaces, along with task descriptions, and selects the best executor for each sub-task. Recent work [48, 59, 86] has focused on automating the discovery and refinement of effective agentic workflows to maximize task performance and minimize human intervention, which Murakkab can benefit from. A feedback loop allows developers to inspect and refine the generated specification, supporting hybrid workflows with both manual and system-generated tasks.

Logical Workflow. This abstract execution plan captures the functional intent of each task without binding to specific models, resources, or hardware. It is represented as a directed acyclic graph (DAG), where nodes are executors and edges denote data flow. This representation remains request-agnostic, containing no per-request details such as query text, input payloads, or SLOs. Execution specifics (*e.g.,* model selection or hardware allocation) are deferred to later stages.

The orchestrator performs type-checking on the DAG to ensure output types from source nodes match input types of destination nodes. In case of mismatches, the workflow is regenerated with error feedback to the LLM. Persistent errors prompt the developer to revise the specification.

3.3 Optimization and Deployment

Profiles. Accurate, fine-grained performance characterization is essential for optimizing multi-tenant agentic workflows with dynamic execution patterns. Inspired by Profile-Guided Optimization (PGO) [60, 84], Murakkab builds offline profiles across diverse configurations to inform runtime decisions. Each profile captures three key metrics per workflow configuration: response quality, end-to-end latency, and resource usage. To support this, Murakkab maintains two profiling layers: workflow profiles and model profiles.

Workflow Profiles. Murakkab profiles representative agentic workflows (*e.g.,* video Q/A) under different workflow-level knobs (*e.g.,* enabling/disabling STT) and executor-level knobs (*e.g.,* varying the number of frames extracted by the extractor tool). *Quality* is assessed using open-source benchmarks and datasets (*e.g.,* VideoMME [30], HumanEval [15], and Math [39]) with ground-truth results. When public datasets fall short, Murakkab can onboard curated evaluation datasets from developers or collect request-response pairs and user feedback (*e.g.,* rating, approvals), similar to deployed systems like ChatGPT [64]. These inputs are periodically incorporated into profile updates to refine configuration choices for active workflows. Recent studies of production agents show that many deployed workflows are largely static [68], making targeted profiling practical and amortizing the profiling cost across many subsequent invocations for the same workflow.

Configuration choices also affect per-request resource demands. Workflow profiles quantify executor-level load, including prompt and completion tokens for LLM-based executors, serving as a proxy for resource usage. These profiles capture metrics similar to those in our earlier characterization (*e.g.,* Figures 2a and 2b).

Model Profiles. To assess workflow resource demands, executor-level load must be mapped to model and hardware metrics. Murakkab maintains profiles for each model, capturing performance across software configurations (*e.g.,* tensor parallelism, prefill/decode separation) and hardware setups (*e.g.,* GPU type, clock frequency). Each profile reports: (1) latency (TTFT and TPOT for LLMs), (2) energy consumption across hardware, and (3) cost per configuration. Profiles span load levels to expose trade-offs and guide the optimizer in allocating load and instances.

Profiling is lightweight; performed once per configuration and reused across workflows. New models and accelerators are profiled upon integration. Decoupling workflow and model profiles enables workflows to benefit immediately from model updates, with selective re-profiling as needed.

This separation enables independent evolution, profile reuse, rapid model onboarding, and efficient hardware adaptation. The optimizer uses profiles as structured priors to evaluate cost–accuracy–latency trade-offs and drive reconfiguration under dynamic conditions.

Workflow Optimizer. The optimizer transforms the *logical workflow* from the orchestrator into an *executable workflow* by jointly selecting workflow knobs, models and tools per executor, and hardware and parallelism strategies. It com-

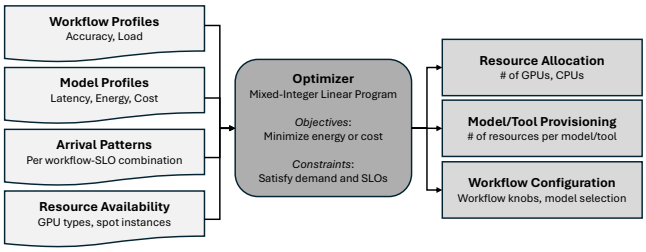


Figure 6: Summary of Murakkab's optimization process.

bines workflow profiles (task accuracy and executor-level load) with model profiles (latency, energy, and cost under varying load) to evaluate candidate configurations, and chooses those that meet per-request SLOs while maximizing efficiency. Beyond individual workflows, the optimizer uses global visibility across all active (workflow, SLO)-pairs to colocate and multiplex executors during each *optimization epoch*, balancing peak and average load. Figure 6 summarizes the inputs, process, and outputs.

3.3.1 MILP Formulation

We formulate the optimizer as a Mixed-Integer Linear Program (MILP). The complete formulation is presented in Section A.5; here we highlight the key components.

Inputs. The MILP takes four classes of inputs:

1. Workflow profiles for each registered logical workflow, capturing per-configuration accuracy and the distribution of executor-level load (e.g., prompt and completion token counts per LLM call) under each setting of the workflow-level knobs.

2. Model profiles for each candidate (model, hardware, parallelism) tuple, reporting throughput, TTFT and TPOT under varying load, energy per token, and dollar cost per GPU-hour.

3. Arrival patterns giving projected request rates for each (workflow, SLO)-pair over the next epoch, derived from recent trends (Section 3.4).

4. Resource constraints specifying the budget of available GPUs by type (including spot capacity), and any per-tenant cost ceilings.

Decisions. For each optimization epoch, the MILP jointly decides the following, which together form the deployment plan written to the workflow registry:

1. Workflow configuration: the workflow-level knob settings (*e.g.,* number of frames, STT on/off, debaters and rounds) for each (workflow, SLO)-pair, considering only those feasible under the accuracy and latency SLO.

2. Model/tool provisioning: the chosen model or tool for each executor in each selected workflow configuration.

3. Resource allocation: the number of instances n_m to launch for each selected model profile m. A profile encodes a specific model, GPU type, and parallelism strategy, so choosing m implicitly fixes the hardware and parallelism degree.

4. Routing map: the fraction of load from each (workflow, SLO)-pair routed to each provisioned instance, encoding cross-workflow multiplexing.

Constraints. The constraints enforce:

1. SLO feasibility: only configurations whose profiled accuracy and latency meet the SLO tier are eligible.

2. Demand satisfaction: the aggregate routed load to each instance must serve the projected peak demand of the workflows assigned to it, leaving headroom for short-term variance that the auto-scaler absorbs (Section 3.4).

3. Capacity: per-instance load stays within the throughput envelope at which the profile satisfies the latency SLO.

4. Resource budget: total GPU usage across all provisioned instances respects the available pool per GPU type.

Objective. The optimizer supports three interchangeable objectives: (a) minimize total energy, (b) minimize total dollar cost, or (c) maximize aggregate accuracy subject to a cost budget. A key design choice is to *decouple peak provisioning from average utilization*: instance counts n_m are sized for projected peak demand (ensuring SLO compliance), while routing fractions are optimized against average demand (driving cost and energy efficiency through colocation).

External and proprietary models. Workflows that call proprietary hosted models (*e.g.,* GPT-4o, Claude) fit the same formulation: such models appear as additional *profiles* with fixed per-token cost and latency, no GPU requirement, and provider-imposed rate limits encoded as capacity constraints. The MILP then naturally trades off self-hosted versus external execution under the same objective.

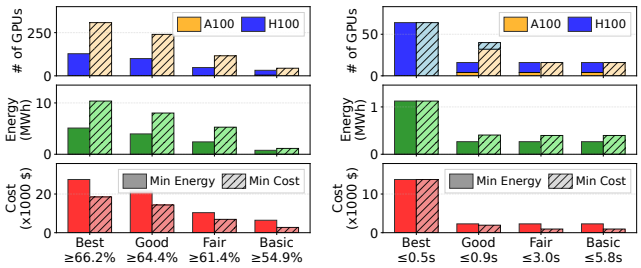
The deployment plan does *not* prescribe per-request dispatch, which remains a runtime responsibility (Section 3.4). Once an executable workflow is generated for all valid SLO tiers of an onboarded workflow, it is added to the Murakkab workflow registry and is ready to serve requests.

3.4 Execution

At runtime, Murakkab receives incoming requests from end-users with a payload that contains the identifier of the agentic workflow being invoked, any input query/data, and the SLOs. Murakkab looks up the registry to obtain the corresponding executable workflow and submits it for execution.

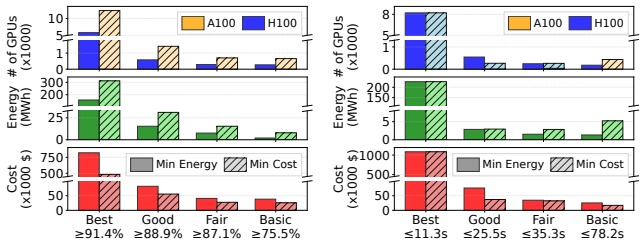
Dynamic Workflow Requests. End-users can either invoke a *particular* agentic workflow or send a request with a natural language query (Figure 5c), any input data to operate on, and SLOs, *without* specifying an agentic workflow to use. The workflow orchestrator parses the query into one or more subtasks, mapping each to an appropriate executor or an existing workflow. Thus, Murakkab dynamically composes workflows from existing building blocks available to it.

SLOs. Each request has the option to specify a quality, latency and cost SLOs. We assign four SLO tiers for quality and



(a) Accuracy SLOs. (b) Latency SLOs.

Figure 7: Video Q/A workflow configured for different SLOs and optimization objectives.



(a) Accuracy SLOs. (b) Latency SLOs.

Figure 8: Code generation workflow configured for different SLOs and optimization objectives.

end-to-end latency: *best*, *good*, *fair*, and *basic*. The SLO tiers correspond to the best, 95^{th}, 80^{th}, and 50^{th} percentile values of accuracy and latency available among the set of all workflow, model, and hardware configurations.

Runtime Optimization. While request dispatch is deterministic given the selected workflow, achieving efficient execution under dynamic, multi-tenant conditions requires continuous adaptation. The optimizer runs in the background after every optimization epoch, in our case every 60 minutes, to adapt to the most up-to-date load and resource availability in the system. The state of the previous epochs is used to project the load for each workflow in the next epoch. Using this information, the optimizer reconfigures the workflows and updates their executable workflows in the registry.

Auto-Scaler. Predicting end-to-end resource usage in agentic workflows is difficult, as input-dependent control flow and intermediate outputs propagating along data flows determine actual demand (*e.g.,* Figures 2b and 2d). For example, a video Q/A workflow with 10 frames and STT on Llava-OneVision-7B [47] produces 600 and 1200 tokens in the 50^{th} and 99^{th} percentile, respectively, highlighting high variance.

To handle such variability, Murakkab includes an *auto-scaler* that monitors per-model instance load over short windows (seconds to minutes) and rapidly scales out when needed. The optimizer can be configured to be conservative (*i.e.,* consider the tail percentile and provision more resources) or optimistic (*i.e.,* consider a more common case and let the auto-scaler handle variations in the short-term).

We set thresholds for auto-scaling based on the performance-throughput characteristic in executor profiles. This mechanism prioritizes avoiding SLO violations over short-term allocation optimality. Murakkab also maintains spare resources to absorb demand spikes and, when it detects significant deviations in workload or resource usage, triggers early re-optimization to adapt quickly.

4 Evaluation

4.1 Experimental Setup

Hardware. We run our experiments on A100 and H100 VMs from Microsoft Azure. Each A100 VM has 8×NVIDIA A100 (80GB) GPUs and an AMD EPYC 7V12 64-Core processor, while each H100 VM has 8×NVIDIA H100 (80GB) GPUs with an Intel Xeon (Sapphire Rapids) processor. We use vLLM (v0.9) [44] as the LLM inference engine, speaches-ai (v0.7) [76] as the speech-to-text model serving engine, and OmDet [89] for object detection model serving.

Production Traces. Since no publicly available traces exist for production agentic workflow serving, we approximate workload arrivals using LLM serving traces collected over a 24-hour period in May 2024 from Azure's LLM inference service for *chat* and *coding* applications [78] (Section A.4).

Agentic Workflows. Our evaluation focuses on the video Q/A and code generation workflows. We map the *chat* requests from the trace to the *video Q/A* workflow and *coding* requests to the *code generation* workflow.

Policies. We consider multiple scheduling policies:

1. *LangGraph* [46] (LG) is a hand-crafted baseline that balances cost and accuracy for both workflows, using Gemma-3-27B and A100 GPUs. However, it lacks visibility into the workflows and has a disconnect between orchestration and resource management, resulting in lack of adaptability to shifting demand.
2. *LangGraph Auto-Scaling* (LG + Auto) is a custom resource auto-scaling policy for LangGraph (since it does not support a default auto-scaling mechanism). This is the same auto-scaling policy used by Murakkab for a fair comparison.
3. *Murakkab Optimized* (Mkb Opt) optimizes each workflow–SLO combination for a specific objective (either minimizing energy or cost).
4. *Murakkab Optimized + Multiplexing* (Mkb Opt+Mult) *jointly* optimizes requests across all workflow–SLO combinations to maximize colocation and sharing of model instances for even better efficiency.

We assume the 90^{th} percentile token generation load from our profiles when making resource allocation decisions for all policies for a fair comparison. When evaluating Murakkab over the production traces, we set the optimization epoch to 60 minutes, and also conduct a sensitivity analysis for this choice in Section 4.7.

4.2 Single-Workflow Optimization

To understand how Murakkab optimizes a single workflow under varying request SLOs and optimization objectives, we run Murakkab with two optimization objectives: *minimize energy consumption* and *minimize execution cost*. We use the workload traces [78] for each workflow and assume that all requests have the same SLO for each experiment. We report the type and number of GPUs allocated, energy consumption in `MWh`, and the cost in `$`.

Accuracy SLOs. Figure 7a shows the requests across SLO accuracies. When minimizing energy, Murakkab lowers energy consumption from 5.1 MWh at 66.2% accuracy (*best*) to 3.9 MWh at 64.4% accuracy (*good*) – an energy reduction of 23.5% with negligible accuracy impact. Tolerating a drop to 61.4% (*fair*) yields substantial savings of 2.6 MWh from the peak consumption. On the other hand, when optimizing for cost, Murakkab reduces from \$18.5k to \$14.3k while marginally dropping accuracy between *best* to *good*. Allowing accuracy to drop further to 61.4% reduces expenses to \$6.9k, a $\approx 4\times$ decrease compared to the most expensive configuration. Murakkab prefers using H100 GPUs when minimizing energy, trading off cost for energy savings.

Figure 8a shows results for the code generation workflow. Energy use ranges from 312 MWh to 2 MWh, and cost from \$820k to \$25k, across varying accuracy levels. Relaxing the accuracy SLO from *best* to *good* yields a sharp drop in energy ($\approx 10.5\times$) and cost ($\approx 8.7\times$), mainly due to Murakkab switching from DeepSeek-Qwen-32B to Gemma-3-27B, which has lower token and operation costs.

Latency SLOs. Murakkab serves latency-sensitive requests with varying SLO tiers, while minimizing energy or cost. A similar pattern in resource and cost savings emerges for both workflows as shown in Figures 7b and 8b. We guarantee a basic accuracy tier even for latency SLO requests (*e.g.,* 50% for video Q/A). For example, Murakkab can reduce energy consumption from 1.1 MWh to 266 kWh at a slight increase in end-to-end latency, from 0.5 s to 0.9 s for the video Q/A workflow. On the other hand, it can reduce energy consumption from 227 MWh to 2.8 MWh (as a result of changing to a different model with different tensor parallelism).

We detail the most common workflow configurations chosen by Murakkab's optimizer for each of the experiments over the entire arrival trace in Sections A.2 and A.3 (Tables 5 and 6). We observe similar trends for the Math Q/A workflow and report the results in Section A.1 (Figures 17a and 17b).

Policy	# of GPUs	Energy (MWh)	Cost ($\times$1000 \$)
LangGraph	2568	82.1	211.7
LangGraph+Auto	2472	80.6	112.3
Murakkab Opt	1164	27.7	57.2
Murakkab Opt+Mult	912	22.1	47.2

Table 2: Comparison of resource usage, energy, and cost. Murakkab substantially reduces GPUs, energy, and cost compared to LangGraph, with further gains from multiplexing across all video Q/A and code generation workflow requests.

4.3 Multi-Workflow Optimization

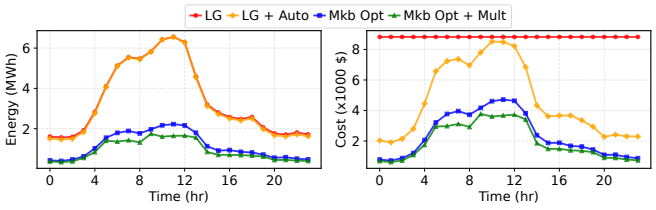
We measure the improvements in efficiency as a result of joint optimization and multiplexing across different workflow–SLO combinations. For this experiment, we run video Q/A and code generation requests together and assign 70% requests to be high-accuracy and 30% requests to low-latency, both with *good tier* (Section 3.4).

Figure 9a shows LangGraph has a fixed allocation with a fixed cost. Its energy consumption varies depending on resource usage, but is significantly higher than the other configurations due to higher idle-power consumption of its GPUs. Table 2 summarizes the results for the 24 hour trace. This configuration uses ≈2,568 A100 GPUs, leading to a flat hourly cost. With auto-scaling, LangGraph reduces idle power consumption and achieves approximately $1.8\times$ cost savings compared to vanilla LangGraph. A non-hand-crafted baseline could be *significantly* more inefficient than our current baseline from requiring bigger models and more resources or result in more SLO violations if under-provisioned.

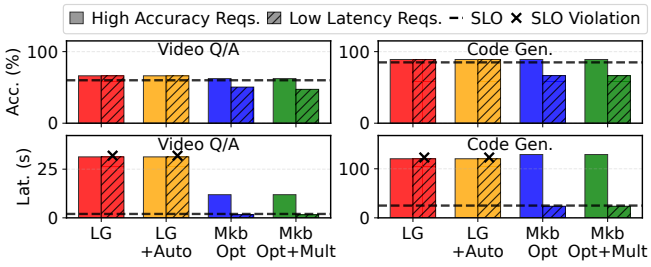
Mkb Opt. This policy optimizes each workflow–SLO combination to minimize cost, although *without* considering the demand across other combinations running in the system. It can adapt to changing load (leveraging the difference between peak and average utilization) to change model and resource allocation over time. It requires 1,164 A100 GPUs, 27.7 MWh energy, and \$57,224 cost.

Mkb Opt+Mult. This policy additionally leverages its *holistic* view of all workflows and their demand to maximize multiplexing opportunities. It requires 912 A100 GPUs (21.6% reduction), 22.1 MWh energy (20.2% reduction), and \$47,238 cost (17.4% reduction) respectively.

Accuracy and Latency. LangGraph cannot distinguish between per-request SLO variations to configure workflows appropriately. Therefore, high-accuracy and low-latency SLO requests have the same accuracy and latency, resulting in latency SLO violations, as shown in Figure 9b. Murakkab leverages per-request SLO information to provision enough resources for each request, as can be seen in the difference in accuracy and latency of the two categories of requests, allowing for higher resource-efficiency.



(a) Energy and cost over time. LangGraph is agnostic to load shifts, LangGraph with auto-scaling simply scales the same models and resources, while Murakkab completely reconfigures workflows to maximize resource-efficiency.



(b) Accuracy and latency for all requests. LangGraph does not distinguish between request SLO categories.

Figure 9: Comparing: (1) a hand-crafted LangGraph configuration (*LG*), (2) LangGraph with auto-scaling (*LG+Auto*), (3) Murakkab optimizing individual workflows (*Mkb Opt*), and (4) Murakkab jointly-optimizing across workflows + multiplexing resources (*Mkb Opt+Mult*).

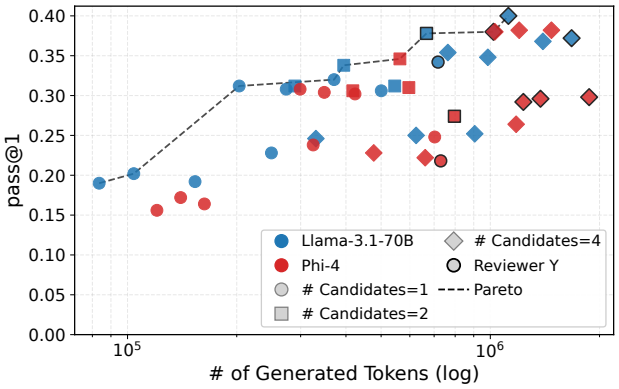


Figure 10: Cost–accuracy frontier of the dynamic coding pipeline on LiveCodeBench. Each point is a workflow configuration; color denotes the code-writer model, marker shape the number of parallel candidates, and a bold edge indicates that a review phase is included. The dashed line is the Pareto frontier across all configurations.

4.4 Dynamic Coding Pipeline

Dynamic coding agents (*e.g.,* Claude Code [4]) provide a challenging setting because execution paths are highly adaptive and cannot be captured by a static workflow graph. To understand how dynamic coding pipelines benefit from Murakkab, we focus on the *workflow-optimization* phase as the evaluation of the other dimensions like hardware allocation and auto-scaling have already been studied extensively in Sections 4.2 and 4.3. We evaluate the workflow on LiveCodeBench-v5 [40] and report pass@1 on hidden test cases that Murakkab has never observed.

The workflow exposes a rich configuration space spanning multiple dimensions: the model used for code writing and reviewing, the number of candidate solutions to generate in parallel, whether to include a code-review phase, and how the review interacts with the iteration loop. For each request, an orchestrator LLM selects an appropriate writer specialization based on the task being solved (*e.g.,* algorithmic or I/O processing etc.) and determines the level of effort to allocate (*e.g.,* detailed reasoning vs. concise). Execution is dynamic at runtime: requests may terminate after a single attempt if public tests pass, or continue through additional refinement iterations driven by reviewer feedback and test failures. Moreover, several workflow components are themselves hierarchical workflows with independent configuration choices, such as reviewer model selection and review iteration budgets.

Results. Figure 10 shows the resulting Pareto frontier across a representative subset of configurations. The frontier spans an order of magnitude in completion tokens and roughly 2× in pass@1, with neither writer model uniformly best: the smaller model anchors the cost-efficient regime while the larger model contributes the highest-accuracy points when paired with a specialized prompt or reviewer. The review phase itself is not universally helpful – it improves accuracy for some (model, task) combinations and degrades it for others – reinforcing that the right configuration is workload- and request-specific. Across all runs, a substantial fraction of requests terminate early on a public-test pass, while a smaller fraction exhaust the iteration cap, confirming that iteration budget is consumed adaptively rather than uniformly.

Implication. These results illustrate the challenge of deploying coding agents with a static configuration. A static deployment of the coding pipeline must commit to one configuration per workload class – and thereby either over-spend on easy problems or under-perform on hard ones. In contrast, Murakkab leverages the heterogeneous operating points exposed by the workflow frontier to select the most appropriate configuration for each request based on predicted difficulty and current system conditions. The hardware-allocation and auto-scaling benefits demonstrated in Sections 4.2 and 4.3 compose naturally with these workflow-level optimizations, enabling end-to-end efficiency improvements across both algorithmic and infrastructure dimensions.

Available A100s	Available H100s	Allocated A100s	Allocated H100s	Energy (MWh)	Cost (×1000 $)
2000	0	1292	0	24.7	55.7
2000	100	780	100	17.8	52.4
2000	200	536	200	14.5	60.3
2000	300	288	300	12.4	64.8
2000	400	50	400	11.1	70.6
2000	500	0	495	11.0	75.4

Table 3: Murakkab serving multiple workflows varying resource constraints minimizing energy. As more H100 GPUs are available, Murakkab trades off cost for energy (same SLO).

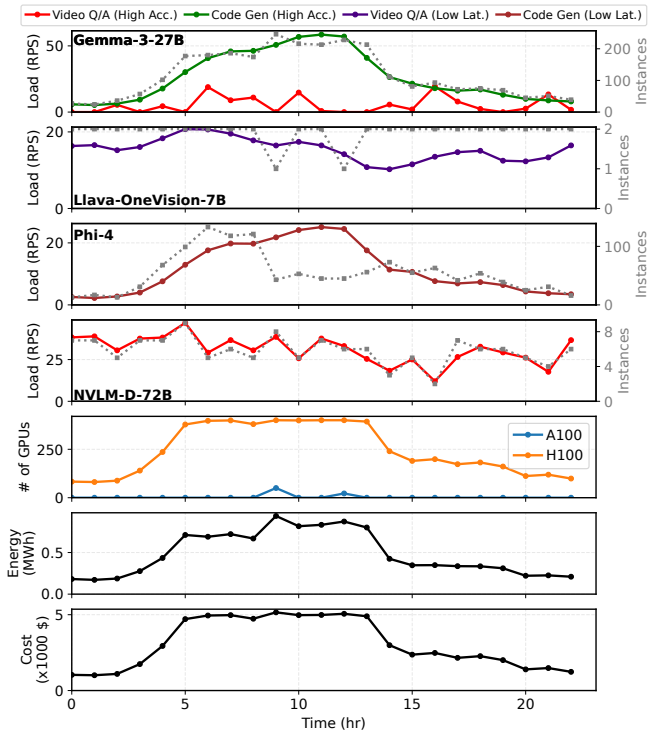


Figure 11: Murakkab adjusts resource allocation and model instances across all workflow-SLO combinations with changing load (under a constraint of 400×H100 GPUs.)

4.5 Adapting to Dynamic Resource Availability

To evaluate how Murakkab adapts to changes in resources (*e.g.,* spot-instances, cloud provider prioritizing resources for other services), we run the same arrival pattern and workflow-SLO distribution from Section 4.3. However, we constrain the type and amount of resources available to Murakkab. We then constrain the resources available to Murakkab: the cluster always provides 2,000 A100 GPUs, while the number of H100 GPUs varies from 0 to 500 in increments of 100. We configure Murakkab to minimize energy consumption while serving incoming workflow requests.

Table 3 shows that Murakkab leverages the H100 GPUs as they become available, while reducing the number of A100 GPUs. Starting with an allocation of 1292 A100 GPUs and no H100 GPUs, which consumes 24.7 MWh energy, Murakkab adapts to the changing H100 availability and leverage most of them (up to 495 H100s) to bring down the energy consumption to 11 MWh. This shows that Murakkab adjusts its resource allocation under resource-constrained settings to satisfy the SLOs while striving for higher resource efficiency.

Execution Analysis. Figure 11 shows Murakkab adapting to changes in load. We consider the *400×H100 GPUs* configuration (without loss of generality) and observe that:

1. Murakkab *scales model instances up/down* after every optimization epoch to adapt to the shift in load dynamics.
2. Murakkab *changes GPU allocation* with system load. It prefers H100 GPUs due to their energy efficiency and uses A100 GPUs for the rest, depending on load fluctuations (*e.g.,* at T=9 hours).
3. Murakkab *reconfigures workflows to maximize colocation*. For example, Gemma serves code generation requests with high-accuracy SLOs and multiplexes load from video Q/A requests with high-accuracy SLOs when there is surplus capacity (*e.g.,* from T=5–9 hours and T=15–19).

4.6 Workflow/DAG-Aware Scheduling

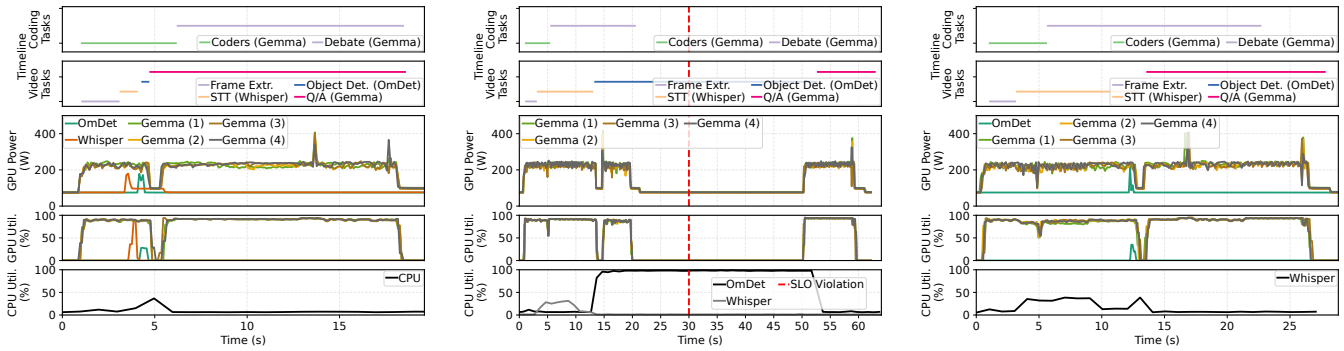
Consider the request: *verify the student's coding solution from a video*, with a *30-second latency SLO*. The orchestrator constructs a DAG with a fan-out for the two sub-tasks that can execute in parallel: (1) video Q/A (Figure 1a) to extract the student's solution, and (2) code generation (Figure 1b) to produce a reference solution for comparison.

Scheduling. Leveraging workflow and model profiles, Murakkab identifies viable scheduling options based on cluster resource availability (Figure 12). All configurations use Gemma-3-27B on 4×A100 GPUs. We highlight setups that leverage both GPUs and CPUs for improved efficiency.

OmDet and Whisper on GPUs. Figure 12a shows OmDet and Whisper running on dedicated A100 GPUs, with a total of 6×A100 GPUs. The two sub-tasks run in near-perfect parallel, with full overlap in execution. However, despite meeting the latency goal, dedicated GPUs are underutilized, and CPUs are mostly idle.

OmDet and Whisper on CPUs. Figure 12b shows Whisper and OmDet running on CPUs, reducing GPU usage to 4×A100 and utilizing idle CPU resources. Whisper runs efficiently without saturating CPUs, making it a good candidate for offloading. In contrast, OmDet fully loads all cores and significantly increases latency, resulting in an SLO violation.

OmDet on GPU, Whisper on CPU. Figure 12c shows OmDet running on 1×A100 GPU and Whisper on CPUs, with a total of 5×A100 GPUs provisioned. Both sub-tasks complete nearly simultaneously, and Whisper's added CPU latency has minimal impact on end-to-end time. This configuration meets the latency SLO while reducing GPU usage compared to the all-GPU setup.

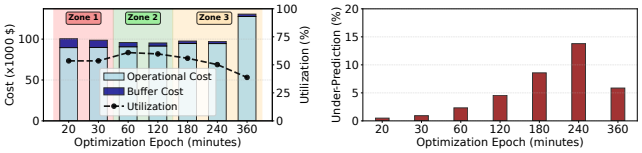


(a) OmDet on 1×A100, Whisper on 1×A100, Gemma on 4×A100s. End-to-end execution completes *within* latency SLO *(6 GPUs)*.

(b) OmDet on CPUs, Whisper on CPUs, Gemma on 4×A100s. End-to-end execution ***violates*** latency SLO *(4 GPUs)*.

(c) OmDet on 1×A100, Whisper on CPUs, Gemma on 4×A100s. End-to-end execution completes *within* latency SLO *(5 GPUs)*.

Figure 12: Executing a user request involving parallel video Q/A and code generation. Murakkab selects the configuration in Figure 12c to minimize energy use while meeting the 30-second latency SLO and maintaining response quality.



(a) Cost *vs.* resource utilization. (b) Demand under-prediction.

Figure 13: Murakkab sensitivity to optimization epoch.

Murakkab in Action. Leveraging workflow stage visibility and pre-generated execution profiles, Murakkab selects the configuration with OmDet on GPU and Whisper on CPU (Figure 12c) as the most resource-efficient option (balancing overall utilization and latency). In contrast, traditional systems treat agentic workflows as opaque, lacking coordination between orchestration and scheduling. This leads to inefficient execution, higher costs, and SLO violations.

4.7 Optimization Frequency Sensitivity Analysis

The choice of optimization epoch presents a fundamental trade-off between resource efficiency, cost, and system responsiveness. We evaluate intervals from 20 minutes to 6 hours, measuring cost, utilization, and worst-case dropped requests. Provisioning new instances (*i.e.,* VM allocation, software setup, and model transfer to GPUs) is assumed to take 20 minutes [31, 38, 62]. We use an exponentially weighted moving average (EWMA) [18], with $\alpha = 0.5$, to predict workload demand at every epoch.

Three-Zone Cost Structure. Figure 13a shows three distinct zones that guide optimal selection.

Zone 1 (10–60 minutes): Buffer-dominated. Frequent reoptimization induces high transition overhead. Excessive GPU provisioning during transitions leads to lower utilization, despite responsive demand adaptation. Frequent model and tool changes can also reduce KV cache efficiency for LLMs.

Zone 2 (60–180 minutes): Balanced. Transition costs and prediction uncertainty offset, yielding peak cost efficiency. Utilization peaks at an epoch of around 60 minutes, reflecting the best balance between adaptation frequency and stability.

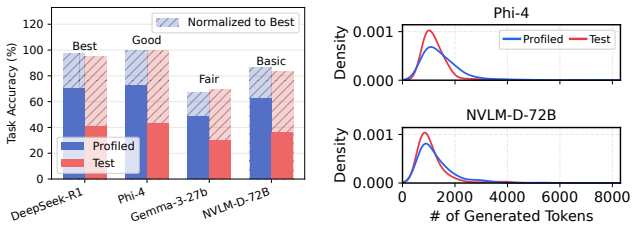
Zone 3 (180+ minutes): Uncertainty-dominated. At long intervals, buffer cost from transition overhead is minimal, but coarse-grained provisioning increasingly diverges from fine-grained, dynamic demand. This reduces prediction accuracy and drives over-provisioning, lowering efficiency.

Worst-case System Responsiveness. We measure responsiveness by demand under-prediction (requests that would be dropped without auto-scaling). This reflects the *worst-case*, as auto-scalers can mitigate mismatches post-provisioning. Figure 13b shows under-prediction is minimal at short intervals but rises steadily, peaking at ∼15% around 240 minutes. Beyond that, longer intervals lead to over-provisioning, lowering under-prediction but hurting utilization.

Operational Implications. This analysis reveals a key trade-off: short intervals improve responsiveness but increase transition costs, while long intervals reduce overhead but hurt prediction accuracy and utilization. Providers can tune this balance based on priorities (*e.g.,* cost, utilization, responsiveness). In our setup, a 60-minute interval offers a middle ground (high utilization, low cost, and manageable shortage risk). While we use EWMA for forecasting, the trade-off holds with more advanced predictors as well.

4.8 Workflow-Profiling Generality Analysis

A core component of Murakkab is its profile-guided optimization, which uses profiles of accuracy, token load, and other performance metrics to guide workflow configuration. To operate reliably across diverse agentic workflows, these profiles must generalize to inputs that were unseen during profiling.



(a) Accuracy and assigned SLOs. (b) PDF of generated tokens.

Figure 14: Murakkab's workflow-profiling generality on an unseen test benchmark.

To evaluate generality, we consider a multi-round mathematical problem solving workflow. Murakkab generates profiles from the Math-500 benchmark [27] (*Profiled*) and tests them on a different MathEval benchmark [32] (*Test*) which was not used during profiling. Figure 14a shows that the same trend in accuracy holds between the profiled and test benchmarks across different models. Therefore, Murakkab can reliably select model configurations based on a request's accuracy SLO requirements without having prior knowledge of the inputs. Figure 14b shows the probability density functions (PDF) of tokens generated by the workflow between the two benchmarks. The distributions closely match with a similar shape, spread and peak indicating that the profiled distribution generalizes well to the test set. Murakkab's optimizer can reliably predict the load demand for incoming requests and provision enough resources to meet the SLOs.

These results demonstrate that Murakkab's profiling captures transferable performance characteristics, enabling the optimizer to make informed decisions on previously unseen workloads (from a similar domain)—a key property for robust, cross-workflow optimization.

5 Related Work

Agentic Workflow Development. Frameworks like LangGraph [46], LangChain [45], and AutoGen [6] build agentic workflows *imperatively*, composing model and tool calls. DSPy [77] and Palimpzest [51, 52] take a *declarative* approach, focusing on prompt and query optimization. However, these frameworks still blur configuration and logic, burden developers with resource management, and struggle to scale efficiently across large configuration spaces.

Automated Workflow Generation and Planning. There has been extensive work from the ML community on automating workflow generation to improve response quality [48, 52, 54, 59, 85, 86]. This line of work is complementary to Murakkab, which can integrate these workflow generation techniques into the Murakkab workflow orchestrator (Section 3.2). Circinus [50] and Aragog [23] perform input-based workflow planning but do not consider resource heterogeneity, dynamism in resource availability and energy/power efficiency. Furthermore, they do not handle dynamism in per-query input/output sizes that may result in load-fluctuations, a common characteristic in LLM-based agentic workflows.

Systems Optimization. Prior systems have focused on accelerating workflow execution with improved data flow management [72, 79], device-aware workflow compilation [5] and scheduling of LLM inference calls in the context of agentic applications [49, 56]. Others target response quality by exploring model selection [14, 29] and scaling properties [13, 63] of test-time compute. Systems like SpotServe [57] and Loki [3] address resource and load dynamism, but remain confined to single-model serving. Some recent works [17, 42, 43] have examined the energy costs of test-time compute and their impact on response quality for individual models. In contrast, Murakkab introduces a fully automated system to navigate these complex tradeoffs for agentic workflows, considering load and resource dynamism in multi-tenant cloud settings.

6 Conclusion

Murakkab brings a declarative programming model and adaptive runtime to the serving of multi-tenant agentic workflows. By decoupling workflow logic from execution configurations and tightly integrating orchestration with cluster management, Murakkab enables dynamic reconfiguration and resource-aware scheduling while honoring diverse SLOs. Our evaluation shows that Murakkab substantially improves efficiency (reducing GPU usage, energy consumption, and cost) without compromising workflow quality or latency. We see opportunities in extending adaptive co-optimization across larger clusters, supporting more heterogeneous accelerators, and exploring workload specialization for emerging classes of agentic applications.

Acknowledgments

We thank our shepherd, Yiying Zhang, anonymous reviewers, and the MIT Parallel and Distributed Systems (PDOS) group for their valuable feedback. This research was supported in part by ACE, one of seven centers in JUMP 2.0, a Semiconductor Research Corporation (SRC) program sponsored by DARPA.

References

[1] Marah Abdin, Jyoti Aneja, Harkirat Behl, Sébastien Bubeck, Ronen Eldan, Suriya Gunasekar, Michael Harrison, Russell J. Hewett, Mojan Javaheripi, Piero Kauffmann, James R. Lee, Yin Tat Lee, Yuanzhi Li, Weishung Liu, Caio C. T. Mendes, Anh Nguyen, Eric Price, Gustavo de Rosa, Olli Saarikivi, Adil Salim, Shital Shah, Xin Wang, Rachel Ward, Yue Wu, Dingli Yu, Cyril Zhang, and Yi Zhang. *Phi-4 Technical Report*. 2024. arXiv: 2412.08905 [cs.CL].

[2] Shubham Agrawal, Adeola Adesoba, Dhruv Nandakumar, Katherine Huang, and Vignesh Srinivasakumar. *Build an Agentic Video Workflow with Video Search and Summarization*. NVIDIA Developer Blog. Available at: https://developer.nvidia.com/blog/build-an-

agentic-video-workflow-with-video-search-and-summarization/. Dec. 2024.

[3] Sohaib Ahmad, Hui Guan, and Ramesh K. Sitaraman. "Loki: A System for Serving ML Inference Pipelines with Hardware and Accuracy Scaling". In: *Proceedings of the 33rd International Symposium on High-Performance Parallel and Distributed Computing*. HPDC '24. Pisa, Italy: Association for Computing Machinery, 2024, pages 267–280.

[4] Anthropic. *Claude Code: Deep coding at terminal velocity*. https://www.anthropic.com/claude-code. Accessed: 19 August 2025. 2025.

[5] Zain Asgar, Michelle Nguyen, and Sachin Katti. *Efficient and Scalable Agentic AI with Heterogeneous Systems*. 2025. arXiv: 2507.19635 [cs.LG].

[6] AutoGen. *AutoGen*. https://microsoft.github.io/autogen/stable//index.html. 2024.

[7] AWS. *Amazon Bedrock Agents*. https://aws.amazon.com/bedrock/agents/. 2025.

[8] Microsoft Azure. *Azure AI Foundry*. https://azure.microsoft.com/en-us/products/ai-foundry. 2025.

[9] Microsoft Azure. *Azure AI Foundry Tool Library*. https://learn.microsoft.com/en-us/azure/ai-foundry/agents/how-to/tools/overview. 2025.

[10] Microsoft Azure. *Understanding costs associated with provisioned throughput units (PTU)*. https://learn.microsoft.com/en-us/azure/ai-foundry/openai/how-to/provisioned-throughput-onboarding. 2025.

[11] Microsoft Azure. *What is an AI Agent?* https://learn.microsoft.com/en-us/azure/ai-foundry/agents/overview. 2025.

[12] Gohar Irfan Chaudhry, Esha Choukse, Íñigo Goiri, Rodrigo Fonseca, Adam Belay, and Ricardo Bianchini. "Towards Resource-Efficient Compound AI Systems". In: *Proceedings of the 2025 Workshop on Hot Topics in Operating Systems*. HotOS '25. Association for Computing Machinery, 2025.

[13] Lingjiao Chen, Jared Quincy Davis, Boris Hanin, Peter Bailis, Ion Stoica, Matei Zaharia, and James Zou. *Are More LLM Calls All You Need? Towards Scaling Laws of Compound Inference Systems*. 2024. arXiv: 2403.02419 [cs.LG].

[14] Lingjiao Chen, Jared Quincy Davis, Boris Hanin, Peter Bailis, Matei Zaharia, James Zou, and Ion Stoica. *Optimizing Model Selection for Compound AI Systems*. 2025. arXiv: 2502.14815 [cs.AI].

[15] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Josh Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. *Evaluating Large Language Models Trained on Code*. 2021. arXiv: 2107.03374 [cs.LG].

[16] Zhendong Chu, Shen Wang, Jian Xie, Tinghui Zhu, Yibo Yan, Jinheng Ye, Aoxiao Zhong, Xuming Hu, Jing Liang, Philip S Yu, et al. "LLM agents for education: Advances and applications". In: *arXiv preprint arXiv:2503.11733* (2025).

[17] Jae-Won Chung, Jiachen Liu, Jeff J. Ma, Ruofan Wu, Oh Jun Kweon, Yuxuan Xia, Zhiyu Wu, and Mosharaf Chowdhury. *The ML.ENERGY Benchmark: Toward Automated Inference Energy Measurement and Optimization*. 2025. arXiv: 2505.06371 [cs.LG].

[18] Petar Cisar, Saša Bošnjak, and Sanja Maravic Cisar. "EWMA algorithm in network practice". In: *International Journal of Computers Communications & Control* 5.2 (2010), pages 160–170.

[19] Google Cloud. *Google Cloud Vertex AI*. https://cloud.google.com/vertex-ai. 2025.

[20] Google Cloud. *What are AI Agents?* https://cloud.google.com/discover/what-are-ai-agents. 2025.

[21] *CrewAI — The Leading Multi-Agent Platform*. https://www.crewai.com/. Accessed: 19 August 2025. 2025.

[22] Ling Dai, Yuan-Hao Jiang, Yuanyuan Chen, Zinuo Guo, Tian-Yi Liu, and Xiaobao Shao. "Agent4EDU: Advancing AI for Education with Agentic Workflows". In: *Proceedings of the 2024 3rd International Conference on Artificial Intelligence and Education*. 2024, pages 180–185.

[23] Yinwei Dai, Zhuofu Chen, Anand Iyer, and Ravi Netravali. *Aragog: Just-in-Time Model Routing for Scalable Serving of Agentic Workflows*. 2025. arXiv: 2511.20975 [cs.DC].

[24] DeepSeek-AI, Daya Guo, Dejian Yang, et al. *DeepSeek-R1: Incentivizing Reasoning Capability in LLMs via Reinforcement Learning*. 2025. arXiv: 2501.12948 [cs.CL].

[25] Yilun Du, Shuang Li, Antonio Torralba, Joshua B. Tenenbaum, and Igor Mordatch. *Improving Factuality and Reasoning in Language Models through Multiagent Debate*. 2023. arXiv: 2305.14325 [cs.CL].

[26] Hugging Face. *Hugging Face Models*. https://huggingface.co/. 2025.

[27] Hugging Face. *MATH-500*. https://huggingface.co/datasets/HuggingFaceH4/MATH-500. 2025.

[28] *Features | Cursor – The AI Code Editor*. https://cursor.com/features. Accessed: 19 August 2025. 2025.

[29] Tao Feng, Yanzhen Shen, and Jiaxuan You. *GraphRouter: A Graph-based Router for LLM Selections*. 2025. arXiv: 2410.03834 [cs.AI].

[30] Chaoyou Fu, Yuhan Dai, Yongdong Luo, Lei Li, Shuhuai Ren, Renrui Zhang, Zihan Wang, Chenyu Zhou, Yunhang Shen, Mengdan Zhang, Peixian Chen, Yanwei Li, Shaohui Lin, Sirui Zhao, Ke Li, Tong Xu, Xiawu Zheng, Enhong Chen, Caifeng Shan, Ran He, and Xing Sun. *Video-MME: The First-Ever Comprehensive Evaluation Benchmark of Multi-modal LLMs in Video Analysis*. 2025. arXiv: 2405.21075 [cs.CV].

[31] Yao Fu, Leyang Xue, Yeqi Huang, Andrei-Octavian Brabete, Dmitrii Ustiugov, Yuvraj Patel, and Luo Mai. "ServerlessLLM: Low-Latency serverless inference for large language models". In: *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*. 2024, pages 135–153.

[32] GitHub. *MathEval*. https://github.com/math-eval/MathEval/tree/main. 2025.

[33] *GitHub Copilot · Your AI pair programmer*. https://github.com/features/copilot. Accessed: 19 August 2025. 2025.

[34] Google. *Google Vertex AI Agent Garden*. https://console.cloud.google.com/vertex-ai/agents/agent-garden. 2025.

[35] Sagar Goyal, Eti Rastogi, Sree Prasanna Rajagopal, Dong Yuan, Fen Zhao, Jai Chintagunta, Gautam Naik, and Jeff Ward. "HealAI: A healthcare LLM for effective medical documentation". In: *Proceedings of the 17th ACM International Conference on Web Search and Data Mining*. 2024, pages 1167–1168.

[36] Aaron Grattafiori, Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, et al. *The Llama 3 Herd of Models*. 2024. arXiv: 2407.21783 [cs.CL].

[37] Gurobi Optimization, LLC. *Gurobi Optimizer Reference Manual*. 2024.

[38] Jianwei Hao, Ting Jiang, Wei Wang, and In Kee Kim. "An empirical analysis of VM startup times in public IaaS clouds". In: *2021 IEEE 14th International Conference on Cloud Computing (CLOUD)*. IEEE. 2021, pages 398–403.

[39] Dan Hendrycks, Collin Burns, Saurav Kadavath, Akul Arora, Steven Basart, Eric Tang, Dawn Song, and Jacob Steinhardt. "Measuring Mathematical Problem Solving With the MATH Dataset". In: *NeurIPS* (2021).

[40] Naman Jain, King Han, Alex Gu, Wen-Ding Li, Fanjia Yan, Tianjun Zhang, Sida Wang, Armando Solar-Lezama, Koushik Sen, and Ion Stoica. *LiveCodeBench: Holistic and Contamination Free Evaluation of Large Language Models for Code*. 2024. arXiv: 2403.07974 [cs.SE].

[41] Haolin Jin, Linghan Huang, Haipeng Cai, Jun Yan, Bo Li, and Huaming Chen. "From LLMs to LLM-based agents for software engineering: A survey of current, challenges and future". In: *arXiv preprint arXiv:2408.02479* (2024).

[42] Yunho Jin, Gu-Yeon Wei, and David Brooks. *The Energy Cost of Reasoning: Analyzing Energy Usage in LLMs with Test-time Compute*. 2025. arXiv: 2505.14733 [cs.LG].

[43] Jiin Kim, Byeongjun Shin, Jinha Chung, and Minsoo Rhu. *The Cost of Dynamic Reasoning: Demystifying AI Agents and Test-Time Scaling from an AI Infrastructure Perspective*. 2025. arXiv: 2506.04301 [cs.LG].

[44] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph E. Gonzalez, Hao Zhang, and Ion Stoica. "Efficient Memory Management for Large Language Model Serving with PagedAttention". In: *Proceedings of the ACM SIGOPS 29th Symposium on Operating Systems Principles*. 2023.

[45] LangChain. *LangChain*. https://github.com/langchain-ai/langchain. 2024.

[46] LangGraph. *LangGraph*. https://www.langchain.com/langgraph. 2024.

[47] Bo Li, Yuanhan Zhang, Dong Guo, Renrui Zhang, Feng Li, Hao Zhang, Kaichen Zhang, Peiyuan Zhang, Yanwei Li, Ziwei Liu, and Chunyuan Li. *LLaVA-OneVision: Easy Visual Task Transfer*. 2024. arXiv: 2408.03326 [cs.CV].

[48] Zelong Li, Shuyuan Xu, Kai Mei, Wenyue Hua, Balaji Rama, Om Raheja, Hao Wang, He Zhu, and Yongfeng Zhang. "Autoflow: Automated workflow generation for large language model agents". In: *arXiv preprint arXiv:2407.12821* (2024).

[49] Chaofan Lin, Zhenhua Han, Chengruidong Zhang, Yuqing Yang, Fan Yang, Chen Chen, and Lili Qiu. "Parrot: efficient serving of LLM-based applications with semantic variable". In: *Proceedings of the 18th USENIX Conference on Operating Systems Design and Implementation*. OSDI'24. Santa Clara, CA, USA: USENIX Association, 2024.

[50] Banruo Liu, Wei-Yu Lin, Minghao Fang, Yihan Jiang, and Fan Lai. *Circinus: Efficient Query Planner for Compound ML Serving*. 2025. arXiv: 2504.16397 [cs.DB].

[51] Chunwei Liu, Matthew Russo, Michael Cafarella, Lei Cao, Peter Baile Chen, Zui Chen, Michael Franklin, Tim Kraska, Samuel Madden, Rana Shahout, and Gerardo Vitagliano. "Palimpzest: Optimizing AI-Powered Analytics with Declarative Query Processing". In: *Proceedings of the Conference on Innovative Database Research (CIDR)*. 2025.

[52] Chunwei Liu, Matthew Russo, Michael Cafarella, Lei Cao, Peter Baille Chen, Zui Chen, Michael Franklin, Tim Kraska, Samuel Madden, and Gerardo Vitagliano. *A Declarative System for Optimizing AI Workloads*. 2024. arXiv: 2405.14696 [cs.CL].

[53] Jerry Liu. *LlamaIndex*. https://github.com/jerryjliu/llama_index. 2022.

[54] Jiale Liu, Yifan Zeng, Shaokun Zhang, Chi Zhang, Malte Højmark-Bertelsen, Marie Normann Gadeberg, Huazheng Wang, and Qingyun Wu. *Divide, Optimize, Merge: Fine-Grained LLM Agent Optimization at Scale*. 2025. arXiv: 2505.03973 [cs.CL].

[55] Zijun Liu, Yanzhe Zhang, Peng Li, Yang Liu, and Diyi Yang. "A dynamic LLM-powered agent network for task-oriented agent collaboration". In: *First Conference on Language Modeling*. 2024.

[56] Michael Luo, Xiaoxiang Shi, Colin Cai, Tianjun Zhang, Justin Wong, Yichuan Wang, Chi Wang, Yanping Huang, Zhifeng Chen, Joseph E. Gonzalez, and Ion Stoica. *Autellix: An Efficient Serving Engine for LLM Agents as General Programs*. 2025. arXiv: 2502.13965 [cs.LG].

[57] Xupeng Miao, Chunan Shi, Jiangfei Duan, Xiaoli Xi, Dahua Lin, Bin Cui, and Zhihao Jia. *SpotServe: Serving Generative Large Language Models on Preemptible Instances*. 2023. arXiv: 2311.15566 [cs.DC].

[58] Daye Nam, Andrew Macvean, Vincent Hellendoorn, Bogdan Vasilescu, and Brad Myers. "Using an LLM to Help With Code Understanding". In: *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*. ICSE '24. Lisbon, Portugal: Association for Computing Machinery, 2024.

[59] Boye Niu, Yiliao Song, Kai Lian, Yifan Shen, Yu Yao, Kun Zhang, and Tongliang Liu. "Flow: Modularized agentic workflow automation". In: *arXiv preprint arXiv:2501.07834* (2025).

[60] Diego Novillo. "SamplePGO - The Power of Profile Guided Optimizations without the Usability Burden". In: *2014 LLVM Compiler Infrastructure in HPC*. 2014, pages 22–28.

[61] NVIDIA. *NeMo Agent Toolkit*. https://developer.nvidia.com/nemo-agent-toolkit. 2025.

[62] NVIDIA. *NVIDIA DOCA Overview*. https://docs.nvidia.com/doca/archive/2-9-0/nvidia+doca+overview/index.html. 2025.

[63] Isaac Ong, Amjad Almahairi, Vincent Wu, Wei-Lin Chiang, Tianhao Wu, Joseph E. Gonzalez, M Waleed Kadous, and Ion Stoica. *RouteLLM: Learning to Route LLMs with Preference Data*. 2025. arXiv: 2406.18665 [cs.LG].

[64] OpenAI. *ChatGPT (Mar 14 version) [Large language model]*. https://chat.openai.com/chat. 2023.

[65] OpenAI. *Whisper Large V3 Model*. https://huggingface.co/openai/whisper-large-v3. 2024.

[66] OpenAI. *Introducing OpenAI o3 and o4-mini*. 2025.

[67] OpenAI. *OpenAI Agents SDK*. https://openai.github.io/openai-agents-python/tools/. 2025.

[68] Melissa Z Pan, Negar Arabzadeh, Riccardo Cogo, Yuxuan Zhu, Alexander Xiong, Lakshya A Agrawal, Huanzhi Mao, Emma Shen, Sid Pallerla, Liana Patel, et al. "Measuring Agents in Production". In: *arXiv preprint arXiv:2512.04123* (2025).

[69] *Plix AI*. https://plix.ai/. Accessed: 19 August 2025. 2025.

[70] Model Context Protocol. *Model Context Protocol (MCP)*. https://modelcontextprotocol.io/docs/getting-started/intro. 2025.

[71] Yujia Qin, Shihao Liang, Yining Ye, Kunlun Zhu, Lan Yan, Yaxi Lu, Yankai Lin, Xin Cong, Xiangru Tang, Bill Qian, et al. "ToolLLM: Facilitating large language models to master 16000+ real-world APIs". In: *arXiv preprint arXiv:2307.16789* (2023).

[72] Deepti Raghavan, Keshav Santhanam, Muhammad Shahir Rahman, Nayani Modugula, Luis Gaspar Schroeder, Maximilien Cura, Houjun Liu, Pratiksha Thaker, Philip Levis, and Matei Zaharia. *Alto: Orchestrating Distributed Compound AI Systems with Nested Ancestry*. 2025. arXiv: 2403.04311 [cs.AI].

[73] Matthew Renze and Erhan Guven. "Self-reflection in llm agents: Effects on problem-solving performance". In: *arXiv preprint arXiv:2405.06682* (2024).

[74] Francisco Romero, Johann Hauswald, Aditi Partap, Daniel Kang, Matei Zaharia, and Christos Kozyrakis. "Optimizing Video Analytics with Declarative Model Relationships". In: *Proc. VLDB Endow.* 16.3 (Nov. 2022), pages 447–460.

[75] Francisco Romero, Mark Zhao, Neeraja J. Yadwadkar, and Christos Kozyrakis. *Llama: A Heterogeneous & Serverless Framework for Auto-Tuning Video Analytics Pipelines*. 2021. arXiv: 2102.01887 [cs.DC].

[76] *Speaches – OpenAI-API compatible server for speech-to-text, speech-to-speech, and translation*. https://speaches.ai/. Accessed: 20 August 2025. 2025.

[77] Stanford NLP Group. *DSPy: The Framework for Programming—Not Prompting—Language Models*. https://github.com/stanfordnlp/dspy. 2023.

[78] Jovan Stojkovic, Chaojie Zhang, Íñigo Goiri, Josep Torrellas, and Esha Choukse. "DynamoLLM: Designing LLM inference clusters for performance and energy efficiency". In: *2025 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE. 2025, pages 1348–1362.

[79] Xin Tan, Yimin Jiang, Yitao Yang, and Hong Xu. *Teola: Towards End-to-End Optimization of LLM-based Applications*. 2025. arXiv: 2407.00326 [cs.DC].

[80] Yunlong Tang, Jing Bi, Siting Xu, Luchuan Song, Susan Liang, Teng Wang, Daoan Zhang, Jie An, Jingyang Lin, Rongyi Zhu, Ali Vosoughi, Chao Huang, Zeliang Zhang, Pinxin Liu, Mingqian Feng, Feng Zheng, Jianguo Zhang, Ping Luo, Jiebo Luo, and Chenliang Xu. *Video Understanding with Large Language Models: A Survey*. 2025. arXiv: 2312.17432 [cs.CV].

[81] Gemma Team, Aishwarya Kamath, Johan Ferret, et al. *Gemma 3 Technical Report*. 2025. arXiv: 2503.19786 [cs.CL].

[82] Jize Wang, Ma Zerun, Yining Li, Songyang Zhang, Cailian Chen, Kai Chen, and Xinyi Le. "GTA: a benchmark for general tool agents". In: *Advances in Neural Information Processing Systems* 37 (2024), pages 75749–75790.

[83] Xingyao Wang, Yangyi Chen, Lifan Yuan, Yizhe Zhang, Yunzhu Li, Hao Peng, and Heng Ji. "Executable Code Actions Elicit Better LLM Agents". In: *Forty-first International Conference on Machine Learning*. 2024.

[84] Patrick Wintermeyer, Maria Apostolaki, Alexander Dietmüller, and Laurent Vanbever. "P2GO: P4 profile-guided optimizations". In: *Proceedings of the 19th ACM Workshop on Hot Topics in Networks*. 2020, pages 146–152.

[85] Shirley Wu, Parth Sarthi, Shiyu Zhao, Aaron Lee, Herumb Shandilya, Adrian Mladenic Grobelnik, Nurendra Choudhary, Eddie Huang, Karthik Subbian, Linjun Zhang, et al. “Optimas: Optimizing Compound AI Systems with Globally Aligned Local Rewards”. In: *arXiv preprint arXiv:2507.03041* (2025).

[86] Jiayi Zhang, Jinyu Xiang, Zhaoyang Yu, Fengwei Teng, Xionghui Chen, Jiaqi Chen, Mingchen Zhuge, Xin Cheng, Sirui Hong, Jinlin Wang, et al. “Aflow: Automating agentic workflow generation”. In: *arXiv preprint arXiv:2410.10762* (2024).

[87] Lu Zhang, Tiancheng Zhao, Heting Ying, Yibo Ma, and Kyusong Lee. *OmAgent: A Multi-modal Agent Framework for Complex Video Understanding with Task Divide-and-Conquer*. 2024. arXiv: 2406.16620 [cs.CL].

[88] Yaolun Zhang, Yinxu Pan, Yudong Wang, and Jie Cai. “Py-Bench: Evaluating LLM agent on various real-world coding tasks”. In: *arXiv preprint arXiv:2407.16732* (2024).

[89] Tiancheng Zhao, Peng Liu, and Kyusong Lee. “OmDet: Large-scale vision-language multi-dataset pre-training with multimodal detection network”. In: *IET Computer Vision* 18.5 (Jan. 2024), pages 626–639.

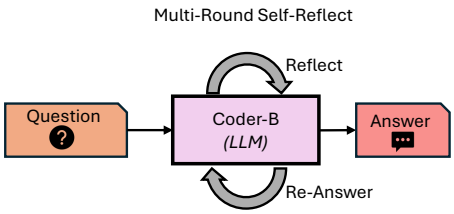


Figure 15: Math Q/A workflow with self-reflect [73] structure.

Policy	# of GPUs	Energy (MWh)	Cost (×1000 $)
Static	4448	169.87	367.2
Mkb Opt	1875	62.88	123.0
Mkb Opt+Mult	1660	52.66	104.6

Table 4: Comparison of resource usage, energy, and cost. Murakkab substantially reduces GPUs, energy, and cost compared to static allocation, with further gains from multiplexing across all math Q/A and code generation workflow requests.

A Appendix

A.1 Math Q/A

Math Q/A, as shown in Figure 15, is a workflow to answer mathematical problems. It uses a self-reflect [73] structure where a *mathematician* agent generates a response to the question, self-reflects on the response, and iteratively continues this until it is confident in the answer or a limit on the maximum number of rounds is reached.

A.1.1 Characterization

We present the results (Figure 16) for different models and different number of self-refine rounds (R) (Figure 16a) and the token generation load in terms of prompt (Figure 16c) and completion tokens (Figure 16b) for different configurations.

A.1.2 Single-Workflow Optimization

Similar to Section 4.2, we optimize the math Q/A workflow using Murakkab for different accuracy and latency SLO tiers and optimization objectives. The results are presented in Figure 17.

A.1.3 Multi-Workflow Optimization

Similar to Section 4.3, we evaluate end-to-end execution of running 24 hours of Azure traces [78]. This time, the two categories of requests from the traces map to two agentic workflows: *chat* requests maps to *math Q/A* workflow requests and *coding* requests map to the *code generation* workflow (Figure 1b). The results are shown in Figure 18. The handcrafted *static* baseline policy (similar to LangGraph [46]) has a fixed allocation of DeepSeek-Qwen-32B for the math Q/A workflow and Gemma-3-27B for code generation workflow – balancing quality and resource usage. As explained in Section 4, the static baseline has no visibility into the workflow, load or resource availability and suffers from underutilization. Murakkab on the other hand is adaptive to load for each workflow–SLO combination and appropriately configures workflows at each optimization epoch, resulting in massive resource savings. When allowed to multiplex across resources, it yields further savings by colocating the requests from all workflows onto shared model instances whenever appropriate. Table 4 summarizes the results for the 24 hour duration. We observe a reduction of $\approx 2.7\times$ in GPUs, $\approx 3.2\times$ in energy, and $\approx 3.5\times$ in cost from the static baseline to Murakkab's best optimization policy.

SLO	Objective	Tier	Model	STT	Frames	GPU	TP	TPOT (s)	TPS
Acc.	Cost	Best	Gemma-3-27B	Y	10	A100	4	0.0624	699
		Good	Gemma-3-27B	Y	5	A100	4	0.0624	700
		Fair	NVLM-D-72B	Y	5	A100	4	0.0966	325
		Basic	Llava-OneVision-7B	Y	5	A100	4	0.0224	2244
	Energy	Best	Gemma-3-27B	Y	10	H100	4	0.0484	1688
		Good	Gemma-3-27B	Y	5	H100	4	0.0472	1668
		Fair	NVLM-D-72B	Y	5	H100	4	0.0650	766
		Basic	Llava-OneVision-7B	Y	5	H100	4	0.0079	3271
Lat.	Cost	Best	Llava-OneVision-7B	Y	1	H100	4	0.0044	479
		Good	Llava-OneVision-7B	Y	1	A100	4	0.0085	926
		Fair	Llava-OneVision-7B	Y	1	A100	4	0.0224	2244
		Basic	Llava-OneVision-7B	Y	1	A100	4	0.0224	2244
	Energy	Best	Llava-OneVision-7B	Y	1	H100	4	0.0044	479
		Good	Llava-OneVision-7B	Y	1	H100	4	0.0070	2836
		Fair	Llava-OneVision-7B	Y	1	H100	4	0.0070	2836
		Basic	Llava-OneVision-7B	Y	1	H100	4	0.0070	2836

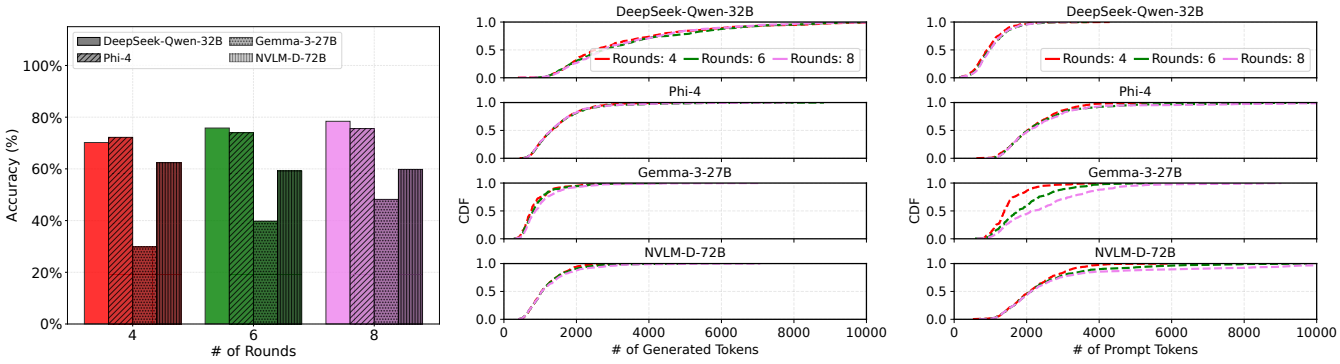
Table 5: Video Configurations with GPU Details

A.2 Video

We present the most commonly chosen configuration by Murakkab for each workflow–SLO combination of the video Q/A workflow under different optimization objectives in Table 5 (corresponding to the experiment presented in Section 4.2). It shows various knobs (*e.g.,* model, number of frames, GPU type etc.) and the offered load per model instance in tokens-per-second (TPS) with the observed latency in time-per-output-token (TPOT) For example, we can observe the changing model and number of frames processed as the accuracy SLO is relaxed. For the latency SLO requests, Murakkab keeps the same workflow- and model-level knobs but changes GPU type and increases the allowed load per model instance to increase batching as the SLO is relaxed.

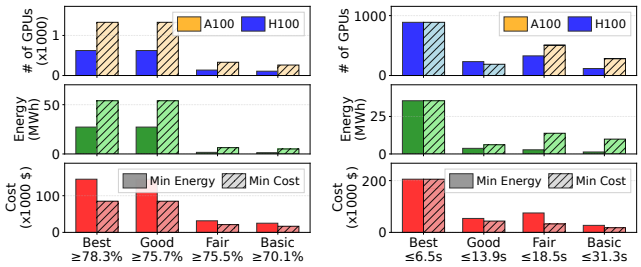
A.3 Code Generation

We present the most commonly chosen configuration by Murakkab for each workflow–SLO combination of the code generation workflow under different optimization objectives in Table 6 (corresponding to the experiment presented in Section 4.2). It shows various knobs (*e.g.,* model, number of debaters, number of debate rounds etc.) and the offered load per model instance in tokens-per-second (TPS) with the observed latency in time-per-output-token (TPOT) For example, we can observe the changing model and number of debaters as the accuracy SLO is relaxed. For the latency SLO requests, Murakkab mostly keeps the same workflow- and model-level knobs but changes GPU type and increases the allowed load



(a) Math Q/A accuracy. (b) Math Q/A model load (generated tokens). (c) Math Q/A model load (prompt tokens).

Figure 16: Workflow accuracy under different configurations and the token processing load on the respective models.



(a) Accuracy SLOs. (b) Latency SLOs.

Figure 17: Math Q/A workflow configured for different SLO and optimization objectives.

SLO	Objective	Tier	Model	Agents	Rounds	GPU	TP	TPOT (s)	TPS
Acc.	Cost	Best	DeepSeek-Qwen-32B	4	4	A100	4	0.0767	653
		Good	Gemma-3-27B	4	4	A100	4	0.0624	700
		Fair	Gemma-3-27B	2	4	A100	4	0.0624	700
		Basic	Phi-4	2	4	A100	2	0.0609	623
	Energy	Best	DeepSeek-Qwen-32B	4	4	H100	4	0.0387	1390
		Good	Gemma-3-27B	4	4	H100	4	0.0496	1709
		Fair	Gemma-3-27B	2	4	H100	4	0.0487	1693
		Basic	Phi-4	2	4	H100	1	0.0373	757
Lat.	Cost	Best	NVLM-D-72B	2	2	H100	8	0.0129	84
		Good	Phi-4	2	2	H100	2	0.0169	1036
		Fair	Phi-4	2	2	H100	2	0.0218	1185
		Basic	Phi-4	2	2	A100	2	0.0609	623
	Energy	Best	NVLM-D-72B	2	2	H100	8	0.0129	84
		Good	Phi-4	2	2	H100	1	0.0165	248
		Fair	Phi-4	2	2	H100	1	0.0253	552
		Basic	Phi-4	2	2	H100	1	0.0372	755

Table 6: Code Generation Configurations with GPU Details

per model instance to increase batching as the SLO is relaxed. Notably, it also changes the tensor parallelism for the same model (Phi-4 [1]) between optimizing for cost and energy.

A.4 Azure Traces

We use a subset of LLM serving traces released by Azure [78] from 08:00 05/15/2024 to 08:00 05/16/2024 shown in Figure 19.

A.5 Optimization Formulation

We formulate our optimization problem (Section 3.3) as a temporal resource allocation task for serving multiple workflows with heterogeneous SLOs across diverse hardware and model profiles.

Sets and Indices.

- $\mathcal{W}$: workflows
- $\mathcal{S}$: SLO types
- $\mathcal{M}$: model profiles
- $\mathcal{C}_w$: workflow configurations for w
- $\mathcal{G}$: resource types

Parameters.

- $\lambda^{\text{peak}}_{w,s}$: Peak request rate for workflow w with SLO s
- $\lambda^{\text{avg}}_{w,s}$: Average request rate for workflow w with SLO s
- α: Unified buffer factor (default 1.15)
- $\tau_{w,s}$: SLO threshold for workflow w and SLO type s
- a_c: Accuracy of workflow configuration $c \in \mathcal{C}_w$
- t_c: Tokens per request for workflow configuration c
- θ_m: Token throughput (tokens/sec) for model profile m
- ℓ^{TTFT}_m: Time to first token for model profile m
- ℓ^{TPOT}_m: Time per output token for model profile m
- g_m: Parallelism for model m
- e_m: Energy consumption (kWh) for model profile m
- c_g: Cost per instance per second for resource type $g \in \mathcal{G}$
- B_g: Maximum available resource instances of type g

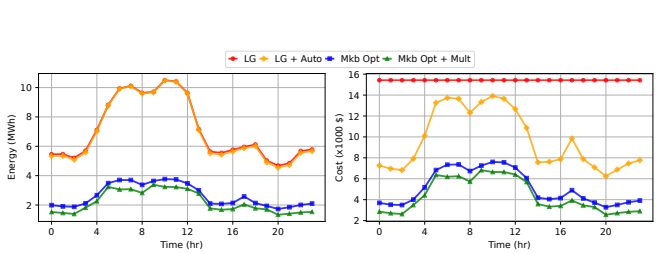
Decision Variables.

- $n_m \in \mathbb{Z}^+$: Number of instances of model profile m
- $x^{\text{peak}}_{w,s,c,m} \in \mathbb{R}^+$: Peak load allocation from (w,s,c) to model m
- $x^{\text{avg}}_{w,s,c,m} \in \mathbb{R}^+$: Average load allocation from (w,s,c) to model m

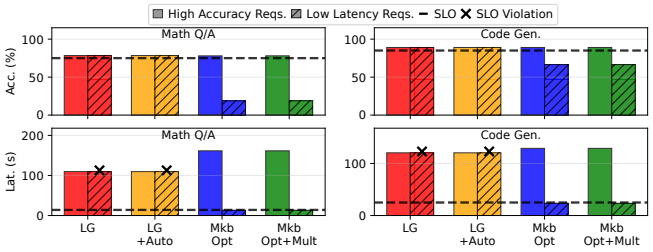
Constraints. Demand Satisfaction (Peak):

Ensure peak demand is met with buffer:

$$\lambda^{\text{peak}}_{w,s} \leq \sum_{c \in \mathcal{C}_w, m \in \mathcal{M}} x^{\text{peak}}_{w,s,c,m} \leq \alpha \cdot \lambda^{\text{peak}}_{w,s}, \quad \forall w \in \mathcal{W}, s \in \mathcal{S} \quad (1)$$



(a) Energy and cost over time. Static policy is agnostic to load shifts while Murakkab reconfigures workflows for higher resource-efficiency.



(b) Accuracy and latency for all requests. Static policy does not distinguish between request SLO categories.

Figure 18: Comparing a hand-crafted static configuration (only A100s) to: (1) Murakkab optimizing individual workflows, and (2) Murakkab jointly-optimizing across workflows + multiplexing resources (Math Q/A + Code Gen).

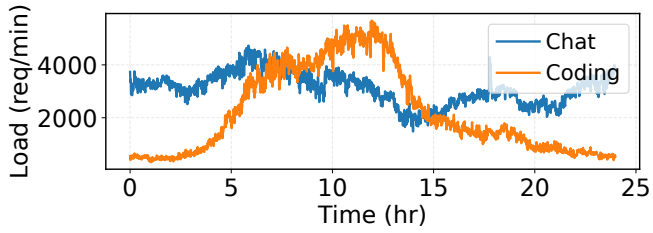


Figure 19: Azure LLM serving traces.

Demand Satisfaction (Average):
Similar bounds for average demand:

$$\lambda^{\text{avg}}_{w,s} \leq \sum_{c \in \mathcal{C}_w, m \in \mathcal{M}} x^{\text{avg}}_{w,s,c,m} \leq \alpha \cdot \lambda^{\text{avg}}_{w,s}, \quad \forall w \in \mathcal{W}, s \in \mathcal{S} \tag{2}$$

Capacity Constraint with Multiplexing:
Account for statistical multiplexing:

$$\mu_m \cdot \sum_{w,s,c} x^{\text{peak}}_{w,s,c,m} \cdot t_c \leq n_m \cdot \theta_m, \quad \forall m \in \mathcal{M} \tag{3}$$

where μ_m is the model-specific multiplexing factor.

SLO Filtering:
For accuracy SLO ($s = \text{max_accuracy}$):

$$x^{\text{peak}}_{w,s,c,m} = 0 \quad \text{if } a_c < \tau_{w,s} \tag{4}$$

For latency SLO ($s = \text{min_latency}$):

$$x^{\text{peak}}_{w,s,c,m} = 0 \quad \text{if } \ell^{\text{TTFT}}_m + t_c \cdot \ell^{\text{TPOT}}_m > \tau_{w,s} \tag{5}$$

Cost Budget Constraint:
If cost SLO is specified:

$$\sum_{w,s,c,m} x^{\text{avg}}_{w,s,c,m} \cdot \frac{t_c}{\theta_m} \cdot g_m \cdot c_{g(m)} \leq \text{Cost}_{\text{budget}} \tag{6}$$

where $\text{Cost}_{\text{budget}} = \sum_{w \in \mathcal{W}} \tau_{w,\text{cost}} \cdot \sum_s \lambda^{\text{avg}}_{w,s}$.

Resource Budget Constraint:

$$\sum_{m:\text{GPU}(m)=g} n_m \cdot g_m \leq B_g, \quad \forall g \in \mathcal{G} \tag{7}$$

SLO Filtering Constraints:

$$x^{\text{peak}}_{w,s,c,m} = 0 \quad \text{if } a_c < \tau_{w,s} \quad \text{(accuracy)} \tag{8}$$

$$x^{\text{peak}}_{w,s,c,m} = 0 \quad \text{if } \ell^{\text{TTFT}}_m + t_c \ell^{\text{TPOT}}_m > \tau_{w,s} \quad \text{(latency)} \tag{9}$$

Cost Budget Constraint:

$$\sum_{w,s,c,m} x^{\text{avg}}_{w,s,c,m} \frac{t_c}{\theta_m} g_m c_{g(m)} \leq \sum_w \tau_{w,\text{cost}} \sum_s \lambda^{\text{avg}}_{w,s} \tag{10}$$

Objective Functions. Minimize Energy:

$$\min \sum_m n_m e_m g_m \tag{11}$$

Minimize Cost:

$$\min \sum_m n_m g_m c_{g(m)} \tag{12}$$

Maximize Accuracy Under a Cost Budget:

$$\max \frac{\sum_{w,s,c,m} x^{\text{avg}}_{w,s,c,m} a_c}{\sum_{w,s} \lambda^{\text{avg}}_{w,s}} - \varepsilon \cdot \text{Cost}_{\text{total}} \tag{13}$$

where $\varepsilon = 0.001$.

Solution Method. The formulated Mixed Integer Linear Program (MILP) is solved using Gurobi [37] with a time limit of 300 seconds. The solution yields an allocation of model instance counts n^*_m and load distributions across model instances for all workflows.

ECO: An AI-Driven Code Efficiency Optimizer for Warehouse Scale Computers (Operational Systems)

Hannah Lin*[1] Martin Maas*[1] Maximilian Roquemore†[2] Arman Hasanzadeh[1] Fred Lewis[2]
Yusuf Simonson[2] Ameya Shringi[2] Hongwen Dai[2] Patrick Musau[1] Tzu-Wei Yang[2]
Amir Yazdanbakhsh[1] Deniz Altinbüken[1] Florin Papa[2] Maggie Nolan Edmonds[2] Aditya Patil[2]
Don Schwarz[2] Satish Chandra‡[2] Chris Kennelly[2] Milad Hashemi[1] Parthasarathy Ranganathan[2]
[1]*Google DeepMind,* [2]*Google*

Abstract

Large Language Models (LLMs) have shown significant promise in automating code efficiency optimization. While prior work demonstrates these techniques on artificial datasets such as programming competitions or small benchmarks, deploying these techniques at scale in production has remained an open problem. Arguably, two challenges have prevented the adoption in large-scale real-world systems: opportunity localization and reliability. First, applying an LLM to every line across a large code base is expensive and prone to generating an overwhelming number of low-quality suggestions, placing unsustainable cognitive load on human code reviewers. Second, the inherent unreliability of LLM-generated code risks introducing errors that can lead to production incidents. These challenges are largely orthogonal to the ML techniques prior work has focused on; they are real-world systems problems.

This paper introduces ECO, a system that automatically modifies source code to improve performance at scale. ECO overcomes the localization problem by combining fleet-wide continuous profiling to identify performance-critical code with an embedding-based search to pinpoint specific optimization candidates, guided by a mined dictionary of performance anti-patterns. It overcomes the reliability problem through a multi-stage verification approach that uses automated testing, LLM-based self-review, and post-deployment monitoring to ensure changes are both correct and effective. Fully productionized and deployed within Google's hyperscale production fleet, ECO has successfully landed over 6,400 commits, changing more than 25,000 lines of production code. Incorrect changes are caught before they are submitted to production, and 99.5% of the submitted commits did not cause any rollbacks. These optimizations have resulted in savings equivalent to several hundred thousand normalized CPU cores, showing that ECO makes LLM-based optimization both practical at scale and highly impactful in real-world settings.

1 Introduction

Over the past few years, an increasing amount of work has shown the potential of Large Language Models (LLMs) to analyze and optimize code for performance [1–10]. By automating a task that traditionally requires extensive and costly manual engineering, LLMs promise to unlock significant efficiency gains, particularly in the long tail of workloads. A study of Google's data centers has shown that a large fraction of CPU cycles in warehouse-scale computers are spent on workloads that are individually too small to justify manual optimization effort [11]. Automating this process could therefore yield substantial resource and energy savings.

However, while these LLM-based approaches show promise on benchmarks such as programming competitions [2, 5, 7, 9, 12, 13], directly applying them in a large-scale, production environment presents significant challenges that hinder their practical adoption. The challenges are not with the LLMs alone (where existing work has focused), but with making them practical in the context of real-world systems:

- **Opportunity Localization**: LLM-based methods need to be provided the code to optimize. In production environments with millions or billions of lines of code, naively running an LLM across the entire repository is computationally prohibitive and generates an overwhelming number of false positives. A practical system must first find and localize high-impact optimization opportunities before attempting to apply a fix.

- **Reliability**: State-of-the-art LLM methods often report success as the best result out of many attempts (e.g., top-$\mathcal{K}$ accuracy) [13]. In a real-world production setting, however, precision is paramount. Any incorrect change can cause outages, and the high rate of subtly buggy or non-performant suggestions from an unguided LLM creates a large verification burden.

In addition, benchmarks fail to capture **system integration** and **attribution**. In complex, large-scale systems, attributing a performance inefficiency to a specific piece of code is difficult [14]. Furthermore, AI-based optimization must integrate with existing engineering workflows, including testing, code review, and deployment, aspects absent in benchmarks.

We present ECO (**E**ngine for **C**ode **O**ptimization), a system that makes LLM-based optimization practical at scale by introducing novel techniques for targeted localization and robust verification. ECO begins by mining historical commits

*Hannah Lin and Martin Maas contributed equally to this work.
†Now at Point72, work done entirely at Google.
‡Now at Meta, work done entirely at Google.

to create a database of performance anti-patterns. These anti-patterns are then used to guide a large-scale vector similarity search [15] across the codebase. To focus this search, we combine it with data from a fleet-wide continuous profiling system, allowing ECO to identify code that is both costly and a likely candidate for a known optimization.

Once a candidate is localized, ECO uses a fine-tuned LLM to generate a code transformation. To ensure the reliability of this edit, it is passed through a multi-stage verification pipeline that includes automated build and unit testing, LLM-based self-review to detect common errors, and finally, human review. Once deployed, the change is continuously monitored to measure its performance impact and avoid regressions.

This paper makes the following contributions:

1. An approach for identifying and localizing performance optimization opportunities by combining continuous profiling with an embedding-based similarity search guided by a mined database of anti-patterns.
2. A methodology for making unreliable LLM-generated edits reliable for production use through automated testing, self-review, and post-deployment monitoring.
3. The design, implementation, and evaluation of ECO, a fully deployed system that embodies these techniques and successfully integrates AI-driven changes into a large-scale, human-in-the-loop engineering workflow.
4. A study focusing on human adoption of code edits, including a human evaluation of 960 code edits to assess the quality of changes, and an at-scale production study of how human engineers adopt ECO-generated changes in Google's production setting.
5. An analysis of ECO's deployment at Google, showing it has successfully landed over 6,400 commits, changing over 25k lines of code and saving the equivalent of several hundred thousand normalized CPU cores. By filtering out and discarding incorrect changes before they are submitted to production, >99.5% of the 6,400 commits did not have to be reverted.

While there has been a substantial amount of prior work on LLM-based code optimization, ECO is largely orthogonal to this work. Instead, it solves the systems problems required to turn these capabilities into scalable impact in a large-scale system infrastructure deployment. We believe it is the first operational system to achieve this.

2 Background & Overview

The use of LLMs in code optimization has seen a large amount of research in the past several years [1]. We begin with an overview of how our approach improves upon the state of the art to solve problems that have prevented adoption of automated optimization in real-world systems.

2.1 Why LLMs for Code Optimization?

Automated code optimization is well-studied, but existing approaches have limitations. Compiler optimizations must be provably correct, preventing them from making semantic changes that a programmer knows are safe in a given context. Static analysis tools like Clang-Tidy [16], Facebook Infer [17], or Semantic Patch [18] can apply rule-based edits to perform such changes, but struggle with the high syntactic variability of real-world code. These tools require narrowly defined patterns and often fail on complex control flow. For example, Clang-Tidy can suggest adding `vector<T>::reserve` to a simple, single-statement loop [19], but cannot handle common real-world cases where a vector is populated across multiple loops, conditional branches, or helper functions (Figure 3b). Writing heuristics for every variant is often more work than fixing the instances manually. Further, many optimizations require domain knowledge that static tools lack, such as when a sorted map can be replaced with a faster, unsorted one.

LLMs overcome these limitations in two ways. First, they can generalize from a few examples to recognize and fix anti-patterns across a wide range of syntactic variations, reducing the need for hand-crafted heuristics. Second, they can often infer developer intent and domain-specific context from the surrounding code and comments, enabling semantic optimizations that are out of reach for traditional tools. Figure 3 shows several examples. While LLMs are more computationally expensive than static checkers, we show in Section 7.4 that the cost of inference is negligible compared to the fleet-wide savings they unlock.

2.2 High-Level Approach

Figure 1 shows an overview of ECO's approach. We start by mining decades of code changes across Google's billions of lines of code spanning over 100,000 packages and thousands of projects [20], to identify a set of canonical code optimizations. Each consists of an *anti-pattern* and a corresponding *optimization* that fixes it. Two examples[1] of these anti-patterns, pre-sizing a vector at allocation time and removing redundant accesses to data structures, are shown in Figure 3. We group similar anti-patterns together into categories. Table 1 shows several of the anti-pattern categories in our database.

Using this anti-pattern database, we identify candidate locations in the codebase for similar optimizations. This problem is challenging, as exact matching fails to surface these locations due to differences in variable names and code structure. We find this problem is well-suited to machine learning and develop a code similarity search technique based on vector similarity [15] to identify a set of *candidates* similar to a given anti-pattern. To focus this search, we leverage fleet-wide con-

[1] Throughout the paper, we rename sensitive variables, names, or values.

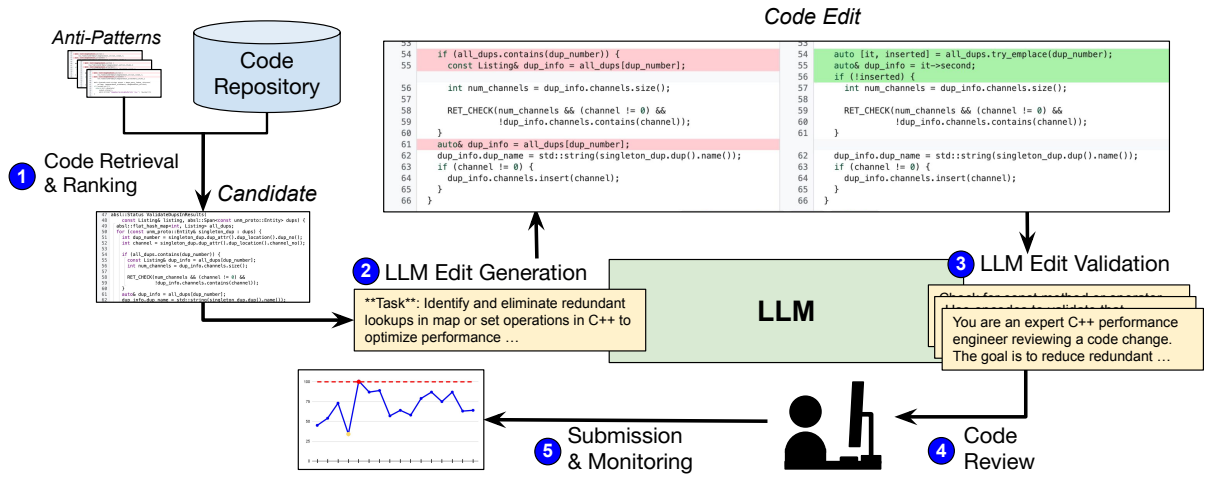


Figure 1: An end-to-end example of ECO, demonstrating the cycle from anti-pattern mining to deployed fix.

Table 1: Anti-Pattern Categories

Category	Description	Count
Alloc	Unnecessary allocations: Allocation of a new object when reuse of an object is possible.	8
Args	Unnecessary copied args: Using an API causing unnecessary copies of arguments from a widely used internal framework.	2501
Copy	Unnecessary copies: Replacing variables with references.	2873
Map	Redundant map operations: See Figure 3a.	985
Move	Missing `std::move`s: The last use of an object is copied instead of moved.	1487
Sort	Unnecessary sorts: Use of a sorted or stable map/set when not required.	91525
Vector	Missing vector reserves: See Figure 3b.	331

tinuous profiling to rank candidates based on resource consumption, identifying the best opportunities.

Subsequently, we use a fine-tuned LLM to generate code that implements the optimization. To mitigate the risk of LLM hallucinations, we improve ECO's reliability by using diverse prompting strategies and a multi-stage verification pipeline. To ensure correctness, we use the LLM to self-review its generated code, similar to [21], run automated tests, and perform checks tailored to each project. We then submit the generated changes for human code review. After submission, we monitor the change in production to detect regressions and measure its performance impact.

ECO's optimization discovery approach results in a database of anti-patterns that can grow over time. Figure 2 outlines the end-to-end process from an operational point-of-view. A small team of engineers operates ECO, beginning by selecting anti-patterns to target. The team ranks known anti-patterns according to observed frequency and savings potential, supported by the data and statistics gathered in the creation of the database. After selecting anti-patterns, they launch ECO to automatically identify optimization candidates from existing examples. The team ranks these candidates (described in Section 4), and uses a library of prompt recipes (evaluated in Section 7) to have ECO automatically apply an optimization across all identified locations, submitting the resulting commits for code review.

Actions by ECO are automated, with engineers directing the system and intervening only when necessary. Most human involvement appears during the review and submission phase. However, this is required for any change generated at Google, regardless of whether or not it is AI-generated. Code changes fail if performance regressions appear in production and they need to be rolled back. ECO's validation process has kept the rate of rollbacks extremely low (Section 7.4). As such, ECO greatly scales up human productivity compared to manually searching for optimization opportunities, writing patches and shepherding them through code review.

3 Performance Optimization Datasets

We now describe how we mine historical commits to create a dataset of efficiency anti-patterns and optimizations.

3.1 A Dataset of Efficiency Anti-Patterns

To enable ECO to drive interpretable and consistent changes, we begin by identifying and categorizing performance anti-patterns from historical data. Google's codebase contains billions of lines of code. It is structured as a monorepo (similar to [22–25]), but the approach would be the same if it was spread across smaller, disparate GitHub repos. While this work targets C++, ECO is not fundamentally limited to it. Despite being organized as a monorepo, this setup covers over 100,000 packages and a wide variety of projects [20], including mirrors of open-source repositories [26]. This diversity

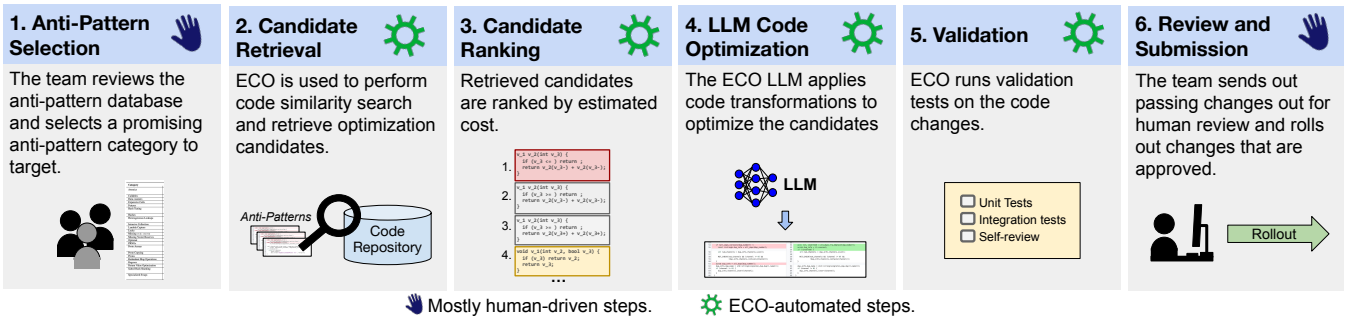


Figure 2: Operational timeline of ECO.

and scale demonstrates ECO's scalability and generalizability across many different kinds of projects.

All code in the repository is searchable, with a complete history of all past commits, including messages, code review comments, and code changes. This is the same setup that a typical organization or open source project building on top of GitHub would encounter. It lets us implement textual search techniques to find historical commits that improved performance. We use two search types: First, we search commit messages for keywords indicating performance improvements (e.g., "speed up", benchmark results), performance-related hashtags, and links to performance-related documentation. Second, we search curated sources like internal newsletters that publicize efficiency improvements. This process is guided by human experts who curate the anti-pattern categories, define relevant search keywords, and set up the data processing pipelines. While our coding culture encourages single-issue commits, ECO is robust to commits with multiple changes.

We run this data extraction on a large-scale distributed computation framework, akin to Apache Beam [27], and maintain a database of these performance-related optimizations and the anti-patterns that they solved. We then use it as the foundation of our optimization approach. Appendix A provides a full list of anti-patterns in our dataset.

3.2 LLM Fine-Tuning Datasets

In addition to using our anti-pattern dataset to identify optimization candidates (Section 4), we also combine it with other data sources to fine-tune an LLM to improve its performance on these specific code refactorings [28]. For the examples and evaluation provided in this paper, we use ECO with Gemini Pro 1.0 [29], which supports fine-tuning via Google's Vertex AI service [30]. Our fine-tuning dataset is highly proprietary and includes source code, selected code commits, metadata of code and commits, and the performance-specific dataset derived from our anti-pattern database ($\approx$ 55k commits). The fine-tuning mechanisms are similar to prior work [31, 32].

While we describe a particular LLM here, ECO's architecture is agnostic to the specific model. We have regularly updated our model and continued to use ECO with more recently released LLMs. However, for consistency, all evaluations in this paper were performed with the original Gemini Pro 1.0 model described above. We note that it is not the goal of this paper to evaluate the capabilities of LLMs at producing code efficiency optimizations, and models continue to rapidly improve in this area, which ECO directly benefits from.

4 Optimization Opportunity Localization

While our anti-patterns dataset contains examples of *past* optimizations, a key challenge is identifying *new* candidates across our codebase. Naively applying an LLM to all of Google's code is computationally expensive with an observed high false-positive rate. In addition, incremental, localized changes are favored to minimize global disruption and facilitate testing and review processes. Therefore, we introduce a search technique based on vector similarity search [15] to identify new potential instances of anti-patterns.

4.1 Performance IR

To search for optimization opportunities, we first design an intermediate representation (IR) of functions across our codebase. We operate at the function level, as functions provide a stable and practical unit of code to track and compare across the millions of daily changes in our repository. This granularity strikes a balance between providing sufficient context for optimization and maintaining tractability. Using the Apache Beam framework [27], we crawl C++ code, parse each file with Clang, and divide it into functions. We annotate each function with its AST-derived types (e.g., of arguments and local variables), which are good indicators for certain optimizations. Figure 4 shows an example. To avoid processing millions of functions that are not performance-critical, we filter this set to focus only on *costly* functions.

4.1.1 Continuous Profiling

To identify costly functions, we leverage *Google Wide Profiling* (GWP), a fleet-wide continuous profiling framework that

```
- bool new_entry = false;
- if (s.current_shard.find(name) == s.current_shard.end()) {
-   s.current_shard[name] = 1;
-   new_entry = true;
- } else {
-   ++s.current_shard[name];
- }
+ int64_t value = ++s.current_shard[name];
+ bool new_entry = (value == 1);
```

(a) **Redundant Map Operations**: Performing duplicated key lookups and reinitializing map values can waste CPU cycles. Here, the LLM had to detect that `[]` may add a missing key, and that for a map with integer values, it gets initialized to 0; the optimization only works in that case and is not sound in general.

```
+ result.reserve(reader->NumRecords());
  for (;;) {
    Info info;
    bool has_msg = reader->ReadToMessage(&info).ValueOrDie();
    if (has_msg) {
      result.push_back(info);
    } else {
      break;
    }
  }
  return result;
```

```
+ error_list.reserve(3);
  int total_error_count = 0;
  for (int i = 0; i < vec1.size(); ++i) {
    const double diff = fabs(vec1[i] - vec2[i]);
    if (diff > abs_error) {
      total_error_count++;
      if (error_list.size() < 3) {
        error_list.push_back(Error(i, diff));
      }
    }
  }
```

(b) **Missing Vector Reserves**: Vectors are a common data structure but can cause unnecessary memory reallocations that pre-allocating space can avoid. In the first case, `reader` is a file API and the LLM had to determine that it can query the full number of entries with `NumRecords`. It also needed the domain-specific context that `has_msg` is rarely false. In the second example, it had to conclude that the list will only have up to 3 entries.

Figure 3: Examples of ECO performance optimizations.

periodically samples performance metrics from all running applications [33]. For each sample, it collects the program counter, stack trace, and performance counters. These profiles are aggregated into a global database. We use this data to identify costly functions as optimization targets. In addition, correlating it with code versions allows us to evaluate changes in resource usage linked to specific commits. This helps us both identify past optimizations and also measure the impact of ECO's commits after deployment.

4.1.2 Code Annotations with Performance Metrics

To find good optimization candidates, we annotate our IR with performance metrics such as CPU usage, memory allocations, and LLC misses from our profiler. A naive attribution of CPU usage is often unhelpful, as cycles are frequently spent in common, *shared library functions* (e.g., `vector<T>::push_back`) rather than *application-specific code*. These shared libraries are not generally profitable targets for optimization at ECO's level, as they are usually already heavily optimized. Therefore, we re-attribute resource usage to the parent functions in the call stack that are most meaningful for optimization.

```
{"definition":
  "string GetTimeLabel(time_t timestamp) {
     int year, month;
     getYearMonth(timestamp, &year, &month);
     string label;
     bool success = toLabel(year, &label);
     if (!success) return getError(year, month);
     return label;
  }",
...
"variables": [
  { "name": "year", "type": "int" },
  { "name": "month", "type": "int" },
  { "name": "label", "type": "string" },
  { "name": "success", "type": "bool" }
],
"performance": [
  { "instructions": 43052 },
  { "cycles": 109572 },
  { "llc_misses": 501 }
]}
```

Figure 4: A performance-annotated function in the dataset.

Algorithm 1: Retrieving costly functions.

Input: A function F and its parent function P
Output: Costly functions under F's call tree

Function `GetCostlyFns`(F, P)
  if `GetCallees`(F) *is empty* **then**
    if `ShouldPrune`(F, P) **then return** $[F]$;
    return $[F]$;
  costly_fns ← [];
  for callee *in* `GetCallees`(F) **do**
    extend costly_fns with `GetCostlyFns`(callee, F) ;
  end
  if costly_fns *is empty and not* `ShouldPrune`(F, P) **then return** $[F]$;
  return costly_fns;

Function `ShouldPrune`(F, P)
  if `getCyclesOfTree`$(P) > C_{max}$ **then return** *false*;
  if `isSharedFn`(F) **then return** *true* ;
  if `getCyclesOfTree`$(F) < C_{min}$ *or* `getCyclesOfTree`$(F) > C_{max}$ **then return** *true* ;
  return *false*

4.1.3 Identifying Costly Functions

We take per-binary function call trees from profiler samples and apply a pruning and attribution step, outlined in Algorithm 1 and depicted in Figure 5. We traverse the call trees to find the lowest-node *application-specific functions* that are responsible for significant cost. We define application-specific functions as those common to fewer than a threshold of binaries. We recursively prune sub-trees and attribute their cost to their caller. Note that this has no effect on the caller's code representation and only affects its accounted resource consumption for the purpose of identifying costly functions. A sub-tree of shared functions is pruned and its cost attributed

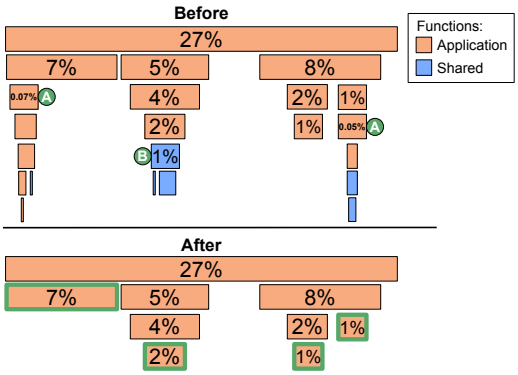


Figure 5: Pruning function call graphs (annotated by percent of the application's total cycles and represented as a flame graph [34], not drawn to scale). Unlabeled functions consume less than 0.01% of the application's cycles. Orange represents application-specific code, blue represents shared code. **A** labels pruned sub-trees roots that form less than C_{min}=0.1% of the app. **B** labels the root of a pruned shared code sub-tree. Boxes outlined green after pruning (bottom) are the leaf nodes of the pruned call graph targeted for optimization.

upwards, unless the caller is already too large (exceeding C_{max} of total binary cycles). Application-specific sub-trees are also pruned upwards until a caller accounts for at least C_{min} and at most C_{max} of the binary's cycles. The upper threshold prevents aggregation at too high a level (e.g., a binary's main function), which is unproductive for localized optimizations. Through empirical tuning, we set C_{min} to 0.1% and C_{max} to 25%. This analysis yields over 10M potential optimization candidates, which form our *search space*.

4.2 Code Retrieval

Given our anti-pattern dictionary and a search space of costly functions, we need to find all candidates that likely match a particular anti-pattern. This is challenging, as instances of an anti-pattern can be textually and structurally diverse (e.g., Figure 3a). Manual heuristics like regular expressions or AST-based techniques such as CodeQL [35] cannot usually capture this diversity. We therefore use a code retrieval technique that discovers new instances of an anti-pattern from past examples.

We first form a database of vector representations (embeddings) of each function in our search space (Figure 6). We then take examples from our anti-patterns database and look for similar functions in our search space using approximate nearest neighbor (ANN) search, a method proven effective in other code recommendation contexts [36]. We use ScaNN [15], an open-source vector similarity search library developed at Google, for this task. Given a vector representation of a *query* (the "before" function state or a code diff embodying an anti-pattern), ScaNN returns an approximate set of similar vectors from the database. To focus on high-impact changes, we only search within the set of costly functions identified previously. For each query, we retrieve the top-$\mathcal{K}$ ($\mathcal{K}=500$) candidates. We then normalize the query and candidates by removing custom names, static strings, and comments, and rank the candidates by a *syntactic similarity* score (Section 4.2.2).

4.2.1 Embedding Design for Semantic Similarity

We explore two strategies for vector embeddings:

(a) Bag of Words (BOW). This approach involves creating embeddings based on a BOW representation. Our intermediate representations provide textual tokens for each function after parsing. The embedding vector of a function is created by forming a token-frequency vector, excluding comments, common keywords (e.g., `return`), and punctuation tokens.

(b) Deep Embedding Models. We use a 1B-parameter dual-encoder *deep text embedding* model [37], originally trained on English query-document pairs, and fine-tune it on 25M samples of: (1) semantically similar function pairs and (2) code diffs and their corresponding "before" function states. Fine-tuning on similar functions allows us to query for similar code snippets. Fine-tuning on diffs allows us to search for specific optimization patterns, focusing the query on the relevant lines of code rather than the entire function. This matches how we later use the deep embedding model (querying it using both code snippets and code diffs) and is similar to how others use diffs or Git commits for training models [38–40].

4.2.2 Ranking Performance Opportunities

After retrieving the approximate top candidates through ANN, we use a *syntactic similarity score* for further, more detailed ranking. The score $\mathcal{S}$ between a query Q and a candidate C is defined as (higher is better):

$$\mathcal{S}(Q,C) = \frac{1}{4}[B(Q,C)+R(Q,C)+T(Q,C)+F(Q,C)]$$

Each component of $\mathcal{S}$ is from 0 to 1 and defined as follows:

- $B(Q,C), R(Q,C)$: BLEU [41] and ROUGE-L [42] score are text similarity metrics. B and R are calculated after normalizing the code (Q and C) to remove custom names, static strings, and comments.
- $T(Q,C) = (|T_Q \cap T_C|)/max(|T_Q \cup T_C|, 1)$, where T_Q, T_C are types present in Q and C from our code representation (Section 4.1.2).
- $F(Q,C)$: Cosine similarity of control flow-related keywords (*e.g.*, `for` , `while`) in the functions' BOW vectors.

The inclusion of B and R captures textual similarity, while T and F incorporate information about types and control flow, improving the representation of semantic similarity and helping further refine the set of candidates, after the approximate match provided by ANN.

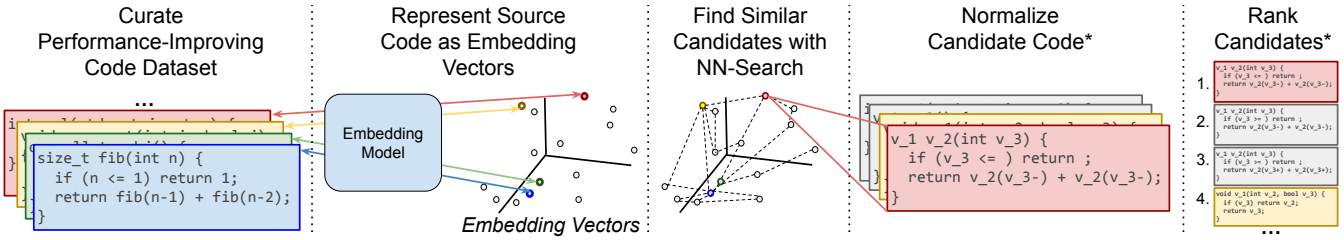


Figure 6: ECO's code retrieval approach. Source code is represented as vectors and ScaNN [15] is used to find nearest neighbors in the vector space. After retrieving nearest neighbors, we optionally rank code by similarity; these steps are marked by (*).

5 Code Edit Generation Strategies

After identifying candidates, we generate code transformations by prompting an LLM (Section 3.2). Since generation quality is highly dependent on the prompt, we explore a variety of prompting strategies, or *prompting recipes*, for engineers using ECO to build upon [43, 44].

LLM prompting is a well-studied area [45]. The simplest approach, *zero-shot prompting*, directly instructs the model to perform a task without giving it any examples. This works for simple edits but struggles with complexity. Since token generation is when the model applies logic, zero-shot prompting leaves few steps for the model to apply complex operations. It also relies on the LLM correctly interpreting the instructions.

Several techniques address this. *Few-shot* prompting includes examples of the desired input-output behavior (e.g., before/after code pairs) in the prompt to help the model with pattern matching, a technique shown to be effective for code [46]. *Chain-of-Thought* (CoT) prompting instructs the model to first output a plan or reasoning steps before generating the final output [43]. This encourages more structured reasoning and works well for code transformation [47]. Finally, *agentic* ReAct-style prompting enables multi-step reasoning by having the model cycle through generating a thought, an action (e.g., a command to execute in an external environment such as, in our case, a shell), and then feeding the output of the command back into the model, allowing it to interactively refine its work [44]. The prompting recipes below apply these paradigms to ECO's code editing task.

5.1 Prompting Recipes

Zero-Shot prompting. Our prompts follow the format in Figure 7a. The LLM response is a code diff compatible with the `patch` command. To improve performance, our fine-tuning data (Section 3.2) includes examples in a similar instruction-following format.

Few-shot prompting. This variant uses a small set of example edits in the prompt to guide the model, as shown in Figure 7b. We select examples of a target optimization type from our database and prompt the model to perform a similar transformation on the target code.

Chain-of-Thought (CoT) prompting. This style elicits intermediate "thoughts" from the model to trace its reasoning [43]. By asking the model to provide ideas for improving code "*step by step*" (Figure 7c), we guide it to identify and then implement a specific optimization strategy.

ReAct prompting. This combines reasoning with action through an iterative loop of thought, action, and observation [44] until a goal is met (Figure 7d). This mimics how humans break down complex tasks. For code editing, this allows the model to examine code, propose a patch, and observe the result, providing a way to navigate ambiguity.

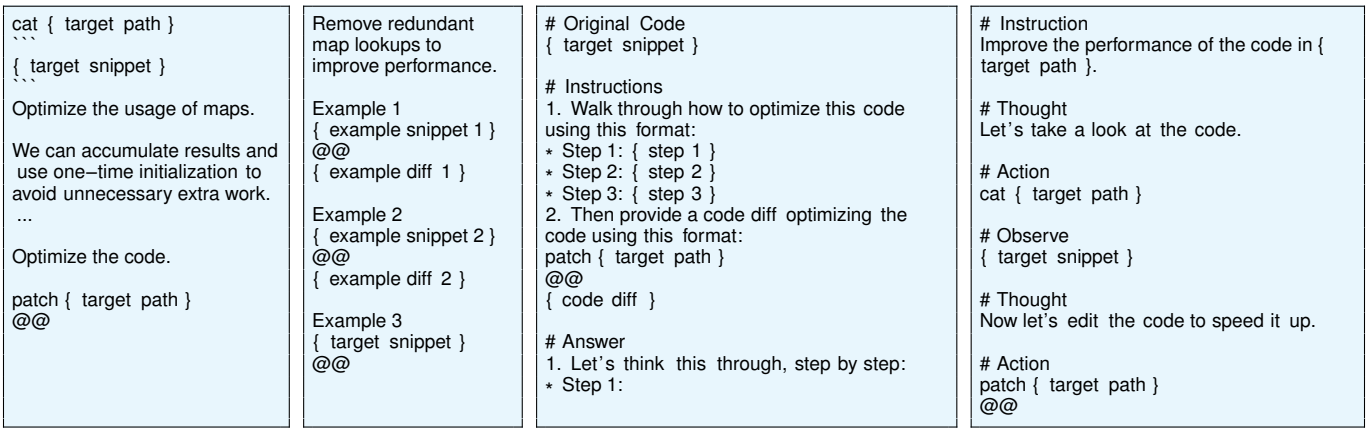
As we show in our evaluation, all four approaches are effective for certain tasks, but no single recipe is universally superior. We therefore maintain a library of these prompting strategies, and engineers select the most appropriate one on a case-by-case basis. Appendix B provides concrete examples of these recipes and their corresponding LLM outputs.

6 Verification, Submission, & Monitoring

Code generated by ECO must meet production standards and achieve a high success rate to minimize the burden on human reviewers. Our edit verification process starts with automated testing. We leverage Google's continuous integration pipelines, which run unit and integration tests, to check the correctness of ECO's edits. We automatically identify and run all tests in our entire monorepo that are affected by a change. If a change causes a build or test failure, we first attempt automated fixes for trivial errors like missing include directives. For more complex errors, we use our fine-tuned LLM to attempt a fix. If the code still fails, the edit is abandoned.

Due to imperfect test coverage, passing automated tests is not a guarantee of correctness. Furthermore, most tests do not evaluate performance and cannot detect neutral or negative efficiency impacts. We therefore incorporate LLM self-review to further improve edit quality. We prompt the model with a checklist of questions about common coding errors, anti-patterns, and optimization goals (see Appendix D). An edit proceeds to human review only after it passes this self-review step. Following Google's policies, all changes are reviewed by code owners, who make the final decision.

Management of these commits is automated using a tool



(a) Zero-shot (b) Few-shot (c) Chain-of-Thought (d) Agentic (ReAct)

Figure 7: Templates for different prompting strategies used by ECO.

called Rosie [48]. Rosie allows ECO to generate a large number of commits (called a *Large-Scale Change* or *LSC* in Google parlance) and then automatically identifies the correct code owners and requests reviews from them. As with any code change at Google, the commits must be reviewed and approved by code owners before they can be submitted. Google engineers are used to these kinds of reviews and perform them carefully [20,49]. This review load is accepted and generally fades into the background. ECO is no different.

A range of Google tools monitor changes in production and automatically roll them back if they cause regressions [50]. After a change is committed, we use our GWP continuous profiler (Section 4.1.1) to monitor its impact. This is non-trivial, since different projects have very different deployment strategies, and services usually roll out gradually over time. We therefore correlate the uptake of a particular commit in binaries measured by GWP with changes in metrics. We track key performance metrics to detect regressions and use monitoring alerts on performance-sensitive functions to notify us of any issues, allowing for prompt investigation. If a change correlates with a performance regression, it can be reverted. This is rare, occurring for <0.5% of ECO's commits (Section 7).

```
std::string CreateSimpleNames(
    absl::flat_hash_map<std::string, Loc> map,
    std::string key) {
  Loc value = map.at(key);
  if (value.locations.empty()) {
    return "simple_name";
  }
  return SimpleStrName(value.locations[0]);
}
```

(a) Original code snippet with an unnecessary copy anti-pattern.

```
     std::string key) {
-  Loc value = map.at(key);
+  const Loc& value = map.at(key);
   if (value.locations.empty()) {
```

(b) Sample of a successful model generated change.

```
 std::string CreateSimpleNames(
-    absl::flat_hash_map<std::string, Loc> map,
-    std::string key) {
+    const absl::flat_hash_map<absl::string_view,
+    Loc>& map,
+    absl::string_view key) {
   Loc value = map.at(key);
```

(c) Sample of a rejected code diff. The proposed type change requires updating the file includes but this was not added by the LLM.

Figure 8: Code optimization samples generated by ECO.

7 Evaluation

We evaluate ECO through measurements from its deployment within our production fleet, and controlled experiments using examples from a small subset of our anti-patterns. We note that while we characterize ECO's code generation abilities, the goal is not to compare LLM generation quality across models (which has been the focus of most prior work [1–10]). Instead, we show how ECO transforms a particular level of carefully characterized LLM capabilities into production impact as measured in end-to-end optimizations landing in production. As LLM capabilities are rapidly improving over time, ECO can directly take advantages of these improvements.

7.1 Microbenchmark Evaluation

We use controlled microbenchmarks to evaluate prompting strategies under conditions not feasible in production, such as testing multiple alternative edits for the same code. These experiments run on dedicated, isolated machines with many repetitions to ensure low-noise measurements. We use three hand-written C++ benchmarks, covering the *Copy*, *Map* and *Vector* anti-patterns in Table 1, and test with zero-shot, few-shot, CoT, and ReAct prompts. Each prompt asks the model to produce *code diffs* to optimize the benchmark (Figure 8). For each benchmark and prompting strategy, we sample the model five times and track the number of lines modified and

Table 2: Microbenchmark edits. We sample the model five times for each prompt type: zero-shot (ZS), few-shot (FS), chain-of-thought (CoT), and ReAct. CodeBLEU scores and speedup compare against unoptimized baselines. We report the number of avg. modified lines (ModLn); avg. generated valid/invalid code diff hunks (ValEd/InvEd); avg. rejected code diffs that broke code (Rej); avg. CodeBLEU score against baselines (CodeBL); Min, Med, and Max speedups; and the speedup of the highest CodeBLEU-scoring sample (CBS). If a generated edit is invalid, we leave the code unchanged when measuring speedup (leading to a speedup of 1).

	Copy	ZS	FS	CoT	ReAct	Human	**Map**	ZS	FS	CoT	ReAct	Human	**Vector**	ZS	FS	CoT	ReAct	Human
Edit Metrics	ModLn	2.80	5.60	15.80	8.00	6.00	ModLn	6.60	7.60	23.60	9.80	23.00	ModLn	2.60	8.60	63.40	20.40	3.00
	ValEd	1.20	1.00	2.40	1.40	1.00	ValEd	1.00	1.00	2.80	1.40	1.00	ValEd	1.00	1.00	1.60	1.00	1.00
	InvEd	0.00	0.20	0.40	0.00	0.00	InvEd	0.00	0.00	0.20	0.00	0.00	InvEd	0.00	0.00	0.20	0.00	0.00
	Rej	0.00	0.00	0.40	0.00	0.00	Rej	0.00	0.00	0.60	0.00	0.00	Rej	0.00	0.00	0.00	0.00	0.00
	CodBL	0.97	0.90	0.85	0.93	0.86	CodBL	0.83	0.83	0.76	0.84	0.73	CodBL	0.96	0.94	0.80	0.82	0.95
Speedup	Min	0.99	0.99	0.07	0.05	3.59	Min	1.05	1.05	1.00	1.04	1.22	Min	1.20	0.03	0.08	0.03	1.82
	Med	1.01	1.00	1.00	1.01	3.59	Med	1.05	1.06	1.00	1.06	1.22	Med	1.22	1.21	0.99	1.00	1.82
	Max	1.01	1.01	1.26	5.90	3.59	Max	1.07	1.06	1.07	1.06	1.22	Max	1.22	1.24	1.22	1.35	1.82
	CBS	1.01	1.01	1.26	1.01	3.59	CBS	1.05	1.05	1.05	1.06	1.22	CBS	1.22	1.21	0.99	1.00	1.82

whether the edits are valid (i.e., the diff hunks apply cleanly).

To assess the impact of different prompting techniques, we want to obtain a diverse set of model outputs with a given prompt. However, the higher the variance of the output, the less predictable the approach. LLMs configure this trade-off through a parameter called temperature. Its value is usually between 0 and 1, where 0 is most predictable. We pick a temperature of 0.3, which we find strikes a good balance.

In all microbenchmarks, we compare against a human baseline where an expert hand-optimizes the code. Since these benchmarks are specifically written for this evaluation, none of them are in the training set of our model. One challenge in a real-world setting is that we do not know which edit performs best until we deploy it. Since it is not generally feasible to try multiple candidate edits in production, we need a way to pick a good edit prior to deploying or measuring it. This impacts the efficacy of an approach. For example, a prompting approach that occasionally provides a better optimization than other strategies does not help us if we cannot identify this optimization among its outputs. We thus need a reliable approach to pick an output to use.

To pick an output, we compare each edit with the *unoptimized* baseline microbenchmarks using CodeBLEU scores which measure syntactic and semantic similarity using code-based n-gram matching [51]. The lower the score the more an edit changed the baseline's code. While this does not directly indicate the quality of an edit, it can be used to pick the output that represents the most *conservative* (i.e., highest) of the generated edits, which we will see is a good strategy.

If a proposed code edit results in a benchmark that still builds, we accept the edit and mark it as valid. If it breaks the build, the edit is marked as invalid. Figure 8 shows examples of accepted and rejected microbenchmark code edits from ECO. After generating edits, we apply accepted edits to the benchmarks and measure the performance in cycles per operation across 10 runs. We calculate speedup as $\texttt{speedup} = \frac{\texttt{new speed}}{\texttt{baseline speed}}$. Since edits that result in build failures are rejected and result in unchanged code, they lead to a speedup of 1. We report the minimum (Min), median (Med), and maximum (Max) speedup from the five prompt samples, as well as the speedup from the most conservative edit (*i.e.,* highest CodeBLEU score). Table 2 shows the results.

The edit metrics reveal that different prompting techniques yield varying results. CoT, for example, generates the most line changes but also more invalid edits, suggesting it is better for exploration than for generating immediately submittable changes. ReAct prompting most closely mimics how programmers edit code, and achieves higher Max speedups, except in the case of the Map microbenchmark. This confirms that there is no single best prompting recipe; the choice depends on the specific optimization task.

Speedup statistics also indicate that in a third of the cases, the edits achieve maximum speedups from the most conservative edits (when Max equals CBS, the speedup of the highest CodeBLEU-scoring sample). In nearly all cases (except Map/FS), CBS meets or exceeds median speedup, highlighting that small, conservative changes can significantly improve efficiency. We thus see that conservatively picking candidates that perform a minimum amount of semantic changes to the original code selects edits that lead to good efficiency improvements. While humans still outperform the AI, superhuman capability is not required for ECO, since the key benefit of AI is its ability to cover much more code.

7.2 Candidate Retrieval

We now evaluate the effectiveness of ECO's candidate retrieval methods (Section 4.2). We create a test set comprising 63 pairs of code diffs and corresponding "*before*" functions. This set includes 21 pairs each for unnecessary copies, redundant map lookups, and missing vector reserves.

To assess retrieval performance, we build a database of

Table 3: Comparison of Bag of Words (BOW), Deep Text Embedding (DTE), and Deep Code Embedding (DCE), with and without ranking, for identifying optimization opportunities.

Model	Query	Ranked	MAP@5	MAP@10	MAP@20
BOW	Function	No	0.0287	0.0173	0.0128
BOW	Function	Yes	0.0728	0.0398	0.0224
DTE	Function	No	0.0259	0.0190	0.0137
DTE	Function	Yes	0.1633	0.0947	0.0554
DCE	Function	No	0.1945	0.1585	0.1111
DCE	Function	Yes	0.2036	0.1316	0.0918

Table 4: Prompting strategy edit metrics. Modified lines (ModLn), valid/invalid/exact edits (ValEd/InvEd/ExtEd), and CodeBLEU (CodBL) and quality scores (QualS) are shown as averages per prompting technique.

Prompt	ModLn	ValEd	InvEd	ExtEd	CodBL	QualS
ZS	9.392	1.752	0.128	0.428	0.967	0.531
FS	12.736	1.796	0.156	0.088	0.958	0.404
CoT	22.644	3.272	0.82	0.152	0.954	0.344
ReAct	10.444	1.804	0.1	0.264	0.969	0.510

1803 functions, including functions from the 63 test pairs and an additional 1740 functions without known opportunities for the targeted optimizations. For each of the 63 test functions, we use its code embedding to query the database. A successful retrieval model would rank functions of the same anti-pattern higher when queried with a function of that anti-pattern. We compare the retrieval models in terms of mean average precision (MAP), a standard measure for evaluating retrieval models [52]. A query and a function are deemed "*relevant*" if they share the same optimization opportunity.

Table 3 presents the performance of these retrieval models using different query types. MAP@K quantifies retrieval quality by considering the top-$\mathcal{K}$ results for each query. When $K = 5$, bag of words (BOW) only achieves low MAP@K scores even after functions are ranked by syntactic similarity scores (Section 4.2.2). The MAP scores drop even further when K increases to 10 and 20. In contrast, MAP@5 more than doubles from 0.0728 to 0.1633 with function rankings on the deep text embedding model (DTE) and 0.2036 with the deep code embedding model (DCE).

Deep text embeddings show significant improvements over BOW, despite their lack of explicit code training. This suggests that text embedding models inherently capture some code semantics. Deep code embeddings show further improvements over deep text embeddings, which is expected from models trained on semantically similar function pairs.

Finally, applying a second-step syntactic ranking boosts performance for the BOW and text embedding models, underscoring the value of syntactic comparison when a model lacks full code understanding. The deep code model, however, shows little benefit from this re-ranking, suggesting that its end-to-end training already captures the necessary semantic and syntactic information. These findings validate the effectiveness of our embedding-based approach for retrieving optimization candidates.

These findings show the efficacy of ECO's embedding-based approach to code optimization candidate retrieval.

7.3 Edit Generation on Production Code

While microbenchmarks allow easy measurement of performance improvements, they do not capture the complexity of a large-scale production code base. We now assess the quality of generated edits on production code. We focus on the four prompting strategies applied to 48 performance-improving commits from our real code repository (not used to fine-tune our LLM). We aim to determine how accurately each prompt replicates the chosen code edits.

Contrary to the microbenchmark study, we cannot perform isolated performance tests on each generated edit. We thus evaluate these edits along multiple dimensions, each of which captures a different aspect of correctness. First, *validity* captures whether the code compiles (i.e., is syntactically correct). Second, *exact match* to the original edit is a very strong signal for correctness when there is a match, but provides little signal otherwise. Third, *CodeBLEU* scoring with respect to the original edit provides a signal how close the edit is syntactically, but small syntactic changes can make a large difference for correctness (e.g., including an additional `&` or `*`). To capture these subtleties, we also obtain a *quality score* for each edit through human scoring. While work-intensive, the human scoring is critical to provide ground truth, as automated scoring strategies can only provide limited insights in this evaluation setting. This is in line with best practices [53].

Given a target performance-improving commit, we use zero-shot, few-shot, CoT, and ReAct prompts to provide the model with the original source code and instructions to optimize the code. For each prompt, we sample our model five times. Each generated edit is then evaluated using CodeBLEU scores and by humans. Figure 9 breaks down the CodeBLEU and human-reviewed code quality scores for each commit, while Table 4 shows averages across the generated edits.

Zero-shot and ReAct prompting achieve the highest CodeBLEU and human-scored quality scores. In contrast, CoT provides the lowest CodeBLEU and quality scores. Figure 9 indicates that CodeBLEU scores generally align with human scores, trending higher with better code quality scores. However, discrepancies arise when the model proposes minor, non-performance-improving changes (*e.g.*, altering variable names or formatting — commits 47 and 48) in which the edit receives a high CodeBLEU score but a low quality score. On the other hand, if the model generates performance-improving

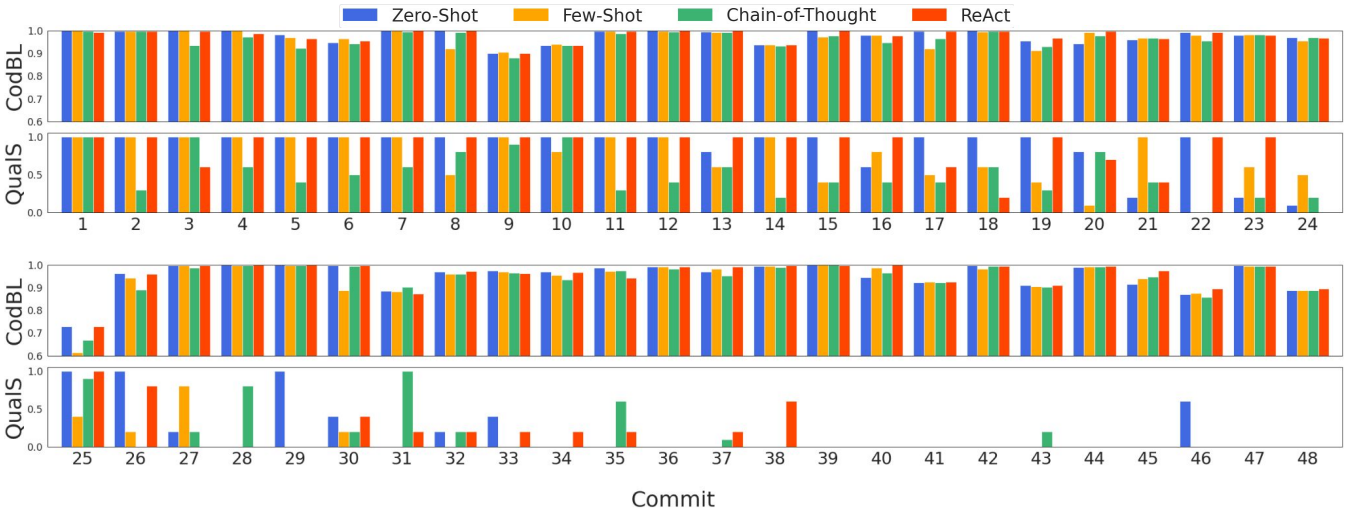


Figure 9: Quality of generated edits by prompting strategy. CodeBLEU scores (CodBL) show comparison to true historical changes. Quality scores (QualS) show code edit quality as scored by human experts reviewing whether the code edit is correct and performance-improving (Rubric: 1 = Yes, 0.5 = Partially, 0 = No). In total, human raters scored 960 examples.

Table 5: Feedback from human and automated code-review.

	Copy	Map	Vector	Description
S_PROD	39.97%	4.86%	40.89%	Submitted to production.
S_USER	6.55%	10.49%	15.99%	Submitted with user discussion (e.g., reviewer questions or suggested changes).
R_REVERT	0.16%	0.14%	0.12%	Submitted to production but later rolled back due to regressions.
R_TEST	16.14%	37.39%	20.07%	Rejected during the validation phase.
R_USER	35.81%	46.55%	22.87%	Rejected by human reviewers.
R_EMPTY	0.00%	0.07%	0.00%	Rejected due to failure to make code modifications beyond formatting.
R_OTHER	1.37%	0.49%	0.05%	Rejected for other reasons, typically non-edit related, such as failure to resolve merge conflicts during submission or failure to identify proper reviewers.
Total Commits	4959	1421	4035	Total number of code commits that were generated.
NC Savings	331k	125k	618k	Efficiency impact of the changes in normalized cores.

changes but also adds extensive documentation (*e.g.*, commit 25), the edit receives a low CodeBLEU score but high quality score. These results suggest that CodeBLEU can provide a weak signal for performance quality when comparing against known performance-improving edits. This study also reveals that insights from microbenchmarks hold true for production edits. CoT prompts produce larger edits but do not always result in the highest quality while targeted approaches like ReAct prompting tend to be more accurate.

7.4 Production Impact Evaluation

We now analyze ECO's production impact and its interactions with Google's infrastructure and processes. ECO has generated thousands of edits to date. We measure compute savings in *normalized cores* (NC), defined as the MIPS provided by a single core on a specific hardware platform [54]. In production, our approach employs a mix of zero-shot, few-shot, CoT, and ReAct prompting strategies on a case-by-case basis. We focus once again on the `Copy`, `Map` and `Vector` anti-patterns.

Edit Quality We quantify the quality of ECO-generated edits at scale by analyzing their code reviews and deployment, which includes human feedback, reverts due to production issues or incorrect changes, and resultant performance improvements using our fleet-wide performance profiler. For every change generated by ECO, a human code owner is asked to review the change before the change is submitted to production. For 40%, 5%, and 41% of `Copy`, `Map` and `Vector` changes, respectively, ECO-generated changes were directly approved and submitted to production without the human reviewers leaving any feedback that had to be resolved.

Table 5 shows a breakdown of additional actions required (see Appendix E for concrete reviewer interactions). Oftentimes, users left comments that were extraneous to the optimization in the code change (`S_USER`). These comments had to be addressed before the code change was submitted. Other times, users rejected changes (`R_USER`) and the changes were not submitted. This could be due to the reviewer judging the change to not be performance enhancing or wanting the code to be untouched for other reasons (e.g., style preferences).

Failure Modes The reasons why optimizations failed were multi-faceted. For example, `Map` has a comparatively low success rate. While `Copy` and `Vector` changes are relatively localized, `Map` changes typically require restructuring and reasoning about large pieces of code while also dealing with subtle semantics of C++ maps. Examples of errors include:

- **Control Flow**: Rewrites often caused subtle changes in behavior along some execution paths (e.g., adding an element too early or too late, confusing the behavior of the [] operator vs. `try_emplace`, `contains`, etc.).
- **Pointer Stability**: The model sometimes assumed that a pointer to a map entry remains valid throughout map modifications even if a `flat_hash_map` is used.
- **Corner Cases**: E.g., `if (!a.contains(k)) { a[k] = v } else { a[k] = min(v, a[k]) }` and `a[k] = min(v, a[k])` are not equivalent.
- **Not a Map**: Sometimes the model thought that something that was not a map was a map (e.g., a proto).
- **No Change**: The change was semantically equivalent or did not improve performance (e.g., replacing a single [] with a `try_emplace`).

Our automated pre-screening filters a significant portion of these before they reach human reviewers, reducing the review burden. Sometimes, we found that it was the reviewer who was mistaken. For example, we saw a case where the model added a `v.reserve(2)`, causing the reviewer to reject the change as erroneous. However, later in the code, there was an assertion `ASSERT_EQ(2, v.size())`, indicating that the change, while unexpected, would have been correct. In one case, a correct ECO code change optimizing a redundant map lookup was sent to a reviewer who suggested using `try_emplace` instead (Appendix E). The reviewer's suggestion was applied instead of the original ECO change. This led to large increases in CPU utilization and had to be reverted (`R_REVERT`). This highlights the subtleties of human-AI collaboration.

Resource Savings Figure 10 shows the savings achieved. Savings per line of code also vary by anti-pattern. Complex `Map` changes require more lines of code and yield lower savings per line than simple `Copy` or `Vector` changes. In contrast, `Vector` edits are often single-line changes but have a high impact, as they can prevent many reallocations of a large data structure.

Over a period of one year, ECO has sent commits covering many anti-patterns (Section 3), worth over several hundred thousand normalized CPU cores. Figure 10 shows the volume of code changes *successfully* submitted, with a rollback rate of less than 0.5%. In total, ECO has submitted over 25k lines of code across 6.4k commits. The majority of these changes modify or add lines, and some edits also remove lines. These results demonstrate the potential of ECO to achieve fleet-wide performance improvements with minimal human effort.

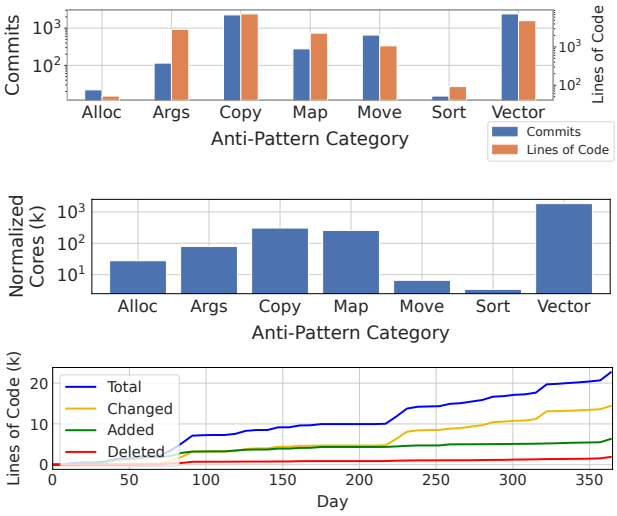


Figure 10: Statistics on landed code edits. Top: submitted code edits. Middle: performance savings. Bottom: landed code edits in production over time.

Cost Analysis An important question is whether the benefits of ECO justify its operational costs. Conservatively using prices for Gemini 3.1 Pro Preview, the most expensive Google model at the time of writing ($2-12 per 1M tokens [55]) and conservatively assuming 10k tokens (90% input) per invocation and 10 invocations per commit (including self-review and refinement), generating 10k commits costs $3,000.

Engineering and infrastructure costs for code review are not unique to ECO and similar to the background review load generated by other automated tools [20], which is generally accepted at Google. Overall, the additional human and infrastructure load for submitting changes from ECO was negligible. While we cannot share the exact number, it represented less than 0.1% of overall resources. This is a one-off cost, while the resource savings are realized on an ongoing basis.

8 Related Work

Code optimization Automatic code optimization has a long history. Stochastic superoptimization [56] explores instruction sequences to find optimal solutions, operating below the source code level. More recently, AlphaTensor [57] and AlphaDev [58] use reinforcement learning for algorithm discovery for specific problems. The Scalene profiler [3] interfaces with ChatGPT to optimize Python code. PIE [2] extends the CodeNet Benchmark [4] and designs evaluation methodologies and prompting techniques for performance improvements. Chen et al. [5] used VAE-based models on Google Code Jam data, while Supersonic [6] applies Seq2Seq models to Codeforces [59] and the CodeNet dataset [4].

Some works attempt to scale these methods beyond competition datasets. Aroma [36] provides general code recommen-

Table 6: Static Analysis vs. ECO vector reserve optimizations. Speedup ranges represent empirical CPU time reductions from micro-benchmarks (* denotes qualitative analysis where no runnable artifact was available).

Pattern	ClangTidy	CppCheck	CARAMEL*	CLARITY*	ECO Speedup	Explanation	Example Code Snippet

A: Standard Loop Insertion	Yes	No	Yes	Yes	0.4% – 7.4%	Trivial loop structure matching directly on standard library vector insertion (`.push_back`).	`dirs.reserve(src.size());`
							`for (auto g : src)`
							`  dirs.push_back(g);`

B: Bulk Move via Algorithms	No	No	No	No	2.1% – 13.8%	Elements are appended via algorithms/wrappers without an explicit local loop in the AST.	`res.reserve(res.size() + src.size());`
							`absl::c_move(src,`
							`  std::back_inserter(res));`

C: Custom Container Assignment	No	No	No	No	5.6% – 9.9%	Target is a protocol buffer; insertion bypasses standard vector analysis via implicit allocations in assignment operators.	`// RepeatedPtrField assignment`
							`cache.mutable_ents()->Reserve(size);`
							`*cache.mutable_ents() =`
							`  {src.begin(), src.end()};`

D: Template Range Copies	No	No	With LTO	Yes	1.4% – 4.1%	Container types are generic template arguments unresolved at static analysis compile-time.	`template <class R> void Copy(const R& s) {`
							`  std::vector<Ent> d;`
							`  d.reserve(s.size());`
							`  d.insert(d.end(), s.begin(), s.end());}`

E: Inter-Procedural Size Maps	No	No	Partial	Partial	3.0% – 13.7%	Sizing context is decoupled. Conditional insertions prevent mapping to `src.size()`; a pre-calculated stat dictates the true needed capacity.	`// st.total_size == count of ok items`
							`d.reserve(st.total_size);`
							`for (const auto& x : src)`
							`  if (x.ok) d.push_back(x);`

F: Custom Pointer Array Buffers	No	No	No	No	3.2% – 12.7%	Target is a custom API class. Static analysis lacks semantic awareness that `.Add()` stores pointers and triggers underlying array expansions.	`CustomPtrBuffer buf;`
							`buf.Reserve(src.size());`
							`for (auto& x : src)`
							`  buf.Add(&x);`

dations across a code base but does not focus on performance. Garg et al. [60] train models on C# performance-improving edits and evaluate on open-source repositories, but focus less on scalability and real-world deployments. Other work like EditLord [61] also fine-tunes LLMs on code edits, but this improves an LLM's capability at code editing as opposed to solving localization and reliability. To the best of our knowledge, ECO is the first to demonstrate and evaluate code optimizations at scale, applying them to billions of lines of code. Unlike prior work, we emphasize automation and the warehouse-scale setting, resulting in thousands of real-world performance improvements.

Code transformations Mining and applying code transformations in existing code is not a new problem. Meng et al. [62] build a sophisticated custom algorithm to generalize from an example and then apply the "same" transformation to other locations. More recently, Miltner et al. [63] present an algorithm to synthesize a script from repetitive changes and enable automated propagation of those changes. In our setting, examples of changes are already provided as anti-patterns, and we find that modern LLMs are capable enough to apply these changes at other locations, with no need for custom algorithms. Overall correctness must be verified, but this is also true of these prior AST-based techniques. Finally, Uber's DR.FIX system [64] also uses LLMs for edits, but applies them to data races in Go, a very different problem.

Repository-level refactoring Our work also relates to recent work in rolling out repository-level code changes, such as CodePlan [65]. While superficially similar, the primary commonality is that the actual transformation is carried out by LLMs. However, the emphasis of CodePlan is to apply a precisely specified edit that requires modifications to multiple files, such as changing a function signature or type, leveraging static analysis to identify additional locations that need to be changed. In contrast, our work focuses on *identifying* opportunities for code optimization across a repository of billions of lines of code, and *generalizing* existing edits to these new contexts. Our work is carried out at a much larger scale—CodePlan's largest repository had ≈ 20k lines of code.

9 Discussion & Future Work

Deployed SOTA Google relies on compilers for optimization, which cannot apply semantics-altering changes like ECO. We also deploy LLVM's Clang-Tidy [16], which uses heuristics to detect errors, including performance issues. However, we found it has notable limitations.

Static Analysis We compare ECO to static analysis tools. Table 6 summarizes vector reserve patterns—a primary anti-pattern category for both static tools and ECO—and whether tools can optimize them. We include Clang-Tidy [16] and CppCheck [66], two widely deployed static analysis tools. We also perform a qualitative comparison to CARAMEL [67] and CLARITY [68], two published static analysis techniques that target performance optimizations (but have no artifacts).

We group common code sequences that can be optimized with vector reserve operations into 6 patterns. For each pattern, we run microbenchmarks spanning 8 to 4,096 container elements and report the resulting speedups, confirming that these changes represent real and significant optimization opportunities. Out of the 6 patterns, Clang-Tidy reliably optimizes only 1, and CppCheck optimizes none, whereas ECO successfully optimizes all 6. The reason is that the first pattern is explicitly encoded in Clang-Tidy as the standard "*performance-inefficient-vector-operation*" [19] check, which identifies simple patterns such as `for (auto e : data) v.push_back(e);`.

In those cases, Clang-Tidy adds a call to `v.reserve`, but this only works if the control flow and structure are trivial.[2]

To determine if this is a shortcoming of specific tools or a fundamental static analysis limitation, we examine CARAMEL [67] and CLARITY [68], two static analysis tools that specifically target loop-related performance optimizations. Directly applying either tool would find none of the 6 patterns, since they are not the target of their analyses. CARAMEL [67] focuses on wasted loop iterations and CLARITY [68] identifies redundant traversal bugs, neither of which are present in the examples in Table 6. We hence only focus on the key ideas of these papers and try to adapt them.

Patterns B, C, and F are out of reach for static analysis since they rely on external functions and domain specific knowledge that are not available to the static analyzer. Pattern D requires templates to have been fully resolved by the time static analysis is applied; they may be within reach with LTO [69]. Pattern E may be within reach for either approach if the `x.ok` field matches one of the patterns it is searching for.

This points to a fundamental limitation of all static analysis tools – they can only identify the specific optimizations that they were designed for. Just like Clang-Tidy and CppCheck, CLARITY and CARAMEL only target specific optimizations. However, within this context, they have a key advantage: It is easier for them to perform inter-procedural analysis that spans disparate locations in the code, a task that used to be challenging for LLMs. Recent work proposed combining static analysis and LLMs to achieve the best of both worlds [70].

Expanding static analysis via custom rules for all corner cases requires prohibitive human engineering effort, quickly surpassing potential resource savings. CodeQL [35] facilitates this but does not solve it. In contrast, ECO is designed to expand beyond a specific optimization. and can perform diverse optimizations via examples, eliminating the need to implement new analysis passes for each optimization.

Deployment Insights One of the early challenges for ECO was to build trust with engineers. We thus prioritized our validation pipelines early on and picked initial edits carefully.

ECO's production success rate exceeded initial expectations, and reviewing changes from ECO helped shift some of our perspectives on the viability of optimizing code at Google with LLMs. Historically, LSCs were generated using deterministic tools, which restricted them to rigid, highly repetitive patterns. ECO helped us expand what we could optimize with automation. However, the increased diversity of commits meant that the team had to carefully monitor and iterate on the review feedback. This feedback loop was critical.

ECO depends heavily on the capabilities of the underlying model. Early on, we developed an evaluation to track the model capabilities on a set of canonical commits and understand what capabilities we could expect. We found it important to expand capabilities incrementally. Some optimization patterns were out of reach initially but later became feasible. E.g., optimizations such as proto arenas (Appendix C) require modifications across many files, which was not possible in early versions of ECO but became feasible over time.

While there are many prompting strategies, we found no clear rule on which works best when. Selecting strategies was highly empirical and sometimes combined techniques.

Some anti-pattern categories are much harder than they appear, e.g., `Sort`. Determining whether a sorted map or set can be replaced with an unsorted version requires judgement and significant context that early LLMs struggled at.

The embedding-based retrieval approach, when scaled to the scale of Google's multi-billion line code base, required careful calibration. There is a trade-off between precision and recall, and getting this balance wrong risks wasting significant reviewer time or missing important opportunities. We found that the most problematic candidates are those that seem plausible but are false positives.

As ECO has been successfully optimizing simple inefficiencies, the remaining opportunities have shifted toward more complex refactorings. These targets yield higher false-positive rates during candidate retrieval (e.g., heap-allocated protobufs that are candidates for arena migration can look semantically similar to code that is not, such as when an external pointer lifetime prohibits safe migration). To catch these false positives, we have begun deploying agent-based LLM validators during opportunity localization.

Future Work Our approach has several potential extensions. Currently, we use a *top-down* approach, leveraging an anti-pattern database to find optimization candidates. A complementary *bottom-up* approach could start from profiling data and then use an LLM to diagnose and fix bottlenecks. Further, our current *reactive* approach improves performance post-deployment. A similar framework could be used *proactively* to suggest edits to developers as they write code.

We believe ECO's core techniques can be replicated elsewhere using openly available tools and LLMs (Appendix F) and apply to smaller and decentralized codebases. We think the primary challenges to wider adoption are often organizational rather than technical, requiring integration with existing developer workflows and review processes.

10 Conclusion

We described ECO, a system for automatic code optimization at data center scale. By assembling a dataset of over 55k performance-improving edits from our code repository, we use code similarity search to identify similar patterns, and LLMs to apply changes automatically. Deployed in production at Google, ECO has changed more than 25k lines of code, saving hundreds of thousands of normalized CPU cores.

[2] Clang-Tidy documentation states that "*the check only detects [...] loops with a single statement body*" [19]. If the loop is more complex (e.g., a while loop or function calls), the pattern matching fails.

Acknowledgements

We would like to especially thank Chandu Thekkath, David Lo, Fredrik Kjolstad, James Laudon, Liqun Cheng, Mangpo Phothilimthana, Niranjan Tulpule, Steve Blackburn, Tipp Moseley and the anonymous reviewers for their feedback. We would also like thank our shepherd for their help in improving the final version of the paper.

References

[1] Jingzhi Gong, Vardan Voskanyan, Paul Brookes, Fan Wu, Wei Jie, Jie Xu, Rafail Giavrimis, Mike Basios, Leslie Kanthan, and Zheng Wang. Language models for code optimization: Survey, challenges and future directions, 2025.

[2] Alexander G Shypula, Aman Madaan, Yimeng Zeng, Uri Alon, Jacob R. Gardner, Yiming Yang, Milad Hashemi, Graham Neubig, Parthasarathy Ranganathan, Osbert Bastani, and Amir Yazdanbakhsh. Learning performance-improving code edits. In *ICLR*, 2024.

[3] Emery D. Berger, Sam Stern, and Juan Altmayer Pizzorno. Triangulating python performance issues with Scalene. In *OSDI*, 2023.

[4] Ruchir Puri, David S. Kung, Geert Janssen, Wei Zhang, Giacomo Domeniconi, Vladimir Zolotov, Julian Dolby, Jie Chen, Mihir Choudhury, Lindsey Decker, Veronika Thost, Luca Buratti, Saurabh Pujar, Shyam Ramji, Ulrich Finkler, Susan Malaika, and Frederick Reiss. Codenet: A large-scale ai for code dataset for learning a diversity of coding tasks, 2021.

[5] Binghong Chen, Daniel Tarlow, Kevin Swersky, Martin Maas, Pablo Heiber, Ashish Naik, Milad Hashemi, and Parthasarathy Ranganathan. Learning to improve code efficiency, 2022.

[6] Zimin Chen, Sen Fang, and Martin Monperrus. Supersonic: Learning to generate source code optimizations in c/c++, 2023.

[7] Dong Huang, Yuhao Qing, Weiyi Shang, Heming Cui, and Jie M. Zhang. Effibench: benchmarking the efficiency of automatically generated code. In *Proceedings of the 38th International Conference on Neural Information Processing Systems*, NeurIPS '24, Red Hook, NY, USA, 2024. Curran Associates Inc.

[8] Jiawei Liu, Songrun Xie, Junhao Wang, Yuxiang Wei, Yifeng Ding, and LINGMING ZHANG. Evaluating language models for efficient code generation. In *First Conference on Language Modeling*, 2024.

[9] Mingzhe Du, Luu Anh Tuan, Bin Ji, Qian Liu, and See-Kiong Ng. Mercury: A code efficiency benchmark for code large language models. *Advances in Neural Information Processing Systems*, 37, 2024.

[10] Dong Huang, Guangtao Zeng, Jianbo Dai, Meng Luo, Han Weng, Yuhao Qing, Heming Cui, Zhijiang Guo, and Jie Zhang. EffiCoder: Enhancing code generation in large language models through efficiency-aware fine-tuning. In Aarti Singh, Maryam Fazel, Daniel Hsu, Simon Lacoste-Julien, Felix Berkenkamp, Tegan Maharaj, Kiri Wagstaff, and Jerry Zhu, editors, *Proceedings of the 42nd International Conference on Machine Learning*, volume 267 of *Proceedings of Machine Learning Research*, pages 26058–26076. PMLR, 13–19 Jul 2025.

[11] Svilen Kanev, Juan Pablo Darago, Kim Hazelwood, Parthasarathy Ranganathan, Tipp Moseley, Gu-Yeon Wei, and David Brooks. Profiling a warehouse-scale computer. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture*, ISCA '15, page 158–169, New York, NY, USA, 2015. Association for Computing Machinery.

[12] F. Agakov, E. Bonilla, J. Cavazos, B. Franke, G. Fursin, M.F.P. O'Boyle, J. Thomson, M. Toussaint, and C.K.I. Williams. Using machine learning to focus iterative optimization. In *CGO*, 2006.

[13] Spandan Garg, Roshanak Zilouchian Moghaddam, Colin B. Clement, Neel Sundaresan, and Chen Wu. Deepdev-perf: a deep learning-based approach for improving software performance. In *ESEC/FSE*, 2022.

[14] Dong Young Yoon, Yang Wang, Miao Yu, Elvis Huang, Juan Ignacio Jones, Abhinay Kukkadapu, Osman Kocas, Jonathan Wiepert, Kapil Goenka, Sherry Chen, Yanjun Lin, Zhihui Huang, Jocelyn Kong, Michael Chow, and Chunqiang Tang. Fbdetect: Catching tiny performance regressions at hyperscale through in-production monitoring. In *Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles*, SOSP '24, page 522–540, New York, NY, USA, 2024. Association for Computing Machinery.

[15] Ruiqi Guo, Philip Sun, Erik Lindgren, Quan Geng, David Simcha, Felix Chern, and Sanjiv Kumar. Accelerating large-scale inference with anisotropic vector quantization. In *ICML*, 2020.

[16] Clang-tidy. https://clang.llvm.org/extra/clang-tidy/.

[17] Dominik Harmim, Vladimır Marcin, and Ondrej Pavela. Scalable static analysis using facebook infer. *I, VI-B*, 2019.

[18] Lucas Serrano, Van-Anh Nguyen, Ferdian Thung, Lingxiao Jiang, David Lo, Julia Lawall, and Gilles Muller. SPINFER: Inferring semantic patches for the linux kernel. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pages 235–248. USENIX Association, July 2020.

[19] Performance inefficient vector operation. https://clang.llvm.org/extra/clang-tidy/checks/performance/inefficient-vector-operation.html.

[20] Eric Christopher, Kevin Crossan, Wolff Dobson, Chris Kennelly, Drew Lewis, Kun Lin, Martin Maas, Parthasarathy Ranganathan, Emma Rapati, and Brian Yang. Instruction set migration at warehouse scale, 2025.

[21] Xinyun Chen, Maxwell Lin, Nathanael Schärli, and Denny Zhou. Teaching large language models to self-debug, 2023.

[22] Rachel Potvin and Josh Levenberg. Why google stores billions of lines of code in a single repository. *Commun. ACM*, 59(7):78–87, jun 2016.

[23] Brian Harry. Scaling git (and some back story). https://devblogs.microsoft.com/bharry/scaling-git-and-some-back-story/, 2017.

[24] Dorothy Ordogh. Pants and monorepos. https://typelevel.org/event/2018-03-summit-boston/, 2018.

[25] Zhongpeng Lin. Building uber's go monorepo with bazel. https://www.uber.com/blog/go-monorepo-bazel/, 2020.

[26] Google. Google open source: Third party. https://opensource.google/documentation/reference/thirdparty.

[27] Aizhamal Nurmamat kyzy, Aljoscha Krettek, Ahmet Altay, Ankur Goenka, Anton Kedin, Bruno Volpato, Charles Chen, Chad Dombrova, Chamikara Jayalath, Danny McCormick, David Cavazos, Davor Bonaci, Dan Halperin, Emily Ye, Frances Perry, Harshit Dwivedi, Heejong Lee, Henry Suryawirawan, Ismaël Mejía, James Malone, Jesse Anderson, John Casey, Julien Phalip, Jack R. McCluskey, Kiley Sok, Kenneth Knowles, Leonid Kuligin, Mark Liu, Mikhail Gryzykhin, Robert Bradshaw, Tyler Akidau, Thomas Groh, Thomas Weise, Eugene Kirpichov, Jean-Baptiste Onofré, Anand Iyer, Alexey Romanenko, Pablo Estrada, Rafael Fernández, Matthias Baetens, Reza Rokni, Tanay Tummalapalli, Udi Meiri, Boyuan Zhang, Rui Wang, Maximilian Michels, Ning Kang, Pedro Galvan, Rion Williams, Saavan Nanavati, Brian Hulette, Robert Burke, Valentyn Tymofieiev, Andrew Pilloud, Kyle Weaver, Daniel Oliviera, Robin Qiu, Mark Zeng, Yifan Zou, Artur Khanin, Ilya Kozyrev, Alex Kosolapov, Brittany Hermann, Svetak Sundhar, Israel Herraiz, Yichi Zhang, Danielle Syse, Ritesh Ghorse, Yi Hu, Pablo Rodriguez Defino, Namita Sharma, and Talat Uyarer. Apache beam: An advanced unified programming model. https://beam.apache.org, 2012.

[28] Aditya Kini, Satish Chandra, Milad Hashemi, Saksham Thakur, Aditya Pandey, Vincent Nguyen, Marc Brockschmidt, Franjo Ivančić, Danny Tarlow, Parthasarathy Ranganathan, Petros Maniatis, Ahmed Omran, Zaheer Abbas, Anita Gergely, Martin Sevenich, Gufeng Zhang, Amy Hua, and Alexander Frömmgen. Customizing an llm for enterprise software engineering, 2026.

[29] Gemini-Team, Rohan Anil, Sebastian Borgeaud, Yonghui Wu, Jean-Baptiste Alayrac, Jiahui Yu, Radu Soricut, Johan Schalkwyk, Andrew M Dai, Anja Hauth, et al. Gemini: a family of highly capable multimodal models. *arXiv preprint arXiv:2312.11805*, 2023.

[30] Generative ai on vertex ai: Overview of model tuning for gemini. https://cloud.google.com/vertex-ai/generative-ai/docs/models/tune-gemini-overview, 2024.

[31] Petros Maniatis and Daniel Tarlow. Large sequence models for software development activities. https://research.google/blog/large-sequence-models-for-software-development-activities/, 2023.

[32] Baptiste Rozière, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Romain Sauvestre, Tal Remez, Jérémy Rapin, Artyom Kozhevnikov, Ivan Evtimov, Joanna Bitton, Manish Bhatt, Cristian Canton Ferrer, Aaron Grattafiori, Wenhan Xiong, Alexandre Défossez, Jade Copet, Faisal Azhar, Hugo Touvron, Louis Martin, Nicolas Usunier, Thomas Scialom, and Gabriel Synnaeve. Code llama: Open foundation models for code, 2024.

[33] Gang Ren, Eric Tune, Tipp Moseley, Yixin Shi, Silvius Rus, and Robert Hundt. Google-wide profiling: A continuous profiling infrastructure for data centers. *IEEE Micro*, pages 65–79, 2010.

[34] Brendan Gregg. The flame graph. *Commun. ACM*, 59(6):48–57, May 2016.

[35] Dongjun Youn, Sungho Lee, and Sukyoung Ryu. Declarative static analysis for multilingual programs using codeql. *Software: Practice and Experience*, 53(7):1472–1495, 2023.

[36] Sifei Luan, Di Yang, Celeste Barnaby, Koushik Sen, and Satish Chandra. Aroma: code recommendation via structural code search. *OOPSLA*, 2019.

[37] Jinhyuk Lee, Zhuyun Dai, Xiaoqi Ren, Blair Chen, Daniel Cer, Jeremy R. Cole, Kai Hui, Michael Boratko, Rajvi Kapadia, Wen Ding, Yi Luan, Sai Meher Karthik Duddu, Gustavo Hernandez Abrego, Weiqiang Shi, Nithi Gupta, Aditya Kusupati, Prateek Jain, Siddhartha Reddy Jonnalagadda, Ming-Wei Chang, and Iftekhar Naim. Gecko: Versatile text embeddings distilled from large language models, 2024.

[38] Raymond Li, Loubna Ben Allal, Yangtian Zi, Niklas Muennighoff, Denis Kocetkov, Chenghao Mou, Marc Marone, Christopher Akiki, Jia Li, Jenny Chim, Qian Liu, Evgenii Zheltonozhskii, Terry Yue Zhuo, Thomas Wang, Olivier Dehaene, Mishig Davaadorj, Joel Lamy-Poirier, João Monteiro, Oleh Shliazhko, Nicolas Gontier, Nicholas Meade, Armel Zebaze, Ming-Ho Yee, Logesh Kumar Umapathi, Jian Zhu, Benjamin Lipkin, Muhtasham Oblokulov, Zhiruo Wang, Rudra Murthy, Jason Stillerman, Siva Sankalp Patel, Dmitry Abulkhanov, Marco Zocca, Manan Dey, Zhihan Zhang, Nour Fahmy, Urvashi Bhattacharyya, Wenhao Yu, Swayam Singh, Sasha Luccioni, Paulo Villegas, Maxim Kunakov, Fedor Zhdanov, Manuel Romero, Tony Lee, Nadav Timor, Jennifer Ding, Claire Schlesinger, Hailey Schoelkopf, Jan Ebert, Tri Dao, Mayank Mishra, Alex Gu, Jennifer Robinson, Carolyn Jane Anderson, Brendan Dolan-Gavitt, Danish Contractor, Siva Reddy, Daniel Fried, Dzmitry Bahdanau, Yacine Jernite, Carlos Muñoz Ferrandis, Sean Hughes, Thomas Wolf, Arjun Guha, Leandro von Werra, and Harm de Vries. Starcoder: may the source be with you!, 2023.

[39] Daoguang Zan, Ailun Yu, Bo Shen, Bei Chen, Wei Li, Yongshun Gong, Xiaolin Chen, Yafen Yao, Weihua Luo, Bei Guan, Yan Liu, Yongji Wang, Qianxiang Wang, and Lizhen Cui. Diffcoder: Enhancing large language model on api invocation via analogical code exercises. *Proc. ACM Softw. Eng.*, 1(FSE), July 2024.

[40] Zhiyu Li, Shuai Lu, Daya Guo, Nan Duan, Shailesh Jannu, Grant Jenks, Deep Majumder, Jared Green, Alexey Svyatkovskiy, Shengyu Fu, and Neel Sundaresan. Automating code review activities by large-scale pre-training. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2022, page 1035–1047, New York, NY, USA, 2022. Association for Computing Machinery.

[41] Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. Bleu: a method for automatic evaluation of machine translation. In *ACL*, 2002.

[42] Chin-Yew Lin. Rouge: A package for automatic evaluation of summaries. In *Text summarization branches out*, pages 74–81, 2004.

[43] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, Denny Zhou, et al. Chain-of-thought prompting elicits reasoning in large language models. *NeurIPS*, 2022.

[44] Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. React: Synergizing reasoning and acting in language models. *arXiv preprint arXiv:2210.03629*, 2022.

[45] Xinyi Hou, Yanjie Zhao, Yue Liu, Zhou Yang, Kailong Wang, Li Li, Xiapu Luo, David Lo, John Grundy, and Haoyu Wang. Large language models for software engineering: A systematic literature review. *ACM Trans. Softw. Eng. Methodol.*, September 2024. Just Accepted.

[46] Noor Nashid, Mifta Sintaha, and Ali Mesbah. Retrieval-based prompt selection for code-related few-shot learning. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, pages 2450–2462, 2023.

[47] Jia Li, Ge Li, Yongmin Li, and Zhi Jin. Structured chain-of-thought prompting for code generation. *ACM Trans. Softw. Eng. Methodol.*, August 2024. Just Accepted.

[48] Titus Winters, Tom Manshreck, and Hyrum Wright. *Software engineering at google: Lessons learned from programming over time*. " O'Reilly Media, Inc.", 2020.

[49] Google LLC. How to do a code review. https://google.github.io/eng-practices/review/reviewer/.

[50] Betsy Beyer, Niall Richard Murphy, David K Rensin, Kent Kawahara, and Stephen Thorne. *The site reliability workbook: practical ways to implement SRE*. " O'Reilly Media, Inc.", 2018.

[51] Shuo Ren, Daya Guo, Shuai Lu, Long Zhou, Shujie Liu, Duyu Tang, Neel Sundaresan, Ming Zhou, Ambrosio Blanco, and Shuai Ma. Codebleu: a method for automatic evaluation of code synthesis. *CoRR*, abs/2009.10297, 2020.

[52] Hinrich Schütze, Christopher D Manning, and Prabhakar Raghavan. *Introduction to information retrieval*, volume 39. Cambridge University Press Cambridge, 2008.

[53] Rachel L. Thomas and David Uminsky. The problem with metrics is a fundamental problem for AI. *CoRR*, abs/2002.08512, 2020.

[54] Muhammad Tirmazi, Adam Barker, Nan Deng, Md Ehtesam Haque, Zhijing Gene Qin, Steven Hand, Mor Harchol-Balter, and John Wilkes. Borg: the next generation. In *EuroSys'20*, Heraklion, Crete, 2020.

[55] Google. Gemini developer api pricing. https://ai.google.dev/gemini-api/docs/pricing.

[56] Eric Schkufza, Rahul Sharma, and Alex Aiken. Stochastic superoptimization. *ACM SIGARCH Computer Architecture News*, 41(1):305–316, 2013.

[57] Alhussein Fawzi, Matej Balog, Aja Huang, Thomas Hubert, Bernardino Romera-Paredes, Mohammadamin Barekatain, Alexander Novikov, Francisco J R Ruiz, Julian Schrittwieser, Grzegorz Swirszcz, et al. Discovering faster matrix multiplication algorithms with reinforcement learning. *Nature*, 610(7930):47–53, 2022.

[58] Daniel J Mankowitz, Andrea Michi, Anton Zhernov, Marco Gelmi, Marco Selvi, Cosmin Paduraru, Edouard Leurent, Shariq Iqbal, Jean-Baptiste Lespiau, Alex Ahern, et al. Faster sorting algorithms discovered using deep reinforcement learning. *Nature*, 618(7964):257–263, 2023.

[59] Codeforces. https://codeforces.com/.

[60] Spandan Garg, Roshanak Zilouchian Moghaddam, Colin B. Clement, Neel Sundaresan, and Chen Wu. Deepdev-perf: a deep learning-based approach for improving software performance. In *ESEC/FSE*, 2022.

[61] Weichen Li, Albert Jan, Baishakhi Ray, Junfeng Yang, Chengzhi Mao, and Kexin Pei. Editlord: Learning code transformation rules for code editing. In *Forty-second International Conference on Machine Learning*, 2025.

[62] Na Meng, Miryung Kim, and Kathryn S. McKinley. Systematic editing: generating program transformations from an example. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '11, page 329–342, New York, NY, USA, 2011. Association for Computing Machinery.

[63] Anders Miltner, Sumit Gulwani, Vu Le, Alan Leung, Arjun Radhakrishna, Gustavo Soares, Ashish Tiwari, and Abhishek Udupa. On the fly synthesis of edit suggestions. *Proc. ACM Program. Lang.*, 3(OOPSLA), October 2019.

[64] Farnaz Behrang, Zhizhou Zhang, Georgian-Vlad Saioc, Peng Liu, and Milind Chabbi. Dr.fix: Automatically fixing data races at industry scale. *Proc. ACM Program. Lang.*, 9(PLDI), June 2025.

[65] Ramakrishna Bairi, Atharv Sonwane, Aditya Kanade, Vageesh D. C., Arun Iyer, Suresh Parthasarathy, Sriram Rajamani, B. Ashok, and Shashank Shet. Codeplan: Repository-level coding using llms and planning. *Proc. ACM Softw. Eng.*, 1(FSE), July 2024.

[66] Daniel Marjamäki. Cppcheck. https://www.cppcheck.com.

[67] Adrian Nistor, Po-Chun Chang, Cosmin Radoi, and Shan Lu. Caramel: Detecting and fixing performance problems that have non-intrusive fixes. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 1, pages 902–912, 2015.

[68] Oswaldo Olivo, Isil Dillig, and Calvin Lin. Static detection of asymptotic performance bugs in collection traversals. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '15, page 369–378, New York, NY, USA, 2015. Association for Computing Machinery.

[69] Teresa Johnson, Mehdi Amini, and Xinliang David Li. Thinlto: Scalable and incremental lto. In *2017 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 111–121, 2017.

[70] Patrick J. Chapman, Cindy Rubio-González, and Aditya V. Thakur. Interleaving static analysis and llm prompting. In *Proceedings of the 13th ACM SIGPLAN International Workshop on the State Of the Art in Program Analysis*, SOAP 2024, page 9–17, New York, NY, USA, 2024. Association for Computing Machinery.

[71] Google LLC. Protobuf. https://protobuf.dev/.

[72] DeepSeek-AI. Deepseek llm: Scaling open-source language models with longtermism. *arXiv preprint arXiv:2401.02954*, 2024.

[73] AI @ Meta Llama Team. The llama 3 herd of models, 2024.

[74] Albert Q. Jiang, Alexandre Sablayrolles, Arthur Mensch, Chris Bamford, Devendra Singh Chaplot, Diego de las Casas, Florian Bressand, Gianna Lengyel, Guillaume Lample, Lucile Saulnier, Lélio Renard Lavaud, Marie-Anne Lachaux, Pierre Stock, Teven Le Scao, Thibaut Lavril, Thomas Wang, Timothée Lacroix, and William El Sayed. Mistral 7b, 2023.

[75] karpathy. nanogpt. https://github.com/karpathy/nanoGPT.

[76] Hugging Face. Peft. https://github.com/huggingface/peft.

[77] PyTorch. torchtune. https://github.com/pytorch/torchtune.

[78] Daniel Han and Michael Han. unsloth. https://unsloth.ai/.

[79] Daniel Gomez Blanco. *Practical OpenTelemetry*. Springer, 2023.

[80] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.

[81] Tensorflow. Github: tensorflow/tensorflow - pull request #74770. https://github.com/tensorflow/tensorflow/pull/74770, 2024.

Supplementary Material

In this appendix, we provide additional details to the main paper, expanding on key areas such as a comprehensive list of anti-patterns identified, full LLM trajectory examples showcasing various querying approaches, multi-file edit considerations, LLM self-validation examples, human code reviewer interactions, and a discussion on the reproducibility of this work. This appendix is provided to offer a deeper understanding of the methodology, challenges, and outcomes of ECO's approach to automating code efficiency optimization.

A Anti-Patterns

Figure 11 summarizes how ECO takes historical code commits and other code sources and turns them into an anti-patterns dataset (Section 3). This provides additional context and examples for the description in the main paper.

We also provide a full list of anti-pattern categories extracted in the dataset in Table 7. Note that in the main paper, we provide more details on a subset of these categories, but include the full table here for completeness. As new anti-patterns are discovered, the table can grow over time.

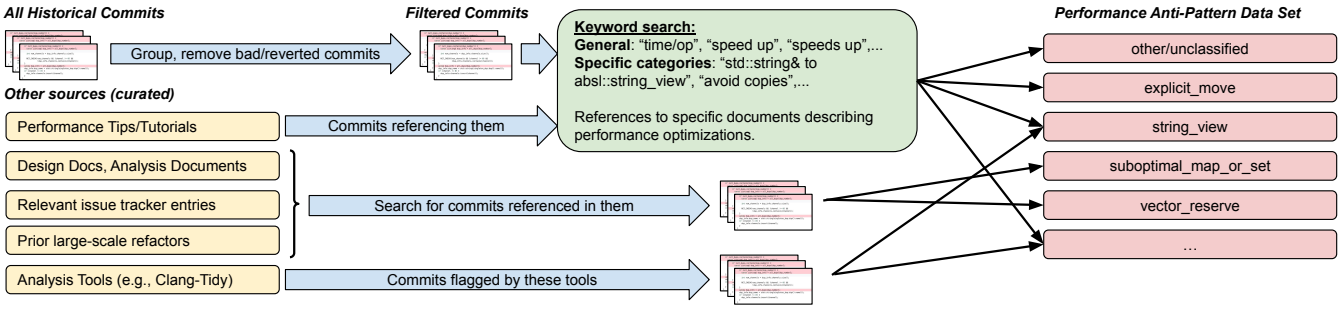


Figure 11: The approach and processing pipeline used to mine performance anti-patterns from our repository's commit history and a number of curated additional data sources.

Table 7: List of all Anti-Pattern Categories

Category	Description
Atomics	Use of std::atomic – std::atomic should be only used for simple properties where the value is returned as is and the property is set at most once with a mutex lock.
Callables	Unnecessary copies of a callable.
Data counters	Inefficient use of data processing counters.
Expensive Calls	Return early, cache results, or change the calling frequency of an expensive function call.
Futures	Using future allows users to treat future-producing APIs as cheaply copyable values with value semantics.
Hash Tuning	Improving hash tables with too many collisions or bits that never change across every hash with well-mixed hash functions provided by Abseil (`https://github.com/abseil/abseil-cpp`).
Hashes	Use of an inefficient hashing library when a more efficient option exists.
Heterogeneous Lookups	Map insertions/lookups causing the conversion of near-equivalent types due to user-specified comparator function to the map.
Intrusive Collection	Use elements of the collection to store state related to the implementation of the collection.
Lambda Capture	Capture by moving rather than copying and capturing individual member variables of an object.
Locks	Use of an inefficient lock type.
Missing `std::moves`	The last use of an object is copied instead of moved.
Missing Vector Reserves	See Figure 3b.
Optional	Use the std::optional when an object can be allocated on the stack but its population is deferred subsequent to construction.
PRNGs	Pseudorandom number generators that can use more efficient random number generation libraries.
Proto Arenas	Reduce the allocation and deallocation costs of protocol buffers by allocating them on a large piece of pre-allocated memory.
Proto Copying	Optimizations to reduce copying expensive protocol buffers.
Protos	Inefficient use of protos.
Redundant Map Operations	See Figure 3a.
Regexes	Inefficient use of regexes.
Return Value Optimization	Eliminate unnecessary copies of objects when returning from a function.
Salted Hash Sharding	To mitigate thread contention associated with a large shared table, shard the table and utilize a salted hash function to determine the appropriate shard for each key.
Specialized Swaps	Define a swap function in the class to swap objects of the class efficiently, and swap the object via std::swap within a scope guard.
String Comparison	String finds that can be replaced with more efficient finding functions.
String Concatenation	String concatenation that can be replaced with more efficient concatenation functions.
String Finds with Char	Use the character overload of `std::string::find()` and friends when the needle is a single character string literal.
String Views	String references that can be stored more efficiently.
Suboptimal Compressions	Changing or updating compression algorithms to improve performance.
Threads	Inefficient use of threads.
Type Erased Views	Use reference-like classes that do not own or copy data to enable code interaction with maps, random access ranges, and function objects without the necessity of templates.
Unmoved Unique Pointers	Avoid using an `std::unique_ptr` if it is never std::move to or from another pointer.
Unnecessary Allocations	Allocation of a new object when reusing an existing object is possible.
Unnecessary Anys	Use of an inefficient any which requires runtime type information and should be avoided in favor of std::variant.
Unnecessary Bound Checking	`std::vector::at` operator adds bounds checking overhead which can be avoided when the index is known to be valid, for example, part of a loop from 0 to size().
Unnecessary Copied Args	Using a particular internal API causing unnecessary copies of arguments from a widely used internal framework.
Unnecessary Copies	Replacing variables with references.
Unnecessary Indirections	Reduce pointer chasing and other forms of indirection
Unnecessary Map Copies	Use of an inefficient mapping function for data processing causing unnecessary copies to occur.
Unnecessary Sorts	Use of a sorted or stable map/set when not required.
Vector Reads	Missed use of a read-only view to the elements of `std::vector<T>`.

B LLM Trajectory Examples

This section provides examples of end-to-end LLM trajectories using the various prompting mechanisms described in Section 5. The prompts in blue show the query that is given to the LLM, and the LLM's responses are shown in grey, with additional syntax highlighting (added by us) to make the generated diff more readable. In the prompts, we added a placeholder for the target code that the prompt is applied to (the responses are based on calling the prompt with a representative example input).

B.1 Zero-Shot

This prompt format only includes the requested optimization, as well as the input code that it should be applied to. The model will respond with a snippet, based on its training.

Prompt 1: Example zero-shot prompt.

```
Reuse object allocations when possible to avoid allocating in a loop
{ target snippet }
```

Code Diff 2: Zero-shot generated code change.

```
@@[redacted]@@

      }
    }

+   [redacted]key;
    for (const auto& [redacted], value] : [redacted]_map) {
-     [redacted]key;
      key.ParseFromString([redacted]);
      emit_fn.Emit(key, value);
    }
```

B.2 Few-Shot

Here, we provide the model with specific positive and negative examples of an optimization. Just like before, the prompt then closes with the input code to optimize, and the model outputs a code diff when prompted.

Code Diff 3: Example few-shot prompt.

```
**Task**: As a C++ expert, you are required to analyze the use of the reserve method for std::vector. Ensure that vectors are pre-allocated ONLY when the
 size can be statically computed in advance.
**Guidelines**:
1. Avoid using reserve for trivial sizes; i.e., don't reserve space for just a few elements.
2. Avoid possible overallocations.
3. Consider special scenarios where reserve might be unnecessary or problematic.
4. The goal is to optimize memory usage without sacrificing runtime performance.
**Negative Examples**:
- **Example**:
  ```cpp
  std::vector<AutomatedRuns> failed_runs;
  std::vector<AutomatedRuns> successful_runs;
  successful_runs.reserve(Runs.size());
  failed_runs.reserve(Runs.size());
  for (auto Run : Runs) {
    if (Run.status == [redacted]::SUCCESS) {
      successful_runs.push_back(Run);
    } else {
      failed_runs.push_back(Run);
    }
  }
  ```
  Issue: Overallocation by reserving space for all Runs.
- **Example**:
  ```cpp
  [redacted].reserve([redacted].size());
  for (const ([redacted]& [redacted]:
  [redacted]) {
    if ([redacted].date >= start_date &&
    [redacted].date <= end_date) {
      [redacted].push_back([redacted]);
    }
  }
  return [redacted]
  ```
```

Issue: Overallocation by reserving space for all
[redacted].
- **Example**:
```cpp
auto arg_landscape = [redacted]->Compute(context);
arg_landscape.Reserve(state_.responses.size());
```
Issue: arg_landscape was already assigned at this point.
- **Example**:
```cpp
[redacted];
[redacted].reserve(options.[redacted]);
[redacted]
[redacted]false,
[redacted],
[redacted]);
if ([redacted].empty()) return;
```
Issue: [redacted]clears the output vector and calls reserve on it.
Positive Examples:
- **Example**:
```cpp
std::vector<std::pair<int, int>> ranges;
ranges.reserve(num_ranges);
//... (rest of the code)
return ranges;
```
- **Example**:
```cpp
std::vector<std::pair<int, int>> ranges;
ranges.reserve(num_ranges);
int last_added = 0;
for (int i = 0; i < num_ranges; i++) {
  const int start = last_added;
  const int end = start + elements_per_range + (i < remainder ? 1 : 0);
  ranges.push_back(std::make_pair(start, end));
  last_added = end;
}
return ranges;
```
Task: Given the above examples and guidelines, determine if we can improve the following code by adding reserve:
{ target snippet }

Code Diff 4: Few-shot generated code change.

```
@@[redacted]@@
  absl::Status s;
  if (expr.union__size() > 0) {
    std::vector<[redacted]*> children;
+   children.reserve(expr.union__size());
    for (const auto& child : expr.union_()) {
      children.push_back([redacted](child, parent_ref, context,
                                    relation_config, policy, source));
```

B.3 Chain-of-Thought

The key part of of this prompt if the last sentence – "Let's think this through step by step". This induces the model to generate a chain-of-thought sequence as opposed to directly outputting the answer.

Prompt 5: Example chain-of-thought prompt.

```
You're a software engineer [redacted] and you have identified the destructor associated with { proto name } in { class name } to be very expensive.
Optimize { class name } by using proto arenas. There are following rules to follow:
1. Declare an instance of proto2::Arena before any instance of { proto name }.
2. Replace any instance of { proto name } with proto2::ArenaSafeUniquePtr<{ proto name }>
3. Every object associated with proto2::ArenaSafeUniquePtr<{ proto_name }> would be constructed using proto2::MakeArenaSafeUnique<{ proto name }>.
4. Ensure that the same arena object is shared across the different instances of proto2::ArenaSafeUniquePtr<{ proto name }>.
5. Use pointer semantics for every object associated with proto2::ArenaSafeUniquePtr<{ proto name }>.

{ target snippet }

Let's think this through step by step.
```

Code Diff 6: Chain-of-thought generated code change.

```
Okay, let's analyze the provided code and apply the requested
changes to optimize [redacted] using proto arenas.

**Understanding the Problem**

The destructor for [redacted] is expensive, indicating
that it is not being allocated in an efficient way. Proto arenas
are a way to manage the allocation and deallocation of protocol
buffer messages in a more efficient way, especially in
high-throughput applications.

**Code Changes**

Here's the diff of the necessary code changes in
[redacted]:

**Explanation of Changes:**

1. **Include arena_safe_unique_ptr.h:** We need to include the
header file that provides proto2::ArenaSafeUniquePtr.
2. **Declare proto2::Arena:** We declare an proto2::Arena
object named arena_ before the declaration of [redacted].
This arena object will be used to allocate the [redacted]
proto.
3. **Replace [redacted] with
proto2::ArenaSafeUniquePtr<[redacted]>:**
The [redacted] member is now of type
proto2::ArenaSafeUniquePtr<[redacted]>.
4. **Allocate [redacted] on the arena:** We use
proto2::MakeArenaSafeUnique<[redacted]>(
&arena_) to allocate the [redacted] proto on the arena when
setting or swapping a new [redacted].
5. **Use pointer semantics:** We use pointer semantics (* or ->)
to access the underlying [redacted] proto. We now check
if the pointer [redacted] is not null and use the pointer
to access the proto.
6. **Clear [redacted]:** Use .reset() instead of =
std::nullopt to clear the [redacted] pointer.

**Impact of Changes**

*   **Improved destructor performance:** The destructor of
[redacted] will be faster because the arena will
deallocate all the allocated [redacted] objects at once,
avoiding the need to recursively call destructors.
*   **Potentially reduced memory fragmentation:** Arena
allocation can reduce memory fragmentation, as all
arena-allocated objects are released together.
*   **Reduced allocation overhead:** Arena-based allocation is
faster than standard new and delete.

**Additional Notes**

*   It is important to note that if [redacted] is used
in a context where the [redacted] is frequently set and
cleared, then the arena might grow significantly, and it might
be advantageous to reset the arena periodically.
*   The provided solution assumes that [redacted] does
```

```
not need to return a raw pointer to the [redacted]. If this
is needed, then it is necessary to return the raw pointer from
the [redacted]member.

This approach leverages the efficiency of proto arenas to
optimize the performance of [redacted]and its handling
of the [redacted]member.


@@[redacted]@@
 #include "[redacted]/public/callbackmetric.h"
 #include "[redacted]/public/concurrent.h"
 #include "[redacted]/public/counter.h"
+#include "[redacted]/arena/arena_safe_unique_ptr.h"
 #include "[redacted]/proto/descriptor.proto.h"
 #include "[redacted]/public/descriptor.h"
 #include "[redacted]/public/dynamic_message.h"
@@[redacted]@@
   }

   bool [redacted]() const override {
-    return [redacted].has_value();
+    return [redacted]!= nullptr;
   }

   const [redacted]& [redacted]() const override {
-    if ([redacted].has_value()) {
+    if ([redacted] != nullptr) {
       return *[redacted];
     }
     ProgrammingErrorDfatal()
@@[redacted]@@

   void Set[redacted](
       const [redacted]& [redacted]) override {
-    [redacted]= [redacted];
+    if ([redacted] == nullptr) {
+      [redacted]=
+          proto2::MakeArenaSafeUnique<[redacted]>(
+              &arena_);
+    }
+    *[redacted] = [redacted];
   }

   void Swap[redacted]([redacted]* [redacted]) override {
     if ([redacted] == nullptr) {
       ProgrammingErrorDfatal()
-          << __LOG__ << " passed `[redacted]`is a nullptr";
+          << __LOG__ << " passed `[redacted]`is a nullptr.";
       return;
     }
-    if (![redacted].has_value()) {
-      [redacted].emplace();
-    }
-    [redacted]->Swap([redacted]);
+    if ([redacted] == nullptr) {
+      [redacted]=
+          proto2::MakeArenaSafeUnique<[redacted]>(
+              &arena_);
+    }
+    [redacted]->Swap([redacted]);
   }

   void Clear[redacted]() override {
-    [redacted]= std::nullopt;
+    [redacted].reset();
   }

   int32_t [redacted]_count() const override { return 1; }
@@[redacted]@@
   [redacted]
   stats_census::TagSetProtoWithAdditionalMetadata
       census_tagset_proto_with_metadata_;
-
+
+  proto2::Arena arena_;
-  std::optional<[redacted]> [redacted];
+  proto2::ArenaSafeUniquePtr<[redacted]> [redacted];
 };
```

B.4 ReAct

When using ReAct, the LLM is invoked multiple times. In each step, it is given the full trajectory of the execution so far. The given prompt format will induce the model to follow a particular structure that results in it generating a "Thought" followed by an "Action", which is an executable command. The model knows this structure from its pre-training data, which included ReAct examples (an alternative approach would be to provide examples of the desired structure as few-shot examples in the prompt itself). The action is then executed, the output of the action is copied back into the prompt, and the next iteration begins. The final action results in the generation of a diff.

Prompt 7: Example ReAct prompt.

````
# Instruction :
Make a performance enhancing code edit that accomplishes the following:
```txt
Restricts the lambda capture scope
```
# Thought:
We need to first take a look at the source code.
# Action:
cat { target path }
# Observe:
```C++
{ target snippet }
```
# Thought:
Now we can write a code diff that will improve the source code's performance.
# Action:
````

Code Diff 8: ReAct generated code change.

```
@@[redacted]@@

   for (auto& query : *mutable_queries) {
     const int [redacted]= query.[redacted]();
     proto2::util::RemoveIf(
-        query.[redacted](), [this](const [redacted]* [redacted]) {
+        query.[redacted](),
+        [slot_position_threshold =
+            slot_position_threshold_](const [redacted]* [redacted]) {
           return ([redacted]->is_pla() &&
-                  [redacted]->slot_position() > slot_position_threshold_);
+                  [redacted]->slot_position() > slot_position_threshold);
         });
     const int [redacted]= query.[redacted]();
     [redacted]->IncrementBy([redacted] -
```

C Multi-File Edits

Although the bulk of this paper uses examples with single-function edits, ECO is not restricted to function-level edits. ECO uses an LLM fine-tuned with multi-file code changes and the techniques used in ECO apply to multi-file edits as well: e.g., localizing through code embeddings; targeted prompting for generating code optimizations; and validation through testing, self-review, and code reviews.

We recently expanded ECO to apply more multi-file code changes and we have found that the primary difference between multi-file and function-level edits is a necessity for descriptive prompting. For example, one of our anti-patterns involves optimizing usage of protocol buffers [71]. These protocol buffer edits are particularly complex because they can involve changes across multiple files (often >5 files at a time). Providing more detailed instructions in the prompts for these edits allows us to ensure that all multi-file dependencies/references are captured in the generated edit.

Prompt 5 shows a prompt that allowed us to effectively generate multi-file protocol buffer edits, and Code Diffs 9 and 10 show edits generated with the prompt.

Code Diff 9: File 1 of a change generated with Prompt 5.

```
@@[redacted]@@

 #include "[redacted]"
 #include "[redacted]"
 #include "[redacted]"
+#include "[redacted]"
 #include "[redacted]"

 namespace [redacted]{
@@[redacted]@@


   [redacted]
   const MutationProto* mutation() const {
-    return (mutation_ != nullptr) ? mutation_.get() : &entry_.mutation();
-  }
-
-  bool is_[redacted]_entry() const { return entry_.header().has_[redacted]_id(); }
-  DatabaseId database_id() const { return entry_.header().database_id(); }
-  int32_t [redacted]_id() const { return entry_.header().[redacted]_id(); }
+    return (mutation_ != nullptr) ? mutation_.get() : &entry_->mutation();
+  }
+
+  bool is_[redacted]_entry() const { return entry_->header().has_[redacted]_id(); }
+  DatabaseId database_id() const { return entry_->header().database_id(); }
+  int32_t [redacted]_id() const { return entry_->header().[redacted]_id(); }
   LogEntryHeaderProto* mutable_entry_header() {
-    return entry_.mutable_header();
+    return entry_->mutable_header();
   }
   bool is_[redacted]_info_persisted_in_entry_header() const {
-    return entry_.header().has_checksummed_[redacted]_encryption_info();
+    return entry_->header().has_checksummed_[redacted]_encryption_info();
   }

   [redacted]
@@[redacted]@@

   const std::string database_monitoring_name_;

   perftools::tracing::OperationTraceSpan trace_span_;
-  LogEntryProto entry_;
+
+  proto2::Arena arena_for_entry_;
+  proto2::ArenaSafeUniquePtr<LogEntryProto> entry_;

   std::string serialized_header_;
```

Code Diff 10: File 2 of a change generated with Prompt 5.

```
@@[redacted]@@

 #include "[redacted]"
 #include "[redacted]"
 #include "[redacted]"
+#include "[redacted]"
+#include "[redacted]"
 #include "[redacted]"
```

```
 #include "[redacted]"
 #include "[redacted]"
@@[redacted]@@

 #include "[redacted]"
 #include "[redacted]"
 #include "[redacted]"
+#include "[redacted]"
 #include "[redacted]"
 #include "[redacted]"
 #include "[redacted]"
@@[redacted]@@

      prev_sequence_(prev_sequence),
      tablet_uid_(tablet_uid.Encode()),
      durable_cb_(durable_cb) {
-  [redacted]* header = entry_.mutable_header();
+  entry_ = proto2::MakeArenaSafeUnique<[redacted]>(&arena_for_entry_);
+  [redacted]* header = entry_->mutable_header();
   header->set_prev_sequence(prev_sequence);
   header->set_database_id(database_id);
   header->set_tablet_uid(tablet_uid_);
@@[redacted]@@

   }

   FillTabletVersion(header->mutable_version());
-  entry_.set_allocated_mutation(mutation.release());
-  entry_.set_creation_wall_time_micros([redacted](creation_time_));
+  proto2::SetMessageField<"mutation">(std::move(mutation), *entry_);
+  entry_->set_creation_wall_time_micros([redacted](creation_time_));
   if (isolation_info) {
     isolation_info_ = *isolation_info;
     isolation_info_->SetDatabaseMonitoringName(database_monitoring_name_);
@@[redacted]@@

 }

 void [redacted]::Finalize(int64_t [redacted]) {
-  entry_.mutable_header()->set_sequence([redacted]);
+  entry_->mutable_header()->set_sequence([redacted]);
   sequence_ = [redacted];
 }

 void [redacted]::SetWriteTime([redacted] time) {
   [redacted]
-  entry_.set_wall_time_micros([redacted](time));
+  entry_->set_wall_time_micros([redacted](time));
 }

 void [redacted]::SetFlushTime([redacted] time) { flush_time_ = time; }
@@[redacted]@@

 const [redacted]& [redacted]::contents() const {
   if (should_encrypt_entry_) {
     [redacted]
-    CHECK(!entry_.[redacted]);
-    CHECK(entry_.[redacted]);
-  }
-  return entry_;
+    CHECK(!entry_->[redacted]);
+    CHECK(entry_->[redacted]);
+  }
+  return *entry_;
 }

 [redacted]::AppendLogEntryToBuffer(LogRecordBuffer& buffer) {
   if (!should_encrypt_entry_) {
     [redacted]
     [redacted]
-    return buffer.Append(entry_, [redacted]::kChecksumFieldNumber);
+    return buffer.Append(*entry_, [redacted]::kChecksumFieldNumber);
   } else {
     [redacted]
     [redacted]
     CHECK(!serialized_header_.empty());
     ScopedEncryptedLogEntryForSerialization to_be_serialized(
-        &entry_, &serialized_header_);
+        entry_.get(), &serialized_header_);
     return buffer.Append(to_be_serialized.MessageToSerialize(),
                          [redacted]::kChecksumFieldNumber);
```

```
   }
@@[redacted]@@

                          const [redacted]& mutation_cord) {
   if (should_encrypt_entry_) {
     [redacted]
-    entry_.header().SerializeToString(&serialized_header_);
+    entry_->header().SerializeToString(&serialized_header_);
     [redacted]
-    mutation_ = [redacted](&entry_, mutation_cord, serialized_header_);
+    mutation_ =
+        [redacted](entry_.get(), mutation_cord, serialized_header_);
   }
 }
```

D Self-Validation

We provide an example of LLM self-validation for edits generated to fix missing vector reserves (Figure 3b). For evaluation of these edits, we prompt the LLM to answer several questions:

1. Is the reserved amount in the vector appropriate?
2. Is the `push_back` of elements in the vector happening within the scope of a conditional block like `if`? (We want to avoid reserving when we cannot pre-determine the vector size.)
3. Is the vector being modified in the surrounding code and are there `push_back`s happening outside the context of a for-loop? (If yes, we want to avoid reserving to avoid miscalculations.)
4. Does the reserve count in vector depend on nested for-loops? (We want to avoid reserving when we cannot easily compute the nested loop values.)

Prompt 11 provides an example of a self-validation prompt template for the first question. When given Code Snippet 12, the LLM validation returns "No" as shown in Response 13. An example code snippet that receives the opposite response ("Yes") is shown in Code Snippet 14 and Response 15.

Prompt 11: Example of a self-validation prompt.

```
You are an expert C++ performance engineer tasked with reviewing code. Your goal is to assess the quality of a code snippet which seeks to preallocate a
vector to avoid reallocations when elements are continuously added in a for loop. You will be presented with a code and a question.

  Code:
  ```cpp
  { target snippet }
  ```

  Question:
  Is the reserved amount in the vector { target vector } appropriate (only reserve when we know the exact size in advance)? (Yes or No)
```

Code Snippet 12: Example input code for Prompt 11.

```
[redacted]<std::vector<std::string>> [redacted](
    const [redacted]& [redacted], [redacted]<const std::string> input_full_names) {
  [redacted]
  std::vector<std::string> input_paths;
  input_paths.reserve(input_full_names.size());
  std::vector<std::string> missing_full_names;
  [redacted]
  [redacted]<std::string, std::string> [redacted]_map;
  for (const auto& data : [redacted].inputs()) {
    [redacted]_map[data.full_name()] = data.path();
  }
  [redacted]
  for (const std::string& input_full_name : input_full_names) {
    if (auto it = [redacted]_map.find(input_full_name);
        it != [redacted]_map.end()) {
      input_paths.push_back(it->second);
    } else {
      missing_full_names.push_back(input_full_name);
    }
  }
  [redacted]
  if (!missing_full_names.empty())
    return [redacted]Error(
        [redacted]("No paths found in input data for below full names: ",
                   [redacted]StrJoin(missing_full_names, ",")));
  [redacted]
  return input_paths;
}
```

Response 13: LLM response to Prompt 11 with input Code 12.

Okay, let's analyze the `reserve` call in the provided C++ code snippet.

Analysis:

The code aims to find matching paths from a `[redacted]` based on a list of `input_full_names`. It builds a hash map for quick lookup and then iterates through the `input_full_names`. If a name is found in the map, its corresponding path is added to the `input_paths` vector. If not, the name is added to a `missing_full_names` vector, and if this vector is non-empty at the end of the iteration, an error will be returned.

The relevant line is:

```c++
input_paths.reserve(input_full_names.size());
```

Here's a breakdown of whether the `reserve` is appropriate:

* **Worst-Case Scenario:** The code reserves space for `input_full_names.size()` elements. This is the *maximum* number of paths that could be added to `input_paths`. This happens only when all `input_full_names` exist in `[redacted].inputs()`.
* **Likely Scenario:** It's possible (and perhaps even probable) that some of the `input_full_names` will *not* be found in the `[redacted]_map`. In this case, the `reserve` call will have allocated *more* space than necessary.
* **Exact size unknown**: We do not know the exact size of the vector in advance.

Conclusion and Explanation:
The `reserve` call is *not perfectly appropriate* because it reserves the maximum possible size, not the *exact* size. The exact size is not known beforehand, because not all names in `input_full_names` are guaranteed to be found in `[redacted].inputs()`. A more precise `reserve` would be impossible without iterating through `input_full_names` and checking the map first, which would duplicate the work of the main loop.

The current `reserve` is a reasonable optimization, as it avoids reallocations in the best-case/worst-case scenario where all names are found. However, it may lead to over-allocation in other cases. The performance benefit of the current approach likely outweighs the potential cost of over-allocation in most realistic scenarios, especially if the expected number of missing names is small compared to the total number of names. If we knew, for instance, that 99% of names were always present, the current approach would be very efficient.

Answer:

No

Code Snippet 14: Example input code for Prompt 11.

```
std::vector<[redacted]>::[redacted]>
[redacted](const std::vector<int> criterions) {
  std::vector<[redacted]::[redacted]> criterion;
  criterion.reserve(criterions.size());
  for (auto id : criterions) {
    [redacted]::[redacted] tc;
    tc.mutable_criterion()->set_criterion_id(id);
    criterion.push_back(tc);
  }
  return criterion;
}
```

Response 15: LLM response to Prompt 11 with input Code 14.

Okay, let's analyze the code snippet and answer the question.

Code Analysis

The code aims to create a vector of `[redacted]` objects. It takes a vector of integers (`criterions`) as input. For each integer in the input vector, it creates a `[redacted]` object, sets its `criterion_id`, and then adds it to the output vector `criterion`. The core of the question is the use of `criterion.reserve(criterions.size());`.

Explanation and Answer

* **`reserve()` vs. `resize()`:** It's crucial to distinguish between `reserve()` and `resize()`.
  * `reserve(n)`: Allocates enough memory to hold `n` elements *without* changing the size of the vector (i.e., `vector.size()` remains unchanged). Future `push_back()` operations (up to `n` elements) will not trigger reallocations, which are expensive.
  * `resize(n)`: Changes the size of the vector to `n`. If `n` is larger than the current size, the vector is expanded, and new elements are value-initialized (or default-constructed for objects). If `n` is smaller, the vector is truncated.

* **Appropriateness of `reserve()` in this case:** The code correctly uses `reserve(criterions.size())`. The loop iterates exactly `criterions.size()` times, and each iteration adds one element to the `criterion` vector. Therefore, we *know* the final size of the `criterion` vector in advance: it will be equal to `criterions.size()`. Using `reserve()` is the optimal approach here. It prevents any reallocations during the loop, leading to better performance, especially when `criterions.size()` is large.

* **Why not `resize()`?** Using `resize(criterions.size())` would be *incorrect* and potentially less efficient. `resize()` would create `criterions.size()` default-constructed `[redacted]` objects *before* the loop. The loop would then *copy* into these existing objects. The initial default construction is wasteful, and depending on the compiler and standard library implementation, might even be *less* efficient than the reserve-then-copy approach, as resize can also call the copy constructor to grow the container. `resize` also affects the `size()` of the vector, which is not the intent. We want the size to grow naturally with the `push_back` calls.

* **Worst case scenario with no `reserve()`:** Without the `reserve()` call, the `std::vector` would start with a small allocated capacity. As elements are added via `push_back()`, the vector would need to reallocate its internal storage multiple times. Each reallocation involves:
  1. Allocating a new, larger memory block.
  2. Copying all existing elements from the old memory block to the new one.
  3. Deallocating the old memory block.

  This reallocation process is O(N) in time complexity, and if it happens repeatedly, the overall performance degrades to O(N^2) in the worst case (though `std::vector` implementations typically use an exponential growth strategy to mitigate this).

Conclusion

The use of `reserve()` is appropriate and beneficial in this code. We know the exact size of the vector in advance, and `reserve()` is used to allocate the memory upfront, preventing costly reallocations.

Answer to the Question:

Yes

E Interactions of ECO and Code Review

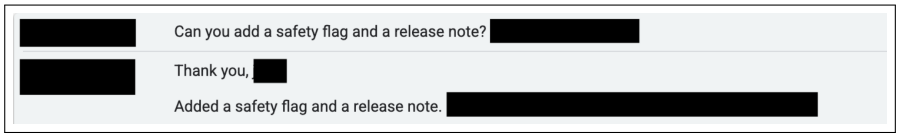
For every change generated by ECO, a human code owner is asked to review the change before the change is submitted to production. For 40%, 5%, and 41% of copy, map, and vector changes, respectively, ECO-generated changes were directly approved and submitted to production without the human reviewers leaving any feedback that had to be resolved. Table 5 in the main portion of the paper shows a breakdown of when additional action was required. We now provide a few examples of these specific reviewer interactions below.

Oftentimes, users left comments that were extraneous to the efficiency optimization on the code change (`S_USER` in Table 5). These comments had to be addressed before the code change was submitted to production. Figure 12a shows an example of feedback provided for a change that was submitted to production after reviewer comments were resolved.

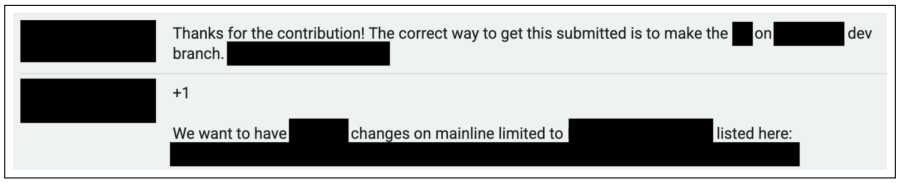
Other times, users rejected changes (`R_USER` in Table 5) and the changes were not submitted to production. This could be due to the human reviewer judging the code change to actually not be performance enhancing or the human reviewer wanting the code to be untouched for other reasons (e.g., style preferences). Figure 12b provides an example of such feedback.

ECO has also encountered code changes where feedback from human reviewers has caused regressions not initially present. For example, ECO generated the change in Code Diff 16. This change attempts to remove the redundant map lookups on `opt_out_handler_`. The reviewer for this change suggested using `try_emplace` instead, and in response to the reviewer's comments, the code was modified as reflected in Code Diff 17.

However `try_emplace` evaluates its argument expression every time, regardless of whether an insertion has occurred. This led to many calls to `make_unique` and caused a large increase in CPU consumption. Due to the regression, the change had to be reverted (`R_REVERT` in Table 5).



(a) Feedback that had to be resolved before ECO's change was submitted to production.



(b) Feedback rejecting change due to reasons unrelated to code performance.

Figure 12: Examples of reviewer feedback.

Code Diff 16: Initial ECO-generated code diff.

```
@@[redacted]@@
    const [redacted]& transform_task,
    const [redacted]<int32_t>* [redacted]) {
  transform_task_ = transform_task;
-  const [redacted]<int32_t>* [redacted]= ([redacted] == nullptr)
-                                        ? &[redacted]
-                                        : [redacted];
-  std::string handler_key = [redacted](*[redacted], "-");
-  if (opt_out_handler_.find(handler_key) == opt_out_handler_*.end()) {
-    opt_out_handler_[handler_key] =
-        std::make_unique<[redacted]>(
-            *[redacted], opt_out_info_, transform_task_;
+  std::string handler_key =
+      [redacted]((([redacted] == nullptr) ? [redacted]_
+                                          : *[redacted],
+                 "-");
+  auto& opt_out_handler = opt_out_handler_[handler_key];
+  if (opt_out_handler == nullptr) {
+    opt_out_handler = std::make_unique<[redacted]>(
+        ([redacted] == nullptr) ? [redacted]
+                                : *[redacted],
+        opt_out_info_, transform_task_);
  }
```

Code Diff 17: ECO-generated code diff after reviewer's feedback.

```
@@[redacted]@@
    const [redacted]& transform_task,
    const [redacted]<int32_t>* [redacted]) {
  transform_task_ = transform_task;
-  const [redacted]<int32_t>* [redacted]= ([redacted] == nullptr)
-                                        ? &[redacted]
-                                        : [redacted];
-  std::string handler_key = [redacted](*[redacted], "-");
-  if (opt_out_handler_.find(handler_key) == opt_out_handler_.end()) {
-    opt_out_handler_[handler_key] =
-        std::make_unique<[redacted]>(
-            *[redacted], opt_out_info_, transform_task_);
-  }
+  std::string handler_key =
+      [redacted]((([redacted] == nullptr) ? [redacted]_
+                                          : *[redacted],
+                 "-");
+  opt_out_handler_.try_emplace(
+      handler_key,
+      std::make_unique<[redacted]>(
+          ([redacted] == nullptr) ? [redacted]
+                                  : *[redacted],
+          opt_out_info_, transform_task_));
```

F Reproducibility

Since ECO is built for Google, the majority of ECO-generated changes shown in this paper have been targeted at Google's code base. However, ECO's techniques can be applied in the context of other organizations' systems infrastructure and code bases. Today, there are many open-source LLMs [72–74] that can be fine-tuned with open-source libraries or with LLM offerings [75–78]. Our methods for identifying performance opportunities and localizing candidate code can also be replicated using various open-source performance profiling tools [79] and similarity-search libraries like ScaNN [15], with required resources scaling with the size of the organization's code base.

Additionally, ECO has generated code changes not only on Google's code base but also on open-source code. Code Diff 18 shows an example of a change generated by ECO on TensorFlow (https://github.com/tensorflow/tensorflow) [80]. This change removes a redundant map lookup and was committed into TensorFlow's upstream repository [81].

Code Diff 18: Example of a code change generated on open-source code.

```
@@ -346,8 +346,9 @@

        fanouts_[output].emplace(node, -1);
      } else {
        max_input_port = i;
-       max_regular_output_port_[output.node] =
-           std::max(max_regular_output_port_[output.node], output.port_id);
+       int& max_regular_output_port = max_regular_output_port_[output.node];
+       max_regular_output_port =
+           std::max(max_regular_output_port, output.port_id);
        fanouts_[output].emplace(node, i);
      }
    }
```

G Comparison With Static Analysis

Table 6 describes 6 different common sub-optimal vector-reserve patterns and lists whether or not static analysis are able to resolve them. 5 out of the 6 are not resolvable with Clang-Tidy and none are optimized by CppCheck, whereas ECO is able to optimize each one. The table also shows speedup results from ECO optimizations on microbenchmarks, run against varying container sizes (from 8 to 4096 elements). Here we describe each vector-reserve pattern in more detail.

Pattern A: Standard Loop Insertion Clang-Tidy and ECO are both able to optimize loops using standard library vector insertions such as `.push_back`.

Code Snippet 19: Optimization example for Pattern A.

```
dirs.reserve(groups_.size());
for (Group g : groups_) {
    dirs.push_back(GroupDir(g));
}
```

Pattern B: Bulk Move via Algorithms This pattern moves elements into a destination vector using an algorithm wrapper (e.g., `absl::c_move` with `std::back_inserter`).

Code Snippet 20: Optimization example for Pattern B.

```
result_entities.reserve(result_entities.size() + source_entities.size());
absl::c_move(source_entities, std::back_inserter(result_entities));
```

Clang-Tidy fails here because the insertion is encapsulated inside an algorithm wrapper and iterator adapter, leaving no explicit loop in the local AST. ECO is able to recognize the element transfer semantics and recommends reserving.

Pattern C: Iterator-based Assignment on Custom Containers In production, developers frequently populate proprietary containers, such as a protocol buffers in a `RepeatedPtrField`. In the example below, `cache_value` is a protocol buffer.

Code Snippet 21: Optimization example for Pattern C.

```
cache_value.mutable_entities()->Reserve(result.entities.size());
*cache_value.mutable_entities() = {result.entities.begin(), result.entities.end()};
```

Clang-Tidy fails because its analysis rules are hardcoded for standard library types (mainly `std::vector`). It cannot reason about custom APIs or allocations triggered implicitly via assignment operators and initializer lists. By using an LLM, ECO can leverage learned API signatures to suggest the custom container's `Reserve()` method.

Pattern D: Generic Template Sizing & Iterators Generic, templated code is commonly used to write reusable copy utilities.

Code Snippet 22: Optimization example for Pattern D.

```
template <typename Range>
std::vector<Entity> Copy(const Range& source) {
  std::vector<Entity> destination;
  destination.reserve(source.size());
  destination.insert(destination.end(), source.begin(), source.end());
  return destination;
}
```

At analysis time, Clang-Tidy cannot resolve the exact type of the template parameter `Range`. It systematically ignores uninstantiated parameters to avoid risking broken code or generating complex fallback rule paths. LLMs are able to infer the range properties of abstract types and apply block insertions.

Pattern E: Inter-Procedural / Multi-file Size Mappings In complex systems, sizing information is often decoupled and evaluated across functional or compilation unit boundaries.

Code Snippet 23: Optimization example for Pattern E.

```
// Sub-optimal Implementation
void PopulateStats(const std::vector<Entity>& src, std::vector<Entity>& dest) {
  for (const auto& item : src) {
    if (item.isValid) { dest.push_back(item); }
  }
}

// Optimized Implementation
void PopulateStats(const std::vector<Entity>& src, const CollectionStats& stats, std::vector<Entity>& dest) {
  dest.reserve(stats.total_size);
  for (const auto& item : src) {
    if (item.isValid) { dest.push_back(item); }
  }
}
```

Clang-Tidy is limited to single-file compilation units and cannot trace size constraints spanning inter-procedural boundaries. Additionally, conditional loops prevent direct mapping of `src.size()`. With an LLM, ECO can map dependencies across compilation boundaries to recommend a `.reserve()` utilizing the pre-calculated size member (`stats.total_size`).

Pattern F: Custom Pointer Array Buffers This pattern involves custom logging or cache buffers that store object pointers instead of copying full structures. In this case, `buffer` is a custom, dynamically sized data structure that holds a set of pointers, which can be added to it using `Add`.

Code Snippet 24: Optimization example for Pattern F.

```
buffer.Reserve(source.size());
for (auto& item : source) { buffer.Add(&item); }
```

Similar to Pattern B, Clang-Tidy ignores this loop because the container is a custom class and uses a non-standard insertion method (`.Add()`). Clang-Tidy has no semantic awareness that `.Add()` triggers underlying array expansions. LLMs can identify the custom buffer's memory semantics and suggest a `Reserve()` call.

StriaTrace: Efficient Tracing and Diagnosis for Online LLM Inference (Operational Systems)

Haonan Wu[1,2] Yanqing Chen[2] Kun Qian[2] Xue Li[2] Jingbo Xu[2]
Erci Xu[1*] Ennan Zhai[2] Wenyuan Yu[2] Guangtao Xue[1,3] Jingren Zhou[2]
[1]*Shanghai Jiao Tong University* [2]*Alibaba Group*
[3]*Shanghai Key Laboratory of Trusted Data Circulation and Governance and Web3*

Abstract

Large Language Model (LLM) inference services in production operate under stringent, fine-grained Service Level Objectives (SLOs). Unlike throughput-oriented LLM training, even sporadic performance anomalies during inference can violate SLOs, underscoring the need for improved tracing and diagnosis solutions. However, existing solutions face two primary limitations: (1) existing tracing tools incur prohibitive overhead; (2) training-centric diagnosis tools are ill-suited for capturing sporadic inference anomalies.

To bridge these gaps, we propose StriaTrace, a novel tracing and diagnosis system tailored for online LLM inference. StriaTrace is built upon three principles distilled from production experience: (1) tracing key synchronization points, (2) tracing critical paths, and (3) detailed tracing only during abnormalities. StriaTrace further constructs a dynamic regression-based roofline model and correlation-based diagnosis to identify why each LLM inference abnormality happens. Evaluations show that StriaTrace reduces tracing overhead by 97.8% relative to alternatives. StriaTrace has been widely used in our development, testing, and production release cycles, and has successfully diagnosed hundreds of abnormalities spanning 19 distinct root causes.

1 Introduction

Large Language Models (LLMs) have emerged as a transformative technology, underpinning diverse applications such as chatbots, content generation, code completion, and scientific discovery. Different from classical Internet services (*e.g.,* web-search and e-commerce), running LLM inference focuses not only on satisfying the end-to-end request response time (*i.e.,* Time To First Token, TTFT) but also on optimizing the performance of each generated token (*i.e.,* Time Per Output Token, TPOT). To improve performance, modern LLM inference systems [17,28,40,46,52] employ sophisticated distributed architectures with advanced optimization techniques such as prefix caching, prefill/decode disaggregation [20,37,53], and various types of parallelism strategies [6,15,23,24,36,38,39,41,51].

However, high throughput does not guarantee high Quality-of-Service (QoS). In production, the common Service Level Objective (SLO) of TTFT is 10s, and the SLO for TPOT is 100ms. Our field analysis on a 2K instance cluster shows that, while the majority of requests have a 3.06s TTFT and 24.83ms TPOT on average, the tails (*i.e.,* 99th percentile) exhibit high variance and consistently exceed SLOs. Despite rigorous unit, sandbox, and end-to-end testing typical of large-scale systems, anomalies persist. We attribute these persistent anomalies to two fundamental factors. First, LLM inference is inherently streaming; it thus becomes extremely difficult to reproduce exact online workloads and execution environment in offline testing [11,50]. Moreover, the series of optimizations (*e.g.,* global prefix KV cache and prefill/decode disaggregation), while facilitating the performance of inference service, actually introduce extra complexity along the stack, requiring more effort for diagnosis.

This operational reality necessitates an end-to-end, detailed and continuous tracing system which can capture abnormal runtime events and expedite the diagnosis via automation. Unfortunately, all existing solutions fall short. General-purpose tracing tools from classical distributed systems primarily focus on CPU-related anomalies and lack the telemetry sources needed to capture GPU-centric performance issues. While specialized LLM profiling tools (*e.g.,* TorchProfiler [35] and Nsight [27]) can capture fine-grained execution traces in LLM contexts, they typically introduce significant performance overhead (ranging from 10% to 20%), making them unsuitable for continuous runtime tracing in production under the demanding SLOs.

On the other hand, emerging LLM-specific diagnostic approaches (*e.g.,* Aegis [9], Mycroft [8] and PerfTracker [13]) mainly target the LLM training scenario. They are also ill-suited for inference workloads due to fundamental differences in workload characteristics. First, LLM training is throughput-oriented while inference is latency-sensitive. In LLM training, only significant performance degradation (*i.e.,* persist-

*Corresponding author: Erci Xu, Email: jostep90@gmail.com

ing stragglers) is fatal for overall throughput, and existing solutions are focusing on these issues. However, in LLM inference, sporadic abnormalities can be fatal, directly causing SLO violations and thus impacting end users' experience [42]. The other key difference is that the LLM training workloads exhibit static execution patterns with fixed batch size and sequence length, and existing solutions rely on these characteristics to facilitate the diagnosis and alleviate overhead. However, in LLM inference scenarios, all requests are generated by end users, therefore these proposed diagnosis rules (*e.g.,* fixed-threshold-based straggler diagnosis in Mycroft [8]) are not directly applicable to the dynamic inference workload.

This paper presents StriaTrace, a novel tracing and diagnosis system specifically designed for online LLM inference services. The primary challenge is low-overhead continuous tracing. To address this challenge, we identify the minimal set of information required to pinpoint root causes of various issues in the inference context and distill three key principles for efficient telemetry collection.

First, effective root-cause diagnosis always relies on observing all cross-machine and cross-process interaction points. Guided by this principle, we identify the essential synchronization locations in typical distributed inference workflows, reducing the number of CPU-side instrumentation points from $O(1000)$ functions to $O(10)$ critical sites, significantly lowering CPU tracing overhead (§5.2).

Second, our operational experience in diagnosing real-world inference issues suggests that, leveraging the deterministic sequential dependencies in inference execution, accurate tracing of the critical path alone suffices to infer anomalies in non-critical paths. The extensive use of parallelism strategies and overlapping mechanisms allows us to focus only on changes in the critical path. Leveraging this insight, we optimize GPU-side tracing collection to focus only on GPU kernel execution and memory copy, thereby significantly reducing GPU tracing overhead by up to 97.5% (§5.3).

Third, although traces are generated for all tokens from all requests, only traces captured during ephemeral abnormalities are critical. Therefore, we need localized real-time diagnosis and filtering to further reduce the transmission cost. Building an online real-time diagnosis fitting the inference scenario is another key challenge. To achieve this, we propose a step-level autoregressive roofline model for LLM inference. This roofline model delivers two key benefits: (1) it enables in-situ, distributed anomaly detection directly within the inference process; (2) it enables distributed filtering based on roofline deviations, reporting only anomalous segments to the global analyzer—dramatically reducing data transmission and backend analysis overhead. In the centralized analyzer, correlation analysis is conducted among different types of traces to pinpoint the primary suspects for each abnormality. We further pre-correlate and persistently store the collected raw traces according to inference semantics, thereby avoiding time-consuming repeated memory accesses and on-the-fly correlation at scale (§5.4).

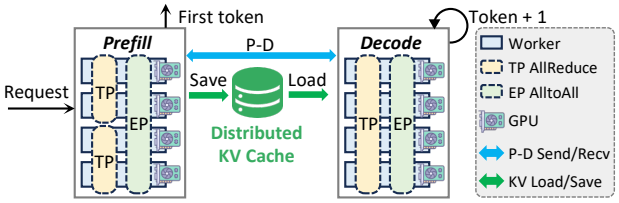


Figure 1: A typical LLM inference workflow with (1) prefix cache, (2) prefill/decode disaggregation and (3) TP+EP.

Our evaluation shows that StriaTrace consistently keeps the tracing overhead to less than 1%, which is 97.8% less compared to existing approaches. StriaTrace has been widely employed in our development, testing and release cycles. StriaTrace has successfully helped us diagnose 19 different root causes of abnormalities and diagnosed several issues that were intractable with conventional tools (§6 and §7). In production, StriaTrace has been running at scale for six months, instrumenting over 1,700 instances across two flagship LLM services and processing upwards of 180 million requests per day. It operates under both TP and DP configurations, and spans monolithic as well as PD-disaggregated deployments. We present the traces and diagnosis results in production in the appendix.

2 Background

Inference workflow. LLM inference follows a prefill-decode workflow. Upon receiving a request, the service initiates the prefill stage to process the input prompt. Then, the first output token is generated and this interval is measured as TTFT. Then, the service proceeds to the decode stage. In each step, the next token is generated based on the current sequence context. At this stage, we measure the TPOT, defined as the inter-token latency between generating token$_i$ and token$_{i+1}$. In production, we typically enforce SLOs of approximately 5-10 s for TTFT and 50-100 ms for TPOT. To better meet the SLOs, we employ a combination of techniques illustrated in Figure 1 and discussed as follows.

Distributed prefix KV cache. The core idea is to reuse KV cache that shares the same prefix across multiple requests. Before prefill, each request attempts to load its matching prefix from a distributed KV cache, while newly generated KV cache during prefill is written back. This approach requires synchronization and data transfer across processes and machines, and substantially improves TTFT.

Prefill/decode disaggregation. This architecture partitions serving instances into distinct prefill and decode pools. After a request completes prefill, the resulting KV cache and the request are transferred to a decode instance. Since prefill is compute-bound while decode is memory-bound, separation enables independent resource allocation and scheduling for each phase.

Inference parallelism. Model parallelism, specifically Tensor Parallelism (TP) and Expert Parallelism (EP), is commonly used in practice. In TP, each Attention or Feed-Forward Network (FFN) layer is partitioned across workers; each worker executes a portion of the computation and synchronizes results with each other. In EP, each token's computation is routed to a subset of experts, and the outputs are subsequently aggregated.

3 Motivation

3.1 Pervasive Performance Degradation

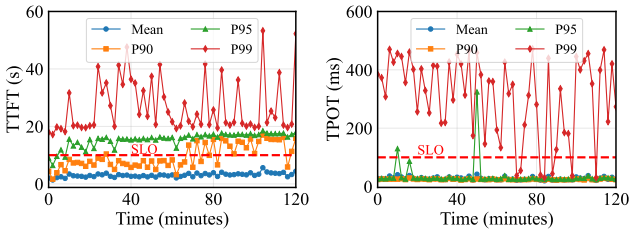
Despite the optimizations in §2, we observe that the average TTFT and TPOT remain within acceptable bounds but the tails of both can still experience severe SLO violations. Figure 2 shows the distribution of TTFT and TPOT from average to 99th percentile in a large-scale commercial LLM chatbot hosted on our platform. Specifically, this inference service contains more than 2K instances and processes 14M requests every day. There are 5.34% of requests violating the TTFT SLO and 1.18% of tokens violating the TPOT SLO.

In this paper, the tail latency is defined by *system-induced*, rather than the semantic variance. A long-tail therefore refers to a latency deviation when executing the *same computational workload* compared to its historical distribution. For instance, a prefix cache miss can drastically enlarge TTFT. However, from the inference engine's perspective, a cache miss simply entails a larger number of tokens to be actually computed during prefill. The resulting TTFT elongation due to increased computation workload is not considered a long-tail. Once semantic variance is factored out, the remaining system-induced tail latency directly manifests as stalls in the TTFT and TPOT.

The criticality of mitigating tail latencies cannot be overstated. Most LLM inference services are human-interactive, *e.g.,* chatbots, AI assistants and coding. Even occasional SLO violations are easily perceived by customers and lead to user-experience degradation. Such degradation severely impacts user retention, resulting in millions of dollars in revenue loss. However, taming tail latencies is non-trivial. First, Figure 2 shows that the tails of TTFT and TPOT exhibit high variance, *i.e.,* slow tokens can be occasional or even one-off events. In addition, as an online service, the workload is intrinsically dynamic, rendering offline reproduction ineffective. Hence, diagnosing the culprits often requires an intensive manual effort. For example, engineers must hypothesize potential performance issues based on experience, add logging statements at suspected code locations, build and deploy a new release, and then repeat this cycle.

3.2 Limitations of Existing Solutions

Although both industry and academia have proposed numerous diagnostic tools, with some even directly targeting LLM scenarios (*e.g.,* Neutrino [14], Mycroft [8] and Minder [7]), these solutions prove fundamentally ill-suited for production LLM inference.



(a) TTFT. (b) TPOT.

Figure 2: TTFT/TPOT spikes are pervasive in production.

High overhead. As shown in Table 1, several tools [4, 10, 14, 27, 29, 34, 35] are tailored for fine-grained tracing. However, our preliminary attempts at integrating these tools proved unsuccessful. The primary impediment is their prohibitive overhead. For example, Neutrino [14] is the state-of-the-art GPU tracing tool, but it may introduce overhead such as ~5% throughput loss due to the intensive memory I/O. We observed comparable or higher overheads when evaluating TorchProfiler [35], Nsight [27] and PerfTracker [13]. Even single-digit overhead is consequential. Owing to the high CapEx and OpEx of running a GPU-based cluster and the scale of today's LLM inference, a 5% loss can easily translate to a multi-million dollar cost increase. As a result, we restrict the usage of these tools on a subset of severe latency anomalies and facilitate certain offline debugging procedures.

Challenge 1: Achieving always-on tracing with negligible overhead.

Inapplicability to inference. A number of diagnostic approaches [7–9, 13] specifically for LLM training have recently emerged, aiming to pinpoint the root causes through cross-layer information collection and correlation analysis. Nevertheless, these approaches fall short as well. First, training scenarios are throughput-oriented. Consequently, anomaly detection in training targets persistent stragglers rather than the transient latency spikes. This focus, however, makes it ill-suited for tracing inference workloads. For example, Mycroft [8] employs coarse-grained (second-level) thresholds to identify performance degradations, and Minder [7] even takes several minutes to identify abnormalities. Moreover, LLM training exhibits highly regular, repetitive execution patterns due to fixed batch size and static computational graphs, which can be leveraged to identify stragglers via peer comparison within a synchronous data-parallel group. Inference workloads exhibit significant variance in batch size and sequence length, thus precluding such direct comparisons.

Challenge 2: Accurate diagnosis tailored for dynamic and sporadic inference anomalies.

Table 1: Limitations of existing solutions

Solutions	Data Source†	Overhead	Temporal Strategy	Runtime Diagnosis
TorchProfiler [35]	CPU+GPU	10-20%	On Demand	✗
PyStack [31]/py-spy [30]	Only CPU	<1%	Sample	✗
eBPF [29]	Only CPU	10-20%	Continuous	✗
Scalene [4]	CPU + partial GPU	<5%	Sample	✗
Nsight [27]	Partial CPU + GPU	10-20%	On Demand	✗
Neutrino [14]	Only GPU	∼5%	Continuous	✗
Aegis [9]	Only Comm	<1%	Continuous	Training
Minder [7]	Metric	<1%	Continuous	Training
Mycroft [8]	Only Comm	<1%	Continuous	Training
PerfTracker [13]	CPU+GPU	10-20%	On Demand	Training
StriaTrace (Our Solution)	CPU+GPU	∼1%	Continuous	Inference

† Complete GPU tracing data can cover Comm tracing data, since all collective communication is launched by GPU kernels.

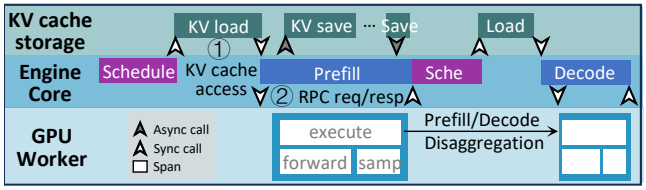


Figure 3: Key synchronization points.

4 Exploring Design Space

To achieve low-overhead and continuous tracing of online inference services, we formulate three design principles grounded in production experience.

Principle 1: Instrument synchronization barriers rather than internal functions. An inference request can traverse complex stages from entry to response. In our profiling, there can be $O(100)$ functions and $O(10K)$ function calls in an inference step (more details in §5.2.1). Tracing each of them would introduce prohibitive overhead. Among these functions, a subset constitutes the key synchronization functions which can be categorized into two types. The first is *inter-process RPC within the inference engine itself*. For example, in vLLM [46], the EngineCore responsible for request scheduling and the GPUWorker responsible for actual computation interact through collective RPC calls (labeled ② in Figure 3). The second is *function calls between the inference engine and external components*. For instance, the engine invokes metadata queries and I/O operations on external KV cache components, as labeled ① in Figure 3. These synchronization points gate the progression of the request to the next stage. Consequently, a core design decision of our tracing system is to instrument at these synchronization boundaries rather than exhaustive profiling.

Principle 2: Prioritize critical path profiling over exhaustive tracing. Even when restricting instrumentation to only synchronization points, exhaustively tracing every event within a request's lifecycle still incurs prohibitive overhead. The root cause lies in the internal concurrency of inference engines. For example, in vLLM, the EngineCore runs multiple concurrent threads: the main inference thread that orchestrates request execution, a dedicated thread for distributed KV cache access, and additional threads for monitoring and metrics collection, etc. Among these concurrent threads, only the main inference thread lies on the critical path that determines the request's completion time. Consequently, our tracing design prioritizes the critical path while discarding detailed tracing for non-critical threads.

Principle 3: Employ anomaly-driven high-fidelity tracing. Approaches employing *full persistence*, *e.g.,* TorchProfiler [35] and Nsight [27], capture all tracing data and lead to a tremendous bandwidth cost (see more details in §6.2). Conversely, *summarized/partial persistence*, *e.g.,* PerfTracker [13], Aegis [9] and Mycroft [8], sacrifices full-fidelity information to achieve lower transmission and storage overhead. However, they lack enough information to pinpoint the culprit in the inference scenario (as discussed in §3.2). Our field experience and statistics show that the vast majority of requests exhibit nominal behavior, and only a small fraction of requests are outliers. Therefore, we adopt *selective persistence*, *i.e.,* only retaining fine-grained tracing data for slow requests.

5 Design

We introduce StriaTrace, a low-overhead tracing and diagnosis tool for LLM inference. We detail the instrumentation mechanisms for CPU and GPU in §5.2 and §5.3, respectively. Finally, §5.4 explains how StriaTrace leverages the collected traces to identify performance anomalies.

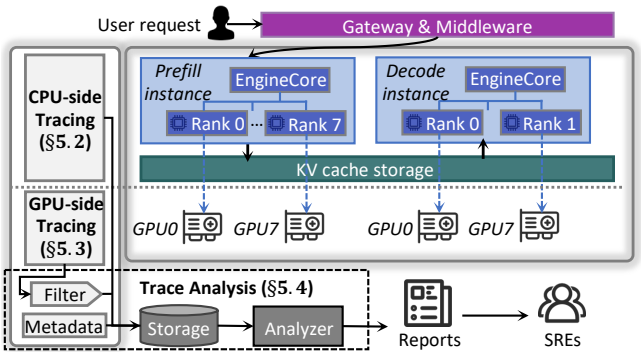


Figure 4: Overview of StriaTrace.

Table 2: Host-side function invocation statistics and instrumentation scope in a single step.

	EngineCore	Worker (P)	Worker (D)
Function types	182	493	284
Function calls	10,000+	3,588	953
StriaTrace spans	13	32	

5.1 Overview

As shown in Figure 4, StriaTrace comprises two components: distributed trace collectors co-located with each serving instance, and a centralized backend for data persistence and analysis. In this paper, a serving instance refers to a vLLM process group on a multi-GPU node.

On the **CPU side** (§5.2), StriaTrace's observability is confined to the vLLM serving instance. For each instance, StriaTrace focuses on the following components: (1) the EngineCore, which handles request scheduling and KV cache management; (2) the parallel ranks (i.e., GPUWorkers), each responsible for token computation on a dedicated GPU. As illustrated in Figure 3, a request's processing flow involves the EngineCore resolving prefix cache and scheduling, followed by RPC dispatch to GPUWorkers for execution, with KV cache I/O interactions with external components. On the **GPU side** (§5.3), StriaTrace operates concurrently on each GPUWorker, capturing kernel executions and memory transfers.

Collectively, StriaTrace records three types of trace data: (1) CPU-side traces, which record the processing latency across system modules; (2) GPU-side traces, which capture hardware-level execution; and (3) essential metadata, including the request ID and the instance IDs traversed by the request to reconstruct the causal chain. These traces are streamed to a centralized analysis backend (§5.4). The backend correlates these heterogeneous streams to reconstruct the complete execution flow of each request. It then leverages a performance roofline model to detect outliers and automatically reports anomalies.

5.2 Host-side Tracing

5.2.1 Coarse-grained Instrumentation

Designed for production monitoring, StriaTrace eschews the exhaustive function-level profiling employed by tools like cProfile [32]. Applying such pervasive profiling to high-throughput engines can incur prohibitive overhead. To quantify the cost, we conduct pervasive profiling on a vLLM instance serving Qwen2.5-0.5B [43]. As shown in Table 2, vLLM's EngineCore executes over 10,000 function calls in a single inference step, while the GPUWorker still triggers thousands of calls during prefill. Consequently, employing pervasive profiling would inflate the latency by tens of milliseconds, which is unacceptable for online inference serving. Following **Principle 1**, StriaTrace employs a hybrid instrumentation strategy to balance visibility and overhead:

Synchronization barrier instrumentation. First, we delineate the critical execution path of the inference engine. Modern inference engines (*e.g.,* vLLM) generally operate on an event-loop mode: cyclically fetching incoming requests, allocating KV cache, and performing per-token inference. Based on this architecture, StriaTrace strategically places tracing hooks at *cross-process synchronization boundaries* along the main loop. Specifically, we categorize these barriers into two types: (1) *Intra-engine cross-process RPCs*: Within vLLM, communication channels among the frontend API server, the EngineCore, and the distributed GPU workers are inherently mediated via Remote Procedure Calls (RPCs). Instrumenting these RPC boundaries naturally covers the inter-process scheduling and orchestration. (2) *Synchronous invocations with external processes*: We also trace mandatory blocking calls to external processes on the main loop. A prime example is the KV cache connector—the dedicated interface class responsible for querying and synchronizing with the distributed KV cache. Targeting these communication boundaries, StriaTrace strictly delimits execution stages and captures blocking latencies without exhaustive profiling.

Semantic span instrumentation. StriaTrace further subdivides the monolithic phases between synchronization barriers into semantic spans. Functionally, a semantic span delimits the execution of a specific logical routine along the inference critical path. In practice, the selection of semantic spans typically aligns with the top-level function invocations between two consecutive synchronization barriers. For instance, we partition the vLLM engine core's execution flow into distinct spans: `schedule` (managing request queues and allocating KV cache), `execute_model` (invoking the forward pass), and `sample_tokens` (generating output tokens based on logits). Crucially, StriaTrace goes beyond mere latency tracking by enriching these semantic spans with runtime context. For example, alongside the `schedule` span, StriaTrace records the batch-level metadata, including the number of

scheduled requests and their respective token status[1]. This context preservation is vital for correlating latency anomalies with workload dynamics. Details of these instrumentation points are provided in Appendix A.

Collectively, these two strategies comprise only dozens of instrumentation points—orders of magnitude fewer than the total function calls in each inference step. This design renders runtime overhead negligible while preserving structural visibility into the system's behavior.

5.2.2 Fine-grained Context Profiling

Through the key instrumentation points in §5.2.1, StriaTrace establishes macro-level observability over the CPU execution. However, when a performance anomaly occurs, this coarse-grained telemetry lacks the fidelity to pinpoint the specific code paths or internal function calls responsible for the anomaly. To bridge this gap, StriaTrace leverages `py-spy` [30], a low-overhead sampling profiler, to capture fine-grained execution contexts. Specifically, StriaTrace operates `py-spy` in non-blocking mode to periodically snapshot the execution state—including thread stacks and Global Interpreter Lock (GIL) ownership—of the vLLM instance. By reading the target process's memory, `py-spy` reconstructs the full Python stack trace without pausing execution or modifying the interpreter state. We configure the sampling interval to 10 ms, sufficient to capture sustained stalls.

A typical application of StriaTrace's `py-spy` integration is detecting GIL contention. Figure 5 visually contrasts a healthy inference step with an anomalous one. In Rank 1 (top), the execution is healthy: the main thread (Thread 1) holds the GIL and executes the `model_forward` logic without interruption, continuously launching GPU kernels (visualized as arrows triggering kernel execution). In contrast, Rank 2 (bottom) exhibits a severe performance degradation. The `model_forward` span is significantly elongated, containing a large "Blocked" gap where no kernels are launched. Without `py-spy`, we can only see a slow `model_forward`, misidentifying it as a computation bottleneck. However, `py-spy` can reveal the true culprit. A background thread (*i.e.,* Thread 2) acquires the GIL, transitioning to the *active+gil* state. Consequently, Thread 1 is forced into an *idle* state, blocked from executing Python bytecode required for kernel launching. As a result, this correlation unambiguously attributes the stall to background thread interference.

5.2.3 Minimizing Runtime Overhead

Having identified the strategic instrumentation points, the subsequent challenge lies in efficiently exporting the telemetry data for further analysis. We initially evaluated vLLM's

[1]Token status encapsulates the input prompt length and the dynamically accumulated decoded sequence length for each request within the batch.

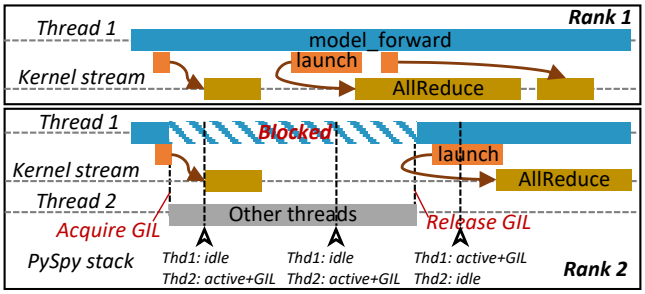
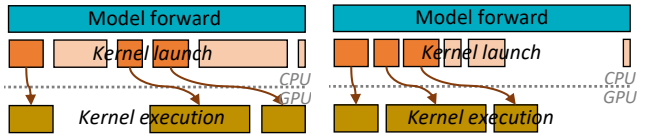


Figure 5: GIL contention between a background thread and the main inference thread.



(a) CPU-bound. (b) GPU-bound.

Figure 6: Two canonical performance regimes.

built-in OpenTelemetry (OTel) [29]. As a widely adopted industry standard, OTel defines a unified trace data schema and provides mature middleware for data transmission.

However, practical deployment revealed that the standard OTel Python SDK introduces unacceptable latency jitter. The root cause stems from its *asynchronous batch export mechanism*. Specifically, OTel buffers trace data and spawns a background thread to export the batch via HTTP or gRPC once the buffer fills. Crucially, the serialization (*e.g.,* Protobuf encoding) required for transmission is CPU-intensive and must hold the Python GIL. Consequently, when the background export triggers, it contends with the main inference thread, causing blocking stalls that can exceed 100 ms.

To eliminate this non-deterministic interference, StriaTrace abandons the background export model in favor of a deterministic I/O hiding strategy. Instead of passively waiting for a batch to fill, StriaTrace proactively flushes trace data at precise execution boundaries. Specifically, we exploit the inherent parallelism between the host and the GPU. In vLLM, after the GPU worker issues asynchronous kernel launches, the CPU typically waits for GPU execution to complete. StriaTrace leverages this CPU idle window to perform data export. Hence, StriaTrace can effectively hide the tracing overhead behind the GPU execution shadow, ensuring negligible impact on the critical path.

5.3 Device-side Tracing

5.3.1 Hardware-Centric Instrumentation

Standard profiling solutions within the vLLM ecosystem include Nvidia Nsight Systems and PyTorch Profiler. However, neither is viable for always-on production monitoring due

to prohibitive overhead. The fundamental issue stems from their indiscriminate subscription to the underlying CUPTI (CUDA Profiling Tools Interface) [26] events, the proprietary observability library provided by NVIDIA.

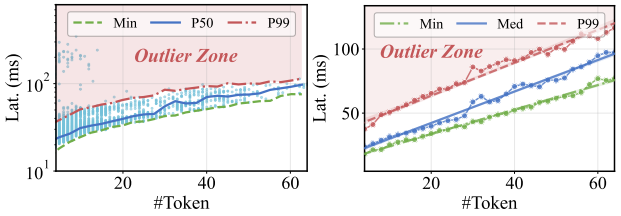
CUPTI provides visibility across the full GPU stack: from software-level CUDA Runtime and Driver APIs to hardware-level kernel execution. Existing tools typically subscribe to all these layers to ensure completeness. In vLLM, however, a single kernel execution is often preceded by a cascade of Runtime and Driver API calls (*e.g.,* argument setup and launch configuration). Capturing this high-frequency API traffic introduces substantial overhead. For instance, TorchProfiler incurs a latency penalty of ∼192%, as it traces both the API calls and the hardware execution.

StriaTrace challenges this "trace-all" convention (following **Principle 2**). We observe that hardware-level information alone is sufficient to reconstruct the critical path of inference. Hence, StriaTrace eschews the tracing of software API calls, focusing on the start and end timestamps of GPU kernels and memory transfers. This design choice drastically reduces overhead while preserving the necessary fidelity for bottleneck analysis. As illustrated in Figure 6, the density of kernel events along the timeline acts as a proxy for the system's bottleneck state: (1) **Scenario 1: CPU-bound (sparse kernels).** As shown in Figure 6a, when the host-side kernel launch is slower than GPU execution, the GPU remains mostly idle, resulting in sparse kernel events. In other words, the bubbles directly reflect host-side bottlenecks, rendering API tracing redundant. (2) **Scenario 2: GPU-bound (packed kernels).** Conversely, as shown in Figure 6b, when kernel execution is slower than the CPU launch, the timeline shows densely packed kernels. The inference engine ultimately stalls waiting for the GPU to complete. Here, the critical path is unambiguously governed by the kernel execution.

In summary, by recording only hardware execution intervals, StriaTrace can unambiguously distinguish between CPU and GPU bottlenecks without paying the overhead of API interception.

5.3.2 Anomaly-Driven Data Retention

While the hardware-centric strategy (§5.3.1) can mitigate runtime interference, it introduces a new challenge. In the vLLM framework, a single inference step typically involves the launch of thousands of GPU kernels. Consequently, recording the complete trace of these events generates a massive telemetry footprint. To quantify this, we profile a vLLM instance serving Qwen3-Coder-30B-Instruct-FP8 [44]. Each GPU worker generates approximately 1,600 kernel records per inference step. With an average step latency of 12 ms, an 8-GPU node produces raw trace data at a rate of 3.54 GiB/min. Extending this to production clusters of 10,000 GPUs, the aggregate bandwidth demand can surge to 590 Gbps. Transmitting and storing this full-fidelity stream is operationally prohibitive.



(a) Latency distribution. (b) Roofline modeling.

Figure 7: Construction of the StriaTrace roofline. (a) Latency distribution versus token count. (b) Modeling the P99 performance boundary via linear regression.

To address this massive footprint, StriaTrace shifts from a "store-everything" to a "store-on-anomaly" paradigm (following **Principle 3**). For production monitoring, detailed GPU kernel traces are redundant for normal inference steps. Consequently, StriaTrace discards kernel traces for normal steps by default, retaining only aggregated metrics. Full-fidelity traces are persisted exclusively when the step is flagged as an outlier by the roofline model (details in §5.4). This anomaly-driven retention strategy reduces the transmission volume to approximately 1.6% of the raw stream.

5.4 Trace Analysis

5.4.1 Adaptive Roofline for Anomaly Detection

To detect performance anomalies, StriaTrace must establish a reliable performance baseline. Simple approaches, such as fixed latency thresholds, fail in this context because the execution time of an inference step varies significantly with its workload (*e.g.,* the number of tokens processed). Consequently, a fixed threshold cannot distinguish between a naturally heavy computation and a true performance degradation.

While theoretical roofline modeling [45, 47, 49] offers a dynamic baseline based on hardware and software configurations, applying it to production environments is operationally intractable. Modern deployments involve a combinatorial variety of model architectures, GPU specifications, parallelization strategies, and serving paradigms (*e.g.,* prefill/decode disaggregation). As these innovations continuously evolve, pre-computing analytical models for every configuration permutation is infeasible.

Instead, StriaTrace employs a data-driven regression approach to construct empirical rooflines tailored to each serving instance. This design is grounded in a key empirical observation: *Inference latency exhibits a strong positive correlation with the number of processed tokens*[2]. This statistical relationship allows us to treat anomaly detection as a regression problem, learning a direct mapping from the token count to

[2]For a *Prefill* instance, this counts the sum of all prompt lengths in the batch; for a *Decode* instance, it equals the batch size.

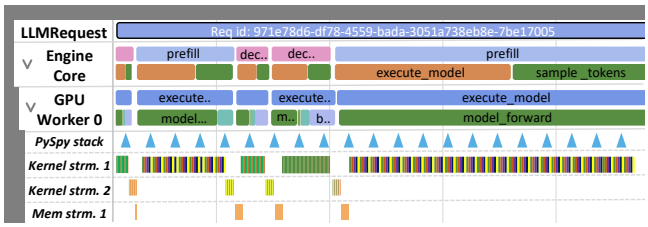


Figure 8: End-to-end trace visualization in the Perfetto UI.

the expected tail latency. Based on this observation, StriaTrace formulates the roofline construction as a data-driven process. The core objective is to determine a dynamic upper-bound for the inference latency, *i.e.,* the *empirical roofline*, by modeling the tail distribution of historical execution data. Figure 7 visualizes this modeling workflow using empirical data captured from a production instance.

Figure 7a plots the distribution of step latency against the number of processed tokens. For each token count (x-axis), we calculate the minimum, median, and P99 latencies from historical data. Two key patterns emerge from Figure 7a. First, the minimum, median, and P99 latency exhibit a strong positive correlation with the token count (x-axis). Second, a subset of samples is distributed far beyond the P99 boundary. StriaTrace defines this high-latency region as the *Outlier Zone*, identifying these specific samples as performance anomalies.

Building on these insights, Figure 7b demonstrates the modeling process. We fit linear regression models to the statistical boundaries derived in Figure 7a. The results show that the P99 latency scales linearly with the token count, achieving an R^2 score of 0.98[3]. StriaTrace adopts this P99 regression line as the empirical roofline. This design allows us to instantiate reliable anomaly detection thresholds using lightweight sampling and robust linear fitting (*e.g.,* RANSAC [12]).

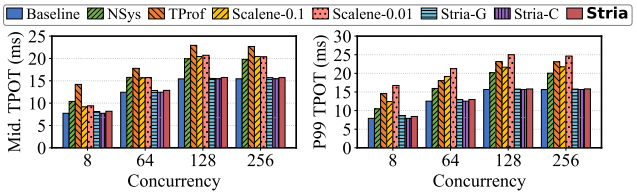
StriaTrace bootstraps the roofline model during the initial serving phase. It collects the token count and latency of each inference step until sufficient data points are accumulated for linear fitting. Subsequently, the system periodically retrains the model during runtime. Furthermore, the construction of the *Prefill* roofline follows this identical "collect-fit-update" paradigm shown in Figure 7.

5.4.2 Hierarchical Analysis and Reporting

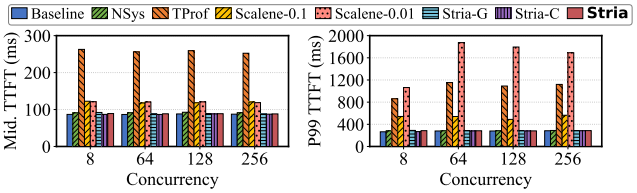
Upon detecting a performance anomaly via the roofline model, StriaTrace automatically generates an *anomaly report* for the site reliability engineers (SREs). The anomaly report integrates two components to accelerate diagnosis: *hierarchical telemetry data* and *primary bottleneck suspects*. To facilitate drill-down analysis, StriaTrace organizes the execution context into three semantic layers, mapped to a unified timeline in the Perfetto UI as shown in Figure 8: (1) *Request layer* visualizes the end-to-end lifecycle, identifying queuing delays and overall latency violations; (2) *Framework layer* exposes vLLM internal behaviors, such as scheduler decisions, KV cache block allocation, and worker coordination events. (3) *Kernel layer* provides hardware execution details, visualizing individual GPU kernel executions (*e.g.,* GEMM, Attention) and PCIe/NVLink transfers.

Beyond hierarchical telemetry, StriaTrace performs correlation analysis to highlight primary suspects. Specifically, StriaTrace identifies the dominant span, *i.e.,* the span contributing most to the latency, and correlates it with the `py-spy` trace to pinpoint the executed function, alongside the GIL status to reveal thread scheduling contention. It also flags outliers among GPU kernels. We further illustrate two representative cases in §7.

[3] An R^2 of 0.98 implies that our linear model accounts for 98% of the latency variability, demonstrating exceptional predictive accuracy.



(a) TPOT statistics collected from the decode instance.



(b) TTFT statistics collected from the prefill instance.

Figure 9: Profiling overhead comparison.

Table 3: Trace data generation rates (GiB/h).

Setup	NSys	TProf	Scalene	Stria-C	Stria-G	**Stria**
Prefill	123.7	489.2	∼0	3.11	0.08	**3.19**
Decode	1003.2	274.9	∼0	0.36	2.39	**2.75**

6 Evaluation

We evaluate StriaTrace to answer the following questions:

- Can StriaTrace operate with negligible performance impact on production vLLM serving, while minimizing the telemetry footprint?
- Is StriaTrace effective in detecting performance anomalies and pinpointing their root causes?
- What concrete impact has StriaTrace had on diagnosing performance anomalies in real-world production?

6.1 Setup

Testbed. Our experiments are conducted on a production-grade node. The node is equipped with dual-socket Intel Xeon processors (160 cores total) and 1.3 TiB of DDR5 memory. The server hosts eight NVIDIA H20-3e GPUs (144 GiB memory per GPU) interconnected via NVLink. The system runs Ubuntu 24.04.1 LTS. Our software stack consists of Python 3.12.3, PyTorch 2.7.0, and CUDA 12.8, using vLLM as the inference engine.

Comparisons. We evaluate vanilla vLLM (*Baseline*) against six instrumentation configurations: (1) *NVIDIA Nsight Systems (NSys)*: To control the overhead, we disable the CPU backtracing (see details in Appendix B.1); (2) *PyTorch Profiler (TProf)*: We use the built-in TProf implementation embedded within vLLM (see Appendix B.2); (3) *Scalene*: We configure Scalene with two distinct CPU sampling intervals (*i.e.,* 10 ms and 100 ms, see details in Appendix B.3); (4) *Stria-C:* StriaTrace with only CPU-side tracing enabled; (5) *Stria-G*: StriaTrace with only GPU-side profiling enabled; (6) *Stria*: The complete StriaTrace system with both CPU and GPU instrumentation active.

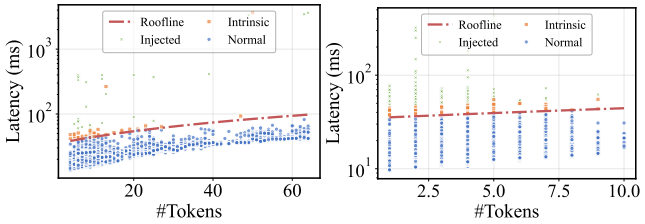
Model and workload. We evaluate StriaTrace using Qwen3-Coder-30B-Instruct-FP8 [44], a state-of-the-art Mixture-of-Experts (MoE) code generation model optimized with FP8 quantization. To strictly mirror our production environment, we deploy this model using a PD disaggregated architecture. We deploy two distinct instances with both instances configured at a Tensor Parallelism degree of 8 (TP=8). These two instances are seamlessly interconnected via our custom distributed KV cache component. For workload generation, we utilize vLLM's native benchmark tool [46] to issue requests with fixed input/output lengths, following a Poisson arrival process at specified rates.

6.2 Overhead Analysis

We evaluate the operational cost of StriaTrace along two dimensions: (1) the *latency penalty* to TPOT and TTFT, and (2) the *telemetry footprint* in terms of storage consumption.

Impact on inference latency. We configure the vLLM benchmark with a fixed request arrival rate of 8 req/s while sweeping the engine's maximum concurrency from 8 to 256. To accurately reflect the performance of the disaggregated architecture, we report the Time-to-First-Token (TTFT) measured at the prefill instance, and the Time-Per-Output-Token (TPOT) measured at the decode instance.

Figure 9a details the TPOT results for the decoding instance. We make three key observations: (1) Existing tools are costly. At a low concurrency of 8, *TProf* incurs prohibitive penalties, inflating the Median and P99 TPOT by 83.6% and 84.1%, respectively. While *NSys* is lighter, it still imposes a non-negligible overhead of 34.3% (Median) and 32.0% (P99). *Scalene-0.1* (default 10 ms sampling interval) also introduces substantial TPOT penalties for both Median (21.4%) and P99 (111.8%). Even when configured with a sparser 100 ms interval, the overhead remains at 18.7% (Median) and 56.9% (P99). (2) StriaTrace's host-side tracing introduces negligible overhead. *Stria-C* introduces virtually no performance regression; the overhead for both Median and P99 TPOT remains strictly below 1% across all settings; (3) GPU overhead amortizes. The cost of full StriaTrace is primarily driven by GPU instrumentation, but this overhead scales inversely with throughput. As the concurrency increases from 8 to 256, the instrumentation cost is effectively amortized, causing the Median TPOT overhead to drop from 5.2% to 0.6%.



(a) Host-side stall. (b) Device-side contention.

Figure 10: Roofline visualization classifying *Injected*, *Intrinsic*, and *Normal* steps under (a) host-side stall and (b) device-side contention scenarios.

Figure 9b depicts the impact on TTFT collected from the prefill instance. Similar to the TPOT results, we observe distinct overhead patterns: (1) *TProf* remains prohibitively heavy. It introduces substantial latency regression, increasing the Median and P99 TTFT by 3.0× and 3.2× over the baseline (at 8 concurrency). (2) *NSys* incurs moderate overhead. Although lighter than *TProf*, it still inflates the Median TTFT by 4.7% and the P99 by 6.3%. (3) StriaTrace demonstrates effective amortization. Consistent with its behavior on TPOT, StriaTrace's overhead on TTFT diminishes as the workload increases. As the concurrency climbs from 8 to 256, the overhead is amortized, dropping from 2.7% to 0.8% (measured in Median).

Tracing data footprint. We measure the rate of tracing data generation (GiB/hour) under a fixed workload of 8 req/s across all instrumentation setups. Table 3 details the tracing throughput for both prefill and decode instances. The results clearly indicate that existing tools incur a substantial data tax. Specifically, for the decode instance, the consumed bandwidth of *NSys* and *TProf* exceeds that of StriaTrace by at least 38×. In contrast, StriaTrace maintains a modest data footprint, making it viable for continuous production tracing.

6.3 Effectiveness of Anomaly Detection

We next evaluate the efficacy of StriaTrace's roofline model in detecting performance outliers. Specifically, we conduct a controlled fault injection campaign on a distributed vLLM cluster (TP=4). After allowing StriaTrace to construct a sta-

Table 4: Abnormalities diagnosed by StriaTrace.

Category	Root Cause	Diagnosis Result
Hardware	1.GPU clock throttling	General GPU*
	2.High GPU temperature	General GPU
	3.Unstable CPU frequency	Random CPU†
	4.CRC error	Precise‡
	5.NIC port down	Precise
	6.NVLink degradation	Precise
Config	7.HugePage misconfig.	General CPU
	8.CUDA Graph misconfig.	General CPU
	9.GPU memory misconfig.	General GPU
	10.Background process	Random CPU¶
	11.NUMA not configured	Precise
Code	12.KV store issue	Precise
	13.Token sampling issue	Precise
	14.Python GIL contention	Precise
	15.CUDA Graph replay error	Precise
	16.Kernel launch anomaly	Precise
	17.FlashAttention issue	Precise
	18.KV Store fallback to CPFS	Precise
	19.Triton JIT compilation	Precise

* General GPU-side kernel execution slow.
† General CPU-side function execution slow.
‡ Precise location of culprit code.
¶ Random CPU-side function execution slow.

ble roofline under normal traffic, we introduce two types of synthetic anomalies to mimic production failures: (1) *Host-side stall*: We temporarily suspend the execution of a vLLM worker process using the `SIGSTOP` signal; (2) *Device-side contention*: We inject high-priority background kernels that aggressively saturate GPU Streaming Multiprocessors (SMs) using Numba [25]. For each injected fault, we randomize the target worker (or GPU) and fault duration.

Figure 10a and 10b illustrate StriaTrace's roofline during the host-side stall and device-side contention experiments, respectively. Each point represents a single inference step, plotting step latency (y-axis) against token count (x-axis). Using the roofline model together with the injected-failure timestamps, we categorize each step into three classes: *Injected* (steps affected by our synthetic faults), *Intrinsic* (outliers stemming from inherent system variance), and *Normal* (healthy steps). We report the recall on injected faults as our primary metric for assessing detection efficacy. Across both fault types, StriaTrace achieves 100% recall on injected failures, indicating that all injected anomalies manifest as clear deviations from the learned roofline.

6.4 StriaTrace in Production

StriaTrace has been extensively used in the development and testing environments of our inference service, supporting multiple release cycles. Over a six-month deployment, StriaTrace monitors two of our flagship models, encompassing over 1,700 production instances and handling more than 180 million user requests daily. StriaTrace supports diverse parallelization strategies, including Tensor Parallelism (TP) and Data Parallelism (DP), and is deployed across both monolithic and PD-disaggregated architectures. StriaTrace has diagnosed hundreds of abnormalities in production, which can be categorized into three types: (1) hardware issues, (2) configuration issues, and (3) code issues. Crucially, we emphasize that StriaTrace is not designed as a fully automated, end-to-end root-cause oracle. Rather, it functions as a *rapid triage mechanism*. It is designed to provide Site Reliability Engineers (SREs) with rich, fine-grained execution context, thereby reducing the diagnostic search space.

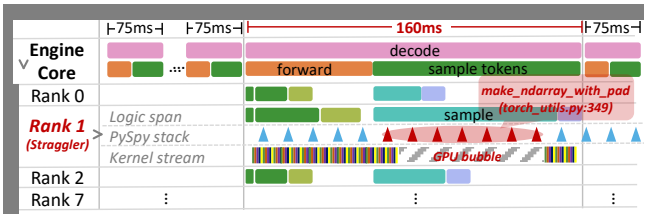
As shown in Table 4, StriaTrace approaches these production issues through two operational paradigms. First, for long-tail latency (*e.g.,* most code-related issues), the degradation explicitly triggers StriaTrace's dynamic roofline model. In these scenarios, StriaTrace executes its complete diagnostic pipeline. By correlating the anomalous semantic span with fine-grained *py-spy* snapshots, the system successfully pinpoints the problematic rank, the specific execution phase (CPU vs. GPU), and the exact line of Python code or CUDA kernel. We detail this kind of case in §7.1.

Second, some anomalies manifest as sustained performance degradations that inherently do not trigger the dynamic roofline alerts. A prime example is when CUDA Graphs are inadvertently disabled (Case 8); this evades roofline detection entirely because the dynamic model constructs its baseline on an already degraded performance state. For such sustained issues, StriaTrace's primary value lies in preserving the execution context. By providing high-fidelity historical traces, StriaTrace empowers SREs with the critical telemetry needed to manually cross-reference contexts and isolate configuration or hardware flaws. We explore this telemetry-accelerated localization in §7.2.

It is important to note that StriaTrace's dynamic roofline model (§5.4.1) is not completely immune to false positives. In our production environment, approximately 7% of the flagged anomalies were eventually classified as natural workload variances rather than true system regressions. For instance, a typical false positive arises from extreme Expert Parallelism (EP) load imbalance. In MoE models, heavily skewed token routing can occasionally cause a specific GPU rank to severely straggle.

7 Case Study

This section details representative cases of diagnosing performance anomalies using StriaTrace. We categorize these cases into two paradigms: (1) *End-to-end pipeline diagnosis* (§7.1) for transient anomalies explicitly flagged by the dynamic roofline model; and (2) *Telemetry-accelerated localization* (§7.2) for broader performance issues where StriaTrace's




```python
def make_ndarray_with_pad(x: list[list[T]], ...):
    ...
    for ind, blocktb in enumerate(x):
        assert len(blocktb) <= max_len
        padded_x[ind, : len(blocktb)] = blocktb
    return padded_x
```

Figure 11: StriaTrace for a 160 ms straggler step. (Top) The end-to-end trace identifies a critical stall on GPU rank 1. (Bottom) The isolated culprit code responsible for the anomaly.

continuous lightweight observability significantly accelerates root-cause identification.

7.1 End-to-End Pipeline Diagnosis

A typical class of performance issues diagnosed by StriaTrace's end-to-end pipeline involves CPU-side logic unexpectedly blocking GPU kernel launches. To resolve these issues, StriaTrace first identifies the macroscopic bottleneck via synchronization barriers and GPU timeline bubbles, and then correlates the anomalous span with *py-spy* stack traces to pinpoint the exact culprit code. These fine-grained contexts are handed over to SREs for final confirmation and mitigation. We illustrate this workflow using a representative production case below, *i.e.,* Case 13 in Table 4.

During a routine production inspection, StriaTrace detected sporadic latency spikes in an 8-GPU decoding instance handling live traffic. As shown in Figure 11, during consecutive decode steps with a stable workload (*i.e.,* no requests joining or finishing), the step latency would occasionally jump from a baseline of 75 ms to over 160 ms. StriaTrace revealed a distributed straggler effect: one of the eight ranks (*i.e.,* Rank 1) was severely delayed, forcing the others into a synchronous stall. On Rank 1, the CPU-side *Logic span* timeline showed that the span named `sample` was abnormally prolonged, and a massive ∼80 ms bubble was observed on the *Kernel stream*. In vLLM, the `sample` is a standard routine that follows the model's forward pass to convert the output logits into token IDs.

The *PySpy stack* of Rank 1 revealed that the vLLM main inference thread (responsible for dispatching GPU kernels) was active and holding the Python Global Interpreter Lock (GIL), implying that the thread was occupied by host-side computations. Furthermore, consecutive stack samples showed the execution was persistently stalled at the `make_ndarray_with_pad` function (see Appendix C.1 for the full stack trace). This function resides in vLLM's `torch_utils.py`, a utility module designed to bridge standard Python data structures with PyTorch tensors.

Then, the SRE team uncovered a subtle memory management bottleneck. Within the token sampling phase, `make_ndarray_with_pad` acts as a conditional branch triggered when users' requests specify penalty constraints (*e.g.,* frequency or length penalties [5]). Its primary function is to pad the historical token sequences of all requests to match the maximum sequence length[4]. The bottleneck originates at line 5 within this function. The `for` loop encompassing line 5 employs native Python list slicing. When operating under a large batch size and extended token lengths, this naive approach leads to two severe consequences: (1) it generates a massive number of temporary Python objects, elevating the risk of GC pauses; and (2) it continuously triggers NumPy's underlying `malloc` and `free`.

To resolve this, the engineering team implemented a custom C++ operator and integrated it into vLLM via Torch JIT [33]. Leveraging direct C++ pointer access to preallocated contiguous memory, this custom operator iterates and pads the sequences entirely at the native C++ layer. This completely eliminates Python object creation and memory allocations. Following this patch, the P99 latency for the `make_ndarray_with_pad` invocation plummeted from 110 ms to 43 ms.

Moreover, StriaTrace's end-to-end pipeline has isolated other straggler effects. For instance, in Case 14, StriaTrace revealed that a background thread managing the KV cache mistakenly executed synchronous operations during block eviction. This implicitly acquired the Python GIL, blocking the main vLLM inference thread. Similarly, in Case 19, StriaTrace correlated an anomalous `sample` span and a massive GPU idle gap with unexpected Triton JIT compilation. The traces demonstrated that a user-defined sampling function, decorated with `torch.compile`, triggered synchronous compiler activities that consumed over 99% of the step's duration.

7.2 Telemetry-accelerated Localization

Beyond the end-to-end diagnosis in §7.1, StriaTrace's continuous observability accelerates the diagnosis of other sustained performance degradations. Below, we present a case illustrating how SREs leverage the fine-grained correlation between StriaTrace's GPU kernel traces and the CPU-side traces to pinpoint the root cause, *i.e.,* Case 17 in Table 4.

During a routine weekly regression test, we observed an unexpected drop in decode token throughput compared to the previous stable release. Since over 400 new commits had been merged that week[5], isolating the root cause via traditional `git bisect` was practically prohibitive. The initial

[4] Sequence length refers to the sum of the initial prompt tokens and the tokens decoded so far.

[5] The weekly average number of commits in our private vLLM repository over the past year is 340.

analysis revealed that, under an identical decode workload, the average decode step latency had inflated from roughly 5 ms to 7 ms. Drilling down into StriaTrace's CPU-side traces, we found that the span named `model_forward` was the sole contributor to this latency increase. In vLLM, this specific span encompasses the critical phase where the GPU worker utilizes CUDA graphs to dispatch kernels for the forward pass.

To pinpoint the exact cause, engineers examined StriaTrace's GPU kernel traces. The data revealed that the GPU execution time inflation was almost entirely attributed to a specific kernel, `16FlashAttnFwdSm90INS1`. This is the critical FlashAttention kernel, invoked dozens of times per decode step to perform the attention forward pass. Crucially, StriaTrace's GPU tracing captures the *mangled name* of each executed kernel. This mangled name embeds not just the function signature, but its exact template instantiation parameters.

By directly comparing the mangled names across the two vLLM builds, a subtle but critical discrepancy emerged: the template arguments had shifted from $(64, 128, 128)$ to $(128, 128, 128)$ (see Appendix C.2 for the full mangled names). These parameters dictate the matrix multiplication tile size used by the GPU thread blocks. The engineering team verified via offline reproduction that on NVIDIA H20 GPUs, the smaller tile size is more optimal, executing in roughly $140\,\mu$s per invocation compared to $210\,\mu$s for the larger size (see Appendix C.2 for detailed settings). Since this kernel executes repeatedly in a decode step, this $70\,\mu$s penalty per invocation accumulated directly into the observed degradation. The fundamental root cause was eventually traced back to a faulty build script. The new vLLM version was shipped with a FlashAttention library compiled with missing template configurations. Consequently, the optimal kernel variant with the 64 tile size was never generated during the compilation. Without the optimal binary available, the FlashAttention runtime silently fell back to dispatching the sub-optimal 128 tile size variant.

Furthermore, StriaTrace's CPU-side telemetry has proven instrumental in diagnosing obscure corner cases. For example, during a recent release rollout, the engineering team reported another unexpected drop in decode token throughput. StriaTrace revealed that *the number of draft tokens proposed by the draft model was strictly less than the number of tokens actually submitted for verification in the subsequent step*. This exposed the silent injection of "ghost tokens", immediately bounding the anomaly to the speculative decoding workflow. Further investigation confirmed this was a secondary bug: a patch intended to resolve a specific EAGLE-3 [19] issue compromised the performance of the N-gram [48] algorithm (detailed analysis in Appendix C.3).

7.3 Experiences & Lessons

Necessity of continuous observability. The rapid development cycles of inference engines dictate that the CI/CD cannot anticipate all performance regressions. Production deployments of vLLM involve a massive combinatorial space of parameters, topologies, and distributed components. Coupled with continuous codebase updates, this sheer complexity makes exhaustive testing in CI/CD practically impossible. Consequently, continuous, *in-vivo* observability is a highly recommended capability for capturing dynamic, workload-dependent anomalies that inevitably emerge in production.

Synergy of cross-layer telemetry. Diagnosing complex anomalies requires both CPU-side semantic context and GPU-side hardware execution details. On the CPU-side, tracking CPU runtime metadata (*e.g.,* dynamic user request parameters and control data) is valuable for isolating logic-level and scheduling bottlenecks. Complementarily, on the GPU-side, extracting mangled kernel names—which inherently encode critical compilation configurations—plays a pivotal role in diagnosing hardware-level anomalies. Correlating these independent dimensions creates a powerful cross-layer synergy that accelerates full-stack root-cause analysis.

Strict hot-path disciplines. In high-throughput inference engines like vLLM, the main inference thread serves as the critical hot path. Any CPU-side stall on this thread directly translates to GPU starvation. Consequently, we recommend that developers follow these guidelines: (1) *Offloading memory-intensive operations to C/C++*: Native Python memory management is inherently opaque. Operations that generate massive temporary objects, such as dynamic list slicing or sequence padding, are highly prone to unpredictable GC pauses. To guarantee latency stability, these logic blocks should be rigorously offloaded to native C/C++ implementations; (2) *Minimizing synchronous interruptions:* Synchronous operations should be introduced with extreme caution. Synchronous calls—such as querying a distributed KV cache, polling remote storage, or waiting on background threads—can inadvertently block the main event loop.

8 Discussion

Tracing coverage vs. overhead. StriaTrace does not blindly sacrifice tracing coverage for low overhead. We discuss this trade-off across the CPU and GPU domains. For CPU tracing, StriaTrace avoids pervasive instrumentation by a hybrid approach. First, it relies on static tracing points embedded in vLLM's critical path. As shown in Table 2, these points cover less than 1% of the total functions. Second, to provide full-stack visibility, StriaTrace employs periodic sampling via *py-spy*. The full-stack sampler remains effective for our target anomalies, *i.e.,*, long-tail latency. Diagnosing sustained performance degradations (*e.g.,* cases in §7.2) presents a more nuanced challenge, as low-frequency sampling might miss

micro-stalls. However, since such sustained issues are highly reproducible, StriaTrace can serve as an initial triage anchor. By preserving the execution scene, StriaTrace narrows the search space, allowing SREs to iteratively inject instrumentation points offline to pinpoint the exact culprit.

On the GPU, StriaTrace is confined to kernel executions. Unlike exhaustive profilers such as *NSys*, we deliberately disable the tracing of CUDA driver and runtime APIs. This is a necessary and unavoidable compromise. Since GPU tracing heavily relies on the closed-source CUPTI [26], tracking high-frequency driver/runtime API calls introduces an untenable overhead. To compensate for this reduced coverage, StriaTrace infers host-to-device interaction bottlenecks heuristically. By analyzing the idle "bubbles" between kernels, SREs can effectively deduce whether a latency anomaly stems from CPU-side CUDA driver overheads or actual GPU anomalies.

Architectural generality and porting effort. In StriaTrace's design, both the CPU-side full-stack sampling via *py-spy* (§5.2.2) and the GPU-side hardware tracing (§5.3.1) are strictly framework-agnostic. These components are applicable to any Python-based inference engine (*e.g.,* vLLM [46], SGLang [40]). The only framework-dependent component—and thus the primary source of engineering effort when porting to a new framework or maintaining across version upgrades—lies in the coarse-grained CPU-side instrumentation (§5.2.1). We emphasize that the manual effort is bounded and manageable. First, regarding cross-framework portability, StriaTrace's instrumentation relies on identifying synchronization barriers along the engine's critical path. Since these points have distinct signatures or structural patterns, they can be efficiently extracted through static code analysis facilitated by modern AI coding agents. Second, regarding temporal maintainability, the critical path of a mature inference engine is inherently stable. While minor feature updates are frequent, the core execution workflow rarely updates. During our six-month production deployment tracking upstream vLLM, the critical path experienced only one major shift, *i.e.,* the transition from synchronous to asynchronous scheduling. Consequently, adapting StriaTrace to engine upgrades requires acceptable effort.

Cross-platform compatibility. Currently, StriaTrace relies on NVIDIA's CUPTI for GPU observability, given that NVIDIA GPUs remain the dominant platform in production-grade LLM serving. However, we acknowledge the growing adoption of alternative accelerators, such as AMD GPUs utilizing the ROCm stack. Porting StriaTrace to AMD GPUs is straightforward due to architectural symmetry. This is facilitated by the availability of ROCTracer [1], an AMD observability tool designed as a near-exact architectural mirror of CUPTI. First, it exposes a standard C library interface, allowing for seamless integration into StriaTrace's GPU tracing module. Second, it reports kernel execution timestamps identical to those of CUPTI, ensuring that the downstream backend analysis logic remains unchanged.

9 Related Work

End-to-end inference profiling. Several frameworks have proposed end-to-end analysis systems for inference scenarios. LLM-Pilot [18] measures inference performance under varying hardware configurations and builds predictive models to estimate performance in new deployment settings. However, it only collects coarse-grained per-request execution time. The ALA framework [2] constructs an analytical model based on collected runtime metrics to perform theoretical analysis and anomaly detection for inference systems. Similarly, ALA relies solely on high-level system metrics.

Training-centric diagnosis. With the rapid evolution of LLMs, there have been numerous studies focusing on locating and diagnosing abnormalities during LLM training [3, 7–9, 16, 21, 22]. However, all these solutions rely on the static training patterns and are ill-suited for dynamic inference workloads. For example, these solutions leverage global barriers in Bulk Synchronous Parallel (BSP) training to distinguish abnormal iterations. In contrast, inference instances process requests independently and asynchronously.

10 Conclusion

In this paper, we present StriaTrace, a tracing and diagnosis system tailored for online LLM inference. StriaTrace employs three principles distilled from production experience to minimize the tracing overhead. Furthermore, StriaTrace introduces a step-level roofline and correlation-based diagnosis for LLM inference abnormalities. StriaTrace reduces tracing overhead by 97.8% compared to state-of-the-art alternatives, and has successfully diagnosed 19 types of issues in production.

Acknowledgments

We sincerely thank our shepherd Hongqiang Harry Liu and the anonymous reviewers for helping us improve our paper significantly. We acknowledge all teams within Alibaba that contributed to the deployment of StriaTrace. This work is supported by Alibaba Group through Alibaba Innovative Research Program and Alibaba Research Intern Program, National Natural Science Foundation of China (NSFC) under Grant No. 62072306, and State Key Laboratory of Digital Finance (In Preparation).

References

[1] AMD. Roctracer documentation, 2025. https://rocm.docs.amd.com/projects/roctracer/en/latest/.

[2] Anonymous. ALA: Analytical Latency Analysis Framework, 2025. Private communication.

[3] AWS. Machine Learning Service - Amazon SageMaker, 2024. https://aws.amazon.com/pm/sagemaker/?nc1=h_ls.

[4] Emery D. Berger, Sam Stern, and Juan Altmayer Pizzorno. Triangulating python performance issues with SCALENE. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*, pages 51–64, Boston, MA, July 2023. USENIX Association.

[5] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arun Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. Language models are few-shot learners. *NeurIPS*, 2020.

[6] DeepSeek-AI, Aixin Liu, Bei Feng, Bing Xue, Bingxuan Wang, Bochao Wu, Chengda Lu, Chenggang Zhao, Chengqi Deng, Chenyu Zhang, et al. Deepseek-v3 technical report, 2025.

[7] Yangtao Deng, Xiang Shi, Zhuo Jiang, Xingjian Zhang, Lei Zhang, Zhang Zhang, Bo Li, Zuquan Song, Hang Zhu, Gaohong Liu, Fuliang Li, Shuguang Wang, Haibin Lin, Jianxi Ye, and Minlan Yu. Minder: Faulty machine detection for large-scale distributed model training. In *22nd USENIX Symposium on Networked Systems Design and Implementation (NSDI 25)*, pages 505–521, Philadelphia, PA, April 2025. USENIX Association.

[8] Yangtao Deng, Lei Zhang, Qinlong Wang, Xiaoyun Zhi, Xinlei Zhang, Zhuo Jiang, Haohan Xu, Lei Wang, Zuquan Song, Gaohong Liu, Yang Bai, Shuguang Wang, Wencong Xiao, Jianxi Ye, Minlan Yu, and Hong Xu. Mycroft: Tracing dependencies in collective communication towards reliable llm training. In *Proceedings of the ACM SIGOPS 31st Symposium on Operating Systems Principles*, SOSP '25, page 254–269, New York, NY, USA, 2025. Association for Computing Machinery.

[9] Jianbo Dong, Kun Qian, Pengcheng Zhang, Zhilong Zheng, Liang Chen, Fei Feng, Yichi Xu, Yikai Zhu, Gang Lu, Xue Li, Zhihui Ren, Zhicheng Wang, Bin Luo, Peng Zhang, Yang Liu, Yanqing Chen, Yu Guan, Weicheng Wang, Chaojie Yang, Yang Zhang, Man Yuan, Hanyu Zhao, Yong Li, Zihan Zhao, Shan Li, Xianlong Zeng, Zhiping Yao, Binzhang Fu, Ennan Zhai, Wei Lin, Chao Wang, and Dennis Cai. Evolution of aegis: Fault diagnosis for AI model training service in production. In *22nd USENIX Symposium on Networked Systems Design and Implementation (NSDI 25)*, pages 865–881, Philadelphia, PA, April 2025. USENIX Association.

[10] eBPF. Dynamically program the kernel for efficient networking, observability, tracing, and security, 2025. https://ebpf.io/.

[11] Xiaoning Feng, Xiaohong Han, Simin Chen, and Wei Yang. Llmeffichecker: Understanding and testing efficiency degradation of large language models. *ACM Trans. Softw. Eng. Methodol.*, 33(7), August 2024.

[12] Martin A. Fischler and Robert C. Bolles. Random sample consensus: a paradigm for model fitting with applications to image analysis and automated cartography. *Commun. ACM*, 24(6):381–395, June 1981.

[13] Yu Guan, Zhiyu Yin, Haoyu Chen, Sheng Cheng, Chaojie Yang, Kun Qian, Tianyin Xu, Yang Zhang, Hanyu Zhao, Yong Li, Wei Lin, Dennis Cai, and Ennan Zhai. Perftracker: Online performance troubleshooting for large-scale model training in production, 2025.

[14] Songlin Huang and Chenshu Wu. Neutrino: fine-grained gpu kernel profiling via programmable probing. In *Proceedings of the 19th USENIX Conference on Operating Systems Design and Implementation*, OSDI '25, USA, 2025. USENIX Association.

[15] Yanping Huang, Youlong Cheng, Ankur Bapna, Orhan Firat, Mia Xu Chen, Dehao Chen, HyoukJoong Lee, Jiquan Ngiam, Quoc V. Le, Yonghui Wu, and Zhifeng Chen. *GPipe: efficient training of giant neural networks using pipeline parallelism*. Curran Associates Inc., Red Hook, NY, USA, 2019.

[16] Ziheng Jiang, Haibin Lin, Yinmin Zhong, Qi Huang, Yangrui Chen, Zhi Zhang, Yanghua Peng, Xiang Li, Cong Xie, Shibiao Nong, Yulu Jia, Sun He, Hongmin Chen, Zhihao Bai, Qi Hou, Shipeng Yan, Ding Zhou, Yiyao Sheng, Zhuo Jiang, Haohan Xu, Haoran Wei, Zhang Zhang, Pengfei Nie, Leqi Zou, Sida Zhao, Liang Xiang, Zherui Liu, Zhe Li, Xiaoying Jia, Jianxi Ye, Xin Jin, and Xin Liu. MegaScale: Scaling large language model training to more than 10,000 GPUs. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*, pages 745–760, Santa Clara, CA, April 2024. USENIX Association.

[17] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph Gonzalez, Hao Zhang, and Ion Stoica. Efficient memory management for large language model serving with pagedattention. In *Proceedings of the 29th Symposium on Operating Systems Principles*, SOSP '23, page 611–626,

New York, NY, USA, 2023. Association for Computing Machinery.

[18] Malgorzata Lazuka, Andreea Anghel, and Thomas Parnell. Llm-pilot: Characterize and optimize performance of your llm inference services. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis*, SC '24. IEEE Press, 2024.

[19] Yuhui Li, Fangyun Wei, Chao Zhang, and Hongyang Zhang. Eagle-3: Scaling up inference acceleration of large language models via training-time test. *arXiv preprint arXiv:2503.01840*, 2025.

[20] Yuhan Liu, Yihua Cheng, Jiayi Yao, Yuwei An, Xiaokun Chen, Shaoting Feng, Yuyang Huang, Samuel Shen, Rui Zhang, Kuntai Du, and Junchen Jiang. Lmcache: An efficient kv cache layer for enterprise-scale llm inference, 2025.

[21] Meta. OPT-175 Logbook, 2023. https://github.com/facebookresearch/metaseq/blob/main/projects/OPT/chronicles/OPT175B_Logbook.pdf.

[22] Meta. Dynolog: a performance monitoring daemon for heterogeneous CPU-GPU systems, 2024. https://github.com/facebookincubator/dynolog.

[23] Deepak Narayanan, Aaron Harlap, Amar Phanishayee, Vivek Seshadri, Nikhil R. Devanur, Gregory R. Ganger, Phillip B. Gibbons, and Matei Zaharia. Pipedream: generalized pipeline parallelism for dnn training. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP '19, page 1–15, New York, NY, USA, 2019. Association for Computing Machinery.

[24] Deepak Narayanan, Mohammad Shoeybi, Jared Casper, Patrick LeGresley, Mostofa Patwary, Vijay Korthikanti, Dmitri Vainbrand, Prethvi Kashinkunti, Julie Bernauer, Bryan Catanzaro, Amar Phanishayee, and Matei Zaharia. Efficient large-scale language model training on gpu clusters using megatron-lm. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '21, New York, NY, USA, 2021. Association for Computing Machinery.

[25] Numba. Numba: A High Performance Python Compiler., 2025. https://numba.pydata.org/.

[26] NVIDIA. NVIDIA CUDA Profiling Tools Interface (CUPTI) - CUDA Toolkit, 2025. https://developer.nvidia.com/cupti.

[27] NVIDIA. NVIDIA Nsight Systems, 2025. https://developer.nvidia.com/nsight-systems.

[28] NVIDIA. TensorRT-LLM: TensorRT LLM provides users with an easy-to-use Python API to define Large Language Models (LLMs) and supports state-of-the-art optimizations to perform inference efficiently on NVIDIA GPUs. TensorRT LLM also contains components to create Python and C++ runtimes that orchestrate the inference execution in a performant way., 2025. https://github.com/NVIDIA/TensorRT-LLM.

[29] OpenTelemetry. OpenTelemetry eBPF Instrumentation, 2025. https://opentelemetry.io/docs/zero-code/obi/.

[30] py spy. py-spy: Sampling profiler for Python programs, 2025. https://github.com/benfred/py-spy.

[31] PyStack. PyStack, 2025. https://bloomberg.github.io/pystack/.

[32] Python. The Python Profilers., 2025. https://docs.python.org/3/library/profile.html.

[33] PyTorch. JIT, 2025. https://residentmario.github.io/pytorch-training-performance-guide/jit.html.

[34] PyTorch. Kineto, 2025. https://github.com/pytorch/kineto.

[35] PyTorch. torch.profiler, 2025. https://docs.pytorch.org/docs/stable/profiler.html.

[36] Penghui Qi, Xinyi Wan, Guangxing Huang, and Min Lin. Zero bubble (almost) pipeline parallelism. In *The Twelfth International Conference on Learning Representations*, 2024.

[37] Ruoyu Qin, Zheming Li, Weiran He, Jialei Cui, Feng Ren, Mingxing Zhang, Yongwei Wu, Weimin Zheng, and Xinran Xu. Mooncake: Trading more storage for less computation — a KVCache-centric architecture for serving LLM chatbot. In *23rd USENIX Conference on File and Storage Technologies (FAST 25)*, pages 155–170, Santa Clara, CA, February 2025. USENIX Association.

[38] Samyam Rajbhandari, Conglong Li, Zhewei Yao, Minjia Zhang, Reza Yazdani Aminabadi, Ammar Ahmad Awan, Jeff Rasley, and Yuxiong He. Deepspeed-moe: Advancing mixture-of-experts inference and training to power next-generation ai scale, 2022.

[39] Samyam Rajbhandari, Jeff Rasley, Olatunji Ruwase, and Yuxiong He. Zero: memory optimizations toward training trillion parameter models. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '20. IEEE Press, 2020.

[40] sglang. SGLang, 2025. https://github.com/sgl-project/sglang.

[41] Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. Megatron-lm: Training multi-billion parameter language models using model parallelism, 2020.

[42] Biao Sun, Ziming Huang, Hanyu Zhao, Wencong Xiao, Xinyi Zhang, Yong Li, and Wei Lin. Llumnix: dynamic scheduling for large language model serving. In *Proceedings of the 18th USENIX Conference on Operating Systems Design and Implementation*, OSDI'24, USA, 2024. USENIX Association.

[43] Qwen Team. Qwen2.5: A party of foundation models, September 2024.

[44] Qwen Team. Qwen3 technical report, 2025.

[45] Marian Verhelst, Luca Benini, and Naveen Verma. How to keep pushing ml accelerator performance? know your rooflines! *IEEE Journal of Solid-State Circuits*, 60(6):1888–1905, 2025.

[46] vLLM. Easy, fast, and cheap LLM serving for everyone, 2025. https://github.com/vllm-project/vllm.

[47] Yunsong Wang, Charlene Yang, Steven Farrell, Yan Zhang, Thorsten Kurth, and Samuel Williams. Time-based roofline for deep learning performance analysis. In *2020 IEEE/ACM Fourth Workshop on Deep Learning on Supercomputers (DLS)*, pages 10–19, 2020.

[48] Wikipedia. Word n-gram language model, 2026. https://en.wikipedia.org/wiki/Word_n-gram_language_model.

[49] Zhihang Yuan, Yuzhang Shang, Yang Zhou, Zhen Dong, Zhe Zhou, Chenhao Xue, Bingzhe Wu, Zhikai Li, Qingyi Gu, Yong Jae Lee, Yan Yan, Beidi Chen, Guangyu Sun, and Kurt Keutzer. Llm inference unveiled: Survey and roofline model insights, 2024.

[50] Chen Zhang, Benyou Wang, and Dawei Song. On elastic language models. *ACM Trans. Inf. Syst.*, 42(6), October 2024.

[51] Lianmin Zheng, Zhuohan Li, Hao Zhang, Yonghao Zhuang, Zhifeng Chen, Yanping Huang, Yida Wang, Yuanzhong Xu, Danyang Zhuo, Eric P. Xing, Joseph E. Gonzalez, and Ion Stoica. Alpa: Automating inter- and Intra-Operator parallelism for distributed deep learning. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 559–578, Carlsbad, CA, July 2022. USENIX Association.

[52] Lianmin Zheng, Liangsheng Yin, Zhiqiang Xie, Chuyue Sun, Jeff Huang, Cody Hao Yu, Shiyi Cao, Christos Kozyrakis, Ion Stoica, Joseph E. Gonzalez, Clark Barrett, and Ying Sheng. Sglang: efficient execution of structured language model programs. In *Proceedings of the 38th International Conference on Neural Information Processing Systems*, NIPS '24, Red Hook, NY, USA, 2024. Curran Associates Inc.

[53] Yinmin Zhong, Shengyu Liu, Junda Chen, Jianbo Hu, Yibo Zhu, Xuanzhe Liu, Xin Jin, and Hao Zhang. Distserve: disaggregating prefill and decoding for goodput-optimized large language model serving. In *Proceedings of the 18th USENIX Conference on Operating Systems Design and Implementation*, OSDI'24, USA, 2024. USENIX Association.

APPENDIX

A Host-Side Instrumentation Points

Table 5: **Catalog of instrumented spans in StriaTrace.**

Span Name	Process	Thread
LLMEngine::Request	Client	Main
EngineCore::Step	Engine	Main
EngineCore::Schedule	Engine	Main
EngineCore::Execute	Engine	Main
EngineCore::Output	Engine	Main
EngineCore::CollectiveRpc	Engine	Main
Worker::Response	Worker	Main
ModelRunner::Preprocess	Worker	Main
ModelRunner::Forward	Worker	Main
ModelRunner::Postprocess	Worker	Main
ModelRunner::Sample	Worker	Main
ModelRunner::Draft	Worker	Main
ModelRunner::Bookkeep	Worker	Main
ModelRunner::EpLB	Worker	Main
ModelRunner::Output	Worker	Main
CacheEngine::Load/Save	Worker	Cache Mgmt.

Table 5 provides a comprehensive catalog of the trace points instrumented within the vLLM serving stack. These spans are strategically selected to cover the end-to-end critical path of an inference request, capturing key state transitions from ingress to the model execution backend.

For each instrumentation point, the table specifies: (1) **Span Name:** The semantic identifier used in trace visualization (e.g., `EngineCore::Step`); (2) **Process:** The operating system process where the span originates, specifically distinguishing between the *Engine Process* (responsible for request scheduling and orchestration) and the *Worker Process* (responsible for distributed model execution); (3) **Thread:** The specific execution context, differentiating the compute-intensive *Main Thread* from asynchronous auxiliary threads such as the *Cache Management Thread*.

This mapping enables StriaTrace to reconstruct the causal chain of execution across process and thread boundaries.

B Detailed Instrumentation Configurations

This appendix details the configuration parameters for the profiling tools used in our evaluation (§6.1): NVIDIA Nsight Systems (NSys), PyTorch Profiler (TProf) and Scalene.

B.1 Config of NVIDIA Nsight System (NSys)

We execute NSys using the Command Line Interface (CLI) to capture system-wide execution traces. To strictly control profiling overhead while maintaining essential hardware observability, the key parameters are configured as follows:

- `-trace=cuda,nvtx,osrt`: Enables tracing of CUDA runtime/driver API calls, NVTX annotations (essential for correlating vLLM logical scopes), and OS runtime libraries.
- `-cuda-graph-trace=node`: Since vLLM heavily utilizes CUDA Graphs to reduce kernel launch overhead, this flag forces NSys to expose the internal structure and execution timing of individual nodes within the CUDA Graphs.
- `-backtrace=none`: Explicitly disables CPU call stack unwinding. As discussed in our baseline setup, collecting full backtraces introduces prohibitive runtime overhead, so we disable it to ensure a fair overhead comparison.
- `-sample=process-tree`: Configures CPU instruction sampling across the entire process tree, but strategically downsamples the frequency to a 16 ms period to further minimize host-side interference.
- `-capture-range=cudaProfilerApi`: Restricts the active trace capture strictly to programmatic triggers via the CUDA Profiler API, ensuring we only record stable inference phases and exclude initialization anomalies.

The complete command used to launch NSys in our evaluation is as follows:

```
nsys profile \
--trace-fork-before-exec=true \
--trace=cuda,nvtx,osrt \
--sample=process-tree \
--cuda-graph-trace=node \
--backtrace=none \
--capture-range=cudaProfilerApi \
--capture-range-end=stop \
--sampling-period=16000000 \
--delay 60 \
--duration 100000 \
vllm serve ...
```

B.2 Config of Torch Profiler (TProf)

We utilize the built-in PyTorch Profiler (TProf) natively provided by the vLLM engine. To capture application-level performance data, the profiler is initialized with the following key parameters:

- `activities=[CPU, CUDA]`: Configures the profiler to capture operations on both the host CPU and the GPU device, enabling the analysis of kernel launch latencies and host-device synchronization.
- `with_stack=True`: Records the full Python call stack for every captured operator. This feature is essential for fine-grained root-cause analysis but serves as the primary source of TProf's memory and latency overhead.

The complete parameters used to launch TProf in our evaluation are as follows:

```
vllm serve ...
...
--profiler-config '{"profiler": "torch", "
    torch_profiler_dir": "\path\to\save"}' \
--profile-torch \
--profile-step-start 50 \
--profile-step-end 9999999 \
--profile-prefill-and-decode
```

B.3 Config of Scalene

By default, vLLM's EngineCore utilizes the `spawn` method to initialize distributed GPUWorker processes. However, as of our evaluation (April 2026), Scalene lacks native, comprehensive support for profiling `spawn`ed subprocesses. Consequently, launching vLLM via a standard Scalene command would exclusively profile the frontend API server process, leaving the critical GPU workers—where the actual inference hot paths reside—completely unobserved.

To overcome this architectural limitation, we implemented a custom hook within vLLM's worker initialization phase. Specifically, when the EngineCore bootstraps a worker, we explicitly force the spawned process to execute a Scalene command, thereby successfully bringing the distributed workers into the profiling scope. The complete command used to launch the target processes with Scalene is as follows:

```
scalene run --off \
--cpu-only \
--cpu-sampling-rate 0.1 \
-o path/to/save \
-m vllm.entrypoints.cli.main serve ...
```

The key parameters controlling the profiling behavior are configured as follows:

- `-off`: Starts the process with profiling initially disabled. This allows us to programmatically trigger the profiler only during the steady-state inference phase.
- `-cpu-only`: Explicitly restricts Scalene to CPU-side profiling. This minimizes instrumentation interference on device-side operations, allowing us to evaluate the host-side overhead cleanly.
- `-cpu-sampling-rate`: Dictates the temporal resolution of CPU sampling. In this example, it is set to `0.1` seconds. As detailed in §6.1, we vary this parameter (*e.g.,* `0.01` for 10 ms and `0.1` for 100 ms) to evaluate the trade-off between profiling fidelity and throughput degradation.

C Details of Use Cases

In this section, we provide supplementary details for the production anomalies introduced in §7. For each highlighted case, we expand upon the initial triage by presenting the fine-grained runtime context captured by StriaTrace, the methodology and results of our offline reproduction, and an in-depth root-cause analysis.

C.1 Case 1: Latency spikes in token sampling

Figure 11 shows the culprit code and Table 6 presents the full Python call stack captured by *py-spy* on the straggler rank. The stack reveals that the vLLM main inference thread was persistently stalled at the `make_ndarray_with_pad` function. From the call stack, we can trace the execution flow: the main worker loop dispatches the sampler, which applies penalty constraints to logits. When such constraints are specified, the `make_ndarray_with_pad` function is triggered to pad historical token sequences.

Table 6: Python call stack on the straggler rank.

Function (File:Line)	
make_ndarray_with_pad (torch_utils.py:349)	← Top
make_tensor_with_pad (torch_utils.py:370)	
_convert_to_tensors (penalties.py:54)	
apply_all_penalties (penalties.py:24)	
apply_penalties (sampler.py:364)	
apply_logits_processors (sampler.py:351)	
forward (sampler.py:92)	
_call_impl (module.py:1762)	
_wrapped_call_impl (module.py:1751)	
_sample (gpu_model_runner.py:3412)	
sample_tokens (gpu_model_runner.py:4434)	
sample_tokens (gpu_worker.py:653)	
worker_busy_loop (multiproc_executor.py:858)	
worker_main (multiproc_executor.py:781)	
run (process.py:108)	
_bootstrap (process.py:314)	
_main (spawn.py:135)	
spawn_main (spawn.py:122)	← Bottom

The implementation of `make_tensor_with_pad` (the immediate caller) is shown below; line 349 in `torch_utils.py` corresponds to line 21:

```
def make_ndarray_with_pad(
    x: list[list[T]],
    pad: T,
    dtype: npt.DTypeLike,
    *,
    max_len: int | None = None,
) -> npt.NDArray:
    """
    Make a padded array from 2D inputs.

    The padding is applied to the end of each inner list
         until it reaches
    `max_len`.
    """
    if max_len is None:
        # Unlike for most functions, map is faster than a
             genexpr over `len`
        max_len = max(map(len, x), default=0)

    padded_x = np.full((len(x), max_len), pad, dtype=
         dtype)
    for ind, blocktb in enumerate(x):
        assert len(blocktb) <= max_len
        padded_x[ind, : len(blocktb)] = blocktb

    return padded_x
```

C.2 Case 2: Silent performance degradation due to FlashAttention Miscompilation

C.2.1 Full mangled name of the culprit FlashAttention kernel

Table 7 presents the full mangled names of the FlashAttention kernel captured by StriaTrace across two vLLM builds. The names are nearly identical except for a critical template argument difference highlighted in red. The template arguments encode the tile size configuration: the old build uses $(64, 128, 128)$ which is optimal for NVIDIA H20 GPUs,

whereas the new build uses $(128, 128, 128)$ as a sub-optimal fallback. This discrepancy directly caused the observed latency regression.

Table 7: Mangled names of FlashAttention kernel across two vLLM builds.

Old Build (Optimal):

```
_ZN7cutlass13device_kernelIN5flash20enable_sm90_or
_laterINS1_16FlashAttnFwdSm90INS1_25CollectiveMai
nloopFwdSm90ILi2EN4cute5tupleIJNS5_1CILi1EEES8_S8
_EEENS6_IJNS7_ILi64EEENS7_ILi128EEESB_EEELi128ENS_
10bfloat16_tEfNS_4arch4Sm90ELb1ELb0ELb0ELb1ELb1EL
b0ELb0ELb1ELb1ELb1
```

New Build (Sub-optimal):

```
_ZN7cutlass13device_kernelIN5flash20enable_sm90_or
_laterINS1_16FlashAttnFwdSm90INS1_25CollectiveMai
nloopFwdSm90ILi2EN4cute5tupleIJNS5_1CILi1EEES8_S8
_EEENS6_IJNS7_ILi128EEESA_SA_EEELi128ENS_10bfloat1
6_tEfNS_4arch4Sm90ELb1ELb0ELb0ELb1ELb1ELb0ELb0ELb
1ELb1ELb1
```

C.2.2 Detailed settings of offline reproduction

To quantify the impact of the tile size discrepancy under production-like setting, we conducted an offline benchmark on the FlashAttention-3 (FA3) kernel. We utilized the `sm_margin` parameter to explicitly govern the Streaming Multiprocessor (SM) allocation. For instance, setting `sm_margin` to 70 restricts the kernel to utilize a maximum of $100 - 70 = 30\%$ of the available SMs, simulating a high-concurrency scenario where SMs are heavily occupied by concurrent tasks. Our benchmark targets the `flash_attn_with_kvcache` function, which internally invokes the kernel introduced in Appendix C.2.2.

As shown in Table 8, under full SM availability, Version 2 exhibited a slight latency regression compared to Version 1 (83.5 µs vs. 73.7 µs). However, under severe SM constraints (`sm_margin`$= 70$), the performance gap drastically widened. Version 1 degraded to 147.7 µs (a 2.0× slowdown), whereas Version 2 suffered a latency spike to 216.8 µs (a 2.6× slowdown).

Table 8: FA-3 kernel latency under different SM availability.

SM Allocation	Version 1 $(64, 128, 128)$	Version 2 $(128, 128, 128)$
Full SM	73.7 µs	83.5 µs
30% SM	147.7 µs	216.8 µs

Root-cause analysis. The root cause lies in the intricate interplay between kernel compilation configurations and GPU hardware limits. The larger tile size $(128, 128, 128)$ in Version 2 dictates higher per-block resource consumption (*e.g.,* shared memory and registers). While the GPU can absorb this pressure when all SMs are idle, it leads to a severe drop in active block occupancy when SM resources are constrained. Consequently, fewer thread blocks can be scheduled concurrently. This case powerfully underscores the necessity of StriaTrace's fine-grained kernel tracing, as such compilation-induced inefficiencies are entirely invisible to standard application-level monitoring.

C.3 Case 3: Diagnosing "ghost tokens" in speculative decoding

As mentioned in §7.2, the engineering team reported a noticeable drop in the decode throughput. StriaTrace's continuous tracing revealed that the median latency of a single decode step had inflated from 7 ms to 9 ms. To understand this 2 ms overhead, SREs delved into the fine-grained CPU-side metadata captured by StriaTrace, *i.e.,* the per-batch scheduling contexts. The trace data exposed an algorithmic anomaly: *during each decode step, the draft token acceptance rate had dropped to nearly zero*. In a standard speculative decoding pipeline, the GPUWorker's draft model proposes a sequence of draft tokens after a forward pass. These tokens are returned to the EngineCore for scheduling and subsequently passed to the target model for verification. However, StriaTrace revealed a critical quantitative mismatch: *the number of draft tokens initially proposed by the worker was strictly less than the number of tokens eventually submitted for verification.* This explicitly confirmed the silent injection of **"ghost tokens"** into the pipeline.

Finally, the SREs confirmed that this was a secondary bug. Previously, the engineering team had encountered an issue where generating JSON-structured outputs under the EAGLE-3 [19] algorithm occasionally produced `NaN` logits. To mitigate this, developers forcefully padded the sequence of draft tokens to a fixed length before returning it to the EngineCore. Unfortunately, this forceful padding logic lacked algorithmic isolation. It leaked into instances utilizing the N-gram [48] speculative algorithm. Consequently, the target model was forced to expend GPU compute cycles verifying these artificially injected, mathematically meaningless padding tokens. This wasted verification compute directly accounted for the inflated step latency and the drop in the token acceptance rate.

ARTIFACT EVALUATED
USENIX
AVAILABLE

Diagnosing Performance Issues in Application-Defined Resources

Yigong Hu
Boston University

You-Liang Huang
Boston University

Haodong Zheng
University of Washington & EPFL

Yicheng Liu
University of Washington & UCLA

Dedong Xie
University of Washington

Baris Kasikci
University of Washington

Abstract

Many performance issues in large software systems are caused by application-defined resources, such as buffer pools, query caches, and temporary data structures. These resources are managed within the application logic and can strongly affect program execution. However, their resource-specific semantics are often not visible through system-level metrics. As a result, inefficient designs in the management of these resources can cause performance degradation that is difficult to observe and diagnose with existing profilers.

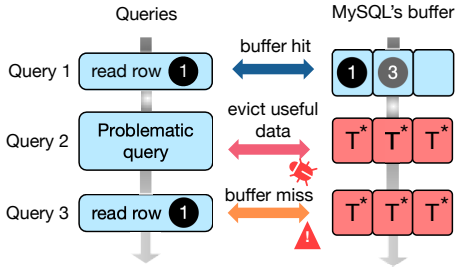
This paper presents *gigiprofiler*, a profiler that diagnoses performance problems caused by application-defined resources. *gigiprofiler* uses a hybrid method that combines LLM-based semantic inference with static analysis: LLM identifies candidate application-defined resources and their usage events from semantic cues, while static analysis validates these candidates against the code. *gigiprofiler* then tracks how each request interacts with inferred resources and records usage events at runtime. *gigiprofiler* detects bottlenecks from aggregate usage events and attributes each resource bottleneck to responsible requests and links the runtime evidence back to code paths to explain how the bottleneck occurs.

We evaluated *gigiprofiler* on 15 real-world performance issues in five widely deployed applications. *gigiprofiler* detects and diagnoses all 15 issues and further uncovers two previously unknown performance issues in MariaDB, both later confirmed by the developers.

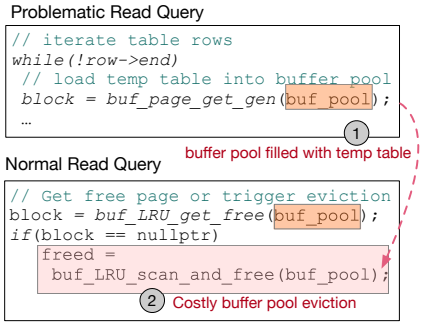
1 Introduction

Diagnosing performance issues in large-scale software is notoriously time-consuming and difficult. A recent survey [4] reports that developers spend nearly 57% of their working hours addressing performance issues. The survey finds that limited visibility into application behavior is a key reason why performance issues are difficult to understand and fix.

Performance issues caused by application-defined resources are especially difficult to diagnose because they introduce a severe visibility gap. Application resources, such as query cache, task queue, or write-ahead logs, are logical abstractions that are defined and used by the application internally. These resources are embedded in application logic and can therefore directly affect program execution and performance. However, their resource semantics, such as how they are allocated, accessed, or scheduled, are often not visible through system-level metrics, such as CPU utilization or memory consumption. As a result, bugs in the application code that manages these resources can cause performance degradation that is difficult for developers to observe and diagnose.



(a) A problematic read query fills the MySQL buffer pool with temporary tables. T^*: intrinsic temporary Innodb table.



(b) Simplified code snippet related to the buffer pool. The elements with orange-colored background represent the buffer pool variable and the pink ones represent slow eviction operations.

Figure 1: (a) Example of performance issue in MySQL buffer pool. (b) The code of the root cause.

Figure 1 shows a real performance bug in MySQL [33, 38]. MySQL uses InnoDB as its default storage engine. InnoDB has a buffer pool to cache data pages accessed by a read query, even if the page contains temporary intermediate results that no other query will reuse. As a result, a query that generates many intermediate results can monopolize the buffer pool and evict useful cached data, forcing other queries to reload pages from disk and causing end-to-end performance degradation. For example, some scan queries produce large intrinsic temporary InnoDB tables to hold intermediate results. When such a query runs, the pages of this temporary table can fill the buffer pool and displace useful cached data.

This performance bug is very difficult to diagnose in practice. Figure 1b shows the bug's code snippet: the buggy scan query repeatedly calls `buf_page_get_gen` function to obtain a large number of buffer pages and writes temporary table data into them. Subsequent read queries thus cannot get a free page from the buffer pool and must invoke a costly `buf_LRU_scan_and_free` function to evict a buffer page and reload useful data. From the code alone, this appears to be a normal buffer write: there are no lock conflicts, long loops, and no explicit error signals that indicate a slowdown. The operating system also does not show memory pressure since the buffer pool is an application-defined resource. As a result, developers have little observable evidence to follow, making the root cause extremely difficult to diagnose.

This lack of observable signals exposes a fundamental limitation in existing profilers. System-level profilers [8, 11, 13, 59] track CPU, memory and lock activity, while code-level profilers search for long loops [18, 43], blocking operations [1, 19, 49, 54], or causal relationships [5, 25, 41]. But performance issues in application resources propagate indirectly through application logic that manages the resource, logic that neither class of profiler observes. In effect, the usage of resource in one request can push later requests onto a much slower execution path, without triggering system events or producing identifiable code patterns such as long loops or hotspots. Consequently, existing profilers surface only weak indirect clues, forcing developers to rely heavily on domain expertise and manual investigation to pinpoint the true root cause.

The key gap is that application resources have semantics that traditional program analyzes alone (static or dynamic) cannot reliably infer. These tools cannot interpret how an application defines, allocates, or competes for its own application resources, making performance issues in application resources difficult to detect or explain.

To make such bugs observable, a tool must expose the interactions between requests and application-defined resources. This is difficult because application resource implementations vary widely across systems and their behavior depends on application-specific semantics not reflected directly in the control or data-flow structure. Our study of 45 real performance issues in application resources shows that, despite various application resource pathologies, four recurring application resource events can always distinguish abnormal usage from normal (Section 2.2 and Section 3.2). Concretely, each event corresponds to a distinct observable interaction with a resource: a task that waits, acquires, accesses, or releases it. We argue that diagnosing application resource performance issues must treat application resource events as first-class signals.

Automatically extracting these usage events requires interpreting both high-level semantics and concrete code behavior. Semantic cues appear in program metadata (names, comments, documentation) that static analysis cannot exploit, but LLMs can. Conversely, confirming that a candidate event truly corresponds to a resource access requires precise control- and data-flow reasoning that LLMs do not provide. We therefore adopt a hybrid design: LLMs identify candidate resources and usage events from metadata, and static analysis validates them against the actual code. This combination increases the coverage and accuracy needed to reliably locate usage events across large codebases.

Building on this hybrid approach, we propose *gigiprofiler*, a diagnosis tool that exposes the internal usage of application-defined resources. *gigiprofiler* identifies application-defined resources, tracks how each request interacts with them, and records usage event sequences during execution. A straightforward way to use usage event sequences is to search for fixed patterns that indicate bottlenecks. However, the hybrid approach may identify incorrect usage events or miss some true events. To make diagnosis robust to incorrect , *gigiprofiler* instead statistically analyzes request-resource interactions, such as how often requests wait for each resource and how often each resource is acquired, to detect bottlenecks. *gigiprofiler* then attributes each bottleneck to responsible requests and performs code-level root-cause analysis to explain which part of the code causes the resource bottleneck.

We applied *gigiprofiler* to five popular large-scale applications. MySQL, MariaDB, PostgreSQL, Apache Web Server, and llama-cpp. To evaluate its effectiveness, we reproduced 15 application resource performance bugs reported in these systems. *gigiprofiler* detected and correctly identified the root cause of all of them. In addition, during the evaluation, *gigiprofiler* discovered two new application resource performance bugs in MariaDB, both of which were confirmed by the developers. *gigiprofiler* is open sourced at `https://github.com/BlizzardLab/OmniProfiler`.

In summary, this paper makes the following contributions.

- We propose a hybrid approach that combines large language models and static analysis to accurately identify application resources and their usage events.
- We design and implement *gigiprofiler*, a performance diagnosis tool that dynamically detects internal-resource performance issues by tracing and analyzing application resource usage.

- We evaluated *gigiprofiler* on five popular applications, demonstrating its effectiveness in detecting 15 real-world internal-resource performance issues and uncovering 2 new performance bugs.

2 Understanding Application Resource Issues

Performance issues caused by application resources are prevalent in large systems. Li et al. [20] found that 22% of 115 performance issues in 20 open-source applications were caused by contention within application resources, and pBox [15] showed that requests frequently interfere with each other through application resources. In this section, we start with a motivating example to show why these issues are difficult to diagnose, and then conduct an empirical study of 45 real-world application resource performance issues to gain insight into the problem's root causes, impact, and guide our solution design.

2.1 Motivating Example

MySQL-75540 [32] is a long-standing performance issue related to the InnoDB UNDO log. Developers have reported that the purge operation can become extremely slow when the UNDO log increases, leading to significant performance degradation [30]. This issue has been discussed in multiple bug reports [29, 31] and community posts [35–37].

The issue comes from a large UNDO log that dominates the InnoDB buffer pool. MySQL uses the UNDO log to maintain older versions of modified rows for transaction isolation. A long-running transaction can therefore cause undo records to accumulate over time. After the transaction commits, these undo records become obsolete and the purge thread must load and clean them through the buffer pool. When the UNDO log is large, this cleanup process can consume many buffer-pool pages, remove useful cached pages, and cause a significant drop in throughput for other transactions. As shown in Figure 2, the MySQL's end-to-end throughput decreases from about 700 QPS to nearly zero when the purge thread starts.

Debugging this issue is challenging because the purge thread does not block other transactions directly on the UNDO log, and the buffer-pool contention is hidden from system-level metrics. The InnoDB buffer pool is a pre-allocated memory region managed by MySQL. Therefore, even when many buffer pool pages are occupied by UNDO-log cleanup, the operating system only observes normal memory usage rather than contention inside the buffer pool. As a result, during the debugging process, multiple incorrect hypotheses were proposed. For example, some developers initially suspected heavy disk I/O under concurrent updates because the system appeared overwhelmed by disk activity [28]. The true root cause was identified only after years of investigation, through careful examination of the source-code and manual tracing of the UNDO-log usage.

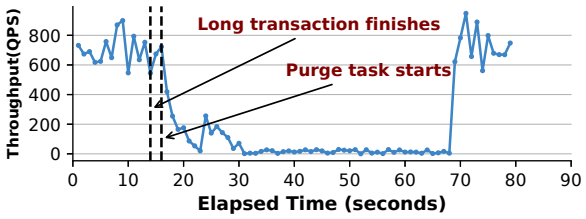


Figure 2: A throughput drop when the purge thread cleans up the huge UNDO log.

```
1   /******** Purge coordinator thread ********/
2   uint trx_purge_attach_undo_recs (node, index, entry) {
3     ...
4     while (n_pages_handled < batch_size) {
5       rec = trx_purge_fetch_next_rec(trx_id,...);
6       ...
7       purge_group.add(rec);
8     }
9     purge_group.assign(thread);
10    ...
11  }
12
13  // Gets the record in an undo log.
14  trx_undo_rec_t* trx_undo_get_rec (page_id, page_size,...) {
15    ...
16    undo_page = trx_undo_page_get(page_id, page_size, ...);
17    ...                    (1) Get a page from buffer pool
18    rec = trx_undo_page_get_first_rec(undo_page, page_no, offset);
19                           (2) Read undo log from page
20    if (rec) {
21      return (rec);
22    }
23  }
```

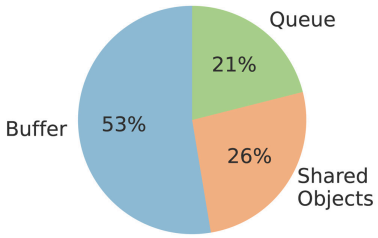
Figure 3: Code snippets showing how the UNDO log fills up the buffer pool during the purge operation.

Figure 3 shows the code relevant to the root cause. When the purge thread processes undo records, it fetches batches of undo pages into the buffer pool. With a large UNDO log, this batching is repeated many times. As a result, the purge thread evicts useful cached pages needed by other queries (step 1 in Figure 3) and keeps many undo pages in the buffer pool (step 2 in Figure 3). This frequent buffer-pool turnover slows down the system.

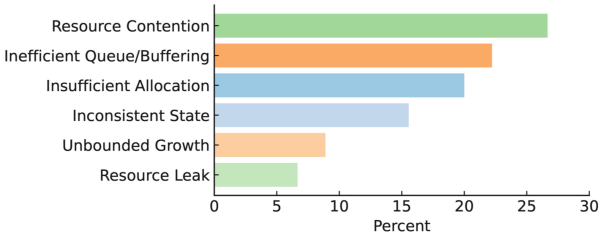
Existing performance debugging techniques have difficulty in locating the root causes: an over-sized UNDO log causes the cleanup process to consume buffer-pool pages. Profilers [19, 57] that track OS-level activity can observe indirect symptoms, such as increased disk I/O or longer execution time, but they cannot directly link the slowdown to the UNDO log and the lack of timely UNDO log truncation. Static analysis tools [18, 43, 44] lack the application-specific semantic context needed to understand how UNDO log, purge thread, and buffer pool interact to degrade performance.

2.2 Empirical Study

Application-defined resources are logical abstractions created and managed by the application code. They may use underlying system resources, such as memory, threads, or synchronization primitives, but their semantics are defined by the application. Performance issues caused by application-defined resources are difficult to diagnose because we lack



(a) Application resource Type.



(b) Root cause Type.

Figure 4: Application resource type and root cause distribution.

a systematic understanding of what kinds of application resource developers define, why they are mismanaged, and what recurring pathologies they exhibit. To fill this gap, we conducted an empirical study of 45 real-world performance issues caused by application-defined resources.

Methodology: We collect performance issues from three widely used large software systems: MySQL, PostgreSQL, and Elasticsearch. These systems contain millions of lines of code and cover two major classes of data-intensive systems: databases and distributed search engines. For each software, we manually examined bug reports and forum posts whose titles or descriptions contain terms such as "slow" or "hung", and select cases whose root causes are related to application-defined resources.

We exclude cases where the synchronization is the direct root cause, such as lock contention or deadlock. However, we keep the cases where contention comes from how an application-defined resource is managed, even if the resource is protected by locks. We keep these cases because diagnosing them requires understanding why the resource becomes monopolized, whether the monopolization is expected, and how the resource-management logic causes other requests to wait. Therefore, in those cases, identifying the contended lock alone does not explain the root cause.

Diverse Resource Types and Root Causes: Our study shows that performance issues are caused by a variety of application-defined resources. Across the 45 issues, we identified 38 distinct application-defined resources, including query caches, UNDO logs, write buffers, and write-ahead logs. A key finding is that the same application-defined resource can be implemented very differently across systems. For example, both MySQL and PostgreSQL implement a UNDO log abstraction. However, MySQL implements UNDO log as linked

Root Cause Category	Total Cases	Matched Cases
Resource contention	12	12
Inefficient policies	10	8
Insufficient allocation	9	7
Inconsistent state	7	5
Unbounded growth	4	4
Resource leak	3	3
Total	**45**	**39 (87%)**

Table 1: Mapping the 45 studied cases to the pathology patterns in our analytical model.

log records, whereas PostgreSQL stores old row versions directly in tables.

We categorize these resources into three broad types. Figure 4a shows their distribution. *Memory-related resources* are resources that provide application-level storage abstraction, such as query caches, read buffers, and shared memory. *Shared state objects* are shared application objects whose state affects the performance of many requests, such as tables, indexes, and logs. *Internal queues* are resources that order, schedule, or coordinate work inside the application, such as task queues and thread pools.

Figure 4b summarizes the root causes of the cases. Performance problems come from six categories: contention for application resources (26.7%), inefficient queuing or buffer-management policies (22.2%), insufficient resource allocation (20%), inconsistent software state (15.6%), unbounded resource growth (8.9%), and resource leaks (6.7%). Despite the diverse root-cause types, these performance issues share a common characteristic: they happen when application-specific resource-management semantics are violated under certain workloads or execution paths. When an application implements a resource, the resource implicitly carries semantics about how it should be managed. For example, the MySQL's buffer-pool implies that no single query should monopolize buffer-pool pages. Violating these semantics can cause the resource to be mismanaged and degrade performance.

Classification of Application-Resource Pathologies. We further analyze the studied cases to answer a central question: given that performance issues in application-defined resources come from application semantics, do these semantic violations share recurring failure patterns? Examining the source code of the 45 cases, we find that application-defined resources can be modeled by a simple but expressive *acquire-use-release* lifecycle, similar to the lifecycle of OS resources. A request obtains a resource, uses it to perform work, and eventually releases it. Deviations from this lifecycle give rise to common pathologies:

- **Resource Contention**: a request monopolizes an application resource, forcing other requests to acquire it through a slow or expensive path, or wait for a long time.
- **Inefficient Policies**: suboptimal eviction, batching, or scheduling policies cause a request to hold a resource that it rarely uses.

- **Insufficient Allocation**: a request receives too few resource units and repeatedly reuses a small share of the resource despite sufficient overall capacity.
- **Unbounded Growth**: an application-defined resource keeps growing unbound.
- **Inconsistent State**: incorrect creation or use of a resource leaves the system in a state that later prevents progress or causes performance degradation.
- **Resource Leaks**: an application resource leaks.

Table 1 shows that 87% of cases align with these pathology patterns. The remaining cases have policies that are too complex to fit into a category.

The general characteristic of these pathologies is that they violate application specific resource-management semantics. Moreover, their performance impact is context-dependent: a given resource usage pattern degrades performance only under particular workloads or execution paths.

2.3 Limitations of Existing Solutions

Traditional performance profilers and debugging tools diagnose bottlenecks by tracing system-level events and inferring relationships such as causal paths [5, 11, 19, 41, 59]. However, these approaches are insufficient to diagnose application-resource performance issues. Application-defined resources do not provide a unified interface that existing analyzes can directly use to construct wait-for dependencies, causal paths, or other diagnostic relations. For example, relational debugging [41] models fine-grained runtime events as relations and uses statistical A/B testing to identify relations that are likely to explain performance differences. However, it requires the relevant events to be visible to the profiler. Without knowing the internal application-defined resources, relational debugging is unable to find the root causes

A straightforward way is to report every function that modify the application-resource at runtime, but this would generate unaccepted overhead. Moreover, application-defined resources have diverse semantics and characteristics, and the OS or runtime lacks the application-specific knowledge needed to interpret their usage patterns.

Another potential solution is to combine LLMs with static analysis to detect potential application-resource pathologies. The pathology patterns in Section 2.2 can guide this process: an LLM can map a pathology to concrete code constructs, and static analysis can validate whether the corresponding resource-usage pattern exists in the code. However, these patterns are context-dependent. Static evidence alone cannot tell whether a pattern actually hurts performance in a running workload. Thus, LLM-based semantic inference and static validation must be combined with dynamic profiling to diagnose application-resource performance issues.

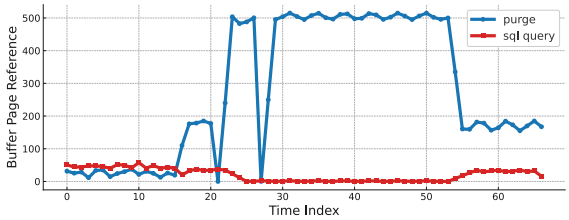


Figure 5: MySQL's buffer-pool page usage over time, split between the purge thread and all other queries.

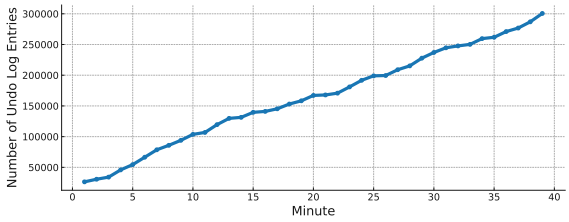


Figure 6: Growth of undo-log entries over the past 40 minutes.

The pathology study guides *gigiprofiler*'s design. It shows that diagnosing application-resource pathologies does not require tracking every internal state change of an application-defined resource. Instead, observed pathologies can be captured using a compact application resource event interface: `WAIT`, `ACQUIRE`, `USE`, and `RELEASE`.

2.4 *gigiprofiler* on the Motivating Example

To show how these application resource events help diagnose real performance issues, we apply *gigiprofiler* to the motivating example of Section 2.1.

We first reproduce the issue by creating a table with 1,000 records and running a long transaction that is active for several hours. In parallel, we issue repeated small write transactions that update the same row, causing undo records to accumulate while the long transaction is active. When the long transaction finally commits, the purge thread begins cleaning the accumulated UNDO log, and triggers the performance degradation shown in Figure 2.

gigiprofiler identifies the InnoDB buffer pool as an application-defined resource. It then identifies `trx_undo_page_get` as an `ACQUIRE` event for this resource and `trx_undo_page_get_first_rec` as a `USE` event, corresponding to lines 17 and 19 in Figure 3. *gigiprofiler* instruments these events, along with the corresponding `WAIT` and `RELEASE` events, and records that request triggers each event during execution. *gigiprofiler* applies the same event-identification process to the UNDO log. After instrumenting the usage events of both the buffer pool and the UNDO log, *gigiprofiler* records the request triggers each event during execution.

Figure 5 compares the `USE` events of the buffer pool triggered by the purge thread and by normal queries, measured as the average number of `USE` events per second. The purge thread dominates the usage of the buffer pool, indicating con-

tention for the buffer pool pages. *gigiprofiler* further identifies that most of these pages are used to store undo records, linking the buffer pool contention to the UNDO log resource. Figure 6 shows the growth of the UNDO log over time. *gigiprofiler* tracks `ACQUIRE` and `RELEASE` events for the UNDO log and estimates the size of the UNDO log as the difference between the number of `ACQUIRE` and `RELEASE`. This unbounded growth explains why the purge thread later consumes many buffer-pool pages and identifies the UNDO log as the root cause of the performance degradation.

2.5 Lesson from Our Initial Attempt

Although application resource usage events provide a useful abstraction for diagnosis, identifying them from code is challenging. Our initial attempts to use LLMs revealed significant limitations. We developed a prompt-based LLM pipeline that provides the model with the definition of application-defined resources, the roles of usage events, and concrete examples. Although this pipeline could identify many application-defined resources and usage events, its precision was low and is difficult to increase simply by improving the prompt.

We identified four key challenges. First, scaling LLM-only analysis to large systems is impractical: these systems contain millions of lines of code, exceed the model's context window, and require many expensive model calls. In our experience, scanning MySQL with GPT-5.2 took 3 days and cost about $1,000. Second, identifying application usage events often requires control and data-flow information, which is rarely available in comments or documentation. Third, comments and documentation can be ambiguous, misleading, or even absent. For example, `is_active` is only a checking function, but its comment includes the phrase "get the status of bin_log," causing the LLM to mistakenly mark it as a `USE` event. Fourth, even when comments are clear, LLMs may still make incorrect inferences or hallucinate resources that do not exist.

These limitations show that resource identification should not be treated as a monolithic LLM task. LLMs provide broad semantic coverage, but their output is inaccuracy; static analysis provides precise code reasoning, but lacks the semantic context. *gigiprofiler* avoids this trade-off by integrating both. The LLM extracts semantic cues, and static analysis validates these candidates using control-flow and data-flow information. This hybrid, agentic design grounds LLM cues in actual program behavior, reducing false positives and enabling accurate identification of application resources and their events.

3 Design of *gigiprofiler*

In this section, we describe the design of *gigiprofiler*. Our empirical study showed that performance issues in application resources can be captured by specific patterns of pathological resource usage and that neither static analysis nor LLMs alone can reliably identify application resource usage. These insights guide the design of *gigiprofiler* 's agentic static analyzer. We begin by introducing the four resource-usage events that *gigiprofiler* instruments, then describe how the agentic static analyzer identifies and validates application resources and their usage events, and finally explain how *gigiprofiler* profiles and analyzes the behavior of application resources at runtime.

Event	Parameters	Description
`WAIT`	(resource_id, `wait`)	Task spends `wait` time on the slow acquire path.
`ACQUIRE`	(resource_id, `units`)	Task obtains `units` number of the resource.
`USE`	(resource_id, use_type, target_id)	Task uses resource by read/write, optionally involving another resource `target_id`.
`RELEASE`	(resource_id, units)	Task releases `units` number of previously acquired resource

Table 2: Event types and parameters used to instrument application-defined resources.

3.1 *gigiprofiler* Workflow

We developed *gigiprofiler* to identify application resources, trace their usage, and diagnose the root causes of performance issues in application resources. Figure 7 shows its workflow. *gigiprofiler* has two phases: an offline static-analysis phase and a dynamic profiling phase. In the static-analysis phase, an agentic static analyzer identifies application resources and their usage events. Software comments, naming conventions, and documentation are preprocessed into structured metadata and provided to an LLM agent, which proposes candidate resources and events. Static analysis then validates these candidates using control-flow and data-flow information, removing false positives and producing a precise list of resources and operator functions.

In the dynamic profiling phase, *gigiprofiler* instruments the application to trace application resource events during a buggy execution. *gigiprofiler* records usage events and value samples for each resource, and the collected traces are analyzed offline to detect contended resources and identify pathological usage patterns. This integrated profiling and analysis provides developers with an explainable code-level diagnosis of performance issues in application resources.

The deployment workflow. *gigiprofiler* is designed as an on-demand diagnostic tool. The static-analysis phase runs once for each application version and produces a reusable list of application resources and usage events. The dynamic-profiling phase is activated only when a performance issue is observed, which keeps runtime overhead negligible during normal operation.

3.2 Application Usage Events

gigiprofiler models application resources using four usage events: `WAIT`, `ACQUIRE`, `USE`, and `RELEASE`, which is derived from

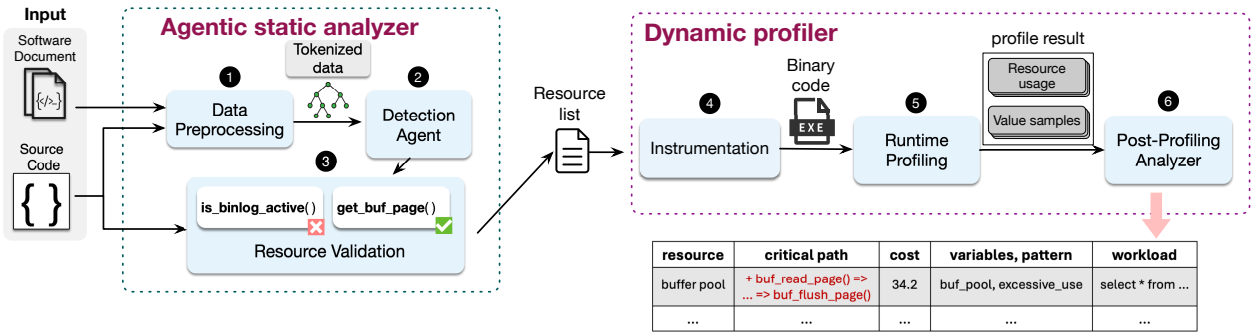


Figure 7: An overview of *gigiprofiler*.

```
/* Get a buffer page */
buf_block_t* buf_LRU_get_free_block(
    buf_pool_t* buf_pool) {
  …
loop:
  block = buf_LRU_get_free_only(buf_pool);
  get(block, 1);
  if (block) {
    return block;
  } else {
    block = buf_LRU_scan_and_free(buf_pool);
  }
  …
  goto loop
}
```

sql/buf0buf.cc

```
/* Free a buffer page */
bool buf_LRU_free(…){
  bpage = buf_pool->LRU
  while (!freed) {
    freed = buf_LRU_free_page(bgpage);
    free(bpage, 1)
    bpage = get_prev(bpage);
  }
  return (freed);
}
```

sql/buf0lru.cc

```
/* Evict a buffer page to get a new page*/
buf_block_t* buf_flush_page_from_LRU(
    buf_pool_t* buf_pool) {
  for (bpage = buf_pool.start(); bpage!=null;
        bpage = buf_pool.next()) {
    …
    if(freed = buf_flush_page(bpage)) {
      break;
    }
    …
  }
  wait(bpage, time);
  return freed;
}
```

sql/buf0flu.cc

```
/* Write to a buffer page */
buf_block_t* trx_undo_get_rec() {
  …
  undo_page = trx_undo_page_get(page_id, …);
  …
  rec = trx_undo_page_get(undo_page);
  use(undo_age,"write",rec)
  …
}
```

sql/trx00undo.cc

Figure 8: Example of tracing buffer pool usage in MySQL.

our empirical study. The API is listed in Table 2. These events provide a unified abstraction for how tasks interact with diverse application resources. `WAIT` records the time a task spends on a slow acquisition path; `ACQUIRE` denotes when a task obtains a unit of the resource; `USE` denotes operations performed using the resource, such as reading or writing; and `RELEASE` denotes when the resource is returned. Although simple, this model captures the pathological patterns observed in our study: contention appears as prolonged `WAIT` events, inefficient policies manifest as resources that are frequently `ACQUIRE` and held for long periods without corresponding `USE` events, insufficient allocation emerges as repeated `ACQUIRE` attempts, and unbounded growth or leaks surface as unmatched `ACQUIRE`/`RELEASE` pairs.

Instrumentation is fully automated: once the agentic static analyzer identifies the code locations corresponding to `WAIT`, `ACQUIRE`, `USE`, and `RELEASE` events, *gigiprofiler* injects lightweight probes at those locations using an LLVM pass. No developer effort or manual annotation is required. Figure 8 shows the instrumentation points that *gigiprofiler* insert into the MySQL buffer pool. The `ACQUIRE` event is implemented as the `get` API, which *gigiprofiler* inserts in `buf_LRU_get_free_block()` after a buffer page has been obtained from `buf_LRU_get_free_only()`. If no free page is available, MySQL chooses an old page via `buf_flush_page_from_LRU()`, and *gigiprofiler* records a `WAIT` event to capture the time spent on this slow-acquiring path. When a page is returned to the buffer pool in `buf_LRU_free()`, *gigiprofiler* emits a `RELEASE` event via the `free` API. Finally, when `trx_undo_get_rec()` writes the undo log to a buffer page, *gigiprofiler* logs a `USE` event to record that the page is being used to store undo records. Together, these events allow *gigiprofiler* to reconstruct how tasks acquire, wait for, use, and release buffer pool pages.

3.3 Finding Application Usage Events

3.3.1 Organizing Input Data

Identifying application-defined resources and usage events in a large scale system is challenging. Large-scale software design consists of dozens to hundreds of resources and spans millions of lines of code. Feeding the entire codebase to an LLM as plain text would cause the LLM to lose essential software context, such as control-flow and cross-file relationships that are needed to identify application resources accurately. A straightforward approach is to split the codebase into smaller chunks and run the LLM iteratively, but naïve chunking breaks the structural relationships between files, classes, variables, and functions. The semantic context needed to recognize how resources are defined and used. Thus, *gigiprofiler* requires a more systematic method for organizing and presenting the software context to the LLM.

To preserve context while searching large codebases, *gigiprofiler* organizes metadata into a hierarchy of file → class → variable → function → source code. The LLM agent traverses this hierarchy, pruning branches early when a higher-level node is unlikely to represent a resource. This structure preserves cross-file context and enables efficient, targeted queries despite LLM context-length limits.

Figure 9 shows a snapshot of the hierarchy structure of *gigiprofiler* for MySQL. It is implemented as a restricted directed acyclic graph that enforces a fixed parent–child order from the file to the class/struct variable, to the member variable in the class or structure variable, to the function descrip-

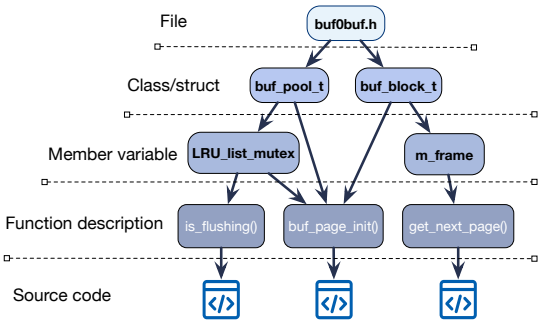


Figure 9: Example of hierarchy structure for MySQL.

tion and finally to the source code. Each node stores relevant metadata, such as names, types, and associated comments or docstrings, to preserve semantic context. A pre-processing module constructs this DAG by parsing the source code and documentation to extract metadata and connect nodes based on data dependencies.

3.3.2 LLM agent

Once the codebase is organized into the hierarchy, the LLM agent must identify candidate application-defined resources and usage events efficiently and accurately. A straightforward approach is to traverse the hierarchy node by node, but this approach is inefficient. First, it may visit many nodes that are clearly unrelated to any resource. Second, the same node can be visited multiple times through different paths. For example, in Figure 9, both `buf_pool_t` and `buf_block_t` may lead the traversal to the function `buf_page_init`.

To avoid redundant and unnecessary queries, *gigiprofiler* uses a top-down traversal with a two-phase reasoning process. Our insight is that application-defined resources are usually represented as class or structure instances, or as variables, whereas usage events are implemented in functions. Therefore, the agent first queries the LLM on high-level class, structure, and variable nodes to identify candidate application-defined resources and then uses these candidates to filter out unrelated function nodes. Because function and source-code nodes contain far more tokens than higher-level metadata nodes, this approach significantly reduces offline analysis time and LLM cost without sacrificing coverage.

The first phase is performed on both the class and variable layers. Figure 10 shows a snapshot of the prompt used by our LLM module to identify the application resources. The prompt contains a short task summary and step-by-step checks that guide the detection agent in deciding whether the variable represents an application resource. We categorize resources into two types: *exclusive resources*, which are accessed by at most one thread (or owner) at a time (e.g., an UNDO log instance tied to a transaction) and *shared resources*, which are concurrently accessed by multiple threads (e.g., buffer pools and work queues).

After identifying candidate application-defined resources, the agent invokes a lightweight data-flow analysis to find

 # Task summary:
You are a class analyzer that analyzes whether a class may represent a key resource in MySQL according to the name of the class and some documents about the functionality of the class.

 # Instruction:
An application-defined resource is a logical software resource whose use can affect performance. Resources are typically accessed across multiple functions and may be acquired, used, or released according to application-specific semantics.
Your goal is to:
1. Infer the high-level purpose of the given class, structure, or function using naming conventions, comments, and metadata.
…
 # Examples:
…

Figure 10: Prompt snapshot for the first phase

functions that use these resources. If a function uses multiple resources, the agent groups it as a function-to-resource pair and analyzes them together, which avoids repeated LLM queries on the same function .

For each selected function, the agent prompts the LLM to determine whether the function contains any of the four usage events. If the function signature and description provide sufficient evidence, the agent makes a candidate decision at the metadata level. Otherwise, the agent inspects the function's source-code node for additional context. To help the LLM interpret the function in context, the agent also provides call-stack information and the source code of related functions when available. If the agent cannot confidently determine whether an operation is a valid usage event, it marks the operation as a candidate and defers validation to the static analyzer.

3.4 Validation of Resource Usage Events

After the LLM agent identifies candidate resources and usage events, the static-analysis module validates whether a candidate event actually waits for, acquires, uses, or releases the corresponding resource. This step is critical because the LLM agent has low precision: without validation, false usage events would be instrumented, and later mislead dynamic diagnosis.

A fundamental challenge is that checking a candidate event requires validating a semantic property that standard program analyzes cannot easily decide on. Existing LLM-assisted static-analysis pipelines often use control-flow or data-flow analysis to validate standard program. However, validating a resource-usage event requires more than such structural consistency. For example, validating that a function is an `ACQUIRE` event for the buffer pool requires checking whether the function actually obtains a buffer pool page and passes it to the caller that will use it to store some internal data. There is no data flow analysis can observe this pattern.

To tackle this challenge, *gigiprofiler* translates each semantic event into a set of necessary code behaviors that the event should exhibit.For example, an `ACQUIRE` event should satisfy three necessary code behaviors: (1) it obtains a resource unit from the resource, (2) it passes the unit to the caller or the

Algorithm 1: Validation for get and wait event

Input : Resource candidates R, Usage events E, Target Function F
Output : Validated resources and events

```
1  foreach candidate (R,E,F) do
2      if E.type == get then
3          CheckAcquire((R,E,F))
4      else if E.type == wait then
5          CheckAcquire((R,E,F))
6  Function checkAcquire(R,E,F):
7      tainted ← GetAllTaintedVar(F,R);
8      if IsResourceWritten(F,tainted) then
9          mark E as invalid acquire;
10         return;
11     if ! IsResourceReturned(F,tainted) then
12         mark E as invalid acquire;
13         return;
14     if ! HasLookupPattern(F,tainted) then
15         mark E as invalid acquire;
16         return;
17     mark E as valid ACQUIRE;
```

Algorithm 2: Validation for release and use event

Input : Resource candidates R, Usage events E, Target Function F
Output : Validated resources and events

```
1  foreach candidate (R,E,F) do
2      if E.type == release then
3          CheckRelease((R,E,F))
4      else if E.type == use then
5          CheckUse((R,E,F))
6  Function CheckRelease(R,E,F):
7      tainted ← GetAllTaintedVar(F,R);
8      sites ← FindReleaseSites(F,tainted);
9      if sites is empty then
10         mark E as invalid release;
11         return;
12     if HasUseAfterRelease(F,tainted,sites) then
13         mark E as invalid release;
14         return;
15     if ReturnsReleasedValue(F,tainted) then
16         mark E as invalid release;
17         return;
18     mark E as valid RELEASE;
19 Function CheckUse(R,E,F):
20     tainted ← GetAllTaintedVar(F,R);
21     if TouchesResource(F,tainted) then
22         mark E as candidate USE for dynamic validation;
23     else
24         discard E as spurious;
```

task that will use it, and (3) it does not modify the variable of the application resource. These behaviors are necessary for an ACQUIRE event. *gigiprofiler* then converts these behaviors into static analysis checks, such as checking whether function F's return value comes from resource variable R. Here, F, R, and the event type are provided by the LLM.

For an ACQUIRE(get) event, *gigiprofiler* checks three properties. First, the function should not modify the resource variable during acquisition because the application resource must be read-only during acquisition. Second, the function's return value must be derived from the resource variable; otherwise, it is not actually "getting" a resource unit. Third, the function should search for the resource through iteration, hashing, or indexed access. The Algorithm 1 shows the core logic. The analyzer taints the resource variable, propagates the taint through dependent values (line 7), and verifies that all tainted values are read-only(line 8). It then checks whether the return value is tainted (line 11) and whether the function performs a search over the resource (line 14).

A WAIT event represents the slow path to acquire an application resource. Statistically, its code structure is the same as the ACQUIRE event but includes an additional delay or retry loop. *gigiprofiler* performs structural validation using the same rules as ACQUIRE and relies on the dynamic profiling phase to determine whether the path is truly slow. If the dynamic profiler does not observe a slowdown, the WAIT event is ignored.

For a RELEASE event, the static analyzer verifies that the function exhibits a release-like effect. Concretely, the function should invoke a known release primitive (e.g., a freelist insertion, LRU return, or deallocation routine), after which the resource variable must not be used on any execution path. Thus, *gigiprofiler* enforces a no–use-after-release rule (Line 12 in Algorithm 2). Because static analysis may miss indirect or dynamic uses, the no–use-after-release rule is also enforced during the dynamic profiling phase, which confirms whether the released resource is ever touched again at runtime. The release operation must also transfer the resource back to the application rather than returning the released object to the caller (Line 15 in Algorithm 2).

For USE event, static analysis only marks the resource as potentially used. Dynamic profiling later verifies whether the resource was acquired by the same task before being used and the function is using the resource.

Scope of validation. The static validation module is not intended to be a complete or sound detector of usage events. Its role is to improve the precision of LLM-inferred candidates. The LLM first identify which program variable is the application resource and which event type the function may represent. Given these candidate functions and variable, static analysis checks whether the function exhibits code behaviors that are necessary for the claimed event type.

This LLM-provided candidate is what makes validation practical. Without this candidate, the static analyzer would need to apply all all checks to all variables, all functions, and

all event types, which would be impractical for large-scale software and would produce many false positives.

3.5 Dynamic Profiling

After the software is instrumented with application-resource event hooks, *gigiprofiler* executes the software using a workload that reproduces the buggy execution. During runtime, *gigiprofiler* only records lightweight event information and defers expensive computation to the post-profiling analysis. Each event is logged with a timestamp, resource ID, thread ID, event type, and event-specific parameters, and is stored in a lightweight thread-local ring buffer to avoid costly inter-thread communication on the critical path. *gigiprofiler* periodically and asynchronously flushes the local buffers to the disk. In addition, *gigiprofiler* integrates with *perf* to periodically sample hardware counters, system-resource usage, and application call stacks.

3.5.1 Post-Profiling Validation

Some usage event's rule cannot be validated statically. The post-profiling validation filters these events using the runtime event trace. The validation first checks whether events follow the expected lifecycle order. Within a request, an application-defined resource should follow the expected event order: `ACQUIRE` $\rightarrow$ `USE` $\rightarrow$ `RELEASE`. That is, a resource should be acquired before it is used, and released only after its last use. Across requests, a `WAIT` event should correspond to a conflicting holder of the same resource. The expected order is: $\texttt{ACQUIRE}_{r_1} \rightarrow \texttt{WAIT}_{r_2} \rightarrow \texttt{ACQUIRE}_{r_2}$, where request or task r_1 acquires the resource first, and request or task r_2 waits before later acquiring it. Events that violate these ordering rules are discarded as invalid usage events.

The post-profiling validation then checks the event-specific runtime behavior. For a `WAIT` event, *gigiprofiler* verifies that the event corresponds to a real delay in the slow acquire path. For a `USE` event, *gigiprofiler* check that the instrumented code actually reads or writes the associated resource unit at runtime. For a `RELEASE` event, *gigiprofiler* checks that the released unit is touched by the release function and is not used again by the same request after release. These runtime checks remove false events that pass static validation.

3.5.2 Diagnosing Performance Issues

Even after static and runtime validation, some false application resource events may remain, and some true events may be missed. This makes traditional wait-for analysis unreliable if it treats every event as exact. Directly applying traditional trace-based analysis to such traces can be unreliable. For example, a false `WAIT` event may create a wrong wait relation, while a missing `ACQUIRE` event after a true `WAIT` may make the wait time to be longer than it actually is.

gigiprofiler therefore diagnoses performance issues using aggregate request-resource statistics rather than individual event matches. For each resource, *gigiprofiler* aggregates event signals across requests and functions and computes four per-task metrics. First, *waiting time* measures how long a task spends on the slow acquire path, using timestamps from `WAIT` and `ACQUIRE` event pairs. Second, *utilization* measures how frequently an acquired resource is actually used, computed as the ratio of `USE` to `ACQUIRE` events. Third, *hold time* measures how long a task holds a resource, computed as the time between `ACQUIRE` and `RELEASE`. Fourth, *acquire frequency* measures how often a task repeatedly acquires resource units, computed as the number of `ACQUIRE` events per second.

After computing these metrics, *gigiprofiler* detects pathological usage patterns as follows. If a resource has high aggregate waiting time across requests, *gigiprofiler* classifies it as resource contention. If a task holds a resource for a long time but rarely uses it, indicated by long hold time and low utilization, the pattern is classified as an inefficient policy. If a task repeatedly acquires small resource units despite high utilization, indicated by high acquire frequency, the pattern is classified as insufficient allocation. If `ACQUIRE` events are not matched by `RELEASE` events after a task finishes, *gigiprofiler* identifies a resource leak. If the aggregate gap between `ACQUIRE` and `RELEASE` events continues to increase with time, *gigiprofiler* identifies unbounded growth.

After identifying a bottleneck resource and the tasks responsible for this performance issue, *gigiprofiler* performs root cause analysis to explain why the performance issue occurs at the code level. This step is critical for developers as it pinpoints the specific code paths that trigger the pathological behavior of the resources. *gigiprofiler* performs a static backward analysis based on the usage events of the misbehaving task, tracing the control and data-flow dependencies to locate the operations that initiated resource acquisition, prolonged hold time, or prevented release. Because *gigiprofiler* also records value samples for variables associated with each event, the analysis can highlight values when pathological behavior occurs. To ensure actionable explanations, *gigiprofiler* focuses this backward analysis on code sites that determine how the resource is held, used, and reclaimed, which are precisely the decisions that lead to contention, insufficient allocation, inefficient policies, or leaks. These locations are flagged as the root cause and reported to developers.

3.6 Implementation

We implement *gigiprofiler* as a combination of Python-based analysis components, an LLVM-based static analyzer, and a C++ runtime profiling library. The data preprocessing module is implemented as a Python script that parses source code and documentation. The LLM agent is implemented as a Python orchestrator. It traverses the hierarchy, maintains traversal state, issues LLM queries for selected nodes, and invokes

lightweight static-analysis routines to connect candidate resources to related functions. The underlying LLM is configurable and can be replaced without changing the rest of the *gigiprofiler* pipeline.

The *gigiprofiler* static analyzer is built on top of LLVM [21]. The *gigiprofiler* profiler is implemented as a C++ library that can be linked with the target software. It injects lightweight profiling hooks at validated event locations and records event traces during execution.

4 Evaluation

We evaluate *gigiprofiler* to answer the following questions: (1) How accurately does *gigiprofiler* identify application-defined resources and their usage events? (2) Can *gigiprofiler* detect and localize real-world performance issues in application resources? (3) Can *gigiprofiler* explain these issues by pinpointing their code-level root causes? (4) What is the runtime overhead of dynamic profiling?

We evaluated *gigiprofiler* on five popular, large-scale applications: MySQL, PostgreSQL, MariaDB, Apache Web Server, and llama.cpp. These systems have up to 2 million source lines of code. *gigiprofiler* scans each system and automatically instruments its APIs without requiring manual annotation. All experiments were conducted on servers equipped with 10-core Intel Xeon E5-2640 CPUs, 64 GB of DRAM, and a 480 GB SSD, with Ubuntu 20.04 as the operating system. The main evaluation uses GPT-4o as the underlying LLM model, which was the newest GPT model available when the experiments were conducted. Unless otherwise specified, all results in the evaluation use this GPT-4o.

4.1 Diagnosing Root Cause

To evaluate the effectiveness of *gigiprofiler*, we collected 15 real-world cases of performance issues in application resources. These issues were collected from the official bug-tracking systems of each application, community forums such as StackOverflow and developer posts on social media. As summarized in Table 3, the selected performance issues span a diverse set of resources and performance issue patterns, providing a comprehensive benchmark for evaluation.

For each performance issue, we prepared a reproduction script to trigger the issue. We compare *gigiprofiler* against three baselines: *perf*, a widely used system-level profiler; *PCatch* [19], a performance-bug detection tool for identifying cascading performance issues; and *Perspect* [41], a a relational debugging tool that automatically pinpoints the root causes of performance problems by analyzing statistical changes in the relations between fine-grained runtime events. Both *PCatch* and *Perspect* require program-internal events as input, such as synchronization operations or selected program variables. To provide these inputs without manually specifying application-defined resources, we use SyncFinder [54] to identify application-specific synchronization primitives as input to both systems.

Id.	App	Resource	Root Causes
c1 (link)	MySQL	UNDO log	Unbounded Growth
c2 (link)	MySQL	Table metadata	Resource Contention
c3 (link)	MySQL	MDL	Inefficient Policy
c4 (link)	MySQL	Buffer pool	Resource Contention
c5 (link)	MySQL	Buffer pool	Resource Contention
c6 (link)	MariaDB	Foreign keys	Inefficient Policy
c7 (link)	MariaDB	Queue	Insufficient allocation
c8 (link)	PostgreSQL	MultiXact cache	Resource Contention
c9 (link)	PostgreSQL	pgss	Resource Contention
c10 (link)	PostgreSQL	Shared memory	Inefficient policies
c11 (link)	PostgreSQL	Write Ahead log	Inconsistent state
c12 (link)	PostgreSQL	Reorder buffer	Resource Leak
c13 (link)	Apache	Buffer	Inefficient policies
c14 (link)	Apache	Thread pool	Insufficient policies
c15 (link)	LLAMA	Result queue	Resource Contention

Table 3: Description of 15 *real-world* application resource performance issues.

As shown in Table 4, *gigiprofiler* successfully find the true root cause(s) as the top candidate in all 15 cases. Three cases have multiple root causes, and *gigiprofiler* correctly ranked all of them ahead of unrelated candidates. The analysis process was also fast. On average, *gigiprofiler* takes 95.15 seconds to detect an issue and identify the root cause.

In comparison, *perf* failed to detect the issues in 10 out of the 15 cases. In the 5 cases where performance problems were reported, the true root cause function was ranked very low, making it difficult for developers to locate root causes. PCatch also failed to detect 10 of the 15 cases. it missed all shared resource performance issues as they do not block other requests through synchronization. The Perspect detected 4 of the 15 cases. Perspect detected 4 of the 15 cases. The reason is that Perspect need the normal and buggy executions to determine which relations are abnormal. But in our evaluation, we only provide the reproduced buggy execution because *gigiprofiler* does not require a normal run. As a result, the Perspect can not performance differential analysis correctly.

We provide a case study to show how *gigiprofiler* detects and diagnoses performance issues in application resources:

Case 10: Send Buffer Exhaustion. Apache has a bug where the server continues writing to the send buffer even after the client has stopped receiving data. This behavior gradually fills the buffer with unneeded data, exhausting the buffer pool and blocking new writes. Because the send buffer is managed as an application resource, both *perf* and *PCatch* fail to detect the issue. By contrast, *gigiprofiler* exposes the buffer resource and identifies prolonged waiting times when tasks attempt to acquire free buffer space. Moreover, *gigiprofiler* finds that the data written in the buffer was never used by any client. This direct linkage between prolonged waits and low resource utilization enabled *gigiprofiler* to locate the Apache overuse of the send buffer.

	gigiprofiler		Perf		PCatch		Perspect
ID	**Detect**	**Diagnose**	**Det.**	**Diag.**	**Det.**	**Diag.**	**Detect**
c1	✓	1st	✓	90th	✗	-	✗
c2	✓	1st	✗	-	✓	9th	✗
c3	✓	1st&2nd	✗	-	✓	9th	✗
c4	✓	1st	✗	-	✗	-	✓
c5	✓	1st&2nd	✓	68th	✗	-	✗
c6	✓	1st	✗	-	✗	-	✗
c7	✓	1st	✓	14th	✗	-	✓
c8	✓	1st	✓	5th	✓	1st	✗
c9	✓	1st	✓	9th	✓	5th	✗
c10	✓	1st	✗	-	✗	-	✗
c11	✓	1st	✗	-	✗	-	✓
c12	✓	1st	✗	-	✗	-	✗
c13	✓	1st	✗	-	✗	-	✗
c14	✓	1st	✗	-	✗	-	✗
c15	✓	1st&2nd	✗	-	✓	1st	✓

Table 4: Detection and diagnosis results for the evaluated cases.

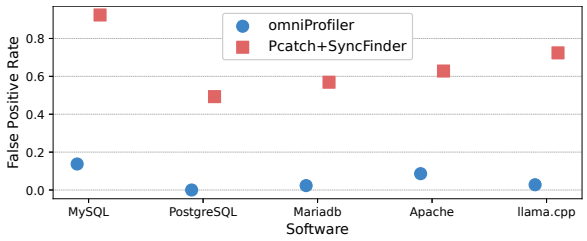


Figure 11: False positive rates across five software.

4.2 Detecting New Performance Issues

To further demonstrate *gigiprofiler* 's effectiveness, we monitored new bug reports from the five evaluated applications. We specifically focused on performance issues with unknown root causes and without clear system-level signals. For each report, we constructed a reproduction script and applied *gigiprofiler* to the corresponding application. During this exercise, *gigiprofiler* successfully diagnosed two new cases of performance issues in application resources. In both cases, the diagnoses were confirmed by developers:

MDEV-34989 [27]. After executing a `SELECT` statement, subsequent `INSERT` operations in the same table would hang for an extended period. *gigiprofiler* identified that the root cause is that if the table is empty, the `SELECT` query holds the table's metadata for too long. This prevents the `INSERT` thread from acquiring the `TABLE_SHARE` resource. The developers confirmed the diagnosis and released a fix.

MDEV-34836 [26]. In a Galera cluster, a read query would unnecessarily hold a parent table while operating on a child table. As a result, a DDL transaction on the parent table would mistakenly operate on the query. However, since the query actually accesses the child table, it would block the DDL transaction on the parent table. *gigiprofiler* identified the root cause by finding that the query that held the parent table never actually accessed it. The developers confirmed this diagnosis and addressed the bug.

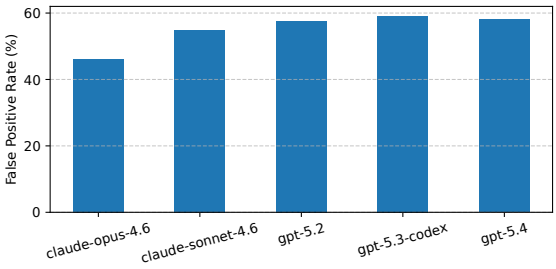


Figure 12: False positive rates across five LLM model

4.3 Accuracy of Finding Software Resources

The precision of identifying application resources is critical for the effectiveness of *gigiprofiler*. To evaluate this, we run the *gigiprofiler* in the five software and manually inspected the accuracy of identifying all the application resource usage events. We compare *gigiprofiler* with SyncFinder+PCatch, the state-of-the-art static analysis tool to identify waiting events in the application code.

As shown in Figure 11, *gigiprofiler* consistently achieves low false-positive rates in the five applications. Even for MySQL, which has the largest codebase in our evaluation, *gigiprofiler* achieves a 13.6% false-positive rate in identifying application-resource usage events. In comparison, the static analysis approach has much higher false-positive rates.

4.4 Sensitivity to different LLM model

Because *gigiprofiler* can use different LLM as the underlying model for its LLM agent, we evaluate whether the coverage and precision of the identification are sensitive to the choice of LLM. We run the LLM agent on MySQL using five different models, including recent GPT and Claude models, e.g., GPT-5.4 and Claude-Opus-4.6, as well as code-agent models(gpt-5.3-codex). To isolate the sensitivity effect of the LLM, we only evaluated the LLM agent without static validation. We manually inspect the identified resources and usage events to determine which candidates are true positives and false positive. For each candidate event, we review the source code to confirm whether the function actually accesses the corresponding application-defined resource.

Figure 12 shows the false-positive rate of each model. All tested models have high false positive rates, ranging from 45% to 60%. The GPT models show similar false-positive rates. The result suggests that the precision problem is not specific to the GPT-4o model used in the main evaluation. Claude-Opus achieves the lowest false-positive rate, but it also identifies fewer candidates(as shown in Sec 4.6)

4.5 Ablation Study on *gigiprofiler* workflow

gigiprofiler uses three modules to improve the accuracy of identifying application-resource usage events: LLM agent, static validation, and post-profiling validation. We conduct

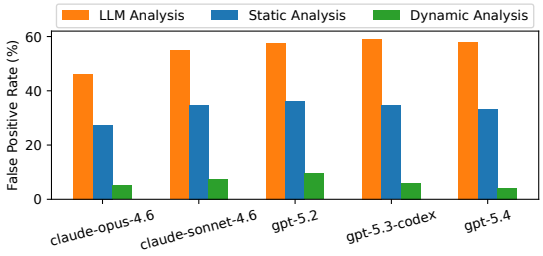


Figure 13: False positive rates in different phases of *gigiprofiler* workflow.

an ablation study to quantify how each module reduces false positives in the identified events. For the LLM agent and static validation, we compute the false-positive rate over all identified events. For post-profiling validation, however, runtime evidence is available only when a workload triggers the corresponding events. We therefore run a set of standard *sysbench* workloads together with our reproduction scripts to cover as many resources as possible, and compute the false-positive rate only over the events observed during execution. We run this study on MySQL using five different LLMs.

As shown in Figure 13, the LLM agent alone produces high false-positive rates, ranging from 45% to 60%. Static validation reduces these rates by 22% on average. Post-profiling validation further reduces the false-positive rate by 24%.

4.6 Coverage of Application Resources

We evaluate whether *gigiprofiler* misses application-resource usage events. Obtaining ground truth on how many application resource events are in a large-scale system is difficult because application-defined resources are not explicitly enumerated in any documentation.We therefore conduct two complementary evaluations. First, we cross-check using the manually validated SyncFinder+PCatch results from Section 4.3. Second, we performed an exhaustive case study on the MySQL buffer-pool subsystem, where we manually enumerate resource-usage functions and events for a small set of well-defined resources.

Cross-check with PCatch. In MySQL, *gigiprofiler* identified 154 usage events but missed 23 valid events found by SyncFinder+PCatch, a false negative rate of 13.0%. These missed events come from two main sources. First, the LLM misses some usage points when comments are absent. Second, the static analyzer misses some resource-access patterns that do not match its validation rules. In comparison, SyncFinder+PCatch misses around 100 usage events found by *gigiprofiler*, a false negative rate of 66.7%.

Case Study: Buffer Pool in MySQL. To obtain a more detailed coverage estimate, we performed a case study on the MySQL buffer-pool subsystem. We focus on three core buffer-pool resources, `buf_pool_t`, `buf_block_t`, and `buf_page_t`. We exhaustively enumerate functions that involve these resources and manually label their usage events. This scope is limited to the buffer-pool subsystem, but gives us a precise ground truth to analyze the events *gigiprofiler* tend to miss.

Model	Missed Events					Missed Functions
	ACQUIRE	USE	WAIT	RELEASE	Total	
Opus-4.6	12	36	38	1	87	25
Sonnet-4.6	10	35	26	1	72	21
GPT-5.2	11	30	29	1	71	26
GPT-5.3-codex	14	35	32	0	81	25
GPT-5.4	11	38	29	0	78	24
Average missing rate	22.7%	24.0%	23.2%	6%	23.0%	12.1%
Total Event	51	145	133	10	339	200

Table 5: Coverage study for the MySQL buffer-pool subsystem. The `ACQUIRE`, `USE`, `WAIT`, and `RELEASE` columns break down the missing events by event type.

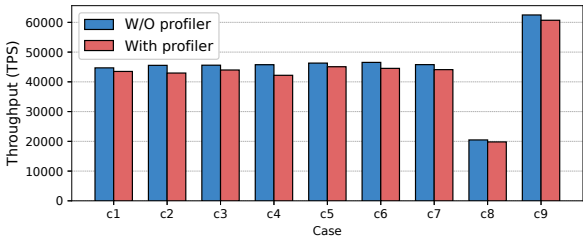


Figure 14: Overhead under 6 cases.

As shown in Table 5, the buffer pool subsystem contains 339 events that span 200 different functions. On average, *gigiprofiler* misses about 23% of events and 12% of functions in the tested LLMs. Among the different models, Opus-4.6 misses the most events, missing 25% of events. We manually checked all missed events and found that most missed events are caused by misclassifications of the event type. For example, If the LLM mis-classifies an event that should be `ACQUIRE` as a `USE` event, the candidate will be checked using the wrong event rule and get rejected. As a result, the true `ACQUIRE` event is missing from the final set of events.

4.7 Runtime Overhead

Finally, we evaluated the runtime overhead of *gigiprofiler*. Because *gigiprofiler* activates its tracing only when a performance issue is triggered, measuring overhead on a generic workload, where tracing is inactive, can not provide a meaningful comparison. We therefore report overhead on 9 of the 15 reproduced issues. We exclude cases 10, 11, and 15 because they lead to CPU hangs making end-to-end performance comparison meaningless. Case 12 is a leak that does not produce measurable throughput degradation. Cases 13 and 14 occur in a web-server setting, where additional client-side instrumentation is needed to trace the overhead.

Figure 14 shows the throughput across the 9 measurable cases. *gigiprofiler* incurs consistently low overhead, averaging 3.7% and never exceeding 7.8%. The observed overhead primarily comes from tracing resource-usage events and writ-

ing them to a ring buffer. Because the current implementation records every event, sampling or batching could further reduce overhead with minimal impact on diagnostic accuracy.

5 Related work

Diagnosing performance bottlenecks in large-scale software systems has been a critical challenge. Existing performance analysis tools can be broadly categorized into general profilers, causality analysis tools, and application-specific methods.

Performance Issue Detection. Profilers [3, 5, 7–9, 11, 13, 14, 39, 57] are fundamental tools for detecting performance problems and identifying associated symptoms. These tools focus on revealing performance-intensive events (e.g., functions, loops, or resource usage) but leave developers to manually infer the underlying reasons for resource contention. Statistical debugging [41, 43, 48] analyzes application-level events to detect performance anomalies. These tools effectively correlate specific code behaviors with performance degradation, making them particularly suitable for identifying issues related to custom synchronization logic or other application-specific patterns. However, interpreting the results of statistical debugging typically requires significant manual effort, as developers must deduce why certain predicates correlate with poor performance. In contrast, *gigiprofiler* identifies application resource usage patterns and directly links these patterns to performance anomalies at runtime, which allows *gigiprofiler* to clearly explain the root causes of performance problems involving resource misuse, thereby reducing the manual investigative effort required by developers.

There are also tools tailored for debugging specific types of performance problems using predefined patterns. For example, loop-oriented debugging tools [34, 44] target performance degradation caused by inefficient or excessive loop iterations, while memory leak detectors [22, 47] focus on identifying and diagnosing memory issues. In contrast, *gigiprofiler* is designed to serve as a general-purpose diagnosis tool, capable of addressing application resource misuse.

Causality Analysis. Numerous techniques have been developed for causality analysis in performance debugging [2, 17, 25, 40]. For example, Argus [50] introduces causality graphs annotated with strong and weak edges, enabling prioritization of relational analyses for complex systems. Similarly, Pensieve [58] demonstrates how runtime logs, combined with static analysis, can reconstruct causal paths and uncover performance bottlenecks. Value-assisted profiling [49] enhances performance diagnosis by capturing and analyzing variable value patterns during execution. While these methods are effective in analyzing runtime interactions and identifying causal dependencies, they rely heavily on existing logs and annotations, which are often unavailable or incomplete for application-defined resources.

LLMs for Software Analysis. Modern large language models have revolutionized code-related tasks, including code generation [10], program repair [12, 42, 51, 52, 55], automated testing [6, 16], and specification generation [23]. Despite their advancements, LLMs face challenges in interpreting application-defined semantics due to their inherent variability.

Recent works [24, 45, 46, 53, 56] combine LLMs with program-analysis techniques for tasks such as binary-code understanding, decompilation, specification generation, and static checker synthesis. These systems use a "propose and validate" approach: it first allows LLMs to propose candidate and then uses program analysis validates these candidates. *gigiprofiler* differs from these systems in two ways. First, *gigiprofiler* targets application-specific resource semantics that can not be validated by standard program analysis. For example, ReSym [53] uses pointer analysis to check the consistency among recovered variables and symbols of the data-structure. In contrast, *gigiprofiler* must decide whether a program variable is a resource defined by the application and whether a function is semantically an event for that resource. There is no standard static-analysis rule for making this semantic judgment. Second, *gigiprofiler* targets large-scale server systems. Existing LLM-assisted approaches that analyze small code fragments do not scale well to large-scale system. They can miss cross-file context, require many expensive LLM calls, and produce low-precision results when the resource semantics depend on broader application logic.

6 Conclusion

Performance issues in application resources are a major source of hidden slowdowns in modern software, yet they remain difficult to detect and diagnose because their effects do not manifest through system-level signals. This paper introduced *gigiprofiler*, a diagnosis tool that exposes the behavior of internal resources in an application by identifying resources defined by the application, tracking how tasks wait for, acquire, use and release them, and describing slowdowns to the underlying code paths. *gigiprofiler* combines LLM-based semantic inference with static analysis to automatically recover resource-usage events across large and heterogeneous codebases, and instruments lightweight runtime tracing to detect pathological usage patterns. Our evaluation on five large applications shows that *gigiprofiler* accurately diagnoses the root cause for all 15 reproduced real-world performance issues in application resources and detected two previously unknown issues confirmed by developers.

Acknowledgments

We thank the anonymous OSDI reviewers and our shepherd for their valuable and detailed feedback that improved our work. This work is supported in part by the NSF CAREER Award 2333885; PRISM, one of the seven centers in JUMP 2.0, a Semiconductor Research Corporation (SRC) program

sponsored by DARPA; as well as generous donations from NVIDIA, AMD, Intel, and Arm.

References

[1] Mona Attariyan, Michael Chow, and Jason Flinn. X-ray: automating root-cause diagnosis of performance anomalies in production software. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, OSDI'12, page 307–320, USA, 2012. USENIX Association.

[2] Mona Attariyan and Jason Flinn. Automating configuration troubleshooting with dynamic information flow analysis. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, OSDI'10, page 237–250, USA, 2010. USENIX Association.

[3] Mike Chow, Yang Wang, William Wang, Ayichew Hailu, Rohan Bopardikar, Bin Zhang, Jialiang Qu, David Meisner, Santosh Sonawane, Yunqi Zhang, Rodrigo Paim, Mack Ward, Ivor Huang, Matt McNally, Daniel Hodges, Zoltan Farkas, Caner Gocmen, Elvis Huang, and Chunqiang Tang. ServiceLab: Preventing tiny performance regressions at hyperscale through Pre-Production testing. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, pages 545–562, Santa Clara, CA, July 2024. USENIX Association.

[4] CISCO. Developers spending more time firefighting issues than delivering innovation. Online report, 2024.

[5] Charlie Curtsinger and Emery D. Berger. Coz: finding code that counts with causal profiling. In *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP '15, page 184–197, New York, NY, USA, 2015. Association for Computing Machinery.

[6] Yinlin Deng, Chunqiu Steven Xia, Chenyuan Yang, Shizhuo Dylan Zhang, Shujing Yang, and Lingming Zhang. Large language models are edge-case generators: Crafting unusual programs for fuzzing deep learning libraries. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, ICSE '24, New York, NY, USA, 2024. Association for Computing Machinery.

[7] OProfile Developers. Oprofile: A system profiler for linux. https://oprofile.sourceforge.io/about/, 2023.

[8] SystemTap Developers. Systemtap: Simplifying system-level observability. https://sourceware.org/systemtap/, 2023.

[9] Valgrind Developers. Valgrind: Instrumentation framework for building dynamic analysis tools. https://valgrind.org/, 2023.

[10] Xueying Du, Mingwei Liu, Kaixin Wang, Hanlin Wang, Junwei Liu, Yixuan Chen, Jiayi Feng, Chaofeng Sha, Xin Peng, and Yiling Lou. Evaluating large language models in class-level code generation. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, ICSE '24, New York, NY, USA, 2024. Association for Computing Machinery.

[11] Linux Foundation. Perf: Linux profiling with performance counters. https://perf.wiki.kernel.org/.

[12] Xiang Gao and Abhik Roychoudhury. Interactive patch generation and suggestion. In *Proceedings of the IEEE/ACM 42nd International Conference on Software Engineering Workshops*, ICSEW'20, page 17–18, New York, NY, USA, 2020. Association for Computing Machinery.

[13] gperftools Developers. gperftools: Google performance tools. https://github.com/gperftools/gperftools, 2023.

[14] Sudheendra Hangal and Monica S. Lam. Tracking down software bugs using automatic anomaly detection. In *Proceedings of the 24th International Conference on Software Engineering*, ICSE '02, page 291–301, New York, NY, USA, 2002. Association for Computing Machinery.

[15] Yigong Hu, Gongqi Huang, and Peng Huang. Pushing performance isolation boundaries into application with pbox. In *Proceedings of the 29th Symposium on Operating Systems Principles*, SOSP '23, pages 247–263, New York, NY, USA, 2023. Association for Computing Machinery.

[16] Sungmin Kang, Juyeon Yoon, and Shin Yoo. Large language models are few-shot testers: Exploring llm-based general bug reproduction. In *Proceedings of the 45th International Conference on Software Engineering*, ICSE '23, page 2312–2323. IEEE Press, 2023.

[17] Baris Kasikci, Benjamin Schubert, Cristiano Pereira, Gilles Pokam, and George Candea. Failure sketching: a technique for automated root cause diagnosis of in-production failures. In *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP '15, page 344–360, New York, NY, USA, 2015. Association for Computing Machinery.

[18] Chi Li, Shu Wang, Henry Hoffmann, and Shan Lu. Statically inferring performance properties of software configurations. In *Proceedings of the Fifteenth European Conference on Computer Systems*, EuroSys '20, New York, NY, USA, 2020. Association for Computing Machinery.

[19] Jiaxin Li, Yuxi Chen, Haopeng Liu, Shan Lu, Yiming Zhang, Haryadi S. Gunawi, Xiaohui Gu, Xicheng Lu, and Dongsheng Li. Pcatch: automatically detecting performance cascading bugs in cloud systems. In *Proceedings of the Thirteenth EuroSys Conference*, EuroSys '18, New York, NY, USA, 2018. Association for Computing Machinery.

[20] Jiaxin Li, Yiming Zhang, Shan Lu, Haryadi S. Gunawi, Xiaohui Gu, Feng Huang, and Dongsheng Li. Performance bug analysis and detection for distributed storage and computing systems. *ACM Trans. Storage*, 19(3), June 2023.

[21] LLVM. Llvm: A compilation framework for lifelong program analysis and transformation.

[22] Chang Lou, Cong Chen, Peng Huang, Yingnong Dang, Si Qin, Xinsheng Yang, Xukun Li, Qingwei Lin, and Murali Chintalapati. RESIN: A holistic service for dealing with memory leaks in production cloud infrastructure. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 109–125, Carlsbad, CA, July 2022. USENIX Association.

[23] Lezhi Ma, Shangqing Liu, Yi Li, Xiaofei Xie, and Lei Bu. Specgen: Automated generation of formal program specifications via large language models, 2024.

[24] Lezhi Ma, Shangqing Liu, Yi Li, Xiaofei Xie, and Lei Bu. Specgen: Automated generation of formal program specifications via large language models. In *Proceedings of the IEEE/ACM 47th International Conference on Software Engineering*, ICSE '25, page 16–28. IEEE Press, 2025.

[25] Jonathan Mace, Ryan Roelke, and Rodrigo Fonseca. Pivot tracing: Dynamic causal monitoring for distributed systems. In *2016 USENIX Annual Technical Conference (USENIX ATC 16)*, Denver, CO, June 2016. USENIX Association.

[26] MariaDB. Mdev-34836: Toi (alter) hangs on a parent table if sr transaction is in progress on a child table. https://jira.mariadb.org/browse/MDEV-34836, 2024.

[27] MariaDB. Mdev-34989: After selecting from empty table with vector key the next insert hangs. https://jira.mariadb.org/browse/MDEV-34989, 2024.

[28] MySQL. Mysql bug #34414: Innodb purge thread causing high cpu usage. https://bugs.mysql.com/bug.php?id=34414, 2008.

[29] MySQL. Mysql bug #54455: Innodb purge thread causing high cpu usage. https://bugs.mysql.com/bug.php?id=54455, 2012.

[30] MySQL. Mysql at facebook. https://www.facebook.com/MySQLatFacebook/posts/10152610642241696, 2014.

[31] MySQL. Mysql bug #74919: Innodb purge thread causes high cpu usage and performance degradation. https://bugs.mysql.com/bug.php?id=74919, 2014.

[32] MySQL. Mysql bug #75540: Purge thread causes performance regression. https://bugs.mysql.com/bug.php?id=75540, 2015.

[33] MySQL. Mysql bug #96236: Performance regression due to excessive purge lag. https://bugs.mysql.com/bug.php?id=96236, 2019.

[34] Adrian Nistor, Linhai Song, Darko Marinov, and Shan Lu. Toddler: detecting performance problems via similar memory-access patterns. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, page 562–571. IEEE Press, 2013.

[35] Percona. Innodb's multi-versioning handling can be achilles' heel Online article, 2014.

[36] Percona. Mysql performance implications of innodb isolation modes Online article, 2015.

[37] Percona. Chasing a hung transaction in mysql: Innodb history length strikes back Online article, 2017.

[38] Percona. Impact of swapping on mysql performance Online article, 2017.

[39] GNU Project. *GNU gprof: a Call Graph Execution Profiler*, 1998.

[40] Lenin Ravindranath, Jitendra Padhye, Sharad Agarwal, Ratul Mahajan, Ian Obermiller, and Shahin Shayandeh. AppInsight: Mobile app performance monitoring in the wild. In *10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*, pages 107–120, Hollywood, CA, October 2012. USENIX Association.

[41] Xiang (Jenny) Ren, Sitao Wang, Zhuqi Jin, David Lion, Adrian Chiu, Tianyin Xu, and Ding Yuan. Relational debugging — pinpointing root causes of performance problems. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*, pages 65–80, Boston, MA, July 2023. USENIX Association.

[42] Dominik Sobania, Martin Briesch, Carol Hanna, and Justyna Petke. An analysis of the automatic bug fixing performance of chatgpt, 2023.

[43] Linhai Song and Shan Lu. Statistical debugging for real-world performance problems. In *Proceedings of*

the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA '14, page 561–578, New York, NY, USA, 2014. Association for Computing Machinery.

[44] Linhai Song and Shan Lu. Performance diagnosis for inefficient loops. In *Proceedings of the 39th International Conference on Software Engineering*, ICSE '17, page 370–380. IEEE Press, 2017.

[45] Zian Su, Xiangzhe Xu, Ziyang Huang, Zhuo Zhang, Yapeng Ye, Jianjun Huang, and Xiangyu Zhang. Codeart: Better code models by attention regularization when symbols are lacking. *Proc. ACM Softw. Eng.*, 1(FSE), July 2024.

[46] Hanzhuo Tan, Qi Luo, Jing Li, and Yuqun Zhang. Llm4decompile: Decompiling binary code with large language models, 2024.

[47] John Vilk and Emery D. Berger. Bleak: automatically debugging memory leaks in web applications. *SIGPLAN Not.*, 53(4):15–29, June 2018.

[48] Shu Wang, Chi Li, Henry Hoffmann, Shan Lu, William Sentosa, and Achmad Imam Kistijantoro. Understanding and auto-adjusting performance-sensitive configurations. *SIGPLAN Not.*, 53(2):154–168, March 2018.

[49] Lingmei Weng, Yigong Hu, Peng Huang, Jason Nieh, and Junfeng Yang. Effective performance issue diagnosis with value-assisted cost profiling. In *Proceedings of the Eighteenth European Conference on Computer Systems*, EuroSys '23, page 1–17, New York, NY, USA, 2023. Association for Computing Machinery.

[50] Lingmei Weng, Peng Huang, Jason Nieh, and Junfeng Yang. Argus: Debugging performance issues in modern desktop applications with annotated causal tracing. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, pages 193–207. USENIX Association, July 2021.

[51] Chunqiu Steven Xia, Yuxiang Wei, and Lingming Zhang. Automated program repair in the era of large pre-trained language models. In *Proceedings of the 45th International Conference on Software Engineering*, ICSE '23, page 1482–1494. IEEE Press, 2023.

[52] Chunqiu Steven Xia and Lingming Zhang. Automated program repair via conversation: Fixing 162 out of 337 bugs for $0.42 each using chatgpt. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2024, page 819–831, New York, NY, USA, 2024. Association for Computing Machinery.

[53] Danning Xie, Zhuo Zhang, Nan Jiang, Xiangzhe Xu, Lin Tan, and Xiangyu Zhang. Resym: Harnessing llms to recover variable and data structure symbols from stripped binaries. In *Proceedings of the 2024 ACM SIGSAC Conference on Computer and Communications Security*, 2024.

[54] Weiwei Xiong, Soyeon Park, Jiaqi Zhang, Yuanyuan Zhou, and Zhiqiang Ma. Ad hoc synchronization considered harmful. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, OSDI'10, page 163–176, USA, 2010. USENIX Association.

[55] Boyang Yang, Haoye Tian, Weiguo Pian, Haoran Yu, Haitao Wang, Jacques Klein, Tegawendé F. Bissyandé, and Shunfu Jin. Cref: An llm-based conversational software repair framework for programming tutors. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2024, page 882–894, New York, NY, USA, 2024. Association for Computing Machinery.

[56] Chenyuan Yang, Zijie Zhao, Zichen Xie, Haoyu Li, and Lingming Zhang. Knighter: Transforming static analysis with llm-synthesized checkers. In *Proceedings of the ACM SIGOPS 31st Symposium on Operating Systems Principles*, SOSP '25, page 655–669, New York, NY, USA, 2025. Association for Computing Machinery.

[57] Dong Young Yoon, Yang Wang, Miao Yu, Elvis Huang, Juan Ignacio Jones, Abhinay Kukkadapu, Osman Kocas, Jonathan Wiepert, Kapil Goenka, Sherry Chen, Yanjun Lin, Zhihui Huang, Jocelyn Kong, Michael Chow, and Chunqiang Tang. Fbdetect: Catching tiny performance regressions at hyperscale through in-production monitoring. In *Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles*, SOSP '24, page 522–540, New York, NY, USA, 2024. Association for Computing Machinery.

[58] Yongle Zhang, Serguei Makarov, Xiang Ren, David Lion, and Ding Yuan. Pensieve: Non-intrusive failure reproduction for distributed systems using the event chaining approach. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, page 19–33, New York, NY, USA, 2017. Association for Computing Machinery.

[59] Fang Zhou, Yifan Gan, Sixiang Ma, and Yang Wang. wPerf: Generic Off-CPU analysis to identify bottleneck waiting events. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 527–543, Carlsbad, CA, October 2018. USENIX Association.

ARTIFACT EVALUATED
USENIX
AVAILABLE

hS: Speculative Script Reordering at Subprocess Granularity

Georgios Liargkovas*,† *Brown University*
Di Jin* *Brown University*
Tianyu (Ezri) Zhu *Stevens Institute of Technology*
Dan Liu *Stevens Institute of Technology*
A. Bolun Thompson *UCLA*
Anirudh Narsipur *Brown University*
Seong-Heon Jung *Brown University*
Siddhartha Prasad *Brown University*
Diomidis Spinellis *AUEB & TU Delft*
Michael Greenberg *Stevens Institute of Technology*
Konstantinos Kallas *UCLA*
Nikos Vasilakis *Brown University*

Abstract

Shell scripts are pervasive, acting as the glue between commands and subprocesses that are written in a variety of languages and perform complex, system-wide effects. Given the black-box nature of these subprocesses, all work that optimizes script performance until now has relied on handwritten annotations that describe subprocess effects. In this paper we introduce hS, a system that brings *out-of-order, speculative execution* to scripts that invoke subprocesses without requiring any user input or annotations about them. hS speculatively executes command instances—typically simple commands, pipelines, or small synchronization regions—dynamically detecting their effects: blocking unsafe ones, like network accesses, and selectively committing independent effects, like file writes, given no conflicts. On a wide range of real-world scripts, hS offers up to 9.3× speedups compared to bash, and up to 7× speedups over PaSh—all while not requiring any developer involvement or command annotations.

1 Introduction

Shell programming is as prevalent as ever, consistently ranking among the top ten programming languages and growing in popularity more quickly than other established languages such as C and Python [24]. Shell programs are often used for various analytics [50, 61], bioinformatics [8, 23], and other performance reliant applications [4, 55, 63]. This has recently led to significant academic activity aimed at accelerating shell scripts through parallelization [29, 54, 62], distribution [42, 50], and other forms of scaleout [18, 35, 55]. This academic work on the shell relies on command annotations to optimize commands, which are either crowd-sourced or baked into each system. Unfortunately, existing approaches face two significant challenges: (1) annotations are restrictive, and (2) they focus primarily on data parallelism.

Existing tools such as PaSh [29, 62] and POSH [50] only optimize fragments of shell programs for which their command annotations fully specify the parallelizability, input order, and side effects of each command. While it is plausible to have a blessed set of annotations for GNU Coreutils or `util-linux`, we cannot hope to have annotations for every command a script might invoke. Some scripts compile and run their own commands—it is unreasonable to assume that *those* commands will be annotated! Relying on annotations significantly reduces the scope and applicability of shell tooling, which is the main way to compose tools written in different languages and frameworks [23, 49]. The usage of shell to compose domain-specific programs are precisely where existing tooling falls short.

Furthermore, existing tools are limited to *order-preserving* data parallelization to avoid introducing concurrency bugs—for example, PaSh and POSH explicitly treat composition primitives, like `;` and `if`, as synchronization barriers and will not typically optimize across any control flow.[1] However, not all order need be preserved! A standard optimization that is done by many modern compilers and architectures is out-of-order execution, where code is reordered, sometimes in a speculative way, and dependencies are dynamically tracked to guarantee that the out-of-order execution will have the same semantics as the original.

In this paper, we propose hS, a novel system that enables out-of-order optimizations in the shell in a fully transparent way: hS executes script regions speculatively out-of-order *without any developer input or annotations*. These regions are the executable units exposed by the shell program's structure—typically simple commands or pipelines, rather than individual Unix processes. In contrast to all prior work, this enables hS to support scripts that compose arbitrary binaries, like bioinformatics tools. hS achieves this: (1) by *dynamically* capturing each region's file-system dependencies and other effects using system-call tracing, reordering the region in the script speculatively at runtime; and (2) by *conditionally* committing or

* Equal contribution.
† Now affiliated with Columbia University.

[1] PaSh does some basic pipelining on loops without carried dependencies.

reverting the effects of a region using lightweight sandboxing techniques.

Orchestrator (§3): Out-of-order execution is challenging to apply correctly as subtle changes can alter the behavior of the original script. The relevant soundness property is *observational equivalence*: the original program and the reordered program must behave the same from the perspective of the user. To ensure that hS's reordering is sound, it implements a novel state-machine-based algorithm that tracks and respects conflicting command dependencies. We use the Alloy [27] model checker to ensure that our algorithm yields observationally equivalent runs that are safe (no deadlocks; dependencies are respected) and sound (speculated commands are committed in the correct, sequential order).

Executor (§4): Speculative execution requires controlling a command's effects on the surrounding system. hS achieves that using a combination of process-level sandboxing and tracing that can contain, block, and selectively defer an individual command's effects. If all of a commands effects are deferrable, e.g., file writes and shell variable assignments, hS can speculate it, containing all its reads and writes, and later commit its effects if it doesn't depend on other previous commands. Even though commands with non-deferrable effects, e.g., network accesses, cannot be speculated, hS still guarantees correct execution by blocking their effects and executing the commands exactly once, without speculation.

Optimizations (§5): Speculative execution can lead to redundant recomputation when mis-speculating, *i.e.*, if a dependency indicates that some speculative run of a command is invalid, we must throw away its results. To improve efficiency, hS introduces a set of domain-specific optimizations that reduce the number of mis-speculations by predicting the starting state from which to speculate commands. First, hS reasons about the shell language directly, eagerly and efficiently evaluating pure assignments without any need to speculate. Second, hS is optimistic, speculating each command in the freshest, potentially uncommitted file-system state, while eagerly stopping speculations that have conflicts.

hS is able to optimize the execution time of arbitrary shell scripts without any user input or command annotations. On a diverse set of 49 real-world shell scripts, some of which have already been optimized to exploit command independencies by their authors, hS achieves a geomean 2.6× speedup (up to 9.3×) over Bash. hS also achieves significant speedups over the prior state of the art, PaSh [29, 62] (geomean: 2.4×, max: 7×), while not requiring any command annotations to enable optimizations, thus being able to support a wider variety of scripts with custom and domain specific commands and utilities. Finally, to show that our techniques are applicable more broadly to scripts that use subprocesses, we develop a Python frontend for hS that can optimize a restricted fragment of Python programs that invoke subprocesses (§6). On a set of 5 real-world Pythons programs hS manages to achieve

```
SAMPLES="ERR1233.1 ERR2445.1 ERR3771.1 ERR4462.1 ..."   # (a)
# (1) Sanitize FASTQ file headers
for SM in $SAMPLES; do
  fastq-sanitize-header --input $SM |
    gzip > $SM/1.sanitize.fastq.gz                      # (b)
done
...
# (2) Genome alignments with minimap2
for SM in $SAMPLES; do
  # Alignment to transcriptome and genome
  minimap2 -ax splice -t 6 ... $SM/1.sanitize.fastq.gz |
    samtools sort -o $SM/Genome.sorted.bam -            # (c)
  minimap2 -ax map-ont -t 6 ... $SM/noribo.fastq.gz |
    samtools sort -o $SM/Transcriptome.sorted.bam -     # (d)
  ...
done
...
# (3) Index genome alignment and convert to SQLite
./prep_db.sh $dbdir/sqlite.db                           # (e)
for SM in $SAMPLES; do
  dbdir=$SAMPLE_DIR/$SM                                 # (f)
  samtools index $SM/Genome.sorted.bam                  # (g)
  samtools view $SM/Genome.sorted.bam |
    sam_to_sqlite --db $dbdir/sqlite.db --table genome  # (h)
  # ... same for Transcriptome.sorted.bam
done
```

Figure 1: The core of a bioinformatics script for RNA sequence processing [23].

speedups of up to 8.4× over standard Python execution.

Availability: hS is open source and available at https://github.com/binpash/hs.

2 Speeding Up Everyday Data Processing

We first explain how hS speeds-up shell scripts using an example: a bioinformatics script that processes RNA sequencing data from mouse samples, taken from the TERA-Seq toolkit [23] (Fig. 1). The script preprocesses genomics data `(1)`, aligns RNA samples to reference genomes or transcriptomes `(2)`, and conducts post-alignment analysis `(3)`.

The script leaves a lot of performance on the table by not taking advantage of various optimization opportunities. First, each sample could be processed independently: for example, alignment's `(2)` two steps (`c`, `d`) are independent and can be performed in parallel. Additionally, some processing could start earlier: the script starts building the SQLite databases `(3)` only after *all* alignments complete (`c`, `d`), even though it would be safe to start building the database for a sample (`g`, `h`) as soon as that sample is aligned.

Optimization Challenges: Identifying independent execution optimization opportunities in shell programs is not trivial. One must accurately understand each command's behavior and the implicit data dependencies between commands. Individual shell commands are black boxes, written in arbitrary languages, sometimes with unavailable or inscrutable source code. Often, commands have implicit dependencies; for example, `samtools index` generates an index file that is not mentioned in its arguments, but whose name is based on the

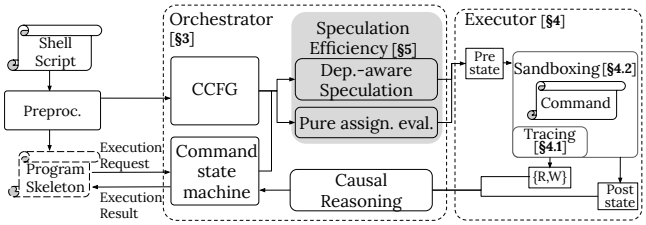


Figure 2: High-level architecture of hS.

input file's name. Even if programmers know all commands' dependencies, independent execution has to be embedded in the script logic, a process that is tedious, error-prone, and complicates development and maintenance.

The state-of-the-art tools for parallelization of shell scripts, PaSh [29] and POSH [50], cannot exploit these opportunities—neither offers any benefits on this script (PaSh takes 7918s vs Bash's 7914s). In their default configuration, neither tool can reason about or optimize invocations to domain-specific tools like `fastq-sanitize-header`, `minimap2`, and `samtools` as there are no annotations to characterize their behavior. As we show in §7.2, even conservative annotations for a representative TERA-Seq workflow are insufficient to expose substantial PaSh speedups without domain-specific split/merge support.

2.1 Overview of hS

To understand how hS is able to correctly and effectively optimize scripts, we overview the system architecture (Fig. 2) in the context of a single iteration of a fragment of the analysis stage of the bioinformatics script (Fig. 1(3)). Before any iterations, the analysis stage runs `./prep_db.sh` (e). Then, for each iteration's sample, it will index that sample's alignment (g) and generate a SQLite database for that sample (h).

Preprocessing: hS uses script rewriting [29] to transform the input script into two objects: a *program skeleton* and a *command control-flow graph (CCFG)*. The program skeleton is run by the real shell interpreter, while the CCFG is managed by hS's orchestrator. The program skeleton preserves the original script's control structure, but replaces executable regions with stubs that communicate with hS's orchestrator. We call each concrete execution of such a region a *command instance*. Despite the name, a command instance is not necessarily a single Unix process: it may be a simple command, a pipeline, or a small shell region needed to preserve explicit synchronization. The CCFG abstracts the script by tracking these regions and how they relate with respect to control flow. In our example, the CCFG (Fig. 1 (top left)) puts (e) first, followed by a loop that contains sequences (g) and (h). Regions in loops can map to multiple command instances in the concrete execution—one per loop iteration.

Command Control Flow Graph (§3 Orchestrator): hS's *orchestrator* is a server that responds to execution requests from the hooks in the program skeleton. The orchestrator manages the CCFG to decide which command instances to execute, possibly speculatively. Sequential execution would run the next instance in the CCFG once the prior instance completes; hS's orchestrator speculates by executing instances further ahead in the CCFG.

Shell construct handling: hS's preprocessing pass is syntax-directed: it keeps the shell in charge of the script's control structure, and exposes executable regions to the orchestrator. A pipeline such as `A | B | C` is one command instance: shell pipe semantics are preserved inside the sandbox, while hS tracks and commits the aggregate dependencies and effects of the whole pipeline. Similarly, a background/wait pattern such as `A & B & wait` is treated as a shell-synchronization region, so hS does not reorder `A` and `B` with respect to each other or check their internal conflicts separately; only the region's combined effects are checked against earlier and later command instances.

Control constructs determine which command instances are reached. Conditionals, cases, loops, short-circuit operators, groups, subshells, and redirections remain in the program skeleton and are evaluated by the shell in the original order. The CCFG exposes possible future instances to the orchestrator, including loop iterations, but a speculative result is committed only when the skeleton later requests that concrete instance and its shell and file-system starting states are compatible. Commands that cannot safely be executed outside the skeleton—for example shell control-flow builtins such as **`break`**, **`continue`**, **`return`**, and `exit`, or commands with non-deferrable effects—are marked unsafe and executed once by the skeleton, acting as speculation barriers. Finally, a nested hS invocation is treated like any other subprocess by the outer hS: the outer execution remains governed by the same dependency and effect checks, but it may expose less parallelism than a single run with one global CCFG.

Command dependencies and effects (§4 Executor): Instead of running each command in order, hS's orchestrator speculatively executes subsequent commands (like (g) and (h)) earlier than in sequential execution. Correct execution depends on only committing the results of commands unaffected by out-of-order execution. The orchestrator commits the effects of speculative executions if previous commands did not affect their execution—*i.e.*, if the speculated commands didn't depend on effects of prior commands. hS is able to speculatively execute commands without jeopardizing execution correctness by tracking each command's *dependencies—i.e.*, what it reads from—and by containing and deferring its *effects—i.e.*, what it writes to. hS does that using its *executor* (§4), which tracks dependencies and effects on two types of state: (i) the shell state (*incl.* environment variables and shell configuration), and (ii) the file system state (*incl.* files, terminals, and file descriptors). The executor identifies command dependencies in a completely black-box

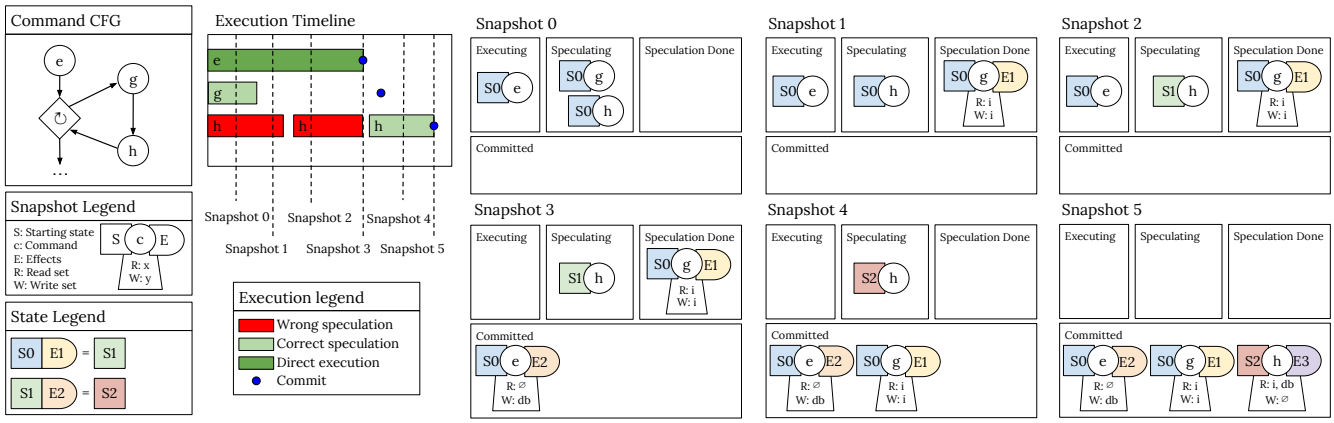


Figure 3: Example hS execution on fragment (3) (Fig. 1). **Top left**: Command control-flow graph and execution timeline; CCFG nodes denote executable script regions, and each concrete execution of such a region is a command instance. **Bottom left**: Legends. **Right**: Each snapshot represents the state of the orchestrator (§3) at a point in the execution timeline.

manner using system call tracing (§4.1) and contains effects using lightweight sandbox techniques, namely OverlayFS and namespaces (§4.2). While hS is able to optimize the execution of commands with effects that are *deferrable*, e.g., file writes and shell variable assignments, it guarantees correct execution even in the presence of non-deferrable effects, e.g., network accesses—when a command performs such an effect, hS blocks it and marks the command as unsafe for speculation, later issuing it for normal execution.

2.2 Speeding up the Script

We will now walk through an execution of hS on a fragment of the RNA processing script (Fig. 1(3)). Figure 3 shows a sequence of execution snapshots and the timeline. Execution starts with snapshot 0, where (e) is issued for normal execution while (g and h) are issued for speculative execution.

Execution snapshot 1: For this example, we assume that building the index (g) completes before the database preparation (e) (Fig. 3, Snapshot 1). When a command completes execution, the executor passes the command's traced dependencies and contained effects to the orchestrator. Here, (g)'s dependencies are the files that it read from, including `$SM/Genome.sorted.bam`, and its effects are that it generates the index file (abbreviated to *i* in Fig. 3) in either `$SM/Genome.sorted.bam.bai` or `$SM/Genome.sorted.bam.csi`. Note that these files aren't explicitly referenced in the command arguments, but implicitly chosen by `samtools index`—which one is chosen depends on the size of the input genome.

Execution snapshot 2: In the meantime, the `samtools view` command (h) has been speculatively executing in a state that does not include the indices generated by (g). At some point, hS will observe a system call by (h) that tries to read these indices and fails, since its computation *implicitly* relies on them. Upon discovering this dependency between (g) and (h)—(g) writes a file that (h) reads—the orchestrator determines that (h) has been mis-speculated, since (h) would only run after (g) in a sequential execution, but hS has run it earlier. The orchestrator then drops the effects of (h), and re-speculates it on top of the (as yet uncommitted) effects generated by g.

Execution snapshot 3: After the database preparation completes (e), the orchestrator can immediately commit the results of (e) to the live system—(e) was the first command, so there can be no mis-speculation or dependency violations. Even so, (e)'s effects are still recorded for later reasoning.

Execution snapshots 4 and 5: The effects of (e) include writes to `$dbdir/sqlite.db`, meaning that (h) will need to be re-speculated once more—the second command in its pipeline reads from `$dbdir/sqlite.db`. The orchestrator observes that (e) and (g) are completely independent, and it can immediately commit the speculated results of index building (g) (Fig. 3, Snapshot 4). Since every prior command has committed when (h)'s third execution completes, the orchestrator can directly commit (h) when it completes (Fig. 3, Snapshot 5).

Pure assignments (§5.1): Note that the assignment (f) between (e) and (g) is *not* part of the CCFG. While assignments can have dependencies (globbing) or external effects (*e.g.*, command substitutions `$(...)`), many of them do not. hS identifies assignments without effects—*i.e.*, *pure* assignments—and simply includes them in the starting speculation state of all following commands. This optimization significantly reduces CCFG size and speculation overhead.

Zooming out: Having examined a single iteration of the `for` loop, we now consider the entirety of the post-processing stage (Fig. 1(3)). Again, we assume that (g) completes before (e). hS uses a configurable speculation window to limit how many commands can be speculated at any time. For example, with a speculation window of two, we will have three com-

mands in execution: (e) executing directly, and (g) and (h) in speculation. When (e) commits, a slot in the speculation window opens up and the orchestrator continues speculation following the would-be sequential execution of the program, by speculating (g) in the next iteration of the loop, even though the prior iteration has not completed.

Dependency-aware speculation (§5.2): When two commands in speculation have recorded dependencies, such as (g) and (h), they should not be speculated starting from the same state. hS avoids such mis-speculations by employing a *dependency-aware speculation strategy* (§5.2): we only start commands speculation after all the (discovered) dependencies finish, and we create a speculative starting state that combines the effects of all previously speculated commands.

Summary: Through speculative execution and the aforementioned optimizations, hS optimizes shell programs by executing independent script segments out-of-order, trying to take advantage of the latent parallelization opportunities. In our example script, hS achieves a 4.5× speedup over sequential execution using Bash (§7).

3 Orchestrator

The orchestrator (Figure 2, middle) manages speculation by keeping track of the command control-flow graph (CCFG) and all command instances in the script. The orchestrator uses the CCFG to look ahead and speculate commands, such that it can return pre-computed results to the program skeleton, achieving effective parallelization. At the same time, it keeps track of commands with unsafe behavior and dependencies across command instances to make sure that they are executed and committed in a safe order. In this section we describe the orchestration algorithm, which uses a state machine (Figure 4) to track the execution state of each command instance.

3.1 Orchestration Algorithm

The orchestrator manages two data structures: the *command control-flow graph* (Fig. 3, top left) and the *set of command instances* (Fig. 3, snapshots). The CCFG is generated from the source program, relating each executable region back to its position in the program skeleton. As introduced in §2, each command instance corresponds to one concrete execution of such a region. In loops, a single CCFG region may therefore have several command instances in the set, one per iteration. A command instance keeps track of: (1) the command's current speculation state (§3.1); (2) the results of speculation, if any; and (3) the command's read and write sets.

State machine without speculation (Ⓡ, Ⓔ, Ⓒ): Each command instance is associated with a *command state* (Fig. 4). To explain command states, we start by considering a sequential orchestrator that doesn't speculate but simply executes a command instance whenever the program skeleton requests

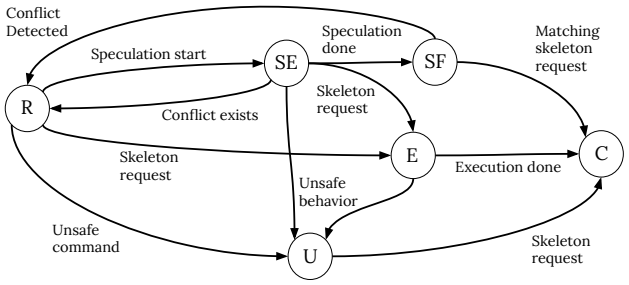


Figure 4: State machine for each command instance.

its execution. At first, every command instance is in Ⓡ, the 'ready' state. When execution begins, the program skeleton requests the execution of the first command instance in the script. The orchestrator then sends the first command instance to the executor, marking it as Ⓔ, the 'direct-execution' state. When the first command instance terminates, the orchestrator commits its saved results, moving the command to Ⓒ, the 'committed' state, responding to the skeleton that it is done. As the program skeleton continues, it makes another execution request—this time, for the second command instance. The orchestrator executes it, running it in the most recent state, *i.e.*, the state after the first command instance has committed, and then commits the results, at which point the skeleton requests the execution of another command until all of them are done executing in the correct, original, sequential order.

Unsafe commands (Ⓤ): Some commands—shell control-flow utilities, like `break`, or commands with non-deferrable effects, like network accesses—cannot be speculated and need to be run *exactly* once by the program skeleton. The orchestrator dynamically detects such commands (§4) and moves them to Ⓤ, the 'unsafe' state. When a command *C* is determined to be unsafe, the orchestrator waits until the program skeleton requests the execution of *C*, at which point it informs it to execute the command itself, in the context of the original script. Upon receiving the next execution request, the orchestrator will move the Ⓤ node into Ⓒ, indicating the command has been executed and its effects are globally visible.

Speculative states (Ⓢⓔ, Ⓢⓕ): Having understood a sequential orchestrator, we now introduce speculation and the remaining states. A speculating orchestrator finds the next command in the CCFG in the Ⓡ state, sends it for speculative execution (§4), and transitions it to Ⓢⓔ, the 'speculatively executing' state. An instance in Ⓢⓔ can make one of three transitions: it can complete its execution, in which case the orchestrator records the result and moves it to Ⓢⓕ, 'speculation finished'; it can be marked as unsafe and transition to Ⓤ; or, the program skeleton can send a request for this instance after speculation has started. When the skeleton sends a request for a command in Ⓢⓔ or Ⓢⓕ, the execution takes a different path depending on whether the speculation starting state is equivalent to the current, real system state or not.

System states and compatibility: Before describing the two paths, we describe what system states are. The system state contains two components: (1) the shell state (*incl.* shell variables, functions, *etc.*), and (2) the filesystem state (*incl.* all files, directories, metadata, *etc.*). The shell state a set of key-value pairs [17, 29] and so comparing two of them amounts to checking for differences point-wise. Comparing filesystem state involves two challenges: (1) representing and storing the *entire* contents of the filesystem is not efficient, and (2) even though there might exist a difference between two states, it might not be relevant for the command instance in question.

File system states as write sets: To address the first challenge, hS represents the FS state as the effects of a command on the file system, namely only the values of the files in its write set (instead of the FS complete contents). For example, the resulting file system state for the command `echo "hello" >f1.txt` only contains the value of file `f1.txt`. hS manages to get such a fine-grained representation of the file system state through its executor (§4).

State equivalence: For the second challenge, consider script `echo "hello" >f1.txt; touch f2.txt`. Even though the output filesystem states after executing these two commands differ—the former has an updated `f1.txt` and the latter read `f2.txt`—there is no dependency between them. To address this, the orchestrator keeps track of all commands reads and writes through the executor (§4) and continuously checks if there are any dependencies. Suppose we are executing a two-command program where the second command reads the results of the first, e.g., `echo "hi" >f1.txt; cat f1.txt`. If we speculatively start executing both programs in the initial filesystem state, `cat` will not see the final correct state of `f1.txt`. When the orchestrator discovers that `echo` writes to `f1.txt` and `cat` reads from `f1.txt`, the dependency is recorded and the orchestrator cancels `cat`'s speculative execution (ⓈⒺ or ⓈⒻ), and moves it back to Ⓡ.

Handling conflicts during speculation: When the orchestrator receives a request for a command instance in ⓈⒺ or ⓈⒻ, it checks if the starting speculation state conflicts with the current real-system state, given the command instance's dependencies. If there is a state conflict, we have mis-speculated and the command instance needs to transition back to Ⓡ. If the states are compatible, we can commit the results of the command instance. An instance in ⓈⒺ will transition to Ⓔ and keep executing until it hits Ⓒ; an instance in ⓈⒻ can be committed immediately, transitioning to Ⓒ directly.

Avoiding speculation across Ⓤ commands: hS does not generate tracing directly in program skeletons (§4.1), so it is impossible to determine the write sets of Ⓤ commands. To ensure that execution is correct, every execution of a Ⓤ command invalidates all later command speculations, transitioning them back to the Ⓡ state.

Orchestrator correctness: We built a model of the orchestrator's core logic in Alloy [27], and we used its bounded model checking capabilities to check several key properties: (1) the orchestrator eventually terminates (no deadlocks); (2) file system dependencies are respected, *i.e.*, a command will be rerun if a command preceding it writes to a file it reads from; and (3) the final command commit order is equivalent to the sequential execution order. The checks are performed over a bounded model with a finite number of files and commands, under the assumption that all commands eventually terminate with arbitrary read/write dependencies.

Speculation policy: hS uses a configurable *speculation window* to limit the amount of speculation. When the window is full, i.e., the number of commands in the ⓈⒺ or ⓈⒻ state are equal to the window size, the orchestrator cannot issue new command instances for speculative execution. Finally, there are points where hS has to pick between more than one command instance to speculate next, e.g., branches of a conditional—a problem akin to branch prediction. This choice does not affect correctness, but it affects performance. hS currently has a fixed policy for these predictions, e.g., loops will keep iterating and conditionals will take the else branch, and we leave a better policy as future work.

4 Executor

The executor's main goal is to identify, contain, and when possible defer effects that commands make to the surrounding state to enable speculation. Before we describe the two key orchestrator components, tracing and sandboxing, we describe which events can be deferred and which can only be blocked.

Deferrable: These effects can be identified by the executor, contained, and deferred to enable speculative execution. These effects include shell state changes (*i.e.*, variables, current working directory, shell options, and open file descriptors), as well as changes to regular files, directories, and a small set of pseudo-files without state (*e.g.*, `/dev/null`).

Non-deferrable: These effects cannot be deferred, but can be detected and blocked, allowing the orchestrator to mark the culprit command as unsafe for speculation and execute it normally without speculation—the rest of the commands can still be safely executed as described in Section 3.1. These effects include network interactions, IPC, and signals.

4.1 Tracing

hS's executor traces system calls to detect all effects. For deferrable effects it traces all path-related system calls and returns a list of paths that the command reads from and writes to. This information is then used by the orchestrator to determine the file-system dependencies. To improve efficiency, hS does not trace all read and write calls, but considers a file to be accessed the moment it is *acquired*, *i.e.*, when its path is passed as a parameter to system calls like `open`, and `rename`. hS uses

seccomp-BPF for `strace` [40] to avoid context switching on untraced system calls.

Path dependencies: hS determines read and write dependencies over paths and not files. For example, an `ENOENT` (no such file) error on a `stat` system call counts as a read to the path even though it didn't read any file. This is because if any previous command creates a file in that path, the result of `stat` will change, affecting the command.

Parent directories: Parent directories require special handling due to the way modifications affect their children. More specifically, a write effect (*e.g.*, `rename`) at the parent directory may alter the child paths without specifically targeting them. Therefore, the tracer treats any effect on a path as read effects on all its parent directories as well, such that the read-write conflict can correctly capture this indirect influence.

Symbolic links: Special care needs to be taken when tracing the creation and use of symbolic links. A symbolic link involves two paths: a link path and a target path. The *creation* of a symbolic link (`symlink`, `symlinkat`) is not affected by the file-system state at the target path. Therefore, although the target path is an argument to the system call, it does not count as either read or write to the path. The *use* of symbolic links, on the other hand, should be treated as *reading* the link path in addition to the effect on the target path. This is because changing to the link path will lead to changes in the outcome of the system call (*i.e.*, opening a different file).

4.2 Sandboxing

hS's sandbox supports two types of deferrable effects: file-system effects and shell-state effects. It handles file-system effects with OverlayFS [7], creating a private view of the file system and storing side effects into separate directories. The shell-internal states are handled by creating a separate shell process and running additional setup and recording utilities at the start and end of the command. Non-deferrable effects are blocked using Linux namespaces [38]; after they are detected with tracing, hS marks such commands as unsafe.

Shell state handling: The shell state includes the (1) current directory, (2) variable and function definitions, (3) shell options, (4) active trap handlers, and (5) opened file descriptors. The executor needs to correctly set up these components when executing any command instance. At the beginning of each runtime hook invocation in the program skeleton, the shell state is recorded, and sent to the orchestrator, which then passes it to the executor. After a command instance finishes execution, the shell state is recorded and sent back to the program skeleton, which then applies the new shell state at the end of the runtime hook invocation.

File system effects: OverlayFS is a union file system that stacks one or more *lower layers* (*i.e.*, 'lowerdirs') and an *upper layer* (*i.e.*, 'upperdir') into a single, merged view. OverlayFS searches for a path starting from the highest layer, meaning that each higher layer shadows lower ones by adding or removing files from them hS isolates commands by setting their root directory to be OverlayFS's merged view using `unshare` [41]. The upperdir records all file-system writes in the sandbox and isolates them from the surrounding system. To commit a write, hS traverses the upperdir and copies the changes back to the real file system. A subtle challenge that hS needs to address is that OverlayFS disallows any pre-existing mount point that cannot be unmounted by the current user to be part of a lowerdir [28]. hS addresses this issue by using mergerfs [1] to shadow these mount points.

Challenge: file descriptors: A challenge that hS needs to address is tracking changes to open file descriptors, i.e., `stdin`, `stdout`, `stderr`, and other files opened with the `exec` command. This differs from the individual command's output redirections, which are already handled by the sandbox's overlay. These file descriptors are opened for the shell program itself, and usually stay opened until the entire program exits. Capturing and controlling the state of file descriptor for every kind of file is beyond the scope of this work, especially those that interact with users or devices. hS supports file descriptors if they are sequentially read from or written to. When recording the state of the file descriptors, hS reads from `/proc` to determine the file descriptor's permission, file path, and current offset. For `stdin` and read-only file descriptors, hS re-opens the corresponding files and uses `lseek` to set the correct offset. For `stdout`, `stderr`, and write-only file descriptors, hS opens a new empty file for each to record the written content. When committing, hS appends the newly written content to the corresponding opened files. Since all the files are re-opened inside the overlay in a separate process, their effects are also isolated from the rest of the system.

If two file descriptors are aliases of each other (*e.g.*, the shell program is run with `2>&1`), their write effects should not be recorded separately; otherwise the intended interleaving of output will be broken. In such cases, hS detects aliasing of file descriptors with the `kcmp` system call, and sets up the aliasing inside the sandbox with proper `exec` redirection, instead of opening separate record files.

Blockable effects: hS uses namespaces and `unshare` to disallow all blockable effects. Linux namespaces [38] are designed to isolate *any* globally visible state changes introduced by a process. Under a non-malicious setting it is reasonable to assume that no unintended effects escape their namespace. hS sets up fresh network, pid, and user namespaces so speculated processes cannot interfere with the outside world through network, signals, and other process control features provided by the operating system. The `/proc` filesystem and a limited set of `/dev` devices are remounted within the namespace to avoid exposing control to other processes or devices.

5 Improving Speculation Efficiency

A key challenge that needs to be addressed by hS is managing mis-speculation. To address that, hS deploys two optimization techniques, pure assignment evaluation (§5.1) and dependency-aware speculation scheduling (§5.2).

5.1 Pure Assignment Evaluation

hS treats any difference in shell state as a conflict, meaning that all shell state changes invalidate all speculated (SE) and (SF) commands. This leads to significant wasted resources, especially in programs with multiple variable assignments.

To address this challenge, and avoid re-executing commands unnecessarily, hS identifies and evaluates pure assignments on the starting shell state. Pure assignments are just variable assignments where the right-hand side does not perform side effects, *e.g.*, `(f)` in Fig. 1. The insight that hS exploits is that pure assignments do not need to be speculated at all, they can simply be run. Furthermore, they can be identified using a simple syntactic pass, ahead of time, during preprocessing. By evaluating pure assignments, the speculator both reduces the number of failed speculations (since the starting shell state of commands is computed correctly), and increases the number of speculated commands that take more time to execute (since pure assignments are no longer considered candidates for speculation).

5.2 Dependency-aware Speculation

To maximize the possibility of correct speculation and minimize resources spent on mis-speculation, we deploy a two-part strategy to make smart speculation decisions about file-system states in hS.

Eager conflict detection and timely restart: Instead of receiving the read-write sets of a command instance after it has completed execution, hS consumes tracing information (§4.1) in a streaming fashion, continuously checking for file-system conflict between executing and speculating commands. Whenever such conflict is discovered, the target command, which is always in (SE) or (SF) state, will be stopped, and reset into (R) state. On top of that, such dependency is recorded by hS. Whenever a command finishes speculation, hS looks up the recorded dependencies, and restarts any commands that have a dependency to the finished one. At this point the following strategy comes into effect.

Opportunistic speculation chaining: Since conflicted speculations are reset to (R) state eagerly, preceding commands that are able to stay in (SF) state are more likely to be committed. Therefore it is beneficial to take their state changes into consideration when launching new speculation. More specifically, suppose a command C_2's only file-system dependency is another command C_1, which just finished speculation. Instead of starting from the current file-system state S, hS starts new speculations on a "predicted" state that contains the effects of C_1. In the case that C_1 is successfully committed, C_2's speculation will not have conflict with the new real file-system state, which will be equivalent to its starting state. In the general case, hS chains all preceding uncommitted effects in (SF) states into the starting states of new speculations. hS implements this state chaining, by layering multiple lower directories into the overlay 4.2, allowing commands to run on an environment with the freshest file-system state available to hS, without it yet being committed in the underlying system.

6 Implementation

The hS implementation comprises 2.4k lines of Python for the preprocessor, orchestrator, executor, 478 lines of shell code for the sandboxing and wrappers, and 285 lines of Alloy for the orchestrator model (§3.1).

Python frontend: As a proof of generality beyond the shell, we also prototype a Python frontend for hS, targeting Python programs that invoke subprocesses, e.g., using `subprocess.run`, to implement their functionality. The frontend is the only thing that changes to support Python, and the orchestrator (§3) and executor (§4) can be used as is. The frontend is not meant to be comprehensive and is currently only able to optimize a limited fragment of Python applications for which (1) the subprocess calls can be easily extracted, i.e., they can be syntactically identified in the program execution, and (2) the subprocess arguments can be easily computed without any side-effects. The frontend currently assumes that a Python program does not include any control flow that affect which external commands run besides for loops, that all the functionality happens in a single `main` function, and that all file system effects go through an external command rather than builtin functions like `sys.write`. Supporting the optimization of arbitrary Python applications goes beyond the scope of this paper and would require significant improvements on Python program analysis that are not possible with the current state-of-the-art. The implementation of the Python frontend consists of an additional 547 lines for a Python-specific preprocessor and runtime.

7 Evaluation

To evaluate hS, we have collected a wide variety of shell scripts from the wild (§7.1), on which we run experiments highlighting various dimensions of hS's performance and correctness. We examine hS's overall performance (§7.2), by comparing it to Bash and PaSh [29, 62] on all the scripts in our suite. We also measure the amount of redundant resource usage of hS due to mis-speculations (§7.3). We then evaluate the overheads of hS's executor using various benchmarks (§7.4),

Table 1: Summary of all benchmarks used to evaluate hS, showing the number of scripts, lines of code, input size, execution time with bash, and source.

	Bench. Set	Scripts	LoC	Input	Exec. Time	Source
1	TERA-Seq	9	2k	177G	25m-8h	[23]
2	Genomics	1	22	97G	1.3h	[8]
3	Sklearn	1	102	0.8G	7m	[4]
4	DGSH	10	308	14G	<1s-40m	[55]
5	NLP	23	204	78G	10s-20m	[9]
6	NOAA	1	20	80G	40m	[63]
7	Unix50	1	58	23G	40s	[46]
8	COVID-mts	1	49	14G	6m	[61]
9	LogAnalysis	1	25	3G	17m	[50]
10	WebIndex	1	67	8G	6m	[62]
11	µbenchmarks	3	82	—	—	custom

as well as its performance under varying window sizes (§7.5). Finally, we evaluate the performance of hS's Python frontend comparing it to standard Python execution on a set of real-world Python programs (§7.6).

Setup: All experiments were conducted on CloudLab [13] `r6525` machines equipped with two 32-core AMD EPYC 7543 CPUs running at 2.8GHz with 256GB of DDR4 RAM, and NVMe SSD storage. The OS was Ubuntu 22.04.2 LTS was kernel version 5.15.0-122-generic, Bash version 5.1.16, PaSh version 0.12.2, and Python version 3.10.12. Docker (27.3.1) was used for the experimental environment. hS is configured with a speculation window of 30 unless noted otherwise, and PaSh is configured with width of 30.

7.1 Benchmarks

For our evaluation we have collected a variety of real-world scripts from the wild (Table 1). Our benchmark selection is complementary to recent shell benchmark characterization efforts such as Koala [33]; we include benchmarks used by prior shell-optimization systems and add domain-specific, long-running workflows such as TERA-Seq to stress scripts with custom tools and implicit dependencies.

TERA-Seq [23] consists of real-world workflows for RNA sequencing that use domain-specific tools and are applied to a multi-GB dataset. These applications run for multiple hours and are on the critical path of bioinformatics research so the authors have already optimized them using `&` and `wait` to exploit command independence and parallelization.

Genomics [8] is a single script that processes 97GB of NGS data, producing chromosome-specific, indexed BAM files, making use of nested loops to iterate across samples.

Sklearn [4] runs a regression model on the kdd99 [57] datasets using custom commands that wrap Scikit-Learn [48].

DGSH [55] contains a collection of shell scripts for text processing, compression, code metrics, log analysis, and git repository analysis. We transliterate the original scripts from the dgsh-dialect to be POSIX-compliant.

NLP [9] contains 23 shell scripts from UNIX-for-poets that perform various text processing tasks using UNIX and Linux utilities. Their input is a set of 120 books amplified to 100× their original size.

NOAA [63] processes multi-year temperature data across the US during the 2000s, consisting of more than 100k compressed files with up to 500MB size. We replace instances of iterations using `seq` and `xargs` with equivalent **`for`** loops.

Unix50 [46] includes classic and recent (circa 2019) scripts making heavy use of UNIX and Linux built-in commands. We evaluate hS on a cumulative script that contains all 36 scripts together on an input of 21GB.

COVID-mts [61] analyzes public transit data during the COVID-19 pandemic, consisting of five pipelines that process 10GB of input data (originally 3.4GB) using UNIX and Linux utilities. We combine the five pipelines into a single script.

LogAnalysis [50] analyzes 2.8GB of web server logs from the World Cup '98 website [2] over a period of 53 days. We extended the original pipeline [50] to handle compressed data and to process multiple log files.

WebIndex [62] is a large, multi-stage script for crawling and indexing Wikipedia. The script uses various utilities like `pandoc` [34] and `awk` to convert HTML to text and extract metrics. We evaluate the script on a fraction of the original input (7280 files, out of approximately 1.5 million).

7.2 Overall Performance and Correctness

Results (Fig. 5): We evaluate hS compared to Bash and PaSh on the 10 benchmark sets, totaling 49 shell scripts and 2.9k LOC. We omit PaSh bars for the full TERA-Seq suite because a meaningful PaSh comparison requires workflow-specific annotations and, for the dominant stages, domain-specific split and merge operators. To investigate this, we manually annotated a representative TERA-Seq workflow, 5TERA3, with conservative PaSh input/output annotations and ran a comparison. PaSh produced output matching Bash, but did not improve performance. Profiling showed that the safely annotatable FASTQ preprocessing work accounts for only 1.6% of profiled time, while SQLite/database mutation and SAM/BAM processing dominate; accelerating those stages would require database partition/merge support or SAM/BAM-aware splitters, not just command annotations. Each bar shows the speedup of hS and PaSh over bash. Across *all* benchmark sets, hS achieves a geomean speedup of 2.6× over Bash, (min: 0.14×, max: 9.3×). Across all benchmarks other than TERA-Seq, PaSh achieves a geomean speedup of 1.2× (min: 0.2×, max: 10.8×). hS slows down 7 scripts compared to bash (DGSH 3, 4, 6, 9, 10, NLP 9, WebIndex), out of which 5 take less than 10 seconds to execute with Bash (DGSH 3, 4, 9, 10, NLP 9). hS is slower than PaSh for only 4 scripts that take more than 10 seconds to execute (DGSH 5, 6,

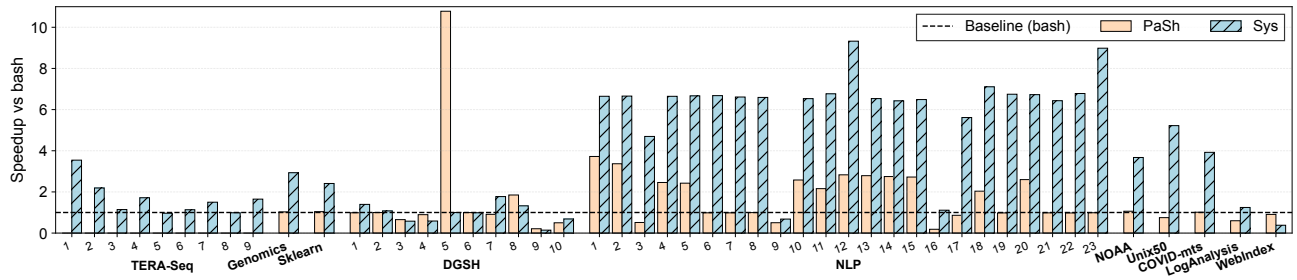


Figure 5: hS performance. hS and PaSh speedup over Bash for Tab. 1 rows 1–10.

8, and WebIndex). For all compared runs, we check that hS's output and exit status match Bash and, when included, PaSh.

Discussion: hS improves execution time on most scripts compared to Bash and PaSh, without any static knowledge about commands and without any input from the user. Importantly, hS manages to improve the performance of 7 out of 9 TERA-Seq scripts (geomean: 1.5×, max: 3.5×) even though their developers had already attempted to optimize them by uncovering parallelization and out-of-order execution opportunities using `&`. This shows that hS can be useful even for scripts that have already been optimized to exploit command independence. The scripts that hS slows down compared to Bash or PaSh either (1) have a short sequential running time below 10s and therefore hS's overheads dominate execution time (DGSH 3, 4, 9, 10, NLP 9), (2) perform many file accesses (WebIndex) (details in §7.4.2), or (3) exhibit strict dependencies across commands and pipelines (DGSH 5, 6, 8). For example, the DGSH 5 script performs a fully-dependent workflow, it finds words in a document (tokenizing, sorting, and removing duplicates) and then spellchecks them against a dictionary, leaving no room for out-of-order execution and so hS performs similarly to Bash (522s vs 523s, respectively). In contrast, PaSh manages to accelerate this script by leveraging its command annotations to exploit data parallelism in the same document, which is not in hS's scope. hS manages to accelerate scripts by exploiting available command invocation independence (across commands that read from or write to different files), for example in Genomics it manages to exploit the fact that different chromosomes can be processed independently. hS achieves that without any static knowledge about the commands in a script, and therefore can also accelerate scripts with domain-specific tools that PaSh cannot optimize.

7.3 Mis-speculation Statistics

We evaluate the amount of redundant resources that hS uses when it mis-speculates a command instance and then has to reexecute it. We measure the time spent on a command by recording the start and end time of each command instance (speculative) execution, then compute the time spent on mis-speculations over the total execution time of all commands.

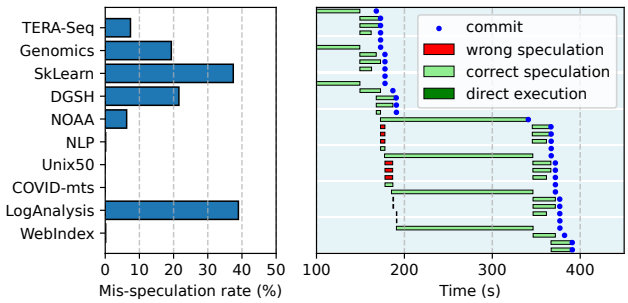


Figure 6: **Left:** Rate of mis-speculation. Percentage of how much execution time hS spends on reexecutions. **Right:** Fragment (100-450s) of Genomics execution in hS. Each row represents a command being executed, different blue regions are different iterations of a loop.

Note that the measured time is not equivalent to CPU time, since a command might have internal parallelization and potential resource contention with other commands.

Results (Fig. 6, left) and Discussion: Wasted execution time due to mis-speculation ranges from 0–39%, with an average of 17% across all scripts. Some scripts (Unix50, COVID-mts, NLP) have minimal dependencies and therefore observe minimal mis-speculation. Some scripts (TERA-Seq, NOAA, WebIndex) have dependencies but mis-speculations can be determined quickly due to eager dependency detection (§5.2), leading to a small amount of wasted execution time 0.1–8%. Finally, some scripts (Genomics, DGSH, LogAnalysis, Sklearn) have dependencies that can only be determined after spending significant time executing their commands, leading to more wasted execution time 19–39%. Sklearn and LogAnalysis both lead to significant mis-speculation because they introduces file-system writes at the end of some of their commands, preventing hS from detecting conflicts earlier.

Zooming in (Fig. 6, right): We show a trace from a Genomics execution with hS. After a quick setup, the script enters a loop, where iterations are independent, but for each iteration the first command is a dependency for all the rest. As the figure shows, subsequent commands in each iteration start

as mis-speculation at first, but are quickly terminated since hS can eagerly detect dependencies (§5.2) hS only reschedules these commands after the first command finishes. All in all, hS is able to speed up this script by 2.9× while only wasting 8% of its total execution time in mis-speculations.

7.4 Executor Overheads

We evaluate the executor's overheads using our benchmark suite and various microbenchmarks.

Constant overheads: We evaluate the constant overheads of hS's executor (*incl.* sandboxing, tracing, and parsing) by executing 1024 `echo` commands with and without sandboxing. On average hS's executor adds an overhead of 217ms per command, out of which 120ms is from sandboxing, a modest overhead considering that it focuses on optimizing programs with long executing commands.

7.4.1 hS overheads

Benchmark	Overhead
Genomics	5.5%
Sklearn	2.4%
DGSH	7.7%
NLP	18.1%
NOAA	49.0%
Unix50	31.8%
COVID-mts	3.0%
LogAnalysis	34.0%

We evaluate the overall overheads of hS's tracing, sandboxing, and orchestration by executing 8 benchmark sets (Tab. 1, rows 2–9) with speculation window 0, meaning that hS executes one command at a time.

Results and Discussion: Overheads of hS compared to Bash range from 2.4% to 49.0%, with an average of 23% across all scripts. Some benchmarks like Unix50, and LogAnalysis have high overheads because they execute a large number of short commands, which are more expensive to sandbox. Tracing and sandboxing introduce higher overheads for I/O intensive workloads (e.g., NOAA) and lower overheads for CPU-intensive ones (Sklearn and COVID-mts).

7.4.2 I/O-heavy overheads

We evaluate hS on two artificial, I/O-heavy workloads to stress test its overheads: one workload writes a large file using `dd` [36], and the other creates a large number of small files.

Results (Fig. 7): For a large output file, hS's performance is 3.2–3.4× slower than Bash, most of which comes from sandboxing (3.0–3.4×), while tracing and parsing add negligible overhead. For many small files, hS's performance is 6.2–7.2× slower than Bash, with tracing causes 310% overhead, while parsing and sandboxing incur smaller part of the overhead 120–130% and 140–200%, respectively. hS's running time is slightly faster than the sandboxing time because tracing and parsing are pipelined.

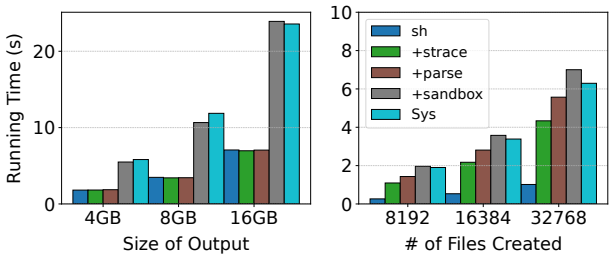


Figure 7: hS overheads on two I/O-heavy workloads.

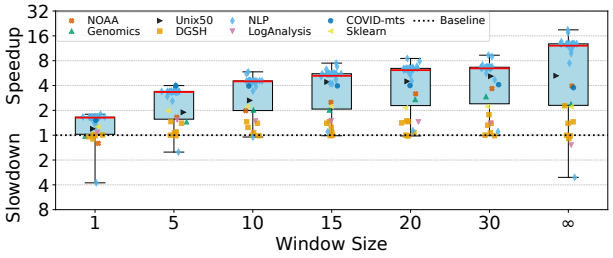


Figure 8: hS performance with varying window sizes.

Discussion: hS can incur substantial overheads on I/O-heavy workloads, both with the increasing size of written output, as well as the increasing number of accessed files. These are due to (1) the additional I/O overhead of moving the files out of the sandbox (between drives), and (2) the overhead of tracing a large number of `open` system calls.

7.5 Varying Window Sizes

We finally evaluate how different speculation window sizes, i.e., how many commands can be speculated concurrently, influence hS's performance. We run the scripts that take longer than 20 seconds out of 8 benchmark sets (Tab. 1, rows 2–9). We vary the speculation window size from 1 to an infinite window size (speculate all subsequent commands).

Results (Fig. 8): Increasing the window size from 1 to 15 improves performance across all sets, from 1.6× to 5.24× on average. Beyond this, benchmarks exhibit distinct behaviors: NLP continues scaling, reaching up to 9.1× on average at infinite window size; LogAnalysis peaks and then degrades due to excessive speculation (33% at infinite window size); and the rest initially benefit but quickly plateau.

Discussion: Optimal window size depends on script complexity, parallelizability, and available CPU cores. Several of our benchmarks benefit from larger windows due to abundant parallelism, *e.g.*, NLP scripts with completely independent loop iterations. The benchmarks that plateau execute fewer commands or are inherently less parallelizable. Finally, some scripts with many short-running commands do not benefit

Table 2: Summary of all Python benchmarks used to evaluate hS and their characteristics. Exec. Time indicates the execution time of the script with standard Python and Speedup indicates the speedup of hS.

	Benchmark	LOC	Input	Python Time	hS Speedup
1	BioAlign [59]	27	1.6M	3h11m	1.75×
2	ProteinInt [26]	122	3.5G	5h7m	2.25×
3	AudioProc [32]	22	28G	4m23s	3.99×
4	VideoProc [67]	32	648G	14m50s	2.76×
5	MRIanalysis [53]	161	25G	2h44m	8.40×

from larger window sizes due to resource saturation, *e.g.*, LogAnalysis invokes more than 700 commands, saturating the 128 threads of the execution machine.

7.6 Performance on Python programs

We evaluate the performance of hS and its Python frontend by comparing its execution time with standard Python execution on a set of real-world Python scripts. We configure hS with a speculation window of 16.

Benchmarks: We use a set of 5 real-world Python programs to evaluate hS's Python frontend (see Tab. 2 for a summary). The programs perform a variety of tasks, two bioinformatics workflows, an MRI image analysis, and audio and video processing; and are taken from a variety of sources including a textbook, an online forum, and a serverless benchmark suite. We did minor modifications to the programs to simplify the frontend: (1) invocations to external processes through a wrapper were inlined to directly call `subprocess.run`; (2) file-manipulation functions, e.g., `sys.write` were replaced with external processes calls so hS's executor could track their side-effects; and (3) user defined functions were inlined in main. BioAlign was also modified to disable JVM's performance logging to enable parallelization (see "Discussion").

Results (Tab. 2): We show the performance of hS compared to standard Python execution. Scripts have varying execution times, ranging from 4m to 5h when executed without hS. hS achieves a geomean speedup of 3.26× (min: 1.75×, max: 8.40×) across the benchmarks.

Discussion: hS improves the execution time across all Python scripts against baseline Python without any user input. For example, in MRIanalysis, hS manages to identify the speculation opportunity across independent loop iterations and achieves a speedup of 8.4×. In ProteinInt hS only achieves a 2.25× speedup because the script is bottlenecked by indexing a reference genome before the parallelizable loop.

Furthermore, hS's dependency tracking enables safe optimizations and performance debugging for target applications. When we first ran BioAlign with hS, we observed a .5% slowdown compared to the baseline. Inspecting hS's dependency logs allowed us to identify and fix a hidden but unnecessary write-write conflict (all JVM instances write perf. monitoring data in `/tmp/hsperfdata`) by passing a JVM flag that disables these logs. This experience shows that hS's tracking facilitates performance debugging and achieves benefits without risking changing a program's behavior—which could happen if one manually reordered invocations that were mistakenly thought to be independent.

8 Related Work

Performance optimizations for shell scripts: Recent work achieves significant speedups on shell scripts through parallelization [20, 29, 55, 62] and distribution [42, 50]. A key component of all these systems is a just-in-time compilation infrastructure [19, 29] that optimizes scripts at the right time, despite them being highly dynamic. hS builds on this just-in-time infrastructure, but extends it with context-passing, side-effect-management, and dependency reasoning to enable speculative execution. All of these systems require annotations for all commands present in a pipeline to enable optimization; hS, on the other hand, can optimize scripts without any user input, by understanding command behavior at runtime through tracing and isolation.

Runtime frameworks for shell scripts: Recent work that improves various aspects of shell scripts has identified the benefit of harnessing runtime execution information through tracing. Riker [11] enables incremental execution for shell scripts that describe software builds by tracing their execution and using the trace to identify the minimal set of commands to re-execute when the script changes. Dozer [22] translates shell scripts to Ansible commands for better maintainability by tracing the system calls of a target script and finding comparable traces in its library of Ansible commands. Autobash [58] helps users debug system configuration errors by tracing command execution to infer state deltas, with which it is able to detect predicate violations and perform rollbacks [44].

Explicit dependency encoding: Workflow and build systems [15, 21, 25, 31, 56] explicitly express dependency graphs by manually encoding all input and output dependencies of each step. Encoding dependencies statically and ahead-of-time yields better program schedules, but (1) requires users to provide *all* dependencies or suffer from stale or incorrect results, and (2) cannot express the high dynamism prevalent in shell scripts. Our approach addresses both these challenges.

Speculative execution: Speculation and rollback are not new ideas, with an extensive history from computer architecture to the application level, *e.g.*, for system configuration [44] or security checks [45]. In some of these proposals speculation is enabled by modifying the application (*e.g.*, Undo [6]), while others support arbitrary black-box applications [44, 45, 58]. Thread-level speculation is a widely

studied technique for extracting parallelism from applications at runtime by speculatively executing parts of them in different threads and rolling them back if dependencies are violated [14, 60]. Spidermine [64] optimizes programs through speculative prefetching by dynamically identifying sites of frequent reads. We build on these ideas, tightly coupling with the shell's language and semantics reducing overhead and the amount of tracked information: instead of considering the whole script as a black-box application, we use the script semantics to separate it into logical application components (command invocations).

Containment and tracing: There has been significant work on extending filesystems with sandboxing and transactions, some examples include: MBOX [30], a sandboxing mechanism; Amino [65], a transactional filesystem; SandFS [3], a filesystem with sandboxing support; filesystems that support snapshots, such as ZFS [5] and BTRFS [51]; and TotalCOW [66], which aims to improve the treatment of copy-on-write pages in Linux. All of this work is orthogonal and complementary to hS, since it could be used instead of or in addition to OverlayFS to improve hS's performance.

Shell script semantics: Finally, recent work has tried to study the semantics of shell scripts, either through collection of real-world shell uses [12], or through formalization [16, 17]. hS draws inspiration from these works to reason about the correctness of its optimizations and uses the parsing infrastructure developed by Smoosh [10].

9 Discussion

Limitations: There are some effects that hS's executor cannot currently identify, block, or revert precisely. First, our implementation blocks, but does not precisely detect, IPC, signals, and effects through device files [43]. These effects are restricted by IPC, pid, and mount namespaces respectively, but precisely classifying them requires more elaborate system-call tracing and, for device-file accesses, comparison with mount structures. Second, hS cannot detect, block, or revert effects through memory-mapped I/O or file systems with external effects, such as NFS [52]. Third, hS does not currently support pseudo-files for GPU devices. Although GPU runtimes typically provide process-level memory isolation, they rely on `ioctl` calls that create persistent state and perform IPC between GPU-using processes, which is not straightforward to detect and block [47]. hS also does not support scripts whose core logic depends on time, including wall-clock time or file access times [39]; executing such scripts with hS may change their results.

hS's dependency abstraction is path-based. It handles parent-directory effects and symbolic-link use conservatively, but it does not canonicalize all path aliases, such as hard links, complex bind mounts, or `/proc/self/fd`-style paths, into a single inode-level identity. Scripts whose correctness depends on such aliases being recognized as the same underlying file are therefore outside hS's current guarantees.

A separate limitation concerns scripts that impose higher-order semantic requirements on file accesses, *e.g.*, `apt`'s use of files as locks [37], or programs that coordinate through shared on-disk caches such as compiler caches, package-manager state, SQLite databases, or generated bytecode caches. These cases can require application-specific merge or locking semantics that are not captured by bulk OverlayFS commit.

hS also does not provide opacity for speculative executions: a speculated command may observe a state that is not globally reachable in the sequential execution, and could therefore crash, loop, or generate excessive output before being discarded. Production deployments could add explicit time, memory, and output quotas to bound such behavior.

10 Conclusion

This paper proposes hS, a system that can automatically exploit latent parallelism in shell scripts using out-of-order speculative execution. hS represents a significant improvement over the state of the art because it can support and optimize scripts with arbitrary executables without any form of user input or command annotations.

Acknowledgments

We are thankful to the anonymous OSDI'26 and AEC reviewers; the Brown, UCLA, and Columbia systems groups; and the many PaSh workshop participants. This material is based upon research supported by NSF awards CNS-2247687, CNS-2312346, and CNS-2312347; DARPA contracts no. HR001120C0191 and HR001124C0486; an Amazon Research Award (Fall 2024); a Google ML-and-Systems Junior Faculty Award; a seed grant from Brown University's Data Science Institute; and a Brown CS Faculty Innovation Award.

References

[1] mergerfs: a featureful union filesystem. https://github.com/trapexit/mergerfs.

[2] Martin Arlitt, Tai Jin, Greg Oster, and Balachander Krishnamurthy. 1998 world cup web site access logs, 1998. Available at https://ita.ee.lbl.gov/html/contrib/WorldCup.html.

[3] Ashish Bijlani and Umakishore Ramachandran. A lightweight and fine-grained file system sandboxing framework. In *Proceedings of the 9th Asia-Pacific Workshop on Systems*, APSys '18, New York, NY, USA, 2018. Association for Computing Machinery.

[4] binpash. Scikit-learn benchmark. https://github.com/binpash/benchmarks/tree/main/sklearn, 2024.

[5] Jeff Bonwick, Matt Ahrens, Val Henson, Mark Maybee, and Mark Shellenbaum. The zettabyte file system. In *Proc. of the 2nd Usenix Conference on File and Storage Technologies*, volume 215, 2003.

[6] Aaron B Brown and David A Patterson. Undo for operators: Building an undoable e-mail store. In *USENIX Annual Technical Conference, General Track*, pages 1–14, 2003.

[7] Neil Brown, Miklos Szeredi, Amir Goldstein, Vivek Goyal, Randy Dunlap, Linus Torvalds, Pavel Tikhomirov, Kevin Locke, Sargun Dhillon, Chengguang Xu, and Deming Wang. The overlay filesystem. *The Linux Kernel documentation*, 2022. Started in 2014.

[8] Enrico Cappellini, Frido Welker, Luca Pandolfi, Jazmín Ramos-Madrigal, Diana Samodova, Patrick L Rüther, Anna K Fotakis, David Lyon, J Víctor Moreno-Mayar, Maia Bukhsianidze, et al. Early pleistocene enamel proteome from dmanisi resolves stephanorhinus phylogeny. *Nature*, 574(7776):103–107, 2019.

[9] Kenneth Ward Church. Unix™ for poets, 1994. Notes of a course from the European Summer School on Language and Speech Communication, Corpus Based Methods.

[10] Github Contributors. libdash, 2024. Accessed 2024-04-19.

[11] Charlie Curtsinger and Daniel W Barowy. Riker: Always-Correct and fast incremental builds from simple specifications. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, pages 885–898, 2022.

[12] Yiwen Dong, Zheyang Li, Yongqiang Tian, Chengnian Sun, Michael W. Godfrey, and Meiyappan Nagappan. Bash in the wild: Language usage, code smells, and bugs. *ACM Transactions on Software Engineering and Methodology*, 32(1):1–22, January 2023.

[13] Dmitry Duplyakin, Robert Ricci, Aleksander Maricq, Gary Wong, Jonathon Duerig, Eric Eide, Leigh Stoller, Mike Hibler, David Johnson, Kirk Webb, Aditya Akella, Kuangching Wang, Glenn Ricart, Larry Landweber, Chip Elliott, Michael Zink, Emmanuel Cecchet, Snigdhaswin Kar, and Prabodh Mishra. The design and operation of CloudLab. In *Proceedings of the USENIX Annual Technical Conference (ATC)*, pages 1–14, July 2019.

[14] Alvaro Estebanez, Diego R Llanos, and Arturo Gonzalez-Escribano. A survey on thread-level speculation techniques. *ACM Computing Surveys (CSUR)*, 49(2):1–39, 2016.

[15] Sadjad Fouladi, Francisco Romero, Dan Iter, Qian Li, Shuvo Chatterjee, Christos Kozyrakis, Matei Zaharia, and Keith Winstein. From laptop to lambda: Outsourcing everyday jobs to thousands of transient functional containers. In *USENIX Annual Technical Conference*, pages 475–488, 2019.

[16] Michael Greenberg. The POSIX shell is an interactive DSL for concurrency, 2018. DSLDI.

[17] Michael Greenberg and Austin J. Blatt. Executable formal semantics for the POSIX shell. *Proc. ACM Program. Lang.*, 4(POPL):43:1–43:30, 2020.

[18] Michael Greenberg, Konstantinos Kallas, and Nikos Vasilakis. The future of the shell: Unix and beyond. In *Proceedings of the Workshop on Hot Topics in Operating Systems*, HotOS '21, pages 240–241, New York, NY, USA, 2021. Association for Computing Machinery.

[19] Michael Greenberg, Konstantinos Kallas, and Nikos Vasilakis. Unix shell programming: The next 50 years. In *Proceedings of the Workshop on Hot Topics in Operating Systems*, HotOS '21, pages 104–111, New York, NY, USA, 2021. Association for Computing Machinery.

[20] Shivam Handa, Konstantinos Kallas, Nikos Vasilakis, and Martin Rinard. An order-aware dataflow model for extracting shell script parallelism. *Proc. ACM Program. Lang.*, 4(ICFP), August 2021.

[21] Bas P Harenslak and Julian de Ruiter. *Data pipelines with Apache Airflow*. Simon and Schuster, 2021.

[22] Eric Horton and Chris Parnin. Dozer: migrating shell commands to ansible modules via execution profiling and synthesis. In *Proceedings of the 44th International Conference on Software Engineering: Software Engineering in Practice*, ICSE-SEIP '22, page 147–148, New York, NY, USA, 2022. Association for Computing Machinery.

[23] Fadia Ibrahim, Jan Oppelt, Manolis Maragkakis, and Zissimos Mourelatos. TERA-Seq: true end-to-end sequencing of native RNA molecules for transcriptome characterization. *Nucleic Acids Research*, 49(20):e115–e115, 08 2021.

[24] Github Inc. The top programming languages. https://octoverse.github.com/2022/top-programming-languages, 2022.

[25] Google Inc. Bazel, 2015.

[26] Hamid D. Ismail. *Bioinformatics of Autoimmune Diseases*. CRC Press, 1st edition, 2026. Under publishing.

[27] Daniel Jackson. Alloy: a lightweight object modelling notation. *ACM Transactions on software engineering and methodology (TOSEM)*, 11(2):256–290, 2002.

[28] Jonathan Corbet. Unprivileged filesystem mounts, 2018 edition. https://lwn.net/Articles/755593/, 2018.

[29] Konstantinos Kallas, Tammam Mustafa, Jan Bielak, Dimitris Karnikis, Thurston H.Y. Dang, Michael Greenberg, and Nikos Vasilakis. Practically correct, just-in-time shell script parallelization. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 1–18. USENIX Association, July 2022.

[30] Taesoo Kim and Nickolai Zeldovich. Practical and effective sandboxing for non-root users. In *2013 USENIX Annual Technical Conference (USENIX ATC 13)*, pages 139–144, San Jose, CA, June 2013. USENIX Association.

[31] Johannes Köster and Sven Rahmann. Snakemake—a scalable bioinformatics workflow engine. *Bioinformatics*, 28(19):2520–2522, 2012.

[32] Oleksii Kuchaiev, Somshubra Majumdar, Eric Harper, et al. Nemo: a toolkit for conversational ai and large language models, 2024.

[33] Evangelos Lamprou, Ethan Williams, Georgios Kaoukis, Zhuoxuan Zhang, Michael Greenberg, Konstantinos Kallas, Lukas Lazarek, and Nikos Vasilakis. The koala benchmarks for the shell: Characterization and implications. In *2025 USENIX Annual Technical Conference (USENIX ATC 25)*, pages 449–464, Boston, MA, July 2025. USENIX Association.

[34] John MacFarlane. Pandoc: a universal document converter. https://pandoc.org/. Accessed: 2025-04-15.

[35] Aurèle Mahéo, Pierre Sutra, and Tristan Tarrant. The serverless shell. In *Proceedings of the 22nd International Middleware Conference: Industrial Track*, pages 9–15, 2021.

[36] Linux man-pages project. dd(1) – linux manual page.

[37] Linux man-pages project. lockf(2) — linux manual page.

[38] Linux man-pages project. namespaces(7) – linux manual page.

[39] Linux man-pages project. stat(3type) — linux manual page.

[40] Linux man-pages project. strace(1) – linux manual page.

[41] Karel Zak Mikhail Gusarov. unshare - run program in new namespaces. *Linux manual page*, 2022.

[42] Tammam Mustafa, Konstantinos Kallas, Pratyush Das, and Nikos Vasilakis. DiSh: Dynamic Shell-Script distribution. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 341–356, Boston, MA, April 2023. USENIX Association.

[43] Binh Nguyen. Linux Filesystem Hierarchy: /dev. https://tldp.org/LDP/Linux-Filesystem-Hierarchy/html/dev.html. Accessed: 2025-04-15.

[44] Edmund B Nightingale, Peter M Chen, and Jason Flinn. Speculative execution in a distributed file system. *ACM SIGOPS operating systems review*, 39(5):191–205, 2005.

[45] Edmund B Nightingale, Daniel Peek, Peter M Chen, and Jason Flinn. Parallelizing security checks on commodity hardware. *ACM SIGARCH Computer Architecture News*, 36(1):308–318, 2008.

[46] Nokia Bell Labs. The unix game—solve puzzles using unix pipes, 2019. Accessed: 2020-03-05.

[47] Nvidia. CUDA C++ Programming Guide: Interprocess Communication. Accessed 2025-04-15.

[48] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[49] Nikos Perdikopanis. Bioinformatics analysis of next generation sequencing (ngs) data with r, 2023. Accessed: 2024-04-19.

[50] Deepti Raghavan, Sadjad Fouladi, Philip Levis, and Matei Zaharia. POSH: A data-aware shell. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pages 617–631, 2020.

[51] Ohad Rodeh, Josef Bacik, and Chris Mason. BTRFS: The linux B-tree filesystem. *ACM Transactions on Storage (TOS)*, 9(3):1–32, 2013.

[52] Russel Sandberg. The sun network file system: Design, implementation and experience. In *in Proceedings of the Summer 1986 USENIX Technical Conference and Exhibition*, 1986.

[53] Derek Schettler. Captk brats preprocessing - [cleaned & commented], 2021. Kaggle Notebook.

[54] Jiasi Shen, Martin Rinard, and Nikos Vasilakis. Automatic synthesis of parallel unix commands and pipelines with kumquat. In *Proceedings of the 27th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '22, pages 431–432, New York, NY, USA, 2022. Association for Computing Machinery.

[55] Diomidis Spinellis and Marios Fragkoulis. Extending unix pipelines to dags. *IEEE Transactions on Computers*, 66(9):1547–1561, 2017.

[56] Richard M Stallman, Roland McGrath, and Paul Smith. Gnu make. *Free Software Foundation, Boston*, 1988.

[57] Salvatore Stolfo, Wei Fan, Wenke Lee, Andreas Prodromidis, and Philip Chan. KDD Cup 1999 Data. UCI Machine Learning Repository, 1999. DOI: https://doi.org/10.24432/C51C7N.

[58] Ya-Yunn Su, Mona Attariyan, and Jason Flinn. Autobash: Improving configuration management with operating system causality analysis. *ACM SIGOPS Operating Systems Review*, 41(6):237–250, 2007.

[59] ta2007. Multiprocessing in python with subprocess to align several files with coding dna sequences, 2016. Biostars Forum Post.

[60] Josep Torrellas. Speculation, thread-level. In David Padua, editor, *Encyclopedia of Parallel Computing*, pages 1894–1900. Springer US, Boston, MA, 2011.

[61] Eleftheria Tsaliki and Diomidis Spinellis. The real statistics of buses in athens. https://bit.ly/3s112R5, 2021.

[62] Nikos Vasilakis, Konstantinos Kallas, Konstantinos Mamouras, Achilles Benetopoulos, and Lazar Cvetković. Pash: Light-touch data-parallel shell processing. In *Proceedings of the Sixteenth European Conference on Computer Systems*, pages 49–66, New York, NY, USA, 2021. Association for Computing Machinery.

[63] Tom White. *Hadoop: The Definitive Guide*. O'Reilly Media, Inc., 4th edition, 2015.

[64] Jiwoong Won, Sangwoon Yun, Ahn Jemin, Jong-Chan Kim, and Kyungtae Kang. Spidermine: Low overhead user-level prefetching. In *Proceedings of the 38th ACM/SIGAPP Symposium on Applied Computing*, SAC '23, page 1332–1341, New York, NY, USA, 2023. Association for Computing Machinery.

[65] Charles P. Wright, Richard Spillane, Gopalan Sivathanu, and Erez Zadok. Extending acid semantics to the file system. *ACM Trans. Storage*, 3(2):4–es, jun 2007.

[66] Xingbo Wu, Wenguang Wang, and Song Jiang. Totalcow: Unleash the power of copy-on-write for thin-provisioned containers. In *Proceedings of the 6th Asia-Pacific Workshop on Systems*, APSys '15, New York, NY, USA, 2015. Association for Computing Machinery.

[67] Tianyi Yu, Qingyuan Liu, Dong Du, Yubin Xia, Binyu Zang, Ziqian Lu, Pingchao Yang, Chenggang Qin, and Haibo Chen. Characterizing serverless platforms with serverlessbench. In *Proceedings of the 11th ACM Symposium on Cloud Computing*, SoCC '20, page 30–44, New York, NY, USA, 2020. Association for Computing Machinery.

A Artifact Appendix

Abstract

The artifact for hS is an archived release of the implementation and evaluation materials for speculative shell execution. It includes the code, artifact-evaluation instructions, automation scripts, precomputed paper results, and dataset manifest.

Scope

The OSDI '26 evaluation grants only the Artifacts Available badge. The artifact therefore first supports checking that the implementation and supporting materials are public and permanently archived. The same instructions also describe optional functional and reproducibility paths: installing hS, running tests, regenerating plots from precomputed results, and rerunning selected benchmarks on suitable hardware.

Contents

The archived release of hS contains the source tree (`hs`, `scheduler/`, `executor/`, `preprocessor/`, `jit_runtime/`, `model-checking/`, `test/`, `report/`, and `python_hs/`); the `try` submodule; setup/archive helper scripts; precomputed results and plotting inputs under `artifact/paper-results/`; and dataset locations in `artifact/datasets.md`, as well as detailed instructions to set up and use the artifact.

Hosting

The stable artifact URL is the Zenodo archive: https://doi.org/10.5281/zenodo.19832407. The corresponding GitHub artifact-evaluation branch is https://github.com/binpash/hs/tree/osdi26-ae. The main development repository is https://github.com/binpash/hs.

Requirements

The quickstart in `INSTRUCTIONS.md` targets an Ubuntu-like Linux host (Ubuntu 20.04 or newer preferred), Python 3.12 or newer, permission to install APT packages, and `sudo`/root access for the tracing, cgroup, OverlayFS, and filesystem facilities used by hS and `try`. The setup script initializes dependencies and builds helper binaries. Full benchmark reruns require multicore machines with at least 1TB of disk space, and optionally privileged Docker access.

ARTIFACT EVALUATED
USENIX
AVAILABLE

Incr: Faster Re-execution via Bolt-on Incrementalization

Yizheng Xie* Evangelos Lamprou* Jerry Xia* Nikos Vasilakis
Brown University

Abstract

While most software development is incremental, most execution environments are not: even small program modifications fail to take advantage of prior executions, at worst requiring full re-execution of all computational stages in the modified program. Such full re-execution decelerates software development and debugging, especially in dynamic polyglot environments such as the Unix and Linux shell. This paper presents INCR, a system that accelerates the re-execution of unmodified shell programs by automatically incrementalizing their execution. INCR analyzes and tracks interdependencies to detect and store key intermediate results, reusing them on subsequent re-executions whenever possible. INCR's effect analysis supports correct re-execution even for non-idempotent computations, and several static and dynamic optimizations reduce the runtime and storage overheads of incrementalization. Applied to diverse real-world scenarios, INCR accelerates re-execution by an average of 34.2× and a maximum of 373.3×—all while requiring no developer annotations or code modifications and remaining behaviorally indistinguishable on over 10,000 test cases.

1 Introduction

Nearly all software development is incremental: layers of modifications, additions, replacements, and deletions are applied iteratively to morph a program toward its intended goal. Such incremental development is particularly common today in data science [41, 60, 83], machine learning [10, 30], exploratory computing [24, 66, 79], and interactions with large language models [28, 67]. It is also the standard approach in dynamic, interactive environments such as the Unix shell, which is used for understanding, exploring, and gradually refining software systems, the opaque components comprising them, and the input they operate on [31, 43, 77].

Unfortunately, these environments do not support efficient re-computation when parts of a program change. Such changes, irrespective of their size, fail to take advantage of past executions—requiring a full re-execution of all the computational stages in the modified program. Even small refinements can result in long waiting cycles, especially on large datasets where re-execution might take the majority of the time. As the program grows, the longer it runs—and the slower it becomes to further iterate, discover errors, hand-tune, debug, and eventually ship.

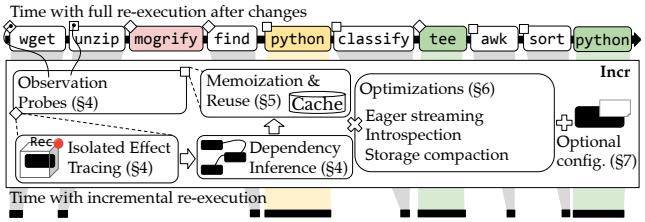


Fig. 1: **INCR overview**. INCR accelerates re-execution in modified shell programs by bolting onto otherwise unmodified execution environments. Upon re-execution, it reuses prior results to avoid redundant computation. Colors indicate types of incrementalization: + additions, − deletions, and ~ in-place modifications.

This paper introduces INCR, a system that accelerates re-execution by bolting incremental re-computation onto unmodified shell programs. Once enabled during development, it trades a small overhead during the first execution to accelerate subsequent re-executions. INCR operates in two phases: the analysis phase, which collects and analyzes program dependencies; and the incrementalization phase, which extracts information about modifications and, combined with the earlier analysis, accelerates subsequent re-executions of modified programs. INCR supports full POSIX *and* Bash semantics, including modifications to command arguments, flags, data flow, control flow, environment variables, and external resources, ensuring behavioral equivalence even for non-idempotent operations. When ready to ship, INCR can be disabled to avoid unnecessary runtime overheads in production environments.

INCR's analysis and incrementalization phases discover and

*Equal contribution

exploit process-level data and control dependencies across commands, automatically detecting, analyzing, and storing effects at runtime (§4). In subsequent runs, INCR determines which parts of a shell program are unchanged and safely reuses their outputs (§5). Several additional optimizations, such as eager stream processing, introspection, and storage compaction, lower runtime and space overheads (§6), making incrementalization practical for real-world workloads (§8). While no user input is necessary, INCR can leverage crowdsourced partial annotations for POSIX, GNU Coreutils, and third-party commands from earlier research [42, 71, 84], as well as developer-specified configurations to enable finer-grained incrementalization or eliminate dependency analysis in parts of programs that will not change (§7).

INCR has been applied to 14 real-world scenarios that include debugging, data exploration, interactions with LLMs, and other common tasks (§8). With no developer input, INCR accelerates re-executions by an average of 34.2× and by up to 373.3× by trading in space (avg: 6.05× of the original input size, max: 55.44×). While INCR slows down the first (typically short) execution by an average of 101.05%, mostly due to dependency tracking, additional optimizations reduce this overhead to 43.55%, raising average re-execution speedup to 38.64× and the maximum to 377.65×. INCR's behavior and outputs are indistinguishable from the underlying shell interpreter across all real-world scenarios and 10,279 out of 10,282 (99.9%) tests from the standard Bash test suite, including unusual behaviors rarely seen in practice.

The paper starts by exemplifying INCR on a real-world workload (§2), followed by its execution model (§3) and key contributions (§4–7):

- Fine-grained dependency tracking via lightweight interposition probes that capture interactions across the filesystem and shell environment (§4).
- Correct incrementalization via memoization of dependencies and effects, including both transient data streams and side effects, and safe reuse of prior effects (§5).
- Diverse runtime optimizations, such as eager stream processing, introspection, and compaction, that make incremental execution practical (§6).
- An optional tuning interface that accepts crowdsourced annotations and developer configurations to enhance, disable, or relax parts of incrementalization (§7).

The paper then characterizes INCR on real-world programs (§8), discusses related work (§9), and concludes (§10). Appendix A presents the paper's software artifact.

2 Applying INCR During Development

Fig. 2 shows a real shell program used to digitize images of hieroglyphs collected during an archaeological expedition [3]. The script (1) fetches the images collected during the expedition; (2) unpacks the images into a local directory; (3) iterates over the images; (4) segments each image into hieroglyph regions using Meta's Segment Anything Model [45]; (5) applies a hieroglyph classifier to each segment; and (6) outputs the formatted mappings between image filenames and their classified segments to the `db.txt` file.

```
1  wget 'http://inst.edu/dpt/images.zip'
2  IMGS=${IMGS:-images}
3  unzip images.zip -d "$IMGS"
4  mogrify -resize 1024x1024\> "$IMGS"/*     # -(1)
5  for img in "$IMGS"/*; do
5′ for img in $(find "$IMGS" -type f); do   # ~(3)
6  python segment.py "$img" |
6′ python segment.py "$img" -s 1024x1024 |   # ~(1)
7  while read -r mask; do
8  python classify.py -i "$img" -r "$mask" |
9  done |
10 tee -a classes.txt |                      # +(4)
11 awk '{print "g:", $5}'
11′ awk -vi="$img" '{print "g:", $5, $6, i}' # ~(2)
12 done | sort > db.txt
13 python plot.py classes.txt                # +(4)
14 rm classes.txt                            # +(4)
```

Fig. 2: **Hieroglyph classification script**. A script that segments, classifies, and visualizes images of hieroglyphs. Highlighted lines indicate modifications made during development: `+` additions, `-` deletions, and `~` in-place modifications. Primed line numbers denote updated versions of earlier lines. Numbers denote modification order; equal numbers belong in the same iteration.

Developing this script involved several modifications, four of which are highlighted in Fig. 2: removing a redundant image resizing step (ln.4, ln.6, #1), changing `awk` to include image paths in the output (ln.11, #2), making the outer loop process images in all subfolders (ln.5, #3), and plotting the classification results (ln.10, ln.13, #4).

Problem: Re-executing the entire script after each modification takes up to 15 minutes, dominated by the segmentation and classification stages.

Incrementalization—re-executing only the parts of the script necessary to reflect each modification—can significantly accelerate re-execution. To achieve incrementalization, a developer could *manually* insert `tee` commands and **`if`** guards (lns. 5–12) to memoize intermediate results and reorganize the script to reuse those intermediate files. However, this approach requires significant effort and can introduce subtle errors, as manual partial re-execution may use stale intermediate results. Furthermore, this approach does not help with change #3, which expands the input image set, and is cumbersome to apply with changes #1 and #4, which

modify the program in non-trivial ways.

Key challenges: There are several challenges in automating the incrementalization of shell programs. First, Fig. 2's script contains opaque, heterogeneous commands with complex and often implicit interdependencies. For example, `plot.py`, introduced in modification #4, consumes `classes.txt` (produced in ln. 10) and thus depends on the outputs of all prior stages (lns. 5–9). Commands also interact with external resources and the execution environment. For instance, `segment.py` (ln. 6) reads images from the `$IMGS` directory (defined in ln. 2) and depends on several Python packages loaded by the `python` interpreter. In addition, such dependencies are typically implicit and dynamic (*e.g.*, `$IMGS` may be externally set). Therefore, the script's behavior can change with its surrounding context.

Second, these commands generate arbitrary side effects, including modifications to the filesystem and transient input/output streams that flow through pipelines and cannot be easily reused across runs. For example, modification #4 alters the pipeline by inserting a `tee` command (ln. 10), which consumes the output stream produced by earlier commands. The `tee` command then emits two effects: one written to disk as `classes.txt`—later consumed by the newly added `plot.py` command (ln. 13)—and another directly streamed to the next command `awk` (ln. 11).

Finally, commands exhibit diverse execution patterns that can make incrementalization via tracing and caching inefficient. For instance, modification #3 changes the set of images the **`for`** loop will iterate over; this changes the set of images processed but does not invalidate memoized results for some commands inside the loop, as they can be reused for the unchanged images. Memoization of command effects can incur substantial storage overhead, particularly for large artifacts produced by commands such as `unzip` or `segment.py`.

Applying INCR: Developers enable INCR during development to automatically incrementalize scripts and disable it when ready to deploy. This workflow for Fig. 2's script is:

```
$ incr dpt.sh                    # Enabling INCR
$ incr dpt.sh ... incr dpt.sh    # #1 #2 #3 #4
$ ./dpt.sh                       # Ready to ship!
```

To solve the earlier challenges, INCR probes the script's execution at runtime to track command dependencies (§4), memoize their intermediate effects (§5), and reuse them when relevant state is unchanged. It first parses the script, inserts *interposition probes* at all command invocations, and executes the transformed script using the underlying shell interpreter. These probes wrap each command and observe its execution: each probe creates an isolated environment for its command to run in, tracks its `stdin` and filesystem dependencies, and memoizes its input/output streams and filesystem effects. During re-execution, each probe compares the current dependencies to those stored and decides locally to re-execute its command if it detects any changes; otherwise, it skips execution and emits its memoized effects. As INCR's probes operate at runtime, they can observe dynamic changes to variables (which may depend on external state) and control flow.

Modification #1 removes the redundant `mogrify` command that resized images before segmentation and makes the `segment.py` command resize images internally instead. Upon re-execution, INCR replays `wget`'s and `unzip`'s effects—their dependencies are unchanged and INCR is not invoked with `-N`, which would re-execute all network-effectful components. It then detects that `segment.py`'s input and invocation have changed; INCR thus re-executes `segment.py` normally, but its output, containing normalized coordinates, has not changed. Then, the probes surrounding `classify.py` and `awk` detect that their environment, inputs/outputs, and filesystem dependencies are unchanged, so INCR skips their executions and emits their memoized results.

Modification #2 includes two changes to the `awk` command: adding classification confidence (`$6`) and the image path (`i`) to its output. The `awk` command's probe detects that its arguments have changed and re-executes it. The downstream probes detect `awk`'s output changes and re-execute their corresponding commands. All prior stages (lns. 5–9) reuse their memoized results.

Modification #3 filters out irregular files in the `$IMGS` directory using `find`. The probe on `segment.py` retrieves and emits memoized results for images it has processed before; otherwise, it executes the command normally.

Finally, modification #4 includes several changes that add a plotting section to visualize the classification results saved by `tee`. On re-execution, INCR skips all unchanged stages before `tee`. Then, INCR detects that `tee` does not alter the input to the `awk` command, and thus continues to reuse memoized results for all subsequent stages—executing only the new `plot.py` command.

Results: INCR improves time spent across all re-executions from 1h25m to 20m41s, resulting in a 4.1× speedup. Modifications #3 and #4, which include exploration of already-processed data, enjoy 91.2× and 119× speedups, respectively. If annotations were used, INCR would achieve an additional 5% speedup by skipping dependency tracking on `awk` (§7).

3 Execution Model

To provide practical incrementalization for real shell programs, INCR does not incrementalize all observable behavior nor does it re-execute all commands when their environment exhibits *any* observable change. Instead, it groups all possible component effects and behaviors into five classes (Tab. 1): (1) *memoizable* effects, which INCR can record and apply identically to the component that originally performed them; (2) *detectable* effects, which it observes to infer dependencies but which it does not memoize; (3) *blocked* effects, which will fail during a command's execution and thus typically surface as errors indicating to the user that INCR cannot be

used for part of the script; (4) *ignored* effects, which INCR's current implementation does not support; and (5) *out-of-scope* behaviors that are fundamentally outside INCR's goals and assumptions.

Memoizable effects are effects which INCR can both detect precisely during tracing and replay exactly during reuse. These are a command's interactions with standard streams, its exit status, and local filesystem effects on regular files, directories, symbolic links, hard links, named pipes when used unidirectionally, including their permission metadata changes. Examples of such effects are writing to `stdout`, moving a directory from one path to another, and creating a symbolic link. Commands whose effects primarily fall into this class include `grep`, `cp`, `mv`, `ln`, `gcc`, `pwd`, and `jq`.

Detectable effects are effects which INCR can observe during tracing but cannot precisely replay, and for which it therefore conservatively re-runs the command when it detects such effects. These include all memoizable effects, the process's environment variables, filesystem effects on character devices, block devices, and sockets. INCR's `-N` flag (§5) extends this set to clock system calls, network system calls, and entropy-supplying system calls. Examples of commands whose effects primarily fall in this class include `wget`, `curl`, and `shuf`. If all components in a shell program only perform effects in this class, INCR can guarantee correct re-execution.

Blocked effects are effects that (attempt to) cross the boundaries of INCR's probes during execution. INCR blocks these effects because its underlying probe environment [50] requires `user`, `mount`, and `pid` namespaces to instantiate a private filesystem view and correctly interpose on the `/proc` filesystem. For instance, sending a signal to a process outside the same process sub-tree, *i.e.*, outside the probe, will fail—with the signal never reaching its intended target. Commands that spawn subprocesses meant to outlive their original runtime will not be reflected outside the probe environment. A large number of process- and job-control commands such as `kill`, `jobs`, `bg`, and `fg` are implemented as shell built-ins, which INCR does not probe and can thus perform their effects freely. Other blocked behaviors include concurrent access to shared state across probe environments, such as command `a` producing files that command `b` consumes concurrently in `a | b`. Command effects do not inherently belong in this class; rather they are specific to cross-command interactions that will be blocked by INCR's underlying semisolation mechanisms. Blocked effects can be safely allowed using INCR's configurations (§7).

Ignored effects are effects that can be observed or blocked by INCR, but are not supported by the current prototype. These effects are assumed to not affect a component's observable behavior and include querying the current system state—*e.g.*, host and kernel identity, scheduling state, and system-wide resource statistics—POSIX message queues, semaphore sets, and shared memory segments. For example, `hostname`'s `stdout` will be replayed even if the system's hostname changes across re-executions. Example commands in this class include `uname`, `getpriority`, `sysinfo`.

Out-of-scope behaviors include timing-sensitive components whose behavior depends on the precise elapsed time; INCR-aware components that change their behavior when they detect they are executing inside probes; non-deterministic components seeded by state undetectable by INCR; ones not terminating, *e.g.*, daemons; and ones that interact with system state outside system-call tracing, *e.g.*, by changing kernel state via module loading. These behaviors will require new insights and approaches that are very different from the ones INCR takes. Examples of commands include `systemd-detect-virt`, binaries seeded by `RDRAND`, `at`, `insmod`, and polling loops built around `sleep`.

Table 1: **Effect classes.** INCR's implementation distinguishes between several classes of effects described below.

Class	Example effect	Commands
Detectable	perform network communication	`wget`, `curl`
Memoizable	create or overwrite a regular file	`cp`, `echo`, `mv`
Blocked	signal a non-child process	`pkill`
Ignored	query system hostname	`hostname`
Out of scope	wait for a given amount of time	`sleep`

4 Dependency Tracking

This section describes how INCR acquires fine-grained information about command dependencies.

Inserting interposition probes: Shell programs feature complex control flow and interaction with the environment, complicating the extraction of command dependencies and their effects. INCR performs all dependency tracking at runtime, following a component-centric approach: it isolates each command's effects and tracks its dependencies individually (Fig. 3). It allows tracking commands on a per-effect basis (§5), enabling far more precise memoization and reuse than probing the script as a monolith.

INCR first parses the shell script using `libbash`, which exposes Bash's parsing subsystem as a library. It then walks the program's abstract-syntax tree (AST) and inserts probes as higher-order commands that wrap the original command invocation. For example, INCR transforms the invocation `rm $path` into `./probe rm $path`. The probe overwrites the `$0` variable to point to the original command and then invokes it with its original arguments. Each probe tracks its command's dependencies and memoizes its effects. This includes any subprocesses spawned by the command, meaning that a command such as `xargs` will be incrementalized as a whole.

Probes are only placed at trackable commands, which leaves out shell-defined functions, built-in commands, and other components that cannot be resolved inside the shell (*e.g.*, `alias`). To deal with these shell-specific constructs, INCR ap-

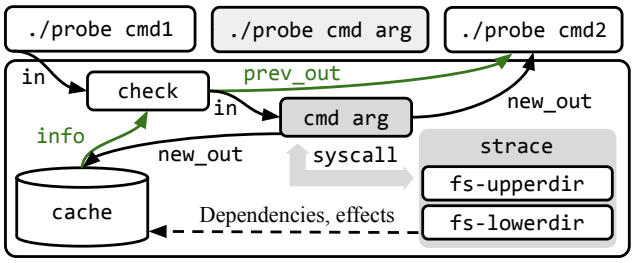


Fig. 3: **Interposition probes**. INCR inserts lightweight interposition probes before each command in a shell program. A command's probe records its dependencies, inputs, outputs, and other effects by tracing its system calls (gray arrows) and memoizing those effects in a cache. On subsequent executions, the probe checks its dependencies against prior runs retrieved from INCR's cache. If they are unchanged, the probe reuses prior results (green arrows); otherwise, it re-executes and re-traces the command (gray arrows).

plies a small set of lightweight syntactic analyses and transformations. In particular, INCR keeps track of and does not place probes on built-in commands (using the closed set given by `compgen -b`), function definitions, `alias`ed invocations, and backgrounded computations (`&` and `coproc`). For other syntactic constructs that create non-trackable side effects, INCR transforms them into equivalent, trackable commands. It converts redirections (`>`, `>>`, `<>`, `>|`, and their number-prefixed variants) into `dd` invocations before inserting probes.

Detecting dependencies: Correct incrementalization requires comprehensive tracking of each command's runtime dependencies.

To precisely capture each command's effects on the filesystem, INCR executes them within a `try` semisolate [50], which provides a private, copy-on-write view of the filesystem. Before each command executes, INCR instantiates a new OverlayFS mount that composes a read-only lower layer (`lowerdir`) with a writable upper layer (`upperdir`) into a unified merged view for each top-level directory in the current filesystem and `unshares` to change the process's root directory to the merged view. To suppress components with unsupported dependencies, INCR also instantiates a `user`, `pid`, and `mount` namespace, blocking non-filesystem cross-process interactions such as signals and shared memory. All modifications performed by the command are captured in the `upperdir`, while the `lowerdirs` offer transparent read-only access to the underlying filesystem. Upon completion, INCR scans the `upperdir` to identify modifications made by the command and commits them from the `upperdir` back to the persistent filesystem. Furthermore, INCR stores the `upperdir` for reuse during subsequent executions (§5).

Scanning the upperdir only reveals write dependencies. To capture read dependencies, INCR monitors system calls made by each command during its execution. To lower overheads, INCR intercepts a subset of system calls within the scope of dependencies it needs to track—specifically, `fork`, `exec`, and all file-related system calls included in `strace`'s `%file` system set. INCR uses `seccomp-BPF` to filter out irrelevant system calls and reduce context switching overhead. The set of observed system calls can be extended to detect other types of dependencies, at the expense of higher tracking overhead.

INCR also tracks environment dependencies such as environment variables and function declarations. These are pervasive in shell scripts as commands often rely on environment variables for configuration (*e.g.*, `LC_ALL` affects `sort`'s ordering behavior). Before executing each command, INCR captures the current environment variables and function declarations in the shell. If this set differs on re-execution, INCR reruns the command. Environment modifications made during execution are handled automatically as INCR captures environment snapshots on a per-component basis.

False and noisy dependencies: Within the execution environment, *false* and *noisy* dependencies may lead to unnecessary re-executions. For example, INCR cannot determine which environment variables a command depends on, because after process startup the environment resides in the process's user-space memory and accesses to it do not go through the system call interface. This forces INCR to conservatively treat all environment variables as dependencies, which may trigger re-executions even when variables that do not impact command behavior change across runs. To mitigate this, INCR implements a distribution-aware filter that discards noisy environment variables. Currently, INCR is aware of noisy variables inside the Debian and Ubuntu distributions, specifically those related to session management and `tty` settings.

5 Memoization and Reuse

This section explains how INCR memoizes command dependencies and effects at runtime and reuses these memoized results when appropriate.

Efficient memoization: INCR stores each command's dependencies and effects collected during effect tracking (§4) in a cache directory on the host system. Commands are indexed by their invocation arguments, environment variables, and `stdin` stream hash. During re-execution, INCR indexes into the cache to check if commands can be skipped and to fetch memoized results.

Each command's probe only has visibility into its own read and write dependencies. After recording them, INCR generates a dependency file. INCR tracks read and write dependencies differently as an optimization, using the insight that components in dynamic runtimes resolve sub-components at runtime, leading to many more read than write dependencies [85]. Write dependencies are tracked using content hashes while read dependencies are tracked using last modification timestamps, which are much cheaper to extract. To check if a write dependency has changed, INCR compares the file's current

hash to its stored hash. This allows INCR to skip re-execution if upstream commands modify a file but produce the same output (*e.g.*, `sort` may produce the same output given different inputs), while also distinguishing between same-content overwrites and appends, ensuring correct reuse for non-idempotent writes. To check if a read dependency has changed, INCR checks if the file's timestamp has changed. However, this over-approximates changes when a command reads a file modified upstream. To mitigate this, INCR avoids updating file timestamps when applying memoized writes, ensuring that commands do not detect spurious timestamp changes.

Furthermore, INCR stores each command's transient data streams, exit code, and filesystem effects alongside its dependency file to memoize its effects. To capture transient data streams, INCR duplicates a command's `stdout` and `stderr` to files within the cache directory at runtime. To capture the exit code, INCR waits for and records the command's exit status after execution. To capture filesystem effects, INCR directly stores the `upperdir` generated by OverlayFS during dependency tracking. INCR memoizes only output streams, exit status, and replayable local filesystem effects (§3). All other externally visible effects are either used to disable reuse, blocked by isolation, or left outside the model.

Safe reuse: Correct incremental execution requires detecting when a command's dependencies remain unchanged and applying prior results correctly. During each execution, INCR compares each command's current dependencies against its dependency file. If any differ, INCR re-executes the command and records its new dependencies and effects; otherwise, INCR skips execution and applies its memoized results.

For output streams, INCR directly streams memoized `stdout` and `stderr` to their corresponding file descriptors. For filesystem effects, INCR scans the memoized OverlayFS `upperdir` that contains a component's post-execution set of filesystem effects. Inside it, OverlayFS represents created or modified files as regular files, deleted files as *whiteouts* (character devices with major and minor numbers `0,0`), created directories as regular directories, and overwritten or deleted directories as directories that have the `user.overlay.opaque` extended attribute set. To reuse these effects, INCR iterates over the `upperdir`, matches each filetype to the corresponding effect, and applies each change to the current environment.

INCR, with the `-N` flag, employs several best-effort heuristics to detect if a component (1) relies on the system time; it executes clock system calls, (2) performs network interactions; it executes network system calls, or (3) may be nondeterministic; it executes entropy-supplying system calls. In these cases, it will conservatively disable reuse for such components. Furthermore, INCR accepts optional annotation and configuration (§7) that allow developers to mark commands as non-incrementalizable.

6 Runtime Optimizations

INCR's tracing (§4) and memoization (§5) mechanisms may introduce significant runtime and storage overheads. This section describes several optimizations that make INCR practical for real-world workloads.

Eager stream-processing: The shell's streaming execution model allows commands in a pipeline to start processing as soon as their upstream commands begin emitting output. This complicates incrementalization. Consider `awk '{print $1}' $f | grep 'x'`. If the file `$f` points to changes but its first column remains the same, then `awk`'s output is unchanged, meaning that `grep 'x'` can reuse its memoized results. However, blocking on each pipeline stage to decide reuse disables the shell's streaming semantics and incurs high overhead for long pipelines.

To address this challenge, INCR employs *eager stream processing*, a mechanism that allows reuse decisions to be made on the fly, while streaming outputs. When executing a pipeline, INCR's probes begin executing each command as soon as input is available and without waiting to check if re-execution can be skipped. Each probe buffers its `stdin` stream in memory while computing a rolling hash and forwarding it to the probe's corresponding command. Then, once the probe finishes hashing `stdin`, it checks if the command's inputs have changed. Importantly, probes typically finish hashing `stdin` *long before* their commands finish processing it, especially for compute-intensive commands, enabling them to determine if memoized results can be reused early in each re-execution run. If the command's dependencies and `stdin` have not changed, then INCR sends a `KILL` signal to the command and outputs its memoized results starting from where it left off. Otherwise, INCR continues running the command and tracking its effects as usual. This process continues in a chain for all the stages in a pipeline.

However, aborting a command that has filesystem side effects early may produce an inconsistent system state if it is in the middle of a modification. INCR's effect isolation mechanism (§4) addresses this: each command's effects are entirely contained within its OverlayFS `upperdir`. Therefore, INCR simply discards a command's `upperdir` when aborting its execution and applies its memoized effects instead.

Introspection: Effect isolation (§4) is necessary to contain and memoize side-effectful commands. However, creating, copying, and committing OverlayFS directories generates potentially expensive overheads. To reduce such overheads, INCR employs *introspection* to detect, using knowledge from prior runs, whether commands need effect isolation. Specifically, INCR uses the tracing information to identify commands that do not perform filesystem modifications. INCR optimistically assumes that, given the same arguments, these commands remain effect-free in subsequent runs, similar to assumptions made by prior systems [42,71]. It then runs these commands without isolation.

However, if such a command modifies the filesystem in a subsequent run—as detected by INCR's tracing mechanism—INCR *still* guarantees correctness. INCR completes the command's execution as normal, correctly applying its effects to the filesystem. It then revokes the command's effect-free designation and invalidates its cache entry to avoid reusing potentially stale effects in future runs. In the next run, INCR re-executes the command with effect isolation enabled.

Certain commands may be practically effect-free but still create temporary files during execution that are cleaned up before they exit. For example, `sort` creates temporary files if its input is too large to fit into memory. INCR assumes that files created and removed within the same command execution are temporary, and does not record them as dependencies. This approach allows INCR to consider such commands effect-free and to skip effect isolation in subsequent runs.

Storage compaction: Memoizing dependencies and effects for commands that produce large outputs or across shell programs that have several opportunities for incrementalization can incur significant storage overheads. To mitigate this issue, INCR employs *storage compaction*, compressing memoized data with a configurable compression level. INCR uses the Zstandard compression algorithm, which provides acceptable performance and compression ratio [17].

To avoid invalidating memoized data when the compression level changes, INCR records the compression level used for each memoized output. Therefore, INCR can decompress and reuse data generated under any previous configuration.

7 Optional Annotations and Configurations

INCR can also leverage command annotations made available by other systems and optional developer configurations to lower its overheads and increase or decrease its incrementalization fidelity.

Existing crowdsourced annotations: INCR leverages crowdsourced command annotations made available by other systems [39, 42, 63, 71, 84] to further increase the granularity of incrementalization. These annotations target parallelization and distribution opportunities, but the information they expose can benefit INCR's analysis and accelerated re-execution. Each annotation maps a command invocation to a set of properties. For example, a combined annotation from POSH [71] and PaSh [42] for `cat` is:

```
cat: [], stateless, splittable_args
```

This annotation indicates that `cat` is stateless and splittable across its arguments when invoked without any flags. The following are the properties that INCR can exploit.

INCR exploits *statelessness*, a classification from both POSH [71] and PaSh [84] for commands that operate on each input line independently without maintaining any internal state across lines. For example, invocations of `grep` without `-c` are classified as stateless over their `stdin` stream. This allows INCR to re-execute only the affected parts of a stateless command's input stream. It splits the inputs of stateless commands into smaller chunks and memoizes each chunk separately and in parallel. Data streams are split using content-defined chunking [89], which produces chunks that are stable across input perturbations. If a modification to a stateless command's input affects only a few chunks (*e.g.*, when a log file is extended with new events), then memoized results of unaffected chunks can be reused.

INCR exploits *purity*, a classification for commands that do not modify the filesystem outside of a defined set of inputs and outputs. This classification comes from PaSh's parallelizability annotations [84]. For example, an annotation for `grep -f p.txt` classifies it as pure with read dependencies from `stdin` and `p.txt`, and a write dependency to `stdout`. INCR skips effect isolation and tracing (§4) on pure commands. For many common commands such as `cat`, `grep`, and `tr`, this reduces INCR's overhead when introspection (§6) has not yet detected that the command is pure.

INCR exploits *argument independence*, a classification from POSH [71] for commands that can be executed independently for each argument. For example, an annotation for `grep -H 'p' f1 f2` classifies it as argument-independent across its arguments. Specifically, if `f2` changes, then only `grep -H 'p' f2` needs to be re-executed. INCR performs argument-level incrementalization on these commands by syntactically transforming each invocation: INCR splits the single large invocation into multiple invocations inside a subshell, each with a single argument. For example, it transforms the previous `grep` invocation into `(grep -H 'p' f1 ; grep -H 'p' f2)`. INCR then places separate probes on each invocation and memoizes and reuses their results independently.

Optional developer configuration: Developers can also optionally configure INCR for specific script fragments to further improve performance, exploiting knowledge of a script's behavior and use patterns. To support such configurations in a backward-compatible fashion, INCR exposes a special annotation that it detects during parsing. These annotations instruct INCR either to disable incrementalization for a command or to group multiple commands together to memoize them as a single unit. Such annotations are expressed as assignments to the placeholder environment variable `INCR` before commands. For example, configuring INCR to skip `cat` and `grep` within a pipeline looks as follows:

```
INCR="s" cat f | INCR="s" grep . | tr a-z A-Z
```

Configuring INCR to group commands together by marking the first and last commands in the group looks as follows:

```
cat f | INCR="g" grep . | INCR="ug" tr a-z A-Z
```

INCR removes the `INCR` environment variable before executing the annotated commands.

Disabling incrementalization reduces runtime overheads for program fragments that perform minimal computation or

Table 2: **Benchmark summary.** Summary of all benchmarks used to evaluate INCR and their characteristics. Benchmarks are categorized based on the delta type: addition (+), deletion (−), modification (~), or a combination thereof and the reason for the change: behavior (B), wrong command (C), wrong flag (F), exploration (E), summarization (S), optimization (O), LLM assistance (L), replacement (R), input update (I), debugging (D), aggregation (A), or visualization (V).

	Benchmark	Description	Input	# deltas	LoC	Deltas	Source
1	**dpt**	Segments and classifies images.	2.4 GB	9	84	O 2F I A F V 2C	[48, 81]
2	**bio**	Extracts genome sequences.	3.5 GB	6	104	I B I D 2E	[11, 70]
3	**dict**	Calculates top-n most frequent words.	30 MB	1	2	S	[5]
4	**ngram**	Extends indexing stage to 1–3 grams.	106 MB	2	58	2B	[16]
5	**uppercase**	Analyzes word distribution.	200 MB	1	18	B	[16]
6	**unixgame**	Solves a Unix game question.	1.0 GB	5	6	2E E 2E	[8, 82]
7	**nginx**	Detects broken links.	974 MB	21	243	7E 5A R 5D S F O	[71]
8	**weather**	Calculates temperature statistics.	887 MB	2	138	2E	[87]
9	**covid**	Calculates public transit statistics.	381 MB	4	48	E E E E	[80]
10	**spell**	Finds spelling errors.	3.1 GB	6	48	3D E 2B	[5]
11	**poet**	Extracts used and rhyming words.	1 GB	3	47	3E	[16]
12	**image**	Renames images with a VLM.	38 MB	6	45	2D 2D L O	[48, 81]
13	**music**	Prepares music files for sharing.	16 MB	6	52	3L D O O	[71, 72, 78]
14	**beginner**	Inspects system logs.	974 MB	13	28	5F 4E A O E S	[71, 76]

have complex side effects. For example, configuring INCR to skip `cat` and `grep` commands in the example above avoids effect memoization for trivial commands that emit large outputs, thereby eliminating both runtime and storage overheads.

Grouping commands reduces overheads for program fragments that are unlikely to benefit from fine-grained, individual incrementalization. Additionally, grouping a fragment that is considered final and not expected to change allows INCR to focus only on fragments that will benefit from incrementalization. For example, configuring INCR to group the pipeline above incurs effect isolation and tracing overheads only once for the entire sequence rather than once per command. This can significantly improve performance for pipelines composed of many inexpensive commands.

8 Evaluation

This section applies INCR to 14 incremental development scenarios (totaling 85 deltas) to characterize its re-execution benefits (§8.1), its runtime overheads (§8.2), its behavioral equivalence (§8.3), the effectiveness of its optimizations (§8.4), and the impact of optional annotations (§8.5).

Benchmarks and modifications: To evaluate INCR, we use benchmarks from the Koala suite of real-world shell workloads [49] and modifications thereof. These benchmarks span data processing, machine learning, and system administration, with input sizes ranging from a few to several gigabytes, and consisting of a total of 85 deltas, summarized in Tab. 2.

Their modifications include fixing wrong commands or arguments, developing new functionality, exploring characteristics of data, and introducing modifications from LLM suggestions—collected from: (1) discussions with the original developers of these workloads, with modifications mirroring the iterations that led to the final script (seconds to minutes between iterations across **dpt**, **covid**, **weather**, **ngram**, **uppercase**, and **beginner**); (2) the Git commit history of these benchmarks (minutes to hours between iterations across **bio**, **unixgame**, and **nginx**); and (3) manually constructed edits to reflect realistic development trajectories (seconds to minutes between iterations across **dict**, **spell**, **poet**, **image**, and **music**). All modifications arise from a diverse set of goals (summarized in Tab. 2), resulting in three types of program deltas: in-place edits to existing commands (~), *e.g.*, changing command arguments; additions of new commands or stages (+); and removals of existing commands or stages from the script (−).

The Koala benchmarks used in INCR are only the ones for which we could identify and collect such program modifications. INCR contains only the digital pyramid text pipeline from **inference** (**dpt**), genome-sequencing from **bio** (**bio**), frequency, n-gram, transliteration, and poem generation from **nlp** (**dict**, **ngram**, **uppercase**, and **poet**), **unixfun**'s chess puzzle (**unixgame**), log-analysis program of **analytics** (**nginx**), core temperature computation from **weather**, excluding the Tufte weather plot (**weather**), all except the last, monolithic script of **covid** (**covid**), spell-checking of **oneliners** (**spell**), VLM-assisted image annotation from **inference** (**image**), and conversion, compression, and encryption programs from **file-mod** (**music**). Finally, **beginner** is an entirely new benchmark not available in Koala, representative of non-expert developers who perform multiple exploratory modifications in their development process.

Scenarios, in detail: The **dpt** benchmark (discussed in §2) segments and classifies hieroglyph images from a single

expedition. Its changes include optimizing by removing a redundant image-resizing stage (O), correcting awk's printing format (2F), filtering out irregular files in the **for** loop condition (I), aggregating the classification results (A F), and finally visualizing classification results with three iterations (V 2C). Inputs total 2.4GB, including model weights and ultra-resolution images.

The **bio** benchmark processes genomic data to extract chromosome-specific subsets using a six-stage pipeline. Changes include iterating over all BAM files instead of a hard-coded testing file (I), extracting per-chromosome reads (B), driving processing from an input file of population-sample pairs (I), ignoring malformed or incomplete entries in that list (D), and plotting summary statistics for per-sample coverage (2E). The script is applied on a corpus of genomic data comprising 15 samples and 3.5GB of aligned reads.

The **dict** benchmark counts the frequency of each word in a corpus using a four-stage pipeline. The change modifies the script to only output the top-n most frequent words, by adding a stage to `sort` words by frequency and a last `head` stage (S). The program operates over a large text corpus of 5.2M words, totaling 30MB of text from Project Gutenberg [35].

The **ngram** benchmark starts with a unigram computation pipeline of `tr`, `sort`, and `uniq`, extended to compute bigrams (B) and trigrams (B). It processes a 16-million-word, 106MB snapshot of Project Gutenberg [35].

The **uppercase** benchmark extracts all unique capitalized words from a large text corpus using multiple stages of `grep`, `tr`, and `sort`. The first delta modifies the script to count the occurrences of unique capitalized words by adding `sort -u` to sort the words by their frequency in descending order in the middle of the pipeline (B). Its input is a 33-million-word snapshot of Project Gutenberg, totaling 200MB [35].

The **unixgame** benchmark solves a series of questions from the Unix 50th anniversary game. It first counts the total number of rounds using `grep '.'` and `wc -l`. Changes insert `grep 'x'` to capture moves from one side (E), then a `cut` and `grep -v '[KQRBN]'` to count specific captures (E), then `cut`, `sort`, and `uniq -c` to count occurrences of each capture (E), normalize lowercase identifiers via `tr '[a-z]' 'P'` (E), and counting occurrence frequency with `sort -r`, `uniq -c`, `head`, and `awk` (E). Inputs are a chess dataset from Lichess [20] totaling 19M chess moves and 1.0 GB.

The **nginx** benchmark analyzes Nginx server log entries to identify broken links. A series of changes expands its scope, *e.g.*, extracting status codes, listing request paths that lead to `402` or `502` errors, identifying suspicious requests, counting unique clients, extracting referrers, sorting and ranking error-inducing URLs, and summarizing top 404-error paths (7E 5A R 5D S). Another set modifies ordering and summarizing behavior to use consistent reverse-numeric sorting (F), and deleting unnecessary outputs (O). It operates on a 5-million-record, 974MB web-server log [71].

The **weather** benchmark processes a large weather dataset to compute maximum temperatures for each day between 1995 and 2000. The changes modify the script to add two additional statistics by computing the minimum (E) and average temperature (E). Its input is an 887MB weather dataset of 3.6M temperature records from the National Oceanic and Atmospheric Administration (NOAA) [64].

The **covid** benchmark analyzes public transit data collected during COVID-19 to compute a series of statistics. The changes introduce more metrics such as total vehicles per day, days per vehicle (E), hours per vehicle (E), monitored hours per day (E), and hours per bus (E). Its input contains 5M bus schedule records, totaling 381MB [80].

The **spell** benchmark analyzes spelling mistakes in a large text corpus. It includes six changes: removing non-printable characters and turning the input into a word stream (D), lowercasing that stream (D), removing punctuation (D), sorting words in alphabetical order (E), reporting words not found in a dictionary (B), finally comparing only unique words against a dictionary to identify misspelled words (B). The input is a collection of 9001 books, totaling 527M words and 3.1GB of text [35].

The **poet** benchmark counts the frequency of each word found in a given directory containing text files. Changes include replacing the single-file input with a concatenated corpus of all poetry files to enable global text statistics (E), adding a second output that reports unique words in alphabetical order (E), and finally introducing a third output that orders words by rhyme by reversing strings prior to sorting and then restoring their original orientation (E). The input is a corpus of 3001 books, totaling 22.4M lines and 1GB of text from Project Gutenberg [35].

The **image** benchmark renames images based on their content using a vision-language model—`GPT-4o mini`. Changes include replacing spaces in LLM-generated titles with underscores to form basic filenames (D), lowercasing all characters for consistency (D), stripping non-alphanumeric, non-underscore, and non-dash characters to guarantee filesystem-safe names (D), separating the cleaned stem into a reusable `base` variable for clearer filename construction (L), consolidating the sanitization steps into a single `sed` invocation (O), and finally removing the mode suffix so outputs use only the cleaned base title (D). Its input is a set of 11 images from a browsing session (totaling 38MB) [48].

The **music** benchmark captures a vibe-coding development loop, where a user iteratively refines a multimedia pipeline through loosely guided interactions with an LLM [29]. Starting from a simple `mp3` to `wav` conversion with `ffmpeg`, changes include `tar`-ing each newly produced `.wav` file alongside conversion so every MP3 immediately gets its own archive (L), refactoring into a single post-loop that aggregates all WAVs into a combined tarball (D), encrypting it with `openssl` (L), introducing a configurable encryption-key variable instead of a hardcoded one (O), compressing individual WAV files to `.gz` before archiving (L),

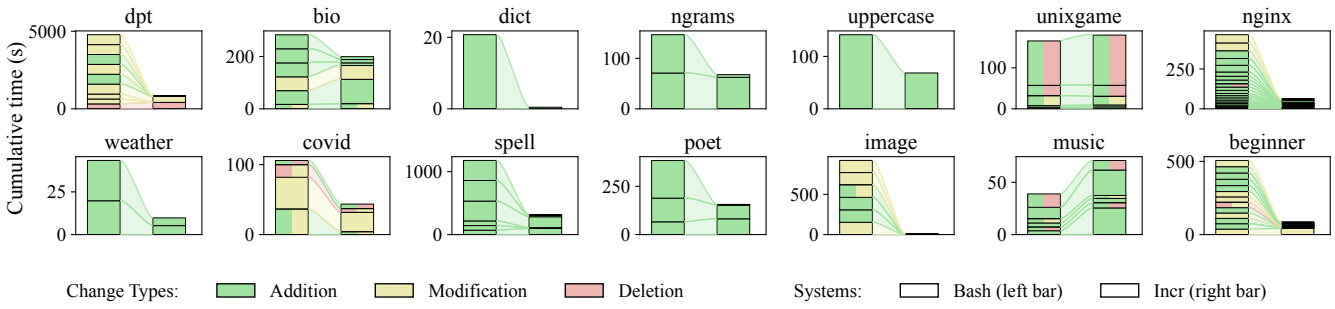


Fig. 4: **INCR's speedup on incremental changes**. Each vertical bar, *i.e.*, group of blocks, represents the execution time of a benchmark change during incremental development. Each block within a bar represents the time taken by the corresponding re-execution of the benchmark after a change. The curves connect the same block across the two systems.

and finally simplifying the layout by dropping per-file compression in favor of `gzip`-ing (`O`). Its input is a collection of 20 public-domain music files, totaling 16MB [49].

The **`beginner`** benchmark inspects system logs to identify failed login attempts. Starting from a numeric sort of the system log, changes include switching from numeric sorting to lexicographic sorting (`F`), counting duplicate lines (`E`), filtering for lines containing specific patterns—first case-sensitive (`E`) and later case-insensitive (`F`), counting matching lines (`A`), then consolidating filtering and counting (`O`), extracting the first two fields from the counted output (`E`), extracting different fields (`F`), grouping identical field pairs after sorting (`E`), numerically sorting the grouped results (`E`), reversing the numeric sort to rank largest first (`A`), selecting the top ten entries (`S`), and adjusting the `head` call to the explicit `head -n 10` form (`F`). Its input contains 5 million system logs, totaling 974MB [71].

Experimental setup: Experiments were conducted on a CloudLab m510 machine with an 8-core Intel Xeon D-1548 CPU at 2.0 GHz, 64 GB RAM, 256 GB NVMe storage, a 10 Gbps connection, running Ubuntu 22.04 on Linux 5.15.

8.1 Re-execution Performance

By how much does INCR accelerate re-execution?

Methodology: For each program modification, we measure the execution time of the script under INCR and Bash, and report the speedup as the ratio of Bash's runtime to INCR's runtime per incremental step. We run each re-execution 3 times and report the mean.

Results: Fig. 4 presents INCR's speedup—achieved without annotations—over Bash across all benchmarks and incremental re-executions. Out of 85 re-executions, INCR achieves a speedup in 69 cases and a slowdown in 16 cases: across the former set, it achieves an average speedup of 34.2×, with a maximum of 373.3× and a minimum of 1.003×; across the latter set, INCR incurs an average slowdown of 0.73×, with a minimum of 0.15× and a maximum of 0.95×.

Discussion: INCR's substantial re-execution speedups stem from its ability to track fine-grained dependencies and safely reuse previously computed results, thereby eliminating redundant computation. For example, INCR reuses LLM-generated image annotations in the **`image`** benchmark when incremental changes only modify the post-processing logic, reducing the execution time from 155.55 seconds to 1.62 seconds, achieving a speedup of 96.02×. INCR does not introduce a significant difference in accelerating different types of changes, primarily because INCR's fine-grained dependency tracking effectively identifies unaffected commands and reuses their results regardless of the type of modification.

INCR's overheads stem primarily from its use of system call tracing and isolation, which together enable the system to capture fine-grained dependencies and manage memoized results. These fixed costs become more pronounced and produce minor slowdowns in benchmarks dominated by many short-lived commands or in those with complex dependency behaviors. For example, in the **`unixgame`** benchmark—which consists of a sequence of short-running commands such as `tr ' ' '\n'`—the final iteration modifies only the third command, requiring full re-execution of the remaining eight commands. For these commands, fixed costs make up a larger fraction of the processing time, resulting in an increase in execution time from 107.8 seconds to 123.6 seconds. Moreover, in the first iteration of the **`music`** benchmark, tracing the `ffmpeg` and `tar` commands increases execution time from 3.7s to 12.4s, as they both perform thousands of system calls.

8.2 Time and Space Overheads

By how much do INCR's tracing, isolation, and memoization mechanisms slow down the initial execution of a script (before it benefits from incremental re-execution) and increase space usage during execution (by memoizing command effects)?

Methodology: To characterize INCR's overheads, we measure its execution time and peak space usage relative to Bash across all benchmarks with no script modifications. Time is re-

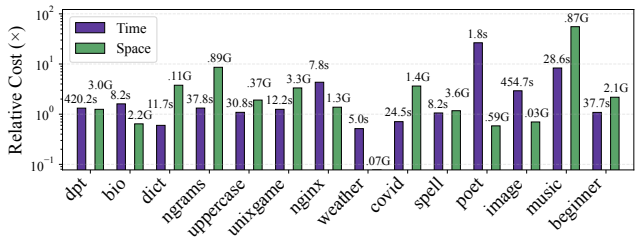


Fig. 5: **INCR's time and space overheads**. Each set of bars represents INCR's overhead per benchmark during the first run and the peak space usage over the benchmark's input size across all re-executions. The x-axis represents benchmarks. INCR's absolute execution time (s) and peak cache size (GB) are noted above each bar.

ported as the ratio between INCR's first execution and Bash's execution on the same script. This is the worst-case scenario for INCR: all tracing, isolation, and memoization overheads are incurred without any benefits from incremental execution. Space consumption is measured as INCR's cache size relative to the benchmark's original input size. In Fig. 5, the numbers above the bars denote the absolute first-run execution time for INCR (time bars) and the absolute cache size (space bars).

Results: Fig. 5 presents INCR's time and space overheads using these relative ratios. For benchmarks where INCR's execution exceeds five seconds, INCR exhibits a mean first-run time ratio of 2.01×, reaching 8.32× in the worst case on **`music`** and 0.52× in the best case on **`weather`**. INCR's cache occupies on average 6.05× of the original input size, ranging from 0.02× (0.04GB) for **`image`** in the best case and up to 55.44× (0.87GB) for **`music`** in the worst case. The average drops to 2.15× without the **`music`** outlier, which contains a tight **`for`** loop that inflates the cache as each iteration corresponds to multiple new cache entries. In absolute terms, INCR's cache size ranges from 0.04GB for **`image`** to 3.6GB for **`spell`**.

Discussion: INCR's time overheads stem from its tracing, isolation, and memoization mechanisms, while its space overheads stem naturally from storing memoized command outputs for later reuse. However, INCR can outperform Bash even on initial executions because its eager stream processing pulls from producers as soon as data is available, often avoiding blocking in ordinary shell pipelines. INCR's cold-start time overheads are often offset by the benefits of incremental execution in real-world development scenarios, where scripts are frequently modified and re-executed.

8.3 Behavioral Equivalence

Does INCR preserve behavioral equivalence to an unmodified shell interpreter during incremental re-execution?

Methodology: We characterize INCR's behavioral equivalence to the underlying shell interpreter in two ways. First, we apply it to all real-world scenarios in Tab. 2 and all unmodified Koala benchmarks (commit `e1610d9`), each executed twice. Using the same experimental setup as in §8.1, we confirm that the final outputs and exit code produced by both INCR and Bash match across all re-executions (confirmed using Koala's `verify.sh`).

Second, we apply INCR to the standard Bash test suite. Version `5.2.37(1)-release` consists of 83 test categories, each corresponding to a specific feature of Bash, totaling 534 test files and 22,064 LoC. The suite's ground-truth files contain 10,282 lines covering features from variable expansion, job control, invocation of external utilities (*e.g.*, `grep`, `cat`, and `awk`), covering the full POSIX shell standard and all of Bash's extensions. Because vanilla Bash itself differs from the bundled expected outputs on 362 lines, we use Bash as the ground truth and compare INCR's outputs against it. The test suite is unusually demanding: it is sensitive to whitespace, quoting, error text, and ordering, and many tests hinge on historically accumulated interpreter corner cases. For example, the suite checks how temporary environment assignments interact with variable expansion, how process substitutions affect `$!` and `wait`, and how shell options affect error handling.

Nineteen parser-error cases are considered out of scope because `libbash`'s unparsing diverges from Bash in line numbers and non-UTF-8 escape printing.

Results: Applied to the Bash test suite, INCR only differs by 3 ground-truth lines out of 10,282 (99.9% equivalence, Tab. 3). Differences come from (1; 2 diffs) recursive `alias` definitions, which INCR cannot identify during probe placement, thus placing a probe to the aliased command whose expansion (unlike normal shell expansion) INCR cannot observe; (2; 1 diff) an `execscript` test that unsets `PATH`, causing INCR to fail to locate its own dependencies. We re-ran all tests without clearing INCR's cache, obtaining identical results, to confirm reuse does not affect equivalence.

Discussion: INCR's broad compatibility required support for

Table 3: **INCR's behavioral equivalence.** The table shows several Bash test groups, example categories, the number of tests per category, and INCR's results.

Group	Example categories	INCR
Globbing	`extglob`, `globstar`	868/868 (100%)
Data structs.	`array`, `array2`, `assoc`	1261/1261 (100%)
Quoting	`quote`, `quotearray`	877/877 (100%)
Expansion	`comsub`, `alias`	1701/1703 (99.9%)
Utils	`getopts`, `strip`	80/80 (100%)
Env.	`attr`, `history`, `read`	1610/1610 (100%)
Constructs	`parser`, `func`, `case`	1547/1547 (100%)
IPC	`jobs`, `execscript`	1355/1356 (99.9%)
POSIX	`posixexp`, `posixpipe`	536/536 (100%)
Options	`set-e`, `set-x`, `shopt`	444/444 (100%)
Total		10,279/10,282 (99.9%)

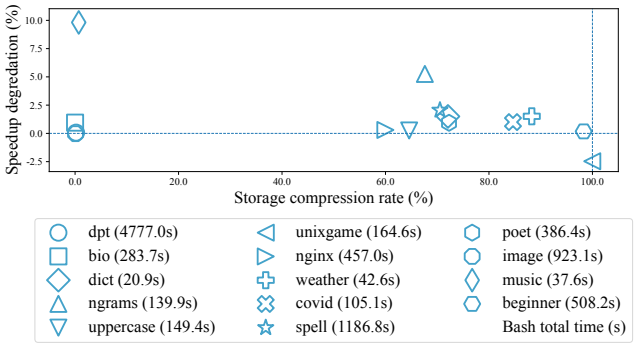


Fig. 6: **INCR's storage compaction impact on performance and storage**. Each point represents the storage reduction *vs.* speedup degradation with compaction.

all of Bash's invocation modes, types of expansion (*e.g.*, process substitution), correct escaping through the shell, probe placement that excludes non-trackable built-ins, backgrounding operators and effects such as permission changes, moves, named pipe creation, and symbolic links. INCR's remaining divergences, in `alias` and `execscript`, arise when scripts manipulate the shell environment in ways that prevent INCR from resolving script contents or locating its own runtime dependencies. Several approaches could be used to address these limitations, including deferring probe placement and packaging INCR as a standalone binary.

8.4 Effects of Runtime Optimizations

What are the benefits of various optimizations, including eager stream processing, introspection, and storage compaction?

Eager stream-processing: This optimization is always enabled (§8.1); when disabled, INCR waits for each command to finish before deciding whether the subsequent command needs to be executed.

We apply INCR to a synthetic program that executes a single 16-stage pipeline twice without any incremental change; all stages are streaming commands, *e.g.*, `grep` filtering lines and `sed` performing text substitutions. On the script's first execution, INCR takes 3m 22s with eager stream processing, reducing the execution time by 65.8% from 9m 50s without eager stream processing. On the script's second execution, INCR takes 10s to fetch the memoized results both with and without this optimization. Eager stream processing adds imperceptible overhead on reuse.

Introspection: This optimization is always enabled (§8.1); when disabled, INCR always re-executes commands within an isolation sandbox during incremental runs whether or not they have write dependencies.

We apply INCR to a synthetic script that includes a pipeline consisting of 20 commands that communicate only through their standard streams. Each incremental change modifies the input data processed by the pipeline, triggering re-execution of all commands. Without introspection, INCR takes 35s on all iterations. With introspection, INCR still takes 35s on the first iteration because it has not yet detected that the commands are pure, but only 31s (speedup: 13%) on subsequent iterations.

Storage compaction: This optimization is disabled by default (§8.1); when enabled, INCR compresses memoized dependencies and effects on disk, achieving significant space savings with slightly increased runtime overheads.

Fig. 6 shows that compaction reduces space usage across all benchmarks by an average of 55.7%, with a maximum of 100.0%, while trading off a 1.9% average speedup degradation and up to 9.8% runtime degradation. This optimization is especially beneficial for scripts that include longer pipelines, as each intermediate result is a separate cache entry.

8.5 Effects of Optional Annotations

How do optional crowdsourced annotations (§7) help INCR further accelerate incrementalization?

Methodology: We apply INCR with annotations to all benchmarks in Tab. 2. Additionally, we design two synthetic shell programs. The first program uses a five-stage pipeline of stateless command invocations—`cat`, `sed`, `awk`, `sed`, and `grep`. Changes append new lines to the input file. The second program invokes two argument-independent commands—`clang` and `sha256sum`—to compile and hash C files. Changes modify one of these C files.

Results: With annotations, INCR introduces an additional average speedup of 1.46× across all benchmarks, and up to 24.40× in the **`music`** benchmark. INCR lowers time overheads from an average of 101.05% to 43.55% and from a maximum of 732.45% to 278.15%. In the first synthetic program, INCR takes 29s on the first run (2.51×), and only 6s on the next two runs (12.17×), compared to 1m 13s on each run without annotations. In the second synthetic program, INCR takes 22.0s (4.9×) after changes, compared to 1m 48.8s without annotations.

Discussion: Annotations allow INCR to make informed decisions based on command semantics. For example, in the **`music`** benchmark, annotations indicating that the `ffmpeg` and `tar` commands are pure allow INCR to skip expensive tracing of these commands. Moreover, INCR applies chunked incrementalization for stateless commands in the first synthetic program, and decomposes each command's argument list into per-file invocations in the second synthetic program.

9 Related Work

Incremental computation: Incrementalization systems [1, 34, 37, 53] typically track fine-grained dependencies within one language, requiring language support or source modifications to expose dependency boundaries. INCR instead discovers dependencies at the *system level*, bringing incremental

behavior to environments that span multiple languages and opaque components.

Data processing systems [14,62,69,91] support incremental computation through dataflow or domain-specific computation models. Several production systems also provide elements of domain-specific incrementalization, such as Deno's aggressive caching of remote dependencies [22], LangChain's caching of language-model calls [51].

In contrast, INCR targets general-purpose shell environments with opaque components written in a variety of languages and arbitrary data-processing semantics.

Build systems: Build systems [26, 46, 56, 59] efficiently rebuild software after source code changes, often using developer-specified dependency graphs. Prior research reduces this burden by automatically inferring dependencies through execution tracing, thereby enabling incremental builds without explicit declarations [19, 38, 57, 75]. INCR fundamentally differs from these systems by targeting general computations that span data dependencies, transient effects, and mutable state that go beyond build artifacts.

Exploratory programming: Interactive environments such as notebooks [44] shorten the development feedback loop by exposing cell-based re-execution, but require manual control of dependencies [92]. Systems such as Jupyter [47, 55, 86] allow users to re-execute cells independently, but dependency boundaries are coarse-grained and users must decide which cells to rerun. INCR instead infers dependencies automatically across command boundaries, preserving the exploratory style of shell scripting without requiring users to manually decompose scripts into reusable cells.

Reactivity and view maintenance: Reactivity and view maintenance automatically maintain results as input data changes [2, 9, 25, 33, 52, 90, 91], often within specific domains such as relational databases [12] or MapReduce-style data processing [21]. INCR does not target reactivity.

Provenance tracking and reproducibility: Provenance systems [4, 13, 61, 68, 74] capture dependencies among entities to support auditing and forensics. INCR's goals and methods for extracting dependencies in shell programs fundamentally differ, but the provenance graphs produced by these systems are complementary to INCR's dependency tracking and could enable incremental execution across network boundaries.

Systems such as CDE [32], ReproZip [15], Nix [23], Guix [18], and Docker [58] package or specify execution dependencies to improve portability, reproducibility, and deployment reliability. Their goal is to recreate executions across environments. INCR instead accelerates repeated executions in scenarios where the environment is largely fixed but the program changes.

Other systems [6, 40, 65] record program execution for faithful replay on often unmodified applications. They target deterministic replay for debugging and reproducibility, whereas INCR targets accelerated incremental execution.

Research on the Shell: Recent systems such as POSH [71], PaSh [42], and Fractal [39] automatically parallelize or distribute shell programs. These systems target large-scale computations, rely on developer annotations to identify parallelizable and distributable fragments, and reduce latency of a single full execution. In contrast, INCR targets early-stage exploratory development, automatically infers dependencies and memoizes command effects at runtime, and reduces repeated work performed across re-executions. It can optionally leverage insights from these systems to increase incrementalization fidelity (§7).

Furthermore, a variety of systems improve the shell along many dimensions, including syscall refinement [27], fusion [36], elision [7], dataflow extension [78], synthesis [73], serverless execution [54], and mobile usage [88]. INCR is complementary to these works.

10 Conclusion

Fundamentally, INCR shows that bolt-on incrementalization atop unmodified shells is not only possible, but also broadly applicable—to accelerate development, debugging, optimization, and other settings where re-execution is prevalent. Its design includes lightweight effect tracking, safe memoization and reuse, optimizations that address key performance bottlenecks, and configurations that enable further incrementalization opportunities. Applied to real-world workflows, INCR delivers substantial re-execution speedups without program or environment modifications and retains behavioral equivalence to normal execution.

Availability

INCR, alongside all of its evaluation programs and data, is available as MIT-licensed open-source software:

https://github.com/atlas-brown/incr

Acknowledgements

We are thankful to the anonymous OSDI'26 reviewers for their feedback; the OSDI'26 Artifact Evaluation reviewers for their time; Justus Adam, Akshay Narayan, Deepti Raghavan, Malte Schwarzkopf, and the Brown CS2952R (Fall '25) participants for their input on several iterations of this paper. This material is based upon research supported by NSF awards CCF-2525351, CNS-2247687, and CNS-2312346, DARPA contract no. HR001124C0486, an Amazon Research Award (Fall 2024), a Google ML-and-Systems Junior Faculty Award, a seed grant from Brown University's Data Science Institute, and a Brown CS Faculty Innovation Award.

References

[1] Umut A. Acar, Guy E. Blelloch, and Robert Harper. Selective memoization. In *Proceedings of the 30th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '03, pages 14–25, New York, NY, USA, 2003. Association for Computing Machinery.

[2] Yanif Ahmad and Christoph Koch. Dbtoaster: a sql compiler for high-performance delta processing in main-memory databases. *Proc. VLDB Endow.*, 2(2):1566–1569, August 2009.

[3] Christelle Alvarez. The variability of ritual texts: Knowledge transfer at the interface of oral and written forms. In *Variability in the Earlier Egyptian Mortuary Texts*, pages 219–249. BRILL, October 2023.

[4] Adam Bates, Dave Tian, Kevin R. B. Butler, and Thomas Moyer. Trustworthy whole-system provenance for the Linux kernel. In *Proceedings of the 24th USENIX Security Symposium*, USENIX Security '15, pages 319–334, USA, 2015. USENIX Association.

[5] Jon Bentley, Don Knuth, and Doug McIlroy. Programming pearls: a literate program. *CACM*, 29(6):471–483, June 1986.

[6] Tom Bergan, Nicholas Hunt, Luis Ceze, and Steven D. Gribble. Deterministic process groups in dOS. In *Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation*, OSDI '10, pages 177–191, USA, 2010. USENIX Association.

[7] Emery D. Berger. Optimizing Shell Scripting Languages. Technical Report UMCS TR-2003-009, University of Massachusetts Amherst, 2003.

[8] Pawan Bhandari. Solutions to unixgame.io. https://git.io/Jf2dn, 2020. Accessed: 2020-04-14.

[9] Jose A. Blakeley, Per-Ake Larson, and Frank Wm Tompa. Efficiently updating materialized views. *SIGMOD Rec.*, 15(2):61–71, June 1986.

[10] French-Owen Calvin. Reflections on Building with OpenAI's API, 2024. Accessed: 2025-10-23.

[11] Enrico Cappellini, Frido Welker, Luca Pandolfi, et al. Early Pleistocene enamel proteome from Dmanisi resolves Stephanorhinus phylogeny. *Nature*, 574(7776):103–107, Oct 2019.

[12] Donald D. Chamberlin and Raymond F. Boyce. SEQUEL: A structured english query language. In *Proceedings of the 1974 ACM SIGFIDET Workshop on Data Description, Access and Control*, SIGFIDET '74, pages 249–264, New York, NY, USA, 1974. Association for Computing Machinery.

[13] Ang Chen, Yang Wu, Andreas Haeberlen, Boon Thau Loo, and Wenchao Zhou. Data provenance at internet scale: Architecture, experiences, and the road ahead. In *Proceedings of the 8th Conference on Innovative Data Systems Research*, CIDR '17, January 2017.

[14] Raymond Cheng, Ji Hong, Aapo Kyrola, Youshan Miao, Xuetian Weng, Ming Wu, Fan Yang, Lidong Zhou, Feng Zhao, and Enhong Chen. Kineograph: taking the pulse of a fast-changing and connected world. In *Proceedings of the 7th ACM European Conference on Computer Systems*, EuroSys '12, pages 85–98, New York, NY, USA, 2012. Association for Computing Machinery.

[15] Fernando Chirigati, Rémi Rampin, Dennis Shasha, and Juliana Freire. Reprozip: Computational reproducibility with ease. In *Proceedings of the 2016 International Conference on Management of Data*, SIGMOD '16, pages 2085–2088, New York, NY, USA, 2016. Association for Computing Machinery.

[16] Kenneth Ward Church. Unix for Poets, 1994.

[17] Yann Collet. Zstandard: Fast real-time compression algorithm. https://github.com/facebook/zstd, 2016. Accessed: June 12, 2026.

[18] Ludovic Courtès. Code staging in GNU Guix. In *Proceedings of the 16th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences*, GPCE '17, pages 41–48, New York, NY, USA, 2017. Association for Computing Machinery.

[19] Charlie Curtsinger and Daniel W. Barowy. Riker: Always-Correct and fast incremental builds from simple specifications. In *Proceedings of the 2022 USENIX Annual Technical Conference*, USENIX ATC '22, pages 885–898, Carlsbad, CA, July 2022. USENIX Association.

[20] Datasnaek. Chess Games Dataset, 2020. Accessed: 2025-10-15.

[21] Jeffrey Dean and Sanjay Ghemawat. MapReduce: simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, January 2008.

[22] Deno Land Inc. Deno. https://deno.com/. Accessed: 2026-06-09.

[23] Eelco Dolstra, Merijn de Jonge, and Eelco Visser. Nix: A safe and policy-free system for software deployment. In *Proceedings of the 18th USENIX Conference on System Administration*, LISA '04, pages 79–92, USA, 2004. USENIX Association.

[24] Mokhtar Ebrahim and Andrew Mallett. *Mastering Linux shell scripting*. Packt Publishing, Birmingham, England, 2 edition, April 2023.

[25] Conal Elliott and Paul Hudak. Functional reactive animation. *SIGPLAN Not.*, 32(8):263–273, August 1997.

[26] Free Software Foundation. GNU make. https://www.gnu.org/software/make/make.html, 2023. Accessed: 2025-10-22.

[27] Alexander J. Gaidis, Vaggelis Atlidakis, and Vasileios P. Kemerlis. SysXCHG: Refining Privilege with Adaptive System Call Filters. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*, CCS '23, pages 1964–1978, New York, NY, USA, 2023. Association for Computing Machinery.

[28] Jie Gao, Simret Araya Gebreegziabher, Kenny Tsu Wei Choo, Toby Jia-Jun Li, Simon Tangi Perrault, and Thomas W Malone. A taxonomy for human-LLM interaction modes: An initial exploration. In *Extended Abstracts of the CHI Conference on Human Factors in Computing Systems*, CHI EA '24, New York, NY, USA, 2024. Association for Computing Machinery.

[29] Yuyao Ge, Lingrui Mei, Zenghao Duan, Tianhao Li, Yujia Zheng, Yiwei Wang, Lexin Wang, Jiayu Yao, Tianyu Liu, Yujun

Cai, Baolong Bi, Fangda Guo, Jiafeng Guo, Shenghua Liu, and Xueqi Cheng. A survey of vibe coding with large language models, 2025.

[30] Aurelien Geron. *Hands-on machine learning with scikit-learn, Keras, and TensorFlow*. O'Reilly Media, Sebastopol, CA, 2 edition, October 2019.

[31] Michael Greenberg, Konstantinos Kallas, and Nikos Vasilakis. Unix shell programming: The next 50 years. In *Proceedings of the Workshop on Hot Topics in Operating Systems*, HotOS '21, pages 104–111, New York, NY, USA, 2021. Association for Computing Machinery.

[32] Philip J. Guo and Dawson Engler. CDE: using system call interposition to automatically create portable software packages. In *Proceedings of the 2011 USENIX Annual Technical Conference*, USENIX ATC '11, page 21, USA, 2011. USENIX Association.

[33] Ashish Gupta and Inderpal Singh Mumick. *Maintenance of materialized views: problems, techniques, and applications*, pages 145–157. MIT Press, Cambridge, MA, USA, 1999.

[34] Matthew A. Hammer, Khoo Yit Phang, Michael Hicks, and Jeffrey S. Foster. Adapton: composable, demand-driven incremental computation. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14, pages 156–166, New York, NY, USA, 2014. Association for Computing Machinery.

[35] Michael S. Hart and Project Gutenberg. Project Gutenberg. https://www.gutenberg.org, 1971.

[36] Anna Herlihy, Periklis Chrysogelos, and Anastasia Ailamaki. Boosting Efficiency of External Pipelines by Blurring Application Boundaries. In *Proceedings of the 12th Conference on Innovative Data Systems Research*, CIDR '22. www.cidrdb.org, 2022.

[37] Roger Hoover. Alphonse: incremental computation as a programming abstraction. In *Proceedings of the ACM SIGPLAN 1992 Conference on Programming Language Design and Implementation*, PLDI '92, pages 261–272, New York, NY, USA, 1992. Association for Computing Machinery.

[38] Ben Hoyt and Simon Alford. Fabricate, 2020.

[39] Zhicheng Huang, Ramiz Dundar, Yizheng Xie, Konstantinos Kallas, and Nikos Vasilakis. Fractal: Fault-tolerant shell-script distribution. In *Proceedings of the 23rd USENIX Symposium on Networked Systems Design and Implementation*, NSDI '26, Renton, WA, May 2026. USENIX Association.

[40] Nicholas Hunt, Tom Bergan, Luis Ceze, and Steven D. Gribble. DDOS: taming nondeterminism in distributed systems. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '13, pages 499–508, New York, NY, USA, 2013. Association for Computing Machinery.

[41] Jeroen Janssens. *Data science at the command line*. O'Reilly Media, Sebastopol, CA, October 2014.

[42] Konstantinos Kallas, Tammam Mustafa, Jan Bielak, Dimitris Karnikis, Thurston H.Y. Dang, Michael Greenberg, and Nikos Vasilakis. Practically correct, Just-in-Time shell script parallelization. In *Proceedings of the 16th USENIX Symposium on Operating Systems Design and Implementation*, OSDI '22, pages 769–785, Carlsbad, CA, July 2022. USENIX Association.

[43] Brian W. Kernighan and Rob Pike. *The UNIX Programming Environment*. Prentice Hall Professional Technical Reference, 1984.

[44] Mary Beth Kery, Marissa Radensky, Mahima Arya, Bonnie E. John, and Brad A. Myers. The story in the notebook: Exploratory data science using a literate programming tool. In *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems*, CHI '18, pages 1–11, New York, NY, USA, 2018. Association for Computing Machinery.

[45] Alexander Kirillov, Eric Mintun, Nikhila Ravi, Hanzi Mao, Chloe Rolland, Laura Gustafson, Tete Xiao, Spencer Whitehead, Alexander C. Berg, Wan-Yen Lo, Piotr Dollár, and Ross Girshick. Segment anything, 2023.

[46] Kitware. CMake. https://cmake.org/. Accessed: 2025-12-05.

[47] Thomas Kluyver, Benjamin Ragan-Kelley, Fernando Pérez, Brian Granger, Matthias Bussonnier, Jonathan Frederic, Kyle Kelley, Jessica Hamrick, Jason Grout, Sylvain Corlay, Paul Ivanov, Damián Avila, Safia Abdalla, Carol Willing, and Jupyter Development Team. Jupyter Notebooks—a publishing format for reproducible computational workflows. In *Positioning and Power in Academic Publishing: Players, Agents and Agendas*, pages 87–90. IOS Press, 2016.

[48] Evangelos Lamprou. Foundation Models and Unix, March 2025.

[49] Evangelos Lamprou, Ethan Williams, Georgios Kaoukis, Zhuoxuan Zhang, Michael Greenberg, Konstantinos Kallas, Lukas Lazarek, and Nikos Vasilakis. The Koala Benchmarks for the Shell: Characterization and Implications. In *Proceedings of the 2025 USENIX Annual Technical Conference*, USENIX ATC '25, pages 449–64, Boston, MA, July 2025. USENIX Association.

[50] Evangelos Lamprou, Tianyu (Ezri) Zhu, Di Jin, Grigoris Ntousakis, Georgios Liargkovas, Calvin Eng, Konstantinos Kallas, Michael Greenberg, and Nikos Vasilakis. Controlling opaque-component effects with semisolates and Try. In *Proceedings of the 20th USENIX Symposium on Operating Systems Design and Implementation*, OSDI '26. USENIX Association, 2026.

[51] LangChain. LangChain. https://www.langchain.com/. Accessed: 2026-06-09.

[52] E.A. Lee and D.G. Messerschmitt. Synchronous data flow. *Proceedings of the IEEE*, 75(9):1235–1245, 1987.

[53] Yanhong A. Liu. Incremental computation: What is the essence? (invited contribution). In *Proceedings of the 2024 ACM SIGPLAN International Workshop on Partial Evaluation and Program Manipulation*, PEPM '24, pages 39–52, New York, NY, USA, 2024. Association for Computing Machinery.

[54] Aurèle Mahéo, Pierre Sutra, and Tristan Tarrant. The serverless shell. In *Proceedings of the 22nd International Middleware Conference: Industrial Track*, Middleware '21, pages 9–15, New York, NY, USA, 2021. Association for Computing Machinery.

[55] marimo Developers. marimo. https://marimo.io/. Accessed: 2025-12-01.

[56] Evan Martin. Ninja: A small build system with a focus on speed. https://ninja-build.org/, 2012. Accessed: 2025-02-14.

[57] Bill McCloskey. Memoize: A replacement for make, 2008. Archived: 2010-09-05.

[58] Dirk Merkel. Docker: lightweight Linux containers for consistent development and deployment. *Linux J.*, 2014(239), March 2014.

[59] Andrey Mokhov, Neil Mitchell, Simon Peyton Jones, and Simon Marlow. Non-recursive make considered harmful: build systems at scale. *SIGPLAN Not.*, 51(12):170–181, September 2016.

[60] Jason Morris, Chris McCubbin, and Raymond Page. *Hands-On Data Science with the Command Line*. Packt Publishing, Birmingham, England, January 2019.

[61] Kiran-Kumar Muniswamy-Reddy, David A. Holland, Uri Braun, and Margo Seltzer. Provenance-aware storage systems. In *Proceedings of the 2006 USENIX Annual Technical Conference*, USENIX ATC '06, page 4, USA, 2006. USENIX Association.

[62] Derek G. Murray, Frank McSherry, Rebecca Isaacs, Michael Isard, Paul Barham, and Martín Abadi. Naiad: a timely dataflow system. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 439–455, New York, NY, USA, 2013. Association for Computing Machinery.

[63] Tammam Mustafa, Konstantinos Kallas, Pratyush Das, and Nikos Vasilakis. DiSh: Dynamic Shell-Script distribution. In *Proceedings of the 20th USENIX Symposium on Networked Systems Design and Implementation*, NSDI '23, pages 341–356, Boston, MA, April 2023. USENIX Association.

[64] National Oceanic and Atmospheric Administration. National Oceanic and Atmospheric Administration (NOAA). https://www.noaa.gov. Accessed: 2025-01-13.

[65] Omar S. Navarro Leija, Kelly Shiptoski, Ryan G. Scott, Baojun Wang, Nicholas Renner, Ryan R. Newton, and Joseph Devietti. Reproducible containers. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '20, pages 167–182, New York, NY, USA, 2020. Association for Computing Machinery.

[66] Evi Nemeth, Garth Snyder, Trent R Hein, Ben Whaley, and Dan Mackin. *UNIX and Linux System Administration Handbook*. Addison-Wesley Educational, Boston, MA, 5 edition, August 2017.

[67] Addy Osmani. *Beyond Vibe Coding: From Coder to AI-Era Developer*. O'Reilly Media, 2025.

[68] Thomas Pasquier, Xueyuan Han, Mark Goldstein, Thomas Moyer, David Eyers, Margo Seltzer, and Jean Bacon. Practical whole-system provenance capture. In *Proceedings of the 2017 Symposium on Cloud Computing*, SoCC '17, pages 405–418, New York, NY, USA, 2017. Association for Computing Machinery.

[69] Daniel Peng and Frank Dabek. Large-scale incremental processing using distributed transactions and notifications. In *Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation*, OSDI '10, pages 251–264, USA, 2010. USENIX Association.

[70] Jon Puritz. Bio594: Using genomic techniques to examine the evolution of populations. https://git.io/JY6J7, 2019.

[71] Deepti Raghavan, Sadjad Fouladi, Philip Levis, and Matei Zaharia. POSH: a data-aware shell. In *Proceedings of the 2020 USENIX Annual Technical Conference*, USENIX ATC '20, USA, 2020. USENIX Association.

[72] Michael Schröder and Jürgen Cito. An empirical investigation of command-line customization. *Empirical Software Engineering*, 27(2):30, 2021.

[73] Jiasi Shen, Martin Rinard, and Nikos Vasilakis. Automatic Synthesis of Parallel Unix Commands and Pipelines with KumQuat. In *Proceedings of the 27th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '22, pages 431–432, New York, NY, USA, 2022. Association for Computing Machinery.

[74] Benjamin H Sigelman, Luiz André Barroso, Mike Burrows, Pat Stephenson, Manoj Plakal, Donald Beaver, Saul Jaspan, and Chandan Shanbhag. Dapper, a large-scale distributed systems tracing infrastructure. Technical report, Google, Inc., 2010.

[75] Sarah Spall, Neil Mitchell, and Sam Tobin-Hochstadt. Build scripts with perfect dependencies. *Proc. ACM Program. Lang.*, 4(OOPSLA), November 2020.

[76] Diomidis Spinellis. DelftX: Unix Tools: Data, Software and Production Engineering. https://www.edx.org/learn/unix/delft-university-of-technology-unix-tools-data-software-and-production-engineering. Accessed: 2025-01-13.

[77] Diomidis Spinellis. Outwit:{UNIX}{Tool-Based} programming meets the Windows world. In *Proceedings of the 2000 USENIX Annual Technical Conference*, USENIX ATC '00, 2000.

[78] Diomidis Spinellis and Marios Fragkoulis. Extending Unix Pipelines to DAGs. *IEEE Transactions on Computers*, 66(9):1547–1561, 2017.

[79] Diomidis Spinellis and Georgios Gousios. How to analyze git repositories with command line tools: we're not in Kansas anymore. In *Proceedings of the 40th International Conference on Software Engineering: Companion Proceedings*, ICSE '18, pages 540–541, New York, NY, USA, 2018. Association for Computing Machinery.

[80] Eleftheria Tsaliki and Diomidis Spinellis. The Real Numbers for Athens Buses, 2020.

[81] Justine Tunney. Bash One-Liners for LLMs. https://justine.lol/oneliners, 2023. Accessed: 2025-06-01.

[82] Unix Game. The Unix Game - 50 Challenges to Master the Command Line, 2024. Accessed: 2025-10-19.

[83] Jake VanderPlas. *Python Data Science Handbook*. O'Reilly Media, Sebastopol, CA, December 2016.

[84] Nikos Vasilakis, Konstantinos Kallas, Konstantinos Mamouras, Achilles Benetopoulos, and Lazar Cvetković. Pash: Light-touch data-parallel shell processing. In *Proceedings of the 16th European Conference on Computer Systems*, EuroSys '21, pages 49–66, New York, NY, USA, 2021. Association for Computing Machinery.

[85] Nikos Vasilakis, Ben Karel, Nick Roessler, Nathan Dautenhahn, André DeHon, and Jonathan M. Smith. Towards fine-grained, automated application compartmentalization. In *Proceedings of the 9th Workshop on Programming Languages and Operating Systems*, PLOS '17, pages 43–50, New York, NY, USA, 2017. Association for Computing Machinery.

[86] VizierDB Developers. VizierDB. https://vizierdb.info/. Accessed: 2025-12-01.

[87] Tom White. *Hadoop: The Definitive Guide*. O'Reilly Media, Inc, 2009.

[88] Keith Winstein and Hari Balakrishnan. Mosh: an interactive remote shell for mobile clients. In *Proceedings of the 2012 USENIX Annual Technical Conference*, USENIX ATC '12, page 15, USA, 2012. USENIX Association.

[89] Wen Xia, Yukun Zhou, Hong Jiang, Dan Feng, Yu Hua, Yuchong Hu, Qing Liu, and Yucheng Zhang. FastCDC: A fast and efficient Content-Defined chunking approach for data deduplication. In *Proceedings of the 2016 USENIX Annual Technical Conference*, USENIX ATC '16, pages 101–114, Denver, CO, USA, June 2016. USENIX Association.

[90] Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica. Spark: cluster computing with working sets. In *Proceedings of the 2nd USENIX Workshop on Hot Topics in Cloud Computing*, HotCloud '10, page 10, USA, 2010. USENIX Association.

[91] Matei Zaharia, Tathagata Das, Haoyuan Li, Timothy Hunter, Scott Shenker, and Ion Stoica. Discretized streams: fault-tolerant streaming computation at scale. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 423–438, New York, NY, USA, 2013. Association for Computing Machinery.

[92] Megan Zheng, Will Crichton, Akshay Narayan, Deepti Raghavan, and Nikos Vasilakis. When are reactive notebooks not reactive?, 2025.

A Software Artifact

Abstract

The software artifact that accompanies this paper includes INCR, its documentation, the behavioral-equivalence test suite, and the scripts used to reproduce the evaluation in §8. It supports three goals: inspecting the implementation and its accompanying materials, exercising the system through its test suite, and reproducing the paper's main evaluation results, including the benchmark-suite performance summary and the behavioral equivalence checks against Bash.

Scope

The artifact covers the paper's main contributions. In particular, it includes: (1) the INCR implementation and runtime mechanisms described throughout the technical sections (§4–§7); (2) the benchmark suite and supporting workloads used in §8; and (3) the automation used to reproduce the key evaluation results in §8, including re-execution performance and behavioral equivalence.

Contents

The artifact contains the following components:

- the INCR implementation itself, including the runtime, effect tracking, memoization, and annotation machinery;
- documentation, including a top-level README and artifact instructions;
- a behavioral-equivalence harness based on the standard Bash test suite;
- the benchmark drivers and supporting utilities for the evaluation workloads; and
- scripts for running the evaluation and generating the key result artifacts.

Hosting

The artifact is publicly available both as a GitHub repository and as a Zenodo archival snapshot. The GitHub version is hosted at https://github.com/atlas-brown/incr, and the archival version is hosted at https://zenodo.org/records/19488802.

ARTIFACT EVALUATED
USENIX
AVAILABLE

A Compilation-based Under-Constrained Execution Engine

Mingjun Yin Zhaorui Li Ju Chen Haochen Zeng Chengyu Song
University of California, Riverside

Abstract

Software bugs continue to pose significant challenges to the security and correctness of computer systems. Finding and eliminating bugs for large-scale software systems, such as the Linux kernel, remains a difficult task. Static analyses can cover the whole codebase, but often produce too many false positives. Whole program dynamic testing is precise but has limited code coverage, and could require special environments. Due to the modular design of large software systems, a promising alternative is to instantiate an execution environment for individual components in isolation, and then apply precise dynamic analyses to these components. Unfortunately, existing execution engines that support such under-constrained execution are all interpreter-based, thus suffering from poor scalability. In this paper, we introduce UCSan, a compilation-based under-constrained execution engine that can compile an arbitrary set of C/C++ functions into a self-contained executable without manual modifications. To demonstrate the scalability and versatility of UCSan, we showcase combining UCSan with a compilation-based concolic execution engine to conduct under-constrained symbolic execution. Our evaluation shows that the resulting analysis engine is up to 15.06x faster on Linux kernel analysis tasks than the KLEE-based engine. This enhanced scalability not only improves the bug detection effectiveness but also enables its application across a broader range of software systems.

1 Introduction

Software bugs remain the major threat to the correctness and security of computer systems. This problem is especially severe for operating system kernels, which typically serve as the trusted computing base (TCB) for the entire system. Once a security vulnerability in the kernel is exploited, it usually can lead to the compromise of the whole system. Therefore, it is critical to identify and fix bugs before they are exploited by attackers.

Dynamic testing techniques, such as gray-box fuzzing [13, 26, 37], are state-of-the-art bug-finding techniques that have been widely adopted in the industry and have discovered a substantial number of bugs: the OSS-Fuzz [16] project has found more than 10,000 security vulnerabilities and 36,000 bugs in 1,000 open source projects; and the continuous kernel fuzzer syzbot [38] has helped fix over 6,500 bugs in the Linux kernel. Although fuzzers have been very successful in finding bugs, they also have notable limitations. One fundamental limitation, which applies not only to fuzzing, but also to all other dynamic analysis techniques, is the need for good *testing harnesses*—if a bug is not reachable from the harness, it cannot be found by fuzzers, no matter how much effort is put into the fuzzing.

Besides the testing harness challenge, another fundamental challenge for dynamic testing is the environmental setup. For example, to test an OS kernel, a fuzzer typically needs to set up a virtual machine; to test a file system, the fuzzer needs to prepare a disk image; and to test a device driver, the fuzzer either needs direct access to the real hardware, or a proper device emulator, which is often unavailable and non-trivial to implement. As a result, not all components in a complex software system can be effectively tested (e.g., an Android image for real phones).

Another widely used technique to find bugs is static analysis. While static analyses are not limited by testing harnesses and can analyze arbitrary sets of code (e.g. the whole kernel), they need to trade off precision for scalability. For large and complex projects like OS kernels, static analyzers tend to generate too many false alarms [2, 10, 12, 21, 22, 25, 46]. For example, the use-before-initialization analyzer [46] used in our evaluation generates 147,643 warnings, but only 52 were confirmed or patched.

Conceptually, it would be ideal if we could extract an arbitrary subset of code from the target software, make it self-contained, and then apply dynamic testing techniques to it. In this way, we can avoid the need for a testing harness and complex environmental setup, while still being able to leverage the precision of dynamic analyses. This approach faces two key challenges. The first challenge is how to initialize the memory properly. A program typically accesses three types

of memory objects: global, stack, and heap. Global and stack objects are relatively easy to handle, as existing compiler toolchains already know how to allocate and initialize them. However, heap objects are usually allocated and initialized dynamically at runtime. When we skip the typical initialization process (e.g., booting the kernel) and start the execution from an arbitrary (internal) function, we need a way to allocate and initialize heap objects expected by the code; otherwise, the execution will likely crash due to invalid memory accesses.

The second challenge is how to handle external dependencies. When we extract an arbitrary set of code from a complex software system, it is likely that the code will invoke functions outside the analysis scope, and we usually want to handle them differently. In most cases, we do not want to actually execute these external functions, otherwise we should have included them in the analysis scope in the first place. In other cases, we may want to provide a simplified model or a replacement. For instance, when running kernel code in the user space, we may want to replace invocations of `kmalloc` with `malloc`.

Unarguably, one can address these challenges manually. For example, researchers have manually extracted the TCP stack from the Linux and FreeBSD kernels for model checking [8]. Similarly, the Linux kernel library (LKL) project [1] aims to allow running Linux kernel code in user space, and has been used to build a fuzzing framework for file systems [43]. However, manual approaches are typically labor-intensive, error-prone, hard to scale (e.g., extract another module), and costly to maintain (e.g., keep updating with the upstream codebase).

Under-constrained execution, as pioneered by UC-KLEE [30], provides an automated approach to the above challenges. Under the hood, UC-KLEE employs a technique called *lazy initialization* to address the memory initialization challenge—when a pointer is dereferenced for the first time, if it points to an uninitialized memory object, the engine will allocate and initialize the object on-the-fly. This way, the engine can avoid the need to fully initialize the memory layout beforehand. To address the external dependency challenge, UC-KLEE provides a set of interfaces to allow users to specify how to handle external function invocations.

Besides UC-KLEE, some recently developed engines can also support under-constrained execution, such as Angr [34]. Unfortunately, these engines are all interpreter-based. As a result, they suffer from severe scalability issues. Yun et al. [44] reported that KLEE is around 3,000 times slower than native execution, and Angr is more than 321,000 times slower. Such performance is insufficient for analyzing slightly larger components like file systems, TCP stack, and device drivers.

Inspired by the recent advances in compilation-based concolic execution engines [9, 28], we propose UCSAN (Under-Constrained Sanitizer), a compilation-based under-constrained execution engine implemented as an LLVM Pass along with a runtime library.

UCSAN has two primary design goals. First, we decouple under-constrained execution from symbolic execution, enabling UCSAN to be combined with a variety of dynamic analysis techniques, including concolic execution, fuzzing, and model checking. Second, as a compilation-based approach, UCSAN is designed to be general-purpose, easy to integrate, and performant.

UCSAN addresses the aforementioned two challenges through compile-time transformations and runtime support. Specifically, we implemented a prototype of UCSAN based on the sanitizer framework of LLVM. It contains an instrumentation pass that (1) instruments all memory accesses to perform runtime address translation to support lazy initialization (i.e., accessing memory through uninitialized pointers), (2) instruments external function invocations based on configuration, and (3) removes all out-of-scope code. UCSAN also contains a runtime library that uses the shadow memory from the sanitizer framework to track aliased pointers, such that it knows: (1) which pointers have been initialized (i.e., have corresponding memory objects), (2) which pointers point to the same memory object, and (3) how to translate an under-constrained pointer to the proper offset within its corresponding memory object.

We then evaluated the performance and compatibility of UCSAN. First, we compared UCSAN with KLEE and Angr on the `nbench` benchmark [27], the results show that UCSAN is orders of magnitude faster. For compatibility, we compiled three versions of Linux kernel (4.14, 5.10, and 6.16) with UCSAN under different configurations. The results show that UCSAN can successfully compile over 80% of the kernel modules.

To demonstrate how UCSAN can benefit bug finding, we additionally developed **Thoroupy**, a Python-based analysis companion that orchestrates exploration and generates input seeds for UCSAN. Together they form a complete concolic execution engine we call UCSAN†. While UCSAN is independently usable, for evaluation purposes, we evaluate it in the UCSAN† setup in this paper and use the two names interchangeably. We conducted a few case studies with UCSAN†. First, we compared UCSAN† with a KLEE-based engine [47] on the task of filtering warnings generated by static analysis of UBITect [46]. UCSAN† can process 63,957 (95.46%) warnings under the same 2 minute timeout, which is 42,155 more than the KLEE-based engine. Such improvement allows UCSAN† to confirm at least 15 more patched vulnerabilities. Second, we applied UCSAN† to reproduce over a hundred known vulnerabilities in user-space open-source projects and the Linux kernel. The results show that UCSAN† can successfully reproduce a significant portion of them, without any manual effort to build test harnesses.

In summary, this paper contributes to the field in the following ways:

- **Novel design**: We present the design of a novel, compilation-based under-constrained execution solution

UCSAN, which is general, easy to use, performant, and can be combined with different dynamic analysis techniques.
- **Bug finding**: To demonstrate how the improved scalability can benefit dynamic analysis, we built an under-constrained concolic execution engine UCSAN† based on UCSAN, and evaluated it on multiple real-world case studies. The evaluation results show that UCSAN† can scale to large kernel modules and find real bugs.
- **Open-source**: UCSAN and UCSAN† are publicly accessible under the Apache 2.0 license.[1]

2 Under-constrained (Concrete) Execution

In this section, we provide an overview of compilation-based under-constrained execution engine, including workflow and key challenges.

2.1 Workflow

Figure 1 illustrates the workflow of a compilation-based under-constrained execution engine. It takes as inputs (1) the target Program Under Test (PUT) in the form of LLVM Intermediate Representation (IR), and (2) a YAML configuration file specifying the analysis scope (i.e. which functions and IR files to include), the entry function (i.e. where to start execution), and handling strategies for out-of-scope functions. The output is a self-contained executable that can run as a normal user-space program. UCSAN imposes no restriction on how the LLVM IR is produced, that is, any toolchain that emits LLVM IR can serve as the front end. For instance, enabling link-time optimization (LTO) in the Linux kernel build system is an easy way to obtain IR for the kernel.

The configuration file (e.g., Figure 2) is the key to remove the dependency on test harnesses, as one can specify any function (e.g., `cal`) as the entry point, and an arbitrary set of functions (e.g., `foo, bar`) as the analysis scope. UCSAN also supports mapping a function in the PUT to a replacement function (e.g., `kmalloc` to `malloc`). The configuration file can be generated manually. Alternatively, it can also be generated automatically using static analysis [30, 41, 46, 47], or using large language models (LLMs). In the evaluation, we used configurations generated by a static security checker UBITect [46], and a simple call graph analysis. For bug reproduction experiments, some configurations were generated manually, some were generated by LLMs.

Using the analysis scope specified in the configuration file, UCSAN first iterates over all included IR files and invokes `opt` from the LLVM toolchain to perform its instrumentation and output instrumented IR files. During this step, UCSAN will also remove out-of-scope functions, and replace their invocations with proper stubs, or a replacement function as specified in the configuration file (e.g. `kmalloc`). Optionally, the instrumented files can then be pipelined into additional dynamic analysis passes. For instance, in the evaluation, we pipelined UCSAN with SymSan [9] to perform under-constrained concolic execution, namely, UCSAN†. These dynamic analysis passes are optional, without them, the output binary can still run normally. Finally, UCSAN compiles the instrumented IR into object files and links them with all necessary runtime libraries (PUT's, UCSAN's, and any from the dynamic analysis passes) to produce a self-contained user-space executable.

The instrumented binary can then be used for dynamic analysis, but there are a few technical challenges to address. First, most internal functions consume memory objects instead of files or network I/O. In other words, dynamic testing should be done by providing different memory objects as inputs. To achieve this, UCSAN defines an input format to serialize memory objects into a file, and later deserialize them back to memory objects during execution. The second challenge is how to support lazy initialization, that is, when we start the execution from an arbitrary (internal) function, heap memory objects that would be accessed are neither allocated nor initialized yet. Similarly, all pointers used to access these memory objects are uninitialized as well. Therefore, we need a way to (1) allocate and initialize memory objects on-demand when they are accessed for the first time, and (2) bind uninitialized pointers to the proper memory objects. Related challenges include but are not limited to, **(C1)** how to handle aliased pointers, **(C2)** how to estimate the size of a memory object, **(C3)** how to adjust the size of a memory object dynamically in case the estimation is wrong, and **(C4)** how to deserialize memory objects from the input file to ensure deterministic execution. In this work, we address all these challenges with a new technique called *Just-In-Time Initialization* (§3.2).

As mentioned earlier, one primary design goal of UCSAN is to decouple under-constrained execution from specific dynamic analysis techniques, so that it can be combined with different dynamic analysis techniques. Therefore, we consider generating new test inputs (i.e., memory objects) as an orthogonal problem. For demonstration purposes, we showcase how to combine UCSAN with a compilation-based concolic execution engine SymSan [9] to perform under-constrained concolic execution UCSAN†. In a nutshell, SymSan performs compile-time instrumentation to generate symbolic traces as dynamic analysis outputs. It then parses the traces into symbolic path constraints, and uses an SMT solver to solve the constraints and generate new test inputs (e.g., memory objects that can negate a branch). The generated inputs can then be executed to explore new paths.

2.2 An Illustrative Example

In this subsection, we use an elevated linked list example as in [30] to explain the key steps in under-constrained (concrete) execution.

[1] Available at https://github.com/R-Fuzz/UCSan

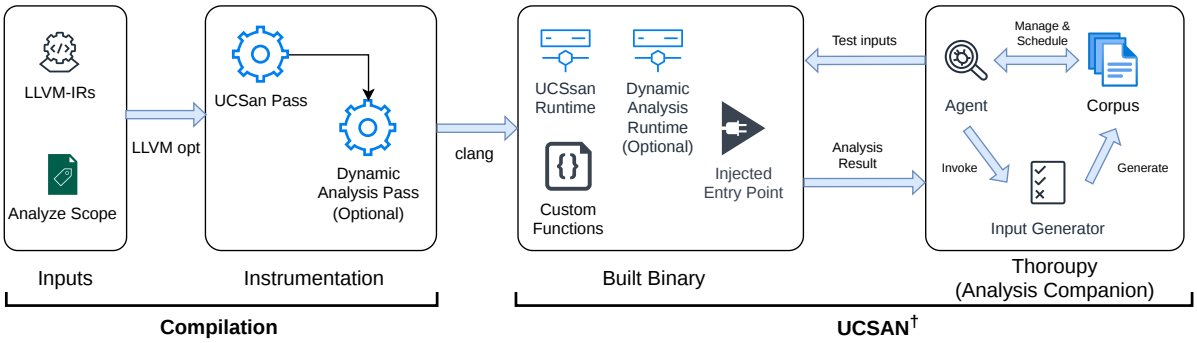


Figure 1: Workflow of a compilation-based under-constrained execution engine, and how it can be combined with different dynamic analysis techniques.

```
entry: cal
scope: [foo, bar]
custom:
  kmalloc:
    ref_name: malloc
    arg_maps: # map first arg and ignore rest
      - idx: 0
        ucsan_idx: 0
```

Figure 2: An example configuration file.

Consider the code snippet in Figure 3, which defines a simple linked list and a function `cal` that computes the sum of all node values. Traditional dynamic testing tools [6, 9, 11, 19, 28, 37, 45] cannot analyze `cal` directly. Instead, they require a testing harness. The testing harness is in charge of reading test inputs (e.g., from a file), instantiating a linked list, and invoking the target function. By generating different test inputs, we hope the execution can trigger the assertion failure at line 20.

In a nutshell, what UCSAN does is to automate the process via compiler techniques. Given `cal` as the entry function, UCSAN will automatically generate a harness (`main`) to invoke it. UCSAN will also instrument the code to instantiate a linked list on demand during the execution, and make sure all accesses through aliased pointers will visit the same memory object (addresses). If a test input is provided, UCSAN will also perform deserialization to populate the linked list accordingly. Next, we explain the key steps in detail.

2.2.1 Code Transformation

UCSAN directly executes `cal` from line 8, and initializes the environment on demand (§3.1). This is done by compile-time code transformations, which are annotated with + and - in Figure 3. Note that the actual transformation is done on the LLVM IR level, here we illustrate it in source code level for better readability. These transformations include three main parts:

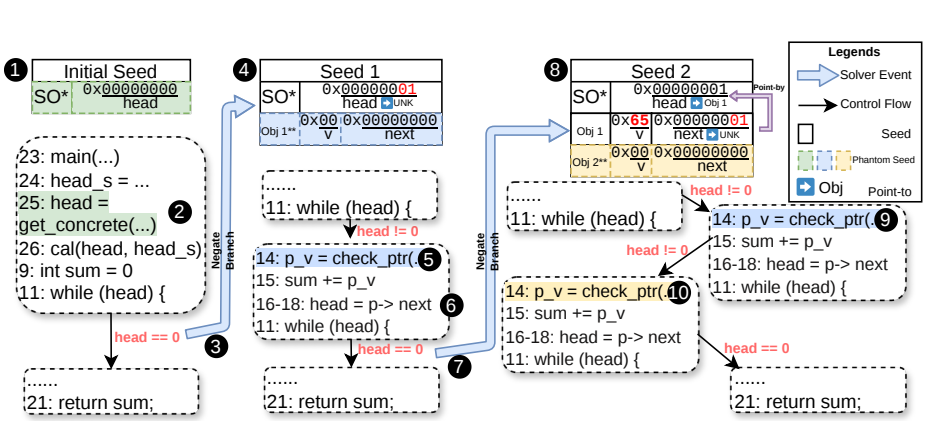
• *Entry point (*`main` *function) generation*. UCSAN will automatically insert a `main` function (lines 23 - 27) to create a self-contained binary (§3.1). The `main` function performs three universal tasks independent of the analyzed function(s): (1) UCSAN tracks aliased pointers using a shadow pointer

```
1    typedef struct {
2      struct list_entry* next;
3    } list_t;
4    typedef struct {
5      unsigned int v;
6      list_t list;
7    } node_t;
   - int cal(list_t* head){
8  + int cal(list_t* head, shadow_t head_s){
9      int sum = 0;
10     node_t* curr = NULL;
11     while (head) {
12 +     check_ptr(head, head_s);
13      curr = container_of(head, node_t, list);
14 +    int *p_v = check_ptr(&curr->v, head_s);
15 +    sum += *p_v;
   -    sum += curr->v;
16 +    list_t** p_next=check_ptr(&head->next,head_s);
17 +    head = *p_next;
18 +    head_s = get_shadow(p_next);
   -    head = head->next;
19   }
20   if (sum > 100) assert(false); /*bug here*/
21   return sum;
22 }
23 + int main(int argc, char** argv) {
24 +   shadow_t head_s = get_shadow_ptr();
25 +   list_t *head = get_concrete(head_s, sizeof(list_t)
       );
26 +   cal(head, head_s);
27 + }
```

Figure 3: Motivating example: linked list sum

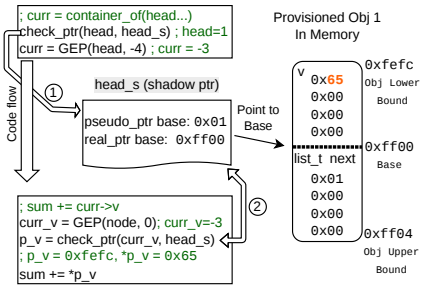
(i.e., all aliased pointers share the same shadow pointer). So the first task is to create a shadow pointer for each pointer argument. In this example, since the argument `head` of `cal` is a pointer, UCSAN creates a shadow pointer `head_s` at line 24. (2) If a test input is provided, UCSAN will deserialize it to populate `head` (line 25). Note that here UCSAN only populates a single list node pointer `head`, the actual node objects will be allocated and populated on-demand when they are accessed. (3) Call target entry function `cal` (line 26).

• *Pointer translation*. As mentioned before, one of the main challenges of under-constrained execution is to allocate memory objects on-demand, and initialize pointers properly, especially when there are aliased pointers. UCSAN solves this through shadow pointers. Conceptually, shadow pointers play a similar role as segmentation registers in virtual address translation: (1) all uninitialized/unbounded pointers will have a shadow pointer associated with them; and (2) all aliased point-



* SO is an abbreviation for Super Object, which stores global objects and initial argument(s) to the entry function. ** SO in Seed 0, Obj 1 in Seed 1, and Obj 2 in Seed 2 are dynamically introduced via JIT-initialization as a phantom seed.

Figure 4: Key steps in exploring different paths in the linklist example.



① The first `check_ptr` invocation creates its shadow pointer, records the pseudo-pointer base, and allocates Obj 1 from Seed 2.
② The second `check_ptr` leverages this data to return the translated real pointer: `curr_v - pseudo_ptr_base + real_ptr_base`.

Figure 5: Shadow pointer handling and real-pointer translation when dealing with Obj 1 in Seed 2

ers will share the same shadow pointer. In this example, the shadow pointer of `head` is `head_s`. And the three aliased pointers `curr`, `curr->v`, and `head->next` also share the same shadow pointer `head_s`. Before *every* memory access, UCSAN will insert a call to the callback function `check_ptr` to translate a potential uninitialized pointer to a valid pointer (line 14 and 16); and replace the use site with the actual pointer (line 15 and 17). Note that because we may need to reallocate the underlying object to adjust the object size on-the-fly (e.g., due to `container_of`), all translated pointers (`p_v` and `p_next`) are treated as "volatile", so `check_ptr` is called twice in this example, even though the pointers (`curr->v`, and `head->next`) are aliasing. For the same reason, we do not store translated pointer back to memory. We explain more about this design choice in §3.2. Finally, whenever a new pointer is loaded from the memory (e.g., line 17), UCSAN will also load the corresponding shadow pointer (line 18).

• *External function handling*. To ensure that the target program is compilable, we need to make sure that all calls to external functions (i.e., functions outside the analysis scope that are not specified in the YAML configuration) can be handled automatically. Note that the example does not contain external functions for brevity. UCSAN offers three options. First, a custom wrapper can be provided to summarize the effects (e.g., `memcpy`). The user can implement the wrapper function in an arbitrary way to simulate the actual behavior of the external function. UCSAN also allows mapping a function in the PUT to a known wrapper function (e.g., `kmalloc` to `malloc`). Second, UCSAN can assume that the function is *pure* and no changes would be made to the current memory state, which is similar to UC-KLEE [30]. Third, UCSAN can assume that the function can perform arbitrary changes to memory objects passed to the function as pointer arguments, and return arbitrary values. For instance, a call of `read(fd,buf)` may fill `buf` with arbitrary values. In this case, UCSAN will reallocate all memory objects passed to the function (i.e. put a new object corresponding to `buf_s`), as under-constrained memory objects.

2.2.2 Path Exploration

Figure 4 demonstrates key steps of UCSAN† in exploring three different paths in the example (Figure 3) by combining UCSAN with a dynamic testing technique. The goal is to cover all paths that can trigger the assertion failure at line 20. In the figure, we use *Seed 0 - 2* to represent three different test inputs and their corresponding execution paths.
• *Seed 0*. ❶ UCSAN executes the compiled binary with an initial seed. This initial seed is empty, i.e., it contains nothing at all. ❷ When the execution reaches line 25 and `get_concrete` is called, UCSAN looks up the seed for an object matching `head`. Because no such entry exists, UCSAN treats the object as a *phantom*: it allocates a new memory object based on the type information of `head` (i.e., a pointer) and provisions it with all zeros. In general, whenever the seed does not contain an entry for a requested object, UCSAN pretends the object exists and initializes it with zeros—we call this a *phantom seed*. We use colored backgrounds to annotate phantom objects in the figure. `get_concrete` then returns `nullptr(0x00)` to initialize the `head` pointer. This new object resides in a memory segment we call *super object* (SO). We use this segment to store global objects and initial arguments for the entry function. When the execution reaches line 11, since `head` is initialized as a `nullptr`, the concrete execution goes to the *false* branch, reaches line 20, does not trigger the assertion failure, returns to `main`, and terminates.

To explore a different path, UCSAN needs a new seed. To clarify again, this step (input generation) is done by the integrated dynamic testing technique, namely UCSAN†. If

we are performing under-constrained symbolic execution, we can solve the negated path constraint `head != null` (❸), by assigning `head` to `0x1`; and generate a new seed. Note that here the engine does not have to assign `head` to a valid address, just a non-null value (i.e., it should point to something, but exactly where, can be determined later). Since the values of `head` and other pointers in the original source code do not directly address real objects, and only their shadow pointers hold references to the actual backing memory, we refer to them as *pseudo-pointers*.

• *Seed 1*. ❹ UCSAN now has the newly generated *Seed 1*. When the execution reaches line 11, because the seed is not empty, UCSAN uses the shadow pointer `head_s` to locate the corresponding content in the seed, and populates values for the allocated memory object accordingly (i.e., `head = 0x1`). Because `head` now satisfies the predicate `head != null`, the concrete execution can enter the loop body and try to read from the `curr->v` pointer. ❺ When `check_ptr` is called, UCSAN lazily allocates a new (phantom) memory object `obj1`. In this example, we highlight three key challenges in lazy initialization in concrete execution. First, how to know the size of `obj1`? If we simply use the type information of `head` in the declaration of `cal`, then the size would be insufficient, as the type of it is just an infrastructure for linked list. To solve this problem, UCSAN instruments pointer arithmetic operations (e.g., `container_of`) and type casting operations to accurately track the allocation size of the underlying object. Note we omit the size tracking instrumentation in the example for brevity. In this case, the size of `obj1` is estimated based on the type information of `curr` (i.e., `sizeof(node_t)`), instead of the type of `head` (i.e., `sizeof(list_t)`). The concrete content of `obj1` is initialized as all zeros.

The second challenge is how to calculate the correct offset within the underlying object. UCSAN solves this problem using the segment register: shadow pointer `head_s`. In particular, when materializing `obj1`, UCSAN updates `head_s` to record the concrete value of the under-constrained pointer, the logical base of the segment. In this example, the pointer `curr->v`'s value is `-3`, due to `container_of` (`1-4=-3`). This allows UCSAN to handle arbitrary pointer arithmetic and type casting operations, as well as growing the segment on-the-fly.

The third challenge is how to locate the content in the seed to initialize `obj1`. UCSAN solves this problem by recording *object relationships*. Specifically, UCSAN records in the shadow pointer `head_s` that `obj1` is *pointed-by* the pointer at offset `0` from `S0`; so next iteration, it can find the content from the seed to initialize `obj1`. After materializing `obj1`, `check_ptr` returns `obj1.v` (offset `0` from `obj1`) to initialize `p_v`. Again, `p_v` is ephemeral: it is only used for the memory read, then discarded immediately after the read and never used again. We call such pointers *real-pointers*.

❻ A pseudo-pointer is loaded from the lazily allocated object `obj1.next`, and its corresponding shadow pointer is created. Note that in this step, `head->next` has a concrete value of `0x1`, but UCSAN can correctly translate it to `obj1.list.next` (offset `0x4`), because `head_s` has recorded that the base pointer is `-3`. ❼ The concrete execution reaches the loop head again. Because `obj1.list.next` is initialized as a `nullptr`, the loop exits.

In this execution, `sum` still has a concrete value of `0`, so the assertion failure is not triggered. However, it is now a symbolic value (`0+obj1.v`). By solving the negated path constraint `((0+obj1.v)>100)==true`, we can generate a new seed that triggers the assertion failure.

To explore a different path, the dynamic analyzer generates a new seed. This can be done by assigning `obj1.list.next` to `0x1`, i.e., *Seed 2*. Notably, `obj1.list.next` does not have to be a valid pointer, and can have the same value as `head` (i.e., same logical address of different segments do not conflict). For simplicity, we also assign `obj1.v` to `0x65` (decimal 101) to trigger the assertion failure.

• *Seed 2*. ❽ UCSAN schedules Seed 2. This seed includes two serialized objects: SO and `obj1`. The `head` field in SO is initialized to `0x1`, and the `obj1.list.next` field is also set to `0x1`. As with Seed 1, the execution enters the loop body. ❾ When `check_ptr` is called, UCSAN lazily allocates `obj1`. However, this time, it loads the content from the seed, based on the recorded pointed-by annotation from ❺. As a result, the `obj1.list.next` field is now initialized to `0x1`. This causes the execution to re-enter the loop body. ❿ When `check_ptr` is called again, UCSAN allocates a new (phantom) memory object, `obj2`, initializes its content to zero, and records the point-to relationship (i.e. pointer defined at offset `4` from `obj1` points to `obj2`). Since `obj2.list.next` is initialized to `null`, the loop terminates. When the execution reaches line 20, the concrete value of `sum` is `101`, so the assertion failure is triggered, and all paths are covered.

3 UCSAN

In this section, we present technical details of UCSAN. UCSAN is built on the sanitizer framework from LLVM, reusing the shadow memory design of DataFlowSanitizer (DFSan) [35], to track shadow pointers and function calls (arguments and return values).

3.1 Compile-time Instrumentation

This instrumentation performs three main tasks, as briefly illustrated in §2.2: (1) `main` function generation, (2) pointer translation, (3) handling external functions.

Preprocessing. UCSAN operates on LLVM IR, so any build system that produces IR can serve as its front end.

Harness Generation. As discussed in §2, UCSAN inserts a `main` function to produce a self-contained binary. It first discards the original `main` (if present), then, if the entry function (e.g., `cal` in Figure 2, 3) is defined in the current module, in-

troduces a new `main` there. As shown in Figure 3, this `main` is a wrapper that populates arguments (lines 24–25, deserializing from the super object if available, otherwise all zeros) and calls the entry.

Pointer Translation. UCSAN refers to all PUT-managed pointers as *Pseudo-Pointers* (pseudo-pointers), which are translated into valid pointers (*Real-Pointers*, real-pointers) only upon dereference. We call this process *Just-In-Time Initialization* (JITI), which is conceptually similar to segmentation-based logical-to-linear address translation in x86 processors, but implemented in software. Unlike classical lazy initialization [30], which allocates objects only when they are about to be accessed, JITI may provision objects eagerly (e.g., during address computation), hence the distinct name. A key invariant of JITI is that real-pointers must never escape into the PUT's own data flow: they are handed only to memory-access instructions (e.g., `Load`) and discarded immediately afterward. The PUT itself carries only pseudo-pointers or constant pointers (e.g., globals in the data/bss segment). UCSAN handles constant pointers specially by having their pseudo-pointer and real-pointer values coincide, so that they can coexist with pseudo-pointers transparently.

To enforce this invariant, UCSAN instruments all LLVM IR instructions and intrinsics that involve pointer operations, including `Load`, `Store`, `AtomicRMW`, `memset`, `memcpy`, `memmove`, `GEP`, and `BitCast`. Before each such instruction, UCSAN inserts a call to the runtime function `check_ptr`. For dereferencing instructions (e.g., `Load`, `Store`, `memcpy`, line 15 in Figure 3), the original pointer operand is replaced with the real-pointer returned by `check_ptr`, ensuring the access targets a valid, allocated memory object. For address-computing instructions (e.g., `GEP`, `BitCast`, line 13 in Figure 3), UCSAN notes the expected object size and updates the pointer arithmetic metadata accordingly. The runtime logic of `check_ptr` is detailed in §3.2.

Shadow Pointers. To perform pointer translation, UCSAN uses *shadow pointers* to track three pieces of information: (1) the logical base address used to materialize the underlying memory object, (2) which memory object a pseudo-pointer points to (**point-to**), and (3) where a pseudo-pointer originates (**point-by**). Shadow pointers propagate alongside the data flow of pseudo-pointers. To implement this, UCSAN builds upon DFSan's shadow memory design. This design associates each byte of application memory with a shadow value. It also provides local shadow variables for fast access, and defines a calling convention to pass shadow values through function arguments and return values. UCSAN also instruments casts between integer and pointer types. A shadow pointer is introduced whenever a pseudo-pointer is created. For instance, loading a pointer from a JITI-ed object produces a fresh shadow pointer.

External Function Handling. To satisfy the linker, UCSAN replaces out-of-scope external functions with dummy stubs using one of three options: (1) *Custom Wrapper* to summarize specific effects (e.g., `memcpy` must copy shadow memory) or to map to a known function (e.g., `kmalloc` mapped to `malloc` as in Figure 2). A custom wrapper can be easily introduced by providing a function that matches the stub's signature; (2) *Assume Pure* (the default) to assume no side effects; and (3) *Assume Arbitrary Changes* to conservatively reallocate all pointer arguments. For the latter two options, return values are treated as newly allocated objects (i.e., their values are drawn from the super object).

Inline Assembly. Inline assembly is prevalent in performance-critical applications and system software (e.g., OpenSSL and OS kernels). In our current prototype, we apply heuristics to handle common inline assembly patterns. More complex assembly blocks can be modeled using the aforementioned custom wrapper functions. Manually modeling every inline assembly instruction is highly case-specific and labor-intensive. It is therefore out of scope for this prototype. However, the ability to resolve inline assembly via wrappers remains a significant advantage over engines like KLEE. Furthermore, we observed that LLMs can automatically generate these wrappers, which significantly reduces the manual effort and improves scalability.

3.2 Just-In-Time Initialization

Starting the analysis from an arbitrary function poses a notable challenge: pointers accessed by the target function may not be initialized. In under-constrained symbolic execution [30], this is handled by *lazy initialization*, where a symbolic object is created when an uninitialized pointer is dereferenced. However, in UCSAN, we deal with concrete executions, so we must provide valid virtual addresses to the CPU, otherwise the program will crash with a segmentation fault. To address this challenge, we propose *Just-In-Time Initialization* (JITI). JITI consists of two key components: a new pointer representation called *Pseudo-Pointer* and a runtime mechanism to handle pointer translation and object allocation.

Pseudo-Pointer. To bridge the gap between uninitialized pointers and concrete execution, we introduce a new concept called *Pseudo-Pointer* (pseudo-pointer). Conceptually, UCSAN's pointer translation mechanism can be viewed as a software-based segmentation mechanism, where each memory object is considered as a segment, each pseudo-pointer is a 64-bit value containing the (pseudo) logical address to the segment, and the shadow pointer of a pseudo-pointer is its segment register. Under this scheme, a pseudo-pointer serves two purposes: (1) it stores the (logical) offset (introduced by pointer arithmetic) within the memory object, and (2) it provides the value required to satisfy path constraints in the concrete execution (e.g., `p != NULL`). The shadow pointer (segment register) of a pseudo-pointer tracks (1) the (logical) base of the segment, (2) the actual virtual address of the underlying memory object, and (3) the point-by relationship (i.e., where

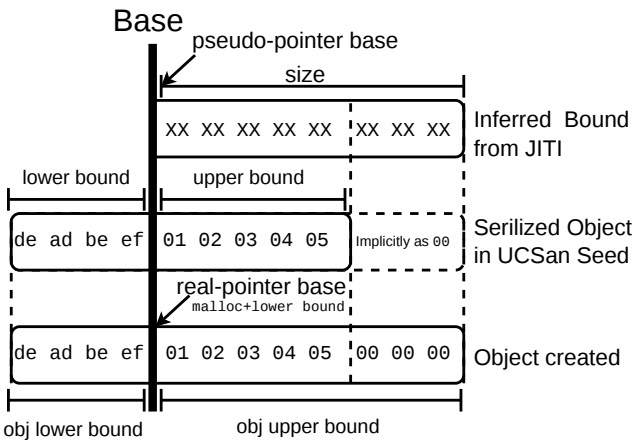


Figure 6: Allocation for one specific object during JITI.

the pointer is loaded from). Therefore, a pseudo-pointer is not a valid virtual address and cannot be dereferenced directly. Instead, it must be translated into a valid *Real Pointer* (real-pointer) before access. This translation is performed by the JITI runtime.

JITI Runtime. The core of JITI is implemented in the runtime function `check_ptr`, which is invoked before every pointer operation (as instrumented in §3.1). `check_ptr` performs the following steps:

1. Pointer Validation. First, `check_ptr` checks if the input pointer is a pseudo-pointer. It does this by inspecting the associated shadow pointer. If the pointer is a valid real-pointer (e.g., pointing to stack, global, or heap memory returned by `malloc`), `check_ptr` simply returns it. If it is a pseudo-pointer with a valid shadow pointer, `check_ptr` continues processing to the next step below.

2. Lazy Allocation. Using the shadow pointer, UCSAN looks into a metadata/segment table to check if the corresponding memory object has been allocated (i.e., does the shadow pointer have a valid object ID). If the pseudo-pointer points to an object that has not been allocated yet, UCSAN allocates a new memory object (Figure 6). The size of the allocation is inferred from the pointer type (e.g., pointer arithmetic and type casting) or the operation (e.g., `memcpy` size). UCSAN maintains an object table (indexed by Object ID) to track allocated objects, including their size, bounds, and base addresses. In certain cases, the inferred size could be larger than the current allocation (e.g., `container_of`). When this happens, UCSAN automatically reallocates the object to accommodate the larger size. As mentioned earlier, because the translation results are always ephemeral (i.e., only used for the immediate memory access and never used again), and because all pointer dereferences must go through `check_ptr`, reallocation can be done transparently without affecting correctness (i.e., no stale real-pointer will be used).

3. Initialization. After allocation, UCSAN initializes the memory content. UCSAN first retrieves the point-by metadata using the shadow pointer and checks if the test input provides specific values for this object (see §3.3). If so, UCSAN populates the memory with these values (Figure 6); otherwise, it initializes the memory with zeros. Then, UCSAN initializes the shadow memory of the memory object with shadow values to track the origin of the data, in the form of `(object_id, offset)`.

4. Pointer Translation. If the pseudo-pointer is used in pointer arithmetic (LLVM's `GEP`) or type casting (LLVM's `BitCast`), UCSAN will return pseudo-pointer value without translation. In this way, `GEP` can update the (logical) offset of the pseudo-pointer normally. When the pseudo-pointer will be dereferenced (e.g., in `Load`, `Store`), `check_ptr` translates pseudo-pointer to the actual real-pointer. This is done as follows. When the shadow pointer is first created, it records the actual value of the pseudo-pointer as the (logical) `base_offset` (which could be arbitrary value, like `-3` in Figure 4). During execution, pointer arithmetic may update the actual value of a pseudo-pointer. To compute the final real-pointer, `check_ptr` retrieves the `object_base` from the object table, based on the object ID from the shadow pointer, and adds the difference between the `current_offset` in pseudo-pointer and the recorded `base_offset` to it. Figure 5 illustrates how an object is created from the seed via JITI for the first time and how a real-pointer is retrieved via shadow pointer afterwards.

3.3 Seed Composition

In traditional symbolic execution, a different execution context is created via forking. For UCSAN, path exploration is driven by concrete inputs. Therefore, we must address the challenge of how to serialize memory objects under the current execution context into a test input; and conversely, how to deserialize a (new) test input into memory objects during JITI. The key correctness guarantee is to ensure the relationships between memory objects.

Object Provisioning. UCSAN consumes the test input during JITI. When JITI materializes a new object, it needs to decide how to initialize it. We call this process *object provisioning*. If the object corresponds to an entry in the test input, UCSAN populates it with the provided concrete values; otherwise, it initializes the object with zeros. To correctly map runtime objects to input entries, UCSAN uses two identification mechanisms: *explicit IDs* for global/argument objects and *dereference chains* for dynamically allocated objects.

1. Structured Seed Format. The test input is organized as a list of (serialized) memory objects in binary format. Each object entry contains: (1) object ID, (2) point-by metadata, (3) object size, and (4) object content.

2. Super Object and Explicit IDs. The *Super Object* (SO) (ID=0) serves as a container for all "root" under-constrained values, similar to the starting points of a mark-and-sweep garbage collector. These include: (1) initial arguments to the entry function. (2) global variables accessed by the code. (3)

return values from external functions. These values are identified by the order in which they are initialized via JITI. During execution, UCSAN maintains a counter for the number of JITI-ed values from the Super Object. When a new value is needed (e.g., an argument is accessed for the first time), UCSAN retrieves the next available value from the SO in the test input.

3. Chained Objects and Dereference Chains. For objects that are accessed via pointers (e.g., linked list nodes), we cannot rely on simple IDs because the allocation order may vary across executions. Instead, we identify these objects based on how they are reached, i.e., their *dereference chain*. This is inspired by the mark-and-sweep algorithm in garbage collection: every accessible object must be reachable from a root (anchor) via a sequence of pointer dereferences. In UCSAN, we track the *point-by* relationship: each object records where its base pointer comes from (i.e., its origin). The origin is a tuple `<source_obj_id, offset>`. By chaining these origins backward, we can uniquely identify any object starting from a root in the SO. For example, in a linked list, the second node (`obj2`) is identified as "pointed to by the `next` field (offset 4) of the first node (`obj1`)." `obj1` in turn is "pointed to by the `head` argument (ID 0) in the SO."

3.4 Memory Safety Checkers

LLVM ships with AddressSanitizer (ASAN) [33], which detects memory safety violations such as out-of-bounds accesses and use-after-free by instrumenting every memory operation and maintaining a shadow memory to track allocation metadata. Because UCSAN also uses shadow memory to associate metadata with pointers and memory, as well as instrumenting every memory-accessing instruction to call `check_ptr` first, we can reuse this infrastructure to perform the same safety checks at negligible additional cost. Specifically, we implemented three checkers to check for typical memory safety violations: out-of-bounds (OOB) access, use-after-free (UAF), and use-before-initialization (UBI). These checkers give UCSAN built-in ASAN-equivalent checking without requiring a separate sanitizer pass.

Out-of-Bounds (OOB). OOB checkers detect out-of-bounds access to *explicitly allocated* stack and heap objects. Specifically, UCSAN instruments the stack allocation site (LLVM `Alloca` instruction) and heap allocators (through custom functions) to record the base address and the size of the allocated object (i.e., the lower and upper bounds). It then stores the bounds information in the metadata table, and uses a shadow pointer to associate the bounds information with the returned pointer. UCSAN uses a field in the metadata table entry to distinguish explicitly allocated pointers from pseudo-pointers. Before every subsequent memory access, during the `check_ptr` call, UCSAN retrieves the bounds information using the shadow pointer, and checks if the access is within the recorded bounds. If the access is out of bounds, it raises an OOB error. Note that our checker operates at the allocation level and does not perform type-safety checks; intra-structure out-of-bounds accesses (e.g., overflowing one field into an adjacent field within the same struct) are not detected.

Use-After-Free (UAF). UAF checker detects accesses to memory objects that have been freed. It reuses the same metadata table entry for OOB checking, but uses a different flag. In particular, UCSAN instruments function exits (for stack objects) and explicit deallocation calls such as `free` and `kfree` (via custom functions). In these runtime functions, UCSAN uses the shadow pointer to retrieve the metadata, and marks the object as freed. Before any subsequent dereference of an aliased dangling pointer, during the `check_ptr` call, UCSAN will check the freed flag in the metadata and raise a UAF error if it is set.

Use-Before-Initialization (UBI). To detect UBI, UCSAN marks the shadow memory of every *explicitly allocated* object with a special uninitialized tag (`UNINIT=-1`). Note that objects allocated through JITI will not be marked, as we consider their content as under-constrained. Since any write to the object will also update the corresponding shadow memory, an initialization operation will clear the uninitialized tag. The UBI checker reports an error upon two types of uses: (1) an uninitialized pointer (from an explicitly allocated object) is dereferenced; and (2) uninitialized data is used as the condition for a branch.

Impact of Analysis Scope on Bug Detection. As a form of dynamic analysis, these memory safety checkers will only report errors that are actually triggered during an execution (i.e., no false positives with regard to the analysis scope). Therefore, for UCSAN to confirm a bug, the execution must traverse a concrete path that connects at least three program points: the entry function, the relevant allocation or deallocation site, and the offending use. This means that the user-specified analysis scope *must* include at least the allocation (or deallocation) and the use site; otherwise, UCSAN lacks sufficient context to report errors. However, as a form of under-constrained analysis, if the analysis scope misses important constraints (e.g., constraints on array indices, initialization of buffers), then the reported errors may still be false positives, as the execution from the real program entries (e.g., `main` or syscall entries) may not be able to reach the offending use.

This is the inherent trade-off in under-constrained analysis: on one extreme, we can analyze the whole program (i.e., reverting to traditional whole program concolic execution [6, 9, 28]), which provides the most accurate context but suffers from path explosion; on the other extreme, we can analyze only a small subset of functions, which is more efficient but may suffer from false positives due to missing constraints. In practice, there are several heuristics to strike a balance between these two extremes, which were utilized in our evaluation. First, we can use a static analysis (static bug checkers) to find potential bugs. Such static bug checkers typically (e.g.,

Sys [5] and UBITect [46]) will report the relevant control- and data-flow paths, which can then be used to define the analysis scope for UCSAN. Second, users can iteratively adjust the analysis scope based on the reported errors and their domain knowledge of the program. Third, with recent advances in large language models (LLMs), we can also leverage LLMs to automatically suggest the analysis scope.

4 Under-Constrained Concolic Execution

To demonstrate UCSAN's ability to support dynamic testing techniques, we build an under-constrained concolic execution engine UCSAN† by combining UCSAN with SymSan [9], a compilation-based concolic execution engine, and a Python-based companion server to orchestrate the path exploration.

Instrumentation. The overall process is relatively straightforward: we first apply UCSAN instrumentation pass to remove out-of-scope functions, and instrument `check_ptr` for pointer translation; then we apply SymSan's instrumentation pass to add concolic execution support; finally, we link both UCSAN's and SymSan's runtime libraries to generate the final executable. The only caveat is that since both UCSAN and SymSan use shadow memory, we need to make sure they use different memory regions to avoid conflicts. In addition, we added bridge functions to allow UCSAN to invoke SymSan to symbolize memory objects.

Path Exploration. We implement a companion server to orchestrate concolic execution by managing path constraints, querying solvers, and scheduling exploration. It is implemented as a hybrid Python/C++ program: a Python-based controller for extensible exploration strategies, and a performance-critical C++-based path constraint solver (from SymSan) for parsing symbolic traces collected from the instrumented executable into Z3 expressions, and querying Z3 to solve the constraints. The path exploration strategy used in our prototype is Breadth-First Search (BFS) (79 LoC). It is worth mentioning that it is straightforward to extend it to other strategies through the Python controller. For instance, we have also implemented a hierarchical scheduler [39] in 166 LoC, and a directed scheduler in 394 LoC.

Solving Pointer Constraints. Thanks to the introduction of pseudo-pointers, solving pointer constraints becomes more straightforward. We treat them as 64-bit bit-vectors, and find values that can satisfy path constraints. Similar to UC-KLEE [30], one limitation is that our prototype does not support the creation of cyclical data structures yet. Since this is a non-trivial problem, we consider it out of the scope of this paper.

5 Evaluation

In this section, we evaluate our prototype implementation of UCSAN, in order to answer the following research questions:

- **RQ1** **Performance Overhead**. As a compilation-based approach, does UCSAN incur lower runtime performance overhead than existing interpretation-based engines?
- **RQ2** **Compatibility**. Does UCSAN have good compatibility in analyzing real-world programs?
- **RQ3** **Impacts on Dynamic Analysis**. Can the improved efficiency of UCSAN improve the effectiveness of bug finding tools?

5.1 Experimental Setup

All experiments are carried out on a server with two Intel(R) Xeon(R) Platinum 8168 CPUs and 755GB memory. The server runs Ubuntu 20.04.3 LTS with the Linux kernel 6.5.0-25-generic. UCSAN is built with Clang 12.0 and LLVM 12.0. Since the original UC-KLEE [30] repo was no longer publicly available at the time of writing, we used the KLEE-based engine from the IncreLux [47] project (commit `80dea92`), which is a follow-up project of UBITect [46]. We refer to this engine as KLEE-IL. Both UCSAN† and KLEE-IL are configured to use Z3 4.8.15 for constraint solving.

5.2 RQ1: Performance Overhead

The key advantage of a compilation-based execution engine is its scalability (i.e., the ability to effectively handle large, complex, or growing software systems while maintaining acceptable performance, precision, and resource usage) over interpretation-based engines. In this subsection, we conduct several experiments to compare UCSAN's performance with two interpretation-based engines: KLEE-IL and Angr [34].

Linked List. First, we use the linked list example from Figure 3 as a micro-benchmark. We set the list size to 20 nodes, so the loop will iterate 20 times. Then we measured the end-to-end execution time, without path exploration. On this micro-benchmark, UCSAN's execution time is 9 seconds, while KLEE-IL took 20 seconds, and Angr took 79 seconds.

Table 1: nbench Performance (Iterations/sec)

Test	Native	UCSAN	UCSAN†	KLEE
NUMERIC SORT	1056.5	80.763	67.478	0.21653
STRING SORT	648.27	41.993	31.017	0.01282
BITFIELD	4.8555e+08	3.342e+07	2.0802e+07	65179
FP EMULATION	112.54	7.3962	5.3332	0.02442
FOURIER	1.7279e+05	95159	87193	103.29
ASSIGNMENT	32.035	1.8866	1.2791	0.0073587
IDEA	11522	349.53	204.61	0.93574
HUFFMAN	2430.7	167.96	123.43	0.51373
NEURAL NET	34.438	1.7772	1.3971	Crash
LU DECOMPOSITION	1264.6	62.549	45.997	0.16605

nbench. We further evaluated UCSAN, UCSAN† (UCSAN with SymSan), and KLEE-IL using the `nbench` benchmark suite [27]. Angr was excluded from this experiment due to severe performance limitations: it failed to complete a single

iteration after 18 hours and ultimately crashed from memory exhaustion. As shown in Table 1 (higher values indicate better performance), UCSAN outperforms KLEE-IL by several orders of magnitude across all tests. Furthermore, since nbench does not involve constraint solving, the additional overhead observed in UCSAN† stems primarily from the symbolic tracing of SymSan.

5.3 RQ2: Compatibility

In this experiment, we evaluate the compatibility of UCSAN, by compiling Linux kernel functions with it. We chose the Linux kernel as the target for this experiment because it is a widely-used open-source project with various coding styles and practices. It also has its own complex build system and dependencies.

We analyze functions from three versions of Linux. The first set is from 4.14.0, and the analysis scope is generated by the static use-before-initialization (UBI) analyzer UBITect [46]. UBITect generated 103,351 warnings from the Linux kernel 4.14.0, where each warning defines an entry function and an analysis scope (i.e., the set of functions that are involved in a UBI bug), which may span across multiple source files. We write a script to parse the list of warnings, extract the analysis scope, and generate the configuration files for UCSAN. The use of KLEE-IL follows the same procedure as described in the UBITect paper [46, 47]. The objective is to check if UCSAN offers similar compatibility as KLEE.

Table 3 reports the number of warnings UCSAN and KLEE-IL can analyze, where both engines failed to analyze all warnings. We analyzed the failed cases and found that most of them are due to the lack of support for inline assembly. Specifically, UCSAN resolves unnecessary external dependencies outside the analysis scope by replacing calls to those functions with dummy functions. However, due to the lack of support for inline assembly, when external functions are referenced in inline assembly, UCSAN will fail to build binaries for those cases and cannot analyze them. Notably, KLEE-IL also suffers from this issue. From the results, we conclude that UCSAN has good compatibility to analyze C functions compared to KLEE-IL.

In addition to 4.14.0, we also analyzed the version 5.10.240 (the longterm version used by Android) and 6.16.0 (the latest stable version at the time of evaluation). For these two versions, we used statically initialized function pointers as entry functions, and applied a simple call graph analysis to include all functions from the same module as the analysis scope. The ideal goal is to be able to compile all analysis scopes (i.e., 100%). The current prototype of UCSAN is able to compile 14,503 (96.2%) and 139,509 (88.9%) analysis scopes; most of the failed cases are also due to unhandled inline assembly.

Table 2: Overall TTF Statistics *T.O.: Timeout is 2 mins.*

	Min	25th%	Median	75th%	Mean
UCSAN†	0.51s	3.98s	4.62s	6.21s	14.37s
KLEE-IL	0.64s	92.13s	T.O.	T.O.	105.71s

5.4 RQ3: Impacts on Dynamic Analysis

We hypothesized that with the improved efficiency of UCSAN, dynamic analysis tools built on top of it, like UCSAN†, could scale to larger analysis scopes, and/or explore more execution paths. As a result, they would be able to find more bugs than previous engines. In this subsection, we conduct three experiments to validate this hypothesis.

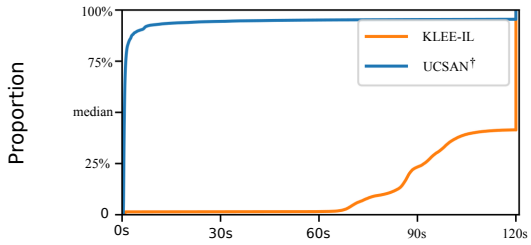
5.4.1 Processing UBI Warnings

Due to its ability to faithfully reason about memory states and path feasibility, under-constrained symbolic execution has been used to filter false positives from static analysis [5,42,46]. However, the scalability issue of existing under-constrained symbolic execution engines has limited their practicality in this context. In this experiment, we evaluate how the improved efficiency of UCSAN would impact its ability to process the results of the static analysis and identify the true positives.

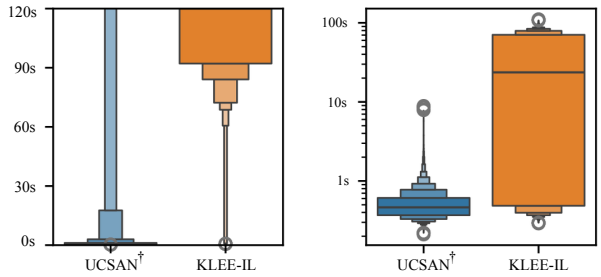
The experiment setup followed the same procedure as described in §5.3. For each warning generated by UBITect, we aim to find a feasible path from the allocation site of a variable to its use site, without going through initialization (thus use-before-initialization). For comparison purposes, we take the Time-To-Finish (TTF) as the primary evaluation metric. For fairness, the TTF includes both compilation time (for UCSAN) / linking time (for KLEE-IL), and analysis time. The compilation time for UCSAN includes the time to instrument the LLVM-IR and assemble the instrumented code to binary; while the linking time for KLEE-IL includes the time to link multiple LLVM IR modules into a single module. For this evaluation, we used 24 parallel instances for both UCSAN† and KLEE-IL, where each instance handles a different analysis task (entry function). Both UCSAN† and KLEE-IL are configured with 2-minute and 2GB limits for time and memory, the same configuration used in the UBITect/IncreLux paper [46, 47].

Figure 7 and Table 2 show that UCSAN† is 6.36x faster than KLEE-IL on average TTF. According to Table 3, when the 2 minute timeout is reached, UCSAN† was able to finish 95% of the warnings, while KLEE-IL only finished 41%. Moreover, all the cases that KLEE-IL finished processing were also completed by UCSAN†. Overall, we observe that UCSAN† processed at 0.32s/warning on average, while KLEE-IL processed at 5.14s/warning (i.e., UCSAN† is 15.06x faster than KLEE-IL on this metric).

We further investigated the 111 more confirmed results from UCSAN†. Given the large numbers, we decided to check



(a) CDF of Time-To-Finish for UCSAN† and KLEE-IL.



(b) Distribution of TTF (compile and solving time)

(c) Distribution of Compilation / Linking Time.

Figure 7: Performance for UCSAN† and KLEE-IL.

Table 3: Analyzed warnings from UBITect.

	Analyzable	Finished	Timeout	Total Time
UCSAN	66999	95.46%	4.54%	5.88h
KLEE-IL	52802	41.29%	58.71%	75.36h

the git history to see if any of these warnings have been patched in later kernel versions. The results show that at least 15 additional UBI bugs, which were reported by UBITect's static analysis but not confirmed by KLEE-IL, have since been patched in later kernel versions. This suggests that these bugs were indeed real and eventually caught by other means, yet they remained unconfirmed at the time of the original study due to the limitations of KLEE-IL. Had the authors of UBITect had access to UCSAN†, these bugs could have been confirmed and reported much earlier, potentially reducing the window of exposure for the affected kernel versions.

5.4.2 Analyzing Vulnerabilities

In this experiment, we evaluate the capability of UCSAN† for bug-finding purposes without the requirement of hand-crafted test harnesses. For reproducibility, we apply UCSAN† to rediscover known vulnerabilities. This experiment includes three datasets.

First, we manually collect 30 known vulnerabilities from the National Vulnerability Database (NVD). These vulnerabilities include both user-mode open-source software (OSS) and the Linux kernel (Table 5 in the Appendix). The selection criteria are as follows: (1) the vulnerability is a memory safety bug (e.g., buffer overflow, use-after-free, etc.) that can be detected by UCSAN's built-in checkers; (2) the vulnerability has a known root cause (i.e., the vulnerable function and the corresponding patch are known); (3) the vulnerability is not trivial to trigger (e.g., it cannot be triggered by simply calling the vulnerable function with random inputs); (4) the vulnerability is well-known or well-studied (e.g., CopyFail [36]).

Table 4: Bugs from SyzSpec analyzed with UCSAN†

Crashed Function	Bug Type	UCSAN†	False Positive
fast_imageblit	out-of-bounds	Y	
addrconf_verify_rtnl	use-after-free	Y	
dup_fd	use-after-free		F
nsim_fib6_rt_nh_del	use-after-free	Y	
rcu_core	use-after-free	Y	
addrconf_ifdown	use-before-init		F
calculate_sigpending	use-before-init	Y	
exit_fs	use-before-init	Y	
neigh_periodic_work	use-before-init	Y	
schedule_timeout	use-before-init		
tipc_release	use-before-init		F
__free_object	null-ptr-deref	Y	
batadv_bla_del. . .	null-ptr-deref	Y	
batadv_iv_send. . .	null-ptr-deref	Y	
blk_mq_put_tag	null-ptr-deref	Y	
fib_table_lookup	null-ptr-deref	Y	
get_mem_cgroup. . .	null-ptr-deref	Y	
mmap_region	null-ptr-deref		F
seccomp_run_filters	null-ptr-deref	Y	
stack_depot_save_flags	null-ptr-deref	Y	
update_io_ticks	null-ptr-deref	Y	
dup_mmap	corrupted stack		
sg_ioctl	corrupted stack		F
worker_thread	corrupted stack		
call_timer_fn	corrupted list		F
dst_init	corrupted list	Y	
free_pgtables	corrupted lock		F
process_measurement	corrupted lock		F
deliver_ptype_list_skb	page fault	Y	
__run_timer_base	deadlock	Y	
hci_dev_do_close	deadlock	Y	
serial8250_console_write	deadlock	Y	
gsm_cleanup_mux	task hung		
kvfree_rcu_bulk	task hung	Y	
rss_counter	logic bug	Y	
depot_fetch_stack	logic bug	Y	
free_pgtables	logic bug	Y	
input_mt_init_slots	logic bug	Y	

For the 30 vulnerabilities we initially selected, we manually extracted the analysis scope and entry function based on the related bug-inducing/fixing commit from the NVD database. The results listed in Table 5 indicate that UCSAN† can indeed rediscover most of the known vulnerabilities (marked as **Y**) without any test harnesses. In cases marked as **P**, UCSAN† can identify the vulnerability without requiring test harnesses, but needs to disable UCSAN's dynamic re-allocation capability. As discussed in §3.2, UCSAN estimates buffer sizes based on information collected from the operations involved. It assumes that mistakes in buffer size estimation will occur, and

will automatically re-allocate the buffer if the estimated size is insufficient. While dynamic re-allocation allows UCSAN to avoid execution errors, it could negatively impact UCSAN's buffer overflow sanitizer *when the buffer size is not known* (e.g., when the buffer is allocated outside the analysis scope). For these cases, because the analysis scopes are manually defined in an iterative manner, we did not observe any false positives.

Second, following the suggestion from the reviewers, we included 94 CVEs from the AFGen [24] paper, where we can find corresponding entries in the NVD database (Table 5 in the Appendix, from `ffjpeg` to `tcpreplay`). AFGen is a fuzzing-based approach that shares the same goal as UCSAN—testing internal functions of OSS. The main difference is that AFGen uses bi-directional program slicing to generate test harnesses (with proper object initialization) for fuzzing, while UCSAN only needs to know the scope and the entry function.

Due to the size of the datasets, we tasked a coding agent (OpenAI Codex with GPT-5.5) to generate the configuration files. Specifically, for each vulnerability, we provide the NVD database entry to the coding agent, and task it to finish three tasks: (1) identify the vulnerable version of the target OSS; (2) locate a proper entry function for the analysis, and (3) define the analysis scope. The analysis scope should contain the root cause chain of the vulnerability (e.g., from `malloc` to the out-of-bound access), but exclude irrelevant functions like `printf`. Then the agent should use UCSAN† to reproduce the vulnerability, and report whether UCSAN† reported any false positives during the analysis. The result shows UCSAN† successfully re-discovered 69 CVEs, and failed to re-discover 25 CVEs. Most of the failed cases are due to a common limitation of symbolic execution—the path explosion problem: if triggering the vulnerability requires looping too many times, or for each loop iteration or recursive call, we need to take different paths, then UCSAN† cannot finish the analysis within the 10 minutes time limit. We also checked if UCSAN† reported false positives during the analysis. The results show that UCSAN† reported false positives in 7 cases, all of which are caused by missing constraints from external functions (i.e., functions outside the analysis scope).

The third dataset is from SyzSpec [18], a recent work that uses KLEE-based under-constrained symbolic execution to extract syscall descriptions for syzkaller [37]. However, due to the performance limitations of KLEE, SyzSpec only aims to extract syscall descriptions, instead of directly finding vulnerabilities. In contrast, we apply UCSAN† to directly analyze kernel modules for bug reproduction.

Similar to the previous experiment, we use a coding agent to generate the configuration files. Moreover, since several bug types (e.g., deadlock, corrupted lock) cannot be detected by UCSAN's built-in checkers, we also tasked the coding agent to write custom checkers for those cases. The results are shown in Table 4. Among the 38 bugs reported in the SyzSpec paper, UCSAN† can successfully re-discover 26 bugs without the need for a fuzzer or syscall descriptions. Among the 12 not confirmed cases, 10 are likely due to incomplete analysis scope. Specifically, the SyzSpec paper [18] only provides the faulty function name (i.e., where a bug has manifested) and the type of the fault. However, it does not provide the detailed root cause information nor a corresponding patch. Its code repository also does not contain any reproducible test cases. As a result, the coding agent can only guess a root cause and the corresponding analysis scope. One involves a loop that exceeds UCSAN's default iteration threshold. The remaining one case is due to concurrency, which is not supported by the current prototype of UCSAN. Due to the analysis scope issues, we also observed 8 false positives among the failed cases.

6 Limitations

The current prototype of UCSAN has several limitations, which we plan to address in future work.

External functions. UCSAN tries to automatically handle external function calls outside the analysis scope in a configurable fashion. However, there may be some external functions that require special modeling, such as string related functions (`strlen`, `strchr`). Currently, UCSAN does not support these functions and may generate incorrect inputs due to missing the dependencies between the input and the return value (i.e., under-constrained). For instance, several of the CVEs from the AFGen [24] paper cannot be reproduced due to missing modeling of file I/O functions (`fstat` `vfprintf`).

Cyclic Data Structures. UCSAN does not support cyclic data structures in the current implementation. We plan to support cyclic data structures in the future, by incorporating more sophisticated shape analysis [31, 40].

Scope Annotations. The current implementation of UCSAN requires the user to provide scope annotations to specify the target functions. As shown in the bug reproduction experiments, an improperly inferred scope may lead to incomplete analysis and result in false positives or false negatives. This challenge can be partially mitigated by pairing UCSAN with a static analysis tool like UBITect [46]. With recent advances in large language models (LLMs), one can also leverage LLM-based coding agents to infer the analysis scope.

Loop and Recursion. The current prototype of UCSAN only supports coarse-grained control of loop iterations and recursive calls (i.e., a global threshold). As a result, if triggering a bug requires increasing the threshold, we can only increase the threshold globally, which may lead to state explosion and prevent UCSAN† from finishing the analysis within the time limit. We plan to support more fine-grained control of loop iterations and recursive calls in the future.

Concurrency. The current prototype of UCSAN only supports a single "thread" and does not support concurrent execution. This limits its applicability to modeling and reproducing

concurrent bugs (e.g., data races). Since concurrency is a complex topic, we consider it out of the scope of this paper and plan to support it in future work.

Multiple Entry Points. The current prototype of UCSAN only supports a single entry point specified by the user. As a result, it cannot effectively analyze stateful code like `tcp_v4_do_rcv` in the Linux kernel, which requires multiple invocations to fully cover the execution state space. Similarly, it also does not support unit testing like chaining of a series of functions. We plan to add such support in the future.

7 Related Work

Symbolic and Concolic Execution. Symbolic execution, a technique that treats program inputs as symbolic variables to explore multiple paths, has been studied extensively for decades [4, 7]. Concolic testing, which combines symbolic execution with concrete inputs, has been implemented in tools like DART [14] and CUTE [32]. Path explosion is a major performance issue for symbolic and concolic execution. SAGE [15] proposed using generational search to mitigate the path explosion issue and increase the number of generated test cases in one execution. Dowser [17] proposed using static analysis to guide concolic execution to places where it is more likely to have buffer overflow vulnerabilities. Under-constrained symbolic execution [30] is an alternative approach to reducing path explosion. This method eliminates the need for harnesses because it begins execution from an arbitrary function. Recent works like Janus [43] and SCENT [8] have demonstrated the effectiveness of analyzing kernel components in isolation to find deep bugs in file systems and kernel protocols. However, these approaches rely on manual effort to extract code or build models. Janus, for instance, manually ports file systems to user space using LibOS to enable fuzzing. In contrast, UCSAN automates the under-constrained execution process by directly compiling and analyzing the target functions, thus removing the manual effort required by previous works.

Recently, a series of works have been proposed to improve the performance of symbolic execution, by leveraging instrumentation, instead of interpretation, including QSYM [44], SymCC [28], SymQEMU [29], and SymSan [9]. UCSAN also adopts this approach to speed up symbolic execution. It extends the SymSan project.

Test Harness Generation. Automated harness generation techniques like Fudge [3], FuzzGen [20], and APICraft [48] synthesize drivers by analyzing code usage or dependencies. LLM-based approaches [23] also show promise but struggle with complex dependencies like kernel functions. UCSAN avoids these issues by leveraging the original build system and directly analyzing target functions without synthesized harnesses.

8 Conclusion

In this paper, we present UCSAN, a compilation-based under-constrained execution engine. Our evaluation shows that UCSAN is more scalable than interpretation-based engines like KLEE and Angr. We further demonstrate how UCSAN can support dynamic analysis by combining it with a compilation-based concolic execution engine to build an under-constrained symbolic execution tool. The resulting tool can effectively analyze both Linux kernel and user-mode programs, and detect bugs that are difficult to uncover using other techniques.

Acknowledgments

We thank the anonymous reviewers for their insightful comments and valuable suggestions from the shepherd. This work is supported, in part, by the National Science Foundation under Grant No. 2046026, Google ASPIRE Fund 2023 Award, and the United States Air Force and DARPA under Agreement No. FA8750-24-2-0002. Any opinions, findings, conclusions, or recommendations expressed in this paper are those of the authors and do not necessarily reflect the views of the funding agencies.

References

[1] Lkl: Linux kernel library. https://lkl.github.io/.

[2] Nathaniel Ayewah, William Pugh, David Hovemeyer, J David Morgenthaler, and John Penix. Using static analysis to find bugs. *IEEE Software*, 25(5):22–29, 2008.

[3] Domagoj Babić, Stefan Bucur, Yaohui Chen, Franjo Ivančić, Tim King, Markus Kusano, Caroline Lemieux, László Szekeres, and Wei Wang. Fudge: fuzz driver generation at scale. In *ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE)*, 2019.

[4] Roberto Baldoni, Emilio Coppa, Daniele Cono D'Elia, Camil Demetrescu, and Irene Finocchi. A survey of symbolic execution techniques. *ACM Comput. Surv.*, 51(3), 2018.

[5] Fraser Brown, Deian Stefan, and Dawson Engler. Sys: A static/symbolic tool for finding good bugs in good (browser) code. In *USENIX Security Symposium (Security)*, 2020.

[6] Cristian Cadar, Daniel Dunbar, and Dawson R Engler. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2008.

[7] Cristian Cadar and Koushik Sen. Symbolic execution for software testing: three decades later. *Communications of the ACM*, 56(2):82–90, 2013.

[8] Yue Cao, Zhongjie Wang, Zhiyun Qian, Chengyu Song, Srikanth V. Krishnamurthy, and Paul Yu. Principled unearthing of tcp side channel vulnerabilities. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, CCS '19, page 211–224, New York, NY, USA, 2019. Association for Computing Machinery.

[9] Ju Chen, Wookhyun Han, Mingjun Yin, Haochen Zeng, Chengyu Song, Byoungyoung Lee, Heng Yin, and Insik Shin. SYMSAN: Time and space efficient concolic execution via dynamic data-flow analysis. In *USENIX Security Symposium (Security)*. USENIX Association, 2022.

[10] Brian Chess and Gary McGraw. Static analysis for security. *IEEE Security & Privacy*, 2(6):76–79, 2004.

[11] Vitaly Chipounov, Volodymyr Kuznetsov, and George Candea. S2e: A platform for in-vivo multi-path analysis of software systems. In *ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2011.

[12] Pär Emanuelsson and Ulf Nilsson. A comparative study of industrial static analysis tools. *Electronic notes in theoretical computer science*, 217:5–21, 2008.

[13] Andrea Fioraldi, Dominik Maier, Heiko Eißfeldt, and Marc Heuse. Afl++: Combining incremental steps of fuzzing research. In *USENIX Workshop on Offensive Technologies (WOOT)*, 2020.

[14] Patrice Godefroid, Nils Klarlund, and Koushik Sen. Dart: directed automated random testing. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2005.

[15] Patrice Godefroid, Michael Y Levin, and David Molnar. Sage: whitebox fuzzing for security testing. *Communications of the ACM*, 55(3):40–44, 2012.

[16] Google. OSS-Fuzz - continuous fuzzing of open source software. https://github.com/google/oss-fuzz, 2016.

[17] Istvan Haller, Asia Slowinska, Matthias Neugschwandtner, and Herbert Bos. Dowser: a guided fuzzer to find buffer overflow vulnerabilities. In *USENIX Security Symposium (Security)*, 2013.

[18] Yu Hao, Juefei Pu, Xingyu Li, Zhiyun Qian, and Ardalan Amiri Sani. Syzspec: Specification generation for linux kernel fuzzing via under-constrained symbolic execution. In *ACM Conference on Computer and Communications Security (CCS)*, 2025.

[19] Marc Heuse, Heiko Eißfeldt, Andrea Fioraldi, and Dominik Maier. american fuzzy lop plus plus (afl++). https://github.com/AFLplusplus/AFLplusplus.

[20] Kyriakos Ispoglou, Daniel Austin, Vishwath Mohan, and Mathias Payer. FuzzGen: Automatic fuzzer generation. In *USENIX Security Symposium (Security)*, 2020.

[21] Tuo Li, Jia-Ju Bai, Yulei Sui, and Shi-Min Hu. Path-sensitive and alias-aware typestate analysis for detecting os bugs. In *ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2022.

[22] Dinghao Liu, Qiushi Wu, Shouling Ji, Kangjie Lu, Zhenguang Liu, Jianhai Chen, and Qinming He. Detecting missed security operations through differential checking of object-based similar paths. In *ACM Conference on Computer and Communications Security (CCS)*, 2021.

[23] Dongge Liu, Jonathan Metzman, Oliver Chang, and Google Open Source Security Team. Ai-powered fuzzing: Breaking the bug hunting barrier. https://security.googleblog.com/2023/08/ai-powered-fuzzing-breaking-bug-hunting.html, 2023.

[24] Yuwei Liu, Yanhao Wang, Xiangkun Jia, Zheng Zhang, and Purui Su. Afgen: Whole-function fuzzing for applications and libraries. In *2024 IEEE Symposium on Security and Privacy (SP)*, pages 1901–1919. IEEE, 2024.

[25] Yunlong Lyu, Yi Fang, Yiwei Zhang, Qibin Sun, Siqi Ma, Elisa Bertino, Kangjie Lu, and Juanru Li. Goshawk: Hunting memory corruptions via structure-aware and object-centric memory operation synopsis. In *IEEE Symposium on Security and Privacy (Oakland)*, 2022.

[26] Valentin JM Manès, HyungSeok Han, Choongwoo Han, Sang Kil Cha, Manuel Egele, Edward J Schwartz, and Maverick Woo. The art, science, and engineering of fuzzing: A survey. *IEEE Transactions on Software Engineering*, 47(11):2312–2331, 2019.

[27] Uwe F. Mayer. Byte magazine's bytemark benchmark program. https://www.math.utah.edu/~mayer/linux/bmark.html, 2017.

[28] Sebastian Poeplau and Aurélien Francillon. Symbolic execution with symcc: Don't interpret, compile! In *USENIX Security Symposium (Security)*, 2020.

[29] Sebastian Poeplau and Aurélien Francillon. SymQEMU: Compilation-based symbolic execution for binaries. In *Annual Network and Distributed System Security Symposium (NDSS)*, 2021.

[30] David A. Ramos and Dawson Engler. Under-constrained symbolic execution: Correctness checking for real code. In *USENIX Security Symposium (Security)*. USENIX Association, 2015.

[31] Noam Rinetzky and Mooly Sagiv. Interprocedural shape analysis for recursive programs. In *International Conference on Compiler Construction*, 2001.

[32] Koushik Sen, Darko Marinov, and Gul Agha. Cute: a concolic unit testing engine for c. In *ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE)*, 2005.

[33] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitry Vyukov. Addresssanitizer: A fast address sanity checker. In *USENIX ATC 2012*, 2012.

[34] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Andrew Dutcher, John Grosen, Siji Feng, Christophe Hauser, and Christopher Kruegel. Sok:(state of) the art of war: Offensive techniques in binary analysis. In *IEEE Symposium on Security and Privacy (Oakland)*, 2016.

[35] the Clang team. Dataflowsanitizer design document. https://clang.llvm.org/docs/DataFlowSanitizerDesign.html, 2018.

[36] Theori. CVE-2026-31431: Copy Fail — Linux Kernel Local Privilege Escalation and Container Escape. https://copy.fail/, 2026. Accessed: 2026-06-02.

[37] Dmitry Vyukov. Syzkaller: the next gen kernel fuzzer. In *Proceedings of the 2015 Linux Plumbers Conference*, 2015.

[38] Dmitry Vyukov and Andrey Konovalov. Syzkaller: an unsupervised coverage-guided kernel fuzzer. https://github.com/google/syzkaller, 2015.

[39] Jinghan Wang, Chengyu Song, and Heng Yin. Reinforcement learning-based hierarchical seed scheduling for greybox fuzzing. In *Annual Network and Distributed System Security Symposium (NDSS)*, 2021.

[40] Reinhard Wilhelm, Mooly Sagiv, and Thomas Reps. Shape analysis. In *International Conference on Compiler Construction*, 2000.

[41] Qiushi Wu, Yang He, Stephen McCamant, and Kangjie Lu. Precisely characterizing security impact in a flood of patches via symbolic rule comparison. In *Annual Network and Distributed System Security Symposium (NDSS)*, 2020.

[42] Meng Xu, Chenxiong Qian, Kangjie Lu, Michael Backes, and Taesoo Kim. Precise and Scalable Detection of Double-Fetch Bugs in OS Kernels. In *IEEE Symposium on Security and Privacy (Oakland)*, 2018.

[43] Wen Xu, Hyungon Moon, Sanidhya Kashyap, Po-Ning Tseng, and Taesoo Kim. Fuzzing file systems via two-dimensional input space exploration. In *IEEE Symposium on Security and Privacy (Oakland)*, 2019.

[44] Insu Yun, Sangho Lee, Meng Xu, Yeongjin Jang, and Taesoo Kim. Qsym: A practical concolic execution engine tailored for hybrid fuzzing. In *USENIX Security Symposium (Security)*, 2018.

[45] Michal Zalewski. American fuzzy lop.(2014). http://lcamtuf.coredump.cx/afl, 2014.

[46] Yizhuo Zhai, Yu Hao, Hang Zhang, Daimeng Wang, Chengyu Song, Zhiyun Qian, Mohsen Lesani, Srikanth V. Krishnamurthy, and Paul Yu. Ubitect: a precise and scalable method to detect use-before-initialization bugs in linux kernel. In *ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE)*, 2020.

[47] Yizhuo Zhai, Yu Hao, Zheng Zhang, Weiteng Chen, Guoren Li, Zhiyun Qian, Chengyu Song, Manu Sridharan, Srikanth V. Krishnamurthy, Trent Jaeger, and Paul Yu. Progressive scrutiny: Incremental detection of ubi bugs in the linux kernel. In *Annual Network and Distributed System Security Symposium (NDSS)*, 2022.

[48] Cen Zhang, Xingwei Lin, Yuekang Li, Yinxing Xue, Jundong Xie, Hongxu Chen, Xinlei Ying, Jiashui Wang, and Yang Liu. APICraft: Fuzz driver generation for closed-source sdk libraries. In *USENIX Security Symposium (Security)*, 2021.

9 Appendix

9.1 Instrumented IR for the motivating example

- Line 10: This is the first time `head` is checked for dereference. Though the subsequent instruction is a `GEP` instruction, which does not dereference the pointer, we still initialize the object pointed to by `head` and return the real-pointer to the `GEP` instruction. Notably, the size of the object is determined by the size of the pointer type which is `sizeof(struct node)`.
- Line 12: `ptr_v` is a pseudo-pointer, which is derived from `head` through pointer arithmetic. Note that `ptr_v` shares the same taint information with `head` as they point to the same object. Thus, no object is introduced in this step. Since the subsequent instruction is a `load` instruction, we return the real-pointer pointing to the desired offset and put it as the operand of the `load`.
- Line 24: Similarly to Line 10, JITI is not engaged in this step, as the object has been initialized.
- Line 26: Similarly to Line 12, JITI is not engaged in this step. Note that the returned real-pointer will be different from the previous one, as the desired offset points to the `next` field.
- Line 10′: Once the first loop iteration is finished and hits this line again, the taint information of `head` will be the next node pointed by the initial one, which points to an uninitialized object. Thus, JITI will be engaged again.

The original **`main`** function in the input code will be dropped and replaced by a crafted **`main`** function as shown in Line 31 in IR. In this function, we insert a call to `__set_label_for_args` to retrieve the value for the argument from the *super object* and set up taint for the argument. Then we invoke the target function `cal` with the argument. Note that the target function is specified by the user via analysis scope annotations.

9.2 Analyzing Additional Kernel Modules

To further demonstrate UCSAN†'s scalability, we analyzed a few kernel modules, with a 1-hour time budget for each module.

- `binder_ioctl`: The main entry function in the Binder IPC module from the Android kernel. Due to its complexity, there have been multiple security vulnerabilities discovered in this module. So we used UCSAN† to analyze this module from the latest Android 5.10 LTS kernel. The analysis scope includes 202 functions and 2,027 basic blocks. UCSAN† is able to cover 1,390 basic blocks (68.6%). The exploration triggered several error exits and 7 null-pointer dereferences. Upon manually replaying the traces and inspecting the logs, all of them were false positives due to missing context.
- `tcp_v4_do_rcv`: The main entry function for processing incoming TCP packets in the Linux kernel. We chose this function as it was analyzed in a prior work [8] where the authors manually extracted related code for model checking. We analyzed the function from the 6.16.0 kernel. The analysis scope includes 1,028 functions across 26 files, and 8,930 basic blocks. UCSAN† is able to cover 1,437 basic blocks (16.1%). After inspection, we found that this function is stateful and re-entrant. However, our current prototype only invokes the entry function once, which limits the code coverage. Neither error exits nor checker-reported errors are triggered in this analysis.
- `acpi_ex_opcode_2A_1T_1R` and `acpi_ex_opcode_1A_0T_1R`: The entry functions of the ACPI AML Interpreter module in the Linux kernel. We analyzed these two functions from the 6.16.0 kernel. The analysis scope of `acpi_ex_opcode_2A_1T_1R` includes 324 functions across 74 files, and 2,415 basic blocks. UCSAN† is able to cover 1,192 basic blocks (49.36%). The analysis scope of `acpi_ex_opcode_1A_0T_1R` includes 315 functions across 73 files, and 2,446 basic blocks. UCSAN† is able to cover 1,382 basic blocks (56.50%). For each of these two entry functions, the checker reported one null-pointer dereference, and one OOB access; both are false positives caused by calling `kmalloc` with size 0.

9.3 CVE Reproduction Case Studies

This section presents the detailed results of UCSAN†'s bug reproduction experiments. We evaluate UCSAN† on three datasets of known vulnerabilities: (1) 30 manually curated CVEs from the NVD, covering both user-mode OSS and the Linux kernel; (2) 94 CVEs from the AFGen [24] benchmark, spanning 11 open-source projects; and (3) 38 kernel bugs from SyzSpec [18].

Table 5 lists the full results for datasets (1) and (2). For each CVE, we report the project, CVE identifier, bug type, and whether UCSAN† successfully reproduced it (**Y**), failed (**N**), or succeeded only with dynamic re-allocation disabled (**P**).

Table 4 lists the SyzSpec kernel bugs. Since these bugs span diverse types (e.g., deadlock, corrupted lock) beyond memory safety, we also tasked a coding agent to generate custom checkers. UCSAN† reproduced 26 out of 38 bugs. Among the 12 unconfirmed cases, 10 are due to structural differences between the kernel version under test (v6.10) and the version described in the SyzSpec specification, and one exceeds the default loop iteration threshold.

```
define i32 @cal(%struct.node* %arg) {
  %sum = alloca i32, align 4
  %head = alloca %struct.node*, align 8
  store %struct.node* %arg, %struct.node** %head
  br label %while_loop
while_loop:
  %icmp = icmp ne %struct.node* %head, null
  br i1 %icmp, label %loop_body, label %loop_end
loop_body:
  call i32* __check_pointer(%head) ;JITI for GEP
  %ptr_v = getelementptr inbounds %struct.node,
    %struct.node* %head, i32 0, i32 0
  %real_v = call i32* __check_pointer(%ptr_v) ;
    JITI for load
  %v.shadow = call i32 @__taint_load(i32* %real_v)
     ; load taint for %v
  %v = load i32, i32* (%real_v)
  %1.shadow = call i32 @__taint_union(
     i32 %sum.shadow, i32 %v.shadow, ; symbolic
    operands
     i16 ADD, ; operator
     i8 32, ; operand size in bits
     i32 %1, i32 %sum ; concrete operands
     )
  %1 = add (load i32, i32* %sum), %v
  call i32 @__taint_store(i32* %sum.shadow, i32 %1
    .shadow) ; store taint for %sum
  store i32 %1, i32* %sum ; write %1 to sum
  call i32* __check_pointer(%head) ;JITI for GEP
  %ptr_next = getelementptr inbounds %struct.node,
     %struct.node* %head, i32 0, i32 1
  %2 = load %struct.node*, (call __check_pointer(
    %ptr_next)) ; JITI for load
  store %struct.node* %2, %struct.node** %head
  br label %while_loop
loop_end:
  ...
}
define i32 @main() {
  %head = call i32* __set_label_for_args(0) ; load
     value for argument 0 from super object
  call cal(bitcast i32* %head to %struct.node*) ;
    invoke the target function
}
define i32 @"original$main"() {
  ; deprecated original main function
  ...
}
```

Figure 8: Instrumented LLVM-IR

Table 5: Re-discover CVEs using UCSAN†. [1] needs zlib stub. [2] needs sprintf stub. [3] needs pcap stubs.

Project	CVE	Type	Result
OpenSSL	CVE-2022-3602	OOB	Y
	CVE-2022-3786	OOB	Y
	CVE-2021-3711	OOB	N
	CVE-2021-3449	UBI	P
	CVE-2016-7054	OOB	Y
SQLite	CVE-2020-13632	OOB	Y
binutils	CVE-2016-4489	UB & OOB	Y
	CVE-2016-4491	OOB	N
	CVE-2016-4492	OOB	Y
sudo	CVE-2021-3156	OOB	P
curl	CVE-2019-5482	OOB	N
redis	CVE-2018-12326	OOB	P
Linux	CVE-2021-43267	OOB	Y
	CVE-2020-14386	UBI & OOB	Y
	Git Hash:ef5305f	OOB	Y
	Git Hash:50508d9	UBI	Y
	CVE-2023-31436	OOB	Y
	CVE-2023-6931	IOF	Y
	CVE-2023-4273	OOB	Y
	CVE-2024-26586	OOB	Y
	CVE-2023-52436	OOB	Y
	CVE-2023-52434	OOB	Y
	CVE-2023-6606	OOB	Y
	CVE-2023-42754	NULL_DEREF	Y
	CVE-2023-52435	NULL_DEREF	Y
	CVE-2024-1085	UAF	Y
	CVE-2023-6817	UAF	Y
	CVE-2023-3609	UAF	Y
	CVE-2023-3776	UAF	Y
	CVE-2026-31431	Convention	Y
ffjpeg	CVE-2020-23851	OOB	Y
	CVE-2020-23852	OOB	Y
	CVE-2021-45385	NULL_DEREF	Y
	pr-46	OOB	Y
	CVE-2019-19888	DIV_ZERO	Y
	CVE-2019-19887	NULL_DEREF	Y
	CVE-2019-16350	OOB	Y
	CVE-2019-16351	OOB	Y
	pr-45	OOB	N external global
	CVE-2020-13440	OOB	Y
	CVE-2019-16352	OOB	Y
	CVE-2020-13439	OOB	Y
	CVE-2020-13438	NULL_DEREF	Y
	CVE-2020-15470	NULL_DEREF	Y
	CVE-2020-23705	OOB	N external global
jasper	CVE-2018-19139	OOB	Y
	CVE-2018-19540	OOB	Y
	CVE-2018-9055	assertion	Y
	CVE-2018-9154	abort	Y
	CVE-2018-9252	assertion	Y
jhead	CVE-2022-28550	OOB	Y
	CVE-2018-6612	OOB	Y
	CVE-2019-1010302	OOB	Y
	CVE-2019-19035	OOB	Y/w FP
	CVE-2020-6624	OOB	Y/w FP
	CVE-2020-6625	OOB	N/w FP
	CVE-2021-3496	OOB	Y
liblouis	CVE-2018-11685	OOB	N loops w/FP
	CVE-2017-13743	OOB	Y
	CVE-2023-26767	OOB	Y
	CVE-2023-26768	OOB	Y
	CVE-2023-26769	OOB	Y
liblouis	CVE-2018-11577	OOB	N vfprintf w/FP
	CVE-2018-11440	OOB	N data-deps
	CVE-2018-17294	OOB	N data-deps w/FP
libtiff	CVE-2018-7456	NULL_DEREF	Y
	CVE-2022-34526	OOB	N VAARG
	CVE-2017-18013	NULL_DEREF	N VAARG
libwav	CVE-2018-14051	Infinite Loop	N loop
lupng	CVE-2018-18581	OOB	Y
	CVE-2018-18582	OOB	Y[1]
	CVE-2018-18583	OOB	Y[1]
oniguruma	CVE-2019-19203	OOB	N missing malloc
libming	CVE-2018-11100	NULL_DEREF	Y
	CVE-2018-20428	NULL_DEREF	N multi functions
	CVE-2018-20591	OOB	Y w/FP
	CVE-2019-16705	OOB	N recursion
	CVE-2019-9113	NULL_DEREF	N recursion
	CVE-2020-6629	NULL_DEREF	N loops
	CVE-2018-7867	OOB	Y[2]
	CVE-2018-7877	OOB	Y[2]
	CVE-2018-11095	OOB	Y
	CVE-2018-9165	NULL_DEREF	Y
	CVE-2018-7873	OOB	Y[2]
	CVE-2018-11226	OOB	Y[2]
	CVE-2018-6315	OOB	Y
	CVE-2021-34342	OOB	Y
	CVE-2018-7868	OOB	N main as entry
	CVE-2018-7870	NULL_DEREF	N loops w/FP
	CVE-2018-7871	OOB	N loops
	CVE-2018-7872	NULL_DEREF	N loops
	CVE-2018-7874	NULL_DEREF	Y
	CVE-2018-7875	OOB	Y
	CVE-2017-16883	NULL_DEREF	Y
ngiflib	CVE-2019-20219	OOB	Y w/FP
	CVE-2019-16346	OOB	N loops w/FP
	CVE-2019-16347	OOB	N data-deps
	CVE-2018-10677	OOB	Y
	CVE-2021-36530	OOB	N main as entry
	CVE-2021-36531	OOB	N main as entry
	CVE-2022-30858	external lib modeling	N
	CVE-2018-11575	OOB	N loops
	CVE-2018-10717	OOB	Y
	CVE-2018-11576	OOB	Y
tcpreplay	CVE-2020-12740	OOB	Y[3]
	CVE-2018-20552	OOB	Y
	CVE-2018-20553	OOB	Y
	CVE-2023-27783	assertion	Y
	CVE-2023-27784	NULL_DEREF	Y
	CVE-2023-27785	NULL_DEREF	Y
	CVE-2023-27786	NULL_DEREF	Y
	CVE-2023-27787	NULL_DEREF	Y
	CVE-2023-27788	assertion	Y
	CVE-2023-27789	assertion	Y
	CVE-2020-23273	OOB	Y
	CVE-2021-45386	assertion	Y
	CVE-2021-45387	assertion	Y
	CVE-2022-25484	assertion	Y
	CVE-2018-17974	OOB	Y[3]
	CVE-2020-24265	OOB	Y
	CVE-2020-24266	OOB	Y[3]
	CVE-2020-18976	OOB	Y[3]
	CVE-2018-18407	OOB	Y[3]
	CVE-2018-13112	OOB	Y[3]

ARTIFACT EVALUATED
USENIX
AVAILABLE

Aletheia: Automated Detection of Data Integrity Violations in Microservices

Mafalda Sofia Ferreira, João Ferreira Loff, João Garcia, Rodrigo Rodrigues
INESC-ID, Instituto Superior Técnico, Universidade de Lisboa

Abstract

The decomposition of applications into microservices results in data being stored across heterogeneous systems, with data schemas partitioned and managed by separate services. As the size and complexity of microservice deployments grow, the consistency of data associations at the application level can be overlooked, and it becomes challenging to manually identify data integrity violations.

In this paper, we present a systematic approach for identifying semantic violations in microservice ecosystems, namely operations and service interactions that break data integrity. We formalize these violations by leveraging the Entity-Relationship model and relational algebra to characterize problematic operation patterns across distributed services. We then introduce Aletheia, a static analysis tool that operationalizes this framework, using the static single assignment (SSA) representation to track data flow and a novel abstract call graph to detect integrity violations. We evaluate Aletheia on 7 open-source applications, detecting 46 previously unreported integrity violations, and use production-scale synthetic applications to demonstrate scalability to large call graphs with up to 500 microservices.

1 Introduction

The microservice architecture is a paradigm in which applications are decomposed into small components that can be independently developed, deployed, and scaled [1–4]. Contrary to a monolithic architecture where application logic is centralized into a single unit, the loose coupling of microservices enables the separation of responsibilities among different teams [5]. This design not only allows teams to work independently on smaller, self-contained codebases, but also to adopt the programming languages and storage systems that best align with their business requirements [5, 6].

Consequently, data management in microservice architectures is also decentralized, namely by allowing each service to choose its storage backend [2], which can range from scalable and weakly consistent NoSQL data stores [7–9] to relational databases with one-copy serializability semantics [10]. While this flexibility aligns with and preserves the benefits of microservices, it also introduces challenges in maintaining data consistency and integrity across the distributed system.

More concretely, when a large-scale service is decomposed into a set of microservices, the logical relationships within the data that form the state of the service should naturally persist even when the actual data is partitioned and therefore physically isolated across these services. As such, it becomes crucial to find systematic ways to model and reason about these logical connections. In this context, the Entity-Relationship (ER) model provides a time-tested, principled formalism for modeling these cross-service relationships, since the ER model already abstracts *logical* relationships from their *physical* implementation.

However, while the logical relationships remain conceptually identical to those captured by the ER model, the microservice decomposition renders database-level enforcement mechanisms (such as foreign key relationships) no longer applicable across service boundaries. Hence, these data dependencies and associations are typically handled by application code. Yet, due to the complexity of microservices, it can be almost impossible for developers to have a comprehensive understanding of the entire system [11, 12]. As a consequence, the preservation of data relationships can be overlooked or dismissed, compromising data integrity [13]. In practice, such inconsistencies have been reported even in large-scale production systems [14].

In this paper, we tackle this problem through a principled approach based on two pillars. First, we develop a formal framework for detecting data integrity violations in microservices by extending concepts from the ER model with relational algebra specifications tailored to cross-service data dependencies. Our framework characterizes how a microservice decomposition can violate three canonical types of data integrity constraints: referential integrity (ensuring that references between entities remain valid), entity integrity (ensuring that primary keys uniquely identify records across partitioned data), and uniqueness constraints (ensuring that unique values remain consistent despite concurrent operations). In particular, for each constraint type, we identify operation patterns — including cascading deletes and concurrent operations — that violate integrity when data relationships span service boundaries, and we formalize these patterns using relational algebra for their systematic detection.

Second, building on these formal specifications, we introduce *Aletheia*[1], a system for automated detection of integrity

[1] Aletheia is available at `https://github.com/aletheia-microservices`.

violations in microservice codebases. Aletheia leverages static analysis to identify the operation patterns formalized in our ER-based framework, translating the corresponding relational algebra specifications into concrete detection rules without relying on user input. Aletheia's detection relies on two key techniques. First, the system analyzes the code of each microservice in its static single assignment (SSA) intermediate representation (IR) [15]. In particular, by building upon taint propagation [16] over an SSA-based IR [17] and annotating variables with information about database operations, Aletheia can infer how values flow throughout any execution into various database operations. Second, we introduce a new *abstract call graph* that represents possible call graphs containing microservice invocations and database operations, along with filtered taint information from the SSA analysis. This graph enables the automatic extraction of the application data schema and integrity constraints, which are then validated against our formal specifications.

We implemented Aletheia as a framework in Go and used it to analyze several microservice benchmarks and applications [18–23] written in Blueprint [24]. These applications span diverse domains (including e-commerce, social networks, media processing, and reservation systems) and employ storage backends ranging from relational databases to document stores, key-value stores, and message brokers.

Our evaluation shows that Aletheia effectively identifies data integrity violations in real-world microservice applications. The system achieves a precision of 81% and a recall of 69% across applications of varying sizes and storage architectures. Notably, Aletheia detected 50 distinct integrity violations, of which 46 had not been previously reported in the literature, including issues in widely used benchmarks such as DeathStarBench and TrainTicket [18, 19]. These findings demonstrate that data integrity violations are pervasive even in well-studied codebases. Furthermore, Aletheia scales to large deployments, processing synthetic codebases containing up to 500 microservices and 2,887 distinct call graphs in under 20 minutes, making it practical for integration into development workflows. These codebases are designed to closely resemble real-world production systems [12, 25].

In summary, this paper makes the following contributions:

- We highlight the problem of data integrity violations in microservice architectures and propose a principled detection framework, built on a formal extension of relational algebra and the ER model.
- We introduce Aletheia, a system that automatically analyzes microservices and detects interactions that may result in such data integrity violations.
- We demonstrate that our system can effectively analyze applications at scale and identify previously unknown integrity anomalies in existing microservice codebases.

The remainder of this paper is organized as follows. §2 discusses data integrity challenges introduced by microservice architectures. §3 describes our principled approach for identifying violations of data integrity constraints in microservices. §4 presents the architecture and implementation of Aletheia. In §5, we present our experimental evaluation, which analyzes real-world microservice applications and assesses the scalability of the framework. Finally, we discuss the limitations of Aletheia in §6, survey the related work in §7, and conclude the paper in §8.

2 Data Integrity in Microservices

In this section, we motivate the main data integrity challenges introduced by microservices and overview our approach.

2.1 Challenges

Microservices introduce two key challenges to data integrity: heterogeneous storage and partitioned schemas.

Heterogeneous storage systems. Microservice applications typically employ heterogeneous storage systems, each with its own consistency model and transactional semantics. While some databases provide ACID transactions with strong consistency, others adopt BASE (Basically Available, Soft State, Eventually Consistent) principles [8], prioritizing availability through eventual consistency [7]. This diversity creates several problems. First, the guarantees provided by individual databases cannot be maintained when systems are combined. Even when each datastore individually offers some level of guarantees, applications cannot enforce a unified consistency model across all datastores. Providing stronger guarantees (e.g., causal consistency or read-your-writes) across heterogeneous systems requires instrumenting code with specialized libraries [20, 26, 27]. Second, transactional semantics do not extend beyond individual databases. Developers must either forego transactions entirely or adopt weaker patterns such as sagas [28], which implement sequences of local transactions at each service but provide no isolation guarantees across datastores. Third, using even a few weakly consistent datastores introduces complexity. For example, in multi-master databases with eventual consistency [29] (found in both NoSQL [30, 31] and SQL [32, 33] systems), concurrent writes may target different replicas and create divergent state. While conflict resolution mechanisms such as CRDTs [34] or last-writer-wins (LWW) policies [30, 35, 36] ensure convergence, they complicate application semantics, particularly when a single request triggers multiple conflicts resolved differently by each service.

Enforcing data integrity across partitioned schemas. As data is split across databases according to the logic of each service, schemas become fragmented and adapted to different database models [1], complicating in several ways the database-level enforcement of constraints that would exist in monolithic applications. First, certain constraints cannot be enforced across separate databases — for instance, foreign keys that establish referential integrity cannot be declared when schemas are distributed across multiple datas-

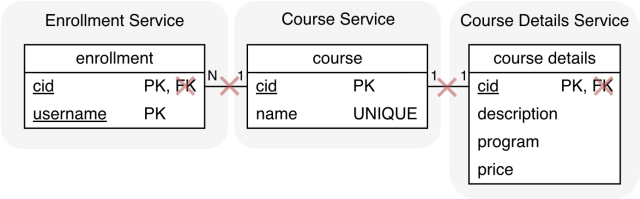


Figure 1: ER diagram with a possible decomposition of a monolithic application into microservices. The cross symbol in red represents the effects of the decomposition on the integrity constraints.

tores [13]. Second, some databases lack support for certain constraint types entirely, as is common for numerical constraints in NoSQL storage [30, 31]. Third, even for constraints that individual databases can enforce, a single request may span operations across partitioned data, requiring developers to understand how different enforcement mechanisms (e.g., distinct conflict resolution policies) interact and influence the overall side effects of an operation.

In practice, these limitations have forced developers to take on the responsibility for maintaining data integrity at the application level [13], yet the inherent complexity of microservices [11, 12, 37] hinders ensuring correctness. To concretely illustrate these challenges, we introduce an example that demonstrates their effects on data integrity constraints.

2.2 Motivating example

Consider an application for online course enrollment with the relational schema depicted in Fig. 1. This shows a monolithic version containing the *course*, *course details*, and *enrollment* entities along with their integrity constraints, and overlays a possible microservice decomposition. This example contains all three canonical types of data integrity constraints.

Referential integrity. The referential integrity constraint ensures that records can only be referenced if they exist in the referred table [38, 39]. In relational databases, referential integrity is preserved by ensuring that, whenever a record delete (or update) operation is issued, the operation is either blocked, returning an error, or the effects of the operation are propagated to all the records referencing the deleted record, thereby also being deleted (or updated).

In the example monolithic system, the database enforces referential integrity through foreign keys (e.g., the `cid` column linking enrollment and course tables). However, after decomposition into microservices, these foreign keys can no longer be defined, and the cross-table relationships they enforced disappear (denoted with a red cross in Fig. 1).

In general terms, microservices cannot rely on referential integrity across databases, shifting this responsibility to the application layer — and ultimately to the developer [13].

Entity integrity. The entity integrity constraint ensures that records in a table are uniquely identified through primary keys, which must be unique and not null [38, 39].

In the exemplified monolith, entity integrity is enforced by the database through the `cid` primary key in both *course* and *course details* tables, and through the composite primary key `cid` and `username` in the *enrollment* table.

After decomposition, all entities share a copy of the same primary key across different databases. Then, while databases still enforce entity integrity, the services must collectively ensure system-wide integrity: if a primary key value can identify one record for a given entity (e.g., *course*) in one database, then the same primary key value must also identify all the remaining partitioned records (e.g., *course details*) for entities in other databases.

Uniqueness constraint. The uniqueness constraint (also enforced by primary keys) ensures that a specific value is never repeated [38, 39]. In our example from Fig. 1, the uniqueness constraint is enforced by the database for the `name` column in the *course* entity.

After decomposition, the *course* entity that contains both uniqueness constraints is bound to a single service (*course* service), whose database includes mechanisms to ensure that values never repeat. However, the *course details* database (whose data may depend on the *course* database) may not be aware of the implications of such mechanisms, and jeopardize the overall data integrity. For example, the *course* database may discard the insertion of a new record if another course with the same name already exists, while the respective details are successfully inserted in the *course details* database, leaving the system with a partially created course.

These examples show how distributing data across microservices increases the risk of violating the canonical constraints that are trivial to enforce in monolithic systems.

2.3 Automating Data Integrity Analysis

Ideally, developers would have access to automated tools that can easily identify situations that endanger integrity constraints, despite the inherent complexity of microservices. However, given the limited research and the low visibility of this problem, developers are either not aware of such risks or are forced to manually debug their systems, thereby increasing the chances of compromising data integrity.

To address this, we introduce Aletheia, whose goal is to identify the parts of the code that may invalidate foreign key, primary key, and uniqueness constraints. However, to determine the code patterns that flag these potential problems, we require a precise understanding of the semantics of microservices, how their state is accessed and shared across different services, and which integrity-related invariants must hold when data is accessed. To this end, we formally model these concepts and define the conditions that can lead to violating the above-mentioned integrity constraints, which enables us to precisely derive a set of problematic sequences of actions. Building on this foundation, Aletheia (detailed in §4) then conducts a static analysis of an entire set of microservices. This starts with an intra-procedural analysis to detect how values flow throughout the execution to infer which variables are used in datastore operations and included in remote ser-

vice calls. Then, for scalability to larger codebases, we move into an inter-microservice data flow analysis based on a new abstract call graph, which is formed by retaining minimal information from the previous intra-procedural step according to our formal model. Ultimately, Aletheia operates exclusively on the abstract call graph, traversing it to infer cross-service foreign key relationships, and then searching for patterns in which database operations on data shared across services suggest that an integrity violation may occur.

3 A Model for Data Integrity in Microservices

In this section we lay out our approach to reasoning about how operations in microservices may break data integrity. We build on relational algebra to specify conditions to break the various integrity constraints, and then reason about the combinations of operations that may trigger those conditions.

3.1 System Model and Notation

We consider microservice applications where data is partitioned across several databases, and operations are issued individually to each database by the corresponding service.

The state S of a database consists of a set of data objects in a domain $\mathcal{D}$. The initial state of a database A is S_{A_0}, and a subsequent state, S_{A_i}, is obtained after applying a sequence of i operations. We restrict the set of database operations $\mathcal{O}$ to writes, reads, and deletes on tabular data, and denote them as $w,\ r,\ d \in \mathcal{O}$. We define a request as a group of operations $\mathcal{O}$ that are collectively triggered by a client application call.

For any two databases, A and B, that are part of a microservice application with current states S_{A_i} and S_{B_j}, the global application state can be denoted as $S_{\{A_i,B_j\}} = S_{A_i} \cup S_{B_i}$, which contains all data objects from both databases. We use T_A and T_B to denote relational tables from databases A and B.

When a database is replicated under weak consistency (which will be necessary to present data inconsistencies related to uniqueness constraints) its state may diverge when operations execute concurrently on different replicas. Here, we make the usual assumption that a convergent state is subsequently obtained by applying a merge procedure [40].

We consider data integrity to hold when a given integrity constraint, defined as a condition over the global application state, is satisfied by the current state. Table 1 presents the notation for defining the most commonly used integrity constraints in relational databases, namely foreign key, primary key, and uniqueness constraints. We do not include other constraint types such as check or not null, since they are either application-specific or enforced locally by a single database whose integrity does not depend on the state of other databases. Note that while we define constraints using relational database terms such as tables, columns, or records, this does not lose generality. In particular, these concepts can be adapted to other data models used in document-based (collections, fields, documents) or key-value stores (tables, attributes), as we will subsequently show by analyzing realistic microservice ecosystems with a diverse set of storage backends. Table 1 also shows the notation to represent operations.

Constraints	$\text{FK}\left(b_j^{T_B}, a_i^{T_A}\right)$	column b_j in table T_B is a foreign key referencing column a_i in table T_A.
	$\text{PK}_{T_A}(a_i)$	column a_i is the primary key of table T_A
	$\text{UNIQ}_{T_A}(a_i)$	column a_i is declared as unique in table T_A
Operations	$w_{T_A}(a_i = x, \ldots)$	write a record to table T_A specifying the values for one or more of its columns (in this example, column a_i with value x).
	$(a_j = y, \ldots) \leftarrow r_{T_A}(a_i = x)$	read from table T_A filtering only the records where column a_i has value x.
	$d_{T_A}(a_i = x)$	delete from table T_A the records where column a_i has value x.
	$[o_1 \wedge o_2]_{req}$	request req, including at least operations o_1 and o_2

Table 1: Notation to represent database constraints and operations.

3.2 Referential Integrity Violations

To understand the operation patterns that endanger referential integrity constraints, we first recall the respective definition: referential integrity states that an association can only be established between two records if the referenced record exists. In a microservice application, this link is established through *implicit inter-microservice associations* [13] between partitioned database records in different microservices, creating a *foreign key* relationship.

The definition of a foreign key is formalized as follows, where databases A and B hold tables T_A and T_B, respectively:

$$FK(b_j^{T_B}, a_i^{T_A}) \triangleq \forall b_j \in T_B \ (b_j \neq null \Rightarrow \exists a_i \in T_A : b_j = a_i)$$

Furthermore, we want to formalize a way to derive the existence of foreign key relationships, even when a schema annotated with those relationships does not exist, which is the case in microservice code. To this end, we infer that a foreign key $FK(b_j^{T_B}, a_i^{T_A})$ is established between tables T_A and T_B whenever a value x that is written to or read from table A is subsequently written in a column of table B (even if column a_i is not uniquely constrained in T_A). The intuition is that the value x is first associated with a record in table A, and its subsequent use for creating a record in table B establishes a referential dependency from T_B to T_A, since the newly written record in T_B depends on the existence of the other record in T_A. This corresponds to the following operation pattern:

$$\left((a_i = x, \ldots) \leftarrow r_{T_A}(\ldots) \vee w_{T_A}(a_i = x, \ldots)\right) \ \wedge \ w_{T_B}(\ldots, b_j = x)$$

Following the notation in Table 1, the left conjunct represents one of two possible operations that must occur in table A: a read to table T_A that returns a record where column a_i has value x, or a write to table T_A to add a new record where column a_i has value x. Then, the right conjunct states that the same value x is subsequently persisted in table T_B of another database. This establishes $FK(b_j^{T_B}, a_i^{T_A})$, where column b_j in table T_B is a foreign key referencing column a_i in table T_A.

As mentioned, this generalizes to other types of datastores such as document-based or key-value stores, namely by generalizing the previous reads and writes as any operations where the shared value x is assigned to an attribute or an underlying

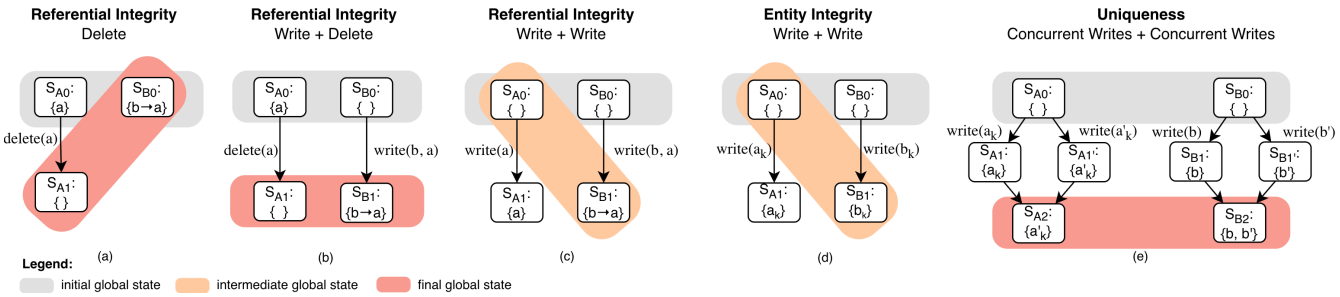


Figure 2: Examples of operations on two databases that may break integrity constraints. Orange and red states are inconsistent intermediate and final states, respectively. In (e), both databases are weakly replicated, and a final state is obtained by merging divergent replicas [40].

field of the object that is passed as an argument in writes, creating a referential dependency.

Once we determined which foreign key relationships exist, we can identify, across all combinations of operation types, which are the ones that may violate the foreign key association condition defined by $FK(b_j^{T_B}, a_i^{T_A})$ when applied concurrently to the state of a pair of microservices. In our particular case, since our generic model only includes three types of operations, we are able to exhaustively enumerate all possibilities, thus gaining confidence in the completeness of the set of problematic scenarios we identified, which are shown in Fig. 2 (a)–(c). Each depicted scenario comprises two databases, managed by different microservices, and the bottom part of the figure shows how two concurrent operations (or a single operation in scenario (a)) cause the system to transition to a final database state. For simplicity, we use $b \to a$ in the figure to represent a record a that has a non-null column referencing another column of record b, establishing a foreign key.

Absence of cascading deletes (RI-1). Fig. 2(a) illustrates how delete operations may break the validity of data associations. The initial global state corresponds to $S_{\{A_0,B_0\}} = \{a, b \to a\}$, where a exists in database A, and b in database B contains a reference to a. The delete operation on database A to delete record a causes the database to transition from $S_{A_0} = \{a\}$ to $S_{A_1} = \{\}$, where record a no longer exists. In contrast, no operations are issued to database B, thus its final state is the same as the initial state $S_{\{B_0\}} = \{b \to a\}$. At the end, the final global state $S_{\{A_1,B_0\}} = \{b \to a\}$ contains a record b with a reference to a, but record a does not exist in database A, breaking referential integrity.

In a monolithic database, this could be seamlessly addressed through schema annotations such as ON DELETE CASCADE[2], which atomically propagates the effects of a delete operation to all records that reference the record being deleted. However, these mechanisms are not present in microservices due to their decomposition, and the absence of equivalent safeguards has been shown to be prevalent [13].

[2]We do not consider patterns involving updates to columns referenced by foreign keys. While ON UPDATE CASCADE can be used to preserve referential integrity in monolithic databases, updating referenced foreign keys is a convoluted approach, dependent on application semantics, and no real examples were observed in microservice codebases.

This leads us to an operation pattern describing this scenario that is formalized as follows.

$$\exists a_i \in T_A,\ b_j \in T_B, x \in \mathcal{D}, d \in \mathcal{O} : \mathrm{FK}\left(b_j^{T_B}, a_i^{T_A}\right) \wedge \\ [d_{T_A}(a_i = x)]_{req} \wedge \nexists d' \in \mathcal{O} : d'_{T_B}(b_j = x) \in req \quad \textbf{RI-1}$$

The first line sets the existence of a foreign key, establishing an association across tables T_B and T_A of different databases. Then, there must exist a request where a delete operation on table T_A is not followed by any subsequent delete operation on the associated column b_j of table T_B, with value x.

Concurrent operations (RI-2). The second set of operations pertains to concurrent delete and write operations across different databases, as illustrated in Fig. 2(b). The initial global state of the application is $S_{\{A_0,B_0\}} = \{a\}$, and a pair of concurrent operations is applied to each database: the deletion of record a is applied to the initial state $S_{A_0} = \{a\}$ of database A, resulting in $S_{A_1} = \{\}$, and the insertion of a reference from record b to a is applied to the initial state $S_{B_0} = \{\}$ of database B, resulting in $S_{B_1} = \{b \to a\}$. However, when we aggregate the final states of each database partition into a global state, we obtain $S_{\{A_1,B_1\}} = \{b \to a\}$, where record b references a, but a does not exist. Therefore, the foreign key is not valid, and referential integrity does not hold. (The same issue appears in non-partitioned, weakly consistent replicated databases, a context for which Bailis et al. [40] showed that delete operations are not $\mathcal{I}$-confluent under foreign key invariants, meaning that concurrent deletes and writes of references are unsafe, and thus require coordination to preserve the invariant.)

The code pattern that flags this potential problem corresponds to the existence of two requests that perform these conflicting operations, as expressed by the following:

$$\exists a_i \in T_A,\ b_j \in T_B,\ x \in \mathcal{D}, r, w, d \in \mathcal{O}, n \neq m : \\ \mathrm{FK}\left(b_j^{T_B},\ a_i^{T_A}\right) \wedge \quad \textbf{RI-2} \\ [r_{T_A}(a_i = x)\ \wedge\ w_{T_B}(\ldots,\ b_j = x)]_{\mathrm{req_n}} \wedge [d_{T_A}(\ldots)]_{\mathrm{req_m}}$$

Here, request req_n performs a write operation on table T_B to create a new association in column b_j referencing an existing record with value x. This value is obtained from a preceding read operation on table T_A, confirming that the precondition for referential integrity is satisfied. Then, a concurrent request, req_m, executes a delete operation on table T_A, resulting in

the removal of a record that violates the referential integrity constraint established by the first request.

Uncoordinated replication (RI-3). Concurrent write operations may also affect referential integrity, namely when this constraint is enforced among objects partitioned across different storage systems.

To illustrate this, consider the scenario in Fig. 2(c), with an initially empty global state $S_{\{A_0,B_0\}} = \{\}$. To establish an association between objects, a request must contain two writes: one that inserts record a in database A, and another that inserts a reference from record b to a in database B. Together, these lead to the final global state of $S_{\{A_1,B_1\}} = \{a,\ b \to a\}$. However, before reaching that state, there may exist a visible intermediate global state $S_{\{A_0,B_1\}} = \{b \to a\}$ where b references a, but a does not yet exist, breaking referential integrity.

This violation stems from the lack of coordination among datastores when replication is asynchronous. In such cases, objects written in the same request may not become visible together. As a result, foreign key references can temporarily point to missing records, breaking referential integrity.

(RI-3) below formalizes the operations that can lead to this class of intermediate states with integrity violations.

$$\exists a_i \in T_A,\ b_j \in T_B,\ x \in \mathcal{D}, r, r', w, w' \in \mathcal{O}, n \neq m:$$
$$\mathrm{FK}\left(b_j^{T_B},\ a_i^{T_A}\right) \wedge \left[w_{T_A}(a_i = x,\ \ldots) \wedge\ w'_{T_B}(\ldots, b_j = x)\right]_{\mathrm{req_n}}$$
$$\wedge \nexists w'' \in \mathcal{O} : w''_{T_A}(a_i = x) \in req_m$$
$$\wedge \left[(\ldots,\ b_j = x) \leftarrow r_{T_B}(\ldots)\ \wedge\ r'_{T_A}(a_i = x)\right)$$
$$\vee\ \left(r_{T_A}(a_i = x) \wedge\ r'_{T_B}(b_j = x)\right]_{\mathrm{req_m}}$$
RI-3

First, there must exist a foreign key in database B referencing an entity in database A. The pattern then starts with a set of two writes in the same request (line 2): w_{T_A} that inserts the object to be referenced, and w'_{T_B} that creates an association with the former. The other request (lines 3-5) begins by excluding the write of the referenced object (line 3). Otherwise, under common read-your-writes semantics, the write would become visible to the subsequent reads, and the reference to the written record would remain valid, preserving integrity. Lines 4-5 describe the two alternative ways in which the same key x may read from table T_B (where it is a foreign key) and from table T_A, potentially exposing a missing reference. The first pair of reads (line 4) retrieves record b with $r_{T_B}(\ldots)$ using an arbitrary key to extract the foreign key value x in the record's column b_j, and then uses that foreign key to read record a with $r'_{T_A}(a_i = x)$. The second pair (line 5) reads record a containing the field x (through $r_{T_A}(a_i = x)$), and then reads a record b holding the value x as a foreign key with $r'_{T_B}(b_j = x)$.

3.3 Entity Integrity Violations

Entity integrity — which states that unique and non-null keys must identify records in a table — is enforced by databases through the use of primary keys. However, in microservices, entities may be partitioned among different services and share the same common primary key, allowing each database to uniquely identify its portion of the partitioned record. In this scenario, these various databases cannot guarantee consistency between themselves, and this lack of coordination can affect the overall application correctness.

Uncoordinated replication (EI-1). The effects of uncoordinated replication are akin to those in the context of referential integrity where, because writes are not coordinated across datastores, the validity of primary keys may be compromised.

Fig. 2(d) illustrates this through a scenario where an entity is decomposed into smaller entities in databases A and B, each managed independently by different services.

Both databases A and B are replicated with eventual consistency with initially empty states $S_{A_0} = \{\}$, $S_{B_0} = \{\}$. Then, two writes are executed. $write(a_k)$ is applied to database A to insert record a with a primary key k, and $write(b_k)$ is applied to database B to write record b using the same primary key k. Before reaching the final global state, there may exist an intermediate global state $S_{\{A_0,B_1\}} = \{b_k\}$ where record b exists, but record a does not. At this moment, the primary key k can be used to identify record b but not record a, and the primary key becomes invalid across the application.

The following pattern captures the microservice operations that may expose this problematic intermediate state.

$$\exists a_i \in T_A,\ b_j \in T_B,\ x \in \mathcal{D}, r, r', w, w' \in \mathcal{O}, n \neq m:$$
$$\mathrm{PK}_{T_A}(a_i)\ \wedge\ \mathrm{PK}_{T_B}(b_j) \wedge\ \left[w_{T_A}(a_i = x) \wedge\ w'_{T_B}(b_j = x)\right]_{\mathrm{req_n}}$$
$$\wedge \nexists w'', w''' \in \mathcal{O} : w''_{T_A}(a_i = x) \in req_m \wedge\ w'''_{T_B}(b_j = x) \in req_m$$
$$\wedge\ \left[r_{T_A}(a_i = x)\ \wedge\ r'_{T_B}(b_j = x)\right]_{\mathrm{req_m}}$$
EI-1

There are two primary key constraints: one on field a_i in table T_A of database A, and another on field b_j in table T_B of database B. The pattern then starts with a set of two writes within the same request (line 2): $w_{T_A}(a_i = x)$ and $w'_{T_B}(b_j = x)$, splitting a single record across T_A and T_B using the same primary key (line 2).

Then, the problematic request (lines 3-4) begins by excluding a write of the two records in the same request as the reads (line 3); otherwise, similarly to pattern RI-3, these newly written values would be visible under read-your-writes. With this exclusion in place, two read operations $r_{T_A}(a_i = x)$ and $r'_{T_B}(b_j = x)$ reading the same primary key value x may expose an inconsistent state of the application, containing only one of the two prior writes (line 4).

3.4 Uniqueness Violations

Similarly to primary keys in entity integrity constraints, databases can individually enforce uniqueness constraints, which can be defined at the column or table level.

However, when entities are decomposed across partitioned systems, the per-database mechanisms to enforce uniqueness may fail to preserve the overall data integrity of applications.

Conflicting writes (Un-1). This entails the occurrence of concurrent writes that conflict in a partitioned database. Consider Fig. 2(e) where databases A and B are both replicated with weak consistency, but only database A has a uniqueness constraint in column k. Then, two clients, c_1 and c_2, access the application from different regions and each issues a request to the application ($write(a_k)$ and $write(b)$ from c_1, and $write(a'_k)$ and $write(b')$ from c_2). We use a_k and a'_k to denote write operations that insert records a and a' using the same key in column k. In this specific execution, the request from client c_1 leads to states $S_{A_1} = \{a_k\}$ and $S_{B_1} = \{b\}$, whereas the request from client c_2 leads to states $S_{A'_1} = \{a'_k\}$ and $S_{B'_1} = \{b'\}$.

Since the same column k was written concurrently to different replicas, a weakly consistent database must be able to merge these writes through a mechanism such as LWW [35, 36] or CRDTs [34]. With the correct mechanisms, this convergence will be automatic and preserve uniqueness constraints, e.g., applying LWW in database A can discard the effects of $write(a_k)$ and preserve the uniqueness of the key in column k in a final state $S_{A_2} = \{a'_k\}$. However, despite data integrity being preserved in A, the application data is now inconsistent across databases A and B. This is because the final global state $S_{\{A_2,\, B_2,\}} = \{a'_k, b, b'\}$ includes the effects of both initial operations on B, but only one operation on A, meaning that the effects of one of the requests are only partially reflected.

To detect such inconsistencies, the following pattern captures requests that include two writes and at least one of them is on a constrained column.

$$\exists a_i, a_u \in T_A,\ b_j \in T_B,\ k,\ v \in \mathcal{D}, w, w', w'' \in \mathcal{O}:$$
$$UNIQ_{T_A}(a_u)\ \wedge \Big[\big(w_{T_A}(a_u = k)\ \vee\ w'_{T_A}(a_i = k,\ a_u = v)\big)$$
$$\wedge\ w''_{T_B}(b_j = k)\Big]_{req} \qquad \textbf{Un-1}$$

Here, there is one write w_{T_A} or w'_{T_A} to a database A that has some uniqueness constraint (line 2, split into inserting value k either (1) in a uniquely constrained column a_u or (2) in any column a_i and another value v in a uniquely constrained column a_u) and another write w''_{T_B} to database B that uses one of the values written in the former operation (line 3), regardless of whether the value is part of the uniqueness constraint. The idea is that, whenever there is a data dependency between the two writes (in this case, through the shared value k) and one of the databases holds a uniqueness constraint, then the overall data integrity can break, as the effects of the constrained operation may be lost upon applying the convergence mechanism, contrary to the other operation.

4 Aletheia

The formal framework from the previous section characterizes code patterns that lead to data integrity violations in microservices. However, manually inspecting codebases to identify these patterns becomes increasingly difficult as applications grow in size and complexity. To address this challenge, we introduce *Aletheia*, a static analysis tool that operationalizes our formal framework to automatically detect integrity violations.

Aletheia has four high-level steps. It begins with an *intra-procedural* analysis based on SSA graphs (§4.1), followed by an *inter-microservice* analysis that constructs an *abstract call graph* (§4.2). This is then used to extract a schema for objects stored across the application's databases (§4.3). Finally, Aletheia identifies the code sections that violate integrity constraints (§4.4). We now detail each of these steps.

4.1 Intra-Procedural Analysis

Statically identifying foreign keys through implicit microservice data associations as well as problematic code patterns requires reasoning about how values assigned to variables flow through the program, as we need to identify the provenance of the inputs to database operations. To this end, we first perform an intra-procedural analysis whose goal is to determine, within each method, how values assigned to variables flow into database operations and are used as input parameters in remote service invocations. The latter is necessary to subsequently broaden the scope of the analysis across services, ultimately reaching different databases.

In particular, we conduct a data flow analysis on graphs based on the SSA [15] intermediate representation of programs, where each variable has exactly one definition site, and new assignments result in new versions of these variables. This identifies the locations where variables are defined and subsequently used while also providing a standardized view of the program that abstracts away unnecessary details. In this step, we also extend the analysis across procedure invocations but within the same microservice, thus obtaining a per-service data flow representation.

Building SSA graphs. These graphs are constructed from the SSA intermediate representation of each method in the microservice code, with nodes corresponding to SSA instructions. We illustrate how the SSA graph allows for tracking data flow through an example of a *product service* (see Listing 1), which takes the *id* and *name* of a product as parameters to be written to its relational database, creates a new composite product variable from the parameter values, and returns the product. Fig. 3 depicts a simplified version of the corresponding SSA graph. In total, there are eight nodes that correspond to the instructions from the SSA representation. In this example, the *id* and *name* parameters are used as input to the database write operation, which is represented by a variable t_0. Variable t_1 is associated with the newly created *product* object with its fields *id* and *name*, respectively t_2 and t_3. The *product* constructor assigns to t_2 and t_3 the values of the parameters *id* and *name*, creating the nodes $*t_2 = id$ and $*t_3 = name$, which are connected to t_2 and t_3.

Tainting SSA graphs. After constructing the SSA graphs for each function, we have the information to determine which variables in the function flow into database operations and

```
1 func RegisterProduct (id, name) (Product, error) {
2     product_db.Exec("INSERT INTO products (id, name)
          VALUES(?, ?)", id, name)
3     product := Product{id: id, name: name}
4     return product, nil
5 }
```

Listing 1: Snippet of Go code for *register product* method of *product service* for an e-commerce application.

microservice invocations, either directly as arguments or indirectly through other assignments and operations. This is accomplished by employing a taint-propagation approach, whose first step is to identify nodes corresponding to these operations, and then proceed to discover all variables whose values flow into these operations. For this, we annotate variables used both as input parameters and return values of those operations. These annotations identify the call, along with the interface parameter (in the case of microservice invocations) or schema field (in the case of database operations) to which variables flow. Then, we propagate these annotations to all reachable nodes in the graph. Hence, whenever a variable contains such annotations, we can determine that its value flows into the corresponding operations.

Fig. 3 illustrates how this technique determines which database operations each function variable flows into. Through graph traversal, the system identifies variable t_0 as a database operation. Then, since *id* and *name* are passed as arguments, they are annotated, and taint information is propagated to the neighboring nodes, following the edge flow in the figure. Both taints on *id* and *name* eventually reach the local composite *product* defined by t_1, indicating that its underlying fields *id* (denoted by $_obj.id$) and *name* (denoted by $_obj.name$) had been written in the *product_db* database on the *product* table and *id* and *name* columns, respectively.

The previous process determines, for this particular function, which variables flow into the write operation (implemented as a SQL insert on *product_db*) whenever the respective service is invoked. Furthermore, if the service makes multiple internal function calls, we extend this analysis by propagating taint information across the SSA graphs of the various methods. To this end, we determine the final taints (and thus their triggering operations) of objects that reach microservice boundaries (i.e., other service invocations). In our example, these are the *id* and *name* parameters, as well as the product t_1 returned by the function, whose taints identify the write operation on *product_db* to which they flow. This enables a subsequent inter-service analysis of taint propagation, specifically how the parameters *id* and *name* supplied by the invoking service are transformed by the remote service invocation, which returns a value t_1 that incorporates taints generated during the current analysis.

Note that this approach not only supports raw SQL, which explicitly specifies record columns, but also NoSQL databases such as key-value or document stores, and queues: by propagating taint annotations through variables and their fields,

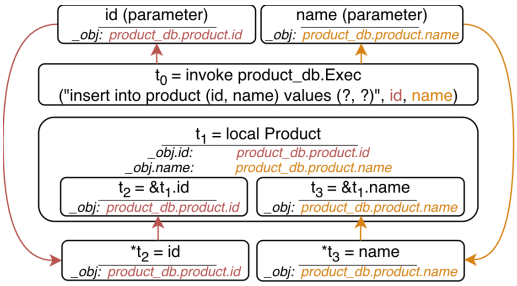


Figure 3: Example of a simplified SSA graph for a *product service*. Arrows represent the flow of taints originating in the write operation on t_0. Nodes affected by the taint contain the corresponding annotation below their SSA instruction. The standard execution flow arrows from the SSA graph are omitted.

we can identify, for a given object used as an argument in a call, the specific object fields that are referenced. The tainting also extends to data passed as binary objects (as employed by queues) as long as writers and readers use data types whose fields can be statically identified (e.g., maps with statically known keys) to construct and parse the binary objects.

4.2 Inter-Service Propagation

The second step broadens the analysis so we can detect when the patterns from §3 occur across multiple microservices. The main challenge is that we cannot simply continue the SSA taint analysis beyond individual services: this does not scale to large codebases, since it would require iterating over the joint SSA graphs of potentially thousands of microservices. Instead, the second step operates at a higher level of abstraction by building an *abstract call graph*, which provides a scalable representation of applications while keeping minimal taint information from the previous SSA graphs. The idea is that the *abstract call graph* only retains, per service, the information about interaction points, namely microservice calls and database operations, and the variables involved in them, including incoming parameters and returned values of these calls. This is the necessary and sufficient information to do a similar analysis to the first phase, namely to detect the patterns formalized in §3, even if they occur across microservices.

Building *abstract call graphs*. This graph comprises nodes representing microservice endpoints or datastore instances, and edges that denote invocations from one node to another. Edges between services represent remote invocations from one service to another. Edges connecting services and datastores represent either a service performing a datastore operation or a datastore triggering the execution of a service through, for example, an asynchronous event.

Furthermore, to track the taint flow through invocation or return arguments, we represent parameters and return values of either microservice or database operations as *abstract objects*. These objects contain the taint information obtained in the previous SSA analysis, which indicates the service or datastore calls into which variables flow. These objects are attached to nodes or edges, depending on whether they are

parameter/return variables associated with method declarations or used in call sites, respectively, as explained through the next example.

Fig. 4 illustrates a simplified *abstract call graph* for an e-commerce application containing three services: *APIService*, *ProductService*, and *InventoryService*. The logic of the *ProductService* and its *RegisterProduct* method closely follows the example introduced in §4.1 and the SSA graph in Fig. 3. The graph consists of five nodes, three for the interfaces exposed by each service (*APIService*, *ProductService*, and *InventoryService*), and two for the database instances (*product_db* and *inventory_db*). There are four *abstract objects*, namely the parameters and returned values involved in the example, annotated with taint information from the preceding intra-service analysis. These objects mediate the propagation of database and service invocation taints across nodes, allowing us to identify how data flows through services and operations.

The flow of interactions (solid arrows in Fig. 4) starts when the client uses the *APIService* interface to register a product, passing its *id*, *name*, and the *amount* to add to the inventory. Within this method, the *APIService* calls the *RegisterProduct* method in *ProductService*, which subsequently writes to its database *product_db* and returns a *product* object ao_2. Then, the *APIService* uses the *id* field in product ao_1 as the *id* argument (ao_3) to call the *AddInventory* method in *InventoryService*. The latter then uses the *id* parameter (ao_4) and *amount*, which we omit, as a value in the write operation to its database *inventory_db*.

Tainting *abstract call graphs*. The *abstract call graph* contains taint information extracted from SSA graphs, but this only captures the flow of objects within the boundaries of each microservice, which is insufficient for an analysis across microservices. Thus, after constructing the *abstract call graph*, the next step is to traverse the graph from each entry point in the application to propagate taints across the invocation chain. This will enable us to determine which objects, exchanged between services, hold taints from different databases, indicating a data dependency between services. This in turn will enable both detecting foreign keys and reasoning about common values used in different operations, as required by the formal model we previously defined.

To achieve this, for every node in the graph, we identify its outgoing edges and propagate the taints present in the objects associated with the call performed by the current node (either as parameters or return values) into the corresponding objects of the callee method declaration. Then, for each *abstract object* that acquires new taints, the algorithm propagates them to subsequent database and service calls the object flows into by merging annotations.

To illustrate this, we return to our example in Fig. 4. Here, we begin the tainting process by visiting the *APIService* on *RegisterProduct*, and then proceed to subsequent nodes according to the outgoing edges. The first database tainting propagation (red dashed arrows in Fig. 4) occurs when we finish visiting the *ProductService* and return the analysis to the *APIService*, propagating the taints from ao_2 to ao_1 through the object returned by the *RegisterProduct* method. This propagation results in a new annotation on ao_1 indicating a write in *products_db*, meaning that the value of ao_1, provided by the *APIService*, flows into the database of *ProductService*.

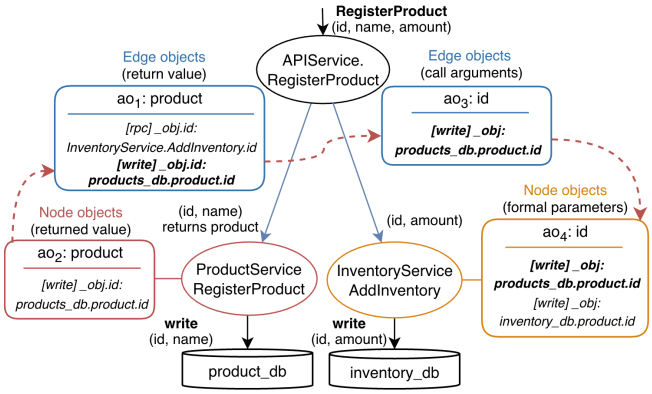


Figure 4: Example of a simplified *abstract call graph* for an e-commerce application. *Abstract objects* are prefixed with *ao*. Taint annotations in bold correspond to new annotations computed during the iteration of the *abstract call graph*. Solid arrows describe execution flow. Dashed arrows describe taint propagation.

Then, the graph traversal returns to the *APIService*. Since ao_1 contains an *rpc* annotation indicating that its *id* field is used as a parameter in the *AddInventory* call, then the new *write* taint on ao_1 is propagated to the corresponding object in that call (ao_3). Finally, when visiting the *InventoryService*, we see that object ao_4 is used as an *id* parameter of the method invoked by the *APIService*. Therefore, following the same logic as in *ProductService*, we propagate the taints from ao_3 to ao_4 by matching caller-callee objects, thus completing the taint chain from *product_db* to *inventory_db*.

This step of connecting the two databases through two taint annotations on object ao_4, is fundamental for our next analysis. For instance, in our example, we observe a group of operations matching pattern RI-3 (line 2) for referential integrity, where related records (*product* and *inventory*) are written in the same request, but may not be visible together during reads due to the lack of coordination. Then, when the detector module identifies another request executing reads on both the newly created records, completing the RI-3 pattern, it identifies a referential integrity violation.

4.3 Extracting the Application Data Schema

The two previous components allow Aletheia to determine how values are exchanged between services and included in different database accesses. This allows us to determine the schema information and search for the access patterns defined by our methodology.

Computing foreign keys. During the graph analysis, any taint propagation that causes an *abstract object* to contain multiple taints originating from different datastore operations results in a foreign key being added to the schema. However,

we still need to determine the direction of the foreign key association, since taints capture the order and type of operations, but do not indicate which field references the other. To determine which table holds the foreign key and which table contains the referenced records, we account for (i) the types of operations, which tell us whether we are obtaining a possible foreign key (through reads or writes whose values are propagated to other operations) or merely storing it (through writes), as well as (ii) the objects used as keys in these operations, whose taints specify the database fields being accessed and for which the foreign key will be constructed.

To this end, we denote each pair of operations as an ordered tuple (op_1, op_2), where $op_i \in \{read, write\}$ and *write* denotes any mutating operations.

Since foreign keys establish a data dependency between two tables, there is always one object whose value is included in operations to both tables, whose taints indicate the fields accessed in each table. Therefore, we use $field_1$ and $field_2$ to denote the fields accessed by that same object in op_1 and op_2, respectively. After having determined the fields being accessed for each operation, we can construct foreign key relationships according to the rules below for the pair (op_1, op_2). This is based on the intuition that if one of the operations is a read and the other is a write, then the write inserted a reference to the field that is read, independently of whether it appears before or after the write (① and ②). In case of two writes (③), since they occur when both the referenced and referencing objects are written in the same request, we assume that the second write inserts a reference to the former, which has just been created.

① if $(read, write)$, then $field_2$ references $field_1$
② if $(write, read)$, then $field_1$ references $field_2$
③ if $(write, write)$, then $field_2$ references $field_1$

For $(read, read)$ pairs, the direction depends on whether the propagated value is a key that was used as a parameter in a read operation ($read_{key}$) or a returned value ($read_{val}$). A field returned as $read_{val}$ and subsequently used as $read_{key}$ (④) is considered the referencing field, as it provides the key that was retrieved and then subsequently used as a foreign key to look up a value. When both operations use $read_{key}$ (⑤), we cannot infer the direction exclusively from the operation that reads a foreign key. Therefore, based on the order of operations, we assume that the first read retrieves the referenced field directly using a foreign key, while the second read looks up the record holding that foreign key.

④ if $(read_{val}, read_{key})$ then $field_1$ references $field_2$
⑤ if $(read_{key}, read_{key})$, then $field_2$ references $field_1$

4.4 Pattern Detection

The role of the detector module is to search for problematic code patterns that violate data integrity. This is enabled by the previous steps: constructing the data schema to extract data integrity constraints (see §4.3) and performing taint propagation in the *abstract call graph* to determine the flow of objects into service invocations and database operations (see §4.2).

With this information, the detector only needs to traverse the *abstract call graph*, locating the database operations, and checking for the presence of the problematic operation patterns defined in §3 (RI-1, RI-2, RI-3, EI-1, Un-1).

4.5 Implementation

Aletheia was implemented in Go using its Go SSA IR [41]. Thus, it can be extended to support other programming languages, since SSA representations can be obtained for multiple languages using analysis or compiler frameworks [42, 43].

Aletheia relies on three important mechanisms to analyze microservice codebases: identifying service methods that implement remote invocation endpoints, the locations where these invocations occur, and locating database accesses. However, microservices may employ a range of frameworks and communication protocols to interact with one another, and also use a variety of libraries and drivers tailored to each specific storage system. This diversity makes it difficult to encompass all frameworks and thus to locate these calls.

Given these difficulties, we analyze applications written with Blueprint [24], which identifies this information. In particular, Blueprint implements microservice invocations as regular function calls, and database operations through interfaces that abstract the underlying storage, thus allowing us to easily locate both of them. In the future, Aletheia could be extended to support other frameworks [44, 45] besides Blueprint, as long as they provide standardized abstractions for database and microservice calls that can be statically identified.

5 Evaluation

We evaluate our system to answer the following questions:
- Can Aletheia detect data integrity violations effectively?
- Which data integrity violations were found by Aletheia?
- How does application complexity impact analysis time?

5.1 Case Studies

We evaluated Aletheia using a dual approach that combines existing open-source microservice codebases (denoted as *realistic applications*), and production trace-based generated code (denoted as *synthetic applications*) to compensate for the lack of production-scale open-source microservice collections while evaluating the scalability of our approach.

Realistic applications. We conducted a broad search for open-source microservice applications across multiple programming languages. Aletheia was evaluated using the identified applications, namely seven microservice applications spanning four domains: e-commerce (Digota, SockShop, ESHopMicroservices), social networks (PostNotification, SocialNetwork), media storage (MediaMicroservices), and booking systems (TrainTicket) [18–23]. The SocialNetwork and MediaMicroservices applications (both part of DeathStarBench [18]) and TrainTicket [19] correspond to benchmarks

Name	#MS	#DS	#LoCs	#RPCs	#CGs	Mem.	RI-1	RI-2	RI-3	EI-1	Un-1	TP	FP	FN
Digota(1) [21]	4	4	1179	27	19	266 MB	2	2	0	0	0	4	0	0
EShopMicroservices(1) [22]	5	5	875	33	23	257 MB	2	6	0	0	0	8	0	2
Sockshop [23]	8	6	1676	49	19	291 MB	3	3	0	0	0	0	6	0
PostNotification(1) [20]	3	2	173	6	3	228 MB	0	0	1	0	0	1	0	0
SocialNetwork [18]	13	10	1374	31	6	341 MB	0	0	3	0	0	3	0	0
MediaMicroservices(1) [18]	14	19	1480	54	12	265 MB	0	0	3	1	1	5	0	0
TrainTicket [19]	31	21	4462	170	58	285 MB	14	17	4	0	0	29	6	20

Table 2: Static analysis results per application. For each application, we report: (1) characteristics such as microservices (#MS), datastores (#DS), lines of code (#LoC), RPC calls (#RPCs), and call graphs (#CG, equaling the number of frontend requests); (2) Aletheia's peak memory usage (Mem.); and (3) detection errors broken down by integrity violation pattern, and its true positives (TP), false positives (FP), and false negatives (FN). Applications marked with (1) were ported to Blueprint.

that approximate large microservice applications in real deployments. The overall set of applications (see overview in Table 2) involve typical microservice operations such as managing product catalogs and orders, creating posts and notifications, storing media content with reviews, and handling ticket reservations with associated services. Applications employ various storage systems, including document stores (MongoDB), message brokers (RabbitMQ), caches (Redis, Memcached), and relational databases (MySQL), with system sizes ranging from 3 to 31 microservices, 2 to 21 datastores, and up to 4462 lines of code. Communication spans 6 to 170 RPC calls in 3 to 58 call graphs.

In two cases, we extended applications with missing logic that seemed intuitively necessary, without which some problematic data partitioning could not be uncovered. In particular, we extended MediaMicroservices by adding endpoints (i) to insert complete movie records in a single request, instead of having the client separately calling four endpoints to write the associated data, and (ii) with a previously missing operation to read movie details, which is necessary to obtain an ID that was writable but not accessible to the client, and is required to read movie pages. In TrainTicket's implementation, ticket reservations trigger the creation of insurance, consignment, food order, and delivery records, but the system was missing an operation to access this state as part of a reservation query. Hence, we added an endpoint to query all records for a given reservation through existing read endpoints.

Synthetic applications. To demonstrate that Aletheia scales to production-like scenarios and to assess how the call graph structure influences performance, we generated synthetic applications based on graph characteristics (call depth, fan-out, and request volume) derived from Alibaba's production microservice traces [12] (see Table 3). Since there is no public access to the service logic, we generated microservice stubs with a configurable number of write and delete operations per method. Additionally, to assess the sensitivity of the results to varying graph parameters, we created five variants that modified one of the call graph characteristics at a time.

Experimental Setup. The experiments were conducted on 2x Intel Xeon Gold 5320 CPUs at 2.2GHz with 52 cores and 256GB of RAM, and all results represent an average of 5 runs.

5.2 How Effective is the Detection in Aletheia?

Table 2 presents the analysis results across all realistic applications. In our methodology, we computed an approximate false negative rate by manually and extensively searching for the patterns of operations defined in §3 to determine an approximation of the ground truth for the total number of problematic code patterns. In our analysis, we refer to the programming issues that may lead to data inconsistencies as bugs, while acknowledging that, in some cases, the application logic may tolerate the resulting semantic anomalies, or work around those issues through careful database configuration.

Aletheia achieved 81% precision and 69% recall, identifying 50 true positives, 12 false positives, and 22 false negatives. Notably, 46 of these true positives represent previously unreported bugs, with the exceptions being PostNotification and SocialNetwork, whose bugs were previously identified as *cross-service inconsistencies* through a runtime consistency enforcement shim [20].

We reported all bugs to the developers of the respective applications. All bugs were confirmed for TrainTicket, Digota, PostNotification, SocialNetwork, and MediaMicroservices by the respective authors. The developers of EShopMicroservices replied only to an initial report on RI-1, but confirmed the bugs and suggested solutions to address them. Furthermore, the authors of TrainTicket indicated these issues would be further investigated and addressed in future implementations due to their relevance in the application.

False positives. We examined the sources of false positives and how developers can interpret these warnings. The primary source of false positives is Aletheia's conservative approach, which flags patterns as potential violations even when application-specific business logic ensures safety: SockShop produces 3 RI-1 false positives from cart deletions that intentionally do not cascade to orders, and 3 RI-2 false positives from concurrent cart deletions and order creation, where the application coordinates safely. TrainTicket exhibits similar patterns where deleting contacts or price configurations does not propagate to reservations, which is correct by design but conservatively flagged, resulting in 3 RI-1 false positives.

A second source of false positives arises from over-approximation in taint tracking. Aletheia propagates taints to all reachable variables, so when variables tainted by differ-

ent database operations are combined in expressions (e.g., `x←a+b`), the result inherits all taints. This can lead to incorrectly inferred foreign key relationships. In TrainTicket, 3 of 35 warnings (1 for RI-1, 2 for RI-2) are such false positives.

To reduce the number of false positives, Aletheia allows developers to suppress cascade warnings for entities not intended for deletion and to be ignored by the detection module.

False negatives. False negatives occur when query patterns prevent foreign key detection. Some applications retrieve related objects by using values as query filters rather than as direct foreign key lookups. For instance, TrainTicket retrieves price configurations by filtering on train type name instead of using a foreign key to access the related object. This prevented Aletheia from identifying 3 foreign key relationships, resulting in 8 undetected bugs (3 for RI-1, 5 for RI-2).

Another source of false negatives arises from *cross-request implicit associations*, where relationships between entities exist outside of the microservice code, and are established through user-provided values that bridge arguments of different RPC requests. For instance, TrainTicket allows users to query station information in one request, then use the returned station name as input for order creation in a subsequent request. Since the order service never queries the station database using the value from the user, but the order creation populates related records (DELIVERY, FOOD-ORDER, ROUTE, TRIP) with this station name, the connection between STATION and these records exists only across separate requests rather than within application code, leaving foreign keys undetected. This limitation caused 12 false negatives in TrainTicket (5 for RI-1, 7 for RI-2). Similarly, a missing foreign key between COUPONS and PRODUCTS caused 2 false negatives in EShopMicroservices (1 for RI-1, 1 for RI-2).

The results in Table 2 show that larger codebases (measured by number of microservices, datastores, and LoC), contain more potential integrity violations. For example, TrainTicket, with 170 service invocations, contains almost eight times more violations than Digota, with 27 invocations. This is because more service invocations and cross-datastore interactions create additional data associations across services, increasing violation likelihood. Importantly, this trend validates Aletheia's relevance: as applications grow in complexity, the number of integrity violations rises substantially, making automated detection essential where manual debugging is challenging.

5.3 Which Data Integrity Violations were Found by Aletheia?

Next, we present a detailed description of the detected bugs.

Digota. Four referential integrity bugs were detected (2 for RI-1 and 2 for RI-2). For RI-1, deletions of PRODUCTS are not propagated to their SKUs (Stock Keeping Units), and deletions of SKUs are not propagated to ORDERS. Although similar patterns involving orders are false positives in other applications, invalid references from ORDERS to non-existent SKUs cause order returns to FAIL whenever the *order service* cannot retrieve the related SKUs, preventing refunds from completing. For RI-2, we found that a PRODUCT can be concurrently deleted while an SKU is created, and an SKU can be concurrently deleted while ORDERS referencing it are created.

EShopMicroservices. Eight referential integrity bugs were detected (2 for RI-1, and 6 for RI-2), which affect the correct processing of baskets. For RI-1, the two bugs occur when the deletion of PRODUCTS or COUPONS is not propagated to the BASKET that contains references to them. For RI-2, the six bugs occur when PRODUCTS or DISCOUNTS are deleted while a BASKET is concurrently created or updated.

PostNotification. One referential integrity bug was detected for RI-3 concerning a foreign key from a NOTIFICATION to its POST. The bug occurs when the *notify service* receives a new NOTIFICATION via a message broker and uses the POST ID to fetch the corresponding post in *storage service*. If replication is stale, users may receive notifications for non-existent posts.

SocialNetwork. Three referential integrity bugs were detected for RI-3 under asynchronous replication, and concern foreign key associations between timelines (USERTIMELINE and HOMETIMELINE) and POST. The bugs occur when POST IDS retrieved from timeline caches or the USERTIMELINE document store are later used to fetch POSTS from the *post-storage* database. Under stale replication, loading timelines may fail whenever one POST is unexpectedly missing from the *post-storage* database.

MediaMicroservices Five bugs were detected regarding referential integrity (3 for RI-3), entity integrity (1 for EI-1), and uniqueness (1 for Un-1) violations under asynchronous replication. For RI-3, reading a page may fail whenever foreign keys obtained from reading MOVIE-INFO and MOVIE-REVIEW are then subsequently used to retrieve the movie details (CAST, PLOT) and the review content (REVIEW-STORAGE), as any referenced record (all created together in the same request) may be missing. For EI-1, obtaining the movie ID and reading the page may fail as MOVIE-ID and MOVIE-INFO are not guaranteed to be visible simultaneously. For Un-1, concurrent movie registrations with the same title require conflict resolution in MOVIE-ID to preserve the uniqueness constraint, discarding one MOVIE-ID, thus leaving an orphaned record in MOVIE-INFO. Consequently, subsequent PAGE reads and REVIEW writes may observe partial results across MOVIE-ID and MOVIE-INFO.

TrainTicket. Twenty-nine referential integrity bugs were detected (10 for RI-1, 15 for RI-2, and 4 for RI-3). RI-1 bugs arise when deleting: (i) ORDERS without propagating effects to INSURANCE, CONSIGNMENT, DELIVERY, or FOOD-ORDER; (ii) STATIONS without propagating effects to ORDERS or CONSIGNMENTS; (iii) ROUTES without propagating effects to PRICE CONFIGURATIONS; (iv) TRIPS or USERS without propagating effects to ORDERS or CONSIGNMENTS. RI-2 bugs involve the previous entity pairs (except those in (i), as all records are created in the same request) and new pairs (ORDERS referencing CONTACTS and PRICE CONFIGURA-

#	Purpose	CD	FO	#DS	#CG	#MS	Mem
App1	Median	6	2	7	267	50	1.17 GB
App2	p90 #DS	6	2	55	267	50	1.70 GB
App3	p90 CD	12	2	7	267	500	9.94 GB
App4	p90 FO	6	16	7	267	500	10.08 GB
App5	p90 #CG	6	2	7	2887	50	11.32 GB

Table 3: Synthetic applications with varying call depth (CD), fan-out (FO), datastores (#DS, equaling the number of writes), and call graphs (#CG, equaling the number of frontend requests) according to Alibaba's report [12], as well as stateless microservice count (#MS).

TIONS), and occur when a record may be deleted while a new reference to it is created. RI-3 bugs occur when the ORDER ID is used to read the ORDER and query related INSURANCE, CONSIGNMENT, DELIVERY, and FOOD-ORDER records.

5.4 How Do Application Size and Complexity Impact Analysis Time?

We evaluate how application size and complexity affect the scalability of our approach. Fig. 5 shows the analysis times for both sets of applications, decomposed into *parsing*, *schema building*, and *detection* phases. *Parsing* includes code retrieval, SSA graph construction and tainting, and *abstract call graph* generation. *Schema building* taints the abstract graph while reasoning about data associations. *Detection* searches for the problematic patterns.

We start by analyzing applications in the realistic set, which complete in under 4 seconds. The analysis time increases mainly with the remote call count, whereas microservice and datastore counts have a smaller impact. For example, more than doubling the number of microservices from 13 (SocialNetwork) to 31 (TrainTicket) results in a smaller than 2× increase in analysis time (2.07 to 3.52 s). Breaking down into phases, *parsing* dominates total time, while *schema building* and *detection* are barely visible.

We now examine synthetic applications (Table 3) to understand how specific parameters influence the analysis time. Compared to realistic applications, the *schema building* and *detection* phases consume larger proportions of both time and memory. This reflects the substantially higher complexity of synthetic applications: up to 2887 call graphs vs. 58 maximum, with greater per-graph complexity from increased microservice and datastore accesses. The shift occurs because more operations generate larger volumes of taint information, increasing propagation overhead through invocation chains, and extending *schema building* and *detection* times.

Our detailed analysis of the synthetic set begins with App1, which represents the median Alibaba call graph parameters, and completes in 20 seconds. Comparing App2 with App1, increasing datastore operations from 7 to 55 per request results in a 3× increase in total time, demonstrating some sensitivity to database access frequency. Comparing the largest applications (App3 and App4) with App1 reveals the primary scalability driver. Despite using the same number of datastores, their 10× increase in microservices (from 50 to 500) produces a corresponding 10× increase in total invocations (from approximately 13k to 130k). This leads to analysis times reaching 801 and 908 seconds, respectively, confirming that total invocation count strongly predicts analysis time. App5 validates this finding by achieving high invocation counts through a different approach than App3/App4, namely maintaining a low microservice count (50) but increasing the number of call graphs (and thus frontend requests) from 267 to 2887, allowing it to reach similar total invocations as the larger applications. The resulting analysis time of 1152 seconds is comparable, confirming that invocation count drives analysis costs regardless of whether complexity arises from more services or more entry points.

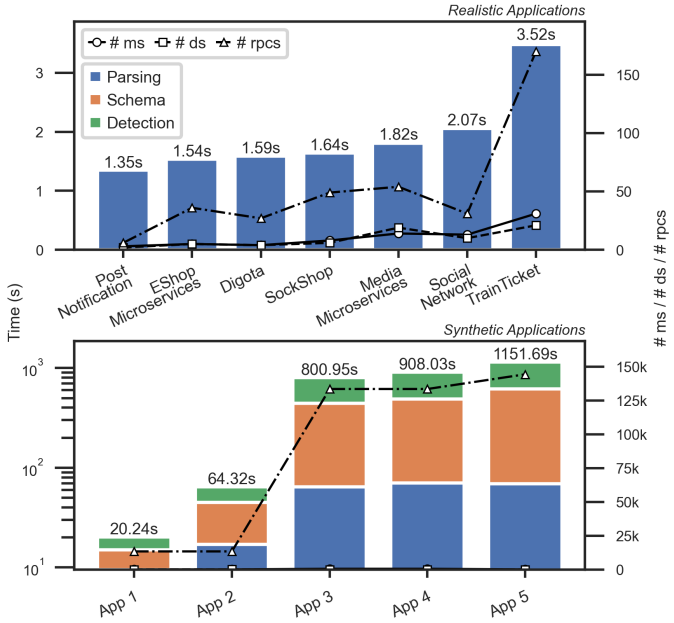


Figure 5: Analysis time per application, sorted by complexity (sum of the number of microservices and datastores). The reported analysis times are given for Total, Parsing, Schema Building, and Detection, where Total is the sum of the other three components.

Finally, comparing App3 and App4 reveals that the structural properties of the call graph have minimal performance impact: despite differing significantly in call depth (12 vs. 6) and fan-out (2 vs. 16), they show nearly identical *schema building* and *detection* costs.

6 Limitations

Parameterized dynamic queries. As a static analysis tool, Aletheia supports only statically defined queries and filters and cannot infer data relationships from dynamically constructed query operations. For instance, when SockShop dynamically adds tag name filters based on user input (`WHERE tag.name=?`), these relationships are not captured by our analysis. However, we did not find instances where this limitation affected the evaluation results for any application.

Cross-request implicit associations. When relationships between entities are established through user input that bridges separate RPC requests, Aletheia cannot infer them.

This caused a few false negatives in two services, as discussed in §5.2. In the future, we could address this by allowing developers to explicitly annotate association variables with foreign key metadata, enabling Aletheia to recognize these relationships while preserving its automated analysis.

Lazy enforcement of data integrity. Certain applications may handle data integrity lazily. For instance, when a user is deleted, dependent data may be retained and deleted later. This can be captured by Aletheia if the delete is executed upon an asynchronous event published as part of the triggering deletion. However, if deletions are propagated outside the request path (e.g., through a background process that periodically cleans *orphaned* records), developers would have to specify which cascade warnings to ignore during the analysis.

7 Related Work

Analyzing Microservices. Laigner et al. [13] conducted a study of data management practices in microservices, identifying fundamental challenges in ensuring application safety across services due to decentralized storage and application-layer logic. Their work provides a valuable high-level characterization of microservice consistency challenges through practical examples where applications can observe inconsistent states due to the lack of database support for transactional guarantees and event coordination. However, it lacks both a formal problem specification to systematically reason about integrity constraints, and automated detection mechanisms.

Romão et al. [46] recently introduced MAD, an anomaly detection tool that takes a monolithic service written in Java and a proposed decomposition, and uses an SMT solver to flag anomalies from non-serializable executions. While this shares the setting of detecting semantic inconsistencies in microservices, it has a different focus on anomalies in the migration into microservices, receiving as input the monolithic implementation of Java applications and a proposed decomposition, instead of a microservice codebase. Furthermore, the correctness criteria of serializable executions may be too restrictive, whereas Aletheia allows broader semantics and pinpoints only the problematic data integrity violations.

Peng et al. [37] introduced an architectural measurement approach through trace analysis, which allows users to identify architectural issues due to poor service decomposition, such as redundant services with similar functionality and invocation chains in which a single service is invoked multiple times. In contrast, we operate at the source code level to identify sections that violate data integrity constraints, which would remain undetected even when addressing architectural issues.

Hutcheson et al. [47] propose a methodology for the high-level understanding of microservice system architectures from service dependency graphs and context maps extracted through static analysis. This approach detects REST endpoints and calls without capturing how data values are exchanged, and, while it identifies overlapping service entities based on similarities, it does not capture the necessary data flow to infer foreign keys or data dependencies that affect integrity.

Addressing Inconstencies in Distributed Systems. FlightTracker [26] is a centralized solution that offers Read-Your-Writes (RYW) session guarantees with support for intra-session concurrency by storing the writes of users in the form of a *ticket* propagated throughout the request. Antipode [20] uses a distributed approach that enforces *cross-service causal consistency* (XCY) by tracking and propagating dependencies from writes in the form of lineages that capture the partial order of events. However, these are runtime consistency enforcement shims [48] for distributed systems. In contrast, our approach executes prior to runtime and identifies the specific causes of potential inconsistencies, enabling developers to decide whether the code should be modified to address them.

Modular Development Frameworks. Service Weaver [49] and Blueprint [24] are frameworks that allow users to write modular applications through regular function calls for inter-component communication, and later deploy them as microservices. Service Weaver [49] separates the logical boundaries (code) from the physical boundaries (deployment) by compiling and automatically deploying components. Blueprint [24] allows users to explore microservices across several dimensions, including application logic, middleware, and frameworks, enabling the reproduction of anomalies such as metastability failures [50], tail latency effects [51], and cross-system inconsistencies [14]. As such, both have a complementary goal of simplifying microservice development and can benefit from automatic detection of data inconsistencies.

8 Conclusion

Data decomposition in microservices creates challenges in maintaining integrity across physically separated but logically related data. This paper presents a formal framework to systematize integrity violations in microservices, identifying specific operation patterns that compromise constraints. We leveraged these foundations to build Aletheia, a static analysis tool for automated violation detection. We demonstrate that Aletheia detects previously unreported bugs in open-source microservice-based applications, while also showing good scalability in synthetic benchmarks. These results suggest that automated integrity verification can become a practical reality in microservice development toolchains.

Acknowledgements

We thank Jonathan Mace, our shepherd Shuai Mu, and the reviewers for their feedback. Work supported by national funds through Fundação para a Ciência e a Tecnologia, I.P. (FCT) under projects: UID/50021/2025 (https://doi.org/10.54499/UID/50021/2025), UID/PRR/50021/2025 (https://doi.org/10.54499/UID/PRR/50021/2025), SmartRetail (ref. C6632206063-00466847), GLOG (ref. LISBOA2030-FEDER-00771200, https://doi.org/10.54499/2023.18452.ICDT), and CCloud (ref. 2023.16986.ICDT, https://doi.org/10.54499/2023.16986.ICDT).

References

[1] Martin Fowler and James Lewis. Microservices, March 2014. Accessed June 2026. https://martinfowler.com/articles/microservices.html.

[2] Chris Richardson. *Microservices Patterns: With examples in Java*. Manning Publications, 2018. https://microservices.io/book.

[3] Johannes Thönes. Microservices. *IEEE Software*, 32(1):116–116, 2015. https://doi.org/10.1109/MS.2015.11.

[4] Jacopo Soldani, Damian Tamburri, and Willem-Jan Heuvel. The pains and gains of microservices: A systematic grey literature review. *Journal of Systems and Software*, 146, September 2018. https://doi.org/10.1016/j.jss.2018.09.082.

[5] Omar Al-Debagy and Peter Martinek. A comparative review of microservices and monolithic architectures. In *Proceedings of the 2018 IEEE 18th International Symposium on Computational Intelligence and Informatics (CINTI)*, pages 149–154, 2018. https://doi.org/10.1109/CINTI.2018.8928192.

[6] Antonio Bucchiarone, Nicola Dragoni, Schahram Dustdar, Patricia Lago, Manuel Mazzara, Victor Rivera, and Andrey Sadovykh. Microservices. *Science and Engineering. Springer*, 2020. https://doi.org/10.1007/978-3-030-31646-4.

[7] Miguel Diogo, Bruno Cabral, and Jorge Bernardino. Consistency Models of NoSQL Databases. *Future Internet*, 11(2), 2019. https://doi.org/10.3390/fi11020043.

[8] Dan Pritchett. BASE: An ACID Alternative. *Queue*, 6(3):48–55, May 2008. https://doi.org/10.1145/1394127.1394128.

[9] Pwint Phyu Khine and Zhaoshun Wang. A review of polyglot persistence in the big data world. *Information*, 10(4), 2019. https://doi.org/10.3390/info10040141.

[10] Philip A. Bernstein, Vassos Hadzilacos, and Nathan Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987. https://dl.acm.org/doi/book/10.5555/17299.

[11] Zhizhou Zhang, Murali Krishna Ramanathan, Prithvi Raj, Abhishek Parwal, Timothy Sherwood, and Milind Chabbi. CRISP: Critical path analysis of Large-Scale microservice architectures. In *Proceedings of the 2022 USENIX Annual Technical Conference (USENIX ATC '22)*, pages 655–672, 2022. https://www.usenix.org/conference/atc22/presentation/zhang-zhizhou.

[12] Shutian Luo, Xu Huanle, Chengzhi Lu, Kejiang Ye, Guoyao Xu, Liping Zhang, Yu Ding, Jian He, and Cheng-Zhong Xu. Characterizing microservice dependency and performance: Alibaba trace analysis. In *Proceedings of the ACM Symposium on Cloud Computing (SoCC '21)*, pages 412–426, 2021. https://doi.org/10.1145/3472883.3487003.

[13] Rodrigo Laigner, Yongluan Zhou, Marcos Antonio Vaz Salles, Yijian Liu, and Marcos Kalinowski. Data management in microservices: state of the practice, challenges, and research directions. *In Proceedings of the VLDB Endowment*, 14(13):3348–3361, September 2021. https://doi.org/10.14778/3484224.3484232.

[14] Phillipe Ajoux, Nathan Bronson, Sanjeev Kumar, Wyatt Lloyd, and Kaushik Veeraraghavan. Challenges to adopting stronger consistency at scale. In *Proceedings of the 15th Workshop on Hot Topics in Operating Systems (HotOS XV)*, 2015. https://www.usenix.org/conference/hotos15/workshop-program/presentation/ajoux.

[15] Ron Cytron, Jeanne Ferrante, Barry K Rosen, Mark N Wegman, and F Kenneth Zadeck. An efficient method of computing static single assignment form. In *Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL '89)*, pages 25–35, 1989. https://doi.org/10.1145/75277.75280.

[16] Omer Tripp, Marco Pistoia, Stephen J. Fink, Manu Sridharan, and Omri Weisman. Taj: effective taint analysis of web applications. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '09)*, pages 87–97, 2009. https://doi.org/10.1145/1542476.1542486.

[17] Yulei Sui and Jingling Xue. SVF: interprocedural static value-flow analysis in LLVM. In *Proceedings of the 25th International Conference on Compiler Construction (CC '16)*, 2016. https://doi.org/10.1145/2892208.2892235.

[18] Yu Gan, Yanqi Zhang, Dailun Cheng, Ankitha Shetty, Priyal Rathi, Nayan Katarki, Ariana Bruno, Justin Hu, Brian Ritchken, Brendon Jackson, Kelvin Hu, Meghna Pancholi, Yuan He, Brett Clancy, Chris Colen, Fukang Wen, Catherine Leung, Siyuan Wang, Leon Zaruvinsky, Mateo Espinosa, Rick Lin, Zhongling Liu, Jake Padilla, and Christina Delimitrou. An open-source benchmark suite for microservices and their hardware-software implications for cloud & edge systems. In *Proceedings of the 24th International Conference on*

Architectural Support for Programming Languages and Operating Systems (ASPLOS '19), pages 3–18, 2019. https://doi.org/10.1145/3297858.3304013.

[19] Xiang Zhou, Xin Peng, Tao Xie, Jun Sun, Chenjie Xu, Chao Ji, and Wenyun Zhao. Benchmarking microservice systems for software engineering research. In *Proceedings of the 40th International Conference on Software Engineering: Companion Proceedings (ICSE '18)*, pages 323–324, 2018. https://doi.org/10.1145/3183440.3194991.

[20] João Ferreira Loff, Daniel Porto, João Garcia, Jonathan Mace, and Rodrigo Rodrigues. Antipode: Enforcing cross-service causal consistency in distributed applications. In *Proceedings of the 29th Symposium on Operating Systems Principles (SOSP '23)*, pages 298–313, 2023. https://doi.org/10.1145/3600006.3613176.

[21] Digota. Digota - ecommerce microservice, 2018. Accessed June 2026. https://github.com/digota/digota.

[22] ASP.NET Run. EShopMicroservices, 2025. Accessed June 2026. https://github.com/aspnetrun/run-aspnetcore-microservices.

[23] OCP Power Demos. Sock Shop: A microservice demo application, 2024. Accessed June 2026. https://github.com/ocp-power-demos/sock-shop-demo.

[24] Vaastav Anand, Deepak Garg, Antoine Kaufmann, and Jonathan Mace. Blueprint: A toolchain for highly-reconfigurable microservice applications. In *Proceedings of the 29th Symposium on Operating Systems Principles (SOSP '23)*, pages 482–497, 2023. https://doi.org/10.1145/3600006.3613138.

[25] Uber. Introducing domain-oriented microservice architecture, July 2020. Accessed June 2026. https://www.uber.com/blog/microservice-architecture/.

[26] Xiao Shi, Scott Pruett, Kevin Doherty, Jinyu Han, Dmitri Petrov, Jim Carrig, John Hugg, and Nathan Bronson. FlightTracker: Consistency across Read-Optimized online stores at facebook. In *Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI '20)*, pages 407–423, 2020. https://www.usenix.org/conference/osdi20/presentation/shi.

[27] Mafalda Sofia Ferreira, João Ferreira Loff, and João Garcia. Rendezvous: Where serverless functions find consistency. In *Proceedings of the 4th Workshop on Resource Disaggregation and Serverless (WORDS '23)*, pages 51–57, 2023. https://doi.org/10.1145/3605181.3626290.

[28] Hector Garcia-Molina and Kenneth Salem. Sagas. In *Proceedings of the 1987 ACM SIGMOD International Conference on Management of Data (SIGMOD '87)*, pages 249–259, 1987. https://doi.org/10.1145/38713.38742.

[29] Emmanuel Cecchet, George Candea, and Anastasia Ailamaki. Middleware-based database replication: the gaps between theory and practice. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data (SIGMOD '08)*, pages 739–752, 2008. https://doi.org/10.1145/1376616.1376691.

[30] AWS. Amazon DynamoDB, 2026. Accessed June 2026. https://aws.amazon.com/dynamodb.

[31] Azure. Cosmos DB, 2026. Accessed June 2026. https://azure.microsoft.com/en-us/products/cosmos-db.

[32] Oracle. Oracle database, 2026. Accessed June 2026. https://docs.oracle.com/en/database/oracle/oracle-database/index.html.

[33] PostgreSQL. BDR Project, September 2020. Accessed June 2026. https://wiki.postgresql.org/wiki/BDR_Project.

[34] Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. Conflict-free replicated data types. In *Proceedings of the 13th International Conference on Stabilization, Safety, and Security of Distributed Systems (SSS '11)*, pages 386–400, 2011. https://dl.acm.org/doi/10.5555/2050613.2050642.

[35] EDB Postgres Distributed (PGD). Conflicts, 2025. Accessed June 2026. https://www.enterprisedb.com/docs/pgd/4/bdr/conflicts.

[36] Oracle. Oracle database advanced replication management API reference, 2025. Accessed June 2026. https://docs.oracle.com/cd/B12037_01/server.101/b10733/rarconfl.htm.

[37] Xin Peng, Chenxi Zhang, Zhongyuan Zhao, Akasaka Isami, Xiaofeng Guo, and Yunna Cui. Trace analysis based microservice architecture measurement. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '22)*, pages 1589–1599, 2022. https://doi.org/10.1145/3540250.3558951.

[38] Shamkant B Navathe. Evolution of data modeling for databases. *Communications of the ACM*, 35(9):112–123, 1992. https://doi.org/10.1145/130994.131001.

[39] Antoni Olivé. *Conceptual modeling of information systems*. Springer, 2007. https://doi.org/10.1007/978-3-540-39390-0.

[40] Peter Bailis, Alan Fekete, Michael J Franklin, Ali Ghodsi, Joseph M Hellerstein, and Ion Stoica. Coordination avoidance in database systems. *Proceedings of the VLDB Endowment*, 8(3):185–196, November 2014. https://doi.org/10.14778/2735508.2735509.

[41] Go. Go SSA Package, May 2026. Accessed June 2026. https://pkg.go.dev/golang.org/x/tools/go/ssa.

[42] WALA. T. J. Watson Libraries for Analysis (WALA), 2026. Accessed June 2026. https://github.com/wala/WALA/.

[43] C. Lattner and V. Adve. LLVM: a compilation framework for lifelong program analysis & transformation. In *International Symposium on Code Generation and Optimization (CGO '04)*, pages 75–86, 2004. https://doi.org/10.1109/CGO.2004.1281665.

[44] Spring. Spring Boot, 2026. Accessed June 2026. https://spring.io/projects/spring-boot.

[45] Microsoft. ASP.NET Core, 2026. Accessed June 2026. https://dotnet.microsoft.com/en-us/apps/aspnet.

[46] Valentim Romão, Rafael Soares, Luís Rodrigues, and Vasco Manquinho. Don't go MAD with anomalies! Design-time microservice anomaly detection in migration to microservices. In *Proceedings of the 29th International Conference on Fundamental Approaches to Software Engineering (FASE '26)*, pages 202–222, 2026. https://doi.org/10.1007/978-3-032-22774-4_11.

[47] Richard Hutcheson, Austin Blanchard, Noah Lambaria, Jack Hale, David Kozak, Amr S. Abdelfattah, and Tomas Cerny. Software architecture reconstruction for microservice systems using static analysis via GraalVM Native Image. In *Proceedings of the 2024 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER '24)*, pages 12–22, 2024. https://doi.org/10.1109/SANER60148.2024.00008.

[48] Peter Bailis, Ali Ghodsi, Joseph M. Hellerstein, and Ion Stoica. Bolt-on causal consistency. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data (SIGMOD '13)*, pages 761–772, 2013. https://doi.org/10.1145/2463676.2465279.

[49] Sanjay Ghemawat, Robert Grandl, Srdjan Petrovic, Michael Whittaker, Parveen Patel, Ivan Posva, and Amin Vahdat. Towards modern development of cloud applications. In *Proceedings of the 19th Workshop on Hot Topics in Operating Systems (HOTOS '23)*, pages 110–117, 2023. https://doi.org/10.1145/3593856.3595909.

[50] Lexiang Huang, Matthew Magnusson, Abishek Bangalore Muralikrishna, Salman Estyak, Rebecca Isaacs, Abutalib Aghayev, Timothy Zhu, and Aleksey Charapko. Metastable failures in the wild. In *Proceedings of the 16th USENIX Symposium on Operating Systems Design and Implementation (OSDI '22)*, pages 73–90, 2022. https://www.usenix.org/conference/osdi22/presentation/huang-lexiang.

[51] Jeffrey Dean and Luiz André Barroso. The tail at scale. *Communications of the ACM*, 56:74–80, February 2013. https://dl.acm.org/doi/10.1145/2408776.2408794.

ARTIFACT EVALUATED
USENIX
AVAILABLE

ARCTIC: a practical lock-free adaptive radix tree

Newton Ni, Nicolas Garza, Jenny Stinehour, Michael Goppert, Michal Friedman*, Emmett Witchel
The University of Texas at Austin *ETH Zürich*

Abstract

Indexing data structures are vital to the modern systems ecosystem, but there are no indexes that offer high performance, lock freedom, *and* range scans. ARCTIC is a lock-free adaptive radix tree that achieves all three: ARCTIC outcompetes lock-based indexes, including a concurrent hash map, on many YCSB configurations, guarantees non-blocking operation through careful metadata layout and an (eponymous) freezing-based coordination protocol, and offers non-linearizable range and prefix scans. ARCTIC also contributes a novel safe memory reclamation scheme that uses operation keys to approximate reachable pointers. We integrate ARCTIC into RocksDB and Turso, improving throughput up to 40% and 12% on their write-heavy benchmarks relative to their default skiplist indexes.

1 Introduction

Indexing structures are a fundamental component of modern systems infrastructure: they underpin persistent key-value stores [10,26], distributed caches [9], and databases [18,41,59]. In these settings, index operations are frequently on the critical path of request processing and must scale across many cores and sockets. We focus on three properties that such indexes should provide. First, **performance**: indexes must offer high throughput and low tail latency on real hardware. Second, **lock-freedom**: indexes should provide strong progress guarantees and scale under high contention without relying on coarse-grained locks. Lock-freedom is important when systems have more threads than physical cores: a thread suspended inside a lock-based index's critical section stalls other threads waiting on that lock, while a lock-free index makes progress regardless of scheduling (§ 4.1). Third, **range scans**: many workloads require efficient range and prefix scans. For example, LSM trees [46] in key-value stores rely on range scans for fast compaction, and databases use range scans to accelerate range queries and index-only plans. We claim that no existing index satisfies all three of these properties, a conclusion supported by previous studies [27,42].

We group index structures into a few families: hash maps, skiplists, B+-trees, prefix trees, and hybrids. Hash maps achieve excellent performance and admit highly concurrent and even lock-free implementations [39], but they do not support efficient range scans. Skiplists [58] are ordered, but suffer from poor cache locality and high contention at higher levels of the list. B+-trees are the default choice for most databases, but lock-based B+-trees using optimistic lock coupling [36] perform far better than the state-of-the-art lock-free Bw-tree [54]. In the prefix-tree family, the lock-based adaptive radix tree (ART) [35,37] is a widely used ordered index, while the lock-free SMART [38] struggles with memory usage and reclamation (§5). We are not aware of any lock-free hybrid or trie-based data structure that consistently outperforms these lock-based alternatives while also supporting efficient range scans.

ARCTIC is an adaptive radix tree based on ART that provides all three properties. (1) ARCTIC out-performs ART on almost every YCSB [24] workload we evaluated (§4.1); at 80 threads, the throughput increase over ART (geomean across seven key distributions) ranges from 1.3× on YCSB-C to 7.7× on YCSB-A. (2) ARCTIC's new metadata layout and (eponymous) freezing-based [14, 15] coordination protocol enable it to achieve lock-freedom at minimal performance cost: ARCTIC introduces no additional pointer indirection relative to ART, and updates nodes in-place (as opposed to read-copy-update), reducing both cache and allocator pressure. (3) ARCTIC provides wait-free (but non-linearizable [31]) range and prefix scans, as well as wait-free reads.

ARCTIC also integrates a novel safe memory reclamation (SMR, §2.5) scheme, which we call hazard keys, that correlates logical operations with physical pointers. We start with hazard pointers (HP) [43], which guarantee bounded unreclaimed memory [48] by announcing the exact set of pointers an operation is accessing, but add overhead to every pointer dereference as a result. In §3.9, we show how we can instead approximate protected pointers "for free" by correlating an operation's key with the tree structure, enabling us to protect all pointers by announcing just the key, up front, once per operation. Our evaluation (§4.3) explores how hazard key reclamation efficiency is sensitive to request distribution, but less sensitive to thread stalls or oversubscription.

In return for performance, lock-freedom, and range scans, ARCTIC requires 128-bit atomic compare and swap (CAS) for correctness, and 128-bit atomic loads for reasonable performance. ARCTIC's memory overhead varies with the key distribution, but is generally higher relative to ART for integer keys (0.97×-1.5×), and lower for string keys (0.19×-0.61×).

In summary, this paper makes the following contributions.

- We present the design of ARCTIC, the first lock-free ordered index to surpass lock-based alternatives in performance, and sketch proofs of its correctness and lock-freedom.
- We contribute a novel SMR scheme—hazard keys—and evaluate its sensitivity to request distribution and thread count.
- We demonstrate the real-world potential of ARCTIC by replacing concurrent skiplists in RocksDB [16] and

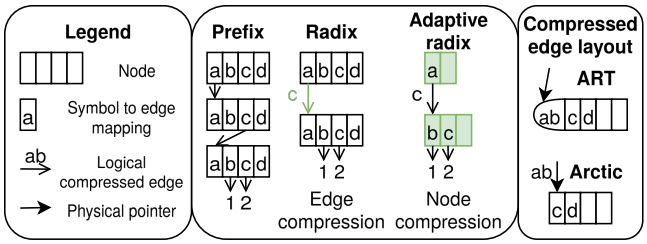


Figure 1: The midddle panel shows the evolution of prefix trees, with fixed nodes and height, to radix trees, which compress shared symbols into edges, to adaptive radix trees, which compress nodes into smaller partial mappings. All three trees represent the map $\{acb=1, acc=2\}$. The right panel shows two physical layouts of a logical compressed edge: ART stores the bytes in child node headers, while ARCTIC stores them alongside child pointers.

Turso [11], resulting in end-to-end speedups up to 40% and 12% on their write-heavy benchmarks.

- ARCTIC is open-source and available at https://github.com/nwtnni/arctic.

2 Background

This section provides technical background on lock-freedom, prefix trees, and safe memory reclamation.

2.1 Lock-freedom

An object is *lock-free* if, in every execution where operations are continually invoked, some operation always completes in a finite number of steps. A data structure is lock-free if, regardless of schedule, threads make progress in every execution. Even if some threads halt or crash, other threads will be able to continue executing their operations on the object, as no thread may block another. As a result, an operation on a lock-free object always keeps the memory in a consistent state.

Lock-free data structures are most commonly implemented using compare-and-swap (CAS) as a hardware synchronization primitive, which can atomically load, compare, and store up to 16B. The x86 has a 16B CAS instruction (`CMPXCHG16B`), as does ARM in the LSE extension (the `CASP` instruction).

2.2 Prefix and radix trees

A prefix tree is an associative data structure composed of nodes and edges that maps keys—sequences of symbols—to values. Each node is a total map from symbol to edge (typically a fixed size array). Each edge has one child, which can be a value or a node (which can be null). A simple example is shown in Figure 1. Prefix trees provide point operations in time complexity linear to the key size, and admit efficient range scans due to their lexicographically ordered structure. In practice, however, prefix trees suffer from (a) high time overhead when keys are long, due to following one pointer per symbol, and (b) high space overhead when keys are sparse, due to fixed size nodes.

Radix trees. Radix trees are a variant of prefix trees. Radix trees still use fixed size nodes, with the radix r determining the node size 2^r. The radix is commonly chosen as 8 so that symbols are bytes, which is efficient for hardware; we assume byte symbols in this paper. Radix trees reduce overhead through edge compression, i.e., labelling edges with shared bytes, which reduces the height of the tree and eliminates nodes with one child. Two physical representations of these edge bytes are shown in Figure 1: ART [35] stores them in child node headers, while ARCTIC stores them inline with child pointers.

Judy arrays [1] and the adaptive radix tree [35] (ART) additionally reduce space overhead through node compression, i.e., choosing different node representations depending on how many children the node has, which reduces the size of sparsely populated nodes. Both of these optimizations are displayed in Figure 1.

Important properties. We will briefly describe some properties that help us reason about correctness. First, every prefix tree element (node, edge, value) can be labeled inductively with a prefix:

Definition 1 (Prefix)**.** *The root edge has the empty prefix. If an edge has prefix p and bytes $[b_1, b_2, \ldots]$, then its child has prefix $p + [b_1, b_2, \ldots]$. If a node has prefix p and maps byte b to edge e, then edge e has prefix $p + [b]$.*

Prefix trees must satisfy two invariants: for a given prefix, there is at most one node in the tree, and for a given node in the tree, it has exactly one prefix. Second, we point out a major precondition of prefix trees:

Precondition 1. *No key is a prefix of any other key.*

This precondition avoids a vexing case where the same key maps to both a node and a value, and the restriction is easy to satisfy in practice. Fixed-size integers and C-style null-terminated strings both have this property; other dynamically-sized keys can append a sentinel byte. ARCTIC and all other prefix tree-based data structures we are aware of have this restriction.

2.3 Structural modification operations (SMOs)

SMOs are operations that modify the physical state of a data structure *without* modifying its logical state, and are typically necessary to maintain data structure invariants. For example, tree rotation maintains balance in a binary search tree, and resizing maintains load factor in a hash map. We will focus on SMOs because they are often the most challenging part of designing lock-free data structures, especially when they require writes to multiple memory locations. For example, a B+-tree node split requires writing both parent and child.

Naive prefix trees have very simple SMOs (bottom of Figure 2) that only write to the parent. Once we introduce edge compression and node compression, however, the number and complexity of SMOs also increase; they are necessary to manage edge bytes and node representations, respectively. We will reference these SMOs extensively, as our design is grounded in implementing SMOs atomically without extra indirection or locking (§3.3).

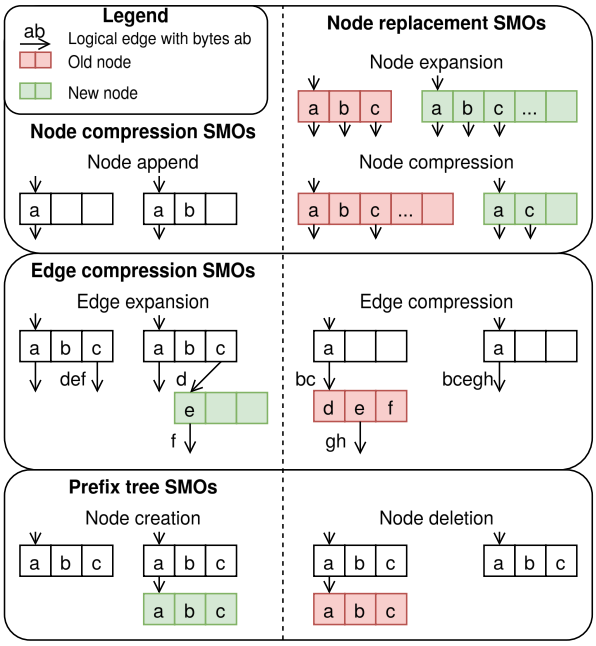


Figure 2: SMOs in prefix tree variants. Prefix trees (bottom) only have two SMOs: node creation and deletion. Edge compression (middle) requires two new SMOs to expand and compress edge bytes. Node compression (top) requires three new SMOs to append to node partial mappings and change the representation of a node. SMOs on the right side require unlinking (Figure 3); we refer to them collectively as node replacement SMOs.

2.4 Node life cycle

Figure 3 illustrates the life cycle of a ARCTIC node: the states a node can occupy and the transitions between them. Informally, we say a node is *reachable* if there exists some path to it from the root. Linking makes a node reachable, and unlinking makes it unreachable; in ARCTIC, these transitions are implemented atomically by CASing a pointer to the node. We will introduce *freeze* next, and *reclaim* in §2.5.

Freezing [14, 15] addresses a specific race between unlinking a node and concurrently accessing that same node. Depending on the data structure, it may still be correct to read an unreachable node, but it is generally incorrect to write to an unreachable node. For example, if thread A loads a pointer to node n, thread B unlinks n, and thread A writes to n, then thread A's write is not observable.

At a high level, freezing prevents unlink-write races by requiring unlinking threads to set a `frozen` bit on every CASable location in a node before unlinking; on the other side, a writing thread promises to CAS only when the frozen bit is clear. This provides a mechanism for writers to detect when the node they are writing *may* be unreachable—it could be frozen but still reachable. For reading threads, it is usually safe to restart when reading a location whose frozen bit is set, but this could be quite slow.

ARCTIC uses freezing as a primitive coordination mechanism, but there are many edge cases that must be specialized

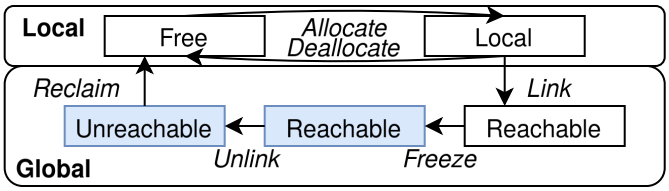


Figure 3: The life cycle of a node in ARCTIC. The top two states are local, meaning that there is at most one thread that can access the node. The bottom three states are global, meaning there may be any number of threads accessing the node. We use freezing to coordinate unlinking (§2.4); a frozen node may be reachable *or* unreachable. Our hazard key protocol (§3.9) determines when an unreachable node becomes safe to reclaim (§2.5).

for our data structure: for example, what does a writing thread do when it fails to CAS? When is it safe for a reader to read an unreachable node? Our design (§3) answers these questions and we sketch a correctness proof (§3.8).

2.5 Safe memory reclamation (SMR)

Any concurrent data structure that allows optimistic traversal (in other words, loading and dereferencing pointers without acquiring a lock) must be especially careful to avoid use-after-free errors. This is the safe memory reclamation (SMR) problem: determining when a pointer will never be dereferenced again, so it can be freed. In managed languages like Java or Go, SMR is handled by the language runtime's garbage collector. In languages with manual memory management like C, C++, or Rust, SMR falls to the programmer.

SMR algorithms make a fundamental trade-off between precision—how closely they can estimate the set of pointers that will no longer be dereferenced—and performance—how much runtime overhead they incur. Hazard pointers [43] keep track of the exact set of pointers being accessed, which bounds unreclaimed memory, but the cost is overhead for every pointer access. On the high performance end of the trade-off, epoch-based reclamation (EBR) [29] tracks quiescent periods between operations. EBR is popular for its consistently high performance, but even a single stalled thread can prevent all memory reclamation.

3 Design

To develop some high-level intuition for our design, and how it differs from ART, we walk through of a series of insertions (§3.1). Then we will go over:

- ARCTIC's metadata layout (§3.2)
- How to perform SMOs atomically (§3.3)
- How to traverse (§3.4) and insert (§3.5)
- How to reclaim memory (§3.9)
- Why ARCTIC is correct, linearizable, and lock-free (§3.8)
- How to optimize ARCTIC (§3.10)

3.1 Walkthrough

Figure 4 illustrates a series of insertion operations on ARCTIC and ART, starting from an empty tree.

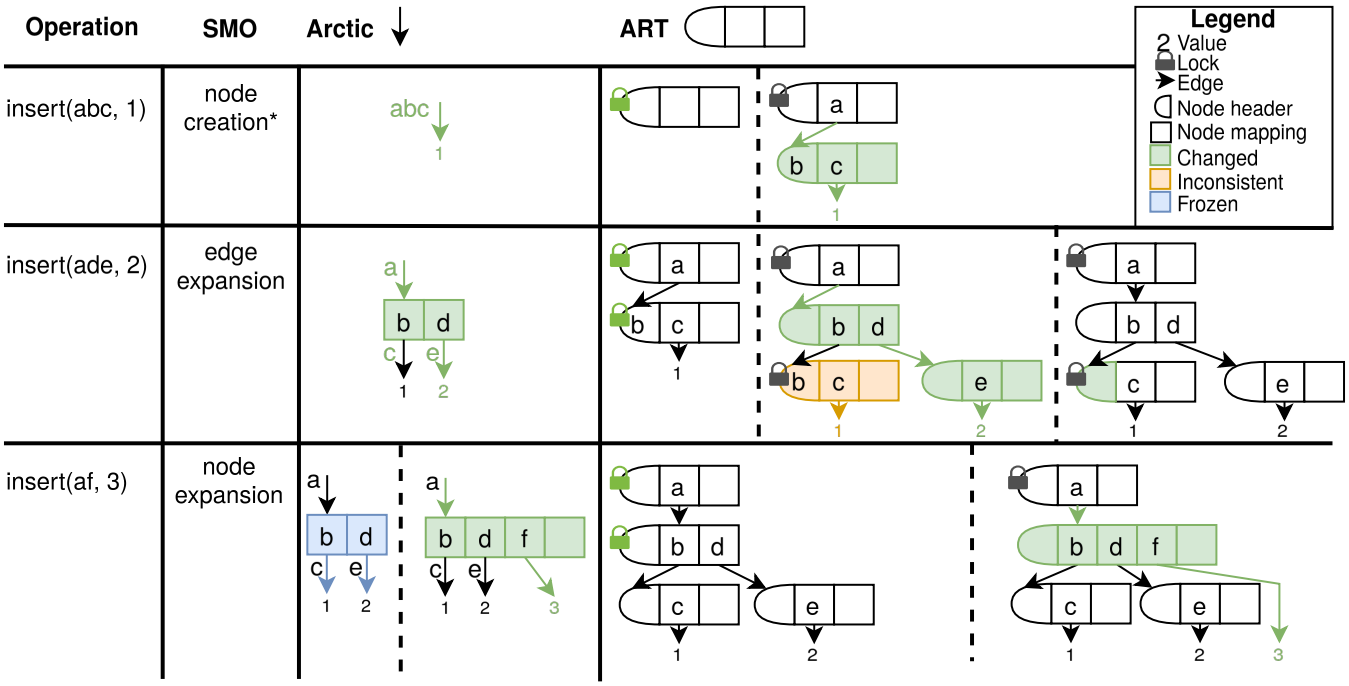


Figure 4: Walkthrough showing a series of insertions in ARCTIC and ART, starting from empty trees. Each insertion requires a different SMO (except the first insertion, which ARCTIC handles without an SMO). Each data structure undergoes atomic transitions between globally visible states, both between rows, and from left to right within a row (separated by dashed lines). To save space, we merge ART's locking and omit its unlocking.

Empty. ARCTIC is rooted at an edge, while ART is rooted at a node because: (1) ARCTIC stores edge bytes alongside edges, (2) ART stores edge bytes in child node headers, and (3) Precondition 1 precludes empty keys. Therefore, ART needs at least one node to represent a non-empty key, while ARCTIC does not.

Insert(abc, 1) (Row 1). ARCTIC can store a value with a non-empty key directly under the root edge, so no SMO is required. ART creates an intermediate node and locks the root node to install it.

Insert(ade, 2) (Row 2). This insertion requires edge expansion. ARCTIC requires only one CAS, but this SMO is painful for ART because ART stores edge bytes in child node headers. As a result, ART takes two locks to update the parent's edge and the child's node header, exposing an inconsistent state in between that readers must detect.

Insert(af, 3) (Row 3). We illustrate this insertion as if it requires a node expansion, though in practice, ARCTIC's smallest node can accommodate 3 entries, and ART can accommodate 4. Node expansion requires unlinking an old node. ARCTIC handles this case by using a freezing-based (§2.4) node replacement protocol (§3.3) that is expensive, but maintains consistency and supports in-place updates. ART takes two locks to prevent updates to the child while updating the parent's child pointer.

Summary. ARCTIC has clear performance benefits just from avoiding locking and unlocking, both in the common case of updates, where ART requires one lock, and in the rarer case of SMOs, where ART requires two locks.

More importantly, by removing ART's node-level locking, ARCTIC achieves much finer-grained concurrency, both vertically—between different levels of the tree—and horizontally—between different edges in a single node. As an example of vertical concurrency, consider two consecutive nodes, where both undergo node expansion. In ARCTIC, these two node expansion SMOs happen fully in parallel (§3.3), while in ART, they conflict on a lock on the second node. As an example of horizontal concurrency, consider updating two edges in a node. In ARCTIC, these two updates also happen in parallel, while in ART, they conflict on the node-level lock.

3.2 Metadata layout

Figure 5 shows the layout of ARCTIC's intermediate node types (Node3, Node15, Node47) which are based on ART's [35] (Node4, Node16, Node48), but ARCTIC (1) reorganizes edge and node metadata to enable atomic SMOs (Figure 2), and (2) ensures all node types have power-of-two sizes to make more efficient use of hardware. We will explain how this layout follows naturally from our constraints.

The biggest constraint on ARCTIC's layout is the edge expansion SMO. As shown in Figure 4, ART's placement of edge bytes in child node headers means edge expansion requires two writes, which is difficult to do atomically without locks. ARCTIC shifts these edge bytes upward into the parent edge. While our design still works with 8B edges and atomics, to fit more edge bytes, we make our edges 16B and require

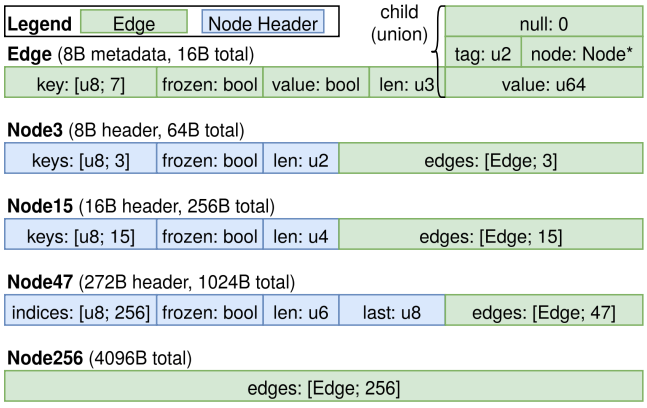


Figure 5: The memory layout of ARCTIC's edge and node types. The child of an edge is a union between a user value (if the `value` bit is set), null, or a node pointer with its type tag in the lower two bits, which are zero due to alignment. We ensure that all types have power-of-two sizes, which works well with hardware and memory allocators, and that our smallest node type (Node3) occupies exactly one cache line to avoid false sharing between different nodes.

the system to support 16B atomic operations.

With 16B edges in place, our next constraint is Node256, which must contain an array of 256 edges, exactly 4KiB. We cannot fit any metadata without causing Node256 to span two pages (bad for memory allocator, page table, TLB); luckily, the only metadata Node256 needs is a node type tag, so we shift node type tags into parent edges as well. (As a side benefit, this eliminates a cache miss: ART unconditionally loads child node headers to determine the type, but ARCTIC avoids this load when the child is a Node256).

Our final constraints come from node replacement and node append SMOs. Our node replacement protocol (§3.3) requires every edge and node header to have a frozen bit (§2.4). For node append, Node3 and Node15 need no additional changes, because their headers fit into 16B and can be CASed atomically, but Node47's header is too large. We add a field—(`last`, the last appended key byte)—and explain how we use it to achieve atomic node appends in §3.3.

3.3 SMOs

We now revisit our adaptive radix tree SMOs (Figure 2) and explain how ARCTIC's layout enables us to implement each SMO atomically. We focus on successful SMOs for now, as failed SMOs are handled at a higher abstraction level (e.g., insertion or deletion). Failed SMOs have no globally visible effects.

Edge expansion. ARCTIC places edge bytes in edges, which makes this SMO straightforward. Following the example in Figure 2, we expand an edge with bytes `def` to some child c. This can be done atomically as follows: (1) create a new Node3 n that maps byte `e` to edge with bytes `f` and child c, and (2) CAS the original edge to bytes `d` and child n.

Node replacement. Four SMOs—node expansion, node compression, edge compression, and node deletion, displayed on the right side of Figure 2—are similar in that they (1) scan an old reachable node to determine the contents of the new replacement node, (2) construct the new node in local memory, and (3) CAS to atomically unlink the old node and link the new node. However, this becomes complicated in the presence of in-place node updates: any number of concurrent updates to the old node may take place between steps (1) and (3), which may not be reflected in the new node when it is successfully linked in step (3).

We address this race condition in two ways. First, we prevent the old node from being updated between steps (1) and (3) through freezing (§2.4): we iterate over every CASable location in the node (header, edges) and CAS to set the frozen bit. The node is frozen once every frozen bit is set, and guaranteed to be immutable. Second, we avoid stale SMOs—for example, one thread starts freezing a full node in order to expand it, but every child is removed before freezing completes—by unifying all four SMOs into a single node replacement SMO. Threads may initiate a node replacement for any reason, but the new node (and SMO) is a deterministic function of the node's contents *after* freezing, instead of being decided up-front. Revisiting the previous example, the thread now deletes the node instead of expanding to a larger node type with zero children.

We wish to highlight some nice properties of freezing and node replacement. Freezing is infallible, as each CAS is retried while the frozen bit is not set (it could be set by another thread concurrently freezing). Freezing is idempotent, and every thread that freezes a node, potentially concurrently, will see the same node contents after freezing. Finally, node replacement is symmetric: every thread that concurrently replaces a node will construct a logically identical replacement node, because the replacement is determined solely by the immutable contents of the old node after freezing.

Node append. Node256 is a total map from byte to edge; it does not need a node append SMO to create more byte to edge mappings. But smaller node types (Node3, Node15, Node47) are partial maps, and do require an SMO to create new mappings.

We note that node append operations always append an edge with a null child, which does not change the tree's logical contents. Updating this newly appended edge is a separate operation. We also note that, like ART, our partial maps grow monotonically (which is why there is only a node append SMO). Unnecessary mappings are removed during node replacement.

Node3 and Node15 are similar: their headers comprise an array mapping from edge index to key byte, and the length, i.e., number of valid mappings. Because their headers are less than 16B, we can atomically CAS the header to update the mapping and the length.

Node47 is different; while its header also contains the length, it uses an array mapping from key byte to edge index (this inverse mapping avoids sorting during range operations). This is too large to be updated in one CAS, yet requires writing both the array and the length atomically to maintain consistency. Naively performing two CASes does not work: CASing the array first can result in two key bytes mapping to

the same edge index, while CASing the length first can result in the same key byte being assigned two edge indices.

We solve this by introducing a `last` field containing the last appended key byte, adjacent to `len`. Now, before inserting, a thread first helps [17] ensure consistency of the `len` and `last` fields. These fields are consistent if the array entry at index `last` is equal to `len` − 1; if they are not consistent, then the thread CASes the array itself. This helping is infallible: if the CAS fails, it is safe to ignore, as another thread must have helped. After helping, the thread appends its own mapping by CASing `len` and `last`, and finishes by helping itself.

This Node47 append algorithm requires some scrutiny: unlike the other SMOs we have discussed, it requires two CASes (one to update `len`/`last`, and one to update the array) and is not obviously atomic. Intuitively, this SMO completes after the second CAS, when the array is updated: readers do not check `len` and `last` at all, and writers ensure the array is always consistent, including before freezing. This means that the intermediate state between two CASes is not observable by any other operation or SMO, and our append is atomic.

3.4 Traversal

Algorithm 1 Traversal algorithm

```
1:  root: &Edge
2:  key: &[u8]
3:  i ← 0
4:  j ← i + LEN(root.bytes)
5:  while key[i..j] == root.bytes do
6:      match root.child? with
7:          Value(value) ⇒ return Some(value)
8:          Node(node) ⇒
9:              root ← node[key[j]]?
10:             i ← j
11:             j ← i + LEN(root.bytes)
    return None
```

Without loss of generality, we focus on traversal to a full key. (Traversal to a key prefix is analogous, except it may end at an intermediate edge or node.) We present pseudocode in Algorithm 1; the `?` operator is Rust syntax for returning early if `None`. Traversal is wait-free, never back-tracks, and does not interact with freezing. Traversal can terminate unsuccessfully in three different ways: (1) when *key*[*i*..*j*] does not match *root.bytes* in line 5, (2) when *root* has no child in line 6, and (3) when *node* has no mapping for byte *key*[*j*] in line 9 (note that *j* is a valid index into *key* here due to Precondition 1).

3.5 Insertion

The happiest path for insertion is when all intermediate edges, nodes, and node mappings for a key already exist—in other words, traversal proceeds until terminating at an edge with no child, which we will call the *final* edge. To insert, we atomically CAS the final edge from null to the new value.

Now, let us consider all the ways in which the happy path might become sad. We will distinguish between two levels of sadness: the first level is when an SMO is necessary to continue traversal, but we still assume all of our CASes succeed. The second level considers when any CAS may fail; thankfully, this level is recursive, so we will not need more.

A little sad. In §3.4, we list the three ways traversal can terminate unsuccessfully. In each case, we perform an SMO, and then continue traversal: (1) if the key bytes do not match the edge bytes, perform edge expansion; (2) if there is no child, and the remaining key bytes do not fit in one edge, perform node creation; (3) if there is no node mapping for the key byte, perform node append, or node replacement if the node is full.

Very sad. We now consider what happens when CASes fail, recalling that the only locations we CAS in ARCTIC are node headers (node append SMO) and edges (everything else). The first thing we do on CAS failure is to deallocate any new node that was created (SMOs with new nodes in Figure 2). This does not require SMR, as these nodes are still local (Figure 3). Then our response depends on whether or not the CAS failed due to freezing.

Case CAS-UF (CAS failure, unfrozen) Somewhat surprisingly, we can handle all of these failures the same way, regardless of what SMO or operation the CAS is a part of, by simply continuing traversal from the current edge. To understand why this works, we can look at some concrete examples. If the CAS failure was due to a concurrent (1) edge expansion, we will match the new edge bytes and child and traverse into the new intermediate node; (2) insert or delete, we will match the same edge bytes and child and retry our operation; (3) node append, we will match the same edge bytes and child and retry our node append.

Case CAS-F (CAS failure, frozen) We can infer in this case that the node that contains this edge or node header is undergoing node replacement, since that is the only time we freeze. We cannot just wait for the node replacement to finish, since that would violate lock-freedom (suppose the thread that started node replacement died), so we help by performing node replacement ourselves.

This case may require back-tracking, which we implement by keeping a stack of edges during traversal: if we failed to CAS an edge, we need to go up to the containing node, and then up to the node's parent edge; if we instead failed to CAS a node header, we do not need to back-track, as we are already at the node's parent edge. Then we perform node replacement, handling CAS failures recursively according to **CAS-UF** and **CAS-F**. Intuitively, this algorithm is correct because node replacement is always safe (§3.3), so even if the current edge has changed since we first traversed it (e.g., a concurrent edge expansion SMO created a new node in between), node replacement will at worst be a no-op. We can tell if the current edge is now unreachable because its frozen bit will be set, in which case we recurse, potentially all the way to the root—but no further,

because the root edge is never frozen; it is not inside a node.

In practice, we almost never hit case **CAS-F**, let alone recursively. Backtracking more than one level requires multiple node replacement SMOs to occur *simultaneously* at *consecutive* levels of the tree. For perspective, we present some numbers from our YCSB insert-only workload (§ 4.1) with 100M random 8B integer keys and 80 threads, which induces the highest number of SMOs and conflicts. Only 324 of these 100M inserts reach case **CAS-F**, and none of them backtrack more than one level. We have never observed more than two levels of backtracking.

3.6 Deletion

Deletion itself is straightforward due to our update-in-place design: traverse to the value and CAS the final edge to null. The difficulty comes from garbage collecting empty nodes. There is a spectrum of viable options, ranging from fast—assume deletion is rare and do no garbage collection—to slow—scan node after deletion and perform node replacement if it can be deleted or compressed, recursing upward if deleted. Garbage collection can require more memory as garbage builds, while node scanning insures that only in-use nodes occupy allocated memory.

A node scan is required because we have no counter of non-null children, because having one would create a single contention point for each node (node header mappings are orthogonal to whether the edges have non-null children). Scanning small nodes (Node3, Node15) is cheap, but it is expensive to scan large ones (Node47, Node256). Another option is to asynchronously scan the tree and perform node replacement, which can be done at any time by any thread.

3.7 Scan

Like most concurrent indexes [18, 37, 38, 57], ARCTIC's range (and prefix) scans are not linearizable [31]. This is acceptable in many use-cases; for example, LSM trees [46] in key-value stores [26] scan when there are no writers, and databases can use multi-version concurrency control [11, 25]. ARCTIC does guarantee some weaker properties: a scan observes keys at most once, in lexicographic order, and observes all keys (within range) that were inserted before the scan starts, and not removed before the scan ends.

3.8 Correctness

We first show that our SMOs preserve the prefix tree invariants (§2.2), namely that for a given prefix there is at most one node in the tree and that each node in the tree has exactly one prefix. We also show that we preserve the freezing invariants (§2.4), i.e., frozen CAS locations are immutable, and nodes are frozen before unlinking.

For prefix tree invariants, we first observe that the only way to violate the uniqueness of node prefixes is for a node to contain an inconsistent mapping, i.e., mapping the same byte to more than one edge. As explained in §3.3, Node3 and Node15 prevent this because their mappings fit within a single CASable header and are updated atomically. Node47 maintains consistency through its helping protocol and Node256's mapping is trivially consistent. A node can only have more than one prefix if it is reachable from multiple parents. This is impossible because ARCTIC SMOs either allocate new nodes or copy existing children while preserving their prefixes.

For freezing invariants, every CASable location contains a frozen bit in headers and edges (§3.2), and all update operations respect it. Furthermore, node replacement freezes all CASable locations before unlinking, ensuring that no node can become unreachable before it is fully frozen. Once a node is fully frozen, no further updates to its CASable locations can succeed, and therefore its contents become immutable. Consequently, invariants (§2.4) are satisfied.

Claim 1 (Locality)**.** *SMOs in* ARCTIC *preserve the reachability and prefixes of their parents and non-null children.*

Sketch proof. No SMO modifies memory above its containing node, so parents are unaffected. Node append and node creation only make new mappings visible. Edge expansion introduces a new intermediate node and edge while preserving the reachability and prefix of the existing child. Node replacement copies all non-null mappings from the frozen node into the replacement node before unlinking the original node. Since the node is fully frozen beforehand, its contents cannot change during replacement and no reachable mapping can be lost. Therefore every non-null child remains reachable with the same prefix. □

This property allows us to reason locally about operations and ignore interference from concurrent SMOs occurring elsewhere in the tree.

3.8.1 Linearizability

Claim 2. *A successful insert of key $k+[b]+k'$ linearizes at the successful CAS that changes the child of the reachable edge with prefix $k+[b]$ and key k' from null to the inserted value. If a value for the same key already exists, then the failed insert linearizes at the load that observes the existing non-null value.*

Sketch proof. After traversal and any required SMOs (§3.5), insertion reaches an unfrozen reachable edge whose prefix matches the key.

The successful CAS makes the value reachable from the root and therefore inserts the key into the logical map. By the prefix-tree invariants, e is unfrozen and therefore reachable, and no other reachable edge can correspond to the same key, so this CAS is the linearization point.

If the child of e is already a non-null value, insertion fails without modification. Therefore the failed insert linearizes at the load that reads the existing value. □

Claim 3. *A successful delete of key $k+[b]+k'$ linearizes at the successful CAS that changes the child of the reachable*

edge with prefix $k+[b]$ and key k' from a value to null. If no value or key exist, then the failed delete linearizes at the load that observes the longest matching prefix and determines that traversal cannot continue or the value does not exist.

Sketch proof. The argument is identical to insertion. The edge is reachable and uniquely identified by the key prefix. The successful CAS atomically removes the value and therefore serves as the linearization point.

If traversal instead observes that the key has no associated value, either because traversal terminates unsuccessfully or because the final edge contains a null child, then deletion fails without modification. The operation linearizes at the load that observes this absence. □

Claim 4. *A get of key $k+[b]+k'$ linearizes at the load of the matching edge when traversal reaches an unfrozen reachable edge with prefix $k+[b]$ and key k'. If no such edge exists, the linearization point is when the operation observes the longest matching prefix and determines that traversal cannot continue. Otherwise, the traversal reaches a frozen matching edge, and the linearization point is the latest point between when the edge was frozen and when the get operation starts.*

Sketch proof. If the edge is not frozen, the claim follows from the same logic as insert (without executing any SMO operations). In the second case, traversal terminates at the first point where the searched key diverges from the tree structure, either due to a byte mismatch, a missing child, or a missing node mapping. If the edge is frozen, the reasoning is more delicate. We know the containing node is reachable (because we reached it through traversal). The edge is immutable after being frozen (freezing invariants). This edge must be the *only* reachable edge with prefix $k+[b]$ until the containing node is replaced (prefix tree invariants). Then we have a window between when the edge is frozen and when the containing node is replaced where the value under this edge cannot change; we choose the earliest time in this window as our linearization point. □

3.8.2 Lock freedom

We prove by contradiction. Assume there is some execution for which no executing operation terminates after a certain point α. In this case, we assume that no operation is invoked after this point, and given a finite number of running processes, the set of running operations is also finite. Then:

Claim 5. *There is a finite number of freezings of reachable nodes after α.*

Proof. A traversal operation must terminate after finding the closest prefix of the searched key traversing from the root, an insert operation must terminate after inserting the relevant key, or updating the value of that key if it already exists, and a delete operation terminates once updating the value of the relevant key edge to null, if it exists. The freezing happens when a node needs to be expended or compressed during an insertion or deletion. Therefore, since we have a finite number of inserted and deleted nodes after α, and since freezing is sequentialized on a node level and irreversible, we can conclude that the number freezings of reachable nodes after α is finite as well. □

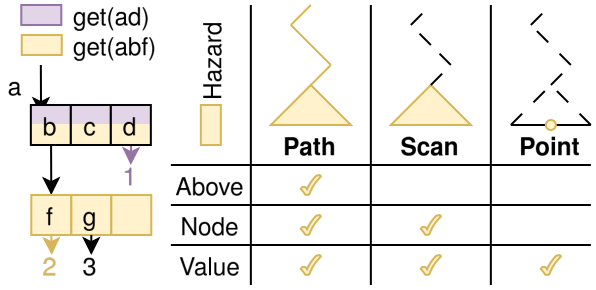


Figure 6: On the left, we show a concrete example of how an operation's key can be correlated with the structure of the tree; the two get operations in the top left will only access the nodes and values of their color. On the right, we show how a hazard key could be refined after traversal to a subtree for a scan operation, or a value for a point operation.

Claim 6. *There is a finite number of reachable node and edges changes after α.*

Proof. The nodes and edges only change along the insertion or deletion paths. They are caused due to SMOs or edge updates. We always keep traversing down the tree, even when CAS operations fail, unless we spot a freeze. Since freezing is finite (by Claim 5), we will backtrack up the tree a finite number of steps, and keep traversing down again. As the number of nodes and edges in the tree is finite, there is a finite number of reachable node and edge changes after α. □

Claim 7. ARCTIC *is lock-free.*

Proof. From Claims 5 and 6, after a certain point, there are no freezings or node and edge changes in the tree. Therefore, we consider the continuation of the execution that contains no state or pointer changes of reachable nodes and edges. From this point, ARCTIC does not change anymore. Since ARCTIC is finite, every traversal operation eventually ends. Furthermore, every insert and delete operation must be unsuccessful since no changes were applied, and so they also terminate. We get a contradiction and therefore, the implementation is lock-free. □

3.9 Hazard keys

We propose a new class of safe memory reclamation schemes by exploiting the correlation between prefix tree operations and structure. We will briefly introduce the key idea before discussing tradeoffs relative to existing schemes.

Recall that each node in ARCTIC is labelled with a prefix p (§2.2). A point operation on key k will only access nodes where p is a prefix of k; contrapositively, if a node's prefix p is *not* a prefix of k, then operation k will not access that node. The left side of Figure 6 shows a simple example: here, we can see that

no operation on key "ad" will access the bottom node (prefix "ab"), value 2 (prefix "abf"), or value 3 (prefix "abg"). Similarly, no operation on key "abf" will access value 1 (prefix "ad").

With this observation in mind, we propose the following SMR scheme: (1) include a prefix along with retired nodes or values, (2) publish a hazard key before each operation, and (3) reclaim allocations that are not protected by a hazard key, where an allocation with prefix p is protected by a hazard key k if p is a prefix of k. In other words, hazard keys use logical operations to cheaply approximate a set of physical hazard pointers [43].

The unique property of hazard keys is that their reclamation efficiency—how tightly an SMR scheme can bound the number of retired but unreclaimed allocations—is dependent on the request and key distribution. Hazard keys can tolerate thread stalls or oversubscription more gracefully than epoch-based schemes [29, 47], because a stalled thread prevents reclamation of a subset of allocations rather than all allocations. The same distribution-dependence is a liability under skew: a prefix is unlikely to be protected for long by uniformly random hazard keys, but could be protected indefinitely if the hazard keys are heavily skewed (so hazard keys, unlike their namesake hazard pointers, are **not robust**; they admit unbounded unreclaimed memory).

Hazard keys have lower overhead than hazard pointers, as a single hazard key can be published before an operation, whereas a hazard pointer must be published (and validated) before each pointer load. But hazard keys have higher overhead than epoch-based reclamation, as each retired allocation prefix must be matched against O(thread count) hazard keys, whereas an epoch comparison allows an entire batch to be freed.

We demonstrate these tradeoffs in our evaluation (§4.3), which suggests that hazard keys provide a reasonable balance of throughput and reclamation efficiency for low to moderately skewed workloads. There are many extensions outside the scope of this paper: for example, the right side of Figure 6 depicts how a hazard key could be refined at runtime to protect a subtree or single pointer, reducing the chance of false positives. Hazard keys could be hybridized with epochs to improve reclamation efficiency under high skew. And hazard keys could be adapted to comparison-based data structures like skiplists, with retired allocations associated with a key range, and protection determined by key comparison rather than prefix matching.

ABA problem. ARCTIC contains pointers to nodes and values. For nodes, we only CAS node pointers during node replacement (§3.3), after we have traversed into the node and while we hold a reference to the node, so any SMR scheme is sufficient to prevent the ABA problem. For values, the value could similarly be protected by SMR before CASing.

3.10 Optimization

Optimistic traversal. Back-tracking up the tree requires maintaining a stack of edge pointers during traversal. But a thread only needs to back-track in the rare case that it fails a CAS due to freezing. So we always try to perform operations first without keeping a stack; if we need to back-track, we restart the operation from the beginning with a stack.

Native integer support. Most radix tree implementations note that the memory representation of little-endian integers is not lexicographically ordered, and proceed to byte swap integers at their API boundary so they work well with internal byte arrays. Because integer keys are ubiquitous, we instead implement edge bytes as 8-byte integers, with endianness matching the key type. Integer keys store bytes from MSB to LSB independent of endianness, using a big-endian edge, while string keys use the system endianness. Integer edge bytes allow for branchless prefix matching, using XOR to diff the bits and counting leading or trailing zeros.

SIMD. We apply SIMD acceleration in two new places relative to ART: bitonic sorting [19] of Node15 headers, and iterating over Node47 headers. We also use SIMD within a register (SWAR) [34] for branchless lookup in Node3 headers.

System	Kind	Language	Lock-free
ARCTIC	Trie	Rust	Y
ART [35, 37]	Trie	C++	N
DashMap [4]	Hash map	Rust	N
FB$^+$-tree [18]	Hybrid	C++	N
Wormhole [57]	Hybrid	C	N

Table 1: Properties of the baseline systems in our evaluation.

4 Evaluation

Our evaluation seeks to answer the following questions:

- How does ARCTIC perform under different key distributions?
- How does ARCTIC affect the end-to-end performance of larger systems?
- When are hazard keys a viable option for safe memory reclamation?
- How effective are ARCTIC's optimizations?

We will present performance on YCSB [24] workloads under a variety of key distributions, performance on RocksDB [26] and Turso [11] benchmarks after integrating ARCTIC, memory reclamation efficiency on allocation-intensive workloads, and an ablation study on basic operations.

Experimental setup. We evaluate on a Chameleon [32] compute_icelake_r650 instance running Ubuntu 22.04.5 LTS and Linux kernel version 5.15. These instances have two Intel Xeon Platinum 8380 CPUs; each CPU runs at 2.30 GHz and has 40 cores, 120 MiB LLC, and 128 GiB of DDR4 3200 DRAM.

For reproducibility, we disable hyper-threading and turbo boost, set the CPU scaling governor to performance, and pin threads to cores.

Baselines. We compare against the latest ordered indexes we could find in the literature, along with the best-performing indexes in their evaluation, summarized in Table 1. ART [35] is the adaptive radix tree that ARCTIC is based on, using

Integer	Count	Description			
ipv4 [8]	139M	4 byte IPv4 addresses			
seq-u64	100M	Sequential 8 byte integers			
rand-u64	100M	Random 8 byte integers			
twitter [2]	100M	8 byte Twitter user IDs			
uuid-v4	100M	Random 16 byte UUIDv4s			
String		**P50**	**P90**	**P99**	**Max**
email [28]	114M	19	25	31	251
url [13]	38M	66	106	171	2045

Table 2: Key distributions, with length distributions for string keys.

ROWEX [37] for synchronization. DashMap [4] is a concurrent hash map written in Rust with consistently high throughput in open-source benchmarks [56]. FB$^+$-tree is a hybrid B+-tree and trie that supports wait-free search and lock-free update, but requires locks for SMOs. Wormhole is a hybrid B+-tree, trie, and hash map with O(log$L$) search complexity (where L is the key length). We exclude RDMA indexes ([38,51,52]) because their optimizations do not make sense for local shared memory.

4.1 YCSB

We run six YCSB [24] workloads across six key distributions (Table 2). Our YCSB workloads are standard: Load is 100% insert, YCSB-A is 50% read and 50% update, YCSB-B is 95% read and 5% update, YCSB-C is 100% read, YCSB-D is 95% read and 5% insert (skewed toward latest insertions), and YCSB-E is 95% scan and 5% insert. We use the default Zipfian request distribution with skew factor 0.99 and execute 100M operations. For values, we use 8-byte integers to reduce noise from memory allocation. In order to demonstrate ARCTIC's scalability, we evenly interleave memory and cores across NUMA nodes; in order to demonstrate scalable performance and lock-freedom, we evaluate at thread counts up to double the physical core count. We will briefly describe some of the key distributions (summarized in Table 2) before analyzing the YCSB results.

IPv4. We use 139M IPv4 addresses from the Stanford Internet Research Data Repository [50], extracted from the `saddr` field of `trial1_http_stanford64_zmap.json.lz4`.

Twitter. We use 100M Twitter user IDs from a public Kaggle dataset [3], extracted from the `ID` field of `twitter_users_20[0-9].idscreenname.csv`. Twitter user IDs (snowflake IDs [2]) are 63 bits, and can be generated without global coordination while providing a meaningful total order. An ID consists of—from most significant bit to least—a 41-bit timestamp, 10-bit machine ID, and a 12-bit machine-local sequence number.

Email. We use 100M emails from a public dataset [28], which we clean by matching against a regex [7], and preprocess by reversing the domain [38].

Url. We use 38M URLs from a public dataset [13], which we clean by removing URL query parameters.

For each combination of YCSB workload and key distribution, Figure 7 presents throughput vs. thread count (left columns) and peak memory usage (right-most column). ARCTIC excels at operations on integer keys, only losing a few times to DashMap. For string keys, ARCTIC provides competitive throughput on write-heavy workloads, but its read performance falls off as tree depth grows linearly with key length. Note that seq-u64 keys are optimal for ARCTIC structure (minimal depth, maximal Node256 count) and performance; we established with perf that YCSB-Load and YCSB-A with seq-u64 keys bottleneck on kernel-level page table contention (allocating new Node256 nodes) and hardware-level cache line contention (CASing final edges), respectively.

ARCTIC performs notably well on YCSB-A because updates modify minimal shared memory, only CASing the final edge to a value. We estimate cache line contention with hardware performance counters for hit modified events (mem_load_l3_hit_retired.xsnp_fwd + mem_load_l3_miss_retired.remote_hitm): ARCTIC averages 0.5 events per YCSB-A operation across all keys, which is consistent with a single CAS per update, and is a geomean reduction of 2.6× relative to DashMap, 3.0× relative to FB$^+$-tree, 3.8× relative to Wormhole, and 1.2× relative to ART.

ARCTIC achieves high throughput on YCSB-C in part through native integer key support, replacing `memcpy` and `memcmp` with efficient register moves and branchless prefix matching (§3.10), respectively. For example, for uuid-v4 keys, ARCTIC averages just 0.17 branch misses per operation, which is an increase of 1.14× relative to DashMap, but a decrease of 12× relative to FB$^+$-tree, 7.8× relative to Wormhole, and 2.5× relative to ART.

ARCTIC maintains throughput when threads are oversubscribed (past 80 threads), demonstrating its lock-free progress guarantees. Lock-based baselines exhibit some noticeable throughput losses, like ART in YCSB-B, and DashMap and Wormhole (which uses spinlocks) in YCSB-D. FB$^+$-tree locks for insertion and removal, but not reads or updates; its throughput is stable because its SMOs are infrequent and its spinlocks implement backoff.

ARCTIC's memory overhead is highly dependent on the key distribution, but generally comparable to ART and higher than other indexes. ARCTIC has low memory overhead for seq-u64 keys due to its dense Node256 nodes, and for string keys due to key bytes being shared along edges. ARCTIC has higher memory overhead for integer keys because its nodes and edges are large relative to the key size.

4.2 RocksDB and Turso

RocksDB [26] is a widely used persistent key-value store backed by an LSM tree [46]. LSM trees typically have an in-memory index component (memtable in RocksDB). RocksDB's default memtable is a lock-free concurrent skiplist, which we swap out for ARCTIC. We run RocksDB's bulk load benchmark, which inserts 100M randomly generated 20-byte keys and 400-byte values with write-ahead logging disabled.

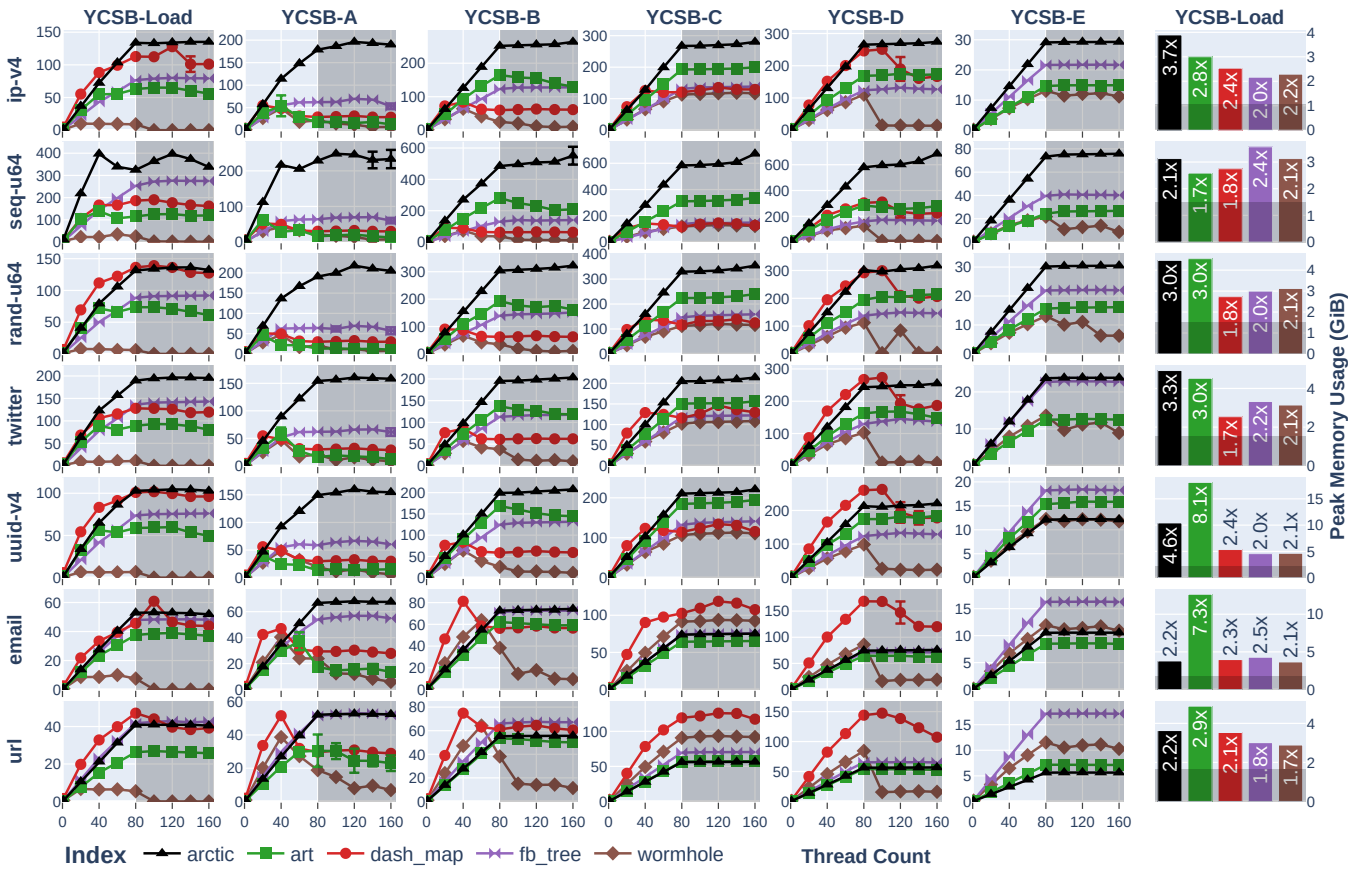


Figure 7: Throughput on YCSB workloads across different key distributions. Each row is a key distribution, and each column is a YCSB workload. The right-most column indicates the peak memory usage (measured by mimalloc) after loading with 80 threads. For the throughput graphs, the shaded area indicates thread oversubscription. For the memory graphs, the shaded area indicates the baseline size of inserted keys and values, and the text indicates memory usage relative to this baseline.

RocksDB Throughput (ops/sec)

Threads	1	2	4	8
skiplist	337K	413K	535K ± 10%	754K
arctic	1.36×	1.40×	1.13× ± 16%	1.05×

Turso Throughput (ops/sec)

Threads	1	2	4	8
skiplist	160K	240K	269K	254K
arctic	1.08×	1.10×	1.08×	1.12× ± 7%

Table 3: Average end-to-end throughput of RocksDB and Turso bulk load benchmarks when using original skiplist indexes vs. ARCTIC. Standard deviations under 5% are omitted.

Turso [11] is an open-source rewrite of SQLite in Rust. Turso supports concurrent writes via multi-version concurrency control (MVCC). The MVCC index uses a lock-free skiplist [6] to (1) map 16B row IDs to lock-synchronized row version data, and (2) map 8B transaction IDs to transaction metadata. We substitute these skiplists for ARCTIC and run Turso's multi-writer benchmark, which transactionally inserts 100K batches of 100 rows (one sequential integer and one short string per row).

Table 3 displays the end-to-end throughput of these two benchmarks using the original skiplists, and the relative throughput after switching to ARCTIC. On these workloads, ARCTIC improves RocksDB throughput by up to 1.40×, and Turso throughput by up to 1.12×. To better understand these results, we run RocksDB with internal performance counters enabled, and Turso with perf record. Figure 8 shows how ARCTIC consistently reduces time spent in index operations at all thread counts, but has less impact on end-to-end throughput at higher thread counts due to other system bottlenecks.

4.3 Hazard keys

To explore how hazard key reclamation efficiency varies with request distribution (§3.9), we run some SMR-intensive workloads—YCSB-A and YCSB-B with rand-u64 keys and 8-byte allocated values—and measure the peak number of retired but unreclaimed allocations. As baselines, we substitute hazard keys with crossbeam-epoch [5], a popular implementation of epoch-based reclamation [29] (EBR), and seize [12], a third-party implementation of Hyaline [45] (a hybrid of EBR and reference counting). For fairness, we configure all three with a retired batch size of 256 (crossbeam-epoch defaults to

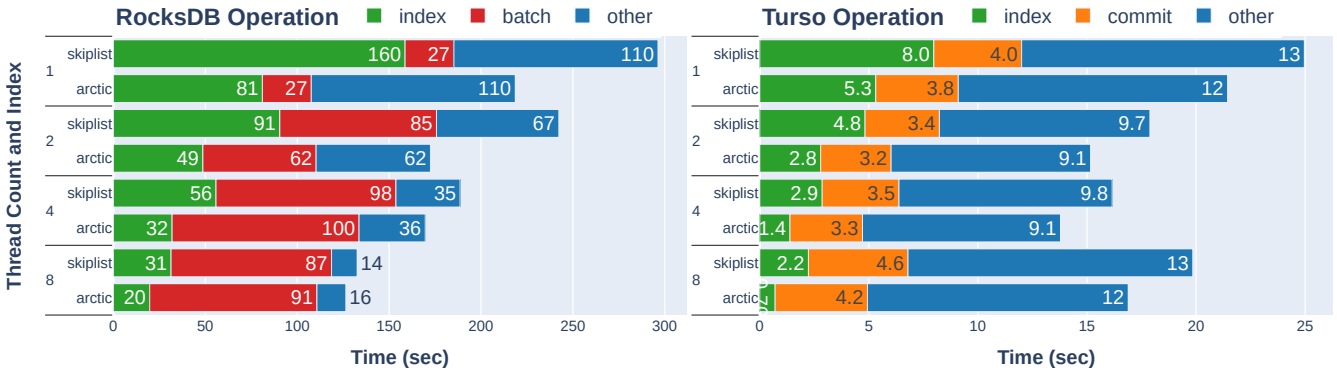


Figure 8: Performance breakdown of RocksDB and Turso bulk load benchmarks, showing how switching to ARCTIC reduces time spent in index-related operations. RocksDB measurements use internal timer metrics; Turso measurements use perf record.

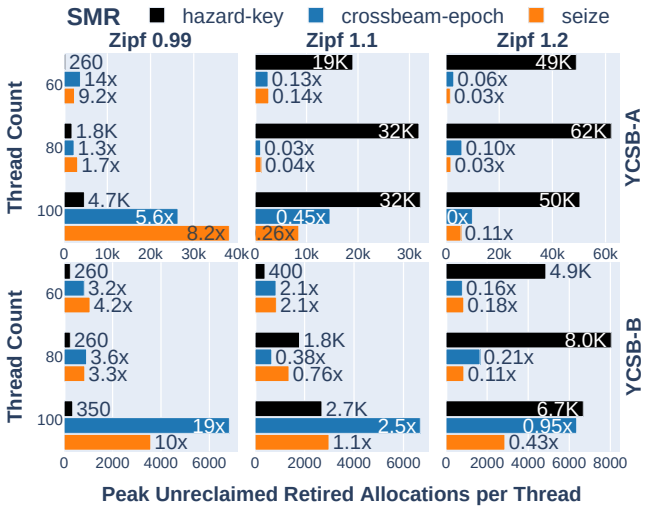


Figure 9: Reclamation efficiency of safe memory reclamation (SMR) schemes on SMR-intensive YCSB workloads with rand-u64 keys and allocated 8-byte values, varying skewness (Zipfian constant) and thread count. At Zipf 0.99, 17% of requests hit the top 10 (of 100M) keys, increasing to 33% and 48% at Zipf 1.1 and Zipf 1.2, respectively.

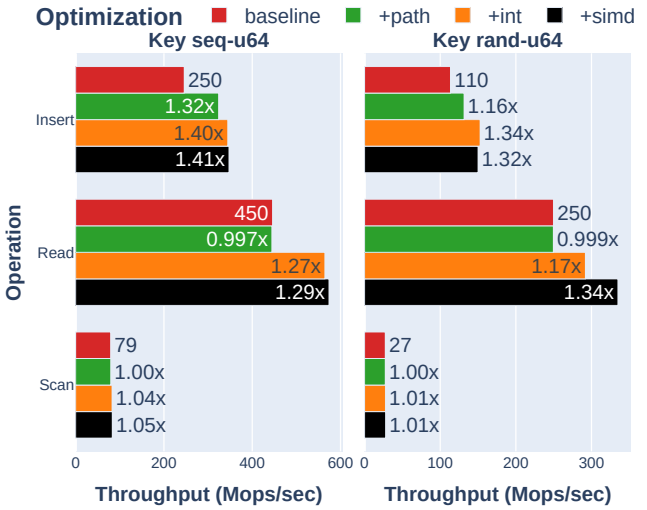


Figure 10: Effect of optimizations on ARCTIC operation throughput at 80 threads. Reads and scans use the default Zipf 0.99 skew from YCSB. *baseline* disables all optimizations; *+path* is optimistic traversal, which avoids maintaining a stack during insertion; *+int* is native integer support, as opposed to treating integer keys as byte arrays; and *+simd* uses SIMD for node header lookups and scans. The baseline is annotated with the absolute throughput; other bars are annotated with throughput relative to the baseline.

64, and seize to 32). (On YCSB-A at 80 threads, this batch size improves throughput by a geomean of 9.5% and 2.8× for crossbeam-epoch and seize, respectively, and affects peak garbage by -24% and +38%.)

Figure 9 shows that hazard keys can reclaim memory more efficiently than epoch-based alternatives when threads are oversubscribed, but only if the skew is low. At Zipf 0.99 and 100 threads, hazard keys reduce peak garbage by 5.6×-19× relative to baselines while retaining throughput close to crossbeam-epoch (+1.3% for YCSB-A, -12% for YCSB-B). Hazard keys can tolerate higher skew as the read proportion increases. The gap widens with thread count: EBR and Hyaline leave progressively more garbage as threads are added, and especially once they oversubscribe physical cores, because a single stalled thread blocks reclamation globally.

Under ideal conditions for all SMR schemes (Zipf 0.99, 80 threads), hazard keys incur throughput decreases of 7.2% and 6.3% relative to crossbeam-epoch for YCSB-A and YCSB-B, respectively, which is similar to seize (12% and 8.4%).

4.4 Ablation study

Figure 10 shows the effect of ARCTIC's optimizations (§3.10), applied cumulatively. Optimistic traversal (*+path*) helps only insertion (1.3× for seq-u64 keys, 1.2× random), as reads never back-track; the effect is larger for seq-u64 keys as threads insert disjoint key ranges, and almost never take the pessimistic path. Native integer keys (*+int*) reduce branches and instructions in the critical path of traversal, benefiting all operations and both keys. SIMD (*+simd*) has little effect on seq-u64 keys—these trees are primarily composed of dense Node256s that can be directly indexed by key bytes—but rand-u64 keys see more benefit due to SWAR acceleration of

Node3 header lookup. Scans, bounded by memory bandwidth, are insensitive to all three optimizations (under 1.05×).

5 Related Work

Lock-free adaptive radix trees. To the best of our knowledge, there are three other ART variants that claim lock-freedom: DART [61] (published concurrently with ARCTIC) and SMART [38] target remote direct memory access (RDMA), while HEART [44] targets persistent memory (PM). SMART side-steps read-unlink races at the cost of memory usage: it uses a single (maximally sized) layout for all nodes, and does not reclaim empty nodes. DART and HEART make a similar observation to ARCTIC that edge expansion SMOs can be made atomic by shifting edge bytes from child node headers to parent edges, but neither takes advantage of 128-bit atomics, and neither system is truly lock-free. HEART embeds versions in parent edges to coordinate unlinking of child nodes, but writers can block between version changes. DART similarly uses parent edges to coordinate, but embeds a frozen bit instead of a version. DART also introduces a grace period mechanism to allow progress in the common case, but writers can still block if node replacement stalls. The key distinction between ARCTIC and DART, which both nominally freeze nodes, is that ARCTIC freezes the *contents* of a node, while DART freezes the *parent edge* of a node. ARCTIC ensures every future update to a node will fail, guaranteeing immutability, whereas DART (and HEART) have a fundamental race condition between updating a node and loading its parent edge. ARCTIC also proposes a deterministic node replacement algorithm that any thread can use to help complete an in-progress node replacement without blocking.

Lock-based and lock-free designs. Concurrent ordered indexes have a long history of trading performance for progress guarantees. Classic high-performance trees such as Masstree [40] and other B+-tree variants rely on fine-grained locking or optimistic lock coupling: they scale well in practice on read-heavy workloads, but contended updates can still block, and careful engineering is required to avoid deadlock and priority inversion. Lock-free designs such as Bw-tree [55] reduce blocking by replacing in-place updates with delta records and indirection through mapping tables, but this complicates traversal and increases pointer chasing.

Other index families like skiplists and binary search trees are typically composed of smaller nodes relative to B+-trees, which makes lock-freedom easier to achieve, but ultimately limits their performance on modern hardware.

ARCTIC provides lock-freedom while preserving the cache efficiency of non-concurrent adaptive radix trees.

Safe memory reclamation (SMR). Safe memory reclamation (SMR) for lock-free data structures is vital for high performance. Hazard pointers [43] provide portable lock-free reclamation but incur per-access pointer publishing and global scans, imposing performance overheads on read-heavy workloads [23]. Epoch-based reclamation (EBR) amortizes costs by tracking quiescent states, but can leak memory in the presence of stalled threads [21, 30]. New SMR schemes like Optimistic Access [23] and AOA [22] automate or streamline reclamation for normalized lock-free algorithms. Hyaline attains near-EBR throughput with hazard-pointer-like robustness and bounded memory usage [45].

SMR and persistent memory. Several systems co-design their data structures with safe memory reclamation (SMR) and persistent memory (PM) constraints in mind. NBTree [60] introduces a lock-free B+-tree tailored to eADR-enabled PM. Elimination (a,b)-trees [49] (OCC-ABtree and Elim-ABtree) feature publishing elimination so concurrent inserts and deletes on the same key can cancel each other, dramatically cutting writes. These trees are durably linearizable and still nearly as fast as the volatile versions. Jiffy [33] is a lock-free skiplist–based ordered index that supports batch updates and linearizable snapshots. UPSkipList [20] provides a recoverable lock-free skiplists by adding epoch tags to nodes so that after a crash, threads can distinguish in-progress updates from stale, inconsistent state and repair accordingly, preserving correctness and scalability on NUMA PMEM machines. Easy Lock-Free Indexing in NVM [53] uses a persistent multi-word CAS primitive to build both a lock-free Bw-tree and a lock-free skiplist record store on PM.

6 Conclusion

ARCTIC is a lock-free adaptive radix tree that supports range and prefix scans while preserving a cache-friendly, low-indirection layout. By reorganizing its metadata to accommodate 128-bit atomic operations, ARCTIC maintains its structural invariants without extra pointer chasing or read-copy-update, making it practical to deploy in latency-sensitive, highly concurrent software.

7 Acknowledgements

We thank our shepherd, Tim Harris, and our anonymous reviewers for their thoughtful suggestions. Our work is supported in part by PRISM, one of the seven centers in JUMP 2.0, a Semiconductor Research Corporation (SRC) program sponsored by DARPA. We used the Chameleon testbed [32] supported by the National Science Foundation for development and artifact evaluation.

References

[1] Judy arrays web page. https://judy.sourceforge.net/, 2004.

[2] twitter-archive/snowflake. https://github.com/twitter-archive/snowflake/tree/b3f6a3c6ca8e1b6847baa6ff42bf72201e2c2231, 2012.

[3] Twitter: 2.8 billion id screenname cross reference. https://www.kaggle.com/datasets/samanthuel/twitter3, 2022.

[4] Dashmap. https://github.com/xacrimon/dashmap/tree/v6.1.0, 2024.

[5] Crossbeam epoch. https://github.com/crossbeam-rs/crossbeam/tree/master/crossbeam-epoch, 2025.

[6] crossbeam-skiplist. https://github.com/crossbeam-rs/crossbeam/tree/983d56b6007ca4c22b56a665a7785f40f55c2a53/crossbeam-skiplist, 2025.

[7] Html living standard. https://html.spec.whatwg.org/multipage/input.html#valid-e-mail-address, 2025.

[8] Macroscopic internet topology data kit (itdk). https://www.caida.org/catalog/datasets/internet-topology-data-kit/#itdk-datasets, 2025.

[9] Redis. https://redis.io/, 2025.

[10] Tikv. https://tikv.org/, 2025.

[11] Turso. https://turso.tech/, 2025.

[12] seize. https://github.com/migopp/seize/tree/863ff99645223c778ae20ec8fee5e0f20f324391, 2026.

[13] Paolo Boldi, Bruno Codenotti, Massimo Santini, and Sebastiano Vigna. Ubicrawler: A scalable fully distributed web crawler. *Software: Practice & Experience*, 34(8):711–726, 2004.

[14] Anastasia Braginsky and Erez Petrank. Locality-conscious lock-free linked lists. In *Proceedings of the 12th international conference on Distributed computing and networking*, ICDCN'11, page 107–118, Berlin, Heidelberg, January 2011. Springer-Verlag.

[15] Anastasia Braginsky and Erez Petrank. A lock-free b+tree. In *Proceedings of the twenty-fourth annual ACM symposium on Parallelism in algorithms and architectures*, SPAA '12, page 58–67, New York, NY, USA, June 2012. Association for Computing Machinery.

[16] Zhichao Cao, Siying Dong, Sagar Vemuri, and David H.C. Du. Characterizing, modeling, and benchmarking RocksDB Key-Value workloads at facebook. In *18th USENIX Conference on File and Storage Technologies (FAST 20)*, pages 209–223, Santa Clara, CA, February 2020. USENIX Association.

[17] Keren Censor-Hillel, Erez Petrank, and Shahar Timnat. Help! PODC '15, page 241–250, New York, NY, USA, 2015. Association for Computing Machinery.

[18] Yuan Chen, Ao Li, Wenhai Li, and Lingfeng Deng. Fb+ -tree: A memory-optimized b+ -tree with latch-free update. *Proceedings of the VLDB Endowment*, 18(6):1579–1592, February 2025.

[19] Jatin Chhugani, Anthony D. Nguyen, Victor W. Lee, William Macy, Mostafa Hagog, Yen-Kuang Chen, Akram Baransi, Sanjeev Kumar, and Pradeep Dubey. Efficient implementation of sorting on multi-core simd cpu architecture. *Proc. VLDB Endow.*, 1(2):1313–1324, August 2008.

[20] Sakib Chowdhury and Wojciech Golab. A scalable recoverable skip list for persistent memory. In *Proceedings of the 33rd ACM Symposium on Parallelism in Algorithms and Architectures (SPAA '21)*, pages 426–428, 2021.

[21] Nachshon Cohen. Every data structure deserves lock-free memory reclamation. *Proceedings of the ACM on Programming Languages*, 2(OOPSLA):143:1–143:24, 2018.

[22] Nachshon Cohen and Erez Petrank. Automatic memory reclamation for lock-free data structures. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA '15, pages 260–279. ACM, 2015.

[23] Nachshon Cohen and Erez Petrank. Efficient memory management for lock-free data structures with optimistic access. In *Proceedings of the 27th ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '15, pages 254–263. ACM, 2015.

[24] Brian F Cooper, Adam Silberstein, Erwin Tam, Robert Eagle, Anmol Thakar, Omar Battikhi, Makoto Tatebe, Russell Smith, Julian Pignataro, Robert James, et al. Benchmarking cloud serving systems with YCSB. In *Proceedings of the 1st ACM symposium on Cloud Computing*. ACM, 2010.

[25] Cristian Diaconu, Craig Freedman, Erik Ismert, Per-Ake Larson, Pravin Mittal, Ryan Stonecipher, Nitin Verma, and Mike Zwilling. Hekaton: Sql server's memory-optimized oltp engine. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, page 1243–1254, New York New York USA, June 2013. ACM.

[26] Siying Dong, Andrew Kryczka, Yanqin Jin, and Michael Stumm. Rocksdb: Evolution of development priorities in a key-value store serving large-scale applications. *ACM Transactions on Storage*, 17(4):1–32, November 2021.

[27] Jose M Faleiro and Daniel J Abadi. Latch-free synchronization in database systems: Silver bullet or fool's gold? *CIDR*, 2017.

[28] fonxat. 300 million email database, 01 2018.

[29] Keir Fraser. *Practical lock-freedom*. Number 579. 2004.

[30] Keir Fraser. *Practical Lock-Freedom*. PhD thesis, University of Cambridge, Cambridge, UK, 2004. Technical Report UCAM-CL-TR-579.

[31] Maurice P. Herlihy and Jeannette M. Wing. Linearizability: a correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems*, 12(3):463–492, 1990.

[32] Kate Keahey, Jason Anderson, Zhuo Zhen, Pierre Riteau, Paul Ruth, Dan Stanzione, Mert Cevik, Jacob Colleran, Haryadi S. Gunawi, Cody Hammock, Joe Mambretti, Alexander Barnes, François Halbach, Alex Rocha, and Joe Stubbs. Lessons learned from the chameleon testbed. In *Proceedings of the 2020 USENIX Annual Technical Conference (USENIX ATC '20)*. USENIX Association, July 2020.

[33] Tadeusz Kobus, Maciej Kokociński, and Paweł T. Wojciechowski. Jiffy: A lock-free skip list with batch updates and snapshots. In *Proceedings of the 27th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '22)*, pages 401–414, 2022.

[34] Leslie Lamport. Multiple byte processing with full-word instructions. *Communications of the ACM*, 18(8):471–475, August 1975.

[35] V. Leis, Alfons Kemper, and T. Neumann. The adaptive radix tree: Artful indexing for main-memory databases. In *2013 IEEE 29th International Conference on Data Engineering (ICDE)*, page 38–49, Brisbane, QLD, April 2013. IEEE.

[36] Viktor Leis, Michael Haubenschild, and Thomas Neumann. Optimistic lock coupling: A scalable and efficient general-purpose synchronization method. *IEEE Data Eng. Bull.*, 2019.

[37] Viktor Leis, Florian Scheibner, Alfons Kemper, and Thomas Neumann. The art of practical synchronization. In *Proceedings of the 12th International Workshop on Data Management on New Hardware*, page 1–8, San Francisco California, June 2016. ACM.

[38] Xuchuan Luo, Pengfei Zuo, Jiacheng Shen, Jiazhen Gu, Xin Wang, Michael R Lyu, and Yangfan Zhou. Smart: A high-performance adaptive radix tree for disaggregated memory. *OSDI*, 2023.

[39] Tobias Maier, Peter Sanders, and Roman Dementiev. Concurrent hash tables: Fast and general(?)! *ACM Transactions on Parallel Computing*, 5(4):1–32, December 2018.

[40] Yandong Mao, Eddie Kohler, and Robert Tappan Morris. Cache craftiness for fast multicore key-value storage. In *Proceedings of the 7th ACM european conference on Computer Systems*, page 183–196, Bern Switzerland, April 2012. ACM.

[41] Yoshinori Matsunobu, Siying Dong, and Herman Lee. Myrocks: LSM-tree database storage engine serving facebook's social graph. *Proceedings of the VLDB Endowment*, 13(12):3217–3230, 2020.

[42] Syed Akbar Mehdi, Deukyeon Hwang, Simon Peter, and Lorenzo Alvisi. Scaledb: A scalable, asynchronous in-memory database. *OSDI*, 2023.

[43] Maged M. Michael. Hazard pointers: Safe memory reclamation for lock-free objects. *IEEE Transactions on Parallel and Distributed Systems*, 15(6):491–504, 2004.

[44] Liangxu Nie, Shengan Zheng, Bowen Zhang, Jinyan Xu, and Linpeng Huang. Heart: a scalable, high-performance art for persistent memory. In *2023 IEEE 41st International Conference on Computer Design (ICCD)*, page 487–490, Washington, DC, USA, November 2023. IEEE.

[45] Ruslan Nikolaev and Binoy Ravindran. Snapshot-free, transparent, and robust memory reclamation for lock-free data structures. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI)*, 2021.

[46] Patrick O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth O'Neil. The log-structured merge-tree (lsm-tree). *Acta Informatica*, 33(4):351–385, June 1996.

[47] Shangyu Qian, Tian Zhou, Xuanhe Liu, Yue Wu, et al. Massively parallel multi-versioned transaction processing. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI '24)*. USENIX Association, 2024.

[48] Ajay Singh, Trevor Alexander Brown, and Ali José Mashtizadeh. Simple, fast and widely applicable concurrent memory reclamation via neutralization. *IEEE Transactions on Parallel and Distributed Systems*, 35(2):203–220, February 2024.

[49] Anubhav Srivastava and Trevor Brown. Elimination (a,b)-trees with fast, durable updates. In *Proceedings of the 27th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '22)*, pages 1–15, 2022.

[50] Gerry Wan, Liz Izhikevich, David Adrian, Katsunari Yoshioka, Ralph Holz, Christian Rossow, and Zakir Durumeric. On the origin of scanning: The impact of location on internet-wide scans. In *Proceedings of the ACM Internet Measurement Conference*, IMC '20,

page 662–679, New York, NY, USA, October 2020. Association for Computing Machinery.

[51] Jing Wang, Qing Wang, Yuhao Zhang, and Jiwu Shu. Deft: A scalable tree index for disaggregated memory. In *Proceedings of the Twentieth European Conference on Computer Systems*, page 886–901, Rotterdam Netherlands, March 2025. ACM.

[52] Qing Wang, Youyou Lu, and Jiwu Shu. Sherman: A write-optimized distributed b+tree index on disaggregated memory. (arXiv:2112.07320), December 2021. arXiv:2112.07320 [cs].

[53] Tianzheng Wang, Justin Levandoski, and Per-Åke Larson. Easy lock-free indexing in non-volatile memory. In *2018 IEEE 34th International Conference on Data Engineering (ICDE)*, pages 461–472, 2018.

[54] Ziqi Wang, Andrew Pavlo, Hyeontaek Lim, Viktor Leis, Huanchen Zhang, Michael Kaminsky, and David G. Andersen. Building a bw-tree takes more than just buzz words. In *Proceedings of the 2018 International Conference on Management of Data*, page 473–488, Houston TX USA, May 2018. ACM.

[55] Ziqi Wang, Andrew Pavlo, Hyeontaek Lim, Viktor Leis, Huanchen Zhang, Michael Kaminsky, and David G. Andersen. Building a bw-tree takes more than just buzz words. In *Proceedings of the 2018 International Conference on Management of Data*, page 473–488, Houston TX USA, May 2018. ACM.

[56] Joel Wejdenstål. conc-map-bench, 06 2024.

[57] Xingbo Wu, Fan Ni, and Song Jiang. Wormhole: A fast ordered index for in-memory data management. In *Proceedings of the Fourteenth EuroSys Conference 2019*, page 1–16, Dresden Germany, March 2019. ACM.

[58] Lu Xing, Venkata Sai Pavan Kumar Vadrevu, and Walid G. Aref. The ubiquitous skiplist: A survey of what cannot be skipped about the skiplist and its applications in data systems. *ACM Comput. Surv.*, 57(11):297:1–297:37, June 2025.

[59] Wen Yang, Tao Li, Gai Fang, and Hong Wei. PASE: Postgresql ultra-high-dimensional approximate nearest neighbor search extension. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, pages 2241–2253. ACM, 2020.

[60] Bowen Zhang, Shengan Zheng, Zhenlin Qi, and Linpeng Huang. NBTree: a lock-free pm-friendly persistent B+-tree for eadr-enabled PM systems. *Proceedings of the VLDB Endowment*, 15(6):1187–1200, 2022.

[61] Bowen Zhang, Shengan Zheng, Shi Shu, Jingxiang Li, Zhenlin Qi, Weiquan Huang, Jianguo Wang, Linpeng Huang, and Hong Mei. Dart: A lock-free two-layer hashed art index for disaggregated memory. *Proceedings of the ACM on Management of Data*, 4(1):1–25, April 2026.

ARTIFACT EVALUATED
USENIX
AVAILABLE

Efficient and Scalable Synchronization via Generalized Cache Coherence

Yanpeng Yu *Yale University*
Seung-seob Lee *Yale University*
Lin Zhong *Yale University*
Anurag Khandelwal *Yale University*

Abstract

We explore the design of efficient and scalable synchronization for disaggregated shared memory. Porting existing synchronization primitives to such architectures results in poor performance scaling due to redundant inter-cache communications, exacerbated by high cache-coherence latency in disaggregated shared memory.

Driven by our insight that synchronization is a generalization of cache coherence in time and space, we argue for minimally extending existing cache coherence protocols to support synchronization primitives, thereby eliminating the redundant inter-cache communication inherent in layered synchronization. We propose a novel *G*eneralized cache-*C*oherence *P*rotocol (GCP) that realizes this insight by leveraging wait queues and variable-size cache lines directly at the cache-coherence layer for temporal and spatial generalization, respectively. We have verified GCP's correctness using model checking. We present Soul, an end-to-end system implementation of GCP atop a disaggregated shared-memory platform. Soul supports popular lock APIs through a user-space library that offers improved performance without requiring any changes to application code. Our evaluation of Soul against state-of-the-art locks shows that it improves the performance of unmodified real-world applications at scale by 1–2 orders of magnitude while incurring $< 8\%$ storage overhead.

1 Introduction

With the end of Moore's Law and challenges in scaling DRAM technologies [1], recent years have seen a push towards rack-scale compute-memory disaggregation [2–16], where server resources are physically decoupled into compute and memory blades connected via a high-speed network fabric. Several recent efforts have also focused on enabling cache-coherent shared-memory abstractions over the same high-speed networks for application transparency via a unified, consistent memory view [16–21]. We focus on page-based cache-coherent shared memory [16–18,20] in this work, where local DRAM on compute blades serves as cache for disaggregated memory, and these caches are kept coherent over high-speed Ethernet (§2.1).

In such systems, synchronization primitives such as locks [22–35] are crucial for the performance scaling of multi-threaded applications such as key-value stores [36–38] and databases [39–42]. Since such locks are already implemented atop cache-coherent substrates in multi-core CPU architectures (Fig. 1 (left)), it follows that these lock algorithms could also be ported to cache-coherent disaggregated memory.

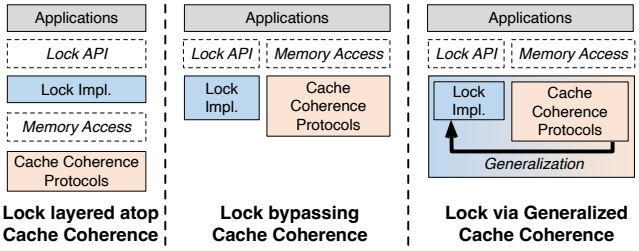


Fig. 1: **Different approaches for realizing synchronization.** Layered approaches (left) incur redundant coherence transactions while standalone approaches (middle) preclude key optimizations, both significantly degrading application performance over disaggregated memory's slower inter-compute and compute-memory interconnects. Our approach (right) *generalizes* cache coherence for efficient locks.

Unfortunately, despite improvements in network speeds, the higher latency of inter-compute and compute-memory links in rack-scale disaggregation renders cache-coherent substrates inadequate for the efficient implementation of lock algorithms, thereby limiting application performance at scale. While inter-core communication in multi-core and NUMA architectures exhibits latencies of 20–100 ns, page-based disaggregated memory exhibits latencies of 5–10 μs [16–18]. We find that even the most performant lock algorithms layered over state-of-the-art disaggregated memory coherence substrates [16,43] lead to poor application performance scaling, degrading performance for real-world synchronization-heavy workloads by as much as 1000×, due to redundant inter-cache coherence messages over higher-latency Ethernet links (§2.2).

To avoid such coherence overheads, prior distributed shared memory (DSM) [17,20,44–47] developed lock services that bypass the cache coherence substrates altogether (Fig. 1 (middle)). Unfortunately, porting such lock services to state-of-the-art disaggregated shared memory systems [16–18,20,21] still incurs intractable performance overheads. On the one hand, software-based lock services remain orders of magnitude slower than shared-memory accesses accelerated via programmable networks [16,18]. On the other hand, implementations of such lock services that leverage programmable networks suffer from stringent processing and storage constraints [48,49], precluding the adoption of many optimizations employed by prior DSM lock services (§3.3).

Toward scalable and performant locks for disaggregated shared memory, we pursue a third approach as shown in Fig. 1 (right), based on the insight that *lock-based synchronization is essentially a generalization of cache coherence in time and space*. Specifically:

Temporal generalization. Cache-coherence protocols guarantee the single-writer-multi-reader (SWMR) invariant [50], i.e., either a single exclusive writer or multiple concurrent readers can access a cache line (a 4KB page in page-based disaggregated memory) for a single instruction. Lock-based synchronization extends this property for an arbitrary number of instructions: the critical section.

Spatial generalization. Cache-coherence protocols ensure the SWMR invariant at a cache line granularity (i.e., a 4KB page), while reader-writer locks require this property for shared states of arbitrary sizes.

Therefore, extending existing coherence protocols to support these generalizations would provide the necessary semantics to enable lock primitives directly, eliminating the redundant communication seen in a layered design. Moreover, generalizing cache coherence protocols not only allows the lock realization to inherit decades of optimization from today's cache coherence protocols but also creates new opportunities to further improve lock scalability. These include directly moving the data associated with a lock during its acquisition and exploiting locality by caching both the lock and the associated data until they are explicitly invalidated.

We incorporate the above insights into GCP (§3), a novel class of *G*eneralized cache-*C*oherence *P*rotocols for lock-based synchronization. GCP minimally extends directory-based cache-coherence protocols using two key approaches:

Wait queues for temporal generalization (§3.1). Cache coherence guarantees that a cache line requested by a node will be held in the requestor's cache for a single instruction — subsequent requests from other nodes may immediately invalidate or downgrade the cache line permission on the original requestor. To allow a requestor to hold the cache line for more than one instruction, GCP prevents other requestors from invalidating or downgrading the cache line until the original requestor explicitly releases it, by suspending their cache coherence transactions in a wait queue.

Variable-size cache lines for spatial generalization (§3.2). While coherence protocols typically track fixed-sized cache lines, our spatial generalization lets each cache line track a list of arbitrarily sized memory regions (dubbed 'the shared memory list'). All shared regions tracked by a single cache line are atomically moved or invalidated from the target cache. This allows requestors to achieve SWMR semantics for arbitrary-sized shared regions during their critical sections.

These generalizations allow transforming some cache lines in the cache-coherence protocol into locks for synchronizing shared state in GCP; the remaining 'regular' cache lines still operate at a fixed size and single-instruction granularity. Moreover, since all our modifications are limited to the cache coherence protocol, GCP imposes no restrictions on the memory consistency model. We have verified GCP's correctness for a broad range of directory-based cache coherence protocols via model checking (§3.4).

We present Soul as a high-performance, end-to-end system implementation of GCP atop a state-of-the-art, Ethernet-based, disaggregated shared-memory platform [16], with minimal metadata overhead. Soul supports standard lock APIs via a user-space library, providing complete application transparency without requiring programmers to interact with low-level GCP interfaces. Soul also provides a lock manager to track the allocation of GCP cache resources and to handle application and system failures. Our evaluation (§5) shows that Soul enables scalable, high-performance synchronization across disaggregated memory. Compared to state-of-the-art lock algorithms, Soul improves real-world application performance at scale by 1–2 orders of magnitude without application modifications.

We also conduct a case study to validate GCP's generalizability to emerging CXL 3.0-based disaggregated shared memory systems [21] via a gem5 [51] simulation of Soul (dubbed SoulCXL; §6). While we demonstrate that the benefits of GCP also extend to hardware-based cache-coherence protocols, we leave a full adaptation to future work.

2 Background & Motivation

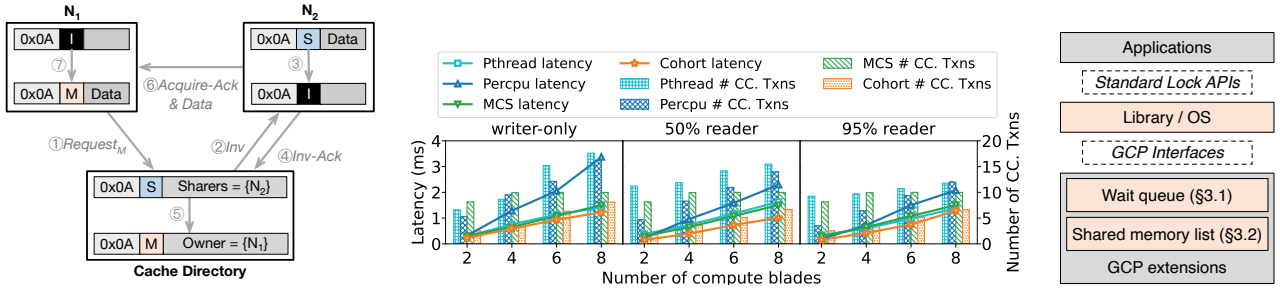
2.1 Page-based Disaggregated Shared Memory

As compute and memory resource scaling within a single server has started to stagnate, several recent efforts have argued for rack-scale memory disaggregation [13–18, 20, 52–56], where physical servers are disaggregated into compute and memory blades connected via a single network switch. To ensure application performance, these approaches adopt a partial disaggregation model, in which most of the memory is disaggregated, while compute blades retain a small amount of local DRAM as cache. In particular, these approaches leverage the OS's swap subsystem on the compute blade to cache memory from Ethernet-attached memory blades (*e.g.*, using RDMA) at page granularity in its local cache.

Increasingly, a large subset of these approaches [16–21, 57] have argued for a cache-coherent shared-memory abstraction over disaggregated memory to enable applications to transparently scale across compute blades. This effectively permits application threads to be placed across arbitrary compute blades and still access disaggregated memory via a unified, consistent memory view — akin to transparent Distributed Shared Memory (DSM) approaches [44–46].

While different systems vary slightly in how they implement disaggregated shared memory, they share a common core set of design elements. We illustrate these elements using MIND [16], an open-source, state-of-the-art disaggregated shared-memory system, on which our approach is implemented.

MIND comprises compute and memory blades connected via



(a) Directory-based MSI Protocol (b) Overheads of layering design (c) Overview of our design

Fig. 2: **Background & motivation (§2)** (a) Coherence example with a single cache line at address `0x0A`, a cache directory, and two compute blades (§2.2). (b) Lock acquisition latency and # of coherence transactions for locks layered atop coherence on disaggregated memory for different reader-to-writer ratios (§2.2). (c) GCP *generalizes* existing coherence protocols to realize locks (§2.2).

a programmable network switch. It keeps DRAM caches on compute blades coherent for transparent memory sharing.

Cache coherence. Similar to traditional DSM [44–47] and other disaggregated shared memory systems [17, 18, 20, 21], MIND uses *directory-based* cache-coherence due to its scalability. These protocols employ a cache directory to track the state of fixed-size memory units — typically referred to as *cache lines* and realized as 4KB *pages* in MIND — across the distributed caches. MIND places the directory on the programmable network switch because of its central location. The per-page state includes the list of compute blades currently holding the page (*sharer list*) and the permissions under which they hold it. When a compute blade attempts to access a page not present in its cache, the page fault handler first contacts the directory to check its current state, then notifies other compute blades on the sharer list of its intent to access the page, and ultimately coordinates subsequent data movement.

Specifically, MIND employs the directory-based MSI protocol, where each page can be in one of the three permissions: **M**odified (**M**), indicating a single compute blade has *exclusive* read and write permissions for the page, **S**hared (**S**), where multiple compute blades have *shared* read permission to the page, and, **I**nvalid (**I**), i.e., the page is not present in any compute blade cache. Fig. 2(a) shows an example with a single page at address `0x0A`, a cache directory, and two compute blades N_1 and N_2. The page is initially cached at N_2 with **S** permission. N_1 then requests it with **M** permission from the directory (①). The directory looks up the sharer list for the page and contacts N_2, its current holder, to *invalidate* it since N_1 needs exclusive access to it (②). Once invalidated (③), N_2 informs the directory (④), which then updates the page's permissions to **M**, records N_1 as the owner (⑤), and acknowledges N_1 with the page data (⑥). N_1 then accesses the page (⑦). While this example demonstrates the **S**→**M** transition in cache permission, other transitions are similar. We refer to these transitions as cache-coherence transactions. Minimizing the number of coherence transactions in synchronization primitives is essential for scaling the performance

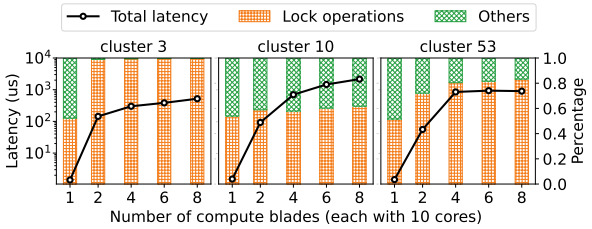


Fig. 3: **End-to-end request latency (left y-axis, log-scale) and %-age of time spent in synchronization (right y-axis)** for the state-of-the-art `Cohort` lock [27] on Twitter KV store workloads [58] atop MIND [16], a disaggregated shared memory system.

of parallel applications on disaggregated memory, since each transaction incurs multiple Ethernet round trips (§2.2).

2.2 Enabling Scalable Synchronization in Disaggregated Shared Memory Systems

Despite improvements in scalable cache coherence for disaggregated shared memory, lock-based synchronization remains a critical performance bottleneck in such systems.

Lock algorithms atop cache coherence. Scalable lock algorithms are typically built atop hardware coherence substrates in multi-core and NUMA architectures to permit either a single exclusive writer or multiple concurrent readers to access a critical section. Most scalable lock designs minimize the number of coherence transactions required for lock acquisition. For instance, queue-based lock algorithms [25, 26, 59] let each requestor thread spin on a core-private cache line, reader-writer locks partition reader-indicators [32] to avoid coherence transactions for concurrent readers, while memory-hierarchy-aware locks [27–31] exploit memory locality to avoid coherence transactions across NUMA nodes.

Unfortunately, even with optimized designs, lock algorithms layered atop cache coherence tend to severely limit application performance at scale in disaggregated shared memory systems. Fig. 3 shows the total request handling latency

and the percentage spent in lock operations for three Twitter key-value (KV) store workloads [58] using `cohort` locks [27], a state-of-the-art high-performance lock, atop MIND [16] (setup in §5). While request handling consumes under 10μs on average on a single compute blade, it can increase to over 1ms at 8 blades, with most of the time spent in lock operations! This is because each lock operation must trigger multiple expensive cache coherence transactions over Ethernet.

These coherence overheads are not limited to `Cohort` locks; they apply to all state-of-the-art locks built atop cache-coherence substrates. Fig. 2(b) shows that representative locking algorithms trigger many coherence transactions with multiple blades for workloads with varying degrees of read-write contentions, translating into milliseconds of lock acquisition and data access latency. Similar overheads have been demonstrated in prior studies [60–63] for multi-core architectures, even with much shorter inter-cache latency.

Locks bypassing cache coherence. To circumvent the inefficiencies of layering locks atop cache-coherence substrates, DSM systems [17, 20, 45–47] employ standalone software-based lock services that *bypass* the cache-coherence layer altogether, *e.g.*, via lock managers that serve lock requests using a client/server model [45], or distributed lock protocols that bypass shared memory [46, 47]. However, with recent disaggregated shared-memory systems [16, 18] that use programmable network hardware to accelerate cache coherence, software-based lock services incur latency that is orders of magnitude higher than cache-coherent memory accesses, making them an inadequate choice for synchronization.

Although programmable network-accelerated lock services [48, 49] can potentially deliver latency comparable to network-accelerated cache-coherent memory accesses, they require additional resources in an already resource-constrained programmable network switch. In fact, even if all of the programmable switch resources were allocated solely to the lock service, it would still fail to support critical performance optimizations that require complex coherence logic to be realized within the lock service [48, 49, 64] (§3.3).

Our approach. Fundamentally, cache coherence and lock-based synchronization strive for the same goal: single-writer, multi-reader (SWMR) invariant [50] over some shared state, i.e., at any point, either a single entity that intends to modify the shared state has *exclusive* access, or multiple entities that intend to read the state have *shared* access. However, while cache-coherence protocols enforce it at a single-instruction granularity (in time) and a fixed cache-line granularity (in space), lock algorithms strive for an SWMR invariant at arbitrary instruction counts (in a critical section) and data-size granularities. In other words, lock-based synchronization *is a generalization* of cache coherence in time and space.

Our approach builds on this observation and argues for *minimal* extensions to cache-coherent substrates to realize generalized coherence protocols (GCP) that can natively support lock primitives (Fig. 2(c)). Our extensions — a wait queue for temporal generalization (§3.1) and a shared memory list for spatial generalization (§3.2) — permit all lock operations to be realized using a single coherence transaction, avoiding overheads of layered designs. Moreover, unlike approaches that bypass coherence, GCP enables new optimizations that leverage the interplay between synchronization and coherence, while still allowing synchronization to benefit transparently from coherence protocol optimizations (§3.3). Finally, minimal extensions to coherence protocols permit easier verification (§3.4) and adoption in emerging coherent interconnects [43] (§6).

3 GCP Design

We provide an architecture-agnostic description of GCP design (§3.1 and §3.2). We then present GCP optimizations enabled by generalizing cache coherence protocols (§3.3) and conclude with a verification of GCP's correctness (§3.4).

3.1 Wait Queue for Temporal Generalization

As we saw in §2.2, a directory-based cache-coherence protocol ensures the SWMR invariant at a single instruction granularity by tracking the permission of the requested cache line (*e.g.*, **M**, **S**, or **I** in the MSI coherence protocol[1]). If an instruction's execution on one blade requests a cache line with a specific permission, the protocol triggers a transaction that immediately makes the cache line available to that instruction with that permission. To enable the temporal generalization of the protocol, where a blade can hold the cache line with its requested permission for an arbitrary number of instructions, GCP defers cache requests with conflicting permissions by enqueuing them in a *wait queue* and dequeuing a request only when the previous blades release the cache line.

Added cache interfaces. We add three new cache request types to interact with the wait queue (Table 1): `GCP_acquire` to request the cache line with the provided permission and mark the beginning of the critical section, `GCP_is_acquired` to check whether the line has been acquired with the provided permission, and `GCP_release` to release the cache line and mark the end of the critical section. To avoid blocking while a blade waits for a lock, the `GCP_acquire` request is *asynchronous*, i.e., the initiating blade does not block while it waits for completion. Once the corresponding transaction completes, the cache line's permission is updated to indicate that it has been acquired, and the blade can check this status via `GCP_is_acquired`. As we show in §4.3, this allows context switching to another thread while waiting for a GCP lock.

[1] While we use the MSI coherence protocol to illustrate our extensions, they apply to any directory-based cache-coherence substrate.

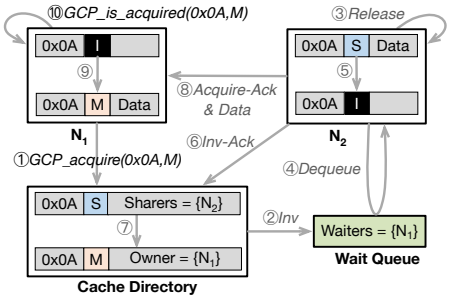


Fig. 4: **Temporal generalization with wait queues (§3.1).**

Shared memory list interfaces
`int GCP_create(size_t len, addr_t *addrs, size_t *sizes);`
`int GCP_destroy(size_t len, addr_t *addrs, size_t *sizes);`
Wait queue interfaces
`int GCP_acquire(addr_t addr, int perm);`
`int GCP_is_acquired(addr_t addr, int perm);`
`int GCP_release(addr_t addr);`

Table 1: **GCP interfaces (§3.1, §3.2).**

Handling evictions. Evictions of cache lines to memory used as locks require careful consideration since they can inadvertently terminate the current lock ownership. A naïve solution could pin such cache lines to prevent them from being evicted, but this can result in starvation or deadlocks. For instance, if a thread holding a lock is switched out by the scheduler, its cache lines used as locks would block space in the cache, starving the currently running thread of cache resources. Moreover, if a blade's entire cache is used as locked cache lines, no more regular or lock-based cache lines can be brought into cache, resulting in a deadlock!

To this end, GCP enables eviction of lock-based cache lines by adding a set of new transient states to the directory entry that can mark a cache line as both evicted and locked after it is cleared from local cache. This allows a thread to retain its lock ownership even when it is switched out, or a cache line is evicted. Any new request for such a cache line from another thread is granted if the requested and currently held permissions for the cache line are compatible (*e.g.*, both are **S** in an MSI protocol); otherwise, it is directed to the wait queue.

Example. Fig. 4 demonstrates how a wait queue can enable temporal generalization for the same example in Fig. 2(a). The target cache line is initially cached at N_2 with **S** permission. N_1 then issues an `GCP_acquire` request for the same cache line with **M** permission to the cache directory (①). The directory looks up the current cache line permission (**S**) and sharer list ($\{N_1\}$), realizing that the request requires N_2 to relinquish the cache line via invalidation. In contrast to standard MSI protocol execution, the directory defers the invalidation and instead enqueues N_1's request in a wait queue associated with the cache line (②). Only when N_2 finishes its critical section and voluntarily releases the cache line via `GCP_release` (③) is the request dequeued (④) and the invalidation performed at N_2 (⑤). The remainder of the cache-coherence transaction proceeds as per the standard MSI protocol — N_2 informs the directory (⑥), which updates the cache line's permission to **M** and marks N_1 as the only sharer (⑦), and sends N_1 an acknowledgment along with the cache line data (⑧). N_1 then transits to **M**, which is detected via `GCP_is_acquired` (⑩).

As with cache coherence, other permission transitions are similar or simpler. **M**→**M** and **M**→**S** transfers require similar deferred invalidations, enqueuing the transfer requests in the wait queue until the blade holding the cache line with **M** permission explicitly releases the cache line. **S**→**S** transfers do not require enqueuing the request since multiple readers can hold the cache line simultaneously under the SWMR invariant. **I**→**S** and **I**→**M** transfers also do not require enqueueing requests since no blade has the cache line to begin with.

The wait queue's location does not affect correctness but does affect performance; we discuss how our implementation navigates performance tradeoffs for queue placement in §4.1.

3.2 Shared Memory List for Spatial Generalization

Unlike coherence protocols that track permissions for a fixed-size cache line, locks must preserve the SWMR invariant across arbitrary amounts of shared state. The shared state may be fragmented and even empty. This requires extending cache-coherence protocols to track multiple shared memory locations of arbitrary size. Specifically, instead of a single address with a fixed size, each address tag in GCP is logically a list of (m_i, s_i) pairs, where m_i and s_i are the base address and size in bytes of a shared memory region, respectively. Note that the *shared memory list* is a cache-movement optimization rather than a lock enforcement mechanism: the wait queue (§3.1) alone enforces the lock semantics, and the shared memory list merely bundles the shared state into a single coherence domain — without it, the lock remains correct, but accesses to shared state incur additional coherence transactions.

Added cache interfaces. We add two new cache interfaces (Table 1) for creating and destroying shared memory lists. `GCP_create` associates a list of address ranges with a shared memory list (to be tracked via GCP), while `GCP_destroy` destroys the shared memory list, marking the associated address ranges as regular cache lines (tracked via regular coherence transactions). To ensure consistency in the shared memory list's creation and destruction, both interfaces are broadcast to all blades and the cache directory, regardless of which blade triggers them, and they force a cache flush for all involved lines at all blades. In addition, GCP requires that all shared memory lists contain non-overlapping memory regions; i.e., `GCP_create` returns an error if any of the involved regions are already associated with another shared memory list.

Example. Fig. 5 shows our spatial generalization for the

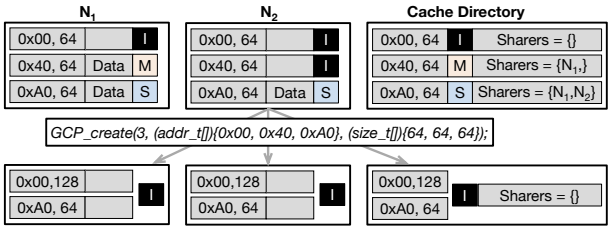


Fig. 5: **Spatial generalization with shared memory lists (§3.2).**

same example as Fig. 4. N_2 initializes the shared memory list by issuing `GCP_create` over three 64*B* cache lines at `0x00`, `0x40` and `0xA0`, which are merged into a list of two shared memory regions (`0x00`, 128) and (`0xA0`, 64) with their cache permissions reset to **I**. Coherence transactions over the shared memory list are identical to Fig. 4, except the invalidation step ⑤ now requires N_2 to remove two memory regions of different sizes from its cache, and in step ⑧, the directory sends N_1 the data for both regions with the acknowledgment. We discuss its implementation in §4.2.

3.3 Cache-coherence Optimizations

Since GCP realizes synchronization by generalizing cache coherence, it automatically inherits optimizations from traditional cache-coherence protocols, which are intractable with stand-alone lock services due to resource constraints (§2.2). We show in §5.3 that each optimization can improve lock acquisition performance by over an order of magnitude.

Acquiring data along with lock. GCP can acquire the lock and the associated data in a single cache coherence transaction, reducing delay and coherence traffic when placing the corresponding data in the requestor's cache. In contrast, even with prefetching, such coupling is impossible with traditional lock algorithms, since they can fetch the shared state only after the lock is acquired, delaying the requestor by at least one coherence transaction.

Exploiting temporal locality for locks. In traditional coherence protocols, once a cache line is placed in a requestor's cache, it remains there until it is invalidated to exploit the temporal locality of data accesses. In extending such protocols, GCP inherits the same optimization: both the lock and the associated data remain in the requestor's cache until another request invalidates them.

3.4 Correctness

Theorem 1. *GCP locks are correct, i.e., GCP satisfies mutual exclusion and deadlock-freedom.*

Intuitively, the mutual exclusion property for reader-writer locks implies that no two writers, or a reader and a writer, can enter the critical section concurrently — which is equivalent to the temporally generalized SWMR invariant of cache coherence. Due to the complexity of the underlying cache coherence protocols, we validate GCP's correctness (mutual exclusion and deadlock-freedom) via model checking. We used ProtoGen [65], a state-of-the-art cache coherence protocol synthesis tool, to generate specifications for four representative directory-based protocols: MSI [66], MESI [67], MOSI [68], and MOESI [69], and to integrate GCP into these specifications. We then used the Murphi [70] model checker to verify deadlock freedom and mutual exclusion in all reachable states. Like many prior works on cache coherence validation [19, 65, 71–73], our model checker is limited to a single address, two data values, three blades with private caches, and a directory to validate protocol correctness, thereby avoiding a state-space explosion. Our microbenchmarks (§5) on up to 8 blades with 80 cores over many addresses also serve as correctness "stress" tests for GCP implementations. While we guarantee correctness at the protocol level, it is still the programmer's responsibility to avoid application-level deadlocks, as we detailed in §4.3.

Finally, we note that GCP locks are also starvation-free (fair) if the wait queue policy is starvation-free (*e.g.*, FIFO). Intuitively, starvation-freedom requires that any lock requests will eventually succeed. GCP can only block lock requests in the wait queue: the remainder of the cache coherence transaction execution is non-blocking by design. As such, if the wait queue is starvation-free, any queued lock requests will eventually be processed, i.e., GCP locks are starvation-free.

4 Soul: An Implementation of GCP

We present Soul, a high-performance end-to-end system realization of GCP atop MIND, a state-of-the-art page-based, disaggregated shared-memory platform (Fig. 6). Soul realizes GCP atop MIND's inter-blade DRAM cache coherence to facilitate inter-blade synchronization. Soul does not affect MIND's regular pages[2] (i.e., those not used as locks), and retains MIND's x86-TSO memory consistency model since all modifications are restricted to cache coherence. Soul makes minimal extensions to MIND's programmable cache coherence substrate to implement GCP's wait queue (§4.1) and shared memory lists (§4.2). To ensure application transparency, Soul provides a user-space lock library as a shim atop its GCP implementation, exposing standard lock APIs (§4.3). Finally, Soul's lock manager allocates GCP cache resources to ensure cache integrity and handle failures (§4.4).

4.1 Implementing the Wait Queue

A natural way to implement the wait queue in Soul is to reuse the coherence protocol's message buffers, blocking cache

[2]Throughout this section, we use *page* as the unit of inter-blade coherence to reflect MIND's 4KB cache granularity.

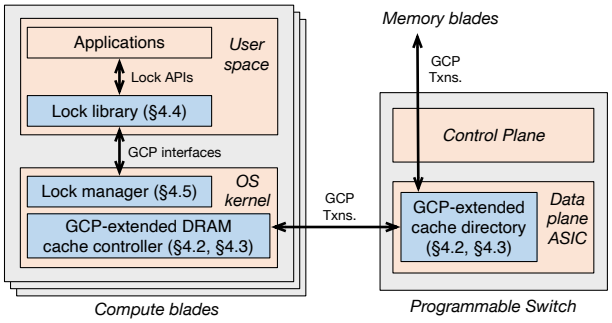


Fig. 6: **Soul is a full-system realization of GCP atop MIND (§4).**

requests at MIND's in-kernel cache controllers or in-network directory. However, this adds significant network delays to lock acquisition due to Ethernet's high latency. When the lock owner releases the lock, additional network trips are needed to dequeue the next request if it is stored in the directory. Instead, placing the queue with the page's current owner (compute blade) avoids these delays and bypasses the limited compute and storage resources of switches [16, 18, 48, 64, 74]. The directory only tracks which compute blade holds the queue (the queue holder) so it can forward access requests accordingly.

However, if multiple compute blades (readers) hold the page, the wait queue must be replicated across all readers so they can locally dequeue the next request, requiring a mechanism to ensure consistency across queue replicas. The problem is further exacerbated when the page ownership changes over time. We address this with **a novel queue transfer protocol** between compute blades, which guarantees (i) no network delay in processing the next request in the queue and (ii) consistent access to the queue itself.

Queue holders: A queue holder enqueues any requests for a page until it is voluntarily released. The directory tracks the queue holder and forwards access requests to it. Soul ensures that there is only one queue holder per page at any time, avoiding consistency issues associated with replication. To achieve this, we leverage GCP's SWMR invariant: since only a single writer (i.e., a thread requesting the page with **M** permission) can hold a page at a given time, only one blade (the one hosting the writer) needs to track the queue at that time. While multiple readers (i.e., threads requesting the page with **S** permission) can hold a page concurrently, placing the page at additional readers does not require enqueuing requests (§3.1), i.e., no queue is required for a page that is only requested by readers. However, if a writer subsequently requests a page that is initially held by multiple readers, a queue is created on the compute blade hosting the writer.

Fig. 7 shows three possible cases for a page's wait queue:
Case (i): The wait queue does not exist when no writers are requesting the page (i.e., for a page with **I** or **S** permission that has no waiting writers).

Case (ii): The wait queue is at the current writer (i.e., for a page with **M** permission)

Case (iii): The wait queue is at the next writer when the page is held by one or more readers (i.e., for a page with **S** permission with a waiting writer).

Queue transfers: When a writer attempts to acquire a page (① in Fig. 4), either the wait queue has already transferred to it (described next) or it must create an empty one (transition to **Case (ii)**). When a writer releases the page (③ in Fig. 4), the writer drops the queue if there is no waiting requestor (transition to **Case (i)**). Otherwise, the writer processes the next queue entry and transfers the queue to the next writer. Specifically, if the next requestor is a writer, the queue is transferred to it (transition to **Case (ii)**). If the next requestor is instead a reader (readers) and a writer is waiting behind it (them), the queue is transferred directly to the writer (transition to **Case (iii)**). If there are no writers behind the reader(s), the queue is dropped (transition to **Case (i)**).

Since readers never hold the queue, the dequeue operation happens only in **Case (ii)**, when the current writer is the queue holder. This property ensures no network delays are incurred when dequeuing and processing the next requestor.

Consistency during queue transfers: If the cache directory forwards an access request to a queue holder transferring its queue to another compute blade, should this request be processed by the original queue holder or the next one? To resolve such ambiguities, Soul employs a versioning mechanism to ensure the queue transfer occurs *atomically*. The directory maintains a version number for each page, tracking the number of access requests it has forwarded to the queue holder. The queue holder maintains its own version number to track the number of access requests it has received from the directory. The switch approves a queue transfer only if the queue holder's version number matches the one in the directory. This ensures that all access requests forwarded by the directory must have been processed at the queue holder before the holder initiates the transfer. If the switch denies a transfer, the queue holder retries after receiving the notification from the switch. On a successful transfer, the version numbers at the switch and queue holder are reset to zero.

Storage overheads: Since the wait queue size is bounded by the number of compute blades (n) in the system, we implement it as a fixed-length array of size n. Each $\log n$-bit entry contains the requestor's ID and the request type, adding $n \cdot \log n$ bits for the entire queue. Each directory entry consumes $2 \cdot \log n$ bits to track the queue holder and version number. Since each page is 4KB and each directory entry's size is 76 bits in MIND, Soul's metadata storage overhead for an 8-blade setup is $< 0.2\%$ at the compute blade kernel and $< 8\%$ at the switch ASIC.

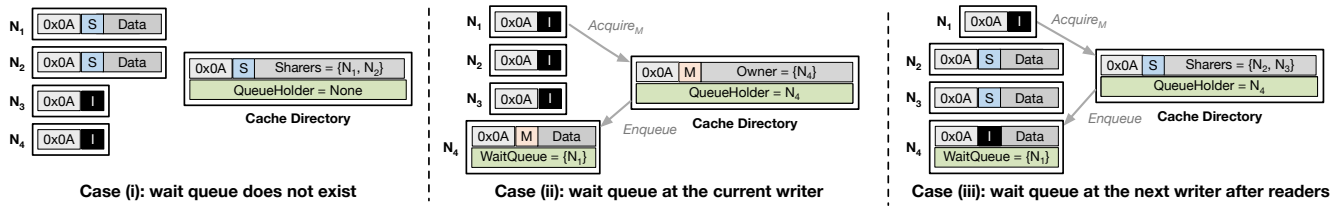


Fig. 7: **Wait queue holders under different cases in Soul (§4.1).** Case (i): the wait queue does not exist without writers; Case (ii): the wait queue is at the current writer; Case (iii): the wait queue is at the next writer after the current readers.

Language	Code Snippets

C	`pthread_rwlock_t lock;`
	`pthread_rwlock_init(&lock, NULL); // Initialize lock`
	`pthread_rwlock_wrlock(&lock);`
	`// Critical section for write lock`
	`pthread_rwlock_unlock(&lock);`
	`pthread_rwlock_rdlock(&lock);`
	`// Critical section for read lock`
	`pthread_rwlock_unlock(&lock);`

Rust	`struct Data { value: u64 }`
	`// Create a linked list of Data`
	`let mut list: LinkedList<Data> = LinkedList::new();`
	`list.push_back(Data{value: 1});`
	`// Protect the linked list with a GCP lock`
	`let lock = RwLock::new(list);`
	`{ // Critical section for write lock`
	`  let w = lock.write().unwrap();`
	`  w.back_mut().unwrap().value = 42;`
	`}`
	`{ // Critical section for read lock`
	`  let r = lock.read().unwrap();`
	`  assert_eq!(r.back().unwrap().value, 42);`
	`}`

Table 2: **Supported standard lock APIs in C and Rust (§4.3).**

4.2 Implementing the Shared Memory List

To overcome the limited memory and processing constraints of programmable switches, we decouple the shared memory list from the directory entries and maintain them on the compute blades, i.e., as metadata associated with a page in the in-kernel cache controller, incurring no storage overhead on the programmable switch ASIC. The compute blades and the switch identify a shared memory list using its lowest address. Our implementation adds 9 bytes to the metadata associated with each page in MIND: 1 byte to track the region length and 8 bytes for the pointer to the next shared memory list entry. As each shared region is at least 4KB, this incurs $\leq 0.3\%$ storage overhead in the compute blade kernel. Consistent with how MIND tracks regular pages, the shared memory lists are tracked as virtual addresses since address translation in MIND occurs *after* coherence resolution [16].

4.3 Soul Lock Library

While Soul provides a high-performance implementation of GCP, ensuring application transparency requires that Soul be usable via standard lock APIs; otherwise, applications would need extensive modifications to exploit GCP's benefits. As such, Soul provides a lock library that serves as a shim atop its GCP implementation (Fig. 6) and exposes standard lock APIs. We find that bridging the gap between GCP interfaces and the standard lock APIs introduces new implementation

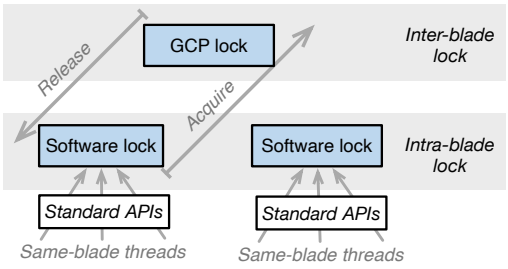


Fig. 8: **Soul lock library's hierarchical design (§4.3).**

challenges. We next describe the standard lock APIs that Soul supports, followed by the challenges and solutions for implementing these APIs in Soul's lock library using GCP interfaces.

Supported lock APIs. Soul lock library supports two standard lock APIs: the POSIX lock (i.e., `pthread_rw_lock`) APIs in C and `std::sync::RwLock` APIs in Rust (Table 2). To use Soul, applications simply link against our lock library rather than the standard libraries (*e.g.*, `libpthread`). The major difference between the two APIs is that while `pthread_rw_lock` provides only locking functionality, `std::sync::RwLock` additionally requires programmers to explicitly specify the lock-protected shared object (*e.g.*, `list` in Table 2) on lock creation for memory safety. Interestingly, this information enables our lock library to transparently implement GCP's combined data optimization (§3.3), that is, the shared object can be prefetched with lock acquisition. Our current lock library does not transparently handle shared objects whose memory footprint changes mid-critical-section (*e.g.*, a Rust write-lock holder calling `clear()` on a lock-protected list, or appending new entries to it); in such cases, the lock holder must destroy the existing shared memory list and create a new one reflecting the updated footprint via `GCP_destroy` followed by `GCP_create` (Table 1). This preserves correctness but is not the most efficient approach: each refresh incurs the broadcast and cache-flush overhead of those interfaces — a current limitation in Soul. A future extension could allow a writer lock holder to modify the shared memory list in place, eliminating this overhead.

Unfortunately, realizing these APIs is not as straightforward as wrapping the corresponding GCP interfaces in Table 1 (*e.g.*, `GCP_acquire with pthread_rwlock_wrlock`). This is

Lock initialization

```
int inter_blade_lock_init(cachepage_t *lock) {
  size_t size = CACHE_PAGE_SIZE;
  return GCP_create(1, &lock, &size);
}
```

Writer lock acquisition

```
int inter_blade_wrlock(cachepage_t *lock) {
  int ret;
  if (ret = GCP_acquire(lock, M))
    return ret;
  while (ret = GCP_is_acquired(lock, M)) {
    if (ret != EAGAIN) return ret;
    /* Spin or yield CPU to another thread */
  }
  return 0;
}
```

Lock release

```
int inter_blade_unlock(cachepage_t *lock) {
  return GCP_release(lock);
}
```

Table 3: **Simplified inter-blade lock implementation for the `pthread_rwlock` API using GCP interfaces (§4.3).**

because GCP only exposes synchronization interfaces between *caches* rather than *software threads* — it is still challenging to distinguish two threads running on the same compute blade, sharing the same cache, as separate lock requestors or holders. Next, we detail how our lock library bridges this gap via *hierarchical locking*.

Supporting thread-level APIs via hierarchical locking. Soul lock library distinguishes threads running on the same compute blades as separate requestors via hierarchical locking. This is akin to lock-cohorting [27], a technique originally used to improve lock acquisition throughput and latency in NUMA machines by prioritizing the lock handover to "nearby" threads (*e.g.*, threads on the same NUMA node) over farther ones (*e.g.*, those on different NUMA nodes). As shown in Fig. 8, a Soul lock acquisition comprises two steps: it first acquires an *intra*-blade lock—implemented as a traditional software-based lock—to enforce SWMR among threads on the same blade, and then acquires an *inter*-blade lock using GCP to enforce SWMR globally. Thus, threads running on the same blade can be identified as different lock requestors or holders since their synchronization is resolved using software-based intra-blade locks. Similar to prior work [31], we bound the number of local lock reuses per global lock acquisition (to 64) to ensure starvation-freedom.

Building inter-blade locks using GCP interfaces. Table 3 shows the simplified inter-blade lock operation for `pthread_rwlock`. On initialization, the page containing the lock variable `lock` is registered as a GCP page via `GCP_create`. On reader or writer lock acquisition (Table 3 only shows writer lock operation for brevity), we first call `GCP_acquire` with corresponding permissions (i.e., Modified (`M`) for write-lock, Shared (`S`) for read-lock), then check for completion of the acquire request via `GCP_is_acquired` — either by spinning on the return value or going to sleep until the return value becomes true. The latter allows the scheduler to context-switch to another thread while the current thread is blocked, and is the default behavior in our implementation. Releasing the lock only requires calling `GCP_release`. We omit the implementation of Rust's `std::sync::RwLock` for brevity, but note that it only differs from `pthread_rwlock` in initialization, where *all* addresses corresponding to the lock-protected shared object (i.e., the shared memory list) are passed to `GCP_create` instead of a single page.

Handling deadlocks. Similar to traditional locks, handling deadlocks is delegated to the application rather than the lock library. For example, an application can enforce certain lock ordering to avoid deadlocks with nested locks. Soul's lock library introduces no new deadlock handling scenarios compared to traditional locks.

4.4 Soul In-kernel Lock Manager

Soul's in-kernel lock manager sits between the lock library and the GCP's DRAM cache controller, also implemented in the kernel in MIND [16] (Fig. 6). It manages allocations of GCP pages and associated metadata (i.e., wait queues, shared memory lists, etc.) in GCP's cache controller via Soul's lock library for two purposes. First, it preserves the integrity and protects the GCP state by rejecting invalid or conflicting requests from Soul's lock library. For instance, it prevents the allocation of two GCP pages that share overlapping memory lists, which can result in locks doomed to deadlock. As another example, it prevents the allocated GCP pages from exceeding a fraction of the system's local DRAM cache size (defaulting to 6% in our implementation) to ensure the system retains effective data caching. If this bound is exceeded, Soul's inter-blade locking protocol falls back to a traditional software-based locking algorithm on top of cache coherence (`pthread_rw_lock` in our implementation).

Second, the lock manager recovers allocated GCP pages (and their associated metadata) upon failure. Soul inherits MIND's failure model, which considers two types of failures: (*i*) *application failures*, where individual processes may crash or terminate unexpectedly, and all resources allocated by the failed process must be reclaimed; and (*ii*) *system failures*, which include failures of compute blades, memory blades, or the network switch. MIND replicates the switch state to facilitate recovery on failure, allowing running processes to resume. However, compute and memory blade failures are not recoverable, analogous to how traditional OSes handle CPU and memory failures. We next describe the additional work performed by Soul's lock manager to recover from application and system failures.

Handling application failures. If a process terminates unexpectedly, the lock manager deallocates all of its allocated GCP pages. This is similar to how Linux resolves unreleased futexes with the robust futex list [75].

Handling system failures. Since the lock manager tracks GCP metadata at the compute blades (i.e., the wait queue and shared memory lists), the only in-switch metadata required for recovery is the wait queue's current holder and version number (§4.1). Our implementation piggybacks on MIND's switch-replication mechanism to recover the GCP directory state upon switch failure.

Packet loss and reordering. Our implementation does not need to cater to failures stemming from coherence messages being dropped or reordered by the network since MIND operates on a standard RoCE deployment over lossless fabric (*e.g.*, via Ethernet PFC [76]), and a reliable, in-order transport (*e.g.*, via RDMA RC [77]).

5 Evaluation

We evaluate Soul to answer the following research questions:

- Compared to state-of-the-art locks, does Soul improve performance for real-world workloads (§5.1)?
- Compared to layered approaches, to what extent does GCP remove redundant coherence messages? (§5.2)
- What are the contributions of GCP's optimizations (§5.3)?

Evaluation setup. We use a cluster with five servers connected via a programmable switch to deploy Soul atop MIND [16]. The switch has a 32-port 6.4 Tbps Tofino programmable switch ASIC. One of the servers is equipped with two 18-core Intel Xeon processors, 384GB of memory, and four Mellanox CX-5 100Gbps NICs, and hosts a single memory-blade VM. The remaining four servers are equipped with two 12-core Intel Xeon processors and two Mellanox CX-5 100Gbps NICs per server, and host two compute blade VMs per server (one per socket), each with 512 MB of DRAM and 10 cores (with the remaining 2 cores dedicated to the OS).

Compared systems. We compare Soul against the two classes of approaches discussed in §2. The first includes several lock algorithms layered atop cache coherence: (i) `MCS` [25], a representative of the queue-based mutex; (ii) `Pthread` [78], the POSIX reader-writer lock with centralized reader-indicators; (iii) `Percpu` [32], a reader-writer lock with decentralized (i.e., per-core) reader-indicators; (iv) `Cohort` [34], a memory hierarchy-aware reader-writer lock.

The second includes standalone locks that bypass coherence. Due to compatibility issues, we were unable to deploy FissLock [49], a state-of-the-art hardware-accelerated rack-scale lock service, on our programmable switch. As such, we reimplemented its logic on our setup, enhancing it with the same lock cohorting technique used in Soul (dubbed `Lock Service`). Consistent with its original design, `Lock Service` does not support combined data and locality optimizations due to switch resource constraints.

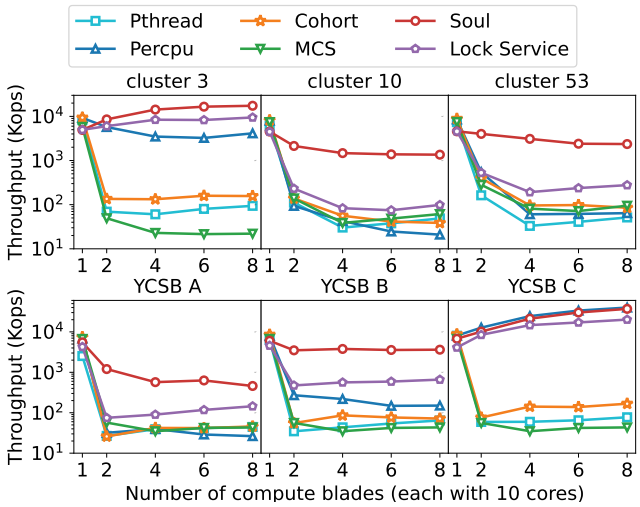


Fig. 9: **Soul throughput scaling for MIND-KVS w/ Twitter cluster 3, 10, 53, and YCSB A, B, C (§5.1).** Y-axis is in log scale.

Real-world workloads. We use two applications: MIND-KVS, the Rust porting of MIND's native in-memory KV store [16] with fine-grained locks, and Kyoto Cabinet [79] with coarse-grained locks. Both use 10 concurrent worker threads per compute blade. Both run on Soul without modifications (§4.3).

MIND-KVS uses a hash table, with a reader-writer lock protecting each hash bucket. We port it to align the hash buckets to 4KB page boundaries, and combine the data in each hash bucket with the lock using Soul's Rust API (§4.3). We evaluate it with both real-world and synthetic workloads. For real-world workloads, we use Twitter's key-value store workloads [58]. We select traces for cluster 3, cluster 10, and cluster 53 for our evaluations as representatives for different proportions of read and write (i.e., `GET`: `SET/ADD/PREPEND`) operations (99% : 1%, 50% : 50%, and 89% : 11%, respectively). For synthetic workloads, we use YCSB workloads A, B, and C [80], corresponding to 50%:50%, 95%:5%, and 100%:0% read-write ratios, respectively.

We run Kyoto Cabinet, a widely used benchmark database system [29, 31, 81, 82], with the TPC-C workload [83], featuring high and low contention (1 and 10 warehouses, respectively). TPC-C transactions hold a global lock to ensure transaction serializability and use the `Pthread` API (§3.3).

5.1 Performance for Real-World Workloads

MIND-KVS. Fig. 9 shows that Soul outperforms the compared systems across various workloads. Soul performs better

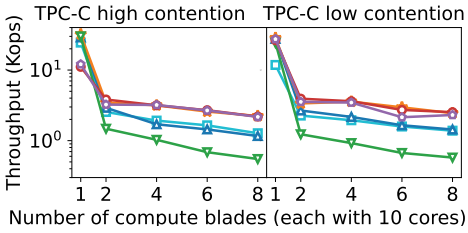


Fig. 10: **Soul throughput scaling for Kyoto Cabinet w/ TPC-C (§5.1).** Y-axis is in log scale.

as the ratio of readers increases, enabling linear scaling for YCSB-C (read-only). It achieves 37.1 Mops at 8 blades, 2-3 orders of magnitude higher than `Pthread`, `Cohort`, `MCS`, and `Lock Service`. These systems write to their lock variables even when acquiring a read lock, causing heavy network-wide cache invalidations. Similarly, `Lock Service` also requires application threads to send acquisition requests over the network for reader locks. In contrast, Soul does not require cache invalidations when there are no writers, as it exploits temporal locality outlined in §3.3. As such, the most frequently accessed locks and data in YCSB can concurrently remain cached across multiple compute blades. Although `Percpu` shows similar performance to Soul for the read-only workload (YCSB-C) due to its per-core reader indicator, it suffers significant performance degradation even with 1% writes (Twitter cluster 3) due to redundant inter-blade traffic in its layered design. Write-heavy workloads (*e.g.*, cluster 10, cluster 53, YCSB-A, and YCSB-B) do not scale with multiple blades with Soul: this is not a limitation of GCP but rather an artifact of long inter-cache latencies over Ethernet. Indeed, GCP may enable performance scaling of these workloads over CXL as we will show in §6.

Kyoto Cabinet. Soul performance is comparable to `Cohort` and `Lock Service` (Fig. 10). Since Kyoto Cabinet's global-exclusive locks allow only one transaction to execute at a time (unlike MIND-KVS's fine-grained locks), its throughput decreases as the number of blades increases. Even so, Soul, `Cohort`, and `Lock Service` outperform other approaches due to their memory-hierarchy-aware design.

5.2 Understanding Soul's Improvements

We investigate Soul's performance improvements — specifically, how it reduces the coherence overheads of layered approaches — by subjecting all schemes under comparison to varying levels of locking contention and read-write ratios. We deploy one thread per compute blade to contend for a single lock; each thread repeatedly acquires the lock, accesses the shared data (4KB), and releases it. We measure the average latency of inter-blade lock and data acquisition (left y-axis in Fig. 11) and the number of cache-coherence transactions per lock and data acquisition (right y-axis in Fig. 11) to highlight the inefficiency incurred by layering. Since `Lock Service` does not use cache coherence for locks, we do not report coherence transactions for it.

Soul observes 100-200 μs lock and data acquisition latency on average at 8 blades — one order of magnitude lower than the fastest compared lock algorithm layered atop cache-coherence. This is because Soul triggers a single coherence transaction for both lock and data acquisition, regardless of the workload. In contrast, locks layered atop coherence trigger coherence transactions proportional to the number of blades (`Pthread`, `Percpu`) or a constant but large number of coherence transactions (`MCS`) as noted in §2.2. While `Lock Service` is not built atop the coherence, it still observes higher latency than Soul because it lacks Soul's combined data optimization and must fetch data over the network separately.

5.3 Effectiveness of Soul Optimizations

We break down the contributions of GCP optimizations (§3.3) by comparing the latency of lock acquisition and data fetch with the same setup in §5.2 for three schemes: (1) Soul with all optimizations, (2) Soul without the optimization for combining lock and data fetch (`w/o combined data opt` in figures), and (3) Soul without the optimization that leverages temporal locality of lock+data (`w/o locality opt`).

The gap between Soul and `w/o locality opt` (Fig. 12) shows that Soul's locality optimization reduces the acquisition latency by 1–2 orders of magnitude by caching the lock and associated data, avoiding expensive network communications. The gap between Soul and `w/o combined data opt` illustrates that Soul's combined data optimization also reduces acquisition latency by avoiding an additional network round trip for data retrieval.

6 Case Study: GCP atop CXL Interconnect

Emerging CXL 3.0 interconnect aims to enable disaggregated shared memory with much finer-grained cache lines and lower latency than the Ethernet-based shared memory we have evaluated. To understand if GCP benefits extend to such interconnects, we conduct a simulation-based study of GCP over CXL 3.0 (dubbed SoulCXL) in the Gem5 [51] simulator, since there is no commercially available hardware for it.

Simulation setup. We evaluate SoulCXL using the same end-to-end applications and compared lock approaches as §5.1. We simulate a cluster with 16 CPU hosts and 1 CXL Global Fabric Attached Memory Device (GFD) connected via a CXL switch. Each CPU host has 8 cores, each employing a 64KB L1 instruction cache and a 64KB L1 data cache. Each core also hosts a 4 MB L2 cache bank. All caches are 2-way set associative. The GFD device hosts sixteen 4 GB DRAM nodes, each with 8× DDR4-2400 channels. Similar to prior CXL-coherence studies [19], we use SynchroTrace [84]

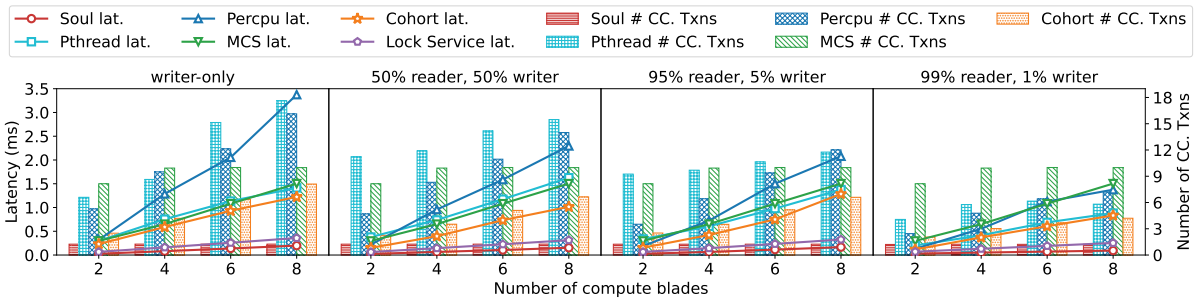


Fig. 11: **Soul avg. latency & # cache-coherence transactions per lock acquisition (§5.2).**

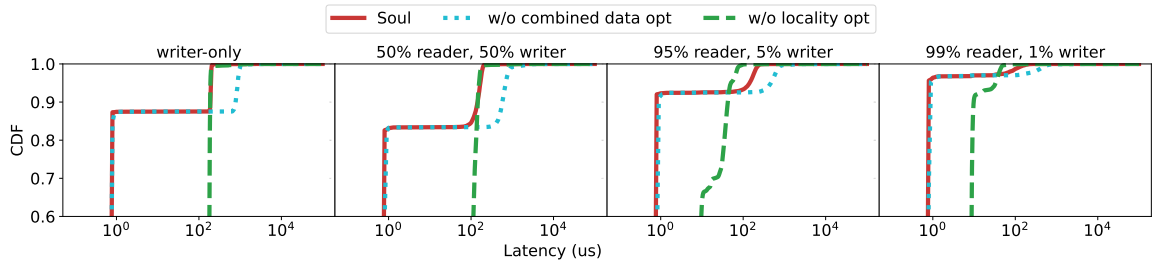


Fig. 12: **Soul latency CDF of lock+data fetch with & without GCP optimizations (§5.3).** X-axis is in log scale.

to capture synchronization-aware traces of all our evaluated workloads and replay them on our simulated CXL cluster. Our simulation models CXL performance as reported in the recent Microsoft study [85], in which the round-trip latency between two hosts is set to an optimistic 300 ns. Since GCP's benefits are more pronounced at larger interconnect latencies, SoulCXL reflects a lower bound on its benefits.

Implementing the wait queue. The low-latency PCIe-based interconnects in CXL 3.0 allow fast access to native coherence protocol message queues at cache controllers and directories. We reuse the CXL.mem M2S channel message buffers in its MOESI protocol to implement GCP wait queues without additional storage overhead. Cache controllers issue `GCP_acquire` requests through the CXL.mem M2S channel, recycling them back to the message buffers if not granted.

Implementing the shared memory list. Since maintaining a general shared memory list in hardware adds non-trivial complexity, our SoulCXL implementation uses a single contiguous memory region instead of a list. We also restrict contiguous regions to power-of-two multiples of 64B to minimize storage overheads. The size of each cache line is embedded within it, using 4 additional bits to indicate the power-of-two multiplier, i.e., `0000` for $2^0 \cdot 64B$ (= 64B) and `1111` for $2^{15} \cdot 64B$ (= 2MB). For regular 64B cache lines, this incurs $< 0.6\%$ storage overhead. Our area estimation using CACTI-7.0 [86] (assuming 22 nm technology) shows that size bits add only 0.0004 mm^2 while the total cache size is 0.0613 mm^2 ($< 0.7\%$) for a 64KB L1 cache.

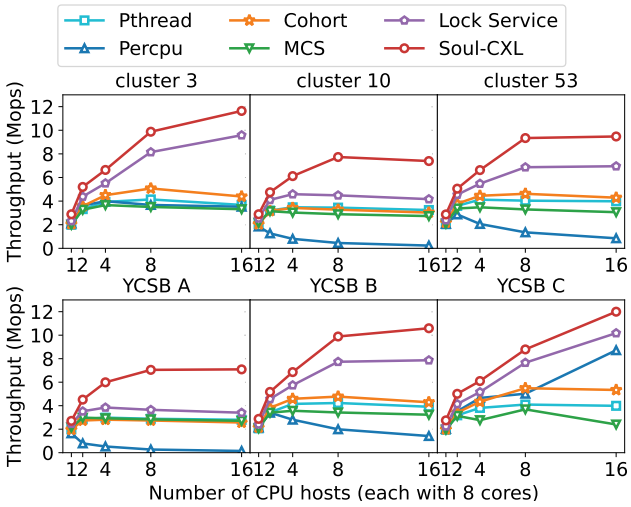


Fig. 13: **SoulCXL throughput scaling for MIND-KVS w/ Twitter cluster 3, 10, 53, and YCSB A, B, C (§6).**

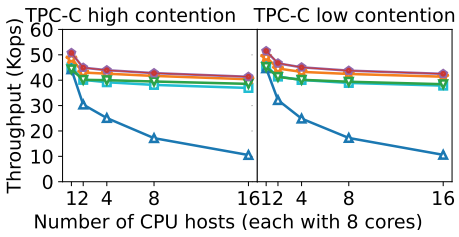


Fig. 14: **SoulCXL throughput scaling for Kyoto Cabinet (§6).**

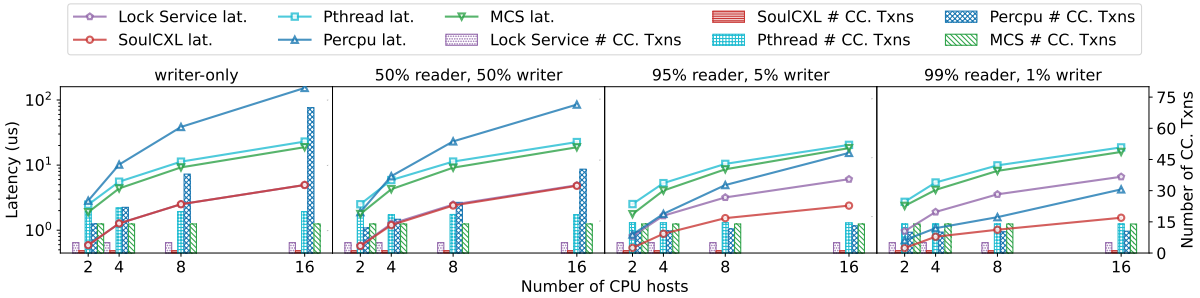


Fig. 15: **SoulCXL avg. latency & # cache-coherence transactions per lock acquire.** Y-axis is in log scale.

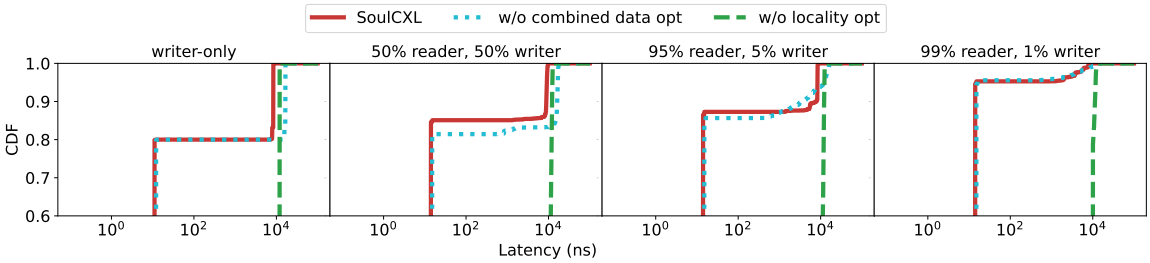


Fig. 16: **SoulCXL latency CDF of lock+data fetch with & without GCP optimizations.** X-axis is in log scale.

6.1 Performance

MIND-KVS. Fig. 13 shows that SoulCXL outperforms the closest-performing compared system by up to 1.7× across Twitter workloads and 2.0× across YCSB workloads. It outperforms `Lock Service` due to the combined data and the locality optimizations. `Percpu` observes much lower throughput than other systems, especially on write-heavy workloads (*e.g.*, cluster 10 and YCSB-A), resulting in worse performance scaling with thread count than on MIND. This is because its per-core lock design is extremely sensitive to contention. CXL's shorter inter-cache latency causes more frequent accesses to lock variables, resulting in higher contention in write-heavy workloads.

Kyoto Cabinet. SoulCXL and `Lock Service` perform similarly because none of SoulCXL's additional optimizations improve performance — Kyoto Cabinet's coarse-grained locks prevent the use of combined data optimization, and since all lock requestors are writers, the locality optimization is not useful either. As with MIND-KVS, `Percpu` observes much lower throughput on CXL than other systems do, due to its sensitivity to lock contention.

Understanding SoulCXL improvements. We conduct the evaluation outlined in §5.3 on the simulated CXL platform. When run on CXL (Fig. 15), while all locks observe lower latency compared to MIND due to CXL's lower latency, they observe more coherence transactions. This is because the lower interconnect latency permits more frequent access to the lock, leading to higher contention. While `Lock Service` observes latency comparable to SoulCXL for write-heavy workloads (*e.g.*, writer-only and 50% writers), it observes higher latency with more readers (*e.g.*, 95% and 99% readers) since SoulCXL's locality optimization reduces latency under writer-reader contentions. `Percpu`'s reduced performance on CXL relative to MIND aligns with our observations in real-world workloads: its per-core lock design is very sensitive to lock contention.

Effectiveness of SoulCXL optimizations. Again, we conduct the evaluation outlined in §5.2 on the simulated CXL platform. The gap between SoulCXL and `w/o locality opt` (Fig. 16) shows that SoulCXL's locality optimization reduces the acquisition latency by 1–2 orders of magnitude by caching the lock and associated data, avoiding expensive network communications. The gap between SoulCXL and `w/o combined data opt` illustrates how SoulCXL's combined data optimization also reduces acquisition latency by avoiding an additional network round-trip for data retrieval.

7 Discussion and Future Research

Supporting flexible queuing policies While Soul's wait queue employs a FIFO queuing policy, it can be extended to support other queuing policies [27, 30, 34] or reader-writer priorities [87, 88]. Since Soul collocates the wait queue and the current lock holder (§4.1), selecting the next candidate to dequeue from the wait queue is performed in the kernel cache controller, which can easily be extended in software. In contrast, supporting flexible queuing policies in the switch ASICs (as proposed in recent hardware lock services [48, 49]) is challenging due to the ASIC's resource constraints.

Supporting other lock APIs. Soul currently only supports basic lock acquire and release APIs as a demonstrator of GCP's performance benefits. It can be extended to support richer lock APIs such as explicit aborts, acquisition with timeouts, try-lock interfaces, etc., via simple modifications to GCP. We leave such extensions to future work.

Related efforts in CPU micro-architectures. Similar to standalone lock services for DSM systems, multi-core architectures have investigated hardware-supported synchronization primitives that either bypass [61,62,89,90] or extend on cache-coherence [63,91–93]. However, the former incurs additional hardware resource consumption and verification complexities, similar to stand-alone lock services for DSMs (§2.2), whereas the latter focuses on specific cache coherence protocols, whereas GCP's applicable to any directory-based cache coherence protocol. Several Hardware Transactional Memory (HTM) systems [94–98] also build on cache coherence protocols. However, they offer different programming models that require modifications to applications. In contrast, GCP's approach ensures transparency for lock-based applications.

8 Conclusion

We observe that lock-based synchronization is a generalization of cache coherence in time and space. We incorporate this insight into a novel *G*eneralized *C*ache-coherence *P*rotocol (GCP) for lock-based synchronization. We provide Soul as an end-to-end system implementation of GCP atop Ethernet-based disaggregated memory and show that it improves performance for real-world applications at scale by 1–2 orders of magnitude over state-of-the-art locks.

Acknowledgments

We would like to thank our shepherd and anonymous reviewers for their valuable comments and insightful feedback. This work is supported in part by NSF Awards #2112562, 2147946, 2118851, 2047220 and a NetApp Faculty Fellowship.

References

[1] Shigeru Shiratake. Scaling and performance challenges of future DRAM. In *International Memory Workshop (IMW)*, 2020.

[2] High throughput computing data center architecture. http://www.huawei.com/ilink/en/download/HW_349607.

[3] The Machine: A new kind of computer. https://www.hpl.hp.com/research/systems-research/themachine/.

[4] Intel rack scale design: Just what is it? https://www.datacenterdynamics.com/en/opinions/intel-rack-scale-design-just-what-is-it/.

[5] Facebook's disaggregated racks strategy provides an early glimpse into next gen cloud computing data center infrastructures. https://dcig.com/2015/01/facebooks-disaggregated-racks-strategy-provides-early-glimpse-next-gen-cloud-computing.html.

[6] Rack-scale computing. https://www.microsoft.com/en-us/research/project/rack-scale-computing/.

[7] Krste Asanović. FireBox: A hardware building block for 2020 warehouse-scale computers. In *USENIX FAST*, 2014.

[8] Stanko Novakovic, Alexandros Daglis, Edouard Bugnion, Babak Falsafi, and Boris Grot. Scale-out NUMA. In *Proc. ACM ASPLOS*, 2014.

[9] Ling Liu, Wenqi Cao, Semih Sahin, Qi Zhang, Juhyun Bae, and Yanzhao Wu. Memory disaggregation: Research problems and opportunities. In *Proc. IEEE ICDCS*, 2019.

[10] Kevin Lim, Jichuan Chang, Trevor Mudge, Parthasarathy Ranganathan, Steven K. Reinhardt, and Thomas F. Wenisch. Disaggregated memory for expansion and sharing in blade servers. In *Proc. ACM/IEEE ISCA*, 2009.

[11] K. Lim, Y. Turner, J. R. Santos, A. AuYoung, J. Chang, P. Ranganathan, and T. F. Wenisch. System-level implications of disaggregated memory. In *Proc. IEEE HPCA*, 2012.

[12] Ahmad Samih, Ren Wang, Christian Maciocco, Mazen Kharbutli, and Yan Solihin. Collaborative memories in clusters: Opportunities and challenges. In *Transactions on Computational Science XXII*. Springer, 2014.

[13] Juncheng Gu, Youngmoon Lee, Yiwen Zhang, Mosharaf Chowdhury, and Kang G. Shin. Efficient memory disaggregation with Infiniswap. In *Proc. USENIX NSDI*, 2017.

[14] Emmanuel Amaro, Christopher Branner-Augmon, Zhihong Luo, Amy Ousterhout, Marcos K. Aguilera, Aurojit Panda, Sylvia Ratnasamy, and Scott Shenker. Can far memory improve job throughput? In *Proc. ACM EuroSys*, 2020.

[15] Yizhou Shan, Yutong Huang, Yilun Chen, and Yiying Zhang. LegoOS: A disseminated, distributed OS for hardware resource disaggregation. In *Proc. USENIX OSDI*, 2018.

[16] Seung-seob Lee, Yanpeng Yu, Yupeng Tang, Anurag Khandelwal, Lin Zhong, and Abhishek Bhattacharjee. MIND: In-network memory management for disaggregated data centers. In *Proc. ACM SOSP*, 2021.

[17] Qingchao Cai, Wentian Guo, Hao Zhang, Divyakant Agrawal, Gang Chen, Beng Chin Ooi, Kian-Lee Tan, Yong Meng Teo, and Sheng Wang. Efficient distributed memory management with RDMA and caching. In *Proc. VLDB Endow.*, 2018.

[18] Qing Wang, Youyou Lu, Erci Xu, Junru Li, Youmin Chen, and Jiwu Shu. Concordia: Distributed shared memory with in-network cache coherence. In *Proc. USENIX FAST*, 2021.

[19] Adarsh Patil, Vijay Nagarajan, Nikos Nikoleris, and Nicolai Oswald. Āpta: Fault-tolerant object-granular CXL disaggregated memory for accelerating FaaS. In *Proc. IEEE/IFIP DSN*, 2023.

[20] Haoran Ma, Yifan Qiao, Shi Liu, Shan Yu, Yuanjiang Ni, Qingda Lu, Jiesheng Wu, Yiying Zhang, Miryung Kim, and Harry Xu. DRust: Language-Guided distributed shared memory with fine granularity, full transparency, and ultra efficiency. In *Proc. USENIX OSDI*, 2024.

[21] CXL Consortium. CXL 3.0 specification. https://www.computeexpresslink.org/download-the-specification, 2022.

[22] Silas Boyd-Wickizer, M. Frans Kaashoek, Robert Tappan Morris, and Nickolai Zeldovich. Non-scalable locks are dangerous. In *Proc. Linux Symposium*, 2012.

[23] Hugo Guiroux, Renaud Lachaize, and Vivien Quéma. Multicore locks: The case is not closed yet. In *Proc. USENIX ATC*, 2016.

[24] Tudor David, Rachid Guerraoui, and Vasileios Trigonakis. Everything you always wanted to know about synchronization but were afraid to ask. In *Proc. ACM SOSP*, 2013.

[25] John M. Mellor-Crummey and Michael L. Scott. Synchronization without contention. In *Proc. ACM ASPLOS*, 1991.

[26] Travis S. Craig. Building FIFO and priority-queuing spin locks from atomic swap. Technical report, Department of Computer Science, University of Washington, 1993.

[27] David Dice, Virendra J. Marathe, and Nir Shavit. Lock cohorting: A general technique for designing NUMA locks. In *Proc. ACM PPoPP*, 2012.

[28] Milind Chabbi, Michael Fagan, and John Mellor-Crummey. High performance locks for multi-level NUMA systems. In *Proc. ACM PPoPP*, 2015.

[29] Dave Dice and Alex Kogan. Compact NUMA-aware locks. In *Proc. ACM EuroSys*, 2019.

[30] Sanidhya Kashyap, Irina Calciu, Xiaohe Cheng, Changwoo Min, and Taesoo Kim. Scalable and practical locking with shuffling. In *Proc. ACM SOSP*, 2019.

[31] Rafael Lourenco de Lima Chehab, Antonio Paolillo, Diogo Behrens, Ming Fu, Hermann Härtig, and Haibo Chen. CLoF: A compositional lock framework for multi-level NUMA systems. In *Proc. ACM SOSP*, 2021.

[32] Linux kernel brlock. https://lwn.net/Articles/378911,2010.

[33] Ran Liu, Heng Zhang, and Haibo Chen. Scalable read-mostly synchronization using passive reader-writer locks. In *Proc. USENIX ATC*, 2014.

[34] Irina Calciu, Dave Dice, Yossi Lev, Victor Luchangco, Virendra J. Marathe, and Nir Shavit. NUMA-aware reader-writer locks. In *Proc. ACM PPoPP*, 2013.

[35] Dave Dice and Alex Kogan. BRAVO—Biased locking for reader-writer locks. In *Proc. USENIX ATC*, 2019.

[36] Shin-Yeh Tsai, Yizhou Shan, and Yiying Zhang. Disaggregating persistent memory and controlling them remotely: An exploration of passive disaggregated key-value stores. In *Proc. USENIX ATC*, 2020.

[37] Jiacheng Shen, Pengfei Zuo, Xuchuan Luo, Tianyi Yang, Yuxin Su, Yangfan Zhou, and Michael R Lyu. FUSEE: A fully memory-disaggregated key-value store. In *Proc. USENIX FAST*, 2023.

[38] Pengfei Li, Yu Hua, Pengfei Zuo, Zhangyu Chen, and Jiajie Sheng. ROLEX: A scalable RDMA-oriented learned key-value store for disaggregated memory systems. In *Proc. USENIX FAST*, pages 99–114, 2023.

[39] Jianguo Wang and Qizhen Zhang. Disaggregated database systems. In *Proc. ACM SIGMOD*, 2023.

[40] Yingqiang Zhang, Chaoyi Ruan, Cheng Li, Xinjun Yang, Wei Cao, Feifei Li, Bo Wang, Jing Fang, Yuhui Wang, Jingze Huo, et al. Towards cost-effective and elastic cloud database deployment via memory disaggregation. *Proc. VLDB*, 2021.

[41] Chaoyi Ruan, Yingqiang Zhang, Chao Bi, Xiaosong Ma, Hao Chen, Feifei Li, Xinjun Yang, Cheng Li, Ashraf Aboulnaga, and Yinlong Xu. Persistent memory disaggregation for cloud-native relational databases. In *Proc. ACM ASPLOS*, 2023.

[42] Wei Cao, Yingqiang Zhang, Xinjun Yang, Feifei Li, Sheng Wang, Qingda Hu, Xuntao Cheng, Zongzhi Chen, Zhenjun Liu, Jing Fang, et al. PolarDB serverless: A cloud native database for disaggregated data centers. In *Proc. ACM SIGMOD*, 2021.

[43] Compute Express Link. https://www.computeexpresslink.org.

[44] Kai Li and Paul Hudak. Memory coherence in shared virtual memory systems. *ACM Trans. Comput. Syst.*, 1989.

[45] Peter Keleher, Alan L. Cox, Sandhya Dwarkadas, and Willy Zwaenepoel. Tread marks: Distributed shared memory on standard workstations and operating systems. In *USENIX Winter 1994 Technical Conference*, 1994.

[46] J. K. Bennett, J. B. Carter, and W. Zwaenepoel. Munin: Distributed shared memory based on type-specific memory coherence. In *Proc. ACM PPoPP*, 1990.

[47] Brian N. Bershad, Matthew J. Zekauskas, and Wayne A. Sawdon. The Midway distributed shared memory system. Technical report, Carnegie Mellon University, 1993.

[48] Zhuolong Yu, Yiwen Zhang, Vladimir Bravermann, Mosharaf Chowdhury, and Xin Jin. NetLock: Fast, centralized lock management using programmable switches. In *Proc. ACM SIGCOMM*, 2020.

[49] Hanze Zhang, Ke Cheng, Rong Chen, and Haibo Chen. Fast and scalable in-network lock management using lock fission. In *Proc. USENIX OSDI*, 2024.

[50] Vijay Nagarajan, Daniel J Sorin, Mark D Hill, and David A Wood. Coherence basics. In *A Primer on Memory Consistency and Cache Coherence*. Morgan & Claypool Publishers, 2020.

[51] gem5. https://www.gem5.org/.

[52] Marcos K. Aguilera, Nadav Amit, Irina Calciu, Xavier Deguillard, Jayneel Gandhi, Pratap Subrahmanyam, Lalith Suresh, Kiran Tati, Rajesh Venkatasubramanian, and Michael Wei. Remote memory in the age of fast networks. In *Proc. ACM SoCC*, 2017.

[53] Chenxi Wang, Yifan Qiao, Haoran Ma, Shi Liu, Wenguang Chen, Ravi Netravali, Miryung Kim, and Guoqing Harry Xu. Canvas: Isolated and adaptive swapping for Multi-Applications on remote memory. In *Proc. USENIX NSDI*, 2023.

[54] Yifan Qiao, Chenxi Wang, Zhenyuan Ruan, Adam Belay, Qingda Lu, Yiying Zhang, Miryung Kim, and Guoqing Harry Xu. Hermit: Low-Latency, High-Throughput, and transparent remote memory via Feedback-Directed asynchrony. In *Proc. USENIX NSDI*, 2023.

[55] Yueyang Pan, Yash Lala, Musa Unal, Yujie Ren, Seungseob Lee, Abhishek Bhattacharjee, Anurag Khandelwal, and Sanidhya Kashyap. Scalable far memory: Balancing faults and evictions. In *Proc. ACM SOSP*, 2025.

[56] Seung-seob Lee, Jachym Putta, Ziming Mao, and Anurag Khandelwal. Spirit: Fair allocation of interdependent resources in remote memory systems. In *Proc. ACM SOSP*, 2025.

[57] Mingxing Zhang, Teng Ma, Jinqi Hua, Zheng Liu, Kang Chen, Ning Ding, Fan Du, Jinlei Jiang, Tao Ma, and Yongwei Wu. Partial failure resilient memory management system for (CXL-based) distributed shared memory. In *Proc. ACM SOSP*, 2023.

[58] Juncheng Yang, Yao Yue, and K. V. Rashmi. A large scale analysis of hundreds of in-memory cache clusters at Twitter. In *Proc. USENIX OSDI*, 2020.

[59] John M. Mellor-Crummey and Michael L. Scott. Scalable reader-writer synchronization for shared-memory multiprocessors. In *Proc. ACM PPoPP*, 1991.

[60] Christina Giannoula, Nandita Vijaykumar, Nikela Papadopoulou, Vasileios Karakostas, Ivan Fernandez, Juan Gómez-Luna, Lois Orosa, Nectarios Koziris, Georgios Goumas, and Onur Mutlu. SynCron: Efficient synchronization support for near-data-processing architectures. In *Proc. IEEE HPCA*, 2021.

[61] Enrique Vallejo, Ramon Beivide, Adrian Cristal, Tim Harris, Fernando Vallejo, Osman Unsal, and Mateo Valero. Architectural support for fair reader-writer locking. In *Proc. IEEE/ACM MICRO*, 2010.

[62] Weirong Zhu, Vugranam C Sreedhar, Ziang Hu, and Guang R. Gao. Synchronization state buffer: supporting efficient fine-grain synchronization on many-core architectures. In *Proc. ACM/IEEE ISCA*, 2007.

[63] James R. Goodman, Mary K. Vernon, and Philip J. Woest. Efficient synchronization primitives for large-scale cache-coherent multiprocessors. In *Proc. ACM ASPLOS*, 1989.

[64] Xin Jin, Xiaozhou Li, Haoyu Zhang, Robert Soulé, Jeongkeun Lee, Nate Foster, Changhoon Kim, and Ion Stoica. NetCache: Balancing key-value stores with fast in-network caching. In *Proc. ACM SOSP*, 2017.

[65] Nicolai Oswald, Vijay Nagarajan, and Daniel J. Sorin. ProtoGen: Automatically generating directory cache

coherence protocols from atomic specifications. In *Proc. ACM/IEEE ISCA*, 2018.

[66] A. Agarwal, R. Simoni, J. Hennessy, and M. Horowitz. An evaluation of directory schemes for cache coherence. In *Proc. ACM/IEEE ISCA*, 1988.

[67] Mark S. Papamarcos and Janak H. Patel. A low-overhead coherence solution for multiprocessors with private cache memories. In *Proc. ACM/IEEE ISCA*, 1984.

[68] James Archibald and Jean-Loup Baer. Cache coherence protocols: Evaluation using a multiprocessor simulation model. *ACM Trans. Comput. Syst.*, 1986.

[69] J. Dorsey, Shawn Searles, M. Ciraula, S. Johnson, N. Bujanos, D. Wu, M. Braganza, S. Meyers, E. Fang, and R. Kumar. An integrated quad-core Opteron processor. In *Proc. IEEE ISSCC*, 2007.

[70] David L. Dill. The Murphi verification system. In *International Conference on Computer Aided Verification*, 1996.

[71] Nicolai Oswald, Vijay Nagarajan, and Daniel J. Sorin. Hieragen: Automated generation of concurrent, hierarchical cache coherence protocols. In *Proc. ACM/IEEE ISCA*, 2020.

[72] Nicolai Oswald, Vijay Nagarajan, Daniel J. Sorin, Vasilis Gavrielatos, Theo Olausson, and Reece Carr. Heterogen: Automatic synthesis of heterogeneous cache coherence protocols. In *Proc. IEEE HPCA*, 2022.

[73] Byn Choi, Rakesh Komuravelli, Hyojin Sung, Robert Smolinski, Nima Honarmand, Sarita V. Adve, Vikram S. Adve, Nicholas P. Carter, and Ching-Tsun Chou. DeNovo: Rethinking the memory hierarchy for disciplined parallelism. In *Proc. PACT*, 2011.

[74] Jialin Li, Jacob Nelson, Ellis Michael, Xin Jin, and Dan R. K. Ports. Pegasus: Tolerating skewed workloads in distributed storage with in-network coherence directories. In *Proc. USENIX OSDI*, 2020.

[75] The kernel development community. The robust futex ABI. https://docs.kernel.org/locking/robust-futex-ABI.html.

[76] Yibo Zhu, Haggai Eran, Daniel Firestone, Chuanxiong Guo, Marina Lipshteyn, Yehonatan Liron, Jitendra Padhye, Shachar Raindel, Mohamad Haj Yahia, and Ming Zhang. Congestion control for large-scale RDMA deployments. In *Proc. ACM SIGCOMM*, 2015.

[77] InfiniBand Trade Association. InfiniBand architecture specification, volume 1: General specifications, 2020. Release 1.4.

[78] POSIX thread library reader-writer lock. https://linux.die.net/man/3/pthread_rwlock_init.

[79] Kyoto Cabinet: a straightforward implementation of DBM. http://fallabs.com/kyotocabinet.

[80] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with YCSB. In *Proc. ACM SoCC*, 2010.

[81] Dave Dice, Alex Kogan, Yossi Lev, Timothy Merrifield, and Mark Moir. Adaptive integration of hardware and software lock elision techniques. In *Proc. ACM SPAA*, 2014.

[82] Jonas Oberhauser, Rafael Lourenco de Lima Chehab, Diogo Behrens, Ming Fu, Antonio Paolillo, Lilith Oberhauser, Koustubha Bhat, Yuzhong Wen, Haibo Chen, Jaeho Kim, and Viktor Vafeiadis. Vsync: Push-button verification and optimization for synchronization primitives on weak memory models. In *Proc. ACM ASPLOS*, 2021.

[83] Transaction Processing Performance Council. TPC-C. http://www.tpc.org/tpcc/, 2020.

[84] Karthik Sangaiah, Michael Lui, Radhika Jagtap, Stephan Diestelhorst, Siddharth Nilakantan, Ankit More, Baris Taskin, and Mark Hempstead. SynchroTrace: Synchronization-aware architecture-agnostic traces for lightweight multicore simulation of CMP and HPC workloads. *ACM Trans. Archit. Code Optim.*, 2018.

[85] Huaicheng Li, Daniel S. Berger, Lisa Hsu, Daniel Ernst, Pantea Zardoshti, Stanko Novakovic, Monish Shah, Samir Rajadnya, Scott Lee, Ishwar Agarwal, Mark D. Hill, Marcus Fontoura, and Ricardo Bianchini. Pond: CXL-based memory pooling systems for cloud platforms. In *Proc. ACM ASPLOS*, 2023.

[86] Naveen Muralimanohar, Rajeev Balasubramonian, and Norm Jouppi. Optimizing NUCA organizations and wiring alternatives for large caches with CACTI 6.0. In *Proc. IEEE/ACM MICRO*, 2007.

[87] P. J. Courtois, F. Heymans, and D. L. Parnas. Concurrent control with "readers" and "writers". *Commun. ACM*, 1971.

[88] Björn B. Brandenburg and James H. Anderson. Spin-based reader-writer synchronization for multiprocessor real-time systems. *Real-Time Syst.*, 2010.

[89] B.E. Saglam and V.J. Mooney. System-on-a-chip processor synchronization support in hardware. In *Proceedings Design, Automation and Test in Europe. Conference and Exhibition 2001*, 2001.

[90] Men-Chow Chiang. *Memory system design for bus-based multiprocessors*. PhD thesis, University of Wisconsin–Madison, 1992.

[91] IEEE. IEEE standard for scalable coherent interface (SCI). *IEEE Std 1596-1992*, 1993.

[92] R. Rajwar, A. Kagi, and J.R. Goodman. Improving the throughput of synchronization by insertion of delays. In *Proc. IEEE HPCA*, 2000.

[93] Hyojin Sung, Rakesh Komuravelli, and Sarita V. Adve. DeNovoND: efficient hardware support for disciplined non-determinism. In *Proc. ACM ASPLOS*, 2013.

[94] Maurice Herlihy and J. Eliot B. Moss. Transactional memory: architectural support for lock-free data structures. In *Proc. ACM/IEEE ISCA*, 1993.

[95] J.M. Stone, H.S. Stone, P. Heidelberger, and J. Turek. Multiple reservations and the oklahoma update. *IEEE Parallel & Distributed Technology: Systems & Applications*, 1993.

[96] Dave Christie, Jae-Woong Chung, Stephan Diestelhorst, Michael Hohmuth, Martin Pohlack, Christof Fetzer, Martin Nowack, Torvald Riegel, Pascal Felber, Patrick Marlier, and Etienne Rivière. Evaluation of AMD's advanced synchronization facility within a complete transactional memory stack. In *Proc. ACM EuroSys*, 2010.

[97] David Dice, Yossi Lev, Mark Moir, and Daniel Nussbaum. Early experience with a commercial hardware transactional memory implementation. In *Proc. ACM ASPLOS*, 2009.

[98] Peter Damron, Alexandra Fedorova, Yossi Lev, Victor Luchangco, Mark Moir, and Daniel Nussbaum. Hybrid transactional memory. In *Proc. ACM ASPLOS*, 2006.

ARTIFACT EVALUATED
USENIX
AVAILABLE

Shaving the Peaks: Taming Tail Latency for Managed Workloads via Disaggregated Garbage Collection

Hongtao Lyu Yuhan Li Mingyu Wu*
Institute of Parallel and Distributed Systems, Shanghai Jiao Tong University

Abstract

Language runtimes are essential systems commonly used in multi-tenant cloud scenarios, such as interactive web services and other cloud workloads. They usually provide memory management services, or garbage collection (GC), to automatically reclaim memory and reduce the labor work of application developers. Recent concurrent collectors allow GC to co-run with application threads (mutators), which reduces application pauses and intends to improve the applications' tail latency. However, this work observes that periodic GC workloads remain a primary source of long tail latency, particularly in resource-constrained multi-tenant environments. In such settings, GC threads consume significant CPU resources, leading to severe performance contention with mutators.

To resolve the contention, this work presents DGC, a disaggregated GC architecture that exposes GC as an external service. DGC decouples the most costly marking phase in concurrent GC and offloads it to a disaggregated marking engine. Through a co-design of the GC marking algorithm and an RDMA-based software paging mechanism, DGC's disaggregated marking engine achieves performance on par with local execution while offloading marking to a remote node. To improve resource utilization, DGC introduces a global GC orchestrator to serve multiple runtimes while minimizing the conflicts due to the overlapping of individual GC triggering points. DGC is implemented on the OpenJDK HotSpot Java virtual machine, and the evaluation results on representative latency-sensitive applications show that DGC reduces P99 latency by up to 64.4% under moderate workloads and improves the peak goodput by up to 24.0%.

1 Introduction

Language runtimes provide execution environments for high-level programming languages, including OpenJDK HotSpot (Java), V8 (JavaScript), ART (Kotlin), etc. Such runtimes power a wide range of mission-critical workloads in modern data centers, including cloud services, large-scale distributed systems, and enterprise management platforms such as enterprise resource planning (ERP). One of the most important features in language runtimes is their automatic memory management module (namely garbage collection or GC), which helps identify unused memory resources and reclaim them for later reuse. Recent garbage collectors usually embrace a *concurrent* design, allowing GC threads to execute simultaneously with application threads (or *mutators*). Compared with other collectors inducing stop-the-world (STW) pauses, concurrent collectors do not require mutators to pause during GC and are designed for latency-sensitive applications.

However, the advantages of concurrent collectors are severely compromised in modern multi-tenant cloud environments, where applications are typically containerized with strict CPU resource limits to maximize resource utilization and optimize overall infrastructure cost. In such scenarios, the concurrent execution model creates a fundamental conflict: CPU-intensive GC marking threads must compete directly with mutators for the CPU resources. Our analysis confirms that this resource contention causes application-side CPU starvation. For instance, under the SPECjbb2015 benchmark, Shenandoah (a representative concurrent collector in OpenJDK) introduces a severe latency spike when GC marking is active: as the application's available CPU share drops to 60% of its normal demand, its average latency rises by more than an order of magnitude. However, once GC threads become inactive, the CPU becomes underutilized under the same workload.

To address the issues of resource utilization fluctuations and tail latency caused by periodic GC tasks, this work proposes a novel architecture that decouples GC tasks from the original runtime and abstracts them into a dedicated service. This approach significantly reduces interference with mutators in the original runtimes (referred to as *served runtimes*). Additionally, the service can be time-shared by multiple runtimes; a dedicated scheduler staggers their GC bursts to avoid overlap, smoothing peak demand and improving overall system resource utilization. Thanks to recent high-performance interconnection technologies, the service can also be initiated

*Corresponding author: mingyuwu@sjtu.edu.cn.

remotely and serve language runtimes running on different physical machines. Inspired by the concept of resource pooling and disaggregation, we name this architecture as *disaggregated GC* (DGC).

The design of DGC mainly faces two challenges:

How to achieve correct and efficient remote marking? Cross-machine concurrent marking inherently introduces consistency and data plane overheads typical of distributed systems. To address these challenges, DGC proposes a *disaggregated marking engine*. It first employs a lightweight Snapshot-at-the-Beginning (SATB) protocol, which minimizes the synchronization overhead between GC markers and application threads. Second, to address data plane inefficiencies, it introduces a co-design of the marking logic and the paging subsystems. DGC utilizes the runtime's coarse-grained memory management unit (namely *regions*) to construct a software-based RDMA paging system, which reduces the address translation cost associated with software page tables. Furthermore, DGC proposes a marking algorithm to operate on incomplete object graphs, which ensures overlapping between object marking and RDMA-based data retrieving and thus achieves marking performance comparable to local execution.

How to serve multiple runtime instances simultaneously? Existing language runtimes schedule GC assuming it monopolizes the computation resources. However, the assumption does not hold with DGC, where the computation resources are shared by multiple runtimes, and relying on the original scheduling leads to suboptimal results. To address this, DGC employs lightweight memory allocation monitors embedded within each runtime to formulate a forward-looking DGC orchestration plan. This design enables DGC to fully utilize the marking offload capabilities to accelerate participating runtimes, all while avoiding contention between concurrent DGC tasks.

DGC is implemented on OpenJDK HotSpot, a widely used JVM. It is based atop Shenandoah, a concurrent garbage collector whose SATB marking design is representative and can be easily migrated to other collectors [12, 59]. The evaluation results on various latency-sensitive applications show that DGC can achieve significant (up to 64.4%) P99 latency reduction against two state-of-the-art collectors with identical CPU resource budgets. Despite the network overhead, its remote version can reach 60.3% P99 latency reduction under moderate workloads.

To summarize, the contribution of this work includes the following:

- A comprehensive analysis to illustrate the periodic resource contention between GC and application threads, especially for multi-tenant cloud scenarios. (§ 3)
- DGC, a disaggregated GC architecture to elastically serve GC tasks in an external instance to reduce GC-mutator conflicts. (§ 4-6)
- Experiments on various latency-critical workloads to show how DGC improves applications' tail latency (§ 8).

2 Background

Garbage Collection. Garbage Collection (GC) automates memory management by dynamically tracking object liveness and reclaiming unused resources. This mechanism relieves developers from low-level manual administration and prevents memory-related bugs, such as leaks, which are prevalent in complex software systems.

The mainstream GC algorithm is *tracing*, which identifies live objects by traversing the object graph and marking all reachable ones. During marking, GC threads start with root objects (referred to as *GC roots*) known to be alive (e.g., used by on-stack references) and traverse the heap (or a part of the heap) to locate live objects. After marking, *non-moving* collectors reclaim areas consumed by dead objects and organize them into free lists, while *moving* collectors copy live objects to adjacent locations (known as *evacuation*) to create a large and contiguous free space.

Concurrent Marking. Performance overhead is a key concern of GC. Owing to the consistency issues between GC threads and application threads (referred to as *mutators*), early garbage collectors relied on stop-the-world (STW) pauses, suspending all application threads during object marking, which significantly affects the application latency. In contrast, modern garbage collectors employ advanced techniques such as barriers and sophisticated algorithmic designs, enabling concurrent execution with mutators, especially for the marking phase (namely *concurrent marking*). Nowadays, many language runtimes (OpenJDK HotSpot [41], V8 [17], Go runtime [16], ART [15], etc.) have shifted towards concurrent marking. The reduced pause times achieved through concurrent marking are crucial for latency-critical applications, such as databases and web services, which are mostly written in high-level languages [19].

Marking Algorithm. Marking algorithms rely on the tri-color abstraction to reason about graph traversal correctness. Objects are categorized into three states:

- *White:* Unvisited candidates. Any object remaining white upon termination is unreachable and eligible for reclamation.
- *Grey:* Live objects that have been discovered but not yet scanned for outgoing references. These reside in the collector's work buffers (e.g., queues) and represent the active wavefront of the traversal.
- *Black:* Fully processed objects whose direct children have all been identified.

The marking process proceeds iteratively: the collector pops a grey object, marks its children grey, and colors the object black. To guarantee convergence and idempotency, modern collectors (e.g., Shenandoah) utilize a global mark bitmap. This atomic side-table ensures that each reachable object transitions from white to grey exactly once, preventing redundant processing of cycles or shared references.

Managed Application Clusters and Multi-tenancy. Modern cloud-native deployments are increasingly organized as clus-

Table 1: Results of SPECjbb2015 for different GCs

	Shenandoah	ZGC	G1
(A): Critical-jOPS (req/s) vs. GC Threads			
2 Threads	4937	3481	2530
4 Threads	6710	5512	3745
6 Threads	6569	6262	5208
8 Threads	**6819**	**6474**	**6288**
(B): Pause Metrics (under Critical-jOPS, 8 Threads)			
Pause Count	45	71	49
Avg Pause (ms)	1.91	0.04	51.44
Total Pause (ms)	85.79	3.20	2520.59

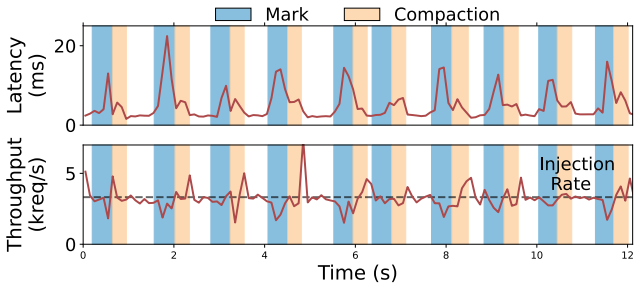
ters of managed application instances such as microservices, FaaS handlers, and partitioned monoliths. These instances share host resources and operate under strict per-instance budgets. Unlike traditional monoliths that rely on vertical scaling (scale-up) of massive, singular instances, modern hyperscale applications are decomposed into thousands of loosely coupled, fine-grained service units. This architectural decomposition is primarily driven by the need for horizontal scalability (scale-out). By breaking down complex logic into smaller components, operators can dynamically provision or terminate replicas of standardized service instances to match real-time traffic fluctuations, ensuring agility and fault tolerance.

Consequently, the deployment strategy has shifted towards co-locating high densities of these smaller service instances to maximize cluster-wide resource utilization. To achieve efficient bin-packing, these instances are typically constrained to moderate specifications (According to a recent report, 82.8% of monitored JVMs use between 1 and 8 cores, and 76.2% are allocated less than 4 GB of memory [39].) While this model optimizes infrastructure costs, it fundamentally constrains language runtimes' execution environment: the surplus CPU cycles previously relied upon for background maintenance tasks are largely eliminated, creating contention between application logic and runtime services like GC.

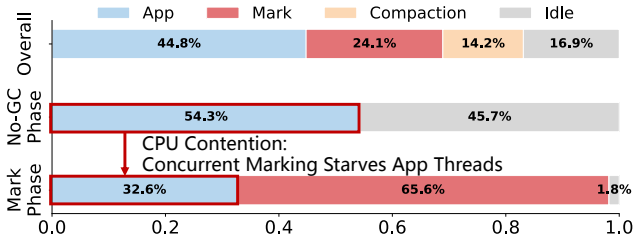
3 Motivation

In this section, we use a set of experiments to quantitatively demonstrate (1) how concurrent marking competes with application threads for CPU resources in resource-constrained environments and leads to performance fluctuations, and (2) why existing configuration adjustments cannot effectively solve these issues.

Experiment Setup. We conducted our experiments using SPECjbb2015 (abbreviated as SPECjbb), a benchmark representing typical online shopping workloads, across three GCs in OpenJDK 17. We provisioned the SPECjbb instance with 8 cores and 4 GB of RAM to reflect a resource-limited, multi-tenant environment. Notably, the 4 GB limit provides twice the minimum memory required by the workload, guaranteeing adequate space for allocation and garbage collection. SPECjbb



(a) SPECjbb2015 latency and throughput details. Note that SPECjbb2015 employs a queue for unhandled requests, which leads to rebounds in throughput after CPU contention finishes.



(b) CPU utilization breakdown during different GC stages.

Figure 1: Performance degradation caused by concurrent GC.

incorporates a built-in load-probing mechanism that automatically evaluates latency characteristics under varying loads and reports the maximum throughput satisfying its latency constraints (referred to as critical-jOPS).

Three collectors are selected for evaluation. *G1* serves as the mostly-STW baseline, utilizing a generational design with frequent pauses for young-generation reclamation. In contrast, *Shenandoah* and *ZGC* represent the concurrent category; they leverage sophisticated consistency protocols to achieve a "pause-less" execution model, confining STW interruptions strictly to sub-millisecond phase transitions.

Performance Analysis. The results presented in Table 1 reveal a counterintuitive phenomenon: although concurrent collectors succeed in reducing STW pauses, they yield only marginal improvements in critical-jOPS (3-8%) compared to G1. To investigate the root cause of this phenomenon, we conducted a detailed performance analysis of Shenandoah (selected over ZGC due to its lighter-weight concurrent algorithm and better critical-jOPS) under the IR=6819 configuration (IR = injection rate, in requests/s). In order to get the performance details, we instrumented SPECjbb to collect fine-grained performance metrics during testing, correlating them with system resource utilization statistics (summarized in Figure 1).

The results uncover a critical pattern: during non-GC periods, CPU utilization remains consistently low (45.7% Idle), indicating CPU underutilization. However, during concurrent marking, total CPU usage spikes to 98.2% as the application's CPU share drops from 53.3% to 32.6%, resulting in a sharp drop in application throughput (47%) and increased average

latency (10.5×). This observation suggests an explanation for the limited critical-jOPS improvement: while traditional STW collectors completely hand over CPU resources to either application or GC threads at different phases, concurrent GCs periodically introduce heavy GC tasks colocated with mutators. Although pauses have been reduced, those GC tasks still interfere with application threads, inducing recurring resource contention. These observations also align with prior reports [60, 66].

Why not overprovisioning? The most straightforward solution to this issue is CPU overprovisioning—allocating sufficient cores to accommodate peak demand from both application and GC threads. This approach would prevent performance fluctuations caused by CPU contention when GC activates, as incidentally observed in our Figure 1 experiment, where the system automatically settled at 6,819 req/s, a moderate load level that balanced jOPS and P99 latency. However, while CPU overprovisioning is simple and effective, it introduces significant efficiency concerns. As demonstrated in our earlier results, this method leads to severe underutilization during non-GC periods, with CPU utilization dropping below 50%—a substantial resource waste. A common variant is to pin GC threads to dedicated cores. This avoids contention but inherits the same idleness: the pinned cores must be sized for peak marking demand and sit idle outside marking windows.

An alternative overprovisioning solution is to grant extra memory to the application. For garbage-collected applications, expanding memory quotas can reduce GC frequency and consequently mitigate its performance impact. However, this approach also has its limitations. The CPU-to-memory ratio of current machines is essentially fixed in existing public clouds or private data centers, where memory capacity is already the primary bottleneck during resource allocation[19, 46]. Therefore, adding more idle memory to the application only for GC frequency reduction is not an effective solution.

Why not reduce GC Threads? GC Thread number is one of the most important parameters for GC performance tuning. Given that the root cause of CPU load fluctuations lies in the periodic activation of GC tasks, one possible solution is to reduce GC threads to stretch GC execution time, thereby smoothing overall system load. Table 2 presents performance data from SPECjbb tests under four different GC thread configurations.

Reducing GC threads from 8 to 6 and 4 initially demonstrates a decline in GC throughput (defined as live object size divided by marking time), confirming the ability to stretch GC duration through this method. However, the system exhibits an 11% increase in GC frequency (8 to 4), while P99 latency remains statistically unchanged and P50 latency even increases by 20%. This counterintuitive result is caused by *floating garbage*. Modern concurrent collectors such as Shenandoah and ZGC take a logical snapshot of the heap at the start of each marking cycle and treat any object that dies mid-cycle as live. Reducing the number of GC threads stretches the marking phase, enlarging this window so that more short-lived objects survive the cycle as floating garbage. Because each cycle reclaims less memory, the collector must run cycles more frequently to keep up with allocation. The CPU savings from fewer GC threads are thus offset by the higher GC frequency.

Table 2: Tail latency with different numbers of marker threads. Tested on SPECjbb2015 (PRESET mode, IR=6819 req/s).

GC Metrics (marking)				App Latency (ms ± 95% CI)		
Thr.	#GC	GB/s	CPU%	P50	P90	P99
2T	41	1.04	24.5%	$1650_{(oom)}$	$3600_{(oom)}$	$11500_{(oom)}$
4T	50	1.98	46.2%	4.8 ± 0.1	19.3 ± 3.9	76.7 ± 9.5
6T	46	2.48	54.6%	4.2 ± 0.1	26.6 ± 2.3	88.8 ± 16.3
8T	45	2.57	55.4%	4.0 ± 0.1	21.4 ± 1.4	78.3 ± 18.9

Opportunities. While local tuning falls short, modern cloud infrastructure presents two key opportunities to resolve GC-induced contention: *(1) Utilizing Idle Clusters Cycles.* In multi-tenant environments, GC bursts are rarely synchronized across all co-located runtimes. While a single runtime struggles with CPU deficits during its GC phase, neighboring runtimes may possess underutilized CPU cycles. This asymmetry creates an opportunity to aggregate these fragmented, idle resources into a shared pool. By offloading GC tasks to this isolated pool, we can achieve agile concurrent marking without contending with local application threads. *(2) The Bandwidth-Compute Gap.* The emergence of high-performance interconnects (e.g., RDMA, CXL) has fundamentally shifted the bottleneck. As shown in Table 2, concurrent marking is often bound by random memory access latency and synchronization overheads, yielding a modest scan speed of 2.6 GB/s in our tests. In contrast, modern 400 Gbps RDMA networks offer bandwidth orders of magnitude higher—theoretically sufficient to sustain traffic for 20 different marking tasks simultaneously. This vast bandwidth headroom makes it feasible to decouple GC to remote nodes, trading abundant network bandwidth for scarce local CPU resources.

Takeaway. Concurrent garbage collectors in multi-tenant environments compromise application stability primarily through CPU contention. Viewing the runtime holistically, this contention manifests as drastic, periodic fluctuations in CPU demand, making precise resource provisioning notoriously difficult. Consequently, the theoretically optimal solution requires elasticity: the runtime can access burstable CPU resources during GC phases (marking in particular) to reduce floating garbage, yet these resources should not be statically provisioned to avoid wastage during non-GC phases. This necessitates a consolidated approach: offloading GC tasks from multiple runtimes onto a shared, dedicated resource pool, where GC tasks are explicitly orchestrated to ensure efficient, interference-free execution.

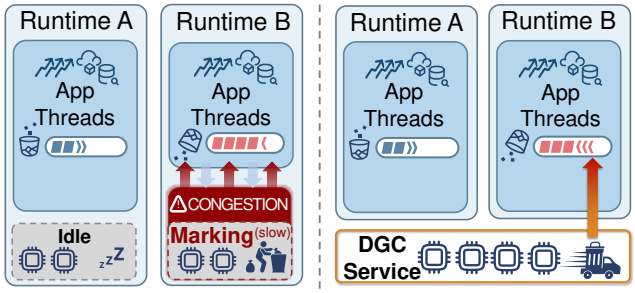


Figure 2: Overview of the baseline (left) and DGC (right).

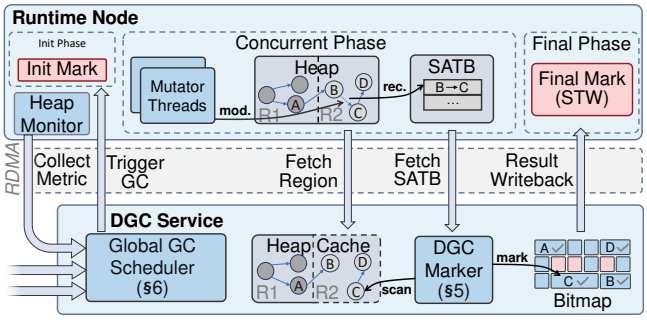


Figure 3: The workflow of DGC's disaggregated marking.

4 Disaggregated GC Service

4.1 Architecture Overview

This work presents Disaggregated Garbage Collection (DGC), a system designed to consolidate concurrent marking tasks from multiple runtimes onto a dedicated, isolated compute pool. By decoupling the marking workload from the application, DGC eliminates contention on CPU resources and accelerates marking throughput (Figure 2). The architecture comprises two core subsystems: a Disaggregated Marking Engine and a Global GC Orchestrator.

A strawman design for pooling would be adaptively increasing/decreasing the number of available cores for each language runtime, but this design faces two disadvantages. First, it contradicts the horizontal scaling mechanism used for fine-grained services and complicates resource management. Second, it cannot scale to a distributed environment. DGC instead employs a data-centric offloading strategy: its *disaggregated marking engine* accesses the runtime's heap through two distinct data planes for concurrent marking.

(1) Co-located Mode: For runtimes residing on the same physical host, the DGC service utilizes shared memory (SHM) as the data plane. By registering heap ranges and GC-related data structures into a memory segment visible only to DGC, the DGC service can directly access the runtime's state. This architecture enables the direct adaptation of standard concurrent marking algorithms, where intermediate metadata is exchanged via zero-copy SHM mechanisms and control signals are managed through simple RPCs. Notably, while currently implemented for single-host co-location, this SHM-based design conceptually extends to Hyper-Node systems providing shared memory semantics(e.g. CXL [30], UB [29]).

(2) Distributed Mode: For runtimes across physical boundaries, DGC employs RDMA-based Remote Marking, leveraging high-performance networking to traverse remote heaps efficiently (details in Section 5).

To serve multiple participating runtimes using a shared resource pool, the DGC system employs a *Global GC Orchestrator*. This component is composed of lightweight monitors embedded in each runtime and a centralized optimization solver. Using allocation-related profiles collected by monitors, the solver optimizes GC resource scheduling to ensure memory availability of all participating runtimes (details in Section 6).

4.2 Deployment Model and Assumptions

Operator-managed JVM stack. DGC is intended to be deployed as part of an operator-managed JVM stack. In this model, the JVM build, the DGC service, and the surrounding deployment infrastructure are owned by a single platform organization. Two industry practices fit this model. The first is the vendor-curated JVM distribution, exemplified by Alibaba Dragonwell [1], Amazon Corretto [3], and Azul Platform Prime [5]; these builds are produced and maintained by the same organization that supports their consumers.

The second is the internal Java platform operated by large enterprises, where a dedicated infrastructure organization maintains both the JVM runtime and the cluster management stack while application teams contribute only standard Java workloads. This practice is widely documented across major internet companies [8, 19, 23, 37, 48, 50, 60, 64]. In both practices, the JVM provider and the cluster operator share a single trust boundary, which allows DGC to read heap state across the operator's components and motivates a unified rather than per-tenant GC accelerator.

Knowledge required from the tenant application. DGC treats each served application as a black box and consumes only JVM-level metrics. The information includes heap free size, allocation rate, live-object size, and estimated marking duration. All of these metrics are already collected by Shenandoah's current monitoring infrastructure.

Hardware assumptions. The SHM data plane requires only that the served runtime and the DGC service share a host address space, or an address space that behaves like one (e.g., CXL [30] or UB [29] fabrics). The RDMA data plane requires that the DGC host and the served host both attach to an RDMA-capable fabric. Such fabrics are already standard in modern data centers; representative examples include AWS Elastic Fabric Adapter [47], and Microsoft Azure RDMA storage networks [6]. DGC therefore depends only on commodity data-center hardware.

5 RDMA-based Remote Marking

5.1 SATB-based Marking Algorithm

To address consistency maintenance in disaggregated marking, the remote marking engine adopts the Snapshot-at-the-Beginning (SATB) algorithm. SATB ensures correctness by effectively taking a logical snapshot of the heap at the start of the cycle. Any references overwritten during the concurrent marking phase are recorded by SATB write barriers to ensure that objects live at the start of the cycle remain reachable. Crucially, these barriers are active only during the marking phase, so DGC adds no steady-state overhead when GC is inactive. Prior work [31, 54] has shown that SATB barriers can ensure the correctness of marking even when GC threads are operating on a stale object graph.

Workflow. Figure 3 illustrates the workflow of the DGC's remote marking algorithm. The cycle begins with a lightweight Stop-The-World (STW) phase, during which the runtime records heap states and the GC root set to establish the logical SATB snapshot. Upon the resumption of the application, the DGC service receives an RPC to start data copying and concurrent marking. DGC adopts a configurable heap cache ratio on the GC node, which allows a tradeoff between marking performance and memory consumption. Throughout the marking phase, the accumulated SATB roots are periodically fetched from the runtime node and processed by the GC marker. Once the scanning converges, the liveness bitmap is returned to the runtime, which then enters a final, brief STW pause to process the small number of remaining SATB roots that were not yet synchronized, ensuring strictly consistent marking results.

Correctness. SATB correctness rests on a single invariant: every reference live in the snapshot is eventually visited by the marker, via heap traversal for unmodified edges or via the write-barrier buffer for overwritten ones. Since marking reclaims nothing, correctness depends only on the union of the two sources covering the snapshot, not on the completeness of either source alone. In the disaggregated setting, the DGC marker always observes a transient intermediate heap rather than a true snapshot. Under DGC-RDMA, it may further see an internally inconsistent view assembled from regions cached at different points in real time. The buffered SATB stream still supplies precisely the edges the marker's traversal might miss, so the marker converges to the same live-object set as an in-process collector.

5.2 Heap Cache Management

The heap cache manages a (partial) local copy of the served runtime's memory states, which can be directly accessed by GC threads of DGC. The heap cache is a copy system instead of a swap system, so it never affects the application's reads, writes, or execution. The cache originally supports two primitives: *retrieving* to fetch more memory states from the served runtime, and *eviction* to replace existing cached states with newly retrieved ones. A RDMA control thread is responsible for managing the heap cache, issuing RDMA read requests according to the primitives. A strawman design would be relying on existing OS-level remote memory paging systems [2, 45] to support those two primitives, but this design is intrusive to operating systems and remains unaware of the memory management behaviors inside language runtime systems, leading to frequent page faults and unacceptable overhead. To this end, DGC adopts a user-space software paging system and manages cached memory in a *region* granularity, which aligns with that in runtimes. The software paging approach also reduces the overhead of RDMA registration: during initialization, DGC pre-registers a fixed memory range as a local heap cache and employs a software address translation layer operating at the granularity of regions.

Region-based software paging. The *region* serves as the fundamental unit of memory management in mainstream collectors [12, 13, 63]. By aligning the cache management granularity with regions, DGC offers two advantages. First, it aligns seamlessly with the runtime's allocation patterns. Specifically, mainstream collectors like G1 and Shenandoah enable a per-region bump-pointer-style allocator, which induces strong temporal locality by grouping related objects to the same region. The locality is also analyzed by prior work, which indicates the number of inter-region references is usually limited [59]. Consequently, retrieving the whole region can improve the "hit rate" of local memory accesses compared with a page-based approach. Second, this approach is highly cache-efficient. For instance, in a 4GB SPECjbb heap with 2MB default region size, the software page table shrinks to merely 4KB (properly compressed based on alignment). This compact footprint allows the table to fit entirely within a modern CPU's L1d cache (48KB in our testbed), significantly reducing address translation overhead.

5.3 Handling Partial Object Graphs

Since the heap cache is empty at the beginning of marking, naively waiting for data copying would introduce significant overhead. Meanwhile, the size of heap cache may not accommodate the whole heap in the runtime node. To this end, DGC refines the original SATB-based GC algorithm to conduct marking when the cached memory states are not complete. The basic idea is to memorize pending references for uncached regions. During marking, DGC's GC threads identify those references by checking the status of regions and push them into a thread-local per-region pending queue. Once the region has been copied, DGC uses the RDMA control thread to aggregate pending queues for this region and dispatch them to a single GC thread, according to the current load. This design is aimed at preventing *bitmap contention*: concurrent scanning of the same region by multiple GC threads would trigger frequent cache coherence traffic on the shared marking bitmap.

Concurrency control. The switch point on the region state should be specially designed to ensure consistency between GC threads and the RDMA control thread. For example, if GC threads do not realize that a region is being evicted, they may introduce wrong marking results after the region has been replaced by RDMA read operations issued by the control thread. To this end, DGC introduces an 8-byte per-region entry for inter-thread synchronization: the high bits store the *state tags* (NotCached, OnTrans, Cached, and OnEvict) while the low bits store a counter serving as a reentrant read lock. Each *state tag* records the current status of DGC's local cache for the corresponding served-runtime region. When accessing an object, a GC thread first issues an atomic increment to the region entry to register its presence, and then validates whether the state tag is Cached. Similarly, it decrements the counter when finish processing the object. The RDMA control thread only changes the state tags when the corresponding counter is zero and leverages compare-and-swap operations to ensure the atomicity of its mutation operation. This concurrency control mechanism allows GC threads to safely initiate tasks with simple atomic instructions while ensuring that critical RDMA overwrites never interrupt an ongoing object scanning.

Running example. Figure 3 traces the following scenario step by step. Consider $A \in R_1$ (Cached) and $B, C, D \in R_2$ (NotCached), with references $A \rightarrow B \rightarrow C \rightarrow D$. The marker scans A, defers $A \rightarrow B$ into R_2's pending queue; the RDMA control thread later copies R_2, flips its tag to Cached, and dispatches the queue to a single marker, which then scans B. If a mutator *modifies* B and overwrites $B \rightarrow C$ before R_2 is copied, the marker would not see C from B alone in the heap cache; the SATB barrier *records* the original $B \rightarrow C$ at the moment of modification (upper part of Fig. 3), and the DGC service fetches this record over RDMA (middle part), so the marker still reaches C, follows $C \rightarrow D$, and sets all four objects in the bitmap (lower part).

5.4 Hotness-driven Paging Policies

To minimize RDMA traffic and avoid frequent retrieving on the same region, DGC uses *hotness scores* to determine which region should be retrieved and evicted. The score is calculated by the aggregate size of underprocessed objects (grey) within each region, which is grounded in our observation that larger objects typically encapsulate a higher density of outgoing references. Therefore, caching them exposes more work to GC threads, sustaining high throughput and preventing thread starvation. For regions not cached, DGC initially estimates the score using the number of pending references and the application's average object size, which can be profiled during each collection cycle. The score can be calibrated to the precise value once the region is retrieved. Furthermore, to prevent thrashing where regions are repeatedly swapped, DGC remains conservative on eviction and enforces a strict hysteresis threshold: A non-resident region is eligible for retrieval only if its score exceeds that of the eviction candidate by a configurable multiplicative factor.

Additionally, to mitigate thread starvation during the initialization phase, DGC introduces a *prefetch window*. This mechanism allows the RDMA controller to proactively populate the local buffer before marking threads start, strictly prioritizing those containing GC roots. Crucially, marking threads remain dormant during this interval. From the perspective of the global GC orchestrator, no CPU resources are allocated for the duration of the prefetch window, thereby preventing idle cycles and maximizing overall resource utilization.

5.5 Class Information Sharing

In addition to heap data, class information is also required during marking, as it stores the locations of all references inside an object. To minimize the cost of sharing this information, DGC embraces OpenJDK's application class data sharing mechanism (APPCDS [40]), which packs the in-memory Klass structures of all loaded classes into a single archive file, along with the reference-offset tables that GC uses to scan each object. When a served runtime first attaches, DGC preloads this archive at the same virtual address as the runtime, so that Klass pointers resolve identically on both sides without per-object translation.

Beyond the archive, Java runtimes can define new classes at runtime through reflection, custom class loaders, or lambda metafactories. Since every such class must be registered with the JVM's system dictionary before it becomes usable, DGC hooks this process: each newly defined Klass triggers a notification, and before the next GC cycle begins, the marker reads the new Klass data through SHM in co-located mode or one-sided RDMA in distributed mode. The marker therefore sees the full type system at the start of every collection.

6 Multi-Runtime GC Orchestration

The disaggregated marker of Section 4 solves only half of the problem. Concurrent GC is bursty: marking takes over the cycle for a short, busy interval, then stays idle until enough garbage accumulates to trigger the next cycle. A GC pool dedicated to a single served runtime would therefore sit idle most of the time, splitting the runtime's total CPU usage rather than reducing it. DGC recovers this efficiency by *time-division multiplexing* a single shared GC pool across many co-located runtimes. However, naive sharing would reintroduce the very contention that disaggregation was meant to eliminate: overlapping marking phases from different runtimes compete for GC threads and drag each other's tail latency back up.

To make sharing safe, DGC introduces a centralized *global GC orchestrator* that decides when each runtime's next GC cycle should start and with how many GC threads it should run. Three logical roles compose the orchestrator. A lightweight *monitor* embedded in each served runtime exposes its free-memory level, allocation rate, and the marking duration of historical cycles. A single *coordinator* reads the monitor state

across all runtimes and emits trigger decisions by solving the joint scheduling problem. A *trigger hook* on each served runtime's GC-trigger path applies the coordinator's decisions. The remainder of this section describes the design objectives that guide these decisions, the resulting scheduling problem, and how robustness is maintained when the system departs from steady state.

6.1 Design Objectives

Deferring marking as late as possible. As analyzed before, if the trigger point of SATB marking can be deferred, more objects can be included in the snapshot, resulting in less floating garbage and improved GC efficiency. Therefore, DGC's scheduler is designed to defer marking for all GC tasks without encountering any OOM issues.

Adjustable GC thread numbers. The number of GC threads is statically configured in state-of-the-art concurrent collectors to strike a balance between collection throughput and interference with mutators. In DGC's case, since the disaggregated GC tasks are running with dedicated computation resources, the static configuration approach is no longer viable. DGC thereby proposes to dynamically configure the number of GC threads for each task, according to its urgency and the system's resource availability. When a GC task starts late and requires finishing soon to avoid OOM issues, DGC can maximize its computation resources to improve the marking speed. However, since the SATB marking algorithm (graph traversal) does not scale well (see Table 2), the marginal benefit of adding more threads diminishes beyond a certain threshold. Therefore, when multiple GC tasks require serving simultaneously, the computation resources granted for each should be controlled to improve the overall CPU efficiency.

Avoiding Resource Contention. Since the memory footprint of marking is large, co-locating multiple GC threads into the same core leads to severe interference. To this end, the total number of active GC threads should not exceed the available core number at any moment.

6.2 Problem Formulation

According to the principles above, we formulate the scheduling algorithm as a constrained optimization problem below.

Decision Space: *(1) GC Trigger Time*: The time when the GC task should be triggered. *(2) DGC Thread Number*: The number of GC threads to be allocated for the task.

Constraints: *(1) Time Constraint*: Each GC should complete before the mutators exhaust all the available memory (i.e., OOM errors). *(2) Resource Constraint*: The total number of active GC threads should not exceed the number of available cores at any moment.

Algorithm Input: *(1) GC Completion Deadline*: The estimated GC completion deadline for each served runtime, calculated as the available free memory divided by its allocation rate. *(2) Estimated DGC Duration*: The estimated marking duration of each task under DGC, with different number of GC threads. *(3) Ongoing DGC Task Information*: The information of all ongoing GC tasks served by DGC, including the estimated completion time and the number of GC threads.

Optimization Goal: Defer the GC start time as long as possible. Since the start time can be represented as a monotone timestamp, the goal can be transformed to the sum of the GC start timestamps for all tasks.

Discussion: the estimated inputs. The input of algorithms relies on several estimated numbers, which are collected from runtime profiles in the served runtimes. The allocation rate has already been estimated by the runtime's statistical methods and thus can be directly reused by DGC. As for the estimated GC duration, it can be modeled as a linear function of the size of live objects. The function incorporates a thread-number-specific coefficient and a constant term, both of which vary across applications. Given that an application's live object set typically remains stable over short time periods, the number of live objects can be estimated by the historical marking results.

6.3 Efficient Solving Strategy

To address this optimization problem efficiently, we adopt a constraint programming (CP) approach that discretizes all parameters and integrates a dedicated CP-SAT solver. The resulting solving time is typically around 5 ms for single-instance scenarios, well within practical limits given the runtime's 10 ms allocation-rate monitoring tick.

For homogeneous workloads, scheduling decisions can be encoded as a small set of static priorities, because all served runtimes share comparable allocation rates and marking durations. Heterogeneous workloads break this assumption: GC deadlines, allocation rates, and marking durations all diverge across applications, and a static heuristic tuned for one combination typically fails on another. We therefore adopt CP-SAT as a general-purpose solver and leave the designing of simpler heuristics to future work.

6.4 Scheduling Robustness

The scheduler described so far operates in steady state. In practice, three non-steady states could arise: a freshly started runtime with no historical data, a sudden shift in allocation behavior, and an unreachable DGC service. We discuss each in turn.

Cold start. A freshly started runtime lacks the per-application information CP-SAT needs. DGC directly inherits Shenandoah's existing learning-phase convention: while the local heuristic itself is still warming up, the runtime triggers conservative local concurrent GC at default thresholds and DGC remains inactive. Once warmup completes, DGC takes over with conservative initial values for the linear duration coefficients in Section 6.2, refining them online by fitting against each subsequent completed DGC cycle.

Workload drift. Workload behavior may shift—a traffic surge, for instance, inflates the runtime's allocation rate and brings its GC deadline forward—and reusing a stale plan would risk OOM. The coordinator therefore re-solves the CP-SAT problem at every 10 ms scheduling tick against the latest memory state and allocation rate, possibly reordering pending GC tasks. The formulation also reserves deadline headroom to absorb moderate rises within a tick. When this headroom is exhausted, the runtime falls back to local concurrent Shenandoah for the affected cycle: CPU contention with mutators reappears, but worst-case performance stays bounded by baseline Shenandoah rather than risking OOM.

Fault Tolerance. When the DGC service becomes unreachable—whether crashed, slow, or partitioned—failures manifest in two ways depending on timing. If DGC dies while a marking task is in flight, the runtime simply sees the GC miss its deadline: memory keeps filling until OOM, the DGC call is cancelled, and the host falls through to Shenandoah's degenerated (STW) GC path, incurring a roughly 100 ms pause in our SPECjbb2015 experiments. Subsequent cycles no longer receive trigger signals from the coordinator and silently revert to local Shenandoah concurrent GC. Each trigger signal carries a version tag, so stale signals from a crashed coordinator are discarded rather than acted on. Recovery resembles a fresh startup: the new DGC service re-acquires each served runtime's memory layout and class information, after which the coordinator resumes issuing commands and the runtimes rejoin the DGC path.

7 Implementation

DGC is mainly implemented atop the Shenandoah garbage collector of OpenJDK 17 (version: jdk-17+35) in C/C++ and can support any unmodified Java applications. Our implementation has 10,383 LOC, including 946 lines specifically for the local version (referred to as *DGC-SHM*), 3,601 lines specifically for the remote version (referred to as *DGC-RDMA*), 959 lines for the GC scheduler, and the rest are for the common parts (e.g., class data sharing, GC workflow integration, RPC and RDMA support, etc.).

DGC's disaggregated GC service is implemented as a stripped-down version of OpenJDK's Java virtual machine (JVM). This enables DGC to reuse critical JVM components, including APPCDS and the class information resolution mechanism. The low-frequency control flow between the disaggregated marker instance and the served JVM is implemented with RPC, while the data flow is implemented with SHM or RDMA. To reach ideal marking performance, DGC reuses the well-optimized data structures in Shenandoah GC, including its work-stealing task queue, lock-free marking bitmaps, etc.

The GC scheduler is implemented with the CP-SAT solver [43] provided by Google OR-Tools [44]. The CP-SAT solver achieves high-performance solving by automatically reducing Constraint Programming (CP) problems to Boolean Satisfiability Problems (SAT). This capability enables its use for real-time GC scheduling in language runtime scenarios.

8 Evaluation

In this section, we evaluate DGC by answering five questions about its benefit and cost:

Q1. how does DGC cut tail latency and raise throughput across applications (§8.2)?

Q2. do the gains hold for heterogeneous runtimes sharing one marker pool (§8.3)?

Q3. what overhead does disaggregation incur, and how does DGC-RDMA perform under a tight budget (§8.4)?

Q4. does the benefit scale with the runtime count (§8.5)?

Q5. how much does each mechanism contribute (§8.6)?

The results show that, at an equal CPU budget, DGC raises critical-jOPS by 24% over Shenandoah and cuts tail latency by 64% on SPECjbb2015, with these gains holding under co-location and scaling at modest cost.

8.1 Experiment Setup

We ran experiments in a cluster of 2 servers with dual Intel Xeon Gold 6430 CPUs (32 physical cores each) and 128 GB of memory. Hyper-threading was disabled, and all servers were configured to performance mode to prevent CPU frequency fluctuation. Each server has a 200Gbps Nvidia BlueField-3 DPU working as an RDMA NIC, which is connected to the host via PCIe 4.0×16 connection.

Test Applications. We leverage the following applications to evaluate DGC:

• *SPECjbb2015* [11] serves as the de facto standard for assessing Java server performance, particularly in relation to garbage collection [13]. It simulates an online supermarket handling diverse requests, and the evaluated version is 1.0.4. SPECjbb2015 supports a multi-backend deployment mode, which natively enables distributed deployment with multiple JVMs. In our evaluation, we utilize this multi-backend mode to simulate a real-world scenario.

• *HBase* [4] is a distributed database designed for non-relational big data storage. The evaluated HBase version is 2.5.11. We use YCSB's two different mixed workloads (<50% read, 50% update> and <50% read, 50% insert>) [10] to evaluate the performance of DGC in HBase. HBase also supports a native distributed mode, which partitions the data into multiple regions and distributes them across multiple JVMs. In our evaluation, we utilize this multi-RegionServer mode to evaluate the performance of DGC.

• *DaCapo Bench* [7] provides a broad collection of Java applications. We evaluate DGC on the nine workloads in DaCapo 23.11-chopin that natively report per-request latency: `h2`, `tradesoap`, `tradebeans`, `lusearch`, `tomcat`, `spring`, `kafka`, `cassandra`, and `jme`, covering OLTP databases, online stock trading, text search, HTTP servers, messaging, and a 3D game engine. Following Jade [59]'s DaCapo throttle test

suite [58], we cap each workload's request rate at configurable target throughputs and measure per-request tail latency at each load level. We exclude DaCapo's remaining batch workloads. DGC's benefit comes from stabilizing the per-request CPU supply each served runtime receives over time, but this effect is invisible in end-to-end completion times—the only metric these batch benchmarks report.

Baseline GCs. Our evaluation mainly compares DGC (both DGC-SHM and DGC-RDMA) with Shenandoah and G1. Shenandoah is a concurrent collector integrated in OpenJDK 17. It serves as the codebase for DGC and thus has many implementation similarities. As shown in Table 1, Shenandoah also demonstrates the highest critical-jOPS performance in SPECjbb2015. Therefore, we selected Shenandoah as a representative state-of-the-art concurrent garbage collector to clearly illustrate the performance benefits of DGC.

G1 is currently the default garbage collector in JDK 17. It is a generational collector that is largely stop-the-world (STW): both young-generation collections and the compaction phase during full-heap GC (mixed GC) require STW pauses. Only the marking phase during mixed GC is designed to be concurrent. Although this design leads to frequent application pauses, it results in higher garbage collection efficiency. Due to its widespread adoption in various production scenarios, we used G1 as another baseline.

Memory configurations. For all applications, we first evaluate the minimum heap size that can run them with Shenandoah, then set the evaluation JVM heap size to be 2× of the minimum, to simulate a medium workload. DGC-SHM incurs negligible additional memory overhead and can be deployed directly. Conversely, DGC-RDMA relies on a local heap cache to store object graphs. In our primary evaluation, we assume sufficient remote memory availability and thus do not constrain DGC-RDMA's memory usage. We evaluate the impact of stricter memory configurations separately in Section 8.4, demonstrating that capping memory usage does not significantly compromise DGC-RDMA's performance gains.

CPU configurations. To ensure a fair comparison, baseline and DGC configurations consume the same total CPU budget. Baseline runs N JVMs, each allocated $8+c$ cores. DGC runs the same N JVMs each restricted to 8 application cores, plus one shared DGC service occupying $N \times c$ cores. Both deployments therefore consume $N \times (8+c)$ physical cores in total.

The value of c matches each workload's GC pressure under the 2×-minimum heap convention above. SPECjbb2015 and HBase have minimum heaps of several gigabytes; the 2× heap leaves ample allocation headroom, GC triggers are infrequent, and $c = 2$ cores per JVM are sufficient to absorb the marking demand. DaCapo benchmarks, by contrast, have minimum heaps of only tens to hundreds of megabytes; their 2× heap leaves only a thin allocation buffer, so the application reaches its GC trigger frequently and needs $c = 4$ cores per JVM to keep up. This accounting holds across all our experiments, so any throughput or latency improvement DGC demonstrates below stems from the change in GC mechanism rather than from extra CPU resources.

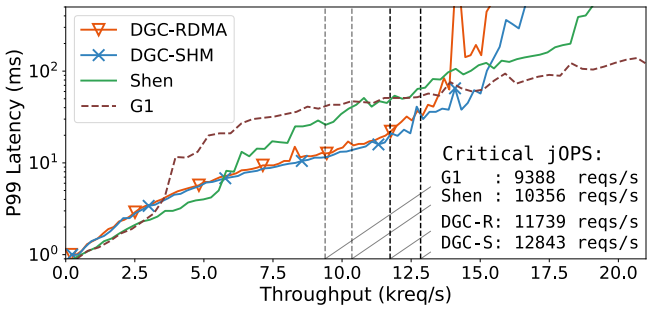


Figure 4: SPECjbb2015's P99 latency curve and Critical jOPS

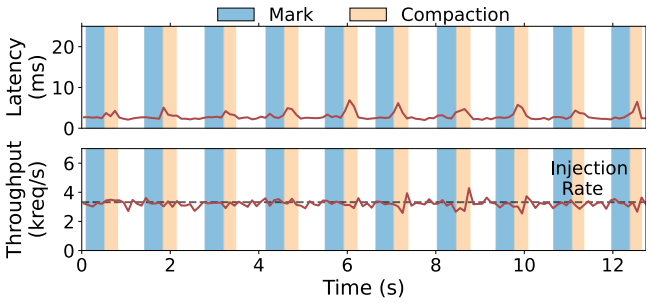


Figure 5: Latency details of SPECjbb2015 with DGC-SHM

8.2 Macro Benchmarks

We first demonstrate DGC's end-to-end performance improvements using macro-benchmarks. Since DGC enhances application performance by time-multiplexing CPU resources across instances to mitigate interference, we establish a minimal experimental setup: each application is deployed with two instances, paired with a DGC service.

SPECjbb2015. We first evaluate the performance of DGC on SPECjbb2015 with its default mode (HBIR_RT). which measures the latency under different issue rates, calculates the critical-jOPS (maximum achievable issue rate within an SLO limit) and reports a curve. The reported throughput and latency are the summary of the two backend clusters, which leads to a difference with previous results in Section 2.

As shown in Figure 4, DGC-SHM achieves the highest critical-jOPS, with 24.0% and 36.8% higher than Shenandoah and G1, respectively. The DGC-RDMA achieves the second highest critical-jOPS, 8.6% lower than DGC-SHM, but 13.4% higher than Shenandoah and 25.0% higher than G1.

From the P99 curve in Figure 4, it can be observed that G1 exhibits higher latency than other low-pause collectors due to its STW pauses. In contrast, both DGC-SHM and DGC-RDMA maintain lower P99 latency than Shenandoah when the issue rate is between 6k and 13k. In this range, the baseline Shenandoah suffers from high CPU contention

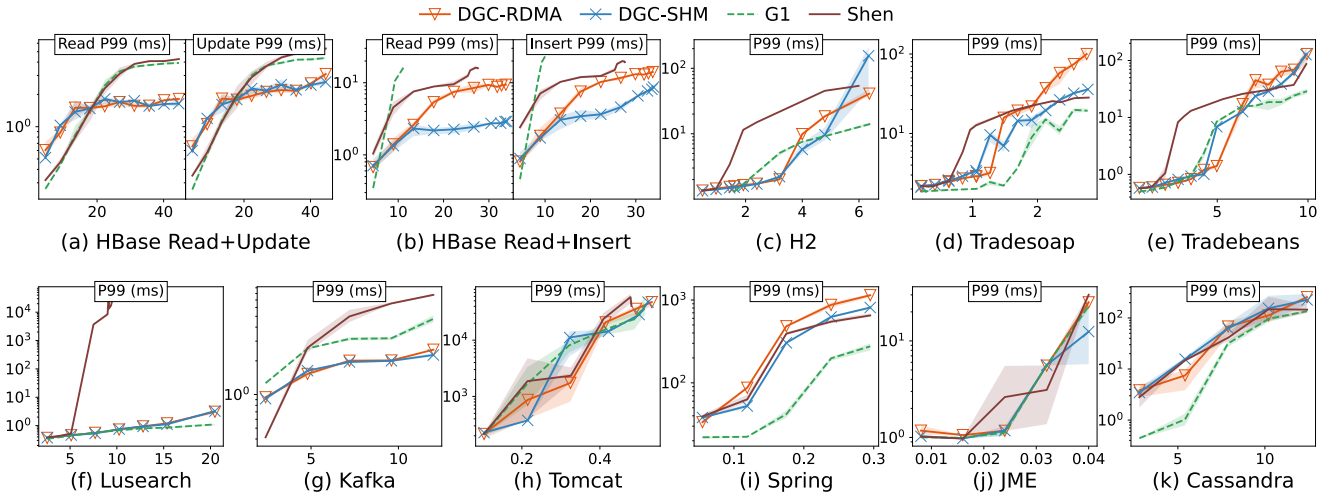


Figure 6: Throughput-latency curves on 2-host homogeneous deployments. (a)(b) HBase YCSB read-update / read-insert workloads, each pair sharing a y-axis. (c)–(k) DaCapo latency-reporting benchmarks. Bands show (min, max) across 3 reps.

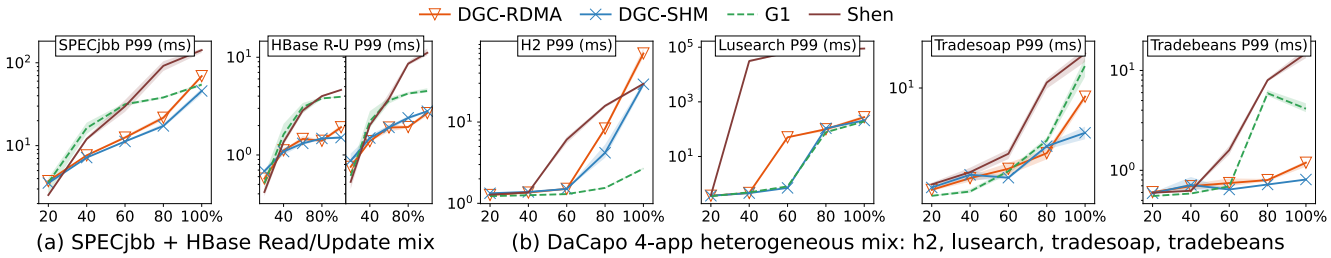


Figure 7: Mixed-workload P99 latency under different load levels; each mix shares one DGC service. (a) SPECjbb + HBase mix, with HBase Read/Update sharing a y-axis. (b) DaCapo 4-app heterogeneous mix. Bands show (min, max) across 3 reps.

between mutator and marker threads, whereas DGC mitigates this by utilizing dedicated physical cores for markers. Consequently, at Shenandoah's critical jOPS (10356 req/s), DGC-SHM and DGC-RDMA reduce P99 latency by 64.4% and 60.3%, respectively.

Additionally, it can be seen on the right side of the figure that DGC-SHM and DGC-RDMA trend towards P99 divergence earlier than the baseline GC. This behavior is attributed to the fact that, in our experimental setup, the DGC runtime operates with two fewer cores compared to the baseline runtime. As a result, DGC's performance degrades faster than the baseline under extremely high loads. This is the trade-off accepted by DGC to optimize application performance under moderate loads.

To identify the source of this performance improvement, we examined the phase-wise latency behavior of DGC under the same configuration used in Figure 1. As observed in Figure 5, the performance fluctuations during the marking phase are significantly smoothed out with DGC. Residual latency fluctuations appear only during the compaction phase, which are attributed to Shenandoah's load barriers.

HBase. We use HBase to further evaluate the performance of DGC in database scenarios. In this experiment, YCSB's two mixed workloads (<50% read, 50% update> and <50% read, 50% insert>) [10] are running on a two-RegionServer configuration, while the RegionServers are equipped with different collectors to evaluate the improvement of DGC.

As shown in Figure 6 (a) and (b), DGC-SHM and DGC-RDMA both achieve lower P99 latency than baselines under most of the issue rates. For the <read, update> workload, DGC-SHM's P99 read and update latencies are 58.3% and 40.3% lower than G1 under peak throughput, while DGC-RDMA's are 53.8% and 29.1% lower than G1. (G1 is the better one of the two baselines.) For the <read, insert> workload, DGC-SHM's P99 read and insert latency are 81.8% and 55.7% lower than Shenandoah under peak throughput, while DGC-RDMA's P99 read and insert latency are 41.4% and 28.5% lower than Shenandoah. G1 fails to achieve a similar peak throughput as Shenandoah or DGC, mainly due to frequent STW full GC (a fallback mechanism triggers when the concurrent marking in mixed GC fails to meet the allocation rate). In contrast, although Shenandoah induces shorter and fewer stop-the-world pauses, its concurrent GC phases cause a consistently slower

request processing speed. This affects a much larger number of requests and leads to worse P99 latency compared to G1. DGC not only maintains the low-pause advantage of Shenandoah but also resolves the CPU contention issue during concurrent GC. As a result, it demonstrates significantly improved tail latency.

DaCapo. Panels (c)–(k) of Figure 6 report P99 latency versus achieved request rate for the nine latency-reporting DaCapo benchmarks, each run as two homogeneous instances of the same workload. Five benchmarks (`h2`, `tradesoap`, `tradebeans`, `lusearch`, `kafka`) show a clear DGC-versus-Shenandoah separation: at the highest request rate each benchmark sustains under Shenandoah, DGC-SHM reduces P99 by 88.6%, 73.1%, 92.8%, four orders of magnitude, and 71.0% respectively. The remaining four benchmarks (`tomcat`, `spring`, `jme`, `cassandra`) show no meaningful separation between DGC and Shenandoah at any rate we tested. G1, by contrast, enjoys a sizable advantage on `tradesoap`, `spring` (i) and `cassandra` (k), where its sub-millisecond young-generation pauses dominate the tail; on the other seven benchmarks G1 is comparable to Shenandoah and DGC.

On these five benchmarks Shenandoah's concurrent marker runs on cores it shares with mutators, and the resulting interference shows up as P99 pressure on the application. The severity varies across the group. On `h2`, `tradesoap`, and `tradebeans` the application drives up to 1.8 GB/s of allocation, the marker stays active for 10 to 27% of wall-clock under high load, and Shenandoah throttles mutators with application-side pauses to keep heap occupancy bounded. This shows up in panels (c)–(e) as a P99 cliff: under Shenandoah, the tail transitions from a flat baseline to a sharp climb once IR crosses a benchmark-specific threshold. At the same IR, DGC-SHM keeps P99 2.3× to 2.7× lower than Shenandoah, and pushes the cliff IR itself out by 1.4× to 2.4×. On `lusearch` above ~5k req/s Shenandoah cannot keep up and falls back to several hundred stop-the-world Degenerated GC cycles per run, producing the multi-second P99. Shenandoah's achieved throughput saturates around 9.5k req/s under this fallback, whereas DGC-SHM scales smoothly up to 20k req/s while keeping P99 within the low-millisecond range. On `kafka` at 12k req/s Shenandoah's marker stays idle for over 98% of wall-clock, but `kafka`'s P99 is sub-millisecond, so even brief marker activity is visible in the tail. As result, DGC-SHM reduces P99 by 71.0% versus Shenandoah and 52.4% versus G1.

The other four benchmarks share a single property the first five lack: GC pressure stays low at every rate we tested. These benchmarks either keep most of their working set off-heap (`cassandra`), run with on-heap working sets of only tens of megabytes (`spring`, `tomcat`), or barely trigger GC at the achievable rates (`jme`). With so little allocation pressure, P99 is dominated by application-logic latency rather than by GC behavior, leaving DGC no contention to remove and Shenandoah none to suffer. G1's young-generation collector benefits from the same property: on a tens-of-megabyte young space its STW pauses are sub-millisecond and invisible in the tail. Shenandoah and DGC instead pay the constant cost of SATB write barriers and concurrent bookkeeping, a fixed tax that is unfavorable when there is little allocation to amortize against, which is why G1 takes the lead on `spring` and `cassandra`.

8.3 Mixed Workloads

The macro benchmarks above each evaluate a single application. We now stress DGC with two heterogeneous co-location scenarios in which multiple JVMs share a single DGC service, directly evaluate the multi-runtime orchestration described in Section 6. Figure 7 sweeps the offered load of every co-located runtime synchronously from 20% to 100% of a per-application target IR calibrated for this mix.

SPECjbb + HBase mix. Panel (a) of Figure 7 co-locates two SPECjbb backends with two HBase RegionServers (YCSB workload A) on a single host, reflecting a common enterprise pattern in which an online service shares hardware with a database. At 100% load, DGC-SHM reduces SPECjbb P99 from 142 ms (Shenandoah) to 46 ms (a 67.6% reduction), YCSB read P99 from 4.64 ms to 1.51 ms (-67.5%), and YCSB update P99 from 11.1 ms to 2.80 ms (-74.8%). DGC-RDMA achieves comparable reductions of 51.5%/58.5%/75.9% on the same three metrics. G1 is the second baseline, it stays 1.4–3× above DGC at the upper half of the IR range.

DaCapo 4-app mix. Panel (b) co-locates the four latency-reporting DaCapo benchmarks (`h2`, `lusearch`, `tradesoap`, `tradebeans`) on a single host. The mix amplifies the DGC-versus-Shenandoah gap relative to the homogeneous case in Figure 6. `lusearch` under Shenandoah collapses to a 64-second P99 already at 60% mix-load, whereas DGC-SHM holds the same workload at 708 μs; `tradebeans` and `tradesoap` at 100% load achieve 94.6% and 66.1% P99 reductions over Shenandoah. DGC-SHM also outperforms G1 by 80.5% on `tradebeans` and 60.0% on `tradesoap` at the same load point, but offers no significant advantage over G1 at other load levels, primarily because G1's generational design achieves higher collection efficiency on these smaller DaCapo workloads.

8.4 RDMA and Memory Efficiency

Figure 8 illustrates the SPECjbb2015 tail latency and corresponding RDMA traffic of DGC-RDMA with varying local heap cache sizes. To eliminate the noise caused by drastic load fluctuations in the HBIR mode, we configured SPECjbb2015 to the fixed-throughput (Preset) mode. The Injection Rate (IR) was fixed at 10,356, corresponding to the best Critical jOPS of the baseline Shenandoah collector (as shown in Figure 4), to explicitly demonstrate the performance uplift enabled by DGC. Additionally, the coordinator enforces strict time-multiplexing of the four GC cores between two Runtimes

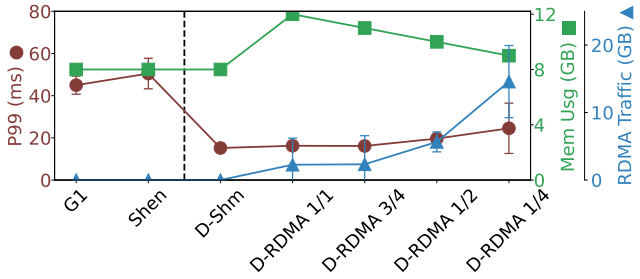


Figure 8: DGC-RDMA's P99 latency, memory usage, and RDMA traffic under different local cache settings.

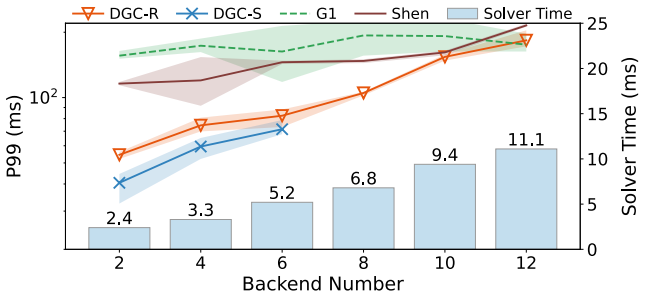


Figure 9: Scalability test results on SPECjbb2015 PRESET.

to prevent simultaneous DGC triggers and avoid doubling peak heap cache usage. Among the four evaluated configurations, even the most constrained setting (1/4 cache) achieves a 51.5% improvement in P99 latency over Shenandoah, since its RDMA traffic is still within the bandwidth limit. Notably, the 1/2 cache configuration strikes an optimal balance between memory overhead and RDMA traffic cost. Compared to the baseline of two 4GB heaps, it requires only 2GB of additional remote memory (a 25% overhead) to serve as the DGC node's local cache. Furthermore, it incurs an average RDMA traffic of only 5.52GB per GC cycle, representing a moderate 37% amplification relative to the 4GB full heap size.

8.5 Scalability Test

SPECjbb2015 homogeneous workload. To evaluate the performance of DGC as the number of runtimes increases, we conducted a series of experiments running SPECjbb2015 with varying numbers of backends in a distributed setup. The experiment adhered to the core allocation policy established earlier. To simplify the testing process, we used SPECjbb2015's PRESET mode, in which the benchmark runs at a fixed throughput and reports P99 latency. The average load per backend was set to 6819 req/s, as measured in Table 1, and the total system load was scaled proportionally with the number of backends.

The results are shown in Figure 9. DGC consistently demonstrated better P99 latency than Shenandoah across all scales, with the improvement of up to 43.3% and 50.9% at 6 backends. Since DGC-SHM requires the served JVM and the disaggregated marker to reside on the same node, it was limited to 6 backends (7 backends would require at least 70 cores per node, exceeding the 64 cores limit in our physical machines). DGC-RDMA mode was tested up to 12 backends, but the performance improvement began to diminish. Analysis of the GC logs indicates that the RDMA NIC bandwidth became a bottleneck at this point.

During the test, we also recorded the time taken by the CP-SAT solving process. The solving time remained efficient, taking only 11.1 ms even in the 12-backend scenario, which is sufficient for scheduling the GC tasks that last for hundreds of milliseconds.

Table 3: Performance breakdown on SPECjbb2015 PRESET

Garbage Collector	P99 (ms)	avg. Mark Time (ms)
DGC	81.5 ± 1.1	416.1 ± 2.8
DGC-noOPT	87.0 ± 10.8	406.9 ± 2.0
DGC-noCOOR	13800 ± 7132	641.2 ± 28.3
Shenandoah	121.3 ± 3.6	284.2 ± 1.6

8.6 Ablation Study

We evaluated the individual contributions of our two primary optimizations using SPECjbb2015's PRESET mode. Given the limited configurability of DGC-SHM, this ablation study focuses on DGC-RDMA, with results detailed in Table 3.

Impact of Pipelined Co-design. The "noOPT" configuration disables the pipelined marker-transfer co-design described in Section 5.2. Instead of concurrent execution, the marker is forced to wait for the entire heap to be copied before starting. This serialization raises P99 latency by only 6.7% over the full configuration (87 ms vs 81.5 ms). This gap confirms that our design effectively masks the majority of data transfer costs.

Impact of Global Scheduling. The "noCOOR" configuration removes the global GC scheduler, relying instead on local JVM heuristics. Without cross-JVM coordination, the per-cycle marking time exhibits a much wider spread. The local heuristic depends on a stable marking-time estimate to predict the next trigger point, so it can no longer place triggers accurately. As triggers diverge, free space is repeatedly exhausted before marking completes, and the system periodically falls back into degenerated GC. The served JVMs can no longer sustain the target injection rate, and end-to-end latency itself diverges (Table 3).

9 Discussion

Generalizability to other collectors. DGC's core mechanism—offloading concurrent marking under an SATB snapshot—is not specific to Shenandoah. It integrates directly with any SATB-style collector, including G1, Generational Shenandoah, LXR [67], and Jade [59], with only data-plane-level changes to the bitmap and SATB-buffer layouts; the correctness argument in Section 5.1 is collector-agnostic. Colored-pointer collectors such as ZGC can be supported by adding a lightweight SATB-style buffer purely to feed the remote marker, while the colored-

pointer machinery used for evacuation and remapping remains on the local collector.

Trust and cross-tenant isolation. DGC's orchestrator, marker, and served runtimes sit inside a single operator-managed trust boundary (Section 4.2); cross-tenant isolation between co-located runtimes is enforced by the same cgroup, NUMA, and namespace mechanisms that already isolate co-located JVMs. Within the DGC instance, the orchestrator serializes marking tasks across tenants on the shared marker pool (Section 6), so one tenant's marking cannot co-run with—and steal CPU from—another's.

Future work. DGC does not currently enforce memory-bandwidth or PCIe-level isolation between concurrent marking tasks; under sustained fan-in to a single NIC, RDMA-mode clients can mildly interfere through shared PCIe bandwidth, which finer-grained mechanisms such as MBA or per-queue-pair rate limiting would tighten. A second extension concerns the scheduler. DGC adopts CP-SAT to remain general across heterogeneous workloads (Section 6.2); identifying when a solver-free heuristic suffices would remove the external optimization dependency and shorten the scheduling tick.

10 Related Work

10.1 Concurrent garbage collection

Concurrent garbage collection (GC) is a critical component in modern Java Virtual Machines (JVMs) to ensure efficient memory management without significant pause times. Several state-of-the-art concurrent marking algorithms have been developed, including those used in the JDK's G1, Shenandoah, and ZGC collectors.

Other research works also focus on optimizing the concurrent marking GC algorithm. C4 [49] builds on Pauseless GC by incorporating a generational approach, while Collie [21] suggests leveraging hardware transactional memory (HTM) for atomic object relocation. Compressor [25] employs hand-over-hand compaction to maintain minimal physical memory usage. Block-free GC [42] introduces non-blocking handshakes for concurrent stack scanning and object copying. Jade [59] provides a group-wise collection mechanism to improve the performance of low-pause GC. Some other concurrent collectors, like LXR [67], use reference counting to discover dead objects but still need to conduct concurrent marking periodically.

10.2 Externalized Garbage Collection

Due to the increasing demand for high-performance computing and data processing, the use of specialized hardware to accelerate applications has become increasingly popular [22, 24, 26, 38, 52, 53, 57, 61, 62, 65, 69].

Previous research has also explored the idea of externalizing GC processes with specialized hardware. Martin Maas et al. [35] proposed a hardware accelerator that is located close to the memory controller, which can significantly improve the GC performance. Andrés Amaya et al. [14] further extended the idea in real-time scenarios. Thomas et al. [51] proposed to optimize the garbage collection with recently developed near-memory processing (NMP) hardware. DGC differs in two ways: it offloads GC onto commodity server CPUs over RDMA or shared memory, and it shares those GC resources across multiple runtimes rather than reserving them exclusively.

Maas et al. [33, 34, 36] also coordinate GC across multiple JVMs, but target *cooperating* workers of a single distributed application (e.g., Spark, Cassandra) by exploiting application-level events to decide when each node collects. DGC instead targets *unrelated, black-box* JVMs co-located by an operator, staggers their GC windows purely from JVM-level metrics, and additionally offloads the marking computation itself into a shared compute pool.

10.3 Disaggregation and Resource Pooling

Resource pooling, particularly memory pooling, has been widely studied [2, 9, 18, 20, 27, 28, 32, 45, 46, 56, 68].

Some works also focus on building a garbage collector for memory-disaggregated systems. Semeru [54] and Mako [31] are memory-disaggregated managed runtimes with a distributed garbage collector that is located on the memory node to minimize the memory swapping overhead. MemLiner [55] further reduces the memory swapping overhead by locating the garbage collector on the CPU node and prioritizing the marking task of objects in the CPU Node's local memory. These efforts primarily focus on leveraging disaggregated memory to address memory utilization issues. Their goals are orthogonal to those of DGC, which is chiefly concerned with mitigating CPU load fluctuations during garbage collection and improving CPU utilization through decoupled GC execution.

11 Conclusion

This work presents DGC, a disaggregated architecture to serve GC tasks of language runtimes in an external instance. DGC supports externalizing the costly marking task in GC to idle CPU cores locally or remotely and allowing multiple runtimes to share the GC service with minimal conflicts. The evaluation results on representative latency-sensitive applications show that DGC can significantly improve the application latency and goodput.

Acknowledgments

We thank the anonymous OSDI reviewers and our shepherd for their insightful feedback. We also thank Haibo Chen for refining the presentation of this paper. This research was supported in part by National Key Research & Development Program of China (No. 2023YFB3308501), National Natural Science Foundation of China (No. 62572306, 62132014), and the CCF-Huawei Populus Grove Fund.

References

[1] Alibaba. 2024. Alibaba Dragonwell: A Friendly Downstream Distribution of OpenJDK. GitHub repository. https://github.com/dragonwell-project/dragonwell17

[2] Emmanuel Amaro, Christopher Branner-Augmon, Zhihong Luo, Amy Ousterhout, Marcos K Aguilera, Aurojit Panda, Sylvia Ratnasamy, and Scott Shenker. 2020. Can far memory improve job throughput?. In *Proceedings of the Fifteenth European Conference on Computer Systems*. 1–16.

[3] Amazon Web Services. 2024. Amazon Corretto: No-Cost, Multiplatform, Production-Ready Distribution of OpenJDK. AWS product page. https://aws.amazon.com/corretto/

[4] Apache. 2022. Apache HBase. https://hbase.apache.org/.

[5] Azul Systems. 2024. Azul Prime: an award-winning enhanced build of OpenJDK for superior application performance, responsiveness and efficiency. Azul product page. https://www.azul.com/products/prime/

[6] Wei Bai, Shanim Sainul Abdeen, Ankit Agrawal, Krishan Kumar Attre, Paramvir Bahl, Ameya Bhagat, Gowri Bhaskara, Tanya Brokhman, Lei Cao, Ahmad Cheema, et al. 2023. Empowering azure storage with {RDMA}. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*. 49–67.

[7] Stephen M Blackburn, Zixian Cai, Rui Chen, Xi Yang, John Zhang, and John Zigman. 2025. Rethinking Java Performance Analysis. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1, ASPLOS 2025, Rotterdam, Netherlands, 30 March 2025 - 3 April 2025*. ACM. doi:10.1145/3669940.3707217

[8] ByteDance. 2024. CompoundVM: An Optimized JDK with High Compatibility and Performance. GitHub repository. https://github.com/bytedance/CompoundVM

[9] Lei Chen, Shi Liu, Chenxi Wang, Haoran Ma, Yifan Qiao, Zhe Wang, Chenggang Wu, Youyou Lu, Xiaobing Feng, Huimin Cui, et al. 2024. A Tale of Two Paths: Toward a Hybrid Data Plane for Efficient {Far-Memory} Applications. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*. 77–95.

[10] Brian F Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking cloud serving systems with YCSB. In *Proceedings of the 1st ACM symposium on Cloud computing*. 143–154.

[11] Standard Performance Evaluation Corporation. 2021. The specjbb2015 benchmark. https://www.spec.org/jbb2015/.

[12] David Detlefs, Christine H. Flood, Steve Heller, and Tony Printezis. 2004. Garbage-first garbage collection. In *Proceedings of the 4th International Symposium on Memory Management, ISMM 2004, Vancouver, BC, Canada, October 24-25, 2004*. ACM, 37–48. doi:10.1145/1029873.1029879

[13] Christine H Flood, Roman Kennke, Andrew Dinn, Andrew Haley, and Roland Westrelin. 2016. Shenandoah: An open-source concurrent compacting garbage collector for openjdk. In *Proceedings of the 13th International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools*. 1–9.

[14] Andrés Amaya García, David May, and Ed Nutting. 2021. Integrated hardware garbage collection. *ACM Transactions on Embedded Computing Systems (TECS)* 20, 5 (2021), 1–25.

[15] Google. 2025. Android Open Source Project. https://source.android.com/docs/core.

[16] Google. 2025. The Go Programming Language. https://go.dev.

[17] Google. 2025. V8 JavaScript engine. https://v8.dev.

[18] Juncheng Gu, Youngmoon Lee, Yiwen Zhang, Mosharaf Chowdhury, and Kang G Shin. 2017. Efficient memory disaggregation with infiniswap. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*. 649–667.

[19] Jing Guo, Zihao Chang, Sa Wang, Haiyang Ding, Yihui Feng, Liang Mao, and Yungang Bao. 2019. Who limits the resource efficiency of my datacenter: An analysis of alibaba datacenter traces. In *Proceedings of the international symposium on quality of service*. 1–10.

[20] Zhiyuan Guo, Zijian He, and Yiying Zhang. 2023. Mira: A program-behavior-guided far memory system. In *Proceedings of the 29th Symposium on Operating Systems Principles*. 692–708.

[21] Balaji Iyengar, Gil Tene, Michael Wolf, and Edward Gehringer. 2012. The collie: a wait-free compacting collector. In *Proceedings of the 2012 international symposium on Memory Management*. 85–96.

[22] Houxiang Ji, Mark Mansi, Yan Sun, Yifan Yuan, Jinghan Huang, Reese Kuper, Michael M Swift, and Nam Sung Kim. 2023. {STYX}: Exploiting {SmartNIC} capability to reduce datacenter memory tax. In *2023 USENIX Annual Technical Conference (USENIX ATC 23)*. 619–633.

[23] Jesse Jie. 2022. LinkedIn's Journey to Java 11. LinkedIn Engineering Blog. https://www.linkedin.com/blog/engineering/infrastructure/linkedin-s-journey-to-java-11

[24] Xin Jin, Xiaozhou Li, Haoyu Zhang, Robert Soulé, Jeongkeun Lee, Nate Foster, Changhoon Kim, and Ion Stoica. 2017. Netcache: Balancing key-value stores with fast in-network caching. In *Proceedings of the 26th symposium on operating systems principles*. 121–136.

[25] Haim Kermany and Erez Petrank. 2006. The Compressor: concurrent, incremental, and parallel compaction. In *Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 354–363.

[26] Jongyul Kim, Insu Jang, Waleed Reda, Jaeseong Im, Marco Canini, Dejan Kostić, Youngjin Kwon, Simon Peter, and Emmett Witchel. 2021. Linefs: Efficient smartnic offload of a distributed file system with pipeline parallelism. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*. 756–771.

[27] Andres Lagar-Cavilla, Junwhan Ahn, Suleiman Souhlal, Neha Agarwal, Radoslaw Burny, Shakeel Butt, Jichuan Chang, Ashwin Chaugule, Nan Deng, Junaid Shahid, et al. 2019. Software-defined far memory in warehouse-scale computers. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*. 317–330.

[28] Quanxi Li, Hong Huang, Ying Liu, Yanwen Xia, Jie Zhang, Mosong Zhou, Xiaobing Feng, Huimin Cui, Quan Chen, Yizhou Shan, et al. 2025. Beehive: A Scalable Disaggregated Memory Runtime Exploiting Asynchrony of Multithreaded Programs. In *22nd USENIX Symposium on Networked Systems Design and Implementation (NSDI 25)*. 167–187.

[29] Heng Liao, Bingyang Liu, Xianping Chen, Zhigang Guo, Chuanning Cheng, Jianbing Wang, Xiangyu Chen, Peng Dong, Rui Meng, Wenjie Liu, et al. 2025. UB-Mesh: a Hierarchically Localized nD-FullMesh Datacenter Network Architecture. *arXiv preprint arXiv:2503.20377* (2025).

[30] Compute Express Link. 2025. Compute Express Link (CXL) Specification 4.0. https://computeexpresslink.org/wp-content/uploads/2025/11/CXL_4.0-Specification-Release_FINAL_Website-Copy.pdf.

[31] Haoran Ma, Shi Liu, Chenxi Wang, Yifan Qiao, Michael D Bond, Stephen M Blackburn, Miryung Kim, and Guoqing Harry Xu. 2022. Mako: A low-pause, high-throughput evacuating collector for memory-disaggregated datacenters. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. 92–107.

[32] Haoran Ma, Yifan Qiao, Shi Liu, Shan Yu, Yuanjiang Ni, Qingda Lu, Jiesheng Wu, Yiying Zhang, Miryung Kim, and Harry Xu. 2024. {DRust}:{Language-Guided} Distributed Shared Memory with Fine Granularity, Full Transparency, and Ultra Efficiency. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*. 97–115.

[33] Martin Maas. 2018. *Hardware and Software Support for Managed-Language Workloads in Data Centers*. Ph. D. Dissertation. University of California, Berkeley. https://www2.eecs.berkeley.edu/Pubs/TechRpts/2018/EECS-2018-152.html UC Berkeley EECS Tech Report UCB/EECS-2018-152; covers both cluster-wide GC coordination (Taurus) and hardware GC offload.

[34] Martin Maas, Krste Asanović, Tim Harris, and John Kubiatowicz. 2016. Taurus: A holistic language runtime system for coordinating distributed managed-language applications. *Acm SIGPLAN Notices* 51, 4 (2016), 457–471.

[35] Martin Maas, Krste Asanović, and John Kubiatowicz. 2018. A hardware accelerator for tracing garbage collection. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 138–151.

[36] Martin Maas, Tim Harris, Krste Asanović, and John Kubiatowicz. 2015. Trash day: Coordinating garbage collection in distributed systems. In *15th Workshop on Hot Topics in Operating Systems (HotOS {XV})*.

[37] Meituan Technical Team. 2020. Exploration and Practice of the New-Generation Garbage Collector ZGC at Meituan. Meituan Tech Blog. https://tech.meituan.com/2020/08/06/new-zgc-practice-in-meituan.html

[38] Young Gyoun Moon, Ilwoo Park, Seungeon Lee, and Kyoung Soo Park. 2018. Accelerating flow processing middleboxes with programmable NICs. In *Proceedings of the 9th Asia-Pacific Workshop on Systems*. 1–3.

[39] New Relic. 2024. *2024 State of the Java Ecosystem*. Technical Report. New Relic. https://newrelic.com/sites/default/files/2024-04/new-relic-state-of-the-java-ecosystem-report-2024-04-30.pdf Accessed: 2024-05-20.

[40] OpenJDK. 2018. JEP 310: Application Class-Data Sharing. https://openjdk.java.net/jeps/310

[41] OpenJDK. 2025. OpenJDK HotSpot. https://openjdk.org/groups/hotspot/.

[42] Erik Österlund and Welf Löwe. 2016. Block-free concurrent GC: stack scanning and copying. In *Proceedings of the 2016 ACM SIGPLAN International Symposium on Memory Management*. 1–12.

[43] Laurent Perron and Frédéric Didier. 2024. *CP-SAT*. Google. https://developers.google.com/optimization/cp/cp_solver/

[44] Laurent Perron and Vincent Furnon. 2024. *OR-Tools*. Google. https://developers.google.com/optimization/

[45] Yifan Qiao, Chenxi Wang, Zhenyuan Ruan, Adam Belay, Qingda Lu, Yiying Zhang, Miryung Kim, and Guoqing Harry Xu. 2023. Hermit:{Low-Latency},{High-Throughput}, and Transparent Remote Memory via {Feedback-Directed} Asynchrony. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*. 181–198.

[46] Zhenyuan Ruan, Malte Schwarzkopf, Marcos K Aguilera, and Adam Belay. 2020. {AIFM}:{High-Performance},{Application-Integrated} far memory. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. 315–332.

[47] Leah Shalev, Hani Ayoub, Nafea Bshara, and Erez Sabbag. 2020. A cloud-optimized transport protocol for elastic and scalable hpc. *IEEE micro* 40, 6 (2020), 67–73.

[48] Tencent. 2024. TencentKona-17: Tencent Kona JDK17 – A No-Cost, Production-Ready Distribution of OpenJDK. GitHub repository. https://github.com/Tencent/TencentKona-17

[49] Gil Tene, Balaji Iyengar, and Michael Wolf. 2011. C4: The continuously concurrent compacting collector. In *Proceedings of the international symposium on Memory management*. 79–88.

[50] Danny Thomas and Netflix Technology Blog. 2024. Bending Pause Times to Your Will with Generational ZGC. Netflix Tech Blog. https://netflixtechblog.com/bending-pause-times-to-your-will-with-generational-zgc-256629c9386b

[51] Samuel Thomas, Jiwon Choe, Ofir Gordon, Erez Petrank, Tali Moreshet, Maurice Herlihy, and R Iris Bahar. 2022. Towards Hardware Accelerated Garbage Collection with Near-Memory Processing. In *2022 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE, 1–6.

[52] Maroun Tork, Lina Maudlej, and Mark Silberstein. 2020. Lynx: A smartnic-driven accelerator-centric architecture for network servers. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*. 117–131.

[53] Lluís Vilanova, Lina Maudlej, Shai Bergman, Till Miemietz, Matthias Hille, Nils Asmussen, Michael Roitzsch, Hermann Härtig, and Mark Silberstein. 2022. Slashing the disaggregation tax in heterogeneous data centers with fractos. In *Proceedings of the seventeenth european conference on computer systems*. 352–367.

[54] Chenxi Wang, Haoran Ma, Shi Liu, Yuanqi Li, Zhenyuan Ruan, Khanh Nguyen, Michael D Bond, Ravi Netravali, Miryung Kim, and Guoqing Harry Xu. 2020. Semeru: A {Memory-Disaggregated} managed runtime. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. 261–280.

[55] Chenxi Wang, Haoran Ma, Shi Liu, Yifan Qiao, Jonathan Eyolfson, Christian Navasca, Shan Lu, and Guoqing Harry Xu. 2022. {MemLiner}: Lining up Tracing and Application for a {Far-Memory-Friendly} Runtime. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*. 35–53.

[56] Chenxi Wang, Yifan Qiao, Haoran Ma, Shi Liu, Wenguang Chen, Ravi Netravali, Miryung Kim, and Guoqing Harry Xu. 2023. Canvas: Isolated and adaptive swapping for {Multi-Applications} on remote memory. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*. 161–179.

[57] Xingda Wei, Rongxin Cheng, Yuhan Yang, Rong Chen, and Haibo Chen. 2023. Characterizing off-path {SmartNIC} for accelerating distributed systems. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*. 987–1004.

[58] Mingyu Wu. 2024. H2 Throttle. https://github.com/SJTU-IPADS/jade-artifacts/tree/master/h2-throttle.

[59] Mingyu Wu, Liang Mao, Yude Lin, Yifeng Jin, Zhe Li, Hongtao Lyu, Jiawei Tang, Xiaowei Lu, Hao Tang, Denghui Dong, et al. 2024. Jade: A High-throughput Concurrent Copying Garbage Collector. In *Proceedings of the Nineteenth European Conference on Computer Systems*. 1160–1174.

[60] Mingyu Wu, Ziming Zhao, Yanfei Yang, Haoyu Li, Haibo Chen, Binyu Zang, Haibing Guan, Sanhong Li, Chuansheng Lu, and Tongbao Zhang. 2020. Platinum: A CPU-Efficient Concurrent Garbage Collector for Tail-Reduction of Interactive Services. In *USENIX ATC*. 159–172.

[61] Tong Xing, Hesam Tajbakhsh, Israat Haque, Michio Honda, and Antonio Barbalace. 2022. Towards portable end-to-end network performance characterization of smartnics. In *Proceedings of the 13th ACM SIGOPS Asia-Pacific Workshop on Systems*. 46–52.

[62] Zhaoqi Xiong and Noa Zilberman. 2019. Do switches dream of machine learning? toward in-network classification. In *Proceedings of the 18th ACM workshop on hot topics in networks*. 25–33.

[63] Albert Mingkun Yang and Tobias Wrigstad. 2022. Deep dive into zgc: A modern garbage collector in openjdk. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 44, 4 (2022), 1–34.

[64] Fangxi Yin, Denghui Dong, Sanhong Li, Jianmei Guo, and Kingsum Chow. 2018. Java performance troubleshooting and optimization at Alibaba. In *ICSE-SEIP*. 11–12. doi:10.1145/3183519.3183536

[65] Zhuolong Yu, Yiwen Zhang, Vladimir Braverman, Mosharaf Chowdhury, and Xin Jin. 2020. Netlock: Fast, centralized lock management using programmable switches. In *Proceedings of the Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication*. 126–138.

[66] Junxian Zhao, Aidi Pi, Xiaobo Zhou, Sang-Yoon Chang, and Chengzhong Xu. 2022. Improving Concurrent GC for Latency Critical Services in Multi-tenant Systems. In *Proceedings of the 23rd ACM/IFIP International Middleware Conference*. 43–55.

[67] Wenyu Zhao, Stephen M Blackburn, and Kathryn S McKinley. 2022. Low-latency, high-throughput garbage collection. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. 76–91.

[68] Yang Zhou, Hassan MG Wassel, Sihang Liu, Jiaqi Gao, James Mickens, Minlan Yu, Chris Kennelly, Paul Turner, David E Culler, Henry M Levy, et al. 2022. Carbink:{Fault-Tolerant} far memory. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*. 55–71.

[69] Hang Zhu, Kostis Kaffes, Zixu Chen, Zhenming Liu, Christos Kozyrakis, Ion Stoica, and Xin Jin. 2020. {RackSched}: A {Microsecond-Scale} scheduler for {Rack-Scale} computers. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. 1225–1240.

A Artifact Appendix

Abstract

This artifact reproduces the figures and tables of the paper by running the DGC-enabled OpenJDK 17 fork against SPECjbb2015, HBase + YCSB, and DaCapo workloads under the four GC variants compared in the paper: G1, Shenandoah, DGC-SHM, and DGC-RDMA.

Scope

The artifact lets the evaluator validate the central latency and throughput claims of the paper, namely that disaggregating the concurrent marking phase to dedicated cores (via SHM or RDMA) reduces tail latency on managed workloads compared to G1 and Shenandoah at the same total core budget.

Contents

The artifact tree is rooted at `dgc-artifacts/` and contains:

- `jdk17-snic-gc/` — the bundled OpenJDK 17 fork implementing DGC-SHM and DGC-RDMA.
- `osdi26-scripts/` — per-figure / per-table top-level driver scripts, analyzers, and parsers.
- `conf/gc/` and `conf/workloads/` — GC profiles (G1, Shenandoah, DGC-SHM, DGC-RDMA) and workload knobs.
- `lib/` — shared shell libraries for CPU/NUMA pinning, isolation, and per-suite runners.
- `hbase-conf/` — HBase + YCSB drop-in config.
- `plot/` — Python plotting scripts.

Hosting

The artifact is publicly hosted on GitHub at https://github.com/SJTU-IPADS/dgc-artifacts on the `main` branch.

Requirements

Hardware. A NUMA Intel x86_64 server with hyperthreading disabled. The paper's core-budget settings assume dual-socket Intel Xeon Gold 6430 (32 physical cores per socket; even cores on NUMA node 0, odd cores on node 1) with 128 GB of RAM. DGC-RDMA additionally requires an RDMA NIC; the paper uses a 200 Gbps NVIDIA BlueField-3 DPU.
Operating system. Ubuntu 22.04 LTS.
External dependencies. The artifact tree expects the following sibling directories: the prebuilt DGC JDK image, SPECjbb2015 v1.0.4 (commercial; SPEC license required), HBase 2.5.11 + YCSB 0.18.0, the DaCapo 23.11-chopin throttle fork (built in-tree), and Google OR-Tools v9.10. The repository `README` documents the exact layout, download URLs, and build commands.

B CP-SAT Problem Description

To address the challenge of scheduling multiple GC tasks that can execute with different thread configurations, the following problem is formulated.

Let $\mathcal{J} = \{1, 2, \ldots, N\}$ represent the set of garbage collection tasks, and $\mathcal{M} = \{1, 2, \ldots, M\}$ denote the available thread configurations for each GC operation. For each task $i \in \mathcal{J}$ and thread configuration $n \in \mathcal{M}$, we define the following parameters:

- $d_{i,n}$: duration of GC task i when using n threads
- $r_{i,n}$: number of threads required by task i in configuration n
- δ_i: GC deadline for task i (latest safe completion time)

The total available GC threads in the system is constrained by C, representing the maximum concurrent GC threads the system can support.

B.1 Decision Variables

The mathematical model employs several decision variables to capture the scheduling decisions:

$$
\begin{aligned}
s_i &\in \mathbb{Z}^+ && \text{start time of GC task } i \\
e_i &\in \mathbb{Z}^+ && \text{end time of GC task } i \\
x_{i,n} &\in \{0, 1\} && \text{binary variable indicating whether task } i \\
& && \text{uses thread configuration } n
\end{aligned}
$$

B.2 Constraints

The scheduling must satisfy several hard constraints to ensure system stability and prevent out-of-memory conditions. Each GC task must be assigned exactly one thread configuration:

$$\sum_{n \in \mathcal{M}} x_{i,n} = 1 \quad \forall i \in \mathcal{J} \tag{1}$$

The completion time of each task is determined by its start time and the duration of the selected thread configuration:

$$e_i = s_i + \sum_{n \in \mathcal{M}} d_{i,n} \cdot x_{i,n} \quad \forall i \in \mathcal{J} \tag{2}$$

To prevent out-of-memory errors, each garbage collection must complete before its safety deadline:

$$e_i \leq \delta_i \quad \forall i \in \mathcal{J} \tag{3}$$

The most critical constraint ensures that the total number of active GC threads never exceeds system capacity C at any time instant:

$$\sum_{i \in \mathcal{J}} \sum_{n \in \mathcal{M}} r_{i,n} \cdot \mathbb{I}(s_i \leq t < e_i) \cdot x_{i,n} \leq C \quad \forall t \in [0, T_{\max}] \tag{4}$$

where $\mathbb{I}(\cdot)$ is the indicator function and $T_{\max}$ is the maximum time horizon.

B.3 Objective Function

The optimization aims to maximize the sum of start times to provide scheduling flexibility while discouraging inefficient thread configurations (particularly the configuration that falls back to native concurrent GC):

$$\text{maximize} \quad \sum_{i \in \mathcal{J}} s_i - \alpha \cdot \sum_{i \in \mathcal{J}} \sum_{n \in \mathcal{M}_{\text{ineff}}} x_{i,n} \tag{5}$$

where α is a penalty coefficient and $\mathcal{M}_{\text{ineff}} \subset \mathcal{M}$ represents the set of inefficient configurations.

DeLFS: A Decentralized Log-Structured File System for Manycores

Taehwan Ahn[†], Chanhyeong Yu[†], Sangjin Lee, and Yongseok Son[*]
Systems and Storage Laboratory, Chung-Ang University

Abstract

This paper introduces DeLFS, a novel decentralized log-structured file system (LFS) designed for manycore scalability. DeLFS decentralizes metadata/data organization and management with an LFS-aware decentralized locking scheme that distributes lock ownership to eliminate global contention. In addition, DeLFS disentangles the critical path from the deferrable path to further increase concurrency. We implement DeLFS with three techniques in the Linux kernel and evaluate it on a 128-core machine. The experimental results show that DeLFS outperforms F2FS, MAX, and F2FSJ by up to 4.34×, 4.29×, and 4.50×, respectively, and delivers up to 2.00× higher sustained performance than ScaleLFS.

1 Introduction

Log-structured file systems (LFSs) [9, 11, 12, 15, 16, 18, 20, 21, 24, 27, 30, 34] are attractive for modern storage systems because their append-only, log-structured design transforms small, random updates into large, sequential write streams, achieving high write efficiency on flash-based storage devices [18, 24, 26]. Despite these advantages, existing LFSs face scalability challenges because they rely on centralized architecture and coordination for metadata and data management, where multiple global locks serialize file operations across many cores [17, 23]. This architecture limits concurrency and parallelism, thereby degrading I/O performance.

Challenges: Figure 1 shows the application throughput in various modern LFSs (i.e., F2FS [18], MAX [21], ScaleLFS [11], F2FSJ [9]) under a random write workload generated by a micro-benchmark (i.e., FIO [4]). As shown in the figure, the raw performance of the NVMe SSD [2] scales well with the number of cores, reaching up to 5.24 GB/s. However, the performance of existing LFSs remains limited to around 1.06 GB/s even as the number of cores increases. In particular, even when we combine ScaleCache [29], a scalable page cache, with F2FS and other LFSs (as detailed in Section 3.1), the performance unexpectedly remains almost unchanged or even degrades. We observe that this limitation mainly stems from severe lock contention caused by multiple global locks on metadata and data management, serializing I/O operations and restricting concurrency across cores.

Prior studies: Previous studies have explored LFSs to improve scalability and reduce garbage collection (GC) overhead. F2FSJ [9] introduces a decentralized, epoch-based journaling technique for F2FS with an ordered journal mode to reduce checkpointing time. MAX [21] is a multicore-friendly LFS that scales access to in-memory index and cache while maintaining multiple independent logs, but it still suffers from lock contention. ScaleLFS [11] presents a scalable GC that parallelizes the GC procedure and concurrent victim management. Our study is inspired by and in line with these studies in terms of investigating LFS scalability and performance. In contrast, our study focuses on establishing a holistic decentralized LFS architecture with LFS-aware locking.

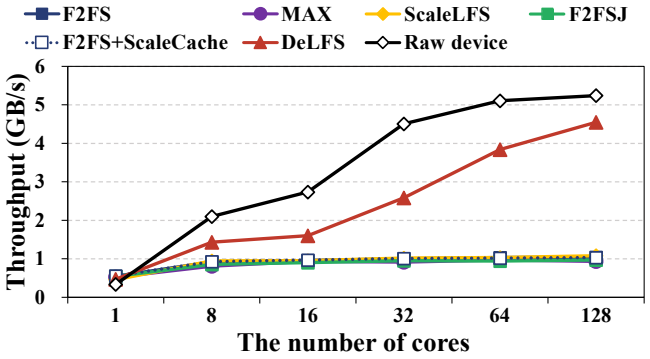
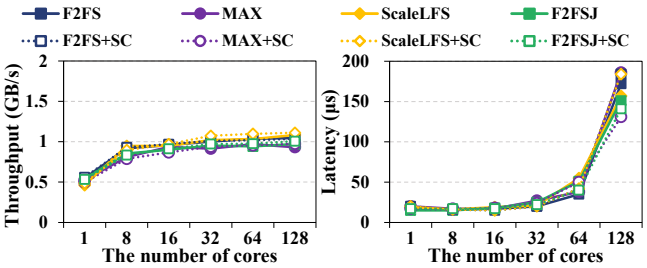


Figure 1: Scalability comparison of log-structured file systems.

Contributions: In this paper, we propose DeLFS, a decentralized log-structured file system designed to improve scalability on manycore systems. Our key idea is to *leverage the potential of distributing lock ownership across cores safely*, enabling a holistic, decentralized LFS architecture. Specifically, DeLFS first proposes *decentralized organization and management* of LFS resources by decomposing metadata and data layouts into per-core domains. This design enables a one(core)-to-one(resource) model and serves as the foundation of our decentralized architecture, where each core primarily operates on its own local metadata and data while minimizing cross-core interference.

Second, DeLFS adopts an LFS-aware *decentralized and decoupled locking* scheme that distributes lock ownership across cores while enabling deadlock-free acquisition. In addition, this scheme decouples certain updates into independently executable operations while still preserving atomic updates to decentralized resources. Thus, this locking mechanism eliminates global lock contention without compromising crash consistency. Finally, DeLFS presents lazy lock coordination, which disentangles critical and deferrable operations to further mitigate contention from inter-core decentralized lock sharing. To do this, DeLFS employs a lazy lock coordinator that handles deferred operations while preserving crash con-

[†]Equal contribution.
[*]Corresponding Author: Yongseok Son (sysganda@cau.ac.kr).



(a) Throughput. (b) Tail latency (99th).

Figure 2: Throughput and tail latency with various LFSs.

sistency. As a result, DeLFS accelerates both application I/O and GC I/O processing by mitigating lock contention and improving core scalability.

We implement DeLFS with three techniques based on F2FS [18] and ScaleCache [29] in Linux kernel 6.6.8. We evaluate it on a 128-core machine with the NVMe SSD [2] using the micro [4] and macro [31] benchmarks and real-world application [10]. The experimental results show DeLFS outperforms F2FS, MAX [21], ScaleLFS [11], and F2FSJ [9] by up to 4.34×, 4.29×, 4.10×, and 4.50×. Note that, even with ScaleCache, the results show DeLFS outperforms F2FS, MAX, ScaleLFS, and F2FSJ by up to 4.55×, 6.32×, 3.99×, and 4.38×. Furthermore, DeLFS delivers up to 2.00× higher sustained performance than ScaleLFS. To the best of our knowledge, DeLFS is the first holistic, decentralized log-structured file system with an LFS-aware locking scheme across cores in modern many-core systems. Finally, we open the source code of DeLFS and baselines used in our evaluation at https://github.com/syslab-CAU/DeLFS to aid future studies in LFSs.

2 Background and Motivation

A log-structured file system (LFS) follows an append-only design, transforming small random writes into large sequential write streams. F2FS manages storage in segments (e.g., 2 MB) and divides the space into a random-write area for on-disk metadata (e.g., SIT (segment information table), NAT (node address table), and SSA (segment summary area)) and a sequential main area for node and data blocks, further separating node/data into hot, warm, and cold temperatures to exploit device parallelism. However, the organization and management of these metadata structures and related data structures are centralized and protected by coarse-grained global locks shared across all cores. As a result, the locks serialize metadata and data updates, preventing full exploitation of parallelism and limiting scalability on many-core systems.

3 Scalability Analysis

3.1 Scalability Analysis on Modern LFSs

To understand the scalability performance of modern LFSs (F2FS, MAX, ScaleLFS, and F2FSJ), both with and without ScaleCache [29], we run FIO with random write on our testbed under the evaluation scenario as described in Section 5.1 and 5.2. Figure 2 shows the throughput and tail latency of LFSs on varying numbers of cores. The LFSs do not scale well and exhibit almost the same throughput even as the number of cores increases. In addition, latency spikes significantly at 128 cores compared with 64 cores. Furthermore, even when combined with ScaleCache [29], the performance of LFSs still does not improve and can even degrade (see details in Section 3.2). Consequently, these results indicate that the design of existing LFSs has limitations.

3.2 Identifying Root Causes

3.2.1 Limitation of Page Cache Flushing

The first root cause of low I/O performance in modern LFSs is the serialized page-flushing path in the page cache. Under data-intensive write workloads, a single flusher thread flushes dirty pages for one inode at a time while throttling application threads, a mechanism originally designed for HDD-based systems [29]. On modern many-core servers with high-performance SSDs, this design cannot provide sufficient I/O parallelism, and the throttling intervals (e.g., `io_schedule_timeout()`) become the first bottleneck (Table 1). ScaleCache [29] addresses this blocking bottleneck with direct page flush (dflush), allowing application threads to flush pages in parallel. Accordingly, we evaluate modern LFSs with ScaleCache. However, as shown in Table 1, the throughput improvement is negligible or even negative, indicating that LFSs suffer from additional bottlenecks.

3.2.2 Scalability Obstacles in LFSs

We analyze the design of modern LFSs to identify the main obstacles that hinder the scalability on manycores.

Serialized I/O process: After the blocking bottleneck by the throttling mechanism is eliminated, modern LFSs encounter the next bottleneck caused by lock contention on the serialization mutex (`writepages`) in the writeback path, as shown in Table 1. This lock serializes writeback on a per-inode basis, forcing flushers to wait until the current holder completes submitting pages for its inode. This design aligns file and segment lifetimes by keeping pages of a file within a segment, thereby avoiding inter-file interleaving. Thus, this design enforces per-file writeback, which significantly limits I/O parallelism. To enable more parallel I/O, we can simply eliminate this serialization lock without compromising correctness or consistency, as it only relaxes the alignment between file and segment lifetimes. However, pages from multiple inodes may become interleaved within a segment, potentially degrading GC efficiency; we report that its impact on write amplification is minor (Section 5.2). Despite this increased parallelism, LFSs still suffer from the scalability bottleneck in multi-head logging management.

File system	Different file system variants	Throughput	Execution time	io_schedule_timeout()	serialization mutex (writepages)	curseg_mutex	io_rwsem	others
F2FS [18]	F2FS	1.05 GB/s	279.7s (100%)	269s (96.17%)	2.2s (0.8%)	0.3s (0.1%)	2.5s (0.9%)	5.7s (2.03%)
	F2FS+SC	1.00 GB/s	293.7s (100%)	0s (0%)	152.7s (52%)	0.3s (0.1%)	0.3s (0.1%)	140.4s (47.8%)
	F2FS+SC+Serialization-free	1.23 GB/s	238.7s (100%)	0s (0%)	0s (0%)	114.5s (47.95%)	5.7s (2.39%)	118.5s (49.66%)
MAX [21]	MAX	1.06 GB/s	277.1s (100%)	269.4s (97.21%)	1.7s (0.6%)	0.6s (0.2%)	1.9s (0.7%)	3.5s (1.29%)
	MAX+SC	0.72 GB/s	407.9s (100%)	0s (0%)	215.8s (52.9%)	0.5s (0.13%)	3.3s (0.8%)	188.3s (46.17%)
	MAX+SC+Serialization-free	0.71 GB/s	413.6s (100%)	0s (0%)	0s (0%)	0.5s (0.1%)	213.8s (51.7%)	199.3s (48.2%)
ScaleLFS [11]	ScaleLFS	1.11 GB/s	264.5s (100%)	253.2s (95.7%)	2.2s (0.83%)	0.3s (0.13%)	1.6s (0.62%)	7.2s (2.72%)
	ScaleLFS+SC	1.14 GB/s	257.5s (100%)	0s (0%)	132.9s (51.6%)	0.5s (0.21%)	2.8s (1.1%)	121.3s (47.09%)
	ScaleLFS+SC+Serialization-free	1.49 GB/s	197.1s (100%)	0s (0%)	0s (0%)	8.4s (4.3%)	74.6s (37.9%)	114.1s (57.8%)
F2FSJ [9]	F2FSJ	1.01 GB/s	290.8s (100%)	281.5s (96.8%)	2.1s (0.72%)	0.5s (0.17%)	3.2s (1.1%)	3.5s (1.21%)
	F2FSJ+SC	1.04 GB/s	282.4s (100%)	0s (0%)	148.5s (52.6%)	0.4s (0.15%)	2.1s (0.74%)	131.3s (46.51%)
	F2FSJ+SC+Serialization-free	1.17 GB/s	251.0s (100%)	0s (0%)	0s (0%)	63.7s (25.4%)	64.5s (25.7%)	122.8s (48.9%)

Table 1: Performance breakdown of modern LFSs on 128 cores (SC: ScaleCache [29]).

Multi-head logging management: When enabling parallel I/O without the serialized design and lock (i.e., serialization-free), the bottleneck shifts to lock contention (`curseg_mutex`) from the space allocation in multi-head logging management as shown in Table 1. F2FS adopts a multi-head logging scheme with six active logs, where each log head corresponds to a current segment (`curseg_info`). Each current segment sequentially allocates new blocks, thereby preserving the append-only write pattern. However, the number of current segments is fixed at six even on many-core systems. Moreover, most data-intensive writes are allocated from the warm-data log, and a mutex, the current segment lock (`curseg_mutex`) must be acquired whenever a block is allocated. As a result, this centralized lock becomes a severe bottleneck, as cores contend for it during block allocation. Note that MAX [21] targets this lock contention and solves it by the mlog scheme, which allocates the space in a round-robin manner. However, the bottleneck shifts to the I/O submit management.

Submit management: LFSs maintain six bio structures (block I/O descriptors [7]), each corresponding to one of the six log streams, so that I/O requests can be batched and managed on a per-stream basis. During a flush, dirty pages are classified by type and temperature and appended to the corresponding bio. Each appended page is checked for merge opportunities with existing pages; if mergeable, the LFS performs the merge while holding the I/O submit lock (`io_rwsem`) to ensure consistency of the bio's internal state. However, this design, which maintains only a fixed set of six bios and protects their operations with the lock per bio, limits parallel I/O on many-core systems. We observe severe contention on this lock in MAX, ScaleLFS, and F2FSJ as shown in Table 1. In particular, when the lock contention on `curseg_mutex` is mitigated, the lock contention on `io_rwsem` becomes the next dominant bottleneck.

Segment management: Even if the bottleneck in submit management is mitigated, additional bottlenecks can still arise in the I/O path, as shown in Table 4. For example, segment-level metadata management also introduces synchronization overhead. The SIT tracks the state of all segments in the main data area. It contains multiple SIT entries, each of which corresponds to a segment and maintains both the number of valid blocks within the segment and a bitmap representing the validity of each block. When a block is allocated for a data page, the associated SIT entry is updated to mark it as valid. However, to ensure correct updates, the SIT entries are protected by a single read-write semaphore, the SIT entry lock (`sentry_lock`). Thus, even if only one SIT entry needs to be updated, this coarse-grained lock must be acquired, thereby preventing concurrent access to the entire SIT.

Segment list management: Furthermore, segments are tracked using multiple bitmaps according to their validity state: segments with a mix of valid and invalid blocks are recorded in the dirty-segment bitmap, while segments whose blocks are all invalid are recorded in the pre-segment bitmap and are later reclaimed as free. However, both the dirty and pre-segment bitmaps are protected by a single mutex (`seglist_lock`). As a result, as the state of segments changes, moving segments across the segment bitmaps requires acquiring this lock, which limits concurrency.

Node management: Beyond segment management, LFSs also face a scalability bottleneck in managing the NAT. Inodes and internal nodes are tracked via NAT entries, which are cached in memory in a global tree protected by a read-write semaphore (`nat_tree_lock`) and managed by an LRU-style NAT list protected by a spinlock (`nat_list_lock`). While this NAT cache reduces storage I/Os for node lookups, its centralized locking becomes a scalability bottleneck (Table 4). MAX [21] mitigates NAT contention using file cells, but its overall scalability remains limited because other bottlenecks are not addressed.

Discard management: Finally, modern LFSs also suffer from scalability bottlenecks in discard management. A global discard command controller maintains a list of discard entries protected by a single mutex (`cmd_lock`). During write I/O, each newly allocated block must acquire `cmd_lock` to check and remove any matching discard entry before reuse. Because `cmd_lock` is a global lock shared by all discard operations, it becomes a major source of lock contention once other bottlenecks are alleviated (see Table 4).

4 Design and Implementation

Our key idea is to *leverage the potential that lock ownership can be distributed across cores safely*, enabling a robust and decentralized LFS architecture. The rationale behind this design choice is that there are a lot of LFS resources and their locks, each of which protects a large critical section. Thus,

it is hard to remove these locks, which may be less robust; instead, we split large resources and their locks and distribute the lock ownership of each resource across cores.

4.1 Design Goals of DeLFS

We design DeLFS to satisfy the following three goals.

Decentralizing LFS resources: To design a scalable LFS, DeLFS should holistically decentralize LFS resources to minimize interference from shared and global resources and maximize core and I/O parallelism.

Minimizing lock contention with safe synchronization: To minimize global lock contention, DeLFS should provide a lightweight locking mechanism to maximize concurrency while preserving correctness via safe synchronization.

Coordinating inter-core sharing: To support inter-core sharing when necessary, DeLFS should provide a coordination technique to avoid waiting on operations from other cores without sacrificing consistency.

4.2 Overall Architecture of DeLFS

Overall, DeLFS redesigns the centralized organization and management of LFS resources into a fully decentralized metadata/data organization and its management, where each decentralized LFS resource is efficiently protected by decentralized locking and orchestrated by decentralized lazy lock coordination across cores. DeLFS consists of the following three techniques and strategies to meet the goals.

Strategy 1: Decentralized organization and management. To minimize interference from shared and global resources among cores, DeLFS decentralizes LFS metadata and data management into per-core domains. Each core owns and manages its own subset of decentralized metadata (e.g., SIT, NAT, multi-head logging, SSA as $M_{0\text{-}0} \ldots M_{k\text{-}0}$ for $core_0$) and its local segments (e.g., $seg_{0\text{-}0} \ldots seg_{k\text{-}0}$ for $core_0$). This one-core–one-resource design reduces cross-core interference and forms the foundation of DeLFS's decentralized architecture.

Strategy 2: Decentralized and decoupled locking. On top of this organization, DeLFS decentralizes locks so that each core primarily acquires locks on its own decentralized resources, rather than contending for global locks. However, when inter-core access is required, naive decentralized locking can still cause cross-domain deadlocks and skewed contention. To address this, DeLFS introduces decentralized and decoupled locking (DDL), which decouples certain updates into individually executable operations while preserving atomicity and enforces a lock-update-unlock discipline on one resource at a time. This single-lock-hold property breaks potential deadlock cycles, reduces lock hold time, and allows lock ownership to be safely distributed across cores while preserving correctness through the existing LFS checkpoint mechanism.

Strategy 3: Lazy lock coordination. To further reduce blocking from inter-core decentralized lock sharing, DeLFS introduces lazy lock coordination. We observe that many operations can be split into a critical path and a lazily deferrable path without affecting correctness. DeLFS therefore employs per-core lock coordinators and delegates remote, deferrable lock-acquisition tasks to the coordinator of the core that owns the target resource, so that the critical path is no longer blocked by remote lock acquisition. These delegated operations are processed lazily but are guaranteed to complete before checkpointing, where DeLFS's checkpoint procedure waits for and cooperatively processes pending tasks at each coordinator. As a result, lazy lock coordination accelerates critical operations while preserving LFS crash consistency.

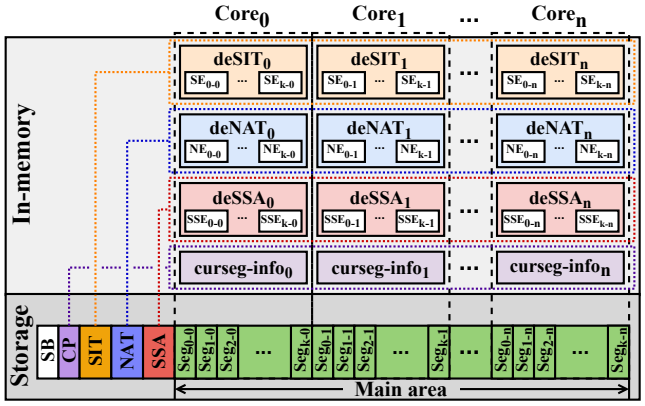


Figure 3: Decentralized organization of DeLFS (de: decentralized, SE: SIT entry, NE: NAT entry, SSE: SSA entry, Seg: Segment).

4.3 Decentralized Architecture

Decentralized organization: DeLFS decentralizes in-memory metadata and on-disk data and node segments into per-core domains, allowing each core to manage its own domain independently and thereby improving I/O scalability. For example, as shown in Figure 3, metadata structures such as $deSIT_0$–$deSIT_n$, $deNAT_0$–$deNAT_n$, $deSSA_0$–$deSSA_n$, and $curseg\text{-}info_0$–$curseg\text{-}info_n$ are decentralized across cores. Thus, each core primarily accesses its own metadata, avoiding contention on shared metadata structures. Specifically, per-core current segment information (curseg-info, curseg_info) allows each core to allocate space from its own multi-head log, naturally decentralizing node and data segment allocation. During page flushing, each core exclusively flushes all dirty data pages of a fetched inode to its own space. This isolates the inode's I/O path from other cores and prevents data pages from being interleaved across cores, further improving scalability and performance. Note that when a core does not have enough free segments in its own space, DeLFS allows it to allocate segments from the next core's domain in a round-robin manner, preserving space efficiency while maintaining decentralized organization. As a result, this architecture establishes a one(core)-to-one(resource) model that eliminates global metadata hot spots and enables highly parallel, low-contention I/O execution across cores.

Decentralized management: Figure 4 and 5 illustrate decentralized management and how the I/O operation is performed

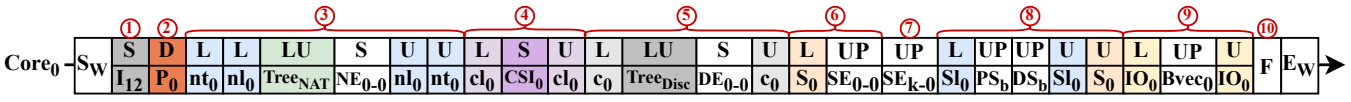


Figure 4: Decentralized management of DeLFS timeline ($\mathbf{S_W}$: start write, **S**: select, **D**: Dirty, **L**: lock, **U**: unlock, **LU**: lookup, **UP**: update, **F**: flush, $\mathbf{E_W}$: end write, $\mathbf{I_x}$: inode$_x$, $\mathbf{P_0}$: page$_0$, $\mathbf{nt_0}$: nat_tree_lock$_0$, $\mathbf{nl_0}$: nat_list_lock$_0$, $\mathbf{Tree_{NAT}}$: NAT tree, $\mathbf{NE_{0\text{-}n}}$: 0th NAT entry in core$_n$, $\mathbf{cl_0}$: curseg_mutex$_0$, $\mathbf{CSI_0}$: current segment information of core$_0$, $\mathbf{c_0}$: cmd_lock$_0$, $\mathbf{Tree_{Disc}}$: discard tree, $\mathbf{DE_{0\text{-}n}}$: 0th discard entry in core$_n$, $\mathbf{S_0}$: sentry_lock in core$_0$, $\mathbf{SE_{0\text{-}n}}$: 0th SIT entry in core$_n$, $\mathbf{Sl_0}$: seglist_lock$_0$, $\mathbf{PS_b}$: pre-segment bitmap, $\mathbf{DS_b}$: dirty-segment bitmap, $\mathbf{IO_0}$: io_rwsem$_0$, $\mathbf{B_{vec0}}$: bio vector$_0$).

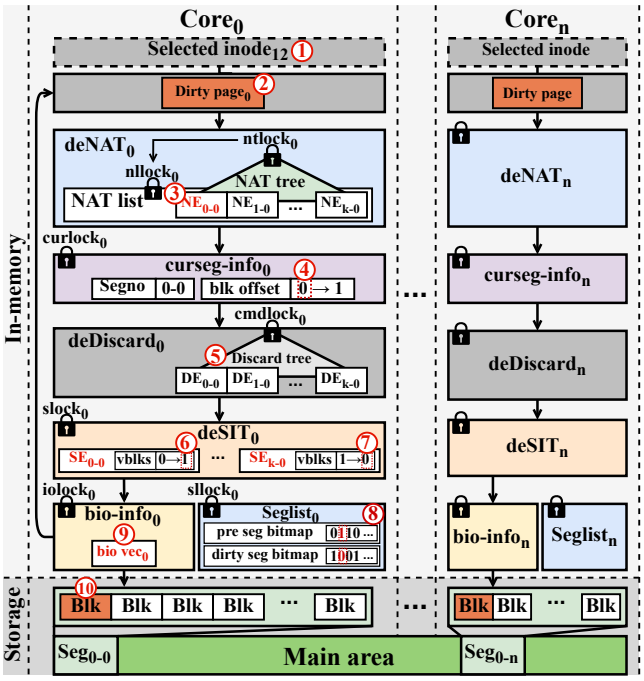


Figure 5: Decentralized management of DeLFS (**NE**: NAT entry, **DE**: Discard entry, **SE**: SIT entry, **Seg**: Segment).

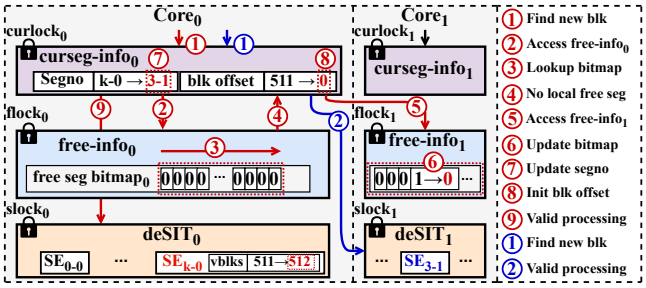


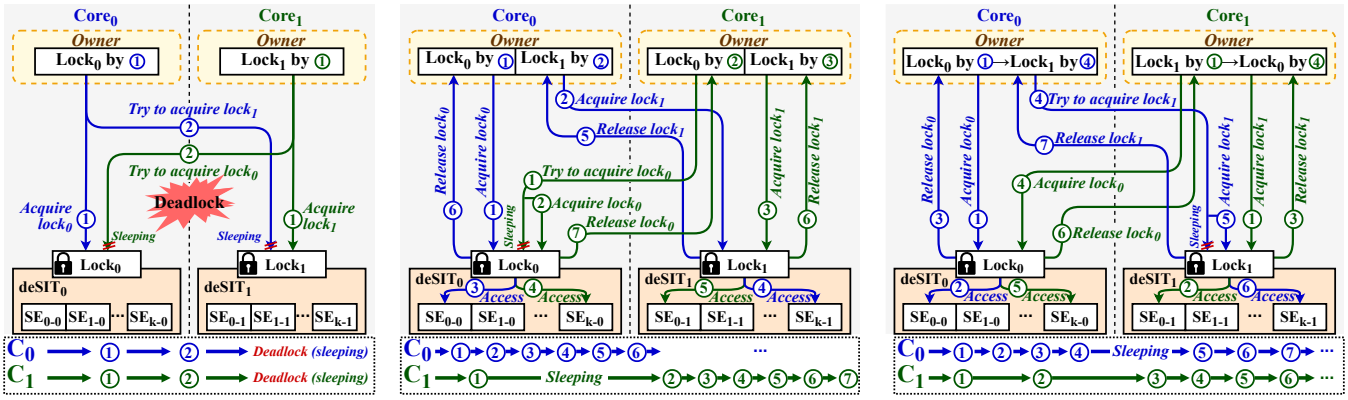
Figure 6: Cross-core space allocation.

in DeLFS. In these figures, a core begins by selecting an inode (inode$_{12}$) to flush dirty data pages belonging to that inode (①). To flush dirty page$_0$ (②), the core first locates the corresponding NAT entry (NE$_{0\text{-}0}$) in deNAT$_0$ (③) and obtains the next writable block address from its own current segment (curseg-info$_0$), which provides the target segment (segno$_{0\text{-}0}$) and the target block offset (④). The core then increments the block offset in curseg-info$_0$ to prepare for the next allocation. After obtaining the target block, the core checks whether the block is associated with a discard entry (⑤). Then, the core updates the SIT entry for segno$_{0\text{-}0}$ by increasing its valid-block count and setting its bit in the valid-block bitmap (⑥). If there is an old block to invalidate, the core decreases the valid-block count and clears its bit in the valid-block bitmap (⑦). If the valid-block count becomes 0, the core sets its bit in the pre-segment bitmap and clears its bit in the dirty-segment bitmap (⑧). Finally, the core records the page and its destination block in bio-vec$_0$ and links it into bio-info$_0$ (⑨). This processing for dirty pages of the inode is repeated until all its dirty pages are processed. After this process is completed, the core flushes current bio-vec entries in its bio-info$_0$ to storage (⑩). Consequently, since each core uses its own decentralized current segment, the entire I/O path for an inode remains isolated to that core, so that its data pages are written using only segments owned by that core, without being interleaved with blocks from other cores.

Cross-core space allocation: Figure 6 illustrates how core$_0$ allocates free space from another core's area when its current segment is exhausted and no local free segments remain. In the I/O process, the core first obtains the block at offset 511 of SE$_{k\text{-}0}$ from curseg-info$_0$ (①); after this allocation, all 512 blocks of SE$_{k\text{-}0}$ are consumed. To prepare subsequent allocations, the core removes SE$_{k\text{-}0}$ from curseg-info$_0$, acquires the free-info lock (flock$_0$), and scans the free segments in free-info$_0$ for a replacement segment (②,③). Because all bits in free-info$_0$ are clear, indicating that no space assigned to core$_0$ is free, it releases flock$_0$ (④) and, following a round-robin policy, accesses the free-space domain of core$_1$ by acquiring flock$_1$ and checking free-info$_1$ (⑤). Then, it finds SE$_{3\text{-}1}$ marked as free and marks it as occupied (⑥), stores SE$_{3\text{-}1}$ into the segno of curseg-info$_0$ (⑦), and resets the block offset to 0 (⑧). After completing this replacement, the core proceeds to the next step (⑨). When a subsequent request for a block occurs (①), it checks curseg-info$_0$ to select block 0 in SE$_{3\text{-}1}$, updates the valid-block count of SE$_{3\text{-}1}$, and completes the allocation (②).

4.4 Decentralized and Decoupled Locking

Structure of decentralized locks: If a decentralized LFS resource is not shared across cores, no lock is required. However, even under a decentralized design, some resources must still be shared among application threads, GC threads, and the checkpoint thread. Thus, as shown in Figure 5, DeLFS devises per-core decentralized locking for each LFS resource and allows each core to acquire the corresponding lock whenever it accesses its decentralized resource. To do this, as shown in the figure, DeLFS provides de-



(a) Deadlock in decentralized locking. (b) Deadlock-free locking with strict ordering. (c) Decentralized and decoupled locking of DeLFS.

Figure 7: Comparison of decentralized locking and ordering (**deSIT$_0$**: decentralized SIT in core$_0$, **SE$_{k\text{-}0}$**: kth decentralized SIT entry in core$_0$, **Lock$_0$**: decentralized SIT lock$_0$ in core$_0$ (sentry_lock$_0$), **C$_n$**: Core$_n$).

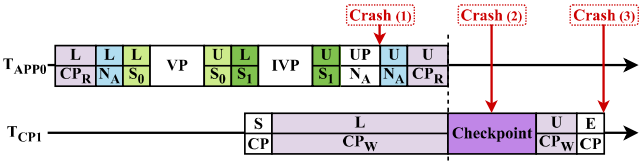


Figure 8: Crash consistency with DDL (**T$_{APP_0}$**: application thread in core$_0$, **T$_{CP_1}$**: checkpoint thread in core$_1$, **CP$_{R/W}$**: checkpoint reader/writer lock, **L**: lock, **VP**: valid processing, **IVP**: invalid processing, **U**: unlock, **UP**: update, **N$_A$**: node A, **S$_n$**: sentry_lock$_n$, **S**: start, **E**: end, **CP**: checkpoint).

centralized locks. For example, in core$_0$, the decentralized NAT tree is protected by nat_tree_lock$_0$ (ntlock$_0$), and the decentralized NAT list is protected by nat_list_lock$_0$ (nllock$_0$). Subsequently, the decentralized current segment, SIT, I/O submission, segment list, and discard management are protected by curseg_mutex$_0$ (curlock$_0$), sentry_lock$_0$ (slock$_0$), io_rwsem$_0$ (iolock$_0$), seglist_lock$_0$ (sllock$_0$), cmd_lock$_0$ (cmdlock$_0$), respectively.

Lock ordering and consistency: Furthermore, DeLFS preserves the original inner lock ordering of F2FS within each core as depicted in Figure 4 and 5. Specifically, when a core updates NAT, it first acquires the NAT-tree lock and, while holding it, acquires the NAT-list lock (③ in Figure 4); similarly, it acquires the SIT lock and, while holding it, acquires the segment-list lock (⑥,⑦,⑧ in Figure 4). Thus, the lock ordering within a core is preserved without consistency issues.

Lock ownership and potential issue: Each core has ownership of its decentralized lock associated with its own local resource instance and primarily acquires the decentralized lock, thereby reducing lock contention by localizing access. However, since decentralized resources can be shared among threads as mentioned, the lock ownership of a resource instance can be shifted across cores. In such cases, this shift in lock ownership can lead to a critical deadlock issue. For example, when a core naively attempts to acquire more than one lock across decentralized resources, the system can fall into a deadlock, which is described in more detail as follows.

4.4.1 Deadlock issue in decentralized locking

Deadlock issue: When accessing a resource in another core's domain, the core must acquire the decentralized lock for the target resource owned by another core. In this case, naive decentralized locking can incur a deadlock. For example, when updating the SIT for an overwrite page, there are two operations: (1) marking the newly allocated page as valid and (2) invalidating the old page. Thus, during write operations, each core allocates a new block for a dirty page in its own space but may also need to invalidate the corresponding old block if that block resides in another core's space. In this process, each core first acquires its own decentralized lock to update its own SIT entry to mark the new block as valid and, while holding this lock, then attempts to acquire the decentralized lock of the core that owns the old block in order to update that core's SIT entry and mark the old block as invalid. When multiple cores behave in this way concurrently, they can end up holding their own locks while waiting for each other's locks, which can create a circular wait and result in a deadlock.

Deadlock scenario: Figure 7a illustrates the deadlock scenario under decentralized locking. Core$_0$ and core$_1$ first acquire their own decentralized locks (lock$_0$ and lock$_1$) (①,①) to update their own SIT entries (deSIT$_0$ and deSIT$_1$) to mark the new blocks as valid. While holding these locks, core$_0$ then tries to acquire lock$_1$ (②), whereas core$_1$ tries to acquire lock$_0$ (②) simultaneously to perform the invalidation of the old blocks. Consequently, the two cores require more than one lock at the same time, thereby leading to a deadlock.

4.4.2 Deadlock-free locking with strict ordering

As a baseline for deadlock prevention, we consider a deadlock-free locking scheme with strict ordering called DFL. In this scheme, each core always acquires locks in ascending core-number order and releases them in descending core-number order. This strict lock-ordering discipline guarantees deadlock

freedom by eliminating cyclic lock dependencies, similar to conservative two-phase locking [5, 32].

Execution example: Figure 7b illustrates an example of how DFL avoids deadlock by enforcing a predetermined global lock order. As shown in the figure, we assume that $core_0$ needs to update both its own decentralized (de) SIT entry ($SE_{0\text{-}0}$) for valid processing and another core's deSIT entry ($SE_{k\text{-}1}$) for invalid processing. $Core_1$, in turn, performs the opposite: it updates its own deSIT entry ($SE_{0\text{-}1}$) for valid processing and $core_0$'s deSIT entry ($SE_{k\text{-}0}$) for invalid processing.

Strict lock ordering: However, note that even if $core_1$ is supposed to update its own deSIT entry for valid processing, it must first acquire $core_0$'s lock to keep the lock-ordering rule of DFL, which enforces that locks must always be acquired in ascending core-number order. Accordingly, under this rule, both cores first attempt to acquire the lock for $deSIT_0$ (①,①) in the ascending order. In this case, $core_0$ wins the contention and acquires the lock, while $core_1$ sleeps waiting for its release. After acquiring the lock for $deSIT_0$, $core_0$ acquires the lock for $deSIT_1$ (②) and performs the updates to $SE_{0\text{-}0}$ and $SE_{k\text{-}1}$ (③,④). Once both locks are released in the descending order (⑤,⑥), the sleeping $core_1$ wakes up, acquires the lock for $deSIT_0$ (②) and then the lock for $deSIT_1$ (③), and performs the updates to $SE_{0\text{-}1}$ and $SE_{k\text{-}0}$ (④,⑤). Finally, $core_1$ releases both locks in the descending order (⑥,⑦).

Scalability limitation: By doing so, DFL successfully prevents deadlock while preserving strong consistency, similar to F2FS, since it performs updates only after acquiring all required locks. However, this design forces cores to block while waiting on each other's locks. As a result, it can incur blocking overhead and degrade scalability on manycores, particularly when there is a large number of invalid pages and capacity is not enough (see Section 5.5).

4.4.3 Decentralized and decoupled locking in DeLFS

To mitigate blocking overhead while preventing deadlocks, DeLFS adopts a decentralized and decoupled locking scheme, called DDL, as its fundamental concurrency control mechanism. In this mechanism, note that the key principle of DDL is that each core never attempts to hold more than one lock of the same type at a time, which provides a loose lock ordering and prevents deadlocks. This principle is enabled by the fact that certain metadata updates can be decoupled while preserving both in-memory and crash consistency. For example, as mentioned in Section 4.4.1 and 4.4.2, overwriting a page requires updating two SIT entries. In this case, we observe that these two updates on disjoint SIT entries can be performed by decoupling valid processing on the new SIT entry from invalid processing on the old SIT entry. This is enabled by two properties: (1) The node is always updated to point to the new valid block only after both SIT entries have been updated under the node lock, ensuring that it always points to a valid block and maintains in-memory consistency (Figure 8); and (2) the checkpoint always recovers a consistent SIT state, guaranteeing crash consistency (Figure 8). With these properties, DDL performs two separate lock-update-unlock sequences: it first acquires the lock for one SIT entry, updates the entry, and releases the lock; it then acquires the lock for the other SIT entry, updates it, and releases the lock.

Procedure: Figure 7c depicts how DDL prevents deadlocks without forcing cores to block on each other's locks. We assume that $core_0$ needs to update $SE_{0\text{-}0}$ and $SE_{k\text{-}1}$, while $core_1$ needs to update $SE_{0\text{-}1}$ and $SE_{k\text{-}0}$. Both cores first acquire their own locks on $deSIT_0$ and $deSIT_1$, respectively (①,①). $Core_0$ then updates $SE_{0\text{-}0}$ (②) and releases its lock (③), and then tries to acquire the lock for $deSIT_1$. In this case, $core_0$ must wait for the lock on $deSIT_1$ (④) since $core_1$ still holds that lock. After $core_1$ completes the update and releases the lock (②,③), $core_0$ acquires the lock for $deSIT_1$ (⑤), updates $SE_{k\text{-}1}$ (⑥), and releases it (⑦). Meanwhile, $core_1$ acquires the lock for $deSIT_0$ (④), updates $SE_{k\text{-}0}$ (⑤), and releases it (⑥). By doing so, DDL reduces cross-core blocking on decentralized locks while guaranteeing deadlock-free execution.

Crash consistency: DDL updates only one SIT entry at a time and can therefore introduce transient inconsistencies between entries. However, DDL preserves correctness and eventually converges to a consistent state by leveraging the existing node lock, LFS checkpoint mechanism, and its reader-writer lock (i.e., cp_rwsem) which ensures eventual consistency. Specifically, as shown in Figure 8, when flushing a dirty page to the device, the application thread (T_{APP0}) first acquires the checkpoint reader lock (CP_R), the node (N_A) lock associated with the page, and the decentralized SIT entry lock in $core_0$ (S_0). Then, the thread performs valid processing for the associated SIT entry in $core_0$ and releases S_0. Subsequently, the thread acquires the decentralized SIT entry lock in $core_1$ (S_1), performs invalid processing for the associated SIT entry, and releases S_1. Meanwhile, during invalid processing, the checkpoint is triggered, at which point the checkpoint thread (T_{CP1}) acquires the checkpoint writer lock (CP_W). After this point, upcoming application threads cannot acquire CP_R and must wait until CP_W is released. Also, the checkpoint thread waits until the prior readers release all CP_R locks before starting the checkpoint. After invalidating the SIT entry, the application thread updates the location of the new page to the node and releases the node lock and CP_R; by doing so, the in-memory consistency is preserved. The checkpoint thread then performs the checkpoint, finalizes it with a commit record, and finally releases CP_W.

As shown in the figure, even if crash (1) or (2) occurs, since the checkpoint is not completed, the recovery restores the previous state. When crash (3) occurs, since the checkpoint includes the commit record, the recovery restores the newly updated node and SIT entries. As a result, the system eventually converges to the same SIT state as an atomic update, and DDL does not incur any crash consistency issues. Note that we observe that the checkpoint occurs infrequently due to the roll-forward recovery [19, 20, 27, 33], resulting in

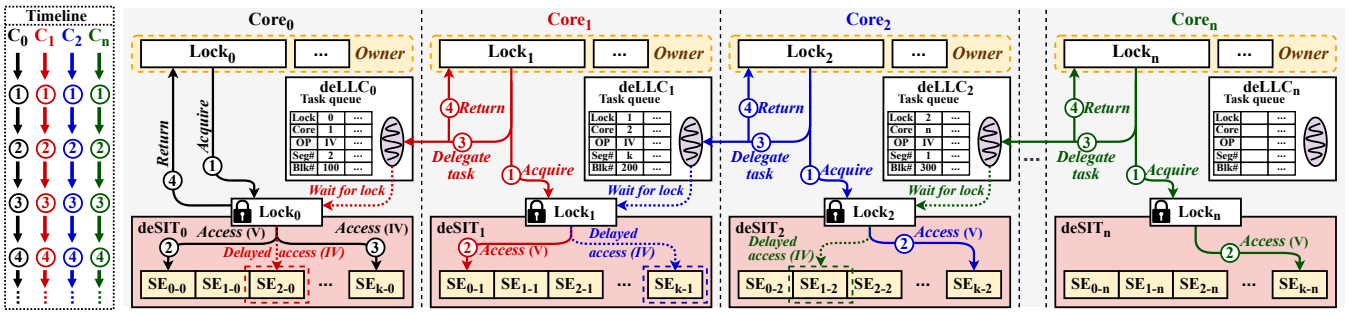


Figure 9: Decentralized lazy lock coordinator in DeLFS (**SE**: SIT entry, **deLLC**: decentralized lazy lock coordinator, **OP**: operation to execute, **V/IV**: valid/invalid processing, $\mathbf{C_n}$: core_n).

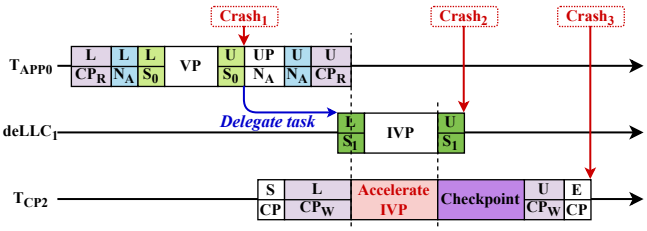


Figure 10: Crash consistency with lazy lock coordination ($\mathbf{T_{APP0}}$: application thread in core_0, $\mathbf{deLLC_1}$: decentralized lazy lock coordinator in core_1, $\mathbf{T_{CP2}}$: checkpoint thread in core_2, $\mathbf{CP_{R/W}}$: checkpoint reader/writer lock, **L**: lock, **U**: unlock, **UP**: update, **S**: start, **E**: end, $\mathbf{N_A}$: node A, $\mathbf{S_0}$: $\texttt{sentry_lock}_0$, **VP**: valid processing, **IVP**: invalid processing, **CP**: checkpoint).

minimal checkpoint overhead and lock contention.

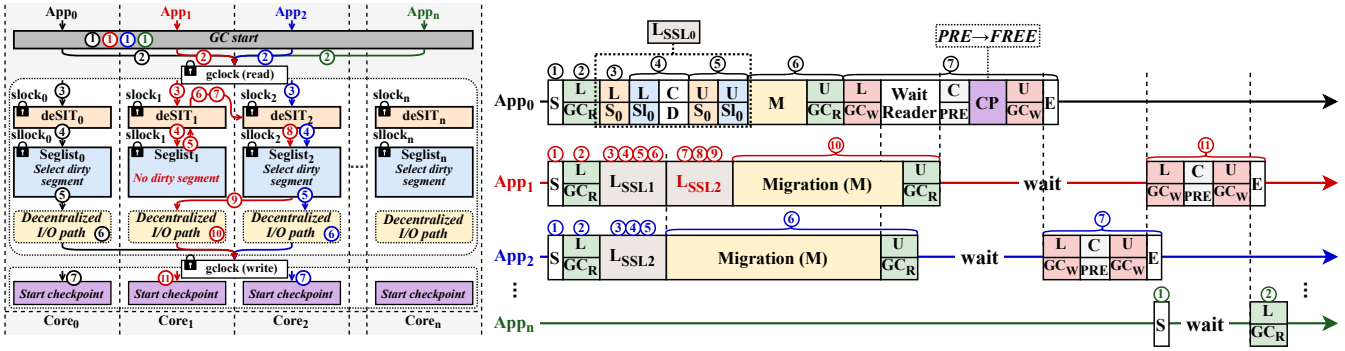
4.5 Lazy Lock Coordination

Even though DDL reduces the blocking overhead caused by the strict lock ordering of DFL, it can still incur substantial blocking because threads must wait on locks held by other cores. In particular, cross-core locking becomes frequent in two situations: (1) when a core has exhausted its local space and is forced to allocate from spaces owned by other cores, and (2) when a core needs to process an invalid page that resides in another core's space. In both cases, the core must acquire a decentralized lock for a resource owned by another core, and this cross-core waiting time can become large, especially under space exhaustion and heavy invalidation (see Section 5.5). For example, as shown in Figure 7c, core_0 must wait for lock_1, which is already held by core_1, even after it finishes valid processing for the new data page, which is a performance critical path.

To this end, DeLFS introduces a decentralized lazy lock coordinator (deLLC) that further mitigates cross-core blocking on decentralized locks while maintaining the DDL rule. In our design, the coordinator performs invalid processing for pages in remote spaces on behalf of the application thread. This decouples invalid processing from the critical I/O path of the application thread, avoiding blocking caused by remote lock acquisition. Specifically, application threads perform the local valid update under their own local locks, release those locks, and then delegate the invalid update to the corresponding coordinators for lazy processing. Thus, this design accelerates the I/O path of application threads. Note that deferring invalidation in this way may transiently leave two valid blocks, but it does not compromise either in-memory or crash consistency for the following reasons: (1) The valid update is always completed first under the node lock, ensuring that the node never points to an invalid block. Once the valid update and the node update are completed, the old block is no longer referenced, and (2) all deferred invalid processing is completed before the checkpoint is committed. As a result, deLLC guarantees consistency and is effective under both space exhaustion and heavy invalidation, since in both situations a core must frequently access decentralized locks owned by other cores.

Procedure: Figure 9 illustrates how deLLC operates. In this example, when a dirty page is flushed, the new page is allocated and marked valid, while the old page is marked invalid. The cores (core_0, core_1, core_2, and core_n) first acquire their local locks (①,①,①,①) to perform valid processing for their new pages (②,②,②,②). In the case of core_0, since both the new and old pages reside in its own decentralized SIT, the application thread can directly update both the valid and invalid bits for the corresponding SIT entries (e.g., $\text{SE}_{0\text{-}0}$ and $\text{SE}_{k\text{-}0}$) (②, ③) and then immediately return (④) without involving deLLC. For other cores, after completing the valid update (i.e., $\text{SE}_{0\text{-}1}$, $\text{SE}_{k\text{-}2}$, $\text{SE}_{k\text{-}n}$) (②,②,②), each thread delegates its invalidation task to the corresponding deLLC instance that owns the target SIT entries (i.e., $\text{SE}_{2\text{-}0}$, $\text{SE}_{k\text{-}1}$, $\text{SE}_{1\text{-}2}$) (③,③,③). After delegation, each thread immediately returns (④,④,④), leaving the invalid processing to the coordinators. By doing so, since the invalidation task is processed asynchronously by deLLC, this can accelerate the critical path of valid processing.

Crash consistency: DeLFS devises a checkpoint mechanism to support the asynchronous operation of deLLC. The checkpoint waits for and cooperatively processes all pending delegated tasks. Specifically, Figure 10 illustrates how DeLFS maintains crash consistency with the devised checkpoint mechanism. The initial procedure is similar to that of DDL until the application thread performs valid processing on the associated SIT entry in core_0. After that, in contrast to DDL, the thread delegates the task of invalidation to deLLC_1. Sub-



(a) GC procedure in DeLFS. (b) GC timeline in DeLFS.

Figure 11: Garbage collection of DeLFS (**slock$_0$**: sentry_lock of core$_0$, **sllock$_0$**: seglist_lock of core$_0$, **gclock**: gc_lock, **App$_0$**: application thread in core$_0$, **S**: start GC, **L**: lock, **U**: unlock, **C**: check, **M**: migration, **GC$_R$**: GC reader lock, **GC$_W$**: GC writer lock, **S$_0$**: slock$_0$, **Sl$_0$**: sllock$_0$, **D**: dirty segment, **L$_{SSL0}$**: handle segment and segment list lock/check dirty segment of core$_0$, **PRE**: pre-segments, **FREE**: free segments, **E**: end GC).

sequently, the thread updates the node with the new page location and releases the node$_A$ (N$_A$) lock and CP$_R$. Meanwhile, after releasing CP$_R$, under CP$_W$, the checkpoint thread starts cooperatively processing the pending tasks in deLLC$_1$, which accelerates the invalid processing. Note that this procedure ensures completion of all pending delegated tasks in all deLLCs. After this procedure, the checkpoint thread performs the checkpoint and writes the commit record. Similar to the case of DDL, when a crash occurs before or after writing the commit record, the recovery restores the previous or the updated state. By doing so, DeLFS safely disentangles the critical path from deferred invalidation work and thereby accelerates the critical path.

4.6 Decentralized Garbage Collection

As an LFS ages, free segments decrease, and once they fall below a threshold, the system triggers garbage collection (GC). In existing LFSs (F2FS, MAX, F2FSJ), GC is serialized: application threads are blocked and a single GC thread performs all GC work until enough free segments are reclaimed [11]. ScaleLFS improves this by using per-core GC daemon threads and per-core cold-data write streams to parallelize GC. However, it still (1) uses a global segment list, so each core must scan all segments to find a victim, and (2) does not optimize normal I/O, so GC can increase the total execution time, including both GC and foreground I/O (see Section 5.2). Meanwhile, DeLFS provides a holistic, decentralized GC procedure as described in Figure 11a and 11b as well as a decentralized normal I/O path. The key idea is to enable each core to perform GC procedure based on its own decentralized resources similar to the normal I/O path, effectively localizing GC. To do this, DeLFS leverages the GC reader lock during GC execution and the GC writer lock at checkpoint time.

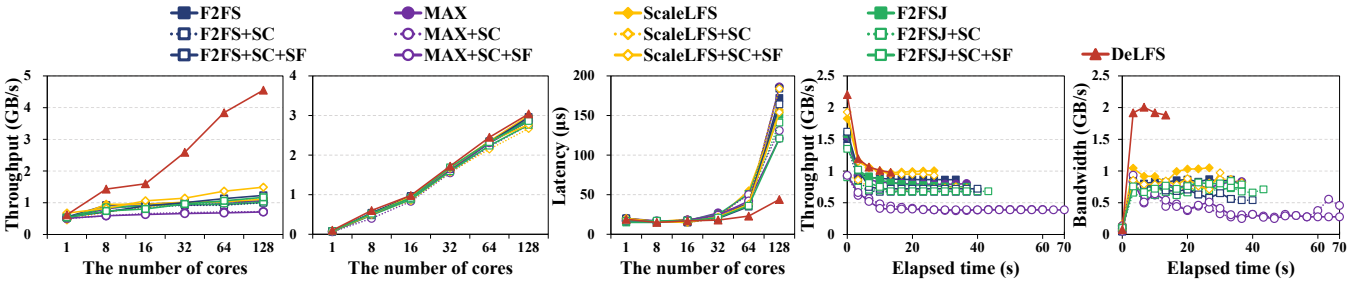
Specifically, when the number of free segments falls below the predefined threshold, DeLFS triggers GC, and application threads begin GC (①,①,①). Unlike serialized GC and ScaleLFS, these threads acquire the shared GC$_R$ lock (②,②,②), enabling concurrent and cooperative GC. Each thread then acquires its own deSIT lock (slock) and segment-list lock (sllock) (③,③,③) and selects a victim segment from its local segment list (④,④,④). If a core (core$_1$) has no dirty segment, the thread (app$_1$) releases its local locks (⑤,⑥) and, in round-robin order, moves to another core's domain (⑦,⑧) to help perform GC for that core. Each thread then performs migration independently through the decentralized I/O path (⑤,⑨,⑤) and releases GC$_R$ when its migration is done (⑥,⑩,⑥). The earliest thread to finish (app$_0$) releases GC$_R$ and then acquires GC$_W$ to block new GC starters, and waits until all remaining threads holding GC$_R$ release it. After app$_2$ and app$_1$ finish migration and release GC$_R$, no thread holds GC$_R$ anymore, so app$_0$, still holding GC$_W$, checks whether any pre-segments consisting only of invalid pages exist as a result of migration. Then, app$_0$ converts the pre-segments into free segments via the checkpoint process under the checkpoint writer lock and releases GC$_W$ (⑦).

Subsequently, app$_2$, which is waiting for GC$_W$, acquires GC$_W$, finds no remaining pre-segments because app$_0$ has already processed them, and immediately releases GC$_W$ (⑦). Likewise, app$_1$ finds no remaining pre-segments and completes GC (⑪). If app$_n$ attempts to start GC (①) while GC$_W$ is held, it cannot acquire GC$_R$ and must wait; once GC$_W$ is released, it starts a new GC cycle (②). Although localized GC may select a higher-cost victim segment than a globally optimal scheme, the impact on the total write amplification is minor in our evaluation (Section 5.2). Even when dirty segments are skewed across cores, GC gradually expands from local to other cores' segments, ensuring global reclamation progress. Finally, decentralized GC remains crash-consistent because all GC activities and checkpoints are safely synchronized by the GC locks.

5 Evaluation

5.1 Experimental Setup

Testbed: We use a 128-core machine equipped with two 64-core AMD EPYC 7713 processors and 96 GB of DRAM. For



(a) Random write. (b) Random read. (c) Tail latency (99th). (d) Application TP. (e) Device-level BW.

Figure 12: Throughput, tail latency, and GC performance of LFSs (SC: ScaleCache, SF: serialization-free, BW: bandwidth, TP: throughput).

LFS \ Combination	Baseline	SC	SC + SF
F2FS	2.42 %	9.56 %	51.5 %
MAX	2.29 %	8.73 %	45.3 %
ScaleLFS	2.99 %	8.26 %	49.37 %
F2FSJ	2.45 %	8.96 %	57.03 %
DeLFS	47.56 %		

Table 2: CPU utilization (SC: ScaleCache, SF: serialization-free).

storage, the machine is equipped with a FireCuda 530 NVMe SSD [2]. The SSD has a capacity of 2 TB and provides raw random write, read, and mixed random read/write throughput of 5.3 GB/s, 3.2 GB/s, and 3.5 GB/s, respectively.

Comparison: We compare DeLFS with modern LFSs, including F2FS [18], MAX [21], ScaleLFS [11], and F2FSJ [9], as well as their variants combined with ScaleCache (SC) [29] and the serialization-free (SF) scheme (mentioned in Section 3.2). We run all LFSs and their variants on Ubuntu 22.04 LTS with the Linux 6.6.8 kernel. For evaluation, we use FIO [4] as a micro-benchmark, filebench [31] as a macro-benchmark, and RocksDB [10] as a real-world application with YCSB [1]. We run each test ten times and report the average result.

5.2 Micro-benchmark

We present the performance results of DeLFS and modern LFSs using the FIO benchmark. We vary the number of threads from 1 to 128, and use a 2.2 GB file size per thread and a 4 KB request size. In the case of 128 threads, the total dataset size is 281.6 GB. We use this configuration as the default in all micro-benchmark tests unless stated otherwise.

Core scalability: Figure 12a shows the core scalability in the case of random write. The existing LFSs show almost the same performance, or even degrade even if the number of cores increases. Meanwhile, DeLFS scales performance well on various number of cores. Specifically, DeLFS achieves 4.34×, 4.29×, 4.10×, and 4.50× compared with F2FS, MAX, ScaleLFS, and F2FSJ at 128 cores, respectively. This is because the existing LFSs suffer from the first bottleneck (i.e., blocking overhead) by I/O throttling in dirty-page management as described in Section 3.2.1. To eliminate the blocking overhead, even when combining LFSs with SC, the existing LFSs still suffer from the serialization lock as the second bottleneck. Furthermore, even if the serialization lock is eliminated (i.e., serialization-free (SF)), contention on subsequent locks becomes the next bottleneck, as shown in Table 1. Thus, with the decentralized design, DeLFS can achieve 3.70×, 6.41×, 3.05×, and 3.89× higher throughput compared with F2FS+SC+SF, MAX+SC+SF, ScaleLFS+SC+SF, and F2FSJ+SC+SF at 128 cores, respectively. Consequently, these results demonstrate that DeLFS effectively scales performance as the number of cores increases by leveraging decentralized architecture and locking schemes.

Read-only performance: We evaluate the read-only performance of DeLFS while increasing the number of cores as shown in Figure 12b. Overall, the read performance of DeLFS is similar to that of the other LFSs. This is because all LFSs already achieve throughput close to the raw read performance since the read path allows multiple threads to issue read operations concurrently under read locks. Consequently, the design of DeLFS does not affect read performance.

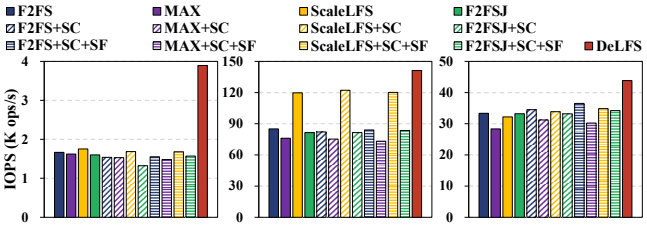
Tail latency: Figure 12c shows tail latency in all the LFS configurations while varying the number of cores. As shown in the figure, at 128 cores, when comparing the best-performing configurations among the LFS variants, DeLFS achieves 3.73×, 2.75×, 3.5×, and 2.75× lower tail latency than F2FS+SC+SF, MAX+SC+SF, ScaleLFS+SC+SF, and F2FSJ+SC+SF, respectively. In particular, while existing LFSs exhibit sharp latency spikes between 64 and 128 cores, DeLFS suppresses these spikes and keeps the latency much more stable. As a result, DeLFS outperforms the existing LFSs in terms of tail latency as well as throughput.

CPU utilization: Table 2 shows CPU utilization across all LFS configurations. As expected, the variant LFSs exhibit higher CPU utilization than the existing LFSs, since they can proceed without I/O throttling and the serialization lock. DeLFS shows the highest CPU utilization among the existing LFSs because it accelerates I/O processing with greater concurrency and parallelism. As a result, DeLFS achieves higher multi-core scalability at the cost of increased CPU utilization.

Garbage collection: Figure 12d and 12e present the application throughput and device-level bandwidth during GC. To fairly evaluate GC performance against ScaleLFS [11], we follow the experimental setup described in the study. Specifically, we use a 30 GB partition and run the experiment with 128 threads, each issuing 228 MB of random writes, resulting

Workload	# of files	# of threads	Avg. file size	R/W ratio	Description
Fileserver	12,800	128	10 MB	2:1	Data-intensive
Varmail	25,600	128	10 MB	1:1	Metadata-intensive
Videoserver	128	128	480 MB	9:1	Read-intensive

Table 3: Workload configurations for macro-benchmark.



(a) Fileserver. (b) Varmail. (c) Videoserver.

Figure 13: Throughput on macro-benchmark.

in approximately 28.5 GB of total writes. We then rerun the same workload as overwrites. As shown in the figures, DeLFS achieves up to 2.77× lower application-level execution time and delivers up to 2.32× higher device-level bandwidth than the compared systems. This is because DeLFS performs a decentralized GC procedure that localizes GC and reduces lock contention. Moreover, by improving both the write I/O path and the GC I/O path, DeLFS further reduces total application execution time compared with ScaleLFS. These results show that DeLFS improves both normal I/O and GC performance.

Write amplification factor: We quantify the effect of suboptimal GC victim selection by comparing the WAF of F2FS and DeLFS. In the above GC evaluation, the total WAF increases slightly by 1.2%, from 1.088 in F2FS to 1.101 in DeLFS. The resulting WAF of 1.101 is broadly consistent with the 13.7% average gap between application-level throughput (1.31 GB/s on average) in Figure 12d and device-level bandwidth (1.49 GB/s on average) in Figure 12e. Specifically, in the non-GC phase, both file systems exhibit a similar WAF of approximately 1.065, mainly due to metadata overhead. Meanwhile, the GC-induced WAF overhead is 0.036 in DeLFS and 0.023 in F2FS. However, this additional overhead contributes only 0.013 to the total WAF increase. Consequently, these results show that although DeLFS may make less-than-optimal victim-selection decisions, the impact on total WAF is minor.

5.3 Macro-benchmark

We use three filebench workloads as listed in Table 3.

Fileserver: Figure 13a shows the performance of the fileserver workload. DeLFS delivers 2.34×, 2.40×, 2.22×, and 2.43× higher performance than F2FS, MAX, ScaleLFS, and F2FSJ, respectively. Meanwhile, the existing and variant LFSs show similar performance, as in the micro-benchmark case. Since fileserver is a write and data-intensive workload, the LFSs are still constrained by bottlenecks in the write path. Consequently, this result demonstrates that DeLFS yields high performance gains under more realistic workload.

Varmail: Figure 13b shows the results for the varmail workload, where DeLFS achieves 1.66×, 1.86×, 1.18×, and 1.73× higher throughput than F2FS, MAX, ScaleLFS, and F2FSJ, respectively. In all systems except ScaleLFS and DeLFS, delete operations contend heavily on the SIT entry lock (sentry_lock). ScaleLFS alleviates this with atomic operations, while DeLFS removes this bottleneck via its decentralized architecture and locking. Thus, DeLFS is effective for both data-intensive and metadata-intensive workloads.

Videoserver: Figure 13c shows the results for the videoserver workload. In the figure, DeLFS delivers 1.31×, 1.55×, 1.36×, and 1.32× higher throughput than F2FS, MAX, ScaleLFS, and F2FSJ, respectively. Although videoserver is read-intensive, DeLFS still accelerates its execution because improvements in the write path can reduce interference between read and write operations. Consequently, DeLFS provides higher performance even for read-intensive workloads.

5.4 Real-world Application

To evaluate DeLFS with a real-world application, we run RocksDB with YCSB workloads A, B, F, and update-only [3]. Workload A uses a 1:1 read-to-update ratio, B uses a 9.5:0.5 read-to-update ratio, F uses a 1:1 read-to-read-modify-write ratio, and update-only issues 100% update operations. We use 1–128 client threads, 10 million records with 10 fields of 4 KB each, and 1.28 million operations per workload.

Workload A: Figure 14a shows that DeLFS achieves 1.78×, 1.60×, 1.81×, and 1.62× higher throughput than F2FS, MAX, ScaleLFS, and F2FSJ, respectively, at 128 cores. We observe that ScaleCache improves throughput by eliminating blocking overhead but the serialization-free scheme is ineffective because the next lock becomes the bottleneck. These results indicate that DeLFS is also effective for the real-world application.

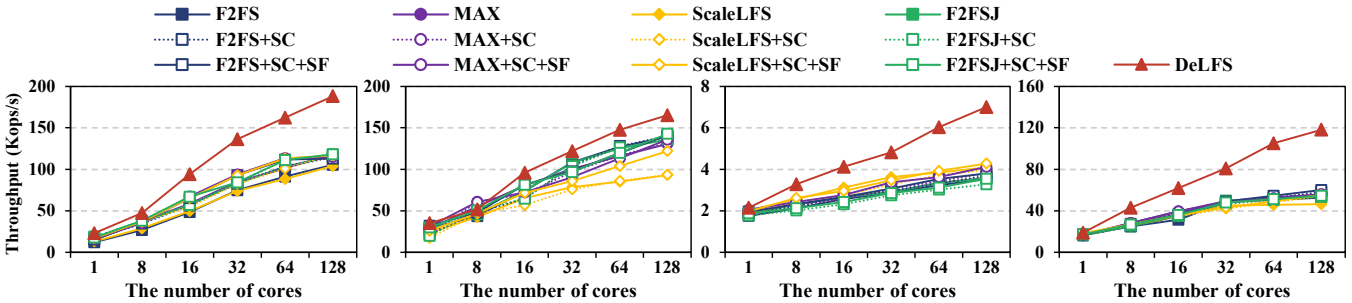
Workload B: As shown in Figure 14b, interestingly, even under the read-intensive workload, DeLFS improves performance by 1.19×, 1.26×, 1.77×, and 1.22× over F2FS, MAX, ScaleLFS, and F2FSJ, respectively, at 128 cores. These results indicate that an inefficient write path can significantly impact read performance even under small write workloads.

Workload F: Figure 14c indicates that DeLFS achieves up to 1.98×, 1.95×, 1.71×, and 1.96× over F2FS, MAX, ScaleLFS, and F2FSJ, respectively. Overall, DeLFS shows larger gains in workload F than in A because the read-modify-write requests in workload F stress both the write and read paths.

Update-only workload: Figure 14d shows DeLFS achieves up to 2.24×, 2.14×, 2.54×, and 2.20× higher throughput than F2FS, MAX, ScaleLFS, and F2FSJ, respectively. These results indicate that DeLFS is particularly effective for write-intensive workloads, the primary target of our design.

5.5 Performance Breakdown

Decentralized architecture: To illustrate the performance impact of DeLFS in more detail, we report the performance



(a) Workload A. (b) Workload B. (c) Workload F. (d) Update-only.

Figure 14: Throughput of various LFSs with YCSB workloads.

File system	Throughput	io_schedule_timeout()	writepages	curseg_mutex	io_rwsem	sentry_lock	nat_tree_lock	nat_list_lock	seglist_lock	cmd_lock	others
F2FS	1.05 GB/s	96.17%	0.8%	0.1%	0.9%	0.2%	0.1%	0.1%	0%	0.1%	1.53%
F2FS+SC	1.00 GB/s	0%	52%	0.1%	0.1%	0.2%	0.1%	0.2%	0%	0.2%	47.09%
F2FS+SC+SF	1.23 GB/s	0%	0%	47.95%	2.39%	0.7%	0.5%	0.1%	0.3%	0.7%	47.36%
De-curseg	0.51 GB/s	0%	0%	0.72%	14.36%	36.30%	0.23%	0.7%	0.1%	0.4%	47.19%
De-io	0.56 GB/s	0%	0%	0.73%	0.10%	49.80%	0.81%	0.62%	1.2%	0.40%	46.34%
De-sentry	1.49 GB/s	0%	0%	0.55%	0.11%	16.81%	5.7%	1.88%	13.17%	0.1%	61.68%
De-NAT	2.45 GB/s	0%	0%	0.77%	0.40%	25.19%	0.30%	0.12%	17.98%	0.3%	54.94%
De-LLC	3.04 GB/s	0%	0%	0.17%	0.31%	13.66%	0.7%	0.5%	70.80%	1.56%	12.3%
De-seglist	4.21 GB/s	0%	0%	0.38%	0.31%	0.33%	0.16%	0.17%	1.23%	21.37%	76.05%
De-discard	4.55 GB/s	0%	0%	0.26%	0.90%	0.29%	0.29%	0.14%	0.30%	0.65%	97.17%

Table 4: Breakdown of DeLFS (De: decentralized, SC: ScaleCache, SF: serialization-free, LLC: lazy lock coordinator).

breakdown as shown in Table 4. As shown in the table, the bottleneck shifts from one lock to another as each decentralized scheme is applied. Finally, DeLFS minimizes the contention of target locks holistically. These results demonstrate that each technique successfully resolves its corresponding scalability bottleneck in the LFS operation.

Decentralized locking schemes: We compare the performance of DFL, DDL, and deLLC in Table 5. For this experiment, we first fill the SSD to 1.8 TB using FIO. We then overwrite the data using the default FIO workload (281.6 GB) and measure performance. During this process, approximately 73 million invalid pages are processed. deLLC outperforms DFL and DDL by 89.63% and 43.83%, respectively. Moreover, the total delegation time is only 1.32s, which is a minor overhead.

5.6 Overhead of DeLFS

Performance under aged state: To measure the overheads of DeLFS under an aged state, we age the file system following the methodology of prior work [8], as shown in Table 6. Specifically, we perform 300 pulls from the Linux git repository and a recursive grep test, and then run the fileserver workload under the aged state. As expected, aging decreases grep performance in both file systems. Interestingly, DeLFS achieves higher grep performance than F2FS in both the unaged and aged states as the number of threads increases, because it localizes blocks into core-local segments and improves read coalescing. In the fileserver evaluation, similar to the grep results, aging affects both file systems. Nevertheless, even in the aged state, DeLFS still outperforms F2FS by up to 1.96×. Overall, aging degrades performance compared with the unaged state, but DeLFS maintains its performance advantage even in the aged state.

Skewed space allocation: We evaluate the impact of cross-core space allocation as local space is gradually exhausted, focusing on the performance overhead caused by lock contention on remote cores. To generate sustained low-free-space pressure, we pre-fill the device at different capacity levels and run the fileserver workload, which continuously performs file creation and deletion while generating substantial writes as shown in Table 7. For the 75%, 50%, 25%, and 10% available space cases, DeLFS outperforms F2FS by 2.37×, 1.93×, 1.63×, and 1.17×, respectively. DeLFS performance gradually decreases as the available space decreases because local free-pool depletion incurs cross-core overheads. Especially in the 1% available space case, both file systems show almost the

Metric \ Scheme	DFL	DDL	deLLC
Throughput	1303 MB/s	1718 MB/s	2471 MB/s
Total execution time	226 s	171 s	119 s
Total delegation count	-	-	71,377,706
Total delegation time	-	-	1.32 s

Table 5: Performance under heavy invalid page processing.

Workload	Threads	F2FS		DeLFS	
		Unaged	Aged	Unaged	Aged
Grep test	1	1.2 s	1.23 s	1.16 s	1.2 s
	32	3.54 s	3.63 s	2.76 s	2.84 s
	64	8.05 s	8.12 s	4.45 s	4.65 s
	128	17.24 s	17.59 s	9.63 s	10.11 s
Fileserver	1	1501 ops/s	1480 ops/s	1970 ops/s	1938 ops/s
	32	1549 ops/s	1530 ops/s	2307 ops/s	2304 ops/s
	64	1553 ops/s	1551 ops/s	2753 ops/s	2492 ops/s
	128	1559 ops/s	1557 ops/s	3469 ops/s	3047 ops/s

Table 6: Performance comparison between unaged and aged states.

LFS	Metric	Available space				
		1.5 TB (75%)	1 TB (50%)	0.5 TB (25%)	0.2 TB (10%)	0.02 TB (1%)
F2FS	# of create/delete	128 K	135 K	132 K	128 K	37 K
	Total writes	1.16 TB	1.19 TB	1.18 TB	1.14 TB	0.75 TB
	Throughput	1574 ops/s	1650 ops/s	1627 ops/s	1578 ops/s	458 ops/s
DeLFS	# of create/delete	305 K	260 K	217 K	151 K	38 K
	Total writes	3.02 TB	2.60 TB	2.24 TB	1.57 TB	0.81 TB
	Throughput	3730 ops/s	3180 ops/s	2662 ops/s	1848 ops/s	468 ops/s

Table 7: LFS performance under various available space conditions.

same performance because the available capacity is severely limited. Nevertheless, these results show that even under severe free-space pressure with continuous file creation/deletion and large writes, DeLFS remains effective and maintains a performance advantage over F2FS in most capacity settings.

Memory consumption: DeLFS requires additional memory to support decentralized organization and management, but the overhead is small. Across multi-head logging metadata, checkpoint bookkeeping (per-log segno/blkoff), decentralized locks and their protected data structures (e.g., XArrays), and lock-coordinator/invalidation structures, DeLFS uses up to about 1.1 MB (1121.67 KB) of extra memory under our experimental setup. Given modern memory capacities, this additional overhead is negligible in practice.

Storage overhead: As the number of multi-head logging instances increases with the number of cores, the checkpointing process requires an additional 11.9 KB of space to store the segment number and block offset information maintained for each multi-head logging instance.

5.7 Crash Consistency

To validate crash consistency of DeLFS, we use CrashMonkey [25]. Specifically, we test DeLFS with checkpoint (`checkpoint_example`) and create/delete (`create_delete`) workloads. The results confirm that DeLFS successfully handles all test cases and maintains crash consistency.

6 Related Work

Scaling file and storage systems: SpanFS [13] scales file systems by composing a set of micro file-system services. ScaleFS [6] decouples in-memory and on-disk file systems to reduce lock contention. CJFS [28] and Z-Journal [14] improve the scalability of journaling file systems using concurrent and per-core schemes. uFS [22] is a semi-microkernel file system that scales through partitioned data structures and inode migration. ScaleCache [29] is a scalable page cache that provides concurrent XArray and direct page flushing. Our work aligns with these studies on scaling file and storage systems [6, 13, 14, 22, 28, 29]. However, unlike these studies, DeLFS targets LFSs and introduces a decentralized management and locking approach that eliminates centralized bottlenecks while accelerating both normal I/O and GC I/O paths.

Scalable LFSs: MAX [21] scales in-memory data-structure access and provides a concurrency-friendly on-disk format, but still limits parallelism. During flushing, MAX selects an mlog cell in a round-robin manner, searches a shared free bitmap, and sends data to the corresponding mlog. This one-to-all allocation model and shared metadata still incur high lock contention. F2FSJ [9] introduces an ordered-journal mode for F2FS to reduce checkpointing time by journaling metadata. Unlike F2FSJ, DeLFS targets data-intensive workloads where checkpoint overhead is relatively small; integrating F2FSJ-like journaling into DeLFS for metadata-intensive workloads remains future work. ScaleLFS [11] scales garbage collection with parallel GC and concurrent victim selection. In contrast, DeLFS decentralizes and localizes GC while reducing lock contention on the normal I/O path. As a result, DeLFS further improves sustained performance. In summary, our work is inspired by these designs and shares the goal of improving scalability, but DeLFS holistically decentralizes metadata and data management and employs decentralized, decoupled locking to scale on many-core systems.

7 Conclusion

We present DeLFS, a decentralized log-structured file system that overcomes the scalability limitations of existing LFSs and scales I/O performance on manycores. DeLFS holistically decentralizes metadata/data organization and management into per-core domains with a novel decentralized locking scheme. Furthermore, DeLFS disentangles the critical path from the deferrable path to further increase concurrency. Our evaluations show that DeLFS achieves up to 4.34×, 4.29×, 4.10×, and 4.50× higher throughput compared with F2FS, MAX, ScaleLFS, and F2FSJ, respectively.

Acknowledgments

We sincerely thank our shepherd, Emmett Witchel, and the anonymous reviewers for their invaluable feedback. This work was supported by the National Research Foundation of Korea (NRF) (No. RS-2025-00554650) (Corresponding Author: Yongseok Son).

References

[1] Yahoo! cloud serving benchmark, 2019. Accessed on May 17, 2025. URL: https://github.com/brianfrankcooper/YCSB.

[2] Seagate FireCuda 530, 2022. Accessed on Aug. 8, 2025. URL: https://www.amazon.com/Seagate-FireCuda-Internal-Solid-State/dp/B0977K2C74.

[3] Raja Appuswamy, Angelos C. Anadiotis, Danica Porobic, Mustafa K. Iman, and Anastasia Ailamaki. Analyzing the impact of system architecture on the scalability of oltp engines for high-contention workloads. Proc. VLDB Endow., 11(2):121–134, oct 2017. doi:10.14778/3149193.3149194.

[4] Jens Axboe. Flexible I/O Tester, 2024. Accessed on Jan. 02, 2024. URL: https://github.com/axboe/fio.

[5] Philip A. Bernstein, Vassos Hadzilacos, and Nathan Goodman. Concurrency Control and Recovery in Database Systems. Addison-Wesley, 1987.

[6] Srivatsa S. Bhat, Rasha Eqbal, Austin T. Clements, M. Frans Kaashoek, and Nickolai Zeldovich. Scaling a file system to many cores using an operation log. In Proceedings of the 26th Symposium on Operating

Systems Principles, SOSP '17, page 69–86, New York, NY, USA, 2017. Association for Computing Machinery. doi:10.1145/3132747.3132779.

[7] Matias Bjørling, Jens Axboe, David Nellans, and Philippe Bonnet. Linux block io: introducing multi-queue ssd access on multi-core systems. In Proceedings of the 6th International Systems and Storage Conference, SYSTOR '13, New York, NY, USA, 2013. Association for Computing Machinery. doi:10.1145/2485732.2485740.

[8] Alex Conway, Ainesh Bakshi, Yizheng Jiao, William Jannen, Yang Zhan, Jun Yuan, Michael A. Bender, Rob Johnson, Bradley C. Kuszmaul, Donald E. Porter, and Martin Farach-Colton. File systems fated for senescence? nonsense, says science! In 15th USENIX Conference on File and Storage Technologies (FAST 17), pages 45–58, Santa Clara, CA, February 2017. USENIX Association. URL: https://www.usenix.org/conference/fast17/technical-sessions/presentation/conway.

[9] Yaotian Cui, Zhiqi Wang, Renhai Chen, and Zili Shao. Decentralized, epoch-based F2FS journaling with fine-grained crash recovery. In Proceedings of the 19th USENIX Conference on Operating Systems Design and Implementation, OSDI '25, USA, 2025. USENIX Association.

[10] Facebook. RocksDB: A Persistent Key-Value Store for Fast Storage Environments, 2012. Accessed on Dec. 20, 2024. URL: https://rocksdb.org/.

[11] Jin Yong Ha, Sangjin Lee, Hyeonsang Eom, and Yongseok Son. ScaleLFS: A Log-Structured File System with Scalable Garbage Collection for Commodity SSDs. In 23rd USENIX Conference on File and Storage Technologies (FAST 25), pages 53–67, Santa Clara, CA, February 2025. USENIX Association. URL: https://www.usenix.org/conference/fast25/presentation/ha.

[12] Inhwi Hwang, Sangjin Lee, Sunggon Kim, Hyeonsang Eom, and Yongseok Son. Z-LFS: A Zoned Namespace-tailored Log-structured File System for Commodity Small-zone ZNS SSDs. In 2025 USENIX Annual Technical Conference (USENIX ATC 25), pages 547–562, 2025.

[13] Junbin Kang, Benlong Zhang, Tianyu Wo, Weiren Yu, Lian Du, Shuai Ma, and Jinpeng Huai. SpanFS: A Scalable File System on Fast Storage Devices. In 2015 USENIX Annual Technical Conference (USENIX ATC 15), pages 249–261, Santa Clara, CA, July 2015. USENIX Association. URL: https://www.usenix.org/conference/atc15/technical-session/presentation/kang.

[14] Jongseok Kim, Cassiano Campes, Joo-Young Hwang, Jinkyu Jeong, and Euiseong Seo. Z-Journal: Scalable Per-Core journaling. In 2021 USENIX Annual Technical Conference (USENIX ATC 21), pages 893–906. USENIX Association, July 2021. URL: https://www.usenix.org/conference/atc21/presentation/kim-jongseok.

[15] Juwon Kim, Minsu Kim, Muhammad Danish Tehseen, Joontaek Oh, and Youjip Won. IPLFS: Log-Structured file system without garbage collection. In 2022 USENIX Annual Technical Conference (USENIX ATC 22), pages 739–754, Carlsbad, CA, July 2022. USENIX Association. URL: https://www.usenix.org/conference/atc22/presentation/kim-juwon.

[16] Ryusuke Konishi, Yoshiji Amagai, Koji Sato, Hisashi Hifumi, Seiji Kihara, and Satoshi Moriai. The Linux implementation of a log-structured file system. SIGOPS Oper. Syst. Rev., 40(3):102–107, jul 2006. doi:10.1145/1151374.1151375.

[17] Chang-Gyu Lee, Hyunki Byun, Sunghyun Noh, Hyeongu Kang, and Youngjae Kim. Write optimization of log-structured flash file system for parallel i/o on manycore servers. In Proceedings of the 12th ACM International Conference on Systems and Storage, SYSTOR '19, page 21–32, New York, NY, USA, 2019. Association for Computing Machinery. doi:10.1145/3319647.3325828.

[18] Changman Lee, Dongho Sim, Jooyoung Hwang, and Sangyeun Cho. F2FS: A New File System for Flash Storage. In 13th USENIX Conference on File and Storage Technologies (FAST 15), pages 273–286, Santa Clara, CA, February 2015. USENIX Association. URL: https://www.usenix.org/conference/fast15/technical-sessions/presentation/lee.

[19] Euidong Lee, Ikjoon Son, and Jin-Soo Kim. An Efficient Order-Preserving Recovery for F2FS with ZNS SSD. In Proceedings of the 15th ACM Workshop on Hot Topics in Storage and File Systems, HotStorage '23, page 116–122, New York, NY, USA, 2023. Association for Computing Machinery. doi:10.1145/3599691.3603416.

[20] Yongmyung Lee, Jong-Hyeok Park, Jonggyu Park, Hyunho Gwak, Dongkun Shin, Young Ik Eom, and Sang-Won Lee. When F2FS meets address remapping. In Proceedings of the 14th ACM Workshop on Hot Topics in Storage and File Systems, HotStorage '22, page 31–36, New York, NY, USA, 2022. Association for Computing Machinery. doi:10.1145/3538643.3539755.

[21] Xiaojian Liao, Youyou Lu, Erci Xu, and Jiwu Shu. Max: A Multicore-Accelerated File System for Flash Storage. In 2021 USENIX Annual Technical Conference (USENIX ATC 21), pages 877–891. USENIX Association, July 2021. URL: https://www.usenix.org/conference/atc21/presentation/liao.

[22] Jing Liu, Anthony Rebello, Yifan Dai, Chenhao Ye, Sudarsun Kannan, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Scale and performance in a filesystem semi-microkernel. In Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles, SOSP '21, page 819–835, New York, NY, USA, 2021. Association for Computing Machinery. doi:10.1145/3477132.3483581.

[23] Changwoo Min, Sanidhya Kashyap, Steffen Maass, and Taesoo Kim. Understanding manycore scalability of file systems. In 2016 USENIX Annual Technical Conference (USENIX ATC 16), pages 71–85, Denver, CO, June 2016. USENIX Association. URL: https://www.usenix.org/conference/atc16/technical-sessions/presentation/min.

[24] Changwoo Min, Kangnyeon Kim, Hyunjin Cho, Sang-Won Lee, and Young Ik Eom. SFS: random write considered harmful in solid state drives. In Proceedings of the 10th USENIX Conference on File and Storage Technologies, FAST'12, page 12, USA, 2012. USENIX Association. URL: https://dl.acm.org/doi/abs/10.5555/2208461.2208473.

[25] Jayashree Mohan, Ashlie Martinez, Soujanya Ponnapalli, Pandian Raju, and Vijay Chidambaram. Finding Crash-Consistency bugs with bounded Black-Box crash testing. In 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18), pages 33–50, Carlsbad, CA, October 2018. USENIX Association. URL: https://www.usenix.org/conference/osdi18/presentation/mohan.

[26] Dushyanth Narayanan, Eno Thereska, Austin Donnelly, Sameh Elnikety, and Antony Rowstron. Migrating server storage to ssds: analysis of tradeoffs. In Proceedings of the 4th ACM European conference on Computer systems, pages 145–158, 2009.

[27] Joontaek Oh, Sion Ji, Yongjin Kim, and Youjip Won. exF2FS: Transaction Support in Log-Structured Filesystem. In 20th USENIX Conference on File and Storage Technologies (FAST 22), pages 345–362, Santa Clara, CA, February 2022. USENIX Association. URL: https://www.usenix.org/conference/fast22/presentation/oh.

[28] Joontaek Oh, Seung Won Yoo, Hojin Nam, Changwoo Min, and Youjip Won. CJFS: Concurrent Journaling for Better Scalability. In 21st USENIX Conference on File and Storage Technologies (FAST 23), pages 167–182, Santa Clara, CA, February 2023. USENIX Association. URL: https://www.usenix.org/conference/fast23/presentation/oh.

[29] Kiet Tuan Pham, Seokjoo Cho, Sangjin Lee, Lan Anh Nguyen, Hyeongi Yeo, Ipoom Jeong, Sungjin Lee, Nam Sung Kim, and Yongseok Son. ScaleCache: A Scalable Page Cache for Multiple Solid-State Drives. In Proceedings of the Nineteenth European Conference on Computer Systems, EuroSys'24, page 641–656, New York, NY, USA, 2024. Association for Computing Machinery. doi:10.1145/3627703.3629588.

[30] Mendel Rosenblum and John K. Ousterhout. The design and implementation of a log-structured file system. ACM Trans. Comput. Syst., 10(1):26–52, feb 1992. doi:10.1145/146941.146943.

[31] Vasily Tarasov, Erez Zadok, and Spencer Shepler. Filebench: A flexible framework for file system benchmarking. USENIX; login, 41(1):6–12, 2016. URL: https://www.usenix.org/system/files/login/articles/login_spring16_02_tarasov.pdf.

[32] Gerhard Weikum and Gottfried Vossen. Transactional information systems: theory, algorithms, and the practice of concurrency control and recovery. Elsevier, 2001.

[33] Lihua Yang, Fang Wang, Zhipeng Tan, Dan Feng, Jiaxing Qian, and Shiyun Tu. ARS: Reducing F2FS Fragmentation for Smartphones using Decision Trees. In 2020 Design, Automation & Test in Europe Conference & Exhibition (DATE), pages 1061–1066, 2020. doi:10.23919/DATE48585.2020.9116318.

[34] Jiacheng Zhang, Jiwu Shu, and Youyou Lu. ParaFS: a log-structured file system to exploit the internal parallelism of flash devices. In Proceedings of the 2016 USENIX Conference on Usenix Annual Technical Conference, USENIX ATC '16, page 87–100, USA, 2016. USENIX Association. URL: https://dl.acm.org/doi/abs/10.5555/3026959.3026968.

ARTIFACT EVALUATED
USENIX
AVAILABLE

WEAVE: Efficient Co-Scheduling for Disaggregated RL Post-Training

Tianyuan Wu[†], *Lunxi Cao*[†], *Yining Wei*[‡], *Wei Gao*[†], *Yuheng Zhao*[†], *Dakai An*[†], *Shaopan Xiong*[§], *Zhiqiang Lv*[§], *Ju Huang*[§], *Siran Yang*[§], *Yinghao Yu*[§], *Jiamang Wang*[§], *Lin Qu*[§], *Wei Wang*[†]
[†]*Hong Kong University of Science and Technology,* [‡]*UIUC,* [§]*Alibaba Group*

Abstract

Rollout–training disaggregation is emerging as the standard architecture for Reinforcement Learning (RL) post-training, where memory-bound rollout and compute-bound training are physically disaggregated onto purpose-built clusters to maximize hardware efficiency. However, the strict synchronization required by on-policy algorithms introduces severe *dependency bubbles*, forcing one cluster to idle while the dependent phase is running on the other. We present WEAVE, a cluster scheduling framework that reclaims these bubbles through cross-cluster orchestration. WEAVE is built on the insight that the structural idleness of one job can be effectively utilized by the active phase of another. To realize this, we introduce the *co-execution group* abstraction, which partitions the cluster into isolated locality domains. This abstraction enables a *two-tier scheduling architecture*: an *inter-group scheduler* that optimizes job placement using conservative stochastic planning, and an *intra-group scheduler* that orchestrates a provably optimal round-robin schedule. The group abstraction also imposes a *residency constraint*, ensuring that massive model states remain cached in host memory to enable "warm-start" context switching. We evaluate WEAVE on a production-scale testbed with 328 H20 and 328 H800 GPUs. WEAVE improves cost efficiency by 1.84× over standard disaggregation and 1.38× over state-of-the-art co-located baselines, all while achieving 100% SLO attainment.

1 Introduction

The development of Large Language Model (LLM) is shifting from pre-training to *Reinforcement Learning (RL) post-training* [1,3,7], which unlocks reasoning capabilities in complex domains such as mathematics [2], coding [33], and tool use [9,54]. To achieve optimal performance and model stability, production deployments have converged on *synchronous, on-policy algorithms* [7,37,41] that prescribe a strict, iterative learning loop through three phases, each with a distinct resource bottleneck: (1) **rollout**, a *memory-bandwidth-bound* inference stage that generates token trajectories; (2) **training**, a *compute-intensive* stage that updates parameters from rewards; and (3) **synchronization**, a *network-bound* stage that propagates updated weights back to inference workers.

This paper targets the deployment that production on-policy RL post-training has converged on—a *disaggregated architecture* [11, 47, 52, 61] that splits rollout and training across two purpose-built clusters (Figure 1-top). The *rollout pool* consists of cost-effective, inference-optimized GPUs (e.g., NVIDIA H20), while the *training pool* uses high-performance, compute-optimized GPUs (e.g., NVIDIA H800). By matching hardware to phase characteristics, disaggregation resolves the resource mismatches that plague monolithic provisioning, achieving superior cost efficiency at comparable throughput despite the added cross-cluster synchronization.

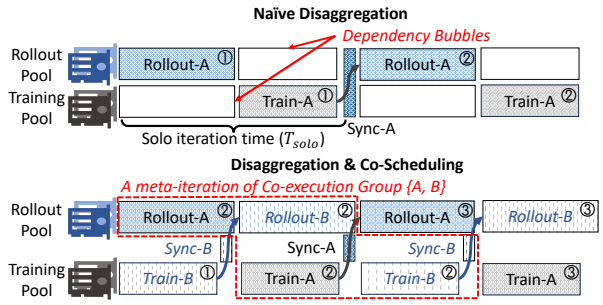


Figure 1: Comparison between existing disaggregated RL architecture and WEAVE's co-scheduling paradigm.

Disaggregation, however, introduces a fundamental efficiency challenge: *dependency bubbles*. The strict synchronization required by on-policy learning forces the training cluster to idle during rollout, and vice versa (Figure 1-top), wasting both pools. Existing systems such as AReaL [11], StreamRL [61], and AsyncFlow [18] eliminate these bubbles only by switching to *asynchronous, off-policy algorithms*, trading away the on-policy guarantee and admitting *sample staleness* that compromises model accuracy and convergence stability—an unacceptable trade for tasks that demand strict on-policy performance.

We present WEAVE, a cross-cluster scheduling framework for disaggregated RL post-training that mitigates dependency bubbles by lifting the optimization scope from the individual job to *cluster-level orchestration*. Our key insight is that the dependency bubbles inherent to one job can be filled by another job's active phase. WEAVE groups multiple jobs into a *co-execution group* and tightly weaves their rollout and training phases across the two resource pools (Figure 1-bottom), time-multiplexing both pools to maximize utilization while preserving the synchronous dependencies required by on-policy learning.

However, realizing this benefit in production is non-trivial

because of three challenges. (**C1**) Production RL jobs exhibit *extreme workload heterogeneity* across model sizes, response lengths, and interaction patterns (Figure 2), producing widely varying phase durations and resource demands. Naive time-multiplexing causes severe interference—for example, pairing two rollout-heavy jobs creates a bottleneck that substantially delays both (Figure 3). Finding an interference-free schedule across such heterogeneous workloads reduces to a Job-Shop Scheduling problem [25], which is NP-hard. (**C2**) Unlike standard deep-learning workloads with stable iteration times, RL rollouts are *highly stochastic*: LLM generation follows a long-tailed distribution [14, 19, 62] in which a few straggler requests *unpredictably prolong* phase durations, rendering static plans obsolete. (**C3**) Time-multiplexing is constrained by switching costs. RL post-training is inherently *stateful*, requiring each worker to manage hundreds of gigabytes of model weights and optimizer state per job (Table 2); reloading these states from disk or across the bandwidth-limited cross-cluster network on every switch incurs *cold-start latencies of up to 135 s* (Figure 4)—enough to wipe out any gains from co-scheduling. This combination—stateful workers holding hundreds of gigabytes of resident state, a strict cross-phase synchronization barrier, and long-tailed phase durations—is what makes generic VM, container, or GPU multiplexing inadequate here and motivates an RL-specific scheduler.

WEAVE addresses these challenges via a holistic algorithm–system co-design built around a near-optimal scheduling algorithm. The objective is to minimize total resource provisioning cost—and thus dependency bubbles—while honoring per-job SLOs, defined as the acceptable slowdown relative to *solo execution* (Figure 1-top). Although the global problem is NP-hard (**C1**), WEAVE makes it tractable by decomposing it into two decisions: *inter-group scheduling* chooses the jobs that co-execute, and *intra-group scheduling* orchestrates their execution sequences within a co-execution group. When a new job arrives, the inter-group scheduler scans existing co-execution groups for a placement that violates no group member's SLO; among SLO-compliant options, it picks the one with the minimum marginal provisioning cost, and otherwise provisions a new, isolated group. Within each group, the intra-group scheduler runs a round-robin schedule, which we prove is optimal for minimizing dependency bubbles in this context (§4.3).

To handle runtime stochasticity (**C2**), WEAVE pairs *conservative admission control* with *long-tail migration*. For inter-group placement, the scheduler assumes a worst case in which every response reaches the maximum token length, so SLO guarantees hold even under peak load (§4.2). At runtime, the intra-group scheduler adapts to the observed response distribution: it opportunistically migrates ongoing long-tail responses onto a small subset of GPU devices, freeing the bulk of the rollout pool so that the next job can begin pipelined execution immediately (§4.3).

To absorb switching overheads (**C3**), WEAVE introduces a *warm-start* mechanism that cuts switching latency by two orders of magnitude (Figure 4), making fine-grained time-multiplexing practical. WEAVE rightsizes each co-execution group so that the states of all member jobs—model weights, optimizer state, and execution context—fit in each worker's host memory; on a switch, the worker loads the cached state from host DRAM to the GPU rather than fetching it across the slow inter-cluster link or from disk.

To make these schedules executable, WEAVE introduces two system mechanisms. A *phase-centric control model* treats each RL phase as a first-class schedulable entity, exposing the job's internal dependency graph to the scheduler and transparently driving the state loads that warm-start requires (§5.1). A *topology-aware* model-synchronization scheme prevents the cross-cluster sync barrier from becoming a bubble multiplier (§5.2): it pipes a single model copy across the slow inter-cluster link via parallel point-to-point streams and then broadcasts within each cluster over the high-speed local fabric (e.g., NVLink, InfiniBand).

We implemented WEAVE atop ROLL [45] and evaluated it on a production-scale disaggregated testbed with a 328-GPU H20 rollout pool and a 328-GPU H800 training pool. Across micro-benchmarks at identical per-hour provisioning cost, WEAVE delivers 1.82–1.99× higher training throughput than disaggregation without co-scheduling (§6.2). On an end-to-end replay of a two-week production trace, WEAVE sustains the same workload at 1.38× lower provisioning cost than the state-of-the-art veRL [43] and 1.84× lower than naive disaggregation, while maintaining 100% SLO attainment (§6.4). Large-scale trace-driven simulations further show that WEAVE's combined inter- and intra-group scheduling runs within 6% of the theoretical optimum found by brute-force search (§6.5). Finally, WEAVE is robust to workload drift—static placement stays within 1.11× of an optimal dynamic-regrouping baseline—and sustains job failures by re-admitting recovered jobs as new arrivals (§6.6).

2 Background and Motivation

RL Post-Training Workload Characterization. RL post-training has become a cornerstone workload for modern AI infrastructure. In our production clusters, the volume of RL jobs nearly tripled within six months, growing from 5k monthly jobs in April to over 14k in September 2025, driven largely by the need to instill complex reasoning capabilities in LLMs [3, 7, 9, 41, 45]. Unlike LLM pre-training, which is a uniform stream of compute, the standard RL post-training workflow consists of repeated cycles over three phases, each with a distinct resource bottleneck. (1) **Rollout:** the actor LLM generates responses for a batch of input prompts, which are then evaluated to collect reward feedback. This phase exerts *high memory-bandwidth pressure* from KV-cache operations while having relatively *low arithmetic intensity*, so running it on high-end training GPUs severely underutilizes

their compute [11, 19, 61]. (2) **Training:** the actor LLM's parameters are optimized from the reward feedback. This phase is *compute-intensive* and demands both massive floating-point throughput and high-bandwidth interconnects (e.g., NVLink, InfiniBand) for gradient aggregation [44]. (3) **Synchronization:** updated parameters must be broadcast from the training workers to the rollout workers. Synchronization exists in any deployment, but disaggregating rollout and training across separate clusters forces the broadcast to traverse a comparatively slow inter-cluster link, turning this phase into a *network-bound* bottleneck.

The Case for Disaggregated RL. The divergent resource requirements of rollout and training create a fundamental inefficiency in traditional *monolithic, co-located deployments* [43], in which rollout and training are time-multiplexed on a single cluster of homogeneous, compute-optimized GPUs (e.g., NVIDIA H100/H800). This forces the memory-bandwidth-bound rollout phase to run on expensive compute-optimized hardware, inflating the total cost of ownership (TCO) [52, 61].

Disaggregation addresses this mismatch by splitting the RL workload across two purpose-built resource pools (Figure 1-bottom) [11,47,52,61]: training runs on costly, compute-optimized GPUs (e.g., H100/H800), while rollout is offloaded to cost-effective, inference-optimized GPUs (e.g., H20) that offer high HBM capacity and bandwidth at a fraction of the cost (Table 1). By aligning hardware with phase characteristics, disaggregation promises a superior TCO *in theory*.

Dependency Bubbles. While disaggregation addresses hardware mismatches, *dependency bubbles* undermine its efficiency in practice. State-of-the-art RL post-training relies on *synchronous, on-policy* algorithms to ensure training stability and model quality [7, 17, 37, 41]. This synchronization constraint enforces a strict phase dependency: the training pool sits idle while the rollout pool generates fresh experiences, and the rollout pool sits idle while the training pool updates parameters (Figure 1-top).

Existing systems such as AReaL [11, 52], StreamRL [61], and AsyncFlow [18] eliminate these bubbles by switching to *asynchronous, off-policy* algorithms. Decoupling rollout from training, however, introduces *sample staleness* that degrades model convergence and final accuracy, making these solutions unsuitable for tasks that demand strict on-policy performance. As a result, production deployments retain synchronous execution and absorb the resulting idleness.

Worse, this idleness can fully offset disaggregation's hardware savings. Our end-to-end evaluation (§6.4) shows that a naive disaggregated deployment (Solo-D) incurs *higher* provisioning cost than a monolithic co-located baseline (veRL)—$0.94k/h versus $0.71k/h—despite using cheaper GPUs for rollout. Without a scheduling mechanism to reclaim this lost capacity, disaggregation's theoretical TCO advantage is nullified by system-level inefficiencies.

Accelerator	Comp. (TFLOPS)	HBM Cap. (GB)	HBM B/w (TB/s)	Cost ($/h) [64]
H20	148	96	4.0	1.85
H800	989.5	80	3.35	5.28

Table 1: Performance specifications and cost-effectiveness of the GPUs used in our disaggregated clusters [61, 64].

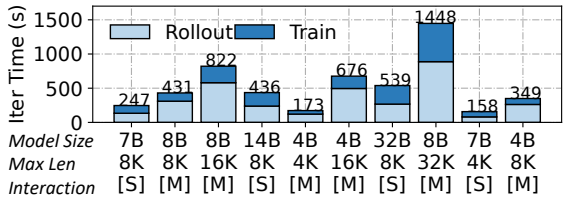


Figure 2: Top 10 popular RL post-training workloads in our production cluster: jobs' phase durations are highly diverse. [S], [M] refers to single/multi-turn interaction during rollout.

3 Scheduling Opportunities and Challenges

In this section, we identify a cluster-level opportunity for mitigating the dependency bubbles of disaggregated RL post-training. We then examine the algorithmic and system challenges of realizing it.

3.1 The Co-Scheduling Opportunity

For a single job, the dependency bubbles described in §2 are unavoidable: eliminating them would violate the synchronization requirements of on-policy algorithms. In *shared, multi-tenant* clusters that run diverse RL workloads, however, these per-job inefficiencies are aggregate capacity that can be reclaimed through cluster-level scheduling.

Our key insight is that the idle resources in one job's dependency bubbles can be utilized to execute the active phase of another. By orchestrating jobs into *co-execution groups*, the scheduler can tightly "weave" together their workflows, ensuring that the compute-intensive training phase of one job executes in parallel with the memory-bound rollout phase of another (Figure 1-bottom). This interleaved execution pattern effectively hides dependency bubbles, allowing the system to simultaneously saturate both the cost-effective rollout pool and the high-performance training pool, maximizing cluster-wide efficiency without compromising the synchronization requirements of on-policy learning.

3.2 Challenges

However, realizing co-scheduling at production scale is non-trivial for three reasons: workload heterogeneity (**C1**), runtime stochasticity (**C2**), and context-switching cost under memory-residency limits (**C3**).

C1: Workload Heterogeneity and Scheduling Intractability. First, production RL workloads are highly diverse, which

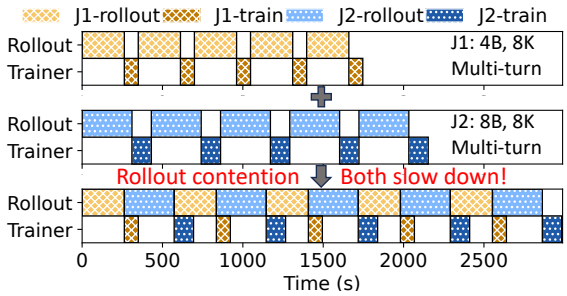


Figure 3: A bad case of naive time-multiplexing: two rollout-heavy jobs compete for a rollout node and both slow down.

complicates efficient co-scheduling. As shown in Figure 2, jobs in our cluster span a wide range of model sizes (3B–32B), response lengths (4k–32k tokens), and interaction modes (single- vs. multi-turn). This diversity translates into phase durations that range from 50s to over 900s and into pronounced *phase skew*: multi-turn agentic workloads, for instance, exhibit rollout phases 3×–4× longer than their training phases.

Naive time-multiplexing therefore often proves detrimental due to resource contention. Arbitrarily pairing two rollout-heavy jobs, for example, creates a bottleneck on the inference nodes, forcing both jobs to stall. As shown in Figure 3, such contention results in severe performance degradation, slowing down concurrent jobs by 1.40× and 1.64×, respectively. To avoid such interference, the scheduler must identify optimal packings that satisfy strict performance SLOs. However, mapping these heterogeneous, phase-skewed workloads to available resources reduces to a Job Shop Scheduling problem [15, 25], which is known to be NP-hard even under the simplifying assumption of deterministic phase durations.

C2: Stochastic Runtime and Skewness Bubbles. Second, RL workloads are inherently *stochastic*. Unlike pre-training, an RL rollout's execution time depends on the generated response lengths, which follow a *long-tail distribution* (Figure 12). This creates two challenges. First, the workload is *non-stationary*: response-length distributions drift across iterations as the model updates, and a few "straggler" requests routinely hit the maximum token limit [14, 61, 62]. Because training cost scales linearly with the number of generated tokens, rollout variance propagates into the subsequent training phase and makes static orchestration plans obsolete.

Second, long-tail responses induce *skewness bubbles* within a rollout [61, 62]: early-finishing GPUs idle while waiting for stragglers, serializing batch completion. The scheduler therefore faces a dynamic variant of the Job Shop Scheduling problem, in which task durations are unpredictable and time-varying, and execution is subject to significant intra-phase fragmentation.

C3: Context Switching Overhead and Memory Residency. Third, the granularity of time-multiplexing is fundamentally constrained by context-switching cost. Unlike stateless LLM inference [12], RL post-training is *stateful*: it maintains a large working set of model weights, optimizer states, dataset pipelines, and environment states. Reconstructing this state on every switch—known as a *cold start*—incurs prohibitive overheads from both data- and control-plane re-initialization. As shown in Figure 4, a cold start of a rollout or training phase on an 8-GPU node (H20 for rollout, H800 for training) takes up to 135 seconds and degrades end-to-end RL training throughput by up to 45%.

Size	7B	14B	32B	70B
Rollout	275.7	445.4	490.3(TP2)	391.2(TP8)
Train	240(TP2)	456.1(TP2)	520.4(TP8)	458.7(TP8PP4)

Table 2: Memory footprint (GB) required for caching rollout or training actors on an 8-GPU node across model sizes.

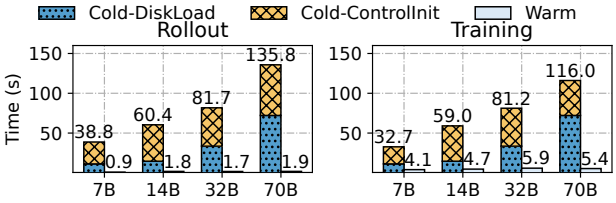


Figure 4: Cold and warm start latency for rollout (**left**) and training (**right**) across model sizes on an 8-GPU node (H20 for rollout, H800 for training).

Unlike serverless systems that mitigate cold starts via high-speed RDMA state transfers [12, 56, 58], disaggregated RL setups are bottlenecked by limited cross-cluster Ethernet bandwidth. The only viable alternative is a *warm-start* strategy that keeps job states cached in local host DRAM. This reduces switching latency by up to 71.5× (Figure 4), but at the cost of severe memory pressure. A single phase's state consumes hundreds of gigabytes (Table 2), so even high-memory nodes (1–2 TB) admit only two to five resident jobs at a time. This per-node footprint is determined by the per-GPU state held on the node, not by the model's total parameter count: larger models use higher tensor or pipeline parallelism to fit GPU HBM and therefore spread state across more GPUs, while smaller models more often replicate state per node via data parallelism. Table 2 confirms this: the 70B configuration (TP8PP4) fits in no more host DRAM than the 32B configuration, so scaling model size does not further tighten the residency budget. The scheduler therefore faces a tight *residency constraint* and must optimize utilization within a bounded memory budget.

4 The WEAVE Scheduling Design

We present WEAVE, a cluster scheduling framework that reclaims dependency bubbles for disaggregated RL post-training through cross-cluster orchestration. WEAVE adopts a holistic algorithm-system co-design: we decouple the logical scheduling policy (§4) from the underlying system design (§5). This section details the core scheduling algorithms.

4.1 Co-Execution Group

WEAVE's scheduling objective is to minimize provisioning cost by re-using dependency bubbles in already-paid-for capacity, while strictly adhering to job performance SLOs and node memory constraints. To achieve this, we introduce the *co-execution group* abstraction. A co-execution group is a set of jobs that *share* a specific pair of rollout and training resource pools via *time-multiplexing*. Within a group, all rollout phases execute on the group's assigned rollout workers, and all training phases execute on the group's training workers. By partitioning jobs into *disjoint groups*, WEAVE transforms the intractable global co-scheduling problem into a collection of independent, parallel sub-problems within groups. This decomposition keeps scheduling tractable at production scale (**C1**), and pinning jobs to specific group nodes keeps their working sets resident in host DRAM so that context switches are served by warm starts rather than slow disk or cross-cluster fetches (**C3**).

The co-execution group abstraction naturally leads to a *two-tier scheduling hierarchy*: (1) *inter-group scheduling* (§4.2), which assigns arriving jobs to groups to minimize provisioning costs while satisfying memory and SLO constraints, and (2) *intra-group scheduling* (§4.3), which orchestrates the runtime execution order of job phases within a group to minimize dependency bubbles.

4.2 Inter-Group Scheduling

Problem Formulation. We model the cluster as a collection of disjoint co-execution groups. We define a co-execution group G as a tuple (J_G, R_G, T_G, Φ_G), where J_G is the set of active RL jobs in the group, R_G and T_G denote the sets of rollout (e.g., H20) and training (e.g., H800) GPUs provisioned for the group, and $\Phi_G = \{P_j\}_{j \in J_G}$ is the set of resource placements, where P_j specifies the exact subset of rollout and training nodes job j is pinned to. This pinning P_j strictly determines where the job's state is cached to enable its warm start.

The inter-group scheduler solves the following *online placement problem*: upon the arrival of a job j, it must assign j to a co-execution group—either an existing one or a newly created one—and allocates specific resource placement P_j. To make optimal placement, we define the provisioning cost of a group, $\text{Cost}(G)$, as the aggregate hourly cost (Table 1) of all allocated GPUs in its rollout and training pools (R_G and T_G). The scheduler's objective is to minimize the *marginal provisioning cost* Δ incurred by admitting job j:

$$\min_G \Delta = \text{Cost}(G') - \text{Cost}(G),$$

where G' represents the group's state after accommodating job j. This formulation naturally encourages "packing" jobs into existing dependency bubbles (where $\Delta = 0$) over provisioning new hardware (where $\Delta > 0$). The placement decision is subject to two critical constraints:

1) Memory Residency. To guarantee warm starts (**C3**), the aggregate working set of all jobs pinned to a specific node must not exceed that node's host memory capacity.

2) SLO Attainment. The placement must satisfy the performance SLOs of both the new job and all existing jobs. The SLO is defined by each job as the *tolerance for co-execution slowdown* (e.g., 1.1×) relative to solo execution.[1] Formally, for every job k in the updated group G, we require:

$$T_k^{\text{co-exec}} \leq \text{SLO}_k \times T_k^{\text{solo}}.$$

Here, T_k^{solo} is the estimated iteration time when job k is running alone (Figure 1-top), which is simply the sum of its rollout and training phase durations; $T_k^{\text{co-exec}}$ is the expected iteration time under co-execution, which is derived by simulating the intra-group schedule (§4.3).

Making Placement Decisions. Navigating the search space to find an optimal placement is non-trivial due to both workload heterogeneity (**C1**) and runtime stochasticity (**C2**). WEAVE addresses these complexities with three strategies.

1) Handling Stochasticity via Conservative Planning. To guarantee SLO compliance despite the volatile, unpredictable job execution time (**C2**), WEAVE decouples admission control from runtime optimization. The inter-group scheduler acts as a "gatekeeper" that makes placement decisions based on *worst-case execution bounds*. Specifically, for an arriving job j, we estimate its phase durations (T_j^{roll} and T_j^{train}) assuming that every generated response reaches the *maximum token limit* defined in the job configuration. By planning against this upper bound, we ensure that the chosen placement satisfies the SLO constraints even under the most adverse stochastic conditions. If the actual runtime durations are shorter, which is typical, the intra-group scheduler dynamically reclaims the resulting slack to improve utilization (§4.3).

2) Optimal Placement Search. With these conservative estimates, WEAVE performs a global search to minimize the marginal provisioning cost Δ. For each arriving job, the scheduler iterates through all candidate groups and evaluates three placement strategies:

- *Direct Packing:* Inserting the job into existing dependency bubbles within a group without provisioning new resources (Figure 5-top). This maximizes utilization of already-paid-for capacity.
- *Rollout Scaling:* If a group has available training capacity but is bottlenecked on inference, which is common with rollout-heavy agentic workloads, WEAVE scales up the group's rollout pool by provisioning *just enough* rollout nodes to accommodate the new job (Figure 5-middle).
- *Isolated Provisioning:* As a fallback, WEAVE provisions a new, isolated group for the new job (Figure 5-bottom).

The scheduler iterates through these strategies[2], selecting the

[1] We assume a tight SLO, e.g., tolerance for up to 2× slowdown.

[2] Online reconfiguration of distributed parallel groups is possible (e.g., Tenplex [46]), but still incurs non-trivial overhead relative to a DP scaling, and thus is not supported by mainstream RL frameworks. WEAVE therefore defers training-pool scaling and simply matches the arriving job's DP degree to the existing training-pool size.

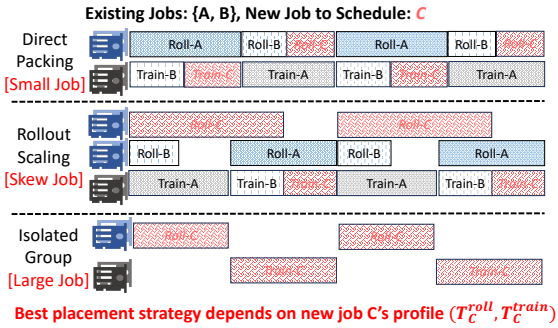


Figure 5: Placement strategies of the inter-group scheduler.

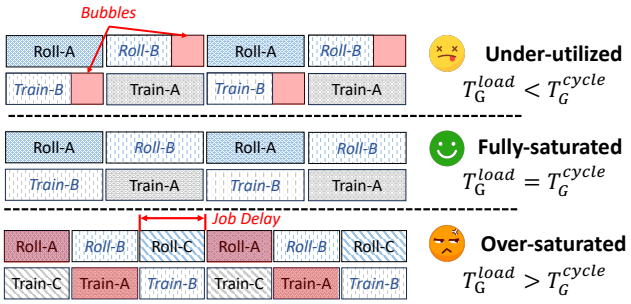


Figure 6: Status of a co-execution group. WEAVE only places new jobs into under-utilized groups with unsaturated dependency bubbles and avoids creating over-saturated groups.

valid placement that minimizes the marginal cost Δ while satisfying both the memory residency and SLO constraints.

3) Pruning Saturated Groups. To ensure this search remains tractable at production scale (**C1**), WEAVE proactively prunes the search space. Before evaluating specific placements, the scheduler filters out groups that are already *saturated*, where the aggregate job load saturates or exceeds the group's bottleneck resource capacity, and adding more work to the group would lead to performance degradation.

Formally, for a group G, let $T_G^{\text{cycle}} = \max_{j \in J_G} T_j^{\text{solo}}$ be the natural cycle iteration time dictated by the *longest job* in the group. We define the group's bottleneck load T_G^{load} as the total time required to process all phases on the bottleneck node. Since all training nodes have identical phases, while rollout nodes may differ[3] (see Figure 5), we have

$$T_G^{\text{load}} = \max\left(\sum_{j \in J_G} T_j^{\text{train}}, \max_{n \in \Phi_G}\left(\sum_{j \text{ on node } n} T_j^{\text{roll}}\right)\right).$$

If $T_G^{\text{load}} \geq T_G^{\text{cycle}}$, the group is saturated, containing no "slack" to absorb new work (Figure 6). Such groups are pruned immediately as any further addition would force delays.

Algorithm Summary. We integrate these strategies into a unified online scheduling logic detailed in Algorithm 1. The algorithm takes a new job j and the set of existing groups

[3]Within each group, WEAVE scales each job's data-parallel degree so that all jobs use the same number of training GPUs and every training node in the group executes the same multiset of training phases; rollout placements may still differ because each job is pinned to its assigned rollout node(s).

Algorithm 1 Inter-Group Scheduling Algorithm

Require: Job to schedule j, all existing groups $\{G_i\}_{i=1}^n$.
Ensure: Best group G^*, best placement P_j^*.

1: **procedure** SCHEDULE($j, \{G_i\}_{i=1}^n$)
2: $\Delta^* \leftarrow \infty$, $G^* \leftarrow$ None, $P_j^* \leftarrow$ None ▷ Initialize best values
3: **for** each group G in $\{G_i\}_{i=1}^n$ **do** ▷ Try all existing groups
4: **if** $T_G^{\text{load}} \geq T_G^{\text{cycle}}$ **then** ▷ Skip saturated groups
5: **continue**
6: $\mathcal{P} \leftarrow$ GENERATEPLACEMENTS(G)
7: **for** each resource placement P_j in $\mathcal{P}$ **do**
8: **if** j.mem_req $\geq \min_{\text{node} \in P_j}$(node.mem_avail) **then**
9: **continue** ▷ Per-node memory constraint violated
10: **if** exists $k \in \{j\} \cup J_G$, s.t., $T_k^{\text{co-exec}} > \text{SLO}_k \times T_k^{\text{solo}}$ **then**
11: **continue** ▷ SLO constraint violated
12: $\Delta \leftarrow \text{Cost}(G \cup \{(j, P_j)\}) - \text{Cost}(G)$
13: **if** $\Delta < \Delta^*$ **then**
14: **Update** $\Delta^* \leftarrow \Delta$, $G^* \leftarrow G$, $P_j^* \leftarrow P_j$
15: $\Delta \leftarrow \text{Cost}(\{j\}, \{\})$ ▷ Try to place j in a new group
16: **if** $\Delta < \Delta^*$ **then**
17: **Update** $\Delta^* \leftarrow \Delta$, $G^* \leftarrow \{j\}$, $P_j^* \leftarrow \{\}$
18: **return** G^*, P_j^*

as input. To find the optimal placement, the algorithm first iterates through all existing groups (line 3), discarding those that are already saturated (line 4). For each remaining candidate group, it evaluates potential placement strategies—direct packing or rollout scaling—for the job (line 6); placements that would violate memory constraints (line 8) or SLO constraints (line 10) are discarded. The algorithm evaluates the marginal cost for each feasible placement (lines 6–12) and updates its records if the placement leads to a lower cost (lines 13–14). Finally, the algorithm compares its records against the baseline cost of provisioning a fresh, isolated group (lines 15–17) and returns the group and job placement that yield the lowest costs.

Running Example. Suppose the cluster currently hosts two active co-execution groups G_1 and G_2, and a new job E arrives for scheduling. Group G_1 is already saturated and gets pruned immediately. Group G_2 still has slack, and contains jobs C and D: C is pinned to nodes (`roll-1`, `train-1`), while D is pinned to (`roll-2`, `train-1`). Thus, C and D use separate rollout nodes but share the same training node. For this group, GENERATEPLACEMENTS produces only a few candidates $\mathcal{P}$. It first tries *direct packing*, e.g., placing E onto (`roll-1`, `train-1`) to share resources with C, or onto (`roll-2`, `train-1`) to share with D. It then tries *rollout scaling*, which keeps E on `train-1` but allocates a new rollout node, e.g., (`roll-3`, `train-1`). Finally, the scheduler compares these feasible placements against *isolated provisioning*, which creates a new group containing only E. Any candidate that violates memory residency or any job's SLO is discarded, and the scheduler chooses the lowest-cost feasible option.

The algorithm allows for highly efficient decision-making. Since $|\mathcal{P}| = O(|R_G|)$ for each group and the SLO check on line 10 is a constant-cost simulation of one meta-iteration (§4.3, Theorem 1), the per-group work is $O(|\mathcal{P}| \cdot |J_G|)$. Since individual groups are small in practice ($|J_G|$ typically below

ten), the overall search complexity is effectively *linear* in the number of active groups. As empirically demonstrated in §6.5, this heuristic allows the scheduler to make optimal decisions in sub-seconds even in clusters with thousands of jobs.

4.3 Intra-Group Scheduling

Once the inter-group scheduler assigns a job to a group, the intra-group scheduler is responsible for orchestrating the runtime execution sequence. Its primary objective is to maximize resource utilization—and thereby minimize dependency bubbles—within the assigned resource pools.

The Round-Robin Policy. WEAVE employs a *cyclic round-robin schedule*. Within a co-execution group, the scheduler defines a *meta-iteration* in which every active job executes exactly one rollout phase and one training phase (Figure 1). These phases are orchestrated sequentially on their assigned resource pools. For example, in a group with jobs $\{A, B\}$, the rollout pool executes $\text{Roll}_A \rightarrow \text{Roll}_B$, while the training pool executes $\text{Train}_A \rightarrow \text{Train}_B$.

While simple, this policy is *optimal* under the preconditions enforced by WEAVE. Recall that the inter-group scheduler (§4.2) proactively prunes any group where the aggregate load exceeds the natural cycle time ($T_G^{\text{load}} \geq T_G^{\text{cycle}}$). For the remaining groups, their optimality is provable.

Theorem 1 (Utilization Optimality) *For any unsaturated or full group G, a meta-iteration schedule that executes each job's phases exactly once in a round-robin order maximizes the aggregate utilization of both rollout and training pools.*

Proof Sketch. The optimality rests on the definition of an *unsaturated (or full) group*: the bottleneck node's total workload T_G^{load} is no more than the longest job's standalone cycle time T_G^{cycle}. Intuitively, this implies that we can pack all other jobs' corresponding phases (e.g., their rollouts) into the longest job's dependency bubbles (e.g., its idle rollout nodes during training). This ensures a round-robin cycle that executes each job's phases exactly once to complete in time T_G^{cycle}. We then show any deviation from this simple schedule is *suboptimal*. (1) *Executing less is impossible*: Omitting any job from the cycle leads to more bubbles and starvation, which is trivially non-optimal. (2) *Executing more is inefficient*: Repeating any job phase prolongs the cycle time as the added phase can only start after finishing the slowest job. However, this added duration is disproportionately larger than the useful work added, leading to a net decrease in utilization.

Therefore, the round-robin schedule, which executes all required work in the shortest possible cycle time, is utilization-optimal. We provide a formal proof in Appendix 8.

Long-Tail Migration. While the round-robin schedule is optimal for deterministic workloads, production RL phases are highly stochastic (**C2**). Specifically, rollout durations follow a *heavy-tailed distribution* where the completion time of an input batch is dictated by a small number of "straggler" responses that reach maximum token limits [14,19,61,62]. This phenomenon creates significant *intra-phase fragmentation*: as the majority of responses finish early, most GPUs in the rollout pool idle wait for the few stragglers to complete, creating "skewness bubbles" (Figure 7-top).

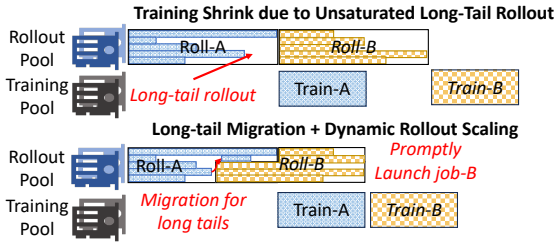


Figure 7: Long-tail migration effectively handles dynamism.

To reclaim this fragmented capacity, WEAVE employs *long-tail migration* to dynamically adapt the schedule at runtime. The intra-group scheduler continuously monitors the progress of active rollout phases. When a phase enters a *tail-bound state*, triggered when a threshold of responses (e.g., 80%) have completed, the system interrupts the execution, consolidates the remaining long-tail responses onto a small subset of workers[4], and immediately starts the *next* job's rollout phase on the newly freed rollout GPUs (Figure 7-bottom). This mechanism effectively pipelines the tail of one job with the head of the next, ensuring high utilization and faster completion. It is also the runtime counterpart of the conservative admission in §4.2: worst-case planning reserves rollout capacity for stragglers that rarely materialize, and this mechanism converts the unused reservation into useful work for the *next* job in the round-robin order. We will show in §6.3 that long-tail migration is semantics-preserving: it changes *where* the tail finishes, not *what* the rollout produces, and does not alter the underlying RL learning signal or convergence.

5 The WEAVE System Design

Figure 8 shows WEAVE's architecture. It is organized around three pieces: a *control plane* (the two-tier scheduler, a per-job phase shim, and runtime hooks), a *data plane* (the disaggregated rollout and training GPU pools, with each job's working set cached in host DRAM for warm starts), and a *cross-cluster sync channel* that carries model updates from the training pool to the rollout pool. WEAVE is implemented in ∼5.2k lines of code atop ROLL[5] [45], with controllers in Python and the placement search in C++.

[4]Migration moves only the *remaining requests* of a tail-bound rollout phase, not the job's model or KV cache, so sampling RNG (random number generator), tokenization, and batch composition are preserved; the target workers are a subset of the same homogeneous rollout pool, so numerical results match what the original placement would have produced.

[5]We choose ROLL [45] as the underpinning RL runtime because it exposes per-phase entry points suitable for shim-based enforcement; any disaggregated RL runtime with comparable hooks could serve as the substrate.

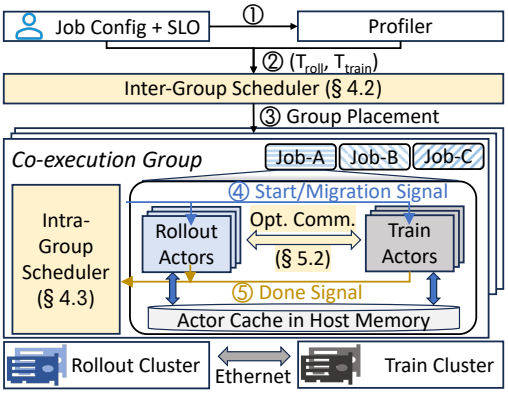


Figure 8: WEAVE's system architecture.

Workflow. The system operates as a closed loop (Figure 8). Upon job submission, WEAVE first launches a lightweight profiler (①)—a one-shot timing pass that runs the user's rollout and training functions at the job's configured maximum token limit—to generate worst-case duration estimates for the job's rollout and training phases (②). These estimates are fed into the inter-group scheduler (§4.2), which identifies the optimal co-execution group and resource placement to minimize marginal cost (③). Once placed, the job comes under the control of the intra-group scheduler (§4.3). This runtime controller orchestrates the round-robin meta-iteration (④), driving phase switches through the per-job phase shim and triggering long-tail migrations based on real-time feedback from its runtime hooks (§5.1). Once a phase completes, WEAVE offloads its state to the actor cache in host DRAM, releases GPU resources, and launches the next job's waiting phase from the scheduler queue (⑤). Model-update traffic between the *Train* and *Rollout* pools crosses a slow cross-cluster link; WEAVE routes it through a topology-aware scatter-then-broadcast (§5.2) so that sync does not itself become a new bubble.

5.1 Phase Control

Conventional cluster schedulers treat a job as the atomic unit of resource allocation, which is too coarse for interleaving distinct RL phases of different jobs on the same hardware. WEAVE instead schedules at the granularity of *individual phases*: each phase is a first-class schedulable entity with its own admission, state, and runtime controls.

Execution and Switching. We model each RL job as a dependency graph of phases. After a one-time initialization (`Init`) of the job states (e.g., models, datasets), the job enters a cyclic dependency loop: `Rollout` → `Train` → `Sync`. WEAVE exposes this internal structure to the scheduler via a declarative Python API. Users simply annotate their phase functions with a `@weave.phase` decorator, which injects a *transparent runtime shim* to manage the execution lifecycle.

When a phase is invoked, this shim first blocks execution until it acquires a run permit from the intra-group scheduler. Upon approval, it performs a *warm start* by loading the phase's resident working set from host DRAM into GPU memory. Once the user function completes, the shim immediately offloads the updated state back to host memory and releases the occupied GPUs, making the hardware instantly available for the next phase in the group's queue. Crucially, WEAVE optimizes this switching process by decoupling data plane state from control plane context. Naively terminating a process after a phase completes would force the system to tear down and rebuild expensive control plane (e.g., NCCL communicators, environment handles) upon every switch. Instead, WEAVE employs a *lightweight suspension* strategy: after offloading, the shim places the process into a sleep loop while retaining its control plane context without consuming GPU resources. On the next wake-up, resuming the phase only requires reloading its cached state onto the GPU, avoiding expensive cold starts from disk and control-plane re-initialization.

Runtime Hook. The system exposes a runtime hook interface `@weave.runtime_hook`. This interface serves two critical roles. First, it drives the round-robin schedule: the intra-group scheduler maintains a FIFO queue for each worker node. When a job's phase completes, the hook signals the scheduler to enqueue the job's next phase onto the alternate resource pool's queue (e.g., moving from rollout to training) and starts the next waiting phase on the now-idle devices. Second, it enables the long-tail migration (§4.3). By exposing internal token generation progress, the hook allows the scheduler to detect tail-bound states and externally trigger migration, dynamically reconfiguring resources in real-time.

5.2 Topology-Aware Model Synchronization

To mitigate the cross-cluster bandwidth bottleneck, WEAVE employs a *topology-aware* communication strategy to efficiently synchronize model parameters from the training cluster to the rollout cluster. State-of-the-art RL frameworks (e.g., veRL [43]) rely on flat collective operations like `AllGather` to propagate model updates. While efficient in monolithic clusters, this approach is pathological in disaggregated setups. It treats the slow cross-cluster Ethernet link and the fast intra-cluster InfiniBand/NVLink fabric as a single uniform network. Consequently, it forces every rollout worker to independently fetch a full copy of the model parameters over the slow cross-cluster link (Figure 9-top), causing a severe bottleneck while leaving local high-speed fabrics idle.

Inspired by LLM auto-scaling and multi-casting systems [4, 13, 56, 58, 65], we exploit the heterogeneous bandwidth and redundant model copies, a unique characteristic of disaggregated RL post-training, to reduce synchronization cost. WEAVE replaces the flat collective with a *hierarchical two-stage transfer*. ① In the first stage (**inter-cluster scatter**), it partitions the updated model into N disjoint shards, where N is the number of training GPUs. Each training GPU transmits a unique shard to a corresponding rollout GPU via parallel

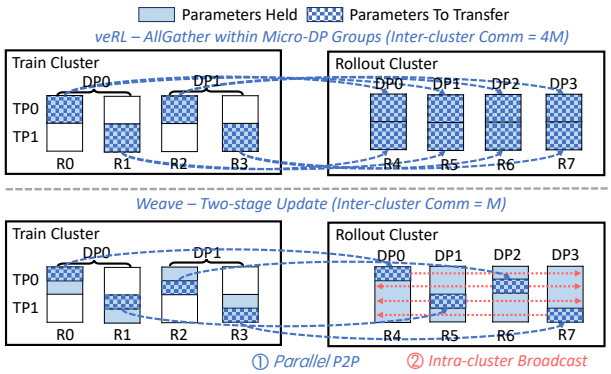


Figure 9: A synchronization example from the training cluster (TP=2, DP=2) to the rollout cluster (DP=4), where WEAVE sends exactly one copy across the cross-cluster network. M denotes the number of total parameters.

point-to-point (P2P) streams. This ensures that *exactly one full copy* of the model traverses the slow cross-cluster link. ② In the second stage (**intra-cluster broadcast**), the receiving GPUs immediately disseminate their shards to all other rollout workers using the high-bandwidth InfiniBand/NVLink fabric. This two-stage pipeline effectively mitigates the slow cross-cluster bottleneck, minimizing overall synchronization time and fully utilizing network hierarchies.

5.3 Reliability and Robustness

Fault Isolation. To ensure production-grade reliability, WEAVE enforces strict fault isolation. Each job runs in a dedicated Ray [30] instance with an isolated runtime environment, and jobs communicate with the scheduler exclusively through a Redis [38] control channel; they never exchange messages directly. A crash in one job is therefore contained within its pod, preventing error propagation and keeping the other jobs in the same co-execution group unaffected. WEAVE treats a recovered job as a *new arrival*: upon resubmission, it re-runs admission and placement via Algorithm 1 and restores the last completed iteration from checkpoint, so the rest of the cluster does not need to halt or regroup. Figure 19 reports end-to-end behavior under an injected crash.

Robustness to Workload Drift. A job's rollout and training profile can *drift* during its lifetime, e.g., as response length grows with training. WEAVE absorbs drift with its admission-time placement rather than an online regrouping loop, for three reasons. First, global repartitioning at every drift event is NP-hard [25] and incurs substantial migration and restart overhead—nearly 9% of trace time in our experiments (§6.6). Second, the conservative admission policy of §4.2 reserves capacity against worst-case token lengths, so moderate drift only shifts *which* bubbles are consumed, without breaching the group's saturation budget. Third, long-tail migration can also absorb the rollout drifts, keeping the round-robin cycle still near-optimal on the drifted workload (Theorem 1). If a job drifts far enough that its group crosses $T_G^{\text{load}} \geq T_G^{\text{cycle}}$, we simply stop and resubmit *that job*; Algorithm 1 re-places it through normal admission without disturbing the rest of the cluster. §6.6 shows static placement stays $> 95\%$ SLO attainment and within $1.11\times$ the provisioning cost of an optimal yet unimplementable dynamic-regrouping baseline.

Job	Turns	Model	Len[6]	Bsz	N_T	N_R
Type-A	Single-Turn	Qwen-2.5-7B	8K	256	8	8
Type-B	Single-Turn	Qwen-2.5-14B	8K	256	8	8
Type-C	Single-Turn	Qwen-2.5-32B	8K	256	16	16
Type-D	Multi-Turn	Qwen-3-8B	8K*	256	8	8
Type-E	Multi-Turn	Qwen-3-14B	16K*	64	8	8

Table 3: Experimental job configurations. N_T and N_R denote training and rollout GPUs.

6 Evaluation

We evaluate WEAVE through five research questions. **RQ1 (Co-Execution Mechanism):** Can WEAVE co-execute job groups with diverse phase profiles (§6.2)? **RQ2 (Performance Breakdown):** How much do long-tail migration and topology-aware model sync each contribute to performance (§6.3)? **RQ3 (Performance at Scale):** What cluster utilization and provisioning cost does WEAVE achieve on a production workload trace (§6.4)? **RQ4 (Scheduling Quality):** How close to optimal is WEAVE's scheduler (§6.5)? **RQ5 (Robustness):** How robust is WEAVE to phase profile drift and job failures (§6.6)?

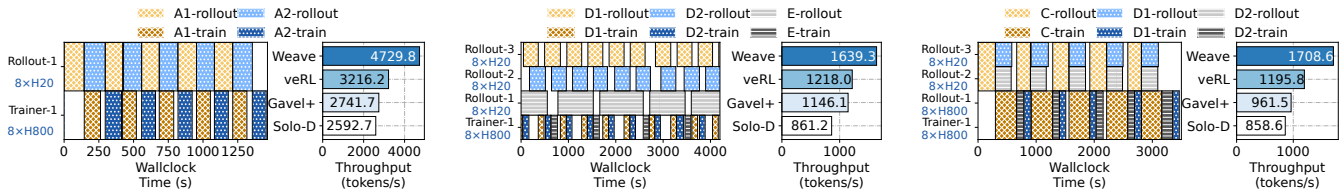
6.1 Experimental Setup

Cluster Setup. Our testbed has two geo-distributed, heterogeneous clusters: the training cluster (`Cluster-T`) uses compute-optimized NVIDIA H800 GPUs, while the rollout cluster (`Cluster-R`) uses lower-cost H20 GPUs. Each cluster has a 400 Gbps InfiniBand fabric, but cross-cluster traffic traverses a bandwidth-constrained 20 Gbps Ethernet link. Table 1 lists hardware specifications and hourly costs; an H800 GPU is $2.85\times$ more expensive than an H20 GPU.

Workloads. We construct workloads from production traces. For **micro-benchmarks** (§6.2–§6.3), we define a suite of five representative job types (Table 3) using Qwen [53] models (7B–32B) with varying batch sizes, sequence lengths, and rollout/training GPUs. They cover single-turn RLVR on DeepMath-103K [41] and multi-turn agentic reasoning on Math-Orz57K [23]. For **at-scale evaluation** (§6.4), we replay a two-week trace of 200 heterogeneous jobs with model sizes from 3B to 32B and datasets spanning mathematics [41], software engineering [33], games [8], and in-house tasks.

Baselines. We compare WEAVE against three baselines.

[6] For multi-turn workloads, Len is the per-turn output length.



(a) Temporal Mux (single-turn, Type-A×2). (b) Train Mux (multi-turn, Type-D×2 + E). (c) Spatial Mux (mixed, Type-C + D×2).

Figure 10: Micro-benchmarking results. For each benchmark, the left panel is a Gantt chart showing the co-execution timeline; the right panel shows WEAVE achieves up to 1.99× higher training throughput under the same cluster provisioning cost.

- *Solo Disaggregation (Solo-D):* The standard disaggregation, where jobs run on dedicated rollout and training pools without time-multiplexing.
- *Co-location (veRL [43]):* The traditional monolithic approach, where all phases run on the high-performance training cluster (`Cluster-T`) using veRL. It avoids network bottlenecks but suffers from hardware mismatch.
- *Gavel+ [31]:* An enhanced version of the heterogeneity-aware Gavel scheduler [31], modified to support RL post-training. It optimizes resource allocation at the job level (calculating optimal GPU fractions) but lacks fine-grained control to interleave phase-level executions.

6.2 Micro-Benchmarks

We use three micro-benchmarks to show how WEAVE reclaims dependency bubbles (§4.3) and handles failures across multiplexing scenarios. Each benchmark compares training throughput against the baselines under the same per-hour cluster provisioning cost.

Temporal Multiplexing. First, we evaluate co-executing two jobs with similar structures (Type-A), an ideal case in which jobs are fully complementary. As shown in Figure 10a, WEAVE interleaves their execution and keeps both rollout and training clusters fully utilized. Consequently, WEAVE improves end-to-end training throughput by 1.82×, 1.73×, and 1.47× over Solo-D, Gavel+, and veRL, respectively, under the same cluster provisioning cost. Solo-D and Gavel+ leave one resource pool idle, while monolithic veRL underutilizes expensive H800 compute during memory-bound rollout.

Handling Rollout-Heavy Jobs. Next, we target rollout-heavy workloads by co-scheduling two Type-D jobs ($T_D^{\text{roll}} \approx 2.5T_D^{\text{train}}$) and one Type-E job ($T_E^{\text{roll}} \approx 6T_E^{\text{train}}$). In this scenario, WEAVE scales the rollout pool to 24 H20 GPUs, dedicates one inference node to each rollout phase, and time-multiplexes a single H800 training node across training phases in round-robin order (Figure 10b). This schedule achieves 1.90×, 1.43×, and 1.35× higher throughput than Solo-D, Gavel+, and veRL under the same budgets. Solo-D performs poorly because prolonged rollouts leave expensive training nodes idle. WEAVE's gains over Gavel+ and veRL are smaller than in (a) because the skewed rollout-heavy pattern leaves some bubbles unfilled (Figure 10b-left).

Micro-benchmark	Solo-Disagg	Ideal	WEAVE
(a) Temporal Mux	257.7s	240.8s (-6.6%)	261.8s (+1.6%)
(b) Train Mux	525.4s	490.3s (-6.7%)	552.0s (+5.1%)
(c) Spatial Mux	574.6s	516.7s (-10.1%)	630.8s (+9.8%)

Table 4: Micro-benchmark iteration time. WEAVE incurs at most 10% overhead over isolated execution; 'Ideal' is the performance ceiling when all phases run on H800 GPUs.

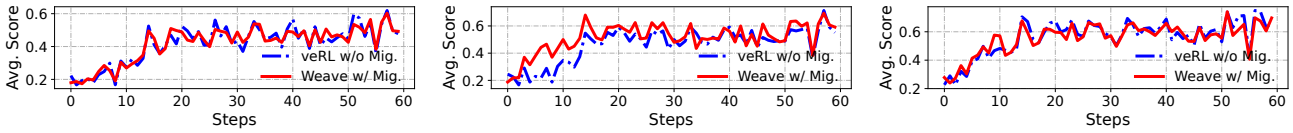
Spatial Multiplexing. Finally, we test heterogeneity by co-scheduling one large Type-C job (requiring 16 × H20 + 16 × H800) with two smaller Type-D jobs (each requiring 8 × H20 + 16 × H800). As shown in Figure 10c, WEAVE identifies idle resources created by the large job's rollout phase and "packs" the two smaller jobs into these bubbles. This spatial packing maximizes aggregate utilization, delivering 1.99×, 1.78×, and 1.43× higher throughput than Solo-D, Gavel+, and veRL. Unlike the baselines, WEAVE can dynamically consolidate diverse jobs onto available capacity.

Interference Overhead. We quantify co-execution cost by measuring per-iteration slowdown from inter-job contention (Table 4). Because the inter-group scheduler prunes placements that would violate residency or SLOs (§4.2), WEAVE is only 1.6–9.8% overhead over solo execution. Even against an idealized co-location upper bound, where each job runs all phases on expensive H800 GPUs with zero network cost, WEAVE is only 8.0–18.1% slower. This confirms that our optimizations mask most context-switching and synchronization costs while excluding contention-prone placements.

6.3 Ablation Study

We next isolate the contribution of WEAVE's runtime optimizations: long-tail migration and topology-aware model synchronization.

Long-Tail Migration. First, we evaluate whether request migration reclaims rollout bubbles without changing RL outcomes (§4.3). Figure 12-left shows that rollout generation lengths are heavy-tailed across model sizes and output lengths: a small fraction of straggler requests remains after most requests complete. Figure 12-right shows that migrating these tail requests to a small GPU subset reclaims the resulting



(a) Qwen2.5-7B. (b) Qwen2.5-14B. (c) Qwen2.5-32B.

Figure 11: Training reward scores for Qwen2.5-7B, 14B, and 32B under veRL without migration and WEAVE with migration.

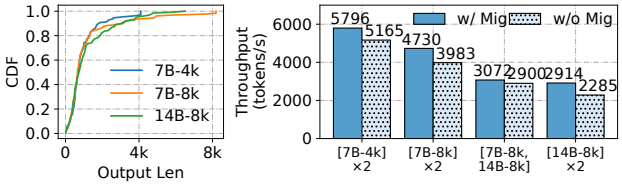


Figure 12: **Left**: the long-tail distribution of LLM generation length in the rollout phase. **Right**: effectiveness of request migration in mitigating long-tail rollouts.

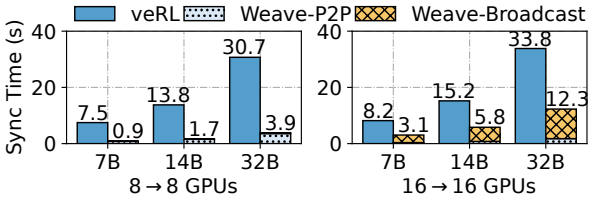


Figure 13: Model synchronization time. **Left**: single-node transfer from 8 H800 to 8 H20 GPUs. **Right**: multi-node transfer from 16 H800 to 16 H20 GPUs. WEAVE's topology-aware model sync is up to 8.33× faster than veRL [43].

"skewness bubbles." WEAVE can then start the next job's rollout on the freed GPUs, improving end-to-end throughput by 1.06×–1.28×. The gains are largest for longer output sequences (e.g., 14B-8k), where stragglers are more pronounced, and more modest when paired jobs have dissimilar phase durations (e.g., 7B-8k with 14B-8k), which already reduce contention on rollout nodes.

We also test whether migration changes the learning outcome. Figure 11 compares WEAVE with disaggregation and migration against a clean veRL baseline that runs on a monolithic H800 cluster without either mechanism, using Qwen-2.5 models at 7B, 14B, and 32B scales. For all three model sizes, WEAVE's average per-step reward tracks veRL throughout training and stays within measurement noise. This is expected because WEAVE only migrates rollout requests between homogeneous GPUs (e.g., H20-to-H20), never across heterogeneous devices, avoiding hardware-induced variability. Thus, long-tail migration improves throughput without measurably degrading rollout reward or training convergence.

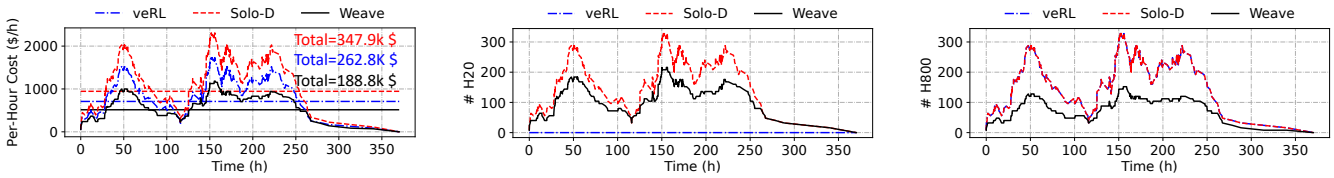
Topology-Aware Model Sync. Next, we evaluate topology-aware model synchronization in geo-disaggregated setups (§5.2). For a single-node update (8 H800s → 8 H20s), WEAVE is 7.87×–8.33× faster than veRL (Figure 13). The gain comes from sending one copy of the model parameters over the slow inter-cluster link and using local NVLink for the intra-cluster broadcast; veRL redundantly fetches independent copies for each rollout GPU. The benefit persists at multi-node scale: for 16 H800s → 16 H20s, WEAVE remains 2.62×–2.75× faster, showing that the protocol mitigates the bandwidth bottleneck introduced by disaggregation.

6.4 WEAVE at Scale

To evaluate WEAVE at production scale, we replay a two-week trace from one cluster tenant. The trace contains 200 heterogeneous RL post-training jobs using Qwen-family models, with both single- and multi-turn interactions across diverse datasets. Model sizes range from 3B to 32B, maximum response lengths range from 4k to 32k tokens (mean: 12.1k), and the mean job duration is 27.9 hours. We assign each job an SLO sampled uniformly from $(1, 2)$ relative to its solo runtime, and compare WEAVE with industry-standard solo disaggregation (Solo-D), which uses 1:1 rollout and training GPUs, and the monolithic co-located baseline (veRL).

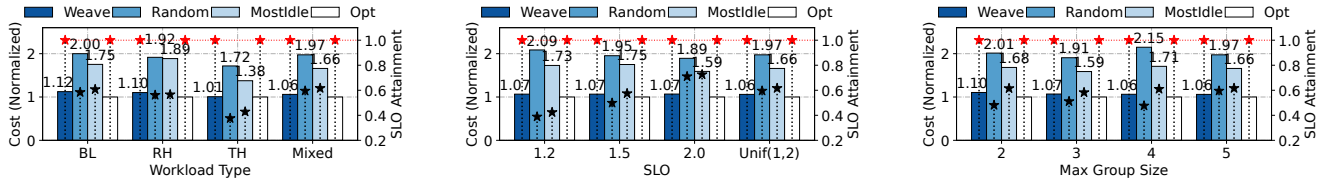
WEAVE accommodates all jobs for $510/hr (total: $188.8k), reducing cluster provisioning cost by 1.84× over Solo-D and 1.38× over veRL while meeting all job SLOs (Figure 14a). The admission decisions also show where the savings come from: among the 200 job arrivals, 60.5% use direct packing to share both training and rollout resources with existing groups, 15% share training resources while scaling the rollout pool, and only 24.5% form new resource groups. Thus, Algorithm 1 usually absorbs arrivals into existing groups, minimizing the marginal provisioning cost of each job.

WEAVE improves resource efficiency by reducing dependency bubbles by 24.4% on the rollout cluster and 43.1% on the training cluster relative to solo disaggregation. The reduction is larger for training because production jobs are typically rollout-heavy, leaving more idle time on training GPUs (Figure 14b and Figure 14c). At peak, WEAVE uses only 152 H800 training GPUs, while both veRL and Solo-D requiring 328 H800s (2.16× more). For rollout, WEAVE uses 216 H20 GPUs at peak, a 1.52× reduction from the 328 H20s required by Solo-D. Although veRL avoids separate rollout GPUs, WEAVE remains 1.38× more cost-effective overall by offloading memory-bandwidth-bound rollouts to cheaper H20 GPUs and filling the resulting dependency bubbles through co-scheduling.



(a) Cluster provisioning cost. (b) Number of rollout (H20) GPUs. (c) Number of training (H800) GPUs.

Figure 14: **[Testbed]** Cluster provisioning cost and GPU usage of WEAVE and baselines under real-world production workloads, where WEAVE reduces total provisioning cost by 1.38× and 1.84× compared to veRL and Solo-D, respectively.



(a) Sensitivity to workload types. (b) Sensitivity to job SLOs. (c) Sensitivity to group residency.

Figure 15: **[Simulation]** Sensitivity analysis of WEAVE's inter-group scheduler.

6.5 Scheduler Performance

We then evaluate the optimality and scalability of WEAVE's inter-group scheduler (§4.2) via large-scale trace simulation.

Experimental Setup. We use job arrival patterns from a 300-job, 580-hour segment of the Microsoft Philly multi-tenant training cluster trace [26], the average job duration is 14.4 hours, and the longest is 142.9 hours. While the trace dictates arrival times and durations, we synthesize the job characteristics to model modern RL post-training workloads. As detailed in Table 6, we define three job profiles based on the ratio of rollout time (T_{roll}) to training time (T_{train}). (1) **Balanced (BL):** Balanced T_{roll} and T_{train}, representative of single-turn workloads like RLHF [32] or RLVR [41]; (2) **Rollout-Heavy (RH):** $T_{\text{roll}} \gg T_{\text{train}}$, modeling multi-turn workloads such as agentic reasoning [33]; and (3) **Train-Heavy (TH):** $T_{\text{train}} \gg T_{\text{roll}}$ for evaluation completeness, but is rare in real-world RL training.

For each profile, we generate jobs of three sizes (Small, Medium, Large), resulting in nine distinct job configurations. We test each profile individually and also use a **Mixed** workload, which contains a uniform mix of all nine configurations, to simulate a realistic production environment.

Baselines. We compare WEAVE against following baselines:

- *Offline Optimal (Opt).* Assigns arriving jobs to *offline optimal* placements found via a brute-force search over all possible groupings and placements–including job reordering and re-grouping that are infeasible in online deployments–and serves as a theoretical upper bound.
- *Random.* Assigns arriving jobs to a random group (or a new one) that can accommodate them. The job is placed on random rollout and train nodes within this group.
- *Greedy (Most-Idle).* Assigns arriving jobs to the group with the highest idle-time percentage. Within that group, it places the job on the most idle rollout and train nodes.

Sensitivity Analysis. To determine how sensitive the scheduler's quality is to its key parameters, we vary a single parameter while all others are set to a default configuration[7].

Impact of Workload Characteristics. Figure 15a shows that WEAVE's near-optimal performance holds across all workload types. It consistently achieves 100% SLO attainment with a cost overhead of just 1.01×–1.12× relative to the optimal. In contrast, the baselines perform poorly across workloads, with Greedy slightly outperforming the Random strategy. The Random strategy's cost is 1.72×–2.00× optimal with only 37–58% SLO attainment, while the Greedy scheduler is slightly better but still inadequate, reaching 1.38×–1.89× optimal cost for 42–61% SLO attainment.

Impact of Job SLOs. We vary the job SLO requirements, testing both uniform SLOs (all jobs set to 1.2, 1.5, or 2.0) and heterogeneous, job-specific SLOs drawn from `Unif(1, 2)`. As shown in Figure 15b, WEAVE's performance is highly stable against SLO tightness, always achieving 100% attainment with a consistent, near-optimal cost. Conversely, the baselines are highly sensitive to the SLO target and more expensive (1.59×–2.09× optimal). As the SLO target loosens from 1.2 to 2.0, the SLO attainment for Random and Greedy improves from 38%/43% to 71%/73%, respectively. This shows that while looser SLOs make it easier for naive heuristics to succeed by chance, they still fail to provide guarantees.

Impact of Group Residency. Since node memory capacity directly limits the group size, we evaluate its impact by varying the maximum allowed group size from 2 to 5. Figure 15c shows that performance is relatively insensitive to the maximum group size for all methods. Across all configurations, WEAVE consistently maintains the lowest cost and 100% SLO attainment, while the baselines remain signifi-

[7]Default configuration: mixed workload types, heterogeneous SLOs drawn from `Unif(1,2)`, and a max group residency of 5.

Decision	Number of Concurrent Jobs						
Lat. (ms)	5	9	13	100	500	1000	2000
WEAVE	5.6	6.5	7.6	41.9	198	318	591
Opt.	113	>1min†	>5h†	—*	—*	—*	—*

† Represents latency exceeding 1 minute and 5 hours, respectively.
* Not applicable; computation is intractable at this scale.

Table 5: Decision latency (ms) vs. number of concurrent jobs. WEAVE scales well; Brute-force Opt's latency grows exponentially and quickly becomes impractical.

cantly defective (only 48%–61% SLO attainment) and more expensive (1.59×–2.15× optimal). This suggests that even small group sizes (e.g., 2 or 3) provide sufficient packing flexibility for WEAVE to find efficient placements; larger groups do not bring improved performance or cost efficiency.

Scheduler Scalability and Latency. We evaluate WEAVE's decision latency and scalability, with results presented in Table 5. WEAVE demonstrates near-linear scalability: its decision time is only 591 ms for 2,000 jobs, confirming that Algorithm 1 efficiently handles production-scale workloads. In stark contrast, the brute-force optimal solver exhibits exponential growth in latency, exceeding five hours for just 13 jobs–rendering it infeasible for any practical workload size.

6.6 Robustness to Drifts and Failures

We finally evaluate WEAVE's robustness against *workload drift*, where a job's rollout and training times change over its lifetime, and *job failures*.

Handling Workload Drift. We evaluate drift using the real-world trace from §6.4. To model long-term non-stationarity, we derive drift templates from production training traces across model sizes. Figure 16 shows representative 7B–32B examples; we omit the remaining model sizes for brevity. For each model family, we instantiate seven templates: no drift, increasing drift at 0.5×/1.0×/1.5×, and decreasing drift at 0.5×/1.0×/1.5×. The increasing 1.0× template is the observed training behavior; the other templates are scaled or mirrored versions of it. We evaluate three settings: **Increasing**, where each job samples from {no drift, increasing 0.5×, 1.0×, 1.5×}; **Decreasing**, defined analogously; and **Mixed**, where each job samples from all templates.

We compare WEAVE's static placement against an *optimal dynamic regrouping* baseline that periodically recomputes placements for all active jobs. The baseline checks for global regrouping every hour and uses brute-force search to find the exact optimal grouping under the jobs' current drifted profiles[8]. When regrouping changes a job's placement, we charge the measured migration and restart cost, 197–419 s for 7B–32B jobs. We do *not* charge the solver time for brute-force regrouping, making the baseline stronger than any practical online implementation.

[8]We choose a one-hour period because more frequent regrouping only increases restart overhead and does not improve this baseline in practice.

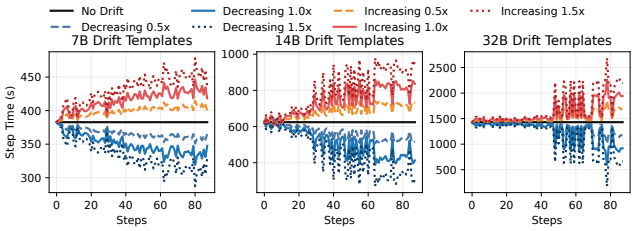


Figure 16: Workload drift templates for 7/14/32B models.

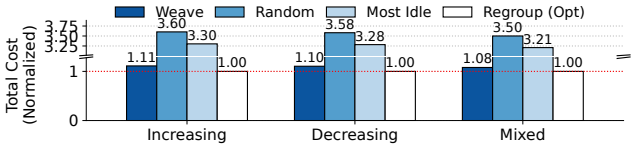


Figure 17: Normalized cost under workload drifts.

Figure 17 shows that static WEAVE stays close to this optimal baseline. Relative to dynamic regrouping, static WEAVE incurs only 1.11×, 1.10×, and 1.08× higher provisioning cost under increasing, decreasing, and mixed drift, respectively, while maintaining 98.6%, 98.7%, and 95.6% SLO attainment. Drift-agnostic baselines perform much worse: Random incurs 3.50×–3.60× the regrouping cost, and Most-Idle incurs 3.21×–3.30×. Figure 18 explains why static placement is usually sufficient. Static WEAVE has only modestly lower rollout utilization than dynamic regrouping (at most 5.9%), but achieves higher training utilization in all settings: drift is driven mainly by rollout-length changes, so regrouping improves rollout-side balance while each restart pausing directly reduces training-side utilization.

These results clarify when WEAVE needs adjustment under drift. Online global regrouping is often unnecessary: in our 370.2-hour trace, it spends 32.9 hours on migration and restart alone, even before solving the regrouping problem. Each regrouping step also requires solving a global NP-hard repartitioning problem, making it difficult to deploy as a frequent online control loop. Dynamic regrouping is therefore useful only when drift is severe and persistent enough to amortize both restart and re-optimization costs. In practice, when one job drifts substantially, the simpler operational response is to stop and resubmit only that job with its updated profile; WEAVE then places it through normal admission without perturbing the rest of the cluster.

Handling Job Failures. Figure 19 shows failure handling in the Train Mux setting from Figure 10b. At $t = 2180$ s, we manually kill job E to emulate a failure. The other jobs in the group (D_1 and D_2) continue unaffected and keep their two-job interleaving pattern. At $t = 4480$ s, we recover and resubmit job E. WEAVE treats the recovered job as a *new job*: it reruns admission and placement, restores the last failed iteration from checkpoint, and resumes training. Depending on cluster state at recovery time, WEAVE may place the recovered job in a different co-execution group; in this run, it is placed back into the original group. After resubmission, the three

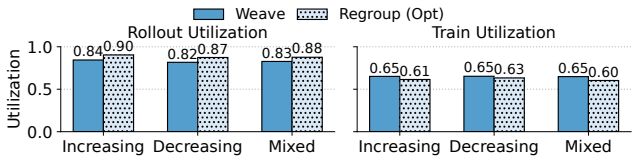


Figure 18: Rollout and training cluster utilization for WEAVE and regrouping under workload drifts.

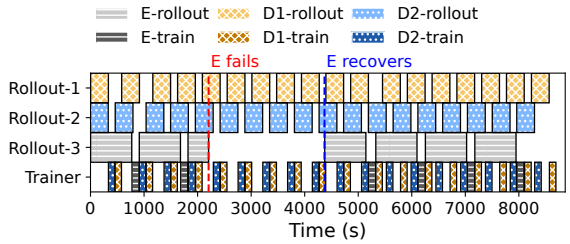


Figure 19: WEAVE's behavior under job failures.

jobs return to the original interleaving pattern and continue execution.

7 Discussion and Related Work

Applicability. WEAVE applies to any RL training regime that exposes a *structural dependency bubble* between rollout and training, the idle window its inter-job scheduler is designed to reclaim. This condition holds for the standard on-policy algorithms, including PPO [39], GRPO [41], and DAPO [57]. It also holds for off-policy regimes that retain such a bubble, such as one-step off-policy training [61], where each training step still waits for a bounded rollout window to complete. The boundary of applicability is reached when this bubble disappears: fully asynchronous RL training (e.g., AReaL [11] and AsyncFlow [18]) overlaps rollout and training continuously, leaving no inter-job idle time for WEAVE to reclaim. Within its applicable regimes, WEAVE's cluster-level orchestration is orthogonal to intra-job optimizations: parameter relocation [29], request-level tail batching [14], and speculative decoding [5, 19, 24, 36] all act within a single job or phase and compose directly with WEAVE.

Deep Learning Workload Scheduling. Extensive research has focused on scheduling for general-purpose deep learning clusters. These works aim to improve fairness [16, 28, 50], GPU sharing efficiency [49, 51], heterogeneity awareness [31, 42], and job goodput [35, 59]. However, these systems universally treat the entire job as the atomic unit of scheduling and assume stable, predictable iteration times. WEAVE is the first phase-centric co-scheduling system for stochastic RL post-training jobs.

RL Post-Training Frameworks. Recent systems have focused on optimizing the performance of a *single, individual* RL post-training job. While early frameworks used static resource partitioning [22, 27, 55], subsequent work improved utilization via co-location [43], multi-controller designs [48], long-tail mitigation [14, 62], and asynchronous algorithms [11, 18, 61]. Different from these single-job optimizations, WEAVE addresses the complementary part by taking a global, cluster-level perspective, orchestrating multiple concurrent jobs via co-scheduling.

Disaggregated Systems. WEAVE builds on the principle of disaggregation, a concept explored in OS kernels [40] and serving systems [21, 34, 60]. The most direct parallel is the prefill-decode disaggregation in LLM inference [21, 34, 60]. Within this domain, recent serving systems have also explored efficient resource management, but all in a single-job scope [6, 10, 20, 63]. In contrast, WEAVE introduces the first scheduling framework specifically designed for multi-tenant clusters, a scenario not addressed by prior works.

8 Conclusion

This paper presents WEAVE, a multi-tenant cluster scheduler tailored for disaggregated RL post-training. WEAVE introduces a two-tier scheduling mechanism that near-optimally partitions RL jobs into co-execution groups and orchestrates their execution in a tightly-woven pattern. It effectively reduces dependency bubbles caused by disaggregation, thereby minimizing cluster provisioning costs. Extensive evaluation using real-world production traces demonstrates that WEAVE achieves up to 1.84× cost savings while maintaining 100% performance SLO attainment.

Acknowledgment

We thank our shepherd and the anonymous reviewers for their invaluable comments that help improve the quality of this work. This work was supported in part by the HKUST-Alibaba Joint Laboratory on Big Data and AI, RGC CRF Grant (Ref. #C6015-23G), RGC GRF Grant (Ref. #16217124), and NSFC/RGC CRS Grant (Ref. #CRS_HKUST601/24).

References

[1] Introducing openai o3 and o4-mini. https://openai.com/index/introducing-o3-and-o4-mini/, 2024.

[2] AIME 2024 Dataset, 2025.

[3] Qwq-32b: Embracing the power of reinforcement learning. https://qwenlm.github.io/blog/qwq-32b/, 2025.

[4] Jonathan Behrens, Sagar Jha, Ken Birman, and Edward Tremel. RDMC: A reliable RDMA multicast for large objects. In *48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2018, Luxembourg City, Luxembourg, June 25-28, 2018*, pages 71–82. IEEE Computer Society, 2018.

[5] Qiaoling Chen, Zijun Liu, Peng Sun, Shenggui Li, Guoteng Wang, Ziming Liu, Yonggang Wen, Siyuan Feng, and Tianwei Zhang. Respec: Towards optimizing speculative decoding in reinforcement learning systems. *arXiv preprint arXiv:2510.26475*, 2025.

[6] Weihao Cui, Yukang Chen, Han Zhao, Ziyi Xu, Quan Chen, Xusheng Chen, Yangjie Zhou, Shixuan Sun, and Minyi Guo. Optimizing slo-oriented llm serving with pd-multiplexing, 2025.

[7] DeepSeek-AI. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning. *arXiv preprint arXiv:2501.12948*, 2025.

[8] Farama Foundation. Gymnasium - frozenlake environment. https://gymnasium.farama.org/environments/toy_text/frozen_lake/, 2024. Accessed: 2025-09.

[9] Jiazhan Feng, Shijue Huang, Xingwei Qu, Ge Zhang, Yujia Qin, Baoquan Zhong, Chengquan Jiang, Jinxin Chi, and Wanjun Zhong. Retool: Reinforcement learning for strategic tool use in llms. *arXiv preprint arXiv:2504.11536*, 2025.

[10] Jingqi Feng, Yukai Huang, Rui Zhang, Sicheng Liang, Ming Yan, and Jie Wu. Windserve: Efficient phase-disaggregated llm serving with stream-based dynamic scheduling. In *Proceedings of the 52nd Annual International Symposium on Computer Architecture*, pages 1283–1295, 2025.

[11] Wei Fu, Jiaxuan Gao, Xujie Shen, Chen Zhu, Zhiyu Mei, Chuyi He, Shusheng Xu, Guo Wei, Jun Mei, Jiashu Wang, et al. Areal: A large-scale asynchronous reinforcement learning system for language reasoning. *arXiv preprint arXiv:2505.24298*, 2025.

[12] Yao Fu, Leyang Xue, Yeqi Huang, Andrei-Octavian Brabete, Dmitrii Ustiugov, Yuvraj Patel, and Luo Mai. ServerlessLLM: Low-latency serverless inference for large language models. In *USENIX OSDI)*, pages 135–153, 2024.

[13] Prasanna Ganesan and Mukund Seshadri. On cooperative content distribution and the price of barter. In *25th International Conference on Distributed Computing Systems (ICDCS 2005), 6-10 June 2005, Columbus, OH, USA*, pages 81–90. IEEE Computer Society, 2005.

[14] Wei Gao, Yuheng Zhao, Dakai An, Tianyuan Wu, Lunxi Cao, Shaopan Xiong, Ju Huang, Weixun Wang, Siran Yang, Wenbo Su, et al. Rollpacker: Mitigating long-tail rollouts for fast, synchronous rl post-training. *arXiv preprint arXiv:2509.21009*, 2025.

[15] Michael R Garey and David S Johnson. *Computers and intractability*, volume 29. wh freeman New York, 2002.

[16] Juncheng Gu, Mosharaf Chowdhury, Kang G Shin, Yibo Zhu, Myeongjae Jeon, Junjie Qian, Hongqiang Liu, and Chuanxiong Guo. Tiresias: A {GPU} cluster manager for distributed deep learning. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 485–500, 2019.

[17] Shixiang Gu, Tim Lillicrap, Richard E. Turner, Zoubin Ghahramani, Bernhard Schölkopf, and Sergey Levine. Interpolated policy gradient: Merging on-policy and off-policy gradient estimation for deep reinforcement learning. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2017.

[18] Zhenyu Han, Ansheng You, Haibo Wang, Kui Luo, Guang Yang, Wenqi Shi, Menglong Chen, Sicheng Zhang, Zeshun Lan, Chunshi Deng, et al. Asyncflow: An asynchronous streaming rl framework for efficient llm post-training. *arXiv preprint arXiv:2507.01663*, 2025.

[19] Jingkai He, Tianjian Li, Erhu Feng, Dong Du, Qian Liu, Tao Liu, Yubin Xia, and Haibo Chen. History rhymes: Accelerating llm reinforcement learning with rhymerl. *arXiv preprint arXiv:2508.18588*, 2025.

[20] Ke Hong, Lufang Chen, Zhong Wang, Xiuhong Li, Qiuli Mao, Jianping Ma, Chao Xiong, Guanyu Wu, Buhe Han, Guohao Dai, et al. semi-pd: Towards efficient llm serving via phase-wise disaggregated computation and unified storage. *arXiv preprint arXiv:2504.19867*, 2025.

[21] Cunchen Hu, Heyang Huang, Liangliang Xu, Xusheng Chen, Jiang Xu, Shuang Chen, Hao Feng, Chenxi Wang, Sa Wang, Yungang Bao, et al. Inference without interference: Disaggregate llm inference for mixed downstream workloads. *arXiv preprint arXiv:2401.11181*, 2024.

[22] Jian Hu, Xibin Wu, Wei Shen, Jason Klein Liu, Zilin Zhu, Weixun Wang, Songlin Jiang, Haoran Wang, Hao Chen, Bin Chen, et al. Openrlhf: An easy-to-use, scalable and high-performance rlhf framework. *arXiv preprint arXiv:2405.11143*, 2024.

[23] Jingcheng Hu, Yinmin Zhang, Qi Han, Daxin Jiang, Xiangyu Zhang, and Heung-Yeung Shum. Open-reasoner-zero: An open source approach to scaling up reinforcement learning on the base model, 2025.

[24] Qinghao Hu, Shang Yang, Junxian Guo, Xiaozhe Yao, Yujun Lin, Yuxian Gu, Han Cai, Chuang Gan, Ana Klimovic, and Song Han. Taming the long-tail: Efficient reasoning rl training with adaptive drafter. *arXiv preprint arXiv:2511.16665*, 2025.

[25] Anant Singh Jain and Sheik Meeran. Deterministic job-shop scheduling: Past, present and future. *European Journal of Operational Research*, 113(2):390–434, 1999.

[26] Myeongjae Jeon, Shivaram Venkataraman, Amar Phanishayee, Junjie Qian, Wencong Xiao, and Fan Yang. Analysis of {Large-Scale}{Multi-Tenant}{GPU} clusters for {DNN} training workloads. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 947–960, 2019.

[27] Oleksii Kuchaiev, Jason Li, Huyen Nguyen, Oleksii Hrinchuk, Ryan Leary, Boris Ginsburg, Samuel Kriman, Stanislav Beliaev, Vitaly Lavrukhin, Jack Cook, et al. Nemo: a toolkit for building ai applications using neural modules. *arXiv preprint arXiv:1909.09577*, 2019.

[28] Kshiteej Mahajan, Arjun Balasubramanian, Arjun Singhvi, Shivaram Venkataraman, Aditya Akella, Amar Phanishayee, and Shuchi Chawla. Themis: Fair and efficient {GPU} cluster scheduling. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 289–304, 2020.

[29] Zhiyu Mei, Wei Fu, Kaiwei Li, Guangju Wang, Huanchen Zhang, and Yi Wu. Real: Efficient rlhf training of large language models with parameter reallocation. *arXiv preprint arXiv:2406.14088*, 2024.

[30] Philipp Moritz, Robert Nishihara, Stephanie Wang, Alexey Tumanov, Richard Liaw, Eric Liang, Melih Elibol, Zongheng Yang, William Paul, Michael I Jordan, et al. Ray: A distributed framework for emerging {AI} applications. In *13th USENIX symposium on operating systems design and implementation (OSDI 18)*, pages 561–577, 2018.

[31] Deepak Narayanan, Keshav Santhanam, Fiodar Kazhamiaka, Amar Phanishayee, and Matei Zaharia. {Heterogeneity-Aware} cluster scheduling policies for deep learning workloads. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 481–498, 2020.

[32] Long Ouyang, Jeffrey Wu, Xu Jiang, Diogo Almeida, Carroll Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, Alex Ray, et al. Training language models to follow instructions with human feedback. *Advances in neural information processing systems*, 35:27730–27744, 2022.

[33] Jiayi Pan, Xingyao Wang, Graham Neubig, Navdeep Jaitly, Heng Ji, Alane Suhr, and Yizhe Zhang. Training software engineering agents and verifiers with swe-gym, 2024.

[34] Pratyush Patel, Esha Choukse, Chaojie Zhang, Aashaka Shah, Íñigo Goiri, Saeed Maleki, and Ricardo Bianchini. Splitwise: Efficient generative llm inference using phase splitting. In *2024 ACM/IEEE 51st Annual International Symposium on Computer Architecture (ISCA)*, pages 118–132. IEEE, 2024.

[35] Aurick Qiao, Sang Keun Choe, Suhas Jayaram Subramanya, Willie Neiswanger, Qirong Ho, Hao Zhang, Gregory R Ganger, and Eric P Xing. Pollux: Co-adaptive cluster scheduling for goodput-optimized deep learning. In *15th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 21)*, 2021.

[36] Ruoyu Qin, Weiran He, Weixiao Huang, Yangkun Zhang, Yikai Zhao, Bo Pang, Xinran Xu, Yingdi Shan, Yongwei Wu, and Mingxing Zhang. Seer: Online context learning for fast synchronous llm reinforcement learning. *arXiv preprint arXiv:2511.14617*, 2025.

[37] Abhinav Rastogi, Albert Q Jiang, Andy Lo, Gabrielle Berrada, Guillaume Lample, Jason Rute, Joep Barmentlo, Karmesh Yadav, Kartik Khandelwal, Khyathi Raghavi Chandu, et al. Magistral. *arXiv preprint arXiv:2506.10910*, 2025.

[38] Salvatore Sanfilippo. Redis - the real-time data platform, 2009. Accessed: 2025-09-08.

[39] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017.

[40] Yizhou Shan, Yutong Huang, Yilun Chen, and Yiying Zhang. {LegoOS}: A disseminated, distributed {OS} for hardware resource disaggregation. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 69–87, 2018.

[41] Zhihong Shao, Peiyi Wang, Qihao Zhu, Runxin Xu, Junxiao Song, Xiao Bi, Haowei Zhang, Mingchuan Zhang, YK Li, Yang Wu, et al. Deepseekmath: Pushing the limits of mathematical reasoning in open language models. *arXiv preprint arXiv:2402.03300*, 2024.

[42] Weihang Shen, Mingcong Han, Jialong Liu, Rong Chen, and Haibo Chen. {XSched}: Preemptive scheduling for diverse {XPUs}. In *19th USENIX Symposium on Operating Systems Design and Implementation (OSDI 25)*, pages 671–692, 2025.

[43] Guangming Sheng, Chi Zhang, Zilingfeng Ye, Xibin Wu, Wang Zhang, Ru Zhang, Yanghua Peng, Haibin Lin, and Chuan Wu. Hybridflow: A flexible and efficient rlhf framework. In *Proceedings of the Twentieth European Conference on Computer Systems*, pages 1279–1297, 2025.

[44] Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. Megatron-lm: Training multi-billion parameter language models using model parallelism. *arXiv preprint arXiv:1909.08053*, 2019.

[45] ROLL Team. Reinforcement Learning Optimization for Large-Scale Learning: An Efficient and User-Friendly Scaling Library. *arXiv preprint arXiv:2506.06122*, 2025.

[46] Marcel Wagenländer, Guo Li, Bo Zhao, Luo Mai, and Peter Pietzuch. Tenplex: Dynamic parallelism for deep learning using parallelizable tensor collections. In *Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles (SOSP '24)*, 2024.

[47] Jinghui Wang, Shaojie Wang, Yinghan Cui, Xuxing Chen, Chao Wang, Xiaojiang Zhang, Minglei Zhang, Jiarong Zhang, Wenhao Zhuang, Yuchen Cao, et al. Seamlessflow: A trainer agent isolation rl framework achieving bubble-free pipelines via tag scheduling. *arXiv preprint arXiv:2508.11553*, 2025.

[48] Zhixin Wang, Tianyi Zhou, Liming Liu, Ao Li, Jiarui Hu, Dian Yang, Yinhui Lu, Jinlong Hou, Siyuan Feng, Yuan Cheng, et al. Distflow: A fully distributed rl framework for scalable and efficient llm post-training. *arXiv preprint arXiv:2507.13833*, 2025.

[49] Qizhen Weng, Lingyun Yang, Yinghao Yu, Wei Wang, Xiaochuan Tang, Guodong Yang, and Liping Zhang. Beware of fragmentation: Scheduling {GPU-Sharing} workloads with fragmentation gradient descent. In *2023 USENIX Annual Technical Conference (USENIX ATC 23)*, pages 995–1008, 2023.

[50] Wencong Xiao, Romil Bhardwaj, Ramachandran Ramjee, Muthian Sivathanu, Nipun Kwatra, Zhenhua Han, Pratyush Patel, Xuan Peng, Hanyu Zhao, Quanlu Zhang, et al. Gandiva: Introspective cluster scheduling for deep learning. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 595–610, 2018.

[51] Wencong Xiao, Shiru Ren, Yong Li, Yang Zhang, Pengyang Hou, Zhi Li, Yihui Feng, Wei Lin, and Yangqing Jia. {AntMan}: Dynamic scaling on {GPU} clusters for deep learning. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 533–548, 2020.

[52] Ran Yan, Youhe Jiang, Tianyuan Wu, Jiaxuan Gao, Zhiyu Mei, Wei Fu, Haohui Mai, Wei Wang, Yi Wu, and Binhang Yuan. AReaL-Hex: Accommodating asynchronous rl training over heterogeneous gpus. *arXiv preprint arXiv:2511.00796*, 2025.

[53] An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengen Huang, Chenxu Lv, Chujie Zheng, Dayiheng Liu, Fan Zhou, Fei Huang, Feng Hu, Hao Ge, Haoran Wei, Huan Lin, Jialong Tang, Jian Yang, Jianhong Tu, Jianwei Zhang, Jianxin Yang, Jiaxi Yang, Jing Zhou, Jingren Zhou, Junyang Lin, Kai Dang, Keqin Bao, Kexin Yang, Le Yu, Lianghao Deng, Mei Li, Mingfeng Xue, Mingze Li, Pei Zhang, Peng Wang, Qin Zhu, Rui Men, Ruize Gao, Shixuan Liu, Shuang Luo, Tianhao Li, Tianyi Tang, Wenbiao Yin, Xingzhang Ren, Xinyu Wang, Xinyu Zhang, Xuancheng Ren, Yang Fan, Yang Su, Yichang Zhang, Yinger Zhang, Yu Wan, Yuqiong Liu, Zekun Wang, Zeyu Cui, Zhenru Zhang, Zhipeng Zhou, and Zihan Qiu. Qwen3 technical report, 2025.

[54] Shunyu Yao, Howard Chen, John Yang, and Karthik Narasimhan. Webshop: Towards scalable real-world web interaction with grounded language agents. *Advances in Neural Information Processing Systems*, 35:20744–20757, 2022.

[55] Zhewei Yao, Reza Yazdani Aminabadi, Olatunji Ruwase, Samyam Rajbhandari, Xiaoxia Wu, Ammar Ahmad Awan, Jeff Rasley, Minjia Zhang, Conglong Li, Connor Holmes, et al. Deepspeed-chat: Easy, fast and affordable rlhf training of chatgpt-like models at all scales. *arXiv preprint arXiv:2308.01320*, 2023.

[56] Minchen Yu, Rui Yang, Chaobo Jia, Zhaoyuan Su, Sheng Yao, Tingfeng Lan, Yuchen Yang, Yue Cheng, Wei Wang, Ao Wang, et al. λScale: Enabling fast scaling for serverless large language model inference. *arXiv preprint arXiv:2502.09922*, 2025.

[57] Qiying Yu, Zheng Zhang, Ruofei Zhu, Yufeng Yuan, Xiaochen Zuo, Yu Yue, Weinan Dai, Tiantian Fan, Gaohong Liu, Lingjun Liu, et al. Dapo: An open-source llm reinforcement learning system at scale. *arXiv preprint arXiv:2503.14476*, 2025.

[58] Dingyan Zhang, Haotian Wang, Yang Liu, Xingda Wei, Yizhou Shan, Rong Chen, and Haibo Chen. {BlitzScale}: Fast and live large model autoscaling with o (1) host caching. In *19th USENIX Symposium on Operating Systems Design and Implementation (OSDI 25)*, pages 275–293, 2025.

[59] Pengfei Zheng, Rui Pan, Tarannum Khan, Shivaram Venkataraman, and Aditya Akella. Shockwave: Fair and efficient cluster scheduling for dynamic adaptation in machine learning. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 703–723, 2023.

[60] Yinmin Zhong, Shengyu Liu, Junda Chen, Jianbo Hu, Yibo Zhu, Xuanzhe Liu, Xin Jin, and Hao Zhang.

{DistServe}: Disaggregating prefill and decoding for goodput-optimized large language model serving. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, pages 193–210, 2024.

[61] Yinmin Zhong, Zili Zhang, Xiaoniu Song, Hanpeng Hu, Chao Jin, Bingyang Wu, Nuo Chen, Yukun Chen, Yu Zhou, Changyi Wan, et al. Streamrl: Scalable, heterogeneous, and elastic rl for llms with disaggregated stream generation. *arXiv preprint arXiv:2504.15930*, 2025.

[62] Yinmin Zhong, Zili Zhang, Bingyang Wu, Shengyu Liu, Yukun Chen, Changyi Wan, Hanpeng Hu, Lei Xia, Ranchen Ming, Yibo Zhu, et al. Optimizing rlhf training for large language models with stage fusion. *arXiv preprint arXiv:2409.13221*, 2024.

[63] Kan Zhu, Yufei Gao, Yilong Zhao, Liangyu Zhao, Gefei Zuo, Yile Gu, Dedong Xie, Tian Tang, Qinyu Xu, Zihao Ye, Keisuke Kamahori, Chien-Yu Lin, Ziren Wang, Stephanie Wang, Arvind Krishnamurthy, and Baris Kasikci. Nanoflow: Towards optimal large language model serving throughput, 2025.

[64] Ruidong Zhu, Ziheng Jiang, Chao Jin, Peng Wu, Cesar A. Stuardo, Dongyang Wang, Xinlei Zhang, Huaping Zhou, Haoran Wei, Yang Cheng, Jianzhe Xiao, Xinyi Zhang, Lingjun Liu, Haibin Lin, Li-Wen Chang, Jianxi Ye, Xiao Yu, Xuanzhe Liu, Xin Jin, and Xin Liu. Megascale-infer: Serving mixture-of-experts at scale with disaggregated expert parallelism, 2025.

[65] Yonghao Zhuang, Lianmin Zheng, Zhuohan Li, Eric Xing, Qirong Ho, Joseph Gonzalez, Ion Stoica, Hao Zhang, and Hexu Zhao. On optimizing the communication of model parallelism. *Proceedings of Machine Learning and Systems*, 5:526–540, 2023.

Appendix

Proof of Meta-iteration Schedule's Optimality. To show the utilization-optimality of the meta-iteration schedule for unsaturated groups, we first formally define the utilization of a group with meta-iteration time T_{meta}, where its rollout (U_R) and training utilization (U_T) are:

$$U_R = \frac{\sum_{j \in J_G} T_j^{\text{roll}}}{T_{\text{meta}}}, \quad \text{and} \quad U_T = \frac{\sum_{j \in J_G} T_j^{\text{train}}}{T_{\text{meta}}}.$$

Since the meta-iteration schedule repeats, it suffices to analyze a single meta-iteration where every job's rollout and training phase executes once. Recall that $T_G^{\text{cycle}} = \max_{j \in J_G} T_j^{\text{solo}}$ is the solo iteration time of the slowest job, and let j_1 be the job attaining this maximum so that $T_G^{\text{cycle}} = T_{j_1}^{\text{roll}} + T_{j_1}^{\text{train}}$. For simplicity, the following proof considers only one rollout node and one training node, while the optimality generalizes to multi-node settings. By the definition of a non-overloaded group, we have

$$T_G^{\text{cycle}} = T_{j_1}^{\text{roll}} + T_{j_1}^{\text{train}} \geq \max\left(\sum_{j \in J_G} T_j^{\text{roll}}, \sum_{j \in J_G} T_j^{\text{train}}\right).$$

This directly implies that the total rollout work of all other jobs can fit within j_1's training phase, and vice-versa:

$$\sum_{j \in J_G \setminus j_1} T_j^{\text{roll}} \leq T_{j_1}^{\text{train}}, \quad \sum_{j \in J_G \setminus j_1} T_j^{\text{train}} \leq T_{j_1}^{\text{roll}}.$$

Consequently, job j_1's execution time decides the entire schedule's cycle time, yielding the cycle time as $T_{j_1}^{\text{solo}}$.

Conversely, any schedule that repeats a job is sub-optimal for a non-overloaded group because it extends the meta-iteration's duration but adds proportionally less work, leading to a net decrease in resource utilization. We show this by trying to add a repetition of any job k to the meta-iteration schedule. Repeating job k adds work to both pools and necessarily prolongs the cycle time by at least T_k^{solo} due to waiting for the longest j_1, causing the new utilization U' lower than the original U. We can bound the change in utilization, $\Delta U = U' - U$, as follows. The change for the rollout and the training pool, ΔU_R and ΔU_T, is bounded by:

$$\Delta U_R = U_R' - U_R \leq \frac{\sum_{j \in J_G} T_j^{\text{roll}} + T_k^{\text{roll}}}{T_{j_1}^{\text{solo}} + T_k^{\text{roll}}} - \frac{\sum_{j \in J_G} T_j^{\text{roll}}}{T_{j_1}^{\text{solo}}},$$

$$\Delta U_T = U_T' - U_T \leq \frac{\sum_{j \in J_G} T_j^{\text{train}} + T_k^{\text{train}}}{T_{j_1}^{\text{solo}} + T_k^{\text{train}}} - \frac{\sum_{j \in J_G} T_j^{\text{train}}}{T_{j_1}^{\text{solo}}}.$$

Therefore, the total change in utilization is

$$\begin{aligned}
\Delta U &= \Delta U_R + \Delta U_T = U_R' - U_R + U_T' - U_T \\
&\leq \frac{T_k^{\text{train}} T_{j_1}^{\text{solo}} - T_k^{\text{solo}} \sum_{j \in J_G} T_j^{\text{train}} + T_{j_1}^{\text{solo}} T_k^{\text{roll}} - T_k^{\text{solo}} \sum_{j \in J_G} T_j^{\text{roll}}}{T_{j_1}^{\text{solo}} (T_{j_1}^{\text{solo}} + T_k^{\text{solo}})} \\
&= \frac{T_k^{\text{solo}} (T_{j_1}^{\text{solo}} - \sum_{j \in J_G} T_j^{\text{solo}})}{T_{j_1}^{\text{solo}} (T_{j_1}^{\text{solo}} + T_k^{\text{solo}})} \leq 0.
\end{aligned}$$

This proves that any repetition degrades overall utilization, making the round-robin schedule utilization-optimal for non-overloaded groups.

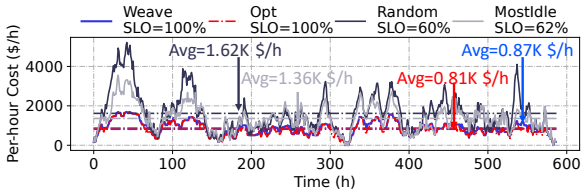


Figure 20: **[Simulation]** Cost-effectiveness and SLO attainment under a realistic mixed workload. (Workload: mixed, SLO∼`Unif(1,2)`, max group size=5).

Table 6: Job configurations used in the simulation. Each job is defined by its rollout time (T_{roll}) and train time (T_{train}), which are drawn from different uniform distributions.

Workload	Size	Rollout Time (T_{roll})	Train Time (T_{train})
Balanced (BL)	Small	`Unif(50, 100)`	`Unif(50, 100)`
	Medium	`Unif(100, 200)`	`Unif(100, 200)`
	Large	`Unif(200, 300)`	`Unif(200, 300)`
Rollout-Heavy (RH)	Small	`Unif(100, 200)`	`Unif(25, 50)`
	Medium	`Unif(200, 400)`	`Unif(50, 100)`
	Large	`Unif(400, 600)`	`Unif(100, 200)`
Train-Heavy (TH)	Small	`Unif(25, 50)`	`Unif(100, 200)`
	Medium	`Unif(50, 100)`	`Unif(200, 400)`
	Large	`Unif(100, 200)`	`Unif(400, 600)`
Mixed	All	All	All

Workloads for Simulation. Table 6 details the job profiles used in § 6.5.

[Simulation] End-to-end Performance. This part reports the end-to-end performance of WEAVE's scheduler under a realistic setting with Mixed workloads with heterogeneous job SLOs. As illustrated in Figure 20, WEAVE, achieves 100% SLO attainment by design, matching the theoretical Offline Optimal scheduler. In contrast, the heuristic-based baselines struggle significantly: the Random and Greedy (Most-Idle) schedulers meet the SLOs for only 60% and 62% of jobs, respectively. This is because WEAVE's cost-based optimization (§ 4.3) explicitly prunes any placements that would violate a job's SLO by assigning it an infinite cost.

In terms of resource efficiency, the average cost of WEAVE is at 0.87K $/h, only 1.06× higher than the Offline Optimal. In contrast, the baselines are far more expensive. The Random and Greedy strategies incur average costs of 1.62K and 1.36K $/h, respectively (1.97× and 1.66× of optimal). This high cost is driven by their sub-optimal group partitioning, where their costs spike to over 5K $/h by scaling out to 1400 GPUs. WEAVE, however, manages load with a peak cost of only ∼1.8K $/h, requiring at most 504 GPUs, demonstrating its effectiveness at identifying SLO-aware, cost-efficient placements in a complex, online setting.

ARTIFACT EVALUATED
USENIX
AVAILABLE

RLinf: Flexible and Efficient Large-Scale Reinforcement Learning via Macro-to-Micro Flow Transformation

Chao Yu[1], Yuanqing Wang[3,4], Zhen Guo[3], Hao Lin[3], Si Xu[3], Hongzhi Zang[1], Quanlu Zhang[3], Yongji Wu[5], Chunyang Zhu[3], Junhao Hu[3], Zixiao Huang[1,3], Mingjie Wei[2], Yuqing Xie[1], Ke Yang[2], Bo Dai[6,3], Zhexuan Xu[1], Jiakun Du[1], Xiangyuan Wang[4,3], Xu Fu[3], Letong Shi[3], Zhihao Liu[2], Kang Chen[4,2], Weilin Liu[3], Gang Liu[1], Boxun Li[3], Jianlei Yang[6], Zhi Yang[4], Guohao Dai[7,3], Yu Wang[1]*

[1]Tsinghua University [2]Zhongguancun Academy [3]Infinigence AI
[4]Peking University [5]UC Berkeley [6]Beihang University [7]Shanghai Jiaotong University
GitHub Repo: https://github.com/RLinf/RLinf

Abstract

Reinforcement learning (RL) has demonstrated immense potential in advancing artificial general intelligence, agentic intelligence, and embodied intelligence. However, the inherent heterogeneity and dynamicity of RL workflows often lead to low hardware utilization and slow training on existing systems. In this paper, we present RLinf, a high-performance RL training system based on our key observation that the major roadblock to efficient RL training lies in *system flexibility*. To maximize flexibility and efficiency, RLinf is built atop a novel RL system design paradigm called *macro-to-micro flow transformation* (M2Flow), which automatically breaks down high-level, easy-to-compose RL workflows at both the temporal and spatial dimensions, and recomposes them into optimized execution flows. Supported by RLinf worker's adaptive communication capability, we devise *context switching* and *elastic pipelining* to realize M2Flow transformation, and a profiling-guided scheduling policy to generate optimal execution plans. Extensive evaluations on both reasoning RL and embodied RL tasks demonstrate that RLinf consistently outperforms state-of-the-art systems, achieving 1.07×∼2.43× speedup in end-to-end training throughput.

1 Introduction

The rapid progress of large language models (LLMs) has reached a point where further scaling the model alone yields diminishing returns. To push intelligence beyond pretraining, reinforcement learning (RL) has emerged as a crucial paradigm. Recent advances such as RLHF [8, 38], GRPO [49], and RL for embodied agents [24, 25] and Deep Research [35, 69] all rely on RL to align LLMs with human preferences, improve reasoning, and enable autonomous interaction with complex environments. OpenAI and others predict that RL workloads will soon consume more computational resources than LLM pretraining [37], making RL training efficiency the most critical system concern.

*Corresponding author. Email: yu-wang@tsinghua.edu.cn

However, efficient RL training for various scenarios such as reasoning, agentic and embodiment at the scale of modern large models is challenging, which combines highly heterogeneous components with diverse workload characteristics and resource demands, such as LLM generation, inference and training, reward models, critic models, agent tooling and embodied environment simulators. For instance, LLM training consumes more accelerator (e.g., GPU) memory than LLM generation and inference (prefill-only generation) to maintain gradients and optimizer states, while LLM generation shows high dynamicity in response lengths, leading to low accelerator utilization. Moreover, components like LLM training support diverse parallelization strategies (e.g., data, tensor, pipeline parallelism), whereas others scale only via instance replication and may yield computation workloads distinct from common tensor computation in LLM, e.g., embodied simulators [10, 54] that require CPU for physics simulation and GPU graphics pipeline for 3D rendering.

Single execution mode of existing RL training systems fails to capture this diversity, leading to suboptimal efficiency. Collocated execution, where components sequentially occupy accelerators [51], suffers from the long-tail problem due to varying generation lengths, leaving accelerators idle. Disaggregated pipelining, where components run concurrently on separate accelerators with pipelining [13], mitigates the long-tail issue but introduces memory and computation imbalance (§2.2). Neither mode is universally optimal. Many RL workloads demand hybrid scheduling of the components to maximize efficiency, i.e., mixing collocation and pipelining in a more flexible way. However, supporting such flexible execution modes for a single programmed workflow is a significant challenge, as they often require different program structures and communication patterns. Also, identifying the right scheduling for a given workflow usually requires considerable manual tuning.

In this paper, we present RLinf, an RL training system that maximizes system flexibility to achieve efficient execution of a logically programmed RL workflow. At its core is a new paradigm called *macro-to-micro flow transformation*

(M2Flow), i.e., macro logical flow with micro execution flow, which decouples the logical programming of RL workflows from their physical execution planning. With M2Flow, developers program RL workflows imperatively, using a natural programming interface to define how components communicate and synchronize at a coarse granularity. RLinf then automatically transforms this logical flow into a fine-grained execution plan tailored to the workload and hardware at both spatial and temporal dimensions. By decoupling program semantics from execution modes, M2Flow lets developers preserve clean, intuitive workflows while the system explores a vast scheduling space, including temporal multiplexing, spatial pipelining, and hybrid scheduling.

RLinf achieves this through three key mechanisms. First, a *worker* abstraction that encapsulates each RL component for flexible placement, and built-in adaptive communication that allows direct and efficient communication between components regardless of worker and data placement. Second, elastic pipelining and automatic context switching that enable M2Flow transformation and expand the scheduling space, achieving pipeline granularity tuning and temporal accelerator multiplexing without modifying the logical workflow. Third, a profiling-guided scheduling policy that automatically selects efficient execution modes, balancing utilization across heterogeneous components. Together, these capabilities deliver high flexibility, efficiency, and programmability.

We implement RLinf using Ray for cluster management and worker process launch on remote nodes. Apart from the core mechanisms, RLinf also provides rich support for common RL components, algorithms, and models to facilitate RL workflow programming (§4). We extensively evaluate RLinf on both reasoning and embodied RL training across diverse models (e.g., Qwen2.5-1.5/7/32B [40], Qwen3 [59], OpenVLA [24], OpenVLA-OFT [23]) and varying scales. Our evaluation shows that RLinf improves end-to-end throughput (i.e., tokens per second) by up to 1.7× compared to state-of-the-art RL training systems in reasoning RL, and by up to 2.43× in embodied RL. We have open-sourced the full codebase of RLinf to accelerate RL innovations in the LLM era.

This paper makes the following contributions:

- Analysis of representative RL algorithms and scenarios, identifying characteristics of modern RL workflows and highlighting inefficiencies in current systems.
- A novel RL system paradigm M2Flow that decouples logical workflow programming from execution planning, enabling intuitive programming with flexible execution.
- A cohesive set of mechanisms, i.e., worker abstraction, elastic pipelining, context switching, adaptive communication, and a profiling-guided scheduler, that jointly realize M2Flow and enable efficient RL training.
- Comprehensive evaluations of RLinf across various RL workloads show that RLinf significantly improves efficiency and flexibility compared to existing approaches.

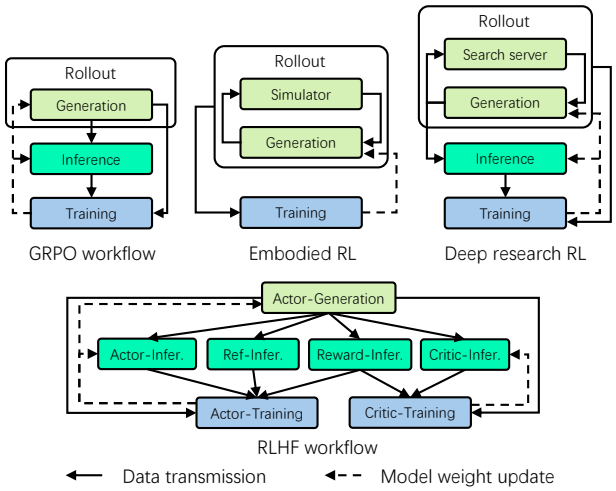


Figure 1: Diverse RL workflows in various scenarios.

2 Background and Motivation

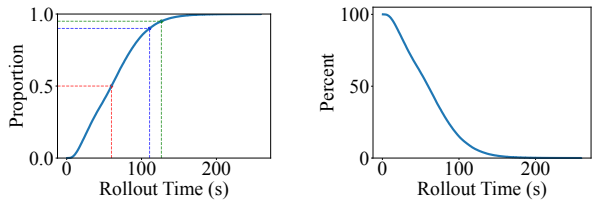
2.1 RL Workflows in LLM Era

Various RL Algorithms and Scenarios. With the slowdown of scaling gains in large language models, reinforcement learning has become increasingly important for advancing LLM intelligence. Unlike traditional RL, RL in the LLM era often involves multiple LLMs in the loop. Given the scale of modern models (tens to hundreds of billions of parameters), fitting RL training into available accelerators (e.g., GPUs) is already challenging. Figure 1 illustrates four representative RL workflows across different scenarios and algorithms.

The simplest is GRPO [49], an RL algorithm designed to reduce reliance on reward models. It involves a single LLM that generates multiple responses, e.g., 8, for a query (i.e., `Generation`), computes logarithmic probabilities for these responses (i.e., `Inference`), and uses the results as training data to update the same model (i.e., `Training`). The updated weights are then synchronized back for inference and generation, completing one training iteration.

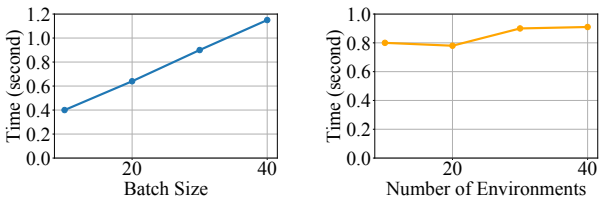
In contrast, the RLHF [38] workflow adopts PPO [48], resulting in a more complex design involving four LLMs. The actor model serves as the core policy, generating responses to queries. The reference model remains fixed to constrain the actor from drifting too far from its initialization. The reward model assigns scalar rewards to generated responses, while the critic model estimates expected rewards to stabilize training. Actor and critic are trainable, whereas reference and reward models are frozen. These components interact closely, as shown in the figure.

Beyond algorithms, RL workflows also vary by application scenario. In embodied intelligence [24,25], RL relies on simulators that simulate the physical world. An LLM interacts with the simulator by generating actions and receiving feedback,



(a) CDF of response time. (b) Unfinished responses.

Figure 2: The distribution of response lengths and the number of unfinished responses over time in the generation phase of a math RL experiment.



(a) Generation time. (b) Simulator time.

Figure 3: The execution time of generation and simulator with different batch sizes respectively, batch size in simulator is the number of environments.

producing trajectories that serve as training data. Similarly, in Deep Research [35, 69], RL improves model performance through interaction with a search server that retrieves online information. The resulting rollout results are fed into training, while inference follows the GRPO workflow.

Characteristics of RL Workflows. RL workflows consist of heterogeneous components with distinct demands on GPU memory, computation cores, accelerator types, and parallelization strategies. For example, training requires substantially more GPU memory than inference to maintain gradients and optimizer states. Unlike training, generation often underutilizes GPU cores, as its matrix and vector multiplications are bottlenecked by memory bandwidth. Some components (e.g., simulator) run on CPUs, or use GPUs for non-tensor computations (e.g., 3D rendering). Parallelization also differs significantly, e.g., LLM training exploits data, tensor, and pipeline parallelism, whereas simulators typically scale only through multiple instances. Maximizing overall utilization across such heterogeneous components is a great challenge.

Further, RL workflows exhibit complex dependencies, primarily through data flow and weight updates. Data flow can occur at different granularities, e.g., per response between generation and inference, or at least a micro-batch of responses between generation and training. Some workflows even introduce cyclic data flows, such as in embodied RL and Deep Research (Figure 1), which further complicates coordination. Weight updates, in contrast, act as barriers that synchronize generation and training. These complex dependencies require more fine-grained scheduling.

2.2 Inefficiencies in Diverse RL Workflows

We analyzed different RL workflows to identify the source of inefficiencies as follows.

Dynamicity in Rollout Wastes Computation. The rollout phase is inherently dynamic. Lengths fluctuate across responses of the same query, and even more so across different queries. Embodied tasks such as grasping can take different numbers of steps. In deep research, generating a report may involve varying numbers of search interactions. Since rollouts are executed in batches, these variations create a long-tail problem where a few slow queries block the entire phase from proceeding to inference or training. The problem commonly exists in the collocated mode (e.g., veRL [51]), where generation, inference, and training sequentially share GPUs by swapping between CPU and GPU memory. We conducted an experiment of a 7B math reasoning RL training [40] on 8 nodes with 8 H100 GPUs each, as shown in Figure 2. The number of unfinished responses quickly shrinks to less than 5%, where a very small set of long-tail responses stalls the generation, leaving many GPUs underutilized or idle. Scaling out with more GPUs worsens the problem as idle time grows.

Pipelining alleviates this by allocating fewer GPUs to generation and leaving the rest for inference and training. In this setup, inference and training start once partial samples are ready. However, pipelining introduces its own inefficiency since inference and training must wait for the first batch to be generated. Thus, neither approach is universally optimal, and supporting both collocated mode and pipelining within a single framework remains a significant challenge.

Simple Execution Modes Cannot Fit Diverse Components. Collocated and pipelined modes are two extremes, i.e., all components on the same GPUs versus fully disaggregated GPUs. Some RL workloads, however, do not fit neatly into either mode. Their diverse component characteristics require more flexible orchestration. Take embodied RL (Figure 1) as an example. Figure 3 shows computation profiles of generation and simulator. The execution time of simulator increases slightly with the number of environments and its GPU utilization remains low (i.e., <24%), while its memory usage grows linearly with the number of environments. In contrast, generation scales linearly in both runtime and memory with batch size, while keeping GPU cores highly utilized (i.e., >70%). Training consumes more memory, but its execution time is only one-third as long as generation's.

This profile rules out simple execution modes. The simulator should scale with as many parallel environments as possible to reduce its total runtime. However, this prevents collocation with generation due to memory contention. A better choice is running on disaggregated GPUs with pipelining for higher efficiency. Training, in contrast, would waste com-

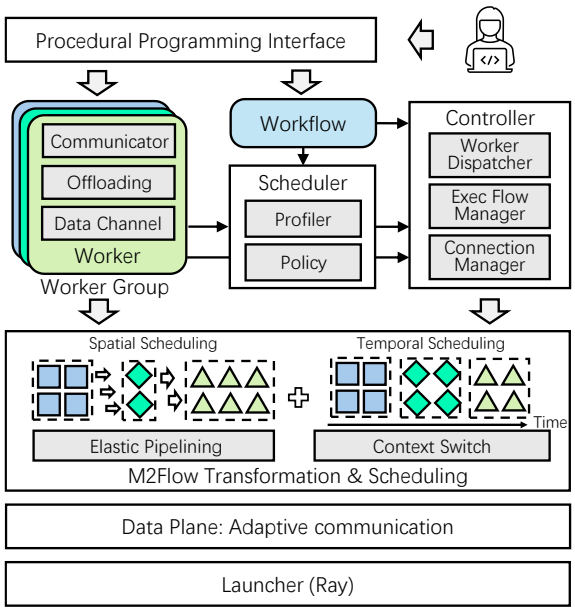


Figure 4: The architecture of RLinf.

pute if fully disaggregated, so it should share GPUs. After rollout, simulator and generation are swapped to CPU, and training takes over the GPU. The result is a hybrid mode that combines collocation and pipelining to balance efficiency.

Identifying Suitable Orchestration is Challenging. The orchestration of the hybrid mode depends on the analysis of the components. However, finding the most suitable orchestration for a given RL workflow is challenging, as the characteristics are diverse and the dependencies are complex. Manually enumerating options is tedious, time-consuming, and risks overlooking better choices. Moreover, no clear guidelines exist to identify the most suitable orchestration.

2.3 Flexibility as a Key to Efficiency

Maximizing computation efficiency for an RL workload requires flexible orchestration that aligns with component and workflow characteristics. However, adjusting the execution mode without changing the programmed workflow is challenging. Collocated and disaggregated pipelining modes differ significantly. Collocated mode operates at coarse-grained, phase-level execution. Each phase starts after the previous phase is complete. However, disaggregated pipelining runs at fine-grained, batch-level with precise timing to minimize pipeline bubbles. Mixing these modes further increases complexity. We advocate a system design that bridges this gap, enabling RL developers to maintain an intuitive, logically organized workflow while achieving high execution efficiency with flexible execution modes.

3 RLinf Design

3.1 Overview

In pursuit of efficient, flexible, and intuitive RL systems, we propose a new design paradigm termed M2Flow, i.e., *macro logical flow executed with micro execution flow*. In this paradigm, developers program the complex RL workflow by imperatively specifying the logical communication flow among the RL components at a coarse granularity (macro logical flow), while the system automatically transforms the workflow into a fine-grained execution flow (micro execution flow). Essentially, M2Flow decouples programmable code logic from the physical execution and scheduling of the individual RL components, so as to maximize the efficiency while minimizing the programming complexity.

Figure 4 shows the architecture of RLinf that realizes M2Flow. As illustrated, RLinf provides an easy-to-use procedural programming interface for users to construct RL workflows imperatively, which describe the data communication and interaction among the RL components. RL components are then encapsulated as *workers* in RLinf, each implementing the main logic of this component. Workers are equipped with communication functionality to freely communicate with each other, as well as a resource offloading mechanism to enable temporal multiplexing of hardware resources. This worker abstraction enables RLinf to retain substantial scheduling flexibility at both the spatial and temporal dimensions, while following the procedural workflow. Spatial scheduling assigns workers to accelerators, temporal scheduling determines their execution periods, and spatio–temporal scheduling governs their pipelined execution granularity.

To produce a desirable micro execution flow across these scheduling dimensions, the core of the scheduler module is a profiling-guided scheduling policy, which utilizes runtime profiling of worker characteristics to search for the optimal execution mode of each worker. Based on the determined execution mode, the Controller assigns workers to accelerators, manages inter-worker connections, and orchestrates the execution flow by dispatching function invocations to the corresponding workers. To realize this, two mechanisms named elastic pipelining and context switch are devised to enable spatial and temporal orchestration of workers, respectively. Adaptive communication utilities such as point-to-point communication and data channel act as the data plane to support scalable worker interactions. The entire system leverages Ray [33] to remotely launch and control workers.

3.2 Workflow Construction Interface

The design philosophy of RLinf is to maximize system flexibility to achieve high efficiency, which is also the guideline for RLinf's programming interface design. To this end, unlike traditional graph-based declarative programming [1] that sac-

```python
class Worker:
  def send(self, obj, dst, async_op)
  def recv(self, src, async_op)
  @overload
  def onload(self)
  @overload
  def offload(self)

class RolloutWorker(Worker):
  def generate(self, in_channel, out_channel):
    with self.device_lock:
      batch_size = 0
      while batch_size < self.total_batch_size:
        batch = in_channel.get()
        res = self.model.gen(batch)
        out_channel.put(res)
        batch_size += batch.size
```

(a) A typical RLinf worker.

```python
class RLWorkflowRunner:
  def __init__(self):
    cluster = Cluster(num_nodes=4, devices_per_node=8)
    self.rollout_group = RolloutWorker.launch(cluster)
    self.actor_group = ActorWorker.launch(cluster)
    self.data_ch = Channel.create("Data")
    self.rollout_ch = Channel.create("Rollout")
  def run(self):
    for batch in batch_iterator:
      self._update_rollout_weights()
      self.data_ch.put(batch)
      self.rollout_group.generate(
        in_channel=self.data_ch,
        out_channel=self.rollout_ch
      )
      self.actor_group.train(self.rollout_ch).wait()
```

(b) A workflow runner example.

Figure 5: RLinf workflow programming interface.

rifices control flow flexibility, debuggability and transparency for optimization opportunity, RLinf adopts a procedural programming paradigm that enables developers to flexibly express workflows imperatively. An example workflow based on this interface is shown in Figure 5. As shown, an RL program based on RLinf consists of two parts: (1) worker programs that define the logic of each RL component (e.g., simulator, LLM generation, actor training), and (2) a workflow runner that orchestrates the overall workflow by invoking the workers' core functions and defining inter-worker interactions.

Figure 5a demonstrates a typical worker implementation atop RLinf. The base `Worker` class provides communication primitives such as `send` and `recv` for inter-worker communication. All workers that inherit from the base class automatically gain the capability to communicate with other workers (§3.5), which is also the foundation for higher-level communication facilities like the data channel. To manage limited device resources like GPU memory, all workers are required to implement resource management functions (`onload` and `offload`) for acquiring and releasing the resources.

After implementing workers, developers can compose the overall RL workflow as in Figure 5b. First, the runner launches workers on a cluster of nodes and devices in an SPMD manner. The scheduler module decides the placement of each worker process before its launch, which can also be manually specified if desired. All processes of the same worker are collectively managed via the `WorkerGroup` abstraction of RLinf (e.g., `rollout_group`), which automatically assumes the public functions defined in the worker class and dispatches them to all (or a selective portion) of the worker processes if invoked. The functions of a `WorkerGroup` are inherently asynchronous and return a result handle, whose `wait` primitive provides synchronization barriers, enabling computation at a certain data granularity. For example, in GRPO training, rollout can proceed per query, but normalization must aggregate all responses for a query, which pauses the pipeline at this step until normalization completes. The core facility for connecting the data flow among distributed worker groups is the data channel (detailed in §3.5), which decouples the control and data flows of dependent components, enabling highly flexible programming while exposing a broad optimization space, as shown below.

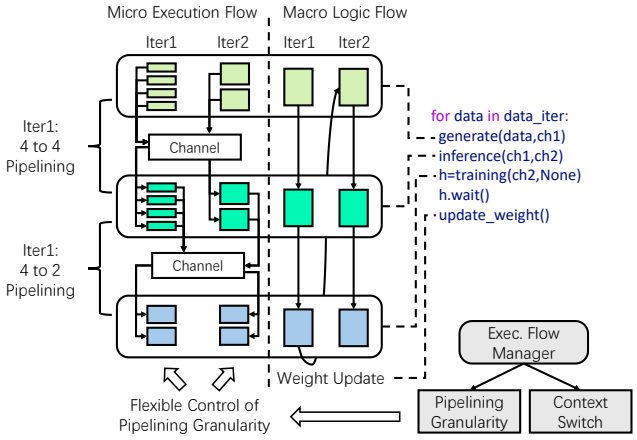


Figure 6: The M2Flow execution logic.

3.3 M2Flow Transformation

The programming interface shown in Figure 5b offers a flow-like programming model, with which developers describe the high-level, logical control and data flows among workers. Having obtained the logical flows, RLinf follows the M2Flow paradigm to transform the logical flows (i.e., how should the workers run) into the concrete execution flow, i.e., where (spatial) and when (temporal) should the workers run. In this section, we focus on two enabling mechanisms of M2Flow transformation and flexible scheduling—elastic pipelining and context switching. In §3.4, we will describe the scheduling policy that determines the optimal execution flow.

Based on the programmed workflow, the key idea of M2Flow transformation is to control the spatial and temporal scheduling of workers by throttling their data processing granularity and concurrent resource accesses, respectively.

Spatial Scheduling via Elastic Pipelining. For spatial scheduling, workers can be executed in a pipelined manner with a different number of accelerators/devices. To maximize

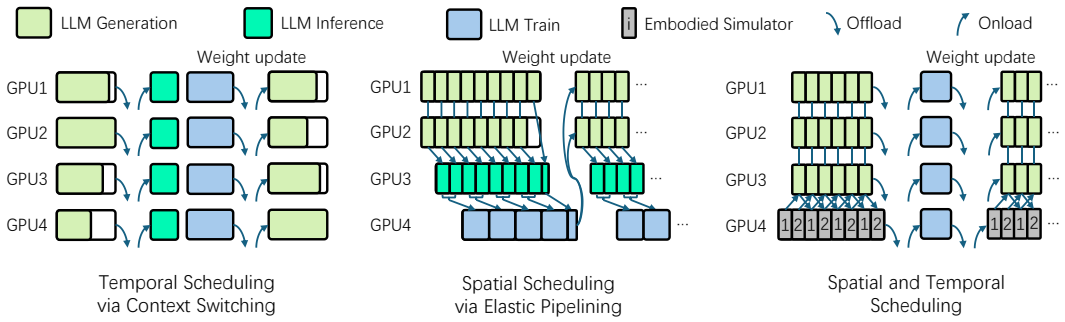


Figure 7: Spatial and temporal scheduling of workers.

pipeline flexibility, RLinf introduces elastic pipelining to enable workers to flexibly process data at different granularity with the given device resources. Elastic pipelining builds upon our insight that in RL training and agentic scenarios, most workers follow the SPMD pattern, allowing execution across varying batch sizes. For instance, LLM serving engines like SGLang and vLLM can process a single prompt or a list of prompts at a time, and inference (i.e., prefill-only computation of a model) similarly supports single-batch or multi-batch execution. This flexibility enables the Execution Flow Manager of RLinf to achieve flexible pipelining of a worker task via dynamic data granularity—output data can be forwarded once a configured size of data batch is ready, allowing downstream workers to start earlier with smaller batches or later with larger batches. Notably, the scheduling space is further affected by individual worker's internal computation semantics. For example, training workers operate with both the micro-batch and global-batch concepts—the micro-batch defines forward/backward units, while the global-batch determines when model updates occur.

Temporal Scheduling via Automatic Context Switching. Beyond spatial scheduling, RLinf also supports natural temporal multiplexing of devices via automatic context switching, further expanding the scheduling space. Context switching enables workers that cannot co-reside in the same accelerators with limited device resources (e.g., GPU memory) to share devices by executing sequentially. In RLinf, this is realized via a distributed device lock of the data channel facility, i.e., `device_lock` as shown in Figure 5a. This lock serves as the primitive to throttle concurrent resource access by multiple workers that are on the same devices and have data flow dependencies, i.e., producers and consumers of the same channel. Before using device resources, a worker must acquire the lock, whose state is globally consistent to all workers and can only be changed atomically, thus ensuring exclusive access to the resources. Upon lock acquiring, the worker's device resources are automatically loaded by calling the `onload` function of the worker if the resources have been offloaded. After completing its task, the worker releases the lock and offloads its resources via the `offload` function to free up device resources for child workers in the workflow. Unlike a conventional lock, the device lock leverages the data channel's data dependency information to define lock acquiring priority, i.e., child workers that depend on the outputs of parent workers can only acquire the lock after the parents have enqueued data and released the lock, so as to avoid lock contention and deadlock. Also, it utilizes device placement information from the Controller to avoid unnecessary resource loading and offloading when workers are placed on different devices.

M2Flow Execution. Figure 6 shows how M2Flow manages execution. The workflow is written imperatively, i.e., a for loop iterating over main logic, with three workers, i.e., rollout, inference, and training. A rollout task such as `generate(data, ch1)` processes data and enqueues results to channel `ch1`. The Execution Flow Manager can divide the input data into smaller chunks, allowing workers to process outputs at a smaller data granularity. Alternatively, tasks can be coalesced into fewer, larger sub-tasks with larger data chunks to realize different temporal scheduling. Meanwhile, the device lock enables automatic resource management among workers with data dependencies, enabling spatial scheduling via context switching.

With M2Flow, user-defined workflows can be orchestrated across a complete spatial–temporal scheduling space. Figure 7 illustrates several representative execution modes suited to different RL workloads and configurations.

The left part of Figure 7 shows pure temporal scheduling, where each worker occupies all accelerators. Once a worker completes its task, it is swapped out and the next worker is swapped in. For the illustrated case, since inference and training share the same model weights, no offloading or reloading is required. This mode is particularly useful when a worker must use all devices to be runnable, such as a large model's training. It also maximizes resource sharing among collocated workers and avoids inter-worker pipeline imbalance. However, it suffers from GPU idleness due to long-tail effects, e.g., the longest response length in the rollout stage determines the overall completion time of the stage. This mode is therefore suboptimal when the workload exhibits high variance in component runtime, as is typical in long-context reasoning or open-ended embodied rollouts.

The middle part illustrates spatial scheduling, where workers are assigned to separate GPUs. Because workers depend on one another, pipelining is applied to mitigate idle time.

Achieving efficient pipelining requires balancing resources across workers so that their execution times align. Spatial scheduling effectively hides the long tail of dynamic stages by overlapping them with downstream components, but it may introduce compute and memory imbalance when one component dominates GPU memory (e.g., training with optimizer states) while others underutilize their allocation. In such cases, dedicating disjoint GPU sets to every component forces compute-light components to retain GPUs they cannot fully exploit, leading to overall resource waste.

Finally, RL workflows are often too complex for purely temporal or spatial scheduling to remain efficient. M2Flow therefore supports hybrid scheduling, as shown on the right of Figure 7. Some workers are distributed across GPUs with pipelined execution, but once the workers' stage completes, they can be swapped out and replaced by successors to continue the workflow. Hybrid scheduling preserves resource sharing for the subset of components with large memory footprints (e.g., training and inference that share weights), while pipelining the dynamic, long-tail-prone components (e.g., generation, simulators) on dedicated devices to mask their variability. This trade-off can be applied at fine granularity, with the choice for each component aligned to its compute and memory profile, its position in the dependency graph, and the dynamicity of its inputs.

3.4 Scheduling Policy

RLinf offers a large scheduling space through flexible orchestration of workers, thus finding the most efficient execution mode is challenging. To this end, RLinf introduces two modules: the profiler and the scheduler. The profiler is used to measure and estimate the execution characteristics of each component under different numbers of GPUs. The scheduler then utilizes this information to compose an overall execution plan, containing specific GPU assignments and pipelining configurations. The two modules form a fully automated profiling-to-scheduling pipeline. After developers define the worker logic and workflow runner, RLinf runs offline profiling over a small set of representative data parallel sizes, extrapolates each component's execution time and memory usage, and passes the results to the scheduler, which finds the global execution plan before training begins. Users do not manually choose among temporal, spatial, and hybrid modes; they only declare the resource budget (§4) and, optionally, parallelism hints for very large models.

Profiler. The profiler measures each component's execution time and memory usage under different data parallel sizes, because data parallel size tends to have a positive (sometimes nearly linear) relation with the component throughput as data is consumed faster with the increasing data parallel size. For non-model components like simulators, the data parallel size is the number of instances. For model training and generation components, the profiler requires users to provide model parallel configuration to decide the data parallel size, which can usually be determined by the model size and GPU memory. With the profiled data, the profiler extrapolates the execution time and memory usage for larger data parallel sizes using polynomial extrapolation, outputting an execution time estimation function E for each component. This function is then fed to the scheduler for it to estimate each component's execution time and whether it can be fitted into memory given a selected number of GPUs. Moreover, during profiling, the overall RL workflow is also captured into a workflow graph in a just-in-time manner, by tracing the data flow among workers through the communication primitives.

Algorithm 1: Worker scheduling policy.

Input : Workflow graph G, execution time estimation function E for each component, and number of devices N.
Output : A worker schedule S_{best} and its estimated time T_{best}.

```
1  D_table ←{}; // graph map to (time, schedule)
2  G_dag ←ConvertCircleToNode (G);
3  T_best, S_best ←FindSchedule (G_dag, N, D_table);
4  Function FindSchedule(G, N, D_table):
5      if (G,N) in D_table then
6          return D_table[(G,N)];
7      end
8      if G is a node then
9          return E_node, S_node;
10     end
11     T_best ←+inf; S_best ←None;
12     for G_s, G_t in TraverseStCuts (G) do
13         /* G_s and G_t share the same gpus */
14         T_s, S_s ←FindSchedule (G_s, N, D_table);
15         T_t, S_t ←FindSchedule (G_t, N, D_table);
16         D_table[(G_s,N)]←(T_s, S_s);
17         D_table[(G_t,N)]←(T_t, S_t);
18         if T_best > T_s + T_t then
19             T_best ←T_s+T_t; S_best ←shared(S_s, S_t);
20         end
21         /* G_s and G_t use different devices */
22         for N_s, N_t in TraverseGpuNum (N) do
23             /* N_s+N_t equals N */
24             T_s, S_s ←FindSchedule (G_s, N_s, D_table);
25             T_t, S_t ←FindSchedule (G_t, N_t, D_table);
26             D_table[(G_s,N_s)]←(T_s, S_s);
27             D_table[(G_t,N_t)]←(T_t, S_t);
28             if T_best >PipeliningTime (T_s, T_t) then
29                 T_best ←PipeliningTime (T_s, T_t);
30                 S_best ←pipeline(S_s, S_t);
31             end
32         end
33     end
34     return T_best, S_best;
35 end
```

Scheduler. The scheduling policy of the scheduler is shown in Algorithm 1. The input of the policy includes the workflow graph G, the execution time estimation function E produced by the profiler, and the total number of GPU devices. Specifically, the scheduling algorithm recursively partitions the workflow graph into two subgraphs, G_s and G_t, connected by directed edges known as the *s–t cuts* [11]. For each parti-

tion, it evaluates the time cost of both the temporal and spatial scheduling policies. In the temporal scheduling, G_s and G_t share the same set of devices (e.g., GPUs)—G_s processes its batch, and upon completion G_t consumes its output. In the spatial scheduling, G_s and G_t are assigned to disjoint device sets and executed in a pipelined fashion. The scheduler uses the profiling results to find the optimal device allocation and data processing granularity for each subgraph. The algorithm then selects the most performant scheduling, and applies this process recursively until each subgraph reduces to a single node, at which point the node returns its profiled execution time under the assigned placement.

Modeling the execution time of G_s and G_t is the key to finding the optimal scheduling. In the temporal scheduling where workers share devices, the cost is the sum of G_s and G_t plus any resource offloading and reloading overhead. In the spatial scheduling, the runtime is estimated as

$$T_{\text{critical}} + (M/m - 1) \times T_{\text{bottleneck}},$$

where T_{critical} is the pipeline warm up and cool down time, $T_{\text{bottleneck}}$ is the runtime of the slowest subgraph, M is the total batch size, and m is the data processing granularity.

Before invoking `FindSchedule`, the workflow graph is preprocessed to collapse cycles into single nodes. When recursion reaches such a node, its computation is evenly partitioned across GPUs. This avoids exhaustive partition enumeration while still achieving near-optimal performance.

Algorithmic Complexity. Let V be the number of nodes in the post-collapse workflow DAG and N the number of devices. With memoization, the DP visits $O(V^2)$ distinct subgraphs for the chain-like workflows typical in RL, and each (subgraph, device-count) state takes $O(V \cdot N)$ time because it enumerates $O(V)$ s-t cuts and, per cut, iterates over device partitions $N_s + N_t = N'$. Summing over all $O(V^2 \cdot N)$ states yields an overall search cost of $O(V^3 \cdot N^2)$. For a given workflow, the search cost is $O(N^2)$ as the topology is fixed. We empirically validate this scaling up to 4096 GPUs in §5.2.

Adaptive Re-Profiling for Runtime Dynamicity. Some RL training is inherently dynamic: for LLM, as the model learns, the rollout response length distribution drifts, causing components such as generation to speed up or slow down over time. A static plan derived from initial profiling may therefore become suboptimal mid-training. To handle this, RLinf embeds a runtime profiler inside each worker that continuously records per-invocation execution time, and a lightweight controller routine that compares these measurements against the scheduler's estimated execution times. When the deviation exceeds a configurable threshold (15% sustained over a sliding window in our default configuration), RLinf checkpoints the training state, re-runs the scheduling algorithm with the updated profile, and resumes training under the new execution plan. In typical reasoning RL training, response-length drift is gradual and re-scheduling is warranted only after thousands of iterations, so the checkpoint-and-redeploy overhead is negligible relative to total training time. We quantify the sensitivity of RLinf to profile noise in §5.2.

3.5 Adaptive Communication

To support flexible worker orchestration, RLinf's communication layer needs to realize two key design goals. **(1) Flexible.** Any two workers should be able to communicate with each other, regardless of the worker placement and program logic. **(2) Adaptive.** Communication primitives should be able to adapt to arbitrary data living in different devices (CPUs and GPUs across nodes), while achieving maximum throughput of the underlying communication links.

However, existing collective communication libraries for GPU and CPU data, e.g., NCCL [36], Gloo [39], and MPI [12], fail to meet our goals as they are mostly built for standard communication patterns across a fixed number of processes in traditional model training and serving scenarios. In contrast, RL components manifest considerable spatial and temporal dynamicity. Spatially, components can collocate on the same device, or be distributed across different devices. Temporally, components can be launched or terminated at arbitrary times. However, state-of-the-art libraries like NCCL do not support flexible rank scaling and efficient intra-device communications. Furthermore, the data communicated between components can be complex data structures beyond standard contiguous GPU/CPU data buffers, e.g., a composition of multiple data buffers with varying sizes. Efficient handling of such data dynamicity is also missing in existing libraries.

Communication Protocol and Primitives. To achieve the design goals, we devise worker and data placement-aware communication protocol and primitives, which enhance existing CPU/GPU communication libraries in terms of both flexibility and performance in the RL scenario.

At the protocol level, RLinf features transparent connection lifecycle management, which avoids manual connection management as in traditional communication libraries, and handles dynamic worker placement and scaling automatically. Specifically, upon launch, each worker's placement, IP and port information will be registered into a global worker manager. Connections among workers are then established with the information lazily when workers invoke communication primitives to reduce connection overhead and enhance scalability. When a group of workers establish a connection, the connection metadata is maintained both locally by workers and globally by a connection manager. When a worker is terminated, the connection manager will notify all connected workers to teardown the connection and release resources.

At the primitive level, like existing libraries, RLinf offers both synchronous and asynchronous `send` and `recv` primitives for point-to-point communication, as well as collective communication primitives like `broadcast`. Differently, RLinf's primitives automatically exploit the worker

and data placement information of the communicating workers to select the most efficient communication backend, e.g., NCCL for GPU-GPU communication, zero-copy cudaIPC for intra-GPU communication, and Gloo for CPU communication. For data dynamicity, both the asynchronous and synchronous primitives support arbitrary Python objects as the communication payload, which are serialized and deserialized in a structure-aware manner, i.e., data buffers are extracted from the objects and communicated directly without serialization/deserialization overhead. Also, data structure information is piggybacked in the communication metadata to facilitate efficient deserialization at the receiver side.

Load-Balancing Data Channel. Atop the above communication primitives, we further build a high-level FIFO queue-like communication facility (termed *data channel*) for producer-consumer worker communication in the workflow. This enables decoupling of both the control and data flows of producer and consumer workers, which is essential for flexible pipelining. The data channel maintains its data queue in a special channel worker process, and can be accessed by any other worker processes by passing the channel handle. The channel supports both CPU and GPU data, and can be configured for offloading GPU data to CPU to reduce GPU memory consumption. Furthermore, the data channel is enhanced with load-balancing capability. Each item enqueued to the channel can be assigned a weight value, which is used to balance the load across multiple consumers dequeuing from the channel. The consumers can also define custom load-balancing policies, which are invoked by the channel upon each dequeue operation to select the desired items from the channel.

4 Implementation

RLinf is implemented in 20K lines of Python code. Among them, 5K lines are for the core worker, controller, and scheduler components. 2K for common workers such as rollout workers based on serving engines like SGLang, vLLM and HuggingFace Transformers, training actors based on Megatron and FSDP, and embodied simulators, which can be used out of the box for any future RL models and workflows. The remaining 13K lines are mostly rich support for various RL algorithms like PPO and GRPO. Notably, a typical workflow runner like the LLM reasoning RL workflow implementation is less than 100 lines of code, and requires no code changes to be scheduled both temporally and spatially.

Currently, RLinf supports not only traditional LLM-based reasoning RL, but also agentic RL with tooling, and embodied RL involving complex workloads like 3D rendering, physics simulation, and robotic control. For RL algorithms, RLinf supports popular algorithms like PPO [48], GRPO [49], DAPO [61], and REINFORCE++ [16], as well as some of their off-policy asynchronous versions. Table 1 summarizes the broader set of algorithms supported by RLinf,

Table 1: RL algorithm categories supported by RLinf.

Category	Algorithms
On-policy	GRPO [49], PPO [48], DAPO [61], REINFORCE++ [16]
Off-policy	Async PPO, SAC [14], CrossQ [5], RLPD [4], SAC-Flow [65], DSRL [56]
Imitation	DPO [41], Full-param SFT, LoRA SFT, VLM SFT, DAgger, HG-DAgger

covering synchronous on-policy methods, off-policy and asynchronous methods, and imitation learning. Notably, the M2Flow paradigm naturally generalizes to complex asynchronous algorithms and to agentic workflows with tool calls and multi-agent settings (e.g., Search-R1 [22], rStar2 [47]), simply by adapting the workflow to the components.

Differently, asynchronous algorithms differ in workflow structure from their synchronous counterparts: instead of executing lock-step phases, they run long-lived concurrent workers and synchronize weights non-blockingly. Despite this, expressing such workflows in RLinf remains concise:

```
class AsyncRunner:
  def run(self):
    self.update_rollout_weights()
    env_h = self.env.interact(
      self.env_ch, self.rollout_ch, self.actor_ch
    )
    rollout_h = self.rollout.generate(
        self.rollout_ch, self.env_ch
    )
    while step < max_steps:
      self.actor.run_training(self.actor_ch).wait()
      if step % sync_interval == 0:
        self.update_rollout_weights()
    self.env.stop(); self.rollout.stop()
```

The environment, rollout, and actor workers run as persistent asynchronous tasks streaming data through channels, while training proceeds concurrently without blocking on rollout completion. This shows that M2Flow's channel-based decoupling generalizes naturally to asynchronous settings.

For models, we have implemented support for language models like Qwen [40], multi-modal models like VLM [3], VLA models like OpenVLA [23, 24] and Pi0 [55], and world models like OpenSora [70] and Wan [57]. The rich RL workflow, algorithm and model support not only accelerates the development of new RL workflows with RLinf, but also demonstrates its generality, versatility and extensibility in practice.

Cluster Management and Device Allocation. RLinf leverages Ray to realize cluster management, launching worker processes on remote nodes, and dispatching worker function executions. RLinf does not rely on Ray for device allocation to workers, because Ray only supports rigid packed-style (i.e., consecutive devices) or spread-style (i.e., spread-across-node first) resource allocation [33], which are not flexible enough for today's complex and dynamic RL workflows. Instead, RLinf offers a flexible device allocation strategy, which allows each worker process to be allocated with any device or devices of any node across the cluster, by simply specifying the target devices' global IDs. Also, hardware devices be-

yond accelerators such as robot arms are also abstracted and managed as schedulable devices in the same manner as accelerators. This enables RLinf to scale on all kinds of hardware that foundation models can interact with.

Multi-Tenant Resource Boundary. In multi-tenant clusters, a cluster manager (e.g., SLURM, Kubernetes) assigns each job a fixed set of nodes. RLinf scopes its runtime to this allocation, and the scheduler explores execution modes only within it, so M2Flow does not alter the job's resource contract with the cluster manager. Within the allocation, the distributed device lock of the data channel (§3.5) enforces exclusive accelerator access when components share devices via temporal scheduling, preventing intra-job contention.

Heterogeneous Hardware Support. RLinf abstracts accelerators behind a unified hardware interface, allowing the scheduler to target clusters that mix GPUs/NPUs from different vendors (e.g., NVIDIA, AMD, Intel, Ascend, MUSA) and non-standard devices such as robotic arms. Nodes are grouped by hardware type, and a declarative placement configuration maps each workflow component to one or more groups. The profiler measures each component on its actual device type, so the scheduler accounts for heterogeneous execution times when searching the spatio-temporal plan.

Fault Tolerance. The distributed, multi-component nature of RLinf makes worker process crashes, node failures, and network issues unavoidable in long training runs. RLinf adopts checkpoint-based recovery: when any worker fails, it detects the failure via heartbeat and notifies the Controller, which halts the entire training job to prevent dependent workers from timing out or producing inconsistent state. Users can then restart from the latest checkpoint, and RLinf replays the offline profiling phase if the resource boundary has changed before resuming under a possibly updated execution plan.

Performance Profiling. Performance profiling support is not only crucial for developers to understand system bottlenecks, but also serves as the key to our scheduling policy. Thus, RLinf provides a worker-group-level timer for every public function invoked remotely by the workflow runner. The timer automatically captures the execution time of a worker function, whose value can be retrieved via the asynchronous handle returned by the corresponding worker group function; the values of all processes in the worker group will be reduced to a single value via a specified reduction method (e.g., mean, max, min). Beyond this, developers can also create custom timers for more fine-grained profiling of any code region, and retrieve the timer values similarly.

5 Evaluation

We extensively evaluate RLinf across math-reasoning and embodied RL workloads, covering four different models of different sizes (i.e., Qwen2.5 [40], Qwen3-MoE [59], Open-

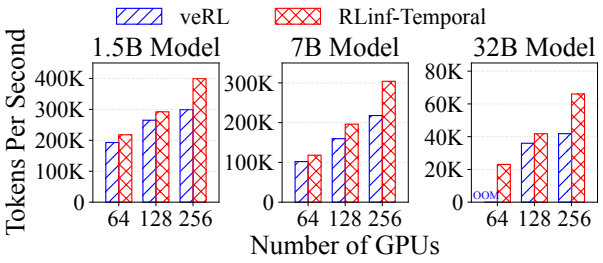


Figure 8: GRPO training throughput of Qwen2.5 on RLinf and veRL under different cluster scales and model sizes.

VLA [24], OpenVLA-OFT [23]), two RL algorithms (i.e., GRPO [49], PPO [48]), and multiple cluster scales. Overall, our key findings include:

- RLinf consistently outperforms state-of-the-art RL systems veRL [51] and Slime [71] by 1.07×∼1.70× on a variety of math-reasoning RL settings. The results also show that different RL settings favor different execution modes.
- RLinf demonstrates higher training throughput on embodied RL tasks in both ManiSkill [54] and LIBERO [27]. On LIBERO, RLinf achieves a 1.05×∼2.43× speedup over SimpleVLA-RL [25]; on ManiSkill, its hybrid mode yields up to 1.87× improvement over other strategies.
- The scheduling policy identifies the best execution mode within $7 \times 10^{-4} \sim 5.98s$ on the clusters of 8 to 1024 GPUs.
- The models trained with RLinf achieve SOTA or better benchmark scores after RL fine-tuning.

5.1 End-to-End Experiments

The end-to-end experiments run on a 32-node cluster, where each node is equipped with 8 NVIDIA H100-80GB GPUs, 2 Intel Xeon Platinum 8558 CPUs (2.1 GHz, 48 cores), and 2 TB of memory. Intra-node communication uses NVLink, while inter-node communication uses 8 Mellanox ConnectX-7 RDMA NICs per node, each providing 400 Gbps bandwidth with RoCEv2.

5.1.1 Reasoning RL Training

Experimental Setup. We evaluate using Qwen2.5 Models [40] distilled by DeepSeek-R1, covering sizes from 1.5B to 32B parameters and Qwen3-30B-A3B [59], which is an MoE model, and using the AReaL-boba-Data dataset [58], which integrates multiple standard datasets including DeepScaleR, Open-Reasoner-Zero, Light-R1, etc. We compare RLinf with two state-of-the-art open-source RL systems, i.e., veRL v0.5 [51] and Slime v0.1 [71]. To ensure fairness, all systems use SGLang [68] for rollouts and Megatron-LM [52] for training, with the same parallelism setting (e.g., tensor parallelism configuration). Training speed is reported in tokens/sec, defined as the total number of prompt and response

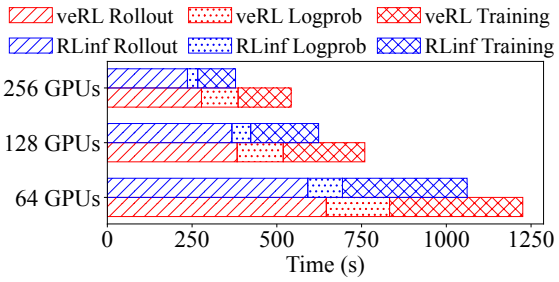


Figure 9: Latency breakdown of Qwen2.5 7B model training.

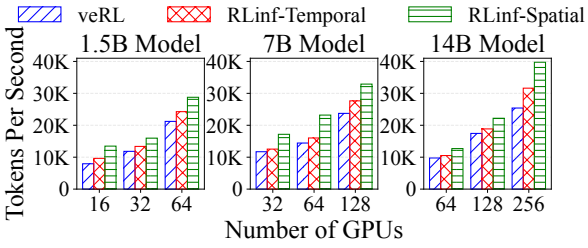


Figure 10: PPO training throughput of Qwen2.5 on RLinf and veRL under different cluster scales and model sizes.

tokens in a global batch divided by the iteration time. All performance results are averaged over 10 training iterations after warm-up.

Qwen2.5 with GRPO. We evaluate the Qwen2.5 1.5B, 7B, and 32B dense models on 64, 128, and 256 GPUs, respectively, using a rollout batch size of 512 and maximum sequence length 28672. As shown in Figure 8, RLinf in temporal mode consistently outperforms veRL across all model scales, achieving 1.10× to 1.58× speedups. Since RLinf 's temporal mode is similar to veRL's design, the gains primarily stem from faster rollouts enabled by larger KV-cache allocations (RLinf has better GPU memory management) and reduced synchronization overhead between upstream and downstream stages of the middle inference stage, as illustrated in Figure 9.

Figure 8 also shows that veRL scales poorly as GPU number increases, largely because inference becomes a growing bottleneck, with its proportion of total execution time rising from 15.2% to 19.9% when scaling from 64 to 256 GPUs. In addition, veRL's unoptimized rollout engine leads to excessive peak memory usage, forcing smaller KV-cache allocations and causing rollout time to decrease only sublinearly.

We also tested spatial mode, but it underperforms veRL by 44.3-68.6% for the 7B model, because rollout, inference, and training run on disjoint smaller sets of GPUs, slowing both rollout and training. With the long sequence length (i.e., 28672), the training stage also waits longer for the first batch of rollouts to be generated. In addition, the rollout batch size is relatively smaller, reducing the overlapping time.

Qwen2.5 with PPO. We train PPO with Qwen2.5 as both the actor and critic, using model sizes of 1.5B, 7B, and 14B, scaling from 16 to 256 GPUs. We use a rule-based reward, a maximum sequence length of 12288, and a rollout batch size ranging from 256 to 1024 depending on GPU count. RLinf-Spatial allocates GPUs in a 4:1:1:1:1 ratio for rollout, actor inference, actor training, critic inference, and critic training. Figure 10 presents the results. For the 1.5B model, RLinf-Spatial outperforms veRL by 69.6%, 35.0%, and 35.6% on 16, 32, and 64 GPUs, respectively. In contrast to the GRPO experiment, RLinf-Spatial surpasses RLinf-Temporal by 39.8%, 19.4%, and 18.6%.

For the 7B model, RLinf-Spatial outperforms both veRL and RLinf-Temporal by 38.7–60.7% and 19.0–44.8%, respectively. As shown in Figure 11, RLinf-Temporal exceeds veRL mainly due to speedups in "Others", which includes context-switch overhead (offload/onload), parameter resharding, advantage computation, and inter-stage synchronization. RLinf-Spatial performs faster than RLinf-Temporal because rollout, inference, and training overlap effectively. Although rollout in spatial mode is 39.3% slower than in temporal mode, inference and training finish quickly once rollout completes. The 14B model shows similar trends with smaller gains, i.e., RLinf-Spatial is 27.2–56.5% and 17.7–25.7% faster than veRL and RLinf-Temporal, respectively.

Qwen3-30B-A3B with GRPO. We evaluate the MoE model Qwen3-30B-A3B on 32, 64, and 128 GPUs with a rollout batch size of 1536 and sequence length 20480. We disable logprob recomputation, a common configuration in RL, which removes the inference stage. Since Slime is optimized for MoE RL training, we compare against two variants: Slime (spatial mode without pipelining, as Slime does not support pipelining) and Slime-Colocate (temporal mode). As shown in Figure 12, Slime is the slowest because rollout and training run on disjoint GPU sets with no execution overlap. For 32 and 64 GPUs, RLinf-Temporal performs similarly to Slime-Colocate, while RLinf-Spatial (1:1 rollout-to-training GPUs) is 31.2% and 7.2% faster than Slime-Colocate, respectively.

Figure 13 presents the performance breakdown. Spatial mode achieves shorter rollout time because temporal mode suffers from memory contention between rollout and training, forcing much smaller parallelism (i.e., *max_running_requests* of 128 vs. 256 in SGLang). In addition, RLinf-Spatial achieves good rollout–training overlap. On 128 GPUs, RLinf-Temporal outperforms Slime-Colocate by 3.7% due to better scaling from decoupled system design; both stages run slightly faster. RLinf-Spatial underperforms because rollout and training do not overlap well, where training continues for 80 seconds after rollout completes.

5.1.2 Agentic RL Training

To demonstrate RLinf's flexibility beyond standard reasoning workloads, we evaluate it on Search-R1-style [22] retrieval-augmented agentic RL workflows that require external tool calls and search interactions. Unlike the rigid structure of standard PPO or GRPO, these workflows introduce highly variable-length tool-call traces. RLinf seamlessly supports

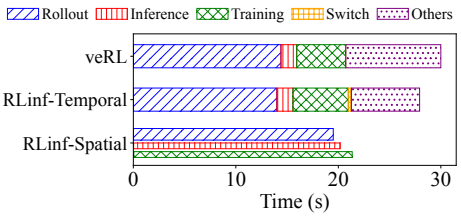


Figure 11: Latency breakdown of Qwen2.5 7B with PPO on 32 GPUs. The vertical width of bar presents the number of GPUs.

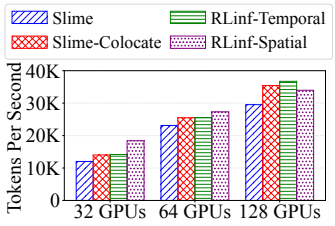


Figure 12: RL training throughput of Qwen3-30B-A3B on RLinf and Slime under different cluster scales.

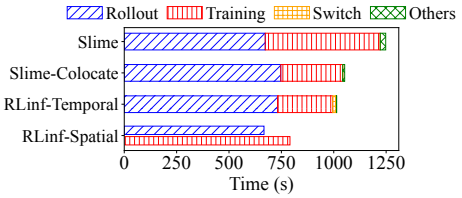
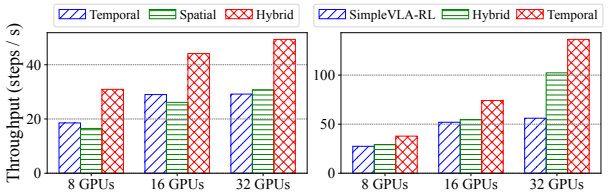


Figure 13: Latency breakdown of Qwen3-30B-A3B with GRPO on 32 GPUs. The vertical width of bar is proportional to the number of GPUs.



(a) ManiSkill (b) LIBERO

Figure 14: End-to-end throughput of RLinf and SimpleVLA-RL under different cluster scales.

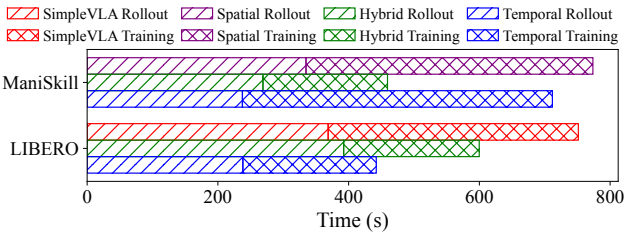


Figure 15: Latency breakdown of ManiSkill and LIBERO.

these complex patterns with its unified worker abstraction. By leveraging M2Flow to overlap generation, external tool invocation, and training steps, RLinf drastically improves execution efficiency.

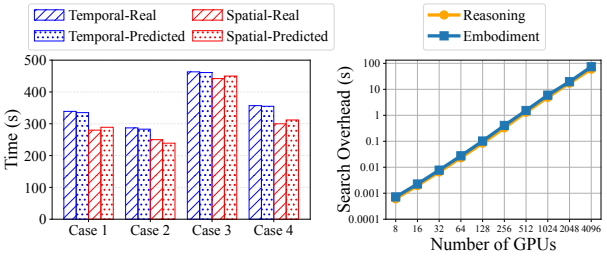
We compare RLinf against the official Search-R1 implementation [21], which is built on veRL. Both systems train Qwen2.5-3B with the Search-R1 agentic workflow on a single node with 8 NVIDIA H100 GPUs, using the same dataset and hyperparameters. RLinf achieves an end-to-end throughput of 67.3 requests/s compared to the 30.4 requests/s of the standard Search-R1 implementation, delivering a 2.2× throughput improvement.

5.1.3 Embodied RL Training

Experimental Setup. We evaluate on OpenVLA [24] and OpenVLA-OFT [23], two vision-language-action models that are supervised-finetuned using RL4VLA [28] and SimpleVLA-RL [25], respectively. We train OpenVLA on ManiSkill [54] and OpenVLA-OFT on LIBERO [27], two widely used embodied environments that emulate physical tasks such as pick-and-place. On LIBERO, we compare RLinf with SimpleVLA-RL (commit d001d) [25], which is built on veRL. On ManiSkill, no distributed RL baseline exists, so we compare different execution modes of RLinf. Training speed is reported in steps/sec, computed as the total number of environment steps divided by the iteration time.

ManiSkill Environment. We train OpenVLA on the "PutCarrotOnPlateInScene-v2" task [28] using 256 parallel ManiSkill environments, each stepping for 80 steps per iteration. Figure 14(a) reports end-to-end throughput across three execution modes in RLinf on 8, 16, and 32 GPUs. RLinf-Hybrid achieves 52.2–69.1% higher throughput than RLinf-Temporal and 60.7–87.2% higher than RLinf-Spatial. For ManiSkill environment, the per-step environment time remains nearly constant as parallelism increases (Figure 3b) and scaling parallel environments is primarily limited by GPU memory. Thus, dedicating GPUs to environment execution is advantageous. As rollout (i.e., environment and generation) and training all require GPU memory, temporally sharing GPUs between rollout and training is more effective. Thus, RLinf-Hybrid performs much better than the other two. In RLinf-Temporal, environments and generation must coexist in memory, constraining both the number of environments and the batch size, ultimately reducing GPU utilization. RLinf-Spatial splits GPUs across the components, forcing training to retain a sufficient number of GPUs, which leaves too few GPUs for rollout and significantly slows it down.

LIBERO Environment. We train OpenVLA-OFT on the public benchmark task groups provided by LIBERO, using 512 parallel environments, each running 64 steps per iteration. The results in Figure 14(b) show that RLinf-Temporal is 37.8%, 42.6%, 143.4% faster than SimpleVLA-RL on 8, 16, 32 GPUs respectively. As shown in Figure 15, both rollout and training are faster in RLinf-Temporal than in SimpleVLA-RL due to (1) the elimination of redundant environment initialization during rollout, and (2) the use of a single forward pass to compute both action and log probability, with only a modest memory increase. RLinf-Temporal also scales better thanks to its decoupled system design and more efficient implementation. In contrast to ManiSkill, RLinf-Hybrid performs worse



(a) (b)

Figure 16: (a) The real and predicted latency in different cases. (b) The search overhead under different cluster scales.

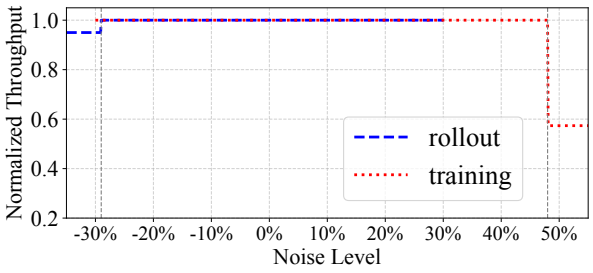


Figure 17: Sensitivity of RLinf's scheduling to profiling noise.

than RLinf-Temporal because LIBERO environment is CPU-intensive; allocating environments on a subset of GPUs limits the utilization of CPU cores, making its rollout stage even slower than that of SimpleVLA-RL.

5.2 Effectiveness of Search Policy

We evaluate the search policy along two dimensions: (a) the accuracy of end-to-end throughput estimates derived from profiling data, and (b) the search speed as the number of GPUs increases.

Estimation Accuracy. Figure 16(a) compares real and predicted iteration times across four Qwen2.5–GRPO settings. Cases 1 and 2 use the 1.5B model on 128 GPUs with temperatures 0.6 and 1.0, while Cases 3 and 4 use the 7B model on 64 GPUs with the same temperature settings. Temporal mode achieves estimation errors below 2% because its end-to-end time is well approximated by summing profiled worker times. Spatial mode with pipelining shows slightly higher errors (<5%), mainly due to response-length variability that causes pipeline imbalance. These errors are small enough that they do not change the ranking of feasible execution modes, allowing the search policy to consistently select the most efficient mode for the above end-to-end experiments.

Search Speed. Search complexity is dominated by the number of workflow-graph nodes and available GPUs. RL workflows typically have fewer than ten nodes, allowing our dynamic-programming search to finish quickly. For a graph with three nodes (e.g., Qwen2.5-GRPO or ManiSkill) and GPU counts ranging from 8 to 4096 (Figure 16(b)), search time grows polynomially with the GPU count, consistent with the $O(N^2)$ analysis in §3.4, and remains under 60 seconds on 4096 GPUs, demonstrating the efficiency of our algorithm.

Sensitivity Analysis. To evaluate RLinf's robustness against profiling inaccuracies, we conduct a sensitivity analysis by injecting synthetic noise into the profiled execution times for different phases and measuring the actual throughput of the resulting placement. The experiment runs on a 1.5B model trained with GRPO on 128 GPUs. As Figure 17 shows, RLinf maintains optimal throughput across a wide noise range.

Table 2: Evaluation scores of RLinf-math-1.5B/7B and open-source models. GPQA represents GPQA-diamond.

Model	AIME24	AIME25	GPQA	Avg.
1.5B models				
Base model	28.33	24.90	27.45	26.89
DeepScaleR [30]	40.41	30.93	27.54	32.96
AReaL [13]	44.42	34.27	33.81	37.50
FastCuRL [53]	43.65	32.49	35.00	37.05
RLinf	**48.44**	**35.63**	**38.46**	**40.84**
7B models				
Base model	54.90	40.20	45.48	46.86
AReaL [13]	61.66	49.38	46.93	52.66
Skywork [15]	66.87	52.49	44.43	54.60
Polaris [2]	**68.55**	51.24	43.88	54.56
AceMath [29]	67.30	**55.00**	45.57	55.96
RLinf	68.33	52.19	**48.18**	**56.23**

Specifically, it tolerates underestimated rollout times (negative noise) down to roughly -29% and overestimated training times (positive noise) up to roughly $+48\%$. Because GPU allocation is discrete, minor variations in estimated phase latencies do not shift performance bottlenecks enough to alter the optimal resource distribution; the search policy simply converges on the same physical placement. The throughput drops only occur when extreme noise tricks the algorithm into a suboptimal configuration. This confirms RLinf achieves optimal scheduling without requiring perfectly precise profiling.

5.3 Model Performance

We comprehensively evaluate the performance of the models produced by the reasoning and the embodied RL training, respectively, to demonstrate the effectiveness of RLinf. We emphasize that RLinf does not modify the underlying RL algorithms. The downstream task scores reported below serve only to confirm that RLinf trains models correctly, and that its higher per-iteration throughput translates into improved model quality within a fixed wall-clock and compute budget.

Reasoning RL Training. We applied GRPO training, a built-in RL algorithm in RLinf, on two base models, i.e., DeepSeek-R1-Distill-Qwen-1.5B and DeepSeek-R1-Distill-Qwen-7B, to improve their math reasoning ability. As shown

Table 3: OpenVLA model success rate results on ManiSkill3.

Model	Vision	Semantic	Position	Avg.
RL4VLA [28]	80.47%	75.00%	81.77%	79.15%
RLinf	**82.03%**	**78.35%**	**85.42%**	**81.93%**

Table 4: OpenVLA-OFT success rate after RL on LIBERO.

Model	Spatial	Object	Goal	Long	Avg.
OpenVLA-OFT(one-traj) [25]	56.45%	25.60%	45.59%	9.68%	34.33%
RLinf	**98.99%**	**98.99%**	**98.99%**	**94.35%**	**97.83%**

in Table 2, the models trained with RLinf achieve the best average performance compared to the baseline systems. RLinf 1.5B model outperforms the baselines across all the three benchmarks (i.e., AIME 24 [63], AIME 25 [64], GPQA-diamond [45]), improving by up to 20 points over its base model. RLinf 7B model achieves the highest performance on GPQA-diamond.

Embodied RL Training. We also evaluate the performance of the models after embodied RL training using RLinf. Table 3 and Table 4 present the evaluation results on the ManiSkill and LIBERO tasks by training OpenVLA and OpenVLA-OFT models respectively, using the PPO algorithm. For ManiSkill, the trained OpenVLA achieves higher success rate than the RL4VLA [28] baseline system. For LIBERO, we further train the publicly released OpenVLA-OFT model (one-trajectory fine-tuned version) from SimpleVLA-RL [25]. By comparing its task success rates with those of the model RL trained using RLinf, we find that RLinf significantly improves OpenVLA-OFT's performance on the LIBERO tasks.

6 Related Works

RL Training Frameworks. RL training frameworks adopt varying system designs to support large-scale alignment tasks [8, 38], typically falling into either task-colocated or task-separated execution modes. For task-colocated systems, DeepSpeed Chat [60] and veRL [51] put all training phases on shared GPUs for simplified orchestration. In contrast, task-separated systems like NeMo-Aligner [50] and Open-RLHF [17] divide components across devices to improve modularity and scalability. AReaL [13] further introduces an asynchronous model-update algorithm upon task-separated systems to increase training throughput. Unlike these frameworks, RLinf provides a more flexible component-to-device placement mechanism, enabling practitioners to explore and deploy more efficient execution configurations tailored to workload characteristics.

Distributed Training Systems for LLM. Distributed training frameworks for LLMs [26, 66, 67] and system-level optimizations for memory [7, 18, 32] and communication [6, 20, 31] have significantly advanced large-model scaling. Megatron-LM [52] combines tensor, pipeline, and data parallelism to improve scalability, while DeepSpeed [44] introduces the ZeRO family of techniques [42, 43, 46] that shard optimizer states, gradients, and activations to reduce GPU memory usage. Although distributed RL systems face similar multi-device scaling challenges, RL workflows are more dynamic due to interactive data generation and asynchronous updates, making system design and optimization substantially more complex.

Dataflow System. Traditional dataflow systems [9, 19, 34, 62] excel at large-scale data processing through static task graphs and centralized scheduling, making them efficient for batch and streaming workloads with predictable structures. In contrast, RL training pipelines feature dynamic task graphs and asynchronous components (e.g., data collection, policy updates), requiring more flexible coordination than existing systems. Modern frameworks such as Ray [33] fill this gap with actor-based execution and decentralized control, making them better suited for orchestrating complex RL workflows.

7 Conclusion

Reinforcement learning is poised to surpass pretraining as the driving force behind LLM progress, but its workflows are too diverse and dynamic for rigid execution models. RLinf shows that by decoupling workflow logic from execution through the novel macro-to-micro transformation mechanism, we can unlock both efficiency and programmability. Beyond RL, we see this approach as a blueprint for future AI runtimes: systems that flexibly orchestrate heterogeneous components, e.g., training, inference, simulation, and reasoning, under one unified execution framework. We believe RLinf marks an early step toward the operating system for AI workloads.

Acknowledgments

We thank our anonymous reviewers and shepherd for their invaluable suggestions. We also express our sincere appreciation to the RLinf open-source community. The continuous stream of code contributions, feature requests, and user experience feedback from our GitHub contributors has been vital to the system's success and optimization. This work was partially supported by National Natural Science Foundation of China (No.62406159, 62325405), Zhongguancun Academy (Grant No. C20250301), Shenzhen Science and Technology Program (No. AI2026016).

References

[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. Tensorflow: a system for large-scale machine learning.

In *12th USENIX symposium on operating systems design and implementation (OSDI 16)*, pages 265–283, 2016.

[2] Chenxin An, Zhihui Xie, Xiaonan Li, Lei Li, Jun Zhang, Shansan Gong, Ming Zhong, Jingjing Xu, Xipeng Qiu, Mingxuan Wang, and Lingpeng Kong. Polaris: A post-training recipe for scaling reinforcement learning on advanced reasoning models, 2025.

[3] Shuai Bai, Keqin Chen, Xuejing Liu, Jialin Wang, Wenbin Ge, Sibo Song, Kai Dang, Peng Wang, Shijie Wang, Jun Tang, et al. Qwen2. 5-vl technical report. *arXiv preprint arXiv:2502.13923*, 2025.

[4] Philip J. Ball, Laura Smith, Ilya Kostrikov, and Sergey Levine. Efficient online reinforcement learning with offline data, 2023.

[5] Aditya Bhatt, Daniel Palenicek, Boris Belousov, Max Argus, Artemij Amiranashvili, Thomas Brox, and Jan Peters. Crossq: Batch normalization in deep reinforcement learning for greater sample efficiency and simplicity. In *International Conference on Learning Representations*, 2024.

[6] Chang Chen, Xiuhong Li, Qianchao Zhu, Jiangfei Duan, Peng Sun, Xingcheng Zhang, and Chao Yang. Centauri: Enabling efficient scheduling for communication-computation overlap in large model training via communication partitioning. In *Proceedings of ASPLOS*, pages 178–191, 2024.

[7] Tianqi Chen, Bing Xu, Chiyuan Zhang, and Carlos Guestrin. Training deep nets with sublinear memory cost. *arXiv preprint arXiv:1604.06174*, 2016.

[8] Paul F Christiano, Jan Leike, Tom Brown, Miljan Martic, Shane Legg, and Dario Amodei. Deep reinforcement learning from human preferences. *Advances in Neural Information Processing Systems*, 2017.

[9] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[10] Jiafei Duan, Samson Yu, Hui Li Tan, Hongyuan Zhu, and Cheston Tan. A survey of embodied ai: From simulators to research tasks. *Proceedings of IEEE*, 2022.

[11] Lester Randolph Ford and Delbert Ray Fulkerson. Flows in networks. 2015.

[12] Message Passing Interface Forum. Mpi: A message-passing interface standard. https://www.mpi-forum.org, 2025.

[13] Wei Fu, Jiaxuan Gao, Xujie Shen, Chen Zhu, Zhiyu Mei, Chuyi He, Shusheng Xu, Guo Wei, Jun Mei, Jiashu Wang, et al. Areal: A large-scale asynchronous reinforcement learning system for language reasoning. *arXiv preprint arXiv:2505.24298*, 2025.

[14] Tuomas Haarnoja, Aurick Zhou, Pieter Abbeel, and Sergey Levine. Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor. *International Conference on Machine Learning (ICML)*, 2018.

[15] Jujie He, Jiacai Liu, Chris Yuhao Liu, Rui Yan, Chaojie Wang, Peng Cheng, Xiaoyu Zhang, Fuxiang Zhang, Jiacheng Xu, Wei Shen, Siyuan Li, Liang Zeng, Tianwen Wei, Cheng Cheng, Yang Liu, and Yahui Zhou. Skywork open reasoner series. https://capricious-hydrogen-41c.notion.site/Skywork-Open-Reaonser-Series-1d0bc9ae823a80459b46c149e4f51680, 2025. Notion Blog.

[16] Jian Hu. Reinforce++: A simple and efficient approach for aligning large language models. *arXiv preprint arXiv:2501.03262*, 2025.

[17] Jian Hu, Xibin Wu, Zilin Zhu, Weixun Wang, Dehao Zhang, Yu Cao, et al. Openrlhf: An easy-to-use, scalable and high-performance rlhf framework. *arXiv preprint arXiv:2405.11143*, 2024.

[18] Zixiao Huang, Junhao Hu, Hao Lin, Chunyang Zhu, Yueran Tang, Quanlu Zhang, Zhen Guo, Zhenhua Li, Shengen Yan, Zhenhua Zhu, Guohao Dai, and Yu Wang. Stalloc: Enhancing memory efficiency in large-scale model training with spatio-temporal planning. In *Proceedings of EuroSys*, page 728–743, 2026.

[19] Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. Dryad: distributed data-parallel programs from sequential building blocks. In *Proceedings of EuroSys*, pages 59–72, 2007.

[20] Anand Jayarajan, Jinliang Wei, Garth Gibson, Alexandra Fedorova, and Gennady Pekhimenko. Priority-based parameter propagation for distributed dnn training. *Proceedings of MLSys*, 1:132–145, 2019.

[21] Bowen Jin, Hansi Zeng, Zhenrui Yue, Dong Wang, Hamed Zamani, and Jiawei Han. Search-R1: An efficient, scalable RL training framework for reasoning & search engine calling interleaved LLM based on veRL. https://github.com/PeterGriffinJin/Search-R1, 2025.

[22] Bowen Jin, Hansi Zeng, Zhenrui Yue, Dong Wang, Hamed Zamani, and Jiawei Han. Search-r1: Training

LLMs to reason and leverage search engines with reinforcement learning. *arXiv preprint arXiv:2503.09516*, 2025.

[23] Moo Jin Kim, Chelsea Finn, and Percy Liang. Fine-tuning vision-language-action models: Optimizing speed and success. *arXiv preprint arXiv:2502.19645*, 2025.

[24] Moo Jin Kim, Karl Pertsch, Siddharth Karamcheti, Ted Xiao, Ashwin Balakrishna, Suraj Nair, Rafael Rafailov, Ethan Foster, Grace Lam, Pannag Sanketi, Quan Vuong, Thomas Kollar, Benjamin Burchfiel, Russ Tedrake, Dorsa Sadigh, Sergey Levine, Percy Liang, and Chelsea Finn. Openvla: An open-source vision-language-action model. *arXiv preprint arXiv:2406.09246*, 2024.

[25] Haozhan Li, Yuxin Zuo, Jiale Yu, Yuhao Zhang, Zhaohui Yang, Kaiyan Zhang, Xuekai Zhu, Yuchen Zhang, Tianxing Chen, Ganqu Cui, Dehui Wang, Dingxiang Luo, Yuchen Fan, Youbang Sun, Jia Zeng, Jiangmiao Pang, Shanghang Zhang, Yu Wang, Yao Mu, Bowen Zhou, and Ning Ding. Simplevla-rl: Scaling vla training via reinforcement learning, 2025.

[26] Shenggui Li, Hongxin Liu, Zhengda Bian, Jiarui Fang, Haichen Huang, Yuliang Liu, Boxiang Wang, and Yang You. Colossal-ai: A unified deep learning system for large-scale parallel training. In *Proceedings of ICPP*, pages 766–775, 2023.

[27] Bo Liu, Yifeng Zhu, Chongkai Gao, Yihao Feng, Qiang Liu, Yuke Zhu, and Peter Stone. Libero: Benchmarking knowledge transfer for lifelong robot learning. *arXiv preprint arXiv:2306.03310*, 2023.

[28] Jijia Liu, Feng Gao, Bingwen Wei, Xinlei Chen, Qingmin Liao, Yi Wu, Chao Yu, and Yu Wang. What can RL bring to VLA generalization? an empirical study. In *The Thirty-ninth Annual Conference on Neural Information Processing Systems*, 2025.

[29] Zihan Liu, Yang Chen, Mohammad Shoeybi, Bryan Catanzaro, and Wei Ping. Acemath: Advancing frontier math reasoning with post-training and reward modeling. *arXiv preprint*, 2024.

[30] Michael Luo, Sijun Tan, Justin Wong, Xiaoxiang Shi, William Y. Tang, Manan Roongta, Colin Cai, Jeffrey Luo, Li Erran Li, Raluca Ada Popa, and Ion Stoica. Deepscaler: Surpassing o1-preview with a 1.5b model by scaling rl. https://pretty-radio-b75.notion.site/DeepScaleR-Surpassing-O1-Preview-with-a-1-5B-Model-by-Scaling-RL-19681902c1468005bed8ca303013a4e2, 2025. Notion Blog.

[31] Kshiteej Mahajan, Ching-Hsiang Chu, Srinivas Sridharan, and Aditya Akella. Better together: Jointly optimizing ml collective scheduling and execution planning using syndicate. In *Proceedings of NSDI*, pages 809–824, 2023.

[32] Chen Meng, Minmin Sun, Jun Yang, Minghui Qiu, and Yang Gu. Training deeper models by gpu memory optimization on tensorflow. In *Proceedings of ML Systems Workshop in NIPS*, volume 7, 2017.

[33] Philipp Moritz, Robert Nishihara, Stephanie Wang, Alexey Tumanov, Richard Liaw, Eric Liang, Melih Elibol, Zongheng Yang, William Paul, Michael I Jordan, et al. Ray: A distributed framework for emerging {AI} applications. In *13th USENIX symposium on operating systems design and implementation (OSDI 18)*, pages 561–577, 2018.

[34] Derek G Murray, Frank McSherry, Rebecca Isaacs, Michael Isard, Paul Barham, and Martín Abadi. Naiad: a timely dataflow system. In *Proceedings of SOSP*, pages 439–455, 2013.

[35] Reiichiro Nakano, Jacob Hilton, Suchir Balaji, Jeff Wu, Long Ouyang, Christina Kim, Christopher Hesse, Shantanu Jain, Vineet Kosaraju, William Saunders, et al. Webgpt: Browser-assisted question-answering with human feedback. *arXiv preprint arXiv:2112.09332*, 2021.

[36] NVIDIA. Nvidia collective communications library (nccl). https://developer.nvidia.com/nccl, 2025.

[37] OpenAI. Learning to reason with llms. https://openai.com/index/learning-to-reason-with-llms/, 2025.

[38] Long Ouyang, Jeffrey Wu, Xu Jiang, Diogo Almeida, Carroll Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, Alex Ray, et al. Training language models to follow instructions with human feedback. *Advances in neural information processing systems*, 35:27730–27744, 2022.

[39] PyTorch. Gloo: Collective communications library. https://github.com/pytorch/gloo, 2025.

[40] Qwen, :, An Yang, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chengyuan Li, Dayiheng Liu, Fei Huang, Haoran Wei, Huan Lin, Jian Yang, Jianhong Tu, Jianwei Zhang, Jianxin Yang, Jiaxi Yang, Jingren Zhou, Junyang Lin, Kai Dang, Keming Lu, Keqin Bao, Kexin Yang, Le Yu, Mei Li, Mingfeng Xue, Pei Zhang, Qin Zhu, Rui Men, Runji Lin, Tianhao Li, Tianyi Tang, Tingyu Xia, Xingzhang Ren, Xuancheng Ren, Yang Fan, Yang Su, Yichang Zhang, Yu Wan, Yuqiong

Liu, Zeyu Cui, Zhenru Zhang, and Zihan Qiu. Qwen2.5 technical report, 2025.

[41] Rafael Rafailov, Archit Sharma, Eric Mitchell, Christopher D Manning, Stefano Ermon, and Chelsea Finn. Direct preference optimization: Your language model is secretly a reward model. *Advances in Neural Information Processing Systems*, 36, 2023.

[42] Samyam Rajbhandari, Jeff Rasley, Olatunji Ruwase, and Yuxiong He. Zero: Memory optimizations toward training trillion parameter models. In *Proceedings of SC*, pages 1–16, 2020.

[43] Samyam Rajbhandari, Olatunji Ruwase, Jeff Rasley, Shaden Smith, and Yuxiong He. Zero-infinity: Breaking the gpu memory wall for extreme scale deep learning. In *Proceedings of SC*, pages 1–14, 2021.

[44] Jeff Rasley, Samyam Rajbhandari, Olatunji Ruwase, and Yuxiong He. Deepspeed: System optimizations enable training deep learning models with over 100 billion parameters. In *Proceedings of SIGKDD*, pages 3505–3506, 2020.

[45] David Rein, Betty Li Hou, Asa Cooper Stickland, Jackson Petty, Richard Yuanzhe Pang, Julien Dirani, Julian Michael, and Samuel R. Bowman. GPQA: A graduate-level google-proof q&a benchmark. In *First Conference on Language Modeling*, 2024.

[46] Jie Ren, Samyam Rajbhandari, Reza Yazdani Aminabadi, Olatunji Ruwase, Shuangyan Yang, Minjia Zhang, Dong Li, and Yuxiong He. Zero-offload: Democratizing billion-scale model training. In *Proceedings of ATC*, pages 551–564, 2021.

[47] Microsoft Research. rStar2-Agent: Agentic reasoning technical report. *arXiv preprint*, 2025.

[48] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017.

[49] Zhihong Shao, Peiyi Wang, Qihao Zhu, Runxin Xu, Junxiao Song, Xiao Bi, Haowei Zhang, Mingchuan Zhang, YK Li, Yang Wu, et al. Deepseekmath: Pushing the limits of mathematical reasoning in open language models. *arXiv preprint arXiv:2402.03300*, 2024.

[50] Gerald Shen, Zhilin Wang, Olivier Delalleau, Jiaqi Zeng, Yi Dong, Daniel Egert, Shengyang Sun, Jimmy Zhang, Sahil Jain, Ali Taghibakhshi, et al. Nemo-aligner: Scalable toolkit for efficient model alignment. *arXiv preprint arXiv:2405.01481*, 2024.

[51] Guangming Sheng, Chi Zhang, Zilingfeng Ye, Xibin Wu, Wang Zhang, Ru Zhang, Yanghua Peng, Haibin Lin, and Chuan Wu. Hybridflow: A flexible and efficient rlhf framework. In *Proceedings of the Twentieth European Conference on Computer Systems*, pages 1279–1297, 2025.

[52] Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. Megatron-lm: Training multi-billion parameter language models using model parallelism. *arXiv preprint arXiv:1909.08053*, 2019.

[53] Mingyang Song, Mao Zheng, Zheng Li, Wenjie Yang, Xuan Luo, Yue Pan, and Feng Zhang. Fastcurl: Curriculum reinforcement learning with stage-wise context scaling for efficient training r1-like reasoning models, 2025.

[54] Stone Tao, Fanbo Xiang, Arth Shukla, Yuzhe Qin, Xander Hinrichsen, Xiaodi Yuan, Chen Bao, Xinsong Lin, Yulin Liu, Tse kai Chan, Yuan Gao, Xuanlin Li, Tongzhou Mu, Nan Xiao, Arnav Gurha, Viswesh Nagaswamy Rajesh, Yong Woo Choi, Yen-Ru Chen, Zhiao Huang, Roberto Calandra, Rui Chen, Shan Luo, and Hao Su. Maniskill3: Gpu parallelized robotics simulation and rendering for generalizable embodied ai. *Robotics: Science and Systems*, 2025.

[55] Physical Intelligence team. openpi holds open-source models and packages for robotics. https://github.com/Physical-Intelligence/openpi, 2025.

[56] Andrew Wagenmaker, Mitsuhiko Nakamoto, Yunchu Zhang, Seohong Park, Waleed Yagoub, Anusha Nagabandi, Abhishek Gupta, and Sergey Levine. Steering your diffusion policy with latent space reinforcement learning, 2025.

[57] Team Wan, Ang Wang, Baole Ai, Bin Wen, Chaojie Mao, Chen-Wei Xie, Di Chen, Feiwu Yu, Haiming Zhao, Jianxiao Yang, Jianyuan Zeng, Jiayu Wang, Jingfeng Zhang, Jingren Zhou, Jinkai Wang, Jixuan Chen, Kai Zhu, Kang Zhao, Keyu Yan, Lianghua Huang, Mengyang Feng, Ningyi Zhang, Pandeng Li, Pingyu Wu, Ruihang Chu, Ruili Feng, Shiwei Zhang, Siyang Sun, Tao Fang, Tianxing Wang, Tianyi Gui, Tingyu Weng, Tong Shen, Wei Lin, Wei Wang, Wei Wang, Wenmeng Zhou, Wente Wang, Wenting Shen, Wenyuan Yu, Xianzhong Shi, Xiaoming Huang, Xin Xu, Yan Kou, Yangyu Lv, Yifei Li, Yijing Liu, Yiming Wang, Yingya Zhang, Yitong Huang, Yong Li, You Wu, Yu Liu, Yulin Pan, Yun Zheng, Yuntao Hong, Yupeng Shi, Yutong Feng, Zeyinzi Jiang, Zhen Han, Zhi-Fan Wu, and Ziyu Liu. Wan: Open and advanced large-scale video generative models, 2025.

[58] Huggingface xDAN-datasets (xDAN Back). xdan-datasets/areal-boba-data. https://huggingface.

co/datasets/xDAN-datasets/AReaL-boba-Data, 2025.

[59] An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengen Huang, Chenxu Lv, et al. Qwen3 technical report. *arXiv preprint arXiv:2505.09388*, 2025.

[60] Zhewei Yao, Reza Yazdani Aminabadi, Olatunji Ruwase, Samyam Rajbhandari, Xiaoxia Wu, Ammar Ahmad Awan, Jeff Rasley, Minjia Zhang, Conglong Li, Connor Holmes, et al. Deepspeed-chat: Easy, fast and affordable rlhf training of chatgpt-like models at all scales. *arXiv preprint arXiv:2308.01320*, 2023.

[61] Qiying Yu, Zheng Zhang, Ruofei Zhu, Yufeng Yuan, Xiaochen Zuo, Yu Yue, Weinan Dai, Tiantian Fan, Gaohong Liu, Lingjun Liu, et al. Dapo: An open-source llm reinforcement learning system at scale. *arXiv preprint arXiv:2503.14476*, 2025.

[62] Matei Zaharia, Reynold S Xin, Patrick Wendell, Tathagata Das, Michael Armbrust, Ankur Dave, Xiangrui Meng, Josh Rosen, Shivaram Venkataraman, Michael J Franklin, et al. Apache spark: a unified engine for big data processing. *Communications of the ACM*, 59(11):56–65, 2016.

[63] Yifan Zhang and Team Math-AI. American invitational mathematics examination (aime) 2024, 2024.

[64] Yifan Zhang and Team Math-AI. American invitational mathematics examination (aime) 2025, 2025.

[65] Yixian Zhang, Shu'ang Yu, Tonghe Zhang, Mo Guang, Haojia Hui, Kaiwen Long, Yu Wang, Chao Yu, and Wenbo Ding. Sac flow: Sample-efficient reinforcement learning of flow-based policies via velocity-reparameterized sequential modeling, 2026.

[66] Yanli Zhao, Andrew Gu, Rohan Varma, Liang Luo, Chien-Chin Huang, Min Xu, Less Wright, Hamid Shojanazeri, Myle Ott, Sam Shleifer, et al. Pytorch fsdp: experiences on scaling fully sharded data parallel. *arXiv preprint arXiv:2304.11277*, 2023.

[67] Lianmin Zheng, Zhuohan Li, Hao Zhang, Yonghao Zhuang, Zhifeng Chen, Yanping Huang, Yida Wang, Yuanzhong Xu, Danyang Zhuo, Eric P Xing, et al. Alpa: Automating inter-and {Intra-Operator} parallelism for distributed deep learning. In *Proceedings of OSDI*, pages 559–578, 2022.

[68] Lianmin Zheng, Liangsheng Yin, Zhiqiang Xie, Chuyue Sun, Jeff Huang, Cody Hao Yu, Shiyi Cao, Christos Kozyrakis, Ion Stoica, Joseph E. Gonzalez, Clark Barrett, and Ying Sheng. Sglang: Efficient execution of structured language model programs, 2024.

[69] Yuxiang Zheng, Dayuan Fu, Xiangkun Hu, Xiaojie Cai, Lyumanshan Ye, Pengrui Lu, and Pengfei Liu. Deepresearcher: Scaling deep research via reinforcement learning in real-world environments. *arXiv preprint arXiv:2504.03160*, 2025.

[70] Zangwei Zheng, Xiangyu Peng, Tianji Yang, Chenhui Shen, Shenggui Li, Hongxin Liu, Yukun Zhou, Tianyi Li, and Yang You. Open-sora: Democratizing efficient video production for all, 2024.

[71] Zilin Zhu, Chengxing Xie, Xin Lv, and slime Contributors. slime: An llm post-training framework for rl scaling. https://github.com/THUDM/slime, 2025. GitHub repository. Corresponding author: Xin Lv.

ARTIFACT EVALUATED
USENIX
AVAILABLE

DynaRL: Flexible and Dynamic Scheduling of Large-Scale Reinforcement Learning Training

Yuanqing Wang[1,2], Hao Lin[2], Junhao Hu[2], Chunyang Zhu[2], Quanlu Zhang[2], Zhen Guo[2], Yuchen Zhang[5,2], Xu Fu[2], Si Xu[2], Bo Dai[6,2], Zixiao Huang[3,2], Chao Yu[3], Boxun Li[2], Guohao Dai[7,2], Zhi Yang[1], Yu Wang[3]
[1]Peking University [2]Infinigence AI [3]Tsinghua University [4]Zhongguancun Academy
[5]Institute of Computing Technology, CAS [6]Beihang University [7]Shanghai Jiao Tong University

Abstract

Modern reinforcement learning (RL) workloads, powering large language models, long-horizon reasoning, and agentic systems, exhibit extreme dynamicity due to heavy-tailed rollouts, irregular multi-turn tool interactions, and time-varying bottlenecks. Static resource allocations in today's distributed RL systems leave large fractions of compute idle and prolong training. This paper presents DynaRL, the first RL system that dynamically reallocates computation, memory, and communication resources across heterogeneous RL components. DynaRL models the entire RL pipeline with a dynamic hypergraph that serves as a centralized, continuously-evolving control surface. Supported by a unified resource migration interface and context-aware data routing, the scheduler reallocates GPUs from overprovisioned components to the current bottleneck via a combination of a multi-level scheduling algorithm and fine-grained resource migration. Comprehensive evaluation demonstrates that DynaRL improves end-to-end throughput on math-reasoning and agentic RL workloads by up to 1.98×, with negligible online scheduling overhead.

1 Introduction

Reinforcement Learning (RL) has rapidly expanded from traditional control and game-playing tasks [6, 7] to the foundation of modern reasoning and agentic systems [2, 15]. Today's RL workloads power large language/vision models, long-horizon planning agents [1], and tool-augmented agentic systems. As these workloads grow in scale and complexity, training increasingly relies on distributed clusters of hundreds of GPUs. However, unlike conventional deep learning workloads, RL pipelines integrate heterogeneous components, e.g., rollout generation, inference, multi-step tool calls, environment interactions, and policy optimization training, that interact in highly irregular and time-varying patterns.

This heterogeneity introduces substantial *dynamicity* throughout the pipeline. Rollout generation involves autoregressive model inference [17] whose response lengths exhibit heavy-tailed distributions, causing long-tail stragglers and significant GPU underutilization. Agentic RL amplifies this effect. Models repeatedly invoke external tools such as search engines, Python interpreters, or theorem provers [10], producing unpredictable multi-turn interaction traces that vary in both length and latency. These irregular interactions disrupt scheduling and create bottlenecks that cascade across the training pipeline. As our characterization shows, such dynamicity can waste up to 60% of available compute and create large oscillations in effective throughput.

Unfortunately, existing RL systems are largely static. They allocate GPUs to each component ahead of time and maintain this allocation throughout RL training. This static partitioning fundamentally mismatches the time-varying demands of modern RL workloads. When long-tail rollouts dominate, the GPUs assigned to rollouts become increasingly underutilized; when multi-turn interactions inflate generation load, rollout workers become severely underprovisioned; and when tool calls bottleneck progress, KV caches in inference engines are repeatedly evicted and rebuilt. The result is unstable throughput, inefficient resource usage, and prolonged training time.

This paper argues that *dynamic scheduling must become a first-class design principle* for next-generation distributed RL systems. RL training is inherently multi-component and feedback-driven, i.e., progress in one stage depends on the outputs and timing of others. If a system can observe runtime signals, detect shifts in bottlenecks, and reallocate resources in real time, while respecting dependencies and preserving training state, it can significantly improve end-to-end training efficiency and stability. However, enabling such dynamic management is challenging. RL components differ widely in structure, statefulness, and communication patterns; migrations must occur safely without violating components' execution semantics; and the training pipeline must remain efficient despite constantly shifting parallelism.

We present DynaRL, a unified runtime and scheduling system that dynamically reallocates computation, memory, and communication resources across heterogeneous RL components. At a high level, DynaRL *captures the entire RL pipeline*

in a single, continuously evolving runtime object and exposes it to the scheduler as a dynamic hypergraph that encapsulates both static structure and time-varying execution signals. Alongside this global control surface, DynaRL provides a unified resource-migration interface that lets the scheduler safely resize the parallelism of diverse components at interruptible points, using statefulness-aware migration strategies. Finally, it embeds context-aware data routing to attach semantic affinity and multi-step metadata to dataflow edges, preserving KV-cache locality and enabling multi-turn-aware request scheduling.

Built on these mechanisms, DynaRL employs a multi-level dynamic scheduling algorithm that detects overprovisioned components from hypergraph signals and reallocates their GPUs to the current bottleneck stage, by jointly adjusting resource allocation across components, within each WorkerGroup, and at the request data level, where it prioritizes short, late-stage tool calls to exploit KV-cache reuse and accelerate episode completion.

We implement DynaRL in around 7K lines of Python atop RLinf [22], an open-source general RL framework, integrating with existing inference engines (e.g., SGLang [32]) and training backends (e.g., Megatron-LM [20]). Our experiments on math-reasoning and multi-turn agentic RL workloads show that DynaRL consistently improves throughput over state-of-the-art RL systems such as verl and RLinf by up to 1.98× while adding less than 1% overhead for online scheduling and migration. We plan to release the full codebase of DynaRL.

This paper makes the following contributions:

- We characterize the dynamicity of modern RL workloads, demonstrating substantial long-tail rollouts, irregular multi-turn tool-use patterns, and resulting system inefficiencies across production-scale deployments.
- We design and implement DynaRL, the first system to realize dynamic scheduling for distributed RL pipelines via a unified control surface, safe migration mechanisms, and overprovision-driven scheduling.
- We comprehensively evaluate DynaRL on math-reasoning and multi-turn agentic RL workloads, showing significant throughput improvements over state-of-the-art RL systems.

2 Background and Motivation

2.1 Diverse RL Workloads

Distributed reinforcement learning. Reinforcement Learning (RL) has evolved from its early use in small-scale game-playing and robotic control to a key technique for training large foundation models for reasoning and decision-making. These workloads increasingly run in distributed settings to meet the compute and data demands of large models. Modern RL pipelines exhibit substantial structural diversity, as shown in Figure 1. A typical workflow comprises generation (producing rollouts given prompts), inference (computing log probabilities), and training (updating model parameters). However, the composition and dependencies of these components vary widely across algorithms. For example, GRPO contains only three major components, whereas PPO-based RLHF requires a substantially more complex seven-component pipeline with more data and control dependencies. This variability complicates system design and resource scheduling.

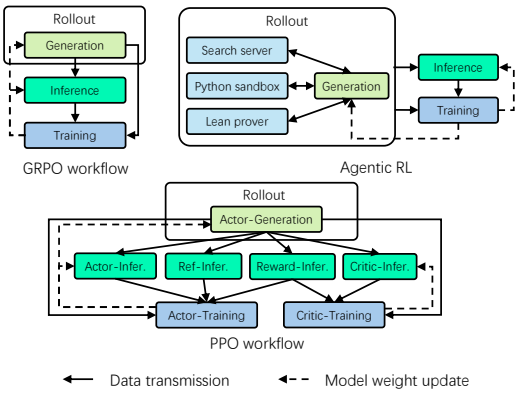


Figure 1: Diverse RL workflows in various scenarios.

Agentic reinforcement learning. Recent agentic RL approaches further expand workflow complexity by integrating external tools into the rollout process. Systems such as DeepResearch combine LLM generation with search engines, databases, and iterative refinement loops; math-reasoning pipelines may involve Python interpreters or theorem provers such as Lean [10]. These tool-augmented rollouts introduce fine-grained and highly irregular dependencies between components. As a result, the training pipeline frequently stalls while waiting for upstream results, creating severe workload imbalance and resource underutilization.

2.2 The Dynamicity Challenge of RL

Dynamic behavior is inherent and pervasive in RL training. An RL pipeline repeatedly generates solutions for given tasks, e.g., responses to prompts or actions to perform, which can vary significantly due to differing response lengths, unpredictable numbers of reasoning steps, and on-demand tool usages. In this section, we quantitatively analyze such dynamicity in production RL workloads and pinpoint two major sources of dynamicity in RL training, namely long-tail model generation and irregular external interactions.

Dynamicity in long-tail generation. Rollout generation with model inference is highly unpredictable—generation response length can vary significantly across queries, which is positively related to the computation time due to the auto-regressive nature of most modern foundation models. To deeply understand this, we studied the variation in gener-

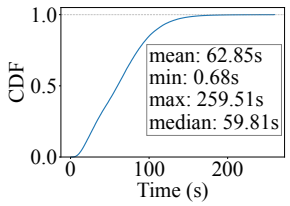


Figure 2: Decode time in a math-reasoning RL task.

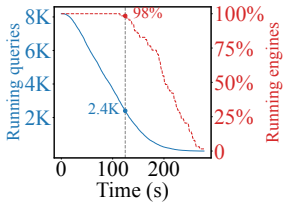


Figure 3: Running query and inference engine variation in a math-reasoning RL task.

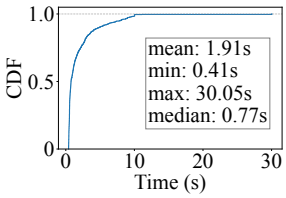


Figure 4: Tool call time in an agentic RL task.

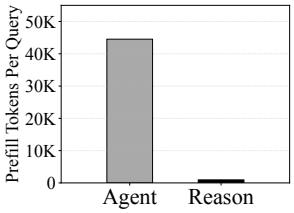


Figure 5: Average prefill tokens in a typical agentic RL and single-turn math RL.

ation response length in a typical math-reasoning RL task on a 1.5B-parameter language model using the GRPO algorithm, which involves a minimal number of components and no other interfering dynamicity like tool calls.

As shown in Figure 2, the decode time of math queries shows a skewed distribution. Only a small fraction of the queries reach near the maximum decode time (259.51s), while roughly 95% of the queries finish in under 150s (about 60% of that limit). Figure 3 further demonstrates the variation of running queries and running inference engine ratio (among 64 engines) during the rollout generation process.

Specifically, the blue curve tracks the number of active queries, while the red curve indicates the active engine ratio at the same time, which represents the percentage of engines still processing incomplete requests. As shown, at the middle of the generation, only less than 30% of queries remain unprocessed. The long-tail responses stall the entire rollout stage and introduce substantial computation waste. Moreover, because long-tail responses are evenly scattered across inference engine instances, many instances are highly underutilized, being stuck on the remaining one or two responses, and thus cannot be terminated to release resources. As in Figure 3, even when $<$30% of queries remain, almost all engines are still running. This idle time and underutilization can waste up to 60% of available computation, without considering KV-cache memory usage.

Dynamicity in irregular external interactions. Apart from generation dynamicity, emerging RL scenarios like agentic RL and embodied RL introduce another major source of dynamicity—irregular external interactions. Based on the generation results (in the form of commands or actions), the training pipeline needs to interact with external tools or environments like search engines, Python interpreters, program compilers, etc., to handle more complex tasks.

Such interactive workflows mostly follow the ReAct paradigm [28], where the model generates an intermediate output, optionally invokes tools, concatenates the tool results with prior context, and uses that as the next-turn input. Figure 4 shows the tool call time in a multi-turn agentic RL setting, which also yields a highly skewed distribution.

Worse still, the default scheduling policy in popular inference engines (e.g., SGLang and vLLM), i.e., First-In-First-Out, or FIFO, may further aggravate the resource waste problem. Engines such as SGLang mark a completed query's KV cache as evictable and can only reuse it if the next-turn input is scheduled promptly before it is evicted. Under FIFO scheduling, however, the second turn of the first query is often delayed for an unpredictable amount of time as it waits for all the first-turn interactions of the remaining queries to finish, causing severe KV cache thrashing and re-prefilling of tokens. Figure 5 compares the number of prefilled tokens per query during the training of agentic RL with multi-turn interactions and math-reasoning RL without interactions using SGLang. Under the same KV cache memory threshold (100 GB), we can clearly observe that tool-call interactions induce considerable re-prefilling due to KV cache thrashing.

2.3 Dynamic Scheduling as the Core

The increasing dynamicity in modern RL workflows has presented serious challenges for existing RL systems, which mostly rely on static resource partitioning for resource allocation—computation resources like GPUs are either physically divided or temporally multiplexed among different components ahead of training. With foundation models beginning to interact more and more with the real world to solve complex tasks, the dynamicity challenge motivates us to rethink the design paradigm of distributed RL systems.

We recognize that the RL training process involves multiple heterogeneous components, including rollout generation, tool interactions, inference, and policy optimization, which interact in complex and time-varying ways.

While this multi-component structure complicates scheduling, it also presents a key opportunity: resources can be dynamically reallocated to match the workload demands, improving overall efficiency. For example, when rollout generation is bottlenecked by long-tail responses, resources can be shifted to accelerate training.

Consequently, we argue that to achieve high efficiency and stable convergence, a distributed RL system must be able to dynamically coordinate computation, memory, and communication resources across nodes and GPUs. This requires a runtime that can observe, predict, adapt to workload variations

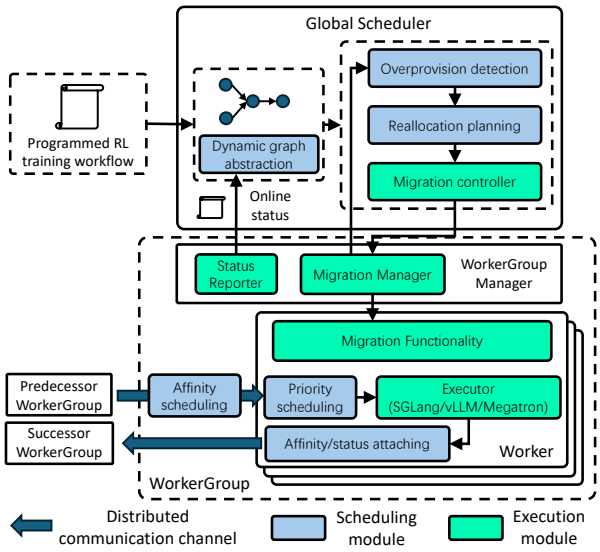


Figure 6: The system architecture of DynaRL.

```python
class HyperNode:
    # static node attributes
    component_type   # dynamic or static
    interruptible    # always, regular, or never
    predecessors: HyperNode[]
    successors: HyperNode[]
    # dynamic node status
    nodes: Node[]    # list of homogeneous nodes
class Node:
    allocated_resources: Resource[]
    utilization: number
class HyperEdge:     # dataflow between nodes
    affinity: Node[] # the preferred nodes
    progress         # data processing progress
#dynamic schedule rollout/train_workers
dynamic_schedule(graph: HyperNode[], resource:
 ↪ Resource[])
```

Figure 7: The definition of the dynamic graph.

in real time, and balance resource usage without disrupting the delicate feedback loops in RL training, posing a unique and complex systems challenge that demands a joint understanding of both system-level performance and algorithmic stability. Therefore, beyond an optimization, we envision dynamic scheduling to be a core design principle and challenge for next-generation distributed RL systems.

3 DynaRL Overview

DynaRL is a unified runtime and scheduling system designed to dynamically reallocate resources across heterogeneous RL components, with the goal of maximizing the end-to-end throughput of RL training pipelines whose components' load shifts rapidly over time.

As shown in Figure 6, DynaRL runs as the scheduling plane of a general RL framework, e.g., verl [19] or RLinf [22], which provides the basic infrastructures for composing RL training pipelines and defining diverse components. In such a framework, an RL component (e.g., rollout, tool caller, trainer) is implemented as a group of distributed *Workers* (termed *WorkerGroup*), each of which is a process running on the allocated compute resources (e.g., GPUs). The framework handles the low-level details of assigning Workers to resources, establishing communication channels among Workers, and executing the distributed program, but assumes a static resource allocation throughout training.

DynaRL runs atop the framework also as a WorkerGroup like other components, which contains a *Global Scheduler* and multiple *WorkerGroup Managers* (one per WorkerGroup). Together, they form a feedback-driven control loop. Runtime signals collected from Workers are reported to the Global Scheduler by the WorkerGroup Managers; the Global Scheduler then computes new allocations based on these signals; and the WorkerGroup Managers enact the decisions with minimal disruption to the ongoing training. This design decouples scheduling from the specific implementation of diverse components, allowing DynaRL to continuously adapt to shifting rollout length distributions, changing inference workloads, and evolving training bottlenecks.

In the following sections, we first present the system-level designs and mechanisms that enable dynamic scheduling in DynaRL (§4), and then describe the core scheduling algorithm that drives resource reallocation (§5).

4 DynaRL System Design

Fundamentally, dynamic scheduling can be viewed as online resource migration across components. A component experiencing resource underutilization can release part of its allocated resources (usually by scaling down) to other components that are bottlenecked (by scaling up).

However, the intricate dependencies and stateful nature of RL training pipelines pose systematic challenges to resource migration at *abstraction, resource, and data levels*: (1) **Abstraction level**, how to effectively capture both static structure and runtime variability of RL pipelines, enabling the scheduler to reason about component dependencies and performance dynamics? (2) **Resource level**, how to safely enact resource migration at runtime without disrupting training correctness, when delicate stateful components such as a distributed trainer with cumulated gradients and optimizer states are involved? (3) **Data level**, how to enable efficient data flows for dynamic and interactive components (e.g., multi-turn rollout) under complex dependencies and caching effects?

To address these, the key design of DynaRL is to *transform*

complex, heterogeneous RL pipelines into a single, controllable runtime object that enables fine-grained, low-overhead rescaling and rebalancing:

- A dynamic hypergraph abstraction that captures both structure and runtime signals.
- Unified interfaces for safe online resource migration and context-aware data routing.

Together, they allow DynaRL to perform predictive, low-overhead, and component-agnostic scheduling with minimal disruption to ongoing training.

4.1 The Dynamic HyperGraph Abstraction

To capture both static structure and runtime variability, DynaRL models the RL pipeline as a continuously-updated dynamic hypergraph, i.e., a special graph where a node (termed HyperNode) contains a set of homogeneous nodes, and an edge (termed HyperEdge) can connect multiple HyperNodes beyond the traditional one-to-one connection in common graphs. As shown in Figure 7, a HyperNode represents the WorkerGroup of a logical component (e.g., rollout, trainer) along with a set of homogeneous nodes that represent its active Worker instances. Each HyperNode maintains both static metadata (e.g., component type, interruptibility, dependency constraints) and runtime metrics collected from its corresponding WorkerGroup, including resource utilization, queue pressure, and per-worker throughput.

HyperEdges encode dataflow between Workers/nodes and their relevant context information, including a data processing task's progress and affinity with Workers, i.e., which Workers it might prefer or require to run on. This allows tracking the readiness of dependent components and pipeline backpressure, enabling the scheduler to detect emerging bottlenecks and rebalance resources proactively. Also, the recorded context information can facilitate data redistribution after migration, allowing the scheduler to reason not only about component-level bottlenecks, but also about the structural and temporal dynamics of data processing task propagation.

The unified dynamic graph serves as a single, continuously-updated control surface between the scheduler and the runtime system. Downstream, it gathers fine-grained signals such as GPU load, rollout generation speed, and training throughput from each worker. Upstream, it exposes a consistent abstraction that allows the scheduling policy to reason over heterogeneous components without entanglement with component-specific details. By decoupling policy from implementation, the dynamic graph provides the foundation for predictive, low-overhead adaptations such as resizing component parallelism, reallocating GPUs, or throttling request generation, and a principled representation that captures both the structural and temporal dynamics needed for robust online scheduling.

```python
# The unified migration interface
class MigrationManager:
    def migrate(dest_resources)
    def interrupt()

# Migration strategy interfaces
class RebootMigration:
    def shutdown()
    def restart(dest_resources)
class WorkloadMigration:
    def data_collect()
    def data_distribute()
    def suspend()
class p2pMigration:
    def create()
    def state_switch()
    def suspend()
```

Figure 8: The unified interface of resource migration.

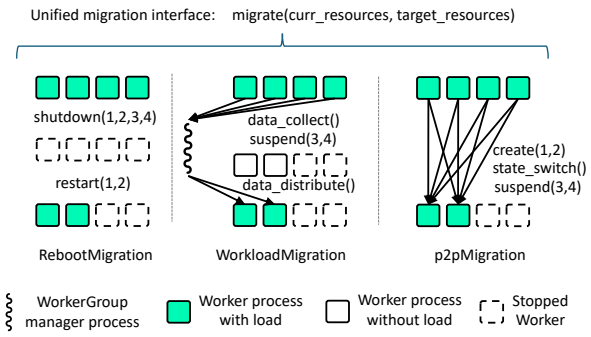


Figure 9: Three types of migration with unified interface.

4.2 Resource Migration

Resource migration is the core mechanism that enables resource rebalancing among components. However, the diverse characteristics of RL components lead to widely varying migration procedures, particularly for stateful components that maintain inter-Worker states or have complex communication patterns. For example, inference engines such as SGLang and vLLM are mostly isolated instances that do not have shared states across Workers, allowing safe migration via process suspension. In contrast, training frameworks such as Megatron-LM maintain tightly coupled states across Workers, e.g., distributed model parameters and optimizer buffers that require synchronization points to be reached before migration.

To accommodate this diversity while maintaining a clean separation between scheduling policy and component-specific details, DynaRL defines a unified migration interface as in Figure 8 that abstracts the resource migration process. The key for safe migration is to identify *interruptible points* in the component's execution flow where migration can occur

without violating correctness. Thus, the interface follows an *interrupt-schedule-migrate* pattern akin to cooperative multitasking in operating systems: (1) The manager process of a WorkerGroup signals the scheduler when it reaches an interruptible point via `interrupt()`. (2) The scheduler determines if a rebalancing is required, (if required) checks for other interruptible components, computes a new resource allocation for all the interruptible components, and instructs the manager to execute `migrate(dest_resources)` with the target resources. (3) The manager performs the migration based on its component-specific migration strategy.

Specifically, certain components (such as the inference engine) are designed to be always interruptible. The `interruptible` attribute in the HyperNode is then set to `always`, allowing the scheduler to trigger migration at any time rather than waiting for specific interruptible points.

Migration Strategy. We identify three fundamental migration strategies that cover a wide range of RL components based on the WorkerGroup statefulness and migration performance, as illustrated in Figure 9. The first is `RebootMigration`, which simply shuts down previous Workers on the source resources, preserves the necessary states, and restarts them on the target resources with the preserved states. This is a generic approach that can work for almost all components, albeit with potentially high overhead. Thus, it is more suitable for lightweight components with negligible restart overhead.

The second is `WorkloadMigration`, which suspends the Workers running on resources to be released, collects their pending and ongoing data processing tasks, and redistributes them to the remaining Workers on the retained resources. This approach is used by stateless components such as inference engines (e.g., SGLang, vLLM) with suspension capability that offload their occupied hardware resources on suspension, allowing safe migration without process restarts.

The third is `p2pMigration`, which launches new Workers on the target resources while keeping the old Workers active. State transfer (e.g., model parameters and optimizer buffers) is performed via planned peer-to-peer communication paths between old and new Workers, after which the old Workers are shut down or suspended. Stateful components such as Megatron-LM leverage this decentralized approach, enabling rapid migration within roughly one second for most cases.

The `p2pMigration` strategy follows a *create-before-destroy* pattern to guarantee atomicity: new Workers are fully initialized and receive complete state transfers before old Workers are suspended or torn down. Specifically, a distributed barrier is enforced before `state_switch()` is invoked. All new-rank Workers must acknowledge full receipt of model parameters, optimizer states, and gradient buffers before the system proceeds. If any transfer fails (e.g., due to a network partition or hardware fault on the destination), the migration is aborted: the newly created Workers are discarded, and the old Workers remain active with their original state intact, effectively rolling back to the previous allocation. This ensures there is no window in which both old and new Workers are partially active for training, and the global training state remains consistent throughout the migration. Only after all new ranks have confirmed readiness does the system atomically switch the active set, tear down the old Workers, and resume training on the new allocation.

```
class DataRouter:
    # HyperEdge
    # affinity: required, preferred, noneed
    def attach_affinity(output, affinity)
    # status: steps, etc
    def attach_status(output, status)
    # requests scheduling
    def distribute_data(data)
    def priority_schedule(data)
```

Figure 10: The interface of the local data router.

Additionally, to handle potential anomalies such as Worker hangs or infinite loops during execution, each Worker in DynaRL runs a background daemon that serves as a heartbeat monitor to receive scheduling commands. Once an anomaly is detected, the scheduler can invoke `RebootMigration` to forcefully shut down the unresponsive previous Workers and restart them on healthy resources.

DynaRL supports all three migration strategies through the unified interface, allowing each WorkerGroup to implement its own migration logic based on its component properties. The interface is shown in Figure 8. We implement the three migration strategies for common RL components including rollout, train, and tool caller as detailed in §6, demonstrating its versatility and generalizability.

4.3 Context-Aware Data Routing

RL training features tightly coupled interactions among components, modeled as HyperEdges in DynaRL. Each HyperEdge aggregates data flowing from one WorkerGroup to another. Efficiently routing these data across and within Workers is essential for end-to-end throughput. For example, in multi-turn rollout, requests belonging to the same episode of interactions may benefit from being processed on the same Worker to leverage the inference engine's KV cache and reduce re-prefilling.

To this end, DynaRL introduces context-aware data routing to realize efficient data flows. The core enabler of this mechanism is a local data router that runs within each WorkerGroup (as a thread or a coroutine), which extracts Worker-specific context and attaches it to outgoing data. Before a Worker forwards data to its successor WorkerGroup, the local router annotates each data item with: (1) affinity infor-

mation indicating where the data was last processed and whether it prefers, requires, or need not use the same Worker (`attach_affinity`); and (2) status metadata, such as sequence IDs that track how many steps the data has progressed across Workers (`attach_status`). The interface is extensible and allows additional metadata to be attached as needed.

With these annotations, the local router governs both inter-Worker data distribution and intra-Worker data ordering. As WorkerGroups communicate through distributed communication channels provided by RL frameworks, the router's `distribute_data` function intercepts outgoing queues to route data to appropriate destination Workers based on affinity constraints. On the receiving side, each Worker invokes `priority_schedule` to decide the execution order of incoming data, enabling fine-grained control under dynamic workloads, particularly in rollout-heavy stages. For example, the SGLang Worker in DynaRL implements a custom priority scheduling policy through this interface to optimize multi-turn interactions (§5).

This context-aware data routing layer allows the system to adapt to the dynamic nature of RL training while preserving efficient data flow across distributed components.

5 DynaRL Scheduling

The goal of DynaRL's scheduling algorithm is to maximize end-to-end RL throughput by dynamically reallocating resources among heterogeneous components, including rollout, inference, and model training, whose resource-performance characteristics diverge substantially over time.

The key idea behind our scheduling policy is to use resource overprovisioning as the primary scheduling signal. A component is considered overprovisioned when (1) allocating additional resources yields only marginal improvements in step-level or end-to-end throughput, and (2) removing a modest amount of resources does not noticeably degrade its performance. Once overprovisioning is detected, DynaRL safely reclaims these underutilized resources and migrates them to the component that currently limits the pipeline, thereby improving overall throughput.

To reason about where overprovision is likely to occur and where reclaimed resources can be most effectively used, DynaRL distinguishes between two types of components—**static** and **dynamic** components. Static components have throughput that scales positively with additional devices (e.g., actor or critic training with abundant input examples), so extra resources directly translate into higher throughput. In contrast, dynamic components exhibit a concave resource-performance curve whose marginal benefit naturally changes as the workload evolves (e.g., rollout whose active sequence count shrinks as trajectories complete), making them prone to entering an overprovisioned regime where additional resources provide little incremental gain.

This design naturally leads to a three-level scheduling framework: *WorkerGroup*-level scheduling on how much resource to allocate across components/WorkerGroups, *Worker*-level scheduling that decides how to balance workload and scale up / down across Workers in one WorkerGroup, and *data*-level scheduling on the distribution and priority of request data for a specific component.

5.1 Online Migration Scheduling

Consider a system with a set of components $\mathcal{C} = \mathcal{C}_S \cup \mathcal{C}_D$, where $\mathcal{C}_S$ is the set of static components and $\mathcal{C}_D$ is the set of dynamic components. The system has a fixed pool of R_{total} homogeneous resources (e.g., GPUs).

- Let $R_i(t)$ be the number of resources allocated to component C_i at time t, with $\sum_{C_i \in \mathcal{C}} R_i(t) \leq R_{\text{total}}$.
- Each component C_i has a throughput function $T_i(R_i)$, which measures its ideal processing rate (e.g., samples per second) when allocated R_i resources.
- For a static component $C_S \in \mathcal{C}_S$, $T_S(R_S)$ is approximately linear: $T_S(R_S) \approx \alpha_S R_S$.
- For a dynamic component $C_D \in \mathcal{C}_D$, $T_D(R_D)$ is a concave function. It has an overprovision point R_D^{oa} where the marginal throughput gain diminishes significantly: $\frac{\partial T_D}{\partial R_D} \ll \alpha_S$ for $R_D > R_D^{\text{oa}}$.

The goal of the scheduler is to find a resource allocation $\mathcal{R} = (R_1, R_2, ...)$ at any given time that maximizes the end-to-end throughput of the pipeline. Since the pipeline is synchronous, the overall throughput is the minimum of the throughputs of its constituent stages (if we ignore the small asynchrony in a pipeline). So the objective of the scheduling algorithm is:

$$\max_{\mathcal{R}} \min_{C_i \in \mathcal{C}} T_i(R_i) \quad \text{s.t.} \quad \sum_{C_i \in \mathcal{C}} R_i \leq R_{\text{total}}. \tag{1}$$

Algorithm 1 describes the procedure of inter- and intra-worker-group resource scheduling. Our algorithm operates in a control loop, continuously monitoring the system and triggering a reallocation when inefficiencies are detected. It consists of two main phases: (1) Overprovision Detection, and (2) Reallocation Planning.

5.1.1 Phase 1: Overprovision Detection

The trigger for rescheduling is the detection of an overprovisioned dynamic component. We implement overprovision detection directly on top of the dynamic hypergraph: for each dynamic component/HyperNode C_D, DynaRL aggregates the per-node utilization reported in its `nodes` field and computes a utilization signal. For rollout components, a practical metric is the average ratio of active Key-Value (KV) cache size to its maximum capacity across all its engines. Formally, for a component with R_D engines,

Algorithm 1: Dynamic Resource Scheduler

Input: Component set $\mathcal{C}$, total resources R_{total}, thresholds $U_{\text{low}}, T, \theta$
Output: Updated resource allocation $\mathcal{R}$

1 **Function** `MonitorLoop()`**:**
2 **while** *true* **do**
3 **foreach** $C_i \in \mathcal{C}$ **do**
4 $(U_i, \textbf{metrics}_i) \leftarrow \text{monitor}(C_i)$;
5 $C_{\text{opv}} \leftarrow$ `DetectOverprovision(`$\mathcal{C}_D, U_{low}, T, \theta$`)`;
6 **if** $C_{opv} \neq None$ **then**
7 $\mathcal{R}_{\text{new}} \leftarrow$ `PlanReallocation(`$C_{opv}, \mathcal{R}, \textbf{metrics}$`)`;
8 migrate$(\mathcal{R}, \mathcal{R}_{\text{new}})$;
9 **Function** `DetectOverprovision(`$\mathcal{C}_D, U_{low}, T, \theta$`)`**:**
10 **foreach** $C_D \in \mathcal{C}_D$ **do**
11 low_count $\leftarrow 0$;
12 **foreach** *worker* $\in C_D.workers$ **do**
13 $U_{\text{avg}} \leftarrow$ avg_utilization(worker, T);
14 **if** $U_{avg} < U_{low}$ **then**
15 low_count $\leftarrow$ low_count $+ 1$;
16 **if** $low_count/|C_D.workers| > \theta$ **then**
17 **return** C_D;
18 **return** *None*;
19 **Function** `PlanReallocation(`$C_{opv}, \mathcal{R}, \textbf{metrics}$`)`**:**
20 $\{\Delta R^{(1)}, \ldots, \Delta R^{(K)}\} \leftarrow$ GenerateCandidates$(C_{\text{opv}}, \mathcal{R}, \textbf{metrics})$;
21 best $\leftarrow \varnothing$;
22 $\tau_{\text{best}} \leftarrow -\infty$;
23 **for** $k \leftarrow 1$ **to** K **do**
24 $R^{(k)}_{\text{opv}} \leftarrow \mathcal{R}[C_{\text{opv}}] - \Delta R^{(k)}$;
25 $T^{(k)}_{\text{opv}} \leftarrow$ PredictThroughput$(C_{\text{opv}}, R^{(k)}_{\text{opv}}, \textbf{metrics})$;
26 $(\text{alloc}^{(k)}, \tau^{(k)}) \leftarrow$ SearchAllocation$(\Delta R^{(k)}, C_{\text{opv}}, R^{(k)}_{\text{opv}}, T^{(k)}_{\text{opv}}, \mathcal{C}, \mathcal{R}, \textbf{metrics})$;
27 **if** $\tau^{(k)} > \tau_{best}$ **then**
28 $\tau_{\text{best}} \leftarrow \tau^{(k)}$;
29 best $\leftarrow (R^{(k)}_{\text{opv}}, \text{alloc}^{(k)})$;
30 $\mathcal{R}_{\text{new}} \leftarrow \mathcal{R}$;
31 $\mathcal{R}_{\text{new}}[C_{\text{opv}}] \leftarrow \text{best}.R_{\text{opv}}$;
32 **foreach** $C_j \in best.alloc$ **do**
33 $\mathcal{R}_{\text{new}}[C_j] \leftarrow \mathcal{R}[C_j] + \text{best.alloc}[C_j]$;
34 **return** $\mathcal{R}_{new}$;

$$U_D(t) = \frac{1}{R_D} \sum_{j=1}^{R_D} \frac{\text{KV_cache_used}_j(t)}{\text{KV_cache_capacity}_j}.$$

C_D is considered overprovisioned if $U_D(t) < U_{\text{low}}$ for a sustained period T. This indicates that most engines are underutilized. To avoid reacting to transient fluctuations, we employ a persistent condition: rescheduling is triggered only if the utilization of more than θ (e.g., 80%) of C_D's engines is below U_{low} for a continuous duration of T seconds.

In practice, U_{low} can be derived as the p-th (e.g., 20th) percentile of historical utilization data for C_D, while T and θ are hyperparameters that balance responsiveness and stability. Once C_D satisfies this overprovision predicate, the scheduler marks it as a candidate donor in the dynamic graph and proceeds to Phase 2 to plan how many resources to reclaim and where to reallocate them.

5.1.2 Phase 2: Reallocation Planning

Once a dynamic component C_D is identified as overprovisioned, the planner considers multiple resource-reduction candidates. For each candidate, the planner forecasts both the local impact on C_D and the global throughput improvement achievable by reallocating the released resources. The scheduler then selects the candidate with the highest predicted performance improvement.

1. **Generate candidate reductions**. We construct a discrete set of reduced allocation candidates $\Delta R_D^{(1)}, \cdots, \Delta R_D^{(K)}$ (line 20). They represent feasible and stable scaling levels suggested by recent utilization and throughput trends.

2. **Predict component performance**. For every allocation $R_D^{(k)}$, the planner predicts the resulting throughput $T_D^{(k)}$ using a lightweight, continuously updated performance model (line 25). This estimation provides a conservative bound on the potential performance degradation introduced by releasing $\Delta R^{(k)}$. Concretely, the throughput predictor is a statistical regression model. Before training begins, a short profiling stage measures per-component latency and throughput across representative batch sizes and sequence lengths, establishing an initial baseline model that avoids cold-start inaccuracy. During execution, the system continuously collects real-time runtime metrics (observed throughput, queue depth, batch composition) and uses them to dynamically recalibrate the model via online regression updates, ensuring predictions remain accurate as workload characteristics evolve over training.

3. **Evaluate global reallocation benefit**. For each candidate adjustment $\Delta R^{(k)}$, the planner estimates how the system would benefit from redistributing the freed resources. It allocates these resources based on each component's observed throughput characteristics and request-queue dynamics, producing a predicted end-to-end throughput $\tau^{(k)}$ for the candidate configuration (line 26).

4. **Select optimal plan**. Finally, the scheduler selects the candidate that offers the highest predicted system throughput (line 27 - 29):

$$k^* = \arg\max_k \tau^{(k)}.$$

Using the migration interface, the dynamic component is migrated to the resulting new allocation set $R'_D = R_D^{(k^*)}$, while the released resources are assigned to C_{S^*}.

This multi-candidate approach enables the scheduler to make informed trade-offs: aggressively reducing resources allocated to C_D may free more resources for other components but risks making C_D the new bottleneck. By evaluating multiple options, we ensure the reallocation maximizes throughput.

This design also keeps the reallocation behavior stable over time. Migrations are bound to interruptible points (§4.2) and thus occur at the natural batch-level divider of the RL pipeline rather than on every transient signal. Within each decision, the sustained-window predicate in Phase 1 smooths out short-lived queue spikes, and Phase 2 only commits to a new allocation when the continuously-recalibrated throughput predictor reports a strict improvement over the current one, so noise-level imbalances do not flip the decision. The migration itself rebalances in-flight requests across the new worker set as part of the procedure (e.g., `data_collect`/`data_distribute` in `WorkloadMigration`), so the new allocation lands at its predicted operating point without a queue-pressure rebound. Our sensitivity study (§7.5) further confirms that this stability holds across a wide range of scheduling parameters, so the scheduler does not rely on subtle parameter tuning.

This `PlanReallocation` step maintains low overhead as the cluster scales. It evaluates $O(K)$ candidate reductions (K is typically 3–10) across $O(|C|)$ components ($|C|$ is typically 3–7 in RL pipelines). The scheduling complexity is thus $O(K \times |C|)$, independent of the total GPU count. Furthermore, since per-Worker metrics are aggregated at the WorkerGroup Manager level, the global scheduler only processes $O(|C|)$ summaries rather than $O(N_{\text{GPU}})$ individual signals, allowing scheduling to complete within 200 ms even at 128 GPUs.

5.2 Request Data Scheduling

Algorithm 2: Priority-Aware Request Scheduler

Input: Request queue Q, batch size limit B

Function `SelectBatch(`Q`)`**:**
- $\mathcal{S} \leftarrow \varnothing$;
- **foreach** $r \in Q$ **do**
  - $(k_r, m_r) \leftarrow \text{extract_priority}(r)$;
- $Q_{sorted} \leftarrow \text{sort}(Q, \text{key}=(k_r, m_r))$;
- **foreach** $r \in Q_{sorted}$ **do**
  - **if** $|\mathcal{S}| < B$ **then**
    - $\mathcal{S} \leftarrow \mathcal{S} \cup \{r\}$;
- **return** $\mathcal{S}$;

While request scheduling has been extensively studied in online serving systems, it remains largely overlooked in training pipelines. In agentic RL, a single high-level task unfolds into a multi-turn interaction between the model and external tools. Each request re-enters the rollout component multiple times, and the system must maintain and reuse state across these turns (e.g., KV cache). If the scheduler ignores this structure and simply treats each turn as an independent request, it can easily waste GPU cycles by repeatedly recomputing KV cache and inflate end-to-end episode latency by letting short, late-stage turns wait behind many first turns.

To understand how multi-turn structure affects workload characteristics, we examine the response-length distribution across tool-call rounds in a ReAct-style agent, and observe two clear patterns: (1) the first response (round 0) is substantially longer than subsequent ones, and (2) average token generation decreases monotonically as the request approaches completion. In other words, early turns are long and compute-heavy, whereas later turns are short and relatively cheap.

These observations motivate a priority-based scheduling policy in which a request's priority is its tool-call count, i.e., the number of completed tool invocations so far. Intuitively, always giving higher priority to requests with more completed tool calls has two key benefits. First, it increases the likelihood of KV-cache reuse: when a new turn arrives shortly after the previous one, prioritizing that request over fresh, long first-turn prompts prevents the cached state from being evicted and avoids expensive re-prefilling. Second, it accelerates the completion of late-stage episodes, significantly reducing mean job completion time by flushing short, near-finished turns out of the system instead of waiting behind long initial turns.

Furthermore, while longer early turns are not a universal rule for all LLM interactions, they are a strong characteristic driven by task designs and agentic frameworks (e.g., ReAct [28], Plan-and-Solve [24]) where the first turn handles a complex prompt and generates a full plan, making it the most compute-intensive. Later turns are typically short observation-evaluation steps. Even for workloads where later turns are not shorter, the priority scheduler degrades gracefully to near-FIFO behavior while still preserving KV-cache reuse benefits.

Algorithm 2 implements this idea. For each request r in the queue, the scheduler extracts a priority key (k_r, m_r), where k_r is the number of completed tool calls (i.e., the current round index), and m_r is the length of the prefix of this request in the prefix-tree-based KV cache commonly used in LLM inference engines. The latter serves as a tie-breaker: among requests in the same round, those with longer prefixes are more likely to reuse more cache and thus have higher priority. The scheduler then sorts the queue by this key and selects the top-B requests to form the next batch. Compared to FIFO, this multi-turn–aware, tool-call-count-based policy explicitly exploits the response-length skew across rounds to improve both KV-cache efficiency and end-to-end agentic RL performance.

6 Implementation

DynaRL is implemented in 7K lines of Python code atop RLinf [22], an open-source RL framework that is flexible enough to support various resource partitioning modes (it supports both physical partitioning and time-sharing). Among them, 2K lines are for the Global Scheduler, 4K lines for

the migration manager and migration strategy implemented for different RL components (currently rollout, tool agent, trainer, inference, reward are supported), and an additional 1K lines for the HyperGraph extraction, data routing, and other auxiliary components.

HyperGraph Extraction. DynaRL automatically builds the HyperGraph from RLinf programs using lightweight static analysis plus minimal runtime hooks, without requiring manual annotations. On the static side, we parse the Python source of the training pipeline and components using the standard `ast` module that exposes the abstract syntax tree of the program. We identify component classes by their inheritance from RLinf's `WorkerGroup` base and registration APIs (e.g., rollout, trainer, tool agent), and instantiate a corresponding `HyperNode` for each logical component. We then traverse all call sites to the framework's inter-worker communication primitives (e.g., channel put/get). Each such call yields a `HyperEdge` between two `HyperNodes`, and we populate their `predecessors` and `successors` fields accordingly, forming the static skeleton of the hypergraph.

At runtime, WorkerGroup Managers register their concrete workers as `Node` entries under the corresponding `HyperNode`, and periodically update fields such as `allocated_resources` and `utilization` from monitoring signals. Similarly, the framework's communication hooks update the associated `HyperEdge.progress` and affinity metadata as dataflow events occur. This combination of Python AST analysis for structure and low-overhead runtime updates for state yields a continuously maintained HyperGraph that accurately reflects component interactions and is directly consumable by the global scheduler and data router.

Migration Strategy. DynaRL instantiates the three migration mechanisms in Figure 9 according to each component's statefulness and performance sensitivity. For stateless or weakly stateful components such as rollout, tool agents, inference, and reward models, we adopt `WorkloadMigration` to redistribute the request data upon resource migration.

For distributed trainers like Megatron-LM, DynaRL implements `p2pMigration` to transfer two kinds of state: model parameters and optimizer states. Given the old and new parallelism configurations (e.g., data-, tensor-, and pipeline-parallel degrees), the migration manager computes a rank-mapping and resharding plan that determines which source ranks send which parameter and optimizer shards to which destination ranks. The new trainer processes are launched on the target GPUs and participate in a series of peer-to-peer transfers following this plan until all state is materialized under the new sharding. Afterward, DynaRL reconstructs all communication groups (e.g., NCCL process groups for data/tensor/pipeline parallelism) to match the new topology and only then tears down the old ranks. This design allows large, stateful training jobs to change their parallelism layout with sub-second to few-second pauses, making frequent elastic rescaling practical in multi-component RL pipelines.

All remaining components fall back to `RebootMigration`, which simply shuts down workers on the old resources and relaunches them on the new ones with reloaded states.

7 Evaluation

We evaluate DynaRL on comprehensive workloads including math-reasoning RL and multi-turn agentic RL with LLMs of different sizes ranging from 1.5B to 32B on real-world datasets. The experiment results are summarized below:

- For math-reasoning RL, DynaRL consistently outperforms state-of-the-art RL systems (e.g., verl, RLinf) by 1.27×–1.98× on a variety of math-reasoning RL training configurations under different cluster scales.
- For agentic RL, DynaRL achieves 1.06×–1.53× higher throughput than RLinf. With priority-aware request scheduling for multi-turn interactions, DynaRL further improves performance to 1.27×–1.64×.
- The scheduling plan can be generated within 200 ms, and DynaRL applies each reallocation plan within 0.5–5 seconds, incurring less than 1% overhead.

7.1 Experiment Setup

Testbed. We deploy DynaRL on an H100 cluster with 16 nodes and 128 GPUs. Each node has 8 NVIDIA H100-80GB GPUs, and 2 Intel Xeon Platinum 8558@2.1 GHz CPUs with 48 cores and 2TB memory. Intra-node communication utilizes NVLink, while inter-node communication uses 8 Mellanox ConnectX-7 RDMA NICs per node, each providing 400 Gbps bandwidth with RoCEv2.

Baselines. For math-reasoning RL, we compare DynaRL against verl [19] v0.5 and RLinf [22], the state-of-the-art open-source RLHF systems. We choose SGLang [32] as their rollout engine and Megatron-LM [20] as their training backend. To further demonstrate the broader scheduling space of DynaRL, we also compare it against RLHFuse [33], which performs fusion in rollout and inference phases. Since RLHFuse lacks a publicly available artifact, we implemented its core fusion logic within RLinf using the same rollout engine and training backend to ensure a fair comparison. For multi-turn agentic LLM, we compare with RLinf, as verl only supports FSDP [31] for this experiment, which is much slower than the Megatron-LM backend.

Metrics. For the end-to-end experiments, we use RLHF throughput (tokens/sec) as the performance metric following verl [19], which is defined as dividing the total number of tokens in prompts and responses in a global batch by one RLHF iteration time. For multi-turn agentic RL, we use number of processed requests per second (abbreviated as requests per

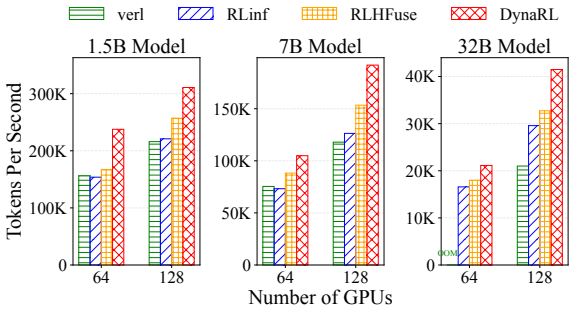


Figure 11: End-to-end throughput of math reasoning RL under different cluster scale and model size settings.

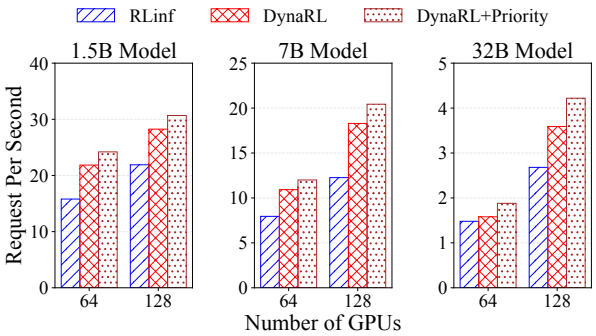


Figure 12: End-to-end throughput of multi-turn agentic RL under different cluster scale and model size settings.

second). All reported performance numbers are averaged over 10 training iterations after warm-up.

Datasets. For math-reasoning RL, we use the AReaL-boba-Data dataset [26]. This dataset integrates multiple standard datasets including DeepScaleR [12], Open-Reasoner-Zero [9], Light-R1 [25], DAPO [30], NuminaMath [14] (AoPS/Olympiad subsets), and ZebraLogic [11]. Overly simple problems are filtered out to ensure dataset quality and effectiveness. For multi-turn agentic RL, we use the rstar2-agent dataset [18], which includes tasks from the DAPO training set, problems from the AoPS forums via OpenMathReasoning [13] and Project Euler [3].

Configuration. For both math-reasoning and multi-turn agentic RL experiments, we use a rollout batch size of 512, group size of 16, and sequence length of 28,672. The trainer tensor-parallel (TP) sizes are 2, 4, and 8 for the 1.5B, 7B, and 32B models, respectively; the corresponding rollout TP sizes are 1, 2 and 4. These configurations remain unchanged as we scale to larger GPU counts.

7.2 End-to-End Experiments

Math-reasoning RL. Math-reasoning RL comprises three stages, i.e., rollout, inference, and training. Existing systems such as verl and RLinf follow a static, sequential execution pattern, dedicating all resources to one stage at a time. Figure 11 compares throughput under baseline's static scheduling and DynaRL 's dynamic scheduling across three model sizes (1.5B, 7B, and 32B) and two GPU scales (64 and 128). As shown, DynaRL consistently improves throughput, with larger gains on larger models.

For the 64 GPU experiments, verl and RLinf exhibit similar performance for 1.5B and 7B models, while DynaRL yields 1.43×–1.55× speedups. Because with static scheduling, rollout eventually runs at low utilization, whereas DynaRL detects overprovisioned components and reallocates their unused capacity to other components. For the 32B model, verl fails due to OOM, and DynaRL achieves a moderate 1.27× speedup over RLinf. The speedup is smaller due to a more constrained scheduling space: with TP size 8, train parallelism can scale when at least 8 idle GPUs are available.

With 128 GPUs, improvements for 1.5B and 7B models stabilize around 1.40×–1.52×. The 32B model, however, benefits significantly more, reaching 1.98× over verl and 1.40× over RLinf, as the larger cluster provides greater flexibility and a wider scheduling space.

Compared to RLHFuse, DynaRL achieves 1.21×–1.42× higher throughput. RLHFuse fuses only the rollout and inference stages, but in practice inference is typically much less time-consuming than rollout, especially at long response lengths, leaving training resources idle as it cannot be fused with other stages. In contrast, DynaRL dynamically schedules resources across all three stages, rollout, inference, and training, jointly, enabling it to better handle diverse components with heterogeneous execution characteristics.

Multi-turn agentic RL. Figure 12 presents the results for multi-turn agentic RL. We compare RLinf with two variants of DynaRL: dynamic allocation alone and dynamic allocation combined with the priority-aware request scheduling policy.

On 64 GPUs, dynamic allocation alone yields 1.06×–1.38× improvement, smaller than in single-turn math reasoning. This highlights the additional scheduling complexity introduced by multi-turn interactions, where rollout becomes more unpredictable. With priority-aware request scheduling, DynaRL achieves 1.51×–1.53× speedups on the 1.5B and 7B models over RLinf. The gains on the 32B model are also smaller (1.27×) due to constrained scheduling space.

When scaling from 64 to 128 GPUs, DynaRL with priority-aware scheduling is 1.40×, 1.64×, 1.58× faster than RLinf on 1.5B, 7B, 32B models, respectively. The benefit of priority-aware scheduling diminishes slightly for the 1.5B and 7B models (from 15% to 11% for 1.5B, and from 14% to 11% for 7B). In contrast, the 32B model experiences increased gains (from 21% to 24%). This divergence stems from KV-cache sufficiency. That is, on 128 GPUs, smaller models can maintain more request prefixes in cache without eviction, reducing the marginal benefit of request-level prioritization.

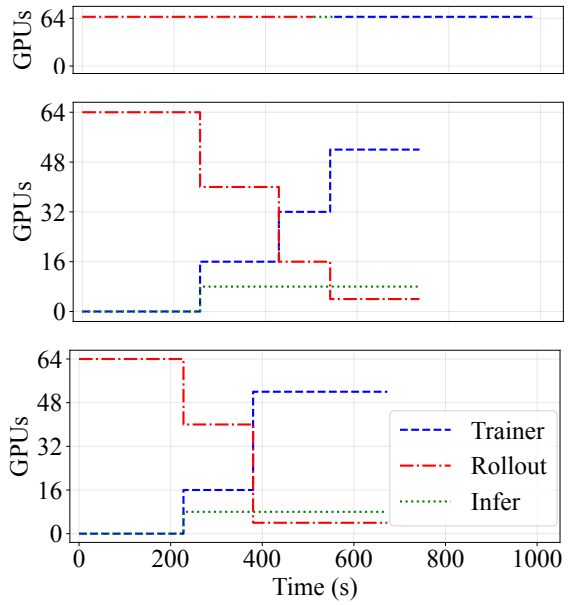


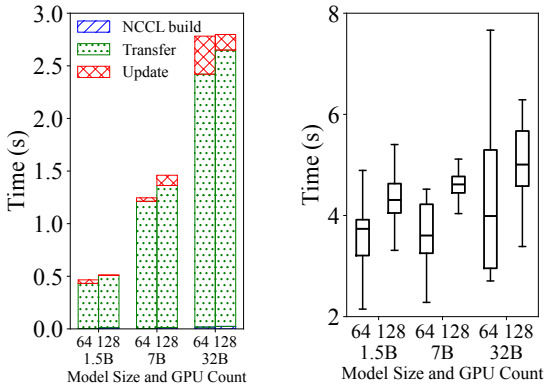
Figure 13: GPU allocation timelines for the three execution modes. From top to bottom: (1) static mode, where all GPUs are assigned to one component at a time; (2) dynamic mode without priority-aware request scheduling, which reallocates resources reactively based on component utilization; and (3) dynamic mode with priority-aware request scheduling, which further prioritizes requests with more completed tool calls. Each plot shows how Trainer, Rollout, and Inference components acquire and release GPUs over time.

For the 32B model, however, KV-cache pressure remains high, and priority-aware scheduling continues to play a significant role, yielding larger improvements.

7.3 Breakdown of Online Scheduling

Figure 13 illustrates how DynaRL allocates 64 GPUs across rollout, inference, and trainer in the 7B math-reasoning RL experiment. In the top panel, the static configuration fully serializes the three phases, assigning the entire cluster to one component at a time.

The middle panel shows how dynamic allocation reshapes the execution timeline by reallocating GPUs at runtime. At the beginning of the iteration, neither inference nor trainer receives resources because no corresponding requests are yet available. As rollout proceeds, its effective parallelism naturally declines as fewer requests remain. The scheduler detects this underutilization and gradually reallocates GPUs to inference and trainer. This overlap shortens the critical path, i.e., downstream stages start earlier and run concurrently with the tail of rollout, eliminating large idle windows and exposing intra-iteration parallelism that static allocation cannot



(a) Trainer. (b) Rollout.

Figure 14: Trainer and rollout migration cost.

exploit. Inference is consistently assigned fewer GPUs than trainer because it performs only forward passes and maintains no external state (e.g., gradients or optimizer slots), thus requiring substantially less compute for the same number of tokens. Since inference scales adequately with a small GPU budget, DynaRL assigns it a stable initial allocation to avoid unnecessary reallocation overhead.

The bottom panel shows that incorporating priority-aware request scheduling further accelerates resource transitions. By prioritizing rollout requests with more completed tool calls, the system reduces straggler effects and frees GPUs earlier. Consequently, both inference and trainer reach their target allocations sooner and at larger scale. For example, trainer scales to 32 GPUs around 420s under vanilla dynamic allocation, but reaches 52 GPUs by roughly 380s with priority-aware scheduling. Toward the end of the iteration, rollout and inference operate with minimal resources to process long-tail requests, while trainer receives most of the cluster to accelerate training. This demonstrates that fine-grained, request-level scheduling is essential for fully exploiting temporal variability in RL workloads.

7.4 Overhead Analysis

Trainer migration overhead. Figure 14a reports the trainer migration cost across different model sizes. The total migration time grows primarily with model scale, increasing from sub-millisecond ranges for 1.5B to several seconds for 32B, yet still contributes less than 0.5% to end-to-end latency even for the 32B model on 128 GPUs. In contrast, the GPU count has negligible impact. This behavior reflects the design of our trainer migration mechanism, i.e., model states are transferred through fully distributed peer-to-peer communication, allowing the migration cost to mainly scale with the size of the model. As a result, trainer replicas can be rebalanced even

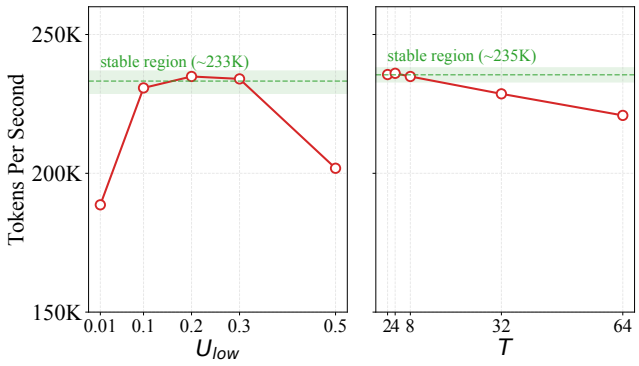


Figure 15: Throughput stationarity of DynaRL under varying scheduling parameters U_{low} (left) and T (right). Shaded bands mark the stable operating regions.

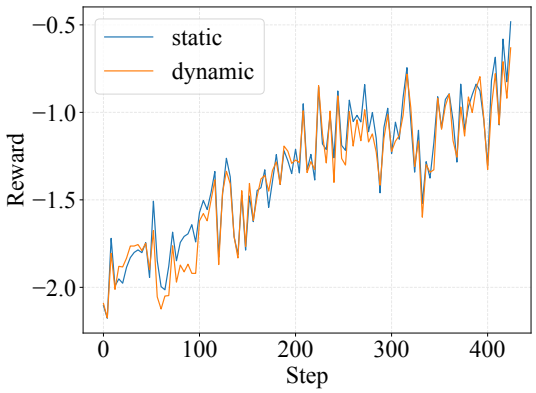


Figure 16: Reward curve of static placement and dynamic scheduling.

on large clusters without incurring significant coordination or communication group rebuild overheads.

Rollout migration overhead. Figure 14b shows rollout migration costs across different model and cluster sizes, averaged over 10 training iterations. Unlike train migration, rollout migration exhibits sensitivity to both model scale and GPU count, as larger rollout groups require more workers to synchronize and transfer running states. Nonetheless, the overall variance remains modest. Migration consistently completes within a few seconds even for the 32B model on 128 GPUs. This small and predictable overhead ensures that migration decisions triggered by dynamic scheduling do not dominate the end-to-end training time. The results confirm that DynaRL 's reallocation mechanism maintains low control-plane overheads and can support frequent, fine-grained resource adjustments during RL training.

Scheduling overhead. We measure the scheduling time during both the reasoning and agentic RL tasks. We find that all scheduling decisions can be made within 200 milliseconds, which is negligible compared to the end-to-end training time. During a training iteration, the total overhead of the scheduling is below 0.5%.

7.5 Parameter Sensitivity

We study the performance sensitivity of DynaRL with respect to its scheduling parameters: the rebalancing threshold U_{low}, which controls how aggressively work is redistributed across stages, and the scheduling window T, which controls how frequently the scheduler reacts to load imbalance. We sweep $U_{low} \in \{0.01, 0.1, 0.2, 0.3, 0.5\}$ with T fixed, and $T \in \{2, 4, 8, 32, 64\}$ with U_{low} fixed, and measure end-to-end token throughput on the 1.5B math-reasoning workload.

Figure 15 shows that DynaRL is largely insensitive to both parameters across more than an order of magnitude. For U_{low}, throughput remains within $\sim$1.5% of its peak (233K tokens/s) over the whole interval $U_{low} \in [0.1, 0.3]$. Performance only degrades at the extremes: an overly small $U_{low} = 0.01$ triggers excessive rebalancing and drops throughput to 188.65K tokens/s ($\downarrow$ 19.7%), while an overly large $U_{low} = 0.5$ suppresses useful rebalancing and yields 201.82K tokens/s ($\downarrow$ 14.1%). For T, throughput stays within 0.5% of peak over $T \in [2, 8]$ (mean 235.5K tokens/s) and degrades only mildly at $T = 32$ ($\downarrow$ 3.1%) and $T = 64$ ($\downarrow$ 6.4%), where the scheduler reacts too slowly to transient imbalance. These results indicate that the stable operating region for DynaRL spans a wide plateau in both dimensions, so practitioners can pick parameters from a broad range without precise per-workload tuning.

7.6 Correctness Verification

Guaranteeing algorithm convergence is a fundamental requirement of DynaRL. To achieve this, all supported migrations strictly preserve the underlying RL semantics by leveraging the interruptible points defined in Section 4.2. During the rollout and inference phases, migrations are executed at the query level; this alters the physical GPU assignment but leaves the execution logic unaffected. During training phases, migrations meticulously preserve model parameters and optimizer states while holding RL hyperparameters (e.g., micro-batch and global batch sizes) constant. For instance, when adjusting the number of GPUs under data parallelism, DynaRL reconfigures how micro-batches are distributed across devices without altering the global batch size. This ensures that the training logic remains intact, providing strict mathematical consistency throughout the migration process.

To empirically validate this consistency, Figure 16 plots the step-wise reward curves for a 1.5B-parameter model on 64 GPUs on a mathematical reasoning task, corresponding to the experiment in Figure 11. As illustrated, DynaRL's dynamic scheduling closely tracks the convergence trajectory of the static placement baseline. This confirms that our dynamic migrations introduce no adverse effects on the RL learning process.

8 Related Works

Reinforcement Learning Frameworks. Existing RL training frameworks for large-scale alignment generally follow static system designs, which can be categorized into task-colocated (e.g., verl [19] and DeepSpeed-Chat [29]) and task-separated (e.g., Slime [34] and OpenRLHF [8]) execution modes. More recently, frameworks with automated scheduling have been introduced. RLHFuse [33], for instance, performs fusion in rollout and inference phases, effectively mitigating long-tail generation bottlenecks and improving end-to-end throughput. Some recent works use async RL algorithms and co-design the infrastructure to alleviate the long-tail issue. AReal [4] introduces an asynchronous model update algorithm for task-separated systems to increase training throughput. Kimi-k1.5 [21] introduces partial rollouts to optimize the handling of complex reasoning trajectories. In contrast to these static or partially optimized designs, DynaRL enables joint dynamic scheduling across rollout, inference, and training phases, allowing the runtime to switch between colocated and separated execution modes based on live workload patterns. Furthermore, while prior systems focus primarily on single-turn reasoning RL pipelines, DynaRL extends dynamic scheduling to agentic RL, where multi-turn environment interaction worsens the impact of long-tail in rollout.

Cluster-Level Deep Learning Schedulers. Beyond RL-specific frameworks, a rich line of systems work has explored dynamic resource management for deep learning clusters. Optimus [16], Gandiva [27], and Tiresias [5] perform introspective GPU scheduling across many concurrent training jobs, adjusting placement and GPU allocations based on job-level progress signals. Traditional cluster managers such as Borg [23], Kubernetes, and their autoscaling extensions similarly treat each training workload as a black box when reallocating resources. These systems operate primarily at the job or container granularity and lack visibility into the internal structure of RL pipelines. In contrast, DynaRL performs fine-grained, intra-job scheduling across rollout, inference, and training components using a dynamic hypergraph abstraction, enabling resource rebalancing that is aware of cross-component dependencies and multi-turn interaction structure inside a single RL workload.

9 Conclusion

Modern RL workloads for reasoning and agentic systems exhibit substantial dynamicity, making static resource allocations highly inefficient. This paper presented DynaRL, the first RL system that makes dynamic scheduling a first-class capability. DynaRL introduces a unified runtime and abstraction that can observe how different RL components behave over time and automatically shift resources to where they are most needed. We demonstrate that DynaRL consistently improves end-to-end throughput while maintaining low online scheduling overhead across diverse RL workloads. In the future, we envision that dynamic resource management will become an essential component of RL systems, enabling more efficient and scalable training through complex interactions.

Acknowledgments

We thank our anonymous reviewers and shepherd for their constructive feedback. Zhi Yang and Yu Wang are the corresponding authors. Quanlu Zhang is the project lead. We also express our sincere gratitude to the RLinf open-source community for their continuous support, feedback, and ecosystem contributions. This work was partially supported by National Natural Science Foundation of China under Grant No.92464301, 62406159, 62325405, Zhongguancun Academy Grant No. C20250301, Shenzhen Science and Technology Program No. AI2026016.

References

[1] Lukas Beckenbauer, Johannes-Lucas Loewe, Ge Zheng, and Alexandra Brintrup. Orchestrator: Active inference for multi-agent systems in long-horizon tasks. arXiv preprint arXiv:2509.05651, 2025.

[2] DeepSeek-AI. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning, 2025.

[3] Project Euler. Project euler is a series of challenging mathematical/computer programming problems that will require more than just mathematical insights to solve. https://projecteuler.net/, 2025.

[4] Wei Fu, Jiaxuan Gao, Xujie Shen, Chen Zhu, Zhiyu Mei, Chuyi He, Shusheng Xu, Guo Wei, Jun Mei, Jiashu Wang, et al. Areal: A large-scale asynchronous reinforcement learning system for language reasoning. arXiv preprint arXiv:2505.24298, 2025.

[5] Juncheng Gu, Mosharaf Chowdhury, Kang G. Shin, Yibo Zhu, Myeongjae Jeon, Junjie Qian, Hongqiang Liu, and Chuanxiong Guo. Tiresias: A GPU cluster manager for distributed deep learning. In 16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19), pages 485–500, Boston, MA, February 2019. USENIX Association.

[6] Openai Gym. Gym is an open source python library for developing and comparing reinforcement learning algorithms. https://github.com/openai/gym, 2025.

[7] Gymnasium. An api standard for reinforcement learning with a diverse collection of reference environments. https://gymnasium.farama.org/, 2025.

[8] Jian Hu, Xibin Wu, Zilin Zhu, Weixun Wang, Dehao Zhang, Yu Cao, et al. Openrlhf: An easy-to-use, scalable and high-performance rlhf framework. arXiv preprint arXiv:2405.11143, 2024.

[9] Jingcheng Hu, Yinmin Zhang, Qi Han, Daxin Jiang, Xiangyu Zhang, and Heung-Yeung Shum. Open-reasoner-zero: An open source approach to scaling up reinforcement learning on the base model, 2025.

[10] Lean. Lean is an open-source programming language and proof assistant that enables correct, maintainable, and formally verified code. https://lean-lang.org/, 2025.

[11] Bill Yuchen Lin, Ronan Le Bras, Kyle Richardson, Ashish Sabharwal, Radha Poovendran, Peter Clark, and Yejin Choi. Zebralogic: On the scaling limits of llms for logical reasoning, 2025.

[12] Michael Luo, Sijun Tan, Justin Wong, Xiaoxiang Shi, William Y. Tang, Manan Roongta, Colin Cai, Jeffrey Luo, Li Erran Li, Raluca Ada Popa, and Ion Stoica. Deepscaler: Surpassing o1-preview with a 1.5b model by scaling rl, 2025. Notion Blog.

[13] Ivan Moshkov, Darragh Hanley, Ivan Sorokin, Shubham Toshniwal, Christof Henkel, Benedikt Schifferer, Wei Du, and Igor Gitman. Aimo-2 winning solution: Building state-of-the-art mathematical reasoning models with openmathreasoning dataset, 2025.

[14] Project Numina. Project euler is a series of challenging mathematical/computer programming problems that will require more than just mathematical insights to solve. https://huggingface.co/collections/AI-MO/numinamath, 2025.

[15] Long Ouyang, Jeffrey Wu, Xu Jiang, Diogo Almeida, Carroll Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, Alex Ray, John Schulman, Jacob Hilton, Fraser Kelton, Luke Miller, Maddie Simens, Amanda Askell, Peter Welinder, Paul F Christiano, Jan Leike, and Ryan Lowe. Training language models to follow instructions with human feedback. In S. Koyejo, S. Mohamed, A. Agarwal, D. Belgrave, K. Cho, and A. Oh, editors, Advances in Neural Information Processing Systems, volume 35, pages 27730–27744. Curran Associates, Inc., 2022.

[16] Yanghua Peng, Yixin Bao, Yangrui Chen, Chuan Wu, and Chuanxiong Guo. Optimus: an efficient dynamic resource scheduler for deep learning clusters. In Proceedings of the Thirteenth EuroSys Conference, EuroSys '18, New York, NY, USA, 2018. Association for Computing Machinery.

[17] Alec Radford, Karthik Narasimhan, Tim Salimans, Ilya Sutskever, et al. Improving language understanding by generative pre-training. 2018.

[18] Ning Shang, Yifei Liu, Yi Zhu, Li Lyna Zhang, Weijiang Xu, Xinyu Guan, Buze Zhang, Bingcheng Dong, Xudong Zhou, Bowen Zhang, Ying Xin, Ziming Miao, Scarlett Li, Fan Yang, and Mao Yang. rstar2-agent: Agentic reasoning technical report, 2025.

[19] Guangming Sheng, Chi Zhang, Zilingfeng Ye, Xibin Wu, Wang Zhang, Ru Zhang, Yanghua Peng, Haibin Lin, and Chuan Wu. Hybridflow: A flexible and efficient rlhf framework. In Proceedings of the Twentieth European Conference on Computer Systems, pages 1279–1297, 2025.

[20] Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. Megatron-lm: Training multi-billion parameter language models using model parallelism. arXiv preprint arXiv:1909.08053, 2019.

[21] Kimi Team, Angang Du, Bofei Gao, Bowei Xing, Changjiu Jiang, Cheng Chen, Cheng Li, Chenjun Xiao, Chenzhuang Du, Chonghua Liao, Chuning Tang, Congcong Wang, Dehao Zhang, Enming Yuan, Enzhe Lu, Fengxiang Tang, Flood Sung, Guangda Wei, Guokun Lai, Haiqing Guo, Han Zhu, Hao Ding, Hao Hu, Hao Yang, Hao Zhang, Haotian Yao, Haotian Zhao, Haoyu Lu, Haoze Li, Haozhen Yu, Hongcheng Gao, Huabin Zheng, Huan Yuan, Jia Chen, Jianhang Guo, Jianlin Su, Jianzhou Wang, Jie Zhao, Jin Zhang, Jingyuan Liu, Junjie Yan, Junyan Wu, Lidong Shi, Ling Ye, Longhui Yu, Mengnan Dong, Neo Zhang, Ningchen Ma, Qiwei Pan, Qucheng Gong, Shaowei Liu, Shengling Ma, Shupeng Wei, Sihan Cao, Siying Huang, Tao Jiang, Weihao Gao, Weimin Xiong, Weiran He, Weixiao Huang, Weixin Xu, Wenhao Wu, Wenyang He, Xianghui Wei, Xianqing Jia, Xingzhe Wu, Xinran Xu, Xinxing Zu, Xinyu Zhou, Xuehai Pan, Y. Charles, Yang Li, Yangyang Hu, Yangyang Liu, Yanru Chen, Yejie Wang, Yibo Liu, Yidao Qin, Yifeng Liu, Ying Yang, Yiping Bao, Yulun Du, Yuxin Wu, Yuzhi Wang, Zaida Zhou, Zhaoji Wang, Zhaowei Li, Zhen Zhu, Zheng Zhang, Zhexu Wang, Zhilin Yang, Zhiqi Huang, Zihao Huang, Ziyao Xu, Zonghan Yang, and Zongyu Lin. Kimi k1.5: Scaling reinforcement learning with llms, 2025.

[22] RLinf Team. Rlinf is a flexible and scalable open-source infrastructure designed for post-training foundation models (llms, vlms, vlas) via reinforcement learning. https://github.com/RLinf/RLinf, 2025.

[23] Abhishek Verma, Luis Pedrosa, Madhukar Korupolu, David Oppenheimer, Eric Tune, and John Wilkes. Large-scale cluster management at google with borg. In

Proceedings of the Tenth European Conference on Computer Systems, EuroSys '15, New York, NY, USA, 2015. Association for Computing Machinery.

[24] Lei Wang, Wanyu Xu, Yihuai Lan, Zhiqiang Hu, Yunshi Lan, Roy Ka-Wei Lee, and Ee-Peng Lim. Plan-and-solve prompting: Improving zero-shot chain-of-thought reasoning by large language models. In Anna Rogers, Jordan Boyd-Graber, and Naoaki Okazaki, editors, Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), pages 2609–2634, Toronto, Canada, July 2023. Association for Computational Linguistics.

[25] Liang Wen, Yunke Cai, Fenrui Xiao, Xin He, Qi An, Zhenyu Duan, Yimin Du, Junchen Liu, Lifu Tang, Xiaowei Lv, Haosheng Zou, Yongchao Deng, Shousheng Jia, and Xiangzheng Zhang. Light-r1: Curriculum sft, dpo and rl for long cot from scratch and beyond, 2025.

[26] Huggingface xDAN-datasets (xDAN Back). xdan-datasets/areal-boba-data. https://huggingface.co/datasets/xDAN-datasets/AReaL-boba-Data, 2025.

[27] Wencong Xiao, Romil Bhardwaj, Ramachandran Ramjee, Muthian Sivathanu, Nipun Kwatra, Zhenhua Han, Pratyush Patel, Xuan Peng, Hanyu Zhao, Quanlu Zhang, Fan Yang, and Lidong Zhou. Gandiva: Introspective cluster scheduling for deep learning. In 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18), pages 595–610, Carlsbad, CA, October 2018. USENIX Association.

[28] Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. React: Synergizing reasoning and acting in language models, 2023.

[29] Zhewei Yao, Reza Yazdani Aminabadi, Olatunji Ruwase, Samyam Rajbhandari, Xiaoxia Wu, Ammar Ahmad Awan, Jeff Rasley, Minjia Zhang, Conglong Li, Connor Holmes, et al. Deepspeed-chat: Easy, fast and affordable rlhf training of chatgpt-like models at all scales. arXiv preprint arXiv:2308.01320, 2023.

[30] Qiying Yu, Zheng Zhang, Ruofei Zhu, Yufeng Yuan, Xiaochen Zuo, Yu Yue, Weinan Dai, Tiantian Fan, Gaohong Liu, Lingjun Liu, et al. Dapo: An open-source llm reinforcement learning system at scale. arXiv preprint arXiv:2503.14476, 2025.

[31] Yanli Zhao, Andrew Gu, Rohan Varma, Liang Luo, Chien-Chin Huang, Min Xu, Less Wright, Hamid Shojanazeri, Myle Ott, Sam Shleifer, et al. Pytorch fsdp: experiences on scaling fully sharded data parallel. arXiv preprint arXiv:2304.11277, 2023.

[32] Lianmin Zheng, Liangsheng Yin, Zhiqiang Xie, Chuyue Sun, Jeff Huang, Cody Hao Yu, Shiyi Cao, Christos Kozyrakis, Ion Stoica, Joseph E. Gonzalez, Clark Barrett, and Ying Sheng. Sglang: Efficient execution of structured language model programs, 2024.

[33] Yinmin Zhong, Zili Zhang, Bingyang Wu, Shengyu Liu, Yukun Chen, Changyi Wan, Hanpeng Hu, Lei Xia, Ranchen Ming, Yibo Zhu, and Xin Jin. Optimizing rlhf training for large language models with stage fusion. In Proceedings of the 22nd USENIX Symposium on Networked Systems Design and Implementation, NSDI '25, USA, 2025. USENIX Association.

[34] Zilin Zhu, Chengxing Xie, Xin Lv, and slime Contributors. slime: An llm post-training framework for rl scaling. https://github.com/THUDM/slime, 2025. GitHub repository. Corresponding author: Xin Lv.

ROLLART: Disaggregated Multi-Task Agentic RL Training at Scale

Wei Gao[†*], Yuheng Zhao[†*], Tianyuan Wu[†*], Shaopan Xiong[‡*], Weixun Wang[‡*], Dakai An[†], Lunxi Cao[†], Dilxat Muhtar[‡], Zichen Liu[‡], Haizhou Zhao[‡], Ju Huang[‡], Siran Yang[‡], Yongbin Li[¶], Wenbo Su[‡], Jiamang Wang[‡], Lin Qu[‡], Bo Zheng[‡], Wei Wang[†]

[†]HKUST [‡]Alibaba Group [¶]Tongyi Lab, Alibaba

Abstract

Agentic Reinforcement Learning (RL) trains LLMs through multi-turn interactions with environments, producing workloads that mix compute-bound prefill, bandwidth-bound decoding, CPU-heavy environment execution, and bursty reward evaluation. Existing systems either colocate all stages on a single GPU cluster or decouple them only at a coarse granularity, overlooking hardware heterogeneity and incurring substantial synchronization overhead across stages.

We present ROLLART, a system for multi-task agentic RL on disaggregated infrastructure. ROLLART maps each pipeline stage to best-fit hardware, routing prefill-heavy tasks to compute-optimized GPUs, decode-heavy tasks to bandwidth-optimized GPUs, and environments to CPU clusters. It decouples rollout at the trajectory level, allowing generation, environment interaction, and reward scoring to proceed independently, so that slow or failed environments never block the others. ROLLART offloads stateless reward computation to serverless infrastructure and overlaps rollout with training via staleness-bounded asynchronous weight synchronization. Our results demonstrate that ROLLART effectively improves training throughput and achieves 1.31–2.05× training time reduction compared to various RL systems. We also evaluated ROLLART by training a hundreds-of-billions-parameter MoE model for Qoder product on an Alibaba cluster with above 3,000 GPUs, demonstrating its stability and scalability.

1 Introduction

Reinforcement Learning (RL) is advancing Large Language Models (LLMs) beyond single-turn reasoning toward autonomous, long-horizon decision-making [4, 7, 36]. In this paradigm, known as *agentic RL*, an LLM interacts with external environments for tool use [16, 58], web navigation [22, 24, 50], and computer control [28, 30, 33] over multiple turns, learning to solve tasks through trial and error.

The agentic RL training pipeline operates as an iterative cycle of three stages: *rollout*, *reward*, and *training*. Unlike standard RL post-training with single-turn responses, during rollout, the agent generates actions and receives environment observations over multiple turns until a termination condition is met, producing a *long, multi-turn trajectory*. Each trajectory is then evaluated in the reward stage, using rule-based scripts [13, 19], code sandboxes, or separate LLM judges [49, 68] to assign scalar reward signals. The training stage consumes scored trajectories to update the agent's weights, which are synchronized back to the rollout workers for the next iteration.

As research labs scale agentic RL to larger models and harder tasks [8, 54], a fundamental resource-heterogeneity problem emerges *across and within* these stages. Within rollout alone, LLM generation alternates between compute-bound prefill and bandwidth-bound decoding whose ratio depends on the task: on cost-equivalent hardware, compute-optimized H800 GPUs cut prefill-heavy rollout time to 0.53× that of bandwidth-optimized H20 GPUs, while H20s cut decode-heavy rollout time to 0.49×–0.79× of H800s (§3). Concurrently, environments are *stateful*, *CPU-bound* processes whose latency is heavy-tailed due to host contention, large variance in interaction turns, and environment failures. Reward workers are *stateless* and exhibit persistently low utilization—dropping to as little as 7.4% on dedicated GPUs—yet require elastic scaling when trajectories complete. Finally, the training stage demands high-end GPUs with fast interconnects. No single hardware type satisfies all stages.

A natural solution is to *disaggregate* the pipeline and route each stage to best-fit hardware. Yet, existing systems fall short. *Monolithic* frameworks such as veRL [46, 47], slime [70], and rLLM [53] co-locate all stages on a single GPU cluster and ignore heterogeneity entirely. *Partially disaggregated* systems such as AWorld [63] and DeepSWE [31] offload environments to Kubernetes [55] but still colocate the resource-heavy rollout and training stages. Even state-of-the-art asynchronous systems address only a subset of these heterogeneity dimensions: StreamRL [67], AsyncFlow [15], and SeamlessFlow [56] split training from rollout but treat rollout as a monolith, overlooking prefill/decode divergence *within* a task. AReaL [10] introduces unbounded-staleness asynchrony between training and generation, yet still batches environment interaction and couples reward to generation GPUs. Laminar [45] schedules trajectories individually but does not exploit within-rollout hardware affinity or offload stateless reward. As a result, no existing system jointly addresses prefill/decode heterogeneity

[*]Wei Gao, Yuheng Zhao, Tianyuan Wu, Shaopan Xiong, and Weixun Wang contributed equally to this work.

within generation, long-tail environment stragglers, stateless reward underutilization, and cross-cluster bandwidth variability at production scale (§3).

To address these challenges, we present ROLLART, a distributed system that maximizes throughput for multi-task agentic RL on disaggregated infrastructure. The design of ROLLART is driven by four requirements derived from empirical workload characterization (§3):

- **R1 Hardware-affinity workload mapping:** each pipeline stage, as well as sub-stages within rollout, should be bound to best-fit hardware so that compute-bound workloads land on compute-optimized GPUs and bandwidth-bound workloads on bandwidth-optimized GPUs. Because prefill/decode, CPU-bound, and stateless characteristics are stable per task domain in production workloads (§3 and §8), this binding can be driven by lightweight, domain-level annotations.
- **R2 Trajectory-level asynchronous rollout:** environment interaction, LLM generation, and reward computation should be decoupled at the granularity of trajectories, so that environment stragglers never stall the pipeline.
- **R3 Serverless offloading:** Stateless components, notably reward computation, should be offloaded to serverless infrastructure, eliminating dedicated hot standby GPUs while gaining autoscaling and fault tolerance.
- **R4 Bounded-staleness asynchronous training:** training and rollout should execute concurrently on separate GPUs, overlapping weight synchronization with rollout. An asynchronous bound eliminates resource bubbles while preserving performance.

To meet these requirements, our system combines a declarative programming model with a heterogeneity-aware distributed runtime. At job submission, users declare hardware preferences for each stage through Python decorators. The runtime binds each stage to its best-fit resource pool (**R1**): compute-optimized GPUs for training and prefill-heavy generation tasks, bandwidth-optimized GPUs for decode-heavy generation tasks, CPU clusters for environments, and serverless endpoints for reward. During rollout, the runtime drives each trajectory independently through generation and environment interaction. A slow or failed environment never blocks other trajectories, and reward scoring begins as soon as any trajectory completes (**R2**). Completed trajectories are scored by stateless reward functions on serverless infrastructure, granting elastic autoscaling without dedicated GPU reservations (**R3**). While rollout continues, a separate GPU cluster consumes scored trajectories for training. A bounded-staleness weight synchronization protocol propagates updated weights back to the rollout stage, overlapping training and cross-cluster transfer with ongoing generation to eliminate resource bubbles (**R4**). Finally, cross-cutting optimizations (§6.3) reduce cross-stage data-movement cost, add resilience to environment failures, and support optional prefill-decoding disaggregation within rollout.

Table 1: Taxonomy of Adopted Agentic Environments.

Environment	Task Domain	Modality	#Turns
SWE-bench [23]	SWE	Text	30–50
WebShop [61]	Web	Text	5–30
FrozenLake [9]	Game	Text, Visual	20–100
GEM-math [3]	Math+Tool Use	Text	< 5
GEM-game [3]	Game	Text	1

We have implemented ROLLART in approximately 60k lines of Python code and evaluated it by training Qwen3 models (8B–32B) [2] on a diverse mixture of agentic tasks across disaggregated H800 and H20 GPU clusters. ROLLART achieves 2.05×, 1.35×, and 1.31×, step-time reductions over a strengthened monolithic synchronous baseline, one-off baseline, and AReaL [10], respectively, and delivers 2.65–4.58× throughput over the synchronous baseline. Offloading reward to a serverless cloud raises reward-GPU utilization from 6% to 88% and halves per-step rollout time. We further quantify the overheads introduced by disaggregation, including cross-cluster weight synchronization via Mooncake [38], serverless reward I/O, and trajectory transfer, and show that they are outweighed by the resulting throughput gains (§7). To validate scalability and stability in a production setting, we have deployed ROLLART to train a hundreds-of-billions-parameter Mixture-of-Experts (MoE) model for Qoder [39] product on a cluster of over 3,000 GPUs. This deployment runs continuously for one week, confirming ROLLART's ability to sustain high throughput and robust fault tolerance at scale.

2 Background

2.1 Agentic RL Training

The Training Pipeline. Multi-task agentic RL training follows an iterative loop with three stages. The first stage, **rollout**, collects experience: an agent LLM (actor) interacts with parallel *environments* to generate training data. Unlike standard LLM inference, rollout is *multi-turn* and *stateful* [8, 54]. At each turn, the agent observes a state, emits an action token sequence, and submits it to the environment. The environment executes the action (e.g., running code or clicking a link) and returns feedback. The loop repeats until termination, producing a sequence of state-action pairs, i.e., a *trajectory*. After rollout, the **reward stage** evaluates trajectory quality via a *reward worker*, producing a scalar reward signal. Reward computation ranges from lightweight rule-based checks [19] to computationally expensive model-based judgments (e.g., LLM-as-a-Judge [49, 68]). Finally, the **training stage** consumes trajectories and rewards to update model weights with RL algorithms (e.g., PPO [40], GRPO [44]). To attain optimal training performance [10], production RL pipelines often adopt *synchronous* training, which enforces strict weight synchronization between rollout and training at every step.

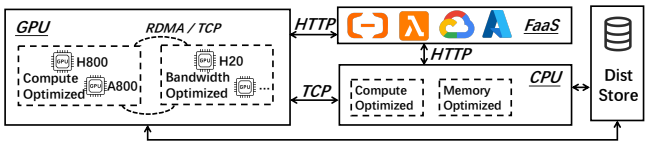


Figure 1: Disaggregated infrastructure for agentic RL.

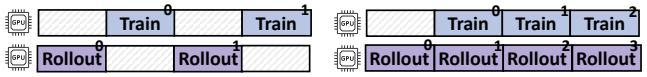


Figure 2: Synchronous vs. asynchronous training.

Environment Heterogeneity. A key challenge in agentic RL is environment diversity, which determines the system compute profile. As summarized in Table 1, agentic tasks vary widely in modality and interaction frequency. Complex reasoning tasks such as `SWE-bench` [23] (software engineering), `FrozenLake` [9] (visual games), and `WebShop` [61] (eCommerce) require long interaction horizons (up to 30–100 turns). Frequent interactions force the agent to repeatedly process growing context histories, making these workloads prefill-heavy and compute intensive. In contrast, tasks like `GEM-Math` and `GEM-Game` [3] may involve fewer turns (up to five) but require longer chains of thought per action. These workloads are decoding-heavy, shifting the bottleneck from compute to memory bandwidth. This variance requires infrastructure that adapts to each task's rollout profile.

2.2 Disaggregated Cluster Infrastructure

The Case for Disaggregation. The extreme heterogeneity of agentic RL workloads, ranging from compute-intensive training to stateful environment simulation (§2.1), renders monolithic architectures inefficient. Consequently, agentic RL training must transition to a *disaggregated infrastructure* that decouples these demand-conflicting computation stages into specialized resource pools. As shown in Figure 1, in a typical disaggregated infrastructure, *training clusters* utilize high-end, compute-optimized GPUs (e.g., NVIDIA H800) to sustain high throughput; *inference clusters* leverage bandwidth-optimized hardware (e.g., H20) to serve memory-bound decoding; *CPU clusters* provide elastic capacity for diverse, containerized runtime environments orchestrated by Kubernetes [55]; and *serverless infrastructure* handles bursty, stateless workloads like reward evaluation. These pools are interconnected through standard network fabrics, relying on distributed storage for persistent logging and fault tolerance.

Sync. vs. Async. Training. While disaggregation resolves resource mismatches, it introduces non-trivial orchestration challenges to the training paradigm that dictates the trade-off between system throughput and algorithmic consistency.

1) Synchronous Training: This paradigm enforces strict consistency by blocking rollout until it receives the latest model weights from the training cluster. In disaggregated settings, this causes substantial "dependency bubbles" (Figure 2-Left), where expensive GPUs sit *idle* during high-latency weight synchronization and straggler-bound environment steps (§3).

2) Asynchronous Training: To mitigate these bubbles, systems can adopt *asynchronous* paradigms (e.g., one-off RL training [32] in Figure 2-Right). Rollout and training then run *in parallel*: training consumes trajectories from slightly older policies (e.g., one iteration stale in one-off training), while rollout continuously produces new data. This design masks synchronization latency and straggler effects inherent to disaggregation, trading some policy staleness for higher hardware utilization and throughput.

Table 2: NVIDIA GPU specifications.

Hardware Specification	H20	H800
TFLOPS	148	989.5
HBM capacity	96GB	80GB
HBM bandwidth	4TB/s	3.35TB/s
NVLink bandwidth	900GB/s	400GB/s
Normalized Cost [69]	1.00	2.85

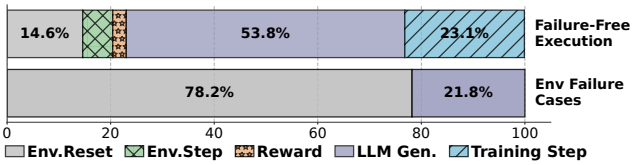


Figure 3: Breakdown of a training step: successful runs (top, avg=365.7s) versus execution with environment failures (bottom, avg=513.3s).

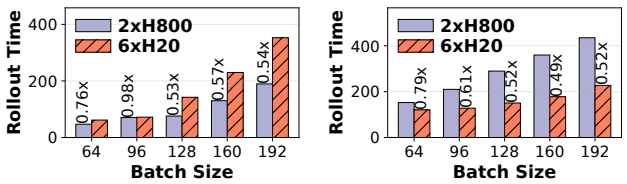
3 Characterization and Requirements

To motivate the design of RollArt, we conduct a comprehensive workload characterization of multi-task agentic RL training. Based on these empirical observations, we derive four critical system requirements that RollArt must satisfy: within-generation hardware affinity mapping (**R1**), trajectory-level asynchrony (**R2**), serverless reward offloading (**R3**), and bounded-staleness asynchronous training (**R4**).

3.1 Stage Computation

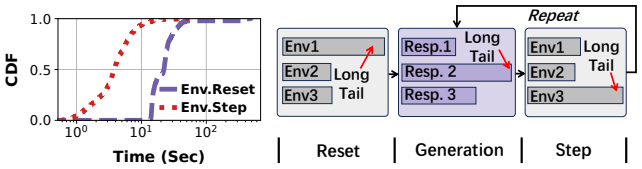
Training Step Latency Breakdown. We first profile the end-to-end latency of a standard training iteration to identify the dominant cost components. We train Qwen3-8B/32K on 32 H800 GPUs using the `SWE-bench` environment (batch size 128), where the agent LLM interacts with a containerized sandbox for software engineering tasks. The interaction involves two core operations: `env.reset` for environment initialization via Docker image pulling and container launching, and `env.step` for agent's action execution.

Figure 3 breaks down the latency of five *successful* iterations against five iterations containing *environment failures*.



(a) `FrozenLake` [Prefill-Heavy]. (b) `GEM-Math` [Decode-Heavy].

Figure 4: End-to-end rollout time (seconds) of different tasks on H20 and H800 GPUs across varying batch sizes.



(a) Env Time Distribution. (b) Batched Env Interaction.

Figure 5: The analysis of environment interaction: (a) Cumulative distribution function of time (log-scaled) taken for environment initialization (`env.reset`) and environment step (`env.step`). (b) Illustration of how long-tail environments affect multi-turn rollouts under batched env interaction.

Even on the success path, the average iteration time is 366 seconds, with LLM generation accounting for only 54%. The rest is split across training (23%), environment initialization (15%), and other overheads, a mixed profile that invalidates the common assumption of overwhelming generation dominance [12,18]. When environment timeouts occur, the average time spikes to 513 seconds and `env.reset` alone consumes 78% of rollout time, shifting the bottleneck entirely from GPU computation to environment overhead. Our production data indicates these failures are not rare corner cases, occurring approximately once every ten iterations (§8). Our empirical study therefore reveals that, in agentic RL, the system must handle four heterogeneous workloads, including generation, training, environment, and reward, rather than optimizing for the generation dominant regime reported by prior systems.

Divergent Hardware Affinities in Generation. Modern GPUs expose different trade-offs between compute capability, memory capacity, and cost (Table 2). While recent RL systems [15,56,67] advocate physically decoupling generation from training, assigning rollout to cost-effective, bandwidth-optimized GPUs and training to compute-optimized GPUs, our characterization reveals that *this static assignment is insufficient for agentic workloads*. LLM generation comprises two distinct phases with opposing resource demands: the compute-bound prefill phase and the memory-bandwidth-bound decoding phase. In environments with a few interaction turns but long chains of thought per action, most of the runtime is spent in decoding (decoding-heavy); conversely, in environments with many turns, the prefill phase dominates and demands high compute throughput (prefill-heavy). In our production clusters, agentic RL tasks exhibit a clear *bimodal distribution*, featuring either a small number of interaction turns (< 5) or a large number (> 10).

To quantify this divergence, we run a prefill-heavy task (`FrozenLake`) and a decoding-heavy task (`GEM-Math`) using Qwen3-8B/32K for ten iterations with prefix caching enabled. For a cost-equivalent comparison, we execute the workloads on two distinct hardware configurations: one with six H20 GPUs and the other with two H800 GPUs. As shown in Figure 4a, the compute-heavy H800 outperforms the H20 on `FrozenLake`, reducing end-to-end rollout time to as low as 0.53×. Conversely, for `GEM-Math` (Figure 4b), the H20's higher memory bandwidth accelerates decoding, reducing rollout time to 0.49×–0.79× of the H800. These results invalidate the assumption that generation is uniformly bandwidth-bound. Instead, maximizing throughput requires dynamically mapping tasks to their best-fit hardware.

R1: Generation is not uniformly bandwidth-bound; the system must bind each task to a best-fit GPU class rather than committing rollout to a single GPU type.

Heavy-Tailed Environment Execution. Since each training step collects a batch of trajectories, it runs hundreds to thousands of environments *simultaneously*, at which scale strong resource isolation becomes essential: without it, concurrent disk I/O exhausts shared quotas and triggers cascading failures. We therefore employ Kubernetes clusters to manage and isolate each containerized environment.

A substantial body of prior work [12, 18, 67] has analyzed the long-tail behavior of LLM generation. In agentic RL, environment interactions introduce additional sources of long-tail execution patterns. Figure 5a illustrates the latency distributions of `env.reset` and `env.step`, both exhibiting pronounced long tails. The long-tail delay of `env.reset` can reach hundreds of seconds in production, mainly due to (1) network contention, where concurrent Docker image pulls saturate network links, and (2) compute and I/O contention on host nodes, where launching containers consumes substantial CPU and disk resources. The cumulative `env.step` time per trajectory also varies widely, driven by the large variance in interaction turns and per-step overhead.

These long-tail trajectories act as stragglers that delay end-to-end rollout latency. Since LLM engines execute requests in batches, it is natural to batch environment interactions with the agent LLM as well. Figure 5b illustrates this pattern: fast environments must wait for the slowest one before the next generation step can proceed. Our profiling (Figure 3) indicates that batched environment interaction increases rollout time by up to 21.3% compared to ideal execution, an overhead that compounds as environment failure rates increase. Trace analysis on our production deployment (§8) confirms that such long-tail conditions are not rare throughout training, and would significantly penalize any batched execution scheme.

Table 3: Transmission overhead from the training cluster to the inference cluster over TCP and RDMA.

Model	Size (GB)	TCP (s)	RDMA (s)	Speedup
Qwen3-8B	15.26	6.911	5.466	1.264×
Qwen3-14B	27.51	14.437	5.817	2.482×
Qwen3-32B	61.02	29.649	9.442	3.140×

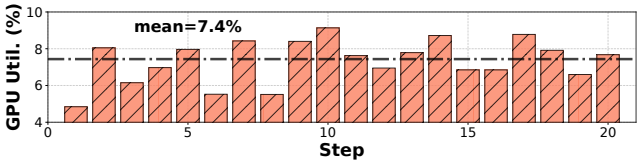


Figure 6: Inefficient resource usage when dedicating local GPUs for reward computation.

R2: Environment execution, including `env.reset` and `env.step`, is prone to extreme long-tail latency. The system must manage environment execution at trajectory granularity rather than batch granularity, so that slow or failed environments do not stall the pipeline.

Stateless Reward Computation. The reward stage follows rollout, and most reward computations can be implemented as *stateless functions*. For lightweight *rule- and code-based rewards*, they have small, bursty resource demand, making them a natural fit for elastic serverless execution that can scale to zero between steps. The *LLM-based reward computation*, on the other hand, demands intensive GPUs: colocating them with rollout contends with generation for GPU memory, KV-cache capacity, and scheduling slots, degrading both stages under concurrent demand. A common workaround is to reserve separate GPUs for the reward LLM: in our Qwen3-8B/32K `SWE-bench` run with batch size 128, we allocate four H800 GPUs to a dedicated 7B reward LLM and 28 H800 GPUs to rollouts, but the reward GPUs achieve only 7.4% average utilization across steps (Figure 6). Since the reward LLM's parameters remain fixed during training, it can be treated as a stateless function, and serverless deployment [11, 64, 65] simultaneously avoids rollout contention and reclaims the idle GPU budget wasted by dedicated reservations.

R3: Reward workers are stateless, bursty, and persistently underutilized; colocating them with rollout contends for GPU memory under concurrent generation, making elastic serverless deployment the better fit.

3.2 Inter-Stage Communication

Beyond computation, inter-stage communication plays a critical role in agentic RL training, comprising two distinct types: stability-critical, small-packet *trajectory transfer* and bandwidth-intensive, large-volume *weight update*.

Stability-Critical Trajectory Transfer. Trajectory and observation payloads are orders of magnitude smaller than the multi-GB weight transfers: a single agent-environment exchange typically carries kilobytes to a few megabytes of tokenized state and action data, yet each rollout issues hundreds to thousands of such exchanges per trajectory. Consequently, cumulative latency, rather than bandwidth, determines end-to-end interaction cost. In practice, environment interaction should therefore prioritize *network stability* over *network bandwidth*, and asynchronous execution between environment interaction and LLM generation prevents network latency from becoming the rollout bottleneck.

Bandwidth-Intensive Weight Update. During training, the agent LLM periodically updates its weights, which must then be synchronized with the rollout stage. This weight synchronization is the dominant source of inter-stage communication overhead. We measure the end-to-end transmission cost of synchronizing model parameters between the training and inference clusters using Mooncake [38] over TCP (200 Gbps Ethernet) and RDMA (400 Gbps InfiniBand), and report the results in Table 3. RDMA provides higher bandwidth and lower communication overhead than TCP. In *synchronous* RL training, the rollout stage can only proceed after the latest agent LLM weights have been synchronized. As a result, the substantial cost of weight transmission over low-bandwidth links increases end-to-end training time and diminishes the speedup of disaggregated training (Figure 2-Left).

In asynchronous training (Figure 2-Right), the training and rollout stages execute in parallel on separate GPUs. Although this introduces data staleness, many prior works [18, 29] empirically observe that asynchronous training can preserve model quality under a staleness bound. Given the dominant rollout overhead, asynchronous training can effectively hide both training and weight synchronization costs with rollout, thereby reducing end-to-end training latency.

R4: Training and rollout must execute concurrently on separate GPU clusters with *bounded* staleness for system efficiency and training quality.

4 System Overview

We present RollArt, a distributed system that orchestrates the rollout, reward, and training stages of agentic RL across disaggregated, heterogeneous resource pools. RollArt is implemented in approximately 60k lines of Python code. We first present the system architecture and then walk through one training iteration to illustrate its workflow.

4.1 Architecture

To meet the requirements in §3, RollArt is organized into three layered planes: resource, data, and control, as shown in Figure 7. The *resource plane* decides placement: it tracks heterogeneous hardware pools and binds each role, task, or

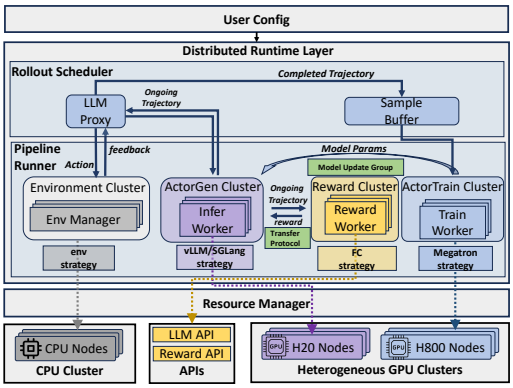


Figure 7: System Architecture of ROLLART.

generation sub-phase to a compatible resource class using hardware-affinity declarations (**R1**). The *data plane* realizes the pipeline through `Worker` and `Cluster` abstractions that encapsulate user-defined execution logic and offload stateless computation to serverless infrastructure (**R3**). The *control plane* coordinates progress: it schedules trajectories independently, dispatches reward computation as trajectories finish, buffers scored samples, and overlaps training with rollout under a bounded-staleness protocol (**R2** & **R4**).

Resource Plane. The resource plane is implemented by the *resource manager* (Figure 7), which monitors the state of disaggregated hardware pools and binds resources to specific `Workers` based on their roles and corresponding hardware affinities (e.g., training to H800 GPUs, rollout to H20 GPUs, environment execution to CPU servers, and reward computation to serverless infrastructure).

Data Plane. The data plane executes the distributed RL job on resources provisioned by the resource plane. It is implemented by the *pipeline runner*, which constructs role-specific `Clusters` for training, generation, environment execution, and reward computation. Each `Cluster` then launches its associated `Workers` and orchestrates their execution over the appropriate backend runtime (e.g., Megatron [48] and vLLM [25]). In this way, the data plane translates user-defined `Worker`/`Cluster` logic into concrete distributed execution.

Control Plane. The control plane coordinates trajectory-level rollout and asynchronous training. It is centered on the *rollout scheduler*, which orchestrates the interaction among environment execution, generation workers, reward workers, and training workers. To decouple these stages, ROLLART introduces two auxiliary components: `LLMProxy`, which mediates generation requests between environment execution and generation `Workers`, and `SampleBuffer`, which buffers completed and scored trajectories for training. Together, these components allow rollout, reward evaluation, and model updates to proceed concurrently, hiding trajectory-level synchronization and weight-update coordination from the user.

4.2 End-to-End Workflow

To illustrate how the three planes interact, we walk through the first asynchronous training iteration in ROLLART. At startup, the resource manager in the resource plane binds heterogeneous GPU, CPU, and serverless resources to the corresponding `Clusters` by allocating and placing their constituent `Workers` according to user-specified hardware affinities. The pipeline runner then materializes the data plane by instantiating the corresponding `Clusters` for training, generation, environment execution, and reward computation, each backed by its assigned `Workers` and execution engine.

Once the pipeline is instantiated, the rollout scheduler in the control plane launches multiple `EnvManager` instances to collect trajectories. Each `EnvManager` alternates between issuing generation requests through the shared `LLMProxy` and applying the returned actions to its environment. Completed trajectories are scored by a reward backend and deposited into the `SampleBuffer`, from which the training `Cluster` consumes batches and updates the model. Rollout, reward evaluation, and training proceed concurrently.

The following sections detail these planes. §5 describes the programming abstractions and mechanisms that configure the resource and data planes. §6 describes the system-managed control plane that drives runtime execution, followed by cross-cutting optimizations that span multiple planes.

5 Programmable Resource and Data Planes

ROLLART provides a declarative programming model that separates user intent from system execution. Users specify execution logic and task-specific mappings for the data plane, as well as hardware preferences for the resource plane, through the `Worker`-level interface. The resource manager interprets the resource-plane declarations to construct concrete bindings over heterogeneous resources, while the pipeline runner materializes the data plane through `Cluster` abstractions.

5.1 `Worker` and `Cluster` Abstractions

The resource and data planes are built around two core abstractions: `Worker` and `Cluster`.

`Worker` Abstraction. A `Worker` is the basic execution unit that spans both the resource and data planes. It encapsulates user-defined computation logic together with method-level declarations, and executes on the hardware provisioned by the resource plane. A `Worker` can be specialized into the roles required across stages, including training, generation, reward, and environment, by subclassing and method annotations.

`Cluster` Abstraction. While `Workers` execute the actual computation logic, managing thousands of `Workers` individually is prohibitive for algorithm developers. To address this, ROLLART introduces the `Cluster` abstraction. A `Cluster`

```python
import rollart.distributed as rdist
from rdist.worker import ActorTrainCls, ActorGenCls,
    RewardCls
from rdist import ResourceManager as RM

# 1. Single Controller Example
class MyActorTrain(ActorTrainCls):
    @rdist.register(mode="execute_all")
    def compute_gradients(self, input_tensor):
        ...

# 2. Define actor_gen on heterogeneous GPUs
# 2.1 heterogeneous GPU allocation.
gen_rm=RM( {"H800": list(range(0, 8))},
           {"H20", list(range(8, 32))})
# 2.2 hardware affinity mapping
class HeteroActorGen(ActorGenCls):
    @rdist.hw_mapping(
    hw_affinity={"FrozenLake": "H800","default": "H20"}
    )
    def generate(self, input_ids:List[int],
                 tag_name:str="default"):
        return self.model.process(prompt)

# 3. Define a serverless reward computation func
class ServerlessRewardWorker(RewardCls):
    @rdist.register_serverless(
        attribute='reward_proxy',
        serverless_url='fc://xxx.xxx')
    def compute_rewards(self, traj: list):
        prompt = f"Evaluate the trajectory:{traj}"
        return ray.get(self.reward_proxy(prompt))
```

Listing 1: Worker-level programming interface of ROLLART.

acts as a proxy and controller for a role-specific `Worker` group. A `Cluster` spawns `Workers`, binds their methods, and manages collective invocation on their behalf. In practice, ROLLART defines four `Clusters`: `ActorTrain`, `ActorGen`, `Reward`, and `Environment`, corresponding to the four RL stages shown in Figure 7. By composing these abstractions, ROLLART maps the agentic RL pipeline onto the disaggregated hardware fabric.

5.2 `Worker` Declarations and Binding

ROLLART exposes three worker-level interfaces through Python decorators, allowing users to implement role-specific computation logic and specify task-specific mappings and hardware preferences under a unified *single-controller* model. The resource manager interprets the resource-related declarations to bind `Workers` to heterogeneous resources.

Decorator-Based Interface. Listing 1 shows the three decorator-based interfaces for `Worker` declarations.

1) Single Controller. Like other industry RL frameworks [46, 70], ROLLART adopts the single controller programming model to streamline pipeline construction. When a `Worker` method is annotated with the `register` decorator and set to `execute_all` mode (Lines 7–8 in Listing 1), the runtime broadcasts inputs and invokes the method across all `Workers` in the corresponding `Cluster`. The runtime also collects the returned results and automatically aggregates them. A typical use case is defining how gradients are computed on each training `Worker` (Line 8 in Listing 1).

2) Hardware-Affinity Mapping. To align workload characteristics with hardware capabilities (**R1**), ROLLART allows users to declare preferred hardware targets for role-specific workers through the `hw_mapping` decorator. By default, training workers (`ActorTrainCls`) are mapped to compute-optimized GPUs (e.g., H800), generation workers (`ActorGenCls`) to bandwidth-optimized GPUs (e.g., H20), environment workers (`EnvironmentCls`) to CPU servers, and reward workers (`RewardCls`) to local GPU servers. Users can override these defaults with finer-grained preferences.

As shown in the `HeteroActorGen` class, a dictionary-based resource specification provisions heterogeneous GPU groups (Lines 13–14), and the `hw_mapping` decorator (Lines 17–19) declares sub-stage affinities. Prefill-heavy `FrozenLake` rollouts are routed to H800 GPUs, while all other tasks default to H20s. At runtime, the rollout scheduler passes the current task's identity (e.g., `FrozenLake`) as the `tag_name` argument (Line 21), allowing the system to route each generation request to the most appropriate hardware.

This declaration is coarse-grained by design: users specify affinity at the task-domain level rather than performing per-request load-balancing. In our production traces, different agentic task domains diverge in computation characteristics (e.g., turn counts and prefill/decode ratios), making domain labels a practical basis for hardware selection (§8). Our evaluation confirms this lightweight annotation captures enough heterogeneity to improve throughput (§7.3). This leaves affinity selection explicit, and §9 discusses how an online profiler could automate or refine these mappings at runtime.

3) Serverless Registration. Reward workers are stateless and exhibit low utilization on dedicated GPUs (**R3**). To address this, ROLLART provides the `register_serverless` decorator (Lines 26–28) to offload reward computation to serverless platforms. With the specified `serverless_url`, the runtime invokes `compute_rewards` (Line 29) as a pure function through the same method-level interface while redirecting execution to an external serverless platform (e.g., Function Compute [1]). This grants the system zero-overhead autoscaling without provisioning costly, hot-standby GPUs.

Resource Binding. The resource manager uses a shared metadata store (e.g., Redis) to maintain a global, real-time view of resource pools, including compute-optimized GPUs (e.g., H800), bandwidth-optimized GPUs (e.g., H20), CPU clusters, and serverless endpoints. `Worker`-level annotations specify hardware affinities and execution targets for individual methods. Upon a `Worker` deployment request, the resource manager interprets these declarations to determine concrete placements and bindings. It first checks the availability of the preferred resource pool and binds the requested `Worker` accordingly. If the preferred hardware is temporarily unavailable, the manager opportunistically falls back to compatible default resources rather than stalling deployment. The resulting binding metadata is recorded for subsequent dispatch, failover, and reconfiguration.

```python
class Cluster:
    def __init__(self, res_manager, worker_cls):
        self._create_worker(worker_cls, res_manager)
        self._bind_worker_method()

    def execute_all(self, method_name, *args, **kwargs):
        result = []
        for worker in self.workers:
            rcall = getattr(worker, method_name)
            result.append(rcall(*args, **kwargs))
        return ray.get(result)

    def hw_mapping(self, hw_affinity, tag_name, *args):
        hw_type = hw_affinity.get(tag_name)
        new_workers = []
        for worker in self.workers:
            if worker.resource_type == hw_type:
                new_workers.append(worker)
        # route requests to new_workers next

    def register_serverless(self, attr, url, *args):
        # define a call_fc to call serverless url
        for worker in self.workers:
            setattr(worker, attr, call_fc)
        # perform execute_all logic next
```

Listing 2: Simplified implementation of `Cluster`.

5.3 `Cluster`-Level Realization

The resource manager realizes the resource plane, while the `Cluster` governs execution in the data plane. We next describe how the default `Cluster` implementation turns `Worker`-level decorators into concrete execution behavior.

`Cluster` Construction and Method Binding. At initialization, `Cluster` assembles a role-specific `Worker` group from the specified `worker_cls` using resources provisioned by the resource manager, and initializes the corresponding backend runtime (e.g., Megatron or vLLM) so that `Worker` methods execute on that backend (Line 3 in Listing 2). Each `Worker` is associated with resource metadata, such as its hardware type, enabling later affinity-aware dispatch. The `_bind_worker_method` function then binds each method of `worker_cls` to the `Cluster` instance, allowing the `Cluster` to act as an invocation proxy (Line 4). For example, if `worker_cls` defines `compute_gradients` (Line 8 in Listing 1), users can directly invoke `Cluster.compute_gradients`.

Realizing `Worker` Declarations. For methods annotated with the `register` decorator, `Cluster` enters the `execute_all` path, which calls the target method on every constituent `Worker` and aggregates results via `ray.get` (Lines 6–11). For methods annotated with `hw_mapping`, `Cluster` inspects the `tag_name` argument, filters for `Workers` whose resource type matches the preferred hardware, and routes the request to those `Workers` (Lines 13–19). For methods annotated with `register_serverless`, `Cluster` replaces the reward proxy attribute (`attr`) with a callable that invokes the registered serverless URL, so reward computation is performed by the serverless backend (Lines 21–25). In all affinity-based paths, if the preferred target is temporarily unavailable, `Cluster` redirects execution to a compatible fallback provided by the resource manager, ensuring forward progress under transient contention.

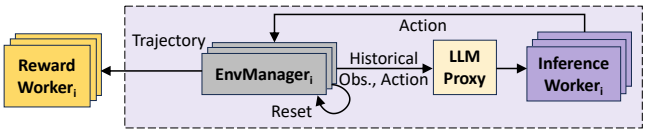


Figure 8: Trajectory-Level Rollout and Reward Overview.

The abstractions and mechanisms above, including `Worker` logic, decorator annotations, resource bindings, and `Cluster`-level realization, constitute what users configure at pipeline setup time. Once the pipeline is materialized, the control plane takes over: it drives trajectory-level rollout, reward evaluation, and asynchronous training without further user intervention, which we explain next.

6 System-Managed Control Plane

ROLLART's control plane is entirely *system-managed*: users specify no coordination logic. In this section, we first describe how the control plane orchestrates trajectory-level rollout, reward, and asynchronous training (§6.1–§6.2), then present cross-cutting optimizations that span the data and control planes to improve end-to-end efficiency (§6.3).

6.1 Trajectory-Level Rollout and Reward

As identified in §3 (**R2**), batched environment interaction forces fast environments to wait for stragglers, inflating end-to-end rollout latency. To eliminate this synchronization barrier, ROLLART adopts a *trajectory-level* design in which each trajectory progresses through generation, environment interaction, and reward computation independently. Figure 8 gives an overview. Three components realize this: `LLMProxy` dispatches requests at per-trajectory granularity, `EnvManager` drives each environment's lifecycle independently, and `Reward Workers` score asynchronously.

`LLMProxy`: Trajectory-Level LLM Generation. As shown in Figure 7, `LLMProxy` sits between `EnvManagers` and inference workers in the control plane. It acts as a gateway that decouples generation clients from the serving instances, dispatching per-trajectory requests across inference workers.

Each inference worker runs a command-driven event loop (Figure 8) that manages an inference engine (e.g., vLLM [25], SGLang [41]). This loop operates continuously in a non-blocking fashion with two components:

1) Step Wise Command Processing. The loop polls for commands dispatched by `LLMProxy`, `ADD` to enqueue requests and `ABORT` to cancel existing ones. When no commands are pending, it advances the engine by executing a decode or prefill step for a batch of requests, keeping GPU utilization high. Because commands are processed between engine steps, adding or aborting a trajectory does not stall ongoing generation.

2) Post-Processing. When the engine finishes a request, the loop immediately invokes a pre-registered callback that post-processes the output and returns the result to the requesting `EnvManager`. This allows each trajectory to proceed to environment interaction as soon as its generation completes, without waiting for stragglers.

`EnvManager`: Trajectory-level Environment Interaction. Each `EnvManager` is a lightweight controller that manages the lifecycle of a single environment to collect trajectories (also depicted in Figure 8). It begins with environment initialization via `reset`, after which it enters an independent event loop that orchestrates the interaction between an environment instance and the shared `LLMProxy` via `step`. During this loop, the `EnvManager` maintains a list of (observation, action) pairs to construct a trajectory. Specifically, it feeds `LLMProxy` with the historical (observation, action) sequence as input to obtain the next action, applies this action to the environment via `step`, and records the resulting observation. Unlike batched environment interaction (Figure 5b), where all environments must synchronize before the next generation step, each `EnvManager` operates on its own timeline, so a slow environment never blocks others.

Overlapping Rollout and Reward. Once a trajectory completes, the runtime dispatches it to a reward worker, which invokes a serverless reward function as a non-blocking task. Because each `EnvManager` yields trajectories independently, reward computation begins as soon as any single trajectory finishes, without waiting for an entire batch. ROLLART launches multiple inference workers, `EnvManagers`, and reward workers so that generation, environment interaction, and reward computation overlap in time, hiding reward latency behind ongoing rollouts and maximizing pipeline throughput. This design avoids colocating reward on the inference cluster, which would force batched computation. Recent RL post-training systems [10, 12, 57, 59, 67] also adopt asynchronous reward computation for this reason. ROLLART further eliminates the underutilization of dedicated reward GPUs (Figure 6) through serverless deployment.

6.2 Asynchronous Training Orchestration

In a disaggregated setup, synchronous training forces rollout workers to idle while model weights are transferred across clusters. This cost grows with model size (§3, **R4**). To hide this overhead, ROLLART overlaps rollout and training: two stages execute concurrently on separate GPU clusters, coordinated through a weight synchronization protocol (Figure 9).

Weight Synchronization Protocol. To propagate model weights to inference workers while preserving the overlap between rollout and training, ROLLART implements a six-step synchronization protocol in each iteration. ① `get_batch`: The runtime invokes a blocking `get_batch` call to retrieve trajectories from the `SampleBuffer`, which buffers scored

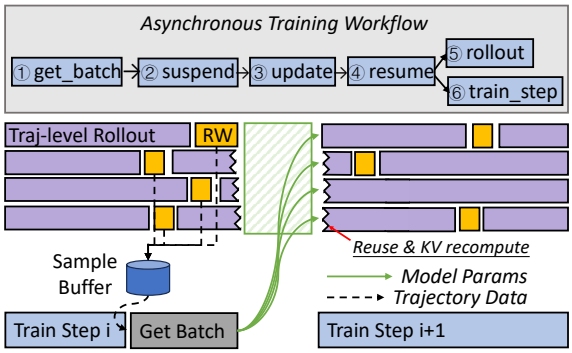


Figure 9: Asynchronous Training Workflow.

trajectories for training (Figure 7). The pipeline waits until a predefined batch size is collected. ② `suspend`: Once the batch is ready, the runtime issues a `suspend` command to the `LLMProxy`, preventing new generation requests from being accepted while preserving in-flight trajectories. ③ `update`: After rollout is suspended, the pipeline fetches the latest LLM weights from the training workers and updates all inference workers. ④ `resume`: Upon completion, a `resume` command is issued to the `LLMProxy`, which then continues pending generation requests. ⑤ `recomp` & `rollout`: To reuse in flight trajectories generated under the previous model version, ROLLART performs KV cache recomputation, rebuilding each trajectory's KV cache under the updated weights so that generation can continue without restarting from scratch. ⑥ `train_step`: In parallel with the resumed rollout, the pipeline executes `train_step` on the batch retrieved in ①, allowing training to overlap with ongoing rollout.

Asynchronous Bound and Buffer Management. The protocol above lets rollout and training proceed concurrently, but trajectories in `SampleBuffer` may come from different model versions. Excessive staleness can increase variance and destabilize training, so ROLLART enforces a per-trajectory *asynchronous bound* α (**R4**). If the current agent LLM is at version n, any buffered trajectory must have been initiated by a version no older than $(n-\alpha)$; trajectories outside this window are aborted. This same version window bounds buffer growth: with E concurrent environments, `SampleBuffer` holds at most $O(\alpha \cdot E)$ pending trajectories across versions. Before `get_batch` forms a training batch, it eagerly evicts stale trajectories, so highly asynchronous or out-of-order completion cannot cause unbounded buffer growth. Our empirical study (§7.2) shows that $\alpha = 1$ balances training speed and stability.[1]

6.3 Cross-Cutting Optimizations

The execution workflow above relies on data movement, hardware-specialized serving, and scheduling mechanisms

[1] Unlike AReaL [10], which bounds staleness only at trajectory start, ROLLART controls trajectory-level staleness in each iteration to prevent long-tail trajectories from spanning multiple model versions.

that span the data and control planes. We describe three optimizations that improve end-to-end efficiency.

Data Movement. ROLLART matches each data path to an appropriate transfer mechanism. The *transfer protocol* streams trajectories and supervision signals between stages as Ray's object references [34]. The exchanged objects are sharded to match each worker's parallelism layout. The *model update group* synchronizes weights between stages. Intra-cluster weight synchronization uses NCCL [35] over NVLink or InfiniBand. The main bottleneck is *cross-cluster* weight updates, which bridge the training cluster and inference cluster over lower-bandwidth Ethernet. ROLLART addresses this with an *asynchronous weight update engine* built on Mooncake [38]: after each training step, updated weights are bucketized (e.g., 1GB) and published to a remote CPU-resident Mooncake store rather than synchronously pushed to remote inference workers. Inference workers then asynchronously fetch the latest weight buckets from the Mooncake store on demand, decoupling weight transfer from ongoing rollouts and avoiding GPU stalls caused by slow cross-cluster communication.

This makes the data movement explicit: training workers write weight buckets once over the lower-bandwidth cross-cluster link, while inference workers pull them over high-bandwidth intra-cluster links. The additional hop through the Mooncake store introduces only minimal overhead, as we quantify in §7.4. While Mooncake adds a pull step, it uses optimized data movement (e.g., zero-copy transfers) and is overlapped with ongoing rollout and weight push. For smaller models, the absolute transfer volume is also lower, limiting the exposed latency. We quantify both the accumulated pull cost and the non-overlapped exposed overhead in §7.4.

Redundant Environment Rollouts. Within the control plane, the trajectory-level design of `LLMProxy` and `EnvManager` enables an optimization we term *redundant environment rollouts*. This allows the system to launch more environments than required for trajectory collection. Once the target number of trajectories has been collected, in-flight trajectories can be terminated. Because rollouts are managed at trajectory granularity, slow or failed environments do not block faster ones, mitigating stragglers (§7.4) and environment failures (§8).

Prefill-Decoding (PD) Disaggregation. Within the data plane, PD disaggregation, which is widely adopted in LLM serving systems [37, 66], refines the `LLMProxy` routing described in §6.1. ROLLART supports PD disaggregation by deploying prefill and decoding workers on compute-optimized and bandwidth-optimized GPUs, respectively, and routing the two phases of each request to the corresponding workers. We evaluate the above optimizations in §7.4.

7 Evaluation

In this section, we present the end-to-end evaluation (§7.2), ablations of the four design requirements(§7.3), cross-cutting optimizations (§7.4), and the system's disaggregation tax (§7.5).

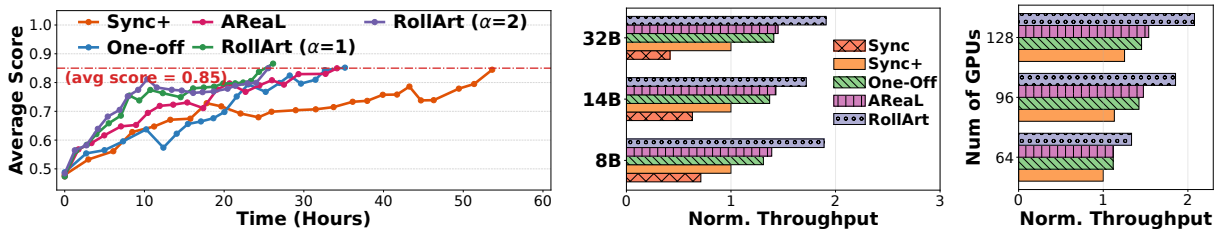
7.1 Evaluation Setup

Models and Tasks. We train Qwen3 [60] models from 8B to 32B on a diverse mix of agentic tasks (Table 1), using a 32k-token maximum context length. Because `SWE-bench` is substantially harder than the other tasks, we train it only on Qwen3-32B. For mathematical tasks, we use Qwen2.5-7B as the reward LLM to validate the reasoning process.

Infrastructure. We deploy ROLLART on a 96-GPU H800 cluster and a 32-GPU H20 cluster. GPU nodes within each cluster are connected by 400 Gbps InfiniBand; cross-cluster communication uses 200 Gbps Ethernet. We use one dedicated CPU cluster for `SWE-bench`, another for the remaining environments, and our internal serverless platform for reward workers. Unless otherwise stated, all experiments use 128 GPUs. We use NCCL [35] v2.26.5 for intra-cluster weight updates and a Mooncake v0.3.7 [38] storage server for cross-cluster communication.

Training Configuration. We train with GRPO [19], a batch size of 512, a group size of 8, and uniform task sampling. For asynchronous training, we reserve 32 H800 GPUs for training and use the remaining H20 and H800 GPUs for rollouts. Rollout tensor-parallelism degrees for Qwen3-8B/14B/32B are 1, 2, and 4, respectively; the corresponding training-side tensor and pipeline parallelism are tuned for throughput. Rollouts run on vLLM 0.8.4 with prefix caching and CUDA graphs enabled, while training runs on Megatron v0.12.2.

Baselines. We compare against four post-training baselines, grouped into sync and async designs. Since no existing open-source system supports our full spectrum of agentic tasks, we implement two sync baselines on top of ROLLART: **Sync**, a standard synchronous RL pipeline, and **Sync+**, which strengthens Sync with async reward computation, async environment interaction, and serverless offloading—techniques that are widely adopted in recent systems [10, 57, 67]. For async baselines, we compare with **One-off** [32] (see Figure 2-Right) and **AReaL**. One-off overlaps rollout and training by consuming trajectories from the previous step, but requires all trajectories in an iteration to finish with stale model weights. Differently, ROLLART with $\alpha = 1$ preempts and resumes ongoing rollouts in the next iteration. We re-implemented AReaL in ROLLART's codebase, adopting AReaL's staleness bound of one, which constrains trajectory-level staleness only at the start rather than at every turn of every trajectory, to ensure fairness. Both One-off and AReaL implementations incorporate the Sync+ optimizations, so they differ from ROLLART mainly in the coordination between training and rollout for staleness control, as well as the lack of hardware affinity routing. **Laminar** [45], another async baseline, repacks long-tail rollouts but does not account for hardware affinity, serverless reward computation, or long-tail environment exe-



(a) Time-to-Score (target 0.85) on Qwen3-32B. (b) Norm. Throughput across LLMs. (c) Scaling Efficiency.

Figure 10: End-to-end results: (a) time-to-score on Qwen3-32B; (b) throughput across LLMs (normalized to Sync+); (c) throughput of Qwen3-14B across H800 GPU counts, normalized to Sync+ on 64 H800 GPUs.

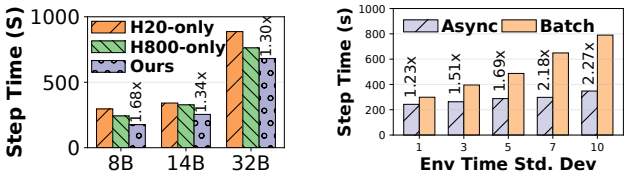
cution, and we provide a quantitative feature-decomposition argument in §7.2 (Throughput Efficiency). We run baselines on 128 H800 GPUs, so ROLLART incurs roughly 83% of the baselines' per-GPU-hour cost.

Metrics. We measure end-to-end latency as the average step time over five iterations, and throughput as the number of prompt and response tokens in a global batch divided by step time [47]. We report the average validation score across tasks.

7.2 End-to-End Evaluation

Model Convergence. We measure validation score every ten iterations on Qwen3-32B, the most network-sensitive model in our setup, and report the time to reach a target score of 0.85 in Figure 10a. The asynchronous configuration with $\alpha = 1$ reduces step time by 2.05×, 1.35×, and 1.31× over Sync+, One-off, and AReaL, respectively. One-off already improves over Sync+ by 1.52× by overlapping rollout and training. AReaL converges faster in the early stage but achieves similar time-to-score performance to the One-off approach because it bounds staleness only at the start of trajectories, collecting more stale long-tail trajectories. ROLLART enforces per-step staleness control within the bound α in §7.3 and promptly aborts stale trajectories. Increasing the bound to $\alpha = 2$ for ROLLART improves initial convergence but slightly worsens later time-to-score relative to $\alpha = 1$, consistent with the throughput–quality tradeoff in §7.3. The results in Figure 10b corroborate the advantages of ROLLART over these baselines across 8B, 14B, and 32B.

Throughput Efficiency. Figure 10b reports throughput efficiency normalized to Sync+; we set $\alpha = 1$ for ROLLART. The results show how throughput improves as the baselines add more overlap and scheduling flexibility. Sync+ first improves throughput by 1.40–2.40× over Sync by adding async reward computation, async environment interaction, and serverless offloading. One-off adds 1.31–1.47× over Sync+ by overlapping rollout and training. AReaL contributes another 1.03–1.06× over One-off. ROLLART then adds 1.22–1.36× over AReaL by layering hardware-affinity mapping on bounded-staleness async training. This ROLLART/AReaL gap is consistent with the hardware-affinity



(a) Rollout Efficiency. (b) Traj.-Level vs. Batch Rollout.

Figure 11: The efficiency of (a) hardware affinity and (b) trajectory-level rollout, where env latency is sampled from a Gaussian distribution with mean $\mu = 10$ s and standard deviation σ varying from 1 to 10 s (x-axis).

contribution measured in isolation in §7.3 (R1, 1.12–1.37× over an H800-only rollout). Overall, ROLLART achieves 2.65–4.58× throughput over Sync.

Laminar [45] is closed source, so we cannot evaluate it directly. Instead, the decomposition above isolates the capabilities that Laminar does not provide. The measured ROLLART/AReaL gap captures the effect of hardware affinity (§7.3, R1). Laminar does not optimize for environment instability, which yields a 1.23–2.27× improvement under environment variability (§7.3, R2), and it lacks serverless reward computation, which can reduce rollout time by up to ∼ 2× (§7.3, R3). Composing these isolated gains lower bounds the gap between ROLLART and Laminar.

Scaling Efficiency. We train Qwen3-14B while sweeping the cluster size from 64 to 128 H800 GPUs. Thus, ROLLART does not support hardware-affinity mapping in this evaluation. We compare the scaling performance of ROLLART under $\alpha = 1$. Figure 10c reports throughput normalized to Sync+ on 64 GPUs. As cluster size increases, Sync+ exhibits diminishing marginal gains, and One-off and AReaL show similar limited scaling performance from 96 to 128 GPUs, whereas ROLLART delivers 1.33–2.08× higher throughput, demonstrating better scaling efficiency.

7.3 Ablations of Design Requirements

We next present ablations of the four design requirements.

R1: Hardware-Affinity Mapping. We evaluate hardware-affinity mapping across LLM sizes on compute-optimized

and bandwidth-optimized GPUs. To isolate routing, we fix training to 32 H800 GPUs and compare three rollout configurations that yield approximately an equal cost: 72 H800 GPUs, 208 H20 GPUs, and an affinity-aware mix of 64 H800 plus 24 H20 GPUs. The mixed configuration routes mathematical and game-oriented agentic tasks to H20 GPUs while keeping prefill-heavy work on H800 GPUs. As shown in Figure 11a, RollArt reduces step time by 1.30–1.68× over H20-only and by 1.12–1.37× over H800-only. H20-only performs worst because many agentic tasks still incur frequent prefill operations, while the mixed configuration adds H20 capacity for decoding-heavy tasks without moving prefill-heavy work off compute-optimized GPUs. Beyond task-level mapping, PD disaggregation extends RollArt's affinity routing from task-level to phase-level placement; we evaluate it as a cross-cutting optimization in §7.4.

R2: Trajectory-Level Asynchrony. Trajectory-level rollout decouples environment interaction across trajectories, so slow turns do not stall an entire batch (§6.1). To isolate this effect, we run Qwen3-8B with a 32k context and inject per-turn environment latency sampled from Gaussian distributions with mean $\mu = 10$ s and standard deviation σ ranging from 1 s to 10 s. This synthetic injection controls latency variance; production-side straggler stacking is reported in §8. Figure 11b compares trajectory-level and batch-level environment interaction using average step time over ten iterations. As σ increases, trajectory-level rollout improves over batch-level interaction from 1.23× to 2.27×, showing that it better absorbs environment variance.

R3: Serverless Offloading. We evaluate serverless offloading against a local-GPU reward setup on a 16-H800 cluster. We run three concurrent mathematical agentic RL jobs with Qwen3-8B/16k as the actor agent and Qwen2.5-7B as the reward LLM. Both serverless and local setups reserve eight GPUs for training. The local setup assigns four remaining GPUs to rollout and the other four to reward, whereas the serverless setup assigns all eight remaining GPUs to rollout and offloads reward computation to an elastic serverless platform. Figure 12 reports GPU utilization and per-step rollout time for a batch size of 84, including asynchronous reward computation. Serverless offloading raises average GPU utilization from 6% to 88% and doubles the local rollout allocation, reducing average rollout time from 158 s to 77 s. Although remote serverless reward calls introduce network I/O, the disaggregation tax measured in §7.5 is small (max 2.1 s, mean 0.01 s per call). Thus, transfer cost does not negate the resource efficiency gains from serverless offloading.

R4: Asynchronous Bound. We quantify the throughput–convergence tradeoff induced by the per-trajectory asynchronous bound α. Figure 13 sweeps α from 1 to 6 and reports average step time across LLMs. Larger bounds reduce staleness-triggered trajectory aborts and lower step time, but the gain quickly plateaus: the best bound differs by LLM

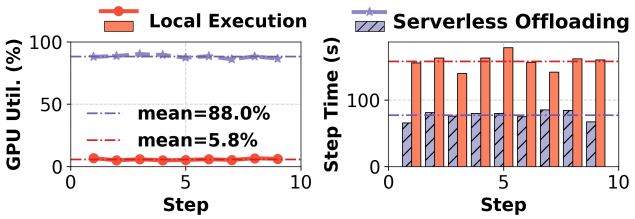


Figure 12: Comparison between dedicating local GPUs and using serverless offloading.

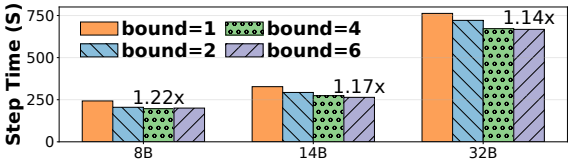


Figure 13: Average step time across LLMs and bounds.

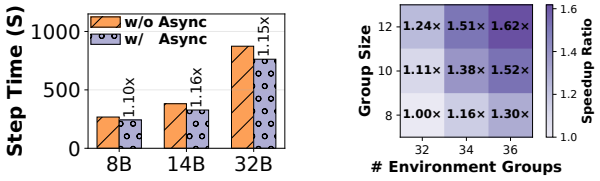
and improves step time by at most 1.22× over $\alpha = 1$. These throughput gains do not necessarily improve time-to-score. Figure 10a shows that $\alpha = 2$ already incurs a measurable late-stage time-to-score regression , and we set $\alpha = 1$ by default.

7.4 Cross-Cutting Optimizations

We evaluate the three cross-cutting optimizations in §6.3.

Asynchronous Weight Transfer. Cross-cluster weight transfer is costly when training and rollout clusters communicate over heterogeneous interconnects. To quantify the benefit of overlap, Figure 14a compares RollArt's asynchronous cross-cluster communication with veRL's NCCL-based scheme, which assumes uniformly high-bandwidth links across servers. Asynchronous communication reduces end-to-end step time by 1.10–1.16× across LLMs. Table 4 decomposes this gain. *Weight Push* is the cost of streaming updated weights from the training cluster to the Mooncake store over the cross-cluster TCP/Ethernet link. *Accumulated Weight Pull* is the total cost of inference workers pulling new weight buckets from Mooncake, while *Exposed Weight Pull* is the residual pull cost not hidden by ongoing rollout. The *Naive (Push + Acc. Pull)* row reports the cost exposed by a synchronous design such as veRL. Asynchronous overlap hides 67–78% of the pull cost, reducing the exposed overhead to at most 9.6 s on 32B versus 38.6–157.0 s without overlap.

Redundant Environment Rollouts. Redundant rollouts exploit GRPO's group structure to mask environment failures



(a) Cross-Cluster Comm. (b) Redundant Env Rollout.

Figure 14: Cross-cutting optimizations: (a) async cross-cluster weight transfer; (b) redundant environment rollouts.

Table 4: Breakdown of ROLLART's asynchronous cross-cluster weight transfer (seconds).

Cost	8B	14B	32B
Naive (Push + Acc. Pull, no overlap)	38.6	84.1	157.0
Weight Push	32.4	67.8	127.3
Accumulated Weight Pull	6.2	16.3	29.7
Exposed Weight Pull (ROLLART)	1.4	5.1	9.6

and stragglers. We run Qwen3-8B/32k on 32 H800 GPUs to vary the number of environment groups and the group size on the `GEM-math` task. Figure 14b reports rollout speedup ratios. Larger groups and more environment groups increase the chance that a fast subset finishes early, sustaining efficiency under straggler conditions. The maximum speedup reaches 1.62×, and increasing either parameter yields a speedup.

PD Disaggregation. PD disaggregation extends hardware-affinity mapping from task-level to phase-level placement. We train Qwen3-32B and Qwen3-30B-A3B models on `SWE` tasks with batch size of 128 and 32k sequence length. We evaluate two configurations: 1P3D and 2P2D where each prefill node uses eight H800 GPUs and each decoding node uses eight H20 GPUs. For the dense model, PD disaggregation achieves relatively modest speedups of 1.03× and 1.05× compared with colocation. For the MoE model, 1P3D and 2P2D achieve 1.11× and 1.21× speedup over colocation, respectively. The performance and optimal configuration[2] of PD disaggregation are workload- and model-dependent. §9 further discusses how to automate these configurations to adapt to workload demands.

7.5 Disaggregation Tax

We next quantify the disaggregation tax along three data paths.

Cross-Cluster Weight Synchronization. Weight transfer is the largest overhead, but the asynchronous data-movement in §7.4 hides most of it. As Table 4 shows, ROLLART exposes 1.4–9.6 s of residual pull cost; without overlap, rollout would block on Push + Accumulated Pull for 38.6–157.0 s.

Env-Interaction I/O. Trajectory-level rollout moves data between the environment and inference clusters at each interaction. Across our evaluation, the maximum transferred volume is 2.7 MB, and the maximum and average per-call transfer overheads are 1.4 s and 0.02 s, respectively. These costs are negligible relative to per-step training time.

Serverless Reward I/O. Remote reward calls transfer trajectory payloads of up to 5.2 MB, with maximum and average per-call overheads of 2.1 s and 0.01 s. The residual disaggregation tax is therefore dominated by cross-cluster weight transfer, which ROLLART overlaps with rollout, so data movement does not erase the end-to-end gains reported in §7.2.

[2]We also evaluated the 3P1D configuration, and both models achieved worst rollout performance, as the single decoding node became a bottleneck.

Table 5: Rollout time (seconds) comparison between colocation and disagg. under different models and configurations.

Model	1P3D	Colocate	2P2D	Colocate
Qwen3-32B	722.7	741.2	701.6	734.9
Qwen3-30B-A3B	294.8	327.4	251.1	305.2

8 ROLLART in Production

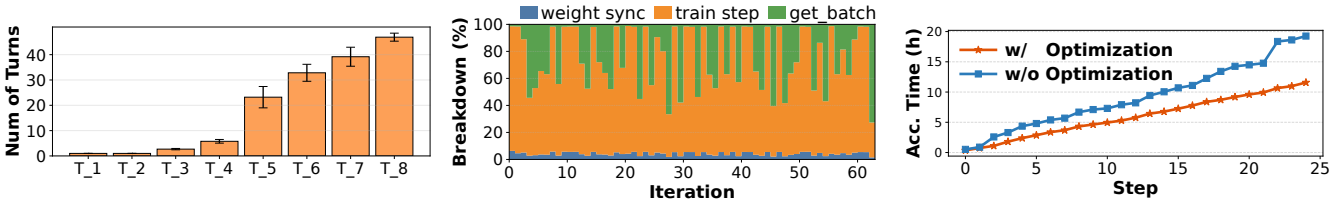
Over the past nine months, thousands of agentic RL jobs have used ROLLART for post-training. We report one large production deployment on more than 3000 GPUs to show how the workload properties in §3 appear in practice and how operators tune ROLLART at scale.

Workload Characterization. We trained a hundreds-of-billions-parameter MoE LLM with ROLLART on in-house mathematical and software-engineering agentic tasks. Prompts and responses reach 12k and 46k tokens, respectively, and the average number of turns per task ranges from 1 to 48 (Figure 15a). This mix exposes both prefill-heavy and decode-heavy behavior. Stragglers persist throughout training: in each step, the maximum response length exceeds 5× the mean and peaks at 9×; the maximum number of environment-interaction turns remains above 40× the mean. These tails explain why production runs need both fast prefill execution and trajectory-level rollout that can absorb slow environments.

The job uses asynchronous training with a 1:5 ratio of training to generation GPUs and an asynchronous bound of one to balance rollout throughput with gradient stability. Still, the longest iteration reaches 1.5 hours (Figure 15b). The bottleneck is the blocking `get_batch` call: after computing log probabilities and gradients, the training stage waits for `SampleBuffer` to collect enough trajectories. This accounts for up to 62% of iteration time as GPU idleness; removing it would ideally reduce training time by 22% (Figure 15b).

Characterization-Driven Optimization. The production trace also shows why coarse-grained characterization is a practical starting point for hardware affinity: task domains are known before training, and their token and turn profiles differ substantially. Using this information, we adjust the training-to-generation resource ratio and tune prefix caching for the MoE architecture. Figure 15c shows that these changes achieve a 1.66× end-to-end speedup over the first 25 steps. Beyond this resource tuning, we apply dedicated optimizations for environment stability and failure recovery.

Optimizing Environment Stability. Across 60 iterations, long environment initialization failures appear in seven iterations. At this scale, Docker-based environments on Kubernetes are sensitive to image-pull failures and network instability during `env.reset`. We therefore use a multi-tiered cache: an internal image registry mirrors external images, and a distributed load-balanced cache between compute nodes and the registry absorbs high-volume requests. During `env.reset`, clients fetch images from cache first. This optimization raises



(a) Average Number of Turns per Task. (b) Step Time Distribution. (c) Accumulated Step Time.

Figure 15: Production-grade agentic RL workload characterization and optimization.

the `env.reset` success rate above 99.99%. It also avoids repeated image-pull retries, keeping over 99.99% of initialization under one minute. After the optimization, no more than ten heavy-tailed initialization events occur among hundreds of thousands of resets; trajectory-level rollout and redundant environments absorb the remaining tails.

System Resilience. RollArt isolates failures across environment, reward, inference, and training workers. Persistent sessions and exponential backoff reduce environment-connection timeouts; Kubernetes manages environment workers, and the serverless platform manages reward workers. If an inference worker fails, RollArt first restarts it on the same GPU; after repeated failure, the worker is removed and its stored trajectories resume on healthy workers. Training-worker failures restart from the latest checkpoint. In a week-long training run, we observed only one failure.

9 Discussion and Related Work

Automating Hardware-Affinity Mapping. RollArt currently requires static, per-task-domain affinity declarations, which is a real limitation when computation profiles shift mid-run. In our production deployment, however, per-domain profiles are stable: turn counts vary widely *across* families but stay bounded *within* each domain across iterations, so coarse `hw_mapping` declarations needed no re-tuning across the week-long 3,000-GPU run. The natural extension is an online profiler integrated with the resource manager: per-domain prefill/decode latency would let RollArt re-route requests when within-domain shifts occur (e.g., a domain alternating between long-observation and long-response prompts) and the same profiling can drive PD-disaggregation ratios (§6.3). Per-iteration optimality is not required: workload profile is dominated by task-domain identity rather than per-step policy variance, so profiling decisions stabilize over a few iterations.

Operational Regime. RollArt targets multi-task agentic RL on heterogeneous, disaggregated infrastructure; it offers limited benefit on homogeneous clusters, where **R1**'s affinity routing collapses to a no-op; on strict on-policy or compute-light RL, where **R4**'s bounded-staleness overlap yields little; and on deployments without elastic compute, where **R3**'s serverless offloading is unrealizable. Trajectory-level rollout (**R2**) and redundant environment rollouts (§7.4) still apply to any agentic RL workload exposed to long-tail environments.

RL Post-Training Systems. Many systems address the systems challenges of RL post-training. Early frameworks [17, 21, 46, 62] adopt appropriate stage and resource mapping to improve utilization. Subsequent systems [5, 6, 10, 15, 18, 27, 43, 45, 67, 68] explore a range of accelerations, including speculative decoding, fusion of pipeline stages, asynchronous or decoupled training to hide latency, and various forms of resource disaggregation. RollArt builds on the disaggregated paradigm and assigns workloads based on hardware affinity.

Resource Disaggregation. Resource disaggregation is adopted in many modern systems to improve utilization [14, 26, 42]. This pattern is prevalent in LLM serving, where systems separate the prefill and decoding phases [20, 37, 38, 51, 52, 66, 69]. Several RL systems apply a similar principle, separating the training and rollout stages across different GPU types [56, 67]. Compared to these approaches, RollArt provides a more general and fine-grained disaggregation model, tailored for the entire lifecycle of multi-task agentic RL.

10 Conclusion

We presented RollArt, a system for multi-task agentic RL training on disaggregated infrastructure. Its design rests on four requirements: hardware-affinity workload mapping (**R1**), trajectory-level asynchronous rollout (**R2**), serverless offloading for stateless reward (**R3**), and bounded-staleness asynchronous training (**R4**). On Qwen3 (8B–32B), RollArt reduces step time by 1.31–2.05× over strengthened synchronous, one-off, and AReaL baselines. A week-long, 3,000-GPU production run training a hundreds-of-billions-parameter MoE model confirms these gains at scale.

Acknowledgment

We thank our shepherd, Shadi Noghabi, and the anonymous reviewers for their valuable comments that help improve the quality of this work. This work was supported in part by the HKUST-Alibaba Joint Laboratory on Big Data and AI, RGC CRF Grant (Ref. #C6015-23G), RGC GRF Grant (Ref. #16217124), and NSFC/RGC CRS Grant (Ref. #CRS_HKUST601/24).

References

[1] Alibaba Cloud. Alibaba Cloud function compute. https://www.alibabacloud.com/en/product/function-compute, 2025.

[2] Alibaba Cloud. Qwen model repos. https://huggingface.co/Qwen, 2025.

[3] Axon-RL. GEM: Generalist environment for multi-task learning. https://github.com/axon-rl/gem, 2025.

[4] Jiaze Chen, Tiantian Fan, Xin Liu, Lingjun Liu, Zhiqi Lin, Mingxuan Wang, Chengyi Wang, Xiangpeng Wei, Wenyuan Xu, Yufeng Yuan, Yu Yue, Lin Yan, Qiying Yu, Xiaochen Zuo, Chi Zhang, Ruofei Zhu, Zhecheng An, Zhihao Bai, Yu Bao, Xingyan Bin, Jiangjie Chen, Feng Chen, Hongmin Chen, Riwei Chen, Liangqiang Chen, Zixin Chen, Jinsong Chen, Siyan Chen, Kaiyuan Chen, Zhi Chen, Jin Chen, Jiecao Chen, Jinxin Chi, Weinan Dai, Ning Dai, Jiahui Dai, Shihan Dou, Yantao Du, Zhengyin Du, Jianhui Duan, Chen Dun, Ting-Han Fan, Jiazhan Feng, Junda Feng, Ziyuan Feng, Yuwei Fu, Wenqi Fu, Hanjie Fu, Hao Ge, Hongyi Guo, Mingji Han, Li Han, Wenhao Hao, Xintong Hao, Qianyu He, Jerry He, Feng He, Wen Heng, Zehua Hong, Qi Hou, Liang Hu, Shengding Hu, Nan Hu, Kai Hua, Qi Huang, Ziyue Huang, Hongzhi Huang, Zihao Huang, Ting Huang, Wenhao Huang, Wei Jia, Bin Jia, Xiaoying Jia, Yuhua Jiang, Haobin Jiang, Ziheng Jiang, Kaihua Jiang, Chengquan Jiang, Jianpeng Jiao, Xiaoran Jin, Xing Jin, Xunhao Lai, Zheng Li, Xiang Li, Liyi Li, Hongkai Li, Zheng Li, Shengxian Wan, Ya Wang, Yunshui Li, Chenggang Li, Niuniu Li, Siyu Li, Xi Li, Xiao Li, Aoyan Li, Yuntao Li, Nianning Liang, and Xinnian Liang. Seed1.5-Thinking: Advancing superb reasoning models with reinforcement learning. *CoRR*, 2025.

[5] Qiaoling Chen, Zijun Liu, Peng Sun, Shenggui Li, Guoteng Wang, Ziming Liu, Yonggang Wen, Siyuan Feng, and Tianwei Zhang. ReSpec: Towards optimizing speculative decoding in reinforcement learning systems. *CoRR*, 2025.

[6] Rongxin Cheng, Kai Zhou, Xingda Wei, Siyuan Liu, Mingcong Han, Mingjing Ai, Yeju Zhou, Baoquan Zhong, Wencong Xiao, Rong Chen, and Haibo Chen. Fast LLM post-training via decoupled and best-of-n speculation. *CoRR*, 2025.

[7] DeepSeek-AI. DeepSeek-R1: Incentivizing reasoning capability in LLMs via reinforcement learning. *CoRR*, 2025.

[8] DeepSeek-AI. DeepSeek-V3.2-Speciale. https://huggingface.co/deepseek-ai/DeepSeek-V3.2-Speciale, 2025.

[9] Farama Foundation. Gymnasium - FrozenLake environment. https://gymnasium.farama.org/environments/toy_text/frozen_lake/, 2024.

[10] Wei Fu, Jiaxuan Gao, Xujie Shen, Chen Zhu, Zhiyu Mei, Chuyi He, Shusheng Xu, Guo Wei, Jun Mei, Jiashu Wang, Tongkai Yang, Binhang Yuan, and Yi Wu. AReaL: A large-scale asynchronous reinforcement learning system for language reasoning. *CoRR*, 2025.

[11] Yao Fu, Leyang Xue, Yeqi Huang, Andrei-Octavian Brabete, Dmitrii Ustiugov, Yuvraj Patel, and Luo Mai. ServerlessLLM: Low-Latency serverless inference for large language models. In *18th USENIX Symposium on Operating Systems Design and Implementation, OSDI*, 2024.

[12] Wei Gao, Yuheng Zhao, Dakai An, Tianyuan Wu, Lunxi Cao, Shaopan Xiong, Ju Huang, Weixun Wang, Siran Yang, Wenbo Su, Jiamang Wang, Lin Qu, Bo Zheng, and Wei Wang. RollPacker: Taming long-tail rollouts for RL post-training with tail batching. In *23rd USENIX Symposium on Networked Systems Design and Implementation, NSDI*, 2026.

[13] Daya Guo, Qihao Zhu, Dejian Yang, Zhenda Xie, Kai Dong, Wentao Zhang, Guanting Chen, Xiao Bi, Y. Wu, Y. K. Li, Fuli Luo, Yingfei Xiong, and Wenfeng Liang. DeepSeek-Coder: When the large language model meets programming - the rise of code intelligence. *CoRR*, 2024.

[14] Zhiyuan Guo, Zijian He, and Yiying Zhang. Mira: A program-behavior-guided far memory system. In *Proceedings of the 29th Symposium on Operating Systems Principles, SOSP*, 2023.

[15] Zhenyu Han, Ansheng You, Haibo Wang, Kui Luo, Guang Yang, Wenqi Shi, Menglong Chen, Sicheng Zhang, Zeshun Lan, Chunshi Deng, Huazhong Ji, Wenjie Liu, Yu Huang, Yixiang Zhang, Chenyi Pan, Jing Wang, Xin Huang, Chunsheng Li, and Jianping Wu. AsyncFlow: An asynchronous streaming RL framework for efficient LLM post-training. *CoRR*, 2025.

[16] Bingguang Hao, Maolin Wang, Zengzhuang Xu, Yicheng Chen, Cunyin Peng, Jinjie GU, and Chenyi Zhuang. Exploring superior function calls via reinforcement learning. *CoRR*, 2025.

[17] Eric Harper, Somshubra Majumdar, Oleksii Kuchaiev, Jason Li, Yang Zhang, Evelina Bakhturina, Vahid Noroozi, Sandeep Subramanian, Nithin Koluguri, Jocelyn Huang, Fei Jia, Jagadeesh Balam, Xuesong Yang, Micha Livne, Yi Dong, Sean Naren, and Boris Ginsburg. NeMo: A toolkit for conversational AI and large language models. https://github.com/NVIDIA/NeMo, 2025.

[18] Jingkai He, Tianjian Li, Erhu Feng, Dong Du, Qian Liu, Tao Liu, Yubin Xia, and Haibo Chen. History doesn't repeat itself but rollouts rhyme: Accelerating reinforcement learning with rhymerl. In *Proceedings of the 31st ACM International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS*, 2026.

[19] Zhiwei He, Tian Liang, Jiahao Xu, Qiuzhi Liu, Xingyu Chen, Yue Wang, Linfeng Song, Dian Yu, Zhenwen Liang, Wenxuan Wang, Zhuosheng Zhang, Rui Wang, Zhaopeng Tu, Haitao Mi, and Dong Yu. DeepMath-103K: A large-scale, challenging, decontaminated, and verifiable mathematical dataset for advancing reasoning. *CoRR*, 2025.

[20] Cunchen Hu, Heyang Huang, Liangliang Xu, Xusheng Chen, Jiang Xu, Shuang Chen, Hao Feng, Chenxi Wang, Sa Wang, Yungang Bao, Ninghui Sun, and Yizhou Shan. Inference without interference: Disaggregate LLM inference for mixed downstream workloads. *CoRR*, 2024.

[21] Jian Hu, Xibin Wu, Weixun Wang, Xianyu, Dehao Zhang, and Yu Cao. OpenRLHF: An easy-to-use, scalable and high-performance RLHF framework. *CoRR*, 2024.

[22] Pengcheng Jiang, Jiacheng Lin, Lang Cao, Runchu Tian, SeongKu Kang, Zifeng Wang, Jimeng Sun, and Jiawei Han. DeepRetrieval: Hacking real search engines and retrievers with large language models via reinforcement learning. *CoRR*, 2025.

[23] Carlos E. Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. SWE-bench: Can language models resolve real-world GitHub issues? *CoRR*, 2024.

[24] Bowen Jin, Hansi Zeng, Zhenrui Yue, Jinsung Yoon, Sercan Arik, Dong Wang, Hamed Zamani, and Jiawei Han. Search-R1: Training LLMs to reason and leverage search engines with reinforcement learning. *CoRR*, 2025.

[25] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph E. Gonzalez, Hao Zhang, and Ion Stoica. Efficient memory management for large language model serving with PagedAttention. In *Proceedings of the 29th Symposium on Operating Systems Principles, SOSP*, 2023.

[26] Kevin Lim, Jichuan Chang, Trevor Mudge, Parthasarathy Ranganathan, Steven K Reinhardt, and Thomas F Wenisch. Disaggregated memory for expansion and sharing in blade servers. In *Proceedings of the International Symposium on Computer Architecture, ISCA*, 2009.

[27] Bingshuai Liu, Ante Wang, Zijun Min, Liang Yao, Haibo Zhang, Yang Liu, Anxiang Zeng, and Jinsong Su. SPEC-RL: Accelerating on-policy reinforcement learning via speculative rollouts. *CoRR*, 2025.

[28] Yuhang Liu, Pengxiang Li, Congkai Xie, Xavier Hu, Xiaotian Han, Shengyu Zhang, Hongxia Yang, and Fei Wu. InfiGUI-R1: Advancing multimodal GUI agents from reactive actors to deliberative reasoners. *CoRR*, 2025.

[29] Han Lu, Zichen Liu, Shaopan Xiong, Yancheng He, Wei Gao, Yanan Wu, Weixun Wang, Jiashun Liu, Yang Li, Haizhou Zhao, Ju Huang, Siran Yang, Xiaoyang Li, Yijia Luo, Zihe Liu, Ling Pan, Junchi Yan, Wei Wang, Wenbo Su, Jiamang Wang, Lin Qu, and Bo Zheng. Part II: ROLL flash - accelerating RLVR and agentic training with asynchrony. *CoRR*, 2025.

[30] Zhengxi Lu, Yuxiang Chai, Yaxuan Guo, Xi Yin, Liang Liu, Hao Wang, Han Xiao, Shuai Ren, Guanjing Xiong, and Hongsheng Li. UI-R1: Enhancing efficient action prediction of GUI agents by reinforcement learning. *CoRR*, 2025.

[31] Michael Luo, Naman Jain, Jaskirat Singh, Sijun Tan, Ameen Patel, Qingyang Wu, Alpay Ariyak, Colin Cai, Tarun Venkat, Shang Zhu, Ben Athiwaratkun, Manan Roongta, Ce Zhang, Li Erran Li, Raluca Ada Popa, Koushik Sen, and Ion Stoica. DeepSWE: Training a state-of-the-art coding agent from scratch by scaling RL, 2025.

[32] Michael Luo, Sijun Tan, Justin Wong, Xiaoxiang Shi, William Y. Tang, Manan Roongta, Colin Cai, Jeffrey Luo, Li Erran Li, Raluca Ada Popa, and Ion Stoica. DeepScaleR: Surpassing O1-Preview with a 1.5b model by scaling RL. https://pretty-radio-b75.notion.site/DeepScaleR-Surpassing-O1-Preview-with-a-1-5B-Model-by-Scaling-RL-19681902c1468005bed8ca303013a4e2, 2025.

[33] Run Luo, Lu Wang, Wanwei He, and Xiaobo Xia. GUI-R1: A generalist R1-style vision-language action model for GUI agents. *CoRR*, 2025.

[34] Philipp Moritz, Robert Nishihara, Stephanie Wang, Alexey Tumanov, Richard Liaw, Eric Liang, Melih Elibol, Zongheng Yang, William Paul, Michael I. Jordan, and Ion Stoica. Ray: A distributed framework for emerging AI applications. In *13th USENIX Symposium on Operating Systems Design and Implementation, OSDI*, 2018.

[35] NVIDIA Corporation. NVIDIA collective communication library (NCCL). https://github.com/NVIDIA/nccl, 2025.

[36] OpenAI. Introducing OpenAI o3 and o4-mini. https://openai.com/index/introducing-o3-and-o4-mini/, 2024.

[37] Pratyush Patel, Esha Choukse, Chaojie Zhang, Aashaka Shah, Íñigo Goiri, Saeed Maleki, and Ricardo Bianchini. Splitwise: Efficient generative LLM inference using phase splitting. In *51st ACM/IEEE Annual International Symposium on Computer Architecture, ISCA*, 2024.

[38] Ruoyu Qin, Zheming Li, Weiran He, Mingxing Zhang, Yongwei Wu, Weimin Zheng, and Xinran Xu. Mooncake: Kimi's KVCache-centric architecture for LLM serving. *CoRR*, 2024.

[39] Alibaba Qoder. Qoder: Agentic coding platform for real software. https://qoder.com/en, 2025.

[40] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *CoRR*, 2017.

[41] SGLang Team. SGLang: Fast serving framework for large language models. https://github.com/sgl-project/sglang, 2025.

[42] Yizhou Shan, Yutong Huang, Yilun Chen, and Yiying Zhang. LegoOS: A disseminated, distributed OS for hardware resource disaggregation. In *13th USENIX Symposium on Operating Systems Design and Implementation, OSDI*, 2018.

[43] Zelei Shao, Vikranth Srivatsa, Sanjana Srivastava, Qingyang Wu, Alpay Ariyak, Xiaoxia Wu, Ameen Patel, Jue Wang, Percy Liang, Tri Dao, Ce Zhang, Yiying Zhang, Ben Athiwaratkun, Chenfeng Xu, and Junxiong Wang. Beat the long tail: Distribution-aware speculative decoding for RL training. *CoRR*, 2025.

[44] Zhihong Shao, Peiyi Wang, Qihao Zhu, Runxin Xu, Junxiao Song, Mingchuan Zhang, Y. K. Li, Y. Wu, and Daya Guo. DeepSeekMath: Pushing the limits of mathematical reasoning in open language models. *CoRR*, 2024.

[45] Guangming Sheng, Yuxuan Tong, Borui Wan, Wang Zhang, Chaobo Jia, Xibin Wu, Yuqi Wu, Xiang Li, Chi Zhang, Yanghua Peng, Haibin Lin, Xin Liu, and Chuan Wu. Laminar: A scalable asynchronous RL post-training framework. In *Proceedings of the 21st European Conference on Computer Systems, EuroSys*, 2026.

[46] Guangming Sheng, Chi Zhang, Zilingfeng Ye, Xibin Wu, Wang Zhang, Ru Zhang, Yanghua Peng, Haibin Lin, and Chuan Wu. verl: Volcano engine reinforcement learning for LLM. https://github.com/volcengine/verl, 2024.

[47] Guangming Sheng, Chi Zhang, Zilingfeng Ye, Xibin Wu, Wang Zhang, Ru Zhang, Yanghua Peng, Haibin Lin, and Chuan Wu. HybridFlow: A flexible and efficient RLHF framework. In *Proceedings of the Twentieth European Conference on Computer Systems, EuroSys*, 2025.

[48] Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. Megatron-LM: Training multi-billion parameter language models using model parallelism. *CoRR*, 2019.

[49] Guijin Son, Hyunwoo Ko, Hoyoung Lee, Yewon Kim, and Seunghyeok Hong. LLM-as-a-Judge & reward model: What they can and cannot do. *CoRR*, 2024.

[50] Huatong Song, Jinhao Jiang, Yingqian Min, Jie Chen, Zhipeng Chen, Wayne Xin Zhao, Lei Fang, and Ji-Rong Wen. R1-Searcher: Incentivizing the search capability in LLMs via reinforcement learning. *CoRR*, 2025.

[51] StepFun. Step-3 is large yet affordable: Model-system co-design for cost-effective decoding. *CoRR*, 2025.

[52] Foteini Strati, Sara McAllister, Amar Phanishayee, Jakub Tarnawski, and Ana Klimovic. Déjàvu: KV-cache streaming for fast, fault-tolerant generative LLM serving. In *Forty-first International Conference on Machine Learning, ICML*, 2024.

[53] Sijun Tan, Michael Luo, Colin Cai, Tarun Venkat, Kyle Montgomery, Aaron Hao, Tianhao Wu, Arnav Balyan, Manan Roongta, Chenguang Wang, Li Erran Li, Raluca Ada Popa, and Ion Stoica. rLLM: A framework for post-training language agents, 2025.

[54] Kimi Team, Yifan Bai, Yiping Bao, Guanduo Chen, Jiahao Chen, Ningxin Chen, Ruijue Chen, Yanru Chen, Yuankun Chen, Yutian Chen, Zhuofu Chen, Jialei Cui, Hao Ding, Mengnan Dong, Angang Du, Chenzhuang Du, Dikang Du, Yulun Du, Yu Fan, Yichen Feng, Kelin Fu, Bofei Gao, Hongcheng Gao, Peizhong Gao, Tong Gao, Xinran Gu, Longyu Guan, Haiqing Guo, Jianhang Guo, Hao Hu, Xiaoru Hao, Tianhong He, Weiran He, Wenyang He, Chao Hong, Yangyang Hu, Zhenxing Hu, Weixiao Huang, Zhiqi Huang, Zihao Huang, Tao Jiang, Zhejun Jiang, Xinyi Jin, Yongsheng Kang, Guokun Lai, Cheng Li, Fang Li, Haoyang Li, Ming Li, Wentao Li, Yanhao Li, Yiwei Li, Zhaowei Li, Zheming Li, Hongzhan Lin, Xiaohan Lin, Zongyu Lin, Chengyin Liu, Chenyu Liu, Hongzhang Liu, Jingyuan Liu, Junqi Liu, Liang Liu, Shaowei Liu, T. Y. Liu, Tianwei Liu, Weizhou Liu, Yangyang Liu, Yibo Liu, Yiping Liu, Yue Liu, Zhengying Liu, Enzhe Lu, Lijun Lu, Shengling Ma, Xinyu Ma, Yingwei Ma, Shaoguang Mao, Jie Mei, Xin Men, Yibo Miao, Siyuan Pan, Yebo Peng, Ruoyu Qin, Bowen Qu, Zeyu Shang, Lidong Shi, Shengyuan Shi, Feifan Song, Jianlin Su, Zhengyuan Su,

Xinjie Sun, Flood Sung, Heyi Tang, Jiawen Tao, Qifeng Teng, Chensi Wang, Dinglu Wang, Feng Wang, Haiming Wang, Jianzhou Wang, Jiaxing Wang, Jinhong Wang, Shengjie Wang, Shuyi Wang, Yao Wang, Yejie Wang, Yiqin Wang, Yuxin Wang, Yuzhi Wang, Zhaoji Wang, Zhengtao Wang, Zhexu Wang, Chu Wei, Qianqian Wei, Wenhao Wu, Xingzhe Wu, Yuxin Wu, Chenjun Xiao, Xiaotong Xie, Weimin Xiong, Boyu Xu, Jing Xu, Jinjing Xu, L. H. Xu, Lin Xu, Suting Xu, Weixin Xu, Xinran Xu, Yangchuan Xu, Ziyao Xu, Junjie Yan, Yuzi Yan, Xiaofei Yang, Ying Yang, Zhen Yang, Zhilin Yang, Zonghan Yang, Haotian Yao, Xingcheng Yao, Wenjie Ye, Zhuorui Ye, Bohong Yin, Longhui Yu, Enming Yuan, Hongbang Yuan, Mengjie Yuan, Haobing Zhan, Dehao Zhang, Hao Zhang, Wanlu Zhang, Xiaobin Zhang, Yangkun Zhang, Yizhi Zhang, Yongting Zhang, Yu Zhang, Yutao Zhang, Yutong Zhang, Zheng Zhang, Haotian Zhao, Yikai Zhao, Huabin Zheng, Shaojie Zheng, Jianren Zhou, Xinyu Zhou, Zaida Zhou, Zhen Zhu, Weiyu Zhuang, and Xinxing Zu. Kimi K2: Open agentic intelligence. *CoRR*, 2025.

[55] The Kubernetes Authors. Kubernetes. https://kubernetes.io, 2025.

[56] Jinghui Wang, Shaojie Wang, Yinghan Cui, Xuxing Chen, Chao Wang, Xiaojiang Zhang, Minglei Zhang, Jiarong Zhang, Wenhao Zhuang, Yuchen Cao, Wankang Bao, Haimo Li, Zheng Lin, Huiming Wang, Haoyang Huang, Zongxian Feng, Zizheng Zhan, Ken Deng, Wen Xiang, Huaixi Tang, Kun Wu, Mengtong Li, Mengfei Xie, Junyi Peng, Haotian Zhang, Bin Chen, and Bing Yu. SeamlessFlow: A trainer agent isolation RL framework achieving bubble-free pipelines via tag scheduling. *CoRR*, 2025.

[57] Weixun Wang, Shaopan Xiong, Gengru Chen, Wei Gao, Sheng Guo, Yancheng He, Ju Huang, Jiaheng Liu, Zhendong Li, Xiaoyang Li, Zichen Liu, Haizhou Zhao, Dakai An, Lunxi Cao, Qiyang Cao, Wanxi Deng, Feilei Du, Yiliang Gu, Jiahe Li, Xiang Li, Mingjie Liu, Yijia Luo, Zihe Liu, Yadao Wang, Pei Wang, Tianyuan Wu, Yanan Wu, Yuheng Zhao, Shuaibing Zhao, Jin Yang, Siran Yang, Yingshui Tan, Huimin Yi, Yuchi Xu, Yujin Yuan, Xingyao Zhang, Lin Qu, Wenbo Su, Wei Wang, Jiamang Wang, and Bo Zheng. Reinforcement learning optimization for large-scale learning: An efficient and user-friendly scaling library. *CoRR*, 2025.

[58] Junde Wu, Jiayuan Zhu, Yuyuan Liu, Min Xu, and Yueming Jin. Agentic reasoning: A streamlined framework for enhancing LLM reasoning with agentic tools. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), ACL*, 2025.

[59] Bingquan Xia, Bowen Shen, Cici, Dawei Zhu, Di Zhang, Gang Wang, Hailin Zhang, Huaqiu Liu, Jiebao Xiao, Jinhao Dong, Liang Zhao, Peidian Li, Peng Wang, Shihua Yu, Shimao Chen, Weikun Wang, Wenhan Ma, Xiangwei Deng, Yi Huang, Yifan Song, Zihan Jiang, Bowen Ye, Can Cai, Chenhong He, Dong Zhang, Duo Zhang, Guoan Wang, Hao Tian, Haochen Zhao, Heng Qu, Hongshen Xu, Jun Shi, Kainan Bao, Kai Fang, Kang Zhou, Kangyang Zhou, Lei Li, Menghang Zhu, Nuo Chen, Qiantong Wang, Shaohui Liu, Shicheng Li, Shuhao Gu, Shuhuai Ren, Shuo Liu, Sirui Deng, Weiji Zhuang, Weiwei Lv, Wenyu Yang, Xin Zhang, Xing Yong, Xing Zhang, Xingchen Song, Xinzhe Xu, Xu Wang, Yihan Yan, Yu Tu, Yuanyuan Tian, Yudong Wang, Yue Yu, Zhenru Lin, Zhichao Song, and Zihao Yue. MiMo: Unlocking the reasoning potential of language model - from pretraining to posttraining. *CoRR*, 2025.

[60] An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengen Huang, Chenxu Lv, Chujie Zheng, Dayiheng Liu, Fan Zhou, Fei Huang, Feng Hu, Hao Ge, Haoran Wei, Huan Lin, Jialong Tang, Jian Yang, Jianhong Tu, Jianwei Zhang, Jianxin Yang, Jiaxi Yang, Jing Zhou, Jingren Zhou, Junyang Lin, Kai Dang, Keqin Bao, Kexin Yang, Le Yu, Lianghao Deng, Mei Li, Mingfeng Xue, Mingze Li, Pei Zhang, Peng Wang, Qin Zhu, Rui Men, Ruize Gao, Shixuan Liu, Shuang Luo, Tianhao Li, Tianyi Tang, Wenbiao Yin, Xingzhang Ren, Xinyu Wang, Xinyu Zhang, Xuancheng Ren, Yang Fan, Yang Su, Yichang Zhang, Yinger Zhang, Yu Wan, Yuqiong Liu, Zekun Wang, Zeyu Cui, Zhenru Zhang, Zhipeng Zhou, and Zihan Qiu. Qwen3 technical report. *CoRR*, 2025.

[61] Shunyu Yao, Howard Chen, John Yang, and Karthik Narasimhan. WebShop: Towards scalable real-world web interaction with grounded language agents. *CoRR*, 2023.

[62] Zhewei Yao, Reza Yazdani Aminabadi, Olatunji Ruwase, Samyam Rajbhandari, Xiaoxia Wu, Ammar Ahmad Awan, Jeff Rasley, Minjia Zhang, Conglong Li, Connor Holmes, Zhongzhu Zhou, Michael Wyatt, Molly Smith, Lev Kurilenko, Heyang Qin, Masahiro Tanaka, Shuai Che, Shuaiwen Leon Song, and Yuxiong He. DeepSpeed-Chat: Easy, fast and affordable RLHF training of ChatGPT-like models at all scales. *CoRR*, 2023.

[63] Chengyue Yu, Siyuan Lu, Chenyi Zhuang, Dong Wang, Qintong Wu, Zongyue Li, Runsheng Gan, Chunfeng Wang, Siqi Hou, Gaochi Huang, Wenlong Yan, Lifeng Hong, Aohui Xue, Yanfeng Wang, Jinjie Gu, David Tsai, and Tao Lin. AWorld: Orchestrating the training recipe for agentic AI. *CoRR*, 2025.

[64] Minchen Yu, Ao Wang, Dong Chen, Haoxuan Yu, Xiaonan Luo, Zhuohao Li, Wei Wang, Ruichuan Chen, Dapeng Nie, Haoran Yang, and Yu Ding. Torpor: GPU-enabled serverless computing for low-latency, resource-efficient inference. In *2025 USENIX Annual Technical Conference, USENIX ATC*, 2025.

[65] Dingyan Zhang, Haotian Wang, Yang Liu, Xingda Wei, Yizhou Shan, Rong Chen, and Haibo Chen. Blitzscale: Fast and live large model autoscaling with O(1) host caching. In *19th USENIX Symposium on Operating Systems Design and Implementation, OSDI*, 2025.

[66] Yinmin Zhong, Shengyu Liu, Junda Chen, Jianbo Hu, Yibo Zhu, Xuanzhe Liu, Xin Jin, and Hao Zhang. DistServe: Disaggregating prefill and decoding for goodput-optimized large language model serving. In *18th USENIX Symposium on Operating Systems Design and Implementation, OSDI*, 2024.

[67] Yinmin Zhong, Zili Zhang, Xiaoniu Song, Hanpeng Hu, Chao Jin, Bingyang Wu, Nuo Chen, Yukun Chen, Yu Zhou, Changyi Wan, Hongyu Zhou, Yimin Jiang, Yibo Zhu, and Daxin Jiang. StreamRL: Scalable, heterogeneous, and elastic RL for LLMs with disaggregated stream generation. *CoRR*, 2025.

[68] Yinmin Zhong, Zili Zhang, Bingyang Wu, Shengyu Liu, Yukun Chen, Changyi Wan, Hanpeng Hu, Lei Xia, Ranchen Ming, Yibo Zhu, and Xin Jin. Optimizing RLHF training for large language models with stage fusion. In *22nd USENIX Symposium on Networked Systems Design and Implementation, NSDI*, 2025.

[69] Ruidong Zhu, Ziheng Jiang, Chao Jin, Peng Wu, Cesar A. Stuardo, Dongyang Wang, Xinlei Zhang, Huaping Zhou, Haoran Wei, Yang Cheng, Jianzhe Xiao, Xinyi Zhang, Lingjun Liu, Haibin Lin, Li-Wen Chang, Jianxi Ye, Xiao Yu, Xuanzhe Liu, Xin Jin, and Xin Liu. MegaScale-Infer: Efficient mixture-of-experts model serving with disaggregated expert parallelism. In *Proceedings of the ACM SIGCOMM 2025 Conference, SIGCOMM*, 2025.

[70] Zilin Zhu, Chengxing Xie, Xin Lv, and slime Contributors. slime: An LLM post-training framework for RL scaling. https://github.com/THUDM/slime, 2025.

Seer: Online Context Learning for Fast Synchronous LLM Reinforcement Learning

Ruoyu Qin[†♢] Weiran He[†] Weixiao Huang[†] Yangkun Zhang[†] Yikai Zhao[†]
Bo Pang[†] Xinran Xu[†] Yingdi Shan[♢] Yongwei Wu[♢] Mingxing Zhang[♢1]
[†]Moonshot AI [♢]Tsinghua University

Abstract

Reinforcement Learning (RL) has emerged as a critical technique for advancing modern Large Language Models (LLMs), yet existing synchronous RL systems face severe performance bottlenecks. The rollout phase, which dominates end-to-end iteration time, suffers from substantial long-tail latency and poor resource utilization due to inherent workload imbalance. We present SEER, a novel context learning RL system that addresses these challenges through a key observation: requests sharing the same prompt exhibit strong similarities in output lengths and response patterns. Leveraging this insight, SEER introduces three coordinated techniques: (1) divided rollout for dynamic load balancing, (2) context-aware scheduling to mitigate long-tail request delays, and (3) adaptive grouped speculative decoding to accelerate generation. These mechanisms work in concert to markedly reduce long-tail latency and improve resource efficiency during rollout. Evaluations on production-grade RL workloads demonstrate that SEER achieves up to 2.04× end-to-end rollout throughput improvement compared to the state-of-the-art synchronous RL systems, while notably reducing long-tail latency by 72–94%.

1 Introduction

Reinforcement Learning (RL) has become a cornerstone in the development of state-of-the-art Large Language Models (LLMs), enabling significant breakthroughs in complex reasoning and problem-solving capabilities [9, 38, 39]. The iterative RL training process alternates between a *rollout phase* for data generation and a *training phase* for updating model parameters. However, the rollout phase consistently emerges as the dominant bottleneck, consuming approximately 80% of the total iteration time (see Table 1). Therefore, improving the efficiency of the rollout phase represents one of the most pressing challenges in modern LLM development.

The primary challenge in RL rollout arises from severe resource inefficiency driven by the increasing demand for long-generation capabilities, especially in tasks requiring complex chain-of-thought (CoT) reasoning. These workloads create two fundamental bottlenecks. First, long-generation requests exhibit highly **unpredictable and rapidly increasing memory footprints**: a CoT request may begin with only a few hundred megabytes of KVCache usage but expand to tens of gigabytes as decoding progresses. This volatility forces the system to shrink batch sizes dynamically or to preempt running requests; both outcomes reduce hardware efficiency, and preemptions are particularly costly because they trigger expensive re-prefills, ultimately degrading rollout throughput.

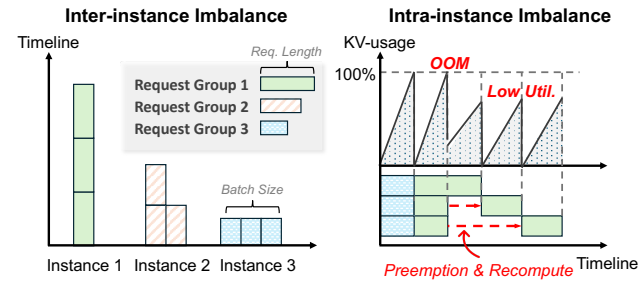
Table 1: Time distribution across RL training phases for different workloads. Detailed configurations given in §4.1.

	Rollout	Training	Weight Update
Moonlight [24]	84%	14%	2%
Qwen2-VL-72B [43]	63%	31%	6%
Kimi-K2 [38]	87%	10%	3%

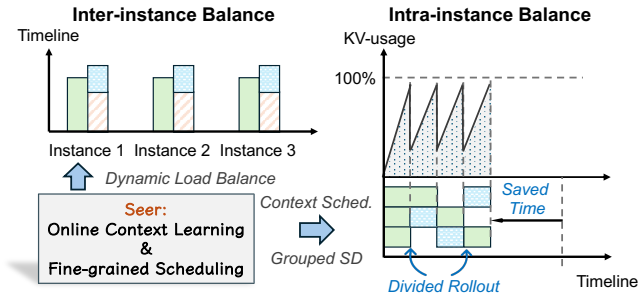
Second, long-generation requests produce a **heavy-tailed distribution of output lengths**, resulting in pronounced load imbalance. Toward the end of a rollout iteration, only a small number of disproportionately long-running requests remain active, leaving most accelerators underutilized. The inefficiency is so severe that attempting to use idle nodes to accelerate these long-tail requests often yields little benefit and can even harm performance due to increased inter-node communication overhead.

To improve hardware utilization, recent works have explored asynchronous rollout systems [5, 10, 11, 29, 33, 51], which overlap the rollout and training phases. While this approach can reduce end-to-end iteration time, it comes at the cost of algorithmic fidelity. By nature, these systems introduce a degree of off-policy learning, as data generated with model parameters from step i may be used to train the model at step $i+1$ or beyond. Furthermore, asynchronous or non-strictly synchronous [8, 53] RL systems often suffer from distributional skew, where faster-to-generate short samples

[1] Corresponding to zhang_mingxing@mail.tsinghua.edu.cn.



(a) Group-level rollout.



(b) SEER's rollout.

Figure 1: Challenges and SEER's solution for long-generation rollout.

disproportionately populate early training batches [40]. These off-policy effects can diminish final model performance and complicate debugging and reproducibility [49]. Consequently, synchronous (or "on-policy") rollout remains critical in many settings for ensuring methodical evaluation, reproducibility, and strict adherence to the underlying algorithm's assumptions. This paper, therefore, focuses on optimizing the synchronous case, although we note that our techniques can also be adapted to improve asynchronous settings.

Speculative decoding [17] (SD) presents another promising direction for accelerating memory-bound generation, as its parallel verification mechanism can utilize more computational resources to accelerate individual requests. However, conventional SD methods struggle to simultaneously achieve high draft accuracy and low draft overhead in RL settings, where both the request workload and the target model itself evolve dynamically. This motivates the need for adaptive SD techniques tailored to the unique characteristics of RL rollout.

To address these challenges, we introduce SEER, a novel system for synchronous RL rollout that dynamically exploits contextual information within the workload to maximize resource efficiency. SEER is built upon a key observation: Group Relative Policy Optimization (GRPO) [31] and its growing family of variants (DAPO [48], GSPO [50], Dr.GRPO [25], etc.), which have become the dominant paradigm for LLM post-training, generate G (typically 8–16) responses per prompt, and **responses within a group tend to exhibit similar length profiles and recurring local token patterns**, which represent a rich structure that existing schedulers and inference engines leave untapped. For instance, generating an early "probe" response per prompt provides a strong, online estimator of that prompt group's remaining work (i.e., expected output length and KVCache footprint), enabling more informed scheduling decisions than static heuristics. SEER leverages this latent intra-group context to make three key contributions:

1) Divided Rollout with Global KVCache: SEER departs from conventional group-level scheduling, where all requests within a prompt group are treated as a single, inseparable unit. This traditional approach causes pronounced inter-instance and intra-instance load imbalance. As illustrated in Figure 1, SEER instead performs *Divided Rollout*, splitting each group not only into G independent requests but further into smaller chunks that are scheduled incrementally. This fine-grained decomposition lets the scheduler pack many short requests together early in rollout to fully utilize VRAM, and later, when long requests dominate, adjust concurrency based on KVCache budgets to avoid preemptions. SEER's global scheduler continuously monitors KVCache usage across instances and can migrate a request to a less-loaded worker when its next chunk is scheduled. This migration is efficient because SEER uses a global KVCache pool adapted from Mooncake [30] and shared across all instances, eliminating the cost of prefill recomputation.

2) Context-Aware Scheduling: SEER leverages a *"speculative request"* from each GRPO group to estimate that group's remaining workload—specifically its likely generation length and KVCache footprint. These lightweight online estimates allow SEER to approximate a longest-job-first scheduling policy that pairs long requests with short ones to maintain dense batches throughout rollout. Experiments in §4.5.1 show that this approach significantly reduces the time spent in the long-tail phase by 89%.

3) Adaptive Grouped Speculative Decoding: To leverage speculative decoding for rollout acceleration while overcoming the acceptance rate collapse of traditional SD, SEER introduces a context-learning-based speculative mechanism. SEER deploys a Distributed Grouped Draft Server (DGDS) that maintains a Compressed Suffix Tree [45] (CST) for each group, aggregating token sequences from all requests within the same group. This approach creates a highly accurate, dynamic "draft model" that is inherently synchronized with the target model. Additionally, DGDS introduces a Marginal-Benefit-Aware Adaptive Speculation policy that balances overall throughput with the latency of high-priority requests. Our ablation experiments in §4.5.2 demonstrate the effectiveness of the SD components. Our adaptive grouped speculative decoding outperforms multiple SD strategies, achieving up to 1.3× performance improvement. Compared to vanilla CST-based SD without group context, it increases the mean acceptance length by 0.22.

Beyond RL rollout, the components and design principles

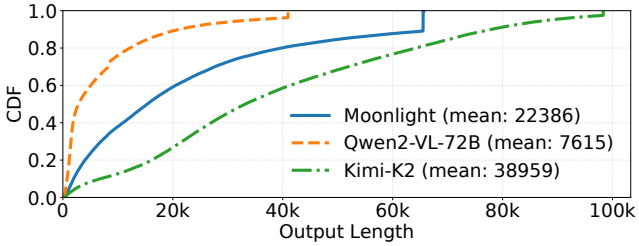
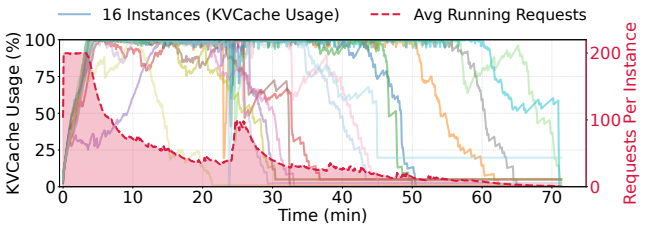
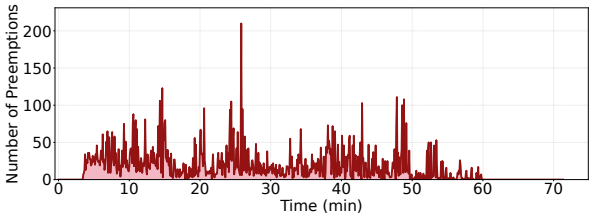


Figure 2: Distribution of output lengths during rollout across three reasoning tasks.



(a) KVCache utilization and average number of running requests across instances.



(b) Request preemption count.

Figure 3: KVCache utilization, number of running requests, and preemption count during a synchronous rollout phase of the Qwen2-VL-72B task. In the early stage of rollout, insufficient KVCache capacity causes frequent request preemptions; in the later stage, a small number of extremely long request groups contribute to a long-tail period that accounts for nearly half of the total rollout time.

of SEER are broadly applicable to other large-scale parallel generation workloads. Divided Rollout addresses KVCache bloat and load imbalance that arise in any high-concurrency, long-form generation serving scenario. Similarly, Context-Aware Scheduling and Adaptive Grouped Speculative Decoding generalize to any multi-sample setting or generation task whose requests exhibit recurring structural similarities.

We have implemented SEER and conducted an extensive evaluation on 256 H800 GPUs across multiple production-grade RL workloads, encompassing models ranging from tens of billions to over one trillion parameters. Our experiments demonstrate that SEER consistently improves end-to-end rollout throughput by 44–104% and cuts long-tail latency by 72–94% relative to a highly optimized synchronous baseline. Through detailed ablations, we isolate the contributions of divided rollout, context-aware scheduling, and adaptive grouped speculative decoding, showing that each component provides substantial and complementary gains. We further compare SEER against state-of-the-art rollout optimization approaches, including a non-strictly synchronous system [53], as well as multiple speculative decoding strategies [17, 22, 28]. We find that SEER achieves the best overall performance while preserving strict on-policy training semantics.

2 Background and Motivation

Reinforcement learning (RL) for LLMs progresses through a repeated iterative loop. In each iteration, the model generates responses for a batch of prompts (rollout), evaluators assign quality scores to those responses (reward computation), these scores are transformed into supervision signals (experience construction), and the model is updated accordingly (training). The updated parameters are then distributed to inference workers to initiate the next iteration (weight update). A series of recent systems aim to accelerate this loop by improving coordination across stages. veRL [34] colocates rollout and training to reduce data movement, RLHFuse [52] overlaps reward computation with experience construction, and Kimi-K2 [38] further optimizes checkpoint conversion and weight distribution. However, Table 1 makes clear that such non-rollout phase optimizations are insufficient: across three representative production workloads, rollout alone occupies 63–87% of total iteration time, overwhelmingly dominating all other stages.

This imbalance is structural and rooted in the nature of training reasoning models. RL increasingly relies on chain-of-thought (CoT) supervision, which encourages models to produce longer and more detailed reasoning traces. As a result, rollout exhibits two defining characteristics: long average output lengths and extremely high variance across requests. Figure 2 shows that generations range from a few hundred tokens to as many as 96k tokens. Long average outputs exert heavy pressure on memory because KVCache consumption scales with request length, while the extreme variance produces severe long-tail effects: a small number of exceptionally long requests monopolize GPU resources near the end of rollout. Figure 3 illustrates how these properties cause substantial resource underutilization and imbalance, motivating the two challenges discussed next.

2.1 Challenge #1: Request Scheduling Dilemma

During the rollout process of long-CoT models, the KVCache memory of requests undergoes dramatic changes, from negligible in the early phase to several gigabytes per request in the final stage. This varying memory consumption creates

a dilemma in concurrency management. If concurrency is not controlled, the increasing request length leads to memory exhaustion, resulting in massive **request preemption**, where the KVCache of some running requests is evicted to allow a few requests to complete inference. This introduces huge overhead for KVCache recomputation. Conversely, if concurrency is controlled to avoid preemption, requests in their early generation phase (with only a few hundred tokens) may occupy the entire memory space for hundreds of seconds, leading to severe **resource underutilization**. Furthermore, this memory footprint is amplified by a factor of G in GRPO-like algorithms, where G responses are generated for each prompt within the same group, further exacerbating the inter-instance imbalance.

To address the inter-instance imbalance caused by group-level scheduling, a concurrent work Roll Flash [26] proposes *prompt replication*, which splits request groups into independent requests for separate scheduling, thereby resolving the amplified imbalance caused by scheduling entire request groups. However, this does not address the dilemma of *intra-instance* concurrent scheduling, and still suffers from uneven request distribution across instances, leading to inter-instance imbalance. The root cause of these issues is treating individual (or groups of) requests as indivisible monolithic units, even though rollout scenarios impose no strict latency constraints on individual requests. This persistent scheduling dilemma motivates our design of *divided rollout* (detailed in §3.2), which schedules requests at the chunk level with little overhead, enabling fine-grained scheduling and dynamic load balancing.

2.2 Challenge #2: Severe Long-tail Effect

The long-tail problem is another critical issue in rollout, widely noted in prior work [8, 11, 53]. As shown in Figure 3a, the long-tail phase can account for nearly 50% of the total rollout time. This phenomenon arises from two factors: (1) In GRPO-like algorithms, requests within the same group tend to have similar lengths. Groups with extremely long average lengths form "monolithic" batches that cause severe load imbalance across instances when scheduling is performed at the group granularity. (2) Under memory constraints, requests may be preempted or delayed, causing extremely long requests to be blocked and further exacerbating tail latency.

To mitigate the long-tail effect, recent works [5, 10, 11, 33, 51] have proposed *asynchronous RL*. Unlike synchronous methods, where all training experiences must originate from the current policy iteration, asynchronous methods allow training on stale rollout data from previous policy iterations, introducing *off-policyness*. Other works [8, 53], while nominally maintaining synchrony, allow deferring a portion of requests to subsequent iterations, thereby sacrificing iteration-level consistency compared to strict synchronous RL systems. Although asynchronous or non-strictly synchronous RL can reduce the long-tail impact, many RL practitioners still prefer synchronous training to achieve the best convergence and maintain reward stability.

Speculative decoding (SD) [2, 6, 14, 17–20, 22, 28] offers a promising approach to accelerate the long-tail phase while preserving synchronous RL training guarantees. SD consists of two stages: *Draft* and *Verification*. In the draft stage, a lightweight mechanism proposes candidate tokens, which are then verified in parallel by the target model in the verification stage. Since parallel verification of n tokens by the LLM is faster than serial generation of n tokens due to reduced memory access, SD can improve the generation speed for individual requests. Draft tokens can be produced by a smaller auxiliary model [17–20], which offer high acceptance rates but incur significant draft overhead, or by retrieval-based methods such as n-gram matching, which look up a request's recent tokens in its own generation history and extend the longest match. SuffixDecoding [14, 28] optimizes n-gram matching with a Compressed Suffix Tree (CST), a space-efficient suffix index that supports longest-match lookup and draft-token generation in linear time. However, when only the request's own history serves as the reference corpus, the acceptance rate remains low. RL rollout scenarios further exacerbate these limitations: batch sizes fluctuate dramatically, and the target LLM undergoes continuous updates, causing rapid "model drift". Existing SD strategies suffer from either high draft overhead or low draft token accuracy, resulting in marginal or even negative performance gains in rollout scenarios.

2.3 Opportunity: Shared Contextual Information in Group Sampling

In LLM reinforcement learning practice, the most widely adopted algorithm is Group Relative Policy Optimization (GRPO) [31]. GRPO performs group-based preference optimization by sampling G candidate responses for each prompt, computing rewards for these responses, and then normalizing the rewards *within* the group to obtain advantages. Several recent works [25, 48, 50] refine GRPO to improve generalization, but they all preserve the core principle of generating multiple responses for the same prompt. Other efforts, such as BroRL [12] and Knapsack RL [21], even scale G to 512 or more to enhance exploration. As a result, GRPO and its variants establish *group sampling* as a dominant training paradigm, in which each prompt is associated with a set of responses generated by the same policy.

This group sampling paradigm naturally introduces strong and stable contextual signals during rollout. Since all responses in a group are conditioned on the same prompt and produced by the same model, they exhibit notable similarity in semantics, structural templates, and generation length. However, existing RL systems typically treat each prompt group as a monolithic unit during rollout, overlooking the potential to externalize and exploit such intra-group similarity

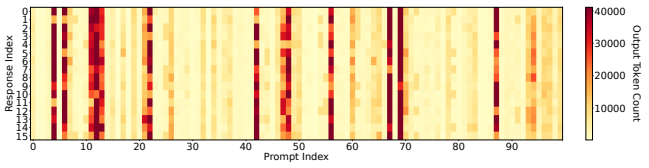


Figure 4: Length correlation within response groups. Each column represents a prompt group in the GRPO rollout, and each cell corresponds to an individual response. The color intensity indicates output length.

Table 2: Mean acceptance length in n-gram speculative decoding with grouped pattern references under different draft strategies. Linear generates one draft sequence per step, while multi-path generates multiple candidate sequences with top-k branching. Values represent mean acceptance length (including bonus token).

Ref. Count	**Linear**	**Multi-Path (k=2)**	**Multi-Path (k=4)**
$n = 0$ (baseline)	1.70	1.77	1.85
$n = 1$	2.04	2.14	2.25
$n = 5$	2.32	2.44	2.59
$n = 15$	2.53	2.69	2.85

as **shared contextual information**. Our statistical analysis shows that this shared context is sufficiently rich to support both scheduling and inference optimizations.

First, at the *length* level, responses within the same group have highly correlated generation lengths. Prior work [3,7,51] has shown that response length is predictable from prompt and model characteristics. Our measurements on real RL workloads, summarized in Figure 4, further confirm that most prompt groups in GRPO rollout exhibit strong length correlation. We observe that responses within the same group tend to have similar lengths, forming visually consistent columns. This length context can be obtained cheaply via speculative sampling and used to inform global scheduling policies, such as approximate longest-first scheduling, which prioritizes groups likely to produce very long responses. Such length- and memory-aware scheduling is a promising direction for mitigating the long-tail effect while respecting device memory constraints.

Second, at the *pattern* level, grouped responses share recurring semantic and syntactic structures that can be exploited to accelerate inference. Instead of relying solely on each request's own history, we can aggregate previously generated tokens from other responses in the same group into a shared pattern dictionary (e.g., via compressed suffix trees) and use it as an n-gram reference for speculative decoding. To quantify this opportunity, we sample 20 prompt groups from a Qwen2-VL-72B task and simulate n-gram speculative decoding under different draft strategies. Table 2 reports the mean acceptance length (including bonus tokens) for varying numbers of grouped references n and different drafting modes. Compared to the self-referencing baseline with no grouped references ($n = 0$), incorporating even a small number of intra-group references (e.g., $n = 1$ or $n = 5$) consistently increases the mean acceptance length, and using more references (e.g., $n = 15$) together with multi-path drafting yields the largest gains. In the later stage of rollout, speculative decoding with full grouped pattern references can improve the number of accepted draft tokens by up to 119% compared to a baseline that only uses per-request history.

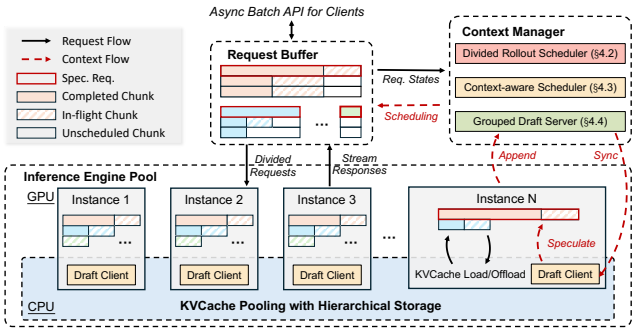


Figure 5: The overview of SEER.

In summary, group sampling in GRPO-like algorithms provides rich shared contextual information, both in terms of length correlation and pattern similarity, that is currently underutilized by existing RL systems. Our statistical data indicates that this contextual signal can be harnessed to enable length- and memory-aware scheduling and grouped speculative decoding, offering a promising opportunity to mitigate long-tail inefficiency and improve rollout throughput in synchronous LLM RL training.

3 Design of SEER

3.1 Overview

SEER is a synchronous, colocated RL system designed to substantially reduce rollout long-tail latency and improve overall system throughput without compromising algorithmic fidelity. The broader pipeline remains standard: training uses Megatron [35] for distributed optimization, rollout uses vLLM [16] for generation, reward computation is handled by an independent server that runs in parallel with generation, and Moonshot Checkpoint Engine [1] ensures rapid movement of updated model weights. SEER's contributions therefore layer onto a conventional architecture without imposing additional integration overhead.

Within rollout, as motivated by the workload analysis in §2.3, SEER introduces a unifying idea that drives its three innovations: *Online Context Learning*. SEER continuously learns the intra-group shared properties and uses them to

guide scheduling and decoding. Length similarities become signals for estimating whether upcoming generations are likely to be long-running; pattern similarities reveal opportunities to construct group-level draft predictions that accelerate decoding without a separate draft model. In addition to these optimizations, SEER also incorporates efficient memory management, load balancing, and asynchronous reward computation.

To support this context-driven approach at scale, the rollout subsystem is organized into three tightly connected components as illustrated in Figure 5. An `Inference Engine Pool` hosts multiple model-serving instances backed by a distributed KVCache pool, enabling KVCache migration across nodes without recomputation. A `Request Buffer` provides a global view of all pending and in-progress rollout work, allowing the system to coordinate concurrency precisely. Atop these components, a logically centralized `Context Manager` maintains group-level contextual views and drives scheduling and draft generation decisions.

On this foundation, SEER introduces three techniques that address the underlying challenges of long-CoT rollout. Divided Rollout (§3.2) resolves the concurrency–memory dilemma by breaking each request into smaller schedulable units and enabling fine-grained placement across instances. Context-Aware Scheduling (§3.3) then leverages learned length context to approximate a longest-first policy that reduces tail latency. Finally, Adaptive Grouped Speculative Decoding (§3.4) exploits shared token-pattern structure to build speculative decoding at the group level and dynamically adjusts draft lengths to maximize throughput. Despite these performance enhancements, SEER maintains logical consistency across all stages of the synchronous RL pipeline, thereby achieving an algorithmically lossless reinforcement learning process.

3.2 Divided Rollout

As illustrated in Figure 1, conventional group-level request scheduling mechanisms lead to severe load imbalance both within and across inference instances. Divided Rollout addresses this core tension by rethinking what constitutes a schedulable unit. Rather than binding any request group to a single execution slot, SEER not only decomposes it into individual requests but further divides each request into a sequence of generation chunks, each representing a bounded segment of progress. A request that would previously monopolize an instance for thousands of tokens becomes a pipeline of shorter units that can be interleaved and redistributed as the system's load evolves. This change in granularity enables continuous rebalancing: after each chunk completes, the next chunk can be scheduled on whichever instance currently has the most available memory and compute.

However, Divided Rollout poses significant challenges to the KVCache system. When requests are rescheduled at the chunk level, all KVCache will be recomputed, and this overhead can even negate the benefits of load balancing. Although existing inference frameworks [4, 41] support offloading request KVCache to DRAM within instances, the dynamic load balancing scheduling policy may reschedule requests to other instances, leading to cache misses. Therefore, Divided Rollout requires a globally shared, high-capacity cache system to support chunk-level request scheduling with minimal overhead.

To address this challenge, SEER builds upon Mooncake [30] to construct a globally shared KVCache pool distributed across inference nodes. The KVCache of all active requests is stored in a hierarchical global storage spanning DRAM and SSD, with RDMA enabling rapid KVCache transfer between nodes. From the perspective of the upper-level scheduling system, request chunk scheduling can be treated as stateless, eliminating the need to consider KVCache distribution. To further improve cache utilization and reduce the migration overhead, SEER layers a cache-aware policy on top of this pool. Using the number of in-flight requests as a load signal, the scheduler routes each incoming request to the node holding the longest matching cache prefix as long as the load remains balanced across instances. Once the spread of in-flight counts exceeds an imbalance threshold, the scheduler switches to a load-first decision and dispatches the request to the least-loaded node. Whenever this load-driven placement leaves a cache-affinity gap larger than a configurable margin such as 512 tokens, the chosen node proactively pulls the missing KVCache from the best-affinity node so that the corresponding prefix is not recomputed. The pool itself further exploits the predictable access pattern of a rollout iteration. Rather than relying solely on LRU, the KVCache of a request is released the moment that request finishes, which maximizes the DRAM capacity available to active requests.

What distinguishes Divided Rollout from prior offloading or preemption-based approaches is that cache movement is no longer a reactive measure taken only when memory pressure becomes intolerable. Instead, since chunk-level work units have well-defined memory footprints, the scheduler can proactively place and move them to optimize global system balance rather than merely avoiding out-of-memory events. This shift from reactive eviction to proactive, mobility-aware scheduling is the key novelty that allows SEER to resolve the concurrency–memory dilemma that has long constrained synchronous RL rollout. Compared with live migration systems for online serving (e.g., Llumnix [37]), it offers greater scheduling flexibility due to the absence of strict per-request latency constraints.

3.3 Context-Aware Scheduling

Long-CoT rollout inevitably runs into memory limits, which means some requests must wait. When the scheduler cannot anticipate which requests will run long, these delays occur

arbitrarily, creating unnecessary congestion in the tail. As shown in §2.3, requests within the same GRPO group tend to exhibit similar output lengths. SEER leverages this structure to predict which requests are likely to be long-running and schedules them more intelligently. The goal is to approximate a longest-first scheduling (LFS) policy (known to reduce tail latency) without needing to know true output lengths in advance.

SEER's key idea is to designate one request per group as a **speculative request**. This request serves as an online probe: by observing how quickly it completes during generation, SEER can infer the approximate length of the other requests in the group. To surface these signals early, speculative requests are placed in a high-priority path and scheduled according to a shortest-first strategy (SFS). Because short requests complete quickly while long ones linger, this "length filtering" step exposes potential long-tail groups early in the rollout, giving SEER time to react before those long generations dominate system resources.

Specifically, the scheduling process unfolds in three conceptual phases. First, speculative requests are executed promptly so that length information can be gathered with minimal delay. Second, the Context Manager maintains and continually updates an estimated output length for each group. This estimate is simply the maximum generation length observed among completed requests in the group, an approach that naturally converges toward the true length as more information becomes available. For groups in which no request has finished yet, SEER conservatively assumes they may be long-tail cases and initializes their estimate to the upper bound on generation length. Third, with group-level length estimates in place, the scheduler shifts to an approximate LFS policy: groups predicted to be long are prioritized so that their requests begin making progress early, rather than being deferred until system load is already high. The detailed scheduling algorithm is provided in §A.1.

SEER also incorporates safeguards to mitigate inaccuracies in early predictions. It occasionally schedules requests from underserved groups to avoid starvation and updates group length estimates conservatively to reflect the longest sample observed so far. These mechanisms ensure stability even when speculative signals are noisy. As shown later in §4.5.1, this context-aware scheduling policy approaches the performance of an oracle LFS scheduler that has perfect knowledge of request lengths.

3.4 Adaptive Grouped Speculative Decoding

To further improve resource utilization during the rollout phase, especially in the long-tail stage, SEER implements Adaptive Grouped Speculative Decoding, which **adaptively** adjusts draft lengths based on computational intensity and exploits the **grouped** pattern context (as analyzed in Table 2) to enhance acceptance rates.

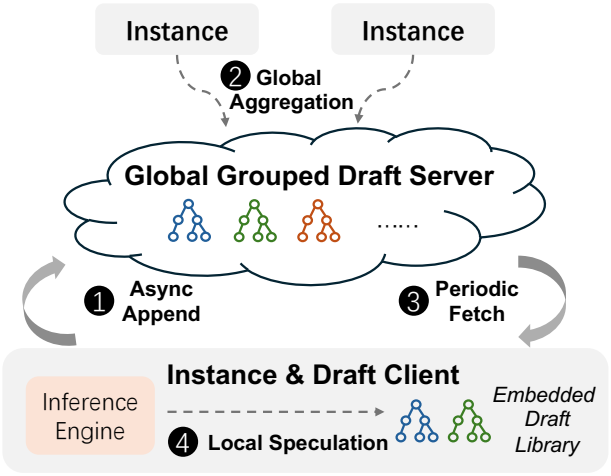


Figure 6: The distributed grouped draft server.

3.4.1 Challenges of Speculative Decoding in Rollout

Speculative decoding (SD) benefits from underutilized computational resources when batch sizes are small, enabling faster parallel verification compared to serial generation. However, rollout scenarios feature dynamically varying batch sizes that start large and rapidly decrease as long requests dominate. Fixed-length SD methods often incur excessive draft or verification overhead, degrading overall throughput. We now model SD throughput [17] in the rollout scenario to identify the key factors affecting performance. Let α denote the acceptance rate, where $\alpha = E(\beta)$ and β is the acceptance probability at each position. Let γ represent the number of draft tokens predicted by the draft model per step, B denote the current batch size, $D(B,\gamma)$ represent the forward time of the draft model, and $T(B,\gamma)$ represent the forward time of the target model.

The expected number of tokens generated per request in each forward is $\frac{1-\alpha^{\gamma+1}}{1-\alpha}$. The expected time for SD to generate one token per request is:

$$T_{SD} = \frac{(1-\alpha)(D(B,\gamma)+T(B,\gamma))}{1-\alpha^{\gamma+1}}$$

SD provides a benefit when T_{SD} is less than the forward time of the target model $T(B,1)$.

When B is small, $D(B,\gamma)+T(B,\gamma)$ is approximately equal to $T(B,1)$, making SD clearly beneficial. However, when B is large, $D(B,\gamma)$ becomes non-negligible and $T(B,\gamma)$ increases rapidly with γ as the target model becomes compute-bound, potentially resulting in negative gains from SD. In rollout scenarios, B dynamically varies and can range from 1 to several hundred. Conventional static SD strategies with fixed draft lengths suffer from either underutilization of computational resources with small γ or excessive verification overhead with

large γ, resulting in marginal or even negative performance gains. Furthermore, to optimize SD performance, the draft system should minimize $D(B,\gamma)$ while maximizing α. In rollout scenarios dominated by large batches, conventional SD methods suffer from either high $D(B,\gamma)$ (e.g., using a separate draft model) or low α (e.g., naive n-gram methods). Therefore, an efficient draft system specifically tailored to the characteristics of rollout workloads is required.

3.4.2 Adaptive Speculation with Disaggregated Architecture

SEER introduces the Distributed Grouped Draft Server (DGDS), a disaggregated SD framework specifically designed for rollout scenarios. As illustrated in Figure 6, DGDS aggregates response pattern context across requests and instances through an independent draft server, which asynchronously distributes the context to the embedded draft client within each inference instance. As shown in Table 2, multi-reference and multi-path n-gram methods can significantly improve the acceptance rate; however, a naive implementation incurs prohibitive overhead. To address this, DGDS employs a distributed draft service and adopts an asynchronously updated Compressed Suffix Tree (CST) to minimize speculative decoding latency $D(B,\gamma)$ on the critical path. The detailed workflow is provided in §A.2. Compared with prior CST implementations [28, 41], DGDS is optimized for rollout scenarios with support for highly concurrent access and dynamic updates. Its distributed architecture enables cross-instance context sharing, substantially improving both the efficiency and acceptance rate of speculative decoding.

To ensure that SD consistently provides positive gains throughout the rollout process, it is necessary to dynamically adjust γ based on the current batch size according to the throughput model. Furthermore, as described in §3.3, requests are classified into high-priority and low-priority categories. High-priority speculative requests are used to probe the length distribution of requests within the same group and should complete faster, thus requiring higher draft budgets. However, acceptance rates decrease rapidly with increasing draft length, making overly disparate budget allocations between high-priority and low-priority requests wasteful. To address this challenge, we propose a **Marginal-Benefit-Aware (MBA) Adaptive Speculation** policy inspired by classical utility maximization and marginal-utility scheduling principles widely used in resource allocation systems [15], which dynamically balances overall throughput with the latency of high-priority requests.

Algorithm 1 presents the MBA strategy. Using offline-profiled T_{SD} models and online-collected acceptance rates and batch sizes, the algorithm determines draft lengths (γ_h, γ_l) for high-priority and low-priority requests. The algorithm is invoked periodically during rollout, as request characteristics remain relatively stable over short time intervals. Given

Algorithm 1 Marginal-Benefit-Aware Adaptive Speculation

Require: High-priority and low-priority batch sizes B_h and B_l; per-position acceptance probabilities $\beta[1], \beta[2], \ldots$; maximum token budget per request $\gamma_{\max}$; priority factor $\lambda \in [1, \infty)$ (e.g., $\lambda = 2$).
Ensure: Draft token counts γ_h and γ_l.

$B \leftarrow B_h + B_l$
$\gamma^* \leftarrow \arg\min_\gamma T_{SD}(B,\gamma)$ ▷ optimal draft length for batch size B
$\Gamma^* \leftarrow \gamma^* \cdot B$ ▷ total token budget
if $\Gamma^* < B_h$ **then**
  return $(\gamma_h = 0, \gamma_l = 0)$ ▷ disable speculation
▷ Allocate budget based on marginal benefit
$\gamma_h \leftarrow 1, \gamma_l \leftarrow 0, \mathit{remaining} \leftarrow \Gamma^* - B_h$
while $\mathit{remaining} > 0$ **do**
  $\mathit{benefit}_h \leftarrow B_h \cdot (\beta[\gamma_h] - \beta[\gamma_h + 1])$
  $\mathit{benefit}_l \leftarrow B_l \cdot (\beta[\gamma_l] - \beta[\gamma_l + 1])$
  if $\mathit{benefit}_h > \lambda \cdot \mathit{benefit}_l$ **and** $\gamma_h < \gamma_{\max}$ **and** $\mathit{remaining} \geq B_h$ **then**
    $\gamma_h \leftarrow \gamma_h + 1$ ▷ allocate to high-priority
    $\mathit{remaining} \leftarrow \mathit{remaining} - B_h$
  else if $B_l > 0$ **and** $\gamma_l < \gamma_{\max}$ **and** $\mathit{remaining} \geq B_l$ **then**
    $\gamma_l \leftarrow \gamma_l + 1$ ▷ allocate to low-priority
    $\mathit{remaining} \leftarrow \mathit{remaining} - B_l$
  else
    break ▷ cannot allocate further
return (γ_h, γ_l)

the draft lengths, the embedded draft client in DGDS generates the corresponding number of draft tokens based on the grouped CSTs for each request. The basic single-path speculation algorithm follows SuffixDecoding [28], where each candidate path is assigned a confidence score computed from suffix probabilities. Building on this foundation, SEER uses these scores to filter low-probability candidates and is capable of returning multiple candidate paths via a beam search that retains the top-k highest-scoring branches at each fork.

For long-tail requests, adaptive grouped speculative decoding offers two particular advantages. First, during the long-tail stage, concurrency is minimal, allowing larger draft lengths to increase the number of accepted tokens per request. SEER also implements multi-path speculative decoding to further improve the acceptance length in the long-tail stage. Second, as more requests in the same group complete over time, the CST aggregates richer contextual information. As demonstrated in Table 2, this enables long-tail requests to achieve substantially longer acceptance lengths.

4 Evaluation

In this section, we evaluate SEER's performance advantages during the rollout phase, particularly in improving throughput and reducing tail latency (§4.2). In the ablation study (§4.3), we provide a detailed analysis of how each technique in SEER contributes to improving end-to-end performance. In §4.4, we compare SEER against non-strictly synchronous RL in

Table 3: Model configurations and RL workload characteristics.

Metric	Moonlight	Qwen2-VL-72B	Kimi-K2
Model Size	32 GB	146 GB	1 TB
Total GPUs	32	128	256
GPUs per Instance	1	8	32
Reqs per Iter	3200	9600	6400
Group Size	8	16	8
Temperature	1.0	0.8	1.0
Max. Gen. Length	65536	40960	98304
Avg. Gen. Length	22386	7615	38959

both rollout performance and training quality, demonstrating that SEER achieves lossless acceleration. Finally, we present extended studies (§4.5) on scheduling, speculative decoding, and the global KVCache pool to demonstrate the effectiveness of our techniques.

4.1 Setup

Testbed. Our experimental infrastructure consists of 32 high-performance compute nodes, each equipped with 8×H800 GPUs, 224 CPU cores, 2TB DRAM, and 4TB NVMe storage, providing sufficient hardware support for the global KVCache pool and distributed grouped draft server. For deployment, we adopt a strategy that balances task performance with resource efficiency by configuring different numbers of GPUs and parallelization strategies to serve models based on their size and architecture.

Models and Workloads. To validate the generalizability of SEER's design, we evaluate three models with diverse sizes and output characteristics: Moonlight [24], Qwen2-VL-72B [43], and Kimi-K2 [38]. All models are trained as **reasoning models** with chain-of-thought capabilities, where Moonlight and Kimi-K2 are trained on mathematical datasets, while Qwen2-VL-72B is trained on language-vision-mixed reasoning tasks using the LLM-as-a-Judge [36] reward model. Their configurations and RL workload characteristics are shown in Table 3. We employ Qwen2-VL-72B with tensor parallelism (TP8) and Kimi-K2 with data parallelism (DP32) and expert parallelism (EP32).

Settings. We use the GRPO [31] algorithm in our experiments. The group size (i.e., the number of requests per prompt) follows RL training conventions unless otherwise specified: we use a smaller group size (8) for mathematical problems and a larger group size (16) for open-ended tasks with LLM-as-a-Judge evaluation. For SEER's SD hyperparameters described in §3.4.2, we set $\gamma_{max} = 8$ and $\lambda = 2$.

Baselines. We evaluate SEER against a set of strong and representative baselines that capture the best available techniques for synchronous rollout scheduling, skewness mitigation, and speculative decoding.

(1) *veRL* [34]: veRL is a state-of-the-art synchronous RL system that supports efficient colocation of training and rollout. It provides a well-engineered baseline for measuring SEER's end-to-end improvements because it already incorporates optimized model execution pipelines and a production-grade rollout subsystem.

(2) *StreamRL-Oracle*: Although StreamRL [51] is designed as a disaggregated asynchronous RL framework, its proposed Skewness-Aware Scheduling directly targets the same long-tail latency issues that appear in synchronous rollout. StreamRL's scheduling relies on a small auxiliary model that predicts prompt lengths and uses those predictions to perform bucketing and LFS-style scheduling. To ensure a **fair and stringent comparison**, we evaluate against StreamRL-Oracle, which uses the ground-truth prompt lengths obtained from veRL's rollout traces rather than model predictions. This removes the confounding effects of prediction errors and reflects the best-case performance achievable by the StreamRL scheduling approach.

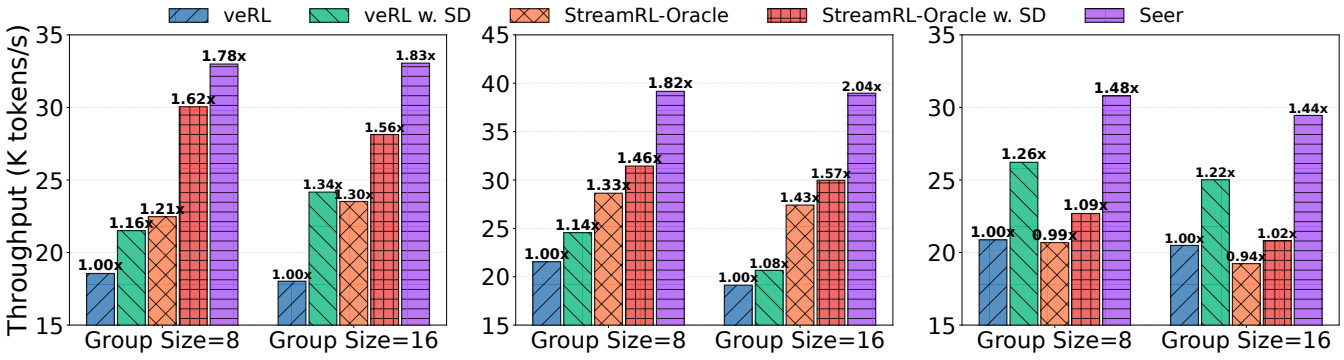
(3) *Vanilla Speculative Decoding (SD)*: To benchmark against modern decoding accelerators, we implement strong speculative decoding baselines tailored to each model family. For Moonlight, we use SuffixDecoding [28] with a maximum draft length of $\gamma_{max} = 16$. For Qwen2-VL-72B, we adopt a dedicated Qwen2-7B-VL draft model with $\gamma_{max} = 3$. For Kimi-K2, we apply Multi-Token Prediction (MTP) [22] with $\gamma_{max} = 1$. Each of these SD methods is integrated into both veRL and StreamRL-Oracle pipelines, enabling direct comparisons across scheduling regimes and decoding strategies.

To isolate the effect of scheduling and speculative mechanisms, all baselines (including SEER) use a unified in-house implementation of vLLM [16] as the inference engine. This ensures that differences in performance arise solely from the scheduling and decoding techniques under evaluation, not from underlying infrastructure discrepancies.

Metrics. For the end-to-end experiments, we measure the average rollout throughput across 5 iterations, which is defined as the average number of output tokens generated per second in each rollout iteration.

4.2 End-to-End Performance

In our end-to-end experiments, we conduct three RL tasks based on the aforementioned models, with workload configurations detailed in Table 3. We provide a comprehensive comparison of SEER's end-to-end performance against different baselines in §4.2.1. To validate SEER's effectiveness



(a) Moonlight (b) Qwen2-VL-72B (c) Kimi-K2

Figure 7: End-to-end rollout throughput of RL systems across different tasks and group sizes.

in mitigating tail latency, we analyze the tail latency phenomenon in our experiments and compare the tail latency between SEER and the baseline system veRL in §4.2.2.

4.2.1 Rollout Throughput

We compare the throughput performance of SEER against state-of-the-art baselines across different workloads and group sizes, as shown in Figure 7. Despite significant variations in model size and workload characteristics across different RL tasks, SEER consistently achieves substantial speedups across all tasks, with throughput improvements ranging from 44% to 104% over veRL.

SEER also outperforms StreamRL-Oracle and multiple SD strategies. StreamRL sets buckets with smaller concurrency for long-request groups to minimize latency. However, it still treats each request group as an atomic, non-preemptible unit and cannot dynamically adjust instance loads at runtime. Consequently, its performance is heavily dependent on the accuracy of the preset bucketing algorithm. When encountering out-of-distribution workloads such as Kimi-K2, StreamRL-Oracle even underperforms veRL, which uses a simple round-robin scheduling strategy. For vanilla SD methods, although we have implemented adaptive draft length, these SD methods still suffer from either excessive overhead or low average acceptance length, resulting in inferior performance compared to our adaptive grouped speculative decoding. A detailed comparison is provided in §4.5.2.

Across different workloads, SEER achieves greater performance improvements on memory-constrained tasks (Moonlight and Qwen2-VL-72B), where SEER's divided rollout mitigates load imbalance and context-aware scheduling reduces long request delays. Furthermore, as the group size increases, the load imbalance caused by veRL's group-level scheduling becomes more severe, leading to throughput degradation. In contrast, SEER addresses the monolithic request group problem through divided rollout and dynamic load balancing, and leverages group context to optimize both scheduling and speculative decoding. As a result, when the group size increases from 8 to 16, SEER achieves an average performance improvement of 5%.

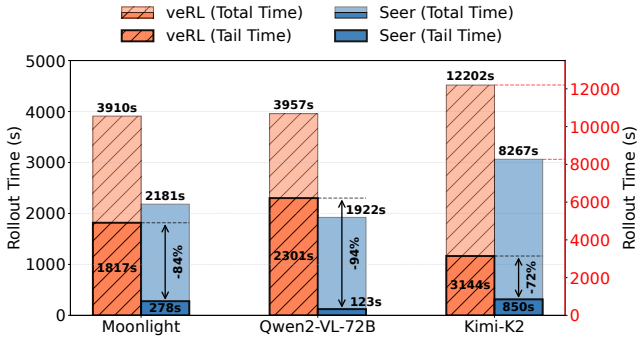


Figure 8: Tail time and total time of three RL tasks.

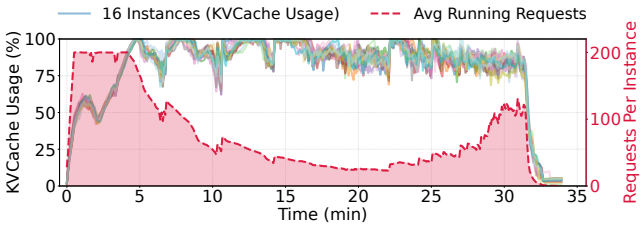


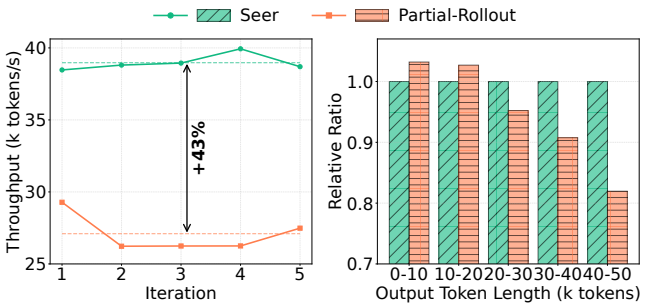
Figure 9: KVCache utilization and average running requests with SEER during a rollout iteration of the Qwen2-VL-72B task.

4.2.2 Long-Tail Time

To analyze SEER's performance advantages, we examine tail latency in the rollout process. We define **tail requests** as the last 10% of requests to complete in a synchronous system rollout, and **tail time** as the time spent **solely** processing these tail requests. Figure 8 shows the tail time and total rollout

Table 4: Performance improvement breakdown across three RL tasks. Context Sched. denotes context-aware scheduling (§3.3), and Grouped SD denotes adaptive grouped speculative decoding (§3.4).

Method	Moonlight	Qwen2-VL-72B	Kimi-K2
Baseline	1.00	1.00	1.00
+ Divided Rollout	1.41×	1.42×	1.16×
+ Context Sched.	1.47×	1.56×	1.27×
+ Grouped SD	1.90×	2.04×	1.53×



(a) Rollout throughput. (b) Output length distribution.

Figure 10: Rollout experiments comparing SEER and Partial Rollout on the Qwen2-VL-72B workload.

completion time, averaged across all iterations. The results demonstrate severe tail latency for memory-constrained tasks like Moonlight and Qwen2-VL-72B, where the last 10% of requests consume up to 50% of total time.

The tail latency problem stems from two primary causes. First, request queuing and preemption lead to delayed scheduling of long-output requests. Second, monolithic request groups result in load imbalance across instances. We provide quantitative evidence using the Qwen2-VL-72B task on veRL, shown in Figure 3. In this rollout iteration, a total of 13,686 preemption events occurred. The last 5% of completed requests have an average length in the top 15th percentile, yet their average execution start time is at 42% of the total time. The load imbalance across instances is even more pronounced: the completion time difference between the earliest and latest instances accounts for 70% of the total time, with each instance idle for an average of 1,580 seconds, representing 37% of the total time. These scheduling and load imbalances result in severe tail latency and throughput degradation.

By leveraging online context learning and fine-grained request scheduling, SEER significantly reduces tail latency by 72% to 94%, thereby substantially improving system throughput. Figure 9 illustrates the impact of these techniques, showing that SEER substantially reduces the tail phase duration compared to the baseline shown in Figure 3a.

4.3 Improvement Breakdown

We present a systematic ablation study to quantify the contribution of each optimization component to SEER's overall performance. Table 4 demonstrates the cumulative end-to-end speedup obtained by incrementally integrating each major optimization component, where each set of experiments is executed on the 5th rollout iteration.

First, SEER's divided rollout mechanism enables dynamic, fine-grained load balancing across instances, mitigating tail latency caused by inter-instance load imbalance and reducing preemption overhead within instances due to varying KV-Cache memory consumption. This optimization yields significant improvements for memory-constrained tasks, achieving up to 42% throughput improvement. Building upon divided rollout, context-aware scheduling leverages learned intra-group length distributions to guide scheduling, providing up to 14% additional throughput improvement. To further address the computational underutilization inherent in tail requests, SEER incorporates adaptive grouped speculative decoding, which contributes an additional 26–48% performance improvement over the scheduling optimizations alone.

4.4 Comparison with Non-Strictly Synchronous RL

Some RL training systems [8, 39, 53] employ non-strictly synchronous RL, which defers or reschedules long-tail requests to subsequent iterations. While non-strictly synchronous RL can improve resource utilization, it introduces some degree of inconsistency with synchronous training algorithms. Partial Rollout [39, 53] is a popular non-strictly synchronous method that over-issues requests during rollout and terminates the rollout phase once a predetermined number of requests are completed, with remaining requests executed with high priority in the next rollout iteration. In this section, we compare SEER against Partial Rollout. Following the experimental setup from APRIL [53], which applies Partial Rollout, each rollout iteration issues 2× the original number of requests and terminates once the original number of request groups have fully completed.

4.4.1 Rollout Performance

We first compare SEER and Partial Rollout on the Qwen2-VL-72B workload over five rollout iterations. As shown in Figure 10a, SEER achieves 43% higher average throughput than Partial Rollout. This advantage stems from two factors. First, in memory-constrained long-CoT scenarios, Partial Rollout's doubling of request volume exacerbates preemption issues, preventing it from fully leveraging its higher degree of parallelism. Second, SEER's adaptive grouped speculative decoding further enhances throughput for long-generation requests. Additionally, Figure 10b shows the output length

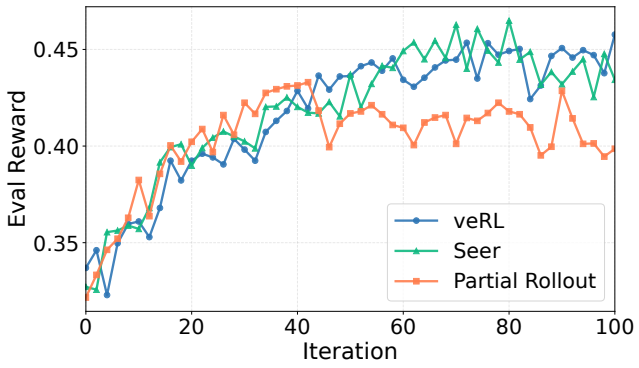


Figure 11: Reward curves of end-to-end RL training.

distributions of SEER and Partial Rollout. Compared to the synchronous method, Partial Rollout generates a significantly lower proportion of requests with longer output lengths. This bias may negatively impact RL training and degrade model performance, as we examine in detail in §4.4.2.

4.4.2 End-to-End Reward

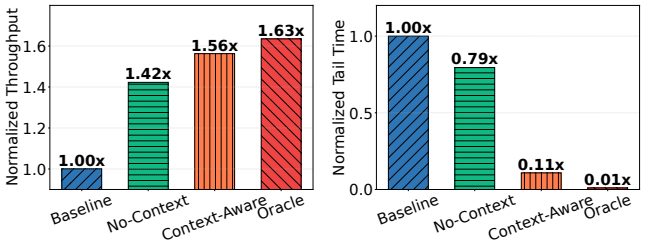
As stated in §3.1, SEER is designed to be algorithmically lossless with respect to strictly synchronous RL. To validate this property and compare the training quality of strictly and non-strictly synchronous RL, we evaluate three systems end-to-end: veRL as the strictly synchronous ground truth, SEER, and Partial Rollout. All three systems train Moonlight for 100 iterations under identical hyperparameters and data, and we evaluate checkpoints every two iterations on a fixed prompt set sampled with a shared random seed.

Figure 11 reports the resulting reward curves. The curves of veRL and SEER track each other closely throughout the run, indicating that SEER preserves the training quality of synchronous RL while delivering substantial performance gains. Partial Rollout, by contrast, falls measurably behind both synchronous baselines after roughly 50 iterations and stays below them thereafter. We attribute this divergence to Partial Rollout's bias against long-output requests, which skews the on-policy training data toward shorter trajectories. While recent work has explored algorithmic remedies such as bounding staleness [5, 10, 26] or adjusting the training objective via importance sampling and modified clipping [5, 26, 29], synchronous RL remains necessary in many scenarios due to its algorithmic stability.

4.5 Extended Studies

4.5.1 Context-Aware Scheduling

To validate the effectiveness of length context, we conduct an ablation study by varying the length information available to SEER's scheduler, as shown in Figure 12. Specifically, we compare: (1) No-Context, which applies only divided rollout



(a) Normalized throughput. (b) Normalized tail latency.

Figure 12: Impact of length context on improving throughput and reducing tail latency (defined in §4.2.2). No-Context applies only divided rollout without using length context to guide scheduling. Oracle obtains all output lengths in advance and applies the LFS strategy.

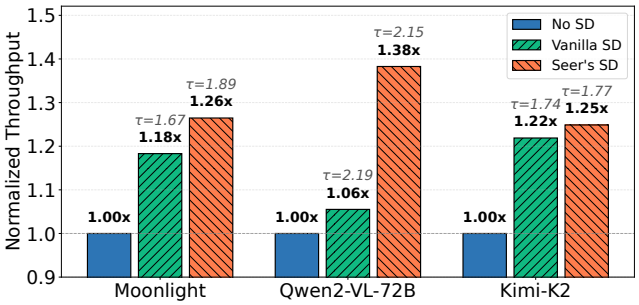


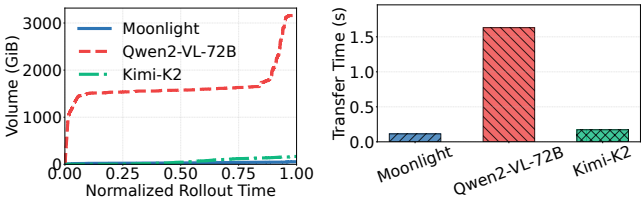
Figure 13: Normalized throughput and mean acceptance length (τ) of SD strategies across three tasks.

without using length context to guide scheduling, and (2) Oracle, which obtains the actual output lengths of all requests in advance and replays the rollout iteration using these precise lengths to guide scheduling.

While divided rollout significantly improves throughput through dynamic load balancing, the tail latency problem persists, with tail latency reduced by only 21% compared to the baseline. In contrast, context-aware scheduling leverages length prediction information from speculative requests to implement approximate longest-first scheduling, substantially reducing tail latency by 89% compared to the baseline. Compared to Oracle, context-aware scheduling achieves 96% of its throughput performance. This demonstrates that despite some performance degradation from prediction errors, context-aware scheduling still provides substantial benefits.

4.5.2 Adaptive Grouped Speculative Decoding

As discussed in §3.4, the benefits of speculative decoding during rollout depend on multiple factors including batch size, draft model overhead, and draft accuracy. We conduct ablation experiments on different SD strategies based on a single rollout iteration executed on veRL. As shown in Figure 13, SEER's adaptive grouped SD consistently outperforms



(a) Cumulative KVCache migration volume.

(b) Per-node time spent on KVCache migration.

Figure 14: Quantitative analysis of the overhead introduced by the Mooncake-based global KVCache pool used in divided rollout.

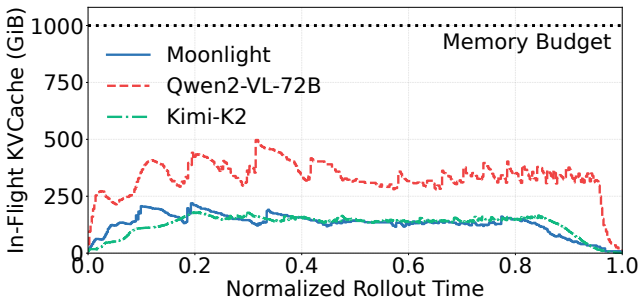


Figure 15: Peak KVCache storage footprint across nodes for in-flight requests during a rollout iteration, reported for the most loaded node in the rollout. Memory budget denotes the DRAM capacity allocated per node for KVCache storage.

vanilla SD in throughput across all tasks, achieving up to 1.3× speedup. Furthermore, by leveraging group context, our approach improves the mean acceptance length by 0.22 compared to the CST-based SD method [28], and exceeds MTP's performance. While vanilla SD with a small draft model achieves slightly higher mean acceptance length than our adaptive grouped SD, its excessive draft model overhead results in the lowest overall throughput gain.

4.5.3 Global KVCache Pool

SEER adopts the Mooncake global KVCache pool to share KVCache across inference nodes and thereby eliminate the re-prefill cost that divided rollout would otherwise introduce. We now quantify the migration traffic and storage occupancy of this pool to assess its impact on system performance. As described in §3.2, SEER layers a cache-aware policy on top of the global KVCache pool to improve cache locality. In the experiments reported below, the scheduler declares load imbalance whenever the spread of in-flight requests across instances exceeds 32, and triggers a KVCache migration whenever the cache hit length gap between the chosen node and the best-affinity node exceeds 512 tokens.

Figure 14a reports both the frequency and the cumulative volume of KVCache migrations observed during a full rollout iteration. The cache-aware scheduling policy keeps migrations sparse throughout the rollout. Even for Qwen2-VL-72B, which carries the largest per-token KVCache among the evaluated workloads, the total migration volume reaches only about 3 TB. This volume is negligible relative to the 8×400 Gbps RDMA fabric provisioned on each node. Figure 14b further reports that the average per-node time spent on KVCache transfer remains below 0.1% of the corresponding rollout duration, and these transfers can be overlapped with GPU computation. Taken together, the migration cost of the global KVCache pool has no measurable impact on end-to-end performance.

Another concern is the storage capacity of the global KVCache pool. Should the pool be unable to hold the KVCache of all active requests, the resulting cache misses would force divided requests to re-prefill and degrade end-to-end performance. Figure 15 reports the aggregate KVCache footprint of in-flight requests, namely requests that have been scheduled but not yet completed. The measurement is taken at the granularity of an 8-GPU node and aggregated over the most loaded node across the rollout cluster. Even under the heaviest load observed in our workloads, the footprint never exceeds half of the allocated DRAM budget. This headroom indicates that the request-aware eviction policy keeps the working set of all active requests resident within the global pool, so capacity-induced cache misses do not arise in practice.

5 Related Work

RL Frameworks for LLM Post-Training. Numerous open-source RL frameworks [13, 32, 34, 42, 44, 47, 54] have been proposed to achieve both efficiency and usability, providing robust infrastructure support for both research and production RL workloads. OpenRLHF [13] and veRL [34] seamlessly integrate training and inference engines, supporting various parallelism strategies to improve RL training efficiency. ROLL [44] targets large-scale RL training optimization, enabling flexible resource allocation and heterogeneous task scheduling. Slime [54] balances high performance with flexibility, supporting highly customizable data generation pipelines. These works primarily focus on overall RL training workflow orchestration, while the long-tail problem in the rollout phase remains unaddressed. SEER complements these frameworks by providing a system-level optimization for the rollout phase, which can be integrated into existing RL frameworks to further enhance their performance.

On-Policy RL Optimization. Some efforts have been made to optimize the RL workflow at both the system and algorithmic levels. RealHF [27] dynamically reallocates model parameters and memory budgets to optimize utilization. RLHFuse [52] proposes stage fusion to overlap reward and experience computation with the rollout long tail, thereby improving resource utilization. RLBoost [46] utilizes preemptible fragmented GPU resources to accelerate rollout at low cost.

Nevertheless, these approaches do not adequately address the long-tail latency problem. For the long-tail problem, speculative decoding [2, 6, 14, 17–20, 22, 28] is regarded as an effective optimization technique. However, these methods suffer from either high draft overhead or low accuracy in RL rollout scenarios due to dynamically changing batch sizes and continuously evolving target LLM weights. RhymeRL [11] and SPEC-RL [23] propose utilizing historical sequences generated from previous RL iterations as references to accelerate decoding. However, in most state-of-the-art model rollout scenarios, different iterations sample different data to improve generalization, leaving no historical sequences to reuse. SEER systematically elevates online group-level context as a first-class citizen in RL rollout optimization, eliminating the reliance on repeated sequences across iterations. Meanwhile, it employs an adaptive speculation policy atop a disaggregated architecture, achieving higher acceptance rates while maintaining low draft overhead.

Off-Policy RL Optimization. Recently, many works [5, 8, 10, 11, 26, 29, 33, 51, 53] have proposed RL workflows that do not maintain complete logical synchronization, trading a degree of off-policy behavior for improved RL efficiency. StreamRL [51], Areal [5], RhymeRL [11], and Laminar [33] propose asynchronous training approaches to improve resource utilization. In asynchronous RL, training and rollout are completely decoupled: rollout workers continuously generate new outputs without waiting, while training workers update the model whenever a batch of samples is collected. Other works relax the synchronization constraints of synchronous RL to achieve speedup, referred to as non-strictly synchronous RL. Partial Rollout [39, 53] defers long-tail requests to continue execution in the next rollout iteration. RollPacker [8] proposes tail batching, which schedules long-tail requests into a few designated long iterations, effectively reducing GPU idle time. While these techniques improve RL efficiency, the off-policyness inherent in these algorithms introduces risks of instability and accuracy degradation, limiting their applicability in research and production environments where algorithmic fidelity is critical. SEER significantly reduces rollout tail latency through fine-grained scheduling within the rollout phase, while maintaining consistency with on-policy algorithms.

6 Conclusion

In this paper, we present SEER, a synchronous RL system that accelerates the rollout process through online context learning. With divided rollout, a fine-grained and dynamically load-balanced scheduling approach, SEER leverages the similarity among requests within the same group in GRPO-like algorithms to achieve efficient scheduling and speculative decoding, while strictly maintaining consistency with on-policy RL algorithms. Experimental results demonstrate that SEER achieves up to 2.04× throughput improvement and up to 94% reduction in long-tail latency compared to the current state-of-the-art RL framework. Beyond RL rollout, SEER's design principles constitute a system-level optimization paradigm for grouped inference that generalizes to other large-scale parallel LLM generation scenarios.

Acknowledgments

We thank the anonymous reviewers and our shepherd for their valuable feedback. The authors affiliated with Tsinghua University are all in the Department of Computer Science and Technology, Beijing National Research Center for Information Science and Technology (BNRist), Tsinghua University, China. This work was supported in part by the Beijing Natural Science Foundation(Grant No.L252014).

References

[1] Moontshot AI. Checkpoint engine. https://github.com/MoonshotAI/checkpoint-engine, 2025.

[2] Tianle Cai, Yuhong Li, Zhengyang Geng, Hongwu Peng, Jason D Lee, Deming Chen, and Tri Dao. Medusa: Simple llm inference acceleration framework with multiple decoding heads. *arXiv preprint arXiv:2401.10774*, 2024.

[3] Ke Cheng, Wen Hu, Zhi Wang, Peng Du, Jianguo Li, and Sheng Zhang. Enabling efficient batch serving for lmaas via generation length prediction. In *2024 IEEE International Conference on Web Services (ICWS)*, pages 853–864. IEEE, 2024.

[4] LMSYS Corp. Sglang. https://github.com/sgl-project/sglang, 2025.

[5] Wei Fu, Jiaxuan Gao, Xujie Shen, Chen Zhu, Zhiyu Mei, Chuyi He, Shusheng Xu, Guo Wei, Jun Mei, Jiashu Wang, et al. Areal: A large-scale asynchronous reinforcement learning system for language reasoning. *Advances in Neural Information Processing Systems*, 38:36256–36282, 2026.

[6] Yichao Fu, Peter Bailis, Ion Stoica, and Hao Zhang. Break the sequential dependency of llm inference using lookahead decoding. *arXiv preprint arXiv:2402.02057*, 2024.

[7] Yichao Fu, Siqi Zhu, Runlong Su, Aurick Qiao, Ion Stoica, and Hao Zhang. Efficient llm scheduling by learning to rank. *Advances in Neural Information Processing Systems*, 37:59006–59029, 2024.

[8] Wei Gao, Yuheng Zhao, Dakai An, Tianyuan Wu, Lunxi Cao, Shaopan Xiong, Ju Huang, Weixun Wang, Siran Yang, Wenbo Su, et al. Rollpacker: Mitigating long-tail

rollouts for fast, synchronous rl post-training. *arXiv preprint arXiv:2509.21009*, 2025.

[9] Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Peiyi Wang, Qihao Zhu, Runxin Xu, Ruoyu Zhang, Shirong Ma, Xiao Bi, et al. Deepseek-r1 incentivizes reasoning in llms through reinforcement learning. *Nature*, 645(8081):633–638, 2025.

[10] Zhenyu Han, Ansheng You, Haibo Wang, Kui Luo, Guang Yang, Wenqi Shi, Menglong Chen, Sicheng Zhang, Zeshun Lan, Chunshi Deng, et al. Asyncflow: An asynchronous streaming rl framework for efficient llm post-training. *arXiv preprint arXiv:2507.01663*, 2025.

[11] Jingkai He, Tianjian Li, Erhu Feng, Dong Du, Qian Liu, Tao Liu, Yubin Xia, and Haibo Chen. History rhymes: Accelerating llm reinforcement learning with rhymerl. *arXiv preprint arXiv:2508.18588*, 2025.

[12] Jian Hu, Mingjie Liu, Ximing Lu, Fang Wu, Zaid Harchaoui, Shizhe Diao, Yejin Choi, Pavlo Molchanov, Jun Yang, Jan Kautz, et al. Brorl: Scaling reinforcement learning via broadened exploration. *arXiv preprint arXiv:2510.01180*, 2025.

[13] Jian Hu, Xibin Wu, Zilin Zhu, Weixun Wang, Dehao Zhang, Yu Cao, et al. Openrlhf: An easy-to-use, scalable and high-performance rlhf framework. *arXiv preprint arXiv:2405.11143*, 2024.

[14] Yuxuan Hu, Ke Wang, Xiaokang Zhang, Fanjin Zhang, Cuiping Li, Hong Chen, and Jing Zhang. Sam decoding: Speculative decoding via suffix automaton. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 12187–12204, 2025.

[15] Frank P Kelly, Aman K Maulloo, and David K Tan. Rate control for communication networks: shadow prices, proportional fairness, and stability. *Journal of the Operational Research Society*, 49(3):237–252, 1998.

[16] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph E. Gonzalez, Hao Zhang, and Ion Stoica. Efficient memory management for large language model serving with pagedattention. In *Proceedings of the ACM SIGOPS 29th Symposium on Operating Systems Principles*, 2023.

[17] Yaniv Leviathan, Matan Kalman, and Yossi Matias. Fast inference from transformers via speculative decoding. In *International Conference on Machine Learning*, pages 19274–19286. PMLR, 2023.

[18] Yuhui Li, Fangyun Wei, Chao Zhang, and Hongyang Zhang. Eagle-2: Faster inference of language models with dynamic draft trees. *arXiv preprint arXiv:2406.16858*, 2024.

[19] Yuhui Li, Fangyun Wei, Chao Zhang, and Hongyang Zhang. Eagle: Speculative sampling requires rethinking feature uncertainty. *arXiv preprint arXiv:2401.15077*, 2024.

[20] Yuhui Li, Fangyun Wei, Chao Zhang, and Hongyang Zhang. Eagle-3: Scaling up inference acceleration of large language models via training-time test. *arXiv preprint arXiv:2503.01840*, 2025.

[21] Ziniu Li, Congliang Chen, Tianyun Yang, Tian Ding, Ruoyu Sun, Ge Zhang, Wenhao Huang, and Zhi-Quan Luo. Knapsack rl: Unlocking exploration of llms via optimizing budget allocation. *arXiv preprint arXiv:2509.25849*, 2025.

[22] Aixin Liu, Bei Feng, Bing Xue, Bingxuan Wang, Bochao Wu, Chengda Lu, Chenggang Zhao, Chengqi Deng, Chenyu Zhang, Chong Ruan, et al. Deepseek-v3 technical report. *arXiv preprint arXiv:2412.19437*, 2024.

[23] Bingshuai Liu, Ante Wang, Zijun Min, Liang Yao, Haibo Zhang, Yang Liu, Anxiang Zeng, and Jinsong Su. Spec-rl: Accelerating on-policy reinforcement learning via speculative rollouts. *arXiv preprint arXiv:2509.23232*, 2025.

[24] Jingyuan Liu, Jianlin Su, Xingcheng Yao, Zhejun Jiang, Guokun Lai, Yulun Du, Yidao Qin, Weixin Xu, Enzhe Lu, Junjie Yan, et al. Muon is scalable for llm training. *arXiv preprint arXiv:2502.16982*, 2025.

[25] Zichen Liu, Changyu Chen, Wenjun Li, Penghui Qi, Tianyu Pang, Chao Du, Wee Sun Lee, and Min Lin. Understanding r1-zero-like training: A critical perspective. In *Second Conference on Language Modeling*.

[26] Han Lu, Zichen Liu, Shaopan Xiong, Yancheng He, Wei Gao, Yanan Wu, Weixun Wang, Jiashun Liu, Yang Li, Haizhou Zhao, et al. Part ii: Roll flash–accelerating rlvr and agentic training with asynchrony. *arXiv preprint arXiv:2510.11345*, 2025.

[27] Zhiyu Mei, Wei Fu, Kaiwei Li, Guangju Wang, Huanchen Zhang, and Yi Wu. Real: Efficient rlhf training of large language models with parameter reallocation. *arXiv preprint arXiv:2406.14088*, 2024.

[28] Gabriele Oliaro, Zhihao Jia, Daniel F Campos, and Aurick Qiao. Suffixdecoding: Extreme speculative decoding for emerging ai applications. In *The Thirty-ninth Annual Conference on Neural Information Processing Systems*, 2025.

[29] Alexandre Piché, Ehsan Kamalloo, Rafael Pardinas, Xiaoyin Chen, and Dzmitry Bahdanau. Pipelinerl: Faster on-policy reinforcement learning for long sequence generation. *arXiv preprint arXiv:2509.19128*, 2025.

[30] Ruoyu Qin, Zheming Li, Weiran He, Jialei Cui, Feng Ren, Mingxing Zhang, Yongwei Wu, Weimin Zheng, and Xinran Xu. Mooncake: Trading more storage for less computation—a KVCache-centric architecture for serving LLM chatbot. In *23rd USENIX Conference on File and Storage Technologies (FAST 25)*, pages 155–170, 2025.

[31] Zhihong Shao, Peiyi Wang, Qihao Zhu, Runxin Xu, Junxiao Song, Xiao Bi, Haowei Zhang, Mingchuan Zhang, YK Li, Yang Wu, et al. Deepseekmath: Pushing the limits of mathematical reasoning in open language models. *arXiv preprint arXiv:2402.03300*, 2024.

[32] Gerald Shen, Zhilin Wang, Olivier Delalleau, Jiaqi Zeng, Yi Dong, Daniel Egert, Shengyang Sun, Jimmy Zhang, Sahil Jain, Ali Taghibakhshi, et al. Nemo-aligner: Scalable toolkit for efficient model alignment. *arXiv preprint arXiv:2405.01481*, 2024.

[33] Guangming Sheng, Yuxuan Tong, Borui Wan, Wang Zhang, Chaobo Jia, Xibin Wu, Yuqi Wu, Xiang Li, Chi Zhang, Yanghua Peng, et al. Laminar: A scalable asynchronous rl post-training framework. *arXiv preprint arXiv:2510.12633*, 2025.

[34] Guangming Sheng, Chi Zhang, Zilingfeng Ye, Xibin Wu, Wang Zhang, Ru Zhang, Yanghua Peng, Haibin Lin, and Chuan Wu. Hybridflow: A flexible and efficient rlhf framework. In *Proceedings of the Twentieth European Conference on Computer Systems*, pages 1279–1297, 2025.

[35] Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. Megatron-lm: Training multi-billion parameter language models using model parallelism. *arXiv preprint arXiv:1909.08053*, 2019.

[36] Guijin Son, Hyunwoo Ko, Hoyoung Lee, Yewon Kim, and Seunghyeok Hong. Llm-as-a-judge & reward model: What they can and cannot do. *arXiv preprint arXiv:2409.11239*, 2024.

[37] Biao Sun, Ziming Huang, Hanyu Zhao, Wencong Xiao, Xinyi Zhang, Yong Li, and Wei Lin. Llumnix: Dynamic scheduling for large language model serving. In *18th USENIX symposium on operating systems design and implementation (OSDI 24)*, pages 173–191, 2024.

[38] Kimi Team, Yifan Bai, Yiping Bao, Guanduo Chen, Jiahao Chen, Ningxin Chen, Ruijue Chen, Yanru Chen, Yuankun Chen, Yutian Chen, et al. Kimi k2: Open agentic intelligence. *arXiv preprint arXiv:2507.20534*, 2025.

[39] Kimi Team, Angang Du, Bofei Gao, Bowei Xing, Changjiu Jiang, Cheng Chen, Cheng Li, Chenjun Xiao, Chenzhuang Du, Chonghua Liao, et al. Kimi k1.5: Scaling reinforcement learning with llms. *arXiv preprint arXiv:2501.12599*, 2025.

[40] Meituan LongCat Team. Introducing longcat-flash-thinking: A technical report. *arXiv preprint arXiv:2509.18883*, 2025.

[41] vLLM Team. vllm. https://github.com/vllm-project/vllm, 2025.

[42] Leandro von Werra, Younes Belkada, Lewis Tunstall, Edward Beeching, Tristan Thrush, Nathan Lambert, Shengyi Huang, Kashif Rasul, and Quentin Gallouédec. Trl: Transformer reinforcement learning. https://github.com/huggingface/trl, 2020.

[43] Peng Wang, Shuai Bai, Sinan Tan, Shijie Wang, Zhihao Fan, Jinze Bai, Keqin Chen, Xuejing Liu, Jialin Wang, Wenbin Ge, et al. Qwen2-vl: Enhancing vision-language model's perception of the world at any resolution. *arXiv preprint arXiv:2409.12191*, 2024.

[44] Weixun Wang, Shaopan Xiong, Gengru Chen, Wei Gao, Sheng Guo, Yancheng He, Ju Huang, Jiaheng Liu, Zhendong Li, Xiaoyang Li, et al. Reinforcement learning optimization for large-scale learning: An efficient and user-friendly scaling library. *arXiv preprint arXiv:2506.06122*, 2025.

[45] Peter Weiner. Linear pattern matching algorithms. In *14th Annual Symposium on Switching and Automata Theory (swat 1973)*, pages 1–11. IEEE, 1973.

[46] Yongji Wu, Xueshen Liu, Haizhong Zheng, Juncheng Gu, Beidi Chen, Z Morley Mao, Arvind Krishnamurthy, and Ion Stoica. Rlboost: Harvesting preemptible resources for cost-efficient reinforcement learning on llms. *arXiv preprint arXiv:2510.19225*, 2025.

[47] Zhewei Yao, Reza Yazdani Aminabadi, Olatunji Ruwase, Samyam Rajbhandari, Xiaoxia Wu, Ammar Ahmad Awan, Jeff Rasley, Minjia Zhang, Conglong Li, Connor Holmes, et al. Deepspeed-chat: Easy, fast and affordable rlhf training of chatgpt-like models at all scales. *arXiv preprint arXiv:2308.01320*, 2023.

[48] Qiying Yu, Zheng Zhang, Ruofei Zhu, Yufeng Yuan, Xiaochen Zuo, Yu Yue, Weinan Dai, Tiantian Fan, Gaohong Liu, Lingjun Liu, et al. Dapo: An open-source llm reinforcement learning system at scale. *arXiv preprint arXiv:2503.14476*, 2025.

[49] Chujie Zheng, Kai Dang, Bowen Yu, Mingze Li, Huiqiang Jiang, Junrong Lin, Yuqiong Liu, An Yang, Jingren Zhou, and Junyang Lin. Stabilizing reinforcement learning with llms: Formulation and practices. *arXiv preprint arXiv:2512.01374*, 2025.

[50] Chujie Zheng, Shixuan Liu, Mingze Li, Xiong-Hui Chen, Bowen Yu, Chang Gao, Kai Dang, Yuqiong Liu, Rui Men, An Yang, et al. Group sequence policy optimization. *arXiv preprint arXiv:2507.18071*, 2025.

[51] Yinmin Zhong, Zili Zhang, Xiaoniu Song, Hanpeng Hu, Chao Jin, Bingyang Wu, Nuo Chen, Yukun Chen, Yu Zhou, Changyi Wan, et al. Streamrl: Scalable, heterogeneous, and elastic rl for llms with disaggregated stream generation. *arXiv preprint arXiv:2504.15930*, 2025.

[52] Yinmin Zhong, Zili Zhang, Bingyang Wu, Shengyu Liu, Yukun Chen, Changyi Wan, Hanpeng Hu, Lei Xia, Ranchen Ming, Yibo Zhu, et al. Optimizing RLHF training for large language models with stage fusion. In *22nd USENIX Symposium on Networked Systems Design and Implementation (NSDI 25)*, pages 489–503, 2025.

[53] Yuzhen Zhou, Jiajun Li, Yusheng Su, Gowtham Ramesh, Zilin Zhu, Xiang Long, Chenyang Zhao, Jin Pan, Xiaodong Yu, Ze Wang, et al. April: Active partial rollouts in reinforcement learning to tame long-tail generation. *arXiv preprint arXiv:2509.18521*, 2025.

[54] Zilin Zhu, Chengxing Xie, Xin Lv, and slime Contributors. slime: An llm post-training framework for rl scaling. https://github.com/THUDM/slime, 2025. GitHub repository. Corresponding author: Xin Lv.

A Implementation Details of SEER

A.1 Context-Aware Scheduling Algorithm

Algorithm 2 presents the context-aware scheduling workflow built on top of divided rollout. The algorithm is invoked continuously by SEER's global scheduler, each time returning a scheduling decision $(r^\star, i^\star)$ that assigns the selected request $r^\star$ to an inference instance $i^\star$, until all requests are completed.

Algorithm 2 Context-Aware Scheduling based on Divided Rollout

Require: Active requests $\mathcal{R} = \{r_{g,i}\}$ grouped by prompt g; group-level length estimates $\{\widehat{L}_g\}$; inference instances I with KV-usage telemetry.
Ensure: A scheduling decision $(r^\star, i^\star)$ with $r^\star \in \mathcal{R}$ and $i^\star \in I$.

for all $r_{g,i} \in \mathcal{R}$ **do**
  if $r_{g,i}$ is finished **then**
    $\widehat{L}_g \leftarrow \textsc{UpdateEstimate}(\widehat{L}_g, L_{g,i})$
    remove $r_{g,i}$ from $\mathcal{R}$
  else if $r_{g,i}$ is the group's speculative request **then**
    keep in high-priority queue Q_{spec}
  else
    add to low-priority candidate set C_{rest}
$r^\star \leftarrow$ **None**
if $\neg\textsc{IsEmpty}(\mathsf{Q}_{\text{spec}})$ **then**
  $r^\star \leftarrow \textsc{PickSFS}(\mathsf{Q}_{\text{spec}})$ ▷ smallest generated length first
else if $\neg\textsc{IsEmpty}(\mathsf{C}_{\text{rest}})$ **then**
  $r^\star \leftarrow \textsc{PickLFS}(\mathsf{C}_{\text{rest}})$ ▷ largest $\widehat{L}_g$ first
else
  return `all requests are finished`
$r^\star.\mathit{max_tokens} \leftarrow \min\big(\mathit{chunk_size}, r^\star.\mathit{ori_max_tokens} - r^\star.\mathit{generated_tokens}\big)$
$i^\star \leftarrow \textsc{SelectInstance}(I, r^\star.\mathit{max_chunk_tokens}, \text{KV-usage})$
if $i^\star \neq$ **None then**
  return $(r^\star, i^\star)$
return `no available instance for this cycle`

A.2 Workflow of Distributed Grouped Draft Server

To minimize speculative decoding latency in the critical path, Distributed Grouped Draft Server (DGDS) adopts asynchronous updates and employs a distributed master-worker architecture for global sharing of grouped context, as illustrated in Figure 6. The system operates through four key steps, with core APIs listed in Table 5 and Table 6:

1) Asynchronous Append: Each inference instance runs an independent process to handle output tokens. When newly generated tokens are produced, the instance invokes the `update_cst` API to send them to DGDS, identified by `group_id`. To reduce communication overhead, each request batches a fixed number of tokens before sending updates, which has negligible impact on draft token quality.

2) Global Aggregation: DGDS aggregates token updates from requests belonging to the same group. To prevent cross-request interference, DGDS isolates updates by `request_id`, mapping each new token only to the corresponding local path in the CST.

3) Periodic Fetch: Each inference instance embeds a draft client as a library component that periodically synchronizes the latest CST from DGDS. The client registers active request groups via `register_group` and then periodically invokes `fetch_cst` to retrieve the latest CSTs for these groups. To reduce communication overhead, the client supports incremental synchronization based on local cache states.

4) Local Speculation: Inference instances perform speculation based on their local CSTs by invoking the `batch_speculate` API. The local CSTs aggregate paths from all requests within the same group, enabling instances to share contextual statistics and obtain higher-quality draft tokens.

Table 5: Grouped draft server API.

Operation	Description	Parameters
`update_cst`	Append generated tokens from a specific request to the compressed suffix tree	`const string& group_id, int request_id, int prev_token_count, const vector<int>& new_tokens`
`fetch_cst`	Fetch incremental draft contexts of request groups based on their current cache states (if have)	`const vector<string>& group_ids, const vector<DraftCacheInfo>& draft_cache_infos`

Table 6: Draft client API.

Operation	Description	Parameters
`register_group`	Register a new request group for draft fetching with TTL	`const string& group_id, int ttl_seconds`
`batch_speculate`	Generate speculative tokens for multiple requests via zero-copy memory access: reads input token patterns and writes predicted tokens directly to inference engine memory buffers	`const vector<string>& group_ids, const vector<size_t>& buffer_offsets, void* pattern_buffer_ptr, void* output_buffer_ptr, const vector<SpeculationArgs>& speculation_args`
SpeculationArgs: `max_spec_tokens` (int), `pattern_lookup_max` (int), `pattern_lookup_min` (int), `top_k` (int)		

Harvesting Sub-Microsecond CXL Memory Stalls with LITESWITCH

Nanqinqin Li
Princeton University

Yuhong Zhong
Columbia University

Asaf Cidon
Columbia University

Michael J. Freedman
Princeton University

Abstract

Compute Express Link (CXL) offers a practical path to scale memory capacity and bandwidth available to a single host. However, CXL memory incurs sub-microsecond latency that is typically 3× or more compared to local memory, exacerbating memory-induced CPU stalls and degrading application performance. This paper presents LITESWITCH, a lightweight hardware-software co-design that opportunistically harvests otherwise idle cycles caused by CXL memory accesses. LITESWITCH introduces: (1) a hardware mechanism that precisely identifies a CXL-induced memory stall and exposes it to software with near-perfect accuracy and minimal overhead; (2) an ultra-fast software path that switches to another ready thread for harvesting in under 20 nanoseconds, an order of magnitude faster than conventional context switches. Together, these mechanisms enable efficient harvesting of sub-µs-scale memory stalls (> 200 nanoseconds), without requiring changes to the application. The evaluation demonstrates that with a sufficient number of available threads per core, LITESWITCH recovers up to 80% of the performance lost due to CXL access latency, enabling the adoption of CXL memory without prohibitive slowdowns.

1 Introduction

Compute Express Link (CXL) has emerged as a promising technology to scale memory capacity and bandwidth at the server/rack levels beyond DDR limitations [48]. CXL is being evaluated in four concrete use cases: (1) capacity expansion via tiered or pooled memory, where CXL-attached DRAM/SSD extends host DRAM for memory-hungry services at lower TCO (total cost of ownership) [6, 10, 25, 33, 44, 63, 64, 66]; (2) utilization improvement by pooling across servers memory [44] and PCIe devices (NICs, SSDs) [67] over CXL; (3) bandwidth scaling by interleaving host DRAM and CXL memory to aggregate DDR and PCIe channels [30]; and (4) pod-level shared memory for distributed systems to reduce messaging overhead [31, 52].

However, these benefits come at a performance cost, as accessing CXL-attached memory incurs much higher latency compared to local DRAM. Measured CXL memory devices report load latencies of 214-394 ns (nanoseconds), compared to 81-117 ns for DRAM [48], while pooled CXL deployments can reach up to 1 µs (microsecond) depending on topology, device type, and multi-tenant interference [10, 44, 67]. This higher latency exacerbates memory-induced CPU stalls (i.e., backend stalls) that already dominate execution time even on systems with only local memory. Empirically, many memory-intensive and cloud-scale workloads spend the majority (20-80%) of their cycles stalled on memory [21, 37, 50]. Under the same synchronous, blocking load/store interface, CXL's higher latency therefore prolongs each stall by hundreds of nanoseconds, amplifying an already significant inefficiency.

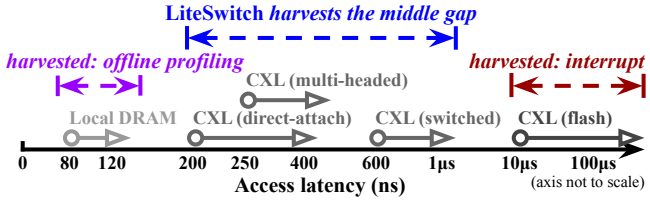


Figure 1: Illustration of CXL memory latency and the scope of prior work. CXL expands memory access latency from roughly 200 ns with direct-attach CXL memory to tens of µs with flash-backed CXL memory. Prior work harvests memory-induced CPU stall cycles near local DRAM latencies and at flash-scale latencies, leaving a substantial sub-µs "middle gap" that LITESWITCH targets.

This problem can be mitigated if the system could still do useful work while a thread is blocked waiting for memory access: a mechanism termed *stall cycles harvesting*. Simultaneous multithreading (SMT) mitigates memory stall cycles by interleaving two independent hardware threads on a shared core: when one thread stalls, the other can continue issuing instructions. However, SMT's concurrency is inherently limited: two hardware threads already saturate most cores' backends without incurring excessive intra-core contention [23, 42, 59]. Under long CXL stalls, both hardware threads can easily stall simultaneously. Moreover, even when only one thread is stalling, SMT cannot 'un-stall' that thread, thus the core falls back to lower utilization achievable by a single thread.

Software-based approaches instead 'un-stall' a blocked thread in place when stall happens and invoke a software routine to run another ready context (termed a scavenger) [50, 65]. MSH [50] relies on offline profiling to identify likely stall sites, while SkyByte [65] targets much higher (tens of µs) CXL-SSD latencies by using interrupts to preempt the blocked thread and then leveraging kernel-level scheduling. However, these designs face three fundamental obstacles when harvesting sub-µs CXL memory stalls (§3). **(1) Detection**. Software lacks advance knowledge (i.e., at compile

time) of when a CXL stall occurs or how long it will last. Because CXL latency is highly variable and topology-dependent, offline profiling is unreliable in dynamic, multi-tenant environments. **(2) Delivery**. Traditional interrupt delivery is a costly hardware path that must drain the CPU's speculative and out-of-order state and establish a precise architectural state, incurring sub-µs overhead that easily consumes the entire harvestable window. **(3) Scheduling**. Although user-level scheduling makes choosing the next runnable context cheap, switching between contexts remains expensive even in optimized runtimes [22, 36], where saving and restoring CPU's extended SIMD/FP state (`xstate`) alone costs tens to hundreds of nanoseconds.

This paper makes four key observations and insights to guide the design for efficiently harvesting CXL memory stalls:

- *Any memory access targeting a CXL memory device can be meaningfully harvested*, given the naturally large gap between local DRAM and CXL latencies (3× or more). This uniform treatment simplifies stall detection into two hardware operations already on the memory request path: a cache lookup (whether in L1/L2/LLC) and a routing decision (whether it targets a CXL memory device), achieving near-perfect accuracy with negligible extra overhead.
- *Harvesting should occur within the same address space*. This insight enables the possibility that stalls can be delivered to software via lightweight branching (i.e., control-flow jumps) than expensive interrupts.
- *Any regular worker thread can act as a scavenger during harvesting*, rather than dedicating separate best-effort threads. This enables simplified scavenger selection logic and avoids paying context switch cost twice per stall.
- *Usage of CPU extended state (`xstate`) is uncommon in most code*, thus saving and restoring `xstate` can be skipped safely when it is not actively used in a context.

This paper presents LITESWITCH, a lightweight hardware-software co-design that opportunistically harvests sub-µs CXL memory stall cycles, without requiring changes to the application. On the hardware side, LITESWITCH proposes *Location-Dependent Memory Branching* (LDMB, §4.1), a simple and novel CPU mechanism that detects stall conditions *online and per-access* based on existing cache lookup and routing operations. Then, LDMB un-stalls the CPU hardware thread and delivers the notification to software via a direct *control-flow jump*. LDMB revives the classic architectural idea of cache-miss-triggered software handling, which was previously shown to support low-overhead memory-performance monitoring and software-driven memory optimizations [28].

On the software side, LDMB's control jump invokes the *Bundled Handoff* handler (§4.2), where a small set of user threads from the same user process (a "bundle") is multiplexed on each hardware thread for fast scavenger selection and handoff, without interfering with the process's regular scheduler. This design is further optimized by `xstate`-*Aware Context Switch* (§4.3) that safely skips unnecessary saving and restoring of SIMD/FP state (`xstate`), reducing context switch overhead to under *20 ns* for non-compute-dense applications.

The observations and insights that guided LITESWITCH's design are grounded in measurements on a real CXL hardware tiering system in Intel Flat Memory Mode [6, 66]. However, a physical LDMB prototype is not implemented mainly because FPGAs cannot accurately model modern CPU's memory hierarchy. Instead, LITESWITCH uses emulation (§5.1) to reproduce realistic CXL memory stall behaviors and provide a software interface for running unmodified applications.

Together, LITESWITCH is a mechanism that can efficiently harvest memory stalls longer than 200 ns, making it effective even on the fastest, locally-attached CXL memory devices (Figure 1). The evaluation (§6) shows that LITESWITCH reduces performance slowdowns by up to 80% (multiplicatively) relative to the setup with 200-ns CXL access latency across diverse workloads. Crucially, LITESWITCH produces consistently low slowdowns even under more complex CXL configurations with higher latency and greater variability, highlighting its potential to enable practical adoption of CXL memory. The artifact of LITESWITCH is open-sourced [45].

LITESWITCH makes the following contributions:

- A characterization of unique challenges, observations, and insights for efficient harvesting of CXL memory stalls.
- A hardware proposal of LDMB for online, per-access stall detection and branch-based delivery that achieves near-perfect accuracy with negligible extra overhead.
- An ultra-lightweight software handler path that enables instant scavenger selection and minimal context switch overhead, without interfering with high-level scheduling policies or compromising correctness.
- A prototype emulation that runs unmodified applications to faithfully evaluate LITESWITCH, with comprehensive results showing it significantly reduces slowdowns.

2 Background and Motivation

CXL is a high-speed interconnect over the PCIe fabric that lets CPUs, accelerators, and device-attached memory participate in a unified memory space [17]. This paper focuses on the key use case of memory expansion via the `CXL.mem` protocol (Type-3 devices), which exposes device-attached memory to the CPU via load/store semantics.

The heterogeneity of CXL memory. As illustrated in Figure 1, CXL-attached memory spans a wide latency spectrum that is largely determined by device type and topology. Its inherent heterogeneity is shaped by device characteristics, CXL memory controller complexity, and topology. Recent measurements on production CXL memory expanders report load-to-use latencies of roughly 214–271 ns for locally attached, ASIC-based devices, only modestly higher than NUMA DRAM, while a single CXL switch or remote NUMA hop can push average latency toward 600 ns and beyond in

multi-hop fabrics [48]. At the other extreme, CXL-attached flash devices incur tens or hundreds of µs when requests miss DRAM caches and access NAND, though DRAM-cache hits can be served in sub-µs time [47, 64, 65].

To mitigate the monetary cost and latency overheads of CXL switching, emerging *multi-headed* CXL devices expose multiple host links into a single DRAM-backed pool, enabling rack-scale shared memory without an external CXL switch; these devices are an appealing solution for multi-host memory sharing [10, 27, 31, 35, 67].

Availability and use cases of CXL memory. CXL memory is available through a growing hardware ecosystem and is being explored in several concrete use cases. 25+ vendors ship CXL-capable CPUs, memory, switches, and controllers [15]. Recent systems (e.g. Intel Xeon 6 and AMD 5th Gen EPYC) provide 64 CXL lanes per socket [1, 2], for an aggregate 200–240 GB/s of bandwidth per direction (each lane provides 3.12–3.75 GB/s in each direction). In late 2025, Microsoft Azure launched the industry's first deployment of CXL-attached memory (in private preview) on its M-series VMs that leverages Intel Xeon 6 in Flat Memory Mode to support multi-terabyte in-memory workloads such as SAP HANA [33]. PNNL's Crete project employs a large pool of CXL-attached coherent memory alongside local DRAM to enable scientific and AI workloads to operate on datasets far larger and more shared than is possible on traditional HPC architectures [55]. At SC'25, XConn demonstrated a CXL memory pool shared across two NVIDIA H100 servers, achieving up to 6.5× higher throughput than RDMA for LLM inference and drastically reducing time-to-first-token [57].

2.1 CXL Exacerbates Memory Stalls

Memory-induced CPU stalls are already a major inefficiency on DRAM-only systems. Empirical studies show that many memory-intensive and cloud-scale workloads spend a large fraction of their total cycles stalled on memory: 25-31% for in-memory key-value and analytics workloads such as Masstree and Sphinx [50], around 50% in warehouse-scale web and storage services [37], and up to 80% in classic scale-out workloads such as OLTP, social networking, and data-serving [21].

CXL's higher access latency materially worsens this inefficiency. Studies on real CXL tiering and pooling systems confirm application-visible slowdowns relative to DRAM-only systems: Memstrata observes performance degradation of up to 34% even with hardware-managed tiering under moderate interference [66], and Pond reports that 21% of evaluated workloads experience $>$ 25% slowdown on an 8-socket, multi-tenant CXL fabric [44].

2.2 Stall Harvesting Meets CXL Memory

With the heterogeneity of CXL memory, the synchronous load/store interface meets far more variable and longer-latency access paths, yielding a unique profile: stalls are *hundreds of nanoseconds with much higher latency variability.*

On the high-latency end, SkyByte [65] targets stalls in CXL-SSDs that are tens of µs. When a request misses the device's internal DRAM cache and thus falls through NAND, the device raises an interrupt to 'un-stall' the thread; the OS kernel then schedules and switches context. This path has µs-scale overhead that is appropriate for misses in CXL-SSDs, but too costly to pay per event for sub-µs CXL memory stalls.

Offline profiling-based stall site identification can eliminate this cost altogether, but it assumes predictable stall locations. While this holds in some systems with only local DRAM [50], CXL introduces substantially greater variability: a memory page may reside on a direct-attach device, a multi-headed device, or behind a switch. Moreover, multi-tenant interference, which is common in CXL memory expansion use cases, further exacerbates unpredictability.

As a result, there is a substantial gap from $\sim$ 200 ns to $\sim$ 1 µs (Figure 1), in which stalls are common and harmful, yet existing solutions fall short. This gap calls for a harvesting mechanism that identifies stalls accurately and promptly in dynamic conditions and is ultra-lightweight in stall delivery and context switching.

3 Problem Statement and Challenges

The goal of LITESWITCH is to *efficiently* and *transparently* harvest CPU stall cycles on the order of *hundreds of nanoseconds* induced by CXL memory access, *without application changes*. LITESWITCH targets a harvestable budget of **200 ns** per stall, representative of CXL access latency for the simple memory extension use case, while more complex configurations such as pooling typically exhibit even longer stalls.

The general problem of stall harvesting decomposes into three steps: **detection**, **delivery**, and **scheduling**. Stall detection refers to the timely and accurate identification of stall conditions long enough to harvest. Stall delivery then 'unstalls' the hardware thread and invokes a software handler, with low overhead. Finally, the stall handler rapidly selects and switches to a ready scavenger thread. For harvesting to be worthwhile, the combined overhead of the three steps must be well below the 200-ns budget.

***Detection:* CXL memory latency is more unpredictable and variable than local DRAM.** The detection task's goal is to decide, early in the memory access path, whether a request will incur a high latency so that harvesting can begin in time. As discussed, offline profiling based methods assume predictable memory behavior. CXL adds layers of variability on top of an already opaque memory hierarchy: memory pages can move transparently between L1/L2/LLC, local DRAM, direct-attach CXL devices, and locations deeper into the CXL fabric. For example, in Intel Flat Memory Mode, a memory load can take anywhere from 1 ns (L1 cache hit)

to $\sim$ 300 ns (local DRAM miss) [66], and software cannot know at compile time what latency the load will exhibit.

This motivates a *runtime hardware hint* that indicates, with sufficient lead time, that a request is on the high latency path. For example, a load reaching the memory controller has already missed the cache and is a candidate for harvesting; more informative hints (e.g., queueing delay) are available deeper in the path (e.g., at the device-side CXL controller), but observing them at that point delays detection and thus reduces the harvestable window. The design questions are: which hint yields high accuracy, and where along the access path should it originate from to balance timeliness and accuracy?

***Delivery:* the load/store interface lacks a lightweight mechanism to 'un-stall' a hardware thread.** After a stall hint is raised, the hardware thread must be notified and control needs to be transferred to a software handler quickly. The closest mechanism today is hardware interrupt, but it is too costly for harvesting cycles in a sub-µs window. To deliver an interrupt safely, the CPU must squash speculative and out-of-order state and establish a precise architectural state before redirecting control. This path can cost hundreds of ns even when no privilege transition is needed (e.g., Intel User Interrupts allow direct user-space delivery of interrupts) [8,26]. An interrupt delivery costs around 600 ns in LITESWITCH's measurements, far exceeding the harvesting budget. Therefore, a hardware primitive that delivers a stall notification directly to a user-space handler, but at a much lower cost, is desired.

***Scheduling:* full context switching is too heavy for sub-µs stalls.** After delivery, the software handler must choose a scavenger thread and switch to it with minimal overhead, without violating critical scheduling properties (e.g., fairness and responsiveness) and without application changes. Existing user-level, µs-scale scheduling designs have significantly optimized the logic for selecting the next runnable thread [22,32,36], but context switches still remain costly [46]. A full context switch saves and restores large architectural state, including general registers, extended SIMD/FP state, and ancillary metadata that could read and write KiBs of data, taking up to $\sim$ 300 ns in LITESWITCH's experiments. Naively, through register liveness analysis, CPU state can be saved selectively because only a subset is actually live at any given context [50], but stalls are inherently arbitrary (i.e., any memory access could end up stalling). As a result, such analysis would either require costly online inspection or be infeasible to perform comprehensively offline. These costs necessitate a lightweight software handoff mechanism.

4 Design

Three principles guided the design of LITESWITCH: (1) while hardware changes are deemed necessary, LITESWITCH must incrementally build on existing hardware capabilities rather than introducing new ones; that is, the hardware mechanisms needed to support LITESWITCH should be trivially feasible on a real ISA platform; (2) unmodified applications should benefit from LITESWITCH across diverse workloads; and (3) the end-to-end overhead of detection, delivery, and scheduling must be as lightweight as possible to maximize harvesting.

Figure 2 demonstrates the overall architectural flow of LITESWITCH. At a high level:

- LITESWITCH detects stalls online and per-access in hardware to leverage first-hand hint of latency formation and performs detection as early as possible in the request path.
- LITESWITCH delivers stall notifications via lightweight control-flow jumps, confining harvesting activities within the same address space.
- LITESWITCH employs a simplified scavenger selection logic and performs context switch with the minimal execution state necessary for correctness.

For detection, LITESWITCH builds on the observation that *any memory access targeting a CXL memory device can be meaningfully harvested*. This intuition arises from the naturally large latency gap (typically 3$\times$ or more) between local DRAM and CXL memory. While more challenging to harvest, treating all CXL memory uniformly allows LITESWITCH to abstract away device and topology variability and simplify stall detection into two hardware operations already on the critical path for every memory request, a cache lookup (i.e., whether in L1/L2/LLC) and a routing decision (i.e., whether the target location resides on a CXL device), making detection trivially feasible. Critically, these operations occur entirely in the host CPU, avoiding the prohibitive delay of traversing the CXL fabric. With the two existing operations, LITESWITCH achieves per-access, online stall detection with near-perfect accuracy and adds negligible hardware overhead. Also, detection only applies to memory *load*, since memory store is either absorbed by CPU cache or triggers a read-for-ownership (RFO) and becomes a load anyways.

For delivery, LITESWITCH makes the design choice that harvesting should occur *within the same address space.* In OS terms, this typically means within the same user process, where the scavenger workers and the stall handler coexist with regular application threads. By restricting harvesting to a single address space, the handler becomes just another function and thus delivery avoids privilege transitions, CR3 changes, IDT lookup, and stack switching. Moreover, because harvestable stalls are tied to specific high-latency loads, the triggering condition is synchronous and precise: the CPU already knows the exact instruction boundary at which control must be redirected (similar to synchronous exceptions like page faults and divide errors) [34]. This further eliminates the need for establishing a precise state for asynchronous events like interrupts. Therefore, the hardware can enter the stall handler via a branch-like redirect that only needs to squash younger speculation, costing only tens of cycles [14, 18, 56].

To realize this, LITESWITCH introduces a new CPU mechanism called *Location-Dependent Memory Branching* (LDMB,

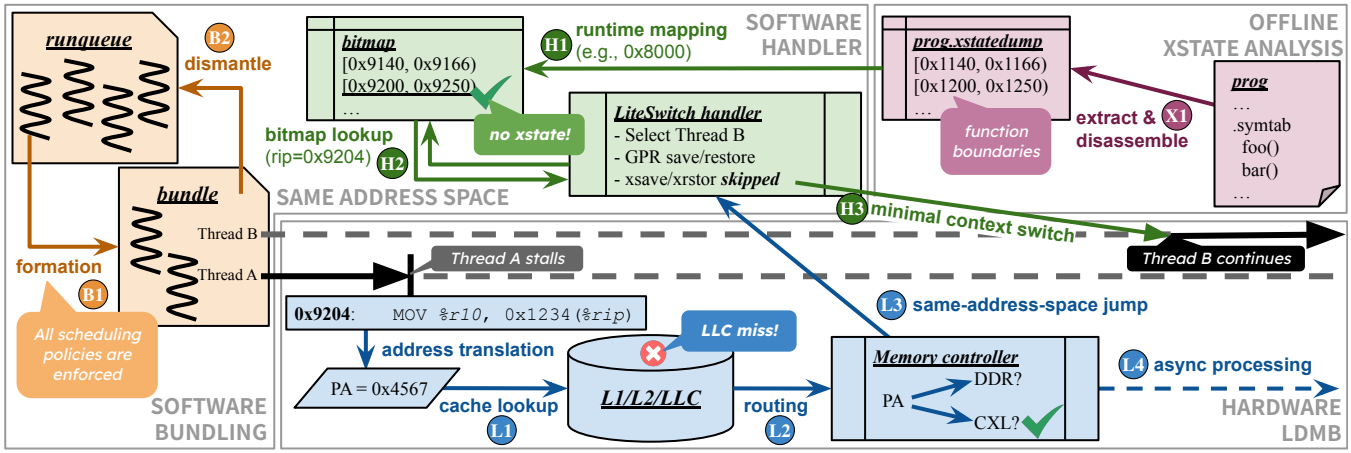


Figure 2: Overall system architecture of LITESWITCH. This figure illustrates the key components and workflow during a stall harvesting event for a multi-threaded program *prog*. When *Thread A* executes a load (rip = `0x9204`) that L1 misses the cache, the memory controller L2 routes it to CXL, triggers a stall, and transfers control to the LITESWITCH handler via L3 a jump while the load L4 is being processed asynchronously. The handler performs H2 a bitmap lookup (derived from X1 offline `xstate` analysis and H1 runtime bitmap mapping) and determines that the stall occurred in an `xstate`-free context, allowing it to skip `xsave`/`xrstor` during H3 the context switch. *Thread B* then executes to harvest the otherwise wasted cycles. Bundles are formed B1 and dismantled B2 dynamically by the application's regular scheduler.

§4.1), which detects stall conditions in the CPU's memory controller based on L1 the cache residency and L2 a routing decision, and delivers the notification to the user process by L3 directly branching to LITESWITCH's software stall handler within the same address space.

For scheduling, LITESWITCH adopts the design choice that *any regular worker thread can act as a scavenger for harvesting*, rather than dedicating separate best-effort threads. This leads to a mechanism called *Bundled Handoff* (§4.2), which multiplexes a small set of user threads, termed a *bundle*, on each CPU hardware thread. A bundle is B1 formed by the user process's regular scheduler (the kernel scheduler or a user-level one), which applies its normal policies when selecting threads, except that it groups multiple threads into a bundle than running a single thread. During the bundle's lifetime, a stall notification from LDMB invokes LITESWITCH's handler, which switches execution among threads within the bundle. Critically, bundle formation is dynamic: when the bundle is de-scheduled (e.g., a thread issued a blocking I/O), the bundle B2 breaks and all threads are returned to the regular scheduler for the next bundling, which will likely not be the same formation. Bundled Handoff does not require changes to the application and operates within the scope and capabilities of the user process's regular scheduler than overshadowing it.

Finally, LITESWITCH minimizes context switch overhead by *selectively skipping saving and restoring CPU's extended SIMD/FP state* (`xstate`), a mechanism termed H3 `xstate`-Aware Context Switch (§4.3). This is based on the observation that `xstate` takes up $>$ 80% of total context switch overhead, but its live usage is sparse in most code paths. For each application, LITESWITCH performs X1 a one-time offline analysis to identify *functions* that do not use `xstate` at all, H2 allowing stalls triggered within these contexts to be harvested safely without doing `xstate` save and restore. This method reduces an impractical register liveness analysis problem to a static, binary lookup, avoids most unnecessary `xstate` save/restore, and preserves context switch correctness.

4.1 Location-Dependent Memory Branching

LDMB is a simple hardware mechanism that LITESWITCH uses for online, per-access stall detection with same-address-space delivery. LDMB builds on the architectural principle in prior "cache-miss trap" mechanisms, which showed that miss-triggered, branch-like delivery to software handlers is feasible in both in-order and out-of-order processors [28]. LITESWITCH does not implement a physical LDMB prototype; instead, it emulates the mechanism's behavioral effect (§5.1). The scope of this work is not to specify a circuit-level design, but to demonstrate that such a mechanism, if available in modern CPUs, enables useful software capabilities. FPGAs are fundamentally unable to reproduce the timing, concurrency, and microarchitectural behaviors of the memory hierarchy in modern server-grade CPUs [13, 38, 39]. This subsection therefore outlines the high-level LDMB design and argues that it is trivially feasible to implement in real CPUs. The design is grounded in modern Intel x86_64, but the ideas should be generally portable to other ISAs.

Detecting CXL memory accesses and branch-based delivery. When a load misses in the LLC, the uncore memory subsystem routes the request to either DDR or CXL. In Intel Flat Memory Mode, CXL serves as a slower tier behind a direct-mapped DRAM cache; the memory controller reads the DRAM line and checks its tag to decide whether the access

missed DRAM and must be re-issued over CXL. In setups where CXL devices are directly exposed to the software (e.g., software-managed tiering), the high-order bits of the physical address (PA) determine DDR versus CXL routing [16]. In both scenarios, cache lookup and routing are already on the standard path; LDMB simply reuses this logic.

After a CXL memory load is detected and a miss status holding register (MSHR) entry is allocated for it, the memory controller raises a short signal back to the core complex, analogous to existing core-interconnect notifications such as "data-return." The signal triggers a brief microcode sequence that flushes younger micro-ops and transfers control to a pre-registered handler address, akin to a branch-misprediction redirect. This redirect is same-privilege by construction: a user-mode LDMB enters a user-mode handler, while a kernel-mode LDMB can only enter a kernel-mode handler. Because signaling occurs after MSHR allocation, the miss has already been issued and will be serviced asynchronously. Therefore, LITESWITCH's handler does not need to explicitly prefetch or manage completion. Upon resumption, the software simply retries the load. By then the line will typically hit in cache. If not, the line buffer coalesces the retry with the outstanding miss, so the retry only waits for the remaining service time rather than incurring another full-latency stall.

Considerations for memory-level parallelism (MLP). In modern out-of-order (OoO) processors, MLP allows multiple outstanding misses to overlap before a data dependency forces the core to stall. Thus, not every CXL-bound LLC miss is a harvestable stall event, and LDMB should redirect only on a *true stall*. Inspired by prior switch-on-event (SOE) mechanisms [5, 11, 24, 53], a true stall can be defined as a CXL-bound miss that is blocking in-order retirement: the miss is unresolved and its corresponding entry is at the head of the reorder-buffer (ROB). This condition is simple to test because OoO cores already maintain the association between an outstanding miss and the instruction that produced it. Therefore, LDMB can follow the principle used in SOE: when the memory controller signals a CXL-bound miss, LDMB's microcode compares that miss's ROB entry with the ROB head. If the entry is already at the head and the miss remains unresolved, LDMB proceeds with the redirect; otherwise, LDMB simply drops the opportunity and lets OoO execution continue.

This one-shot policy avoids complex tracking, such as tagging the ROB entry and redirecting later if it eventually reaches the ROB head. The lost opportunities should be rare. From the time the load issues, the core continues decoding and dispatching while the request traverses the TLB and cache hierarchy, is classified as CXL-bound, and then sends a signal back to the core. This path is a full DDR access ($\sim$100 ns) in Intel Flat Memory mode, and can be modeled as LLC-hit-scale ($\sim$20 ns) in direct-expose CXL configurations. Even at the shorter 20-ns timescale, prior studies show that modern OoO cores often exhaust their effective instruction window on real workloads [40, 41]. Therefore, by the time the LDMB signal arrives, the associated CXL-bound miss is likely already retirement-blocking. The ROB-head test captures these common cases while keeping LDMB's hardware contract deliberately small but effective.

Software interface. LDMB exports a small set of control registers, collectively called `LCR`, modeled after conventional x86 model-specific registers (`MSR`s). Specifically, `LCRc` provides enable bits and optional control knobs (e.g., range masks for advanced features), while `LCRh` stores the address of the LITESWITCH stall handler entry stub that LDMB jumps to. `LCRh` is banked by privilege level, analogous to how the x86 TSS stores per-ring stack pointers RSP0–RSP2.

For resumption, LDMB pushes a minimal *stall frame* onto the user stack, containing only the instruction pointer and flags (`rip` and `rflags`). The handler later uses a single `LRET` return instruction to resume execution, analogous to an interrupt return (`IRET` and `URET`) but without privilege transitions.

Cost estimation. The cost consists of (1) the signaling delay from the memory controller back to the core complex and (2) the microcode-driven branch redirection. A conservative estimate is 10 ns for the signaling [61] and another 10 ns for the branch flush and redirection [14, 18, 56], totaling roughly **20 ns** . This is an order of magnitude lower than a traditional interrupt and well within the 200 ns budget for harvesting.

4.2 Bundled Handoff

When LDMB delivers a stall notification, LITESWITCH transitions into its software handler to select another ready thread to execute. Bundled Handoff is the software runtime mechanism that performs this selection and in-process context switch. Its design follows one key principle: harvesting should operate *within* the scope of normal scheduling. The process's regular scheduler retains full control over fairness, core assignment, etc., while Bundled Handoff merely redistributes execution opportunities at a much finer granularity.

Any regular worker thread is a scavenger. Instead of maintaining a separate pool of best-effort, low-priority threads, LITESWITCH lets any regular worker thread in the same process act as a scavenger. This design rests on two observations. First, threads of a process commonly execute the same hot paths and share frequently accessed data structures (e.g., task queues, metadata, synchronization variables) [29, 51, 62]. Reusing an existing worker therefore preserves locality: L1/L2 cache state and TLB entries remain largely valid across handoffs, minimizing cold starts and interference. By contrast, an unrelated best-effort pool perturbs caches and TLBs, undermining the very cycles being harvested. Second, best-effort tasks must explicitly yield back before the stall resolves to not delay the main tasks, which doubles the switching cost and consequently, wipes out most of the potential gain. Moreover, colocating unrelated tasks with regular worker threads introduces security concerns [50].

Dynamic bundling. When the process's regular scheduler makes a decision, it picks a small bundle of ready threads (rather than a single one) from the runqueue. If a running thread triggers a CXL stall and LDMB branches into the handler, the handler selects another thread in the same bundle (in round-robin) and resumes its saved context. The handoff is *one-way*: the new thread runs until it incurs another stall. Bundles are dismantled at normal descheduling events, including explicit yield, blocking, time-slice expiration, and preemptive interrupts. When any thread in the bundle triggers one of these events, the bundle is torn down and all threads are returned to the regular scheduler. A subsequent scheduling decision forms a new bundle, which may differ due to fairness and load-balancing policies. Therefore, bundling is *dynamic*, not a static pre-assignment, and bundles contain only runnable threads, maximizing the utilization of harvestable stalls.

Bundled Handoff yields a two-tier scheduling hierarchy. The process's regular scheduler forms bundles, enforcing fairness, priority, and load balancing. Within a bundle, LITESWITCH performs handoffs triggered by LDMB at a finer granularity. Because all threads in a bundle have already been admitted by the regular scheduler, Bundled Handoff can bypass policy checks for scavenger selection.

Bundle sizing. A larger bundle exposes more ready threads for harvesting but increases intra-bundle contention on CPU resources; a smaller bundle reduces such contention but risks under-utilization if all threads simultaneously stall. In practice, bundles of *two* threads are often sufficient: returning to the original thread requires two handoffs, by which time the original stall typically resolves; otherwise, the retry simply waits out the remaining stall time than a full stall.

Implementation scope and application transparency. Bundled Handoff naturally fits user-level threading models, where context frames and metadata are accessible without privilege transitions. Language runtimes and user-space schedulers register a stall handler via a write to `LCRh` and use their native threading primitives for intra-bundle handoff. Kernel models can adopt a hybrid interface such as Linux's vDSO: the kernel scheduler manages bundling, while a vDSO-mapped handler runs in user space, so normal scheduling remains in kernel space and stall harvesting occurs entirely in user space. Because Bundled Handoff operates within existing threading abstractions, it preserves application transparency.

Operating regime and workload suitability. Because LITESWITCH harvests stalls by diverting execution to sibling threads, its primary effect is higher aggregate core utilization rather than shorter request latency. In latency-critical services whose end-to-end latency is dominated by the slowest sub-request (e.g., fan-out queries), in-stall handoff may not translate to visible benefits. By contrast, throughput-oriented workloads naturally tolerate short stalls in one thread while others continue. In these settings, LITESWITCH's ability to reclaim tens to hundreds of nanoseconds per stall raises effective instructions-per-core and, consequently, throughput.

How does Bundled Handoff compare to SMT. At a high level, both LITESWITCH and simultaneous multithreading (SMT) aim to improve CPU utilization by interleaving multiple threads on the same core, but they do so in fundamentally different ways. SMT exposes true hardware concurrency: multiple threads issue instructions in the same cycle, allowing more of a core's heterogeneous functional units (issue queues, ALUs, load/store units, etc.) to be utilized than a single thread possibly could. This overlap also allows one thread to make forward progress when its sibling is stalled, but the stalled thread stays blocked, reducing aggregate utilization. By contrast, LITESWITCH turns a blocking wait into an asynchronous one by 'un-stalling' the hardware thread with LDMB, exposing more runnable cycles to the software that SMT would otherwise leave unexploited.

Therefore, the two mechanisms are *orthogonal and complementary*. when hardware threads are stalled, LITESWITCH can continue harvesting cycles independently *within* each SMT thread context, extending SMT's ability to achieve higher aggregate core utilization under long CXL stalls (§6.1).

Cost estimation. The overhead has three parts. First, bundling happens at regular scheduling granularity and thus is amortized to negligible cost. Second, per-stall scavenger selection is lightweight and incurs *no lock contention*: the bundle metadata is small and strictly per-core and implemented via thread-local storage, so accesses are likely cache-hit with no cross-core sharing; empirically, selection takes about 4 ns.

4.3 `xstate`-Aware Context Switch

Context switch is the last step before a scavenger can begin harvesting, and it is also the dominant software cost due to unavoidable memory loads and stores. In this paper, *context switch* means saving the outgoing thread's architectural state and restoring the incoming's last saved state. LITESWITCH's prototype targets Linux on Intel x86_64, but the ideas apply to other ISAs with similar extended state handling.

Raw costs of context switch. Since LDMB appears a raw "interrupt" that can arrive at an *arbitrary* instruction boundary, the handler must treat the CPU's entire architectural state as volatile and handle save and restore properly.

The complete set of CPU state comprises: (i) *general state* with 15 general-purpose registers (GPRs) plus `rip`, `rsp`, and `rflags`; (ii) *extended state* (`xstate`) including floating-point (FP) and single-instruction multiple-data (SIMD) registers (x87/SSE/AVX/AVX-512), opmasks, etc. Saving/restoring the general state is a handful of cache-line stores/loads when laid out contiguously; in LITESWITCH's prototype, this path averages about 10 ns in real workloads. By contrast, `xstate` spans KiBs of data (up to 32 512-bit `zmm` registers plus ancillary state) and must be saved and restored using `xsave` and `xrstor`, whose latency grows with enabled features. In LITESWITCH's prototype, one `xsave` followed by `xrstor`

costs 70–300 ns across workloads, consuming most of a 200 ns harvestable window if paid on every handoff.

Observation: `xstate` usage is sparse in most code. Empirically, most application code paths do not touch `xstate`. Hot `xstate` usage concentrates in dense numerical kernels (e.g., linear algebra, ML inference, cryptography) and in a few library routines (e.g., `memcpy`) that are easy to identify. By contrast, control- and memory-centric paths (e.g., servers, iterator and graph logic, queue management) often run for long stretches without using `xstate`. Figure 12 shows the fraction of execution time spent in `xstate`-using functions across workloads: graph workloads are $< 5\%$ and most others are $< 55\%$. This motivates a design that *skips* `xsave` and `xrstor` when the current execution point is known to be `xstate`-free.

Design: `xstate`-aware skipping via function-range metadata. LITESWITCH performs a one-time offline binary analysis to identify *functions* that contain no `xstate`-relevant instructions. The output, termed `xstatedump`, records the precise ranges of such functions that remain `xstate`-free after inlining and code generation, using function boundaries from the ELF symbol table (`.symtab`).

At process load time, LITESWITCH ingests `xstatedump`, maps these ranges to the process's runtime addresses (via `/proc/self/mappings`), and builds a global, read-only *code bitmap* indexed at a fine granularity (e.g., 64-byte blocks). On each stall delivery, the handler uses the saved `rip` from the LDMB stall frame to index the bitmap, which translates to one shift and one lock-free, likely cache-resident load. If the current block is marked `xstate`-free, the handler skips the unnecessary `xsave` and `xrstor` and switches only the general state; otherwise it falls back to the full path, or simply gives up on this harvestable window if the workload's `xstate` overhead generally exceeds the window, which can be easily analyzed offline as well.

However, this design assumes statically compiled, symbol-rich binaries (e.g., C/C++, Rust, Go, Swift). Stripped binaries or JIT-managed runtimes (e.g., Python, Java) may not expose reliable function boundaries; supporting them require additional mechanisms and is left for future work.

Why function boundaries suffice for correctness. Under the Linux ABI [49], all `xstate` registers are *caller-saved*. A caller that needs `xstate` across a call must save and restore it explicitly; a callee may freely clobber volatile `xstate`. Consequently, `xstate` usage is strictly confined to the bodies of `xstate`-using functions and does not propagate implicitly along the call chain. Therefore, if a function's body contains no `xstate` instructions, switching without `xsave` and `xrstor` is safe. For systems that differ, notably Windows x64, where a subset of `xmm` registers are callee-saved [54], those registers must be saved and restored explicitly on every context switch.

Importantly, the `xstatedump` analysis takes a conservative approach: a code block is marked `xstate`-free only if it lies entirely within `xstate`-free functions. All other blocks (e.g., thunks/PLTs, unknown code) are treated as `xstate`-present.

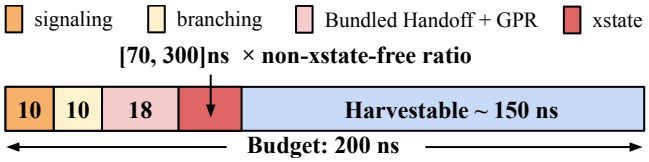


Figure 3: Cost breakdown of LITESWITCH's stall handling path. The costs for LDMB signaling and delivery are estimated. A fixed cost of 18 ns includes scavenger selection, bitmap lookup, and general registers save and restore. A variable cost is paid only when the current context is not `xstate`-free.

Practical considerations. `xstatedump` analyzes only the main program binary; libraries and other mappings default to conservative handling. A code-bitmap in 64-byte blocks balances accuracy and space, typically adding only KiBs of metadata, though some workloads are more sensitive and need a smaller block size to yield effective skipping. The bitmap is read-only and global, so lookup is lock-free. With these choices, the mechanism is simple to implement and effectively skips unnecessary `xsave` and `xrstor` operations (§6.2).

Cost estimation. The *fixed* per-handoff cost is the general-state switch plus a bitmap lookup (14 ns on average). A *variable* cost (i.e., one round of `xsave`/`xrstor`) is paid only when in non-`xstate`-free blocks. Empirically, non-compute-dense kernels take the fast path for most handoffs (§6.2). Even when `xsave`/`xrstor` is needed in every handoff, the total cost remains within the 200 ns budget for some workloads.

4.4 Putting It Together

Figure 3 summarizes the estimated costs of LITESWITCH's three core components: LDMB (detection and branching), Bundled Handoff, and `xstate`-Aware Context Switch. For workloads that require little `xsave`/`xrstor` (e.g., non-compute-dense) and with sufficient number of worker threads per core, the combined per-stall overhead is typically within **50 ns**, leaving most of a 200-ns CXL stall window available for useful scavenger work.

5 Implementation

The prototype of LITESWITCH is implemented on an Intel Emerald Rapids platform running Linux 6.8.12. The LDMB mechanism is realized through a hybrid simulation-emulation methodology (§5.1). The software components, Bundled Handoff and `xstate`-Aware Context Switch, are implemented within the Caladan user-level threading framework [22], which provides complete user-level scheduling and transparent threading semantics for unmodified applications. The resulting prototype is a software environment capable of running real, multithreaded workloads under controlled stall conditions. The artifact of LITESWITCH is open-sourced [45].

5.1 LDMB Emulation

The emulation of LDMB is implemented on a local-DRAM system using a hardware-assisted mechanism that injects stall events into a running workload. The goal is to reproduce two key properties of a real LDMB system. First, it needs to inject breakpoints into a running workload and divert execution to LITESWITCH's stall handler upon injections, as if a hardware LDMB had delivered a stall notification. Second, these injections should match the timing, frequency and duration of CXL-induced stalls observed on real CXL systems.

Repurposing PMIs for stall injection with frequency empirically derived from a real CXL system. The emulation leverages Intel's Performance Monitoring Unit (PMU), which can generate Performance Monitoring Interrupts (PMIs) upon counter overflows. Each core's PMU counter is programmed to track LLC load misses (`mem_load_retired.l3_miss`), a close proxy for DRAM-landing memory loads. The frequency at which these PMIs are triggered is the percentage of LLC load misses that would land on CXL memory, termed CXL access ratio (Table 2). This ratio is empirically derived per-workload on a CXL hardware-tiering system in Intel Flat Memory Mode with 50:50 DRAM-to-CXL ratio. For example, if the CXL access ratio is 20% for a given workload, the PMU counter overflow threshold is set such that a PMI is triggered for every 5 LLC load misses.

Upon a PMI, a kernel-space shim is invoked to inject an emulated stall, prepare an LDMB stall frame containing the user `rip` and `rflags`, and upcall the registered user-space LITESWITCH handler. The stall duration and handler stub are configured via `ioctl`. Since PMU operations are privileged, using a kernel shim as a relay is unavoidable, which introduces microarchitectural side effects (e.g., cache pollution and TLB flushes) absent in real hardware LDMB. To minimize this, the shim performs only minimal PMU bookkeeping and counter resets; its critical path is about 76 lines of code with a footprint of only a few instruction and data cache lines. Moreover, the shim disables Kernel Page Table Isolation (KPTI) and enables Process Context Identifiers (PCID), avoiding address-space switches and TLB invalidations on transitions.

Importantly, the shim's overhead is measured and subtracted from reported workload results. Because PMIs (delivered as NMIs) are non-preemptible, per-invocation cost is computed as lost wall-clock time over equivalent work and normalized by the number of PMIs per run, ensuring fair comparison across configurations with different stall rates.

The emulation also configures PMIs to fire only in user mode, as kernel context is often vulnerable to rapid, arbitrary switching due to severe reentrancy and correctness problems.

PMIs closely mirror the behavior of Intel Flat Memory Mode. PMIs are asynchronous events that arrive after the triggering LLC miss has completed. This largely mirrors how CXL accesses are identified under Intel Flat Memory Mode: the memory controller first reads the corresponding DRAM line and checks its tag to determine a DRAM miss, then immediately re-issues the request over CXL. The key difference is timing: a PMI may arrive many instructions after the triggering LLC miss, whereas in Flat Memory Mode, the tag check and CXL issuance occur back-to-back.

Config	Stall (ns)	Handler
`IFM-{len}`	`len`	none
`LITESWITCH`	20	Bundled Handoff
`SkyByte`	600	full scheduling

Table 1: System configurations. `IFM` stands for Intel Flat Memory Mode. Stall durations for `LITESWITCH` and `SkyByte` are the cost of their delivery mechanisms.

Application	Workload	CXL ratio	CXL-PKI
Breadth-First Search (`bfs`)	`urand`	24%	5.81
	`web`	12%	0.97
Connected Components (`cc`)	`urand`	25%	4.49
	`web`	17%	2.32
SPEC CPU 2017 (`spec`)	`619.lbm_s`	33%	11.57
	`638.imagick_s`	25%	0.01
	`657.xz_s`	25%	0.09
Memcached (`memcached`)	`ycsb_b`	11%	0.01
FASTER KV (`faster`)	`ycsb_a`	16%	0.90
Silo (`silo`)	`tpcc`	16%	0.09

Table 2: Applications and workloads. The CXL access ratios and CXL accesses per thousand instructions (CXL-PKI) are measured in Intel Flat Memory Mode.

Stall emulation. The shim uses Intel `tpause` to emulate stalls of specified lengths. `tpause` quiesces the hardware thread for a programmable number of cycles without microarchitectural side effects, providing a faithful approximation of a true memory stall. Importantly, `tpause` does not affect the sibling SMT hardware thread; the sibling continues to run while the other thread is `tpause`-d, enabling faithful evaluation of SMT under CXL-scale stalls (§6.1). The shim also enforces stall duration on a per-user-thread basis: if LITESWITCH resumes a thread too soon, the shim ensures the outstanding stall completes before that thread proceeds, for cases where a thread is switched back before its prior stall has resolved.

Emulating system configurations. The emulation uses three parameters to model diverse system configurations : (1) *CXL access ratio* (Table 2), the fraction of LLC misses generating injections; (2) *stall duration* (Table 1), the delay before the handler invocation, which could represent either the CXL access latency (without harvesting) or the cost for stall delivery (with harvesting); and (3) *handler activation* (Table 1), whether a software stall handler needs to be invoked.

LDMB's emulation did not factor in memory-level parallelism (MLP). As discussed in §4.1, not every LLC miss induces a pipeline stall due to MLP, but LDMB's emulation conservatively treats every CXL-bound LLC miss as a stall event, which overstates the frequency of harvestable stalls. As future work, one can measure the per-workload fraction of CXL-bound misses that overlap (i.e., a subsequent miss fires with out-of-order execution, while a prior miss is still

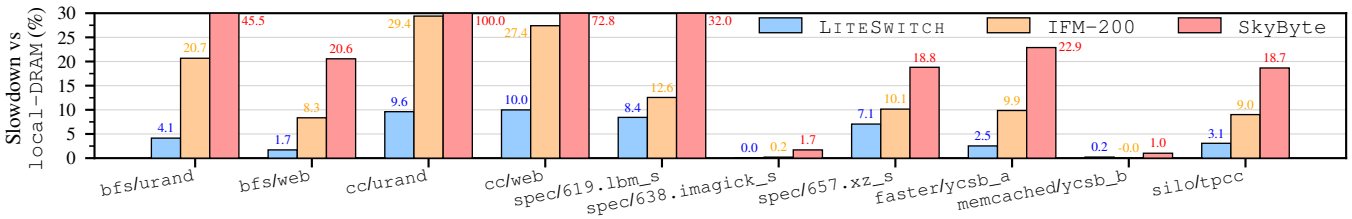


Figure 4: End-to-end application performance slowdown. All slowdown numbers are relative to `local-DRAM` with no CXL stalls.

in flight) and reduce the stall injection rate accordingly. For instance, if 10% of a workload's CXL-bound misses overlap, the emulation should inject 10% fewer stalls.

Modeling accuracy and limitations. The emulation injects stalls at a relatively flat rate and therefore does not model burstiness or short-term correlations in the miss stream. This abstraction aims at reporting end-to-end throughput under a given average CXL access ratio and stall duration. At this macro scale, the dominant factors are the total injected stall time and the fraction of that time LiteSwitch can harvest; preserving these factors should yield similar aggregate completed work even if the exact inter-stall timing differs.

This abstraction is less suitable for micro-scale timing questions. It does not faithfully model request latency, tail behavior, or concurrent misses under MLP. Studying those effects would require a methodology that preserves miss-stream burst structure and dependency behavior and is left for future work.

6 Evaluation

This section answers the following questions:

- How much performance loss induced by CXL access latency can LiteSwitch recover (§6.1.1)?
- How does LiteSwitch perform with SMT (§6.1.2)?
- Can LiteSwitch still deliver meaningful gains if the cost of LDMB is higher than the estimated 20 ns (§6.1.3)?
- How does LiteSwitch's effectiveness scale with increasing CXL latency (§6.1.4)?
- What is the runtime overhead of LiteSwitch's software path (§6.2.1, §6.2.2)?
- How effective is LiteSwitch in avoiding unnecessary `xsave` and `xrstor` operations (§6.2.3)?

Target CXL memory use case. The evaluation models the hardware tiering use case exemplified by the Intel Flat Memory Mode [6, 66], which is a relatively simple use of CXL memory as single-host memory extension where local DRAM acts as a direct-mapped cache for CXL. The CXL access ratios per workload are measured on a real CXL system in Flat Memory Mode configured with a 50:50 DRAM-to-CXL ratio (Table 2). This setting represents the conservative "lower end" of CXL memory stalls, where more complex configurations like CXL memory pools likely exhibit higher CXL latencies. The emulation and evaluation of LiteSwitch runs on a different system with the Intel Xeon Gold 5512U and only local DRAM, using the empirically derived CXL ratios to reproduce realistic stall behavior.

Applications and workloads. The experiments cover a diverse set of memory-intensive workloads (Table 2) representative of several key domains: (1) sparse graph analytics from the GAP Benchmark Suite [9]; (2) dense numerical computation from SPEC CPU 2017 [4]; (3) in-memory key-value stores (Memcached [3] and FASTER KV [12]); and (4) in-memory databases (Silo [58]). Each application-workload pair is denoted as `app/workload`.

System configurations. Four system configurations are compared in end-to-end experiments (Table 1): (1) `local-DRAM`, an oracle baseline assuming infinite local DRAM capacity and no CXL stalls; (2) `IFM-{len}`, a tiered-memory configuration using Intel Flat Memory Mode, with CXL access latency of `len` nanoseconds; (3) `LiteSwitch`, the proposed system with a 20 ns LDMB delivery cost; and (4) `SkyByte`, prior work that uses interrupts for stall delivery and takes a full scheduling path for stall handling [65].

All configurations disable SMT (unless specified otherwise) and use an equal number of physical cores. The number of worker threads per core is chosen independently for each configuration that yields the best performance. Throughput is used as the end-to-end metric, since LiteSwitch's primary effect is improving aggregate CPU utilization.

6.1 End-to-End Performance

Overall, LiteSwitch consistently reduces end-to-end performance slowdowns across a variety of workloads, with or without SMT, and under higher LDMB cost and CXL latency.

6.1.1 End-to-End Slowdown Across Workloads

Figure 4[1] reports the performance slowdown of each configuration relative to the `local-DRAM` oracle across all evaluated workloads. LiteSwitch consistently reduces slowdown compared to `IFM-200` for every workload, empirically validating that the mechanism can extract meaningful harvesting gains even at the lower end of real CXL latencies (200 ns).

LiteSwitch reduces slowdown by 30–80%[2] relative to `IFM-200` for most workloads: `bfs` sees a much smaller 1.7–4.1% slowdown versus 8–21% under `IFM-200`; in-memory

[1] For all figures in the evaluation section, lower values on the y-axis indicate better outcomes.

[2] Throughout this paper, phrases like "reduce slowdown by x%" and "x% reduction" denote *relative* reduction, not absolute percentage-point change.

Figure 5: Individual and combined effects of LiteSwitch and SMT. Completion time measures the time to complete the same amount of work, and is normalized to the baseline without LiteSwitch or SMT. All configurations use a 200-ns CXL latency setting. SMT uses a 2× hyperthreading factor. The right y-axis marks the corresponding speedup, with 2× being the perfect speedup from 2× hyperthreading.

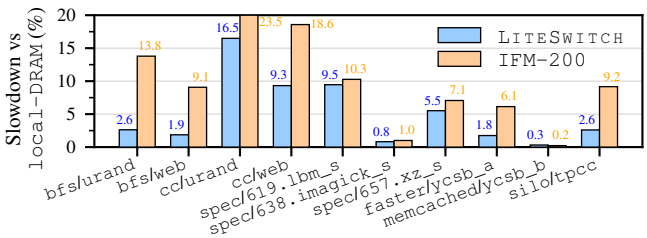


Figure 6: End-to-end slowdown with SMT. All configurations are measured with SMT turned on. SkyByte is excluded from this figure because it yields consistently the worst slowdowns.

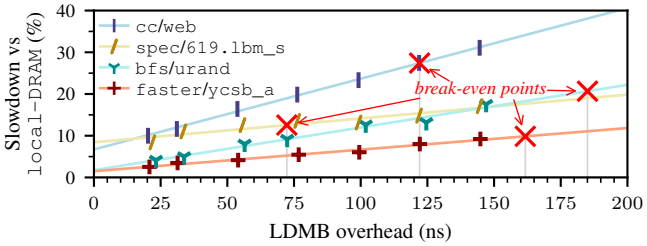


Figure 7: LiteSwitch effectiveness under higher LDMB overhead, ranging roughly 20–150 ns at 200 ns CXL latency. The lines are linear fit. The red break-even points mark where each fit matches the corresponding workload's IFM-200 slowdown.

stores, faster and silo, see 2.5–3.1% versus 9–10%. These workloads all remain under 5% slowdown, enabling practical CXL memory adoption. memcached and 638.imagick_s are particularly cache-friendly with negligible slowdowns for all configurations. cc is more sensitive to memory latency, exhibiting 9.6–10% slowdown in LiteSwitch, but still reduces slowdown by roughly 65% relative to IFM-200's 27–29%.

SPEC workloads (619.lbm_s, 657.xz_s) show higher slowdowns of 8.4% and 7.1%, respectively, with a more modest 30% reduction over IFM-200. These benchmarks make heavy use of SIMD/FP instructions, preventing xstate-Aware Context Switch from skipping xsave/xrstor on most context switches. Even so, LiteSwitch remains beneficial. SkyByte exhibits the worst performance by a wide margin across all workloads. This is because its interrupt-based delivery incurs a ∼600 ns overhead inappropriate for harvesting CXL memory stalls.

6.1.2 LiteSwitch with SMT

Figure 6 reports slowdown with SMT enabled. LiteSwitch continues to provide substantial slowdown reduction: graph workloads and KV engines see comparable reduction (30–80%) to the no-SMT case. By contrast, SPEC workloads 619.lbm_s and 657.xz_s show smaller reductions of 7% and 22%, versus 33% and 31% without SMT. The likely cause is higher intra-core contention inherent to SMT: by un-stalling hardware threads, LiteSwitch increases the fraction of time both SMT siblings are active contending for core resources. SMT alone also partially mitigates IFM-200 stalls. For example, cc workloads slow down by 27–29% without SMT and 18–23% with SMT. While SMT can hide some CXL memory stalls, the benefits of LiteSwitch are complementary to SMT, since LiteSwitch 'un-stalls' SMT hardware threads that are blocked on CXL access.

To quantify the individual and combined effects, Figure 5 compares the normalized completion time (and the corresponding speedup) under a baseline (no LiteSwitch, no SMT), LiteSwitch alone, SMT alone, and both combined.

LiteSwitch and SMT each improve performance independently, and their benefits largely compose when combined. For example, bfs/urand speeds up by 1.16× with LiteSwitch alone, 1.69× with SMT alone, and 1.87× when both are enabled. A similar pattern holds for graph workloads and KV engines. For most workloads, SMT alone provides higher speedup than LiteSwitch alone. This is expected: SMT enables true hardware concurrency at the instruction level, whereas LiteSwitch only opportunistically harvests stall cycles when CXL-bound misses occur. They both show considerable variation across workloads. 638.imagick_s is the extreme case where SMT sees little speedup, likely due to intense contention between SMT siblings.

6.1.3 Effectiveness under Higher Cost of LDMB

Figure 7 isolates the effect of increasing LDMB delivery cost while holding the target CXL latency at 200 ns. Across workloads, higher LDMB overhead shifts LiteSwitch's slowdown upward by a fairly fixed per-event penalty. It does not introduce a disproportionate or qualitatively different failure mode: the fitted lines remain close to linear, indicating that the added cost is paid per delivered stall event.

The marked *break-even points* indicate where each fitted LiteSwitch slowdown matches the corresponding workload's IFM-200 slowdown. Beyond this point, increasing LDMB overhead starts to eliminate LiteSwitch's net end-

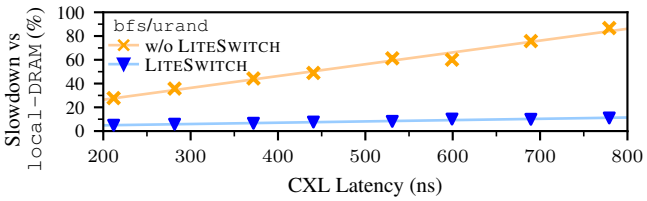


Figure 8: LiteSwitch effectiveness under varying CXL latencies (bfs/urand), ranging roughly 200–800 ns. Lines are linear fit.

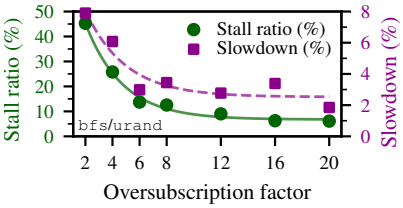


Figure 9: Oversubscription reduces stall ratio and slowdown (bfs/urand). The curves are exponential fit.

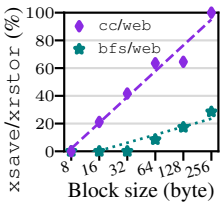


Figure 10: Sensitivity of bitmap block size.

to-end gain. The x-axis value varies substantially across workloads because the software-path overhead differs by workload (§6.2). For instance, spec/619.lbm_s reaches break-even near 72 ns, whereas bfs/urand remains beneficial until roughly 185 ns. This gap reflects their different software-path costs: bfs/urand rarely pays for expensive xsave/xrstor because it exercises much less xstate than spec/619.lbm_s (§6.2.3). These break-even values can be interpreted as the available LDMB-overhead "headroom" for preserving positive end-to-end gains.

The fitted slopes also differ across workloads. A steeper slope means that slowdown is more sensitive to additional per-event cost, which largely tracks how sensitive the workload is to memory-access latency in the first place.

6.1.4 Effectiveness under Varying CXL Latency

Figure 8 examines how slowdown evolves as CXL latency increases from roughly 200 ns to 800 ns, using bfs/urand as a representative workload (all other workloads see the same qualitative pattern). Without LiteSwitch, slowdown grows roughly linearly with CXL latency: longer memory stalls inject proportionally more idle time per access.

With LiteSwitch, slowdown remains nearly flat across the tested range. The slight upward trend arises because a bundle can occasionally contain only one runnable thread, leaving no scavenger available; those stalls must be waited out in full. Empirically, such single-thread cases are mostly fewer than 10%, so the aggregate impact is small (§6.2.1).

The key observation is that overall, LiteSwitch pays a rather *fixed* per-stall cost per workload, largely independent of the underlying CXL latency. This highlights LiteSwitch's potential to convert a wide range of CXL latencies into a fixed, per-invocation cost in perceived progress, yielding consistently low slowdowns even under greater CXL device variability and deeper topologies.

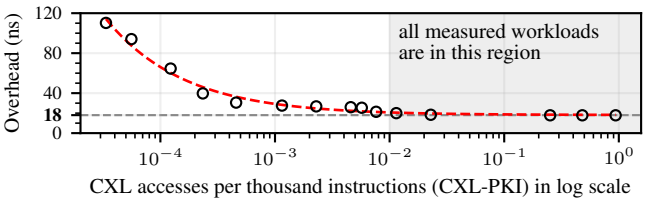


Figure 11: Handler overhead (excluding xsave/xrstor). The data points are measured on a synthetic matrix multiplication workload where CXL-PKI can be easily tuned; The 18-ns stable cost is verified on all workloads. The curve is exponential fit.

6.2 Overhead of the Software Path

The per-invocation cost of LiteSwitch's software path has three parts: (1) unharvestable stalls when the bundle has only one runnable thread (the stall is waited out); (2) a fixed handler cost, including scavenger selection, xstatedump bitmap lookup, and general-register save and restore; and (3) a variable cost for xsave and xrstor, paid only when the current context is not xstate-free.

6.2.1 Thread Oversubscription

A strong empirical correlation exists between the stall ratio (the fraction of LiteSwitch invocations with only one runnable thread in the bundle) and the thread oversubscription factor (total software worker threads divided by CPU hardware threads). For bfs/urand (Figure 9), at 2× oversubscription, 45% of invocations have only one runnable thread because workers frequently block on synchronization; as oversubscription increases, the stall ratio falls quickly (to 12–14% at 6–8×) and then plateaus. Workload slowdown closely tracks this decline. In practice, about 4–8× oversubscription sharply reduces the stall ratio, and roughly 6–12× keeps a scavenger available for >90% of stalls for most workloads.

Oversubscription is a general multi-threaded tuning knob rather than a requirement specific to LiteSwitch: too little oversubscription risks under-utilization when threads block, while too much increases contention and scheduler overhead. LiteSwitch benefits when the workload provides enough runnable threads to exploit the exposed stall windows.

6.2.2 Handler Overhead Excluding xstate

Figure 11 reports the fixed overhead of LiteSwitch's handler (scavenger selection, general-register save/restore, and xstatedump bitmap lookup) as a function of trigger rate, expressed as CXL accesses per thousand instructions (CXL-PKI). At very low rates ($< 10^{-4}$), overhead exceeds 100 ns; it then drops sharply and plateaus near 18 ns once the rate reaches 10^{-2}. The effect is due to cache residency: at higher frequencies, the handler's memory footprint (bundle metadata, bitmap, context frames) stays hot between invocations. All evaluated workloads operate at or above 10^{-2} CXL-PKI (Table 2) and often much higher. Thus, 18 ns is the effective fixed handler cost in practice.

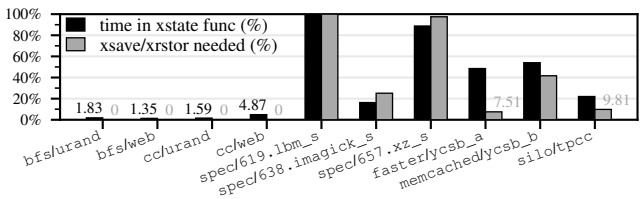


Figure 12: Effectiveness of `xstate`-aware skipping. The left bar measures the execution time spent in `xstate`-using functions; the right bar measures the fraction of handoffs requiring `xsave`/`xrstor`.

6.2.3 Effectiveness of `xstate`-Awareness

Figure 12 compares time spent in `xstate`-using functions with the fraction of LITESWITCH context switches that require `xsave`/`xrstor`. Across workloads, these two measurements are largely equivalent, indicating that `xstate`-aware skipping is effective. A few cases deviate: `faster`/`ycsb_a` requires far fewer `xsave`/`xrstor` than its time in `xstate` functions. This discrepancy is because `xstate`-heavy functions tend to be compute-bound with few memory loads, so stalls are disproportionately triggered in non-`xstate` code paths.

Figure 10 shows how the granularity of `xstatedump` bitmap impacts the fraction of context switches that require `xsave`/`xrstor` for two `xstate`-light workloads. For `cc`/`web`, the default 64-byte blocks still pay `xsave`/`xrstor` for more than half of the switches, whereas 8-byte blocks reduce this to near zero. This is likely because `cc` has a scattered `xstate` usage pattern that widely taints blocks when the granularity is coarse. `bfs`/`web` is less sensitive but follows the same trend. Empirically, 64 bytes suffices for most workloads.

7 Discussions and Limitations

The applicability of LITESWITCH. LDMB detects CXL accesses by relying on the CPU memory subsystem's cache-miss and routing path. Regardless of topology (e.g., direct-attach, multi-headed, switched), on-chip routing is the earliest point in the request path that can reliably determine whether a load will go off-chip to CXL. Measurements show that LITESWITCH's mechanisms are already effective for the simplest memory-expansion case with 200 ns CXL latency; as latencies grow (e.g., deeper fabrics, contention, heterogeneous devices), the recoverable stall window only increases. While there is ongoing debate about which CXL deployments are actually viable [43], LITESWITCH remains broadly useful and benefits more as stalls lengthen.

Considerations for Intel Flat Memory Mode. In this simple setup, the memory controller must read in the corresponding DRAM line first to determine whether an access missed DRAM. With this model, while LDMB can still make detection before a CXL access is issued, thus leaving the full latency window for harvesting, detection is still technically delayed by a full DRAM read. In deployments where CXL devices are directly exposed rather than a slower caching tier, detection does not need a full leading DRAM read and can use the high-order bits of the physical address. The speculation is that LITESWITCH could even outperform the oracle case (i.e., infinite local DRAM), because LITESWITCH's detection and harvesting routine begin at roughly the same moment a DRAM or CXL request is sent off-chip. Therefore, if LITESWITCH can switch to a scavenger faster than a DRAM load completes, LITESWITCH observes stalls even shorter than DRAM in perceived progress, thus better performance.

Potential real-world effects of LITESWITCH on memory-level parallelism (MLP). LDMB's pipeline flush might reduce Out-of-Order (OoO) depth, thus decreasing MLP. As discussed in §4.1, however, LDMB's return signal arrives only after a load traverses the cache hierarchy and is detected as CXL-bound; this path is at least LLC-hit-scale. During that interval, the core continues dispatching independent instructions, and prior studies show that OoO cores typically exhaust their instruction window within that timescale [40,41]. Consequently, data-independent loads that could exploit MLP have typically already been issued before LDMB can redirect, so the pipeline flush sacrifices little MLP opportunity. Critically, Bundled Handoff allows additional threads to issue their own independent memory loads while the original miss is in flight, effectively widening the core's useful miss concurrency beyond what a single stalled thread could achieve.

Double-emulation methodology. LITESWITCH emulates both LDMB delivery and CXL memory latency, compounding uncertainty into the evaluation. Two discrepancies matter most. First, the emulation injects stalls at fixed intervals derived from each workload's CXL access ratio. Real inter-miss intervals are more variable, and closely spaced misses can overlap through MLP rather than producing separate pipeline stalls. This can overstate the number of harvestable events and therefore modestly overstate LITESWITCH's absolute gain. Second, the evaluation uses 200 ns CXL latency, representative of the simplest direct-attached CXL. Deeper topologies (e.g., multi-headed, switched) and systems under load (e.g., in Intel Flat Memory Mode, every DRAM miss doubles bandwidth demand) push real latencies higher. Because LITESWITCH pays a mostly fixed per-stall cost, longer stalls widen the harvestable window; in those settings, the 200 ns results likely understate potential gains.

8 Related Work

Switch-on-event (SOE) multithreading. SOE is a hardware scheme from earlier multithreaded processors that switches between hardware threads when a retirement-blocking stall occurs [5, 11, 24, 53]. SOE shares the basic intuitions, but it always performs a costly full context switch with all architectural state. While this cost was acceptable for the more limited architectural state of older processors, modern CPUs continue to expand `xstate` (e.g., AVX-512 and beyond), making full-state switching increasingly prohibitive. LITESWITCH reduces this cost by selectively saving

xstate only when needed. Besides, LITESWITCH triggers only when a CXL-bound miss is determined, which selects for stalls with a sufficiently long expected latency, rather than committing to potentially short stalls when the hardware does not distinguish the root cause of a retirement-blocking stall.

Software-based harvesting. MSH [50] harvests DRAM-scale stalls within a single address space and uses fine-grained register-liveness analysis to minimize general-register save/restore. However, its reliance on offline profiling to place yield points is brittle in CXL. SkyByte [65] targets CXL-SSD latencies (tens-hundreds of µs) using device-provided hints delivered as interrupts so the OS can preempt and reschedule. That interrupt-driven path is appropriate at SSD timescales but is too costly for sub-µs CXL memory stalls. These limitations motivate LITESWITCH's combination of reactive, per-access hardware detection with same-address-space handoff.

Mitigating SIMD/FP overhead in context switch. In µs-scale user-level scheduling, context-switch overhead can also dominate. Shinjuku [36] made a similar observation that contexts that do not use SIMD/FP need not preserve xstate, and arranges its scheduler worker to be xstate-free so that switches between a user context and the worker omit FP save in one direction and FP restore in the other. However, an end-to-end handoff between two user contexts still incurs a full round of xstate save and restore. LITESWITCH's xstate-Aware Context Switch instead skips xsave/xrstor for direct handoffs between two user contexts.

Existing hardware capabilities for hiding memory stalls. Modern Intel server-class CPUs employ aggressive microarchitectural techniques such as out-of-order (OoO) execution, speculative execution, and prefetching to overlap backend stalls with useful work. These mechanisms can hide some memory latency, but are fundamentally limited by bounded resources such as the reorder buffer (ROB), issue queues, and the number of outstanding cache misses tracked by MSHRs. Long-latency DRAM accesses typically exceed what the ROB can cover, and prefetchers often fail on irregular access patterns or cause contention. Prior work has quantified these limitations and shown that cores still frequently stall despite deep speculation and scheduling windows [7, 20, 40, 41].

Asynchronous memory semantics. Asynchronous memory access instructions (AMIs) [60] decouple memory request issuance from completion in hardware, exposing explicit memory-level parallelism to the software but requiring a radical asynchronous execution model. Beehive [46] applies a similar principle in software for disaggregated memory, using compiler-generated Rust coroutines to overlap µs-scale remote accesses while preserving a synchronous-style programming model. At a high level, both AMI and Beehive hide memory latency by fitting in more concurrent outstanding memory requests through asynchrony. LITESWITCH is orthogonal: it targets cases where memory-level parallelism may still fall short, turning true pipeline stalls into useful work by intentionally switching to another worker thread.

9 Conclusions

LITESWITCH is a lightweight system for stall detection, delivery, and scheduling that effectively harvests sub-µs CXL memory stalls. LITESWITCH uses a combination of a simple hardware-based detection method with lightweight branching into the same user address space, together with fast scavenger selection while avoiding the cost of saving and restoring xstate registers when not necessary. Importantly, the ability to sustain consistently low workload slowdowns largely independent of CXL latency variability across diverse workloads highlights LITESWITCH's broad applicability.

Acknowledgements

The authors thank the anonymous shepherd and reviewers for their constructive feedback that greatly improved this paper. The authors also thank Leon Schuermann, Kaifeng Xu, Haoda Wang, Daniel S. Berger, Kostis Kaffes, and Brian Hirano for their helpful discussions and feedback on this work. The implementation and evaluation of LITESWITCH were conducted on resources generously provided by CloudLab [19].

References

[1] 5th Gen AMD EPYC™ Processor Architecture. https://www.amd.com/content/dam/amd/en/documents/epyc-business-docs/white-papers/5th-gen-amd-epyc-processor-architecture-white-paper.pdf. Accessed: 2026-05-20.

[2] Intel® Xeon® 6 Processors. https://www.intel.com/content/www/us/en/products/details/processors/xeon/xeon6-e-cores.html. Accessed: 2026-05-20.

[3] memcached: A distributed memory object caching system. https://memcached.org/. Accessed: 2025-12-09.

[4] SPEC CPU® 2017 Benchmark Suite. https://www.spec.org/cpu2017/, 2017. Standard Performance Evaluation Corporation.

[5] A. Agarwal, J. Kubiatowicz, D. Kranz, B.H. Lim, D. Yeung, G. D'Souza, and M. Parkin. Sparcle: an evolutionary processor design for large-scale multiprocessors. *IEEE Micro*, 13(3):48–61, 1993.

[6] Minseon Ahn, Thomas Willhalm, Donghun Lee, Norman May, Jungmin Kim, Daniel Ritter, and Oliver Rebholz. Exploiting Locality in Flat Memory with CXL for In-Memory Database Management Systems. In *Proceedings of the 21st International Workshop on Data Management on New Hardware*, DaMoN '25, New York, NY, USA, 2025. Association for Computing Machinery.

[7] Newsha Ardalani, Clint Lestourgeon, Karthikeyan Sankaralingam, and Xiaojin Zhu. Cross-architecture performance prediction (xapp) using cpu code to predict gpu performance. In *Proceedings of the 48th International Symposium on Microarchitecture*, MICRO-48, page 725–737, New York, NY, USA, 2015. Association for Computing Machinery.

[8] Berk Aydogmus, Linsong Guo, Danial Zuberi, Tal Garfinkel, Dean Tullsen, Amy Ousterhout, and Kazem Taram. *Extended User Interrupts (xUI): Fast and Flexible Notification without Polling*, page 373–389. Association for Computing Machinery, New York, NY, USA, 2025.

[9] Scott Beamer, Krste Asanovic, and David A. Patterson. The GAP Benchmark Suite. *arXiv*, 1508.03619 [cs.DC], 2015.

[10] Daniel S. Berger, Yuhong Zhong, Fiodar Kazhamiaka, Pantea Zardoshti, Shuwei Teng, Mark D. Hill, and Rodrigo Fonseca. Octopus: Scalable Low-Cost CXL Memory Pooling. Microsoft Technical Report, 2025.

[11] J. M. Borkenhagen, R. J. Eickemeyer, R. N. Kalla, and S. R. Kunkel. A multithreaded PowerPC processor for commercial servers. *IBM J. Res. Dev.*, 44(6):885–898, November 2000.

[12] Badrish Chandramouli, Guna Prasaad, Donald Kossmann, Justin Levandoski, James Hunter, and Mike Barnett. FASTER: A Concurrent Key-Value Store with In-Place Updates. In *Proceedings of the 2018 International Conference on Management of Data*, SIGMOD '18, page 275–290, New York, NY, USA, 2018. Association for Computing Machinery.

[13] Derek Chiou, Dam Sunwoo, Joonsoo Kim, Nikhil A. Patil, William Reinhart, Darrel Eric Johnson, Jebediah Keefe, and Hari Angepat. FPGA-Accelerated Simulation Technologies (FAST): Fast, Full-System, Cycle-Accurate Simulators. In *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 40, page 249–261, USA, 2007. IEEE Computer Society.

[14] Jiho Choi, Thomas Shull, Maria J. Garzaran, and Josep Torrellas. ShortCut: Architectural Support for Fast Object Access in Scripting Languages. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*, ISCA '17, page 494–506, New York, NY, USA, 2017. Association for Computing Machinery.

[15] Compute Express Link Consortium. CXL Integrators List. https://www.computeexpresslink.org/integrators-list, 2025. Accessed: 2025-12-09.

[16] Intel Corporation. CXL Type 3 Memory Device Software Guide, Rev. 1.1. https://cdrdv2-public.intel.com/643805/643805_CXL_Memory_Device_SW_Guide_Rev1_1.pdf, August 2024.

[17] Debendra Das Sharma, Robert Blankenship, and Daniel Berger. An Introduction to the Compute Express Link (CXL) Interconnect. *ACM Comput. Surv.*, 56(11), July 2024.

[18] Aniket Deshmukh, LingzheChester Cai, and Yale N. Patt. Timely, Efficient, and Accurate Branch Precomputation. In *2024 57th IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 480–492, 2024.

[19] Dmitry Duplyakin, Robert Ricci, Aleksander Maricq, Gary Wong, Jonathon Duerig, Eric Eide, Leigh Stoller, Mike Hibler, David Johnson, Kirk Webb, Aditya Akella, Kuangching Wang, Glenn Ricart, Larry Landweber, Chip Elliott, Michael Zink, Emmanuel Cecchet, Snigdhaswin Kar, and Prabodh Mishra. The design and operation of CloudLab. In *Proceedings of the USENIX Annual Technical Conference (ATC)*, pages 1–14, July 2019.

[20] Stijn Eyerman and Lieven Eeckhout. System-level performance metrics for multiprogram workloads. *IEEE Micro*, 28(3):42–53, May 2008.

[21] Michael Ferdman, Almutaz Adileh, Onur Kocberber, Stavros Volos, Mohammad Alisafaee, Djordje Jevdjic, Cansu Kaynak, Adrian Daniel Popescu, Anastasia Ailamaki, and Babak Falsafi. Clearing the clouds: a study of emerging scale-out workloads on modern hardware. In *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVII, page 37–48, New York, NY, USA, 2012. Association for Computing Machinery.

[22] Joshua Fried, Zhenyuan Ruan, Amy Ousterhout, and Adam Belay. Caladan: Mitigating Interference at Microsecond Timescales. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 281–297. USENIX Association, November 2020.

[23] Justin R. Funston, Kaoutar El Maghraoui, Joefon Jann, Pratap Pattnaik, and Alexandra Fedorova. An SMT-Selection Metric to Improve Multithreaded Applications' Performance. In *2012 IEEE 26th International Parallel and Distributed Processing Symposium*, pages 1388–1399, 2012.

[24] Ron Gabor, Shlomo Weiss, and Avi Mendelson. Fairness enforcement in switch on event multithreading.

ACM Trans. Archit. Code Optim., 4(3):15–es, September 2007.

[25] Donghyun Gouk, Sangwon Lee, Miryeong Kwon, and Myoungsoo Jung. Direct Access, High-Performance Memory Disaggregation with DirectCXL. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, pages 287–294, Carlsbad, CA, July 2022. USENIX Association.

[26] Linsong Guo, Danial Zuberi, Tal Garfinkel, and Amy Ousterhout. The benefits and limitations of user interrupts for preemptive userspace scheduling. In *22nd USENIX Symposium on Networked Systems Design and Implementation (NSDI 25)*, pages 1015–1032, Philadelphia, PA, April 2025. USENIX Association.

[27] Vincent Haché and Jérôme Glisse. Introduction to CXL Multi-Headed Devices. OCP Global Summit, 2022.

[28] Mark Horowitz, Margaret Martonosi, Todd C. Mowry, and Michael D. Smith. Informing memory operations: providing memory performance feedback in modern processors. In *Proceedings of the 23rd Annual International Symposium on Computer Architecture*, ISCA '96, page 260–270, New York, NY, USA, 1996. Association for Computing Machinery.

[29] Kaisong Huang, Tianzheng Wang, Qingqing Zhou, and Qingzhong Meng. The Art of Latency Hiding in Modern Database Engines. *Proc. VLDB Endow.*, 17(3):577–590, November 2023.

[30] Wentao Huang, Mo Sha, Mian Lu, Yuqiang Chen, Bingsheng He, and Kian-Lee Tan. Bandwidth Expansion via CXL: A Pathway to Accelerating In-Memory Analytical Processing. In *VLDB 2024 Workshop: Fifteenth International Workshop on Accelerating Analytics and Data Management Systems Using Modern Processor and Storage Architectures*, 2024.

[31] Yibo Huang, Haowei Chen, Newton Ni, Yan Sun, Vijay Chidambaram, Dixin Tang, and Emmett Witchel. Tigon: A Distributed Database for a CXL Pod. In *Proceedings of the 19th USENIX Conference on Operating Systems Design and Implementation*, OSDI '25, USA, 2025. USENIX Association.

[32] Jack Tigar Humphries, Neel Natu, Ashwin Chaugule, Ofir Weisse, Barret Rhoden, Josh Don, Luigi Rizzo, Oleg Rombakh, Paul Turner, and Christos Kozyrakis. ghOSt: Fast & Flexible User-Space Delegation of Linux Scheduling. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, SOSP '21, page 588–604, New York, NY, USA, 2021. Association for Computing Machinery.

[33] Elyse Ge Hylander and Phyllis Ng. Azure delivers the first cloud VM with Intel Xeon 6 and CXL memory - now in Private Preview. https://techcommunity.microsoft.com/blog/SAPApplications/azure-delivers-the-first-cloud-vm-with-intel-xeon-6-and-cxl-memory--now-in-priv/4470067, November 2025. Accessed: 2025-12-04.

[34] Intel Corporation. Intel® 64 and IA-32 Architectures Software Developer Manuals. https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html, October 2025.

[35] Sunita Jain, Nagaradhesh Yeleswarapu, Hasan Al Maruf, and Rita Gupta. Memory Sharing with CXL: Hardware and Software Design Approaches. In *Proceedings of the 3rd Workshop on Heterogeneous Composable and Disaggregated Systems*, HCDS 2024, 2024.

[36] Kostis Kaffes, Timothy Chong, Jack Tigar Humphries, Adam Belay, David Mazières, and Christos Kozyrakis. Shinjuku: Preemptive Scheduling for µsecond-scale Tail Latency. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 345–360, Boston, MA, February 2019. USENIX Association.

[37] Svilen Kanev, Juan Pablo Darago, Kim Hazelwood, Parthasarathy Ranganathan, Tipp Moseley, Gu-Yeon Wei, and David Brooks. Profiling a warehouse-scale computer. *SIGARCH Comput. Archit. News*, 43(3S):158–169, June 2015.

[38] Sagar Karandikar, Howard Mao, Donggyu Kim, David Biancolin, Alon Amid, Dayeol Lee, Nathan Pemberton, Emmanuel Amaro, Colin Schmidt, Aditya Chopra, Qijing Huang, Kyle Kovacs, Borivoje Nikolic, Randy Katz, Jonathan Bachrach, and Krste Asanovic. FireSim: FPGA-Accelerated Cycle-Exact Scale-Out System Simulation in the Public Cloud. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, pages 29–42, 2018.

[39] Asif Imtiaz Khan. *Cycle-Accurate Modeling of Multicore Processors on FPGAs*. PhD thesis, Massachusetts Institute of Technology, USA, 2013. AAI0829707.

[40] Vladimir Kiriansky, Ilia Lebedev, Saman Amarasinghe, Srinivas Devadas, and Joel Emer. Dawg: A defense against cache timing attacks in speculative execution processors. In *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 974–987, 2018.

[41] Vladimir Kiriansky, Haoran Xu, Martin Rinard, and Saman Amarasinghe. Cimple: instruction and memory level parallelism: a dsl for uncovering ilp and mlp. In *Proceedings of the 27th International Conference on Parallel Architectures and Compilation Techniques*, PACT '18, New York, NY, USA, 2018. Association for Computing Machinery.

[42] V. Krishnan and J. Torrellas. A clustered approach to multithreaded processors. In *Proceedings of the First Merged International Parallel Processing Symposium and Symposium on Parallel and Distributed Processing*, pages 627–634, 1998.

[43] Philip Levis, Kun Lin, and Amy Tai. A Case Against CXL Memory Pooling. In *Proceedings of the 22nd ACM Workshop on Hot Topics in Networks*, HotNets '23, page 18–24, New York, NY, USA, 2023. Association for Computing Machinery.

[44] Huaicheng Li, Daniel S. Berger, Lisa Hsu, Daniel Ernst, Pantea Zardoshti, Stanko Novakovic, Monish Shah, Samir Rajadnya, Scott Lee, Ishwar Agarwal, Mark D. Hill, Marcus Fontoura, and Ricardo Bianchini. Pond: CXL-Based Memory Pooling Systems for Cloud Platforms. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, ASPLOS 2023, page 574–587, New York, NY, USA, 2023. Association for Computing Machinery.

[45] Nanqinqin Li. LiteSwitch Artifact. https://github.com/princeton-sns/liteswitch.

[46] Quanxi Li, Hong Huang, Ying Liu, Yanwen Xia, Jie Zhang, Mosong Zhou, Xiaobing Feng, Huimin Cui, Quan Chen, Yizhou Shan, and Chenxi Wang. Beehive: A Scalable Disaggregated Memory Runtime Exploiting Asynchrony of Multithreaded Programs. In *22nd USENIX Symposium on Networked Systems Design and Implementation (NSDI 25)*, pages 167–187, Philadelphia, PA, April 2025. USENIX Association.

[47] Shaobo Li, Yirui (Eric) Zhou, Hao Ren, and Jian Huang. ByteFS: System Support for (CXL-based) Memory-Semantic Solid-State Drives. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1*, ASPLOS '25, page 116–132, New York, NY, USA, 2025. Association for Computing Machinery.

[48] Jinshu Liu, Hamid Hadian, Yuyue Wang, Daniel S. Berger, Marie Nguyen, Xun Jian, Sam H. Noh, and Huaicheng Li. *Systematic CXL Memory Characterization and Performance Analysis at Scale*, page 1203–1217. Association for Computing Machinery, New York, NY, USA, 2025.

[49] H.J. Lu, Michael Matz, Milind Girkar, Jan Hubička, Andreas Jaeger, and Mark Mitchell. System V Application Binary Interface: AMD64 Architecture Processor Supplement, March 2025.

[50] Zhihong Luo, Sam Son, Sylvia Ratnasamy, and Scott Shenker. Harvesting Memory-bound CPU Stall Cycles in Software with MSH. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, pages 57–75, Santa Clara, CA, July 2024. USENIX Association.

[51] Haoran Ma, Yifan Qiao, Shi Liu, Shan Yu, Yuanjiang Ni, Qingda Lu, Jiesheng Wu, Yiying Zhang, Miryung Kim, and Harry Xu. DRust: Language-Guided Distributed Shared Memory with Fine Granularity, Full Transparency, and Ultra Efficiency. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, pages 97–115, Santa Clara, CA, July 2024. USENIX Association.

[52] Teng Ma, Zheng Liu, Chengkun Wei, Jialiang Huang, Youwei Zhuo, Haoyu Li, Ning Zhang, Yijin Guan, Dimin Niu, Mingxing Zhang, and Tao Ma. HydraRPC: RPC in the CXL Era. In *2024 USENIX Annual Technical Conference (USENIX ATC 24)*, pages 387–395, Santa Clara, CA, July 2024. USENIX Association.

[53] C. McNairy and R. Bhatia. Montecito: a dual-core, dual-thread Itanium processor. *IEEE Micro*, 25(2):10–20, 2005.

[54] Microsoft. Overview of x64 ABI conventions. https://learn.microsoft.com/en-us/cpp/build/x64-software-conventions?view=msvc-170. Accessed: 2025-12-07.

[55] Pacific Northwest National Laboratory. Unique Active Memory Computer Purpose-Built for AI Science Applications. https://www.pnnl.gov/news-media/unique-active-memory-computer-purpose-built-ai-science-applications, August 2025. Accessed: 2025-12-04.

[56] Arthur Perais and André Seznec. Practical data value speculation for future high-end processors. In *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*, pages 428–439, 2014.

[57] XConn Technologies. Overcoming the AI Memory Wall: How CXL Memory Pooling Powers the Next Leap in Scalable AI Computing. https://computeexpresslink.org/blog/

overcoming-the-ai-memory-wall-how-cxl-memory-pooling-powers-the-next-leap-in-scalable-ai-computing-4267/, November 2025. Accessed: 2025-12-04.

[58] Stephen Tu, Wenting Zheng, Eddie Kohler, Barbara Liskov, and Samuel Madden. Speedy transactions in multicore in-memory databases. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, page 18–32, New York, NY, USA, 2013. Association for Computing Machinery.

[59] N. Tuck and D.M. Tullsen. Initial observations of the simultaneous multithreading pentium 4 processor. In *2003 12th International Conference on Parallel Architectures and Compilation Techniques*, pages 26–34, 2003.

[60] Luming Wang, Xu Zhang, Songyue Wang, Zhuolun Jiang, Tianyue Lu, Mingyu Chen, Siwei Luo, and Keji Huang. Asynchronous memory access unit: Exploiting massive parallelism for far memory access. *ACM Trans. Archit. Code Optim.*, 21(3), September 2024.

[61] Kaifeng Xu, Georgios Tziantzioulis, and David Wentzlaff. Evaluation of MindPalace for Chip Design Tradeoffs on Function-as-a-Service. In *2025 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 201–212, 2025.

[62] Juncheng Yang, Yao Yue, and Rashmi Vinayak. Segcache: a memory-efficient and scalable in-memory key-value cache for small objects. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*, pages 503–518. USENIX Association, April 2021.

[63] Qirui Yang, Runyu Jin, Bridget Davis, Devasena Inupakutika, and Ming Zhao. Performance Evaluation on CXL-enabled Hybrid Memory Pool. In *2022 IEEE International Conference on Networking, Architecture and Storage (NAS)*, pages 1–5, 2022.

[64] Shao-Peng Yang, Minjae Kim, Sanghyun Nam, Juhyung Park, Jin yong Choi, Eyee Hyun Nam, Eunji Lee, Sungjin Lee, and Bryan S. Kim. Overcoming the Memory Wall with CXL-Enabled SSDs. In *2023 USENIX Annual Technical Conference (USENIX ATC 23)*, pages 601–617, Boston, MA, July 2023. USENIX Association.

[65] Haoyang Zhang, Yuqi Xue, Yirui Eric Zhou, Shaobo Li, and Jian Huang. SkyByte: Architecting an Efficient Memory-Semantic CXL-based SSD with OS and Hardware Co-design . In *2025 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 577–593, Los Alamitos, CA, USA, March 2025. IEEE Computer Society.

[66] Yuhong Zhong, Daniel S. Berger, Carl Waldspurger, Ryan Wee, Ishwar Agarwal, Rajat Agarwal, Frank Hady, Karthik Kumar, Mark D. Hill, Mosharaf Chowdhury, and Asaf Cidon. Managing Memory Tiers with CXL in Virtualized Environments. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, pages 37–56, Santa Clara, CA, July 2024. USENIX Association.

[67] Yuhong Zhong, Daniel S. Berger, Pantea Zardoshti, Enrique Saurez, Jacob Nelson, Dan R. K. Ports, Antonis Psistakis, Joshua Fried, and Asaf Cidon. Oasis: Pooling PCIe Devices Over CXL to Boost Utilization. In *Proceedings of the ACM SIGOPS 31st Symposium on Operating Systems Principles*, SOSP '25, pages 101–119, New York, NY, USA, 2025. Association for Computing Machinery.

ARTIFACT EVALUATED
USENIX
AVAILABLE

Duhu: Shared Disaggregated Memory for Distributed Data Processing Frameworks

Qiutong Men[†] Tao Wang[†] Jongryool Kim[⋆] Hane Yie[⋆]
Emmanuel Amaro[‡] Marcos K. Aguilera[◇] Aurojit Panda[†]
[†] NYU [⋆] SK Hynix [‡] Microsoft [◇] NVIDIA

Abstract

Today's distributed data processing frameworks (DDFs) have large memory and network transfer overheads because these frameworks require that each node (server or VM) copy objects into local memory before processing. Emerging shared disaggregated memory (SDM) clusters enable an alternate approach because they allow nodes to access data in a shared memory. However, using SDMs for DDFs is challenging: current SDM clusters provide weak coherence guarantees, and even for emerging SDMs coherence poses a scalability and complexity challenge. Thus, to adopt SDMs, a DDF would need to modify its logic and implement software coordination. In this paper, we describe Duhu, an SDM-based object store that is designed to allow DDFs to use SDMs without these changes, simplifying their adoption. We have integrated Duhu with Ray, and evaluated our system on an SDM cluster with a prototype CXL-attached memory pool. We show that Duhu can improve job completion time (JCT) by up to 3.39× on a shuffle workload.

1 Introduction

Distributed data processing frameworks (DDFs), including Ray [38] and Spark [60], play a crucial role in data analytics and machine learning. These frameworks make it easy to distribute program logic across multiple nodes (servers or VMs), making it feasible to process large datasets.

Performance, scheduling flexibility, and fault tolerance have long been concerns for these frameworks, and most frameworks address these concerns by using in-memory object stores to transfer immutable intermediate data between nodes. These data stores decouple tasks that produce data from those that consume them, thus allowing for more flexibility in when (and where) tasks are scheduled and reducing the amount of recomputation required when a node fails. By storing objects in memory rather than on disk, performance is greatly improved.

However, in-memory object stores incur significant overheads because, prior to access, they require that a node copy the object into local memory. This copying consumes CPU cycles (for serialization and deserialization) and network capacity (for the transfer). Furthermore, it increases the memory required to execute a job so that it far exceeds the job's memory footprint (i.e., the size of the job's inputs, outputs, and intermediate data).

Copying objects into local memory before access—a paradigm we refer to as pass-by-value—is needed because, until recently, processors could only access data in local memory. However, as the amount of data being processed by DDFs increases, this paradigm is increasingly unsustainable. Thus, we need to develop an alternative paradigm that eliminates these overheads without compromising the performance, fault tolerance, or scheduling flexibility of today's DDFs.

Emerging shared disaggregated memory (SDM) clusters provide an opportunity to move away from the pass-by-value paradigm used by DDFs today. These clusters consist of several memory blades that are connected to cluster nodes via a dedicated fabric running protocols such as CXL [10] or OpenCAPI [42]. Here, processors can use load-store instructions to directly access data in the memory blades, and thus SDM allows multiple nodes to share access to the same data. In the DDF context, this potentially allows multiple nodes to access the same copy of an object—rather than having to create local copies—enabling a paradigm that we refer to as *pass-by-reference*. Thus, the pass-by-reference paradigm alleviates memory pressure by reducing the number of object copies, without affecting fault tolerance or scheduling flexibility.

Prior work [38, 56] has suggested moving to a pass-by-reference-like approach where remote procedure calls (RPCs) include references (keys) to objects rather than the objects themselves. However, because these proposals did not target SDM clusters, servers still create a local copy (by fetching the object from the store) before access. Therefore, these proposals do not reduce network transfer overheads, nor do they reduce memory footprint. We detail these differences in §9, and our evaluation (§8) shows that our proposed approach improves on Ray.

In this paper, we describe Duhu, a pass-by-reference approach that targets SDM systems—specifically SDMs that do not implement global cache coherence protocols and hence do not guarantee cache coherence across servers. As we explain in §2.3, although emerging standards (e.g., CXL 3.0) include support for global cache coherence, this support carries a *coherence tax* [21] that limits scalability and performance, which Duhu avoids. Additionally, this choice allows Duhu to be deployed on existing memory blades [8, 22, 24] that do not provide global cache coherence.

The lack of coherence posed the most significant challenge when designing Duhu: it required that nodes coordinate and

synchronize caches when accessing shared data stored on the SDM. The need to coordinate adds performance overhead and makes it harder to integrate with existing DDFs, which were designed without awareness of SDMs. Our core contribution is our design (§3) that addresses both challenges.

Reducing Coordination Overheads. Our approach to reducing coordination leverages differences in mutability and access frequency for DDF objects and metadata. Specifically, our approach builds on the observation that object metadata is mutable (e.g., changes when other nodes get access to objects) but object data is immutable (it is only written when the object is first created). Furthermore, in most workloads, access to an object's data is more frequent and voluminous than to its metadata.

Our design requires inter-node coordination when accessing metadata. We reduce overheads for this coordination by partitioning the SDM into *segments* and associating each segment with an *owner*—a node responsible for sequencing and performing metadata operations on objects in the segment. This partitioning eliminates cross-node contention and reduces the cost of metadata operations.

Partitioning metadata into segments necessitates inter-node communication for metadata operations. To reduce communication overheads for these operations, we use Duhu-Channels (§5), a low-latency inter-node RPC mechanism that we designed to work over SDM. A Duhu-Channel uses SDM to send RPC requests and receive responses, and uses the network to signal the other endpoint. This combination enables low-latency communication.

Data access in Duhu does not necessitate communication or coordination. This is because objects are immutable, and thus the data changes only when an object is created. When creating an object, nodes use non-temporal writes to ensure that SDM contains a consistent version of the object. Additionally, nodes flush the corresponding cache lines before reading an object for the first time. This combination of non-temporal writes and cache flushes before first access ensures that readers always access the last consistent version of an object from SDM.

Simplifying DDF Integration. Our design simplifies integration with existing DDFs in two ways. First, Duhu provides a key-value store API that is similar to the API of object stores used by DDFs like Ray. Duhu implements its coordination logic within these API calls, thus ensuring that DDF logic need not change when using Duhu.

Second, Duhu implements reference counting to eliminate safety problems due to the garbage collector implemented by existing DDFs. Pass-by-value DDFs include a garbage collector that deletes the local copy of an object to regain space. This is safe, even if another node is concurrently accessing the object, because each node operates on its own copy. However, this is not true for Duhu or other pass-by-reference object stores that use a single copy of the object. We address this by tracking which nodes have references to the object, and freeing the object only when it is safe to do so, without requiring changes to DDF logic.

We implemented Duhu and integrated it with Ray [38]. We evaluated the Duhu version of Ray on an SDM cluster with four nodes and a prototype CXL-attached memory pool. The pass-by-reference semantics open up new possibilities; our evaluation (§8) describes FlexShuffle, a new shuffle approach enabled by Duhu. We show that, by replacing the physical transfer of shuffle data with only metadata reorganization, FlexShuffle improves job completion time for a multi-stage shuffle job by up to 3.39×. Further, in this case, Duhu can reduce an individual shuffle stage's runtime by 3.59–13.81×. Finally, we evaluated its impact on existing applications by modifying Modin [44] to use Duhu, and show that Duhu improves TPC-H query completion time for Modin by 1.08×.

2 Background

2.1 Distributed Data-Processing Frameworks

A Distributed Data Processing Framework (DDF) is a framework that allows users to process large amounts of data by distributing computation across multiple nodes (servers or VMs). These frameworks play an important role in data analytics and machine learning, and frameworks such as Spark [60, 61], TensorFlow [1], and Ray [38, 57] are widely used for this purpose.

In these frameworks, the nodes that implement the processing logic need to communicate with each other by sending intermediate values of the computation. These intermediate values are *immutable* data chunks that are written once and consumed one or more times, possibly by other nodes, depending on the workflow and scheduling of the computation. Most frameworks use object stores for this inter-node communication. The use of object stores provides several benefits: it increases scheduler flexibility by decoupling data producers and consumers, it allows intermediate data to be reused multiple times, and it enables fault tolerance because backup nodes can retrieve and operate on a failed worker's inputs and intermediate results.

Early DDFs, including Google's MapReduce [11] and Hadoop, used disk-based object stores (GFS [15] and HDFS [50], respectively). But the use of disk-based object stores was a performance bottleneck. Therefore, recent DDFs including Spark and Ray use in-memory immutable object stores: Spark stores intermediate data in RDDs [60], which are managed by a centralized block store, and Ray stores intermediate data in the Plasma store [43].

However, the pass-by-value paradigm used by in-memory object stores causes DDFs to experience a performance and scalability wall. Because processors can traditionally only operate on local DRAM, each node must copy an object from the object store into local memory before processing. This creates multiple physical copies of the same immutable object across the cluster. In turn, this copying wastes limited memory capacity, imposes serialization and deserialization

CPU overheads, and consumes valuable memory bandwidth and network resources that could otherwise support compute.

Our goal in this paper is to build an immutable object store that enables an alternative paradigm: pass-by-reference that does not copy objects into local memory. To do so, we need a mechanism that allows processes running on different nodes to directly access data. Thus, we target shared disaggregated memory (SDM) clusters, which we describe next.

2.2 Shared Disaggregated Memory

Duhu is designed to run on shared disaggregated memory (SDM) clusters, where memory blades [8, 22, 24] are decoupled from compute nodes and accessed via a high-performance fabric supporting load/store semantics, such as CXL [10] or OpenCAPI [42].

Modern and future SDM systems rely on one of two coherence implementations: *Hardware-coherent systems* (e.g., future CXL 3.0 SDMs) enforce global cache coherence via directory-based protocols. In contrast, *software-defined coherent systems* only provide coherence within a compute node; they do not coordinate accesses from different nodes. Duhu specifically targets the latter, and for our implementation we use CXL 2.0 memory expanders. Concretely, we use Type 3 devices that provide memory expansion but do not implement global coherence [10]. As we discuss below, this is to reduce performance overheads, improve scalability, and enable compatibility with today's memory blades.

Performance. These memory blades allow servers to access memory at a byte granularity. Prior work [28, 32] has reported expected latencies approximately 3–6× higher than local DRAM, and bandwidth approximately 4–16× lower than the total bandwidth of a local CPU socket, and our SDM prototype achieves 600–800ns latency and ≈10GB/s bandwidth (details in §8).

Failure domains. We assume memory blades are deployed in groups called memory units. Each unit has its own power supply and fabric connection. Thus, when a memory unit fails, the data in its blades is lost, but other units remain unaffected. We detail Duhu's assumed failure model in §3.

2.3 The Coherence Tax

While hardware coherence simplifies programming, it imposes significant overheads for cost, performance, and complexity. We refer to these overheads as the Coherence Tax, and discuss them below.

Straggler penalty. In directory-based protocols (e.g., MESI), writing to shared memory requires collecting invalidation acknowledgments from sharers. Thus, in the worst case, the write latency is bounded by the tail latency of the slowest sharer in the cluster.

Bandwidth tax. Hardware cache coherence adds protocol overhead (e.g., state updates) to data transfers. For bulk transfer of immutable objects, this incurs unnecessary overheads for every cache line written and fetched.

Complexity barrier. Scaling hardware coherence to petabytes of memory faces a storage wall. Directory structures (e.g., snoop filters) must track state for every cache line. Scaling to disaggregated capacities would require untenable amounts of on-chip SRAM and complex lookup and update logic in the fabric.

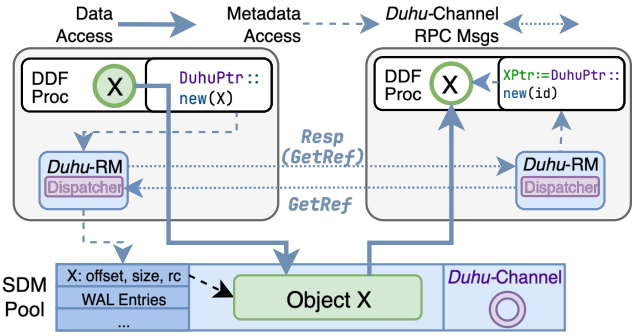


Figure 1: *Architecture of Duhu integrated with a DDF. Duhu-RM provides the same pointer view in address space for nodes to load and store (solid arrows) as native local memory objects. DDF processes call the Duhu API to perform metadata operations (dashed arrows). The API calls are handled by the local Duhu-RM, which uses the in-SDM Duhu-Channel mechanism to coordinate operations for remote objects (dotted arrows).*

3 Duhu Overview

Duhu is a pass-by-reference object store for in-memory DDFs running on SDM clusters that do not provide hardware cache coherence between nodes (§2.2). It improves DDF performance by reducing network overheads and decreases memory usage by eliminating redundant object copies. Duhu (Figure 1) stores two types of information in SDM: (1) *metadata* indicates where objects are stored and which nodes have access, and tracks information for failure recovery; (2) *data* consists of the actual content of an object.

Beyond enabling pass-by-reference semantics, our design aims to meet four requirements:

1. **Simplify integration with existing DDFs.** Our design ensures that core DDF logic does not need to change when integrating Duhu. We accomplish this by providing an API (§4) that is similar to what is provided by the immutable data stores [43] used by DDFs today. However, as we explain below, to meet our remaining goals, we limit the choice of keys (unlike standard key-value stores). Through the API, frameworks can put objects, get references, and drop references with a key, while directly reading bytes in SDM without copying the object to a local variable.
2. **Minimize coordination overheads.** Cross-node coordination is required to ensure the correctness of operations that read or modify object metadata. For example, coordination is required when allocating an object (so that the memory for any two objects does not overlap) and freeing an object (to ensure an object is freed only if no nodes plan to access it). Our design minimizes coordination overheads for these operations by partitioning the object store into segments, each of which is owned by a single node. When other nodes need to access the object, they issue an RPC to the owner. To reduce the overhead of contacting the owner, we provide an efficient

RPC mechanism, Duhu-Channel (§5), that uses the SDM for data transfer with the help of the network for signaling.

3. **Efficient but consistent access to shared object data.** DDFs allow multiple nodes to access the same object stored in an object store, and assume that all nodes get a consistent version of this object. Our design leverages object immutability to provide efficient, consistent object access from multiple nodes: when creating an object, a node uses non-temporal instructions to ensure that all updates are written to the SDM, and before reading an object for the first time, nodes flush cache lines to ensure that they read data from the SDM. Nodes need to flush cache lines before they read the object for the first time because an object might reuse SDM memory that was previously allocated to a different (now freed) object.

4. **Avoid resource leaks.** Current DDFs assume that each node accesses a locally stored copy of the object and can garbage collect that copy independently (in response to the node being done or local memory pressure). Duhu's pass-by-reference model violates the assumption made by these frameworks because multiple nodes can access the same copy stored in disaggregated memory. This in turn renders the garbage collection logic implemented by these frameworks incorrect.To address this, Duhu asks its user (the DDF) to indicate when it drops references to an object, so Duhu can track outstanding references (§4.2). This is usually straightforward, since most DDFs already have a local garbage collection mechanism where they drop references. With this information, Duhu ensures that a shared object is freed only after no nodes have references to it.

Failure model. Duhu is designed for use with DDFs that can reconstruct objects lost due to failures, using lineage or other recomputation mechanisms. Nearly all commonly used DDFs, including Ray and Spark, meet this requirement because alternative approaches such as object replication require significant additional memory.

We further assume that Duhu and the DDF that uses it are deployed in a cluster with multiple memory units and multiple nodes. The cluster is monitored by an orchestrator that notifies Duhu when a memory blade or node fails. Further, we assume that node failure does not affect the contents of the SDM; that is, data written by a node to the SDM remains available even after the node fails.

Duhu places all data and metadata for an object in the same memory unit to limit the blast radius. Under this model, our failure mechanisms (§6) ensure that a DDF that uses Duhu continues to function as long as enough nodes are available (dictated by the DDF) and the available memory units provide sufficient memory (dictated by the workload).

3.1 Duhu Design Principles

1. **Share data, but not metadata.** Sharing read-only data—such as the contents of immutable objects—is relatively easy even with incoherent memory. But metadata is constantly changing, so we keep it private: a single node is responsible for reading and writing it (possibly on behalf of other nodes). This node may change if there are failures.

2. **Streamline coordination using the SDM.** Cross-node coordination requires communication, and cross-node communication adds latency. Duhu reduces this latency by implementing Duhu-Channel, an RPC channel that communicates over SDM. The lack of coherence means that one node cannot be notified about SDM writes made by another node, i.e., `MWAIT` and similar instructions cannot be used with SDM. Therefore, when necessary, Duhu-Channel uses the network to signal nodes. This combination enables low-overhead cross-node communication, thus reducing coordination overheads.

3.2 Architecture Overview

Duhu provides an object store that DDFs use to store and access objects. The Duhu API (§4) provides a key-value-store-like API that DDFs can use to allocate objects, gain access to an existing object, and release references. Our design requires that objects be immutable: they are written once when allocated and are not modified subsequently. This assumption holds for most common DDFs, including Ray and Spark. Note that the Duhu API is called by DDFs, rather than applications. Application code does not need to change when using Duhu.

Duhu-RM. The Duhu API is implemented by a Duhu Reference Manager or Duhu-RM (§4). A Duhu-RM instance runs on each node and implements all of the coherence and coordination logic required to safely access the SDM. Duhu-RM is also responsible for updating and maintaining object metadata. Any node may wish to modify an object's metadata, and thus metadata maintenance requires coordination between Duhu-RMs running on different nodes. For performance and scalability, each Duhu-RM instance runs multiple threads.

The Duhu-RM also maintains the write-ahead log (WAL) that is used for failure recovery (§6): each thread writes an entry to the WAL before making changes to the SDM.

Duhu-Channel. Duhu-RMs coordinate with each other using a Duhu-Channel (§5), an RPC channel that allows nodes to communicate over SDM. Each Duhu-Channel provides client-server style communication. Multiple threads on the client Duhu-RM can issue RPC calls by writing to the same channel, and any thread on the server Duhu-RM can execute an RPC received on a channel. Furthermore, multiple RPC calls received on a single channel can be processed concurrently by different threads on the server Duhu-RM. Duhu-Channels are designed for small messages of less than 63 bytes in size. The restriction on small messages does not impact our design: all current inter-node requests in Duhu are smaller than 63 bytes; if larger payloads are required in the future they can be accommodated by passing SDM references.

When communicating over a Duhu-Channel, clients might need to notify the server about pending requests, or servers might need to notify clients about pending responses. The lack of global coherence prevents us from using SDM for notifications: a node receives no communication when another node modifies an SDM location. Therefore, we rely on messages sent over a traditional network for notifications.

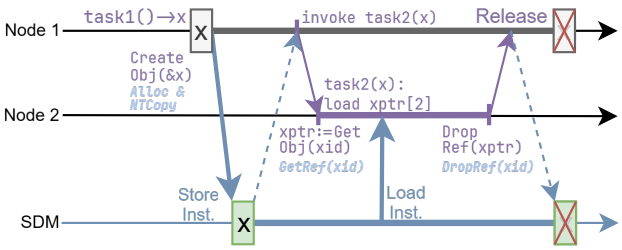


Figure 2: *A timeline view of object lifecycle events when integrating distributed computing frameworks with Duhu. DDF elements are purple, and Duhu elements are blue. Duhu-RM internal calls are specified under corresponding DuhuPtr API invocations.*

Dispatcher. Each Duhu-RM includes a dispatcher that is responsible for assigning requests received on a Duhu-Channel to a Duhu-RM thread. For correctness, the recovery logic requires that operations on a single object must be performed in the same order as recorded in the WAL. To ensure this, all requests to the same object are dispatched to the same thread: all RPCs issued by Duhu-RMs include an object ID that the dispatcher uses to select the appropriate thread. The dispatcher is also responsible for sending and processing channel notifications.

3.2.1 Illustrative Example

We illustrate how DDFs integrate with Duhu below by describing how a version of Ray that is integrated with Duhu creates an object, gains access to an object, and eventually releases its reference to an object (Figure 2).

Referencing objects. DDFs use two types of references, `ID` and `DuhuPtr`, to refer to objects stored in Duhu. An `ID` is meaningful for the entire cluster and serves as a reference that can be shared between different nodes. The `ID` encodes the segment in which an object is stored, which is used to route calls to the appropriate node.

However, an `ID` cannot be used to access object data. On the other hand, a `DuhuPtr` is node-specific and can be dereferenced to access object data. Duhu's reference counting keeps track of which nodes have valid `DuhuPtr`s to each object. DDFs can convert between the two types: the `GetObj` function (Table 1) returns a `DuhuPtr` given an `ID`, and the `GetID` function does the opposite.

Object creation. A DDF worker on node 1 creates and initializes a copy of object `x` in local memory. Once the object is initialized, the worker calls `CreateObj` to allocate and place object `x` in the object store. Once this call completes, `x` is stored in the SDM and is immutable. The call also returns a `DuhuPtr` through which node 1 can read `x`. Finally, node 1 can use the `GetID` API to get an ID that other nodes can use to get a reference to `x`.

Getting an object reference. A DDF worker on node 2 executing a newly launched task with a reference to `x` can gain access by calling the `GetObj` API. The `GetObj` and `CreateObj` calls update object metadata to track what nodes have outstanding references, thus ensuring that space allocated to an object cannot be reclaimed while a valid reference exists.

Figure 3: *Duhu-RM's memory layout in SDM pool.*

Releasing a reference. To enable space reclamation, a DDF worker must eventually release its reference to object `x` and does so by calling the `DropRef` method. To simplify integration, we invoke `DropRef` automatically from the `DuhuPtr` destructor when the reference goes out of scope. The object is freed and space is reclaimed once no node has a reference to `x`.

4 Duhu Reference Manager

Next, we detail the design of Duhu-RM. Our design partitions the SDM into segments to reduce coordination overheads for metadata operations. Furthermore, it leverages the immutability of objects to allow DDF workers to directly access object data while maintaining data consistency (§4.1).

The Duhu-RM implements the API (Table 1). DDFs use this API to create and gain access to objects, and to eventually release a reference as described in §3.2.1. The API itself closely resembles those of existing object stores, making it easier for DDFs to integrate with Duhu.

Internally, each Duhu-RM instance requires consistent access to object metadata, despite the metadata being stored in incoherent memory. Consequently, Duhu-RM instances must coordinate among themselves. Duhu simplifies and distributes this coordination by partitioning the data structures in SDM into segments (Figure 3).

Each segment has an *owner*, a node that has exclusive access to the metadata in the segment. Limiting metadata access to a single node ensures consistency. If a non-owner node wishes to access metadata in the segment, it issues an RPC to the owner's Duhu-RM. Table 2 shows the RPC interface exposed by Duhu-RMs. The RPCs require cross-node communication over a Duhu-Channel (§5).

Duhu guarantees that a segment has at most one owner. If a segment's owner fails, the Duhu recovery mechanism (§6) reassigns ownership to a different node and ensures that the segment's metadata is consistent.

Partitioning memory into segments whose ownership is distributed provides the benefits of centralized coordination without the scalability bottleneck of a central metadata manager, and without the locking complications of schemes where all nodes can access the metadata concurrently. In fact, we tried to design such concurrent schemes and found that the lack of cache coherence created overhead and complexity. Conceptually, Duhu shifts this complexity into the Duhu-Channel.

Segment layout. Each Duhu segment consists of four regions: (a) a Duhu-Channel region, containing data structures of channel used by the owner to send RPC requests to other nodes; (b) a write-ahead log (WAL) that is used during failure recovery to ensure metadata consistency; (c) a metadata region, containing a hash-map that maps *object identifiers* to object

API	Description
`CreateObj(*src, siz, owner) -> DuhuPtr`	Create an object in SDM with the data in `src` (in local memory). Returns a pointer to the SDM object.
`GetID(DuhuPtr) -> ID`	Get an ID from a `DuhuPtr`. The ID can be shared with other workers.
`GetObj(ID) -> DuhuPtr`	Get a `DuhuPtr` for an object ID, increasing the object's reference count in the process.
`DropRef(DuhuPtr)`	Invalidate the `DuhuPtr`, thus decreasing the object's reference count.

Table 1: *The API that DDFs use to interface with Duhu.*

Duhu-RM internal RPC	Description
`Alloc(id,*src,siz) -> *dst`	Requests space in the segment owned by a remote Duhu-RM.
`GetRef(id) -> *obj`	Gets a reference to object `id`, returns the address of the object. Records the client in the reference bitmap.
`DropRef(id)`	Releases a reference previously obtained by the node using a `GetRef` call. Erases the client in the reference bitmap.
`CreateId(segment)->id`	Returns an unused object ID that will be stored in the given segment.

Table 2: *The Duhu-RM internal RPC endpoints that Duhu-RMs use to talk to each other.*

metadata including the object's address, its size, and a bitmap recording nodes that hold references to the object; and (d) a data region, containing the data of objects. An object's metadata and data are co-located in the same segment. Co-location simplifies failure recovery because an object's data and metadata share fate, and avoids coordination for object allocation and reclamation, as updates to the object metadata and free list are performed by the same node.

Implementing the Duhu API. DDFs that use Duhu are unaware of segments, and the API they use (Table 1) exposes object IDs (which are recorded as a part of the object's metadata) and references to the object data (wrapped in a `DuhuPtr` type, which includes the ID).

API calls made by the DDF worker are handled by the local Duhu-RM running on the same node. Most of these API calls require manipulating object metadata, and the Duhu-RM uses RPCs (described below) for this if the object metadata is stored in a segment that it does not own.

API calls are cheaper when handled by the node that owns the segment containing the target object's metadata because no communication is required. Therefore, object placement can impact performance, and the `CreateObj` call allows DDFs to use the *owner* parameter to specify object placement.

RPCs to manipulate metadata. As we mentioned above, nodes must use RPCs (Table 2) to modify the metadata for any segment they do not own. To aid in failure recovery (§6), all RPC operations are idempotent. Furthermore, we require that, before executing a metadata operation, a node write (and flush) operations to the segment's WAL. The WAL entry includes all arguments, the Duhu-Channel ID, and additional metadata (including slot ID and version) that are required for the recovery logic to safely respond to the request.

Figure 4 illustrates API handling and RPC calls in Duhu:

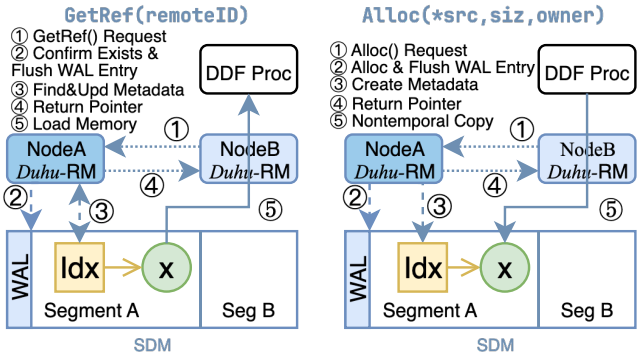


Figure 4: *The detailed protocol of GetRef and Alloc internal RPCs.*

the DDF worker running on node B issues an API call that is processed by the local Duhu-RM. The API call requires changing metadata in a segment owned by node A, and node B sends an RPC request to node A. Node A first writes the RPC request to its WAL, then performs the necessary operations and sends the response to node B. Communication between the two nodes is performed over a Duhu-Channel (§5).

4.1 Safe and Direct Data Access

A DDF can access data objects by dereferencing a `DuhuPtr`. Duhu implements this by having all nodes map SDM segments to the same virtual address, thus ensuring that no additional address translation is necessary. While this ensures direct data access, additional mechanisms are necessary to ensure that all nodes access consistent data, i.e., to ensure safety. Object immutability helps Duhu ensure consistency.

Immutable objects and consistency. As an immutable object store, Duhu requires that nodes first create and populate objects in local memory before adding them to Duhu. DDFs add objects to Duhu by calling the `CreateObj` API and providing a reference to the object in their local memory. Duhu then allocates space in the appropriate SDM segment and copies the object to this location. We copy the object asynchronously to allow pipelining and further reduce Duhu overheads.

The immutability of objects *almost* solves the problem of data consistency in the face of incoherent memory, but not quite: objects are eventually freed and new objects are created in the same locations in SDM. Thus, the actual data in an SDM location is not immutable; if not handled carefully, a node could potentially access stale data from a previous object that remains in the node's processor cache. To address this problem, nodes in Duhu carefully manage their processor caches as we explain next.

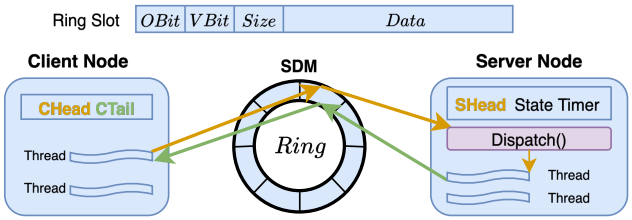


Figure 5: *Duhu-Channel Architecture Overview*

Low-overhead cache synchronization. With Duhu, the data in SDM changes infrequently and at well-defined points, namely when new objects are allocated in the same place in SDM as previously deleted objects. As observed by prior work [2, 18], this pattern allows us to use cache synchronization instructions sparingly to avoid their overhead, while ensuring consistency of access at all times.

More specifically, objects are cache-line aligned, and the object's creator uses non-temporal instructions (e.g., the `_mm512_stream_si512` intrinsic [23]) to ensure that writes are flushed to the SDM. This ensures that after the writer is done, any reads from the SDM return a consistent version of the object. Because no other worker can gain access before the writer finishes and no mutations are performed after `CreateObj`, reads from other workers can be safely served from the SDM without worrying about data inconsistency. To ensure that nodes access the SDM copy, the `GetObj` function invalidates the appropriate cache lines before returning a pointer to the object's data.

4.2 Reclaiming SDM Memory

Nodes allocate SDM memory when they invoke `CreateObj`. Duhu uses reference counting to safely reclaim an object's memory. Initially, an object's creator holds a reference to the object, and the reference count is 1. Subsequent calls to `GetObj` by other nodes increment the reference count. Nodes (regardless of whether they are the object's creator or not) can signal that they are done accessing the object by calling Duhu's `DropRef` function. It is unsafe for a node to access object data after calling `DropRef` without invoking `GetObj` again. Duhu reclaims the object's memory when the reference count reaches 0.

5 Duhu-Channel

A Duhu-Channel is an RPC channel that Duhu-RMs use for inter-node communication and coordination. A Duhu-Channel uses SDM to transfer RPC requests, arguments, and responses, and a network connection to signal Duhu-RM dispatchers when required. A Duhu-Channel connects a client Duhu-RM—which issues RPC requests—to a server Duhu-RM—which executes and responds to requests. Note that a channel is used by only two nodes (the client and the server).

5.1 Architecture and Functioning

A Duhu-Channel (Figure 5) has components in the SDM, and at the client and server Duhu-RMs:

1. **SDM**: Each channel allocates a ring-buffer (i.e., an array which is indexed using modulo arithmetic) consisting of n cache-line sized (64-byte sized) slots in the SDM, allowing up to $n-1$ RPC requests to be issued simultaneously. We ensure that slots are cache-line aligned, which in addition to them being cache-line sized means that nodes can read or write the slots atomically. Clients write to slots to issue an RPC, and servers write their responses back into the same slot.
2. **Client**: At the client Duhu-RM, a Duhu-Channel maintains a *client head* (*CHead*), the index of the slot that the client should use for the next RPC it issues. The client also maintains a *client tail* (*CTail*) which tracks the index of the slot beyond which all slots are unused. The *CTail* ensures that the client does not overwrite a slot being used by an in-flight RPC.
3. **Server**: At the server Duhu-RM, a Duhu-Channel maintains a *server head* (*SHead*) that tracks the next slot that a request will arrive on; and a variable tracking the state (*State*) of the server and a timer (*Timer*) that are used to switch between polling and using notifications for requests.

In addition to these components, each Duhu-RM maintains a network connection to every other Duhu-RM, and the dispatcher uses this to notify other nodes.

Slot structure. Figure 5 shows the structure of a channel slot: it contains metadata indicating message size, an 'ownership' bit (*OBit*) that indicates which endpoint (i.e., the client or the server) can access the slot, and a version bit (*VBit*) that the failure recovery logic (§6) uses to ensure correct re-execution. Initially, the *OBit* is set to *C*, indicating that the client owns the slot, and *VBit* is set to 0.

The slot also contains a data section: the client Duhu-RM issues an RPC by writing the request (and all required arguments) to this section, and similarly, the server responds to an RPC by writing the return value to this portion of the slot.

Observe that we use a single Duhu-Channel to carry both requests (from client to server) and responses: we chose this design in order to minimize Duhu's working set size, and thus reduce cache pressure on the nodes.

Algorithm 1: Client Request Submission

```
   Input: size, data                          // Payload size and data
   Output: result                             // Slot index or error
1  if CHead + 1 = CTail then return E_NO_SPACE
2  do
3      slot ← CHead.load()
4      next ← (slot + 1)  mod RingSize
5      until next ≠ CTail.load()
6  while CHead.CompareSwap(slot, next) fails
7  Msg ← PackedMsg{S; V; size; data}
   // S: set OBit to server.
   // V: current version.
8  WriteToSDM(Ring[slot] ← Msg)
```

5.1.1 Submitting an RPC Request

Algorithm 1 shows the procedure that a thread running on the client Duhu-RM invokes to submit a request to a Duhu-Channel. The client thread uses atomic operations to reserve a slot (lines 1–6); then it prepares a local copy of the message (line 8); and finally it writes the message to SDM (line 9).

When preparing the message (line 8), the client thread

Algorithm 2: Server Dispatch Loop

```
   Input: Timer
   Input: ChannelID
1  while State ≠ EXIT do
2      do
3          if State = EXIT then return
4          if State = IDLE then block until notified
5          InvalidateCacheline(Ring[SHead])
6          OBit ← Ring[SHead].owner
7      while OBit ≠ S
8      Timer.Reset()
9      slot ← SHead
10     size, data ←Load(Ring[slot])
11     SHead ← (SHead + 1)  mod RingSize
12     Dispatch(ChannelID, slot, size, data)
```

updates the slot's *OBit* to indicate that a request has been issued and the server Duhu-RM should take ownership.

The `WriteToSDM` function (line 9) atomically writes the message to SDM. It does so by using non-temporal write instructions that invalidate and flush the cache line containing $Ring[slot]$ to SDM. It follows the non-temporal write with a store-fence to ensure that the write cannot be reordered with respect to other store operations.

Finally, the client increments the slot's *VBit* (thus flipping it) upon receiving an RPC response.

5.1.2 Dispatching an RPC Request

Algorithm 2 shows the server side logic for receiving and dispatching RPC requests: The server polls the slot pointed to by *SHead* to check if its *OBit* is set to *S*, i.e., if the slot is owned by the server (lines 2–7). Once it owns the slot, the server copies the data from SDM into local memory (line 10), and then calls the Duhu-RM dispatcher which dispatches the request to a thread. Recall that the Duhu-RM dispatcher ensures that requests to the same object are dispatched to the same thread. Once a thread has finished processing the request, it writes a response to the same slot and changes the slot's *OBit* to *C* to indicate that a response is available. We omit this response logic in the interest of space. We also omit the logic of how the client receives this response, except to note that it is similar to the RPC request dispatch logic discussed here.

For safety and efficiency, the server must consider a few factors when polling for the slot to change ownership:

First, frequent polling affects the performance of other nodes (and threads) accessing the SDM. We address this using a backoff mechanism: the server uses a timer (*Timer*) to track time since an RPC request was last received. When the timer expires, the server marks the channel as `IDLE`, which causes the poll loop to block (line 4). Clients send a network message when sending a request over an `IDLE` channel (not shown), and on receiving this message the Duhu-RM dispatcher notifies (and thus unblocks) the poll loop. The server resets the timer each time it receives a message (line 8).

Second, while polling the server must ensure it is reading the value from SDM rather than from the local cache. To do so, it uses the `InvalidateCacheline` function (line 5) to invalidate the cacheline before the read. This function uses a memory fence to ensure that load reordering does not result in the poll loop reading a stale value.

6 Failure Recovery

Duhu depends on an orchestrator[1] to detect failed memory blades and nodes, and implements a failure recovery algorithm to deal with both. Below, we describe the core invariants used by both protocols, and then describe the protocols themselves.

6.1 Invariants

Duhu's mechanism for recovering from memory blade failures builds on the assumption that DDFs include mechanisms, e.g., lineage-based mechanisms, to recover lost data.

The mechanism for recovering from node failures depends on the following three invariants:

1. The Duhu-RM records all RPC requests to a write-ahead log (WAL) before executing them. WAL entries are used to ensure metadata consistency by re-executing operations that were interrupted by node failure.
2. The `Alloc()` logic ensures that updates to the segment's metadata hashmap have been flushed to SDM (by using non-temporal writes and an `sfence`) before returning a reference to the newly allocated memory. This ensures that all allocations are tracked in SDM, and prevents in-use memory from being reallocated after recovering from node failures.
3. Duhu-Channels allow at most k requests to be outstanding at a time; as we explain below, this helps ensure idempotent re-execution of incomplete operations.

6.2 Memory Blade Failure

Memory blade failure results in object data and metadata being lost. Duhu relies on the lineage-based fault-tolerance mechanisms implemented by DDFs to reconstruct this lost data on available blades. This recovery mechanism uses the object store APIs we provide, and thus implicitly recovers the lost metadata. We rely on these existing DDF-specific mechanisms because they can be better optimized.

However, Duhu's pass-by-reference paradigm does add one complication when memory blades fail: a node might hold a reference that points to an address on the failed blade. We address this by requiring that the node's operating system generate a SIGBUS signal when the DDF accesses memory on a failed blade, and Duhu catches this signal and invokes the DDF routine responsible for handling unavailable objects.

6.3 Node Failures

The failure of a node affects Duhu in two ways: (a) The metadata for any segments that the node owns might be inconsistent because of partially completed operations; and (b) References held by the failed node and Duhu-Channels created by it will never be freed, leading to resource leaks. Duhu uses different recovery algorithms for each effect. The orchestrator triggers both when a node failure is detected.

[1] Duhu uses the DDFs scheduler as its orchestrator, only requiring that it be extended to detect SDM failures (in addition to node failures).

6.3.1 Metadata Consistency

A Duhu metadata operation requires at least two accesses to SDM: one to read (or update) the metadata, and a second to write the return value to the appropriate channel slot. Many require more than one update. Consequently, node n crashing can render the metadata for any segment s owned by n inconsistent. Furthermore, no node can perform metadata operations on s until the segment is assigned a new owner. Our metadata consistency logic addresses both these problems.

During normal operation, each Duhu-RM thread records operations to a per-segment write-ahead log (WAL). The WAL entry records the request, the channel on which it was received, the channel slot, and the slot's *VBit*.

When the orchestrator detects that node n has failed, it assigns each segment owned by n to a different node (e.g., segment s to node n') and broadcasts this change.

Upon taking ownership of segment s, n' iterates through the metadata hashmap to reconstruct what portions of the segment memory have already been allocated, and to which objects. This process allows node n' to reconstruct (a) the set of objects in segment s; and (b) information required to reconstruct local allocator state. The latter is required to ensure that subsequent allocations (including those required during the recovery process) do not overlap existing objects, preserving data consistency.

Next, n' inspects the segment's WAL entries in descending timestamp order to find the last k entries for each Duhu-Channel, and re-executes them. Dropping all other entries is safe because a Duhu-Channel can only hold k outstanding requests, and thus older entries have already been processed. However, note that a previous owner (e.g., n) might already have processed and responded to some of the last k entries. Therefore, for correctness, Duhu needs to ensure that (a) responses generated during replay do not lead to bugs; and (b) replayed operations are idempotent.

Our use of in-band responses can be a problem when replaying and responding to operations: a previous owner might already have responded to a re-executed request before crashing, and the client (who issued the request) might already have reused the slot to issue a different request. We address this problem by restricting the number of requests re-executed from each ring and using the per-slot version bit. Before replaying a WAL entry, n' first reloads the Duhu-Channel slot for the request and compares the slot's version bit to that recorded in the WAL entry. If the channel slot and WAL entry have the same version, then the client has not reused the slot, and is still awaiting a response. The recovery logic re-executes the entry in this case, and discards it otherwise. Checking that version bits match suffices because we replay at most k WAL entries, and the version bit can change at most once in k entries.

A previous owner might have failed while processing a request, and thus ensuring idempotence during replay is necessary despite the Duhu-Channel slot version check described above. Ensuring idempotence during replay is simple for `GetRef()` and `DropRef()` calls, since they manipulate a bitmap using bitwise operations and are thus inherently idempotent. Similarly, for `CreateId()`, Duhu records the generated key in the WAL, and replay merely returns the recorded key, thus ensuring idempotence. Idempotence for `Alloc()` calls is slightly more involved: the replay logic first checks if the object ID is present in the segment's metadata hashmap. If so, the object has already been allocated, and the recovery logic uses the metadata entry to find the appropriate location. On the other hand, if the object ID is not present, it allocates space for the object.

Finally, the recovery procedure needs to ensure that any references held by n to objects on other nodes are released, thus avoiding resource leaks. To this end, on detecting that node n has failed, the orchestrator broadcasts a failure message to all other nodes. On receiving a failure message, node m finds and releases any references n holds to objects in segments owned by m by clearing the corresponding bit in each object's reference bitmap.

In sum, the use of a write-ahead log and idempotent operations ensures that replay can correctly restore metadata consistency.

6.3.2 Recovery Time

Finally, we analyze recovery time for node failures when using Duhu. As we show in §8, our current cluster has relatively high SDM access latency (600–800 ns) and 10GB/s bandwidth. In addition, the lack of coherence means that a newly assigned segment owner must invalidate the appropriate cache line before reading a slot or metadata. Lastly, for consistency, we need to use non-temporal writes and an `sfence` when writing to SDM. The net result is that recovery time is dominated by SDM access time rather than by computation, and we use SDM accesses as a way to estimate recovery time.

Metadata dictionary scan. The new owner of a segment needs to iterate over the segment's metadata hashmap to reconstruct local allocator state. The metadata hashmap can be quite large, and the recovery logic cannot make progress until this is completed. This stage tends to be a significant contributor to recovery time.

WAL scan. Next, the recovery logic needs to read the last k WAL entries for each channel. The layout of the write-ahead log means that newer entries are towards the beginning of the WAL and can be scanned sequentially. In the common case, where all Duhu-Channels are used at the same rate, only $O(p \times k)$ entries need to be read (where p is the number of channels) to find the last k WAL entries for each channel.

Replay of WAL entries. During replay, each operation requires at least one SDM access to check the version bit in the corresponding Duhu-Channel slot. Additional accesses are necessary during re-execution, e.g., to update metadata and write responses. These writes need to be synchronized with `sfence`s. Therefore, in most cases, this step is a significant contributor to recovery time.

Releasing references held by the failed node. This step has nearly no impact on recovery time because it runs concurrently with the rest of the recovery protocol and requires fewer SDM accesses than the other steps.

Thus, recovery time in Duhu is dominated by the metadata dictionary scan and operation replay. Changing the layout of the metadata dictionary might help improve both, though this might affect performance in the non-failure case.

7 Implementation and Integration

We implemented Duhu in C++ and integrated it with Ray, a popular open-source DDF. Integrating Duhu with Ray only required changes to Ray's object manager logic: we modified Ray's `PushLocalObject` method (which is used to transmit data from the local store to the remote one) to lazily allocate and move objects to the Duhu SDM. Similarly, we modified Ray's `GetRef` method so that it no longer allocated a local copy of the object, and instead (when appropriate) called Duhu's `GetRef` method to get a pointer to the object in SDM. Finally, we modified the `FreeObject` method to call `DropRef` when necessary. All of these changes were contained in a single module, demonstrating that our API simplifies integration.

During integration, we observed that many Ray objects are only accessed by the node that creates them. Copying objects that are only accessed locally to SDM adds significant overheads. Therefore, our implementation lazily copies objects: we delay calling `Alloc` on a sealed object until either (a) a remote node tries accessing the object, which causes Ray's object manager to try to send a copy over the network; or (b) the node that created the object encounters memory pressure.

Our implementation uses a standard free-list-based allocator to manage memory within each segment.

8 Evaluation

We evaluated Duhu by comparing the performance of a Duhu-integrated Ray (denoted as DR in the graphs) to a baseline Ray without Duhu (denoted as Ray in the graphs). Our evaluation answers the following questions:

1. Does Duhu enable new compute paradigms? Do they improve application performance? (§8.2.1)
2. Do our gains translate to future SDM implementations that have higher bandwidth and lower access latency? (§8.2.1)
3. Can Duhu improve the performance of Ray jobs? (§8.2.2)

Our results show the benefits of the pass-by-reference strategy enabled by Duhu, which reduces both memory usage and redundant data movement, albeit with additional overheads when accessing SDM.

In addition, we present results from microbenchmarks (§8.3) that quantify the benefit of Duhu-Channel, Duhu's impact on intermediate-data transfer time, and its benefits for applications whose tasks only randomly access a portion of the intermediate data they are provided.

8.1 Evaluation setup

Our evaluation was run on an SDM cluster that consists of four servers connected to an external disaggregated memory pool over CXL. Each server has four Intel Xeon Gold 6530 processors, 512 GB of memory, and runs Linux (kernel version 6.6.0). Each server is equipped with a ConnectX NIC, and all servers can communicate with each other over a 100Gbps network. We disabled hyperthreading and power saving features on the servers.

The disaggregated memory is an early FPGA-based research prototype with 128GB of RAM, with memory bandwidth of ≈10GB/s and access latency of 600–800 ns. The memory was partitioned into 16 segments, and in each segment we reserved 65536 WAL slots. The WAL slots and metadata region for each segment consumed 8MB of space, and in total ≈128MB of space was used by the WAL and metadata.

Our experiments use containers to partition server resources: each container is provided 12 cores and 30GB of memory. In each container, 8 cores are used for computation, and the remaining are used for communication (and other auxiliary functions). Local memory was not a bottleneck in any of our experiments.

To avoid cross-container interference, each container is placed on a different NUMA node and is allocated node local memory. Containers on the same server communicate over the Linux bridge, and we use `tc` to limit inter-container bandwidth within the server to 5Gbps (except in §8.3). This ensures that the aggregate bandwidth over the network is the same as SDM bandwidth (i.e., 10GB/s or 80Gbps), enabling fair comparison. Finally, for inter-server communication, containers are bound to the physical NIC using `macvlan`.

Even when using Duhu, Ray requires a local object store (whose size is specified using the `object-store-memory` flag). We specify the local object-store size used for this when describing individual experiments.

8.2 End-to-end Applications

We use two workloads based on existing DDFs to answer the three evaluation questions: shuffle-based workloads, for which we implement a new approach and compare to Exoshuffle [34]; and database workloads, for which we use Modin [44], a distributed dataframe library to run TPC-H [54].

8.2.1 Duhu's Impact on Shuffle

Duhu can improve the performance of distributed shuffle jobs by enabling new approaches. Below, we describe and compare Exoshuffle, an existing approach for distributed shuffle; and FlexShuffle, an approach enabled by Duhu.

Exoshuffle [34]. Exoshuffle is a state-of-the-art efficient distributed shuffle architecture built atop Ray. Like other pass-by-value shuffle architectures, Exoshuffle requires map tasks (data producers) to partition intermediate data using a user-defined partitioning scheme. These partitions are written to each mapper's local object store. Reduce tasks are assigned partitions that they copy from the mappers' object store and then combine to produce their output.

FlexShuffle. FlexShuffle is an architecture that we developed to target pass-by-reference implementations, e.g., Duhu. The

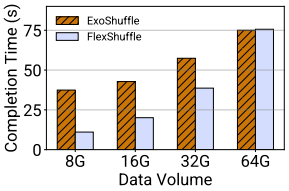


Figure 6: *Job-completion time for a four-stage shuffle job when using FlexShuffle and Exoshuffle.*

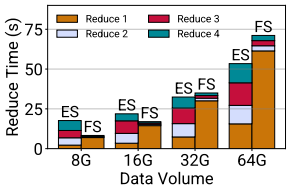


Figure 7: *Breakdown for each stage of the jobs shown in Figure 6.*

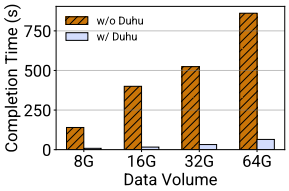


Figure 8: *JCT for a job with one shuffle stage when using FlexShuffle with and without Duhu.*

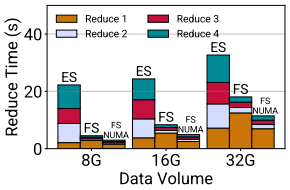


Figure 9: *JCT for a four-stage shuffle job on Exoshuffle, FlexShuffle, and FlexShuffle on NUMA.*

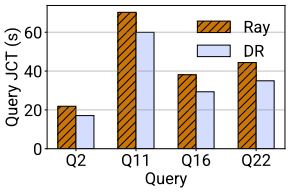


Figure 10: *Query execution time for the four most improved TPC-H queries.*

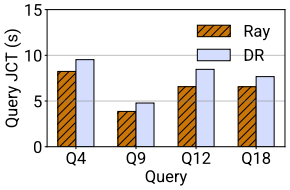


Figure 11: *Query execution time for the four slowest TPC-H queries when executed using Duhu-Ray.*

core difference is that in FlexShuffle, map tasks do not partition their output. Instead, map tasks write the unpartitioned output to Duhu, and produce a *slice* for each partition.

A partition's slice contains the set of offsets (and length) within the intermediate data that belong to the partition.

Reduce tasks use the slices and references to the mappers' output to access inputs and compute their output.

Thus, FlexShuffle replaces data movement with metadata reorganization. This change is particularly beneficial for jobs with multiple shuffle stages: after the first shuffle stage, only the set of slices changes, and thus no additional data transfer is required for these stages. Multiple shuffle stages are common for many applications including implementations of optimization algorithms like stochastic gradient descent [20, 41, 55].

FlexShuffle improves multi-stage shuffle performance. We evaluated FlexShuffle and Exoshuffle using 32 map tasks and 32 reduce tasks (2 per instance). We tested four scenarios where we varied the amount of shuffle data and the size of the local store: (1) 8GB data and 0.75GB object store; (2) 16GB data and 1.5GB object store; (3) 32GB data and 3GB object store; and (4) 64GB data and 6GB object store. In all scenarios, the intermediate data is split uniformly across 32 partitions, one for each reduce task.

Figure 6 shows job completion time (JCT) for a job with four shuffle stages in each scenario. Observe that FlexShuffle can improve JCT by up to 3.39×. However, we also observe a slight slowdown with large reduce size: FlexShuffle takes 1.01× longer than Exoshuffle. Figure 7 explains this performance difference: the first reduce stage takes longer under FlexShuffle than under Exoshuffle, because all intermediate data needs to be copied over to the SDM. In subsequent reduce stages, FlexShuffle outperforms Exoshuffle (by 3.59–13.81×) because no additional data movement is required. In the 64GB scenario, the initial data copy onto SDM dominates, leading to the slight slowdown.

FlexShuffle is impractical without SDM and Duhu. Next, we consider an implementation of FlexShuffle without SDM: in this case all workers keep a copy of the intermediate data in their local object store, and thus reduce inputs require only slices. The significant increase in local memory requirements is a problem in this case, since the data must be spilled into SSDs and reloaded from there. Figure 8 shows that without Duhu (and SDM), the JCT for a job with a single shuffle stage is 13.34–24.69× higher.

FlexShuffle's benefits increase with newer hardware. Finally, we considered how these improvements would translate to a better SDM implementation. As we noted above, our results are from using an FPGA-based SDM prototype, which likely has lower bandwidth and higher access latencies. Actual ASIC-based implementations are likely to perform better, and we use an approach inspired by Pond [28] to evaluate FlexShuffle's performance on these future implementations.

For this experiment, we emulated a faster SDM using NUMA nodes on a single server. The server had the same specifications as what we used for the experiments so far, and we used NUMA nodes 0 and 1 as servers, and reserved 100GB of memory attached to NUMA node 2 for use as SDM. Because we used a smaller number of servers, we only used up to 32GB of reduce data.

In Figure 9, we compare the performance of FlexShuffle on this NUMA setup (FS NUMA) to (a) FlexShuffle on a 2-node SDM cluster (FS); and (b) Exoshuffle on a 2-node cluster. As expected, the NUMA setup improves performance, and the biggest gains are in the first reduce stage where data movement costs dominate. In this setup, FlexShuffle with NUMA improves over FlexShuffle running on CXL by 1.10×. Furthermore, FlexShuffle on NUMA always outperforms Exoshuffle (it is 2.43–5.79× faster), leading us to conclude that FlexShuffle's benefits will increase as better SDM implementations become available.

8.2.2 TPC-H Benchmark on Modin

Next, we evaluate Duhu's impact on existing jobs, by measuring its impact on TPC-H [54] performance when the benchmark is executed using Modin [44].

For this evaluation, we used a scale factor of 10, which produced a 3.4GB compressed Parquet file as input. We set the per-node object-store size for each node to 5GB. As expected, this led to memory pressure in the Ray object store for some join queries. We found that running four containers on each server led to significant slowdowns for both unmodified Ray

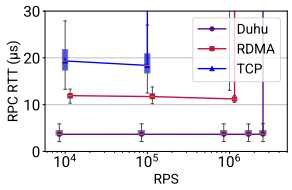


Figure 12: *Duhu RPC performance compared to RDMA and TCP.*

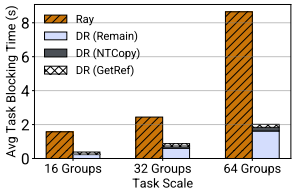


Figure 13: *Fan-out Microbenchmark: Time consumer waits for intermediate data by task scale.*

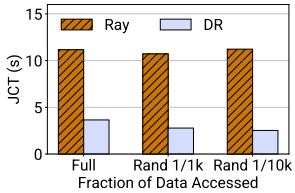


Figure 14: *Partial intermediate access: Job Completion Time by fraction of data accessed.*

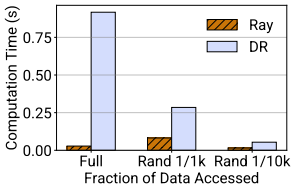


Figure 15: *Partial intermediate access: Computation Time by fraction of data accessed.*

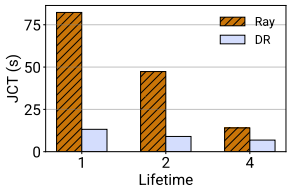


Figure 16: *JCT as object lifetimes change.*

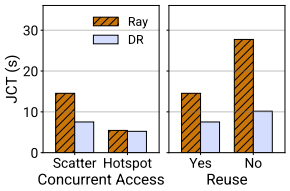


Figure 17: *JCT as the number of concurrent processes and object reuse change.*

and Duhu because of redundant data transfers to each server. Therefore, we ran only one container per physical server. Furthermore, we found that unmodified Ray workers frequently crashed due to memory exhaustion when running query 5, so we omitted it in our evaluation.

On average, across queries, Duhu-Ray had 1.08× better query time than vanilla Ray. However, the degree of improvement depended on the queries' intermediate data size and access patterns. Duhu seems to be particularly helpful for 'big' queries where each worker accesses a significant amount of data which is unlikely to fit in the worker's local object store. In this case, Duhu-Ray's ability to store objects in SDM is a significant benefit: Figure 10 shows query processing time for the four queries where we observed the most benefits, and we observe that at best Duhu-Ray can improve query completion time by 1.26× on average and by 1.30× at most.

However, for small queries that often run in less than 10 seconds and access a small amount of data, the use of Duhu can worsen performance. This is because each worker accesses only a limited amount of data in this case, and the objects being accessed can fit in the local object store for vanilla Ray. On the other hand, Duhu-Ray places objects in SDM by default, and the additional access latency leads to a slowdown. Figure 11 shows query execution times for the four slowest queries where Duhu-Ray is 1.2× slower.

Our results show that Duhu is most helpful when a query's intermediate data size exceeds a node's object store capacity. For smaller queries, SDM's increased access latency reduces Duhu's benefits, suggesting that a hybrid policy that keeps small or frequently reused objects on a server could improve Duhu-Ray's performance.

8.3 Microbenchmarks

Finally, we use microbenchmarks to evaluate Duhu's RPC and object transfer performance, the impact of accessing a portion of an object, and the impact of workload characteristics. All microbenchmarks use a producer-consumer pattern, and to ensure fair comparison, we increase network bandwidth for producers so that the aggregate bandwidth across producers is 80Gbps, matching write-bandwidth to SDM.

8.3.1 RPC Performance

First, we evaluate the benefits of using Duhu-Channel as compared to using network-based RPCs. We consider both RPCs implemented over TCP (using the kernel network stack), and implemented using the RDMA WriteImm operator. To fairly evaluate across these settings, our RDMA implementation has one outstanding request at a time.

In Figure 12, we report latency as we change the request rate. Our results show that Duhu-Channels have lower latency and higher throughput than either TCP or RDMA: they can achieve a latency of 3.8μs while supporting 3 million RPS. By comparison, RDMA achieves a latency of 11.74μs at 1 million RPS.

8.3.2 Object Transfer Performance

Next, we evaluate the benefits of transferring objects over SDM (as enabled by Duhu) rather than over the network. We do so using a fan-out benchmark, where we run a varying number of *fan-out task groups*.

A fan-out task group consists of a producer task that creates a 200MB array of floats, and four consumer tasks. Each consumer task is run on its own server, and computes the sum of all elements in the array. Observe that this setup means that three out of the four consumer tasks are remote, i.e., they are on a different server than the producer. In both Ray and Duhu-Ray, remote consumers need to wait for data transfer before they can begin computation, i.e., they are blocked. In Ray, tasks are blocked so that the remote worker can access a local copy of the data, while in Duhu-Ray tasks are blocked due to our lazy-copy optimization (§7).

In Figure 13 we report average blocking time when running 16, 32, and 64 fan-out task groups. Increasing the number of task groups increases the amount of data produced from 3.2GB to 12.8GB. We observe that Duhu-Ray's average blocking times are 2.80–4.29× smaller than Ray's. This is because Ray needs to transfer 3× more data, since the array must be copied into each server's local memory.

Furthermore, we analyzed where Duhu-Ray's blocking time went, by measuring time spent by the producer copying data into SDM, time spent by a consumer getting a reference, and the remainder. We found that allocating and putting data into SDM (done by the producer) comprises 18.83% of the

blocking time, and getting a remote reference and getting access at the consumer takes 10.96%. This is expected because the producer portion requires more data movement.

8.3.3 Accessing Partial Intermediate Data

Tasks that access a small portion of a large amount of data can benefit significantly from Duhu's pass-by-reference semantics. We evaluated this benefit using a benchmark where 128 tasks (8 on each of the 16 containers) read portions of a 6.4GB array of floats and computed the sum of the portions they read. We report on three scenarios while varying the portion of the array read by each task: (a) whole array; (b) $\frac{1}{1000}^{th}$ of the array; and (c) $\frac{1}{10000}^{th}$ of the array. For the latter two scenarios, each task chose elements at random.

Figure 14 shows JCT for all three scenarios: as expected, Duhu-Ray's benefits increase as a smaller portion of the array is accessed. JCT when reading $\frac{1}{10000}^{th}$ of the array is 4.45× smaller than when reading the entire array. By contrast, Ray's performance does not change significantly because data transfer costs dominate computation time.

However, the high access latency and low bandwidth of SDM (compared to local memory) might be a concern for computation. In Figure 15 we report compute time (averaged across the 128 tasks) for the experiment above. We observe that Ray always spends less time computing: for the $\frac{1}{1000}^{th}$ and $\frac{1}{10000}^{th}$ scenarios, Ray takes approximately 0.29× the compute time of Duhu-Ray. The compute difference is higher when summing over the entire array, because tasks then perform sequential access and can benefit from prefetching and caching. Nevertheless, our results show that the reduced data transfer time leads to lower overall JCT.

8.3.4 Impact of Workload Characteristics

The previous evaluations report on how data size affects Duhu-Ray's performance. We also measured the impact of three other workload characteristics: object lifetimes; number of tasks concurrently accessing an object; and access pattern.

We relied on the same benchmark for all three: we started 4 producer tasks (one on each server), and 128 consumer tasks (evenly distributed across the whole cluster). Producers and consumers ran for four iterations. Every few iterations (specified as a parameter), each producer created 8 float arrays, each 200MB in size. In each batch, a consumer was assigned an array and summed a randomly selected 1% of its elements.

Object lifetimes. As object lifetimes increase, objects are more likely to be re-accessed by some task on a server, and thus less likely to incur data movement for both Ray and Duhu-Ray. We measured this effect by varying the number of iterations after which a producer creates new arrays, e.g., if lifetime is 2, producers create 8 new arrays every other iteration (i.e., new objects will be created in iterations 1 and 3).

Figure 16 shows JCT as we vary lifetime from 1 to 4. As expected, both Ray and Duhu-Ray have better performance as the lifetime increases because data transfer costs are amortized. The benefit is more pronounced for Ray because later batches access objects from local memory, encountering significantly lower latency. By contrast, for Duhu-Ray, SDM access latency and bandwidth limit gains for later batches.

For the rest of the workload evaluation we set object lifetime to 4, i.e., arrays are allocated once for each experiment.

Concurrent access. The number of objects being accessed also has an effect on data transfer, and thus performance: data transfer can be amortized if all tasks access the same object. We evaluate this by comparing the performance in two scenarios: (a) scatter, where each consumer accesses a different array; and (b) hotspot, where all consumers access the same array. Figure 17 (left) shows the result. As expected, hotspot has better performance than scatter for both Ray and Duhu-Ray. Furthermore, we also observed that for the hotspot scenario, Ray and Duhu-Ray have similar performance. This is because each array is small, thus requiring minimal time for data transfer.

Task access pattern. Finally, task access patterns can affect the amount of data movement, and thus performance. We evaluated this by implementing two scenarios: (a) same, where each consumer accesses the same array for all four iterations; and (b) random, where each consumer accesses a different array in each iteration. Figure 17 (right) reports job-completion time for this experiment, and as one would expect, same has lower job-completion time for both Ray and Duhu-Ray.

9 Related Work

The idea of a hardware-based disaggregated memory predates CXL and was first proposed in [31]. However, that work does not consider using the shared memory as a means of communication (data sharing) among clients.

Recently, CXL has attracted the attention of the research community and several papers have appeared to address different aspects of this technology, including *memory pooling*, *memory tiering*, and *performance*. With memory pooling [16, 28], external CXL memory is dynamically assigned to hosts to improve memory utilization. A given chunk of memory is assigned to at most one host, so hosts do not use CXL memory to share data. With memory tiering [27, 35, 51, 58, 62, 64], the CXL memory is viewed as a slower tier of memory compared to local memory, and the system tries to place frequently accessed hot data in the faster tier, while moving colder data to the slower (CXL) tier. Other work provides a performance characterization of CXL memory and its implications for applications [17, 28, 32, 52, 53]. None of these works considers CXL memory as a medium to share data across hosts.

There is significant prior work on building distributed object stores and key-value stores on disaggregated memory accessible via RDMA (e.g., [12,13,25,30,33,37,49]). Similar to our setting, one or more memory servers store the key-value data. But unlike our setting, clients use RDMA operations to access the data in the memory server(s). RDMA is a different

mechanism to access remote data compared to memory operations in SDM: RDMA requires submitting IO operations, so there are no problems of hardware coherence of client caches; RDMA has lower performance on small transfers (say 64 bytes) due to the overhead of issuing IOs; and RDMA servers typically have some computation power, which many systems use to run client operations (e.g., RPCs). By contrast, we are focused on SDM, which is accessible by processors via loads and stores without cache coherence across nodes, and without the ability to run code at the memory. Therefore, our techniques differ from prior work.

Work on far memory systems (e.g., [4, 14, 19, 46, 59, 63]) aims to augment local memory with additional external memory. Early work assumes the external memory is accessible over the network, while more recent work considers CXL.

We are not the first to consider sharing data in CXL without full coherence [8, 22, 24]. Prior work differs from ours in two ways. First, they make a different hardware assumption: the CXL memory has a coherent and a non-coherent region, and applications can use both. Second, they target a different problem. Pasha [22] aims to build a database system using CXL shared memory, while [24] proposes general high-level ideas and directions of how to use this hybrid coherence model, without targeting specific applications.

Systems without cache coherence have been explored in both research [9, 26] and industry [6, 36]. The general consensus is that it is hard to program with such systems.

Using pass-by-reference instead of pass-by-value for efficiency is an idea that goes back to early work in programming languages. Recent distributed systems such as Ray [38, 56], Ciel [39], and Distributed PyTorch [45] use pass-by-reference in remote procedure calls (RPCs), whereby data gets stored in an object store, and applications use the key rather than the value as the parameters (or results) of the RPC; the RPC handler (or caller) later fetches the parameters (or results) from the object store. These systems differ from ours because their references are keys in an object store rather than addresses in SDM, and ultimately nodes still create copies of the object by fetching it from the object store, which is pass-by-value. As a result, these systems do not address the problem of memory and network transfer overheads in DDFs that we address (they solve a different problem), and they are not faced with the SDM cache-coherence issues that we tackle.

Lightning [65] is an in-memory object store that allows multiple threads to directly address objects in local memory without requiring inter-process communication or a mediating daemon. Thus, Lightning implements pass-by-reference semantics within a single node, but cannot be extended to the distributed setting we target because it requires cache coherence and does not consider cross-node synchronization.

There is a long line of work on distributed shared memory (e.g., [3, 5, 7, 29, 40, 47, 48]), which seek to implement a shared memory abstraction on top of a message-passing system, including both software and hardware-based implementations. Our goal is not to build shared memory but rather to use it.

10 Discussion

Support for copies. Duhu reduces unwanted copies of the data in DDFs by keeping objects in SDM. However, in some cases it is desirable to have copies to avoid bottlenecks. For example, a small immutable object may need to be accessed frequently by all nodes. We observe that Duhu can be used to provide copies—and can do so in a controlled manner. The DDF can put many objects with the same content in Duhu, and these objects can be given to different nodes. The DDF scheduler can decide how many objects (replicas) to create, including having a replica per node if desired.

Creating objects in SDM. Currently in Duhu, an object needs to be fully initialized in local memory before putting it in SDM. It could be useful to create the object directly in SDM to support large objects that do not fit in the remaining local memory, and to avoid the cost of copying the object from local memory to SDM. We can extend Duhu to support this functionality by adding a function that allocates space in a segment's data region and returns a pointer to SDM. Then, the DDF uses this pointer to populate the object. Finally, the DDF can use `Put`, augmented with a flag to indicate the object is already in SDM, to make it accessible by other nodes.

Loading objects from disk to Duhu. A DDF job often takes as input data from disk; currently, Duhu incurs some overhead in loading from disk, because objects must first be loaded into local memory before putting them into Duhu. Instead, it would be useful to extend Duhu with a function that loads data from disk by directly performing IO on buffers in SDM.

11 Conclusion

Duhu is an immutable object store designed for use with current and future shared disaggregated memory systems. It stores data in SDMs, which may not offer full cache coherence guarantees. Duhu is targeted at distributed data processing frameworks, which often copy data from the node that produces it to the nodes that wish to consume it; these copies incur significant memory and network transfer overheads. With Duhu, such frameworks avoid these copies by placing data in a common shared space. We integrate Duhu with Ray and evaluate the resulting system, Duhu-Ray, on a prototype CXL disaggregated memory system. We find that Duhu-Ray can outperform Ray by up to 3.39× on a shuffle workload.

Acknowledgments

We thank Stephanie Wang for initial discussions about the technical direction explored in this work, and the anonymous OSDI reviewers for comments that greatly improved its presentation. This work was funded in part by gifts from the Stellar Foundation and Google.

References

[1] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, M. Kudlur, J. Levenberg, R. Monga, S. Moore, D. G.

Murray, B. Steiner, P. Tucker, V. Vasudevan, P. Warden, M. Wicke, Y. Yu, and X. Zheng. Tensorflow: A system for large-scale machine learning. In *USENIX OSDI*, 2016.

[2] R. Abrahamse, Á. Hadnagy, and Z. Al-Ars. Memory-disaggregated in-memory object store framework for big data applications. In *IEEE International Parallel and Distributed Processing Symposium Workshops*, 2022.

[3] A. Agarwal, R. Bianchini, D. Chaiken, K. L. Johnson, D. Kranz, J. Kubiatowicz, B.-H. Lim, K. Mackenzie, and D. Yeung. The MIT Alewife machine: Architecture and performance. *ACM SIGARCH Computer Architecture News*, 23(2):2–13, 1995.

[4] E. Amaro, C. Branner-Augmon, Z. Luo, A. Ousterhout, M. K. Aguilera, A. Panda, S. Ratnasamy, and S. Shenker. Can far memory improve job throughput? In *ACM EuroSys*, pages 1–16, Apr. 2020.

[5] C. Amza, A. L. Cox, S. Dwarkadas, P. Keleher, H. Lu, R. Rajamony, W. Yu, and W. Zwaenepoel. TreadMarks: Shared memory computing on networks of workstations. *IEEE Computer*, 29(2):18–28, Feb. 1996.

[6] ARM922T Technical Reference Manual. https://developer.arm.com/documentation/ddi0184/b/caches--write-buffer--and-physical-address-tag--pa-tag--ram/cache-coherence.

[7] J. K. Bennett, J. B. Carter, and W. Zwaenepoel. Munin: Distributed shared memory based on type-specific memory coherence. In *Symposium on Principles and Practice of Parallel Programming*, pages 168–176, Mar. 1990.

[8] D. S. Berger. Realistic expectations for CXL memory pools. Keynote talk at the 2nd Workshop on Disruptive Memory Systems (DIMES'24) in conjunction with the ACM Symposium on Operating Systems Principles (SOSP'24), 2024. 2025-01-14.

[9] N. P. Carter, A. Agrawal, S. Borkar, R. Cledat, H. David, D. Dunning, J. Fryman, I. Ganev, R. A. Golliver, R. Knauerhase, et al. Runnemede: An architecture for ubiquitous high-performance computing. In *2013 IEEE 19th International Symposium on High Performance Computer Architecture (HPCA)*, pages 198–209. IEEE, 2013.

[10] Compute Express Link. https://computeexpresslink.org.

[11] J. Dean and S. Ghemawat. MapReduce: simplified data processing on large clusters. *Communications of the ACM*, 2008.

[12] A. Dragojević, D. Narayanan, M. Castro, and O. Hodson. FaRM: Fast remote memory. In *USENIX NSDI*, pages 401–414, Apr. 2014.

[13] A. Dragojević, D. Narayanan, E. Nightingale, M. Renzelmann, A. Shamis, A. Badam, and M. Castro. No compromises: distributed transactions with consistency, availability, and performance. In *ACM SOSP*, pages 54–70, Oct. 2015.

[14] E. Felten and J. Zahorjan. Issues in the implementation of a remote memory paging system. Technical Report CSE TR 91-03-09, University of Washington, Mar. 1991.

[15] S. Ghemawat, H. Gobioff, and S.-T. Leung. The Google file system. In *ACM SOSP*, 2003.

[16] D. Gouk, M. Kwon, H. Bae, S. Lee, and M. Jung. Memory pooling with CXL. *IEEE Micro*, 43(2):48–57, Mar. 2023.

[17] D. Gouk, S. Lee, M. Kwon, and M. Jung. Direct access, high-performance memory disaggregation with DirectCXL. In *USENIX ATC*, July 2022.

[18] P. Groet, J. Hoozemans, A. Grapentin, F. Eberhardt, Z. Al-Ars, and H. P. Hofstee. Leveraging apache arrow for zero-copy, zero-serialization cluster shared memory, 2024.

[19] J. Gu, Y. Lee, Y. Zhang, M. Chowdhury, and K. G. Shin. Efficient memory disaggregation with Infiniswap. In *USENIX NSDI*, pages 649–667, Mar. 2017.

[20] J. Haochen and S. Sra. Random shuffling beats SGD after finite epochs. In *International Conference on Machine Learning*, pages 2624–2633. PMLR, 2019.

[21] J. Hong, M. K. Aguilera, E. Amaro, V. Liu, A. Panda, and I. Stoica. The dawn of disaggregation and the coherence conundrum: A call for federated coherence. https://arxiv.org/abs/2504.16324, Apr. 2025.

[22] Y. Huang, N. Ni, V. Chidambaram, E. Witchel, and D. Tang. Pasha: An efficient, scalable database architecture for CXL pods. In *CIDR*, 2025.

[23] Intel. Intrinsics for integer load and store operations. https://www.intel.com/content/www/us/en/docs/cpp-compiler/developer-guide-reference/2021-10/intrinsics-for-integer-load-and-store-operations.html, 2021.

[24] S. Jain, N. Yeleswarapu, H. A. Maruf, and R. Gupta. Memory sharing with CXL: Hardware and software design approaches. https://arxiv.org/abs/2404.03245, 2024.

[25] A. Kalia, M. Kaminsky, and D. G. Andersen. Using RDMA efficiently for key-value services. In *ACM SIGCOMM*, Aug. 2014.

[26] W. Kim, S. Tavarageri, P. Sadayappan, and J. Torrellas. Architecting and programming a hardware-incoherent multiprocessor cache hierarchy. In *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 555–565. IEEE, 2016.

[27] T. Lee, S. K. Monga, C. Min, and Y. I. Eom. MEMTIS: Efficient memory tiering with dynamic page classification and page size determination. In *ACM SOSP*, pages 17–34, Oct. 2023.

[28] H. Li, D. S. Berger, S. Novakovic, L. Hsu, D. Ernst, P. Zardoshti, M. Shah, S. Rajadnya, S. Lee, I. Agarwal, M. D. Hill, M. Fontoura, and R. Bianchini. Pond: CXL-based memory pooling systems for cloud platforms. In *ACM ASPLOS*, Mar. 2023.

[29] K. Li and P. Hudak. Memory coherence in shared virtual memory systems. *ACM Transactions on Computer Systems (TOCS)*, 7(4):321–359, 1989.

[30] P. Li, Y. Hua, P. Zuo, Z. Chen, and J. Sheng. A high-performance RDMA-oriented learned key-value store for disaggregated memory systems. *ACM Transactions on Storage*, 19(4):1–30, Oct. 2023.

[31] K. Lim, J. Chang, T. Mudge, P. Ranganathan, S. K. Reinhardt, and T. F. Wenisch. Disaggregated memory for expansion and sharing in blade servers. In *ACM/IEEE ISCA*, pages 267–278, June 2009.

[32] J. Liu, H. Hadian, Y. Wang, D. S. Berger, M. Nguyen, X. Jian, S. H. Noh, and H. Li. Systematic CXL memory characterization and performance analysis at scale. In *ACM ASPLOS*, pages 1203–1217, Mar. 2025.

[33] Y. Liu, M. Xie, S. Shi, Y. Xu, H. Litz, and C. Qian. Outback: Fast and communication-efficient index for key-value store on disaggregated memory. *Proceedings of the VLDB Endowment*, 18(2):335–348, Oct. 2024.

[34] F. S. Luan, S. Wang, S. Yagati, S. Kim, K. Lien, I. Ong, T. Hong, S. Cho, E. Liang, and I. Stoica. Exoshuffle: An extensible shuffle architecture. In *ACM SIGCOMM*, 2023.

[35] H. A. Maruf, H. Wang, A. Dhanotia, J. Weiner, N. Agarwal, P. Bhattacharya, C. Petersen, M. Chowdhury, S. Kanaujia, and P. Chauhan. TPP: Transparent page placement for CXL-enabled tiered-memory. In *ACM ASPLOS*, page 742–755, 2023.

[36] A. Maximo, G. Cox, C. Bentes, and R. Farias. Unleashing the power of the Playstation 3 to boost graphics programming. In *2009 Tutorials of the XXII Brazilian Symposium on Computer Graphics and Image Processing*, pages 45–58. IEEE, 2009.

[37] C. Mitchell, Y. Geng, and J. Li. Using one-sided RDMA reads to build a fast, CPU-efficient key-value store. In *USENIX ATC*, June 2013.

[38] P. Moritz, R. Nishihara, S. Wang, A. Tumanov, R. Liaw, E. Liang, M. Elibol, Z. Yang, W. Paul, M. I. Jordan, and I. Stoica. Ray: A distributed framework for emerging ai applications. In *USENIX OSDI*, 2018.

[39] D. G. Murray, M. Schwarzkopf, C. Smowton, S. Smith, A. Madhavapeddy, and S. Hand. CIEL: A universal execution engine for distributed data-flow computing. In *USENIX NSDI*, 2011.

[40] J. Nelson, B. Holt, B. Myers, P. Briggs, L. Ceze, S. Kahan, and M. Oskin. Latency-tolerant software distributed shared memory. In *USENIX ATC*, pages 291–305, July 2015.

[41] T. T. Nguyen, F. Trahay, J. Domke, A. Drozd, E. Vatai, J. Liao, M. Wahib, and B. Gerofi. Why globally reshuffle? revisiting data shuffling in large scale deep learning. In *2022 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 1085–1096. IEEE, 2022.

[42] OpenCAPI (Open Coherent Accelerator Processor Interface). https://docs.kernel.org/userspace-api/accelerators/ocxl.html.

[43] The Plasma in-memory object store. https://arrow.apache.org/blog/2017/08/08/plasma-in-memory-object-store.

[44] M. Project. Modin: Speed up your pandas workflows by changing one line of code. https://github.com/modin-project/modin, 2025.

[45] Pytorch - remote reference protocol. https://pytorch.org/docs/stable/notes/rref.html.

[46] Z. Ruan, M. Schwarzkopf, M. K. Aguilera, and A. Belay. AIFM: High-performance, application-integrated far memory. In *USENIX OSDI*, pages 315–332, Nov. 2020.

[47] D. J. Scales, K. Gharachorloo, and C. A. Thekkath. Shasta: A low overhead, software-only approach for supporting fine-grain shared memory. In *ACM ASPLOS*, pages 174–185, Oct. 1996.

[48] I. Schoinas, B. Falsafi, A. R. Lebeck, S. K. Reinhardt, J. R. Larus, and D. A. Wood. Fine-grain access control for distributed shared memory. In *ACM ASPLOS*, pages 297–306, Oct. 1994.

[49] J. Shen, P. Zuo, X. Luo, T. Yang, Y. Su, Y. Zhou, and M. R. Lyu. FUSEE: A fully memory-disaggregated key-value store. In *Proceedings of the 21st USENIX Conference on File and Storage Technologies*, FAST '23, pages 81–98, Berkeley, CA, USA, 2023. USENIX Association.

[50] K. Shvachko, H. Kuang, S. Radia, and R. Chansler. The Hadoop distributed file system. In *IEEE MSST*, 2010.

[51] Y. Sun, J. Kim, Z. Yu, J. Zhang, S. Chai, M. J. Kim, H. Nam, J. Park, E. Na, Y. Yuan, R. Wang, J. H. Ahn, T. Xu, and N. S. Kim. M5: Mastering page migration and memory management for CXL-based tiered memory systems. In *ACM ASPLOS*, pages 604–621, Mar. 2025.

[52] Y. Sun, Y. Yuan, Z. Yu, R. Kuper, C. Song, J. Huang, H. Ji, S. Agarwal, J. Lou, I. Jeong, R. Wang, J. H. Ahn, T. Xu, and N. S. Kim. Demystifying CXL memory with genuine CXL-ready systems and devices. In *IEEE MICRO*, pages 105–121, Dec. 2023.

[53] Y. Tang, P. Zhou, W. Zhang, H. Hu, Q. Yang, H. Xiang, T. Liu, J. Shan, R. Huang, C. Zhao, C. Chen, H. Zhang, F. Liu, S. Zhang, X. Ding, and J. Chen. Exploring performance and cost optimization with ASIC-based CXL memory. In *ACM EuroSys*, pages 818–833, Apr. 2024.

[54] T. P. P. C. (TPC). TPC benchmark H (TPC-H). http://www.tpc.org/tpch.

[55] S. Venkatraman, A. Panda, G. Ananthanarayanan, M. Franklin, and I. Stoica. The power of choice in data-aware cluster scheduling. In *USENIX OSDI*, pages 301–316, 2014.

[56] S. Wang, B. Hindman, and I. Stoica. In reference to RPC: it's time to add distributed memory. In *ACM HotOS*, 2021.

[57] S. Wang, E. Liang, E. Oakes, B. Hindman, F. S. Luan, A. Cheng, and I. Stoica. Ownership: A distributed futures system for fine-grained tasks. In *USENIX NSDI*, 2021.

[58] L. Xiang, Z. Lin, W. Deng, H. Lu, J. Rao, Y. Yuan, and R. Wang. NOMAD: non-exclusive memory tiering via transactional page migration. In *USENIX OSDI*, pages 19–35, July 2024.

[59] A. Yelam, S. Grant, S. Deshpande, N. Amit, R. N. Mysore, A. Ousterhout, M. K. Aguilera, and A. C. Snoeren. Eden: Developer-friendly application-integrated far memory. In *USENIX NSDI*, Apr. 2025.

[60] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica. Resilient distributed datasets: a fault-tolerant abstraction for in-memory cluster computing. In *USENIX NSDI*, 2012.

[61] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. Spark: cluster computing with working sets. In *USENIX HotCloud*, 2010.

[62] Y. Zhong, D. S. Berger, C. Waldspurger, R. Wee, I. Agarwal, R. Agarwal, F. Hady, K. Kumar, M. D. Hill, M. Chowdhury, and A. Cidon. Managing memory tiers with CXL in virtualized environments. In *USENIX OSDI*, pages 37–56, July 2024.

[63] Y. Zhou, H. M. Wassel, S. Liu, J. Gao, J. Mickens, M. Yu, C. Kennelly, P. Turner, D. E. Culler, H. M. Levy, et al. Carbink: Fault-tolerant far memory. In *USENIX OSDI*, pages 55–71, July 2022.

[64] Z. Zhou, Y. Chen, T. Zhang, Y. Wang, R. Shu, S. Xu, P. Cheng, L. Qu, Y. Xiong, J. Zhang, and G. Sun. Neomem: Hardware/software co-design for CXL-native memory tiering. In *IEEE MICRO*, pages 1518–1531, Nov. 2024.

[65] D. Zhuo, K. Zhang, Z. Li, S. Zhuang, S. Wang, A. Chen, and I. Stoica. Rearchitecting in-memory object stores for low latency. *Proceedings of the VLDB Endowment*, 15(3):555–568, Nov. 2021.

Content	Branch	Location
Artifact README	`duhu`	`README.md`
Implementation		
Duhu-Ray	`duhu`	Main changes under `src/ray/object_manager/`
Ray	`releases/2.40.0`	No change
Experiments	`duhu`	`duhu-eval/`
FlexShuffle	`duhu`	`duhu-eval/flexshuffle`
Modin	`duhu`	`duhu-eval/modin`
Microbenchmarks	`duhu`	`duhu-eval/microbenchmark`

Table 3: *The contents of our artifact.*

A Artifact Appendix

Abstract

This paper's artifact includes the code for Duhu, and a version of Ray that is integrated with it. It builds on Ray branch `releases/2.40.0`, and most of the modifications are in the `object_manager`. These changes include logic to allocate, get (and drop) references, and delete objects in a Duhu-managed SDM.

The repository also includes the workloads we used in our evaluation.

Scope

When run in the same setup as described in the paper, the artifact reproduces the workload-performance results reported in §8.

Contents

The artifact is within a single repository, with (1) a branch forked from the original Ray repository, (2) a modified Ray with Duhu integrated, and (3) a set of workloads we run to evaluate the performance of Ray and Duhu-Ray, including FlexShuffle, TPC-H on Modin, and microbenchmarks. Table 3 lists the contents of the artifact in detail.

Hosting

The artifact is hosted in a GitHub repository (https://github.com/nyu-systems/duhu-ray), with all contents listed above.

ARTIFACT EVALUATED
USENIX
AVAILABLE

Blowfish: Elastic Virtual Machine Memory for Disaggregated Memory

Yulong Zhang [1,2] Yilong Luo [3] Diyu Zhou [3] Quan Chen [4] Quanxi Li [1,2]
Mosong Zhou [5] Lei Zhu [5] Senbo Fu [5] Qian Peng [5] Huimin Cui [1,2]
Xiaobing Feng [1,2] Tao Xie [3] Chenxi Wang [1,2]*

[1]*SKLP, Institute of Computing Technology, CAS* [2]*University of Chinese Academy of Sciences*
[3]*SCS, Peking University, China* [4]*Shanghai Jiao Tong University* [5]*Huawei Cloud*

Abstract

Cold memory rebalancing exhibits unique challenges to the existing memory overcommitment mechanisms, such as page tracking under Transparent Huge Pages (THP) and frequent remapping of page tables, leading to a series of throughput and latency issues. In this paper, we propose Blowfish, a memory overcommitment framework built on disaggregated memory, performing cold (and free) memory reclamation and restoration at µs-scale. Based on paravirtualization, Blowfish leverages a lightweight guest-level THP-aware hotness tracker to monitor page access and let the hypervisor directly reclaim and reallocate host physical memory across VMs with a dedicated cross-layer path for cold memory, bypassing the modifications of guest page table and modifications of I/O page table while benefiting from the rich program semantics of guest VM when recognizing the cold memory. As a result, Blowfish significantly speeds up the page reclamation and restoration, 2.48× and 2.14× faster than the state-of-the-art solution, HyperAlloc, respectively, and improves memory reclamation ratios by 1.6×–6.1× within 5% performance degradation.

1 Introduction

Memory overcommitment promises to improve memory utilization and has been widely adopted in multi-tenant data centers. However, the existing memory overcommitment research [9, 24, 27, 30, 39, 53, 56, 59] mainly focuses on free memory rebalancing, *i.e.*, reclaiming free memory and reallocating it across virtual machines (VMs), overlooking cold memory, which accounts for a large portion of VM memory usage [19, 40, 50, 58]. In the existing research, the conventional approach to reclaiming cold memory relies on either guest-level swapping, in which cold pages are first swapped out within the guest and then reclaimed as free pages, or host-level swapping, in which the guest's underlying host pages are swapped without the guest's awareness. Both techniques require moving infrequently accessed data to disk. Accessing reclaimed cold memory then requires restoring it, typically by triggering a page fault, allocating free memory, and swapping the data back. However, writing to and reading from disk are slow and costly (up to milliseconds for a 4 KB page), leading to Service-Level Objective (SLO) violations. As a result, prior work often overlooks opportunities to reclaim cold memory for memory overcommitment, even though doing so could provide substantial benefits.

Memory disaggregation enables µs-scale memory overcommitment. Thanks to the unprecedented development of emerging fabrics, *e.g.*, InfiniBand [1], RoCE NICs [43], and CXL [22, 37], the time of sending a page to another server is reduced to a few microseconds (µs) and more than 200× faster than swapping to disk [23, 49, 54, 61], bringing great opportunities to further improve the memory utilization through cold memory swapping. Given that more than 50% of data center memory remains idle [7, 21, 58] due to the bin-packing inefficiencies in VM placement, the emerging fabrics make it possible to use this idle memory of other servers as the secondary memory (referred to as *far memory*) and swap out the cold data of a VM to far memory to reclaim the memory space for overcommitment.

Software overhead starts dominating the inefficiency of VM memory overcommitment. Even with high-speed interconnects, existing memory overcommitment frameworks still suffer from substantial inefficiency, as presented in §3. There are two main reasons for this inefficiency: (1) With guest-level swapping, reclaiming and restoring cold memory involves three page tables: the guest page table (GPT), which maps guest virtual addresses to guest physical addresses, the extended page table (EPT), which maps guest physical addresses to host physical addresses, and the I/O page table (IOPT), which maps I/O virtual addresses to host physical addresses, causing software overhead to be 3.4×–6.8× higher than network data transfer. (2) With host-level swapping, reclaiming cold memory requires tracking page hotness on the host side, which involves frequently resetting the host's PTE

* Corresponding author: Chenxi Wang.

access bits and results in 5× more TLB flushes than tracking on the guest side.

Transparent Huge Pages (THPs) further complicate memory overcommitment. Making things more challenging, virtualized environments commonly rely on THPs to reduce page-table translation overhead. However, cold memory is often non-contiguously distributed, creating complex trade-offs in page granularity. Keeping huge pages preserves translation efficiency but inflates apparent hotness, since a single hot subpage can mark the entire huge page as hot. Splitting huge pages allows more accurate hotness tracking but sacrifices the benefits of huge pages and incurs overheads of fine-grained page-table operations. Moreover, memory overcommitment mechanisms often need to manage pages of mixed granularity when THP is enabled, because fine-grained memory operations (such as allocation, deallocation, permission changes, and swapping) frequently break huge pages. Existing mechanisms treat all pages uniformly, leading to unfair hotness comparisons between 2 MB huge pages and 4 KB base pages, since a huge page aggregates the hotness of all its constituent base pages.

Major insights. **First, cold memory overcommitment suffers from a misplaced division of responsibilities across layers.** We propose that an ideal decomposition should place hotness tracking in the guest, where access semantics are naturally visible and can be monitored at low cost, while leaving data transfer and physical memory management to the host, which actually controls memory resources. This organization minimizes software overhead of page tracking and repeated page-table modifications. **Second, THP fundamentally changes the design space by introducing mixed-granularity memory.** We identify that an effective mechanism should be explicitly aware of mixed granularity: it should detect cold subpages within hot huge pages, fairly compare pages across sizes, and exploit THP benefits when possible. This requirement calls for rethinking hotness tracking and huge page management to efficiently operate across granularities.

Blowfish. Based on the preceding insights, we propose Blowfish, an elastic VM memory adjustment framework built for the memory-disaggregated server. Blowfish's design targets two main challenges:

First, *how to reduce the software overhead to avoid SLO violations during cold memory overcommitment?* Blowfish adopts a *paravirtualized, cross-layer* design that splits responsibilities between the guest and the host. The guest VM runs a lightweight hotness tracker that integrates with existing kernel mechanisms (*e.g.*, MGLRU stack) to accurately identify cold pages without introducing additional page-table or TLB overheads. The host hypervisor runs a backend that transfers cold data and reclaims pages by modifying only the EPT mappings, avoiding the costly modification overhead of guest page table and IOPT normally incurred by guest-level I/O. In addition, the guest and host exchange only a small amount of metadata through a shared memory interface, an effective and widely used technique [10, 56, 59].

Second, *how can we efficiently support mixed-granularity memory under THPs during cold memory overcommitment?* Blowfish leverages a mixed-granularity hotness tracker in the guest to identify cold subpages within otherwise hot huge pages in a lightweight manner. This identification is done by dynamically splitting and merging guest page table mappings rather than actual physical pages, minimizing overhead. To ensure fair hotness comparison across page sizes, Blowfish adjusts the promotion policy of huge pages, giving them lower weight than base pages. Together, these mechanisms allow Blowfish to efficiently rebalance memory even in the presence of THP-induced fragmentation and uneven hotness distribution.

Our evaluation shows that Blowfish reduces software overhead by more than 50% compared to state-of-the-art work, such as HyperAlloc [59]. Moreover, it further improves memory reclamation ratios by 1.6×–6.1× within a 5% performance degradation. Blowfish is available at https://github.com/ICTPLSys/blowfish.

In summary, this paper makes the following contributions:

- We analyze cold memory overcommitment in virtualized environments, revealing inefficiencies in existing mechanisms and outlining the principles of an ideal system, *i.e.*, guest-level hotness tracking guiding host-level reclamation, and THP awareness.
- We design a lightweight, subpage-aware hotness tracker that enables fair comparison across mixed-granularity pages, benefiting any scenario that requires hotness tracking under THPs.
- We propose Blowfish, an efficient memory overcommitment framework optimized for THP and cold memory.
- We implement Blowfish in Linux and QEMU, and demonstrate its effectiveness with seven real-world workloads.

2 Background and Related Work

2.1 Disaggregated Memory System

Benefiting from emerging fabrics, the disaggregated memory systems [8, 16, 37, 49, 51, 54, 61, 63] can provide extremely low data movement latency and high throughput, such as moving a 4 KB page to another server in less than 5µs through the 100Gbps InfiniBand [49, 61], enabling the opportunity of rebalancing the cold memory across VMs. Among different approaches to realizing disaggregated memory, the kernel-paging-based approach [8, 17, 23, 49, 54, 61] attracts attention for its transparency to applications and easy-to-procure hardware platforms, *e.g.*, RoCE NIC [43] and InfiniBand [42].

The disaggregated memory system uses the far memory (offering bandwidths up to 800Gbps [44]), rather than slow disks (around 200MB/s), as the secondary memory via the swap system. Under memory pressure, the kernel swaps out cold data to the far memory to release the local memory. When the evicted data is accessed, the memory management unit (MMU) raises a page fault exception, which traps the control flow into the kernel to swap in the original data from far memory and reconstruct the corresponding page table entry. In this work, we use the state-of-the-art paging-based data path, *i.e.*, Hermit [49], to manage the data transmission at the microsecond scale.

2.2 VM Memory Overcommitment

This section discusses how to integrate the paging-based disaggregated memory with the existing memory overcommitment framework to support cold memory overcommitment.

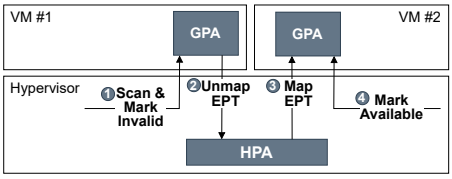


Figure 1: Reallocating free memory from VM1 to VM2.

Free memory rebalancing. A series of memory overcommitment techniques supporting free memory rebalancing across VMs have been proposed, such as Balloon [53], free page reporting [57], virtio-mem [24], V-Probe [56], HyperAlloc [59] *etc.*, some of which are even adopted by the industry virtualization platforms [5,12,31]. As shown in Figure 1, **reclaiming a free guest page** (❶–❷) needs only to unmap the extended page table (EPT), while **reallocating a host page** (❸–❹) to another VM also involves only one page table, *i.e.*, EPT. Taking HyperAlloc as an example, ❶ the hypervisor first checks the physical memory allocator of the VM to select a free guest page and marks it as invalid to prevent the guest VM allocator using it. ❷ Then the hypervisor unmaps the corresponding guest-physical-to-host-physical mapping. At this point, the reclaimed host page can be reallocated to another VM by ❸ mapping EPT and ❹ marking the originally invalidated guest page as available.

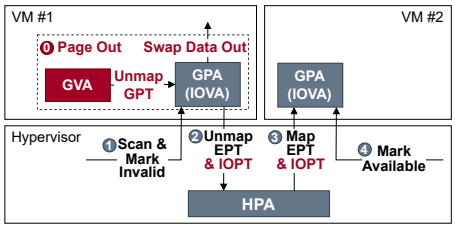


Figure 2: Reallocating cold memory from VM1 to VM2.

Cold memory rebalancing. The free memory overcommitment framework can be extended to rebalance cold memory by cooperating with the kernel paging (swapping)—swapping infrequently used data to the second-tier memory through emerging IO fabrics to release the cold memory, and then, the released memory pages can be reallocated to other VMs as conventional free pages. The paging can be done in either the guest VM or the hypervisor. These two options are referred to as *guest-level swapping* and *host-level swapping*, respectively.

Figure 2 demonstrates the procedure of reclaiming a cold page through the guest-level swapping (❶–❷) and reallocating it to another VM. Reclaiming a cold page first ❶ needs to select a cold page and swap out the infrequently used data on it to release the cold page by unmapping the GPT. And then, similar to reclaiming a free guest page (❶–❷), the hypervisor needs to unmap the corresponding EPT to reclaim the corresponding free host page. Note that the hypervisor also needs to unmap the IOPT (❷) due to the utilization of the emerging fabrics for data transmission. The IOPT translates the I/O virtual addresses (typically guest physical addresses in the VM) to the actual host physical addresses, ensuring correct DMA operations for passthrough devices. To maintain DMA correctness, memory overcommitment must synchronize updates to the IOPT whenever the underlying physical pages are reclaimed or remapped. The cold-page reclamation procedure involves three page tables, the guest page table (GPT), the extended page table (EPT) and the I/O page table (IOPT), consuming non-trivial CPU resources. Restoring a cold page performs the reverse steps of reclaiming a cold page.

The host-level swapping, which is usually referred to as *uncooperative swapping* meaning that hypervisor swaps out data and reclaims host pages without cooperating with the guest VM, is usually less efficient due to a lack of VM visibility (detailed in §3.2.3). In particular, double-paging issues [53] can arise when both host-level and guest-level swapping are enabled independently. In this case, the guest may attempt to swap out a page that has already been swapped out by the host. This double-paging behavior can cause the page to be faulted back into memory, only to be re-evicted shortly thereafter, leading to redundant data movement.

Pressure Stall Information (PSI) . To assess the performance risks of memory overcommitment, we leverage Pressure Stall Information (PSI) [58], which quantifies the fraction of time that tasks are stalled due to resource shortages. To adopt PSI as a common control and evaluation lens for VM memory overcommitment with memory reclaim allows us to compare systems under a consistent "bounded pressure" objective.

3 Motivation

In this section, we first demonstrate that the datacenter applications contain a large amount of cold memory (§3.1). Then

we discuss why existing overcommitment mechanisms fail to effectively reclaim cold memory (§3.2).

3.1 Cold-Memory Reclamation Opportunity

Prior studies [38,40,50,58] show that datacenter applications keep a large amount of cold memory (*e.g.*, 19–62% in Meta's clusters). This finding suggests considerable reclamation opportunity: a large fraction of cold pages can be reclaimed with negligible performance impact and repurposed to satisfy memory demands from busy tenants or newly spawned tasks.

To better quantify the upper bound of this opportunity, we evaluate seven representative datacenter applications (see Table 2) without THP. Disabling THP avoids hotness amplification and mixed-granularity artifacts, allowing us to measure the cold-memory reclamation space under ideal fine-grained page accounting. Guided by PSI [58], we dynamically swap out memory pages by triggering Linux swapping, using an RDMA-based far-memory backend [49]. By tuning the PSI threshold to bound performance degradation within 5%, we measure the corresponding amount of swapped-out memory to quantify the reclamation opportunity. In our experiments, we observe that 33–49% of application memory can be swapped out with less than 5% performance degradation.

However, turning this large cold-memory reserve into practical benefits in virtualized environments is non-trivial. Strict per-VM memory isolation limits the ability to reclaim cold memory, since adjusting VM allocations requires overcommitment mechanisms. Unfortunately, as we show next, existing mechanisms struggle to effectively exploit this cold-memory reclamation opportunity.

3.2 Inefficiencies in Existing Mechanisms

Existing overcommitment mechanisms are primarily optimized for free memory. Reclaiming cold memory instead requires cooperation with guest- or host-level swapping, both of which exhibit fundamental inefficiencies. To illustrate these inefficiencies, we evaluate a representative state-of-the-art mechanism, HyperAlloc [59].

3.2.1 THP Complicates Cold-Memory Reclamation

To reduce page-table-walk overhead, modern virtualized systems often enable THP, typically configured in the `always` mode. However, THP introduces additional complexity for cold-memory reclamation. We repeat the earlier opportunity study (§3.1) with THP enabled, keeping all other settings by default. In our experiments, the available cold-memory reclamation space drops to 16–25% of application memory.

Two factors drive this degradation. First, enabling THP results in a mixture of 2 MB and 4 KB pages for multiple reasons, such as Linux's allocation behavior, fine-grained deallocations, and subpage-level permission changes. These

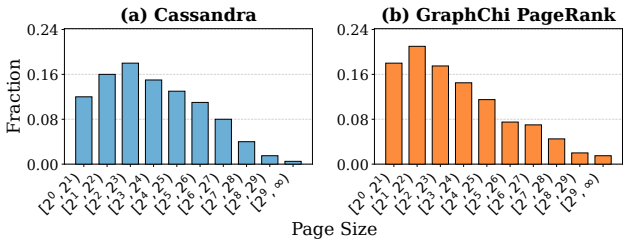


Figure 3: Cold memory fraction by region size at mid-phase. Cold memory is defined as untouched memory in 10s time window, monitored by DAMON [47,48].

mixed-granularity pages are managed uniformly (*e.g.*, `folio`), creating unfair hotness ordering. A 2 MB page aggregates 512 base pages and therefore has many more chances to be accessed, making it appear much hotter than scattered 4 KB pages. As a result, the swapping often evicts hot 4 KB pages while overlooking cold 2 MB pages, ultimately degrading application performance. We refer to this phenomenon as *unfair hotness*. Second, THP suffers from the well-known *hot bloat* problem: a few hot 4 KB subpages can cause the entire 2 MB huge page to appear hot, preventing the system from reclaiming the numerous cold subpages that it contains. Several prior efforts [35,36,41] attempt to mitigate this effect by selectively splitting hot-skewed huge pages. However, they struggle to balance the overheads of fine-grained monitoring and splitting against the potential reclamation benefits under memory overcommitment. Moreover, the 4 KB pages created by proactive splitting further exacerbate the unfair hotness problem.

Together, the effects of **unfair hotness** and **hot bloat** highlight the need for a THP-aware cold-memory reclamation design, rather than relying on an existing paging system that remains oblivious to the unfair and skewed hotness of huge pages.

3.2.2 Limitations of Guest-Level Swapping

We evaluate guest-level-swapping-augmented HyperAlloc on representative datacenter applications, tuning PSI threshold to bound performance degradation within 5%. We enable THP in both the guest and the host, following the recommended practice for virtualized environments [29]. In our experiments, HyperAlloc reclaims only 35–47% of the cold memory identified as reclaimable in prior profiling (§3.2.1). The shortfall stems from a fundamental granularity mismatch. HyperAlloc performs reclamation at a granularity of 2 MB huge pages to reduce the overhead of page-table operations. However, cold pages in real applications are highly scattered (as shown in Figure 3), and guest-level swapping faces a granularity dilemma. On one hand, swapping a 2 MB huge page as a whole preserves contiguity but exacerbates read/write amplification effects. On the other hand, swapping at 4 KB granularity produces fragmented free pages, making it hard for HyperAlloc to reclaim at 2 MB granularity, thus leaving most

Table 1: TLB flush comparison between host-level and guest-level swapping HyperAlloc under KMeans workload.

System	TLB Flush	KMeans Elapsed (s)
HyperAlloc-H	14819979	142.58
HyperAlloc-G	2946318	56.77

cold memory unreclaimable. We follow Linux's default behavior: THPs are swapped out as a whole, but swap-in happens at 4 KB granularity. The reason is that (1) the kernel usually cannot find a contiguous 2 MB free page for THP swap-in, and (2) swapping in a 2 MB page would severely stall application execution as swapping in lies at the critical path. Under this configuration, about 43% of overall reclaimable memory is at 4 KB granularity, directly limiting the effectiveness of 2 MB-based reclamation. Other designs that rely on 2 MB huge pages, such as virtio-mem [24] and hugepage-balloon [25], suffer from the same limitation.

A natural question is whether simply reducing the granularity can close this gap. To answer this question, we extend HyperAlloc to operate at the 4 KB base-page granularity while still opportunistically reclaiming 2 MB huge pages when available. We refer to this variant as HyperAlloc-4K. Under the same experimental setup, HyperAlloc-4K reclaims only 46–60% of the reclaimable cold memory. The root cause is the significant overhead of fine-grained page-table operation. Reclaiming and restoring a single 4 KB cold page of VM incur 36 µs and 21 µs, respectively, latencies that are 1.7× and 2.1× higher than those of guest-level swapping alone. The additional page-table overhead introduced by overcommitment slows both reclamation and restoration. The reduced reclamation throughput prevents HyperAlloc-4K from reclaiming cold pages at the rate that they are identified, while the high restoration latency lies directly on the application's memory-access critical path, tightening the allowable performance headroom. Together, these factors significantly limit the amount of cold memory that can actually be reclaimed. Similar drawbacks arise in other 4 KB-granularity mechanisms (*e.g.*, V-Probe [56] and Balloon [53]), which face the same tradeoff between granularity and page-table overhead.

In summary, guest-level-assisted mechanisms inevitably face a fundamental granularity tradeoff. Using 2 MB huge pages amortizes overhead of page-table operations, but the inherently non-contiguous distribution of cold memory limits the applicability of hugepage-level reclamation. Extending the design to support 4 KB granularity increases coverage, yet the high overhead of 4 KB page-table operations lowers throughput and tightens performance constraints. These limitations, **granularity mismatch** and **high-latency reclamation and restoration at 4 KB granularity**, highlight the need for a low-latency design that supports mixed granularity.

3.2.3 Limitations of Host-Level Swapping

We use HyperAlloc to reclaim free memory and delegate cold-memory reclamation to host-level swapping. However, when applied to VMs, host-level swapping suffers from well-known limitations. First, the host swapping cannot observe guest memory accesses. To overcome this limitation, we integrate Linux PTE scanning with the KVM MMU notifier, enabling the host to track guest accesses by scanning and clearing EPT access bits. Second, the host cannot distinguish free memory from cold memory, since the entire VM footprint appears as allocated memory. We bridge this semantic gap by exposing HyperAlloc's guest-allocator metadata to the host, allowing host-level swapping to bypass free guest pages. Under the same experimental setup (§3.1), this hybrid design reclaims only 34–55% of the reclaimable cold memory.

One primary bottleneck is the high overhead of tracking page hotness at the host level. As shown in Table 1, host-level swapping performs 5.0× more TLB flushes than guest-level swapping to reclaim the same amount of cold memory, including expensive full TLB flushes. These flushes directly degrade application performance, a finding also reported in prior work [26]. Moreover, host-level swapping is invisible to the guest. As a result, guest kernel services (such as khugepaged and kcompactd) may access reclaimed pages, triggering unnecessary page restorations. Our measurements show that 26% of all restorations under host-level swapping are caused by such system-service accesses. These unintended restorations affect the amount of reclaimed memory.

In summary, the preceding limitations—**heavy hotness tracking** and **guest-blind reclamation**—are inherent to host-level swapping and persist regardless of the overcommitment mechanism used for free memory. These limitations highlight the need for a guest-aware design with lightweight hotness monitoring.

4 Design and Implementation

4.1 Design Goals

To guide the design of Blowfish, we distill three fundamental observations. First, cold memory is inherently non-contiguous, making granularity tradeoff inevitable: huge pages inflate actual hotness, while 4 KB pages incur excessive overheads. Therefore, effective reclamation should operate and be optimized at mixed granularity. Second, accurate hotness information cannot be efficiently obtained at the host level due to limited visibility. Third, reclamation should be guest-aware to prevent misaligned or unintended interactions with guest kernel services. These observations point to three key design requirements.

- Lightweight mixed-granularity hotness tracking: monitor page temperature with minimal overhead, identify

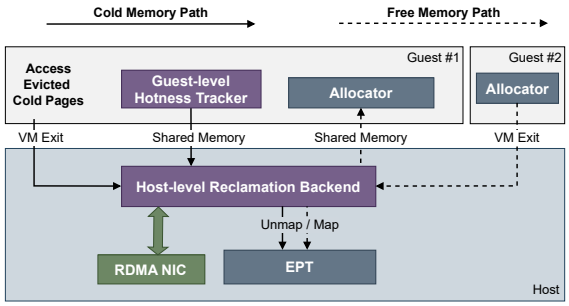


Figure 4: Design overview of Blowfish.

cold subpages within huge pages, and place mixed-granularity pages under a fair hotness comparison.

- Efficient mixed-granularity operations: enable low-latency fine-grained (4 KB) reclamation/restoration while preserving contiguity when beneficial.
- Guest-aware reclamation: align host-level overcommitment with guest kernel services to prevent blind restorations.

4.2 Design Overview

Following the requirements, we introduce Blowfish, whose architecture is shown in Figure 4. At a high level, Blowfish leverages guest-level hotness tracking to instruct the host-level reclamation to enable efficient overcommitment.

At the guest level, Blowfish piggybacks on existing Linux MGLRU, and dynamically splits or collapses THPs to capture subpage coldness. It also adjusts MGLRU's promotion rules to ensure fair comparison between 4 KB and 2 MB pages (§4.3). At the host level, Blowfish performs reclamation and restoration without modifying GPT or IOPT, avoiding expensive page-table operations (§4.4). Blowfish further leverages guest-reported hotness to coordinate guest kernel services and prevent behaviors that conflict with host overcommitment (§4.5).

We describe each component of Blowfish in the following sections, and then present a practical automatic policy (§4.6) together with the mechanisms that ensure correct behavior under limited guest cooperation (§4.7).

4.3 Guest-Level Hotness Tracker

A key design choice in Blowfish is to build on Linux's existing MGLRU mechanism for hotness tracking. Unlike hardware-assisted schemes (*e.g.*, PEBS [26,35] or PML [18,52]), which introduce additional accuracy-overhead trade-offs, reusing MGLRU gives Blowfish an effective in-kernel hotness ordering without new monitoring costs. However, performing hotness tracking at the host level would require page-table scans that trigger expensive global TLB flushes (as shown in §3.2.3). To avoid heavy costs, Blowfish shifts hotness tracking into the guest and integrates it into the guest MGLRU.

To mitigate unfair hotness and hot bloat problems introduced by THP, Blowfish makes two lightweight extensions: a fair promotion rule of MGLRU that equalizes hotness treatment across 4 KB and 2 MB pages (§4.3.1), and a subpage tracker that refines THP-level hotness (§4.3.2).

4.3.1 Fair MGLRU

Traditional MGLRU promotes a page of any granularity to the youngest generation when its PTE or PMD is scanned as accessed, and gradually ages unaccessed pages until they reach the oldest generation for reclamation. However, this policy implicitly favors huge pages. A 2 MB page is far more likely than a 4 KB page to receive accesses, allowing 2 MB huge pages to stay young disproportionately longer and making them much harder to reclaim. This page-size-amplified hotness distortion is the root cause of unfair hotness.

Based on this observation, Blowfish introduces Fair MGLRU. The key idea is to discount the reward associated with an access to a 2 MB page: a 2 MB page should not receive the same reward as a 4 KB page, because the probability of observing an access on a 2 MB page is inherently 512× higher than on a 4 KB page. Fair MGLRU preserves the original promotion behavior for 4 KB pages, that is, each access immediately promotes a 4 KB page to the youngest generation. For 2 MB pages, in contrast, Fair MGLRU performs incremental promotion: upon each observed access, the page is promoted only to the next younger generation, rather than being promoted to the youngest one in a single step. This design exploits MGLRU's structure. Since reclamation drains only the oldest generation, a truly hot 2 MB huge page will still climb toward younger generations through repeated accesses, whereas a huge page that is only moderately warm will eventually drift into the oldest generation and become reclaimable.

We note that incremental promotion is meaningful only when multiple generations exist. With two generations, the original and Fair MGLRU behave identically. Linux, however, uses four generations by default, so Fair MGLRU is effective in practice, which is also confirmed by evaluation (§5).

4.3.2 Subpage Tracker

To address hot bloat, Blowfish introduces a subpage tracker that identifies cold 4 KB base pages inside a 2 MB huge page. Prior approaches face inherent limitations. HugeScope [36] obtains subpage-level access bits by splitting EPT entries and relying on transient inconsistencies in EPT state; this approach becomes impractical in overcommitment settings where EPT updates are frequent. PEBS-based sampling [35], on the other hand, incurs well-known accuracy-overhead trade-offs and can easily miss accesses. In contrast, Blowfish's

subpage tracker is designed to maintain both low overhead and high accuracy.

Blowfish's subpage tracker works by dynamically splitting and re-coalescing guest huge-page mappings to expose the PTE-level access bits of constituent 4 KB subpages. The core observation is that tracking accesses on subpages requires splitting only the page table mapping, instead of the underlying physical huge page, avoiding heavy `struct page` overhead (the main overhead as reported in previous work [36]).

Specifically, the subpage tracker operates in periodic epochs. Every 100 ms, it selects a small set of huge pages from the second-youngest MGLRU generation, where hotbloat pages naturally concentrate, and temporarily splits their mappings, *i.e.*, turning PMD-mapped THP to PTE-mapped THP. We cap each batch to at most 32 huge pages to bound overhead. After splitting, the tracker monitors subpage access bits in a 100 ms window. All the parameters are empirically determined to impose negligible overhead.

After the monitoring phase, the tracker classifies each huge page based on access skewness. If fewer than 20% of its subpages are accessed, the tracker considers the huge page hotbloat and reports its cold subpages to the host for reclamation. Regardless of whether a huge page exhibits access skewness, the subpage tracker re-coalesces huge-page mapping to avoid generating extra fine-grained page table entries. For huge pages with more than 80% of subpages accessed (*e.g.*, evenly hot), the tracker marks them as balanced and skips them in the next epoch to avoid unnecessary splitting.

4.4 Host-Level Reclamation Backend

While Blowfish performs hotness tracking inside the guest to minimize monitoring overhead, relying on guest-level swapping for overcommitment introduces substantial latency of modifying the guest page table and the IOPT (see §3.2.2). In contrast, Blowfish introduces a host-level reclamation backend that executes both reclamation and restoration entirely on the host side and optimizes the critical path of each operation. The backend consumes guest-reported hotness, reclaims cold pages through host-managed page remapping, and supports restoring pages without involving the guest page table or IOPT. This design is based on the following observations. (1) The isolation boundary for VM memory is enforced by the EPT, not the guest page table, meaning that the guest page table can remain unmodified during overcommitment while the host manages reclamation purely via EPT remapping. (2) Moving the RDMA-based data path from the guest into the host avoids the expensive IOPT updates that passthrough NICs would otherwise trigger during data movement.

We next detail the procedures of the reclamation and restoration. Unless otherwise noted, we describe reclamation and restoration without distinguishing page granularity. The operations for 2 MB and 4 KB pages are structurally identical: the former operates on PMD entries and the latter on PTE entries.

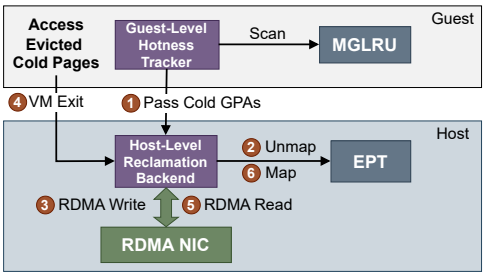


Figure 5: Cold memory path of Blowfish.

Reclaim a cold page. As shown in Figure 5, the guest-side hotness tracker selects cold pages from the oldest MGLRU generation, moves them to an isolated list to avoid repeated reclamation, and pushes their guest physical addresses into a shared-memory channel. On the host side, Blowfish's reclamation backend runs a lightweight daemon thread that ❶ polls this channel for incoming guest physical addresses. The thread sleeps when the channel is empty and can be awakened via a hypercall, following a standard low-overhead design that avoids occupying a dedicated core. The reclaiming process proceeds as follows. ❷ Upon receiving a cold guest physical address, the backend invalidates the corresponding EPT entry and flushes its TLB entry. This mechanism guarantees that any subsequent access will trigger an EPT violation, enabling Blowfish to intercept the access and restore the page safely or prevent in-flight modifications before data backup. ❸ The backend extracts the host physical address from the invalidated EPT entry, allocates a far-memory address, and transmits the page's content to far memory via RDMA. After the transfer completes, the backend encodes the far-memory address into the EPT entry's upper bits, which are ignored when the present bits of EPT entry are cleared [28]. This process finalizes reclamation without modifying either the GPT or the IOPT.

Restore a cold page. As shown in Figure 5, when the guest accesses a reclaimed cold page, ❹ the access triggers an *EPT violation* because the corresponding EPT entry is not present. Blowfish installs a lightweight EPT violation handler in the host-level backend to restore both the mapping and the data for the faulting guest physical address with minimal latency. ❺ Upon a fault, the backend extracts the far-memory address embedded in the invalidated EPT entry, allocates a free host page, and fetches the data from far memory. ❻ It then reconstructs the EPT entry for the restored guest page and places the guest physical address into a shared buffer, enabling the guest to reinsert the page into its MGLRU structures during routine maintenance. This restoration completes with only a single far-memory read and one EPT update, introducing min-

imal overhead and mirroring the low-cost design principles of Blowfish.

Reclaim and restore free pages. Identifying free pages is straightforward: the backend only needs to scan the guest's free page list or allocator metadata. Reclaiming and restoring free pages are also simple and efficient, as they merely require updating the corresponding EPT entries and can naturally benefit from defragmentation and huge-page optimizations [59,60]. We extend HyperAlloc [59] and integrate it into Blowfish's unified reclamation backend, delegating both free page reclamation and restoration. Specifically, the backend scans the shared guest allocator to identify free pages, unmaps their EPT entries, and synchronizes their state with the guest allocator using atomic updates. This process completes the reclamation of a free page. When the guest later allocates a page marked as reclaimed, it issues a hypercall to the backend, which allocates a fresh host page and reconstructs the corresponding EPT mapping. The guest allocator naturally prefers pages not marked as reclaimed, thereby avoiding unnecessary restorations.

4.5 Hotness-Aware Kernel Service

Modern Linux kernel services (*e.g.*, kcompactd, khugepaged) make page-level decisions without considering access hotness, a long-standing inefficiency in Linux memory management. Under host-level reclamation, however, these services can unnecessarily trigger restorations of reclaimed pages, amplifying their cost. To mitigate all the preceding problems while preserving normal kernel functionality, Blowfish introduces minimal but effective hotness-aware adjustments to existing guest kernel services.

Among these services, khugepaged is both always enabled under THP and highly active [6]. It periodically scans guest page tables with bounded throughput to identify 2 MB regions that can be promoted to huge pages, moving data to physically contiguous 2 MB pages and forming a new huge-page mapping. Because khugepaged is oblivious to page hotness, it may (1) merge unevenly hot pages, exacerbating hot bloat, and (2) inadvertently touch reclaimed cold pages under host overcommitment. Blowfish augments khugepaged with a lightweight filter: during the scan, it inspects access bits of all PTEs in the candidate region and skips promotion if more than half of the pages were not recently accessed. This simple check adds little extra scanning cost and effectively suppresses low-benefit promotions in practice.

kcompactd is triggered when the guest kernel requires high-order pages, *e.g.*, when khugepaged finds no available free 2 MB pages. It compacts memory by migrating data from lower addresses toward higher addresses to create contiguous free pages. During scanning, kcompactd may migrate hot pages, hurting application performance, or touch reclaimed cold pages, triggering unnecessary restorations. Blowfish extends kcompactd with a lightweight filter: during the scan, it inspects the generation of each page and skips pages in the youngest generation (treated as hot). It also skips reclaimed pages by checking whether the page is absent from the MGLRU; in Blowfish, all reclaimed cold pages are removed from the MGLRU to avoid repeated reclamation. In more aggressive paths (*e.g.*, direct compaction), Blowfish issues explicit restoration requests to the host backend and allows hot and reclaimed pages to be migrated, ensuring that memory compaction remains effective. Since direct compaction occurs infrequently and involves only a limited number of pages at a time, these restorations introduce only modest overhead.

We primarily optimize khugepaged and kcompactd because, with THP enabled, khugepaged is always active, kcompactd is frequently triggered during huge-page allocation, and together they account for the vast majority of unnecessary accesses in our experiments. Other kernel services, such as NUMA balancing, are either inactive in typical VM deployments or can be extended in a similar fashion, so we omit their details.

4.6 Automatic Policy

In this section, we present an automatic policy used in our evaluation: (1) prioritize reclaiming free memory, whose pages can be efficiently restored with negligible impact on application performance; and (2) reclaim cold memory only when free memory becomes scarce, while regulating its performance impact through PSI. This PSI-guided policy is purely practical and heuristic; Blowfish does not rely on a specific control policy, and alternative policies can be easily plugged in.

Initially, Blowfish tries to reclaim free memory until the remaining amount reaches a predefined watermark (*e.g.*, 100MB by default), a design inspired by the Linux memory watermark. Specifically, the host-level reclamation backend periodically scans the guest allocator every 100ms to identify and reclaim free guest pages, a frequency that has proven effective in practice with minimal CPU consumption. Restoration of reclaimed free memory is performed on demand when the guest allocates it, following HyperAlloc's policy.

Once the remaining free memory falls below the predefined watermark, Blowfish begins reclaiming cold memory, with the reclamation rate regulated by PSI to minimize application performance degradation. The PSI follows the definition introduced in §2, *i.e.*, the real-time process stalls due to memory shortages. Specifically, when PSI remains below a predefined threshold, Blowfish continues reclaiming cold memory. As reclamation progresses, reclaimed pages may be accessed more frequently, causing PSI to rise. Once PSI exceeds the threshold, Blowfish pauses cold-memory reclamation and proactively restores cold pages until PSI falls back below the threshold.

4.7 Guest Cooperation and Isolation

Similar to prior cooperative memory overcommitment mechanisms (*e.g.*, Balloon [25, 53], virtio-mem [24], HyperAlloc [59]), Blowfish operates under the assumption that the guest and the host adhere to a simple, well-defined protocol. In practice, this cooperation model is widely adopted in modern cloud virtualization environments, where guests typically adhere to the prescribed interface. Nevertheless, protocol deviations, whether accidental or intentional, may still arise. We discuss how Blowfish handles such cases and what guarantees it preserves. We focus on the mixed-granularity cold-memory reclamation path unique to Blowfish and exclude free-memory reclamation, whose correctness properties are inherited from HyperAlloc.

For cold-memory reclamation, a guest may provide guest physical addresses that do not correspond to valid memory regions. Blowfish's reclamation backend verifies each guest physical address against the EPT mappings and simply ignores those that are invalid. A guest may also fail to report its cold guest physical addresses. In this case, if necessary, Blowfish can safely fall back to the uncooperative host swapping, which identifies cold pages by scanning EPT access bits and directly reclaims guest-occupied host pages. This fallback preserves correctness and, in critical situations, prevents host-level out-of-memory conditions, though it may reduce the guest's performance benefits. If a guest submits an unusually large number of guest physical addresses, the backend continues to process only the valid ones. The backend employs a balanced polling strategy across all VMs, ensuring that excessive submissions from one VM do not impact the progress or performance isolation of others.

We also handle concurrent host and guest operations. On the guest side, pages selected for reclamation are synchronized using LRU locks. For the shared memory channel between host and guest, atomic indexing ensures that each location is accessed by at most one reader or writer at a time [55]. On the host side, Blowfish coordinates concurrent reclamation and restoration using per-EPT-entry locks.

5 Evaluation

We implement Blowfish in C, based on the Linux kernel (version 6.1) and QEMU (version 8.2.1), with approximately 6000 lines of code added to the host kernel and 1700 lines to the guest kernel. The RDMA backend is inspired by Hermit [49], a state-of-the-art paging-based far memory system.

Setup. We run experiments in a cluster with one CPU server and one memory server connected by NVIDIA 100Gbps Mellanox ConnectX-5 InfiniBand adapters. Each server has two Intel Xeon Gold 6342 CPUs and 256GB of memory. We configure the servers following common practice for low latency [45], including disabling Turbo Boost, CPU frequency scaling and hyperthreading. Besides, we enable Transparent Huge Page (THP). All systems being evaluated are running on Ubuntu 20.04. The virtual CPUs on the guests maintain a 1:1 mapping with the host physical CPUs. The threads for reclamation and proactive restoration share physical CPU cores with the VM and utilize no more than a single core.

Baseline and methodology. We evaluate against HyperAlloc [59], and its 4 KB-granularity variant HyperAlloc-4K for fine-grained 4 KB reallocation. Because HyperAlloc targets free-page reclamation, we extend HyperAlloc and HyperAlloc-4K to support cold-memory rebalancing: we install Hermit [49] for µs-scale data movement and add a reclaimer to support paging. All baselines are configured to follow the same policy as Blowfish.

For paging type (§2.2), guest paging offers low overhead of hotness tracking but introduces granularity mismatch and high-latency for memory adjustment; host paging is transparent but risks mis-eviction and extra context switches/data movement due to limited visibility and burdens heavy hotness tracking. We pair HyperAlloc and HyperAlloc-4K with guest paging to enable cold-page reclamation, where HyperAlloc-H denotes the host-paging variant of HyperAlloc, HyperAlloc-G denotes its guest-paging variant, and the same notation applies to HyperAlloc-4K (*i.e.*, HyperAlloc-4K-H and HyperAlloc-4K-G). Since HyperAlloc and HyperAlloc-4K show no significant performance difference when reclaiming contiguous free memory, the host paging responsible for reclaiming cold memory is implemented only on HyperAlloc (*i.e.*, HyperAlloc-H).

Slow Down for throughput is defined as the ratio between the runtime increase under the current configuration and the runtime when using all-local memory, while for latency, it represents the percentage of latency increase. *Reclamation Ratio* is formally defined as

$$ReclamationRatio = 1 - \frac{1}{n} * \frac{\sum M_i^{now}}{M^{VM}}$$

where n denotes the number of sampling points. M_i^{now} represents the memory usage of the VM at sampling point i, and M^{VM} denotes the initial memory size allocated to the VM. We measure the throughput and latency of applications under different PSI thresholds.

For §5.1, we use different baseline systems to reclaim as much VM memory as possible under the PSI threshold constraint, in order to assess their ultimate capabilities. For §5.2, we follow the unified PSI threshold setting used in Senpai [58], namely 0.04. We run different applications in pairs under the condition that their combined memory usage is at most 50% of their combined peak memory demand to investigate how different systems perform memory rebalancing.

Note that for host-level reclamation mechanisms (*e.g.*, HyperAlloc-H and Blowfish), guest-level PSI cannot accurately reflect the actual memory demand, because host-level page faults are invisible to the guest. To address this gap, we place each VM process in a separate host cgroup and monitor the PSI at the cgroup level for these mechanisms. In this

Table 2: Benchmarks and workloads.

Application	Dataset	Size	Characteristics	Memory Footprint	#vCPU
Native applications					
Memcached [3]	Meta's trace Cachelib [14]	411M records	Highly skewed with churn	20G	8
Liblinear [20]	Kaggle 2014 Criteo [2]	40.4M records	Sparse sequential access	8G	4
GAPBS [13] Triangle Count	Twitter trace [32]	61.5M vertices, 2.4G edges	Localized access with skewed distribution	20G	4
GraphChi [34] Page Rank	partial Hyperlink2014 [46]	812M vertices, 33G edges	Block-based random access with locality	23G	4
SPEC CPU2017 [15] 602.gcc_s	N/A	N/A	Irregular, pointer-heavy memory access with low data locality	10G	4
Managed applications					
Spark-KMeans [62]	Wikipedia France [4]	137.5M records	Sequential access with spatial locality	15G	4
Cassandra [11]	YCSB, Synthetic, Zipf distribution	10M records	Random access with skewness	36G	8

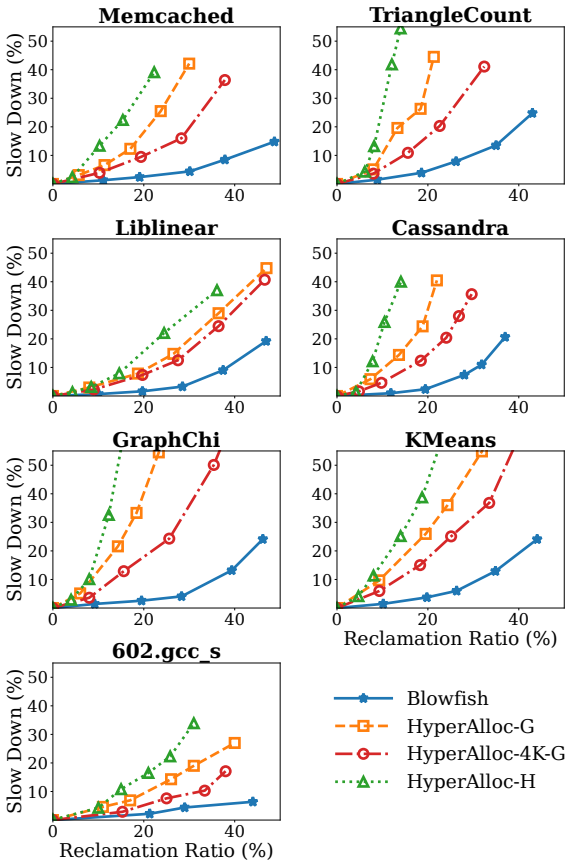


Figure 6: End-to-End performance of benchmarks.

way, host-level PSI captures memory pressure induced by host-level page faults during restoration, providing a more accurate measurement of memory demand.

Workloads. As shown in Table 2, we evaluate five native applications written in C/C++ and two managed applications written in Java, covering varied memory access patterns. We evaluate the throughput (§5.1.1) and latency (§5.1.2) of applications listed in Table 2.

5.1 Performance of Single VM

5.1.1 Throughput

At the coordinate origin, the memory overcommitment mechanism is disabled, such that no system engages in memory allocation or reclamation, and applications observe no slowdown. Across different PSI thresholds, Blowfish demonstrates superior performance by achieving a higher reclamation ratio and incurring less performance degradation than the baselines.

For application performance, Blowfish uses fair MGLRU to get the hotness of THP, and avoids maintaining the guest page table (GPT) and IOPT, enabling faster memory provisioning when application demand increases. Moreover, its more efficient reclamation and restoration reduce CPU contention with applications. For memory reclamation ratio, rather than depending on contiguous memory pages, Blowfish performs adjustments at the granularity of 4KB pages through a dedicated mechanism, thereby enabling fast reclamation of a substantial amount of non-contiguous memory. Although HyperAlloc-H avoids page table modifications by adopting host paging, the applications running on it perform worse than those on Blowfish and other variants of HyperAlloc. The underlying causes are twofold: (1) HyperAlloc-H relies on heavy hotness tracking at the host level, inducing up to 4.78× more TLB flushes than HyperAlloc-G. (2) Due to its lack of program semantics, host paging often misidentifies cold pages, resulting in unnecessary evictions, context switches (incurring 33% more VM exits), and extra data transfers. For reclamation, HyperAlloc-H suffers from host-side semantic loss: it fails to differentiate page states (*e.g.*, short-lived cache, writeback pages), leading to misguided evictions and unnecessary swapping. In addition, daemon in kernel service introduces hotness interference, and the absence of guest-level semantics further causes invalid reclamations that are quickly restored, collectively degrading throughput. HyperAlloc-G imposes minimal performance overhead on applications, but fails to take advantage of potential memory reclamation opportunities due to its dependence on contiguous pages. In HyperAlloc-4K-G, costly cold memory reclamation reduces its reclamation throughput (on average 2× lower than Blowfish across the applications), and costly restoration increases reclaimed-page access latency, and this higher latency degrades application performance.

Memcached and Cassandra The workload of Memcached is skewed with churn, while the one of Cassandra is more random. This behavior stems from Cassandra's garbage collection threads accessing memory via graph traversal. Under the constraint that application performance degradation does not exceed 5%, for average reclamation ratio, Blowfish has 3.1×, 5.3×, and 3.2× improvement compared to HyperAlloc-4K-G, HyperAlloc-H, and HyperAlloc-G, respectively in Memcached. Besides, in Cassandra, provided that application performance degrades by no more than 5%, Blowfish improves the reclamation ratio by 2.1×, 4.0×, and 3.3×. The reclamation of Cassandra is more challenging due to its larger working set and more complex memory access patterns. Our profiling reveals that more than 80% of reclaimable free memory pages are non-contiguous, severely limiting HyperAlloc's ability to reclaim memory. For HyperAlloc-H, Memcached consists of 96% anonymous pages versus 67% in Cassandra, yielding higher reclamation ratio as fewer file pages are misevicted. However, due to the semantics gap, 27% of reclaimed pages are restored within 1s, resulting in a lower reclamation ratio compared to Blowfish.

GAPBS TriangleCount and GraphChi PageRank are graph-centric workloads that traverse the entire graph and exhibit pronounced phase-changing behavior in their memory usage patterns. TriangleCount is executed on an evolving graph [17], where we divide the input dataset into three parts and feed them to GAPBS. During the update and analysis of the evolving graph, HyperAlloc-4K-G cannot reclaim the memory in time due to its low reclamation throughput. Subject to the condition that application performance drops by no more than 5%, Blowfish offers a favorable trade-off between reclamation and performance, achieving up to 2.1×, 2.8×, and 2.4× higher reclamation ratios than baseline systems (HyperAlloc-4K-G, HyperAlloc-H, and HyperAlloc-G, respectively). GraphChi PageRank reads and processes the graph in blocks, during which it repeatedly allocates and frees memory. The frequent memory allocation and deallocation in GraphChi [33] exacerbate the problem of memory fragmentation. Its slowdown on HyperAlloc-H stems primarily from (1) frequent TLB flush. (2) reclaiming an extra 48% short-lived page cache pages; this behavior interferes with hot-cold classification, causing many hot pages to be swapped out. In our measurements, VM exits due to this factor account for 32% (296M in total). For HyperAlloc-4K-G, rapid working set fluctuations lead to excessive memory restoration operations, incurring high overhead. Blowfish detects the gaps during application working set transitions and promptly reclaims memory. At the constraint of 5% slowdown, Blowfish reaches 3.4×, 6.1×, and 3.9× higher memory reclamation ratio than HyperAlloc-4K-G, HyperAlloc-H, and HyperAlloc-G, respectively.

Liblinear and KMeans are representative of machine learning applications. The memory fluctuation characteristics of Liblinear are subdued because matrix multiplication exhibits sequential access patterns, where non-contiguous pages account for only 34% of the total. Hence, the improvement of Blowfish is modest compared to the baselines. With respect to HyperAlloc-4K-G, HyperAlloc-H, and HyperAlloc-G (at 5% slowdown), Blowfish improves the reclamation ratio by 2.0×, 2.9× and 2.2×, respectively. KMeans is an iterative algorithm composed of multiple phases. Furthermore, as KMeans has a relatively large working set, the reclamation ratio tends to stabilize with increasing PSI threshold. In KMeans under a slowdown of 5%, for reclamation ratio, Blowfish has 3.0×, 5.6×, and 3.7× improvement compared to HyperAlloc-4K-G, HyperAlloc-H, and HyperAlloc-G, respectively.

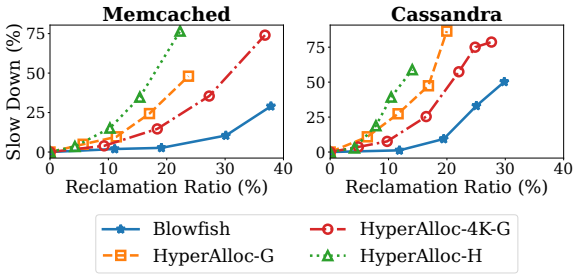


Figure 7: Tail latency of benchmarks.

602.gcc_s is a benchmark that simulates the GCC compiler compiling large C programs, displaying multi-phase memory usage characteristics. HyperAlloc-G's reclamation ratio remains significantly lower across PSI thresholds, owing to its limited ability to reclaim non-contiguous memory (up to 75%). The memory access pattern of 602.gcc_s aligns well with Blowfish's optimizations. With a slowdown of 5%, Blowfish outperforms HyperAlloc-4K-G, HyperAlloc-H, and HyperAlloc-G by 1.6 ×, 3.2 × and 1.9× for reclamation ratio. Notably 52% of the pages swapped out by HyperAlloc-H are page cache.

5.1.2 Latency

We evaluate the tail latency (p95) of Memcached and Cassandra. As shown in Figure 7, with increasing reclamation ratio, applications running on HyperAlloc-G and HyperAlloc-H exhibit a sharp increase in p95 latency. Besides low restoration overhead, Blowfish consumes only 11% and 14% of the CPU resources allocated to the VM for Memcached and Cassandra, respectively (compared to 24% and 26% in HyperAlloc-4K-G), reducing competition with application threads during memory rebalancing. At the challenging case with a throughput slowdown of 5%, Blowfish reduces the latency slowdown of Memcached by 2.8×, 4.7×, and 3.9× compared to HyperAlloc-4K-G, HyperAlloc-H, and HyperAlloc-G, respectively (1.9×, 2.5×, and 2.1× for Cassandra).

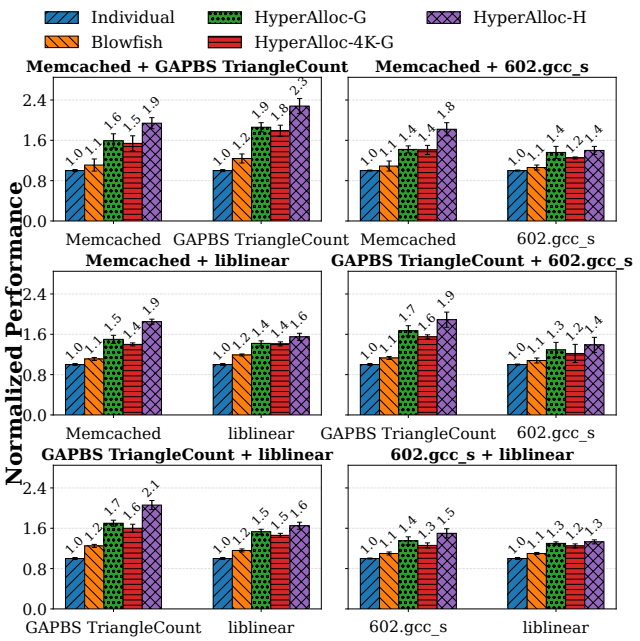


Figure 8: Normalized end-to-end performance when co-running workloads. Lower is better.

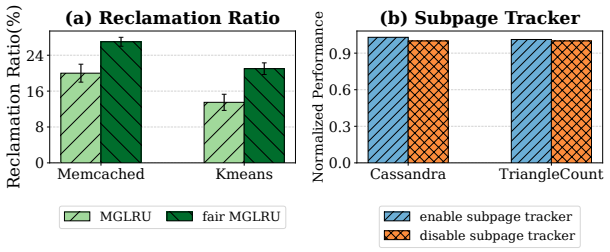


Figure 9: (a) Effectiveness of Fair MGLRU. (b) Normalized performance with and without Subpage Tracker. Lower is better.

5.2 Memory Rebalancing Across VMs

We select four applications and evaluate end-to-end memory rebalancing under co-running VMs. Figure 8 shows that Blowfish consistently preserves co-run performance closer to running individually ("Individual" refers to running an application in isolation on a VM without memory pressure). Across all six workload pairs, Blowfish's normalized performance degradation factor remains in a narrow range (1.06×–1.25×), whereas the baselines exhibit substantially larger degradation. Blowfish can quickly adjust the amount of memory available to an application by leveraging its phase-changing behavior, thereby avoiding SLO violations.

5.3 Drill Down

Effectiveness of Fair MGLRU. We measure the effectiveness of fair MGLRU. We select two applications and replace

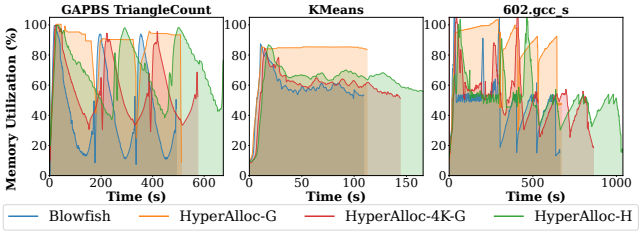


Figure 10: Memory utilization over time during rebalancing, defined as the ratio of the current VM size to the initial VM size.

the fair MGLRU in Blowfish with the standard MGLRU. As shown in Figure 9(a), fair MGLRU effectively distinguishes pages at different granularities and improves the memory reclamation ratio. Enabling fair MGLRU increases the reclamation ratio by 7.2% for Memcached and by 6.5% for KMeans.

Overhead of Subpage Tracker. To investigate the overhead introduced by the Subpage Tracker in Blowfish, we select two applications and evaluate the performance impact of enabling or disabling this configuration. As shown in Figure 9(b), the Subpage Tracker has a modest impact on application performance (3.1% for Cassandra and 0.9% for TriangleCount). This modest overhead is mainly due to that it splits only the page table mappings, rather than the actual physical pages, at a modest rate, avoiding the high cost of updating the `struct page` metadata.

Automatic Reclamation. Figure 10 illustrates how Blowfish and the baseline systems dynamically resize VM memory. To balance the performance and accuracy of sampling, we sample the memory footprint of guest every 0.5s at the PSI threshold to guarantee the slowdown of 5% on Blowfish. For GAPBS TriangleCount, we split the input graph into three parts to construct the evolving graph [17]. The algorithm involves three passes over the entire graph structure. For each pass, cold pages are gradually reclaimed. In comparison to HyperAlloc-4K-G and HyperAlloc-H, Blowfish can promptly reclaim memory regions that are unlikely to be accessed in the near future with higher reclamation throughput. HyperAlloc-G, on the other hand, is hindered by memory fragmentation and can reclaim only a limited amount of memory shortly after each pass. For KMeans, there are multiple distinct stages. Upon JVM startup, the heap is initialized with a relatively low memory footprint. As the application proceeds to initialize memory and load data, memory consumption gradually increases. During the subsequent phases, memory usage exhibits cyclical fluctuations. Blowfish maintains high performance while offering flexibility in memory usage across various applications. However, the baselines miss the opportunity to harvest the cold memory due to their poor elasticity. For 602.gcc_s, Blowfish is capable of aggressively reclaiming memory while swiftly responding to sudden bursts of memory

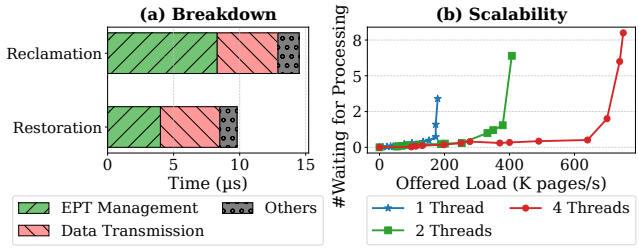


Figure 11: (a) Latency breakdown for reclaiming and restoring. (b) The scalability of the reclamation backend.

requests, achieving a sweet spot between performance and reclamation ratio.

Latency Breakdown. We use a microbenchmark to measure the single-page reclaim and restore latency of Blowfish. We run a program with random memory access in the VM and trigger Blowfish to reclaim pages individually, thereby avoiding batching optimizations. Figure 11(a) presents the results. Blowfish eliminates the overhead of guest page table and I/O page table modifications, making EPT handling the dominant software overhead, accounting for 41%-57% of the total. By removing redundant overheads of page table modifications, Blowfish reduces the single-page reclaim and restore latencies to 14.5µs and 9.8µs, which are 53% and 60% lower than HyperAlloc-4K-G, respectively. The latency could be further reduced with lower-latency hardware.

Scalability of the Reclamation Backend. We evaluate the scalability of the reclamation backend. "#Waiting for Processing" denotes the average number of pages waiting to be reclaimed. We assess scalability by running a microbenchmark with a large cold memory footprint in the VM and varying the reclamation request rate (the offered load). As shown in Figure 11(b), with a single reclamation thread, the number of pages queuing quickly reaches an inflection point as the load increases. At a load of about 170K pages/s, the queue length surges, indicating that a single thread is nearing its processing capacity. Increasing the number of threads alleviates this bottleneck, enabling the reclamation backend to handle higher loads. For example, using 2 threads and 4 threads increases the processing capacity of the backend by nearly 2.0× and 3.9×.

6 Limitations

The components of Blowfish exhibit different portability characteristics. The host-level reclamation backend operates entirely in the host and has no guest kernel dependency. The subpage tracker relies only on access bits and page table operations, making it applicable to any OS with standard page table support. The Fair MGLRU component, however, depends on MGLRU's multi-generational structure. On systems with classical two-list LRU implementations, the same principle can still be approximated by gating huge-page promotion with an additional access-count condition, requiring multiple observed accesses before promoting a 2 MB page to the active list. While this approximation may reduce effectiveness due to coarser promotion granularity and additional gating overhead, the core idea of correcting unfair hotness still applies.

7 Prior Work

Memtis [35] and Demeter [26] are the two pieces of prior work most closely related to Blowfish. Although these systems focus on tiered memory management, Blowfish shares several design considerations with them while introducing new ideas to tackle the related but distinct problem of VM memory overcommitment. This section discusses the similarities and differences between Blowfish and these prior systems.

7.1 Memtis

Both Blowfish and Memtis adopt THP-aware hotness tracking, but in different problem settings. This similarity stems from a shared observation: THPs introduce hot bloat issues, where a few hot subpages can cause the entire THP to appear hot, leaving many cold subpages unreclaimable (for Blowfish) or unevicted (for Memtis). Both systems, therefore, need to identify such THPs and split them to mitigate this issue.

However, the two systems differ substantially in both deployment environment and problem setting. Memtis is designed for bare-metal tiered memory management, while Blowfish targets VM memory overcommitment in virtualized systems. These differences lead to distinct designs. Memtis relies on PEBS-based sampling for page access tracking, which introduces an accuracy-overhead tradeoff and requires non-trivial deployment effort [26, 40]. Blowfish instead uses A/D-bit-based tracking, which is more virtualization-friendly and provides more deterministic access signals. Moreover, Memtis assumes a relatively stable fast tier for computing hot-set/hit-ratio statistics, an assumption that becomes difficult to maintain under dynamic memory overcommitment.

7.2 Demeter

Both Blowfish and Demeter adopt a decoupled architecture and perform page tracking within the guest. This similarity stems from a shared observation in virtualized environments: performing page tracking at the host incurs substantial system overhead.

However, the two systems differ in how to partition page table management functionality between the guest and the host. Demeter delegates the full tiered memory management pipeline, including data migration, to the guest, introducing overheads from GPT and IOPT updates under memory overcommitment. In contrast, Blowfish decouples page tracking and page management: the guest is responsible only for page

tracking, while the host handles data transfer and page table updates, reducing the latency of individual operations by 53-60%. In addition, Demeter operates on coarse-grained memory regions (*e.g.*, 2 MB), making it difficult to handle hot bloat and unfair hotness under mixed-granularity scenarios. It also lacks page-type awareness, reducing reclamation effectiveness. Blowfish instead uses subpage-level tracking and page-type awareness for more effective reclamation.

7.3 Takeaways

Blowfish, Memtis, and Demeter address different problems (VM memory overcommitment vs. tiered memory management) in different deployment contexts (virtualized vs. bare-metal), yet converge on similar techniques due to shared requirements for effective hot-bloat tracking and low-overhead page access tracking. However, different goals lead to distinct designs, as discussed earlier. These comparisons show that efficient VM memory overcommitment requires rethinking both tracking mechanisms and system design for responsive, low-overhead memory management.

8 Conclusion

In this paper, we have proposed Blowfish, a memory overcommitment framework built on disaggregated memory. Blowfish can perform cold/free memory reclamation and restoration at a µs-scale, reaching up to 2.48× and 2.14× faster than the state-of-the-art work, *i.e.*, HyperAlloc.

Acknowledgments

We thank the anonymous reviewers for their comments and are particularly grateful to our shepherd for valuable feedback. This work was supported in part by the National Key Research and Development Program of China (No. 2024YFE0204100), the Pioneer Initiative Talent Program B of Chinese Academy of Sciences (No. E545030, E401430), Fundamental and Interdisciplinary Disciplines Breakthrough Plan of the Ministry of Education of China (No. JYB2025XDXM118), National Natural Science Foundation of China under Grant No. U25A6023, 92464301, 62302479, U23B2020. Tao Xie is also affiliated with the Key Laboratory of High Confidence Software Technologies (Peking University), Ministry of Education, China; Beijing Tongming Lake Information Technology Application Innovation Center, China; Institute of Systems for Advanced Computing at Fudan University, China; Shanghai Research Institute of Systems of Open Computing, China.

References

[1] *White paper: Introduction to InfiniBand*. Mellanox Technologies Inc., 2003.

[2] LIBSVM data. https://www.csie.ntu.edu.tw/~cjlin/libsvmtools/datasets, 2012.

[3] Memcached - a distributed memory object caching system. http://memcached.org, 2020.

[4] KONECT networks data. http://konect.cc/networks/, 2021.

[5] QEMU. https://www.qemu.org/2024/04/23/qemu-9-0-0/, 2024.

[6] Transparent Hugepage Support. https://docs.kernel.org/admin-guide/mm/transhuge.html, 2025.

[7] Alibaba. Alibaba Cluster Trace Program. https://github.com/alibaba/clusterdata, 2018.

[8] Emmanuel Amaro, Christopher Branner-Augmon, Zhihong Luo, Amy Ousterhout, Marcos K. Aguilera, Aurojit Panda, Sylvia Ratnasamy, and Scott Shenker. Can far memory improve job throughput? In *Proceedings of the 15th European Conference on Computer Systems*, pages 14:1–14:16, 2020.

[9] Nadav Amit, Dan Tsafrir, and Assaf Schuster. VSwapper: A memory swapper for virtualized environments. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 349–366, 2014.

[10] Nadav Amit and Michael Wei. The design and implementation of hyperupcalls. In *Proceedings of the 2018 USENIX Annual Technical Conference*, pages 97–112, 2018.

[11] Apache. Apache Cassandra. https://cassandra.apache.org, 2021.

[12] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles*, pages 164–177, 2003.

[13] Scott Beamer, Krste Asanovic, and David A. Patterson. The GAP benchmark suite. *CoRR*, abs/1508.03619, 2015.

[14] Benjamin Berg, Daniel S. Berger, Sara McAllister, Isaac Grosof, Sathya Gunasekar, Jimmy Lu, Michael Uhlar, Jim Carrig, Nathan Beckmann, Mor Harchol-Balter, and Gregory R. Ganger. The CacheLib caching engine: Design and experiences at scale. In *Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation*, pages 753–768, 2020.

[15] James Bucek, Klaus-Dieter Lange, and Jóakim v. Kistowski. SPEC CPU2017: Next-generation compute benchmark. In *Companion of the 2018 ACM/SPEC International Conference on Performance Engineering*, pages 41–42, 2018.

[16] Irina Calciu, M. Talha Imran, Ivan Puddu, Sanidhya Kashyap, Hasan Al Maruf, Onur Mutlu, and Aasheesh Kolli. Rethinking software runtimes for disaggregated memory. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 79–92, 2021.

[17] Lei Chen, Shi Liu, Chenxi Wang, Haoran Ma, Yifan Qiao, Zhe Wang, Chenggang Wu, Youyou Lu, Xiaobing Feng, Huimin Cui, Shan Lu, and Harry Xu. A tale of two paths: Toward a hybrid data plane for efficient far-memory applications. In *Proceedings of the 18th USENIX Symposium on Operating Systems Design and Implementation*, pages 77–95, 2024.

[18] Intel Corporation. Page modification logging for virtual machine monitor white paper. https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html, 2022.

[19] Padmapriya Duraisamy, Wei Xu, Scott Hare, Ravi Rajwar, David Culler, Zhiyi Xu, Jianing Fan, Christopher Kennelly, Bill McCloskey, Danijela Mijailovic, Brian Morris, Chiranjit Mukherjee, Jingliang Ren, Greg Thelen, Paul Turner, Carlos Villavieja, Parthasarathy Ranganathan, and Amin Vahdat. Towards an adaptable systems architecture for memory tiering at warehouse-scale. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 727–741, 2023.

[20] Rong-En Fan, Kai-Wei Chang, Cho-Jui Hsieh, Xiang-Rui Wang, and Chih-Jen Lin. Liblinear: A library for large linear classification. *Journal of Machine Learning Research*, 9(61):1871–1874, 2008.

[21] Google. Google production cluster trace data. https://github.com/google/cluster-data, 2019.

[22] Donghyun Gouk, Sangwon Lee, Miryeong Kwon, and Myoungsoo Jung. Direct access, High-Performance memory disaggregation with DirectCXL. In *Proceedings of the 2022 USENIX Annual Technical Conference*, pages 287–294, 2022.

[23] Juncheng Gu, Youngmoon Lee, Yiwen Zhang, Mosharaf Chowdhury, and Kang G. Shin. Efficient memory disaggregation with Infiniswap. In *Proceedings of the 14th USENIX Symposium on Networked Systems Design and Implementation*, pages 649–667, 2017.

[24] David Hildenbrand and Martin Schulz. virtio-mem: paravirtualized memory hot(un)plug. In *Proceedings of the 17th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, pages 1–14, 2021.

[25] Jingyuan Hu, Xiaokuang Bai, Sai Sha, Yingwei Luo, Xiaolin Wang, and Zhenlin Wang. HUB: hugepage ballooning in kernel-based virtual machines. In *Proceedings of International Symposium on Memory Systems*, pages 31–37, 2018.

[26] Junliang Hu, Zhisheng Hu, Chun-Feng Wu, and Ming-Chang Yang. Demeter: A scalable and elastic tiered memory solution for virtualized cloud via guest delegation. In *Proceedings of the 31st ACM SIGOPS Symposium on Operating Systems Principles*, pages 169–185, 2025.

[27] Woomin Hwang, Yangwoo Roh, Youngwoo Park, Ki-Woong Park, and Kyu Ho Park. Hyperdealer: Reference-pattern-aware instant memory balancing for consolidated virtual machines. In *Proceedings of the 3rd IEEE International Conference on Cloud Computing*, pages 426–434, 2010.

[28] Intel Corporation. *Intel 64 and IA-32 Architectures Software Developer's Manual Volume 3C: System Programming Guide, Part 3*, 2025.

[29] Weiwei Jia, Jiyuan Zhang, Jianchen Shan, and Xiaoning Ding. Making dynamic page coalescing effective on virtualized clouds. In *Proceedings of the 18th European Conference on Computer Systems*, pages 298–313, 2023.

[30] Hwanju Kim, Heeseung Jo, and Joonwon Lee. Xhive: Efficient cooperative caching for virtual machines. *IEEE Transactions on Computers*, 60(1):106–119, 2011.

[31] Avi Kivity, Yaniv Kamay, Dor Laor, Uri Lublin, and Anthony Liguori. KVM: The Linux virtual machine monitor. In *Proceedings of the 2007 Linux Symposium, Volume 1*, pages 225–230, 2007.

[32] Haewoon Kwak, Changhyun Lee, Hosung Park, and Sue Moon. What is Twitter, a social network or a news media? In *Proceedings of the 19th International Conference on World Wide Web*, pages 591–600, 2010.

[33] Youngjin Kwon, Hangchen Yu, Simon Peter, Christopher J. Rossbach, and Emmett Witchel. Coordinated and efficient huge page management with ingens. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation*, pages 705–721, 2016.

[34] Aapo Kyrola, Guy Blelloch, and Carlos Guestrin. GraphChi: Large-Scale graph computation on just a PC. In *Proceedings of the 10th USENIX Symposium on*

Operating Systems Design and Implementation, pages 31–46, 2012.

[35] Taehyung Lee, Sumit Kumar Monga, Changwoo Min, and Young Ik Eom. Memtis: Efficient memory tiering with dynamic page classification and page size determination. In *Proceedings of the 29th ACM SIGOPS Symposium on Operating Systems Principles*, pages 17–34, 2023.

[36] Chuandong Li, Sai Sha, Yangqing Zeng, Xiran Yang, Yingwei Luo, Xiaolin Wang, Zhenlin Wang, and Diyu Zhou. Taming hot bloat under virtualization with HUGESCOPE. In *Proceedings of the 2024 USENIX Annual Technical Conference*, pages 999–1012, 2024.

[37] Huaicheng Li, Daniel S. Berger, Lisa Hsu, Daniel Ernst, Pantea Zardoshti, Stanko Novakovic, Monish Shah, Samir Rajadnya, Scott Lee, Ishwar Agarwal, Mark D. Hill, Marcus Fontoura, and Ricardo Bianchini. Pond: CXL-based memory pooling systems for cloud platforms. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 574–587, 2023.

[38] Chengzhi Lu, Kejiang Ye, Guoyao Xu, Cheng-Zhong Xu, and Tongxin Bai. Imbalance in the cloud: An analysis on Alibaba cluster trace. In *Proceedings of the 2017 IEEE International Conference on Big Data*, pages 2884–2892, 2017.

[39] Dan Magenheimer, Chris Mason, Dave McCracken, and Kurt Hackel. Transcendent memory and Linux. In *Proceedings of the 2009 Linux Symposium*, pages 191–200, 2009.

[40] Hasan Al Maruf, Hao Wang, Abhishek Dhanotia, Johannes Weiner, Niket Agarwal, Pallab Bhattacharya, Chris Petersen, Mosharaf Chowdhury, Shobhit Kanaujia, and Prakash Chauhan. TPP: Transparent page placement for CXL-enabled tiered-memory. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, pages 742–755, 2023.

[41] Juan Navarro, Sitararn Iyer, Peter Druschel, and Alan Cox. Practical, transparent operating system support for superpages. *SIGOPS Oper. Syst. Rev.*, 36(SI):89–104, 2003.

[42] NVIDIA. The ConnectX InfiniBand In-Network Computing Adapter. https://www.nvidia.com/en-us/networking/infiniband-adapters, 2022.

[43] NVIDIA. NVIDIA Ethernet SuperNICs. https://www.nvidia.com/en-us/networking/ethernet-adapters, 2024.

[44] NVIDIA. NVIDIA ConnectX-8 SuperNIC. https://www.nvidia.com/en-us/networking/products/ethernet/supernic, 2025.

[45] Amy Ousterhout, Joshua Fried, Jonathan Behrens, Adam Belay, and Hari Balakrishnan. Shenango: Achieving high CPU efficiency for latency-sensitive datacenter workloads. In *Proceedings of the 16th USENIX Symposium on Networked Systems Design and Implementation*, pages 361–378, 2019.

[46] Alexander Outman. Web data commons - hyperlink graphs, 2017.

[47] SeongJae Park, Madhuparna Bhowmik, and Alexandru Uta. DAOS: Data access-aware operating system. In *Proceedings of the 31st International Symposium on High-Performance Parallel and Distributed Computing*, pages 4–15, 2022.

[48] SeongJae Park, Yunjae Lee, and Heon Y. Yeom. Profiling dynamic data access patterns with controlled overhead and quality. In *Proceedings of the 20th International Middleware Conference Industrial Track*, pages 1–7, 2019.

[49] Yifan Qiao, Chenxi Wang, Zhenyuan Ruan, Adam Belay, Qingda Lu, Yiying Zhang, Miryung Kim, and Guoqing Harry Xu. Hermit: Low-latency, high-throughput, and transparent remote memory via feedback-directed asynchrony. In *Proceedings of the 20th USENIX Symposium on Networked Systems Design and Implementation*, pages 181–198, 2023.

[50] Charles Reiss, Alexey Tumanov, Gregory R. Ganger, Randy H. Katz, and Michael A. Kozuch. Heterogeneity and dynamicity of clouds at scale: Google trace analysis. In *Proceedings of the 3rd ACM Symposium on Cloud Computing*, pages 7:1–7:13, 2012.

[51] Zhenyuan Ruan, Malte Schwarzkopf, Marcos K. Aguilera, and Adam Belay. AIFM: High-performance, application-integrated far memory. In *Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation*, pages 315–332, 2020.

[52] Sai Sha, Chuandong Li, Yingwei Luo, Xiaolin Wang, and Zhenlin Wang. vTMM: Tiered memory management for virtual machines. In *Proceedings of the 18th European Conference on Computer Systems*, pages 283–297, 2023.

[53] Carl A. Waldspurger. Memory resource management in VMware ESX server. *SIGOPS Oper. Syst. Rev.*, 36(SI):181–194, 2003.

[54] Chenxi Wang, Yifan Qiao, Haoran Ma, Shi Liu, Yiying Zhang, Wenguang Chen, Ravi Netravali, Miryung Kim, and Guoqing Harry Xu. Canvas: Isolated and adaptive swapping for multi-applications on remote memory. In *Proceedings of the 20th USENIX Symposium on Networked Systems Design and Implementation*, pages 161–179, 2023.

[55] Jiawei Wang, Diogo Behrens, Ming Fu, Lilith Oberhauser, Jonas Oberhauser, Jitang Lei, Geng Chen, Hermann Härtig, and Haibo Chen. BBQ: A block-based bounded queue for exchanging data and profiling. In *Proceedings of the 2022 USENIX Annual Technical Conference*, pages 249–262, 2022.

[56] Yaohui Wang, Ben Luo, and Yibin Shen. Efficient memory overcommitment for I/O passthrough enabled VMs via fine-grained page meta-data management. In *Proceedings of the 2023 USENIX Annual Technical Conference*, pages 769–783, 2023.

[57] Wang Wei. Provide support for free page reporting. https://lwn.net/Articles/808807/, 2020.

[58] Johannes Weiner, Niket Agarwal, Dan Schatzberg, Leon Yang, Hao Wang, Blaise Sanouillet, Bikash Sharma, Tejun Heo, Mayank Jain, Chunqiang Tang, and Dimitrios Skarlatos. TMO: Transparent memory offloading in datacenters. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 609–621, 2022.

[59] Lars Wrenger, Kenny Albes, Marco Wurps, Christian Dietrich, and Daniel Lohmann. HyperAlloc: Efficient VM memory de/inflation via hypervisor-shared page-frame allocators. In *Proceedings of the 20th European Conference on Computer Systems*, pages 702–719, 2025.

[60] Lars Wrenger, Florian Rommel, Alexander Halbuer, Christian Dietrich, and Daniel Lohmann. LLFree: Scalable and optionally-persistent page-frame allocation. In *Proceedings of the 2023 USENIX Annual Technical Conference*, pages 897–914, 2023.

[61] Wonsup Yoon, Jisu Ok, Jinyoung Oh, Sue Moon, and Youngjin Kwon. DiLOS: Do not trade compatibility for performance in memory disaggregation. In *Proceedings of the 18th European Conference on Computer Systems*, pages 266–282, 2023.

[62] Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica. Spark: Cluster computing with working sets. In *Proceedings of the 2nd USENIX Conference on Hot Topics in Cloud Computing*, pages 10:1–10:7, 2010.

[63] Yuhong Zhong, Daniel S. Berger, Carl Waldspurger, Ryan Wee, Ishwar Agarwal, Rajat Agarwal, Frank Hady, Karthik Kumar, Mark D. Hill, Mosharaf Chowdhury, and Asaf Cidon. Managing memory tiers with CXL in virtualized environments. In *Proceedings of the 18th USENIX Symposium on Operating Systems Design and Implementation*, pages 37–56, 2024.

Espresso: Constructing Cost-Efficient CXL JBOF via Inter-SSD Computing Resource Sharing

Shushu Yi[1] Yuda An[1] Li Peng[1] Xiurui Pan[1] Qiao Li[2] Jieming Yin[3] Guangyan Zhang[4] Wenfei Wu[1]
Chenxi Wang[5] Diyu Zhou[1] Zhenlin Wang[6] Xiaolin Wang[1] Yingwei Luo[1] Ke Zhou[7] Jie Zhang[1]

Computer Hardware and System Evolution Laboratory (ChaseLab)
Beijing Key Laboratory of Software and Hardware Cooperative Artificial Intelligence Systems
Peking University[1], Mohamed bin Zayed University of Artificial Intelligence[2]
Nanjing University of Posts and Telecommunications[3], Tsinghua University[4]
University of Chinese Academy of Sciences[5], Michigan Tech[6], Huazhong University of Science and Technology[7]

Abstract

Enterprise SSDs integrate substantial computing resources (e.g., ARM processor and onboard DRAM) to handle I/O bursts. However, these resources significantly raise SSD monetary cost and suffer severely underutilized in JBOF deployments due to the sporadic nature of I/O bursts. In this paper, we present *Espresso*, a cost-efficient JBOF design that provisions only moderate computing resources per SSD at low monetary cost, while delivering demanded I/O performance through efficient inter-SSD resource sharing. Specifically, Espresso first disaggregates SSD architecture into functionally distinct components, enabling fine-grained SSD internal resource management. Espresso then employs a decentralized scheme to manage these disaggregated resources and harvests the computing resources of idle SSDs to assist busy SSDs in handling I/O bursts. This idea is facilitated by the cache-coherent CXL fabric, with which the busy SSDs can directly utilize the harvested computing resources to accelerate metadata processing. The evaluation results show that Espresso can improve SSD resource utilization by 50.4% and reduce monetary cost by 19.0% with negligible performance degradation, compared to the state-of-the-art JBOF designs.

1 Introduction

I/O-intensive scenarios, such as cloud storage, large language model inference, and burst cache [46, 61, 69, 70, 97, 118], eagerly demand extremely high I/O throughput to accelerate the ever-expanding dataset access. Following this trend, solid-state drives (SSDs) have become one of the most indispensable storage media and have experienced continuous technical advancements in both *scale-up* and *scale-out* ways.

From the *scale-up* aspect, SSD manufacturers integrate more hardware resources within each SSD device to enhance I/O parallelism. For example, the emerging PCIe 5.0 SSDs [85] boost the performance of embedded ARM processors by 1.7× over the PCIe 4.0 ones [84] to accelerate the execution of SSD firmware. Moreover, SSDs typically equip large onboard DRAM (1 GB per TB flash [90]) to accommodate the entire metadata (mainly FTL mapping tables [16]) for fast access. From the *scale-out* aspect, SSD suppliers cluster tens of high-performance SSDs as *just a bunch of flash* (JBOF) [34, 40, 75, 108, 110], which aggregates the hardware resources from every SSD to deliver extremely high throughput.

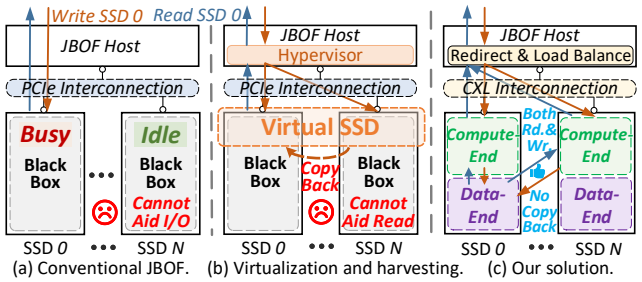


Figure 1: Comparison of different JBOF designs.

Unfortunately, these technical trends place SSD consumers in a *cost-utilization dilemma*. To be more precise, while the increasing hardware resources elevate the *bill of material (BOM)* costs of SSDs to satisfy the performance requirement of burst I/O, the sporadic nature of I/O bursts causes severe SSD resource underutilization in JBOF level (cf. Figure 1a). For instance, in 94.6% uptime of a Tencent storage server [134] equipped with 25 drives, at least 20 drives are underutilized (i.e., bandwidth utilization is lower than 75%, cf. § 2.2). This is because in modern cloud platforms [66, 86], storage drives (e.g., SSDs) are commonly allocated (or sold) to different tenants. These tenants utilize their own drives to serve different application instances with diverse I/O patterns, which experience I/O bursts at different times.

A straightforward idea to improve utilization is storage virtualization and harvesting [5, 86, 107, 119, 126]. As shown in Figure 1b, the hypervisor [86, 107] can harvest idle SSDs by dynamically grouping a busy SSD with multiple idle ones as a *virtual SSD*. Subsequently, parts of write requests originally targeting the busy SSD are redirected to the idle ones via the virtual SSD abstraction, leading to a higher burst performance and SSD utilization. Once the burst period concludes, these

idle SSDs will be reclaimed and set aside for future harvesting. While this approach succeeds in exploiting idle SSDs, it unfortunately faces three prominent challenges:

• *Coarse-grained SSD harvesting causes resource stranding issues:* Different I/O tasks impose varied degrees of burden on the computing (e.g., ARM processor and DRAM) and flash (e.g., flash channels) resources within SSDs. For example, 4 KB sequential reads consume 96% of processor clocks while only exploiting 39% of flash times. 4 KB sequential writes, in contrast, are flash-hungry (99%) while keeping the processor underutilized (27%, cf. § 3.1 for details). However, the hypervisor treats SSDs as monolithic black boxes, which leads to *resource stranding* issues [5, 58, 119]. For instance, when an SSD is flash-hectic for write bursts, its computing resources may still be idle. These computing resources cannot be harvested as the entire SSD is considered busy.

• *Limited profit in read-dominated workloads:* The simple virtualization and harvesting approach yields minor benefits for read-dominated workloads. This is because, without storing the target data of incoming read requests beforehand, the temporarily harvested SSDs cannot aid the busy SSDs in read services. Our evaluation shows that this simple approach only brings 0.5% throughput improvement in read-dominated workloads (cf. § 3.1 for details).

• *High overheads brought by written data copyback and resource management:* Although redirecting write requests to the harvested SSDs can improve SSD utilization, this benefit comes at high costs. To be specific, when reclaiming the harvested SSDs, the hypervisor has to copy back the written data to the initial destination SSD for availability [86]. Such write amplification drastically shortens SSD lifetimes (22.5% reduction). Moreover, the centralized virtual SSD management in the hypervisor imposes huge burdens [52, 86] on the host CPU (e.g., 16 ARM cores in SuperMicro SSG-229J [110]), compelling it to become the performance bottleneck of JBOF (21.4% throughput loss, cf. § 3.1).

In response to these challenges, we introduce *Espresso*[1], a cost-efficient JBOF design, which tackles the cost-utilization dilemma by only reserving moderate computing resources in SSDs while achieving satisfactory burst I/O performance through inter-SSD resource sharing (cf. Figure 1c). Our key insight is that the high-speed and cache-coherent *Compute eXpress Link (CXL)* interconnections [19, 58, 109] can be harnessed to facilitate fine-grained, high-profit, and low-overhead SSD harvesting. Specifically, recognizing the black-box limitation of conventional SSDs, Espresso first disaggregates the SSD architecture into two parts, *compute-end* and *data-end*, based on their functionality. Compute-end encloses computing resources, such as ARM processor and onboard DRAM, which are responsible for executing firmware tasks (e.g., address translation [16, 35]). Data-end comprises flash resources (e.g., flash channels) and data-related components (e.g., DMA engine), which are in charge of data transfer and flash I/O. Espresso exposes them separately to the host and peer SSDs via the high-speed CXL interconnection. This design enables fine-grained resource management and promises to mitigate the resource stranding issues.

With the disaggregated SSD architecture, Espresso can benefit both read and write requests by harvesting idle compute-end to alleviate burdens of the SSDs, which are busy with firmware tasks. This idea is facilitated by the cache-coherent feature of CXL, with which the harvested compute-end can precisely operate the essential metadata of busy SSDs (e.g., FTL mapping table [16]) for I/O request handling. Moreover, this design avoids detrimental copyback because it harvests only the stateless computing resource to expedite metadata processing while keeping the data path on the stateful flash memory unchanged. In addition, Espresso employs a holistic I/O redirection design and a miss-ratio-curve-based scheme to deliver reasonable performance isolation for harvested SSDs. Considering the overhead of centralized resource management, Espresso first leverages the global coherent memory constructed by CXL to enable inter-SSD communication. Espresso then implements decentralized and self-governing resource management in SSDs to relieve host CPU burdens. Comprehensive evaluation results demonstrate that Espresso outperforms existing JBOF designs, improving SSD resource utilization by 50.4%, saving 19.0% BOM costs while having negligible performance loss.

Our main **contributions** can be summarized as follows:

- We deeply review the cost-utilization dilemma of JBOF and reveal that the black-box constraint of conventional SSD designs can lead to severe resource stranding issues.
- We propose a novel SSD architecture that disaggregates SSD internal resources into compute-end and data-end based on their functionality. This design lays the foundation for fine-grained and efficient SSD resource harvesting.
- We propose a cost-efficient JBOF design that reserves moderate computing resources in each SSD at low BOM costs while achieving demanded I/O performance by leveraging CXL to facilitate inter-SSD resource sharing.

2 Background and Motivation

2.1 JBOF and NVMe SSD

JBOF architecture. Just a bunch of flash (JBOF) is a type of storage server that can incorporate multiple high-performance SSDs as a whole, thereby satisfying the ever-increasing performance demands in a scale-out manner. As shown in Figure 2a, a JBOF comprises CPU, DRAM, NIC, and tens of SSDs, connected via PCIe interconnections. For high cost and energy efficiency [34, 75, 108], recent advanced JBOF products typically equip Data Processing Units (DPUs) to replace separated CPU, DRAM, and NIC, serving as the *host*. For example,

[1] Espresso is accessible at https://github.com/ChaseLab-PKU/Espresso.

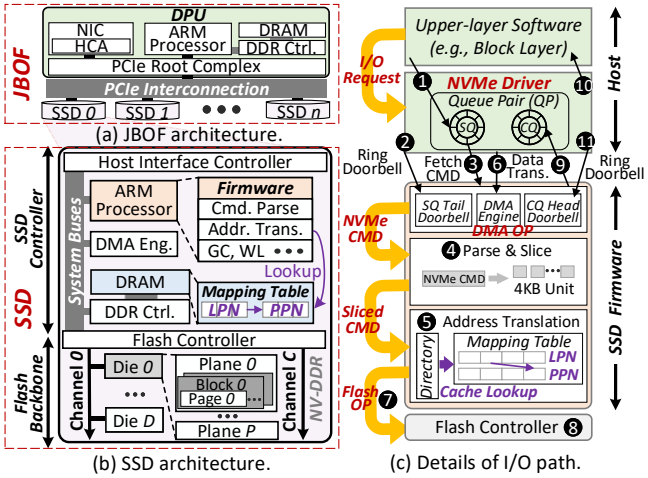


Figure 2: Details of the traditional JBOF system.

SuperMicro SSG-229J-5BU24JBF [110], one of the up-to-date JBOF products, supports 24 SSDs with two BlueField-3 DPUs [78], each containing a 16-core ARM processor and 16 GB DRAM. In the following, we mainly focus on DPU-based JBOF to enjoy its superior cost and energy efficiency, which aligns with the target of this paper. Nevertheless, our designs are also applicable to conventional JBOF products.

SSD architecture. Figure 2b presents a typical architecture of modern SSDs [91]. The SSD controller connects to the host through PCIe lanes and a host interface controller. Moreover, it integrates an ARM processor, a DDR DRAM controller, and some specialized processing elements (e.g., DMA engine). The ARM processor is mainly responsible for performing firmware tasks (e.g., command parsing, address translation, and garbage collection). These components are coupled with a flash backbone through the flash controller. The flash backbone consists of 8 to 16 flash channels, each enclosing several flash dies. A flash die can be further divided into multiple flash planes, blocks, and pages.

I/O path. Figure 2c illustrates the I/O path in JBOF systems. When I/O requests arrive, the NVMe driver in the host submits NVMe commands to submission queues (SQ, ❶). It then notifies the SSD of the command arrivals by ringing the SQ doorbells corresponding to the queues (❷). Afterward, the SSD firmware operates the host interface controller to fetch NVMe commands from SQ (❸). SSD firmware then parses these commands, slices them into unit size (e.g., 4 KB, ❹), and translates the host *logical address (LPN)* to the flash *physical address (PPN)* by referring to the *mapping table* in *flash translation layer (FTL,* ❺*)*. The mapping table is persisted in the flash backbone for crash consistency and cached in onboard DRAM for high performance. A *mapping directory* is used to locate cached mapping table entries. SSD firmware also needs to orchestrate the host-SSD data transfers by issuing host *DMA operations* to the DMA engine (❻). Subsequently, the SSD firmware sends *flash operations* to the flash controller (❼), which performs flash I/O following the Open NAND Flash interface (ONFi) protocol [82] (❽). In the backward, the firmware writes the results to the completion queues (CQ) and then notifies the host by generating MSI-X interrupts [26] (❾). Finally, the NVMe driver reports request completions to upper-layer software (❿) and acknowledges interrupts by updating the CQ doorbell (⓫).

2.2 Motivation: Cost-Utilization Dilemma

High cost of SSD. An obvious trend in SSD advancement is that SSD manufacturers tend to integrate numerous computing resources within SSDs to conduct firmware tasks rapidly, thereby boosting I/O performance. Figure 3a presents the computing power of embedded processors in varied SSD controllers on Dhrystone v2.1 benchmark [120]. The results show that the computing power of embedded processors has increased exponentially over the last decade. Moreover, enterprise SSDs [73, 91] typically demand 1 GB onboard DRAM per TB flash capacity to accommodate their entire metadata (mainly the FTL mapping table) for fast access. These abundant computing resources cause high *bill of material (BOM)* costs of SSDs. For instance, the computing resources (i.e., SSD controllers and DRAM) account for 23.2% and 31.8% of BOM costs to manufacture 4 TB PCIe 4.0 SSDs and PCIe 5.0 SSDs [27, 67, 113, 114] (cf. `Compute` in Figure 3b).

Low utilization of JBOF. Contrary to the continually increasing burst performance demands and BOM costs, SSD utilization in JBOF remains low due to the sporadic nature of I/O bursts. Quantitative analysis reveals that in any uptime of a Tencent storage server equipped with 25 drives [134], the probability of at least 20 drives being underutilized (i.e., under 75% bandwidth utilization) is 94.6% (cf. Figure 3c). This is because in modern cloud platforms, especially the Infrastructure-as-a-Service (IaaS) cloud [66, 86], storage drives are commonly allocated (or sold) to different tenants. These tenants utilize their own drives to serve varied applications with diverse I/O patterns, which experience I/O bursts at different times [99, 107]. This phenomenon also exists in other storage servers over diverse storage service providers. As depicted in Figure 3d, the average drive bandwidth utilization is only 8.0%, 27.8%, and 15.3% in Alibaba [2], Tencent [134], and Fujitsu [56] clusters, respectively.

3 Preliminary Study

3.1 Simple Solution and Challenge

The cost-utilization dilemma inspires future SSD and JBOF designs to be cost-efficient. Ideally, future SSDs may only reserve moderate hardware resources at low BOM costs while achieving required I/O performance by harvesting underutilized resources within the same JBOF.

JBOF consisted of host-managed SSDs. *Host-managed SSD (HMSSD)*, such as open-channel SSD (OCSSD) [12,

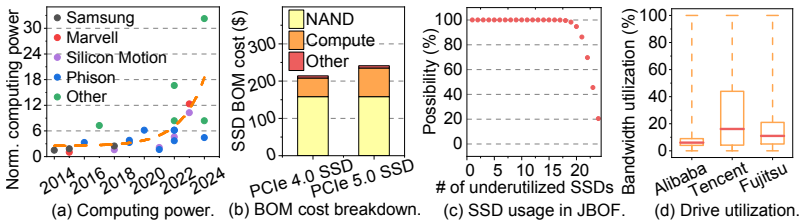


Figure 3: Analysis of the cost-utilization dilemma.

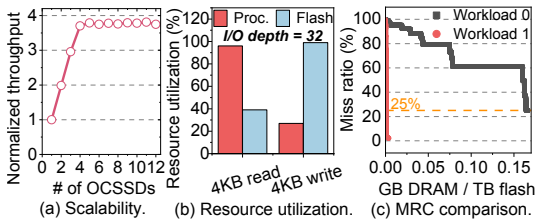


Figure 4: Preliminary study.

86, 87, 107], zoned-namespace (ZNS) SSD [11], and DirectFlash [105], is a representative SSD architecture that retains minimum hardware (e.g., flash controller and flash backbone) in SSD while relying on host computing resources and user-specified driver to conduct firmware tasks and cache metadata. Although HMSSD is superior in low cost, it, unfortunately, hampers scalability and compatibility. To be specific, the tens of HMSSDs in JBOF cause substantial computing overhead, compelling the centralized and limited host resources to become the performance bottleneck. Here we take OCSSD-based JBOF as an example, for its mature software support (i.e., Linux LightNVM [63]). Figure 4a shows the aggregated throughput of a 16-core-DPU-enabled JBOF with varying numbers of OCSSDs (cf. § 5.1 for experimental setups). The performance of the JBOF gets saturated with only 4 SSDs, because of the limited host resources. Note that, even for other JBOF products equipping costly x86 CPUs, the average resource for each OCSSD is also restricted (e.g., two 24-core CPUs for 97 SSDs in Dell PowerStore 500T [22]). In production environments, applications also compete for the essential host resources, further exaggerating the scalability crisis. Moreover, HMSSD-based JBOF faces severe compatibility issues as it requires huge manpower for OS and application adaptation to explicitly conduct firmware tasks, preventing it from wide deployment [4].

SSD virtualization and harvesting. A more scalable and compatible solution is SSD virtualization [7, 21, 75, 88, 126] and harvesting [5, 58, 86, 96, 107, 119]. Specifically, although SSDs in JBOF are logically allocated to different tenants for separate use, an extra storage virtualization layer in hypervisor [86, 107] can harvest the resources of idle SSDs by dynamically grouping busy SSDs (named *borrower*) and several idle SSDs (named *lenders*) as virtual SSDs. Thereafter, user write requests originally targeting the borrower can spread across both the borrower and lenders through the virtual SSD abstraction, thereby achieving higher burst write performance and SSD utilization. Note that the written data is temporarily stored in the storage space of the lenders. Once the burst period concludes, data should be copied back to the borrower to restore space ownership of the lenders [86].

Challenges. While this approach succeeds in harvesting idle resources, it unfortunately faces prominent challenges:

• *Coarse-grained SSD harvesting causes resource stranding issues* ***(Challenge 1)***. We analyze the pressure of different I/O tasks on SSD internal computing (i.e., ARM processor and DRAM) and flash (i.e., flash channels) resources, the results of which are shown in Figures 4b and 4c. 4 KB sequential reads on a real DaisyPlus SSD (cf. § 4.6 for details) consume 96% of the processor clocks while merely utilizing 39% of flash clocks (cf. Figure 4b). In comparison, 4 KB sequential writes are flash-intensive (99%), while leaving the processor underutilized (27%). Moreover, Figure 4c illustrates the *miss ratio curve (MRC)* [95, 117] of LRU-based metadata (i.e., FTL mapping table) cache in onboard DRAM. `Workload 1` [134] only consumes 0.001 GB DRAM (per TB flash) to achieve a 25% miss ratio, while that is 0.17 GB for `Workload 0`. In conclusion, different I/O tasks can impose varying levels of pressure on computing and flash resources within SSDs. Unfortunately, upper-layer software treats SSDs as monolithic black boxes, which leads to resource stranding issues. For example, when the flash backbone of an SSD is heavily engaged in 4 KB write bursts, its computing resources may remain idle. These idle resources cannot be lent to other SSDs in the simple approach because the entire SSD is considered busy.

• *Limited profit in read-dominated workloads* ***(Challenge 2)***. Our evaluation reveals that the simple virtualization and harvesting approach only brings 0.5% and 0.8% throughput improvements in Tencent [134] and Alibaba [2, 23] workloads, respectively, where read requests dominate the I/O (cf. § 5.2). This is because the target data of read requests is exclusively stored in the borrower's flash backbone. Lenders cannot assist the borrower in serving read requests due to the lack of the target data in the lenders' flash backbone.

• *High overheads brought by written data copyback* ***(Challenge 3.1)*** *and resource management* ***(Challenge 3.2)***. In our tests, the simple approach causes 0.29 more drive writes per day (DWDP) on Tencent traces [134]. This is because, when lenders are reclaimed, the hypervisor must copy the written data back to the borrower for availability [86]. These extra writes lead to 22.5% shorter SSD lifetime for enterprise SSDs [91] typically with 1 DWDP endurance. Moreover, the centralized virtual SSD management in hypervisor can introduce substantial software overhead [52, 86], which compels the host CPU to become the performance bottleneck and causes severe throughput loss (e.g., 21.4% in Tencent [134]).

3.2 Key Insight: Compute Express Link

CXL outline. *Compute eXpress Link (CXL)* [19] is an advanced interconnect standard designed to facilitate high-performance and cache-coherent communication among the

host and various peripheral devices (e.g., SSD) [42, 59, 131]. CXL comprises three sub-protocols: *CXL.io* undertakes PCIe backward-compatible operations; *CXL.cache* empowers cache coherence in CXL fabric; *CXL.mem* enables device memory to be accessed by the host and other devices via `load/store` instructions. For higher scalability, CXL 3.0 [20] introduces port-based routing and multi-level switching. These new features extend CXL fabric to rack-level and enable cache-coherent peer-to-peer communication among up to 4096 points (i.e., hosts or devices). In this work, we conform to CXL 3.0 standard and equip SSDs with all three sub-protocols (i.e., as CXL Type-2 devices [39]), enabling cache-coherent memory access within the entire JBOF.

Opportunities. Recognizing the black-box constraint of conventional SSDs, we can first disaggregate the hardware resources of SSDs into multiple disjoint parts and then expose these parts separately to the host and other SSDs through the high-speed CXL interconnection. This design facilitates fine-grained resource management, laying a foundation to tackle the resource stranding issues (solution of ***Challenge 1***). Additionally, with the support of cache-coherent memory access, the lender's processor can help the borrower handle I/O requests (e.g., command parsing and address translation) by directly operating the borrower's metadata stored in its onboard DRAM through CXL fabric. Moreover, the borrower can cache parts of its mapping table in the lender's DRAM for a lower miss ratio. This method benefits both reads and writes and avoids data copyback overhead (solution of ***Challenges 2 and 3.1***), as it only harvests the stateless computing resources (i.e., processor and DRAM) to accelerate metadata processing, without redirecting data. Finally, we can exploit the cache-coherent memory to facilitate efficient inter-SSD communication. Thereafter, a self-governing and decentralized SSD resource management scheme can be implemented to alleviate the CPU burden imposed by the centralized virtualization and harvesting (solution of ***Challenge 3.2***).

4 Design and Implementation

Inspired by the aforementioned preliminary analysis, we propose *Espresso*, a novel JBOF design that facilitates fine-grained, high-profit, and low-overhead inter-SSD resource sharing to tackle the cost-utilization dilemma.

4.1 Overview

Target scenarios. Espresso is designed for DPU-based JBOF (cf. § 2.1) to enjoy its extraordinary energy and cost efficiency, aligning with the objective of this paper. In such systems, SSDs are typically allocated to different tenants for separate use and accessed as raw devices. Simultaneously, an elastic resource management mechanism is required to achieve higher utilization [86]. Espresso designs can also benefit high-performance storage servers by enabling resource sharing

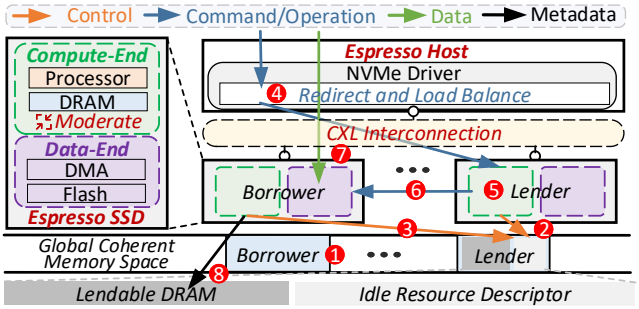


Figure 5: Overview of Espresso.

at the SSD level, thereby simplifying upper-layer software implementations and freeing host computing resources for other components of the storage stack.

Base components. Figure 5 illustrates the overview of Espresso. Compared with existing JBOF designs, there are four main differences: (1) Espresso replaces conventional PCIe interconnections with CXL to enjoy its high performance and cache coherence; (2) Espresso breaks the black-box constraint of traditional SSDs and enables fine-grained management of SSD internal resources (§ 4.2); (3) Espresso only reserves moderate computing resources (i.e., weaker processor and smaller DRAM) within each SSD at low BOM cost while satisfying burst I/O performance demands via resource harvesting; (4) Espresso makes a minor modification to the host NVMe driver for I/O redirection and load balance. For simplicity, we assume SSDs in Espresso are homogeneous (i.e., equipping the same hardware and running the same firmware), matching the common practice in JBOF markets [110]. We will discuss heterogeneous scenarios in § 6.

Workflows. During device initialization (e.g., system reboot [80]), each SSD exposes portions of its onboard DRAM via CXL interconnections. These exposed DRAM make up a global coherent memory space, facilitating inter-SSD communication (❶). Afterward, if an SSD is idle (i.e., *lender*), it calculates how many computing resources (i.e., processor and DRAM) can be lent out and announces this availability by writing its *idle resource descriptors* (❷, § 4.3). When the computing resources in one SSD (i.e., *borrower*) are in short supply (e.g., an I/O burst comes or a high DRAM miss ratio occurs), it searches all other SSDs' idle resource descriptors and chooses a lender for resource harvesting (❸, § 4.3). This step can be repeated multiple times to borrow more resources from multiple SSDs. For processor harvesting (§ 4.4), the host NVMe driver redirects portions of NVMe I/O commands originally targeting the borrower to the lender (❹). The lender then assists in serving I/O commands with its idle processor by operating the borrower's metadata (❺). Subsequently, the lender sends DMA and flash operations to the DMA engine and flash controller of the borrower (❻). The borrower then executes these operations to transfer data **directly** between the host and its flash backbone, without passing through the lender (❼). For DRAM harvesting (§ 4.5), the borrower can directly cache parts of its mapping table in the lender's

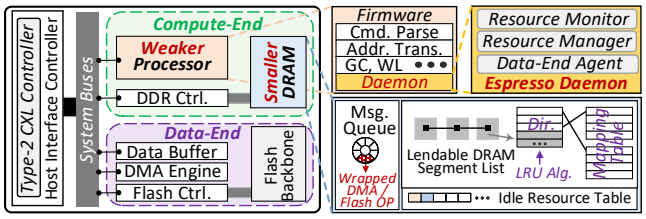


Figure 6: Disaggregated SSD architecture.

DRAM (❸). This harvested DRAM improves the cache hit ratio and reduces access to the slow flash backbone for the mapping table, leading to a higher I/O performance. SSD internal housekeeping tasks (e.g., garbage collection [123] and bad block management [47]) occur infrequently and consume fewer computing resources than I/O handling. Therefore, in Espresso, these tasks are executed by the borrower itself without resorting to the lender. However, when an SSD is busy with housekeeping, it can borrow computing resources for I/O handling, mitigating performance impacts.

4.2 Disaggregated SSD Architecture

Conventional SSDs are black boxes in which the onboard computing and flash resources are tightly coupled and invisible to external systems. This agnostic causes resource stranding issues (cf. § 3.1). Tackling this challenge, Espresso employs a disaggregated SSD architecture that decouples SSD into two disjoint parts: *compute-end* and *data-end*, as shown in Figure 6. Compute-end comprises computing resources, such as ARM processor, DDR controller, and onboard DRAM. It is responsible for executing SSD firmware tasks (e.g., I/O parsing and address translation). Data-end encloses flash resources (e.g., flash controller and backbone) and data-related components (e.g., DMA engine and data buffer). This part is in charge of data transfer and flash I/O. Moreover, a Type-2 CXL controller [39] is employed to facilitate CXL-related operations, such as exposing onboard DRAM to the host and peer SSDs in Espresso and operating the exposed DRAM of other SSDs coherently. Specifically, during system initialization, the CXL controller of each Espresso SSD registers its local DRAM as *global fabric-attached memory (G-FAM)* to the CXL fabric manager (FM) [20]. Afterward, SSD processor can access peer SSDs' G-FAM via `load/store` instructions (e.g., `LDR/STR` in `aarch64` ISA [25]), which will be interpreted as CXL `MemRd/MemWr` requests and then executed by the CXL controller. Moreover, the CXL controller is responsible for maintaining the coherence of the SSD's local DRAM with `BISnp` and `BIRsp` requests (i.e., in HDM-DB mode [20]).

In the compute-end, an *Espresso daemon* is implemented in SSD firmware and runs on the ARM processor. It contains three components: (1) A *resource monitor* is deployed to monitor the resource utilization of both compute-end and data-end. Specifically, for compute-end, the resource monitor periodically (e.g., 10 ms) polls the Performance Monitor Unit (PMU) [17, 24] of the ARM processor to track its utilization (i.e., busy clocks). DRAM resource is measured by the miss ratio of mapping table lookup. For data-end, the resource monitor gets utilization from the embedded flash monitor module, which is typically implemented as channel-level hardware busy clock counters in existing flash controller designs [51, 104]. Espresso takes the average utilization across all flash channels; (2) A *resource manager* is employed to borrow or lend computing resources based on the current utilization reported by the resource monitor; (3) A *data-end agent* bridges lender's compute-end with borrower's data-end. To be specific, the borrower's data-end agent maintains portions of onboard DRAM as globally visible message queues [94]. Thereby, the lender's compute-end can access the borrower's data-end by enqueueing wrapped DMA and flash operations (cf. § 2.1) to the borrower's message queues. The borrower's data-end agent then dequeues and unwraps these operations and sends them to the DMA engine and flash controller for data transfer and flash I/O.

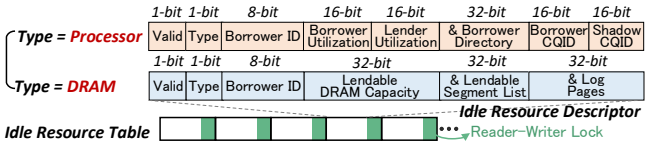


Figure 7: Data structures for resource management.

4.3 Decentralized Resource Management

As shown in Figure 7, each Espresso SSD maintains portions of its onboard DRAM as *idle resource table*, consisting of multiple *idle resource descriptors*. These data structures are visible to the host and all peer SSDs in Espresso and are synchronized with reader-writer locks [13] (i.e., a `pthread_rwlock_t` for each idle resource descriptor in C program). There are two formats of idle resource descriptors used to describe idle processor and DRAM resources, respectively. Both contain five messages: (1) One valid bit points out whether this descriptor is valid; (2) One type bit presents the type of idle computing resources (i.e., processor or DRAM); (3) 8 bits record the identification of the borrower (0xFF means that the resource has not been borrowed). Espresso assigns a unique identification to each SSD during device initialization [80]; (4) 32 bits depict the amount of idle resources. Specifically, for idle processor, these 32 bits are used to indicate the current processor utilization of both the borrower and the lender (16 bits for each) for load balance (cf. § 4.4). Both the lender and the borrower will update this field periodically (e.g., 10 ms) after harvesting begins. For idle DRAM, these 32 bits indicate the capacity of the lendable DRAM, maintained by the lender; (5) 64 bits record the essential information for resource sharing. In particular, for idle processor, 32 bits indicate the address of the borrower's mapping table directory, while the other 32 bits record the CQIDs of borrower CQ and shadow CQ (16 bits for each) for I/O redirection (cf. § 4.4); For idle DRAM, 32 of the 64 bits

point to the header of the lendable DRAM segment list, while the other 32 bits point to the start address of log pages for crash consistency (cf. § 4.5).

With the aforementioned data structures, Espresso enables decentralized resource management. Specifically, SSDs in Espresso can lend out their idle resources by writing the idle resource descriptors with writer locks. Moreover, resource borrowing is facilitated by searching the idle resource tables of all peer SSDs with reader locks and choosing one (or more) idle SSD based on best-fit algorithm [98] to harvest (i.e., writing the borrower identification of the chosen idle resource descriptor with writer lock). If the borrower no longer requires the borrowed resources (e.g., I/O burst completes), it sets the borrower identification of the idle resource descriptors to 0xFF. In turn, if the lender no longer wants to lend resources, it tags the valid bit of the descriptor as invalid. The lender, borrower, and host need to periodically check and update the idle resource descriptor according to current resource utilization and take corresponding actions to start and end harvesting (cf. § 4.4 and § 4.5). Note that Espresso does not require accurate clock synchronization among lender, borrower, and host. However, if the synchronization period is too long, Espresso may suffer from delays in the start and end of harvesting, thereby affecting performance (e.g., when a lender stops lending resources due to bursts, if the borrower and host fail to synchronize in time, some borrower's requests will still be redirected to the lender, causing resource contention). We empirically set the period as 10 ms with a trade-off between this performance loss and the synchronization overhead.

4.4 Transparent Processor Harvesting

Trigger conditions. Espresso SSD triggers processor resource harvesting based on the busy status of both the processor and data-end, as summarized in Table 1. They are regarded as busy if their current utilization exceeds a configurable watermark (e.g., 75%), otherwise considered underutilized. If both the processor and the data-end are busy (❶), the SSD does nothing as it has no available processor for sharing. Also, borrowing extra processor yields minor as the data-end has been exhausted. In comparison, whenever the processor is underutilized (❷ and ❸), the SSD can lend out this resource. This can happen when the SSD is bottlenecked by the flash backbone (❷) or the whole SSD is idle (❸). Lastly, if only the processor is busy (❹), the SSD can borrow processor resources to maximize the I/O parallelism of the data-end, achieving a higher throughput. Correspondingly, SSDs cancel borrowing or lending when the status of their resources no longer satisfies the above trigger conditions.

Transparent I/O redirection. Espresso harvests idle processors by redirecting parts of the borrower's NVMe I/O commands to the lender. The lender then locates the borrower's mapping directory and table with the address recorded in the idle resource descriptor (cf. § 4.3). Afterward, the lender can

ID	Processor	Data-end	Action	Possible reason
❶	Busy	Busy	None	Mixed I/O burst
❷	Underutilized	Busy	Lend out	Write burst (cf. § 3.1)
❸	Underutilized	Underutilized	Lend out	No I/O arrives
❹	Busy	Underutilized	Borrow in	Read burst (cf. § 3.1)

Table 1: Trigger conditions of processor harvesting.

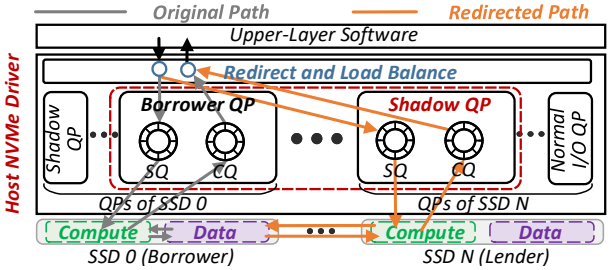


Figure 8: Transparent I/O redirection.

access the borrower's mapping tables and help serve I/O commands with its idle processor. Espresso SSD employs fine-grained locks to enable efficient inter-SSD synchronization of the mapping table, inheriting from prior multi-core SSD designs [133]. Espresso realizes I/O redirection by slightly modifying the host NVMe driver, which is the unique entrance of NVMe SSDs. This modification is transparent to the upper-layer applications (e.g., file system), ensuring compatibility. As shown in Figure 8, when initializing NVMe SSDs [80], Espresso reserves a few NVMe I/O queue pairs (QPs, each enclosing an SQ and a CQ) of each SSD as *shadow QPs*. When lending processor resources, the lender points out the identification (e.g., CQID [79]) of one of its shadow QPs in the idle resource descriptor. Subsequently, the borrower chooses one of its normal I/O QPs (named *borrower QPs*) for I/O redirection and also records its identification in the idle resource descriptor (cf. § 4.3). Thereafter, the host NVMe driver binds the borrower QP with the shadow QP, and submits parts of NVMe I/O commands targeting the borrower SQ to the shadow SQ. Following this, the lender fetches NVMe I/O commands from the shadow SQ and helps handle these commands. In the backward, the NVMe driver collects results from both borrower CQ and shadow CQ and then commits I/O completions to upper-layer software. When ending harvesting, the host NVMe driver no longer redirects I/O commands to the shadow SQ. Once all I/O commands within the shadow QP are completed, the host unbinds the shadow QP from the borrower QP, leaving them for future resource harvesting.

Holistic load balance. The host NVMe driver selectively redirects I/O commands to the lender with a holistic load balance algorithm. This algorithm is two-fold. On the one hand, the NVMe weighted round-robin (WRR) feature [80, 121] enables setting a weight for each NVMe I/O SQ, which indicates the priority of command fetching. For example, if the weights of two SQs are 2 and 1, respectively, the SSD serves two I/O commands from the former SQ and then serves one command from the latter. With this feature, Espresso can assign the shadow SQ of the lender with a low weight if it wants to minimize the performance impact on the lender's

own I/O. On the other hand, the host reads the idle resource descriptor periodically (e.g., 10 ms) to figure out the current processor utilization of both the borrower and lender (cf. § 4.3). Then, it controls the number of NVMe I/O commands sent to the borrower and lender with the following formula:

$$\frac{N_{borrow}}{N_{lend}} = \frac{U_{lend}}{U_{borrow}} \times \frac{\sum_{lend} W}{W_{shadowSQ}} \times \frac{W_{borrowSQ}}{\sum_{borrow} W}$$

N_{borrow} and N_{lend} represent the numbers of I/O commands sent to the borrower and lender, respectively. U_{borrow} and U_{lend} are the processor utilization of the borrower and the lender. $W_{borrowSQ}$ and $W_{shadowSQ}$ represent the weights of the borrower SQ and the shadow SQ. Lastly, $\sum_{borrow} W$ and $\sum_{lend} W$ are the total weights of all NVMe I/O SQs in the borrower and the lender. With this formula, Espresso can balance processor utilization by selectively redirecting I/O commands. For example, if N_{borrow}/N_{lend} is 3, Espresso redirects I/O commands to the lender with a 25% probability.

4.5 Persistent DRAM Harvesting

Trigger conditions. Espresso SSD manages DRAM resources in *segments* (2 MB by default) and caches mapping table with LRU replacement algorithm. Espresso SSD decides whether to borrow or lend DRAM based on the miss ratio curve (MRC) of current mapping table lookup patterns. Specifically, Espresso SSD adopts SHARDS [117], a lightweight and efficient algorithm, to predict MRC online. Based on the predicted MRC, SSD can lend out all the spare DRAM segments, which has no help on a lower miss ratio (i.e., the cached mapping table will not be accessed in the near future), minimizing the effect of cache pollution from the borrower. Moreover, SSD tries to borrow sufficient DRAM that reduces its miss ratio to below a given threshold (e.g., 10%). Correspondingly, lender reclaims DRAM when it predicts more DRAM in need to achieve a lower miss ratio. In turn, borrower returns DRAM when its local DRAM is sufficient to keep the miss ratio within the given threshold.

Crash consistency. Borrower harvests DRAM by temporarily caching parts of its mapping table in lender's lendable DRAM segments (i.e., as recorded in the idle resource descriptor, cf. § 4.3). In turn, borrower needs to read back the *offsite metadata* (i.e., the mapping table cached in the lender's DRAM) to its flash backbone when ending harvesting. While this idea can be facilitated by the cache-coherent capability of CXL, there still remains an open question in practice, that is, how to guarantee crash consistency of the off-site metadata. To be specific, for high availability, enterprise SSDs are typically demanded to deliver *power loss protection (PLP)* [80]. When an SSD suddenly loses power [77, 116, 135], the power hold-up circuit and capacitor in the SSD (e.g., the 2.3 mF tantalum capacitor in Samsung 845DC SSD [89]) immediately flushes the data and metadata buffered in the processor cache and onboard DRAM to the persistent flash backbone. This design ensures crash consistency of the SSD. However, if the borrower's dirty mapping table is exclusively cached in lender's DRAM, the borrower cannot provide PLP to the offsite metadata. For instance, if the lender SSD is permanently unplugged from the JBOF, the borrower cannot recover its mapping table to locate data and thus suffers from data loss (or needs sophisticated reconstruction through special hardware [37, 138]).

Tackling this issue, Espresso deploys a log-based crash consistency mechanism to protect the offsite metadata. In particular, when DRAM harvesting begins, the borrower vacates a 4 KB log page in its local DRAM for each harvested DRAM segment. Afterward, whenever modifying offsite metadata in the harvested segment, the SSD, either borrower or lender, needs to commit a log (e.g., redo log [33]) to the log page associated with the segment. Moreover, lenders need to ensure the log has been written back to the borrower with cacheline flush instructions. When the log page of a harvested DRAM segment is full, the segment will be flushed back to the borrower's flash backbone, after which the corresponding log page can be cleared and reused. Note that while the log operation introduces extra memory write overhead, it is relatively minor (i.e., hundreds of *ns*) compared to the flash I/O (i.e., tens of μs) caused by DRAM miss. If the lender fails (e.g., multiple I/O timeouts occur [80]), the host NVMe driver first notifies the borrower to recover its offsite metadata by replaying the logs in the local log pages. Afterwards, NVMe driver resubmits in-flight NVMe I/O commands in the shadow SQ to the borrower and then unbinds the borrower QP with the shadow QP. If the borrower fails, the host NVMe driver first notifies the lender to recycle the lent resources by clearing the harvested DRAM segments and resetting the corresponding idle resource descriptors. Subsequently, NVMe driver clears the corresponding shadow QP, leaving it for future harvesting.

4.6 Implementation

Cache coherence region. In Espresso, data are transferred without cache coherence using CXL.io sub-protocol [20]. However, Espresso does necessitate cache coherence among the host and SSDs for the following metadata: (1) the message queues within the data-end agent (cf. § 4.2), accounting for 128 KB per SSD by default; (2) the idle resource table (cf. § 4.3), which takes 256 B per SSD; (3) the cached mapping table and its auxiliary structures (e.g., locks). As will be shown in § 5.4, 0.25 GB per TB flash capacity is sufficient (i.e., 12 GB for twelve 4 TB SSDs).

Note that CXL standard does not mandate a specific implementation of hardware-managed cache coherence. Espresso employs a directory-based approach [1], in which the hardware overhead primarily depends on the number of cachelines that need to be tracked simultaneously. This is proportional to the number of in-flight requests of the borrower in Espresso, rather than the total metadata size. Our default setting employs 1 K directory entries to track 64 KB cache per SSD.

Prototype. We implement the host-side design of Espresso (e.g., I/O redirection and load balance, cf. § 4.4) in the NVMe driver of Linux kernel v5.15 [62], following the NVMe specification [80]. Due to the lack of publicly available CXL 3.0 hardware, we validate the firmware-side modification of our disaggregated SSD designs on DaisyPlus OpenSSD board [51, 112], a PCIe-based SSD development platform. This board integrates a quad-core ARM Cortex-A53 processor, 2 GB DRAM, and adequate FPGA resources. SSD firmware runs on the ARM processor, while the host interface controller and flash controller are implemented on the FPGA part. We inherit the core functionalities (e.g., garbage collection) of the SSD firmware (in C language) from DaisyPlus, but modify the I/O path to realize Espresso daemon. The data-end agent takes 114.2 ns, on average, to dequeue and unwrap a DMA/flash operation from the message queue in local DRAM (cf. § 4.2). Also, it takes 321.9 ns to commit a redo log to the local log pages for crash consistency (i.e., prepare and flush a log to the local DRAM, cf. § 4.5). We cross-validate the performance model used in our simulator and emulator with these results.

Simulator. We use SimpleSSD [100], a popular full-stack simulator, to evaluate Espresso. Prior studies [32, 43] have demonstrated that SimpleSSD can accurately model the performance of host (e.g., CPU and DRAM) and modern SSD components (e.g., embedded ARM processor, DRAM, and flash backbone). We also extend SimpleSSD with our industrial collaborators by 18 K LOC to support lots of detailed SSD techniques, such as incremental step pulse programming [106], SLC cache [115, 130], and read retry [127]. These efforts ensure the accuracy of the simulator. To evaluate the CXL fabrics, we integrate Xerxes [6], a cycle-accurate CXL simulator, which can accurately model the features defined in CXL 3.0 standard [20]. We use McPAT [60] and DRAMPower [14] to examine energy consumption.

Emulator. For cross-validation, we also port Espresso to a NUMA-based emulation platform. As recommended by prior work [58, 59, 83, 132], we mimic CXL fabric with cross-NUMA access and emulate each SSD with a dedicated socket (i.e., Intel Xeon 8562Y+ CPU [38]) running NVMeVirt [49], a popular SSD emulator. However, constrained by the number of sockets (e.g., 2), such an emulation platform cannot accurately mimic the tens of SSDs in JBOF. Therefore, we opt to conduct most of our evaluation on the simulator, while taking a NUMA emulation in § 5.6.

5 Evaluation

5.1 Experimental Setup

System configurations. As listed in Table 2, we configure the simulated JBOF system based on SuperMicro SSG-229J-5BU24JBF [110], one of the up-to-date JBOF products, which supports at most 12 SSDs per DPU. The simulated DPU contains a 16-core 2.1 GHz ARM processor and 16 GB DRAM, aligning with BlueField-3 [78], and acts as JBOF host (cf. § 2.1). The simulated SSD follows the configuration of commodity storage device [18, 71, 91], which delivers 14 GB/s and 10 GB/s peak read and write bandwidths. The SSD's processor encloses 6 ARM cores with `aarch64` ISA [25] running at 1 GHz frequency. In addition, its onboard DRAM can accommodate entire mapping table (i.e., 1 GB per TB flash). DPU and SSDs are connected with CXL fabric in tree topology (cf. Figure 9).

Host	16-core 2.1 GHz ARM processor and 16 GB DDR5-5600 DRAM		
SSD	**Performance**	**ARM processor**	**Flash backbone**
	Read/Write: 14/10 GB/s	6 Cores @ 1 GHz ISA: aarch64 (ARMv8)	8 Channel (2400 MT/s, 8-bit) / 8 Die / 4 Plane / 1024 Block / 1024 Page / 16 KB, 4TB in total
	Detailed techniques	**DRAM**	Read/Program: LSB: 30/200 us, CSB: 45/280 us, MSB: 60/400 ms. Erase: 3 ms
	ISPP, multi-plane, SLC cache, read retry...	1 GB per TB flash, DDR4-3200	
CXL	CXL 3.0 / PCIe 6.0 * 2 lanes = 16 GB/s per SSD, 256B FILT, tree topology		
Energy param.	Flash op. voltage=3.3V, $I_{READ,PROG,ERASE}$=25mA, $I_{BUSIDLE}$=5mA, I_{STDBY}=10uA $CXL/PCIe_{PHY}$=6 pJ/bit; SSD processor=6.45W; DRAM read/write=22 pJ/bit		

Table 2: System configurations.

Workloads	src	DAP	MSNFS	mds	DB	Fuji-0	Fuji-1
Read cnt. ratio (%)	11.3	56.2	67.2	92.8	98.0	82.7	86.3
Avg. read size (KB)	8.1	62.1	9.6	60.1	9.5	35.7	32.7
Avg. write size (KB)	7.1	97.2	11.1	13.8	743.3	10.7	13.3
Workloads	**Fuji-2**	**Tencent-0**	**Tencent-1**	**Tencent-2**	**Ali-0**	**Ali-1**	**Ali-2**
Read cnt. ratio (%)	87.6	84.3	2.0	98.2	98.1	81.3	11.0
Avg. read size (KB)	39.3	31.2	12.5	47.0	37.0	370.4	26.0
Avg. write size (KB)	6.7	8.8	289.5	7.0	16.8	394.5	30.0

Table 3: Workload characteristics.

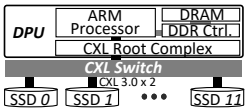


Figure 9: Topology.

JBOF platforms. We compare Espresso with six other designs. (1) `Conv`: conventional JBOF design, in which all SSDs equip abundant computing resources (i.e., 6 ARM cores and 1 GB DRAM per TB flash) for high I/O parallelism; (2) `OC`: OCSSD-based JBOF design that reserves minimum computing resources in SSDs but utilizes host resources to execute firmware and cache metadata (cf. § 3.1); (3) `Shrunk`: compared to `Conv`, it **shrinks** the computing resources in each SSD. By default, it halves the resources (i.e., 3 ARM cores and 0.5 GB DRAM per TB flash). We also evaluate different reservation settings in § 5.4; (4) `VH`: based on `Shrunk`, it uses the simple virtualization and harvesting approach (cf. § 3.1) to improve I/O performance; (5) `VH(ideal)`: an ideal variant of `VH`, in which no copyback is required; (6) `ProcH`: `Shrunk` with our processor harvesting designs (cf. § 4.4); (7) `Espresso`: a cost-efficient JBOF design that includes all the techniques proposed in this paper. It reserves the same amount of computing resources in each SSD as `Shrunk` (i.e., **half** of the computing resources in each SSD of `Conv`).

Workloads. We conduct evaluations with both microbenchmarks and various real workloads collected from production environments [3, 45, 56, 102, 122, 134]. Table 3 lists their key characteristics. To intuitively demystify the interactions between borrowers and lenders, we run workloads on 6 of the 12 SSDs (i.e., `borrowers`) by default, while keeping the other

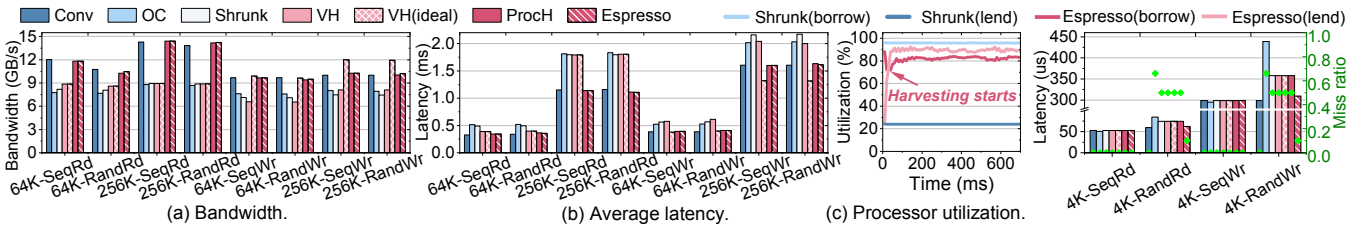


Figure 10: Performance benefits of processor harvesting.

Figure 11: DRAM harvesting.

6 SSDs idle (i.e., `lenders`). We also take sensitivity studies on varied numbers of borrowers and lenders in § 5.4 and examine complex scenarios where all 12 SSDs have different workloads in § 5.5. To demonstrate end-to-end performance improvement brought by our designs, we conduct comparisons on varied application instances in § 5.6.

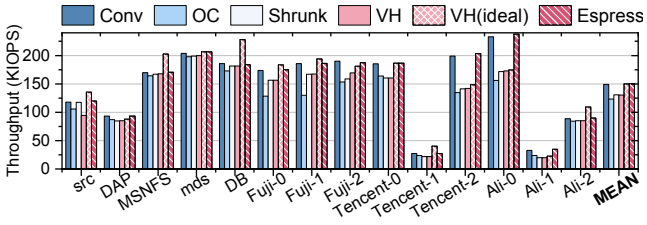


Figure 12: Throughput comparison in real workloads.

5.2 Benefit Analysis

Processor harvesting. Figure 10 illustrates the throughput and average latency comparison in microbenchmarks. For simplicity, we present the average performance of the 6 SSDs that run workloads (i.e., `borrowers`). We set the I/O depth as 64 to mimic the throughput-intensive scenarios and examine different I/O sizes from 64 KB to 256 KB. In comparison to `Conv`, `OC` and `Shrunk` suffer 27.8% and 29.2% throughput loss in all workloads, on average. Similar deterioration can also be observed in the latency comparison (i.e., 44.1% and 46.4% higher). This is because the insufficient processor resources in `OC` and `Shrunk` cannot afford such intensive I/O patterns and become the performance bottleneck. `VH` and `VH(ideal)` also lag far behind `Conv` in read workloads, as the simple virtualization and harvesting approach has no help to read requests (cf. § 3.1). Compared with `Shrunk`, `VH` and `VH(ideal)` succeed in improving write performance by redirecting parts of write requests to `lenders`. VH(ideal) can even outperform `Conv` by 10.2%. This can be attributed to the harvested flash resources of `lenders`, as data is temporarily stored in the `lenders'` flash backbone via their flash channels. However, such gains are swept out after copyback occurs. As a result, `VH` still falls behind `Conv` by 25.6%. On the contrary, `Espresso` achieves comparable performance to `Conv` in all workloads with only half of the computing resources. This is because `Espresso` can harvest idle processors of `lenders` for I/O serving. Figure 10c shows the average processor utilization of `borrowers` and `lenders` in 256 KB sequential read test. `Espresso` achieves 50.4% higher utilization than `Shrunk`.

DRAM harvesting. To evaluate how much Espresso can benefit from DRAM harvesting, we set an experiment to analyze the I/O performance in latency-sensitive scenarios (i.e., 4 KB reads and writes). We set the I/O depth to 1 in this test. Figure 11 illustrates the average latency and mapping table miss ratio of different JBOF settings. Without sufficient DRAM to buffer the entire mapping table, `OC`, `Shrunk`, and `ProcH` experience 66.2%, 49.7%, and 49.7% miss ratios in random read workloads, thereby causing 41.4%, 24.7%, and 24.7% higher latencies, compared to `Conv`. Similar degradation also exists in random write tests. The virtualization and harvesting approach (`VH`) does not work for DRAM harvesting. This is because even if redirecting write requests to `lenders`, `lenders` still suffer from the miss penalty due to insufficient DRAM. In contrast, the DRAM harvesting designs (cf. § 4.5) in `Espresso` enable `borrowers` to borrow idle DRAM resources from `lenders` to buffer mapping table. As a result, `Espresso` achieves comparable latencies to `Conv`.

Improvements in real workloads. Figure 12 presents the throughput comparison in diverse real workloads collected from production environments. Compared with `Conv`, `OC` and `Shrunk` suffer 16.2% and 13.4% throughput loss in all workloads, on average, owing to the stressed computing resources. `VH(ideal)` outperforms `Shrunk` in write-dominated workloads (e.g., 15.5% in `src`) via write request redirecting. However, the substantial overhead of data copyback dispels such a mirage. Consequently, `VH` still lags behind `Conv` by 14.0%. In contrast, `Espresso` can aid all workloads with diverse I/O types, eliminating the copyback overhead. Specifically, although employing the same amount of computing resources, `Espresso` outperforms `Shrunk` and `VH` by 19.2% and 20.0%. Moreover, `Espresso` performs better than `VH(ideal)` in some workloads (e.g., `Ali-0`). This is because, in these read-dominated workloads, the requested data resides exclusively in the borrower's flash backbone. Without our harvesting design, lenders cannot assist the borrower in serving read requests due to the lack of the target data in the lenders' flash backbone. `Espresso` also achieves comparable throughput to `Conv` even with only half the computing resources (cf. § 5.1), which proves that our design can satisfy demanded performance targets via inter-SSD resource sharing. Note that in some I/O-intensive workloads, `Espresso` slightly outperforms `Conv` (e.g., 2.0% in `Ali-0`). We attribute this minor difference to the doubled CXL lanes for NVMe command

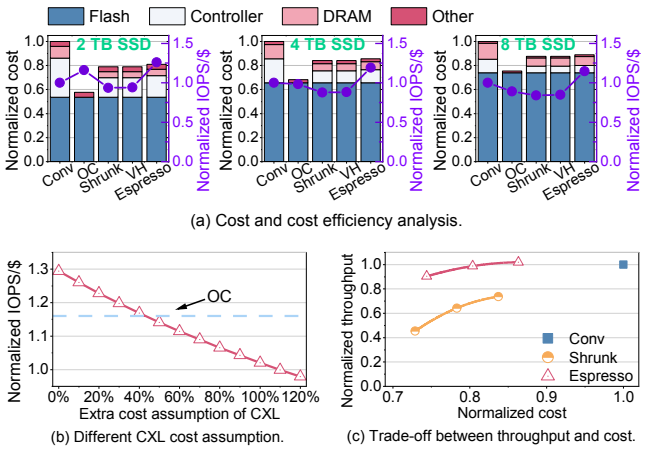


Figure 13: BOM cost evaluation normalized to `Conv`.

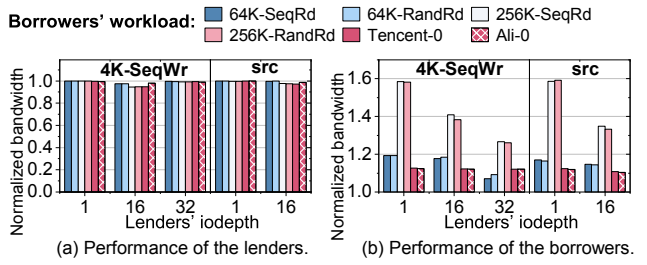


Figure 14: Interaction between lenders and borrowers.

fetching (i.e., `lenders` and `borrowers` in `Espresso` can concurrently fetch commands with their own CXL lanes).

BOM cost saving. We further evaluate the BOM costs of SSDs in different JBOF platforms. According to the current prices on the market [27, 65, 67, 74, 101, 113, 114], we identify the costs of NAND flash, DDR4 DRAM, enterprise SSD controller, and other expenses (e.g., PCB board and packaging) as $4.95 per 128 GB, $7.2 per GB, $48, and $6, respectively. We assume the halved computing resources (i.e., SSD controller and DRAM) in `Shrunk` and `VH` consume halved costs. According to prior work [111], we estimate the prices of CXL-enabled SSD controller and DRAM in `Espresso` are 10% higher than those in `Shrunk` by default. As shown in Figure 13a, `Espresso` succeeds in saving BOM cost by 19.0% for 2 TB SSDs, compared with `Conv`. Although SSDs in `Espresso` are more expensive than those in `OC` and `Shrunk`, such expenses are worth given the improved performance. Figure 13a also depicts the normalized cost efficiency (i.e., IOPS/$) in `Ali-0` workload. `Espresso` outperforms all other designs (e.g., 19.7% higher than `OC`) in this metric. Figure 13b further shows the cost efficiency of `Espresso` under different CXL cost assumptions (i.e., how much more expensive the CXL-enabled SSD controller and DRAM are than those in `Shrunk`). `Espresso` outperforms `OC` as long as the extra BOM costs brought by CXL do not exceed 40%. Figure 13c illustrates the trade-off between throughput and cost of different designs (cf. § 5.4 for how to vary these two parameters). We assume there is a sufficient number of idle SSDs for harvesting. `Espresso`

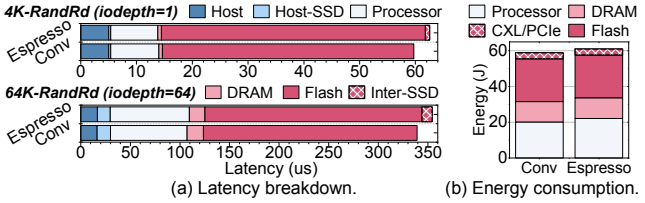


Figure 15: Overhead analysis (latency and energy).

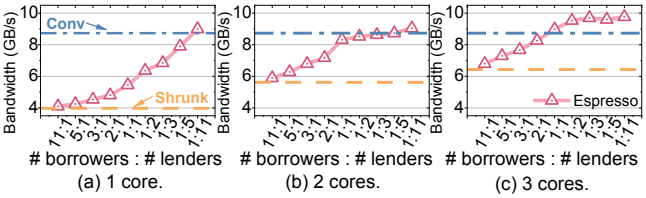


Figure 16: Sensitivity study on different processor resources.

demonstrates better balance between these two metrics than `Conv` and `Shurnk` (i.e., the curve of `Espresso` is more toward the upper-left corner).

5.3 Overhead Analysis

Performance impact on lenders. In this test, we run varied I/O-intensive workloads on `borrowers`, while `lenders` serve moderate I/O requests. We set the I/O depth as 64 for `borrowers` while varying the I/O depth from 1 to 32 for `lenders` to mimic different degrees of I/O pressure. Note that `lenders'` processors are too busy to be lent when running `src` workload in 32 I/O depth. Therefore, we omit this result. Figure 14 presents the throughput of `lenders` and `borrowers` in `Espresso`. We have normalized the results to that of `Shrunk`, where no resource is lent out (i.e., best case for `lenders`) or borrowed in (i.e., worst case for `borrowers`). Resource lending causes negligible performance loss (1.3% on average, cf. Figure 14a) for `lenders`. This is because, with our holistic load balance algorithm (cf. § 4.4), `lenders` can reserve sufficient computing resources to handle their own I/O commands. The throughput of `borrowers` improves 15.5%, 23.3%, and 30.0%, when the `lenders` serve 4 KB sequential writes in I/O depths of 32, 16, and 1 (cf. Figure 14b). This is because, with lighter workloads, `lenders` can lend out more resources to help `borrowers` serve I/O commands.

Extra latency. Espresso designs (e.g., remote metadata access and synchronization) can introduce extra latency. To understand such overhead, we break down the latency of 4 KB and 64 KB random reads into six parts, as shown in Figure 15a. `Host` is the time consumed by the host I/O stack (e.g., NVMe driver). `Host-SSD` is the time for data and NVMe command transfer between the host and SSD. `Processor` is the time consumed to execute SSD firmware (e.g., I/O parsing and address translation). `DRAM` encloses the time of onboard DRAM accesses (e.g., reading mapping table). `Flash` represents the time of flash operations (e.g., flash read). Finally,

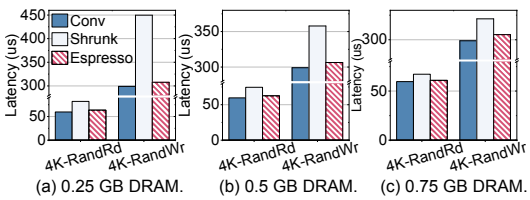


Figure 17: Analysis of varied DRAM resources.

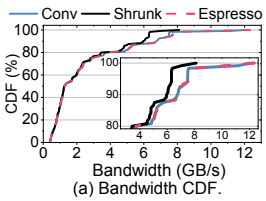


Figure 18: Comparison in complex scenarios.

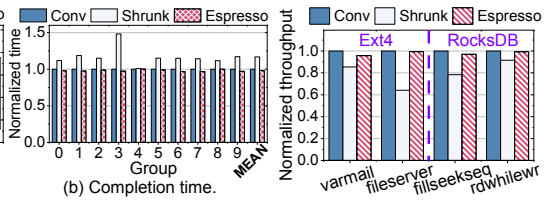


Figure 19: NUMA test.

`Inter-SSD` includes the time taken on the CXL interconnection. In both `Conv` and `Espresso`, `Flash` dominates the latency. Compared with `Conv`, `Espresso` causes 3.3% more `Flash` overhead because of the sporadic DRAM miss (cf. § 5.2). `Espresso` also takes 3.1% more `Processor` time for inter-SSD synchronization. There is no obvious difference in terms of `Host` overhead, thanks to our decentralized resource management scheme (cf. § 4.3). Moreover, Espresso only takes 20 ns more amortized host CPU time for each I/O command to compute the load balance formula for redirecting (cf. § 4.4). `Espresso` takes minor `Inter-SSD` overhead (up to 2.9%), because CXL interconnection has extremely high speed and delivers sub-microsecond remote access.

Energy consumption. Figure 15b illustrates the energy consumption to conduct `Fuji-0` workload of different designs. Compared with `Conv`, `Espresso` takes 3.5% more energy, because of the added Espresso daemon and CXL-enabled inter-SSD communication (cf. § 4.2). However, this minor overhead brings huge rewards, as it allows resource sharing to exploit the idle SSDs in JBOF, thereby achieving required I/O performance with reduced resources and costs (cf. § 5.2).

5.4 Sensitivity Study

Different processor resources. We examine the benefits brought by our designs in different processor resource configurations. For fair comparisons, we equip SSDs in `Shrunk` and `Espresso` with the same capacity of DRAM as SSDs in `Conv`. We vary the number of ARM cores in each SSD of `Shrunk` and `Espresso` from 1 to 3. We also change the ratios of numbers of `borrowers` and `lenders` from 11:1 to 1:11. Figure 16 shows the throughput in `Ali-0` workload. Without inter-SSD resource harvesting, the throughput of `Shrunk` decreases with fewer cores (up to 54.6% degradation in the 1-core setting, cf. Figure 16a). In contrast, `Espresso` achieves comparable throughput to `Conv` when there are enough `lenders` for harvesting. For example, in 2-core tests, `Espresso` achieves 97.7% performance of `Conv` when each `borrower` harvests two `lenders` (i.e., `1:2`). Note that, with excessive `lenders` for harvesting (e.g., when ratio is `1:11`), `Espresso` cannot further boost performance, owing to the limited throughput of flash backbone and overwhelming synchronization overhead.

Different DRAM resources. We reserve different capacities of DRAM in the SSDs within `Shrunk` and `Espresso`. For fair comparisons, we equip SSDs in all JBOF platforms with 6 ARM cores. We assume there is enough number of idle SSDs for DRAM harvesting, which has been witnessed in production environments (cf. § 2.2). Figure 17 illustrates the latency comparison. `Shrunk` experiences 44.0%, 22.3%, and 10.0% higher latency with 0.25, 0.5, and 0.75 GB DRAM per TB flash capacity, respectively. On the contrary, benefiting from our DRAM harvesting designs (cf. § 4.5), `Espresso` only introduces negligible latency increases (3.4% on average).

5.5 Complex Scenario

We examine complex scenarios where all SSDs have their own workloads. We randomly select 12 workloads from Tencent traces [134] as a group, and assign each workload to each of the 12 SSD. We repeat the experiment 10 times. Figure 18a presents the cumulative distribution function (CDF) of throughput across all 120 workloads. Echoing our findings in the previous tests, `Espresso` succeeds in fulfilling the burst I/O performance demands, even with only halved computing resources. To be specific, SSDs in `Espresso` achieve 12.3 GB/s peak throughput, while this value is only 8.1 GB/s in `Shrunk`. Figure 18b shows the workload completion time comparison. `Espresso` shortens at most 34.3% time than `Shrunk`, thanks to our inter-SSD resource sharing designs.

5.6 Application Test on NUMA Platform

We now evaluate Espresso designs on a 2-socket NUMA-based emulation platform (cf. § 4.6). We use one socket to mimic the borrower SSD, while the other socket acts as the lender SSD and the host. We choose Ext4 filesystem [68] and RocksDB [30] as the representative applications of JBOF and run them with `filebench` [31] and `db_bench` [29], respectively. Figure 19 shows the throughput comparison of different designs. Similar to our previous tests, `Espresso` outperforms `Shrunk` by 24.8% and achieves comparable throughput to `Conv` even with reduced computing resources, cross-validating the superiority of our designs.

6 Related Work and Discussion

SSD architecture. Multiple studies [28, 42, 47, 48, 76, 83, 124, 133] have been proposed to renovate SSD architectures. Decoupled SSD [47] decomposes SSD architecture into front-end (i.e., SSD controller) and back-end (i.e., flash backbone) and introduces a network-on-chip to facilitate communication among flash controllers. It improves the efficiency of

SSD internal data movement (i.e., garbage collection). In contrast, Espresso disaggregates SSD components based on their functionalities to enable fine-grained resource management and sharing. XHarvest [83] renovates SSD architecture and achieves high I/O performance through secure host resource harvesting. On the contrary, Espresso satisfies performance requirement via inter-SSD resource sharing.

Communication protocol. NVMe is the de facto communication protocol for most high-performance SSDs. We opt to implement I/O redirection and load balance on NVMe protocol to be compatible with existing I/O stacks, avoiding excessive software modification. To adapt to prior works [42,54,59,124,131] that access SSDs via `load/store` instructions, a similar I/O redirection and load balance mechanism can be implemented in the memory access path (e.g., the memory management subsystem of Linux kernel [64]).

Storage virtualization. Great efforts [36, 75, 86, 107] have been taken to virtualize storage devices and improve their utilization. BlockFlex [86] and FleetIO [107] enhance SSD utilization by harvesting idle flash resources, thereby facing critical challenges, such as limited read profit and huge data copyback overhead (cf. § 3.1). In contrast, Espresso focuses on the stateless computing resources and enables general and lightweight inter-SSD resources sharing with CXL fabric.

RAID. RAID [15] is a storage organization scheme that distributes I/O requests across multiple SSDs to improve utilization. However, RAID also treats SSDs as black boxes, which prevents fine-grained utilization of SSD internal resources, leading to stranding issues (cf. § 3.1). Moreover, RAID has been widely criticized for its substantial software overhead on the centralized host, resulting in poor scalability [41,128,129]. In contrast, Espresso enables fine-grained resource harvesting and supports decentralized and SSD-governing resource management, achieving high scalability. Additionally, Espresso is aimed to maximize the utilization of the underlying storage hardware (i.e., JBOF), which are effective to any types of software storage organizations (e.g., RAID) that users adopt.

JBOF with heterogeneous SSDs. Our design can be adapted to JBOFs comprised of heterogeneous SSDs (i.e., equipping different firmware and computing resources) with minor revisions. Specifically, the borrower can expose its firmware tasks as non-decompilable executable files [55, 125] in trusted execution environments (TEE) [8, 44, 83]. Therefore, the lender can seamlessly execute the borrower's firmware tasks with its general-purpose ARM processor while ensuring security. Considering the computing power disparity of varied SSDs, Espresso can replace the busy indicator (which is processor utilization now, cf. § 4.4) with a more general and absolute metric (e.g., current waiting queue depth) for load balance. Similar designs can also be applied in scenarios where SSD manufacturers are reluctant to expose SSD firmware details. Note that Espresso host only needs to know the current status of SSDs (e.g., idle resource table), which can be exposed via standardized specifications (e.g., NVMe Telemetry logs).

Espresso with future CXL. Considering the hardware cost, the trend of CXL evolution is toward supporting coherence over only portions of the memory region [9,57,136,137]. The limited coherence requirements of Espresso (cf. § 4.6) are aligned with this trend. One can further shrink the cache coherence region by dividing the tens of SSDs in Espresso into multiple groups and only allowing harvesting within the same group. In scenarios without hardware-supported cache coherence, software-based coherence mechanisms are required, albeit with significantly high synchronization overhead.

Practicality and compatibility. The main hardware modifications of Espresso are only the CXL interconnections and reduced SSD computing resources, while the other innovations (e.g., Espresso daemon) are implemented as software/firmware. Thanks to the backward-compatibility of CXL to PCIe and our support to NVMe protocol, Espresso SSD can act as a conventional SSD and adapt to existing I/O stacks and applications without violating compatibility. SSD industry [10, 50, 53, 72, 81, 103] is also actively exploring to evolve SSDs with CXL (e.g., Samsung CMM-H [103]) and decouple SSD architectures (e.g., Samsung AM9C1 [92, 93]). Espresso aligns with this trend and holds the promise.

7 Conclusion

While the numerous computing resources significantly elevate the BOM costs of SSDs to satisfy the performance requirement of burst I/O, the sporadic nature of I/O bursts causes severe SSD underutilization in JBOF scenarios. Tackling this issue, we propose Espresso, a novel JBOF design that reserves moderate computing resources in SSDs at low costs while achieving demanded I/O performance by employing CXL to facilitate fine-grained inter-SSD resource sharing. Our evaluation results reveal that Espresso improves SSD resource utilization by 50.4% and saves 19.0% BOM costs with negligible performance loss, compared to existing JBOF designs.

Acknowledgement

We sincerely thank the anonymous reviewers and our shepherd for their insightful feedback. This work is mainly supported by the National Key Research and Development Program of China under Grant No. 2023YFB4502702, the Natural Science Foundation of China under Grant No. 624B2004, 62472007, and 62332021. Guangyan Zhang is partly supported by the National Natural Science Foundation of China under Grant No. 62025203. Chenxi Wang is partly supported by the Chinese Academy of Sciences E545030 and E401430. Yiwei Luo is partly supported by the National Science Foundation of China under Grant No. 62372011, 92582118, and 62032001. The corresponding author is Jie Zhang.

References

[1] Anant Agarwal, Richard Simoni, John Hennessy, and Mark Horowitz. An evaluation of directory schemes for cache coherence. *ACM SIGARCH Computer Architecture News*, 16(2):280–298, 1988.

[2] Alibaba. Alibaba open cluster trace program. https://github.com/alibaba/clusterdata/blob/master/cluster-trace-v2018/trace_2018.md, 2018.

[3] Alibaba. Alibaba block traces. https://github.com/alibaba/block-traces, 2024.

[4] Michael Allison, Arun George, Javier Gonzalez, Dan Helmick, Vikash Kumar, Roshan R Nair, and Vivek Shah. Towards efficient flash caches with emerging nvme flexible data placement ssds. In *Proceedings of the Twentieth European Conference on Computer Systems*, pages 1142–1160, 2025.

[5] Pradeep Ambati, Íñigo Goiri, Felipe Frujeri, Alper Gun, Ke Wang, Brian Dolan, Brian Corell, Sekhar Pasupuleti, Thomas Moscibroda, Sameh Elnikety, Marcus Fontoura, and Ricardo Bianchini. Providing slos for resource-harvestingvms in cloud platforms. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 735–751, 2020.

[6] Yuda An, Shushu Yi, Bo Mao, Qiao Li, Mingzhe Zhang, Ke Zhou, Nong Xiao, Guangyu Sun, Xiaolin Wang, Yingwei Luo, and Jie Zhang. Xerxes: Extensive exploration of scalable hardware systems with cxl-based simulation framework. In *24th USENIX conference on file and storage technologies (FAST 26)*, 2026.

[7] Erick Bauman, Gbadebo Ayoade, and Zhiqiang Lin. A survey on hypervisor-based monitoring: approaches, applications, and evolutions. *ACM Computing Surveys (CSUR)*, 48(1):1–33, 2015.

[8] Erick Bauman, Huibo Wang, Mingwei Zhang, and Zhiqiang Lin. Sgxelide: enabling enclave code secrecy via self-modification. In *Proceedings of the 2018 International Symposium on Code Generation and Optimization*, pages 75–86, 2018.

[9] Daniel S Berger, Yuhong Zhong, Pantea Zardoshti, Shuwei Teng, Fiodar Kazhamiaka, and Rodrigo Fonseca. Octopus: Scalable low-cost cxl memory pooling. *arXiv e-prints*, pages arXiv–2501, 2025.

[10] Terry Cheng Bernard Shung, San Chang. Nvm express® (nvme®) technology support for cxl®: Unleashing computational workloads. https://sc23.supercomputing.org/proceedings/exhibitor_forum/exhibitor_forum_files/exforum118s2-file2.pdf, 2025.

[11] Matias Bjørling, Abutalib Aghayev, Hans Holmberg, Aravind Ramesh, Damien Le Moal, Gregory R Ganger, and George Amvrosiadis. Zns: Avoiding the block interface tax for flash-based ssds. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, pages 689–703, 2021.

[12] Matias Bjørling, Javier Gonzalez, and Philippe Bonnet. Lightnvm: The linux open-channelssd subsystem. In *15th USENIX Conference on File and Storage Technologies (FAST 17)*, pages 359–374, 2017.

[13] Irina Calciu, Dave Dice, Yossi Lev, Victor Luchangco, Virendra J Marathe, and Nir Shavit. Numa-aware reader-writer locks. In *Proceedings of the 18th ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 157–166, 2013.

[14] Karthik Chandrasekar, Christian Weis, Yonghui Li, Benny Akesson, Norbert Wehn, and Kees Goossens. Drampower: Open-source dram power & energy estimation tool. *URL: http://www. drampower. info*, 22, 2012.

[15] Peter M Chen, Edward K Lee, Garth A Gibson, Randy H Katz, and David A Patterson. Raid: High-performance, reliable secondary storage. *ACM Computing Surveys (CSUR)*, 26(2):145–185, 1994.

[16] Tae-Sun Chung, Dong-Joo Park, Sangwon Park, Dong-Ho Lee, Sang-Won Lee, and Ha-Joo Song. A survey of flash translation layer. *Journal of Systems Architecture*, 55(5-6):332–343, 2009.

[17] Gilberto Contreras and Margaret Martonosi. Power prediction for intel xscale® processors using performance monitoring unit events. In *Proceedings of the 2005 international symposium on Low power electronics and design*, pages 221–226, 2005.

[18] Crucial. Crucial t705 pcie 5.0 nvme. https://www.crucial.com/ssd/t705/ct2000t705ssd5a, 2025.

[19] CXL. Compute express link. https://computeexpresslink.org, 2025.

[20] CXL. Compute express linktm (cxltm) specification 3.2. https://computeexpresslink.org/wp-content/uploads/2024/12/CXL_3.2-Spec-Announcement_FINAL-1.pdf, 2025.

[21] Christoffer Dall and Jason Nieh. Kvm/arm: the design and implementation of the linux arm hypervisor. *Acm Sigplan Notices*, 49(4):333–348, 2014.

[22] Dell. Dell dell powerstore 500t storage array. https://www.delltechnologies.com/asset/en-ca/products/storage/technical-support/dell-powerstore-gen2-spec-sheet.pdf, 2023.

[23] Li Deng, Yu-Lin Ren, Fei Xu, Heng He, and Chao Li. Resource utilization analysis of alibaba cloud. In *Intelligent Computing Theories and Application: 14th International Conference, ICIC 2018, Wuhan, China, August 15-18, 2018, Proceedings, Part I 14*, pages 183–194. Springer, 2018.

[24] ARM Developer. Interaction with the performance monitoring unit (pmu). https://developer.arm.com/documentation/ddi0469/b/functional-description/operation/interaction-with-the-performance-monitoring-unit--pmu-, 2025.

[25] ARM Developer. Overview of the armv8 architecture. https://developer.arm.com/documentation/dui0801/l/Overview-of-the-Armv8-Architecture, 2025.

[26] Yaozu Dong, Zhao Yu, and Greg Rose. Sr-iov networking in xen: Architecture, design and implementation. In *Workshop on I/O virtualization*, volume 2. San Diego, CA, USA, 2008.

[27] DRAMeXchange. The global price of nand flash and lpddr. https://www.dramexchange.com/, 2024.

[28] Zelin Du, Shaoqi Li, Zixuan Huang, Jin Xue, Kecheng Huang, Tianyu Wang, and Zili Shao. Pipessd: A lock-free pipelined ssd firmware design for multi-core architecture. In *Proceedings of the 61st ACM/IEEE Design Automation Conference*, pages 1–6, 2024.

[29] facebook. db_bench. https://github.com/facebook/rocksdb/wiki/Benchmarking-tools, 2025.

[30] Facebook. Rocksdb. http://rocksdb.org/, 2025.

[31] filebench. A model based file system workload generator. https://github.com/filebench/filebench, 2025.

[32] Donghyun Gouk, Miryeong Kwon, Jie Zhang, Sungjoon Koh, Wonil Choi, Nam Sung Kim, Mahmut Kandemir, and Myoungsoo Jung. Amber: Enabling precise full-system simulation with detailed modeling of all ssd resources. In *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 469–481. IEEE, 2018.

[33] Jim Gray, Paul McJones, Mike Blasgen, Bruce Lindsay, Raymond Lorie, Tom Price, Franco Putzolu, and Irving Traiger. The recovery manager of the system r database manager. *ACM Computing Surveys (CSUR)*, 13(2):223–242, 1981.

[34] Zerui Guo, Hua Zhang, Chenxingyu Zhao, Yuebin Bai, Michael Swift, and Ming Liu. Leed: A low-power, fast persistent key-value store on smartnic jbofs. In *Proceedings of the ACM SIGCOMM 2023 Conference*, pages 1012–1027, 2023.

[35] Aayush Gupta, Youngjae Kim, and Bhuvan Urgaonkar. Dftl: a flash translation layer employing demand-based selective caching of page-level address mappings. *Acm Sigplan Notices*, 44(3):229–240, 2009.

[36] Jian Huang, Anirudh Badam, Laura Caulfield, Suman Nath, Sudipta Sengupta, Bikash Sharma, and Moinuddin K Qureshi. Flashblox: Achieving both performance isolation and uniform lifetime for virtualized ssds. In *15th USENIX Conference on File and Storage Technologies (FAST 17)*, pages 375–390, 2017.

[37] Soojun Im and Dongkun Shin. Flash-aware raid techniques for dependable and high-performance flash memory ssd. *IEEE Transactions on Computers*, 60(1):80–92, 2010.

[38] Intel. Intel® xeon® platinum 8562y+ processor. https://www.intel.com/content/www/us/en/products/sku/237558/intel-xeon-platinum-8562y-processor-60m-cache-2-80-ghz/specifications.html, 2025.

[39] Houxiang Ji, Srikar Vanavasam, Yang Zhou, Qirong Xia, Jinghan Huang, Yifan Yuan, Ren Wang, Pekon Gupta, Bhushan Chitlur, Ipoom Jeong, and Nam Sung Kim. Demystifying a cxl type-2 device: A heterogeneous cooperative computing perspective. In *2024 57th IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 1504–1517. IEEE, 2024.

[40] Sheng Jiang and Ming Liu. Building an elastic block storage over ebofs using shadow views. In *22nd USENIX Symposium on Networked Systems Design and Implementation (NSDI 25)*, pages 1137–1153, 2025.

[41] Tianyang Jiang, Guangyan Zhang, Zican Huang, Xiaosong Ma, Junyu Wei, Zhiyue Li, and Weimin Zheng. Fusionraid: Achieving consistent low latency for commodity ssd arrays. In *19th USENIX Conference on File and Storage Technologies (FAST 21)*, pages 355–370, 2021.

[42] Myoungsoo Jung. Hello bytes, bye blocks: Pcie storage meets compute express link for memory expansion (cxl-ssd). In *Proceedings of the 14th ACM Workshop*

on Hot Topics in Storage and File Systems, pages 45–51, 2022.

[43] Myoungsoo Jung, Jie Zhang, Ahmed Abulila, Miryeong Kwon, Narges Shahidi, John Shalf, Nam Sung Kim, and Mahmut Kandemir. Simplessd: Modeling solid state drives for holistic system simulation. *IEEE Computer Architecture Letters*, 17(1):37–41, 2017.

[44] Luyi Kang, Yuqi Xue, Weiwei Jia, Xiaohao Wang, Jongryool Kim, Changhwan Youn, Myeong Joon Kang, Hyung Jin Lim, Bruce Jacob, and Jian Huang. Iceclave: A trusted execution environment for in-storage computing. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 199–211, 2021.

[45] Swaroop Kavalanekar, Bruce Worthington, Qi Zhang, and Vishal Sharda. Characterization of storage workload traces from production windows servers. In *2008 IEEE International Symposium on Workload Characterization*, pages 119–128. IEEE, 2008.

[46] Aleksandr Khasymski, M Mustafa Rafique, Ali R Butt, Sudharshan S Vazhkudai, and Dimitrios S Nikolopoulos. On the use of gpus in realizing cost-effective distributed raid. In *2012 IEEE 20th International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems*, pages 469–478. IEEE, 2012.

[47] Jiho Kim, Myoungsoo Jung, and John Kim. Decoupled ssd: Rethinking ssd architecture through network-based flash controllers. In *Proceedings of the 50th Annual International Symposium on Computer Architecture*, pages 1–13, 2023.

[48] Jiho Kim, Seokwon Kang, Yongjun Park, and John Kim. Networked ssd: Flash memory interconnection network for high-bandwidth ssd. In *2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 388–403. IEEE, 2022.

[49] Sang-Hoon Kim, Jaehoon Shim, Euidong Lee, Seongyeop Jeong, Ilkueon Kang, and Jin-Soo Kim. Nvmevirt: A versatile software-defined virtual nvme device. In *21st USENIX Conference on File and Storage Technologies (FAST 23)*, pages 379–394, 2023.

[50] Kioxia. Cxl technology: High capacity, cost-effective memory in the big data era. https://blog-us.kioxia.com/post/2025/02/14/cxl-technology-high-capacity-cost-effective-memory-in-the-big-data-era, 2025.

[51] Jaewook Kwak, Sangjin Lee, Kibin Park, Jinwoo Jeong, and Yong Ho Song. Cosmos+ openssd: Rapid prototype for flash storage systems. *ACM Transactions on Storage (TOS)*, 16(3):1–35, 2020.

[52] Dongup Kwon, Junehyuk Boo, Dongryeong Kim, and Jangwoo Kim. Fvm:fpga-assisted virtual device emulation for fast, scalable, and flexible storage virtualization. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 955–971, 2020.

[53] Miryeong Kwon, Donghyun Gouk, Junhyeok Jang, Jinwoo Baek, Hyunwoo You, Sangyoon Ji, Hongjoo Jung, Junseok Moon, Seungkwan Kang, Seungjun Lee, et al. From block to byte: Transforming pcie ssds with cxl memory protocol and instruction annotation. *arXiv preprint arXiv:2506.15613*, 2025.

[54] Miryeong Kwon, Sangwon Lee, and Myoungsoo Jung. Cache in hand: Expander-driven cxl prefetcher for next generation cxl-ssd. In *Proceedings of the 15th ACM Workshop on Hot Topics in Storage and File Systems*, pages 24–30, 2023.

[55] James R Larus and Thomas Ball. Rewriting executable files to measure program behavior. *Software: Practice and Experience*, 24(2):197–218, 1994.

[56] Chunghan Lee, Tatsuo Kumano, Tatsuma Matsuki, Hiroshi Endo, Naoto Fukumoto, and Mariko Sugawara. Understanding storage traffic characteristics on enterprise virtual desktop infrastructure. In *Proceedings of the 10th ACM International Systems and Storage Conference*, pages 1–11, 2017.

[57] Philip Levis, Kun Lin, and Amy Tai. A case against cxl memory pooling. In *Proceedings of the 22nd ACM Workshop on Hot Topics in Networks*, pages 18–24, 2023.

[58] Huaicheng Li, Daniel S Berger, Lisa Hsu, Daniel Ernst, Pantea Zardoshti, Stanko Novakovic, Monish Shah, Samir Rajadnya, Scott Lee, Ishwar Agarwal, Mark D. Hill, Marcus Fontoura, and Ricardo Bianchini. Pond: Cxl-based memory pooling systems for cloud platforms. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, pages 574–587, 2023.

[59] Shaobo Li, Yirui Zhou, Hao Ren, and Jian Huang. Bytefs: System support for (cxl-based) memory-semantic solid-state drives. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1*, pages 116–132, 2025.

[60] Sheng Li, Jung Ho Ahn, Richard D Strong, Jay B Brockman, Dean M Tullsen, and Norman P Jouppi. Mcpat: An integrated power, area, and timing modeling framework for multicore and manycore architectures. In *Proceedings of the 42nd annual ieee/acm international symposium on microarchitecture*, pages 469–480, 2009.

[61] Changyue Liao, Mo Sun, Zihan Yang, Kaiqi Chen, Binhang Yuan, Fei Wu, and Zeke Wang. Adding nvme ssds to enable and accelerate 100b model fine-tuning on a single gpu. *arXiv preprint arXiv:2403.06504*, 2024.

[62] Linux. Linux v5.15. https://github.com/torvalds/linux/tree/v5.15, 2021.

[63] Linux. Lightnvm driver. https://github.com/torvalds/linux/tree/v5.10-rc3/drivers/lightnvm, 2025.

[64] Linux. Memory management. https://docs.kernel.org/admin-guide/mm/index.html, 2025.

[65] Longsys. Application documents for initial public offering and listing on gem of shenzhen jiang bolong electronics co. initial public offering of shares and listing on gem board application documents report in response to the audit inquiry letter from. https://qccdata.qichacha.com/ReportData/PDF/ef39602f078481d5136f3c4253bc56b5.pdf, 2021.

[66] Sunilkumar S Manvi and Gopal Krishna Shyam. Resource management for infrastructure as a service (iaas) in cloud computing: A survey. *Journal of network and computer applications*, 41:424–440, 2014.

[67] China Flash Market. The price of nand flash and lpddr. https://en.chinaflashmarket.com/, 2024.

[68] Avantika Mathur, Mingming Cao, Suparna Bhattacharya, Andreas Dilger, Alex Tomas, and Laurent Vivier. The new ext4 filesystem: current status and future plans. In *Proceedings of the Linux symposium*, volume 2, pages 21–33. Citeseer, 2007.

[69] Sara McAllister, Benjamin Berg, Julian Tutuncu-Macias, Juncheng Yang, Sathya Gunasekar, Jimmy Lu, Daniel S Berger, Nathan Beckmann, and Gregory R Ganger. Kangaroo: Caching billions of tiny objects on flash. In *Proceedings of the ACM SIGOPS 28th symposium on operating systems principles*, pages 243–262, 2021.

[70] Rino Micheloni, Alessia Marelli, Kam Eshghi, and G Wong. Ssd market overview. *Inside Solid State Drives (SSDs)*, pages 1–17, 2013.

[71] Micron. Micron 9550 nvme ssd. https://www.micron.com/products/storage/ssd/data-center-ssd/9550-ssd, 2025.

[72] Micron. Micron cz122. https://www.micron.com/products/memory/cxl-memory?srsltid=AfmBOoo9x83Q9TQMQpUPdyrUgA3gVUJFi2Vp-aj6bKzENgiggYgZXFtc, 2025.

[73] Micron. Micron leads ecosystem: first to develop pcie gen6 data center ssd. https://www.micron.com/about/blog/storage/ssd/micron-leads-ecosystem-first-to-develop-pcie-gen6-data-center-ssd, 2025.

[74] Micron. Micron mt40a512m16td-062e. https://www.mouser.com/ProductDetail/Micron/MT40A512M16TD-062E-AITR?qs=3Rah4i%252BhyCENjNab2Szuaw%3D%3D, 2025.

[75] Jaehong Min, Ming Liu, Tapan Chugh, Chenxingyu Zhao, Andrew Wei, In Hwan Doh, and Arvind Krishnamurthy. Gimbal: enabling multi-tenant storage disaggregation on smartnic jbofs. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*, pages 106–122, 2021.

[76] Rakesh Nadig, Mohammad Sadrosadati, Haiyu Mao, Nika Mansouri Ghiasi, Arash Tavakkol, Jisung Park, Hamid Sarbazi-Azad, Juan Gómez Luna, and Onur Mutlu. Venice: Improving solid-state drive parallelism at low cost via conflict-free accesses. In *Proceedings of the 50th Annual International Symposium on Computer Architecture*, pages 1–16, 2023.

[77] Iyswarya Narayanan, Di Wang, Myeongjae Jeon, Bikash Sharma, Laura Caulfield, Anand Sivasubramaniam, Ben Cutler, Jie Liu, Badriddine Khessib, and Kushagra Vaid. Ssd failures in datacenters: What? when? and why? In *Proceedings of the 9th ACM International on Systems and Storage Conference*, pages 1–11, 2016.

[78] Nvidia. Nvidia bluefield-3 dpu. https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/documents/datasheet-nvidia-bluefield-3-dpu.pdf, 2025.

[79] NVMe. Nvm command set specification. https://nvmexpress.org/wp-content/uploads/NVM-Express-NVM-Command-Set-Specification-1.0c-2022.10.03-Ratified.pdf, 2022.

[80] NVMe. Nvm express® base specification. https://nvmexpress.org/specification/nvm-express-base-specification/, 2025.

[81] NVMe. Nvm express® (nvme®) technology support for cxl®: Unleashing computational workloads. https://nvmexpress.org/nvm-express-nvme-technology-support-for-cxl-unleashing-computational-workloads/, 2025.

[82] ONFi. Open nand flash interface specifications. https://onfi.org/specs.html, 2025.

[83] Li Peng, Wenbo Wu, Shushu Yi, Xianzhang Chen, Chenxi Wang, Shengwen Liang, Zhe Wang, Nong Xiao, Qiao Li, Mingzhe Zhang, and Jie Zhang. Xharvest: Rethinking high-performance and cost-efficient ssd architecture with cxl-driven harvesting. In *Proceedings of the 52nd Annual International Symposium on Computer Architecture*, pages 434–449, 2025.

[84] Phison. Ps5021-e21t. https://www.phison.com/en/products/ssd/ps5021-e21t, 2025.

[85] Phison. Ps5026-e26. https://www.phison.com/en/products/ssd/ps5026-e26, 2025.

[86] Benjamin Reidys, Jinghan Sun, Anirudh Badam, Shadi Noghabi, and Jian Huang. Blockflex: Enabling storage harvesting with software-defined flash in modern cloud platforms. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 17–33, 2022.

[87] Benjamin Reidys, Yuqi Xue, Daixuan Li, Bharat Sukhwani, Wen-Mei Hwu, Deming Chen, Sameh Asaad, and Jian Huang. Rackblox: A software-defined rack-scale storage system with network-storage co-design. In *Proceedings of the 29th Symposium on Operating Systems Principles*, pages 182–199, 2023.

[88] Rusty Russell. virtio: towards a de-facto standard for virtual i/o devices. *ACM SIGOPS Operating Systems Review*, 42(5):95–103, 2008.

[89] Samsung. Power loss protection (plp) protect your data against sudden power loss. https://download.semiconductor.samsung.com/resources/others/Samsung_SSD_845DC_05_Power_loss_protection_PLP.pdf, 2014.

[90] Samsung. Samsung 980pro nvme ssd. https://www.samsung.com/us/computing/memory-storage/solid-state-drives/980-pro-pcie-4-0-nvme-ssd-1tb-mz-v8p1t0b-am/, 2020.

[91] Samsung. Samsung pm1743. https://semiconductor.samsung.com/ssd/enterprise-ssd/pm1743/, 2023.

[92] Samsung. Am9c1. https://news.samsung.com/global/samsung-electronics-develops-industrys-first-automotive-ssd-based-on-8th-generation-v-nand, 2025.

[93] Samsung. Detachable ssd, am9c1 e1.a. https://www.ces.tech/ces-innovation-awards/2026/detachable-ssd-am9c1-e1a/, 2025.

[94] Henry N Schuh, Arvind Krishnamurthy, David Culler, Henry M Levy, Luigi Rizzo, Samira Khan, and Brent E Stephens. Cc-nic: a cache-coherent interface to the nic. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1*, pages 52–68, 2024.

[95] Kia Shakiba, Sari Sultan, and Michael Stumm. Kosmo: efficient online miss ratio curve generation for eviction policy evaluation. In *22nd USENIX Conference on File and Storage Technologies (FAST 24)*, pages 89–105, 2024.

[96] Yizhou Shan, Yutong Huang, Yilun Chen, and Yiying Zhang. Legoos: A disseminated, distributed os for hardware resource disaggregation. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 69–87, 2018.

[97] Xuanhua Shi, Ming Li, Wei Liu, Hai Jin, Chen Yu, and Yong Chen. Ssdup: a traffic-aware ssd burst buffer for hpc systems. In *Proceedings of the international conference on supercomputing*, pages 1–10, 2017.

[98] John E. Shore. On the external storage fragmentation produced by first-fit and best-fit allocation strategies. *Commun. ACM*, 18(8):433–440, August 1975.

[99] Junyi Shu, Kun Qian, Ennan Zhai, Xuanzhe Liu, and Xin Jin. Burstable cloud block storage with data processing units. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, pages 783–799, 2024.

[100] SimpleSSD. Simplessd version 2.0: Open-source licensed educational ssd simulator for high-performance storage and full-system evaluations. https://github.com/SimpleSSD/SimpleSSD, 2020.

[101] Ryan Smith. Ssd calculator soruces. https://www.soothsawyer.com/wp-content/uploads/2020/03/Public_SSD_Cost_Calculator_Share.pdf, 2020.

[102] SNIA. Msr cambridge traces. http://iotta.snia.org/traces/block-io/388, 2025.

[103] Mohammadreza Soltaniyeh, Gongjin Sun, Xuebin Yao, Amir Beygi, Ramdas Kachare, Dongwan Zhao, Hingkwan Huen, Andrew Chang, Senthil Murugesapandian, and Caroline Kahn. Revisiting memory hierarchies with cmm-h: Use device-side caching to integrate dram and ssd for a hybrid cxl memory. In *Proceedings of the 17th ACM Workshop on Hot Topics in Storage and File Systems*, pages 45–51, 2025.

[104] Yong Ho Song, Sanghyuk Jung, Sang-Won Lee, and Jin-Soo Kim. Cosmos openssd: A pcie-based open source ssd platform. *Proc. Flash Memory Summit*, pages 1–30, 2014.

[105] Pure Storage. What is directflash and how does it work? https://www.purestorage.com/knowledge/what-is-directflash-and-how-does-it-work.html, 2025.

[106] Kang-Deog Suh, Byung-Hoon Suh, Young-Ho Lim, Jin-Ki Kim, Young-Joon Choi, Yong-Nam Koh, Sung-Soo Lee, Suk-Chon Kwon, Byung-Soon Choi, Jin-Sun Yum, Jung-Hyuk Choi, Jang-Rae Kim, and Hyung-Kyu Lim. A 3.3 v 32 mb nand flash memory with incremental step pulse programming scheme. *IEEE Journal of Solid-State Circuits*, 30(11):1149–1156, 1995.

[107] Jinghan Sun, Benjamin Reidys, Daixuan Li, Jichuan Chang, Marc Snir, and Jian Huang. Fleetio: Managing multi-tenant cloud storage with multi-agent reinforcement learning. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1*, pages 478–492, 2025.

[108] Xun Sun, Mingxing Zhang, Yingdi Shan, Kang Chen, Jinlei Jiang, and Yongwei Wu. Scalio: Scaling up dpu-basedjbof key-value store with nvme-of target offload. In *19th USENIX Symposium on Operating Systems Design and Implementation (OSDI 25)*, pages 449–464, 2025.

[109] Yan Sun, Yifan Yuan, Zeduo Yu, Reese Kuper, Chihun Song, Jinghan Huang, Houxiang Ji, Siddharth Agarwal, Jiaqi Lou, Ipoom Jeong, Ren Wang, Jung Ho Ahn, Tianyin Xu, and Nam Sung Kim. Demystifying cxl memory with genuine cxl-ready systems and devices. In *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 105–121, 2023.

[110] SuperMicro. Storage superserver ssg-229j-5bu24jbf. https://www.supermicro.com/en/products/system/storage/2u/ssg-229j-5bu24jbf, 2025.

[111] Yupeng Tang, Ping Zhou, Wenhui Zhang, Henry Hu, Qirui Yang, Hao Xiang, Tongping Liu, Jiaxin Shan, Ruoyun Huang, Cheng Zhao, Cheng Chen, Hui Zhang, Fei Liu, Shuai Zhang, Xiaoning Ding, and Jianjun Chen. Exploring performance and cost optimization with asic-based cxl memory. In *Proceedings of the Nineteenth European Conference on Computer Systems*, pages 818–833, 2024.

[112] CRZ Technology. Daisyplus openssd. https://www.crz-tech.com/crz/article/DaisyPlus/, 2025.

[113] Maxio Technology. About lianyun technology (hangzhou) co. initial public offering and listing on technology and innovation board (technology) response to the audit inquiry letter on the filing documents. https://static.sse.com.cn/stock/disclosure/announcement/c/202405/001363_20240523_AP3M.pdf, 2024.

[114] TRENDFORCE. Nand flash price trends. https://www.trendforce.com/price/flash, 2024.

[115] Shivani Tripathy and Manoranjan Satpathy. Ssd internal cache management policies: A survey. *Journal of Systems Architecture*, 122:102334, 2022.

[116] Hung-Wei Tseng, Laura Grupp, and Steven Swanson. Understanding the impact of power loss on flash memory. In *Proceedings of the 48th Design Automation Conference*, pages 35–40, 2011.

[117] Carl A Waldspurger, Nohhyun Park, Alexander Garthwaite, and Irfan Ahmad. Efficient mrc construction with shards. In *13th USENIX Conference on File and Storage Technologies (FAST 15)*, pages 95–110, 2015.

[118] Rui Wang, Yongkun Li, Hong Xie, Yinlong Xu, and John CS Lui. Graphwalker: An i/o-efficient and resource-friendly graph analytic system for fast and scalable random walks. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pages 559–571, 2020.

[119] Yawen Wang, Kapil Arya, Marios Kogias, Manohar Vanga, Aditya Bhandari, Neeraja J Yadwadkar, Siddhartha Sen, Sameh Elnikety, Christos Kozyrakis, and Ricardo Bianchini. Smartharvest: Harvesting idle cpus safely and efficiently in the cloud. In *Proceedings of the Sixteenth European Conference on Computer Systems*, pages 1–16, 2021.

[120] Reinhold P Weicker. Dhrystone: a synthetic systems programming benchmark. *Communications of the ACM*, 27(10):1013–1030, 1984.

[121] Jiwon Woo, Minwoo Ahn, Gyusun Lee, and Jinkyu Jeong. D2fq: Device-direct fair queueing for nvmessds. In *19th USENIX Conference on File and Storage Technologies (FAST 21)*, pages 403–415, 2021.

[122] Gala Yadgar, MOSHE Gabel, Shehbaz Jaffer, and Bianca Schroeder. Ssd-based workload characteristics and their performance implications. *ACM Transactions on Storage (TOS)*, 17(1):1–26, 2021.

[123] Shiqin Yan, Huaicheng Li, Mingzhe Hao, Michael Hao Tong, Swaminathan Sundararaman, Andrew A Chien, and Haryadi S Gunawi. Tiny-tail flash: Near-perfect elimination of garbage collection tail latencies in nand ssds. *ACM Transactions on Storage (TOS)*, 13(3):1–26, 2017.

[124] Shao-Peng Yang, Minjae Kim, Sanghyun Nam, Juhyung Park, Jin-Yong Choi, Eyee Hyun Nam, Eunji Lee, Sungjin Lee, and Bryan S Kim. Overcoming the memory wall with cxl-enabledssds. In *2023 USENIX Annual Technical Conference (USENIX ATC 23)*, pages 601–617, 2023.

[125] Zhe Yang, Youyou Lu, Xiaojian Liao, Youmin Chen, Junru Li, Siyu He, and Jiwu Shu. λ-io: A unified io stack for computational storage. In *21st USENIX Conference on File and Storage Technologies (FAST 23)*, pages 347–362, 2023.

[126] Ziye Yang, Changpeng Liu, Yanbo Zhou, Xiaodong Liu, and Gang Cao. Spdk vhost-nvme: Accelerating i/os in virtual machines on nvme ssds via user space vhost target. In *2018 IEEE 8th International Symposium on Cloud and Service Computing (SC2)*, pages 67–76. IEEE, 2018.

[127] Min Ye, Qiao Li, Yina Lv, Jie Zhang, Tianyu Ren, Daniel Wen, Tei-Wei Kuo, and Chun Jason Xue. Achieving near-zero read retry for 3d nand flash memory. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, pages 55–70, 2024.

[128] Shushu Yi, Xiurui Pan, Qiao Li, Qiang Li, Chenxi Wang, Bo Mao, Myoungsoo Jung, and Jie Zhang. ScalaAFA: Constructing User-Space All-Flash array engine with holistic designs. In *2024 USENIX Annual Technical Conference (USENIX ATC 24)*, pages 141–156, Santa Clara, CA, July 2024. USENIX Association.

[129] Shushu Yi, Yanning Yang, Yunxiao Tang, Zixuan Zhou, Junzhe Li, Chen Yue, Myoungsoo Jung, and Jie Zhang. Scalaraid: Optimizing linux software raid system for next-generation storage. In *Proceedings of the 14th ACM Workshop on Hot Topics in Storage and File Systems*, pages 119–125, 2022.

[130] Sangjin Yoo and Dongkun Shin. Reinforcement learning-basedslc cache technique for enhancing ssd write performance. In *12th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 20)*, 2020.

[131] Haoyang Zhang, Yuqi Xue, Yirui Eric Zhou, Shaobo Li, and Jian Huang. Skybyte: Architecting an efficient memory-semantic cxl-based ssd with os and hardware co-design. *arXiv preprint arXiv:2501.10682*, 2025.

[132] Jian Zhang, Yujie Ren, Marie Nguyen, Changwoo Min, and Sudarsun Kannan. Omnicache: Collaborative caching for near-storage accelerators. In *22nd USENIX Conference on File and Storage Technologies (FAST 24)*, pages 35–50, 2024.

[133] Jie Zhang, Miryeong Kwon, Michael Swift, and Myoungsoo Jung. Scalable parallel flash firmware for many-core architectures. In *18th USENIX Conference on File and Storage Technologies (FAST 20)*, pages 121–136, 2020.

[134] Yu Zhang, Ping Huang, Ke Zhou, Hua Wang, Jianying Hu, Yongguang Ji, and Bin Cheng. Osca: An online-model based cache allocation scheme in cloud block storage systems. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pages 785–798, 2020.

[135] Mai Zheng, Joseph Tucek, Feng Qin, and Mark Lillibridge. Understanding the robustness of ssds under power fault. In *11th USENIX Conference on File and Storage Technologies (FAST 13)*, pages 271–284, 2013.

[136] Yuhong Zhong, Daniel S Berger, Pantea Zardoshti, Enrique Saurez, Jacob Nelson, Dan RK Ports, Antonis Psistakis, Joshua Fried, and Asaf Cidon. Oasis: Pooling pcie devices over cxl to boost utilization. In *Proceedings of the ACM SIGOPS 31st Symposium on Operating Systems Principles*, pages 101–119, 2025.

[137] Yuhong Zhong, Fiodar Kazhamiaka, Pantea Zardoshti, Shuwei Teng, Rodrigo Fonseca, Mark D Hill, and Daniel S Berger. Octopus: Enhancing {CXL} memory pods via sparse topology. In *23rd USENIX Symposium on Networked Systems Design and Implementation (NSDI 26)*, pages 1303–1322, 2026.

[138] You Zhou, Qiulin Wu, Fei Wu, Hong Jiang, Jian Zhou, and Changsheng Xie. {Remap-SSD}: Safely and efficiently exploiting {SSD} address remapping to eliminate duplicate writes. In *19th USENIX Conference on File and Storage Technologies (FAST 21)*, pages 187–202, 2021.

ARTIFACT EVALUATED
USENIX
AVAILABLE

FORGE: Mitigating Synchronization Amplification for Memory-Disaggregated Caching Systems

Zhijun Yang, Yu Hua*, Ming Zhang, Menglei Chen, Yixiao Wang
Huazhong University of Science and Technology
Corresponding Author: Yu Hua (csyhua@hust.edu.cn)

Abstract

Disaggregated Memory (DM) architectures offer caching systems the potential for elastic scaling and improved resource utilization by decoupling compute and memory. However, this advantage is undermined by costly cross-node synchronization, which exacerbates the overheads of critical cache operations, including hotness tracking, eviction coordination, and memory defragmentation. To address this challenge, we present FORGE, a caching system tailored for DM that prioritizes synchronization efficiency. FORGE groups cached objects based on similarity and performs group-level synchronizations to amortize overheads. It evicts cold groups via a contention-free and hotness-aware FIFO queue, efficiently sustaining high hit ratios while mitigating memory fragmentation. Leveraging the predictability of FIFO evictions, FORGE adopts a lazy synchronization strategy that updates hotness metrics just-in-time for eviction and offloads this process to on-chip memory in RDMA NICs for acceleration. Extensive evaluations on YCSB and real-world workloads demonstrate that FORGE achieves up to 4.5× higher throughput, 4.0×/7.5× lower P50/P99 latency, and an average of 1.14× higher cache hit ratio compared with state-of-the-art systems.

1 Introduction

In-memory caching systems like Memcached [3] and Redis [4] are fundamental infrastructure in cloud datacenters [6, 17, 18, 54, 75]. They accelerate data access for various applications, including web services [14], machine learning [55], and serverless computing [37]. These applications often exhibit highly dynamic and bursty workloads [58, 60, 65, 72], making it crucial for caching systems to elastically scale resources to ensure both high performance and utilization. However, traditional caches run on monolithic servers where compute and memory resources are tightly coupled, preventing independent scaling of compute and memory and often leading to resource overprovisioning and underutilization.

To improve resource utilization and scalability, Disaggregated Memory (DM) architectures [42, 61] decouple compute and memory from monolithic servers into separated pools. Compute Nodes (CNs) in compute pools provide abundant CPU cores, while Memory Nodes (MNs) in memory pools offer large memory capacity. CNs access remote memory on MNs via high-speed interconnects like Compute Express Link (CXL) [1] and Remote Direct Memory Access (RDMA) [2], bypassing limited compute units on MNs. This allows caching systems to independently scale compute (via CNs) for high throughput and memory (via MNs) for large capacity, aligning resources more closely with application demands.

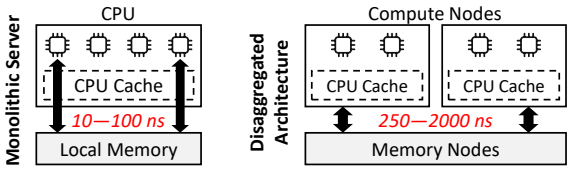


Figure 1: Memory access latencies in different architectures.

Despite the advantages of DM, caching systems face a critical new challenge: **synchronization amplification**. While prior work has extensively optimized key-value indexing for DM [43, 45, 46, 63, 71, 85], the efficiency of essential housekeeping tasks, including hotness tracking, eviction, and garbage collection, remains largely unaddressed. In monolithic servers, these tasks are lightweight, leveraging low-latency (10–100 ns), cache-coherent CPU load/store to access compact metadata. However, in disaggregated architectures (Fig. 1), where MNs lack computational capability [42, 43, 45, 62, 79], housekeeping must be offloaded to CNs. This shift introduces **cross-node synchronizations** over high-latency interconnects like CXL (∼350 ns [78]) or RDMA (∼2,000 ns [34]), incurring delays at least 20× higher than monolithic setups. Consequently, the housekeeping synchronization overheads become severe performance bottlenecks, directly interfering with latency-sensitive `Get`/`Set` operations. To fully exploit DM's potential, caching systems need to rethink how to efficiently coordinate housekeeping synchronizations across nodes in DM by overcoming three key challenges:

1) Efficiency–Effectiveness Trade-off in Eviction. While evicting groups of cold objects appears promising for amortizing synchronization costs, its efficacy heavily depends on grouping strategies and workload characteristics. In skewed workloads [7, 12, 13, 75], groups often contain a mix of hot and cold objects, resulting in aggregated group-level hotness metrics that obscure fine-grained object popularity. This can mislead CNs into retaining cold objects grouped with hot ones while evicting uniformly warm groups, degrading cache hit ratios and undermining benefits of batched eviction.

2) Accuracy–Overhead Dilemma in Hotness Tracking. Group-level hotness tracking mitigates synchronization overhead by aggregating metadata updates, but leads to misleading coarse-grained metrics under skew. On the other hand, object-level tracking ensures accuracy but demands frequent fine-grained updates across distributed CNs, reintroducing prohibitive synchronization costs. While First-In-First-Out (FIFO) replacement avoids hotness tracking altogether by enforcing a static eviction order, it ignores access patterns and often results in suboptimal hit ratios [20, 77].

3) Timeliness–Cost Conflict in Synchronization. For hotness-aware eviction to be effective, MNs need to hold up-to-date hotness metrics that accurately reflect dynamic access patterns. In DM, achieving this timeliness demands frequent synchronizations from distributed CNs, incurring substantial network overhead. While delaying synchronizations for batching can amortize costs, it introduces staleness in hotness metrics, causing misidentification of hot/cold objects and ultimately degrading cache performance.

As a result, these fundamental trade-offs leave caching systems struggling to balance management granularity, synchronization efficiency, and eviction effectiveness, remaining open and critical problems in DM environments.

To address these challenges, we present FORGE, a **F**IF**O**-based g**r**oup-level caching system on disa**g**gregated m**e**mory. By organizing objects into groups sharing similar access patterns, FORGE coalesces the accuracy of object-level tracking with the efficiency of group-level coordination. Specifically, CNs execute `Get`/`Set` operations and track fine-grained object hotness to ensure precision, while synchronizing metrics to MNs only in coarse-grained groups to amortize network overheads. Leveraging these fine-grained metrics during group eviction, a lightweight regrouping scheme extracts and consolidates hot objects while filtering out cold ones, ensuring high adaptivity. To enable scalable eviction, FORGE transforms the standard FIFO policy into a contention-free, hotness-aware mechanism via ring-array-based virtual segmentation. By mapping groups to distinct virtual segments based on their hotness, FORGE accelerates the eviction of cold groups while preserving hot ones, effectively improving hit ratios without the synchronization bottlenecks of traditional priority queues.

Our co-design of group-level caching and FIFO replacement introduces a novel lazy synchronization scheme tailored for DM. We observe that *synchronized object hotness metrics are not needed immediately upon cache hits but can be deferred until predictable group evictions, triggered as groups approach the head of the FIFO queue*. CNs monitor FIFO queue metadata to identify groups nearing eviction and initiate just-in-time hotness flushing, ensuring accuracy while minimizing synchronization frequency. Furthermore, despite the large cache capacity of MNs, the number of objects evicted concurrently remains small, enabling CNs to efficiently leverage the fast on-chip memory of RDMA NICs (RNICs) in MNs as synchronization destinations. This strategy significantly reduces synchronization overhead and mitigates interference with performance-critical `Get`/`Set` operations, balancing timeliness, efficiency, and effectiveness.

In summary, this paper makes the following contributions:

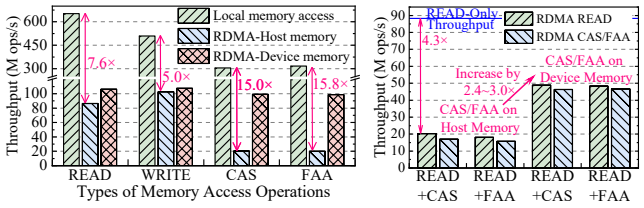
- We identify and analyze the critical challenge of synchronization amplification in DM caching systems, where housekeeping synchronizations become a major bottleneck.
- To address this challenge, FORGE introduces three key innovations: 1) DM-tailored group-level management to amortize synchronization overheads; 2) fine-grained object hotness tracking combined with hotness-aware FIFO queue to enhance hit ratios despite group-level eviction; 3) a lazy synchronization mechanism that exploits predictable eviction timing to defer and batch hotness updates, accelerated by RNIC on-chip memory, ensuring both efficiency and timeliness.
- We implement FORGE and perform extensive experiments that show the performance advantages of FORGE over state-of-the-art systems. The source code of FORGE is publicly available at https://github.com/cszjyang/forge.

2 Background and Motivation

2.1 In-Memory Caching Systems

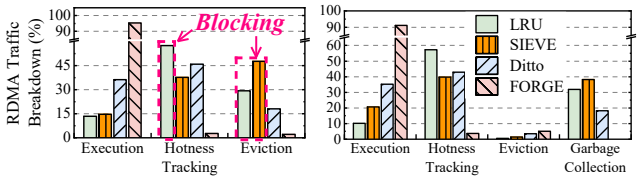
In-memory caching systems store frequently accessed (hot) objects in fast but limited-size memory and serve `Get`/`Set` requests from applications, reducing access latency and offloading backend storage. When the cache is full, replacement algorithms evict less accessed (cold) objects to make room for new ones, maximizing the cache's utility [20, 38].

Traditional caching systems scale at the granularity of instances on monolithic servers, where compute and memory resources are tightly coupled. This rigid bundling prevents independent scaling of individual resources and often leads to overprovisioning and underutilization. For example, during peak demand, the system launches additional instances to handle the increased load. The new instances bring both compute and memory, even if only one resource type is the bottleneck, leaving the other resource underutilized. Moreover, instance-based scaling often requires expensive data repartitioning and cache warming [5, 21, 23], discouraging frequent resizing.



(a) Performance: Local memory access vs. RDMA in Host/Device memory (b) Performance interference between RDMA verbs

Figure 2: Performance characteristics of RDMA-based DM.



(a) YCSB with a 10% cache size (b) Twitter with a 10% cache size

Figure 3: Traffic breakdowns of conventional queue-based (LRU [20] and SIEVE [82]), sampling-based (Ditto [62]) and our (FORGE) caching systems.

2.2 Adapting Caching Systems to DM

To improve resource utilization in datacenters, Disaggregated Memory (DM) architectures [42, 61] deconstruct monolithic servers by decoupling compute and memory into distinct resource pools. Compute Nodes (CNs) offer abundant CPU cores and limited local memory for temporary buffering, while Memory Nodes (MNs) provide large memory capacity but limited compute power for control-plane operations (e.g., connection management). These pools are interconnected via high-speed links such as CXL [1] and RDMA [2], enabling CNs to directly access remote memory on MNs without involving the MNs' weak compute units. This decoupled design allows independent scaling of compute and memory by adding CNs or MNs. Moreover, since MNs are shared across all CNs, scaling does not necessitate data repartitioning, simplifying resource management and load balancing.

The demand for elastic scaling makes caching systems ideal for DM. CNs handle compute-heavy tasks including `Get`/`Set` requests, hotness tracking, and eviction, while MNs store the cached objects and hotness metrics such as access timestamps (for LRU [20]) and frequency counters (for LFU [38]). The systems independently scale the CNs and MNs on demand, improving performance and resource efficiency.

However, DM introduces fundamental performance challenges. As shown in Fig. 2a, RDMA throughputs are lower than local memory by 5.0–15.8×. Atomic `RDMA_CAS/FAA` (compare-and-swap/fetch-and-add) operations are particularly costly due to RNIC serialization: they incur 4–5× higher overhead than basic reads/writes and degrade concurrent `RDMA_READ` throughput by 4.3× due to interference (Fig. 2b). Although using RNIC on-chip memory as the `RDMA_CAS/FAA` destination can shorten the serialization critical path, improving atomic throughput by 4.9× and reducing interference by 2.4–3.0×, this memory's scarce capacity (e.g., 256 KB in a Mellanox ConnectX-5 RNIC) limits broader applicability.

These hardware constraints pose a critical challenge for caching systems, which continuously perform housekeeping tasks (hotness tracking, eviction, and garbage collection) to maintain high hit ratios. In DM, ensuring coherence and correctness for these tasks necessitates frequent cross-node synchronization. This requirement transforms housekeeping from lightweight local operations into heavy network burdens that interfere with performance-critical `Get`/`Set` operations, severely undermining overall system efficiency.

Hotness Tracking and Eviction. Existing replacement algorithms broadly fall into two main categories: queue-based and sampling-based. Traditional queue-based schemes (e.g., LRU [20], LFU [38]) maintain a global priority queue (often a linked list). On cache hits, CNs promote accessed objects within the queue; for eviction, CNs remove cold objects at the queue tail. However, such queue manipulations under exclusive locks necessitate a sequence of `RDMA_CAS`, `RDMA_READ`, and `RDMA_WRITE`. For consistency, only one CN can update the queue at a time, forcing other CNs to block and retry under contention. Exacerbated by DM's high latency [44, 84], such blocking severely degrades concurrency and performance.

Recent algorithms for monolithic servers, like SIEVE [82], mitigate the hit-path overhead of traditional queue-based schemes. Specifically, on every cache hit, a thread checks and sets the object's visited bit using an atomic operation (e.g., `RDMA_CAS` in DM), bypassing the overheads of promoting the object within the queue. However, for eviction, a thread still needs to lock the queue and move a global pointer to sequentially scan the linked objects, reading and clearing their visited bits until an unvisited cold object is found. Although fast in a monolithic server, this scanning is expensive in DM, since it triggers serialized RDMA operations with multiple network round trips while holding the lock, severely inflating eviction latency and blocking other CNs' concurrent evictions.

To quantify these synchronization overheads, we evaluate LRU and SIEVE on RDMA-based DM using YCSB [19] and real-world Twitter traces [75], with 16 threads on CNs and a 10% cache ratio on MNs. The experimental setup is detailed in § 6. To reduce blocking contention, we partition the queues of both LRU and SIEVE into 32 shards, allowing different CNs to simultaneously operate on different shards. Despite this synchronization optimization, the RDMA traffic in LRU remains overwhelmingly dominated by hotness tracking (57.4%) and eviction (29.2%), vastly exceeding the mere 13.4% consumed by actual `Get`/`Set` operations, as shown in Fig. 3a. While SIEVE reduces the hotness tracking traffic (37.7%), it shifts the synchronization bottleneck to the blocking eviction path (47.6%). These findings demonstrate

that conventional queue-based policies, though effective in monolithic systems, incur synchronization overheads that are severely amplified into major performance bottlenecks in DM.

To avoid the costly blocking overhead of global queues, state-of-the-art DM systems like Ditto [62] employ sampling-based algorithms [10, 56]. On cache hits, CNs locally accumulate hotness metrics and periodically flush them in batches to MNs using `RDMA_WRITE` (for access timestamps) and `RDMA_FAA` (for frequency counters). To select eviction candidates, CNs read a random sample of objects via `RDMA_READ`, compare their (potentially stale) hotness metrics, and evict the coldest ones using `RDMA_CAS` to ensure consistency [62, 81]. Consequently, Ditto eliminates blocking among CNs during synchronization for hotness tracking and eviction, achieving 3.1–5.0× higher throughput than SIEVE in DM.

However, these benefits come with two critical trade-offs. First, while the periodic and batched hotness flushing reduces network traffic, it creates a dilemma: short flushing intervals maintain metric accuracy but limit traffic reduction, whereas long intervals reduce traffic but risk stale metrics leading to erroneous eviction of hot objects. Second, the sampling-based eviction introduces another trade-off: limited sampling may overlook colder objects and reduce hit ratios, while extensive sampling increases `RDMA_READ` traffic and overhead. Consequently, these mechanisms struggle to balance synchronization efficiency, metric freshness, and eviction effectiveness. In Fig. 3a, our evaluation of Ditto reveals that hotness tracking and eviction still consume 45.9% and 18.0% of total RDMA traffic, respectively, exceeding the 36.1% for `Get`/`Set` executions. These synchronization overheads significantly hinder the scalability and efficiency of state-of-the-art DM systems.

Garbage Collection (GC). In dynamic workloads, object size distributions fluctuate over time [75]. The frequent insertion and eviction of variably-sized objects lead to memory fragmentation, where free space becomes divided into small, non-contiguous fragments dispersed across multiple CNs, making each CN struggle to allocate contiguous space for large objects. Consequently, fragmentation diminishes effective memory utilization and cache hit ratios. To mitigate this, caching systems leverage GC [57] for defragmentation. However, GC in DM necessitates costly cross-node coordination using `RDMA_READ` and `RDMA_CAS` to track and reclaim fragments. As shown in Fig. 3b, while GC mitigates eviction costs by reclaiming space for new objects, it consumes 31.9%, 38.2%, and 18.3% of traffic respectively in LRU, SIEVE, and Ditto. In contrast, for YCSB workloads with uniform object sizes, GC is unnecessary. These results highlight the persistent tension between fragmentation management and synchronization efficiency in DM caching systems.

2.3 Towards Efficient Synchronization

To overcome synchronization bottlenecks, caching systems in DM need to rethink the coordination of housekeeping tasks across distributed CNs. A promising approach involves organizing cached objects into groups, enabling group-level hotness tracking and eviction to amortize synchronization overheads across multiple objects. Moreover, fixed-size group allocation can mitigate memory fragmentation [36]. However, despite these potential benefits, critical challenges remain in realizing practical group-level caching for DM.

C1: Efficiency–Effectiveness Trade-off in Eviction. Despite efficiency, coarse-grained group eviction can be ineffective at retaining hot objects and evicting cold ones. For example, in skewed access patterns [41, 83], aggregated group-level hotness metrics (e.g., access timestamps and frequency counts) often mislead CNs to mistakenly retain skewed groups containing both hot and cold objects, while evicting more uniformly warm groups, thereby reducing hit ratios.

C2: Accuracy–Overhead Dilemma in Hotness Tracking. To avoid inaccurate group-level hotness metrics, caching systems employ accurate object-level tracking. But this requires fine-grained updates of hotness metrics from distributed CNs, reintroducing prohibitive synchronization overheads. Another method is to evict groups in FIFO order, thus avoiding expensive hotness tracking. However, FIFO eviction is unaware of object hotness and often yields low hit ratios [20, 77].

C3: Timeliness–Cost Conflict in Synchronization. For hotness-aware eviction to be effective, the hotness metrics on MNs need to be timely, which requires frequent costly synchronizations from distributed CNs, as analyzed in § 2.2. Delaying and batching synchronization reduces cost but results in stale metrics, leading to misidentification of hot objects.

3 FORGE Overview

To address these challenges, FORGE coalesces the strengths of group-level caching and FIFO eviction, and introduces novel DM-friendly designs. Fig. 4 shows the overview of FORGE. To serve application `Get`/`Set` requests, CNs access and manage the cached objects on MNs via one-sided RDMA. The MN hash table indexes the cached objects, which are organized into groups based on access patterns (§ 4.1). FORGE decouples operation granularity: performance-critical `Get`/`Set` requests operate on individual objects to avoid read/write amplification, while housekeeping tasks are performed at the group level to amortize synchronization overheads.

For a `Set` request, a CN inserts the object into an allocated group and updates the hash table (§ 4.2). Each group occupies a fixed-size memory chunk to prevent external fragmentation. When a group becomes full (i.e., reaching chunk capacity or a configurable maximum object number such as 256 [74, 76]), the CN pushes the group pointer into the tail of the contention-free FIFO queue (§ 5.1) on an MN. Upon exhausting groups with available space for insertion, the CN dequeues and evicts cold groups from the queue head to reclaim their space for new allocations. This group-level eviction removes overheads from the critical path of latency-sensitive `Get`/`Set` operations,

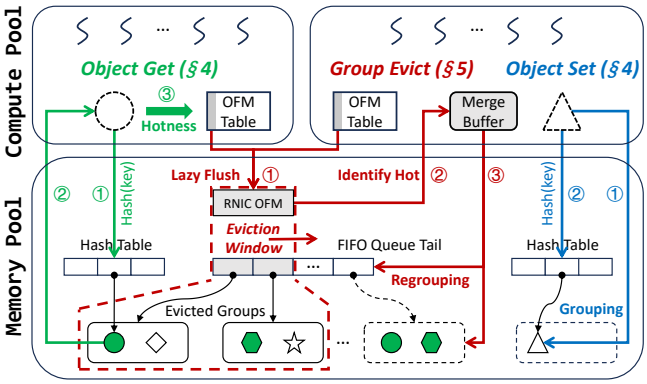


Figure 4: The overview of FORGE.

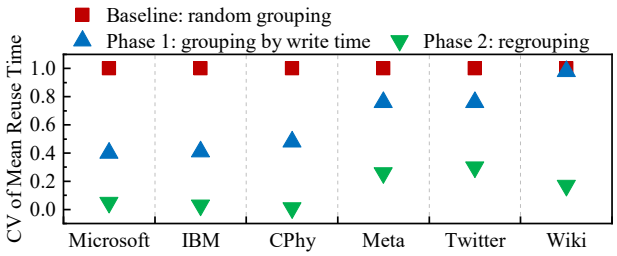


Figure 5: Similarity of objects grouped by different schemes.

amortizing synchronization costs across multiple objects.

For `Get` requests, all CNs can read objects from any groups (full or partially filled) via the hash table. On cache hits, CNs increase the objects' frequency counters in their local fine-grained Object Frequency Maps (OFMs). These updates are buffered and lazily flushed to MNs at the group level to track global hotness of objects (§ 4.3). This design directly addresses **C2**: object-level tracking guarantees accuracy, while group-level flushing amortizes synchronization overheads.

To tackle **C1**, FORGE significantly enhances basic group-FIFO eviction through two key techniques. First, it integrates a lightweight regrouping mechanism. During eviction, CNs fetch synchronized OFMs from MNs to identify hot objects. FORGE's Two-Way Index Map (§ 5.3) allows CNs to efficiently extract and consolidate only the hot objects for reinsertion, bypassing the overheads of scanning entire groups and querying hash tables. Second, FORGE introduces virtual segmentation (§ 5.4) to infuse the FIFO queue with hotness-awareness, assigning hotter groups to deeper segments within the queue to systematically delay their eviction and prioritize the removal of colder groups. Together, these strategies effectively improve cache hit ratios with minimal coordination.

To resolve **C3**, FORGE introduces a novel lazy-yet-timely hotness synchronization strategy (§ 5.2) that leverages the predictable, sequential eviction order of the FIFO queue. By anticipating which groups are nearing the queue head and thus imminent eviction, distributed CNs defer flushing fine-grained object hotness metrics (OFMs) until groups enter the eviction window, effectively eliminating unnecessary synchronization traffic. This approach ensures metrics are updated precisely when needed for eviction decisions, balancing freshness and efficiency. Moreover, FORGE directs this synchronized traffic to the fast on-chip memory of RNICs in MNs, which is small but sufficient for handling the limited number of groups within the eviction window, further reducing latency and overhead.

As previewed in Fig. 3, FORGE drastically reduces synchronization overheads, thereby achieving significantly higher throughput and lower latency while improving the cache hit ratio compared with state-of-the-art systems [62].

4 Grouping Management for Cached Objects

4.1 Object Grouping Scheme

FORGE groups cached objects with similar access patterns to enable effective group-level eviction. The rationale is that by collectively evicting groups of cold objects while preserving hot ones, the system can maintain high hit ratios while amortizing synchronization overheads. To enhance intra-group similarity, FORGE adopts a two-phase grouping scheme.

Phase 1: Write-Time Grouping. During `Set` operations, each CN exploits temporal locality by sequentially inserting newly written objects into a group with available space, ceasing additions once the group reaches capacity. This approach leverages empirical findings that objects written contemporaneously often exhibit correlated access patterns [59, 74, 76]. Moreover, this method remains application-agnostic by relying exclusively on object write time, a universally available and lightweight attribute. This design ensures broad generalizability without requiring prior knowledge of object semantics or access characteristics.

Phase 2: Hotness-Based Regrouping. When a group reaches eviction, the hotness metrics accumulated over its lifecycle serve as a robust indicator of object access patterns. FORGE leverages these metrics to identify and extract hot objects from the evicted group, merging them into a new hot group for reinsertion. This strategy effectively consolidates frequently accessed objects while filtering out colder ones, thus enhancing cache efficiency with minimal synchronization overhead.

To validate the grouping effectiveness, we measure intra-group similarity using the real-world workloads in §6. We calculate the coefficient of variation (CV, standard deviation divided by the mean) of object reuse times within the same group, where a lower CV indicates higher similarity. Fig.5 reports the average CV of 10,000 groups of 256 objects under three schemes: 1) random grouping, 2) Phase-1 write-time grouping, and 3) Phase-2 hotness-based regrouping. The results show that Phase-1 grouping improves intra-group similarity over random grouping, while Phase-2 regrouping further refines it. Similar trends hold for the other group sizes (16–1024 objects), but the results are omitted due to space limitation. As a result, the two-phase grouping scheme enables FORGE to evict cold objects in batches while preserving the

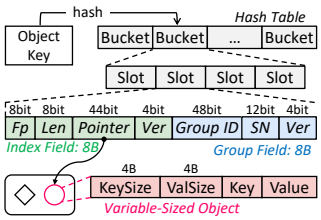


Figure 6: The data structures of the hash table and object.

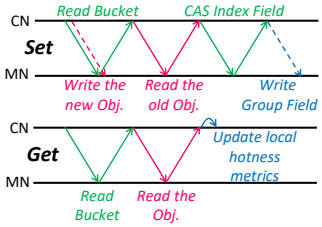


Figure 7: The workflows of Set and Get operations.

hot ones with similar access patterns for future reuse.

4.2 Grouping-Aware Caching Operation

To ensure low latencies and high throughputs for caching systems in DM, FORGE proposes performance-critical `Get`/`Set` operations to become lock-free and non-blocking, while supporting efficient group-level management.

As shown in Fig. 6, each slot in the hash table in MNs contains two atomic 8-byte fields, i.e., the index field and group field. The atomic index field contains a 1-byte fingerprint (*fp*) of the object key for quick filtering, a 1-byte object length in the granularity of fixed-size memory blocks (e.g., 256 bytes), and a pointer to an object in MNs. The atomic group field contains the object's group ID, and an intra-group sequence number (*SN*). The group ID is the unique key of the group determined at group allocation time, while intra-group *SN* indicates the object's insertion order within the group.

Lock-free insertion typically uses pointer swapping: CN writes an object to MN and then atomically installs the pointer into the index using `RDMA_CAS` [46, 52, 62, 85]. But since `RDMA_CAS` works on 8-byte granularity, it cannot update both the index and group fields simultaneously. A naive solution, i.e., locking the slot to atomically update both fields, would block concurrent accesses and degrade `Get`/`Set` performance.

FORGE employs lightweight versioning to ensure non-blocking progress and consistency without locks. Each slot in the system includes a 4-bit version number for both its index and group fields. During a `Set` operation, the CN first updates the index field with an incremented version number and later updates the group field to match the version. This two-phase versioning mechanism guarantees that subsequent `Get` operations can safely read consistent group information by verifying version matches, eliminating the need for locks while maintaining data integrity and high concurrency.

Version-Assisted Lock-Free Set. Fig. 7 shows the processing of `Set` operations. In the first Round Trip Time (RTT), a CN issues an `RDMA_READ` to fetch the hash bucket that the object key corresponds to, and in parallel writes the object to a group with available space in MNs (Phase-1 grouping in § 4.1). The CN then scans *fp*-matched slots in the fetched bucket and follows pointers to read objects and identify a key match. If a key-matched slot is found, the CN sends `RDMA_CAS` to update the slot's index field with the new object pointer and an incremented version number. If not matching, an empty slot is selected and updated in a similar manner. Upon a successful `RDMA_CAS`, the CN asynchronously issues an `RDMA_WRITE` to update the group field with the object's group ID, *SN*, and a version number matching the index field. Since the CN does not need to wait for the final asynchronous `RDMA_WRITE` to complete, the group-level management does not cause additional RTTs to the critical path of `Set` operations.

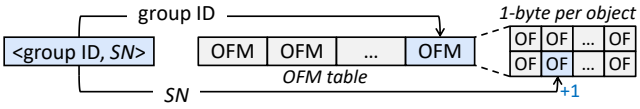


Figure 8: Object hotness tracking in the CN-side OFM table.

Version-Assisted Lock-Free Get. As shown in Fig. 7, the workflow of `Get` is similar to `Set` when locating the object with the target key, except that `Get` does not write a new object or update the slot. Since the `Set` operation atomically updates the index fields using `RDMA_CAS`, the `Get` operation can always follow the pointer to read the consistent and up-to-date object data without blocking. Upon locating a target object (i.e., a cache hit), the CN verifies that the version numbers of the object's index and group fields match, ensuring both correspond to the same atomic `Set` operation. In rare cases where a `Get` operation races with a concurrent `Set`, it may transiently observe a stale version in the group field, when the `Set` has updated the index field via `RDMA_CAS` but its asynchronous `RDMA_WRITE` to the group field is still in flight. This timing window is inherently brief (spanning one RDMA RTT of a few μs) and does not compromise correctness, since the index field that is updated first always references consistent object data. When versions match, the CN extracts the object's group ID and intra-group *SN* from the group field to accurately track hotness, maintaining integrity without locking.

4.3 Group-Based Object Hotness Tracking

As shown in Fig.8, a CN tracks object hotness by increasing access counters in a local Object Frequency Map (OFM) table. Specifically, each OFM corresponds to a group, structured as a compact 1-byte-per-object frequency array indexed by the intra-group *SN*. This layout ensures that the *i*-th counter directly maps to the *i*-th object within the group. Prior to group eviction, OFMs buffered across distributed CNs are flushed to MNs to synchronize global hotness metrics. The OFM design offers the following key advantages in DM architectures.

CN-Side Memory Efficiency. CN local memory is scarce, necessitating efficient utilization [43, 45, 46]. Existing object-level systems [62] waste limited CN memory by storing object keys to index buffered hotness metrics, which is inefficient given that real-world keys are often long (e.g., $\geq$38 bytes for 50% of objects at Twitter [75]). FORGE addresses this

inefficiency via its OFM table, which indexes metrics using compact group IDs and intra-group *SNs*. Since *SNs* serve as implicit offsets within arrays, they require no explicit storage, eliminating key-induced memory bloat. This design dramatically reduces the metadata footprint: a mere 1GB table can track hotness for one billion objects, maximizing CN memory efficiency while maintaining full hotness tracking capability.

Network Efficiency. Since the 1-byte-per-object frequency counters within a group are stored contiguously, the counters for 8 objects can be flushed in a single 8-byte `RDMA_FAA`, significantly reducing synchronization traffic.

In DM caching, timely synchronization of hotness metrics from distributed CNs to MNs is critical. Without this, stale metrics on MNs cause hot objects to be misidentified as cold, inducing erroneous evictions and degraded hit ratios. State-of-the-art systems [62] rely on random sampling to select eviction candidates, which makes future metric needs unpredictable and forces periodic synchronization, resulting in a dilemma: short intervals increase overhead, while long intervals risk staleness. In §5.2, FORGE resolves this dilemma by leveraging the predictable FIFO eviction order to enable low-overhead lazy synchronization of hotness metrics precisely when needed, ensuring accuracy without unnecessary traffic.

5 FIFO-Based Replacement for Groups

FORGE adopts a group-level FIFO replacement to overcome the synchronization amplification in DM. First, evicting groups in FIFO order avoids inefficient random sampling and expensive per-object evictions in DM. Moreover, the strict FIFO order prevents unfair group evictions misled by highly aggregated group-grained hotness metrics. Furthermore, the predictability inherent in FIFO eviction unlocks novel and efficient synchronization schemes in DM.

5.1 Contention-Free FIFO Management

Conventional FIFO queues structured as linked lists [8, 31, 82] require exclusive locks to serialize concurrent operations. Akin to LRU as analyzed in § 2.2, this serialization imposes a severe bottleneck in DM by allowing only a single CN to progress while others are blocked. FORGE eliminates this contention by structuring the queue as a lock-free ring array.

As shown in Fig. 9, the FIFO queue on an MN is organized as a ring array of N contiguous nodes. For convenience, we denote the i-th node in the ring array as $A[i]$. The 4-byte head cursor points to the FIFO head node $A[head \bmod N]$, which becomes the oldest in the queue and will be dequeued next. The tail cursor points to $A[tail \bmod N]$ where the next FIFO node will be enqueued. Crucially, the two cursors are packed into an 8-byte aligned word on the MN to enable `RDMA_FAA`-based contention-free FIFO enqueue and dequeue as follows.

When a group reaches capacity, the CN filling it enqueues a FIFO node referencing the group by first performing an

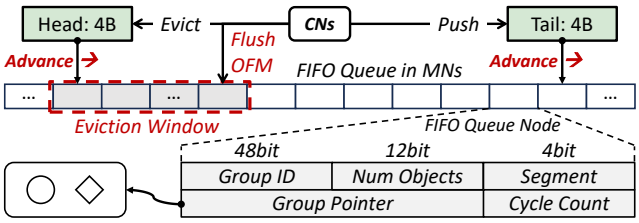


Figure 9: The CN-driven FIFO queue management.

`RDMA_FAA` to atomically advance the tail cursor by one and retrieve the *oldTail* value. The CN then issues an asynchronous `RDMA_WRITE` to populate the FIFO node at $A[oldTail \bmod N]$ with the queue cycle count ($\lfloor oldTail/N \rfloor$), the group's unique ID, the number of grouped objects, and a pointer to the group's location, thereby efficiently linking the group to the queue.

When a cache is full, a CN evicts cold groups in the eviction window starting from the FIFO queue head. Specifically, the CN dequeues FIFO nodes by advancing the head cursor by k (e.g., 4) with `RDMA_FAA` and reads the k FIFO nodes starting at $A[oldHead \bmod N]$ via `RDMA_READ`. To ensure consistency, the CN verifies whether the fetched FIFO nodes belong to the current queue cycle by comparing their cycle counts against the expected cycle. Any mismatch indicates an unfilled or stale node from the last cycle, prompting the CN to retry the fetch. Once consistency is ensured, the CN uses the metadata in the fetched FIFO nodes to conduct group-level evictions. A `Segment` field in the FIFO node enables hotness-aware optimizations during eviction to improve hit ratios (§ 5.4).

In state-of-the-art systems [62], multiple CNs can independently sample and then race to evict the same cold objects using `RDMA_CAS`, where only one CN succeeds while the others suffer failed attempts and retry overhead under contention. In contrast, our `RDMA_FAA`-based management is contention-free. Each atomic `RDMA_FAA` on the cursors succeeds without retries and returns a unique old value, ensuring that multiple CNs can concurrently and safely enqueue and dequeue groups without locks or `RDMA_CAS`-retry loops. This design substantially reduces atomic operation overhead, thereby mitigating interference with performance-critical `Get`/`Set` executions.

5.2 Lazy Hotness Synchronization

To maintain high hit ratios, accurate hotness metrics (tracked via OFMs in § 4.3) are important for identifying hot objects during eviction. However, synchronizing OFMs from distributed CNs to MNs eagerly upon every cache hit incurs significant overheads. To achieve high efficiency, we observe that *the synchronized hotness metrics are not necessary right away upon cache hits, but only later during evictions, whose timing is predictable in the FIFO-based replacement.*

Our key insight is that, under FIFO-based replacement, a group only becomes eligible for eviction once its corresponding FIFO node enters the eviction window near the queue

head (Fig. 9). For groups outside this window, CNs track OFMs locally without synchronization, avoiding unnecessary DM traffic. When a group's FIFO node enters the eviction window, CNs proactively flush their buffered OFM updates for that group to MNs using efficient batched operations, i.e., one RDMA_FAA per 8 counters. This flush-on-entry strategy ensures that by the time a CN dequeues the FIFO node to evict the target group, the group's OFM on MNs has been globally synchronized across all CNs, providing accurate hotness metrics for informed eviction decisions.

However, if the eviction window is too small, a group may be quickly evicted after entering the window, before CNs can timely detect its entry and flush the buffered OFMs. FORGE prevents this detection failure by determining a safe eviction window size that guarantees CNs have sufficient time to detect new groups and flush their OFMs before eviction.

Robustness via Physical Bounds. To determine the window size, we analyze the maximum eviction rate under a worst-case write-miss workload (i.e., 0% hit ratio), where each operation is a cache-miss Set involving at least 2 RDMA_WRITE, 1 RDMA_READ, and 1 RDMA_CAS. As shown in Fig. 2a, RNIC throughput limits are 107 Mops/s for WRITE, 88 Mops/s for READ, and 20 Mops/s for CAS. Consequently, the throughput bottleneck of cache-miss Set becomes: $\frac{1}{\frac{2}{107}+\frac{1}{88}+\frac{1}{20}} \approx 12\,\text{Mops/s} = 12\,\text{ops/}\mu\text{s}$. Thus, in each μs, at most 12 new objects are inserted and the other 12 are evicted. Suppose the eviction window holds 16 groups and each group contains 256 objects, evicting them all consumes at least $\frac{16\times256}{12} \approx 341\,\mu$s. Therefore, CNs only need to check the eviction window every 100 μs, providing a 3.4× safety margin over the maximum eviction rate, to ensure timely OFM flushing before group eviction. In practice, real-world workloads [9, 53, 69, 75] exhibit higher hit ratios than this worst case and consequently lower eviction rates, permitting even more relaxed detection intervals. For system configurations (e.g., more RNICs per MN) that sustain higher maximum eviction rates, FORGE dynamically adjusts both the eviction window size and detection intervals to ensure robust and timely synchronization.

In every detection interval (e.g., 100 μs), each CN issues one probing RDMA_READ to fetch the current FIFO head value *headNow*, in parallel with normal Get/Set RDMA RTTs to minimize detection overhead. The CN compares *headNow* with the previous value *headLast* from the last detection interval. If $headNow > headLast$, $headNow - headLast$ new FIFO nodes have entered the eviction window. In this case, the CN issues another RDMA_READ to fetch these FIFO nodes in the eviction window, extracts their group IDs, and flushes the corresponding buffered OFMs from the CN to MNs using one RDMA_FAA per 8 bytes. This design ensures each CN incurs negligible detection overhead, while all groups' OFMs are lazily yet timely synchronized before evictions.

Acceleration via RNIC Device Memory. Our analysis of the maximum eviction rate reveals an insight: *although the cache capacity in MNs could be large, the number of objects concurrently evicted by CNs remains small*. Our FIFO-based replacement further confines concurrent evictions to a limited number of objects within the eviction window. This eviction locality enables FORGE to leverage the fast yet limited (256KB) on-chip device memory of RNICs in MNs to accelerate synchronization of hotness metrics (OFMs).

Based on this insight, CNs direct their synchronization traffic to a dedicated region (e.g., 16 KB) in the on-chip memory of the RNIC in MNs. This region is organized as an OFM Ring Array (short for *RA*) and consists of M contiguous OFMs. The value of M is configured to exceed the number of groups in the eviction window. This is highly practical, since a 16KB buffer can accommodate 1-byte-per-object OFMs for up to 16K objects, which are 4× more than the 16 groups of 256 objects in the eviction window. When the FIFO node $A[i \bmod N]$ enters the window, CNs flush the buffered OFM of the corresponding group to $RA[i \bmod M]$ via RDMA_FAA. Furthermore, the head and tail cursors of the FIFO queue, occupying only 8 bytes, are also maintained in RNIC on-chip memory to accelerate RDMA_FAA-based enqueue and dequeue.

As shown in Fig. 2, this on-chip synchronization reduces RDMA_FAA overhead by up to 5× and alleviates performance interference with concurrent Get/Set executions. Importantly, while most commodity RNICs provide such on-chip memory, this acceleration is an optimization, not a dependency. If on-chip memory is unavailable, FORGE seamlessly places the OFM Ring Array and cursors in MN host memory while preserving correctness and the benefits of lazy synchronization.

5.3 Lightweight Object Regrouping

When a CN evicts a batch of k (e.g., 4) groups by advancing the FIFO head (obtaining $oldHead$), it fetches the corresponding k FIFO nodes starting from $A[oldHead \bmod N]$ and their fully synchronized OFMs starting from $RA[oldHead \bmod M]$. The CN then zeroes out these OFM locations in RA via RDMA_WRITE, initializing them for future reuse in the ring.

Guided by the access frequencies in the fetched OFMs, the CN ranks the objects across the k evicted groups and consolidates the hottest objects into a new group until it becomes full (Phase-2 regrouping in § 4.1). As shown in Fig. 10, this process entails a sequence of parallelized operations: reading hot objects from their original locations, compacting them into a consolidated buffer, writing the new group to the MN, and updating the corresponding hash table slots to reference the objects' new locations.

However, this process faces a critical performance challenge: locating specific objects and their hash table slots is prohibitively expensive without fine-grained indexing. A straightforward approach would be to read the entire group chunk via RDMA_READ, scan it sequentially to extract object keys, and then perform a hash table lookup for each key. Such a strategy is impractical due to generating excessive network traffic, fundamentally undermining FORGE's efficiency.

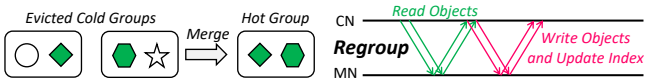


Figure 10: Regrouping hot objects from evicted groups.

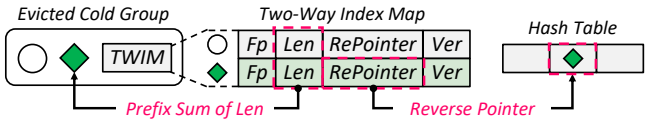
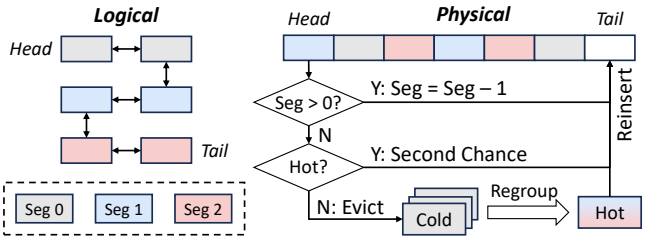


Figure 11: The Two-Way Index Map for regrouping.



(a) Queue segmentation (b) Ring-array-based segmentation

Figure 12: The virtual segmentation of the FIFO queue.

To enable lightweight regrouping, FORGE introduces a Two-Way Index Map (TWIM) embedded at the tail of each group's chunk. As shown in Fig. 11, the TWIM is an array where each 8-byte entry corresponds to an object indexed by its intra-group *SN*. Each TWIM entry mirrors the object's index field in the hash table but replaces the object pointer with a reverse pointer to the object's hash table slot. The TWIM is constructed by the CN that populates the group, allowing object start offsets within the group to be calculated by summing preceding object lengths stored in the TWIM.

With the aid of TWIM, regrouping becomes highly efficient. When evicting a group, the CN uses FIFO node metadata to derive the TWIM address ($GroupPointer + ChunkSize - NumObject \times 8$) and fetches the OFM and TWIM in parallel. The OFM identifies the *SNs* of hot objects, while the TWIM provides their exact data locations and index slot addresses. This allows the CN to only read the necessary hot objects and update their index slots with minimal RDMA operations, eliminating costly full-group scanning and hash table lookups.

5.4 Hotness-Aware FIFO Queue

Although the contention-free FIFO queue minimizes synchronization overheads for replacements, its assumption that older objects are colder does not always hold. Many real-world workloads exhibit skewed access patterns [7, 12, 13, 75]: older objects can remain frequently accessed, while newly inserted ones are often short-lived [77]. As a result, plain FIFO often achieves lower hit ratios than adaptive schemes like LRU [20].

To improve adaptability, FORGE partitions the FIFO queue into segments to prioritize evicting cold objects. Groups are inserted into different segments based on their hotness: the hotter a group is, the farther from the head it is placed. For example, groups in Phase-1 containing likely-cold new objects are inserted into the first segment (i.e., Segment 0 in Fig. 12a), near the head and thus evicted quickly. In contrast, a merged group in Phase-2 is likely hotter and thus inserted into the deeper segments (e.g., Segments 1 or 2) according to its aggregated access frequency, delaying their eviction. Consequently, the groups with colder objects are evicted sooner than hotter ones, thus increasing the proportion of hot objects retained in the cache. However, this segmented FIFO queue requires a linked-list-based structure that allows random insertion, limiting only one CN holding the lock to modify the queue, while other CNs are blocked. This conflicts with FORGE's contention-free ring-array design.

To avoid this conflict, FORGE adopts virtual segmentation on top of its contention-free ring array by embedding a *Segment* field in each FIFO node. As shown in Fig. 12b, CNs simulate segment-based insertion by assigning *Segment* values to FIFO nodes while physically enqueuing them at the queue tail. Phase-1 groups are enqueued with *Segment = 0*, and Phase-2 merged groups are enqueued with *Segment > 0*.

When a FIFO node with *Segment > 0* is dequeued, its group is not evicted. Instead, the CN decrements its *Segment* value and reinserts it into the queue tail. Only nodes with *Segment = 0* are considered for eviction. Before eviction, the CN inspects the group's OFM: if most objects are active (e.g., over half the counters are non-zero), the group is reinserted rather than evicted, and the CN proceeds to the next candidate. This reinsertion avoids unnecessary regrouping for hot objects already co-located in one group, and speeds up the eviction of subsequent groups with more cold objects. Moreover, since the OFM is zeroed out upon fetch (§ 5.3), the reinsertion restarts the group's hotness tracking from zero for the next eviction cycle, preventing stale metrics from accumulating indefinitely. Overall, virtual segmentation enables hotness-aware eviction while preserving FORGE's ring-array structure for efficient synchronization.

5.5 Discussions

Support for multi-queue designs. Prior work [22, 28, 32, 33, 77] employs multiple queues to filter out unpopular objects and mitigate cache pollution. As an orthogonal optimization, FORGE also incorporates this approach by maintaining a 20% small FIFO queue and an 80% main FIFO queue. Each queue is independently managed with our virtual segmentation and lazy hotness synchronization, accelerated via RNIC on-chip memory. New groups first enter the small queue. When the *small* queue is full, its head group is either promoted and reinserted into the *main* queue if hot, or evicted and merged if cold. The main queue applies the same policy when full.

Considerations for tiered caching. In addition to MN-tier

caching, CNs can leverage their limited local memory to cache objects, thereby accelerating `Get`/`Set` operations. However, hotness tracking, eviction, and GC in the MN tier still require cross-CN synchronization to ensure coherence and consistency. Therefore, the relative synchronization cost can become even higher in systems with CN-tier caching, further highlighting the motivations and benefits of FORGE's synchronization-efficient mechanisms for managing the global tier on MNs. FORGE currently does not employ CN-tier caching, aligning with state-of-the-art systems [62] for fair comparisons.

Adaptability to CXL-based DM. While FORGE primarily focuses on RDMA-based DM and assumes one-sided RDMA verbs, its design principles are potentially applicable to CXL-based DM using memory-semantic instructions. Recent research [29, 30] indicates that CXL shared memory pools introduce new architectural constraints for cross-CN coherent synchronization. As a result, caching systems like Ditto [62], which rely on fine-grained synchronization for object-level housekeeping, may encounter scalability bottlenecks due to excessive synchronization traffic. In contrast, FORGE significantly reduces cross-node synchronization demand through coarse-grained group management and lazy hotness flushing, making it highly adaptable to the architectural constraints of future CXL environments.

6 Performance Evaluation

6.1 Experimental Setup

Testbed. We conduct our evaluation on a cluster of 6 machines (3 CNs and 3 MNs), each with two Intel Xeon Gold 6230R CPUs, 256 GB DRAM, and a 100Gbps Mellanox ConnectX-5 RNIC connected to a 100Gbps InfiniBand Switch.

Comparisons. We compare FORGE with three state-of-the-art systems, Ditto [62], S3-FIFO [77], and GL-Cache [74].

Ditto [62] represents the state-of-the-art in DM caching systems, employing sophisticated object-level hotness tracking and sampling-based eviction. Similar to LeCaR [68], CNs utilize multiple eviction policies (e.g., LRU and LFU), select the highest-weight policy to predict the utility of sampled objects, and evict the lowest-utility objects. CNs maintain a history of recently evicted objects, which they use to adjust policy weights to correct mispredictions. **S3-FIFO** [77] is the latest object-level replacement algorithm. It uses multiple FIFO queues to filter out unpopular objects and promote hot objects through the queues. **GL-Cache** [74] is an efficient group-level caching system. It uses decision trees to predict the utility of sampled groups based on their group-level hotness metrics and evicts those with the lowest predicted utility.

However, S3-FIFO and GL-Cache are designed for traditional monolithic servers and cannot directly work on DM. Thus, based on their original papers [74, 77], we faithfully re-implement them on RDMA-based DM (i.e., S3FIFO-DM and GLCache-DM). For fair comparisons, we further enhance them with DM-relevant designs proposed by FORGE, including lock-free `Get`/`Set` operations, contention-free FIFO management, asynchronous and batched RDMA for hotness flushing, and lightweight object regrouping. In GLCache-DM, CNs periodically flush collected training data (e.g., group hotness metrics and utility scores) to MNs, where decision trees are asynchronously retrained to avoid blocking CNs. Although SIEVE [82] is another prominent FIFO-based algorithm for monolithic servers, our analysis in § 2.2 demonstrates its severe synchronization bottlenecks in DM due to serialized doubly-linked queue scanning. As a result, SIEVE-DM delivers 3.1–5.0× lower throughput than Ditto. In contrast, our enhanced S3FIFO-DM leverages a contention-free ring-array to mitigate these synchronization bottlenecks, making S3FIFO-DM a much stronger and robust representative for object-level FIFO schemes throughout our evaluation. All the evaluated systems are implemented in C/C++. Since Ditto's open-source prototype only supports a single MN, we use one MN for fair comparisons across all four systems, and further evaluate the other three systems with multiple MNs in §6.5.

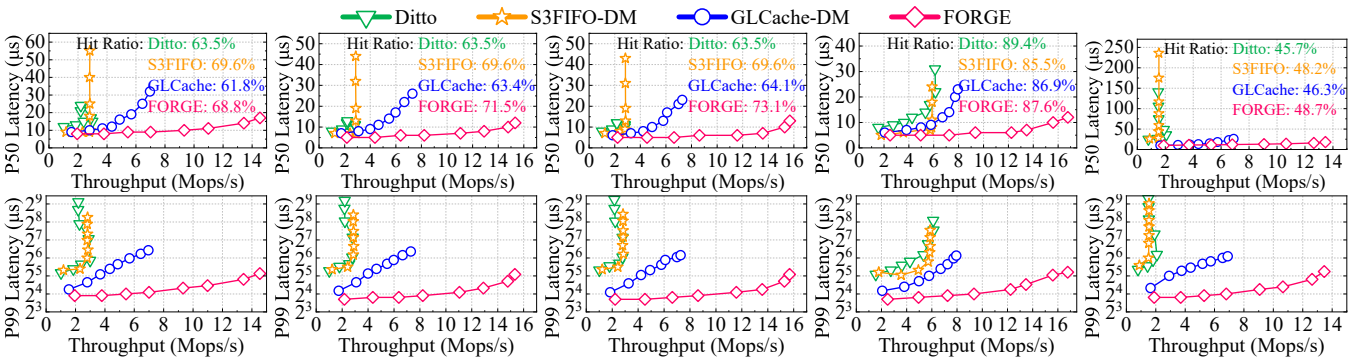
Configurations. We use the recommended configurations of all baselines, e.g., a sample size of 5 and a history size of 1024 for Ditto [62], a 10% small queue and an object frequency ceiling of 3 for S3-FIFO [77], 7-dimensional group-level hotness metrics and XGBoost-based [16] decision trees for GL-Cache [74]. For FORGE and GLCache-DM, the group size is 64 objects, and each eviction merges 8 groups. We analyze the sensitivity of FORGE to group sizes in §6.6.

Workloads and Methodology. The YCSB [19] workloads with a Zipfian distribution ($\theta = 0.99$) include: A (50% GET, 50% UPDATE), B (95% GET, 5% UPDATE), C (100% GET), D (95% GET, 5% INSERT), and E (95% SCAN, 5% INSERT). The real-world workloads cover object stores from Wikimedia [64] and IBM [25], block I/O from CloudPhysics [69] and Microsoft Research Cambridge [53], and key-value caches from Twitter [75] and Meta [9]. Each workload comprises up to 100 million requests, while the IBM workload contains 10–40 million requests due to trace limitations. Before evaluation, we warm up the cache with the first 20% of requests. Following state-of-the-art methodologies [62, 74, 77, 82], we evaluate the cache as a closed system with on-demand fill, where each `Get` miss includes the overhead of an uncounted `Set`. This setup provides performance isolation and generality across storage backends. We further analyze the sensitivity of system performance to different miss penalties in §6.6.

6.2 Performance in YCSB Workloads

We evaluate the four systems under YCSB workloads with 256-byte objects and a 20% cache size, a standard setting in DM caching [62]. The performance trends and bottlenecks remain consistent with the smaller 10% cache ratio in Fig. 3. We vary the number of threads across 3 CNs from 16 to 256.

As shown in Fig. 13, Ditto and S3FIFO-DM exhibit the



(a) YCSB A-20% (b) YCSB B-20% (c) YCSB C-20% (d) YCSB D-20% (e) YCSB E-20%

Figure 13: Throughputs, 50th (upper) and 99th (lower) percentile latencies of systems in YCSB workloads at 20% cache sizes.

worst throughputs and latencies, bottlenecked by the costly object-level synchronizations. Their CNs frequently synchronize object hotness updates to MNs when cache hits, creating significant RDMA traffic (corroborating the traffic breakdown in Fig. 3) that interferes with `Get`/`Set` and inflates P50 latencies. Ditto embeds the hotness metadata in hash buckets to facilitate tracking and sampling, but the enlarged hash buckets increase the RDMA payload of each index lookup. Moreover, on cache misses, their object-level eviction mechanisms lie on the critical path and lead to high P99 latencies. In Ditto, the random sampling and `RDMA_CAS`-based eviction incur large overheads and contention-induced retries. S3FIFO-DM, despite being improved by using our contention-free FIFO queue, still suffers from high eviction overheads because its policy requires intensive RDMA operations to demote cold objects and promote hot ones through multiple queues.

GLCache-DM and FORGE remove the group-level hotness tracking and eviction from `Get`/`Set` critical paths and significantly reduce synchronization traffic. Thus, they achieve higher throughputs and lower latencies than Ditto. However, like Ditto, GLCache-DM suffers from inefficient sampling and retry under contention, and the buffered hotness metrics need to be synchronized at intervals to avoid stale metrics on MNs. Moreover, each inference of the decision trees consumes hundreds of microseconds of CN CPU cycles. Consequently, the performance of GLCache-DM is still constrained.

FORGE excels by eliminating synchronization bottlenecks. Its contention-free FIFO management eliminates the inefficient sampling that plagues Ditto and GLCache-DM. Its version-assisted lock-free index and TWIM-based lightweight regrouping overcome the potential inefficiency of coarse-grained group management. More critically, FORGE's lazy synchronization ensures that hotness metrics are flushed just-in-time for eviction, radically reducing synchronization traffic (validating the benefits previewed in Fig. 3) without sacrificing metric accuracy. Consequently, FORGE achieves 2.0–8.7× higher throughput, 1.3–5.9× lower P50 latency, and 3.9–13.3× lower P99 latency than all baselines.

FORGE demonstrates superior cache performance, achieving hit ratios in the top two across workloads. Its lazy synchronization equips CNs with globally accurate hotness information, enabling precise eviction decisions and the effective identification and reinsertion of hot objects. In contrast, the hit ratios of Ditto (and GLCache-DM) are hampered by their dependence on random sampling and stale (or misleading group-grained) hotness metrics, with read-latest YCSB D being an exception where their recency-aware algorithms like LRU align well with the access pattern. Although S3FIFO-DM also benefits from accurate metrics, FORGE maintains a hit ratio advantage in YCSB B–E by augmenting the FIFO foundation with a hotness-aware virtual segmentation that proactively evicts colder groups within the same queues, adapting better to shifting access patterns. The write-intensive YCSB A proves challenging for all grouping methodologies, yet FORGE sustains a highly competitive stance. These collective results validate that FORGE's synchronization efficiency delivers comprehensive system improvements, boosting not only throughput and latency but also enhancing hit ratios.

6.3 Performance in Real-World Workloads

We evaluate the four systems with real-world workloads from CloudPhysics (CPhy), Wikimedia (Wiki), Microsoft Research Cambridge (MSR), and IBM, using 128 threads across 3 CNs. Aligned with the setup in Ditto's paper [62], object sizes are fixed at 256 bytes, while cache sizes are varied among 5%, 10%, and 20% of the trace footprints.

As shown in Fig. 14, the performance of both Ditto and S3FIFO-DM remains constrained by their costly object-level synchronization mechanisms. In particular, S3FIFO-DM delivers the lowest throughput and highest latency in most cases. Its eviction policy necessitates frequent object demotion and promotion across multiple queues. Such frequent queue management triggers a high volume of atomic `RDMA_FAA` operations, which are especially expensive and significantly interfere with performance-critical `Get`/`Set` as shown in Fig. 2.

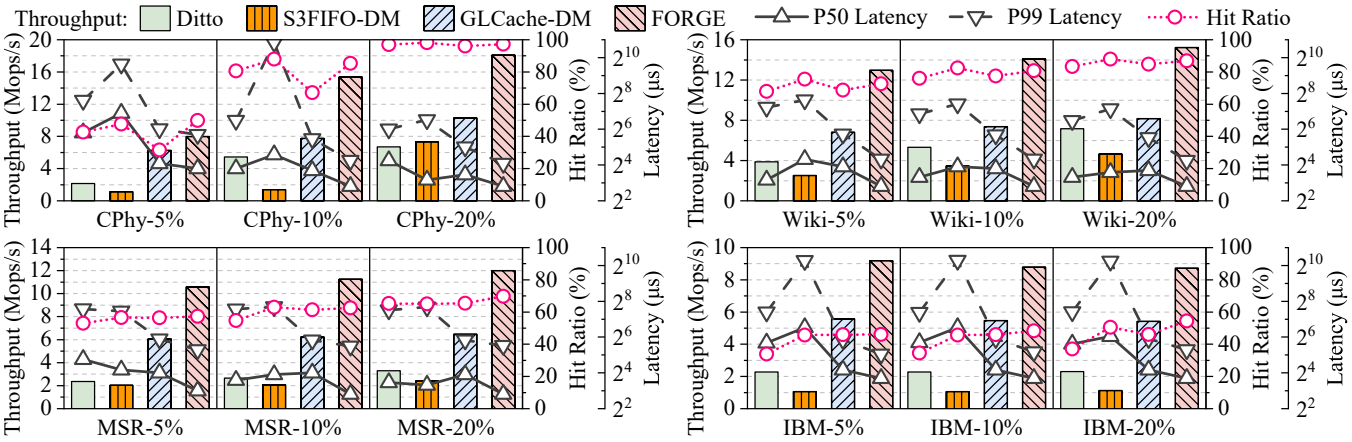


Figure 14: Throughputs, latencies, and hit ratios in real-world workloads at 5%, 10%, 20% cache sizes with fixed object sizes.

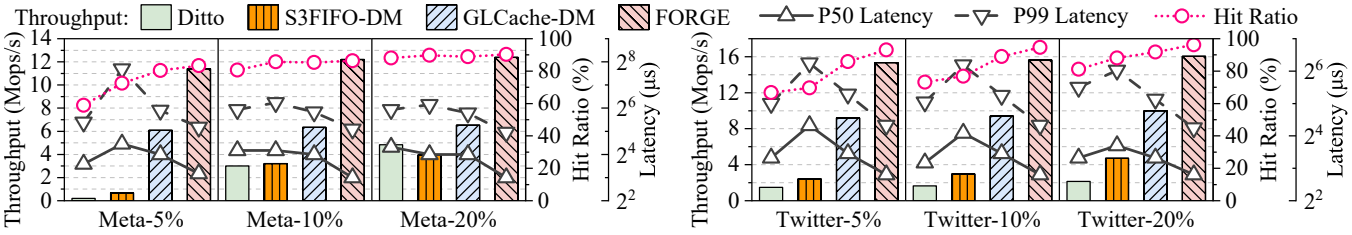


Figure 15: Throughputs, latencies, and hit ratios in real-world workloads at 5%, 10%, 20% cache sizes with variable object sizes.

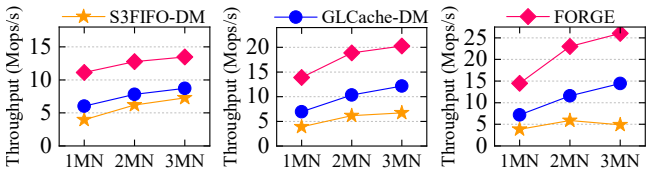
In CPhy with a 5% cache size (short for CPhy-5%), the small cache cannot hold enough groups for GLCache-DM to train reliable decision trees, impairing the model's robustness and reducing eviction effectiveness and hit ratios. In CPhy-20%, the large cache and skewed access pattern lead to high hit ratios (≈96%) for all systems. Higher hit ratios reduce eviction frequency and overhead, thereby improving the throughputs and latencies of all systems. However, despite lower rates of eviction and random sampling, the eviction timing of any specific object (and group) in Ditto (and GLCache-DM) remains unpredictable. Therefore, CNs in both systems still need to periodically flush buffered hotness metrics to MNs, incurring DM traffic and limiting performance.

In contrast, FORGE synchronizes hotness just-in-time for eviction, thereby adapting to dynamic hit ratios. Leveraging this lazy mechanism, CNs flush metrics only when a new group enters the eviction window. Under high hit ratios in CPhy-20%, evictions are rare, and new groups enter the eviction window slowly. Consequently, the lazy synchronization is infrequent, improving the throughput and latency of FORGE. On the other hand, under low hit ratios in workloads like IBM, evictions occur more frequently to make room for new objects, triggering more rapid hotness synchronization to maintain metric accuracy. This adaptive approach, combined with efficient FIFO-based hotness-aware group eviction, drastically reduces synchronization overheads while maintaining high hit ratios. As a result, compared with Ditto and GLCache-DM, FORGE achieves 1.3–4.5× higher throughput, 1.2–4.0× lower P50 latency, and 1.2–7.5× lower P99 latency across workloads, with an average 1.14× improvement in hit ratios.

6.4 Robustness to Memory Fragmentation

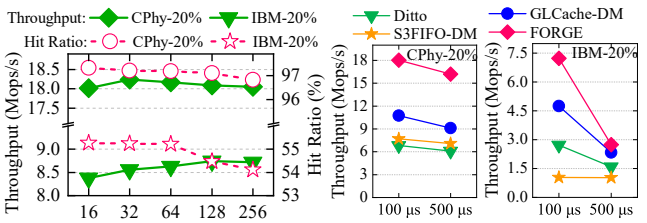
We evaluate the systems in real-world workloads from Meta and Twitter with their dynamic object sizes, using 128 threads across 3 CNs. To establish a stringent baseline for comparison, we enhance the object-level systems (Ditto and S3FIFO-DM) with a slab-based allocator that organizes object blocks into 24 size classes, a standard technique to mitigate external fragmentation [11, 24]. Even with this enhancement, Fig. 15 shows that their performances sharply degrade under small cache sizes (e.g., Meta-5% and Twitter-5%). This is because object-level evictions divide free memory into small, disjoint fragments reclaimed by multiple CNs. Each CN struggles to allocate contiguous memory for large objects, reducing effective memory utilization and lowering hit ratios. With larger cache capacities (e.g., Meta-20% and Twitter-20%) mitigating fragmentation, the performances of Ditto and S3FIFO-DM recover but remain fundamentally constrained by their costly object-level hotness tracking, evictions, and GC.

In contrast, GLCache-DM and FORGE perform evictions and space reclamation at group granularity, avoiding external fragmentation and achieving robustness under dynamic object sizes. Furthermore, FORGE's enhanced FIFO eviction



(a) 100 CN threads (b) 200 CN threads (c) 300 CN threads

Figure 16: The scalability of systems with CNs and MNs in the Meta workload at 20% cache sizes.



(a) Group size (objects) (b) Miss penalty (μs)

Figure 17: The sensitivity analysis of system performance.

and lazy synchronization mechanisms give it a performance advantage. Compared with GLCache-DM (the best baseline), FORGE achieves 1.61–1.93× higher throughput, and reduces P50 and P99 latency by 1.43–2.00× and 1.67–1.95× respectively, with 1.02–1.09× improvements in hit ratios. Since the workloads contain large objects, both systems are ultimately bottlenecked by the single MN's finite network bandwidth.

6.5 Scalability Analysis

Our scalability evaluation employs the Meta workload with a 20% cache size, scaling MNs from 1 to 3 and varying the thread count across 3 CNs from 100 to 300; Ditto is excluded as its open-source prototype lacks multi-MN support, and its CN scalability was previously assessed in §6.2. As depicted in Fig. 16, both FORGE and GLCache-DM demonstrate excellent scalability, where adding MNs increases aggregate network bandwidth, alleviating the network bottleneck and boosting throughput. This improvement is marginal under low thread counts (e.g., 100) where systems are compute-bound, but becomes pronounced with abundant CN threads (e.g., 300) where network bandwidth is the primary constraint. The non-linear compute scaling with thread count stems from overheads and resource contention in Hyper-Threading [48]. S3FIFO-DM scales poorly due to exacerbated memory fragmentation under scale. Ultimately, FORGE's highly efficient synchronization design enables it to outperform GLCache-DM (the best baseline) by 1.54–2.02× across configurations.

6.6 Sensitivity Analysis

We evaluate FORGE's sensitivity to group size and cache miss penalty in CPhy-20% and IBM-20%, the workloads with the highest and lowest hit ratios in § 6.3, respectively.

Group Size. As shown in Fig. 17a, in the low-hit-ratio IBM workload, where evictions occur frequently, larger groups better amortize eviction overheads and thus yield higher throughput. Conversely, smaller groups can maintain higher intra-group access similarity, leading to slightly better hit ratios. In the high-hit-ratio CPhy workload, where evictions are rare, performance is largely insensitive to group size. These results show that FORGE is robust across different group sizes.

Miss Penalty. For each `Get` miss, CN threads are stalled for 100 μs or 500 μs before inserting the missed object into the cache, emulating the realistic latency of fetching objects from distributed storage [35, 47]. As shown in Fig. 17b, the impact of this penalty is highly workload-dependent: in the high-hit-ratio CPhy workload, misses are rare, minimizing the penalty's effect and allowing FORGE to outperform all baselines by 1.68–2.64× and 1.78–2.66× under the 100 μs and 500 μs penalties, respectively. Conversely, the low-hit-ratio IBM workload triggers frequent miss penalties, leading to substantial thread stalls that limit overall throughput and leave the network significantly underutilized, thereby diminishing the relative advantage of FORGE's synchronization efficiency; despite this, FORGE still maintains a clear superiority with performance gains of 1.52–6.94× (100 μs) and 1.17–2.68× (500 μs). More importantly, in datacenters, a large number of concurrent application clients [54] would sustain network saturation despite such stalls, thus giving FORGE's efficient synchronization a decisive performance advantage.

6.7 Contributions of Different Techniques

Fig. 18 presents an ablation study of FORGE on YCSB workloads with a 20% cache size. Starting from a baseline with group-level FIFO eviction, we apply each proposed technique one by one. We use 256 CN threads across 3 CNs.

+ Lock-free Index. Our version-assisted lock-free index eliminates serialization on hash table slots during `Set` operations. This is especially critical for the write-heavy YCSB A, where it improves the throughput by 21.5× and P50 latency by 6.7×.

+ Contention-free FIFO Management. By minimizing the synchronization overheads of FIFO enqueuing and dequeuing, this technique excels on workloads with low hit ratios that trigger frequent evictions. It boosts the throughput by 23.1%–57.5% and reduces P99 latency by 1.25–10.36× across all workloads, with peak gains observed in YCSB E.

+ Lightweight Regrouping. Leveraging TWIMs for object regrouping, CNs efficiently read only the necessary hot objects and update their pointers with minimal RDMA operations. This improves throughput by 32.6%–55.8%, and cuts P50 and P99 latency respectively by 20.0%–65.2% and 45.2%–61.0%.

+ Hotness-aware FIFO Queue. This design enhances FIFO queues through virtual segmentation, effectively retaining hot

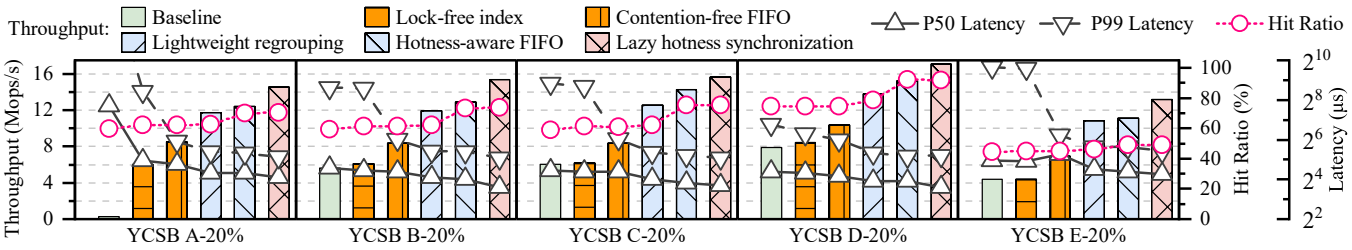


Figure 18: Performance contributions of different techniques across YCSB workloads at 20% cache sizes.

groups longer while prioritizing the eviction of cold ones. Built upon the multi-queue scheme, it yields a substantial 1.12–1.21× improvement in cache hit ratios for YCSB A–D. The gain is more modest (1.06×) in the scan-heavy YCSB E.

+ Lazy Hotness Synchronization. Leveraging the predictability of FIFO evictions, distributed CNs wait until a group nears the queue head to flush its hotness metrics, eliminating unnecessary synchronization traffic. FORGE further accelerates the traffic via RNIC on-chip memory to minimize interference with performance-critical `Get`/`Set` operations. Even on a baseline already heavily optimized by the preceding techniques, this lazy synchronization further improves the throughput and P50 latency by up to 18.9% and 33.3%, respectively.

7 Related Work

Memory Disaggregation. To improve resource utilization, existing studies leverage memory disaggregation across various domains, including indexes [43, 45, 46, 85], key-value stores [26, 39, 52, 63, 67], transaction processing [29, 79, 80], and memory management [15, 40, 42, 49, 61, 70, 83]. Unlike them, FORGE focuses on caching systems in DM. Its closest competitor, Ditto [62], employs an eager object-level housekeeping mechanism that introduces substantial synchronization overhead and undermines overall caching efficiency. In contrast, FORGE pioneers an innovative lazy hotness synchronization mechanism integrated into a DM-tailored group-level framework and enhanced-FIFO replacement strategy, which fundamentally reduces synchronization demands through coordinated eviction of groups rather than individual objects.

Group-based Cache and Storage Management. Grouping objects to amortize operational overheads has been widely explored across various storage tiers. In distributed file systems, SmartStore [27] aggregates correlated files into semantic-aware groups via latent semantic indexing to optimize complex metadata queries. For local flash storage, Kangaroo [50] and RIPQ [66] group small objects into large writes to mitigate flash write amplification. In monolithic in-memory caches, Segcache [76] groups objects with similar lifecycles to enable efficient bulk expiration, whereas GL-Cache [74] clusters objects to amortize machine-learning overheads for eviction. In contrast, FORGE introduces group-level management specifically for DM architectures to resolve the unique bottleneck of cross-node synchronization amplification.

Replacement Algorithms. Prior research [8, 22, 28, 31, 33, 51, 59, 73, 77] focuses on sophisticated eviction algorithms to optimize hit ratios and throughputs in monolithic servers, but fundamentally overlooks the critical challenge of synchronization amplification in DM. For instance, recent algorithms like SIEVE [82] achieve efficiency via simplified structures in monolithic environments. However, as demonstrated in § 2.2, applying SIEVE directly to DM still incurs prohibitive object-level synchronization overhead, especially due to the eager checking and setting of visited bits upon cache hits, and the blocking queue scans during eviction. In contrast, FORGE lazily defers hotness metric flushing until object groups enter the eviction window, bypassing unnecessary DM traffic for hotness tracking upon cache hits. Moreover, FORGE performs group-level eviction using a contention-free and hotness-aware FIFO design, eliminating the costly blocking overhead while maintaining high hit ratios. Overall, FORGE enables a high-performance DM caching system via a holistic redesign with DM-friendly data structures, efficient eviction schemes, and novel synchronization mechanisms.

8 Conclusion

DM amplifies the synchronization overheads in crucial cache housekeeping, creating a critical bottleneck that undermines the benefits of resource disaggregation. FORGE addresses this challenge through a synergistic design with DM-tailored group-level management, a contention-free hotness-aware FIFO policy, and a lazy-yet-timely hotness synchronization mechanism. This cohesive approach minimizes unnecessary cross-node synchronization while preserving accuracy, enabling FORGE to significantly outperform state-of-the-art DM caching systems. The source code of FORGE is publicly available at https://github.com/cszjyang/forge.

Acknowledgments

This work was supported in part by National Natural Science Foundation of China (NSFC) under Grant No. 62125202 and U22B2022. We are grateful to our shepherd and anonymous reviewers for their constructive suggestions and feedback.

References

[1] Compute express link. https://computeexpresslink.org/, 2024.

[2] Infiniband. https://www.infinibandta.org/, 2024.

[3] Memcached. https://memcached.org/, 2024.

[4] Redis. https://redis.io/, 2024.

[5] Dulcardo Arteaga, Jorge Cabrera, Jing Xu, Swaminathan Sundararaman, and Ming Zhao. CloudCache: On-demand flash cache management for cloud computing. In *14th USENIX Conference on File and Storage Technologies (FAST 16)*, pages 355–369, Santa Clara, CA, February 2016. USENIX Association.

[6] Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. Workload analysis of a large-scale key-value store. In *Proceedings of the 12th ACM SIGMETRICS/PERFORMANCE Joint International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '12, page 53–64, New York, NY, USA, 2012. Association for Computing Machinery.

[7] Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. Workload analysis of a large-scale key-value store. In *Proceedings of the 12th ACM SIGMETRICS/PERFORMANCE joint international conference on Measurement and Modeling of Computer Systems*, pages 53–64, 2012.

[8] Sorav Bansal and Dharmendra S. Modha. CAR: Clock with adaptive replacement. In *3rd USENIX Conference on File and Storage Technologies (FAST 04)*, San Francisco, CA, March 2004. USENIX Association.

[9] Benjamin Berg, Daniel S. Berger, Sara McAllister, Isaac Grosof, Sathya Gunasekar, Jimmy Lu, Michael Uhlar, Jim Carrig, Nathan Beckmann, Mor Harchol-Balter, and Gregory R. Ganger. The CacheLib caching engine: Design and experiences at scale. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 753–768. USENIX Association, November 2020.

[10] Aaron Blankstein, Siddhartha Sen, and Michael J. Freedman. Hyperbolic caching: Flexible caching for web applications. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, pages 499–511, Santa Clara, CA, July 2017. USENIX Association.

[11] Jeff Bonwick. The slab allocator: An Object-Caching kernel. In *USENIX Summer 1994 Technical Conference (USENIX Summer 1994 Technical Conference)*, Boston, MA, June 1994. USENIX Association.

[12] Lee Breslau, Pei Cao, Li Fan, Graham Phillips, and Scott Shenker. On the implications of zipf's law for web caching. Technical report, University of Wisconsin-Madison Department of Computer Sciences, 1998.

[13] Lee Breslau, Pei Cao, Li Fan, Graham Phillips, and Scott Shenker. Web caching and zipf-like distributions: Evidence and implications. In *IEEE INFOCOM'99. Conference on Computer Communications. Proceedings. Eighteenth Annual Joint Conference of the IEEE Computer and Communications Societies. The Future is Now (Cat. No. 99CH36320)*, volume 1, pages 126–134. IEEE, 1999.

[14] Nathan Bronson, Zach Amsden, George Cabrera, Prasad Chakka, Peter Dimov, Hui Ding, Jack Ferris, Anthony Giardullo, Sachin Kulkarni, Harry Li, et al. {TAO}:{Facebook's} distributed data store for the social graph. In *2013 USENIX Annual Technical Conference (USENIX ATC 13)*, pages 49–60, 2013.

[15] Irina Calciu, M. Talha Imran, Ivan Puddu, Sanidhya Kashyap, Hasan Al Maruf, Onur Mutlu, and Aasheesh Kolli. Rethinking software runtimes for disaggregated memory. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '21, page 79–92, New York, NY, USA, 2021. Association for Computing Machinery.

[16] Tianqi Chen and Carlos Guestrin. Xgboost: A scalable tree boosting system. In *Proceedings of the 22nd acm sigkdd international conference on knowledge discovery and data mining*, pages 785–794, 2016.

[17] Google Cloud. Tulip: Building a smarter, leading-edge retail platform, 2024. Accessed: 2024-06-15.

[18] The Pokemon Company. The pokemon company migrates to aws purpose-built databases, 2024. Accessed: 2024-06-15.

[19] Brian F Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with ycsb. In *Proceedings of the 1st ACM symposium on Cloud computing*, pages 143–154, 2010.

[20] Asit Dan and Donald F. Towsley. An approximate analysis of the LRU and FIFO buffer replacement schemes. In *Proceedings of the 1990 ACM SIGMETRICS conference on Measurement and modeling of computer systems (SIGMETRICS'90)*, pages 143–152, 1990.

[21] Sudipto Das, Shoji Nishimura, Divyakant Agrawal, and Amr El Abbadi. Albatross: lightweight elasticity in shared storage databases for the cloud using live data

migration. *Proc. VLDB Endow.*, 4(8):494–505, may 2011.

[22] Gil Einziger, Roy Friedman, and Ben Manes. Tinylfu: A highly efficient cache admission policy. *ACM Trans. Storage*, 13(4), nov 2017.

[23] Aaron J. Elmore, Sudipto Das, Divyakant Agrawal, and Amr El Abbadi. Zephyr: live migration in shared nothing databases for elastic cloud platforms. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data*, SIGMOD '11, page 301–312, New York, NY, USA, 2011. Association for Computing Machinery.

[24] Jason Evans. A scalable concurrent malloc(3) implementation for freebsd. Technical report, FreeBSD Project, April 2006. Presented at the 2006 BSDCan conference.

[25] Ohad Eytan, Danny Harnik, Effi Ofer, Roy Friedman, and Ronen Kat. It's time to revisit LRU vs. FIFO. In *12th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 20)*. USENIX Association, July 2020.

[26] Zhisheng Hu, Pengfei Zuo, Yizou Chen, Chao Wang, Junliang Hu, and Ming-Chang Yang. Aceso: Achieving efficient fault tolerance in memory-disaggregated key-value stores. In *Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles*, SOSP '24, page 127–143, New York, NY, USA, 2024. Association for Computing Machinery.

[27] Yu Hua, Hong Jiang, Yifeng Zhu, Dan Feng, and Lei Tian. Smartstore: a new metadata organization paradigm with semantic-awareness for next-generation file systems. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, SC '09, New York, NY, USA, 2009. Association for Computing Machinery.

[28] Qi Huang, Ken Birman, Robbert van Renesse, Wyatt Lloyd, Sanjeev Kumar, and Harry C. Li. An analysis of facebook photo caching. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, page 167–181, New York, NY, USA, 2013. Association for Computing Machinery.

[29] Yibo Huang, Haowei Chen, Newton Ni, Yan Sun, Vijay Chidambaram, Dixin Tang, and Emmett Witchel. Tigon: A distributed database for a CXL pod. In *19th USENIX Symposium on Operating Systems Design and Implementation (OSDI 25)*, Boston, MA, July 2025. USENIX Association.

[30] Yibo Huang, Newton Ni, Vijay Chidambaram, Emmett Witchel, and Dixin Tang. Pasha: An efficient, scalable database architecture for CXL pods. In *15th Conference on Innovative Data Systems Research, CIDR 2025, Amsterdam, The Netherlands, January 19-22, 2025*. www.cidrdb.org, 2025.

[31] Song Jiang, Feng Chen, and Xiaodong Zhang. CLOCK-Pro: An effective improvement of the CLOCK replacement. In *2005 USENIX Annual Technical Conference (USENIX ATC 05)*, Anaheim, CA, April 2005. USENIX Association.

[32] Song Jiang and Xiaodong Zhang. Lirs: an efficient low inter-reference recency set replacement policy to improve buffer cache performance. In *Proceedings of the 2002 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '02, page 31–42, New York, NY, USA, 2002. Association for Computing Machinery.

[33] Theodore Johnson and Dennis Shasha. 2q: A low overhead high performance buffer management replacement algorithm. In *Proceedings of the 20th International Conference on Very Large Data Bases*, VLDB '94, page 439–450, San Francisco, CA, USA, 1994. Morgan Kaufmann Publishers Inc.

[34] Anuj Kalia, Michael Kaminsky, and David G. Andersen. Using rdma efficiently for key-value services. In *Proceedings of the 2014 ACM Conference on SIGCOMM*, SIGCOMM '14, page 295–306, New York, NY, USA, 2014. Association for Computing Machinery.

[35] Yuyuan Kang and Ming Liu. Understanding and profiling NVMe-over-TCP using ntprof. In *22nd USENIX Symposium on Networked Systems Design and Implementation (NSDI 25)*, pages 1117–1136, Philadelphia, PA, April 2025. USENIX Association.

[36] Tom Kilburn, David BG Edwards, Michael J Lanigan, and Frank H Sumner. One-level storage system. *IRE Transactions on Electronic Computers*, (2):223–235, 1962.

[37] Ana Klimovic, Yawen Wang, Patrick Stuedi, Animesh Trivedi, Jonas Pfefferle, and Christos Kozyrakis. Pocket: Elastic ephemeral storage for serverless analytics. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 427–444, 2018.

[38] Donghee Lee, Jongmoo Choi, Jong-Hun Kim, Sam H. Noh, Sang Lyul Min, Yookun Cho, and Chong-Sang Kim. On the existence of a spectrum of policies that subsumes the least recently used (LRU) and least frequently used (LFU) policies. In *Proceedings of the 1999 ACM SIGMETRICS international conference on*

Measurement and modeling of computer systems (SIGMETRICS'99), pages 134–143, 1999.

[39] Sekwon Lee, Soujanya Ponnapalli, Sharad Singhal, Marcos K. Aguilera, Kimberly Keeton, and Vijay Chidambaram. Dinomo: An elastic, scalable, high-performance key-value store for disaggregated persistent memory. *Proc. VLDB Endow.*, 15(13):4023–4037, sep 2022.

[40] Seung-seob Lee, Yanpeng Yu, Yupeng Tang, Anurag Khandelwal, Lin Zhong, and Abhishek Bhattacharjee. Mind: In-network memory management for disaggregated data centers. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, SOSP '21, page 488–504, New York, NY, USA, 2021. Association for Computing Machinery.

[41] Cheng Li, Philip Shilane, Fred Douglis, and Grant Wallace. Pannier: A container-based flash cache for compound objects. In *Proceedings of the 16th Annual Middleware Conference*, pages 50–62, 2015.

[42] Huaicheng Li, Daniel S. Berger, Lisa Hsu, Daniel Ernst, Pantea Zardoshti, Stanko Novakovic, Monish Shah, Samir Rajadnya, Scott Lee, Ishwar Agarwal, Mark D. Hill, Marcus Fontoura, and Ricardo Bianchini. Pond: Cxl-based memory pooling systems for cloud platforms. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, ASPLOS 2023, page 574–587, New York, NY, USA, 2023. Association for Computing Machinery.

[43] Pengfei Li, Yu Hua, Pengfei Zuo, Zhangyu Chen, and Jiajie Sheng. ROLEX: A scalable RDMA-oriented learned Key-Value store for disaggregated memory systems. In *21st USENIX Conference on File and Storage Technologies (FAST 23)*, pages 99–114, Santa Clara, CA, February 2023. USENIX Association.

[44] Jinshu Liu, Hamid Hadian, Hanchen Xu, Daniel S Berger, and Huaicheng Li. Dissecting cxl memory performance at scale: Analysis, modeling, and optimization. *arXiv preprint arXiv:2409.14317*, 2024.

[45] Xuchuan Luo, Jiacheng Shen, Pengfei Zuo, Xin Wang, Michael R Lyu, and Yangfan Zhou. Chime: A cache-efficient and high-performance hybrid index on disaggregated memory. In *Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles*, pages 110–126, 2024.

[46] Xuchuan Luo, Pengfei Zuo, Jiacheng Shen, Jiazhen Gu, Xin Wang, Michael R. Lyu, and Yangfan Zhou. SMART: A High-Performance adaptive radix tree for disaggregated memory. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*, pages 553–571, Boston, MA, July 2023. USENIX Association.

[47] Wenhao Lv, Youyou Lu, Yiming Zhang, Peile Duan, and Jiwu Shu. InfiniFS: An efficient metadata service for Large-Scale distributed filesystems. In *20th USENIX Conference on File and Storage Technologies (FAST 22)*, pages 313–328, Santa Clara, CA, February 2022. USENIX Association.

[48] Deborah T Marr, Frank Binns, David L Hill, Glenn Hinton, David A Koufaty, J Alan Miller, and Michael Upton. Hyper-threading technology architecture and microarchitecture. *Intel Technology Journal*, 6(1):4–15, 2002.

[49] Hasan Al Maruf, Hao Wang, Abhishek Dhanotia, Johannes Weiner, Niket Agarwal, Pallab Bhattacharya, Chris Petersen, Mosharaf Chowdhury, Shobhit Kanaujia, and Prakash Chauhan. Tpp: Transparent page placement for cxl-enabled tiered-memory. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, pages 742–755, 2023.

[50] Sara McAllister, Benjamin Berg, Julian Tutuncu-Macias, Juncheng Yang, Sathya Gunasekar, Jimmy Lu, Daniel S Berger, Nathan Beckmann, and Gregory R Ganger. Kangaroo: Caching billions of tiny objects on flash. In *Proceedings of the ACM SIGOPS 28th symposium on operating systems principles*, pages 243–262, 2021.

[51] Nimrod Megiddo and Dharmendra S. Modha. ARC: A Self-Tuning, low overhead replacement cache. In *2nd USENIX Conference on File and Storage Technologies (FAST 03)*, San Francisco, CA, March 2003. USENIX Association.

[52] Antoine Murat, Clément Burgelin, Athanasios Xygkis, Igor Zablotchi, Marcos Kawazoe Aguilera, and Rachid Guerraoui. Swarm: Replicating shared disaggregated-memory data in no time. In *Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles*, pages 24–45, 2024.

[53] Dushyanth Narayanan, Austin Donnelly, and Antony Rowstron. Write Off-Loading: Practical power management for enterprise storage. In *6th USENIX Conference on File and Storage Technologies (FAST 08)*, San Jose, CA, February 2008. USENIX Association.

[54] Rajesh Nishtala, Hans Fugal, Steven Grimm, Marc Kwiatkowski, Herman Lee, Harry C. Li, Ryan McElroy, Mike Paleczny, Daniel Peek, Paul Saab, David Stafford, Tony Tung, and Venkateshwaran Venkataramani. Scaling memcache at facebook. In *10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*, pages 385–398, Lombard, IL, April 2013. USENIX Association.

[55] Ruoyu Qin, Zheming Li, Weiran He, Jialei Cui, Feng Ren, Mingxing Zhang, Yongwei Wu, Weimin Zheng, and Xinran Xu. Mooncake: Trading more storage for less computation—a {KVCache-centric} architecture for serving {LLM} chatbot. In *23rd USENIX Conference on File and Storage Technologies (FAST 25)*, pages 155–170, 2025.

[56] Redis. Key eviction. https://redis.io/docs/latest/develop/reference/eviction/. Accessed: 2024-12-08.

[57] Redis. Memory optimization. https://redis.io/docs/latest/operate/oss_and_stack/management/optimization/memory-optimization/. Accessed: 2024-12-08.

[58] Charles Reiss, Alexey Tumanov, Gregory R. Ganger, Randy H. Katz, and Michael A. Kozuch. Heterogeneity and dynamicity of clouds at scale: Google trace analysis. In *Proceedings of the Third ACM Symposium on Cloud Computing*, SoCC '12, New York, NY, USA, 2012. Association for Computing Machinery.

[59] Liana V. Rodriguez, Farzana Yusuf, Steven Lyons, Eysler Paz, Raju Rangaswami, Jason Liu, Ming Zhao, and Giri Narasimhan. Learning cache replacement with CACHEUS. In *19th USENIX Conference on File and Storage Technologies (FAST 21)*, pages 341–354. USENIX Association, February 2021.

[60] Mohammad Shahrad, Rodrigo Fonseca, Inigo Goiri, Gohar Chaudhry, Paul Batum, Jason Cooke, Eduardo Laureano, Colby Tresness, Mark Russinovich, and Ricardo Bianchini. Serverless in the wild: Characterizing and optimizing the serverless workload at a large cloud provider. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pages 205–218. USENIX Association, July 2020.

[61] Yizhou Shan, Yutong Huang, Yilun Chen, and Yiying Zhang. LegoOS: A disseminated, distributed OS for hardware resource disaggregation. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 69–87, Carlsbad, CA, October 2018. USENIX Association.

[62] Jiacheng Shen, Pengfei Zuo, Xuchuan Luo, Yuxin Su, Jiazhen Gu, Hao Feng, Yangfan Zhou, and Michael R Lyu. Ditto: An elastic and adaptive memory-disaggregated caching system. In *Proceedings of the 29th Symposium on Operating Systems Principles*, pages 675–691, 2023.

[63] Jiacheng Shen, Pengfei Zuo, Xuchuan Luo, Tianyi Yang, Yuxin Su, Yangfan Zhou, and Michael R. Lyu. FUSEE: A fully Memory-Disaggregated Key-Value store. In *21st USENIX Conference on File and Storage Technologies (FAST 23)*, pages 81–98, Santa Clara, CA, February 2023. USENIX Association.

[64] Zhenyu Song, Daniel S. Berger, Kai Li, and Wyatt Lloyd. Learning relaxed belady for content distribution network caching. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 529–544, Santa Clara, CA, February 2020. USENIX Association.

[65] Lalith Suresh, Marco Canini, Stefan Schmid, and Anja Feldmann. C3: Cutting tail latency in cloud data stores via adaptive replica selection. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*, pages 513–527, Oakland, CA, May 2015. USENIX Association.

[66] Linpeng Tang, Qi Huang, Wyatt Lloyd, Sanjeev Kumar, and Kai Li. {RIPQ}: Advanced photo caching on flash for facebook. In *13th USENIX Conference on File and Storage Technologies (FAST 15)*, pages 373–386, 2015.

[67] Shin-Yeh Tsai, Yizhou Shan, and Yiying Zhang. Disaggregating persistent memory and controlling them remotely: An exploration of passive disaggregated Key-Value stores. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pages 33–48. USENIX Association, July 2020.

[68] Giuseppe Vietri, Liana V. Rodriguez, Wendy A. Martinez, Steven Lyons, Jason Liu, Raju Rangaswami, Ming Zhao, and Giri Narasimhan. Driving cache replacement with ML-based LeCaR. In *10th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 18)*, Boston, MA, July 2018. USENIX Association.

[69] Carl A. Waldspurger, Nohhyun Park, Alexander Garthwaite, and Irfan Ahmad. Efficient MRC construction with SHARDS. In *13th USENIX Conference on File and Storage Technologies (FAST 15)*, pages 95–110, Santa Clara, CA, February 2015. USENIX Association.

[70] Chenxi Wang, Haoran Ma, Shi Liu, Yuanqi Li, Zhenyuan Ruan, Khanh Nguyen, Michael D. Bond, Ravi Netravali, Miryung Kim, and Guoqing Harry Xu. Semeru: A Memory-Disaggregated managed runtime. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 261–280. USENIX Association, November 2020.

[71] Qing Wang, Youyou Lu, and Jiwu Shu. Sherman: A write-optimized distributed b+tree index on disaggregated memory. In *Proceedings of the 2022 International Conference on Management of Data*, SIGMOD '22, page 1033–1048, New York, NY, USA, 2022. Association for Computing Machinery.

[72] Qizhen Weng, Wencong Xiao, Yinghao Yu, Wei Wang, Cheng Wang, Jian He, Yong Li, Liping Zhang, Wei Lin, and Yu Ding. MLaaS in the wild: Workload analysis and scheduling in Large-Scale heterogeneous GPU clusters. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, pages 945–960, Renton, WA, April 2022. USENIX Association.

[73] Daniel Lin-Kit Wong, Hao Wu, Carson Molder, Sathya Gunasekar, Jimmy Lu, Snehal Khandkar, Abhinav Sharma, Daniel S. Berger, Nathan Beckmann, and Gregory R. Ganger. Baleen: ML admission & prefetching for flash caches. In *22nd USENIX Conference on File and Storage Technologies (FAST 24)*, pages 347–371, Santa Clara, CA, February 2024. USENIX Association.

[74] Juncheng Yang, Ziming Mao, Yao Yue, and K. V. Rashmi. Gl-cache: Group-level learning for efficient and high-performance caching. In *21st USENIX Conference on File and Storage Technologies (FAST 23)*, pages 115–134, 2023.

[75] Juncheng Yang, Yao Yue, and K. V. Rashmi. A large scale analysis of hundreds of in-memory cache clusters at twitter. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association, November 2020.

[76] Juncheng Yang, Yao Yue, and Rashmi Vinayak. Segcache: a memory-efficient and scalable in-memory key-value cache for small objects. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*, pages 503–518. USENIX Association, April 2021.

[77] Juncheng Yang, Yazhuo Zhang, Ziyue Qiu, Yao Yue, and Rashmi Vinayak. Fifo queues are all you need for cache eviction. In *Proceedings of the 29th Symposium on Operating Systems Principles*, SOSP '23, page 130–149, New York, NY, USA, 2023. Association for Computing Machinery.

[78] Xinjun Yang, Yingqiang Zhang, Hao Chen, Feifei Li, Gerry Fan, Yang Kong, Bo Wang, Jing Fang, Yuhui Wang, Tao Huang, Wenpu Hu, Jim Kao, and Jianping Jiang. Unlocking the potential of cxl for disaggregated memory in cloud-native databases. In *Companion of the 2025 International Conference on Management of Data*, SIGMOD/PODS '25, page 689–702, New York, NY, USA, 2025. Association for Computing Machinery.

[79] Ming Zhang, Yu Hua, and Zhijun Yang. Motor: Enabling {Multi-Versioning} for distributed transactions on disaggregated memory. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, pages 801–819, 2024.

[80] Ming Zhang, Yu Hua, Pengfei Zuo, and Lurong Liu. FORD: Fast one-sided RDMA-based distributed transactions for disaggregated persistent memory. In *20th USENIX Conference on File and Storage Technologies (FAST 22)*, pages 51–68, Santa Clara, CA, February 2022. USENIX Association.

[81] Mingxing Zhang, Teng Ma, Jinqi Hua, Zheng Liu, Kang Chen, Ning Ding, Fan Du, Jinlei Jiang, Tao Ma, and Yongwei Wu. Partial failure resilient memory management system for (cxl-based) distributed shared memory. In *Proceedings of the 29th Symposium on Operating Systems Principles*, SOSP '23, page 658–674, New York, NY, USA, 2023. Association for Computing Machinery.

[82] Yazhuo Zhang, Juncheng Yang, Yao Yue, Ymir Vigfusson, and K.V. Rashmi. SIEVE is simpler than LRU: an efficient Turn-Key eviction algorithm for web caches. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*, pages 1229–1246, Santa Clara, CA, April 2024. USENIX Association.

[83] Yuhong Zhong, Daniel S Berger, Carl Waldspurger, Ishwar Agarwal, Rajat Agarwal, Frank Hady, Karthik Kumar, Mark D Hill, Mosharaf Chowdhury, and Asaf Cidon. Managing memory tiers with cxl in virtualized environments. In *Symposium on Operating Systems Design and Implementation*, 2024.

[84] Tobias Ziegler, Jacob Nelson-Slivon, Viktor Leis, and Carsten Binnig. Design guidelines for correct, efficient, and scalable synchronization using one-sided rdma. *Proceedings of the ACM on Management of Data*, 1(2):1–26, 2023.

[85] Pengfei Zuo, Jiazhao Sun, Liu Yang, Shuangwu Zhang, and Yu Hua. One-sided RDMA-Conscious extendible hashing for disaggregated memory. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, pages 15–29. USENIX Association, July 2021.

ARTIFACT EVALUATED
USENIX
AVAILABLE

Accelerating Confidential Databases with Crypto-free Mappings

Wenxuan Huang Zhanbo Wang Mingyu Li

Key Laboratory of System Software (Chinese Academy of Sciences)
Institute of Software, Chinese Academy of Sciences
University of Chinese Academy of Sciences

Abstract

Confidential databases (CDBs) enable secure queries over sensitive data in untrusted cloud environments using confidential computing hardware. While adoption is growing, widespread deployment is hindered by high overheads from frequent synchronous cryptographic operations, which cause significant computational and I/O bottlenecks.

ZENO is a novel CDB design that removes cryptographic operations from the critical path. It introduces crypto-free mappings that maintain data-independent identifiers within the database while securely mapping them to plaintext secrets in a trusted domain. This paradigm shift yields substantial performance gains across industry-standard benchmarks (TPC-C, TPC-H) and a real-world industrial workload. Specifically, ZENO speeds up TPC-H queries by up to 53.1× on ARM S-EL2 and 94.7× on x86 TDX compared to HEDB. ZENO's optimization techniques have been integrated into GaussDB.

1 Introduction

Cloud databases have become the backbone of modern digital infrastructure, with organizations increasingly outsourcing their data management to Database-as-a-Service (DBaaS) providers [7, 10, 14]. However, this convenience comes with a critical security risk: cloud administrators have unrestricted access to all tenant data, leaving opportunities for both malicious and accidental exposure. Recent breaches, such as the Allianz Life exposure of 1.1 million personal records [20], underscore this threat. For sensitive databases regulated by laws such as HIPAA [5] and PCI-DSS [4], entrusting cloud infrastructure with plaintext data is untenable.

Confidential databases (CDBs) have emerged as a compelling solution. By leveraging trusted execution environments (TEEs), CDBs enable secure data upload and query execution while keeping data encrypted from the cloud provider. Over the past two decades, researchers have explored various CDB designs [25, 28, 42, 65, 73], and major cloud providers have deployed CDB systems [24, 45, 56, 74, 77].

Despite this industry momentum, CDB systems face a critical barrier to widespread adoption: severe performance degradation compared to plaintext databases, with analytical queries experiencing up to 79.5× higher latency [56]. This gap arises from a deliberate system design choice in industry deployments: rather than enclosing the entire DBMS within a TEE, which would prevent database administrators from performing essential maintenance tasks, modern CDBs execute only security-critical operators (e.g., numeric `SUM` operations) inside TEEs. The resulting split architecture necessitates synchronous software-based decryption and encryption for every remote procedure call (RPC) across domains, as illustrated in Figure 1. The computational burden scales linearly with data volume. For instance, a `SUM` aggregation over an N-row table requires $2N$-2 decryptions and N-1 encryptions in total. Even worse, to support arbitrary computation across fields, modern CDBs employ field-level encryption, inflating each 4-byte integer to 32 bytes (8× expansion) due to cryptographic metadata (a 12-byte nonce and a 16-byte authentication tag required by schemes like AES-GCM). This dramatically amplifies cross-domain data movement, memory pressure, and I/O bandwidth demands. These limitations raise a fundamental question: **Is cryptographic overhead inherent to CDB design, or can we achieve equivalent security guarantees with far cheaper alternatives?**

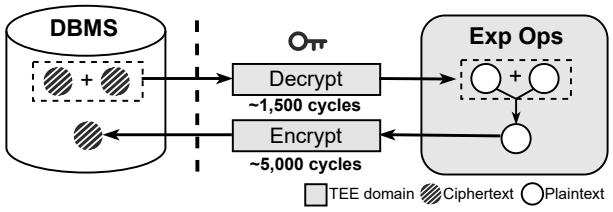


Figure 1: In modern CDBs, a cross-domain invocation involves two decryptions and one encryption, incurring high computational overhead (CPU cycles evaluated on ARM).

Key insight. We argue that the performance crisis stems from an unexamined assumption that has persisted for decades: *any data leaving TEEs must be encrypted.* Modern CDBs store ciphertexts in the DBMS and decrypt them in TEEs for compu-

Table 1: Comparison between modern CDBs and ZENO (CPU cycles evaluated on ARM).

	Modern CDBs	ZENO
Indirection	Ciphertexts	Data-independent FIDs
Protection	Ciphertexts	Encryption at rest
Indirection cost	Decrypt (∼1,500 cycles) Encrypt (∼5,000 cycles)	Get (∼30 cycles) Put (∼200 cycles)

tation. We observe that this is fundamentally a mapping-based protection scheme: the DBMS uses ciphertexts as pointers to reference plaintext data in TEEs. Current practice conflates indirection with protection, encrypting both pointers and data. We decouple these concerns: pointers use lightweight indirection; only data at rest uses encryption. Based on this insight, we aim to replace the costly ciphertext-to-plaintext mappings with highly efficient index-to-plaintext mappings, while preserving both data confidentiality and database correctness.

Our approach. We present ZENO, a CDB system that eliminates compute-intensive cryptography from the critical path without sacrificing security. ZENO introduces *crypto-free mappings* that bind data-independent field identifiers (FIDs) to their corresponding plaintext fields within TEEs. This allows the DBMS to manipulate data as before, while replacing slow per-field en/decryption with ultra-low-latency put/get operations, as shown in Table 1. In addition, to avoid encrypting every field synchronously, ZENO defers encryption: when data must be evicted from TEE memory to untrusted storage, ZENO performs asynchronous block-level encryption rather than fine-grained field-level encryption. In short, crypto-free mappings deliver two key benefits: they remove cryptographic operations from query execution hot paths and reduce metadata overhead via coarser-grained encryption.

Our design brings two main challenges. First, as datasets scale, the mapping lookup time could grow unbounded and ultimately exceed the cost of constant-time cryptographic operations. Second, maintaining transaction semantics and mapping consistency requires new mechanisms because both domains are now stateful, a situation not encountered in conventional CDB systems.

To overcome the first challenge, ZENO introduces a scalable, low-latency data structure called the *mapping store*. This mapping store provides $O(1)$ access complexity by treating FIDs as direct array indices, eliminating the need for tree traversals or hash-table probing. ZENO further preserves data locality across domains through storage-layout-aware mapping store organization. This layout awareness, combined with proactive prefetching, keeps the corresponding data resident in TEE memory when the DBMS accesses an FID block, thereby minimizing additional I/O overhead.

The second challenge requires a subtle observation: the state of the mapping store is internal to the system and invisible to the DBMS. The DBMS observes only the FIDs it maintains—the system's external state. This asymmetry allows us to adopt external synchrony [61]: only FIDs maintained by the DBMS should be mapped to valid, up-to-date plaintexts; the reverse mapping (plaintexts to FIDs) need not even exist. As a result, ZENO only needs to maintain the invariant that every FID in the DBMS has a corresponding valid value in the mapping store. To uphold this invariant across transactions and crashes, ZENO introduces lightweight coordination protocols that ensure cross-domain updates remain consistent and crash-tolerant with minimal overhead.

We evaluate ZENO against HEDB [56], a state-of-the-art modern CDB that achieves the trifecta of functionality, maintainability, and security. Our results across both ARM S-EL2 and x86 TDX show that ZENO achieves substantial performance improvements while preserving all desired properties. On the transactional benchmark TPC-C, ZENO reduces HEDB's throughput loss relative to plaintext PostgreSQL by up to 49.8% on ARM S-EL2 and 73.8% on x86 TDX. On the analytical benchmark TPC-H, ZENO speeds up query execution by up to 53.1× on ARM S-EL2 and 94.7× on x86 TDX over HEDB. On real-world industrial queries, ZENO outperforms HEDB with average speedups of 4.6× on ARM S-EL2 and 2.9× on x86 TDX. Additionally, ZENO reduces storage consumption by 38.9%, 52.8%, and 42.4% across the three workloads, respectively. The main performance techniques of ZENO have been integrated into GaussDB.

Contributions. We make the following contributions:

- We identify and analyze the performance bottlenecks of modern CDBs through comprehensive profiling and empirical analysis.
- We propose a novel CDB design with crypto-free mappings, which remove costly cryptographic operations from the critical path of existing modern CDBs.
- We build and deploy ZENO, which outperforms state-of-the-art modern CDBs across transactional and analytical benchmarks, as well as a real-world industrial workload.

2 Background and Motivation

2.1 Confidential Computing

Confidential computing [9, 12, 16, 17, 22, 41, 47, 52, 55, 57] provides hardware-based trusted execution environments (TEEs) with three security guarantees. First, secure isolation through either enclaves [16, 41, 52] or confidential virtual machines (CVMs) [12, 17, 22] protects against inspection or tampering by untrusted privileged software. Second, hardware root-of-trust enables clients to verify code authenticity via remote attestation. Third, memory encryption and optional integrity checks resist physical attacks. Confidential computing is now widely available on commercial clouds [2, 6, 9, 13, 15].

2.2 Confidential Databases

Database-as-a-Service (DBaaS) [46] offers fully managed databases that handle updates, backups, and maintenance

for users. However, since cloud environments are untrusted, sensitive data like healthcare and financial information requires secure storage and processing. Confidential databases (CDBs) [24, 25, 28, 42, 45, 56, 65, 67, 73, 74, 77] leverage confidential computing to enable SQL query processing over encrypted data without compromising confidentiality.

Existing CDB systems can be broadly categorized into two types. One type [28, 65, 67, 77] places the entire DBMS inside the trusted domain (TEE), where the DBMS processes data in plaintext and encrypts it when interacting with untrusted environments (e.g., network or storage). However, this type imposes an inherent limitation: the black-box TEE hinders DBMS maintainability for database administrators (DBAs) [24, 56]. In contrast, another type [24, 45, 56, 73, 74] adopts a split architecture: only essential expression operators (e.g., arithmetic, comparison[1]) execute inside TEEs, while most DBMS logic (e.g., SQL parsing, planning, buffer management) remains in the untrusted domain. This minimizes the TCB and preserves maintainability: DBAs can inspect query plans and debug DBMS issues. We refer to the latter as *"modern CDBs"* throughout the paper.

To enable flexible computation across arbitrary fields, modern CDBs apply field-level encryption. For example, since all fields from the column `Salary` are encrypted, `"SELECT SUM(Salary) FROM Employee"` requires the DBMS to iteratively call the `addition` operator inside the TEE, which decrypts operands, computes the sum, re-encrypts the result, and returns it to the DBMS. While secure, this design incurs substantial performance overhead, as we will analyze later.

HEDB [56]: a representative modern CDB. Modern CDBs could be vulnerable to *smuggle attacks* [56]: malicious DBAs exploit operator interfaces to construct known ciphertexts (via arithmetic operators, e.g., division yields encrypted "one," addition builds sequences) and compare them against victim ciphertexts (via equality operators), thereby extracting secrets without authorization. HEDB addresses smuggle attacks through a dual-TEE architecture: an *integrity zone* that hosts the DBMS and a *privacy zone* that hosts operators, with a secure channel preventing direct interface access by DBAs. For maintainability, HEDB records cross-zone calls as ciphertext, enabling DBAs to replay and debug queries without plaintext access. Therefore, ZENO builds on HEDB's security foundation. Notably, the performance limitations addressed by ZENO exist across many split-architecture CDBs [24, 45, 73, 74], and its crypto-free mappings generalize beyond HEDB.

2.3 What Makes Modern CDB Systems Slow?

Modern CDBs exemplified by HEDB have demonstrated significant performance overheads. Benchmarks on TPC-H show that HEDB can incur up to a 79.5× increase in latency compared to a plaintext database [56]. These overheads render modern CDBs impractical for latency-sensitive applications, hindering broader industry adoption. Our analysis attributes them to two main factors.

[1]All operators yield encrypted results, except for comparisons (e.g., $<$, $=$, $>$), which output plaintext boolean values for use in indexing ciphertexts.

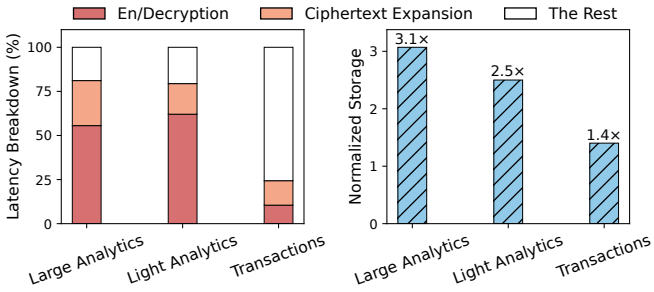


Figure 2: Latency and storage overhead analysis across three typical workloads. **Left (a)**: profiling end-to-end SQL execution time; the rest denotes normal DBMS execution and RPC invocations. **Right (b)**: storage overhead normalized to plaintext databases.

Factor-#1: expensive encryption and decryption. In modern CDBs, each computation on sensitive data requires a cross-domain round-trip, which involves two decryptions and one encryption, each taking thousands of cycles, as shown in Table 1. For an operation like `SUM` on a table with N rows, the computation incurs $N-1$ round-trips, resulting in $2N-2$ decryptions and $N-1$ encryptions. Given that N is often *tens of millions* in real-world scenarios (reported by several CDB vendors we surveyed), this high frequency of expensive cryptographic operations inevitably introduces high overhead on the critical path. While batching multiple RPC invocations into a single round trip (e.g., for aggregation operators such as `SUM`, `AVG`, `MAX`, and `MIN`) can help amortize overhead, the core issue remains that cryptographic operations must still be performed on every single data item.

Factor-#2: ciphertext expansion. Modern CDBs employ field-level encryption with AES-GCM. To guarantee data confidentiality and integrity, each encrypted field requires a nonce (12 bytes) and message authentication code (MAC, 16 bytes), totaling 28 B of cryptographic metadata. This introduces significant ciphertext expansion: a 4-byte integer expands to 32 bytes, an 8× size increase. This expansion creates two performance bottlenecks. First, it enlarges the data transferred during frequent inter-domain RPCs, resulting in higher memory copy latency. Second, it increases the overall storage footprint of the database, leading to more I/O operations between the DBMS buffer pool and storage.

Empirical analysis. We profile HEDB [56] (commit f48742de) using three workloads: (i) TPC-H Q1 with a dataset larger than memory, representing large table computation; (ii) TPC-H Q1 with a dataset fitting in memory, capturing lightweight analytical settings; and (iii) a transactional benchmark with many write-heavy small queries. Despite the batch-

ing optimization, HEDB exhibits significantly higher overhead, with an overall latency slowdown of up to 5.3× relative to the plaintext baseline. As illustrated by the performance breakdown in Figure 2(a), en/decryption operations and ciphertext expansion account for a substantial fraction of the total execution time, contributing 10.5–62.6% and 13.9–25.5%, respectively, across workloads. Specifically, the contribution of I/O amplification is more pronounced in (i) than in (ii). This shift shows that the overhead distribution varies with the workload, particularly with respect to data scale and memory residency. Meanwhile, the storage overhead (Figure 2(b)) reveals that field-level encryption incurs a 1.4× to 3.1× storage expansion relative to the plaintext baseline.

Strawman: a ciphertext-to-plaintext cache. Since confidential VM-based TEEs (e.g., Intel TDX [17], AMD SEV-SNP [22]) provide sufficiently large secure memory, one may consider maintaining a ciphertext-to-plaintext cache within TEEs to remove cryptographic operations from the critical path. However, as the cache scales with the protected dataset, searching a large ciphertext-keyed structure can itself become a bottleneck. As our ablation study shows (§7.2), such a cache helps reduce cryptographic cost but remains limited by the overhead of ciphertext-based lookups.

3 ZENO Overview

Design goals. ZENO has two design goals:

- G1: *Compatibility.* ZENO aims to be compatible with modern CDB designs by preserving their functionality (standard SQL and ACID), while retaining HEDB-level DBA maintainability and security (data confidentiality and database integrity).
- G2: *Efficiency.* ZENO aims to reduce the performance gap between existing modern CDBs and plaintext databases by optimizing the two factors identified in § 2.3.

Threat model. We assume a powerful adversary who controls the entire cloud infrastructure except TEE hardware. This includes compromised server software, rogue cloud and database administrators, and external attackers. Adversaries can read and modify any data outside TEE domains—data at rest, in untrusted memory, and in transit—and observe all network traffic and I/O patterns visible outside TEE domains.

We assume TEEs provide two essential properties: *memory isolation*, preventing external access to TEE-internal state, and *remote attestation*, enabling users to verify code integrity before establishing trust. While TEE-agnostic, ZENO requires these guarantees to ensure sensitive plaintext data remains visible only within TEE domains. We do not consider TEE side-channel attacks (e.g., [36, 69, 72, 76]); these can be addressed by orthogonal hardening techniques [33, 59, 81]. Nor do we consider database vulnerabilities [11], TEE defects [50], physical attacks [51], or denial-of-service.

Functional leakage inherent to split-architecture CDBs is

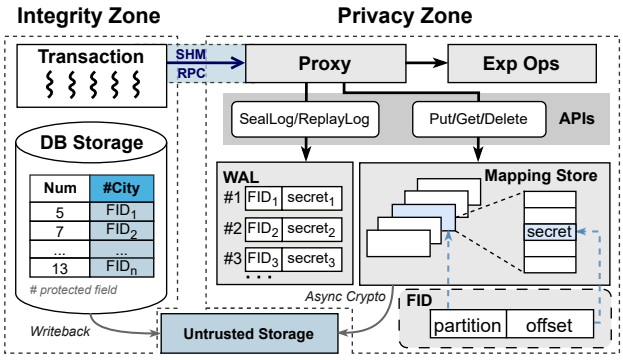


Figure 3: High-level architecture of ZENO.

permitted. This reflects a fundamental trade-off between functionality and privacy: exposing certain metadata is sometimes necessary to support core database functionalities, such as order for index construction [56]. Because our design substitutes ciphertexts with FIDs, we additionally account for any metadata exposed through this indirection mechanism; its security implications are analyzed in § 8.

Insights. Current practice in modern CDBs employs a mapping-based protection scheme, using ciphertexts as pointers to plaintext data in TEEs. This design conflates two orthogonal concerns: *indirection* (cross-domain references) and *protection* (data confidentiality). Because ciphertexts serve both roles, this coupling necessitates expensive en/decryption on every RPC. ZENO decouples them: FIDs provide efficient indirection, while encryption protects data at rest.

In fact, separating indirection from protection is a long-standing systems principle. Capability systems [39, 53] used unforgeable tokens protected by OS confinement, rather than cryptography. Similarly, OSes have long used opaque handles, e.g., Unix file descriptors [21, 68], which are mapped internally by the kernel. In both cases, security comes from address-space isolation, keeping references simple and fast.

Architecture. ZENO's high-level architecture is depicted in Figure 3. Following the security principles of HEDB's dual-zone design, ZENO redesigns how protected values flow between zones. The integrity zone runs a commodity DBMS engine, and the privacy zone comprises four components: (1) a proxy that performs mapping operations and invokes the appropriate operators, (2) expression operators that perform computation over plaintexts, (3) the mapping store that manages the FID-to-secret mappings, and (4) a write-ahead log (WAL) for cross-domain consistency. The mapping store is divided into temporary and permanent partitions for fine-grained management. The proxy interacts with the remaining components through a set of well-defined APIs.

APIs. ZENO provides five APIs to manage the mapping store:

- `Put(partition_num, secret)` → `FID`: Stores the given secret in the specified partition of the mapping store.

Returns an FID with the partition number embedded, as a reference to the stored secret.

- `Get(FID)` → `secret`: Retrieves the secret associated with the specified FID, or returns a null value if the FID has no corresponding mapping.
- `Delete(FID)`: Invalidates the specified FID-to-secret mapping in the mapping store, marking the entry as available for future allocation.
- `SealLog(records)` → `encrypted_records`: Encrypts the given mapping-store WAL records and returns the encrypted records.
- `ReplayLog(encrypted_records)`: Replays the specified encrypted mapping-store WAL records to reconstruct FID-to-secret mappings.

At the operator interface, existing modern CDBs [24, 45, 56, 73, 74] can benefit from ZENO's gains by replacing per-field en/decryption with `Put()`/`Get()` calls (see Listing 1).

```
/* expression operators (unchanged) */
int int_add(int l, int r) { return l + r; }

/* proxy code snippet */
- EncInt enc_int(Ops ops, EncInt left, EncInt right)
+ FID enc_int(Ops ops, FID left, FID right)
{
- int l = Decrypt(left), r = Decrypt(right);
+ int l = Get(left), r = Get(right);
  int result;
  switch (ops) {
      case ADD: result = int_add(l, r); break;
      // other cases...
  }
- return Encrypt(result);
+ // use temporary partitions for intermediate results
+ return Put(TMP_PARTITION_NUM, result);
}
```

Listing 1: Example code for extending the original expression operator using ZENO's APIs.

Workflow. ZENO's query execution follows the workflow of modern CDB architectures. To perform computation, the proxy fetches secrets associated with each input FID from the mapping store via `Get()`, invokes the appropriate expression operator to compute over these retrieved secrets, and then stores the result back to the mapping store via `Put()`, which generates a new FID representing the output. The DBMS proceeds with query execution using this returned FID. Finally, the proxy resolves FIDs back into ciphertexts by calling `Get()` followed by encrypting the retrieved results within the privacy zone before returning them to users. Throughout this process, FIDs remain opaque to users. This design preserves *compatibility* (G1) with the client-side encryption model used by modern CDBs [24, 45, 56, 74].

Challenges. Since ZENO introduces state into the privacy zone, it brings two challenges. First, the lookup latency in the mapping store may grow with dataset size and eventually exceed constant-time cryptographic operations (§ 4). Second, maintaining state consistency between the integrity and privacy zones is necessary for database semantics (§ 5).

4 Achieving Efficiency via Mapping Store

This section details the mapping store, which plays a crucial role in achieving *efficiency* (G2).

4.1 Mapping Management

Data structure design. The mapping store is a linear array of tightly packed fields. Each sensitive field is assigned a unique integer FID that serves as a direct array index, hiding the size of individual fields from the adversary. This design choice offers two advantages. First, employing a compact integer FID effectively restricts the metadata overhead per field to a mere 8 B, representing a 71.4% reduction compared to the 28 B metadata required by modern CDBs. Second, the use of integer-based direct indexing enables $O(1)$, cache-friendly array access to the mapping store, thereby eliminating the pointer-chasing and lookup overhead inherent in tree-based or hash-based key-value stores (e.g., [35, 40, 43]).

Mapping store operations. For fixed-length types (e.g., `FLOAT8`, `TIMESTAMP`), `Delete()` marks the corresponding mapping slots as reusable, and `Put()` either overwrites these invalidated slots or appends to the end. For variable-length types (e.g., `VARCHAR`, `NUMERIC`), the situation is subtle because values cannot be overwritten in place. We therefore maintain two arrays: an *offset array* and a *data array*. The offset array is indexed directly by FIDs and stores pointers into the data array. The data array is inspired by the slab allocator [31]: fields are grouped into size-class buckets, each maintained as a contiguous packed array to minimize fragmentation under frequent allocation and deallocation. Within each bucket, insertions and deletions follow the same overwrite/append logic as the fixed-length case.

Partition-based state management. The mapping store cannot directly infer the lifetime of each field. For instance, when an expression operator produces a result, it is unclear whether the value will be discarded after use or eventually persisted. Careless reclamation risks leaving live FIDs pointing to freed storage, causing data loss. To address this, we classify values into two categories: ephemeral values (e.g., query constants and intermediate results) and persistent values (e.g., data that outlives query completion). We separate their management by mapping them to distinct mapping store partitions: a temporary partition for ephemeral values and permanent partitions for persistent values.

For ephemeral values, reclamation occurs throughout their lifecycle. They are first placed in the temporary partition via `Put()` with the temporary partition identifier `TMP_PARTITION_NUM`. If the DBMS later decides to persist such a value, ZENO intercepts the write path and performs another `Put()` with the permanent partition identifier. During this `Put()` call, ZENO allocates a new persistent FID, copies the value into the permanent partition, and returns the new FID to the DBMS. To avoid accumulating stale values in the tempo-

rary partition, ZENO reclaims ephemeral values in the privacy zone as soon as they are known to be dead, e.g., temporary operands consumed by aggregation or temporary FIDs that have already been migrated to the permanent partition. At the end of each query, any values still remaining in the temporary partition, i.e., ephemeral values whose lifetime extends to the query boundary, can be safely discarded.

For persistent values, reclamation is triggered when the DBMS instructs the proxy to invoke `Delete()` on mappings associated with specific FIDs, e.g., during maintenance operations such as PostgreSQL `VACUUM`, which removes dead tuples.

4.2 Exploiting Data Locality

Mismatches in data access patterns between the DBMS and mapping store can cause *major page faults* on the critical path. For example, when the DBMS accesses an FID whose corresponding secret is not in memory, ZENO must perform an extra I/O in the privacy zone, thereby increasing latency. To reduce such latency, we make the mapping store organization storage-layout-aware and proactively prefetch secrets likely to be accessed by the DBMS. This is achieved by exploiting both temporal and spatial locality.

Exploiting temporal locality. In ZENO, the DBMS first retrieves the FID from storage, and then resolves it through the mapping store to obtain the corresponding secret. This lookup dependency creates a natural temporal alignment in access patterns between the DBMS and the mapping store. For instance, when a query accesses fields associated with $\text{FID}_1, \ldots, \text{FID}_n$ in sequence, the mapping store performs the same sequence to retrieve $\text{secret}_1, \ldots, \text{secret}_n$. This dependency also enables lock-efficient management for the mapping store. Accesses to the same allocated FID do not introduce conflicts, because they are already coordinated by the DBMS's concurrency control. As a result, ZENO synchronizes only during FID allocation, while reads and slot reuse need no additional locking.

Exploiting spatial locality. To further improve performance, ZENO organizes the mapping store into separate partitions based on the storage layout of the underlying database. The partitioning policy is customizable depending on the specific DBMS or storage engine. For example, in PostgreSQL, each table is a separate heap file with unordered field placement and reclamation. ZENO assigns one mapping store partition per heap file to preserve field colocation. In InnoDB, data resides in globally ordered fixed-size pages (e.g., 16 KB) with unordered fields inside each page. ZENO assigns one partition per page to preserve page-level locality. We encode the partition number in the high bits of FID to segment the FID space.

Prefetching. A random lookup in ZENO may involve two serial I/O operations: one for the DBMS to fetch FIDs and another for the mapping store to retrieve secrets. ZENO reduces this overhead by maximizing I/O overlap. Specifically, when the DBMS issues a disk read, ZENO intercepts it and identifies the partition being accessed. Since ZENO uses storage-layout-aware mapping store partitioning, it proactively sends an RPC to the proxy specifying which partition will be accessed. The mapping store then prefetches the corresponding partition concurrently with the DBMS's I/O operation.

5 Ensuring Cross-domain Consistency

Modern CDBs rely on the underlying DBMS to provide ACID properties. ZENO must preserve the same guarantees. However, doing so in ZENO is more subtle because transaction-relevant state now spans two isolated domains, requiring new mechanisms to keep these states consistent throughout the transaction lifecycle.

For cross-domain correctness, the key constraint is that the DBMS interacts with the privacy zone only through FIDs; it cannot directly inspect the mapping store state. Therefore, correctness reduces to a one-way invariant: *every DBMS-maintained FID maps to a valid, up-to-date plaintext in the mapping store*. In contrast, orphan plaintexts in the mapping store are permissible because they are unreachable from DBMS-visible state. This asymmetry naturally aligns with *external synchrony* [61], a consistency model that distinguishes between internal system state and externally visible state. In ZENO, the mapping store constitutes the internal state, while the FIDs maintained by the DBMS constitute the external state with respect to the privacy zone.

This observation pinpoints the moments that require cross-domain coordination: when the set of DBMS-visible FIDs changes, or when such visibility must be preserved across failures. These cases include aborts, where old FIDs become visible again after rollback; commits, where new FIDs become durable; and recovery, where state is restored after failures. Accordingly, ZENO introduces lightweight synchronization between the database and the mapping store at transaction boundaries and during recovery. We describe these mechanisms next.

Handling transaction aborts. When a transaction aborts, rollback may make old FIDs DBMS-visible again. To preserve the invariant, ZENO must keep the mappings for these old FIDs until the DBMS itself determines that they are no longer visible.

To this end, ZENO employs a multi-version concurrency control (MVCC)-like approach: stale values (e.g., logically deleted entries and pre-update versions) are retained at runtime and reclaimed only when the underlying DBMS runs garbage collection (e.g., PostgreSQL's off-path `VACUUM`). For instance, as illustrated in Figure 4(a), when a transaction issues an update, ZENO retains the original mapping entry intact (①) and inserts the new value out-of-place (②). If the transaction is aborted, the DBMS can access the original values as usual (③). The aborted value is eventually reclaimed by the DBMS's off-path garbage collection process (④).

Handling transaction commits. Commits must preserve the external-synchrony invariant across crashes. A privacy-zone crash may lose recently created secrets, while the DBMS can still recover the committed database state. After recovery, the DBMS may contain committed tuples with FIDs whose corresponding secrets are no longer recoverable in the mapping store. To prevent this case, the commit protocol extends external synchrony to crash-recoverable state: if an FID is recoverable after a crash, its corresponding secret must also be. Since both PostgreSQL and the mapping store recover committed state from their WALs, enforcing this invariant reduces to a commit-time WAL-ordering rule: for each transaction, mapping-store WAL updates must be durable before PostgreSQL persists its WAL. A straightforward design would maintain a separate mapping-store WAL and synchronously flush it before PostgreSQL persists its own WAL. Although correct, this design is costly: under synchronous commit, the mapping-store flush becomes an additional serialized I/O on the critical path. This also forces ZENO to track a separate global order for mapping-store WAL records, duplicating PostgreSQL's WAL ordering and increasing commit contention.

To avoid these costs, ZENO co-designs mapping recovery with PostgreSQL's WAL recovery by embedding mapping-store WAL records into the PostgreSQL WAL stream. At commit time, ZENO calls `SealLog()` to generate a block-granular encrypted mapping-store WAL record containing the newly exposed mappings from the transaction. It then adds this record to PostgreSQL's WAL stream before PostgreSQL emits the transaction commit record. Only after this insertion does PostgreSQL continue its normal commit path, placing the resulting WAL records in LSN order and flushing them using its native WAL machinery. To further reduce commit-time overhead, ZENO pipelines and parallelizes block-granular encryption: the privacy zone encrypts full blocks as they accumulate during transaction execution and handles multiple blocks in parallel. This protocol is therefore lightweight, adding only one RPC and residual block-granular encryption to the commit critical path.

Recovering from system crashes. After a system crash, ZENO must restore mappings for all FIDs recovered by PostgreSQL. Recovery is driven by PostgreSQL's standard WAL replay. As PostgreSQL replays the WAL stream, ZENO identifies embedded mapping-store WAL records and then calls `ReplayLog()` to reconstruct the corresponding mappings.

Figure 4(b) illustrates three recovery outcomes. In case ①, the crash occurs before PostgreSQL flushes the WAL stream; both the mapping-store WAL record and the transaction commit record are lost, so no new FIDs become visible after recovery. In case ③, the transaction commit record is durable; by order, the preceding mapping-store WAL record is also durable, so ZENO can reconstruct mappings for all recovered DBMS-visible FIDs. Thus, both cases preserve the external-synchrony invariant. Case ② is the only intermediate case: the

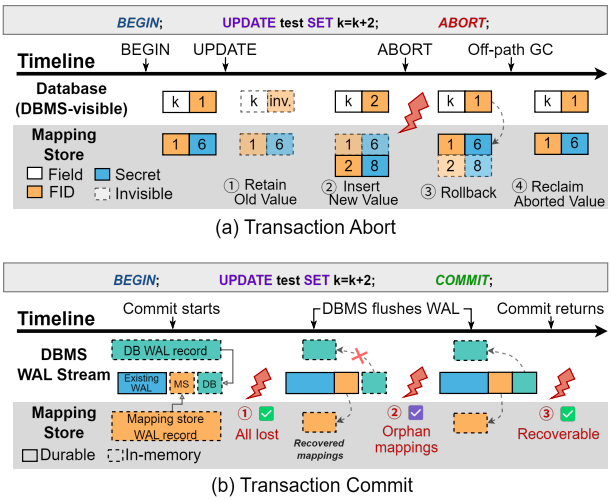


Figure 4: Timeline of a transaction execution in ZENO, illustrating the handling of (a) abort and (b) commit paths.

system crashes during WAL flushing, after the mapping-store WAL record is written back but before the transaction commit record is completely written to disk. WAL replay may reconstruct these mappings, but PostgreSQL will not expose their FIDs in the recovered database state. This is safe because the recovered mappings do not overwrite pre-existing visible data: as described in the abort protocol, ZENO only reuses FIDs after the DBMS has garbage-collected their previous references. Therefore, the reconstructed mappings are merely orphans and do not constitute an inconsistent state. They remain unreachable from DBMS-visible state, preserving consistency; ZENO can reclaim them later via a global scan.

6 Implementation

We implemented ZENO on top of PostgreSQL 15.5 in 6.1K lines of C/C++ code. Table 2 provides a detailed breakdown by component. ZENO runs atop CVMs (i.e., Intel TDX [17] and ARM Secure EL2 [8, 55]) and deploys its integrity and privacy zones in isolated VM domains. To reduce cross-domain RPC overhead, ZENO adopts a context-switchless implementation [24, 26, 75]: the integrity zone sends requests to secure shared memory, while the privacy zone polls the shared memory and dispatches requests for processing. The integrity zone uses dm-integrity [19] for corruption detection, while the privacy zone uses dm-crypt [18] with HMAC-SHA256 for transparent I/O encryption and authentication. The mapping store is backed by `mmap`ed arrays for persistence. Temporal locality is exploited by an LRU cache combined with Linux sequential prefetching; spatial locality is improved by organizing permanent partitions at table granularity. Within each partition, ZENO assigns FIDs by reusing safely reclaimed slots or appending new ones via an atomic monotonic counter. Currently, each 64-bit FID encodes a configurable 16-bit partition prefix and a 48-bit offset. Users can also tune FID length

Table 2: Lines of code for each component in ZENO.

ZENO Components	Language	LoC
Mapping Store	C++	1,074
WAL	C++	461
Commit Protocol (in PostgreSQL source)	C	550
Proxy	C++	2,155
Operators (modified from HEDB source)	C++	1,708
Crypto Library	C/C++	110

and partition-prefix size according to expected data volume: shorter FIDs reduce storage overhead and improve lookup efficiency. An open-source version of ZENO is available at https://github.com/ISCAS-OSLab/ZENO.

7 Performance Evaluation

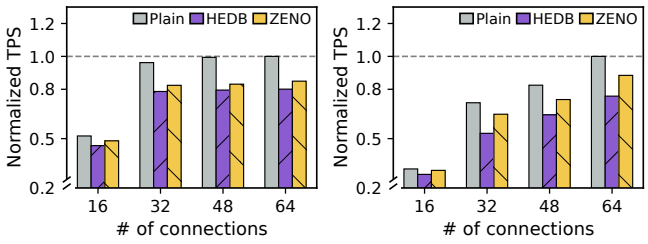
We evaluate ZENO by answering the following questions:

- RQ1: Does ZENO deliver end-to-end performance improvements across platforms and workloads? (§ 7.1)
- RQ2: What are the sources of ZENO's performance gains and consistency overheads? (§ 7.2)
- RQ3: How robust is ZENO under different workloads and resource configurations? (§ 7.3)

Experimental setup. We evaluate ZENO on two platforms. The ARM platform runs on a HiSilicon Kunpeng-920 @ 2.6 GHz server with S-EL2 support, equipped with a 1 TB SSD and Ubuntu 22.04 LTS, hosting two Secure-EL2 VMs running Linux 6.8.0. The x86 platform runs on an Intel Xeon Platinum 8581C @ 4.0 GHz server with TDX support, equipped with a 4 TB SSD and Ubuntu 24.04 LTS.

For both platforms, ZENO and HEDB deploy the integrity and privacy zones in two isolated VM domains with a total resource budget of 32 vCPUs and 64 GB of memory. Under this budget, we sweep several vCPU and memory splits between the two zones for each experiment and use the best-performing split for each system. Both zones disable memory swapping. To ensure resource fairness, baseline plaintext PostgreSQL is deployed in a virtual machine with 32 vCPUs and 64 GB of memory. Across all systems, the benchmark client uses 16 vCPUs and 32 GB of memory. To reduce cross-NUMA RPC latency and hypervisor scheduling noise, we pin all VMs to the same NUMA node when possible, or otherwise to nearby NUMA nodes with low NUMA distance.

For ARM S-EL2, the shared memory for inter-CVM RPCs is protected by the TrustZone hypervisor. For x86 TDX, since no CVM vendors [17, 22] currently provide secure channels between CVMs (their shared memory remains plaintext), we exploit TD Partitioning [3], which allows multiple L2 VMs to run within one CVM. However, TD Partitioning is documented primarily for single-L2-VM deployments, offering little engineering guidance for multi-L2 setups. As such, we adopt a microkernel-inspired deployment: the privacy zone runs in the L1 VM and the integrity-zone DBMS (given its large TCB) runs in an L2 VM managed by the L1, with RPC shared memory controlled by the L1 to prevent malicious host introspection.



(a) ARM S-EL2 (b) x86 TDX

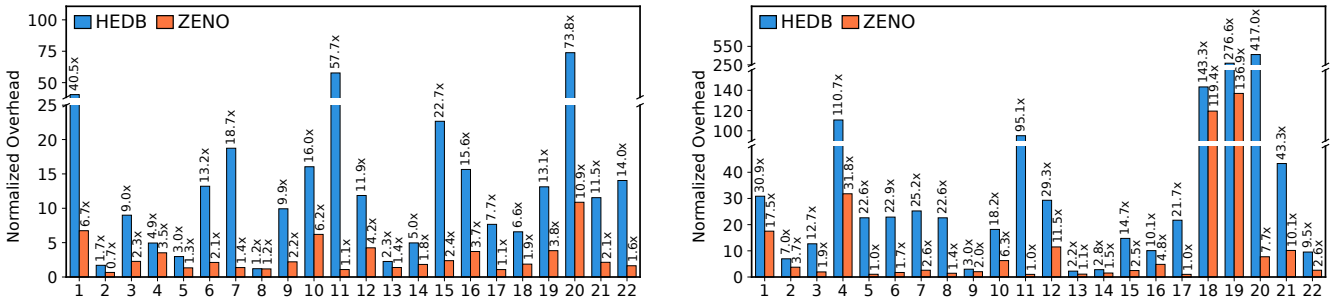
Figure 5: TPC-C throughput with varying numbers of connections at $W = 100$, normalized to the peak plaintext PostgreSQL TPS on each platform (higher is better). ZENO reduces HEDB's throughput loss by up to 49.8% on ARM S-EL2 and 73.8% on x86 TDX.

7.1 End-to-End Performance (RQ1)

We use two standard benchmarks: TPC-C and TPC-H. Since these standard benchmarks are not privacy-oriented, we further use a real-world workload from the industry. For comparison, we include vanilla PostgreSQL as the plaintext baseline and the state-of-the-art HEDB.

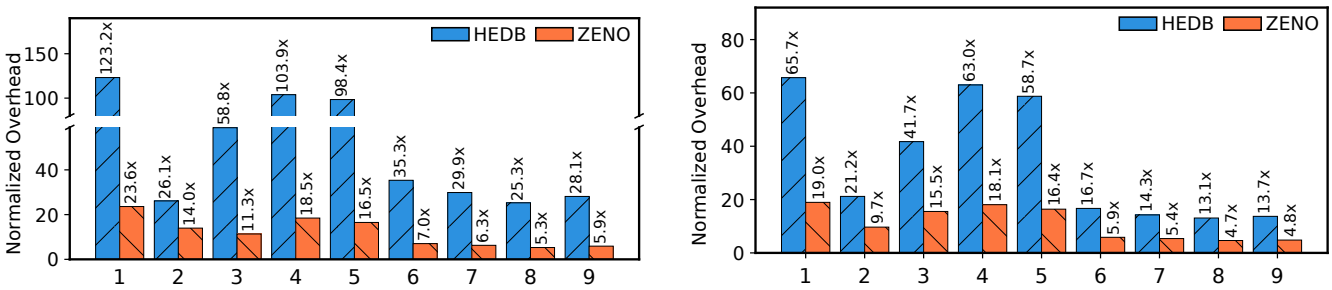
TPC-C. We evaluate TPC-C throughput (transactions per second, or TPS) under varying connection counts using 100 warehouses, treating all non-ID columns as sensitive and placing the database on tmpfs. After a 60-second warm-up, we measure for 300 seconds and normalize TPS to the peak plaintext PostgreSQL throughput across all connection counts. As shown in Figure 5, ZENO narrows the throughput gap between HEDB and plaintext PostgreSQL on both platforms. We report this gap as throughput loss relative to plaintext PostgreSQL at the same connection count. On ARM S-EL2, ZENO reduces HEDB's throughput loss by 18.1–49.8%; on x86 TDX, ZENO reduces HEDB's throughput loss by 51.5–73.8%. The two platforms exhibit different scaling trends due to platform-specific hardware and TEE-stack characteristics.

TPC-H. We use TPC-H (scale factor = 3), a standard OLAP benchmark with large joins, group-bys, aggregations, and arithmetic operations. We make all non-key columns sensitive, as TPC-H does not define security attributes. To evaluate performance under large-dataset execution, we use a 6 GB memory budget. PostgreSQL and HEDB each use the full 6 GB as DBMS-side memory, while ZENO splits the same total budget between the integrity and privacy zones, with 4.5 GB and 1.5 GB, respectively. Results are averaged over 5 runs per query after one warm-up run. Figure 6 shows the slowdown of HEDB and ZENO relative to plaintext PostgreSQL. HEDB incurs an average slowdown of 10.0× on ARM S-EL2 and 23.8× on x86 TDX, driven by excessive cryptographic



(a) ARM S-EL2 (b) x86 TDX

Figure 6: TPC-H execution time of HEDB and ZENO normalized to plaintext PostgreSQL. ZENO substantially reduces HEDB's overhead, achieving speedups of up to 53.1× on ARM S-EL2 and up to 94.7× on x86 TDX.



(a) ARM S-EL2 (b) x86 TDX

Figure 7: Execution time of 9 industrial queries on ARM S-EL2 and x86 TDX, normalized to that of plaintext PostgreSQL. ZENO consistently outperforms HEDB, achieving up to 6.0× and 3.6× speedups on ARM S-EL2 and x86 TDX, respectively.

Table 3: Overall storage usage (in GB) across three workloads. ZENO reduces storage cost by up to 52.8% compared to HEDB.

Database	TPC-C (W=100)	TPC-H (SF=3)	Industrial
Plaintext	10.3	5.2	1.5
HEDB	22.1 (2.1×)	17.8 (3.4×)	3.3 (2.2×)
ZENO	13.5 (1.3×)	8.4 (1.6×)	1.9 (1.3×)

operations and I/O amplification due to ciphertext expansion. In contrast, ZENO restricts the average slowdown to 2.3× on ARM S-EL2 and 4.5× on x86 TDX, outperforming HEDB by 4.4× and 5.3×, respectively. The performance discrepancy is primarily attributable to the Intel TDX implementation, which relies on bounce buffers (i.e., Linux SWIOTLB) for I/O [54]. Consequently, I/O-intensive queries experience higher overhead on x86 TDX compared to ARM S-EL2.

Industrial workload. We evaluate ZENO using 9 real-world complex queries under strict confidentiality constraints. Schemas are anonymized prior to evaluation. We generate synthetic data matching the anonymized schemas at 1 million rows per table, with all columns treated as sensitive. Figure 7 reports the end-to-end execution time normalized to plaintext PostgreSQL. ZENO outperforms HEDB on both ARM S-EL2 and x86 TDX. On ARM S-EL2, ZENO reduces HEDB's 48.4× average slowdown to 10.5× (up to 6.0× speedup); on x86 TDX, it reduces 27.6× to 9.4× (up to 3.6× speedup). These results confirm ZENO's performance gains on real-world industrial workloads.

Storage consumption. As shown in Table 3, ZENO reduces storage consumption by 38.9–52.8% compared to HEDB across all evaluated workloads. This is because ZENO adds an 8-byte FID per value, while existing modern CDBs attach 28 bytes of cryptographic metadata per field.

7.2 Breakdown and Overhead Analysis (RQ2)

To address RQ2, we use TPC-H and TPC-C to isolate different performance factors. TPC-H is ideal for decomposing ZENO's gains; its heavy aggregations and large data volumes expose the specific overheads of cryptographic computation and ciphertext expansion. For aspects not covered by TPC-H, specifically the commit protocol and crash recovery, we use TPC-C to measure their overhead.

Unless otherwise stated, the following experiments use our ARM S-EL2 VM setup. These studies address platform-independent design questions. Fixing one primary platform avoids conflating ZENO's design effects with platform-specific factors such as I/O-path behaviors (e.g., bounce buffers), TEE virtualization overheads, and scheduling.

Factor analysis. We perform a cumulative factor analysis on TPC-H to characterize the sources of ZENO's performance gains. Starting from unoptimized HEDB, we incrementally

enable HEDB's optimization and then introduce ZENO's optimization techniques one by one. Figure 8 reports the geometric mean execution time normalized to plaintext PostgreSQL. Below, all reported reductions and speedups are relative to the preceding variant.

+*Batching* groups aggregation inputs for operators such as `SUM` and `AVG`. Instead of invoking the privacy zone for every intermediate accumulation step, the integrity zone sends a batch of fields per RPC (batch size = 256), reducing RPC frequency and intermediate cryptographic operations. Batching reduces the execution time by 5.8% and speeds up individual queries by up to 1.6×.

+*Decryption cache* represents an optimized version of the ciphertext-to-plaintext cache discussed in § 2. It caches decryption results in the privacy zone, using `boost_unordered_flat_map` [32] with `XXH3_64bits` [37] and a 512 MB cache shared across SQL statements. The decryption cache further reduces the execution time by 21.8% and speeds up individual queries by up to 2.4×. To understand why the gains remain modest, we further instrument per-operation costs under the evaluated workload. A hash-table lookup and an insertion cost 334.9 and 2649.4 cycles on average, respectively, compared with 113.8 and 316.0 cycles for ZENO's direct-indexing `Get()` and `Put()`. Although avoiding repeated decryptions is beneficial, the improvement is bounded by two factors: the relatively high cost of hash-table lookup/insertion operations over ciphertext keys, and the overhead of ciphertext expansion, which remains unchanged.

+*O(1) FID lookup* replaces each ciphertext with a 28-byte padded FID, which is resolved through direct indexing in the mapping store. Together, the padded FID and plaintext value match HEDB's per-field ciphertext footprint, separating the benefit of direct indexing from that of reduced data expansion. O(1) FID lookup reduces execution time by 25.7% and speeds up individual queries by up to 24.7×, confirming that direct indexing substantially lowers privacy-zone processing overhead.

+*Reduced expansion* switches to compact 8-byte FIDs, removing field-level expansion and reducing the data footprint of TPC-H's large, I/O-bound queries. By reducing I/O and memory movement, this step lowers execution time by 53.0% and speeds up individual queries by up to 11.5×.

Finally, +*Partition* enables ZENO's partition-based mapping-store layout. By organizing partitions according to the DBMS storage layout, ZENO improves spatial locality between DBMS-visible FIDs and privacy-zone plaintext values. This optimization further reduces the execution time by 16.6% and speeds up individual queries by up to 6.4×.

Overall, the factor analysis confirms the combined benefits of direct FID lookup, compact field representation, and locality-aware partitioning.

Commit overhead. To capture the runtime overhead of ZENO's commit protocol, we place the database on persistent storage and enable `synchronous_commit`, forcing each transaction commit to wait for WAL durability. We use TPC-C with $W = 100$ and 16 connections for this experiment, and compare ZENO with and without embedded mapping-store WAL records. Enabling mapping-store WAL incurs a modest 2.6% reduction in TPS. To further understand this overhead, we instrument the commit path and directly measure the per-commit cost introduced by ZENO's commit protocol. An invocation producing a 997-byte encrypted payload adds only 18.3 μs to the overall commit latency.

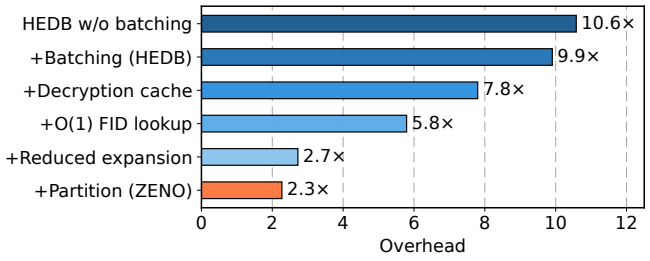


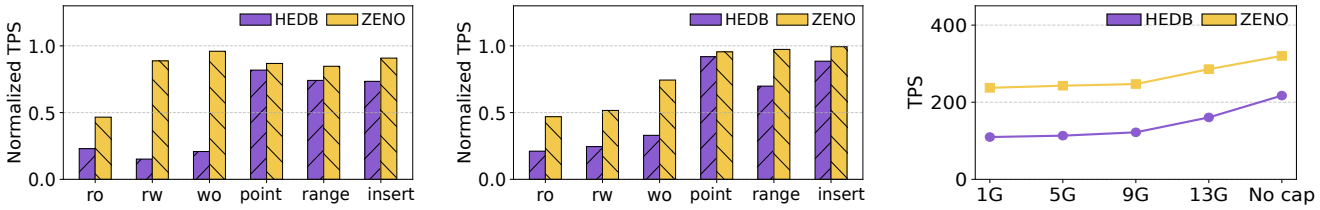
Figure 8: Factor analysis of performance gains across different techniques using TPC-H. Overhead is the execution time normalized to the plaintext baseline.

Recovery overhead. During the initial data loading phase of TPC-C (configured with 8 warehouses and 8 threads), we abruptly terminate the process to simulate a crash, leaving 2.6 GB of WAL for recovery. After restart, PostgreSQL replays the WAL stream that contains both database records and embedded mapping-store records. The complete WAL replay takes 26.3 seconds, with mapping-store record replay accounting for 4.1 seconds (15.6%) of the total time.

7.3 Workload and Resource Sensitivity (RQ3)

The performance gain of ZENO depends on both workload characteristics and system configuration. To characterize the impact of these factors, we use Sysbench [62] to issue transactional microbenchmarks with configurable read/write ratios and access patterns. The benchmark runs on a synthetic dataset consisting of 32 tables with 1M rows each, occupying approximately 7 GB in plaintext PostgreSQL, 10 GB in ZENO, and 12 GB in HEDB. To control for other variables, we conduct all experiments entirely in memory, with the exception of the memory-sensitivity analysis. Each experiment includes a 120-second warm-up followed by a 600-second measurement, using 16 connections by default.

Transaction sensitivity. Figure 9(a) presents normalized throughput under Zipfian data distribution (factor = 0.8) across six operation modes: read-only, read-write, write-only, point-select, range-select and insert-only. Overall, ZENO achieves speedups of 1.1–5.8× over HEDB and delivers 46.6–95.0% of plaintext throughput. In write-only mode, ZENO is 4.6× faster than HEDB. This gap arises from frequent `UPDATE` queries such as "`UPDATE test SET`



(a) Transaction Modes (Zipfian, factor=0.8) (b) Transaction Modes (Uniform) (c) Total Memory Size

Figure 9: Microbenchmark performance under different configurations. ZENO consistently outperforms HEDB across all settings.

`k=k+1;`", where ZENO eliminates the cryptographic overhead in HEDB's decrypt-compute-re-encrypt cycle.

Skew sensitivity. We also evaluate a random access pattern generated from a uniform distribution, which exhibits relatively poor locality. As illustrated in Figure 9(b), even under a non-skewed access pattern that yields frequent CPU cache and TLB misses, ZENO still outperforms HEDB by up to 2.3× across all six modes, thanks to its crypto-free mappings.

Memory sensitivity. Both HEDB and ZENO introduce additional metadata overhead (cryptographic metadata and FIDs, respectively) over PostgreSQL, making them more sensitive to available memory size. To quantify this sensitivity, we measure their performance under different total memory sizes. We use a read-write workload with 8 threads and a random access pattern. As illustrated in Figure 9(c), ZENO outperforms HEDB, with an average speedup of 1.9× and a 2.2× speedup even when the total available memory is limited to 1 GB (10% of the dataset for ZENO). This is because ZENO has a smaller storage footprint than HEDB and preserves locality between the mapping store and the database.

8 Security Analysis

This section analyzes ZENO's security guarantees. Following prior modern CDBs [24, 45, 56, 73, 74], we consider two adversary classes: active adversaries can arbitrarily tamper with storage and modify untrusted memory, and passive adversaries (honest-but-curious) do not modify data but attempt to infer sensitive information through observation.

Defense against active attacks. ZENO ensures that sensitive data remains confidential and fresh. First, TEE hardware provides memory encryption and isolation, preventing adversaries from accessing plaintext data during computation. Second, since FIDs are completely opaque to users, a malicious user cannot access others' private data by crafting an FID. Last, ZENO secures untrusted storage against tampering across both zones. In the privacy zone, ZENO uses dm-crypt [18] with HMAC-SHA256 to provide block-level authenticated encryption and detect storage corruption. In the integrity zone, ZENO enforces full-disk integrity [19] to guard against adversaries who may attempt to swap FIDs across records to gain unauthorized access to a victim's secret. Both zones exploit a counter-based freshness mechanism [1] against rollback attacks [23, 34].

ZENO ensures faithful query execution through two mechanisms. First, remote attestation allows users to verify that legitimate ZENO code is running in the TEEs before uploading data. Second, in our implementation based on HEDB, ZENO places both the DBMS (in the integrity zone) and expression operators (in the privacy zone) within trusted domains, with secure channels between TEEs that prevent message injection, replay, man-in-the-middle, and smuggle attacks.

Defense against passive attacks. The primary concern is whether an adversary observing FIDs can learn anything about the corresponding plaintext secrets. We prove that ZENO achieves indistinguishability via a simple observation: FID allocation is *stateful but plaintext-independent*. The allocator maintains internal state that identifies the next available FID and assigns FIDs deterministically from this state, independent of the plaintext m. Consequently, after n prior `Put()` calls, either plaintext m_0 or m_1, when submitted as the $(n+1)$-th item, receives the exact same FID, so the adversary's view of the FID sequence is indistinguishable between the two cases. As a result, ZENO's crypto-free mappings provide confidentiality comparable to *semantic security* [44]: no information about plaintexts is revealed (beyond leakage from other channels, discussed later).

Additionally, because FIDs are assigned independently of plaintext values, the same plaintext may be represented by multiple distinct FIDs, mirroring the behavior of non-deterministic encryption schemes like AES-GCM.

Leakage analysis. While ZENO protects plaintext contents, practical query functionality requires revealing certain metadata to the untrusted infrastructure. We characterize ZENO's leakage profile using the definition of $\mathcal{L}$-security [63, 73], which models security as a leakage function $\mathcal{L}$ that precisely specifies what information an adversary learns. We define ZENO's $\mathcal{L} = (\mathcal{L}_{init}, \mathcal{L}_{query}, \mathcal{L}_{update})$ as follows:

- $\mathcal{L}_{init}(D)$: During database initialization with dataset D, the adversary observes the database storage layout, including its total volume and the number and positions of protected fields. Since these fields are replaced by constant-size FIDs, individual field sizes remain hidden.
- $\mathcal{L}_{query}(q, D)$: For query q over D, the adversary learns:
  1. *I/O access pattern* $ap(q)$: which storage blocks in the

DBMS and mapping store are accessed during query execution.
2. *Search pattern* $sp(q)$: whether two queries access overlapping record sets, revealing query repetition.
3. *Timing* $t(q)$: when and how long the query executes.
4. *Result volume* $|R_q|$: the number of records returned or affected by the query.
5. *Comparison results*: the boolean outcomes (e.g., "$>$") necessary for indexing and filtering, a common trade-off for many enterprise modern CDBs [24, 45, 56, 74].

- $\mathcal{L}_{update}(op, D)$: For an update operation op (i.e., `INSERT`, `UPDATE`, `DELETE`), the adversary learns which storage blocks are modified and the operation type, but not the plaintext values being written or deleted. Since FIDs are allocated monotonically by default, the adversary may also infer the order in which protected fields are assigned FIDs.

This leakage profile provides $\mathcal{L}$*-semantic security*: an adversary observing ZENO's execution learns nothing beyond $\mathcal{L}$ about the underlying plaintext data.

FID metadata discussion. Like ciphertexts in HEDB, FIDs carry observable metadata about protected fields. First, FIDs expose positional information (via embedded partition numbers), analogous to how HEDB leaks ciphertext position through untrusted storage layout. Second, FIDs reveal allocation order, as they are allocated monotonically by default. This information is also observable in HEDB: newly generated ciphertexts are eventually persisted to untrusted storage, and their appearance in logs or batch flushes can reveal generation order, at least at batch granularity. To coarsen this metadata, ZENO could pre-allocate FIDs in batches and assign them via random permutation within each batch, which balances allocation-order confidentiality, allocation efficiency, and spatial locality.

Side-channel discussion. ZENO's stateful privacy zone retains plaintext data in TEE memory for extended periods (until eviction to storage). This residency could increase the attack window for digital side-channels (e.g., cache timing or memory access patterns). However, modern CDBs already face this risk: encryption keys reside permanently in TEE memory; key compromise would breach all sensitive data. Mitigations such as strict resource isolation [33, 81] apply equally to both residual plaintext and long-lived keys.

9 Related Work

Confidential databases (CDBs). To address rising privacy concerns, CDBs on untrusted clouds have long been a goal [58]. TrustedDB [28], EnclaveDB [65], DBStore [67], and SecuDB [77] shield an entire DBMS in TEEs, providing strong security but limiting maintainability. Modern CDBs, such as AlwaysEncrypted [24], GaussDB [45], Operon [74], and HEDB [56], prioritize maintainability by keeping most DBMS logic outside TEEs, but incur severe overhead from per-field cryptography. Other CDB systems take orthogonal approaches: fully cryptography-based databases [29, 30, 63, 64, 66, 71, 78, 79] require no trusted hardware, and oblivious databases [38, 60, 82] hide access patterns.

Motivated by the growing industrial adoption of split-architecture CDBs, ZENO improves their efficiency through crypto-free mappings. Full resistance to active DBAs in this setting further requires defending against smuggle attacks, which HEDB addresses by isolating DBAs from operator interfaces. Systems such as AlwaysEncrypted, GaussDB, and Operon could adopt ZENO's mapping abstraction as a drop-in performance optimization, yet would still need HEDB-style isolation to achieve equivalent security guarantees.

Indirection-protection separation. Separating indirection from protection is a classic systems principle. Capability systems [39, 53] pioneered the use of unforgeable tokens (capabilities) as references to resources protected by OS isolation. Similarly, OSes have long used opaque handles [21, 68]: file descriptors in Unix are integers mapped by the kernel to internal objects. ZENO shares this insight: data-independent FIDs serve as capabilities protected by TEE isolation, removing cryptographic overhead for every reference.

Applicability beyond CDBs. Compartmentalization-based confidential analytics systems [48, 80] encrypt data at TEE boundaries among operators, incurring significant cryptographic overhead. These systems may benefit from the abstraction of crypto-free mappings for better efficiency.

TEE optimizations. Several techniques have been proposed to optimize TEE-based applications, such as context-switchless calls [24, 26, 75], userspace I/O [27, 70], and other optimizations [49, 56].

10 Conclusion

This paper challenges a core assumption in modern CDB design: mappings between untrusted and trusted domains must be cryptographic. Existing CDB systems using ciphertext-to-plaintext mappings suffer from critical performance bottlenecks. ZENO introduces crypto-free mappings, which eliminate costly en/decryption from the critical path and avoid ciphertext expansion. The integration of ZENO's key performance techniques into a production DBaaS system validates the practicality of this approach.

Acknowledgments

We sincerely thank our shepherd and the anonymous reviewers of OSDI 2026 for their constructive comments. This work was supported in part by the National Natural Science Foundation of China (No. 62502510), and was sponsored by CCF-Huawei Populus Grove Fund. Mingyu Li (limingyu@ios.ac.cn) is the corresponding author.

References

[1] Rollback protection in TF-M secure boot. https://trustedfirmware-m.readthedocs.io/en/latest/design_docs/booting/secure_boot_rollback_protection.html, 2020.

[2] Build with SGX enclaves - Azure Virtual Machines. https://learn.microsoft.com/en-us/azure/confidential-computing/confidential-computing-enclaves, 2022.

[3] Intel® TDX Module v1.5 TD Partitioning Architecture Specification. https://www.intel.com/content/www/us/en/content-details/773039/intel-tdx-module-v1-5-td-partitioning-architecture-specification.html, 2023.

[4] Payment Card Data Security Standard (PCI-DSS). https://www.pcisecuritystandards.org/standards/, 2024.

[5] The HIPAA Privacy Rule. https://www.hhs.gov/hipaa/for-professionals/privacy/index.html, 2024.

[6] Alibaba Cloud - TEE-based confidential computing. https://www.alibabacloud.com/help/en/ack/ack-managed-and-ack-dedicated/user-guide/tee-based-confidential-computing/, 2025.

[7] Amazon Aurora. https://aws.amazon.com/rds/aurora/, 2025.

[8] Arm architecture reference manual for a-profile architecture. https://developer.arm.com/documentation/ddi0487/latest/, 2025.

[9] AWS Nitro Enclaves. https://aws.amazon.com/ec2/nitro/nitro-enclaves/, 2025.

[10] Azure SQL Database. https://azure.microsoft.com/en-us/products/azure-sql/database/, 2025.

[11] Bugs found in Database Management Systems. https://www.manuelrigger.at/dbms-bugs/, 2025.

[12] Confidential Compute Architecture. https://www.arm.com/architecture/security-features/arm-confidential-compute-architecture, 2025.

[13] Confidential VM overview - Google Cloud. https://cloud.google.com/security/products/confidential-computing?hl=en, 2025.

[14] Google Cloud SQL. https://cloud.google.com/sql, 2025.

[15] IBM Cloud - Confidential computing solutions. https://www.ibm.com/solutions/confidential-computing, 2025.

[16] Intel Software Guard Extensions. https://www.intel.com/content/www/us/en/developer/tools/software-guard-extensions/overview.html, 2025.

[17] Intel Trust Domain Extensions. https://www.intel.com/content/www/us/en/developer/articles/technical/intel-trust-domain-extensions.html, 2025.

[18] Linux dm-crypt. https://www.kernel.org/doc/html/latest/admin-guide/device-mapper/dm-crypt.html, 2025.

[19] Linux dm-integrity. https://www.kernel.org/doc/html/latest/admin-guide/device-mapper/dm-integrity.html, 2025.

[20] Recent data breaches 2025: latest cybersecurity incidents & lessons. https://preyproject.com/blog/worst-data-security-breaches, 2025.

[21] Michael J. Accetta, Robert V. Baron, William J. Bolosky, David B. Golub, Richard F. Rashid, Avadis Tevanian, and Michael Young. Mach: A new kernel foundation for UNIX development. In *Proceedings of the USENIX Summer Conference, Atlanta, GA, USA, June 1986*. USENIX Association, 1986.

[22] AMD. AMD Secure Encrypted Virtualization (SEV). https://developer.amd.com/sev/, 2025.

[23] Sebastian Angel, Aditya Basu, Weidong Cui, Trent Jaeger, Stella Lau, Srinath T. V. Setty, and Sudheesh Singanamalla. Nimble: Rollback protection for confidential cloud services. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2023.

[24] Panagiotis Antonopoulos, Arvind Arasu, Kunal D. Singh, Ken Eguro, Nitish Gupta, Rajat Jain, Raghav Kaushik, Hanuma Kodavalla, Donald Kossmann, Nikolas Ogg, Ravi Ramamurthy, Jakub Szymaszek, Jeffrey Trimmer, Kapil Vaswani, Ramarathnam Venkatesan, and Mike Zwilling. Azure SQL database always encrypted. In *Proceedings of the ACM SIGMOD Conference*, 2020.

[25] Arvind Arasu, Spyros Blanas, Ken Eguro, Raghav Kaushik, Donald Kossmann, Ravishankar Ramamurthy, and Ramarathnam Venkatesan. Orthogonal security with cipherbase. In *Proceedings of the Conference on Innovative Data Systems Research (CIDR)*, 2013.

[26] Sergei Arnautov, Bohdan Trach, Franz Gregor, Thomas Knauth, André Martin, Christian Priebe, Joshua Lind, Divya Muthukumaran, Dan O'Keeffe, Mark Stillwell, David Goltzsche, David M. Eyers, Rüdiger Kapitza, Peter R. Pietzuch, and Christof Fetzer. SCONE: secure linux containers with intel SGX. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 689–703. USENIX Association, 2016.

[27] Maurice Bailleu, Jörg Thalheim, Pramod Bhatotia, Christof Fetzer, Michio Honda, and Kapil Vaswani. SPEICHER: securing lsm-based key-value stores using shielded execution. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)*, 2019.

[28] Sumeet Bajaj and Radu Sion. Trusteddb: a trusted hardware based database with privacy and data confidentiality. In *Proceedings of the ACM SIGMOD Conference*, 2011.

[29] Song Bian, Haowen Pan, Jiaqi Hu, Zhou Zhang, Yunhao Fu, Jiafeng Hua, Yunyi Chen, Bo Zhang, Yier Jin, Jin Dong, and Zhenyu Guan. Engorgio: An arbitrary-precision unbounded-size hybrid encrypted database via quantized fully homomorphic encryption. In *Proceedings of the USENIX Security Symposium*, 2025.

[30] Song Bian, Zhou Zhang, Haowen Pan, Ran Mao, Zian Zhao, Yier Jin, and Zhenyu Guan. HE3DB: an efficient and elastic encrypted database via arithmetic-and-logic fully homomorphic encryption. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2023.

[31] Jeff Bonwick et al. The slab allocator: An object-caching kernel memory allocator. In *USENIX summer*, volume 16. Boston, MA, USA, 1994.

[32] Boost C++ Libraries. Boost.Unordered: `boost::unordered_flat_map`. https://www.boost.org/doc/libs/latest/libs/unordered/doc/html/unordered/reference/unordered_flat_map.html, 2026. Last accessed on 2026-05-13.

[33] Charly Castes and Andrew Baumann. Sharing is leaking: blocking transient-execution attacks with core-gapped confidential vms. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2024.

[34] David Chu, Aditya Balasubramanian, Dee Bao, Natacha Crooks, Heidi Howard, Lucky E Katahanas, and Soujanya Ponnapalli. Rollbaccine: Herd Immunity against Storage Rollback Attacks in TEEs. *Proceedings of the ACM SIGMOD Conference*, 2026.

[35] Howard Chu. Mdb: A memory-mapped database and backend for openldap. In *Proceedings of the 3rd International Conference on LDAP, Heidelberg, Germany*, volume 35, page 34, 2011.

[36] Jalen Chuang, Alex Seto, Nicolas Berrios, Stephan van Schaik, Christina Garman, and Daniel Genkin. Tee.fail: Breaking trusted execution environments via ddr5 memory bus interposition. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*. IEEE, 2026.

[37] Yann Collet. xxHash: Extremely fast non-cryptographic hash algorithm. https://xxhash.com/, 2026. Last accessed on 2026-05-13.

[38] Natacha Crooks, Matthew Burke, Ethan Cecchetti, Sitar Harel, Rachit Agarwal, and Lorenzo Alvisi. Obladi: Oblivious serializable transactions in the cloud. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 727–743, 2018.

[39] T. Don Dennis. *A capability architecture*. PhD thesis, Purdue University, USA, 1980.

[40] Facebook. Rocksdb. http://rocksdb.org, 2013.

[41] Erhu Feng, Xu Lu, Dong Du, Bicheng Yang, Xueqiang Jiang, Yubin Xia, Binyu Zang, and Haibo Chen. Scalable memory protection in the PENGLAI enclave. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2021.

[42] Benny Fuhry, Jayanth Jain H. A, and Florian Kerschbaum. Encdbdb: Searchable encrypted, fast, compressed, in-memory database using enclaves. In *Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2021.

[43] Sanjay Ghemawat and Jeff Dean. Leveldb. http://code.google.com/p/leveldb, 2011.

[44] Shafi Goldwasser and Silvio Micali. Probabilistic encryption & how to play mental poker keeping secret all partial information. In *Proceedings of the Fourteenth Annual ACM Symposium on Theory of Computing*, STOC '82, page 365–377.

[45] Liang Guo, Jinwei Zhu, Jiayang Liu, and Kun Cheng. Full encryption: An end to end encryption mechanism in gaussdb. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, 2021.

[46] Hakan Hacigümüs, Sharad Mehrotra, and Balakrishna R. Iyer. Providing database as a service. IEEE Computer Society, 2002.

[47] Guerney D. H. Hunt, Ramachandra Pai, Michael V. Le, Hani Jamjoom, Sukadev Bhattiprolu, Rick Boivie, Laurent Dufour, Brad Frey, Mohit Kapur, Kenneth A. Goldman, Ryan Grimm, Janani Janakirman, John M. Ludden, Paul Mackerras, Cathy May, Elaine R. Palmer, Bharata Bhasker Rao, Lawrence Roy, William A. Starke, Jeff Stuecheli, Enriquillo Valdez, and Wendel Voigt. Confidential computing for openpower. In *Proceedings of the ACM European Conference on Computer Systems (EuroSys)*, 2021.

[48] Byeongwook Kim, Jaewon Hur, Adil Ahmad, and Byoungyoung Lee. Secure data analytics in apache spark with fine-grained policy enforcement and isolated execution. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*. The Internet Society, 2025.

[49] Taehoon Kim, Joongun Park, Jaewook Woo, Seungheun Jeon, and Jaehyuk Huh. Shieldstore: Shielded in-memory key-value storage with sgx. In *Proceedings of the ACM European Conference on Computer Systems (EuroSys)*, 2019.

[50] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. Spectre attacks: Exploiting speculative execution. pages 1–19. IEEE.

[51] Dayeol Lee, Dongha Jung, Ian T. Fang, Chia-Che Tsai, and Raluca Ada Popa. An off-chip attack on hardware enclaves via the memory bus. In *Proceedings of the USENIX Security Symposium*, pages 487–504. USENIX Association, 2020.

[52] Dayeol Lee, David Kohlbrenner, Shweta Shinde, Krste Asanovic, and Dawn Song. Keystone: an open framework for architecting trusted execution environments. In *Proceedings of the ACM European Conference on Computer Systems (EuroSys)*. ACM, 2020.

[53] Henry M Levy. *Capability-based computer systems*. Digital Press, 2014.

[54] Dingji Li, Zeyu Mi, Chenhui Ji, Yifan Tan, Binyu Zang, Haibing Guan, and Haibo Chen. Bifrost: Analysis and optimization of network I/O tax in confidential virtual machines. In *2023 USENIX Annual Technical Conference (USENIX ATC 23)*. USENIX Association, 2023.

[55] Dingji Li, Zeyu Mi, Yubin Xia, Binyu Zang, Haibo Chen, and Haibing Guan. Twinvisor: Hardware-isolated confidential virtual machines for ARM. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, 2021.

[56] Mingyu Li, Xuyang Zhao, Le Chen, Cheng Tan, Huorong Li, Sheng Wang, Zeyu Mi, Yubin Xia, Feifei Li, and Haibo Chen. Encrypted databases made secure yet maintainable. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2023.

[57] Xupeng Li, Xuheng Li, Christoffer Dall, Ronghui Gu, Jason Nieh, Yousuf Sait, and Gareth Stockwell. Design and verification of the arm confidential compute architecture. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2022.

[58] Umesh Maheshwari, Radek Vingralek, and William Shapiro. How to build a trusted database system on untrusted storage. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 135–150. USENIX Association, 2000.

[59] Apostolos Mavrogiannakis, Xian Wang, Ioannis Demertzis, Dimitrios Papadopoulos, and Minos N. Garofalakis. OBLIVIATOR: oblivious parallel joins and other operators in shared memory environments. In *Proceedings of the USENIX Security Symposium*, 2025.

[60] Pratyush Mishra, Rishabh Poddar, Jerry Chen, Alessandro Chiesa, and Raluca Ada Popa. Oblix: An efficient oblivious search index. In *2018 IEEE symposium on security and privacy (SP)*, pages 279–296. IEEE, 2018.

[61] Edmund B Nightingale, Kaushik Veeraraghavan, Peter M Chen, and Jason Flinn. Rethink the sync. *ACM Transactions on Computer Systems (TOCS)*, 2008.

[62] Oracle. Sysbench benchmark tool, 2021.

[63] Rishabh Poddar, Tobias Boelter, and Raluca A. Popa. Arx: An encrypted database using semantically secure encryption. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, 2019.

[64] Raluca A. Popa, Catherine M. S. Redfield, Nickolai Zeldovich, and Hari Balakrishnan. Cryptdb: protecting confidentiality with encrypted query processing. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, 2011.

[65] Christian Priebe, Kapil Vaswani, and Manuel Costa. Enclavedb: A secure database using SGX. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, 2018.

[66] Xuanle Ren, Le Su, Zhen Gu, Sheng Wang, Feifei Li, Yuan Xie, Song Bian, Chao Li, and Fan Zhang. HEDA:

multi-attribute unbounded aggregation over homomorphically encrypted database. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, 2022.

[67] Pedro S. Ribeiro, Nuno Santos, and Nuno O. Duarte. Dbstore: A trustzone-backed database management system for mobile applications. In *International Conference on E-Business and Telecommunication Networks*, 2018.

[68] Dennis M Ritchie and Ken Thompson. The unix time-sharing system. *Communications of the ACM*, 17(7):365–375, 1974.

[69] Benedict Schlüter, Supraja Sridhara, Mark Kuhne, Andrin Bertschi, and Shweta Shinde. {HECKLER}: Breaking confidential {VMs} with malicious interrupts. In *33rd USENIX Security Symposium (USENIX Security 24)*, pages 3459–3476, 2024.

[70] Jörg Thalheim, Harshavardhan Unnibhavi, Christian Priebe, Pramod Bhatotia, and Peter R. Pietzuch. rkt-io: a direct I/O stack for shielded execution. In *Proceedings of the ACM European Conference on Computer Systems (EuroSys)*, 2021.

[71] Stephen Tu, M. Frans Kaashoek, Samuel Madden, and Nickolai Zeldovich. Processing analytical queries over encrypted data. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, 2013.

[72] Stephan van Schaik, Alexander Seto, Thomas Yurek, Adam Batori, Bader AlBassam, Daniel Genkin, Andrew Miller, Eyal Ronen, Yuval Yarom, and Christina Garman. Sok: Sgx.fail: How stuff gets exposed. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*. IEEE, 2024.

[73] Dhinakaran Vinayagamurthy, Alexey Gribov, and Sergey Gorbunov. Stealthdb: a scalable encrypted database with full SQL query support. *Proceedings of the Privacy Enhancing Technologies Symposium (PETS)*, 2019.

[74] Sheng Wang, Yiran Li, Huorong Li, Feifei Li, Chengjin Tian, Le Su, Yanshan Zhang, Yubing Ma, Lie Yan, Yuanyuan Sun, Xuntao Cheng, Xiaolong Xie, and Yu Zou. Operon: An encrypted database for ownership-preserving data management. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, 2022.

[75] Ofir Weisse, Valeria Bertacco, and Todd M. Austin. Regaining lost cycles with hotcalls: A fast interface for SGX secure enclaves. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, 2017.

[76] Yuanzhong Xu, Weidong Cui, and Marcus Peinado. Controlled-channel attacks: Deterministic side channels for untrusted operating systems. pages 640–656. IEEE Computer Society, 2015.

[77] Xinying Yang, Cong Yue, Wenhui Zhang, Yang Liu, Beng Chin Ooi, and Jianjun Chen. Secudb: An in-enclave privacy-preserving and tamper-resistant relational database. 2024.

[78] Zhou Zhang, Song Bian, Zian Zhao, Ran Mao, Haoyi Zhou, Jiafeng Hua, Yier Jin, and Zhenyu Guan. Arcedb: An arbitrary-precision encrypted database via (amortized) modular homomorphic encryption. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, pages 4613–4627. ACM, 2024.

[79] Dongfang Zhao. Hermes: High-performance homomorphically encrypted vector databases. *arXiv preprint arXiv:2506.03308*, 2025.

[80] Wenting Zheng, Ankur Dave, Jethro G Beekman, Raluca Ada Popa, Joseph E Gonzalez, and Ion Stoica. Opaque: An oblivious and encrypted distributed analytics platform. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2017.

[81] Ziqiao Zhou, Yizhou Shan, Weidong Cui, Xinyang Ge, Marcus Peinado, and Andrew Baumann. Core slicing: closing the gap between leaky confidential vms and bare-metal cloud. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 247–267. USENIX Association, 2023.

[82] Jinhao Zhu, Liana Patel, Matei Zaharia, and Raluca Ada Popa. Compass: encrypted semantic search with high accuracy. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 915–938, 2025.

A Artifact Appendix

Abstract

The artifact for this paper contains the source code for ZENO. It also includes scripts for the TPC-C and TPC-H workloads used in the paper, documentation for the two platforms described in § 7, and instructions for building ZENO and running its smoke tests.

Scope

The artifact is intended primarily to support availability and basic functionality checks. It enables users to inspect the ZENO implementation, build the prototype, run the provided

single-machine smoke test, and examine the scripts and documentation for the two evaluated deployment configurations: ARM S-EL2 and x86 TDX. The artifact does not include the proprietary industrial workload used in the paper.

Contents

The artifact has three main parts:

1. The ZENO implementation in `src/`, including the components listed in Table 2. The PostgreSQL integration patch is in `src/integrity_zone/postgresql-patches/` and implements the WAL coordination path used by ZENO's commit protocol. Parts of the protected-operator implementation are derived from HEDB's operator framework.

2. Tests and benchmark scripts. `test/` contains SQL smoke tests; `benchmark/tpcc/` contains plaintext and ZENO TPC-C runners based on sysbench/TPC-C Lua scripts derived from Percona code; and `benchmark/tpch/` contains TPC-H scripts derived from HEDB.

3. Documentation in `README.md` and `doc/`. These files describe the quick-start build, single-machine simulation mode, protected-storage setup, classic two-VM deployment notes, and TD-partitioning deployment notes for x86 TDX.

Third-party code and license notices are summarized in `THIRD_PARTY_LICENSES.md`.

Hosting

The artifact is hosted publicly on GitHub at https://github.com/ISCAS-OSLab/ZENO. The local artifact version used to prepare this appendix corresponds to commit `213036c`.

Requirements

The quick-start configuration requires no special confidential-computing hardware. In this mode, PostgreSQL and the privacy-zone proxy run as two local processes, which is sufficient for build and functional checks.

JANUS: Cross-World, Cooperative Nested Virtualization for Secure Containers

Jiangshan Lai[*,†], Hang Huang[*,‡,§], Quan Xu[‡], Zhen Ren[‡], Wenlong Hou[†], Wei Guo[‡], Jia Rao[¶], Hui Lu[¶], Weidong Han[‡], Jiesheng Wu[‡], Jiang Liu[‡], Naixuan Guan[‡], Yibin Shen[‡], Feng Yu[†], Xu Wang[†], Shiqiang Zhang[‡], Zhiheng Tao[†], Yisheng Xie[‡], Song Wu[§], and Hai Jin[§]
[†]Ant Group [‡]Alibaba Cloud
[§]National Engineering Research Center for Big Data Technology and System, Services Computing Technology and System Lab, Cluster and Grid Computing Lab, School of Computer Science and Technology, Huazhong University of Science and Technology, Wuhan, 430074, China
[¶]The University of Texas at Arlington, USA

Abstract

Secure containers such as Kata Containers strengthen isolation by running each container inside a lightweight VM. On today's virtualized clouds, this creates an unavoidable form of nested virtualization with dominant cost in memory virtualization. Existing approaches entangle CPU virtualization and three-level page-table management across hypervisors, forcing frequent cross-world synchronization and producing severe overheads for mixed memory-access workloads.

We present JANUS, a cross-world, cooperative nested virtualization architecture that cleanly separates CPU and memory virtualization responsibilities. JANUS performs all guest world switches entirely within the guest hypervisor through a lightweight switcher mechanism, while delegating all memory translation to the host hypervisor. This separation removes the host from the critical path of CPU events and eliminates the intermediate shadow or nested page tables that burden existing designs. JANUS introduces several key techniques, including VMFUNC-based EPTP switching for trap-free transitions between guest and nested-guest address spaces; a shadow-root mechanism that protects world-switch integration while allowing direct updates to the nested guest's page tables; and in-guest virtualization exception handling that enables the guest hypervisor to resolve second-level faults with only a single lightweight host interaction. Evaluations demonstrate that JANUS delivers an average performance improvement of 144% over PVM and 28.6% over KVM-based nested virtualization for real-world applications, and imposes less than 5% overhead compared to native containers in production deployment. JANUS demonstrates that rethinking nested virtualization around cross-world cooperation yields strong isolation with near-native container performance.

1 Introduction

Containers, such as RunC [43] and LXC [15], underpin today's cloud-native software stack by offering portable and efficient isolation to cloud workloads [10]. But their reliance on a shared host kernel [13, 17, 44] exposes applications to kernel-level vulnerabilities and cross-tenant attacks. A single exploited container (e.g. CVE-2025-9074) can compromise the host kernel and cascade across tenants [24]. To mitigate this risk, cloud platforms (e.g., Alibaba MaxCompute [1]) increasingly deploy *secure containers*, such as Kata Containers [25], which place each container inside a lightweight virtual machine (microVM). This design restores hardware-enforced isolation for executing untrusted code (e.g., *user-defined functions* (UDFs) [16]), while preserving the agility of container-based workflows and maintaining high performance under bare-metal deployment [2, 54].

However, provisioning bare-metal instances (for deploying secure containers) remains inefficient and inflexible due to long startup times [28], hindering elastic scale-out under bursty workloads. As a result, platforms must maintain a statically overprovisioned pool of instances to absorb peak demand, leading to substantial resource underutilization during typical steady-state workloads. This has driven platforms to increasingly favor cloud VM instances, which offer greater elasticity by enabling on-demand instance allocation to handle dynamic workload fluctuations [21]. Yet, deploying secure containers on cloud VMs faces a fundamental challenge: secure containers must run within an already virtualized environment, leading to **nested virtualization**, where each container's microVM executes inside a cloud VM. The additional layer of CPU and, especially, memory virtualization introduces substantial overheads that make secure containers slower than their traditional counterparts.

Empirical studies reveal that while CPU-bound workloads exhibit negligible slowdown, memory-intensive workloads can degrade by up to two orders of magnitude under nested virtualization [21]. The primary source of this overhead is the need to reconcile *three* levels of page tables – i.e., the *host page table* (*HPT*) in host (L_0), the *guest page table* (GPT_{L_1}) in cloud VM (L_1 guest), and the nested guest page table (GPT_{L_2}) in secure container (L_2 nested guest) – within the *two-level* hardware translation model commonly supported by modern

* Equal contribution.

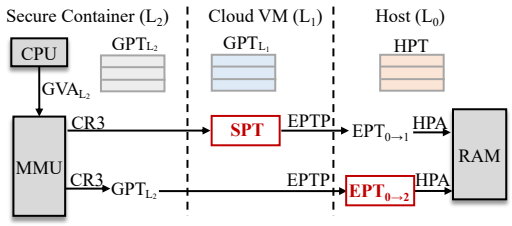


Figure 1: Two approaches for nested memory translation

processors. As illustrated in Figure 1, the hardware MMU traverses two translation structures: the first-level page table (pointed to by CR3) and the second-level extended page table (pointed to by EPTP). With the two-level hardware translation model, nested memory virtualization must compose these three page tables into two to translate a nested guest virtual address (GVA_{L_2}) to the *host physical address* (*HPA*).

More specifically, a classic L_1-dominated approach introduces a *shadow page table* (*SPT*) that merges GPT_{L_2} and GPT_{L_1}, enabling the MMU to perform a two-step translation – from GVA_{L_2} to GPA_{L_1} via *SPT*, and from GPA_{L_1} to *HPA* via $EPT_{0\to1}$[1] [21]. Although straightforward, this approach forces L_1 to constantly synchronize *SPT* with read-only GPT_{L_2}, resulting in high VM exit rates. An alternative L_0-dominated design constructs an $EPT_{0\to2}$ within L_0, directly maintaining the mapping from GPA_{L_2} to *HPA* to complement GPT_{L_2} in supporting two-stage translation [12]. While this design enables L_2 to manage its own page tables, it shifts the synchronization burden to the L_0, which must maintain consistency between the $EPT_{1\to2}$ in the L_1 and $EPT_{0\to2}$.

More recent effort has attempted to circumvent this synchronization bottleneck. CKI [38] constructs GPT_{L_2} to map GVA_{L_2} directly to GPA_{L_1} and leverages Supervisor Protection Keys (PKS) [30] to restrict L_2's memory access, thereby eliminating the need to maintain *SPT*. HyperTurtle [56] takes a different approach by allowing L_1-registered eBPF programs to apply updates directly to $EPT_{1\to2}$ inside L_0, removing the synchronization cost. However, both techniques rely on rigid memory-allocation assumptions, such as preallocation and contiguous allocation, that are difficult to satisfy in dynamic, cloud-native environments.

In this paper, we present JANUS, a cross-world, cooperative nested virtualization architecture that restructures how CPU and memory virtualization are divided across hypervisor layers. As shown in Figure 2, JANUS differs from existing L_0- or L_1-dominated approaches by cleanly separating (nested) virtualization responsibilities: L_1 performs all L_2 world switches (e.g., due to system calls, interrupts, and exceptions) through a lightweight, switcher-based software mechanism, while L_0 exclusively manages nested memory translation via a host-controlled $EPT_{0\to2}$. This division removes L_0 from the critical path of CPU events and hence eliminates the costly cross-world synchronization.

Realizing this architecture requires overcoming several

[1]The notation $0 \to 1$ indicates that the EPT is constructed and maintained by L_0 on behalf of L_1, while $0 \to 2$ and $1 \to 2$ follow analogous semantics.

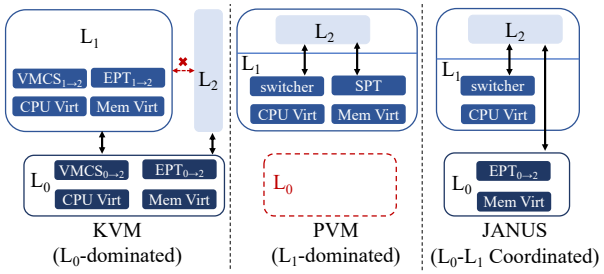


Figure 2: Comparison of nested virtualization approaches among KVM, PVM, and JANUS

challenges in enabling fast and safe cross-world execution. First, JANUS builds upon a software-based CPU virtualization layer, i.e., the *switcher*, that intercepts all L_2 virtualization events within L_1, allowing fast L_1–L_2 world switches without trapping to L_0. Second, seamless transitions between the two worlds require switching not only CPU state but also memory views; JANUS addresses this by using `VMFUNC`-based EPTP switching, allowing the processor to transition between the L_1 view ($EPT_{0\to1}$) and the L_2 view ($EPT_{0\to2}$) entirely within non-root mode (no L_0 participation). Finally, to ensure these transitions remain isolated and tamper-proof, JANUS introduces a shadow-root mechanism that embeds control metadata within the nested guest's page table, preventing a compromised L_2 from subverting world-switch integrity.

Building on this switcher-based CPU virtualization, JANUS realizes nested memory virtualization by using GPT_{L_2} directly as the first-level translation and delegating all second-level mapping to a host-managed $EPT_{0\to2}$, thereby removing the intermediate shadow or nested page tables traditionally maintained in L_1. Making this design both safe and efficient requires addressing two additional challenges. First, writable GPT_{L_2} must not allow a compromised L_2 to remap privileged regions (e.g., the switcher and shadow-root); JANUS enforces this through a *GPA disaggregation* mechanism, which isolates switcher and shadow-root pages from the *GPA* range exposed to L_2. Second, constructing missing $EPT_{0\to2}$ entries must avoid the cascade of exits that plagues prior designs; JANUS achieves this using *in-guest #VE handling*, allowing L_1 to resolve EPT faults locally and issue a single lightweight hypercall to L_0. Together, these mechanisms eliminate costly cross-world synchronization across L_2, L_1, and L_0 and provide fast, secure nested memory management. Beyond basic execution, JANUS extends its design to support key cloud-management operations. For memory reclamation, it provides a lightweight hypercall interface that lets L_1 delegate invalidations to L_0, ensuring coherence across nested translations. For live migration, JANUS 's GPA-disaggregated layout enables hardware *Page Modification Logging* (PML) to track nested-guest dirty pages directly in $EPT_{0\to2}$, eliminating the high overhead of write-protection–based dirty-page tracking.

JANUS delivers system-wide improvements across all layers of the virtualization stack. At the L_2 layer, it enables di-

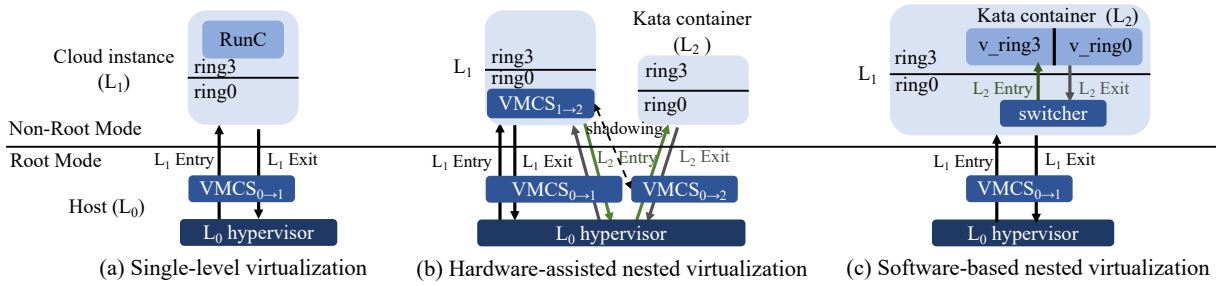


Figure 3: CPU virtualization in single-level virtualization and nested virtualization

rect manipulation of nested guest page tables, allowing fast, trap-free resolution of GPT_{L_2} faults without involving lower layers. The L_1 hypervisor is freed from the burden of maintaining intermediate page tables, effectively eliminating lock contention and frequent VM exits induced by cross-world synchronization. At the L_0 layer, JANUS requires only minimal extensions to deploy $EPT_{0\to2}$, thereby avoiding the complexity and maintenance overhead typically associated with managing $EPT_{1\to2}$. Our evaluation using both microbenchmarks and representative cloud workloads demonstrates that JANUS substantially enhances the efficiency of secure containers on virtualized cloud instances. By eliminating costly cross-world synchronization, JANUS delivers performance gains of up to 3x and 6x over state-of-the-art KVM- and PVM-based nested virtualization approaches.

2 Background

In this section, we provide the necessary background on single and nested virtualization, with a focus on CPU and memory virtualization mechanisms.

As shown in Figure 3, cloud providers deliver virtualized compute instances atop host hypervisors such as KVM [18]. This *single-level* virtualization model is commonly supported by hardware extensions (e.g., Intel VT-x and AMD-V), which enforce a clear division between root and non-root execution modes to isolate host and guest environments. In contrast, containers like RunC [43] rely exclusively on OS-level virtualization techniques, e.g., namespaces [13], control groups (cgroups) [17], and *Linux security modules* (LSM) [44], and thus can operate seamlessly within such virtualized environments with negligible additional complexity.

Recently, the rising demand for stronger isolation guarantees in multi-tenant cloud platforms has driven the adoption of *secure containers*, like Kata Containers [24]. These systems encapsulate conventional containers within lightweight microVMs, retaining container agility while delivering VM-level isolation. At the same time, modern cloud services are commonly provisioned and managed as VM instances, which provide elastic resource allocation, standardized isolation boundaries, and compatibility with existing cloud control planes. To preserve these operational advantages, secure containers are often deployed on top of cloud VMs rather than as a separate bare-metal substrate. In this deployment model, each secure container introduces its own microVM inside an already virtualized cloud instance, resulting in a second layer of virtualization, namely *nested virtualization*, that current hardware does not natively accelerate. This gap exposes severe inefficiencies, particularly in memory management (i.e., the dominant source of overhead in nested virtualization).

2.1 Nested CPU Virtualization

A key challenge in nested virtualization lies in maintaining distinct privilege levels among the host (L_0), guest (L_1), and nested guest (L_2), despite the fact that modern hardware virtualization extensions (e.g., Intel VT-x, AMD-V) expose only a single hardware separation between root and non-root modes. In single-level virtualization, transitions between the L_0 hypervisor and the guest (L_1) are efficiently managed by hardware through the *Virtual Machine Control Structure* (VMCS), which saves and restores processor states across mode switches. However, enabling a second level of virtualization requires these transitions to be emulated entirely in non-root mode, since current hardware lacks native support for an additional privilege boundary [12].

This emulation can be realized in two ways: (1) through hardware-assisted support from L_0 or via (2) software-based mechanisms implemented within L_1. In the hardware-assisted design, as shown in Figure 3(b), L_0 maintains a physical control structure *VMCS* for L_2, synchronizing it with the virtual *VMCS* managed by L_1. Whenever L_1 executes VMCS operations such as VMLAUNCH or VMRESUME (running L_2), the instruction traps to L_0, which must translate virtual *VMCS* into physical *VMCS* and then perform an actual VM entry (on behalf of L_1). Similarly, each VM exit from L_2 first transitions to L_0, which must then reinject the exit event into L_1 to maintain the correct execution semantics. These multi-level transitions introduce high overhead, as every L_2 world switch requires multiple VM exits and reentries involving L_0.

Inspired by paravirtualization techniques such as Xen [11] and Lguest [6], the recent PVM approach [21] introduces a small paravirtualized component called the *switcher*. The switcher implements the minimal subset of VMX state transitions in non-root mode (L_1), allowing efficient world switches between L_1 and L_2 without trapping into L_0.

Specifically, the L_1 hypervisor maps the switcher into both the L_1 and L_2 at identical virtual addresses, allowing control

to transfer seamlessly across world switches. For example, the switcher transitions between worlds by switching the page table and can continue executing the remaining instructions needed to complete the privilege-level transition even after the address space changes. The switcher maintains VMCS-like state structures in per-CPU entry areas and uses them to save and restore processor state during transitions. To enter L_2, the switcher loads the L_2 state (e.g., syscall entry points, IDT, TSS, trampoline stacks, and LDT), switches into the L_2 address space, and issues an `IRET` to drop into the L_2 kernel or user context in non-root ring 3. L_2 kernel and user code remain isolated through separate page tables and a virtual ring hierarchy enforced by L_1. The syscall and IDT handlers installed for L_2 intercept privileged events (syscalls, hypercalls, interrupts, exceptions) and return control to L_1 by restoring its saved state and executing a `RET`. In essence, the switcher implements lightweight VM-entry/VM-exit logic in software, allowing L_2 to reuse the existing *VMCS* of L_1 while providing strong privilege separation via ring 0 and ring 3 state. By localizing VM entry and exit emulation within L_1, such switcher-based CPU virtualization eliminates expensive L_0 traps while preserving correct nested execution semantics.

2.2 Nested Memory Virtualization

Modern processors natively support single-level memory virtualization by performing a two-dimensional page-table walk that translates *guest virtual addresses* (*GVAs*) to *host physical addresses* (*HPAs*): first from *GVAs* to *guest physical addresses* (*GPAs*) using the guest page table, and then from *GPAs* to *HPAs* using a hardware-assisted second-level page table such as Intel's *Extended Page Tables* (EPT) or AMD's *Nested Page Tables* (NPT). Throughout this paper, we refer to EPT as the second-level page table without loss of generality.

In a nested virtualization setting, however, L_0 observes only the mappings from GPA_{L_1} to *HPAs*. It has no visibility into how L_1 manages the nested guest's memory – specifically, how GPA_{L_2} is mapped to GPA_{L_1}. This lack of cross-world visibility between L_0 and L_2 complicates the construction of a direct translation path from GPA_{L_2} to *HPA*. To bridge this gap, two techniques have emerged in practice: *SPT-on-EPT* and *EPT-on-EPT* [12]. The former introduces a software-maintained *SPT* to translate GVA_{L_2} to GPA_{L_1}, while the latter constructs a nested EPT hierarchy that composes mappings from both L_1 and L_0 to translate GPA_{L_2} directly to *HPA*.

PVM-on-EPT. We first describe the advanced SPT-on-EPT model used by PVM-on-EPT [21], which complements PVM's lightweight, switcher-based CPU virtualization (§2.1).

As shown in Figure 4(a), while L_2 maintains its own guest page tables, the effective first-level address translation is carried out by *SPT* maintained by L_1 [2]. During L_2 execution, L_1 hypervisor loads the *SPT* into *CR*3, and protects the memory pages containing the L_2 page table entries as *read-only*. It

[2]For clarity, the construction of $EPT_{0\to1}$ is omitted, as it is orthogonal to the nested translation path under discussion.

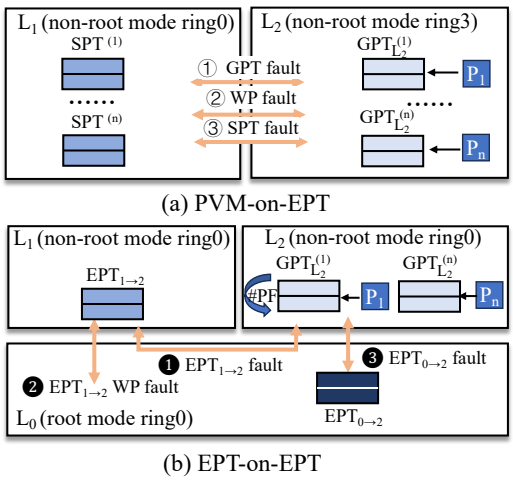


Figure 4: Two approaches for nested memory virtualization

ensures that any modifications to GPT_{L_2} are intercepted and mediated exclusively by L_1.

Therefore, when L_2 accesses a GVA_{L_2} without a valid mapping in *SPT*, a page fault is raised and delivered to L_1. L_1 inspects GPT_{L_2} to identify the cause of the fault. If the corresponding entry is missing in GPT_{L_2}, indicating a *GPT* fault (①), L_1 injects the fault back into L_2, allowing L_2 to update its own page table using the faulting address recorded in *CR*2. This process triggers another page fault because updating the read-only guest page table entries causes a *write-protect* (WP) fault (②). The system must then exit to L_1 to emulate the write to the corresponding L_2 page table page. Once L_2 page table is updated, a subsequent access to the same address triggers another *SPT* fault (③), which L_1 can now resolve by walking both GPT_{L_2} and GPT_{L_1} to construct the translation entry in *SPT*. L_1 then marks the relevant GPT_{L_2} entries as read-only to capture future updates. This process involves *three* world switches between L_1 and L_2 and may repeat until the full translation hierarchy is established.

As *SPT* operates at process granularity and each L_2 process has its own GPT_{L_2}, L_1 must allocate and manage a distinct *SPT* for each L_2 process (e.g., $SPT^{(n)}$ corresponds to the $GPT_{L_2}^{(n)}$ of process *n*). When L_2 runs multi-process workloads, constructing multiple *SPT* significantly increases the frequency of shadow page faults. Moreover, any modification to GPT_{L_2} by L_2 applications triggers a VM exit to L_1, which must update the corresponding entries in *SPT*. Although it can be handled through lightweight world switches between L_1 and L_2 (i.e., via the switcher), their cumulative frequency amplifies lock contention within L_1's memory management subsystem, leading to degraded scalability under multi-process workloads.

EPT-on-EPT. Existing KVM hypervisors employ an EPT-on-EPT approach in conjunction with the hardware-assisted CPU virtualization (§2.1), which introduces significant cross-world synchronization overhead.

As shown in Figure 4(b), GPT_{L_2} directly serves as the first-

level page table, enabling L_2 to handle its own page faults locally without trapping into lower layers. However, when L_2 accesses a GPA_{L_2} whose translation is missing in $EPT_{0\to2}$, an EPT fault is raised. Because L_0 cannot interpret the intermediate mapping between GPA_{L_2} and GPA_{L_1} (maintained by L_1), this EPT fault must first be injected into L_1 (❶). L_1 then constructs the missing translation in $EPT_{1\to2}$, which requires modifying one level of the table entry and triggers a VM exit (due to WP fault) to L_0 for write emulation (❷). After L_0 emulating the page modification, control returns to L_1, which resumes L_2 execution. The resumed L_2 then traps to L_0 once more for the actual VM resume operation. Since $EPT_{0\to2}$ remains incomplete, a final EPT fault occurs (❸). In response, L_0 repairs $EPT_{0\to2}$ by synchronizing it with $EPT_{1\to2}$ and marking the corresponding entries in $EPT_{1\to2}$ as read-only to capture furture updates. Once this fixup completes, L_0 resumes L_2, allowing subsequent accesses to the same GPA_{L_2} to proceed without trapping. Each missing GPA_{L_2} translation therefore requires *three* costly world switches across L_0, L_1, and L_2, and this procedure may repeat multiple times until all levels of GPT_{L_2} are resolved.

3 Motivation and Opportunities

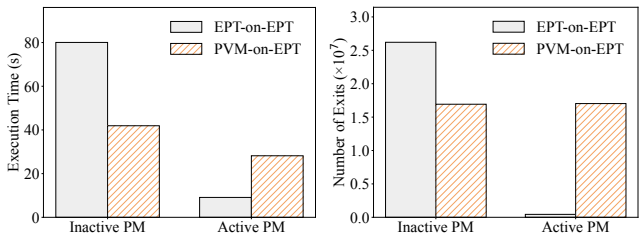
Nested virtualization is a key enabler for cloud platforms to run secure containers inside VM instances while preserving the elasticity and cost efficiency of modern cloud infrastructure. However, the performance costs of existing approaches, especially in nested memory virtualization, remain a critical bottleneck. These overheads continue to hinder the efficient deployment of secure containers, and the impact becomes especially visible for memory-intensive workloads.

3.1 Limitation: Missing L_0-L_1 Coordination

To demonstrate this, we evaluated the performance of accessing 32 GB of physical memory under the EPT-on-EPT and PVM-on-EPT by repeatedly invoking `mmap`, performing memory accesses, and then calling `munmap`. As shown in Figure 5, our experiments include tests on *inactive* physical memory that has never been touched and *active* physical memory whose low-level mappings have already been established.

When accessing *inactive* physical memory, EPT-on-EPT requires nearly twice the execution time of PVM-on-EPT (Figure 5 (a)) because it must construct the lower-level $EPT_{0\to2}$ entries, triggering a larger number of expensive exits to L_0 with most exits caused by EPT violations and a significant portion caused by VMRESUME emulation (Figure 5 (b)).

When accessing *active* physical memory, EPT-on-EPT becomes more efficient than PVM-on-EPT because the necessary $EPT_{0\to2}$ mappings are already present and only a small number of exits to L_0 occur. In contrast, PVM-on-EPT continues to incur a similar number of exits in each iteration because the `mmap` and `munmap` operations modify the L_2 page table, which repeatedly induces GPT_{L_2} and SPT synchronization.



(a) Time (b) Exits

Figure 5: Performance of physical memory (PM) access

When applications run inside secure containers within VMs, their memory access behavior combines both active and inactive physical memory. However, existing nested virtualization approaches perform well in isolated cases – either active or inactive memory accesses – but fail to handle the mixed access patterns that real applications typically exhibit.

These inefficiencies are rooted in the lack of cross-world coordination between L_0 and L_1 in existing nested designs. Hardware-assisted approaches (e.g., EPT-on-EPT) delegate most memory virtualization tasks to the L_0 hypervisor and depend on hardware CPU virtualization to perform costly world switches. In contrast, software-based approaches (e.g., PVM) enable efficient world switches between L_1 and L_2, but at the expense of frequent transitions between these two, imposing significant overhead on L_1 to manage multiple, software-based shadow page tables across multi-process workloads.

3.2 Opportunities: Advanced HW Features

These observations underscore the opportunity for a *cross-world, coordinated* nested memory virtualization architecture that unifies the advantages of efficient software-based world switching with the simplicity of hardware-assisted memory virtualization. Recent hardware advances make such coordination not only feasible but also efficient.

Multi-level Page Tables. Software-based world switching requires that the switching code and its data be mapped at identical virtual addresses (within a PUD range) in both GPT_{L_2} and GPT_{L_1}. Making GPT_{L_2} read-only would degenerate into full shadow paging and undermine JANUS's goal of allowing L_2 to manage its own address space. Instead, JANUS can exploit hierarchical page tables to enforce fine-grained, root-level mediation. It interposes only on top-level (PGD) updates, maintaining a protected shadow PGD as the authoritative root. While L_2 can modify its in-memory PGD, only updates issued via a hypercall to L_1 are validated and applied to the shadow PGD, which is the root page loaded into CR3 during L_2 execution. L_1 allows updates to non-critical regions, but strictly blocks any attempt by L_2 to modify the PGD entry referencing the PUD covering the switching range. This yields a partially shadowed page-table abstraction: the PGD entry for the switching-critical PUD is pinned and immutable, while other entries–and their lower-level page tables

(PUD/PMD/PTE)–remain fully writable. Thus, JANUS preserves L_2's flexibility while ensuring the invariant required for correct and secure world switching.

VMFUNC and EPTP Switching. JANUS enforces L_2 to execute with a dedicated EPT, which is maintained by the L_0 and isolated from L_1's EPT, to enable second-level address translation. Switching between L_1 and L_2 therefore requires changing the active EPT, which would incur significant overhead if handled via host intervention. Modern Intel CPUs provide `VMFUNC` (function 0), enabling EPTP switching entirely in non-root mode and atomic transitions without VM exits. However, because `VMFUNC` is unprivileged, L_2 might switch to any EPTP in the pre-configured EPTP list by the host, potentially including those not intended for its execution. To prevent such malicious switches from taking effect, JANUS binds each L_2 execution context to its designated EPT by restricting the visibility of its shadow root to that EPT only. Consequently, if L_2 switches to a different EPT via `VMFUNC`, the shadow root required for first-level address translation becomes inaccessible, triggering a fault that traps into L_1, which can immediately detect the violation and terminate L_2. This design preserves the performance benefits of `VMFUNC` while enforcing strict EPT isolation without restricting its usage.

Virtualization Exception. The host maintains EPT mappings for L_2, while relying on L_1 to provide the mapping from GPA_{L_2} to GPA_{L_1}. Traditionally, an EPT violation triggers a VM exit to the L_0, which injects an exception into the L_1; the L_1 then forwards the required mapping information back to the host. This back-and-forth interaction significantly increases VM exit frequency and limits the efficiency of $EPT_{0\to2}$ construction. Modern Intel CPUs introduce the *Virtualization Exception* (#VE) mechanism, which delivers EPT violations for designated address ranges directly to a guest-level handler. By resolving violations within the L_1, #VE eliminates L_0 involvement in exception injection and substantially improves cross-layer coordination efficiency.

3.3 Threat Model

Cross-world cooperative nested virtualization preserves the isolation guarantees of conventional single-level virtualization. We assume trusted hardware CPU protection mechanisms, including VMX-based root and non-root modes, as well as ring-based privilege separation between kernel and user execution. We further rely on hardware-enforced memory protection: first-level page tables constrain accesses to guest physical memory, while second-level EPTs restrict accesses to host physical memory within designated regions. Together, these mechanisms enforce strong isolation between the host and its guests, and extend naturally to nested guests.

Our focus is on software-level vulnerabilities in paravirtualized nested guest kernels, guest hypervisors, and the host hypervisor, which might be exposed through virtualization interfaces. We consider the following threat scenarios: (1) a malicious secure container may attempt to compromise nested CPU and memory virtualization by exploiting shared-memory regions or unauthorized address-space switching via the non-privileged `VMFUNC` instruction, potentially targeting the guest or co-located nested guests; and (2) a malicious guest may attempt to violate single-level virtualization boundaries by exploiting the memory virtualization coordination interface or vulnerabilities in the host hypervisor, thereby targeting the host or other guests.

These threats directly guide the design principles of JANUS. First, the system must protect the shared-memory region used for world switching from modification by the nested guest, even when it is mapped into the nested guest page table, in order to preserve the integrity of the switching mechanism and ensure correct guest-level execution. To this end, we design a shadow root mechanism (§4.2) that exposes a controlled view of the root page table (PGD) to the nested guest while maintaining consistency with the actual page-table state. Second, the system must intercept `VMFUNC` invocations that attempt to switch to unauthorized EPTs, thereby preventing illicit cross-domain memory access at the guest level. Accordingly, we develop an enhanced EPT mapping design that leverages GPA disaggregation to detect and block unauthorized EPTP switching via first-level page-table enforcement (§4.3). Finally, when extending memory-related hypercall interfaces to the host hypervisor, the system must enforce strict validation of memory mappings to ensure consistency with kernel memory management (§4.4).

4 JANUS: Design and Implementation

We present JANUS, a high-performance, nested virtualization framework that enables close cooperation between the L_0 and L_1 hypervisors to achieve efficient CPU and memory virtualization. Unlike prior nested virtualization approaches that consolidate CPU virtualization (world switching) and memory virtualization (three-level address translation) within a single hypervisor (PVM-on-EPT) or across both hypervisors (EPT-on-EPT), JANUS separates these mechanisms by assigning the responsibilities of world switching and page-table management to distinct hypervisors. Specifically, JANUS employs a switcher-like, software-based approach in L_1 to handle world switches due to L_2's system calls, page faults, interrupts, and exceptions, while delegating all memory virtualization to the L_0 hypervisor. The separation of CPU and memory virtualization provides two key benefits: (1) world switches become cheaper because all L_2 CPU state transitions can be handled locally within the L_1 hypervisor, closer to the L_2 guest and without involving L_0; and (2) the consolidation of nested three-dimensional page tables onto the two-dimensional paging hardware is delegated entirely to the most privileged hypervisor, L_0, thereby avoiding repetitive and costly world switches across L_2, L_1, and L_0.

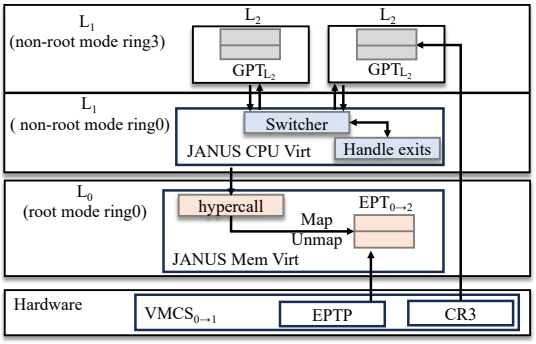


Figure 6: The JANUS architecture

4.1 Architecture Overview

Figure 6 illustrates the overall architecture of JANUS. JANUS leverages the two dimensional page tables in hardware to facilitate the three-level memory address translation in nested virtualization. It directly loads the L_2 guest's page table onto `CR3` for the first level address translation. Accordingly, JANUS loads $EPT_{0\to2}$ onto EPTP to directly translate GPA_{L_2} to HPA, and manages $EPT_{0\to2}$ exclusively in the L_0 hypervisor. While this design is architecturally similar to EPT-on-EPT, JANUS's key contribution is its ability to guarantee that (1) all L_2 visible events, such as page faults, system calls, and interrupts, are either handled natively by L_2 or efficiently processed within L_1, and (2) the management of $EPT_{0\to2}$ at L_0 bypasses any intermediate L_1 page tables and requires only a single world switch to L_0 per $EPT_{0\to2}$ update.

As shown in Figure 6, JANUS's key mechanisms include a software-based switcher that performs world switches for handling L_2 events, and a hypercall interface that enables cooperation between L_1 and L_0 for managing $EPT_{0\to2}$.

4.2 World Switch

Inspired by CPU para-virtualization techniques used in traditional hypervisors such as Xen [11], Lguest [6], and particularly PVM [21], the switcher in JANUS is implemented as a code and data segment within the per-CPU entry area, enabling efficient CPU state transitions in response to L_2 events. In general, switcher is a software-based mechanism that emulates VMCS functionality within non-root mode (i.e., the L_1 hypervisor) to save and restore CPU states across mode switches. Similar to the switcher design in PVM, JANUS places both the L_2 user and kernel spaces in hardware ring 3 of non-root mode, while the switcher itself resides in ring 0 of non-root mode. Thus, all L_2 privileged instructions, system calls, and interrupts trap into the switcher for world switching. To execute correctly across domains during switchover (which involves switching page tables and address spaces), the switcher must reside at the same virtual address in the L_2 user space, L_2 kernel space, and the L_1 hypervisor.

The delegation of memory virtualization, i.e., the management of $EPT_{0\to2}$, entirely to the L_0 hypervisor introduces unique challenges in JANUS's switcher design. Since L_2 directly loads its page table onto `CR3` for the first level address translation and there is no intermediate page table in L_1, the switcher must be indexed by both GPT_{L_2} and GPT_{L_1} to enable L_2-L_1 world switches. Consequently, the world switches involve the switching of GPT_{L_2} and GPT_{L_1} on `CR3`, and the switching of $EPT_{0\to2}$ and $EPT_{0\to1}$ on EPTP. Building on this design, JANUS's switcher offers two key advantages over prior approaches. (1) It resides on the L_1-maintained shadow root page table during L_2 execution, requiring only synchronization with the actual L_2 root page table, analogous to how the switcher resides within GPT_{L_2}. (2) It performs all required EPT switching without trapping to L_0, and efficiently handles switcher-induced EPT faults at runtime.

Table 1: World switches within non-root mode.

Execution Context	CR3	EPTP	Switcher mapping in EPT
L_1 Hypervisor	GPT_{L_1}	$EPT_{0\to1}$	Always present
① **call vm_entry**			
VM Entry	② $\to GPT_{L_2}$	$EPT_{0\to1}$	Always present
	GPT_{L_2}	③ $\to EPT_{0\to2}$	**On demand**
④ **IRET to** L_2			
L_2 Context (h_ring3)	GPT_{L_2}	$EPT_{0\to2}$	not used
⑤ **interrupt/exception/syscall/hypercall**			
VM Exit	GPT_{L_2}	⑥ $\to EPT_{0\to1}$	Always present
	⑦ $\to GPT_{L_1}$	$EPT_{0\to1}$	Always present
⑧ **ret to** L_1 **Hypervisor**			

Algorithm 1 VM Entry–Exit Cycle: $L_1 \to L_2 \to L_1$

Require: Initial state: $CR3 = GPT_{L_1}$, $EPTP = EPT_{0\to1}$.
Ensure: After cycle: L_2 executed and L_1 state restored.

VM Entry ($L_1 \to L_2$):
  Construct L_2 context onto L_2 entry stack
  Save L_1 context on L_1 kernel stack
  Switch to L_1 entry stack
  Switch $CR3$ to GPT_{L_2} (switch to L_2 entry stack)
  Invoke `VMFUNC(0, index)` to switch to $EPT_{0\to2}$
  Execute `IRET` (switch ring, pop stack and restore L_2)
L_2 **Execution**:
  State: $CR3 = GPT_{L_2}$, $EPTP = EPT_{0\to2}$
  Execute L_2 user process or kernel
VM Exit ($L_2 \to L_1$):
  Save L_2 context on L_2 entry stack
  Invoke `VMFUNC(0, 0)` to switch back to $EPT_{0\to1}$
  Restore $CR3$ to GPT_{L_1} (switch to L_1 entry stack)
  Restore L_1 context.

Table 1 summarizes page table switches for various mode transitions and execution contexts, and Algorithm 1 illustrates the execution flow of the complete VM entry–exit cycle. We begin with the L_1 execution context before an L_2 guest is launched. Upon the creation of an L_2 guest and before entering this guest, L_1 hypervisor installs a corresponding switcher in both GPT_{L_1} and $EPT_{0\to1}$. To facilitate L_2 mode switching, the L_1 hypervisor pins the switcher address mapping in L_1. To switch from the L_1 to L_2 context (①), i.e., performing a VM entry, the switcher first loads L_2's page table GPT_{L_2} onto

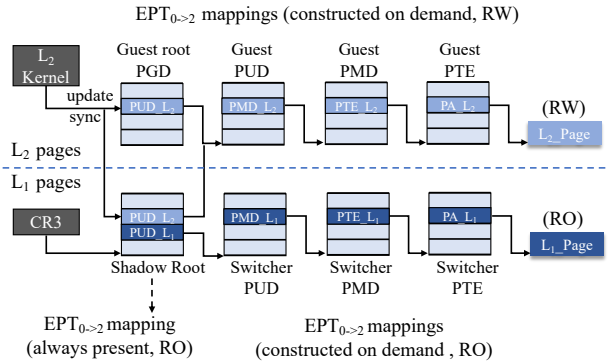


Figure 7: The relationship between shadow root and GPT_{L_2}

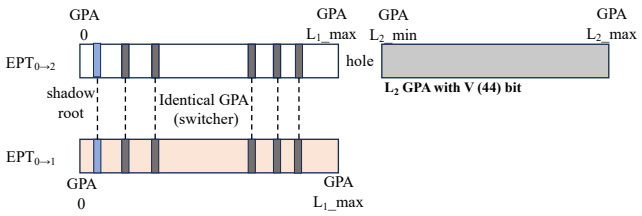


Figure 8: The layout of $EPT_{0\to1}$ and $EPT_{0\to2}$

`CR3` by simply using instruction `mov cr3, phys_addr` (②, Algorithm 1, line 5). However, the switching from $EPT_{0\to1}$ to $EPT_{0\to2}$ (③) is challenging. To ensure all page table switching does not trap into L_0, JANUS uses **VMFUNC-based EPTP switching** to change EPT entirely within the non-root mode, eliminating costly VM exits to L_0. With VMFUNC function 0, the hardware can switch among a set of preconfigured EPT roots supplied by L_0. L_1 uses EPT index 0 for $EPT_{0\to1}$ and maintains a mapping from each L_2 context to its corresponding $EPT_{0\to2}$ entry. During an $L_1 \to L_2$ transition, the switcher executes `VMFUNC` with the target EPT index (Algorithm 1, line 6), enabling a fast, trap-free EPTP switch.

Maintaining per L_2 switchers and directly placing the switcher inside GPT_{L_2} is unsafe because a compromised or malicious L_2 kernel could modify its page tables to remap or overwrite the switcher, potentially subverting the world-switching mechanism. To ensure integrity, JANUS introduces a *shadow-root* mechanism that constructs a shadow page global directory (PGD) mirroring the real GPT_{L_2}. As shown in Figure 7, the corresponding shadow PGD entries are pre-mapped as read-only in $EPT_{0\to2}$ to prevent unauthorized modification. The L_1 hypervisor installs the switcher by configuring the relevant PGD entries to reference the switcher's PUD and its subordinate tables (PUD, PMD, PTE). When an L_2 attempts to modify PGD entries, it also issues a hypercall to L_1 which validates the update and synchronizes it with the shadow root before applying the change. Otherwise, updates to PGD entries in GPT_{L_2} do not take effect, as only modifications propagated to the L_1-authorized shadow root are visible to the hardware MMU. Importantly, such PGD-level updates are infrequent in L_2, while lower-level page table modifications (PUD, PMD, PTE) proceed by L_2 kernel without L_1 involvement, incurring no overhead in the common case. The shadow root mechanism also enforces the security of EPTP switching, as the lower-level EPT mappings associated with each shadow root exist only within the EPT corresponding to its own L_2. Any unauthorized VMFUNC attempt by a malicious L_2 triggers an EPT violation that is trapped by L_1, due to the absence of the required first-level page table mappings, thereby preventing cross-L_2 memory access.

Note that $EPT_{0\to2}$ initially contains only the shadow-root mappings, and the switcher mappings are created on demand. When an access to the switcher triggers an EPT fault, L_0 intercepts, switches to $EPT_{0\to1}$, and completes the exit to L_1, which already has the correct switcher mappings. L_0 then installs the required switcher entries into $EPT_{0\to2}$ with proper permissions. This incurs only a one-time overhead because the switcher region is small and must be initialized only once per $EPT_{0\to2}$.

To finally enter L_2, the switcher performs an `IRET` (Algorithm 1, line 7) using a pre-built cross-page `IRET` frame: the *ss* field is mapped read-only to protect the user-mode *rip*, while the rest of the frame sits on a writable adjacent page. Execution then proceeds in hardware ring 3 under non-root mode, and although $EPT_{0\to2}$ includes the switcher, L_2 cannot access it. System calls and interrupts in L_2 trap to hardware ring 0 (non-root mode), where the switcher performs a VMFUNC-based switch to $EPT_{0\to1}$, loads GPT_{L_1}, and returns to the L_1 hypervisor via `ret` (⑤-⑧, Algorithm 1, line 12-15). This design enables secure, low-latency transitions while preserving strong isolation between L_1 and L_2.

4.3 Memory Virtualization

JANUS achieves nested memory virtualization using the L_2 guest's own page table as the first-level translation, paired with a second-level translation layer managed by the host (L_0) hypervisor. Thus, a GVA_{L_2} is first translated to GPA_{L_2} via GPT_{L_2}, and then to a *host physical address* (HPA) through $EPT_{0\to2}$. Unlike PVM-on-EPT, which marks GPT_{L_2} read-only to intercept and emulate every page-table update, JANUS keeps GPT_{L_2} *writable* so that L_2 can update its page tables without trapping into L_1. Unlike KVM's EPT-on-EPT design, JANUS removes the intermediate $EPT_{1\to2}$ maintained by L_1, avoiding both the synchronization between $EPT_{1\to2}$ and $EPT_{0\to2}$ and the additional exits required for write emulation. These two choices – writable GPT_{L_2} and a single host-managed $EPT_{0\to2}$ – reduce cross-layer switches and shrink the amount of page-table state maintained by L_1.

We now describe how JANUS safely supports writable guest page tables and efficient fault resolution without compromising isolation or incurring cross-layer overhead.

GPA disaggregation and mapping validation. Because GPT_{L_2} is writable, JANUS must ensure that updates made by L_2 cannot map over the switcher. To enforce this separation, JANUS arranges the L_2 GPA layout exposed through $EPT_{0\to2}$ into two regions: (1) the *read-only* shadow-root and switcher

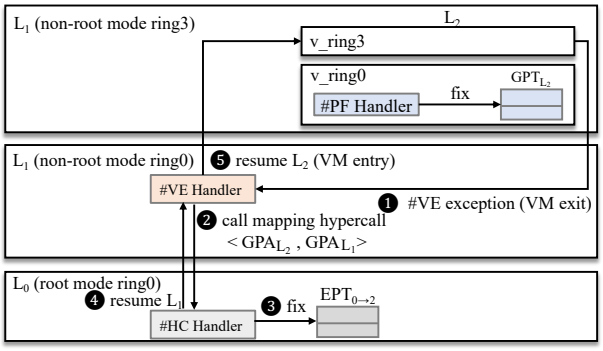


Figure 9: The memory virtualization of JANUS

pages, and (2) the regular *writable* GPA range assigned to L_2.

As shown in Figure 8, $EPT_{0\to2}$ maps the shadow-root and switcher pages at the same GPA positions as in $EPT_{0\to1}$. These entries are marked *read-only* to protect the switcher, except for the writable portion of the `IRET` frame required for world switches. The GPA region assigned to L_2 remains unmapped initially; entries are installed on demand by L_0 as translation faults occur. To ensure that L_1 interprets only legitimate GPT_{L_2} updates, each GPT_{L_2} entry mapping an L_2-assigned page carries a *software-defined V-bit* – set by the paravirtualized L_2 kernel. The V-bit marks the entry as created by L_2, allowing L_1 to distinguish such updates from other arbitrary memory contents (e.g., the shadow-root and switcher pages). When validating a mapping request during a translation fault, L_1 checks the V-bit and rejects any request that lacks it.[3] These checks prevent a compromised L_2 from mapping over privileged memory while allowing ordinary page-table updates to proceed. In this way, JANUS maintains isolation without marking GPT_{L_2} read-only or intercepting its internal updates.

GPT fault handling. With GPT_{L_2} kept writable, JANUS allows L_2 to handle its own first-level translation faults without involving L_1.

Specifically, when a page fault is triggered due to a missing GPT_{L_2} entry, the switcher's #PF handler intercepts the exception and performs *direct page-fault injection* into the L_2 kernel without exiting to the L_1 hypervisor – i.e., the switcher (1) saves the current execution context, including the faulting address stored in `CR2`, into its state-save area; (2) switches to the L_2 kernel context by updating execution stack and adress space (via `SWAPGS` and `MOV CR3`); and (3) returns control to the L_2 guest kernel using `IRET`. Subsequently, the paravirtualized L_2 kernel retrieves the `CR2` value and fault vector to handle the exception natively. Once L_2 resolves the fault and updates its GPT_{L_2}, execution resumes directly in L_2 user mode at ring 3 without any L_1 hypervisor intervention. The only exception arises when the fault modifies the root page directory (PGD). In such cases, a hypercall is issued to synchronize the shadow root maintained by L_1 (§4.2).

[3] L_2 cannot set the V-bit on GPT_{L_2} entries corresponding to the shadow-root or switcher pages, as they are read-only protected from L_2.

EPT fault handling. Faults in $EPT_{0\to2}$ are handled by the L_1 hypervisor via the *Virtualization Exception* (#VE), an Intel VT-x feature that enables selective in-guest handling of virtualization events without exiting to the host L_0 hypervisor.

When #VE is enabled for a vCPU (via the VE enable bit in the VMCS control field), EPT violations that would normally trigger a VM exit can instead raise a #VE (vector 20) within the guest. The #VE Information Area, defined by the VMCS, records details such as the faulting GPA, access type (read-/write/execute), violation cause (e.g., missing translation or permission fault), and the faulting instruction pointer, allowing L_1 to diagnose and handle the fault with L_0 assistance [4].

More specifically, when a GPA_{L_2} lacks a valid mapping in $EPT_{0\to2}$ (Figure 9), a #VE is raised and delivered to L_1 (❶). The L_1 handler (1) inspects the faulting GPA_{L_2}; (2) performs a GPT_{L_1} walk to resolve the corresponding GPA_{L_1}; and (3) issues a `KVM_HC_JANUS_OPS` hypercall with `JANUS_MAP` operation (as listed in Table 2) to L_0 containing the mapping information including GPA_{L_1} and GPA_{L_2} (❷). The L_0 hypervisor then locates the *host physical address* (HPA) associated with the GPA_{L_1} and updates $EPT_{0\to2}$ based on GPA_{L_2} accordingly (❸). By eliminating intermediate shadow page tables in L_1, this process requires only a single lightweight exit to L_1 followed by one exit to L_0, substantially reducing cross-world synchronization overhead compared to conventional nested memory virtualization mechanisms.

Table 2: JANUS hypercall operations and their usage

Operation	Description
JANUS_CREATE	Create the $EPT_{0\to2}$ structure corresponding to the specified JANUS index.
JANUS_DESTROY	Destroy the corresponding $EPT_{0\to2}$ and release all associated structures within L_0.
JANUS_MAP	Construct $EPT_{0\to2}$ mappings based on GPA_{L_2}, GPA_{L_1}, permission, and length.
JANUS_UNMAP	Remove (*zap*) the existing $EPT_{0\to2}$ mappings.
JANUS_CHECK	Check if existing mapping is young and clear it if required.

JANUS introduces only a single hypercall for managing $EPT_{0\to2}$ operations (Table 2), whose semantics are simple and well-defined compared with the complex nested virtualization logic in conventional KVM. Frequent invocation of the new hypercall behaves similarly to standard KVM hypercalls, causing additional vCPU exits without compromising host stability. However, the `JANUS_CREATE` operation requires special handling because it triggers structural initialization in L_0 and must be restricted to control the number of $EPT_{0\to2}$ instances allocated. For data protection, L_0 performs strict validation to ensure that no mapping references *HPAs* beyond the range owned by its corresponding L_1. These checks include boundary and length verification, and the removal of stale mappings after reconfiguration.

[4] Note that, in our design, the Suppress #VE bit (bit 63) is set for *switcher EPT entries* to force a normal EPT violation VM exit for on-demand switcher construction, while it remains cleared for regular L_2 EPT entries to enable in-guest #VE handling.

4.4 JANUS-Assisted Memory Management

Having offloaded nested memory virtualization to the host (L_0) and eliminated the intermediate guest page tables maintained by L_1, JANUS must ensure that this two-level translation model continues to interoperate correctly with existing memory management in both the host hypervisor (L_0) and the guest hypervisor (L_1). This section examines two important memory management scenarios that arise frequently in production clouds – i.e., memory reclamation and secure container live migration – and describes how JANUS enables these operations to proceed efficiently under its design.

4.4.1 Memory Reclamation

Both the L_0 host and the L_1 guest maintain their own memory management subsystems. Under memory pressure, each may reclaim memory previously allocated to guests or nested guests. In current Linux systems, the host kernel at L_0 or L_1 uses the `mmu_notifier` interface (e.g., `invalidate_range`, `change_pte`, and `clear_young`) to inform the L_0 or L_1 hypervisor of modifications to the host page tables that back guest or nested guest physical memory. These notifications ensure that the hypervisor's memory-virtualization structures (e.g., EPT mappings) remain coherent with the kernel's memory view. With the introduction of $EPT_{0\to2}$ for L_2, JANUS extends this mechanism to ensure that host kernel side invalidations propagate correctly across all layers.

At L_0, any `unmap` or reclamation operation may affect the memory view of both L_1 and L_2. Because the host kernel at L_0 performs reclamation using the GPA_{L_1} ranges that back the guest memory, the L_0 hypervisor must be able to locate all second-level mappings derived from a given GPA_{L_1}. To support this, each entry in $EPT_{0\to2}$ is associated with a reverse-mapping (`rmap`) structure keyed by GPA_{L_1}, allowing L_0 to identify and invalidate all relevant $EPT_{0\to2}$ entries when the host kernel reclaims memory. This ensures that memory invalidations initiated by L_0 propagate consistently across the multi-level translation hierarchy.

At L_1, in conventional nested virtualization (SPT-on-EPT), L_1 updates the shadow table to reflect changes to the page tables backing L_2 and all metadata associated with its shadow PTEs (e.g., access permissions and dirty bits). Under JANUS, however, all second-level translations are maintained by L_0 in $EPT_{0\to2}$, so memory invalidations initiated at L_1 must now be delegated directly to L_0. Specifically, when `invalidate_range` is invoked at L_1, the L_1 hypervisor issues a `JANUS_UNMAP` hypercall to L_0, which removes the corresponding entries from $EPT_{0\to2}$ for the affected GPA_{L_1} range. Although L_1 no longer updates a shadow page table for L_2, it still manages its own GPAs and thus must update its internal per-page attributes to keep its local memory state consistent with the mappings removed by L_0. More concretely, pages that become read-only are reflected via `set_pfn_access`, while writable pages record dirtiness using `set_pfn_dirty`.

This way, JANUS enables coherent memory reclamation across layers by allowing L_0 to invalidate nested mappings directly and by having L_1 forward its invalidations while still maintaining its own page metadata.

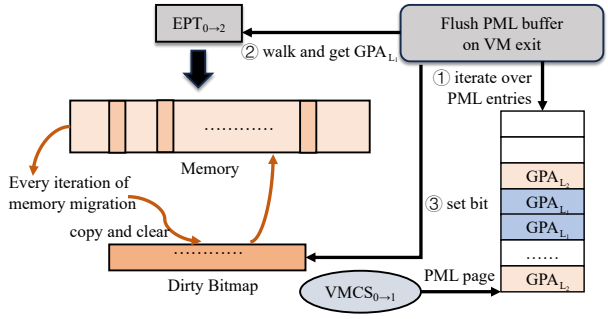


Figure 10: The live migration progress of L_1 with JANUS.

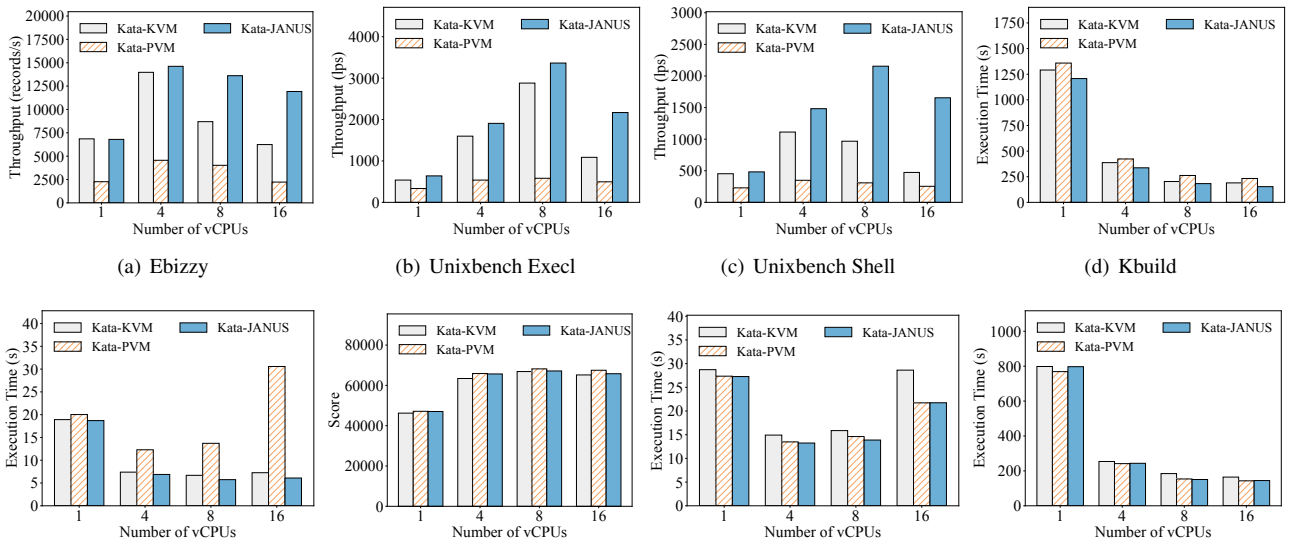
4.4.2 Live Migration

Live migration is a critical capability in cloud environments, enabling VMs to be relocated transparently without disrupting ongoing services. In nested virtualization settings, it is the L_1 VM that is migrated by the cloud provider; all state belonging to an L_2 guest is encapsulated within L_1's memory and is therefore transferred as part of the L_1 migration. Among the stages of live migration, memory migration is particularly important, as it moves the full in-memory state of L_1 (and transitively L_2) to the destination host to ensure seamless resumption of execution. A key requirement for this process is accurate dirty-page tracking, which identifies pages modified during migration so that only changed memory must be resent.

Under conventional hardware-assisted nested virtualization (EPT-on-EPT), VMX's *Page Modification Logging* (PML)[5] cannot be used to track modifications made by L_2. PML logs only the guest physical addresses of trapped writes, but provides no indication of which EPT translation context generated them; writes serviced through $EPT_{0\to1}$ (from L_1) and those serviced through $EPT_{0\to2}$ (from L_2) are indistinguishable in the PML buffer. Consequently, even with PML enabled for L_1 during migration, the host hypervisor must resort to software-based dirty-page tracking for $EPT_{0\to2}$, typically by clearing the write bit of those entries and trapping subsequent updates. This write-protection approach severely degrades L_2 performance during migration because every modified page incurs a fault and exits to L_0.

JANUS overcomes this by leveraging its GPA disaggregation (§4.3), where L_2's GPAs are assigned to a dedicated, non-overlapping region. Since PML logs the guest physical address of each trapped write, this separation allows L_0 to identify PML entries originating from L_2 simply by checking whether the logged GPA falls within the GPA_{L_2} range. As shown in Figure 10, (1) during iterative pre-copy migration,

[5]PML is an Intel VT-x mechanism that records guest-physical pages modified by a VM by logging their GPAs into a hardware-managed buffer, allowing the hypervisor to track dirty pages for operations such as live migration without trapping each write.



(a) Ebizzy (b) Unixbench Execl (c) Unixbench Shell (d) Kbuild

(e) Dedup (f) Specjbb (g) In-memory Analysis (h) Graph Analysis

Figure 11: The performance of eight memory-intensive applications

when a PML entry corresponds to an L_2 GPA (①), the migration procedure walks $EPT_{0\to2}$ (its `rmap`, §4.4.1) to translate the GPA_{L_2} to its backing GPA_{L_1} (②). (2) The resulting GPA_{L_1} is then marked dirty in L_1's dirty bitmap (③), ensuring that all L_2 memory modifications are correctly reflected in the set of pages to be transferred. (3) After each iteration, the dirty bits are cleared on the source host, and (4) once no additional dirty pages remain, the L_1 VM is suspended and resumed on the destination host, completing the migration.

5 Evaluation

Real-world adoption. We implement JANUS with 1662 lines of modifications in the host KVM hypervisor, 3702 lines in the PVM guest hypervisor, and 264 lines in the PVM nested guest kernel. JANUS has been deployed in production on a leading cloud provider's PaaS platform to support the Flink distributed stream and batch processing system, enabling secure container execution without compromising elasticity. PVM [21] – a state-of-the-art software-based nested virtualization system – increases total query time across 17 Flink queries by roughly 30% for C++ and 20% for Java engines compared to traditional RunC containers [43]. In contrast, JANUS-based secure containers add less than 5% overhead relative to RunC. These results demonstrate that JANUS achieves hardware-grade isolation with near-native performance, making it well suited for latency-sensitive, multi-tenant cloud environments.

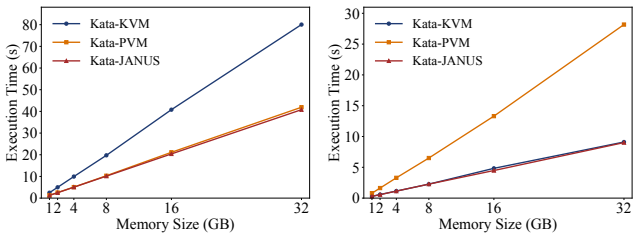
Testbed and configurations. To further assess JANUS under controlled conditions, we conducted experiments within L_1 VMs running atop the L_0 host. The host machine was equipped with an Intel(R) Xeon(R) Platinum 8475B processor (3200 MHz) featuring 192 physical CPUs and 384 GB RAM, running the Linux 5.10.134 kernel, the dominant version in our production environment, consistently across all layers (L_0, L_1, and L_2). Both L_0 and L_1 KVM hypervisors are configured with `tpd_mmu=y`, a Google-proposed enhancement that improves parallelism in memory virtualization by mitigating contention on the global MMU lock. The L_0 hypervisor enabled VMCS Shadowing, allowing the L_1 KVM hypervisor to directly manage its own VMCS structures for L_2 guests without causing additional exits. To align with common cloud configurations, Transparent Huge Pages (THP) were enabled in L_1 (`transparent_hugepage=always`, and `shmem_enabled=advise`) to accelerate memory efficiency.

Baselines for comparison. We evaluated three nested container configurations for comparison: (1) Kata-KVM: Kata Containers using hardware-assisted nested virtualization via the L_1 KVM hypervisor (standard nested KVM); (2) Kata-PVM: Kata Containers using software-based nested virtualization via the L_1 PVM hypervisor (the PVM [21] approach); and (3) Kata-JANUS: our JANUS-based nested virtualization.

5.1 Memory-Intensive Applications

In this section, we first evaluated eight representative memory-intensive applications under multiple resource configurations. The benchmarks consisted of four multi-process applications, including ebizzy [4], UnixBench execl and shell [8], and Kbuild [5]; and four multi-threaded applications, including SPECjbb [39] and Dedup from the PARSEC3.0 [52] and In-Memory Analytics and Graph Analytics from the CloudSuite [3]. We then used the stress-ng [7] memory test to capture memory accesses to both inactive guest physical memory that was never touched and to active guest physical memory whose mappings were already established. Together, these applications exhibit a wide range of memory behaviors, including random memory initialization, frequent modification, and high-intensity data access.

The performance results for memory-intensive applications are shown in Figure 11. Across both multi-process and multi-



(a) Inactive physical memory (b) Active physical memory

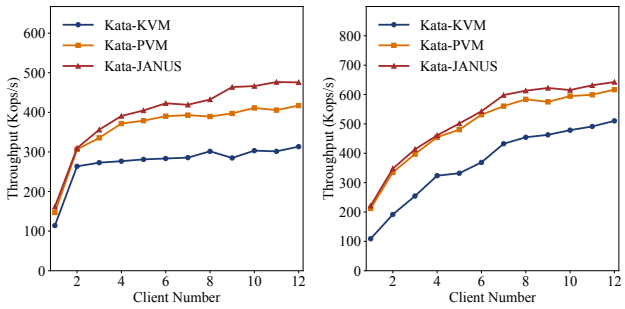
Figure 12: Performance of physical memory (PM) access.

threaded workloads, Kata-JANUS achieves the best overall performance and superior scalability by eliminating intermediate page-table management within the guest hypervisor. Specifically, for multi-process applications, Kata-PVM consistently delivers the lowest performance because it must maintain a distinct shadow page table for each process, which incurs substantial overhead from intensive SPT management and synchronization. In contrast, Kata-JANUS improves multi-process performance by 339.7% on average over Kata-PVM and further outperforms Kata-KVM by 51.8% when using 8 vCPUs. For multi-threaded applications, Kata-PVM outperforms Kata-KVM because it avoids the expensive EPT reconstructions that dominate Kata-KVM's execution time. Dedup is the only exception, as its frequent GPT_{L_2} updates trigger numerous exits for synchronizing with $SPT_{1\to2}$. Kata-JANUS still achieves an average performance improvement of 37% over Kata-PVM and 13.3% over Kata-KVM when using 8 vCPUs, due to its faster mapping-construction path.

The results in Figure 12 show that Kata-JANUS consistently achieves the best performance for both inactive and active memory access patterns. Specifically, when the system accesses inactive guest physical memory, Kata-KVM experiences a substantial performance drop because the construction of the low-level EPT mapping requires multiple expensive exits to L_0. When the system accesses active guest physical memory, Kata-PVM becomes the slowest design because it repeatedly synchronizes the L_2 guest page table with the corresponding shadow page table in L_1. In contrast, Kata-JANUS performs slightly better than Kata-PVM during the initial construction of memory mappings, and this behavior confirms the benefit of its direct page-fault injection for repairing the L_2 guest page table together with its lightweight procedure for constructing the low-level EPT mapping.

5.2 In-memory Key-Value Stores

We also evaluated the performance of two widely deployed in-memory key-value stores, Redis and Memcached, to assess system efficiency under realistic and latency-sensitive workloads. These systems are representative of memory-centric cloud services that issue frequent fine-grained read and write operations and are therefore highly sensitive to the overhead introduced by nested memory virtualization. In our tests, the Redis and Memcached servers executed inside the L_2, while



(a) Redis (b) Memcached

Figure 13: Throughput of in-memory key-value stores

clients ran in L_1 and continuously generated requests.

The performance results are shown in Figure 13. Across all workloads, Kata-JANUS provides the highest throughput and best scalability. For example, Kata-JANUS increases Redis throughput by an average of 9.7% and 45.1% relative to Kata-PVM and Kata-KVM, respectively, and boosts Memcached throughput by an average of 4.4% and 48.2%. Again, JANUS's performance advantage arises from its lightweight, host-managed memory virtualization design, which minimizes cross-world exits and eliminates the costly synchronization overhead.

We also evaluated these applications in both host and guest, while scaling the number of co-located malicious guests from 0 to 8 to stress the JANUS-introduced hypercalls. The results show that the performance impact is negligible.

5.3 Micro-benchmarks

We finally used various microbenchmarks to quantify the fundamental costs introduced by nested virtualization, focusing on world-switch latency, two-level address mapping, and memory modification tracking.

Table 3: Comparison of world-switch costs

Configuration	Switching Path	Cycles
Kata-KVM	$L_2 \to L_0 \to L_1 \to L_0 \to L_2$	16,002
Kata-PVM	$L_2 \to L_1 \to L_2$	2,681
Kata-JANUS	$L_2 \to L_1 \to L_2$	2,700

Cost of world switch. A world switch occurs when the processor exits to the hypervisor to handle events of the guest and then re-enters the guest after completion. These events include the execution of sensitive instructions such as `CPUID` or `HLT`, accesses to privileged registers, external interrupts, page faults, I/O operations, and guest-issued hypercalls that require hypervisor intervention to preserve virtualization integrity.

During Kata container execution, the world switch occurs between L_2 and L_1, as L_2 runs on top of L_1. With hardware-assisted nested virtualization (i.e., Kata-KVM), the switching process involves two exits to L_0, while switcher-based nested virtualization (i.e., Kata-PVM) completes the switch with a single exit to L_1. JANUS inherits this switcher-based world switch design and introduces an additional `VMFUNC`-based mechanism to switch EPT mappings between L_1 and L_2.

We conducted a microbenchmark that invoked one million hypercalls, each triggering a pure world switch. We measured the average cost of a complete switch cycle. The results are summarized in Table 3. The world switch of Kata-KVM takes approximately 16,002 cycles, as the L_0 KVM must perform costly synchronization between $VMCS_{0\to2}$ and $VMCS_{1\to2}$ during nested virtualization transitions. PVM achieves 2,681 cycles per switch owing to its lightweight switcher-based CPU virtualization. JANUS achieves a similar latency of 2,700 cycles, demonstrating that the additional `VMFUNC` operation introduces negligible overhead.

Cost of mapping construction. Constructing a two-level hardware memory mapping involves handling both GPT faults and SPT/EPT faults. For Kata-KVM and Kata-JANUS, GPT_{L_2} faults can be resolved locally without VM exits, whereas Kata-PVM must exit to the L_1 hypervisor to reinject the #PF fault into L_2. After the GPT_{L_2} is repaired, Kata-PVM performs the $SPT_{1\to2}$ fixup within L_1, while Kata-KVM and Kata-JANUS require L_0 assistance to update $EPT_{0\to2}$. As discussed earlier, Kata-JANUS incurs only one additional exit to L_0, whereas Kata-KVM requires multiple exits.

Table 4: Mapping construction overheads (cycles)

Configuration	GPT fault (Fix GPT)	EPT/SPT fault	Total
Kata-KVM	1,960 (576)	55,980	57,840
Kata-PVM	7,048 (2,736)	12,218	19,266
Kata-JANUS	3,186 (1,546)	24,914	28,100
Kata-JANUS (#VE)	3,026 (1,316)	23,934	26,960

We implemented a microbenchmark that repeatedly triggered the mapping construction path. As shown in Table 4, Kata-PVM incurs higher overhead for handling GPT faults, while Kata-KVM suffers from substantial costs associated with EPT fault handling. In contrast, Kata-JANUS benefits from direct #PF injection, enabling more efficient GPT fault processing, and outperforms Kata-KVM in EPT fault handling by issuing fewer exits to L_0 and using lightweight hypercall-based EPT updates. Notably, when constructing an initial mapping, Kata-JANUS is slightly slower than Kata-PVM due to its use of $EPT_{0\to2}$, which requires one additional expensive exit to L_0. However, once the low-level $EPT_{0\to2}$ mapping is established, the overhead of both Kata-KVM and Kata-JANUS becomes dominated solely by GPT faults, whereas Kata-PVM must continue synchronizing SPT updates with the GPT mapping, resulting in additional overhead.

Performance impact of dirty-page tracking. Maintaining stable performance during L_1 migration is essential for cloud providers. We evaluated memory modification overheads before and after enabling dirty-page tracking.

As summarized in Table 5, since the low-level $EPT_{0\to2}$ mappings are already established, memory writes do not trigger additional mapping construction, and both Kata-KVM and Kata-JANUS initially exhibit performance comparable to native L_1 execution. However, once dirty-page tracking is enabled, Kata-KVM experiences substantial degradation because all $EPT_{0\to2}$ entries must be write-protected to track dirty pages, causing every memory write to trigger an exit to L_0. As a result, Kata-KVM incurs a 175.5% increase in memory modification overhead on average. In contrast, JANUS leverages hardware PML directly on $EPT_{0\to2}$ without requiring write protection, thereby avoiding excessive VM exits and maintaining near-identical performance.

Table 5: Memory performance before and after enabling dirty-page tracking

	Execution Time (s)			
	1GB	2GB	4GB	8GB
L_1 VM	0.27/0.35	0.55/0.70	1.16/1.48	2.29/2.93
Kata-KVM	0.294/0.80	0.59/1.70	1.20/3.23	2.38/6.49
Kata-JANUS	0.28/0.37	0.57/0.75	1.16/1.67	2.36/2.97

6 Discussion and Future Work

JANUS currently relies on CR3-based page table switching to isolate the kernel and user spaces of L_2, requiring transitions through the switcher to update CR3 during context switches. In future work, we plan to leverage EPT-based isolation between the L_2 kernel and user spaces, enabling direct and lightweight context switches entirely within L_2 without involving the switcher. In addition, we are exploring the use of *Hypervisor-managed Linear Address Translation* (HLAT) [48] to protect the switcher while eliminating the need to maintain a shadow root for the GPT_{L_2}.

7 Related Work

Securing containers. The security of containerized workloads has drawn extensive attention due to the shared-kernel architecture of conventional containers, which allows a single compromised container to potentially escalate privileges and compromise the host system [19, 20, 24, 31]. Existing work on shared-kernel security enhancement emphasizes strengthening memory isolation [45] and privilege protection [22, 46], reducing the exposed attack surface by intercepting or restricting system calls [34, 47], and enforcing runtime security monitoring [41] to detect or mitigate malicious behavior within containers. Hardware-assisted designs such as RContainer [55], BlackBox [41], and LightZone [51] leverage trusted execution environments to ensure strong isolation of container workloads. Microkernel-based systems including MettEagle [33], LITESHIELD [32], and split containers [37] limit the impact of compromised containers on critical system components, while recent WebAssembly-based containers [23, 29, 50] enhance security by exploiting the inherent isolation provided by the Wasm runtime. In contrast to kernel-sharing approaches, some systems duplicate kernel functionality for each container, with gVisor [49] implementing a user-space kernel that intercepts and validates system calls to decouple containerized applications from the host kernel, and X-Container [36] and μKontainer [40] relying on separate Library OS instances to provide stronger isolation. Secure container frameworks such as Kata Containers [25] and Firecracker [9] further en-

hance isolation by running each container inside a lightweight virtual machine, combining container agility with hardware-enforced security guarantees, and secure containers have now emerged as a practical foundation for cloud-native security with production deployment in modern hyperscale platforms such as Alibaba Cloud [25].

Enabling nested virtualization. Virtualization has long been a cornerstone of cloud computing, evolving from early full virtualization to paravirtualization [11], then hardware-assisted virtualization [18], and more recently to delegated user-level virtualization [14]. Nested virtualization remains a challenging problem, as it must support running guest hypervisors within virtualized guests, which is essential for deploying secure containers in cloud-native infrastructures. Early systems such as Turtles [12] and NEVE [26] demonstrated the feasibility of nested execution by emulating hardware virtualization features at the host level. Later processor extensions, including VMCS shadowing, reduced the number of VM exits by allowing guest hypervisors to directly access certain control structures. More recent work adopts architectural co-design strategies to minimize cross-world semantic gaps. For example, SVT [42] leverages *simultaneous multithreading* (SMT) to maintain guest hypervisor and nested guest on separate logical threads, while DVH [27] enables direct virtual hardware provisioning to streamline interrupt delivery and device emulation. Despite these advances, memory virtualization remains the primary bottleneck, as cross-world synchronization continues to dominate overhead in nested environments.

Optimizing nested memory virtualization. Nested memory virtualization requires composing three translation page tables into two hardware-supported stages, which often incurs substantial overhead due to multi-level synchronization. Prior efforts have sought to mitigate costly world switches through optimized page-fault handling. PVM [21] enables lightweight world switches and handles L_2 page faults entirely within L_1, avoiding L_0 involvement, but frequent updates to GPT_{L_2} increase the overall number of page faults. CKI [38] embeds GVA_{L_2}-GPA_{L_1} mappings directly within GPT_{L_2} using contiguous segment allocation, trading flexibility for improved page-fault handling efficiency. Other designs [35, 53] reduce translation overhead through direct address exposure or huge-page coalescing, while Hyperturtle [56] leverages eBPF-based hyper-upcalls to synchronize EPT hierarchies across layers, lowering update latency yet retaining cross-world dependencies. In contrast, JANUS redefines nested memory virtualization by decoupling CPU and memory control across layers and exposing nested guest page tables directly to the host-managed EPT hierarchy, achieving near-native performance with minimal modifications to the host.

8 Conclusion

This paper presents JANUS, a high-performance nested virtualization architecture that reimagines memory virtualization for secure containers. JANUS decouples CPU and memory virtualization across layers, eliminates intermediate shadow mappings, and exposes the nested guest page table directly to the host-managed EPT hierarchy. Leveraging optimized fault-handling mechanisms, it provides efficient, trap-free memory virtualization with minimal host intervention. As a self-contained and host-transparent design, JANUS simplifies the nested virtualization stack while achieving better performance for secure containers, bridging the longstanding gap between strong workload isolation and cloud-native efficiency. JANUS has been adopted by multiple Alibaba Cloud PaaS platforms to host secure containers, providing a more performant and cost-effective solution for cloud deployments.

9 Acknowledgment

We thank the anonymous reviewers and our shepherd for their constructive feedback and suggestions. This work is supported by the Fundamental and Interdisciplinary Disciplines Breakthrough Plan of the Ministry of Education of China under grant JYB2025XDXM103, Hubei Provincial Natural Science Foundation of China under grant 2026AFA002, Science and Technology Plan Project of Hubei Province under grant 2025CSA056, and the Hubei Provincial Natural Science Foundation of China under grant 2026AFB102. The corresponding author is Hang Huang.

References

[1] *Alibaba MaxCompute*. https://www.alibabacloud.com/help/en/maxcompute/security-and-compliance/maxcompute-security-white-paper/, 2026.

[2] *Bare-metal instance*. https://www.alibabacloud.com/en/product/ebm, 2026.

[3] *CloudSuite*. https://github.com/parsa-epfl/cloudsuite, 2026.

[4] *Ebizzy*. https://sourceforge.net/projects/ebizzy/, 2026.

[5] *Kcbench*. https://gitlab.com/knurd42/kcbench, 2026.

[6] *Lguest*. http://lguest.ozlabs.org/, 2026.

[7] *Stress-ng*. https://github.com/ColinIanKing/stress-ng, 2026.

[8] *UnixBench*. https://github.com/kdlucas/byte-unixbench, 2026.

[9] Alexandru Agache, Marc Brooker, Alexandra Iordache, Anthony Liguori, Rolf Neugebauer, Phil Piwonka, and

Diana-Maria Popa. Firecracker: Lightweight virtualization for serverless applications. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 419–434, 2020.

[10] Armin Balalaie, Abbas Heydarnoori, and Pooyan Jamshidi. Microservices architecture enables devops: Migration to a cloud-native architecture. *IEEE Software*, 33(3):42–52, 2016.

[11] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. *ACM SIGOPS Operating Systems Review*, 37(5):164–177, 2003.

[12] Muli Ben-Yehuda, Michael D. Day, Zvi Dubitzky, Michael Factor, Nadav Har'El, Abel Gordon, Anthony Liguori, Orit Wasserman, and Ben-Ami Yassour. The turtles project: Design and implementation of nested virtualization. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 423–436, 2010.

[13] Eric W. Biederman and Linux Networx. Multiple instances of the global linux namespaces. In *Proceedings of the Linux Symposium*, volume 1, pages 101–112, 2006.

[14] Jiahao Chen, Dingji Li, Zeyu Mi, Yuxuan Liu, Binyu Zang, Haibing Guan, and Haibo Chen. Security and performance in the delegated user-level virtualization. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 209–226, 2023.

[15] Wes Felter, Alexandre Ferreira, Ram Rajamony, and Juan Rubio. An updated performance comparison of virtual machines and linux containers. In *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 171–172, 2015.

[16] Yannis Foufoulas and Alkis Simitsis. User-defined functions in modern data engines. In *Proceedings of the IEEE International Conference on Data Engineering (ICDE)*, pages 3593–3598, 2023.

[17] Xing Gao, Zhongshu Gu, Zhengfa Li, Hani Jamjoom, and Cong Wang. Houdini's escape: Breaking the resource rein of linux control groups. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 1073–1086, 2019.

[18] Irfan Habib. Virtualization with kvm. *Linux Journal*, 2008(166):8, 2008.

[19] Md Sadun Haq, Thien Duc Nguyen, Ali Şaman Tosun, Franziska Vollmer, Turgay Korkmaz, and Ahmad-Reza Sadeghi. Sok: A comprehensive analysis and evaluation of docker container attack and defense mechanisms. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, pages 4573–4590. IEEE, 2024.

[20] Yi He, Roland Guo, Yunlong Xing, Xijia Che, Kun Sun, Zhuotao Liu, Ke Xu, and Qi Li. Cross container attacks: The bewildered ebpf on clouds. In *Proceedings of the USENIX Security Symposium (Security)*, pages 5971–5988, 2023.

[21] Hang Huang, Jiangshan Lai, Jia Rao, Hui Lu, Wenlong Hou, Hang Su, Quan Xu, Jiang Zhong, Jiahao Zeng, Xu Wang, Zhengyu He, Weidong Han, Jiang Liu, Tao Ma, and Song Wu. Pvm: Efficient shadow paging for deploying secure containers in cloud-native environment. In *Proceedings of the Symposium on Operating Systems Principles (SOSP)*, pages 515–530, 2023.

[22] Hang Huang, Honglei Wang, Jia Rao, Song Wu, Hao Fan, Chen Yu, Hai Jin, Kun Suo, and Lisong Pan. vkernel: Enhancing container isolation via private code and data. *IEEE Transactions on Computers*, 73(7):1711–1723, 2024.

[23] Zhuo Huang, Hao Fan, Junhui Peng, Qi Wu, Song Wu, Chen Yu, Hai Jin, Qiming Liu, Wei Yang, and Shuo Yu. Waf: An efficient webassembly-based execution environment for user-defined functions. In *Proceedings of the IEEE International Conference on Data Engineering (ICDE)*, pages 1966–1980, 2025.

[24] Omar Jarkas, Ryan Ko, Naipeng Dong, and Redowan Mahmud. A container security survey: Exploits, attacks, and defenses. *ACM Computing Surveys*, 57(7):1–36, 2025.

[25] Zijun Li, Jiagan Cheng, Quan Chen, Eryu Guan, Zizheng Bian, Yi Tao, Bin Zha, Qiang Wang, Weidong Han, and Minyi Guo. Rund: A lightweight secure container runtime for high-density deployment and high-concurrency startup in serverless computing. In *Proceedings of the USENIX Annual Technical Conference (ATC)*, pages 53–68, 2022.

[26] Jin Tack Lim, Christoffer Dall, Shih-Wei Li, Jason Nieh, and Marc Zyngier. Neve: Nested virtualization extensions for arm. In *Proceedings of the Symposium on Operating Systems Principles (SOSP)*, pages 201–217, 2017.

[27] Jin Tack Lim and Jason Nieh. Optimizing nested virtualization performance using direct virtual hardware. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 557–574, 2020.

[28] Likai Liu, Fu Xiao, Lei Han, Weibei Fan, Xin He, and Zheng Wu. Nbbm: An efficient smartnic-based architecture for bare-metal management in cloud platforms. *IEEE Transactions on Computers*, 2025.

[29] Mugeng Liu, Haiyang Shen, Yixuan Zhang, Hong Mei, and Yun Ma. Webassembly for container runtime: Are we there yet? *ACM Transactions on Software Engineering and Methodology*, 34(6):1–22, 2025.

[30] Lukas Maar, Martin Schwarzl, Fabian Rauscher, Daniel Gruss, and Stefan Mangard. Dope: Domain protection enforcement with pks. In *Proceedings of the Annual Computer Security Applications Conference (ACSAC)*, pages 662–676, 2023.

[31] Ruchika Malhotra, Anjali Bansal, and Marouane Kessentini. A systematic literature review on maintenance of software containers. *ACM Computing Surveys*, 56(8):1–38, 2024.

[32] Kaesi Manakkal, Nathan Daughety, Marcus Pendleton, and Hui Lu. Liteshield: Secure containers via lightweight, composable userspace μkernel services. In *Proceedings of the USENIX Annual Technical Conference (ATC)*, pages 973–985, 2025.

[33] Till Miemietz, Viktor Reusch, Matthias Hille, Lars Wrenger, Jana Eisoldt, Jan Klötzke, Max Kurze, Adam Lackorzynski, Michael Roitzsch, and Hermann Härtig. Metteagle: Costs and benefits of implementing containers on microkernels. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 979–996, 2025.

[34] Maryam Rostamipoor, Seyedhamed Ghavamnia, and Michalis Polychronakis. Confine: Fine-grained system call filtering for container attack surface reduction. *Computers & Security*, 132:103325, 2023.

[35] Xiaowei Shang, Weiwei Jia, Jianchen Shan, Xiaoning Ding, and Cristian Borcea. Reestablishing page placement mechanisms for nested virtualization. *IEEE Transactions on Cloud Computing*, 11(3):3239–3250, 2023.

[36] Zhiming Shen, Zhen Sun, Gur-Eyal Sela, Eugene Bagdasaryan, Christina Delimitrou, Robbert Van Renesse, and Hakim Weatherspoon. X-containers: Breaking down barriers to improve performance and isolation of cloud-native containers. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 121–135, 2019.

[37] Jiacheng Shi, Jinyu Gu, Yubin Xia, and Haibo Chen. Serverless functions made confidential and efficient with split containers. In *Proceedings of the USENIX Security Symposium (Security)*, pages 1091–1110, 2025.

[38] Jiacheng Shi, Yang Yu, Jinyu Gu, and Yubin Xia. A hardware-software co-design for efficient secure containers. In *Proceedings of the European Conference on Computer Systems (EuroSys)*, pages 1229–1245, 2025.

[39] SPEC SPECjbb2005. Release 1.07. *Standard Performance Evaluation Corporation*, 2006.

[40] Hajime Tazaki, Akira Moroo, Yohei Kuga, and Ryo Nakamura. How to design a library os for practical containers? In *Proceedings of the ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE)*, pages 15–28, 2021.

[41] Alexander Van't Hof and Jason Nieh. Blackbox: a container security monitor for protecting containers on untrusted operating systems. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 683–700, 2022.

[42] Lluís Vilanova, Nadav Amit, and Yoav Etsion. Using smt to accelerate nested virtualization. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, pages 750–761, 2019.

[43] Xingyu Wang, Junzhao Du, and Hui Liu. Performance and isolation analysis of runc, gvisor and kata containers runtimes. *Cluster Computing*, 25(2):1497–1513, 2022.

[44] Chris Wright, Crispin Cowan, Stephen Smalley, James Morris, and Greg Kroah-Hartman. Linux security modules: General security support for the linux kernel. In *Proceedings of the USENIX Security Symposium (Security)*, 2002.

[45] Shaowen Xu, Qihang Zhou, Zhicong Zhang, Xiaoqi Jia, Donglin Liu, Heqing Huang, Haichao Du, and Zhenyu Song. Conmonitor: Lightweight container protection with virtualization and vm functions. In *Proceedings of the ACM Symposium on Cloud Computing (SoCC)*, pages 755–773, 2024.

[46] Shouyin Xu, Yuewu Wang, Lingguang Lei, Kun Sun, Jiwu Jing, Siyuan Ma, Jie Wang, and Heqing Huang. Condo: enhancing container isolation through kernel permission data protection. *IEEE Transactions on Information Forensics and Security*, 19:6168–6183, 2024.

[47] Seungyong Yang, Brent Byunghoon Kang, and Jaehyun Nam. Optimus: association-based dynamic system call filtering for container attack surface reduction. *Journal of Cloud Computing*, 13(1):71, 2024.

[48] Kenichi Yasukata, Hajime Tazaki, and Pierre-Louis Aublin. Exit-less, isolated, and shared access for virtual machines. In *Proceedings of the ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 224–237, 2023.

[49] Ethan G. Young, Pengfei Zhu, Tyler Caraza-Harter, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. The true cost of containing: A gvisor case study. In *Proceedings of the USENIX Workshop on Hot Topics in Cloud Computing (HotCloud)*, 2019.

[50] Zhaofeng Yu, Dongyang Zhan, Lin Ye, Haining Yu, Hongli Zhang, and Zhihong Tian. Exploring and exploiting the resource isolation attack surface of webassembly containers. In *Proceedings of the USENIX Security Symposium (Security)*, pages 1111–1128, 2025.

[51] Ziqi Yuan, Siyu Hong, Ruorong Guo, Rui Chang, Mingyu Gao, Wenbo Shen, and Yajin Zhou. Lightzone: Lightweight hardware-assisted in-process isolation for arm64. In *Proceedings of the International Middleware Conference (Middleware)*, pages 467–480, 2024.

[52] Xusheng Zhan, Yungang Bao, Christian Bienia, and Kai Li. Parsec 3. 0: A multicore benchmark suite with network stacks and splash-2x. *ACM SIGARCH Computer Architecture News*, 44(5):1–16, 2017.

[53] Jiyuan Zhang, Weiwei Jia, Siyuan Chai, Peizhe Liu, Jongyul Kim, and Tianyin Xu. Direct memory translation for virtualized clouds. In *Proceedings of the ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 287–304, 2024.

[54] Xiantao Zhang, Xiao Zheng, Zhi Wang, Hang Yang, Yibin Shen, and Xin Long. High-density multi-tenant bare-metal cloud. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 483–495, 2020.

[55] Qihang Zhou, Wenzhuo Cao, Xiaoqi Jia, Peng Liu, Shengzhi Zhang, Jiayun Chen, Shaowen Xu, and Zhenyu Song. Rcontainer: A secure container architecture through extending arm cca hardware primitives. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2025.

[56] Ori Ben Zur, Jakob Krebs, Shai Aviram Bergman, and Mark Silberstein. Accelerating nested virtualization with hyperturtle. In *Proceedings of the USENIX Annual Technical Conference (ATC)*, pages 987–1002, 2025.

ARTIFACT EVALUATED
USENIX
AVAILABLE

Osprey: Transparent and Efficient Virtual Memory for Secure Computation

Yicheng Liu *UCLA*
Alice Yeh *UC Berkeley*
Harry Xu *UCLA*
Raluca Ada Popa *UC Berkeley*
Sam Kumar *UCLA*

Abstract

There is increasing interest in privacy-preserving data analytics applications. These applications rely on Secure Computation (SC), a family of cryptographic techniques for computing on encrypted data. Unfortunately, SC amplifies the memory overhead of data-intensive data analytics applications, presenting a serious obstacle to their widespread deployment.

Osprey is a memory management framework that enables SC to efficiently page to an SSD. It works at runtime and is transparent to SC applications, similar to classical OS virtual memory. This resolves a serious limitation in prior SC-aware memory management, which requires up-front planning and rewriting applications in a new programming framework.

Osprey achieves this using speculative execution. While speculative execution is powerful, it normally requires complex and error-prone in-kernel support, and therefore is not widely deployed for OS processes. Our central observation in Osprey is that, by carefully leveraging SC's obliviousness, we can make speculative execution for SC workloads practical and efficient. Osprey requires changing < 200 lines of code in each SC library that we tested, and no lines of code in SC applications written against those libraries.

1 Introduction

Data analytics is pervasive across applications and industries. But security and privacy concerns are a significant obstacle. For example, an organization may be hesitant to outsource its sensitive data to the cloud for data analytics. As another example, hospitals may want to combine their patient datasets to do research on rare diseases, or financial institutions may wish to combine their customers' data for fraud detection [14]—but privacy concerns prevent them from directly sharing their data with each other. These concerns have led to research on secure/private data analytics systems [2, 43, 47, 52, 57, 65, 66].

A compelling approach to secure data analytics systems is Secure Computation (SC). SC is a collection of techniques for computing on encrypted data, including Secure Multi-Party Computation (SMPC) and Homomorphic Encryption (HE). Google and Meta use SC in their advertising products [23, 46, 58], an industry consortium has formed around SMPC [40], researchers have used SC to facilitate secure data sharing for clinical research [2, 47], and Web3 wallets use SC to secure billions of dollars' worth of digital assets [16, 50].

Still, SC is not widely used for data-intensive analytics [2, 23, 57]. One reason is that SC is very resource-intensive (e.g., in CPU, memory, and network usage). In particular, SC can introduce a significant *expansion factor* on all encrypted data. For example, when using garbled circuits (a type of SMPC), each bit of plaintext takes up 16 B in encrypted form—a 128× expansion factor. As a result, SC-based data analytics systems run out of memory on even moderately-sized datasets [44, 57, 67, 69]. For example, the authors of Conclave report that SMPC "in practice only scales to a few thousand input records" (e.g., 30,000 records per party for a join) [57]. Once memory is exhausted, these workloads become infeasibly slow because the OS starts paging to secondary storage [44].

State of the Art. A promising direction is to make paging more efficient for SC workloads. There exist two approaches for this, memory programming and speculative execution. Both have significant downsides, which we now explain.

The first approach, memory programming, is to calculate the SC program's memory access pattern in advance, to pre-plan memory management [28]. This is possible because SC protocols are *oblivious*—that is, their memory access patterns are independent of the input data. Unfortunately, it requires changing applications to use a particular DSL/programming framework—a serious limitation because the SC ecosystem has a huge diversity of DSLs/frameworks [10, 12, 13, 20, 25, 29, 31, 42, 51, 56, 60], and it is not feasible to port all of them and their applications to a single DSL framework and runtime. Furthermore, this planner-based approach cannot respond to memory pressure at runtime, and produces a memory management plan that can be very large (e.g., gigabytes for a program that runs in just a few seconds).

The second approach, speculative execution, is to run the program speculatively [7, 8, 17, 30, 32, 41], *concurrently* with

the actual execution, to inform memory management [17]. If done by running a full duplicate execution, speculation incurs the program's memory overhead *again*, requiring more paging and worsening the problem. To address this, existing approaches use heuristics to skip I/Os and drop pages from memory when speculating. Unfortunately, this means that speculation may make mistakes—that is, "misspeculate." Process state must then be rolled back, which is complex and error-prone (e.g., implemented by "re-forking" processes [17]). Due to this complexity, no mainstream operating system, to our knowledge, uses speculative execution. It is only used in cases where the cost of a full duplicate execution is acceptable—for example, for straggler tasks in Hadoop and Spark [54, 63].

Main Insight. Our work uses an observation regarding *obliviousness*, a core property of SC, to overcome the challenges of speculative execution. There are actually *two* important aspects of SC's obliviousness. The first aspect, leveraged by prior work, is that access patterns remain unchanged regardless of program inputs, a property referred to as "input-oblivious" (IO). We identify a second aspect that prior work has largely missed: overwriting ciphertext data with arbitrary bytes during execution does not alter the program's memory access pattern. We refer to this as *"content-oblivious"* (CO).

The key insight behind our approach is that **CO enables us to eliminate misspeculation entirely, making speculative execution practical**. Traditional speculation requires dropping pages to control memory overhead, which inevitably leads to misspeculation. In SC, however, we can safely drop any data that is CO. Since ciphertexts, which dominate memory usage, are CO, discarding them dramatically reduces memory overhead without risking misspeculation.

Normally, speculative execution requires executing the program twice—speculatively and then concretely—doubling the CPU load. Yet, we circumvent this by leveraging CO again. We omit operations on ciphertexts during speculation and instead only record their would-be memory accesses. Because the dominant CPU cost arises from cryptographic operations on ciphertexts, this greatly reduces CPU usage of speculation.

By dropping CO data from memory and skipping computation on CO data, our approach addresses both the memory overhead and compute overhead of speculative execution. The result is a highly efficient speculative engine that provides timely guidance to the actual execution at a very low cost.

Design. Leveraging this insight, we design Osprey,[1] the first memory-management framework that runs SC workloads with *exceptionally low overhead* while being *transparent* to SC workloads. By "transparent" we mean *application-level* transparency: SC applications written against an SC library require zero source changes to use Osprey. SC library maintainers, however, must perform a small one-time port ($<$ 200 LoC in each library we tested, discussed in §3.2 and §9.7) to annotate memory allocations of CO data and computations over CO data. Osprey depends on these annotations for performance, not correctness, and it can bring substantial benefits even if the developer's annotations are not complete. Osprey uses speculative execution to predict an SC program's future memory accesses, and then uses those predictions to manage memory in the concrete execution, issuing `madvise` syscalls to prefetch and reclaim pages. We next describe four key challenges in realizing Osprey.

First, how and when should Osprey drop CO data during speculation? Prior speculation systems use complex heuristics to determine when to drop pages [17, §6.1]. We observe that Osprey can instead take a simple approach: it maps all virtual pages containing only CO data to a single physical page from the beginning, to avoid having to drop pages later. To do this, and to skip computations on CO data, we rely on changes/annotations in source code. However, these annotations can be confined to the *SC library* that defines the objects/classes for ciphertexts and provides functions to operate on them; this requires changing only a few lines of library code. The SC programs that *use* an SC library remain unmodified, making Osprey a transparent approach for legacy programs.

The second challenge is that each page may contain a mix of CO and non-CO data, preventing Osprey from being able to drop those pages when speculating. For example, pointers are *not* CO, so using `malloc` to allocate CO data mixes non-CO data (from `malloc`'s data structures) with CO data. To prevent mixing CO and non-CO data, Osprey uses a two-step solution. The first step is to use a dedicated section of its address space for CO data. The second step is to use a new kind of memory allocator that can allocate space in the CO region, while keeping the memory allocator's own pointer-rich data structures outside of the CO region. Non-CO data is duplicated between the speculative and concrete executions, so we design this memory allocator to have low memory overhead regardless of the allocation pattern.

Third, Osprey relies on the OS virtual memory system for flexibility (e.g., reacting to memory pressure, etc.), but this raises challenges for performance. Prior approaches achieve good performance via significant kernel changes—for example, by making policy decisions in the kernel [4] or modifying the page migration datapath [1]. However, we observe that most of the overhead of OS virtual memory comes from one operation: deciding which pages to reclaim from a process, for `cgroup` accounting. We remove this overhead by extending the `madvise` syscall with an option to reclaim specific pages. We also extend `userfaultfd` to support private anonymous segments. These simple, general-purpose extensions to Linux enable Osprey to run in user space with good performance.

Fourth, multithreaded applications are a special challenge, because the memory access patterns may differ between speculation and true execution. Furthermore, Osprey relies on page faults to trigger prefetching; multithreading complicates this because a different thread than intended may fault in a page. We address this problem with a novel use of Memory

[1] Osprey stands for "**O**blivious **S**peculation for **Pr**ogramming M**e**mor**y**."

Protection Keys (MPK) [21, 55]. We use MPKs to trigger page faults independently in each thread, so that each thread gets the same page faults regardless of the schedule.

We implemented Osprey in C++ and applied it to two SC protocol libraries: CKKS homomorphic encryption in SEAL [51] and garbled circuits in EMP-Toolkit [28]. We evaluated Osprey with 8 workloads. Osprey outperforms the operating system's virtual memory for all of the workloads, by up to 12× in the best case. For 7 workloads, Osprey performs comparably to or better than MAGE [28], the state-of-the-art SC-aware memory management system—but unlike MAGE, Osprey does not require developers to rewrite SC applications in a new DSL and is not limited to planning memory management in advance. For 6 workloads, Osprey performs within 60% of having enough memory to fit the entire computation. Osprey required changing less than 200 lines of code in each SC protocol library, and required no source code instrumentation in SC applications written against these libraries.

Our implementation of Osprey is available on GitHub at https://github.com/mpc-systems/osprey-project.

2 Background

2.1 Secure Computation

Secure Computation (SC) refers to techniques for computing on encrypted data. Below, we discuss two SC protocols that we use in our evaluation: *CKKS* and *garbled circuits*.

2.1.1 CKKS Homomorphic Encryption

CKKS [9] is a scheme for *Homomorphic Encryption (HE)*. A party can directly compute on HE ciphertexts to obtain an encrypted result, without ever decrypting the ciphertexts or having access to the decryption key. The input and output of the computation are both in ciphertext form (i.e., encrypted).

In the CKKS scheme, a ciphertext represents a vector of numbers. A party can compute element-wise addition and multiplication directly on the ciphertexts. The maximum depth in multiplications is limited by a *level* parameter chosen during key generation. Specifically, each ciphertext has a level, and each time a multiplication is performed, the result's level is one less than the operands' level (both operands must be the same level). Once the level reaches 0, no more multiplications can be performed. A procedure called *bootstrapping* can sidestep this limitation, but it is computationally expensive.

2.1.2 Garbled Circuits

Garbled circuits [61] are a protocol for Secure Multi-Party Computation (SMPC). SMPC enables multiple parties to compute a program over their private data and learn the result, without revealing each party's data to the others. To compute the function, the parties must communicate over the network.

Garbled circuits enable SMPC between *two* parties, called the *garbler* and the *evaluator*. The program to compute is represented as a circuit of AND and XOR gates. The garbler computes an encryption of the circuit, called the *garbled circuit*, and sends it to the evaluator. The evaluator obtains encrypted inputs of both parties, using a protocol called *oblivious transfer* for the garbler's input. The evaluator then executes each gate of the circuit—given the encryption of a gate, and the encrypted values of its input wires, the evaluator can compute the encrypted value of its output wire. In this way, they compute the encrypted values of all wires of the circuit. Then, they decrypt the output with the help of the garbler.

2.2 Memory Overhead

The memory overhead is a key factor that degrades the performance of SC protocols. Below, we discuss the memory demand in CKKS and garbled circuits.

For CKKS homomorphic encryption, the size of the ciphertext is determined by its level and the dimension of the vector. The higher the level, the larger the ciphertext, and the larger the dimension, the larger the ciphertext. For a vector with 8192 elements, the size of the ciphertext is around tens of kilobytes when the level is between 0 and 2.

For garbled circuits, the size of the garbled circuit is determined by the number of wires and gates in the circuit. While gates consume memory, they can be generated by the garbler and consumed by the evaluator in a streaming fashion, without ever all being materialized at once [22]. Additionally, only the wires that correspond to in-scope variables of the program being executed must be kept in memory; the values of out-of-scope or not-yet-computed wires may be discarded. Thus, the memory overhead of garbled circuits, for both garbler and evaluator, is dominated only by the program's variables that are live at any instant [27]. This can be large due to garbled circuits' large expansion factor; each bit of plaintext (i.e., a single wire) corresponds to a 128-bit ciphertext.

As we discuss in §10, prior work explores reducing the memory overhead of SC using approaches such as improving the cryptographic protocol, enhancing the SC framework, or finding ways to perform parts of the computation in plaintext. These techniques do not guarantee that memory will fit within a given limit. Memory programming and speculative execution (explained in §1) optimize performance given limited memory, and are complementary to the above approaches.

3 System Overview

Osprey is a memory management framework for SC workloads, with two goals. The first goal is to make virtual memory **efficient**, so that SC applications that do not fit in main memory can very efficiently page to an SSD. The second goal is to be **transparent to applications**, so that SC applications written against an SC library require no code changes

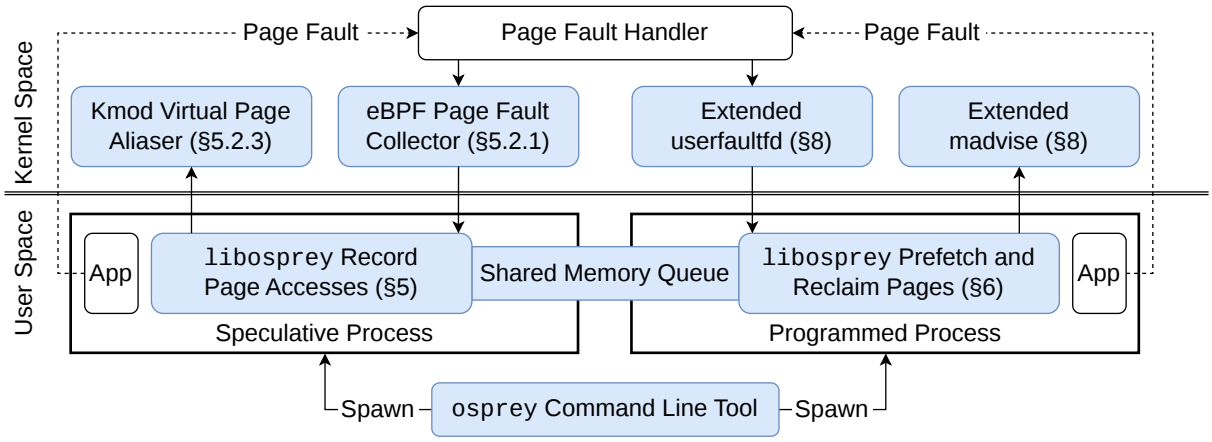


Figure 1: Osprey's system architecture. Osprey's components are marked blue.

to use Osprey. The SC library, however, must be annotated (i.e., lightly modified) to mark allocations of CO memory (e.g., ciphertexts) and computation on CO data (§3.2). These annotations enable Osprey's performance gains, but are not required for correctness; Osprey brings substantial performance benefits even with incomplete annotations (results in §9.7).

As shown in Fig. 1, Osprey executes an SC program as two concurrent user space processes. The first process executes a *speculative pass*, which transparently records the program's memory accesses; the second process executes a *programmed pass*, which uses the recorded memory accesses to manage memory. Both processes link against the same lightly ported SC library and use Osprey's *CO allocator* (§4) to place ciphertext data in a dedicated *CO region* of the virtual address space. Most of Osprey, including its core prefetching and speculation algorithms, lives in user space, within these two processes. To support these algorithms efficiently, Osprey also includes some extensions to the kernel. Osprey's kernel-side extensions comprise three components: (1) an eBPF program that forwards page-fault addresses to user space, (2) a very simple (124 LoC) kernel module that allows virtual pages to be efficiently aliased to the same physical page, and (3) general-purpose extensions to `userfaultfd` and `madvise`, which we anticipate could be mainlined.

3.1 Making Speculation Simple and Efficient

Naïvely running a speculative pass concurrently with the programmed pass would effectively double the program's memory usage and CPU usage. Osprey's main insight is to make the speculative pass very efficient, in both memory usage and CPU usage, by exploiting a property of SC called *content obliviousness (CO)*. CO is the property that overwriting ciphertext data with arbitrary bytes at runtime does not change the memory access pattern. We refer to memory that can be so altered without affecting the access pattern as *content-oblivious (CO) memory*. For example, ciphertext data is CO.

```
class Bit {
    __m128i bit; // ciphertext data (16 B)
};
class Integer {
    std::vector<Bit> bits;
};
```

Figure 2: Structure for an encrypted integer in EMP-Toolkit.

In the speculative pass, Osprey uses CO in two ways. First, Osprey maps all virtual pages containing ciphertext data to the same physical page. This eliminates most of the speculative pass's memory usage, because ciphertext data normally dominates memory usage (its expansion factor is the reason why SC programs are so memory-intensive). Second, Osprey skips computations on ciphertext data, instead recording their would-be memory accesses. This eliminates most of the speculative pass's CPU usage, because cryptographic routines for computing on the ciphertexts normally dominate CPU time. Crucially, CO means that these changes will not affect the access pattern recorded by the speculative pass.

3.2 Annotating SC Libraries

An important challenge in realizing Osprey is that not all of the memory in an SC program is CO. In particular, *any memory containing a pointer is not CO!* For example, stack memory is not CO, since the return addresses pushed to the stack constitute pointers and cannot be safely overwritten. As another example, SC libraries often provide "ciphertext" types to an application that contain metadata about the ciphertext, and a pointer to the actual ciphertext data allocated elsewhere. In Fig. 2, for example, the memory storing the `Bit` class is CO, because it directly contains ciphertext data. However, the memory backing an instance of the `Integer` class, which

```
// (a) Annotate CO Memory
class Integer {
  std::vector<Bit, COAllocator> bits;
};

// (b) Annotate Operations
Integer add(Integer a, Integer b) {
  Integer res;
  OSPREY_TOUCH(a, b, res);
  ...
  return res;
}
```

Figure 3: An example porting EMP-Toolkit to Osprey.

represents an encrypted integer, is not CO, because its memory contains (within the `std::vector` instance) the length of the integer in bits and a pointer to the actual ciphertext data, and changing these fields would affect the memory access pattern.

Osprey relies on annotations in the SC library to identify CO memory. Concretely, Osprey provides a memory allocator for CO memory that allocates all CO memory in a dedicated region of the virtual address space called the *CO region*. The SC library should use Osprey's CO memory allocator for ciphertext data. This is designed not to require modifications to the SC application, only to the type declarations in the SC library. In Fig. 3(a), for example, the only required change is to pass Osprey's `COAllocator` as the allocator template argument to the `std::vector` that holds an `Integer`'s bits. After this one-line change, the `Bit` elements stored in any `Integer`—no matter how the application constructs or uses it—reside in the CO region. The `Bit` class itself is unchanged.

Osprey reduces CPU overhead of speculation by skipping computations on CO memory. To enable this, we designed a one-line annotation API, `OSPREY_TOUCH` (Fig. 3(b)). In the programmed pass, `OSPREY_TOUCH` does nothing, but in the speculative pass it does the following: it (1) touches the memory region provided as an argument, and then (2) returns from the calling function. The idea is that by placing a call to `OSPREY_TOUCH` at the start of a function that implements a cryptographic operation, one skips the computation when speculating and simply records which memory the operation would have touched. It is not mandatory to use `OSPREY_TOUCH`, but more complete `OSPREY_TOUCH` annotations generally lead to greater CPU savings during speculation. We optimize this process further in §5.2.2.

Any non-CO memory, and computations on non-CO memory, are duplicated between the speculative and programmed passes. For SC programs, where ciphertexts dominate memory usage (due to their expansion factor) and cryptographic operations dominate CPU usage, this duplication is a tiny cost compared to running the SC program.

3.3 Transparent Interface

Osprey is implemented as two components: `osprey`, a command-line tool, and `libosprey`, a runtime library. Osprey-enabled SC libraries must be linked with `libosprey`, which makes Osprey's annotation APIs available to the SC library developer and adds a runtime component (e.g., threads) to execute Osprey's logic in the speculative and programmed passes. The `osprey` command-line tool is a frontend, allowing applications to be invoked as `osprey ./a.out`. It spawns the speculative and programmed passes as child processes, and it sets up a shared-memory buffer to transfer predicted memory accesses from the speculative pass to the programmed pass.

Osprey requires the source code of the SC library—but not the SC application that uses it—to be annotated, by way of calls into `libosprey`. Osprey is designed to make these annotations as minor and lightweight as possible (i.e., requiring very few changes to the SC library source code).

3.4 Comparison with Prior Approaches

Prior work on memory programming, like Osprey, predicts future memory accesses and exploits the obliviousness of SC. In this sense, Osprey combines ideas from both speculative execution and memory programming—it may be considered "speculative memory programming." However, by leveraging CO, Osprey overcomes the shortcomings of both approaches.

The state-of-the-art speculative execution system for memory management [17] can misspeculate, hurting performance and requiring complex in-kernel support (e.g., checkpointing or re-forking). In contrast, Osprey leverages CO in a way that ensures that the speculative pass will never misspeculate, improving performance and avoiding complexity.

The state-of-the-art memory programming system, MAGE [28], exploits only the weaker notion of *input obliviousness (IO)*—the property that changing an SC program's inputs does not change the program's memory access pattern. This leads to three shortcomings: (1) MAGE requires SC applications to be rewritten in a DSL, (2) MAGE requires SC libraries to be rewritten to work with MAGE's backend, and (3) MAGE's planner materializes a memory management plan in a file that is concretely very large, and proportional in length to the program's execution time. By exploiting CO and not merely IO, Osprey addresses all three shortcomings: (1) it avoids changes to SC applications, (2) it requires only minor annotations to SC libraries, and (3) the speculative pass runs "on the fly," concurrently with the programmed pass, so no "plan" must be materialized.

4 Content-Oblivious Memory Allocator

In Osprey, we design a new memory allocator that is suitable for allocating memory in the CO region. We refer to it as the "CO allocator." The CO allocator keeps *all* of its data

structures in regular memory. The reason for this is that a memory allocator's structures (pointers, bitmaps, etc.) are not CO and therefore cannot be kept in the CO region.

Existing memory allocators are unsuitable for allocating CO memory because they place their own pointer-rich data structures in or next to the blocks that they allocate. Hoard [3], `tcmalloc`, and `mimalloc` use linked lists for each allocation size, storing a pointer—which is not CO—on each free block. `jemalloc` [15], which uses a bitmap instead of a list, is a step in the right direction, as the bitmap is not mixed with the data. But `jemalloc` still relies on the metadata and data for each run[2] being nearby in the address space, to easily find the metadata for a data block (e.g., when `free`'ing it).

For Osprey's memory allocator, we use a slab allocator design, to minimize the space overhead of our allocator. Unlike existing slab allocators, we store slab data in the CO region, but slab metadata in regular memory. Whereas prior allocators depend on the data and metadata being nearby (e.g., adjacent) in the address space, our design requires a way to "link" the two. To get from the metadata for a slab to its data, we can simply include a pointer in the metadata. However, getting from a slab's data to its metadata—necessary to `free` an object—is tricky, as we cannot place a pointer in CO memory.

A simple way to map from slab data to slab metadata is to create a hash table in regular memory for this purpose. An early version of Osprey did this, but we found that it measurably impacted performance. Therefore, Osprey uses an array-based mapping instead. We treat the CO region as an array of slabs, and keep a parallel array in regular memory that stores, for each slab, a pointer (8 B) to its corresponding slab metadata (or `nullptr` if unallocated). We set the slab size to 2 MiB and the CO region to 1 TiB; for these parameters, our array-based mapping is only 4 MiB, which is negligible compared to the actual data of an SC application.

5 Speculative Pass

5.1 Collecting Memory Accesses

To collect memory accesses, a straightforward solution is binary instrumentation [33]: inserting instructions around each memory access instruction to record the address. This strawman is known to be very inefficient and produces memory access traces that are impractically large [4, 28].

For Osprey's purpose, memory management, it is sufficient to collect a more coarse-grained access trace, at page granularity. To achieve this goal, we rely on page faults to detect accesses to new pages so their addresses can be recorded [4,6]. The lazy allocation strategy used by kernels like Linux ensures that a page fault is triggered on the first access to each page, giving Osprey a chance to record its address. This can be achieved without modifying the target program.

[2] A "run" is the slab-like structure in `jemalloc`.

So far, this approach records only the first access to each page. To address this, we limit the number of pages that can be mapped simultaneously, by unmapping pages that were mapped in on prior page faults. This causes subsequent accesses to those pages to trigger page faults, allowing us to detect and record subsequent accesses to those pages. Allowing few pages to be mapped at a time allows for a more precise trace, at the cost of making the tracer bigger and longer to collect. Allowing many mapped pages makes the trace shorter and faster to collect, but makes it less precise.

Multiple solutions have been proposed for choosing which pages to unmap in this scheme, including FIFO [6]. Osprey defaults to the *microset* strategy from 3PO [4], which unmaps *all* mapped pages once the number of mapped pages reaches a threshold; we call that threshold the *window size*. We use this strategy because it allows the access trace to be interpreted as a sequence of working sets of the program.

5.2 Making Collection Efficient

3PO uses a similar page-fault-based approach to trace collection, but implements it entirely in the kernel for efficiency and performs it during a preparation/planning phase where performance is not critical. We now contribute techniques to perform trace collection without significant changes to the kernel source, and to make trace collection efficient enough to be run alongside the program's main execution.

5.2.1 Avoiding Kernel Source Changes

An intuitive solution is to use *userfaultfd* [62], a Linux feature that allows an application to register a custom page fault handler in user space. However, this approach adds overhead to each page fault, requiring additional user-kernel mode switches and thread scheduling operations.

Instead, our approach is to inject minimal logic into the kernel using eBPF, hooking into a pre-handler in the page fault handler. When a page fault occurs, Osprey's eBPF module writes the address and relevant error information into a ring buffer shared between kernel space and user space. In user space, a separate thread asynchronously reads page fault addresses and metadata from the ring buffer, and performs the remaining parts of the collection logic as described in §5.1. This hybrid approach balances performance with flexibility, avoiding kernel source changes while achieving faster performance than `userfaultfd`.

5.2.2 Accelerating Collection with `OSPREY_TOUCH`

We use `OSPREY_TOUCH` (§3.2) to not only skip the cryptographic computations, but also accelerate trace collection. Annotating the data touched by cryptographic operations with `OSPREY_TOUCH` completely bypasses the faulting mechanism and explicitly provides the tracing algorithm with the addresses that would have been accessed.

Note that Osprey collects memory accesses using eBPF and `OSPREY_TOUCH` at the same time. To maintain their order, it timestamps each access when it is recorded and then uses these timestamps to correctly order them during aggregation.

5.2.3 Aliasing Memory Efficiently

As described in §3.1, Osprey maps all content-oblivious memory to a single physical page in the speculative pass. The most straightforward approach is to create a 4 KiB in-memory file and repeatedly use the `mmap` system call to map each virtual page to this file. However, each such mapping creates a separate Virtual Memory Area (VMA) in the Linux kernel. An excessive number of VMAs leads to two major issues: 1. Increased overhead for the kernel to search VMAs, which are organized as a red-black tree. 2. Unnecessary memory overhead from maintaining metadata for each VMA.

Our solution is to implement a simple (124 LoC) kernel module providing a virtual device, `/dev/aliased`. In the speculative pass, Osprey creates the CO region by using `open` and `mmap` on `/dev/aliased`. In our kernel module, we implement a custom `mmap` handler for this device, so that the memory region produced by `mmap` (i.e., the CO region) has all virtual pages mapped to a single 4 KiB physical page, while using only a single VMA for the whole region.

5.3 Isolating Side Effects

For speculation, isolation of side effects is just as critical as performance optimization. In particular, interactions with external systems—such as disk and network I/O—must be isolated during speculative execution.

For disk isolation, we utilize `overlayfs` to achieve transparent separation. By creating a read-only mapping of the existing working directory, the program under speculative execution observes a file system view identical to that of normal execution. Any modifications to existing files or creation of new files are transparently redirected to newly created files in the upper layer of the overlay. With the help of `overlayfs`, we achieve seamless and effective disk isolation.

For SMPC protocols (e.g., garbled circuits; see §2.1.2), it is also necessary to isolate the network interactions of the speculative pass. We observe that SMPC typically only involves network communication in the following scenarios: (1) during initialization, to establish network connections; (2) during teardown, to disconnect from the network; and (3) during arithmetic or boolean operations on ciphertexts, which involve data exchange over the network.

For the first two cases, we achieve isolation with minimal changes by inserting `OSPREY_TOUCH` without providing address ranges to bypass the network connection setup and teardown phases. For the third case, since `OSPREY_TOUCH` is already present in the computation logic—as described in §5.2.2—and inherently includes the ability to skip operations, it naturally avoids triggering this type of network interaction. Through our APIs, we provide a lightweight and minimally invasive mechanism for achieving complete network isolation during speculative execution.

6 Programmed Pass

6.1 Page-Fault-Based Synchronization

In the programmed pass, the *memory programmer* is responsible for processing the memory access sequence streamed from the speculative pass, in order to prefetch and evict pages. To do so, the memory programmer must keep track of the program's execution position within the memory access sequence. This is crucial so that Osprey can prefetch each page in the future memory access sequence at the right time.

Inspired by 3PO [4], we use a mechanism based on page faults to synchronize memory access processing and program execution. In short, we periodically select a **key page** every fixed-size batch of memory accesses along the predicted memory access sequence. We ensure that, once a key page is selected, the next access to that page triggers a page fault. 3PO ensured this property by always selecting, as the key page, a page that is not currently mapped in the page table. In Osprey, our approach is instead to first select a key page, and then alter its page table entry so that a page fault occurs the next time it is accessed; we explain this mechanism in §7.3. When a page fault is triggered, the memory programmer can accurately determine the current execution position within the memory access sequence (i.e., the execution position is at the chosen key page). Based on this position, it prefetches the next batch of pages along the memory access sequence and designates the next key page. This repeats on each key page, giving the memory programmer a chance to periodically execute at known positions along the memory access sequence, until the entire memory access sequence is processed.

Whereas 3PO places its prefetcher entirely in the kernel, we use `userfaultfd` to implement the memory programming logic purely in user space. When a page fault occurs, it is captured by `userfaultfd`, giving the memory programmer in user space a chance to run. We contribute an eviction codepath and the requisite user/kernel APIs so that the memory programmer can efficiently prefetch and evict pages.

6.2 Asynchronous Swapping Mechanism

The batching optimization described above reduces user-kernel mode switches, but does not mask disk I/O. To deliver satisfactory performance, the missing piece is **asynchrony**.

To this end, we carefully design the logic for handling a batch of memory accesses as two decoupled steps: (1) **Issuing I/O requests** to asynchronously load the pages corresponding to the memory access sequence from disk into memory. (2)

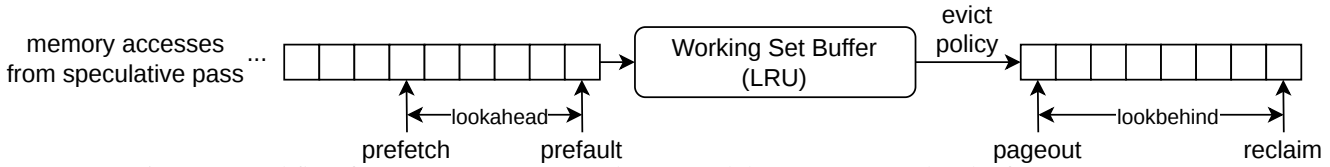


Figure 4: Workflow for Osprey's programmed pass. Each box represents a batch of memory accesses.

Waiting for completion of the I/O requests, ensuring the pages are fully loaded into memory and ready for access.

The time gap (in batches) between Step 1 and Step 2 is controlled by a tunable parameter called **lookahead**. By appropriately setting the lookahead value, we can ensure that memory pages are prefetched and ready by the time they are needed by the program—thus achieving a nearly wait-free memory swap-in mechanism.

While the above supports prefetching pages into memory, we still need a swap-out strategy to complement it. Initially, we tried relying on `cgroup`s to limit memory usage, relying on OS mechanisms to swap out pages. Unfortunately, this resulted in poor performance due to the swap-out mechanism in the kernel: on every kernel-to-user transition, the system checks the `cgroup`'s configured memory limits, and when the limit is exceeded, it synchronously selects and then swaps out pages. This takes significant time and ultimately becomes a performance bottleneck.

To address this, and to complement our asynchronous swap-in mechanism, we implement a *fully asynchronous swap-out mechanism* as well. This process consists of two steps: (1) Issuing disk I/O requests to begin swapping out a batch of memory pages. (2) Polling for completion to ensure the pages are safely backed up to disk, after which they are unmapped from the virtual address space. The time gap (in batches) between Step 1 and Step 2 is controlled by a tunable parameter called **lookbehind**.

6.3 Swap-Out Policy Design

Osprey's speculation approach provides only a limited window of future accesses, precluding MAGE's approach of using Belady's algorithm. Different swap-out policies have only limited impact on overall performance. We therefore implement a simple, intuitive, and lightweight strategy that is well-suited to our system design. Our policy addresses two key questions:

- **When** should the system begin swapping out?
- **Which** pages should be selected to be swapped out?

6.3.1 When to Swap Out

Inspired by Linux `cgroup`s, we define three memory thresholds to control swap-out behavior:

- `mem-limit-low` ("low watermark"): Indicates low memory pressure. No swap-out operations are triggered.
- `mem-limit-high` ("high watermark"): Memory usage has reached a high watermark. The system begins asynchronous swap-out operations to reduce usage to the low watermark.
- `mem-limit-max` ("maximum watermark"): Memory usage has reached the maximum allowable limit. The system halts speculative execution (stop-the-world) and immediately evicts all *content-oblivious* memory pages.

6.3.2 Which Pages to Swap Out

Our page selection strategy is guided by two eligibility conditions: First, the page must have resided in memory for at least a certain amount of time. This duration is controlled by the parameter `execnow`. Second, the page must be known to not be accessed again in the near future, based on the window of future accesses. Among the candidate pages satisfying the above conditions, we apply the following eviction priorities:

- **Priority 1:** Prefer evicting pages that are likely to be non-dirty, based on read/write information provided by the speculative pass. Evicting non-dirty pages does not require disk writes, so their eviction incurs less overhead.
- **Priority 2:** Among remaining pages, select those that have stayed in memory the longest since being swapped in—this follows an LRU-like strategy.

This policy enables efficient and controlled swap-out while minimizing interference with program execution.

7 Multithreaded Applications

Because SC applications must have CO access patterns, it is generally possible to write them in such a way that each thread operates on a fixed, predetermined partition of the input data, in order to avoid most multithreading issues. Prior work [28] leveraged this fact, and this property holds in our evaluation.

Still, the partitions assigned to the threads may partially overlap. For example, when parallelizing a matrix-vector multiplication, the matrix may be split among the threads, but the vector is shared (i.e., the vector is in every thread's partition). Therefore, we design support for multithreaded SC applications that can work even if threads share data.

7.1 CO Allocator with Multithreading

We provide each thread with its own CO region for its allocations and its own CO allocator. This ensures that, although different executions may interleave the threads' allocations differently, they will still have the same memory layout. We separately collect the memory access trace for each thread in

the speculative pass, and run a separate instance of the memory programmer per thread in the programmed pass. This "one trace per thread" design helps eliminate scheduling-induced nondeterminism and synchronization overhead.

7.2 Speculative Pass with Multithreading

Thread scheduling nondeterminism impacts the page-fault-based memory access collection and the memory programming strategy. We illustrate this with two concrete scenarios:

Scenario 1: Fault Skipping. Suppose two threads access a shared data segment sequentially. The first thread accesses the data and triggers a `page fault`. When the second thread accesses the same data later, no fault will be triggered. This results in incomplete memory access logs during the speculative phase. A high degree of such omissions significantly degrades the performance of the memory programmer.

Scenario 2: Out-of-Order Access. Assume thread 1 and thread 2 access memory regions A and B respectively. During speculative execution, the collected memory access sequence is $[A,B]$. However, due to nondeterministic scheduling during actual execution, thread 2 may run ahead and access region B before thread 1 accesses region A, resulting in the real access sequence $[B,A]$. The memory programmer, still operating under the assumption of the $[A,B]$ sequence, designates key page 1 in region A and key page 2 in region B. When key page 1 triggers a page fault, it swaps in region B next, expecting key page 2 to be the next fault trigger. However, since region B has already been accessed, no fault occurs. This breaks the synchronization between the memory programmer and the actual program execution.

A simple solution, as employed by prior work [4], is to bind all threads to a single CPU core when collecting the access trace, effectively serializing the multithreaded execution. While this approach guarantees determinism, it imposes a severe performance penalty—particularly unacceptable for Osprey, where the speculative pass occurs at runtime.

Instead, two of Osprey's key design choices enable Osprey to sidestep these challenges and run the threads in the speculative pass on separate cores. First, because the `OSPREY_TOUCH` APIs enable collecting the memory access trace without triggering page faults, it decouples the memory access trace from thread scheduling. Thus, if the SC library is fully annotated with `OSPREY_TOUCH` APIs, a fully deterministic memory access sequence can be collected for each thread. Second, because we use a separate trace for each thread, the threads' memory access patterns are processed separately regardless of their interleaving.

7.3 Programmed Pass with Multithreading

As explained in §6.1, in the programmed pass, Osprey uses page faults on key pages for synchronization purposes. Therefore, it is essential that, once a key page is selected, the next access to that page triggers a page fault. The multithreaded setting makes this significantly more challenging to achieve. Below, we first describe how we achieve this in the single-threaded setting, and then discuss the additional challenges due to multithreading and our approach to overcoming them.

7.3.1 Single-Threaded Setting

After selecting a key page, a natural approach is to use `madvise` to ask the kernel to unmap the page. Unfortunately, this approach does not work reliably—the kernel may not immediately take the advice. To address this, our approach for the single-threaded setting is instead to use `mprotect` to set page permissions to `PROT_NONE`. This strategy is attractive because it is lightweight, deterministic, and synchronous.

7.3.2 Multithreaded Setting

In the multithreaded setting, additional nondeterminism arises from interleaving of threads. Osprey's prefetcher must remain synchronized with each thread. But page faults, which Osprey uses for synchronization, occur at a per-process level—a page fault on a given page fires at most once (until the page is evicted again), regardless of which thread caused it.

As a concrete example, consider matrix-vector multiplication, in which the matrix is partitioned row-wise across threads but the vector is shared by all threads. Suppose the memory programmer for thread A picks a page backing the shared vector as A's next key page, and evicts that page so A's upcoming access will fault. If another thread B accesses the page first—a likely event, since every thread also scans the vector—the access faults and `userfaultfd` delivers it to the handler. But this fault belongs to B, not A: the handler sees an unexpected access and simply maps the page in. When thread A later reaches the page, the access no longer faults (the page is already resident), so A's expected key-page fault never arrives, and the memory programmer for A loses synchronization for the remainder of the run.

The root cause of this issue is that all of the threads share the same page table, so that a page fault that occurs in one thread prevents page faults on that page from occurring in other threads. In principle, this could be solved by changing the kernel to have a separate page table for each thread, but this would be a complex and serious change.

Instead, we observe that we can get the intended effect entirely in user space by using Intel Memory Protection Keys (MPKs) in a novel way. Our use of MPKs solves all three issues, producing a robust solution for Osprey to support multithreaded applications. The idea is to trigger page faults using the MPK bits in the page table entry. Each thread is assigned an MPK ID, and a page can be marked as the "key page" for a thread in such a way that *only* that thread would get a page fault on that page. If another thread were to page fault on a thread's key page and map it into the address space,

the MPK field would still guarantee that the intended thread will get a page fault when accessing it.

This MPK-based approach (i.e., using `pkey_mprotect` instead of `mprotect`) brings efficiency advantages, and our Osprey implementation uses it even in the single-threaded case.

8 Kernel Modifications

To support efficient and reliable memory programming, we implemented several new APIs and kernel-level extensions based on Linux 5.15.0. These extensions fill critical functionality gaps in the existing Linux kernel interface and significantly broaden the applicability of our system.

- **New `madvise` Flag: `MADV_RECLAIM`**
  We introduced a new `madvise` operation, `MADV_RECLAIM`, which synchronously writes the specified pages to the swap space and removes their mapping from memory.
- **Extending `userfaultfd`: `PRIVATE_ANON` Memory**
  The Linux kernel's `userfaultfd` interface has traditionally only supported monitoring both minor and major page faults for `SHARED_ANON` memory regions. However, writebacks on `SHARED` memory pages are conservative and always trigger I/O regardless of dirtiness, due to ambiguity introduced by multiple PTEs. We extended `userfaultfd` to support `PRIVATE_ANON` memory as well, enabling precise and efficient tracking of faults without unnecessary disk I/O.
- **Extending `userfaultfd`: Handling Access Violations**
  Accessing pages marked as `PROT_NONE` using `mprotect` results in access violation faults (`SEGV_ACCERR`). The original Linux `userfaultfd` interface does not forward such faults to user space for handling, breaking the ability to programmatically respond to controlled faults used in our key page strategy. We extended `userfaultfd` to intercept and forward access errors caused by protection violations, enabling full support for page-fault-driven progress tracking.

These kernel modifications support our design goals while being general enough to be potentially mainlined.

9 Evaluation

9.1 SC Protocols

We evaluate Osprey using two types of SC: homomorphic encryption (HE) and SMPC. For HE, we use CKKS [9] as implemented in the Microsoft SEAL library [51]. For SMPC, we focus on Yao's Garbled Circuits [61] with the HalfGates optimization [64], provided by EMP-Toolkit [60].

9.2 Workloads

For evaluation, we use workloads representing data-intensive "kernels" that could be part of larger SC applications, like data analytics or ML/linear algebra applications. Using the same workloads as MAGE allows for a convenient comparison of MAGE and Osprey. To use them with Osprey, we re-implemented them to directly use the SC libraries instead of using MAGE's DSL; in cases where the SC libraries supported slightly different primitive operations, we sometimes modified the implementations of the MAGE workloads to keep the computational work similar between the two.

For CKKS (SEAL), we use the following workloads:

- **Sum**: *Sum a list of numbers and store it as a new ciphertext.*
- **Statistics**: *Calculate the mean and the variance of a list of numbers and store it in two new ciphertexts.*
- **Matrix-vector multiply**: *Multiply a matrix with a vector and store the result in a new vector.*
- **Tiled matrix multiply**: *Matrix multiply with manual tiling optimizations.*

CKKS ciphertexts can encrypt vectors of numbers and operate on them in a SIMD fashion. In Osprey, we treat each ciphertext as a single element, so that our workloads could be applied to many instances of each problem (one per slot in the vector) without additional overhead. While techniques exist to use the vectorized nature of these protocols to accelerate a single problem instance, they are out of scope for our study.

For garbled circuits (EMP-Toolkit), we use the workloads:

- **Merge sorted**: *Merge two sorted lists into a larger list.*
- **Full sort**: *Sort a list in place.*
- **Loop join**: *Join two tables into a larger table.*
- **Matrix-vector multiply**: *Multiply a matrix with a vector and store the result in a new vector.*

We do not include the **binfclayer** workload from MAGE [28]. The reason is that **binfclayer** calls the SC library at a very fine granularity (e.g., processing one bit at a time), calling `OSPREY_TOUCH` very frequently. Though the final access trace is small, the speculative pass takes a long time, even longer than the programmed pass. The programmed pass still achieves good performance for this workload (e.g., if the speculative pass is run separately in advance).

9.3 Empirical Methodology

We compare Osprey's performance to three baselines:

1. *Unbounded.* We run the original program with enough memory to fit the whole program, without any `cgroup`. This gives a lower bound on the program's runtime.
2. *OS Swapping.* We run the program with a `cgroup` that limits the available memory, resulting in OS swapping. This is an upper bound on time that Osprey should beat.
3. *MAGE.* We use MAGE, which plans paging in advance, to execute a version of the program written in MAGE's DSL.

Since MAGE performs its planning in advance, whereas Osprey performs paging on the fly, we expect MAGE to, in general, outperform Osprey. Note that our measurements for Unbounded and OS Swapping are not comparable to those presented in MAGE's evaluation [28, §8] because we use a program written directly against the SC library for these base-

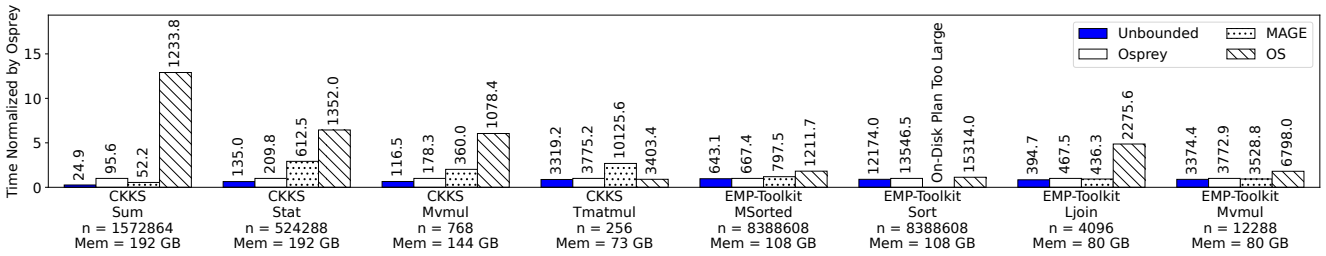


Figure 5: Performance of Unbounded, OS Swapping, MAGE, and Osprey, normalized by the time for Osprey. Each x-axis label gives the protocol, workload name, problem size n, and the workload's approximate unbounded memory consumption (e.g., *CKKS, Sum, n = 1572864, Mem = 192 GB*); under the bounded configurations, all workloads are capped at 32 GB (Table 1).

Parameter	Value
low watermark	24 GB
high watermark	28 GB
maximum watermark	32 GB
lookahead	4 batches
lookbehind	2 batches
workingset	2 batches
batch size	131072 per thread
window size	32768 per thread

Table 1: Configuration for Osprey in our evaluation.

lines. In contrast, MAGE's evaluation, for these baselines, uses MAGE's own interpreter to run the SC protocols, to normalize for differences in performance between MAGE's protocol backend and SC libraries [28, §8.3]. As a result, MAGE's "Unbounded" and "OS" baselines incur additional costs that our baselines do not, like serialization/deserialization costs for SEAL ciphertexts (explained in §9.4).

We ran experiments on rack servers with INTEL(R) XEON(R) GOLD 5520+ @ 2.20 GHz in an 8×32 GB @ 4800 MT/s memory configuration. We configured swap space on a 4 TB Micron 7450 PRO SSD, with theoretical read and write speeds of 6800 MB/s and 5300 MB/s, respectively. For SMPC workloads, we run the two parties on two servers connected via a direct bidirectional 200 Gbit/s link, using the Linux network stack; our actual network bandwidth usage was much less, likely $\approx$ 10 Gbit/s or less. While SMPC is often run with a wide-area network, previous experiments [28, §8.7 and §8.8] using MAGE suggest that, with appropriate TCP tuning, a wide-area network is not a significant performance bottleneck for HalfGates compared to a local network like we used in our experiments.

Across all experiments, Osprey uses the same set of configurations, shown in Table 1. All runtimes reported in this section are end-to-end and include Osprey's warm-up: spawning the speculative and programmed passes as child processes, attaching the eBPF page-fault pre-handler, initializing the custom kernel module that backs the CO region, allocating the shared-memory access-trace ring buffer, and filling the first batch of the trace before the programmed pass begins prefetching. Warm-up is a fixed, one-time cost: it does not grow with workload size and therefore is amortized over longer runs.

We do not run Osprey in a `cgroup`, since we observed that the OS would swap out pages even though Osprey stays within the limit. Instead, we rely on Osprey's own internal mechanisms (low watermark, high watermark, etc.) to bound its memory usage; to check that the memory usage is as intended, we poll the RSS inside Osprey, and also check memory usage externally using `htop`. For the other configurations—Unbounded, OS, and MAGE—we limited memory to 32 GB.

9.4 Performance Comparison

We compare performance across our workloads in Fig. 5.

Osprey consistently improves performance compared to OS and is often competitive with Unbounded. For many workloads it provides nearly the same performance as Unbounded while using a fraction of the memory. For workloads that are less compute-intensive, like the Sum and Stat workloads with SEAL, Osprey outperforms OS by up to 12×.

For CKKS/SEAL workloads, MAGE performs worse than Osprey in most workloads. This is because MAGE's CKKS backend requires each SEAL ciphertext to be serialized and deserialized on each operation, adding computational overhead [28, §7.4 and §8.3]. For the tiled matrix multiplication workload, for example, this serialization/deserialization in MAGE's backend accounts for $\approx$ 40% of the overall runtime. In contrast, Osprey and OS can work with programs that are directly written against the SEAL library, and therefore do not have these overheads. The only workload where MAGE outperforms Osprey is the Sum workload. The reason for this is that this workload is so memory-intensive that the overhead from `madvise` dominates the total runtime. MAGE avoids this cost because its memory management happens entirely in user space via async I/O operations, bypassing the kernel's virtual memory system.

For EMP-Toolkit workloads, MAGE uses its own re-

implemented version of EMP-Toolkit to remove overheads caused by memory-irrelevant inefficiencies, such as frequent virtual function calls through a vTable and a small network buffer that underutilizes available throughput. We applied the same optimizations used by MAGE to the baseline EMP-Toolkit so that their unbounded performance becomes comparable (within 5% across the evaluated workloads). Under memory constraints, the performance difference between Osprey and MAGE remains below 10%. This at-most 10% gap provides an upper bound on the overhead introduced by Osprey's use of a suboptimal eviction policy. In particular, unlike MAGE, which plans offline with full knowledge of the entire memory access trace and can apply Belady's algorithm, Osprey must make decisions online and only sees a limited window of future accesses.

The performance of the matrix multiplication workload is dependent on the tiling optimization. Without tiling, Osprey performs worse than OS. The reason is that Osprey's speculative pass produces much more output for a naïvely written matrix multiplication, as the microset strategy for compressing the output is far less effective; this causes the programmed pass to spend extra time prefetching pages already in memory. Even so, the performance degradation is small, within 2× of the tiled case (and still better than MAGE).

9.5 End-to-End Applications

The workloads above are data-intensive kernels; we now turn to end-to-end SC applications drawn from prior work [2, 28, 44, 59]: password reuse detection ("pwd") and comorbidity analysis ("com"). The password reuse detection workload uses EMP-Toolkit's HalfGates implementation to check, in two-party SMPC, whether any of the n users in one party's dataset appears with a matching password hash in the other party's dataset. Each dataset contains user IDs and password hashes. We use $n = 8388608$ (2^{23}), for which the workload's unbounded memory footprint is approximately 163 GB. The comorbidity analysis workload checks whether patients in a cohort with a specified diagnosis also jointly have other diagnoses. It takes as input a cohort table containing the patients with the given diagnosis and a diagnosis table containing diagnosis records for all patients. The task is to join the cohort patients with the diagnosis table, count the occurrences of each diagnosis among the cohort, and return the top-10 most common diagnoses. We use $n = 32768 \times 1000$: the cohort table contains 32768 rows, and the diagnosis table is 1000× larger. The workload's unbounded memory footprint is approximately 91.9 GB. All bounded configurations are capped at 32 GB (Table 1). Fig. 6 reports end-to-end runtime.

For password reuse detection, Osprey achieves near-unbounded performance under the memory cap. For comorbidity analysis, Osprey substantially outperforms OS because the workload has a highly structured, phase-by-phase memory access pattern. The first phase linearly scans patient IDs; the second phase performs a bitonic sort on the valid bit; and from the third phase onward, the computation primarily focuses on the leading n rows of the large table. Without application-level hints, the OS replacement policy preserves data that is not accessed again in the future. Osprey identifies the phase structure and preserves the useful working set, thereby achieving performance close to the unbounded configuration.

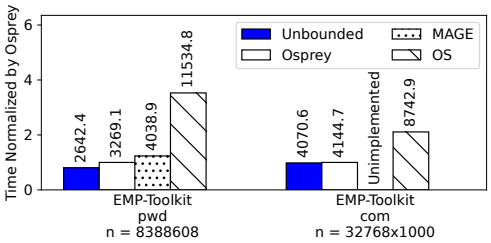


Figure 6: SC applications' end-to-end runtime.

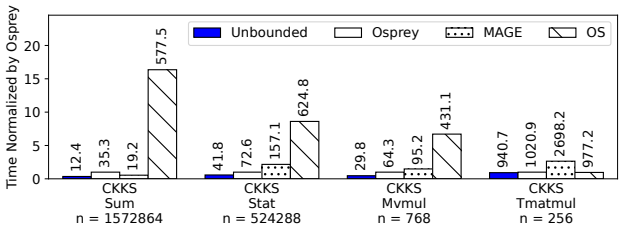


Figure 7: Multithreaded performance with four threads.

9.6 Impact of Parallelism

To evaluate Osprey's support for multithreaded programs, we focus only on SEAL. The reason is that, whereas SEAL allows applications to use HE from multiple threads, the EMP-Toolkit framework only supports single-threaded applications.

Fig. 7 shows the results using four threads. Fig. 8 shows the results as we vary the number of threads for the matrix-vector multiply and tiled matrix multiply workloads.

The performance gap between OS Swapping and Osprey grows under multithreading, reaching up to 16×, because OS Swapping is bottlenecked by I/O latency. Osprey avoids this bottleneck through prefetching.

For matrix-vector multiply, Osprey scales linearly with the number of threads and consistently outperforms both MAGE and OS Swapping. For matrix-matrix multiply, Osprey is slightly slower than OS Swapping when using 1, 2, or 4 threads because the workload is highly compute-intensive. However, when the thread count increases to 8, Osprey becomes faster than OS Swapping. As compute resources grow, the bottleneck shifts from computation to memory, a scenario that Osprey handles more efficiently.

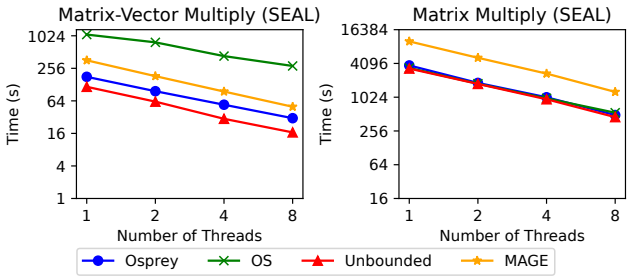


Figure 8: Multithreaded performance.

SC Library	Minimal	Recommended	Total Library LoC
SEAL	96	143	69211
EMP-Toolkit	7	106	9207

Table 2: Lines of code changed for Osprey.

9.7 Source Code Modifications to SC Library

With Osprey, we aim to require no source code changes to SC applications, and minimal changes to SC libraries. We count the lines of code we changed using `git diff --stat` and show the results in Table 2. The "Minimal" column shows the number of lines changed to use Osprey's CO memory allocator and isolate side effects in the speculative pass. The "Recommended" column includes both the "Minimal" changes and `OSPREY_TOUCH` annotations in the cryptographic routines to get the best performance. Overall, Osprey requires changing only a few lines of code in each SC library, compared to the total size of these libraries ("Total Library LoC").

In contrast, MAGE requires developers to rewrite the application in a special DSL. MAGE also requires integrating new SC libraries via implementing new protocol drivers, which can add significant overheads (e.g., serialization costs described in §9.4) or require significant rewriting effort.

9.8 Benefits of Data Annotations

Fig. 9 shows the performance difference between the two levels of Osprey integration in SC libraries explained in §9.7: "Minimal" (using CO memory allocator and isolating side effects in the speculative pass) and "Recommended" (using the `OSPREY_TOUCH` API, in addition to the "Minimal" changes). We evaluate both CKKS and EMP-Toolkit using the single-threaded matrix-vector multiplication workload.

From the "Calculation Time" graph, we see that "Minimal" and "Recommended" integration lead to almost identical runtime performance. This result shows that it is very easy to adopt Osprey in an SC library when the main goal is simply to preserve performance.

"CPU Usage per Thread" represents the average CPU consumption of each thread in an SC workload. Since Osprey

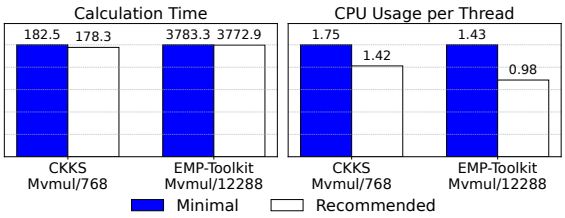


Figure 9: Comparison between "Minimal" and "Recommended" changes for single-threaded workloads.

creates two background threads for every application thread, i.e., one speculative thread and one programming thread, the average CPU usage per application thread can exceed 1. From the "CPU Usage per Thread" graph, we observe that "Recommended" integration significantly reduces CPU usage, by up to 45%. This means that for users who care not only about performance but also about reducing resource consumption, adding data annotations can be very helpful. Importantly, data annotations are not an all-or-nothing feature but are gradual: the more complete the annotations are, the more resource savings the system achieves.

A notable case is EMP-Toolkit: under "Recommended" integration, its CPU usage per thread drops below 1. This result shows that EMP-Toolkit can achieve over 90% of the unbounded performance while using only about 30% of the memory and the same amount of CPU (at most 1 CPU per application thread).

10 Related Work

Much research has focused on developing faster or better cryptographic protocols for SC by improving the underlying theory [9, 11, 18, 48]. Prior research has also developed efficient implementations of SC [22, 26, 53] and programming frameworks to make SC accessible to non-experts [12, 20]. Osprey aims to improve the performance of SC without introducing a new DSL/programming framework, and without requiring the SC protocols to be re-implemented or new SC protocols to be designed. Thus, Osprey is complementary to these existing lines of research.

An existing line of work focuses on making SMPC scale to large data analytics workloads by identifying portions of an analytics query that can be safely executed at a single party or a subset of parties [2, 44, 47, 57]. Another line of work performs similar analysis for workloads like cooperative machine learning—in effect, developing custom SC protocols for particular tasks of interest [24, 36, 37]. Osprey is complementary to these efforts as well—one can first apply techniques from these lines of work to minimize the size of the SC computation, and then apply Osprey to execute the resulting computation efficiently.

In one line of work, the compiler analyzes the program

and inserts prefetch and release directives in the generated code [5, 38, 39]. This approach is mostly limited to matrix computations consisting of array accesses in a loop, and does not generalize easily to cases where the computation is recursively structured or where the access patterns are in an SC library removed from the looping array accesses. Later work used software speculation to remove such assumptions about the structure of the computation [17]. However, software speculation adds additional overhead and complexity in the kernel due to the need to recover from misspeculation.

Some recent systems focus on *oblivious* workloads. One such prior work is MAGE [28], to which we have already compared Osprey extensively. Another such work is 3PO [4], which has to run the speculative pass separately to get memory accesses beforehand. Osprey's main improvement over 3PO is that it leverages content obliviousness to make the speculative pass efficient enough to be run together with the main execution. Osprey makes two additional improvements compared to 3PO: (1) whereas 3PO requires all of the program's heap accesses to be oblivious, Osprey allows program annotations to mark content-oblivious memory, and (2) whereas Osprey is implemented mostly in user space with minimally extended user/kernel interfaces, 3PO is implemented almost entirely in the kernel with significant changes to the page fault handler. Another difference is that Osprey focuses on paging to secondary storage, while 3PO focuses on prefetching from far memory. In §5 and §6, we explained how we build on 3PO's techniques and the new techniques that we contribute.

There is also a growing body of research on far memory [1, 19,34,45,49,68]. Some of these approaches rely on significant application modifications, which Osprey avoids. Others aim to work transparently, using paging. None of these systems leverage the obliviousness or content obliviousness of SC applications for memory management, as Osprey does.

11 Discussion

11.1 Misspeculation Tolerance

Our earlier claim that "CO enables us to eliminate misspeculation entirely" (§1) deserves a more precise statement. In a certain case, Osprey's speculative pass may "miss" some accesses, and Osprey tolerates this. Osprey eliminates misspeculation in the sense that the speculative pass never predicts "incorrect" accesses that the program does not actually make.

If the SC library does not have complete `OSPREY_TOUCH` annotations, then the speculative pass may miss some accesses. In this setting, the speculative pass's trace is (at least partially) collected from page faults via the eBPF handler (§5). If thread A is the first to access a page, then the eBPF handler records the access; if thread B later accesses the same page before it is swapped out, then B's access does not fault and therefore does not appear in the trace. As a result, the programmed pass may not prefetch the page in time, but the program runs correctly, with (perhaps) a small loss in performance.

Osprey does not tolerate phantom accesses, where the speculative pass predicts an access that does not actually occur in the programmed pass. Such accesses cannot arise in our setting: the speculative pass's trace is derived only from real page faults and from `OSPREY_TOUCH` calls on cryptographic operations the application will actually execute.

11.2 Speculation Cost

It is a deployment choice whether, for a multithreaded workload, to run the speculative passes for the threads on a single core or on multiple cores. The trade-off is that consolidating speculative threads onto fewer cores may cause speculation to become CPU-bottlenecked and generate predictions more slowly, which could slow down the programmed pass. As a concrete example, with full `OSPREY_TOUCH` annotations (Fig. 9), the speculative pass uses approximately 42% of the programmed pass's CPU per thread for SEAL/CKKS's matrix-vector multiplication, but under 10% for EMP-Toolkit's matrix-vector multiplication. This means that co-locating many speculative threads on a single core may be viable for EMP-Toolkit's matrix-vector multiplication.

11.3 Swap Backends

Because Osprey pages aggressively to an SSD, a natural concern is that the resulting write traffic may shorten the SSD's lifetime. This is a real cost, but two points put it in perspective. First, this is a trade-off that the industry accommodates: major cloud providers offer general-purpose VM instance families with local NVMe SSDs (e.g., AWS `m5d`, Azure `Ddsv6`), and swap space is one of the recommended uses of this local storage [35]. Second, Osprey's techniques could be generalized to swap backends beyond SSD: its paging design is agnostic to the backing store, so its techniques could be applied using backends without write-endurance limits, such as far memory or CXL-attached memory.

12 Conclusion

We presented Osprey, a system for addressing the high memory overhead of SC by very efficiently paging to secondary storage. By leveraging speculative execution, it achieves transparency—it does not require applications to be rewritten. We observe that SC has a property called *content obliviousness*, which is a different aspect of obliviousness than what prior work has explored. We leverage this property to make speculative execution simpler and more efficient.

Osprey's observations and techniques can also potentially be applied to certain non-SC workloads like neural network training/inference and oblivious database engines. This could be an interesting direction for future work.

Acknowledgments

We would like to thank David E. Culler for formative conversations in the early stages of this research project. We would also like to thank the anonymous reviewers and shepherd for their constructive feedback, and members of the systems group at UCLA for their feedback on a draft of this paper. This work was supported in part by the U.S. National Science Foundation under the grants CNS-2403254, CNS-2330831, CNS-2106838, IIS-2546642, and CNS-1943347. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

References

[1] Emmanuel Amaro, Christopher Branner-Augmon, Zhihong Luo, Amy Ousterhout, Marcos K. Aguilera, Aurojit Panda, Sylvia Ratnasamy, and Scott Shenker. Can far memory improve job throughput? In *EuroSys*. ACM, 2020.

[2] Johes Bater, Gregory Elliott, Craig Eggen, Satyender Goel, Abel Kho, and Jennie Rogers. SMCQL: Secure querying for federated databases. *VLDB*, 10(6), 2017.

[3] Emery D. Berger, Kathryn S. McKinley, Robert D. Blumofe, and Paul R. Wilson. Hoard: a scalable memory allocator for multithreaded applications. In *ASPLOS*. ACM, 2000.

[4] Christopher Branner-Augmon, Narek Galstyan, Sam Kumar, Emmanuel Amaro, Amy Ousterhout, Aurojit Panda, Sylvia Ratnasamy, and Scott Shenker. 3PO: Programmed far-memory prefetching for oblivious applications, 2022.

[5] Angela Demke Brown and Todd C. Mowry. Taming the memory hogs: Using compiler-inserted releases to manage physical memory intelligently. In *OSDI*. USENIX, 2000.

[6] Ariel N. Burton and Paul H. J. Kelly. Performance prediction of paging workloads using lightweight tracing. In *IPDPS*. IEEE Computer Society, 2003.

[7] Fay Chang and Garth A. Gibson. Automatic I/O hint generation through speculative execution. In *OSDI*. USENIX, 1999.

[8] Yong Chen, Surendra Byna, Xian-He Sun, Rajeev Thakur, and William Gropp. Hiding I/O latency with pre-execution prefetching for parallel applications. In *SC*. IEEE, 2008.

[9] Jung Hee Cheon, Andrey Kim, Miran Kim, and Yongsoo Song. Homomorphic encryption for arithmetic of approximate numbers. In *ASIACRYPT*, 2017.

[10] Ilaria Chillotti, Nicolas Gama, Mariya Georgieva, and Malika Izabachène. TFHE: Fast fully homomorphic encryption library. https://tfhe.github.io/tfhe/, 2016.

[11] Ivan Damgård, Valerio Pastro, Nigel Smart, and Sarah Zakarias. Multiparty computation from somewhat homomorphic encryption. Cryptology ePrint Archive, Report 2011/535, 2011. https://eprint.iacr.org/2011/535.

[12] Roshan Dathathri, Blagovesta Kostova, Olli Saarikivi, Wei Dai, Kim Laine, and Madan Musuvathi. EVA: An encrypted vector arithmetic language and compiler for efficient homomorphic computation. In *PLDI*. ACM, 2020.

[13] Daniel Demmler, Thomas Schneider, and Michael Zohner. ABY – a framework for efficient mixed-protocol secure two-party computation. In *NDSS*. Internet Society, 2015.

[14] Erez Eizenman. Scotiabank's chief risk officer on the state of anti-money laundering. https://www.mckinsey.com/capabilities/risk-and-resilience/our-insights/scotiabanks-chief-risk-officer-on-the-state-of-anti-money-laundering, 2019. Accessed: June 23, 2025.

[15] Jason Evans. A scalable concurrent malloc(3) implementation for FreeBSD. https://people.freebsd.org/~jasone/jemalloc/bsdcan2006/jemalloc.pdf, BSDCan, April 2006. Accessed: December 10, 2025.

[16] Fireblocks. Secure multi-party computation framework | Fireblocks. https://www.fireblocks.com/secure-multi-party-computation-framework/. Accessed: April 9, 2025.

[17] Keir Fraser and Fay Chang. Operating system I/O speculation: How two invocations are faster than one. In *ATC*. USENIX, 2003.

[18] Craig Gentry, Shai Halevi, and Nigel P. Smart. Fully homomorphic encryption with polylog overhead. In *EUROCRYPT*. Springer, Berlin, Heidelberg, 2012.

[19] Juncheng Gu, Youngmoon Lee, Yiwen Zhang, Mosharaf Chowdhury, and Kang Shin. Efficient memory disaggregation with Infiniswap. In *NSDI*. USENIX, 2017.

[20] Marcella Hastings, Brett Hemenway, Daniel Noble, and Steve Zdancewic. SoK: General purpose compilers for secure multi-party computation. In *S&P*. IEEE, 2019.

[21] Mohammad Hedayati, Spyridoula Gravani, Ethan Johnson, John Criswell, Michael L. Scott, Kai Shen, and Mike Marty. Hodor: Intra-process isolation for high-throughput data plane libraries. In *ATC*. USENIX, 2019.

[22] Yan Huang, David Evans, Jonathan Katz, and Lior Malka. Faster secure two-party computation using garbled circuits. In *USENIX Security*. USENIX, 2011.

[23] Mihaela Ion, Ben Kreuter, Ahmet Erhan Nergiz, Sarvar Patel, Mariana Raykova, Shobhit Saxena, Karn Seth, David Shanahan, and Moti Yung. On deploying secure computing: Private intersection-sum-with-cardinality protocols. In *EuroS&P*. IEEE, 2020.

[24] Chiraag Juvekar, Vinod Vaikuntanathan, and Anantha Chandrakasan. GAZELLE: A low latency framework for secure neural network inference. In *USENIX Security*. USENIX, 2018.

[25] Marcel Keller. MP-SPDZ: A versatile framework for multi-party computation. In *CCS*. ACM, 2020.

[26] Ben Kreuter, Benjamin Mood, abhi shelat, and Kevin Butler. PCF: A portable circuit format for scalable two-party secure computation. In *USENIX Security*. USENIX, 2013.

[27] Benjamin Kreuter, Abhi Shelat, and Chih hao Shen. Billion-Gate secure computation with malicious adversaries. In *USENIX Security*. USENIX, 2012.

[28] Sam Kumar, David E. Culler, and Raluca Ada Popa. MAGE: Nearly zero-cost virtual memory for secure computation. In *OSDI*. USENIX, 2021.

[29] Lattigo v6. https://github.com/tuneinsight/lattigo, August 2024. EPFL-LDS, Tune Insight SA.

[30] Tong Li, Carla S. Ellis, Alvin R. Lebeck, and Daniel J. Sorin. Pulse: A dynamic deadlock detection mechanism using speculative execution. In *ATC*. USENIX, 2005.

[31] Chang Liu, Xiao Shaun Wang, Kartik Nayak, Yan Huang, and Elaine Shi. ObliVM: A programming framework for secure computation. In *S&P*. IEEE, 2015.

[32] Michael E. Locasto and Angelos D. Keromytis. Speculative execution as an operating system service. Technical Report CUCS-024, Columbia University, 2006.

[33] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. In *PLDI*. ACM, 2005.

[34] Hasan Al Maruf and Mosharaf Chowdhury. Effectively prefetching remote memory with leap. In *ATC*. USENIX, 2020.

[35] Drew McDaniel. Virtual machines best practices: Single VMs, temporary storage and uploaded disks. https://azure.microsoft.com/en-us/blog/virtual-machines-best-practices-single-vms-temporary-storage-and-uploaded-disks/.

[36] Pratyush Mishra, Ryan Lehmkuhl, Akshayaram Srinivasan, Wenting Zheng, and Raluca Ada Popa. Delphi: A cryptographic inference service for neural networks. In *USENIX Security*. USENIX Association, 2020.

[37] Payman Mohassel and Yupeng Zhang. SecureML: A system for scalable privacy-preserving machine learning. In *S&P*. IEEE, 2017.

[38] Todd C. Mowry, Angela K. Demke, and Orran Krieger. Automatic compiler-inserted I/O prefetching for out-of-core applications. In *OSDI*. USENIX, 1996.

[39] Todd Carl Mowry. *Tolerating latency through software-controlled data prefetching*. PhD thesis, Stanford University, 1994.

[40] MPC alliance. https://www.mpcalliance.org/. Accessed: June 30, 2023.

[41] Edmund B. Nightingale, Peter M. Chen, and Jason Flinn. Speculative execution in a distributed file system. In *SOSP*. ACM, 2005.

[42] Doni Setio Pambudi, Hudan Studiawan, Baskoro Adi Pratomo, and Diana Purwitasari. Performance analysis of leading homomorphic encryption libraries: A benchmark study of SEAL, HElib, OpenFHE, and Lattigo. In *CSAIDE*. ACM, 2025.

[43] Antonis Papadimitriou, Ranjita Bhagwan, Nishanth Chandran, Ramachandran Ramjee, Andreas Haeberlen, Harmeet Singh, Abishek Modi, and Saikrishna Badrinarayanan. Big data analytics over encrypted datasets with Seabed. In *OSDI*. USENIX, 2016.

[44] Rishabh Poddar, Sukrit Kalra, Avishay Yanai, Ryan Deng, Raluca Ada Popa, and Joseph M. Hellerstein. Senate: A Maliciously-Secure MPC platform for collaborative analytics. In *USENIX Security*. USENIX, 2021.

[45] Yifan Qiao, Chenxi Wang, Zhenyuan Ruan, Adam Belay, Qingda Lu, Yiying Zhang, Miryung Kim, and Guoqing Harry Xu. Hermit: Low-latency, high-throughput, and transparent remote memory via feedback-directed asynchrony. In *NSDI*. USENIX, 2023.

[46] James Reyes. Building the next generation of digital advertising with MPC. Real World Crypto, 2022. https://youtu.be/6Gb0xO8csVU?t=2533.

[47] Jennie Rogers, Elizabeth Adetoro, Johes Bater, Talia Canter, Dong Fu, Andrew Hamilton, Amro Hassan, Ashley Martinez, Erick Michalski, Vesna Mitrovic, Fred Rachman, Raj Shah, Matt Sterling, Kyra VanDoren, Theresa L. Walunas, Xiao Wang, and Abel Kho. VaultDB: A real-world pilot of secure multi-party computation within a clinical research network, 2022.

[48] Mike Rosulek. A brief history of practical garbled circuit optimizations. https://simons.berkeley.edu/talks/mike-rosulek-2015-06-09, https://www.youtube.com/watch?v=FTxh908u9y8, 2015. Accessed: April 27, 2020.

[49] Zhenyuan Ruan, Malte Schwarzkopf, Marcos K. Aguilera, and Adam Belay. AIFM: High-performance, application-integrated far memory. In *OSDI*. USENIX, 2020.

[50] Safeheron. Safeheron MPC self-custody. https://safeheron.com/products/mpc-self-custody/. Accessed: April 9, 2025.

[51] Microsoft SEAL. https://github.com/Microsoft/SEAL, January 2023. Microsoft Research, Redmond, WA.

[52] Mahdi Soleimani, Grace Jia, and Anurag Khandelwal. Weave: Efficient and expressive oblivious analytics at scale. In *OSDI*. USENIX, 2025.

[53] Ebrahim M. Songhori, Siam U. Hussain, Ahmad-Reza Sadeghi, Thomas Schneider, and Farinaz Koushanfar. TinyGarble: Highly compressed and scalable sequential garbled circuits. In *S&P*. IEEE, 2015.

[54] Mounika Tarigopula. Understanding speculative execution. https://kb.databricks.com/scala/understanding-speculative-execution, November 2022.

[55] Anjo Vahldiek-Oberwagner, Eslam Elnikety, Nuno O. Duarte, Michael Sammler, Peter Druschel, and Deepak Garg. ERIM: Secure, efficient in-process isolation with protection keys (MPK). In *USENIX Security*. USENIX, 2019.

[56] Alexander Viand, Patrick Jattke, and Anwar Hithnawi. SoK: Fully homomorphic encryption compilers. In *S&P*. IEEE, 2021.

[57] Nikolaj Volgushev, Malte Schwarzkopf, Ben Getchell, Mayank Varia, Andrei Lapets, and Azer Bestavros. Conclave: secure multi-party computation on big data. In *EuroSys*. ACM, 2019.

[58] Amanda Walker, Sarvar Patel, and Moti Yung. Helping organizations do more without collecting more data. Google Security Blog, 2019. https://security.googleblog.com/2019/06/helping-organizations-do-more-without-collecting-more-data.html.

[59] Ke Coby Wang and Michael K. Reiter. How to end password reuse on the web. In *NDSS*. Internet Society, 2019.

[60] Xiao Wang, Alex J. Malozemoff, and Jonathan Katz. EMP-toolkit: Efficient MultiParty computation toolkit. https://github.com/emp-toolkit, 2016.

[61] Sophia Yakoubov. A gentle introduction to Yao's garbled circuits. http://web.mit.edu/sonka89/www/papers/2017ygc.pdf, 2017.

[62] Anil Yelam, Stewart Grant, Enze Liu, Radhika Niranjan Mysore, Marcos K. Aguilera, Amy Ousterhout, and Alex C. Snoeren. Limited access: The truth behind far memory. In *WORDS*. ACM, 2023.

[63] Matei Zaharia, Andy Konwinski, Anthony D. Joseph, Randy Katz, and Ion Stoica. Improving MapReduce performance in heterogeneous environments. In *OSDI*. USENIX, 2008.

[64] Samee Zahur, Mike Rosulek, and David Evans. Two halves make a whole: Reducing data transfer in garbled circuits using half gates. In *EUROCRYPT*. Springer, Berlin, Heidelberg, 2015.

[65] Wenting Zheng, Ankur Dave, Jethro G. Beekman, Raluca Ada Popa, Joseph E. Gonzalez, and Ion Stoica. Opaque: An oblivious and encrypted distributed analytics platform. In *NSDI*. USENIX, 2017.

[66] Wenting Zheng, Ryan Deng, Weikeng Chen, Raluca Ada Popa, Aurojit Panda, and Ion Stoica. Cerebro: A platform for multi-party cryptographic collaborative learning. In *USENIX Security*. USENIX, 2021.

[67] Wenting Zheng, Raluca Ada Popa, Joseph E. Gonzalez, and Ion Stoica. Helen: Maliciously secure coopetitive learning for linear models. In *S&P*. IEEE, 2019.

[68] Yang Zhou, Hassan M. G. Wassel, Sihang Liu, Jiaqi Gao, James Mickens, Minlan Yu, Chris Kennelly, Paul Turner, David E. Culler, Henry M. Levy, and Amin Vahdat. Carbink: Fault-tolerant far memory. In *ATC*. USENIX, 2022.

[69] Ruiyu Zhu, Darion Cassel, Amr Sabry, and Yan Huang. NANOPI: Extreme-scale actively-secure multi-party computation. In *CCS*. ACM, 2018.

Nested SEV: Secure and Generic SEV Support for Nested Virtualization

Kazuki Takiguchi
Kyushu Institute of Technology

Kenichi Kourai
Kyushu Institute of Technology

Abstract

In cloud environments, sensitive information in virtual machines (VMs) is exposed to insider threats. To protect VMs from malicious cloud insiders, modern clouds provide confidential VMs based on technologies such as AMD SEV, which transparently encrypts the memory of VMs and the state of CPU registers while ensuring their integrity. However, existing SEV support is not sufficient for nested virtualization, where a guest hypervisor (L1 hypervisor) runs inside a host VM (L1 VM) managed by the host hypervisor (L0 hypervisor) and creates guest VMs (L2 VMs). This paper proposes *nested SEV* to provide more secure and generic SEV support for nested virtualization. Nested SEV allows SEV-enabled L2 VMs to run inside an SEV-enabled L1 VM. It supports two trust models: (1) both the L0 and L1 hypervisors are untrusted, and (2) the L0 hypervisor is untrusted but the L1 hypervisor is trusted. For these trust models, nested SEV provides two mechanisms. *SEV virtualization* protects L2 VMs against both the L0 and L1 hypervisors. In contrast, *SEV passthrough* protects them against the L0 hypervisor but allows the L1 hypervisor to access the memory of L2 VMs by sharing the SEV context. These mechanisms rely on *emulation-less multiplexing* and *SEV context decoupling*. We implemented nested SEV in three different types of hypervisors and showed that the average performance degradation ranged from 0.9% to 30% across three SEV variants.

1 Introduction

As users deal with sensitive information in virtual machines (VMs) provided by clouds, the threat from cloud insiders is growing. To protect sensitive information in users' VMs, modern clouds such as Amazon Web Services, Google Cloud, and Microsoft Azure offer confidential VMs [8, 19, 33] based on technologies such as AMD SEV [3]. SEV transparently encrypts the memory of VMs and allows its contents to be decrypted only inside the VMs. SEV-ES [1] extends SEV by protecting the state of CPU registers, while SEV-SNP [2] further enhances protection by ensuring memory integrity. Using these technologies, even cloud insiders cannot eavesdrop on or tamper with sensitive information in users' VMs.

On the other hand, nested virtualization [10, 28] has been used for various purposes, including virtual clouds [14, 21, 30, 36, 47]. It enables a guest hypervisor (hereafter, the L1 hypervisor) to run inside a host VM (L1 VM) managed by the host hypervisor (L0 hypervisor) and to create guest VMs (L2 VMs). However, existing SEV support for nested virtualization is limited. For example, Microsoft's patch [40] enables SEV-SNP for L2 VMs but cannot protect the underlying L1 VM with SEV. As a result, the L0 hypervisor can compromise the L1 hypervisor running inside the L1 VM. Hecate [16] and OpenHCL [37] enable SEV-SNP for both L1 and L2 VMs, but they support only a single L2 VM.

This paper proposes *nested SEV* to provide more secure and generic SEV support for nested virtualization. Nested SEV allows multiple SEV-enabled L2 VMs to run inside an SEV-enabled L1 VM. It supports two trust models: (1) both the L0 and L1 hypervisors are untrusted, and (2) the L0 hypervisor is untrusted but the L1 hypervisor is trusted. For these trust models, nested SEV provides two mechanisms: *SEV virtualization* and *SEV passthrough*. *SEV virtualization* applies virtual SEV to an L2 VM and protects its memory and register state using a different SEV context from that of the L1 VM, e.g., an encryption key. As a result, it protects L2 VMs against both the L0 and L1 hypervisors. In contrast, *SEV passthrough* partially virtualizes SEV and uses the same SEV context as the L1 VM. It protects L2 VMs against the L0 hypervisor but allows the L1 hypervisor to access the internal state of L2 VMs.

These mechanisms rely on two key techniques. The first technique, *emulation-less multiplexing*, securely manages sensitive SEV contexts for L2 VMs. Rather than emulating the AMD Secure Processor (AMD-SP) in an untrusted L0 hypervisor, it multiplexes SEV contexts for both L1 and L2 VMs on the physical AMD-SP. The second technique, *SEV context decoupling*, enables an L1 VM and its L2 VMs to share a single SEV context. It decouples the SEV context from the VM and allows each L2 VM to use the same SEV context as its L1

VM. We have implemented nested SEV using KVM [26] as the L0 hypervisor and KVM, BitVisor [41], and Xen (paravirtualization) [9] as L1 hypervisors. Nested SEV supports three SEV variants and allows users to balance security and performance. We analyzed the security of nested SEV and showed that the average performance degradation ranged from 0.9% to 30% across two mechanisms, three SEV variants, and three L1 hypervisors.

In summary, this paper makes the following contributions.

- We propose nested SEV, which enables the secure execution of an L1 hypervisor and multiple L2 VMs inside an SEV-enabled L1 VM under both trusted and untrusted L1 hypervisors.
- We design and implement SEV virtualization and SEV passthrough using two novel techniques and support three SEV variants, allowing users to balance security and performance.
- We evaluate nested SEV using three different types of L1 hypervisors and quantify its overhead under various workloads.

Nested SEV is open-sourced and publicly available at https://github.com/ksl-kyutech/nested-sev.

2 Background

2.1 AMD SEV

Secure Encrypted Virtualization (SEV). SEV [3] is a security feature that enables transparent memory encryption for VMs on AMD EPYC processors. The memory controller encrypts data in a VM when it is written to physical memory and decrypts it when it is read. The AMD Secure Processor (AMD-SP) generates a distinct encryption key for each VM and manages it as part of an SEV context. Therefore, the memory of a VM cannot be eavesdropped on by the hypervisor, devices, or other VMs. In an SEV-enabled VM, a memory page is encrypted when the guest operating system (OS) sets the C-bit of the corresponding page table entry (PTE). For memory regions shared with the hypervisor, e.g., DMA bounce buffers, the guest OS clears the C-bit to mark the corresponding page as unencrypted. In this paper, we refer to this baseline version as SEV_0 to distinguish it from the generic term SEV, which collectively refers to SEV, SEV-ES, and SEV-SNP.

SEV Encrypted State (ES). SEV-ES [1] is an extension of SEV_0, introduced in second-generation EPYC processors. It additionally encrypts the state of CPU registers on VM exits. Traditionally, the Secure Virtual Machine (SVM) extension in the AMD Virtualization (AMD-V) architecture stores the register state in a memory region called the VM Control Block (VMCB) when a VM exit occurs. In contrast, SEV-ES encrypts this state using the VM-specific key and stores it in a memory region called the VM save area (VMSA). This mechanism can prevent information leakage from CPU registers. As a downside, the hypervisor cannot handle VM exits in a traditional manner because it needs access to the register state to emulate instructions.

In SEV-ES, a VMM communication exception (#VC) is raised in a VM when an event that would normally trigger a VM exit occurs. At this time, the #VC handler of the guest OS is invoked. It stores only the necessary register state for handling the event in a memory region shared with the hypervisor, called the Guest-Hypervisor Communication Block (GHCB) [6]. Then, it executes the `VMGEXIT` instruction to trigger a VM exit. The hypervisor reads the values stored in the GHCB, handles the event, and writes the results back to the GHCB. Note that several events, such as external interrupts, do not raise #VC exceptions but directly cause VM exits.

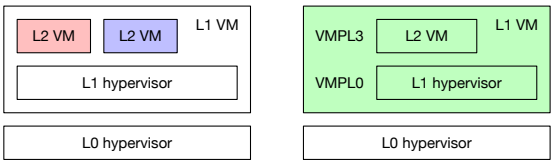
SEV Secure Nested Paging (SNP). SEV-SNP [3] is an extension introduced in third-generation EPYC processors. It prevents software-based integrity attacks such as data corruption, replay attacks, memory aliasing, and memory remapping. To enable this, it uses the Reverse Map Table (RMP), which has an entry for each host physical page and maintains ownership information for that page. A guest-owned page is a page assigned to a VM, while a hypervisor-owned page is a page that is shared with or not assigned to a VM. Since the hypervisor cannot write data to guest-owned pages, it cannot corrupt the memory of a VM or replace it with stale data.

The RMP is used to verify the page owner more rigorously. When a guest physical address (GPA) is translated into a host physical address (HPA), an RMP check fails if the address space identifier (ASID) of the current VM does not match that stored in the corresponding RMP entry. If the GPA does not match that in the RMP entry, the RMP check also fails. This prevents aliasing attacks, where the hypervisor maps a host physical page to multiple VMs. However, the hypervisor can alter the mapping by executing the `RMPUPDATE` instruction. To prevent such remapping attacks, the guest OS in a VM validates its memory pages by executing the `PVALIDATE` instruction. A page is changed to the invalid state when the hypervisor executes `RMPUPDATE`. Since a VM cannot access invalid pages, remapping attacks can be detected.

SEV-SNP also introduces VM Privilege Levels (VMPLs), which enable a VM to divide its GPA space into four levels. VMPL0 is the highest privilege level and is protected from lower VMPLs.

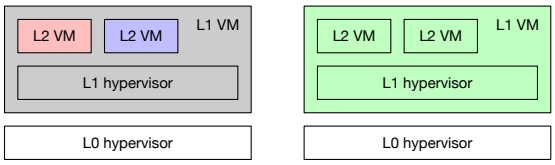
2.2 Nested Virtualization

Nested virtualization [10, 28] enables a guest hypervisor to run inside a host VM and create guest VMs on top of it. A system using nested virtualization consists of three levels. The host hypervisor runs at level 0 (L0) and is called the L0 hypervisor. A host VM managed by the L0 hypervisor runs at level 1 (L1) and is called an L1 VM. The guest hypervisor running inside an L1 VM is called the L1 hypervisor. A guest



(a) Microsoft's patch (b) Hecate

Figure 1: Current SEV support for nested virtualization. Different colors indicate different SEV contexts.



(a) Untrusted L1 hypervisor (b) Trusted L1 hypervisor

Figure 2: Two trust models in nested SEV.

VM created by the L1 hypervisor runs at level 2 (L2) and is called an L2 VM.

One application of nested virtualization is virtual clouds [14, 21, 30, 36, 47] constructed on existing clouds. Virtual cloud providers can run their custom cloud infrastructure using L1 VMs without preparing their own physical hosts. Cloud users can benefit from virtual clouds if such virtual clouds provide more attractive services than existing clouds, such as better instance types and lower prices. Since users increasingly rely on SEV to protect their VMs in untrusted public clouds, it is natural to expect SEV protection for L2 VMs in virtual clouds. Likewise, virtual cloud providers would benefit from being able to protect the L1 VMs hosting their cloud infrastructure against the underlying public clouds.

However, SEV support for nested virtualization is still limited. Microsoft's patch [40] enables SEV-SNP only for L2 VMs, as shown in Figure 1(a). Since it cannot apply SEV to the L1 VM running L2 VMs, the L1 hypervisor in the L1 VM is not protected against the L0 hypervisor. This means that the L1 hypervisor must trust the L0 hypervisor. This trust assumption is a fundamental limitation for virtual clouds running on untrusted public clouds. Even though L2 VMs are protected by SEV-SNP, they cannot be expected to operate correctly if the underlying L1 hypervisor has been compromised.

Hecate [16] and OpenHCL [37] can apply SEV-SNP not only to L2 VMs but also to L1 VMs, as shown in Figure 1(b). However, they can run only one L2 VM per L1 VM because they are implemented using VMPLs, which lack support for MMU virtualization. This limitation severely restricts the applicability of virtual clouds, which need to run many L2 VMs. In terms of security, Hecate and OpenHCL cannot protect the L2 VM from the L1 hypervisor. They divide the GPA space of the L1 VM using VMPLs and assign the portion that is not used by the L1 hypervisor running in VMPL0 to the L2 VM. Consequently, the L2 VM shares the SEV context with the L1 VM, and its memory is encrypted using the same key as the L1 VM. Although this allows the L2 VM to directly share encrypted memory with the L1 hypervisor, the L1 hypervisor can freely access the internal state of the L2 VM, which runs at a lower privilege level. Therefore, the L2 VM must trust the L1 hypervisor. Since VMPLs are used to implement nested virtualization, neither L1 nor L2 VMs can use them for other purposes, such as vTPM [35]. Table 1 summarizes the characteristics of these existing approaches.

2.3 Threat Model

The threat model of this paper is similar to that of SEV. We assume that the processors and the AMD-SP are trusted. Since the L0 hypervisor is untrusted, it can mount attacks against both L1 and L2 VMs. Unlike the original threat model of SEV, we consider both untrusted and trusted L1 hypervisors. When the L1 hypervisor is untrusted, it can also mount attacks against L2 VMs.

As attack vectors, we consider attacks against the confidentiality and integrity of the memory and register states of L1 and L2 VMs. However, we do not consider control-flow manipulation attacks that change the handling of VM exits or modify the VMCB in the hypervisors. Therefore, we exclude attacks in which the L0 hypervisor bypasses the L1 hypervisor by not redirecting VM exits to it. Also, we do not consider side-channel attacks against L1 and L2 VMs. For example, the L0 hypervisor can observe control flows, e.g., interactions between the L1 hypervisor and L2 VMs, as well as memory access patterns, to infer the internal states of L2 VMs. These attacks are inherent limitations of SEV because the hypervisor must control SEV-enabled VMs. Furthermore, these limitations are not specific to SEV but are common to most trusted execution environments (TEEs). In addition, we do not consider collusion attacks between the L0 hypervisor and L2 VMs because L2 VMs do not trust the L0 hypervisor. Denial-of-service (DoS) attacks against L1 and L2 VMs are also out of scope.

3 Nested SEV

This paper proposes *nested SEV* to provide more secure and generic SEV support for nested virtualization. Nested SEV allows multiple SEV-enabled L2 VMs to run inside an SEV-enabled L1 VM. It enables the L1 hypervisor to run in an SEV-enabled L1 VM and applies SEV to L2 VMs running on top of the L1 hypervisor. Nested SEV supports two trust models, as illustrated in Figure 2. In both models, the L0 hypervisor is untrusted. When the L1 hypervisor is also untrusted, L2 VMs need to be protected not only against the L0 hypervisor but also against the L1 hypervisor. When the L1 hypervisor is trusted, L2 VMs do not need to be protected against the

Table 1: Comparison of previous systems and our two mechanisms for nested SEV

	L1's trust	L2's trust	# of L2 VMs	VMPLs availability	SEV context	L1-L2 shared memory	available L2 GPA
Microsoft's patch [40]	L0	none	many	L2	L2 only	unencrypted	all
Hecate [16]	none	L1	one	–	L1-L2 shared	encrypted	all but VMPL0's
SEV virtualization	none	none	many	L1 / L2	L1 / L2	unencrypted	all
SEV passthrough	none	L1	many	L1	L1-L2 shared	encrypted	part (§5.3)

L1 hypervisor, but they still have to be protected against the L0 hypervisor. To achieve these two trust models, nested SEV provides two mechanisms: *SEV virtualization* and *SEV passthrough*. Table 1 compares these mechanisms with the previous systems.

SEV virtualization protects L2 VMs against an untrusted L1 hypervisor by using different SEV contexts, such as encryption keys and ASIDs, for L1 and L2 VMs. It provides virtual SEV to an L1 VM and enables the L1 hypervisor to apply virtual SEV to L2 VMs. To support virtual SEV, it virtualizes the AMD-SP and the RMP using a technique called *emulation-less multiplexing*. For security, the L0 hypervisor does not emulate these hardware components. Specifically, SEV contexts for L2 VMs remain in the physical AMD-SP rather than being managed by a virtual AMD-SP. Similarly, RMP entries for L2 VMs are maintained by the physical processors and AMD-SP rather than by virtual CPUs and AMD-SP. This is achieved by multiplexing SEV contexts for both L1 and L2 VMs on the physical AMD-SP and accommodating memory mappings for both L1 and L2 VMs in the RMP, which is protected by hardware. SEV virtualization shares some similarities with Microsoft's patch, but it additionally applies SEV to the L1 VM, thereby protecting the L1 hypervisor against the L0 hypervisor.

SEV virtualization can be used, for example, to construct virtual *public* clouds on top of untrusted public clouds. Public cloud providers run the L0 hypervisors, while virtual cloud providers build their own virtual public clouds using confidential VMs, i.e., SEV-enabled L1 VMs, as virtual hosts. Each virtual public cloud then provides users with SEV-enabled L2 VMs as confidential VMs. Similarly, such virtual public clouds can provide confidential containers [11], which are based on confidential VMs. Even without nested SEV, virtual cloud providers can run their custom cloud infrastructure on public bare-metal clouds. However, they cannot protect their cloud infrastructure, including hypervisors, because the underlying public clouds have direct access to the hardware. Although physical memory can be encrypted using AMD Secure Memory Encryption (SME) [4], SME does not provide memory integrity or attestation support.

On the other hand, *SEV passthrough* allows a trusted L1 hypervisor to access the memory and register states of L2 VMs by sharing the same SEV context between an L1 VM and its L2 VMs. To enable such a non-standard use of SEV contexts without VMPLs, it uses a technique called *SEV context decoupling*. In traditional SEV and SEV virtualization, each VM has a unique SEV context in the AMD-SP and the processors. In SEV passthrough, in contrast, the SEV context is decoupled from the VM. As a result, an L1 VM and its L2 VMs can share a single SEV context. This is achieved by partially virtualizing SEV and running SEV-enabled L2 VMs without involving the AMD-SP. Since the AMD-SP does not recognize L2 VMs, SEV passthrough also supports L2 VMs that do not rely on SVM, such as paravirtualized VMs in Xen [9] and software-based L2 VMs in PVM [21]. It is similar to Hecate and OpenHCL in that it trusts the L1 hypervisor, but it enables the L1 hypervisor to run multiple L2 VMs. Hecate and OpenHCL can also use the same SEV context between an L1 VM and its L2 VM using VMPLs provided only by SEV-SNP, but SEV passthrough does not rely on VMPLs. Therefore, it allows L1 VMs to use VMPLs for their own purposes.

SEV passthrough can be used, for example, to construct virtual *private* clouds. As with virtual public clouds, virtual cloud providers can construct their own virtual private clouds using confidential VMs, i.e., SEV-enabled L1 VMs, provided by public clouds. Since both administrators and users belong to the same organization, virtual private clouds can provide users with non-confidential VMs, i.e., SEV-disabled L2 VMs. Note that L2 VMs are still protected against the L0 hypervisor by SEV-enabled L1 VMs. Even without nested SEV, existing clouds already provide virtual private clouds, such as Amazon Virtual Private Cloud (VPC) [7]. Using such services, virtual cloud providers can build their own private networks, but they cannot run custom hypervisors or protect their infrastructure. Another application of SEV passthrough is to monitor the internal states of L2 VMs using VM introspection [15]. Since the L1 hypervisor can access the memory and register states of L2 VMs, it can obtain system information inside L2 VMs.

Nested SEV allows users to apply any of the three SEV variants to both L1 and L2 VMs because security often comes at the expense of performance. The strongest security is not always the best solution. Specifically, SEV-SNP provides the strongest security but incurs the highest performance overhead due to the check of memory integrity. In contrast, SEV_0 offers the highest performance by simply encrypting the memory of VMs, although it provides weaker security. SEV-ES lies between these two extremes. As a result, users can use the SEV variant that best matches their security and performance requirements.

4 SEV Virtualization

To run the L1 hypervisor in an SEV-enabled L1 VM, the L1 hypervisor needs to be modified to support SEV, as the L1 OS does. In addition, both the L1 and L0 hypervisors require additional SEV support to run L2 VMs in the L1 VM. Emulation-less multiplexing is used for RMP virtualization (§4.5) and AMD-SP virtualization (§4.6). Unless otherwise specified, this section focuses on SEV-SNP.

4.1 Unencrypted Memory Sharing for L2 VMs

In single-level virtualization with SEV, it is sufficient for the guest OS in a VM to mark memory as unencrypted to share it with the hypervisor, e.g., DMA bounce buffers. In nested virtualization, not only the L2 OS but also the L1 hypervisor needs to clear the C-bits of the corresponding PTEs in their page tables. This is because the L1 hypervisor cannot access memory encrypted with the key of the L2 VM. However, it is not easy for the L1 hypervisor to identify such pages because the L2 VM dynamically marks memory as unencrypted.

Therefore, the L1 hypervisor clears the C-bits of the PTEs corresponding to all the pages assigned to the L2 VM. If the L2 VM also clears the C-bit of its PTE, both the L2 VM and the L1 hypervisor can access the page without memory encryption. Note that the L1 hypervisor still cannot access the encrypted memory of the L2 VM unless the L2 OS explicitly clears the C-bits. As a result of this sharing mechanism, the L0 hypervisor can also access the memory shared between an L2 VM and the L1 hypervisor by clearing the C-bits. However, data in the shared memory is expected to be encrypted by the L2 VM because the L1 hypervisor is untrusted.

4.2 VMCB Virtualization

When the L1 hypervisor runs an L2 VM, it executes the `VMRUN` instruction. Since this instruction cannot be executed in an L1 VM, it causes a VM exit to the L0 hypervisor. Then, the L0 hypervisor executes the instruction on behalf of the L1 hypervisor. This instruction uses a VMCB, which is used to control the execution of an L2 VM and is stored in the memory of the L1 VM. This VMCB is called $VMCB_{12}$ to distinguish it from the VMCB used for the L1 VM. Since $VMCB_{12}$ is virtualized for the L1 hypervisor, the L0 hypervisor translates it into $VMCB_{02}$, also known as a *shadow VMCB*.

Since the memory of the L1 VM is encrypted, the L0 hypervisor cannot access $VMCB_{12}$ to generate the shadow VMCB. Therefore, the L1 hypervisor keeps $VMCB_{12}$ unencrypted, as illustrated in Figure 3. This does not lower the security of L2 VMs because the VMCB is inherently under the control of an untrusted hypervisor. As a result, attacks on the VMCB are outside the threat model of SEV.

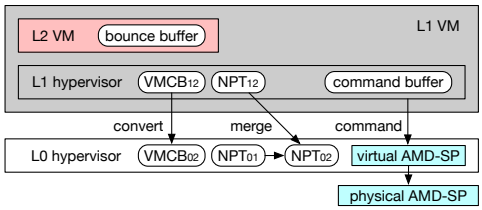


Figure 3: Unencrypted memory regions.

4.3 NPT Virtualization

In nested virtualization, the L1 hypervisor creates Nested Page Tables (NPT) for each L2 VM to translate an L2 GPA into an L1 GPA. This NPT is called NPT_{12}. Similarly, the L0 hypervisor creates NPT_{01} for each L1 VM to translate an L1 GPA into an L0 HPA. Since processors can use only one NPT during the translation of a guest virtual address (GVA) into an HPA, the L0 hypervisor merges these two NPTs and generates NPT_{02}. This NPT, also known as a *shadow NPT*, can be used to directly translate an L2 GPA into an L0 HPA.

To generate the shadow NPT, the L0 hypervisor needs to read NPT_{12} from the memory of the L1 VM, but it cannot access that NPT because that memory is encrypted. Therefore, the L1 hypervisor keeps NPT_{12} unencrypted. Consequently, the L0 hypervisor can modify the NPT_{12}, but this does not introduce additional security risks because the NPT is inherently managed by an untrusted hypervisor.

In general, the shadow NPT can be implemented using a synchronous or asynchronous method. However, the synchronous method cannot be used for SEV-enabled L1 VMs. Traditionally, the L0 hypervisor write-protects NPT_{12} in the L1 VM and configures the VMCB for the L1 VM so that a VM exit occurs when the L1 hypervisor attempts to update NPT_{12}. When a VM exit occurs, the L0 hypervisor emulates the instruction used to update NPT_{12} and modifies both NPT_{12} and its shadow NPT, i.e., NPT_{02}. To emulate the instruction, a #VC exception must be triggered by setting a reserved bit in NPT_{01} so that a reserved-bit error occurs on access. However, this significantly degrades the performance of NPT access because the #VC exception occurs even for read access.

Therefore, SEV virtualization uses the asynchronous method. This method also write-protects NPT_{12}, as in the synchronous method, but it does not emulate the instruction on a VM exit. Instead, the L0 hypervisor removes write protection from the PTE to allow it to be updated and makes the L1 hypervisor resume execution at that instruction. As a result, NPT_{12} is updated, while its shadow NPT is not. Although NPT_{12} and the shadow NPT are temporarily out of sync, they are synchronized later upon a TLB flush, which is always performed by the L1 hypervisor after NPT updates. At that time, the L0 hypervisor updates the shadow NPT to synchronize it with NPT_{12} and write-protects the PTE of NPT_{12} again.

4.4 MMIO Virtualization

To virtualize memory-mapped I/O (MMIO) in an L2 VM, the L1 hypervisor writes invalid values to the PTEs corresponding to MMIO regions in NPT_{12}. As a result, a VM exit is caused to the L0 hypervisor when the L2 VM accesses the regions. To redirect it to the L1 hypervisor, the L0 hypervisor invokes the L1 VM by executing the `VMRUN` instruction. To emulate the instruction for accessing the MMIO region, the L1 hypervisor needs to decode the instruction stored in the memory of the L2 VM.

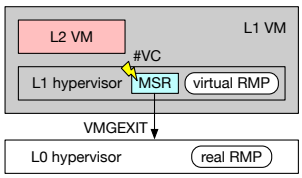
Unlike the VMCB and the NPT, however, the L2 VM cannot mark the code area as unencrypted because SEV requires the code area to remain encrypted. Therefore, the L1 hypervisor sets a reserved bit in the corresponding PTEs of NPT_{12} so that a #VC exception is raised when the L2 VM accesses the MMIO region. To virtualize the reserved bit, the L0 hypervisor synchronizes the reserved bit in the shadow NPT with that in NPT_{12}. In the L2 VM, the #VC handler decodes the instruction and stores its bytes in the GHCB. In addition, it stores the L2 GPA of the accessed MMIO region in the GHCB. Then, it executes the `VMGEXIT` instruction to cause a VM exit to the L1 hypervisor via the L0 hypervisor.

When SEV_0 is applied to an L2 VM, the L1 hypervisor cannot obtain the instruction used to access the MMIO region through a #VC exception. Therefore, the L1 hypervisor uses the Decode Assists feature in SVM, which stores the bytes of the instruction that causes a VM exit in the VMCB. Since the instruction bytes are stored in the shadow VMCB in nested virtualization, the L0 hypervisor synchronizes the corresponding fields in $VMCB_{12}$ with those in the shadow VMCB to virtualize Decode Assists.

Using PCI passthrough, an L2 VM can directly access the virtual devices provided by the L0 hypervisor rather than those provided by the L1 hypervisor [29]. To enable MMIO accesses to the L0 virtual devices, the L1 hypervisor maps the MMIO regions of the L0 virtual devices into the GPA space of the L2 VM. When the L2 VM accesses one of these MMIO regions, a VM exit occurs. To emulate the MMIO access, the L0 hypervisor needs the corresponding L1 GPA, but the GHCB or the shadow VMCB contains only the accessed L2 GPA. Therefore, the L0 hypervisor translates the L2 GPA into an L1 GPA using NPT_{12}, which is kept unencrypted by the L1 hypervisor.

4.5 RMP Virtualization

The `RMPUPDATE` instruction for updating an RMP entry can be executed only by the L0 hypervisor. To virtualize this instruction, it is recommended to use the `VIRT_RMPUPDATE` model-specific register (MSR) [5]. When the L1 hypervisor writes data to this MSR, a #VC exception is raised, as illustrated in Figure 4(a). After the #VC handler executes the `VMGEXIT` instruction, the L0 hypervisor translates the specified virtual ASID, which is assigned to the L2 VM by the L1 hypervisor, into the real ASID. It also translates the specified L1 GPA into an L0 HPA using NPT_{01}. Then, it executes `RMPUPDATE` with the real ASID, the L0 HPA, and the specified L2 GPA. As such, the L0 hypervisor does not emulate the RMP. Instead, it leverages the existing RMP by translating L1 virtual mappings into their corresponding L0 mappings.



(a) RMP for L2 VMs

HPA	Assigned	ASID	GPA
:			
0x1000	0	0	0
:			
0x3000	1	L1's	L1's
:			
0x5000	1	L2's real	L2's
:			

(b) Real RMP

Figure 4: Virtualizing the RMP.

Since the L1 hypervisor needs to read the RMP directly from memory, it creates its own virtual RMP. The virtual RMP is similar to the real RMP but maintains the virtual mapping from an L1 GPA to its owner. For a guest-owned page, the L1 hypervisor sets the Assigned field and stores the virtual ASID and GPA of an L2 VM in the RMP entry. For a hypervisor-owned page, it clears the Assigned field. The L1 hypervisor updates the virtual RMP when writing data to the `VIRT_RMPUPDATE` MSR and synchronizes it with the real RMP, which is managed by the L0 hypervisor. Since the virtual RMP is just a cache, tampering with it does not affect the security of L2 VMs.

To virtualize the RMP, it is not sufficient to manage only the traditional two types of owners. Since a guest-owned page can be assigned to either an L1 or L2 VM, it is necessary to manage three types of owners. An L2-owned page is a guest-owned page accessible only by an L2 VM. An L1-owned page is a guest-owned page that the L1 hypervisor does not assign to any L2 VMs in the L1 VM. An L0-owned page is a hypervisor-owned page that is either shared between the L0 and L1 hypervisors, shared between the L1 hypervisor and an L2 VM, or not assigned to any L1 VMs. Note that a shared page between the L1 hypervisor and an L2 VM is not an L1-owned page because it must be accessible to both the L1 and L2 VMs.

To accommodate these three types of owners in the real RMP, the L0 hypervisor constructs the real RMP as shown in Figure 4(b). It sets the Assigned field of the RMP entry for an L2-owned page and stores the real ASID and GPA of an L2 VM in the entry. For an L1-owned page, it also sets the Assigned field and stores the ASID and GPA of an L1 VM. For an L0-owned page, it clears the Assigned field. When the L1 hypervisor changes a page from L0-owned or L1-owned to L2-owned, it writes the virtual ASID of an L2 VM to the `VIRT_RMPUPDATE` MSR. When changing a page to L1-owned, it writes -1, which represents the ASID of the L1 VM itself, to

the MSR. In this case, the L0 hypervisor executes RMPUPDATE with the ASID of the L1 VM.

4.6 AMD-SP Virtualization

To boot an SEV-enabled VM, the hypervisor issues various commands for guest management to the AMD-SP. For example, it issues the SNP_GCTX_CREATE command to generate a memory encryption key and store it in the guest context page. Since the guest context page is protected by memory encryption and the RMP, the hypervisor cannot eavesdrop on or tamper with it. Then, it issues the SNP_ACTIVATE command to install the key into the memory controller and bind the specified ASID to the VM. Next, the hypervisor issues the SNP_LAUNCH_UPDATE command to encrypt the specified memory regions in the VM, such as firmware, and the VMSAs using the key. In addition, the AMD-SP marks the pages as guest-owned and validated by updating the RMP.

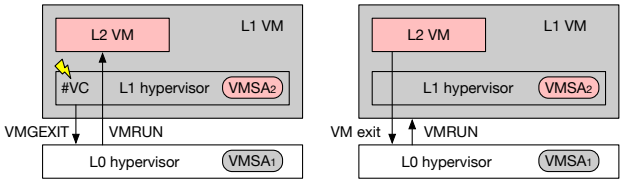
To boot an SEV-enabled L2 VM, the L0 hypervisor provides a *virtual AMD-SP* to an L1 VM. The L1 hypervisor issues a set of commands to the virtual AMD-SP. However, it would not be secure for the virtual AMD-SP itself to manage sensitive SEV contexts, e.g., encryption keys. Therefore, the L0 hypervisor does not emulate the AMD-SP. SEV contexts are securely managed only inside the physical AMD-SP, as in single-level virtualization. The guest context page is also protected against both the L0 and L1 hypervisors. When the virtual AMD-SP receives a command, it simply forwards that command to the physical AMD-SP. To allow both the virtual and physical AMD-SPs to access the command buffer, the L1 hypervisor keeps it unencrypted. If necessary, the virtual AMD-SP translates several parameters in the command buffer, e.g., the virtual ASID and the L1 GPA.

The AMD-SP provides a different set of commands for SEV_0 and SEV-ES. The command set is similar to that of SEV-SNP, but it does not use the guest context page. Sensitive SEV contexts are stored only in the AMD-SP, instead of the guest context pages. Since the virtual AMD-SP does not store any sensitive data for SEV_0 and SEV-ES, it does not introduce additional security risks.

4.7 Direct Context Switching with VMSAs

In single-level virtualization, when the hypervisor executes the VMRUN instruction to run an SEV-enabled VM, the processor saves the minimal state, such as the instruction pointer, to the host save area. Then, it loads the entire register state from the encrypted VMSA for the VM. In contrast, when a VM exit occurs, the processor encrypts the entire register state and saves it to the VMSA. Then, it loads the minimal state from the host save area.

In nested virtualization, when the L1 hypervisor executes VMRUN to run an SEV-enabled L2 VM, a #VC exception occurs, and the #VC handler invokes the L0 hypervisor using the



(a) VMRUN (b) VM exit

Figure 5: Context switching between the L1 hypervisor and an L2 VM.

VMGEXIT instruction, as illustrated in Figure 5(a). To emulate VMRUN, the L0 hypervisor needs to save the state of the L1 VM to the host save area in the L1 VM. When a VM exit occurs in the L2 VM, the L0 hypervisor needs to load the state of the L1 VM from the host save area. However, it cannot access the register state or the host save area of the L1 VM because the memory and register state of the L1 VM are encrypted.

Therefore, the L0 hypervisor simply executes VMRUN to load the VMSA for the L2 VM. It does not explicitly save the state of the L1 VM because the state is implicitly saved to the VMSA by the processor on the VM exit caused by VMGEXIT. Similarly, when a VM exit occurs in an L2 VM, the L0 hypervisor is invoked and simply executes VMRUN to load the VMSA for the L1 VM, as in Figure 5(b). It does not explicitly load the state of the L1 VM from the host save area.

For optimization, the L1 hypervisor directly configures the GHCB and executes VMGEXIT, instead of executing VMRUN. It is wasteful to cause a #VC exception by executing that instruction. This optimization can eliminate the overhead of the #VC exception.

5 SEV Passthrough

In addition to SEV support for the L1 hypervisor, SEV passthrough requires mechanisms for RMP virtualization and direct context switches, as well as SEV support for VMCB virtualization and NPT virtualization. However, it does not require the remaining mechanisms used by SEV virtualization, namely unencrypted memory sharing, SEV support for MMIO virtualization, ASID virtualization, and AMD-SP virtualization.

5.1 SEV Context Decoupling

Unlike SEV virtualization, SEV passthrough does not virtualize the AMD-SP to enable L2 VMs to share the same SEV context with their L1 VM. The L1 hypervisor does not issue commands to the virtual AMD-SP to boot L2 VMs. Also, it does not assign virtual ASIDs to L2 VMs. Instead, it assigns the ASID of the L1 VM to all the L2 VMs running in the L1 VM, so that the L2 VMs can use the same encryption key as the L1 VM. This assignment is possible because SEV

passthrough boots L2 VMs without involving the AMD-SP. Otherwise, the AMD-SP would bind a unique ASID to each L2 VM. In addition, not only the L2 OS but also the L1 hypervisor can validate the memory of the L2 VM. This does not introduce a security concern because the L1 hypervisor is trusted.

5.2 Encrypted Memory Sharing for L2 VMs

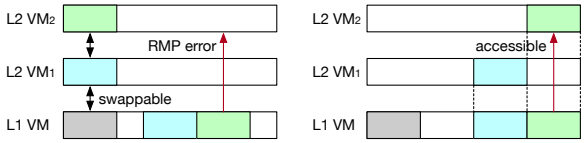
SEV passthrough enables an L2 VM to share encrypted memory with the L1 hypervisor. In SEV virtualization, shared memory needs to be kept unencrypted because the encryption keys differ between L1 and L2 VMs. Since the same ASID is assigned to both L1 and L2 VMs in SEV passthrough, the L1 hypervisor can directly access the encrypted memory of the L2 VM by using the encryption key associated with that ASID. Therefore, both the L2 OS and the L1 hypervisor can leave the corresponding C-bits set in their page tables. Consequently, the L0 hypervisor cannot eavesdrop on the shared memory between the L1 hypervisor and the L2 VM because the memory remains encrypted.

Furthermore, this can improve the I/O performance of the L2 VM. The L2 VM can perform DMA using encrypted buffers without copying data to and from unencrypted bounce buffers. In addition, the L1 hypervisor can decode instructions for MMIO virtualization by directly accessing the encrypted memory of the L2 VM. It can access the encrypted VMSAs of the L2 VM, which are stored in the memory of the L1 VM. Since a #VC exception is not necessary, the L1 hypervisor does not set the reserved bit in the PTEs of NPT_{12} corresponding to MMIO regions. This prevents a #VC exception from being raised. Instead, the L1 hypervisor writes invalid values to the PTEs. When the L2 VM accesses the MMIO region, a VM exit occurs directly to the L0 hypervisor. Then, the invoked L1 hypervisor emulates the instruction for accessing the region using the register values saved in the VMSA.

5.3 Exclusive GPA Assignment

In SEV-SNP, encrypted memory sharing between L1 and L2 VMs conflicts with RMP checks. When the L1 hypervisor accesses the memory of an L2 VM, the processor performs an RMP check. Although the ASID of the L1 VM matches that of the L2 VM, the L2 GPA registered in the entry of the real RMP generally differs from the corresponding L1 GPA. Therefore, the RMP check fails, as illustrated in Figure 6(a). As a result, the L1 hypervisor cannot access the memory of the L2 VM. Conversely, if different memory pages are assigned the same L1 or L2 GPA, the L0 hypervisor can swap those pages between L1 and L2 VMs or between L2 VMs. In this case, the RMP check still passes, leading to incomplete integrity protection.

To address these issues, the L1 hypervisor exclusively assigns memory pages to the GPA spaces of L2 VMs so that



(a) Normal assignment (b) Exclusive assignment

Figure 6: Memory assignment in the GPA spaces of an L1 VM and its L2 VMs.

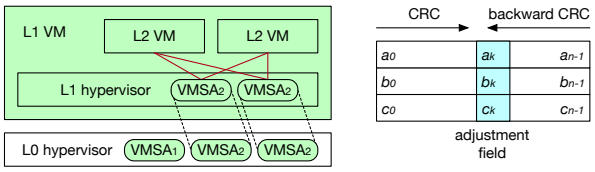
the L2 GPA of each page is identical to its L1 GPA, as illustrated in Figure 6(b). Since the L1 GPA matches the L2 GPA registered in the RMP entry, the RMP check passes when the L1 hypervisor accesses the memory of the L2 VM. In this assignment, GPAs are unique across all L2 VMs. Therefore, the RMP check fails even if the L0 hypervisor swaps memory pages because the GPAs assigned to these pages do not match those in the RMP entries. To simplify this memory assignment, the L1 hypervisor uses contiguous physical memory regions that are as large as possible for each L2 VM. Specifically, the L0 hypervisor allocates 1-GB huge pages for an L1 VM and assigns them to the GPA range above 4 GB in the L1 VM.

To boot an L2 VM with arbitrary GPAs, SEV passthrough uses custom firmware for the L2 VM. Since the x86 architecture starts the execution of firmware at a fixed physical address, an L2 VM whose memory is allocated only in the GPA space above 4 GB cannot run existing firmware such as OVMF. The custom firmware is based on qboot [38], which is designed for booting the Linux kernel and supports execution at arbitrary GPAs. It starts execution from a specified address and loads the ACPI table. To support GPAs above 4 GB, the ACPI table uses the 64-bit extension.

To initialize application processors (APs) with arbitrary GPAs, SEV passthrough uses the Wakeup Mailbox added in ACPI 6.4 [45]. Traditionally, APs are initialized in real mode using the INIT-SIPI mechanism, which requires low memory below 1 MB. This mechanism cannot be used when the kernel is placed in higher GPA regions. With the Wakeup Mailbox, APs start execution in long mode when the kernel writes the Wakeup command and the wakeup address to a dedicated memory region. In addition, the L1 hypervisor adds information on the Wakeup Mailbox to the ACPI table and provides the corresponding MMIO region.

5.4 Paravirtualized L2 VMs

SEV passthrough can support paravirtualized L2 VMs, which are created without using SVM. Since such L2 VMs are not recognized as VMs by processors, they share the same SEV context as the L1 VM. Therefore, SEV seamlessly encrypts their memory as part of the memory of the L1 VM. Note that the L2 OS can still control memory encryption by setting the C-bits in its page tables. For paravirtualized L2 VMs,



(a) VMSA pooling (b) VMSA adjustment

Figure 7: VMSA pooling and backward CRC computation.

the L1 hypervisor does not use NPT and directly assigns L1 GPAs, e.g., via pseudo-physical addresses in Xen [9], which naturally achieves exclusive GPA assignment.

When the L2 OS validates the memory of the L2 VM, it cannot execute the `PVALIDATE` instruction because this instruction is privileged. Its execution causes a general-protection exception (#GP), and the #GP handler in the L1 hypervisor executes that instruction on behalf of the L2 VM. Similarly, the L2 VM cannot use the `VMGEXIT` instruction to change the page owner because the L0 hypervisor cannot distinguish whether a VM exit originates from the L1 or L2 VM. Therefore, the L2 VM issues a new hypervisor call for changing the page owner to the L1 hypervisor.

5.5 VMSA Pooling for SEV-ES

Since SEV passthrough does not provide a virtual AMD-SP to an L1 VM, the L1 hypervisor cannot directly issue the `LAUNCH_UPDATE_VMSA` command to encrypt newly allocated VMSAs when booting ES-enabled L2 VMs. Therefore, the L0 hypervisor pre-allocates VMSAs for L2 VMs and issues the command to the physical AMD-SP when booting an L1 VM. The number of prepared VMSAs equals that of the virtual CPUs assigned to the L1 VM, but it can be smaller than the total number of virtual CPUs assigned to all L2 VMs. To support an arbitrary number of virtual CPUs, the L1 hypervisor does not statically assign the pre-allocated VMSAs to the virtual CPUs of L2 VMs. Instead, it pools VMSAs and dynamically assigns them to active virtual CPUs, as illustrated in Figure 7(a).

Since SEV passthrough shares the same SEV context between the L1 VM and its L2 VMs, the L1 hypervisor can directly initialize the register states in encrypted VMSAs when booting an L2 VM. When switching virtual CPUs, it can save the register state stored in the VMSA for the previously running virtual CPU to its memory and then load the saved register state of the next virtual CPU into the same VMSA. However, SEV-ES computes a checksum of a VMSA on each VM exit from an L2 VM to prevent rollback attacks. The checksum is securely stored in the Trusted Memory Region (TMR), which is inaccessible to software. If the L1 hypervisor modifies a VMSA, its checksum becomes inconsistent with the value stored in the TMR. This causes the failure of the `VMRUN` instruction executed by the L1 hypervisor.

To address this issue, the L1 hypervisor preserves the checksum by adjusting fields in a VMSA. The L1 hypervisor computes the checksum of a VMSA before modifying it. After modification, the L1 hypervisor computes the adjustment required to restore the original checksum. The checksum algorithm in SEV-ES is undocumented, but we found that EPYC processors compute three CRC-32C values for each VMSA. To adjust these values, the L1 hypervisor uses three 32-bit fields that are only used when a VM exit occurs. These fields are unused when the L1 hypervisor executes `VMRUN`. Note that the L0 hypervisor cannot modify VMSAs or adjust their checksums using the same technique because it cannot access encrypted VMSAs.

Specifically, the L1 hypervisor computes the adjustment value using the backward CRC algorithm [42], as in Figure 7(b). For the k-bit data before the field a_k used for the adjustment, it efficiently computes the CRC of $a_0 \cdots a_{k-1}$ using the `CRC32` instruction with the standard polynomial $P(x)$. For the $(n-k)$-bit data after that field, it must compute the CRC of $a_{n-1} \cdots a_k$ using a non-standard polynomial $x^{32} \cdot P(-x)$. To efficiently compute this value, the L1 hypervisor uses Barrett reduction with the `PCLMULQDQ` instruction [24], which computes the CRC using multiplication over 128-bit SSE registers. It further accelerates this computation using the `VPCLMULQDQ` instruction with 256-bit AVX registers.

VMSA pooling and the backward CRC computation are necessary only for SEV-ES. In SEV-SNP, the L1 hypervisor can initialize VMSAs when booting an L2 VM. It can dynamically allocate a new VMSA by updating the VMSA field in the RMP entry using the `RMPADJUST` instruction. Since VMSAs are protected from the L0 hypervisor by the RMP, their checksums are no longer stored in the TMR.

6 Security Analysis

In SEV virtualization, the L0 hypervisor can also access the memory shared between L2 VMs and the L1 hypervisor because such shared memory is kept unencrypted. However, L2 VMs can protect I/O data stored in shared memory against not only the L1 hypervisor but also the L0 hypervisor by using full disk encryption and end-to-end encryption. Even for data that is difficult to encrypt, the L0 hypervisor can mount only the same attacks as the L1 hypervisor. Note that it cannot access this shared memory in SEV passthrough because shared memory is encrypted.

Even in single-level virtualization, the hypervisor can alter the VMCB and the NPT used for VMs. In nested SEV, not only the L1 hypervisor but also the L0 hypervisor can alter unencrypted VMCB_{12} and NPT_{12} used for L2 VMs. More directly, the L0 hypervisor can alter the shadow VMCB and the shadow NPT. However, it can mount only the same class of attacks against L2 VMs as in single-level virtualization. Tampering with the NPT can be detected by RMP checks

in SEV-SNP. In addition, such attacks do not directly affect the security of the L1 hypervisor because the VMCB and the NPT control only the behavior of L2 VMs. As another attack, the L1 hypervisor can keep the NPT_{12} out of sync with the shadow NPT by not flushing the TLB. This attack is equivalent to the case where the L0 hypervisor does not update the shadow NPT as expected. Therefore, both the L0 and L1 hypervisors can mount only the same class of attacks against L2 VMs as in single-level virtualization.

The L0 hypervisor can compromise the virtual AMD-SP, but no sensitive data is stored in it. SEV contexts for L2 VMs are stored only in the physical AMD-SP, which is protected against the L0 hypervisor. Since the L1 hypervisor needs to pass an unencrypted command buffer to the virtual AMD-SP, the L0 hypervisor can tamper with this buffer. This type of attack can be detected by command-specific countermeasures, e.g., remote attestation. Similarly, the L0 hypervisor can alter the initial code, data, and register state of an L2 VM. This is because the L1 hypervisor needs to prepare them in the memory regions that are not yet protected and issue the `SNP_LAUNCH_UPDATE` command to the virtual AMD-SP. However, the owner of the L2 VM can verify that these memory regions have not been tampered with by the L0 and L1 hypervisors using remote attestation.

Hardware-based attestation of L2 VMs is not supported in SEV passthrough because L2 VMs are not directly managed by the AMD-SP. However, the correct boot of the L2 VM is ensured under the assumption that the L1 hypervisor is trusted. The memory of the L1 hypervisor and the L2 VM is protected against the L0 hypervisor. If necessary, software-based attestation using vTPM can be used.

The L0 hypervisor could bypass the L1 hypervisor if it handles all VM exits from L2 VMs because VM exits are first delivered to the L0 hypervisor. While this attack is relatively difficult in SEV passthrough, it is easier in SEV virtualization. This is because SEV virtualization does not encrypt not only the VMCB and the NPT for L2 VMs but also the GHCB and the shared memory between L2 VMs and the L1 hypervisor. Our threat model excludes such control-flow manipulation attacks as in that of SEV. A possible mitigation is that the L2 VM encrypts the GHCB and the shared memory by itself using the key shared only with the L1 hypervisor. Since the L1 hypervisor should not collude with an untrusted L0 hypervisor, the L0 hypervisor cannot obtain the necessary information to handle VM exits.

Since L2 VMs are not fully hidden from the L0 hypervisor, they could suffer from some of the side-channel attacks that the hypervisor can mount against SEV-enabled VMs in single-level virtualization. For example, the L0 hypervisor could infer the internal states of L2 VMs by observing VM exits. Using SEV passthrough and paravirtualized L2 VMs, nested SEV can hide most interactions between the L1 hypervisor and L2 VMs because L2 VMs are not mediated by the L0 hypervisor. If fully virtualized L2 VMs are used, hardware support is needed to protect such interactions. For example, TD partitioning in Intel TDX [25] enables nested virtualization similar to Hecate. Unlike Hecate, the L0 hypervisor cannot observe VM exits caused by L2 VMs because control is transferred via the trusted TDX module to the L1 hypervisor. Note that side-channel attacks are outside the scope of SEV.

The L0 hypervisor can also observe the VM exits of L1 VMs. Unlike traditional VMs and L2 VMs, the L1 VM runs a hypervisor and causes different types of VM exits, for example, by executing the `VMRUN` instruction. Such VM exits may introduce a new attack surface for L1 VMs. This is an inherent limitation of SEV because the L0 hypervisor must control the L1 VM that runs the L1 hypervisor.

In SEV passthrough, the L0 hypervisor can abuse the NPT for an L1 VM as the shadow NPT for its L2 VM. Since the L2 GPA is identical to the L1 GPA due to exclusive GPA assignment, and the ASID is the same, this shadow NPT can be used without RMP violations. As a result, the L2 VM can potentially access the entire memory of the L1 VM, including the memory used by other L2 VMs. However, this attack requires the collusion between the L0 hypervisor and the L2 VM. Since the L2 VM does not trust the L0 hypervisor, such an attack falls outside the scope of our threat model. A possible solution is to verify the GPA range used by an L2 VM. When the L2 OS sets the page table base register, the L1 hypervisor can inspect the page tables and write-protect them to monitor subsequent updates.

7 Experiments

We conducted several experiments to examine the performance of L2 VMs in nested SEV. In these experiments, we used two mechanisms in nested SEV and three SEV variants. We ran modified Linux/KVM 6.11.0 with QEMU 9.1.0 as the L0 hypervisor. In an L1 VM, we ran either modified Linux/KVM 6.11.0 with QEMU 9.1.0, BitVisor, or Xen 4.16 as the L1 hypervisor. KVM and BitVisor support full virtualization, while Xen supports paravirtualization without SVM. BitVisor adopts a passthrough-based architecture and supports only a single VM. In an L2 VM, we ran either modified OVMF from edk2-stable201903 or custom firmware as UEFI BIOS, and Linux 6.11.0 as the guest OS. For Xen, we used paravirtualized Domain 0 and Domain U. In the experiment examining the impact of VMPLs, we used the COCONUT Secure VM Service Module (SVSM) [43] in both the L1 and L2 VMs. For comparison, we also examined the performance of SEV-disabled L2 VMs and that of L1 VMs in single-level virtualization using KVM.

We used a server with a fourth-generation AMD EPYC 9334 processor, 128 GB of DDR5-4800 RDIMM memory, a 1.6 TB SAS SSD, and a 10 GbE NIC. We assigned resources to the L1 and L2 VMs as summarized in Table 2. The VMs used virtio-net as the network interface, except for Xen's Do-

Table 2: Resource allocation for VMs

	KVM		BitVisor		Xen		single
	L1	L2	L1	L2	L1	L2	virt
virtual CPU	12	6	6	6	12	6	6
memory (GB)	16	8	8	8	16	8	8

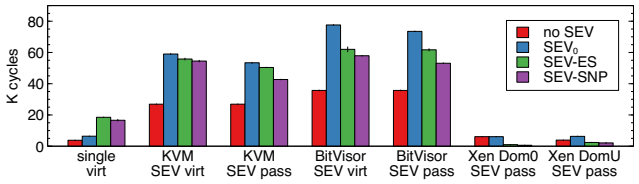


Figure 8: Overhead of VM exits.

main U, which used the xen-netfront driver. As a remote host, we used a server equipped with two Intel Xeon Silver 4110 processors, 256 GB of memory, and a 10 GbE NIC and ran Linux 5.4.189.

7.1 Overhead of VM Exits

To examine the overhead of a VM exit, we measured the number of CPU clock cycles required to execute the `VMMCALL` instruction. Figure 8 shows the results. Even in single-level virtualization, SEV_0 degraded the performance of a VM exit by 41%. SEV-ES was 2.9x slower than SEV_0 due to the #VC exception and the encryption of the register state. SEV-SNP was 12% faster than SEV-ES because CRC computation for VMSAs was not performed.

In nested virtualization, the performance of a VM exit significantly degraded. A VM exit in an L2 VM first invoked the L0 hypervisor, which executed the `VMRUN` instruction for the L1 VM. The L1 hypervisor then handled `VMMCALL` and executed `VMRUN` for the L2 VM. This instruction caused another VM exit to the L0 hypervisor, which executed `VMRUN` again to resume the L2 VM. As a result, SEV_0 virtualization was twice as slow as nested virtualization without SEV. Surprisingly, SEV-ES virtualization was faster than SEV_0 virtualization. This is because the L0 hypervisor could simply switch VMSAs between the L1 and L2 VMs during context switches. SEV-SNP virtualization was also faster than SEV-ES virtualization because the backward CRC computation was not required. Consequently, the overhead of a VM exit decreased in the order of SEV_0, SEV-ES, and SEV-SNP virtualization.

SEV passthrough was 0.5–28% faster than SEV virtualization because SEV is partially virtualized. In Xen, the #VC handler in the L1 hypervisor was invoked in SEV-ES and SEV-SNP passthrough because the L2 VMs were paravirtualized. Therefore, the #VC handler did not need to execute `VMGEXIT` to handle `VMMCALL`.

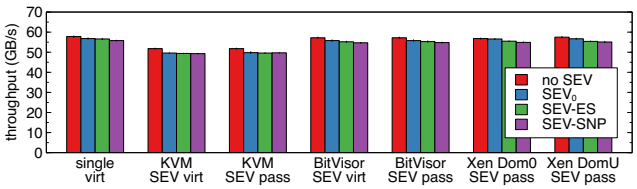


Figure 9: Performance of a memory-intensive workload.

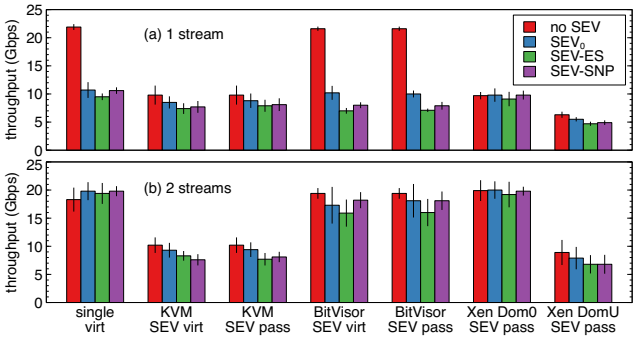


Figure 10: TCP/IP performance.

7.2 Memory Performance

We measured the performance of a memory-intensive workload using the STREAM 1.3.4 benchmark [31]. We performed a 2-GB memory copy so that the copy size was sufficiently larger than the CPU cache size. As shown in Figure 9, the overhead of nested virtualization was 0.4–3.6% in BitVisor and Xen, regardless of whether SEV was enabled. The performance decreased in the order of SEV disabled, SEV_0, SEV-ES, and SEV-SNP. In contrast, the overhead of nested virtualization was 10–12% in KVM. Since it was 10% even for an SEV-disabled L2 VM, this performance degradation is attributable to KVM's implementation rather than nested SEV.

7.3 Network Performance

We measured TCP/IP performance using the iperf 3.9 benchmark [12]. We ran the client in an L2 VM and connected it to the server running on the host. As shown in Figure 10(a), the throughput decreased by 51–53% due to SEV even in single-level virtualization. This is because the VM caused many VM exits and copied data between encrypted buffers and unencrypted bounce buffers to allow the hypervisor to access the data [27].

In nested virtualization with KVM, the throughput degraded significantly, even for an SEV-disabled L2 VM, due to the high overhead of VM exits. Enabling SEV further degraded performance, but the differences from the SEV-disabled L2 VM were relatively small. For BitVisor, the results were similar to those in single-level virtualization. The reason is that the L2 VM directly accessed the virtio-net device provided by the L0 hypervisor via PCI passthrough.

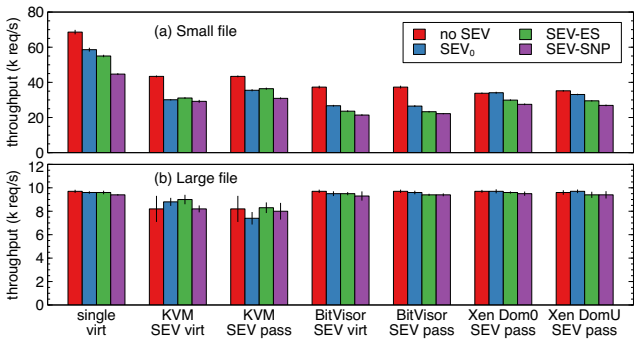


Figure 11: Performance of the Web server.

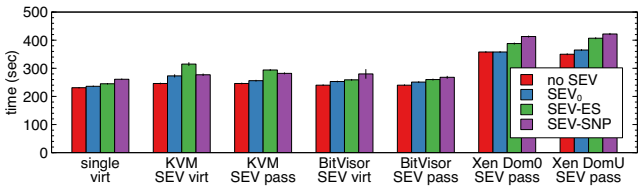


Figure 12: Build time of the Linux kernel.

Xen's Domain 0 also used PCI passthrough, but the performance of the SEV-disabled L2 VM was much lower. The throughput of Xen's Domain U was the lowest because of the split driver model.

Next, we ran two pairs of iperf3 clients and servers concurrently and measured their aggregate throughput. Figure 10(b) shows the total throughput of the two streams. The performance of the SEV-enabled VMs improved significantly in single-level virtualization, BitVisor, and Xen's Domain 0. Compared with single-level virtualization, the throughput decreased by at most 18% in BitVisor and 1.0% in Xen's Domain 0. This is because the overhead introduced by SEV was hidden by concurrent data transfers. In KVM, however, the performance improved only marginally. The throughput increased by only 12% in SEV-ES virtualization but decreased by 2.5% in SEV-ES passthrough. Although the throughput of Xen's Domain U improved by 39–45%, it remained lower than that of KVM.

Surprisingly, the performance of SEV-SNP was higher than that of SEV-ES in nested SEV. One reason is that VM exits are faster in SEV-SNP, as shown in Section 7.1. Since many VM exits occur under I/O-intensive workloads, SEV-ES suffers more from the higher overhead of VM exits. Another possible reason is the immaturity of the implementation of nested SEV. For example, SEV-ES virtualization uses our custom memory manager for L2 VMs, while SEV-SNP virtualization uses the standard `guest_memfd` provided by KVM. This could contribute to the relatively better performance of SEV-SNP virtualization with KVM.

7.4 Performance of the Web Server

We measured the performance of Apache HTTP Server 2.4.62 [44] using the bombardier 1.2.6 benchmark [13]. First, we repeatedly requested a small 45-byte HTML file from the remote host. Figure 11(a) shows the request processing rate with 200 concurrent connections. Even in single-level virtualization, SEV-SNP degraded performance by 35%. The overhead of nested virtualization was 31–58% with or without SEV. Unlike the iperf3 benchmark, the performance in BitVisor degraded significantly. This is because the transferred data was small, making the processing overhead relatively more significant than the I/O overhead. In contrast, the performance in Xen was 24–29% better than that in BitVisor when nested SEV was enabled. In KVM, the performance was up to 15% better with SEV passthrough than with SEV virtualization. One reason is that SEV passthrough eliminates the need for bounce buffers in the L2 VM.

Next, we repeatedly requested a large HTML file of approximately 100 KB. As shown in Figure 11(b), the overhead of nested virtualization became much smaller with or without SEV. This is because the network bandwidth became a bottleneck. In BitVisor and Xen, the performance was almost identical to that of single-level virtualization. In KVM, however, the performance degraded by 6.3–23%. These results are similar to those of the iperf3 benchmark.

7.5 Performance of the Kernel Build

We measured the time required to build Linux kernel 6.1.0 with the default configuration. We used as many virtual CPUs as assigned to an L2 VM. As shown in Figure 12, SEV increased the build time by up to 12% even in single-level virtualization. In nested virtualization with KVM, the SEV-disabled L2 VM increased the build time by 6.1%, while the SEV-enabled L2 VMs increased it by 5.8–22%, compared with single-level virtualization. Notably, the slowdown was greatest in SEV-ES virtualization and smallest in SEV-SNP virtualization. The reasons are likely the same as those for the iperf3 benchmark because the kernel build generated intensive disk I/O. In BitVisor, the slowdown in nested SEV was only 2.6–6.8%. This is because the L2 VM directly accessed the virtual disk via PCI passthrough. In contrast, the build time was much longer in Xen. The slowdown was 34–40% in the SEV-enabled L2 VMs and 34–36% even in the SEV-disabled L2 VMs. Since building a large Linux kernel spawned many processes, the L2 VM needed to modify the page tables frequently. This overhead was higher because a paravirtualized L2 VM in Xen issued a hypercall whenever it modified a PTE.

7.6 Boot Performance

We measured the time required to boot an L2 VM. We defined the boot time as the interval between the creation of the L2 VM and the point at which the SSH server became available

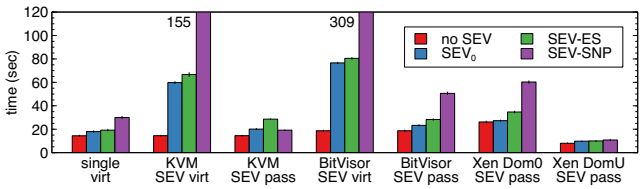


Figure 13: Boot time of an L2 VM.

in the VM. In KVM, OVMF or the custom firmware executed GRUB2, which loaded the Linux kernel and the initial RAM disk. Note that, for BitVisor and Xen's Domain 0, we included the boot time of the L1 VM. This is because the L2 VM was automatically booted immediately after the L1 hypervisor booted, making it difficult to isolate the boot time of the L2 VM.

Figure 13 shows the boot time of an L2 VM. In KVM, SEV_0 virtualization was 4.1x slower than nested virtualization without SEV. This is because the cost of NPT virtualization was high. Since the shadow NPT initially had no entries, a VM exit occurred whenever the L2 VM accessed a memory page for the first time. Then, the L0 hypervisor updated the shadow NPT through complex interactions with the L1 hypervisor, as described in Section 4.3. In the current implementation of SEV_0 virtualization, the L1 hypervisor cannot assign huge pages to the L2 VM. As a result, the L2 VM generated a large number of NPT violations and VM exits during boot.

For BitVisor, in contrast, the overhead of NPT virtualization was smaller because BitVisor could assign 2-MB huge pages to the L2 VM. However, it encrypted a 10-MB Linux kernel image and a 100-MB initial RAM disk using the relatively slow AMD-SP. Therefore, the boot time was 4.1x longer in SEV_0 virtualization. If BitVisor encrypts only the hash values of the kernel and the initial RAM disk using the AMD-SP and lets the firmware verify them, as implemented in KVM [34], this overhead could be reduced.

SEV-SNP slowed down the boot of the VM because the VM had to validate assigned memory using the `PVALIDATE` instruction. OVMF validated approximately 2 GB of memory in advance, while the Linux kernel validated the remaining memory on demand in 2-MB units. In this experiment, the kernel validated approximately 800 MB of memory. As a result, SEV-SNP virtualization significantly degraded boot performance. In particular, BitVisor exhibited the longest boot time. The reason is that BitVisor validated the entire memory of the L2 VM during boot. In addition, the custom firmware did not assign huge pages to the L2 VM, which further increased the overhead of NPT virtualization.

In contrast, SEV passthrough was comparable to single-level virtualization. In KVM, the boot time increased by only 12% and 49% for SEV_0 and SEV-ES, respectively. This is because the L1 hypervisor could assign 2-MB huge pages to the L2 VM, thereby reducing the overhead of NPT virtualization.

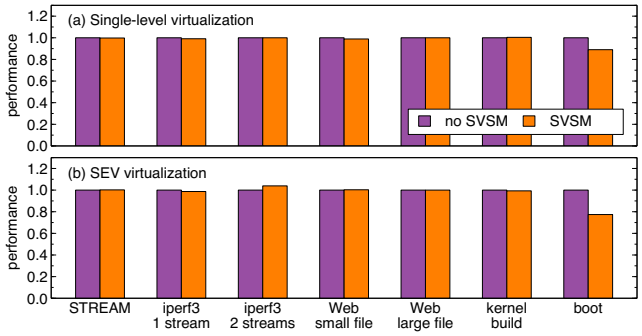


Figure 14: Performance with SVSM enabled.

For SEV-SNP, the boot became 1.6x faster than in single-level virtualization. The reason is that the L1 hypervisor could use 1-GB huge pages, which further reduced the overhead of NPT virtualization.

7.7 Impact of SVSM

We conducted experiments on SNP-enabled VMs running SVSM under single-level virtualization and SEV virtualization with KVM. Figure 14 shows the relative performance in the five experiments described above, using performance without SVSM as the baseline. Except for the boot time, SVSM had little impact on performance. In contrast, the boot performance degraded by 12% and 29% in single-level virtualization and SEV virtualization, respectively. The reason is that the guest OS needed to invoke SVSM to validate memory in VMPL0. The overhead was larger in nested virtualization because each SVSM call required a VM exit to the L0 hypervisor and the execution of `VMRUN` to enter the L1 VM.

7.8 Discussion

SEV virtualization assumes a stronger threat model in which neither the L0 nor the L1 hypervisor is trusted, while SEV passthrough assumes a weaker threat model in which only the L0 hypervisor is untrusted. SEV passthrough consistently performs better for VM exits and the boot time of L2 VMs. For the other workloads, however, a faster mechanism depends on the SEV variant and the L1 hypervisor. Note that SEV passthrough remains useful even when its performance is lower than that of SEV virtualization. This is because several features, such as VM introspection [15], are possible only with SEV passthrough.

For SEV variants, the threat model becomes stronger in the order of SEV_0, SEV-ES, and SEV-SNP, but performance does not necessarily decrease in this order. Since a stronger threat model does not always result in lower performance, users need to select an appropriate SEV variant based on their workloads. For example, they may prefer SEV-SNP for network-intensive workloads. Although SEV-SNP generally outperformed SEV-

ES, the boot time was much longer in SEV-SNP virtualization. When users need to boot L2 VMs frequently, they may choose SEV-ES virtualization if its threat model is acceptable.

Hecate's threat model is similar to that of SEV passthrough. Compared with the performance results reported in the literature [16], SEV passthrough incurs lower overhead for VM exits, while Hecate achieves better performance in network communication and Web access. Direct comparison is difficult because the experimental conditions differ, but Hecate could reduce overhead by supporting only a single L2 VM. The threat model of Microsoft's patch is similar to that of SEV virtualization, except that the L1 hypervisor must trust the L0 hypervisor. Therefore, its threat model is weaker than that of SEV virtualization. The performance results for Microsoft's patch have not been reported, but it is expected to outperform SEV virtualization because the L0 hypervisor can directly access the memory and register state of the L1 VM to virtualize the L1 hypervisor. However, the additional overhead in SEV virtualization is necessary to protect the L1 hypervisor and ensure the correct execution of L2 VMs.

8 Related Work

Microsoft's patch [40] runs KVM as the L1 hypervisor on top of Hyper-V and enables only SEV-SNP for L2 VMs. Like SEV virtualization, the L0 hypervisor provides a virtual AMD-SP and a virtual RMP to the L1 VM. However, the patch cannot protect the L1 hypervisor against the L0 hypervisor because it cannot apply SEV-SNP to the L1 VM. Therefore, it assumes that the L0 hypervisor is trusted. Since the L0 hypervisor can directly access the memory of the L1 VM, support for SEV-enabled L2 VMs can be implemented more easily.

Hecate [16] also runs KVM as the L1 hypervisor on top of Hyper-V but enables SEV-SNP for both L1 and L2 VMs. Its purpose is to run a legacy guest OS without SEV-SNP support in an L2 VM. Unlike nested SEV, Hecate can support only SEV-SNP because it is implemented using VMPLs. In addition, it cannot run multiple L2 VMs in an L1 VM due to the lack of support for MMU virtualization in VMPLs. Since the L1 hypervisor runs at the highest privilege level, L2 VMs cannot be protected from the L1 hypervisor. This trust model is the same as that of SEV passthrough, but Hecate cannot support the stronger trust model assumed by SEV virtualization. OpenHCL [37] is similar to Hecate and offers various services, such as device emulation, via paravisor running in VMPL0. It supports not only SEV-SNP but also Intel TDX [25].

SGX virtualization [22] has been proposed to use Intel SGX [32] in VMs. SGX is another TEE and enables a process to securely execute code in a protection domain called an enclave. SGX virtualization exposes only a portion of the enclave page cache to a VM and emulates the `CPUID` instruction and MSRs. However, SGX cannot be used in confidential VMs protected by TDX. NestedSGX [46] enables applications to use SGX-compatible enclaves in confidential VMs protected by SEV-SNP. It securely runs enclaves in VMPL0 while running the untrusted parts of the application and the OS at a lower privilege level. vSGX [48] emulates SGX-compatible enclaves in a VM by leveraging another SEV-enabled VM.

Nested enclave [39] can run inner enclaves in an SGX enclave. An outer enclave cannot access its inner enclaves, while an inner enclave can access its outer enclave. Each inner enclave is isolated from other inner enclaves. For example, nested enclave can be used to protect an application from untrusted third-party libraries in an enclave. Unlike nested SEV, it requires extensions to SGX hardware.

Ryoan [23] creates an SGX enclave in clouds and constructs a sandbox inside it using Google NaCl [18]. NaCl verifies the code executed in the sandbox and performs runtime checks to securely run cloud services. Similarly, AccTEE [17] constructs a bi-directional sandbox inside an SGX enclave using WebAssembly [20]. Like NaCl, WebAssembly enables code to run inside the sandbox. In nested SEV, an SEV-enabled L2 VM can be used as a sandbox to isolate untrusted code.

9 Conclusion

This paper proposed nested SEV to provide secure and generic SEV support for nested virtualization. Nested SEV supports two trust models that differ in whether the L1 hypervisor is trusted. To support these trust models, it provides two mechanisms: SEV virtualization and SEV passthrough. These mechanisms rely on two techniques called emulation-less multiplexing and SEV context decoupling. Nested SEV supports three SEV variants and enables users to balance security and performance. We implemented nested SEV using KVM, BitVisor, and Xen (paravirtualization) as L1 hypervisors. We conducted experiments to evaluate the overhead of nested SEV and showed that the average performance degradation ranged from 0.9% to 30% across SEV variants and L1 hypervisors.

One direction for future work is to improve the performance of nested SEV, particularly its network performance. Another is to extend our approach to Intel TDX and validate the generality of the design of nested SEV.

Acknowledgments

We would like to thank our shepherd and the anonymous reviewers for their valuable comments and suggestions, which helped improve this paper. This work was partially supported by JST, CREST Grant Number JPMJCR21M4, Japan.

References

[1] Advanced Micro Devices, Inc. Protecting VM Register State with SEV-ES. White paper, AMD, 2017.

[2] Advanced Micro Devices, Inc. AMD SEV-SNP: Strengthening VM Isolation with Integrity Protection and More. White paper, AMD, 2020.

[3] Advanced Micro Devices, Inc. Secure Encrypted Virtualization API Version 0.24. White paper, AMD, 2020.

[4] Advanced Micro Devices, Inc. AMD Memory Encryption. White paper, AMD, 2021.

[5] Advanced Micro Devices, Inc. *AMD64 Architecture Programmer's Manual Revision 4.08*, 2024.

[6] Advanced Micro Devices, Inc. SEV-ES Guest-Hypervisor Communication Block Standardization Revision 2.04, 2025.

[7] Amazon Web Services, Inc. Amazon Virtual Private Cloud. https://aws.amazon.com/vpc/.

[8] Amazon Web Services, Inc. AMD SEV-SNP for Amazon EC2 Instances. https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/sev-snp.html.

[9] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the Art of Virtualization. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles*, pages 164–177, 2003.

[10] M. Ben-Yehuda, M. D. Day, Z. Dubitzky, M. Factor, N. Har'El, A. Gordon, A. Liguori, O. Wasserman, and B.-A. Yassour. The Turtles Project: Design and Implementation of Nested Virtualization. In *Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation*, pages 423–436, 2010.

[11] Confidential Containers Contributors. Confidential Containers. https://confidentialcontainers.org/.

[12] ESnet. iperf3: A TCP, UDP, and SCTP Network Bandwidth Measurement Tool. https://github.com/esnet/iperf.

[13] M. Fedoseev. bombardier. https://github.com/codesenberg/bombardier.

[14] A. Fishman, M. Rapoport, E. Budilovsky, and I. Eidus. HVX: Virtualizing the Cloud. In *Proceedings of the 5th USENIX Workshop on Hot Topics in Cloud Computing*, 2013.

[15] T. Garfinkel and M. Rosenblum. A Virtual Machine Introspection Based Architecture for Intrusion Detection. In *Proceedings of Network and Distributed System Security Symposium*, pages 191–206, 2003.

[16] X. Ge, H. Kuo, and W. Cui. Hecate: Lifting and Shifting On-premises Workloads to an Untrusted Cloud. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, pages 1231–1242, 2022.

[17] D. Goltzsche, M. Nieke, T. Knauth, and R. Kapitza. AccTEE: A WebAssembly-based Two-way Sandbox for Trusted Resource Accounting. In *Proceedings of the 20th International Middleware Conference*, pages 123–135, 2019.

[18] Google, Inc. Native Client. https://developer.chrome.com/docs/native-client/.

[19] Google LLC. Confidential Computing. https://cloud.google.com/security/products/confidential-computing.

[20] A. Haas, A. Rossberg, D. Schuff, B. Titzer, M. Holman, D. Gohman, L. Wagner, A. Zakai, and J. Bastien. Bringing the Web Up to Speed with WebAssembly. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 185–200, 2017.

[21] H. Huang, J. Lai, J. Rao, H. Lu, W. Hou, H. Su, Q. Xu, J. Zhong, J. Zeng, X. Wang, Z. He, W. Han, J. Liu, T. Ma, and S. Wu. PVM: Efficient Shadow Paging for Deploying Secure Containers in Cloud-native Environment. In *Proceedings of the 29th Symposium on Operating Systems Principles*, pages 515–530, 2023.

[22] K. Huang. Introduction to Intel SGX and SGX Virtualization. Xen Project Developer and Design Summit, 2017.

[23] T. Hunt, Z. Zhu, Y. Xu, S. Peter, and E. Witchel. A Distributed Sandbox for Untrusted Computation on Secret Data. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation*, pages 533–549, 2016.

[24] Intel Corporation. Fast CRC Computation for Generic Polynomials Using PCLMULQDQ Instruction. White paper, Intel, 2009.

[25] Intel Corporation. Intel Trust Domain Extension. White paper, Intel, 2023.

[26] A. Kivity, Y. Kamay, D. Laor, U. Lublin, and A. Liguori. kvm: the Linux Virtual Machine Monitor. In *Proceedings of Ottawa Linux Symposium*, pages 225–230, 2007.

[27] D. Li, Z. Mi, C. Ji, Y. Tan, B. Zang, H. Guan, and H. Chen. Bifrost: Analysis and Optimization of Network I/O Tax in Confidential Virtual Machines. In *Proceedings of the 2023 USENIX Annual Technical Conference*, pages 1–15, 2023.

[28] J. Lim, C. Dall, S. Li, J. Nieh, and M. Zyngier. NEVE: Nested Virtualization Extensions for ARM. In *Proceedings of the 26th Symposium on Operating Systems Principles*, pages 201–217, 2017.

[29] J. Lim and J. Nieh. Optimizing Nested Virtualization Performance Using Direct Virtual Hardware. In *Proceedings of the 25th International Conference on Architectural Support for Programming Languages and Operating Systems*, page 557–574, 2020.

[30] C. Liu and Y. Mao. Inception: Towards a Nested Cloud Architecture. In *Proceedings of the 5th USENIX Workshop on Hot Topics in Cloud Computing*, 2013.

[31] J. McCalpin. Memory Bandwidth and Machine Balance in Current High Performance Computers. *IEEE Computer Society Technical Committee on Computer Architecture Newsletter*, pages 19–25, 1995.

[32] F. McKeen, I. Alexandrovich, A. Berenzon, C. Rozas, H. Shafi, V. Shanbhogue, and U. Savagaonkar. Innovative Instructions and Software Model for Isolated Execution. In *Proceedings of the 2nd International Workshop on Hardware and Architectural Support for Security and Privacy*, 2013.

[33] Microsoft Corporation. Azure Confidential Computing. https://learn.microsoft.com/en-us/azure/confidential-computing/.

[34] D. Murik and H. Franke. Securing Linux VM Boot with AMD SEV Measurement. KVM Forum 2021, 2021.

[35] V. Narayanan, C. Carvalho, A. Ruocco, G. Almasi, J. Bottomley, M. Ye, T. Feldman-Fitzthum, D. Buono, H. Franke, and A. Burtsev. Remote Attestation of Confidential VMs Using Ephemeral vTPMs. In *Proceedings of the 39th Annual Computer Security Applications Conference*, pages 732–743, 2023.

[36] K. Negy, T. Nightingale, H. Weatherspoon, and Z. Shen. Towards Swap-Free, Continuous Ballooning for Fast, Cloud-Based Virtual Machine Migrations. In *Proceedings of the 2024 ACM Symposium on Cloud Computing*, pages 269–283, 2024.

[37] C. Oo. OpenHCL: A Linux based paravisor for Confidential VMs. Linux Plumbers Conference 2024, 2024.

[38] P. Bonzini. Minimal x86 Firmware for Booting Linux Kernels. https://github.com/bonzini/qboot.

[39] J. Park, N. Kang, T. Kim, Y. Kwon, and J. Huh. Nested Enclave: Supporting Fine-grained Hierarchical Isolation with SGX. In *Proceedings of the 47th ACM/IEEE Annual International Symposium on Computer Architecture*, pages 776–789, 2020.

[40] J. Piotrowski. Support Nested SNP KVM Guests on Hyper-V. https://lwn.net/Articles/923183/, 2023.

[41] T. Shinagawa, H. Eiraku, K. Tanimoto, K. Omote, S. Hasegawa, T. Horie, M. Hirano, K. Kourai, Y. Oyama, E. Kawai, K. Kono, S. Chiba, Y. Shinjo, and K. Kato. BitVisor: A Thin Hypervisor for Enforcing I/O Device Security. In *Proceedings of the 2009 ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, pages 121–130, 2009.

[42] M. Stigge, H. Plötz, W. Müller, and J. Redlich. Reversing CRC – Theory and Practice. Technical Report SAR-PR-2006-05, Humboldt University Berlin, 2006.

[43] SUSE. COCONUT Secure VM Service Module. https://github.com/coconut-svsm/svsm.

[44] The Apache Software Foundation. The Apache HTTP Server Project. https://httpd.apache.org/.

[45] UEFI Forum, Inc. Advanced Configuration and Power Interface (ACPI) Specification Version 6.4. https://uefi.org/htmlspecs/ACPI_Spec_6_4_html/, 2021.

[46] W. Wang, L. Song, B. Mei, S. Liu, S. Zhao, S. Yan, X. Wang, D. Meng, and R. Hou. The Road to Trust: Building Enclaves within Confidential VMs. In *Proceedings of Network and Distributed System Security Symposium 2025*, 2025.

[47] D. Williams, H. Jamjoom, and H. Weatherspoon. The Xen-Blanket: Virtualize Once, Run Everywhere. In *Proceedings of the 7th ACM European Conference on Computer Systems*, pages 113–126, 2012.

[48] S. Zhao, M. Li, Y. Zhangyz, and Z. Lin. vSGX: Virtualizing SGX Enclaves on AMD SEV. In *Proceedings of the 2022 IEEE Symposium on Security and Privacy*, pages 321–336, 2022.

μUSB: Practical and Safe USB Driver Reuse for Arm TrustZone

Xuankai Zhang*, Sijin Li*, Pei Meng*, Meng Wang†,
Yongzhao Zhang, Ting Chen, Xiaosong Zhang, and Liwei Guo✉
University of Electronic Science and Technology of China
†CISPA Helmholtz Center for Information Security

Abstract

TrustZone isolates device access with secure I/O, yet lacks support for USB devices, which constitute the most diverse peripherals. Existing approaches to integrate USB drivers into the TEE are impractical due to the complex USB protocols, high-frequency DMA accesses, and varying vendor implementations.

In this paper, we present μUSB, a system that derives a functional, *micro* USB driver for TrustZone from the observable concrete execution traces of a full-fledged, complex USB driver, an approach we dub *record, lift, and replay*. Based on the key insight of kernel specialization and the deterministic nature of USB FSMs, μUSB proposes a lightweight mutational recorder and a novel program analysis technique, addressing two challenges: how to record USB execution sufficiently and analyze the traces deeply?

Across four major USB classes and six diverse devices, μUSB generates performant, ready-to-use USB drivers for TrustZone, enabling in-TEE apps to, for the first time, leverage complex USB devices with near-native performance while still enjoying full TrustZone protection.

1 Introduction

TrustZone is the Trusted Execution Environment (TEE[1]) for Arm devices, widely deployed on mobile/embedded System-on-Chips (SoCs) [29, 7, 1]. TrustZone partitions the execution environment into two hardware-isolated worlds – the normal and secure world, with the latter executing a fully shielded baremetal kernel (e.g., OPTEE-OS [54]), independent of the normal world OS. Beyond CPU, secure world's protection further extends to I/O, providing a security feature known as *secure I/O*. It allows the TEE software to interact directly with I/O devices for data harvesting and processing, while ensuring confidentiality and integrity even under a compromised OS. This capability underpins security-critical, in-TEE *trusted apps* [20] demanding stringent protection of sensitive data such as biometric authentication [1, 25], credential management [29, 90], data analytics [85], and model inference [63].

*These authors contributed equally to this work.
✉: Liwei Guo (lwg@uestc.edu.cn) is the corresponding author.
[1]This paper refers to the TEE as TrustZone, which is our focus.

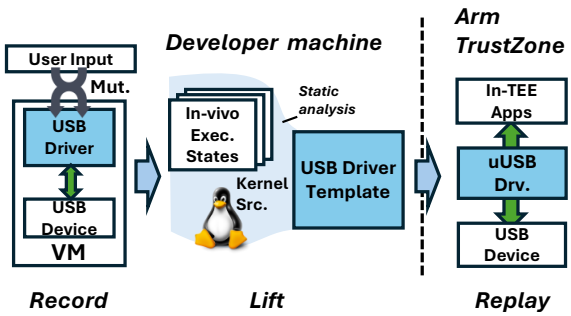


Figure 1: The overview of μUSB.

Secure I/O, however, lacks critical support for USB devices. As the most diverse class of I/O peripherals, they handle data *virtually* from all sources, e.g., from various sensors (including keypads and mice) to multimedia devices such as cameras and microphones; supporting them would bridge existing trusted apps on a Commodity-Off-The-Shelf (COTS) mobile SoC to a much broader range of *trusted* data sources, and empower more trusted apps. For instance, with TEE-protected USB sensors, a trusted app at the edge executing stream analytics [85] directly operates on trusted raw ingress sensor data; it does not relay any requests to an untrusted normal world OS, which is buggy [77, 66, 72] and may be malicious [14].

Yet to close the gap, the key missing link is USB device drivers inside the TrustZone TEE. To the best of our knowledge, no commodity TEE OS supports them to date: 1) commercial TEE OSes such as Android Trusty [29], Qualcomm QSEE [7] and Huawei iTrustee [1] scope their secure I/O support to a few selected proprietary devices, e.g., RPMB storage [47], fingerprint sensors [1]; 2) open-source TEE OSes (e.g., OPTEE-OS [54]), which are incorporating more emerging I/O device drivers (e.g., GPIO [4], SPI [5]), still have no plans for USB despite the first USB feature request dating back to 2017 [6]. As we will show in Section 2.2, it is likely due to a combination of large efforts required to support them and their security implications to the TEE.

How to bring USB drivers into the TEE? One may reuse existing Linux driver code, either wholly (e.g., porting [87, 44]) or partially [31, 110]. Nevertheless, this approach misfits USB drivers which are diverse, complex, and with deep kernel dependencies: porting incurs non-trivial manual efforts, bloats the Trusted Computing Base (TCB), and infects TEE with

kernel bugs which we quantitatively analyze in Section 6.2; partitioning [110, 31] needs to reason about the partitioning boundary which spans over a few thousand kernel functions of major kernel subsystems, and is known challenging [48, 14, 35, 53]. Instead of code, one may opt to reuse driver/device interactions by recording the low-level interaction events induced by a given I/O job and replaying them at run time [32, 83, 106, 56]. This route, however, is difficult too. First, recording by *manual annotations* [83, 106, 56] is tedious and error-prone, where USB drivers exploit in-memory data structures and massive pointer accesses for device interactions; a single missed annotation may result in incorrect results. Second, recording by *symbolic tracing* [32, 16] is unsuitable, because many USB devices interact with drivers through high-frequency (e.g., over 100K per I/O job) *isochronous transfers* bound by strict timing intervals. The high tracing and analysis overhead incurred by a complex symbolic/concolic execution engine (e.g., S2E [17]) is intolerable to such devices and triggers persistent time-outs, as we will discuss in detail in § 2.

Our approach. In light of these, we ask a natural question: can we record USB driver/device interactions *concretely*, and later generalize them for runtime reuse, purely based on these concrete interactions? To answer it, our key insight is that while a full USB driver is complex, its low-level interactions for a specific I/O function follow a highly deterministic *beaten* path, which allows us to sidestep its full complexity.

To this end, our approach adopts a simple idea – *record, lift, and replay*: for a desired USB I/O function, we *in vivo* record the *concrete* low-level USB driver/device interactions, *ex vivo* lift them into a *symbolized* USB driver template, and replay the template with dynamic input at run time to realize the same I/O function. Figure 1 shows the approach overview.

Benefits. Our approach enables *practical* and *safe* USB driver reuse: the developer only needs to specify the desired USB I/O function (e.g., audio record/playback) and corresponding inputs, just as she would expect to use the USB device in TEE, and our system instantly generates the corresponding ready-to-use driver template, faster than driver porting or refactoring, as will be shown in § 6.2; the driver template dictates sequential, replayable low-level driver/device interactions, which precludes kernel code and potential vulnerabilities contained within it (e.g., 0-days).

Challenges. To *record, lift, and replay*, we face two major challenges. First, how to record necessary and sufficient *in-vivo* execution states from USB driver executions? As mentioned earlier, USB driver/device interactions are dominated by high-frequency DMA accesses, where many allocated addresses/parameters vary dynamically over time. A successful recording must capture such accesses completely and promptly: it does not disrupt USB execution; it preserves sufficient information to allow faithful reconstruction of the USB I/O function which tolerates runtime variations. Second, how to lift the concrete traces into templates which accept dynamic inputs at run time? The process essentially recovers the dependencies and constraints missed from concrete traces back to their symbolic and programmatic form. Doing so soundly requires high-fidelity and deep static analysis. However, such analysis is known difficult for the complex kernel code [112, 12], which exploits wild pointer conversions and recursive data structures (e.g., a self-referencing page struct).

μUSB. Towards practical and safe USB driver reuse, we present μUSB, a system which has addressed the aforementioned challenges by employing a suite of novel techniques. For recording necessary and sufficient execution states, μUSB employs a *mutational recorder* in a lightweight VM that captures complete hardware I/O traces without disrupting the timing-sensitive USB protocol. In addition to user-supplied concrete input, the recorder automatically mutates these inputs using a set of mutation rules designed to make mutated inputs closely approximate the *valid* inputs of the recorded I/O function. With the new inputs, it then re-records the I/O function to gather more traces, until they converge structurally (i.e., isomorphically) with the same I/O function.

For automatically lifting concrete traces into templates, μUSB integrates and extends two successful program analysis techniques and proposes *symbolic trace differential analysis* and *qualified taint tracking*. With simple heuristics, the former effectively filters out and symbolizes concrete values that *may* change during execution (e.g., a virtual address), whereas the latter is a form of trace-guided analysis that leverages the recording as an oracle path qualifier to make a deep, high-fidelity (i.e., context-, path-, flow- and field-sensitive) data-flow analysis on complex kernel code scalable and precise. The two novel techniques harmonize in exploiting the inherent invariance and variability of the deterministic USB FSM to produce the USB driver template used for replay, which at a high level specifies the behavior of the USB driver – *when* should *which* data go *where* in *what* format.

Results. We apply μUSB to four important USB classes (i.e., Mass Storage, Audio, Video, HID) encompassing six different devices from five different vendors, whose drivers were considered too complex for Arm TrustZone [110, 32]. With light efforts and basic knowledge of USB device internals, μUSB successfully produced eight diverse templates; each template comprises a vastly different number of events (e.g., from 1K to over 400K) and encodes necessary and sufficient dependencies and constraints to achieve its designated I/O functions. The resultant μUSB drivers have decent performance: on Raspberry Pi 5, a popular COTS Arm development board, and for the first time without a normal OS, the TEE software can read/write USB storage at 10-20 MiB/s (close to a full-fledged native driver), stream video at 30FPS, record audio reliably in real time, and receive keyboard/mouse inputs at low latencies.

Contributions. Our contributions are as follows:

• The design and implementation of μUSB, a novel *record, lift, and replay* solution to reusing mature USB device drivers for TrustZone. μUSB demonstrates: by combining concrete USB driver execution traces with simple static analysis, it is possible to reproduce and reuse complex USB I/O functions.

• Qualified taint tracking, a novel trace-guided static analysis technique that makes high-fidelity dependency recovery in complex kernel code practical and fast.

• A comprehensive evaluation of μUSB. By deploying μUSB on various USB devices (e.g., streaming webcam, speaker) on real Arm TrustZone-enabled hardware, we demonstrate the practicality and scalability of μUSB.

• Ready-to-use, performant μUSB drivers for TEE. To the best of our knowledge, it is the first time for complex USB device drivers to enjoy end-to-end TrustZone protection.

For a TrustZone TEE, μUSB [2] opens a new door to trusted apps accessing diverse USB devices.

2 Background & Motivations

2.1 A Primer on USB

USB protocol. Universal Serial Bus (USB) is the dominant peripheral interconnect standard, evolving from USB1.0 [102] to USB4.x [103]. To standardize peripheral-to-host communication, it provides two key abstractions. *Endpoints* represent data sources or sinks on a device and communicate with the host via logical channels called pipes. *Transfers* encapsulate communication between endpoints and the host, with four types defined according to data content and purpose: *1) control transfers* for device setup and configuration; *2) interrupt transfers* for regular, low-latency communication such as keyboards; *3) bulk transfers* for reliable, high-throughput exchange as in storage devices; and *4) isochronous (ISOC) transfers* for time-sensitive streaming with bounded latency, essential for multimedia devices. Through these transfer types, USB abstracts diverse peripherals into 25 device classes, e.g., Human Interface Device (HID), Video, Audio, Mass Storage.

eXtensible Host Controller Interface (xHCI). A host controller interface (HCI) implements the interface specified by the USB protocol for device interaction, including OHCI/EHCI, UHCI, and xHCI. This paper mainly focuses on xHCI, as it is the most recent and widely adopted standard [101]. Mandated by the specification, xHCI is architecture-independent and exposes a large group of memory-mapped registers for host configuration. These include *capability registers*, which report protocol information, and reference *operational registers* and *runtime registers*. The latter configure USB data transmission parameters at run time, e.g., data page size, and commands. Beyond registers, xHCI mainly manages USB transfers with in-memory data structures. It employs *Transfer Descriptors (TDs)*, each containing one or more *Transfer Request Blocks (TRBs)* depending on the transfer size and type, which refers actual data pages. All TRBs are organized within a ring buffer, the *Transfer Ring (TR)*.

[2]The code is at https://github.com/SECG-UESTC/uusb/

2.2 The USB Secure I/O Dilemma

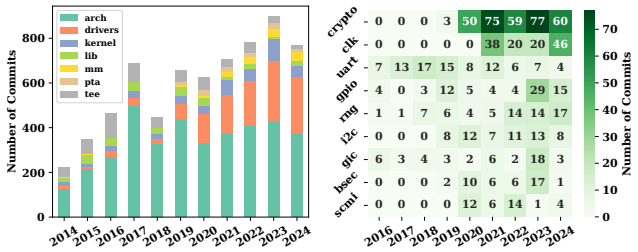
What USB devices matter for TEE? Following example trusted apps [20] motivate our design and benefit from TEE-protected USB devices: *1) Secure storage*: trusted apps manage sensitive data such as credentials [90], authentication tokens [29], and biometric data [1, 25]. They acquire/persist the data on storage. *2) Trusted perception*: they acquire sensor data [110, 56] for executing stream analytics [85] and NN inference [109, 97]; notably, multimedia data are often unencrypted thus cannot be exposed to normal OS. *3) Trusted UI*: they render privacy-sensitive contents, e.g. bank account; the UI reads in user inputs such as presses from a keypad or movements from a mouse. TEE needs to isolate input devices for confidentiality and integrity [82, 107].

Why TEEs lack USB support? Despite their strong use cases, USB support remains untapped by existing TEEs. We examine OPTEE-OS [54], TrustZone's reference secure OS, analyzing its development and commit history over the past decade (Figure 2(a)). The data reveals a clear upward trend in driver-related commits, which have doubled in recent four years, and now account for 35% of total commits. However, closer inspection shows that the majority (73%) target relatively simple character devices such as CRYPTO, UART, and GPIO which provide only basic I/O services (Figure 2(b)).

Why has USB remained overlooked by TEE, with no indication of future development? Our conjecture is twofold: *1) USB couples with heterogeneous and diverse standards.* Supporting different device classes (i.e., 25 in total) demands compliance with heterogeneous standards and class-specific encodings, e.g., video and audio codecs; bringing them into TEE is unwarranted. *2) The large USB software stack deters developers.* Existing in-TEE drivers are compact and simple (under 77k SLoC shown in Figure 2(b)). A single USB audio driver spans 245K SLoC (Linux v6.13), compared to only 300K SLoC for the entire OPTEE-OS; its dependencies span over 600 header files of tens of kernel subsystems (e.g., IRQ, DMA, file systems) accounting for additional 560K SLoC.

Why are existing approaches inadequate? We examine existing approaches which may bring USB drivers into TEE.

1. *Porting.* One may port the full or partial USB driver and shift them to the TEE, e.g., as a library OS [44]. This approach is unfit for the TEE, as it introduces a large and unwarranted USB driver along with its deep kernel dependencies to TEE (as discussed earlier). As will be shown quantitatively later, it incurs large manual efforts (§ 6.1), and inherits existing kernel bugs [67, 71, 75] (§ 6.2), which makes the TEE susceptible to attacks from malicious USB devices [57, 79].



(a) Yearly commits grouped by kernel subsystems from 2014.

(b) A breakdown of OP-TEE driver commits from 2016.

Figure 2: OP-TEE OS commits over the past decade.

2. *Partitioned execution.* Prior works [110, 22, 36] partition the existing drivers and execute only the security-sensitive code inside the TEE while leaving the rest to normal OS. Despite having less in-TEE code, it has two security issues. First, reliance on non-secure OS code expands TCB and exposes the TEE to Iago attacks [14], which compromises data confidentiality and integrity as USB devices commonly lack end-to-end encryption. Second, defining partitioning boundaries is challenging [48], as it requires extensive auditing of interfaces and data structures (e.g., USB TRBs, DMA descriptors, kernel pages) that may handle USB data, which likely results in a major overhaul of USB drivers and kernel code.

3. *Record-and-Replay (RnR).* As old wisdom among software testing [26, 111] and bug finding techniques [23, 33], it recently proves effective for TEE driver reuse, e.g., for simple I/O devices [32, 106, 56], GPUs [83]. Its key idea is to record the driver/device interactions in a commodity OS and replay them in TEE. To record, they either manually annotate the kernel driver to instrument software/hardware interactions [83, 106] or symbolically trace these interactions using a symbolic/concolic execution engine [32, 16].

However, both recording methods are inadequate for USB devices. As mentioned earlier, USB primarily uses fine-grained DMA accesses for driver/device interactions [41], which does not retrofit kernel register accessors (e.g., `readl`, `writel`) and differs from GPUs [83] and other I/O devices [56, 106]. Manually instrumenting these DMA accesses hence is tedious and fragile, and incurs prohibitive efforts when scaling across diverse USB devices and classes. Conversely, symbolic tracing alleviates this instrumentation burden but is too heavyweight for the USB protocol – the USB device times out and halts if an interaction misses its timing (e.g., 1ms interval of isochronous transfers), which is hard, if not impossible, for a symbolic/concolic execution engine to keep up with. Therefore, driverlet [32] as a notable representation of this approach, only supports USB bulk transfer which tolerates analysis delays but does not work for other transfer types. Furthermore, even for USB storage, it needs to explicitly disable Start-of-Frame as a workaround for EHCI protocol's timing control, stated in its § 7.2.2.

2.3 Goals, opportunities, and choices

Design goals. To extend TEE protection to USB for COTS SoCs at a low cost, we set the following goals.

- **Practical.** It is our top concern. The system shall scale to major classes of USB devices (e.g., audio/video) which have important I/O functions with little to no manual effort.
- **Safe.** USB drivers execute entirely inside the TEE to leverage secure I/O, and remain safe against a malicious USB device.
- **Performant.** Multimedia devices often have real-time data transmission, e.g., microphones stream audio samples at constant rates for sound quality. The system and the drivers must deliver deployable and usable performance.

Observations. We have exploited the following observations.

– The beaten USB FSM state transition path. The USB FSMs are by design deterministic and largely data-independent [104, 102, 103]. For a given I/O function and fixed inputs, the USB device's internal FSM follows the same, *beaten* state transition path; conversely, driving the device to go through the same path reproduces the same I/O function.

– Architecture-independent USB driver/device interactions. Operating a USB device entails driver–host controller (xHCI) interaction via MMIO, DMA and interrupts, with architecture-independent data structures and register specifications designed for scalability across platforms.

– Limited dynamism of trusted apps. Trusted apps (§ 2.2) are statically bound to TEE, constraining the USB usage: the TEE expects persistent USB device connection (e.g., no hotplug); they demand I/O data, which can be served with corresponding I/O functions without complex and dynamic device features, e.g., power management.

Design choices. The above observations and goals motivate our design choices: (1) prioritize specific USB I/O functions over full device features for practicality; (2) decouple analysis from USB driver execution by *in vivo* recording concrete execution states, and by *ex vivo* analyzing concrete recordings exploiting the deterministic, beaten USB FSM path.

3 Design Overview

3.1 System model

Threat model. We follow the threat model of TrustZone and USB security [31, 42]. We trust the Arm SoC hardware, including TrustZone Address Space Controller (TZASC) and TrustZone Protection Controller (TZPC), and firmware enforcing the secure/normal world isolation. We trust the secure world software, including the TEE kernel and trusted apps. We do not trust the normal world OS, which may probe or tamper with USB data. Physical attacks [45, 34] are out of scope.

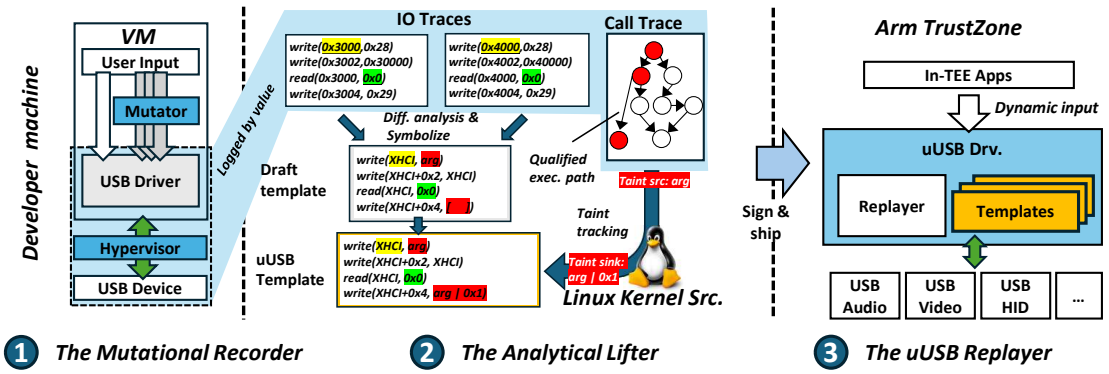


Figure 3: Overall workflow of μUSB

We assume software and hardware on the developer's machine used for recording are trustworthy, including the OS, USB devices and drivers. We also assume that the recorded USB drivers are *gold*: they implement sufficient driver/device interactions, so that it can assess if the device has finished the state transitions needed by given requests. Our rationale is similar to other recording-based TEE drivers [32, 83, 84, 106].

Targeted USB devices and SoC hardware. We target mobile/embedded Arm SoCs with TrustZone enabled. We focus on USB I/O devices with strong TEE use cases (§ 2.2). These devices handle sensitive user data but lack end-to-end encryption at the protocol layer. We do not target USB networking or hub devices, as their safety can be complemented by protocol-level encryption [92]. We also do not consider USB gadgets due to unclear TEE use cases. The SoC has multiple instances of the USB host controller, which is a common practice in today's SoCs [21, 3]. By configuring TZPC and TZASC, the TEE exclusively owns one instance of the USB host controller.

3.2 μUSB overview

Our key idea is to specialize a TEE USB driver for specific I/O functions by ex vivo deriving a *micro USB* (μUSB) driver from the concrete in-vivo execution of a mature kernel driver. Figure 3 shows workflow. The core mechanism is as follows.

1. *Mutational Recording.* The developer invokes the USB IO function desired by trusted apps with concrete inputs, e.g., `read 10 blocks from sector 0`. μUSB mutates the inputs, invokes the same I/O function for multiple times, and logs complete interaction events from the USB driver, including register/DMA accesses, interrupts, and kernel inputs.

2. *Static Lifting.* μUSB processes traces logged from multiple concrete inputs into a template which adapts to dynamic inputs at run time: it detects variables and symbolizes them by differentially comparing their values across traces; it uses traces to *qualify* taint analysis for distilling dependencies and constraints between variables. The lifted template comprises a sequence of low-level hardware/software interaction events (e.g., MMIO, IRQs) required to complete the specific I/O function, which precludes the original kernel code.

3. *Runtime Replay.* μUSB signs and ships the template to TEE as the ready-to-use μUSB driver, which exposes a simple interface to trusted apps for requesting I/O data. Upon request, the μUSB driver replays the events in the template sequentially with runtime inputs, e.g., `read X blocks from sector Y`. A faithful replay fulfills the same recorded I/O function while adapting to various dynamic inputs.

Applicability to other TEEs. A μUSB driver relies on secure I/O provided by TrustZone for strong protection but is not strictly limited to it. It also supports other TEEs providing similar features. For instance, AMD SEV-SNP uses SEV-TIO (i.e., Trusted I/O) [9], similar to TrustZone's secure I/O; RISC-V's Keystone TEE [46] combines PMP (i.e., memory/register isolation) and processors M-mode to achieve secure I/O.

3.3 Why μUSB works

Understanding μUSB drivers at a high level. Based on our rationale of the USB device FSM (§ 2.3), a USB device generalizes as a set of deterministic FSM state transition paths, which collectively fulfill the I/O functions of the device. The corresponding full USB driver hence can be regarded as *traces* of low-level device interactions, i.e., MMIO, DMA, and IRQs; each trace 1) drives the device state transitions of a path, and 2) also assesses if the transitions are correct. A μUSB driver captures a subset of full traces by recording low-level interactions from demanded I/O functions; replaying the trace drives the device FSM to go through the same state transitions to fulfill the recorded I/O function. The appendix presents a sketch proof showing a μUSB driver achieves trace equivalence with its recorded USB driver under the same I/O function.

Replay correctness. The μUSB driver offers the same level of correctness guarantee as a kernel driver does: the μUSB replayer's assertion that a recorded I/O function (e.g., read/write a block, streaming audio for some interval) has completed successfully is as sound as the assertion from the kernel driver.

Our rationale is: the full driver continuously assesses if state transitions are correct via low-level interactions. By recording

Recorded Interaction Events and Formats			Description
I/O events	DMA/ MMIO	write(**addr**, **val**, **len**)	Write len bytes of val to addr
		val = read(**addr**, **len**)	Read len bytes of val from addr
	IRQ	on_irq(**n**)	Receive an IRQ at n-th interrupt line
Kernel Inputs	Clock Inputs	**ts** = ktime_get_real/raw/current()	Get current timestamp ts
	DMA Buffer Allocation	**addr** = dma_alloc_coherent (dev, **size**, handle, gfp)	Allocate size bytes of DMA buffer at addr
	Random number	**num**=random()	Generate a random number num

Table 1: Events recorded by μUSB. μUSB records **highlighted** fields by value.

and matching these interactions, the μUSB driver also asserts execution correctness of the I/O function. During replay, if it observes the same trace of interactions with all parameters matched, then to the best of the driver's knowledge, the USB device performs the same state transitions and completes the same I/O function, and the replay is correct.

At a higher level, μUSB resembles duck typing [55] for USB driver construction: upon user inputs, a μUSB driver executes as a USB driver; upon device inputs, it reacts as a driver; then it must be as correct as a driver.

4 μUSB Design

4.1 The Mutational Recorder

The recorder records the *in-vivo* execution states of the USB driver by value as the basis for the template. To this end, we have addressed two challenges: 1) how to capture *necessary* interaction events to faithfully reconstruct the execution states needed to recover USB driver execution? 2) how to obtain *sufficient* concrete values such that the lifter can distill data dependencies and symbolize them for dynamic inputs?

4.1.1 Record necessary in-vivo execution states

To initiate, the developer launches a *recording campaign* by invoking the entry function provided by the recorder on the developer's machine: `record(f, [arg...], [var...])`, where `f` is the desired USB I/O function (e.g., `read/write` for USB storage), `[arg]` is an array of concrete inputs needed by `f`, and `var` is a binary array indicating which arguments in `[arg]` accept dynamic inputs at run time. More I/O functions recorded by μUSB are in Table 4.

What to record? Two types of runtime traces are necessary: **1) I/O traces:** these include the low-level USB driver/device interaction events, i.e., xHCI register and DMA accesses, and interrupts; **2) Kernel inputs:** USB devices rely on inputs from a few kernel APIs to alter states. For instance, its device FSM expects a DMA buffer from kernel to initiate data transfer. The recorder logs their arguments and return values in the I/O trace; Table 1 summarizes all observed kernel inputs.

The recorder additionally captures call traces. Section 4.2 demonstrates existing static analysis is infeasible for the lifter to analyze kernel deeply and precisely without them.

How to record? The recorder must capture complete traces. It is challenged by USB's high-frequency pointer-based DMA accesses (e.g., 10KHz) whose strict timing must be respected. To address this, μUSB records the USB driver execution in a controlled, lightweight virtual machine, and traces I/O at the lowest level via stage-2 page faults. Specifically, the recorder compiles the guest kernel using tinyconfig and enables only necessary USB drivers. It configures the hypervisor to monitor: 1) interrupts and 2) memory regions belonging to USB host controller and 3) DMA pools allocated to USB; tracing extra regions adds overhead and may interfere with USB execution. The recorder thus transparently traps these I/O events and records them by value in the format given in Table 1.

An alternative is to trace via stage-1 page faults, which was our initial attempt. To our surprise, their latency exceeds the USB isochronous transfer interval, which is erroneous and causes the USB transfer to abort; we will show measurements in Section 6. During recording, the recorder logs the I/O events sequentially and segregates them into two contexts (i.e., the CPU and the interrupt context) to match the asynchronous USB job submission/execution.

4.1.2 Mutational recording

Relying on a single trace is brittle, as many values and events are timing- and input-dependent. To accommodate input and runtime variations, our idea is to reveal such dependencies and variability *proactively* by stress testing the given I/O function with *mutations* which approximate these variations.

Mutation strategy. In addition to recording using the user-supplied input, the recorder mutates inputs in two dimensions and re-records. *First, on user inputs*. The recorder mutates all inputs which the user expects to use dynamically at run time, i.e., specified by `[var]` of the recording entry. As explained earlier, the recorder does not aim for code coverage (i.e., desired by an OS fuzzer [42, 86]) but to approximate input variations under the same I/O function. It hence adopts simple, semantic-aware mutators in Table 2 to generate more user inputs for all `arg[i]`, if its corresponding `var[i]` is true (i.e., variable at run time). *Second, on kernel inputs*. The recorder exploits Kernel Address Space Layout Randomization (KASLR) default in today's kernel and restarts the VM between each re-run for a clean-slate kernel state. This helps expose those DMA addresses which appear identical across different runs but are in fact recycled.

Stop condition. During operation, the recorder gathers a *reference trace* from user-supplied input, and additional *mutated traces* from extra runs. The recorder decides to stop upon two conditions. *First, the traces converge*. It deems the exploration sufficient if after N runs, the majority of mutated traces are isomorphic (i.e., structural equivalence [15]) to the reference trace, where the trace lengths, the ordering and types of the events are identical. N is a user-configurable parameter and is at least 10. Larger N (e.g., 100) is fine but incurs longer

Type	Mutator	Example user input fields
Arithmetic (dec/inc)rement	(dec/inc)rement 1,2,3,…,10, 64,128,256,1024	**Storage**: sector, len **Audio**: record/playback duration **Video**: # of requested frames
Address shuffling	Shuffle address space & supply a new buffer	**Storage/Audio/Video/HID**: data buffer address
Random configuration	Randomly sample from valid configurations	**Audio**: sampling rate, audio encoding **Video**: resolution, video encoding

Table 2: Mutation strategy defined by μUSB recorder.

delays as the recording is sequential to ensure determinism; it does not bring additional benefits, for our evaluations in Section 6.3.3 show that as few as six runs is sufficient. *Second, recording times out*. The traces may never converge, due to excessive mutable inputs. For example, the developer specifies all arguments `[sector, blk, buf]` of `read` I/O function as mutable. This apparently will not converge, as allowing *all* inputs to be mutable requires exploring the full driver state space. To deal with such cases, the recorder sets the default timeout to 10 minutes. After timeout, it stops recording and prompts the developer to reduce the number of mutable input in $[var]$, i.e., reducing the size of state space to explore. In practice, we have not found a case where traces do not converge when the number of mutable input reduces to one. This is because in such a case, the mutable input has to be a runtime buffer address, changing which is expected by the USB FSM and still follows the same I/O transition beaten path (§ 2.3).

Recording outcome. Upon completion, the recorder gathers an enclave of runtime traces in a recording campaign, including a reference trace and several mutated traces. These traces capture concrete and varying values, but are structurally equivalent, suggesting a sufficient exploration of the USB device state transition path determined by mutable user inputs.

4.2 The Analytical Lifter

The lifter is the core to μUSB, which distills concrete recordings into a replayable template. To this end, we have addressed two challenges: *given a corpus of concrete execution traces, how to 1) identify the variants (i.e., values that change w.r.t. new inputs) and 2) recover their data dependencies?*

4.2.1 Symbolic differential trace analysis

To generalize to dynamic inputs, the first step is to locate *variants* from concrete I/O traces. A central difficulty is that invariants and variants in I/O traces can appear identical. For instance, a SCSI `read(10)` CMD (0x28) appears no different than a block count or sector number.

To address this, our observation is that invariants capture the constants (e.g., commands) and substructures (e.g., linked lists) preserved across traces, while the variants reflect the input-induced divergences. The lifter thus exploits such inherent dichotomy by differentially comparing the traces.

First, symbolizing addresses and inputs. For a concrete trace, the lifter instantiates three symbolization rules for i-th event $v_i = E_i(args, ...)$ of the trace:

Rule 1 (indexed symbols). Base addresses of USB host controller and allocated DMA buffers are assigned unique symbols w.r.t. their sizes and respective indices i; user inputs, and all v_i returned by kernel inputs and DMA/MMIO reads are assigned unique symbols w.r.t. their indices i.

Rule 2 (address matching). All concrete addresses, i.e., DMA/MMIO read/write addresses, are symbolized by relatively addressing their closest base address.

Rule 3 (argument matching). Concrete values $args$ consumed by MMIO/DMA writes are symbolized by relatively addressing known symbols which prioritize addresses.

To justify, rule 1 first addresses four intuitive sources of variants: device/DMA addresses, inputs from kernel, device, and users (Table 1); the naming allows unique references from other variants within the trace and differential analysis across traces. Rule 2 associates values *known* to be virtual addresses (e.g., USB MMIO read) with already symbolized base addresses. Rule 3 handles the tricky case where a kernel input or address is used as an argument, by conservatively assuming all arguments are variants and associating them with existing symbols. The rationale for the value matching heuristic is that a USB device often uses kernel inputs (i.e., timestamps, random numbers) *as-is*. This may introduce false positives (e.g., a static encoding or a timestamp coincides with a DMA address) but the key is not to leave out true positives. The lifter filters false positives as follows.

Second, detecting invariants differentially. The lifter sequentially scans each I/O trace and creates key-value pairs `<var, [val]>`, where `var` is a symbolized variant and `[val]` is a list of symbols and concrete values (i.e., not symbolized by rule 1-3) associated with the variant in a DMA/MMIO read/write. It detects invariants by differentially comparing the reference trace with all mutated traces: a `val[i]` is an invariant *if and only if* its counterparts of mutated traces are symbolically or numerically equivalent. Figure 4 shows an example: `var_4k_2` and 0xd0 are invariants corresponding to `var_4k_1` and `var_4k_1+0x8` if and only if `var_4k_1` and `var_4k_1+0x8` *always* writes `var_4k_2` and 0xd0 across all traces. It also resolves false positives: as the recorder restarts VM and leverages KASLR (§ 4.1.1), it is unlikely for kernel inputs to persistently coincide with address reads/writes.

At first glance, this approach may appear ad hoc or unsound, i.e., dependent on number of traces, may miss some variants, etc. We highlight the lifter's goal is not to identify *all* variants but only those observable under variable inputs. We argue that these values subject to change would have changed and been captured during the mutational recording (§ 4.1.2). In practice, we find the strategy effective and fitting for USB I/O

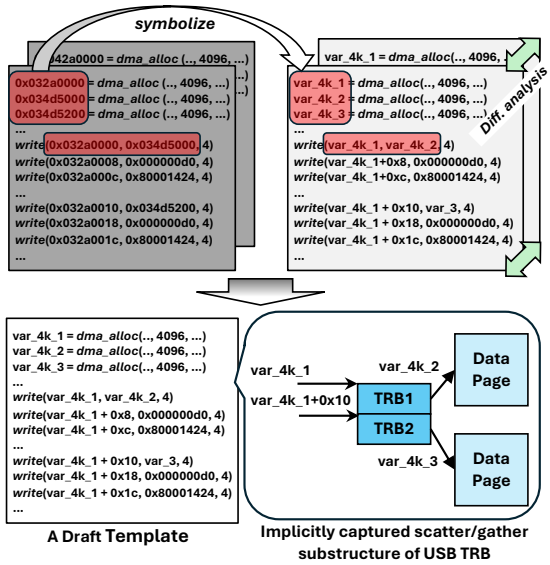


Figure 4: µUSB reconstructs the scatter/gather substructure of TRBs via differential analysis with the simple heuristics.

events, which are highly deterministic. Figure 4 demonstrates this by showing that the actual USB driver sets up TRBs in the TR for submitting I/O jobs with a sequence of DMA writes. With our technique, the lifter identifies the invariants and preserves the substructure, which implicitly reconstructs the same scatter/gather topology of TRBs when being replayed. Section 6.3.3 quantitatively shows the impacts of the number of traces.

Third, generating a draft template. The outcome is a draft template built upon the reference trace: addresses are symbolized canonically; the invariants are preserved as-is, either as symbols or concrete values; kernel inputs are symbolized with best efforts and remaining variants are marked as placeholders for further analysis.

4.2.2 Qualified taint tracking

Variants in the draft template are critical to the USB driver execution, which often involves data transformations (e.g., DMA address alignment) dictated by the driver and hardware specifications. The lifter must resolve their dependencies with user/kernel inputs and bind them with path constraints conforming to the driver behavior. While a high-fidelity static data-flow analysis (i.e., path-, context-, flow-, and field-sensitive) could theoretically resolve the issue, this method is unscalable for complex kernel code due to state explosion[112, 17].

To overcome this, our insight is to specialize the data flow analysis using the call traces collected along with I/O traces (§ 4.1.1) as the sharpest data flow information to *qualify* possible data flows of kernel code paths, i.e., a unique solution to the qualified data flow problem [37]. We present *qualified taint tracking* as follows.

The lifter sets all inputs (i.e., mutable arg of the recording entry, kernel inputs, and read from device states) as taint sources and the driver code which consumes them as sinks. For taint rules, the lifter adapts existing standard rules from [58] and introduces two crucial extensions: 1) since existing taint rules only concern taint propagation in the kernel space, it extends them to allow taints to propagate in field level across the user/kernel boundary through kernel utility functions such as `copy_from_user`; 2) to maintain field-sensitivity for recursive data structures like linked lists (a known challenge for static analysis [112]) for chaining URBs (i.e., the upper-layer kernel abstraction for TRB), the lifter introduces trace-guided alias analysis. Essentially, the improved alias analysis leverages the draft template as an oracle and determines if an object (e.g., URB) is an alias of another from its symbolized address encoded in the draft template. To track taint propagation, the lifter navigates the analysis with the call trace as a path qualifier: at control-flow branches (e.g., `if` statements), it chooses the path suggested by the call trace, and checks if the corresponding path condition is dependent on a tainted value. If so, the lifter collects the path condition as a constraint of the tainted input, which ensures the execution stays on the USB state transition beaten path. Since the analysis is static, the lifter saves these constraints symbolically in the final template. Alongside the taint propagation, the lifter accumulates the taint operations and saves them to generate the final template. We discuss the security implication in Section 6.2.

By design, a taint sink (i.e., a variant) corresponds to at least one taint source. However, if a variant depends on kernel inputs unlisted in Table 1, it cannot be correlated to a taint source and is untagged. To address this, the lifter marks the variant as a taint source, consults the call trace to perform *backward* qualified taint tracking, and adds the taint sink (i.e., a kernel input) to Table 1. The lifter then prompts the recorder to re-record leveraging the newly discovered kernel input. This efficiently and effectively recovers the dependencies and constraints missed from the draft template. For common USB devices (e.g., storage/video/audio/HID), the lifter does not conduct backward taint tracking, as the events in Table 1 are sufficient; nevertheless, we design the mechanism to cover emerging USB devices that may require more events to trace.

4.2.3 Generate the final template

To generate the final template, the lifter simply assigns a symbolic expression to each taint sink (i.e., a variant). The symbolic expression programmatically captures the series of computational operations originated from the variant's original taint sources (e.g., user or kernel inputs), as a result of qualified taint tracking.

Optimization: Loop re-rolling. Multimedia devices use high-frequency isochronous transfer which guarantees high-bandwidth (e.g. 160Mbps for 1080p capture) but bloats the

template, e.g., over 400K events for capturing 10 frames. Directly using the uncompressed template is possible but neither practical nor efficient. Observing the USB TR ring structure and repeated memory access pattern, μUSB leverages peephole optimization with a peephole size of 30 for detecting the pattern(s) and re-rolling the accesses into fixed-iteration loops. After re-rolling, the template size reduces by a few orders of magnitude, which we show quantitatively in Table 4.

4.3 The μUSB Replayer

Running inside Arm TrustZone, the replayer and templates are statically linked with the TEE OS, which incarnates as the **μUSB driver**. It provides a unified and easy-to-use API for in-TEE apps to invoke, executes the events of the template corresponding to the API, and handles possible failures.

The API. The replayer provides one unified API for in-TEE apps to request for USB I/O services. The API has two key arguments. The first specifies the template identified by its I/O function, and the second points to the arguments of the I/O function, which are mutable inputs specified at the recording entry. The design abstracts away the complexity of kernel abstraction (e.g., devfs) in the original recording and allows the in-TEE app to focus on the I/O functions. Figure 10 shows an end-to-end example.

Template execution. Upon API invocation, the replayer instantiates the template and then sequentially executes its events. To instantiate, the replayer configures TZASC/TZPC to isolate and map xHCI for memory accesses and reuses the interrupt handler of the TEE OS for top-half IRQ management. It redirects the events in Table 1 of the template to the TEE OS. Note these events are commonly provided by TEE OSes [54] or can be emulated, e.g., `malloc` of OPTEE-OS already allocates physically contiguous pages which emulates `dma_alloc_coherent`. The replayer maintains two contexts (CPU and interrupt), each holding only a pointer to the next event (like a PC). For writes, it re-applies the accumulated taint operations and writes dynamic values to corresponding addresses, which are mediated and checked to be valid DMA/MMIO addresses and non-executable; for reads, it matches the invariants while ignoring variants. It does not pace the replay, e.g., by injecting any artificial delays, because the USB protocol allows faster driver responses but not late ones. When execution finishes, the replayer frees up the allocated resources from the TEE OS, e.g., DMA.

Handling replay failures. The replayer handles two failure conditions. First, incorrect API invocation. This case includes selecting wrong templates and/or supplying arguments which do not satisfy the path constraints. In this case, the replayer directly returns an error code. Second, reading unexpected values. During replay, the replayer matches interaction events (i.e., IRQs and values read from the USB device) against those recorded in the template. If the lifter deems a value read from the USB device as an invariant but appears different or an unrecorded IRQ emerges, it suggests the USB FSM has gone off the beaten path. For instance, the USB device is unplugged or reboots unexpectedly due to a firmware crash amid execution, which causes spurious IRQs and deviations from the recording. In this case, the replayer halts, resets the device, and re-executes the template. If the error persists, the replayer reports an error code back to the app and releases the allocated resources (e.g., DMA pages), without executing any device-specific recovery logic. The replayer does so because a deviation is precisely the condition under which the original driver itself would not have proceeded; therefore failing is not just safe but is the same decision the original driver would have made (§ 3.3). Doing so also fully confines the μUSB driver execution inside TEE to ensure the confidentiality and integrity of the μUSB driver code and data, which are protected by TEE and never leave TEE memory; Denial-of-Service (DoS) (e.g., unplugging the USB device as mentioned above) is possible but cannot persist, as the replayer detects such deviations from the recording. In practice, no such case was found. See more in Section 6.

5 Implementation and Experiences

μUSB comprises about 4K SLoC across C, C++, and Python. Its core components are detailed below.

The recorder. We built it atop QEMU and KVM (Linux v6.8) in 2K lines of C code. We use QEMU as the frontend to manage the virtual machine which also runs Linux kernel v6.8. For I/O trace, we modify KVM to trap and record them by leveraging the `PT_ACCESS_MASK` to cause EPT page fault and IRQ virtualization to intercept USB interrupts. For call trace, we retrofit the existing kernel tracing facility, i.e., `ftrace` and `kprobe`. We implemented our own mutator and draw valid configurations from device utilities (e.g., `V4L2` for Video).

The lifter. We have implemented the lifter based on SUTURE [112] in 500 lines of C++ code and 1K lines of Python code. We choose it as it is the state-of-the-art kernel static analyzer, which integrates mature taint rules. To implement qualified taint tracking, we modified its taint tracking module to allow the traces to 1) direct the taint propagation upon branches, to 2) collect the corresponding path constraints, and to 3) qualify alias analysis upon recursive data structures.

The replayer. We have implemented the replayer as a simple executor of the events of the template in 500 lines of C code based on OPTEE-OS v4.4. For DMA/MMIO accesses, the replayer implements them as uncached pointer accesses and insert memory barriers for cache coherence. For call events, the replayer redirects them to existing services already provided by OPTEE-OS, e.g., `dma_alloc_coherent` to `malloc` of OPTEE-OS which allocates physically contiguous pages.

Test Plat.	**SoC**	Raspberry Pi 5	**CPU**	4 x Cortex-A76@2.4GHz
	Normal OS	Linux v6.4	**Secure OS**	OP-TEE 4.4.0
Rec. Env.	**CPU**	i9-14900K	**Memory**	32GB DDR4@2400MHZ
	OS	Linux v6.4	**HDD**	KINGSTON NV2S1000G

Table 3: The test platform and the recording machine.

USB Class	Device Model	IO Function	# Recorded events			# Events after Lifting
			DMA	MMIO	IRQ	
Stor.	Kingston DTX-32G	Read(sector, 8, buffer)	2230	1784	223	37
		Write(sector, 8, buffer)	2230	1784	223	37
Vid.	Logitech C910 (cam1)	Capture(10, 1080P, buffer)	248088	165392	20674	59
	Off-brand (camera2)	Capture(1, 480P, buffer)	15444	10296	1287	51
Aud.	Generic AB13x USB Audio	Capture(2, 10, S16_LE, 44100, buffer)	110256	73504	9188	51
		Playback(2, 10, S16_LE, 44100, buffer)	55128	36752	4594	51
HID	Dell KB216	Keystroke(50)	500	400	50	21
	Dell MS116	MouseMove(50)	500	400	50	21

Table 4: The USB devices, recorded I/O functions, and number of events in the generated templates. Underlined args are mutable at run time.

5.1 USB devices and I/O functions

We test µUSB with four classes of USB devices which have important use cases in TEE but commonly lack protocol-layer end-to-end encryption: **1) USB Mass Storage:** it extends secure storage from on-chip RPMB [2] to a diverse range of external commodity storage devices, e.g., a thumb drive; **2) USB Video/Audio:** it enables TEE to obtain *trusted* images/audio for provable video authenticity [56]. We choose two distinct cameras with different rates (1080P@5FPS and 480P@30FPS) to demonstrate cross-vendor practicality; **3) USB HID:** it enables TEE to process privacy-sensitive human inputs, e.g., passwords. We choose keyboards and mice as they are the most common HID devices. We record their typical I/O functions and device initialization. The setup and I/O functions are in Table 3 and 4.

5.2 Recording outcome and post analysis

We report the recording outcome for each of the devices in Table 4 in interaction events, including raw recorded events and the events after lifting.

USB Mass Storage. µUSB recorded two SCSI commands: `read(10)` and `write(10)`. For each data transmission round, three interrupts are received, for sending SCSI command, transporting data and receiving status respectively. The number of TRBs scales linearly with transfer data size, which suggests each TRB carries a fixed amount of data.

USB Video/Audio. To configure them, both drivers allocate DMA buffers, encode configurations, and write buffer addresses to xHCI, which is exactly recovered by µUSB, with the buffer symbolized and configurations correctly preserved as invariants, e.g., the audio sampling rate specified in user input (i.e., 44100) is encoded and preserved as 0xbb80 (i.e., 48000). Unsurprisingly, both devices stream raw data with ISOC transfers; without the secure I/O's end-to-end protection, they would have been easily intercepted and exploited by a malicious OS. We are surprised to find camera 2 (480P) requests data buffers of the same size (i.e., 96KB) as camera 1 (1080P), which greatly exceeds its actual needs and is likely an implementation defect. Figure 5 illustrates a fragment of the template for USB Audio playback, where configurations are encoded as invariants, and variants (e.g., buffer addresses) are correctly symbolized.

```
#define S16_LE  0x0001
// mutable input: buffer
void Playback(2, 10, S16_LE,
              44100, void *buffer)
{
    val = read(xhci_base+0x44, 0x4);   //read xHCI
    // matching invariants for correctness
    if (val != 0x0) goto replay_error;
    // write a 4-byte invariant to xHCI
    write(xhci_base+0x0044, 4, 0x8);
    ......
    // alloc DMA
    tmp = dma_alloc(0x3);
    // UAC_SET_CUR
    write(tmp, 3, 0xBB80);  //Resample to 48 kHz
    write(trb_base+0x67f8, 0x8, tmp);
    write(xhci_base+0x2004, 4, 0x1);  //ring doorbell
    ......
    // alloc DMA
    tmp = dma_alloc(0xc0);
    // transfer user data
    write(tmp, 4, buffer);
    write(tmp + 0x4, 4, buffer + 0x4);
    write(tmp + 0x8, 4, buffer + 0x8);
    ...
    // send data to xHCI
    write(trb_base+0x7a50, 0x8, tmp);
    ......
    on_irq(10);          // wait for irq line 10
    ......
}
```

Figure 5: An example of a µUSB template for audio playback, which specifies a sequence of MMIO, DMA, and IRQ events to complete a single isochronous transfer of USB audio. Comments are added for readability.

USB HID. It only uses interrupt transfer for data and is highly deterministic. Exactly one TRB is transferred to report each movement of the mouse and a keystroke. More interestingly, all address/value fields remain identical within one trace. By simply observing its concrete trace, a less-seasoned developer would have treated them as invariants. Yet, µUSB's lifter detects the addresses and symbolizes them as variants.

6 Evaluation

We answer the following questions w.r.t. our goals (§ 2.3):

1. Why is µUSB practical? (§ 6.1)
2. Why do µUSB drivers work correctly and safely? (§ 6.2)
3. What is the performance of µUSB and its drivers? (§ 6.3)

6.1 Analysis of developer efforts

We compare three approaches to implementing the same I/O functions as described in Section 5.

Build from scratch. Developing a USB driver from scratch demands thorough knowledge of the device specification, core functions, configurations, and data structures. We quantify such knowledge required to realize same I/O functions and summarize it in Table 5. For each USB class, we estimate the development typically takes at least several months.

Port. Compared with **Build from scratch**, porting requires less manual effort, yet developers still need to analyze the essential USB functions that implement the desired functionality. From μUSB's recorded call trace, a single USB I/O function involves at least 100 diverse functions from over 10 subsystems, e.g., the device framework, block layer, power and memory management. Porting them and emulating kernel frameworks takes at least 5K SLoC and several months.

μUSB. In contrast, we needed much less developer work. The essential USB-specific knowledge is the xHCI memory layout and interrupts, which took us a few days to familiarize. Building the framework took a couple of weeks, which is a one-time effort. Using our framework is easy, with basic knowledge of USB drivers (e.g., the specific I/O functions and respective arguments): the developer simply prompts μUSB with an input and gets a ready-to-use template; if she demands other I/O functions, she simply re-records, which often takes within a minute. Note that, while **driverlet** [32] also generates drivers, its support for USB is limited to storage (as explained in § 2.2), which took 1-3 days, much longer than our approach.

6.2 Security and correctness analysis

6.2.1 Security Analysis

The key security benefits of μUSB are from removing the large kernel dependencies and to lock USB driver execution on a beaten path inside TEE [52].

Thwarted attacks. μUSB mitigates existing attacks which exploit memory bugs and race conditions of kernel dependencies [71, 75] and USB drivers [68, 70, 73] by design: μUSB does not incorporate kernel dependencies and driver code (§ 4.1); templates accept limited inputs and have no dynamic memory allocation (§ 4.2); μUSB does not allow fine-grained sharing between in-TEE apps (§ 4.3). We show more CVE examples in Table 6. Note that [76] was disclosed *after* μUSB development, which μUSB also defends against as it does not incorporate the code. By confining the execution fully inside the TEE, μUSB drivers also defend against software attacks from the malicious OS through memory [77], interrupt [78], or untrusted RPCs [14]. For logic bugs, e.g., resource exhaustion due to incorrectly handled inputs [8], μUSB mitigates them: 1) incorrect executions will be exposed by mutational recording with user-supplied inputs (§ 4.1); 2) μUSB soft-resets the device for a clean-slate replay (§ 4.3). A corner case is when μUSB might inherit taint-style vulnerabilities encoded in input *constraints* (e.g., integer overflow) which μUSB preserves. However, these are fundamentally difficult to exploit: 1) inputs from in-TEE apps are trusted and limited (1-2), which are validated by the template and audited manually; 2) malformed inputs from devices fail to replay and are immediately rejected; 3) all memory writes (as replayable events) are mediated to ensure they hit valid DMA/MMIO addresses which are non-executable.

Attacks against μUSB. 1) Template fabrication or tampering is unlikely, as the trusted lifter signs them. 2) Attacks from malicious USB devices are defended. They may be mounted against μUSB at both the recording and the replay stage. During recording, they are defended by complementary USB protection measures [40] on the developer machine (§ 4.1). During replay, they may cause the replay to fail due to constant divergence from the template, which can be detected and does not compromise data confidentiality and integrity; they cannot inject malicious payloads (e.g., shellcode) to TEE because the template only supports the pre-recorded I/O function but not other features such as dynamic interface discovery needed by BadUSB-like attacks to emulate HID devices [81]. Exploiting the replayer is also hard, due to its small codebase (500 SLoC) and the simple logic.

	Opcode	Spec.	Conf. Desc.	Data Struct
xHCI	17	645	**Nah**	421
USB Stor.	13	554	3+	50
USB Video	8	169	93+	31
USB Audio	11	239	18+	36
USB HID	8	97	2	3

Table 5: The knowledge required to build a driver from scratch. Conf. Desc. refers to configuration descriptor.

6.2.2 Correctness

We statically vet the templates by cross-checking them with the call traces, drivers, and specifications; we experimentally verify their functional correctness and continuously stress test them for robustness for over two weeks which confirms sustained throughput. An ideal guarantee requires full formal verification of the recordings, which we leave as future work.

USB Mass Storage. We verify the templates encode correct SCSI commands; we generate 1K read/write requests and have verified that 1) they are either rejected by the replayer due to invalid input arguments, or 2) the data read by the μUSB driver match native drivers and that writes reach the storage.

USB Video/Audio. We verify the templates encode correct configurations (e.g., sampling rate, resolutions), following the same control transfers mandated by the respective specification. We dump and manually examine the captured data. For audio, we utilize the μUSB driver to playback the recorded audio to verify record and playback both works correctly; Figure 8 shows their waveforms. For video, we check video frames for correctness (e.g., no missing content, abnormal

µUSB's Design	Example CVEs	Description	Effect	Vuln.
Eliminate kernel dependencies	CVE-2023-53580 (N/A)	Deadlock during driver unbind procedure [67].	Kernel Panic (DoS)	Kernel.A
	CVE-2024-50302 (Med)	Uninitialized memory in HID core dependency [71].	Kernel Info Leak	Kernel.C
	CVE-2024-57981 (Med)	Null pointer dereference in command abort logic [75].	Kernel Panic (DoS)	Kernel.A
No original USB driver code	CVE-2024-26659 (Med)	Incorrect xHC ownership assumption leads to UAF [68].	Memory overwrite	Kernel.I
	CVE-2024-30212 (High)	Unchecked parameter allows arbitrary RAM r/w [70].	Arbitrary mem r/w	Driver.I/C
	CVE-2024-53197 (High)	OOB access in ALSA legacy quirks [73].	Arbitrary mem r/w	Kernel.I
	CVE-2024-57980 (High)	Double-free in uvcvideo status URB handling [74].	Arbitrary code execution	Kernel.I
	CVE-2025-38680 (N/A)	OOB read due to missing len check [76].	Unauthorized mem access	Kernel.I
Isolate normal OS	CVE-2023-34403 (N/A)	Network service confuses OS to access USB files [66].	Info Leak	OS.I
	CVE-2024-27059 (Med)	Malicious device triggers divide-by-zero [69].	Kernel Panic (DoS)	OS.A
	CVE-2024-53150 (High)	Out-of-bounds read in ALSA USB sound driver [72].	Info Leak	OS.I

`I: Integrity; C: Confidentiality; A: Availability`

Table 6: µUSB mitigated common vulnerabilities and exposures (CVEs) in the USB stack by design. Vuln.: vulnerabilities.

color, or jagged edges). To stress test, we streamed video and audio by repeatedly executing `capture` templates and have verified both endured for over 24 hours.

USB HID. We manually verify them. For keyboard, we verify the µUSB driver captures the same key code as we pressed specified in the usage table [100]. For mouse, we verify the coordinates are identical to those reported by native drivers.

The corner case. Despite the above manual verification, one corner case is when the µUSB driver successfully replays an I/O function (i.e., no error codes, and all interactions match the recording) but the data is silently lost. However, this means the recorded driver itself does not implement sufficient driver/device interactions for inferring and handling device states and errors, contrasting our assumption of a *gold* driver (§ 3.1).

6.3 Overhead

6.3.1 Methodology

Our recording machine, test platform, and USB devices are listed in Table 3. We assign one of the two xHCI controllers of the platform to TrustZone and connect the USB devices with the corresponding ports. We reserve 8MB of TEE RAM for DMA allocation.

Benchmarks. We choose benchmarks specific to USB classes and their I/O functions in Table 4. 1) **Storage:** we use FIO, a famous open-source I/O testing tool [11]. We vary transfer block sizes, and report the throughput on 64M of data. 2) **Camera (OneShot/ShortBurst/LongBurst):** we request the cameras to capture 1, 10, 100 image frames at the highest resolution. We report the latency of each request. 3) **Audio (Recording):** We request the microphone to record audio clips from PEASS, a widely used benchmark in audio processing [24], for 10 seconds. We use cosine similarity of MFCC as the measure for audio similarity, a common metric in signal processing [18]. We also report the request latency. 4) **HID (Keyboard/Mouse):** We randomly press the keyboard and move the mouse, and report the latency between receiving an interrupt and obtaining the keystrokes or coordinates.

Comparisons. We compare µUSB drivers with two baselines. 1) **native**: we run the same benchmarks on Linux v6.4 invoking the full-fledged native drivers. 2) **circle**: we run the benchmarks on USB drivers of Circle [93], a baremetal USB driver for Rpi which supports USB Mass Storage and HID, but not Audio/Video. We additionally consider **Driverlet** [32] as it also supports storage and video devices; we use it as a theoretical reference point and do not compare quantitatively, as it is unable to support same USB devices as we do.

6.3.2 µUSB driver performance

Storage. As shown in Figure 6, µUSB storage driver achieves excellent performance. On average, its read/write throughput is 16.26/11.01 MiB/s, close to **native** (17.42/6.68 MiB/s), and higher than **circle** (7% and 35% higher for read, write respectively). µUSB's good performance comes from its simplistic design, which *de-indirects* kernel and has most benefits for CPU-bound workloads (e.g., small random block writes) – both µUSB and **circle** outperform **native** by 5× and 2.5× respectively, where µUSB has better performance due to an even shorter execution path. SSDs have also observed a similar result [113].

Camera. Figure 7(a) and 7(b) show the results. On two distinct cameras, µUSB drivers achieve the lowest per-frame latency, i.e., 2964 and 87 ms, 3% and 20% faster than **native**, respectively. Their performance is consistently close to or better than **native** in **ShortBust** (7.8% better) and **LongBurst** (13.5% better). Noticeably, on camera 2 (480P), the µUSB driver is up to 26% faster than **native** (i.e., **LongBurst**). This is because transmitting lower-resolution images is CPU-bound, which benefits most from µUSB's simplistic design.

Audio. Figure 8 shows audio similarity of two samples and Figure 7(c) shows the latency. As can be seen both visually and quantitatively, µUSB captures almost identical audio with **native** without distortion or packet loss during the 10-second recording. Its latency appears little to no difference with **na-**

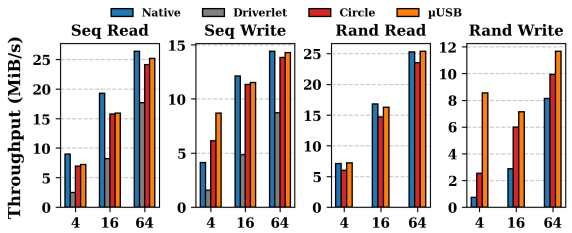
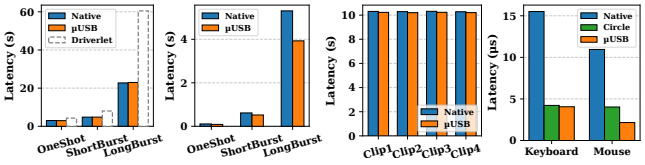


Figure 6: FIO benchmarks for USB Storage. X-axis: transfer block size in KiB. **Driverlet** [32] is kept as a theoretical reference point.



(a) Camera 1 (b) Camera 2 (c) Audio (d) HID

Figure 7: The latency for USB Video, Audio and HID. **Driverlet** [32] is kept as a theoretical reference point.

tive because of a constant audio sampling rate.

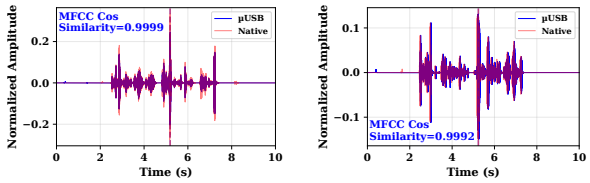
HID. Figure 7(d) shows the results. On average, µUSB is 4.5× faster than **native** and is 1.3× faster than **circle**, which are the best results µUSB have achieved. It is because this class is most latency-sensitive (i.e., µs-level), where each instruction counts. For it, µUSB only executes several (8-12) events in sequence without branches or jumps.

Memory overhead. Table 7 compares the sizes of µUSB driver executables with **native**. µUSB driver sizes range from 15 to 400 KB, which are one to two orders of magnitude smaller than native drivers. µUSB achieves the substantial reduction since it reuses driver/device interactions instead of the driver code and kernel dependencies.

6.3.3 µUSB overhead

End-to-end results. We use one I/O function of each device class in Table 4 and show the template generation time in Figure 9. µUSB is efficient and lightweight: on average, generating a template takes 56.9 seconds, up to 135.1 seconds (**Capture(10s)**). For **Audio** and **Video**, most time is spent on recording which takes 84.9% and 70.6% respectively, since they need to stream data for a fixed duration. **Storage** and **HID** have brief I/O functions thus taking more time to lift, i.e., 44.6% and 54.3% respectively. We highlight the offline analysis does not affect runtime performance of a µUSB driver.

Tracing overhead. The recorder delays USB driver execution by 2.3%, which is negligible. The overhead mainly comes from tracing via KVM: we add 0.03ms to each DMA/MMIO event, and 0.009ms to each interrupt; `ftrace` is fast, as it is static. In comparison, tracing via stage-1 page fault costs



(a) Clip 1 (b) Clip 2

Figure 8: Waveforms of audio clips recorded by µUSB and the native driver. All cosine similarities of MFCC are >0.99.

Dev. Class	Ours	Native	Size Reduction
Storage	24KB	2.5MB	107×
Video	400KB	4.6MB	12×
Audio	160KB	7.1MB	45×
HID	15KB	1.7MB	116×

Table 7: The sizes of µUSB drivers compared to corresponding native USB driver kernel modules.

0.14ms which is 4.5× slower and disrupts USB execution. This significant difference is likely due to a much heavier kernel exception handler than a lightweight one used by the hypervisor, backing our decision for the recorder (§ 4.1).

Static analysis overhead. Symbolic differential trace analysis is fast, whose execution time is linear to the recorded `<var, [val]>` pairs. For **Video** which has the most events (i.e., over 400K), it takes only 3.29 seconds and can be further optimized through SIMD instructions. It effectively detects variants of most I/O functions with just two traces, except for camera 1 which needs six; all are covered by the recorder which mutates at least 10 times (§ 4.1). We also tried continuing mutation, which did not yield more variants. We show the detailed results in Table 8. Qualified taint tracking takes slightly more time. On average, each I/O function takes 11.6 seconds, with **Audio** being the slowest (i.e., 18.14 seconds) due to the longest code path. The results are not surprising, as the lifter only analyzes a *specialized, beaten* kernel code path. In comparison, without the call trace to qualify static analysis, the lifter experiences serious state explosion and does not finish execution within 12 hours, whose intermediate states consume over 64GB of disk space.

6.3.4 End-to-end use case

To showcase the real-world practicality of µUSB, we construct an end-to-end trusted surveillance app inside the TEE, as shown in Figure 10. This app uses the USB devices from Table 3, where each device owns its dedicated ports while coarsely sharing the host controller. Invoking µUSB APIs to select different templates (i.e., `VID_CAPTURE`, `WRITE_64`), this app continuously captures image frames from the camera and dumps them to storage. With µUSB drivers, it runs fully inside TEE to enjoy the strong protection for data confidentiality and integrity, despite the lack of end-to-end data encryption from these USB devices. As we have measured,

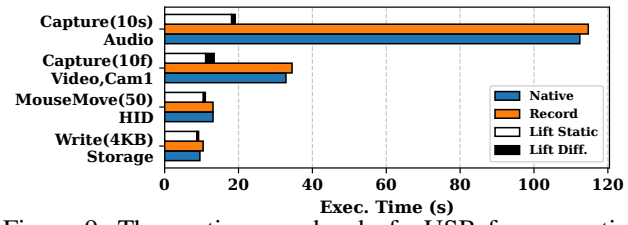


Figure 9: The runtime overhead of µUSB for generating µUSB driver templates.

	Min. # of mut. → 1	2	3	4	5	6	8	10
Capture (10s)	18376	18376	18376	18376	18376	18376	18376	18376
Audio	/165384	/165384	/165384	/165384	/165384	/165384	/165384	/165384
Capture (10f)	41348	41348	41348	41348	41348	**41364**	41364	41364
Video, Cam1	/372132	/372132	/372132	/372132	/372132	/372116	/372116	/372116
Write(4KB)	446	446	446	446	446	446	446	446
Storage	/3568	/3568	/3568	/3568	/3568	/3568	/3568	/3568
MouseMove(50)	100	100	100	100	100	100	100	100
HID	/800	/800	/800	/800	/800	/800	/800	/800

Table 8: Detected variants with more traces after mutation.

µUSB sustains 1.92 FPS on the 1080P camera and 11.6 FPS on the 480P camera for over a day, which achieves comparable performance with native drivers, i.e., 1.94/10.7 FPS on 1080P/480P cameras respectively. The performance is practical since industry webcams often use low (2-15) frame rates to reduce storage costs [108].

7 Related Work

Device drivers. They have been extensively studied [43, 28, 60, 91]. For better security and reliability, pioneering works propose new architectures [27, 98]. µUSB is inspired by them with a focus on USB and reuses mature drivers for derivation. RevNIC [16] and driverlet [32] also derives from mature drivers but both rely on symbolic execution which is impractical to USB; LDR [110] reuses the original driver through partitioning, which only supports simple I/O drivers.

Program analysis. It examines static program structure and or dynamic runtime behavior to detects errors [58, 112, 99]. Among the techniques, fuzzing analyzes programs via semi-random inputs [64, 114]; differential analysis compares behaviors across versions or environment [105, 49]. µUSB builds upon them, and extends them by leveraging kernel I/O/call traces as path qualifier for precision and efficiency, which is a new form a trace-guided analysis [10, 62].

Notably, the techniques can be used to *debloat* software [89, 39, 30, 38] reduces the attack surface by trimming unused driver code while retaining original functionality. However, debloated drivers still depend on large portions of the original implementation, preserving large TCB and vulnerabilities.

Trusted execution environment. It is widely adopted to isolate sensitive operations from hostile OSes [31, 80, 63]. For I/O, existing solutions often rely on normal OS [51, 50] or only support simple sensors [106]. µUSB complements them, providing USB support the TEE critically lacks.

```c
#include <uusb.h> // uUSB API
void trusted_surveillance_app() {
  // 1. Allocate DMA buffer in TEE
  void *buf = dma_malloc(FRAME_SIZE);
  int sector = 0;
  while (true) {
    // 2. Replay frame capturing; buf is mutable arg.
    int ret = uusb_replay(VID_CAPTURE, buf);
    if (ret != SUCCESS) { handle_err(); }
    // 3. Dump to USB Storage
    int blk = FRAME_SIZE/SECTOR_SIZE;
    int rounds = blk/64;
    for (int i = 0; i < rounds; i++) {
      // recorded 64-blk write, with 2 mutable args.
      ret = uusb_replay(WRITE_64,
                        sector, //1st arg
                        buf+i*64*BLK_SIZE); //2nd arg
        if (ret != SUCCESS) { handle_err(); }
    }
    sector += 64;
  }
}
```

Figure 10: An end-to-end trusted surveillance app built atop µUSB inside TrustZone.

Driver synthesis. One may synthesize drivers from scratch and deploy them inside the TEE using DSL [19, 61, 96] and considering execution context interfaces [94, 60, 95]. While synthesis can theoretically yield safe TEE drivers, it requires correct specifications and synthesizers, which is challenging due to the manual effort and error-prone nature of register-level specification development [65]. Notably, µUSB echoes with the pioneering Synthesis OS [88] which collapses kernel layers for flat execution and factors out invariants. Differently, µUSB takes a lightweight recording approach which *ex vivo* synthesizes USB drivers for simplicity and safety.

8 Concluding remarks

We present a novel record, lift, and replay approach to deriving USB drivers for TrustZone. We design and implement µUSB, a system to record USB driver/device interactions with mutations, lift the concrete traces into a template which accepts dynamic inputs while ensuring correctness. We show µUSB drivers have decent performance on USB Storage, Audio, Video and HID. For the first time, µUSB opens a door for TEE software to access complex yet essential USB devices.

Acknowledgments

The authors were supported by Sichuan Science and Technology Plan "Unveiling and Leading" Project (No. 2024YFCY0001) and by CCF-Huawei Populus Grove Fund (No. CCF-HuaweiTC202408). The authors thank the anonymous reviewers and the shepherd for their insightful feedback. The authors also thank Alan Xiangyu Li from Peking University for his helpful comments on µUSB's formal proof.

References

[1]Harmonyos 2 security technical white paper. https://consumer.huawei.com/content/dam/huawei-cbg-site/common/mkt/privacy/overview-new/downen/harmonyos-2-security-technical-white-paper-v2.0.pdf". [Online; accessed 2025-08-20].

[2]optee_os/secure_storage.md at master · op-tee/optee_os. https://github.com/OP-TEE/optee_os/blob/master/documentation/secure_storage.md. (Accessed on 02/15/2019).

[3]Processors - raspberry pi 5 documentation. https://www.raspberrypi.com/documentation/computers/processors.html#bcm2712. [Online; accessed 2025-08-20].

[4]Accessing gpio from ta in raspberry pi 3. https://github.com/OP-TEE/optee_os/issues/1496, 2017. GitHub issue opened 2017-04-25. Accessed: 2025-12.

[5]Ta compilation with spi driver. https://github.com/OP-TEE/optee_os/issues/1461, 2017. GitHub issue opened 2017-04-04. Accessed: 2025-12.

[6]Usb access from pseudo-ta. https://github.com/OP-TEE/optee_os/issues/1499, 2017. GitHub issue opened 2017-04-26. Accessed: 2025-12.

[7]Qualcomm tee and trustzone secure application. https://docs.qualcomm.com/bundle/publicresource/topics/80-88500-4/77_TrustZone_and_secure_application.html, 8 2023. [Online; accessed 2025-08-20].

[8]Allocation of resources without limits or throttling affecting golang-github-openprinting-ipp-usb.src (snyk-centos10-golanggithubopenprintingippusbsrc-13797686). https://security.snyk.io/vuln/SNYK-CENTOS10-GOLANGGITHUBOPENPRINTINGIPPUSBSRC-13797686, 2025. Snyk vulnerability database entry. Published 1 Nov 2025; disclosed 29 Oct 2025; CVE-2025-61723; CWE-770.

[9]Advanced Micro Devices. AMD SEV-TIO: Trusted I/O for secure encrypted virtualization. Technical report, Advanced Micro Devices, 2023.

[10]G. Ammons and J. R. Larus. Improving data-flow analysis with path profiles. In J. W. Davidson, K. D. Cooper, and A. M. Berman, editors, *Proceedings of the ACM SIGPLAN '98 Conference on Programming Language Design and Implementation (PLDI), Montreal, Canada, June 17-19, 1998*, 1998.

[11]Axboe. Flexble i/o testing. https://github.com/axboe/fio.git.

[12]J. Bai, T. Li, K. Lu, and S. Hu. Static detection of unsafe DMA accesses in device drivers. In M. D. Bailey and R. Greenstadt, editors, *30th USENIX Security Symposium, USENIX Security 2021, August 11-13, 2021*, 2021.

[13]O. Burkart, D. Caucal, F. Moller, and B. Steffen. Chapter 9 - verification on infinite structures. In J. Bergstra, A. Ponse, and S. Smolka, editors, *Handbook of Process Algebra*, pages 545–623. Elsevier Science, Amsterdam, 2001.

[14]S. Checkoway and H. Shacham. Iago attacks: why the system call API is a bad untrusted RPC interface. In *Architectural Support for Programming Languages and Operating Systems, ASPLOS '13, Houston, TX, USA - March 16 - 20, 2013*, 2013.

[15]V. Cheval, H. Comon-Lundh, and S. Delaune. Trace equivalence decision: negative tests and non-determinism. In Y. Chen, G. Danezis, and V. Shmatikov, editors, *Proceedings of the 18th ACM Conference on Computer and Communications Security, CCS 2011, Chicago, Illinois, USA, October 17-21, 2011*, 2011.

[16]V. Chipounov and G. Candea. Reverse engineering of binary device drivers with revnic. In C. Morin and G. Muller, editors, *European Conference on Computer Systems, Proceedings of the 5th European conference on Computer systems, EuroSys 2010, Paris, France, April 13-16, 2010*, 2010.

[17]V. Chipounov, V. Kuznetsov, and G. Candea. S2E: a platform for in-vivo multi-path analysis of software systems. In R. Gupta and T. C. Mowry, editors, *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2011, Newport Beach, CA, USA, March 5-11, 2011*, 2011.

[18]A. Chowdhury and A. Ross. Fusing MFCC and LPC features using 1d triplet CNN for speaker recognition in severely degraded audio signals. *IEEE Trans. Inf. Forensics Secur.*, 15:1616–1629, 2020.

[19]C. L. Conway and S. A. Edwards. NDL: a domain-specific language for device drivers. In D. B. Whalley and R. Cytron, editors, *Proceedings of the 2004 ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES'04), Washington, DC, USA, June 11-13, 2004*, 2004.

[20]R. Coombs. Securing the future of authentication with arm trustzone-based trusted execution environment and fast identity online (fido). White paper, ARM Limited, 2015. Accessed: 2025-12-11.

[21]N. Corporation. Nvidia tegra2 family: Technical reference manual. https://developer.nvidia.com/tegra2-reference-manual, 2011. Accessed: 2025-07-09.

[22]Y. Deng, C. Wang, S. Yu, S. Liu, Z. Ning, K. Leach, J. Li, S. Yan, Z. He, J. Cao, and F. Zhang. Strongbox:

A GPU TEE on arm endpoints. In H. Yin, A. Stavrou, C. Cremers, and E. Shi, editors, *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security, CCS 2022, Los Angeles, CA, USA, November 7-11, 2022*, 2022.

[23]G. W. Dunlap, S. T. King, S. Cinar, M. A. Basrai, and P. M. Chen. Revirt: Enabling intrusion analysis through virtual-machine logging and replay. In D. E. Culler and P. Druschel, editors, *5th Symposium on Operating System Design and Implementation (OSDI 2002), Boston, Massachusetts, USA, December 9-11, 2002*, 2002.

[24]V. Emiya, E. Vincent, N. Harlander, and V. Hohmann. Subjective and objective quality assessment of audio source separation. *IEEE Trans. Speech Audio Process.*, 19(7):2046–2057, 2011.

[25]T. Feng, N. DeSalvo, L. Xu, X. Zhao, X. Wang, and W. Shi. Secure session on mobile: An exploration on combining biometric, trustzone, and user behavior. In C. Julien, N. D. Lane, and S. Mishra, editors, *6th International Conference on Mobile Computing, Applications and Services, MobiCASE 2014, Austin, TX, USA, November 6-7, 2014*, 2014.

[26]X. Fu, S. Meng, W. Zhang, L. Guo, K. Sato, D. H. Ahn, I. Laguna, G. L. Lee, and M. Schulz. Distributed order recording techniques for efficient record-and-replay of multi - threaded programs. In *IEEE International Conference on Cluster Computing, CLUSTER 2024, Kobe, Japan, September 24-27, 2024*, 2024.

[27]V. Ganapathy, M. J. Renzelmann, A. Balakrishnan, M. M. Swift, and S. Jha. The design and implementation of microdrivers. In S. J. Eggers and J. R. Larus, editors, *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2008, Seattle, WA, USA, March 1-5, 2008*, 2008.

[28]B. Gerofi, A. Santogidis, D. Martinet, and Y. Ishikawa. Picodriver: fast-path device drivers for multi-kernel operating systems. In M. Zhao, A. Chandra, and L. Ramakrishnan, editors, *Proceedings of the 27th International Symposium on High-Performance Parallel and Distributed Computing, HPDC 2018, Tempe, AZ, USA, June 11-15, 2018*, 2018.

[29]Google. Trusty tee: Uses and examples. Android Security Documentation,https://source.android.com/security/trusty#uses_examples, 2022. Accessed on 02/08/2025.

[30]Z. Gu, W. N. Sumner, Z. Deng, X. Zhang, and D. Xu. DRIP: A framework for purifying trojaned kernel drivers. In *2013 43rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN), Budapest, Hungary, June 24-27, 2013*, 2013.

[31]L. Guan, P. Liu, X. Xing, X. Ge, S. Zhang, M. Yu, and T. Jaeger. Trustshadow: Secure execution of unmodified applications with arm trustzone. In *Proceedings of the 15th Annual International Conference on Mobile Systems, Applications, and Services*, 2017.

[32]L. Guo and F. X. Lin. Minimum viable device drivers for ARM trustzone. In Y. Bromberg, A. Kermarrec, and C. Kozyrakis, editors, *EuroSys '22: Seventeenth European Conference on Computer Systems, Rennes, France, April 5 - 8, 2022*, 2022.

[33]Z. Guo, X. Wang, J. Tang, X. Liu, Z. Xu, M. Wu, M. F. Kaashoek, and Z. Zhang. R2: an application-level kernel for record and replay. In R. Draves and R. van Renesse, editors, *8th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2008, December 8-10, 2008, San Diego, California, USA, Proceedings*, 2008.

[34]M. Guri, M. Monitz, and Y. Elovici. Usbee: Air-gap covert-channel via electromagnetic emission from USB. *CoRR*, abs/1608.08397, 2016.

[35]S. Han and J. Jang. Mytee: Own the trusted execution environment on embedded devices. In *30th Annual Network and Distributed System Security Symposium, NDSS 2023, San Diego, California, USA, February 27 - March 3, 2023*, 2023.

[36]D. M. Hein, J. Winter, and A. Fitzek. Secure block device - secure, flexible, and efficient data storage for ARM trustzone systems. In *2015 IEEE TrustCom/BigDataSE/ISPA, Helsinki, Finland, August 20-22, 2015, Volume 1*, 2015.

[37]L. H. Holley and B. K. Rosen. Qualified data flow problems. In P. W. Abrahams, R. J. Lipton, and S. R. Bourne, editors, *Conference Record of the Seventh Annual ACM Symposium on Principles of Programming Languages, Las Vegas, Nevada, USA, January 1980*, 1980.

[38]Z. Hu and B. Dolan-Gavitt. Irqdebloat: Reducing driver attack surface in embedded devices. In *43rd IEEE Symposium on Security and Privacy, SP 2022, San Francisco, CA, USA, May 22-26, 2022*, 2022.

[39]Z. Hu, S. Lee, and M. Peinado. Hacksaw: Hardware-centric kernel debloating via device inventory and dependency analysis. In W. Meng, C. D. Jensen, C. Cremers, and E. Kirda, editors, *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security, CCS 2023, Copenhagen, Denmark, November 26-30, 2023*, 2023.

[40]O. Inc. Metascan. https://www.opswat.com/products/metascan, 2013. Accessed: 2025-08-20.

[41]Intel Corporation. Intel® extensible host controller interface (xhci) specification. Technical report, Intel Corporation, May 2019. Accessed: 2025-07-09.

[42]J. Jang, M. Kang, and D. Song. Reusb: Replay-guided USB driver fuzzing. In J. A. Calandrino and C. Troncoso, editors, *32nd USENIX Security Symposium, USENIX Security 2023, Anaheim, CA, USA, August 9-11, 2023*, 2023.

[43]A. Kadav and M. M. Swift. Understanding modern device drivers. In T. Harris and M. L. Scott, editors, *Proceedings of the 17th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2012, London, UK, March 3-7, 2012*, 2012.

[44]A. Kantee and J. Cormack. Rump kernels: No os? no problem! *login Usenix Mag.*, 39(5), 2014.

[45]D. Lee, D. Jung, I. T. Fang, C. Tsai, and R. A. Popa. An off-chip attack on hardware enclaves via the memory bus. In S. Capkun and F. Roesner, editors, *29th USENIX Security Symposium, USENIX Security 2020, August 12-14, 2020*, 2020.

[46]D. Lee, D. Kohlbrenner, S. Shinde, K. Asanović, and D. Song. Keystone: An open framework for architecting trusted execution environments. In *Proceedings of the Fifteenth European Conference on Computer Systems*, 2020.

[47]K. Lee and Y. Won. Smart layers and dumb result: IO characterization of an android-based smartphone. In A. Jerraya, L. P. Carloni, F. Maraninchi, and J. Regehr, editors, *Proceedings of the 12th International Conference on Embedded Software, EMSOFT 2012, part of the Eighth Embedded Systems Week, ESWeek 2012, Tampere, Finland, October 7-12, 2012*, 2012.

[48]H. Lefeuvre, D. Chisnall, M. Kogias, and P. Olivier. Towards (really) safe and fast confidential I/O. In M. Schwarzkopf, A. Baumann, and N. Crooks, editors, *Proceedings of the 19th Workshop on Hot Topics in Operating Systems, HOTOS 2023, Providence, RI, USA, June 22-24, 2023*, 2023.

[49]Z. Lei, G. S. Tuncay, B. C. Williem, Z. B. Celik, and A. Bianchi. Scopeverif: Analyzing the security of android's scoped storage via differential analysis. In *32nd Annual Network and Distributed System Security Symposium, NDSS 2025, San Diego, California, USA, February 24-28, 2025*, 2025.

[50]M. Lentz, R. Sen, P. Druschel, and B. Bhattacharjee. Secloak: Arm trustzone-based mobile peripheral control. In *Proceedings of the 16th Annual International Conference on Mobile Systems, Applications, and Services*, 2018.

[51]W. Li, M. Ma, J. Han, Y. Xia, B. Zang, C.-K. Chu, and T. Li. Building trusted path on untrusted device drivers for mobile devices. In *Proceedings of 5th Asia-Pacific Workshop on Systems*, 2014.

[52]Y. Li, B. Dolan-Gavitt, S. Weber, and J. Cappos. Lock-in-pop: Securing privileged operating system kernels by keeping on the beaten path. In D. D. Silva and B. Ford, editors, *Proceedings of the 2017 USENIX Annual Technical Conference, USENIX ATC 2017, Santa Clara, CA, USA, July 12-14, 2017*, 2017.

[53]Y. Li, L. Lei, Y. Wang, J. Jing, and Q. Zhou. Trustsamp: Securing streaming music against multivector attacks on ARM platform. *IEEE Trans. Inf. Forensics Secur.*, 17:1709–1724, 2022.

[54]Linaro. Op-tee: Open portable trusted execution environment. https://www.op-tee.org/, 2017.

[55]X. Liu, X. Zheng, C. Fu, X. Xie, and P. Di. Grayduck: The sword of damocles for duck typing in dynamic language deserialization. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*, 2024.

[56]Y. M. Liu, Z. Yao, M. Chen, A. A. Sani, S. Agarwal, and G. Tsudik. Provcam: A camera module with self-contained TCB for producing verifiable videos. In W. Shi, D. Ganesan, and N. D. Lane, editors, *Proceedings of the 30th Annual International Conference on Mobile Computing and Networking, ACM MobiCom 2024, Washington D.C., DC, USA, November 18-22, 2024*, 2024.

[57]H. Lu, Y. Wu, S. Li, Y. Lin, C. Zhang, and F. Zhang. BADUSB-C: revisiting badusb with type-c. In *IEEE Security and Privacy Workshops, SP Workshops 2021, San Francisco, CA, USA, May 27, 2021*, 2021.

[58]A. Machiry, C. Spensky, J. Corina, N. Stephens, C. Kruegel, and G. Vigna. DR. CHECKER: A soundy analysis for linux kernel drivers. In E. Kirda and T. Ristenpart, editors, *26th USENIX Security Symposium, USENIX Security 2017, Vancouver, BC, Canada, August 16-18, 2017*, 2017.

[59]mCRL2. Labelled transition system. https://mcrl2.org/web/user_manual/fundamentals/labelled_transition_systems.html, 2025. Accessed: 2025-12-02.

[60]A. Menon, S. Schubert, and W. Zwaenepoel. Twindrivers: semi-automatic derivation of fast and safe hypervisor network drivers from guest OS drivers. In M. L. Soffa and M. J. Irwin, editors, *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2009, Washington, DC, USA, March 7-11, 2009*, 2009.

[61]F. Mérillon, L. Réveillère, C. Consel, R. Marlet, and G. Muller. Devil: An IDL for hardware programming. In M. B. Jones and M. F. Kaashoek, editors, *4th Symposium on Operating System Design and Implementation (OSDI*

2000), San Diego, California, USA, October 23-25, 2000, 2000.

[62]J. Ming, D. Wu, J. Wang, G. Xiao, and P. Liu. Straight-taint: decoupled offline symbolic taint analysis. In D. Lo, S. Apel, and S. Khurshid, editors, *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, ASE 2016, Singapore, September 3-7, 2016*, 2016.

[63]F. Mo, A. S. Shamsabadi, K. Katevas, S. Demetriou, I. Leontiadis, A. Cavallaro, and H. Haddadi. Darknetz: towards model privacy at the edge using trusted execution environments. In E. de Lara, I. Mohomed, J. Nieh, and E. M. Belding, editors, *MobiSys '20: The 18th Annual International Conference on Mobile Systems, Applications, and Services, Toronto, Ontario, Canada, June 15-19, 2020*, 2020.

[64]J. Mohan, A. Martinez, S. Ponnapalli, P. Raju, and V. Chidambaram. Finding crash-consistency bugs with bounded black-box crash testing. In A. C. Arpaci-Dusseau and G. Voelker, editors, *13th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2018, Carlsbad, CA, USA, October 8-10, 2018*, 2018.

[65]J. Müller, M. R. Fadiheh, A. L. D. Antón, T. Eisenbarth, D. Stoffel, and W. Kunz. A formal approach to confidentiality verification in socs at the register transfer level. In *58th ACM/IEEE Design Automation Conference, DAC 2021, San Francisco, CA, USA, December 5-9, 2021*, 2021.

[66]National Institute of Standards and Technology (NIST). CVE-2023-34403: Race Condition in Mercedes-Benz NTG6 Head-Unit Allowing Unauthorized File Access via Ethernet and USB Backup. https://nvd.nist.gov/vuln/detail/CVE-2023-34403.

[67]National Institute of Standards and Technology (NIST). CVE-2023-53580: xHCI Deadlock during USB Driver Unbind. https://nvd.nist.gov/vuln/detail/CVE-2023-53580.

[68]National Institute of Standards and Technology (NIST). CVE-2024-26659: Incorrect xHC Ownership Leading to Use-after-Free. https://nvd.nist.gov/vuln/detail/CVE-2024-26659.

[69]National Institute of Standards and Technology (NIST). CVE-2024-27059: Divide-by-Zero Triggered by Malicious USB Device. https://nvd.nist.gov/vuln/detail/CVE-2024-27059.

[70]National Institute of Standards and Technology (NIST). CVE-2024-30212: Unchecked Device Parameters Allow Arbitrary RAM Access. https://nvd.nist.gov/vuln/detail/CVE-2024-30212.

[71]National Institute of Standards and Technology (NIST). CVE-2024-50302: Uninitialized Memory Use in HID Core. https://nvd.nist.gov/vuln/detail/CVE-2024-50302.

[72]National Institute of Standards and Technology (NIST). CVE-2024-53150: OOB Read in ALSA USB Sound Driver. https://nvd.nist.gov/vuln/detail/CVE-2024-53150.

[73]National Institute of Standards and Technology (NIST). CVE-2024-53197: OOB Access in ALSA USB Legacy Quirks. https://nvd.nist.gov/vuln/detail/CVE-2024-53197.

[74]National Institute of Standards and Technology (NIST). CVE-2024-57980: Double-Free in UVC Status URB Handling. httpsvd.nist.gov/vuln/detail/CVE-2024-57980.

[75]National Institute of Standards and Technology (NIST). CVE-2024-57981: Null Pointer Dereference in USB Command Abort Logic. https://nvd.nist.gov/vuln/detail/CVE-2024-57981.

[76]National Institute of Standards and Technology (NIST). CVE-2025-38680: OOB Read in uvc format parsing. https://nvd.nist.gov/vuln/detail/CVE-2025-38680.

[77]National Institute of Standards and Technology (NIST). CVE-2023-32784: Cleartext Transmission of Sensitive Information in KeePass. https://nvd.nist.gov/vuln/detail/CVE-2023-32784, May 2023. National Vulnerability Database (NVD); Patch available in KeePass 2.54+.

[78]National Institute of Standards and Technology (NIST). CVE-2024-53881: NVIDIA vGPU Host Driver Interrupt Storm Vulnerability. https://nvd.nist.gov/vuln/detail/CVE-2024-53881, Jan. 2025. CVSS 3.1 Score: 5.5 (MEDIUM); Patch: NVIDIA vGPU Software Update Required.

[79]N. Nissim, R. Yahalom, and Y. Elovici. Usb-based attacks. *Comput. Secur.*, 70:675–688, 2017.

[80]J. Niu, X. Wen, G. Wu, S. Liu, J. Yu, and Y. Zhang. Achilles: Efficient tee-assisted BFT consensus via rollback resilient recovery. In *Proceedings of the Twentieth European Conference on Computer Systems, EuroSys 2025, Rotterdam, The Netherlands, 30 March 2025 - 3 April 2025*, 2025.

[81]K. Nohl and J. Lell. Badusb - on accessories that turn evil. Blackhat USA 2014, 2014.

[82]C. M. Park, D. Kim, D. V. Sidhwani, A. Fuchs, A. Paul, S. Lee, K. Dantu, and S. Y. Ko. Rushmore: securely displaying static and animated images using trustzone. In S. Banerjee, L. Mottola, and X. Zhou, editors, *MobiSys '21: The 19th Annual International Conference on Mobile Systems, Applications, and Services, Virtual Event, Wisconsin, USA, 24 June - 2 July, 2021*, 2021.

[83]H. Park and F. X. Lin. Gpureplay: a 50-kb GPU stack for client ML. In B. Falsafi, M. Ferdman, S. Lu, and T. F. Wenisch, editors, *ASPLOS '22: 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Lausanne, Switzerland, 28 February 2022 - 4 March 2022*, 2022.

[84]H. Park and F. X. Lin. Safe and practical GPU computation in trustzone. In G. A. D. Luna, L. Querzoni, A. Fedorova, and D. Narayanan, editors, *Proceedings of the Eighteenth European Conference on Computer Systems, EuroSys 2023, Rome, Italy, May 8-12, 2023*, 2023.

[85]H. Park, S. Zhai, L. Lu, and F. X. Lin. Streambox-tz: Secure stream analytics at the edge with trustzone. In D. Malkhi and D. Tsafrir, editors, *2019 USENIX Annual Technical Conference, USENIX ATC 2019, Renton, WA, USA, July 10-12, 2019*, 2019.

[86]H. Peng and M. Payer. Usbfuzz: A framework for fuzzing USB drivers by device emulation. In S. Capkun and F. Roesner, editors, *29th USENIX Security Symposium, USENIX Security 2020, August 12-14, 2020*, 2020.

[87]C. Priebe, D. Muthukumaran, J. Lind, H. Zhu, S. Cui, V. A. Sartakov, and P. Pietzuch. SGX-LKL: Securing the host OS interface for trusted execution. 2020.

[88]C. Pu and H. Massalin. An overview of the synthesis operating system. Technical Report CUCS-470-89, Columbia University, 1989.

[89]A. Quach, R. Erinfolami, D. Demicco, and A. Prakash. A multi-os cross-layer study of bloating in user programs, kernel and managed execution environments. In T. Kim, C. Wang, and D. Wu, editors, *Proceedings of the 2017 Workshop on Forming an Ecosystem Around Software Transformation, FEAST@CCS 2017, Dallas, TX, USA, November 3, 2017*, 2017.

[90]Qualcomm Technologies, Inc. Guard your data with the qualcomm snapdragon mobile platform (rpmb and secure storage). https://tinyurl.com/63yn8k9j, 2022.

[91]M. J. Renzelmann, A. Kadav, and M. M. Swift. Symdrive: Testing drivers without devices. In C. Thekkath and A. Vahdat, editors, *10th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2012, Hollywood, CA, USA, October 8-10, 2012*, 2012.

[92]E. Rescorla. Rfc 8446: The transport layer security (tls) protocol version 1.3, 2018.

[93]rsta2. Circle. https://github.com/rsta2/circle, 2025.

[94]L. Ryzhyk, P. Chubb, I. Kuz, E. L. Sueur, and G. Heiser. Automatic device driver synthesis with termite. In J. N. Matthews and T. E. Anderson, editors, *Proceedings of the 22nd ACM Symposium on Operating Systems Principles 2009, SOSP 2009, Big Sky, Montana, USA, October 11-14, 2009*, 2009.

[95]D. Schwyn, Z. Liu, and T. Roscoe. Efeu: generating efficient, verified, hybrid hardware/software drivers for I2C devices. In *Proceedings of the Twentieth European Conference on Computer Systems, EuroSys 2025, Rotterdam, The Netherlands, 30 March 2025 - 3 April 2025*, 2025.

[96]J. Sun, W. Yuan, M. Kallahalla, and N. Islam. HAIL: a language for easy and correct device access. In W. H. Wolf, editor, *EMSOFT 2005, September 18-22, 2005, Jersey City, NJ, USA, 5th ACM International Conference On Embedded Software, Proceedings*, 2005.

[97]T. Sun, B. Jiang, H. Lin, B. Li, Y. Teng, Y. Gao, and W. Dong. Tensorshield: Safeguarding on-device inference by shielding critical dnn tensors with tee. In *Proceedings of the 2025 ACM SIGSAC Conference on Computer and Communications Security*, 2025.

[98]M. M. Swift, M. Annamalai, B. N. Bershad, and H. M. Levy. Recovering device drivers (awarded best paper!). In E. A. Brewer and P. Chen, editors, *6th Symposium on Operating System Design and Implementation (OSDI 2004), San Francisco, California, USA, December 6-8, 2004*, 2004.

[99]S. M. S. Talebi, H. Tavakoli, H. Zhang, Z. Zhang, A. A. Sani, and Z. Qian. Charm: Facilitating dynamic analysis of device drivers of mobile systems. In W. Enck and A. P. Felt, editors, *27th USENIX Security Symposium, USENIX Security 2018, Baltimore, MD, USA, August 15-17, 2018*, 2018.

[100]USB-IF. Hid usage table. https://www.usb.org/sites/default/files/hut1_6.pdf, 2025.

[101]USB Implementers Forum. Usb-if compliance program update: Deprecation of ehci testing. https://compliance.usb.org/index.asp?Format=Standard&UpdateFile=USBCV, 2019. Accessed: 2025-07-09.

[102]USB Implementers Forum, Inc. Usb1 specification. Technical report, USB Implementers Forum (USB-IF), Dec. 1997. Accessed: 2025-07-09.

[103]USB Implementers Forum, Inc. Usb4 specification. Technical report, USB Implementers Forum (USB-IF), Dec. 2024. Accessed: 2025-07-09.

[104]USB Implementers Forum, Inc. Usb 2.0 specification. Technical report, USB Implementers Forum (USB-IF), June 2025. Accessed: 2025-07-09.

[105]M. Venturini, F. Freda, E. Miotto, M. Conti, and A. Giaretta. Differential area analysis for ransomware: Attacks, countermeasures, and limitations. *IEEE Trans. Dependable Secur. Comput.*, 22(4):3449–3464, 2025.

[106]J. Wang, A. Li, H. Li, C. Lu, and N. Zhang. RT-TEE: real-time system availability for cyber-physical systems using ARM trustzone. In *43rd IEEE Symposium on Security and Privacy, SP 2022, San Francisco, CA, USA, May 22-26, 2022*, 2022.

[107]N. Wei and A. A. Sani. Schrodintext: Strong protection of sensitive textual content of mobile applications. *IEEE Trans. Mob. Comput.*, 21(4):1402–1419, 2022.

[108]T. Xu and F. X. Lin. Video analytics with zero-streaming cameras. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, 2021.

[109]W. Xu, H. Zhu, Y. Zheng, F. Wang, J. Hua, D. Feng, and H. Li. Tonn: An oblivious neural network prediction scheme with semi-honest tee. *IEEE Transactions on Information Forensics and Security*, 2024.

[110]H. Yan, Z. Ling, H. Li, L. Luo, X. Shao, K. Dong, P. Jiang, M. Yang, J. Luo, and X. Fu. LDR: secure and efficient linux driver runtime for embedded TEE systems. In *31st Annual Network and Distributed System Security Symposium, NDSS 2024, San Diego, California, USA, February 26 - March 1, 2024*, 2024.

[111]M. Yan, Y. Shalabi, and J. Torrellas. Replayconfusion: Detecting cache-based covert channel attacks using record and replay. In *49th Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 2016, Taipei, Taiwan, October 15-19, 2016*, 2016.

[112]H. Zhang, W. Chen, Y. Hao, G. Li, Y. Zhai, X. Zou, and Z. Qian. Statically discovering high-order taint style vulnerabilities in OS kernels. In Y. Kim, J. Kim, G. Vigna, and E. Shi, editors, *CCS '21: 2021 ACM SIGSAC Conference on Computer and Communications Security, Virtual Event, Republic of Korea, November 15 - 19, 2021*, 2021.

[113]Y. Zhang, L. P. Arulraj, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. De-indirection for flash-based ssds with nameless writes. In W. J. Bolosky and J. Flinn, editors, *Proceedings of the 10th USENIX conference on File and Storage Technologies, FAST 2012, San Jose, CA, USA, February 14-17, 2012*, 2012.

[114]A. Zhou, H. Huang, and C. Zhang. KRAKEN: program-adaptive parallel fuzzing. *Proc. ACM Softw. Eng.*, 2(ISSTA):274–296, 2025.

A Appendix

We retrofit Labelled Transition System [13, 59] to show μUSB drivers are as correct as a USB driver for a recorded I/O function.

A.1 Definitions

Definition 1: Labelled Transition System. Labelled Transition System (LTS) [13, 59] uses a tuple $(S, A, \rightarrow, s_0)$ to describe the states of system and a set of transitions between those states, where S is a finite set of states, A is a finite set of actions, $\rightarrow$ is a transition relation (i.e., $s \xrightarrow{a} t$ denotes state s transfers to t with action a), and s_0 is the initial state of system.

Definition 2: Trace and Trace Equivalent. We use A^* to describe all possible finite action sequences of A. An element of A^* is a concrete finite action sequence and is called a *trace*, denoted as $\sigma = a_0 \dots a_n$. Formally, for a given LTS $(S, A, \rightarrow, s_0)$, a set of all possible traces from s_0 is:

$$\mathrm{Tr}(s_0) = \{\sigma \in A^* | \exists t \in S, s_0 \xrightarrow{\sigma} t\}$$

Based on the above, given two LTSs, $T = (S, A, \rightarrow, s_0)$, $T' = (S', A', \rightarrow', s_0')$, we say that they are trace equivalent *iff* $\mathrm{Tr}(s_0) = \mathrm{Tr}(s_0')$.

Example. We give the intuition behind trace equivalence in Figure 11. Suppose to configure a USB device, drivers need to read its capability first to decide the exact configuration. While the two LTSes implement the behavior differently, they are trace equivalent, i.e., $\mathrm{Tr}(s_0) = \mathrm{Tr}(s_0')$, containing two traces {read_usb_cap, write_config_a} and {read_usb_cap, write_config_b}.

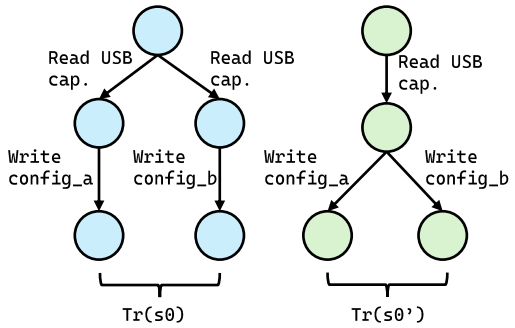


Figure 11: A minimum example of two trace equivalent LTSes.

Definition 3: Minimum Trace. We define state-changing actions as an action set D which when observed by the LTS changes its current state s to t:

$$\forall d \in D,\ \exists s, t \in S.\ s \xrightarrow{d} t \wedge (s \neq t)$$

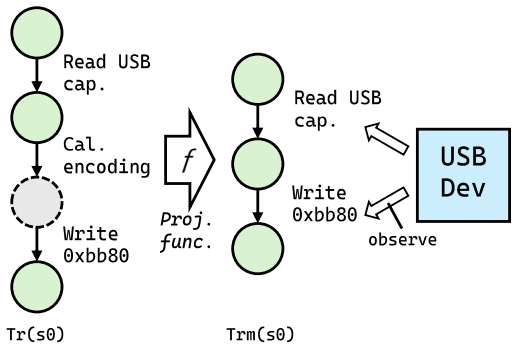


Figure 12: An example of Tr vs. Trm, where calculating encodings are oblivious to a device.

$\mathrm{Trm}(s)$ is minimum trace if all actions are state-changing: $\mathrm{Trm}(s) = \{\sigma \in D^* \mid \exists t \in S,\ s \xrightarrow{\sigma} t\}$. We further define a projection function $f : A^* \rightarrow D^*$, which filters out non-state changing actions of a trace σ. A key corollary is that any trace in Trm is itself after projection:

$$\forall \sigma_m \in \mathrm{Trm}(s_0),\ f(\sigma_m) = \sigma_m$$

The function represents how the device observes actions, where a driver's *internal* actions (e.g., calculating command encodings, allocating slab caches) are oblivious to the device. An example is shown in Figure 12.

A.2 Lemma

With minimum traces, we land on an interesting lemma – if any traces of two LTSes, after projection f, belongs to the other LTS'es minimum trace, then these two LTSes' minimum traces are equivalent. Formally:

$$\begin{aligned} & (\forall \sigma \in \mathrm{Tr}(s_0), f(\sigma) \in \mathrm{Trm}(s_0')) \\ \wedge\ & (\forall \sigma' \in \mathrm{Tr}(s_0'), f(\sigma') \in \mathrm{Trm}(s_0)) \\ \rightarrow\ & \mathrm{Trm}(s_0) = \mathrm{Trm}(s_0') \end{aligned}$$

We prove the lemma by contradiction. Suppose the two minimum traces are **not** equivalent under the same premises, i.e., $\mathrm{Trm}(s_0) \neq \mathrm{Trm}(s_0')$ while $\forall \sigma \in \mathrm{Tr}(s_0), f(\sigma) \in \mathrm{Trm}(s_0')$ (***Pa***) and $\forall \sigma' \in \mathrm{Tr}(s_0'), f(\sigma') \in \mathrm{Trm}(s_0)$ (***Pb***). That is, there exists an outstanding minimum trace σ_x in either Trm:

$$\sigma_x \in \mathrm{Trm}(s_0) \quad \wedge \quad \sigma_x \notin \mathrm{Trm}(s_0')$$

or

$$\sigma_x \in \mathrm{Trm}(s_0') \quad \wedge \quad \sigma_x \notin \mathrm{Trm}(s_0)$$

We first prove the former: as Trm is minimum trace, we have $\sigma_x \in \mathrm{Tr}(s_0)$ and $f(\sigma_x) = \sigma_x$ from corollary. Combined with ***Pa***, we further have $\sigma_x \in \mathrm{Trm}(s_0')$, which apparently contradicts the condition that $\sigma_x \notin \mathrm{Trm}(s_0')$; such a case hence does not exist. By duality, the proof of the latter is symmetric

to the former: we again obtain $\sigma_x \in \mathrm{Tr}(s'_0)$ and $f(\sigma_x) = \sigma_x$ from corollary. By ***Pb***, it follows that $\sigma_x \in \mathrm{Trm}(s_0)$, which also contradicts the condition that $\sigma_x \notin \mathrm{Trm}(s_0)$. Therefore, this case cannot hold either. As both cases do not exist, the predicate $\mathrm{Trm}(s_0) \neq \mathrm{Trm}(s'_0)$ is false and that $\mathrm{Trm}(s_0) = \mathrm{Trm}(s'_0)$ is therefore true ■.

A.3 The Proof

Without loss of generality, we consider the USB device with one single I/O function and initial state s_0. For USB devices, the state-changing actions are low-level driver/device interactions. We thus define $D = \{\mathrm{MMIO}, \mathrm{DMA}, \mathrm{IRQ}\}$. For the native kernel driver, its LTS T has an action set $A = D \cup I$, where I are driver-internal, e.g., slab allocation, encoding calculation. Its trace $\sigma \in \mathrm{Tr}(s_0)$ is mixed with both D and I, We next define the LTS T' for a μUSB driver with action D and trace σ': by recording the low-level interactions of a native driver T, it effectively applies the projection function f to σ and logs $f(\sigma)$; by replaying, it re-executes its own actions σ' which is essentially $f(\sigma')$.

Based on the above, T and T' further have two properties tied to our design.

Property 1. *For the recorded I/O function, T''s converging traces σ' (i.e., after mutational recording) capture sufficient traces σ from T. In other words, the projection of traces σ of T constitutes minimum traces of T'. That is,*

$$\forall \sigma \in \mathrm{Tr}(s_0), f(\sigma) \in \mathrm{Trm}(s'_0)$$

Property 1 is not derived here; it follows from the convergence criterion of § 4.1.2 together with the gold-driver assumption of § 3.1.

Property 2. *A μUSB driver does not make up actions; all σ' replayed by T' come from actual recordings of T for the I/O function which are minimum traces. That is,*

$$\forall \sigma' \in \mathrm{Tr}(s'_0), f(\sigma') \in \mathrm{Trm}(s_0)$$

With the previous lemma, we conclude $\mathrm{Trm}(s_0) = \mathrm{Trm}(s'_0)$■. To understand more intuitively, from the perspective of a USB device, it only observes and reacts to state-changing actions which form *minimum traces*; whether such traces are emitted by a native driver T or a μUSB driver T', they are trace equivalent to the device and equally correct.

Achieving Cloud-Grade SLOs for Local Mixture-of-Experts Inference through CPU–GPU Hybrid Design

Wenxin Wang[1], *Yule Hou*[2], *Yu Ji*[2], *Peng Qu*[1,3], *Youhui Zhang*[1,3]

[1]Tsinghua University
[2]Xingyun Integrated Circuits Co., Ltd.
[3]Beijing National Research Center for Information Science and Technology

Abstract

Local deployment of large Mixture-of-Experts (MoE) models falls short of the service quality achieved in cloud-scale environments, even under low-concurrency workloads. We identify four key gaps in local MoE inference: reliance on capacity-reduced models (quantized, distilled, rerouted), inability to meet 30-second TTFT for long prefills (>12K), sub-baseline decode throughput (<20 tokens/s), and poor concurrency under mixed prefill–decode and batched decode workloads. We present a CPU–GPU hybrid system that achieves cloud-level SLOs on dual-socket commodity CPUs and consumer GPUs by (1) stream-loading prefill (SLP), boosting prefill throughput to 1,200 tokens/s and enabling 32K prompts within 30 seconds; (2) distributed SLP (DSLP) with SmallEP expert parallelism, reaching 1,800 tokens/s and 45K prompts in 30 seconds on two RTX 5090s; (3) intra-node prefill–decode disaggregation with zero-copy shared weights and a dual-batch attention–MoE overlap scheme, sustaining concurrency with <15% latency increase and 50% throughput gains; (4) an AVX-512–optimized FP8 GEMV kernel, enabling native CPU FP8 inference while delivering 4–5× lower CPU latency; and (5) fine-grained CPU parallelism that attains 28 tokens/s on INT4 DeepSeek-V3 and 21.5 tokens/s on intact FP8 V3. Evaluations show our system delivers cloud-level QoS for flagship MoE models on consumer CPU–GPU platforms, reshaping local deployment with intact, original-precision inference and enabling high-quality, cost-effective access without datacenter infrastructure.

1 Introduction

Large-scale Mixture-of-Experts (MoE) architectures have become a central design point for state-of-the-art language models, enabling their parameter counts approaching the trillion scale while sustaining practical training and inference costs through sparse expert activation [17, 28, 38, 44]. This sparsity reshapes the deployment landscape. On one end, massive cloud deployments—often serving tens of thousands of concurrent requests—benefit from the highly overlapped expert activation, which drives hardware utilization toward peak levels [55, 59]. On the other end, local or low-concurrency deployments benefit from the sparse activation from small batch sizes, which significantly cuts down memory traffic, reducing the cost of accessing expert parameters and enabling efficient inference on resource-limited hardware [8, 23].

Significant engineering effort has gone into scaling MoE inference in large cloud environments, where service-level objectives are well established [12, 51] and system designs have converged on robust tradeoffs among throughput, latency, and cost. But these methods won't apply to local scenarios, where the hardware landscape and workload characteristics differ substantially. In contrast, local deployment of MoE models remains underexplored. A promising approach for local deployment is CPU–GPU hybrid inference, which alleviates the constraints of limited local VRAM by storing the majority of model parameters in larger and more affordable system memory.

Prior systems follow the arithmetic-intensity principle: compute-heavy layers—such as attention and shared experts—are assigned to the GPU, whereas lower-intensity, parameter-heavy components—such as routed experts—are offloaded to the CPU. The state-of-the-art system, KTransformers [8], demonstrates that careful CPU–GPU orchestration can yield superior performance. By adopting quantized model and altering routing, KTransformers achieves an acceptable 22 tokens/s decoding throughput on an INT4-quantized, rerouted DeepSeek-V3/R1 model.

However, compared to service quality achievable in cloud-scale systems, current local deployment systems still lag substantially. This gap manifests along several dimensions. (1) **Model quality**. Local deployments typically rely on quantized, distilled, rerouted, or otherwise capacity-reduced models, imposed by tight VRAM and memory-bandwidth budgets. Consequently, users cannot access the full-precision, unmodified MoE models commonly available in cloud services, leading to a pronounced gap in model quality. (2) **Prefill Latency (time-to-first-token, TTFT)**. TTFT is critical

for emerging agent workloads and document QA or summarization. Cloud services target a 30-second TTFT [37]. In contrast, prior local systems—despite using advanced matrix units in high-end CPUs—struggle to meet the 30-second target for 12K-token or longer prefills on the DeepSeek-V3 INT4 model [8]. (3) **Decode Latency (time-per-output-token, TPOT; or tokens-per-second, TPS)**. Cloud systems typically deliver >20 tokens/s per-request decoding as a responsiveness baseline [12, 51]. Prior local systems, even with quantized models, fall short (e.g., 16 tokens/s on DeepSeek-V3 INT4 [8]), leading to noticeably reduced responsiveness relative to cloud deployments. (4) **Concurrency**. Two gaps emerge under concurrency even considering the inherent low concurrency requirements in local deployments. First, prefilling and decoding are often mutually exclusive on local systems, producing either impaired TPOT for decoding under prefill-first scheduling or long TTFT under decode-first scheduling. Second, concurrent decode requests—even at small batch sizes—can substantially degrade per-request latency (TPOT), as expert activations effectively multiply with each additional request.

Throughout the paper, we use these cloud-level targets as local QoS reference points. Our key insight is that local MoE inference should be co-designed around the resources available on commodity nodes: large CPU DRAM capacity and bandwidth, consumer-GPU compute, limited VRAM, and PCIe-class interconnects. Based on this principle, our system reorganizes prefill, decode, parallelism, and concurrency to close the four SLO gaps locally. Our contributions include:

- **Stream-loading prefill.** We introduce a stream-loading prefill (SLP) execution model that overcomes consumer GPU VRAM limits, maximizes GPU utilization. SLP increases prefill throughput to up to **1,200 tokens/s** and supports prefilling 32K-token prompts within a 30-second TTFT budget.
- **Distributed stream-loading prefill.** Building on the SLP model, we develop a parallelism strategy that enables multi-GPU prefilling in local settings, with a SmallEP optimization for expert parallelism tailored to regimes with small expert-parallel sizes. The distributed stream-loaded prefill (DSLP) design delivers over **1,800 tokens/s** prefill throughput and supports prefilling **45K-token** prompts within a 30-second TTFT budget using only two consumer-grade RTX 5090 GPUs—offering better cost efficiency than CPU-prefill approaches relying on high-end CPU matrix units.
- **Concurrency via intra-node prefill–decode disaggregation and dual-batch attention–MoE overlap.** For prefill–decode concurrency, we perform prefill–decode disaggregation within a node atop the SLP execution model, and employ zero-copy, shared model weights, enabling concurrent prefill and decode with minimal interference—less than 15% latency increase. For batched decode concurrency, we introduce a dual-batch attention–MoE parallelism scheme that interleaves attention and MoE computations across two microbatches, increasing system throughput by 50% with limited impact on per-sequence QoS.
- **High-performance FP8 GEMV kernel.** We develop an AVX-512–optimized FP8 GEMV kernel tailored for native FP8 MoE inference, achieving 4–5× lower latency and higher bandwidth utilization compared to existing BLAS libraries. This enables practical original-precision inference on CPUs.
- **Fine-grained CPU Parallelism for MoE Inference** We design fine-grained synchronizations, minimize inter-socket communication and synchronization overheads, and fuse quantization-specific operations on CPUs. This improves decode latency for both full-precision and quantized inference, delivering a 28 tokens/second decode rate on INT4 DeepSeek-R1 model (1.25× improvement over KTransformers) and 21.5 tokens/second on the intact FP8 DeepSeek-R1 model.

We evaluate our system on a dual-socket low-end server-class CPU platform with 1–2 consumer GPUs, against mainstream CPU inference frameworks (KTransformers, llama.cpp, ik_llama.cpp) and state-of-the-art CPU BLAS libraries (OpenBLAS, AOCL-BLAS). Our results demonstrate that such consumer-grade CPU–GPU platforms can serve flagship large MoE models at cloud-level quality of service, at a fraction of the cost of GPU-centric supernode or cluster deployments. We further discuss the expected behavior on lower-end hardware, including single-socket CPUs, different CPU SKUs, and lower-VRAM GPUs in Section 6.

2 Background and Motivation

2.1 Background

Mixture-of-Experts (MoE) architectures introduce conditional computation into transformer models by replacing dense feed-forward blocks with large pools of expert networks [40]. A learned gating mechanism selects a small subset of experts—typically top-k or grouped top-k—for each token, enabling the model to scale its total parameter count without proportionally increasing compute per token. Modern designs such as DeepSeek [13, 28] series augment this with additional shared experts that are always active, providing a common representation path aside token-specific specialization.

To sustain inference throughput for models with large parameter counts, serving systems must deliver both high memory capacity and high effective memory bandwidth. Figure 1 surveys recent models and situates their capacity and bandwidth requirements against contemporary device trends. With

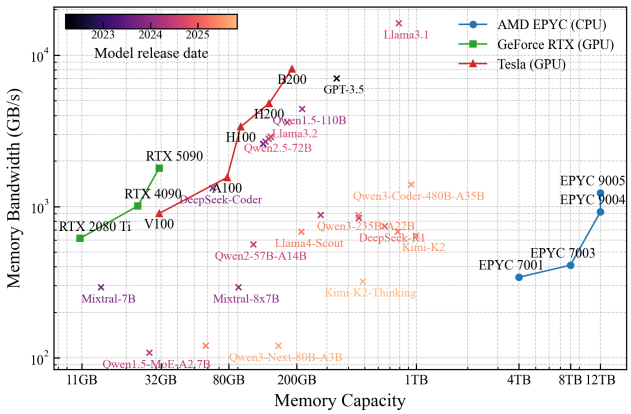


Figure 1: Models' memory demands and device comparison. Model memory capacity reflects parameter size at original weight dtype; memory bandwidth reflects activated weight accesses under a 20 tokens/second service-level objective (i.e., 20 serial activated-weight reads per second). Devices shown include NVIDIA GeForce RTX, NVIDIA Tesla data center GPUs, and AMD EPYC CPUs.

brighter colors indicating more recent releases, a clear trajectory emerges: total parameter capacity continues to climb (approaching 1 trillion parameters, or roughly 1 TB in raw weight size), while activated weights per token are reduced via selective activation. This shift lowers the bandwidth pressure for inference to approximately 1 TB/s, as evidenced by recent large MoE deployments [28, 44].

As Figure 1 illustrates, modern multi-socket CPU systems can deliver aggregate DDR5 bandwidth in the same ballpark as prior GDDR or HBM solutions for the activated-weight access patterns typical of large MoE inference. Combined with substantially lower memory cost (about $3 per GB for DDR5 versus about $30 per GB for HBM) and improved system-level power efficiency, CPU-based deployments become particularly attractive for low-concurrency inference scenarios, where capacity per node and cost/performance at moderate bandwidth are the dominant considerations.

Leveraging modern CPU platforms, hybrid systems such as KTransformers [8] report approximately 16 tokens/second for decode and up to 500 tokens/second for prefill on the DeepSeek-V3 Int4 model—yielding around 20 seconds TTFT for an 8K prompt and an estimated 30+ seconds for 12K or longer. Nonetheless, a substantial gap remains between local inference performance and cloud-level service quality. In the next section, we dissect this gap and its root causes.

2.2 Motivations

Below, we dissect the gap between state-of-the-art local deployment systems and cloud-level service quality across five dimensions—decode, prefill, multi-GPU parallelism, concurrency, and FP8 kernel support—and, based on this analysis, introduce targeted improvements for each. These challenges are not isolated. In local deployment, decode is limited mainly by CPU-side bandwidth efficiency, prefill by how much compute can be exposed under tight VRAM budgets, parallelism by commodity PCIe interconnects, and concurrency by whether these resource pressures can be separated and reorganized rather than forced onto a fixed execution path. This coupling motivates the key principle of our design: local MoE inference should be co-designed around CPU DRAM capacity/bandwidth and consumer-GPU compute, rather than directly inheriting cloud-style assumptions.

2.2.1 Decode

To accommodate limited VRAM in local deployments, hybrid inference systems employ computation offloading [8, 15, 20, 23]: MoE layer weights are persisted in CPU memory and their expert computations execute on the CPU, while densely activated weights and corresponding kernels remain resident and execute on the GPU. Building on this split, KTransformers [8] specifically targets decode-phase bottlenecks—CPU–GPU synchronization and kernel launch overheads—achieving state-of-the-art decode throughput compared to prior local systems such as llama.cpp and Fiddler.

Despite these improvements, decode performance remains short of cloud-level responsiveness: KTransformers delivers approximately 16 tokens/second on the Int4 quantized model, whereas cloud services running full precision models commonly target $\geq$20 tokens/second as a responsive baseline [12, 51]. Decode in local deployment of large MoE models is predominantly memory-bandwidth bound. Using the published figures—74% CPU time and 20.4B activated MoE parameters—KTransformers' decode throughput implies an effective memory bandwidth of roughly 221 GB/s. On the reported dual-socket Xeon DDR5 system, this is only about 50% of the nominal aggregate DDR5 bandwidth. This breakdown highlights significant hardware underutilization and motivates our work to close the gap to cloud-level service quality, as elaborated in Section 3.5.

2.2.2 Prefill

In popular hybrid systems, sparsely activated MoE computations run on the CPU while densely activated components such as attention execute on the GPU. During prefill—where compute intensity rises—computation increasingly dominates end-to-end latency. As sequence length grows beyond roughly 512 tokens, CPU-side MoE execution saturates available compute, and its latency becomes the primary contributor to time-to-first-token (TTFT).

To mitigate CPU limitation, KTransformers leverages Intel's Advanced Matrix Extensions (AMX) instruction set with carefully specialized CPU GEMM kernels, achieving state-of-the-art peak prefill throughput of over 500 tokens/second at a 2048-token prompt, surpassing prior hybrid approaches.

Nonetheless, the approach remains bounded by CPU arithmetic throughput: CPU-centric pipelines cannot match GPU-resident dense GEMM performance. As prompt lengths grow to 8K, 16K and 32K (common in agent, document Q&A and text summary tasks [5, 6]), CPU-side MoE computation becomes the dominant bottleneck, driving time-to-first-token upward to around 20, 40 (estimated) and 100 (estimated) seconds.

Conversely, to exploit GPU compute, weight-offloading systems [4, 18, 19, 42, 43] prefetch model weights to the GPU prior to kernel launches, using predictive scheduling, quantized or compressed weights and reordered pipelines to reduce reactive latency. These designs typically (1) apply a unified optimization strategy to both decode and prefill that prioritizes minimal transfer latency and proactive fetching, and (2) target edge deployments of small- to mid-scale models with lower compute intensity. Consequently, they are optimized for latency rather than throughput, making them ill-suited for compute-heavy prefill on large MoE models and deliver suboptimal prefill performance compared to contemporary computation-offloading approaches.

These limitations motivate a different direction: keep the benefits of weight offloading to avoid CPU-bound compute, but focus explicitly on prefill and optimize for throughput on large MoE models across both weight transfer and computation. In Section 3.1 we introduce a fine-grained stream-loading prefill (SLP) mechanism that pipelines and fully overlaps expert weight transfer with GPU kernel execution at sub-layer granularity. SLP eliminates reactive stalls without prediction overhead, quantization, or rerouting—preserving model quality while sustaining low TTFT at scales.

2.2.3 Multi-GPU Parallelism

To accelerate prefill, large-scale GPU deployments rely on context, sequence, tensor, pipeline, and expert parallelism [7, 22, 26, 29, 37, 41, 53, 55, 57]. Local deployments, however, operate at a different point: low concurrency with strict latency targets, PCIe-only interconnects often lacking GPU–GPU P2P, and constrained VRAM. Under these constraints, many cluster-oriented schemes do not translate directly. We implement a communication-lean hybrid of context and expert parallelism to distribute long-context prefill; yet, even with reduced exchanges, PCIe dispatch/combine remain significant, leaving communication as the primary bottleneck.

Figure 2 illustrates StripedAttention (CP=2, Pass-KV) [7, 58] combined with standard expert parallelism (EP=2) [26, 55]. StripedAttention splits input tokens evenly across two ranks. Each rank first computes causal attention over its local KVs (Attn P1, ①), then performs a ring Pass-KV to exchange the remaining KVs, with which each rank completes the causal attention (Attn P2, ②). In the expert-parallel stage, tokens on each rank are gated and sorted to form per-expert token lists (③). An All-to-All Dispatch exchanges tokens so each GPU receives the tokens assigned to its local experts.

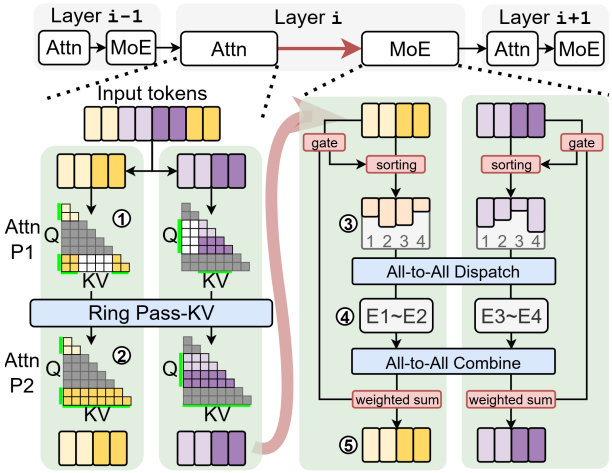


Figure 2: Example workflow: zigzag StripedAttention (Pass-KV, CP=2, left) followed by expert parallelism (EP=2, right). Tokens are split across two ranks; each rank attends locally (phase 1), performs one ring Pass-KV exchange, then completes attention (phase 2). Outputs then flow through MoE gating, dispatch/compute/combine, and weighted aggregation.

Local experts execute on those tokens (④), after which an All-to-All Combine returns the expert outputs for the originating tokens. A final weighted sum produces the per-token outputs (⑤).

Implementing StripedAttention with standard expert parallelism confirms that distributed prefill can effectively reduce computation time. As shown by the top-down microbenchmark of a 20K-token prefill on a single decoder layer in Figure 3, standard CP2+EP2 reduces end-to-end layer latency by 21% over the single-GPU baseline. However, EP still incurs substantial PCIe traffic: Dispatch and Combine account for roughly 31% of the layer time in local deployments without GPU–GPU P2P support. To mitigate this bottleneck, in Section 3.2 we propose SmallEP, which redesigns the expert-parallel exchange and reduces transferred MoE communication volume by approximately 50%. This further lowers end-to-end layer latency by 18% over standard EP in the same 20K-token prefill microbenchmark. With SmallEP, the 2-GPU distributed stream-loading prefill (DSLP) achieves a 1.64× throughput gain over single-GPU SLP.

2.2.4 Concurrency

Local deployments face markedly different concurrency profiles from what is evaluated and optimized for on cloud platforms [25, 27, 41, 56]. Given inherently low parallelism, two common strategies help—but with trade-offs: (1) Chunked prefill [2] partitions long prompts so prefill and decode can interleave, which reduces mutual blocking between a lengthy prefill and concurrent decodes on hybrid CPU–GPU systems. However, segmentation induces frequent kernel transitions and short compute windows, destabilizing per-token

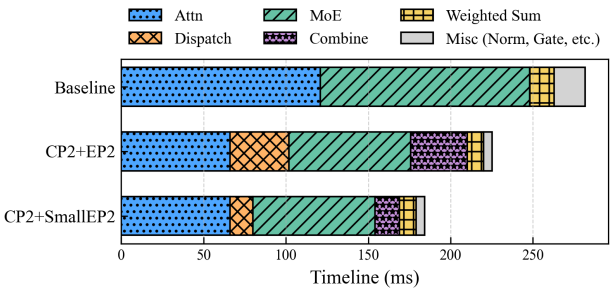


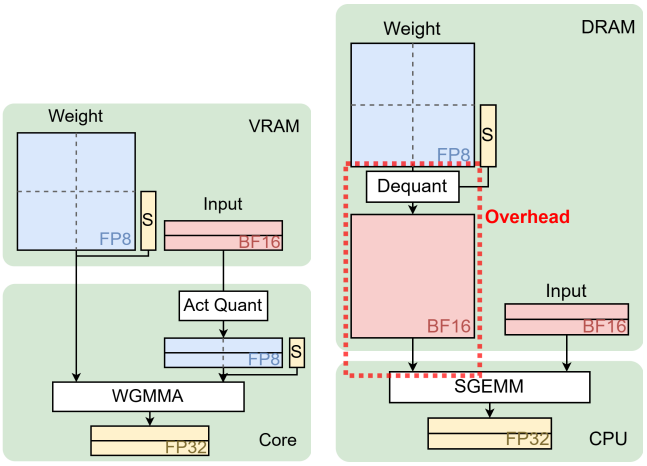
Figure 3: Top-down comparison of parallelism designs on a 20K-token prefill for a single decoder layer (one Attention + one MoE). Baseline: single-GPU sequential. CP2+EP2: StripedAttention (CP=2) with standard expert parallelism (EP=2). CP2+SmallEP2: StripedAttention with our optimized expert parallelism that reduces inter-GPU communication by roughly 50%.

latency (TPOT) for decode and often increasing time-to-first-token (TTFT) for prefill versus isolated execution due to altered scheduling. (2) Batched decode and continuous batching [54] coalesce decode steps across requests to execute jointly. While highly effective in the cloud—large batches saturate expert-activation paths and raise compute utilization—the gains diminish locally. With small batch sizes and sparse expert activation, batching multiplies the number of active experts per step, amplifying memory-bandwidth pressure; consequently, each request experiences worse TPOT compared to non-batched local execution. In Section 3.3 we provide intra-node prefill-decode disaggregation and dual-batch attention-MoE overlap to overcome these two types of contentions.

2.2.5 FP8 Block Scaling Format

Modern models increasingly train and serve with native low-bit formats such as FP8 and FP4 [30, 31, 52], expanding parameter capacity while reducing memory pressure and preserving capability and generality [14, 17, 28, 32, 36, 44]. FP8 typically uses per-tensor or per-block scaling to map BF16 ranges into E4M3 (4-bit exponent, 3-bit mantissa), restoring dynamic range via shared FP32 scales. On modern GPUs, hardware support [34] and optimized kernels [11] enable a straightforward FP8 GEMM path (Figure 4a): FP8 weights and scales are loaded from VRAM, activations are quantized to FP8, and WGMMA executes native FP8 GEMM with FP32 accumulation.

In contrast, CPUs lack native FP8 units; hybrid systems run quantized [48,50] or upcasted BF16 models [47,49] to execute on CPUs [8]. Quantization can degrade precision and stability, while upcasting doubles memory requirements. As shown in Figure 4b, FP8 weights must be dequantized to BF16 before SGEMM, inflating DRAM capacity and memory traffic—the red boxed region marks this dequantization overhead.



(a) Example FP8 GEMM execution path on a modern GPU.

(b) Example FP8 GEMM execution path on a CPU.

Figure 4: FP8 GEMM execution on GPU versus CPU. "S" denotes per-tensor or per-block scale factors. On CPU, FP8 weights and scales are dequantized to BF16 prior to GEMM; the dequantization and doubled memory footprint are highlighted by the red box.

To overcome these limitations and enable original-precision FP8 inference on CPUs, we develop an AVX-512–optimized FP8 GEMV kernel tailored for online FP8 MoE serving, As detailed in Section 3.4. Our FP8 kernel delivers 4–5× lower latency and higher effective memory bandwidth than existing BLAS libraries, making native CPU FP8 inference practical without quantization/dequantization.

3 System Design

Hardware and model target. Our goal is to meet cloud-level SLOs on a single, cost-efficient local node. The system prioritizes price/performance: dual-socket, value-tier server CPUs (e.g., AMD EPYC 9355–class) supplies enough DRAM capacity and bandwidth, while 1–2 consumer GPUs (e.g., RTX 5090–class) provide the bulk of compute at low cost per TFLOP. On the model side, we primarily target intact, original-precision large MoE models whose full weights fit in host DRAM, especially DeepSeek-V3-class models and similar state-of-the-art public MoE models.

3.1 Stream-loading Prefill

To maximize GPU utilization and reduce TTFT, we prefill exclusively on GPUs with a computation–communication overlap design inspired by weight-offloading systems. Unlike prior approaches that rely on modified routing or compression, our design targets overlapping operations and maximizing prefill throughput for intact large MoE models.

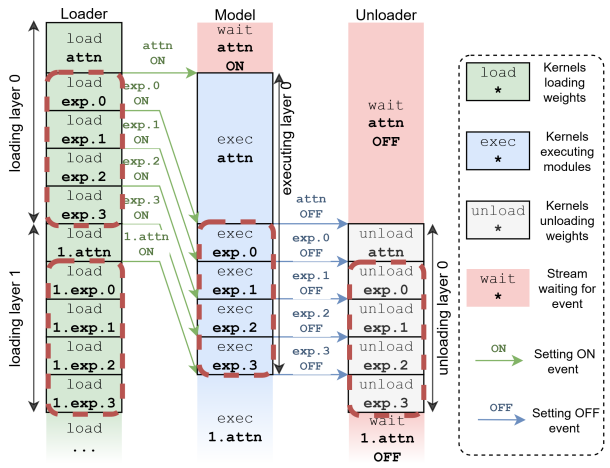


Figure 5: Stream-loading prefill pipeline with three concurrent threads. The loader (left, green) loads weights for modules (attn & exp.i) and sets ON events. The model thread (center, blue) wait ON events, executes modules and set OFF events. The unloader (right, gray) wait OFF events then unloads weights to free GPU memory. Arrows indicate event signaling and dependency ordering. Dark red boxes indicates weights that are managed by experts ring buffer.

Stream-loading prefill scheme. We implement prefill as a three-thread (as well as three CUDA streams) pipeline—loader, model, and unloader—coordinated at a sub-layer granularity, with event tuples that each pairs a CUDA event (device-side) with a host threading event (CPU-side). The loader runs concurrently with the model thread, proactively staging module weights from DRAM into GPU global memory. For each module, the loader signals readiness by setting its ON events, which the model thread waits on before executing that module. After a module finishes, the model thread sets the corresponding OFF events to indicate its weights can be released. The unloader waits on OFF events and, once signaled, evicts the associated weights from GPU memory. This design overlaps weight I/O with computation and minimizes stalls in computation, and bounds VRAM use by actively reclaiming weights after their execution.

Mitigating frequent memory operations. A key engineering challenge is the sheer number of weight tensors. In DeepSeek-V3, 58 MoE layers × 256 experts × 3 FP8 linear modules yields 44.5K tensors; naïvely allocating and freeing them incurs substantial cudaMalloc/cudaFree overhead. We therefore use a manually managed experts ring buffer shared across layers, as marked in dark red boxes in Figure 5. Its length adapts to the compute–communication balance and the peak memory footprint of long-context prefill. For short prompts (transfer-dominated), we set the length to the number of experts of a layer to maximize preload lookahead and saturate host–device bandwidth. Beyond a prompt-length turning point (e.g. 50K), computation becomes the bottleneck and intermediate tensors dominate VRAM; we shrink the length to 2 (ping–pong) to maintain overlap while capping memory use.

Loader and unloader strategy. The loader must decide whether the next module's weights fit in VRAM. For tensors not covered by the ring buffer, we probe available device memory before loading. These tensors are relatively small; a fixed budget of about 1 GB suffices to stage one layer's dense weights. The loader streams weights until that budget would be exceeded, then blocks until the unloader frees space. The unloader follows different policies for different tensor classes. For expert weights managed by the ring buffer, the authoritative copy always remains in host memory, so unloader simply frees or reuses the corresponding GPU-side ring-buffer slot rather than copying the weights back to the CPU. For other tensors, such as the QKV projection weights, which are few in number and irregular in shape and are therefore not managed by the experts ring buffer, the unloader simply releases their temporary GPU allocations after use.

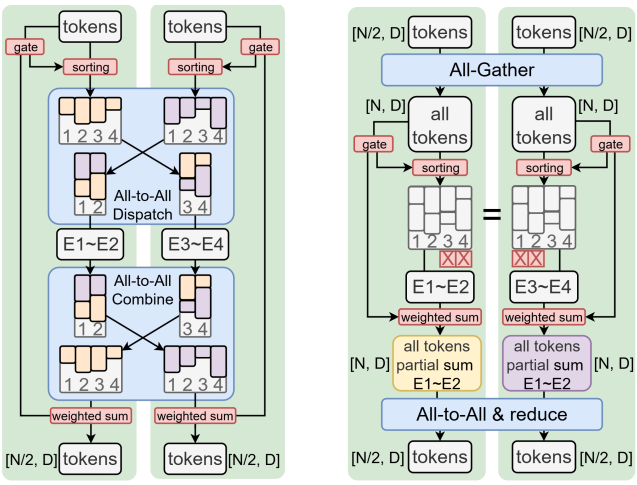
Compared with more implicit memory-management and prefetching mechanisms such as CUDA Unified Virtual Memory and `cudaMemPrefetchAsync`, our design uses explicitly orchestrated loader/model/unloader threads to obtain predictable memory usage and precise control over computation-transfer overlap. This choice does introduce additional engineering complexity, but the added logic is mostly confined to reusable ring-buffer management and synchronization primitives, making the design straightforward to adapt across different large MoE models.

3.2 Distributed Stream-loading Prefill

In few-GPU settings typical of local deployments, we distribute prefill with a combination of context parallelism (CP) and expert parallelism (EP) across GPUs to maximize hardware utilization and shorten time-to-first-token (TTFT); we refer to this distributed version as DSLP. The design targets low parallelism sizes and inferior interconnect regimes by carefully balancing computation and communication, co-optimizing data layout to reduce inter-GPU transfers.

For attention, we adopt the zig-zag variant of striped attention [7, 58], which achieves load balance across GPUs while simplifying implementation.

We optimize EP for small EP sizes to reduce communication overhead on our few-GPU scenario with commodity interconnects. In the standard EP workflow (Figure 6a), tokens are routed by the gate, sorted by expert, and then exchanged via an All-to-All dispatch so each device gathers the tokens for its local expert group. The expected volume communicated per device pair is $\frac{N \cdot A \cdot D}{S}$, where N is the total number of tokens across devices, A is the number of activated experts per token, S is the EP size, and D is the hidden-state size; the



(a) Standard EP workflow.

(b) SmallEP workflow.

Figure 6: Workflow comparison between standard expert parallel and SmallEP optimization under an EP size=2 configuration. N is the number of tokens on all devices, D is hidden size.

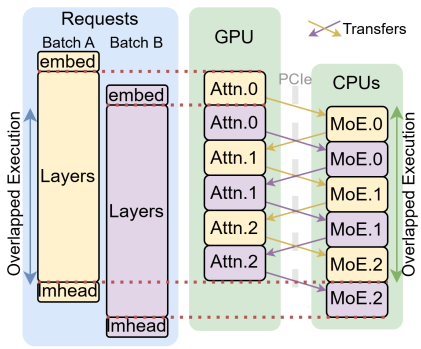


Figure 7: Dual-batch attention-MoE parallelism scheme for simultaneous inference of two batches A and B. "embed" stands for input token embedding linear layer; "lmhead" stands for output language model head linear layer.

combine phase mirrors this cost. When S is large, traffic is evenly spread across links and latency can be effectively overlapped. However, with small S—typical in local deployments without high-bandwidth interconnects like NVLink—the per-link volume becomes concentrated and the transmission cost dominates, as depicted in Figure 3.

SmallEP design. To reduce communication in small EP regimes, we first All-Gather the unsorted tokens so that every rank holds a full copy of all tokens, then perform gate evaluation and token sorting locally on this replicated batch. Each rank retains tokens assigned to its local experts and discards the rest before expert computation. After experts finish, we apply a per-rank partial weighted sum to produce a reduced tensor of shape $[N, D]$, perform an All-to-All on these partial results, and then complete the local reduction to obtain the final output.

This scheme fixes the All-Gather traffic at $N \cdot D$ bytes per link and limits the All-to-All to the partially reduced $N \cdot D$-sized tensors. Compared to standard EP's communication volume $\frac{N \cdot A \cdot D}{S}$, we deduce that when the EP size S is no larger than the number of activated experts A, SmallEP reduces peak per-link bandwidth and improves end-to-end communication efficiency.

This tradeoff is favorable in the small-EP regime we target, especially for long-context prefills over PCIe-class interconnects, where dispatch/combine communication becomes a dominant cost. SmallEP does incur redundant local gating and sorting because each rank processes a full copy of all tokens before discarding non-local expert assignments. However, our profiling shows that this overhead—included in the "misc" component of Figure 3—remains below 10 ms per layer ($< 5\%$ of end-to-end latency), and is therefore small relative to the communication overhead that SmallEP eliminates.

Expert load imbalance is a well-known concern in large-scale EP deployments. However, in our target small-EP regime, each device serves a relatively large set of experts rather than only a few, so routing skew is averaged out more effectively within each device. This makes stragglers from expert imbalance much less severe than in traditional large-EP settings.

3.3 Concurrency-Targeted Optimizations

3.3.1 Dual-Batch Attention-MoE Overlap

Batched decoding presents a fundamental bottleneck on CPU-centric systems due to independent MoE activation patterns. As the number of concurrent requests increases, the set of activated experts expands multiplicatively, creating proportional increases in memory bandwidth demand that quickly saturate available resources.

On the other hand, hybrid CPU-GPU inference reveals a sequential execution pattern where CPU and GPU components alternate but cannot overlap due to compute dependencies. When the CPU processes MoE layers, the GPU remains idle, and vice versa. Our measurements show that attention processing requires approximately 350µs while MoE computation takes 450µs per layer, creating significant idle time that represents wasted computational capacity.

To address both the concurrency limitations and hardware underutilization, we propose a dual-batch interleaving execution scheme. As illustrated in Figure 7, we create two threads as well as two CUDA streams (A and B, execution flow represented by arrows) for concurrent request processing. The execution pattern interleaves two streams of attention (on the GPU) and MoE computation (on the CPU), creating a tightly coupled pipeline that minimizes idle time. The only synchro-

nization bottlenecks occur during GPU-exclusive operations such as token embedding and model head computation, but these contribute minimal overhead (<5ms) compared to the 61 decoder layers that dominate execution time (approximately 50ms per token).

3.3.2 Intra-node Prefill-Decode Disaggregation

Building on stream-loaded prefill, we push prefill compute fully onto the GPU with minimal CPU time consumed. This enables disaggregation within a single node: we dedicate one GPU to prefill while the other handles decoding or chunked prefill, thereby isolating prefill tasks from decode ones, increasing concurrency and reducing contention impacts.

To avoid duplicated storage of weights in DRAM, we share a single resident copy of model weights in our system with the SLP or DSLP pipelines. Inter-process communication uses the experts ring buffer to achieve zero memory copies and minimizes DRAM bandwidth consumption. During disaggregated SLP+decode, Although weights still necessarily traverse the CPU–GPU path over a PCIe 5.0 x16 link, this cost is modest relative to the >1 TB/s DRAM bandwidth and does minimal impact on decode performance.

Intra-node prefill–decode disaggregation, combined with DSLP, expands the scheduling design space beyond plain chunked prefill. Our policy is: use chunked prefill for short requests (e.g., <2K tokens). For long prefill requests, if there are no decode requests within a short window (e.g., 5 minutes), route to DSLP to maximize throughput; otherwise, route to disaggregated SLP to preserve decode latency. Prefill requests are served first-come, first-served, while decode requests are dynamically batched up to a target size (e.g., Y=6) to balance latency and efficiency.

3.4 AVX-512 FP8 GEMV Kernel

At the core of our system lies an FP8 matrix-vector (GEMV) kernel optimized for modern AVX-512 CPUs where modern GPU-style FP8 units are not available. The design targets MoE inference workloads, where extreme throughput and memory efficiency are critical.

3.4.1 FP8 Format and Dequantization Strategy

DeepSeek-V3 adopt the E4M3FN format—1 sign bit, 4-bit exponent, and 3-bit mantissa without NaNs—as the FP8 variant for weight storage [28]. Unlike GPUs, CPUs lack native FP8 processing units [21]. Therefore, we implement E4M3FN dequantization via integer bit manipulations.

We implemented a standard FP8-BF16 dot-product kernel, as illustrated in Figure 8, it requires:

1. Expanding FP8 to FP32 (without fixing exponent bias diff),
2. Applying FP32 scale inverse and exponent bias fix,

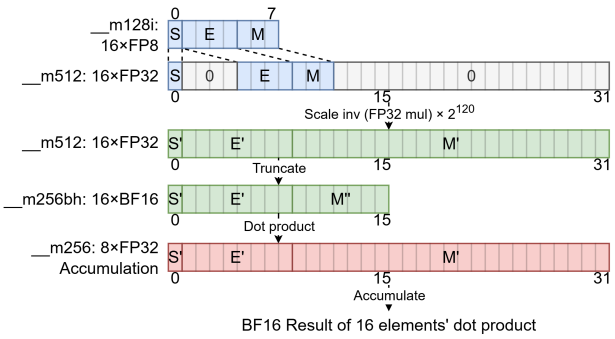


Figure 8: Baseline FP8-BF16 dot-product kernel in AVX512.

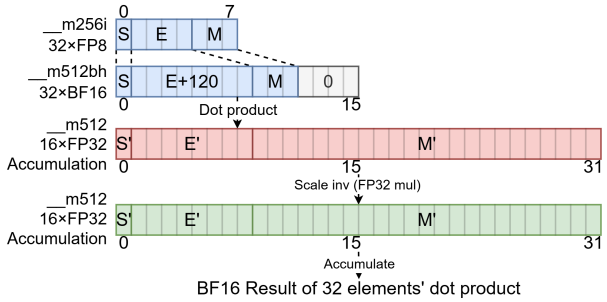


Figure 9: Our post-scaling kernel of FP8-BF16 dot product.

3. Truncating to BF16,
4. Performing BF16 dot-product.

This approach is inspired by the one that is adopted by Chitu engine [45]. Although precise, it involves over 15 vector instructions and suffers from pipeline pressure due to FP32 register density (16 FP32 elements per 512-bit register). On AVX-512, this limits vectorized throughput and results in an estimated latency of ∼36 cycles, and throughput of 12 cycles per iteration (per 16 elements).

Our optimized post-scaling design avoids FP32 in the hot path. As illustrated in Figure 9, we:

1. Expand FP8 weights directly to BF16,
2. Load BF16 activations into `__m512bh`,
3. Execute `vdpbf16ps` for fused BF16-BF16 dot products into FP32 accumulators,
4. Apply post-scaling to the accumulated FP32 result.

This optimized execution path circumvents the register pressure caused by frequent FP32 use. By working directly in BF16 for both weights and activations, we effectively double AVX-512 vector slot utilization—from 16 FP32 to 32 BF16 elements per register—maximizing instruction throughput and widening the execution pipeline. An estimated latency of 27 cycles and throughput of 4.96 cycles per iteration (per 32 elements) are deduced.

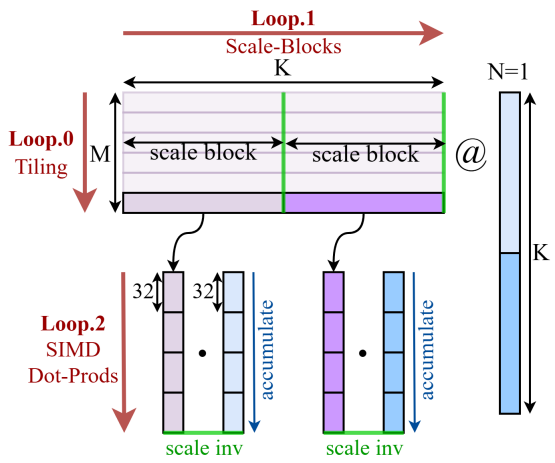


Figure 10: Hierarchical structure of the FP8 GEMV kernel. Computation is organized into **Tiling** along the M-axis, **Scale-Block** iteration along the K-axis (128 elements per block), and **SIMD Dot-Prods** that pipeline accumulations within each block. A single scale factor is loaded per block and applied after all accumulations, reducing scaling overhead and sustaining high bandwidth.

The numerical impact of this transformation is minimal: the end-to-end GEMV operation exhibits only 0.0017 as the 95 percentile of the L1 differences of all elements when compared to PyTorch's reference BF16 GEMV implementation on GPUs. This suggests the BF16 intermediate format is sufficient to preserve inference fidelity for most workloads. Beyond this kernel-level result, Section 4.7 further evaluates end-to-end model quality.

Measured on a our dual-socket AVX-512 CPU platform, the post-scaling kernel achieves 815 GB/s memory bandwidth. Supporting CPU FP8 inference even in the absence of dedicated FP8 hardware.

3.4.2 Tiling and Scale-Block Optimizations

Building upon the post-scaling optimization that lifted bandwidth to 815 GB/s, we further refine the GEMV kernel by restructuring the loop hierarchy. Specifically, we decompose the computation into three nested levels (see Figure 10): (1) *Tiling* (M-axis): the matrix is partitioned into tiles that do not cross scale-block boundaries, ensuring clean alignment between data layout and scaling granularity. (2) *Scale-Blocks* (K-axis): the K dimension is iterated in fixed-size blocks (typically 128 elements), each governed by a single FP8 scale factor. Instead of applying the inverse scale per sub-iteration, we accumulate all partial dot-products first and apply the scale_inv once per block—reducing the number of multiplications by 4× compared to the baseline post-scale kernel. (3) *SIMD Dot-Prods*: within each block, unrolled inner loops perform fused multiply-accumulate operations in SIMD registers, pipelining partial sums before applying the shared scale factor.

This hierarchy minimizes redundant scaling operations while retaining numerical accuracy, maximizing bandwidth utilization. As a result, the optimized kernel sustains up to 947 GB/s memory bandwidth—surpassing both the baseline and prior post-scaling design.

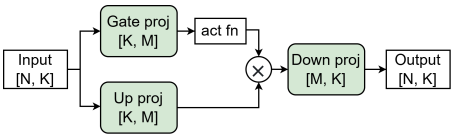


Figure 11: MoE expert compute graph: each expert has gate, up, and down projections. Gate and up can run concurrently; their outputs synchronize before down. N is batch size (N=1 for single-request inference).

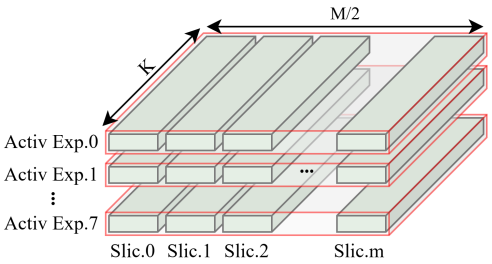


Figure 12: Task partitioning of gate/up on a NUMA node. Output dim M is sliced into m segments per expert, yielding m×8 independent tasks (8 activated experts). Each thread runs a vectorized FP8–BF16 kernel. Red borders mark per-expert barriers that satisfy dependencies for down projection while minimizing synchronization.

3.5 Fine-grained CPU Parallelism for MoE Inference

3.5.1 Fine-Grained Synchronization Design

In MoE layers [28, 38], each expert comprises gate, up, and down projections with the dependencies in Figure 11. Gate and up are embarrassingly parallel and execute concurrently; their results synchronize before down.

In KTransformers' CPU backend, stage 1 computes gate and up. As shown in Figure 12, each GEMV is sliced into m segments along the output dimension M, deepening tensor parallelism on the gather axis, while activated experts are evenly distributed across cores within each NUMA node. Because gate and up must be combined (activations and multiplications) before down, naive global barriers could be adopted but are costly. Instead, we use fine-grained per-expert barriers and fuse the combine ops into stage one, cutting synchronization overhead and improving scalability.

3.5.2 Operator Fusion for Quantized Inference

In quantized MoE models, expert weights (gate, up, down) use low-bit formats (e.g., Q4_K_M, a mixed blockwise GGUF quantization format with per-block scales/metadata). In the KTransformers configuration we study, `Q4_K_M` is a mixed K-quant storage scheme: `gate_proj` and `up_proj` use `Q4_K`, while `down_proj` uses `Q6_K` [3]. The expert inference path can be summarized as

$$\text{input conversion:} \quad \texttt{hidden_type} \rightarrow \texttt{Q8_K},$$

$$\text{gate/up:} \quad \texttt{Q4_K} \times \texttt{Q8_K} \rightarrow \texttt{F32},$$

$$\text{activation and elementwise multiply:} \quad \texttt{SiLU}(\cdot) \odot (\cdot) \rightarrow \texttt{F32},$$

$$\text{down-input conversion:} \quad \texttt{F32} \rightarrow \texttt{Q8_K},$$

$$\text{down:} \quad \texttt{Q6_K} \times \texttt{Q8_K} \rightarrow \texttt{F32},$$

followed by router-weighted accumulation in `F32` and conversion back to `hidden_type`.

This requires converting activations among FP32, Q8_0, and BF16 to preserve precision for SiLU and expert aggregation. We fuse these conversions into surrounding kernels: input conversions for gate/up and down are fused into the parallel gate/up projections (stage 1); output conversion is fused into down (stage 2). The result reduces launch overhead and limits synchronization to two global barriers per MoE layer—after gate/up combination and after down.

4 Evaluations

Our evaluation is organized compositionally around the targeted deployment setting. The claimed SLOs apply to flagship MoE models on the dual-socket CPU plus 1–2 consumer-GPU class of hardware described in Section 4.1. We then isolate the major components of the system: Section 4.2 evaluates SLP/DSLP and SmallEP for long-context prefill; Sections 4.3 and 4.4 evaluate fine-grained CPU parallelism and dual-batch attention–MoE overlap for full-precision and quantized decode; and Section 4.5 shows how these components interact under concurrent requests in a single-node deployment. Finally, Section 4.6 isolates FP8 kernel performance, while Section 4.7 reports end-to-end model quality.

4.1 Evaluation Platform Setup

Hardware Configuration Overview Our primary evaluation platform is a hybrid inference node built for cost-effective large-scale model serving. Our platform is equipped with two AMD EPYC 9355 processors, 24 DDR5-6400 48GB memory modules distributed across 24 memory channels, yielding a total system memory capacity of 1.15TB and a theoretical aggregate bandwidth of 1228GB/s. Complementing the CPU is two NVIDIA RTX 5090 GPUs with 32 GB of GDDR memory and up to 419 FP8 TFLOPS (with FP32 accumulation). Unless otherwise noted, all experimental results for both our engine and KTransformers are measured on this platform; we also cite KTransformers' published numbers when comparing against their original reported setup. Our empirical evaluation therefore primarily covers this dual-socket, PCIe-attached commodity CPU–GPU node. For lower-end or otherwise different configurations, such as reduced effective DRAM bandwidth or lower-bandwidth PCIe interconnects, we provide discussion in Section 6.

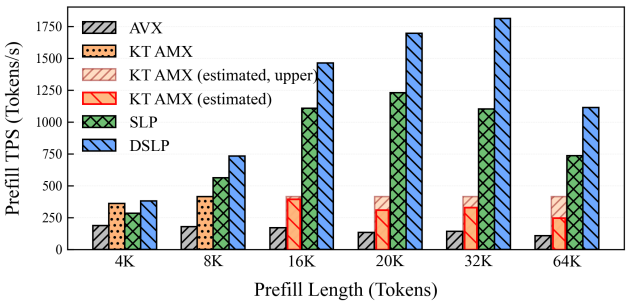


Figure 13: Prefill throughput (tokens/s) for different configurations. We compare CPU AVX baseline (FP8), KTransformers (KT) AMX (INT4), Stream-Loading Prefill (SLP, FP8), and 2-GPU Distributed SLP (DSLP, FP8). KT AMX values for prefill lengths >8K are shown as estimates: "estimated" is obtained by linearly scaling the AVX baseline based on the ratio observed at $\leq$8K tokens. "estimated, upper" is an optimistic projection of the measured KT AMX results by assuming both attention time and MoE time scale linearly with prefill length.

Model Benchmarks Our primary benchmark target is the full-precision FP8 DeepSeek-R1 [17] and Kimi-K2 [44] model, with the Q4_K_M quantized DeepSeek-R1 [48] evaluated alongside them to contextualize the tradeoffs and benefits of our optimizations.

4.2 Prefill Performance

Figure 13 reports prefill throughput as a function of sequence length for different configurations. On our platform, we evaluate our engine on the original FP8 model with three prefill modes: baseline CPU prefill (AVX), Stream-Loading Prefill (SLP), and its 2-GPU distributed variant (DSLP). For comparison, we also cite KTransformers' published AMX-based prefill results (KT AMX), which were obtained on their own platform rather than ours. Across the long-context regime (8K–64K), SLP and DSLP substantially outperform both AVX and KT AMX. For short contexts (4K), weight transfer dominates SLP and DSLP latency, thus no dominant advantage over KT AMX. As the prefill length increases into the long-context regime (16–64K), the advantage of SLP be-

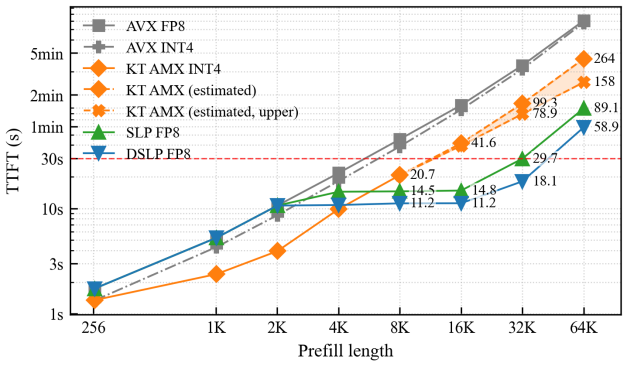


Figure 14: Time-to-First-Token (TTFT) comparison across prefill lengths from 256 to 64K tokens. We report CPU baselines using AVX FP8 and AVX INT4, KTransformers on AMX (KT AMX INT4), and our Stream-Loading Prefill (SLP, FP8) and 2-GPU Distributed SLP (DSLP, FP8). KT AMX results beyond 8K tokens are shown as estimates.

comes dominant. At 20–32K tokens, SLP delivers more than an order-of-magnitude speedup over AVX and a 2.8× improvement relative to the estimated KT AMX performance, while DSLP further increases throughput to 1.64× SLP. The "estimated upper" KT AMX bars, obtained by linearly scaling the measured KT AVX throughput, remain far below SLP and DSLP, implying that even an ideal AMX implementation that scales linearly with sequence length cannot match the gains from our SLP and DSLP.

For a more comprehensive comparison, we evaluate TTFT across a wider range of prefill lengths. Because SLP and DSLP incur additional latency from weight transfer and stream setup, they are disadvantaged at very short prefixes. Consequently, we use the AVX FP8 kernel for requests up to 4K tokens and switch to SLP/DSLP for longer contexts.

At small prefill lengths (256–2K), the CPU baselines are competitive: AVX FP8 and AVX INT4 track each other closely, and KT AMX INT4 achieves the lowest TTFT. Beyond 4K tokens, offloading with stream loading becomes clearly preferable. All CPU-only configurations quickly exceed the 30 s TTFT threshold, while SLP and DSLP grow much more slowly. At 4K–32K tokens, SLP and DSLP keep TTFT below the 30 s target. For very long contexts (32–64K), SLP and especially DSLP remain much closer to practical latency budgets than KT-AMX configuration.

4.3 Full-Precision Decode Performance

As shown in Figure 15, on our evaluation platform, the 671B-parameter DeepSeek-R1 model sustains 21.5 tokens/s on short prompts and maintains ∼20 tokens/s up to 32K context length. We observe similar performance with the Kimi-K2 model, which achieves comparable single-stream throughput of 22.4 tokens/s. With two equal-length, concurrent requests under

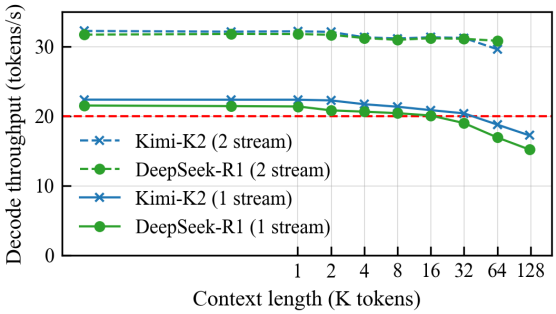


Figure 15: Decode throughput of the full-precision FP8 DeepSeek-R1 671B and Kimi-K2 1T model under single-stream and dual-stream execution across varying context lengths.

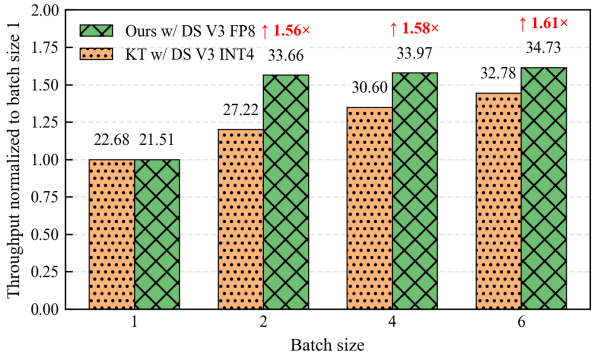


Figure 16: Decode throughput comparison of our engine running DeepSeek-R1 FP8 versus KTransformers (KT) running Q4_K_M (INT4) across batch sizes of 1, 2, 4, and 6. Bars are normalized to each engine's batch size 1 result; absolute system throughputs (tokens/s) are annotated. Red markers report our throughput gains relative to batch size 1.

our dual-batch parallel execution, total system throughput rises to 33.6 tokens/s and remains 31.1 tokens/s at 32K, ensuring each request still achieves ∼16 tokens/s. These results show that our design not only meets the single-stream QoS target but also scales gracefully under concurrency.

Figure 16 compares decode throughput as a function of batch size for our engine running DeepSeek-R1 FP8 and KTransformers (KT) with Q4_K_M. Normalizing to batch size 1 highlights scaling efficiency: our engine achieves 1.56×, 1.58×, and 1.61× throughput at batch sizes 2, 4, and 6 respectively, while KT scales more modestly.

Per-request decode speed, computed as system throughput divided by batch size, shows our dual-batch Attention-MoE parallelism preserves user-level fluency under small batches: at batch size 2, our engine sustains 16.8 tok/s per request, a 21.7% reduction from batch size 1, whereas KT drops from 22.7 to 13.6 tok/s (-40.0%). This narrower per-request degra-

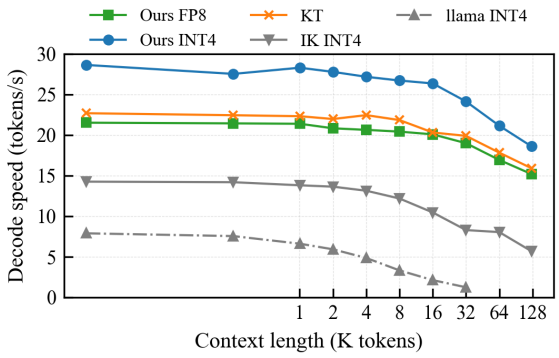


Figure 17: Decode speed comparison including Q4_K_M quantized DeepSeek-R1 671B model. IK: `ik_llama.cpp`; llama: `llama.cpp`.

dation indicates our design delivers better per-user responsiveness and higher aggregate throughput as concurrent requests increase.

4.4 Quantized Model Decode Performance

To contextualize our inference engine's performance and validate our optimizations on mixed-precision inference, we also benchmarked the Q4_K_M variant of the DeepSeek-R1. While lower in numerical precision and sacrifices model fidelity, Q4_K_M quantized DeepSeek-R1 requires only 404GB of RAM to run, displaying its role in resource-constrained deployment scenarios.

We conducted a head-to-head throughput comparison across varying context lengths from 1K to 128K tokens, evaluating: (1) Our engine, (2) llama.cpp, (3) ik_llama.cpp, (4) KTransformers.

As shown in Fig. 17, our design consistently leads across all context lengths. At short sequences (1K–8K), it sustains 28 tokens/s, nearly doubling the throughput of ik_llama.cpp (14 tokens/s) and outperforms KTransformers (22 tokens/s) by 25%. As the context grows, competing engines degrade sharply: KTransformers drops to around 15 tokens/s at 128K. In contrast, our engine declines gracefully, maintaining 19 tokens/s at the maximum length. KTransformers on INT4 quantized model reaches similar decode speed to our engine running the original FP8 model. These results highlight the impact of our tailored CPU backend and parallelism design.

4.5 Performance Under Concurrent Requests

We evaluate concurrency in four representative scenarios to measure engine performance under concurrent prefill–decode and batched decode workloads. For isolation, we construct requests that each lasts about 20 seconds when running alone. Considering the different prefill throughput of our disaggregated SLP and KT on the AVX platform, we set the prefill length to 4K tokens for KT and 24K tokens for our engine, with no shared prefixes and only a few decoded tokens. Decode requests use short 10-token prompts and are forced to generate 480 tokens. Under these settings, when requests arrive simultaneously (regardless of whether they are prefill or decode), their execution could largely overlaps in time, allowing us to minimize variability and focus on end-to-end latency metrics.

Workloads consist of: (1) a single prefill request (1P), (2) a single decode request (1D) (these two serve as baselines), (3) one prefill request plus one decode request (1P+1D), (4) two concurrent decode requests (2D), and (5) one prefill request plus two decode requests (1P+2D). Each workload is executed multiple times on each engine. For metrics, we report the average TTFT for prefill requests and the average TPOT for decode requests, and we also measure the total latency to complete all requests within each workload.

We summarize the results in three perspectives: prefill latency (TTFT), decode latency (TPOT), and total workload latency, as shown in Figure 18. Overall, our engine consistently handles concurrent workloads more robustly than KT. From the prefill perspective, TTFT remains stable for both engines under 1P and 1P+1D. For our engine, this is because prefill–decode disaggregation isolates the 1P request from concurrent decodes. Under 1P+2D, however, the additional DRAM traffic introduced by Attention–MoE parallelism slightly delays Stream-Loading Prefill's weight transfers, leading to an 18% TTFT increase. We expect this effect to become slightly more pronounced as the number of concurrent decode requests further increases, because denser decode-side DRAM accesses leave less bandwidth headroom for SLP weight streaming. For KT, TTFT behavior is governed by its scheduling strategy of executing one 256-token chunked prefill per decode step, effectively prioritizing prefill over decode; this tradeoff becomes clearer when we examine TPOT.

From the decode perspective, our engine maintains stable TPOT across all workloads, while KT's TPOT degrades significantly whenever prefill and decode coexist. KT's average TPOT increases by up to 2.45× over the 1D baseline, whereas ours increases by at most 1.54× and scales smoothly under various concurrency workloads.

Finally, in terms of total latency to complete all requests in each workload, our engine is consistently better than KT: our worst-case latency increase is 1.67× over the baseline, compared to KT's 2.53×, and we also exhibit more favorable scaling when serving multiple concurrent decode requests (2D and 1P+2D).

4.6 FP8 GEMV Kernel Performance

To isolate kernel performance from end-to-end system effects, we evaluate our FP8 GEMV kernels under real MoE workload and compare them against established CPU libraries. As shown in Table 1, our optimized kernel achieves a latency of

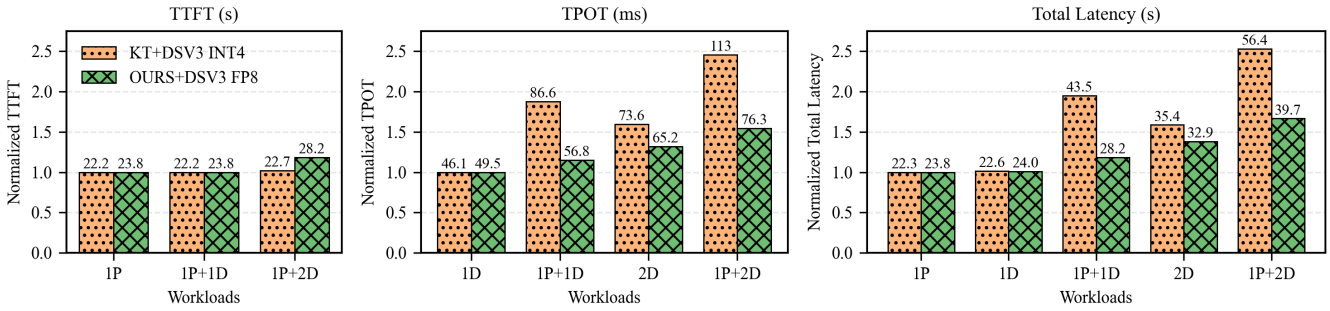


Figure 18: Microbenchmark of system latency under concurrent workloads. We compare KT with INT4 DeepSeek-R1 against our engine with FP8 DeepSeek-R1. All results are normalized to the first bar group in each subfigure, with the corresponding absolute values annotated on top of the bars. Workload labels: "1P" denotes one prefill request, while "1D" and "2D" denote one and two concurrent decode requests, respectively.

Table 1: Comparison of GEMV kernels on MoE workload. Our implementation is the only one supporting FP8 natively, while remaining competitive in bandwidth and significantly reducing latency.

Library	Format	Latency (μs)	BW (GB/s)
OpenBLAS	FP32	69.5	844.42
AOCL-BLAS	FP32	59.2	992.70
AOCL-BLAS	BF16	69.5	422.26
Ours (Std.)	FP8	21.7	678.0
Ours (Optim.)	FP8	**15.5**	**947.1**

15.5 µs and bandwidth utilization of 947 GB/s — surpassing the our standard kernel (21.7 µs, 678 GB/s) and significantly outperforming OpenBLAS [35], and AOCL-BLAS [1] across FP32 and BF16 formats in latency while matching or exceeding bandwidth utilization. These results demonstrate that our kernels unlocks 4–5× latency reduction over existing CPU libraries with native FP8 support and delicate design.

4.7 End-to-End Quality Evaluation

We evaluate end-to-end model quality on benchmark tasks. Table 2 reports Exact-Match results on MMLU-Redux [16] and MMLU-Pro [46] for our deployments of DeepSeek-V3.1 [28] and Kimi-K2-Instruct [3], compared against the officially published results of these models.

Overall, our system preserves end-to-end quality well: the measured scores are close to the official references, with degradations within about 1–1.2 percentage points across the evaluated benchmarks. We believe these differences are minor and likely stem from small mismatches in generation configuration, rather than from a systematic loss of model capability introduced by our system. These results therefore provide additional evidence that our design preserves the quality of the original models while improving local serving performance.

Table 2: End-to-end quality evaluation using Exact Match. Official results are taken from the corresponding model releases.

Model	Benchmark	Ours	Official
DeepSeek-V3.1	MMLU-Redux	91.00	91.8
	MMLU-Pro	83.41	83.7
Kimi-K2-Instruct	MMLU-Redux	91.53	92.7
	MMLU-Pro	80.00	81.1

5 Related Work

Cloud-service inference systems GPU-centric engines such as vLLM, SGLang and TensorRT-LLM are optimized for multi-GPU nodes and datacenters to sustain high concurrency and throughput [25,33,56]. They converge on KV cache management (paged or radix-indexed) [25,56], concurrency-aware scheduling (Chunked Prefill, Continuous Batching) [2,54], distributed inference across context, sequence, tensor, pipeline, and expert parallelism [22,26,29,37,41,53,55,57], and operator-level optimizations (kernel fusion, FlashAttention variants, new precisions) [9, 10, 39, 45]. While highly effective, these designs assume abundant GPU memory bandwidth, large aggregated VRAM, and fast interconnects, limiting flexibility and cost-efficiency outside GPU-first environments. Our work targets the local regime, adapting MoE-specific execution and parallelism to small-EP, low-concurrency settings on commodity CPU–GPU platforms while retaining cloud-level QoS.

Local inference systems for MoE models Mainstream frameworks [25,33,56] prioritize datacenters and are impractical under constrained VRAM and bandwidth. Consumer-first solutions like Llama.cpp [15], ik_llama.cpp [20], and KTransformers [8] enable CPU–GPU hybrid serving; KTransformers combines quantization, rerouting, advanced CPU instruc-

tions, and NUMA-aware orchestration to deliver competitive throughput on commodity platforms. We advance this trajectory by achieving cloud-level SLOs for intact, original-precision MoE models on consumer CPU–GPU systems. Compared with KTransformers, our work differs most clearly in target of achieving cloud-grade SLOs rather than enabling inference, and technically in three places: long-context prefill is made explicitly GPU-centric through stream loading instead of remaining CPU-centric; expert parallelism is redesigned for small EP sizes on PCIe-only local nodes; and concurrency is improved through intra-node prefill/decode disaggregation and dual-batch overlap rather than only through chunking and scheduling.

Expert offloading techniques. Serial expert-processing approaches such as ES-MoE [24] improve efficiency in memory-constrained settings by executing experts sequentially, avoiding zero-padding and large dispatch masks while improving load balance. However, ES-MoE differs from our target and design in three aspects. First, it targets training, where offloading and sequential processing enable larger microbatches and higher throughput, whereas our work targets latency-critical local inference, especially low-concurrency, long-context prefilling where TTFT is the primary objective. Second, ES-MoE plans expert transfer based on routing decisions, but this routing-dependent synchronization point leaves less opportunity to overlap transfer with the substantial pre-MoE computation on the critical path, which accounts for more than 40% of per-layer execution time on a 20K-token prefill. In contrast, our SLP strategy stream-loads expert weights at a finer granularity and overlaps transfer with dense-module execution as well as expert computation. Third, for multi-GPU local prefilling, we further introduce SmallEP to reduce communication overhead under small EP sizes, improving distributed-prefill TTFT in PCIe-class local deployments.

6 Discussion

Lower-end CPUs While prefill is largely addressed by our stream-loading prefill design, the main impact of lower-end CPUs is on decode. Small-batch decoding is well known to be heavily memory-bound. For the MoE computation executed on CPUs, the involved skinny GEMMs have an arithmetic intensity of roughly $2\times$ batch size. In local deployment, the batch size is typically small (1–4), which remains far below the compute-to-memory ratio of current CPUs (around 50 FLOPs/byte). As a result, for lower-end CPU configurations, the dominant concern is reduced effective DRAM bandwidth rather than raw compute capability. According to our profiling, CPU-side MoE execution accounts for about 60% of total decode time, with even greater exposure under dual-batch decode. We therefore expect decode performance to scale approximately linearly with available DRAM bandwidth, both in single-stream decoding and under dual-batch overlap.

Lower-VRAM or mid-range GPUs The main hardware sensitivity on the GPU side lies in SLP/DSLP. For moderate prefill lengths (e.g., 16K tokens), performance depends on a tradeoff between experts ring-buffer capacity in VRAM and the ability to overlap weight transfer with computation. Our default design uses a full ring buffer covering all experts in one layer, which best smooths intra-layer transfer/computation imbalance and maximizes overlap. For DeepSeek-V3 and Kimi-K2, this corresponds to 11.3–16.9 GB of ring-buffer space, or 5.6–8.5 GB per GPU when distributed across two GPUs. If a lower-VRAM GPU cannot support this buffer size, overlap becomes less effective, and TTFT for medium-length prefills may degrade. By contrast, at very long prefills (e.g., beyond 50K tokens), the workload is dominated more by computation than by transfer, so performance should depend more directly on the GPU's available compute throughput.

7 Conclusion

This work demonstrates that flagship large MoE models can be served locally with cloud-level quality of service on commodity CPU–GPU platforms. By introducing stream-loading prefill (SLP) and its distributed variant (DSLP) with SmallEP, we achieve 30-second TTFT for 32K–45K prompts. Our intra-node prefill–decode disaggregation and dual-batch attention–MoE overlap sustains concurrency with minimal interference, while an AVX-512–optimized FP8 GEMV and fine-grained CPU parallelism close decode latency gaps for both full-precision and quantized models. Evaluations against mainstream CPU frameworks and BLAS libraries show that our design attains >20 TPS decode responsiveness and high-throughput prefills at substantially lower cost than GPU-centric deployments. Together, these results reshape the local MoE deployment landscape, enabling intact, original-precision inference and robust QoS without datacenter infrastructure, and point to a path for broader, cost-effective access to state-of-the-art applications.

Acknowledgments

This work was supported in part by the Brain Science and Brain-like Intelligence Technology—National Science and Technology Major Project under Grant No. 2025ZD0215500/2025ZD0215502; the Tsinghua University Initiative Scientific Research Program under Grant No. 2022Z11ZRB002; the National Key Research and Development Program of China under Grant No. 2025YFB3003200; and the Jiangsu Provincial Science and Technology Program under Grant No. BE2023005-3.

References

[1] Advanced Micro Devices, Inc. Amd optimizing cpu libraries (aocl). https://www.amd.com/en/developer/aocl.html, n.d. Accessed: 2025-08-29.

[2] Amey Agrawal, Ashish Panwar, Jayashree Mohan, Nipun Kwatra, Bhargav S Gulavani, and Ramachandran Ramjee. Sarathi: Efficient llm inference by piggybacking decodes with chunked prefills. *arXiv preprint arXiv:2308.16369*, 2023.

[3] Moonshot AI. Kimi-k2-instruct (revision 2f7e011), 2025.

[4] Keivan Alizadeh, Seyed Iman Mirzadeh, Dmitry Belenko, S Khatamifard, Minsik Cho, Carlo C Del Mundo, Mohammad Rastegari, and Mehrdad Farajtabar. Llm in a flash: Efficient large language model inference with limited memory. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 12562–12584, 2024.

[5] Yushi Bai, Xin Lv, Jiajie Zhang, Hongchang Lyu, Jiankai Tang, Zhidian Huang, Zhengxiao Du, Xiao Liu, Aohan Zeng, Lei Hou, Yuxiao Dong, Jie Tang, and Juanzi Li. LongBench: A bilingual, multitask benchmark for long context understanding. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 3119–3137, Bangkok, Thailand, August 2024. Association for Computational Linguistics.

[6] Yushi Bai, Shangqing Tu, Jiajie Zhang, Hao Peng, Xiaozhi Wang, Xin Lv, Shulin Cao, Jiazheng Xu, Lei Hou, Yuxiao Dong, Jie Tang, and Juanzi Li. Longbench v2: Towards deeper understanding and reasoning on realistic long-context multitasks. *arXiv preprint arXiv:2412.15204*, 2024.

[7] William Brandon, Aniruddha Nrusimha, Kevin Qian, Zachary Ankner, Tian Jin, Zhiye Song, and Jonathan Ragan-Kelley. Striped attention: Faster ring attention for causal transformers. *arXiv preprint arXiv:2311.09431*, 2023.

[8] Hongtao Chen, Weiyu Xie, Boxin Zhang, Jingqi Tang, Jiahao Wang, Jianwei Dong, Shaoyuan Chen, Ziwei Yuan, Chen Lin, Chengyu Qiu, et al. Ktransformers: Unleashing the full potential of cpu/gpu hybrid inference for moe models. In *Proceedings of the ACM SIGOPS 31st Symposium on Operating Systems Principles*, pages 1014–1029, 2025.

[9] Tri Dao. Flashattention-2: Faster attention with better parallelism and work partitioning. *arXiv preprint arXiv:2307.08691*, 2023.

[10] Tri Dao, Dan Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. Flashattention: Fast and memory-efficient exact attention with io-awareness. *Advances in neural information processing systems*, 35:16344–16359, 2022.

[11] DeepSeek-AI. Deepgemm: Clean and efficient fp8 gemm kernels with fine-grained scaling. https://github.com/deepseek-ai/DeepGEMM, 2025. Accessed: 2025-12-04.

[12] DeepSeek-AI. One more thing, deepseek-v3/r1 inference system overview. https://github.com/deepseek-ai/open-infra-index/blob/main/20250 2OpenSourceWeek/day_6_one_more_thing_deepseekV3R1_inference_system_overview.md, 2025. Accessed: 2025-12-04.

[13] DeepSeek-AI, Aixin Liu, Bei Feng, Bin Wang, Bingxuan Wang, Bo Liu, Chenggang Zhao, Chengqi Dengr, Chong Ruan, Damai Dai, Daya Guo, Dejian Yang, Deli Chen, Dongjie Ji, Erhang Li, Fangyun Lin, Fuli Luo, Guangbo Hao, Guanting Chen, Guowei Li, H. Zhang, Hanwei Xu, Hao Yang, Haowei Zhang, Honghui Ding, Huajian Xin, Huazuo Gao, Hui Li, Hui Qu, J. L. Cai, Jian Liang, Jianzhong Guo, Jiaqi Ni, Jiashi Li, Jin Chen, Jingyang Yuan, Junjie Qiu, Junxiao Song, Kai Dong, Kaige Gao, Kang Guan, Lean Wang, Lecong Zhang, Lei Xu, Leyi Xia, Liang Zhao, Liyue Zhang, Meng Li, Miaojun Wang, Mingchuan Zhang, Minghua Zhang, Minghui Tang, Mingming Li, Ning Tian, Panpan Huang, Peiyi Wang, Peng Zhang, Qihao Zhu, Qinyu Chen, Qiushi Du, R. J. Chen, R. L. Jin, Ruiqi Ge, Ruizhe Pan, Runxin Xu, Ruyi Chen, S. S. Li, Shanghao Lu, Shangyan Zhou, Shanhuang Chen, Shaoqing Wu, Shengfeng Ye, Shirong Ma, Shiyu Wang, Shuang Zhou, Shuiping Yu, Shunfeng Zhou, Size Zheng, T. Wang, Tian Pei, Tian Yuan, Tianyu Sun, W. L. Xiao, Wangding Zeng, Wei An, Wen Liu, Wenfeng Liang, Wenjun Gao, Wentao Zhang, X. Q. Li, Xiangyue Jin, Xianzu Wang, Xiao Bi, Xiaodong Liu, Xiaohan Wang, Xiaojin Shen, Xiaokang Chen, Xiaosha Chen, Xiaotao Nie, Xiaowen Sun, Xiaoxiang Wang, Xin Liu, Xin Xie, Xingkai Yu, Xinnan Song, Xinyi Zhou, Xinyu Yang, Xuan Lu, Xuecheng Su, Y. Wu, Y. K. Li, Y. X. Wei, Y. X. Zhu, Yanhong Xu, Yanping Huang, Yao Li, Yao Zhao, Yaofeng Sun, Yaohui Li, Yaohui Wang, Yi Zheng, Yichao Zhang, Yiliang Xiong, Yilong Zhao, Ying He, Ying Tang, Yishi Piao, Yixin Dong, Yixuan Tan, Yiyuan Liu, Yongji Wang, Yongqiang Guo, Yuchen Zhu, Yuduan Wang, Yuheng Zou, Yukun Zha, Yunxian Ma, Yuting Yan, Yuxiang You, Yuxuan Liu, Z. Z. Ren, Zehui Ren, Zhangli Sha, Zhe Fu, Zhen Huang, Zhen Zhang, Zhenda Xie, Zhewen Hao, Zhihong Shao, Zhiniu Wen, Zhipeng Xu, Zhongyu Zhang, Zhuoshu Li, Zihan Wang, Zihui Gu, Zilin Li, and Ziwei Xie. DeepSeek-V2:

A Strong, Economical, and Efficient Mixture-of-Experts Language Model, June 2024. arXiv:2405.04434 [cs].

[14] Maxim Fishman, Brian Chmiel, Ron Banner, and Daniel Soudry. Scaling fp8 training to trillion-token llms. *arXiv preprint arXiv:2409.12517*, 2024.

[15] ggml-org. llama.cpp: Llm inference in c/c++. https://github.com/ggml-org/llama.cpp, 2025. Accessed: 2025-12-04.

[16] Edinburgh Dataset Analytics Working Group. mmlu-redux (revision 53bc457), 2024.

[17] Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Ruoyu Zhang, Runxin Xu, Qihao Zhu, Shirong Ma, Peiyi Wang, Xiao Bi, et al. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning. *arXiv preprint arXiv:2501.12948*, 2025.

[18] Xin He, Shunkang Zhang, Yuxin Wang, Haiyan Yin, Zihao Zeng, Shaohuai Shi, Zhenheng Tang, Xiaowen Chu, Ivor Tsang, and Ong Yew Soon. Expertflow: Optimized expert activation and token allocation for efficient mixture-of-experts inference. *arXiv preprint arXiv:2410.17954*, 2024.

[19] Ranggi Hwang, Jianyu Wei, Shijie Cao, Changho Hwang, Xiaohu Tang, Ting Cao, and Mao Yang. Pre-gated moe: An algorithm-system co-design for fast and scalable mixture-of-expert inference. In *2024 ACM/IEEE 51st Annual International Symposium on Computer Architecture (ISCA)*, pages 1018–1031. IEEE, 2024.

[20] ikawrakow. ik_llama.cpp: llama.cpp fork with additional sota quants and improved performance. https://github.com/ikawrakow/ik_llama.cpp, 2025. Accessed: 2025-08-31.

[21] Intel. Intel® 64 and ia-32 architectures software developer manuals, 3/5/2025. Accessed: 2025.

[22] Sam Ade Jacobs, Masahiro Tanaka, Chengming Zhang, Minjia Zhang, Shuaiwen Leon Song, Samyam Rajbhandari, and Yuxiong He. Deepspeed ulysses: System optimizations for enabling training of extreme long sequence transformer models. *arXiv preprint arXiv:2309.14509*, 2023.

[23] Keisuke Kamahori, Tian Tang, Yile Gu, Kan Zhu, and Baris Kasikci. Fiddler: Cpu-gpu orchestration for fast inference of mixture-of-experts models. *arXiv preprint arXiv:2402.07033*, 2024.

[24] Yechan Kim, Hwijoon Lim, and Dongsu Han. Scaling beyond the gpu memory limit for large mixture-of-experts model training. In *Forty-first International Conference on Machine Learning*, 2024.

[25] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph Gonzalez, Hao Zhang, and Ion Stoica. Efficient memory management for large language model serving with page-dattention. In *Proceedings of the 29th Symposium on Operating Systems Principles*, pages 611–626, 2023.

[26] Dmitry Lepikhin, HyoukJoong Lee, Yuanzhong Xu, Dehao Chen, Orhan Firat, Yanping Huang, Maxim Krikun, Noam Shazeer, and Zhifeng Chen. Gshard: Scaling giant models with conditional computation and automatic sharding. *arXiv preprint arXiv:2006.16668*, 2020.

[27] Zhuohan Li, Lianmin Zheng, Yinmin Zhong, Vincent Liu, Ying Sheng, Xin Jin, Yanping Huang, Zhifeng Chen, Hao Zhang, Joseph E Gonzalez, et al. {AlpaServe}: Statistical multiplexing with model parallelism for deep learning serving. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*, pages 663–679, 2023.

[28] Aixin Liu, Bei Feng, Bing Xue, Bingxuan Wang, Bochao Wu, Chengda Lu, Chenggang Zhao, Chengqi Deng, Chenyu Zhang, Chong Ruan, et al. Deepseek-v3 technical report. *arXiv preprint arXiv:2412.19437*, 2024.

[29] Hao Liu, Matei Zaharia, and Pieter Abbeel. Ring attention with blockwise transformers for near-infinite context. *arXiv preprint arXiv:2310.01889*, 2023.

[30] Paulius Micikevicius, Stuart Oberman, Pradeep Dubey, Marius Cornea, Andres Rodriguez, Ian Bratt, Richard Grisenthwaite, Norm Jouppi, Chiachen Chou, Amber Huffman, et al. Ocp 8-bit floating point specification (ofp8). *Open Compute Project*, 2023.

[31] Paulius Micikevicius, Dusan Stosic, Neil Burgess, Marius Cornea, Pradeep Dubey, Richard Grisenthwaite, Sangwon Ha, Alexander Heinecke, Patrick Judd, John Kamalu, et al. Fp8 formats for deep learning. *arXiv preprint arXiv:2209.05433*, 2022.

[32] Paulius Micikevicius, Dusan Stosic, Neil Burgess, Marius Cornea, Pradeep Dubey, Richard Grisenthwaite, Sangwon Ha, Alexander Heinecke, Patrick Judd, John Kamalu, et al. Fp8 formats for deep learning. *arXiv preprint arXiv:2209.05433*, 2022.

[33] NVIDIA. Tensorrt-llm's documentation. Accessed: 2025.

[34] NVIDIA Corporation. NVIDIA H100 Tensor Core GPU Architecture. White paper, NVIDIA, 2022. H100 Tensor Core GPU; describes the Hopper GPU architecture.

[35] OpenBLAS Project. Openblas: An optimized blas and lapack library. https://openblas.net, n.d. Accessed: 2025-08-29.

[36] Houwen Peng, Kan Wu, Yixuan Wei, Guoshuai Zhao, Yuxiang Yang, Ze Liu, Yifan Xiong, Ziyue Yang, Bolin Ni, Jingcheng Hu, et al. Fp8-lm: Training fp8 large language models. *arXiv preprint arXiv:2310.18313*, 2023.

[37] Ruoyu Qin, Zheming Li, Weiran He, Jialei Cui, Feng Ren, Mingxing Zhang, Yongwei Wu, Weimin Zheng, and Xinran Xu. Mooncake: Trading more storage for less computation—a {KVCache-centric} architecture for serving {LLM} chatbot. In *23rd USENIX Conference on File and Storage Technologies (FAST 25)*, pages 155–170, 2025.

[38] Qwen Team. Qwen3. https://qwenlm.github.io/blog/qwen3/, 2025. Accessed: 2025-12-04.

[39] Jay Shah, Ganesh Bikshandi, Ying Zhang, Vijay Thakkar, Pradeep Ramani, and Tri Dao. Flashattention-3: Fast and accurate attention with asynchrony and low-precision. *Advances in Neural Information Processing Systems*, 37:68658–68685, 2024.

[40] Noam Shazeer, Azalia Mirhoseini, Krzysztof Maziarz, Andy Davis, Quoc Le, Geoffrey Hinton, and Jeff Dean. Outrageously large neural networks: The sparsely-gated mixture-of-experts layer. *arXiv preprint arXiv:1701.06538*, 2017.

[41] Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. Megatron-lm: Training multi-billion parameter language models using model parallelism. *arXiv preprint arXiv:1909.08053*, 2019.

[42] Xiaoniu Song, Zihang Zhong, Rong Chen, and Haibo Chen. Promoe: Fast moe-based llm serving using proactive caching. *arXiv preprint arXiv:2410.22134*, 2024.

[43] Peng Tang, Jiacheng Liu, Xiaofeng Hou, Yifei Pu, Jing Wang, Pheng-Ann Heng, Chao Li, and Minyi Guo. Hobbit: A mixed precision expert offloading system for fast moe inference. *arXiv preprint arXiv:2411.01433*, 2024.

[44] Kimi Team, Yifan Bai, Yiping Bao, Guanduo Chen, Jiahao Chen, Ningxin Chen, Ruijue Chen, Yanru Chen, Yuankun Chen, Yutian Chen, Zhuofu Chen, Jialei Cui, Hao Ding, Mengnan Dong, Angang Du, Chenzhuang Du, Dikang Du, Yulun Du, Yu Fan, Yichen Feng, Kelin Fu, Bofei Gao, Hongcheng Gao, Peizhong Gao, Tong Gao, Xinran Gu, Longyu Guan, Haiqing Guo, Jianhang Guo, Hao Hu, Xiaoru Hao, Tianhong He, Weiran He, Wenyang He, Chao Hong, Yangyang Hu, Zhenxing Hu, Weixiao Huang, Zhiqi Huang, Zihao Huang, Tao Jiang, Zhejun Jiang, Xinyi Jin, Yongsheng Kang, Guokun Lai, Cheng Li, Fang Li, Haoyang Li, Ming Li, Wentao Li, Yanhao Li, Yiwei Li, Zhaowei Li, Zheming Li, Hongzhan Lin, Xiaohan Lin, Zongyu Lin, Chengyin Liu, Chenyu Liu, Hongzhang Liu, Jingyuan Liu, Junqi Liu, Liang Liu, Shaowei Liu, T. Y. Liu, Tianwei Liu, Weizhou Liu, Yangyang Liu, Yibo Liu, Yiping Liu, Yue Liu, Zhengying Liu, Enzhe Lu, Lijun Lu, Shengling Ma, Xinyu Ma, Yingwei Ma, Shaoguang Mao, Jie Mei, Xin Men, Yibo Miao, Siyuan Pan, Yebo Peng, Ruoyu Qin, Bowen Qu, Zeyu Shang, Lidong Shi, Shengyuan Shi, Feifan Song, Jianlin Su, Zhengyuan Su, Xinjie Sun, Flood Sung, Heyi Tang, Jiawen Tao, Qifeng Teng, Chensi Wang, Dinglu Wang, Feng Wang, Haiming Wang, Jianzhou Wang, Jiaxing Wang, Jinhong Wang, Shengjie Wang, Shuyi Wang, Yao Wang, Yejie Wang, Yiqin Wang, Yuxin Wang, Yuzhi Wang, Zhaoji Wang, Zhengtao Wang, Zhexu Wang, Chu Wei, Qianqian Wei, Wenhao Wu, Xingzhe Wu, Yuxin Wu, Chenjun Xiao, Xiaotong Xie, Weimin Xiong, Boyu Xu, Jing Xu, Jinjing Xu, L. H. Xu, Lin Xu, Suting Xu, Weixin Xu, Xinran Xu, Yangchuan Xu, Ziyao Xu, Junjie Yan, Yuzi Yan, Xiaofei Yang, Ying Yang, Zhen Yang, Zhilin Yang, Zonghan Yang, Haotian Yao, Xingcheng Yao, Wenjie Ye, Zhuorui Ye, Bohong Yin, Longhui Yu, Enming Yuan, Hongbang Yuan, Mengjie Yuan, Haobing Zhan, Dehao Zhang, Hao Zhang, Wanlu Zhang, Xiaobin Zhang, Yangkun Zhang, Yizhi Zhang, Yongting Zhang, Yu Zhang, Yutao Zhang, Yutong Zhang, Zheng Zhang, Haotian Zhao, Yikai Zhao, Huabin Zheng, Shaojie Zheng, Jianren Zhou, Xinyu Zhou, Zaida Zhou, Zhen Zhu, Weiyu Zhuang, and Xinxing Zu. Kimi k2: Open agentic intelligence, 2025.

[45] thu-pacman. Chitu: High-performance inference framework for large language models. https://github.com/thu-pacman/chitu, 2025. Accessed: 2025-12-04.

[46] TIGER-Lab. Mmlu-pro (revision efb53c9), 2024.

[47] unsloth. unsloth/deepseek-r1-0528-bf16. https://huggingface.co/unsloth/DeepSeek-R1-0528-BF16, 2025. Accessed: 2025-12-04.

[48] unsloth. unsloth/deepseek-r1-gguf. https://huggingface.co/unsloth/DeepSeek-R1-GGUF, 2025. Accessed: 2025-12-04.

[49] unsloth. unsloth/deepseek-v3-0324-bf16. https://huggingface.co/unsloth/DeepSeek-V3-0324-BF16, 2025. Accessed: 2025-12-04.

[50] unsloth. unsloth/deepseek-v3-gguf. https://huggingface.co/unsloth/DeepSeek-V3-GGUF, 2025. Accessed: 2025-12-04.

[51] Bin Wang, Bojun Wang, Changyi Wan, Guanzhe Huang, Hanpeng Hu, Haonan Jia, Hao Nie, Mingliang Li, Nuo Chen, Siyu Chen, et al. Step-3 is large yet affordable: Model-system co-design for cost-effective decoding. *arXiv preprint arXiv:2507.19427*, 2025.

[52] Ruizhe Wang, Yeyun Gong, Xiao Liu, Guoshuai Zhao, Ziyue Yang, Baining Guo, Zhengjun Zha, and Peng Cheng. Optimizing large language model training using fp4 quantization. *arXiv preprint arXiv:2501.17116*, 2025.

[53] Amy Yang, Jingyi Yang, Aya Ibrahim, Xinfeng Xie, Bangsheng Tang, Grigory Sizov, Jeremy Reizenstein, Jongsoo Park, and Jianyu Huang. Context parallelism for scalable million-token inference. *arXiv preprint arXiv:2411.01783*, 2024.

[54] Gyeong-In Yu, Joo Seong Jeong, Geon-Woo Kim, Soojeong Kim, and Byung-Gon Chun. Orca: A distributed serving system for {Transformer-Based} generative models. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 521–538, 2022.

[55] Chenggang Zhao, Shangyan Zhou, Liyue Zhang, Chengqi Deng, Zhean Xu, Yuxuan Liu, Kuai Yu, Jiashi Li, and Liang Zhao. Deepep: an efficient expert-parallel communication library. https://github.com/deepseek-ai/DeepEP, 2025.

[56] Lianmin Zheng, Liangsheng Yin, Zhiqiang Xie, Chuyue Livia Sun, Jeff Huang, Cody Hao Yu, Shiyi Cao, Christos Kozyrakis, Ion Stoica, Joseph E Gonzalez, et al. Sglang: Efficient execution of structured language model programs. *Advances in Neural Information Processing Systems*, 37:62557–62583, 2024.

[57] Kan Zhu, Yilong Zhao, Liangyu Zhao, Gefei Zuo, Yile Gu, Dedong Xie, Yufei Gao, Qinyu Xu, Tian Tang, Zihao Ye, et al. Nanoflow: Towards optimal large language model serving throughput. *arXiv preprint arXiv:2408.12757*, 2024.

[58] Zilin Zhu. ring-flash-attention. https://github.com/zhuzilin/ring-flash-attention, 2024. GitHub repository.

[59] Pengfei Zuo, Huimin Lin, Junbo Deng, Nan Zou, Xingkun Yang, Yingyu Diao, Weifeng Gao, Ke Xu, Zhangyu Chen, Shirui Lu, et al. Serving large language models on huawei cloudmatrix384. *arXiv preprint arXiv:2506.12708*, 2025.

UEP: Portable Expert-Parallel Communication

Ziming Mao[†] Yihan Zhang[‡] Chihan Cui[§] Zhen Huang[¶] Kaichao You[♣] Zhongjie Chen[♡]
Zhiying Xu[♠*] Zhenyu Gu[¶] Scott Shenker[†◇] Costin Raiciu[⋆] Yang Zhou[‡] Ion Stoica[†]
[†]*UC Berkeley* [‡]*UC Davis* [§]*UW–Madison* [¶]*AMD* [♣]*Independent Researcher* [♡]*Tsinghua University*
[♠]*Amazon Web Services* [◇]*ICSI* [⋆]*Broadcom & University Politehnica of Bucharest*

Abstract

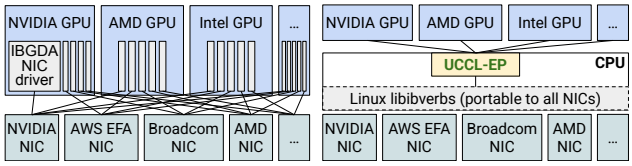
Modern Mixture-of-Experts (MoE) workloads rely on expert parallelism (EP) to achieve high GPU efficiency. State-of-the-art EP communication libraries, such as DeepEP, rely on GPU-initiated RDMA communication. Although performant, they have poor portability across heterogeneous GPU and NIC hardware. The poor portability is rooted in its architecture: GPU-initiated RDMA communication requires tight vertical integration between GPUs and NICs, e.g., GPU writing to NIC driver/MMIO interfaces.

We present UEP, a portable EP communication system that delivers high performance across heterogeneous GPU and NIC hardware. UEP replaces GPU-initiated RDMA with a high-throughput GPU-CPU control channel: compact token-routing commands are transferred to multithreaded CPU proxies, which then issue GPUDirect RDMA operations on behalf of GPUs. UEP further emulates various ordering semantics required by specialized EP communication modes using RDMA immediate data, enabling correctness on NICs that lack such ordering, e.g., AWS EFA. We implement UEP on NVIDIA and AMD GPUs with EFA and Broadcom NICs. On EFA, it outperforms the best existing EP solution by 2.1× for dispatch and combine throughput. UEP also improves token throughput on SGLang by up to 40% on the NVIDIA+EFA platform, and improves DeepSeek-V3 training throughput over the AMD Primus/Megatron-LM framework by up to 45% on a 16-node AMD+Broadcom platform.

1 Introduction

State-of-the-art large language models (LLMs), such as DeepSeek-V3 [12,30], OpenAI gpt-oss [45], Google Gemini-3 Pro [16], and Meta Llama 4 [2], are increasingly based on the Mixture-of-Experts (MoE) architecture. In a MoE layer, a gating network running on GPUs selects a small subset of experts for each token activation, dispatches the token activation to those experts, and then aggregates their output activations. Modern MoE models typically instantiate hundreds of experts that specialize in different input patterns, so that only a few experts are active for each token. This sparsity allows MoE models to achieve accuracy comparable to large dense models while using only a fraction of the per-token inference cost, making them the *de facto* choice for many frontier LLMs.

*This work does not relate to the position at Amazon.



(a) IBGDA-style. **(b)** UEP.

Figure 1: Assuming m GPU vendors and n NIC vendors, UEP enables $O(m)$ effort, instead of IBGDA's $O(m \times n)$, to support GPU-initiated token-level communication for expert parallelism.

Training and serving large MoE models require expert parallelism (EP), which places different experts on different GPUs and communicates token activations among GPUs in an all-to-all manner. By sparsely sharding experts on different GPUs, EP leaves enough GPU memory for matrix multiplication on large batch sizes (e.g., 4096 in DeepSeek-V3 [30]), thus enabling high GPU resource efficiency. Expert-parallel communication plays a pivotal role in the EP efficiency [30, 63], because token activations are small (e.g., 7KB), dispatch and combine operations are frequent (e.g., selecting 8 experts per token), and routing destinations are only determined at runtime in GPUs (§2.1).

GPU-initiated token-level (fine-grained) communication (§2.2) is an emerging and key communication pattern for efficient token dispatch and combine at runtime, where DeepEP [64] by DeepSeek is a popular communication system implementing it. Different from CPU-initiated bulk-transfer (coarse-grained) communication in NCCL/RCCL [3, 41], DeepEP leverages the advanced NVIDIA IBGDA (InfiniBand GPUDirect Async) [43] technique that enables GPUs to directly operate RDMA NICs (network interface cards) to write out small activations. DeepEP has been widely adopted by various training and serving frameworks such as Megatron-LM [6], vLLM [59], and SGLang [58] with state-of-the-art performance.

Although GPU-initiated token-level communication leads to high performance, its design unfortunately results in poor portability. There are two key reasons: GPUs directly issuing RDMA operations to NICs prevent interoperability across different GPUs and NICs, as it requires the GPU to write to NIC-defined MMIO interfaces typically requiring proprietary register layouts and tight hardware-level co-design between the NICs and GPUs [26, 44, 51]; GPU kernels also impose strict ordering and delivery semantics assumptions on the underlying network, which are often misaligned with the capabilities and semantics

of heterogeneous NICs. As shown in Figure 1a, ML infrastructure developers need to vertically integrate the GPU and NIC software ecosystems. This involves complex and subtle code migration and maintenance that are both NIC vendor-specific and GPU vendor-specific. Assume there are m types of GPU accelerators and n types of NICs. Developers need to pay $O(m \times n)$ effort to enable such communication on heterogeneous hardware. Because of this portability issue, the official DeepEP [64] only supports NVIDIA GPUs and NICs, creating severe vendor lock-in and high portability effort for alternative GPU and NIC devices. For example, at the time of writing, it remains impossible to run DeepEP on any AWS GPU instances with AWS EFA RDMA NICs; it is also challenging to run DeepEP on AMD GPUs with Broadcom NICs, the second-biggest GPU and NIC vendors, respectively. At a high level, such vertical integration is somewhat reminiscent of mainframe servers, which get replaced by less-coupled and more portable commodity servers from heterogeneous vendors in modern cloud computing.

This paper introduces a new portable expert-parallel communication architecture for GPU-initiated token-level communication with high performance. We envision that with our architecture, developers only need to pay m times effort to enable systems like DeepEP on heterogeneous GPU and NIC hardware, as shown in Figure 1b.

Our key insight is to leverage the host CPU to help break the tight coupling between GPUs and NICs, where the CPU is essentially portable to any GPUs and NICs via the `libibverbs` library [48] maintained by the Linux community and NIC vendors. Unlike traditional CPU-driven approaches, communication is still initiated by the GPU: the library passes only *small control information* from the GPU to the CPU, while the data payload is transferred directly between GPUs. GPUs and CPUs are connected through high-throughput, low-latency interconnects such as PCIe or even faster NVLink-C2C [42], which allows efficient transferring of small control data from GPUs to CPUs. By embedding the token dispatching information, such as source and destination address, into the control data, CPUs directly issue GPUDirect RDMA to write out the activations on behalf of GPUs. We note that modern GPU servers usually have hundreds of CPU cores, which are often heavily underutilized, e.g., 20%-45% CPU utilization reported by several industry companies in their GPU clusters [19, 23]; From discussion with a model training team inside a major GPU vendor, their model training using Megatron-LM [40] yields on average 14.5% CPU utilization.

Moreover, flexible host CPUs can help bridge the semantic gap between the guarantees required by specialized EP communication systems and the primitives actually provided by heterogeneous NICs. For example, EP communication libraries often require a specific write-then-atomic ordering to inform remote GPUs of data arrival, but such a requirement is typically not enforced by all RDMA NICs, such as AWS EFA NICs, which do not guarantee ordering.

To realize these insights, we need to address two challenges: i) how to efficiently transfer control data from GPUs to CPUs so that the CPU does not become the bottleneck. This is challenging, especially given the frequent token dispatch/combine operations as high as 7Mop/s/GPU under 7KB activations [30] and a 400G network. ii) How to use CPUs to express and enforce various delivery semantics (e.g., write-then-atomic) in CPU to accommodate heterogeneous NICs that do not support them on their own.

We present UEP, a portable expert-parallel communication system with GPU-initiated token-level communication. UEP decouples communication initiation from communication execution: it keeps GPUs initiating communication for fine-grained token transfer, and delegates the communication tasks to the host CPU. UEP addresses the first challenge with an efficient multithreaded, lock-free FIFO communication channel between GPUs and CPUs. This channel minimizes the PCIe traversing overhead and allows multiple CPU proxy threads to forward and execute the GPU-generated token routing decisions, achieving millions of messages per second from GPUs to CPUs. To address the second challenge, UEP leverages the immediate data feature that is widely available in RDMA NICs (this immediate data field has been standardized into the RoCEv2 packet header [1]). UEP embeds sequence information into this immediate data and lets the receiver hold messages (e.g., atomic updates) until all previous corresponding writes have finished. We show how host CPUs can flexibly bridge the delivery guarantees that GPUs require and the ones actually provided by the networking layer.

We have implemented UEP and enabled it on various heterogeneous platforms, including NVIDIA GPUs + AWS EFA NICs, and AMD GPUs + Broadcom NICs. On the NVIDIA+EFA platform, compared to the second-best EP solution PPLX [29], UEP achieved up to 2.1× higher throughput. On the AMD+Broadcom platform, UEP achieves comparable performance to the original DeepEP on the NVIDIA-only platform. UEP supports drop-in replacement for DeepEP-compatible applications without any line of code change. It speeds up serving throughput in the SGLang framework by up to 40% on the NVIDIA+EFA platform, and improves DeepSeek-V3 training throughput in the AMD Primus/Megatron-LM framework by up to 45% on a 16-node AMD+Broadcom platform. To the best of our knowledge, UEP is the first work that enables running GPU-initiated token-level communication on non-NVIDIA platforms. UEP is open-sourced.[1]

2 Background

2.1 Expert-parallel communication

Mixture-of-Experts (MoE) architectures have emerged as a leading architectural pattern for building state-of-the-art LLMs as they provide massive parameter capacity while keeping per-token compute low by activating only a small subset

[1] https://github.com/uccl-project/uccl/tree/main/ep

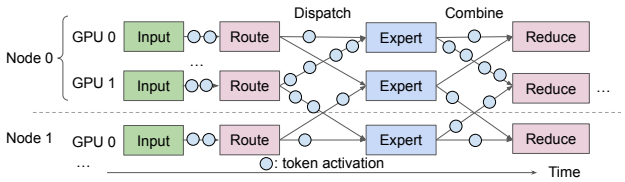


Figure 2: The MoE communication pattern consists of the dispatch phase and the combine phase. In the dispatch phase, the router sends each token's input activations to one or more selected experts. In the combine phase, expert activations are collected and aggregated back to the sender. The dispatch and combine phases feature *irregular*, *fine-grained* communication.

of experts. This sparsity enables specialization—experts learn domain-specific behaviors—while preserving general-purpose performance. As a result, many frontier models now adopt MoE designs, including OpenAI's gpt-oss [45], Google's Gemini-3 Pro [16] and GLaM [14], Mistral's Mixtral 8×7B [21], DeepSeek's DeepSeekMoE [8] and DeepSeek-V3 [12, 30] family, the Allen Institute's OLMoE [34], and Meta's Llama 4 [33], all of which use MoE to push model capacity and quality without the prohibitive compute costs of dense models. These systems power a wide range of applications—from interactive agents and large-scale pretraining to multimodal reasoning and domain-specialized tasks.

MoE models introduce a distinctive communication pattern that differs fundamentally from dense all-reduce or pipeline-parallel patterns found in other models. In a MoE layer, each input token activation is dynamically routed to a small subset of experts based on a learned gating function. This induces a sparse all-to-all pattern: Every GPU holds a subset of experts, so token activations must be dispatched from the originating GPU to the GPUs that host the selected experts and then gathered back to their origin (Figure 2). The destination GPUs are determined based on the learned gating function during runtime. Every MoE layer's forward pass has two communication phases: *dispatch*, where activations are sent to expert GPUs, followed by a *combine* phase, where expert outputs are returned and merged in the original token order. Expert-parallel communication has several new characteristics:

Fine-grained token-level[2] transfers. In MoE dispatch and combine, each token activation is small, on the order of 7KB (e.g., FP8, hidden size of 7168 [30]), so a naive implementation issues a large number of tiny GPU-GPU transfers. Unlike traditional collective communication (e.g., large, batched all-reduces or all-to-alls) that aggregates data into large bandwidth-efficient messages, MoE communication is naturally fragmented. Traditional approaches must therefore either push many small work-queue entries to the NIC, or first pack activations into contiguous per-expert buffers on the GPU, which consume SM cycles and add packing latency on the critical path.

[2]For simplicity, we use token and token activation interchangeably.

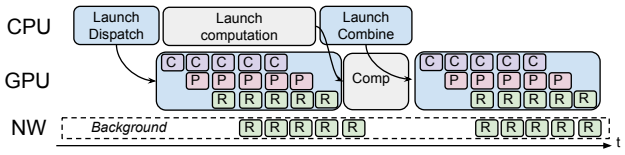


Figure 3: GPU-initiated token-level communication in DeepEP (High-throughput mode, single batch for illustration). *C* stands for intra-node data copying via NVLink, *P* stands for processing, and *R* stands for RDMA communication. Different phases (computation, communication, and copying) interleave.

Irregular communication. The number of tokens routed to each expert also varies every iteration, because routing decisions are computed at runtime by the MoE gating network. In contrast, traditional collectives assume a fixed set of participants and largely predictable, symmetric message sizes, which allows for static, precomputed communication schedules. However, MoE communication breaks these assumptions: dynamic routing creates significant load imbalance and introduces substantial overhead from fine-grained, data-dependent transfers [11, 15]. This irregularity makes it hard to precompute efficient communication schedules or reuse static buffer layouts, forcing implementations to dynamically size and route messages every step and amplifying the overheads of the already fine-grained transfers.

Communication is the bottleneck of MoE training and serving. Expert-parallel communication is already a major performance bottleneck in production systems: it can consume 43.6% of the forward pass and 32% of end-to-end training time on GPUs [24]. Serving faces the same challenge, with inter-device communication accounting for up to 47% of total execution time across popular MoE models and frameworks [61].

2.2 Expert-parallel communication requires GPU-initiated token-level communication

Given the central role of MoE communication, recent specialized systems—notably DeepEP [64]—have introduced *GPU-initiated token-level communication*, as illustrated in Figure 3. This design involves GPU threads *directly* submitting transfer commands to the NIC, using NVIDIA IBGDA (InfiniBand GPUDirect Async) [43].

GPU-initiated communication enables fine-grained and pipelined overlap *on a token basis*, where transfer for a single token or a chunk of tokens can overlap with other *phases* of communication, such as data copying (denoted "C") between application tensor buffers and RDMA transport buffers, token forwarding from the scale-out domain to the scale-up domain, and necessary computation (denoted "P") such as the reduce during the combine phase, or data type conversion such as from BF16 to FP8 to reduce communication volume. By breaking communication into these smaller GPU-triggered units (e.g., a few tokens), the system can better utilize both network and compute resources and significantly reduce end-to-end communication latency.

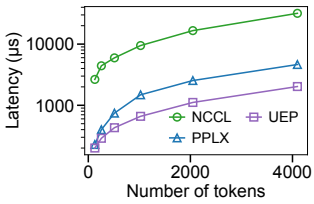


Figure 4: GPU-initiated token-level communication outperforms coarse-grained bulk transfer (e.g., packing tokens into a contiguous buffer on GPU then CPU initiating a single contiguous transfer) on NV_EFA3 (testbed details listed in Table 2). The overhead of packing becomes high as the number of tokens increases. The y-axis is in log scale.

GPU-initiated token-level communication also enables many optimization opportunities, such as message deduplication: if the token activation is routed to experts residing on multiple GPUs *on the same node*, the communication library can only send the token activation once with RDMA, and rely on intra-node forwarding to multiple experts for maximal speed. A second optimization enabled by GPU-initiated token-level communication is hierarchical reduce: an intra-node reduce (weighted sum) is performed on each node for a *chunk of tokens*, the result is sent back to the sender rank for another inter-node reduce: all of which is pipelined with background network transfer. Such optimization techniques were previously not feasible without GPU-initiated token-level communication, and have enabled a significant reduction in the amount of traffic needed to send over the network, and improvement in end-to-end performance.

To compare, some inference and training frameworks adopt coarse-grained transfer, such as with NCCL [41] or RCCL [3], or other general-purpose collective libraries. They require either the application packing tokens into a contiguous per-destination-rank transfer buffer or transferring small tokens one by one. The former incurs a high overhead of packing the token; the latter suffers from limited transfer throughput with small messages. For example, PPLX [29] adopts CPU-initiated communication with on-GPU token packing without fine-grained token deduplication and hierarchical reduce, and it does not scale as the number of tokens increases. The evaluation in Figure 4 shows that such approaches suffer from poor performance.

2.3 Existing GPU-initiated solutions couple GPUs and NICs, harming portability

While GPU-initiated token-level communication has been shown to improve performance, unfortunately, it compromises portability by *tightly coupling* GPUs and NICs. The reasons are two-fold:

Lack of hardware interoperability. The mainstream mechanism of GPU-initiated token-level communication is GPU threads directly posting to the RDMA NICs, or known as IBGDA [43]. IBGDA requires InfiniBand-capable NICs that support GPUDirect RDMA so the GPU can directly write to the NIC driver-defined MMIO interface. It also depends on a compatible software stack, typically including NVSHMEM and the necessary CUDA/RDMA drivers, to enable GPU-initiated network operations. Unfortunately, this makes it difficult, if not impossible, to support such communication patterns for non-NVIDIA GPUs or non-InfiniBand-capable NICs.

GPUs impose strict delivery semantics on NICs. While GPU threads are good for massive parallelism, they are limited in terms of the ability to control and manage the transfer. These communication libraries typically expect strict delivery semantics (e.g., "write-then-atomic" patterns, applying an atomic only after x number of writes have been delivered). This requires the networking layer to respect such delivery semantics, such as providing ordering guarantees. Furthermore, GPU threads lack the flexibility to manage communication. For example, DeepEP issues one-sided writes and relies on the receiver to busy poll a flag to detect incoming transfer. Flow control (such as pacing the sender's rate of communication) becomes an issue since the sender is unaware of the message delay or any potential congestion.

Taken together, this means that the GPU-initiated communication introduces strict requirements on the underlying networking layer, relying on it to provide reliability, ordering, as well as efficient network control. Unfortunately, maintaining them not only restricts a narrow selection of networking devices, as well as increasing the necessary fixed cost of enforcing these guarantees in hardware.

As a result, GPU-initiated token-level communication cannot be supported on cloud NICs (e.g., AWS EFA [52] with SRD transport [54]) that lack ordering guarantees, as well as non-NVIDIA GPUs, which lack a hardware mechanism to directly issue commands from the GPU to NICs. As GPU threads have little visibility into transfer delays and network congestion, this also poses significant challenges to practical deployment, as the networking environment needs to be carefully tuned to ensure congestion does not occur [64].

Importance of portability. Portability is essential for cost efficiency and avoiding vendor lock-in. Prior works have demonstrated significant performance and cost benefits of using heterogeneous hardware [17, 20, 22, 28, 32, 62]. Contemporary large-language-model systems increasingly rely on heterogeneous hardware: different data-centers and cloud providers deploy different NICs with distinct transport protocols (e.g., RC vs. SRD) and capabilities. As such, achieving portability in the communication layer is critical to reduce cost and improve performance, simplify integration into existing inference or training engines, and make efficient expert-parallel communication usable for a wide range of users.

2.4 The challenges of designing a portable and performant expert-parallel communication system

Designing a portable expert-parallel communication system requires breaking the coupling between GPUs and NICs. We describe the main challenges in the following.

	Support hete GPUs	Support hete NICs	Token dedup&reduce	High perf w/ small tokens
NCCL/RCCL [3,41]	✓	✓	✗	✗
PPLX [29]	✗	✓	✗	✓
DeepEP [64]	✗	✗	✓	✓
Mori-EP [50]	✗	✗	✓	✓
ROCm-DeepEP [49]	✗	✗	✓	✓
CPU-asst IBGDA [38]	✓	✗	–	–
UEP	✓	✓	✓	✓

Table 1: Comparison of major existing EP communication systems. "Hete." denotes support for heterogeneous (multi-vendor) GPUs or NICs.

Heterogeneous GPUs and NICs. Both GPUs and NICs are heterogeneous, coming from different vendors with different software ecosystems, e.g., NVIDIA, AMD, AWS EFA, Broadcom, Intel. To enable the GPU to directly initiate communication to the NIC (without involving the CPUs) often requires GPU knowing and accessing proprietary register layouts and tight hardware-level co-design between the NICs and GPUs. This is, at best, error-prone and difficult to do cross-vendor, and more often simply not supported.

Delivery semantics guarantees. Hardware transports differ in whether they guarantee in-order delivery of messages. For example, the ConnectX [37] RC transport offers in-order semantics, while the EFA SRD protocol offers reliable but unordered delivery. When the GPU kernel assumes ordering guarantees from the networking layer (for instance, issuing operations in a strict sequence without additional synchronization) or requiring a certain group of messages to arrive before a control message is delivered, moving to a transport that delivers out-of-order breaks correctness. A portable communication architecture, therefore, must assume minimal guarantees on the networking layer, or better, allow easy configuration in the software layer to adapt to the heterogeneous networking layer.

Existing solutions remained specific to each GPU-NIC. We observe that existing GPU-initiated communication libraries have remained largely ad hoc and involve one-time solutions on a particular combination of GPUs and NICs. We summarize existing systems in Table 1. Concurrent works MORI-EP [50] and the ROCm port of DeepEP [49] are specialized libraries for RDMA and GPU integration that work primarily on AMD GPUs with NVIDIA NICs. CPU-assisted IBGDA [27] relies on a single CPU proxy to relay transfer commands from NVIDIA GPUs, suffering from scalability issues, and it can only work on NVIDIA NICs that provide strict ordering guarantees. These solutions require a ported solution for each GPU and NIC vendor pair, inflating the development cost. We provide a more exhaustive discussion on related work in §7. Existing systems either preserve portability by falling back to coarse-grained CPU-initiated communication (e.g., Perplexity Kernels [29] pack tokens on the GPU into contiguous per-destination buffers and issue one RDMA transfer per destination rank), or achieve state-of-the-art performance by specializing to a specific GPU-NIC stack.

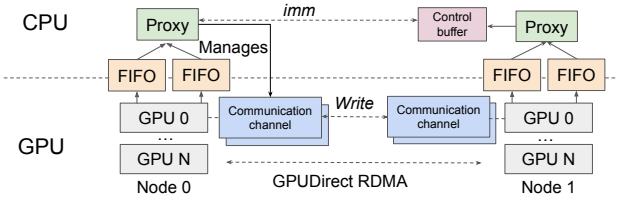


Figure 5: UEP-architecture. Control buffer temporarily buffers control messages (e.g., atomics) until the conditional check specified by the message is passed, upon which the values carried by these control messages are applied. Multiple communication channels are used for moving the data payloads (e.g., tokens) via GPUDirect RDMA.

We argue that the expert-parallel communication system should be portable by *design* - and UEP is based on a set of simple *primitives* that ease portability across various hardware, as well as supporting new future GPUs and NICs with minimal additional overhead.

3 Design

The key observation of UEP is that the GPU only needs to initiate token-level transfers for maximal performance with fine-grained overlapping with other phases of communication (e.g., data copying and computation), while the responsibility for monitoring and managing those transfers does not need to remain on the GPU. CPU, on the other hand, is portable to any GPUs via CUDA/ROCm and any NICs via the `libibverbs` library [48] maintained by the Linux community and NIC vendors, and CPU is also flexible in terms of enforcing various delivery semantics expected by the GPU.

In UEP (Figure 5), GPUs initiate the *fine-grained* data-movement (e.g., of expert token activations) by delegating the transfer tasks to CPU proxies. In effect, GPU threads issue lightweight control messages over PCIe to the CPU; the CPU proxies then intercept these control messages and issue token-level data communication on behalf of the GPU threads. This separation gives us the best of both worlds: the GPU retains token-level and pipelined data communication necessary for high performance, while the CPU proxy abstracts away heterogeneous NIC semantics with portability across platforms.

We discuss two high-level primitives proposed in UEP—an efficient CPU-GPU communication channel (§3.1) and a multi-threaded CPU proxy to delegate and manage GPU-initiated communication (§3.2 and 3.3).

3.1 Efficient CPU-GPU communication channel

UEP employs a lightweight, fixed-size command descriptor, termed `TransferCmd`, that the GPU threads enqueue into shared, lock-free FIFO channels with the CPU proxy as consumers (Figure 6). The GPU acts as the producer: it reserves space by advancing the head and then writes the command into the queue. The CPU proxy acts as the consumer: after processing a command, it advances the tail to make the entry available for reuse. After the GPU side writes to the head of the FIFO

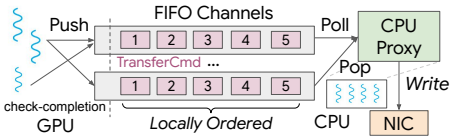


Figure 6: CPU-GPU FIFO channel. The FIFO channel transmits 128-bit `TransferCmd`. The GPU threads (left) enqueue commands to the FIFO channel. The CPU proxy threads (right) either read the commands with `Poll` or dequeue the command with `Pop`. UEP employs multiple FIFO channels per GPU, and each CPU proxy uses multiple threads to read from FIFO channels for scalability.

channel, it then proceeds with data packing and forwarding, while the CPU side reads from the tail of the FIFO buffer and issues the corresponding RDMA work request to the NIC.

Each `TransferCmd` bundles control information necessary for initiating GPU-direct communication, such as destination peer, buffer address, length, and sequence number. This leaves the data payload still residing on the GPU memory. This approach decouples communication initiation (by the GPU) from communication execution (CPU posting the actual RDMA commands).

UEP FIFO channels. Inspired by MSCCL++ [53], UEP uses a 128-bit (16 bytes) descriptor per command to improve the speed of each transfer, as 16 bytes can be written with a single GPU instruction and MMIO doorbell, aligning with PCIe's natural transaction and payload granularity. Since MSCCL++ does not target EP and it assumes strict ordering for the NICs, UEP is significantly different: UEP uses multi-threaded proxies and multiple parallel FIFO channels to make it scalable for small messages (§5.4); UEP further leverages the CPU proxy to enforce delivery semantics among `TransferCmd`s among a subset of FIFO channels as specified by the GPU threads (§3.3) for heterogeneous NICs.

The CPU-GPU channel exposes both CPU-side APIs and GPU-side APIs. UEP bounds the channel size with a parameter `kMaxInflight`. A message can be enqueued from the GPU only if there is space in the buffer (the number of outstanding messages is equal to or smaller than `kMaxInflight`) – otherwise, it will have to wait for the message to be dequeued by the CPU. Therefore, the channel size serves as a way to enforce *control* onto the GPU-initiated communication; this is important for pacing the GPU sender rate. The GPU threads can also, through checking the FIFO channel, know the completion of a prior message (e.g., important for implementing a barrier).

UEP's CPU-GPU communication channel is *one-sided*: communication only flows from GPU to CPU. However, the CPU can rate-limit the GPU by controlling how quickly the messages are consumed from the FIFO buffer. Delaying consumption of the transfer command will make the buffer full, making the GPU thread pause on enqueueing more messages onto the FIFO channel.

The communication channel is split between the host and GPU memory to place each portion of the metadata (e.g., head of the FIFO channel on CPU and tail of the FIFO channel on GPU) where it is most frequently read: the head is set by the GPU but polled by the CPU. The tail is set by the CPU but polled by the GPU. An alternative design is placing both head and tail on either GPU memory or host memory. However, one side (e.g., CPU or GPU) has to incur repeated access over PCIe when busy polling, incurring multiple PCIe crossings and hence heavy latency cost.

Memory consistency. Memory consistency is a key challenge for an efficient CPU-GPU communication channel. If the GPU sees stale data, this can lead to incorrect decisions (e.g., being stalled on a barrier). If the CPU sees stale data, it might read a stale value in the channel (leading to incorrectness). UEP ensures that when the GPU accesses the channel, it will bypass its hardware caches. CPU-side also ensures that the writes are flushed to the host memory rather than buffered in its L2 cache. Alternatively, the cache-coherent C-C interconnect (e.g., in GH200 [42]) can mitigate this issue (Figure 13 in evaluation).

Impact of GPU-side contention. Due to the large number of GPU threads, multiple threads of the same GPU SM naively pushing commands into the same FIFO channel can cause severe contention. To reduce contention, UEP uses multiple FIFO channels and carefully routes each GPU thread to push to the relevant FIFO channel. Cross-FIFO channels ordering is not guaranteed, though commands enqueued within the same FIFO channels are guaranteed to be ordered as they are read sequentially. UEP ensures that the sequence of messages that the GPU kernel requires ordering from the GPU side is mapped to the same FIFO channel.

Channel APIs. We next discuss APIs exposed at both the CPU-side and GPU-side to enable GPU-initiated token-level expert-parallel communication.

CPU-side:

- **Poll**: CPU proxy reads the `TransferCmd` from the channel, without dequeueing the command from the channel.
- **Pop**: CPU proxy removes the `TransferCmd` from the channel, indicating that the `TransferCmd` has been consumed.

GPU-side:

- **Push**: GPU thread enqueues a `TransferCmd` into the FIFO channel and gets a `Idx` for the command.
- **Check-completion(Idx)**: The GPU thread checks whether the `TransferCmd` with `Idx` has been popped from the CPU side.

UEP decouples between initiating the transfer and finishing the transfer. The proxy first becomes aware of the transfer when calling `Poll`, reading the message from the FIFO queue. This allows the CPU proxy thread to initiate a particular operation. For example, the CPU proxy thread can issue an RDMA write after polling a `Write` `TransferCmd` (described later).

The FIFO queue exposes an API for the GPU thread to check completion. This is important for GPU-side operations. For example, the GPU-side might need to wait for a prior dispatch or combine operation to finish to proceed (so as not

to overwrite the transport buffer); this logic often requires the communication layer to initiate a barrier across a subset of participating ranks. In this example, the GPU-side needs to wait for the barrier operation to finish by checking completion on the Barrier `TransferCmd`.

Types of TransferCmd. We added four types of `TransferCmd`s to support GPU-initiated token-level communication. These types are not meant to be an exhaustive list for expert-parallel communication. However, they suffice in our example to support expert-parallel communication in both training and serving.

Write: GPU thread delegates a write request to the CPU proxy thread to execute. We pass in the `addr offset` on the source rank, `addr offset` on the destination rank, the number of bytes to transmit, and the destination rank. Only offsets are needed, rather than a global address, as UEP CPU proxies exchange each other's base address during initiation (§3.2). The write command can optionally piggyback an atomic message to signal completion. This can be useful when GPU threads both want to deliver a data payload to the destination memory address, as well as incrementing a related counter (e.g., the number of the tokens received) at the receiver.

Atomics: GPU thread delegates an atomic operation to the CPU proxy thread to execute. The request contains destination offset, atomic value, and destination rank. This operation should execute atomically with concurrent operations sent by senders from multiple ranks. The atomic operations serve two logical purposes. First, they act as a remote doorbell with a memory-barrier semantics to ensure that previous writes have completed. For example, an atomic write to a flag bit following three token writes can establish that these three writes have been delivered. Second, atomics can also act as a way for the receiver to track how many tokens each expert has received in total. For example, suppose expert 5 on rank 2 receives 2 tokens from rank 5 and 5 tokens from rank 7. A counter for expert 5 must be maintained and updated by both ranks to reflect the total of 7 tokens. Different networking vendors have varying support for atomics; we describe one example in §4.1 on EFA.

Drain: Delegates a CPU proxy thread to drain the RDMA completion queue and ensure that all outstanding RDMA operations have completed. This is critical to ensure that the GPU threads can safely proceed without having their local states overwritten by unfinished requests from previous iterations of dispatch and combine. UEP optionally allows the GPU threads to pass a parameter in the Drain command to drain in-flight messages up to a certain TransferCmd index `Idx`.

Barrier: Barrier message delegates the CPU proxy thread to establish a synchronization barrier: the barrier message can support an all-peer barrier (where all participating peers need to enter the barrier), or a same-rail barrier that, under a rail-optimized topology, synchronizes only the peers mapped to the same rail (typically same GPU index across different nodes). For the former, the UEP Proxy thread creates an intra-node shared memory to enforce a hierarchical barrier—synchronization among local peers first, then across nodes. A per-node leader is established for intra-node barrier, and a cross-node leader (e.g., typically the first node) is established then for inter-node synchronization. For the latter, UEP relies on the RDMA immediate data to signal the leader rank that a barrier request is received, then the leader rank replies with the immediate data to signal the other ranks that a barrier has been successfully established.

Both Barrier and Drain are blocking on the GPU side, meaning that the GPU thread will need to check completions with the correct `TransferCmd Idx` before it can proceed with the `Check-completion(Idx)` API. While more message types can be supported, UEP supports the above four as they are adequate to cover the communication patterns for GPU-initiated EP communication.

3.2 Flexible CPU proxy to delegate communication

UEP keeps one CPU proxy per GPU, and each CPU proxy has multiple threads. While the use of CPU threads is common in collective transfers, such as in NCCL [41] and CPU-assisted IBGDA [38], UEP differs fundamentally from these approaches in three key aspects. First, the communication pattern is GPU-initiated and *token-level*, posing challenges for CPU proxy threads to quickly handle and process messages from GPUs. Second, as the message size is smaller, the number of messages needed to saturate network bandwidth is substantially higher; UEP uses more CPU threads for scalability. Third, UEP's CPU proxy is tasked with bridging the delivery semantics expected by the GPU (§2.4) and provided by the networking layer, and efficiently expressing these requirements without hurting performance is a non-trivial problem.

Multiple threads for each CPU proxy UEP uses up to 4 CPU threads per proxy. Each CPU thread is pinned to a CPU core. We note that modern GPU servers usually have hundreds of CPU cores, which are often heavily underutilized, e.g., 20%-45% CPU utilization reported by several industry companies in their GPU clusters [19, 23].

Establishing connection. With multiple threads in a CPU proxy, UEP bounds the number of connections by only allowing the i-th thread of a CPU proxy to establish connections with the i-th threads of remote CPU proxies. Communication is bidirectional: each thread is in charge of both polling the sender's completion queue for any successful outgoing write or atomic message, and polling the receiver's completion queue to handle any remote incoming requests.

Symmetric memory. The CPU proxy thread maintains the semantics of symmetric memory, as symmetric memory has been noted to be useful by prior works [35, 47, 64]. This enables the GPU thread to only notify CPU threads of the *offsets* of the transfer. CPU threads then handle the address translation, bounds checking before issuing RDMA writes, and atomic messages. Each CPU proxy registers a memory region during initialization, and exchanges the base address during handshake. Using offsets in symmetric memory

reduces the number of bits needed in control messages, compared to passing full addresses. For example, if each GPU exposes only a 2 GB transport buffer for RDMA, and the address is 16-byte aligned, a 27-bit offset is sufficient to identify any byte within the buffer, whereas passing a full virtual address would require 64 bits per transfer, taking up more bits in the TransferCmd. This also eliminates the need to use vendor-specific shared memory libraries, such as NVSHMEM [35] and rocSHMEM [56], improving portability.

Addressing delivery semantics. Heterogeneous NICs exhibit different delivery semantics: for example, some NICs may deliver RDMA writes out of order. To handle this, the CPU proxies embed a sequence number in each RDMA write via immediate data; the receiver uses the sequence number to reorder or delay processing the control messages until all prior writes have arrived. Immediate data is a 32-bit piece of data that RDMA send/write operations can piggyback in the packet header and deliver to remote CPUs over the network. Immediate data field has been standardized into the RoCEv2 packet header [1]. By having CPU proxy interprets this immediate data and enforce guarantees, the system remains correct even when the transport does not guarantee strict delivery guarantees. For comparison, in IBGDA, all transfers *completely* bypass the CPU. We delve into the details of this design next.

3.3 Expressing GPU communication requirements with CPU proxy

Similar to DeepEP [64], UEP presents two modes: low latency (LL) mode and high-throughput (HT) mode. The LL and HT kernels are not specific to one model or UEP itself; they correspond to two general expert-parallel communication modes. LL mode targets decode-style workloads with small batches, where tokens should be sent as soon as routing decisions are available. HT mode targets prefill/training-style workloads with larger batches that benefit from higher bandwidth. LL mode immediately sends the token activation initiated by the GPU via the CPU proxy, requiring no synchronization between transfers. HT mode implements message deduplication, intra-node token forwarding, as well as batching multiple tokens before sending. These optimizations can significantly improve throughput; however, they can have higher latency compared to the LL mode. High-throughput kernel employs multiple communication channels (Figure 5), a set of ring buffers per GPU that temporarily buffer tokens to send in the granularity of chunks (a configurable parameter, typically 32 tokens).

In the remainder of the subsection, we use UEP's low-latency (LL) kernel and high-throughput (HT) kernel as two illustrative examples of UEP's proxy threads enforcing delivery semantics required by GPU kernels. Note that RDMA writes are still immediately applied, though the receiver CPU proxy is notified of any delivered writes through polling the RDMA completion queue. The CPU proxy tracks metadata about these delivered writes, such as the number of writes delivered for a particular expert. Atomic is implemented via the immediate data and is not immediately applied: CPU proxy thread extracts the offset and value for the atomic itself from the 32-bit immediate data, buffers them in the control buffer (Figure 5) if needed, and selectively applies them based on the delivery semantics expected of these atomics.

Low-latency kernel requires partial completion fence. Atomic messages are used to signal token delivery (e.g., X number of writes have completed), requiring that the receiver wait until the required number of writes for a specific expert has finished. This requires *completion fence semantics*: if atomic arrives before X tokens, it should not be applied. However, it does not matter if these X tokens are delivered in order or out-of-order. This guarantee is *partial*: completions of writes to other experts do not affect updating the number of delivered tokens of this expert. Similarly, the arrival order of these X tokens does not matter; so long as these X tokens arrived before applying the atomic.

Solution: UEP lets the CPU proxy temporarily buffer the atomic message in the control buffer until the required number of tokens for a destination expert has been received. To achieve that, UEP packs the destination offset and the expert index into the 32-bit immediate data, and the receiving CPU thread will parse the immediate data, extract the expert index as well as the source rank of the connection. Each subsequent atomic message will go through a lightweight conditional check (has X number of writes to the specified expert being received?): UEP only applies the atomic update when the conditional check has passed to enforce the delivery requirements of these atomic messages.

High-throughput kernel requires partial ordering. The high-throughput kernel employs multiple ring buffers as communication channels per GPU to temporarily buffer and batch tokens to send. Batching improves throughput; however, it also means the GPU should communicate to maintain the head and tail of these ring buffers on remote ranks. Each token is written to a slot in the destination ring buffer, and by carefully controlling the head and tail values of the ring buffer, UEP ensures that no token will overwrite other yet-to-be-read tokens on the same slot of the ring buffer.

A write message is typically followed by an atomic operation to increment either the head or tail values of the ring buffer. If the write and atomic become reordered with other writes and atomics, this can lead to the receiver reading stale data from the communication channel, or the sender overwriting data written by a prior transfer. For example, consider two writes. The first write is issued to slot 4 on the receiver. The second write is issued to slot 5 on the receiver. Both writes carry an atomic to increment the head value of the receiving ring buffer to indicate to the remote kernel that a new message has been delivered. However, if the atomic for the second write arrives first before the atomic for the first write, the remote GPU kernel can read the slot 4, before the delivery of the write to slot 4, leading to reading wrong results.

Similar to LL mode, enforcing ordering only needs to be

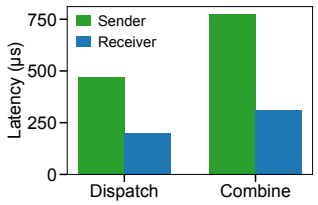


Figure 7: Sender-side vs. receiver-side on enforcing delivery semantics (on testbed NV_EFA3 detailed in Table 2). *Receiver-side* gives better performance compared to sender-side.

partial. The ordering guarantees only need to be done for each communication channel, rather than globally across all messages, which is expensive if not done in hardware.

Solution: UEP ensures that per-channel communication is locally ordered. It achieves this by enqueueing messages from the same communication channel into the same GPU-to-CPU FIFO queue. Similar to the low low-latency kernel, the receiver CPU proxy thread will extract the sequence number from the immediate data; if the received message arrives out-of-order, it will temporarily buffer these atomic messages (that are out-of-order with writes) in the control buffer. Only after the prior writes and atomics have been applied (e.g., message $1-5$ from communication channel i has been applied), the receiver thread will sequentially apply the next buffered atomic message (e.g., atomic with index 6 from communication channel i).

Where to enforce the delivery semantics. An alternative approach is for the *sender*, as opposed to the receiver, to delay sending the atomic messages only after it has received the completion queue entry for the writes sent for that token. This approach is typically used by the NIC hardware to guarantee strict ordering when adaptive routing is used in the network and packet reordering happens [13]. It has the advantage of saving hardware SRAM resources without tracking per-packet states. However, compared to tracking at the receiver, this approach makes the sender wait for one extra RTT for the atomic messages sent, as in this approach, the sender can only issue atomic messages after it polls the completion entry for the prior writes. We have observed suboptimal performance (Figure 7), as the latency penalty of waiting for token completion accumulates.

In summary, the networking layer typically provides either a stronger or weaker guarantee in hardware than what is typically required by the application, e.g., ensuring that *all* messages are strictly ordered, or guaranteeing none at all. From discussing with industry practitioners, installing various guarantees comes at a hardware cost trade-off: supporting strong guarantees at the NIC hardware typically means that the hardware NICs are more expensive to make. In comparison, CPU is flexible in customizing and enforcing various delivery semantics. The minimal requirement for UEP is reliable delivery of RDMA operations (e.g., as commonly provided by RC or SRD protocol), and a way to carry small control metadata to the receiver, such as RDMA immediate data. These requirements are typically provided by existing RDMA NICs.

4 Implementation

We implement UEP by extending DeepEP with 20.8K lines of C++ (including 2.4K lines of CUDA/ROCm C++) and 1K lines of Python, while remaining API-compatible with DeepEP. UEP significantly extends DeepEP in two ways: i) supporting heterogeneous GPUs and NICs, including NVIDIA and AMD GPUs, and NVIDIA, AWS EFA, Broadcom, and Intel NICs (other NIC vendors should be naturally supported via the portable `libibverbs`); and ii) supporting token-level and customizable communication requirements across NICs and EP modes (i.e., LL or HT), as described in §3.3. UEP is architecturally portable: porting to AMD GPUs and AWS EFA NICs is done with *only* 3 person-months, requiring relatively less effort compared to supporting such communication across every NIC and GPU pair, which typically requires dedicated teams.

Removing GPU-vendor-specific software stack. Existing GPU-driven communication stacks often rely on NVSHMEM for device-side synchronization, memory ordering, and GPU-initiated one-sided operations. However, such a software stack assumes a specific GPU vendor and depends on hardware features that may not exist on AMD GPUs or non-IB NICs. UEP instead manages symmetric memory with CPU proxy threads, and expresses NVSHMEM-related GPU-initiated communication APIs via its own CPU proxy.

Queue Pair (QP) load balancing. Each proxy thread for a particular GPU is in charge of managing a set of NICs in the same NUMA group as the GPU (e.g., there are 2×200G EFA NICs per H200 GPU on AWS). In low-latency (LL) mode, each thread creates an RDMA queue pair for a given destination rank; in high-throughput (HT) mode, each thread creates multiple QPs (corresponding to the number of communication channels, §3.3) between pairs of ranks. Depending on communication requirements, the GPU might require a set of messages to be sent out from a single QP, or it does not impose any requirements on which QPs to send, where the CPU thread round-robins among the QPs it manages.

Aggregating NICs of different bandwidths. Beyond QP load balancing, UEP aggregates bandwidth across multiple NICs per GPU: e.g., one ConnectX-7 NIC may deliver 400 Gbps, but achieving similar bandwidth with EFA NICs may require aggregating two 200 Gbps NICs per GPU. UEP relies on CPU threads to load balance across different NICs. We omit the details for brevity.

Interfaces for adding support of new NIC or GPU To add a new NIC, UEP requires the NIC to expose a standard host-side RDMA interface, ideally through libibverbs or a compatible verbs provider. This is common for most, if not all, RDMA-capable NICs. For new NICs, UEP provides a modular interface that allows CPU proxy threads to translate TransferCmd into NIC work requests, manage QPs/CQs, exchange base addresses, and handle NIC-specific limits such as maximum in-flight writes. To add support for a new

Name	# servers	Network	GPU	HBM, # SMs, NVLink	NIC	CPU	Cloud
NV_EFA3	4	Ethernet	NVIDIA H200×8	141 GB, 132, 900 GB/s	AWS EFAv3 200G×16	192 cores	AWS
NV_EFA4	4	Ethernet	NVIDIA B200×8	192 GB, 160, 1800 GB/s	AWS EFAv4 400G×8	192 cores	AWS
NV_IB	4	InfiniBand	NVIDIA H100×8	80 GB, 132, 900 GB/s	NVIDIA ConnectX-7 400G×8	128 cores	Nebius
NV_C2C_IB	2	InfiniBand	NVIDIA GH200×1	96 GB, 132, –	NVIDIA ConnectX-7 200G×1	72 cores	Lambda
AMD_IB	4-16	InfiniBand	AMD MI300X×8	192 GB, 304, 896 GB/s	NVIDIA ConnectX-7 400G×8	128 cores	OCI
AMD_BRC	4	Ethernet	AMD MI300X×8	192 GB, 304, 896 GB/s	Broadcom Thor-2 400G×8	128 cores	Vultr

Table 2: Evaluation testbeds. All testbeds are rented from public cloud providers.

GPU, UEP requires translating GPU-side kernels (e.g., from NVIDIA to AMD) and having the kernel call UEP transport channel APIs (e.g., supporting various TransferCmd types as in §3.1). We next present two case studies to illustrate this.

4.1 Supporting EFA

Emulating atomics with CPU proxy threads. EFA NICs currently do not provide hardware RDMA atomics (e.g., global counters). UEP implements software-based atomics: the sender issues a payload write followed by a small RDMA write carrying an immediate value encoding the new counter or flag; the CPU proxy or receiver thread updates local completion counters allocated on the *host* memory (e.g., via `cudaMallocHost()`) upon detection of the immediate data. UEP carefully ensures that the GPU observes the host-allocated counter and uses it for control decisions. This ensures the correctness of completion notification without relying on vendor-specific atomic support.

4.2 Supporting AMD

UEP generalizes the expert-parallel communication kernels so that they no longer assume NVIDIA-style warps, NVIDIA-specific PTX intrinsics, and hardware engine. In particular, UEP does the following changes:

- Migrating CUDA-specific PTX intrinsics to use ROCm alternatives, including atomics, memory fences, and timers.
- Migrating CUDA warp-based programming to support AMD wavefront, including switching `WARP_SIZE` from 32 to 64, using AMD wavefront-level synchronizations.
- Migrating NVIDIA TMA-based data copy [5] to support AMD CU-based (i.e., compute units, like NVIDIA SMs).
- UEP uses wavefront (i.e., warp in NVIDIA's term) specialization for AMD. For the HT kernel, UEP merges its `coordinator`-role wavefronts into `receiver`-role wavefronts [10], as AMD GPUs usually support fewer wavefronts than NVIDIA warps but more threads per wavefront.

Note that to support AMD GPUs with heterogeneous NICs, only GPU kernels need to be ported, rather than the CPU-side code to operate on heterogeneous NICs; UEP's approach allows us to immediately run on AMD platforms with heterogeneous NICs after AMD GPU-side changes, without having to do independent development between AMD GPU and each individual NIC. This shows that UEP enables $O(m)$ effort, instead of IBGDA's $O(m \times n)$, to support GPU-initiated token-level communication for expert parallelism.

5 Evaluation

Our evaluation aims to answer the following questions:

- How does UEP improve MoE model training? (§5.2.1, §5.2.2)
- How does UEP improve MoE model serving? (§5.2.3)
- How does UEP's performance compare to baselines on heterogeneous devices? (§5.3.1 and §5.3.2)
- How does UEP's performance compare to DeepEP on NVIDIA GPUs and NICs? (§5.3.1)
- How do different design choices (e.g., number of CPU threads) impact UEP performance? (§5.4)

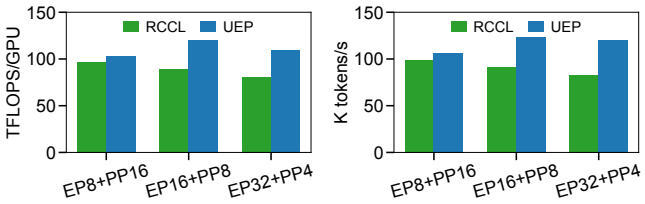
5.1 Methodology

Experimental setups. A list of testbeds can be found in Table 2. We used testbeds from a variety of GPU vendors (NVIDIA and AMD) and NIC vendors (AWS, NVIDIA, Broadcom) to show that UEP is portable across platforms and benchmark UEP's performance.

Baselines.

- **NCCL [41] / RCCL [3].** We use collective communication libraries as a baseline for the EP communication stack on NVIDIA (NCCL) and AMD (RCCL) GPUs.
- **DeepEP [64].** DeepEP is a state-of-the-art, GPU-initiated RDMA communication system for expert-parallel MoE.
- **Perplexity Kernels (PPLX) [29].** We evaluate against Perplexity's custom MoE communication kernels (denoted PPLX), which are highly optimized for low-latency decode.
- **CPU-assisted IBGDA [38].** We emulate a CPU-assisted IBGDA design using a single UEP proxy thread, similar to the existing CPU-assisted IBGDA approach.
- **Mori [50].** Mori is a modular RDMA library; we include it to compare against a ROCm-native RDMA stack.
- **Theoretical Best**: Theoretical results for HT mode derived from available RDMA bandwidth.

To ensure a fair comparison, we use the same amount of GPU resources (e.g., the same number of SMs per GPU as DeepEP, and the same number of CUs as Mori). UEP is designed as a drop-in replacement for DeepEP-style applications. It has already supported various inference frameworks, such as vLLM and SGLang, and training frameworks such as MegatronLM and AMD Primus, with no application-level changes.

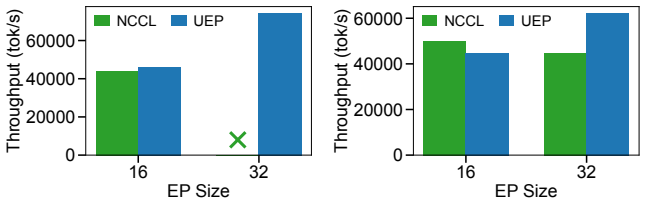


(a) TFLOPS/GPU. **(b)** Tokens/s.

Figure 8: Training throughput of DeepSeek-V3 (downscaled to 32 layers and 379B parameters) under AMD Primus/Megatron-LM [4]. We scale AMD_IB to 16 nodes temporarily for this experiment.

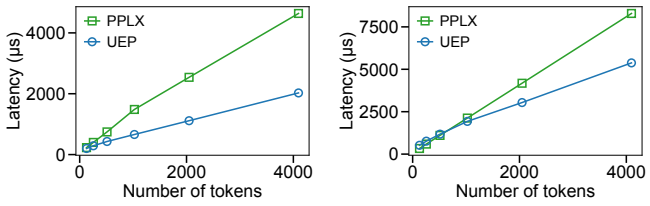
Table 3: MoE training throughput (in TFLOPS/GPU) on NV_EFA3 (32×H200, EFAv3) with Megatron-LM.

Model	NCCL	UEP
Qwen3-235B, long-seq	153.0	**187.0**
Qwen3-235B, large-batch	190.5	**213.9**
DeepSeek-V3	152.2	**188.0**



(a) DeepSeek R1 (671B) **(b)** Qwen3 (235B)

Figure 9: SGLang throughput comparison across two MoE models using DeepEP and NCCL on NV_EFA3.



(a) Dispatch. **(b)** Combine.

Figure 10: EP32 comparison when varying numbers of tokens on NV_EFA3. UEP uses the minimum of HT and LL latency, while PPLX only has one mode.

5.2 Application performance

5.2.1 Training on AMD Primus/Megatron-LM

Figure 8 reports end-to-end Megatron-LM training performance in TFLOPS and tokens per second for DeepSeek-V3 [30] over 16 servers. Across all models, UEP matches or exceeds the TFLOPS (by 7-36%) and throughput achieved by RCCL (by 7-45%). These results show that UEP leads to performance benefits compared to RCCL for Megatron-LM training on AMD.

5.2.2 Training on H200/EFA with Megatron-LM

Table 3 reports end-to-end Megatron-LM training performance in TFLOPS per GPU on NV_EFA3 with 4 servers (32 H200 GPUs over EFAv3). We evaluate three configurations: *Qwen3-235B, long-seq* runs Qwen3-235B-A22B [60] with seq=49152, GBS=8, EP=16, TP=4, PP=2, CP=2; *Qwen3-235B, large-batch* uses the same model with seq=20480, GBS=128, EP=16, TP=4, PP=2, CP=2; and *DeepSeek-V3* [30] truncated to 12 layers (∼135B total, ∼5B activated; 256 experts, top-k=8, hidden=7168) with EP=32. Across all three settings, UEP exceeds the TFLOPS achieved by NCCL by 12–24%: 22% for *Qwen3-235B, long-seq*, 12% for *Qwen3-235B, large-batch*, and 24% for *DeepSeek-V3*. UEP improves Megatron-LM training throughput without modifications to the model code.

5.2.3 Inference on SGLang with EFA

We evaluate UEP in SGLang v0.5.3 with EP set to either 16 or 32 on a prefill-heavy workload (input length 4096, output length 5) using `deepseek-ai/DeepSeek-R1-0528` [18] and `Qwen/Qwen3-235B-A22B-FP8` [60]. The results are presented in Figure 9. We compare against NCCL, as DeepEP does not run on EFA; at the time of writing, PPLX had not been integrated into any open-sourced inference engine. For DeepSeek R1 at EP=16, UEP reaches an input throughput of 46K tok/s, about 5% higher than NCCL. Scaling to EP=32, UEP further improves to 74K tok/s input, a 1.6× speedup over its own EP=16 run. NCCL cannot yet run [9, 57] on EP=32 (confirmed with SGLang maintainers). For Qwen3, at EP=32, UEP reaches 62K tok/s throughput versus 44K tok/s for NCCL (about 40% higher). We also observe that CPU utilization increases modestly with UEP, from an average 8% CPU utilization to 22% utilization.

5.3 Microbenchmark

In this subsection, we compare the dispatch and combine latency on NVIDIA GPUs (§5.3.1) as well as AMD GPUs (§5.3.2). Under each GPU vendor, we compare different NIC setups, including AWS EFA, CX7 with IB, and Broadcom Thor-2 NICs. UEP is able to run across all testbeds. In each testbed, we compare UEP against the available baselines that are specifically optimized for each setup.

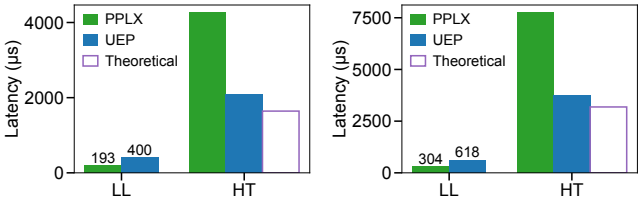
5.3.1 On NVIDIA GPUs

Comparison on AWS EFA. Figure 10 and Figure 11 show EP32 dispatch and combine latency on NV_EFAv3 (on AWS `p5en` instances) as we vary the number of tokens. As we increase the number of tokens, UEP outperforms PPLX and the gap widens: for medium and large batches, UEP consistently delivers substantially lower dispatch (2.3×) and combine latency (1.1-1.5×), demonstrating better scalability as token counts increase.

We would like to make two observations on the LL mode comparison between PPLX and UEP: First, PPLX does not support in-kernel BF16-to-FP8 conversion, assuming that the input data is already passed in as the FP8 data type. This makes com-

Table 4: Dispatch performance on NV_EFA3. w/ Conv additionally adds a small data conversion latency (from BF16 to FP8) for PPLX.

	UEP		PPLX	
	No Batch	Batch	Raw	w/ Conv.
Latency (μs)	216.61	**177.70**	232.30	266.90



(a) Dispatch. **(b)** Combine.

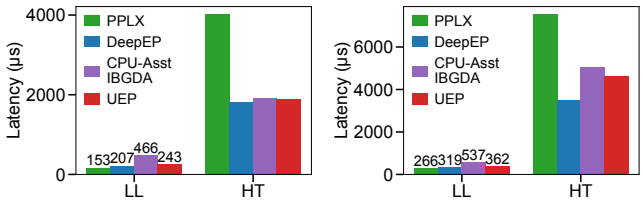
Figure 11: EP32 comparison on NV_EFA4.

parison tricky, as the original DeepEP and many large-scale MoE models use BF16 as input and BF16-to-FP8 conversion is consequently a *key* step to save communication traffic. To be favorable to PPLX, in the figures of this paper, we *ignore* the PPLX data-type conversion latency and directly compare the PPLX dispatch latency (assuming FP8 input) versus UEP dispatch latency (assuming BF16 input), where UEP does an additional step of converting BF16 to FP8 in the GPU kernel. We present a comparison in Table 4. If we add the data type conversion time to PPLX, PPLX's dispatch time rises to 266.9 μs from 232.3 μs. Second, the latest UEP additionally supports best-effort per-expert token batching in the low-latency kernel. On 2 nodes of NV_EFA3, best-effort token batching improves the UEP latency from 216.6 μs to 177.7 μs. Unfortunately, as our rented AWS VMs run out of time, we are not able to further measure these optimizations on a larger scale (EP=32).[3]

Comparison on CX7 with IB. Figure 12 compares EP32 dispatch and combine latency on CX7 with InfiniBand for both low-latency (LL, small tokens) and high-throughput (HT, large tokens) modes. In LL mode, UEP incurs slightly higher latency than DeepEP and PPLX due to the overhead of its CPU proxy on small messages. However, in HT mode, UEP achieves latency comparable to the original DeepEP (within 5% for dispatch) while outperforming PPLX for both dispatch (2.1×) and combine (1.6×), showing that UEP preserves high performance on throughput-oriented workloads.

Comparison on GH200 with NVLink-C2C. Figure 13 reports latency on a single-GPU GH200 node with NVLink-C2C between the CPU and GPU. The HT (high-throughput) mode of DeepEP requires an 8-GPU NVLink/NVSwitch topology and therefore cannot run on this platform, so we only compare LL mode. In this setting, UEP achieves lower transfer latency than the original DeepEP, demonstrating that UEP operates efficiently over a cache-coherent C2C CPU–GPU interconnect,

[3]The results in Figure 11 are without the best-effort per-token batching optimization.



(a) Dispatch. **(b)** Combine.

Figure 12: EP32 comparison on NV_IB.

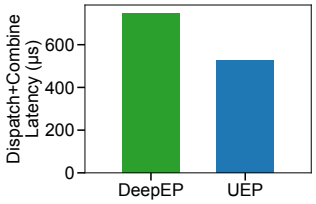
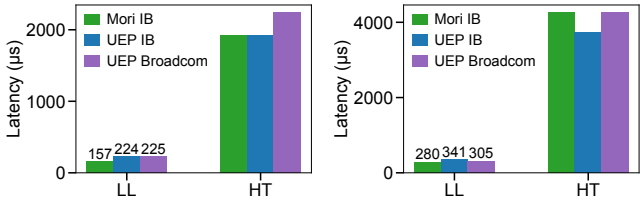


Figure 13: EP2 LL comparison on two GH200 nodes. While EP2 is a less practical setting, it shows the current trend of unified memory between GPUs and CPUs (e.g. with NVLink-C2C) enables us to obtain better performance by enabling fast CPU-GPU communication.



(a) Dispatch. **(b)** Combine.

Figure 14: EP32 comparison on AMD_IB and AMD_BRC.

which we believe to be a future trend. We hypothesize that this benefit comes from two reasons. First, different from DeepEP, which issues a write operation followed by an atomic operation, UEP piggybacks an atomic operation over a write operation with the remote CPU proxy applying the immediate data carried along the write operation, saving one extra message (§5.4). Second, UEP removes the NVSHMEM dependency and its associated software overheads.

5.3.2 On AMD GPUs

Figure 14 shows EP32 dispatch and combine latency on AMD GPUs when using Broadcom NICs (UEP Broadcom) and NVIDIA InfiniBand NICs (UEP IB and Mori IB). At the time of writing, Mori only officially supported NVIDIA NICs. The results demonstrate that UEP runs efficiently across heterogeneous NICs, achieving similar performance on Broadcom and IB in both LL and HT modes. On InfiniBand, UEP IB even outperforms Mori IB in the HT (high-throughput) configuration for combine, while Mori IB has a slightly lower latency than UEP IB in LL mode.

5.4 Design drill-down

Stress testing UEP FIFOs. Figure 15 benchmarks how the latency of the FIFO queue compares to network latency with increasing 7KB message load. The latency incurred along the

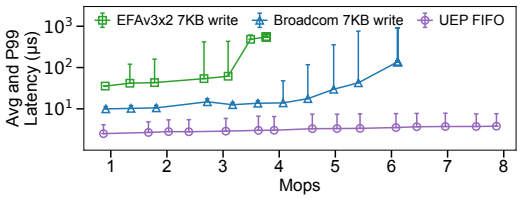
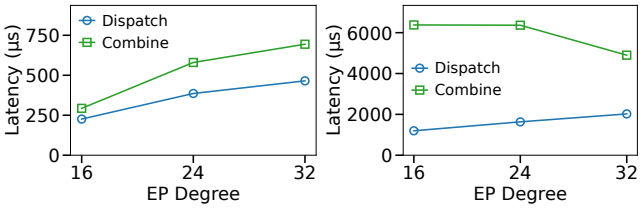
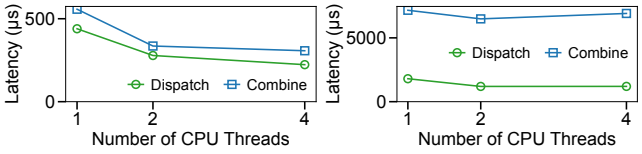


Figure 15: UEP FIFO performance. We run 8 FIFOs simultaneously on NV_EFA3 (each for one GPU) and report the first FIFO's result. Note that the y-axis is in log scale.



(a) LL **(b)** HT

Figure 16: Sensitivity to EP size. We measure the UEP HT and LL mode latency as we increase the EP size from 16 to 32.



(a) LL **(b)** HT

Figure 17: Sensitivity to number of CPU threads (NV_EFA3).

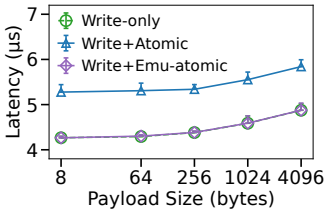


Figure 18: Comparing the latency of emulated atomics across different payload sizes.

FIFO queue is an order of magnitude smaller than the network latency. Furthermore, UEP FIFO queues are able to scale to large QPS (e.g., 8 Mops), capable of handling modern MoE workloads. Compared to the latency of typical dispatch and combine ($\sim$200 μs for LL, and $>$ 2000 μs for HT), CPU proxy (3$\sim$5 μs) introduces negligible latency overhead compared to native GPU-initiated RDMA with IBGDA. Note that this benchmark does not batch requests from GPUs to CPUs; we believe that with lightweight batching, it is feasible to handle the next-generation 800G network [7].

Varying EP degrees. Figure 16 shows that UEP's dispatch and combine latency increases modestly as we increase the EP size from 16 to 32. The trend holds in both LL and HT modes; under the HT mode, combine latency stays flat (and even dropping at EP=32). Overall, UEP scales gracefully across the 16–32 EP range we evaluated.

Varying number of proxy threads. Figure 17 shows that LL and HT kernels suffer from suboptimal performance when the number of CPU threads is 1. The performance significantly improves when we add more CPU threads (e.g., to 4). With more threads per CPU proxy, UEP is able to use more cores to drive higher-throughput communication. We note that CPU utilization is typically low, as GPU servers typically have a large number of CPU cores (Table 2); we observe that even with 4 CPU threads, the CPU utilization only increases modestly.

Latency impact of using emulated atomics. Figure 18 quantifies the latency impact of UEP's emulated atomics, compared against a plain RDMA write and a write followed by an RDMA atomic operation (e.g., in DeepEP). All three paths are issued in a chain RDMA doorbell with the data write unsignaled, so the receiver never waits between operations. Across the write payload sizes from 8 B to 4 KiB, the emulated path follows closely to the write-only baseline: the atomic update by the CPU proxy is essentially free compared to the network delay. In contrast, the write followed by an RDMA atomic costs a roughly $\sim$1 μs extra, because the atomic is an additional operation that the responder NIC must execute. UEP avoids the additional message by piggybacking the atomic update on the immediate data and letting the remote CPU proxy apply it upon polling the receiver work completion entry.

6 Discussion and Future Work

Congestion control with CPU proxy. We observe that the number of outstanding requests can have a significant impact on various NICs, affecting tail latency [46, 66]. This becomes increasingly significant as the number of destinations increases, where having one straggler can significantly slow down dispatch and combine time. Software-based flow control is challenging to implement with IBGDA, as GPU threads typically are not flexible enough for implementing network-level policies.

Instead, UEP delegates *control decisions* to the flexible CPU proxy, which could easily support request tracking and pacing. If the outstanding requests become high, the CPU proxy thread temporarily buffers the messages at the sender, so that the messages will not cause an incast at the receiver side. The CPU proxy also bears responsibility for multi-QP management: because each GPU may be connected to multiple NICs or multiple QPs, the proxy can throttle or shard the outgoing requests across NICs and QPs to avoid congestion [65], and adapt to NIC-specific characteristics (e.g., maximum outstanding write requests) without burdening the GPU kernel logic. This separation allows exploiting multiple NICs transparently and optimizing throughput while still keeping the GPU's logic simple.

Elastic EP with CPU proxy. NVSHMEM, including IBGDA, CPU-assisted IBGDA, and IBRC, does not expose error handling to GPU kernels and assumes RDMA operations always succeed. However, events such as GPU failure, scale up, and down can happen in practical scenarios, where

NVSHMEM would need to restart the existing cluster. UEP could achieve elastic EP by using the flexible CPU proxy to handle failure and scaling events, transparently masking them from the GPU kernel.

More efficient low-latency kernel. DeepEP's low-latency kernel (without token deduplication or hierarchical reduce) could be further optimized. This would particularly benefit AWS EFA NICs that cannot process small message transfers efficiently (see Figure 15). UEP currently extends DeepEP LL kernels, which we plan to further improve as future work. We deem these optimizations as orthogonal to UEP's contribution in a portable expert-parallel communication architecture by decoupling NICs and GPUs.

Intended deployment mode for UEP. UEP is primarily designed for platforms where IBGDA is unavailable or impractical, such as non-NVIDIA GPUs or NICs without the required GPU-initiated RDMA support. However, UEP can also run on IBGDA-capable platforms, where the CPU proxy can provide additional flexibility for communication management.

Failure semantics. In practice, the CPU and GPU on the same server often fate-share: CPU processes are already responsible for launching GPU kernels, and widely used systems such as NCCL also rely on host CPU threads to initiate and coordinate communication. Thus, UEP does not introduce a separate failure domain.

7 Related Work

EP communication systems mainly have two categories.

CPU-initiated. CPU-initiated communication, such as NCCL [41], RCCL [3] (for AMD GPUs), and PPLX [29] relies on the CPU to initiate communication, including constructing and posting RDMA work requests, polling completion queues, and managing QP state transitions. Nearly all MoE training and serving frameworks support NCCL/RCCL for MoE communication. However, as mentioned in §2.2, they would require either re-packing tokens or transferring small tokens, and do not support token deduplication and hierarchical reduce, both leading to low performance as the number of tokens increases. Recent efforts such as UCCL-Tran [65] and NCCLX [55] preserve NCCL/RCCL's CPU-driven design, with various performance optimizations: UCCL-Tran focuses on CPU-driven collective transport optimizations such as software packet spraying, while UEP targets expert-parallel GPU-initiated token-level communication. NCCLX introduces topology-aware collective algorithms, zero-copy, and SM-free transfers. Both are orthogonal to UEP.

GPU-initiated. GPU-initiated communication, by contrast, issues network transfers directly from the GPU, enabling fine-grained token-level operations like deduplication and hierarchical reduce that are critical to EP communication efficiency. DeepEP [64], Mooncake-EP [25], Mori-EP [50], and ROCm-DeepEP [49] (based on rocSHMEM [56]) belong to this category. Although providing high performance, they only work for either NVIDIA GPUs or AMD GPUs, and require strict ordering for the underlying NICs. NVIDIA's recent HybridEP [39] atop of DeepEP supports intra-node and multi-node NVLink scenarios with reduced SM usage, while UEP focuses on inter-node RDMA scenarios.

MSCCL++ [53] adopts the GPU-initiated approach to implement NCCL/RCCL collectives such as all-reduce and reduce-scatter, but does not support irregular EP communication. The CPU-assisted IBGDA [38] and IBRC [31] transports from NVSHMEM [35] could theoretically support non-NVIDIA GPUs via their CPU proxy designs. But they suffer from lower performance for small token activations due to single-threaded proxies and assume strict ordering for the NICs. The EFA transport [36] from NVSHMEM supports AWS EFA NICs, but suffers from poor performance [29] by using a single-threaded proxy. Compared to them, UEP supports both heterogeneous GPUs and NICs, enables fine-grained token-level operations, and provides high performance even with small tokens.

8 Conclusion

Modern MoE workloads require fast and scalable expert-parallel communication, yet existing systems that support token-level GPU-initiated communication remain tied to NVIDIA-specific GPU-initiated RDMA, limiting portability across the increasingly heterogeneous accelerators and NICs. Such an approach results in brittle systems: the communication kernel requires strict delivery semantics from the underlying networking stack, and with limited visibility into error handling, congestion control, and network management. This paper introduces UEP, a portable EP communication system that achieves DeepEP-level performance without relying on specialized GPU–NIC integrations. Our implementation across NVIDIA and AMD GPUs and multiple NIC vendors shows that UEP outperforms the best existing EP solution by 2.1×. UEP is a drop-in replacement for DeepEP-compatible applications such as SGLang and AMD Primus/Megatron-LM, improving token throughput by up to 40% on an NVIDIA+EFA platform, and DeepSeek-V3 training throughput by up to 45% on a 16-node AMD+Broadcom platform.

Acknowledgement

We sincerely thank the anonymous reviewers and our shepherd for their insightful comments. We would also like to thank Chon Lam Lao for helpful discussions. This work is in part supported by gifts from Accenture, AMD, Anyscale, Broadcom, Cisco, Google, IBM, Intel, Intesa Sanpaolo, Lambda, Lightspeed, Mibura, Microsoft, NVIDIA, Samsung SDS, and SAP. We additionally acknowledge AWS, in particular Jun Wu, Yida Wang, Brian Barrett, and Nafea Bshara, for their sponsorship and partnership in this research. Costin Raiciu was partly funded by HRIA (project no. 351416).

References

[1] Infiniband trade association — ibta specification portal. Accessed 2025.

[2] M. AI. Introducing llama 4: Advancing multimodal intelligence. https://ai.meta.com/blog/llama-4-multimodal-intelligence/, 2025. Accessed: 2025-12-08.

[3] AMD. ROCm Communication Collectives Library (RCCL). https://github.com/ROCm/rccl, 2025.

[4] AMD-AGI/Primus. AMD Primus training framework for large-scale foundation model training and inference on AMD GPUs. https://github.com/AMD-AGI/Primus.

[5] Andersch, Michael and Palmer, Greg and Krashinsky, Ronny and Stam, Nick and Mehta, Vishal and Brito, Gonzalo and Ramaswamy, Sridhar. Nvidia hopper architecture in-depth. https://developer.nvidia.com/blog/nvidia-hopper-architecture-in-depth/, Mar. 2022. NVIDIA Developer Blog; Accessed: 2025-12-06.

[6] N.-L. Contributors. Benchmarking deepep guide #1721. https://github.com/nvidia/megatron-lm/issues/1721, 2025. GitHub issue, accessed 2025.

[7] N. Corporation. NVIDIA Ethernet SuperNICs: Next-generation networking for the next wave of AI. https://www.nvidia.com/en-us/networking/products/ethernet/supernic/, 2025.

[8] D. Dai, C. Deng, C. Zhao, R. Xu, H. Gao, D. Chen, J. Li, W. Zeng, X. Yu, Y. Wu, Z. Xie, Y. Li, P. Huang, F. Luo, C. Ruan, Z. Sui, and W. Liang. Deepseekmoe: Towards ultimate expert specialization in mixture-of-experts language models. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Long Papers)*, pages 1280–1297. Association for Computational Linguistics, 2024. DeepSeekMoE model paper.

[9] DavideHe. [usage]: deepseek v3 cannot set tensor_parallel_size=32 (issue #12256). https://github.com/vllm-project/vllm/issues/12256, Jan. 2025. GitHub issue on the vllm-project/vllm repository. Accessed: 2025-12-10.

[10] deepseek ai. DeepEP: `internode.cu` at commit `b57e5e212ab75350f53c72064333e4fe1076b1da`. https://github.com/deepseek-ai/DeepEP/blob/b57e5e212ab75350f53c72064333e4fe1076b1da/csrc/kernels/internode.cu#L1741, 2023. Accessed: 2025-12-09.

[11] deepseek-ai. Eplb: Expert parallelism load balancer. https://github.com/deepseek-ai/EPLB, 2025. GitHub repository, last accessed YYYY-MM-DD.

[12] DeepSeek-AI, A. Liu, A. Mei, B. Lin, B. Xue, B. Wang, B. Xu, B. Wu, B. Zhang, C. Lin, C. Dong, C. Lu, C. Zhao, C. Deng, C. Xu, C. Ruan, D. Dai, D. Guo, D. Yang, ..., Z. Gu, Z. Zhu, Z. Li, and Z. Zhang. Deepseek-v3.2: Pushing the frontier of open large language models. *arXiv*, abs/2512.02556, 2025.

[13] J. Dong, B. Luo, J. Zhang, P. Zhang, F. Feng, Y. Zhu, A. Liu, Z. Chen, Y. Shi, H. Jiao, et al. Boosting Large-Scale Parallel Training Efficiency with C4: A Communication-Driven Approach. *arXiv preprint arXiv:2406.04594*, 2024.

[14] N. Du, Y. Huang, A. M. Dai, S. Tong, D. Lepikhin, Y. Xu, M. Krikun, Y. Zhou, A. Wei Yu, O. Firat, B. Zoph, L. Dixon, Z. Chen, and C. Cui. Efficient scaling of language models with mixture-of-experts. *arXiv preprint arXiv:2112.06905*, 2022. GLaM (Generalist Language Model) introduced by Google as a sparse MoE model.

[15] T. Gale, D. Narayanan, C. Young, and M. Zaharia. Megablocks: Efficient sparse training with mixture-of-experts. *Proceedings of Machine Learning and Systems*, 5:288–304, 2023.

[16] G. D. . Google. Gemini 3 pro, 2025. Released November 18, 2025.

[17] T. Griggs, X. Liu, J. Yu, D. Kim, W.-L. Chiang, A. Cheung, and I. Stoica. M\'elange: Cost efficient large language model serving by exploiting gpu heterogeneity. *arXiv preprint arXiv:2404.14527*, 2024.

[18] D. Guo, D. Yang, H. Zhang, J. Song, R. Zhang, R. Xu, Q. Zhu, S. Ma, P. Wang, X. Bi, et al. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning. *arXiv preprint arXiv:2501.12948*, 2025.

[19] Q. Hu, Z. Ye, Z. Wang, G. Wang, M. Zhang, Q. Chen, P. Sun, D. Lin, X. Wang, Y. Luo, et al. Characterization of large language model development in the datacenter. In *Proceedings of USENIX NSDI*, pages 709–729, 2024.

[20] S. Jaiswal, K. Jain, Y. Simmhan, A. Parayil, A. Mallick, R. Wang, R. St Amant, C. Bansal, V. Rühle, A. Kulkarni, et al. Serving models, fast and slow: optimizing heterogeneous llm inferencing workloads at scale. *arXiv e-prints*, pages arXiv–2502, 2025.

[21] A. Q. Jiang, A. Sablayrolles, A. Roux, A. Mensch, B. Savary, C. Bamford, D. S. Chaplot, D. de las Casas, E. Bou Hanna, F. Bressand, G. Lengyel, G. Lample, L. R. Lavaud, L. Saulnier, M.-A. Lachaux, P. Stock, and S. Subramanian. Mixtral of experts. *arXiv preprint arXiv:2401.04088*, 2024. Mixtral 8×7B sparse Mixture-of-Experts model by Mistral AI.

[22] Y. Jiang, R. Yan, and B. Yuan. Hexgen-2: Disaggregated generative inference of llms in heterogeneous environment. *arXiv preprint arXiv:2502.07903*, 2025.

[23] Y. Jiang, Y. Zhu, C. Lan, B. Yi, Y. Cui, and C. Guo. A unified architecture for accelerating distributed {DNN} training in heterogeneous {GPU/CPU} clusters. In *Proceedings of USENIX OSDI*, pages 463–479, 2020.

[24] C. Jin, Z. Jiang, Z. Bai, Z. Zhong, J. Liu, X. Li, N. Zheng, X. Wang, C. Xie, Q. Huang, W. Heng, Y. Ma, W. Bao, S. Zheng, X. Zheng, Y. Peng, H. Lin, X. Liu, X. Jin, and X. Liu. Megascale-moe: Large-scale communication-efficient training of mixture-of-experts models in production. In *Proceedings of the 21st European Conference on Computer Systems*, pages 366–382. ACM, 2026.

[25] kvcache ai. Mooncake-ep: Expert-parallel extension of mooncake. https://github.com/kvcache-ai/Mooncake/tree/main/mooncake-ep, 2025. Accessed: 2025-12-11.

[26] kvcache-ai/Mooncake contributors. mlx5_ifc.h — mooncake ibgda mlx5 interface header. https://github.com/kvcache-ai/Mooncake/blob/main/mooncake-ep/include/mooncake_ibgda/mlx5_ifc.h, 2025. Accessed: 2025-12-11.

[27] A. Langer, S. Howell, A. Goel, P. Markthub, H. Petty, and F. Oh. Enhancing application portability and compatibility across new platforms using nvidia magnum io nvshmem 3.0. *NVIDIA Technical Blog*, Sept. 06 2024.

[28] R. Li, R. Du, Z. Chu, S. Zhao, C. Han, Z. Shi, Y. Shao, H. Han, L. Huang, Z. Liu, et al. Taming the chaos: Coordinated autoscaling for heterogeneous and disaggregated llm inference. *arXiv preprint arXiv:2508.19559*, 2025.

[29] N. Licker, K. Hu, V. Zaytsev, and L. Chen. Rdma point-to-point communication for llm systems. *arXiv preprint arXiv:2510.27656*, 2025.

[30] A. Liu, B. Feng, B. Xue, B. Wang, B. Wu, C. Lu, C. Zhao, C. Deng, C. Zhang, C. Ruan, et al. DeepSeek-V3 Technical Report. *arXiv preprint arXiv:2412.19437*, 2024.

[31] P. Markthub, J. Dinan, S. Potluri, and S. Howell. Improving network performance of hpc systems using nvidia magnum io, nvshmem and gpudirect async, Nov 22 2022.

[32] Y. Mei, Y. Zhuang, X. Miao, J. Yang, Z. Jia, and R. Vinayak. Helix: Serving large language models over heterogeneous gpus and network via max-flow. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1*, pages 586–602, 2025.

[33] Meta AI. Introducing llama 4: Multimodal intelligence with mixture-of-experts. https://ai.meta.com/blog/llama-4-multimodal-intelligence/, 2025. Meta's official announcement of LLaMA 4 Scout and Maverick.

[34] N. Muennighoff, L. Soldaini, D. Groeneveld, et al. Olmoe: Open mixture-of-experts language models. *arXiv preprint arXiv:2409.02060*, 2024. Open Mixture-of-Experts models by the Allen Institute.

[35] NVIDIA. Nvshmem. https://developer.nvidia.com/nvshmem, 2025. Accessed: 2025-12-06.

[36] NVIDIA. Nvshmem: Libfabric transport backend (efa support). https://github.com/NVIDIA/nvshmem/blob/9cc869bc28e565e6944c4ddf76976ada4a1ebbf7/src/modules/transport/libfabric/libfabric.h#L192, 2025. Implements NVSHMEM support for AWS EFA via the Libfabric transport layer. Commit: 9cc869bc28e565e6944c4ddf76976ada4a1ebbf7.

[37] NVIDIA Corporation. ConnectX-7 400G Adapters. https://resources.nvidia.com/en-us-accelerated-networking-resource-library/connectx-7-datasheet, 2024.

[38] NVIDIA Corporation. CPU-assisted InfiniBand GPU Direct Async. https://developer.nvidia.com/blog/enhancing-application-portability-and-compatibility-across-new-platforms-using-nvidia-magnum-io-nvshmem-3-0/#cpu-assisted_infiniband_gpu_direct_async%C2%A0, 2024.

[39] NVIDIA Corporation. Hybridep for high-performance intra-node token dispatching. https://github.com/deepseek-ai/DeepEP/tree/hybrid-ep, 2024. Technical documentation published within the DeepEP repository.

[40] NVIDIA Corporation. Megatron-LM & Megatron-Core. https://github.com/NVIDIA/Megatron-LM, 2025.

[41] NVIDIA Corporation. NVIDIA Collective Communications Library (NCCL). https://github.com/NVIDIA/nccl, 2025.

[42] NVIDIA Corporation. Nvidia gh200 grace hopper superchip. https://www.nvidia.com/en-us/data-center/grace-hopper-superchip/, 2025. Accessed: 2025-12-08.

[43] NVIDIA Corporation. Using the NVSHMEM InfiniBand GPUDirect Async Transport. https://docs.nvidia.com/nvshmem/api/using.html#using-the-nvshmem-infiniband-gpudirect-async-transport, 2025.

[44] NVIDIA/nvshmem contributors. mlx5_ifc.h — nvshmem mlx5 transport interface header. https://github.com/NVIDIA/nvshmem/blob/devel/src/modules/transport/common/mlx5_ifc.h, 2025. Accessed: 2025-12-11.

[45] OpenAI. Introducing gpt-oss: Openai's open-weight reasoning models. https://openai.com/index/introducing-gpt-oss/, 2025. Accessed: 2025-12-08.

[46] Perplexity AI. Enabling trillion-parameter models on aws efa. https://research.perplexity.ai/articles/enabling-trillion-parameter-models-on-aws-efa, November 2025. Accessed: 2025-12-10.

[47] PyTorch. Symmetric memory. https://docs.pytorch.org/docs/stable/symmetric_memory.html, 2025. Accessed: 2025-12-06.

[48] RDMA Consortium. *libibverbs: RDMA Userspace Verbs API*. Linux RDMA Project, 2024. Accessed: 2025-12-06.

[49] ROCm. Deepep: a high-performance expert-parallel communication library. https://github.com/ROCm/DeepEP, 2025. Accessed: 2025-12-05.

[50] ROCm. MORI: Modular rdma interface. https://github.com/ROCm/mori, 2025. Accessed: 2025-12-05.

[51] ROCm/mori contributors. bnxt_re_hsi.h — mori rdma provider bnxt hsi header. https://github.com/ROCm/mori/blob/main/include/mori/core/transport/rdma/providers/bnxt/bnxt_re_hsi.h, 2025. Accessed: 2025-12-11.

[52] A. W. Services. Elastic Fabric Adapter. https://aws.amazon.com/hpc/efa/, 2025.

[53] A. Shah, A. Jangda, B. Li, C. Rocha, C. Hwang, J. Jose, M. Musuvathi, O. Saarikivi, P. Cheng, Q. Zhou, R. Dathathri, S. Maleki, and Z. Yang. Msccl++: Rethinking gpu communication abstractions for cutting-edge ai applications, 2025.

[54] L. Shalev, H. Ayoub, N. Bshara, and E. Sabbag. A Cloud-Optimized Transport Protocol for Elastic and Scalable HPC. *IEEE micro*, 40(6):67–73, 2020.

[55] M. Si, P. Balaji, Y. Chen, C.-H. Chu, A. Gangidi, S. Hasan, S. Iyengar, D. Johnson, B. Liu, J. Ren, et al. Collective communication for 100k+ gpus. *arXiv preprint arXiv:2510.20171*, 2025.

[56] A. R. Team. rocSHMEM: Gpu-centric openshmem runtime for amd rocm. https://github.com/ROCm/rocSHMEM, 2025. Accessed: 2025-12-07.

[57] TexasRangers86. [bug] deepseek-r1 with 4*a100 got error (issue #3491). https://github.com/sgl-project/sglang/issues/3491, Feb. 2025. GitHub issue on the sgl-project/sglang repository. Accessed: 2025-12-10.

[58] The SGLang Team. Deploying deepseek with pd disaggregation and large-scale expert parallelism on 96 h100 gpus. https://lmsys.org/blog/2025-05-05-large-scale-ep/, 2025. Accessed 2025.

[59] vLLM Documentation Team. Expert parallel deployment. https://docs.vllm.ai/en/latest/serving/expert_parallel_deployment/, 2025. Accessed 2025.

[60] A. Yang, A. Li, B. Yang, B. Zhang, B. Hui, B. Zheng, B. Yu, C. Gao, C. Huang, C. Lv, et al. Qwen3 technical report. *arXiv preprint arXiv:2505.09388*, 2025.

[61] S. Zhang, N. Zheng, H. Lin, Z. Jiang, W. Bao, C. Jiang, Q. Hou, W. Cui, S. Zheng, L.-W. Chang, et al. Comet: Fine-grained computation-communication overlapping for mixture-of-experts. *arXiv preprint arXiv:2502.19811*, 2025.

[62] Y. Zhang, H. Shen, R. Yang, D. Tian, Y. Luo, M. Zhang, L. Li, C. Hu, T. Wo, C. Song, et al. Cauchy: A cost-efficient llm serving system through adaptive heterogeneous deployment. 2025.

[63] C. Zhao, C. Deng, C. Ruan, D. Dai, H. Gao, J. Li, L. Zhang, P. Huang, S. Zhou, S. Ma, et al. Insights into deepseek-v3: Scaling challenges and reflections on hardware for ai architectures. In *Proceedings of the 52nd Annual International Symposium on Computer Architecture*, pages 1731–1745, 2025.

[64] C. Zhao, S. Zhou, L. Zhang, C. Deng, Z. Xu, Y. Liu, K. Yu, J. Li, and L. Zhao. Deepep: an efficient expert-parallel communication library. https://github.com/deepseek-ai/DeepEP, 2025.

[65] Y. Zhou, Z. Chen, Z. Mao, C. Lao, S. Yang, P. G. Kannan, J. Gao, Y. Zhao, Y. Wu, K. You, et al. An extensible software transport layer for gpu networking. *arXiv preprint arXiv:2504.17307*, 2025.

[66] R. Zhu, Z. Jiang, C. Jin, P. Wu, C. A. Stuardo, D. Wang, X. Zhang, H. Zhou, H. Wei, Y. Cheng, et al. Megascale-infer: Efficient mixture-of-experts model serving with disaggregated expert parallelism. In *Proceedings of the ACM SIGCOMM 2025 Conference*, pages 592–608, 2025.

BatchGen: An Architecture for Scalable and Efficient Batch Inference

Tairan Xu[†*], Leyang Xue[†*], Zhan Lu[†*], Jinfu Deng[‡], Hongyang Xiao[‡], Yinsicheng Jiang[†], Congjie He[†], Matej Sandor[†], Le Xu[†], Luo Mai[†]

[†]*University of Edinburgh,* [‡]*Tencent*

Abstract

Batch inference has become a central mode of AI computation, yet existing inference engines still rely on execution models designed for interactive serving. When scaled to millions of sequences, batch workloads reveal two fundamental requirements: the ability to handle extreme inter- and intra-sequence load variation that emerges only at runtime, and the ability to sustain high utilization across large fleets of GPUs. Existing systems fail to meet these requirements, losing substantial fractions of achievable throughput.

We introduce a new architectural foundation for batch inference: the sequence coroutine compute model, which represents each sequence as a fine-grained, event-driven coroutine. This model exposes expressive primitives that allow the runtime to reorganize work dynamically, enabling larger expert-level batches, mitigating stragglers, reallocating work across devices, and maintaining utilization even on cost-effective or memory-constrained GPUs. Building on this abstraction, we implement BatchGen, a production-ready system that uses the coroutine model at cluster scale. On a 128-GPU cluster, BatchGen reduces batch completion time by up to 2.3×, and on memory-constrained accelerators it outperforms the strongest offloading baseline by up to 9.6×.

1 Introduction

Batch inference has become one of the fastest-growing and most resource-intensive modes of AI computation [28, 58]. Offline inference pipelines [14, 53], synthetic data generation [26, 40], model evaluation [2, 5, 24], test-time scaling [44, 45], and RL rollouts [36] now dominate the compute budgets of major AI deployments. Unlike interactive serving [22, 32, 56], these workloads optimize batch completion time (BCT), operate at massive sequence scales [1, 34], and often run on memory-limited accelerators.

Batch inference introduces fundamentally new system requirements. A batch engine must (1) dynamically increase expert-level batch sizes for sparse models, (2) continuously rebalance load as long-tail sequences emerge, and (3) sustain high utilization across large fleets of GPUs. Crucially, these requirements arise directly from how modern AI is being scaled. State-of-the-art models rely on extreme sparsity, especially large Mixture-of-Experts (MoE) architectures, where tokens activate only a few experts among hundreds. Even million-token global batches decompose into small per-expert batches that fall far below GPU-saturating throughput. Meanwhile, test-time scaling and heavy reasoning workloads generate persistent long-tail sequences, creating stragglers that dominate BCT and leave many GPUs idle. These structural properties – expert sparsity, modular computation, and heavy-tailed generation – define a new operating mode that demands a scheduler capable of reorganizing computation at fine granularity.

Existing systems cannot meet these requirements because they inherit a latency-first execution model from interactive serving. Systems such as vLLM [22], SGLang [56], and TensorRT-LLM [32] statically bind each sequence's computation and KV state to a fixed GPU, executing forward passes atomically to minimize per-sequence latency. This prevents pausing between neural modules, forming larger expert batches at runtime, or redistributing load as long-tail sequences appear. Disaggregated designs [57, 60] retain fixed placement assumptions and thus continue to underutilize GPUs. Even throughput-oriented systems [7, 59] inherit the same static scheduling model and cannot address the root causes of insufficient expert batching or decoding stragglers.

Our key intuition is that sequences themselves provide the right granularity for dynamic scheduling. Neural networks expose natural yield points at module boundaries, and per-sequence state is compact and migratable. This motivates a shift in design: treat sequences not as long-lived tasks pinned to devices, but as fine-grained, event-driven coroutines that can pause, resume, combine, partition, and migrate across GPUs. This architecture directly addresses the structural inefficiencies of batch inference. Specifically, this paper makes the following contributions:

(1) Sequence coroutine compute model. Each sequence is represented as a coroutine that carries all state required for correct execution. The model provides four expressive primitives, `yield`, `combine`, `partition`, and `migrate`. These operations allow the system to reorganize computation while preserving correctness. They make it possible to pause execu-

[*]Co-leading authors.

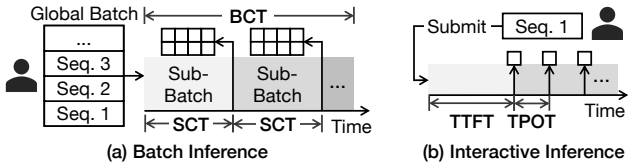


Figure 1: New system design goals for batch inference.

tion and combine sequences to form larger batches for sparse modules, partition stragglers across idle GPUs, and migrate coroutine state for effective load balancing. Existing inference architectures cannot naturally express these behaviors.

(2) Efficient sequence coroutine runtime. The runtime implements event-driven scheduling at cluster scale and is grounded in two key observations. Prefill and decode place very different demands on memory and therefore benefit from different strategies for managing coroutine state. Meanwhile, both phases can run within a unified execution framework that jointly schedules, combines, and partitions coroutines over a shared pool of GPUs. This approach maximizes utilization and avoids leaving devices idle when workloads fluctuate.

(3) BatchGen, a production-ready system that utilizes this design. BatchGen supports multi-node and multi-GPU execution, provides a compatible and expressive batch inference interface, and includes a checkpoint management service that enables rapid cold starts. These capabilities are essential for practical deployment, especially when resource availability changes over time.

(4) Extensive evaluation of the system. Across MoE models, test-time scaling workloads, and reinforcement learning rollouts, BatchGen delivers strong and consistent improvements. On a 128-GPU cluster, it increases batching efficiency for expert layers, reduces the impact of long tail decoding stragglers [16], and lowers batch completion time by up to 2.3×. It outperforms leading inference engines [22, 32, 56] even on memory-constrained accelerators.

BatchGen has been deployed on production clusters to support batch-inference applications. We have open sourced BatchGen at `https://github.com/batchgen-project/batchgen` to benefit the community. We believe the event-driven sequence coroutine architecture represents a fundamental shift for batch inference, akin to the transition from thread-per-request servers (e.g., Apache [3]) to event-driven architectures (e.g., NGINX [31]) that redefined large-scale web systems.

2 Background and Motivation

2.1 Batch Inference and Its Key Objective

Major vendors now provide dedicated batch inference services, including OpenAI Batch Inference [34], AWS Bedrock batch processing [1], and Azure batch endpoints [29]. These services follow a simple model: users submit a batch of input sequences, such as prompts for LLMs, and receive the outputs only after the entire batch has finished processing. Depending on the application, batch sizes can vary widely, from tens of sequences in test-time scaling or thousands in reinforcement learning rollouts to millions in large-scale offline inference.

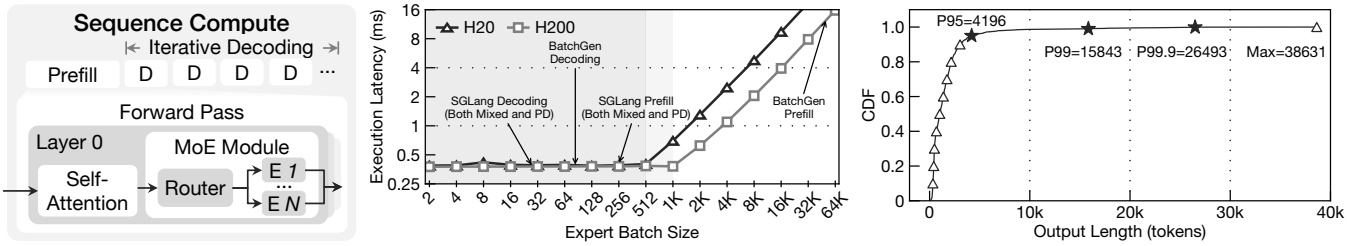
Batch inference systems focus on minimizing the completion time of the entire batch, which is fundamentally different from the responsiveness metrics emphasized in interactive inference. As shown in Figure 1, we define the *batch completion time* (BCT) as the total time from batch submission until every sequence in the batch has been processed. When an application requires partial or early results, the system divides the input into sub-batches, and returns outputs incrementally as each sub-batch finishes. In these cases, we define the *sequence completion time* (SCT) as the time from batch submission until a specific sequence's output becomes available. SCT is determined by the completion time of the sub-batch that contains the sequence. Our primary optimization goal is to minimize BCT, since the user receives the full result only when the entire batch is done. In contrast, interactive inference prioritizes user-perceived responsiveness, focusing on metrics such as time to first token (TTFT) and time per output token (TPOT).

2.2 The Missing Batch Inference System

As adoption of batch inference increases across both industry and academic workflows, the demand for an efficient batch inference engine becomes even more evident. However, there are several challenges in scaling batch workloads: (i) batches continue to grow in size, which requires inference engines to sustain very high levels of concurrency. (ii) the computation required for each sequence can vary widely, which leads to substantial imbalance across GPUs and directly affects batch completion time. (iii) modern models introduce additional variation even within a single sequence because different neural network modules can have very different computational costs, further increasing the imbalance.

These limitations are not confined to a few specialized workloads. Instead, they appear consistently across batch inference settings. Below, we explain why high concurrency and load imbalance naturally arise from the way modern AI models and applications scale.

Insight 1: Intra-sequence imbalance is inherent to sparsity-driven model scaling. State-of-the-art AI models increasingly rely on sparse architectures, most notably Mixture-of-Experts (MoE), as the primary mechanism for scaling model capacity. Shown in Figure 2a, modern models such as DeepSeek-R1 [11], Kimi-K2 [21], GPT-5 [35], Gemini 3 Pro [38], and Grok [47] all deploy MoE models with hundreds of experts. The MoE layers contain the majority of



(a) MoE sequence compute. (b) Insufficient batch size for experts. (c) CDF of output lengths for DeepSeek-R1.

Figure 2: Challenges in MoE batch inference.

the model parameters and contribute a large portion of the compute required for each token.

Since each token activates only a small subset of experts, different tokens within the same sequence follow different execution paths. Some experts are selected far more often than others, and these variations accumulate as model capacity grows. As a result, intra-sequence imbalance is not a product of specific implementations but an intrinsic consequence of sparsity-driven scaling.

The intra-sequence imbalance causes today's state-of-the-art inference engines, even when they are configured in disaggregated mode (prefill-decoding separation), to suffer from underutilizing the GPUs by around 50% even though the input batch has sufficiently massive sequences (shown by Figure 2b). SGLang mixed means that prefill and decoding phases are deployed on the same servers, while SGLang PD uses prefill-decoding disaggregation, with the two phases deployed on separate servers.

Insight 2: Test-time scaling amplifies long-tail generation, causing inter-sequence imbalance. Batch inference workloads exhibit a persistent long-tail effect, where a small fraction of sequences dominate the overall computation. This phenomenon is especially critical in scenarios where batch completion time dominates the pipeline time, such as RL rollouts and deep-thinking pipelines.

Figure 2c illustrates this effect for DeepSeek-R1 on a production dataset trace. The imbalance in decoding lengths is severe: the P99 output length is 3.78× longer than P95, and the maximum length reaches 9.2× that of P95. These stragglers determine batch completion time, leaving most GPUs idle while waiting for a few long-running sequences.

The long tail issues create two key scalability challenges. (1) First, because decoding is sequential, a few extremely long sequences determine the batch completion time, acting as stragglers. (2) Second, GPUs processing shorter sequences finish early and remain idle, amplifying load imbalance between GPUs. In production workloads we consistently observe that this imbalance causes existing engines [22, 32, 56], and their disaggregated variants to lose roughly 10% to 70% of achievable GPU performance. This demonstrates that inter-sequence imbalance is not a rare occurrence but an inherent property of large batch inference workloads.

Takeaway. Together, these requirements and insights reveal the need for a new class of systems: **batch-native inference engines** that can sustain high utilization across large fleets of parallel GPUs while processing massive numbers of concurrent sequences, despite inter- and intra-sequence load variation that is only revealed at runtime.

3 BatchGen Design Intuitions

We begin by examining the design philosophy underlying existing inference engines.

Today's latency-driven scheduling model is a fundamental mismatch for batch inference. State-of-the-art systems such as SGLang [56] and vLLM [22] – regardless of whether they operate in continuous batching mode, chunked prefill, or a disaggregated configuration – are architected around a single objective: *complete each sequence as quickly as possible*. To meet this objective, they *bind* each sequence's computation and state (e.g., KV-cache) to a fixed device and execute it without interruption.

This binding philosophy appears at two levels. *Within* a forward pass, execution is atomic: once a batch enters the model, all sequences traverse every layer together before any scheduling decision can be made. There is no mechanism to pause between modules–e.g., to accumulate more sequences after attention before executing MoE. *Across* forward passes, sequences remain attached to the same device; moving a sequence's state elsewhere is treated as an exceptional, expensive operation to avoid. As a result, each sequence is permanently bound to its assigned GPU and cannot be moved or rescheduled. Once a sequence joins a batch, it stays with that batch until its own completion.

This design is well suited for interactive serving, where minimizing per-sequence latency dominates all other concerns. However, in batch inference, the rigidity becomes a core limitation-both in the *scheduling plans* the system can express and in the *granularity* at which it can intervene. The engine cannot pause a sequence mid-layer to form a more efficient expert batch, cannot adapt the plan as imbalance de-

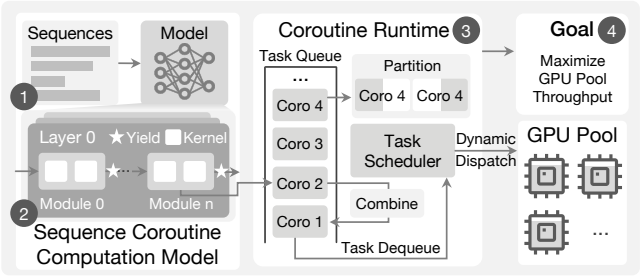


Figure 3: Event-driven Sequence Coroutine Architecture.

velops, and cannot redistribute work when some sequences finish far earlier than others.

Requirements for new batch-native inference scheduling model. In this paper, we revisit batch inference from first principles. In existing systems, a sequence is assigned to specific devices, and this assignment remains fixed because re-binding is viewed as adding unnecessary latency. Even in continuous batching [22, 50, 56], where micro-batches may be reorganized, the *sequence itself* remains implicitly tied to a particular device and cannot freely move or be decomposed.

This observation leads us to a novel system design opportunity: Can we model sequence computation as fine-grained coroutines and dynamically schedule them in a throughput-oriented, event-driven architecture?

We make four key insights, illustrated in Figure 3, that show why such a system design is practical and effective for today's batch inference workloads:

❶ **Sequences provide a natural granularity for coroutine-based scaling.** In batch inference, both compute and memory load scale with the number of sequences, and each sequence has an independent execution lifetime. This makes sequences a natural unit for coroutine computation.

❷ **Neural network modules offer natural yield points.** Sequence computation performs forward passes through modular components (e.g., attention, MoE layers) in prefill and decode. Module boundaries naturally serve as coroutine yield points for event-driven scheduling.

❸ **Sequence state can be dynamically combined, partitioned, and migrated.** The state associated with each sequence (e.g., hidden states in prefill, KV cache in decode) is tensor-structured and self-contained, allowing flexible pausing, combining, partitioning, and migration without affecting correctness.

❹ **Batch inference optimizes batch completion time, hiding the dynamic scheduling latency of sequences.** The extra overhead of coroutine scheduling may be undesirable for interactive workloads, but in batch settings, where end-to-end batch completion time is the primary metric, this overhead is amortized and outweighed by improved throughput from high-concurrency coroutine execution.

To the best of our knowledge, no existing system follows our proposed architectural model. Interactive inference engines such as SGLang [56] and vLLM [22] bind each sequence to a fixed GPU for its entire lifetime. Disaggregated inference systems, including DistServe [57], Splitwise [37], and Mooncake [39], relax this constraint by assigning each inference phase, such as prefill or decode, to a specific GPU, but the phases still remain statically placed. MegaScale-Infer [60] further disaggregates decode into attention and expert modules, but continues to rely on fixed placement of these components. Kernel oriented systems such as NanoFlow [59] improve execution efficiency, but do not address their underlying scheduling and placement limitations.

	Interactive Inference	Disaggregated Inference	Sequence Coroutine (Ours)
Scheduling plan	Static	Static	**Adaptive**
Scheduling granularity	Coarse-grained, fixed	Coarse-grained, fixed	**Fine-grained, flexible**
Operational cost	Medium	High	**Low**
Throughput	Medium	Medium	**High**

Table 1: Scheduling characteristics of three inference system architectures.

Architectural advantages. We name our new system architecture the *event-driven sequence coroutine architecture*. Table 1 summarizes its potential effectiveness. By exploiting sequence-level coroutines and providing system mechanisms for their yield and combination, we can significantly increase batch sizes on sparse operations (*e.g.* each expert in MoE) at runtime and thereby achieve significantly higher GPU utilization than interactive engines (shown in Figure 2b).

Meanwhile, when long-tail generation or load imbalance is detected, low-cost event-driven scheduling can yield, migrate, partition, and recombine sequence coroutines to avoid stragglers and maintain balanced loads.

Because GPUs are managed as a shared resource pool rather than being pre-allocated to specific sequences, sequence coroutines can be dynamically dispatched to any available GPU, making the system easy to deploy across batch jobs with widely varying input–output length distributions.

Finally, sequence coroutines execute under an event-driven model in which GPUs always have useful events to process, while the scheduler keeps most coroutine state in host memory. This substantially reduces peak GPU memory usage and lowers the operational cost of batch inference.

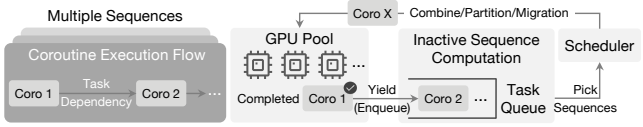
A powerful analogy is the transition from thread-per-connection servers (e.g., Apache HTTP Server [3]) to event-driven architectures (e.g., NGINX [31], SEDA [46]). Apache performs well under modest load but degrades under highly variable, large-scale traffic; NGINX treats requests as coroutines and uses event-driven scheduling to sustain scalability. Batch inference demands the same paradigm shift.

```python
class SequenceCompute:
  seq_id: int; max_out: int; max_in: int; tokens: str
  phase: Literal["prefill", "decoding"]
  status: Literal["init", "active", "inactive", "done"]
  state: object; output: object

  async def yield(...): # Scheduler takes control
  async def combine(...): # Multiple states
  async def partition(...): # Model parallelism
  async def migrate(...): # Physical location
  async def run(...): # Call module forward

  def callback(...): # Custom local state handling
```

(a) Sequence computation states and functions.

```python
def module_wrapper(module: Module):
  async def coroutine(
    s_instance: SequenceCompute,
    combine_with: List[SequenceCompute] = None,
    n_part: int = 1, part_mode: Literal["TP"]):
    # Scheduler implementation
    return s_instance
  return coroutine
# applies to attention, MoE, prefill, decoding
attention.forward = module_wrapper(attention)
```

(b) Module wrapper definition and coroutine creation.



(c) Concurrent sequence coroutine execution.

Figure 4: Sequence coroutine abstraction.

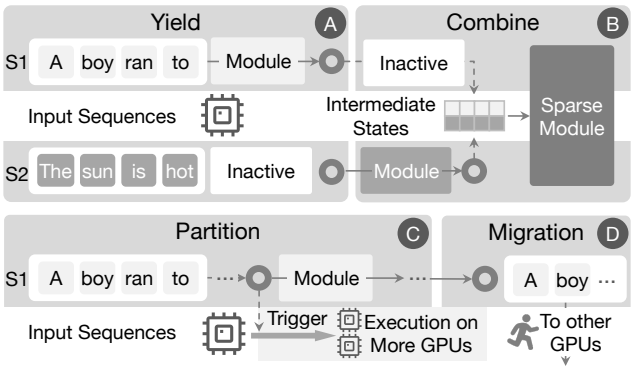


Figure 5: Coroutine primitives used on sequence.

4 The Sequence Coroutine Model

4.1 The Sequence Coroutine Abstraction

We abstract sequence coroutines as follows: a representation of a neural network model's per-sequence execution that can be paused, migrated, combined, partitioned, and resumed without losing correctness.

Establishing this abstraction requires answering three key questions: (i) How to model the sequence's computation process and intermediate state? (ii) How to enable sequence coroutine execution in a neural network that is typically built using modular abstractions? (iii) How to execute concurrent sequence coroutines on a limited number of compute units?

Modeling sequence computation and state. The sequence computation (defined in Figure 2a) requires states that fully determine the current and future computations. We define the data structure as shown in Figure 4a, using language models as an example for the values in each field. Each sequence can be uniquely identified, with the amount of computation to be scheduled (*i.e.* `max_out`, `max_in`). The coroutine state stores the KV-cache, while the output consists of the hidden states passed between module forward calls, enabling the coroutine to resume execution correctly at the next scheduling point. Sequence computation exposes primitives (*i.e.* `yield`, `combine`, `partition`, `migrate`) to enable coroutine scheduling on sequence computation, and callbacks for customized operations on sequence states.

Creating coroutines through module wrappers. We provide an abstraction layer that preserves the modular semantics of existing model definitions (*e.g.* `torch.nn.Module`), while inserting scheduling points at runtime (Figure 4b). During model initialization, BatchGen applies wrappers on selected modules, allowing coroutine steps to be generated automatically. The wrapper allows any custom scheduling policy to be implemented using coroutine primitives. By default, the coroutine yields the sequence compute on exit.

Concurrent sequence coroutine execution. Concurrency is managed by a coroutine scheduler for coordination over the GPU pool (Figure 4c). Once modules are wrapped, a sequence computation naturally forms a *coroutine execution flow* that defines the successor relationships between computation stages. When a coroutine yields, it enqueues its successor, while the scheduler selects one or more inactive sequence coroutines from the global queue to dispatch onto available compute units. A many-to-one mapping corresponds to `combine`, a one-to-many mapping corresponds to `partition`, and a one-to-one mapping corresponds to a simple resume.

4.2 Sequence Coroutine Mechanisms

When designing mechanisms for sequence coroutines, we have three requirements: (i) they must provide yield primitive that preserves correctness for stateful neural network execution; (ii) they must enable coroutine combination to inflate expert-level batch sizes in MoE computation; and (iii) they must support coroutine partitioning to mitigate stragglers in long-tail prefill and decoding stages.

Sequence coroutine yield mechanism. A `YIELD` in sequence compute behaves analogously to the await primitive in pro-

gramming languages such as Python: yield suspends a sequence coroutine, checkpointing its state produced and releasing GPUs, such that control can safely return to the scheduler. Figure 5 Ⓐ illustrates this process: sequence S_1 yields after a module computation and becomes inactive, allowing the scheduler to resume inactive S_2 on the same GPU.

Sequence coroutine combination mechanism. Standard coroutine semantics require an explicit `resume` primitive to continue suspended execution. In our model, resume semantics are part of `COMBINE`: since the purpose of combining sequences is to form a batch for computation, resumption occurs implicitly. This design eliminates a separate primitive while preserving expressiveness.

Analogous to data-stream operators in Spark [51], `COMBINE` merges multiple yielded sequences into a single batch for efficient GPU execution. Figure 5 Ⓑ illustrates this process: intermediate states (in the form of tensors) are concatenated from S_1 and S_2, doubling the batch size for the sparse module and increasing compute density on a single GPU.

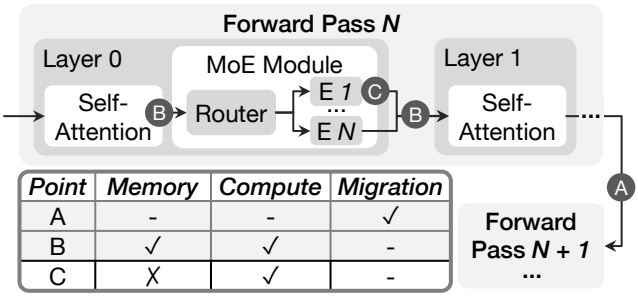
Sequence coroutine partitioning mechanism. `PARTITION` distributes a single long-running sequence across multiple GPUs using tensor parallelism to accelerate completion. Each GPU executes the same program over its assigned portion of the data. Figure 5 Ⓒ illustrates this on a single sequence: the scheduler finds another idle GPU to form a parallelism group for the module. The GPU group can be local to a node or distributed, depending on the scheduler's decision.

Sequence coroutine migration mechanism. `MIGRATE` transfers a sequence's state (*e.g.* KV-cache and metadata) to a different physical device for load balancing. The call is asynchronous while still ensuring consistency. The physical location of the sequence is left to the scheduler for book-keeping. Figure 5 Ⓓ illustrates this for moving to another GPU. This also applies to sequences in Host and other media as well.

4.3 How to Use Sequence Coroutine?

In this subsection, we use an MoE model to illustrate how to use the sequence coroutine abstraction for batch inference.

Two types of yield points. *Intra-forward* yield points occur within a single forward pass. When a downstream module reaches good device utilization only at a different batch size than the module preceding it, sequences yield after the upstream module so that the runtime can accumulate and `COMBINE` them into a batch downstream. For instance, attention saturates the GPU at a modest batch size while the sparsely activated MoE requires a much larger one. This applies to any sequence of modules with heterogeneous batch-size demand. Because the decisive factor is the model's compute characteristics, intra-forward yield points are selected once, statically, per model (§5.4). *Inter-forward* yield points occur across forward passes and are driven by runtime conditions rather than model structure. A sequence yields either



Point	Memory	Compute	Migration
A	-	-	✓
B	✓	✓	-
C	✗	✓	-

Figure 6: Yield point selection.

to achieve better utilization for batch progression (*e.g.* switch from decode to prefill to refill sequence) or under resource pressure (*e.g.* evict sequence under growing decode length that exceeds GPU memory). The yielded sequence can be combined again later with other active sequences running on GPU. Because these conditions depend on runtime state, inter-forward yield points are decided dynamically (§5.3).

Key development question: the choice of yield points. A central question arises when adopting sequence coroutines: given a model with multiple modules-and thus many candidate yield points-where should developers place yield points to maximize throughput?

For MoE architectures, we observe three practical and reusable yield-point options (shown in Figure 6): Ⓐ treating the attention + MoE layers as a single coroutine unit, Ⓑ splitting the attention and MoE layers into two coroutine units, and Ⓒ treating each expert in an MoE layer as an independent coroutine unit.

These options reflect throughput-memory trade-offs: Ⓐ a combined attention-MoE coroutine minimizes intermediate states for sequence computation but limits expert-level batching and sequence concurrency; Ⓑ a split design allows the system to dynamically increase expert-level batch sizes without overwhelming memory usage; and Ⓒ a per-expert coroutine maximizes sequence computation concurrency but has prohibitive memory costs when initiating millions of coroutines.

According to these insights, we design the coroutine schedule to be hierarchical. For MoE batching, we adopt option Ⓑ, inflating the batch size at MoE layers to achieve substantial throughput gains while keeping memory usage controlled at attention layers. To mitigate long-tailed decoding, we additionally employ option Ⓐ. This enables lightweight migration and allows parallelism configurations to be adjusted dynamically through state checkpointing already made during yield.

The coroutine callbacks allow users to inject custom logic to handle sequence local state without modifying the runtime. These callbacks can be used to implement techniques such as dynamic quantization of KV-cache on a per-sequence basis [13], which can respond to runtime memory pressure or accuracy requirements.

Algorithm 1: Scheduling model forward pass

Input: B_{attn}: attention batch size (`COMBINE` on attention);
B_{moe}: MoE batch size (`COMBINE` on MoE);
Sequence group $\mathcal{S}$, $|\mathcal{S}| = B_{\text{moe}}$;
Output: Final hidden states for all sequences in $\mathcal{S}$.

1 **Function** FORWARDPASS($\mathcal{S}, B_{attn}, B_{moe}$)**:**
2 **foreach** *layer* $\ell = 1$ **to** L **do**
3 Partition $\mathcal{S}$ into attention sub-batches of size B_{attn};
4 **foreach** *sub-batch* $g \subseteq \mathcal{S}$ **do**
5 $h_g \leftarrow \text{ATTENTION}(g, \ell)$ `// Kept in GPU`
6 ASYNCCKPT(g, ℓ);
7 **YIELD** g, $g.h \leftarrow h_g$; Attn Coroutine
8 $h_{\text{combined}} \leftarrow$ **COMBINE** $\forall g.h, g \in \mathcal{S}$;
9 $h_{\text{out}} \leftarrow \text{MOE}(h_{\text{combined}}, \ell)$ `// Kept in GPU`
10 **YIELD** $\mathcal{S}$, $\mathcal{S}.h \leftarrow h_{\text{out}}$; MoE Coroutine
11 **YIELD** $\mathcal{S}$;
12

Algorithm 2: Scheduling Sequence Computation

Input: B_{attn}: attention batch size;
B_{moe}: MoE batch size;
Global sequence batch $\mathcal{S}_{\text{global}}$, Node sequence batch $\mathcal{S}_{\text{node}}$

1 Distribute $\mathcal{S}_{\text{global}}$ evenly;
2 **while** $\exists s \in \mathcal{S}_{node} : \neg\text{FINISHED}(s)$ **do**
 `/* Run on each node */`
3 $\mathcal{S} \leftarrow \text{SELECTBATCH}(\mathcal{S}_{\text{node}}, B_{\text{moe}}, \text{status=inactive})$;
4 **COMBINE** $\mathcal{S}$ as active sequences;
5 FORWARDPASS($\mathcal{S}, B_{\text{attn}}, B_{\text{moe}}$);
6 DYNAMICSEQUENCEMANAGEMENT($\mathcal{S}$);
7 **if** ONREFILLNODE*(nodes)* **then**
8 wait until all $s \in \mathcal{S}_{\text{node}}$ **YIELD**;
 `/* Trigger prefill temporally */`
9 $\mathcal{S} \leftarrow \text{SELECTBATCH}(\mathcal{S}_{\text{node}}, B_{\text{moe}}, \text{status=init})$;
10 **COMBINE** $\mathcal{S}$ as active sequences;
11 FORWARDPASS($\mathcal{S}, B_{\text{attn}}, B_{\text{moe}}$);
 `/* decoding continues here */`
12 **if** ONLONGTAIL*(nodes, gpus)* **then**
13 wait until all $s \in \mathcal{S}_{\text{node}}$ **YIELD** on target GPUs;
14 **PARTITION** remaining sequences to target GPUs;

5 System Design and Implementation

5.1 Sequence Coroutine Scheduling Algorithm

We design a sequence coroutine scheduler to sustain high compute efficiency across prefill and decoding. The core challenges stem from sparsely activated MoE layers, which require large batches, and from unpredictable long-tailed decoding. Our scheduler uses coroutine primitives to accumulate batches, adapt memory usage, and reduce long-tail inefficiency through continuous batching and dynamic parallelism.

Providing sufficient batch size for sparse module. Our scheduling algorithm aims to provide a sufficient batch size for MoE model by `COMBINE`-ing yielded sequences from attention. The design is motivated by a key insight: *a module-level roofline model enables static planning for batch accumulation*. Given the system's capacity for both active (GPU-resident) and inactive (host-resident) sequences, we can select the largest feasible module batch size that remains within memory constraints.

We design the coroutine scheduling in alignment with Figure 4b for attention, MoE, and model forward pass as in Algorithm 1, where the shaded area represents the code inside module wrapper that creates a coroutine. For attention, the batch is split into sub-batches of size B_{attn}; each sub-batch executes attention, buffers its hidden states, optionally offloads KV-cache, and then yields. For MoE, all sub-batch hidden states are combined into a large batch of size B_{moe}.

Mitigating long-tailed and unbalanced decoding. Long-tailed decoding can only be managed at runtime, as output lengths are unknown in advance. We address this challenge through two mechanisms: *Dynamic sequence management* maintains high GPU utilization by continuously refilling the batch with prefilled sequences. When sequences complete or are evicted, the scheduler invokes `COMBINE` to merge waiting sequences into the active batch, preventing GPUs from idling. We detail this in §5.3. *Straggler acceleration and load balancing* handle the final phase of batch completion. When only a few long-running sequences remain, `PARTITION` redistributes their computation across idle GPUs—using tensor parallelism for single stragglers or data parallelism for multiple. Throughout execution, `MIGRATE` rebalances sequences across nodes to prevent skew from uneven completion rates.

We illustrate the scheduling loop in Algorithm 2. Nodes begin with prefilled sequences stored in host memory (line 1). The main loop (lines 2–6) repeatedly selects batches, executes forward passes, and invokes dynamic sequence management. The ONREFILLNODE callback (lines 7–11) triggers prefill when GPUs become underutilized. The ONLONGTAIL callback (lines 12–14) detects stragglers and applies `PARTITION`, which requires yielding active sequences to reconfigure GPU memory for the new parallelism strategy with its impact demonstrated in §6.3.

We design two callbacks, ONREFILLNODE and ONLONGTAIL, to manage long-tail behavior. ONREFILLNODE supplies additional inactive sequences to the decoding phase so that idle or underutilized GPUs remain fully occupied (lines 7-11). When long-tail sequences persist and no inactive sequences remain to refill the batch, the scheduler invokes ONLONGTAIL (lines 12-14). This callback waits for all active sequences on the target GPUs to yield, ensuring their states are safely checkpointed before reconfiguring model parallelism and redistributing the remaining work.

5.2 System Memory Model

We design the system memory layout to support coroutine scheduling with minimal overhead. The key challenges arise from the different memory demands of prefill and decoding, the sensitivity of batch combination size to peak memory usage, and the need to shift efficiently between phases while preserving accumulated state.

Our memory layout addresses these issues by unifying host and device memory management, paged KV-cache management, and providing phase-agnostic GPU buffers that can be reconfigured without disrupting coroutine execution. The residence of the KV cache and model weights, i.e., whether each is offloaded to host memory, is a throughput-driven decision resolved by the coroutine scheduling plan (§5.4), depending on whether the host-device transfer cost is outweighed by the throughput gain from larger-batch execution.

Figure 7 illustrates BatchGen's memory layout and execution flow for both prefill and decode phases, rooted in the following principles.

Host memory should serve as a unified store. On each node, the host memory holds shared model parameters and KV-cache for all sequences scheduled to the node (in ❶). This provides two benefits: (i) *a single source of truth* for all model states, avoiding redundant per-GPU copies, and (ii) *local coroutine store* for checkpointing KV-cache on `YIELD` and restoring it on `COMBINE` without distributed synchronization. Parameter movement is unidirectional: host memory serves as the source for GPU prefetching, not a destination for offloading.

Supporting local phase switching with fine-grained GPU buffers. BatchGen partitions GPU parameter memory into two regions (in ❷ and ❸). (i) *Resident parameters* store parameters that remain in the GPU throughout the current phase, which is typically small (*e.g.*, LayerNorm). During decoding, it may include all attention parameters when memory permits. (ii) *Parameter and KV buffer* is a transient staging area for prefetching and offloading. The buffer slot ownership is released after the module completes execution. As prefill and decoding share the same design, we reconfigure the size of these components during the phase swap.

Increasing peak memory budget for prefill. Prefill-phase attention exhibits high peak memory demand due to FlashAttention's $O(N)$ memory complexity, where N is the sequence length. In contrast, decode-phase attention requires only $O(1)$ memory per sequence, as each iteration processes a single token against the existing KV-cache. We design the prefill phase to offload more aggressively by asynchronously offloading the microbatch KV-cache from each layer's attention to host and reclaiming its GPU buffer immediately. The prefill phase has enough computation to hide the offloading cost. At any instant the GPU therefore holds at most two layers' worth of KV-cache, making prefill memory less demanding from both batch size and sequence length. This results in minimizing

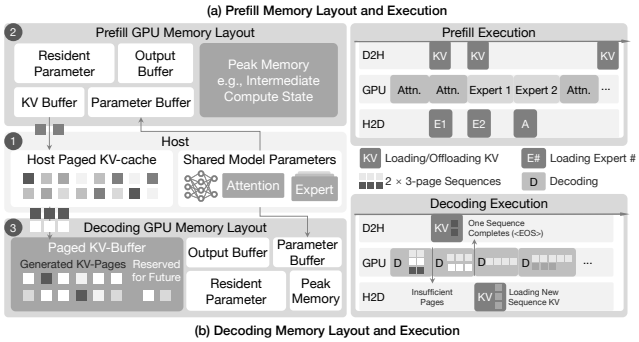


Figure 7: Memory layout and execution flow for prefill (top) and decode (bottom) phases. Host memory serves as checkpoint storage for parameters and KV-cache, while GPU memory is partitioned into persistent and transient regions.

buffer size in prefill (in ❷). The example for the ring parameter buffer and single KV buffer is shown in Figure 7a, with the running and prefetching expert using the ring buffer and KV-cache offloading (asynchronously after attention) using the single buffer.

Increasing sequence concurrency for decoding. Each GPU maintains a paged KV-cache manager (in ❸) that allocates and tracks KV pages at a fine granularity. During decoding, BatchGen employs a two-page buffer strategy: each sequence reserves only two KV pages for subsequent decoding iterations, with additional pages allocated on demand as the sequence grows. This lazy allocation maximizes the number of active sequences. Offloading during decoding considers a trade-off between parameters and KV-cache transfer cost and the throughput gain from larger batch size. Decode has low arithmetic intensity thus offers little computation to hide host-to-device transfers. We demonstrate a case for benefits under tight memory budgets (§6.5). When GPU memory becomes insufficient, we evict sequences until two pages can be allocated for all active sequences. Detailed design will be covered in §5.3.

5.3 Dynamic Sequence Management

The dynamic sequence management design is unique to batch inference: to maximize throughput, we must maintain large batch sizes or high parallelism throughout both prefill and decoding phases.

BatchGen checks memory state and adjusts the active batch every P tokens (i.e., after decoding one KV page of tokens). At each page boundary, the scheduler executes four phases: (i) *Sync*: wait for pending asynchronous KV append operations to complete—these operations continuously propagate newly generated KV entries from GPU to host memory, maintaining host as the single source of truth; (ii) *Eviction*: `YIELD` completed sequences, releasing their GPU and host KV pages;

(iii) *Extension*: for sequences approaching their allocated page limit, either extend allocation if GPU pages are available, or `YIELD` them to host (marking as suspended) to free pages for other sequences; (iv) *Refill*: `COMBINE` new sequences from the prefilled or suspended pool, launching async KV restoration that overlaps with the next page's forward passes.

These operations provide the following properties:

Ensuring decoding batch size is sufficient. BatchGen uses *adaptive sequence selection*. When GPU memory is insufficient for all active sequences, BatchGen must decide which sequences to `YIELD`. The scheduler prioritizes yielding sequences with the most progress (highest decoded length), as they have more KV-cache already checkpointed to host and are closer to completion. This strategy maximizes the number of sequences that can make forward progress.

Ensuring load-balance on refill. Long-tail generation creates imbalance across GPUs as sequences complete at different times. When a GPU finishes sequences, BatchGen uses `MIGRATE` to redistribute suspended sequences from other nodes, keeping all GPUs utilized. Refill candidates are selected in FIFO order to preserve fairness. When all nodes lack sufficient suspended sequences, BatchGen temporarily switches to prefill mode to generate new sequences. Note that `MIGRATE` requires the coroutine task queue to be blocked to (i) avoid races between active execution and memory transfer, and (ii) synchronize on the global count of inactive sequences. This blocking is low-cost since migration overlaps with sequence computation on other GPUs.

Ensuring single sequence straggler is parallelized. When straggler sequences remain, batching benefits diminish and most GPUs become idle. BatchGen allows configuring a threshold on global batch size to trigger `PARTITION`: for multiple stragglers, sequences are distributed across GPUs using data parallelism; for a single straggler, computation is partitioned across GPUs using tensor parallelism. The KV-cache distribution depends on the attention architecture: for MLA, the compressed latent KV-cache is replicated across GPUs; for MHA/GQA, the KV-cache is split across attention heads.

5.4 Optimization of Scheduling Plans

The runtime also follows a static plan that fixes batch sizes, buffer allocations, and yield points. Manual tuning is infeasible given the combinatorial space of model scales and hardware configurations, so BatchGen derives this plan automatically through lightweight profiling and simulation.

Module-level performance modeling. Modern LLMs are structurally homogeneous, allowing BatchGen to profile a single representative layer rather than the entire model. It measures attention, MoE, and collective communication kernels over multiple batch sizes and fits the results to roofline-style models that predict runtime as a function of batch size and memory allocation.

Category	Operation	Cost
Intra-forward `YIELD`	Hidden state checkpoint	$<5\,\mu$s
	KV-cache offload	overlapped
Inter-forward `YIELD`	CUDA event sync	$<1\,\mu$s
	Metadata update	$<10\,\mu$s
`COMBINE`	Without offloading	0 (GPU resident)
	With offloading	$\sim$0.2 ms per seq. per layer
`PARTITION`	Parallelism reconfig	$\sim$5–10 s (when triggered)
`MIGRATE`	KV-cache transfer	Opportunistic, overlapped
Scheduling	Intra-forward	0 (static plan)
	Cross-node sync	5–10 ms / 64 tokens

Table 2: Coroutine primitive overhead for DeepSeek-R1 on 8×H20 with 10K context length and batch size 512.

Execution DAG construction. BatchGen models a single layer's forward pass as a directed acyclic graph (DAG) to simulate execution under different configurations. Each node in the DAG represents either a computation (attention, MoE, `YIELD` checkpointing) or a data transfer (parameter prefetch, KV-cache offload/restore for `COMBINE`). Edges encode dependencies: a module cannot execute until its parameters are loaded; `COMBINE` cannot proceed until KV-cache restoration completes (if needed). Node costs are assigned from the profiled execution times and memory requirements at each candidate batch size. The critical path through this DAG—the longest dependency chain—determines the layer's execution time, capturing how computation and data movement overlap.

Configuration search. Given a DAG, finding the critical path requires a single topological traversal using dynamic programming, with complexity $O(V+E)$ where V and E are the number of nodes and edges. Since a layer's DAG contains fewer than 100 nodes, this computation is negligible. BatchGen enumerates candidate configurations (B_{attn}, B_{moe}, and buffer sizes), constructs the corresponding DAG for each, and selects the configuration with the shortest critical path.

5.5 Coroutine Overhead Analysis

A natural question is whether coroutine-based scheduling introduces noticeable overhead. Table 2 summarizes the cost of all coroutine primitives.

Frequently invoked primitives have negligible overhead. Coroutine boundaries occur once per module. Hidden-state checkpointing writes only $\sim$7 MB for 512 sequences (7168-dim), completing in under 5 μs at HBM bandwidth, compared to $\sim$3 ms for the module's computation. KV-cache offloading is issued asynchronously and fully overlaps with MoE execution. Between forward passes, `YIELD` performs a CUDA event sync ($<1\,\mu$s) and updates sequence metadata on the CPU ($<10\,\mu$s for 512 sequences).

`COMBINE` overhead depends on whether KV-cache offloading is active. Without offloading, combined sequences are

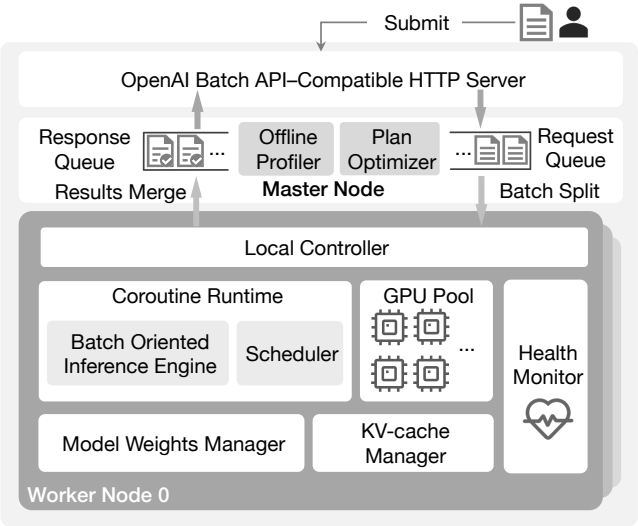


Figure 8: BatchGen system architecture with master-worker organization, API endpoints, and inference engine.

already GPU-resident, incurring zero cost. With offloading, `COMBINE` must restore KV-cache from host memory; for a 10K-token sequence this takes ∼0.2 ms per layer over PCIe 5.0. This cost arises only when refilling batches and is amortized by the larger batches that offloading enables.

Primitives with millisecond-level overhead are negligible to overall computation time. `MIGRATE` is invoked opportunistically to balance the number of inactive sequences across nodes. The number of migrations over a batch stays small relative to the total sequence count and the migration cost can be smaller than prefill. When `MIGRATE` is scheduled in advance, the KV-cache transfer is fully overlapped with ongoing computation on other sequences, adding zero latency to the critical path. For instance, on a standard H100 HGX node with 400 Gb/s (∼50 GB/s) InfiniBand link, prefilling a 2K-token DeepSeek-R1 sequence costs ∼112 ms even at the GPU's peak throughput (BF16 MLA attention at 989 TFLOP/s plus FP8 MoE GEMMs at 1979 TFLOP/s), whereas migrating the sequence's entire 144 MB MLA KV-cache over that link takes only ∼2.9 ms, which is less than 3% of the prefill compute it overlaps with. Although `PARTITION` takes seconds, it is triggered only when long-tail sequences are detected (typically once per batch near completion) and enables idle GPUs to accelerate remaining stragglers. Meanwhile, batch completion time is more than 10 minutes. Dynamic scheduling occurs only at page boundaries (every 64 tokens), requiring 5–10 ms for cross-node synchronization—approximately 0.1–0.2% of the compute time for 64 decode iterations.

5.6 Implementation Details

We implement BatchGen in 13K LoC of C++ and 49K LoC of Python. The C++ layer implements the host-side KV-cache system and the asynchronous host–device copy engine, while the Python layer implements the coroutine scheduler and the PyTorch-based forward-pass skeleton for each model. Beyond third-party kernels (e.g., FlashAttention [8], DeepGEMM [10]), BatchGen develops and maintains its own library of kernels for batch inference, adapted to BatchGen's KV-cache and input/output buffer layout.

Figure 8 shows the BatchGen system architecture for deployment at scale with the control plane implemented on an industrially customized Ray [30]. Key features include:

OpenAI-compatible batch API. BatchGen exposes an HTTP endpoint compatible with the OpenAI Batch Inference API [34]. A master node per model-parallel group accepts batch submissions, partitions sequences across workers via the coroutine scheduler, and returns results preserving input order. This enables BatchGen to serve as a drop-in replacement for existing batch inference pipelines.

Batch-oriented inference engine. The inference engine integrates high-performance compute and communication kernels (e.g., DeepEP [9]) and includes additional kernel fusions tailored for model types. It supports phase-specific parallelism strategies: (i) *DP prefill with offloading*, where prefill is compute-bound and parameter/KV transfers are fully hidden, enabling scalable data parallelism; and (ii) *DP+EP decoding*, which combines data-parallel attention with expert-parallel MoE to maximize bandwidth utilization during generation.

Cold-start optimization. Batch inference often runs on spot instances where cold-start latency impacts usable compute time. BatchGen includes a *model weights manager* with: (i) *Huge pages*: Allocating GPU-accessible host memory incurs page faults proportional to pool size. BatchGen uses 2 MB huge pages, reducing initialization time from minutes to seconds for TB-scale pools. (ii) *Fast checkpoint loading*: BatchGen adopts the memory-mapped checkpoint format from ServerlessLLM [15], enabling rapid weight loading that fits within typical spot instance lifetimes.

Failure detection and recovery. At scale, GPU and node failures are inevitable. BatchGen implements *health monitoring* where per-node responses carry all GPU statuses. For recovery, `MIGRATE` can either transfer KV-cache state or trigger recomputation. Since migrating hundreds of gigabytes may be slower than regenerating, BatchGen uses its performance model to estimate both costs and selects the faster path.

6 Evaluation

Testbed. We evaluate BatchGen on a production-grade multi-GPU cluster representative of real LLM deployment environments. The testbed includes NVIDIA H20 and H200 servers scaled from 8 to 128 GPUs. Each node offers high-bandwidth intra-node connectivity via NVLink and PCIe 5.0, with 2 TB of host memory, and nodes are interconnected through a

Model	Framework	H20 (8 GPUs)		H20 (16 GPUs)		H200 (8 GPUs)	
		8K-2K	2K-8K	8K-2K	2K-8K	8K-2K	2K-8K
DeepSeek-R1 (671B)	vLLM	1020.6	3233.8	373.8	746.3	1285.2	1076.7
	SGLang	816.7	3116.7	375.0	981.9	258.9	880.8
	SGLang-Opt	816.7	3116.7	137.0	242.5	95.2	208.3
	BatchGen	**625.1**	**1684.1**	**82.6**	**194.3**	**75.1**	**165.0**
	Speedup	*1.31×*	*1.85×*	*1.66×*	*1.25×*	*1.27×*	*1.26×*
Kimi-K2 (1T)	vLLM	OOM	OOM	1862.0	2159.2	3474.5	3480.5
	SGLang	OOM	OOM	447.9	1468.1	293.8	975.6
	SGLang-Opt	OOM	OOM	150.6	430.6	167.5	525.0
	BatchGen	**659.8**	**1693.6**	**116.8**	**317.5**	**125.0**	**392.0**
	Speedup	–	–	*1.29×*	*1.36×*	*1.34×*	*1.34×*

Table 3: Batch completion time (minutes) for 6K sequences on LongBench. OOM indicates out-of-memory. Speedup is relative to SGLang-Optimized. For Kimi-K2 on 8×H20, BatchGen is the only system that completes execution.

200 Gb/s RDMA-enabled InfiniBand fabric, matching common datacenter configurations for large-scale model serving.

Baselines. Our primary baselines are SGLang (v0.5.5.post3) [56] and vLLM (v0.11.2) [22], which are widely recognized as SOTA open-source systems. We evaluate SGLang-Optimized, which applies exhaustive tuning for maximum throughput: we configure 16 DP-attention ranks (a throughput-friendly parallelism strategy), tune the memory allocation fraction to the threshold before OOM, and selectively restrict CUDA graph capture to necessary batch sizes to reduce GPU memory consumption and allow larger runtime batches.

For memory-constrained scenarios (§ 6.5), we compare against systems with offloading optimizations: DeepSpeed [12], FlexGen† [41], and MoE-Lightning† [7]. For disaggregated inference (§6.4), we evaluate prefill-decode (PD) disaggregation using SGLang's implementation. We exclude attention-expert disaggregation [60] due to the lack of open-source implementations.

Models. Our evaluation spans four SOTA MoE models, including Mixtral-8×7B (47B parameters) and Mixtral-8×22B (141B parameters) [18], DeepSeek-R1 (671B parameters) [11], and Kimi-K2 (1T parameters) [21].

6.1 Application 1: Offline Inference

We evaluate BatchGen using LongBench [4]. We evaluate on two common production patterns: (i) *prefill-heavy* (8K input, 2K output tokens): reflects summarization and information extraction tasks, and (ii) *decoding-heavy* (2K input, 8K output tokens): captures reasoning-intensive workloads such as chain-of-thought question answering. Table 3 reports batch completion time for 6K sequences.

BatchGen achieves larger speedup on weaker GPUs. On 16×H20 and 8×H200 clusters, BatchGen delivers 1.25–1.66× speedup over SGLang-Optimized across all configurations. The improvement is consistently larger on H20 than on H200. H200's higher compute throughput shrinks the relative cost of prefill, where BatchGen's batch accumulation provides the most benefits. On H20, prefill dominates end-to-end time, and BatchGen's ability to form large expert batches yields proportionally greater gains.

BatchGen enables larger models and far larger batches under tight memory budgets. On 8×H20 GPUs (768 GB total HBM), baseline behavior diverges by model. DeepSeek-R1's parameters (approximately 642 GB in FP8) fit in HBM, but baselines must partition remaining capacity between KV-cache and working memory, limiting batch sizes to 8–16 sequences. SGLang-Optimized cannot improve over default SGLang in this regime—exhaustive tuning yields no additional headroom when memory is saturated. BatchGen achieves 1.31–1.85× speedup by offloading KV-cache to host memory, enabling batch sizes of 1800+ sequences. For Kimi-K2 (1T parameters), no baseline completes execution: the model exceeds HBM capacity entirely. BatchGen is the only system that runs Kimi-K2 on this hardware configuration.

The offloading regime exhibits a counterintuitive effect: BatchGen delivers nearly identical throughput on DeepSeek-R1 and Kimi-K2, even though Kimi-K2 is about 1.5× larger. With offloading, performance becomes PCIe-bandwidth-bound rather than compute-bound. Because both models have similar per-token KV-cache sizes and PCIe bandwidth is fixed, their achievable batch sizes-and thus throughput-converge. This trade-off remains advantageous: offloading enables much larger expert batches, keeping MoE layers in the memory-bound regime where compute scales sublinearly with batch size, while greater sequence-level parallelism amortizes transfer costs.

Comparison with TensorRT-LLM. We additionally compare against TensorRT-LLM [32], which represents state-of-the-art kernel optimization but requires non-trivial deployment effort. On 16×H20 GPUs with the 8K-2K workload, BatchGen outperforms TensorRT-LLM by 10%, demonstrating that coroutine-based scheduling provides benefits orthogonal to kernel-level optimizations.

6.2 Application 2: Test-Time Scaling

Unlike offline batch inference, test-time scaling places SLO constraints on batch completion time to maintain acceptable user experience, requiring systems to choose batch sizes that finish before the deadline. We use Recursive Self-Aggregation (RSA) [43] as a representative workload. After the first round, each subsequent round combines K prior answers as the prefill input, producing a prefill-to-decode ratio of K:1. We evaluate this setting on DeepSeek-R1 using 16×H20 GPUs with a fixed 8K decoding length per round.

BatchGen enables efficient SLO–throughput tradeoffs. Table 4 reports the number of sequences served under 30-minute and 60-minute SLO targets. BatchGen processes 1.25–1.57× more sequences under the 30-minute SLO and 1.66–

Model	Framework	T=4, N=8, K=4		T=2, N=16, K=4		T=3, N=8, K=2	
		30min	60min	30min	60min	30min	60min
DeepSeek-R1	vLLM	10.9	23.1	10.7	20.1	7.3	15.0
	SGLang-Opt	12.8	25.9	13.4	24.7	34.0	69.2
	BatchGen	**16**	**43**	**21**	**42**	**53**	**121**
	Speedup	*1.25×*	*1.66×*	*1.57×*	*1.70×*	*1.56×*	*1.75×*

Table 4: Number of sequences served under SLO constraints for RSA test-time scaling on DeepSeek-R1 with 16×H20 GPUs. RSA uses T rounds, N candidates per round, selects K candidates for the next round, and fixes decoding length at 8K. Higher is better. Speedup is relative to SGLang-Optimized.

1.75× more under the 60-minute SLO compared to SGLang-Optimized. The larger gains under relaxed SLOs reflect BatchGen's ability to form larger batches when deadlines permit, improving expert-level compute density. Prefill-heavy configurations (higher K) show greater improvements because compute-bound prefill enables effective overlap of parameter and KV-cache transfers at no additional cost.

BatchGen maintains advantages even under constrained batch sizes. Under tighter SLOs, BatchGen cannot always operate at its theoretically optimal batch configuration. For instance, in the (T=4, N=8, K=4) 30-minute scenario, meeting the completion deadline requires reducing the MoE batch size below the compute-saturating threshold. This sub-optimal configuration explains why BatchGen's speedup at 30 minutes (1.25×) is lower than at 60 minutes (1.66×)—the SLO constraint limits batching flexibility. Nevertheless, BatchGen still outperforms all baselines, demonstrating that its coroutine-based scheduling provides benefits even when batch sizes are externally constrained. This makes BatchGen suitable for online services with relaxed (minute-to-hour scale) SLOs where some latency–throughput tradeoff is acceptable.

6.3 Application 3: RL Training Acceleration

Reinforcement learning from human feedback (RLHF) [36] is a standard stage in LLM training, and its rollout phase often dominates total training time. Unlike offline inference, RL rollout fixes the batch size (typically 256–1024) [17], removing the primary lever used in batch inference: enlarging batches to improve expert utilization.

A major bottleneck in rollout is straggler sequences. Long-tailed generation lengths create synchronization barriers in synchronous training, forcing all workers to wait for the slowest trajectories. As the batch drains, GPU utilization collapses: once only a few sequences remain, most devices idle.

BatchGen mitigates stragglers via its coroutine callbacks. The OnLongTail callback triggers when the number of active sequences falls below a threshold, allowing the runtime to apply `PARTITION` and redistribute remaining work across idle GPUs, switching from data to tensor parallelism for single long-tail sequences. The same mechanism supports alternatives such as FP8 decoding or speculative decoding without changing the core runtime.

Workload	Framework	H20 GPUs		
		32	64	128
12K-4K	vLLM	866.0	437.2	218.9
	SGLang-Opt	239.2	120.8	61.1
	BatchGen	**139.7**	**66.3**	**33.5**
	BatchGen *	**110.7**	**52.6**	**26.3**
6.5K-2.8K	vLLM	236.9	120.5	61.1
	SGLang-Opt	139.9	71.2	36.0
	BatchGen	**63.9**	**30.4**	**15.5**
	BatchGen *	**59.4**	**28.3**	**14.6**

Table 5: End-to-end time (minutes) for processing 10K requests on DeepSeek-R1. BatchGen * uses SGLang kernels with the BatchGen coroutine runtime.

We evaluate on VeRL [6] with DeepSeek-R1 on 16×H20 GPUs. In a typical setting, each RL iteration uses 256 prompts with one response per prompt, for 256 rollout sequences in total, evenly distributed across the 16 GPUs at 16 active sequences per GPU initially. Generation lengths are heavily long-tailed, so the batch drains unevenly: OnLongTail triggers once the global active batch falls to ≤ 8 sequences whose generation length exceeds 40K tokens. In our traces these final sequences account for 30–80% of rollout time, yet leave at most 0.5 active sequences per GPU on average, exposing substantial idle capacity. BatchGen reclaims this capacity by applying `PARTITION` with FP8 decoding to the stragglers, reducing per-iteration rollout time by 5–10%, with the gain increasing as the final generation length grows, consistent with [16]. Since rollout accounts for 60–80% of training time, these savings translate directly into shorter end-to-end training.

6.4 Scaling to Large Deployments

We evaluate BatchGen at production scale with up to 128 GPUs using two trace-derived workloads: a prefill-heavy setting (12K input, 4K output tokens) and a balanced setting (6.5K input, 2.8K output). To disentangle runtime gains from kernel improvements, we also report results for BatchGen *, which replaces BatchGen's kernels with SGLang's while preserving the coroutine scheduler.

Table 5 shows end-to-end time for 10K requests. Because the evaluated SGLang and vLLM versions exhibit stability issues when scaling DeepSeek-R1 beyond two nodes (16 GPUs), we extend them to 32-128 GPUs by running multiple independent 16-GPU data-parallel groups and aggregating throughput. BatchGen achieves 1.71-1.82× speedup on the 12K-4K workload and 2.2-2.3× on the 6.5K-2.8K workload relative to SGLang-Optimized.

P:D Ratio	1:7	2:6	3:5	4:4	5:3	6:2	7:1
PD Disagg. (min)	137.9	67.4	49.0	38.6	40.1	55.6	112.0
BatchGen (min)				**17.5**			
Speedup	*7.9×*	*3.9×*	*2.8×*	*2.2×*	*2.3×*	*3.2×*	*6.4×*

Table 6: Comparison with PD disaggregation on 128×H20 GPUs (10K requests, 8K-2K workload). Each ratio unit is 16 GPUs. BatchGen requires no ratio tuning.

The performance gains from scaling stem from increased batch sizes: more GPUs provide additional memory capacity to accumulate larger batches before MoE execution, improving expert utilization. However, scaling exhibits sub-linear speedup due to communication overhead in MoE layers—the all-to-all collective for expert routing grows with GPU count. In our implementation, gains plateau beyond 64 GPUs; for the 128-GPU configuration, we deploy two 64-GPU instances and partition the input batch across them.

Scaling beyond single-instance limits. For a fixed model and parallelism strategy, there is a GPU count beyond which adding devices reduces per-GPU efficiency, as MoE all-to-all communication grows while computation per GPU stays constant. BatchGen treats this saturation point as the natural size of a single instance. Larger deployments are formed by replicating multiple independent instances, with an upper-level scheduler partitioning inputs and aggregating outputs. This hierarchical design preserves per-instance efficiency while scaling throughput linearly with instance count.

We compare against prefill-decode (PD) disaggregation [57] on SGLang at 128 GPUs. Each P or D unit consists of 16 GPUs, yielding seven P:D ratios. Table 6 shows that PD disaggregation performance varies by 3.6× across configurations (38.6 to 137.9 minutes), requiring exhaustive profiling to identify the optimal ratio. BatchGen outperforms even the best PD configuration (4:4) by 2.2× without manual tuning.

Production deployment. BatchGen is deployed as the batch-inference engine on an industry partner's production cluster. It runs as independent BatchGen instances launched and managed by an industry-customized Ray [30] orchestrator. Each instance exposes an OpenAI-compatible API, and the scheduler intentionally over-subscribes it, dispatching far more requests than its concurrent capacity. Over-subscription serves two purposes. First, it keeps a large pool of sequences resident on each instance, giving the BatchGen runtime freedom to form batches by `COMBINE`-ing sequences selected from the pool rather than in arrival order (§4.2), which maximizes expert-level batching. Second, it keeps the instance continuously saturated: as sequences complete, the runtime immediately backfills from the resident pool, so concurrency stays high and the GPUs never drain. Completed requests are streamed to an instance-local directory that the scheduler polls periodically to collect results and dispatch new work.

→Models	Mixtral-8×7B		Mixtral-8×22B		DeepSeek-R1-671B	
→Datasets	GSM8K	ChatBot	GSM8K	ChatBot	GSM8K	ChatBot
vLLM	20.4h	N/A	375.0h	N/A	N/A	N/A
DeepSpeed	26.0h	187.4h	116.0h	950.4h	N/A	N/A
FlexGen†	18.8h	156.0h	83.5h	789.0h	N/A	N/A
MoE-Lightning†	<u>7.3h</u>	<u>58.5h</u>	<u>32.5h</u>	<u>295.0h</u>	N/A	N/A
BatchGen‡	**1.7h**	**10.0h**	**5.1h**	**30.8h**	**41.3h**	**328.5h**

Table 7: Time to complete GSM8K (8.5K samples) and ChatBotArena (36K samples) on a single A5000 GPU (24 GB). N/A indicates the system either runs out of memory or would take longer than 1000 hours to complete. Best results in **bold**, second best <u>underlined</u>.

6.5 Working with Limited Memory

We evaluate BatchGen's offloading capability on commodity hardware by running large-scale MoE inference on a single NVIDIA A5000 GPU (24 GB) with a 28-core AMD 7453 CPU and 1 TB of host memory. This setup highlights BatchGen's ability to execute models far beyond GPU capacity.

As shown in Table 7, we report end-to-end processing time on GSM8K (512→256) and ChatBotArena (256→512). FlexGen only supports dense models and MoE-Lightning is not open-sourced, so we reproduce their offloading strategies based on the original papers and integrate them into our runtime†. For the single-GPU case, BatchGen additionally offloads attention computation to the CPU to relieve GPU memory pressure; this optimization is disabled in multi-GPU settings where CPU bandwidth is insignificant relative to total GPU resources‡.

BatchGen achieves significantly higher efficiency under limited resources. Under constrained hardware settings where most baselines cannot complete the dataset within a reasonable time, BatchGen achieves up to a 9.6× speedup. As model size increases, baseline systems fail entirely, while BatchGen continues to finish the workload. FlexGen and MoE-Lightning suffer from extremely small per-expert batch sizes–40-1000× smaller than what is required to fully utilize the GPU–because they retain full MoE layers on device, capping the memory available for batching. In contrast, BatchGen accumulates large batches via coroutine *yields* and fully offloads experts, enabling per-expert batch sizes that reach the compute-bound regime. vLLM remains bottlenecked by on-demand parameter fetching, whereas BatchGen closes this gap by fully overlapping parameter transfers with computation in the coroutine runtime and its buffer.

7 Limitations and Future Work

Batch-size regime and best-effort operation. For a given model, its compute characteristics determine an optimal batch size at the `COMBINE` point that saturates the device. In practice, however, this saturation point cannot always be reached: even

when enough candidate requests are available to batch, the system's memory capacity may be insufficient to hold them, in which case BatchGen forms the largest batch memory permits and operates below saturation on a best-effort basis. Figure 2b illustrates this regime: saturating the compute of every expert requires 16384 tokens to reach the MoE gate. This target is comparatively easy to meet during prefill, where a single sequence of length 16384 suffices, but during decoding it corresponds to a global batch of 16384 concurrent requests—rarely attainable given the long-context requests typical of production. Decoding therefore runs in the best-effort regime. In our limited-memory evaluation (§6.5), a batch size of 6000 still yields gains, bounded by host memory.

Generalization and future work. The coroutine abstraction is general: it applies to any generation workload whose stages have distinct compute characteristics and memory requirements. Our current implementation targets transformer MoE models, a widely deployed architecture that benefits directly from intra-model yielding at attention–MoE boundaries. Extending BatchGen to other models and tasks is future work. Within a single model, vision-language models are a natural candidate: image encoding is substantially more compute-heavy than the language stages, so a yield point after the encoder could let the runtime rebatch the lighter stages. The abstraction also extends across models: where a generation pipeline connects multiple models with distinct computation characteristics—as in omni-modal systems that couple a language model with separate speech or vision models [48,49]—each model can be treated as a coroutine task, applying yield and `COMBINE` at model boundaries to batch each at its own optimal granularity.

8 Related Work

Throughput-optimized offloading systems. Current offloading systems have not targeted sparse modules and long-tailed decoding. FlexGen [41] is designed for dense modules, thus treating MoE layer as a single dense layer for offloading. DeepSpeed [12] and MoE-Lightning [7] are agnostic to expert batching, with a global batch size on the whole model decided. PowerInfer [42] and Fiddler [20] partition computation between the GPU and CPU, with light weighted attention and expert compute running on CPU, aiming for small batch size where compute is not intensive.

Continuous batching for high throughput. Continuous batching (used in vLLM [22], Orca [50], and Llama.cpp [33]) was introduced to address long-tail related TTFT concerns in interactive inference. These frameworks insert small prefill batches into the decoding phase. vLLM [22], Orca [50], and Llama.cpp [33] follow this strategy directly. NEO [19] interleaves prefill and decoding across GPU and CPU resources, while systems such as BlendServe [54] and other micro-batching approaches [27] share the GPU in the temporal domain. The effective average batch size over the entire execution becomes even smaller, limiting throughput.

Batching in training systems. Training fixes batch sizes to ensure gradient stability and convergence, while inference can vary batch sizes freely. In addition, the training systems are optimized for prefill and gradient updates [55], and decoding speed is not a factor. The MoE training line of work [23,25,52] is orthogonal to BatchGen, although BatchGen is compatible with the fixed batch size in RL post-training of LLMs [36].

9 Conclusion

This paper introduces the event-driven sequence coroutine architecture, a new computation model for large-scale batch inference. By enabling fine-grained yielding, combining, partitioning, and migration of sequence computation, BatchGen overcomes structural bottlenecks in sparse models and long-tail decoding, delivering substantial improvements in batch completion time across various applications. Beyond immediate throughput gains, our results suggest broader implications: extending coroutine abstractions to multimodal workflows, and co-designing the coroutine with the underlying high-performance GPU kernels. We believe this architecture opens a rich systems research agenda for the next generation of AI engine design.

References

[1] Amazon AWS. Process multiple prompts with batch inference. https://docs.aws.amazon.com/bedrock/latest/userguide/batch-inference.html, 2025. Accessed: 2025-12-02.

[2] Chenxin An, Shansan Gong, Ming Zhong, Xingjian Zhao, Mukai Li, Jun Zhang, Lingpeng Kong, and Xipeng Qiu. L-eval: Instituting standardized evaluation for long context language models. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 14388–14411, 2024.

[3] Apache HTTP Server Project Members. Apache HTTP server project. https://httpd.apache.org/, 2025. Accessed: 2025-12-06.

[4] Yushi Bai, Xin Lv, Jiajie Zhang, Hongchang Lyu, Jiankai Tang, Zhidian Huang, Zhengxiao Du, Xiao Liu, Aohan Zeng, Lei Hou, Yuxiao Dong, Jie Tang, and Juanzi Li. LongBench: A bilingual, multitask benchmark for long context understanding. In *ACL (1)*, pages 3119–3137. Association for Computational Linguistics, 2024.

[5] BIG-bench authors. Beyond the imitation game: Quantifying and extrapolating the capabilities of language models. *Trans. Mach. Learn. Res.*, 2023, 2023.

[6] Bytedance Seed. verl: Volcano Engine Reinforcement Learning for LLMs . https://github.com/volcengine/verl, 2025. Accessed: 2025-12-09.

[7] Shiyi Cao, Shu Liu, Tyler Griggs, Peter Schafhalter, Xiaoxuan Liu, Ying Sheng, Joseph E. Gonzalez, Matei Zaharia, and Ion Stoica. MoE-Lightning: High-throughput moe inference on memory-constrained gpus. In *ASPLOS (1)*, pages 715–730. ACM, 2025.

[8] Tri Dao, Daniel Y. Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. FlashAttention: Fast and memory-efficient exact attention with io-awareness. In *NeurIPS*, 2022.

[9] Deepseek AI. DeepEP. https://github.com/deepseek-ai/DeepEP, 2025. Accessed: 2025-12-06.

[10] Deepseek AI. DeepGEMM. https://github.com/deepseek-ai/DeepGEMM, 2025. Accessed: 2025-12-06.

[11] DeepSeek-AI. DeepSeek-R1: Incentivizing reasoning capability in llms via reinforcement learning. 2025.

[12] DeepSpeed Team. DeepSpeed. https://github.com/deepspeedai/DeepSpeed, 2025. Accessed: 2025-12-09.

[13] Dayou Du, Shijie Cao, Jianyi Cheng, Luo Mai, Ting Cao, and Mao Yang. Bitdecoding: Unlocking tensor cores for long-context llms with low-bit kv cache, 2025.

[14] Mohamed Y. Eltabakh, Zan Ahmad Naeem, Mohammad Shahmeer Ahmad, Mourad Ouzzani, and Nan Tang. RetClean: Retrieval-based tabular data cleaning using llms and data lakes. *Proc. VLDB Endow.*, 17(12):4421–4424, 2024.

[15] Yao Fu, Leyang Xue, Yeqi Huang, Andrei-Octavian Brabete, Dmitrii Ustiugov, Yuvraj Patel, and Luo Mai. ServerlessLLM: Low-latency serverless inference for large language models. In *OSDI*. USENIX Association, 2024.

[16] Wei Gao, Yuheng Zhao, Dakai An, Tianyuan Wu, Lunxi Cao, Shaopan Xiong, Ju Huang, Weixun Wang, Siran Yang, Wenbo Su, Jiamang Wang, Lin Qu, Bo Zheng, and Wei Wang. RollPacker: Mitigating long-tail rollouts for fast, synchronous RL post-training. 2025.

[17] Jian Hu, Xibin Wu, Wei Shen, Jason Klein Liu, Weixun Wang, Songlin Jiang, Haoran Wang, Hao Chen, Bin Chen, Wenkai Fang, et al. OpenRLHF: A ray-based easy-to-use, scalable and high-performance rlhf framework. In *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*, pages 656–666, 2025.

[18] Albert Q. Jiang, Alexandre Sablayrolles, Antoine Roux, Arthur Mensch, Blanche Savary, Chris Bamford, Devendra Singh Chaplot, Diego de las Casas, Emma Bou Hanna, Florian Bressand, Gianna Lengyel, Guillaume Bour, Guillaume Lample, Lélio Renard Lavaud, Lucile Saulnier, Marie-Anne Lachaux, Pierre Stock, Sandeep Subramanian, Sophia Yang, Szymon Antoniak, Teven Le Scao, Théophile Gervet, Thibaut Lavril, Thomas Wang, Timothée Lacroix, and William El Sayed. Mixtral of experts, 2024.

[19] Xuanlin Jiang, Yang Zhou, Shiyi Cao, Ion Stoica, and Minlan Yu. NEO: saving GPU memory crisis with CPU offloading for online LLM inference, 2024.

[20] Keisuke Kamahori, Yile Gu, Kan Zhu, and Baris Kasikci. Fiddler: CPU-GPU orchestration for fast inference of mixture-of-experts models, 2024.

[21] Kimi Team. Kimi K2: open agentic intelligence, 2025.

[22] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph Gonzalez, Hao Zhang, and Ion Stoica. Efficient memory management for large language model serving with pagedattention. In *SOSP*, pages 611–626. ACM, 2023.

[23] Jiamin Li, Yimin Jiang, Yibo Zhu, Cong Wang, and Hong Xu. Accelerating distributed MoE training and inference with Lina. In *USENIX Annual Technical Conference*, pages 945–959. USENIX Association, 2023.

[24] Percy Liang, Rishi Bommasani, Tony Lee, Dimitris Tsipras, Dilara Soylu, Michihiro Yasunaga, Yian Zhang, Deepak Narayanan, Yuhuai Wu, and et al. Holistic evaluation of language models. *Trans. Mach. Learn. Res.*, 2023, 2023.

[25] Juncai Liu, Jessie Hui Wang, and Yimin Jiang. Janus: A unified distributed training framework for sparse Mixture-of-Experts models. In *SIGCOMM*, pages 486–498. ACM, 2023.

[26] Lin Long, Rui Wang, Ruixuan Xiao, Junbo Zhao, Xiao Ding, Gang Chen, and Haobo Wang. On LLMs-driven synthetic data generation, curation, and evaluation: A survey. In *ACL (Findings)*, pages 11065–11082. Association for Computational Linguistics, 2024.

[27] Frank Sifei Luan, Ziming Mao, Ron Yifeng Wang, Charlotte Lin, Amog Kamsetty, Hao Chen, Cheng Su, Balaji Veeramani, Scott Lee, SangBin Cho, Clark Zinzow, Eric Liang, Ion Stoica, and Stephanie Wang. The streaming batch model for efficient and fault-tolerant heterogeneous execution, 2024.

[28] Xupeng Miao, Gabriele Oliaro, Zhihao Zhang, Xinhao Cheng, Hongyi Jin, Tianqi Chen, and Zhihao Jia. Towards efficient generative large language model serving: A survey from algorithms to systems. *arXiv preprint arXiv:2312.15234*, 2023.

[29] Microsoft Azure. Batch Endpoints. https://learn.microsoft.com/en-us/azure/machine-learning/concept-endpoints-batch?view=azureml-api-2, 2025. Accessed: 2025-12-02.

[30] Philipp Moritz, Robert Nishihara, Stephanie Wang, Alexey Tumanov, Richard Liaw, Eric Liang, Melih Elibol, Zongheng Yang, William Paul, Michael I. Jordan, and Ion Stoica. Ray: A distributed framework for emerging AI applications. In *OSDI*, pages 561–577. USENIX Association, 2018.

[31] NGINX Team. NGINX open source. https://nginx.org/index.html, 2025. Accessed: 2025-12-06.

[32] NVIDIA. TensorRT-LLM. https://github.com/NVIDIA/TensorRT-LLM, 2024. Accessed: 2024-05-17.

[33] Ollama. Ollama. https://github.com/ollama/ollama, 2025.

[34] OpenAI. Batch API. https://platform.openai.com/docs/guides/batch, 2025. Accessed: 2025-12-02.

[35] OpenAI. Gpt-5 system card. Technical report, OpenAI, August 2025. Accessed: 2025-12-11.

[36] Long Ouyang, Jeffrey Wu, Xu Jiang, Diogo Almeida, Carroll L. Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, Alex Ray, John Schulman, Jacob Hilton, Fraser Kelton, Luke Miller, Maddie Simens, Amanda Askell, Peter Welinder, Paul F. Christiano, Jan Leike, and Ryan Lowe. Training language models to follow instructions with human feedback. In *NeurIPS*, 2022.

[37] Pratyush Patel, Esha Choukse, Chaojie Zhang, Aashaka Shah, Íñigo Goiri, Saeed Maleki, and Ricardo Bianchini. Splitwise: Efficient generative LLM inference using phase splitting. In *ISCA*, pages 118–132. IEEE, 2024.

[38] Sundar Pichai. A new era of intelligence with gemini 3. https://blog.google/products/gemini/gemini-3/#note-from-ceo, November 2025. Accessed: 2025-12-11.

[39] Ruoyu Qin, Zheming Li, Weiran He, Jialei Cui, Feng Ren, Mingxing Zhang, Yongwei Wu, Weimin Zheng, and Xinran Xu. Mooncake: Trading more storage for less computation - A kvcache-centric architecture for serving LLM chatbot. In *FAST*, pages 155–170. USENIX Association, 2025.

[40] Gaurav Sahu, Pau Rodríguez, Issam H. Laradji, Parmida Atighehchian, David Vázquez, and Dzmitry Bahdanau. Data augmentation for intent classification with off-the-shelf large language models. In *ConvAI@ACL*, pages 47–57. Association for Computational Linguistics, 2022.

[41] Ying Sheng, Lianmin Zheng, Binhang Yuan, Zhuohan Li, Max Ryabinin, Beidi Chen, Percy Liang, Christopher Ré, Ion Stoica, and Ce Zhang. FlexGen: High-throughput generative inference of large language models with a single GPU. In *ICML*, volume 202 of *Proceedings of Machine Learning Research*, pages 31094–31116. PMLR, 2023.

[42] Yixin Song, Zeyu Mi, Haotong Xie, and Haibo Chen. PowerInfer: Fast large language model serving with a consumer-grade GPU, 2023.

[43] Siddarth Venkatraman, Vineet Jain, Sarthak Mittal, Vedant Shah, Johan S. Obando-Ceron, Yoshua Bengio, Brian R. Bartoldson, Bhavya Kailkhura, Guillaume Lajoie, Glen Berseth, Nikolay Malkin, and Moksh Jain. Recursive self-aggregation unlocks deep thinking in large language models, 2025.

[44] Xuezhi Wang, Jason Wei, Dale Schuurmans, Quoc V. Le, Ed H. Chi, Sharan Narang, Aakanksha Chowdhery, and Denny Zhou. Self-consistency improves chain of thought reasoning in language models. In *ICLR*. OpenReview.net, 2023.

[45] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed H. Chi, Quoc V. Le, and Denny Zhou. Chain-of-Thought prompting elicits reasoning in large language models. In *NeurIPS*, 2022.

[46] Matt Welsh, David E. Culler, and Eric A. Brewer. SEDA: an architecture for well-conditioned, scalable internet services. In *SOSP*, pages 230–243. ACM, 2001.

[47] xAI. Grok 4 model card. Technical report, xAI, August 2025. Accessed: 2025-12-11.

[48] Jin Xu, Zhifang Guo, Jinzheng He, Hangrui Hu, Ting He, Shuai Bai, Keqin Chen, Jialin Wang, Yang Fan, Kai Dang, Bin Zhang, Xiong Wang, Yunfei Chu, and Junyang Lin. Qwen2.5-omni technical report, 2025.

[49] Peiqi Yin, Jiangyun Zhu, Han Gao, Chenguang Zheng, Yongxiang Huang, Taichang Zhou, Ruirui Yang, Weizhi Liu, Weiqing Chen, Canlin Guo, Didan Deng, Zifeng Mo, Cong Wang, James Cheng, Roger Wang, and Hongsheng Liu. vllm-omni: Fully disaggregated serving for any-to-any multimodal models, 2026.

[50] Gyeong-In Yu, Joo Seong Jeong, Geon-Woo Kim, Soojeong Kim, and Byung-Gon Chun. Orca: A distributed serving system for transformer-based generative models. In *OSDI*, pages 521–538. USENIX Association, 2022.

[51] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauly, Michael J Franklin, Scott Shenker, and Ion Stoica. Resilient distributed datasets: A {Fault-Tolerant} abstraction for {In-Memory} cluster computing. In *9th USENIX symposium on networked systems design and implementation (NSDI 12)*, pages 15–28, 2012.

[52] Mingshu Zhai, Jiaao He, Zixuan Ma, Zan Zong, Runqing Zhang, and Jidong Zhai. SmartMoE: Efficiently training sparsely-activated models through combining offline and online parallelization. In *USENIX Annual Technical Conference*, pages 961–975. USENIX Association, 2023.

[53] Shuo Zhang, Zezhou Huang, and Eugene Wu. Data cleaning using large language models. In *ICDEW*, pages 28–32. IEEE, 2025.

[54] Yilong Zhao, Shuo Yang, Kan Zhu, Lianmin Zheng, Baris Kasikci, Yifan Qiao, Yang Zhou, Jiarong Xing, and Ion Stoica. Blendserve: Optimizing offline inference with resource-aware batching. In *Proceedings of the 31st ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, ASPLOS '26, pages 255–273, New York, NY, USA, 2026. Association for Computing Machinery.

[55] Lianmin Zheng, Zhuohan Li, Hao Zhang, Yonghao Zhuang, Zhifeng Chen, Yanping Huang, Yida Wang, Yuanzhong Xu, Danyang Zhuo, Eric P. Xing, Joseph E. Gonzalez, and Ion Stoica. Alpa: Automating inter- and intra-operator parallelism for distributed deep learning. In *OSDI*, pages 559–578. USENIX Association, 2022.

[56] Lianmin Zheng, Liangsheng Yin, Zhiqiang Xie, Chuyue Sun, Jeff Huang, Cody Hao Yu, Shiyi Cao, Christos Kozyrakis, Ion Stoica, Joseph E. Gonzalez, Clark W. Barrett, and Ying Sheng. Sglang: Efficient execution of structured language model programs. In *NeurIPS*, 2024.

[57] Yinmin Zhong, Shengyu Liu, Junda Chen, Jianbo Hu, Yibo Zhu, Xuanzhe Liu, Xin Jin, and Hao Zhang. DistServe: Disaggregating prefill and decoding for goodput-optimized large language model serving. In *OSDI*, pages 193–210. USENIX Association, 2024.

[58] Zixuan Zhou, Xuefei Ning, Ke Hong, Tianyu Fu, Jiaming Xu, Shiyao Li, Yuming Lou, Luning Wang, Zhihang Yuan, Xiuhong Li, et al. A survey on efficient inference for large language models. *arXiv preprint arXiv:2404.14294*, 2024.

[59] Kan Zhu, Yufei Gao, Yilong Zhao, Liangyu Zhao, Gefei Zuo, Yile Gu, Dedong Xie, Zihao Ye, Keisuke Kamahori, Chien-Yu Lin, Ziren Wang, Stephanie Wang, Arvind Krishnamurthy, and Baris Kasikci. Nanoflow: Towards optimal large language model serving throughput. In *OSDI*, pages 749–765. USENIX Association, 2025.

[60] Ruidong Zhu, Ziheng Jiang, Chao Jin, Peng Wu, Cesar A. Stuardo, Dongyang Wang, Xinlei Zhang, Huaping Zhou, Haoran Wei, Yang Cheng, Jianzhe Xiao, Xinyi Zhang, Lingjun Liu, Haibin Lin, Li-Wen Chang, Jianxi Ye, Xiao Yu, Xuanzhe Liu, Xin Jin, and Xin Liu. MegaScale-Infer: Efficient mixture-of-experts model serving with disaggregated expert parallelism. In *SIGCOMM*, pages 592–608. ACM, 2025.

ARTIFACT EVALUATED
USENIX
AVAILABLE

UCCL-Tran: An Extensible Software Transport Layer for GPU Networking

Yang Zhou[†‡*] Zhongjie Chen[§*] Ziming Mao[†] ChonLam Lao[¶] Shuo Yang[†]
Pravein Govindan Kannan[♣] Xizhi Zhang[§] Jiaqi Gao[♡] Yilong Zhao[†] Yongji Wu[†]
Kaichao You[♡] Fengyuan Ren[§] Zhiying Xu[♠‖] Costin Raiciu[◇] Ion Stoica[†]

[†]*UC Berkeley* [‡]*UC Davis* [§]*Tsinghua University* [¶]*Harvard University* [♣]*IBM Research*
[♡]*Independent Researcher* [♠]*Amazon Web Services* [◇]*Broadcom & University Politehnica of Bucharest*

Abstract

Fast-evolving machine learning (ML) workloads have increasing requirements for networking. However, host network transport on RDMA NICs is hard to evolve, causing problems for ML workloads. For example, single-path RDMA traffic is prone to flow collisions that severely degrade collective communication performance. We present UCCL-Tran, an extensible software transport layer to evolve GPU networking. UCCL-Tran decouples the data path and control path of existing RDMA NICs and efficiently runs the control-path transport on host CPUs. This software extensibility brings in transport innovations that cannot be achieved in hardware for ML workloads, e.g., a multipath transport to resolve flow collisions. ML collectives atop UCCL-Tran achieve up to 4.5× higher performance compared to existing RDMA NICs.

1 Introduction

Machine learning (ML) workloads and their requirements for networking are evolving rapidly. Less than ten years ago, deep neural networks only had millions of parameters, and were trained atop hundreds of CPUs/GPUs with parameter servers or allreduce collective communication [62]. After five years, large language models (LLMs) began to surge with billions of parameters, and were trained atop thousands of more powerful GPUs with multi-level parallelism and diverse collectives like allreduce, allgather, and reduce-scatter [13, 40]. In the recent two years, large-scale LLM serving has become the norm; prefill-decode disaggregation [109], as an efficient serving technique, requires fast peer-to-peer communication. This year, serving Mixture-of-Experts (MoE) models like DeepSeek-V3 [66] became very popular, featuring challenging all-to-all communication among hundreds of GPUs.

However, networking techniques especially the host network transport on RDMA NICs are hard to adapt and evolve to better suit the needs of ML workloads. Essentially, hardware changes are time-consuming and take much longer time than software changes. This can lead to a mismatch between the application needs and existing hardware optimizations, which often translates into poor performance. For example, Meta has reported that DCQCN [111]—a popular congestion control (CC) algorithm in datacenters supported by RDMA NICs—does not work well for LLM training workloads with low flow entropy and high traffic burstiness [34]. As a result, Meta decided to disable the CC support in NICs and instead implement traffic scheduling at the application layer. Similarly, DeepSeek disabled the CC when running large-scale all-to-all for serving MoE models [29]. However, running a large-scale RDMA network without CC is brittle, as it can lead to deadlocks, head-of-line blocking, and pervasive congestion [12, 35, 41, 45, 73, 111].

In another example, Alibaba has observed severe performance degradation for collective communication during LLM training. This was due to the high level of flow collisions, which in turn was caused by the RDMA NICs supporting only single-flow/path per connection [87]. To avoid this problem, Alibaba has redesigned the network topology for LLM training using a rail-optimized dual-plane architecture. However, such a redesign is costly to build and maintain. As we will show in §3.2, a software-only solution implementing multipath at the application level would have avoided such topology changes.

In §2.2, we discuss four more examples where ML workloads require adapting RDMA hardware support, including network incast in MoE serving, semi-reliable gradient transmission, efficient loss recovery, and avoiding vendor lock-in. Furthermore, the existing hardware-baked host network transport layer makes it difficult to productize new research proposals [16, 59, 85, 102] to improve the performance of ML workloads in real-world scenarios.

These examples point to *network extensibility* as one of the key challenges in datacenter networks. In this paper, we focus on addressing this challenge in the context of RDMA-enabled hosts, by proposing and implementing a software-only extensible transport layer called UCCL-Tran (abbreviated as uTran). This software-only approach makes it much easier to

[*]Equal contribution.
[‖]This work does not relate to the position at Amazon.

efficiently support new ML workloads, e.g., CC algorithms that can better support ML traffic than existing DCQCN, and multipathing to mitigate flow collisions. To implement this approach, we need to address two challenges: (1) How to decouple the data and control paths for existing RDMA NICs? The data path handles network data transfers for the GPU, while the control path manages transport control decisions like CC, packet reliability, and multipath load balancing (LB). Only if the control path is decoupled from the data path can we implement it on the CPU. (2) How can we achieve hardware-level performance with the control path running on the CPU? This is challenging given the high inter-server bandwidth, e.g., 3.2 Tbps [4, 71].

To address the first challenge, UCCL-Tran repurposes the features of existing RDMA NICs to achieve efficient data and control path separation. In particular, UCCL-Tran leverages RDMA Unreliable Connection (UC) to bypass hardware-baked CC and packet reliability logic, and uses RDMA immediate data to pass transport control states between sender and receiver CPUs. For some NICs that do not support UC, such as AWS EFA [92], UCCL-Tran leverages the scatter-gather feature over RDMA Unreliable Datagram (UD) to achieve separation.

To address the second challenge, UCCL-Tran leverages an array of techniques tailored to the unique characteristics of ML workloads. On the one hand, UCCL-Tran leverages GPUDirect [28] to alleviate CPU overhead. On the other hand, UCCL-Tran employs control coalescing to make transport control decisions for every 32KB data chunk rather than every packet; this works well as many transport decisions such as CC do not require per-packet reaction, but just per-RTT to avoid overreaction [39, 64]. As a result, UCCL-Tran can handle 400 Gbps unidirectional traffic with a single CPU core, and achieve the same message latency as hardware-based transport. UCCL-Tran leverages 256 RDMA QPs (Queue Pairs) per connection to perform multipathing, without worrying about the high QP context swapping overhead highlighted in previous work [104]; this is because ML workloads feature bulk data transfer with mostly MTU-sized large packets, which effectively amortizes the swapping overhead. UCCL-Tran's software transport is practical and economical in modern GPU servers, which usually have hundreds of powerful CPU cores; in fact, these CPU cores are often heavily underutilized. For example, several companies report cluster-wide CPU utilization of 20%-45% in their GPU clusters [44, 52]. From our private conversation with the model training team of a major GPU vendor, their CPU utilization is averagely 14.5% out of 128 cores, when using Megatron-LM [82] for model training.

UCCL-Tran provides an expressive yet easy-to-use interface. To demonstrate the versatility of this interface and the power of UCCL-Tran's extensibility, we use three case studies. First, we implement a multipath transport protocol that mitigates flow collisions by leveraging packet spraying, i.e., randomly sending packets from a single connection across different paths [31]. Compared to existing RDMA transport on hardware, ML collectives using our transport achieve up to 4.5× higher throughput on NVIDIA ConnectX-7 NICs, and up to 1.9× higher on Broadcom Thor-2 NICs with rail-optimized topology, which improves end-to-end LLM training throughput by 7.5%. Second, we implement a receiver-driven protocol EQDS [85] to handle network incast in MoE-like workloads, achieving 4.9× better message tail latency over InfiniBand built-in transport. Third, we implement selective retransmission [70] for efficient transport loss recovery, and show its superiority over RDMA hardware transport under packet loss. These case studies highlight that UCCL-Tran can effectively enable innovations in the transport protocols that are hard to implement in today's network stack without expensive and time-consuming changes. UCCL-Tran is open-sourced at https://github.com/uccl-project/uccl.

2 Background and Motivation

2.1 RDMA and Collectives

GPU networking is extremely heterogeneous. At the high level, GPU collective communication libraries like NVIDIA NCCL [83] and AMD RCCL [7] use RDMA and kernel TCP (non-RDMA) for inter-server networking, with RDMA preferred since it is faster and more efficient. RDMA provides various communication primitives called Queue Pairs (QPs) including Reliable Connection (RC), Unreliable Connection (UC), and Unreliable Datagram (UD):

- RC provides one-to-one message semantics (up to 1GB per operation), with NIC hardware handling packet reliability and CC. Some vendors like NVIDIA allow disabling CC.
- UC also provides one-to-one message semantics but without NIC hardware logic for packet reliability or CC.
- UD provides one-to-many datagram semantics, i.e., under one MTU per operation, without packet reliability or CC.

Intuitively, one can think of RDMA QPs as TCP/UDP sockets: RC is like a TCP socket that provides one-to-one message abstraction with reliability; UD is like a UDP socket that provides one-to-many datagram semantics without reliability; UC is one-to-one without reliability. Some cloud providers build their own RDMA NICs and QPs. For example, AWS EFA NICs replace RC with SRD (Scalable Reliable Datagram) [93] that implements datagram semantics, multipathing, packet reliability, and CC.

To use RDMA NICs, the CPU issues *verb* operations such as two-sided send/recv and one-sided read/write that transfer data over QPs. Internally, the verb builds a Work Queue Entry (WQE) and performs MMIO-write into the RDMA NIC registers. Upon completion, and depending on whether the verb is two-sided, the RDMA NIC generates a Completion Queue Entry (CQE) for the software to consume. Note that UD supports only two-sided verbs, UC supports all verbs except RDMA read, while RC supports all verbs. RDMA traf-

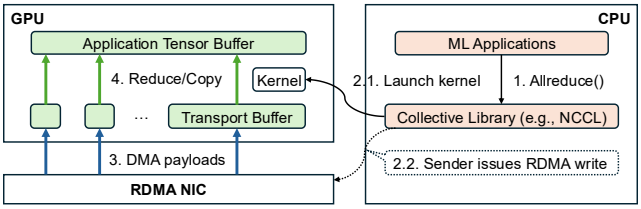


Figure 1: Collective communication over RDMA (receiver side). The receiver directly receives data into the GPU memory (e.g., GPUDirect [28]); the sender side works similarly except that it will additionally issue an RDMA write in step 2.

fic could go through different network fabrics such as RoCE (RDMA over Converged Ethernet) and InfiniBand.

Figure 1 shows an overview of how collective communication uses RDMA. Once the ML application calls a collective like allreduce, the collective library launches a *reduction kernel* on each participant GPU to handle data reduction/copy. Next, the sender CPU issues multiple RDMA writes over RC QPs to transfer data chunk by chunk. The receiver CPU polls completion flags in its memory, which the sender sets upon write completions. The library manages a set of *transport buffers* on the GPU memory to buffer the RDMA data, and relies on the GPU kernel to copy data between the transport buffers and application tensor buffers. The GPU kernel also performs reductions (e.g., sum, max) on the transport buffers (from multiple sender GPUs) into the tensor buffers.

2.2 Motivation on Extensibility

Host network transport on RDMA NICs is hard to evolve compared to software applications. This creates problems for fast-evolving ML workloads. We have shown two such examples in §1; below, we give four additional examples to motivate the need for transport layer extensibility.

Receiver-driven CC for incast. Recent MoE serving workloads are prone to network incast problems. In DeepSeek's online deployment of their 671B V3 model [66], each of the 320 GPUs holds a single expert module, where hidden states are exchanged between expert modules across GPUs, i.e., Expert Parallelism (EP). As the request pattern and load change over time, some experts become much hotter than others, receiving more network traffic from other experts leading to network incast issues. DeepSeek reports the hottest expert could receive 10× more load than the average one. Expert load balancing algorithms such as EPLB [30] try to balance the load by dynamically replicating experts. However, this happens at a much slower pace (e.g., 10 minutes [29]) to avoid the high cost of moving experts, thus unable to handle transient incast. Such transient network incast can be better handled by receiver-driven CC [43, 85], which controls last-hop congestion—unfortunately, there is no receiver-driven CC on commercial RDMA NICs.

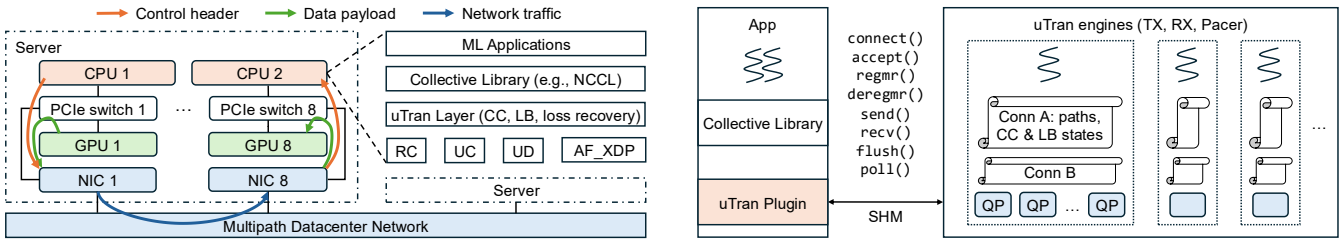
Application-transport codesigns. Codesigning applications and transport behaviors could bring huge performance benefits. For example, recent work MLT [102] customizes loss recovery behaviors for ML training to allow semi-reliable transmission based on the gradient importance from applications. Despite achieving great performance improvement, it is not feasible to integrate MLT into existing RDMA NICs even for the latest NVIDIA ConnectX-7 [79], due to a lack of enough programmability.

Inefficient loss recovery. RDMA NICs are known to perform poorly under packet loss, especially for old-generation NICs [63, 73, 97, 104]. This is caused by the inefficient go-back-N retransmission logic hardcoded on these NICs due to limited on-chip SRAM constraints. As a result, RDMA deployment normally requires Priority Flow Control (PFC) to achieve a lossless network fabric. However, PFC may lead to deadlocks, head-of-line blocking, and victim flows [45, 73], and its likelihood is higher as the GPU networking bandwidth keeps increasing (for the reason, we kindly refer to page 4 in [45]). If we could extend the transport layer of GPU networking with more efficient selective retransmission, we could better handle packet loss and rely less on PFC [73].

Heterogeneous NICs. Datacenters usually consist of multiple generations and vendors of RDMA NICs due to continuous expansion, cost optimization, and to avoid vendor lock-in. While NVIDIA, Broadcom, AMD, and more vendors all have 400 Gbps RDMA NICs for ML [9, 18, 79], they come with subtly different control path logic such as packet reliability and CC. In practice, this heterogeneity reduces achievable bandwidth by 2-33× when communicating between NICs from different generations/vendors, as reported by Alibaba [63]. Prior work Flor [63] has shown that extensibly aligning these NICs' control path logic in software could avoid such issues.

2.3 Prior Work on Extensibility

Leveraging SmartNICs. Several recent efforts aim to make RDMA transport programmable by offloading it to the SmartNIC RISC cores, but they have constraints in extensibility and performance. Google Falcon SmartNICs [38, 39] only support programming rate update actions for latency-based Swift CC [58] and path selection decisions with limited paths [88]; similar constraints also apply to firmware updates on hardware RDMA NICs. AMD Pensando SmartNICs [9] support using P4 language [17] to program their transport layer, but P4 has limited programmability, e.g., hard to implement efficient loss recovery; it is also unclear if they could support receiver-driven CC. FPGA-based SmartNICs provide higher performance but have limited extensibility due to hardware resource constraints [104]. AWS EFA SmartNICs [92] implement a proprietary multipath reliable transport SRD [93] with out-of-order packet delivery using NIC ARM cores, to address network congestion in HPC and ML workloads. The SRD protocol is implemented using EFA-specific firmware and supports live upgrade with good extensibility. However, we empirically find that the EFA SmartNICs on AWS `p4d.24xlarge` GPU VMs perform poorly for connection-intensive all-to-all



(a) UCCL-Tran architecture.

(b) UCCL-Tran threading model.

Figure 2: Overview of UCCL-Tran extensible transport for GPU networking. We assume a common intra-server topology for GPU servers [23] where individual PCIe switches directly connect a few GPUs, NICs, and CPU, providing high-bandwidth data transfer among them.

collectives (see §6.1). We attribute this to the limited processing power and cache capacity of SmartNIC ARM cores due to power constraints, as also demonstrated by prior research on publicly available SmartNICs [67, 86, 90].

Note that the above all-to-all measurement uses the AWS EFA NICs on the `p4d.24xlarge` instances, and thus it may not apply to their new-generation EFA NICs on `p5/p5en/p6` instances. Specifically, the ever-changing EFA firmware and upgraded EFA hardware may lead to different performance results for connection-intensive all-to-all collectives. In general, if these SmartNICs get upgraded with higher processing power and cache capacity to better handle all-to-all, we view it as an echo to UCCL-Tran's high-level methodology of software extensibility for GPU networking.

Leveraging CPUs. A line of work has leveraged host CPUs to make better control decisions in GPU networking. ZeroNIC [97] modifies the NIC hardware to run the control path of RDMA transport on CPUs, while leaving the data path on the NIC following GPUDirect. In contrast, UCCL-Tran aims to be more practical without modifying existing hardware; UCCL-Tran by design supports efficient multipathing, compared to ZeroNIC's single path transport. Flor [63] leverages RDMA UC to bypass the control path of the RDMA hardware and implements flexible software control on CPUs. Flor targets CPU-based storage applications with 100 Gbps traffic per server, while UCCL-Tran targets more network-intensive ML applications that have 3.2+ Tbps traffic per server [4, 71] and develops multipathing to avoid flow collisions. To this end, UCCL-Tran adopts several different designs such as multi-QP and connection splitting (see §3.2 and §3.3).

Other efforts design transport protocols to address specific network challenges. Ultra Ethernet Consortium (UEC) [22] standardizes several multipath transport protocols with packet spraying to address flow collisions for ML workloads: a sender-driven one based on STrack [59] and SMaRTT-REPS [16], and a receiver-driven one based on EQDS [85]. Prior to UEC, MP-RDMA [68] and MPTCP [77] designed multipath protocols for CPU workloads to improve robustness under network failures. Overall, these protocols leverage various congestion signals such as Explicit Congestion Notification (ECN) [1], RTT [72], and packet trimming status [21, 43, 85] to make multipath CC and LB decisions.

3 UCCL-Tran Design

Figure 2a shows the high-level architecture of UCCL-Tran. UCCL-Tran layer sits between the collective library such as NCCL and the low-level communication primitives exposed by NIC hardware, e.g., RC, UC, and UD for RDMA NICs and AF_XDP [98] (a user-space fast packet IO) for non-RDMA NICs. ML applications use collective APIs such as `allreduce` and point-to-point APIs such as `SendRecv` exposed by the collective library without directly interacting with the UCCL-Tran layer. Both the collective library and the UCCL-Tran layer are compiled into individual shared libraries, i.e., `libnccl.so` and `libnccl-net.so` for NCCL, providing a drop-in replacement for ML applications without code modification or recompilation. UCCL-Tran leverages the network plugin system of existing collective libraries [6, 24] to avoid changing the library code in most cases, with exceptions for UCCL-Tran over UD that requires a slight code modification (detailed in §3.1). For brevity, the remaining paper targets RDMA NICs such as NVIDIA ConnectX NICs and AWS EFA NICs [92]; we will explicitly mention when targeting non-RDMA NICs.

Figure 2b shows the threading model of the UCCL-Tran layer. The UCCL-Tran plugin interacts with a group of UCCL-Tran engine threads via shared memory to create connections, register/deregister GPU memory regions with the RDMA NICs, and send/recv/flush/poll network messages. Each engine thread runs TX, RX, and pacing functionalities of UCCL-Tran multipath reliable transport for multiple UCCL-Tran connections. As described in §3.1, UCCL-Tran engines instruct the RDMA NIC to receive network data, split control headers and application data payloads, and directly DMA them into the CPU and GPU memory separately. The whole process bypasses the packet reliability and CC logic on RDMA NIC hardware as much as possible by using proper RDMA primitives. After getting control headers in CPU, UCCL-Tran engines make transport decisions such as CC, LB, and handling packet loss and reordering. Since these decisions are executed by a normal user-space process on the CPU, instead of on RDMA NIC hardware, they can be easily extended by

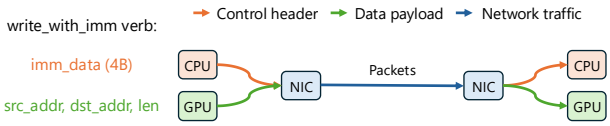


Figure 3: Leveraging RDMA `write_with_imm` to separate control header and data payload for UC/RC.

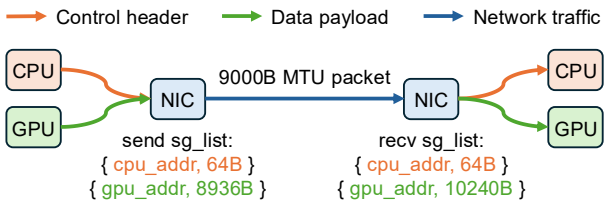


Figure 4: Leveraging RDMA send/recv scatter-gather to separate control header and data payload for UD.

collective library or ML application developers.

In UCCL-Tran, all connections between a specific pair of NICs share the same set of QPs (e.g., 256), including the case of multiple GPUs sharing one NIC. This design fully harnesses the underlying multipath datacenter network while not burning too many QPs (§3.2). UCCL-Tran further integrates a bag of techniques to run software transport as efficiently as possible, e.g., control coalescing, connection splitting, chained posting, and more (§3.3). UCCL-Tran also supports extending transport for non-RDMA NICs through the AF_XDP user-space packet IO, bypassing conventional kernel TCP stacks. We now describe them in the rest of this section.

3.1 Separating Control Path and Data Path

The overall goal of separating the control path and the data path is to enable running extensible transport on the CPU, while efficiently transferring data to/from GPUs in a GPUDirect manner. This goal has three specific aspects: (1) We should involve as little control logic as possible in the data path to let the CPU make more transport decisions, like CC and packet reliability. (2) We must achieve GPUDirect for the data path efficiency [36, 97]. (3) We should support heterogeneous RDMA NICs. For example, NVIDIA NICs support UC, while Broadcom and AWS EFA do not.

UC as the preferred QP. UCCL-Tran chooses UC as the preferred QP whenever available on the RDMA NIC, similar to Flor [63], because it supports efficient segmentation and reassembly offloading to the NIC (in contrast to UD), while bypassing hardware-baked CC, loss recovery, and out-of-order packet handling (in contrast to RC). As shown in Figure 3, UCCL-Tran uses the efficient RDMA verb of write with immediate to transfer data chunks over UC; this verb operates in a two-sided mode, so that both sender and receiver CPUs can react to the data transfers. For the data path, the sender CPU specifies the addresses of source and destination data chunks when issuing the verb, and then the sender NIC will automatically segment the source data chunk into MTU-sized packets with packet headers prepended and send them out. When receiving these packets, the receiver NIC will remove the packet headers and reassemble payloads into the *contiguous* memory region specified by the sender CPU.

For the control path, write with immediate allows carrying a 32-bit `imm_data` from the sender CPU to the receiver CPU, serving as the control header for UCCL-Tran transport. Figure 3 shows an example. UC guarantees that any successfully arrived chunk will generate a CQE with the `imm_data` embedded, which is then consumed by the receiver CPU. Within the 32-bit budget, UCCL-Tran allocates 8 bits for connection ID and 7 bits for message ID, supporting 256 connections for each pair of NICs on a UCCL-Tran engine and 128 in-flight messages for each connection, which is sufficient for collective communication. UCCL-Tran then allocates 8 bits to the chunk sequence number (CSN) to identify the position of a chunk in the message being transferred. Another 1 bit is allocated for marking the last chunk of a message. The remaining 8 bits are reserved for more advanced CC such as receiver-driven ones (see §4.2).

RC with CC disabled. In practice, UC is not always supported across different RDMA NIC vendors [63], e.g., Broadcom [18]. In these cases, UCCL-Tran will opt to RC with CC configured to be disabled, and then leverage RDMA write with immediate in a similar way as UC. On the one hand, RC prevents UCCL-Tran from customizing the packet reliability mechanism baked in NIC hardware; on the other hand, it allows faster ACKs and more precise RTT estimation in hardware.

UD as the last resort. Some RDMA NICs do not allow disabling CC for their RC QPs, e.g., AWS EFA NICs (to be precise, EFA NICs do not have RC but only SRD). To support them, UCCL-Tran leverages UD at the cost of higher CPU usage compared to UC and RC. On the one hand, UD totally bypasses any hardware control logic on RDMA NICs, well-aligned with our goal. On the other hand, UD only supports sending/receiving MTU-sized data (i.e., no segmentation or reassembly offloading); so UCCL-Tran needs to consume more CPU cycles to do segmentation and reassembly.

Challenge of separation. One key challenge for UCCL-Tran over UD is that UD does not support RDMA write with immediate (but only the `send/recv` verbs), so that UCCL-Tran over UD cannot designate the immediate data as the transport control header. Then *how can UCCL-Tran separate control header and data payload using UD (i.e., placing them to CPU and GPU separately)*? UCCL-Tran must guarantee that the control header and data payload are *fate-sharing* in terms of loss status and arrival order, so that UCCL-Tran can make valid transport decisions based on the control header. One strawman solution would be transferring the control header and data payload together as a single packet into the destination GPU memory, and then the CPU reading the control header from the GPU memory. But this incurs additional

performance overhead.

UCCL-Tran's approach is to leverage the scatter-gather feature to let the NIC hardware automatically merge the control header and data payload during RDMA send, and split these two during RDMA recv; Figure 4 shows an example. On the sender side, the CPU issues an RDMA send verb with a two-entry `sg_list` that specifies the control header address+length on the CPU, and the data payload address+length on the GPU. The RDMA NIC will then read the header and payload from the CPU and GPU respectively, and merge them into a single network packet to send out, as long as the total length does not exceed the MTU size. On the receiver side, the CPU pre-posts a recv verb with a two-entry `sg_list` that specifies the receiving address+length for the header and payload, respectively. Note that the header length must be a fixed value that the sender and receiver agree on, e.g., 64B in this example; the payload length specified in the recv verb need not exactly match the send verb, but should be no smaller. Later, when the packet arrives at the receiver NIC, the NIC will automatically split the header and payload across CPU and GPU, following the boundary of the fixed header length. UCCL-Tran's approach always keeps the control header fate-sharing with the data payload, and avoids the CPU reading any extra header from the GPU.

Challenge of reassembly. UCCL-Tran over UD still faces another challenge of *how to correctly and efficiently reassemble packets on the receiver GPU*. Recall that UD does not support reassembly offloading to the NIC, and only allows sending/receiving a single packet in one verb (§2.1). We note that sender-side segmentation is relatively easy, as the CPU could partition the transport sending buffer into individual data payloads (based on MTU size), and specify their addresses in send verbs. However, for the receiver-side reassembly, even if the CPU pre-posts recv verbs that specify in-order individual data payload addresses drawn from the transport receiving buffer, the packets will land into the buffer in an out-of-order manner due to packet loss or reordering over the multipath network. This reassembly challenge is unique to UD, as UC/RC allows the sender to directly specify the receiver-side GPU buffer addresses in the write with immediate verb.

Addressing this challenge requires some form of *scattered memcpy* GPU kernel that copies out-of-order data payloads to the transport receiving buffer (following the right order given by the receiver CPU). But the question is where to launch and run such a kernel. To avoid extra kernel launching overhead, UCCL-Tran chooses to fuse such scattered memcpy operations into the existing reduction kernel in collective libraries (§2.1). Our fused kernel will first do scattered memcpy to copy out-of-order data payloads into the transport buffers, then do the original reduction work from the transport buffers to the application tensor buffers. The only overhead of this approach is the extra GPU memory bandwidth consumption, but this is bounded by the network bandwidth. Given the high GPU memory bandwidth (e.g., 1.6-2.0 TB/s in A100), such extra bandwidth consumption is negligible.

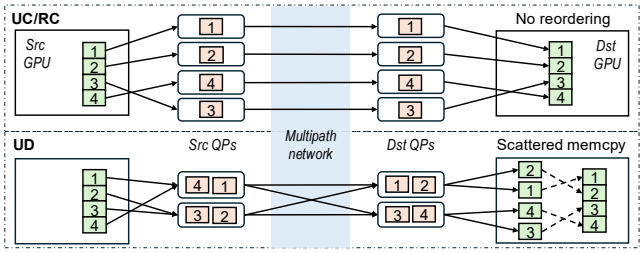


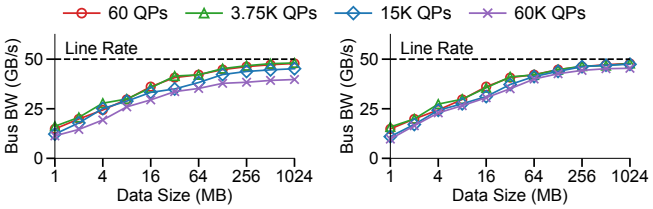
Figure 5: Multipathing and handling packet reordering in UCCL-Tran.

For non-RDMA NICs, UCCL-Tran builds a reliable transport atop UDP with the AF_XDP technique, an efficient kernel socket that lets the NIC directly DMA network packets to user-space memory regions. We choose AF_XDP as it achieves similar high performance as DPDK [110], but is kernel-native and does not require special NIC drivers, thus easy to deploy [100]. Similar to how collective libraries such as NCCL use kernel TCP for non-RDMA NICs, UCCL-Tran over AF_XDP does packet reassembly on the CPU, followed by a `cudaMemcpy()` to transfer the received message to the GPU.

3.2 Harnessing Multipath

One of the key motivations for GPU network extensibility is to harness the multipath capacity of modern datacenter networks (§2.2). UCCL-Tran achieves this by using multiple UC, RC, or UD QPs, as shown in Figure 5. Basically, network traffic from different QPs will likely go through different network paths, as both RoCE and Infiniband usually use ECMP (Equal-Cost Multi-Path) for multipath routing with source and destination QP numbers as the hash inputs [34, 81]. For UC and RC, UCCL-Tran by default uses 256 QPs, which provides maximum 256 different network paths as used by recent transport research [16, 59]. For UD, UCCL-Tran uses a much smaller number of QPs by combining different source and destination QPs. For example, 16 source UD QPs and 16 destination UD QPs will provide maximum 16×16=256 different network paths, because for connection-less UD, each source QP can send packets to any destination QP. UCCL-Tran also supports a configurable number of QPs for different collectives, e.g., a smaller number might work well for all-to-all with relatively high entropy [34]. To avoid burning too many QPs, especially when the collective library creates multiple connections between the same pair of NICs (e.g., because of multiple GPUs sharing one NIC), UCCL-Tran lets all these connections *share the same set of QPs*.

We note that making this multi-QP design choice is not trivial, especially as a line of prior work has highlighted the severe QP scalability issues on RDMA NICs [20, 53, 54, 57, 61, 75, 104]. For example, SRNIC [104] reports ∼23% bandwidth drop when scaling RC QPs from 256 to 512, and 46% drop



(a) For RC QPs. **(b)** For UC QPs.

Figure 6: UCCL-Tran all-to-all network bandwidth on CX_IB testbed (16 400G NICs, see §6 for details) with different numbers of QPs per NIC. 60 QPs per NIC correspond to the typical NCCL QP scaling factor of 4 (i.e., 4 QPs per connection) in production [34, 59]. 60K QPs per NIC essentially model the QP swapping overhead for UCCL-Tran all-to-all across 241 GPUs using 256 paths per connection (i.e., 61440/256=240, plus 1 to include itself) or 961 GPUs using 64 paths. This has covered the largest-scale collective as far as we know, i.e., 256 GPUs by Meta [34] and 320 GPUs by DeepSeek-V3 [66].

when scaling to 16k. This bandwidth drop is caused by the *QP swapping overhead*: the NIC can only hold/cache limited QP contexts on its SRAM, and must spill/swap the excessive rest to the host DRAM over PCIe, incurring frequent QP swapping. Surprisingly, we do not observe such a severe performance drop for collective communication—Figure 6 shows only ∼17% drop when scaling RC QPs from 60 to 60k, and a negligible drop for UC (that has a smaller QP context size than RC).

There are two reasons behind this counter-intuitive phenomenon. First, ML workloads feature large messages, and thus collectives mostly transfer MTU-sized packets; such large transfers effectively amortize the QP swapping overhead. Second, with GPUDirect, GPU data transfers only go through the PCIe switch, but not the PCIe root complex that is connected to the CPU [56]; therefore, there is no PCIe contention between GPU-NIC traffic and CPU-NIC traffic (that is incurred when NIC swaps QP contexts and CPU posts verbs). In contrast, prior work focuses on *small-message CPU workloads* such as in-memory Key-Value stores that only transfer dozens of bytes at once for each QP, thus bottlenecked by the QP swapping overhead and PCIe traffic contention. Later in §3.3, we show that UCCL-Tran optimizes transport efficiency by transferring data in a larger chunk granularity (e.g., 32KB), further reducing the QP swapping overhead.

For non-RDMA NIC multipathing, UCCL-Tran specifies different UDP ports in packets before sending them out in AF_XDP. This adds no overhead compared to single-path transport.

Handling out-of-order packets. Many factors could cause out-of-order packet delivery, including multipathing, packet loss, and the unpredictable multi-QP scheduler in RDMA hardware [104]. Existing RDMA NICs perform poorly when handling out-of-order packets, as they cannot maintain large reordering buffers and states due to limited on-chip SRAM constraints [73, 104]. In contrast, UCCL-Tran is able to handle out-of-order packets efficiently thanks to its software flexibility and separation of data and control paths. Basically, UCCL-Tran follows typical TCP designs with `seq` and `ack` numbers to guide packet reordering, fast retransmission (upon duplicate ACKs), and timeout retransmission. UCCL-Tran sets a larger duplicate ACK threshold for fast retransmission, instead of the default three in TCP, to accommodate more frequent packet reordering caused by multipathing. Different from TCP, UCCL-Tran maintains its packet reordering buffers in the GPU memory and lets the NIC directly DMA network data there. Figure 5 depicts this process with examples. For UC/RC, the reordering buffers are individual data chunks, and the sender CPU specifies in-order chunk addresses when posting verbs. For UD, the reordering buffers are individual packet payloads, and the GPU reduction kernel reorders packets when copying them into the transport buffers (§3.1).

3.3 Towards Efficient Software Transport

So far, we have discussed how UCCL-Tran decouples the control path and data path to make flexible transport decisions on the CPU, and how UCCL-Tran achieves multipathing. The next question is *how to efficiently implement a software multipath transport to support the high bandwidth in GPU networking*. This is challenging as a single GPU server could have 8×400 Gbps RDMA NICs, totaling 3.2 Tbps bandwidth bidirectionally [46]; the next generation RDMA NIC will achieve 800 Gbps [27], rendering 6.4 Tbps bandwidth. As a reference, Google's software transport Snap [69] can handle 80 Gbps traffic on a CPU core (though they do not use RDMA NICs). Our goal is to use 1 CPU core to handle 400G *unidirectional* traffic (i.e., 2 cores for 400G bidirectional traffic; excluding possible pacer cores for receiver-driven CC). To this end, we leverage the following techniques:

Run-to-completion execution. Each UCCL-Tran engine thread runs RX, TX, pacing, timeout detection, and retransmission functionalities for a set of connections in an efficient run-to-completion manner [14, 51]. UCCL-Tran employs Deficit Round Robin (DRR) [94] scheduling to fairly multiplex one engine thread among multiple functionalities and connections.

Connection splitting. To handle 400+ Gbps traffic per NIC more efficiently, UCCL-Tran pivots away from the Flor [63] design of a single CPU core for one connection, but leverages multiple cores for one connection with connection splitting. Basically, UCCL-Tran equally partitions the 256 QPs among all engine threads responsible for a specific NIC; each engine thread gets its own connection states for CC and LB, forming a sub-connection. Within each sub-connection, UCCL-Tran uses RDMA SRQ and SCQ (Shared Recv/Completion Queues) to reduce the overhead when polling multiple recv and completion queues. The application threads atop the UCCL-Tran plugin are responsible for choosing the least-loaded engine (e.g., the engine with the least unconsumed messages) when dispatching messages via SHM. In this way,

UCCL-Tran could scale transport processing of a single connection to multiple cores, and handle transient load imbalance among CPUs at runtime. It also reduces TX packet bursts by avoiding sending all messages at once from a single core.

Control coalescing. There is an inherent tradeoff between the control decision granularity and software transport efficiency. One could run CC, LB, and reliability logic for each packet to achieve precise control of the transport behaviors, at the cost of consuming more CPU cores. Alternatively, one could relax the control granularity by coalescing several same-path packets and making control decisions together, thus with lower CPU consumption. For UC/RC, this also means an RDMA write could directly transmit several packets as a single data chunk, leveraging NIC-offloaded segmentation and reassembly. UCCL-Tran employs this control coalescing design with 32KB chunk size as default, striking a balanced tradeoff. Under this chunk size, UCCL-Tran can saturate 400 Gbps unidirectional bandwidth with 1 CPU core (§6.4.3), while not severely disrupting transport behaviors/performance (see our packet-level simulation in §C.5.1). Nevertheless, UCCL-Tran could also adaptively adjust chunk size based on the congestion level, e.g., switching to a small chunk size to make more precise control when congestion window (`cwnd`) drops below a threshold or severe packet loss happens.

Chained posting. UD does not support NIC offloading for segmentation and reassembly, thus it incurs more MMIO writes than UC/RC when issuing send/recv verbs (e.g., for individual packets). To reduce such overhead, UCCL-Tran leverages the chained posting feature of RDMA NICs to issue one MMIO write for posting up to 32 send/recv verbs. Concretely, the WQEs of these 32 verbs are chained together through the `next` pointer in previous WQEs, and get posted to the RDMA NIC in one MMIO write.

3.4 Congestion Signals

Relatively restricted congestion signal is a common limitation for software-based reliable transport atop RDMA NICs, including UCCL-Tran and Flor [63]. This is because existing RDMA NICs consume packet headers that contain congestion signals like ECN marks [1] and packet trimming status [21, 43, 85], and deliver only the packet payload to the software. Fortunately, the software can still use the RTT congestion signal by leveraging hardware TX/RX timestamping supported by many RDMA NICs, and rely on packet loss as the last-resort congestion signal. Therefore, our current UCCL-Tran implementation uses per-path RTT and packet loss to detect congestion and choose paths. In fact, latency-based CC and LB are being widely used in Google's datacenters [58, 88].

Signal fidelity. Running CC and LB in software using RTT also raises signal fidelity concerns. Overall, there are three factors affecting the fidelity: ① accurate RTT estimation at the sender, ② CC decision delay at either the sender or receiver, i.e., the software delay from receiving the congestion signal (e.g., ACK-derived timestamps) to updating the congestion window/rate, and ③ ACK turnaround delay at the receiver, i.e., delay between receiving data chunks and sending back ACK. For ①, UCCL-Tran leverages the NIC hardware timestamps and excludes ACK turnaround delay from the RTT (similar to Swift [58]). For ② and ③, theoretically, these delays impact how fast the sender can react to network condition changes, thus impacting the decision precision; however, in practice, even hardware-based transport handles CC events in a per-RTT granularity (e.g., tens of microseconds) rather than per-ACK to avoid overreaction [64]. For example, Google Falcon hardware transport runs CC to update rate once per RTT [39]. Thus, the few microseconds of decision delay introduced by the software is negligible. Nevertheless, UCCL-Tran still employs several techniques to reduce the two delays: similar to Flor [63], UCCL-Tran uses a dedicated high-priority QP for ACK (using in-network priority such as DSCP) and always first polls its completion queue; UCCL-Tran further allocates the ACK polling a higher processing budget during DRR scheduling. We quantify these delays in §6.5.3.

3.5 NIC Capabilities

UCCL-Tran supports a wide range of RDMA NICs and non-RDMA NICs. For RDMA NICs, UCCL-Tran relies on their segmentation and reassembly offloading capabilities to achieve high CPU efficiency. This capability is widely supported through the RC and UC QPs in commodity NICs. For non-RDMA NICs, UCCL-Tran relies on fast user-space packet IO to efficiently drive the underlying NICs through UDP. This capability is also usually enabled in commodity NICs via DPDK or AF_XDP. For both NICs, UCCL-Tran further relies on the ECMP feature of the underlying network switches to achieve multi-pathing: for RDMA, UCCL-Tran uses different RC/UC QPs; for non-RDMA, UCCL-Tran chooses different UDP ports.

UCCL-Tran leverages timestamping to derive the RTT congestion signal used for congestion control and multi-path load balancing. The signal fidelity can be further strengthened with hardware timestamping that is supported by NVIDIA RDMA NICs. UCCL-Tran can also benefit from NIC vendors exposing more congestion signals, such as ECN and packet trimming status.

4 Extensibility Case Studies

UCCL-Tran provides expressive interfaces to implement and extend multipath transport. Due to space limitations, we elaborate on them in Appendix A. Collective library and application developers could also directly extend UCCL-Tran transport code, e.g., with the new loss recovery scheme in MLT [102], and deploy it quickly in a normal user-space process. We now demonstrate how UCCL-Tran extensibility enables new transport designs that work best for different ML workloads.

4.1 Multipath Transport with Packet Spraying

Recent transport research [16, 59] and UEC advocate packet spraying with hundreds of paths as an effective way to address flow collisions in ML workloads. It is naturally challenging for hardware NICs to implement packet spraying because of excessive per-path states, such as path RTTs. Instead, UCCL-Tran can easily support packet spraying by maintaining per-path RTT in software. UCCL-Tran's software transport uses Power-of-Two sampling [74] to select a path with the lowest RTT, then runs CC to decide how many packets and what rate to transmit. UCCL-Tran implements two CC algorithms: one is CUBIC [42] used in Linux kernel TCP as the default CC, and another is Swift [58], an RTT-based CC used by Google. UCCL-Tran supports both per-path CC states (e.g., per-path `cwnd`) and global CC states controlling all paths; both achieve similar collective performance in our testbeds. UCCL-Tran by default uses global CC during evaluation.

4.2 Receiver-Driven CC

Receiver-driven transports, such as EQDS [85], NDP [43], and Homa [76], proactively control packet sending rate at the receiver by allocating credits to senders. These transports have proven to resolve the last-hop congestion for network incast effectively, which could happen in MoE serving (§2.2). However, to the best of our knowledge, there are no off-the-shelf NICs that support receiver-driven transports. One reason is that they are vastly different from popular sender-driven ones, and implementing them would require NIC hardware modification. Instead, UCCL-Tran's extensibility enables developers to quickly implement and tune receiver-driven transports in software. We choose to implement EQDS [85] in UCCL-Tran as an example, the state-of-the-art receiver-driven transport adopted by UEC [22]. Our EQDS implementation closely follows the EQDS paper [85] with a dedicated pacer thread per NIC on the receiver to issue credit packets for senders. For more details, we refer to Appendix B.

4.3 Efficient Loss Recovery

UCCL-Tran allows customizing the transport loss recovery logic to support more advanced mechanisms other than the go-back-N retransmission baked into many RDMA NICs. Go-back-N directly drops out-of-order packets to avoid buffering them on expensive on-chip SRAM, but this gives poor performance when packet loss happens [53, 104, 111]. Instead, we implement a more efficient selective retransmission [70] in UCCL-Tran, by maintaining reordering buffers in the GPU memory (§3.2). Our implementation follows the standard selective retransmission mechanism in TCP, and uses a `std::map` to track an arbitrary number of out-of-order packets. With more efficient loss recovery in UCCL-Tran, ML workloads could possibly run in lossy datacenter networks without PFC (§2.2).

We note that the latest generation of NVIDIA RDMA NICs [79] has implemented a limited form of selective retransmission based on NACK with a small fixed tracking window (to keep low SRAM usage). UCCL-Tran's selective retransmission based on SACK+bitmap is better than these NACK-based one, which has been proven to lead to slow, imprecise recovery and high tail latency [96].

5 Implementation

We implement UCCL-Tran in 28.4K lines of C++ as a network plugin to NCCL/RCCL, the standard collective library for NVIDIA/AMD GPUs. Our current implementation supports NVIDIA RDMA NICs via RC and UC, Broadcom RDMA NICs via RC, AWS EFA NICs via UD/SRD, and AWS ENA non-RDMA NICs via AF_XDP. UCCL-Tran leverages the standard `libibverbs` for RDMA NICs and kernel built-in AF_XDP for non-RDMA NICs; therefore, it should naturally support other vendors' NICs. To enable scattered memcpy when using UD for AWS EFA NICs, we add or modify ∼170 LoC to the NCCL codebase, including a scattered memcpy GPU kernel and C++ code that passes packet pointers to the kernel via CPU-GPU shared memory (created by `cudaHostAlloc()`). We added two new interfaces to the NCCL network plugin system: `irecv_scattered` that receives data in a list of scattered packets, and `irecv_free_ptrs` that frees the scattered packet buffers after scattered memcpy. UCCL-Tran also supports running NCCL with multiple processes.

There are some implementation details worthy of note. For NVIDIA NICs, to calculate network RTT based on NIC hardware TX/RX timestamps and CPU ACK timestamps, we implement a NIC-CPU clock synchronization scheme following Swift [58]. We use a synchronization time interval of 100μs. For AWS EFA NICs that do not support hardware timestamps yet, we calculate RTT based on software timestamps minus estimated packet queueing delay at the host. We estimate the queueing delay by dividing the size of queued packets by the NIC bandwidth. For AWS EFA NICs, we find that allocating UD QPs with consecutive QP numbers for a connection yields much better performance than not. We suspect this is because the EFA NICs do round-robin assignments when mapping a QP (and its associated packets) to a ARM core; consecutive allocations help map the QPs of a connection to different ARM cores, avoiding load imbalance.

6 Evaluation

In this section, we aim to answer the following questions:

- What is the collective performance of UCCL-Tran software multipath transport compared to hardware-based ones (§6.1)?
- Can UCCL-Tran improve ML application performance (§6.2)?
- Can UCCL-Tran extensibility benefit certain workloads (§6.3)?
- What is the scalability of UCCL-Tran (§6.4)?

Name	# of Servers	Network	Topology	MTU	NIC	GPU	CPU
CX_ETH	6	Ethernet	Cross racks, fat-tree	4KB	NVIDIA ConnectX-7 400G ×8	NVIDIA H100-80G ×8	160 cores
AMD	4	Ethernet	Cross racks, rail-optimized	4KB	Broadcom Thor-2 400G ×8	AMD MI300X-192G ×8	128 cores
EFA	4	Ethernet	Cross racks, fat-tree	9KB	AWS EFA 100G ×4	NVIDIA A100-40G ×8	96 cores
CX_IB	2	InfiniBand	Same rack	4KB	NVIDIA ConnectX-7 400G ×8	NVIDIA H100-80G ×8	128 cores

Table 1: Evaluation testbeds. CX_ETH, AMD, and EFA are rented from TensorWave, IBM Cloud, and AWS (`p4d.24xlarge`), respectively.

• How do different designs impact UCCL-Tran (§6.5)?

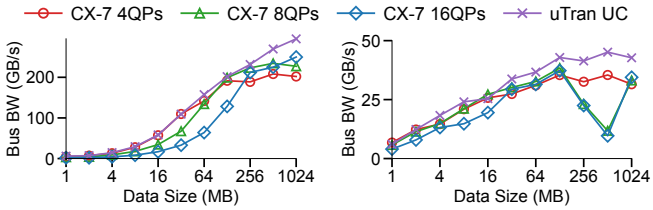
Testbed. Table 1 describes our evaluation testbeds. The first three testbeds across racks evaluate how UCCL-Tran handles network congestion and flow collisions. They also demonstrate the genericity of UCCL-Tran, i.e., applying to AMD GPUs and Broadcom/EFA NICs. The CX_IB testbed under the same rack stress-tests the UCCL-Tran software implementation, as it has no network congestion or flow collisions. For some tests, we try to simulate a larger testbed by disabling NCCL NVLink and SHM communication. This forces GPUs within a server to communicate through the network, and each GPU essentially behaves like a virtual server, e.g., CX_IB becomes 16 virtual GPU servers each with 400G network bandwidth.

Experiment setup. For collective performance evaluation, we use NCCL v2.23.4 [26] and NCCL-tests `9d26b84` [25], the latest versions when we start integrating UCCL-Tran into NCCL, and focus on the representative allreduce and all-to-all collectives. NCCL-tests vary the collective message size and measure the achieved bus bandwidth. We focus on message sizes from 1MB to 1GB, commonly used in real ML workloads [87, 103, 105]. For AMD GPUs, we use RCCL `532f54c` [48] and RCCL-test `5b27b96` [47].

Comparison baselines. On CX_ETH and CX_IB, we compare UCCL-Tran to the NCCL built-in RDMA support that uses RC on ConnectX-7/CX-7. UCCL-Tran uses 256 UC/RC QPs for each NIC pair, while the NCCL built-in uses QP scaling with 4 RC QPs per connection [34, 59]. UCCL-Tran uses CUBIC CC on CX_IB unless specified, which performs better than Swift in the lossless InfiniBand. Since InfiniBand has no packet loss for CUBIC to reduce `cwnd`, to avoid severe network congestion, we enforce a maximum `cwnd` value in UCCL-Tran CUBIC CC to limit the maximum inflight bytes, similar to TCP flow control. Similar setups apply to the AMD testbed.

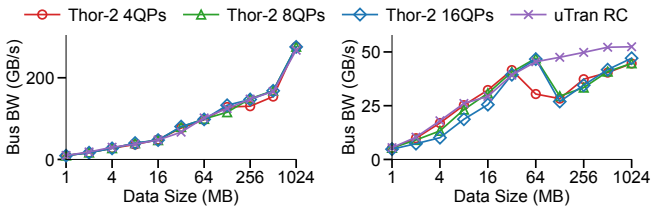
On EFA, we compare UCCL-Tran to the official AWS NCCL-EFA plugin v1.13.2 [3] that uses the multipath SRD [93]. UCCL-Tran uses 10×26=260 UD QPs to serve all connections, which yields the best performance empirically, while the NCCL-EFA plugin uses the best parameters recommended by AWS. UCCL-Tran uses CUBIC CC on EFA by default, as EFA NICs do not support hardware timestamping that is critical to Swift CC. In all testbeds, UCCL-Tran uses per-path RTT for multipath LB (§4.1).

As the cost of extensible software transport, NCCL atop UCCL-Tran uses more CPU cores than vanilla NCCL. By default, the vanilla NCCL uses 2 cores per GPU, while UCCL-Tran uses 2 more cores per NIC to run the engine threads, with 1 additional core per NIC for receiver-driven CC due to the pacer.



(a) Allreduce. **(b)** All-to-All.

Figure 7: NCCL-tests results on CX_ETH.



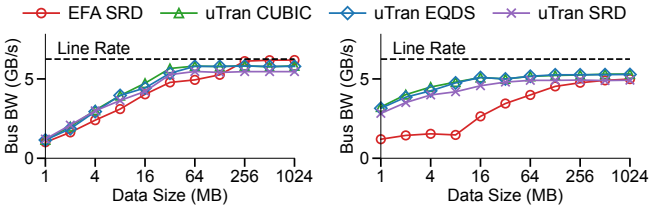
(a) Allreduce. **(b)** All-to-All.

Figure 8: RCCL-tests results on AMD.

6.1 Collective Performance

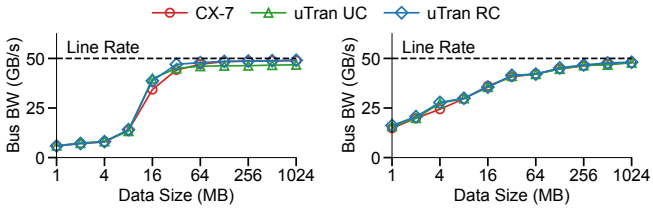
On CX_ETH. Figure 7 compares the performance of UCCL-Tran vs. ConnextX-7 on the CX_ETH testbed. For NCCL over CX-7, we use multiple QPs for each connection between two GPUs by adjusting `NCCL_IB_QPS_PER_CONNECTION`. However, it still suffers from significant performance drops under larger message sizes and lower peak performance. This is because flow collisions in the fat-tree topology cause severe network congestion, which in turn leads to exponential backoffs in the CX-7 CC mechanism. Instead, UCCL-Tran is able to scale up performance stably with message size increasing, by making smarter LB and CC decisions in software. Overall, UCCL-Tran outperforms CX-7 by up to 2.32/1.60/1.24× (corresponding to 4/8/16 QPs) and 1.79/3.82/4.54× for allreduce and all-to-all, respectively.

On AMD. Figure 8 compares the performance on the AMD testbed. This testbed has a rail-optimized topology that helps reduce network congestion when used together with the PXN technique [78]. Under this topology, UCCL-Tran achieves comparable performance as Thor-2 for allreduce, and outperforms Thor-2 by up to 1.68/1.61/1.78× (corresponding



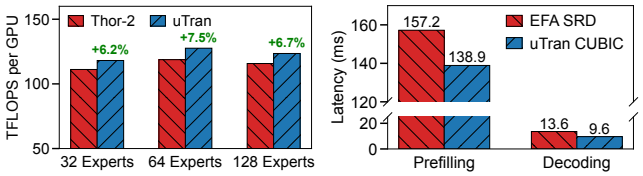
(a) Allreduce. **(b)** All-to-All.

Figure 9: NCCL-tests results on EFA (NVLink+SHM disabled to simulate a larger testbed).



(a) Allreduce. **(b)** All-to-All.

Figure 10: NCCL-tests results on CX_IB (NVLink+SHM disabled).



(a) DeepSeek-V2-Lite [65] training on AMD. **(b)** DeepSeek-V3 [66] serving on EFA (trace-driven emulation).

Figure 11: Application performance evaluation.

to 4/8/16 QPs) for all-to-all, by better controlling network congestion.

On EFA. Figure 9 compares the collective performance of UCCL-Tran and SRD on the EFA testbed. UCCL-Tran CUBIC and EQDS achieve similar performance with significant improvement over SRD (except for ≥256MB allreduce): for allreduce, UCCL-Tran outperforms SRD by up to 1.27×; for all-to-all, the speedup is up to 3.27×. UCCL-Tran outperforms SRD because beefy CPU cores are faster in making transport decisions than wimpy ARM cores on `p4d.24xlarge` EFA NICs (that run SRD), especially when handling connection-intensive all-to-all. Surprisingly, simple CUBIC CC performs very well. This is mainly because UCCL-Tran leverages hundreds of network paths and avoids congested ones before sending packets; so most of the time, CC does not get involved. The small performance gap after 256MB is caused by the extra control header added by UCCL-Tran, while SRD directly reuses the UDP header for control (UCCL-Tran cannot reuse as it is not exposed to CPUs).

UCCL-Tran could also directly leverage the SRD protocol of EFA NICs to send/recv individual datagram packets. UCCL-Tran over SRD does connection management and multipath load balancing in CPU, and packet reassembly in GPU (§3.1), overcoming the limited processing power and cache capacity of SmartNICs on `p4d.24xlarge`. As a result, it achieves similar high performance to UCCL-Tran CUBIC and EQDS.

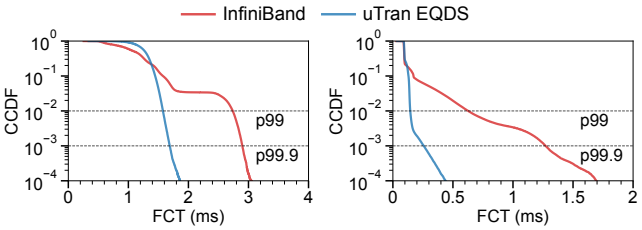
On CX_IB. Figure 10 compares the allreduce and all-to-all performance of UCCL-Tran vs. ConnextX-7 on the CX_IB testbed. UCCL-Tran UC/RC performs almost the same as ConnectX-7 with various data sizes, except that UCCL-Tran UC performs slightly worse (i.e., less than 4%) than ConnextX-7 and UCCL-Tran RC for allreduce after 128MB. For allreduce, this exception is because of the extra overhead of handling packet reliability in UC; for all-to-all, ConnextX-7 and UCCL-Tran RC do not perform better than UCCL-Tran UC, because all-to-all is connection-intensive, causing more QP swapping for RC. These experiments confirm that UCCL-Tran can make highly efficient control decisions in software, reaching the performance of ASIC-based RDMA NICs (i.e., NVIDIA ConnectX-7) for ML collectives.

Due to space limitations, we show the results of more collectives in Appendix C.1, on EFA with NVLink+SHM enabled in Appendix C.2 (up to 2.14× faster), and on non-RDMA NICs in Appendix C.3 (up to 4.1× faster).

6.2 Application Performance

UCCL-Tran is motivated by the practical needs of both LLM training and serving workloads. We run two applications to evaluate how UCCL-Tran improves ML workload performance. One is an end-to-end training of the DeepSeek-V2-Lite model [65] with 16B parameters and 64 routed experts on the AMD testbed using expert parallelism. We use the AMD Primus [8] training framework and its Megatron-LM backend. We also scale the model up and down to 128 and 32 routed experts to evaluate more training scenarios. Another is a DeepSeek-V3-like MoE serving application. Due to the testbed constraint (e.g., hundreds of GPUs required for meaningful DeepSeek-V3 serving with expert parallelism [66, 99]), we use a realistic DeepSeek-V3 trace including GPU compute time and network message sizes (i.e., hidden state sizes) from [106], and emulate the computation and communication behaviors with PyTorch and NCCL on EFA (NVLink+SHM disabled).

Figure 11a shows the results for DeepSeek-V2-Lite training. UCCL-Tran improves the end-to-end training speedup of up to 7.5%, without any line of code change to the training framework. Figure 11b shows the results for DeepSeek-V3 serving. UCCL-Tran reduces the per-request prefilling and decoding latency by 1.13× and 1.42× respectively. These experiments demonstrate that UCCL-Tran is able to bring significant benefits to real-world ML applications.



(a) Incast traffic. **(b)** Permutation traffic.

Figure 12: Complementary CDF of FCT (Flow Completion Time) on CX_IB when co-locating 15-to-1 incast and permutation traffic.

6.3 UCCL-Tran Extensibility

6.3.1 Handling network incast

In a shared RDMA network, network incast can lead to victim flows due to PFC propagation [45, 73, 104]. Such network incast can be caused by the Expert Parallelism in MoE serving (§2.2) and gather/reduce collectives in multi-level parallelism of LLM training [13, 40]. This experiment tries to create a scenario where network incast co-exists with other collective traffic, and evaluates how UCCL-Tran performs compared to the RDMA hardware transport on CX_IB. We co-locate 15-to-1 incast traffic and 16-NIC permutation traffic [16, 19, 59, 89], where each NIC streams data to another and no NIC receives more than one stream, representing typical collectives. For both types of traffic, each NIC sends 1MB messages with at most four in flight. We compare the receiver-driven EQDS in UCCL-Tran vs. the sender-driven InfiniBand CC in ConnectX-7.

Figure 12a and 12b show the FCT distributions for incast traffic and permutation traffic, respectively. Compared to InfiniBand, UCCL-Tran EQDS reduces P99/P99.9 latency by 1.73×/1.72× for incast traffic and 4.50×/4.88× for permutation traffic. This is because the InfiniBand CC only reduces the sending rate when severe congestion and queue build-up have occurred on the incast switch port; but this is too late, as the Credit-Based Flow Control [49] (PFC equivalent in InfiniBand) has already paused all upstream ports and heavily disturbed victim flows (i.e., permutation traffic). Conversely, UCCL-Tran EQDS proactively controls the rate of all senders on the receiver side, which reduces queue build-up and avoids upstream ports entering the pausing state.

6.3.2 Handling packet loss

We then evaluate how UCCL-Tran selective retransmission handles packet loss, compared to hardware-based loss recovery. Furthermore, UCCL-Tran coalesces transport decisions like loss recovery at chunk granularity for high software efficiency; therefore, we also test whether this design choice would cause poor loss recovery performance. To this end, we instrument different packet loss rates in software when running NCCL collectives over UCCL-Tran UC. Our packet loss instrumentation has considered the chunking nature in UCCL-Tran, where any packet loss will cause the whole chunk loss. Unfortunately, we cannot instrument packet loss for our CX_IB NICs as that would require reconfiguring the NIC or switch behaviors, both of which we do not have access to. Instead, we cite comparable numbers from prior literature Flor [63].

We pick two GPUs from the AMD testbed, each using one NIC for communication. The client NIC establishes a connection with the server NIC and keeps 16 inflight messages. Similar to Flor [63], we use a single QP for the connection, and disable congestion control to avoid sending rate backoff. We vary the message size (32KB∼1MB) and measure the goodput under different packet drop ratios. Figure 13 shows the results. UCCL-Tran has only ∼1% performance drop under 1/16384, 1/4096 drop ratio, compared to the reported 26%∼42% drop of RDMA hardware transport in Flor [63, Figure 7]. Even under a high drop ratio of 1/1024 and 1/256, UCCL-Tran only experiences a performance drop of 6%∼30%, compared to the reported 59%∼76% drop in RDMA hardware transport.

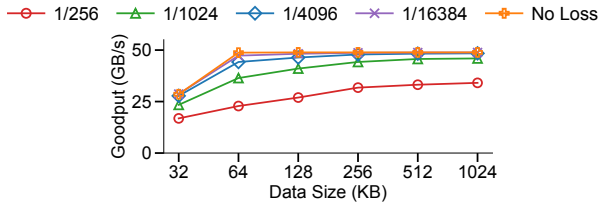


Figure 13: Goodput of UCCL-Tran under different packet drop ratios. UCCL-Tran implements selective retransmission with a dynamic RTO threshold (around several RTTs).

	Training w/ NCCL	w/ NCCL + uTran
# CPU cores / # 400G NICs	2.3	4.3

Table 2: CPU core consumption relative to the number of active 400G NICs during DeepSeek-V2-Lite training (§6.2).

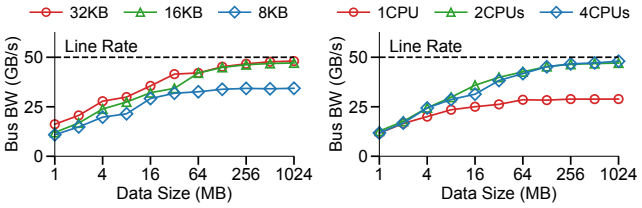
6.4 UCCL-Tran Scalability

6.4.1 CPU core consumption

In addition to the training-framework CPU usage, which is averagely 14.5% out of 128 cores in practice (§1, UCCL-Tran additionally uses 16 CPU cores (2 per GPU). Table 2 summarizes the CPU core consumption relative to the number of active GPUs. As the GPU price and CPU core count keep increasing (e.g., 192 CPU cores in AWS Hopper GPU VMs [5]), we believe trading 16 additional CPU cores for a 7.5% GPU efficiency boost is becoming increasingly worthwhile.

6.4.2 Varying chunk size

Figure 14a shows the all-to-all performance on the CX_IB testbed when varying chunk sizes. Our results reveal that saturating the line rate requires a somewhat medium chunk



(a) Varying chunk size. **(b)** Varying CPU cores.

Figure 14: UCCL-Tran all-to-all on CX_IB with different chunk sizes and number of CPU cores per NIC (NVLink+SHM disabled).

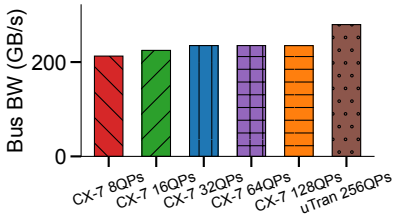


Figure 15: The highest allreduce bandwidth with different number of QPs and transports on CX_ETH. When preparing this figure during camera ready, we are no longer able to access any testbed due to the long period between the original evaluation and camera-ready, and the high expenses of RDMA-connected GPU testbeds. We are only able to retrieve the CX-7 results with up to 128 QPs from our previous logs, but we believe the trends do not change when CX-7 scales to 256 QPs (due to RC's overhead in RDMA NICs).

size, i.e., ≥16KB or 4 MTU-sized chunks. This experiment demonstrates the performance benefits of control coalescing for UCCL-Tran software transport.

6.4.3 Varying number of CPU cores

Figure 14b shows the all-to-all performance when varying the number of UCCL-Tran engines/CPU cores per NIC. For ASIC-based NICs with segmentation and reassembly offloading, even though one CPU core can only saturate bidirectional 29GB/s, two CPU cores are enough to saturate the full 50GB/s line rate. The slight performance drop for 16MB when switching from "2CPUs" to "4CPUs" should be caused by runtime variability. This experiment confirms the high efficiency of UCCL-Tran software transport, i.e., 1 CPU core saturating 400G unidirectional traffic (2 CPU cores for 400G bidirectional).

We show more scalability studies in Appendix C.4 when varying numbers of CPU cores, GPUs, and paths on EFA.

6.5 Design Drill-Down

6.5.1 Performance improvement attribution

Figure 15 shows the highest allreduce bandwidth when varying the number of QPs in different transports. There are two takeaways: 1) the performance of the raw CX-7 transport saturates at 32 QPs, and increasing QPs beyond even incurs a slight performance penalty (due to RC's high overhead), 2)

Max Bus BW (GB/s)	Allreduce	All-to-All
w/o conn split (ZeroNIC [97], Flor [63])	45.7	39.9
w/ conn split (UCCL-Tran)	48.9	48.5

Table 3: Impact of connection splitting on CX_IB.

Delay (μs)	Allreduce Light	All-to-All Light	Allreduce Heavy	All-to-All Heavy
CC decision P50	1.7	1.7	2.1	2.9
CC decision P99	3.6	3.4	7.0	10.8
ACK turnaround P50	2.0	3.0	4.0	7.0
ACK turnaround P99	3.0	10.0	10.0	36.0

Table 4: CC decision delay and ACK turnaround delay on CX_IB. "Light" uses 1KB-64KB messages, while "Heavy" uses 1GB.

CX-7 with 128 QPs still falls behind UCCL-Tran performance by a large margin. Therefore, we think simply scaling QPs to 256 is not the leading factor of performance improvement, but the dynamic path selection (i.e., power-of-two sampling based on per-path RTT) is the key to better performance. We note that such advanced dynamic path selection can hardly be done on traditional NICs due to the large per-path states they need to maintain.

6.5.2 Connection splitting

We now evaluate how connection splitting impacts collective performance in UCCL-Tran. Without connection splitting, each UCCL-Tran connection is mapped to a single engine out of the two UCCL-Tran engines per NIC; with splitting, each UCCL-Tran connection dispatches and load balances its messages among the two UCCL-Tran engines. Table 3 shows that UCCL-Tran without connection splitting (e.g., ZeroNIC [97], Flor [63]) only achieves 45.7 and 39.9 GB/s max busbw for allreduce and all-to-all, while UCCL-Tran with splitting achieves 48.9 and 48.5 GB/s respectively, saturating the line rate. This is because connection splitting enables multiple CPU cores to process inflight messages in a load-balanced manner. We anticipate connecting splitting will become more critical as the NIC scales to 800 Gbps [27].

6.5.3 Congestion signal fidelity in software

UCCL-Tran makes a fundamental shift of moving transport from hardware to software. This experiment evaluates how much software transport sacrifices in terms of congestion signal fidelity. We look at the two important metrics as mentioned in §3.4: CC decision delay and ACK turnaround delay. Table 4 shows the results. At light loads, the two delays are limited to at most 10μs in both P50 and P99; at high loads, they grow up to 36μs for P99. Both are on par with typical datacenter RTTs of 10-40 μs [37]. Therefore, this experiment confirms that UCCL-Tran software transport has high enough CC fidelity to make precise per-RTT transport decisions [39,64]. In

addition, we emphasize that while software RTT introduces some noise, the bulk data nature of ML traffic makes it more tolerant to this than latency-sensitive RPC workloads.

We show more design drill-downs in Appendix C.5, including the impact of chunk size, LB policy, kernel fusion (for scattered memcpy), and PCIe overhead analyses.

7 Discussion

HW-SW interface. UCCL-Tran works around many challenges to support extensible transport with efficient multipathing on existing RDMA NICs, at the cost of QP swapping overhead (though not high for ML collectives), control coalescing, and more. UCCL-Tran performance and control granularity would get further boosted if there were a better HW-SW interface in RDMA NICs.

We would like to highlight a few points based on our development experience: (1) UC abstraction is powerful with segmentation and reassembly offloading, but using many UC QPs for multipathing incurs some degree of QP swapping overhead; instead, it should evolve into a multipath UC abstraction that allows the software to specify different flow entropies for different verbs over a single QP, just like how UCCL-Tran specifies different UDP ports for AF_XDP. (2) NIC HW should expose more congestion signals to SW, such as ECN marks and packet trimming information; these signals can be embedded into CQEs just like hardware timestamps.

Software jitters. UCCL-Tran runs network transport on the host CPU, and may suffer from software jitters. We argue that such jitters could be avoided or reduced to a minimal level for ML workloads. For example, storage traffic nowadays could directly go from GPU to storage devices via GPUDirect Storaage, so no intensive CPU involvement. ML cluster scheduler already schedules dedicated cores for NCCL CPU proxy, which works well. We just need to reserves extra cores for UCCL-Tran. Futhermore, we can disable interrupts for certain cores that run NCCL CPU proxy and UCCL-Tran transport.

GPU-initiated communication like DeepEP [29] leverages NVIDIA IBGDA [84] to issue RDMA verbs directly from GPUs to the RDMA NICs. Although seemingly conflicted with UCCL-Tran's CPU-initiated designs, GPU-initiated communication could still be made compatible with UCCL-Tran. The key is to leverage the IBGDA CPU-assisted mode [80], where the GPU forwards RDMA requests to a CPU proxy that issues RDMA verbs; with this, we could implement UCCL-Tran's extensible transport layer inside the CPU proxy. The trade-off is on performance: NVIDIA has reported that this CPU-assisted IBGDA would sacrifice 10% performance compared to the traditional IBGDA [80]. We intend to leave the integration and performance enhancement of CPU-assisted IBGDA + UCCL-Tran as future work.

Observability. During the private conversation with GPU cluster managers, we found that network observability is critical. For example, once observing which path has congestion and when retransmission happens, the cluster manager can drain traffic away from the hotspot to improve the completion time of synchronous ML training. UCCL-Tran's software approach provides a better observability vantage point compared to the hardware-baked RDMA transport. UCCL-Tran users could directly instrument the transport layer and quickly recompile in software to, e.g., log per-path congestion level and retransmission events.

8 Other Related Work

Recent work SCR [107] implements receiver-driven CC and multipathing on the DPA (DataPath Accelerator, essentially 16 RISC-V cores) of NVIDIA BlueField-3 SmartNICs. It would have similar performance issues as AWS EFA NICs. Moreover, DPA programmability is limited to what the NIC hardware supports, e.g., only rate-based control, while packet reliability and retransmission are still baked into the hardware. Because of this, SCR needs to alter the original receiver-driven CC to let credits represent available bandwidth rather than bytes. For multipathing, SCR only demonstrates two paths; given the limited L1/L2 cache in DPA [107], it is not clear if SCR could scale to hundreds of paths [16, 59, 85]. The recent MRC [10] protocol builds a multipath RC protocol into programmable NICs such as NVIDIA ConnectX-8 and Broadcom Thor Ultra. We note it is released one year after UCCL-Tran and might suffer from similar issues as SCR.

Google GPUDirect-TCPX [2] integrates GPUDirect into the kernel TCP stack by leveraging the Header-Data Split feature in certain non-RDMA NICs. Instead, UCCL-Tran targets both RDMA and non-RDMA NICs, and further supports efficient multipathing. C4 [32] does coarse-grained flow-level traffic planning and path selection in LLM training, without programmability for low-level RDMA transport components such as CC and loss recovery. MSCCL [11] supports customizing collective communication algorithms and could work together with UCCL-Tran. Lastly, UCCL-Tran is inspired by a line of work that targets extensibility for CPU applications, e.g., Google 1RMA [95], eRPC [53], RoGUE [60], and IO-TCP [55] for extensible RDMA transport, and SPIN [15], Exokernel [33], and VINO [91] for extensible OS.

9 Conclusion

UCCL-Tran is an extensible and efficient software transport layer for GPU networking. It achieves network extensibility by separating the control path and data path for existing RDMA NICs and running the transport control path in software. Meanwhile, it achieves hardware-level performance by leveraging techniques like control coalescing and connection splitting. We hope UCCL-Tran opens the door to the productization of new research proposals on network transports for ML workloads. UCCL-Tran is open-sourced at https://github.com/uccl-project/uccl.

Acknowledgement

We sincerely thank our shepherd Nitin Agrawal and the anonymous reviewers for their insightful comments. This work is in part supported by gifts from Accenture, AMD, Anyscale, Broadcom, Cisco, Google, IBM, Intel, Intesa Sanpaolo, Lambda, Lightspeed, Mibura, Microsoft, NVIDIA, Samsung SDS, and SAP. Costin Raiciu was partly funded by HRIA (project no. 351416). We additionally acknowledge AWS, in particular Zhenyu Song, Jun Wu, and Yida Wang, for their sponsorship and partnership in this research.

References

[1] Mohammad Alizadeh, Albert Greenberg, David A. Maltz, Jitendra Padhye, Parveen Patel, Balaji Prabhakar, Sudipta Sengupta, and Murari Sridharan. Data Center TCP (DCTCP). In *Proceedings of ACM SIGCOMM*, page 63–74, 2010.

[2] Mina Almasry and Willem de Bruijn, Eric Dumazet, Kaiyuan Zhang. Device memory TCP: Transferring data from/to device memory efficiently. Netdev 0x17 Conference, 2023.

[3] Amazon Web Services. AWS OFI NCCL. https://github.com/aws/aws-ofi-nccl/tree/v1.13.2-aws, 2024.

[4] Amazon Web Services. Amazon EC2 P5 Instances. https://aws.amazon.com/ec2/instance-types/p5/, 2025.

[5] Inc. or its affiliates Amazon Web Services. Amazon EC2 P5 instances. https://aws.amazon.com/ec2/instance-types/p5/, 2026.

[6] AMD. RCCL Net Plugin Documentation. https://github.com/ROCm/rccl/tree/develop/ext-net, 2025.

[7] AMD. ROCm Communication Collectives Library (RCCL). https://github.com/ROCm/rccl, 2025.

[8] AMD-AGI/Primus. AMD Primus training framework for large-scale foundation model training and inference on AMD GPUs. https://github.com/AMD-AGI/Primus.

[9] AMD Pensando. AMD Pollara 400 Card. https://www.amd.com/content/dam/amd/en/documents/pensando-technical-docs/product-briefs/pensando-pollara-400-product-brief.pdf, 2024.

[10] Joao Araujo, Alex Chow, Mark Handley, Ryder Lewis, Christoph Paasch, Jitendra Padhye, Michael Papamichael, Greg Steinbrecher, Amin Tootoonchian, Lihua Yuan, et al. Resilient AI Supercomputer Networking using MRC and SRv6. *arXiv preprint arXiv:2605.04333*, 2026.

[11] Azure/msccl. Microsoft Collective Communication Library (MSCCL). https://github.com/Azure/msccl, 2024.

[12] Wei Bai, Shanim Sainul Abdeen, Ankit Agrawal, Krishan Kumar Attre, Paramvir Bahl, Ameya Bhagat, Gowri Bhaskara, Tanya Brokhman, Lei Cao, Ahmad Cheema, et al. Empowering Azure Storage with RDMA. In *Proceedings of USENIX NSDI*, pages 49–67, 2023.

[13] Paul Barham, Aakanksha Chowdhery, Jeff Dean, Sanjay Ghemawat, Steven Hand, Daniel Hurt, Michael Isard, Hyeontaek Lim, Ruoming Pang, Sudip Roy, et al. Pathways: Asynchronous Distributed Dataflow for ML. *Proceedings of Machine Learning and Systems*, 4:430–449, 2022.

[14] Adam Belay, George Prekas, Ana Klimovic, Samuel Grossman, Christos Kozyrakis, and Edouard Bugnion. IX: A Protected Dataplane Operating System for High Throughput and Low Latency. In *Proceedings of USENIX OSDI*, pages 49–65, 2014.

[15] Brian N Bershad, Stefan Savage, Przemyslaw Pardyak, Emin Gün Sirer, Marc E Fiuczynski, David Becker, Craig Chambers, and Susan Eggers. Extensibility Safety and Performance in the SPIN Operating System. In *Proceedings of ACM SOSP*, pages 267–283, 1995.

[16] Tommaso Bonato, Abdul Kabbani, Daniele De Sensi, Rong Pan, Yanfang Le, Costin Raiciu, Mark Handley, Timo Schneider, Nils Blach, Ahmad Ghalayini, et al. SMaRTT-REPS: Sender-based Marked Rapidly-adapting Trimmed & Timed Transport with Recycled Entropies. *arXiv e-prints*, pages arXiv–2404, 2024.

[17] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, et al. P4: Programming Protocol-Independent Packet Processors. *ACM SIGCOMM Computer Communication Review*, 44(3):87–95, 2014.

[18] Broadcom. Broadcom High-Performance 400G RoCE / RDMA NICs. https://www.broadcom.com/info/nic/performance-ethernet-adapters, 2024.

[19] Jiaxin Cao, Rui Xia, Pengkun Yang, Chuanxiong Guo, Guohan Lu, Lihua Yuan, Yixin Zheng, Haitao Wu, Yongqiang Xiong, and Dave Maltz. Per-Packet Load-Balanced, Low-Latency Routing for Clos-Based Data

Center Networks. In *Proceedings of the ACM CoNEXT*, pages 49–60, 2013.

[20] Youmin Chen, Youyou Lu, and Jiwu Shu. Scalable RDMA RPC on Reliable Connection with Efficient Resource Sharing. In *Proceedings of EuroSys*, pages 1–14, 2019.

[21] Peng Cheng, Fengyuan Ren, Ran Shu, and Chuang Lin. Catch the whole lot in an action: Rapid precise packet loss notification in data center. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, pages 17–28, 2014.

[22] Ultra Ethernet Consortium. The New Era Needs a New Network. https://ultraethernet.org/, 2023.

[23] NVIDIA Corporation. DGX H100/200 System Topology. https://docs.nvidia.com/dgx/dgxh100-user-guide/introduction-to-dgxh100.html#dgx-h100-200-system-topology, 2025.

[24] NVIDIA Corporation. NCCL Net Plugin Documentation. https://github.com/NVIDIA/nccl/blob/master/ext-net/README.md, 2025.

[25] NVIDIA Corporation. NCCL Tests. https://github.com/NVIDIA/nccl-tests/tree/9d26b8422ba76c098df996b96e13b8ddf3a71165, 2025.

[26] NVIDIA Corporation. NCCL v2.23.4-1. https://github.com/NVIDIA/nccl/tree/2ea4ee94bfb04c886c79ccae60ac9961000fdee2, 2025.

[27] NVIDIA Corporation. NVIDIA Ethernet SuperNICs: Next-generation networking for the next wave of AI. https://www.nvidia.com/en-us/networking/products/ethernet/supernic/, 2025.

[28] NVIDIA Corporation. NVIDIA GPUDirect: Enhancing Data Movement and Access for GPUs. https://developer.nvidia.com/gpudirect, 2025.

[29] DeepSeek. DeepEP: an Efficient Expert-Parallel Communication Library. https://github.com/deepseek-ai/DeepEP, 2025.

[30] DeepSeek AI. Expert Parallelism Load Balancer (EPLB). https://github.com/deepseek-ai/EPLB, 2025.

[31] Advait Dixit, Pawan Prakash, Y Charlie Hu, and Ramana Rao Kompella. On the Impact of Packet Spraying in Data Center Networks. In *Proceedings of IEEE INFOCOM*, pages 2130–2138. IEEE, 2013.

[32] Jianbo Dong, Bin Luo, Jun Zhang, Pengcheng Zhang, Fei Feng, Yikai Zhu, Ang Liu, Zian Chen, Yi Shi, Hairong Jiao, et al. Boosting Large-Scale Parallel Training Efficiency with C4: A Communication-Driven Approach. *arXiv preprint arXiv:2406.04594*, 2024.

[33] Dawson R Engler, M Frans Kaashoek, and James O'Toole Jr. Exokernel: An Operating System Architecture for Application-Level Resource Management. *ACM SIGOPS Operating Systems Review*, 29(5):251–266, 1995.

[34] Adithya Gangidi, Rui Miao, Shengbao Zheng, Sai Jayesh Bondu, Guilherme Goes, Hany Morsy, Rohit Puri, Mohammad Riftadi, Ashmitha Jeevaraj Shetty, Jingyi Yang, Shuqiang Zhang, Mikel Jimenez Fernandez, Shashidhar Gandham, and Hongyi Zeng. RDMA over Ethernet for Distributed Training at Meta Scale. In *Proceedings of ACM SIGCOMM 2024 Conference*, page 57–70, 2024.

[35] Yixiao Gao, Qiang Li, Lingbo Tang, Yongqing Xi, Pengcheng Zhang, Wenwen Peng, Bo Li, Yaohui Wu, Shaozong Liu, Lei Yan, et al. When Cloud Storage Meets RDMA. In *Proceedings of USENIX NSDI*, pages 519–533, 2021.

[36] Talia Gershon, Seetharami Seelam, Brian Belgodere, Milton Bonilla, Lan Hoang, Danny Barnett, I Chung, Apoorve Mohan, Ming-Hung Chen, Lixiang Luo, et al. The Infrastructure Powering IBM's Gen AI Model Development. *arXiv preprint arXiv:2407.05467*, 2024.

[37] Dan Gibson, Hema Hariharan, Eric Lance, Moray McLaren, Behnam Montazeri, Arjun Singh, Stephen Wang, Hassan MG Wassel, Zhehua Wu, Sunghwan Yoo, et al. Aquila: A unified, low-latency fabric for datacenter networks. In *Proceedings of USENIX NSDI*, pages 1249–1266, 2022.

[38] Google. Falcon Transport Protocol. https://github.com/opencomputeproject/OCP-NET-Falcon, 2024.

[39] Google. Introduction to Falcon Reliable Transport. https://netdevconf.info/0x18/sessions/talk/introduction-to-falcon-reliable-transport.html, 2024.

[40] Aaron Grattafiori, Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Alex Vaughan, et al. The Llama 3 Herd of Models. *arXiv preprint arXiv:2407.21783*, 2024.

[41] Chuanxiong Guo, Haitao Wu, Zhong Deng, Gaurav Soni, Jianxi Ye, Jitu Padhye, and Marina Lipshteyn. RDMA over Commodity Ethernet at Scale. In *Proceedings of ACM SIGCOMM*, pages 202–215, 2016.

[42] Sangtae Ha, Injong Rhee, and Lisong Xu. CUBIC: a New TCP-Friendly High-Speed TCP Variant. *ACM SIGOPS operating systems review*, 42(5):64–74, 2008.

[43] Mark Handley, Costin Raiciu, Alexandru Agache, Andrei Voinescu, Andrew W. Moore, Gianni Antichi, and Marcin Wójcik. Re-architecting Datacenter Networks and Stacks for Low Latency and High Performance. In *Proceedings of ACM SIGCOMM*, page 29–42, 2017.

[44] Qinghao Hu, Zhisheng Ye, Zerui Wang, Guoteng Wang, Meng Zhang, Qiaoling Chen, Peng Sun, Dahua Lin, Xiaolin Wang, Yingwei Luo, et al. Characterization of large language model development in the datacenter. In *Proceedings of USENIX NSDI*, pages 709–729, 2024.

[45] Shuihai Hu, Yibo Zhu, Peng Cheng, Chuanxiong Guo, Kun Tan, Jitendra Padhye, and Kai Chen. Deadlocks in Datacenter Networks: Why Do They Form, and How to Avoid Them. In *Proceedings of ACM HotNets*, pages 92–98, 2016.

[46] Cheng Huang, Huseyin Simitci, Yikang Xu, Aaron Ogus, Brad Calder, Parikshit Gopalan, Jin Li, and Sergey Yekhanin. Erasure Coding in Windows Azure Storage. In *Proceedings of USENIX ATC*, pages 15–26, 2012.

[47] Advanced Micro Devices Inc. RCCL Tests. https://github.com/ROCm/rccl-tests/tree/5b27b961b2543b3af2bb1cf5ca8ee0505226ba92, 2025.

[48] Advanced Micro Devices Inc. ROCm Communication Collectives Library (RCCL). https://github.com/ROCm/rccl/tree/532f54c2444501b3655e65fbce6d00d4bfc19c0b, 2025.

[49] InfiniBand Trade Association. InfiniBandTM Architecture Specification Volume 1 Release 1.4, 2020.

[50] Intel. Performance Counter Monitor. https://github.com/intel/pcm/, 2025.

[51] Muhammad Asim Jamshed, YoungGyoun Moon, Donghwi Kim, Dongsu Han, and KyoungSoo Park. mOS: A Reusable Networking Stack for Flow Monitoring Middleboxes. In *Proceedings of USENIX NSDI*, pages 113–129, 2017.

[52] Yimin Jiang, Yibo Zhu, Chang Lan, Bairen Yi, Yong Cui, and Chuanxiong Guo. A unified architecture for accelerating distributed {DNN} training in heterogeneous {GPU/CPU} clusters. In *Proceedings of USENIX OSDI*, pages 463–479, 2020.

[53] Anuj Kalia, Michael Kaminsky, and David Andersen. Datacenter RPCs can be General and Fast. In *Proceedings of USENIX NSDI*, pages 1–16, 2019.

[54] Anuj Kalia, Michael Kaminsky, and David G Andersen. FaSST: Fast, Scalable and Simple Distributed Transactions with Two-Sided (RDMA) Datagram RPCs. In *Proceedings of USENIX OSDI*, pages 185–201, 2016.

[55] Taehyun Kim, Deondre Martin Ng, Junzhi Gong, Youngjin Kwon, Minlan Yu, and KyoungSoo Park. Rearchitecting the TCP Stack for I/O-Offloaded Content Delivery. In *Proceedings of USENIX NSDI*, pages 275–292, 2023.

[56] Xinhao Kong, Jiaqi Lou, Wei Bai, Nam Sung Kim, and Danyang Zhuo. Towards a manageable intra-host network. In *Proceedings of the 19th Workshop on Hot Topics in Operating Systems*, pages 206–213, 2023.

[57] Xinhao Kong, Yibo Zhu, Huaping Zhou, Zhuo Jiang, Jianxi Ye, Chuanxiong Guo, and Danyang Zhuo. Collie: Finding Performance Anomalies in RDMA Subsystems. In *Proceedings of USENIX NSDI*, pages 287–305, 2022.

[58] Gautam Kumar, Nandita Dukkipati, Keon Jang, Hassan MG Wassel, Xian Wu, Behnam Montazeri, Yaogong Wang, Kevin Springborn, Christopher Alfeld, Michael Ryan, et al. Swift: Delay is Simple and Effective for Congestion Control in the Datacenter. In *Proceedings of ACM SIGCOMM*, pages 514–528, 2020.

[59] Yanfang Le, Rong Pan, Peter Newman, Jeremias Blendin, Abdul Kabbani, Vipin Jain, Raghava Sivaramu, and Francis Matus. STrack: A Reliable Multipath Transport for AI/ML Clusters. *arXiv preprint arXiv:2407.15266*, 2024.

[60] Yanfang Le, Brent Stephens, Arjun Singhvi, Aditya Akella, and Michael M Swift. RoGUE: RDMA over Generic Unconverged Ethernet. In *Proceedings of ACM SoCC*, pages 225–236, 2018.

[61] Sugi Lee, Mingyu Choi, Ikjun Yeom, and Younghoon Kim. PeRF: Preemption-Enabled RDMA Framework. In *Proceedings of USENIX ATC*, pages 209–225, 2024.

[62] Mu Li, David G Andersen, Alexander Smola, and Kai Yu. Communication efficient distributed machine learning with the parameter server. *Advances in neural information processing systems*, 27, 2014.

[63] Qiang Li, Yixiao Gao, Xiaoliang Wang, Haonan Qiu, Yanfang Le, Derui Liu, Qiao Xiang, Fei Feng, Peng Zhang, Bo Li, et al. Flor: An Open High Performance RDMA Framework over Heterogeneous RNICs. In *Proceedings of USENIX OSDI*, pages 931–948, 2023.

[64] Yuliang Li, Rui Miao, Hongqiang Harry Liu, Yan Zhuang, Fei Feng, Lingbo Tang, Zheng Cao, Ming Zhang, Frank Kelly, Mohammad Alizadeh, et al.

HPCC: High Precision Congestion Control. In *Proceedings of ACM SIGCOMM*, pages 44–58. 2019.

[65] Aixin Liu, Bei Feng, Bin Wang, Bingxuan Wang, Bo Liu, Chenggang Zhao, Chengqi Dengr, Chong Ruan, Damai Dai, Daya Guo, et al. Deepseek-v2: A strong, economical, and efficient mixture-of-experts language model. *arXiv preprint arXiv:2405.04434*, 2024.

[66] Aixin Liu, Bei Feng, Bing Xue, Bingxuan Wang, Bochao Wu, Chengda Lu, Chenggang Zhao, Chengqi Deng, Chenyu Zhang, Chong Ruan, et al. DeepSeek-V3 Technical Report. *arXiv preprint arXiv:2412.19437*, 2024.

[67] Ming Liu, Tianyi Cui, Henry Schuh, Arvind Krishnamurthy, Simon Peter, and Karan Gupta. Offloading Distributed Applications onto SmartNICs Using iPipe. In *Proceedings of ACM SIGCOMM*, pages 318–333. 2019.

[68] Yuanwei Lu, Guo Chen, Bojie Li, Kun Tan, Yongqiang Xiong, Peng Cheng, Jiansong Zhang, Enhong Chen, and Thomas Moscibroda. Multi-Path Transport for RDMA in Datacenters. In *Proceedings of USENIX NSDI*, pages 357–371, 2018.

[69] Michael Marty, Marc de Kruijf, Jacob Adriaens, Christopher Alfeld, Sean Bauer, Carlo Contavalli, Michael Dalton, Nandita Dukkipati, William C Evans, Steve Gribble, et al. Snap: A Microkernel Approach to Host Networking. In *Proceedings of ACM SOSP*, pages 399–413, 2019.

[70] Matt Mathis, Jamshid Mahdavi, Sally Floyd, and Allyn Romanow. TCP Selective Acknowledgment Options. https://datatracker.ietf.org/doc/html/rfc2018, 1996.

[71] Microsoft Azure. ND-H100-v5 sizes series. https://learn.microsoft.com/en-us/azure/virtual-machines/sizes/gpu-accelerated/ndh100v5-series, 2024.

[72] Radhika Mittal, Vinh The Lam, Nandita Dukkipati, Emily Blem, Hassan Wassel, Monia Ghobadi, Amin Vahdat, Yaogong Wang, David Wetherall, and David Zats. TIMELY: RTT-based Congestion Control for the Datacenter. *ACM SIGCOMM Computer Communication Review*, 45(4):537–550, 2015.

[73] Radhika Mittal, Alexander Shpiner, Aurojit Panda, Eitan Zahavi, Arvind Krishnamurthy, Sylvia Ratnasamy, and Scott Shenker. Revisiting Network Support for RDMA. In *Proceedings of ACM SIGCOMM*, pages 313–326, 2018.

[74] Michael Mitzenmacher. The Power of Two Choices in Randomized Load Balancing. *IEEE Transactions on Parallel and Distributed Systems*, 12(10):1094–1104, 2001.

[75] Sumit Kumar Monga, Sanidhya Kashyap, and Changwoo Min. Birds of a feather flock together: Scaling RDMA RPCs with Flock. In *Proceedings of ACM SOSP*, pages 212–227, 2021.

[76] Behnam Montazeri, Yilong Li, Mohammad Alizadeh, and John Ousterhout. Homa: A Receiver-Driven Low-Latency Transport Protocol Using Network Priorities. In *Proceedings of ACM SIGCOMM*, pages 221–235, 2018.

[77] Multipath TCP community. Multipath TCP. https://www.multipath-tcp.org/, 2023.

[78] NVIDIA Corporation. Doubling all2all Performance with NVIDIA Collective Communication Library 2.12. https://developer.nvidia.com/blog/doubling-all2all-performance-with-nvidia-collective-communication-library-2-12/, 2022.

[79] NVIDIA Corporation. ConnectX-7 400G Adapters. https://resources.nvidia.com/en-us-accelerated-networking-resource-library/connectx-7-datasheet, 2024.

[80] NVIDIA Corporation. CPU-assisted InfiniBand GPU Direct Async. https://developer.nvidia.com/blog/enhancing-application-portability-and-compatibility-across-new-platforms-using-nvidia-magnum-io-nvshmem-3-0/#cpu-assisted_infiniband_gpu_direct_async%C2%A0, 2024.

[81] NVIDIA Corporation. Recommended Topologies for Implementing an HPC Cluster with NVIDIA Quantum InfiniBand Solutions - Part 2: Hash-Based Forwarding. https://enterprise-support.nvidia.com/s/article/Recommended-Topologies-for-Implementing-an-HPC-Cluster-with-NVIDIA-Quantum-InfiniBand-Solutions-Part-2#Hash-BasedForwarding, 2024.

[82] NVIDIA Corporation. Megatron-LM & Megatron-Core. https://github.com/NVIDIA/Megatron-LM, 2025.

[83] NVIDIA Corporation. NVIDIA Collective Communications Library (NCCL). https://github.com/NVIDIA/nccl, 2025.

[84] NVIDIA Corporation. Using the NVSHMEM InfiniBand GPUDirect Async Transport. https://docs.nvidia.com/nvshmem/api/using.html#using-t

he-nvshmem-infiniband-gpudirect-async-transport, 2025.

[85] Vladimir Olteanu, Haggai Eran, Dragos Dumitrescu, Adrian Popa, Cristi Baciu, Mark Silberstein, Georgios Nikolaidis, Mark Handley, and Costin Raiciu. An edge-queued datagram service for all datacenter traffic. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, pages 761–777, 2022.

[86] Phitchaya Mangpo Phothilimthana, Ming Liu, Antoine Kaufmann, Simon Peter, Rastislav Bodik, and Thomas Anderson. Floem: A Programming System for NIC-Accelerated Network Applications. In *Proceedings of USENIX OSDI*, pages 663–679, 2018.

[87] Kun Qian, Yongqing Xi, Jiamin Cao, Jiaqi Gao, Yichi Xu, Yu Guan, Binzhang Fu, Xuemei Shi, Fangbo Zhu, Rui Miao, Chao Wang, Peng Wang, Pengcheng Zhang, Xianlong Zeng, Eddie Ruan, Zhiping Yao, Ennan Zhai, and Dennis Cai. Alibaba HPN: A Data Center Network for Large Language Model Training. In *Proceedings of ACM SIGCOMM*, page 691–706, 2024.

[88] Mubashir Adnan Qureshi, Yuchung Cheng, Qianwen Yin, Qiaobin Fu, Gautam Kumar, Masoud Moshref, Junhua Yan, Van Jacobson, David Wetherall, and Abdul Kabbani. PLB: Congestion Signals Are Simple and Effective for Network Load Balancing. In *Proceedings of ACM SIGCOMM*, pages 207–218, 2022.

[89] Costin Raiciu, Sebastien Barre, Christopher Pluntke, Adam Greenhalgh, Damon Wischik, and Mark Handley. Improving Datacenter Performance and Robustness with Multipath TCP. *ACM SIGCOMM Computer Communication Review*, 41(4):266–277, 2011.

[90] Henry N Schuh, Weihao Liang, Ming Liu, Jacob Nelson, and Arvind Krishnamurthy. Xenic: SmartNIC-Accelerated Distributed Transactions. In *Proceedings of ACM SOSP*, pages 740–755, 2021.

[91] Margo I Seltzer, Yasuhiro Endo, Christopher Small, and Keith A Smith. Dealing with Disaster: Surviving Misbehaved Kernel Extensions. *ACM SIGOPS Operating Systems Review*, 30(213-228):10–1145, 1996.

[92] Amazon Web Services. Elastic Fabric Adapter. https://aws.amazon.com/hpc/efa/, 2025.

[93] Leah Shalev, Hani Ayoub, Nafea Bshara, and Erez Sabbag. A Cloud-Optimized Transport Protocol for Elastic and Scalable HPC. *IEEE micro*, 40(6):67–73, 2020.

[94] Madhavapeddi Shreedhar and George Varghese. Efficient Fair Queueing using Deficit Round Robin. In *Proceedings of the conference on Applications, technologies, architectures, and protocols for computer communication*, pages 231–242, 1995.

[95] Arjun Singhvi, Aditya Akella, Dan Gibson, Thomas F. Wenisch, Monica Wong-Chan, Sean Clark, Milo M.K. Martin, Moray McLaren, Prashant Chandra, Rob Cauble, and et al. 1RMA: Re-Envisioning Remote Memory Access for Multi-Tenant Datacenters. In *Proceedings of ACM SIGCOMM*, pages 708–721, 2020.

[96] Arjun Singhvi, Nandita Dukkipati, Prashant Chandra, Hassan MG Wassel, Naveen Kr Sharma, Anthony Rebello, Henry Schuh, Praveen Kumar, Behnam Montazeri, Neelesh Bansod, et al. Falcon: A reliable, low latency hardware transport. In *Proceedings of ACM SIGCOMM*, pages 248–263, 2025.

[97] Athinagoras Skiadopoulos, Zhiqiang Xie, Mark Zhao, Qizhe Cai, Saksham Agarwal, Jacob Adelmann, David Ahern, Carlo Contavalli, Michael Goldflam, Vitaly Mayatskikh, et al. High-Throughput and Flexible Host Networking for Accelerated Computing. In *Proceedings of USENIX OSDI*, pages 405–423, 2024.

[98] The Linux kernel development community. AF_XDP. https://docs.kernel.org/networking/af_xdp.html, 2025.

[99] The SGLang Team. Deploying DeepSeek with PD Disaggregation and Large-Scale Expert Parallelism on 96 H100 GPUs. https://lmsys.org/blog/2025-05-05-large-scale-ep/, 2025.

[100] William Tu, Yi-Hung Wei, Gianni Antichi, and Ben Pfaff. Revisiting the Open vSwitch Dataplane Ten Years Later. In *Proceedings of ACM SIGCOMM*, pages 245–257, 2021.

[101] ultraethernet/uet-htsim. htsim Network Simulator. https://github.com/ultraethernet/uet-htsim, 2023.

[102] Hao Wang, Han Tian, Jingrong Chen, Xinchen Wan, Jiacheng Xia, Gaoxiong Zeng, Wei Bai, Junchen Jiang, Yong Wang, and Kai Chen. Towards Domain-Specific Network Transport for Distributed DNN Training. In *Proceedings of USENIX NSDI*, pages 1421–1443, 2024.

[103] Xizheng Wang, Qingxu Li, Yichi Xu, Gang Lu, Dan Li, Li Chen, Heyang Zhou, Linkang Zheng, Sen Zhang, Yikai Zhu, et al. SimAI: Unifying Architecture Design and Performance Tuning for Large-Scale Large Language Model Training with Scalability and Precision. In *Proceedings of USENIX NSDI*, pages 541–558, 2025.

[104] Zilong Wang, Layong Luo, Qingsong Ning, Chaoliang Zeng, Wenxue Li, Xinchen Wan, Peng Xie, Tao Feng, Ke Cheng, Xiongfei Geng, et al. SRNIC: A Scalable Architecture for RDMA NICs. In *Proceedings of USENIX NSDI*, pages 1–14, 2023.

[105] Guanbin Xu, Zhihao Le, Yinhe Chen, Zhiqi Lin, Zewen Jin, Youshan Miao, and Cheng Li. AutoCCL: Automated Collective Communication Tuning for Accelerating Distributed and Parallel DNN Training. In *Proceedings of USENIX NSDI*, pages 667–683, 2025.

[106] zartbot/shallowsim. DeepSeek-V3/R1 Inference Performance Simulator. https://github.com/zartbot/shallowsim/tree/main, 2025.

[107] Chenxingyu Zhao, Jaehong Min, Ming Liu, and Arvind Krishnamurthy. White-Boxing RDMA with Packet-Granular Software Control. In *Proceedings of USENIX NSDI*, 2025.

[108] Yanli Zhao, Andrew Gu, Rohan Varma, Liang Luo, Chien-Chin Huang, Min Xu, Less Wright, Hamid Shojanazeri, Myle Ott, Sam Shleifer, et al. PyTorch FSDP: Experiences on Scaling Fully Sharded Data Parallel. *arXiv preprint arXiv:2304.11277*, 2023.

[109] Yinmin Zhong, Shengyu Liu, Junda Chen, Jianbo Hu, Yibo Zhu, Xuanzhe Liu, Xin Jin, and Hao Zhang. DistServe: Disaggregating Prefill and Decoding for Goodput-optimized Large Language Model Serving. In *Proceedings of USENIX OSDI*, pages 193–210, 2024.

[110] Yang Zhou, Xingyu Xiang, Matthew Kiley, Sowmya Dharanipragada, and Minlan Yu. DINT: Fast In-Kernel Distributed Transactions with eBPF. In *Proceedings of USENIX NSDI*, 2024.

[111] Yibo Zhu, Haggai Eran, Daniel Firestone, Chuanxiong Guo, Marina Lipshteyn, Yehonatan Liron, Jitendra Padhye, Shachar Raindel, Mohamad Haj Yahia, and Ming Zhang. Congestion Control for Large-Scale RDMA Deployments. In *Proceedings of ACM SIGCOMM*, page 523–536, 2015.

A Interface to Extensibility

By executing control decisions in software, UCCL-Tran allows flexibly extending its transport implementation for different scenarios. To ease the development of new multipath transport such as new CC or LB policies among different paths, UCCL-Tran exposes a set of expressive interfaces to collective library or ML application developers, as shown in Listing 1.

- `onChunkSize` is called when UCCL-Tran chunks a message for transmission, and it returns the permitted chunk size for now. CC could enforce window control here. After it returns, UCCL-Tran will build a `chunk_desc` for the chunk.
- `onPacingChunk` determines if a chunk needs to queue in the timing wheel for rate pacing, and returns true if so.
- `onSelectPath` is called when a chunk is ready for transmission. `conn_state` contains rich information for selecting path, e.g., RTT scoreboard for each path. It returns the selected `path_id` (i.e., QP ID) for transmission.
- `onTxRtxChunk` is called when the reliable transport wants to retransmit a chunk, and it returns true if this chunk is permitted to retransmit. CC could enforce window control for the retransmitted chunk here.
- `onRxChunk` is called when receiving a data chunk.
- `onRxRtxChunk` is called when receiving a retransmitted chunk. CC could react to the retransmission chunk here.
- `onRxACK` is called when receiving an ACK. CC could react to the ACK here.
- `onRxCredit` is called when receiving a credit. Receiver-driven CC could react to the credit here.

```
1 func onChunkSize(conn_state, remaining_bytes) ->
      chunk_sz;
2 func onPacingChunk(conn_state, chunk_desc) ->
      pacing_or_not;
3 func onSelectPath(conn_state, chunk_desc) -> path_id;
4 func onTxRtxChunk(conn_state, chunk_desc) -> rtx_or_not;
5 func onRxChunk(conn_state, ctrl_hdr);
6 func onRxRtxChunk(conn_state, ctrl_hdr);
7 func onRxACK(conn_state, sack_hdr);
8 func onRxCredit(conn_state, credit_hdr):
```

Listing 1: UCCL-Tran interface to extending multipath transport.

B EQDS Implementation Under UCCL-Tran

Figure 16 shows the overall implementation. For each NIC, UCCL-Tran creates a dedicated pacer thread that operates at a constant rate (derived from the NIC bandwidth) to select candidate senders, allocate credits, and send credits following the EQDS algorithm. Each pacer thread uses a credit UD QP for sending credit packets, and each TX&RX thread also has its own credit QP for receiving credit packets from remote pacers. The pacer thread maintains three lists, i.e., rtx (retransmission), active, and idle sender list, with priorities from high to low. After the TX&RX threads receive data chunks, they will notify the pacer thread via an efficient atomic write in the SHM. Then the pacer will update the sender list: senders who encounter packet loss will be put into the rtx list; senders who have satisfied their requirements will be put into the idle list; otherwise, they will be put into the active list. One note is that as packet tramming is not available in our RDMA NICs and switches, UCCL-Tran uses timeout + RTS (Request-To-Send)

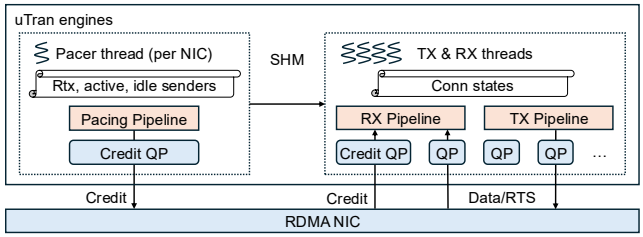
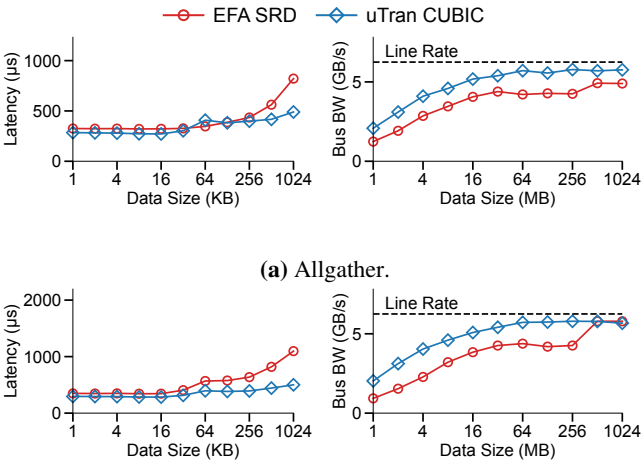


Figure 16: Implementing receiver-driven CC under UCCL-Tran.



(a) Allgather.

(b) Reduce-Scatter.

Figure 17: NCCL-tests results on EFA (NVLink+SHM disabled to simulate a larger testbed).

as a replacement, as suggested by the EQDS paper.

C More Evaluation Results

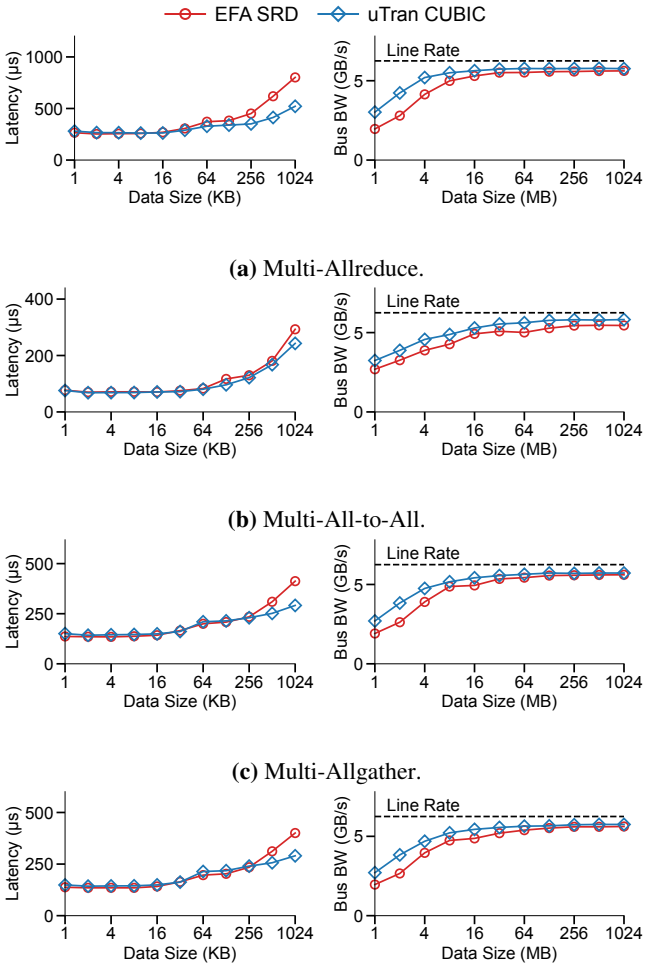
C.1 More collectives in ML workloads

C.1.1 Allgather and reduce-scatter

These two are representative collectives used in PyTorch FSDP (Fully Sharded Data Parallelism) [108], exhibiting low network congestion [34]. Figure 17a and Figure 17b compare the allgather and reduce-scatter performance of UCCL-Tran and SRD on the EFA testbed. UCCL-Tran outperforms SRD by up to 1.68× for allgather and up to 2.18× for reduce-scatter.

C.1.2 Multi-collectives

In multi-collectives, GPUs with the same local rank in each server form a collective group, and multiple groups conduct collectives in parallel. For example, multi-allreduce is used in ML workloads with intra-server Tensor Parallelism (TP) + inter-server Data Parallelism (DP). We evaluate multi-collective performance by setting the environment variable `NCCL_TESTS_SPLIT_MASK=0x7` in NCCL-tests. Figure 18a, 18b, 18c and 18d compare the multi-allreduce, multi-all-to-all, multi-allgather and multi-reduce-scatter performance of UCCL-Tran and SRD on the EFA testbed. UCCL-Tran outperforms SRD by up to 1.54×, 1.22×, 1.46×, and 1.44× for multi-allreduce, multi-all-to-all, multi-allgather and multi-reduce-scatter, respectively.



(a) Multi-Allreduce.

(b) Multi-All-to-All.

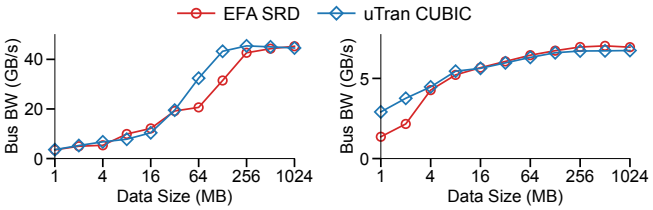
(c) Multi-Allgather.

(d) Multi-Reduce-Scatter.

Figure 18: Multi-collective results on EFA (NVLink+SHM disabled).

C.2 EFA with NVLink+SHM enabled

Figure 19 shows the collective performance comparison on the EFA testbed when NVLink and SHM are enabled (meaning a smaller testbed). Even though a major amount of data traffic goes through the high-bandwidth NVLink, UCCL-Tran still achieves much higher or comparable collective performance than SRD, i.e., up to 1.57× for allreduce and 2.14× for all-to-all.



(a) Allreduce. **(b)** All-to-All.

Figure 19: NCCL-tests results on EFA (NVLink+SHM enabled).

C.3 UCCL-Tran AF_XDP performance

Figure 20 compares the collective performance of UCCL-Tran AF_XDP vs. NCCL kernel TCP with and without proximity placement group (PPG). This experiment is done on AWS with two 50 Gbps VMs connected through AWS ENA NICs that do not support RDMA. We configure NCCL to use multiple TCP connections that achieve the best performance. With PPG that gives low network RTTs between VMs, UCCL-Tran achieves up to 4.1×/2.3× higher performance than NCCL for small/large message ranges; without PPG, UCCL-Tran achieves up to 2.7×/2.1× higher performance. This huge gain is because UCCL-Tran implements an efficient multipath transport atop the fast user-space packet IO technique AF_XDP, thus saving significant user-kernel context switching overhead and heavy-weight networking stack traversing in kernel TCP. For data sizes exceeding 16MB, both AF_XDP and TCP experience bottlenecks due to no GPUDirect support.

C.4 UCCL-Tran Scalability

C.4.1 Varying number of CPU cores on EFA

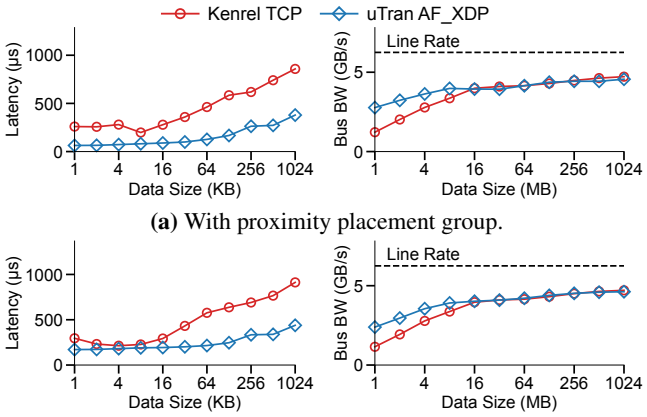
Figure 21 shows how the number of CPU cores per NIC impacts UCCL-Tran performance on EFA. As mentioned in §6.4.3, 2 cores are sufficient to saturate the EFA line rate, even for the connection-intensive all-to-all collective. Overall, thanks to chained posting, UCCL-Tran over UD is able to use 1 core to handle 100 Gbps unidirectional traffic on EFA NICs.

C.4.2 Varying number of GPUs on EFA

Figure 22 shows how UCCL-Tran scales with the number of GPUs on EFA. As expected, with less GPUs, UCCL-Tran achieves lower latency and higher bus bandwidth. But eventually, UCCL-Tran is able to approach line rate with ≥64MB data size. UCCL-Tran leverages the connection-less UD on EFA NICs, thus not suffering from QP scalability issues.

C.4.3 Varying number of paths on EFA

Figure 23 varies the number of paths on the EFA testbed (which crosses racks with multiple network paths). The high-level take here is that multipathing helps mitigate network congestion caused by, e.g., flow collisions. We expect multipath transport to shine more on a larger testbed with more network paths.



(a) With proximity placement group.

(b) Without proximity placement group.

Figure 20: Allreduce performance on non-RDMA NICs using two AWS `g4dn.8xlarge` VMs each with a 50 Gbps AWS ENA NIC.

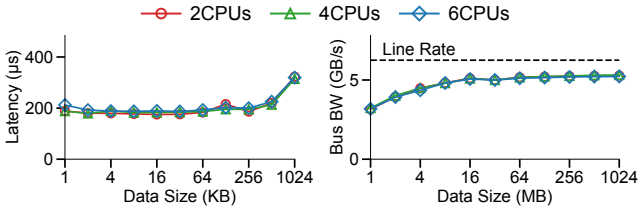
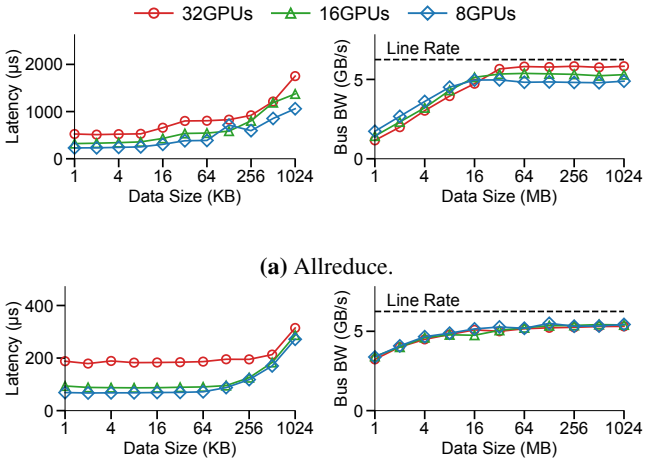


Figure 21: UCCL-Tran all-to-all on EFA with different numbers of CPU cores per NIC (NVLink+SHM disabled).



(a) Allreduce.

(b) All-to-All.

Figure 22: UCCL-Tran collectives on EFA with different numbers of GPUs (NVLink+SHM disabled).

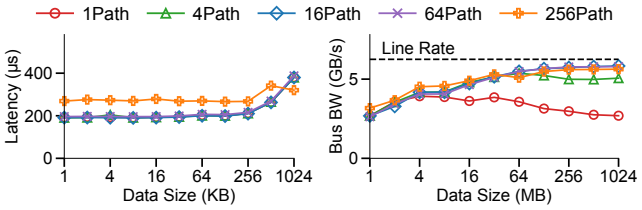
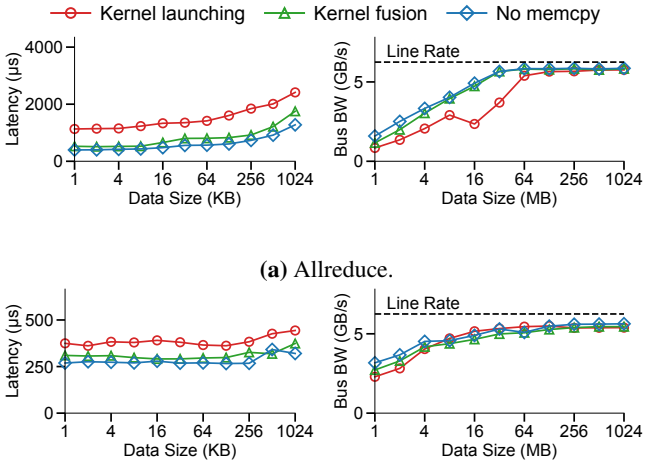


Figure 23: UCCL-Tran all-to-all on EFA with different numbers of paths (NVLink + SHM disabled).

Permulation traffic completion time (ms)	Sender-driven	Receiver-driven
4KB + ECN (UEC default)	1.45	1.43
32KB + ECN	1.62 (+11.7%)	1.47 (+2.8%)
32KB + RTT	1.71 (+17.9%)	1.47 (+2.8%)
32KB + RTT + ConnSplit	1.70 (+17.2%)	1.47 (+2.8%)
16KB + RTT + ConnSplit	1.51 (+4.1%)	1.42 (-0.7%)

Table 5: Impact of chunk size and LB policy. "ConnSplit" means connection splitting, which first selects the least loaded engine and then selects the least loaded path from that engine's paths/QPs using Power-of-Two sampling (§4.1).



(a) Allreduce.

(b) All-to-All.

Figure 24: NCCL-tests results on EFA (NVLink+SHM disabled).

C.5 Design Drill-Down

C.5.1 Impact of chunk size and LB policy

This experiment aims to study the impact of chunk size and LB policy on the performance of a broad range of ML transports (not just the ones implemented in UCCL-Tran). To this end, we use UEC's packet-level network simulator `htsim` [22,101] with UEC-standard multipath transport implementation. We further modify the transport implementation in `htsim` to vary chunk size and LB policy, e.g., based on ECN or RTT, connection splitting or not. We vary the chunk size by changing the MTU size in the simulator without modifying any transport simulation code. We vary the LB policy by maintaining per-path ECN/RTT and modifying the code where the transport sender selects the path for each packet. Similar to prior work [16,19,59,89], we focus on the permutation traffic pattern to stress-test the transport, where each NIC streams traffic to another and no one receives more than one stream. We simulate 1024 400G NICs under a fully-provisioned three-tier fattree, and each NIC uses 256 paths to stream 64MB traffic with an ideal completion time of 1.28ms.

Table 5 shows the permutation traffic completion time under different designs. Overall, using 32KB chunk size and switching to RTT degrade transport performance by 17.9% for sender-driven transport and only 2.8% for receiver-driven, while connection splitting does not degrade. With a 16KB chunk size, the performance degradation of sender-driven transport becomes only 4.1%. Receiver-driven performs better because EQDS leverages in-switch packet trimming to quickly detect network congestion and react [43,85]. Overall, this experiment shows that control coalescing indeed causes transport performance degradation, but the degradation is moderate in most cases; meanwhile, connection splitting has a negligible impact on transport performance.

C.5.2 Impact of kernel fusion.

This experiment aims to quantify the overhead of scattered memcpy and justify UCCL-Tran's design choice of employing kernel fusion over kernel launching (§3.1). In the kernel launching approach, UCCL-Tran launches a dedicated copy thread (in CPU) for each UCCL-Tran engine at boot time; after receiving all packets of a transport buffer, the engine notifies its associated copy thread via a shared-memory queue; then the copy thread launches a GPU kernel to perform scattered memcpy asynchronously (this kernel is different from the reduction kernel). We further optimize the kernel launching overhead with adaptive batching [14]. Note that we cannot run the GPU kernel persistently, as this would cause deadlock with the commonly-used `cudaDeviceSynchronize()`. We also emulate the performance with no scattered memcpy by skipping packet reassembly entirely and disabling data correctness checks in NCCL-tests.

Figure 24 shows the collective performance. Scattered memcpy introduces minor performance overhead compared to no memcpy (less than 8% and 5% for allreduce and all-to-all, respectively). Kernel launching performs worse than kernel fusion for both small message latency and large message bandwidth. The suboptimal performance of Kernel launching is because of the high kernel launching overhead, especially for small messages.

C.5.3 PCIe overhead

This experiment aims to study the PCIe overhead introduced by UCCL-Tran's multipath design. We measure the PCIe

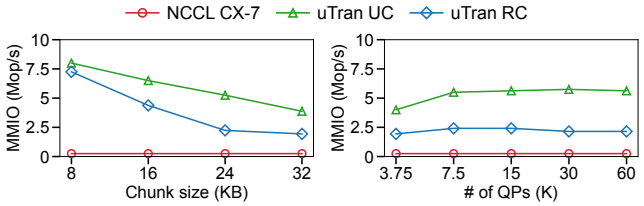


Figure 25: Number of MMIO events for each NIC under various chunk sizes and numbers of QPs on CX_IB.

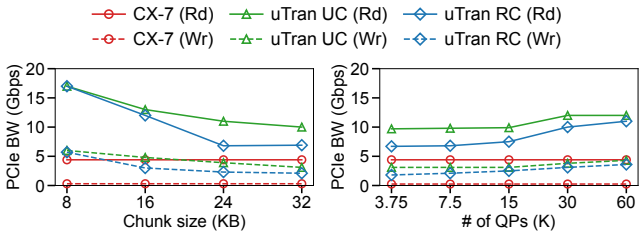


Figure 26: Extra PCIe traffic for each NIC under various chunk sizes and numbers of QPs on CX_IB. Rd: PCIe device reads from CPU; Wr: PCIe device writes to CPU.

overhead on our CX_IB testbed, while the AWS virtualization layer prevents us from accessing low-level PCIe metrics. We rerun the all-to-all experiments in §6.1 and quantify MMIO events and CPU-NIC PCIe traffic using `pcm-pcie` [50]. Figure 25 and 26 show the results. As expected, UCCL-Tran incurs higher MMIO activity and PCIe bandwidth consumption than the vanilla NCCL atop CX-7. UCCL-Tran incurs more MMIO events because it posts more verbs: UCCL-Tran transfers a small 32KB chunk per verb, while the vanilla NCCL atop CX-7 transfers a transport-buffer-sized message (e.g., default 128KB) per verb. The extra PCIe traffic of UCCL-Tran primarily comes from NIC swapping QPs (i.e., fetching uncached QP contexts from CPU memory) as UCCL-Tran uses 256 UC/RC QPs for multipath. UCCL-Tran UC consumes slightly more bandwidth than RC because it implements reliability and selective retransmission in software on the CPU.

We conclude that the number of MMIO events and extra PCIe traffic increase significantly with smaller chunk sizes, but the number of QPs does not nearly affect them. UCCL-Tran makes a reasonable trade-off between the control decision granularity and performance based on this observation and uses a 32KB chunk size as default (§3.3). We note that the extra PCIe bandwidth overhead is minor compared to the PCIe link capacity. For example, the capacity of PCIe 5.0 $\times$ 16 on CX_IB is 512 Gbps in each direction, and the overhead of UCCL-Tran (32KB, 60k QPs) accounts for less than 2.5% (12/512=2.3%).

Hardware Lifecycle-Aware Power Planning in Commercial Hyperscale Datacenters (Operational Systems)

Ruihao Li *Meta & UT-Austin*
Leonardo Piga *Meta*
Wei Su *Meta*
Carlos Torres *Meta*
Jovan Stojkovic *Meta & UT-Austin*
Neeraja J. Yadwadkar *UT-Austin*
Lizy K. John *UT-Austin*
Abhishek Dhanotia *Meta*

Abstract

Modern commercial datacenters operate with heterogeneous hardware, deploying new servers to meet the growing compute demands while retaining older servers due to budgetary and environmental constraints. Datacenter planners can use daily performance and power profiling data to continually refine power budgets for legacy hardware. By contrast, power planning for new hardware must begin as early as the pre-silicon stage of server development when such data is not yet available. This paper presents our practical experience in power planning for Meta's datacenters over the past decade, emphasizing heterogeneous hardware, and shares strategies for future power planning and management.

To plan power more efficiently, we begin with a comprehensive power characterization study for hyperscale workloads in Meta's datacenters, using live production traffic data spanning millions of servers across multiple hardware generations. Building on characterization insights, we present a *hardware lifecycle*-aware rack power budgeting methodology accounting for both the hardware and workload heterogeneity. This methodology has been deployed at scale in Meta's datacenters for over a decade, enabling an average power oversubscription of approximately 20% across the fleet.

Though effective, *the current power planning approach requires production-level power telemetry, which is typically unavailable during the early stages of hardware development.* To address this challenge, we develop *PowerSight*, a machine learning-based model to predict system power without relying on power sensor data. We demonstrate practical use cases of *PowerSight* for improving power planning in future system deployments. To the best of our knowledge, this is the first comprehensive study to formally introduce the concept of *hardware lifecycles* in commercial datacenters, along with tailored power budgeting strategies for each phase.

1 Introduction

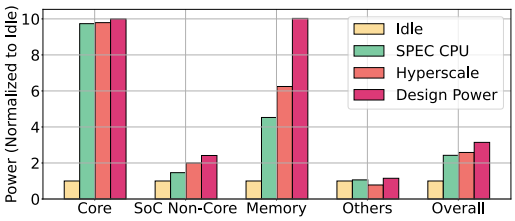
As the demand for digital services continues to rise, the power consumption of the datacenters to support them is growing rapidly [10, 44, 50, 51, 65]. To meet the growing demands, Meta – like other major datacenter operators such as Google [30, 56], Microsoft [25, 51], and Alibaba [73, 74] – continues to grow its server fleet by adding new hardware, while maintaining older systems to balance both budgetary and environmental constraints [24, 52, 68].

This mix of old and new hardware creates *heterogeneity*, making it necessary to account for *hardware lifecycles* when conducting power planning. A hardware lifecycle (§ 2) is the complete sequence of stages that a rack program goes through, from initial planning and acquisition, through deployment and operation in mass production, to eventual decommissioning and disposal. For older servers in mass production, a wealth of historical power and performance data is available across the fleet, enabling power planning methodologies based on past observations [23, 27, 34, 52, 64, 65, 77]. In contrast, when a new hardware generation is introduced, large-scale power sensor data typically becomes available only after mass production begins. However, the initial phase of rack installation demands power planning decisions before comprehensive data is available.

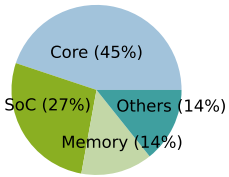
To address the challenges, we develop the *hardware lifecycle-aware power budgeting* methodology by answering the following questions:

- What are the power characteristics of heterogeneous datacenter hardware and workloads? What are the power trends across generations of hardware?
- How can power planning be adapted across hardware lifecycle stages at scale using these power insights?
- What are alternative power planning methods for early lifecycle phases when power sensors are unavailable?

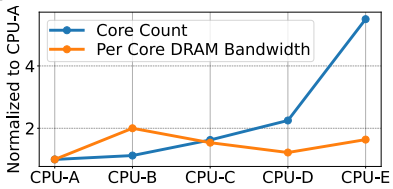
Effective power planning for hyperscale datacenters – such as Meta's, which deploy microservices across globally distributed infrastructure – requires in-depth workload characterization throughout the hardware lifecycle for power planning. To this end, we present a comprehensive analysis of system power based on live production traffic collected across



(a) Hyperscale vs. *SPEC CPU*.



(b) Power breakdown.



(c) Server trend across time.

Figure 1: (a) Core power approaches its design power, while memory power stays well below, especially for *SPEC CPU*. (b) System power breakdown of a hyperscale web service; core power is less than half of the total. (c) Server core counts have increased over CPU generations, whereas memory bandwidth per core has not kept pace.

multiple system types (Compute, Storage, AI) within Meta's datacenters at the beginning of our study.

Hyperscale workloads exhibit behaviors that differ significantly from those of standard benchmarks [22,33,61,62]. Figure 1a shows component-wise power contributions for an idle system, a *SPEC CPU2017* workload (geometric mean of int rate), and a representative hyperscale web service. Both workloads achieve peak CPU utilization, while hyperscale workloads place substantially greater power demands on the memory subsystem and other non-core components than *SPEC CPU2017* workloads. Furthermore, focusing solely on processor core power is insufficient, as it accounts for less than 50% of the system power (Figure 1b). Additionally, as illustrated by Figure 1c, server compute and memory capabilities differ across generations in Meta's datacenters.

Building on insights from our characterization study, we have enabled the infrastructure to collect component-level performance counters and power sensor data across all machines in our fleet. This fleet-wide telemetry supports power planning that accounts for the heterogeneity of both workloads and hardware throughout the hardware lifecycle, capturing a critical reality: *not all services or machines reach peak power simultaneously.* Leveraging this heterogeneity, we have developed a power oversubscription methodology that has been deployed across Meta's datacenters for over a decade. By aligning power budgeting with each stage of the hardware deployment lifecycle, this approach has enabled Meta to provision approximately 20% more racks in its global datacenter fleet without increasing the overall power footprint. *Notably, this work is the first to introduce the concept of hardware lifecycles in heterogeneous datacenter environments, along with phase-specific power budgeting strategies – an area not explored in prior datacenter power studies [27,34,52,59,64,77].*

Finally, *in the early stages of the hardware lifecycle, power planning must proceed without the production-level telemetry available in later phases*. To address this challenge, we introduce *PowerSight*, a machine learning (ML)-based model that predicts system power consumption using performance counters[1], which are available throughout all phases of the hardware lifecycle. *PowerSight* is a holistic power model capable of predicting the power consumption of compute, storage, and AI servers, as co-locating different types of servers within the same datacenter is essential to minimize cross-region communication overhead, which is particularly important for services such as AI training [15]. We also present case studies on how *PowerSight* facilitates power planning during the initial phases of the hardware lifecycle in datacenters.

In summary, our key contributions include:

- We conduct a comprehensive power characterization study using live production data at Meta, covering thousands of services deployed on multiple generations of compute, storage, and AI servers.
- We present the power oversubscription methodology that has been deployed in Meta's datacenters for over a decade, which allows us to safely oversubscribe power by ∼20% on average in compute, storage, and AI racks.
- We introduce *PowerSight*, an ML-based model for early-stage datacenter power planning. *PowerSight* can be easily deployed at scale using existing datacenter infrastructure. We also plan to publicly release artifacts to benefit the broader community in the future.

2 Background

Our study begins by introducing the foundational concepts of power distribution in Meta's datacenters.

Definition of hardware lifecycles: Commercial datacenters typically use a mix of old and new machines to handle a massive number of workloads [25,30,51,52,56,68,73,74]. As service load increases over time, datacenters add new machines to increase capacity, but often keep older machines in operation due to cost and environmental concerns [4,24,72]. Introducing a new hardware generation into production is a multi-phase process, which we refer to as the *hardware lifecycles* (Table 1). The process begins with the *pre-engineering validation test (pre-EVT)* phase, which focuses on platform

[1]Though prior works have explored using ML-based power models for power prediction [12,17,19,37,47,82], we are the first to demonstrate the real use cases of ML models for power planning in datacenters. The primary goal of this paper is to present a power planning system across hardware lifecycles that could be deployed at scale.

Table 1: Hardware lifecycle of a server. It takes 18+ months before hardware moves into the mass production (MP) stage. Power sensors are usually equipped in the DVT/PVT stage.

Stage	Timeline	Goal	Usage
Pre-EVT	2-3 years before MP	Silicon & platform design	Forecast RPB
EVT	9-12 months before MP	Tooling bring-up	Forecast RPB
DVT	6 months before MP	App test & power plan	Forecast RPB
PVT	3 months before MP	Production evaluation	Initial RPB
MP		Mass production	Initial RPB
MP+1 year		Scale deployment	Refined RPB

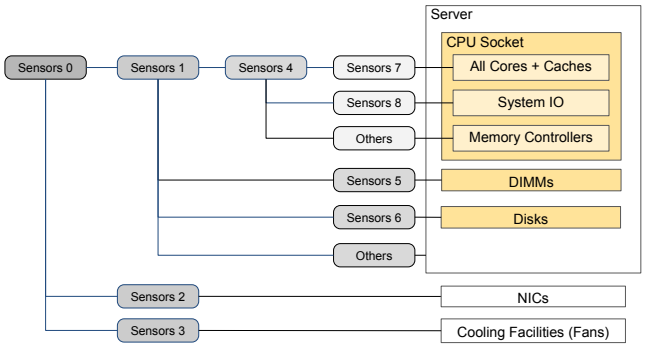


Figure 2: Telemetry sensors in Meta's datacenters. Sensors are deployed at different levels to measure power across different components and subsystems.

design and pre-silicon performance evaluation. In the *engineering validation test (EVT)* phase, where hardware is available, efforts shift to system bring-up and performance benchmarking. The subsequent *design validation test (DVT)* and *production validation test (PVT)* phases involve application testing, power planning, and production validation. During this period, servers are seamlessly equipped with baseboard management controllers (BMCs) to enable reliable power data collection through physical power sensors deployed at various system levels, as shown in Figure 2. After approximately nine months of extensive testing, the platform enters the mass production (MP) phase. Even during mass deployment, machine configurations – such as CPU frequency – are continuously tuned to maximize performance-per-Watt and overall energy efficiency (MP + 1 year).

Datacenter power distribution hierarchy: Figure 3 shows the hierarchical power distribution architecture in Meta's datacenters, which enables efficient power delivery and centralized management across the fleet. At the top, the main switch delivers up to ∼2.5 MW to IT equipment. Downstream, intermediate switchboards draw ∼1.25 MW each and feed multiple leaf-level panels, each sustaining up to ∼190 kW. These panels power tens of *heterogeneous* racks hosting diverse machine types hosting different services [27, 52].

Definition of design power: In datacenters, *power control is typically managed at higher aggregation points, such as main*

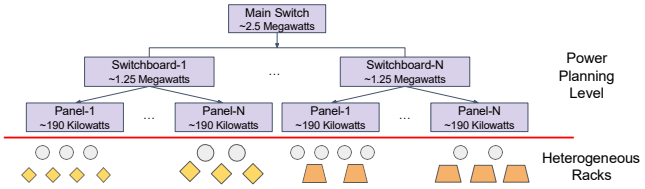


Figure 3: Power distribution hierarchy of the fleet.

switches, switchboards, or panels, rather than at individual racks [27, 52, 77]. This simplifies power management by reducing the number of monitored units. However, each rack still has a fixed power limit set by its electrical design, and the total power drawn by all components, including servers and the top-of-rack (TOR) switch, must stay within this limit. To enforce this constraint, datacenter operators define the concept of *design power*, denoted as D, which captures the peak power usage of all servers under the most demanding workload, plus overheads such as TOR switch power.

Definition of rack density: In datacenter power planning, an important metric alongside *design power* is *rack density*, defined as the maximum number of servers, denoted as n, that can be deployed without exceeding the *design power* of each rack. A typical method to calculate rack density involves three steps: (a) determine the peak server power, P_s, under the most power-intensive workload; (b) account for the power consumed by other rack components, such as TOR switch, P_{TOR}; and (c) calculate n accordingly: $n = (D - P_{\text{TOR}})/P_s$.

Definition of rack power budget: In practice, the *design power* of a rack is often unattainable, as not all services can fully stress every server component simultaneously [27, 34, 64, 77]. To this end, we define the *rack power budget* (RPB) as the expected power draw of a given rack type during peak traffic, accounting for worst-case fleet-level failure scenarios. We routinely update RPB throughout the hardware lifecycle (see Table 1) to optimize overall datacenter power utilization. Additionally, accurate RPB planning is essential for new hardware during the early stages of its lifecycle, as these initial estimates play a key role in determining whether additional datacenters are needed to support increased fleet capacity. *However, this process is challenging due to the lack of fleet-wide power sensor data before massive production.*

3 Understanding Power at Hyperscale

Leveraging fleet-wide performance counters and per-component power sensors available on recent platforms, we characterize the power consumption of hyperscale workloads in Meta's production datacenters. We cover experimental setup (§ 3.1), benchmarks vs. production services (§ 3.2), power variance across services (§ 3.3), cross-generational trends (§ 3.4), and performance-power correlations (§ 3.5).

Table 2: Machine configurations of all types of services (* denotes a variant configuration).

Service	CPU Gen	GPU Gen	Data Drive	# of configs
Compute	CPU-A, CPU-B, CPU-C, CPU-D, CPU-E	N/A	N/A	13
Storage	CPU-B*, CPU-C*, CPU-F	N/A	SSDs	10
AI (Inference)	CPU-C*, CPU-G	GPU-A, GPU-B	SSDs	9
AI (Training)	CPU-C*, CPU-F, CPU-H	GPU-A*, GPU-B*	SSDs	10

3.1 Experimental Setup

Our study focuses on hyperscale workloads running on millions of machines, encompassing thousands of services, from traditional web and network services [33, 34, 62, 63, 81] to AI-related workloads [50, 51, 65]. We specifically analyze the power consumption of in-production servers handling live production traffic spanning eight CPU generations (Intel Xeon and AMD EPYC architectures) and two GPU generations (GPU-A and GPU-B, corresponding to NVIDIA Ampere and Hopper architectures, respectively) running (general) compute, storage, and AI services[2]. We list machine configurations in Table 2 (further details are publicly available in the Open Compute Project [2]). Compute services run on nodes that have CPUs and boot drives (see Table 3). Storage services use servers equipped with flash devices. AI services, both inference and training, operate on hosts provisioned with GPUs. In total, we used 42 machine configurations with different combinations of core counts and socket counts, as well as cache and DRAM capacities to conduct our study[3].

We measured each service in the production environment under its default deployment configuration and collected system-level performance counters by running a lightweight daemon process *dynolog* on each production server [1]. This infrastructure setup allows us to query historical performance metrics data from a massive database that contains information from millions of machines. We read system module power from Sensor 0 (measuring module input power), server power from Sensor 1 (measuring server power including CPU sockets, DIMMs, and disks), and CPU socket power from Sensor 4 (measuring processor die power) using an internal datacenter-wide power management system that monitors the entire fleet power hierarchy [77], as labeled in Figure 2, which shows the physical placement of each sensor in the power delivery path.

[2]Each type of service is deployed on a specific cluster of machines, with each cluster having its own unique machine configuration.

[3]We did not count a machine configuration with different data drive capacity as a new configuration, but if considered, there would be over 100 distinct machine configurations in total.

Table 3: CPU configs across compute server generations.

CPU Gen	CPU-A	CPU-B	CPU-C	CPU-D	CPU-E
Year Deployed	2016	2018	2021	2022	2023
# of Cores	16	18	26	36	88
DRAM Tech	DDR4	DDR4	DDR4	DDR4	DDR5
DRAM Capacity	32	32	64	64	256
DRAM Channels	2	4	4	4	8
DRAM Speed	2133	2400	2666	2933	4800

3.2 Benchmarks vs Production Services

In the early stages of power planning, benchmarks are commonly used as proxies for in-production services to estimate system design power. However, standard benchmark suites often underestimate design power, as they fail to match the power intensity of hyperscale workloads. This section examines the differences in power characteristics between hyperscale services and conventional benchmarks.

From the thousands of services in the fleet, we select representative ones[4] to conduct in-depth case studies.

Compute Services: *Web1* and *Web2* are web servers implemented using HipHop Virtual Machine (HHVM) deployed across the fleet [5, 46, 61, 62, 81]. *Feed* is a key microservice for the News feed service [61, 62, 84]. *Ads* maintains user-specific data, ranks advertisements, and determines their retrieval order [61, 62]. *Video* is a service for video transcoding.

Storage Services: *UDB* is a user database service built on MySQL and deployed under a distributed data store architecture. *ZippyDB* is a distributed key-value store that builds on RocksDB to ensure data consistency and reliability [14].

Benchmarks: We use *SPEC CPU2017* [13, 18] to compare its performance and power characteristics with hyperscale workloads. If not specified, we run *int rate* using n copies, where n represents the number of cores in the system.

While *SPEC CPU2017* benchmarks are designed to represent a broad range of computing tasks, hyperscale workloads are tailored to specific applications and exhibit unique power consumption patterns. Figure 4 shows that *SPEC* workloads exhibit lower power consumption, whereas hyperscale workloads operate closer to the design power. On average, hyperscale workloads utilize 85.6% of design power, whereas *SPEC* workloads reach only 75.5%, approximately 11.8% lower in relative terms. In contrast, hyperscale workloads exhibit higher power consumption for (a) SoC non-core power, namely network-on-chip, system IO, and memory controller power; (b) memory power; and (c) other power such as network interface, rack top-of-switch power, and fan power. This is because hyperscale workloads typically have a larger in-

[4]The microservices deployed at Meta's datacenters follow a long-tail distribution, where a small subset of workloads accounts for the majority of hosts within a given rack type [67]. For instance, in the latest compute rack, the top ten workloads collectively occupy 55% of all hosts.

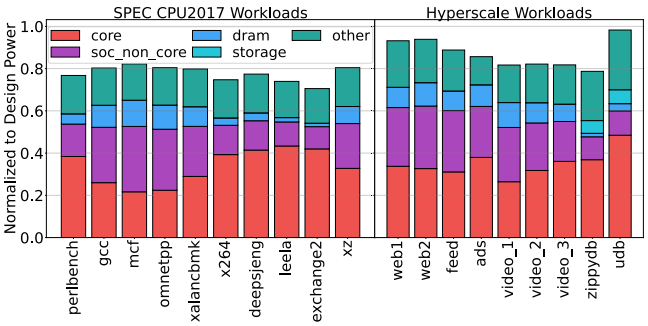


Figure 4: Power breakdown of *SPEC CPU2017* benchmarks and Hyperscale workloads. Hyperscale workloads consume similar core power but higher system power than *SPEC* benchmarks. On average, hyperscale workloads reach 85.60% of design power, whereas *SPEC* benchmarks reach only 75.51%.

struction footprint, which places greater demands on the memory subsystem [33, 43, 62].

Takeaway 1: (1) Conventional benchmarks, like *SPEC CPU2017*, can effectively measure core power consumption but often fail to capture the power profile of the whole system. (2) By leveraging live production data for power planning, we avoid underestimating design power by 11.8% compared to relying solely on standard benchmarks.

3.3 Power Variance Across Services

Instead of general-purpose benchmarks, we characterize the power consumption of production services. In Figure 5a, we show a snapshot of system power usage for thousands of live traffic services running on millions of machines in the fleet, including compute, storage, and AI services. Typically, services operate within 40-70% of their design power capacity. However, our analysis reveals that actual power consumption can fluctuate substantially, ranging from as low as 20% to as high as 90% of design power for different machine variants. The power variance shown in Figure 5a reflects both *inter-service* and *intra-service* differences. Different services can place varying demands on system components (e.g., a memory-intensive caching service vs. a compute-bound video transcoding service), and even instances of the same service can exhibit power variation due to regional load imbalances and hardware configuration differences. The significant variation in power consumption resulting from fluctuating system loads in commercial datacenters [27, 51, 52, 64, 65] highlights the importance of using data from the entire fleet to accurately characterize system power.

Takeaway 2: The system power varies significantly during runtime (varies between 20%-90% of the design power), which poses great challenges for datacenter RPB planning.

Challenges in RPB planning: Workload power consumption varies substantially, with different workloads using different proportions of their design power capacity (Figure 5a). Consequently, servers within a compute rack show considerable variation in power usage (Figure 5b). Notably, only about 85% of compute rack servers – similar to those in storage and AI racks – operate above 50% of their design power as shown in Figure 5b. This variability highlights a key limitation of relying solely on *design power* for datacenter power budgeting, as it can lead to underutilized power capacity and inefficient resource allocation. To address this, we continuously refine the RPB throughout the entire hardware lifecycle (§ 4).

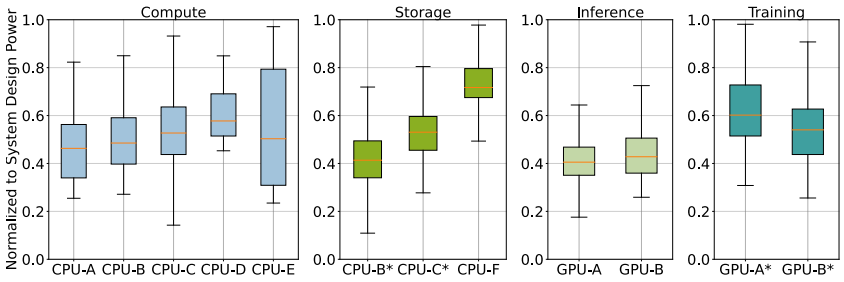
3.4 Power Trend Across Machine Generations

The characterization results in the previous section show that each machine generation in the fleet has its unique power characteristics, but there is also a noticeable trend across generations that can provide valuable insights for hardware architects planning future datacenter capacity. For instance, as shown in Figure 5a, newer machine generations like CPU-E exhibit increased variability in system power consumption. This wider spread is driven by the broader diversity of workload profiles supported by newer platforms – with significantly more cores (e.g., 88 cores in CPU-E vs. 18 in CPU-B) and memory channels. As a result, some services run compute-bound while others are memory- or I/O-dominant, leading to a wider range of component utilization levels and thus a wider power distribution even when normalized to design power.

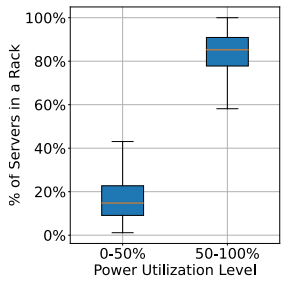
In addition to overall system power consumption, we also observe trends across machine generations for individual components within the system. Figure 6a shows the trend of CPU power consumption per unit of throughput (measured as committed instructions per time frame) across various machine generations. Our analysis reveals that as the number of cores increases across machine generations, the per-throughput socket power decreases. This trend is particularly prominent when comparing later machine generations, such as CPU-E, where we see nearly a 2× reduction in per-throughput CPU socket power compared to CPU-B (CPU-E has roughly 5× more cores than CPU-B and about 2.5× more cores than CPU-D). In contrast, components of the memory subsystem in servers did not improve at the same rate as cores (only 1.2× when comparing DDR5 versus DDR4, as shown in Figure 6b). As a result, there is a shift in power distribution across different system components over time.

Figures 7a (focus on compute servers) and Figures 7b (focus on AI training servers) illustrate the trend in system power breakdown for each component across machine generations. The proportion of memory components over system power has increased significantly over time[5]. Notably, the next-generation pre-production system, CPU-X and GPU-X (GPU-0 represents the previous decommissioned generation), exhibits an even more pronounced trend, indicating that mem-

[5]The only exception is the transition from GPU-A to GPU-B, where advancements in HBM technology enhanced power efficiency (the power efficiency of HBM3 has improved by more than 50% compared to HBM2 [36]).

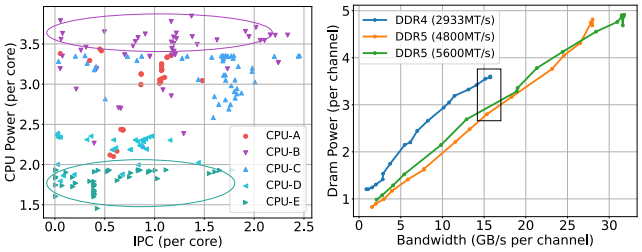


(a) Fleet-wide service system power distribution for all machine generations.



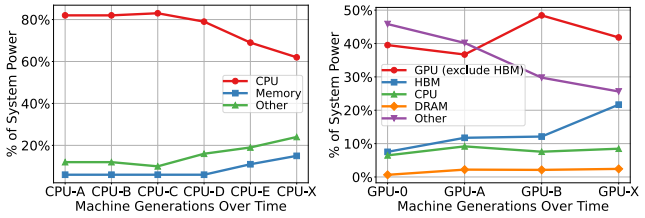
(b) Server distribution within a rack.

Figure 5: (a) Service power distribution across the fleet shows significant variability across all service types. (b) Server distribution within a compute rack with different levels of design power utilization. 15%+ of servers have relatively low power utilization (less than 50% of design power).



(a) CPU power. (b) Memory power.

Figure 6: CPU core power and memory power trend across machine generations at different performance levels. Core power drops by ∼2× in newer systems at the same IPC level (CPU-E vs. CPU-B), while memory power shows a more modest reduction of ∼1.2× when delivering equivalent bandwidth (DDR5 vs. DDR4).



(a) Components (Compute). (b) Components (AI Training).

Figure 7: System power breakdown across machine generations – (a) compute (Memory line) and (b) AI training servers (HBM line) – shows memory components contributing increasingly over time.

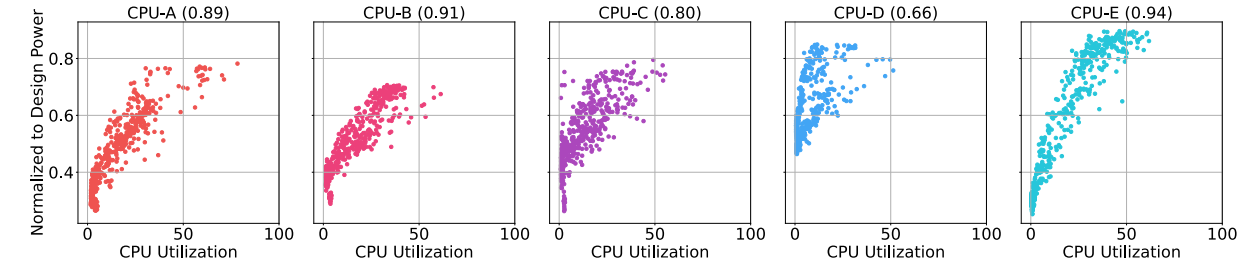
ory components will continue to play an increasingly important role in overall system power usage.

This trend has two competing effects on future power planning. On one hand, memory power exhibits a relatively stable relationship with bandwidth utilization (Figure 6b), which may simplify power prediction as a larger share of system power becomes more predictable. On the other hand, the growing memory power fraction suggests that memory-aware power management techniques will become increasingly important. The effectiveness of such techniques depends on workload characteristics, as different services place varying demands on the memory subsystem, and memory behavior of each workload may change over time [33, 43, 62]. This motivates program-phase-aware power management solutions that could, for example, dynamically reallocate power from the memory subsystem to CPUs or other components during memory-light phases, improving performance under the same power budget.

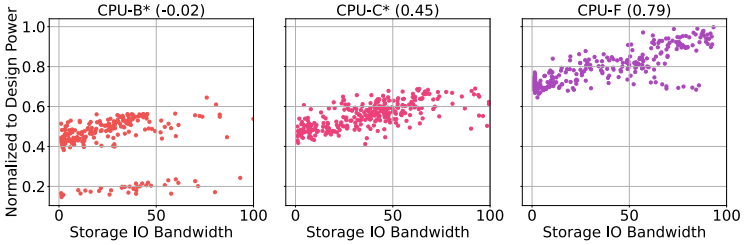
Takeaway 3: (1) The power efficiency of system components evolves at different rates. Over time, memory power has grown to form an increasing percentage of total system power (>20% in the latest generation), which presents both an opportunity (more predictable power fraction) and a challenge (need for memory-aware power management techniques) for datacenter power management. (2) Rather than focusing solely on CPU power, we include all system components in power planning across hardware lifecycles. In recent platform generations, we have added dedicated power sensors for the memory subsystem to account for the evolving hardware trend (see Figure 2).

3.5 Perf. Metrics – System Power Correlation

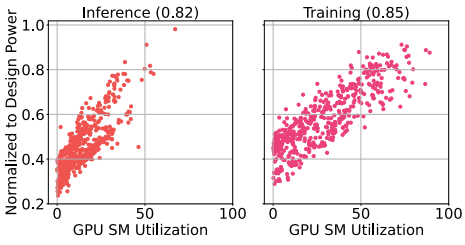
In addition to studying system power breakdown trends across machine generations, we analyze correlations between performance metrics and system power to gain deeper insights into the challenges of power modeling. We find that the system power is affected by multiple performance metrics, and this relationship differs across machine generations. For instance, in some generations like CPU-A, peak system power is reached at approximately 70% CPU utilization, whereas in others, such as CPU-E, peak power occurs at a much lower utilization level, around 40% (as shown in Figure 8a). One explanation for this is that newer CPUs optimize SoC power usage by maximizing frequency, which enables them to reach



(a) Correlations between CPU utilization and system power.



(b) Correlations between storage IO utilization and system power.



(c) GPU utilization and system power.

Figure 8: Fleet-wide correlations between CPU utilization, Storage IO bandwidth, GPU utilization, and system power. No single metric can decide the system power, and the correlation (listed in figure titles) changes across machine generations.

maximum power at lower utilization levels. The relationships between storage IO bandwidth, GPU SM utilization, and system power in storage and AI services also exhibit complex patterns, emphasizing the need for a multi-metric approach to accurately characterize system power.

Takeaway 4: (1) Modeling the power of an entire system requires multiple inputs beyond just the metrics of the main components (e.g., CPUs/GPUs), and thus firmware-level readings, such as NVIDIA Nsight Systems reporting only GPU power, are insufficient. *To address this, we deploy a lightweight daemon process dynolog [1] on each server to collect performance counters of all components in the system.* (2) It is unreliable to directly extrapolate power models from current systems to future platforms. *A holistic power model capable of predicting the power consumption of future compute, storage, and AI servers is necessary to support the hardware heterogeneity in datacenters [15, 52].*

4 Hardware Lifecycle-aware Power Planning

Having characterized power at the individual machine level (§ 3), we turn to higher-level planning. Although each service and machine exhibits a distinct power profile, effective datacenter power planning requires decisions at coarser granularities, such as the rack. To address this, we have developed a methodology for oversubscribing rack power across the hardware lifecycle, accounting for both workload and hardware heterogeneity (§ 4.1). In early deployment stages, however, power sensors may not yet be available, necessitating a cross-architecture model for power prediction (§ 4.2).

4.1 Oversubscribe RPB across Hardware Lifecycles

The wide variation in power consumption across machines and services (Figure 5a) demonstrates the drawbacks of relying solely on design power for capacity planning, as this results in stranded power capacity throughout the fleet. In practice, all machines within a rack rarely operate at their peak power simultaneously (Figure 5b). Consequently, datacenter planners can safely set the RPB below design power, which improves overall power utilization across the fleet. Setting the right RPB requires balancing multiple competing constraints. If the RPB is set too low, more racks are placed under each power device, and aggregate power draw may exceed upstream device limits (RPP, SB, MSB), which triggers power capping [23, 77] and degrades both performance and efficiency. Conversely, an RPB set too high leaves provisioned power capacity unused, requiring more datacenters to be built for the same total compute capacity, which raises costs and environmental impact. In this section, we present our RPB planning methodology, deployed at scale for over a decade, that spans the full hardware lifecycle from early design through production optimization.

Forecast RPB (Pre-EVT/EVT/DVT): In the early stages of hardware lifecycles, datacenter planners must estimate the RPB to anticipate future power demands as machines begin to arrive. Accurate forecasting is critical: (a) underestimating forecasted RPB means deployed racks may exceed the power provisioned for their location, forcing a late RPB increase that reduces rack density per power device and necessitating rack relocations to stay within device limits; (b) overestimating

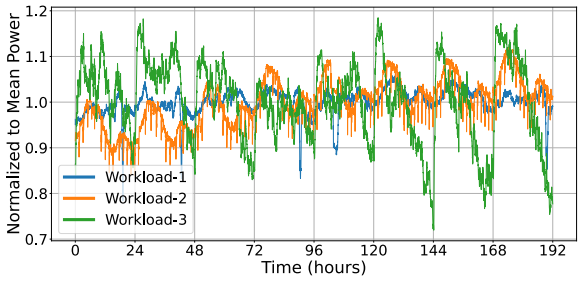


Figure 9: Power variance of three top services over a week. Each workload reaches its peak power at different time.

RPB under-projects compute density, potentially triggering unnecessary datacenter construction. Without measured data, this projection relies on the maximum power ratings defined in the electrical specifications of each rack, adjusted using derating factors based on historical experience.

Initial RPB (PVT/MP): Once systems are delivered, we work with service owners, who have detailed knowledge of their workloads, to determine the RPB. Service owners first conduct load tests to identify the highest sustainable load before performance degradation occurs. They then measure the utilization of the most critical resource (e.g., CPU, GPU, or disk), referred to as the target utilization, for that specific rack type. We use values supplied by service owners to calculate the power budget for each service. By generating a power-versus-utilization curve for every workload and applying the target utilization value, we determine the corresponding power demand, denoted as p_w. In addition to p_w, we also calculate a weighted average distribution of each workload within the rack, denoted as f_w. Finally, we derive the RPB (Equation 1)[6], where each w denotes a distinct workload, n indicates the number of hosts and P_{TOR} represents additional power consumed by TOR switches.

$$RPB = \sum_{w=0}^{W-1} (f_w \times p_w) \times n + P_{\text{TOR}} \tag{1}$$

Refined RPB (MP+1 year): The initial RPB would be sufficient if all services deployed within a rack reach their peak power consumption simultaneously. However, this rarely happens even during peak periods under worst-case failure scenarios due to the workload heterogeneity [27]. To illustrate this, Figure 9 presents the system power consumption of three key workloads over one week. The data shows that peak power periods for individual workloads typically do not overlap, enabling opportunities for power *oversubscription* and creating a "power buffer" that can be leveraged to optimize power allocation. This buffer cannot be identified during initial RPB planning, as only a limited number of services can be profiled and proxy data may have to be used in the absence of fleet-wide production traffic. To quantify the size of this power buffer, we introduce a metric called the inter-workload max correlation score, denoted as *CR* (Equation 2).

$$CR = \frac{\sum_{i=0}^{W} \max_{0 \le t < T} w_{i,t}}{\max_{0 \le t < T} \sum_{i=0}^{W} w_{i,t}} \tag{2}$$

The numerator sums the worst-case power consumption of each workload (w) over a given time window ($0 \le t < T$), without considering when these peaks occur. The denominator captures the maximum total power drawn by all workloads at any single point in time within that window. Because peak power demands of different workloads usually occur at different times, the *CR* is typically greater than one. A higher *CR* reflects a larger "power buffer", introducing opportunities for power oversubscription without violating power constraints. We refine the RPB[7] formula by including the *CR* as follows:

$$RPB = \frac{\sum_{w=0}^{W-1} (f_w \times p_w)}{CR} \times n + P_{\text{TOR}} \tag{3}$$

The refined RPB in Equation 3 incorporates the power oversubscription principles from Equation 1 and Equation 2. It first addresses spatial power oversubscription, which involves the distribution of power usage across various workloads operating on different servers (see Equation 1 and Figure 5b). Additionally, it accounts for temporal power oversubscription, considering the timing and occurrence of peak power consumption events (see Equation 2 and Figure 9).

The refined RPB methodology allows us to expand overall fleet capacity. As illustrated in Figure 10, we compare design power with both forecast RPB and refined RPB across various rack types. By using forecast RPB, the fleet can achieve a 13% power oversubscription compared to using only design power. Furthermore, the refined RPB enables safe power oversubscription by an additional 11% beyond the forecast power budget, across compute, storage, and AI racks. This increased power oversubscription unlocks additional datacenter capacity, so more racks can be installed within the constraints of the power devices (see Figure 3). As a result, this approach allows us to make better use of the existing power capacity within our current fleet and reduces the need to construct new datacenters in the future.

Notably, this methodology has proven robust even as the fleet's workload mix evolved over time [16, 58, 69, 79]. The anticipated growth of AI training and inference workloads would not fundamentally alter our power planning methodology, but would introduce new rack types (AI training and inference) with distinct power profiles. The core refined RPB methodology applies uniformly across compute, storage, and

[6] The weighted-average f_w in Equation 1 reflects the planned workload distribution for each rack type, enforced by scheduling systems such as SmoothOperator [27] that actively prevent "hot rack" scenarios. The target utilization used to derive p_w is set conservatively based on load-test peaks, and the refined RPB (Equation 3) further guards against worst-case temporal alignment through the *CR* metric.

[7] Since it relies on fleet-wide historical service load and power data, the decision making of the refined RPB is deferred until MP+1 year.

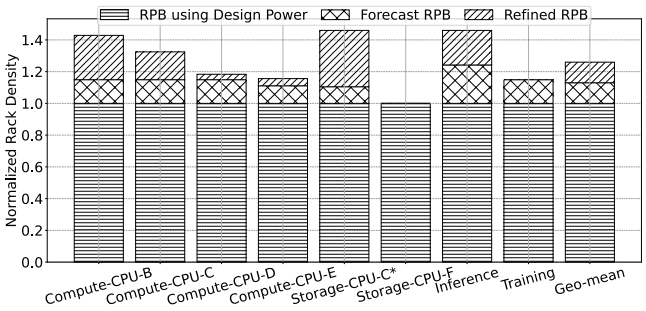


Figure 10: Lifecyle-aware RPB planning expands the fleet capacity. Using the forecast RPB increases rack density by 13% compared to design power (middle shade), while the refined RPB provides an additional 11% gain over the forecast RPB (top shade).

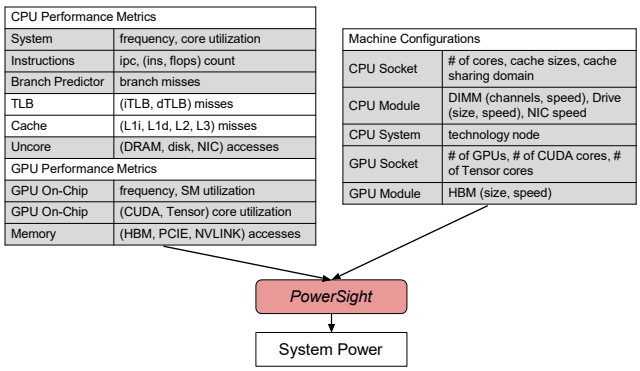


Figure 11: Performance metrics and machine configurations (input) to predict system power (output) for *PowerSight*. Highlights are representative input features (less than 0.1% accuracy loss compared using the full set).

AI racks. However, AI workloads introduce two main differences: (1) GPU-dominated racks may exhibit less temporal power variance than CPU-dominated racks as training jobs sustain high utilization for longer periods, resulting in lower CR values; and (2) the introduction of AI racks increased overall fleet heterogeneity, reinforcing the need for a cross-architecture power model that can generalize across compute, storage, and AI servers (§ 5).

Beyond power budgeting, the refined RPB approach is also compatible with workload placement and scheduling strategies, such as SmoothOperator [27] and Twine [70]. In addition to initial workload placement, our refined RPB approach considers the entire hardware lifecycle. As service loads fluctuate over time, we periodically adjust the RPB, even after mass production (MP + 1 year), to ensure continued alignment with evolving workload demands. Additionally, by taking advantage of rack heterogeneity (each power device can accommodate different types of racks, as illustrated in Figure 3), the refined RPB allows each power panel to support more racks while reducing power fragmentation.

4.2 System Power Forecasting is Necessary

While the refined RPB has enabled provisioning of approximately 20% more racks in our global fleet (based on the weighted average of power oversubscription and rack power footprints), it has a key limitation: it requires (a) fleet-wide workload information to determine f_w and (b) ground-truth power readings from sensors to determine p_w, which would not be available at early stages of hardware lifecycles.

Accurate RPB estimation is essential for effective datacenter power planning, even during the early stages of the hardware lifecycle. Unlike service-demand variance, which determines how many servers are needed, RPB accuracy determines how many racks fit within fixed power infrastructure – even a few percentage points of overestimation can translate to thousands of fewer racks fleet-wide, making it a critical input to datacenter capacity planning. Currently, *our power budget model requires power sensor data, which typically becomes available only in later phases (DVT/PVT).* However, the RPB must be determined much earlier to guide datacenter layout and support decisions such as whether to build new datacenters, a process that takes years. We introduce *PowerSight*, a system that forecasts power consumption without power sensors during the early hardware lifecycle stages (§ 5).

5 Future Power Budgeting Practice

Commercial datacenter planners need to make early tradeoffs between power and performance of future systems to mitigate the risk of selecting a power-inefficient configuration. In this section, we present our practice on power budgeting for future machine generations at earlier phases of hardware lifecycles, where power sensor readings are generally not available (before DVT/PVT phases).

As discussed in Table 1, machine configurations are finalized at pre-EVT (2–3 years before mass production), while engineering samples with calibrated power sensors typically become available only at DVT (6 months before mass production). To bridge this 12–18 month gap, we introduce *PowerSight* (§ 5.1), an ML-based power model to predict system power consumption of compute, storage, and AI servers (§ 5.2). *PowerSight can be easily deployed at scale, as it requires only machine configuration and performance counter data—both of which can be collected with minimal performance overhead using existing datacenter infrastructure [30, 31, 75], and are obtainable from early EVT samples before reliable power sensor data exists. PowerSight* can also facilitate informed decision-making during future system power planning (§ 5.3).

5.1 *PowerSight* Overview

5.1.1 *PowerSight* Prediction Target

The primary objective of *PowerSight* is to accurately predict machine system power (a continuous numerical value). We formulate this as a regression problem and train ML-based models for prediction.

5.1.2 *PowerSight* Inputs and Outputs

PowerSight takes both workload and machine characteristics as inputs (as our characterization results in § 3.5 show that the system power is determined by multiple factors) to predict system power consumption as its output. Figure 11 shows the different machine characteristics and performance events that *PowerSight* is built upon, all of which can be collected efficiently with minimal overhead using existing datacenter performance monitoring infrastructures [1, 30, 31, 75]. To enable large-scale deployment in production environments, we excluded kernel-level GPU performance counters, as profiling such events incurs significant runtime profiling overhead [26, 66, 78]. To model system power more accurately, we include DRAM/HBM, drives, and network accesses in addition to core performance metrics used in prior studies [82]. For machine configurations, we consider a comprehensive range of features to include not only the core architecture characteristics but also other non-core properties. Using these inputs, *PowerSight* employs an ML-based regression model to predict system power as its output.

Input Feature Selection: To improve accuracy while keeping training effort manageable (retraining or fine-tuning *PowerSight* can further enhance accuracy, more in § 5.2.5), we carefully choose the input features. Although incorporating more performance metrics into ML models can generally improve accuracy, prior research has shown that many performance counters are strongly correlated [12, 40, 54], indicating that not all features contribute equally to power prediction[8]. We apply principal component analysis-based clustering methods [38, 48, 54] to select a representative subset of input features (as highlighted in Figure 11). This approach achieves nearly the same accuracy – within 0.1% difference – as using the full set of input features.

5.1.3 *PowerSight* Machine Learning Formulation

To accurately predict system power using ML-based regression models, *PowerSight* requires selecting the most suitable regression model and training on high-quality datasets.

ML Model Type: We evaluate six common ML models used in performance and power modeling to predict system power consumption [3, 7, 83], including *Lasso Regression (Lasso), Ridge Regression (Ridge), SVM Regression (SVR), Decision Tree Regression (DT), Gradient Boosting Decision Tree Regression (GBDT), and MLP Regression (MLP)*. Our evaluation process consists of three stages. Initially, we evaluate each model by using it to predict system-level power for the same generation of hardware on which it was trained, providing a preliminary assessment of its accuracy. This initial evaluation helps us narrow down our selection to the top-performing models, specifically DT, GBDT, and MLP (§ 5.2.3). Next, we expand our evaluation to assess the performance of models across multiple generations for all compute, storage, and AI services. This step allows us to evaluate the robustness of *PowerSight* (§ 5.2.4). Finally, we conduct a rigorous test by evaluating the model on a machine generation it has never seen before. Based on this ultimate evaluation, we select MLP as the ML model for *PowerSight* (§ 5.2.5).

ML Model Training Dataset: We demonstrate the effectiveness of *PowerSight* by leveraging a vast dataset, collected by existing datacenter performance monitoring infrastructures [1, 30, 31, 75]. The dataset comprises real-world data from in-production services running on millions of diverse machines (more in § 5.2.2). This comprehensive dataset enables *PowerSight* to achieve good accuracy.

5.2 Training and Evaluating *PowerSight*

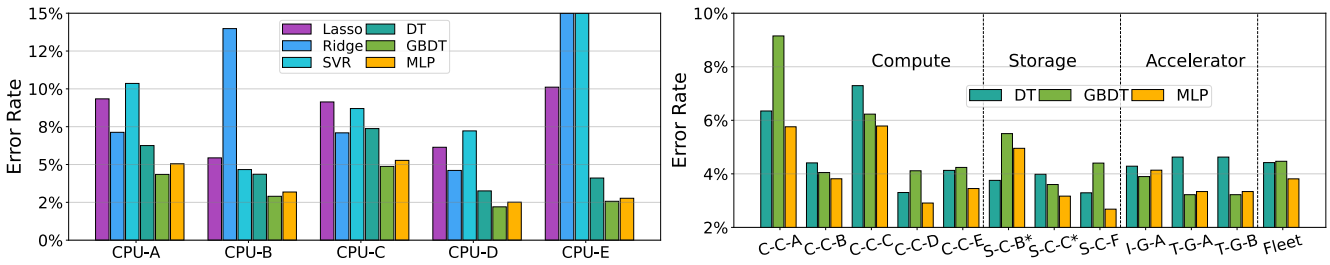
5.2.1 Training Methodology

To optimize the performance of each model, we use a *grid search* approach to select the most suitable model parameters that yield good accuracy. We train the models on live production data from actual measurements of performance counters and power sensors. At fleet scale, such data inevitably contains measurement noise–including temperature-related variance across server locations, manufacturing variability across silicon batches, and sensor calibration differences across BMC firmware versions. We mitigate these effects by aggregating across thousands of machines per configuration and applying standard data cleaning (e.g., removing outliers). As shown in § 5.2.2, the scale of our dataset provides inherent robustness against per-machine noise. To evaluate the accuracy of the models, we use data from separate days that was not used during training. This ensures that our evaluation is robust and unbiased, providing a true assessment of the predictive capability of each model.

5.2.2 *PowerSight* Training Dataset

The accuracy of a machine learning model depends heavily on the training dataset size. *PowerSight* leverages an extensive dataset consisting of millions of in-production data points collected from the entire datacenter fleet, which allows *PowerSight* to achieve high accuracy in predicting system power.

[8] There is also an opposite case where profiling and training overhead can be ignored while accuracy is the top priority. However, this is not the focus of this paper, as fleet-wide increasing even one counter would result in significant storage overhead when deploying *PowerSight* at scale.



(a) Single architecture.

(b) Multiple architectures.

Figure 12: System power prediction error rate (MAPE) of (a) trained and test on the same architecture; (b) train on all types of services and test on each architecture. DT, GBDT, and MLP achieve better accuracy than Lasso, Ridge, and SVM.

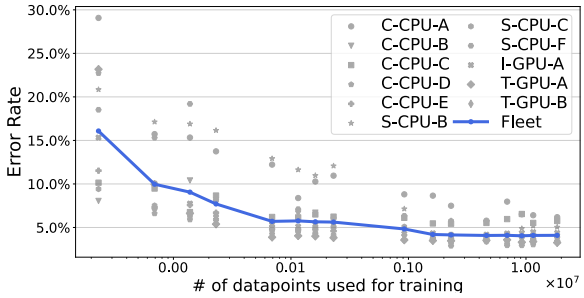


Figure 13: System power prediction MAPE (on testing dataset) when training *PowerSight* using different number of data points. The accuracy improves with more data points.

To demonstrate the effect of dataset size on model accuracy, we train *PowerSight* using varying numbers of data points from our entire fleet and present the results in Figure 13. Our analysis reveals that expanding the training dataset leads to significant improvements in the accuracy of *PowerSight*. However, when the model is trained with relatively small datasets (tens of thousands of data points), it yields a system power prediction error rate of over 10%. To attain optimal performance, we recommend training the model with at least several million data points. This ensures *PowerSight* uses sufficient data to learn effectively and make accurate predictions.

5.2.3 Single-Architecture Power Prediction

We begin with the simplest evaluation scenario: predicting system-level power for a single architecture, where the model is trained and tested on the same generation. Figure 12a shows the prediction error rate for all machine generations running compute services.

Among the six ML models, DT, GBDT, and MLP consistently outperform Lasso, Ridge, and SVM across all machine generations. DT, GBDT, and MLP achieve above 95% system power prediction accuracy for all machine generations. This is because prior works have shown that the power model may exhibit non-linear correlations between performance metrics [12] and linear regression models are particularly sensitive to the variability introduced by multiple machine configurations [9, 11]. Although SVM achieves good accuracy, its training time complexity increases quadratically with the number of samples, making it difficult to scale to large datasets with millions of data points (we had to subsample only 1% of the data to train SVM for this evaluation). Given that the accuracy of *PowerSight* benefits from using a large dataset with millions of in-production data points (more in § 5.2.2), the scalability limitations of SVM make it less suitable for this application. Therefore, DT, GBDT, and MLP are more effective for power prediction compared to other linear and support vector models, and we use these three models for our later analysis in this section.

5.2.4 Multiple-Architecture Power Prediction

As DT, GBDT, and MLP demonstrate better accuracy for single-architecture power prediction, we evaluate these three ML models using the fleet-wide data for all machine generations. We show the mean absolute percentage error (MAPE) of each model for all types of services (Figure 12b).

When applying the three models for all compute, storage, and AI services, all three models achieve good system power prediction accuracy (MLP achieves 96.19% accuracy, while DT and GBDT achieve 95.59% and 95.53% accuracy, respectively). This is consistent with their robustness to outliers, as also observed when training on individual machine generations. The tens of machine configurations (Table 2) introduce additional noise into the data, but neural network methods (MLP) can improve generalization by reducing overfitting compared to tree-based methodologies (DT and GBDT) [12, 19, 82]. In addition, the good accuracy of MLP can be attributed to the large dataset, comprising millions of data points across thousands of services and tens of machine configurations.

5.2.5 Predicting the power of a new system

To further assess the effectiveness of *PowerSight*, we evaluate and compare the performance of DT, GBDT, and MLP by predicting the system power of an inference platform

Table 4: Comparing ML models in predicting the power of existing architectures but new workload and new architectures unseen in the training set. MLP achieves the best accuracy.

Prediction Scenario	Model	Error Rate
Existing Architectures New Workload	DT	4.41%
	GBDT	4.47%
	MLP	3.81%
New Architecture New Workload	DT	9.39%
	GBDT	11.26%
	MLP	7.89%

equipped with CPU-H and GPU-B. Notably, these components were not included in our training dataset, which makes this evaluation a robust assessment of the ability of *PowerSight* to generalize to new scenarios. Specifically, CPU-H features significant upgrades compared to CPU-E, including more cores and increased memory bandwidth. Meanwhile, GPU-B also has different HBM configurations compared with GPU-B*.

We compare the performance of different ML models in two scenarios: predicting power consumption for existing architectures (top part of Table 4) and predicting power consumption for a new, unseen system not included in the training data (bottom part of Table 4)[9]. The results indicate that MLP achieves the highest accuracy in both cases, as it is capable of learning complex, nonlinear relationships between features when provided with enough data [60]. Given its robust performance in predicting system power across different machine generations, including those not seen during training, we ultimately select MLP as the ML model for *PowerSight*. Its ability to generalize well across various architectures and workloads makes it an ideal choice for our system power prediction framework.

5.3 Use Cases of *PowerSight*

In this section, we present case studies demonstrating how *PowerSight* supports future power budget projections in datacenters. *PowerSight* can be used to estimate the rack density (§ 5.3.1), estimate the rack power budget (§ 5.3.2), and identify optimal system configurations based on performance-per-Watt (Perf/W) metrics (§ 5.3.3) during the early phases of hardware lifecycles.

5.3.1 Rack Density

PowerSight can help estimate rack density during the pre-production phase, specifically *before the availability of power telemetry in the DVT/PVT stage.* Using *PowerSight* at this early stage allows datacenter planners to expedite rack demand projections and purchase planning, reducing the wait time by several months. We validate rack density prediction by training *PowerSight* on all machine generations except CPU-E and comparing its predictions against the current CPU-E planning. We use the vendor-labeled power for P_{TOR} (defined in § 2) as prior knowledge. Eventually, *PowerSight* can predict the rack design power with an average error rate (MAPE) of 1.7%, which results in an estimate of n with an error of 8.7%.

Although *PowerSight* may overestimate or underestimate power, which results in fewer or more machines per rack than ideal, these estimates are sufficiently accurate to guide early-stage power planning. When power is overestimated and fewer servers are deployed than necessary, it is relatively straightforward to add more servers to a rack during the mass production phase[10]. Conversely, if power is underestimated, datacenter planners can utilize techniques such as DVFS or other methods to further expand the planned capacity [52, 62].

5.3.2 Rack Power Budget

In addition to rack density, *PowerSight* can also be used to estimate the RPB early in the PVT phase. Before power telemetry is available, *PowerSight* can predict the p_w in the RPB formula $RPB = \sum_{w=0}^{W-1}(f_w \times p_w) \times n + P_{\text{TOR}}$ (details in § 4.1). We use the same P_{TOR} and n as when predicting the rack density using *PowerSight* (§ 5.3.1). We use the fraction of hosts for a given workload, f_w, based on the previous hardware generation as prior knowledge. With all information, *PowerSight* can predict p_w for each of the workloads and eventually predict the *RPB*. Finally, *PowerSight* achieves an *RPB* prediction with a 2.5% error, demonstrating its capability to be incorporated into the production flow. This level of accuracy provides datacenter planners with reliable guidance, such as determining whether new datacenters need to be constructed.

5.3.3 System Configuration

PowerSight can also be used within the same machine generation to identify a more efficient system configuration, e.g., searching for the optimal core frequency to maximize Perf/W.

Within a fixed power budget per rack, datacenter planners can use Dynamic Voltage and Frequency Scaling (DVFS) to optimize energy efficiency [34, 52, 62, 64]. One approach to achieving this is to sweep through all frequencies of a workload to identify the most efficient one. However, given the thousands of services running in a datacenter fleet, it is impractical to perform such a sweep study for every case.

[9] Results in Table 4 indicate a slight decrease in accuracy for newer systems, primarily because *PowerSight* is trained without vendor-specific hardware information. Once power sensor data from new platforms becomes available, we retrain the model to prevent such error accumulation.

[10] We adopt a conservative strategy by initially setting a higher RPB to accommodate unexpected increases. This approach is preferred because raising RPB later may require physically relocating many racks, which is operationally costly.

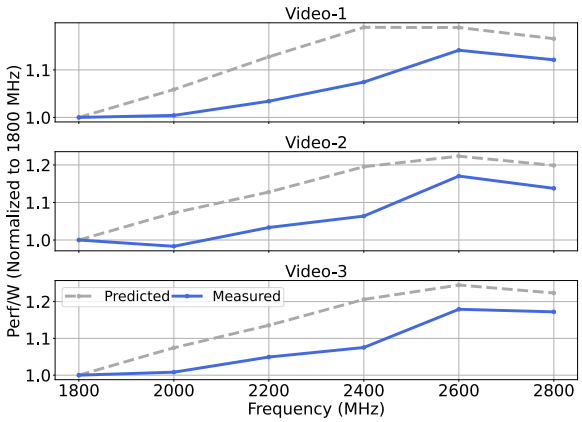


Figure 14: *PowerSight* can predict most Perf/W efficient frequency. While the absolute power has errors, *PowerSight* effectively captures the trend across different frequencies.

To address this challenge, *PowerSight* can predict the Perf/W of each workload based on its performance metrics at its current frequency. We verify this capability by using the *Video* workload as an example. As shown in Figure 14, real measurements indicate that 2.6 GHz is the most energy-efficient frequency for this workload. *PowerSight* predicts the most Perf/W efficient frequency as 2.4GHz-2.8GHz, closely aligning with the actual optimal value. In production environments, this prediction can serve as a starting point for fine-tuning frequency within a narrow range. This significantly streamlines the optimization process, enabling datacenter planners to efficiently identify the optimal frequency (or other system configurations) for each workload.

6 Related Work

Datacenter Power: Although prior studies have examined both machine-level [20, 28, 39, 41, 42, 62, 83] and rack-level [21, 55, 57, 59, 77, 80] power budgeting in datacenters, they largely focus on improving workload energy efficiency but overlook the impact of hardware and workload heterogeneity. Although some studies, such as SmoothOperator [27], have touched on workload heterogeneity, we are the first to introduce the concept of *hardware lifecycles* and a power budgeting methodology that can be adapted across different lifecycle phases.

Workload Coverage: Previous studies have primarily focused on power of specific workloads. For instance, works such as [23, 27, 34, 35, 64] studied general benchmarks and web services, while others [50, 65] targeted large language models. Our study encompasses thousands of AI and non-AI services, evaluated under live production traffic at scale across datacenters with millions of machines.

Power Modeling: Prior works have focused on predicting the power of either CPUs [17, 29, 37, 47, 49, 53, 71] or GPUs [6, 32, 45, 76]. Though a few have discussed the entire system power, these analyses were limited to either small-scale systems or existing machine generations [12, 19, 82]. Although using Running Average Power Limit (RAPL) counters is another common approach to estimate power, RAPL is not supported by all vendors and may omit components such as system voltage regulators [8]. *PowerSight* takes a more holistic approach by considering the power of the entire system, including memory, storage, accelerator, and other components. *Additionally, we are the first to demonstrate the real use cases of machine learning for power modeling by incorporating the context of hardware lifecycles in datacenters, particularly when power sensors are unavailable.*

Our work is distinct from prior studies: we are the first to (a) present a comprehensive power characterization study across machine generations, showing how these insights can guide power planning at scale; (b) introduce the concept of hardware lifecycle in datacenters, showing how power planning is tailored to the specific characteristics of the heterogeneous hardware and workload in each phase; and (c) employ ML-based models for early-stage datacenter power budgeting before power sensor data is available, and provide case studies on future power budget strategies.

7 Conclusion

As commercial hyperscale datacenters continue to grow, power planning must account for multiple hardware generations and their lifecycles. In this paper, we first present a comprehensive power characterization on hyperscale workloads, based on measurements collected from thousands of services using live production traffic across generations of servers in Meta's datacenters. We present our hardware *lifecycle-aware* datacenter power planning strategies leveraging characterization insights, enabling us to achieve large-scale power oversubscription. We also introduce *PowerSight*, which employs ML-based models to predict system power consumption for early-stage datacenter power budgeting, even before power sensor data is available. Our ultimate goal is to advance sustainable hyperscale datacenters that meet growing computing demands while minimizing environmental impact.

Acknowledgment

The authors would like to thank the anonymous reviewers for their constructive feedback. We would like to thank the current and former Meta engineers who contributed to this project, especially Vjekoslav Svilan. We also thank Kalyan Subramanian, Shobhit Kanaujia, and Chunqiang (CQ) Tang for the management support.

References

[1] Dynolog: a performance monitoring daemon for heterogeneous cpu-gpu systems, 2025.

[2] Open compute project, 2025.

[3] Cristinel Ababei and Milad Ghorbani Moghaddam. A survey of prediction and classification techniques in multicore processor systems. *IEEE Transactions on Parallel and Distributed Systems*, 30(5):1184–1200, 2018.

[4] Bilge Acun, Benjamin Lee, Fiodar Kazhamiaka, Kiwan Maeng, Udit Gupta, Manoj Chakkaravarthy, David Brooks, and Carole-Jean Wu. Carbon explorer: A holistic framework for designing carbon aware datacenters. *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, 2022.

[5] Keith Adams, Jason Evans, Bertrand Maher, Guilherme Ottoni, Andrew Paroski, Brett Simmers, Edwin Smith, and Owen Yamauchi. The hiphop virtual machine. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications*, pages 777–790, 2014.

[6] Gargi Alavani, Jineet Desai, Snehanshu Saha, and Santonu Sarkar. Program analysis and machine learning–based approach to predict power consumption of cuda kernel. *ACM Transactions on Modeling and Performance Evaluation of Computing Systems*, 8(4):1–24, 2023.

[7] Erika S Alcorta and Andreas Gerstlauer. Learning-based workload phase classification and prediction using performance monitoring counters. In *2021 ACM/IEEE 3rd Workshop on Machine Learning for CAD (MLCAD)*, pages 1–6. IEEE, 2021.

[8] Lukas Alt, Anara Kozhokanova, Thomas Ilsche, Christian Terboven, and Matthias S Mueller. An experimental setup to evaluate rapl energy counters for heterogeneous memory. In *Proceedings of the 15th ACM/SPEC International Conference on Performance Engineering*, pages 71–82, 2024.

[9] David F Andrews. A robust method for multiple linear regression. *Technometrics*, 16(4):523–531, 1974.

[10] Ricardo Bianchini, Christian Belady, and Anand Sivasubramaniam. Datacenter power and energy management: past, present, and future. *IEEE Micro*, 2024.

[11] Nicholas H Bingham and John M Fry. *Regression: Linear models in statistics*. Springer Science & Business Media, 2010.

[12] William Lloyd Bircher and Lizy K John. Complete system power estimation using processor performance events. *IEEE Transactions on Computers*, 61(4):563–577, 2011.

[13] James Bucek, Klaus-Dieter Lange, and Jóakim v. Kistowski. Spec cpu2017: Next-generation compute benchmark. In *Companion of the 2018 ACM/SPEC International Conference on Performance Engineering*, pages 41–42, 2018.

[14] Zhichao Cao, Siying Dong, Sagar Vemuri, and David HC Du. Characterizing, modeling, and benchmarking {RocksDB}{Key-Value} workloads at facebook. In *18th USENIX Conference on File and Storage Technologies (FAST 20)*, pages 209–223, 2020.

[15] Arnab Choudhury, Yang Wang, Tuomas Pelkonen, Kutta Srinivasan, Abha Jain, Shenghao Lin, Delia David, Siavash Soleimanifard, Michael Chen, Abhishek Yadav, et al. {MAST}: Global scheduling of {ML} training across {Geo-Distributed} datacenters at hyperscale. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, pages 563–580, 2024.

[16] Mike Chow, Yang Wang, William Wang, Ayichew Hailu, Rohan Bopardikar, Bin Zhang, Jialiang Qu, David Meisner, Santosh Sonawane, Yunqi Zhang, et al. {ServiceLab}: Preventing tiny performance regressions at hyperscale through {Pre-Production} testing. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, pages 545–562, 2024.

[17] Gilberto Contreras and Margaret Martonosi. Power prediction for intel xscale® processors using performance monitoring unit events. In *Proceedings of the 2005 international symposium on Low power electronics and design*, pages 221–226, 2005.

[18] Standard Performance Evaluation Corporation. Spec cpu2017 benchmark suite, 2017.

[19] Georges DA Costa, Jean-Marc Pierson, and Leandro Fontoura-Cupertino. Effectiveness of neural networks for power modeling for cloud and hpc: It's worth it! *ACM Transactions on Modeling and Performance Evaluation of Computing Systems (TOMPECS)*, 5(3):1–36, 2020.

[20] Bryan Donyanavard, Tiago Mück, Amir M Rahmani, Nikil Dutt, Armin Sadighi, Florian Maurer, and Andreas Herkersdorf. Sosa: Self-optimizing learning with self-adaptive control for hierarchical system-on-chip management. In *Proceedings of the 52nd annual IEEE/ACM international symposium on microarchitecture*, pages 685–698, 2019.

[21] Xiaobo Fan, Wolf-Dietrich Weber, and Luiz Andre Barroso. Power provisioning for a warehouse-sized computer. *ACM SIGARCH computer architecture news*, 35(2):13–23, 2007.

[22] Abraham Gonzalez, Aasheesh Kolli, Samira Khan, Sihang Liu, Vidushi Dadu, Sagar Karandikar, Jichuan Chang, Krste Asanovic, and Parthasarathy Ranganathan. Profiling hyperscale big data processing. In *Proceedings of the 50th Annual International Symposium on Computer Architecture*, pages 1–16, 2023.

[23] Sriram Govindan, Jeonghwan Choi, Bhuvan Urgaonkar, Anand Sivasubramaniam, and Andrea Baldini. Statistical profiling-based techniques for effective power provisioning in data centers. In *Proceedings of the 4th ACM European conference on Computer systems*, pages 317–330, 2009.

[24] Udit Gupta, Mariam Elgamal, Gage Hills, Gu-Yeon Wei, Hsien-Hsin S. Lee, David Brooks, and Carole-Jean Wu. Act: designing sustainable computer systems with an architectural carbon modeling tool. *Proceedings of the 49th Annual International Symposium on Computer Architecture*, 2022.

[25] Ori Hadary, Luke Marshall, Ishai Menache, Abhisek Pan, Esaias E Greeff, David Dion, Star Dorminey, Shailesh Joshi, Yang Chen, Mark Russinovich, et al. Protean:{VM} allocation service at scale. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 845–861, 2020.

[26] Yueming Hao, Nikhil Jain, Rob Van der Wijngaart, Nirmal Saxena, Yuanbo Fan, and Xu Liu. Drgpu: A top-down profiler for gpu applications. In *Proceedings of the 2023 ACM/SPEC International Conference on Performance Engineering*, pages 43–53, 2023.

[27] Chang-Hong Hsu, Qingyuan Deng, Jason Mars, and Lingjia Tang. Smoothoperator: Reducing power fragmentation and improving power utilization in large-scale datacenters. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 535–548, 2018.

[28] Chang-Hong Hsu, Yunqi Zhang, Michael A Laurenzano, David Meisner, Thomas Wenisch, Jason Mars, Lingjia Tang, and Ronald G Dreslinski. Adrenaline: Pinpointing and reining in tail queries with quick voltage boosting. In *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*, pages 271–282. IEEE, 2015.

[29] Leila Ismail and Huned Materwala. Computing server power modeling in a data center: Survey, taxonomy, and performance evaluation. *ACM Comput. Surv.*, 53(3), June 2020.

[30] Akanksha Jain, Hannah Lin, Carlos Villavieja, Baris Kasikci, Chris Kennelly, Milad Hashemi, and Parthasarathy Ranganathan. Limoncello: Prefetchers for scale. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, pages 577–590, 2024.

[31] Saurabh Jha, Archit Patke, Jim Brandt, Ann Gentile, Benjamin Lim, Mike Showerman, Greg Bauer, Larry Kaplan, Zbigniew Kalbarczyk, William Kramer, et al. Measuring congestion in {High-Performance} datacenter interconnects. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 37–57, 2020.

[32] Vijay Kandiah, Scott Peverelle, Mahmoud Khairy, Junrui Pan, Amogh Manjunath, Timothy G. Rogers, Tor M. Aamodt, and Nikos Hardavellas. Accelwattch: A power modeling framework for modern gpus. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '21, page 738–753, New York, NY, USA, 2021. Association for Computing Machinery.

[33] Svilen Kanev, Juan Pablo Darago, Kim Hazelwood, Parthasarathy Ranganathan, Tipp Moseley, Gu-Yeon Wei, and David Brooks. Profiling a warehouse-scale computer. In *Proceedings of the 42nd annual international symposium on computer architecture*, pages 158–169, 2015.

[34] Svilen Kanev, Kim Hazelwood, Gu-Yeon Wei, and David Brooks. Tradeoffs between power management and tail latency in warehouse-scale applications. In *2014 IEEE International Symposium on Workload Characterization (IISWC)*, pages 31–40. IEEE, 2014.

[35] Harshad Kasture, Davide B Bartolini, Nathan Beckmann, and Daniel Sanchez. Rubik: Fast analytical power management for latency-critical systems. In *Proceedings of the 48th International Symposium on Microarchitecture*, pages 598–610, 2015.

[36] Kwiwook Kim and Myeong-jae Park. Present and future, challenges of high bandwith memory (hbm). In *2024 IEEE International Memory Workshop (IMW)*, pages 1–4. IEEE, 2024.

[37] Benjamin C Lee and David M Brooks. Accurate and efficient regression modeling for microarchitectural performance and power prediction. *ACM SIGOPS operating systems review*, 40(5):185–194, 2006.

[38] Ruihao Li, Andrew Jacob, Neeraja J Yadwadkar, and Lizy K John. Spec cpu2026: Characterization, representativeness, and cross-suite comparison. *arXiv preprint arXiv:2605.03713*, 2026.

[39] Shaohong Li, Xi Wang, Faria Kalim, Xiao Zhang, Sangeetha Abdu Jyothi, Karan Grover, Vasileios Kontorinis, Nina Narodytska, Owolabi Legunsen, Sreekumar Kodakara, et al. Thunderbolt:{Throughput-Optimized},{Quality-of-Service-Aware} power capping at scale. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 1241–1255, 2020.

[40] Tao Li and Lizy Kurian John. Run-time modeling and estimation of operating system power consumption. In *Proceedings of the 2003 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, pages 160–171, 2003.

[41] David Lo, Liqun Cheng, Rama Govindaraju, Luiz André Barroso, and Christos Kozyrakis. Towards energy proportionality for large-scale latency-critical workloads. *ACM SIGARCH Computer Architecture News*, 42(3):301–312, 2014.

[42] David Lo, Liqun Cheng, Rama Govindaraju, Parthasarathy Ranganathan, and Christos Kozyrakis. Heracles: Improving resource efficiency at scale. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture*, pages 450–462, 2015.

[43] Suyash Mahar, Hao Wang, Wei Shu, and Abhishek Dhanotia. Workload behavior driven memory subsystem design for hyperscale. *arXiv preprint arXiv:2303.08396*, 2023.

[44] Eric Masanet, Arman Shehabi, Nuoa Lei, Sarah Josephine Smith, and Jonathan Koomey. Recalibrating global data center energy-use estimates. *Science*, 367:984 – 986, 2020.

[45] Diksha Moolchandani, Anshul Kumar, and Smruti R Sarangi. Performance and power prediction for concurrent execution on gpus. *ACM Transactions on Architecture and Code Optimization (TACO)*, 19(3):1–27, 2022.

[46] Guilherme Ottoni. Hhvm jit: A profile-guided, region-based compiler for php and hack. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 151–165, 2018.

[47] Kenneth O'brien, Ilia Pietri, Ravi Reddy, Alexey Lastovetsky, and Rizos Sakellariou. A survey of power and energy predictive models in hpc systems and applications. *ACM Comput. Surv.*, 50(3), June 2017.

[48] Reena Panda, Shuang Song, Joseph Dean, and Lizy K John. Wait of a decade: Did spec cpu 2017 broaden the performance horizon? In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 271–282. IEEE, 2018.

[49] George Papadimitriou, Athanasios Chatzidimitriou, and Dimitris Gizopoulos. Adaptive voltage/frequency scaling and core allocation for balanced energy and performance on multicore cpus. In *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 133–146, 2019.

[50] Pratyush Patel, Esha Choukse, Chaojie Zhang, Íñigo Goiri, Brijesh Warrier, Nithish Mahalingam, and Ricardo Bianchini. Characterizing power management opportunities for llms in the cloud. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, pages 207–222, 2024.

[51] Pratyush Patel, Esha Choukse, Chaojie Zhang, Aashaka Shah, Íñigo Goiri, Saeed Maleki, and Ricardo Bianchini. Splitwise: Efficient generative llm inference using phase splitting. In *2024 ACM/IEEE 51st Annual International Symposium on Computer Architecture (ISCA)*, pages 118–132. IEEE, 2024.

[52] Leonardo Piga, Iyswarya Narayanan, Aditya Sundarrajan, Matt Skach, Qingyuan Deng, Biswadip Maity, Manoj Chakkaravarthy, Alison Huang, Abhishek Dhanotia, and Parth Malani. Expanding datacenter capacity with dvfs boosting: A safe and scalable deployment experience. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1*, pages 150–165, 2024.

[53] Mihai Pricopi, Thannirmalai Somu Muthukaruppan, Vanchinathan Venkataramani, Tulika Mitra, and Sanjay Vishin. Power-performance modeling on asymmetric multi-cores. In *2013 International Conference on Compilers, Architecture and Synthesis for Embedded Systems (CASES)*, pages 1–10, 2013.

[54] Aleksandar Prokopec, Andrea Rosà, David Leopoldseder, Gilles Duboscq, Petr Tůma, Martin Studener, Lubomír Bulej, Yudi Zheng, Alex Villazón, Doug Simon, et al. Renaissance: Benchmarking suite for parallel applications on the jvm. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 31–47, 2019.

[55] Ramya Raghavendra, Parthasarathy Ranganathan, Vanish Talwar, Zhikui Wang, and Xiaoyun Zhu. No" power" struggles: coordinated multi-level power management for the data center. In *Proceedings of the 13th international conference on Architectural support for programming languages and operating systems*, pages 48–59, 2008.

[56] Parthasarathy Ranganathan and Urs Holzle. Twenty five years of warehouse-scale computing. *IEEE Micro*, 2024.

[57] Parthasarathy Ranganathan, Phil Leech, David Irwin, and Jeffrey Chase. Ensemble-level power management for dense blade servers. *ACM SIGARCH computer architecture news*, 34(2):66–77, 2006.

[58] Alireza Sahraei, Soteris Demetriou, Amirali Sobhgol, Haoran Zhang, Abhigna Nagaraja, Neeraj Pathak, Girish Joshi, Carla Souza, Bo Huang, Wyatt Cook, et al. Xfaas: Hyperscale and low cost serverless functions at meta. In *Proceedings of the 29th Symposium on Operating Systems Principles*, pages 231–246, 2023.

[59] Varun Sakalkar, Vasileios Kontorinis, David Landhuis, Shaohong Li, Darren De Ronde, Thomas Blooming, Anand Ramesh, James Kennedy, Christopher Malone, Jimmy Clidaras, et al. Data center power oversubscription with a medium voltage power plane and priority-aware capping. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 497–511, 2020.

[60] Madan Somvanshi, Pranjali Chavan, Shital Tambade, and SV Shinde. A review of machine learning techniques using decision tree and support vector machine. In *2016 international conference on computing communication control and automation (ICCUBEA)*, pages 1–7. IEEE, 2016.

[61] Akshitha Sriraman and Abhishek Dhanotia. Accelerometer: Understanding acceleration opportunities for data center overheads at hyperscale. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 733–750, 2020.

[62] Akshitha Sriraman, Abhishek Dhanotia, and Thomas F Wenisch. Softsku: Optimizing server architectures for microservice diversity@ scale. In *Proceedings of the 46th International Symposium on Computer Architecture*, pages 513–526, 2019.

[63] Jovan Stojkovic, Chunao Liu, Muhammad Shahbaz, and Josep Torrellas. μmanycore: A cloud-native cpu for tail at scale. In *Proceedings of the 50th Annual International Symposium on Computer Architecture*, pages 1–15, 2023.

[64] Jovan Stojkovic, Pulkit A Misra, Íñigo Goiri, Sam Whitlock, Esha Choukse, Mayukh Das, Chetan Bansal, Jason Lee, Zoey Sun, Haoran Qiu, et al. Smartoclock: Workload-and risk-aware overclocking in the cloud. In *2024 ACM/IEEE 51st Annual International Symposium on Computer Architecture (ISCA)*, pages 437–451. IEEE, 2024.

[65] Jovan Stojkovic, Chaojie Zhang, Íñigo Goiri, Josep Torrellas, and Esha Choukse. Dynamollm: Designing llm inference clusters for performance and energy efficiency. In *2025 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 1348–1362. IEEE, 2025.

[66] Foteini Strati, Xianzhe Ma, and Ana Klimovic. Orion: Interference-aware, fine-grained gpu sharing for ml applications. In *Proceedings of the Nineteenth European Conference on Computer Systems*, pages 1075–1092, 2024.

[67] Wei Su, Abhishek Dhanotia, Carlos Torres, Jayneel Gandhi, Neha Gholkar, Shobhit Kanaujia, Maxim Naumov, Kalyan Subramanian, Valentin Andrei, Yifan Yuan, et al. Dcperf: An open-source, battle-tested performance benchmark suite for datacenter workloads. In *Proceedings of the 52nd Annual International Symposium on Computer Architecture*, pages 1717–1730, 2025.

[68] Chunqiang Tang. Meta's hyperscale infrastructure: Overview and insights. *Communications of the ACM*, 2025.

[69] Chunqiang Tang, Thawan Kooburat, Pradeep Venkatachalam, Akshay Chander, Zhe Wen, Aravind Narayanan, Patrick Dowell, and Robert Karl. Holistic configuration management at facebook. In *Proceedings of the 25th symposium on operating systems principles*, pages 328–343, 2015.

[70] Chunqiang Tang, Kenny Yu, Kaushik Veeraraghavan, Jonathan Kaldor, Scott Michelson, Thawan Kooburat, Aravind Anbudurai, Matthew Clark, Kabir Gogia, Long Cheng, et al. Twine: A unified cluster management system for shared infrastructure. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 787–803, 2020.

[71] Matthew J. Walker, Stephan Diestelhorst, Andreas Hansson, Anup K. Das, Sheng Yang, Bashir M. Al-Hashimi, and Geoff V. Merrett. Accurate and stable run-time power modeling for mobile and embedded cpus. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 36(1):106–119, 2017.

[72] Jaylen Wang, Daniel S Berger, Fiodar Kazhamiaka, Celine Irvene, Chaojie Zhang, Esha Choukse, Kali Frost, Rodrigo Fonseca, Brijesh Warrier, Chetan Bansal, et al. Designing cloud servers for lower carbon. In *2024 ACM/IEEE 51st Annual International Symposium on Computer Architecture (ISCA)*, pages 452–470. IEEE, 2024.

[73] Qizhen Weng, Wencong Xiao, Yinghao Yu, Wei Wang, Cheng Wang, Jian He, Yong Li, Liping Zhang, Wei Lin, and Yu Ding. {MLaaS} in the wild: Workload analysis and scheduling in {Large-Scale} heterogeneous {GPU} clusters. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, pages 945–960, 2022.

[74] Qizhen Weng, Lingyun Yang, Yinghao Yu, Wei Wang, Xiaochuan Tang, Guodong Yang, and Liping Zhang. Beware of fragmentation: Scheduling {GPU-Sharing} workloads with fragmentation gradient descent. In *2023 USENIX Annual Technical Conference (USENIX ATC 23)*, pages 995–1008, 2023.

[75] Lukasz Wesolowski, Bilge Acun, Valentin Andrei, Adnan Aziz, Gisle Dankel, Christopher Gregg, Xiaoqiao Meng, Cyril Meurillon, Denis Sheahan, Lei Tian, et al. Datacenter-scale analysis and optimization of gpu machine learning workloads. *IEEE Micro*, 41(5):101–112, 2021.

[76] Gene Wu, Joseph L Greathouse, Alexander Lyashevsky, Nuwan Jayasena, and Derek Chiou. Gpgpu performance and power estimation using machine learning. In *2015 IEEE 21st international symposium on high performance computer architecture (HPCA)*, pages 564–576. IEEE, 2015.

[77] Qiang Wu, Qingyuan Deng, Lakshmi Ganesh, Chang-Hong Hsu, Yun Jin, Sanjeev Kumar, Bin Li, Justin Meza, and Yee Jiun Song. Dynamo: Facebook's data center-wide power management system. *ACM SIGARCH Computer Architecture News*, 44(3):469–480, 2016.

[78] Siyu Wu, Hailong Yang, Xin You, Ruihao Gong, Yi Liu, Zhongzhi Luan, and Depei Qian. Proof: A comprehensive hierarchical profiling framework for deep neural networks with roofline analysis. In *Proceedings of the 53rd International Conference on Parallel Processing*, pages 822–832, 2024.

[79] Dong Young Yoon, Yang Wang, Miao Yu, Elvis Huang, Juan Ignacio Jones, Abhinay Kukkadapu, Osman Kocas, Jonathan Wiepert, Kapil Goenka, Sherry Chen, et al. Fbdetect: Catching tiny performance regressions at hyperscale through in-production monitoring. In *Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles*, pages 522–540, 2024.

[80] Chaojie Zhang, Alok Gautam Kumbhare, Ioannis Manousakis, Deli Zhang, Pulkit A Misra, Rod Assis, Kyle Woolcock, Nithish Mahalingam, Brijesh Warrier, David Gauthier, et al. Flex: High-availability datacenters with zero reserved power. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*, pages 319–332. IEEE, 2021.

[81] Kaiyang Zhao, Kaiwen Xue, Ziqi Wang, Dan Schatzberg, Leon Yang, Antonis Manousis, Johannes Weiner, Rik Van Riel, Bikash Sharma, Chunqiang Tang, et al. Contiguitas: The pursuit of physical memory contiguity in datacenters. In *Proceedings of the 50th Annual International Symposium on Computer Architecture*, pages 1–15, 2023.

[82] Xinnian Zheng, Lizy K John, and Andreas Gerstlauer. Accurate phase-level cross-platform power and performance estimation. In *Proceedings of the 53rd Annual Design Automation Conference*, pages 1–6, 2016.

[83] An Zou, Karthik Garimella, Benjamin Lee, Christopher Gill, and Xuan Zhang. F-lemma: Fast learning-based energy management for multi-/many-core processors. In *Proceedings of the 2020 ACM/IEEE Workshop on Machine Learning for CAD*, pages 43–48, 2020.

[84] Mark Zuckerberg, Ruchi Sanghvi, Andrew Bosworth, Chris Cox, Aaron Sittig, Chris Hughes, Katie Geminder, and Dan Corson. Dynamically providing a news feed about a user of a social network, February 23 2010. US Patent 7,669,123.

ARTIFACT EVALUATED
USENIX
AVAILABLE

Kareus: Joint Reduction of Dynamic and Static Energy in Large Model Training

Ruofan Wu Jae-Won Chung Mosharaf Chowdhury
University of Michigan

Abstract

The computing demand of AI is growing at an unprecedented rate, but energy supply is not keeping pace. As a result, energy has become an expensive and contended resource that requires explicit management and optimization. Although recent works have made significant progress in large model training optimization, they focus on optimizing either dynamic or static energy consumption.

We find that fine-grained kernel scheduling and frequency scaling *jointly* and *interdependently* impact both dynamic and static energy consumption. Based on this finding, we design Kareus, a training system that pushes the time–energy tradeoff frontier by optimizing both aspects. Kareus decomposes the intractable joint optimization problem into local, partition-based subproblems. It then uses a multi-pass multi-objective optimization algorithm to find execution schedules that push the time–energy tradeoff frontier. Compared to the state of the art, Kareus reduces training energy by up to 28.3% at the same training time, or reduces training time by up to 27.5% at the same energy consumption.[1]

1 Introduction

Today, energy is the ultimate bottleneck for scaling AI [7, 31, 42]. The energy demand of training large models and serving them to millions is growing at an unprecedented rate [21, 33, 37], with projections indicating that by 2035, nearly 10% of US electricity demand could be from datacenters [31]. However, procuring energy at scale is slow, e.g., three years for natural gas and five to ten years for nuclear [46]. This mismatch makes energy an expensive and contended resource that must be explicitly budgeted, managed, and allocated with efficient systems and optimization methods.

In this context, large model training is a key target for optimization; a single training run can consume enough energy to power more than 24,000 average US households for a month [1, 33]. To understand the current state of large model training optimizations, we analyze them through the lens of *dynamic* energy (consumed by actual work) and *static* energy (consumed at all times regardless of work) consumption of GPUs (§2). Practically, dynamic energy can be reduced by lowering GPU frequency or activity, while static energy can be reduced by shortening iteration time. Perseus [15] reduces dynamic energy by scaling the frequency of computations off the critical path, but does not reschedule kernels. Recent works in fine-grained kernel scheduling [18, 20, 48, 49, 57] reduce static energy by overlapping computation and communication, but they ignore dynamic energy and frequency scaling. Therefore, we ask: can we combine both approaches to *jointly* reduce dynamic and static energy?

To answer this question, we study how *execution schedules* affect the time and energy consumption of large model training (§3). Any large model training iteration boils down to computation and communication kernels launched on the GPU. We define an *execution schedule* as the combination of three factors: (1) the *timing* of when communication kernels are launched within a sequence of computation kernels, (2) the *number of GPU Streaming Multiprocessors (SMs)* allocated to communication kernels, and (3) the *GPU frequency*.

We demonstrate that these factors *jointly* and *interdependently* determine time and energy consumption. Different execution schedules lead to different time and energy consumption—varying by as much as 3.29×—even when the total amount of work remains the same. The optimal schedule is achieved by carefully balancing the resource consumption of computation and communication kernels, which depends on the interplay of all three factors. Crucially, *the best SM allocation and kernel launch timing depend on GPU frequency*; even for the same sequence of kernels, we cannot use the same SM allocation and launch timing at different frequencies. Intuitively, frequency changes how overlapping kernels contend for resources: lowering frequency slows computation while leaving memory and communication bandwidth largely unchanged, thereby reshaping the optimal SM allocation and kernel launch timing. Existing solutions for kernel scheduling [18, 20, 48, 49, 57] and frequency scaling [15] optimize disjoint subsets of these factors, and combining them naively is suboptimal for pushing the time–energy tradeoff frontier.

Based on these observations, we present *Kareus*, an energy-efficient large model training system that automatically finds the best execution schedule by jointly optimizing SM allocation, launch timing, and GPU frequency (§4). For all kernels in the training iteration, it identifies (1) the right number of SMs for communication kernels, and (2) the right amount of computation–communication overlap, both in a frequency-specific manner. A collateral benefit is that executions on the critical path also speeds up due to improved overlap, reducing training time and thus shifting the entire time–energy tradeoff frontier toward the origin.

[1] Kareus is open-source at https://github.com/ml-energy/kareus.

Unfortunately, jointly optimizing all three factors via exhaustive search is impractical. The search space is prohibitively large to profile each configuration. Worse, profiling heats up the GPU, and without sufficient cooling intervals, one configuration's measurement affects subsequent ones.

Kareus addresses these challenges by introducing the *partitioned overlap* execution model. The execution graphs of forward and backward passes are divided into fixed partitions, decomposing the global optimization problem into local subproblems. Partitioned overlap generalizes recent *nanobatching* techniques [18,20,48,49,57] and adds fine-grained control over all three execution schedule factors, enabling precise control over the overlap between each communication kernel and its surrounding computation. For each partition, Kareus uses a multi-pass multi-objective optimization algorithm to identify candidates on the time–energy tradeoff frontier. It then hierarchically composes local (partition-level) frontiers into a global (iteration-level) one for the entire training iteration.

We implement Kareus by integrating with Perseus [15] and Megatron-LM [35] (§5). Kareus's optimizer is informed by time and energy profiling results from our *thermally stable* profiler, which minimizes thermal interference between different configurations during profiling. Once the best execution schedule is identified, Kareus interacts with Perseus and Megatron-LM to realize the plan throughout the entire training execution.

We evaluate Kareus across 14 representative workloads, including real testbed training on Llama 3.2 3B and Qwen 3 1.7B, and large-scale emulation on Llama 3.3 70B (§6). Compared to the state-of-the-art Perseus, Kareus reduces energy by up to 28.3% under the same time budget, or reduces time by up to 27.5% under the same energy budget.

In summary, we make the following contributions:

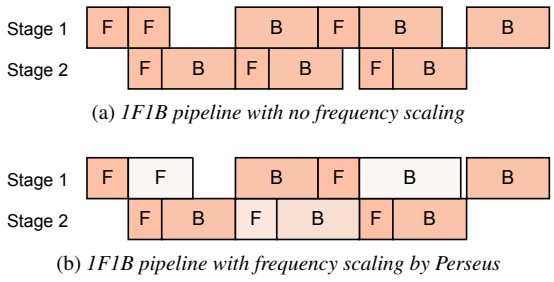
- We show that SM allocation, launch timing, and GPU frequency jointly determine time and energy consumption, and that existing solutions optimizing subsets of these factors and naive combinations thereof are suboptimal.
- We design Kareus around the *partitioned overlap* execution model, decomposing the global optimization problem into tractable local subproblems.
- We evaluate Kareus on large model training workloads, demonstrating significant time and energy reduction compared to the state-of-the-art.

2 Background

We begin by providing a brief background on GPU power consumption (§2.1). Then, we compare existing training systems in terms of their goals and techniques (§2.2), and discuss implications for training iteration time and energy (§2.3).

2.1 GPU Power Consumption

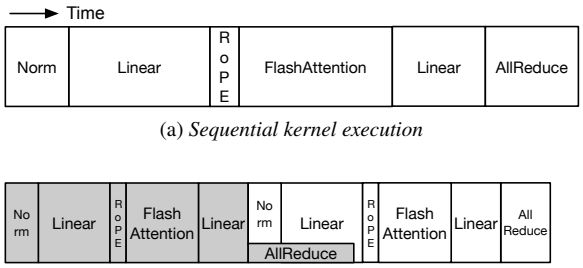
A GPU's power consumption can be divided into two components: *dynamic power and static power* [15, 32, 41]. Dynamic power is the power consumed by the chip's *compute and memory activity*; it is proportional to core frequency and the square of voltage. In contrast, static power is consumed by *all parts of the chip* at all times regardless of utilization or activity.



(a) *1F1B pipeline with no frequency scaling*

(b) *1F1B pipeline with frequency scaling by Perseus*

Figure 1: ***Existing training systems running the 1F1B pipeline schedule [35]. Redder colors indicate higher GPU frequency and power draw. (a) Megatron-LM [35] does not perform any frequency scaling, whereas (b) Perseus [15] scales GPU frequencies of non-critical forward and backward computations to reduce energy consumption while maintaining the same latency.***



(a) *Sequential kernel execution*

(b) *Nanobatching*

Figure 2: ***Transformer [47] Attention layer with tensor parallelism run with different execution models. (a) The sequential kernel execution model only runs one kernel at a time strictly following data dependencies. (b) Nanobatching [18, 20, 48, 49, 57] splits a pipeline microbatch into two nanobatches (illustrated with different colors) and staggers their execution, creating opportunities to overlap communication and computation.***

2.2 Existing Training Systems

Large model training relies on multi-GPU parallelism, which distributes model weights and computation while introducing communication. On each GPU, kernel scheduling determines how computation and communication are executed. Figure 1a illustrates Megatron-LM's [35] implementation of the 1F1B pipeline schedule. Each pipeline microbatch (boxes labeled F and B) contains multiple Transformer [47] blocks, each with an Attention layer and a feed-forward (MLP) layer. Figure 2a shows the sequential kernel execution model adopted by Megatron-LM [35], where kernels execute one after another following data dependencies.

Beyond parallelizing work across devices [5, 14, 25, 28, 35, 54], recent works have optimized latency with *nanobatching* [18, 20, 48, 49, 57], where a single pipeline microbatch is split into two equal-sized *nanobatches* with no data dependencies between them. This creates opportunities to over-

Table 1: ***Training iteration time (seconds) and energy breakdown (Joules) of Megatron-LM, Nanobatching, and each combined with Perseus, training Qwen 3 1.7B on 16 NVIDIA A100 GPUs. Megatron-LM achieves 99.0 TFLOP/s/GPU.***

	Iteration time	Static energy	Dynamic energy	Total energy
Megatron-LM	5.60	5,372	21,374	26,745
Megatron-LM + Perseus	5.60	5,374	19,531	24,905
Nanobatching	5.31	5,096	21,445	26,541
Nanobatching + Perseus	5.37	5,160	19,729	24,889

lap communication and computation kernels from different nanobatches,[2] as shown in Figure 2b.

A separate line of work has optimized the energy consumption of large model training by scaling GPU frequencies [13, 15, 51]. Notably, Perseus [15] dynamically reduces the GPU frequencies of microbatches off the critical path of computation, as shown in Figure 1b, to reduce energy consumption while keeping overall training iteration time the same. However, it follows the sequential kernel execution model (Figure 2a), missing opportunities from fine-grained kernel scheduling like nanobatching.

2.3 Breaking Down Training Energy Consumption

We can better understand the energy consumption of existing training systems and techniques by breaking down energy into static and dynamic components. Table 1 presents such a breakdown along with iteration time for Qwen 3 1.7B [44] training on 16 NVIDIA A100 GPUs.[3] Static energy is static power[4] multiplied by iteration time, and dynamic energy is static energy subtracted from total energy.

Compared to baseline Megatron-LM (first row), Nanobatching (third row) reduces iteration time, which directly lowers static energy. Dynamic energy is slightly higher because nanobatching incurs additional GPU activity from more memory accesses and extra gradient accumulations per nanobatch. Perseus applied to Megatron-LM (second row) reduces dynamic energy through explicit frequency scaling while keeping iteration time nearly the same, so static energy remains unchanged. Applying Perseus on top of Nanobatching (fourth row) combines both benefits: reduced static energy from shorter time and dynamic energy from frequency scaling.

However, we find that there is further opportunity to reduce energy beyond what is achieved by simply combining Nanobatching and Perseus. To understand this gap, we analyze how communication and kernel scheduling affect energy consumption (§3). The opportunities we find motivate the design of Kareus, which jointly optimizes kernel scheduling and frequency scaling (§4).

[2]Computation and communication exert different GPU resources, so overlapping them increases overall GPU utilization and can lead to speedup.

[3]Trained with 8 microbatches, each with size 16 and sequence length 4K, using pipeline parallelism 2, context parallelism 2, and tensor parallelism 4. The same experiment also appears in Table 3, Table 4, and Figure 11.

[4]Static power is set to be the GPU's power draw when it is in *ready* state (power state P0) without running any significant computations.

3 Energy Impact of Execution Schedules

We investigate the relationship between energy consumption and execution schedule. We first discuss how different execution schedules for the same *work* can lead to different energy consumption (§3.1), and then analyze the impact of key factors through case studies (§3.2). This reveals the necessity of joint control and the opportunity for significant energy savings (§3.3).

3.1 Execution Schedule and Energy Consumption

Regardless of *how* work (computation, memory access, and communication) is performed, the total amount of work done is the same. However, different execution schedules can still lead to different energy consumption.

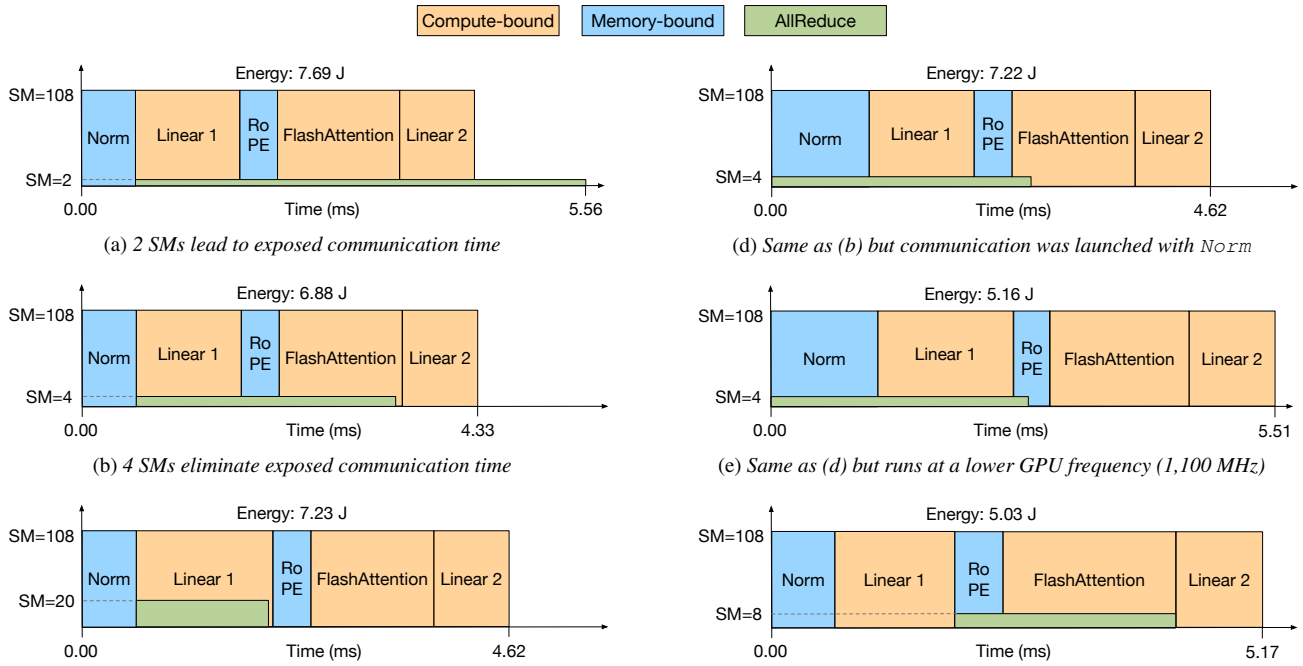
The distinction between dynamic and static power is key to understanding this. Dynamic energy reflects the total work done on the hardware (i.e., transistors switching from computation, memory access, and communication). At *the same GPU frequency*, it remains largely constant across execution schedules. In contrast, all parts of the hardware consume static power as long as they are powered on. Compute components (e.g., SMs) account for a significant portion of chip area and thus static power. When GPU resources are underutilized, static power is still dissipated without as much useful work being done, increasing total energy consumption.

In sum, GPU frequency primarily influences dynamic energy, whereas execution schedules largely determine static energy by changing end-to-end execution time. However, these factors are interdependent and cannot be decoupled and optimized separately, as we show next.

3.2 Interdependent Factors in Execution Schedules

Execution schedules differ in which kernels run concurrently on the GPU, how they split GPU resources, and at what GPU frequency they run. We study the impact of execution schedules on time and energy consumption by varying three key factors that correspond to each: (1) communication kernel launch timing, (2) number of SMs allocated to the communication kernel, and (3) GPU frequency. This definition of execution schedule generalizes nanobatching. The original nanobatching model (1) launches communication kernels as soon as possible, (2) uses default communication kernels like NCCL optimized for sequential execution (which may use excessive SMs assuming no concurrent kernels), and (3) does not perform GPU frequency scaling.

Figure 3 shows the timelines and total energy consumption of different execution schedules for a single Transformer Attention layer of Llama 3.2 3B [33], with tensor parallelism degree 4 on four fully-connected NVIDIA A100 GPUs. For simplicity of exposition, we focus on a single repeating segment: the computation kernels of one nanobatch and the communication kernel of the *previous* nanobatch. Thus, the communication kernel has no data dependencies with any



(a) *2 SMs lead to exposed communication time*

(b) *4 SMs eliminate exposed communication time*

(c) *20 SMs reduce communication latency but lead to compute interference*

(d) *Same as (b) but communication was launched with* `Norm`

(e) *Same as (d) but runs at a lower GPU frequency (1,100 MHz)*

(f) *Lower frequency (1,100 MHz) changes the energy-optimal schedule*

Figure 3: ***The time and total energy consumption of execution schedules for one Transformer Attention layer forward pass with varying SM allocation, communication launch timing, and GPU frequency. (a)–(c) show the effect of allocating different numbers of SMs to the communication kernel at 1,410 MHz, with (b) being the energy-optimal schedule. (d) is the same as (b), except that communication was launched earlier together with*** `Norm`***. Finally, (e) and (f) run at a lower GPU frequency of 1,100 MHz. (e) is the same as (d) other than frequency, whereas (f) is the energy-optimal schedule at 1,100 MHz, which is different from the schedule in (b).***

of the computation kernels in the same segment. This is the common-case execution pattern throughout training.

3.2.1 SM Allocation

First, we study the impact of GPU resource sharing via SM allocation. Allocating more SMs to communication kernels leaves fewer SMs for overlapped computation kernels given a fixed number of SMs. Figures 3a, 3b, and 3c show execution schedules that vary only in the number of SMs allocated to the communication kernel (2, 4, and 20 SMs, respectively) while launching the communication kernel together with `Linear 1`. With two SMs allocated to the communication kernel (Figure 3a), the communication does not complete before the computations finish, leaving an exposed communication period during which 106 SMs are idle. Static power is wasted during this exposed communication time, increasing total energy consumption. Increasing the number of SMs to four (Figure 3b) lets the communication kernel complete before the computation kernels, minimizing static power wastage and total energy consumption. However, further increasing the number of SMs to 20 (Figure 3c) speeds up communication only at the cost of taking SMs away from `Linear 1`, which ultimately increases total time and energy consumption. This is because the excess SMs allocated to the communication kernel slow down computation while remaining nearly idle themselves, again wasting static power.

3.2.2 Communication Launch Timing

Next, we study the impact of kernels that run together by changing when to launch the communication kernel. Figure 3d shows an execution schedule where the communication kernel runs with four SMs (the same as Figure 3b) but starts with `Norm` instead of `Linear 1`. As shown in colors, `Norm` and `RoPE` kernels are memory-bound, and with sufficient SMs, the communication kernel also requires high memory bandwidth to write the communicated data to memory. When such a communication kernel starts together with another memory-bound kernel like `Norm`, the two compete for memory bandwidth, prolonging both. This extends the period where compute resources are underutilized, wasting static power. Furthermore, compared to Figure 3b, memory bandwidth is underutilized during `FlashAttention`, wasting static power in memory components.

3.2.3 Varying GPU Frequency

Finally, we study the impact of changing the GPU frequency. Reducing GPU frequency lowers dynamic energy [15, 51]. More importantly, the energy-optimal execution schedule changes at lower frequencies. At lower frequencies, all kernels become relatively more compute-bound because reducing frequency only affects computation throughput and not memory

Table 2: ***Summary of factors and observations from case studies on the energy impact of execution schedules.***

Factor	Observation	Explanation
SM allocation	A middle-ground sweet spot exists	Too few SMs for a kernel can lead to periods where only one small kernel is running, wasting static power; too many SMs slow down other kernels without significantly speeding up the target kernel.
Launch timing	Resource demands of kernels running together matter	Communication, which can be memory-bound, running together with memory-bound operations (e.g., `Norm`, `RoPE`) causes bandwidth contention and slowdown for everyone.
GPU frequency	Changes relative kernel resource demands	Lower frequency makes kernels relatively more compute-bound, changing which kernels benefit from overlapping; it also amplifies static power's relative impact, making exposed communication more harmful.
All three	Time & energy impacts of all factors are interdependent	The energy-optimal SM allocation and launch timing depend on the GPU frequency; changing one factor shifts the energy-optimal configurations of the others.

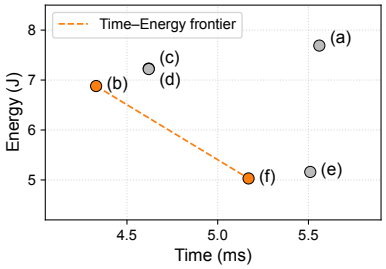


Figure 4: ***Time and energy consumption of the execution schedules in Figure 3 (a)–(f).***

throughput [38].[5] Thus, running the communication kernel with a memory-bound kernel causes less interference since memory bandwidth is less contended. Conversely, running it with a compute-bound kernel and taking SMs away from it causes more severe interference, with larger slowdowns and more static power wastage.

Figure 3e is the same as Figure 3d, but runs at a lower frequency (1,100 MHz). Reduced dynamic energy contributes substantially to lower total energy. However, execution time increases significantly because the communication kernel takes SMs away from `Linear 1`, which is now even more compute-bound at the lower frequency. The energy-optimal schedule at this frequency, shown in Figure 3f, launches communication together with `RoPE`. This lets the compute-bound `Linear` kernels run without interference, reducing total execution time and static power wastage. Figure 4 plots the time–energy frontier of the six execution schedules.

We also note that exposed communication time is relatively more harmful at lower frequencies. Dynamic power decreases significantly with frequency while static power does not—this increases static power's proportion in total power draw and thus its relative impact when wasted.

3.3 Joint Control and Opportunities

Table 2 summarizes observations from the case studies. The three factors that determine the execution schedule—SM allocation, communication launch timing, and GPU frequency—are (1) interdependent: changing one shifts the energy-optimal configuration of the others, and (2) highly impactful, with up to a 3.29× gap in time and energy consumption across the observed schedules. Therefore, optimizing time and energy requires *joint control* of all three. This is the key insight motivating the design of Kareus's optimization algorithm.

In fact, joint control of GPU frequency and kernel scheduling is provably more energy-efficient than leaving GPU frequency control to the hardware's power controller (Appendix A). The intuition is that since dynamic power grows roughly with the cube of frequency,[6] high-frequency periods cost more energy than what low-frequency periods save, so removing frequency variation reduces total dynamic energy.

Our analysis also shows that opportunities for energy savings with execution schedule control are significant. For simplicity of analysis, we have studied a single Attention layer with modest communication volume (tensor parallelism degree 4). Even in this setting, without any GPU frequency scaling, the energy-optimal execution schedule (Figure 3b) reduces energy consumption by 24.3% and time by 12.3% compared to Megatron-LM, and 7.1% energy and 5.4% time compared to Nanobatching. Larger models, more GPUs, and additional parallelism dimensions (e.g., context parallelism) provide even greater opportunities, as we show in Section 6.

4 Algorithm Design

In this section, we present the optimization algorithm of Kareus, whose goal is to find the time–energy tradeoff frontier of executing computation and communication kernels of large model training. We begin by formulating the optimization problem (§4.1), and then introduce the partitioned overlap execution model, which decomposes the problem into tractable subproblems (§4.2). Building on this model, Kareus efficiently explores the execution schedule space to derive the time–energy frontier for each subproblem (§4.3), and composes these local frontiers into a global frontier that characterizes the training iteration (§4.4). Finally, we discuss how Kareus generalizes to broader computation and communication patterns (§4.5).

[5] Essentially, the hardware roofline model's horizontal compute-bound ceiling is lowered, making it easier for kernels to be compute-bound.

[6] Dynamic power is proportional to $V^2 f$. In NVIDIA GPUs, voltage scales roughly linearly with frequency (up to a certain point), so $V^2 f$ becomes f^3.

4.1 Optimization Formulation

Objective. The goal of Kareus is to find an efficient time–energy frontier for the training iteration. Finding the whole frontier, instead of just the minimum-time point, is valuable because it allows users to select tradeoff points that meet job-level requirements (e.g., time deadlines, energy budgets) or perform dynamic adaptation to changing environments (e.g., stragglers) [15, 51]. Since the iteration frontier can be constructed by combining *microbatch* (forward and backward) frontiers (§4.4), Kareus's objective reduces to finding microbatch time–energy frontiers that Pareto-dominate those of prior works.

Decision variables. We consider three decision variables corresponding to the three key factors that influence time and energy (§3): (1) the number of SMs allocated to communication kernels; (2) kernel launch timing, which determines the execution order and overlap between kernels while respecting data dependencies; and (3) GPU frequency.

Solution space. Unfortunately, the combined solution space of these three factors is extremely large. For a typical Transformer-based LLM on an A100 GPU, this space comprises 85K candidate configurations (details in Appendix B). Each candidate in this large search space requires lengthy profiling for accurate energy measurements—on average, 13 seconds per candidate—for thermal stability (§5). An exhaustive search over the entire solution space can take up to 4,912 GPU-hours!

4.2 Partitioned Overlap Execution Model

To shrink the search space, we identify *repeating patterns* in computation and communication sequences, optimize them independently, and compose them into a global schedule.

Partition. Kareus groups kernels executing in repeating patterns into *partitions*. A partition consists of one communication kernel from one nanobatch and the longest contiguous sequence of computation kernels from the other nanobatch. There are no data dependencies between the communication kernel and the computation kernels in the same partition. This allows the communication kernel to overlap with any contiguous subsequence of computation kernels in the partition.

The second row of Figure 5 illustrates the repeating pattern of partitions. The two orange boxes (Attention–AllReduce partition) are identical, as are the two blue boxes (MLP–AllReduce partition), and both partition types repeat across all Transformer blocks.

Optimization overview. After automatically detecting the partitions, Kareus's optimizer first characterizes the time–energy frontier of each partition (§4.3), as shown in the fourth row of Figure 5. Each point on the frontier represents an efficient execution schedule for that partition, like the one depicted in the third row of Figure 5. The time–energy frontiers of all partitions are then combined to form the time–energy frontier of each forward and backward microbatch, and then combined again to form the global, iteration-level time–energy frontier (§4.4).

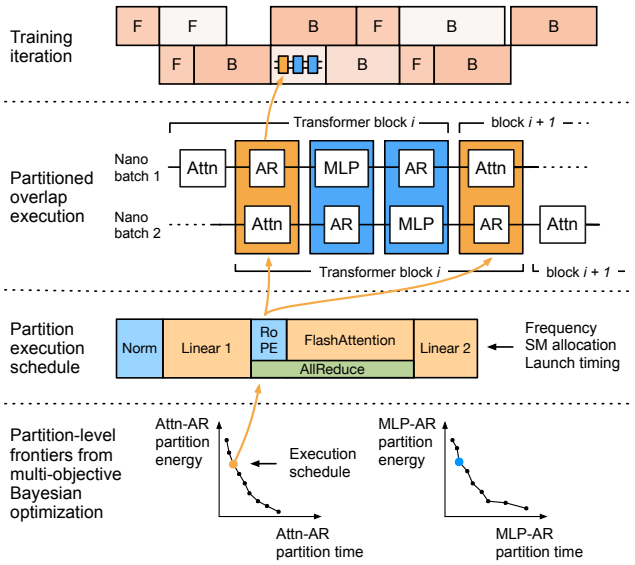


Figure 5: ***Kareus execution and optimization overview. The first row shows one training iteration with the 1F1B pipeline schedule. Inside each microbatch, partitions are sequentially executed; the second row illustrates this with the Transformer forward pass with tensor parallelism. The third row shows the execution schedule of the Attention–AllReduce partition. Each partition's execution schedule is chosen from the partition-level time–energy frontiers (fourth row) characterized by Kareus's optimization algorithm.***

4.3 Multi-Objective Bayesian Optimization

4.3.1 Bayesian Optimization Primer

For one partition, our optimization problem still involves a complex search space with mixed discrete and categorical variables. Importantly, the time and energy of each configuration have no known closed-form expression and can only be obtained through profiling. Bayesian Optimization (BO) is well-suited to this setting; it optimizes objectives over complex search spaces where the objective is unknown in form and costly to evaluate [19].

A BO algorithm has two key components. First, a *surrogate model* is trained on evaluated candidates to approximate the objective function. It is initialized with random candidates and updated as new candidates are evaluated. Second, an *acquisition function* uses the surrogate model to rank candidates and select the next one(s) to evaluate. BO then proceeds iteratively: it selects a batch of candidates using the acquisition function, evaluates them, updates the surrogate model, and repeats until convergence.

4.3.2 Kareus's MBO Algorithm

Although BO has been widely applied to systems problems [6, 11, 22, 52], it does not fit our setting as is. Standard BO targets a single objective, whereas we seek the Pareto frontier of time

and energy. We therefore design a Multi-objective Bayesian Optimization (MBO) algorithm with two tailored components: separate surrogate models for time and energy prediction, and a multi-pass candidate selection procedure that explores the frontier from multiple directions.

Surrogate model. We train two surrogate models: $\widehat{T}(x)$ for time and $\widehat{E}(x)$ for dynamic energy,[7] where x is a candidate configuration. Since time is primarily influenced by SM allocation and launch timing and dynamic energy by GPU frequency, the two models are largely orthogonal. Total energy consumption can then be computed as $\widehat{T}(x) \cdot P_{\text{static}} + \widehat{E}(x)$, where P_{static} is the GPU's static power consumption.

We adopt gradient-boosted decision trees (XGBoost) [10] as the surrogate model for two reasons. First, XGBoost training scales linearly with data points (versus cubic for Gaussian Processes), allowing fast retraining. Second, its tree-based structure handles discrete (GPU frequencies and SM allocations) and categorical (launch timing) parameters well.

Multi-pass candidate selection. Since our MBO aims to find solutions on the time–energy Pareto frontier rather than a single optimum, we design a *multi-pass* candidate selection procedure. At each iteration of MBO, we select a batch of k candidates to evaluate next. The batch consists of candidates selected based on multiple different acquisition functions that are designed to (1) cover complementary regions of the time–energy tradeoff space and (2) perform both exploration and exploitation to avoid over- or under-exploring certain regions.

Exploitation with hypervolume improvement. The acquisition functions for exploitation are based on *hypervolume improvement* (HVI), which quantifies how much a candidate can expand the current time–energy frontier toward lower time and energy (Figure 6). It is defined as

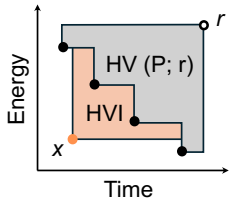


Figure 6: ***HVI.***

$$\text{HVI}(x) = \text{HV}\left(\mathcal{P} \cup \left\{\left(\widehat{T}(x), \text{Energy}(x)\right)\right\}; r\right) - \text{HV}(\mathcal{P}; r)$$

where $\mathcal{P}$ is the current set of candidates on the time–energy frontier, and r is a reference point set slightly worse (outward) than the worst observed points so that the hypervolume (HV) is computable. By switching the definition of $\text{Energy}(x)$ to total energy, dynamic energy, or static energy, each derived or taken as is from the surrogate models $\widehat{T}(x)$ and $\widehat{E}(x)$, we obtain three HVI acquisition functions. Each function guides frontier expansion in a distinct direction: dynamic energy favors lower-frequency candidates, static energy favors faster candidates, and total energy lies in the middle.

Exploration with uncertainty. To reduce the risk of under-exploring certain regions of the search space, we add an uncertainty-based acquisition function. We quantify uncertainty with *bootstrap ensembles*: multiple surrogate models trained on resampled datasets. The degree of *disagreement* between the surrogate models serves as a proxy for predictive uncertainty (i.e., how uncertain we are about the prediction). To promote exploration in either objective, we use the sum of per-objective standard deviations (rather than variances) as the acquisition score to prioritize candidates with high uncertainty.

[7] We use the hat notation to denote predictions, not true measurements.

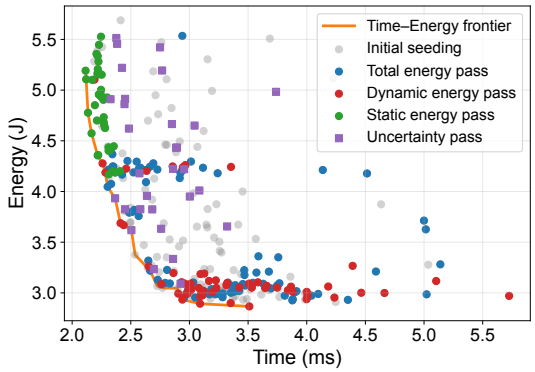


Figure 7: ***Multi-pass MBO in action with the Llama 3.2 3B model's MLP–AllReduce partition. The three exploitation passes (total energy, dynamic energy, static energy) and the exploration pass expand the efficient frontier in complementary directions, as shown by the colors of the points that land on the time–energy frontier.***

The full picture. Figure 7 illustrates frontier expansion for the Llama 3.2 3B model's MLP–AllReduce partition.[8]

Each pass expands the frontier in a complementary direction aligned with its acquisition function: dynamic energy pushes the frontier downward (lower energy), static energy leftward (lower time), and total energy toward the origin. In contrast, a single acquisition function as used in standard BO is fundamentally insufficient for MBO, as it only targets one direction. Algorithm 1 summarizes the overall Kareus MBO procedure for one partition, and this is repeated for each type of partition independently. Appendix C provides detailed settings of hyperparameters including initial random sample size, batch size, ensemble size, and stopping criteria.

4.4 Composing Frontiers

With each partition's time–energy frontier in hand, Kareus composes them into microbatch frontiers, and then into the iteration frontier.

Partition frontiers to microbatch frontier. A microbatch consists of multiple partitions executed sequentially. To construct the microbatch frontier, Kareus enumerates combinations of partition execution schedules (frequency, SM allocation, launch timing), sums their time and energy, and prunes suboptimal combinations. Algorithm 2 illustrates this process.

Two design decisions keep enumeration tractable. First, Kareus enforces a uniform GPU frequency across all partitions within a microbatch, since frequency switching takes several milliseconds and is non-negligible compared to the

[8] Microbatch size 8, sequence length 4K, and tensor parallelism degree 8.

Algorithm 1: Multi-pass multi-objective Bayesian Optimization for one partition

// Construct initial dataset for surrogate models
$\mathcal{D} \leftarrow N_{\text{init}}$ random candidates sampled and evaluated
for $b = 1, 2, \ldots, B_{\max}$ **do**
  Train surrogate models $\widehat{T}(x)$ and $\widehat{E}(x)$ on $\mathcal{D}$.
  // Exploitation: Hypervolume improvement
  foreach $x \in \mathcal{X} \setminus \mathcal{D}$ **do**
    Compute $\text{HVI}_{\text{tot}}(x)$, $\text{HVI}_{\text{dyn}}(x)$, $\text{HVI}_{\text{stat}}(x)$.
  // Exploration: Uncertainty via bootstrap ensemble
  for $m = 1$ **to** M **do**
    Train $\widehat{T}^m(x)$ and $\widehat{E}^m(x)$ on resampled $\mathcal{D}$
  foreach $x \in \mathcal{X} \setminus \mathcal{D}$ **do**
    $\text{Unc}(x) \leftarrow \sigma(\{\widehat{T}^m(x)\}) + \sigma(\{\widehat{E}^m(x)\})$
  // Multi-pass candidate selection
  $\mathcal{C} \leftarrow \text{TopK}(\{\text{HVI}_{\text{tot}}(x) : \forall x \in \mathcal{X} \setminus \mathcal{D}\}, k_1)$
  $\mathcal{C} \leftarrow \mathcal{C} \cup \text{TopK}(\{\text{HVI}_{\text{dyn}}(x) : \forall x \in \mathcal{X} \setminus \mathcal{D}\}, k_2)$
  $\mathcal{C} \leftarrow \mathcal{C} \cup \text{TopK}(\{\text{HVI}_{\text{stat}}(x) : \forall x \in \mathcal{X} \setminus \mathcal{D}\}, k_3)$
  $\mathcal{C} \leftarrow \mathcal{C} \cup \text{TopK}(\{\text{Unc}(x) : \forall x \in \mathcal{X} \setminus \mathcal{D} \setminus \mathcal{C}\}, k - |\mathcal{C}|)$
  // Evaluate candidates and update dataset
  $\mathcal{D} \leftarrow \mathcal{D} \cup \{(x, T(x), E(x)) : x \in \mathcal{C}\}$
  // Stopping condition
  $\Delta \leftarrow$ Average HV improvement over R batches
  if $\Delta < \varepsilon$ **then**
    break
return GetFrontier($\mathcal{D}$)

latency of one partition. Second, partitions of the same type share the same SM allocation and launch timing. For instance, all Attention–AllReduce partitions in Figure 5 use identical configurations. Allowing per-partition configurations would grow the search space exponentially in the number of partitions with little benefit.

Microbatch frontiers to iteration frontier. The first row of Figure 5 shows an example of how forward (F) and backward (B) microbatches are scheduled across pipeline stages. Iteration time is the sum of microbatch latencies along the critical path, while iteration energy combines the energy of all microbatches and the static energy consumed during idle times. Kareus adopts Perseus's iterative algorithm [15] to construct the training iteration frontier from microbatch frontiers.

4.5 Generalizations

Multiple communication kernels. Section 4.2 illustrated partitions using tensor parallelism as a running example. Real workloads, however, present more complex patterns. When consecutive communication kernels appear (e.g., multiple All-Gather operations under context parallelism to aggregate key and value tensors [14, 33]), Kareus fuses them into a single kernel that shares an SM allocation. This reduces kernel launch overhead and keeps scheduling tractable. With this, any model and communication scheme reduces to alternating sequences of computations and a single (possibly fused) communication, fitting the partitioned overlap model.

Algorithm 2: Microbatch frontier construction

$\mathcal{C} \leftarrow \emptyset$ // All feasible (time, energy) pairs for microbatch
foreach $f \in$ GPU frequencies **do**
  // Cartesian product of configs across all partitions
  $\Theta \leftarrow \prod_{p \in \mathcal{P}} (\text{SM allocations} \times \text{launch timings})_p$
  foreach $\theta \in \Theta$ **do**
    // Accumulate time and energy for microbatch
    $T_m \leftarrow 0, E_m \leftarrow 0$
    foreach *partition* $p \in \mathcal{P}$ **do**
      $T_m \leftarrow T_m + T_p(f, \theta[p])$
      $E_m \leftarrow E_m + E_p(f, \theta[p])$
    foreach *non-partition component* c **do**
      $T_m \leftarrow T_m + T_c(f)$
      $E_m \leftarrow E_m + E_c(f)$
    $\mathcal{C} \leftarrow \mathcal{C} \cup \{(T_m, E_m)\}$
return GetFrontier($\mathcal{C}$)

Short consecutive memory-bound computations. When multiple short, memory-bound operations appear consecutively (e.g., `BiasDropoutAdd` followed by `Norm`), Kareus groups them into one logical operation. Treating them separately would expand the launch-timing search space with only marginal gains in solution quality.

Execution model switching. Partitioned overlap is not universally superior. When the amount of work within a microbatch is small (e.g., small models or small microbatch sizes), splitting the microbatch can further reduce arithmetic intensity, leading to compute underutilization and higher static power wastage. In these cases, sequential execution can be more energy-efficient. To capture this, Kareus also profiles each sequentially executed microbatch at each frontier and includes them as candidates when constructing the microbatch time–energy frontier. Thus, the final microbatch frontier will automatically be constructed with the better execution model.

GPU power model. Kareus adopts a simplified two-component GPU power model that decomposes GPU power into dynamic power and constant static power [15, 32, 41], which suffices for reasoning about GPU energy consumption in large model training. There are finer-grained models that further divide static power into chip leakage and constant power [29, 45], where chip leakage is sensitive to voltage. We leave extending Kareus to such models as future work.

5 Implementation

5.1 System Overview

Figure 8 shows an overview of the Kareus system. Given a workload, Kareus identifies partitions (❶; §4.2) and runs multi-objective Bayesian optimization per partition (❷; §4.3), evaluating candidates with the thermally stable profiler (§5.3).

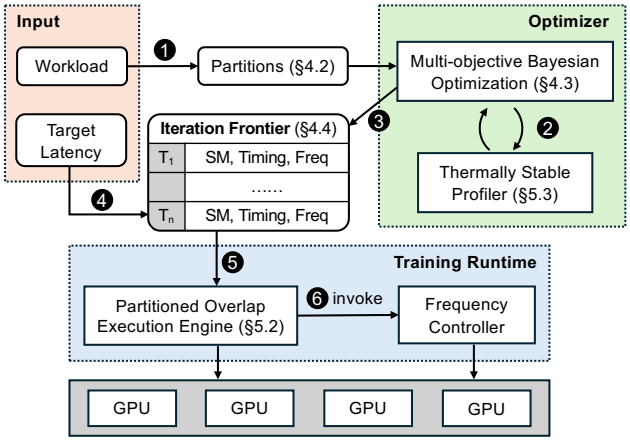


Figure 8: ***Kareus system overview.***

The per-partition frontiers are then composed into the iteration time–energy frontier (❸; §4.4). At runtime, given a target iteration latency, Kareus selects an execution schedule from the frontier (❹) and deploys it to the partitioned overlap execution engine (❺; §5.2), which invokes the GPU frequency controller to switch each GPU's frequency asynchronously as planned (❻).

5.2 Partitioned Overlap Execution Engine

Kareus's training execution engine builds on Megatron-LM [35], executing Transformer blocks as a sequence of partitions. Before each microbatch execution, Kareus loads the corresponding microbatch configuration, switches between Megatron-LM's default sequential execution and our partitioned overlap execution mode, and configures the SM allocation and launch timing for partitioned overlap execution. A custom `torch.autograd.Function` wraps all partitions of a Transformer block, allowing different partitions and partition schedules for forward and backward passes.

Communication kernels in Kareus are implemented with MSCCL++ [2,23], which provides fine-grained SM allocation control with grid size. Computation and communication run on separate CUDA streams to enable overlap, and CUDA events control launch timing. The GPU frequency controller is adapted from Perseus's open-source implementation [4,15].

5.3 Thermally Stable Profiling

Kareus measures the time and energy of a partition during MBO using Zeus [4,51], which internally uses NVML [3]. Accurate energy measurements require care; we present a detailed experimental analysis in Section 6.7.

Measurement window. NVML's sampling interval on NVIDIA GPUs is approximately 100 ms, so millisecond-scale measurements incur large errors. Kareus executes each partition repeatedly over a 5-second window, after which energy measurements stabilize.

Thermal cooldown. The power consumption of any hardware is temperature-dependent. Without cooldown between candidates, profiles of earlier partitions may heat up the GPU and bias subsequent measurements. Kareus inserts a 5-second cooldown period, which reliably brings the GPU below 32°C in our environment; the required duration depends on the server's cooling capability. In sum, profiling each candidate takes approximately 13 seconds in our setup, including initialization, warm-up, measurement, and cooldown.

6 Evaluation

We evaluate Kareus on 14 workloads and compare it against state-of-the-art baselines. Our key findings are as follows:

- Kareus achieves a superior time–energy frontier compared to prior approaches. In end-to-end training on real GPUs, it delivers up to 28.3% energy reduction under the same time budget and up to 27.5% time reduction under the same energy budget compared to the baselines (§6.2).
- In emulated large-scale training, Kareus consistently outperforms the baselines, achieving time and energy reductions comparable to real-world training results (§6.3).
- Ablation and sensitivity analyses show that jointly optimizing dynamic and static energy is necessary (§6.4) and is effective across a range of microbatch sizes (§6.5).
- Kareus's MBO algorithm constructs the time–energy frontier with reasonable overhead, and each pass in multi-pass candidate selection is indispensable (§6.6); the thermally stable profiler delivers stable energy measurements (§6.7).

6.1 Experimental Setup

Testbed. Experiments are conducted on 16 NVIDIA A100 40GB GPUs deployed across two AWS p4d.24xlarge instances. GPUs are fully connected intra-node via NVSwitch, and cross-node bandwidth is 400 Gbps.

Workloads. We evaluate Kareus on Llama 3.2 3B [33] and Qwen 3 1.7B [44] on the physical testbed, and perform large-scale emulation for Llama 3.3 70B [33]. For pipeline parallelism, we manually partition stages such that stages are as balanced as possible, following Perseus [15]. For context parallelism, we follow the scheme used in Llama 3 [33], where key–value tensors are collected across GPUs via AllGather. We use activation checkpointing to reduce memory pressure.

Baselines. We mainly compare against three baselines:

- **Megatron-LM ("M").** Baseline Megatron-LM [35] with the sequential execution model and maximum GPU frequency. Produces a single point on the time–energy plane.
- **Megatron-LM + Perseus ("M+P").** Perseus [15] applied to the above. This produces a time–energy frontier.
- **Nanobatching + Perseus ("N+P").** A training engine based on Megatron-LM that implements nanobatching, a special case of Kareus's partitioned overlap execution model (§3.2), integrated with Perseus like the above. This also produces a time–energy frontier.

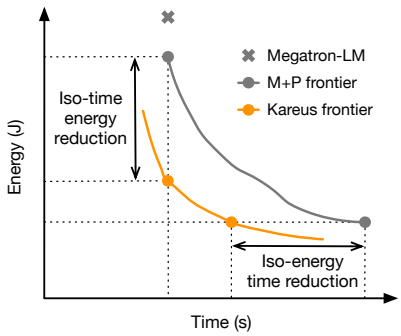


Figure 9: ***Illustrative example showing iteration time–energy frontiers. For the max-throughput comparison, we compare the leftmost points of each frontier with Megatron-LM. For frontier improvement, we report the iso-time energy reduction and iso-energy time reduction compared to Megatron-LM + Perseus.***

Metrics. Kareus's gain is fundamentally a superior time–energy tradeoff frontier, which cannot be fully captured by a single number. Therefore, we define two modes of comparison based on two viable use cases.

- **Max-throughput comparison.** When iteration time constraints (e.g., deadlines, stragglers) are not present, the training pipeline operates in maximum-throughput mode. For methods that produce a time–energy frontier, this means operating at the leftmost (lowest time) point. We set Megatron-LM as the baseline, and report iteration time and energy reduction (%) of M+P, N+P, and Kareus.
- **Frontier improvement.** We set M+P as the baseline, and define two metrics that quantify frontier improvements for N+P and Kareus (Figure 9): (1) *Iso-time energy reduction* (%): energy reduction with iteration time deadline set as M+P's minimum iteration time; and (2) *Iso-energy time reduction* (%): time reduction with iteration energy budget set as M+P's minimum iteration energy.

6.2 End-to-End Results

In this section, we evaluate the iteration time and energy performance on the testbed GPUs for the max-throughput (§6.2.1) and frontier improvement (§6.2.2) comparisons. The workload configurations (parallelism, microbatch size, and sequence length) can be found in Table 3.

6.2.1 Max-Throughput Comparison

Table 3 reports the iteration time and energy reductions achieved by all methods under the max-throughput regime. Overall, Kareus achieves up to 14.9% reduction in iteration time and 22.1% reduction in energy consumption compared to Megatron-LM, strictly outperforming the baselines on time and energy.

Time reduction. The time reduction mainly comes from reducing SM idle time caused by resource underutilization due to exposed communication or resource contention by poorly scheduled kernel overlaps. As a result, the time reduction trend largely follows the communication overhead induced by different model configurations. Compared to Megatron-LM, both overlap execution models achieve larger gains under tensor parallelism (TP) than under context parallelism with tensor parallelism (CP+TP). Moreover, increasing the microbatch size yields greater time reduction than increasing the sequence length, since the former introduces relatively higher communication overhead.

When compared to Nanobatching, Kareus provides larger additional gains under CP+TP than under TP alone, because fine-grained control of communication scheduling becomes more critical for complex communication patterns. Moreover, Nanobatching can significantly slow down computation (e.g., on Qwen 1.7B with CP2+TP4, microbatch size 8, and sequence length 4K), as it leads to GPU underutilization when the workload is small. In contrast, Kareus can automatically switch to the sequential execution model (§4.5).

Energy reduction. Energy reflects the combined effects of time and power, which is why energy reduction trends do not always track time reduction trends. In many cases, Nanobatching + Perseus reduces time but yields smaller energy reduction than Megatron-LM + Perseus, because overlap raises GPU utilization and thus also power, and the power increase offsets time reduction. In contrast, Kareus achieves larger energy reductions by simultaneously optimizing overlap and GPU frequency, enabling greater time reduction while also identifying opportunities to reduce power, as the following case studies show.

Case studies. We find an interesting case in Qwen 3 1.7B with TP8, microbatch size 8, and sequence length 4K. Both baselines operate the critical-path microbatch at 1,410 MHz (the maximum GPU frequency). However, Kareus discovers an overlap configuration that runs at 1,350 MHz yet attains the fastest latency, resulting in an additional 10.7% energy reduction over Nanobatching + Perseus. This happens because nanobatching overlap at 1,410 MHz improves resource utilization but raises instantaneous power, which triggers GPU frequency throttling. Thus, the time-averaged GPU frequency (which determines execution time) is close to 1,350 MHz, while the time-averaged dynamic power (which determines dynamic energy) remains closer to that of 1,410 MHz. Operating at a constant frequency provably consumes less energy than operating at a fluctuating frequency with the same average, which highlights the importance of jointly optimizing kernel scheduling and GPU frequency scaling (§3.3).

Figure 10 further illustrates representative partition execution schedules across microbatches of the aforementioned configuration. In the forward and backward Attention–AllReduce partitions, Kareus decides not to overlap `AllReduce` with memory-bound kernels (e.g., `Norm`) at a higher frequency (1,350 MHz); instead, it shifts the overlap to more memory-bound kernels at a lower frequency (1,290 MHz), consistent with the analysis in Section 3. For the MLP–AllReduce partitions, the selected configurations differ due to variations

Table 3: ***[Experiment] Iteration time and energy reductions (%) relative to Megatron-LM (higher is better) for all methods under the max-throughput configuration. Pipeline parallelism degree is fixed at 2, and number of microbatches is fixed at 8. `OOM` indicates that the GPU runs out of memory at the corresponding settings. Negative values indicate increased time or energy relative to the baseline.***

Model	Parallelism	ubatch Size	Sequence Length	Time Reduction (%)			Energy Reduction (%)			TFLOP/s /GPU
				Megatron-LM +Perseus	Nanobatching +Perseus	Kareus	Megatron-LM +Perseus	Nanobatching +Perseus	Kareus	
Llama 3.2 3B	TP8	8	4K	−0.3	8.5	12.3	10.0	15.5	19.6	104.8
		8	8K			OOM				
		16	4K			OOM				
Llama 3.2 3B	CP2TP4	8	4K	0.2	0.0	5.2	7.4	7.3	14.4	117.3
		8	8K	−1.4	1.5	6.2	11.5	7.3	14.9	138.0
		16	4K	−0.7	2.4	8.0	8.9	7.0	16.2	131.0
Qwen 3 1.7B	TP8	8	4K	−0.5	5.6	12.2	7.7	12.8	22.1	88.5
		8	8K	0.1	9.0	14.9	7.9	12.2	15.1	107.2
		16	4K	−0.1	9.4	14.8	7.6	13.4	17.3	97.0
Qwen 3 1.7B	CP2TP4	8	4K	−0.5	−20.4	−0.5	7.1	3.1	7.1	88.9
		8	8K	0.5	1.7	8.0	7.4	6.7	8.7	119.4
		16	4K	0.0	3.9	10.5	6.9	6.9	11.2	110.6

in kernel lengths and types. Kareus starts overlap from the beginning and allocates only 6 SMs to `AllReduce` at 1,350 MHz, which provides enough time to hide communication while leaving most SMs available for the long compute-bound `Linear 1` kernels.

6.2.2 Frontier Improvement

Table 4 reports the iso-time energy reduction and iso-energy time reduction for Nanobatching + Perseus and Kareus, which demonstrate the expansion of the time–energy frontier. Overall, Kareus achieves up to 28.3% iso-time energy reduction and 27.5% iso-energy time reduction compared to Megatron-LM + Perseus, establishing a strictly better time–energy tradeoff over Nanobatching + Perseus. Figure 11 shows a representative frontier comparison for Qwen 3 1.7B with CP2, TP4, microbatch size 16, and sequence length 4K. We present the complete frontier comparison plots for all model configurations in Appendix D.

Under the same time budget, where static energy is fixed, Kareus achieves higher GPU utilization and can therefore execute at lower GPU frequencies to complete the workload, reducing dynamic energy consumption. Conversely, under the same energy budget, Kareus minimizes static energy wastage, allowing more energy to be allocated to dynamic execution at higher GPU frequencies to complete the workload faster.

6.3 Large-Scale Emulation

In this section, we conduct large-scale emulation of Llama 3.3 70B based on smaller-scale profiling on our testbed hardware and compare against Megatron-LM + Perseus. We use the emulator from Perseus [15]. Results show that trends in time and energy reduction are consistent with those in Section 6.2 measured on testbed GPUs.

Methodology and parameters. For Kareus, we obtain the iteration-level time–energy frontier using the MBO algorithm described in Section 4 by composing partition-level frontiers.

Table 4: ***[Experiment] Iso-time energy reduction (%) and iso-energy time reduction (%) relative to Megatron-LM + Perseus for Nanobatching + Perseus and Kareus. "−" indicates that no configuration satisfies that constraint. For example, if Nanobatching + Perseus has a longer iteration time than Megatron-LM + Perseus at the leftmost point, then no iso-time point exists.***

				Iso-Time Energy Reduction (%)		Iso-Energy Time Reduction (%)	
				N+P	Kareus	N+P	Kareus
3B	TP8	8	4K	21.0	24.3	18.9	24.0
		8	8K		OOM		
		16	4K		OOM		
3B	CP2TP4	8	4K	–	17.4	7.0	13.9
		8	8K	4.0	12.4	4.4	16.4
		16	4K	6.6	20.4	5.1	12.5
1.7B	TP8	8	4K	16.8	26.8	17.8	27.5
		8	8K	20.0	23.1	17.7	23.1
		16	4K	20.4	28.3	19.9	26.7
1.7B	CP2TP4	8	4K	–	0.0	–	5.1
		8	8K	−0.7	15.0	−0.4	12.1
		16	4K	8.9	20.1	8.4	19.7

For Megatron-LM + Perseus, we profile the time and energy of each Transformer block using the same profiling methodology described in Section 5.3, and construct the iteration-level frontier. We perform strong scaling as we vary the number of GPUs (Table 5), while fixing the global batch size to 2048, which is adopted in the Llama 3 training [33]. We use pipeline parallelism degree 10 and tensor parallelism degree 8, with microbatch size 4 and sequence length 4K [33].

Max-throughput comparison. We report the max-throughput comparison results for Megatron-LM + Perseus and Kareus across different numbers of microbatches in Table 6, and plot the corresponding iteration-level time–energy frontiers in Appendix E. The energy reductions achieved by both methods are generally higher than those observed in the

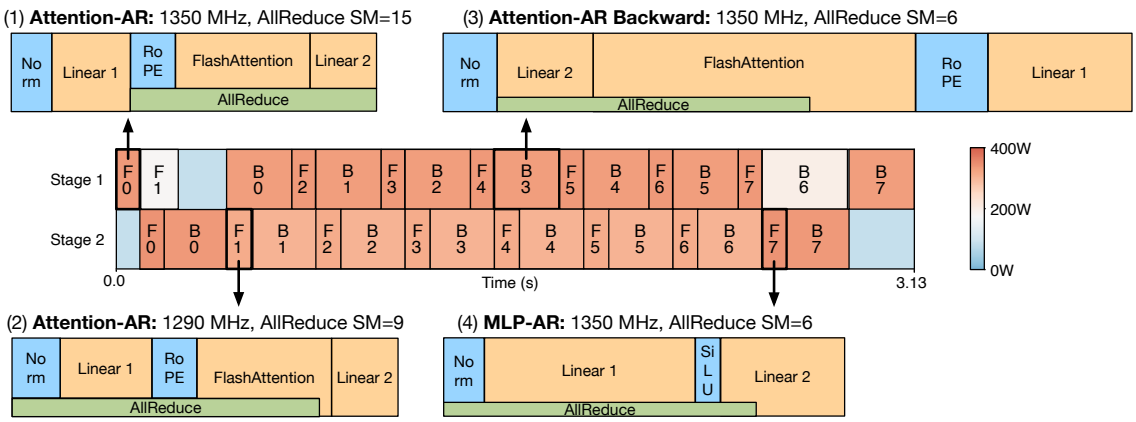


Figure 10: *Case study of Qwen 3 1.7B with TP8, microbatch size 8, and sequence length 4K. The pipeline schedule is colored by average GPU power draw, and the partition execution schedules are drawn to scale for time. (1) and (2) compare the forward Attention–AllReduce partition at 1,350 MHz and 1,290 MHz, respectively. (3) and (4) show the backward Attention–AllReduce and forward MLP–AllReduce partitions at 1,350 MHz. Backward microbatch is partitioned following the definition in Section 4.2;* `Norm` *is treated as the first kernel because it follows the AllReduce kernel in the transformer block, and other kernels are ordered in reverse.*

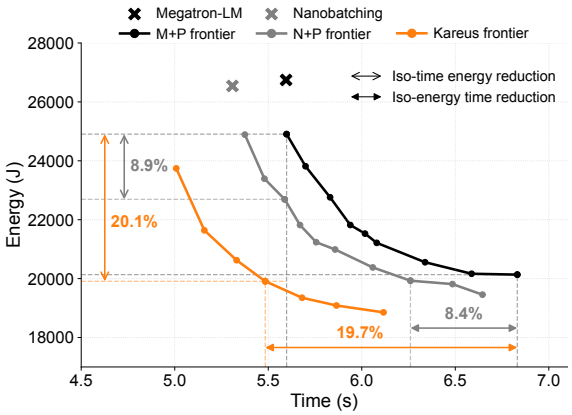


Figure 11: *[Experiment] Representative iteration time–energy frontier comparison for Qwen 3 1.7B with CP2, TP4, microbatch size 16, and sequence length 4K.*

Table 5: *Strong scaling configurations for large-scale emulation.*

# GPUs	# Pipelines	# Microbatches Per Pipeline	Global Batch Size
10240	128	16	2048
5120	64	32	
2560	32	64	
1280	16	128	

testbed experiments, because the larger number of pipeline stages and microbatches on non-critical paths can be slowed down to save dynamic energy. As the number of microbatches increases, the relative portion of pipeline bubbles during the warm-up and cooldown phases—which are normally reduced down to the lowest frequency—decreases, leading to a slight decrease in energy reduction. The time reduction achieved by Kareus is slightly lower than that of the TP=8 case in the testbed experiments, because a smaller microbatch size is used in this setting due to the GPU memory constraints.

Table 6: *[Emulation] Iteration time and energy reduction (%) relative to Megatron-LM of Megatron-LM + Perseus and Kareus under the max-throughput configuration for Llama 3.3 70B.*

# Microbatches	Time Reduction (%)		Energy Reduction (%)	
	M+P	Kareus	M+P	Kareus
16	0.0	9.3	15.0	20.2
32	0.0	9.2	14.3	20.0
64	0.0	9.1	13.8	19.8
128	0.0	9.1	13.5	19.7

Table 7: *[Emulation] Iso-time energy reduction (%) and iso-energy time reduction (%) relative to Megatron-LM + Perseus of Kareus for Llama 3.3 70B.*

	# Microbatches			
	16	32	64	128
Iso-Time Energy Reduction (%)	11.6	14.1	15.3	15.1
Iso-Energy Time Reduction (%)	19.1	16.8	16.4	16.0

Frontier improvement. We report the frontier improvement results for Kareus across different numbers of microbatches in Table 7. As the number of microbatches increases, the iso-time energy reduction improves, because the absolute iteration time grows, providing more slack for dynamic energy savings. In contrast, the iso-energy time reduction decreases, since static energy consumes a larger fraction of the fixed energy budget.

6.4 Ablation Study on Search Space

In this section, we conduct an ablation study to demonstrate the necessity of jointly optimizing dynamic and static energy in Kareus. We compare Kareus against three ablated variants: removing dynamic energy optimization (i.e., frequency scaling), removing static energy optimization (i.e., kernel scheduling with SM allocation and launch timing), and removing both (i.e., Nanobatching). We evaluate all systems on Qwen 3 1.7B with PP=2, TP=8, 8 microbatches of size

Table 8: ***[Experiment] Iteration time and energy increase (%) of ablated variants relative to Kareus under the max-throughput configuration.***

System	Time Inc. (%)	Energy Inc. (%)
Kareus w/o frequency	1.0	12.9
Kareus w/o kernel schedule	8.2	10.8
Nanobatching	7.8	20.6

8, and sequence length 4K, the same configuration used in end-to-end experiments (Tables 3 and 4).

Table 8 reports the iteration time and energy increases of each ablated variant relative to Kareus under the max-throughput configuration. Removing frequency scaling results in a 12.9% increase in energy, and removing kernel scheduling leads to a 10.8% increase in energy. As expected, removing either of the optimization dimensions fails to deliver the full optimization potential of Kareus, demonstrating the necessity of joint optimization of both dynamic and static energy.

6.5 Sensitivity to Microbatch Size

In this section, we further study how the microbatch size, an important training parameter that directly affects computation arithmetic intensity, the ratio of communication to computation, and pipeline imbalance, affects the effectiveness of Kareus. We choose Qwen 3 1.7B with TP=8 and sequence length 4K as in Section 6.4 and vary the microbatch size from 8 to 20.[9] Tables 9 and 10 report the max-throughput comparison and frontier improvement results, respectively; Appendix F shows the corresponding time–energy frontiers.

Overall, Kareus demonstrates consistent effectiveness across varying microbatch sizes. As microbatch size increases, Kareus achieves greater time reduction (Table 9), as communication–computation overlap more effectively utilizes SMs when decomposing microbatches into nanobatches. This is also why microbatch size 20 achieves the best iso-time and iso-energy reductions (Table 10). Max-throughput energy reduction (Table 9) shows a more complex trend for two reasons. First, microbatch size 8 has a pronounced energy reduction due to the same reason as the case study in Section 6.2.1, where Kareus discovers a more energy-efficient critical path execution schedule that runs at a lower frequency. Second, changing the microbatch size impacts pipeline stage imbalance, which affects the potential for energy savings by slowing down non-critical microbatches—this worked in favor of microbatch size 12 and 20 by increasing imbalance and allowing more energy savings.

6.6 MBO Analysis

Overhead and breakdown. Kareus's MBO (§4.3) obtains the time–energy frontier for each partition. In our testbed experiments, MBO for different partitions is conducted in parallel and takes 2 hours (32 GPU hours) on average. This overhead is substantially lower than that of exhaustive search (307 hours, as discussed in Section 4.1), and is negligible compared to typical end-to-end training times (e.g., 54 days for Llama 3 [33]).

The overhead of MBO is dominated by thermally stable profiling of candidates (§5.3), accounting for 97% of the total overhead. For a typical partition in our setting, which evaluates a batch of 32 candidates per MBO iteration, thermally stable profiling takes 6.9 minutes on average, while surrogate model training and acquisition function evaluation takes 11 seconds on average.

Multi-pass candidate selection. After random candidate initialization, MBO selects candidates across four passes: total energy pass, dynamic energy pass, static energy pass, and uncertainty pass (§4.3). On average, random initialization contributes 35% of the selected candidates—they happen to be on the frontier. The other 65% of the selected candidates are discovered by the total energy pass, dynamic energy pass, static energy pass, and uncertainty pass, each 25%, 22%, 12%, and 6% of the selected candidates, respectively. All passes contribute a non-negligible portion of the final candidates.

[9] Larger microbatch sizes are not evaluated due to GPU memory capacity.

Table 9: ***[Experiment] Iteration time and energy reduction (%) relative to Megatron-LM of Megatron-LM + Perseus and Kareus across different microbatch sizes.***

ubatch Size	Time Reduction (%)		Energy Reduction (%)	
	M+P	Kareus	M+P	Kareus
8	-0.5	12.2	7.7	22.1
12	-0.4	14.7	9.4	17.6
16	-0.1	14.8	7.6	17.3
20	-0.2	18.1	9.4	18.8

Table 10: ***[Experiment] Iso-time energy reduction (%) and iso-energy time reduction (%) relative to Megatron-LM + Perseus of Kareus across different microbatch sizes.***

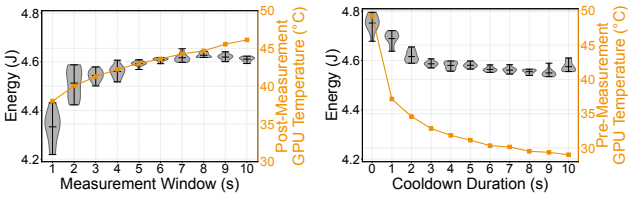
	Microbatch Size			
	8	12	16	20
Iso-Time Energy Reduction (%)	26.8	28.6	28.3	29.8
Iso-Energy Time Reduction (%)	27.5	27.3	26.7	28.8

6.7 Thermally Stable Profiler

During MBO, Kareus adopts thermally stable profiling to measure the time and energy of each candidate (§5.3). We conduct an experimental study using the Attention–AllReduce partition of Llama 3.2 3B on 8 NVIDIA A100 GPUs[10] to analyze the impact of measurement window and cooldown duration on the energy measurement accuracy.

Measurement duration. We fix the cooldown duration to 5 seconds and vary the measurement window from 1 second to 10 seconds, each with 10 repeated profiling trials. Figure 12a shows the distribution of measured energy consumption under different measurement windows, together with the average GPU temperature *after* each measurement. With very short

[10] Batch size is 4 and sequence length is 4K, with tensor parallelism degree 8, running at 1,410 MHz.



(a) *Measurement duration* (b) *Cooldown duration*

Figure 12: ***Impact of changing (a) measurement duration and (b) cooldown duration for the Thermally Stable Profiler. We report the distribution of energy across 10 repeated trials, along with the average GPU temperature before and after measurement.***

measurement windows (e.g., below 2 seconds), energy measurements exhibit large variability due to the 100 ms NVML counter update interval, and have a lower mean because the GPU has not fully warmed up. Energy measurement stabilizes from 5 seconds onward, which is why we chose 5 seconds as our measurement duration.

Cooldown duration. Cooldown duration is the waiting time between profiling consecutive partitions. We fix the measurement duration to 5 seconds and vary the cooldown duration from 0 second (no cooldown) to 10 seconds, with 10 repeated profiling trials. Figure 12b shows the distribution of measured energy consumption under different cooldown durations, along with the average GPU temperature *before* each measurement. We observe that mean energy consumption strongly correlates with the GPU temperature at the start of measurement, making sufficient cooldown essential for accurate measurement. Temperature and measurements stabilize from 5 seconds onward, so we cool down for 5 seconds between partitions. We note that the optimal cooldown duration would require tuning for different computing environments.

7 Related Work

7.1 ML Energy Optimization

The energy consumption of ML workloads has attracted extensive attention, prompting a growing body of work on energy measurement and optimization [13, 15–17, 24, 40, 43, 51, 53]. In particular, the time–energy tradeoff frontier has been a key tool for reasoning about performance and energy efficiency. For LLM serving, DynamoLLM [43] explores time–energy tradeoffs via model parallelism and GPU frequency scaling under latency constraints. The ML.ENERGY benchmark [16] characterizes the time–energy frontier across deployment configurations to guide latency-aware energy optimization. On the training side, Zeus [51] identifies job-level time–energy tradeoffs on recurring training workloads. Perseus [15] dynamically scales GPU frequency per microbatch to optimize energy under pipeline iteration time deadlines. However, these approaches often yield suboptimal frontiers because they overlook the impact of low-level execution scheduling, such as communication and computation overlap. Jayaweera et al. [27] propose energy-aware tile size selection for GPU kernels, but do not consider interactions with system-level latency targets, limiting the potential for energy optimization.

7.2 Communication Scheduling

As model scale grows, communication incurs increasingly significant overhead, motivating extensive work on overlapping communication with computation [8, 9, 12, 18, 20, 26, 28, 34, 36, 39, 48–50, 55–57]. Nanobatching is a key technique that removes data dependencies and enables overlap opportunities. In LLM serving, NanoFlow [57], TokenWeave [20], and vLLM Dual Batch Overlap [48] improve throughput via nanobatching with fine-grained kernel overlap. For training, DeepSpeed Domino [49] applies nanobatching to overlap tensor-parallel communication, while DeepSeek DualPipe [18] pipelines forward and backward nanobatches for communication overlap. Concerto [12] optimizes communication scheduling from a compiler perspective and adopts partial nanobatching when data dependencies are present. Other hand-crafted kernels fuse communication into computation [8, 36, 55, 56], but are tightly tied to specific parallelism strategies and operators. For example, Megatron-LM supports tensor-parallel communication overlap [36], which relies on its native sequence parallelism [30]. From the energy perspective, these approaches reduce static energy by reducing time, but do not explicitly optimize energy in their design. Kareus reveals the energy impact of communication scheduling and generalizes the nanobatching strategy to enable fine-grained SM allocation and launch-timing control, further minimizing static energy while jointly reducing dynamic energy by optimizing GPU frequency.

8 Conclusion

We present Kareus, an execution-aware energy optimizer for large model training. Kareus demonstrates that SM allocation, kernel launch timing, and GPU frequency jointly influence static and dynamic energy, and that the optimal time–energy frontier cannot be attained by optimizing them in isolation. Building on this insight, Kareus introduces a partitioned overlap execution model and a multi-objective Bayesian optimization framework that jointly searches over execution schedules, exposing a strictly better time–energy tradeoff frontier than prior systems.

Looking forward, we believe energy-aware execution scheduling should be a first-class concern in ML systems, not an afterthought. As AI scaling continues and energy constraints tighten for gigawatt-scale AI datacenters, systems that co-optimize computation, communication, and power will be essential, not just for cost savings but for enabling training runs that would otherwise be infeasible. Kareus takes a step in this direction, and we hope it spurs further work on energy as a core system design metric.

Acknowledgements

We would like to thank the OSDI reviewers, our shepherd, and SymbioticLab members for their insightful feedback. This work was supported in part by NSF grants CCF-2450085 and CNS-2106184, DARPA ML2P Award HR0011-26-9-E190, and by grants from Cisco, Ford, Mozilla Foundation, and Laude Institute. Jae-Won Chung is additionally supported by the Kwanjeong Educational Foundation.

References

[1] How much electricity does an American home use? https://www.eia.gov/tools/faqs/faq.php?id=97&t=3.

[2] MSCCL++. https://github.com/microsoft/mscclpp.git.

[3] NVIDIA Management Library (NVML). https://developer.nvidia.com/nvidia-management-library-nvml.

[4] Zeus. https://github.com/ml-energy/zeus.

[5] Sami Alabed, Daniel Belov, Bart Chrzaszcz, Juliana Franco, Dominik Grewe, Dougal Maclaurin, James Molloy, Tom Natan, Tamara Norman, Xiaoyue Pan, Adam Paszke, Norman A. Rink, Michael Schaarschmidt, Timur Sitdikov, Agnieszka Swietlik, Dimitrios Vytiniotis, and Joel Wee. PartIR: Composing SPMD partitioning strategies for machine learning. In *ASPLOS*, 2025.

[6] Omid Alipourfard, Hongqiang Harry Liu, Jianshu Chen, Shivaram Venkataraman, Minlan Yu, and Ming Zhang. CherryPick: Adaptively unearthing the best cloud configurations for big data analytics. In *NSDI*, 2017.

[7] CBRE. Global data center trends 2025. https://www.cbre.com/insights/reports/global-data-center-trends-2025, 2025.

[8] Li-Wen Chang, Wenlei Bao, Qi Hou, Chengquan Jiang, Ningxin Zheng, Yinmin Zhong, Xuanrun Zhang, Zuquan Song, Chengji Yao, Ziheng Jiang, Haibin Lin, Xin Jin, and Xin Liu. FLUX: Fast software-based communication overlap on GPUs through kernel fusion. *arXiv preprint arXiv:2406.06858*, 2024.

[9] Chang Chen, Xiuhong Li, Qianchao Zhu, Jiangfei Duan, Peng Sun, Xingcheng Zhang, and Chao Yang. Centauri: Enabling efficient scheduling for communication-computation overlap in large model training via communication partitioning. In *ASPLOS*, 2024.

[10] Tianqi Chen and Carlos Guestrin. XGBoost: A scalable tree boosting system. In *KDD*, 2016.

[11] Tianqi Chen, Lianmin Zheng, Eddie Yan, Ziheng Jiang, Thierry Moreau, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. Learning to optimize tensor programs. In *NeurIPS*, 2018.

[12] Shenggan Cheng, Shengjie Lin, Lansong Diao, Hao Wu, Siyu Wang, Chang Si, Ziming Liu, Xuanlei Zhao, Jiangsu Du, Wei Lin, and Yang You. Concerto: Automatic communication optimization and scheduling for large-scale deep learning. In *ASPLOS*, 2025.

[13] Sangjin Choi, Inhoe Koo, Jeongseob Ahn, Myeongjae Jeon, and Youngjin Kwon. EnvPipe: Performance-preserving DNN training framework for saving energy. In *ATC*, 2023.

[14] Weiwei Chu, Xinfeng Xie, Jiecao Yu, Jie Wang, Amar Phanishayee, Chunqiang Tang, Yuchen Hao, Jianyu Huang, Mustafa Ozdal, Jun Wang, Vedanuj Goswami, Naman Goyal, Abhishek Kadian, Andrew Gu, Chris Cai, Feng Tian, Xiaodong Wang, Min Si, Pavan Balaji, Ching-Hsiang Chu, and Jongsoo Park. Scaling Llama 3 training with efficient parallelism strategies. In *ISCA*, 2025.

[15] Jae-Won Chung, Yile Gu, Insu Jang, Luoxi Meng, Nikhil Bansal, and Mosharaf Chowdhury. Reducing energy bloat in large model training. In *SOSP*, 2024.

[16] Jae-Won Chung, Jeff J. Ma, Ruofan Wu, Jiachen Liu, Oh Jun Kweon, Yuxuan Xia, Zhiyu Wu, and Mosharaf Chowdhury. The ML.ENERGY benchmark: Toward automated inference energy measurement and optimization. In *NeurIPS Datasets and Benchmarks*, 2025.

[17] Jae-Won Chung, Ruofan Wu, Jeff J. Ma, and Mosharaf Chowdhury. Where do the Joules go? diagnosing inference energy consumption. *arXiv preprint arXiv:2601.22076*, 2026.

[18] DeepSeek-AI. DeepSeek-V3 technical report. *arXiv preprint arXiv:2412.19437*, 2024.

[19] Peter I. Frazier. A tutorial on Bayesian optimization. *arXiv preprint arXiv:1807.02811*, 2018.

[20] Raja Gond, Nipun Kwatra, and Ramachandran Ramjee. TokenWeave: Efficient compute-communication overlap for distributed LLM inference. In *MLSys*, 2026.

[21] James Hamilton. Constraint-driven innovation (CIDR 2024 keynote talk). https://mvdirona.com/jrh/talksandpapers/JamesHamiltonCIDR2024.pdf, 2024.

[22] Erik Orm Hellsten, Artur L. F. Souza, Johannes Lenfers, Rubens Lacouture, Olivia Hsu, Adel Ejjeh, Fredrik Kjolstad, Michel Steuwer, Kunle Olukotun, and Luigi Nardi. BaCO: A fast and portable Bayesian compiler optimization framework. In *ASPLOS*, 2023.

[23] Changho Hwang, Peng Cheng, Roshan Dathathri, Abhinav Jangda, Saeed Maleki, Madan Musuvathi, Olli Saarikivi, Aashaka Shah, Ziyue Yang, Binyang Li, Caio Rocha, Qinghua Zhou, Mahdieh Ghazimirsaeed, Sreevatsa Anantharamu, and Jithin Jose. MSCCL++: Rethinking GPU communication abstractions for AI inference. In *ASPLOS*, 2026.

[24] Md. Monzurul Amin Ifath and Israat Haque. Characterizing performance–energy trade-offs of large language models in multi-request workflows. *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, 2026.

[25] Insu Jang, Runyu Lu, Nikhil Bansal, Ang Chen, and Mosharaf Chowdhury. Efficient distributed MLLM training with Cornstarch. In *ICML*, 2026.

[26] Abhinav Jangda, Jun Huang, Guodong Liu, Amir Hossein Nodehi Sabet, Saeed Maleki, Youshan Miao, Madanlal Musuvathi, Todd Mytkowicz, and Olli Saarikivi. Breaking the computation and communication abstraction barrier in distributed machine learning workloads. In *ASPLOS*, 2022.

[27] Malith Jayaweera, Martin Kong, Yanzhi Wang, and David Kaeli. Energy-aware tile size selection for affine programs on GPUs. In *CGO*, 2024.

[28] Ziheng Jiang, Haibin Lin, Yinmin Zhong, Qi Huang, Yangrui Chen, Zhi Zhang, Yanghua Peng, Xiang Li, Cong Xie, Shibiao Nong, Yulu Jia, Sun He, Hongmin Chen, Zhihao Bai, Qi Hou, Shipeng Yan, Ding Zhou, Yiyao Sheng, Zhuo Jiang, Haohan Xu, Haoran Wei, Zhang Zhang, Pengfei Nie, Leqi Zou, Sida Zhao, Liang Xiang, Zherui Liu, Zhe Li, Xiaoying Jia, Jianxi Ye, Xin Jin, and Xin Liu. MegaScale: Scaling large language model training to more than 10,000 GPUs. In *NSDI*, 2024.

[29] Vijay Kandiah, Scott Peverelle, Mahmoud Khairy, Junrui Pan, Amogh Manjunath, Timothy G Rogers, Tor M Aamodt, and Nikos Hardavellas. AccelWattch: A power modeling framework for modern GPUs. In *MICRO*, 2021.

[30] Vijay Korthikanti, Jared Casper, Sangkug Lym, Lawrence McAfee, Michael Andersch, Mohammad Shoeybi, and Bryan Catanzaro. Reducing activation recomputation in large transformer models. In *MLSys*, 2023.

[31] Helen Kou and Nathalie Limandibhratha. Power for AI: Easier said than built. https://about.bnef.com/insights/commodities/power-for-ai-easier-said-than-built/, 2025.

[32] Jingwen Leng, Tayler H. Hetherington, Ahmed ElTantawy, Syed Zohaib Gilani, Nam Sung Kim, Tor M. Aamodt, and Vijay Janapa Reddi. GPUWattch: Enabling energy optimizations in GPGPUs. In *ISCA*, 2013.

[33] AI Meta Llama Team. The Llama 3 herd of models. *arXiv preprint arXiv:2407.21783*, 2024.

[34] Runyu Lu, Shiqi He, Wenxuan Tan, Shenggui Li, Ruofan Wu, Jeff J. Ma, Ang Chen, and Mosharaf Chowdhury. TetriServe: Efficiently serving mixed DiT workloads. In *ASPLOS*, 2026.

[35] Deepak Narayanan, Mohammad Shoeybi, Jared Casper, Patrick LeGresley, Mostofa Patwary, Vijay Korthikanti, Dmitri Vainbrand, Prethvi Kashinkunti, Julie Bernauer, Bryan Catanzaro, Amar Phanishayee, and Matei Zaharia. Efficient large-scale language model training on GPU clusters using Megatron-LM. In *SC*, 2021.

[36] NVIDIA. Megatron Core tp_comm_overlap. https://docs.nvidia.com/megatron-core/developer-guide/0.17.0/apidocs/core/core.model_parallel_config.html#core.model_parallel_config.ModelParallelConfig.tp_comm_overlap, 2026.

[37] David Patterson, Joseph Gonzalez, Quoc Le, Chen Liang, Lluis-Miquel Munguia, Daniel Rothchild, David So, Maud Texier, and Jeff Dean. Carbon emissions and large neural network training. *arXiv preprint arXiv:2104.10350*, 2021.

[38] Ryan Prescott. Advanced API performance: SetStablePowerState. https://developer.nvidia.com/blog/advanced-api-performance-setstablepowerstate/, 2022.

[39] Xinwei Qiang, Yue Guan, Zhengding Hu, Keren Zhou, Yufei Ding, and Adnan Aziz. Syncopate: Efficient multi-GPU AI kernels via automatic chunk-centric compute-communication overlap. In *OSDI*, 2026.

[40] Haoran Qiu, Weichao Mao, Archit Patke, Shengkun Cui, Saurabh Jha, Chen Wang, Hubertus Franke, Zbigniew Kalbarczyk, Tamer Başar, and Ravishankar K. Iyer. Power-aware deep learning model serving with μ-Serve. In *ATC*, 2024.

[41] Saurabhsingh Rajput, Alexander Brandt, Vadim Elisseev, and Tushar Sharma. FlipFlop: A static analysis-based energy optimization framework for GPU kernels. In *ICSE*, 2026.

[42] SemiAnalysis. InferenceMAX: Open source inference benchmarking. https://newsletter.semianalysis.com/p/inferencemax-open-source-inference, 2025.

[43] Jovan Stojkovic, Chaojie Zhang, Inigo Goiri, Josep Torrellas, and Esha Choukse. DynamoLLM: Designing LLM inference clusters for performance and energy efficiency. In *HPCA*, 2025.

[44] Qwen Team. Qwen3 technical report. *arXiv preprint arXiv:2505.09388*, 2025.

[45] Brandon Tran, Matthias Maiterth, Woong Shin, Matthew D. Sinclair, and Shivaram Venkataraman. Wattchmen: Watching the wattchers – high fidelity, flexible GPU energy modeling. In *ICS*, 2026.

[46] U.S. Energy Information Administration (EIA). Capital cost and performance characteristics for utility-scale electric power generating technologies. https://www.eia.gov/analysis/studies/powerplants/capitalcost/pdf/capital_cost_AEO2025.pdf, 2024.

[47] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *NeurIPS*, 2017.

[48] vLLM Team. Dual Batch Overlap. https://docs.vllm.ai/en/v0.20.0/design/dbo/, 2026.

[49] Guanhua Wang, Chengming Zhang, Zheyu Shen, Ang Li, and Olatunji Ruwase. Domino: Eliminating communication in LLM training via generic tensor slicing and overlapping. *arXiv preprint arXiv:2409.15241*, 2024.

[50] Shibo Wang, Jinliang Wei, Amit Sabne, Andy Davis, Berkin Ilbeyi, Blake Hechtman, Dehao Chen, Karthik Srinivasa Murthy, Marcello Maggioni, Qiao Zhang, Sameer Kumar, Tongfei Guo, Yuanzhong Xu, and Zongwei Zhou. Overlap communication with dependent computation via decomposition in large deep learning models. In *ASPLOS*, 2023.

[51] Jie You, Jae-Won Chung, and Mosharaf Chowdhury. Zeus: Understanding and optimizing GPU energy consumption of DNN training. In *NSDI*, 2023.

[52] Yiwen Zhang, Xumiao Zhang, Ganesh Ananthanarayanan, Anand Iyer, Yuanchao Shu, Victor Bahl, Z. Morley Mao, and Mosharaf Chowdhury. Vulcan: Automatic query planning for live ML analytics. In *NSDI*, 2024.

[53] Zhendong Zhang, Foteini Strati, and Ana Klimovic. Characterizing energy and performance for distributed training of large language models. In *EuroMLSys*, 2026.

[54] Lianmin Zheng, Zhuohan Li, Hao Zhang, Yonghao Zhuang, Zhifeng Chen, Yanping Huang, Yida Wang, Yuanzhong Xu, Danyang Zhuo, Eric P. Xing, Joseph E. Gonzalez, and Ion Stoica. Alpa: Automating inter- and intra-operator parallelism for distributed deep learning. In *OSDI*, 2022.

[55] Size Zheng, Wenlei Bao, Qi Hou, Xuegui Zheng, Jin Fang, Chenhui Huang, Tianqi Li, Haojie Duanmu, Renze Chen, Ruifan Xu, Yifan Guo, Ningxin Zheng, Ziheng Jiang, Xinyi Di, Dongyang Wang, Jianxi Ye, Haibin Lin, Li-Wen Chang, Liqiang Lu, Yun Liang, Jidong Zhai, and Xin Liu. Triton-distributed: Programming overlapping kernels on distributed AI systems with the Triton compiler. *arXiv preprint arXiv:2504.19442*, 2025.

[56] Size Zheng, Jin Fang, Xuegui Zheng, Qi Hou, Wenlei Bao, Ningxin Zheng, Ziheng Jiang, Dongyang Wang, Jianxi Ye, Haibin Lin, Li-Wen Chang, and Xin Liu. TileLink: Generating efficient compute-communication overlapping kernels using tile-centric primitives. In *MLSys*, 2025.

[57] Kan Zhu, Yufei Gao, Yilong Zhao, Liangyu Zhao, Gefei Zuo, Yile Gu, Dedong Xie, Tian Tang, Qinyu Xu, Zihao Ye, Keisuke Kamahori, Chien-Yu Lin, Ziren Wang, Stephanie Wang, Arvind Krishnamurthy, and Baris Kasikci. NanoFlow: Towards optimal large language model serving throughput. In *OSDI*, 2025.

A Energy Efficiency of Constant Frequency

Below, we formalize why steady GPU frequency minimizes energy at a fixed average frequency.

Theorem 1 (Energy Efficiency of Constant Frequency)**.** *Let $f(t)$ denote the GPU frequency over a time interval $[0,T]$, and let $\bar{f} = \frac{1}{T}\int_0^T f(t)\,dt$ be its time-average. Under the following assumptions:*

1. *Dynamic power scales cubically with frequency: $P_{dyn}(t) = k \cdot f(t)^3$ for some constant $k > 0$.*
2. *Static power is constant: $P_{static}(t) = P_s$ for some constant $P_s \geq 0$.*
3. *Execution time depends only on average frequency: workloads with the same average frequency $\bar{f}$ complete in the same time T. This is supported by empirical observations from Kareus optimization results.*

Then the total energy consumption is minimized when frequency is held constant at $\bar{f}$.

Proof. Total energy is the sum of dynamic and static energy:

$$E_{\text{total}} = E_{\text{dyn}} + E_{\text{static}} = \int_0^T k \cdot f(t)^3\,dt + P_s \cdot T.$$

By Assumption 3, T is identical for both the fluctuating and constant frequency cases. Thus, static energy $E_{\text{static}} = P_s \cdot T$ is equal in both cases.

For dynamic energy, consider the function $g(x) = x^3$, which is convex for $x > 0$ since $g''(x) = 6x > 0$. By Jensen's Inequality:

$$g(\bar{f}) = \bar{f}^3 \leq \frac{1}{T}\int_0^T f(t)^3\,dt.$$

Multiplying both sides by kT:

$$E_{\text{dyn}}^{\text{constant}} = kT \cdot \bar{f}^3 \leq k\int_0^T f(t)^3\,dt = E_{\text{dyn}}^{\text{fluctuating}}.$$

Equality holds if and only if $f(t) = \bar{f}$ almost everywhere, i.e., when frequency is constant. Therefore:

$$E_{\text{total}}^{\text{constant}} \leq E_{\text{total}}^{\text{fluctuating}},$$

with strict inequality when $f(t)$ varies over time. □

Implications for GPU frequency scaling. This theorem explains why frequency fluctuation due to power limits can be energy-inefficient. When a GPU operates at a high frequency and triggers power throttling, the instantaneous frequency fluctuates, while the time-averaged frequency remains close to what could be achieved by operating steadily at a lower frequency. Consequently, the execution time—and thus static energy consumption—are the same as those under steady lower-frequency operation. However, this fluctuation incurs higher dynamic energy consumption because dynamic power is a strictly convex function of frequency. By Jensen's Inequality, any variance in frequency increases the expected dynamic energy consumption.

B Global Solution Space

The global solution space of combining GPU frequency, SM allocation, and kernel launch timing is extremely large. In this section, we illustrate the space using the NVIDIA A100 GPU as an example.

GPU frequencies. Supported GPU frequencies by A100 are from 210 MHz to 1,410 MHz at a stride of 15 MHz. We restrict the search space to 900 MHz–1,410 MHz (35 choices) because lowering the frequency below 900 MHz no longer reduces energy.[11]

SM allocations. A100 has 108 SMs, and we restrict the search space to up to 30 SMs because allocating additional SMs beyond this empirically no longer improved communication latency. This leaves 30 choices for SM allocation.

Kernel launch timing. Finally, given a sequence of computation and communication kernels, possible execution orders can be expressed as a recurrence relation, which enumerates the number of subproblems. We elaborate on this formulation in the following paragraphs. In summary, for a typical LLM composed of Transformer blocks, this can result in 81 possible groupings, leading to a *global solution space* with in total 85,050 candidates.

Formulation of launch timing. Kareus generalizes the nanobatching scheme to eliminate dependencies between computation and communication, which enables overlap and results in two operation sequences:

$$\mathcal{S}_1 = \{O_0, O_1, \ldots, O_i\}, \qquad \mathcal{S}_2 = \{O'_0, O'_1, \ldots, O'_j\}.$$

where operations within the same sequence must be executed in order, while operations across sequences are independent and may overlap. We regard the consecutive communication operations as a single operation. The time–energy frontier of the launch timing schedule can be formulated as a dynamic programming (DP) recurrence:

$$\mathcal{P}(i,j) = \min_{\text{Pareto}} \begin{cases} (E(O_i) + E(\mathcal{P}(i+1,j)),\ T(O_i) + T(\mathcal{P}(i+1,j))) \\ (E(O'_j) + E(\mathcal{P}(i,j+1)),\ T(O'_j) + T(\mathcal{P}(i,j+1))) \\ (E(O_i \parallel O'_{j..j+k}),\ T(O_i \parallel O'_{j..j+k})) + \mathcal{P}(i+1,j+k) \\ (E(O'_j \parallel O_{i..i+k}),\ T(O'_j \parallel O_{i..i+k})) + \mathcal{P}(i+k,j+1) \end{cases}$$

Here, $O_i \parallel O'_{j..j+k}$ means overlapping operation O_i with a subsequence $\{O'_j, \ldots, O'_{j+k}\}$; the notation is symmetric for the opposite case, $O'_j \parallel O_{i..i+k}$. The cost functions $E(\cdot)$ and $T(\cdot)$ account for the interference when overlapping.

This recurrence enumerates the time and energy of all single operations and all feasible overlap patterns between the

[11] Power reduction slows down but time increases, leading to higher energy.

two sequences. For a typical Transformer block with 9 computation operations and 1 AllReduce, if we restrict overlap to occur only between communication and computation and cap the maximum overlap length at 9 (assuming the communication is no longer than a full Transformer block), there are 81 possible overlap patterns. Including the non-overlapped cases, this yields a total of 91 subproblems.

C Adaptation of MBO Hyperparameters

In this section, we describe how Kareus configures MBO hyperparameters. Some are adapted across different partitions based on partition complexity, while others remain fixed.

Search space. For GPU frequency, we restrict the search space to 900–1,410 MHz with a stride of 30 MHz, since the time and energy differences between adjacent 15 MHz settings are marginal. For SM allocation, the search space is determined by the communication group size. If the number of GPUs in a communication group is fewer than 4, we search SM allocations from 1 to 20 with a stride of 1. If the group size is 4 or larger (e.g., the common cases of 4 and 8), we search SM allocations from 3 to 30 with a stride of 3. For launch timing, we enumerate all computation operators within a partition and exclude options that always lead to exposed communication, such as launching AllReduce from Linear2 in Figure 3a.

Sample size. Our goal is to control the total profiling time while preserving the quality of the time–energy frontier. We adopt a relatively large initialization set for informative exploration and scale the batch search budget based on partition complexity. Specifically, we classify partitions into three categories: *small* partitions containing only one computation, *medium* partitions containing 2–3 computations, and *large* partitions containing more than three computations. We then configure the sample sizes as follows: initial sample size $N_{\text{init}} = 36$, maximum number of batches $B_{\max} = 3$, batch size $k = 16$ for small partition; $N_{\text{init}} = 48$, $B_{\max} = 4$, $k = 16$ for medium partition; $N_{\text{init}} = 96$, $B_{\max} = 4$, $k = 32$ for large partition. We set the proportions of the total energy pass, dynamic energy pass, static energy pass, and uncertainty pass to 0.4, 0.2, 0.2, and 0.2, respectively. As shown in Section 6.6, the MBO overhead of Kareus is controlled within two hours on average.

XGBoost hyperparameters. Since the configuration space is three-dimensional, we adopt XGBoost hyperparameter settings commonly used for low-dimensional regression. To ensure fast and stable convergence while avoiding overfitting, we use relatively shallow trees with $max_depth = 6$, a high learning rate $\eta = 0.3$, and a modest model capacity with $num_boost_round = 100$. For the bootstrap ensemble, we set the ensemble size to 5, use a bootstrap sampling fraction of 0.8, and vary the random seed across bootstrap resamples.

HV reference point. At each batch iteration, we compute the HV reference point using values slightly worse than the worst observed measurements to ensure boundedness. Specifically, we set $r = \big(1.1 \times \max T(x),\ 1.1 \times \max E(x)\big)$.

Stopping conditions. The MBO algorithm is terminated when either a sufficient number of samples has been collected or the objective has converged, according to the following criteria: (1) Stop after a fixed number of batches $B_{\max}$. (2) After each batch, compute the dominated hypervolume of the measured frontier. Stop if the moving average of the relative HV improvement over the last R batches falls below ε.

We set the window size in the stopping criterion to $R = 2$ and the convergence threshold to $\varepsilon = 10^{-3}$. Energy and time are normalized throughout the MBO process, ensuring stable stopping behavior across partitions.

D End-to-End Results

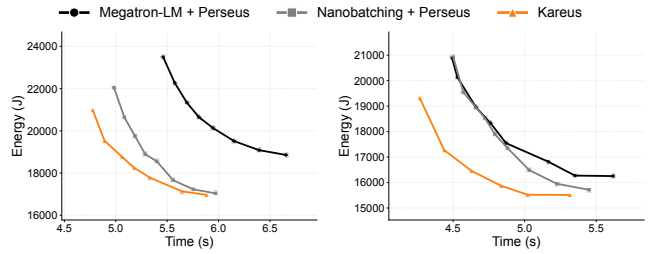
Figure 13 shows the iteration time–energy frontiers measured on the testbed GPUs for all model configurations, complementing the evaluation in Section 6.2.2.

E Large-Scale Emulation

Figure 14 shows the iteration time–energy frontiers in large-scale emulation for Llama 3.3 70B across different numbers of microbatches, complementing the evaluation in Section 6.3.

F Sensitivity to Microbatch Size

Figure 15 shows the iteration time–energy frontiers measured on the testbed GPUs for Qwen 3 1.7B with TP=8 and sequence length 4K across different microbatch sizes, complementing the sensitivity study in Section 6.5.



(a) *Llama 3.2 3B, CP=1, TP=8, μBS=8, Seq=4096*

(b) *Llama 3.2 3B, CP=2, TP=4, μBS=8, Seq=4096*

(c) *Llama 3.2 3B, CP=2, TP=4, μBS=8, Seq=8192*

(d) *Llama 3.2 3B, CP=2, TP=4, μBS=16, Seq=4096*

(e) *Microbatch size = 8*

(f) *Qwen 3 1.7B, CP=1, TP=8, μBS=8, Seq=8192*

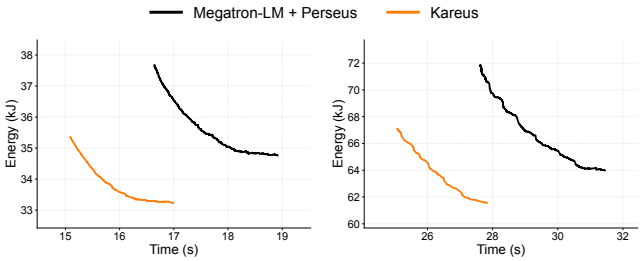
(g) *Qwen 3 1.7B, CP=1, TP=8, μBS=16, Seq=4096*

(h) *Qwen 3 1.7B, CP=2, TP=4, μBS=8, Seq=4096*

(i) *Qwen 3 1.7B, CP=2, TP=4, μBS=8, Seq=8192*

(j) *Qwen 3 1.7B, CP=2, TP=4, μBS=16, Seq=4096*

Figure 13: ***[Experiment] Itertaion time–energy frontiers of Megatron-LM + Perseus, Nanobatching + Perseus, and Kareus for all model configurations. CP: Context Parallelism, TP: Tensor Parallelism, μBS: Microbatch Size, Seq: Sequence Length.***



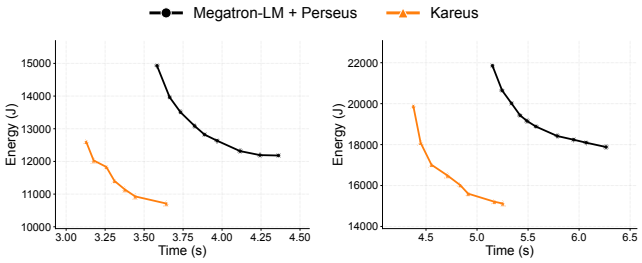
(a) *# Microbatches = 16*

(b) *# Microbatches = 32*

(c) *# Microbatches = 64*

(d) *# Microbatches = 128*

Figure 14: ***[Emulation] Iteration time–energy frontiers of Megatron-LM + Perseus and Kareus for Llama 3.3 70B across different numbers of microbatches.***



(a) *Microbatch size = 8*

(b) *Microbatch size = 12*

(c) *Microbatch size = 16*

(d) *Microbatch size = 20*

Figure 15: ***[Experiment] Iteration time–energy frontiers of Megatron-LM + Perseus and Kareus for Qwen 3 1.7B across different microbatch sizes.***

SPADE: Signal-Aware DAG Scheduling and Dynamic Provisioning for Data Processing Clusters

Adam Lechowicz
University of Massachusetts Amherst

Rohan Shenoy
University of California Berkeley

Noman Bashir
Massachusetts Institute of Technology

Mohammad Hajiesmaili
University of Massachusetts Amherst

Adam Wierman
California Institute of Technology

Christina Delimitrou
Massachusetts Institute of Technology

Abstract

As AI-driven demand reshapes the data center landscape, external signals—such as energy cost, carbon intensity, power availability, and water usage—are increasingly dictating how much compute is available at any moment. These signals tend to vary over time, challenging traditional cluster schedulers, which implicitly assume stable resource supply, and calls for systems that continuously adapt to time-varying conditions. We focus on batch data-processing workloads, which are delay-tolerant but constitute a healthy fraction of total compute, making them a natural target for such flexibility. The directed acyclic graph (DAG) structure of these data-processing jobs makes decisions uniquely challenging, since delaying certain tasks in the DAG (e.g., bottleneck tasks) can stall entire pipelines. We introduce SPADE, a signal-aware scheduling and provisioning system that jointly considers workload DAG structure and external time-varying signals when deciding how (provisioning) and when (scheduling) to allocate resources. To underscore the importance of coupling these decisions, we evaluate SAP, an ablated system that preserves SPADE's signal-aware provisioning but delegates scheduling to arbitrary signal-agnostic policies. Using a Spark prototype deployed on a 100-node Kubernetes cluster, we show that SPADE reduces a secondary objective (e.g., the cost associated with carbon intensity or energy price) by 32.9% while maintaining overall cluster throughput.

1 Introduction

Data centers are expanding at an unprecedented pace. Global demand for computing infrastructure is projected to more than double this decade [63], with energy consumption expected to rise by over 165% [61]. Yet this growth is increasingly colliding with physical and environmental limits. Several recent high profile disputes, from reports of reduced water pressure and degraded water quality [67] to concerns about increased air pollution associated with a data center [14], highlight that cluster performance can now be constrained as much by external conditions as by internal hardware capacity. As these external signals begin to dictate how much compute capacity a data center can responsibly or affordably use at any moment, cluster management can no longer optimize solely for classic, internally visible objectives such as makespan or latency. Instead, schedulers must continuously adapt execution to conditions that originate outside the cluster itself.

Supporting adaptiveness to these external conditions represents a substantial departure from the status quo. Rather than treating the cluster as an isolated system with fixed resources, a *signal-aware* system must decide how to provision resources, not just when (via scheduling) to allocate them. This shift requires mechanisms that react to external signals and incorporate them directly into job scheduling decisions, allowing clusters to adjust resource usage in response to evolving environmental and operational constraints.

A key challenge is that these external signals are inherently *time-varying*. Carbon intensity evolves with the generation mix on the grid; water demand and cooling efficiency change with ambient temperature [44]; and grid health, often inferred from indicators such as locational marginal prices, varies as a function of power generation and demand [5]. To make response to these dynamics actionable, recent work such as Middlebox [48] proposes receiving (and acting on) a *time-varying signal* broadcast by entities like grid operators [16,18] that indicates when conditions are favorable or unfavorable. The high-level goal is to scale down resource usage during high-cost or high-impact periods and scale up when conditions improve.

Not all workloads are equally suited to respond to such signals. Interactive services are typically not temporally flexible enough to meaningfully shift their demand in response to an external signal. In contrast, non-interactive workloads centrally coordinated by the data center are natural candidates. Prior work has shown that regularly scheduled, highly parallelizable ML training jobs with predictable SLOs can be resized over time to track signals such as carbon intensity without significant performance loss [33]. This suggests a broader opportunity: can we systematically expand these signal-aware concepts to *batch data-processing workloads*?

We focus on batch data-processing jobs (e.g., Apache Spark), which constitute a substantial fraction of data center compute and are often delay-tolerant [1]. These workloads are expressed as directed acyclic graphs (DAGs) of precedence-constrained tasks: the outputs of one stage become the inputs of another [75]. Stages exhibit heterogeneity, with some benefiting from parallelism while others do not [53]. Many production systems already expose these dependencies explicitly in DAG form [11, 36, 59]. To execute these jobs, DAG schedulers (and cluster schedulers broadly) look "internally" to understand dependencies and user requirements when deciding how to schedule and assign resources to the job. However, these schedulers are largely *signal-agnostic*: they optimize for makespan or throughput under an implicitly stable resource supply and do not respond to external signals.

In parallel, several lines of work have proposed signal-aware provisioning schemes [33, 72] that scale resources up and down to leverage temporal signal variations. However, these do not consider "internal" characteristics such as DAG dependencies. In the setting of shared data-processing clusters, this separation is problematic: scaling resources down at the wrong time can stall critical paths and propagate delays, while scaling up prematurely can leave resources idle. As a result, a mechanism that only adapts resources in response to the signal while ignoring DAG structure and job interactions can degrade *both* performance and the secondary objective imposed by the external signal.

DAG structure introduces a three-dimensional trade-off between (i) classic performance metrics such as makespan, (ii) a secondary objective defined by the external signal (e.g., carbon emissions, energy cost, or power overloading), and (iii) how granularly the system engages with the DAG structure of jobs, ranging from *black-box* treatments that defer or scale jobs as a whole to *granular* task-level scheduling that exploits internal dependencies—see Fig. 1 for an illustration of these dynamics for an example job. This third axis is not a free parameter: a black-box treatment of signal-awareness is straightforward to bolt onto an existing scheduler but sacrifices performance on (i) and/or (ii), while granular, task-aware scheduling requires deeper integration and yields a more favorable trade-off on the first two axes.

This motivates a notion of *signal-aware DAG execution*: jointly coordinating task scheduling and resource provisioning as a function of both DAG structure and external time-varying signals. While prior signal-aware systems have identified a trade-off between performance and secondary objectives, they typically operate at coarse granularity (e.g., job-level scaling with deadlines) and do not consider DAG structure. Conversely, systems that incorporate DAG structure (e.g., RL-based schedulers for Spark) do not consider external signals. Existing work that considers both DAGs and signals, such as Caribou [32], targets serverless workflows and exploits spatial variations across regions rather than temporal variation within a single data center. One might hope to merge these two threads simply by composing them, e.g., by layering a signal-aware provisioning policy on top of an off-the-shelf DAG scheduler, or vice versa. This decomposition is insufficient because the two decisions are tightly coupled: whether deferring a given task is beneficial depends jointly on the task's position in the DAG and the current signal value, since deferring a bottleneck task stalls every downstream stage even when the signal momentarily favors doing so. We confirm this empirically in § 5 via our SAP ablation, which retains SPADE's signal-aware provisioning but delegates scheduling to signal-agnostic policies and consistently realizes a strictly worse trade-off between throughput and the secondary objective.

This paper: SPADE. In this work, we ask: *How should a cluster scheduler execute delay-tolerant DAG-structured batch jobs in response to a time-varying external signal?* We introduce SPADE (**S**cheduling and **P**rovisioning for **A**daptive **D**AG **E**xecution), a signal-aware system that unifies scheduling and provisioning for DAG jobs. SPADE introduces the notion of *relative importance*, which allows the system to account for both DAG structure and the dynamics of a time-varying signal. This enables SPADE to make fine-grained trade-offs—for example, continuing to prioritize certain tasks during periods when the signal suggests reducing resource consumption—so that clusters can respect secondary objectives without unnecessarily stalling critical paths. SPADE exposes a single *signal-awareness parameter* to control how much it is willing to sacrifice in throughput to align with a time-varying signal—rather than specifying a deadline for each job, an operator can specify how SPADE prioritizes objectives.

To underscore the importance of coupling these decisions, we also study an ablation, SAP (**S**ignal-**A**ware **P**rovisioning), which varies the available resources while delegating scheduling to a signal-agnostic policy. SAP captures the intuition of scaling down during high-signal periods and vice-versa, but because it ignores inter-task dependencies, its performance is strictly worse: the trade-off it achieves between makespan and the secondary objective is dominated by SPADE.

We implement SPADE and SAP as modules for Spark on Kubernetes and as extensions to a high-fidelity Spark simulator [53]. Our experiments consider real and synthetic workloads from Alibaba traces and TPC-H [2, 69]. We replay several signals, including carbon intensity traces from six regions [23] and available power traces from Google [60]. In our prototype, we implement SPADE and SAP on a 100-node Spark cluster. We report the impact of different signals on the makespan and cost of tested schemes, showing that SPADE's configurable design and joint scheduling and provisioning enables a superior trade-off between the secondary objective and makespan. Our key contributions are as follows:

1. SPADE, a signal-aware DAG execution system that defines a notion of relative importance for tasks, allowing for fine-grained decisions. SPADE jointly considers this importance

with an external signal for each decision, achieving a superior trade-off between cost and performance.

2. We implement SPADE and SAP in a Spark-on-Kubernetes prototype and a realistic Spark simulator (see § 4).
3. A comprehensive evaluation against baselines and a state-of-the-art ML-based scheduler. We show that SPADE achieves significant improvements in secondary objectives (e.g., 32.9% lower carbon emissions and 51% better alignment with an available power signal) while maintaining competitive makespan.

2 Problem and Motivation

This section formalizes the signal-aware DAG scheduling problem and presents insights to contextualize our desiderata.

2.1 Signal-aware DAG scheduling problem

Each job (i.e., a data processing workflow) is represented as a directed acyclic graph (DAG) $\mathcal{J} = \{\mathcal{V}, \mathcal{E}\}$, where each node in $\mathcal{V}$ is one of n tasks, and each edge in $\mathcal{E}$ encodes precedence constraints between tasks, e.g., for tasks $j, j' \in \mathcal{V}$, an edge $j \to j'$ indicates that j' cannot start until after j has completed. A cluster includes $K \geq 1$ executors (or machines). More than one job can simultaneously run on a cluster—e.g., given a set of current jobs $\{\mathcal{J}\}$, the scheduler assigns tasks to executors over time while respecting precedence and capacity constraints. We index continuous time by $t \geq 0$.

The traditional goal is to minimize the *makespan* of a schedule, which is the total time to complete all jobs. In this work, we additionally consider the goal of *signal-awareness*: given a time-varying signal described by $s(t) : t \geq 0$, a signal-aware system's objective is to minimize a combination of typical metrics (i.e., makespan, average job completion time) and the total cost (due to the signal) incurred during execution.

We generally assume that future values of the time-varying signal $s(t)$ are unknown to the system, giving the problem an online nature. We follow prior work [7, 51] and place one minimal assumption on the signal, namely that the signal values are bounded by constants $s_{\min} \leq s(t) \leq s_{\max}$ that are known in advance. In practice, these can be defined based on historical data or short-term forecasts of the signal of interest over a given time horizon. We assume without loss of generality that the system attempts to minimize the signal cost, i.e., so $s_{\min}$ is a "good" signal, and vice versa.

We note a subtle distinction between *signal-awareness* (treating the signal as a secondary objective, as in SPADE) and *signal-adherence* (treating the signal as a hard constraint, e.g., instantaneous power available to the cluster). The two are complementary: when power availability is genuinely constrained due to curtailment or contention, a separate enforcement layer is typically responsible for arbitrating capacity across workloads (e.g., [16]). SPADE's role is to schedule *within* whatever capacity is currently available, and its ideas apply whether the cluster voluntarily uses fewer resources during high-signal periods or is involuntarily curtailed by such an enforcement layer; in both cases, SPADE optimizes makespan with the knowledge that resources may vary over time.

Our code is available at github.com/umass-solar/adaptive-dag.

2.2 Guiding intuition and preliminaries

Modern data-processing frameworks (e.g., Spark) express jobs as DAGs of precedence-constrained stages and, by default, schedule them using straightforward strategies such as first-in, first-out (FIFO) or fair-share [25]. To exploit DAG structure, recent heuristic and reinforcement learning (RL)-based schedulers [30, 31, 37, 43, 53, 74, 77] rank ready tasks by a score or probability distribution—a representation that SPADE leverages directly.

Compared to a classic (i.e., *signal-agnostic*) scheduler, we have two primary metrics of interest: *stretch factor*, which describes the increase in makespan due to signal-aware actions, and *cost savings*, which describes the improvement on the secondary objective due to changes in the schedule. We give formal definitions of both quantities in Appendix B. Between these two metrics, there is an inherent trade-off between e.g., executing a task during the current signal and waiting for potential better signals at the expense of increasing makespan.

This trade-off inherits challenges that are familiar in the online decision-making literature, specifically problems known as *online search*—in these problems, the player must purchase an asset at a price that varies with time [22, 51]. These problems have natural applications in, e.g., financial markets, but they have also been used to model systems problems such as energy cost minimization for data centers [41]. A key challenge in these problems is the inherent uncertainty of future signal values (i.e., prices). The common method to handle this uncertainty is a technique known as *threshold-based design* [6, 40, 72], which defines a threshold function that is used to guide decision-making at each time step.

In the context of our two key metrics, we translate the trade-off between executing and waiting into three basic conditions that a signal-aware system should satisfy, outlined below. In Fig. 1, we give an illustration of this "desired behavior" for SPADE, FIFO, and optimal schedules.

i) If the fluctuation of the time-varying signal is *low* (e.g., $s_{\min}$ and $s_{\max}$ are close), makespan should be similar to a signal-agnostic schedule (i.e., stretch factor close to 1).

ii) If the fluctuation is *high* (e.g., $s_{\min}$ and $s_{\max}$ are not close), the stretch factor should be *finite*, i.e., the scheduler does not wait indefinitely to complete the job.

In existing work on online search and signal-aware schedulers, schemes often satisfy condition **ii)** by imposing a *deadline* on each job, forcing it to execute when the deadline is near [7, 28, 33]. When applicable, per-job deadlines can be a more user-friendly interface than tuning a trade-off parameter, but prior signal-aware deadline-based schemes generally

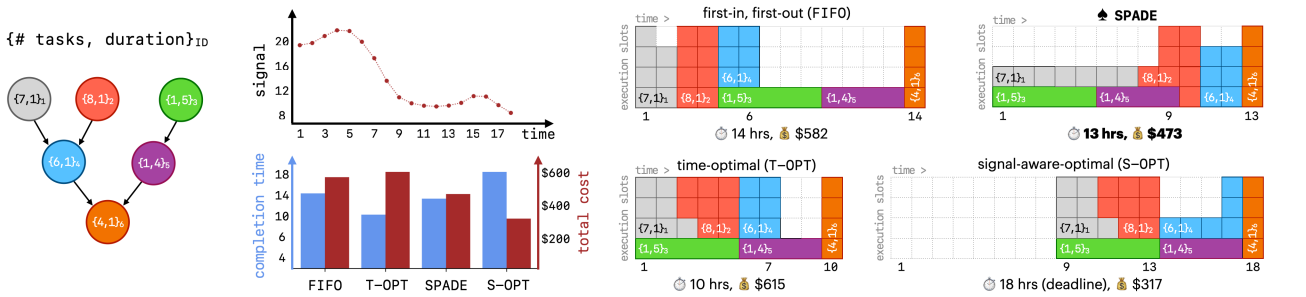


Figure 1: Four schedules for a motivating DAG and 18-hour signal trace (left hand side). When the signal is high, computations are more costly. Compared to an agnostic FIFO scheduler, the time-optimal scheme (T-OPT) prioritizes the green and purple stages of the DAG to reduce overall makespan. A signal-aware-optimal scheme (S-OPT) with a *deadline* to finish the DAG within 18 hours reduces cost by 51.2%, at the expense of increasing time by 28.5% compared to FIFO. By prioritizing green and purple stages when signal is high, SPADE reduces cost by 23.1% and still completes 7% faster than FIFO in this scenario.

consider only one job at a time. Extending this interface to a multi-tenant batch cluster is non-trivial: when multiple jobs with similar deadlines arrive simultaneously and resources are limited, all schedules (signal-aware or otherwise) may be unable to honor every deadline. To address this challenge, our schemes (see § 3) are designed to guarantee a *minimum amount of progress* on jobs as long as the queue is not empty.

The unique precedence-constrained nature of the DAG scheduling problem introduces one additional challenge to consider in our design. In particular, a signal-aware scheduler that does not consider the structure of task graphs may make actions that block important bottleneck tasks from processing, resulting in an outsized negative impact on makespan. This gives a natural third condition:

iii) When the fluctuation is *high* (i.e., $s_{\min}$ and $s_{\max}$ are not close) and the system is in a high-signal period, a scheduler should carefully consider the *structure* of a job's DAG, prioritizing *bottleneck* tasks to use the limited resources.

This condition captures the idea that signal-aware DAG execution must consider the trade-off between executing a task now (unlocking tasks that depend on it sooner), and waiting to execute the task until the signal (may) improve. To approach this trade-off, we combine the notion of threshold-based design, which allows for a rigorous characterization of the trade-off between e.g., executing a given task and waiting for better signals, with the notion of score- or probability-based task prioritization seen in state-of-the-art DAG schedulers. In the following section, we present SPADE, our main system that satisfies conditions **i - iii)** and captures the desired behavior of signal-aware execution for DAG workloads.

3 Design

In this section, we present SPADE (Scheduling and Provisioning for Adaptive DAG Execution), our main system. We also define an ablation study, SAP (Signal-Aware Provisioning) to demonstrate the benefits of the joint scheduling and provisioning decisions made by SPADE.

3.1 SPADE

From the discussion in § 2.2, we seek an interpretable and configurable system that satisfies conditions **i - iii)**. To this end, we introduce SPADE (**S**cheduling and **P**rovisioning for **A**daptive **D**AG **E**xecution). SPADE's key idea is a metric of *relative importance* (Def. 3.2) implicitly embedded in a score distribution or probability distribution over tasks SPADE uses this metric, combined with a configurable signal-aware threshold function, to make fine-grained, per-task scheduling decisions, as illustrated in Fig. 2.

SPADE design. We first define a notion of *score or probability distributions* over tasks that SPADE uses to interpret DAG structure.

Definition 3.1 (Score or Probability Distribution)**.** *At each* ***scheduling event****,*[1] *a score or probability distribution is described by* $\mathcal{D}(t) := \{p_{v,t} : v \in \mathcal{A}_t\}$*, where* $\mathcal{A}_t$ *denotes the set of tasks that are ready to be executed at time* t*.*

Several state-of-the-art DAG schedulers use a variety of heuristic or learning-based techniques to compute score or probability distributions that satisfy Def. 3.1, including Graphene [30], Decima [53], and others [31, 43, 74, 77]. Two notes on this abstraction. *(i)* These score distributions are typically computed over *all* ready-to-execute tasks across *all* active jobs (not per-job) so $\mathcal{D}(t)$ already encodes cross-job prioritization. *(ii)* How the scores are computed varies (Decima learns them via reinforcement learning; Graphene combines critical-path length, packing efficiency, and priority), but SPADE treats the scorer as a black box with one working assumption: high-scoring tasks correspond to bottleneck tasks in the makespan sense. This assumption is inherited: if the scorer fails to identify bottlenecks, it would already perform poorly on makespan, even without any signal-awareness layered on top. Recall from § 2.2 (condition **iii)**) that bottleneck tasks (those with a large score or probability in the above distribution) should be scheduled even during high-signal periods to

[1] Scheduling events include job arrivals, task completions, and executors becoming available.

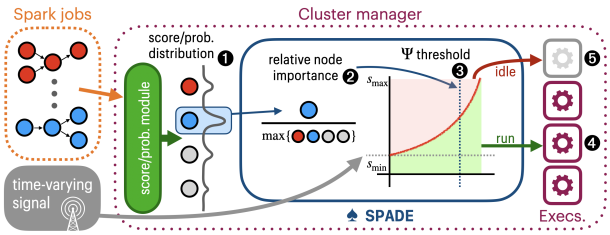


Figure 2: SPADE takes a probability (or score) distribution over tasks as input. Given such a distribution ❶, SPADE computes a *relative importance* score ❷ that is used to determine which tasks should run based on the current signal ❸—e.g., bottleneck tasks run regardless of the signal ❹, while less important ones are deferred to later (low-signal) periods ❺.

preserve makespan. To this end, we formally define a notion of *relative importance* that compares the score/probability assigned to a single task v against other tasks in $\mathcal{A}_t$.

Definition 3.2 (Relative Importance)**.** *Given a time $t \geq 0$ and task $v \in \mathcal{A}_t$, the relative importance $r_{v,t}$ is defined as:*

$$r_{v,t} \coloneqq \frac{p_{v,t}}{\max_{u \in \mathcal{A}_t} p_{u,t}} \in [0,1].$$

If a task's relative importance is closer to 1, the task is relatively *more important*, and a value closer to 0 implies the opposite. Note that if $|\mathcal{A}_t| = 1$ (i.e., only one task can be scheduled), the importance of that task is always 1.

Recall that the goal is to optimize a combination of performance objectives (e.g., makespan, JCT) and a secondary objective that is defined by a time-varying signal. Using the metric of relative importance, we define a *scheduling filter* using a threshold function Ψ_γ that considers the current signal and the importance of a task. $\gamma \in [0,1]$ is a user-specified signal-awareness parameter that controls how "strictly" this function follows the time-varying signal: $\gamma = 0$ recovers signal-agnostic scheduling, while $\gamma = 1$ is maximally signal-aware.

$$\Psi_\gamma(r) = (\gamma s_{\min} + (1-\gamma)s_{\max}) + [\gamma s_{\max} - \gamma s_{\min}] \frac{\exp(\gamma r) - 1}{\exp(\gamma) - 1},$$

Ψ_γ is an exponential function of the relative importance of a task r—this draws on literature from online search [22,78] and captures a trade-off between executing a given task now at the current signal, and the risk that better signals exist in the future. The choice of an exponential inherits a risk-aversion intuition from online search problems such as one-way trading [22]: in these problems, a player who must purchase N units before a deadline at time-varying prices optimally accepts the first unit at a relatively loose threshold (the risk of paying a worst-case price later for the bulk of remaining units dominates) and tightens the threshold exponentially as more units are bought. The optimal reservation-price curve in this class of problems is provably exponential. In our setting, the analogy is as follows: a task with low relative importance (a "first unit") should be deferred only when the signal is unusually bad, since deferring it risks stalling downstream tasks; a task

Algorithm 1 SPADE (Scheduling and Provisioning for Adaptive DAG Execution)

1: **input:** user-specified parameter γ, threshold function Ψ_γ, score/probability distribution $\mathcal{D}(t)$
2: **define:** *invocation* occurs when the signal $s(t)$ changes or when a scheduling event happens
3: **while** cluster active at time $t \geq 0$ **do**
4: **if** invocation at time t **then**
5: Sample $v \in \mathcal{A}_t$ from score/prob. distribution $\mathcal{D}(t)$
6: Compute relative importance $r_{v,t} = \frac{p_{v,t}}{\max_{u \in \mathcal{A}_t} p_{u,t}}$
7: **if** $\Psi_\gamma(r_{v,t}) \geq s(t)$ **or** no executors currently busy **then**
8: Send task v to an available machine at time t
9: **else**
10: Idle until next invocation

with high relative importance (a "last unit") is admitted at almost any signal because the cost of stalling its dependents outweighs paying a high signal cost. [2] We formalize the usage of this function in Algorithm 1. Note that if the input $\mathcal{D}$ is a score distribution, we first use a softmax operation to sample from the corresponding probability distribution over tasks (line 5).

SPADE's signal-awareness filter accomplishes all three of the motivation points defined in § 2.2. It schedules (or defers) tasks based on the current signal $s(t)$, with the effect of reducing execution during high-signal periods. Furthermore, the likelihood of a task being scheduled irrespective of the current signal is proportional to its importance in its DAG. Note that $\Psi_\gamma(1) = s_{\max}$, which means tasks with high relative importance are always scheduled. The choice to *sample* from $\mathcal{D}(t)$ (line 5) rather than taking the argmax is essential: the argmax always has relative importance 1, and $\Psi_\gamma(1) = s_{\max}$, so such a policy would never defer tasks. Sampling lets the algorithm consider lower-scoring tasks proportionally to their probability—when the signal is high, the threshold filter defers these candidates; when the signal is low, they are admitted, opportunistically using available capacity. The bottleneck task is never starved (if sampled, its relative importance of 1 guarantees admission), while low-importance tasks are selectively held back during costly periods. While deferring a task has a negative impact on an individual job's completion time, prioritizing bottleneck tasks across all active jobs allows SPADE to manage the system's overall makespan. To ensure progress on jobs, note that tasks are also scheduled if the entire cluster would otherwise become idle (line 6).

Inherited properties from the underlying scorer. SPADE does not directly enforce fairness, priority classes, or similar cross-job properties; instead, it inherits these from whichever

[2] In our experiments, we briefly considered linear and quadratic threshold functions (normalized to the same range). Threshold shape plays a role similar to γ: a linear threshold sits above the exponential one, so it defers fewer tasks and produces correspondingly smaller signal savings and smaller makespan increases. Because varying the threshold's shape is well-approximated by simply re-tuning γ, we report only the exponential variant.

technique produced the input score/probability distribution $\mathcal{D}(t)$. The relative-importance transform in Def. 3.2 is a per-event normalization by $\max_{u \in \mathcal{A}_t} p_{u,t}$, so it preserves the ordering induced by the scorer: two tasks assigned identical scores are treated identically by SPADE. If the scorer is fairness-aware (e.g., Graphene assigns scores across *all* active jobs to prevent starvation), SPADE preserves that property: rather than greedily picking the top-scoring task, line 5 of Algorithm 1 *samples* from $\mathcal{D}(t)$, which introduces additional randomness across jobs and intuitively maintains or improves the underlying fairness profile. The same observation extends to weighted fairness or priority classes encoded in $\mathcal{D}(t)$.

In Fig. 3, we illustrate the intuitions behind SPADE's signal-awareness filter using two sample job DAGs. In Appendix B.1, we prove that for any signal $\mathbf{s}$ and given K executors, the stretch factor of SPADE is upper bounded by $1 + \frac{\texttt{exp.defs.}(\gamma,\mathbf{s})\cdot K}{2-1/K}$, where $\texttt{exp.defs.}(\gamma,\mathbf{s}) \in [0,1]$ is the expected fraction of tasks deferred under signal $\mathbf{s}$ with parameter γ. Two endpoints anchor the intuition: $\texttt{exp.defs.}(0,\mathbf{s}) = 0$ for any $\mathbf{s}$, so $\gamma = 0$ recovers the underlying scheduler's makespan, and $\texttt{exp.defs.}(\gamma,\mathbf{s}) \leq 1$ always, so the bound remains finite even at $\gamma = 1$. A closed form for general $\gamma, \mathbf{s}$ is not available, but exp.defs. grows monotonically with γ—matching the experimental trends in § 5.

Complementary operator-facing knobs. Beyond γ, SPADE exposes two more design features that allow an operator to control the trade-off between makespan and signal-awareness. *i) Minimum throughput.* Line 7 of Algorithm 1 already requires at least one executor to remain active regardless of $s(t)$, guaranteeing continuous progress on the queue. This constraint generalizes naturally: an operator can mandate that some fraction $\rho \in (0,1]$ of the cluster always runs, regardless of the signal value. A larger ρ keeps more of the cluster productive during high-signal periods (favoring makespan at the cost of signal-responsiveness), while a smaller ρ lets SPADE defer more aggressively. *ii) Target deadlines.* SPADE can also accommodate a relaxed deadline interface: each job is submitted with a *target* deadline, and if it has not completed by that time, it enters a *priority phase* during which its tasks are scheduled regardless of the current signal—combining the interpretability of a deadline with SPADE's signal-aware execution on the path to that deadline. We focus on γ in our experiments, but these knobs are complementary and can be composed with it.

3.2 Ablation design

While SPADE captures all three intuition points from § 2.2 by jointly provisioning the cluster and scheduling tasks in jobs, it is classically more natural to consider the scheduling and provisioning tasks as separate ones. For this reason, we also consider an ablation study of SPADE that isolates these tasks and quantifies the benefits of considering them jointly.

First, note that a "scheduling-only" ablation of SPADE removes its ability to idle executors—instead, tasks are scheduled in descending order of relative importance. This is similar to running the schedulers that computed the original score or probability distribution (i.e., Graphene [30], Decima [53]), and these are baselines in our experiments (see § 5).

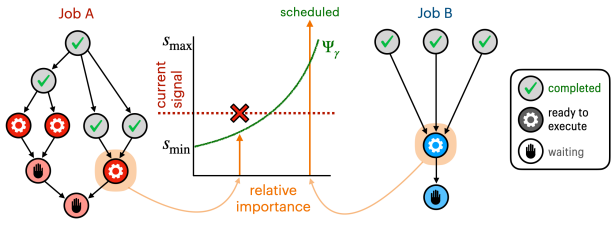


Figure 3: Illustrating the signal-aware filter of SPADE. Jobs A and B are actual DAGs drawn from the TPC-H and Alibaba workloads used in our experiments [2, 69]. Highlighted tasks illustrate two possible outcomes. In A, the sampled task has low relative importance, so it is deferred. In contrast, B's sampled task is a *bottleneck* with high *relative importance*: even during high signal, it is scheduled to avoid increasing makespan.

It is also intuitive to consider a "provisioning-only" ablation that extracts the key resource provisioning ideas implicitly performed by SPADE, while delegating scheduling to another policy. To capture this, we study SAP (**S**ignal-**A**ware **P**rovisioning), which applies a time-varying resource quota to the cluster and coexists with any scheduler (see § 5 for implementations with existing baselines). We detail it below.

SAP design. Given a cluster with K executors, the possible resource quotas are given by $\{0,1,\ldots,K\}$. SAP defines a discrete threshold set analogous to Ψ_γ in SPADE —in this case, the set is defined such that the value of the time-varying signal directly maps to a certain amount of resources that can be used at that time step. Analogous to the signal-awareness parameter γ in SPADE, we define a user-specified minimum quota $B \in \{1,\ldots,K\}$ that always allows the cluster to use up to B executors, ensuring continuous progress on jobs.

Given possible quotas $\mathcal{R} = \{B,\ldots K\}$, the thresholds are a set of values $\{\Phi_i\}_{i\in\mathcal{R}}$, and a quota is set based on how many values are *above* the current signal—see Appendix B.2 for

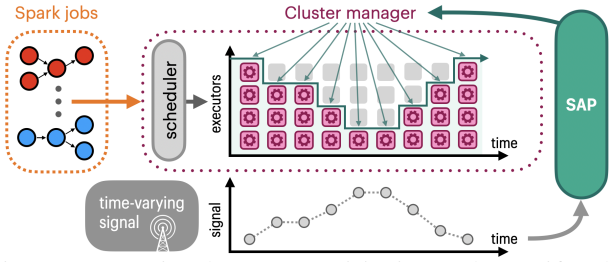


Figure 4: SAP (Signal-Aware Provisioning) only specifies the *resource amount* (e.g., no. of executors) that can be used based on the external signal, interfacing with a cluster manager. SAP can be implemented without changing an existing scheduler.

a formal definition of these.[3] Given signal $s(t)$, the resource quota is $r(t) \leftarrow \arg\max_{i \in \mathcal{R}} \Phi_i : \Phi_i \leq s(t)$. For ease of implementation, this quota is enforced without preemption; when executors become available, new task assignments are only allowed if $r(t)$ is greater than the number of busy executors. In Fig. 4, we illustrate the operation of SAP as an adaptive provisioning module. In Appendix B.2, we give worst-case analytical bounds on the stretch factor for SAP. The B parameter (minimum resource quota) ensures that the stretch factor is finite and decreasing in B, while greater potential cost savings are unlocked as the worst-case stretch factor increases.

4 Implementation

We have implemented proof-of-concepts of SPADE and SAP for Apache Spark on Kubernetes—see § 4.1 for details. We also conduct large-scale experiments in a realistic Spark simulator—see § 4.2 for how we extend a simulation environment [53] to evaluate our signal-aware systems.

4.1 Spark and Kubernetes integration

Resource provisioning & stage scheduling. In Spark deployed on a Kubernetes cluster, each application is submitted to the API server [8] that creates a "driver" running in a pod. We use Spark's dynamic allocation feature, which enables the driver to create executor pods dynamically as needed by the application—these executors connect with the driver and execute application code. Kubernetes handles the scheduling of (driver and executor) pods for each application, while the Spark driver selects stages to execute within an application.

We implemented SPADE as a pluggable scheduling service that coordinates between Spark and Kubernetes. The service includes inference code for a state-of-the-art DAG scheduler Decima [53] that provides probability distributions for SPADE's relative importance calculation. The time-varying signal is obtained from an external API, and context about the cluster and job states is collected from Kubernetes and Spark. To facilitate this, we made the following modifications: First, we implemented a Kubernetes *scheduler plugin* [17] that communicates with SPADE to determine which application should receive available resources. This uses APIs exposed by the default `kube-scheduler` and requires building/configuring a custom scheduler pod. We restrict the scope of our plugin to a dedicated namespace for Spark apps. Next, we made changes to Spark [76] so that each application communicates with SPADE when choosing the next stage to execute—Spark provides scripts to build pod Docker images [55] from source.

To implement the SAP ablation study, we wrote a Python daemon to retrieve the current signal from an API and adjust the resources available to Spark by setting a *resource quota* [3] within a dedicated namespace—our implementation adjusts *CPU and memory* quotas to correspond with a maximum number of executors. When the quota is *lowered*, existing pods are *not* preempted, but new pods are not scheduled until usage falls below the quota.

Setting level of parallelism. In a Spark DAG, each *stage* (i.e., task) includes multiple operations that may be parallelizable over multiple executors. Setting a *parallelism limit* (number of executors working on a stage) is a key component of Spark resource management (e.g., see [53, Section 5.2]). More executors are not necessarily better: assigning many executors to a stage that does not benefit from it will *block* other jobs in the queue. For *signal-awareness*, we enable SPADE and SAP to set new parallelism limits for the current job each time a stage is scheduled, and particularly to set *lower* limits when signal is high (e.g., see **i)** and **ii)**, § 2.2).

In our implementation of SPADE, if a stage is deferred, it *idles* (see Alg. 1) the newly freed executors that prompted a scheduling event. Otherwise, the stage's parallelism limit is set to $P' := \lceil P \cdot \min\{\exp(\gamma(s_{\min} - s(t))), (1-\gamma)\}\rceil$, where P is the limit chosen by the scheduler that generated the input score or probability distribution for SPADE (in our experiments, either Decima or Graphene). This mirrors the exponential trade-off in SPADE's design—e.g., when the current signal $s(t)$ is close to $s_{\min}$, the limit is set to $\lceil (1-\gamma)P \rceil$, and as $s(t)$ grows, the limit decreases exponentially to 1.

For the SAP ablation, if the signal-agnostic scheduler specifies a parallelism limit P, SAP first attempts to schedule a stage with $P' = \lceil P \cdot r(t)/K \rceil$, where $r(t)/K$ is the ratio of the resource quota vs. the total number of executors. If the number of available executors is less than P', the current stage takes all of the remaining available executors.

4.2 Spark simulator environment

Mao et al. [53] developed a faithful simulator of Spark's *standalone mode* (i.e., where Spark is the cluster manager), achieving an error (in run times) of within 5% [53, Fig. 18]. This simulator captures all first-order effects of Spark execution (e.g., delays in executor movement, parallelism overheads)—it has since seen wide use in Spark contexts [4, 27, 35, 46, 54, 64]. We implement SPADE, the SAP ablation study, and additional baselines through the following modifications:

▶ *Accounting:* Each job's cost due to the time-varying signal is measured *ex post facto* to avoid impacting simulator fidelity. After an experiment completes, existing metrics (e.g., executor lifetimes) and a signal trace are used to tally the cost.

▶ SPADE*:* We implement SPADE, which uses score or probability distributions over tasks computed by either Graphene or Decima.

▶ SAP*:* We implement SAP as a wrapper over each *signal-agnostic* scheduler in the simulator—see § 5.1.

▶ *Baselines:* We implement GreenHadoop [28] and Graphene [30] as additional baselines for our experiments. See Appendix A.1 for implementation details.

[3] Recall that a low signal implies using resources is less costly, WLOG.

With these modifications, the simulator allows us to quickly test many scenarios with a high degree of accuracy.

5 Evaluation

We evaluate our proposed system in a prototype cluster and a realistic Spark simulator, using workloads from TPC-H benchmarks [69] and Alibaba production DAG traces [2]. We conduct our main experiments using *carbon intensity* as the time-varying signal, as it is readily available for several regions with different characteristics and exhibits the intra-day variability that motivates systems adapting to time-varying conditions [23]. We also conduct simulator experiments using an *available power* signal obtained from Google cluster traces [73]. In doing so, we answer the following questions:

1. How do SPADE and SAP navigate the trade-off between signal-awareness and makespan?
2. How do SPADE and SAP adapt to changes in signal characteristics (e.g., variation) and workload characteristics?

Table 1: Statistics for carbon signal traces, including the duration, granularity, min., max., mean, and coefficient of variation (*higher value implies more variation*).

Trace	Duration	Unit	Min.	Max.	Mean	Co. var
PJM	01/01/2020-12/31/2022 Hourly granularity 26,304 pts.	$gCO_2eq./kWh$	293	567	425	0.110
CAISO			83	451	274	0.309
ON			12	179	50	0.654
DE			130	765	440	0.280
NSW			267	817	647	0.143
ZA			586	785	713	0.046

5.1 Experimental setup

Signal traces. For carbon intensity, we use historical traces from six regions—each trace provides hourly carbon intensity data in grams of CO_2 equivalent per kilowatt-hour ($gCO_2eq./kWh$) [23]. For these traces, the secondary objective is to reduce the carbon footprint of running jobs. We define the carbon footprint as follows: If the per-time-step energy usage of the Spark cluster at full utilization is given by a constant E, the footprint at time t is given by $s(t) \times E \times u(t)$, where $u(t)$ is the cluster's current executor utilization.

In Table 1 and Fig. 15, we give snapshots of each region, showing how grid characteristics impact the signal. Larger *coefficients of variation* (ratio of the standard deviation to the mean) correspond to greater renewable penetration—for instance, a large fraction of CAISO's capacity is solar PV, while the capacity in ZA is predominantly coal. To better observe the behavior of our signal-aware systems, we follow prior work [28] and scale time and workloads in experiments that use carbon traces such that 1 minute of *real time* corresponds to 1 hour of *simulated time*—since the carbon signal is reported hourly, this approximates a scenario where jobs work with large amounts of data and run for several hours, as is becoming common in e.g., data curation for LLMs [9, 10, 12, 50, 70]. We provide an experiment to characterize the effect of this scaling in Appendix A.1.4.

For the available power signal, we use Google cluster traces [73]—these traces provide power utilization as a percentage at 5-minute intervals in 8 production clusters for May 2019—see Table 5 for statistics of each trace. We use the average power utilization across a single cluster (i.e., "cell") as the time-varying signal, and the secondary objective is to minimize the power overloading due to running jobs. We define power overloading as follows: Each cluster has a minimum and a maximum avg. power utilization, denoted by $s_{\min}$ and $s_{\max}$. Then the overloading is given by $s(t) + (s_{\max} - s_{\min}) \times u(t) - s_{\max}$. Note that the overloading is 0 when our Spark cluster is at full utilization but the broader cluster utilization is minimal (i.e., $s(t) = s_{\min}$).

Workload traces. For workloads, we use TPC-H benchmarks [69] and real DAGs from production Alibaba traces [2]. We construct workloads such that the inter-arrival times follow a Poisson distribution while specific jobs are randomly picked from the respective traces. In the main body, we consider an average inter-arrival time of 30 real-time seconds (30 minutes in the carbon traces), with additional experiments measuring the impact of arrival rate in Appendix A.2.

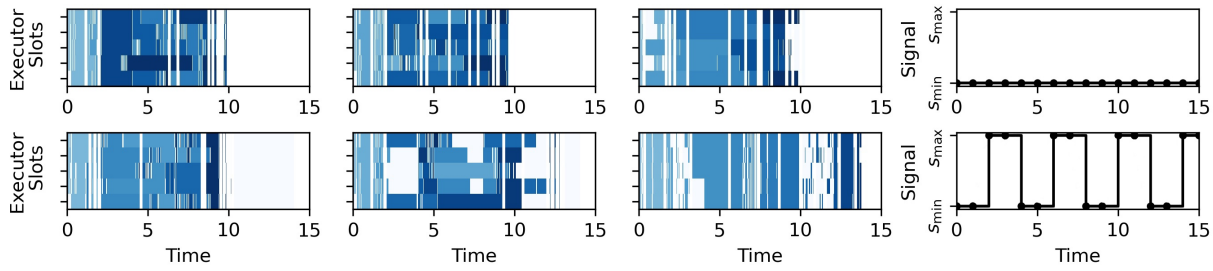
The TPC-H queries we experiment with operate on synthetic data with scales of 2 GB, 10 GB, and 50 GB—these correspond to average real durations of 180 seconds, 386 seconds, and 1,261 seconds when given a single executor. In our prototype experiments, we also construct workloads based on DAG information from the Alibaba trace [2]. These DAGs exhibit a realistic power law distribution (many DAGs of short duration, few DAGs of long duration), they have 66 nodes on average, and an average total duration (on one executor) of 7,989 seconds. We uniformly scale these Alibaba DAG durations by $1/60$ to match our carbon scale—this yields jobs that take 2.2 real-time minutes to complete on average.

In the simulator, each experiment is run over a full trace; each carbon trace is three years of hourly data, while each power trace is one month of 5-minute data. In the prototype, each experiment is run for several trials, where each starts at a uniformly randomly chosen time in a particular carbon trace. Across all experiments, the upper and lower bounds of $s_{\max}$ and $s_{\min}$ are set to the maximum and minimum signal values over a lookahead forecast window of 48 time slots.

Baselines. We compare against the following baselines:

▶ *Default Spark/Kubernetes behavior (default)*: The default behavior of Spark on Kubernetes—Spark uses first in, first out (FIFO) to choose stages within a job, while the Kubernetes scheduler mediates between pods of each job during execution [26]. In the simulator's Spark standalone mode, this baseline implements only the FIFO scheduling.

▶ *Decima*: An RL scheduler for Spark that is optimized for job completion time [53]. We use the simulator's training envi-



(a) Decima *(b)* SPADE-Decima *(c)* SAP-FIFO *(d)* Time-varying signal

Figure 5: Visualizing executor usage over time for three schedules, *(a)* Decima, *(b)* SPADE-Decima, and *(c)* SAP-FIFO in a small simulator with 5 executors and 20 TPC-H jobs arriving over time. The top plot illustrates results when the signal is *flat* during the given time horizon (see ***(d)***), while the bottom plot illustrates results for a signal with *high fluctuations*. In executor plots ***(a-c)***, jobs are unique shades of blue, while "idle" periods are indicated by the white background.

ronment to train Decima for 20,000 epochs. When SPADE uses node probability distributions from Decima's policy network to interpret DAG structure, we denote it as SPADE-Decima.

▶ *Weighted Fair*: A heuristic that assigns executors proportionally to each job's workload, with tuned weights to improve performance on the simulated workloads [53].

▶ *Graphene*: A DAG scheduler optimized for throughput and heterogeneous resource demands [30]. When SPADE uses node score distributions from Graphene's scoring module to interpret DAG structure, we denote it as SPADE-Graphene.

▶ *GreenHadoop:* A MapReduce framework proposed to leverage green energy by matching workloads with the availability of solar [28]. We evaluate this baseline as the closest related work in the carbon experiments. This framework predates Spark, so we adapt its key ideas for DAG scheduling in the simulator—see Appendix A.1 for adaptation details.

Metrics. We use four metrics to evaluate our approaches.

▶ *Carbon Reduction*: When carbon intensity is used as a signal, we report the carbon footprint for each scheduling policy as a percentage decrease relative to the default baseline unless stated otherwise. The values are in the range of [-100%, ∞), with negative values indicating a reduction and positive values indicating an increase relative to the baseline.

▶ *Power Overload (POL) Reduction*: When available power is used as a signal, we report the power overloading for each scheduling policy as a percentage decrease relative to the default baseline unless stated otherwise. Its values lie in the same range as carbon reduction.

▶ *Job Completion Time (JCT)*: We report statistics of job completion time across all the jobs in each experimental run. We report JCT as a fraction of the average JCT for the carbon-agnostic default baseline unless stated otherwise. The values can be in the $(0, \infty)$ range, with below 1 indicating a reduction in JCT and above 1 indicating an increase in JCT.

▶ *Makespan*: We report the total time to complete all the jobs in a given experiment as a fraction of the makespan for the default. Its values lie in the same range as JCT. While JCT focuses on individual jobs, makespan represents the system's overall throughput and efficiency.

5.2 Signal-aware systems in action

Before moving to our main results, we demonstrate signal-aware behavior in Fig. 5, which visualizes the schedules generated by Decima, SPADE-Decima, and SAP-FIFO for two toy example signal traces. In the first (top) case of Fig. 5, the signal is consistently low (i.e., $s_{\min}$), so Decima, SPADE-Decima, and SAP-FIFO all focus on minimizing makespan. Note that FIFO executes jobs in arrival order (light to dark shading) while both Decima and SPADE-Decima pack individual tasks to optimize throughput. The synchronized idle periods visible across all three schedules correspond to intervals when no jobs are in the queue, since jobs arrive over time (with a 15-second inter-arrival in real-time) rather than all at $t = 0$. In the second (bottom) case of Fig. 5, the signal fluctuates between $s_{\max}$ and $s_{\min}$ every two time steps. SPADE-Decima makes fine-grained scheduling decisions to maximize throughput during low-signal periods while idling executors during high-signal periods. This is in contrast to SAP-FIFO, which applies a uniform resource quota across the cluster without consideration of bottlenecks or the remaining processing time of active tasks. As a result, it does not reduce utilization as much during high-signal periods, and still increases overall makespan more than SPADE-Decima.

5.3 Prototype experiments

Our prototype is deployed on an OpenStack cluster running Kubernetes v1.31 and Spark v3.5.3 (both modified per § 4) in Chameleon Cloud [38]. Our testbed consists of 51 `m1.xlarge` virtual machines, each with 8 VCPUs and 16GB of RAM. One VM is designated as the control plane node, while the remaining 50 are workers, each hosting two executor pods. Our Spark configuration allocates 4 VCPUs and 7GB of RAM to each of the 100 executors.[4] To avoid a known issue with Spark's dynamic allocation feature that can cause it to hang on Kubernetes [26], we configure an upper limit of 25 executors that can be allocated to any single job. In the prototype,

[4] Spark's default *memory overhead factor* is 10%. The difference between 7GB (×2) and the 16GB RAM of each worker is to accommodate this [24].

Table 2: Summary of prototype results averaged over all six carbon traces. Each metric is normalized with respect to the Spark / Kubernetes default. SPADE-Decima and SAP-Default are configured to be moderately signal-aware.

Metric *(normalized to Default)*	Default	Decima [53]	SAP-Default	SPADE-Decima
Carbon Reduction (%)	0%	1.2%	24.7%	**32.9%**
Avg. Makespan	1.0	0.857	1.126	**1.013**

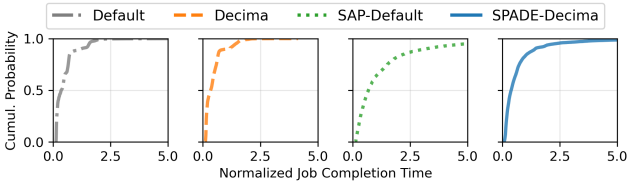


Figure 6: Prototype JCT results aggregated over all six carbon traces, normalized by the average JCT for Default.

we implement a small API that replays given signal traces to test our signal-aware systems, and we implement default and Decima as the baselines. Unless stated otherwise, the results are averaged over batch sizes of 25, 50, and 100 jobs. Furthermore, the results for each experimental configuration are averaged over 10 trials.

Results. Table 2 presents the top-line results for our prototype experiments. SPADE-Decima and SAP-Default configured to be *moderately signal-aware* (γ is set to 0.5 for SPADE and B is set to 20 for SAP) achieve average carbon reductions of 32.9% and 24.7% compared to the default baseline, respectively. Compared to Decima, SPADE-Decima reduces carbon by 32.1%. Fig. 6 plots cumulative distribution functions (CDFs) of the JCT for all four techniques, normalized by the average JCT of default. We find that signal-awareness increases JCT mainly in the tail—for instance, SPADE's median JCT is 7.8% and 11.8% worse with respect to default and Decima, respectively, whereas 95$^{\text{th}}$ percentile JCT is 45.4% and 59.6% worse than default and Decima. SAP sees even larger increases, increasing 95$^{\text{th}}$ percentile JCT by 223% and 254.4% compared to default and Decima, respectively. For signal-awareness, this behavior is expected—for instance, a job that arrives while the signal is high may wait a long time before receiving service. Thus, at a global level, we also consider makespan as a proxy for the cluster's overall efficiency and throughput. SPADE increases average makespan by only 12.4% compared to Decima and 1.3% compared to the default.

Trade-offs between carbon and makespan. We test several parameter settings for SPADE and SAP to configure their signal-awareness in the DE grid region with batches of 50 TPC-H or Alibaba jobs. Fig. 7 plots the carbon-makespan trade-off for five settings of SPADE relative to the Spark/Kubernetes default. Increasing the signal-awareness of SPADE improves savings on the secondary objective at the expense of longer makespan, with a pronounced effect for values of γ approaching 1. Conversely, Fig. 8 plots the same carbon-makespan trade-off for

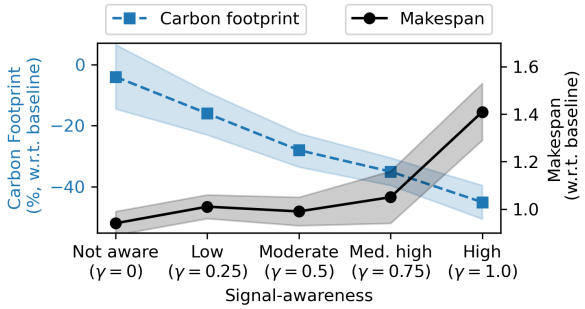


Figure 7: Relative carbon reduction and makespans (w.r.t. the Spark/Kubernetes default) for SPADE-Decima in the prototype, given different degrees of signal-awareness (γ). Shading denotes standard deviation across 10 random trials.

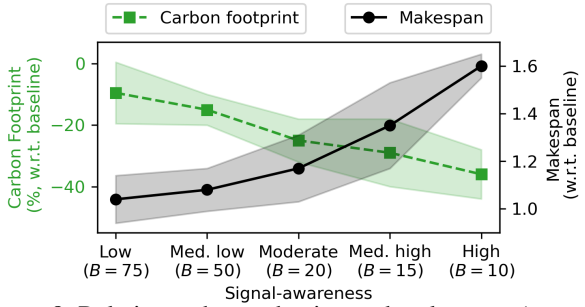


Figure 8: Relative carbon reduction and makespans (w.r.t. the Spark/Kubernetes default) for SAP-Default in the prototype, given different degrees of signal-awareness (B). Shading denotes standard deviation across 10 random trials.

five settings of SAP; SAP sacrifices more in makespan (relative to SPADE) for the same amount of carbon savings (i.e., for the same amount of signal-awareness).

Effects of carbon trace characteristics. Next, we analyze the effect of grid characteristics on our signal-aware systems using subsets of each signal trace (via 30 trials with 25, 50, and 100 jobs). Fig. 9 plots the average carbon reduction and makespan of SPADE, SAP, and Decima. Decima is signal-agnostic and shows a minimal carbon reduction that stays relatively constant across all traces. SPADE and SAP incorporate the external signal into decisions, and we observe a positive relationship between the variability of a trace and the resulting carbon reduction. For example, in ZA where the sig-

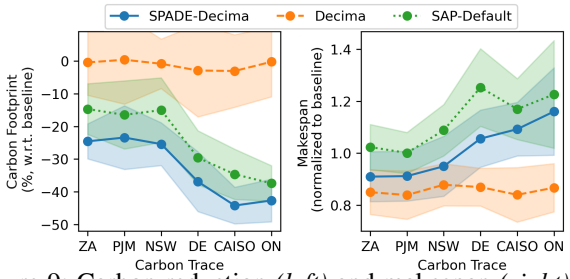


Figure 9: Carbon reduction *(left)* and makespan *(right)* for SPADE, SAP, and Decima in six carbon traces. Shading denotes standard deviation across 30 trials.

Table 3: Summary of results for simulator experiments averaged over all tested carbon & power traces. Each metric is normalized with respect to the default Spark FIFO behavior. SPADE and SAP are configured to be moderately carbon-aware, and makespan measures the total flow time for batches of jobs arriving continuously.

Metric *(normalized to FIFO)*	**FIFO**	**W. Fair**	**Decima** [53]	**Graphene** [30]	**GreenHadoop** [28]	**SAP**				**SPADE**	
						FIFO	W. Fair	Decima	Graphene	**Decima**	**Graphene**
Carbon Reduction (%)	0%	12.1%	21.5%	19.3%	8.2%	22.7%	34.2%	31.1%	37.1%	**39.7%**	**40.1%**
POL Reduction (%)	0%	0.9%	31.8%	33.3%	—	16.2%	14.3%	39.8%	41.7%	**43.9%**	**51.0%**
Avg. Makespan	1.0	0.972	0.970	0.983	1.077	1.108	1.011	1.061	1.076	**1.045**	**1.011**

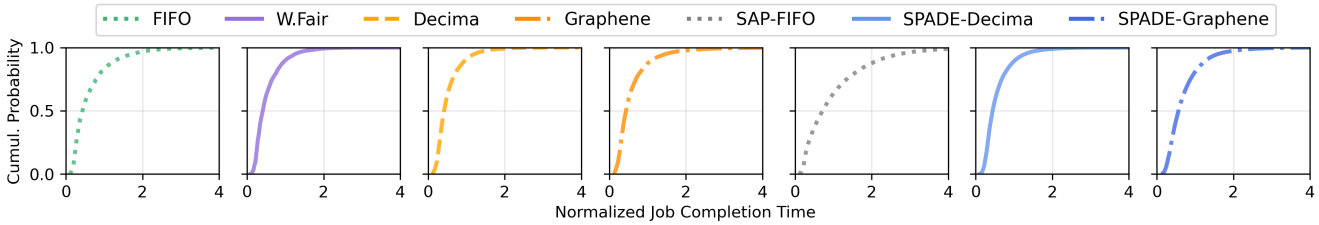


Figure 10: JCT results aggregated over all tested traces, normalized by the average JCT of FIFO.

nal is relatively constant, high-carbon periods do not prompt SPADE to defer tasks since the *potential future reductions* are insignificant. In contrast, high-signal periods in the CAISO grid correspond to nighttime scenarios on the grid, where daytime solar bolsters future reductions. These interactions are also illustrated through makespan—carbon traces with more variable energy mixes drive increases in makespan in exchange for more carbon reduction, since SPADE and SAP *wait* for future low-signal periods.

5.4 Simulator experiments

To conduct large-scale experiments over full traces, we evaluate SPADE in the Spark simulator using TPC-H workloads, comparing it against **Weighted Fair**, **GreenHadoop**, and **Graphene** baselines, in addition to the Decima and default baselines from prototype experiments and SAP ablation study. We refer to the default baseline as FIFO in simulation-based experiments for accurate representation.

Simulator behavior. To quantify differences between our prototype and the simulator (meant to model Spark's standalone mode of operation), we illustrate behavior for an identical batch of 50 TPC-H jobs in Appendix A.1.3. A notable difference between the prototype and the simulator is the relative performance of the built-in baseline (FIFO in the simulator, Spark/Kubernetes default in the prototype). In short, the simulator's FIFO scheduler *over-assigns* executors to individual jobs, blocking others from entering service (thus increasing JCT)—this also increases its relative cost on the secondary objective compared to the default behavior in our prototype.

Table 3 presents the top-line results for simulator experiments using carbon intensity and power traces.

Carbon trace results. We observe that SPADE achieves significant reductions in carbon emissions compared to the baselines—configured to be moderately carbon-aware, SPADE-Decima achieves an average reduction of 23.1% compared to Decima, and a reduction of 39.7% compared to FIFO. SPADE-Graphene achieves reductions of 25.7% compared to Graphene, and a reduction of 40.1% compared to FIFO. Finally, our ablation study SAP achieves an average carbon reduction of 22.7% when implemented on top of FIFO, 25.1% with Weighted Fair, 14.5% with Decima, and 22.1% with Graphene, falling behind SPADE in the latter two cases.

Power trace results. Across experiments that use the available power signal, SPADE significantly reduces the power overloading metric, which corresponds to a *better alignment* with the signal—configured to be moderately signal-aware, SPADE-Decima achieves an average improvement of 17.7% compared to Decima, and 43.9% compared to FIFO. SPADE-Graphene achieves reductions of 26.5% compared to Graphene, and a reduction of 51% compared to FIFO. Finally, our ablation study SAP improves overloading by 16.2% when implemented on top of (and compared to) FIFO, 13.5% with Weighted Fair, 11.7% with Decima, and 12.6% with Graphene.

Makespan & performance. For 25, 50, and 100 jobs, SPADE increases average makespan by 7.7% (Decima) and 2.9% (Graphene), which are only 4.5% and 1.1% degradations compared to FIFO, respectively. For the SAP ablation study, makespan increases by 10.8% when implemented on top of (and compared to) FIFO, 4.0% on top of Weighted Fair, 9.3% on top of Decima, and 9.5% on top of Graphene.

Fig. 10 plots cumulative distribution functions (CDFs) of the JCT for all tested techniques, normalized by the average JCT of FIFO. Similar to the prototype, we find that signal-awareness increases tail JCT: SPADE's median JCT degrades by 6.7% (Decima) and 2.3% (Graphene), while 95th percentile JCT degrades by 42.7% (Decima) and 35.8% (Graphene). For visual clarity, CDFs for the remaining tested policies (GreenHadoop, SAP-Partition, SAP-Decima, and SAP-Graphene) are deferred to Appendix A.2.4; their qualitative behavior matches the trade-offs summarized in Table 3.

Trade-offs between signal-awareness and makespan. In the results summary, we observe improvements in the secondary objective(s) in exchange for degradation in perfor-

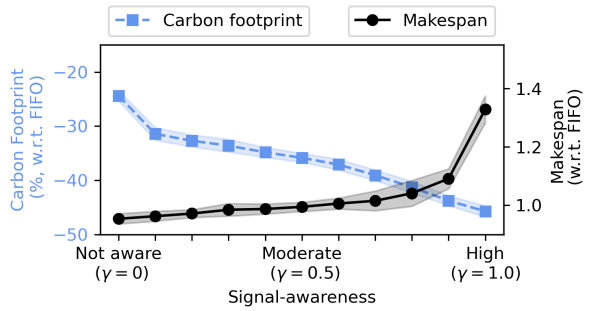


Figure 11: Relative carbon reduction and makespans (with respect to FIFO) for SPADE-Decima in the simulator, given different degrees of signal-awareness (γ). Shading denotes standard deviation across DE carbon trace.

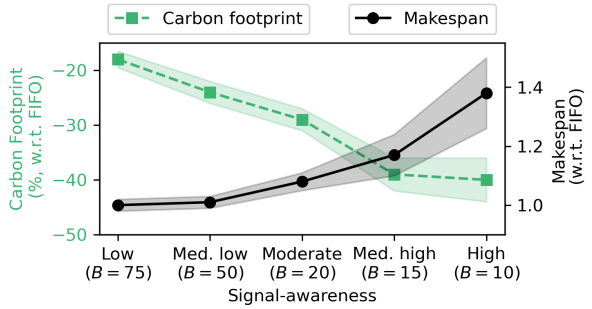


Figure 12: Relative carbon reduction and makespans (with respect to FIFO) for SAP-FIFO in the simulator, given different degrees of signal-awareness (B). Shading denotes standard deviation across DE carbon trace.

mance. Since SPADE and SAP can be configured to be more or less signal-aware, we explore this *trade-off* in the DE carbon trace with batches of 50 jobs. We vary hyperparameters γ (SPADE) and B (SAP) to measure the impact on both objectives. Fig. 11 and Fig. 12 plot the carbon-makespan trade-off for SPADE-Decima and SAP-FIFO, respectively, compared against FIFO. The qualitative trends match the prototype (Fig. 7, Fig. 8), with SAP again sacrificing more makespan than SPADE for the same level of signal-awareness.

A natural follow-up question is whether sweeping γ alone traces the full Pareto frontier of the multi-objective problem. The NP-hardness of DAG scheduling prevents a clean analytical answer, but the experiments give an empirical one: across our parameter sweep, SPADE's carbon-makespan outcomes lie on a frontier that Pareto-dominates the baselines and the SAP ablation (e.g., see Fig. 13), and the $\gamma \sim 0$ endpoint recovers the underlying signal-agnostic scheduler. Sweeping γ from 0 to 1 thus smoothly interpolates between the makespan-optimal and signal-aware regimes along the dominant frontier we observe, though it does not necessarily coincide with the (unknown) true Pareto frontier of the problem.

Advantages of joint scheduling & provisioning. By comparing SPADE against the SAP ablation, we can quantify the advantages of the key DAG scheduling ideas of SPADE, namely relative importance (see § 3.1). We examine this in detail using the DE grid region with batches of 50 jobs. We configure SPADE and SAP with ten parameter settings for varying degrees of signal-awareness. Fig. 13 plots the result of this experiment, where each dot denotes the outcome of one trial—results for Decima are in the left-hand plot, while results for Graphene are in the right-hand plot. We fit a cubic polynomial to the outcomes of all four methods to illustrate the key trend: SPADE exhibits a strictly better trade-off between carbon and makespan. For trials where carbon savings lie between 35% and 45%, SPADE increases makespan by an average of 7.9% (Decima) and 1.2% (Graphene), while SAP increases it by an average of 42.7% (Decima) and 17.7% (Graphene). Conversely, for trials where makespan is increased by between 0% and 10%, SPADE achieves average carbon savings of 35.6% (Decima) and 52.4% (Graphene), while SAP achieves an average savings of only 20.1% (Decima) and 24.8% (Graphene).

Carbon intensity trace characteristics. The top-line results in Table 3 average over all carbon traces—as in the prototype, Fig. 14 plots the carbon reduction and makespan for each of SAP-FIFO, SPADE-Decima, and Decima across each of the six carbon traces. Decima's carbon reduction relative to the baseline is higher relative to that observed in the prototype—this is due to differences between Spark's standalone FIFO scheduler and the Spark/Kubernetes behavior of our prototype (see Appendix A.1.3). Similarly, SPADE's reductions increase alongside Decima's, and SAP-FIFO in the simulator does slightly worse on the secondary objective compared to SAP in the prototype.

When does SPADE help most? SPADE's advantage over structure-agnostic alternatives (e.g., the SAP ablation wrapping FIFO or Weighted Fair) is largest when the workload contains DAGs with non-trivial structure—jobs whose critical path runs through a subset of highly-connected tasks, or jobs with long chains of dependent stages. Both TPC-H and the Alibaba trace contribute such DAGs (Jobs A and B in Fig. 3 are representative), and SPADE's relative-importance metric is what lets it preserve makespan by protecting bottleneck tasks during high-signal periods. Conversely, when the workload is dominated by "flat" DAGs whose ready tasks are essentially

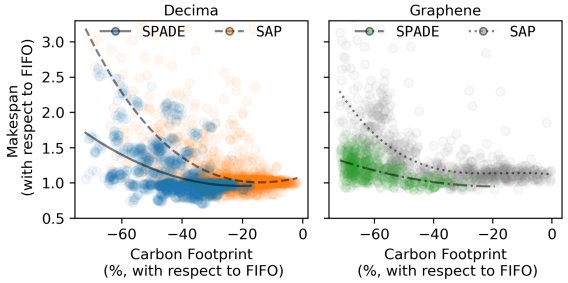


Figure 13: Relative carbon reduction vs. makespan for SPADE and SAP in simulator experiments, given varying parameters $\gamma \in [0.1, 1.0]$ and $B \in \{1, 5, 10, \ldots, 85\}$ that correspond to degrees of signal-awareness. Each dot represents an individual trial, and lines represent a cubic polynomial of best fit.

Table 4: Summary and comparison with related work.

Reference	Signal-aware	Precedence-aware	Workload type	Single vs. multiple concurrent jobs	Implementation	Theoretical guarantee
Decima [53]	✗	✓	Batch	Multiple	✓	✗
Graphene [30]	✗	✓	Batch	Multiple	✓	✗
Su et al. [65]	✗	✓	N/A	Multiple	✗	✓
DTPR [40]	✓	✗	Batch	Single	✗	✓
GreenHadoop [28]	✓	✗	Batch	Multiple	✓	✗
Blink [62]	✓	✗	N/A	N/A	✓	✗
CarbonScaler [33]	✓	✗	Batch	Single	✓	✗
GAIA [34]	✓	✗	Batch	Multiple	✓	✗
Caribou [32]	✓	✓	Interactive	N/A	✓	✗
SPADE (this work)	✓	✓	**Batch**	**Multiple**	✓	✓

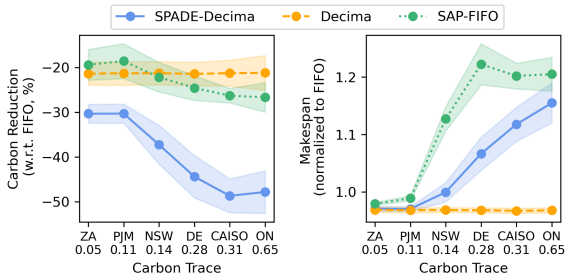


Figure 14: Carbon reduction *(left)* and makespan increase *(right)* for SAP, SPADE, and Decima (relative to FIFO) in six carbon traces. Shading denotes standard deviation across full experiment.

all available at once (or by effectively single-stage jobs), the notion of a bottleneck task is less informative and SPADE's advantage over a SAP-style wrapper narrows: structure-agnostic provisioning suffices because there is little structure to exploit.

5.5 Takeaways

Through evaluation in a prototype cluster and realistic simulator, we show that SPADE is effective, reducing carbon by 32.9% and reducing power overloading by up to 51% in exchange for modest increases ($< 5\%$) in makespan. SAP, our ablation study, shows that the joint scheduling and provisioning decisions of SPADE are necessary to obtain a favorable trade-off between signal-awareness and performance (see Fig. 13). Intuitively, the behavior of all our signal-aware techniques depends on the characteristics of a particular signal trace—our experiments demonstrate that SPADE works for a variety of signals and exhibits high configurability to meet any desired priority between signal-awareness and performance. An orthogonal route to reducing a secondary objective while maintaining the same performance is to expand cluster capacity (e.g., add additional executors) and ramp up *more* during low-signal periods—prior work has shown that this can hold performance roughly constant while still reducing a secondary objective (such as carbon footprint), at the cost of extra hardware capex (and, in the carbon case, extra embodied emissions) [1, 33, 34, 48]. SPADE accommodates this regime naturally: when additional executors are available during low-signal periods, its filter Ψ_γ admits more tasks as $s(t)$ falls, so the system opportunistically uses this extra capacity without any changes to the system.

6 Related Work

To contextualize the related work, Table 4 summarizes guiding questions that we ask: 1) does the work consider scheduling and/or provisioning in response to an external time-varying signal, 2) does it consider DAG structure (stages and precedence constraints), 3) are batch or interactive workloads considered (if applicable), 4) does the proposed scheme process a single job in isolation or multiple workloads concurrently, 5) has the scheme been implemented in a prototype, and 6) does the work provide any theoretical guarantees about, e.g., the approximation quality of the proposed scheme? SPADE is unique in answering all of the above yes/no questions in the positive for the setting of multiple batch DAG workloads.

For the performance objective of makespan minimization, the scheduling of precedence-constrained tasks (i.e., tasks with DAG structure) has been extensively studied—classic results show that solving the problem optimally is NP-hard even in simple forms [42], so most studies have focused on heuristics and approximations [13, 15, 19–21, 29, 39, 45, 52, 58, 66]. As DAG-structured workloads have gained salience in applications, systems such as Graphene [30], Decima [53], and others [31, 43, 47, 74, 77] have been proposed to optimize the throughput of DAG workflows and/or Spark jobs.

A few works have considered DAG scheduling with other objectives beyond makespan [49, 56, 57, 65, 68]. These balance reducing makespan alongside one other metric, such as energy usage. For example, Su et al. [65] theoretically studied *energy-aware* list scheduling for precedence constrained tasks, giving bounds for a combined objective of energy consumption and performance. However, these works typically focus on a secondary objective such as energy or cost, which do not capture the time-varying dynamics of a signal-aware objective such as carbon emissions or grid integration.

In recent years, several studies have proposed new theo-

retical techniques and schemes for signal-awareness and/or carbon-awareness in different workloads. For example, CarbonScaler [33] is a provisioning scheme for batch workloads that exploits temporal differences in carbon intensity signals, using more parallelism when the signal is low. In the context of this work, CarbonScaler can be interpreted as similar to our ablation study SAP, although the original paper considered a single job at a time and assigns deadlines to each job, rather than offering a tunable trade-off between objectives.

CarbonScaler is one of several studies that explicitly consider external signals such as carbon [34, 40] or power [62], but do not consider the internal DAG structure of workloads. Closer to this work's focus on data processing workloads, GreenHadoop [28] is a MapReduce framework that considers an external signal (available renewable energy) when scheduling, although it does not consider DAG structure because it predates Spark. As outlined in § 2, considering these constraints is important for signal-aware execution of DAG workloads—ignoring this structure yields suboptimal performance on the secondary objective, performance objectives, or both.

One recent system, Caribou [32], considers both carbon signals and the DAG structure of serverless workflows. Caribou optimizes the *geospatial* carbon efficiency of serverless workflows by deploying them across data centers to exploit spatial variations in carbon (e.g., when one region's signal is high, another's may be low). However, since it focuses on interactive workloads, it does not exploit temporal signal variations—in other words, deploying it in one data center would not reduce carbon. Furthermore, applying Caribou to Spark would require "moving" a Spark cluster (including data being processed) across data centers. These differences in the "meaning" of job DAGs change the problem significantly: with SPADE, we do not attempt to exploit spatial differences in the time-varying signal—rather, SPADE attempts to anticipate and respond to uncertain signal values in one location.

7 Conclusion

We introduce SPADE, a signal-aware execution system for data processing workloads designed to address emerging scenarios in data center operations that balance between performance and a secondary objective defined by an external signal, such as power availability, carbon emissions, or water use. Through experiments in a prototype and simulator, we show that SPADE's joint consideration of scheduling tasks and provisioning resources is necessary to achieve a favorable trade-off between these objectives—as a result, our experiments demonstrate that SPADE can improve performance on a secondary objective of interest by 32.9% without prohibitive performance degradation.

Acknowledgments

This material is based upon work supported by the U.S. Department of Energy, Office of Science, Office of Advanced Scientific Computing Research, Department of Energy Computational Science Graduate Fellowship under Award Number DE-SC0024386, and National Science Foundation grants CAREER-2045641, CNS-2325956, and CNS-2533814.

Disclaimers

This report was prepared as an account of work sponsored by an agency of the U.S. Government. Neither the U.S. Government nor any agency thereof, nor any of their employees, makes any warranty, express or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. References to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise do not constitute or imply endorsement, recommendation, or favoring by the U.S. Government or any agency thereof. The views and opinions of authors expressed herein do not state or reflect those of the U.S. Government or any agency thereof.

References

[1] Bilge Acun, Benjamin Lee, Fiodar Kazhamiaka, Kiwan Maeng, Udit Gupta, Manoj Chakkaravarthy, David Brooks, and Carole-Jean Wu. Carbon Explorer: A Holistic Framework for Designing Carbon Aware Datacenters. In *Proc. of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, ASPLOS 2023, page 118–132, 2023.

[2] Alibaba. Cluster data collected from production clusters in Alibaba for cluster management research, 2018.

[3] The Kubernetes Authors. Resource Quotas – Kubernetes Documentation . https://kubernetes.io/docs/concepts/policy/resource-quotas/, 2025.

[4] Vivek Bengre, M Reza HoseinyFarahabady, Mohammad Pivezhandi, Albert Y Zomaya, and Ali Jannesari. A learning-based scheduler for high volume processing in data warehouse using graph neural networks. In *International Conference on Parallel and Distributed Computing: Applications and Technologies*, pages 175–186. Springer, 2021.

[5] Siddharth Bhela, Vassilis Kekatos, and Sriharsha Veeramachaneni. Enhancing Observability in Distribution Grids using Smart Meter Data, 2016. arXiv:1612.06669.

[6] Roozbeh Bostandoost, Walid A. Hanafy, Adam Lechowicz, Noman Bashir, Prashant Shenoy, and Mohammad Hajiesmaili. Data-driven Algorithm Selection for Carbon-Aware Scheduling. In *Proceedings of the 3rd Workshop on Sustainable Computer Systems*, HotCarbon '24, July 2024.

[7] Roozbeh Bostandoost, Adam Lechowicz, Walid A. Hanafy, Noman Bashir, Prashant Shenoy, and Mohammad Hajiesmaili. LACS: Learning-Augmented Algorithms for Carbon-Aware Resource Scaling with Uncertain Demand. In *Proceedings of the 15th ACM International Conference on Future and Sustainable Energy Systems*, e-Energy '24, page 27–45, New York, NY, USA, 2024. Association for Computing Machinery.

[8] Eric Brewer. Kubernetes and the Path to Cloud Native. Santa Clara, CA, July 2015. USENIX Association.

[9] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language Models are Few-Shot Learners, 2020.

[10] Maximilian Böther, Dan Graur, Xiaozhe Yao, and Ana Klimovic. Decluttering the data mess in LLM training. Austin, 2024. HotInfra 2024.

[11] Craig Chambers, Ashish Raniwala, Frances Perry, Stephen Adams, Robert R. Henry, Robert Bradshaw, and Nathan Weizenbaum. FlumeJava: Easy, Efficient Data-Parallel Pipelines. In *Proceedings of the 2010 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '10)*, PLDI '10, pages 363–375, New York, NY, USA, 2010. ACM.

[12] Daoyuan Chen, Yilun Huang, Zhijian Ma, Hesen Chen, Xuchen Pan, Ce Ge, Dawei Gao, Yuexiang Xie, Zhaoyang Liu, Jinyang Gao, Yaliang Li, Bolin Ding, and Jingren Zhou. Data-Juicer: A One-Stop Data Processing System for Large Language Models. In *Companion of the 2024 International Conference on Management of Data*, SIGMOD/PODS '24, page 120–134, 2024.

[13] Runwei Cheng, Mitsuo Gen, and Yasuhiro Tsujimura. A tutorial survey of job-shop scheduling problems using genetic algorithms—I. Representation. *Computers & Industrial Engineering*, 30(4), 1996.

[14] Andrew Chow R. 'We Are the Last of the Forgotten:' Inside the Memphis Community Battling Elon Musk's xAI. *TIME*, August 2025. Accessed: 2025-12-10.

[15] Fabián A Chudak and David B Shmoys. Approximation Algorithms for Precedence-Constrained Scheduling Problems on Parallel Machines that Run at Different Speeds. *Journal of Algorithms*, 30(2):323–343, Feb 1999.

[16] Philip Colangelo, Ayse K Coskun, Jack Megrue, Ciaran Roberts, Shayan Sengupta, Varun Sivaram, Ethan Tiao, Aroon Vijaykar, Chris Williams, Daniel C Wilson, Brandon Records, Zack MacFarland, Daniel Dreiling, Nathan Morey, Anuja Ratnayake, and Baskar Vairamohan. AI data centres as grid-interactive assets. *Nature Energy*, December 2025.

[17] Kubernetes Community. Scheduler Plugins, 2021.

[18] Data Center Dynamics. Grid demand will require active participation from data centers, June 2024.

[19] Sami Davies, Janardhan Kulkarni, Thomas Rothvoss, Jakub Tarnawski, and Yihao Zhang. Scheduling with Communication Delays via LP Hierarchies and Clustering, 2020. arXiv:2004.09682.

[20] Sami Davies, Janardhan Kulkarni, Thomas Rothvoss, Jakub Tarnawski, and Yihao Zhang. *Scheduling with Communication Delays via LP Hierarchies and Clustering II: Weighted Completion Times on Related Machines*, page 2958–2977. Society for Industrial and Applied Mathematics, January 2021.

[21] Lawrence Davis. Job shop scheduling with genetic algorithms. In *Proceedings of the first International Conference on Genetic Algorithms and their Applications*, pages 136–140, 2014.

[22] Ran El-Yaniv, Amos Fiat, Richard M. Karp, and G. Turpin. Optimal Search and One-Way Trading Online Algorithms. *Algorithmica*, 30(1):101–139, May 2001.

[23] Electricity Maps. Electricity Map. https://www.electricitymap.org/map, 2025.

[24] The Apache Software Foundation. Configuration – Spark Documentation. https://spark.apache.org/docs/3.5.3/configuration.html, 2024.

[25] The Apache Software Foundation. Job Scheduling – Spark Documentation. https://spark.apache.org/docs/3.5.3/job-scheduling.html, 2024.

[26] The Apache Software Foundation. Running Spark on Kubernetes – Spark Documentation. https://spark.apache.org/docs/3.5.3/running-on-kubernetes.html, 2024.

[27] Arkadiy Gertsman. A faster reinforcement learning approach to efficient job scheduling in Apache Spark. Master's thesis, University of Illinois at Urbana-Champaign, 2023.

[28] Íñigo Goiri, Kien Le, Thu D. Nguyen, Jordi Guitart, Jordi Torres, and Ricardo Bianchini. GreenHadoop: Leveraging Green Energy in Data-Processing Frameworks. In *Proceedings of the 7th ACM European Conference on Computer Systems*, EuroSys '12, page 57–70, 2012.

[29] R. L. Graham. Bounds for certain multiprocessing anomalies. *The Bell System Technical Journal*, 45(9):1563–1581, 1966.

[30] Robert Grandl, Srikanth Kandula, Sriram Rao, Aditya Akella, and Janardhan Kulkarni. Graphene: packing and dependency-aware scheduling for data-parallel clusters. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI'16, pages 81–97, USA, 2016. USENIX Association.

[31] Nathan Grinsztajn, Olivier Beaumont, Emmanuel Jeannot, and Philippe Preux. Geometric deep reinforcement learning for dynamic DAG scheduling. In *2020 IEEE Symposium Series on Computational Intelligence (SSCI)*, page 258–265. IEEE, December 2020.

[32] Viktor Urban Gsteiger, Pin Hong (Daniel) Long, Yiran (Jerry) Sun, Parshan Javanrood, and Mohammad Shahrad. Caribou: Fine-Grained Geospatial Shifting of Serverless Applications for Sustainability. In *Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles*, SOSP '24, page 403–420, 2024.

[33] Walid A. Hanafy, Qianlin Liang, Noman Bashir, David Irwin, and Prashant Shenoy. Carbonscaler: Leveraging cloud workload elasticity for optimizing carbon-efficiency. *Proc. of the ACM on Measurement and Analysis of Computing Systems*, 7(3), December 2023.

[34] Walid A. Hanafy, Qianlin Liang, Noman Bashir, Abel Souza, David Irwin, and Prashant Shenoy. Going Green for Less Green: Optimizing the Cost of Reducing Cloud Carbon Emissions. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, ASPLOS '24, page 479–496, 2024.

[35] Zhibo Hu, Chen Wang, Helen Paik, Yanfeng Shu, and Liming Zhu. Learning Interpretable Scheduling Algorithms for Data Processing Clusters, 2024. arXiv:2405.19131.

[36] Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. Dryad: Distributed Data-Parallel Programs from Sequential Building Blocks. In *Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems (EuroSys '07)*, pages 59–72, New York, NY, USA, 2007. ACM.

[37] Muhammed Tawfiqul Islam, Shanika Karunasekera, and Rajkumar Buyya. Performance and cost-efficient spark job scheduling based on deep reinforcement learning in cloud computing environments. *IEEE Transactions on Parallel and Distributed Systems*, 33(7):1695–1710, 2021.

[38] Kate Keahey, Jason Anderson, Zhuo Zhen, Pierre Riteau, Paul Ruth, Dan Stanzione, Mert Cevik, Jacob Colleran, Haryadi S. Gunawi, Cody Hammock, Joe Mambretti, Alexander Barnes, François Halbach, Alex Rocha, and Joe Stubbs. Lessons Learned from the Chameleon Testbed. In *Proceedings of the 2020 USENIX Annual Technical Conference (USENIX ATC '20)*. USENIX Association, July 2020.

[39] Alexandra Anna Lassota, Alexander Lindermayr, Nicole Megow, and Jens Schlöter. Minimalistic Predictions to

Schedule Jobs with Online Precedence Constraints. In Andreas Krause, Emma Brunskill, Kyunghyun Cho, Barbara Engelhardt, Sivan Sabato, and Jonathan Scarlett, editors, *Proceedings of the 40th International Conference on Machine Learning*, volume 202 of *Proceedings of Machine Learning Research*, pages 18563–18583. PMLR, 23–29 Jul 2023.

[40] Adam Lechowicz, Nicolas Christianson, Jinhang Zuo, Noman Bashir, Mohammad Hajiesmaili, Adam Wierman, and Prashant Shenoy. The Online Pause and Resume Problem: Optimal Algorithms and An Application to Carbon-Aware Load Shifting. *Proc. of the ACM on Measurement and Analysis of Computing Systems*, 7(3), Dec 2023.

[41] Russell Lee, Jessica Maghakian, Mohammad Hajiesmaili, Jian Li, Ramesh Sitaraman, and Zhenhua Liu. Online peak-aware energy scheduling with untrusted advice. *SIGENERGY Energy Inform. Rev.*, 1(1):59–77, January 2022.

[42] J. K. Lenstra and A. H. G. Rinnooy Kan. Complexity of Scheduling under Precedence Constraints. *Operations Research*, 26(1):22–35, 1978.

[43] Hongjian Li, Liang Lu, Wenhu Shi, Gangfan Tan, and Hao Luo. Energy-aware scheduling for spark job based on deep reinforcement learning in cloud. *Computing*, 105(8):1717–1743, March 2023.

[44] Pengfei Li, Jianyi Yang, Mohammad A. Islam, and Shaolei Ren. Making AI Less "Thirsty": Uncovering and Addressing the Secret Water Footprint of AI Models. *Communications of the ACM*, 2025.

[45] Shi Li. Scheduling to Minimize Total Weighted Completion Time via Time-Indexed Linear Programming Relaxations. In *2017 IEEE 58th Annual Symposium on Foundations of Computer Science (FOCS)*, page 283–294. IEEE, October 2017.

[46] Xinran Li and Zhaohao Ding. Cost Efficient Job Scheduling Scheme for Large Scale Data Center. In *2023 IEEE/IAS Industrial and Commercial Power System Asia (I&CPS Asia)*, pages 2267–2272, 2023.

[47] Erica Lin, Luna Xu, Suraj Bramhavar, Marco Montes de Oca, Sean Gorsky, Lingyun Yi, Arianna Groetsema, and Jeffrey Chou. Global Optimization of Data Pipelines in Heterogeneous Cloud Environments, 2022. arXiv:2202.05711.

[48] Liuzixuan Lin and Andrew A. Chien. Middlebox: Unlocking datacenter growth and grid decarbonization. In *Proceedings of the 2025 ACM Symposium on Cloud Computing*, SoCC '25, 2025.

[49] Wenyu Liu, Yuejun Yan, Yimeng Sun, Hongju Mao, Ming Cheng, Peng Wang, and Zhaohao Ding. Online job scheduling scheme for low-carbon data center operation: An information and energy nexus perspective. *Applied Energy*, 338:120918, 2023.

[50] Yiheng Liu, Hao He, Tianle Han, Xu Zhang, Mengyuan Liu, Jiaming Tian, Yutong Zhang, Jiaqi Wang, Xiaohui Gao, Tianyang Zhong, Yi Pan, Shaochen Xu, Zihao Wu, Zhengliang Liu, Xin Zhang, Shu Zhang, Xintao Hu, Tuo Zhang, Ning Qiang, Tianming Liu, and Bao Ge. Understanding LLMs: A Comprehensive Overview from Training to Inference, 2024. arXiv:2401.02038.

[51] Julian Lorenz, Konstantinos Panagiotou, and Angelika Steger. Optimal Algorithms for k-Search with Application in Option Pricing. *Algorithmica*, 55(2):311–328, August 2008.

[52] Biswaroop Maiti, Rajmohan Rajaraman, David Stalfa, Zoya Svitkina, and Aravindan Vijayaraghavan. Scheduling Precedence-Constrained Jobs on Related Machines with Communication Delay, 2020. arXiv:2004.10776.

[53] Hongzi Mao, Malte Schwarzkopf, Shaileshh Bojja Venkatakrishnan, Zili Meng, and Mohammad Alizadeh. Learning Scheduling Algorithms for Data Processing Clusters. In *Proceedings of the ACM Special Interest Group on Data Communication*, SIGCOMM '19, page 270–288, 2019.

[54] Yamini Mathur. Torgraphina: A scheduler for data processing during high-frequency job arrival using Upside Down Reinforcement Learning. Master's thesis, Iowa State University, 2023.

[55] Dirk Merkel. Docker: lightweight linux containers for consistent development and deployment. *Linux Journal*, 2014(239):2, 2014.

[56] Jesus Para, Javier Del Ser, and Antonio J. Nebro. Energy-Aware Multi-Objective Job Shop Scheduling Optimization with Metaheuristics in Manufacturing Industries: A Critical Survey, Results, and Perspectives. *Applied Sciences*, 12(3):1491, January 2022.

[57] Myoung-Ju Park and Andy Ham. Energy-aware flexible job shop scheduling under time-of-use pricing. *International Journal of Production Economics*, 248:108507, June 2022.

[58] Ferdinando Pezzella, Gianluca Morganti, and Giampiero Ciaschetti. A genetic algorithm for the flexible job-shop scheduling problem. *Computers & Operations Research*, 35(10):3202–3212, 2008.

[59] Bikas Saha, Hitesh Shah, Siddharth Seth, Gopal Vijayaraghavan, Arun Murthy, and Carlo Curino. Apache Tez: A Unifying Framework for Modeling and Building Data Processing Applications. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data (SIGMOD '15)*, pages 1357–1369, Melbourne, Australia, 2015. ACM.

[60] Varun Sakalkar, Vasileios Kontorinis, David Landhuis, Shaohong Li, Darren De Ronde, Thomas Blooming, Anand Ramesh, James Kennedy, Christopher Malone, Jimmy Clidaras, and Parthasarathy Ranganathan. Data Center Power Oversubscription with a Medium Voltage Power Plane and Priority-Aware Capping. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '20, page 497–511, 2020.

[61] James Schneider. AI to drive 165% increase in data center power demand by 2030, Jun 2024. Goldman Sachs Research.

[62] Navin Sharma, Sean Barker, David Irwin, and Prashant Shenoy. Blink: managing server clusters on intermittent power. *SIGARCH Comput. Archit. News*, 39(1):185–198, March 2011.

[63] Arman Shehabi, Alex Newkirk, Sarah J. Smith, Alex Hubbard, Nuoa Lei, Md Abu Bakar Siddik, Billie Holecek, Jonathan G. Koomey, Eric R. Masanet, and Dale A. Sartor. 2024 United States Data Center Energy Usage Report. Berkeley, CA, USA, 2024. Lawrence Berkeley National Laboratory.

[64] Jungeun Shin, Diana Arroyo, Asser Tantawi, Chen Wang, Alaa Youssef, and Rakesh Nagi. Cloud-native Workflow Scheduling using a Hybrid Priority Rule, Dynamic Resource Allocation, and Dynamic Task Partition. In *Proceedings of the 2024 ACM Symposium on Cloud Computing*, SoCC '24, page 830–846, 2024.

[65] Yu Su, Vivek Anand, Jannie Yu, Jian Tan, and Adam Wierman. Learning-Augmented Energy-Aware List Scheduling for Precedence-Constrained Tasks. *ACM Trans. Model. Perform. Eval. Comput. Syst.*, 2024.

[66] Yu Su, Shai Vardi, Xiaoqi Ren, and Adam Wierman. Communication-aware scheduling of precedence-constrained tasks on related machines. *Operations Research Letters*, 51(6):709–716, 2023.

[67] Eli Tan. Their water taps ran dry when meta built next door. *The New York Times*, July 2025. Accessed: 2025-12-10.

[68] Hajo Terbrack and Thorsten Claus. The generalized energy-aware flexible job shop scheduling model: A constraint programming approach. *Computers & Industrial Engineering*, 204:111065, June 2025.

[69] TPC-H. The TPC-H Benchmarks, 2018.

[70] Pablo Villalobos, Anson Ho, Jaime Sevilla, Tamay Besiroglu, Lennart Heim, and Marius Hobbhahn. Will we run out of data? Limits of LLM scaling based on human-generated data, 2024. arXiv:2211.04325.

[71] WattTime. WattTime. https://www.watttime.org, 2024.

[72] Philipp Wiesner, Ilja Behnke, Dominik Scheinert, Kordian Gontarska, and Lauritz Thamsen. Let's Wait Awhile: How Temporal Workload Shifting Can Reduce Carbon Emissions in the Cloud. In *Proceedings of the 22nd International Middleware Conference*, Middleware '21, page 260–272, 2021.

[73] John Wilkes. Google cluster-usage traces v3. Technical report, Google Inc., Mountain View, CA, USA, Apr 2020. Posted at https://github.com/google/cluster-data/blob/master/ClusterData2019.md.

[74] Qing Wu, Zhiwei Wu, Yuehui Zhuang, and Yuxia Cheng. *Adaptive DAG Tasks Scheduling with Deep Reinforcement Learning*, page 477–490. Springer International Publishing, 2018.

[75] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J. Franklin, Scott Shenker, and Ion Stoica. Resilient distributed datasets: a fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, NSDI'12, page 2, USA, 2012. USENIX Association.

[76] Matei Zaharia, Reynold S. Xin, Patrick Wendell, Tathagata Das, Michael Armbrust, Ankur Dave, Xiangrui Meng, Josh Rosen, Shivaram Venkataraman, Michael J. Franklin, Ali Ghodsi, Joseph Gonzalez, Scott Shenker, and Ion Stoica. Apache Spark: a unified engine for big data processing. *Communications of the ACM*, 59(11):56–65, Oct 2016.

[77] Yunfan Zhou, Xijun Li, Jinhong Luo, Mingxuan Yuan, Jia Zeng, and Jianguo Yao. Learning to Optimize DAG Scheduling in Heterogeneous Environment. In *2022 23rd IEEE International Conference on Mobile Data Management (MDM)*, pages 137–146, 2022.

[78] Yunhong Zhou, Deeparnab Chakrabarty, and Rajan Lukose. Budget Constrained Bidding in Keyword Auctions and Online Knapsack Problems. In *Lecture Notes in Computer Science*, pages 566–576. Springer Berlin Heidelberg, 2008.

Quota Marketplace: Dynamic Pricing for Efficient Allocation of ML Training Resources

Balasubramanian Sivan[*,1] Renato Paes Leme[*,1] Mihai Tiuca[*,1] Ian McFarlane[1] Vasilis Gkatzelis[1,2]
Nehal Mehta[1] Soheil Hassas Yeganeh[1] Vahab Mirrokni[1] Amin Vahdat[1] *([1]Google, [2]Drexel University)*

Abstract

The escalating demand for Machine Learning (ML) training resources in recent years has resulted in a substantial gap between the high demand and the available supply. Efficient allocation of these scarce and expensive resources is crucial for organizations to maximize their return on investment. Existing resource allocation mechanisms, like Karma [26], are designed to guarantee Pareto efficiency and max-min fairness in settings with dynamic (time-varying) user demands, but fail to preserve these key properties in the presence of demands with heterogeneous values. Given the ubiquity and inevitability of heterogeneity in organizational values of different workloads, effective resource allocation policies must accommodate these variations.

In this paper, we describe the design, implementation, deployment, and theoretical analysis of Quota Marketplace, a market-based mechanism to efficiently allocate ML training chips (like GPUs), explicitly addressing scenarios with demands of heterogeneous value. We detail the implementation of this mechanism within Google and present metrics that demonstrate its impact. We also discuss many business-critical requirements that the Quota Marketplace handles quite effectively, and document the gains and opportunities it has unlocked. We establish theoretically how this market-based approach achieves the essential properties of Pareto efficiency and max-min fairness by allowing the users to express the value of their workloads and enabling dynamic resource pricing based on supply and demand fluctuations. Ultimately, the market facilitates resource allocation that aligns with organizational priorities.

1 Introduction

The demand for Artificial Intelligence (AI) compute has experienced unprecedented growth, increasing by over tenfold annually for eight consecutive years, resulting in a 100 million-fold increase over that 8-year period [1]. Major technology corporations have publicly disclosed substantial capital expenditure (CapEx) budgets, tens of billions of dollars each for 2025, to address this growing demand. This exponential growth in demand creates a widening gap between the demand and supply of AI compute resources, specifically Machine Learning (ML) accelerators like GPUs. Consequently, due to the high cost and limited availability of these resources, organizations must implement return-on-investment (ROI) maximizing, efficient allocation strategies for ML accelerators among individual teams and products. In this paper, we describe Quota Marketplace (QM) – a market-based system designed, implemented, theoretically analyzed, and fully deployed at Google, to allocate ML chips to teams across business units. The deployed system allocates several hundreds of thousands of ML accelerators for *ML training workloads* across all the business units in Google. The footprint of the Quota Marketplace is a double-digit percentage of the *entire ML fleet of Google*.

We begin with an overview of QM's position in the ML scheduling stack and the notion of pools. A detailed description of QM concepts and implementation follow in Section 2.

QM and the ML Training Scheduling Stack in a Nutshell. ML compute resources (ML accelerators or ML chips), much like non-ML compute resources (CPUs), have traditionally been partitioned into *static* pools. Each pool corresponds to a business unit (BU) with distinct goals, cost accounting, and resources, and the pool administrators/planners explicitly assign accelerators to teams within that BU. The demand assessment and prioritization procedure typically takes place quarterly or semi-annually and it determines the allocation of chips across pools and across teams within a given pool. This assignment then remains more or less static (especially the distribution across pools) until the next assessment.

Given the rising prominence of ML training workloads and their need for batch compute (which tolerates preemptions much better than serving workloads)—the Quota Marketplace was conceptualized to make the accelerator allocations both dynamic and agile, and economically more efficient, by incor-

[*]Equal contribution.

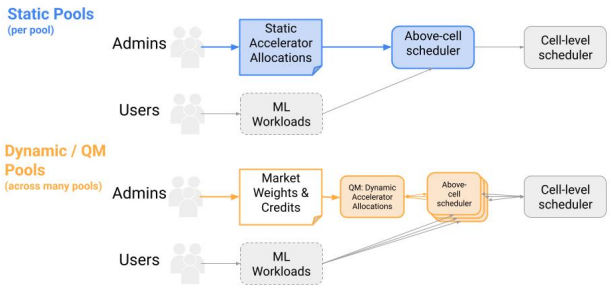


Figure 1: QM in the ML (Training) Scheduling Stack

porating a cross-pool view into allocation decisions. Figure 1 is a short summary of QM's position in the ML scheduling stack. Conceptually, when compared to static pools, there are two main differences in a *dynamic pool* (a) QM replaces static accelerator allocations with dynamic ones, and (b) QM makes these allocation decisions across all the dynamic pools instead of the per-pool allocation decisions in the static pools.

In dynamic pools, which is where QM operates, company-level administrators are tasked with allocating market weights to the different pools that reflect the business impact delivered by these units. Per-pool administrators are tasked with allocating credit, which are location agnostic, resource-type agnostic tokens, to the various teams within their business unit, which again reflect the impact the various teams deliver. The toilsome, communication-heavy, quarter-long human-driven negotiations involved in the traditional prioritization exercises of static allocation are now replaced by supply-demand-based resource allocation decisions both across teams in a pool, and also across pools, at a frequency of roughly every 1 minute. Beyond the several orders of magnitude more-responsive resource allocation that QM brings, QM's cross-pool view while clearing supply and demand automatically makes the allocation economically more efficient.

In a little more detail, QM determines resource allocation by computing clearing prices for each resource type at any given instant. Prices are dynamically chosen to equate *real-time* demand from scheduler queues with the *real-time* supply of the relevant ML accelerator. The market "clears" roughly every minute (often faster than that). By comparing real-time team bids generated by an automated bidding algorithm (that QM provides teams with) against these prices, QM decides which teams receive resources and in what quantity. Consequently, quota in pools (aka dynamic pools) is not guaranteed, but is derived from market conditions, making it well-suited for exploratory workloads and tasks with flexible SLAs, as is the case with most ML training workloads.

We now discuss the many benefits and opportunities unlocked by the QuotaMarketplace.

Agility in the Face of Highly Fluctuating Supply and Demand. The available supply of ML chips is highly dynamic. New hardware capacity comes online daily, often on an imperfect schedule. Resources are also frequently and unpredictably clawed back, at short notice, for a variety of reasons, including, for example, being able to serve production load due to a sudden surge in traffic. On the other hand, human-driven demand assessment and prioritization procedures, as mentioned earlier, are typically conducted quarterly or semi-annually. To maintain allocation speed, various processes used to rely on readily available resource buffers that remain significantly underutilized and unprioritized. A robust system must respond to both rapid capacity addition and swift clawback/reallocation in the order of minutes, rather than quarters. Doing so would directly contribute to significant reduction in the buffer maintained by production teams, as they could promptly claw back resources upon detecting impending traffic surges. As we make the point with data in Section 3, this reclaimed "bonus" capacity is substantial, sometimes on the order of the total committed capacity, and its underprioritized usage results in significant opportunity cost. In other words, QM results in a big increase in company-prioritized occupancy of chips, rather than opportunistic occupancy of chips (which can be thought of as randomly assigning chips), which we demonstrate with data in Section 3. That said, QM also increases the overall occupancy of chips (not just company-prioritized occupancy), because it avoids the siloing inherent to static pools (also demonstrated with data in Section 3). Siloing results in unused capacity, which is often not easily available to use even opportunistically for other pool's workloads.

The demand for ML chips is no less dynamic than their supply. Indeed, deriving the specific ML chip demand for each team directly from the company's business priorities is a challenging task. Our fluctuating usage patterns range from scheduled model training runs to short-duration bursts for demos, exploratory research, and conference submission sprints, exacerbating this problem. Therefore, individuals responsible for resource allocation at a company level (i.e., across all the pools), while knowledgeable about the high-level business goals, often lack granular insight into the time sensitivity, spatial flexibility (i.e., where chips are needed), usage volume, and relative priority of individual requests. Conversely, engineering teams understand the detailed requirements of each workload but lack the broader perspective needed to prioritize across different projects and business units. Therefore, the inherent assumption behind the static pools' prioritization procedures, that we are able to get reasonably accurate predictions of chip requirements for an entire quarter, is highly questionable. Further, regardless of how good our predictions are, a static allocation of chips is very suboptimal in the presence of highly dynamic demand patterns. What's needed is a mechanism to swiftly move allocations in response to real-time demands. QM accomplishes this by enabling continuous, dynamic bidding based on real-time needs.

Enhanced Productivity and Engineering Resource Conservation. QM significantly reduces the frequency of teams having to negotiate with their BU's pool administrators for more capacity in the middle of a quarter or a half year. In static pools, when there is a demand surge (which, as discussed, is not uncommon), it inevitably leads engineers to interact with the administrators to temporarily get more capacity. It is infeasible to solve this problem by statically allocating chips that meets everyone's *peak* demands. Dynamic allocation is inevitable. In QM, these negotiations are relatively rare, and even when it is necessary, QM makes the administrator's tasks much easier — simply increase the credit allocation of the concerned team (if their ask is deemed deserving), without concerns for which team to reclaim chips from, to fund this request. Unlike chips and chip hours, credits are (a) not zero-sum and (b) {resource-type, location}-agnostic. Therefore increasing one team's credits without altering any other team's credits is perfectly valid.

Incentive Alignment in the Face of Intense Scarcity. ML compute resources are very scarce (demand far exceeds supply), and scarcity of resources inevitably drives participants toward strategic behavior. Consequently, a robust resource allocation mechanism must align user incentives with the designer's objectives, such as efficiency and fairness.

To achieve these goals, resource allocation systems should incentivize demand deferral to less congested time periods, geographic locales, or resource-types. A good mechanism should preempt "use-it-or-lose-it" scenarios by incentivizing teams to relinquish resources that they do not have a high value for so that they can be consumed by teams with more pressing needs instead. Similarly, given the variety of available resource types that arises from different accelerator generations, the mechanism should incentivize teams to use less congested types of resources.

Finally, the system must provide clear signals that enable users to plan effectively where and when to run their workloads. In QM, prices serve a dual purpose: they provide (a) economic incentives, and (b) a transparent signal of resource congestion and how it varies across time and location.

Guaranteed Efficiency and Fairness. When allocating scarce resources, a central systems goal is to achieve efficiency and fairness properties. A fundamental notion of efficiency is Pareto efficiency: an allocation is Pareto efficient if there is no other allocation that would result in (a) weakly higher utility for all users and (b) strictly higher utility for at least one user. A less stringent requirement, directly implied by Pareto efficiency, is to avoid situations where resources are unused despite existing demand. In terms of fairness, the standard goal when allocating computational resources is to achieve (approximate) max-min fairness, which aims to maximize the minimum amount of resources (or utility) any user is allocated. In a large organizational context, this is commonly extended to a weighted max-min notion that normalizes the "fair share" for each team based on its potential to positively impact core business priorities. In Section 4 we define Pareto efficiency and max-min fairness precisely, and show how QM guarantees both.

Shock Absorber. QM functions as a shock absorber for the entire corporation, capable of handling significant capacity fluctuations arising from external events. Whether due to the launch of a marquee LLM application that has gained significant traction from customers, requiring hundreds of thousands of accelerators, a high-priority external demo, or surges in cloud customer demand, the system effectively absorbs these shocks without manual intervention. This is not just the earlier point about the agility in handling these requests, but also about distributing the benefits or pains from additions or clawbacks in the fairest way possible according to organizational priority. Shocks can go the other direction as well: QM enables the economically efficient usage of highly fragmented resources added to the system for temporary periods of time, either from serving buffer, or from holding pools.

Prior Approaches. Two broad classes of approaches in prior work are "static allocation" and "chip-hour" mechanisms.

Static allocation. This is the idea behind the traditional static pools discussed earlier. Consider a simple setting with a single type of resource, like an ML accelerator, which has to be divided among a number of users. One of the most popular solutions is to statically allocate [2, 25] this resource by splitting it among the users (or teams) independent of their demands. It effectively creates siloed pools, it gives strong availability isolation, and it upholds fairness. However, crucially, it doesn't satisfy Pareto efficiency because it ignores demands. Max-min fairness [10, 12–16, 19, 20, 22] addresses this problem, and simultaneously achieves Pareto efficiency and fairness, by incorporating demand into decision-making: it maximizes the minimum allocation across all participants, but subject to not allocating more than their demand. But the efficiency guarantees of max-min fairness mechanisms heavily depends on the assumption that user demand is static over time, which is unrealistic for most real world systems [26]. In settings with dynamic demand, static allocations are bound to be far from Pareto efficient.

Chip-hour mechanisms. Aiming for more efficient allocations in the presence of dynamic user demands, "chip-hour" mechanisms determine the amount of access to the resources that each user should be allocated, but provide the users with some flexibility regarding when to use their access rights. A notable example is the Karma mechanism, introduced by Vuppalapati et al. [26], which asks the users to report the amount of resources that they need in each round, and then determines how the resources will be allocated as a function of each user's access history; specifically, this mechanisms strictly prioritizes users who have had the least amount of access to the resources in the past. This elegant mechanism provides incentives for users to request a resource only if they

value it, it allocates resources only to users who need them, and it strives to optimize max-min fairness by prioritizing the user with the minimum amount of resource access in the past. However, despite the very significant improvements that this mechanism achieves, it has an important limitation: it assumes that the user's value for each of these requests is uniform over time and across users, which is quite unrealistic. In fact, as we show in Section 4, the appealing properties of all chip-hour mechanisms break down even for bi-valued instances, where each agent's value for a request can take just one of two values, say HIGH or LOW. Different workloads necessarily hold different values in an organizational context, and it is vital for a resource allocation mechanism to accommodate this. Furthermore, chip-hours are still zero-sum quantities, like chips themselves, because they are still tied to a specific resource type at a specific location. Allocating more chip hours to one team necessarily involves reclaiming these chip hours from some other team(s)[1].

Our Contributions. Our main contributions are:

1. We describe the design, implementation and deployment of Quota Marketplace in Google. The deployed system has several thousands of daily active users, and several billions of accelerator hours already consumed via the system.

2. We provide metrics from the actual deployment of QM.

3. We provide a theoretical analysis in a simplified single-resource setting which exhibits the shortcomings of the chip-hours mechanisms, especially in the presence of heterogeneous user values (an inevitable characteristic of workloads in realistic settings). We analyze our market-based mechanism in the same setting, and show that it guarantees Pareto efficiency and approximate max-min fairness, even in settings with heterogeneous values.

4. Finally, we present several avenues for future work, some that require solving new problems, and others that require careful UX design to ensure that the system does not compromise user-friendliness while aiming to be robust to manipulations. We also describe how we address some of the challenges raised in prior work (e.g. [21]) regarding the integration of markets and systems.

While several benefits of QM have already been discussed earlier, we describe a few more in the paragraphs below.

The Power of Dynamic Pricing. The primary shortcoming of prior works in the face of heterogeneity in values is that they don't provide users a way to express their utility (i.e., their value for being allocated the resource at each point in time), so every user request is treated with the same priority. In the presence of heterogeneous values, we instead need a mechanism which allows users to express these values and then allocate the available resources accordingly. This is precisely what the Quota Marketplace achieves. Given that real-time prices for different resource types are published in a company-wide dashboard, deciding to not start a workload when the price exceeds a certain amount is a definitive way to express that one's value for that workload is below a certain price. Further as already discussed, prices provide a clear incentive for users to shift their demands towards cheaper (and, hence, less congested) resources that they still highly value.

The Benefits of the Credit Abstraction. The role of the credits allotted to the users in our market-based mechanism and their use toward dynamic resource pricing, provide some important benefits relative to static allocation and chip-hour mechanisms. While we briefly alluded to these in the earlier discussion in various contexts, we collect them all below in a little more detail.

First, *credits are fully decoupled from physical resources*: indeed, the sum of credits in circulation by our mechanism is not constrained in any way by the number of available physical resources. This property enables the market to effortlessly handle supply additions and clawbacks: *the addition or clawback of hundreds of thousands of chips does not require the market to adjust the agents' credit balance in any way*: it simply runs the same market clearance algorithm in the next cycle, using the updated supply value when equating supply and demand to compute clearing price. There is no equivalent to this in static allocations or chip-hour mechanisms.

Second, very related to first, but subtly different, *credits are not zero-sum quantities* — this directly enables the market to reprioritize in minutes. For instance, granting a certain team or business unit additional power simply requires minting more credits and handing it over to that team, or increasing the market weight of that BU. In contrast, *chips or chip-hours are both zero-sum* since the physical resources they are tied to are, of course, zero sum. As a result, reprioritization to grant additional power to one team necessarily involves determining which teams to withdraw supply from.

Third, *credits are agnostic to the type, location, or other attributes of each resource, whereas chips and chip-hours are tied to a particular chip and location*. Therefore, when allocating credits, the allocation committee deals with a single-dimensional problem of deciding who gets how much purchasing power. Allocating chips or chip hours, on the other hand, requires a separate resource allocation decision for each chip type and location. Further, it is easy for the allocation committee to measure the return on investment from different teams based on a single-dimensional quantity of credits. Otherwise, the committee has to wrestle with the very hard problem of comparing apples and oranges when evaluating the impact of one team that was allocated one set of chip types in some set of locations relative to another team that received a different set of chip types and locations.

[1] One can resort to oversubscription, but of course that has it limits too, and the core point that chip hours are zero-sum quantities remains the same.

Organization of the Paper. In Section 2, we give a formal description of the Quota Marketplace system and its implementation, along with the features in the market that prevent market abuse. Section 3 presents metrics from the deployment of the Quota Marketplace and demonstrates how the desiderata that we have described thus far are satisfied by the market-based mechanism. In Section 4 we provide our theoretical analysis. In Section 5 we discuss some of the old and new challenges, as well as new opportunities and interesting open problems for future research, and in Section 6 we conclude with an overview of related work.

2 Quota Marketplace System

The Quota Marketplace is an iterative resource allocation system based on dynamic pricing using a generated currency of "credits". To describe how this market operates, we first provide some core concepts.

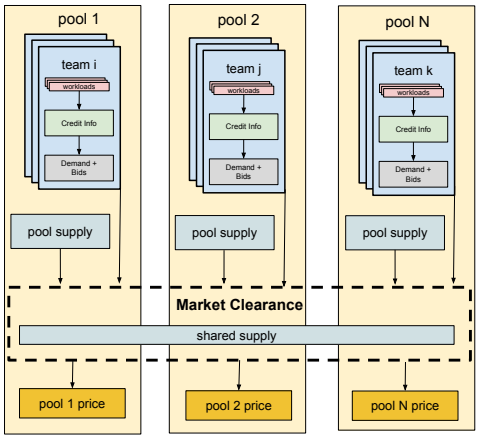


Figure 2: The high-level concepts of Quota Marketplace

Supply. Supply (or resources) of the marketplace correspond to specialized ML-accelerator chips. The chips are distributed across various geographic locations (which we refer to as cells) and are of different generations (which we refer to as resource types or chip types). Each cell has a certain capacity of each chip type, which varies over time.

While the intention of QM is to manage supply shared across all pools, for legacy reasons the system also supports pool-specific supply, which is supply previously purchased and owned by each business unit to which this unit has preferential access. Indeed, Figure 2 shows that market clearance takes into account both pool supply and shared supply.

Teams. Users are grouped into teams and submit demand to the system in the form of individual workloads. As workloads are submitted, they enter the team priority queue, allowing for relative prioritization. The team is the fundamental unit within the system to which resources are allocated.

Pools. Pools are groups of teams representing business units within the larger organization with their own goals, cost accounting, governance, and resources.

Income and Balance. Pool administrators assign credit income to teams to reflect their relative priority in the pool. Income is defined as credits to be deposited into a team's account at a certain rate. As teams incur charges for resource usage, their credit balance gets reduced. Any newly assigned income (say for a new team, or increased income for an existing team) increases the rate of minting new credits, thereby increasing the credits available within the pool.

Each pool independently manages its credit allocation much like a country manages its own currency. However, the total purchasing power of a pool remains fixed regardless of the number of credits minted within the pool. This invariance is enforced by a mechanism called *market weights*.

Market Weights. Google adjusts priority between pools by assigning market weights, which correspond to the total purchasing power of each business unit. This is implemented by mapping the local credits of each pool to a standard global unit, similar to currency conversion in international trade.

Formally, if a pool k with an assigned market weight of W_k, mints an sincome of C_k credits per unit time, each local credit owned by pool k is valued at W_k/C_k standard global units.

This design decouples global prioritization (managed by Google-level administrators via W_k) from intra-pool allocation (managed by local administrators via C_k). However, this exposes teams to monetary dynamics akin to those in international economics. E.g., if pool administrators mint excessive credits (increasing C_k) while the market weight W_k remains constant, the pool experiences inflation and credit devaluation.

User Journey: Reducing Users' Cognitive Burden via Automated Bidding. Incentivizing users to change their behavior necessarily entails imposing at least a small amount of cognitive awareness from the users. That said, QM is designed to offer most of its incentive benefits with very little investment of time from users. A user's journey involves starting a job, using exactly the same workflow as they did in the pre-QM world. Next, the user places their job in a priority queue of active jobs from their team, with the possibility of being asked to adjust their job's position by their team admin, if they so feel. Once a job is on the queue, the most important task of submitting a bid for the job is handled by an automated bidder. The system allows users to set team-level settings and job-level settings for bidding. These include (a) limit orders, which is a job-level setting, letting users specify the maximum price they are willing to pay for that job (b) income-leverage, which is a team-level setting that upper bounds the maximum bid allowed from a team at the product of the team's income and the income-leverage (c) a tool that lets team admins to configure policies for their team's jobs, from a simple priority queue to order their jobs to more sophisticated policies to dy-

namically manage importance of their jobs. Users throughout the company have access to a dashboard with real-time prices of all chip types, price history for the past several months, their team's credit balance, income rate, credit burning rate to help them make informed decisions while deciding on these settings. The automated bidder descends the priority queue for each team, estimates the cost of running each each job, and then, governed by the settings, aggregates costs to compute team level bids. In practice, system provided defaults work quite well for the settings, and indeed most users rely on them, except in periods of high price volatility, which we explicitly mitigate as described in later sections (see "Volatility and Churn" in Section 2.1 for example).

Demand and Bid Formulation. In every market cycle, the system aggregates the demand of queued workloads for each team. We denote d_i^r as the aggregate demand of team i for chips of type r, where r encodes both the hardware generation and the specific cell. The automated bidder translates the team's balance, income rate, and settings into a scalar bid b_i^r according to relative workload priority. This value represents the total credits the team is willing to spend to secure resource r in the current cycle. Because the system resolves markets per resource type, the agent automatically proportions the team's total buying power across the relevant markets before clearing occurs.

Market Clearing. Let b_i^r and d_i^r denote the bid and demand, respectively, of team i for resource type r. For each pool k, we compute a specific clearing price p_k^r to equate the pool's internal demand with its available supply. Pool prices differ from the global market price because pools effectively blend their own static resources (which are free to the pool) with resources purchased from the shared market.

For clarity, we assume all bids are normalized to standard credit units, i.e., we ignore currency conversion rates, as it doesn't change the equations much except for adding notational clutter. Let s_k^r denote the static supply owned by pool k, and let z_k^r denote the supply purchased by the pool from the shared market supply at a reference price p_*^r. The pool sets its internal price p_k^r so that it breaks-even: i.e., the credits it generates from its teams for the $s_k^r + z_k^r$ chips they consume is equal to the credits it has to pay to the central market for having purchased z_k^r credits from it. This yields the following break-even condition:

$$p_k^r(s_k^r + z_k^r) = p_*^r z_k^r \quad \Longrightarrow \quad p_k^r = p_*^r \frac{z_k^r}{s_k^r + z_k^r} \tag{1}$$

Equation (1) highlights the discount effect: if a pool relies entirely on shared resources ($s_k^r = 0$), the pool price equals the reference price ($p_k^r = p_*^r$). Conversely, if a pool relies heavily on owned supply, the internal price is significantly lower than the market reference price.

Given a pool price p_k^r, team i demands $\min(d_i^r, b_i^r/p_k^r)$ units, i.e., a team's demand is simply the minimum of what it actually needs (d_i^r) and what it can afford at a given bid (b_i^r/p_k^r). To clear the pool-local market, the total demand from all teams i in pool k must equal the total available supply:

$$\sum_{i \in \text{Pool } k} \min\left(d_i^r, \frac{b_i^r}{p_k^r}\right) = s_k^r + z_k^r \tag{2}$$

For any fixed reference price p_*^r, Equations (1) and (2) have just two unknowns, namely p_k^r and z_k^r, or more technically $p_k^r(p_*^r)$ and $z_k^r(p_*^r)$ as these are functions of p_*^r. Intuitively, this solution represents the equilibrium between the pool's internal supply-demand clearance and its blended supply cost. Equation (2) is the pool's internal supply-demand clearance, where the LHS is the pool's *aggregate demand curve*, mapping the internal price p_k^r to the total quantity demanded by teams, and the RHS is the pool's total supply arising from it's own supply and purchased supply. Equation (1) represents the *cost-blending curve*, which dictates how the internal price p_k^r must rise as the pool purchases more shared supply z_k^r. The intersection of these two curves determines the unique purchase volume z_k^r and internal price p_k^r for that specific market reference price.

Finally, we are left with determining p_*^r. Let Z^r be the total capacity of resource r available in the shared marketplace. We seek the lowest price p_*^r such that the aggregate demand from all pools does not exceed the shared supply:

$$\sum_k z_k^r(p_*^r) \le Z^r \tag{3}$$

Since the pool demand $z_k^r(p_*^r)$ is monotonically decreasing with respect to price p_*^r, the price p_*^r is efficiently computable using binary search. If demand is low (i.e., even at zero prices there is enough supply: $\sum_k z_k^r(0) \le Z^r$), the price p_*^r is set to zero.

The algorithm described is implementable by a nested binary search, where the inner binary search is used to solve equations (1) and (2) and the outer level to solve equation (3). With the right pre-processing, we implement the algorithm in nearly-linear time using the algorithmic technique in [17].

Resource Allocation. Once the specific pool price p_k^r is established, the allocation x_i^r for each team i is calculated as:

$$x_i^r = \min\left(d_i^r, \frac{b_i^r}{p_k^r}\right)$$

This allocated quota is subsequently passed to the ML scheduler for execution.

Charges and Capacity Import In separate cycles from the market, teams are charged for the resources that they utilize and capacity is read from the resource management system. In the charge cycle, credits are deducted according to the resources that the team actually occupied since the last charge cycle. Capacity for all resource types read are persisted for the market to read in each cycle. Both of these cycles are run less frequently than the market cycle.

2.1 Market Implementation

The Quota Marketplace is implemented as a global coordinator layered atop the hierarchical scheduler stack (comprising above-cell and cell-level schedulers). The system operates as a monolithic binary replicated for availability. While the architecture could theoretically support sharding—distributed bidding across teams and distributed clearing across resource types—we opted for a centralized design to prioritize simplicity and coherency in the initial deployment.

Centralizing the market logic in a single process introduces a potential single point of failure with a large blast radius. We mitigate this risk by decoupling quota generation from job execution. QM does not directly schedule jobs; instead, it generates quota inputs for the underlying scheduler. In the event of an outage, the system gracefully degrades by reverting to static pool scheduling using the last valid quota snapshot calculated by the market. Consequently, while the scheduler may rely more heavily on opportunistic rather than prioritized allocation—reducing efficiency—the system avoids catastrophic downtime.

Modular Design. As illustrated in Figure 3, QM employs a modular architecture where distinct phases of the market cycle are isolated via strict interfaces. While the system communicates with external components—specifically the scheduler and the resource management system—via standard Remote Procedure Calls (RPCs), internal interactions utilize the same framework primitives without incurring the overhead of network serialization. This design choice allows us to inherit robust distributed systems capabilities, such as deadline propagation, request cancellation, structured logging, and monitoring, within a unified process. Figure 3 highlights these external boundaries: the system ingests demand signals from the above-cell scheduler (`SampleDemand()`) and physical capacity data from the resource management system (`ImportCapacity()`), subsequently pushing allocated quotas back to the scheduler (`UpdateSchedulerInput()`).

Periodic Routines. The QM system executes three independent periodic routines: the market cycle, accounting (processing charges and income), and capacity ingestion. Each routine executes within a strict periodic closure that enforces serial execution, preventing overrun. The execution frequencies are tuned to the volatility of the data: the market cycle runs at high frequency ($<$ 1 minute), income and charge accounting runs every 5 minutes, and capacity ingestion occurs roughly hourly ($O(h)$). All critical system state—credits, capacity, quota, and demand—is persisted in a strongly consistent datastore. Market cycles are keyed by timestamp to ensure consistency across potentially overlapping executions.

By decoupling charging, income generation, and capacity import from the market, the system is able to react much more rapidly to demand changes. Capacity is relatively static on the order of minutes (emergency capacity changes can still be manually propagated in $O(m)$). Income and charge resolution indeed affect bidding power each cycle. But there is a feature of the market called *minimum affordable duration* that ensures that the 5 minutes frequency of processing income and charges is more than enough to make bids fluctuate quickly *only because of demand fluctuations*. Minimum affordable duration is a measure we introduced to ensure market stability and prevent abuse. It enforces that a team can bid an amount only if the burn rate of credits at that bid is sustainable for the next 120 minutes. Therefore, income and charges not being processed for the next 5 minutes will not affect the bidding power of any team.

Replayable Market. The core market clearing logic is encapsulated as a stateless, pure function, `RunAuction`. All function inputs and outputs are persisted alongside auxiliary system data. Isolating the auction logic from the runtime environment enables deterministic offline simulation and replay of historical iterations using the exact same RPC handlers. This architecture has proven indispensable for reproducing production incidents, iterating on mechanism design, and verifying the correctness of changes.

Deployment and Verification. The monolithic binary architecture complicates standard canary deployment strategies. To compensate, we rely heavily on offline simulation and deterministic replay. Furthermore, although the binary is deployed globally, the market is logically sharded by resource type. This allows us to gate new features or configuration changes incrementally per resource type, enabling granular rollouts despite the unified deployment artifact.

To prevent regressions and ensure high-level correctness, we enforce a suite of invariant checks on every change. These checks leverage the market's replayability to compare test outputs and key metrics against historical baselines.

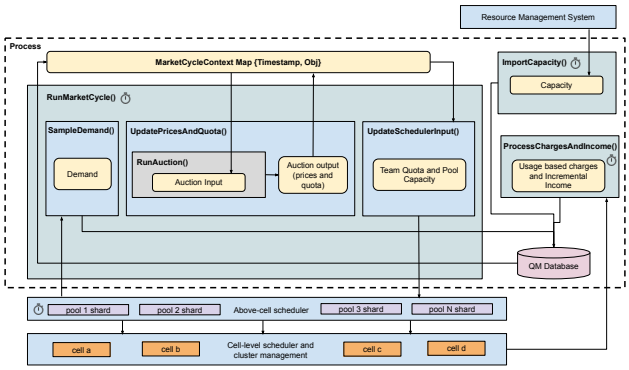


Figure 3: Architecture and components of the Quota Marketplace

Spatial Flexibility and Global Quota The system provides a location abstraction allowing users to submit *spatially flexible* workloads (i.e., workloads that can run in any cell). We

initially considered sharding the market by both resource type and cell. However, our analysis revealed that supply at the cell level is highly fragmented; running independent cell-level auctions yields "thin" markets with highly volatile pricing and unstable scheduling outcomes (smaller cells averaged $\sim$3% poorer occupancy than larger ones). To maximize market liquidity, we aggregate demand and supply into a single *Global Cell* abstraction, run a global market per resource type, and allocate global quota.

This abstraction introduces a trade-off between market stability and placement granularity. Global quota guarantees that capacity exists *somewhere* in the fleet, but not necessarily where to a specific workload's is location constrained. Consequently, in roughly about 10% of the cases, we observe cell-level contention despite being allocated sufficient quota in the global market. We accept this reduction in theoretical market efficiency as a necessary cost to ensure price stability and a usable experience for engineers.

Scheduler Interaction and Sharded Pool Scheduling. QM began operating as a naive asynchronous coordinator atop a hierarchical scheduler. It published quota and capacity updates, which the *Above-Cell Scheduler* consumed to place logical workloads in some cell, followed by the *Cell-Level Scheduler* which performed physical bin-packing of the workload in the chosen cell. Because these layers operated asynchronously, state conflicts were inevitable. We deliberately chose this loosely coupled design over deep integration to accelerate the initial deployment and isolate the market logic from the critical path of the scheduler kernel. While this introduces consistency challenges, it allowed us to iterate on the market mechanism without destabilizing the underlying cluster management stack.

Relatedly, due to legacy constraints, the above-cell scheduler made separate scheduling decisions for each pool. This forced the QM system to physically move capacity between pools so that the above-cell scheduler can realize the quota allocated by the market. This operational lag between the market (which allocates quota immediately) and the resource manager (which moves capacity slowly), compounded mismatches between workload constraints (e.g. global quota vs. location constraints discussed earlier) and available supply.

We very recently mitigated these issues by moving up the above-cell scheduling logic into the market system and thereby unifying scheduling across pools. This removes the asynchrony between the market and the scheduler and removes the expressivity limitations imposed by the quota interface. This change allows the market to define the inevitable contended cell queuing behavior described in the section above. Importantly, it also paves the way for the market to have a variety of workload-level signals, which it didn't have earlier. This architectural shift has major current and future benefits, but comes with a tradeoff: it trades off increased failure domain risk (centralization).

Market Cycle Latency Improvements. The latency of the market cycle determines the system's control loop frequency and its ability to react to demand spikes. Through optimizations in the auction algorithm and RPC handling, we reduced the p50 market cycle latency from 100s to under 30s. End-to-end scheduling latency acts as a "tax" on the system, resulting in a measurable 1.3% occupancy wastage across the fleet. As another significant benefit of the architectural shift discussed in the preceding paragraph, we were able to reduce the end-to-end latency to approximately 30s. This recovers some of the occupancy wastage and improves engineering velocity by reducing the time between workload submission and feedback.

Volatility and Churn. Because QM adjusts quotas dynamically based on price fluctuations, it naturally incurs somewhat higher preemption rates than static systems. To prevent high-frequency volatility, as previously mentioned, we enforce that teams must demonstrate sufficient credit balance to sustain a bid for a minimum duration. This smoothing mechanism ensures that bids do not oscillate with the granular income and charge cycles. We treat market stability not merely as an optimization but as a critical usability requirement; users rely on stable price signals to interpret scarcity effectively. Beyond this, there are a host of other scheduling algorithm features designed to mitigate scheduling churn.

Workload Boundaries and Cube Aware Scheduling. While the market models demand as a scalar fluid, physical ML workloads are discrete and topological (e.g., a 64-chip request often requires a specific $4 \times 4 \times 4$ cube, corresponding to 16 interconnected machines). The cell-level scheduler handles the physical bin-packing and defragmentation of these shapes. QM is currently *topology-oblivious*: it assigns aggregate quota without guaranteeing that the free chips form a contiguous cube. This abstraction gap inevitably leads to defragmentation-based preemptions where quota exists but geometry does not, and today it arises roughly in about 1% of the cases. For the initial deployment, we accepted this inefficiency to avoid the complexity of solving a multi-dimensional knapsack problem in the market loop. Future iterations aim to incorporate topology-aware pricing to bridge this gap.

Auxiliary Resource Requirements. QM employs a *principal resource pricing* model: it explicitly prices only ML accelerators, while auxiliary resources (CPU, RAM, disk, network) are allocated proportionally to the accelerator quota. This design is grounded in the observation that accelerators are the dominant bottleneck resource for ML training at Google. Further, this design choice avoids a combinatorial marketplace, and presents users with a single-dimensional price signal, significantly reducing the cognitive load required to participate in the market. While combinatorial auctions could theoretically offer higher allocative efficiency, we prioritize the operability of a simpler pricing model.

3 Deployment Metrics and Takeaways

In this section, we quantify the gains unlocked by the deployment of Quota Marketplace, Google-wide. There are many dimensions along which we discuss improvements. For some cases, the comparison we present is with the conventional static pools. In other cases, when there are entirely new opportunities unlocked by the Quota Marketplace, we discuss them with metrics.

Occupancy & Company-Prioritized Occupancy. In the Introduction, we described how QM would increase both company-prioritized occupancy and overall occupancy. Here, we quantify these increases. Figures 4 and 5 illustrate this increase by showing occupancy fractions of prioritized and opportunistic allocations of the same, very popular, resource type in two different pools: Quota Marketplace and conventional, respectively. Both pools are comparable in their sizes with hundreds of thousands of chips each. The difference in occupancies is stark. The average overall occupancy (i.e., prioritized + opportunistic) in the conventional pool is at 75%, and the same quantity in the Quota Marketplace pool is at 93%. This marks a ∼24% increase in overall occupancy. Also, average opportunistic occupancy in the Quota Marketplace pool is at a small 5.67%, as opposed to 33.5% in the conventional pool. This fully validates our expectation that the market-based system would substantially eliminate opportunistic occupancy, and what remains of it would be small, arising from some system inefficiencies.

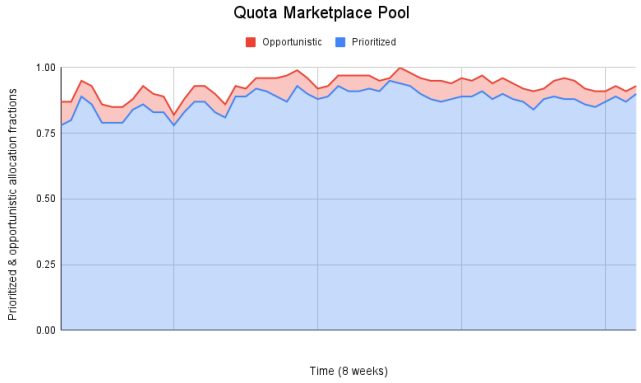


Figure 4: Occupancy fractions of company-prioritized and opportunistic allocations in the Quota Marketplace pool for a popular resource type.

Time Varying Supply. The available pool of compute resources often experiences significant fluctuations driven by evolving business priorities and supply chain dynamics. Common events include supply reductions via clawbacks for reprioritization, temporary capacity injections, reallocation of resources across different geographical locations or accelerator

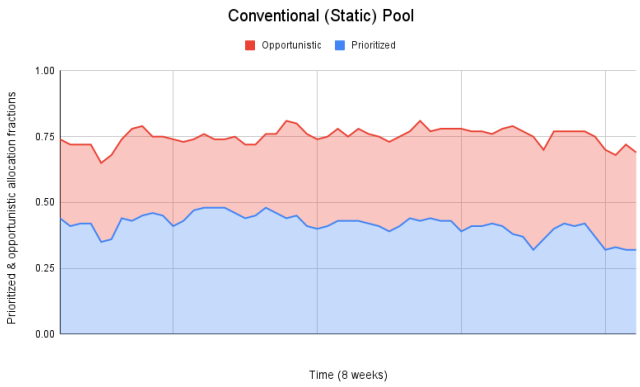


Figure 5: Occupancy fractions of company-prioritized and opportunistic allocations in the conventional (static) pool for the same popular resource type as in Figure 4.

types, and changes to anticipated delivery timelines for new hardware. These frequent supply adjustments pose a challenge to traditional resource allocation systems, often resulting in slower user adaptation and, consequently, suboptimal fleet utilization. Figure 6 shows that supply variations over time are substantial, and are not just a theoretical worst-case situation to reason about. While the figure captures shifts in accelerator types, note that supply fluctuations extend beyond this. For instance, even within a single accelerator type, numerous changes in geographical location (among the 3-digit number of data centers) can occur.

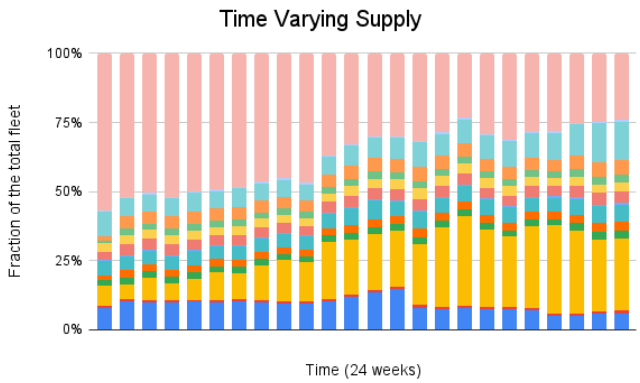


Figure 6: The total fleet managed by Quota Marketplace broken down by resource type over a period of 24 weeks.

Minting Supply from Transient Resources. As discussed in the Introduction, historically, in order to maintain allocation speed, various processes had to rely on substantial amounts of readily available buffer resources, that were underutilized and unprioritized in the meantime. Since the Quota Marketplace allows clawbacks in a matter of minutes, this enabled

the addition of a large fraction of these buffer resources to Quota Marketplace, resulting in additional minted supply. We club multiple such buffer sources into an umbrella category named "serving buffer". Likewise, new capacity keeps arriving on a daily basis, even though the promised delivery date of these chips could be several days or weeks down the line. This is because fulfilment of a large consignment necessarily has to happen in batches, slowly building up to the deadline where the entire consignment is due. On the other hand, even if one were to allocate these chips that arrived ahead of time to the business units that are due to receive them later, the business units cannot be charged back ahead of their scheduled start date. Therefore, these soon-to-be assigned chips can neither be charged back for, nor be put to other high priority uses (given that clawbacks are time-consuming in traditional pools). Such temporarily available resources tend to be underutilized and unprioritized in traditional systems, but they can be readily utilized by the Quota Marketplace. We call the supply that the Quota Marketplace receives from such sources as the "holding pool". Figures 7 and 8 serve to illustrate the magnitude of bonus capacities unlocked by the Quota Marketplace from serving buffer and the holding pool. The bonus capacity unlocked by the Quota Marketplace consistently is in the order of hundreds of thousands of chips, reflecting the substantial volume of reclaimed resources integrated into the marketplace. Note that given the 8-year period over which demand has been growing tenfold annually, and correspondingly companies have been heavily investing in these chips, these holding pools are inevitably substantial. Thus, a crucial contribution of the Quota Marketplace is its ability to effectively harness large quantities of transiently available resources, improving overall system utilization.

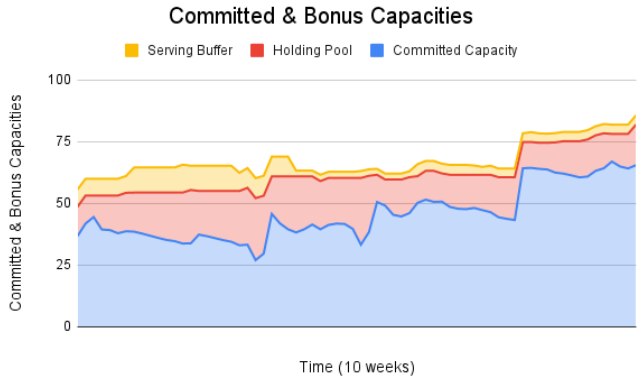


Figure 7: Over the period of 10 weeks, we see the bonus capacities that the Quota Marketplace has made available from different sources (y-axis is normalized). The total bonus is substantial, and remains consistently so.

Incentives in Action: Occupancy. The operational dynamics of the Quota Marketplace periodically offer empirical

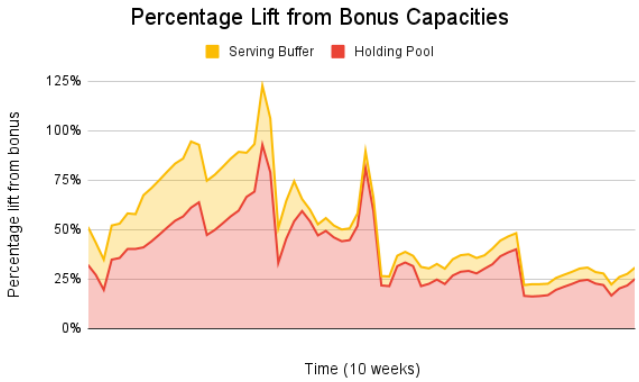


Figure 8: Another way to visualize Figure 7: over the same period of 10 weeks as in Figure 7, the percentage lift in capacities offered by the two different sources. The lowest point in this 10 weeks period is about 25%.

insights into user responses to its incentive structures. A key example arises during the regular deployment of new accelerator hardware batches to specific geographic locations or cells. By measuring the time lag between the introduction of this new supply and the corresponding rise in resource occupancy, we can quantify the market's efficiency in absorbing new capacity. We conducted a comparative analysis focusing on this phenomenon. As shown in Figures 9 and 10, two resource pools received simultaneous additions of new accelerators: one operating under the Quota Marketplace framework and a control pool managed via traditional static allocation. Within the Quota Marketplace pool, occupancy levels rapidly tracked the increase in available supply, while in the conventional pool, occupancy substantially lagged supply. Significantly, this swift uptake in the Quota Marketplace pool occurred organically, without requiring broad organizational announcements or specific notifications regarding the new hardware's availability or location. Users, guided predominantly by the price signals within the marketplace, efficiently discovered and migrated towards these newly-introduced and lower-priced resources.

Incentives in Action: Demand Shaping. The introduction of a new resource type, that is a partial substitute to a pre-existing resource type, provides another opportunity to obtain useful insights into user responses to incentives. Initially available at very low prices due to lack of demand, this new resource type requires users to adapt their workflows for compatibility. Despite the existence of this friction, the price of this new resource type rapidly increases to almost catch up with the pre-existing resource type. This price convergence occurs organically through user responses to price signals and peer communication via chat and email about the existence of a very low-priced alternative, rather than management directives or mandates. This behavior confirms that the Quota Market-

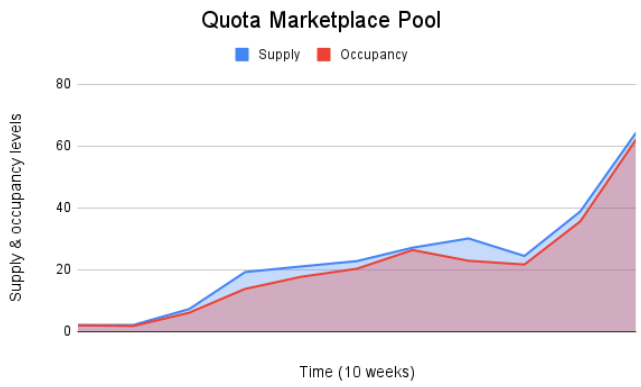


Figure 9: Over a 10-week period when a new resource type was introduced, we see how occupancy closely tracks supply, even during substantial supply spikes (y-axis is normalized).

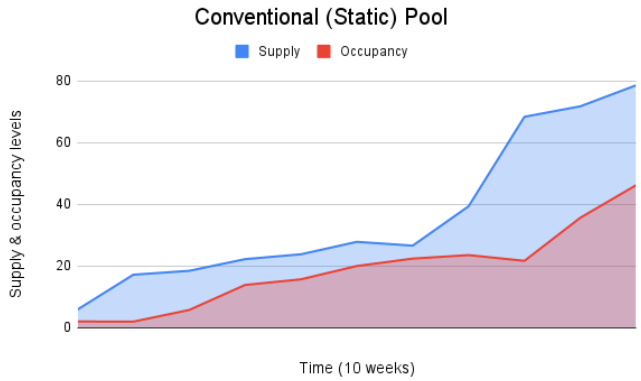


Figure 10: Over the same 10-week period as in Figure 9, when the same new resource type was introduced in the conventional static pool, we see how occupancy substantially lags behind supply (y-axis is normalized as in Figure 9).

place effectively shapes demand through pricing mechanisms, i.e., it indeed makes demand flow towards less-congested resource types.

Tackling Highly Non-trivial Tasks. Another important benefit of the QM and the flexibility that it provides is that it makes it feasible to tackle highly non-trivial large-scale tasks that would have previously required multiple quarters to complete. Such tasks can now be completed much faster (in a matter of weeks) due to the unprecedented acceleration that the Quota Marketplace is able to provide by reprioritizing portions of its massive fleet. Note that even such massive reprioritizations need not severely impact any particular business unit, or team, as their impact is seamlessly amortized across the whole market through price increases.

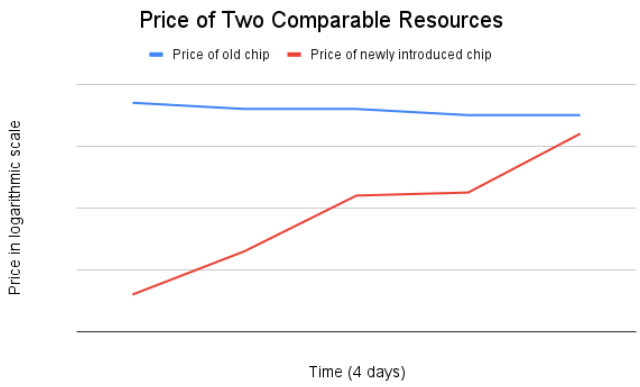


Figure 11: When a new resource type is introduced in the market, it starts at a low price, but the price grows exponentially (y-axis is in logarithmic scale) to catch up with that of a comparable pre-existing resource type.

4 Theoretical Analysis

We now formally prove that the market-based mechanism achieves efficiency and fairness guarantees, and we provide examples exhibiting how previously proposed solutions, like Karma, fail to achieve the same. We exhibit this separation in a simple model with a fixed shared supply of a single resource and ignore pool-specific supply. Due to space limitations, most of our proofs are deferred to Appendix A.

Model. A set N of n agents (teams) compete over s units of a single resource, and our goal is to design a fair and efficient mechanism that determines how these units are allocated over time. Specifically, given some time horizon of $T \geq 1$ rounds, the mechanism decides how many units each agent is allocated in each round $t \in [T]$ (where $[T] = \{1, 2, \ldots, T\}$). An allocation $x \in \mathbb{N}^{n \times T}$ determines the number of units $x_t[i]$ that each agent i will receive in each round $t \in [T]$ (for simplicity, we assume units are divisible, so $x_t[i] \in \mathbb{R}$). For an allocation to be feasible, it needs to allocate no more than s units in each round, i.e., $\sum_{i \in N} x_t[i] \leq s$ for all $t \in [T]$.

At each round t, each agent i demands $d_t[i] \geq 0$ units and has some value $v_t[i] \geq 0$ for each of these units. Therefore, i's value for receiving $x_t[i]$ units in round t, denoted $v_t[i](x_t[i])$, is equal to $\min\{x_t[i], d_t[i]\} \cdot v_t[i]$, and the overall value of i in allocation x is $v_i(x) = \sum_{t \in [T]} v_t[i](x_t[i])$. This strictly generalizes the class of valuations used in the Karma paper [26], which correspond to the special case where $v_t[i] = 1$ for all i and t, i.e., all the agent values are uniform over time. Another interesting intermediate class that we consider are bi-valued agent preferences, which are the mild generalization of uniform values where each agent's values can take one of two values, e.g., either HIGH or LOW.

Pareto Efficiency and Max-min Fairness. An allocation x is Pareto-dominated by allocation x' if $v_i(x') \geq v_i(x)$ for every agent i and $v_i(x') > v_i(x)$ for at least one agent i. An allocation is *Pareto efficient* if it is not Pareto-dominated by any other feasible allocation. If $w_i > 0$ captures the relative weight, or significance, of each agent $i \in N$ (often referred to as the agent's "fair share"), then the (weighted) *max-min fair* allocation assigns to each agent i a fraction $\frac{w_i}{\sum_{j \in N} w_j}$ of the $T \cdot s$ available units. We call an allocation x α-*fair* if the total number of units allocated to each agent i is at least

$$\alpha \cdot \frac{w_i}{\sum_{j \in N} w_j} \cdot T \cdot s.$$

Market-Based Mechanism. Without loss of generality, we assume agent weights are normalized so that $\sum_{i \in N} w_i = s$, and the market-based mechanism starts by allotting a budget of $T \cdot w_i$ credits to each agent i. Then, at the beginning of each round t, each agent i reports the number $d_t[i]$ of units they request, along with a limit price $\bar{p}_t[i]$ (the maximum amount they would be willing to pay per unit), and amount of budget $b_t[i]$ that they are willing to spend in that round. Given this information and some price p_t, the mechanism can compute the demand of each agent i at round t as

$$D_t[i](p_t) = \begin{cases} 0 & \text{if } p_t > \bar{p}_t[i] \\ \min\{b_t[i]/p_t,\ d_t[i]\} & \text{otherwise.} \end{cases}$$

The total demand at round t for a price p_t is then equal to $D_t^*(p_t) = \sum_{i \in N} D_t[i](p_t)$, and the market-clearing price is the maximum price that ensures all units are allocated, i.e., the price p_t that satisfies $D_t^*(p_t) \geq s$ and $D_t^*(p_t + \varepsilon) < s$ for any $\varepsilon > 0$. The mechanism computes this market clearing price and allocates to each agent i a total of $D_t[i](p_t)$ units (if $D_t^*(p_t) > s$, the requests that would remain at price $p_t + \varepsilon$ are satisfied first, and a tie-breaking rule is used for the rest). The mechanism then subtracts $p_t \cdot x_t[i]$ credits from the total budget of each bidder i and proceeds to the next round.

We make the common "large market" assumption that no single agent can significantly affect prices, so they behave as price-takers, and since these are very scarce resources, the number of units requested at each round t is at least s, i.e., $\sum_{i \in N} d_t[i] \geq s$. We also assume that agents can estimate their demands and the prices for future rounds, and they choose how much to spend in each round t accordingly.

Chip-Hour Mechanisms. To overcome the limitations of traditional approaches that allocate chips in a static fashion (i.e., each agent has a fixed number of chips in every round), Chip-Hour mechanisms allocate "hours" of chip access to the agents, and then provide them with some flexibility regarding when to request these hours. In each round t, each agent i requests $d_t[i]$ units and, based on what fraction of their chip-hours they have used, and the overall demand, the Chip-Hour mechanism decides how many chips to assign. A prominent example is the Karma system of Vuppalapati et al. [26], which strictly prioritizes agents who have used less of their chip-hours so far, aiming to optimize max-min fairness. Some aspects of this system are similar to the market-based mechanism: agents are bootstrapped with some credits and they also earn an income of such credits (in proportion to their chip-hours); they then spend these credits to gain access to resources. However, there is a crucial difference: the "price" of every resource in the Karma system is fixed over time, so spending one credit always buys the user access to the same amount of resource. As we show in this section, this limitation restricts the agents' ability to optimize their allocation and it can lead to unfair and inefficient outcomes.

4.1 Limitations of Chip-Hour Mechanisms

To exhibit some limitations of Chip-Hour mechanisms, we first consider the class of uniform valuations (i.e., $v_t[i] = 1$ for all i and t), which was the focus of Vuppalapati et al. [26] in their analysis of Karma. For uniform valuations, they were able to achieve significant improvements over prior work which assumed that agent requests are static over time (i.e., $d_t[i] = d_{t'}[i]$ for any $t, t' \in [T]$). Specifically, they showed that Karma is Pareto efficient for uniform valuations and achieves a "myopic" notion of fairness: at any round t, given the resource allocation choices made in previous rounds (which may be unfair but cannot be changed anymore), the allocation at round t aims to optimize max-min fairness.

We first exhibit that moving beyond this myopic notion of fairness and aiming for a global notion like max-min fairness is, in fact, impossible, not just for Karma, but for any Chip-Hour mechanism. This is in contrast to the market-based mechanism which leverages the agents foresight regarding their future demand to move beyond myopic fairness notions and achieve (approximate) max-min fairness even in settings with general heterogeneous valuations (Theorem 4).

Theorem 1. *Chip-Hour mechanisms cannot guarantee* α*-fairness for* $\alpha > 1/n$*, even for uniform-value instances.*

Proof. Consider a simple instance with n agents of equal significance $w_i = 1$, $s = n$ units, and $T = n$ rounds. Agent 1 requests $d_1[1] = s$ units in round 1 and $d_t[1] = 0$ for all subsequent rounds $t > 1$. All other agents request s units in every round. The max-min fair solution for this instance would allocate all the s units in the first round to agent 1 and split the units of the remaining rounds equally among the other agents, allocating $s = n$ units to every i. On the other hand, given this instance, and lacking any foresight regarding the drop in future demand from agent 1, any Chip-Hour mechanism would distribute the units in the first round equally among the agents (since all agents have equal w_i and, hence, equal chip-hours). This yields a single unit ($s/n = 1$) for agent 1, leading to an outcome that is only $1/n$-fair.

In comparison, consider the outcome of the market-based mechanism facing this instance. Since agent 1 does not need any units after the first round, she is willing to spend all of her s credits in the first round. This leads to unit price of at least 1 for the first round. If any other agent i also spent some of their budget on the first round, then at least one round $t > 1$ would have a unit price that is strictly less than 1 (this is because the budget spent on subsequent rounds would be strictly less than $(n-1)s$, while the number of available units in subsequent rounds is $(n-1)s$). Therefore, this would not be a market equilibrium since agent i would prefer to spend on round t rather than round 1. As a result, in the market equilibrium every agent receives an equal number of units, leading to the max-min allocation, i.e., 1-fairness. □

We now show that for the slightly more general class of bi-valued preferences, Chip-Hour mechanisms are unable to guarantee Pareto efficiency too. In fact, moving beyond uniform valuations introduces incentives for the agents to manipulate these mechanisms and strategically misreport their demands. Our next result shows that their Pareto efficiency is violated, regardless of whether the users behave truthfully or strategically. This is in contrast to the market-based mechanism which achieves Pareto efficiency even for the general class of heterogeneous valuations (Theorem 3).

Theorem 2. *Chip-Hour mechanisms violate Pareto efficiency even for bi-valued instances, irrespective of whether the agents behave truthfully or strategically.*

4.2 Efficiency & Fairness of Markets

In contrast to Chip-Hour mechanisms, the market mechanism always guarantees Pareto efficiency, even for instances where the agents have general heterogeneous values.

Theorem 3 (Pareto Efficiency)**.** *The market mechanism achieves Pareto efficiency for general heterogeneous valuations, irrespective of how much the prices may vary.*

Apart from Pareto efficiency, we also show that the allocations induced by the market mechanism are (approximately) fair. Specifically, one of the things that we observed though the deployment of the market mechanism is that although the prices of the units can vary significantly over time, the average price that each agent pays per unit in the long run does not vary as much. One way to interpret this is that there is no reason why any one agent's requests may be strongly correlated with higher prices. If we let $\tilde{p}$ denote the average price per unit across the T rounds and let $f_i \cdot \tilde{p}$ be the average price paid by agent i over this time horizon, then we can bound the fairness of the induced allocation using the f_i values. Specifically, the resulting allocation is $\min_{i \in N} \frac{1}{f_i}$-fair, converging to max-min fairness as the f_i values converge to 1. Note that this is in stark contrast to Chip-Hour mechanisms which cannot even achieve better than $1/n$-fairness when $f_i = 1$ (see the instance used in the proof of Theorem 1).

Theorem 4 (Fairness)**.** *Let $\tilde{p}$ be the average price of a resource over time. If the average price paid by some agent i is at most $f_i \cdot \tilde{p}$, then the resulting allocation is $\min_{i \in N} \frac{1}{f_i}$-fair, even for general heterogeneous valuations.*

5 Discussions and Future Directions

While the system is deployed in Google to serve thousands of users and manage resources at scale, there remain significant improvements to be explored in maximizing efficiency, flexibility, and user experience. We outline here some unresolved challenges, and opportunities being currently explored.

Shaping User Behavior. Users were shown to respond to market prices which opens the opportunity to use prices as a vehicle for driving other organizational goals. One example is carbon-efficiency: prices can be adjusted to incentivize scheduling in more carbon-efficient datacenters or use more carbon-efficient configurations. Similarly, the market can be changed to provide discounts (or penalties) guiding towards desired behavior. For example, workloads that are well configured to make good use of the hardware, don't crash, save checkpoints can be rewarded. These mechanisms can also be used to drive adoption for workload improvements that require additional user effort or opt-in.

Currently, our system simply balances supply and demand without taking many other factors into account, but the existence of prices creates the possibility of shaping the behavior of users using economics rather than mandates.

Reducing Failure Blast Radius. Centralized resource allocation with the QM improves efficiency but also creates a single point of failure. One way to reduce the impact of a bad rollout or bug in the code is to shard the system into multiple deployments, each managing a subsection of the fleet. There are many ways to consider sharding which offer tradeoffs in efficiency and user experience. One approach is to thin out the necessary global component while transferring as much logic as possible to sharded replicas. We can also imagine separating parts the system by location or resource type.

Potential for Gaming. Our system operates within Google, where users and teams prioritize their own resource needs but are policy constrained to not intentionally disrupt others. We now briefly discuss a few potential gaming strategies in general adversarial environments. Appendix B discusses their removal at the cost of increased complexity for users.

Gaming: Currency Conversion. When designing QM, we made the conscious decision of letting each business unit mint their own currency. All the business-unit-local credits get converted to a global currency when the market clears. The local-to-global currency exchange rate is a function of

both the number of credits minted by that business unit, and the market weight assigned to that business unit by the company. This freedom to mint credits has ergonomic benefits: for example, credits minted per hour, can be maintained in proportion to the number of engineers in the team. However, this freedom also comes with an ability to game the system. Consider a scenario where a team mints a lot of credits per hour for a while, accumulates a substantial number of credits in its savings, and then dramatically reduces the rate at which it mints credits per hour. The newly minted credits are worth much more because the currency's value had significantly risen. But if the system were to value the accumulated old credits also at the same level as the new credits, the team would have successfully managed to mint credits out of thin air. If we value these credits differently, we introduce complexity. We discuss a few possible solutions and the tradeoffs they present in appendix B, and a good middle-ground.

Gaming: Adversarially Depleting Other Teams' Credits. Teams bid for demanded resources, but are only charged credits for jobs that run. A bad actor could launch a job which is unlikely to schedule and is set to immediately fail if it does. For example, this job may require all the resources of a specific type in a specific location to start. The bad actor is thus able to inflate prices for all other market participants without paying any credits to do so. We discuss a few solution approaches and the tradeoffs they present in appendix B.

6 Related Work

Systems literature on resource allocation mechanisms is rich and vast. Here, we focus on a few related streams of work. To the best of our knowledge, ours is the first work that (a) guarantees Pareto efficiency and (approximate) max-min fairness in the presence of dynamic demand with heterogeneous values, and (b) has designed, implemented, and deployed a market-based mechanism for system resources at this scale.

Market-based mechanisms for pricing computer resources is an old idea [4, 6, 23]. However, pricing for shared resource allocation was not a first-order concern in time-shared systems, leading to this research being short-lived. There have been other works that took a market approach to resource allocation [24, 27]. Two decades ago, when [21] described why the time was right to revisit market-based solution for systems resource allocation, they cite the demand-supply gap as the primary reason. This has never been any more true than it is today. Therefore, research on market-based ideas for resource allocation is indeed timely. Documented experiences of markets allocating resources in a wide variety of settings are valuable, as they serve as catalysts for wide-spread adoption of this powerful tool. With this motivation, we go over how we addressed some of the markets-systems integration challenges that were raised in [21].

Allocation Policy Must be Explicit. The main challenge identified is that of landing on an allocation policy that all the users can agree on: whether to maximize allocation efficiency, or to favor jobs from under-represented users etc. In the environment we operate in (a large company) all players are in agreement that they want to maximize the value to the company. Disagreements about what is more valuable can be resolved top-down following the organizational structure – which mirrors the way credits are distributed.

Calculating and Expressing Valuation. Despite the significant influence of bidding interfaces on user perception of market mechanisms, this area remains under-researched. Our system allows users to express workload value via parameters specifying job credits. While finer-grained controls are possible, the system must balance expressivity with interface simplicity, as Shneidman et al. [21] note that ease of use often surpasses the importance of complex features.

Well-Defined Currency. Shneidman et al. [21] also note that virtual currencies have a lot of issues, particularly that they are not exchangeable outside the system. While our credits too cannot be used for buying anything outside the system, we note that business units (not individual teams) are charged back for the market power they are assigned. I.e., business units pay in real dollars for getting credits, and indeed they could have used those dollars towards headcount, other equipment etc. Therefore, credits in QM have a tangible dollar value. Designing a good currency also includes creating mechanisms to control inflation and volatility in prices. QM has guardrails like bid upper bounds to ensure that teams are unable to hoard credits and flood the market with credits.

Pricing-Based Resource Allocation. Spot-instance marketplace and virtual machine auctions [3, 7, 9, 29, 31, 32] are examples of pricing-based resource allocation. Nevertheless, fair resource allocation is not their goal, and even in rare cases where fairness was the goal [28], it is qualitatively different from the long-term fairness that we focus on [26]. Beyond resource allocation, other places where credits have been used in game-theoretic contexts include [5, 8, 18, 30].

Max-min Fairness and Generalizations. Being able to satisfy Pareto-efficiency and fairness has inspired a lot of work on this topic [10, 12–16, 19, 20, 22], including generalizations to multiple resources [10–12]. As described earlier, the efficiency guarantees of these mechanisms crucially rely on the demand being static; the recent Karma mechanism addresses this exact problem [26]. In this paper, we further this line of work by including heterogeneous values for demand.

Acknowledgments

We sincerely thank the reviewers for their insightful feedback, and the shepherd for their support with the revision process.

References

[1] Amin Vahdat. Google Cloud Next 25 Opening Keynote: The New Way to Cloud. https://youtu.be/Md4Fs-Zc3tg, 2025.

[2] Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. Workload Analysis of a Large-Scale Key-Value Store. In *SIGMETRICS*, 2012.

[3] Orna Agmon Ben-Yehuda, Eyal Posener, Muli Ben-Yehuda, Assaf Schuster, and Ahuva Mu'alem. Ginseng: Market-Driven Memory Allocation. In *VEE*, 2014.

[4] F. J. Corbató and V. A. Vyssotsky. Introduction and overview of the multics system. In *Proceedings of the November 30–December 1, 1965, Fall Joint Computer Conference, Part I*, page 185–196, 1965.

[5] Landon P Cox and Brian D Noble. Samsara: Honor Among Thieves in Peer-to-Peer Storage. *Operating Systems Review*, 2003.

[6] David S. Diamond and Lee L. Selwyn. Considerations for computer utility pricing policies. In *Proceedings of the 1968 23rd ACM national conference*, pages 189–200. ACM Press, 1968.

[7] Amazon EC2. Spot Instances. https://aws.amazon.com/ec2/spot/.

[8] Joan Feigenbaum and Scott Shenker. Distributed Algorithmic Mechanism Design: Recent Results and Future Directions. In *Current Trends in Theoretical Computer Science: The Challenge of the New Century Vol 1: Algorithms and Complexity Vol 2: Formal Models and Semantics*. 2004.

[9] Liran Funaro, Orna Agmon Ben-Yehuda, and Assaf Schuster. Ginseng: Market-Driven LLC Allocation. In *ATC*, 2016.

[10] Ali Ghodsi, Matei Zaharia, Benjamin Hindman, Andy Konwinski, Scott Shenker, and Ion Stoica. Dominant resource fairness: fair allocation of multiple resource types. In *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation*, NSDI'11, page 323–336, USA, 2011. USENIX Association.

[11] Robert Grandl, Ganesh Ananthanarayanan, Srikanth Kandula, Sriram Rao, and Aditya Akella. Multi-Resource Packing for Cluster Schedulers. In *SIGCOMM*, 2014.

[12] Robert Grandl, Mosharaf Chowdhury, Aditya Akella, and Ganesh Ananthanarayanan. Altruistic Scheduling in Multi-Resource Clusters. In *OSDI*, 2016.

[13] Chi-Yao Hong, Srikanth Kandula, Ratul Mahajan, Ming Zhang, Vijay Gill, Mohan Nanduri, and Roger Wattenhofer. Achieving High Utilization with Software-Driven WAN. In *SIGCOMM*, 2013.

[14] Sushant Jain, Alok Kumar, Subhasree Mandal, Joon Ong, Leon Poutievski, Arjun Singh, Subbaiah Venkata, Jim Wanderer, Junlan Zhou, and Min Zhu et al. B4: Experience with a Globally-Deployed Software Defined WAN. In *SIGCOMM*, 2013.

[15] Deepak Narayanan, Keshav Santhanam, Fiodar Kazhamiaka, Amar Phanishayee, and Matei Zaharia. Heterogeneity-Aware Cluster Scheduling Policies for Deep Learning Workloads. In *OSDI*, 2020.

[16] Deepak Narayanan, Fiodar Kazhamiaka, Firas Abuzaid, Peter Kraft, Akshay Agrawal, Srikanth Kandula, Stephen Boyd, and Matei Zaharia. Solving Large-Scale Granular Resource Allocation Problems Efficiently with POP. In *SOSP*, 2021.

[17] Renato Paes Leme, Martin Pal, and Sergei Vassilvitskii. A field guide to personalized reserve prices. In *Proceedings of the 25th international conference on world wide web*, pages 1093–1102, 2016.

[18] Michael Piatek, Tomas Isdal, Thomas Anderson, Arvind Krishnamurthy, and Arun Venkataramani. Do Incentives Build Robustness in BitTorrent. In *NSDI*, 2007.

[19] Lucian Popa, Gautam Kumar, Mosharaf Chowdhury, Arvind Krishnamurthy, Sylvia Ratnasamy, and Ion Stoica. FairCloud: Sharing the Network in Cloud Computing. In *SIGCOMM*, 2012.

[20] Qifan Pu, Haoyuan Li, Matei Zaharia, Ali Ghodsi, and Ion Stoica. FairRide: Near-Optimal, Fair Cache Sharing. In *NSDI*, 2016.

[21] Jeffrey Shneidman, Chaki Ng, David C. Parkes, Alvin AuYoung, Alex C. Snoeren, Amin Vahdat, and Brent N. Chun. Why markets could (but don't currently) solve resource allocation problems in systems. In *Proceedings of HotOS'05: 10th Workshop on Hot Topics in Operating Systems, June 12-15, 2005, Santa Fe, New Mexico, USA*. USENIX Association, 2005. URL http://www.usenix.org/events/hotos05/final_papers/full_papers/shneidman/shneidman.pdf.

[22] David Shue, Michael J Freedman, and Anees Shaikh. Performance Isolation and Fairness for Multi-tenant Cloud Storage. In *OSDI*, 2012.

[23] Ivan E. Sutherland. A futures market in computer time. *Commun. ACM*, 11(6):449–451, 1968.

[24] Andrew S. Tanenbaum, Sape J. Mullender, and Robbert van Renesse. Using sparse capabilities in a distributed operating system. In *Proceedings of the 6th International Conference on Distributed Computing Systems*, pages 558–563, 1986.

[25] Midhul Vuppalapati, Justin Miron, Rachit Agarwal, Dan Truong, Ashish Motivala, and Thierry Cruanes. Building an Elastic Query Engine on Disaggregated Storage. In *NSDI*, 2020.

[26] Midhul Vuppalapati, Giannis Fikioris, Rachit Agarwal, Asaf Cidon, Anurag Khandelwal, and Éva Tardos. Karma: Resource allocation for dynamic demands. In Roxana Geambasu and Ed Nightingale, editors, *17th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2023, Boston, MA, USA, July 10-12, 2023*, pages 645–662. USENIX Association, 2023.

[27] Carl A. Waldspurger, Tad Hogg, Bernardo A. Huberman, Jeffrey O. Kephart, and Scott Stornetta. SPAWN: A distributed computational economy. *IEEE Transactions on Software Engineering*, 18(2):103–117, February 1992.

[28] Xiaodong Wang and José F Martínez. XChange: A Market-Based Approach to Scalable Dynamic Multi-Resource Allocation in Multicore Architectures. In *HPCA*, 2015.

[29] Rich Wolski, John Brevik, Ryan Chard, and Kyle Chard. Probabilistic Guarantees of Execution Duration for Amazon Spot Instances. In *SC*, 2017.

[30] Gala Yadgar, Michael Factor, and Assaf Schuster. Cooperative Caching with Return on Investment. In *MSST*, 2013.

[31] Liang Zheng, Carlee Joe-Wong, Chee Wei Tan, Mung Chiang, and Xinyu Wang. How to Bid the Cloud. In *SIGCOMM*, 2015.

[32] Liang Zheng, Carlee Joe-Wong, Christopher G Brinton, Chee Wei Tan, Sangtae Ha, and Mung Chiang. On the Viability of a Cloud Virtual Service Provider. In *SIGMETRICS*, 2016.

A Proofs omitted from Section 4

In this section we provide all the proofs that were omitted from the main body of the paper due to space limitations.

Theorem 2. *Chip-Hour mechanisms violate Pareto efficiency even for bi-valued instances, irrespective of whether the agents behave truthfully or strategically.*

Proof. Note that Chip-Hour mechanisms ask each agent i to report their $d_t[i]$ demand. We first consider the case where the agents truthfully report this value in each round, and we then also consider the case where the agents may strategically misreport this value (e.g., by reducing the $d_t[i]$ in rounds where their value is LOW, aiming to save their credits for subsequent rounds of HIGH value). For this proof we normalize the HIGH value to 1 and the LOW value to some $\varepsilon \in (0,1)$.

Truthful Reporting. To verify the lack of Pareto efficiency for the case where the agents truthfully report their demand, $d_t[i]$, consider the following simple instance involving just two agents and two rounds. In this instance, we have $d_t[i] = 2$ for both agents and rounds, but the values of agent 1 are 1 for the first round and some arbitrarily small ε for the second, while the values of agent 2 are ε for the first round and 1 for the second. In other words, agent 1 strongly prefers the units of the first round while agent 2 strongly prefers those of the second round. However, since Chip-Hour mechanisms do not elicit the agents' values (implicitly assuming that they are uniform over time), the two agents look identical, and the outcome would equally split the two units of each round among the two agents, leading to an overall value of $(1+\varepsilon)$ for each. To verify that this is not Pareto efficient, note that if we were to give agent 1 both of the first-round units and give agent 2 the second-round units, then both agents would have a value of 2 instead, leading to an almost 100% increase in both values.

Strategic Reporting. It is natural to assume that an agent facing a mechanism that disregards the intensity of their demand (as in the instance above), the agent may choose to strategically misreport their $d_t[i]$ requests, aiming to maximize their value (e.g., by submitting no requests during rounds of lower value). We now show that even if agents report strategically aiming to optimize their allocation with respect to their underlying values, Chip-Hour mechanisms still fail to achieve Pareto efficiency. We prove this using small instances with two agents, two units, and two rounds. In all instances that we consider, both agents request both units of the second round, so the Chip-Hour mechanism's decision regarding how to allocate those two units is purely a function of the units they were allocated in the previous round (or, equivalently, a function of their remaining budget). We prove that irrespective of how the mechanism uses this information, there exists an instance for which the resulting outcome will be Pareto inefficient.

First, we focus on Chip-Hour mechanisms which prioritize the agents who have used less of their credits, like Karma does. Specifically, for our two-round instance, we consider mechanisms that would allocate both of the second round units to an agent who was not allocated any units in the first round if the other agent was allocated at least one. For this class of Chip-Hour mechanisms, we consider the instance where agent 1 values the first round units $\varepsilon < 0.5$ and both agents value the second round units 1. In this case, although agent 1 has a positive value for the two first-round units, he has a higher value for a second round unit, so they would prefer to use no credits in the first round, as this would lose them one (more valuable) unit in the second round. Therefore, the strategic choice of agent 1 would be to report a demand $d_1[1] = 0$ for the first round and then split the two second-round units with agent 2. This leads to a value of 1 for each agent. However, note that this is not Pareto efficient since the two first-round units remain unallocated even though agent 1 has positive value for them. If we were to instead allocate those first-round units to agent 1, this would yield a value of $1+2\varepsilon$ for this agent (an up to 100% increase), which Pareto dominates the previous allocation.

A similar argument can prove the Pareto inefficiency of the (much less natural) class of Chip-Hour mechanisms that instead prioritize agents who have spent more of their credits. In this case, facing the same instance as above, agent 1 would report their true demand, but agent 2 would prefer to lie by requesting the units in the first round (although he has no value for them). This way, he would be allocated one of them (by symmetry) and he would ensure that he is also allocated one unit in the second round (by symmetry). However, this is clearly not Pareto efficient since one of the first-round units that agent 1 has positive value for is allocated to agent 2 who has no value for it.

Finally, we conclude with the class of Chip-Hour mechanisms that would split the second round units equally among the agents irrespective of what happened in the previous round (i.e., they are independent of how many credits each agent has left). For these mechanisms, we consider the same instance that we used for the truthful reporting case above. In this case, no agent has any incentive to misreport (since the mechanism essentially disregards history), so the Pareto inefficient outcome that arises is the same as the one in the truthful case. □

Theorem 3 (Pareto Efficiency)**.** *The market mechanism achieves Pareto efficiency for general heterogeneous valuations, irrespective of how much the prices may vary.*

Proof. Given some instance, let p be the market clearing prices computed by the market mechanism and x be the corresponding allocation that it yields. Assume that the allocation x returned by the market mechanism is Pareto dominated by some other allocation x', i.e., there exists some agent i who strictly prefers their allocation in x' over the one in x. This must mean that the total price, according to p, of the units that i is allocated in x' must be greater than the units allocated to i in x, i.e., $\sum_{t\in[T]} x'_t[i]p_t > \sum_{t\in[T]} x_t[i]p_t$, otherwise i could just buy them and strictly increase their value (this is due to the common assumption that i spends their budget optimally given the price vector p). Also, since x' Pareto dominates x, it must be the case that every other agent $j \neq i$ is at least as happy in x'. Similarly, this implies that $\sum_{t\in[T]} x'_t[j]p_t \geq \sum_{t\in[T]} x_t[j]p_t$, otherwise j must be spending their budget sub-optimally in x; specifically, they could instead buy their x' units by spending less than their budget and then spend the remaining budget on some additional units and strictly increase their value. This uses the common assumption that no agent receives everything that they have positive value for, so there are always additional (fractions of) units that they can buy. Summing the aforementioned inequalities over all agents k (both i and $j \neq i$), we get

$$\sum_{k\in N}\sum_{t\in[T]} x'_t[k]p_t > \sum_{k\in N}\sum_{t\in[T]} x_t[k]p_t$$
$$\sum_{t\in[T]}\sum_{k\in N} x'_t[k]p_t > \sum_{t\in[T]}\sum_{k\in N} x_t[k]p_t$$
$$\sum_{t\in[T]} s\cdot p_t > \sum_{t\in[T]} s\cdot p_t,$$

which is a contradiction. This result can also be seen as an implication of the First Welfare Theorem. □

Theorem 4 (Fairness)**.** *Let $\tilde{p}$ be the average price of a resource over time. If the average price paid by some agent i is at most $f_i \cdot \tilde{p}$, then the resulting allocation is* $\min_{i\in N} \frac{1}{f_i}$*-fair, even for general heterogeneous valuations.*

Proof. Since each agent i is allotted a budget of $T \cdot w_i$ credits, the total number of allotted credits is $T \cdot s$. Also, the total number of units allocated per round is s, so $T \cdot s$ units allocated overall. Therefore, the average price per unit is $\tilde{p} = 1$, and the average price paid by any agent i is at most f_i. Also, since agent i has a total budget of $w_i \cdot T$, this means that the number of units that they were able to buy using this budget is at least $\frac{w_i T}{f_i}$ overall, which is a $\frac{w_i}{s\cdot f_i}$ fraction of all available units. This implies that every agent receives at least a $\min_{i\in N} \frac{1}{f_i} \cdot \frac{w_i}{\sum_{i\in N} w_i}$ fraction of the $T \cdot s$ available units, so the resulting allocation is $\min_{i\in N} \frac{1}{f_i}$ -fair. □

B Gaming Strategies

Gaming: Currency Conversion. When designing QM, we made the conscious decision of letting each business unit mint their own currency. All the business-unit-local credits get converted to a global currency when the market clears. The local-to-global currency exchange rate is a function of both the number of credits minted by that business unit, and the market weight assigned to that business unit by the company. This freedom to mint credits has ergonomic benefits: for example, credits minted per hour, can be maintained in proportion to the number of engineers in the team. However, this freedom also comes with an ability to game the system. Consider a scenario where a team mints a lot of credits per hour for a while, accumulates a substantial number of credits in its savings, and then dramatically reduces the rate at which it mints credits per hour. The newly minted credits are worth much more because the currency's value had significantly risen. But if the system were to value the accumulated old credits also at the same level as the new credits, the team would have successfully managed to mint credits out of thin air. If we value these credits differently, we introduce complexity.

An inelegant solution for this problem is to declare that each credit comes with the exchange rate at which it was minted, and therefore different credits are worth differently to a team. This creates too much complexity and the meaning of a credit is lost when different credits are worth differently. Another solution with jarring optics is to say that the bank accounts of teams will get depleted by a certain amount (that depends on how much the business unit decided to raise its currency's value), thereby maintaining the same value for all the credits, old and new. But a team's credits vanishing for no choice of theirs (income rate decisions are made by the head of the business unit) is a recipe for many escalations and confusions. Designing good mechanisms that balance simplicity+usability with robustness to manipulations is a useful problem to solve. We consider this as a UX problem and indeed as [21] notes, the interface has the biggest influence on user perception.

Gaming: Adversarially Depleting Other Teams' Credits. Teams bid for demanded resources, but are only charged credits for jobs that run. A bad actor could launch a job which is unlikely to schedule and is set to immediately fail if it does. For example, this job may require all the resources of a specific type in a specific location to start. The bad actor is thus able to inflate prices for all other market participants without paying any credits to do so.

One approach to disincentivize this behavior is to charge for resource allocations, not usage. However, potentially conflicting scheduling outcomes later in the stack can lead to significant charges for teams that did not get to run their workloads. This becomes less probable as the market becomes more deeply integrated with the scheduler.

Another approach is to use the Vickrey-Clarke-Groves (VCG) mechanism. This would enable the system to price each workload's resources based on the externality it imposes. In other words, each workload's cost would be the difference between the value of the other workloads that would have run in this workload's absence and the value of the other workloads that ran in this workload's presence. A small job which has no measurable effect on the system would be more or less free. A large or hard-to-schedule job, like the one described above, would not be directly displaced by any single workload, so it would not affect their prices. But VCG has its own complications, like having to price each workload differently, as opposed to uniformly pricing every workload that ran at any given point of time at a given location or cell. I.e., VCG's pricing is complicated for users to understand, as opposed to the simple price-per-cell concept that the market guarantees.

ARTIFACT EVALUATED
USENIX
AVAILABLE

Bodega: Localized Linearizable Reads at Anywhere Anytime via Roster Leases

Guanzhou Hu*
Amazon Web Services
Seattle, WA, USA
josehgz@amazon.com

Andrea C. Arpaci-Dusseau
University of Wisconsin–Madison
Madison, WI, USA
dusseau@cs.wisc.edu

Remzi H. Arpaci-Dusseau
University of Wisconsin–Madison
Madison, WI, USA
remzi@cs.wisc.edu

Abstract

We present Bodega, the first consensus protocol that serves linearizable reads locally from any desired node, regardless of interfering writes. Bodega attains this capability via a new notion of cluster metadata called the *roster*, which is a generalization of leadership; it tracks arbitrary subsets of replicas as *responder* nodes for local reads. A consistent agreement on the roster is established through *roster leases*, an all-to-all leasing mechanism that generalizes existing all-to-one leasing approaches (Leader Leases, Quorum Leases), unlocking a new point in the protocol design space. Bodega further employs *optimistic holding* and *early accept notifications* optimizations to minimize interruption from interfering writes, and *smart roster coverage* and *lightweight heartbeats* to maximize practicality. Bodega is a non-intrusive extension to classic consensus; it imposes no special requirements on writes other than a responder-covering quorum.

We implement Bodega and related works in Summerset, a protocol-generic replicated key-value store written in async Rust. We compare it to previous protocols (Leader Leases, EPaxos, PQR, and Quorum Leases) and two production coordination services (etcd and ZooKeeper). Bodega speeds up average client read requests by 5.6x~13.1x on real WAN clusters versus previous approaches under moderate write interference, and closely matches the performance of sequentially-consistent etcd and ZooKeeper deployments across YCSB workloads. Bodega supports fast proactive roster changes and delivers on-par write performance. Our Summerset codebase is open-sourced at https://github.com/josehu07/summerset/tree/bodega-artifact.

ACM Reference Format:
Guanzhou Hu, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2026. Bodega: Localized Linearizable Reads at Anywhere Anytime via Roster Leases. In *Proceedings of the 20th USENIX Symposium on Operating Systems Design and Implementation (OSDI '26).* ACM, Seattle, WA, USA, 39 pages. https://doi.org/xxxxxxx.xxxxxxx

1 Introduction

Paxos-based consensus [60, 61, 65] (and protocols alike [85, 87]) serves as a critical foundation for modern distributed systems infrastructure. Originally used in limited contexts, e.g., for bespoke configuration information as in Petal [67], for critical metadata stores such as etcd [35, 97] and FireScroll [36, 95], and for lock services such as Chubby [21], consensus now forms the foundation of widely used cloud-native databases such as Spanner [30], CockroachDB [107], TiDB [50], ScyllaDB [100], and Physalia [19].

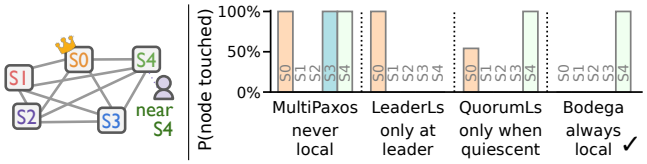


Figure 1. Frequency of touching a node on the critical path of reads *by a client near S4, in a cluster where S0 is the leader, with infrequent writes. Ideal outcome is 100% at S4.*

In these systems, simple access semantics are critical, enabling scalable services to be readily built atop them [4, 15]. In particular, *linearizability* is a strong consistency level they strive to provide: for interrelated requests, clients observe a real-time serial ordering, as if talking to a single node [47, 49].

Local Linearizable Reads. Delivering high performance in linearizable systems remains a daunting challenge. In the cloud era, systems replicate critical data across multiple geographically-distinct availability zones [7, 28, 79], to guard against correlated failures caused by power outage, fire, natural disaster, or operator error [22, 24, 43, 108]. By spreading replicas globally, robustness is achieved, but at the cost of performance due to quorum round trips [66].

The physical distribution of replicas yields an opportunity to serve read requests locally from a client's nearest replica. Reads comprise a majority of the workloads [8, 29, 88, 91]; achieving local reads can greatly reduce read latency and drastically increase overall throughput.

Existing Solutions Fall Short. Existing consensus protocols have demonstrated effective wide-area optimizations, but none, to our knowledge, supports coherently fast linearizable reads for workloads containing even small amounts of interfering writes. Leaderless protocols [27, 56, 62, 76, 80, 98] allow near quorums but not local reads. Others explore flexible quorums [42, 46, 48, 64, 106], utilize special hardware or client validation [5, 32, 40, 69, 93, 101, 112], or exploit API semantics favoring writes [37–39, 74, 86, 90, 96, 114]. Read leases (covered later) [10, 18, 25, 26, 41, 81, 82] are so far the most compelling, but are only effective at the leader or during quiescent periods without interfering writes.

*Work was done when at UW–Madison.

Self-Containment Necessitates Leases. A primary challenge of designing consensus protocols is self-containment: they cannot assume an external oracle. To enable local linearizable reads, the protocol must judge whether replica-local data is the most recent; contacting an external service to obtain this information would defeat the initial purpose. Moreover, having external dependencies would reduce the system's fault tolerance guarantees to those of the external services [57, 95, 107].

Within the design space of self-contained protocols, *leases* are a vital and powerful primitive. They carry timed promises that naturally tolerate faults through expiration [41], while only requiring bounded clock drifts (typical in today's cloud environments [40, 53, 68, 77]). This opens the gateway to local linearizable read protocols.

Leases Were Not Fully Exploited. Existing lease-infused protocols, however, do not employ the most suitable types of promises for local reads, and thus cannot fully realize their potential. As a motivating example, Figure 1 shows a 5-site cluster where S0 is the leader (and S4 is a local-read-enabled replica, if eligible), and reports the frequency of servers being touched by read requests from a client near S4. The workload contains 99% reads and only 1% writes, which favors existing approaches. Classic consensus (MultiPaxos) requires a majority accept quorum around S0. Leader Leases only protect stable leadership, so S0 can reply to reads directly, yet the delay between client and S0 persists. Quorum Leases allow granting leases to followers but use leases to guard against individual writes, rendering them vulnerable to even small amounts of concurrent writes to the key. As a result, a significant portion are redirected to the leader S0.

Our Approach: Roster Leases. We introduce the notion of a *roster*: a generalized form of cluster metadata that dictates not only leadership but also an assignment of local-read-enabled replicas (called *responders*) for arbitrary keys. Accordingly, we introduce *roster leases*, a novel all-to-all generalization of leader leases, deployed off the critical path to protect the agreement on the roster with no observable overhead. Roster leases stay valid in the absence of failures or proactive changes.

We present Bodega, a consensus protocol that uses roster leases to empower local linearizable reads. Bodega assures that writes never commit before reaching all of the key's active responders. A responder that holds a majority of leases can thus serve reads directly from its local state, if it knows the latest value will commit. When unsure, the responder *optimistically holds* the read locally until enough information is gathered, optionally utilizing *early accept notifications* to accelerate the hold. The roster may be changed manually by users, automatically according to runtime statistics, or in reaction to failures.

Our evaluation shows that Bodega speeds up client read requests by 5.6x~13.1x versus previous approaches under slight write interference, delivers comparable write performance, supports proactive roster changes in two message rounds as well as self-contained fault tolerance via leases, and matches the performance of sequentially-consistent etcd [35] and ZooKeeper [52] deployments across all YCSB variants.

Summary of Contributions. ❶ We introduce the notion of roster and propose Bodega, a consensus protocol equipped with a novel all-to-all roster leases algorithm, enabling local linearizable reads from any node at any time. ❷ We provide a thorough qualitative comparison across wide-area linearizable read approaches. ❸ We implement Bodega and related protocols in Summerset, a protocol-generic replicated key-value store, with 25.6k lines of async Rust. ❹ We evaluate Bodega comprehensively against previous works and two production coordination services, etcd and ZooKeeper, on 5-site wide-area clusters, delivering aforementioned evaluation results. ❺ We provide a formal TLA$^+$ specification of the full algorithm in the appendix.

The rest of the paper is organized as follows: §2 provides background and discusses existing solutions; §3 presents the Bodega design; §4 gives a formal comparison and proof; §5 covers the implementation of Bodega in Summerset; §6 presents our evaluation; §7-8 add discussions and related work; §9 concludes.

2 Background and Motivation

We provide background on consensus and linearizable reads, discuss existing solutions, and derive our goals for Bodega.

2.1 Consensus & Linearizable Reads

We consider the typical consensus problem of reaching agreement across message-passing server nodes, where nodes can be fail-slow/stop and the network is asynchronous [60].

Following well-established practice, nodes agree upon an ordered *log* of commands to behave as a replicated state machine (RSM) [61, 99]. For clarity, we use key-value `Put`/`Get` commands. In practice, `Get`s map to read-only requests that only query but do not update state (hereby referred to as *reads*), and `Put`s map to all other requests (*writes*). We restrict our discussion to non-transactional commands [17]; these are out of the scope of this paper.

Linearizable Reads. The consistency level dictates what results are allowed to be observed by clients [49]. *Linearizability* is the strongest non-transactional consistency level, where clients expect a serial ordering of commands with the real-time property: a command on key k issued at physical time t must be ordered after all the interfering writes on k acknowledged before t, and observe its latest value [6, 47, 80]. This semantic is mandatory for critical use cases such as metadata storage and coordination [19, 21, 35, 36], and is generally desirable as clients would otherwise get stale reads from the past (*sequential consistency*) [6, 59] or weaker guarantees [102, 109], complicating development. Linearizable reads are the primary focus of this paper.

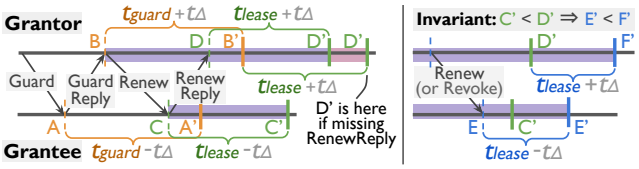


Figure 2. Demonstration of standard leasing. *Left: the guard phase helps establish the first iteration of promise coverage safely; grantee welcomes the first* `Renew` *only if it is received within the guarded period (C < A'). This allows the grantor to derive a safe expiration D' = B'+t_{lease}+t_Δ even if the* `RenewReply` *is lost, such that C' < D'. Right: grantor attempts to extend the promise with a* `Renew` *(or to actively revoke it with a* `Revoke`*), but has not received the grantee's reply. The leasing logic assures that E' < F'; therefore, if the grantee indeed failed, after F' the grantor can safely assert lease expiration. The purple-colored ranges depict when the lease is considered granted/held by the corresponding party.*

Availability Requirements. A practical protocol must also offer *availability* for fault tolerance, allowing client requests to proceed under any minority number of node/network faults, and retaining consistency in all circumstances [49, 61].

2.2 Distributed Lease

Leases are a common distributed system technique [41]. They may be deployed as user-facing APIs through locks [21] and TTL-tagged objects [35], or as protocol-internal optimizations; we focus on the latter.

A lease is, conceptually, a directional limited-time *promise* that a *grantor* node makes to a *grantee*. It relies on bounded clock speed drift between the two ends, that is, over a given physical expiration time t_{lease} elapsed, the two nodes' clocks do not deviate more than a small t_Δ (no clock drift). This is typically true in today's distributed system environments [40, 53, 68, 77]; note that it does not assume synchronized clock timestamps [10, 30, 70] (no clock skew).

Standard One-to-One Leasing. The procedure of activating one lease between two nodes consists of an initial *guard* phase and repeated *renew* phases, depicted in Figure 2. The guard phase (left half) helps establish the first iteration in the presence of clock skew assuming bounded clock drift, and the renew phases (right half) keep it refreshed periodically [41, 81, 82]. Grantor starts to consider the lease granted when the first `Renew` is sent out (not when the `Guard` is sent), and grantee starts to consider the lease held when the first `Renew` is received (not when the `Guard` is received), using carefully designed timer durations as shown in Figure 2. The goal is to maintain this invariant: the grantor-side expiration time is *never earlier than* the grantee-side. A lease is considered held by the grantee when its clock has not surpassed $t_{\text{lease}}-t_\Delta$ after the last renewal received. The grantor can proactively deactivate the lease with a `Revoke` or, in the case of unresponsiveness, wait for $t_{\text{lease}}+t_\Delta$ without granting to let it safely expire.

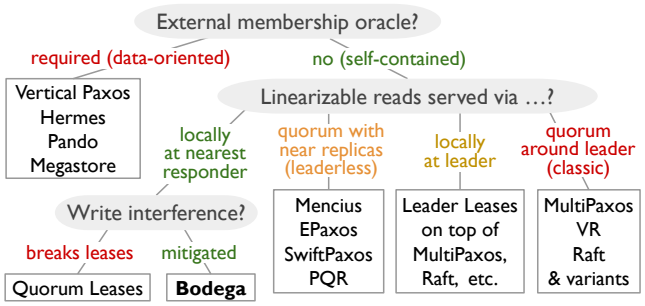


Figure 3. Categorization chart of relevant protocols. *Ideal properties for local reads are marked in green. See §2.3.*

2.3 Previous Work on Read Optimizations

Figure 3 presents a coarse-grained categorization of previous approaches to linearizable reads. The following sections discuss them in the general order from right to left.

2.3.1 Classic Protocols & Leader Leases

Protocols such as **MultiPaxos** [61], **VR** [85], and **Raft** [87] are the de-facto standards implemented in the wild [30, 35, 50, 103, 107]. While stale read options exist [25, 86], normal reads are treated obliviously just like other commands.

With MultiPaxos terminology, a typical protocol is as follows. A node S first makes a "covering-all" `Prepare` phase to settle a unique, higher *ballot* number for all non-committed *instances* (i.e., *slots*) in the log, effectively stepping up as a leader. Without competing leaders, S takes a client command, assigns the next vacant log slot, broadcasts `Accept` messages, and waits for $\geqslant m = \lceil \frac{n}{2} \rceil$ `AcceptReplys` with matching ballot including self (where n is an odd cluster size), after which the slot is marked committed and `Commit` notifications are broadcast asynchronously as announcement. S executes the commands in contiguously-committed slots in order and replies to their clients.

Leader Leases [25] are a commonly deployed optimization to establish *stable leadership*. All nodes grant lease to the most recent leader they are aware of (including self) after invalidating any old lease given out. If a leader S is holding $\geqslant m$ leases, it can safely assert that it is the only such leader in the cluster, i.e., the stable leader. Therefore, S (and only S) can reply to read requests locally using the latest committed value, knowing that no newer values could have committed.

2.3.2 Leaderless or Multi-Leader Approaches

Leaderless (or multi-leader) protocols distribute the responsibilities of a leader onto all nodes, improving scalability and latency under wide-area settings by allowing a fast-path quorum nearer to the clients. However, they are sensitive to command interference and often make local reads infeasible without degrading back to a leader-based protocol.

Mencius [76] assigns the leader role Round-Robin across nodes based on slot index. This mainly benefits scalability.

EPaxos [80, 104] absorbs the idea of inter-command dependencies from **Generalized Paxos** [62] and lets any node to act as the *command leader* for nearby clients. Nodes attach to each command its dependency set; without concurrent conflicting proposals, consensus can be reached on the fast path of `PreAccepts` by a (super-)majority quorum. Conflicts in dependencies require a second phase to resolve. Local reads are inherently hard to achieve in such a protocol without degrading to a leader-based protocol on keys of interest [104].

SwiftPaxos [98] improves EPaxos slow path to 1.5 RTTs (vs. 2) by re-introducing a leader in the slow phase.

PQR [27, 42] applies EPaxos-like leaderless optimization to only reads and not writes. Clients broadcast read requests directly to the nearest majority of servers. If all replies contain the same latest-seen value, all in committed status, then this value must be a valid linearizable read result. Otherwise, the client starts a *rinse* phase where it repeatedly polls arbitrary servers for commit confirmation of the value.

WPaxos [1] is a multi-leader design that partitions the key space and assigns a different leader per partition, improving locality of regional hot keys. However, it expects workload concentration and does not offer local reads at multiple replicas. Its adaptive partitioning approach can be applied orthogonally to consensus protocols including Bodega.

2.3.3 Enhanced Read Leases

Several works explored enhancements to *read leases* beyond stable leadership, enabling broader local reads.

Megastore [10] grants read leases to all replicas by a standalone coordinator. These leases carry the promise of not permitting any writes to covered keys. When writes arrive, leases are actively revoked (requiring an extra round-trip to all replicas) and local reads at followers are disabled until leases are re-granted. Megastore leases cover either all replicas or none; they also require external coordination and experience long downtimes during concurrent writes.

Quorum Leases [81, 82] extend read leases to configurable subsets of replicas. Leases are granted by replicas themselves, removing the need for an external coordinator. Upon writes, revocations are merged with the natural `Accept` messages and their replies, avoiding extra round-trips for writes in failure-free cases. Quorum Leases improve the configurability and write performance aspects of Megastore, but three insufficiencies with reads remain. ❶ Lease actions remain on the critical path, leading to frequent interruptions from writes. ❷ When fast-path local reads fail during lease downtimes, they are redirected to the leader or retried indefinitely by clients, leading to suboptimal slow-path latency. ❸ Assignment of grantees is configured with normal consensus commands, making failure cases hard to reason about and implement. Bodega eliminates all three insufficiencies via the powerful mechanism of background roster leases, as will become clear in §3.

2.3.4 With External Coordination

Protocols below assume external coordination.

Hermes [56] is a primary-backup replication protocol inspired by cache coherence protocols. It allows reads to be completed by individual nodes assuming that writes reach all nodes and are resolved synchronously with respect to each other (similar to CPU shared cache invalidation). Hermes inherits its architectural assumption from **Vertical Paxos** [64], requiring an external membership manager for reconfigurations upon failures.

Pando [106] is a WAN-aware, erasure-coded protocol that emphasizes cost efficiency. It allows statically tunable read-write quorums, which are settable ahead-of-time before deployment, and assumes a network topology with frontends and an external service for membership management.

2.4 Summary of Goals

After reviewing existing solutions, we summarize the desired properties of a linearizable read protocol as our design goals:

- *Self-contained*: no external metadata oracle dependencies.
- *Local reads anywhere*: enable local linearizable reads at arbitrary subsets of replicas as appropriate.
- *Local reads at anytime*: keep reads localized during concurrent interfering writes, minimizing degradation time and maintaining good slow-case latency.
- *Configurable*: tunable against arbitrary ranges of keys.
- *Non-intrusive*: designed atop classic consensus, introducing marginal performance impacts on writes and retaining availability under any minority number of failures.

Via these goals, Bodega delivers superior performance characteristics compared to aforementioned approaches. We will show them both theoretically (§4) and experimentally (§6).

3 Design

In this section, we present the core design of Bodega.

Design Outline. We derive the complete design in three steps: ❶ define the roster, ❷ design optimal normal case operations assuming replicas agree on the same stable roster, and ❸ introduce all-to-all roster leases, the enabler behind the fault-resilient agreement on the roster.

For clarity, we adhere to Paxos-style terminology throughout this paper. All the optimizations are applicable to Raft-style protocols due to their fundamental duality [110].

3.1 The Roster

We start by introducing the core concepts behind Bodega: responder status and the roster. A node is a *responder* for a key if it is expected to serve read requests on that key locally without actively contacting other nodes. A *roster* is the collection of each node's desired capabilities at a certain time; specifically, it dictates:

- The node ID of the current leader node.
- For each (range of) key(s): the node IDs of its responders.

The roster is a generalization of leadership from classic protocols: besides the one special leader role, we now have special responder roles for selected keys. The leader can be implicitly treated as a responder for all keys, and different keys can additionally mark different nodes as responders.

The optimal choice of responders for each key depends on various factors: ❶ client locations and proximity, ❷ workload read-heaviness and skewness, and ❸ cluster topology and status. This paper focuses on the mechanisms supporting the roster rather than the policies for tuning it; we recognize that the latter could be an intriguing study on its own.

The system starts from an empty roster with a null leader ID and an empty responder set for the entire keyspace. Every newly-proposed roster is associated with (and identified by) a unique ballot number, forming a $\langle bal, ros \rangle$ pair, where *bal* is the ballot number formed by concatenating a monotonically increasing integer b with the proposing node's ID r to ensure uniqueness. Rosters of different ballots may contain the same content but are still considered different. Roster changes may happen due to explicit requests by users, automatic tuning from statistics, or mandatorily in reaction to failures.

3.2 Normal Case Operations

We first describe normal case operations, using Figure 4 as a demonstrative example. In a 5-node cluster, S0 is the leader (depicted by the crown) and S2,3,4 are additional responders (depicted by the red star symbols) for a specific key k. Assume, in this section, that this is the latest roster all nodes know and consider stable according to leases. Nodes use their known roster to assure that writes to k would never commit before reaching all of its active responders. A responder can therefore serve reads directly if it knows the latest value of k will commit; when unsure, optimizations exist.

3.2.1 Writes

Writes follow the same leader-based process as in MultiPaxos (Figure 4 blue arrows), except for an updated commit condition. Normally, a write to key k can be marked as committed and acknowledged once $\geqslant m$ `AcceptReplys` are received. We impose an additional constraint that it must also have received replies from all the responders for k, according to the leader's current roster.

Requiring a write quorum that covers all responders is an unavoidable penalty that any local linearizable read algorithm must pay. Luckily, without far-off responders, this penalty is marginal as the system usually picks a leader with relatively uniform distances to other replicas, and the write must anyway reach a majority. This aligns with previous observations [81] and our evaluation results (§6). Distant responders could still be appropriate for certain workloads.

3.2.2 Reads

Clients send read requests on key k to the closest responder server for k (Figure 4 green arrows). It is common for clients of wide-area systems to be co-located with some replica; for example, consensus is usually part of an outer system, e.g., a database, where requests come directly from participating sites, but this is not a requirement.

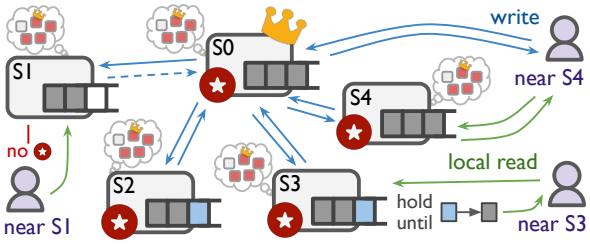


Figure 4. Normal case operations of Bodega. *Assume all nodes agree on the same example roster: S0 is the leader and S0,2-4 are responders for a key while S1 is not. See §3.2.*

When a server S (with a stable roster) takes a read, there are three cases. ❶ S is the leader, in which case S simply finds the highest committed slot in its log that contains a write to k and returns the value. The leader does not need to worry about in-progress writes [25]. ❷ S is neither a leader nor a responder for k (e.g., S1 in Fig. 4), in which case S rejects the read and promptly redirects the client to another server, preferably a close-by responder or the leader. ❸ S is a non-leader responder. In this case, S looks up the highest slot in its log that contains any write to k. If the slot is in committed status, S immediately replies with the value (e.g., the read at S4 in Fig. 4); otherwise, S cannot yet determine whether that value will surely commit or will be overwritten due to impending failures (e.g., the read at S3), in which case S optimistically *holds* the read.

Optimistic Holding. S expects its connection with the leader to be normally healthy, hence anticipates quick commitment for in-flight writes. It is usually faster and more efficient for S to withhold local reads that depend on an in-flight write and to reply as soon as commitment is known, than rejecting the reads and letting clients redirect or retry. In failure-free cases, the commit notification for an interfering write arrives in at most one RTT (from when S receives the `Accept` from the leader and replies to it). Note that even with a constant stream of writes, held reads are not blocked indefinitely: they are released as soon as their associated slot turns committed.

A responder S optimistically holds a local read by adding it to a *pending set* attached to the corresponding slot. Upon receiving the commit notification for a slot, S releases the pending reads and replies with the committed value. To handle cases where the leader fails to notify S promptly, clients start an *unhold* timeout when sending local read requests; if the timeout is reached, clients proactively issue the same request to another responder or the leader (with the same req ID, which is safe since reads are idempotent) and use the earliest reply. A good timeout length is longer than the usual RTT between S and the current leader.

Early Accept Notifications. Bodega can optionally incorporate a further optimization that reduces average holding time when under low load: followers not only reply to `Accepts` on key k to the leader, but also broadcast notifications to k's responders. Once a responder S has received m notifications (counting self), it can assert that a pending slot will surely commit even across minority failures. A similar optimization exists for BFT writes [23]. On average, this halves the expected holding time for interfered local reads. This optimization is optional and can be turned off when write traffic increases.

3.3 Roster Leases

So far, we have assumed a consistent roster across the cluster without showing how it is achieved. The idea is to exchange roster leases in an all-to-all manner, between at least a majority of nodes and all nodes that may be responders for some key. When holding a majority number of leases, responders know that the roster is stable, and the leader (as an implicit responder) also knows the identity of all responders and will not commit writes without notifying them.

	Lease Primitive	Leader Leases	Roster Leases
Pattern	one-to-one	all-to-one	all-to-all
Safe #grantors	1	m	m
Must to whom	-	leader	responders

We present how Bodega deploys off-the-critical-path *roster leases* to establish a stable roster elegantly and efficiently. We use Figure 5 as an illustration when needed.

Lease-related States. Besides the SMR log and the $\langle bal, ros \rangle$ pair, we let every node S act as both a lease grantor and a grantee (recall §2.2 for how a standard lease grant primitive works). This means S maintains the following additional data structures: ❶ two lists of grantor-side timers $T_{\text{guardTo},\,p}$ and $T_{\text{renewTo},\,p}$ per peer node P; ❷ two corresponding sets $\{\}_{\text{guardTo}}$ and $\{\}_{\text{renewTo}}$ for tracking which peers are S currently guarding/renewing to; ❸ two lists of grantee-side timers $T_{\text{guardBy},\,p}$ and $T_{\text{renewBy},\,p}$ per peer P; ❹ two sets $\{\}_{\text{guardBy}}$ and $\{\}_{\text{renewBy}}$ for tracking the guards/renewals S currently holds; ❺ a list of safety slot numbers $thresh_p$, that specifies the highest slot S has accepted from each peer P.

3.3.1 Roster Leases Activation

We first describe how roster leases are activated. Consider node X wants to announce a new roster ros'; this could be due to, e.g., stepping up as new leader (by setting X as the leader in ros') or other reasons covered in §3.3.2. X composes a unique, higher ballot bal' by concatenating $(b+1)$ with its node ID, where b is the higher part of the current bal. X then broadcasts the $\langle bal', ros' \rangle$ pair to all nodes including self.

For any node S upon receiving a ballot bal' higher than ever seen, it first ensures all old leases are safely revoked or expired (discussed later in §3.3.2). Then, it moves on to $\langle bal', ros' \rangle$ and starts a `initiate_leases(`bal'`)` procedure, where it begins granting leases for the new roster to all peers asynchronously in parallel.

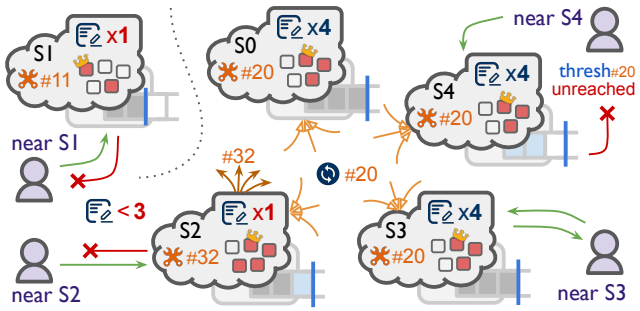


Figure 5. All-to-all roster leases demonstrated. *S0,3,4 are each holding ≥ majority grants of roster #20; S4 has not seen all slots up to #20's safety threshold. S1 is stuck with an older roster of #11. S2 is initiating a new roster of #32. See §3.3.*

To each peer P, the procedure obeys standard lease granting: S and P first complete the guard phase, exchanging a sequence of `Guard`, `GuardReply`, `Renew`, and `RenewReply`, and utilizing proper timers along the way. If all goes well, S should have P in its $\{\}_{\text{renewTo}}$ and have T_{renewTo} properly extended; it repeats renewals periodically to keep the S-to-P lease refreshed. Similarly, P should have S in its $\{\}_{\text{renewBy}}$ and have T_{renewBy} kicked off properly. Whenever a $T_{\text{intent},\,p}$ times out for any intent among the four, the peer is removed from the corresponding set $\{\}_{\text{intent}}$, leading to a retry of the guard phase or a proposal of a new roster.

After transitioning to ros', if S sees itself being the leader of ros', it does the usual step-up routine of redoing the `Prepare` phase for non-committed slots of its log.

Stable Condition & Safety Thresholds. As shown in §2.2 and above, a node P is considered granted a lease by S when S ∈ P's $\{\}_{\text{renewBy}}$ set. Assume P itself is always in the set. The size of this set, $|\{\}_{\text{renewBy}}|$, indicates the number of lease grants P currently holds. When $|\{\}_{\text{renewBy}}| \geqslant m$, then P knows at least a majority number of nodes in the cluster has the same latest $\langle bal, ros \rangle$ as P and that at most one such roster exists; this is called the *stable* roster of the cluster and is a necessary precondition for all optimizations described in §3.2. For example, in Figure 5, the local reads at S1 and S2 are rejected due to an insufficient lease count.

This condition alone is not enough, though. When a node directly inspects its log and uses the highest slot index for local reads, it is assuming that its log is up to date and contains all the recently committed instances; this is usually true, but could be violated during the short period of time when a node that has fallen behind joins a new roster. To address this, a node should be informed of other peers' acceptance progress when transitioning to a new roster.

We let `Guard` messages from S to P carry an extra number, which is the highest slot number that S has ever accepted. P stores the number in its *thresh* list. A node permits local reads only if it has committed all the slots up to the m-th smallest slot number in its *thresh* list; otherwise, it might not have observed the latest committed writes yet. S4 in Figure 5,

for example, has not reached this condition and thus cannot start serving reads locally yet.

In summary, all the stable leader and local read operations of §3.2 are preceded by the following stable condition check:

$$|\{\}_{\text{renewBy}}| \geqslant m \quad (1)$$
$$\wedge \ \exists \text{ size-}m \text{ subset } E \subseteq \{\}_{\text{renewBy}} :$$
$$\text{committed all slots up to } \mathit{thresh}_p, \forall p \in E$$

If the check fails, the operation falls back to classic consensus as if it is a write, which does not require this check.

3.3.2 Revocation & Expiration

Most roster lease activations happen when there are ongoing old leases in the system. Broadly speaking, a roster change may be triggered by one of the following reasons.

- Node initiates a new roster in reaction to suspected failures, removing failed nodes from special responder roles.
- Node autonomously proposes a new, more optimized roster according to collected workload statistics.
- Node receives an explicit roster change request from user.

In either case, before `initiate_leases()`, a node S always invokes the `revoke_leases(`*bal*`)` procedure synchronously to ensure that all the leases it is granting or holding with the older ballot *bal* are safely revoked and removed. To do so, S clears its $\{\}_{\text{guardTo}}$ set and broadcasts `Revoke` messages carrying the old ballot. Whenever a node P receives a `Revoke` with matching ballot from S, it removes S from the $\{\}_{\text{guardBy}}$ and $\{\}_{\text{renewBy}}$ sets and replies with `RevokeReply`.

S either receives a `RevokeReply` from P promptly (common fast case) or has to wait for expiration timeout (failure case), after which it removes P from $\{\}_{\text{renewTo}}$. Note that, unless failures occur and force a wait on expiration, a roster change completes swiftly within two message rounds: one for the revocation and the other for the initiation guards.

3.3.3 Piggybacking on Heartbeats

Distributed systems already deploy heartbeats for tasks such as health tracking [35, 52, 80, 94]. This opens the opportunity to enable roster leases without common-case overheads, by piggybacking lease messages onto existing periodic heartbeats. Bodega piggybacks all the `Renew` and `RenewReply` messages onto heartbeats, and uses a proper heartbeat interval $t_{\text{hb_send}}$ such that leases are refreshed in time. Heartbeat messages also carry the sender's $\langle \mathit{bal}, \mathit{ros} \rangle$ pair to let receivers discover roster changes.

Each node has per-peer timers $T_{\text{heartbeat},\,p}$ which are used for promptly detecting failures from peers; a peer is considered down if no heartbeats were received from it for $t_{\text{hb_fail}}$. A rule of thumb for choosing good timeout lengths for a cluster is:

$$\text{avg. RTT} < t_{\text{hb_send}} \ll t_{\text{hb_fail}} < t_{\text{guard}} = t_{\text{lease}} \quad (2)$$

Bodega uses the following defaults for wide-area replication: $t_{\text{hb_send}} = 120\text{ms}$, $t_{\text{hb_fail}} \approx 1200\text{ms}$, $t_{\text{guard}} = t_{\text{lease}} = 2500\text{ms}$.

Protocol	W	R	R^*	D^*	⚡	⬚
Leader Ls	$l+m$	l	l	-	●	○
EPaxos	$c+M$	$c+M$	$c+M+m$	M	●	○
Hermes	$c+N$	c	$c \sim c+\frac{N}{2}$	$\frac{N}{2}$	◐	○
PQR	$l+m$	$c+m$	$c+m*\text{rinses}$	m	●	○
PQR (+ Ldr Ls)	$l+m$	$c+m$	$c+m+l$	m	●	○
Pando	$c+m$	$c+m$	$c+N$	N	◐	◐
Megastore	$l+2N$	c	$c+l$	$2N$	◐	○
Quorum Ls	$l+N$	c	$c+l$	$2N$	●	●
Qrm Ls (passive)	$l+N$	c	$c+l$	N	◐	◐
Bodega	$l+N$	c	$c \sim c+\frac{m}{2}$	$\frac{m}{2}$	●	●

Table 1. Qualitative comparison across protocols *assuming the most read-optimized config of each protocol.* ***W****: write latency;* ***R****: read latency if quiescent;* ***R**** *: read latency if there is an interfering write;* ***D**** *: read performance degradation period length.* ⚡*: fault tolerance (without external oracle).* ⬚*: allows tunable rosters.* $\underline{l}$*: client-leader RTT,* $\underline{c}$*: client-nearest server RTT,* $\underline{m}$*: time to establish simple majority,* $\underline{M}$*: time to establish super majority,* $\underline{N}$*: time to form all-nodes quorum. Cell is shaded darker if value is higher.*

4 Formal Presentation and Proof

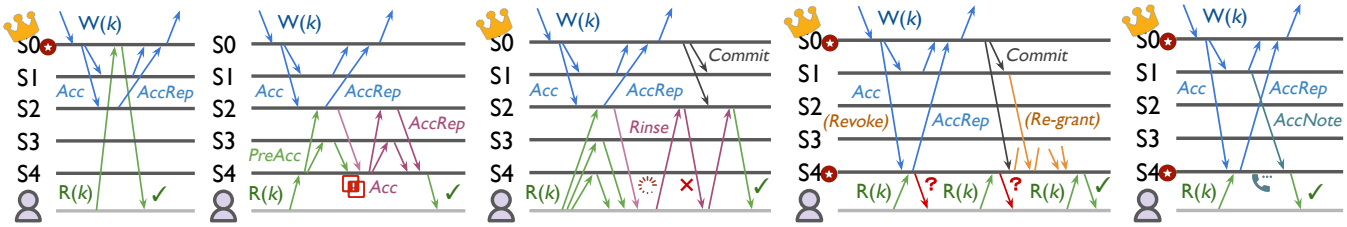
We provide a qualitative comparison across notable protocols (Figure 6, Table 1), a formal presentation of the Bodega algorithm (Figure 7), and a concise proof of correctness.

4.1 Comparison Across Protocols

In Table 1, we model the normal-case write and read latency, degraded read latency under write interference, and degradation period length of related protocols. Cells are shaded according to example values from the Figure 8(b) GEO setting (lighter is better). We also indicate whether a protocol retains the fault tolerance of classic protocols and whether it allows tunable configs. If tunable, we use its most read-optimized config that tolerates $f = \lfloor \frac{n}{2} \rfloor$ faults. Assume only one interfering write.

The following symbols are used to model performance. $\underline{l}$: RTT between client and the leader, $\underline{c}$: RTT between client and its nearest server, $\underline{m}$: time to establish a simple majority quorum (i.e., to reach majority nodes from some server and receive replies), $\underline{M}$: time to establish a super majority quorum (as in EPaxos [80]), $\underline{N}$: time to form a quorum composed of all nodes. For an average client in typical WAN-scale settings, one should expect $c \ll l \approx m < M < N$.

Most results are derived naturally from Figure 6, §2.3, and §3.1-3.3. We provide supplementary explanations. **PQR (+ Ldr Ls)** is a straightforward variant of PQR combined with Leader Leases; if a near quorum read attempt fails, the client contacts the stable leader directly, bounding slow-path latency by $c + m + l$. We assume Quorum Leases always incorporate Leader Leases. **Qrm Ls (passive)** is a variant of Quorum Leases where we deliberately let grantees keep



(a) Leader Leases **(b)** EPaxos **(c)** PQR **(d)** Quorum Leases **(e)** Bodega

Figure 6. Timeline comparison across protocols of linearizable reads *in the presence of an interfering write.* **W(k)***: write key k,* **R(k)***: read key k,* ***Acc****: Accept,* ***AccRep****: AcceptReply,* ***PreAcc****: EPaxos PreAccept, : EPaxos dependencies conflict,* ***Rinse*** *: PQR repeated poll.* ***Commit****: commit notification,* ***AccNote****: accept notification (optional), : Bodega optimistic holding.*

the leases upon accept to show the upper bound of Quorum Leases performance, saving one re-granting RTT from the degradation time. Doing so risks blocking fault-induced roster change commands as described in §3.3. Hermes uses primary-backup broadcast and thus requires external coordination for fault tolerance; Megastore is similar. Pando uses a pre-deployment planner to dictate erasure coding and quorum composition. Bodega achieves the best across all metrics and retains fault tolerance and configurability.

4.2 Formal Presentation

Figure 7 presents a complete, code-oblivious summary of the Bodega algorithm, which can be used as a reference for implementors. We also provide a formal, model-checked TLA$^+$ specification in Appendix A.

4.3 Proof

We provide a proof of Bodega's local read linearizability and write liveness, assuming well-established results of the safety and liveness of leases [41]. For linearizability, only locally-served reads need proof, as Bodega behaves the same as classic consensus otherwise.

Linearizability. A local read R served by server S observes any write W acknowledged cluster-wise before R was issued.

Proof. R is served locally only if S is a responder that passes the stable roster check (1). Let the stable roster be $\langle bal, ros \rangle$.

Case #1: W was committed on a ballot $> bal$. It is impossible because the latest ballot on at least a majority is bal.

Case #2: W was committed on a ballot $= bal$. By the injective ballot-roster mapping and the commit condition of writes, S must be in W's write quorum and have W in its log.

Case #3: W was committed on a ballot $< bal$. By majority intersection, for any size-m subset $E \in$ S's $\{\}_{\text{renewBy}}$, at least one of the lease grantors $P \in E$ accepted W at its committed slot x before granting to S. This implies $thresh_p \geqslant x$. □

Liveness of Writes. A write W can always eventually make progress if retried on a majority group G of healthy servers.

Proof. By the property of leases, after old leases expire, a roster change can eventually be made on all servers $\in G$ to restrict the leader and all the responders to be contained in G. Then, normal consensus applies. □

5 Implementation

The Summerset KV-store. We develop Summerset, a distributed, replicated, protocol-generic key-value store. Summerset is written in async Rust/`tokio` using a lock-less architecture and serves as a fair codebase for evaluating consensus and replication protocols.

The codebase has 13.6k lines of infrastructure code and covers five protocols of interest: MultiPaxos w/ Leader Leases (2.5k), EPaxos (3.1k), PQR & variant (2.8k), Quorum Leases & variant (3.2k), and Bodega (3.0k). All protocol implementations have passed extensive tests. The full Summerset codebase is open-sourced at https://github.com/josehu07/summerset/tree/bodega-artifact.

5.1 Smart Roster Coverage

In cases where users desire local reads but cannot observe workload patterns externally, Bodega servers can collect statistics and automatically propose roster changes to mark servers as responders for proper keys. Our default implementation traces per-key read/write request counts grouped by clients' preferred nearby server IDs. For a key, if > 95% requests are reads at a periodic check, then servers near > 20% of the reads are added as responders. More sophisticated strategies exist; for example, straggler detection can help remove fail-slow nodes from responders promptly [51, 54].

5.2 Lightweight Heartbeats

In §3.3, we described roster leases as if all heartbeats carry the complete roster data structure. In practice, rosters with fine-grained key ranges can get large (tens of KBs). Luckily, most heartbeats in Bodega are *lightweight heartbeats*: the sender puts in only the ballot number to indicate that the roster has not changed from previous heartbeats. Full-sized heartbeats are sent when changes occur.

Similarly, clients may request a server to send the roster along with a command reply, and then cache this roster as a heuristic for choosing the best responder for local reads.

5.3 Other Practical Details

Request Batching. As is common practice, Bodega deploys request batching at servers (at 1 ms intervals) for non-local-read commands. Each log slot contains a batch of requests

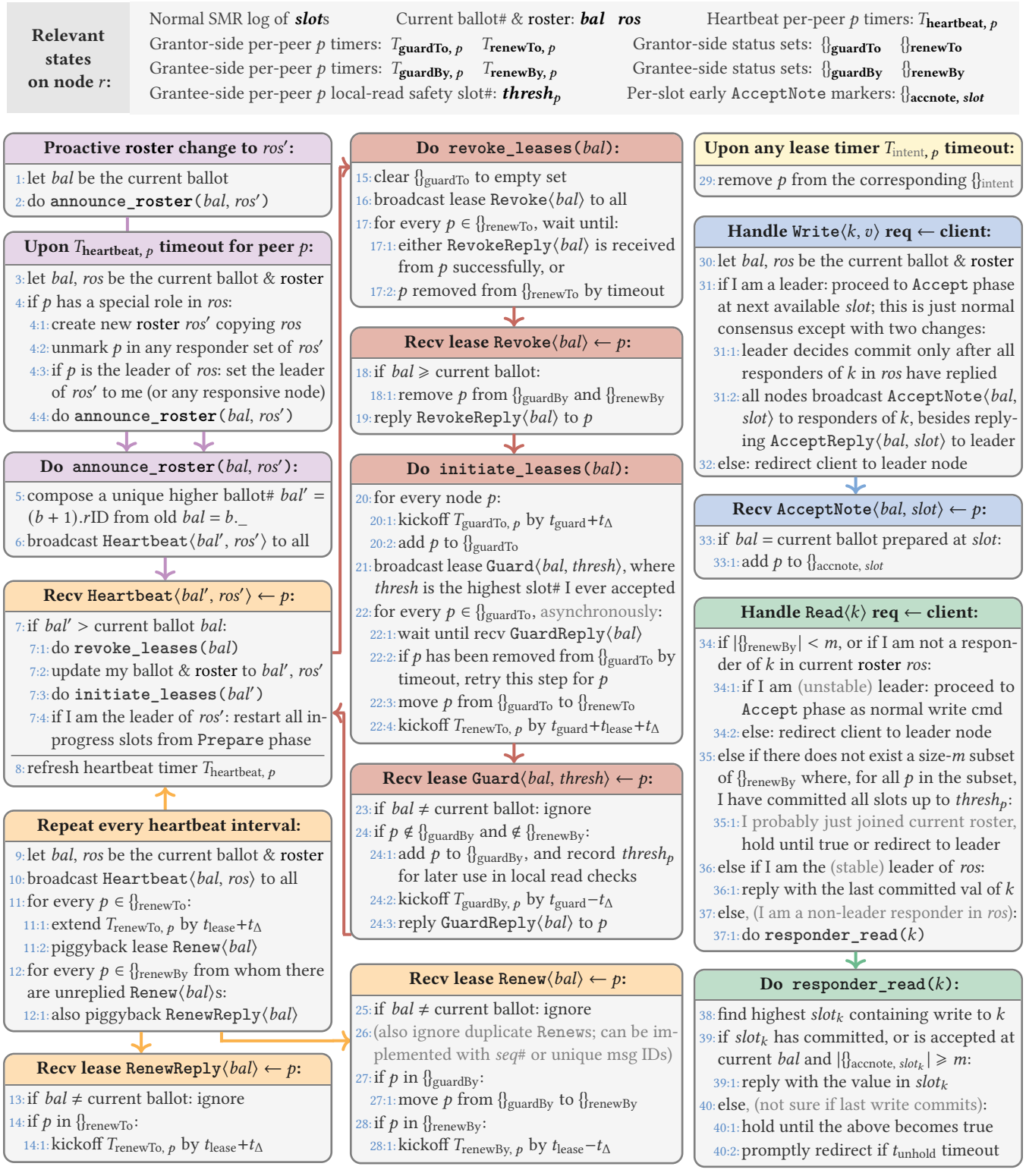


Figure 7. Complete Summary of the Bodega algorithm. *Lists all actions a node r would take upon certain conditions, grouped by purposes for clarity: triggers for a new roster, granting procedure of new roster leases, heartbeats and lease renewals, handling client write requests, handling client read requests. The description is based on a regular key-value API. Nodes implicitly retransmit non-acked messages. Broadcast msg receivers include the sender itself. Clock drift between nodes is assumed to be bounded by t_Δ, as is required by any distributed lease algorithm; clock skews are irrelevant thanks to Guards. The arrows annotate a natural reading order that follows the usual flow of the protocol.*

and the commit condition is checked for all writes contained.

Snapshots. Bodega servers take periodic snapshots of the executed prefix of the log [87]. Local reads past the beginning of truncated log look up the latest snapshot directly.

Membership Management. Membership changes are handled identically to *reconfigurations* in other protocols [25, 80, 85], just with an extra step of proposing and stabilizing an empty roster with no responders ahead of the change.

6 Evaluation

We do comprehensive evaluations to answer these questions:

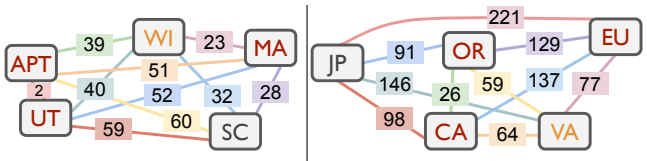
- How does Bodega perform compared to other protocols under microbenchmarks of various write intensities? (§6.1)
- What do the request latency distributions look like? (§6.2)
- What are the impacts of write interference, and how does performance change with write ratios & value sizes? (§6.2)
- How do roster changes impact performance? (§6.3)
- How does Bodega behave with different choices of responders and different coverages of keys? (§6.3)
- How does Bodega compare with production coordination services, etcd & ZooKeeper, under YCSB workloads? (§6.4)

Experimental Setup. All experiments are run on two CloudLab [33] clusters, hereafter called WAN and GEO, shown in Figure 8. WAN is a wide-area cluster spanning five CloudLab sites with nodes of similar hardware types: WI-`c220g5`, UT-`xl170`, SC-`c6320`, MA-`rs620`, and APT-`r320`. §6.1 also includes a GEO cluster of five `c220g5` nodes emulated with Google Cloud RTTs reported in previous work [104] using Linux kernel `netem` [71]. All nodes' public NICs have 1Gbps bandwidth. The orange-colored site in Figure 8 denotes the leader and the red-colored sites denote responders.

Clients are launched on machines evenly distributed across all datacenters, each marking the nearby server as their preferred server for local reads when eligible. All machines run Linux kernel v6.1.64 and pin processes to disjoint cores. All protocols use 120 ms heartbeat interval, 1200±300 ms randomized heartbeat timeout, and 2500±100 ms lease expiration (if applicable). All protocols send immediate `Commit` notifications: whenever a commit decision is made, `Commits` are broadcast to other servers promptly.

6.1 Normal Case Performance

We run microbenchmarks on both cluster settings and compare: ordinary MultiPaxos, Leader Leases, EPaxos, PQR, PQR (+ Leader Leases) variant, Quorum Leases, Quorum Leases (passive) variant, and Bodega. We spawn 50 closed-loop clients with 10 near each server and let all clients run a microbenchmark with 1k 8B-size keys and 128B values; keys are chosen uniformly. We test three write percentages in the workload mix: 0%, 1%, and 10%. Figure 9 shows the normalized throughput (w.r.t. Leader Leases), avg. read latency, and avg. write latency perceived by clients at different locations. Leader and responders are set as depicted in Figure 8. The



(a) WAN (CloudLab sites) **(b)** GEO (GCP [104], emulated)

Figure 8. Evaluation settings. *Orange denotes designated leader node and Red denotes other responders, if relevant. The edges mark per-pair RTT values in milliseconds. See §6.*

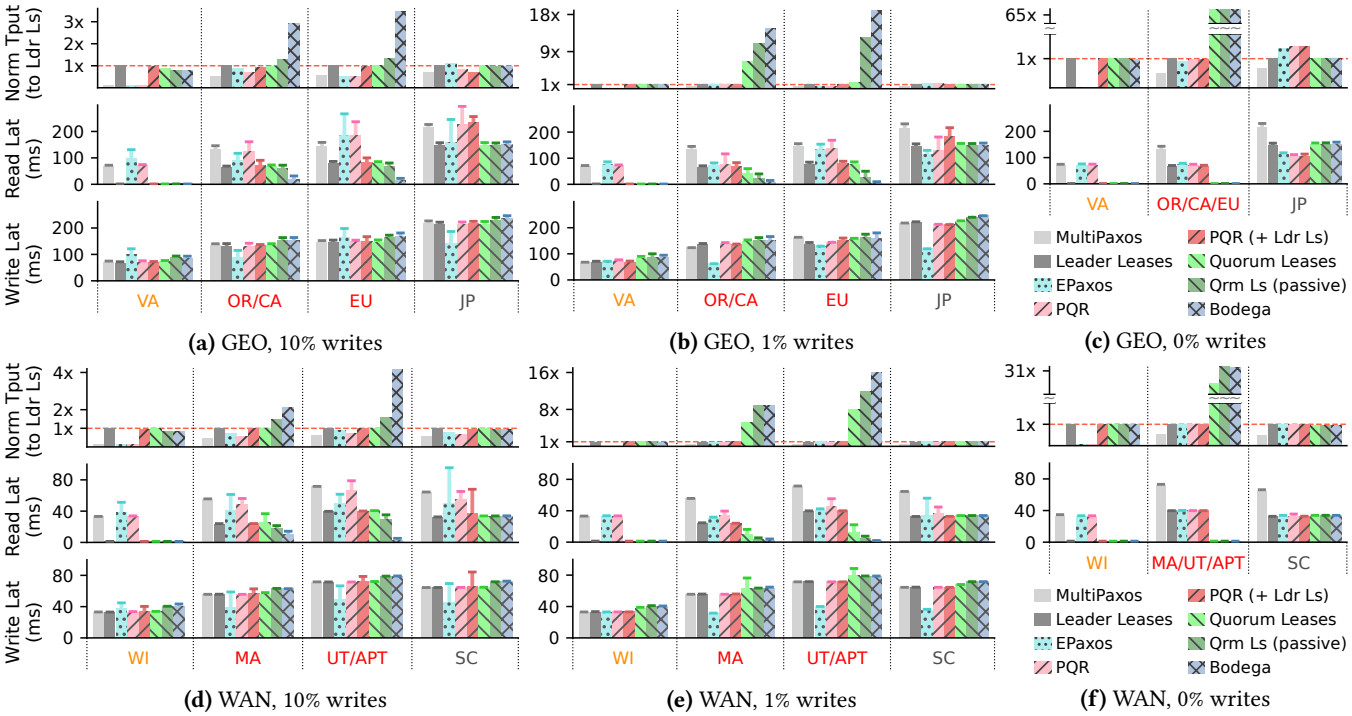
red dashed lines indicate baseline Leader Leases throughput, and the top Ts on latency bars indicate P99 latency.

The results yield the following observations. First, except for a few datapoints (which we soon discuss), both GEO and WAN clusters exhibit similar performance patterns, just with different absolute values due to RTT differences.

Second, for writes, all protocols except EPaxos exhibit similar performance. Quorum Leases and Bodega have higher write latency due to the requirement of writes reaching responders; this explains the small throughput gap between them and Leader Leases for near-leader clients with 10% writes. EPaxos delivers better average (but not P99) write latency due to its leaderless write protocol design.

Third, we observe coherent patterns for read performance. ❶ Compared to ordinary MultiPaxos, Leader Leases cut read latency for near-leader clients to nearly zero, but other clients still pay an RTT to the leader for reads. ❷ PQR (and its Leader Leases variant) show worse (or identical) performance compared to Leader Leases. The only exception is in the GEO, 0% writes setting for the JP clients; they are so far away from the leader that a nearer majority quorum actually helps, letting them outperform local read protocols (since JP is not marked as a responder). ❸ EPaxos has similar read performance as PQR but with higher P99 latency when there are writes. ❹ Both Quorum Leases variants and Bodega show the same performance as Leader Leases for clients near the leader or a non-responder. ❺ Quorum Leases and Bodega both deliver extraordinary read performance for clients near responders when with 0% writes. ❻ Bodega sustains this read performance advantage and keeps read latency close to zero for higher write intensities. In contrast, Quorum Leases performance quickly drops and almost degrades back to Leader Leases for 10% writes. This shows Bodega's resilience to write interference, a crucial advantage.

Throughput to Read Latency Curves. We take the 10% writes WAN setting of Figure 9(d) and run open-loop clients with varying request concurrency up to 100 reqs/sec to profile the throughput-latency curves, which we plot in Figure 10. The x-axis is aggregated throughput and the y-axis is average latency of all requests. The never-local-read protocols (MultiPaxos, PQR, EPaxos) form a cluster at the upper left as expected, due to unavoidable network traffic per read. Leader Leases reveal a throughput upper bound at ~1.8k where the leader node starts to overload; this bound



(a) GEO, 10% writes **(b)** GEO, 1% writes **(c)** GEO, 0% writes

(d) WAN, 10% writes **(e)** WAN, 1% writes **(f)** WAN, 0% writes

Figure 9. Normalized throughput and latency at different client locations *across different write intensities. See §6.1.*

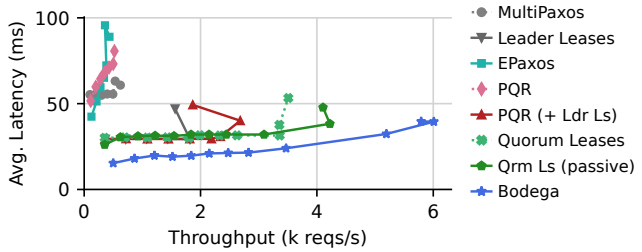


Figure 10. Tput to avg. latency curves *on WAN. See §6.1.*

is slightly higher for PQR + Leader Leases at ~2.2k. Quorum Leases see similar latency under low load and a higher throughput bound at ~3.4k, matching Figure 9. Bodega sustains 1.5x better latency because of most reads being served locally without fallbacks; throughput limit is accordingly higher at ~6k.

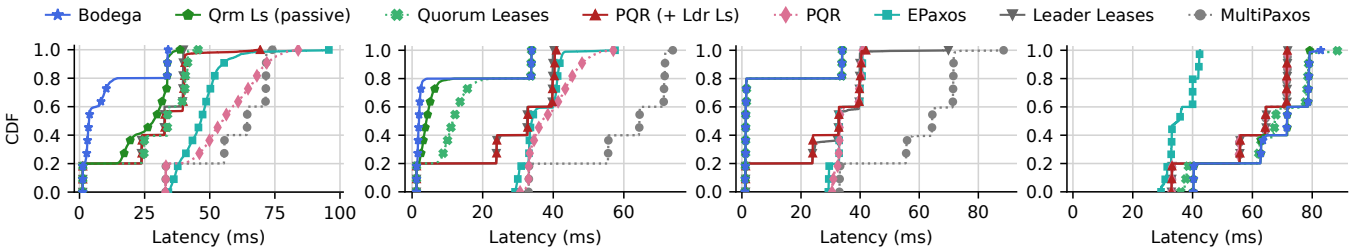
6.2 Detailed Performance Anatomy

We conduct a zoomed-in study across various workloads and metric dimensions.

Latency CDFs. We collect request latency CDFs across all 50 clients of the WAN setting (Fig.9(d)-9(f)) and plot them in Figure 11. Results are filtered to show a single key for a clean pattern. Each site contributes an equal 20% of datapoints.

We make four observations. ❶ Write latencies across all workloads are similar and are presented as one Figure 11(d). Quorum Leases and Bodega show slightly higher write latencies in favor of responder local reads, while EPaxos delivers the same level of latencies as its reads due to its leaderless design. These results align with §6.1. ❷ At 0% writes, all protocols deliver a read performance close to their theoretical best, though a few outlier datapoints remain. ❸ At 1% writes, slight write interference occurs. Quorum Leases reads deviate from Bodega, with the passive variant delivering roughly half the latency of the original variant. ❹ At 10% writes, differences in read latency distributions are the most obvious. MultiPaxos clients' read latency clearly correlates with their distance to the leader; Leader Leases are similar but with a majority-quorum latency subtracted. PQR and EPaxos exhibit suboptimal latency and have high tail latency of up to 100 ms for a read; this is due to the need for conflict resolution. Quorum Leases variants both degrade to Leader Leases. Bodega delivers outstanding local read performance as expected (except for the 20% non-local SC clients).

Visualizing Write Interference. We use a similar setting to the read-only workload in §6.1 on the WAN cluster, but this time with open-loop clients, each sending reads at a rate of 400 reqs/sec to a key. At ~15 secs, we let one client issue a write command to the key. We monitor the average read latency across the three non-leader responders for the local read protocol variants, and plot them over a time axis in Figure 12. We see that the write introduces an interruption to local reads for all three protocols. Both Quorum Leases variants degrade to 40 ms read latency, which is the average RTT to the stable leader; the passive variant shows a shorter degradation duration. Bodega ❶ shortens the degradation time to ~25 ms; recall Table 1, this corresponds to ~$\frac{m}{2}$, and ❷ allows all reads to be held locally and be released at the end of the degradation, leading to better latencies also for disrupted reads.



(a) Reads w/ 10% writes **(b)** Reads w/ 1% writes **(c)** Reads w/ 0% writes **(d)** Writes

Figure 11. Latency CDFs of requests *in the WAN setting across different write intensities, focusing on one specific key. See §6.2.*

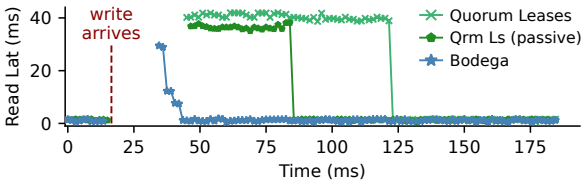


Figure 12. Read latency after an interfering write. *The x-axis is time at which the reads finish. See §6.2.*

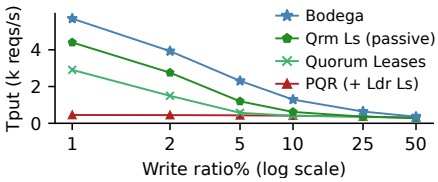


Figure 13. Throughput vs. write ratio. *The x-axis is log-scale (same for Fig. 14).*

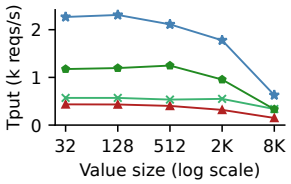


Figure 14. Throughput vs. value size. *See §6.2.*

Varying Write Ratios. We take the same setup as §6.1 on the WAN cluster and vary the write ratios of the workload mix from 1% to 50% while fixing value size to 128B. We report the aggregate throughput in Figure 13. All protocols except the PQR (+ Ldr Ls) baseline show a trend of lower throughput with higher write interference as local reads become less profitable. The results match Figure 9(d)-9(e).

Varying Value Sizes. We repeat the same setup as above and vary the value size while fixing the write ratio at 5%. As expected, Figure 14 shows that smaller values have little impact on performance, but throughput drops with larger values due to slower writes and larger read results to transfer.

6.3 Roster Changes & Composition

We evaluate roster change performance and roster coverage.

Impact of Frequent Failures (Simulation). We first justify roster leases' stability under frequent failures through an exaggerated simulation. We run a 10ms-timeslice-based Monte Carlo simulation with 10% writes using performance numbers measured in Figure 9(d). We apply a failure probability of 0.5% every second–much larger than real-world servers [3]–and a recovery probability of 1% every second. After any failure, the cluster delivers 3 seconds of zero throughput to simulate lease expiration; after any recovery of a responder, the cluster delivers 100ms of zero throughput to simulate a proactive roster change. Figure 15 shows aggregated throughput of 1 million simulated seconds. We can see that, despite these exaggerated conditions, Bodega still delivers up to 2.2x throughput compared to traditional Leader Leases.

Roster Change Duration. On the WAN cluster, we zoom in and compare the duration of two types of roster changes: failure-induced (where waiting for lease expiration is necessary) vs. regular (where revocations complete quickly). Regular roster changes finish in just two message rounds, because they are no more than a lease revocation followed by an initiation. We create an open-loop client near WI and let it issue a 50%-read workload at a rate of 400 reqs/sec to 1k keys. We plot real-time latencies in Figure 16.

At ~800 ms, we crash the UT node, which is one of the responders for the full key range. Writes are immediately blocked since UT as a responder is unreachable. Reads are still served locally without interruption until ~1.1 secs later, when some healthy server in the cluster raises a heartbeat timeout and initiates a change to a new roster where UT is removed from all responder roles. After waiting 2.6 secs for expiration, normal operations continue.

At ~7.2 secs, we make an explicit roster change request to a server. In contrast to the failure case, this roster change completes in just ~75 ms, which is ~2x cluster-wise RTTs as expected. Impacts on client requests are minor.

Choice of Responders. We run a 10%-write workload on all clients in the WAN setting, while using an increasing set of responders for all keys; WI node is still the leader. We report the cluster-wide average read/write latency and their standard deviation in Figure 17. With more nodes added as responders, read latency tends to zero while write latency increases, revealing the expected tradeoff. This demonstrates the importance of allowing adjustable rosters to help avoid unnecessary taxes on writes.

Coverage of Keys. We repeat the same experiment, but vary the percentage of local-read-enabled keys while fixing the choice of responders to Figure 8(a). The cluster-wide average latencies and their standard deviation across the coverage spectrum are plotted in Figure 18. Results show an expected decrease in read latency and a corresponding increase in write latency as local reads are enabled on more keys. This implies the general strategy of enabling local reads for read-heavy keys while avoiding local reads for write-heavy keys.

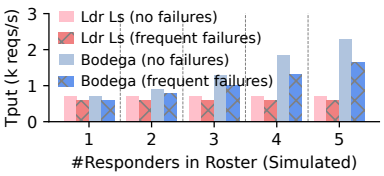


Figure 15. Exaggerated impact of failures *(simulation). See §6.3.*

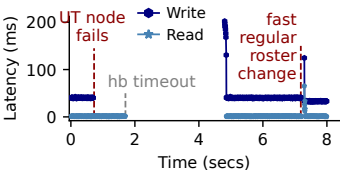


Figure 16. On-failure vs. regular roster changes.

Figure 17. Latency vs. coverage of responders.

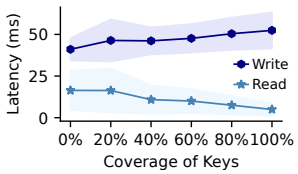
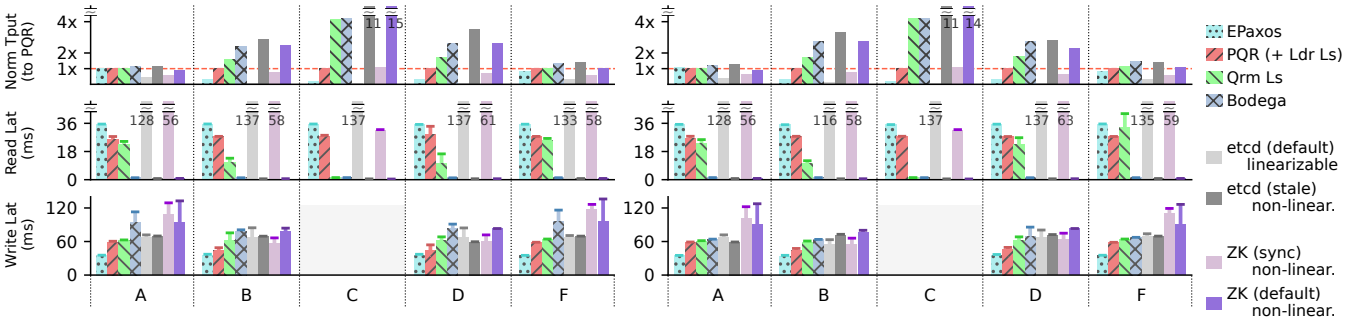


Figure 18. Latency vs. keys covered by roster.



(a) Uniform, full-coverage roster

(b) Zipfian, top-20%-coverage roster

Figure 19. YCSB workloads on Summerset, etcd, & ZooKeeper. *etcd (stale) & ZK (both modes) are not linearizable. See §6.4.*

6.4 Macrobenchmark vs. etcd & ZooKeeper

To evaluate the protocols in a more realistic setup, we compare Summerset protocols with two widely-used coordination services, etcd [35] and ZooKeeper [52], on the WAN cluster. We drive all systems with YCSB [29], the standard KV macrobenchmark. Workloads have the following approximate write ratios (we treat insertions as updates). A: 50% w, B: 5% w, C: 0% w, D: 5% w, F: 25% w.

YCSB Request Distributions. We use 10k keys and construct two scenarios corresponding to two request distributions. ❶ For the Uniform distribution, clients at all locations choose keys uniformly randomly across the key space. Since there are no site-specific preferences for keys, all sites are added as responders for all keys to secure local reads. ❷ For Zipfian, clients at each location choose keys according to a Zipfian-0.99 distribution, skewed towards different sets of keys at different sites; this creates per-site preferences for keys. We then add each site as a responder only for its top-20% accessed keys to derive an asymmetric roster that imposes unnoticeable impacts on write performance.

etcd Modes. We deploy etcd in two modes, both with 120 ms heartbeat intervals. The default mode showcases a standard implementation of vanilla Raft [87]. The *stale* mode turns on the serializable member-local read option for all read requests, delivering sequential consistency by always serving reads locally with past committed values at any server; this represents the ideal upper bound for Bodega.

ZooKeeper Modes. Similarly, we deploy ZooKeeper in two modes, though both are non-linearizable. The default mode is a standard implementation of the ZAB primary-backup protocol [52] that pushes writes to all servers and serves reads locally from anywhere. The *sync* mode is the closest mode to linearizable reads that ZooKeeper clients can get: every read request is preceded by a `sync` API call to force flush all the in-progress writes from the leader to its endpoint server, but all writes that may have completed after the start of the flush are not guaranteed to be seen by the read.

Results. We present the performance results in Figure 19, grouped by workload type and with PQR (+ Ldr Ls) as the normalized throughput baseline. We make the following observations. ❶ Bodega matches (and sometimes surpasses) the performance of sequentially-consistent default ZooKeeper, and is able to keep up with stale etcd across all workloads. This illustrates Bodega's powerful local linearizable read capabilities. The advantage over ZK is due to avoiding Java runtime overheads. ❷ In workload C, both non-linearizable services deliver ~0.3 ms read latency and over 10x throughput gain, while Bodega and Quorum Leases deliver ~1.2 ms latency due to the 1 ms request batching applied. ❸ EPaxos, PQR (+ Ldr Ls), Quorum Leases, and Bodega all show similar patterns coherent to §6.1. With no writes (C), both local read protocols deliver excellent performance. With higher write ratios, Bodega sustains this advantage better than Quorum Leases. ❹ Default etcd and sync ZK have high read latencies of >50 ms because they are classic consensus without leases. ❺ Comparing the Uniform scenario with Zipfian, the only notable difference is that Bodega exhibits higher write latencies close to ZK in Uniform. This is expected because Bodega writes need to reach all nodes as they are all responders.

7 Discussion

We discuss topics that are out of the scope of this paper but are interesting directions for future work. We also clarify non-goals of Bodega.

Partial Network Partitioning. Using heartbeat timeout-based failure detection for leader step-up is known to risk liveness under partial network partitioning [86], and the same holds true for roster lease activations. Common techniques such as pre-votes [86] and transparent re-routing [2] can be deployed to easily eliminate this issue.

Generalization of Roster Leases. We observe that the activation procedure of roster leases shares similarities with *broadcast-based* (randomized) consensus [16, 83, 89]. It is a practical application of all-to-all broadcast in a non-adversarial setting for one-off agreements on the roster ballot. Combined with leases, this technique can be used to establish fault-tolerant agreement on any general "roster metadata" that change infrequently, not limited to leadership and assignment of responders as in Bodega. Possible extensions may include cluster membership, asymmetric quorum sizes, and node-specific performance and reliability hints.

Bounded Staleness Support. With simple modifications, Bodega can extend beyond linearizability and support fast local reads that can tolerate (but require) bounded staleness measured in the maximum version difference with the latest committed write. When a non-leader responder receives a read that allows up to x versions stale, it can traverse the tail of its log in reverse and search for at most x occurrences of the key, returning the latest committed version if found.

Non-Goals of Bodega. Bodega is primarily designed for wide-area linearizable read workloads; its improvements become less pronounced in other consensus use cases outside of this regime. These include: ❶ intra-datacenter replication where locality isn't significant, ❷ write-dominant workloads (see Figure 13 showing diminishing throughput improvement at 50% writes), and ❸ weaker consistency models where arbitrarily stale reads or inconsistent reads are tolerable. They may still leverage Bodega's roster leases as a mechanism to share cluster-wise information at negligible overhead (see the paragraph on generalization above), but would not observe the most immediate gains from localized linearizable reads.

8 Related Work

We list additional notable related work in this section.

Consensus & Read Optimizations. §2.3 and §4.1 have covered in detail the most essential related work, including classic consensus algorithms [60, 61, 65, 85–87], leaderless or multi-leader approaches [1, 27, 34, 56, 62, 63, 76, 80, 98, 104, 106], and read leases [10, 25, 41, 81, 82, 105].

Flexible quorum sizes have been discussed in classic literature [46] as well as in recent proposals such as Flexible Paxos [42, 48, 84, 111], where quorum shapes other than majority are explored to establish asymmetric performance between commands; Bodega is a modern manifestation of fault-tolerant, asymmetric read/write quorums leveraging leases. Optimistic holding shares similarity to wait-vs.-abort in database concurrency control [44, 58] and spin-then-park mutex locks [55]; we wait when waiting is expected to be faster.

Shared Logs & Lazy Ordering for Writes. Shared logs are a common abstraction found in cloud systems and are usually backed by primary-backup-style protocols [11–14, 20, 31, 73]. CAD [38], Skyros [39], and LazyLog [74] are a series of work on a *lazy ordering* optimization for writes and shared log appends. It hides a significant portion of write latency but could hurt read performance in contended cases.

Synchronized Clocks. Recent works demonstrate production-ready implementations of synchronized clocks [30, 70] and designs that take advantage of them through timestamp heuristics [32, 40]. Chandra et al. presented a formal, optimal lease algorithm that assumes synchronized clocks [18, 26].

Weaker Consistency Models. While this paper focuses on linearizable solutions, services may choose to provide weaker consistency models, which are less intuitive to reason about and harder to program against, but usually offer better performance. Common choices include sequential consistency that allows time-traveling reads [6, 45, 59], causal consistency that focuses on client session-local properties [9, 72, 78], and eventual consistency with bounded staleness [92, 109]. There are also studies that explore supporting a mixture of consistency models in one system [75, 113].

9 Conclusion

We present Bodega, a wide-area consensus protocol that enables localized linearizable reads at anywhere, anytime. Bodega achieves this via all-to-all roster leases off the critical path to establish responder assignment without compromising fault tolerance, delivering extreme read performance comparable to sequentially-consistent production systems. We believe Bodega is a valuable step towards performance-optimal wide-area replication for critical workloads of the modern cloud.

References

[1] Ailidani Ailijiang, Aleksey Charapko, Murat Demirbas, and Tevfik Kosar. 2020. WPaxos: Wide Area Network Flexible Consensus. *IEEE Trans. Parallel Distrib. Syst.* 31, 1 (Jan. 2020), 211–223. doi:10.1109/TPDS.2019.2929793

[2] Mohammed Alfatafta, Basil Alkhatib, Ahmed Alquraan, and Samer Al-Kiswany. 2020. Toward a Generic Fault Tolerance Technique for Partial Network Partitioning. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association, 351–368. https://www.usenix.org/conference/osdi20/presentation/alfatafta

[3] Amazon Web Services, Inc. 2021. *Availability and Beyond: Understanding and Improving the Resilience of Distributed Systems on AWS*. Whitepaper AWS Whitepaper. Amazon Web Services, Inc. https://docs.aws.amazon.com/pdfs/whitepapers/latest/availability-and-beyond-improving-resilience/availability-and-beyond-improving-resilience.pdf Accessed: 2025-12-10.

[4] Artem on StackOverflow. 2017. Is ZooKeeper always consistent in terms of CAP theorem? https://stackoverflow.com/questions/35387774. Accessed: 2024-12-01.

[5] Balaji Arun, Sebastiano Peluso, Roberto Palmieri, Giuliano Losa, and Binoy Ravindran. 2017. Speeding up Consensus by Chasing Fast Decisions. In *47th IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE, 49–60. doi:10.1109/DSN.2017.35

[6] Hagit Attiya and Jennifer L. Welch. 1994. Sequential Consistency versus Linearizability. *ACM Trans. Comput. Syst.* 12, 2 (may 1994), 91–122. doi:10.1145/176575.176576

[7] AWS. 2024. AWS Global Infrastructure. https://aws.amazon.com/about-aws/global-infrastructure/, Last accessed on 2024-04-28.

[8] AWS. 2024. Workload Characteristics. https://docs.aws.amazon.com/prescriptive-guidance/latest/oracle-exadata-blueprint/workload-characteristics.html. Accessed: 2024-12-01.

[9] Peter Bailis, Ali Ghodsi, Joseph M. Hellerstein, and Ion Stoica. 2013. Bolt-on causal consistency. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data* (New York, New York, USA) *(SIGMOD '13)*. Association for Computing Machinery, New York, NY, USA, 761–772. doi:10.1145/2463676.2465279

[10] Jason Baker, Chris Bond, James C. Corbett, JJ Furman, Andrey Khorlin, James Larson, Jean-Michel Leon, Yawei Li, Alexander Lloyd, and Vadim Yushprakh. 2011. Megastore: Providing Scalable, Highly Available Storage for Interactive Services. In *Proceedings of the Conference on Innovative Data system Research (CIDR)*. 223–234. http://www.cidrdb.org/cidr2011/Papers/CIDR11_Paper32.pdf

[11] Mahesh Balakrishnan, Jason Flinn, Chen Shen, Mihir Dharamshi, Ahmed Jafri, Xiao Shi, Santosh Ghosh, Hazem Hassan, Aaryaman Sagar, Rhed Shi, Jingming Liu, Filip Gruszczynski, Xianan Zhang, Huy Hoang, Ahmed Yossef, Francois Richard, and Yee Jiun Song. 2020. Virtual Consensus in Delos. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association, 617–632. https://www.usenix.org/conference/osdi20/presentation/balakrishnan

[12] Mahesh Balakrishnan, Dahlia Malkhi, John D. Davis, Vijayan Prabhakaran, Michael Wei, and Ted Wobber. 2013. CORFU: A distributed shared log. *ACM Trans. Comput. Syst.* 31, 4, Article 10 (Dec. 2013), 24 pages. doi:10.1145/2535930

[13] Mahesh Balakrishnan, Dahlia Malkhi, Ted Wobber, Ming Wu, Vijayan Prabhakaran, Michael Wei, John D. Davis, Sriram Rao, Tao Zou, and Aviad Zuck. 2013. Tango: Distributed Data Structures over a Shared Log. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles* (Farminton, Pennsylvania) *(SOSP '13)*. Association for Computing Machinery, New York, NY, USA, 325–340. doi:10.1145/2517349.2522732

[14] Mahesh Balakrishnan, Chen Shen, Ahmed Jafri, Suyog Mapara, David Geraghty, Jason Flinn, Vidhya Venkat, Ivailo Nedelchev, Santosh Ghosh, Mihir Dharamshi, Jingming Liu, Filip Gruszczynski, Jun Li, Rounak Tibrewal, Ali Zaveri, Rajeev Nagar, Ahmed Yossef, Francois Richard, and Yee Jiun Song. 2021. Log-structured Protocols in Delos. In *Proceedings of the ACM 28th Symposium on Operating Systems Principles* (Germany) *(SOSP '21)*. Association for Computing Machinery, New York, NY, USA, 538–552. doi:10.1145/3477132.3483544

[15] Jeff Barr. 2023. Amazon S3 Update – Strong Read-After-Write Consistency. https://aws.amazon.com/blogs/aws/amazon-s3-update-strong-read-after-write-consistency/, Last accessed on 2023-11-19.

[16] Michael Ben-Or. 1983. Another advantage of free choice (Extended Abstract): Completely asynchronous agreement protocols. In *Proceedings of the Second Annual ACM Symposium on Principles of Distributed Computing* (Montreal, Quebec, Canada) *(PODC '83)*. Association for Computing Machinery, New York, NY, USA, 27–30. doi:10.1145/800221.806707

[17] Hal Berenson, Phil Bernstein, Jim Gray, Jim Melton, Elizabeth O'Neil, and Patrick O'Neil. 1995. A critique of ANSI SQL isolation levels. In *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data* (San Jose, California, USA) *(SIGMOD '95)*. Association for Computing Machinery, New York, NY, USA, 1–10. doi:10.1145/223784.223785

[18] Changyu Bi, Vassos Hadzilacos, and Sam Toueg. 2022. Parameterized algorithm for replicated objects with local reads. arXiv:2204.01228 [cs.DC] https://arxiv.org/abs/2204.01228

[19] Marc Brooker, Tao Chen, and Fan Ping. 2020. Millions of tiny databases. In *NSDI 2020*. https://www.amazon.science/publications/millions-of-tiny-databases

[20] Matthew Burke, Audrey Cheng, and Wyatt Lloyd. 2020. Gryff: Unifying Consensus and Shared Registers. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*. USENIX Association, Santa Clara, CA, 591–617. https://www.usenix.org/conference/nsdi20/presentation/burke

[21] Mike Burrows. 2006. The Chubby Lock Service for Loosely-Coupled Distributed Systems. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation* (Seattle, Washington) *(OSDI '06)*. USENIX Association, USA, 335–350.

[22] Georgia Butler. 2024. Google Cloud accidentally deleted UniSuper's Private Cloud Subscription. *Data Center Dynamics* (2024). https://www.datacenterdynamics.com/en/news/google-cloud-accidentally-deleted-unisupers-private-cloud-subscription/

[23] Miguel Castro and Barbara Liskov. 2002. Practical byzantine fault tolerance and proactive recovery. *ACM Trans. Comput. Syst.* 20, 4 (Nov. 2002), 398–461. doi:10.1145/571637.571640

[24] Data Centre. 2024. Alibaba Cloud hit by Digital Realty fire in Singapore. *Frontier Enterprise* (2024). https://www.frontier-enterprise.com/alibaba-cloud-hit-by-digital-realty-fire-in-singapore/

[25] Tushar D. Chandra, Robert Griesemer, and Joshua Redstone. 2007. Paxos Made Live: An Engineering Perspective. In *Proceedings of the 26th ACM Symposium on Principles of Distributed Computing* (Portland, OR, USA) *(PODC '07)*. Association for Computing Machinery, New York, NY, USA, 398–407. doi:10.1145/1281100.1281103

[26] Tushar D. Chandra, Vassos Hadzilacos, and Sam Toueg. 2016. An Algorithm for Replicated Objects with Efficient Reads. In *Proceedings of the 2016 ACM Symposium on Principles of Distributed Computing* (Chicago, Illinois, USA) *(PODC '16)*. Association for Computing Machinery, New York, NY, USA, 325–334. doi:10.1145/2933057.2933111

[27] Aleksey Charapko, Ailidani Ailijiang, and Murat Demirbas. 2019. Linearizable Quorum Reads in Paxos. In *11th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 19)*. USENIX Association, Renton, WA. https://www.usenix.org/conference/hotstorage19/presentation/charapko

[28] Google Cloud. 2024. Google Cloud locations. https://cloud.google.com/about/locations/, Last accessed on 2024-11-30.

[29] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking Cloud Serving Systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing* (Indianapolis, IN, USA) *(SoCC '10)*. Association for Computing Machinery, New York, NY, USA, 143–154. doi:10.1145/1807128.1807152

[30] James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, J. J. Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, Wilson Hsieh, Sebastian Kanthak, Eugene Kogan, Hongyi Li, Alexander Lloyd, Sergey Melnik, David Mwaura, David Nagle, Sean Quinlan, Rajesh Rao, Lindsay Rolig, Yasushi Saito, Michal Szymaniak, Christopher Taylor, Ruth Wang, and Dale Woodford. 2013. Spanner: Google's Globally Distributed Database. *ACM Trans. Comput. Syst.* 31, 3, Article 8 (aug 2013), 22 pages. doi:10.1145/2491245

[31] Cong Ding, David Chu, Evan Zhao, Xiang Li, Lorenzo Alvisi, and Robbert Van Renesse. 2020. Scalog: Seamless Reconfiguration and Total Order in a Scalable Shared Log. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*. USENIX Association, Santa Clara, CA, 325–338. https://www.usenix.org/conference/nsdi20/presentation/ding

[32] Jiaqing Du, Daniele Sciascia, Sameh Elnikety, Willy Zwaenepoel, and Fernando Pedone. 2014. Clock-RSM: Low-Latency Inter-datacenter State Machine Replication Using Loosely Synchronized Physical Clocks. In *2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*. IEEE, 343–354. doi:10.1109/DSN.2014.42

[33] Dmitry Duplyakin, Robert Ricci, Aleksander Maricq, Gary Wong, Jonathon Duerig, Eric Eide, Leigh Stoller, Mike Hibler, David Johnson, Kirk Webb, Aditya Akella, Kuangching Wang, Glenn Ricart, Larry Landweber, Chip Elliott, Michael Zink, Emmanuel Cecchet, Snigdhaswin Kar, and Prabodh Mishra. 2019. The Design and Operation of CloudLab. In *Proceedings of the USENIX Annual Technical Conference*. 1–14. https://www.flux.utah.edu/paper/duplyakin-atc19

[34] Vitor Enes, Carlos Baquero, Tuanir França Rezende, Alexey Gotsman, Matthieu Perrin, and Pierre Sutra. 2020. State-Machine Replication for Planet-Scale Systems. In *Proceedings of the Fifteenth European Conference on Computer Systems* (Heraklion, Greece) *(EuroSys '20)*. Association for Computing Machinery, New York, NY, USA, Article 24, 15 pages. doi:10.1145/3342195.3387543

[35] etcd. 2023. etcd: A distributed, reliable key-value store for the most critical data. https://etcd.io/, Last accessed on 2023-11-13.

[36] FireScroll. 2023. FireScroll: The config database to deploy everywhere. https://github.com/FireScroll/FireScroll, Last accessed on 2024-09-05.

[37] Pedro Fouto, Nuno Preguiça, and Joao Leitão. 2022. High Throughput Replication with Integrated Membership Management. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*. USENIX Association, Carlsbad, CA, 575–592. https://www.usenix.org/conference/atc22/presentation/fouto

[38] Aishwarya Ganesan, Ramnatthan Alagappan, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. 2020. Strong and Efficient Consistency with Consistency-Aware Durability. In *18th USENIX Conference on File and Storage Technologies (FAST 20)*. USENIX Association, Santa Clara, CA, 323–337. https://www.usenix.org/conference/fast20/presentation/ganesan

[39] Aishwarya Ganesan, Ramnatthan Alagappan, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2021. Exploiting Nil-Externality for Fast Replicated Storage. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles* (Virtual Event, Germany) *(SOSP '21)*. Association for Computing Machinery, New York, NY, USA, 440–456. doi:10.1145/3477132.3483543

[40] Jinkun Geng, Anirudh Sivaraman, Balaji Prabhakar, and Mendel Rosenblum. 2022. Nezha: Deployable and High-Performance Consensus Using Synchronized Clocks. *Proc. VLDB Endow.* 16, 4 (dec 2022), 629–642. doi:10.14778/3574245.3574250

[41] C. Gray and D. Cheriton. 1989. Leases: an efficient fault-tolerant mechanism for distributed file cache consistency. In *Proceedings of the Twelfth ACM Symposium on Operating Systems Principles (SOSP '89)*. Association for Computing Machinery, New York, NY, USA, 202–210. doi:10.1145/74850.74870

[42] Joshua Guarnieri and Aleksey Charapko. 2023. Linearizable Low-latency Reads at the Edge. In *Proceedings of the 10th Workshop on Principles and Practice of Consistency for Distributed Data* (Rome, Italy) *(PaPoC '23)*. Association for Computing Machinery, New York, NY, USA, 77–83. doi:10.1145/3578358.3591327

[43] Haryadi S. Gunawi, Mingzhe Hao, Tanakorn Leesatapornwongsa, Tiratat Patana-anake, Thanh Do, Jeffry Adityatama, Kurnia J. Eliazar, Agung Laksono, Jeffrey F. Lukman, Vincentius Martin, and Anang D. Satria. 2014. What Bugs Live in the Cloud? A Study of 3000+ Issues in Cloud Systems. In *Proceedings of the ACM Symposium on Cloud Computing* (Seattle, WA, USA) *(SOCC '14)*. Association for Computing Machinery, New York, NY, USA, 1–14. doi:10.1145/2670979.2670986

[44] Rachael Harding, Dana Van Aken, Andrew Pavlo, and Michael Stonebraker. 2017. An evaluation of distributed concurrency control. *Proc. VLDB Endow.* 10, 5 (Jan 2017), 553–564. doi:10.14778/3055540.3055548

[45] Jeffrey Helt, Matthew Burke, Amit Levy, and Wyatt Lloyd. 2021. Regular Sequential Serializability and Regular Sequential Consistency. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles* (Virtual Event, Germany) *(SOSP '21)*. Association for Computing Machinery, New York, NY, USA, 163–179. doi:10.1145/3477132.3483566

[46] Maurice Herlihy. 1987. Dynamic quorum adjustment for partitioned data. *ACM Trans. Database Syst.* 12, 2 (Jun 1987), 170–194. doi:10.1145/22952.22953

[47] Maurice P. Herlihy and Jeannette M. Wing. 1990. Linearizability: A Correctness Condition for Concurrent Objects. *ACM Trans. Program. Lang. Syst.* 12, 3 (jul 1990), 463–492. doi:10.1145/78969.78972

[48] Heidi Howard, Dahlia Malkhi, and Alexander Spiegelman. 2016. Flexible Paxos: Quorum intersection revisited. arXiv:1608.06696 [cs.DC]

[49] Guanzhou Hu, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. 2024. A Unified, Practical, and Understandable Summary of Non-transactional Consistency Levels in Distributed Replication. arXiv:2409.01576 [cs.DC] https://arxiv.org/abs/2409.01576

[50] Dongxu Huang, Qi Liu, Qiu Cui, Zhuhe Fang, Xiaoyu Ma, Fei Xu, Li Shen, Liu Tang, Yuxing Zhou, Menglong Huang, Wan Wei, Cong Liu, Jian Zhang, Jianjun Li, Xuelian Wu, Lingyu Song, Ruoxi Sun, Shuaipeng Yu, Lei Zhao, Nicholas Cameron, Liquan Pei, and Xin Tang. 2020. TiDB: A Raft-Based HTAP Database. *Proc. VLDB Endow.* 13, 12 (aug 2020), 3072–3084. doi:10.14778/3415478.3415535

[51] Peng Huang, Chuanxiong Guo, Lidong Zhou, Jacob R. Lorch, Yingnong Dang, Murali Chintalapati, and Randolph Yao. 2017. Gray Failure: The Achilles' Heel of Cloud-Scale Systems. In *Proceedings of the 16th Workshop on Hot Topics in Operating Systems* (Whistler, BC, Canada) *(HotOS '17)*. Association for Computing Machinery, New York, NY, USA, 150–155. doi:10.1145/3102980.3103005

[52] Patrick Hunt, Mahadev Konar, Flavio P. Junqueira, and Benjamin Reed. 2010. ZooKeeper: Wait-Free Coordination for Internet-Scale Systems. In *Proceedings of the 2010 USENIX Conference on USENIX Annual Technical Conference* (Boston, MA) *(USENIXATC'10)*. USENIX Association, USA, 11.

[53] Randall Hunt. 2017. Keeping Time With Amazon Time Sync Service. https://aws.amazon.com/blogs/aws/keeping-time-with-amazon-time-sync-service/. Accessed: 2017-11-29.

[54] Jonathan Kaldor, Jonathan Mace, Michał Bejda, Edison Gao, Wiktor Kuropatwa, Joe O'Neill, Kian Win Ong, Bill Schaller, Pingjia Shan, Brendan Viscomi, Vinod Venkataraman, Kaushik Veeraraghavan, and Yee Jiun Song. 2017. Canopy: An End-to-End Performance Tracing And Analysis System. In *Proceedings of the 26th Symposium on Operating Systems Principles* (Shanghai, China) *(SOSP '17)*. Association for Computing Machinery, New York, NY, USA, 34–50. doi:10.1145/3132747.3132749

[55] Anna R. Karlin, Kai Li, Mark S. Manasse, and Susan Owicki. 1991. Empirical studies of competitve spinning for a shared-memory multiprocessor. In *Proceedings of the Thirteenth ACM Symposium on Operating Systems Principles* (Pacific Grove, California, USA) *(SOSP '91)*. Association for Computing Machinery, New York, NY, USA, 41–55. doi:10.1145/121132.286599

[56] Antonios Katsarakis, Vasilis Gavrielatos, M.R. Siavash Katebzadeh, Arpit Joshi, Aleksandar Dragojevic, Boris Grot, and Vijay Nagarajan. 2020. Hermes: A Fast, Fault-Tolerant and Linearizable Replication Protocol. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems* (Lausanne, Switzerland) *(ASPLOS '20)*. Association for Computing Machinery, New York, NY, USA, 201–217. doi:10.1145/3373376.3378496

[57] KRaft. 2025. KRaft: Apache Kafka Without ZooKeeper. https://developer.confluent.io/learn/kraft/, Last accessed on 2025-04-12.

[58] H. T. Kung and John T. Robinson. 1981. On optimistic methods for concurrency control. *ACM Trans. Database Syst.* 6, 2 (jun 1981), 213–226. doi:10.1145/319566.319567

[59] Leslie Lamport. 1979. How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs. *IEEE Transactions on Computers C-28* 9, 9 (September 1979), 690–691. doi:10.1109/TC.1979.1675439

[60] Leslie Lamport. 1998. The Part-Time Parliament. *ACM Trans. Comput. Syst.* 16, 2 (may 1998), 133–169. doi:10.1145/279227.279229

[61] Leslie Lamport. 2001. Paxos Made Simple. *ACM SIGACT News (Distributed Computing Column) 32, 4 (Whole Number 121, December 2001)* (December 2001), 51–58. https://www.microsoft.com/en-us/research/publication/paxos-made-simple/

[62] Leslie Lamport. 2005. Generalized consensus and Paxos. *Microsoft Research Technical Report* (2005).

[63] Leslie Lamport. 2006. Fast Paxos. *Distrib. Comput.* 19, 2 (oct 2006), 79–103. doi:10.1007/s00446-006-0005-x

[64] Leslie Lamport, Dahlia Malkhi, and Lidong Zhou. 2009. Vertical paxos and primary-backup replication. In *Proceedings of the 28th ACM Symposium on Principles of Distributed Computing* (Calgary, AB, Canada) *(PODC '09)*. Association for Computing Machinery, New York, NY, USA, 312–313. doi:10.1145/1582716.1582783

[65] Butler Lampson. 2001. The ABCD's of Paxos. In *Proceedings of the Twentieth Annual ACM Symposium on Principles of Distributed Computing* (Newport, RI, USA) *(PODC '01)*. Association for Computing Machinery, New York, NY, USA, 13. doi:10.1145/383962.383969

[66] Collin Lee, Seo Jin Park, Ankita Kejriwal, Satoshi Matsushita, and John Ousterhout. 2015. Implementing Linearizability at Large Scale and Low Latency. In *Proceedings of the 25th Symposium on Operating Systems Principles* (CA) *(SOSP '15)*. Association for Computing Machinery, New York, NY, USA, 71–86. doi:10.1145/2815400.2815416

[67] Edward K. F. Lee and Chandramohan A. Thekkath. 1996. Petal: distributed virtual disks. In *ASPLOS VII (ASPLOS96)*. ACM, 84–92. doi:10.1145/237090.237157

[68] Ki Suh Lee, Han Wang, Vishal Shrivastav, and Hakim Weatherspoon. 2016. Globally Synchronized Time via Datacenter Networks. In *Proceedings of the 2016 ACM SIGCOMM Conference* (Florianopolis, Brazil) *(SIGCOMM '16)*. Association for Computing Machinery, New York, NY, USA, 454–467. doi:10.1145/2934872.2934885

[69] Jialin Li, Ellis Michael, Naveen Kr. Sharma, Adriana Szekeres, and Dan R. K. Ports. 2016. Just Say NO to Paxos Overhead: Replacing Consensus with Network Ordering. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. USENIX Association, Savannah, GA, 467–483. https://www.usenix.org/conference/osdi16/technical-sessions/presentation/li

[70] Yuliang Li, Gautam Kumar, Hema Hariharan, Hassan Wassel, Peter Hochschild, Dave Platt, Simon Sabato, Minlan Yu, Nandita Dukkipati, Prashant Chandra, and Amin Vahdat. 2020. Sundial: Fault-tolerant Clock Synchronization for Datacenters. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association, 1171–1186. https://www.usenix.org/conference/osdi20/presentation/li-yuliang

[71] Linux man pages. 2011. tc-netem(8) — Linux manual page. https://man7.org/linux/man-pages/man8/tc-netem.8.html. [Online; accessed 29-November-2023].

[72] Wyatt Lloyd, Michael J. Freedman, Michael Kaminsky, and David G. Andersen. 2011. Don't settle for eventual: scalable causal consistency for wide-area storage with COPS. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles* (Cascais, Portugal) *(SOSP '11)*. Association for Computing Machinery, New York, NY, USA, 401–416. doi:10.1145/2043556.2043593

[73] Joshua Lockerman, Jose M. Faleiro, Juno Kim, Soham Sankaran, Daniel J. Abadi, James Aspnes, Siddhartha Sen, and Mahesh Balakrishnan. 2018. The FuzzyLog: A Partially Ordered Shared Log. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. USENIX Association, Carlsbad, CA, 357–372. https://www.usenix.org/conference/osdi18/presentation/lockerman

[74] Xuhao Luo, Shreesha G. Bhat, Jiyu Hu, Ramnatthan Alagappan, and Aishwarya Ganesan. 2024. LazyLog: A New Shared Log Abstraction for Low-Latency Applications. In *Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles* (Austin, TX, USA) *(SOSP '24)*. Association for Computing Machinery, New York, NY, USA, 296–312. doi:10.1145/3694715.3695983

[75] Kai Ma, Cheng Li, Enzuo Zhu, Ruichuan Chen, Feng Yan, and Kang Chen. 2024. Noctua: Towards Automated and Practical Fine-grained Consistency Analysis. In *Proceedings of the Nineteenth European Conference on Computer Systems* (Athens, Greece) *(EuroSys '24)*. Association for Computing Machinery, New York, NY, USA, 704–719. doi:10.1145/3627703.3629570

[76] Yanhua Mao, Flavio P. Junqueira, and Keith Marzullo. 2008. Mencius: Building Efficient Replicated State Machines for WANs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation* (San Diego, California) *(OSDI'08)*. USENIX Association, USA, 369–384.

[77] Jim Martin, Jack Burbank, William Kasch, and Professor David L. Mills. 2010. Network Time Protocol Version 4: Protocol and Algorithms Specification. RFC 5905. doi:10.17487/RFC5905

[78] Syed Akbar Mehdi, Cody Littley, Natacha Crooks, Lorenzo Alvisi, Nathan Bronson, and Wyatt Lloyd. 2017. I Can't Believe It's Not Causal! Scalable Causal Consistency with No Slowdown Cascades. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*. USENIX Association, Boston, MA, 453–468. https://www.usenix.org/conference/nsdi17/technical-sessions/presentation/mehdi

[79] Microsoft. 2024. Azure Global Infrastructure. https://datacenters.microsoft.com/globe/explore/, Last accessed on 2024-11-30.

[80] Iulian Moraru, David G. Andersen, and Michael Kaminsky. 2013. There is More Consensus in Egalitarian Parliaments. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles* (Farminton, Pennsylvania) *(SOSP '13)*. Association for Computing Machinery, New York, NY, USA, 358–372. doi:10.1145/2517349.2517350

[81] Iulian Moraru, David G. Andersen, and Michael Kaminsky. 2014. Paxos Quorum Leases: Fast Reads Without Sacrificing Writes. In *Proceedings of the ACM Symposium on Cloud Computing* (Seattle, WA, USA) *(SOCC '14)*. Association for Computing Machinery, New York, NY, USA, 1–13. doi:10.1145/2670979.2671001

[82] Iulian Moraru, David G. Andersen, and Michael Kaminsky. 2014. Paxos Quorum Leases: Fast Reads Without Sacrificing Writes. *Carnegie Mellon University PDL Technical Report* (2014).

[83] Achour Mostefaoui, Matthieu Perrin, and Julien Weibel. 2024. Brief Announcement: Randomized Consensus: Common Coins Are not the Holy Grail!. In *Proceedings of the 43rd ACM Symposium on Principles of Distributed Computing* (Nantes, France) *(PODC '24)*. Association for Computing Machinery, New York, NY, USA, 36–39. doi:10.1145/3662158.3662824

[84] Faisal Nawab, Divyakant Agrawal, and Amr El Abbadi. 2018. DPaxos: Managing Data Closer to Users for Low-Latency and Mobile Applications. In *Proceedings of the 2018 International Conference on Management of Data* (Houston, TX, USA) *(SIGMOD '18)*. Association for Computing Machinery, New York, NY, USA, 1221–1236. doi:10.1145/3183713.3196928

[85] Brian M. Oki and Barbara H. Liskov. 1988. Viewstamped Replication: A New Primary Copy Method to Support Highly-Available Distributed Systems. In *Proceedings of the Seventh Annual ACM Symposium on Principles of Distributed Computing* (Toronto, Ontario, Canada) *(PODC '88)*. Association for Computing Machinery, New York, NY, USA, 8–17. doi:10.1145/62546.62549

[86] Diego Ongaro. 2014. *Consensus: Bridging Theory and Practice*. Ph. D. Dissertation. Stanford, CA, USA. Advisor(s) K., Ousterhout, John and David, Mazières, and Mendel, Rosenblum,. AAI28121474.

[87] Diego Ongaro and John Ousterhout. 2014. In Search of an Understandable Consensus Algorithm. In *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference* (Philadelphia, PA) *(USENIX ATC'14)*. USENIX Association, USA, 305–320.

[88] Team Live Optics. 2021. Live Optics Basics: Read / Write Ratio. https://support.liveoptics.com/hc/en-us/articles/229590547-Live-Optics-Basics-Read-Write-Ratio. Accessed: 2024-12-01.

[89] Haochen Pan, Jesse Tuglu, Neo Zhou, Tianshu Wang, Yicheng Shen, Xiong Zheng, Joseph Tassarotti, Lewis Tseng, and Roberto Palmieri. 2021. Rabia: Simplifying State-Machine Replication Through Randomization. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles* (Virtual Event, Germany) *(SOSP '21)*. Association for Computing Machinery, New York, NY, USA, 472–487. doi:10.1145/3477132.3483582

[90] Seo Jin Park and John Ousterhout. 2019. Exploiting Commutativity For Practical Fast Replication. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*. USENIX Association, Boston, MA, 47–64. https://www.usenix.org/conference/nsdi19/presentation/park

[91] Suraj Pasuparthy and Lokesh Agarwal. 2023. Benchmarking Spanner's price-performance for key-value workloads. https://cloud.google.com/blog/products/databases/benchmarking-spanner-for-key-value-workloads/. Accessed: 2024-12-01.

[92] Karin Petersen, Mike J. Spreitzer, Douglas B. Terry, Marvin M. Theimer, and Alan J. Demers. 1997. Flexible update propagation for weakly consistent replication. In *Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles* (Saint Malo, France) *(SOSP '97)*. Association for Computing Machinery, New York, NY, USA, 288–301. doi:10.1145/268998.266711

[93] Dan R. K. Ports, Jialin Li, Vincent Liu, Naveen Kr. Sharma, and Arvind Krishnamurthy. 2015. Designing Distributed Systems Using Approximate Synchrony in Data Center Networks. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*. USENIX Association, Oakland, CA, 43–57. https://www.usenix.org/conference/nsdi15/technical-sessions/presentation/ports

[94] RabbitMQ. 2025. RabbitMQ: One broker to queue them all. https://www.rabbitmq.com/, Last accessed on 2025-04-08.

[95] Redpanda. 2024. Redpanda: The Unified Streaming Data Platform. https://www.redpanda.com/, Last accessed on 2024-09-05.

[96] Robbert Van Renesse and Fred B. Schneider. 2004. Chain Replication for Supporting High Throughput and Availability. In *6th Symposium on Operating Systems Design & Implementation (OSDI 04)*. USENIX Association, CA. https://www.usenix.org/conference/osdi-04/chain-replication-supporting-high-throughput-and-availability

[97] David K. Rensin. 2015. *Kubernetes - Scheduling the Future at Cloud Scale*. O'Reilly and Associates, 1005 Gravenstein Highway North Sebastopol, CA 95472. All pages. http://www.oreilly.com/webops-perf/free/kubernetes.csp

[98] Fedor Ryabinin, Alexey Gotsman, and Pierre Sutra. 2024. SwiftPaxos: Fast Geo-Replicated State Machines. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*. USENIX Association, Santa Clara, CA, 345–369. https://www.usenix.org/conference/nsdi24/presentation/ryabinin

[99] Fred B. Schneider. 1990. Implementing Fault-Tolerant Services Using the State Machine Approach: A Tutorial. *ACM Comput. Surv.* 22, 4 (dec 1990), 299–319. doi:10.1145/98163.98167

[100] ScyllaDB. 2023. Beyond Legacy NoSQL: 7 Design Principles Behind ScyllaDB. https://lp.scylladb.com/real-time-big-data-database-principles-thanks.html, Last accessed on 2023-11-13.

[101] Hatem Takruri, Ibrahim Kettaneh, Ahmed Alquraan, and Samer Al-Kiswany. 2020. FLAIR: Accelerating Reads with Consistency-Aware Network Routing. In *17th USENIX Symposium on Networked Systems Design & Implementation (NSDI 20)*. USENIX Association, CA, 723–737. https://usenix.org/conference/nsdi20/presentation/takruri

[102] Douglas B. Terry, Alan J. Demers, Karin Petersen, Mike J. Spreitzer, Marvin M. Theimer, and Brent B. Welch. 1994. Session Guarantees for Weakly Consistent Replicated Data. In *Proceedings of the Third International Conference on on Parallel and Distributed Information Systems* (Autin, Texas, USA) *(PDIS '94)*. IEEE Computer Society Press, Washington, DC, USA, 140–150. doi:10.1109/PDIS.1994.331722

[103] TigerBeetle. 2024. TigerBeetle: The Financial Transactions Database. https://tigerbeetle.com/, Last accessed on 2024-11-12.

[104] Sarah Tollman, Seo Jin Park, and John Ousterhout. 2021. EPaxos Revisited. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*. USENIX Association, 613–632. https://www.usenix.org/conference/nsdi21/presentation/tollman

[105] Bohdan Trach, Rasha Faqeh, Oleksii Oleksenko, Wojciech Ozga, Pramod Bhatotia, and Christof Fetzer. 2020. T-Lease: a trusted lease primitive for distributed systems. In *Proceedings of the 11th ACM Symposium on Cloud Computing* (Virtual Event, USA) *(SoCC '20)*. Association for Computing Machinery, New York, NY, USA, 387–400. doi:10.1145/3419111.3421273

[106] Muhammed Uluyol, Anthony Huang, Ayush Goel, Mosharaf Chowdhury, and Harsha V. Madhyastha. 2020. Near-Optimal Latency Versus Cost Tradeoffs in Geo-Distributed Storage. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*. USENIX Association, Santa Clara, CA, 157–180. https://www.usenix.org/conference/nsdi20/presentation/uluyol

[107] Nathan VanBenschoten, Arul Ajmani, Marcus Gartner, Andrei Matei, Aayush Shah, Irfan Sharif, Alexander Shraer, Adam Storm, Rebecca Taft, Oliver Tan, Andy Woods, and Peyton Walters. 2022. Enabling the Next Generation of Multi-Region Applications with CockroachDB. In *Proceedings of the 2022 International Conference on Management of Data* (Philadelphia, PA, USA) *(SIGMOD '22)*. Association for Computing Machinery, New York, NY, USA, 2312–2325. doi:10.1145/3514221.3526053

[108] Kaushik Veeraraghavan, Justin Meza, Scott Michelson, Sankaralingam Panneerselvam, Alex Gyori, David Chou, Sonia Margulis, Daniel Obenshain, Shruti Padmanabha, Ashish Shah, Yee Jiun Song, and Tianyin Xu. 2018. Maelstrom: Mitigating Datacenter-level Disasters by Draining Interdependent Traffic Safely and Efficiently. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. USENIX Association, Carlsbad, CA, 373–389. https://www.usenix.org/conference/osdi18/presentation/veeraraghavan

[109] Werner Vogels. 2008. Eventually Consistent: Building Reliable Distributed Systems at a Worldwide Scale Demands Trade-Offs Between Consistency and Availability. *Queue* 6, 6 (oct 2008), 14–19. doi:10.1145/1466443.1466448

[110] Zhaoguo Wang, Changgeng Zhao, Shuai Mu, Haibo Chen, and Jinyang Li. 2019. On the Parallels between Paxos and Raft, and how to Port Optimizations. In *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing* (Toronto ON, Canada) *(PODC '19)*. Association for Computing Machinery, New York, NY, USA, 445–454. doi:10.1145/3293611.3331595

[111] Michael Whittaker, Aleksey Charapko, Joseph M. Hellerstein, Heidi Howard, and Ion Stoica. 2021. Read-Write Quorum Systems Made Practical. In *Proceedings of the 8th Workshop on Principles and Practice of Consistency for Distributed Data* (Online, United Kingdom) *(PaPoC '21)*. Association for Computing Machinery, New York, NY, USA, Article 7, 8 pages. doi:10.1145/3447865.3457962

[112] Jian Yi, Qing Li, Bin Zhang, Yong Jiang, Dan Zhao, Yuan Yang, and Zhenhui Yuan. 2023. Gleaning the Consensus for Linearizable and Conflict-Free Per-Replica Local Reads. In *Proceedings of the 7th Asia-Pacific Workshop on Networking* (Hong Kong, China) *(APNet '23)*. Association for Computing Machinery, New York, NY, USA, 143–149. doi:10.1145/3600061.3603175

[113] Haifeng Yu and Amin Vahdat. 2000. Design and evaluation of a continuous consistency model for replicated services. In *Proceedings*

of the 4th Conference on Symposium on Operating System Design & Implementation - Volume 4 (San Diego, California) *(OSDI'00)*. USENIX Association, USA, Article 21.

[114] Hanyu Zhao, Quanlu Zhang, Zhi Yang, Ming Wu, and Yafei Dai. 2018. SDPaxos: Building Efficient Semi-Decentralized Geo-replicated State Machines. In *Proceedings of the ACM Symposium on Cloud Computing* (Carlsbad, CA, USA) *(SoCC '18)*. Association for Computing Machinery, New York, NY, USA, 68–81. doi:10.1145/3267809.3267837

A Formal TLA$^+$ Specification of Bodega

We also present a formal specification of Bodega written as a PlusCal algorithm that can be auto-translated into TLA$^+$. The specification builds on top of MultiPaxos and introduces all-to-all roster lease actions to support local reads, faithful to the presentation in Figure 7. Please refer to the inlined comments for specification details.

The specification has been model-checked for both *linearizability* and *fault-tolerance* properties on symbolic inputs of 3 nodes (thus 1 allowed failure), 3 ballot numbers, 2 distinct write requests plus 2 distinct read requests, 3 lease time ticks, and all possible choices of responders. These inputs should be large enough to explore the distinct, interesting execution paths of the protocol. Model checking took 43 hours on a single machine with 96 cores and 768GiB memory, and generated 4,274,883,464 distinct states.

B Artifact Evaluation Instructions

Bodega's source code (as part of the Summerset key-value store) is publicly available on GitHub at https://github.com/josehu07/summerset/tree/bodega-artifact.

Artifact evaluation instructions can be found at https://github.com/josehu07/summerset/blob/bodega-artifact/publish/bodega/ARTIFACT.md. The document contains detailed instructions on setting up a distributed consensus system environment and reproducing all the figures presented in this paper.

MODULE *Bodega*

EXTENDS *FiniteSets*, *Sequences*, *Integers*, *TLC*

Model inputs & assumptions.

CONSTANT *Replicas*, *symmetric set of server nodes*
Writes, *symmetric set of write commands(each w/unique value)*
Reads, *symmetric set of read commands*
MaxBallot, *maximum ballot pickable for leader preemption*
TGuard, *lease guard phase window length(in abstract ticks)*
TLease, *lease renewal extend window length(in abstract ticks)*
MaxTime, *upper bound on abstract time for model checking*
NodeFailuresOn *if true, turn on node failures injection*

$$ReplicasAssumption \triangleq \land IsFiniteSet(Replicas) \land Cardinality(Replicas) \geq 1 \land \text{“none”} \notin Replicas$$

$$Population \triangleq Cardinality(Replicas)$$

$$MajorityNum \triangleq (Population \div 2) + 1$$

$$WritesAssumption \triangleq \land IsFiniteSet(Writes) \land Cardinality(Writes) \geq 1 \land \text{“nil”} \notin Writes$$

$$ReadsAssumption \triangleq \land IsFiniteSet(Reads) \land Cardinality(Reads) \geq 0 \land \text{“nil”} \notin Writes$$

$$MaxBallotAssumption \triangleq \land MaxBallot \in Nat \land MaxBallot \geq 2$$

$$TGuardAssumption \triangleq \land TGuard \in Nat \land TGuard \geq 1$$

$$TLeaseAssumption \triangleq \land TLease \in Nat \land TLease \geq 1$$

$$MaxTimeAssumption \triangleq \land MaxTime \in Nat \land MaxTime \geq TGuard + TLease$$

$$NodeFailuresOnAssumption \triangleq NodeFailuresOn \in \text{BOOLEAN}$$

ASSUME $\land ReplicasAssumption \land WritesAssumption \land ReadsAssumption \land MaxBallotAssumption \land TGuardAssumption \land TLeaseAssumption \land MaxTimeAssumption \land NodeFailuresOnAssumption$

Useful constants & typedefs.

$$Commands \triangleq Writes \cup Reads$$

$$NumWrites \triangleq Cardinality(Writes)$$

$NumReads \triangleq Cardinality(Reads)$

$NumCommands \triangleq Cardinality(Commands)$

$Range(seq) \triangleq \{seq[i] : i \in 1 \,..\, Len(seq)\}$

Client observable events.

$$\begin{aligned} ClientEvents \triangleq\ & [type : \{\text{“Req”}\},\ cmd : Commands] \\ \cup\ & [type : \{\text{“Ack”}\},\ cmd : Commands, \\ & \qquad val\ : \{\text{“nil”}\} \cup Writes, \\ & \qquad by : Replicas] \end{aligned}$$

$ReqEvent(c) \triangleq [type \mapsto \text{“Req”},\ cmd \mapsto c]$

$AckEvent(c, v, n) \triangleq [type \mapsto \text{“Ack”},\ cmd \mapsto c,\ val \mapsto v,\ by \mapsto n]$

val is the old value for a write command

$$\begin{aligned} InitPending \triangleq\ & (\text{CHOOSE } ws \in [1 \,..\, Cardinality(Writes) \to Writes] \\ & \qquad : Range(ws) = Writes) \\ \circ\ & (\text{CHOOSE } rs \in [1 \,..\, Cardinality(Reads) \to Reads] \\ & \qquad : Range(rs) = Reads) \end{aligned}$$

Server – side consensus constants & states.

$Ballots \triangleq 1 \,..\, MaxBallot$

$Slots \triangleq 1 \,..\, NumWrites$

$InfinitySlot \triangleq NumWrites + 1$

sentinel “infinitely-high” *slot, used as commitPrev s*

safe “haven’t learned yet” marker

$$\begin{aligned} Rosters \triangleq\ & \{ros \in [bal : Ballots, \\ & \qquad leader : Replicas, \\ & \qquad responders : \text{SUBSET } Replicas] : \\ & ros.leader \notin ros.responders\} \end{aligned}$$

for a smaller state space we exclude leader from the literal set of *responders* (but always treating it as a responder)

$Roster(b, l, resps) \triangleq [bal \mapsto b,\ leader \mapsto l,\ responders \mapsto resps]$

each new ballot number maps to a new roster; this includes the change of leader (as in classic *MultiPaxos*) and/or the change of who’re *responders*

$NullRoster \triangleq [bal \mapsto 0,\ leader \mapsto \text{“none”},\ responders \mapsto \{\}]$

$Statuses \triangleq \{\text{“Preparing”}, \text{“Accepting”}, \text{“Committed”}\}$

$$\begin{aligned} InstStates \triangleq\ & [status : \{\text{“Empty”}\} \cup Statuses, \\ & \quad write : \{\text{“nil”}\} \cup Writes, \\ & \quad voted : [bal : \{0\} \cup Ballots, \\ & \qquad write : \{\text{“nil”}\} \cup Writes]] \end{aligned}$$

$$\begin{aligned} NullInst \triangleq\ & [status \mapsto \text{“Empty”}, \\ & \quad write \mapsto \text{“nil”}, \\ & \quad voted \mapsto [bal \mapsto 0,\ write \mapsto \text{“nil”}]] \end{aligned}$$

Lease-side constants & typedefs.

$Times \triangleq 1 \,..\, MaxTime$

$ExpireTimes \triangleq 0 \,..\, (MaxTime + TGuard + TLease)$

stored expiration / guard deadlines; 0 is the "null" time

$SeqNums \triangleq Nat$

per-pair monotone *seq* nums on lease messages for dedup

$GrantorStatuses \triangleq \{\text{“None”}, \text{“Guarding”}, \text{“Renewing”}, \text{“Revoking”}\}$
$GranteeStatuses \triangleq \{\text{“None”}, \text{“Guarded”}, \text{“Renewed”}\}$

$$\begin{aligned} GrantorState \triangleq [&status : GrantorStatuses,\\ &guardExpire : ExpireTimes,\\ &leaseExpire\ \ : ExpireTimes,\\ &ros : \{NullRoster\} \cup Rosters,\\ &seq : SeqNums] \end{aligned}$$

$$\begin{aligned} GranteeState \triangleq [&status : GranteeStatuses,\\ &guardExpire : ExpireTimes,\\ &leaseExpire\ \ : ExpireTimes,\\ &ros : \{NullRoster\} \cup Rosters,\\ &seq : SeqNums] \end{aligned}$$

$$\begin{aligned} NullGrantorState \triangleq [&status \mapsto \text{“None”},\\ &guardExpire \mapsto 0,\\ &leaseExpire\ \ \mapsto 0,\\ &ros \mapsto NullRoster,\\ &seq \mapsto 0] \end{aligned}$$

$$\begin{aligned} NullGranteeState \triangleq [&status \mapsto \text{“None”},\\ &guardExpire \mapsto 0,\\ &leaseExpire\ \ \mapsto 0,\\ &ros \mapsto NullRoster,\\ &seq \mapsto 0] \end{aligned}$$

Merged per-node state: consensus *fields* + lease *fields*.

$$\begin{aligned} NodeStates \triangleq [&leader : \{\text{“none”}\} \cup Replicas,\\ &commitUpTo : \{0\} \cup Slots,\\ &commitPrev\ \ : \{0\} \cup Slots \cup \{InfinitySlot\},\\ &balPrepared\ \ \ \ : \{0\} \cup Ballots,\\ &balMaxKnown : \{0\} \cup Ballots,\\ &rosMaxKnown : \{NullRoster\} \cup Rosters,\\ &insts : [Slots \to InstStates],\\ &asGrantor : [Replicas \to GrantorState],\\ &asGrantee : [Replicas \to GranteeState]] \end{aligned}$$

$$\begin{aligned} NullNode \triangleq [&leader \mapsto \text{“none”},\\ &commitUpTo \mapsto 0,\\ &commitPrev\ \ \mapsto 0,\\ &balPrepared\ \ \ \ \mapsto 0,\\ &balMaxKnown \mapsto 0,\\ &rosMaxKnown \mapsto NullRoster,\\ &insts \mapsto [s \in Slots \mapsto NullInst],\\ &asGrantor \mapsto [p \in Replicas \mapsto NullGrantorState],\\ &asGrantee \mapsto [f \in Replicas \mapsto NullGranteeState]] \end{aligned}$$

commitPrev is the last slot which might have been committed by an old leader; a newly joined node can safely serve reads locally only after its log has been committed up to this slot

rosMaxKnown is the roster that came with *balMaxKnown*

$asGrantor[p]$ is the grantor-side lease state & timers toward grantee p; $asGrantee[f]$ is the grantee-side state for grantor f

$$FirstEmptySlot(insts) \triangleq$$
IF $\forall s \in Slots : insts[s].status \neq$ "Empty"
THEN $InfinitySlot$
ELSE CHOOSE $s \in Slots :$
$\wedge\, insts[s].status =$ "Empty"
$\wedge \forall t \in 1 \,..\, (s-1) : insts[t].status \neq$ "Empty"

$$LastNonEmptySlot(insts) \triangleq$$
IF $\forall s \in Slots : insts[s].status =$ "Empty"
THEN 0
ELSE CHOOSE $s \in Slots :$
$\wedge\, insts[s].status \neq$ "Empty"
$\wedge \forall t \in (s+1) \,..\, NumWrites : insts[t].status =$ "Empty"

Service-internal consensus messages.

$$PrepareMsgs \triangleq [type : \{\text{"Prepare"}\}, src : Replicas, bal : Ballots]$$

$$PrepareMsg(r, b) \triangleq [type \mapsto \text{"Prepare"}, src \mapsto r, bal \mapsto b]$$

$$InstsVotes \triangleq [Slots \to [bal : \{0\} \cup Ballots, write : \{\text{"nil"}\} \cup Writes]]$$

$$VotesByNode(n) \triangleq [s \in Slots \mapsto n.insts[s].voted]$$

$$PrepareReplyMsgs \triangleq [type : \{\text{"PrepareReply"}\}, src : Replicas, bal : Ballots, votes : InstsVotes]$$

$$PrepareReplyMsg(r, b, iv) \triangleq [type \mapsto \text{"PrepareReply"}, src \mapsto r, bal \mapsto b, votes \mapsto iv]$$

$$PeakVotedWrite(prs, s) \triangleq$$
IF $\forall pr \in prs : pr.votes[s].bal = 0$
THEN "nil"
ELSE LET $ppr \triangleq$
CHOOSE $ppr \in prs :$
$\forall pr \in prs : pr.votes[s].bal \leq ppr.votes[s].bal$
IN $ppr.votes[s].write$

$$LastTouchedSlot(prs) \triangleq$$
IF $\forall s \in Slots : PeakVotedWrite(prs, s) =$ "nil"
THEN 0
ELSE CHOOSE $s \in Slots :$
$\wedge\, PeakVotedWrite(prs, s) \neq$ "nil"
$\wedge \forall t \in (s+1) \,..\, NumWrites : PeakVotedWrite(prs, t) =$ "nil"

$$PrepareNoticeMsgs \triangleq [type : \{\text{"PrepareNotice"}\}, src : Replicas, bal : Ballots, commit_prev : \{0\} \cup Slots]$$

$$PrepareNoticeMsg(r, b, cp) \triangleq [type \mapsto \text{"PrepareNotice"}, src \mapsto r,$$

$bal \mapsto b,$
$commit_prev \mapsto cp]$

$AcceptMsgs \triangleq [type : \{\text{“Accept”}\}, src : Replicas,$
$bal : Ballots,$
$slot : Slots,$
$write : Writes]$

$AcceptMsg(r, b, s, c) \triangleq [type \mapsto \text{“Accept”}, src \mapsto r,$
$bal \mapsto b,$
$slot \mapsto s,$
$write \mapsto c]$

$AcceptReplyMsgs \triangleq [type : \{\text{“AcceptReply”}\}, src : Replicas,$
$bal : Ballots,$
$slot : Slots]$

$AcceptReplyMsg(r, b, s) \triangleq [type \mapsto \text{“AcceptReply”}, src \mapsto r,$
$bal \mapsto b,$
$slot \mapsto s]$

$CommitNoticeMsgs \triangleq [type : \{\text{“CommitNotice”}\}, upto : Slots]$

$CommitNoticeMsg(u) \triangleq [type \mapsto \text{“CommitNotice”}, upto \mapsto u]$

Lease protocol messages. Guard/$Renew$ carry the roster and a per-pair seq num; replies and $Revoke(Reply)$ carry only the ballot $+ seq$.

$GuardMsgs \triangleq [type : \{\text{“Guard”}\}, grantor : Replicas,$
$grantee : Replicas,$
$bal : Ballots,$
$ros : Rosters,$
$seq : SeqNums]$

$GuardMsg(f, p, b, ro, s) \triangleq [type \mapsto \text{“Guard”}, grantor \mapsto f,$
$grantee \mapsto p,$
$bal \mapsto b,$
$ros \mapsto ro,$
$seq \mapsto s]$

$GuardReplyMsgs \triangleq [type : \{\text{“GuardReply”}\}, grantee : Replicas,$
$grantor : Replicas,$
$bal : Ballots,$
$seq : SeqNums]$

$GuardReplyMsg(p, f, b, s) \triangleq [type \mapsto \text{“GuardReply”}, grantee \mapsto p,$
$grantor \mapsto f,$
$bal \mapsto b,$
$seq \mapsto s]$

$RenewMsgs \triangleq [type : \{\text{“Renew”}\}, grantor : Replicas,$
$grantee : Replicas,$
$bal : Ballots,$
$ros : Rosters,$
$seq : SeqNums]$

$RenewMsg(f, p, b, ro, s) \triangleq [type \mapsto \text{“Renew”}, grantor \mapsto f,$
$grantee \mapsto p,$
$bal \mapsto b,$

$ros \mapsto ro,$
$seq \mapsto s]$

$RenewReplyMsgs \triangleq [type : \{\text{“RenewReply”}\}, grantee : Replicas,$
$grantor : Replicas,$
$bal : Ballots,$
$seq : SeqNums]$

$RenewReplyMsg(p, f, b, s) \triangleq [type \mapsto \text{“RenewReply”}, grantee \mapsto p,$
$grantor \mapsto f,$
$bal \mapsto b,$
$seq \mapsto s]$

$RevokeMsgs \triangleq [type : \{\text{“Revoke”}\}, grantor : Replicas,$
$grantee : Replicas,$
$bal : Ballots,$
$seq : SeqNums]$

$RevokeMsg(f, p, b, s) \triangleq [type \mapsto \text{“Revoke”}, grantor \mapsto f,$
$grantee \mapsto p,$
$bal \mapsto b,$
$seq \mapsto s]$

$RevokeReplyMsgs \triangleq [type : \{\text{“RevokeReply”}\}, grantee : Replicas,$
$grantor : Replicas,$
$bal : Ballots,$
$seq : SeqNums]$

$RevokeReplyMsg(p, f, b, s) \triangleq [type \mapsto \text{“RevokeReply”}, grantee \mapsto p,$
$grantor \mapsto f,$
$bal \mapsto b,$
$seq \mapsto s]$

$$\begin{aligned} Messages \triangleq\ & PrepareMsgs \\ \cup\ & PrepareReplyMsgs \\ \cup\ & PrepareNoticeMsgs \\ \cup\ & AcceptMsgs \\ \cup\ & AcceptReplyMsgs \\ \cup\ & CommitNoticeMsgs \\ \cup\ & GuardMsgs \\ \cup\ & GuardReplyMsgs \\ \cup\ & RenewMsgs \\ \cup\ & RenewReplyMsgs \\ \cup\ & RevokeMsgs \\ \cup\ & RevokeReplyMsgs \end{aligned}$$

Main algorithm in *PlusCal*.

algorithm *Bodega*

variable $msgs = \{\}$, messages in the network
$node = [r \in Replicas \mapsto NullNode]$, replica merged state
$pending = InitPending$, sequence of pending reqs
$observed = \langle\rangle$, client observed events
$crashed = [r \in Replicas \mapsto \text{FALSE}]$, replica crashed flag
$time = [r \in Replicas \mapsto 1]$; per-node monotone clock

define

A lease from grantee p's perspective is "active" iff p's $asGrantee[f]$ is in *Renewed* status, un-expired, and recording roster ros.

$$FGrantsPWithRos(f, p, ros) \triangleq \begin{array}{l} \land\, node[p].asGrantee[f].status = \text{"Renewed"} \\ \land\, node[p].asGrantee[f].leaseExpire > time[p] \\ \land\, node[p].asGrantee[f].ros = ros \end{array}$$

True if a node r thinks of itself as leader of the latest roster.

$$\begin{array}{l} ThinkAmLeader(r) \triangleq \\ \quad \textsc{let}\ ros \triangleq node[r].rosMaxKnown \\ \quad \textsc{in}\ \ \land\, node[r].leader = r \\ \qquad \land\, node[r].balPrepared = node[r].balMaxKnown \\ \qquad \land\, ros.bal = node[r].balMaxKnown \\ \qquad \land\, ros.leader = r \\ \qquad \land\, Cardinality(\{f \in Replicas : \\ \qquad\qquad FGrantsPWithRos(f, r, ros)\}) \\ \qquad\quad \geq MajorityNum \end{array}$$

True if a node r thinks of itself as a follower of the latest roster.

$$\begin{array}{l} ThinkAmFollower(r) \triangleq \\ \quad \textsc{let}\ ros \triangleq node[r].rosMaxKnown \\ \quad \textsc{in}\ \ \land\, node[r].leader \neq r \\ \qquad \land\, ros.bal = node[r].balMaxKnown \\ \qquad \land\, ros.leader \neq r \\ \qquad \land\, ros.leader \neq \text{"none"} \\ \qquad \land\, Cardinality(\{f \in Replicas : \\ \qquad\qquad FGrantsPWithRos(f, ros.leader, ros)\}) \\ \qquad\quad \geq MajorityNum \end{array}$$

True if a node r thinks of itself as a responder of the latest roster.

$$\begin{array}{l} ThinkAmResponder(r) \triangleq \\ \quad \land\, ThinkAmFollower(r) \\ \quad \land\, r \in node[r].rosMaxKnown.responders \end{array}$$

The "safety threshold" condition: a node serves a read locally only if it has committed up to where previous leaders could have committed.

$$BallotTransferred(r) \triangleq node[r].commitUpTo \geq node[r].commitPrev$$

Given caller node r (a leader or a responder) and a set of $AcceptReplies$, decide whether the write is committable: majority quorum AND the caller's known roster's *responders* all voted.

$$\begin{array}{l} WriteCommittable(r, ars) \triangleq \\ \quad \textsc{let}\ acceptors \triangleq \{ar.src : ar \in ars\} \\ \quad \textsc{in}\ \ \land\, Cardinality(acceptors) \geq MajorityNum \\ \qquad \land\, node[r].rosMaxKnown.responders \subseteq acceptors \end{array}$$

When node r's ballot rises to nb, any live outgoing grant whose roster is at an older ballot must be revoked: *Guarding* grants reset locally, *Renewing* grants move to *Revoking* with *seq* bumped. Expired grants are untouched (cleared later by $TimeTick$).

$$\begin{array}{l} RevokeStaleAsGrantor(r, nb) \triangleq \\ \quad [p \in Replicas \mapsto \\ \qquad \textsc{let}\ g \triangleq node[r].asGrantor[p] \\ \qquad \textsc{in}\ \ \textsc{if}\ \ \land\, g.ros.bal < nb \\ \qquad\qquad\qquad\ \ \land\, g.status = \text{"Guarding"} \\ \qquad\qquad\qquad\ \ \land\, g.guardExpire > time[r] \end{array}$$

```
                THEN [NullGrantorState EXCEPT !.seq = g.seq]
            ELSE IF ∧ g.ros.bal < nb
                    ∧ g.status = "Renewing"
                    ∧ g.leaseExpire > time[r]
                THEN [g EXCEPT !.status = "Revoking", !.seq = g.seq + 1]
            ELSE g]

    Revoke messages to emit alongside RevokeStaleAsGrantor(r, nb). Seq num
    matches the bumped seq in the Revoking state above.
    RevokeStaleSendMsgs(r, nb) ≜
        {RevokeMsg(r, p, nb, node[r].asGrantor[p].seq + 1) :
            p ∈ {p ∈ Replicas :
                    ∧ node[r].asGrantor[p].ros.bal < nb
                    ∧ node[r].asGrantor[p].status = "Renewing"
                    ∧ node[r].asGrantor[p].leaseExpire > time[r]}}

    Miscellaneous model checking helpers:
    reqsMade ≜ {e.cmd : e ∈ {e ∈ Range(observed) : e.type = "Req"}}

    acksRecv ≜ {e.cmd : e ∈ {e ∈ Range(observed) : e.type = "Ack"}}

    AppendObserved(seq) ≜
        LET filter(e) ≜ IF e.type = "Req" THEN e.cmd ∉ reqsMade
                                          ELSE e.cmd ∉ acksRecv
        IN observed ∘ SelectSeq(seq, filter)

    UnseenPending(r) ≜
        LET filter(c) ≜ ∀ s ∈ Slots : node[r].insts[s].write ≠ c
        IN SelectSeq(pending, filter)

    RemovePending(cmd) ≜
        LET filter(c) ≜ c ≠ cmd
        IN SelectSeq(pending, filter)

    terminated ≜ ∧ Len(pending) = 0
                 ∧ Cardinality(reqsMade) = NumCommands
                 ∧ Cardinality(acksRecv) = NumCommands

    numCrashed ≜ Cardinality({r ∈ Replicas : crashed[r]})

    timeExhausted ≜ ∀ r ∈ Replicas : time[r] = MaxTime
end define ;

Send a set of messages helper.
macro Send(set) begin
    msgs := msgs ∪ set ;
end macro ;

Observe client events helper.
macro Observe(seq) begin
    observed := AppendObserved(seq) ;
end macro ;

Resolve a pending command helper.
macro Resolve(c) begin
    pending := RemovePending(c) ;
end macro ;
```

```
Someone steps up as leader: picks a greater ballot, picks a fresh
responders set, and broadcasts Prepare. Also revokes any stale outgoing
lease grants: Guarding ones are reset locally; Renewing ones transition
to Revoking and send Revoke to the respective grantees.
macro BecomeLeader(r) begin
    if I'm not a current leader
    await node[r].leader ≠ r ;
    pick a greater ballot and an arbitrary responders set for new roster
    with b ∈ Ballots,
         resps ∈ SUBSET {f ∈ Replicas : f ≠ r},
         ros = Roster(b, r, resps)
     do
        await ∧ b > node[r].balMaxKnown
              ∧ ¬∃ m ∈ msgs : (m.type = "Prepare") ∧ (m.bal = b) ;
                    W.L.O.G., using this clause to model that ballot
                    numbers from different proposers be unique
        update states and restart Prepare phase for in-progress instances;
        also revoke any stale outgoing lease grants
        node[r].leader := r ||
        node[r].commitPrev := InfinitySlot ||
        node[r].balPrepared    := 0 ||
        node[r].balMaxKnown := b ||
        node[r].rosMaxKnown := ros ||
        node[r].insts :=
            [s ∈ Slots ↦
                [node[r].insts[s]
                    EXCEPT !.status = IF @ = "Accepting"
                                      THEN "Preparing"
                                      ELSE @]] ||
        node[r].asGrantor := RevokeStaleAsGrantor(r, b) ;
        broadcast Prepare and reply to myself instantly; also send Revokes
        for the just-Revoking grants
        Send(     {PrepareMsg(r, b),
                   PrepareReplyMsg(r, b, VotesByNode(node[r]))}
             ∪ RevokeStaleSendMsgs(r, b)) ;
    end with ;
end macro ;

Replica replies to a Prepare message. Also revokes any stale outgoing lease
grants the same way as BecomeLeader.
macro HandlePrepare(r) begin
    if receiving a Prepare message with larger ballot than ever seen
    with m ∈ msgs do
        await ∧ m.type = "Prepare"
              ∧ m.bal > node[r].balMaxKnown
              Prepare arrives along with an active lease from its sender;
              here we assert there's a Renewed grantee-side record for it.
              This stands in for "Prepare piggybacks the new roster"
              ∧ node[r].asGrantee[m.src].status = "Renewed"
              ∧ node[r].asGrantee[m.src].leaseExpire > time[r]
              ∧ node[r].asGrantee[m.src].ros.bal = m.bal ;
        with ros = node[r].asGrantee[m.src].ros do
            update states and reset statuses; also revoke stale outgoing
            lease grants
```

```
            node[r].leader := m.src ||
            node[r].commitPrev := InfinitySlot ||
            node[r].balMaxKnown := m.bal ||
            node[r].rosMaxKnown := ros ||
            node[r].insts :=
                [s ∈ Slots ↦
                    [node[r].insts[s]
                        EXCEPT !.status = IF @ = "Accepting"
                                          THEN "Preparing"
                                          ELSE @]] ||
            node[r].asGrantor := RevokeStaleAsGrantor(r, m.bal);
            send back PrepareReply with my voted list; also send Revokes
            for the just-Revoking grants
            Send(     {PrepareReplyMsg(r, m.bal, VotesByNode(node[r]))}
                  ∪ RevokeStaleSendMsgs(r, m.bal));
        end with ;
    end with ;
end macro ;

Leader gathers PrepareReply messages until condition met, then marks
the corresponding ballot as prepared and saves highest voted commands.
macro HandlePrepareReplies(r) begin
    if I'm waiting for PrepareReplies
    await ∧ node[r].leader = r
          ∧ node[r].balPrepared = 0;
    when there are enough number of PrepareReplies of desired ballot
    with prs = {m ∈ msgs : ∧ m.type = "PrepareReply"
                           ∧ m.bal = node[r].balMaxKnown}
     do
        await Cardinality({pr.src : pr ∈ prs}) ≥ MajorityNum;
        with prsGot ∈ {prsGot ∈ SUBSET prs :
                          Cardinality({pr.src : pr ∈ prsGot}) ≥ MajorityNum},
             lts = LastTouchedSlot(prsGot)
         do
            marks this ballot as prepared and saves highest voted command
            in each slot if any
            node[r].balPrepared := node[r].balMaxKnown ||
            node[r].insts :=
                [s ∈ Slots ↦
                    LET pvw ≜ PeakVotedWrite(prsGot, s)
                        adopted ≜ ∨ node[r].insts[s].status = "Preparing"
                                  ∨ ∧ node[r].insts[s].status = "Empty"
                                    ∧ pvw ≠ "nil"
                    IN  [node[r].insts[s]
                            EXCEPT !.status = IF adopted
                                              THEN "Accepting"
                                              ELSE @,
                                   !.write  = pvw,
                                   !.voted  = IF adopted
                                              THEN [bal ↦ node[r].balMaxKnown,
                                                    write ↦ pvw]
                                              ELSE @]] ||
            node[r].commitPrev := lts;
            send Accept messages for in-progress instances and reply to myself
```

```
            instantly; send PrepareNotices as well (mimicking threshold-carrying
            messages, in paper manuscript this is achieved via Guard messages)
            Send(   UNION
                        {{AcceptMsg(r, node[r].balPrepared, s, node[r].insts[s].write),
                          AcceptReplyMsg(r, node[r].balPrepared, s)} :
                         s ∈ {s ∈ Slots : node[r].insts[s].status = "Accepting"}}
                    ∪ {PrepareNoticeMsg(r, node[r].balPrepared, lts)});
        end with ;
    end with ;
end macro ;

Follower receives PrepareNotice from a prepared and recovered leader,
updating its commitPrev accordingly.
macro HandlePrepareNotice(r) begin
    if I'm a follower waiting on PrepareNotice
    await ∧ ThinkAmFollower(r)
          ∧ node[r].commitPrev = InfinitySlot ;
    when there's a PrepareNotice message in effect
    with m ∈ msgs do
        await ∧ m.type = "PrepareNotice"
              ∧ m.bal = node[r].balMaxKnown ;
        update my commitPrev
        node[r].commitPrev := m.commit_prev ;
    end with ;
end macro ;

A prepared leader takes a new write request into the next empty slot.
macro TakeNewWriteRequest(r) begin
    if I'm a prepared leader and there's pending write request
    with unseen = UnseenPending(r) do
        await ∧ ThinkAmLeader(r)
              ∧ ∃ s ∈ Slots : node[r].insts[s].status = "Empty"
              ∧ Len(unseen) > 0
              ∧ Head(unseen) ∈ Writes ;
        find the next empty slot and pick a pending request
        with s = FirstEmptySlot(node[r].insts),
             c = Head(unseen)
         do
            update slot status and voted
            node[r].insts[s].status := "Accepting" ||
            node[r].insts[s].write := c ||
            node[r].insts[s].voted.bal := node[r].balPrepared ||
            node[r].insts[s].voted.write := c ;
            broadcast Accept and reply to myself instantly
            Send({AcceptMsg(r, node[r].balPrepared, s, c),
                  AcceptReplyMsg(r, node[r].balPrepared, s)}) ;
            append to observed events sequence if haven't yet
            Observe(⟨ReqEvent(c)⟩) ;
        end with ;
    end with ;
end macro ;

Replica replies to an Accept message. If the Accept's ballot is higher than
what we've known, also revokes any stale outgoing lease grants.
macro HandleAccept(r) begin
```

```
    if I'm a follower
    await ThinkAmFollower(r);
    if receiving an unreplied Accept message with valid ballot, piggybacked
    with an active lease from the sender
    with m ∈ msgs do
        await ∧ m.type = "Accept"
              ∧ m.bal ≥ node[r].balMaxKnown
              ∧ m.bal ≥ node[r].insts[m.slot].voted.bal
              ∧ node[r].asGrantee[m.src].status = "Renewed"
              ∧ node[r].asGrantee[m.src].leaseExpire > time[r]
              ∧ node[r].asGrantee[m.src].ros.bal = m.bal;
        update node states and corresponding instance's states; also
        revoke any stale outgoing lease grants
        node[r].leader := m.src ||
        node[r].balMaxKnown := m.bal ||
        node[r].rosMaxKnown := node[r].asGrantee[m.src].ros ||
        node[r].insts[m.slot].status := "Accepting" ||
        node[r].insts[m.slot].write := m.write ||
        node[r].insts[m.slot].voted.bal := m.bal ||
        node[r].insts[m.slot].voted.write := m.write ||
        node[r].asGrantor := RevokeStaleAsGrantor(r, m.bal);
        send back AcceptReply; also send Revokes for the just-Revoking grants
        Send(    {AcceptReplyMsg(r, m.bal, m.slot)}
             ∪  RevokeStaleSendMsgs(r, m.bal));
    end with ;
end macro ;

Leader gathers AcceptReply messages for a slot until condition met,
then marks the slot as committed and acknowledges the client.
macro HandleAcceptReplies(r) begin
    if I'm a prepared leader
    await ∧ ThinkAmLeader(r)
          ∧ node[r].commitUpTo < NumWrites
          ∧ node[r].insts[node[r].commitUpTo + 1].status = "Accepting";
              W.L.O.G., only enabling the next slot after commitUpTo
    for this slot, when there is a good set of AcceptReplies that is at
    least a majority number and that covers all responders
    with s = node[r].commitUpTo + 1,
         c = node[r].insts[s].write,
         ls = s − 1,
         v = IF ls = 0 THEN "nil" ELSE node[r].insts[ls].write,
         ars = {m ∈ msgs : ∧ m.type = "AcceptReply"
                           ∧ m.slot = s
                           ∧ m.bal = node[r].balPrepared}
     do
        await WriteCommittable(r, ars);
        with arsGot ∈ {arsGot ∈ SUBSET ars :
                          WriteCommittable(r, arsGot)}
         do
            marks this slot as committed and apply command
            node[r].insts[s].status := "Committed" ||
            node[r].commitUpTo := s;
            append to observed events sequence if haven't yet, and remove
            the command from pending
```

```
            Observe(⟨AckEvent(c, v, r)⟩);
            Resolve(c);
            broadcast CommitNotice to followers
            Send({CommitNoticeMsg(s)});
        end with ;
    end with ;
end macro ;

Replica receives new commit notification.
macro HandleCommitNotice(r) begin
    if I'm a follower waiting on CommitNotice
    await ∧ ThinkAmFollower(r)
          ∧ node[r].commitUpTo < NumWrites
          ∧ node[r].insts[node[r].commitUpTo + 1].status = "Accepting";
    for this slot, when there's a CommitNotice message
    with s = node[r].commitUpTo + 1,
         c = node[r].insts[s].write,
         m ∈ msgs
     do
        await ∧ m.type = "CommitNotice"
              ∧ m.upto = s;
        marks this slot as committed and apply command
        node[r].insts[s].status := "Committed" ||
        node[r].commitUpTo := s;
    end with ;
end macro ;

A prepared leader or a responder follower takes a new read request and
serves it locally.
macro TakeNewReadRequest(r) begin
    if I'm a caught-up leader or responder follower
    with unseen = UnseenPending(r) do
        await ∧ ∨ ThinkAmLeader(r)
                ∨ ThinkAmResponder(r)
              ∧ BallotTransferred(r)
              ∧ Len(unseen) > 0
              ∧ Head(unseen) ∈ Reads;
        pick a pending request; examine my log and find the last non-empty
        slot, check its status
        with s = LastNonEmptySlot(node[r].insts),
             v = IF s = 0 THEN "nil" ELSE node[r].insts[s].write,
             c = Head(unseen)
         do
            if the latest value is in Committed status, can directly reply;
            otherwise, should hold until I've received enough broadcasted
            AcceptReplies indicating that the write is surely to be committed
            await ∨ s = 0
                  ∨ s > 0 ∧ node[r].insts[s].status = "Committed"
                  ∨ LET ars ≜ {m ∈ msgs : ∧ m.type = "AcceptReply"
                                          ∧ m.slot = s
                                          ∧ m.bal = node[r].balMaxKnown}
                    IN WriteCommittable(r, ars);
            acknowledge client with the latest value, and remove the command
            from pending
            Observe(⟨ReqEvent(c), AckEvent(c, v, r)⟩);
```

```
            Resolve(c);
        end with ;
    end with ;
end macro ;

Replica node crashes itself under promised conditions.
macro ReplicaCrashes(r) begin
    if less than (N − MajorityNum) number of replicas have failed
    await ∧ MajorityNum + numCrashed < Cardinality(Replicas)
          ∧ ¬crashed[r]
          ∧ node[r].balMaxKnown < MaxBallot;
    mark myself as crashed
    crashed[r] := TRUE;
end macro ;

Grantor f opens new leases to peers for its known current roster. Gated
by the condition that f has a non-null rosMaxKnown and f is not actively
granting to anyone.
macro GrantorInitiateLeases(f) begin
    await ∧ node[f].rosMaxKnown ≠ NullRoster
          ∧ ∀ p ∈ Replicas : node[f].asGrantor[p].status = "None";
    node[f].asGrantor :=
        [p ∈ Replicas ↦
            [status ↦ "Guarding",
             guardExpire ↦ time[f] + TGuard,
             leaseExpire ↦ 0,
             ros ↦ node[f].rosMaxKnown,
             seq ↦ node[f].asGrantor[p].seq + 1]];
    Send({GuardMsg(f, p,
                   node[f].balMaxKnown,
                   node[f].rosMaxKnown,
                   node[f].asGrantor[p].seq + 1) :
          p ∈ Replicas});
end macro ;

Grantee p receives a Guard from grantor f. Accept iff ballot is at least as high as
p's view and the message's seq is strictly higher than any seq ever observed on
this pair. If ballot rises, also revokes stale outgoing lease grants from p.
macro HandleGuard(p) begin
    with m ∈ msgs do
        await ∧ m.type = "Guard"
              ∧ m.grantee = p
              ∧ m.bal ≥ node[p].balMaxKnown
              ∧ m.seq > node[p].asGrantee[m.grantor].seq
              ∧ node[p].asGrantee[m.grantor].status = "None";
        update ballot in case higher, and start Guarded state; bump seq;
        also revoke any stale outgoing lease grants
        node[p].balMaxKnown := m.bal ||
        node[p].asGrantee[m.grantor] :=
            [status ↦ "Guarded",
             guardExpire ↦ time[p] + TGuard,
             leaseExpire ↦ 0,
             ros ↦ m.ros,
             seq ↦ m.seq + 1] ||
        node[p].asGrantor := RevokeStaleAsGrantor(p, m.bal);
```

```
        reply GuardReply back to grantor, stamped with the new seq; also
        send Revokes for the just-Revoking outgoing grants
        Send(      {GuardReplyMsg(p, m.grantor, m.bal, m.seq + 1)}
              ∪  RevokeStaleSendMsgs(p, m.bal));
    end with ;
end macro ;

Grantor f receives a GuardReply: transition asGrantor[m.grantee] from
Guarding to Renewing; send first Renew.
macro HandleGuardReply(f) begin
    with m ∈ msgs do
        await ∧ m.type = "GuardReply"
              ∧ m.grantor = f
              ∧ m.bal = node[f].balMaxKnown
              ∧ m.seq > node[f].asGrantor[m.grantee].seq
              ∧ node[f].asGrantor[m.grantee].status = "Guarding";
        loose expiry for safety, tightened at first RenewReply
        node[f].asGrantor[m.grantee] :=
            [node[f].asGrantor[m.grantee] EXCEPT
                ! .status = "Renewing",
                ! .leaseExpire = time[f] + TGuard + TLease,
                ! .seq = m.seq + 1];
        Send({RenewMsg(f, m.grantee, m.bal,
                       node[f].rosMaxKnown, m.seq + 1)});
    end with ;
end macro ;

Grantor f spontaneously sends subsequent Renews to its current grantees.
macro SpontaneousRenew(f) begin
    await ∃ p ∈ Replicas :
            ∧ node[f].asGrantor[p].ros.bal = node[f].balMaxKnown
            ∧ node[f].asGrantor[p].status = "Renewing"
            ∧ time[f] < node[f].asGrantor[p].leaseExpire
            ∧ time[f] + TLease > node[f].asGrantor[p].leaseExpire;
    with ps = {p ∈ Replicas :
                  ∧ node[f].asGrantor[p].ros.bal = node[f].balMaxKnown
                  ∧ node[f].asGrantor[p].status = "Renewing"
                  ∧ time[f] < node[f].asGrantor[p].leaseExpire
                  ∧ time[f] + TLease > node[f].asGrantor[p].leaseExpire} do
        node[f].asGrantor :=
            [p ∈ Replicas ↦
              IF p ∈ ps
                THEN [node[f].asGrantor[p] EXCEPT
                        !.leaseExpire = node[f].asGrantor[p].leaseExpire + TLease,
                        !.seq = node[f].asGrantor[p].seq + 1]
                ELSE node[f].asGrantor[p]];
        Send({RenewMsg(f, p,
                       node[f].balMaxKnown,
                       node[f].rosMaxKnown,
                       node[f].asGrantor[p].seq + 1) :
              p ∈ ps});
    end with ;
end macro ;

Grantee p receives a Renew.
```

```
macro HandleRenew(p) begin
    with m ∈ msgs do
        await ∧ m.type = "Renew"
              ∧ m.grantee = p
              ∧ m.bal = node[p].balMaxKnown
              ∧ m.seq > node[p].asGrantee[m.grantor].seq
              ∧ ∨ ∧ node[p].asGrantee[m.grantor].status = "Guarded"
                  ∧ time[p] < node[p].asGrantee[m.grantor].guardExpire
                ∨ ∧ node[p].asGrantee[m.grantor].status = "Renewed"
                  ∧ time[p] < node[p].asGrantee[m.grantor].leaseExpire
                  ∧ time[p] + TLease > node[p].asGrantee[m.grantor].leaseExpire ;
        node[p].asGrantee[m.grantor] :=
            [node[p].asGrantee[m.grantor] EXCEPT
                ! .status = "Renewed",
                ! .leaseExpire = time[p] + TLease,
                ! .ros = m.ros,
                ! .seq = m.seq + 1] ;
        Send({RenewReplyMsg(p, m.grantor, m.bal, m.seq + 1)}) ;
    end with ;
end macro ;

Grantor f receives a RenewReply; tightens down the loose expiry.
macro HandleRenewReply(f) begin
    with m ∈ msgs do
        await ∧ m.type = "RenewReply"
              ∧ m.grantor = f
              ∧ m.bal = node[f].balMaxKnown
              ∧ m.seq > node[f].asGrantor[m.grantee].seq
              ∧ node[f].asGrantor[m.grantee].status = "Renewing"
              ∧ time[f] + TLease > node[f].asGrantor[m.grantee].leaseExpire ;
        node[f].asGrantor[m.grantee] :=
            [node[f].asGrantor[m.grantee] EXCEPT
                ! .leaseExpire = time[f] + TLease,
                ! .seq = m.seq] ;
    end with ;
end macro ;

Grantee p processes a Revoke from grantor f. If ballot rises, also
revokes stale outgoing lease grants from p.
macro HandleRevoke(p) begin
    with m ∈ msgs do
        await ∧ m.type = "Revoke"
              ∧ m.grantee = p
              ∧ m.bal ≥ node[p].balMaxKnown
              ∧ m.seq > node[p].asGrantee[m.grantor].seq
              ∧ node[p].asGrantee[m.grantor].status ∈ {"Guarded", "Renewed"} ;
        update ballot in case higher, clear grantee state but preserve seq;
        also revoke any stale outgoing lease grants
        node[p].balMaxKnown := m.bal ||
        node[p].asGrantee[m.grantor] :=
            [NullGranteeState EXCEPT !.seq = m.seq + 1] ||
        node[p].asGrantor := RevokeStaleAsGrantor(p, m.bal) ;
        Send(     {RevokeReplyMsg(p, m.grantor, m.bal, m.seq + 1)}
              ∪ RevokeStaleSendMsgs(p, m.bal)) ;
    end with ;
```

```
end macro ;

Grantor f receives a RevokeReply; drops the lease promptly.
macro HandleRevokeReply(f) begin
    with m ∈ msgs do
        await ∧ m.type = "RevokeReply"
              ∧ m.grantor = f
              ∧ m.bal = node[f].balMaxKnown
              ∧ m.seq > node[f].asGrantor[m.grantee].seq
              ∧ node[f].asGrantor[m.grantee].status = "Revoking" ;
        node[f].asGrantor[m.grantee] :=
            [NullGrantorState EXCEPT !.seq = m.seq] ;
    end with ;
end macro ;

Advances time by one tick globally, and garbage-collects expired lease
state. Per-pair seq counters are preserved across GC.
macro TimeTick() begin
    await ∀ r ∈ Replicas : time[r] < MaxTime ;
    time := [r ∈ Replicas ↦ time[r] + 1] ;
    node := [r ∈ Replicas ↦
        [node[r] EXCEPT
            ! .asGrantor =
                [p ∈ Replicas ↦
                    IF ∨ ∧ node[r].asGrantor[p].status ∈ {"Renewing", "Revoking"}
                         ∧ node[r].asGrantor[p].leaseExpire ≤ time[r]
                       ∨ ∧ node[r].asGrantor[p].status = "Guarding"
                         ∧ node[r].asGrantor[p].guardExpire ≤ time[r]
                    THEN [NullGrantorState EXCEPT !.seq = node[r].asGrantor[p].seq]
                    ELSE node[r].asGrantor[p]],
            ! .asGrantee =
                [f ∈ Replicas ↦
                    IF ∨ ∧ node[r].asGrantee[f].status = "Renewed"
                         ∧ node[r].asGrantee[f].leaseExpire ≤ time[r]
                       ∨ ∧ node[r].asGrantee[f].status = "Guarded"
                         ∧ node[r].asGrantee[f].guardExpire ≤ time[r]
                    THEN [NullGranteeState EXCEPT !.seq = node[r].asGrantee[f].seq]
                    ELSE node[r].asGrantee[f]]]] ;
end macro ;

Replica server node main loop: consensus actions + lease protocol actions
+ global time tick are all available via either/or.
process Replica ∈ Replicas
 begin
    rloop: while (¬terminated) ∧ (¬timeExhausted) ∧ (¬crashed[self]) do
        either
            BecomeLeader(self) ;
        or
            HandlePrepare(self) ;
        or
            HandlePrepareReplies(self) ;
        or
            HandlePrepareNotice(self) ;
        or
            TakeNewWriteRequest(self) ;
```

```
            or
                HandleAccept(self);
            or
                HandleAcceptReplies(self);
            or
                HandleCommitNotice(self);
            or
                TakeNewReadRequest(self);
            or
                GrantorInitiateLeases(self);
            or
                HandleGuard(self);
            or
                HandleGuardReply(self);
            or
                SpontaneousRenew(self);
            or
                HandleRenew(self);
            or
                HandleRenewReply(self);
            or
                HandleRevoke(self);
            or
                HandleRevokeReply(self);
            or
                TimeTick();
            or
                if NodeFailuresOn then
                    ReplicaCrashes(self);
                end if ;
            end either ;
        end while ;
    end process ;

    end algorithm ;
```

```
MODULE Bodega_MC
EXTENDS Bodega

TLC config-related defs.
ConditionalPerm(set) ≜ IF Cardinality(set) > 1
                       THEN Permutations(set)
                       ELSE {}

SymmetricPerms ≜ ConditionalPerm(Replicas)
                 ∪ ConditionalPerm(Writes)
                 ∪ ConditionalPerm(Reads)

ConstMaxBallot ≜ 2

ConstTGuard ≜ 1
ConstTLease ≜ 1
ConstMaxTime ≜ 3
```

Type check invariant.

$$
\begin{aligned}
TypeOK \triangleq\ & \land \forall\, m \in msgs : m \in Messages \\
& \land \forall\, r \in Replicas : node[r] \in NodeStates \\
& \land \forall\, r \in Replicas : time[r] \in Times \\
& \land Len(pending) \leq NumCommands \\
& \land Cardinality(Range(pending)) = Len(pending) \\
& \land \forall\, c \in Range(pending) : c \in Commands \\
& \land Len(observed) \leq 2 * NumCommands \\
& \land Cardinality(Range(observed)) = Len(observed) \\
& \land Cardinality(reqsMade) \geq Cardinality(acksRecv) \\
& \land \forall\, e \in Range(observed) : e \in ClientEvents \\
& \land \forall\, r \in Replicas : crashed[r] \in \text{BOOLEAN}
\end{aligned}
$$

THEOREM $Spec \Rightarrow \Box\, TypeOK$

Lease expiration safety property.

$$
\begin{aligned}
& LeaseExpirationSafety \triangleq \\
& \quad \forall\, f, p \in Replicas : \\
& \qquad (\land node[p].asGrantee[f].status = \text{“Renewed”} \\
& \qquad\ \land node[p].asGrantee[f].leaseExpire > time[p]) \\
& \qquad\quad \Rightarrow (\land node[f].asGrantor[p].status \in \{\text{“Renewing”}, \text{“Revoking”}\} \\
& \qquad\qquad \land node[f].asGrantor[p].leaseExpire \\
& \qquad\qquad\quad \geq node[p].asGrantee[f].leaseExpire)
\end{aligned}
$$

THEOREM $Spec \Rightarrow \Box\, LeaseExpirationSafety$

Lease uniqueness guarantee assertions.

$$
\begin{aligned}
& AtMostGrantsOneRoster \triangleq \\
& \quad \forall\, f \in Replicas, b \in Ballots : \\
& \qquad Cardinality(\{ros \in Rosters : \\
& \qquad\qquad \exists\, p \in Replicas : \land FGrantsPWithRos(f, p, ros) \\
& \qquad\qquad\qquad\qquad \land ros.bal = b\}) \leq 1
\end{aligned}
$$

$$
\begin{aligned}
& AtMostOneStableRoster \triangleq \\
& \quad \forall\, ros1, ros2 \in Rosters : \\
& \qquad (\land Cardinality(\{f \in Replicas : \\
& \qquad\qquad\qquad \exists\, p \in Replicas : FGrantsPWithRos(f, p, ros1)\}) \\
& \qquad\qquad \geq MajorityNum \\
& \qquad\ \land Cardinality(\{f \in Replicas : \\
& \qquad\qquad\qquad \exists\, p \in Replicas : FGrantsPWithRos(f, p, ros2)\}) \\
& \qquad\qquad \geq MajorityNum) \\
& \qquad \Rightarrow (ros1 = ros2)
\end{aligned}
$$

$$
\begin{aligned}
\text{THEOREM } Spec \Rightarrow\ & \land \Box\, AtMostGrantsOneRoster \\
& \land \Box\, AtMostOneStableRoster
\end{aligned}
$$

Linearizability constraint.

$$
\begin{aligned}
ReqPosOfCmd(c) \triangleq\ & \text{CHOOSE } i \in 1 \,..\, Len(observed) : \\
& \land observed[i].type = \text{“Req”} \\
& \land observed[i].cmd = c
\end{aligned}
$$

$$AckPosOfCmd(c) \triangleq \text{CHOOSE } i \in 1 \,..\, Len(observed) :$$
$$\wedge\, observed[i].type = \text{“Ack”}$$
$$\wedge\, observed[i].cmd = c$$

$$ResultOfCmd(c) \triangleq observed[AckPosOfCmd(c)].val$$

$$OrderIdxOfCmd(order, c) \triangleq \text{CHOOSE } j \in 1 \,..\, Len(order) : order[j] = c$$

$$LastWriteBefore(order, j) \triangleq$$
$$\text{LET } k \triangleq \text{CHOOSE } k \in 0 \,..\, (j-1) :$$
$$\wedge\, (k = 0 \vee order[k] \in Writes)$$
$$\wedge\, \forall\, l \in (k+1) \,..\, (j-1) : order[l] \in Reads$$
$$\text{IN IF } k = 0 \text{ THEN “nil” ELSE } order[k]$$

$$IsLinearOrder(order) \triangleq$$
$$\wedge\, \{order[j] : j \in 1 \,..\, Len(order)\} = Commands$$
$$\wedge\, \forall\, j \in 1 \,..\, Len(order) :$$
$$ResultOfCmd(order[j]) = LastWriteBefore(order, j)$$

$$ObeysRealTime(order) \triangleq$$
$$\forall\, c1, c2 \in Commands :$$
$$(AckPosOfCmd(c1) < ReqPosOfCmd(c2))$$
$$\Rightarrow (OrderIdxOfCmd(order, c1) < OrderIdxOfCmd(order, c2))$$

$$Linearizability \triangleq$$
$$terminated \Rightarrow$$
$$\exists\, order \in [1 \,..\, NumCommands \to Commands] :$$
$$\wedge\, IsLinearOrder(order)$$
$$\wedge\, ObeysRealTime(order)$$

$$\text{THEOREM } Spec \Rightarrow Linearizability$$

Bodega_MC.cfg

```
SPECIFICATION Spec

CONSTANTS
    Replicas = {s1, s2, s3}
    Writes = {w1, w2}
    Reads = {r1, r2}
    MaxBallot <- ConstMaxBallot
    NodeFailuresOn <- TRUE
    TGuard <- ConstTGuard
    TLease <- ConstTLease
    MaxTime <- ConstMaxTime

SYMMETRY SymmetricPerms

INVARIANTS
    TypeOK
    LeaseExpirationSafety
    AtMostGrantsOneRoster
    AtMostOneStableRoster
    Linearizability

CHECK_DEADLOCK FALSE
```

ARTIFACT EVALUATED
USENIX
AVAILABLE

Equal Opportunity: A Correctness Condition for Ordered Consensus

Yunhao Zhang*
Cornell University

Haobin Ni*
University of Washington

Soumya Basu
OpenReserve Holdings

Shir Cohen
Cornell University

Maofan Yin
University of California, Santa Barbara

Lorenzo Alvisi
Cornell University

Robbert van Renesse
Cornell University

Qi Chen
Microsoft Research

Lidong Zhou
Microsoft Research

Abstract

In proof-of-stake blockchains based on State Machine Replication (SMR), the order of transactions directly affects client-visible financial outcomes. *Ordered consensus* augments the SMR specification by imposing correctness conditions on transaction ordering, with a focus on limiting Byzantine influence. However, real-world ordering attacks can occur even when these conditions hold, often enabled by advantages such as faster networks or proximity to the blockchain infrastructure that allow an adversary to systematically bias outcomes without violating the conditions. To address this gap, we extend ordered consensus with a new model and correctness condition based on *equal opportunity*, a notion of fairness widely used in legal contexts. Equal opportunity requires that candidates who are equally qualified—according to criteria deemed *relevant*—have equal chances of being selected (here, for a given position in the total order). We show how carefully introduced randomness can bound ordering bias, and we introduce the Secret Random Oracle (SRO), a fault-tolerant abstraction for generating such randomness. We present two SRO constructions, based on trusted hardware and threshold verifiable random functions, respectively, and use them to build Pompē-SRO, a new ordered consensus protocol that mitigates well-known ordering attacks. Our evaluation shows that Pompē-SRO effectively mitigates front-running and sandwich attacks at a moderate cost to latency.

1 Introduction

This paper extends *ordered consensus* [81] by motivating, expressing, and enforcing *equal opportunity*, a correctness condition governing how a state machine replication (SMR) [72] protocol orders client requests.

At its core, SMR requires a set of replicas to agree on a single, totally ordered sequence of client requests (*i.e.,* a ledger). When SMR is used for fault tolerance, this order serves only to ensure that all correct replicas process requests consistently. As long as all requests from correct clients eventually appear in the ledger, their relative order does not affect correctness. In blockchains, however, the specific order matters, as it can directly impact the financial rewards associated with the transactions recorded in the ledger.

Ordered consensus [81] formalizes correctness conditions on this order. Each replica associates an *ordering indicator* (*e.g.,* a timestamp) with every client request, expressing how it prefers to order requests relative to one another. Using these indicators, one can show that while Byzantine influence over the ledger order cannot be eliminated, it can be curtailed. In particular, ordered consensus can guarantee *ordering linearizability*: if the lowest indicator assigned by any correct node for command c_2 exceeds the highest indicator assigned to c_1, then c_1 appears before c_2 in the ledger. This guarantee is enforced by the Pompē protocol [81].

The starting point of this paper is the observation that limiting Byzantine influence alone is not sufficient: ordering linearizability fails to capture fairness requirements arising in real-world ordering-based attacks with significant financial consequences [68]. Ordering linearizability only constrains the order of requests whose timestamps are unambiguously separated; it says nothing about how to order requests that arrive close together in time, leaving room for network advantages to systematically determine outcomes.

Consider, for example, the unfair practice known in financial markets as *front-running*, in which a party learns of a large pending buy order for a certain stock and submits its own buy order in advance, so it can later profit from the price increase driven by the ensuing large buy order. Such behavior has been widely documented in both traditional financial markets [57] and decentralized ones [37, 75], often enabled by advantages such as lower network latency. While financial regulations prohibit these practices, Pompē does not prevent them: requests from clients with faster network access are likely to appear earlier in the output ledger.

Our first contribution is a model of fairness grounded in real-world concerns. Drawing on the legal and economic literature on *equal opportunity*, we observe that, whenever items are ranked, their positions depend on their characteristics (or *features*). Some features are deemed *relevant* to the rank-

*The first two authors contributed equally to this work.

ing's purpose, while others are *irrelevant*. For example, U.S. employers and lending agencies are legally forbidden from considering certain irrelevant features (so-called *protected classes*) when making decisions [1, 2]. Intuitively, a fair ranking depends only on relevant features and ignores all others: entries with indistinguishable relevant features should have equal chances of being ranked ahead of one another.

Building on the expressiveness of ordered consensus, we instantiate this notion of equal opportunity for blockchains. In this context, two features are typically considered relevant: the *time* at which clients submit requests and the *fee* they attach to them. Other features, such as geographic location, are irrelevant. Existing ordered consensus protocols, however, do not distinguish between relevant and irrelevant features, nor do they reason about this distinction. As a result, protocols such as Pompē [81], Aequitas [50], and Themis [49] remain vulnerable to adversaries that exploit irrelevant features—for example, faster network connectivity—to perform front-running and related attacks, such as *sandwich attacks*.

Our second contribution is to formalize this notion through two correctness conditions: ε-*Ordering Equality* and Δ-*Ordering Separation*. Informally, ε-Ordering Equality requires that, for requests issued at the same time, the probability of any permutation deviates from the uniform distribution by at most ε. Δ-*Ordering Separation* ensures that, for requests submitted at least Δ time apart, the earlier request precedes the later one in the ledger.

To enforce these conditions, we introduce the Secret Random Oracle (SRO), an abstraction that provides a fault-tolerant and unbiased source of randomness. The SRO is used to inject controlled randomness into the final order derived from replicas' ordering indicators. We present two SRO designs: one based on Trusted Execution Environments (TEEs) and one based on threshold verifiable random functions [30, 40]. Because the profitability of front-running and sandwich attacks depends on obtaining a specific ordering of requests, introducing randomness reduces the likelihood of adversarially preferred outcomes and thereby mitigates them.

Our third contribution is to characterize the trade-off between ε-Ordering Equality and Δ-Ordering Separation. Ideally, both ε and Δ would be small. However, while increasing randomness mitigates the attacks, it can also compromise the role of a request's *relevant* features in determining its position in the ledger: thus, decreasing ε comes at the price of increasing the Δ required to ensure that requests separated in time are ordered correctly despite Byzantine behavior.

Finally, we explore the practical implications of this trade-off by designing, implementing, and evaluating Pompē-SRO, a new ordered consensus protocol for the *partially synchronous* model [38] introduced to circumvent the impossibility of achieving safe, live, and fault-tolerant consensus in fully asynchronous systems [39]. Pompē-SRO is always safe and guarantees liveness during sufficiently long periods of synchrony (formally, after some unknown Global Stabilization Time). During such periods, Pompē-SRO enforces both ε-Ordering Equality and Δ-Ordering Separation. Pompē-SRO extends Pompē by injecting SRO-generated noise into the fault-tolerant timestamps used to order client requests.

Our evaluation shows that Pompē-SRO can be effective in mitigating front-running and sandwich attacks while incurring moderate performance overhead. For example, when adding random noise sampled from $[0, 5\Delta_{net}]$, where Δ_{net} bounds message delay during synchronous periods (*e.g.,* $\Delta_{net} = 400$ms in Solana Alpenglow [51]), Pompē-SRO maintains ε below 5%—a threshold considered acceptable in other equal-opportunity contexts [2, 3]—while matching Pompē's throughput. The added noise increases the median and 99th-percentile latency by $1.12\times$–$1.42\times$ in a geo-distributed deployment spanning 12 cities.

2 Equal opportunity

Consider a system in which clients invoke commands, and nodes (*i.e.,* servers) aim to produce a total order that reflects the time at which commands are invoked—earlier commands should precede later ones.

2.1 Motivating equal opportunity

Informally, if two commands have the same invocation time, equal opportunity requires the two possible orders to be equally likely in the system output. Similarly, if three commands are invoked at the same time, all six possible orders should be equally likely. To illustrate how equal opportunity is violated in practice, we analyze publicly available traces from Ethereum.

Bias under two concurrent invocations. Violations of equal opportunity with two concurrent invocations not only indicate bias but can also enable front-running [37, 75]. Empirical studies show that both phenomena have significantly affected the allocation of $89M over 32 months in the Ethereum blockchain [68]. Although Ethereum has since transitioned from proof-of-work to proof-of-stake [11], the ordering problem we study depends only on the semantics of client requests; we defer a discussion of proof-of-stake to Section 4.

In Ethereum, an invocation originating in Europe is more likely to precede a simultaneous one from Australia because a larger fraction of Ethereum nodes are located in Europe. If the system orders European invocations before Australian ones more than half the time, we say it is *biased* toward Europe. Geographic location is an *irrelevant* feature (a protected characteristic) under equal-opportunity legislation [1], yet such bias has been observed in Ethereum and other blockchains [61].

This bias can lead to significant consequences in applications such as blockchain liquidations. In traditional finance, liquidation occurs when an asset is sold below market value to repay debt, creating profit opportunities for buyers. Similar

Output order	Victim's profit	Attacker's profit
i_2, i_1, i_3	-$500	$800
i_3, i_1, i_2	$700	-$400
other order	$300	$0

Figure 1: An example of a sandwich attack in which a victim invokes i_1 and an attacker invokes i_2 and i_3. The example is drawn from a decentralized exchange; details of the associated values are explained in Section 5.2.

$$I_{gt} = \{\langle c_1, \langle 5\text{pm}\rangle\rangle_{i_1}, \langle c_2, \langle 5\text{pm}\rangle\rangle_{i_2}, \langle c_3, \langle 5{:}01\text{pm}\rangle\rangle_{i_3}\}$$
$$\mathcal{P}_1 = \langle\{\langle i_1, 1\rangle, \langle i_2, 2\rangle, \langle i_3, 3\rangle\}\rangle$$
$$\mathcal{P}_2 = \langle\{\langle i_2, 1\rangle, \langle i_1, 2\rangle, \langle i_3, 3\rangle\}\rangle$$

Figure 2: In a system that uses invocation time as the only relevant feature and sequence numbers as ordering indicators, impartiality requires preference profiles $\mathcal{P}_1$ and $\mathcal{P}_2$ to be equally likely.

mechanisms exist in blockchains, where they provide a common profit mechanism in the stablecoin [6] and lending [9] applications. The buyer whose command is ordered first on the blockchain is typically the one to realize the profit.

Consider two clients, one in Europe and one in Australia, invoking a liquidation simultaneously, trying to secure a $200K profit. If the system orders the European request first with probability 0.75, then the expected profit is $0.2M $\times$ 0.75 = $150K for the European client and $0.2M $\times$ 0.25 = $50K for the Australian client. Thus, irrelevant features can lead to substantial disparities among clients that should be treated equally. Similarly, a client intent on becoming the beneficiary of a liquidation's profits could leverage faster network devices to further bias ordering and front-run other clients.

Three-invocation sandwich attack. Violations of equal opportunity under three simultaneous invocations enable sandwich attacks [82]. Empirical studies estimate such attacks extracted more than $174M over 32 months on Ethereum [68].

Figure 1 illustrates a sandwich attack observed in a decentralized exchange. After a victim submits command i_1, an attacker submits i_2 and i_3. The attacker profits only if the system outputs the order i_2, i_1, i_3. Thus, a successful attack relies on making this particular ordering significantly more likely than equal opportunity would permit. A common strategy is to privately relay i_2 and i_3 to colluding nodes that then exclusively propose blocks containing the order i_2, i_1, i_3 [68]. While attackers are free to choose their trading strategy, and different strategies may lead to different expected profits, the system should not allow them to influence the probabilities of its possible outputs. Under equal opportunity, all six permutations of the three commands should occur with equal (or near-equal) likelihood.

2.2 A model for equal opportunity

We model equal opportunity in terms of two classical principles from the economics literature [79]: *impartiality* and *consistency*. In our setting, impartiality captures the idea that irrelevant features (*e.g.*, client geolocation) should not affect the order of commands, while consistency requires that introducing new commands should not alter the relative ordering of existing ones. Together, these principles suggest a natural mechanism for enforcing equal opportunity: a *point system*.

We now formalize these principles in the client-server setting and the ordered consensus framework [81].

Client invocation. An *invocation* is a pair $\langle c, \vec{f_r}\rangle$, where c is a command and $\vec{f_r}$ is a vector of relevant features, *i.e.,* the only attributes that should influence the command's position in the order. Irrelevant features include, for example, client identity, geographic location, wealth, and network connectivity. In blockchain settings, relevant features typically include invocation time and transaction fee. An *invocation profile*, denoted I, is a set of invocations.

Node preference. Given an invocation profile I, each node observes the invocations and expresses a *preference*. A preference is a set of pairs $\langle i, o\rangle$, where i is an invocation and o is an ordering indicator (*e.g.*, a timestamp or score). Intuitively, ordering indicators encode how a node would rank invocations. A *preference profile*, denoted $\mathcal{P}$, is a vector of preferences from all correct nodes. We say that $\mathcal{P}$ is *well-formed* under I if it references only invocations in I.

Chance relation. In practice, due to nondeterminism (*e.g.*, network delays or scheduling effects), a given set of invocations may result in different preference profiles. We assume a ground-truth invocation profile I_{gt} and model the uncertainty in the preferences of correct nodes through *chance relations*. For two preference profiles $\mathcal{P}_1$ and $\mathcal{P}_2$ well-formed under I_{gt}, $\mathcal{P}_1 \succ_c \mathcal{P}_2$ denotes that $\mathcal{P}_1$ is more likely to occur than $\mathcal{P}_2$, and $\mathcal{P}_1 \sim_c \mathcal{P}_2$ denotes that they occur with equal likelihood. Note that $\mathcal{P}_1 \sim_c \mathcal{P}_2$ is equivalent to $\neg(\mathcal{P}_1 \succ_c \mathcal{P}_2) \wedge \neg(\mathcal{P}_2 \succ_c \mathcal{P}_1)$.

Impartiality. Impartiality requires that invocations with identical relevant features be treated symmetrically. Formally, for any $i_1, i_2 \in I_{gt}$ with identical $\vec{f_r}$, a system is *impartial* if and only if swapping i_1 and i_2 in any preference profile does not change its likelihood. That is, if $\mathcal{P}_2$ is obtained from $\mathcal{P}_1$ by swapping i_1 and i_2, then $\mathcal{P}_1 \sim_c \mathcal{P}_2$.

Impartiality is the first pillar of equal opportunity and it is illustrated in Figure 2. If i_1 and i_2 share the same relevant feature (*e.g.*, invocation time), then their relative order must be independent of irrelevant features and $\mathcal{P}_1$ and $\mathcal{P}_2$ must be equally likely.

Consistency. Consistency, the second pillar of equal opportunity, requires that the relative ordering of invocations depend only on their own features and not on the presence or absence of other invocations. Figure 3 illustrates this principle: the

$$I_{gt} = \{\langle c_1, 1\text{pm}\rangle_{i_1}, \langle c_2, 1{:}01\text{pm}\rangle_{i_2}, \langle c_3, 1{:}05\text{pm}\rangle_{i_3}\}$$
$$\mathcal{P}_1 = \langle\{\langle i_1,1\rangle, \langle i_2,2\rangle\}\rangle \quad \mathcal{P}_3 = \langle\{\langle i_1,1\rangle, \langle i_2,2\rangle, \langle i_3,3\rangle\}\rangle$$
$$\mathcal{P}_2 = \langle\{\langle i_1,2\rangle, \langle i_2,1\rangle\}\rangle \quad \mathcal{P}_4 = \langle\{\langle i_1,2\rangle, \langle i_2,1\rangle, \langle i_3,3\rangle\}\rangle$$

Figure 3: An example of consistency where $I_1 = \{i_1, i_2\}$ and $I_2 = \{i_3\}$. This example originates from real-world legislation on equal opportunity in the context of kidney exchange, as explained in Section 6.

key idea is that the relative ordering of i_1 and i_2 should be unaffected by the presence or absence of a third invocation i_3. Consistency requires $\mathcal{P}_1 \succ_c \mathcal{P}_2 \iff \mathcal{P}_3 \succ_c \mathcal{P}_4$: the order of i_1 and i_2 should depend solely on their *own* relevant features.

Let $\mathcal{P}_1 =_I \mathcal{P}_2$ denote that $\mathcal{P}_1$ and $\mathcal{P}_2$ agree on invocations in I, and let $I_{gt} = \{I_1, I_2\}$ denote a partition. A system is *consistent* if and only if, for all preference profiles $\mathcal{P}_1, \mathcal{P}_2, \mathcal{P}_3, \mathcal{P}_4$ well-formed under I_{gt}, whenever $\mathcal{P}_1 =_{I_1} \mathcal{P}_3$, $\mathcal{P}_2 =_{I_1} \mathcal{P}_4$, $\mathcal{P}_1 =_{I_2} \mathcal{P}_2$, and $\mathcal{P}_3 =_{I_2} \mathcal{P}_4$, it holds that $\mathcal{P}_1 \succ_c \mathcal{P}_2 \iff \mathcal{P}_3 \succ_c \mathcal{P}_4$.

Point systems. A natural way to satisfy both principles is through a *point system*. In such a system, each invocation is assigned a score that depends only on its relevant features. Invocations are ordered by their scores, breaking ties uniformly at random.

A point system satisfies impartiality because identical relevant features yield identical scores, leading to symmetric treatment. It satisfies consistency because each score depends only on the invocation itself, and is therefore independent of other invocations.

Indeed, in a different model, point systems have been shown to be the only mechanisms that satisfy both impartiality and consistency [79]. Establishing an analogous result in the ordered consensus model remains an interesting direction for future work.

2.3 Correctness conditions for systems

In computer systems, invocation time is the primary relevant feature: commands are intended to be ordered according to *when* they are invoked. In principle, equal opportunity can be enforced by a point system that uses the invocation time as the score.

In practice, however, invocation time cannot be measured precisely. Nodes can only observe when they receive a command, and these observations are affected not only by the true invocation time but also by irrelevant features such as network latency and geographic location. As a result, a perfect point system based on exact invocation times is unattainable.

To accommodate this uncertainty, we relax the point-system ideal using two parameters, ε and Δ, and define the corresponding correctness conditions: ε-*Ordering Equality* and Δ-*Ordering Separation*. These can be viewed as approximate counterparts of earlier principles: ε-Ordering Equality relaxes impartiality by allowing small deviations from uniform ordering, while Δ-Ordering Separation relaxes consistency by tolerating bounded uncertainty in invocation times.

ε-Ordering Equality. This condition captures approximate fairness among invocations with identical relevant features. For any subset $I \subseteq I_{gt}$ in which all invocations share the same invocation time, let n denote the number of invocations in I. For any total order $\succ$ over I, $|Pr[\succ] - \frac{1}{n!}| \leq \varepsilon(n)$.

Here, $\varepsilon(n)$ bounds how far the distribution over permutations can deviate from uniform. When $\varepsilon = 0$, all permutations are equally likely, recovering impartiality. More generally, smaller values of ε ensure that invocations with identical relevant features have similar chances of appearing in any position, approximating how a point system breaks ties.

Δ-Ordering Separation. This condition captures the ability to respect differences in invocation time. It requires that for any invocations $i_1, i_2 \in I_{gt}$, if i_1 is invoked at least Δ time units before i_2, then i_1 precedes i_2 in the system output.

Intuitively, Δ represents the minimum separation required for the system to reliably distinguish between two invocations despite noise in timing observations. Smaller values of Δ correspond to stronger guarantees that earlier invocations appear earlier in the output.

Ideally, one would like $\varepsilon = 0$ and $\Delta = 0$, in which case both perfect fairness and perfect temporal ordering are achieved. This ideal is attainable (alas, only) in an idealized setting where all nodes are correct and observe invocation times without error, as captured by the following theorem.

Theorem 2.1. *If all nodes are correct and accurately measure the invocation time of all invocations, ordering equality and ordering separation with $\varepsilon = \Delta = 0$ can be achieved using a point system.*

Proof. Assuming all observations are accurate to a certain time unit (*e.g.*, microsecond), a straightforward approach is to assign each invocation a score equal to its observed invocation time. In such a point system, we have: (1) all invocations are sorted by this score in ascending order; and (2) the order of invocations with the same score is chosen uniformly at random from all permutations. (1) guarantees $\Delta = 0$ because invocations with different observed invocation times are guaranteed to be ordered in the order they are invoked. (2) guarantees $\varepsilon = 0$ because any two total orders have the same probability when chosen uniformly at random. □

2.4 Mitigating ordering-based attacks

A system that enforces equal opportunity can significantly mitigate ordering-based attacks, though it does not necessarily make them completely unprofitable. In existing systems, victims of front-running attacks are likely to incur losses because the system is biased toward clients with lower network

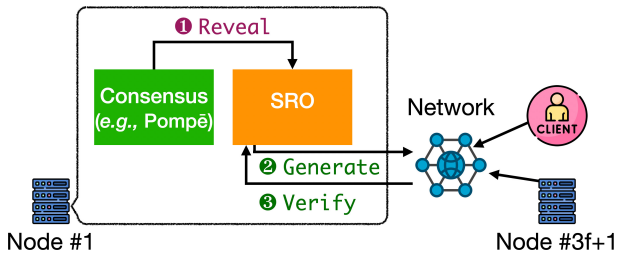


Figure 4: The protocol design overview of Pompē-SRO. The consensus component invokes the `Reveal` interface, which returns a random number. The TEE-based SRO returns the random number immediately. The TVRF-based SRO sends a message generated by the `Generate` interface and verifies messages received from other nodes. After receiving enough valid messages, the SRO will construct and return a random number to the consensus component.

latency. Under equal opportunity, this advantage is reduced: the gap between an attacker (who may have superior network access) and a victim narrows, as the system treats competing requests more uniformly. The ideal case of equal opportunity, with $\varepsilon = \Delta = 0$, provides the strongest possible mitigation that is agnostic to the semantics of client commands. Further reducing an attacker's advantage would require identifying and treating certain commands as adversarial, which risks introducing new sources of bias.

3 Secret Random Oracle

The point-system abstraction suggests that randomness is essential for achieving ordering equality. Crucially, this randomness must remain hidden until after other system components commit to their outputs; otherwise, adversaries could exploit it to bias ordering. We address this requirement with the *Secret Random Oracle* (SRO), a system component that generates random values and reveals them only after a coordinated commit, preventing ordering decisions from depending on them.

Design overview. Figure 4 shows how the SRO integrates with a consensus system. We consider a BFT system with $n = 3f + 1$ nodes running a protocol such as Pompē [81]. When using a point-system approach, the consensus component invokes the SRO to obtain randomness via the `Reveal` interface.

We present two realizations of the SRO abstraction. A TEE-based design generates and reveals randomness locally without network communication. A cryptographic design based on Threshold Verifiable Random Functions (TVRF) requires communication: nodes invoke `Generate` to produce messages, use `Verify` to validate them, and combine sufficiently many valid messages to recover the random value.

The remainder of this section details the SRO interface and its guarantees (§3.1), presents two concrete realizations

```
Reveal(Int k, Set<Signature> s) → Int | Error
Generate(Int k) → Proof
Verify(Int k, Proof p, Int r) → Bool
```

Figure 5: The interface of a Secret Random Oracle (SRO).

based on trusted hardware (§3.2) and cryptography (§3.3), shows how SRO integrates into Pompē-SRO to enforce the desired ordering properties(§3.4), and quantifies the fundamental trade-off that exists in trying to achieve both ε-Ordering Equality and Δ-Ordering Separation(§ 3.5).

3.1 SRO interface and guarantees

The SRO provides a simple abstraction for generating and revealing random values in a coordinated and adversary-resistant manner.

We consider a system with n nodes, at most f of which may be Byzantine. Each node holds a private key and knows the public keys of all others. A function (not shown) maps an integer k to a pseudorandom number. The SRO interface, shown in Figure 5, consists of three operations:

- `Reveal(k, s)` returns the output of the pseudorandom function evaluated at k, once s distinct signatures indicate a quorum of nodes wishes to disclose it.
- `Generate(k)` takes in k and produces a cryptographic proof.
- `Verify(k, p, r)` given a proof p, checks whether the value r returned when invoking `Reveal` on k is correct.

`Reveal(k)` returns its output only after enough nodes signal their readiness to do so. This ensures that randomness remains hidden until it can no longer influence earlier decisions.

The SRO provides the following guarantees.

Uniqueness. All valid invocations of `Reveal(k)` return the same value. A set of signatures is valid if it contains valid signatures on k from at least $n-f$ distinct nodes. Invalid inputs cause `Reveal` to return an error.

Secrecy. For any k, let r be the value returned by `Reveal(k)`. If an adversary does not obtain valid signatures from $n-f$ nodes, it is computationally infeasible to distinguish with non-negligible probability r from a uniform random value.

Randomness. For all k, the value r returned by `Reveal(k)` is a non-error uniform random sample from its codomain.

Validity. If `Generate(k)` outputs proof p and `Reveal(k)` returns r, then `Verify(k, p, r)` returns `True`, and it is computationally infeasible to find some integer $r' \neq r$ such that `Verify(k, p, r')` also returns `True`.

3.2 An SRO design using trusted hardware

Trusted Execution Environments (TEEs) provide secure enclaves that protect the integrity and confidentiality of code and data. They have already been used in blockchain systems such as Ethereum's Sepolia testnet [12] and Solana's Block Assembly Marketplace [21]. Here, we show how they can be used to implement an SRO.

Initialization. Each node hosts a TEE running the SRO logic. During initialization, TEEs generate candidate random values (*e.g.*, via `RDRAND` in x86) and run a consensus protocol to agree on a shared secret one. This number remains confined within the TEEs and will be used as the `seed` of a pseudorandom function denoted as `RAND`.

Reveal, Generate and Verify. To invoke `Reveal`, a node forwards an integer and a set of signatures to its local TEE, which returns `RAND(seed, k)` if enough valid signatures are present; otherwise, it returns an error. Similarly, `Generate` forwards `k` to the TEE, which returns `HASH(RAND(seed, k))` where `HASH` is a one-way function. Lastly, `Verify` checks that its parameter `p` equals `HASH(r)`.

Correctness. Determinism of `RAND` ensures uniqueness, while its pseudorandomness guarantees randomness. Secrecy follows from TEE confidentiality, and validity from the collision resistance of the hash function.

We assume that initialization (*i.e.,* consensus on the random seed) eventually completes, thereby ensuring liveness: all invocations of SRO operations on correct nodes eventually terminate.

3.3 An SRO design using threshold VRF

A Threshold Verifiable Random Function (TVRF) is a cryptographic primitive that enables a set of nodes to jointly evaluate a pseudorandom function in a distributed and fault-tolerant manner [30,40]. Such constructions have been widely used in Byzantine agreement protocols for tasks such as committee selection and randomized consensus.

In our setting, a TVRF provides a natural way to implement the SRO abstraction: random values are generated collaboratively and remain unpredictable until sufficiently many nodes agree to disclose them.

Let `TVRF` denote a pseudorandom function mapping an integer k to a pseudorandom value. Figure 6 shows the interface for evaluating `TVRF(k)`. Each node invokes `Produce(k)` to generate a *share* using its private key. After collecting enough valid shares, a node invokes `Combine` to recover `TVRF(k)`. To handle Byzantine behavior, `Valid` checks the correctness of each share. TVRFs provide the following guarantees [30].

Robustness. For all integers `k`, it is computationally infeasible for an adversary to produce enough valid shares such that the integer output of `Combine` is not `TVRF(k)`.

Unpredictability. Without enough valid shares for `TVRF(k)`, an adversary cannot distinguish `TVRF(k)` from a uniform random sample with non-negligible probability.

```
Threshold VRF node-side function
Produce(Int k) → Share
Threshold VRF client-side functions
Combine(Set<Share> s) → Int | Error
Valid(Int k, Int node_id, Share s) → Bool
Modified node-side function for SRO
Produce(Int k, Set<Signature> s) → Share | Error
```

Figure 6: The interface of (modified) threshold VRF.

We make a slight modification to threshold VRF. For the `Produce` interface, we add a parameter: a set of signatures of `k` to be verified. If verification fails, correct nodes must return an error instead of a share. We can now design an SRO as follows: `Reveal` forwards the two parameters to all nodes, collects enough valid shares, and invokes `Combine`. `Generate` returns the set of public keys. `Verify` takes all the public keys and a set of shares as input and returns whether all the shares are valid (using the `Valid` interface).

Correctness. Under the random oracle model, a threshold VRF outputs cryptographically secure pseudorandom numbers. Robustness implies uniqueness and validity: for all integers `k`, combining enough valid shares can only produce `TVRF(k)`, since an adversary cannot create valid shares leading to a combined value different from `TVRF(k)`. Unpredictability implies secrecy because an adversary cannot obtain enough valid shares without valid signatures of `k` out of $n-f$ nodes, and without enough valid shares, it has no information about `TVRF(k)`. Liveness is ensured, assuming all network messages are eventually delivered.

3.4 Integrating an SRO with Pompē

We now introduce Pompē-SRO, which integrates an SRO with Pompē [81], a state-of-the-art ordered consensus protocol. The goal is to enforce the conditions in Section 2.3.

The Pompē protocol. Pompē employs any standard leader-based BFT SMR protocol (*e.g.,* [32]) that offers a primitive to agree on a value for each slot in a sequence of consensus decisions. Pompē transforms such a protocol into a new one that enforces correctness conditions on ordering.

Specifically, Pompē associates the slots with consecutive time intervals. For example, one slot may be associated with time interval $[t,\ t+500ms)$, and the next slot could be associated with interval $[t+500ms,\ t+1000ms)$. For simplicity, Pompē assumes that such a mapping from slots to time intervals is common knowledge. Within each consensus slot, the value to agree on is a set of $\langle c,\ ats\rangle$ pairs where c is a command and *ats* is called an *assigned timestamp*. Pompē provides two guarantees for this assigned timestamp: (1) *ats* falls in the time interval associated with the slot; (2) *ats* is

bounded by the lowest and highest timestamps provided by correct nodes. The commands are then ordered by their assigned timestamps.

Pompē requires $3f+1$ nodes and ensures guarantee (2) by collecting $2f+1$ timestamps from different nodes for each command c. The *median* of the $2f+1$ timestamps is chosen as the assigned timestamp *ats* for command c. Since there are at most f faulty nodes, the median of any $2f+1$ timestamps is upper-bounded and lower-bounded by timestamps provided by correct nodes.

Integrating Pompē with an SRO. Pompē-SRO augments Pompē by adding controlled randomness to the assigned timestamps of each command. After consensus is reached for slot `k`, a node obtains a quorum of signatures certifying the decision. Using `k` and these signatures, a correct node invokes `Reveal` and obtains a random seed to be used by the pseudorandom number generator. Crucially, this seed remains unknown until consensus is finalized, ensuring that it is independent of the consensus decision.

The pseudorandom number generator assigns an independent value r to each command, which is used to sample from a distribution D over $[0,\ \Delta_{noise}]$. Commands are then ordered by $ats + \texttt{sample}(D,r)$ instead of *ats* alone. Section 3.5 discusses how to select the distribution and Δ_{noise}.

Stability. A command is *stable* (or finalized) when it appears in the output ledger. In Pompē, stability coincides with reaching consensus. In Pompē-SRO, the added random noise may delay stability. Let the latest finalized slot correspond to interval $[ts, ts')$. A command c becomes stable when $ats + \texttt{sample}(D,r) < ts'$, ensuring that c and all preceding commands can be safely added to the output ledger.

Safety and liveness. Pompē guarantees the same safety and liveness properties as classic SMR protocols [32, 55]. Pompē-SRO inherits these guarantees and differs only in how commands are ordered. We now turn to proving the corresponding correctness conditions on ordering.

3.5 A trade-off between ε-Ordering Equality and Δ-Ordering Separation

The key question is how to choose the noise range Δ_{noise} and the distribution D used for random perturbation. We first derive a lower bound on $\varepsilon(2)$ for any fixed Δ, as a function of Δ_{noise} and the system parameter Δ_{net}, which bounds message delay, processing time, and clock drift under partial synchrony [38].

Partial synchrony model. One variant of the partial synchrony model introduces the *Global Stabilization Time* (GST) [38]. Specifically, there is an unknown time GST such that, after this time, there is a known bound Δ_{net} on network latency and processing time. The safety and liveness of Pompē are proven under this model. We now analyze the ordering properties of Pompē-SRO in the same model. More precisely:

Assumption 3.1. *After GST, if a command is invoked at time T, correct nodes will provide timestamps in the range $[T, T+\Delta_{net})$ for this command.*

Note that a simple clock synchronization protocol has been given as part of the Pompē protocol.

We now prove that any probability distribution D on $[0,\Delta_{noise}]$ achieves Δ-Ordering Separation with $\Delta = \Delta_{net} + \Delta_{noise}$ and, in the best case, provides ε-Ordering Equality with $\varepsilon(2) \geq \frac{1}{2(k+1)}$ (k is the natural number such that $k\Delta_{net} \leq \Delta_{noise} < (k+1)\Delta_{net}$). We also construct a discrete probability distribution that achieves this lower bound. These results establish a fundamental trade-off: improving ordering equality (reducing ε) necessarily worsens ordering separation (increasing Δ).

Lemma 3.1. *The assigned timestamp of a command is bounded by timestamps provided by correct nodes.*

Proof. See [81].

Theorem 3.1. *(Δ-Ordering Separation) After GST, for all invocations i_1 and i_2, if i_1 is invoked more than $\Delta_{net}+\Delta_{noise}$ time units earlier than i_2, then i_1 is guaranteed to appear before i_2 in the output.*

Proof. Suppose i_1 and i_2 are invoked at time T_1 and T_2. By Assumption 3.1 and Lemma 3.1, the assigned timestamp of i_1 is in the range $[T_1, T_1+\Delta_{net})$. After adding the random noise, the resulting timestamp is in the range $[T_1, T_1+\Delta_{net}+\Delta_{noise})$. Similarly, the resulting timestamp for i_2 is in the range $[T_2, T_2+\Delta_{net}+\Delta_{noise})$. Therefore, if $T_2 > T_1+\Delta_{net}+\Delta_{noise}$, i_2 will appear after i_1 in the output. □

Theorem 3.2. *($\varepsilon(2)$-Ordering Equality lower bound) After GST, for any probability distribution D on $[0,\Delta_{noise}]$, for all invocations i_1 and i_2 invoked at the same time, $|Pr[i_1 \prec i_2] - \frac{1}{2}| \geq \frac{1}{2(k+1)}$ where k is the natural number such that $k\Delta_{net} \leq \Delta_{noise} < (k+1)\Delta_{net}$.*

Proof. The key idea is that if two random samples fall within the same Δ_{net}-sized interval, their relative order can be determined by adversarially chosen timestamp differences.

Suppose i_1 and i_2 are both invoked at time T. By Assumption 3.1 and Lemma 3.1, the assigned timestamps are in the range $[T, T+\Delta_{net})$, which the adversary may pick. Let the two samples from D be $s_1, s_2 \in [0,\Delta_{noise}]$. Let k be the natural number defined above. We split the interval $[0,\Delta_{noise}]$ into $k+1$ buckets: $B_0 = [0,\Delta_{net}), B_1 = [\Delta_{net}, 2\Delta_{net}), \ldots, B_k = [k\Delta_{net}, \Delta_{noise}]$. If s_1 and s_2 fall into the same bucket, then $|s_1 - s_2| < \Delta_{net}$, and the adversary can decide the ordering of the two commands. Let the probability mass of D in those $k+1$ buckets be $pb_0, pb_1, \ldots, pb_k$. We know $\sum_i pb_i = 1$ and the probability that s_1 and s_2 fall into the same bucket is $\sum_i pb_i^2$. By the relation between the arithmetic mean and the quadratic mean, $\sum_i pb_i^2 \geq \frac{1}{k+1}$. Thus, the adversary can decide the ordering with at least a probability of $\frac{1}{k+1}$. This gives the lower bound $\varepsilon(2) \geq \frac{1}{2(k+1)}$. □

Theorem 3.3. *(Discrete distribution optimality) The above lower bound on* $\varepsilon(2)$ *can be achieved by setting D to be a discrete distribution that is uniformly random across the discrete points of* $0, \Delta_{net}, 2\Delta_{net}, \ldots, k\Delta_{net}$.

Proof. The proof of the lower bound above also shows us how to achieve Pareto-optimality, as the arithmetic mean only equals the quadratic mean if all elements are equal. This discrete distribution guarantees this lower bound of $\frac{1}{2(k+1)}$ for $\varepsilon(2)$, as two commands can only be ordered arbitrarily if they happen to get the same noise, which has a probability of exactly $\frac{1}{k+1}$ happening as the distribution has $k+1$ equally weighted outcomes. □

We defer the full proof of the general case $n > 2$ to the extended technical report [80]. As suggested in Section 2.1, real-world concerns about ordering could also arise due to violating ordering equality with three invocations (*i.e.,* enforcing $\varepsilon(3)$-Ordering Equality is necessary to mitigate sandwich attacks).

Theorem 3.4. *($\varepsilon(n)$-Ordering Equality) After GST, for all invocations $i_1..i_n$ invoked at the same time, for the discrete distribution constructed as above, for any total order $\succ$ of $i_1..i_n$,* $|Pr[\succ] - \frac{1}{n!}]| \leq \frac{\binom{k+n}{n}}{(k+1)^n} - \frac{1}{n!}$.

Proof. See [80].

Choosing the Δ_{noise} parameter. In practice, system designers already tune Δ_{net} in Pompē and similar protocols. To mitigate ordering bias (*e.g.*, front-running and sandwich attacks), they can instead tune Δ_{noise} to achieve a target ε.

For example, when considering attacks involving two or three invocations, designers may focus on $\varepsilon(2)$ and $\varepsilon(3)$. In legal contexts, $\varepsilon(2) \approx 0.05$ is often considered acceptable. The four-fifths rule [2,3] requires that selection probabilities differ by no more than 45% versus 55%, corresponding to $\varepsilon(2) = 0.05$. Similar thresholds have influenced fairness definitions in machine learning (*e.g.*, demographic parity [27]).

4 Implementation

We implement two SRO variants corresponding to the designs in Section 3.2 and Section 3.3.

For the TEE-based variant, we use Intel SGX [17]. Random values are computed as `SHA256(seed + k)`, where `seed` is established during initialization and `k` is the input to `Reveal`. The SGX SDK provides an optimized `SHA256` implementation for Linux [16]. We note that larger hash outputs (*e.g.*, `SHA512`) could provide stronger security, but are not supported in this environment. In practice, the integer type used by the SRO need not be limited to the standard 4-byte size: blockchain systems commonly use 32-byte or 64-byte values, which we also support.

For the TVRF-based variant, we build on an existing C++ implementation of threshold VRF [10, 40] and extend the `Produce` interface with signature verification. Random values are computed as `SHA512` hashes of a combined threshold of shares. The default configuration of this implementation uses the `mcl` cryptographic library [18] and the BN256 curve. Unlike in many BFT systems, cryptographic operations do not dominate the performance overhead in Pompē-SRO. As we show in Section 5, the overhead is primarily driven by the added randomness required to balance ε and Δ. Since cryptographic operations are not the bottleneck, we prioritize implementation simplicity when selecting cryptographic libraries.

In both variants, signature verification in `Reveal` relies on the `secp256k1` library [19], consistent with the Pompē implementation.

Discussion Permissionless blockchains, pioneered by Bitcoin [65], require nodes to know neither the system size nor the identities of the participants. In contrast, Pompē-SRO follows the standard BFT SMR model, where nodes know n, f, and participant public keys. In this sense, Pompē-SRO is not permissionless. Similarly, systems such as Ethereum and Solana are no longer permissionless after switching from proof-of-work to proof-of-stake. Pompē-SRO is well-suited for such proof-of-stake blockchains.

There is, however, a minor difference between the models used in proof-of-stake and traditional BFT SMR systems. With proof-of-stake, different nodes can *stake* different amounts of native currency (*e.g.,* ETH tokens). In this setting, the BFT condition $n = 3f + 1$ maps to a bound of at most 33% adversarial stake, while quorums correspond to 67% of total stake. We incorporate this interpretation in our evaluation by adapting baseline protocols accordingly.

Finally, our current Pompē-SRO prototype does not support the possibility that nodes may join or leave the blockchain. However, Pompē-SRO could easily be extended to support such dynamic participation by integrating standard techniques used for membership changes in traditional SMR protocols.

5 Experimental evaluation

We ask three main questions in our evaluation: (1) Do the state-of-the-art protocols and Pompē-SRO address real-world problems like front-running or sandwich attacks? (2) What do the correctness conditions mean to attackers who try to launch ordering attacks? (3) What is the end-to-end performance of Pompē-SRO? Figure 7 shows the answers.

We choose three baselines: HotStuff [78], Pompē [81], and Themis [49] representing the state-of-the-art fairness notions. HotStuff adopts the *rotating leadership* notion. Themis guarantees a fairness property called γ-*batch-order-fairness*, and we choose $\gamma = 1$, which means that if all correct nodes receive i_1 before i_2, then i_1 should be ordered no later than i_2 in the output. Pompē adopts the fairness notion of *removing oligarchy*, meaning that the output order cannot be dictated by

State-of-the-art systems trying to provide fairness are all vulnerable to bias and front-running.	§5.1
Among all possible financial rewards, an attacker gets the *mean* reward in Pompē-SRO and the *max* in baselines.	§5.2
Pompē-SRO maintains the same end-to-end throughput as Pompē, and the latency increases by 1.12x-1.42x.	§5.3, 5.4

Figure 7: Table of evaluation results.

a single leader. In contrast, Pompē-SRO enforces the *equal opportunity* notion of fairness.

Setup and metrics. We run the three baselines and Pompē-SRO on 12 machines in CloudLab [13] (ds430, Intel Xeon E5-2630, 64GB memory, Ubuntu Linux 24.04 LTS). Since Intel Xeon E5-2630 does not support SGX, we run the TEE-based SRO on a separate machine with the Intel Xeon Silver 4410Y processor. We provide detailed instructions on running the experiments in an open-source artifact [22].

Statistics show that the top countries running Ethereum nodes are: US (33%), Germany (15%), France (6%), Finland (6%), Singapore (4%), UK (4%), Canada (3%), Japan (3%), Netherlands (3%), and Australia (3%) [14]. We thus start with a setup of 80 nodes where each node represents 1% in the statistics. For the US, we assume 11 nodes in Austin, 11 in San Francisco, and 11 in Washington, representing the central, western, and eastern US. For the other countries, we assume that all the nodes are in one of their major cities. There are thus 12 cities in this setup, and we map these 12 cities to the 12 machines in CloudLab, emulating the latency between different cities with the Linux traffic control (`tc`) utility. The latency information is from WonderNetwork [15]. We assume $\Delta_{net} = 400ms$ in the partial synchrony model: $400ms$ is used in Solana's latest consensus protocol [51], and $400ms$ is higher than the maximum latency in this setup (*i.e.,* $296ms$ from Canberra in Australia to Oulu in Finland).

We then introduce proof-of-stake, which reduces resource contention on the machines. Specifically, instead of running 11 instances of the server-side software on one machine (*e.g.,* the machine mapped to Washington), we run only one instance with $11/80$ of the total stake. As explained in Section 4, this is essentially the same as the standard SMR model with 80 nodes that we just described. Moreover, similar to real-world blockchains like Solana, we assign slot leaders based on their stake. For example, in HotStuff, this machine will be the leader for $11/80$ of all the consensus slots.

Given a set of commands, the key metric for fairness is the difference between the probabilities of two permutations in the output. For example, given two commands invoked at the same time in Washington and London, the probability of the London one appearing first in the output is 0.475 (denoted as $Pr[L \prec W] = 0.475$) in our measurement of HotStuff. Therefore, $Pr[L \prec W] - Pr[W \prec L] = 0.475 - 0.525 = -0.05$ is shown in the first row of Figure 8. Since $|Pr[L \prec W] - 1/2| = 0.025$ is lower than 0.05, HotStuff treats the two clients from London and Washington equally if we target $\varepsilon(2) = 0.05$. We will show that HotStuff is biased when considering clients from some other cities.

To answer the question about end-to-end performance, we measure the latency and throughput of HotStuff, Pompē and Pompē-SRO in the 12-city setup. We assume that clients are evenly distributed across the 12 cities, and we increase the number of clients until the system is saturated. Every client invokes commands in a closed loop (*i.e.,* it waits for enough responses of its currently outstanding command before invoking the next one), and before saturation, the throughput increases as the number of clients increases. We aim to measure the overhead of Pompē-SRO over Pompē and compare their performance with HotStuff. As for batching, we use $2000ms$ as the time interval associated with each consensus slot in Pompē and Pompē-SRO. We disabled batching in HotStuff and in the ordering phase of Pompē and Pompē-SRO.

5.1 Bias and front-running

Figure 8 shows that geographical bias could be significant in the baselines. The output order produced by Pompē and Themis is deterministic. When commands are invoked at the same time, Pompē always produces London $\prec$ Munich $\prec$ Washington $\prec$ Tokyo, reflecting the order of the median timestamps obtained by clients from these cities. Themis produces Washington $\prec$ London because more than half (42/80) of the total stake receives the Washington command before a simultaneous one from London. Since 42/80 is close to one half, the γ-batch-order-fairness property of Themis does not require Themis to order the Washington command first. In contrast, 62/80 of the total stake receives the London command before a simultaneous one from Munich, and γ-batch-order-fairness requires Themis to be biased towards London because 62/80 is higher than a threshold derived from the γ parameter. While γ-batch-order-fairness sometimes requires the system to be biased, the notion of removing oligarchy does not require bias despite the fact that Pompē is significantly biased.

HotStuff rotates leadership, and the invocation from Tokyo would have a chance to appear first in the ledger when a Tokyo node serves as the leader. However, this chance is still low compared to Munich invocations. Specifically, we find that $Pr[\mathrm{M} \prec \mathrm{T}] = 0.74$, leading to the $0.74 - 0.26 = 0.48$ in the last row of Figure 8. This difference increases to 0.83 when compared to Washington. While Tokyo and Washington are geographically distant, bias can happen between nearby cities. London and Munich are both in Europe, but HotStuff is still biased towards London, as shown in the third row.

Figure 9 shows how Pompē-SRO reduces geographical bias. After adding a random noise sampled uniformly from

Baselines	HotStuff	Pompē	Themis
$Pr[L \prec W] - Pr[W \prec L]$	-0.05 ✓	1	-1
$Pr[W \prec T] - Pr[T \prec W]$	0.83	1	1
$Pr[L \prec M] - Pr[M \prec L]$	0.48	1	1
$Pr[M \prec T] - Pr[T \prec M]$	0.48	1	1

Figure 8: Geographical bias in HotStuff, Pompē, and Themis. W, L, M, and T stand for Washington, London, Munich, and Tokyo. For two simultaneous invocations from two cities, $Pr[\text{A} \prec \text{B}]$ stands for the probability of the invocation from city A being the first in the system output. The checkmark for HotStuff means that bias is controlled by $\varepsilon(2) = 0.05$.

Pompē-SRO	$\Delta_{noise} = \Delta_{net}$	$\Delta_{noise} = 5 * \Delta_{net}$
$Pr[L \prec W] - Pr[W \prec L]$	0.24	0.05 ✓
$Pr[W \prec T] - Pr[T \prec W]$	0.31	0.07 ✓
$Pr[L \prec M] - Pr[M \prec L]$	0.07 ✓	0.01 ✓
$Pr[M \prec T] - Pr[T \prec M]$	0.46	0.10 ✓

Figure 9: Geographical bias measured in Pompē-SRO with $\Delta_{noise} = \Delta_{net}$ (*i.e.,* 400ms) and $\Delta_{noise} = 5 * \Delta_{net}$ (*i.e.,* 2000ms).

$[0, \Delta_{net}]$ (*i.e.,* $[0, 400ms]$) to the median timestamp of Pompē, clients from London and Munich will be treated equally. With a target $\varepsilon(2) = 0.05$, system designers could choose $\Delta_{noise} = 5 * \Delta_{net}$ (*i.e.,* $2000ms$), and the worst-case bias across these four cities will be effectively controlled. Specifically, given the definition of ε-Ordering Equality, $\varepsilon(2) = 0.05$ means that $|Pr[M \prec T] - 1/2|$ should be at most 0.05. Therefore, the 0.10 in Figure 9 means that $Pr[M \prec T] = 0.55$ and $Pr[T \prec M] = 0.45$, satisfying the constraint put by $\varepsilon(2) = 0.05$. In real-world deployments, Δ_{noise} can be chosen by considering clients with the highest network latency. Pompē-SRO does not provide equal opportunity to clients with unbounded network latency to the system. Indeed, even liveness cannot be guaranteed without a bound on network latency [39], and equal opportunity is meaningless without liveness.

Front-running typically occurs when one client has lower network latencies to a majority of the nodes than the other clients, as London or Washington do in this experiment. By adding randomness, Pompē-SRO could give clients more even chances of obtaining the $89M profit in the liquidation events (explained in Section 2.1). Equal opportunity also plays an important role in mitigating sandwich attacks, which is closely related to front-running.

5.2 Sandwich attacks

To explain sandwich attacks, we start with some background on decentralized exchange. An exchange maintains a pool of some token A (*e.g.,* USD) and some token B (*e.g.,* CNY). For example, people traveling from the US to China may put some USD into the pool and take some CNY away. Similar pools exist on blockchains, and the trading volume of Uniswap, a decentralized exchange application running on Ethereum, has exceeded one trillion dollars [7].

Financial impact	Victim	Attacker
HotStuff / Pompē / Themis	-$500	$800
Pompē-SRO	$233	$67

Figure 10: The expected financial rewards to the attacker and victim within the example sandwich attack shown in Figure 1.

Pools in these exchanges follow a constraint: *amount of tokenA* $*$ *amount of tokenB* $=$ *constant*, which is called the automated market maker (AMM) approach [20]. Suppose the constant is 1800; the number of tokens A and B in the pool could be, for instance, $\langle 45, 40\rangle$, $\langle 60, 30\rangle$, or $\langle 75, 24\rangle$. Say $\langle 75, 24\rangle$ is the current status, and Alice needs 15 token A. Alice can put 6 token B into the pool and take 15 token A away so that the pool state moves to $\langle 60, 30\rangle$.

The sandwich attack works as follows. After seeing Alice's transaction, an attacker, Bob, first buys 15 token A, moving the pool status to $\langle 60, 30\rangle$. Alice now needs to pay 10 (instead of 6) token B to exchange for 15 token A, moving the pool status to $\langle 45, 40\rangle$. Bob can then exchange 15 token A back to 10 token B, making a profit of 4 token B. The three steps reflect the three invocations in Figure 1. If the market value of the tokens are $100 and $200, respectively, we will get the dollar values in Figure 1. Specifically, if the attack succeeds, the victim will pay 10 token B (*i.e.,* $2000) in exchange for 15 token A (*i.e.,* $1500), losing $500 as shown in the first row of Figure 1. Without the attack, the victim will pay 6 token B in exchange for 15 token A, making a profit of $300.

A common way of conducting such sandwich attacks is to use faster network connections [57]. The attacker is typically co-located with the victim and, after observing a trading command from the victim (i_1 in Figure 1), the attacker uses faster network connections to send its own command (i_2 in Figure 1) to the nodes. Figure 10 shows the expected financial rewards in different systems.

In this experiment, we assume that an attacker co-located with the victim could deliver its command to all nodes in the system $10ms$ earlier than the victim. The attack deterministically succeeds in all state-of-the-art systems because they all follow the order in which nodes receive commands. In contrast, all 6 permutations are equally likely in Pompē-SRO, lowering the attacker's expected reward by an order of magnitude. Specifically, $\$233 = (-\$500 + \$700 + \$300 * 4)/6$ is the victim's expected reward, and $\$67 = (\$800 - \$400)/6$ is the attacker's expected reward. Given a much lower expected reward and a caveat of losing $400, Pompē-SRO effectively disincentivizes sandwich attacks.

5.3 Latency of the two SRO designs

Figure 12 shows the latency of the two SRO designs. We show two cases of |s|=0 and |s|=200, where |s| denotes the number of signatures to be verified in the second parameter of `Reveal`. When |s|=0, `Reveal` does not verify signatures,

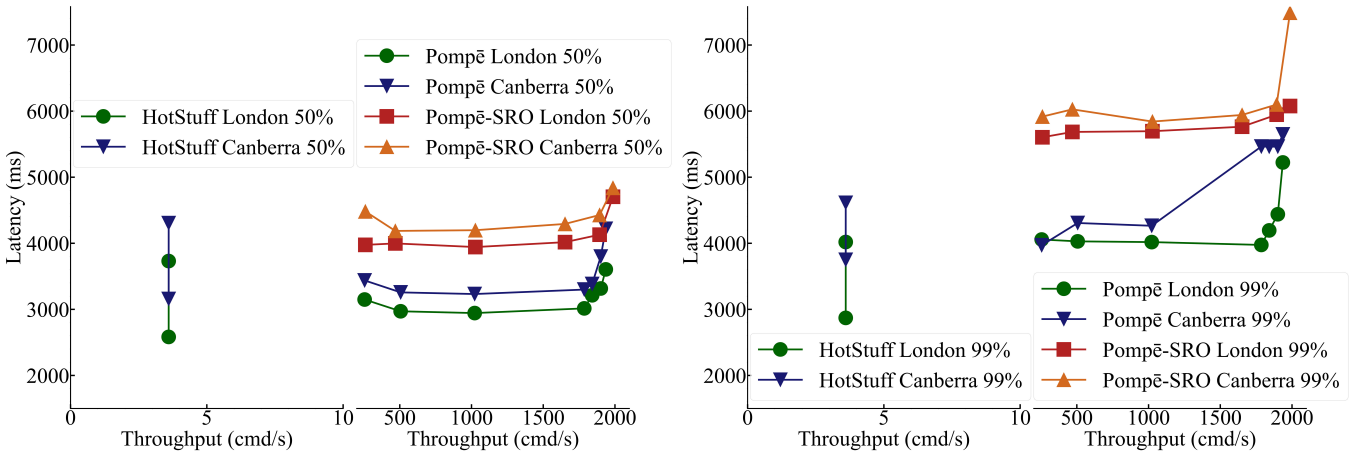


Figure 11: End-to-end performance of HotStuff, Pompē, and Pompē-SRO with $\Delta_{noise} = 5 * \Delta_{net}$ (*i.e.,* 2000*ms*) in the 12-city setup. Batching is disabled in HotStuff and in the ordering phase of Pompē or Pompē-SRO. For example, HotStuff handles 3.6 consensus slots per second, and if we enable a batch of 1000 commands for each slot, the throughput of HotStuff will become 3600 commands per second. Disabling batching is useful because, in practice, protocols only reach consensus on the slot hashes.

	|s|=0 (`base`)	|s|=200
TEE	3us	`base`+20.2ms
TVRF (67/100)	416+6346us	`base`+19.9ms
TVRF (133/200)	394+12330us	`base`+19.9ms

Figure 12: Latency of the `Reveal` interface in different SRO implementations. |s| denotes the number of signatures to be verified in the second parameter of `Reveal`. The numbers in the parentheses are the threshold and total number of nodes for TVRF. The `base` case of TVRF consists of two latency results for generating and combining shares.

and the latency is solely for generating random numbers. For the TEE variant, the latency consists of entering an SGX enclave and computing a `SHA256` function, which takes only 3*us*. For the threshold VRF variant, the latency consists of three parts: (1) generating shares, (2) collecting the shares over the network, (3) combining a threshold of shares. The results in Figure 12 show (1) and (3). Under a setup of 100 nodes and 67 as the threshold, the latency of generating a share is 0.4*ms*, and the latency of combining 67 shares is 6.3*ms*. When moving to a setup of 200 nodes with 133 as the threshold, the latency of generating a share stays at 0.4*ms*, while the latency of combining the shares doubles to 12*ms*. Lastly, when $|s| = 200$, the latency of verifying the 200 signatures is about 20*ms*, both within and outside an SGX enclave.

These results show a trade-off between performance and decentralization. The SRO based on SGX has a much lower latency, making it more practical, but it requires trusting Intel, a centralized party. In the following experiments, we choose performance in this trade-off and use the TEE variant of SRO, since TEE has been used in Ethereum's Sepolia testnet [12] and Solana's Block Assembly Marketplace [21].

5.4 End-to-end performance of Pompē-SRO

Figure 11 shows the end-to-end performance of HotStuff, Pompē, and Pompē-SRO. We made three observations.

- Pompē-SRO keeps the same throughput as Pompē, and the latency increases by 1.12x-1.42x.
- Pompē and Pompē-SRO achieve a significantly higher throughput than HotStuff in the geo-distributed setup.
- The 99% latency of HotStuff is 287-592*ms* higher than its 50% latency, while the 99% latency of Pompē-SRO is 1672-1816*ms* higher than its 50% latency.

When deploying 120 clients in each city (*i.e.,* $120 * 12 = 1440$ clients in total), Pompē and Pompē-SRO reach their peak throughput: 1842 cmd/s for Pompē and 1893 cmd/s for Pompē-SRO. When considering clients from London, the 50% latency in Pompē-SRO is *1.29*x of the 50% latency in Pompē, and the 99% latency is *1.42*x of the 99% latency in Pompē. For Canberra clients, the increase ratios are *1.31*x and *1.12*x for the 50% and 99% latencies, respectively. This is how we derive the last statement in Figure 7.

HotStuff's performance issue in a geo-distributed setup has already been reported in prior work [81]. Specifically, with rotating leadership in HotStuff, the leader of the current slot needs to wait for enough votes on the previous slot before proposing for the current slot. Such waiting results in a low frequency of consensus proposals in HotStuff. Pompē and Pompē-SRO do not suffer from such waiting, although they reuse HotStuff for consensus. Specifically, their leaderless ordering phase proposes commands continuously without waiting, and consensus only happens every 2000*ms* for all commands proposed by the leaderless ordering phase in a 2000*ms* time window.

Lastly, the gap between the 50% and 99% latencies in Pompē-SRO is obviously larger than that gap in HotStuff, and two factors cause such a larger gap. First, since consensus happens every 2000*ms*, there is a delay between the time when a client invokes a command and the time when consensus happens. Second, due to random noise, it takes longer for a command to become stable in Pompē-SRO, as explained in Section 3.4. Both factors enlarge the gap between the 50% and 99% latencies in Pompē-SRO.

6 Related work

Traditional BFT SMR systems. There is a long line of work on traditional BFT SMR systems, including [28, 32, 33, 35, 36, 43, 44, 46, 52, 53, 58–62, 67, 73, 74, 77]. These works focus on enforcing safety and liveness, removing or constraining Byzantine influence, and improving performance or theoretical complexity, among other goals. Our work focuses on how to choose the output order in SMR, which is not considered by the traditional specification. Some works [28, 34, 47, 56] use trusted hardware to increase the ratio of Byzantine nodes that the system can tolerate. Other works [30, 40, 42] use randomness to elect a committee or achieve safety and liveness in a fully asynchronous model. Unlike these works, our work uses trusted hardware to provide a fault-tolerant source of randomness and applies it to ensure equal opportunity.

Rotating leadership. Some works adopt a leader and rotate the leadership frequently, but they focus on reducing theoretical complexity or preventing faulty leaders from degrading performance. Aardvark [36] employs periodic leader changes to ensure a certain degree of performance in the presence of faulty leaders. Aardvark sets an expectation on the minimal throughput a leader must ensure and triggers a leader change if the current leader fails to meet this expectation. Unlike Aardvark, HotStuff [78] employs leader rotation and optimizes the communication complexity. Specifically, HotStuff's communication complexity is linear in the number of nodes, which makes it more suitable for blockchains. Adopted by Diem [8], rotating leadership in HotStuff aims to provide some sense of fairness in a permissioned blockchain. Our work instead specifies and enforces a concrete notion of fairness, and our evaluation results show that rotating leadership could cause significant bias in a real-world deployment.

Removing oligarchy. *Leaderless* protocols argue against having a leader node who can unilaterally decide the output order. EPaxos [63], or Egalitarian Paxos, is an SMR protocol that attempts to make the system egalitarian. While the concept of egalitarianism is closely related to equal opportunity, EPaxos does not specify or enforce egalitarian ideals except being a leaderless protocol. Byzantine oligarchy [81] is the first attempt to specify the goal of leaderless protocols in the context of ordering, and Pompē is the first leaderless protocol that provably removes a Byzantine oligarchy. To achieve this, Pompē requires a client to collect timestamps from a quorum of nodes, and the median is used to order a command. However, our evaluation shows that using such a median timestamp could make the system even more biased than prior approaches such as HotStuff.

Decentralized first-come-first-served. Some recent works define and enforce fairness concepts related to first-come-first-served [31, 48–50, 64, 69]. Specifically, these protocols enforce variants of the receive-order-fairness property [50], which essentially says that if a majority of the nodes receive an invocation first, it should be ordered first in the output. We argue that this property can be unfair because, without distinguishing relevant features from irrelevant ones, it can amplify systemic bias in real-world blockchains, as shown in the evaluation section.

While these works enforce specific properties related to first-come-first-served, the framework of ordered consensus makes it possible to prove that it is impossible, in general, to prevent Byzantine replicas from manipulating the order (*e.g.,* from conducting front-running) [81]. This result is inspired by Arrow's impossibility theorem [26] and the Gibbard-Satterthwaite impossibility theorem [41, 70] from the field of social choice theory. In the past two decades, computer scientists became interested in social choice, leading to the creation of the field of computational social choice [29].

Game theory. The BAR model [23, 24] explores how to connect Byzantine fault tolerance to game theory. The core of the connection is adopting the Nash theorem [66], which states that every normal-form game must have an equilibrium. The Nash theorem connects the rationality model [71, 76] with the Brouwer fixed-point theorem (*i.e.,* a fixed point is interpreted as an equilibrium). We find the concept of equal opportunity and its violations more prevalent in real-life scenarios than the Nash equilibrium. We reuse the key concept of expected value from the rationality model when explaining violations of equal opportunity in Section 2.1. We are also inspired by one of the axioms at the core of the rationality model [71] when defining consistency in Section 2.2. However, unlike BAR, we do not reuse the concept of Nash equilibrium in this work.

Another line of work studies how to eliminate sandwich attacks in blockchains within a game-theoretic model [25]. Similar to our work, this work also argues that randomness is the key to addressing the problem. The proposed solution only randomizes commands within a block, making it vulnerable to destructive front-running attacks described in an empirical study [68]. Destructive front-running means that, in leader-based protocols, the leader of a slot could front-run a command by excluding it from this slot, so this victim command can only be included in a later slot.

Equal opportunity in real-life scenarios. The principles of impartiality and consistency have been scrutinized in the context of how society allocates resources. In his book [79],

Young studies the two principles in a variety of contexts, from employment to kidney exchange, and discusses how they are embodied in key pieces of legislation [1, 2, 4, 5]. The book includes a proof that a point system is the only mechanism that satisfies both impartiality and consistency. Our work adopts this book's approach: we use the invocation time as the score for ordering and use a Secret Random Oracle to break ties.

Kidneys for transplant used to be exchanged through a free market, and wealthy patients had a better chance of getting kidneys. This raised fairness concerns and led to legislation that transferred the operation to the government to overcome such bias towards wealthy patients [4]. While this law enforced *impartiality* between the rich and poor, *consistency* became a concern. The first algorithm proposed was inconsistent, and the order of two patients getting kidneys could be switched due to a third patient joining the system. As a result, a new algorithm was proposed as an amendment that enforces the consistency principle [5]. Details of the relevant features and the point system in this case have been discussed in Chapter 2 of Young's book [79].

Financial regulation laws require financial exchanges to be impartial to all traders. A recent exposé [57], however, has concluded that high-frequency traders have routinely engaged in market-exploiting behaviors, such as aggressive latency optimizations and front-running by exploiting their location or the availability of fast network connections—which our framework would classify as irrelevant features.

In the Olympic Games, the order of athletes is decided, and equal opportunity is required. The Olympic rules specify which relevant features should be measured. Similar to geolocation, a key irrelevant feature is national origin, and judgments should not be biased toward any nation. Besides, due to cognitive bias, a judge may make correlated decisions when judging a sequence of observations. For example, in diving, athletes dive alternately, and an athlete's performance may impact the scores given to the next athlete [54]. Judges typically need professional training to combat such implicit bias and follow the consistency principle.

7 Conclusion

This paper introduces a model for equal opportunity, a notion of fairness based on the distinction between relevant and irrelevant features. Existing protocols—including ones attempting to provide some fairness—can be significantly biased and vulnerable to ordering-based attacks.

We design, implement, and evaluate Pompē-SRO, a new ordered consensus protocol that guarantees two correctness conditions for equal opportunity, ε-Ordering Equality and Δ-Ordering Separation. Pompē-SRO effectively mitigates the well-known ordering-based attacks, given that eliminating such attacks in the presence of Byzantine influence has been proved impossible by prior work.

Acknowledgments

We thank the anonymous shepherd and OSDI reviewers for their insightful and constructive comments. The initial steps towards a theory model for equal opportunity benefited from conversations with Joseph Halpern and his book on theories of uncertainty [45]. Roger Wattenhofer helped us confirm our understanding of Solana's new consensus protocol. This work was supported in part by a Facebook Fellowship.

References

[1] Civil Rights Act. https://www.congress.gov/bill/88th-congress/house-bill/7152, 1964.

[2] Equal Employment Opportunity Act. https://www.eeoc.gov/history/equal-employment-opportunity-act-1972, 1972.

[3] Uniform guidelines on employee selection procedures. https://www.ecfr.gov/current/title-29/subtitle-B/chapter-XIV/part-1607, 1978.

[4] National Organ Transplant Act. https://www.congress.gov/bill/98th-congress/senate-bill/2048, 1984.

[5] National Organ Transplant Program Extension Act. https://www.congress.gov/bill/101st-congress/house-bill/5146, 1990.

[6] MakerDAO. https://makerdao.com, 2017.

[7] Uniswap. https://uniswap.org/, 2018.

[8] Diem. https://www.diem.com/en-us/, 2019.

[9] Aave liquidity protocol. https://aave.com/, 2020.

[10] A threshold VRF implementation. https://github.com/fetchai/research-dvrf, 2020.

[11] The Merge in Ethereum. https://ethereum.org/roadmap/merge/, 2022.

[12] Block building inside SGX. https://writings.flashbots.net/block-building-inside-sgx, 2023.

[13] CloudLab. https://cloudlab.us/, 2023.

[14] Ethereum mainnet statistics. https://ethernodes.org/countries?synced=1, 2023.

[15] Global ping statistics: Ping times between WonderNetwork servers. https://wondernetwork.com/pings, 2023.

[16] Intel software guard extensions for Linux OS. https://github.com/intel/linux-sgx, 2023.

[17] Intel Software Guard Extensions (SGX). https://www.intel.com/content/www/us/en/architecture-and-technology/software-guard-extensions.html, 2023.

[18] mcl: A portable and fast pairing-based cryptography library. https://github.com/herumi/mcl, 2023.

[19] The secp256k1 library from Bitcoin. https://github.com/bitcoin-core/secp256k1, 2023.

[20] What are Automated Market Makers (AMMs)? https://chain.link/education-hub/what-is-an-automated-market-maker-amm, 2023.

[21] Solana's Block Assembly Marketplace. https://bam.dev/, 2025.

[22] Benchmark for equal opportunity in ordered consensus. https://github.com/yhzhang0128/cob, 2026.

[23] Ittai Abraham, Lorenzo Alvisi, and Joseph Y. Halpern. Distributed computing meets game theory: Combining insights from two fields. *ACM Sigact News*, 42(2):69–76, 2011.

[24] Amitanand S. Aiyer, Lorenzo Alvisi, Allen Clement, Mike Dahlin, Jean-Philippe Martin, and Carl Porth. BAR fault tolerance for cooperative services. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, pages 45–58, 2005.

[25] Orestis Alpos, Ignacio Amores-Sesar, Christian Cachin, and Michelle Yeo. Eating sandwiches: Modular and lightweight elimination of transaction reordering attacks. arXiv:2307.02954, 2023.

[26] Kenneth J. Arrow. *Social choice and individual values*, volume 12. Yale University Press, 1951.

[27] Solon Barocas, Moritz Hardt, and Arvind Narayanan. *Fairness and Machine Learning: Limitations and Opportunities*. MIT Press, 2023.

[28] Johannes Behl, Tobias Distler, and Rudiger Kapitza. Hybrids on steroids: SGX-based high performance BFT. In *Proceedings of the ACM European Conference on Computer Systems (EuroSys)*, pages 222–237, 2017.

[29] Felix Brandt, Vincent Conitzer, Ulle Endriss, Jérôme Lang, and Ariel D. Procaccia. *Handbook of computational social choice*. Cambridge University Press, 2016.

[30] Christian Cachin, Klaus Kursawe, and Victor Shoup. Random oracles in Constantinople: Practical asynchronous Byzantine agreement using cryptography. In *Proceedings of the ACM Symposium on Principles of Distributed Computing (PODC)*, page 123–132, 2000.

[31] Christian Cachin, Jovana Mićić, Nathalie Steinhauer, and Luca Zanolini. Quick Order Fairness. arXiv:2112.06615, 2022.

[32] Miguel Castro and Barbara Liskov. Practical Byzantine fault tolerance and proactive recovery. *ACM Transactions on Computer Systems (TOCS)*, 20(4):398–461, 2002.

[33] Miguel Castro, Rodrigo Rodrigues, and Barbara Liskov. BASE: Using abstraction to improve fault tolerance. *ACM Transactions on Computer Systems (TOCS)*, 21(3):236–269, 2003.

[34] Byung-Gon Chun, Petros Maniatis, Scott Shenker, and John Kubiatowicz. Attested Append-only Memory: Making adversaries stick to their word. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, pages 189–204, 2007.

[35] Allen Clement, Manos Kapritsos, Sangmin Lee, Yang Wang, Lorenzo Alvisi, Mike Dahlin, and Taylor Riche. UpRight cluster services. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, pages 277–290, 2009.

[36] Allen Clement, Edmund Wong, Lorenzo Alvisi, Mike Dahlin, and Mirco Marchetti. Making Byzantine fault tolerant systems tolerate Byzantine faults. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 153–168, 2009.

[37] Philip Daian, Steven Goldfeder, Tyler Kell, Yunqi Li, Xueyuan Zhao, Iddo Bentov, Lorenz Breidenbach, and Ari Juels. Flash Boys 2.0: Frontrunning, transaction reordering, and consensus instability in decentralized exchanges. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, pages 910–927, 2020.

[38] Cynthia Dwork, Nancy Lynch, and Larry Stockmeyer. Consensus in the presence of partial synchrony. *Journal of the ACM (JACM)*, 35(2), 1988.

[39] Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. Impossibility of distributed consensus with one faulty process. In *Proceedings of the Symposium on Principles of Database Systems*, pages 1–7, 1983.

[40] David Galindo, Jia Liu, Mihair Ordean, and Jin-Mann Wong. Fully distributed verifiable random functions and their application to decentralised random beacons.

In *Proceedings of the IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 88–102, 2021.

[41] Allan Gibbard. Manipulation of voting schemes: a general result. *Econometrica: Journal of the Econometric Society*, pages 587–601, 1973.

[42] Yossi Gilad, Rotem Hemo, Silvio M. Micali, Georgios Vlachos, and Nickolai Zeldovich. Algorand: Scaling Byzantine agreements for cryptocurrencies. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, pages 51–68, 2017.

[43] Guy Golan Gueta, Ittai Abraham, Shelly Grossman, Dahlia Malkhi, Benny Pinkas, Michael K. Reiter, Dragos-Adrian Seredinschi, Orr Tamir, and Alin Tomescu. SBFT: A scalable decentralized trust infrastructure for blockchains. arxiv:1804/01626v1, 2018.

[44] Andreas Haeberlen, Petr Kouznetsov, and Peter Druschel. PeerReview: Practical accountability for distributed systems. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, pages 175–188, 2007.

[45] Joseph Y. Halpern. *Reasoning about Uncertainty*. MIT Press, 2017.

[46] James Hendricks, Shafeeq Sinnamohideen, Gregory R. Ganger, and Michael K. Reiter. Zzyzx: Scalable fault tolerance through Byzantine locking. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN)*, pages 363–372, 2010.

[47] Rüdiger Kapitza, Johannes Behl, Christian Cachin, Tobias Distler, Simon Kuhnle, Seyed Vahid Mohammadi, Wolfgang Schröder-Preikschat, and Klaus Stengel. CheapBFT: Resource-efficient Byzantine fault tolerance. In *Proceedings of the ACM European Conference on Computer Systems (EuroSys)*, pages 295–308, 2012.

[48] Mahimna Kelkar, Soubhik Deb, and Sreeram Kannan. Order-fair consensus in the permissionless setting. In *Proceedings of the ACM on ASIA Public-Key Cryptography Workshop*, pages 3–14, 2022.

[49] Mahimna Kelkar, Soubhik Deb, Sishan Long, Ari Juels, and Sreeram Kannan. Themis: Fast, strong order-fairness in Byzantine consensus. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, page 475–489, 2023.

[50] Mahimna Kelkar, Fan Zhang, Steven Goldfeder, and Ari Juels. Order-fairness for Byzantine consensus. In *Proceedings of the International Cryptology Conference (CRYPTO)*, pages 451–480, 2020.

[51] Quentin Kniep, Jakub Sliwinski, and Roger Wattenhofer. Solana Alpenglow consensus: Increased bandwidth, reduced latency. Technical report, Anza, 2025. https://www.anza.xyz/alpenglow-1-1.

[52] Ramakrishna Kotla, Lorenzo Alvisi, Mike Dahlin, Allen Clement, and Edmund Wong. Zyzzyva: Speculative Byzantine fault tolerance. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, pages 45–58, 2007.

[53] Ramakrishna Kotla and Mike Dahlin. High throughput Byzantine fault tolerance. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN)*, pages 575–584, 2004.

[54] Robin S.S. Kramer. Sequential effects in Olympic synchronized diving scores. *Royal Society Open Science*, 4(1), 2017.

[55] Leslie Lamport. The part-time parliament. *ACM Transactions on Computer Systems (TOCS)*, 16(2):133–169, 1998.

[56] Dave Levin, John R. Douceur, Jacob R. Lorch, and Thomas Moscibroda. TrInc: Small trusted hardware for large distributed systems. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 1–14, 2009.

[57] Michael Lewis. *Flash Boys: A wall street revolt*. W. W. Norton & Company, 2014.

[58] Jinyuan Li and David Maziéres. Beyond one-third faulty replicas in Byzantine fault tolerant systems. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 131–144, 2007.

[59] Jian Liu, Wenting Li, Ghassan O. Karame, and N. Asokan. Scalable Byzantine consensus via hardware-assisted secret sharing. *IEEE Transactions on Computers*, 68(1), 2019.

[60] Shengyun Liu, Paolo Viotti, Christian Cachin, Vivien Quéma, and Marko Vukolic. XFT: Practical fault tolerance beyond crashes. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 485–500, 2016.

[61] Shengyun Liu, Wenbo Xu, Chen Shan, Xiaofeng Yan, Tianjing Xu, Bo Wang, Lei Fan, Fuxi Deng, Ying Yan, and Hui Zhang. Flexible advancement in asynchronous BFT consensus. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, page 264–280, 2023.

[62] Jean-Philippe Martin and Lorenzo Alvisi. Fast Byzantine consensus. *IEEE Transactions on Dependable and Secure Computing*, 3(3):202–215, 2006.

[63] Iulian Moraru, David G. Andersen, and Michael Kaminsky. There is more consensus in egalitarian parliaments. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, pages 358–372, 2013.

[64] Ke Mu, Bo Yin, Alia Asheralieva, and Xuetao Wei. Separation is good: A faster order-fairness Byzantine consensus. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2024.

[65] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system. Whitepaper, October 2008.

[66] John Nash. Non-cooperative games. *Annals of Mathematics*, pages 286–295, 1951.

[67] Daniel Porto, João Leitão, Cheng Li, Allen Clement, Aniket Kate, Flavio Junqueira, and Rodrigo Rodrigues. Visigoth fault tolerance. In *Proceedings of the ACM European Conference on Computer Systems (EuroSys)*, pages 1–14, 2015.

[68] Kaihua Qin, Liyi Zhou, and Arthur Gervais. Quantifying blockchain extractable value: How dark is the forest? In *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, pages 198–214, 2022.

[69] Geoffrey Ramseyer and Ashish Goel. Fair ordering via streaming social choice theory. arXiv:2304.02730, 2023.

[70] Mark Allen Satterthwaite. Strategy-proofness and arrow's conditions: Existence and correspondence theorems for voting procedures and social welfare functions. *Journal of Economic Theory*, 10(2):187–217, 1975.

[71] Leonard J. Savage. *The Foundations of Statistics*. Courier Corporation, 1972.

[72] Fred B. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys*, 22(4):299–319, 1990.

[73] Joao Sousa, Alysson Bessani, and Marko Vukolic. A Byzantine fault-tolerant ordering service for the Hyperledger Fabric blockchain platform. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN)*, pages 51–58, 2018.

[74] Florian Suri-Payer, Matthew Burke, Zheng Wang, Yunhao Zhang, Lorenzo Alvisi, and Natacha Crooks. Basil: Breaking up BFT with ACID (transactions). In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, pages 1–17, 2021.

[75] Christof Ferreira Torres, Ramiro Camino, and Radu State. Frontrunner jones and the raiders of the dark forest: An empirical study of frontrunning on the Ethereum blockchain. In *Proceedings of the USENIX Security Symposium*, pages 1343–1359, 2021.

[76] John von Neumann and Oskar Morgenstern. *Theory of Games and Economic Behavior*. Princeton University Press, 1947.

[77] Jian Yin, Jean-Philippe Martin, Arun Venkataramani, Lorenzo Alvisi, and Mike Dahlin. Separating agreement from execution for Byzantine fault tolerant services. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, pages 253–267, 2003.

[78] Maofan Yin, Dahlia Malkhi, Michael K. Reiter, Guy Golan Gueta, and Ittai Abraham. HotStuff: BFT consensus with linearity and responsiveness. In *Proceedings of the ACM Symposium on Principles of Distributed Computing (PODC)*, pages 347–356, 2019.

[79] H. Peyton Young. *Equity: In Theory and Practice*. Princeton University Press, 1995.

[80] Yunhao Zhang, Haobin Ni, Soumya Basu, Shir Cohen, Maofan Yin, Lorenzo Alvisi, Robbert van Renesse, Qi Chen, and Lidong Zhou. Equal opportunity: A correctness condition for ordered consensus. arXiv:2509.09868, 2025.

[81] Yunhao Zhang, Srinath Setty, Qi Chen, Lidong Zhou, and Lorenzo Alvisi. Byzantine ordered consensus without Byzantine oligarchy. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 633–649, 2020.

[82] Liyi Zhou, Kaihua Qin, Christof Ferreira Torres, Duc V. Le, and Arthur Gervais. High-frequency trading on decentralized on-chain exchanges. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, pages 428–445, 2021.

ARTIFACT EVALUATED
USENIX
AVAILABLE

Jetpack: Consensus Made Generally Fast

Ze Tang[1], Zihao Zhang[1,†], Weihai Shen[1], Jicheng Shi[2], Shuai Mu[1]
[1]*Stony Brook University* [2]*DatenLord*

Abstract

Classic consensus protocols such as Raft require 2 round-trip times (RTTs) for a client to commit a command. Recent protocols reduce this latency to 1 RTT through a carefully designed fast path, but these fast paths are tightly coupled to their host protocols and cannot be retrofitted to existing systems. This paper presents Jetpack, a plugin framework that adds a 1-RTT fast path to existing consensus protocols with minimal modifications.

Jetpack issues commands concurrently to both a fast path and the original path, committing when the fast path succeeds while ensuring the original path honors the same decision. The key challenge is maintaining this guarantee across view changes: we identify the *view change hazard*, a subtle correctness issue where promises made during stable operation can become invalid after leader elections. Jetpack captures this in two structural requirements and two design principles that a fast-path layer can apply across consensus environments, conditions that are easy to overlook when a fast-path idea is adapted to a new setting, as we illustrate with prior designs and a deployed system.

We evaluate Jetpack on six consensus systems across 10 AWS datacenters. Results show that Jetpack achieves 1-RTT latency, reducing average commit latency by as much as 60%, while preserving the original path's performance and properties when the fast path is not used.

1 Introduction

State machine replication and consensus protocols are the backbone of highly reliable distributed systems [55, 23, 41, 27, 54, 39, 61, 38, 68, 67, 30]. These protocols maintain consistency and availability despite machine failures, but achieving low latency remains challenging because consensus fundamentally requires replica synchronization. Classic protocols like Viewstamped Replication [54], MultiPaxos [23, 26] and Raft [55] require 2 round-trip times (RTTs) to commit a command—one RTT from client to leader and back, another for the leader to replicate to followers. This latency is particularly costly in geo-replicated deployments.

To address this, recent consensus protocols offer a "fast path" that allows clients to replicate commands directly to all replicas, bypassing leader serialization and reducing latency to 1 RTT for common cases. Fast Paxos [42] established the theoretical foundation: correctness is maintained by increasing the fast-path quorum size from a simple majority to a supermajority. Subsequent systems such as EPaxos [49], Tapir [65], and SwiftPaxos [59] realize this 1-RTT fast path in complete system designs.

However, the fast paths in these systems are tightly coupled to their protocol designs. They are neither intended nor designed to be generalized to other systems. One cannot extract the fast path from these protocols and apply it to existing systems without adopting the entire design. Yet replacing the consensus layer of a mature system is practically impossible. Even new systems tend to choose reliable, widely adopted protocols. For example, most recent consensus systems [38, 61, 68, 7, 2, 12] in industry are Raft-based.

Given this gap between state-of-the-art and practice, this paper asks: can we develop a universal fast-path framework applicable to a broad range of consensus protocols and systems? We answer this question with Jetpack, a plugin approach that layers fast-path optimization on top of existing protocols without requiring extensive protocol redesign or code modification.

Jetpack aims to achieve a transparent fast path, but a transparent fast path alone is not sufficient in practice. Consider a strawman: implement a Fast Paxos-like fast path as a "gateway" that all requests pass through before reaching the original path, or even skip the original path entirely. While this could apply to almost any consensus protocol, it is impractical: in production systems such as databases, such a gateway disables the original system's optimizations and features (e.g., request batching and slow replica tolerance).

We therefore set three goals for Jetpack beyond achieving 1-RTT latency. First, the fast path and original path must coexist: clients can dynamically choose either path, regardless of other clients' choices. Second, Jetpack must not add overhead to the original path—if all clients use the original path, performance should match the unmodified system. Third, Jetpack must not disable features of the original protocol. For example, Copilot [53] tolerates slow leaders. Copilot enhanced by Jetpack retains this property on its original path while also offering a 1-RTT fast path (which is not slow-tolerant).

To meet these goals, Jetpack runs the fast path and the original protocol path in parallel: each client request travels both paths simultaneously. The fast path exploits commutativity to commit in 1 RTT when concurrent commands do not conflict. The original path proceeds independently to handle conflicts and failures, preserving the original protocol's properties. A client commits as soon as either path returns, and the two paths are designed to converge on the same outcome.

The key safety mechanism is a *promise* from the original

[†]Corresponding author.

path: when the fast path commits a command, the original path promises not to commit any conflicting command before eventually committing the same one. Enforcing this promise is straightforward during stable operation: the original path typically has one or a few leaders responsible for proposing commands, so only those leaders need to acknowledge the promise. However, this promise can become invalid after a view change (e.g., leader re-election due to failure). A new leader may not know about the promise, leading to potential safety violations. We identify this problem as the *view change hazard*, and capture it in two structural requirements that a fast-path layer must meet when paired with a consensus protocol. These requirements are easy to overlook when a fast-path idea is adapted from one environment to another. We illustrate this with three examples whose treatment of view changes leaves these requirements incompletely addressed: an appendix-level sketch in CURP [57] that extends its primary-backup fast path to Raft, a deployed system built on that sketch (Xline [11]), and Carousel [64]. In each case the gap is subtle, and Jetpack's two design principles close it.

We implemented Jetpack on six consensus systems: Raft [55], the most widely used protocol; Mencius [48], a round-robin multi-leader protocol; Copilot [53], which provides fail-slow fault tolerance; MongoDB [68], a production database with customized consensus; etcd [2], a widely used Raft-based key-value store; and ZooKeeper [39], a coordination service built on Zab. All integrations are specified in TLA+ and model-checked with no safety violations across view changes [3]. We evaluated these systems on AWS EC2 across 10 datacenters. Results show that Jetpack achieves 1-RTT latency in common workloads, reducing average latency by as much as 60%. The enhanced systems match original performance when fast paths are not used (§6.4), demonstrating that Jetpack achieves its goals.

Limitation. Jetpack essentially trades throughput for latency: additional protocol overhead on the critical path reduces latency. This tradeoff is fundamental to any fast-path design. For Jetpack specifically, overhead scales with the number of leaders in the original protocol; we demonstrate this with multi-leader protocols like Mencius, which incur higher overhead. However, this is a theoretical bound rather than a design flaw. As a mitigation, we show that a simple adaptive strategy helps the system achieve the same maximum throughput as the original when saturated.

2 Background, Related Works, and Goals

Consensus. Consensus is a fundamental problem in distributed systems and the core building block for replicated state machines. A group of replicas, each with its own input value, must agree on the same output value. In state machine replication, this value is typically a command to execute; by running a sequence of consensus instances, the system ensures all replicas execute the same commands in the same order, maintaining identical state. We use "value" when discussing consensus abstractions and "command" when discussing practical state machine replication.

What makes consensus exceptionally challenging is that replicas must reach agreement by exchanging messages in an asynchronous environment where messages can be delayed arbitrarily. Replicas can crash, slow down, or become isolated by network partitions. No perfect failure detector can distinguish crashed replicas from slow or partitioned ones. The system must make progress as long as a majority of replicas remain responsive.

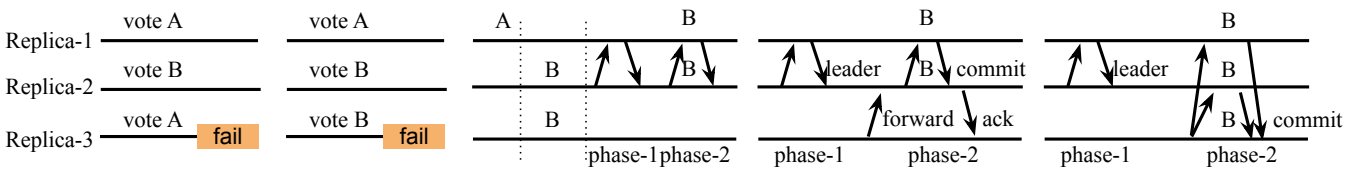
Intuitively, consensus proceeds by replicas voting on values; the value receiving a majority of votes (a quorum) becomes the output. This process is difficult given asynchrony and failures. Figure 1(a) shows a typical scenario. In one case, two replicas vote for A and one for B; A is chosen by majority. In another case, B is chosen. If Replica-3 then fails, the two cases become indistinguishable—one replica voted for A and one for B. Recovery faces a dilemma: either A or B could have been chosen before the failure, and without knowing the failed replica's state, the system gets stuck.

Classic solutions. Three classic solutions are Viewstamped Replication [54], Paxos [40], and Raft [55]. Despite differing in details, they share the same key idea, illustrated in Figure 1(b). Votes are divided into rounds, where only one value can be proposed per round, and all future rounds must adopt any value already chosen. To enforce this, each round is divided into two phases. In the first phase, a replica (e.g., Replica-2) tries to recover any possibly chosen values from past rounds (e.g., B), and makes other replicas promise not to vote for previous rounds anymore. In the second phase, this replica, often serving as a leader, can propose its value, either a previously chosen value or a new one.

In practice, the first phase is usually performed collectively for many consensus instances by the leader, before any commands are ready to propose. When a command arrives, the leader directly enters phase-2 for the appropriate consensus instance. Hence, phase-1 is often called leader election, or a view change. This leader-based structure means only the leader can commit a command in 1 RTT; other replicas require 2 RTTs to commit: 0.5 RTT to forward the command, 1 RTT for the leader to complete phase-2, and 0.5 RTT for the leader to acknowledge the result, as shown in Figure 1(c).

From the client's perspective, latency is higher since clients are not always co-located with replicas [17, 19, 27]. A client needs at least 2 RTTs to commit a request: 1 RTT to send to the leader and receive a reply, in between which the leader takes another RTT to reach consensus.

Reducing latency. The 2-RTT latency is a bottleneck, especially in geo-replicated systems. Some protocols reduce this without modifying the 2-phase structure. Mencius [48] shards consensus instances so each replica leads a subset, achieving amortized 1.5 RTT for co-located clients. However, remote clients still face 2.5 RTT.



(a) A challenging case in consensus

(b) A common structure in classic consensus protocols

(c) 2RTT, with follower forwarding to the leader

(d) Fast path optimization with a super-quorum (>3/4)

Figure 1: Literature review of consensus protocols. From left to right, the figures demonstrate a key challenge in consensus protocols, its common two-phase solution, the 2-RTT latency nature dictated by the solution, and the superquorum-based fast-path optimization. Figure (a) shows two cases illustrating the challenge of split votes in consensus. The two cases are indistinguishable after the failure, despite having achieved a majority vote on different values. Figure (b) shows the common solution in classic consensus protocols, dividing votes into rounds. Rounds are divided by the dotted line in the figure. Figure (c) shows the 2-RTT with follower to leader forwarding. Figure (d) shows the fast path with a superquorum. With 3 replicas, the quorum size is 3.

Further reducing latency is difficult within the 2-phase structure. Doing so requires allowing multiple clients to directly enter phase-2 without phase-1, which returns us to the problem in Figure 1(a): a decided value may be lost after failures. Lamport's Fast Paxos [43] provides the theory to address this problem. The key idea is to enlarge the quorum size to ∼3/4 of replicas for the "fast round," allowing requests to be proposed directly without going through the leader. During recovery, a value that might have been chosen in the fast round dominates by count in the recovery phase of the next round. Figure 1(d) illustrates this: in phase-2, a value is proposed directly to all replicas rather than forwarded to the leader.

Fast Paxos inspired many subsequent works [42, 49, 65, 50, 18, 57, 59, 22, 58, 35, 44, 66] that adopt fast-path optimization to achieve 1 RTT for common workloads. EPaxos [49] and SwiftPaxos [59] use the fast path to reach consensus on command dependencies. CURP [57] uses the fast path to directly commit commands. Tapir [65] and Janus [50] use the fast path for transaction commit.

Question: can we make fast-path optimization universal? Despite these advances, all fast-path protocols are designed from scratch. What if one wants to add fast-path optimization to an existing system? Can we build on top of existing protocols rather than replacing them entirely?

From a research perspective, generalizing fast-path optimization would fill many gaps in the consensus design space. Many protocols have specialized goals: Copilot tolerates slow leaders; Mencius and MultiRaft [38] distribute load evenly among replicas; SAUCR [14] reduces disk write latency by leveraging the time between replica failures. A general fast-path framework could apply to these protocols directly, without redesign.

From a practical perspective, many systems use customized consensus protocols where drastic changes are not an option. For example, MongoDB uses heavily modified Raft [68] to fit its architecture. Even if adding a fast path were possible, it would require significant effort. A generalized approach would be valuable for such systems.

Our goal is to provide a universal fast-path framework applicable to existing consensus protocols, targeting geo-distributed deployments where cross-datacenter round trips dominate commit latency. The framework should provide:

- **1-RTT fast path:** A 1-RTT fast path for clients to commit requests in common cases.
- **Low overhead for the original path:** The framework must not impair the original protocol. This has two aspects: (i) if no requests use the fast path, performance (throughput and latency) should match the unmodified system; (ii) if the original protocol has special properties, like Copilot's slowdown tolerance, the framework preserves them.

We also aim to minimize integration effort. The framework should not drastically change the original protocol. Some modifications are unavoidable, but they should remain minimal, localized, and not require additional correctness proofs. Critically, such changes must never alter the original protocol's outcomes, such as turning a rejected command into an accepted one.

Non-goals. We do not target BFT protocols [33, 15, 63] such as PBFT [25]. We also assume every command is independent, i.e., we do not support interactive transactions, such as Tapir [65]. These are promising directions for future work.

3 Main Idea and Normal Operation

Although our goals may seem ambitious, the underlying rationale is straightforward. This section presents the core idea in the simplified setting of single-decree consensus (§3.1), extends it to multi-decree consensus (§3.2), and concludes with Jetpack's normal-case workflow (§3.3).

3.1 Main idea with single-decree consensus

In single-decree consensus, the system reaches agreement on a single value. This abstraction has been widely used to illustrate the core principles of consensus, most notably in Paxos. Multi-decree protocols used in practice, such as Multi-Paxos, can be viewed as repeated instances of single-decree consensus, each corresponding to a particular log index.

Specializing the framework goals from §2 (a 1-RTT fast path with no impact on the original path) to single-decree

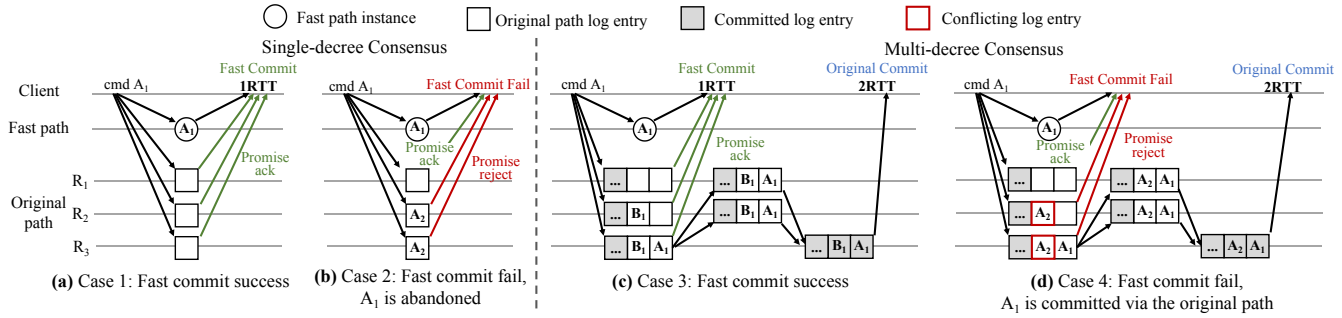


Figure 2: Illustration of the core idea. A_i, B_i denote commands on data A and B. The client sends each command in parallel to the fast path and to three replicas R_1, R_2, R_3 on the original path. The client accepts whichever path commits first (fast: 1 RTT; original: 2 RTT). For simplicity, we show the fast path only by its overall outcome (A_1 is always chosen) and focus the per-replica state on the original path; the overall fast commit succeeds only if the original path also acks the promise. **(a,b)** show the single-decree setting, in which an original replica can ack the promise when it has not proposed or accepted a different command than A_1. In (a), all instances are empty, and fast commit succeeds; in (b), R_2, R_3 hold a different command A_2, so A_1 is abandoned, since A_2 has already been chosen by the original path for the single instance. **(c,d)** show the multi-decree setting, in which an original replica can ack the promise when its log has no uncommitted command conflicting with A_1. In (c), no replica has a conflict, and fast commit succeeds; in (d), R_2, R_3 have a conflicting command A_2, so fast commit fails, R_3 then proposes and commits A_1 via the original path in 2 RTT.

consensus, our goals are twofold: 1) The system can commit the fast-path value in 1 RTT. 2) The original path still operates independently from the fast path.

Consider an arbitrary single-decree protocol serving as the original path, and Fast Paxos as the fast path. The two paths run *in parallel*: a client sends each command to both paths, and each path attempts to commit independently—neither waits for the other. The fast path uses a Fast Paxos-style superquorum to commit in 1 RTT when no conflict arises; the original path runs its own 2 RTT consensus. For correctness, the two paths must agree on the *same* committed value.

Since the fast path finishes earlier (1 RTT vs. 2 RTTs), for the fast-path commit to take effect while keeping the two paths in agreement on the same value, we require the original path to make a *promise* that constrains its future commit value. Only when the original path makes such a promise can the fast-path value be safely considered committed in 1 RTT. Concretely:

Definition 1 (Original Path Commit Promise, Single-decree)**.** *If the fast path commits a value v, the original path guarantees that it will not commit a different value w ($w \neq v$).*

This promise can be realized through a stronger, easier-to-implement variant:

Definition 2 (Original Path Proposal Promise, Single-decree)**.** *If the fast path commits a value v, the original path guarantees that it will not propose a different value w ($w \neq v$).*

As shown in Figure 2(a,b), whenever a value is proposed on the fast path, it is also sent to the original path to request a promise. If the original path has not chosen a different value, it can issue the promise, enabling the fast path to win (Case 1); otherwise, the promise is rejected, and fast commitment fails (Case 2).

This variant is easier to implement because it only requires modifying the proposal logic, which in most protocols is simple logic that sends a value in a message e.g., Accept in Paxos. A brute-force way to implement this promise is to forbid all original-path proposers from proposing any value other than the fast-path value, but this effectively disables the original path, so this does not fit our goals.

A more practical way to implement the promise and fit our goals is to involve only the original path's *initial proposer(s)*: the replicas designated to drive proposals in the system's initial state. In most consensus protocols, these are the replicas that have completed phase-1 of the original protocol before the system begins serving client requests (see §2). Other replicas are not yet proposers; they may only propose after explicitly running phase-1 themselves. This restriction is what makes the promise tractable: only the small set of initial proposers needs to acknowledge it. Most consensus protocols already maintain this distinction, so adopting it adds no new machinery.

With this distinction in place, the protocol works as follows. An initial proposer may propose any value in the original path; if it commits, the value is considered committed for the entire system. This fulfills the goal of the original path still operating and being unaware of the fast path. To integrate with the fast path, we modify the Fast Paxos protocol as follows. When a client attempts to fast-commit a value, it sends the value to both the fast path and the initial proposers (i.e., the Fast Accept quorum includes all initial proposers in the original path). Each initial proposer checks whether it has already proposed any value; if not, it promises not to propose a different value in the future. The client considers the fast-path value committed only if all initial proposers make this promise; otherwise, the fast path fails.

If the system subsequently crashes, a newly elected leader runs a recovery step on the fast path before becoming a proposer. If the fast path has committed a value, the recovery step restores that value and proposes it on the original path, honoring the promise that the original path will match the fast-path value. Otherwise, the recovery step commits a no-op on the fast path, closing it so that no later fast-path commit can occur for this consensus instance, and the system then honors whatever value the original path will commit.

3.2 From single-decree to multi-decree

The single-decree model in §3.1 captures the core insight, but practical systems require multi-decree consensus to build a replicated state machine. In multi-decree consensus, clients continuously send commands to replicas, and replicas agree on which commands to accept and in what order to execute them. A common realization is for replicas to maintain a log of commands, where each log position corresponds to a consensus instance. This is logically equivalent to running many single-decree consensus instances, one per log index. Replicas typically execute commands sequentially in log order to maintain consistent state.

This multi-decree model introduces a key challenge for fast commitment: how do instances in the two paths correspond? In the single-decree setting, the fast path and the original path naturally operate on the same (only) instance, yielding a one-to-one mapping as shown in Figure 2(a). In multi-decree consensus, however, the mapping is no longer naturally one-to-one. The original path can have multiple concurrent consensus instances running, e.g., in Raft, a new leader may propose a new command while the previous command is still pending, and these commands could be modifying the same data item. The system must also allow multiple fast-path instances to run. Therefore, we are looking for a many-to-many mapping between the fast path and the original path.

To simplify the discussion, assume the system runs only one fast-path consensus instance at a time. This reduces the problem to a one-to-many mapping between the fast path and the original path, as illustrated in Figure 2(c,d): a single fast-path command (e.g., A_1) maps to a "window" of candidate instances on the original path (e.g., B_1 and A_1 in (c), A_1 and A_2 in (d)).

To make this one-to-many mapping work, we introduce a refinement:

Leveraging commutativity. In a replicated state machine, correctness (maintaining consistent states among replicas) depends only on the execution order of *conflicting* commands. Non-conflicting commands commute and may execute in any order without affecting the final state. This observation simplifies the problem: we need only ensure that conflicting commands appear in the same order across replicas; non-conflicting commands can be handled independently.

Conflict detection. Jetpack requires the application to provide a predicate $\mathsf{Conflict}(c_1, c_2)$ that returns true when the order of c_1 and c_2 can affect application state or response. For common workloads such as key-value stores, this is straightforward (e.g., two commands conflict if they access the same key and at least one is a write); we discuss application-specific considerations and our key-value implementation in §5.

Refining the original path promise. Following the idea from §3.1, we require the original path to make a promise. But the promise must be refined for the multi-decree setting:

Definition 3 (Original Path Proposal Promise, Multi-decree)**.** *If the fast path commits a command c, the original path guarantees that it will not propose any concurrent conflicting command before c in the log.*

We first observe that most multi-decree consensus systems operate in *views*: stable periods during which the set of replicas authorized to propose is fixed. This stable-view proposer set plays the same role as the initial proposers in single-decree (§3.1): controlling its inputs constrains the command order in the log. From this point on, we use *proposer* to refer to a member of this stable-view proposer set. Concretely, when a client issues a fast-path command, it sends the request to both the fast path *and* the original path (which must include all proposers). Each replica receiving the fast-path request checks for conflicts with all uncommitted commands in its log. If the command is conflict-free, the replica promises not to propose any concurrent conflicting command before this command in the log.

If a superquorum of fast-path replicas acknowledge the command *and* all proposers make the promise, the command can be safely fast-committed. The command will still eventually commit on the original path as well, but the client need not wait for that. For example, in Figure 2(c), command A_1 can obtain the promise, whereas A_1 in (d) cannot because the original path already contains a conflicting command A_2 ahead of it. As a result, A_1 in (c) succeeds in fast commitment, while A_1 in (d) fails and needs to be committed via the original path.

Note that the logic above is discussed in the context of a stable proposer set. View changes are subtle: promises made in the old view must survive into the new one, even though the newly elected proposer was not party to them. We defer this case to §4, where we identify the structural requirements every fast-path layer must meet, derive Jetpack's two design principles, and present the recovery procedure. We first describe how Jetpack processes commands during stable operation.

3.3 Workflow without failures

Jetpack runs on $2f+1$ replicas to tolerate f crash failures. Each replica serves both the original and fast path. A client library routes each command: fast-path commands are broadcast to all replicas while original-path commands are sent only

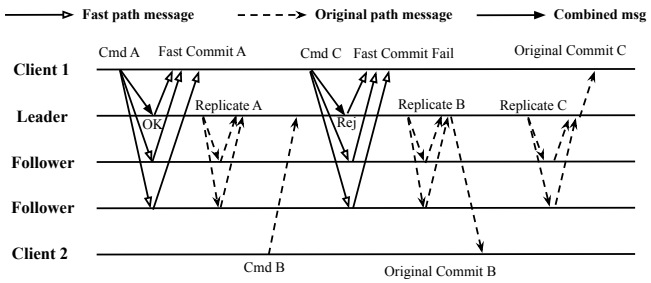


Figure 3: Example workflow of Jetpack integrated with Raft.

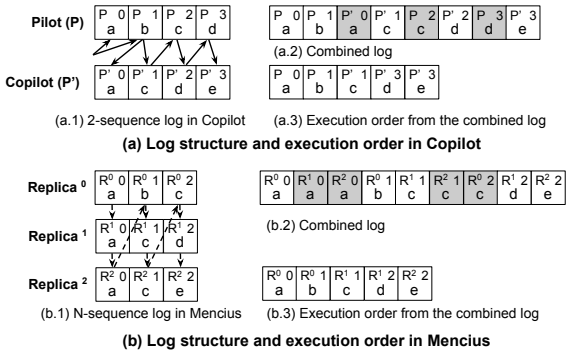


Figure 4: Command execution examples. Each box represents a log entry storing a command, with gray boxes indicating duplicate commands. In (a), the solid arrows show the dependency execution order in Copilot, while in (b), the dashed arrows illustrate the round-robin execution order in Mencius.

to the original path. Clients can select the path explicitly or rely on an adaptive policy (§5). Both clients and replicas track the current original-path view, and every message carries the sender's view so receivers can detect stale operations.

Fast-path processing. Upon receiving a fast-path command, a replica checks for conflicts with uncommitted commands, in both the fast path and the original path. If the command is conflict-free and its view matches the replica's current view, the replica inserts the command into its fast-path log and returns a fast-path acknowledgment. If this replica is a proposer on the original path, it also proposes the command in the original path, regardless of conflicts.

Once the client collects acknowledgments from a superquorum which is at least $(f + \lceil f/2 \rceil + 1)$ replicas and includes all replicas that are original path proposers, the command is fast-committed.

Conflict handling. When conflicts occur, the client may be unable to collect sufficient acknowledgments and promises, causing fast commitment to fail. Jetpack resolves conflicts via the original path, relying on the underlying consensus protocol to order and commit conflicting commands. This design is sound: all commands, fast or not, are eventually handled by the original path. When a fast-path command conflicts with other commands, it can still proceed with the original path as a regular command. From the client's perspective, when fast commitment fails, it waits for the command to commit through the original path.

Figure 3 shows Jetpack integrated with Raft. Commands A, B, and C are conflicting. Command A is fast-committed and also committed on the original path. Command B, intended for the original path only, is committed by the Raft leader without fast-path messages. Fast commitment of command C fails due to conflict with ongoing command B; the leader later commits C via the original path and notifies the client.

Execution correctness. The workflow above places every fast-committed command ahead of any concurrent conflicting commands in the original path log. To ensure the original path produces the same state as the speculative execution on the fast path, execution must respect this log ordering. We require the following prerequisite for the original path:

PR 1. *For two conflicting commands A and B proposed by the same proposer, if A is proposed before B, then A precedes B in the log and is guaranteed to execute before B.*

Although this requirement may appear strict, it is commonly satisfied in practice. The proposer proposes commands by inserting them into its log sequence in an append-only manner, ensuring that any command proposed earlier will always precede those proposed later. Consensus protocols then typically enforce that commit and execution follow the same order as the log. For example, Raft commits and executes log entries strictly in log-index order, meeting this prerequisite. For protocols with multiple log sequences, such as Mencius (Figure 4(b)), a log entry commits only after all earlier entries across all sequences have committed. Copilot, which maintains two log sequences, enforces execution according to dependency order. As shown in Figure 4(a), as long as each proposer inserts command *a* before command *b* in its own sequence, the merged global execution order (Figure 4(a.3)) places *a* before *b*.

To demonstrate generality, we conducted a compatibility study across 16 consensus systems published in OSDI and SOSP since 2000, excluding BFT systems [36, 29, 52, 37] and transactional consensus systems [50, 65, 27]. Figure 5 shows the results. Of these, 6 already have a built-in fast path. Of the remaining 10, 9 satisfy this prerequisite. This indicates that PR 1 captures properties already common in modern consensus protocol design.

Only EPaxos [49] violates PR 1. EPaxos enforces execution according to dependency order, constructed from dependency graphs that may reorder commands through topological sorting. Whether EPaxos can be adapted to work with Jetpack remains open; a promising direction is forcing the topological sort to respect command insertion order.

4 Handling View Changes

The workflow of §3.3 assumed a stable proposer set. We now turn to the harder case: failures and reconfigurations that change the proposer set. We first identify the structural

Protocol(s)	Compatibility
NOPaxos [44], SWARM [51], QuePaxa [62], SKYROS [31], SwiftPaxos [59], LazyLog [46]	already have 1 RTT fast path
Delos [20], Mu [13], Rabia [56] Copilot [53], SAUCR [14], Scatter [34], Mencius [48], Chubby [23], CRANE [28],	compatible with Jetpack (i.e., satisfy PR 1 and PR 2)
EPaxos [49]	violates PR 1

Figure 5: Jetpack compatibility across consensus papers from OSDI/SOSP since 2000.

requirements that any fast-path layer must meet across a view change (§4.1), then derive Jetpack's two design principles from those requirements (§4.2) and describe the recovery procedure (§4.3). Finally, we use prior fast-path designs as evidence that the requirements are real and easy to miss (§4.4).

4.1 The view change hazard

Recall the promise from §3.2: when a fast-path command c is acknowledged, the original path's proposers cannot commit a conflicting command before c. This promise was made by a specific set of proposers in a specific view. The promise itself is *not* durable state in the original protocol—it lives in the fast-path replicas' acknowledgments and in the old proposer's local state. After a view change, none of that is guaranteed to survive: fast-path replicas may have advanced their views, the old proposer may have crashed, and the new proposer was not party to the promise at all.

For the system to remain correct across view changes, the new proposer must do two things on behalf of the old one:

- **(R1) Recover every fast-committed command from the previous view**. A command that was fast-committed before the view change must remain visible—clients have already received acknowledgments based on it.
- **(R2) Place each recovered command before any conflicting uncommitted command in the new view's log.** The fast-path promise was specifically about ordering: the recovered command must execute before any conflicting command the new proposer might propose.

We call the failure of either requirement the *view change hazard*. Both requirements are subtle:

- **R1 is harder than it appears** because the "previous view" is itself ambiguous. Different fast-path replicas may have advanced to different views, and acknowledgments for a single command may straddle multiple views. A naive implementation that simply collects logs from a quorum of survivors can miss commands the client believed were committed.
- **R2 conflicts with standard view-change behavior** in many consensus protocols. Many systems tend to let new proposers carry uncommitted entries forward from prior views (e.g., MongoDB). This is safe in the original protocol, since those entries carry no ordering promise, but it directly violates R2 once a fast path is added: a stale uncommitted entry from an old view can land *before* a recovered fast-committed command.

R1 and R2 are the structural requirements every fast-path layer must meet. We next derive two principles that together guarantee R1 and R2.

4.2 Design principles

As R1 suggests, each time the original path enters a new view, it must recover fast-committed commands and commit them on the original path. For recovery to be tractable, fast-path commands must be cleanly partitioned by view, with each recovery restoring only commands from the previous view. This partitioning requires the fast path to track its own view, separate from the original path's: if a fast-path command's view drifted whenever its co-located original-path replica advanced, the partition would blur and recovery could not identify which commands belong to which view.

Therefore, we propose Principle 1 to address R1:

Principle 1. *The fast path maintains its own view, independent of the original path's view. A command is fast-committed only when both paths are in the same view, and it then belongs to that view in the fast-path log.*

Recoverability requires a fast-committed command to be acknowledged by a superquorum on the fast path, as in most fast-path protocols [42, 49, 65, 50, 57, 59]. But when two paths coexist, this is not sufficient: if acknowledgments come from different views, the recovery quorum for any specific view can be broken. A fast-path acknowledgment issued in an old view cannot contribute to the new view's superquorum, and recovery may miss the fast-committed command. Principle 1 prevents this by requiring same-view acknowledgments, ensuring that any fast-committed value remains recoverable.

With Principle 1 addressing R1, we still need to address R2: every fast-committed command must appear before any conflicting uncommitted command in the new view's log. We achieve this by giving fast-path recovery a higher priority than other commands. We capture this priority as Principle 2:

Principle 2. *On entering a new view v, the new proposers must complete recovery of the previous view's fast-committed commands before committing any other command in v.*

Jetpack achieves this without intrusive changes to the original protocol's recovery procedure: the new proposers commit all recovered fast-committed commands before they propose other commands. This entry serves as the *stability marker* for v: a log position that all future proposers must include and that future view changes use as the recovery boundary. The marker prevents stale uncommitted commands from older views from being committed in v before the recovered fast-path commands.

This priority is not naturally provided by some consensus protocols. Protocols without a fast path, such as MongoDB's pull-based Raft, may commit uncommitted commands from older views during a view change. This is safe in the original protocol (which makes no ordering promise), but once a fast path is added without the stability marker, a stale uncommitted entry could land ahead of a recovered fast-committed command, violating the principle.

Together, these principles guarantee correctness: Principle 1 satisfies R1, ensuring fast-committed commands remain recoverable, while Principle 2 satisfies R2, ensuring consistent ordering of recovered commands across views. §4.3 turns these principles into a concrete recovery protocol. Readers interested in examples of the consequences of violating these principles may jump to §4.4.

4.3 Recovery procedure

Jetpack leverages the original protocol for failure detection and view changes, adding only the synchronization needed to preserve fast-path promises across views.

Two events trigger a view change: proposer reelection changes the proposer set, and membership reconfiguration changes the replica set. Jetpack handles both uniformly by tapping into an interface the original protocol already exposes: a *Propose*() entry point invoked once per command, and a *view* value carrying the current proposer set, membership, and a monotonic identifier.

The proposer maintains the view in which it last called *Propose*(). On each *Propose*() call, it compares the current *view* against this stored view; a mismatch triggers the Jetpack recovery procedure, which carries any fast-committed commands from the old view into the new one.

The entire recovery procedure has three phases:

Phase 1: Detect view change and pause. When a replica detects the view has changed to v, it probes the original path for the most recent committed recovery set (see Phase 3). That set belongs to v_n, the *last normal view*: all views before v_n have already been recovered, so the system could process commands normally in v_n. The replica then acts as the recovery coordinator for v_n. The coordinator broadcasts BEGINRECOVERY to the replica set of v_n, instructing them to stop processing client requests and enter recovery mode. It retries until a majority of replicas acknowledge.

Phase 2: Determine the recovery set. Jetpack runs a Paxos-like procedure to agree on the *recovery set*—the fast-path commands issued in v_n. The coordinator broadcasts PREPARE($v_n \to v$), requesting stored commands from the replica set of v_n.

Each replica returns its previously accepted recovery set for v_n if it has one, otherwise returns its acknowledged commands in v_n (which may be empty). If it already has the consensus result on v_n's recovery set, it can return that instead. By Principle 1, all fast-committed commands in v_n appear on a superquorum and are therefore recoverable.

Once the coordinator collects replies from a majority, it builds the recovery set: if any reply carries an accepted set, it adopts that set; otherwise, it aggregates the returned commands—a command is included if it appears in at least $\lceil f/2 \rceil + 1$ replies, indicating possible fast commitment. Two conflicting commands may both reach $\lceil f/2 \rceil + 1$; in such cases, neither was fast committed, so recovering either is safe.

The coordinator broadcasts ACCEPT($v_n \to v$) containing the recovery set. The set is finalized once at least $f+1$ replicas store and acknowledge it. If recovery fails, any replica that previously accepted a recovery set for a given view includes it in future PREPARE replies; the coordinator simply reuses the attached set. This guarantees all recoveries for the same view recover the same command set. The same mechanism extends to cascading failures across views: by Principle 2, a new view cannot become normal until recovery of the last normal view completes, so all coordinators in successive failed views target the same last normal view and converge to the same recovery set via the accepted-set carry-forward.

Phase 3: Resubmit and resume. The coordinator resubmits the recovery set to the original protocol by sending the set to the new proposer and waiting for it to commit. By committing the recovery set in the new view, the proposer enforces Principle 2. If the recovery set is empty, the coordinator resubmits a single no-op. This commit of the recovery set (or the no-op) serves as the marker that future recoveries use to identify the last normal view in Phase 1.

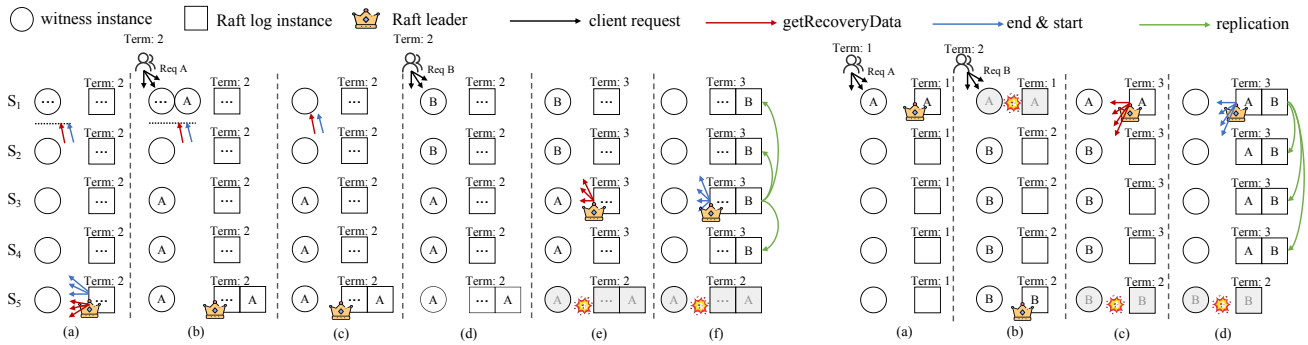
Once these entries commit, the coordinator broadcasts FINISHRECOVERY with the new view v to the replica set of v. Each replica updates to the new view, switches back to normal mode, and begins accepting client commands.

Appendix B provides the full recovery algorithm and optimizations to reduce latency and communication cost.

4.4 The hazard in prior fast-path designs

R1 and R2 are easy to overlook when adapting a fast-path idea to a new consensus environment. To see where the hazard tends to appear in practice, we examine three fast-path designs whose treatment of view changes leaves R1 or R2 incompletely addressed: CURP's sketched Raft extension, a production system that builds on this sketch, and Carousel.

CURP is the protocol most closely related to Jetpack. Both share the high-level idea of running a fast path alongside the original path, but they sit at different abstraction levels. CURP is principally a primary-backup fast-path design; its extension to Raft appears as a brief sketch in the paper's appendix and is not the focus of CURP's main contribution. Jetpack works at a different level: it tries to articulate the structural conditions a fast-path layer must satisfy when paired with a consensus protocol, so they can be reused across consensus environments rather than re-derived for each combination. The two cases below illustrate gaps in the appendix-level Raft sketch, not



Case 1: In (a), S_5 becomes the new leader and all replicas advance to term 2. S_5 sends getRecoveryData to freeze all witnesses, and then sends end & start messages to reset all witnesses after recovery, but S_1 doesn't receive them due to network delay. In (b), the client issues command A at term 2. S_5 and four witnesses, including S_1's stale witness, ack the command. In (c), S_1 later receives the delayed end & start messages and clears its witness state. In (d), the client issues command B at term 2, only two witnesses acknowledge it, so fast commitment fails. In (e), S_5 crashes and S_3 is elected as the leader. S_3 collects commands from witnesses. In (f), S_3 collects commands from a quorum of S_1, S_2, and S_3, and observes that B appears in a majority of 3 servers. S_3 therefore recovers command B and commits it through Raft replication, causing the true fast-committed command A to be lost permanently.

Case 2: In (a), S_1 is the leader when Client issues command A. Command A only gets ack from S_1, so fast commitment fails. In (b), S_1 crashes and S_5 becomes the leader. Command B receives acks from S_5 and four witnesses, to be fast committed. In (c), S5 crashes, S1 recovers and is re-elected. It retains uncommitted command A in its local log and collects fast-committed commands. In (d), S_1 recovers command B and appends it after A in its log, then commits both in that order. This violates the ordering guarantee made in previous term, where command B is committed without any conflicting before it.

Figure 6: Time sequence illustrating two view-change scenarios in CURP+Raft with conflicting commands A and B.

flaws in CURP's primary-backup design. They show why such structural conditions are useful: even careful sketches can leave subtle conditions unaddressed when extended to a new environment.

The CURP recovery procedure proceeds through four phases when a view change occurs: ① the consensus protocol elects a new leader; ② the new leader broadcasts `getRecoveryData` to all witnesses (the term CURP uses for fast-path replicas), stopping them and collecting stored commands; ③ it determines the set of fast-committed commands and replicates them; ④ it sends an `end` message to clear witness state, then `start` to resume normal processing.

Figure 6 illustrates two scenarios that this procedure does not fully cover, one for each of R1 and R2.

Case 1: a gap related to R1, where a fast-committed command can be silently lost. In Figure 6 Case 1(b), command *A* is fast-committed after being acknowledged by a superquorum of witnesses, including the witness at S_1, which is still in the old term. Later, when S_1 receives a delayed `end` message, it erases all stored commands, including *A*. As a result, *A* no longer has superquorum durable storage, and a subsequent leader failure can lose it.

The cause is subtle: in the appendix sketch, CURP uses the corresponding replica's term as the witness term, so a witness lagging in term 1 can approve a command in term 2 based on stale view information. When S_1 later clears its state during recovery, the fast-committed command is gone, and the new leader can insert a different command *B* where *A* should appear. This corresponds to the situation R1 calls out (acknowledgments straddling views), and Principle 1 closes the gap.

Case 2: a gap related to R2, where the promised order can be broken. In Figure 6 Case 2(b), command *B* is fast-committed after S_5 (the old leader) acknowledges that no conflicting command exists. However, when S_1 is elected as the new leader, its log contains an earlier uncommitted command *A*. When the system recovers, command *A* appears *before* the recovered command *B* in the log. This corresponds to the situation R2 calls out (uncommitted entries from prior views landing ahead of recovered fast-committed commands), and Principle 2 (the stability marker) closes the gap.

Production system. To see how the gap manifests in deployed code, we examined a production system that implements CURP-on-Raft, namely Xline [11]. As a production system, Xline is engineered conservatively, with extensive safety checks: it pauses the entire system when a view change occurs and only resumes the fast path after the original path has committed many commands in the new view. This conservatism happens to cover Case 1, but Case 2 still occurs and can lead to data inconsistency. We reported the issue and filed a patch.

Our own implementation with MongoDB tells a similar story: integrating a CURP-style fast path directly into MongoDB ran into a similar gap, because MongoDB's pull-based optimizations are correct under Raft alone but no longer satisfy R2 once a fast path is added (§6.6).

Carousel. A third example appears in Carousel [64], a transactional protocol whose new-leader procedure replicates uncommitted log entries before recovering fast-path transactions; this corresponds to the same R2 situation as Case 2. The full scenario is in Appendix A.

Takeaway. Across these three examples, the same gaps appear because each fast-path integration rediscovers the structural conditions for view changes. Spelling them out once at the framework level allows future fast-path integrations to avoid these gaps by construction.

5 Implementation: Jetpack as a shim layer

Directly extending the original consensus protocol with conflict checking and promise making would require intrusive modifications. To keep changes minimal, we implement Jetpack as a shim layer which we refer to as Jetpack replicas. A Jetpack replica is a process colocated with the originl replica and it sits between clients and the original path. All commands are first sent to Jetpack replicas. Upon receiving a command, each Jetpack replica: (i) forwards it to the paired original replica if this replica is a proposer, and (ii) inserts the command into Jetpack replica's own log. For fast-path commands, a Jetpack replica checks for conflicts and, if conflict-free, issues a promise on behalf of its paired original replica.

A command is fast-committed once the client receives promises from a supermajority of Jetpack replicas, including all those paired with original proposers. If insufficient promises are collected (e.g., due to conflicts), the client waits for the original path to commit the command (§3.3).

Correctness requirement. This shim-layer design preserves correctness by ensuring proposers receive commands in an order consistent with the promise order made by Jetpack. However, this relies on a crucial assumption: the arrival order of commands at each proposer must match their insertion order in the log. Under this condition, a supermajority promise guarantees all proposers place a fast-committed command ahead of any uncommitted conflicting commands, enforcing the promise defined in §3.2. With PR 1 ensuring that the proposed order matches the log order, we also require the following prerequisite to guarantee that commands' arrival order on the proposer matches their proposal order:

PR 2. *For any two commands A and B, if a proposer receives A before B, it will propose A before B.*

Most consensus protocols naturally satisfy this property. However, protocols with additional buffering layers for performance may violate it. For example, Nezha [32] from VLDB employs two intermediate buffers where commands may reside before being appended to the log, breaking equivalence between arrival and log order. As shown in Figure 5, each consensus protocol we surveyed satisfies this prerequisite.

Fast paths beyond PR1 and PR2. When PR2 is violated, arrival order no longer matches proposal order, so the fast-path reply must wait until the command is proposed into the original log. When PR1 is violated (e.g., EPaxos), neither arrival nor proposal order matches execution order; the fast path has no latency advantage and Jetpack cannot meaningfully apply.

Conflict detection. Jetpack replicas detect conflicts based on the key-value interface against in-flight commands stored in command pool, which is a hash map from keys to lists of in-flight commands; the replica reports a conflict on key k if either the new command or any in-flight command on k is a write. A Jetpack replica returns a fast-path acknowledgment only if no conflict is detected; otherwise, the command commits via the original path. All commands, both fast-path and original-path, are inserted into the list, ensuring future fast-path commands detect potential conflicts with all in-flight commands. Other applications can replace this data structure with one matching their detection needs, e.g., interval indexes for ranges or coarser table/namespace-level tracking.

Richer workloads (range queries, SQL predicates, aliased objects, etc.) would need support through the application-provided Conflict interface, e.g., with interval indexes for ranges or coarser scope-based detection (tables, namespaces) for opaque dependencies. Jetpack follows other fast-path protocols in delegating this conflict logic to the application [50, 57, 59, 49, 64].

Garbage collection. To avoid unnecessary promise rejections from stale entries in the command pool, Jetpack replicas must promptly remove completed commands. Once a command is finalized by the original protocol (committed and executed), the result is sent back to the client. Jetpack, as the shim layer, observes these replies and initiates garbage collection: (i) the Jetpack replica that receives the reply removes the command from its own command pool and forwards the result to the client; (ii) it notifies other Jetpack replicas to evict the command from their command pools.

Command execution. After commands commit on the original path, they are executed and applied to the state machine. Since Jetpack proposes fast-path commands to all proposers, multi-sequence protocols like Copilot and Mencius may insert the same command into multiple sequences, resulting in duplicates (e.g., commands a, c, and d in Figure 4).

Copilot naturally tolerates such duplication because a command can be proposed by both pilot and copilot. After merging the two sequences, Copilot executes each command only at its first position (Figure 4(a)). Protocols like Mencius can apply a similar strategy. After ordering log entries round-robin, as shown in Figure 4(b.2), a replica skips duplicate commands (highlighted in gray) and executes only the first occurrence, yielding the final execution order in Figure 4(b.3).

Adaptive path selection. Although the fast path reduces latency, it is not always beneficial. It sends each command to all proposers, increasing CPU load and lowering throughput under saturation. Server locality also matters: when a client is close to the original proposer, replicating to a majority may be faster than obtaining a superquorum for fast commitment.

Jetpack supports adaptive path selection to balance these trade-offs. When CPU load is low, latency dominates, so the fast path is preferred if it is faster; when the system is heavily loaded or the fast path offers no advantage, the client backs off to protect throughput. Each client monitors recent latencies and CPU usage and adjusts its fast-path attempt rate accordingly. Appendix C details the policy, which §6.4 shows preserves original-path performance.

6 Case Studies and Evaluation

We apply Jetpack to six representative systems: Raft, a widely deployed single-leader protocol; Copilot, a dual-leader proto-

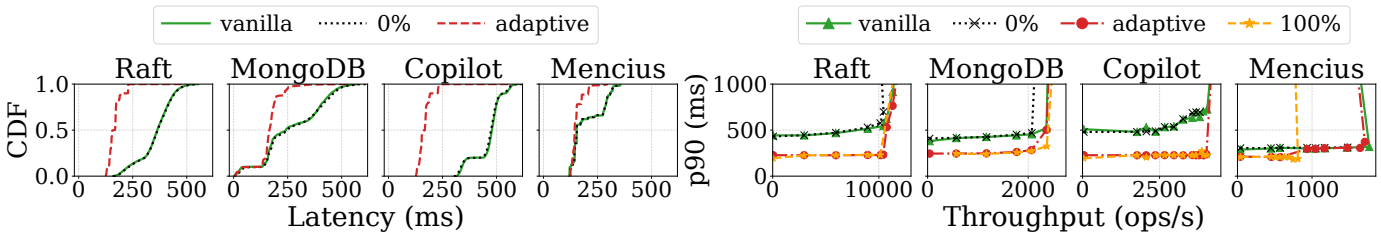


Figure 7: Latency CDF (left) and throughput–latency (right) for Raft, MongoDB, Copilot, and Mencius under a uniform 50/50 read/write workload, comparing the vanilla protocol, Jetpack with fixed attempt rates, and Jetpack in adaptive mode.

col that tolerates fail-slow behavior; Mencius, a classic multi-leader protocol; and MongoDB, a production system with a customized Raft variant. Two additional production case studies on etcd and ZooKeeper appear in Appendix E. In addition to the empirical results below, we model-check each integration in TLA+ for view-change correctness; the specs and artifacts are available at [3].

6.1 Integration

Applying Jetpack requires only minimal modifications to the normal execution path, leaving core protocol logic unchanged. The primary effort lies in handling recovery and view changes through a few lightweight hooks.

On the client side, we assume the original protocol exposes a *Submit*(*cmd*) function. This function is invoked when the client receives either a combined message or an original-path message, and it may also be invoked to resubmit commands during Jetpack recovery.

On the server side, the assumptions and hooks are described in §4.3. The MongoDB integration required 60 lines of code on the original protocol server side; the etcd and ZooKeeper integrations required 68 and 52 lines, respectively.

To prevent Case 2 (§4.4), a new proposer must commit at least one log entry in the new view before acknowledging fast-path commands (§4.2). This requires no modification to the original protocol: Jetpack ensures it by resubmitting recovered commands during recovery. If the recovery set is empty, Jetpack resubmits a no-op and waits for it to commit.

6.2 Experimental setup

Implementation. We implemented Jetpack in C++. For Raft and Copilot, we built on DepFast [47]. For Mencius, we implemented it in the same codebase. For MongoDB, we built on the open-source 8.2.0 release.

Testbed. Experiments ran on AWS EC2 across 10 geo-distributed datacenters (latencies in Appendix D): replicas in datacenters 0–4, clients in all 10. Each 8-vCPU / 16 GB-RAM instance places Raft, etcd, and ZooKeeper's leader in datacenter 0, and Copilot's pilot and copilot in datacenters 0 and 1.

Workload. We use a YCSB-inspired workload similar to CURP: 50% reads and 50% writes over 1M Zipfian-distributed keys by default; we sweep skew 0.5–1.0 and key range 100–1M to vary contention. 60 open-loop clients with multi-threaded request issuance drive load.

6.3 Raft

Integration with Raft follows the standard workflow in §6.1. In Raft, a newly elected leader may contain stale uncommitted entries, potentially triggering Case 2 (Figure 6). With Principle 2 enforced, replicas carrying stale commands are never eligible for election, preventing this case by construction.

Figure 7 shows the latency CDF for vanilla Raft and Jetpack-integrated Raft. With adaptive path selection, Jetpack reduces latency by 53.39% on average, achieving the first design goal (§2). When the fast-path attempt rate is zero—all commands use the original path—the Jetpack curve nearly overlaps with vanilla Raft (Figure 7), confirming no performance degradation when the fast path is unused.

Figure 7 shows behavior under varying load. At low concurrency, adaptive mode aggressively uses the fast path, yielding substantially lower latency. As load grows, the adaptive policy reduces the fast-path attempt rate, causing Jetpack to converge smoothly to vanilla Raft. This demonstrates that Jetpack captures fast-path benefits when available while preserving original performance under saturation. Past the knee, some curves bend back due to queuing at oversaturated replicas—an overload effect rather than a measurement artifact.

Figure 8 shows results under high contention (increased Zipfian skew, smaller key range). The latency benefit diminishes as contention grows due to more conflicts, yet Jetpack consistently outperforms vanilla Raft on both average and p99 latency across all contention levels.

During view changes, Jetpack must recover fast-path commands, introducing additional latency. Figure 9 shows this effect: we trigger a view change by shutting down the leader. After the 10-second timeout, both systems begin recovery. Vanilla Raft completes recovery almost immediately after electing a new leader (orange line). Raft+Jetpack incurs an additional ∼1 second, the time to recover fast-path commands.

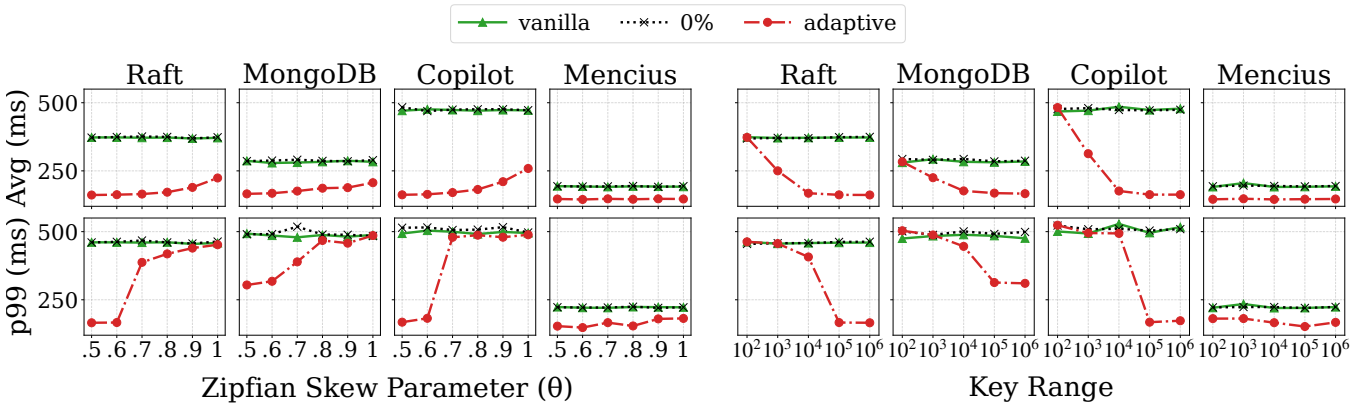


Figure 8: Average and p99 latency for Raft, MongoDB, Copilot, and Mencius under a 50% read / 50% write workload with varied contention.

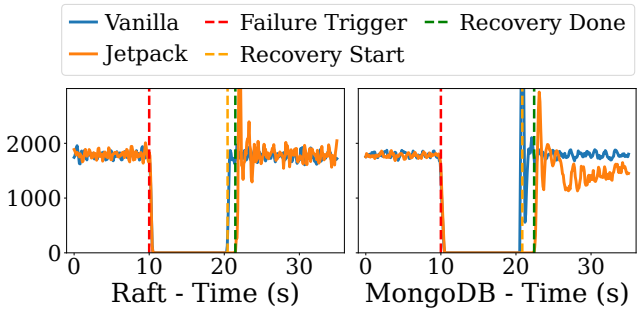


Figure 9: Failure recovery of Raft and MongoDB.

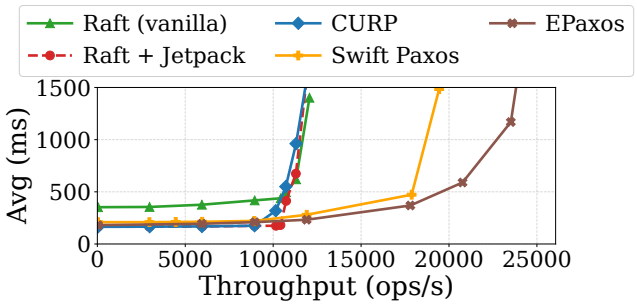


Figure 10: Throughput-latency comparison across Raft, Jetpack, CURP, SwiftPaxos, and EPaxos.

6.3.1 Comparison with native fast-path protocols

We compare Jetpack-Raft against three protocols that achieve a 1-RTT critical path natively: CURP [57] (a witness-based fast path layered on Raft with the view-change hazard fixed), SwiftPaxos [59] (clients broadcast directly to all replicas), and EPaxos [49] (a leaderless protocol where any replica proposes). Vanilla Raft serves as the 2-RTT baseline. All five share a common codebase and run on the testbed of §6.2; for a fair comparison, each replica's consensus thread is pinned to one core and we use a uniform 50/50 read/write workload over 1 M keys to isolate protocol overhead from contention.

Figure 10 shows the throughput–latency curves. At low load, the four 1-RTT protocols (Jetpack-Raft, CURP, SwiftPaxos, EPaxos) commit at 165–210 ms, while vanilla Raft commits at ∼355 ms. Jetpack's plug-in fast path therefore matches the latency of native fast-path designs.

Throughput differentiates the five: Raft, Jetpack-Raft, and CURP saturate at 12 k; SwiftPaxos peaks at ∼19 k; EPaxos at ∼25 k. This gap reflects the tradeoff between generality and deep integration: SwiftPaxos integrates the fast-path broadcast into consensus so the leader skips replicating, and EPaxos makes every replica a command leader to distribute replication across replicas; Jetpack's plug-in design keeps the original path unaware of the fast path and so inherits its throughput envelope, trading throughput for broad applicability.

6.3.2 Composing with production latency mitigations

We further examine whether Jetpack composes with latency mitigations already used in production. Modern geo-distributed deployments often combine two latency mitigations: *leader locality*, which places the leader near the dominant client population, and *lease-based local reads*, which let the leader serve linearizable reads from local state under a quorum-anchored lease. A natural question is whether these techniques subsume Jetpack's fast path, or whether Jetpack remains complementary.

We answer this with a 5-region AWS deployment driven by Facebook's Akkio traces [16]. Akkio migrates each microshard of data toward its dominant access region but cannot perfectly co-locate every client with its shard's leader; the residual steady-state access distribution is roughly 73/8/9/7/3% across the five regions, with 73% of requests accessing the local shard. We mirror this directly: each shard's leader pins to one region, and clients access the five shards in this ratio. Other workload knobs follow Facebook production measurements: 50-byte keys from ZippyDB [24] (a primary Akkio workload), Zipfian skew $\theta = 0.8$ from TAO [21], and a 50/50 read/write ratio from ViewState [45]. We compare five variants: a random-access baseline (Raft on a uniformly random workload); vanilla Raft (Raft on the Akkio distribution, achieving leader locality); Raft+Lease (adds a quorum-

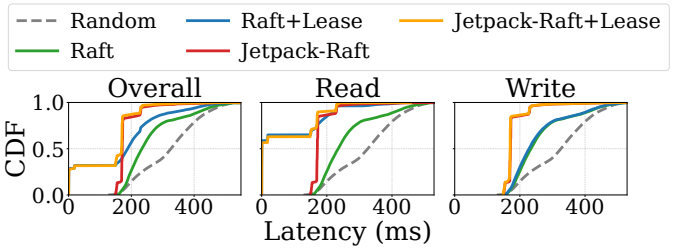


Figure 11: End-to-end latency CDFs (overall, reads, writes) on an Akkio-style simulated workload, comparing random leader placement, vanilla Raft, Raft with read leases, Jetpack-integrated Raft, and Jetpack-integrated Raft with read leases.

anchored read lease on top); and the corresponding Jetpack-Raft and Jetpack-Raft+Lease that integrate Jetpack's 1-RTT fast path.

Figure 11 plots the resulting latency CDFs, with reads and writes shown separately. Leader locality is already effective for the dominant 73% of requests, which reach the local leader within an intra-region hop; the remaining 27%, however, reach a remote shard's leader, paying 2 RTTs. Adding a quorum-anchored read lease lowers latency further, but only on the read side: lease reads complete sub-millisecond when served by the local leader, whereas writes still traverse the entire consensus path. Together, these two limits leave the upper half of Raft+Lease's overall CDF above 300 ms. In contrast, Jetpack's 1-RTT fast path is general, applying to reads and writes uniformly. Its advantage therefore emerges across the entire upper half of Raft+Lease's overall CDF, dominated by writes that the read lease cannot accelerate.

Jetpack is also orthogonal to leader locality and the read lease, composing naturally with both. With locality alone, the 27% remote-shard requests pay 2 RTTs under vanilla Raft. Jetpack's fast path absorbs this overhead, so Jetpack-Raft outperforms vanilla Raft. Adding the read lease on top yields the best regime in Figure 11: local reads complete sub-millisecond through the lease, every write commits in 1 RTT through Jetpack's fast path, and remote-shard reads complete in 1 RTT through either Jetpack's fast path or the read lease. No single mechanism attains this regime in isolation.

6.4 Copilot

Copilot is a dual-leader protocol: a pilot and copilot commit commands independently in separate logs. This provides 1-slowdown tolerance—if one leader becomes slow, the other continues committing. After commitment, the two leaders reconcile logs by establishing cross-log dependencies; this dependency-based ordering preserves per-log order, so Copilot satisfies PR 1.

Despite having its own fast path, Copilot still takes 2 RTTs to commit. Applying Jetpack remains beneficial: Figure 7 shows Jetpack reduces Copilot's latency by 63.90% on average because Copilot frequently fails its `FastAccept` phase

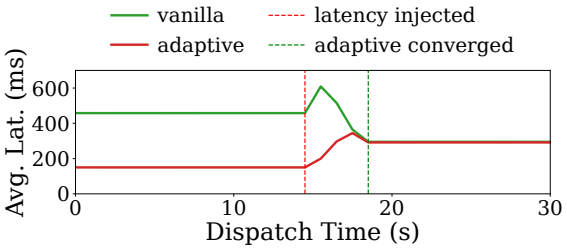


Figure 12: Copilot slowdown tolerance preservation

in geo-distributed deployments and falls back to its 3-RTT slow path. Jetpack's 1-RTT fast path yields large gains even under high contention (Figure 8).

The integration preserves Copilot's slowdown tolerance (Figure 12). We compare vanilla Copilot against Jetpack-integrated Copilot under a simulated pilot slowdown, injecting 300 ms one-way latency using `tc`. The results unfold in three phases: (i) Baseline: Jetpack-integrated Copilot achieves lower latency via its fast path. (ii) Slowdown: when the pilot slows, latency rises for both systems. (iii) Adaptation: vanilla Copilot's copilot performs a takeover; Jetpack's adaptive mechanism detects the slower fast path and reduces its attempt rate, falling back to vanilla Copilot with nearly identical latency.

6.5 Mencius

Mencius is a multi-leader protocol with an N-sequence log for N replicas; each replica leads one dimension. A client normally takes 2.5 amortized RTTs to commit in Mencius.

Integration follows §6.1. With Jetpack, a client can commit in 1 RTT. However, as Figure 7 shows, the fastest 50% of requests show no improvement—vanilla Mencius and Jetpack nearly overlap. Clients co-located with a proposer (5 of 10) already pay only one cross-datacenter RTT; the intra-datacenter hop is negligible, so the fast path yields little benefit. Remote clients incur two cross-datacenter RTTs and see improvement, visible in the slower half of the distribution.

Because all replicas are leaders, each command must be broadcast to every replica, causing each command to be processed multiple times with substantial CPU overhead. As Figure 7 shows, at 100% fast-path rate, Jetpack-integrated Mencius saturates quickly. Adaptive mode mitigates this by dynamically choosing paths: under saturation, it backs off automatically, matching the original protocol's throughput.

Overall, Jetpack's benefit on Mencius is smaller than that in other case studies; moreover, the fast path becomes unavailable upon any replica failure, since fast commitment requires all leaders.

6.6 MongoDB

MongoDB uses a pull-based Raft variant: followers pull entries from a sync source rather than the leader actively repli-

	Memory		CPU	
	Base (MB)	Extra (MB)	Base (%)	Extra (%)
Raft	618.69	30.16	54.26	9.57
Copilot	630.22	28.45	54.52	5.45
Mencius	560.37	5.61	16.70	36.83
MongoDB	919.43	38.86	73.47	3.30
etcd	727.04	27.12	61.22	4.76
ZooKeeper	774.18	5.81	95.96	3.03

Figure 13: Jetpack overhead

cating. During leader election, the new leader may pull uncommitted entries from other replicas, and such stale entries can be inserted ahead of fast-committed commands. We modified the catch-up procedure to skip those uncommitted entries (safe because they were either never acknowledged to clients or acknowledged under weak consistency without durability promises), preventing the case in §4.4.

With Jetpack, MongoDB achieves 39.06% average latency reduction (Figure 7). Approximately 5% of requests show near-zero reduction because reads from clients co-located with the leader are served locally without replication. Under contention (Figure 8), the benefit decreases due to higher conflict rates, yet Jetpack consistently outperforms vanilla MongoDB.

Figure 9 shows recovery performance: once a new leader is elected, Jetpack incurs an additional 1.5 seconds compared to vanilla MongoDB, the time to recover fast-path commands.

6.7 Jetpack overhead

Jetpack reduces latency at the cost of additional resources. We measure CPU and memory overhead across all systems using a uniform workload with 50% reads and 50% writes, with adaptive mode enabled. Figure 13 summarizes baseline consumption and Jetpack's additional overhead.

Across all systems, Jetpack introduces low CPU overhead and modest command pool memory overhead. Mencius is the exception, incurring substantially higher CPU overhead.

Mencius's high CPU overhead explains its limited improvement (§6.5): the extra load saturates the system early, leaving little headroom for the fast path to translate into client-visible gains. This is structural rather than an implementation artifact: Theorem 4 (Appendix G) shows that non-intrusive integration of a client-side 1-RTT fast path with a multi-leader protocol incurs per-command CPU overhead that grows linearly with the number of leaders.

6.8 Discussion of Jetpack's applicability

The empirical results above motivate where Jetpack fits in practice. Jetpack targets geo-distributed settings, where cross-datacenter round trips dominate commit latency. The benefit is largest in single-leader protocols, as remote clients pay a 2-RTT penalty. Production systems therefore commonly adopt multi-leader designs to spread leaders across regions for client-leader co-location. Jetpack still provides improvement on top of these designs, though the magnitude depends on the specific multi-leader pattern, as we discuss below.

One common multi-leader design is the *sharded multi-leader* pattern: data is sharded, each shard runs an independent consensus group with a single leader, and clients co-locate with the leader of the shard they access. Jetpack fits particularly well here. Production traces show that no sharding strategy is perfect [16, 60]: some clients inevitably access shards whose leaders are remote, reaching up to 53% in Facebook's Akkio trace. Jetpack's 1-RTT fast path delivers substantial latency reduction precisely on these remote-shard accesses with low overhead, as our Akkio results show (§6.3.2).

Another design is the *shared-log multi-leader* design (e.g., Mencius), where multiple leaders propose to a single log. Applying Jetpack here incurs structural per-command CPU overhead linear in the number of leaders. Although clients still see latency benefit, this overhead causes the system to saturate sooner, as our Mencius case study shows (§6.5).

A survey of the top 10 DB-Engines databases [1] confirms this: six support multi-shard deployment with leader-based per-shard replication [5, 10, 8, 6, 9, 4]; none use shared-log multi-leader. Mainstream geo-distributed NewSQL systems show the same pattern [38, 61, 12, 27]. Together, these show that the *sharded multi-leader* pattern is the common practice, where Jetpack naturally applies.

7 Conclusion

This paper presents Jetpack, a general approach for adding a 1-RTT fast path to existing consensus systems. The central technical insight is the *view change hazard*: promises made by the original path during stable operation can become invalid after a leader election, breaking the fast path's safety guarantee. Jetpack captures the hazard in two structural requirements that a fast-path layer must meet when paired with a consensus protocol, and addresses them with two design principles. We integrated Jetpack with six representative consensus systems. Across these integrations, Jetpack reduces average commit latency by up to 60% on the fast path and matches the original system's performance when the fast path is not used. We see Jetpack as a step toward making fast-path optimization a reusable layer across consensus systems.

Acknowledgments

We are grateful to our shepherd, Manos Kapritsos, as well as the anonymous reviewers of OSDI '25 and OSDI '26 for their thorough and insightful feedback. We thank Ye Tian for his work in the early stage of the project, and Chenyu Wang for taking charge of running AWS experiments for the first submission. This project was supported in part by NSF awards CNS-2321725, CNS-2238768, and CNS-2130590.

References

[1] DB-Engines ranking. https://db-engines.com/en/ranking, 2026. Accessed: 10 May 2026.

[2] etcd. https://etcd.io/, 2026. Accessed: 10 May 2026.

[3] Jetpack tla+ specifications and model-checking artifacts. https://github.com/stonysystems/jetpack/tree/jetpack/tla, 2026.

[4] Microsoft SQL Server. https://www.microsoft.com/en-us/sql-server/, 2026. Accessed: 10 May 2026.

[5] MongoDB. https://www.mongodb.com/, 2026. Accessed: 10 May 2026.

[6] MySQL. https://www.mysql.com/, 2026. Accessed: 10 May 2026.

[7] Neo4j. https://neo4j.com/, 2026. Accessed: 10 May 2026.

[8] Oracle Database. https://www.oracle.com/database/, 2026. Accessed: 10 May 2026.

[9] PostgreSQL. https://www.postgresql.org/, 2026. Accessed: 10 May 2026.

[10] Redis. https://redis.io/, 2026. Accessed: 10 May 2026.

[11] Xline: a geo-distributed kv store for metadata management. https://github.com/xline-kv/Xline, 2026. Accessed: 7 May 2026.

[12] Yugabytedb. https://www.yugabyte.com/, 2026. Accessed: 10 May 2026.

[13] Marcos K. Aguilera, Naama Ben-David, Rachid Guerraoui, Virendra J. Marathe, Athanasios Xygkis, and Igor Zablotchi. Microsecond consensus for microsecond applications. In *Proceedings of USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2020.

[14] Ramnatthan Alagappan, Aishwarya Ganesan, Jing Liu, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. Fault-Tolerance, fast and slow: Exploiting failure asynchrony in distributed systems. In *Proceedings of USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2018.

[15] Mohammad Javad Amiri, Chenyuan Wu, Divyakant Agrawal, Amr El Abbadi, Boon Thau Loo, and Mohammad Sadoghi. The bedrock of Byzantine fault tolerance: A unified platform for BFT protocols analysis, implementation, and experimentation. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2024.

[16] Muthukaruppan Annamalai, Kaushik Ravichandran, Harish Srinivas, Igor Zinkovsky, Luning Pan, Tony Savor, David Nagle, and Michael Stumm. Sharding the shards: Managing datastore locality at scale with Akkio. In *Proceedings of USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2018.

[17] Masoud Saeida Ardekani and Douglas B. Terry. A self-configurable geo-replicated cloud storage system. In *Proceedings of USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2014.

[18] Balaji Arun, Sebastiano Peluso, Roberto Palmieri, Giuliano Losa, and Binoy Ravindran. Speeding up consensus by chasing fast decisions. In *Proceedings of IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2017.

[19] Jason Baker, Chris Bond, James Corbett, JJ Furman, Andrey Khorlin, James Larson, Jean-Michel Léon, Yawei Li, Alexander Lloyd, and Vadim Yushprakh. Megastore: providing scalable, highly available storage for interactive services. In *Proceedings of Biennial Conference on Innovative Data Systems Research (CIDR)*, 2011.

[20] Mahesh Balakrishnan, Jason Flinn, Chen Shen, Mihir Dharamshi, Ahmed Jafri, Xiao Shi, Santosh Ghosh, Hazem Hassan, Aaryaman Sagar, Rhed Shi, Jingming Liu, Filip Gruszczynski, Xianan Zhang, Huy Hoang, Ahmed Yossef, Francois Richard, and Yee Jiun Song. Virtual consensus in Delos. In *Proceedings of USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2020.

[21] Nathan Bronson, Zachary Amsden, George Cabrera, Prasad Chakka, Peter Dimov, Hui Ding, Jack Ferris, Anthony Giardullo, Sachin Kulkarni, Harry Li, Mark Marchukov, Dmitri Petrov, Lovro Puzar, Yee Jiun Song, and Venkat Venkataramani. TAO: Facebook's distributed data store for the social graph. In *Proceedings of USENIX Annual Technical Conference (ATC)*, 2013.

[22] Matthew Burke, Audrey Cheng, and Wyatt Lloyd. Gryff: Unifying consensus and shared registers. In *Proceedings of USENIX Conference on Networked Systems Design and Implementation (NSDI)*, 2020.

[23] Michael Burrows. The Chubby lock service for loosely-coupled distributed systems. In *Proceedings of USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2006.

[24] Zhichao Cao, Siying Dong, Sagar Vemuri, and David H. C. Du. Characterizing, modeling, and benchmarking RocksDB key-value workloads at Facebook. In *Proceedings of USENIX Conference on File and Storage Technologies (FAST)*, 2020.

[25] Miguel Castro and Barbara Liskov. Practical Byzantine fault tolerance. In *Proceedings of USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 1999.

[26] Tushar Deepak Chandra, Robert Griesemer, and Joshua Redstone. Paxos made live: an engineering perspective. In *Proceedings of ACM Symposium on Principles of Distributed Computing (PODC)*, 2007.

[27] James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, J. J. Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, Wilson Hsieh, Sebastian Kanthak, Eugene Kogan, Hongyi Li, Alexander Lloyd, Sergey Melnik, David Mwaura, David Nagle, Sean Quinlan, Rajesh Rao, Lindsay Rolig, Yasushi Saito, Michal Szymaniak, Christopher Taylor, Ruth Wang, and Dale Woodford. Spanner: Google's globally distributed database. In *Proceedings of USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2012.

[28] Heming Cui, Rui Gu, Cheng Liu, Tianyu Chen, and Junfeng Yang. Paxos made transparent. In *Proceedings of ACM Symposium on Operating Systems Principles (SOSP)*, 2015.

[29] Tobias Distler and Rudiger Kapitza. Increasing performance in Byzantine fault-tolerant systems with on-demand replica consistency. In *Proceedings of ACM European Conference on Computer Systems (EuroSys)*, 2011.

[30] Mostafa Elhemali, Niall Gallagher, Nick Gordon, Joseph Idziorek, Richard Krog, Colin Lazier, Erben Mo, Akhilesh Mritunjai, Somasundaram Perianayagam, Tim Rath, Swami Sivasubramanian, James Christopher Sorenson III, Sroaj Sosothikul, Doug Terry, and Akshat Vig. Amazon DynamoDB: A scalable, predictably performant, and fully managed NoSQL database service. In *Proceedings of USENIX Annual Technical Conference (ATC)*, 2022.

[31] Aishwarya Ganesan, Ramnatthan Alagappan, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Exploiting nil-externality for fast replicated storage. In *Proceedings of ACM Symposium on Operating Systems Principles (SOSP)*, 2021.

[32] Jinkun Geng, Anirudh Sivaraman, Balaji Prabhakar, and Mendel Rosenblum. Nezha: Deployable and high-performance consensus using synchronized clocks. *The Proceedings of the VLDB Endowment (PVLDB)*, 2022.

[33] Neil Giridharan, Florian Suri-Payer, Ittai Abraham, Lorenzo Alvisi, and Natacha Crooks. Autobahn: Seamless high speed BFT. In *Proceedings of ACM Symposium on Operating Systems Principles (SOSP)*, 2024.

[34] Lisa Glendenning, Ivan Beschastnikh, Arvind Krishnamurthy, and Thomas Anderson. Scalable consistency in scatter. In *Proceedings of ACM Symposium on Operating Systems Principles (SOSP)*, 2011.

[35] Rachid Guerraoui, Viktor Kuncak, and Giuliano Losa. Speculative linearizability. In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2012.

[36] Suyash Gupta, Sajjad Rahnama, Shubham Pandey, Natacha Crooks, and Mohammad Sadoghi. Dissecting BFT consensus: In trusted components we trust! In *Proceedings of ACM European Conference on Computer Systems (EuroSys)*, 2023.

[37] Ruomu Hou, Haifeng Yu, and Prateek Saxena. Using throughput-centric Byzantine broadcast to tolerate malicious majority in blockchains. In *IEEE Symposium on Security and Privacy (S&P)*, 2022.

[38] Dongxu Huang, Qi Liu, Qiu Cui, Zhuhe Fang, Xiaoyu Ma, Fei Xu, Li Shen, Liu Tang, Yuxing Zhou, Menglong Huang, Wan Wei, Cong Liu, Jian Zhang, Jianjun Li, Xuelian Wu, Lingyu Song, Ruoxi Sun, Shuaipeng Yu, Lei Zhao, Nicholas Cameron, Liquan Pei, and Xin Tang. TiDB: a Raft-based HTAP database. *The Proceedings of the VLDB Endowment (PVLDB)*, 2020.

[39] Patrick Hunt, Mahadev Konar, Flavio P Junqueira, and Benjamin Reed. ZooKeeper: wait-free coordination for internet-scale systems. In *Proceedings of USENIX Annual Technical Conference (ATC)*, 2010.

[40] Leslie Lamport. The part-time parliament. *ACM Transactions on Computer Systems (TOCS)*, 1998.

[41] Leslie Lamport. Paxos made simple. *ACM SIGACT News*, 2001.

[42] Leslie Lamport. Generalized consensus and Paxos. Technical Report MSR-TR-2005-33, Microsoft Research, 2005.

[43] Leslie Lamport. Fast Paxos. *Distributed Computing (DC)*, 2006.

[44] Jialin Li, Ellis Michael, Naveen Kr. Sharma, Adriana Szekeres, and Dan R. K. Ports. Just say no to Paxos overhead: replacing consensus with network ordering.

In *Proceedings of USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2016.

[45] Haonan Lu, Kaushik Veeraraghavan, Philippe Ajoux, Jim Hunt, Yee Jiun Song, Wendy Tobagus, Sanjeev Kumar, and Wyatt Lloyd. Existential consistency: measuring and understanding consistency at Facebook. In *Proceedings of ACM Symposium on Operating Systems Principles (SOSP)*, 2015.

[46] Xuhao Luo, Shreesha G Bhat, Jiyu Hu, Ramnatthan Alagappan, and Aishwarya Ganesan. LazyLog: A new shared log abstraction for low-latency applications. In *Proceedings of ACM Symposium on Operating Systems Principles (SOSP)*, 2024.

[47] Xuhao Luo, Weihai Shen, Shuai Mu, and Tianyin Xu. Depfast: Orchestrating code of quorum systems. In *Proceedings of USENIX Annual Technical Conference (ATC)*, 2022.

[48] Yanhua Mao, Flavio Paiva Junqueira, and Keith Marzullo. Mencius: building efficient replicated state machines for WANs. In *Proceedings of USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2008.

[49] Iulian Moraru, David G Andersen, and Michael Kaminsky. There is more consensus in egalitarian parliaments. In *Proceedings of ACM Symposium on Operating Systems Principles (SOSP)*, 2013.

[50] Shuai Mu, Lamont Nelson, Wyatt Lloyd, and Jinyang Li. Consolidating concurrency control and consensus for commits under conflicts. In *Proceedings of USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2016.

[51] Antoine Murat, Clément Burgelin, Athanasios Xygkis, Igor Zablotchi, Marcos Kawazoe Aguilera, and Rachid Guerraoui. SWARM: Replicating shared disaggregated-memory data in no time. In *Proceedings of ACM Symposium on Operating Systems Principles (SOSP)*, 2024.

[52] Ray Neiheiser, Miguel Matos, and Luis Rodrigues. Kauri: Scalable BFT consensus with pipelined tree-based dissemination and aggregation. In *Proceedings of ACM Symposium on Operating Systems Principles (SOSP)*, 2021.

[53] Khiem Ngo, Siddhartha Sen, and Wyatt Lloyd. Tolerating slowdowns in replicated state machines using Copilots. In *Proceedings of USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2020.

[54] Brian M Oki and Barbara H Liskov. Viewstamped replication: A new primary copy method to support highly-available distributed systems. In *Proceedings of ACM Symposium on Principles of Distributed Computing (PODC)*, 1988.

[55] Diego Ongaro and John K Ousterhout. In search of an understandable consensus algorithm. In *Proceedings of USENIX Annual Technical Conference (ATC)*, 2014.

[56] Haochen Pan, Jesse Tuglu, Neo Zhou, Tianshu Wang, Yicheng Shen, Xiong Zheng, Joseph Tassarotti, Lewis Tseng, and Roberto Palmieri. Rabia: Simplifying state-machine replication through randomization. In *Proceedings of ACM Symposium on Operating Systems Principles (SOSP)*, 2021.

[57] Seo Jin Park and John Ousterhout. Exploiting commutativity for practical fast replication. In *Proceedings of USENIX Conference on Networked Systems Design and Implementation (NSDI)*, 2019.

[58] Dan RK Ports, Jialin Li, Vincent Liu, Naveen Kr Sharma, and Arvind Krishnamurthy. Designing distributed systems using approximate synchrony in data center networks. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2015.

[59] Fedor Ryabinin, Alexey Gotsman, and Pierre Sutra. SwiftPaxos: Fast geo-replicated state machines. In *Proceedings of USENIX Conference on Networked Systems Design and Implementation (NSDI)*, 2024.

[60] Artyom Sharov, Alexander Shraer, Arif Merchant, and Murray Stokely. Take me to your leader!: online optimization of distributed storage configurations. *The Proceedings of the VLDB Endowment (PVLDB)*, 8(12), 2015.

[61] Rebecca Taft, Irfan Sharif, Andrei Matei, Nathan VanBenschoten, Jordan Lewis, Tobias Grieger, Kai Niemi, Andy Woods, Anne Birzin, Raphael Poss, Paul Bardea, Amruta Ranade, Ben Darnell, Bram Gruneir, Justin Jaffray, Lucy Zhang, and Peter Mattis. CockroachDB: The resilient geo-distributed SQL database. In *Proceedings of ACM International Conference on Management of Data (SIGMOD)*, 2020.

[62] Pasindu Tennage, Cristina Basescu, Lefteris Kokoris-Kogias, Ewa Syta, Philipp Jovanovic, Vero Estrada-Galinanes, and Bryan Ford. QuePaxa: Escaping the tyranny of timeouts in consensus. In *Proceedings of ACM Symposium on Operating Systems Principles (SOSP)*, 2023.

[63] Chenyuan Wu, Haoyun Qin, Mohammad Javad Amiri, Boon Thau Loo, Dahlia Malkhi, and Ryan Marcus. BFT-Brain: Adaptive BFT consensus with reinforcement

learning. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2025.

[64] Xinan Yan, Linguan Yang, Hongbo Zhang, Xiayue Charles Lin, Bernard Wong, Kenneth Salem, and Tim Brecht. Carousel: low-latency transaction processing for globally-distributed data. In *Proceedings of ACM International Conference on Management of Data (SIGMOD)*, 2018.

[65] Irene Zhang, Naveen Kr. Sharma, Adriana Szekeres, Arvind Krishnamurthy, and Dan R. K. Ports. Building consistent transactions with inconsistent replication. In *Proceedings of ACM Symposium on Operating Systems Principles (SOSP)*, 2015.

[66] Zihao Zhang, Huiqi Hu, Xuan Zhou, and Jiang Wang. Starry: multi-master transaction processing on semi-leader architecture. *The Proceedings of the VLDB Endowment (PVLDB)*, 2022.

[67] Jingyu Zhou, Meng Xu, Alexander Shraer, Bala Namasivayam, Alex Miller, Evan Tschannen, Steve Atherton, Andrew J Beamon, Rusty Sears, John Leach, et al. FoundationDB: A distributed unbundled transactional key value store. In *Proceedings of ACM International Conference on Management of Data (SIGMOD)*, 2021.

[68] Siyuan Zhou and Shuai Mu. Fault-tolerant replication with pull-based consensus in MongoDB. In *Proceedings of USENIX Conference on Networked Systems Design and Implementation (NSDI)*, 2021.

A View change hazard in Carousel

Carousel [64] is a transactional protocol that integrates consensus with distributed transaction processing. Within each data shard, Carousel relies on Raft to replicate transaction results among replicas. To reduce replication latency, Carousel adds a fast-path optimization that exhibits the Case 2 hazard pattern (§4.4).

For clarity, we peel the cross-shard distributed transaction coordination from the complete Carousel protocol and focus on the replication within a single shard. To enable fast replication, each transaction is broadcast to all replicas. Upon receiving a transaction, a replica places it into a local *pending list* and checks whether it conflicts with any earlier transaction. If no conflict is found, the replica responds with PREPARED. Once a coordinator collects PREPARED responses from a superquorum of replicas in the same view, the transaction is considered PREPARED on that shard, and the coordinator replies to the client if all shards are PREPARED. The transaction still goes through the normal Raft replication to complete, but the coordinator does not need to wait for it.

When the leader fails, Carousel runs the standard Raft leader election procedure to elect a new leader. During the election, each replica includes its pending list in its vote, allowing the new leader to identify transactions that have succeeded on the fast path.

After a new leader is elected, however, the recovery procedure creates a subtle ordering issue. As specified in §4.3.3 of the Carousel paper [64], the *new leader first completes replicating any uncommitted log entries in its log to followers*, before recovering the fast-path transactions stored in pending lists. If the new leader's log contains uncommitted entries from an earlier term, those entries will be replicated and committed *before* the recovered fast-path transactions.

This ordering does not preserve the fast-path promise from the previous term: once a transaction is fast-PREPARED as non-conflicting, no conflicting transaction should appear ahead of it in the committed order. Committing earlier-term entries ahead of recovered fast-path transactions can invalidate a previously fast-prepared transaction.

An example. Consider two transactions, tx_1 and tx_2, issued by clients C_1 and C_2, respectively. Initially, both data items a and b are 0. Transaction tx_1 writes $a \rightarrow 5$. Transaction tx_2 reads a (obtaining v_a) and then writes $b \rightarrow v_a + 5$. Assume a and b reside on the same shard with five replicas R_1–R_5, and R_1 is the leader in term 1. Carousel processes two transactions in the following order:

1. C_1 starts tx_1.
2. C_1 sends tx_1's Prepare message to all replicas. Due to a network partition, only R_1 receives it.
3. R_1 validates tx_1 (conflict-free), marks it PREPARED, and appends tx_1 to its log.
4. Before replicating tx_1, R_1 crashes.
5. The remaining replicas elect R_3 as the leader for term 2.
6. C_2 starts tx_2, reads a on R_3 obtaining $v_a = 0$, and writes $b = 5$.
7. C_2 sends tx_2's Prepare message to all replicas.
8. R_2, R_3, R_4, and R_5 validate tx_2 (conflict-free), insert it into their pending lists, and reply PREPARED to the coordinator.
9. As the leader, R_3 adds tx_2 into its log.
10. The coordinator collects PREPARED from four replicas, which form a superquorum, and therefore considers tx_2 PREPARED and committed (since tx_2 involves only one shard).
11. R_1 recovers; then, before tx_2 is replicated, R_3 and R_2 crash, leaving R_1, R_4, and R_5 alive.
12. R_1 is elected leader in term 3 (since R_3 crashed before replicating tx_2, the logs of R_4 and R_5 are empty, while R_1's log still carries tx_1 from term 1, so R_1's log is more up-to-date). During the election, replicas R_4 and R_5 send their pending lists, both containing tx_2.
13. R_1 has an uncommitted tx_1 in its log; it first replicates and commits tx_1.
14. R_1 attempts to recover fast-path transactions from the pending lists and discovers tx_2.
15. R_1 validates tx_2, detects a conflict with tx_1. According to §4.3.3 step 4 of the Carousel paper, it considers that tx_2 cannot succeed on the fast path and discards tx_2.

As a result, tx_2 is dropped during recovery and its fast-path completion guarantee is violated, even though the client has already been notified that tx_2 is committed. This example is an instance of violating R2 (§4.1).

We would like to note that our analysis is based solely on the specification available in the paper. We appreciate the Carousel authors' helpful clarification, through personal communication, that Carousel includes additional mechanisms that may address some of the issues discussed here.

B Recovery and correctness

B.1 Recovery protocol

This appendix provides the complete pseudocode (Figure 14) for the recovery procedure described in §4.3.

The recovery protocol proceeds in three phases.

Phase 1 (fast-path freeze). When a coordinator receives a new view from the original protocol's proposer, it first broadcasts BEGINRECOVERY to all replicas in the old view, and instructs these replicas to freeze the fast path so that no fast path can succeed during recovery.

Phase 2 (recovery-set agreement via Paxos). The coordinator then runs a standard Paxos instance to agree on a *recovery set*. The Paxos value is this recovery set, which is computed from the logs returned by the old set and includes all commands that may have succeeded on the fast path.

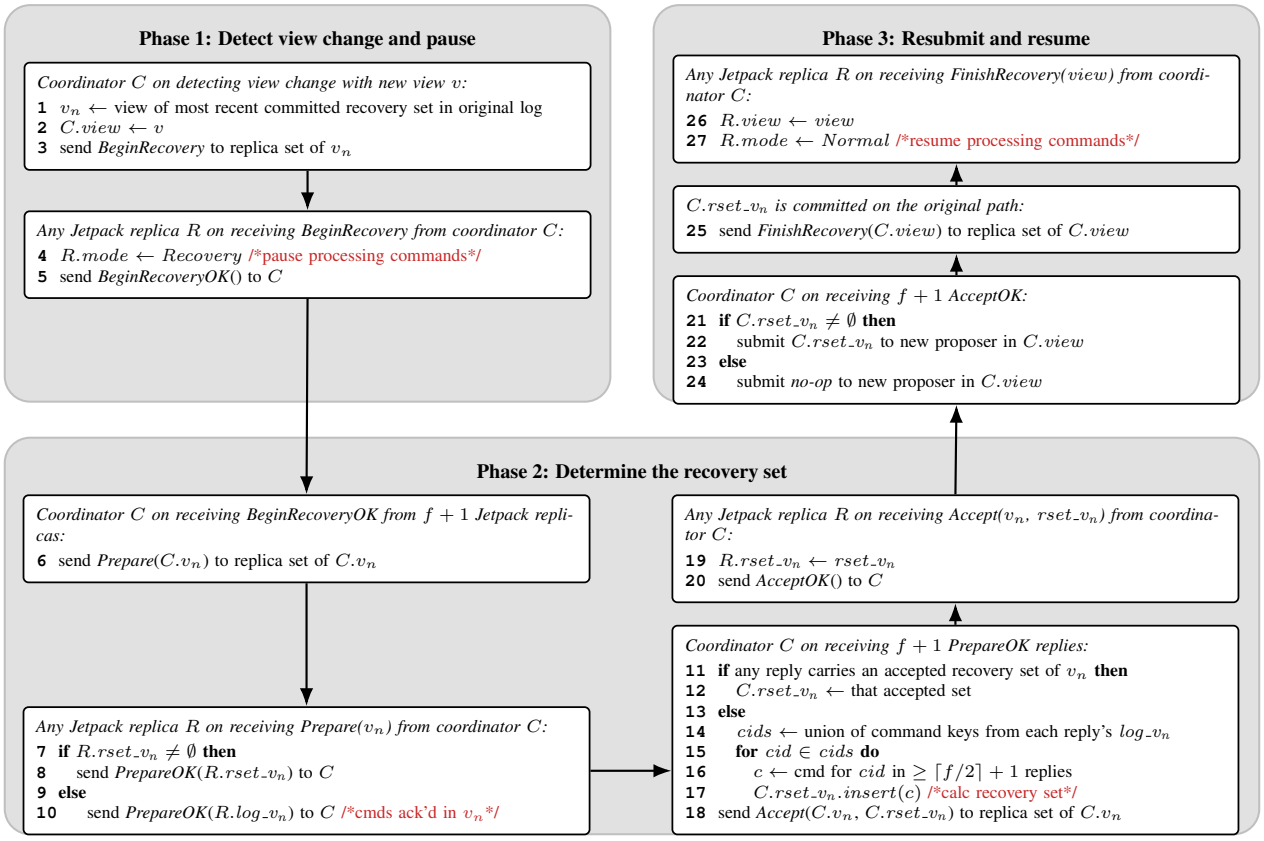


Figure 14: The recovery protocol of Jetpack.

Phase 3 (resubmit and resume the fast path). Finally, the coordinator resubmits the agreed recovery set to the original protocol, or a no-op if the recovery set is empty. The commit of this entry serves as the stability marker that future recoveries use to identify the last normal view in Phase 1. After the entry commits, the coordinator broadcasts a FINISHRECOVERY message to the new replica set, enabling the fast path in the new view.

B.2 Recovery optimizations

Naively, this protocol requires 3 RTT exchanges before any command can be resubmitted: BEGINRECOVERY, PREPARE, and ACCEPT. Moreover, two of these messages carry the entire recovery set or full logs, which can be very large.

We consider two potential optimizations. First, to avoid transmitting the full recovery set or logs, we try to transmit metadata when possible: (1) one RPC to pull log metadata from replicas and compute the recovery set; (2) one RPC to record the metadata and a recovery-set identifier; and (3) one RPC to fetch any command that the current coordinator does not already hold. This optimization replaces large, log-sized messages with smaller metadata messages plus on-demand fetching of missing commands.

However, this refinement increases the total number of RPCs to six. Our second optimization reduces downtime by merging independent steps and thereby reducing the number of round-trips. Concretely, we merge them into two RPCs: one RPC that performs BEGINRECOVERY, pulls log metadata, and executes PREPARE, and a second RPC that records the metadata and recovery-set identifier, pulls missing commands, and executes ACCEPT. The three operations combined in each RPC do not depend on one another, so they can be executed together without changing the protocol's semantics while reducing the number of network round-trips from six to two.

B.3 Correctness proof

We prove that Jetpack preserves correctness when integrated with any original consensus protocol. The proof is structured as follows: we first state our assumptions about the original protocol and define key concepts, then prove three properties—durability, execution consistency, and linearizability.

B.3.1 Assumptions and definitions

We assume the original protocol satisfies the following properties; the *in-order execution* is also enforced in PR 1:

A1. Durability. Once a command is committed on the original path, it will not be lost across view changes.

A2. Consistent ordering. Once a command is committed on the original path at a log position, its position never changes.

A3. In-order execution. The original protocol executes and replies to commands in log order. A command at position j is not executed until all commands at positions $i < j$ are committed.

A4. Linearizability. If command A is committed on the original path before command B is proposed, then B will be executed after A in the log.

In addition, Jetpack enforces the following design principles (from §4.2):

- **Principle 1** (Same-view acknowledgment): A command is fast-committed only when all acknowledgments from both paths are in the same view.
- **Principle 2** (Stability marker): Before activating command processing in a new view v, the new proposers must finish recovering the last normal view v_n and commit at least one entry via the original path in v.

Definition 4 (Fast-path invariant)**.** *If a command A is fast-committed in view v', then during v', no concurrent conflicting command can be committed on the original path at a position before A.*

Justification. This is exactly the fast-path promise made by all original proposers. A fast-committed command A is stored on $f + \lceil f/2 \rceil + 1$ replicas, which includes all proposers. Since every proposer has already received A, any conflicting command B arriving later will be inserted after A in the original log (by PR 2).

We then prove three correctness properties. For each, Cases 1 and 2 (original-path and fast-path-fail commands) follow directly from the original protocol's guarantees. The nontrivial case is always Case 3: a fast-committed command that has not yet been committed on the original path when a view change occurs.

B.3.2 C1: Durability

Any command that has been replied to the client, whether through the fast path or the original path, will not be lost.

Case 1: Original-path command. The command is replied only after it is committed on the original path. Durability follows from **A1.**.

Case 2: Fast-path command, fast commitment fails. The command is replied only after being committed through the original path. Durability follows from **A1.**.

Case 3: Fast-path command, fast commitment succeeds. Once this command is eventually committed on the original path, durability follows from **A1.**. The nontrivial case is when a view change occurs before the command is committed on the original path.

Let v be the new view and v_n be the last normal view. We show that any command fast-committed in v_n is recovered in v.

The command has been stored on $f + \lceil f/2 \rceil + 1$ replicas in v_n's membership (by Principle 1), and by design, these replicas retain fast-path entries until the recovery of v_n completes. Among any $f+1$ replicas of v_n's membership participating in recovery, at least $\lceil f/2 \rceil + 1$ contain the command. Since recovery selects exactly the commands that appear in at least $\lceil f/2 \rceil + 1$ replies (lines 16–17 in Figure 14), the command is included in the recovery set and recommitted on the original path. Once recommitted, durability follows from **A1.**.

Commands from older views. When entering view v_n, the proposers of v_n have already recovered and committed all fast-committed commands from the previous normal view v'' (the normal view immediately before v_n). By induction, every fast-committed command from any earlier view is already committed on the original path. Durability of these commands follows from **A1.**.

B.3.3 C2: Execution consistency

If a command is executed and replied to the client with result r, then in any subsequent execution (including after view changes), the command produces the same result r.

A command's execution result is determined by its position relative to all conflicting commands in the log. Therefore, it suffices to show that a command's position (relative to conflicting commands) never changes after it is replied.

Case 1: Original-path command. The command is executed only after being committed on the original path. By **A2.**, its log position never changes. Therefore, its execution result is stable.

Case 2: Fast-path command, fast commitment fails. The command is executed only after being committed via the original path. Same as Case 1.

Case 3: Fast-path command, fast commitment succeeds. The command A is speculatively executed under the fast-path invariant (Definition 4): the proposer guarantees that no conflicting command will be committed before A on the original path. Once A is committed on the original path, its position is fixed by **A2.**, and all future executions produce the same result.

The nontrivial case is when a view change occurs before A is committed on the original path. Let v be the new view and v_n be the last normal view. We must show that after recovery, no conflicting command appears before A. We prove this in two steps.

Lemma 1 (Original-path recovery safety)**.** *During original-path recovery in view v, no uncommitted conflicting command is recovered at a position before A.*

The original path finishes its own recovery before the fast-path command recovery starts. By Principle 2, the new proposer in view v recovers only uncommitted commands belonging to v_n and does not introduce uncommitted commands from other views.

All commands in v_n were proposed by the proposers of v_n, and by the fast-path invariant (Definition 4), they either:

1. do not conflict with A (recovery of these commands does not affect A's execution result), or
2. conflict with A and appear *after* A in the original log (i.e., at index $j > i$ where A is at index i).

For case (2), consider a conflicting command B at index $j > i$. By PR 1, A at index i will be replicated before B at index j, and therefore be recovered before B. Even in a protocol that allows out-of-order replication (where B could reach a majority while A does not), the recovery may miss A and leave a log hole at index i. Since B at index $j > i$ has not been executed or replied to the client (by **A3.**, it cannot execute while position i is uncommitted), the new leader can safely discard all entries after a log hole. Therefore, original-path recovery never introduces an uncommitted conflicting command before A's position.

Lemma 2 (Fast-path recovery completeness and safety)**.** *Every fast-committed command in v_n is included in the recovery set, and no command in the recovery set conflicts with it.*

Proof. Completeness. By the same quorum argument as C1 Case 3, a fast-committed command appears in at least $\lceil f/2 \rceil + 1$ of the $f+1$ replies during view change, and is therefore included in the recovery set.

Safety. Any conflicting fast-path command C would have failed fast commitment, because A already occupies the superquorum: C cannot obtain a superquorum of conflict-free acknowledgments. Therefore, C cannot appear in $\lceil f/2 \rceil + 1$ fast-path logs, and is excluded from the recovery set. □

By Lemmas 1 and 2, after recovery in view v, the fast-path invariant still holds: no conflicting command appears before A. Therefore, A's execution result remains consistent with its speculative execution.

B.3.4 C3: Linearizability

For any two conflicting commands A and B, if B is issued after A is replied to the client, then A must always be executed before B.

Case 1: A is an original-path command. A is replied only after it is committed on the original path. Since A is committed before B is proposed, by **A4.**, B will be placed and executed after A.

Case 2: A is a fast-path command, fast commitment fails. A is replied only after being committed through the original path. Same as Case 1.

Case 3: A is a fast-path command, fast commitment succeeds. Once A is committed on the original path, A is committed before B is proposed, and by **A4.**, B will be placed and executed after A.

The nontrivial case is when A is fast-committed but not yet committed on the original path when B is issued.

Case 3a: B is issued in the same view as A. If B is a fast-path command, it will fail fast commitment: since A is stored on a superquorum, B cannot obtain conflict-free acknowledgments from a superquorum. Therefore, regardless of whether B is a fast-path or original-path command, B is handled by the proposers and inserted into the original log after A (by the fast-path invariant, Definition 4).

If no view change occurs, B is committed after A and executed after A. If a view change occurs, by Lemmas 1 and 2, A's position is preserved and no conflicting command appears before A. Therefore, B is still executed after A.

Case 3b: B is issued in a higher view than A. A view change occurs before B is issued. By Principle 2, the recovery of v_n (including A) must complete before the new view accepts client commands. Therefore, B can only be processed after A has been recovered and committed on the original path. At that point, A is committed, and by **A4.**, B will be placed after A and executed after A.

C Adaptive path selection strategy

This appendix presents the detailed strategy used by Jetpack to adjust its fast-path attempt rate. §5 introduced the intuition; here we specify the concrete mechanism.

Each client maintains a circular buffer of the most recent 100 requests, recording for each request: (i) whether the fast path was attempted, (ii) its end-to-end latency, and (iii) CPU utilization at issuance time. From this buffer, the client computes running averages: the fast-path latency $\bar{L}_{\text{fast}}$, the original-path latency $\bar{L}_{\text{orig}}$, the recent attempt rate $\bar{A}$, and the current CPU load.

Using these measurements, the client updates its future fast-path attempt rate according to the following rule:

$$\text{AttemptRate} = \begin{cases} \min(\bar{A} + 20\%,\ 100\%) & \bar{L}_{\text{fast}} < \bar{L}_{\text{orig}} \wedge \text{CPU} \le 80\% \\ \max(\bar{A} - 20\%,\ 5\%) & \bar{L}_{\text{fast}} < \bar{L}_{\text{orig}} \wedge \text{CPU} > 80\% \\ 5\% & \bar{L}_{\text{fast}} \ge \bar{L}_{\text{orig}} \end{cases}$$

This rule captures three behaviors: (1) When the fast path is faster and CPU usage is modest, the attempt rate gradually increases. (2) When CPU load is high, even a faster fast path is throttled to protect throughput. (3) When the fast path is slower, it is effectively disabled except for a minimum 5% probing rate, which allows the system to detect future improvements.

During cold start, the client enables the fast path for the first 5 requests to gather baseline measurements; afterward,

ID	0	1	2	3	4	5	6	7	8	9
0	0.2	18.7	236.2	149.0	171.0	145.3	153.3	167.5	135.0	142.9
1		0.3	224.4	140.8	153.7	127.2	141.8	157.4	116.9	132.5
2			0.3	128.6	161.1	115.3	100.6	63.7	118.7	106.5
3				0.4	30.0	12.9	192.6	159.0	27.6	9.5
4					0.1	38.1	221.7	187.6	39.4	38.5
5						0.2	210.9	173.4	10.6	7.6
6							0.2	36.8	258.0	197.9
7								0.1	182.7	167.8
8									0.1	18.9
9										0.1

Figure 15: Network latency between DCs.

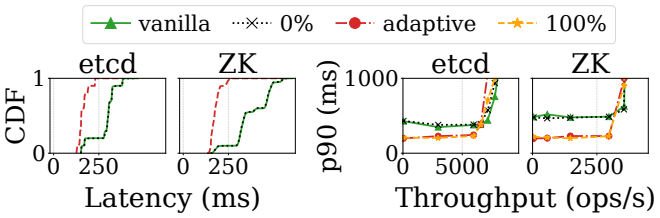


Figure 16: Latency CDF (left) and throughput–latency (right) for etcd and ZooKeeper under a uniform 50% read / 50% write workload, comparing the vanilla protocol, Jetpack with fixed fast-path attempt rates, and Jetpack in adaptive mode.

the adaptive rule takes control. This mechanism enables Jetpack to automatically exploit the fast path when beneficial, while avoiding performance degradation when system load or contention makes the fast path ineffective.

D Network latency between datacenters

We measured round-trip time (ms) between datacenters. These datacenters spanned North America (California, ID 0; Oregon, ID 1), Europe (Frankfurt, ID 3; Stockholm, ID 4; London, ID 5; Ireland, ID 8; Paris, ID 9), and Asia Pacific (Mumbai, ID 2; Hong Kong, ID 6; Singapore, ID 7).

E Additional case studies: etcd and ZooKeeper

In addition to the systems evaluated in §6.6, we also integrated Jetpack with etcd and ZooKeeper. The integration methodology is identical to §6.1, so we report only the performance results here; the latency CDFs and throughput–latency curves appear in Figure 16, and the workload-axis sweeps appear in Figure 17.

etcd. We built on etcd 3.7.0-alpha.0, a widely deployed Raft-based key-value store using a standard Raft implementation. With Jetpack, etcd reduces average commit latency by 42.36% (Figure 16).

ZooKeeper. We built on ZooKeeper 3.9.4, which uses Zab (ZooKeeper Atomic Broadcast), a leader-based atomic broadcast in which a designated leader assigns a total order to write

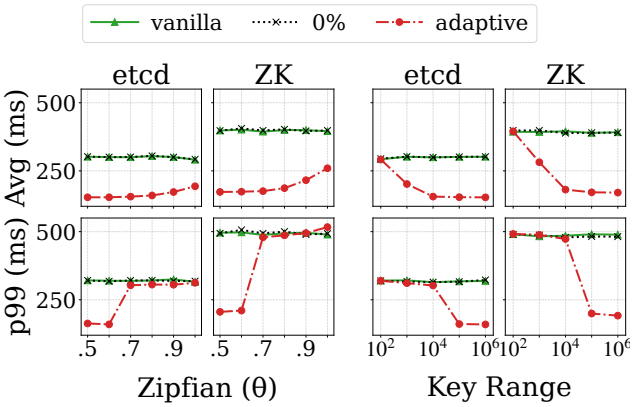


Figure 17: Average and p99 latency for etcd and ZooKeeper under a 50% read / 50% write workload with varied contention (Zipfian skew and key-range size).

requests and replicates them to followers via a propose-ack-commit broadcast. Like Raft, Zab is leader-based and satisfies Jetpack's prerequisites. With Jetpack, ZooKeeper reduces average commit latency by 52.59% (Figure 16).

F Benefit rate of fast path in Jetpack

In the adaptive mode, Jetpack's effectiveness depends on three conditions being met for any given request: (1) the fast path must be attempted, (2) the attempt must succeed, and (3) the resulting latency must be lower than the original path's latency. We define the *fast-path benefit rate* as the percentage of total requests that satisfy all three conditions. This metric quantifies the practical advantage delivered by Jetpack.

Figure 18 shows this benefit rate under two scenarios: a balanced (50% read / 50% write) and a read-intensive (95% read/5% write) workload, with varying Zipfian skewness (θ). In the balanced workload, the benefit rate for all four systems gradually decreases as θ increases. This is because a higher skew leads to greater data contention, which in turn increases the failure rate of fast-path attempts. In the read-intensive workload, Raft and Copilot integration exhibit higher true benefit rates than those in the read-write balanced workload, as read operations do not conflict in Jetpack's fast path, increasing the likelihood of a successful fast path. MongoDB's benefit rate is capped at 5%, since 95% of operations are reads that will complete in 1 RTT without replication.

To further analyze the true benefit rate, we examined a read-write balanced workload (50% read / 50% write) with high contention ($\theta = 1$). For Raft and Copilot, the benefit rates are 65.91% and 64.53%, respectively. In these systems, the fast path is faster than the original path in most datacenters, leading to a high attempt rate ($90-100\%$). This, combined with a success rate of around 60% under high contention, yields the final benefit rate. The rate for Mencius (18.87%)

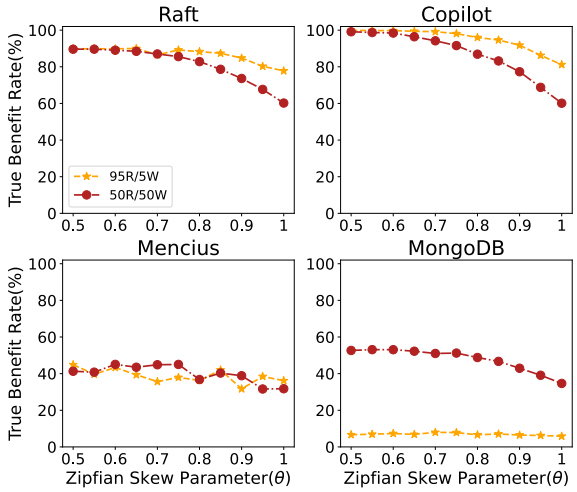


Figure 18: Fast-path benefit rate in a geo-distributed environment

is roughly half of that, as clients co-located with a proposing replica do not gain a latency advantage from the fast path. Similarly, MongoDB's benefit rate (34.66%) is also halved because 50% of its read requests are non-replicated and are already served in a single RTT, receiving no additional benefit from Jetpack.

Even under extreme contention (Zipfian $\theta = 1.0$, 50% read / 50% write), Jetpack benefits between 18.87% and 65.91% of requests, demonstrating substantial fast-path gains across all systems.

G Multi-leader fast-path overhead

This appendix formalizes the argument referenced in the main text: in multi-leader settings with rotating leaders, Jetpack's per-command processing overhead is a structural consequence of its design goals, not an artifact of the particular integration.

G.1 Model and design goals

We consider a *shared-log multi-leader* consensus protocol with $M \geq 2$ leaders $\mathcal{L} = \{l_1, \ldots, l_M\}$, where multiple leaders concurrently propose to a single global log Λ and conflicting commands may be issued to different leaders. Each leader l_i maintains a log sequence Seq_i, and Λ is obtained as

$$\Lambda \;=\; \text{merge}(Seq_1, \ldots, Seq_M)$$

under a fixed deterministic merge rule (e.g., round-robin slot interleaving in Mencius). Replicas execute commands in Λ-order, deduplicating any command that appears in multiple Seq_i (executed only at its first occurrence in Λ). For any command c appearing in Λ, we write p_c for the position of its first occurrence; this is the position that determines execution order.

We adopt the standard asynchronous distributed system model: leader progress depends on incoming commands and message delays, both of which can vary independently. As a consequence, no leader can be perpetually ahead of every other leader; over time, any leader can be transiently the slowest-progressing one.

Lemma 3 (Existence of a min-position leader)**.** *At any time T, the next available slots of the M leaders correspond to M distinct positions in Λ. Hence some leader has a strictly minimum next-slot position in Λ among all leaders.*

Proof. Each leader owns a disjoint subset of Λ's slot positions, so the next-slot Λ-positions of distinct leaders are distinct. The minimum of M distinct values exists. □

Jetpack's three design goals are restated in this model as follows.

- **G1 (Client-side 1-RTT fast path).** For every fast-path command c, the client receives the commit acknowledgment within one round-trip of the request.
- **G2 (Order consistency).** For any fast-path-committed command c and any command d that conflicts with c and has not yet been committed at the time of c's fast-path commit acknowledgment, whenever both c and d appear in Λ they satisfy $p_c < p_d$. That is, the order established by a fast-path commit must be honored by the original protocol's eventual placement in Λ.
- **G3 (Non-intrusive integration).** Jetpack is layered atop the original protocol, whose logic remains unchanged. As a consequence, on any given replica the original protocol cannot observe the fast-path acknowledgment order produced by Jetpack.

G.2 Each command must be proposed by every leader

Theorem 4. *Under **G1**, **G2**, and **G3**, every fast-path-committed command c must appear in every leader's local sequence: $c \in Seq_i$ for all $i \in \{1, \ldots, M\}$.*

Proof sketch. Suppose for contradiction that there exists an execution in which c is fast-path committed but $c \notin Seq_j$ for some leader l_j.

Construction. Let T_0 denote the time the client issues c. By Lemma 3 and the flexibility of the asynchronous model, choose an execution in which l_j is the strictly min-position leader throughout $[T_0, T_d]$, with constant next-slot Λ-position p_j^*, and c has not propagated to Seq_j during this interval.

Deriving the contradiction.

1. By **G1**, the client receives a fast-path commit acknowledgment for c at some $T_c > T_0$.
2. At time $T_d > T_c$, the client issues a command d conflicting with c and routes d to l_j. Since d is issued after c's fast-path commit acknowledgment, the fast-path order is $p_c < p_d$.

3. By **G3**, l_j's decision of which slot in Seq_j to assign to d depends only on l_j's local log state. Since $c \notin Seq_j$ by construction, l_j has no record of c and proposes d at p_j^*.
4. Meanwhile, l_i proposes c on the original path at some time $T_c' \in (T_0, T_d]$. Since l_j is strictly min throughout $[T_0, T_d]$ (Lemma 3 + construction), l_i's next-slot at T_c' exceeds p_j^*, so $p_c > p_j^*$.
5. Combining steps 3 and 4: $p_d = p_j^* < p_c$, contradicting the fast-path order $p_c < p_d$ from step 2 and violating **G2**.

Therefore the supposition fails: $c \in Seq_j$ for every l_j. □

Corollary 5. *Any integration satisfying **G1**, **G2**, and **G3** with a multi-leader protocol incurs per-command processing cost $\Omega(M)$, where M is the number of leaders.*

This corollary directly explains the empirical CPU overhead observed for Mencius+Jetpack in §6.5: with $M = 5$ leaders, each fast-path command is processed five times, matching the observed high CPU increase.

Ambulance: saving BFT through racing

Neil Giridharan* Shubham Mishra* Lorenzo Alvisi† Natacha Crooks*
Benjamin Marsh‡ Hein Meling[1] Kartik Nayak[2] Grzegorz Prusak‡
*UC Berkeley †Cornell University ‡Sei Labs [1]University of Stavanger [2]Duke University

Abstract

Today's practical Byzantine Fault Tolerant (BFT) state machine replication deployments are vulnerable to slowdowns [70, 71]. The main culprit is timeouts. Aggressive timeouts spuriously trigger expensive leader changes, while conservative timeouts leave the system idle and let slowdowns severely inflate latency. Two main alternatives exist: hedging [37, 80], which improves recovery from slow leaders but still incurs a time-based hedging delay, and cooperative asynchronous protocols [19, 66], which recover quickly from slowdowns but suffer from high common-case latency and low throughput. This paper presents Ambulance: a BFT state machine replication protocol that sidesteps this trade-off through *protocol-rigged races*, where replicas, rather than race against the clock, race against each other by executing protocol steps. This enables Ambulance to achieve high throughput and low latency comparable to state-of-the-art timeout-based BFT, while matching the robustness of cooperative approaches.

1 Introduction

This paper presents Ambulance, a Byzantine Fault Tolerant (BFT) state machine replication (SMR) protocol that achieves low latency, high throughput, and robustness to slowdowns (including failures and network blips).

Distributed trust systems allow multiple mutually distrustful parties to secure data even when a subset of them are compromised or misbehave. These systems have been used for secure key recovery [35], code transparency [4], securing land records [57], and stablecoins [8]. Signal [35], for instance, distributes its key recovery logic across three clouds for better security. At the core of these systems lies a BFT SMR protocol, which implements the abstraction of a totally ordered log. This log can then be materialized into application state that is consistent across all parties, even when a subset crashes or misbehaves. These protocols must (*i*) sustain high throughput, (*ii*) ensure low latency, and (*iii*) remain robust to actual failures and network events.

Much of the focus on robustness has been on tolerating discrete events: a replica failure [34] or network events in which messages are dropped [45]. Much less attention has been given to slowdowns [71]. Slowdowns are events such as network misconfigurations, partial hardware failures, etc. [23, 53, 59] that cause a replica to respond more slowly than normal. While slowdowns of course encapsulate machine crashes or network failures (a crashed node is infinitely slow), they also include events that cause latency spikes over longer time periods, without necessarily triggering timeouts.

These events are of major concern to real distributed system deployments. Sei [65], a blockchain company running the Autobahn [45] protocol, as well as Etcd and Neo4j (both Raft-based production datastores) report slowdowns of over seven seconds as a result of data synchronization issues, I/O contention, and garbage collection [7, 65, 69].

Although there exist crash-fault-tolerant protocols robust to slowdowns [70], most existing state-of-the-art BFT protocols are vulnerable to this issue. Even systems like Autobahn [45], which explicitly target recovering from network events, are vulnerable. Because they rely on a leader to drive protocol execution, tail latency can spike when that leader is slow. Techniques such as monitoring leader performance [34] can detect some slowdowns, but are not comprehensive and offer only partial protection [70].

This paper argues that *timeout-triggered fault-detection is fundamentally the wrong mechanism for detecting slowdowns in an SMR protocol*, for two reasons. First, it is a *destructive* process. If timeouts are configured too aggressively, the system might unfairly accuse a leader of being down, unnecessarily triggering an expensive recovery process to elect a new leader. Moreover, trying to elect a new leader while the old leader is still alive can lead to concurrent proposals from both replicas. These proposals will interfere with each other, precluding either from making progress. Second, timeout-triggered fault detection is a *blocking* process. Setting the timeout too conservatively causes the system to wait idle for long periods of time prior to detecting a failure. This is especially problematic as production deployments today recommend setting timeouts to at least ten times the network round-trip time [9]. Timeouts are thus almost always configured to be on the order of seconds for geo-distributed deployments, with some production deployment choosing upwards of 30 seconds [13, 27]! As a result, slowdowns often remain undetected, while truly failed leaders cause the system to stall for seconds [45].

To gracefully handle slowdowns, we need a new fault-detection mechanism that is instead *guaranteed to make forward progress*. It should 1) be *cooperative* rather than *destructive*: it should not destructively interfere with ongoing proposal invocations from existing leaders; and 2) be *productive*: it should contribute useful work towards committing operations. Together, these properties minimize the system's recovery time when there is a slow or failed leader.

Hedging [80] recognizes the issues of timeout-based approaches and offers a promising avenue for better handling of slowdown events. It replaces the traditional competitive leader-based proposal process with a novel cooperative multi-

proposer approach. Each replica proposes in a staggered fashion, according to a known delay schedule. Concurrent proposals contribute to faster termination when there are slowdowns or faults. Consequently, hedging delays can be set much smaller than timeouts. Hedging, unfortunately, still relies on time-based delays. If the first proposer (leader) is slow, then the next replica must stall for the full hedging delay before it can propose. As such, hedging remains a blocking process that is not productive: stalling does not contribute to the protocol's forward progress.

Fundamentally, timeout-based approaches detect slowdowns by having the leader *race against the clock*. The leader is given a fixed deadline to finish its work. If it fails to finish before that deadline, it loses the race and is deemed faulty or slow. Hedging, on the other hand, replaces this fixed deadline with a *rigged race*. Instead of racing against the clock, the leader races against other replicas that are also proposing. Time is used only to give the leader a head start, ensuring that when the leader is not slow, it comfortably wins the race.

In this work, we argue that *rigged racing* is indeed the correct mechanism for handling slowdowns. But racing against the clock or using time to rig the race 1) causes replicas to wait idle 2) requires precise knowledge of what value to set the clock to. Replicas should instead *race against each other*, using protocol messages to rig the race. We purposely design the leader's proposing protocol to take fewer steps than the other replicas' proposing protocol, so that the leader finishes proposing first in the absence of slowdowns. Unlike time-based rigging, non-leader replicas stay productive as they do work that can be used to commit an operation *if they happen to win the race*.

To instantiate this idea, we propose Ambulance, a new consensus protocol that achieves low latency, high throughput, and better robustness to slowdowns through *protocol rigged races*. In the absence of a slowdown, the leader always wins the race, matching the latency of traditional leader-based BFT. In the presence of slowdowns, the leader will lose the race, but the other replicas will quickly recover using the work they did during the race.

Naturally, realizing this in practice requires addressing two key challenges: (*i*) replicas may disagree on whether the leader won the race, and (*ii*) replicas may commit at different stages, with some committing the leader's value during the race and others committing only after the race's dust settles.

Our results are promising. Ambulance matches the common-case performance of Autobahn [45], a state-of-the-art timeout-based BFT protocol, while outperforming it under slowdowns, achieving 1.6–3.0x lower peak latency for 1–2 second slowdowns and up to 10.8x lower latency for more severe cases. Compared to ParBFT2 [37], a leading hedging-based BFT protocol, Ambulance delivers 1.3x higher throughput and 1.9x lower common-case latency, along with 1.7–3.1x lower peak latency during slowdowns.

This paper makes three core contributions:

- We identify the two necessary conditions for slowdown-detection mechanisms to be precise and efficient: being cooperative and productive.
- We introduce a new slowdown-detection mechanism, protocol-rigged racing, the first to satisfy both conditions.
- We implement and evaluate Ambulance, a novel BFT protocol that uses protocol-rigged racing to offer high throughput, low latency, and robustness to slowdowns. Ambulance is deployed in production at a leading distributed trust company.

2 Background

We first survey the slowdowns observed in real deployments, and then examine the mechanisms used to detect them.

2.1 Slowdowns in practice

In production, distributed systems often experience *slowdowns*: periods when a replica remains correct and available, yet its service rate falls below its usual baseline. Even brief slowdowns can cause end-to-end latency spikes and trigger timeouts.

Slowdowns arise from a mix of hardware, software, and network effects. Fail-slow behavior is prevalent across the hardware stack: disks may suffer throughput degradation from vibrations, memory cards from loose connections, CPUs from power loss. Even NICs can drop packets because of firmware or driver bugs [48, 63]. Software slowdowns arise because of I/O contention, slow data syncs, or garbage collection pauses. I/O contention in etcd, for instance, can induce long `fsync` latencies [7]; since proposals must be persisted to disk, these stalls directly translate into large latency spikes. Data synchronization likewise causes sufficiently long latency spikes in HBase that developers have added a dedicated feature to detect slow syncs [5]. Sui [15], a production blockchain system, similarly observes that slow data synchronization can trigger slowdowns lasting up to 5 seconds [13]. Elasticsearch and Neo4j both report that rapid heap allocation can trigger stop-the-world garbage collection pauses lasting seconds to minutes, inadvertently causing leader election timeouts [11, 69]. Finally, network pathologies, including misconfigurations and flaky links have been shown to cause partial network partitions in several production data stores [22].

2.2 Slowdown detection mechanisms

Slowdowns are managed in a variety of ways today.

Timeouts. The most common way to detect slowdowns is to use timeouts. Timeouts effectively make the leader race against the clock to finish committing its proposal. If the leader loses, the timeout fires, triggering a leader election process to replace it.

Tuning timeouts is notoriously difficult [80]. It requires significant administrative effort, and a poorly chosen value can degrade performance or compromise availability. Aggressive timeouts risk frequent leader elections, in which new leaders

override the progress of old leaders, hurting liveness. Conservative timeouts give leaders plenty of time to win the race, but cause the system to stall in the process. This is especially problematic when the leader fails, as other replicas must wait out the full timeout to detect that failure. Moreover, timeouts are typically set statically, which makes them adapt poorly to changing conditions, such as network latency fluctuations or replicas that become gradually slower over time. Taken together, these limitations make clear that timeouts are an overly coarse heuristic for detecting slowdowns.

Production systems recommend setting timeouts to be at least ten times the network RTT, though in practice they are even more conservative. Default timeout values include 2 seconds for Sei [65], 5 s for Microsoft CCF [10], 7 s for Neo4j [69], and 30 s for Diem [13, 79].

Cooperative Proposing. Timeout-based protocols make progress only if the leader "beats the clock". When it cannot, the system stalls until the replicas elect a new leader. Effectively, these protocols are built around the assumption that, in the typical case, the leader usually wins this race [20, 34].

Asynchronous consensus protocols [19, 60, 66, 73] instead start from a different premise: *all* leaders may consistently lose the race against the clock. This can be a result of poorly configured timeouts or because the network is controlled by an adversary that selectively delays the leader's messages. Thus, to avoid blocking on a single leader, asynchronous protocols let all replicas propose cooperatively. During slowdowns, this approach achieves low latency and remains robust up to f slow replicas, since $n-f$ correct replicas are guaranteed to eventually respond. However, this approach is more expensive in the common case, since extra coordination is required to handle multiple proposers. Asynchronous protocols thus have significantly higher latency in the absence of slowdowns.

Hedging. The most recent slowdown detection mechanism, hedging [37, 80], takes a middle-ground between the timeout and cooperative proposing approaches. With hedging, all replicas cooperatively propose (like in asynchronous protocols), but they do so in a staggered fashion according to a known delay schedule. This creates a race between the leader (the first proposer) and the remaining replicas to finish proposing. If the leader wins the race, the latency matches that of traditional leader-based BFT. Otherwise, the other proposers cooperatively work together to terminate the protocol. Hedging preserves the efficiency of timeout-based protocols in the absence of slowdowns. Unfortunately, during a slowdown, recovery is unnecessarily delayed until after the leader's head start. Furthermore, extra coordination is needed to reconcile the leader-based and cooperative mechanisms. During slowdowns, ParBFT2 [37], the state-of-the-art hedging BFT protocol, has a consensus latency of 22 message delays on top of the hedging delay, compared to only 10 message delays for SMVBA [49], the state-of-the-art cooperative BFT protocol.

Toward Rigged Racing. In short, current detection mechanisms are inadequate. Timeouts are too slow to react to slowdowns, as the system stalls until the timeout triggers. Cooperative approaches are not practical, as they unnecessarily pay the cost of higher latency always. Hedging, while an improvement, still stalls until the hedging delay is over.

A better slowdown detection mechanism should meet three requirements. First, it must be *cooperative*: concurrent proposals should not destructively interfere with each other. Second, it must be *productive*: the work done during detection should directly accelerate commitment once the slowdown is detected. Third, it must detect real slowdowns quickly without mislabeling a healthy replica as slow.

These requirements rule out timeouts and hedging, since both rely on time-based delays that fundamentally do not contribute to useful protocol progress. A more promising direction instead quantifies slowness by measuring how quickly replicas advance through protocol steps that replicas must anyway execute. Truly slow replicas will consistently lag behind. However, raw relative progress is noisy: routine jitter can briefly make a healthy replica look slow. To satisfy the third requirement, we need a mechanism that separates genuine delay from benign variation. Our solution, *protocol-rigged racing*, meets all three requirements by racing replicas while carefully adding sufficient bias to prevent false accusations.

3 System Model

Ambulance adopts the assumptions of prior BFT protocols. We assume $n=3f+1$ replicas, of which at most f are faulty, as well as a PKI for digital signatures, and a trusted setup for threshold signatures. We consider a replica correct if it adheres to the protocol specification; a replica that deviates is considered faulty (or Byzantine). We make no assumptions about the number of faulty clients. We assume the existence of a strong, yet static adversary, that can corrupt and coordinate all faulty participants' actions but cannot break standard cryptographic primitives. Replicas communicate through reliable, authenticated, point-to-point channels; we write $\langle M \rangle_p$ for messages signed by replica p. We assume all signatures and quorums are validated and omit the check in protocol descriptions. Ambulance operates under the asynchronous network model: it makes no synchrony assumptions for safety or liveness.

4 Ambulance: Overview

Ambulance seeks to achieve two goals: (*i*) match the high performance of timeout-based BFT protocols in the absence of slowdowns, and (*ii*) provide stronger robustness to slowdowns than both hedging-based and timeout-based BFT protocols.

Timeout-based BFT protocols enjoy low latency by relying heavily on a designated leader: they assume the leader will beat the clock most of the time. When that happens, the leader becomes the sole replica authorized to propose new operations, and agreement can be reached quickly, with only three message delays [30]. Unfortunately, these protocols pay

a steep price when the leader loses the race. Hedging-based protocols improve robustness by having all replicas propose concurrently when the leader falls behind, but they still pay both the hedging delay and additional coordination needed to reconcile all proposals. Ambulance overcomes these limitations and achieves both objectives through a new technique called *protocol-rigged racing*.

In Ambulance, a designated leader drives commitment of its proposal in its *proposal lane*. Additionally, each replica has a proposal lane of its own, where it proposes operations and votes in the lanes of others.

Ambulance consists of two components: a race, in which replicas try to beat the leader (but assume that they won't), and a recovery path in which the replicas try to commit their own proposal if they believe that the leader lost. During the race, the leader runs a faster protocol that ensures that it will win if it is not slow. If the leader loses, replicas do not have to turn back and start over. Instead, the steps taken to beat the leader are directly useful towards committing.

Race. Replicas race with the leader by submitting their own proposals. Rather than having replicas perform meaningless busy work during the race, Ambulance's key insight is to have them execute required protocol operations during this time, shortening the amount of work left post-race. *All* consensus protocols fundamentally consist of two phases: a *non-equivocation* phase and a *persistence* phase. Non-equivocation guarantees that malicious replicas cannot cause conflicting operations to commit; persistence ensures committed operations survive. Ambulance uses the non-equivocation phase as the race itself! Both leaders and replicas need it *anyways* to commit.

To ensure a non-slow leader will win the race, the race is rigged in the leader's favor in two ways. First, the leader completes the two-step, quadratic message exchange of PBFT's non-equivocation phase while the other replicas instead perform a slower three-way linear message exchange. Second, the leader does not have to finish first to win; it only has to beat the cutoff. This cutoff is chosen to maximize the leader's chances of winning, while still allowing replicas to recover quickly if the leader is slow. When the cutoff is reached locally, the race outcome is determined. The key challenge is that replicas may disagree on the race outcome. One replica might observe the leader winning, while another sees the leader losing. Despite this divergence, Ambulance ensures agreement across all participants.

Recovery Path. If a replica failed to commit the leader's proposal, it moves to the recovery path. Unfortunately, there is no longer a dedicated leader from which to choose a proposal. Instead, all replicas potentially have formed a distinct non-equivocation certificate for their lane. To make matters worse, the leader lane could have committed with only some replicas knowing. The recovery path addresses each issue in turn and proceeds in three steps. First, a replica chooses a value to recover for its lane. Second, it persists this value. And third, a lane is randomly selected to be committed.

Step 1. Replicas must recover any potential proposal that *could* have committed from the leader. If a leader proposal could have committed, replicas must select that value as input to the recovery path. If no leader proposal was committed, replicas recover their proposal from the race to ensure that the progress made during the race was productive.

Step 2. Replicas then persist the recovered value, so that it is safe to commit it. This step has two parts. The first is an optional *race exclusion phase* that addresses a subtle issue. If a leader non-equivocation certificate forms but the leader does not successfully commit, a malicious replica may successfully obtain both a non-equivocation certificate from the leader and for its own lane. It cannot equivocate on any proposal, but it still can equivocate between the leader's proposal and its own lane proposal. Thankfully, we find that, in most cases, the leader is slow enough that replicas can directly generate a proof that the leader's non-equivocation certificate *cannot exist*. Most of the time, replicas thus safely omit this phase. Only when they cannot generate this proof must they proceed to run this additional exclusion phase to guarantee uniqueness between the leader's non-equivocation certificate and the replica's. The second part is the *persistence phase*, which performs the same role as in traditional BFT. It guarantees that if the lane proposal is committed, then it survives despite failures.

Step 3. The third and final step is that a lane is *randomly* selected for commitment. Note that this election process is run *after* each replica has completed all the steps required to commit a proposal in its lane. This is by design: if the winner were chosen deterministically, a network adversary, could intentionally delay messages necessary for the anointed winner to commit the operation, defeating the point of not having a leader. If the randomly selected lane does not have a proposal ready for commit, the protocol simply retries the recovery path.

Performance Summary Using these techniques, Ambulance achieves a latency of three message delays under normal conditions, matching the performance of latency-optimal PBFT [30]. When the leader experiences a slowdown, the expected latency is comparable with state-of-the-art pessimistic (asynchronous) protocols [19, 49]: 9.5 message delays if the leader fails to propose during the race, and 10.5 message delays otherwise. These results meet our goal of delivering state-of-the-art latency in both normal and slowdown scenarios.

5 Ambulance: The Protocol

We now describe the protocol in more detail. We first focus on how the protocol works for a single slot, before expanding it to support efficient agreement across multiple slots. For ease of exposition, we omit the standard checks performed by all replicas—*i.e.*, whether messages are well-formed and from the correct view, and signatures are valid. Replica state is summarized in Algorithm 1. We defer formal proofs of

correctness to Appendix §B (Theorems 1 and 2).

Execution in Ambulance is organized into a sequence of views. Within a view, replicas try to commit a proposal. The protocol consists of two components: a *race*, in which replicas determine whether the leader is fast enough to commit, and a *recovery path*, in which they try to commit their own proposal if the leader is deemed slow. As previously stated, each replica in Ambulance has its own proposal lane. Replicas drive commitment for their own lane, but additionally vote in other replicas' proposal lanes. Note that the leader runs both a leader lane and a replica lane.

Algorithm 1 Replica state cheat sheet for a given slot s.

```
1: v                          ▷ replica's current view
2: sc-prep                    ▷ SC-PREPARE from the leader or ⟨⊥_prep⟩
3: sportscar-cert             ▷ sports car certificate or ⟨⊥_sc⟩
4: truck-cert                 ▷ truck certificate
5: race-cutoff cert           ▷ race-cutoff certificate from race
```

5.1 The Race

The protocol begins with a race whose purpose is two-fold: (*i*) correctly and quickly identify whether the leader is slow and (*ii*) allow a non-slow leader to commit.

To achieve this, the leader and replicas simultaneously issue proposals in their own lanes and race to form non-equivocation certificates for their lane. Such certificates guarantee that, in any view, only a single value can be proposed in the corresponding lane. We call the leader's non-equivocation certificate a *sports car certificate*, and a replica's non-equivocation certificate a *truck certificate*[1]. Non-equivocation is a necessary step in essentially all BFT protocols, so replicas make progress toward commitment while participating in the race.

We ensure that leaders progressing normally are not accidentally identified as slow by rigging the race in the leader's favor in two ways. First, the leader runs a faster non-equivocation procedure that produces a sports car certificate in only two message delays, whereas each replica lane requires three message delays to produce a truck certificate. This structural asymmetry naturally biases the race toward a non-slow leader. Second, the leader does not need to finish first overall to avoid being detected as slow. It only needs to finish prior to a *cutoff*, which occurs when a quorum of truck certificates has formed. If a sports car certificate forms prior to the cutoff, the leader wins the race and replicas continue toward committing the leader's value. Otherwise, the leader loses the race and replicas proceed to the recovery path.

5.1.1 Leader Lane: Sports Car Certificate

Ambulance structures the leader lane to guarantee that a non-slow leader will beat the cutoff while still making useful progress toward commitment. The leader lane implements the standard non-equivocation step required for commitment with a two-step all-to-all communication pattern, enabling a non-slow leader to produce a *sports car certificate* in only two message delays (illustrated in Figure 1).

1: L → R: Leader L sends SC-PREPARE to all.

The leader broadcasts a SC-PREPARE $:= \langle v, val \rangle_L$ message, where v is the view number and val is the value proposed by L.

2: R → R: Replica R processes SC-PREPARE and sends vote to all replicas.

On receiving a SC-PREPARE message $prep$ from the leader, R checks that $prep$ is valid and stores it in $sc\text{-}prep = prep$. It then broadcasts a SC-PREP-VOTE $:= \langle v, dig = h(prep.val) \rangle_R$ message to all replicas, where h is a cryptographic hash function.

3: R: Replica R assembles a sports car certificate.

R waits for at least $n - f = 2f + 1$ matching SC-PREP-VOTE messages and aggregates them into a *sports car certificate*. It sets *sportscar-cert* $:= (v, dig, \{\text{SC-PREP-VOTE}\})$.

This certificate has two purposes. First, it guarantees non-equivocation—the uniqueness of the leader's proposal within a view—since correct replicas vote only once and any two quorums of size $n - f$ intersect in at least one correct replica. Second, as we are about to see, it serves to determine whether the leader beat the cutoff. R stores this certificate in *sportscar-cert* for potential use during recovery.

5.1.2 Replica Lanes: Truck Certificate

Each replica lane similarly produces a non-equivocation certificate, called a *truck certificate*. Unlike the leader lane, a replica lane uses a three-step linear communication pattern, so truck certificates form more slowly than a sports car certificate by design (illustrated in Figure 2).

We describe the protocol from the perspective of a single proposer replica P, which drives one such lane to completion. In practice, all replicas concurrently act as proposers in their own lanes, and they all contribute to each other's lanes.

1: P → R: Replica P sends TRUCK-PREPARE to all replicas.

A proposer P broadcasts a message TRUCK-PREPARE $:= \langle v, val \rangle_P$, where v is the view number and val is the value proposed by P.

2: R → P: Replica R processes TRUCK-PREPARE and sends vote to P.

Upon receiving a TRUCK-PREPARE message $prep$ from P, R verifies its validity and, if valid, replies to P with a TRUCK-PREP-VOTE $:= \langle v, dig = h(prep.val) \rangle_R$ message.

3a: P: Replica P assembles a truck certificate.

P aggregates $n - f$ distinct matching TRUCK-PREP-VOTE messages into a truck certificate, and sets *truck-cert* $:= (v, dig, \{\text{TRUCK-PREP-VOTE}\})$.

[1] Sports cars are optimized for speed but are fragile, whereas trucks are slower but more reliable.

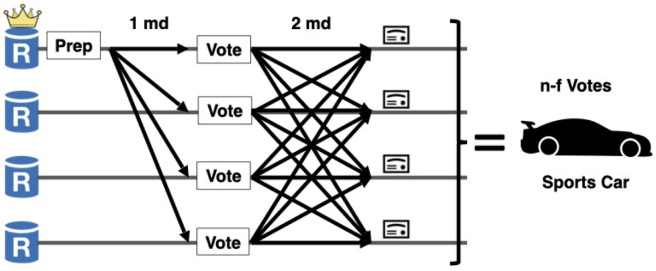


Figure 1: Leader lane protocol pattern.

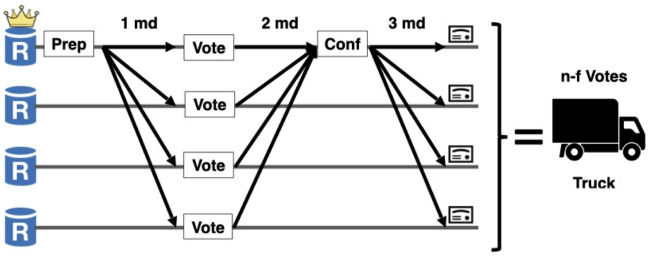


Figure 2: Replica lane protocol pattern.

3b: P → R: Replica P broadcasts TRUCK-CONFIRM.

After forming a truck certificate for that lane, P sends a TRUCK-CONFIRM $:= \langle v, \textit{truck-cert} \rangle_P$ message to all replicas.

5.1.3 Race Cutoff

The race ends locally at a replica once it has received $n-f$ truck certificates. This cutoff is chosen to give the leader as much room as possible to win the race. Since up to f replicas may be faulty, no correct replica can wait for more than $n-f$ truck certificates. At the same time, setting the cutoff below $n-f$ would shorten the race unnecessarily, increasing the chance that the leader is falsely detected as slow even when it is not. Once the cutoff is reached, the replica either continues toward committing the leader's proposal, if it observed the leader win the race, or enters the recovery path, if it observed the leader lose.

1: R: Replica R assembles a race-cutoff certificate.

Upon receiving $n-f$ matching valid TRUCK-CONFIRM messages from distinct replicas for this proposal lane, R assembles a *race-cutoff cert* $:=$ {TRUCK-CONFIRM} certificate. This certifies that the $(n-f)$th certificate has formed and hence cutoff is reached.

After the cutoff, each replica is in one of two cases: either (*i*) it received a sports car certificate prior to the cutoff (Figure 3) or (*ii*) the leader missed the cutoff (Figure 4). As replicas can receive certificates in a different order, their belief about who won the race may diverge. The sports car certificate forming prior to cutoff indicates that the leader is making progress at the expected speed. If instead a cutoff forms first, the leader is behaving unexpectedly slowly. We describe each case in turn.

Case 1: The leader beat the cutoff. Suppose replica R receives the sports car certificate before $n-f$ truck certificates. Then R concludes locally that the leader is behaving as expected, and therefore its proposal should be committed. The protocol then moves toward committing the leader's value.

1: R → R: Replica R broadcasts SC-COMMIT.

R broadcasts SC-COMMIT $:= \langle v, dig \rangle_R$. The SC-COMMIT message serves two purposes: (*i*) it tells other replicas that R locally observed the leader beating the cutoff and that the leader's proposal should therefore be committed; and (*ii*) it ensures that R has persisted the leader's proposal for use in recovery, in case recovery begins before R commits the leader proposal. This is equivalent to the commit phase of traditional BFT protocols.

2: R → R: Replica R assembles and forwards a leader commit certificate.

Replica R aggregates $n-f$ matching SC-COMMIT messages into a leader commit certificate. It then locally commits the leader's value and forwards the commit certificate to all replicas so that they can commit as well. Because the leader lane has a single designated leader, a quorum of SC-COMMIT messages suffices to commit the leader's proposal. Conflicting leader proposals cannot arise in the same view because the truck certificate guarantees non-equivocation. Note that even if R saw the leader missing the cutoff locally, it can still commit the leader's proposal if it receives a quorum of SC-COMMIT messages from other replicas.

Case 2: The leader missed the cutoff. Suppose replica R observes $n-f$ truck certificates before the sports car certificate, then it concludes the leader is slow. It therefore abandons participation in the leader lane and stops sending SC-PREP-VOTE or SC-COMMIT messages. However, it continues processing SC-COMMIT messages from other replicas, since those messages may still certify that the leader's proposal has committed.

5.2 Recovery Path

Once a replica locally reaches the cutoff, it enters the recovery path. The goal of the recovery path is to guarantee commitment of *some* proposal across all replicas. This may be the leader's proposal if some replica committed it during the race, or it may be some other replica's proposal.

In traditional BFT protocols, recovery proceeds in three steps: (*i*) replicas elect a new leader, (*ii*) that leader chooses a value to recover, and (*iii*) the protocol then runs the usual non-equivocation and persistence phases for that value. However, this ordering of steps is problematic in our setting, where liveness does not rely on any synchrony assumptions. If the new leader is identified at the start of recovery, a network adversary can repeatedly target that replica and prevent progress. To avoid this vulnerability, we reverse the usual order of steps. Rather than electing a leader first, every replica initially acts as a potential leader by advancing its own recovery value

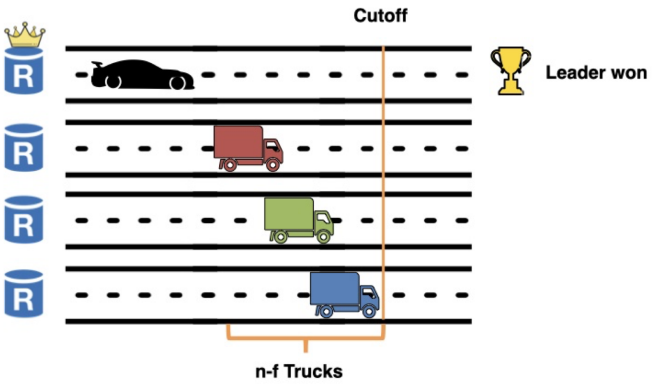


Figure 3: Leader beating the cutoff.

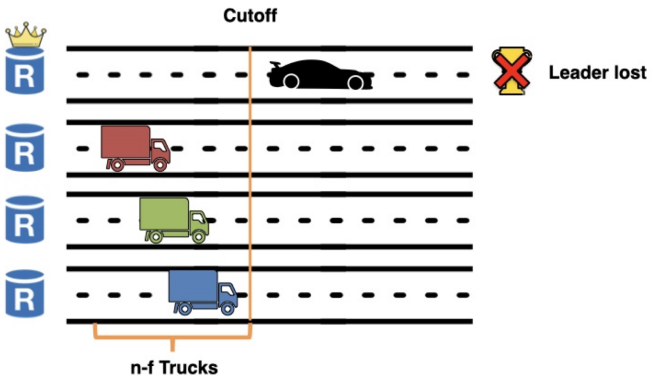


Figure 4: Leader missing the cutoff.

through the consensus steps needed for commitment, namely non-equivocation and persistence. Leader election is deferred until a quorum has completed these steps, so the adversary learns whom to target only at the very end, when it is too late to interfere. This makes it likely that the election selects a replica from that quorum, at which point the protocol can terminate because that value has already been persisted.

Algorithm 2 Replica state cheat sheet for recovery path.

```
1: cert-data   ▷ elected recovery-non-equiv-cert or ⟨⊥_elect⟩
2: qcs         ▷ map: replica ID → recovery path certificate
3: finish      ▷ set of FINISH messages received
```

5.2.1 Step 1: Choosing a value to recover per lane

In Step 1, each replica selects, for its lane, a value to recover based on the evidence gathered during the race. The key safety requirement is that if some value was already committed (via the sports car certificate), then replicas must recover that value.

Replicas first share with each other any messages they received from the leader during the race. This evidence will be used to determine what proposal should be recovered.

1: R → R: Replica R broadcasts STATUS.

The replica broadcasts all state received from the leader during the race, including any proposal received from the leader (*sc-prep*). If a replica did not receive a *sc-prep* it sets $sc\text{-}prep = \langle \perp_{prep} \rangle_R$. Similarly, it sends any sports car certificate it received. If a replica did not receive a sports car certificate it sets $sportscar\text{-}cert = \langle \perp_{sc} \rangle_R$.

R broadcasts this information to all other replicas as part of a STATUS $:= \langle sc\text{-}prep, sportscar\text{-}cert \rangle_R$ message.

2: P receives quorum of STATUS messages

A replica P selects a value to recover for its lane based on a quorum of $n-f$ STATUS messages. P considers three cases:

Case 1: P receives a sports car certificate.

If a sports car certificate exists as part of any received STATUS message, then some replica may have formed a leader commit certificate (follows from quorum intersection). To preserve safety, P therefore sets its recovery input to the leader's value: $r\text{-}input = sportscar\text{-}cert$.

The next two cases handle the situation where P does not receive a sports car certificate. In both cases, P chooses the same recovery input; however, in the latter case P can safely skip the race exclusion phase.

Case 2a: P receives $n-f$ $\langle \perp_{sc} \rangle$ and a SC-PREPARE.

Receiving $n-f$ $\langle \perp_{sc} \rangle$ messages definitively indicates that no correct replica committed on the leader's lane. This also follows from quorum intersection: if a leader commit certificate existed, P would be guaranteed to receive a *sportscar-cert*, which it did not. In principle, P could now safely recover any value, since no leader commit occurred. However, we require P to recover its own truck certificate from the race, thereby preserving the progress already achieved on its lane.

In this case, P sets its recovery input to $r\text{-}input = truck\text{-}cert$, aggregates the $n-f$ signatures on $\langle \perp_{sc} \rangle$ into a no-commit certificate *nc-cert*, which proves that a leader commit could not have occurred, and proceeds to the next step.

Case 2b: P receives $n-f$ $\langle \perp_{sc} \rangle$ and $n-f$ $\langle \perp_{prep} \rangle$.

Receiving $n-f$ $\langle \perp_{prep} \rangle$ messages definitively indicates that no sports car certificate exists. This follows from quorum intersection: if a sports car certificate existed, P would be guaranteed to receive a corresponding SC-PREPARE, which it did not. Since there is no possible sports car certificate to equivocate with, P can safely skip the race exclusion phase (see section 5.2.2): its own racing truck certificate (*truck-cert*) already ensures non-equivocation within its lane. P aggregates the $n-f$ signatures on $\langle \perp_{prep} \rangle$ into a no-lock certificate *nl-cert*, which proves that no sports car certificate exists, and sets its recovery input to $r\text{-}input = (truck\text{-}cert, nl\text{-}cert)$.

5.2.2 Step 2: Persisting the recovered value per lane

In Step 2, each replica persists its recovered value so that its value can be safely committed in Step 3. This step has two parts: the aforementioned optional race exclusion phase followed by a persistence phase that mirrors the familiar two-phase structure of traditional BFT. The optional race exclusion phase is needed because, although the racing path already guarantees non-equivocation within each lane, a Byzantine replica could still try to equivocate between a possible sports car certificate and its own truck certificate. The race exclusion phase prevents this. The persistence phase then plays the usual role from traditional BFT: it records the recovered value so that subsequent protocol steps treat it as the lane's durable candidate.

Race Exclusion Phase. The race exclusion phase prevents a Byzantine replica from equivocating across lanes between its own truck certificate and a possible sports car certificate. By the end of this phase, at most one of these values can advance in P's lane: either P's truck certificate or the sports car certificate, but not both.

1: P → R: Replica P broadcasts REC-PREPARE.

P broadcasts REC-PREPARE $:= \langle v, r\text{-}input, nc\text{-}cert \rangle_P$ where v is the view, *r-input* is the selected recovery input (from step 1), and *nc-cert* is the no-commit certificate formed from case 2a if it exists.

2: R → P: Replica R processes REC-PREPARE and votes.

R checks whether the received REC-PREPARE is valid. If so, R replies by sending REC-PREP-VOTE $:= \langle v, dig = h(r\text{-}input) \rangle_R$ back to the proposer.

3: P: Replica P assembles a quorum of REC-PREP-VOTE.

P waits for at least $n - f$ matching REC-PREP-VOTE messages, and aggregates these votes into a recovery non-equivocation certificate, *recovery-non-equiv-cert* $:= (v, P, dig, \{$REC-PREP-VOTE $\})$ No two recovery non-equivocation certificate can exist for the same lane because the necessary quorums intersect in at least one correct replica (which votes only once). For P's lane, all replicas will either have a recovery non-equivocation certificate for the leader's value or P's value from the racing path but not both.

Persistence Phase.

1: P → R: Replica P broadcasts REC-CONFIRM.

If P could skip the race exclusion phase, P sets *recovery-non-equiv-cert* $=$ *r-input*. This effectively upgrades its truck certificate into a recovery non-equivocation certificate. Otherwise, P uses the recovery non-equivocation certificate it formed in the race exclusion phase. P broadcasts a message REC-CONFIRM $:= \langle v, recovery\text{-}non\text{-}equiv\text{-}cert \rangle_P$ where v is the view, and *recovery-non-equiv-cert* is the recovery non-equivocation certificate P is proposing.

2: R → P: R receives and acknowledges REC-CONFIRM.

R checks whether the received REC-CONFIRM is valid. If the validity checks pass, R stores a copy of the recovery non-equivocation certificate in $qcs[P] =$ *recovery-non-equiv-cert*. Then, it replies by sending REC-CONFIRM-VOTE $:= \langle v, dig = h(recovery\text{-}non\text{-}equiv\text{-}cert\ ) \rangle_R$ back to the proposer.

3: P → R: Replica P broadcasts FINISH.

P waits for at least $n-f$ REC-CONFIRM messages with matching digests, and aggregates these votes into a recovery confirm certificate, *rec-confirm-cert* $:= (v, dig, \{$REC-CONFIRM-VOTE $\})$. It then forwards the recovery confirm certificate it formed in a FINISH $:= \langle v, rec\text{-}confirm\text{-}cert \rangle_P$ message.

4: R: Replica R receives quorum of FINISH messages.

For each FINISH message, R stores a copy of the recovery confirm certificate in $qcs[R_i] =$ *rec-confirm-cert*, where R_i is the replica who proposed *rec-confirm-cert*. Once R receives $n-f$ FINISH messages, it starts the lane election step.

5.2.3 Step 3: Selecting a lane to commit

Unlike traditional BFT, which has a single dedicated proposal lane from the leader, Ambulance has multiple lanes, so it is not immediately clear which lane to commit. To mirror traditional BFT's single-lane behavior, once $n-f$ lanes have completed the persistence phase, Ambulance randomly elects a single lane and checks whether to commit its proposal. If the elected lane has completed the persistence phase, the election succeeds and replicas commit the proposal in that lane. Otherwise, replicas move to the next view and rerun the recovery path. The lane election process must satisfy two properties: (*i*) all correct replicas must agree on who won the election—this is necessary for safety; and (*ii*) more surprisingly, the lucky winner cannot be predicted prior to running the protocol. The latter requirement originates from the potentially adversarial nature of slowdowns: if the "eventual" winner is known prior to its committing, a network adversary could systematically target it. Ambulance achieves both properties by leveraging threshold signatures to elect the winning lane.

Lane Election. Each replica R produces a threshold-signature share on the view number, v, and broadcasts it. After collecting $2f + 1$ valid shares for view v, a replica combines them into a single unique threshold signature σ and computes the elected lane proposer as $E := h(\sigma) \bmod n$, where h is a cryptographic hash function. Agreement follows because σ is unique, so all correct replicas compute the same E; unpredictability follows because E cannot be determined before the shares are revealed.

Commit rule. Replica R commits if it completed the persistence phase for the elected lane ($qcs[E]$ contains a recovery confirm certificate). Otherwise, R stops participating in view v, advances to the next view, and forwards σ to help other replicas derive the same election outcome.

Table 1: RTTs between regions (ms)

RTT	us-east1	us-east2	us-west1	us-west2
us-east1	-	20	72	67
us-east2	21	-	57	55
us-west1	69	53	-	23
us-west2	66	49	24	-

5.3 Retry Protocol

If a replica fails to commit in view 0, it advances to the next view. In all views >0, replicas execute only the recovery path (there is no racing path). The recovery path is identical to the recovery path of view 0, with one small (but key) difference: the race-exclusion phase is always executed as there could be multiple valid lane inputs to recover. We defer the details of the retry protocol to Appendix A.

5.4 Multi-Shot Agreement

For multi-shot agreement, Ambulance constructs a totally ordered log of slots, where each slot runs an independent instance of the single-shot protocol. To avoid sequential delays, Ambulance pipelines these instances, so that a slot can begin without waiting on the previous slot to finish. We adopt the same pipelining strategy as Autobahn [45], where we start a new slot upon receiving a proposal from the previous slot, and cap the number of concurrent uncommitted slots to a pipelining parameter, k.

5.5 Motorization

Autobahn [45] introduced the concept of motorization, which allows any consensus protocol to achieve higher throughput by adopting Autobahn's data layer. Ambulance can use Autobahn's data layer to parallelize the data dissemination process and efficiently utilize all replicas' bandwidth, leading to high throughput.

5.6 Implementation Notes

Leader message sharing. For liveness, the leader needs to run both a fast-track lane and a replica lane. Instead of sending separate messages for each lane, a single message can be used with a boolean flag indicating the lane it belongs to.

All-to-all communication. The recovery path utilizes linear style communication, where replicas forward their votes to the proposer instead of broadcasting them to all replicas. This reduces communication complexity at the trade-off of increasing latency. To reduce recovery path latency further, Ambulance supports an all-to-all voting strategy to shave off extra message delays.

Signature aggregation. Ambulance uses threshold signatures, but only for the lane election phase. We can also adopt threshold signatures for protocol votes to reduce the communication complexity of Ambulance by a factor of n. In practice, this is only necessary for large values of n.

6 Evaluation

Our evaluation seeks to answer three questions:

Table 2: Default timeout values used in production systems

System	Timeout
CockroachDB [3]	2 s
Microsoft CCF [10]	5 s
Neo4j [69]	7 s
TiKV [16]	10 s
HotShot [12]	12 s

1. **Performance**: How well does Ambulance perform in the absence of slowdowns? (§6.1)
2. **Slowdown Robustness**: How well does Ambulance tolerate slowdowns? (§6.2)
3. **Real-world Performance**: How well does Ambulance perform in production under realistic conditions? (§6.3)

We implemented a prototype of Ambulance[2] in Rust, using the same codebase as the open source implementation of Autobahn [45]. We implement all the notes in §5.6 except for signature aggregation. We use Tokio's TCP [17] for networking, and RocksDB [14] for persistent storage of payload data to disk. Finally, we use ed25519-dalek [6] signatures for authentication.

Baseline systems. We compare Ambulance against several state-of-the-art BFT protocols: 1) Autobahn [45], a leading timeout-based BFT protocol, 2) ParBFT2 [37], a state-of-the-art hedging-based protocol, as well as 3) SMVBA (Speeding Dumbo) [49], a state-of-the-art pessimistic (asynchronous) protocol. ParBFT2 uses HotStuff [83] as its optimistic path, and after a hedging delay, it launches its pessimistic path, which runs SMVBA combined with an asynchronous binary agreement protocol [18]. We use the available open-source codebase each time[3]. Ambulance and Autobahn share the same data layer design, where all replicas disseminate batches through structured data lanes, enabling parallelized data dissemination. In contrast, ParBFT2 and SMVBA adopt the Batched HotStuff data layer [38], where replicas optimistically stream batches and consensus leaders propose hashes of any batches they have received.

Ambulance and Autobahn use a PBFT-style multi-shot consensus with a totally ordered log, allowing multiple consensus instances to be pipelined in parallel. This eliminates sequential delays and reduces end-to-end latency. ParBFT2 and SMVBA instead follow the HotStuff [83] chaining approach, which introduces inherent sequential latency. In ParBFT2's optimistic path, a new consensus instance begins only after the leader collects a quorum of votes for the previous one. In its pessimistic path-and in SMVBA—there is no pipelining at all; consensus instances run strictly sequentially. As a result, requests must wait for the next consensus instance to start, adding further end-to-end latency.

Evaluation Setup. We evaluate all systems on AWS EC2. For sections §6.1 and 6.2, we evenly distributed nodes in `us-west1`, `us-west2`, `us-east1` and `us-east2`.

[2] https://github.com/neilgiri/ambulance-artifact

[3] We validated our Autobahn and ParBFT2 results with respective authors

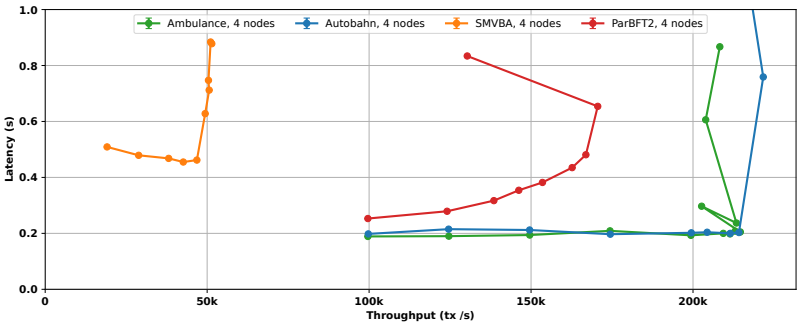


Figure 5: Throughput and Latency under increasing load.

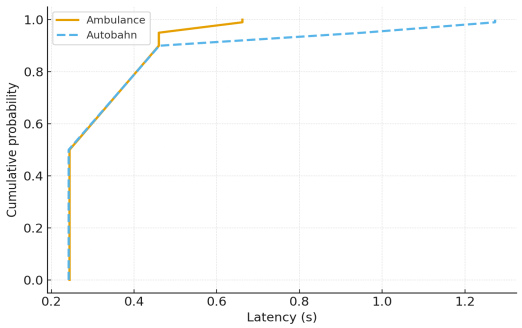


Figure 6: Production Latency CDF

We summarize RTTs between regions in Table 1. We use machine type `m6a.4xlarge` [1] (16 vCPUs and 64GB RAM) with 30GB gp3 disk (SSD) and 12.5GB/s network bandwidth. Clients are co-located in the same region as the replica, and issue a constant stream of no-op transactions (tx), consisting of 512 random bytes [38, 75]; no-op transactions allow us to stress the consensus system, without risking being bottlenecked on execution. We set a batch size of 500KB (1000 transactions) for all baselines, but allow consensus proposals to include/reference more than one batch if available; this *mini-batching* design [40, 41] allows replicas to organically reach larger effective batch sizes with reduced latency trade-off. For the systems that use leaders, they are elected using a deterministic round-robin scheme.

6.1 Common Case Performance

Figure 5 presents performance in a fault-free, synchronous setting with $n=4$ nodes. Each experiment runs for 60 seconds under increasing client load. Ambulance reaches a peak throughput of 214k tx/s, matching Autobahn's peak throughput of 214k tx/s. This high throughput stems from motorization as Ambulance adopts Autobahn's data layer. Ambulance makes efficient use of network bandwidth, which dominates overall cost, as consensus messages are small compared to large batched proposals. Although Ambulance introduces multiple consensus lanes, the metadata overhead is negligible in practice, and both Ambulance and Autobahn ultimately bottleneck on data layer work (serializing/deserializing).

Ambulance achieves a latency of 205 ms compared to Autobahn's 203 ms. In the absence of failures and synchrony, both systems essentially run PBFT, leading to similar latency.

ParBFT2 reaches a peak throughput of 167k tx/s, substantially higher than SMVBA but 1.3x lower than Ambulance. This gap is primarily due to its weaker data layer, which incurs substantial data-synchronization overhead at high load [38, 45], limiting throughput even when consensus is not the bottleneck. In the good case, ParBFT2's consensus latency is five message delays, compared to three for Ambulance, contributing to its 382 ms latency (1.9x higher than Ambulance).

SMVBA achieves only 50.6k tx/s peak throughput and exhibits the highest latency of 462 ms (2.3x higher than Ambulance). It shares the same data layer as ParBFT2, but its low throughput follows directly from its sequential consensus design: consensus instances run strictly one after another, so throughput is constrained by consensus latency. In the good case, SMVBA's consensus latency is six message delays, more than ParBFT2 (five) and Ambulance (three). Together with the additional inclusion latency introduced by this sequential design, these extra message delays yield higher end-to-end latency than Ambulance.

Overall, Ambulance matches the common-case throughput and latency of state-of-the-art timeout-based BFT (Autobahn) while outperforming pessimistic and hedging-based systems.

6.2 Slowdown Robustness

The previous section demonstrated that Ambulance performs well under ideal conditions. We now examine Ambulance's tolerance to slowdowns of varying severity in an $n=4$ deployment. At time $t=3$ seconds, we inject a transient slowdown whose duration we vary across experiments. The slowdown is introduced by pausing all processing (i.e., sleeping) on a single replica. For ParBFT2, we run only the pessimistic path and do not inject slowdowns, due to bugs in the optimistic–pessimistic path switching logic.[4] This configuration still captures slowdown latency but does not reflect ParBFT2's common-case latency, which should closely match Figure 5 (382 ms).

We plot latency over time for slowdown durations of 1, 2, 5, 7, and 10 seconds, where each point is the median latency over a 500 ms window across 5 runs. For Autobahn, we use the timeout values from Table 2, which are taken from production deployments of major reliable datastores.

In the slowdown experiments, we run Autobahn with two different leader-election timeout values. One timeout value is set above the injected slowdown (timeouts of 2, 5, 7, 10, and 12 seconds for slowdowns of 1, 2, 5, 7, and 10 seconds, respectively) so that the plots capture the full effect of the slowdown. The other timeout value is set below the slowdown duration, so that the timeout detects the slowdown. Using a more aggressive timeout for a given slowdown duration caps the peak latency at the timeout value, making it closely match the peak latency of a smaller slowdown. For ParBFT2, we use

[4] We are working with the authors of [37] to resolve these issues.

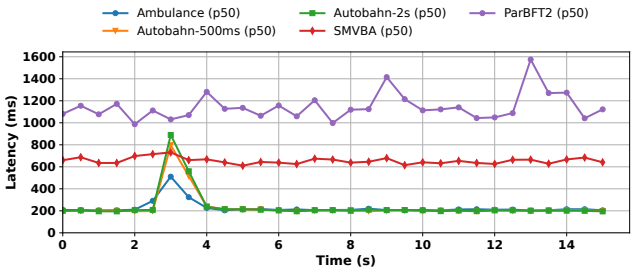


Figure 7: 1s slowdown

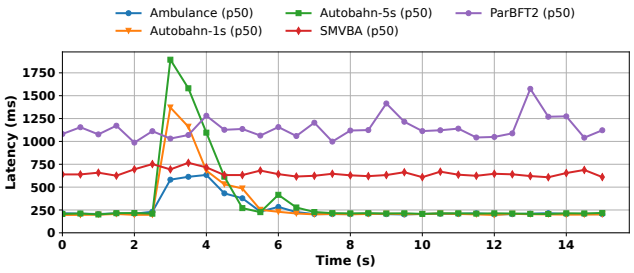


Figure 8: 2s slowdown

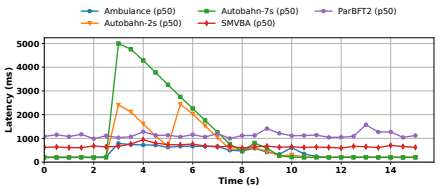


Figure 9: 5s slowdown

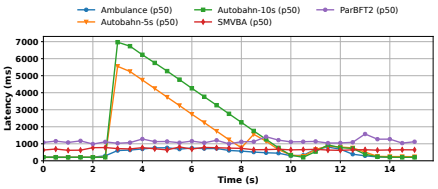


Figure 10: 7s slowdown

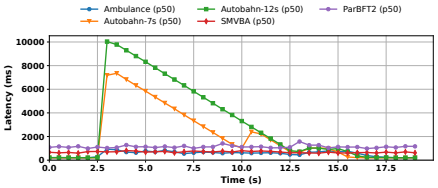


Figure 11: 10s slowdown

an aggressive 75 ms hedging delay to minimize slowdown latency, consistent with the maximum RTT in Table 1 ([37] specifies a hedging delay of 2Δ, equal to the maximum RTT).

1 second slowdown (Figure 7). All systems are evaluated under a conservative load ([70]) to minimize queuing effects (Ambulance and Autobahn at 100k, and ParBFT2 and SMVBA at 10k). Ambulance has the lowest peak latency at 510.4 ms. This is 1.6x better than Autobahn with a 500 ms timeout (796.1 ms), 1.7x better latency than Autobahn with a 2 s timeout (889.5 ms), 1.4x better than SMVBA (731 ms), and 3.1x better than ParBFT2 (1.58 s). Ambulance promptly detects the slowdown and switches to the recovery path, which takes 7 message delays for the first view. Autobahn, configured with a 2 s timeout (from Table 2), does not detect the slowdown, causing latency to rise proportionally to the slowdown duration as replicas block on the slow leader. Autobahn with a 500 ms timeout detects the slowdown; however, it must elect a new leader and pay the consensus latency cost, so the overall latency ends up being similar to Autobahn with a 2 s timeout. SMVBA's consensus latency is similar to Ambulance's, but its lack of pipelining increases end-to-end inclusion latency. ParBFT2 exhibits the highest latency due to the hedging delay and the requirement to run both SMVBA and an asynchronous binary agreement protocol for its pessimistic path.

2 second slowdown (Figure 8). With a 2 second slowdown, Ambulance's peak latency increases mildly to 633.4 ms, but is still 2.2x better than Autobahn with a 1 s timeout (1.37 s), 3.0x better than Autobahn with a 5 s timeout (1.89 s), 1.2x better than SMVBA (766 ms), and 2.5x better than ParBFT2 (1.58s). Longer slowdowns increase the probability that multiple recovery views are needed, raising Ambulance and SMVBA's latency. Autobahn with a 5 s timeout again fails to detect the slowdown, resulting in latency degradation proportional to the slowdown duration. Autobahn with a 1 s timeout detects the slowdown after 1 second, but must also pay the consensus latency afterwards, so the latency is still higher compared to Ambulance. ParBFT2 behaves similarly to the 1 s case. Figures 9–11.

5-10 second slowdowns (Figures 9-11) The same trend holds. Ambulance reaches peak latencies of 787.7 ms, 810 ms, and 932 ms for 5, 7, and 10 second slowdowns, respectively, increasing only modestly due to additional failed views. These are still 3.1-6.3x, 6.9-8.6x, and 7.9-10.8x better than Autobahn for 5, 7, and 10 second slowdowns, respectively. SMVBA closely tracks Ambulance at these longer slowdowns (with peak latencies of 812 ms and 945 ms), while ParBFT2 remains at 1.58 s (1.7x worse than Ambulance).

Across all slowdown durations, Ambulance consistently delivers lower peak latency than Autobahn—1.6x–3.0x better for short (1–2 second) slowdowns and up to 10.8x better for more severe (5–10 second) slowdowns. Ambulance also closely matches SMVBA, the state-of-the-art pessimistic protocol, while achieving 1.7x–3.1x lower latency than ParBFT2, the state-of-the-art hedging protocol.

6.3 Production Performance

We evaluate the performance of Ambulance as deployed in Sei [65], a production-grade distributed ledger system, and compare it to their current implementation of Autobahn. The deployment consists of $n = 40$ replicas distributed across 20 AWS EC2 regions in North America, South America, Asia, and Europe. Each replica runs on an `m6i.12xlarge` [2] instance (48 vCPUs, 192 GB RAM, 18.75 GB/s network bandwidth).

Figure 6 reports latency over a 24-hour production run. The system experiences slowdowns approximately once every thousand slots, consistent with rates observed in deployed distributed ledger systems [12, 13, 65]. Both Ambulance and the production Autobahn implementation operate at 180k load (with a peak throughput of 220k) on the same workload;

Autobahn uses 2-second timeouts. Ambulance matches the median latency of Autobahn (244ms vs 242ms), while significantly improving tail performance: its p99 latency of 662 ms represents a 1.92x reduction compared to Autobahn's 1.27 s. Because the slowdown events dominate the p99, these results show that Ambulance maintains Autobahn's common-case latency in production while substantially reducing latency during slowdowns.

7 Related Work

Ambulance is inspired by several prior works. Ambulance's leader racing path draws from traditional BFT protocols, in particular, PBFT [30]. The recovery path draws from the VABA (MVBA) family of asynchronous protocols, in particular, 2PAC [73]. Ambulance directly adopts Autobahn's [45] data layer and pipelining strategy to achieve high throughput and reduce end-to-end latency.

Slowdown tolerance and failure detectors The most common failure detectors [32, 33, 55] rely on heartbeats or timeouts. These mechanisms are slow to react to failures, and are fundamentally not cooperative nor productive. Other failure detection mechanisms [21, 52, 58, 61] are less generic and inspect the actual processes themselves. These mechanisms can detect more types of failures, including gray failures, but do not easily extend to Byzantine environments. PeerReview [51] does detect provably faulty behavior but can neither detect nor correct slow behavior. Copilot [70] is the first SMR protocol to introduce slowdown tolerance, limited to the CFT setting. It defines and satisfies the *s*-slowdown-tolerance property, which guarantees performance similar to the no-slowdown case even with *s* slow replicas. We instead assess slowdown robustness by measuring each protocol's latency under slowdown, since even Copilot can suffer performance degradation in certain slowdown scenarios [62].

Partially synchronous BFT These protocols [25, 26, 30, 31, 44, 46, 47, 50, 56, 68, 74, 76, 78, 83] are optimized for the failure-free case, minimizing latency when there are no slowdowns. Because they rely on timeouts to detect failures, a slow leader causes latency to spike by an amount proportional to the leader timeout or the duration of the slowdown.

Robust BFT Aardvark [34] monitors leader performance and invokes view changes if the leader does not meet performance thresholds. The system offers only partial protection against slowdowns [70] as it cannot easily detect transient slowdowns. Autobahn [45] is robust to blips, a subset of slowdowns that includes network events such as partial partitions or replica failures that trigger timeouts. It ensures that latency returns to baseline immediately after a blip but allows latency to spike during the blip.

Asynchronous BFT protocols [19, 38, 54, 66] are pessimistic: they assume the leader always loses the race against the clock. There are two main families of asynchronous BFT protocols: the HoneyBadger (BKR) family [24, 60, 66, 82] and the VABA family [19, 42, 49, 73, 77]. HoneyBadger-style protocols typically run *n* instances of reliable broadcast [29] and *n* instances of binary agreement to decide which broadcasts completed, while VABA-style protocols run *n* instances of a leader-based BFT protocol and use a common coin to randomly select one instance to commit. Asynchronous protocols have lower latency than traditional BFT during slowdowns, since they do not require a failure detection mechanism such as timeouts. However, they have much higher common-case latency than traditional BFT protocols, limiting their adoption in practice.

Hybrid BFT protocols [28, 36, 37, 43, 64, 72, 75] aim to combine the advantages of traditional and asynchronous BFT. Ditto [43] and BDT [64] run a traditional BFT protocol first and fall back to an asynchronous one when a timeout fires. Relying on timeouts causes them to suffer high latency during slowdowns, particularly when switching protocols. ParBFT [37], Abraxas [28], and Ipotane [36] instead run one or more asynchronous protocol instances to detect failures of the traditional BFT protocol. Unfortunately, commitment during a slowdown requires still requires completing multiple asynchronous instances. Hedging [37, 80] detects failures by having the leader race against other replicas, but giving the leader a time-based head start. This mechanism is cooperative, but not productive; it stalls recovery until the hedging delay is over.

Leaderless SMR protocols [39, 67, 81] distribute the ordering task across multiple replicas instead of relying on a single leader. Because no single leader orders all requests, replicas commit dependency graphs that induce an order among instances. This design avoids stalling on a single slow leader, but slowdowns still harm performance: replicas can depend on commands proposed by slow replicas and must wait for those commands to commit before executing. In the worst case, they must invoke an expensive recovery procedure for those commands.

8 Conclusion

This paper introduces Ambulance, a novel BFT protocol, that uses a novel *protocol-rigged racing* mechanism to detect slowdowns. This allows Ambulance to offer the best of both worlds: the good-interval performance of timeout-based systems and slowdown performance comparable to asynchronous protocols.

Acknowledgments

We thank our shepherd, and the anonymous reviewers for their thorough and insightful comments. This work was supported in part by the NSF grants CSR-CORE 2106842, NSF CAREER 2442542, a Sui Research Award as well as gifts from Accenture, AMD, Anyscale, Google, IBM, Intel, Mohamed Bin Zayed University of Artificial Intelligence, Samsung SDS, SAP, and VMware.

References

[1] [n. d.]. Amazon EC2 M6a Instances. https://aws.amazon.com/ec2/instance-types/m6a/ (last accessed on 12/09/25).

[2] [n. d.]. Amazon EC2 M6i Instances. https://aws.amazon.com/ec2/instance-types/m6i/ (last accessed on 12/09/25).

[3] [n. d.]. CockroachDB Replication Layer. https://www.cockroachlabs.com/docs/stable/architecture/replication-layer (last accessed on 12/09/25).

[4] [n. d.]. *Confidential Consortium Framework, Microsoft*. https://ccf.microsoft.com/ (last accessed on 09/23/24).

[5] [n. d.]. Consider rolling the WAL if the HDFS write pipeline is slow. https://issues.apache.org/jira/browse/HBASE-22301 (last accessed on 12/09/25).

[6] [n. d.]. Dalek elliptic curve cryptography. https://github.com/dalek-cryptography/ed25519-dalek (last accessed on 09/23/24).

[7] [n. d.]. Delayed heartbeat from etcd leader. https://github.com/etcd-io/etcd/issues/7312 (last accessed on 12/09/25).

[8] [n. d.]. Digital Euro. https://www.ecb.europa.eu/euro/digital_euro/html/index.en.html (last accessed on 09/23/24).

[9] [n. d.]. etcd Tuning. https://etcd.io/docs/v3.4/tuning/ (last accessed on 12/09/25).

[10] [n. d.]. Microsoft CCF Configuration. https://microsoft.github.io/CCF/main/operations/configuration.html (last accessed on 12/09/25).

[11] [n. d.]. minimum master nodes does not prevent split-brain if splits are intersecting. https://github.com/elastic/elasticsearch/issues/2488 (last accessed on 12/09/25).

[12] [n. d.]. Private communications with engineers at the blockchain company, Espresso, running HotStuff in production. March 2025.

[13] [n. d.]. Private communications with researchers at Mysten Labs, a leading blockchain company), and formerly of Facebook Novi. March 2024.

[14] [n. d.]. RocksDB, version 0.16.0. https://rocksdb.org/ (last accessed on 09/23/24).

[15] [n. d.]. Sui Blockchain. https://sui.io/ (last accessed on 09/23/24).

[16] [n. d.]. TiKV Config. https://tikv.org/docs/6.1/deploy/configure/tikv-configuration-file/ (last accessed on 12/09/25).

[17] [n. d.]. Tokio, version 1.5.0. https://tokio.rs/ (last accessed on 09/23/24).

[18] Ittai Abraham, Naama Ben-David, and Sravya Yandamuri. 2022. Efficient and Adaptively Secure Asynchronous Binary Agreement via Binding Crusader Agreement. In *Proceedings of the 2022 ACM Symposium on Principles of Distributed Computing*. 381–391.

[19] Ittai Abraham, Dahlia Malkhi, and Alexander Spiegelman. 2019. Asymptotically Optimal Validated Asynchronous Byzantine Agreement. In *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing* (Toronto ON, Canada) *(PODC '19)*. Association for Computing Machinery, New York, NY, USA, 337–346. https://doi.org/10.1145/3293611.3331612

[20] Ittai Abraham, Kartik Nayak, Ling Ren, and Zhuolun Xiang. 2021. Good-Case Latency of Byzantine Broadcast: A Complete Categorization. In *Proceedings of the 2021 ACM Symposium on Principles of Distributed Computing* (Virtual Event, Italy) *(PODC'21)*. Association for Computing Machinery, New York, NY, USA, 331–341. https://doi.org/10.1145/3465084.3467899

[21] Marcos K. Aguilera and Michael Walfish. 2009. No time for asynchrony. In *Proceedings of the 12th Conference on Hot Topics in Operating Systems* (Monte Verità, Switzerland) *(HotOS'09)*. USENIX Association, USA, 3.

[22] Mohammed Alfatafta, Basil Alkhatib, Ahmed Alquraan, and Samer Al-Kiswany. 2020. Toward a Generic Fault Tolerance Technique for Partial Network Partitioning. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association, 351–368. https://www.usenix.org/conference/osdi20/presentation/alfatafta

[23] Ahmed Alquraan, Hatem Takruri, Mohammed Alfatafta, and Samer Al-Kiswany. 2018. An Analysis of Network-Partitioning Failures in Cloud Systems. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. USENIX Association, Carlsbad, CA, 51–68. https://www.usenix.org/conference/osdi18/presentation/alquraan

[24] Diogo S. Antunes, Afonso N. Oliveira, André Breda, Matheus Guilherme Franco, Henrique Moniz, and

Rodrigo Rodrigues. 2024. Alea-BFT: Practical Asynchronous Byzantine Fault Tolerance. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*. USENIX Association, Santa Clara, CA, 313–328. https://www.usenix.org/conference/nsdi24/presentation/antunes

[25] Balaji Arun, Zekun Li, Florian Suri-Payer, Sourav Das, and Alexander Spiegelman. 2025. Shoal++: High Throughput DAG BFT Can Be Fast and Robust!. In *22nd USENIX Symposium on Networked Systems Design and Implementation (NSDI 25)*. USENIX Association, Philadelphia, PA, 813–826. https://www.usenix.org/conference/nsdi25/presentation/arun

[26] Kushal Babel, Andrey Chursin, George Danezis, Lefteris Kokoris-Kogias, and Alberto Sonnino. 2023. Mysticeti: Low-Latency DAG Consensus with Fast Commit Path. *arXiv preprint arXiv:2310.14821* (2023).

[27] Mathieu Baudet, Avery Ching, Andrey Chursin, George Danezis, François Garillot, Zekun Li, Dahlia Malkhi, Oded Naor, Dmitri Perelman, and Alberto Sonnino. 2019. State machine replication in the Libra Blockchain. *The Libra Association Technical Report* (2019).

[28] Erica Blum, Jonathan Katz, Julian Loss, Kartik Nayak, and Simon Ochsenreither. 2023. Abraxas: Throughput-Efficient Hybrid Asynchronous Consensus. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security* (Copenhagen, Denmark) *(CCS '23)*. Association for Computing Machinery, New York, NY, USA, 519–533. https://doi.org/10.1145/3576915.3623191

[29] Gabriel Bracha and Sam Toueg. 1985. Asynchronous consensus and broadcast protocols. *Journal of the ACM (JACM)* 32, 4 (1985), 824–840.

[30] Miguel Castro and Barbara Liskov. 1999. Practical Byzantine Fault Tolerance. In *Proceedings of the Third Symposium on Operating Systems Design and Implementation* (New Orleans, Louisiana, USA) *(OSDI '99)*. USENIX Association, USA, 173–186.

[31] Benjamin Y. Chan and Rafael Pass. 2023. Simplex Consensus: A Simple and Fast Consensus Protocol. In *Theory of Cryptography: 21st International Conference, TCC 2023, Taipei, Taiwan, November 29–December 2, 2023, Proceedings, Part IV* (Taipei, Taiwan). Springer-Verlag, Berlin, Heidelberg, 452–479. https://doi.org/10.1007/978-3-031-48624-1_17

[32] Tushar Deepak Chandra, Vassos Hadzilacos, and Sam Toueg. 1996. The weakest failure detector for solving consensus. *J. ACM* 43, 4 (July 1996), 685–722. https://doi.org/10.1145/234533.234549

[33] Tushar Deepak Chandra and Sam Toueg. 1996. Unreliable failure detectors for reliable distributed systems. *J. ACM* 43, 2 (March 1996), 225–267. https://doi.org/10.1145/226643.226647

[34] Allen Clement, Edmund Wong, Lorenzo Alvisi, Mike Dahlin, and Mirco Marchetti. 2009. Making Byzantine Fault Tolerant Systems Tolerate Byzantine Faults. In *Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation* (Boston, Massachusetts) *(NSDI'09)*. USENIX Association, USA, 153–168.

[35] Graeme Connell, Vivian Fang, Rolfe Schmidt, Emma Dauterman, and Raluca Ada Popa. 2024. Secret Key Recovery in a Global-Scale End-to-End Encryption System. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*. USENIX Association, Santa Clara, CA, 703–719. https://www.usenix.org/conference/osdi24/presentation/connell

[36] Xiaohai Dai, Chaozheng Ding, Hai Jin, Julian Loss, and Ling Ren. 2024. Ipotane: Balancing the Good and Bad Cases of Asynchronous BFT. Cryptology ePrint Archive, Paper 2024/653. https://doi.org/10.14722/ndss.2026.230003

[37] Xiaohai Dai, Bolin Zhang, Hai Jin, and Ling Ren. 2023. ParBFT: Faster Asynchronous BFT Consensus with a Parallel Optimistic Path. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security* (Copenhagen, Denmark) *(CCS '23)*. Association for Computing Machinery, New York, NY, USA, 504–518. https://doi.org/10.1145/3576915.3623101

[38] George Danezis, Lefteris Kokoris-Kogias, Alberto Sonnino, and Alexander Spiegelman. 2022. Narwhal and Tusk: a DAG-based mempool and efficient BFT consensus. In *Proceedings of the Seventeenth European Conference on Computer Systems*. 34–50.

[39] Vitor Enes, Carlos Baquero, Tuanir França Rezende, Alexey Gotsman, Matthieu Perrin, and Pierre Sutra. 2020. State-machine replication for planet-scale systems. In *Proceedings of the Fifteenth European Conference on Computer Systems* (Heraklion, Greece) *(EuroSys '20)*. Association for Computing Machinery, New York, NY, USA, Article 24, 15 pages. https://doi.org/10.1145/3342195.3387543

[40] Novi Facebook Research. [n.d.]. Hotstuff Implementation. https://github.com/asonnino/hotstuff/commit/d771d4868db301bcb5e3deaa915b5017220463f6 (last accessed on 09/10/24).

[41] Novi Facebook Research. [n.d.]. Narwahl and Bullshark implementation. https://github.com/asonnino/narwhal (last accessed on 09/23/24).

[42] Yingzi Gao, Yuan Lu, Zhenliang Lu, Qiang Tang, Jing Xu, and Zhenfeng Zhang. 2022. Dumbo-ng: Fast asynchronous bft consensus with throughput-oblivious latency. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*. 1187–1201.

[43] Rati Gelashvili, Lefteris Kokoris-Kogias, Alberto Sonnino, Alexander Spiegelman, and Zhuolun Xiang. 2022. Jolteon and Ditto: Network-Adaptive Efficient Consensus with Asynchronous Fallback. In *Financial Cryptography and Data Security: 26th International Conference, FC 2022, Grenada, May 2–6, 2022, Revised Selected Papers* (Grenada, Grenada). Springer-Verlag, Berlin, Heidelberg, 296–315. https://doi.org/10.1007/978-3-031-18283-9_14

[44] Neil Giridharan, Heidi Howard, Ittai Abraham, Natacha Crooks, and Alin Tomescu. 2021. No-commit proofs: Defeating livelock in bft. *Cryptology ePrint Archive* (2021).

[45] Neil Giridharan, Florian Suri-Payer, Ittai Abraham, Lorenzo Alvisi, and Natacha Crooks. 2024. Autobahn: Seamless high speed BFT. In *Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles* (Austin, TX, USA) *(SOSP '24)*. Association for Computing Machinery, New York, NY, USA, 1–23. https://doi.org/10.1145/3694715.3695942

[46] Neil Giridharan, Florian Suri-Payer, Matthew Ding, Heidi Howard, Ittai Abraham, and Natacha Crooks. 2023. BeeGees: stayin'alive in chained BFT. In *Proceedings of the 2023 ACM Symposium on Principles of Distributed Computing*. 233–243.

[47] Guy Golan Gueta, Ittai Abraham, Shelly Grossman, Dahlia Malkhi, Benny Pinkas, Michael Reiter, Dragos-Adrian Seredinschi, Orr Tamir, and Alin Tomescu. 2019. SBFT: A Scalable and Decentralized Trust Infrastructure. In *2019 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE, USA, 568–580. https://doi.org/10.1109/DSN.2019.00063

[48] Haryadi S. Gunawi, Riza O. Suminto, Russell Sears, Casey Golliher, Swaminathan Sundararaman, Xing Lin, Tim Emami, Weiguang Sheng, Nematollah Bidokhti, Caitie McCaffrey, Gary Grider, Parks M. Fields, Kevin Harms, Robert B. Ross, Andree Jacobson, Robert Ricci, Kirk Webb, Peter Alvaro, H. Birali Runesha, Mingzhe Hao, and Huaicheng Li. 2018. Fail-Slow at Scale: Evidence of Hardware Performance Faults in Large Production Systems. In *16th USENIX Conference on File and Storage Technologies (FAST 18)*. USENIX Association, Oakland, CA, 1–14. https://www.usenix.org/conference/fast18/presentation/gunawi

[49] Bingyong Guo, Yuan Lu, Zhenliang Lu, Qiang Tang, Jing Xu, and Zhenfeng Zhang. 2022. Speeding Dumbo: Pushing Asynchronous BFT Closer to Practice. Cryptology ePrint Archive, Paper 2022/027. https://eprint.iacr.org/2022/027

[50] Suyash Gupta, Jelle Hellings, and Mohammad Sadoghi. 2021. RCC: resilient concurrent consensus for high-throughput secure transaction processing. In *2021 IEEE 37th International Conference on Data Engineering (ICDE)*. IEEE, 1392–1403.

[51] Andreas Haeberlen, Petr Kouznetsov, and Peter Druschel. 2007. PeerReview: practical accountability for distributed systems. *SIGOPS Oper. Syst. Rev.* 41, 6 (Oct. 2007), 175–188. https://doi.org/10.1145/1323293.1294279

[52] Peng Huang, Chuanxiong Guo, Jacob R. Lorch, Lidong Zhou, and Yingnong Dang. 2018. Capturing and Enhancing In Situ System Observability for Failure Detection. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. USENIX Association, Carlsbad, CA, 1–16. https://www.usenix.org/conference/osdi18/presentation/huang

[53] Peng Huang, Chuanxiong Guo, Lidong Zhou, Jacob R. Lorch, Yingnong Dang, Murali Chintalapati, and Randolph Yao. 2017. Gray Failure: The Achilles' Heel of Cloud-Scale Systems. In *Proceedings of the 16th Workshop on Hot Topics in Operating Systems* (Whistler, BC, Canada) *(HotOS '17)*. Association for Computing Machinery, New York, NY, USA, 150–155. https://doi.org/10.1145/3102980.3103005

[54] Philipp Jovanovic, Lefteris Kokoris Kogias, Bryan Kumara, Alberto Sonnino, Pasindu Tennage, and Igor Zablotchi. 2024. Mahi-Mahi: Low-Latency Asynchronous BFT DAG-Based Consensus. arXiv:2410.08670 [cs.DC] https://arxiv.org/abs/2410.08670

[55] Idit Keidar and Alexander Shraer. 2006. Timeliness, failure-detectors, and consensus performance. In *Proceedings of the Twenty-Fifth Annual ACM Symposium on Principles of Distributed Computing* (Denver, Colorado, USA) *(PODC '06)*. Association for Computing Machinery, New York, NY, USA, 169–178. https://doi.org/10.1145/1146381.1146408

[56] Ramakrishna Kotla, Lorenzo Alvisi, Mike Dahlin, Allen Clement, and Edmund Wong. 2010.

Zyzzyva: Speculative Byzantine Fault Tolerance. *ACM Transactions on Computer Systems (TOCS)* 27, 4, Article 7 (Jan. 2010), 39 pages. https://doi.org/10.1145/1658357.1658358

[57] S Krishnapriya and Greeshma Sarath. 2020. Securing Land Registration using Blockchain. *Procedia computer science* 171 (2020), 1708–1715.

[58] Joshua B. Leners, Hao Wu, Wei-Lun Hung, Marcos K. Aguilera, and Michael Walfish. 2011. Detecting failures in distributed systems with the Falcon spy network. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles* (Cascais, Portugal) *(SOSP '11)*. Association for Computing Machinery, New York, NY, USA, 279–294. https://doi.org/10.1145/2043556.2043583

[59] Tom Lianza and Chris Snook. [n. d.]. A Byzantine failure in the real world. https://blog.cloudflare.com/a-byzantine-failure-in-the-real-world/

[60] Shengyun Liu, Wenbo Xu, Chen Shan, Xiaofeng Yan, Tianjing Xu, Bo Wang, Lei Fan, Fuxi Deng, Ying Yan, and Hui Zhang. 2023. Flexible Advancement in Asynchronous BFT Consensus. In *Proceedings of the 29th Symposium on Operating Systems Principles*. 264–280.

[61] Chang Lou, Peng Huang, and Scott Smith. 2019. Comprehensive and Efficient Runtime Checking in System Software through Watchdogs. In *Proceedings of the Workshop on Hot Topics in Operating Systems* (Bertinoro, Italy) *(HotOS '19)*. Association for Computing Machinery, New York, NY, USA, 51–57. https://doi.org/10.1145/3317550.3321440

[62] Ruiming Lu, Yunchi Lu, Yuxuan Jiang, Guangtao Xue, and Peng Huang. 2025. One-size-fits-none: understanding and enhancing slow-fault tolerance in modern distributed systems. In *Proceedings of the 22nd USENIX Symposium on Networked Systems Design and Implementation* (Philadelphia, PA, USA) *(NSDI '25)*. USENIX Association, USA, Article 20, 20 pages.

[63] Ruiming Lu, Erci Xu, Yiming Zhang, Zhaosheng Zhu, Mengtian Wang, Zongpeng Zhu, Guangtao Xue, Minglu Li, and Jiesheng Wu. 2022. NVMe SSD Failures in the Field: the Fail-Stop and the Fail-Slow. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*. USENIX Association, Carlsbad, CA, 1005–1020. https://www.usenix.org/conference/atc22/presentation/lu

[64] Yuan Lu, Zhenliang Lu, and Qiang Tang. 2022. Bolt-Dumbo Transformer: Asynchronous Consensus As Fast As the Pipelined BFT. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security* (Los Angeles, CA, USA) *(CCS '22)*. Association for Computing Machinery, New York, NY, USA, 2159–2173. https://doi.org/10.1145/3548606.3559346

[65] Benjamin Marsh, Steven Landers, and Jayendra Jog. 2025. Sei Giga. arXiv:2505.14914 [cs.DC] https://arxiv.org/abs/2505.14914

[66] Andrew Miller, Yu Xia, Kyle Croman, Elaine Shi, and Dawn Song. 2016. The Honey Badger of BFT Protocols. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security* (Vienna, Austria) *(CCS '16)*. Association for Computing Machinery, New York, NY, USA, 31–42. https://doi.org/10.1145/2976749.2978399

[67] Iulian Moraru, David G. Andersen, and Michael Kaminsky. 2013. There is more consensus in Egalitarian parliaments. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles* (Farminton, Pennsylvania) *(SOSP '13)*. Association for Computing Machinery, New York, NY, USA, 358–372. https://doi.org/10.1145/2517349.2517350

[68] Ray Neiheiser, Miguel Matos, and Luís Rodrigues. 2021. Kauri: Scalable BFT Consensus with Pipelined Tree-Based Dissemination and Aggregation. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles* (Virtual Event, Germany) *(SOSP '21)*. Association for Computing Machinery, New York, NY, USA, 35–48. https://doi.org/10.1145/3477132.3483584

[69] Neo4j. [n. d.]. Mitigating Causal Cluster Re-elections. https://tinyurl.com/mtus387s Last accessed: 2025-12-09.

[70] Khiem Ngo, Siddhartha Sen, and Wyatt Lloyd. 2020. Tolerating Slowdowns in Replicated State Machines using Copilots. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association, 583–598. https://www.usenix.org/conference/osdi20/presentation/ngo

[71] Daniel Porto, João Leitão, Cheng Li, Allen Clement, Aniket Kate, Flavio Junqueira, and Rodrigo Rodrigues. 2015. Visigoth fault tolerance. In *Proceedings of the Tenth European Conference on Computer Systems* (Bordeaux, France) *(EuroSys '15)*. Association for Computing Machinery, New York, NY, USA, Article 8, 14 pages. https://doi.org/10.1145/2741948.2741979

[72] HariGovind V. Ramasamy and Christian Cachin. 2006. Parsimonious Asynchronous Byzantine-Fault-Tolerant

Atomic Broadcast. In *Principles of Distributed Systems*, James H. Anderson, Giuseppe Prencipe, and Roger Wattenhofer (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 88–102.

[73] Matthieu Rambaud. 2024. Faster Asynchronous Blockchain Consensus and MVBA. Cryptology ePrint Archive, Paper 2024/1108. https://eprint.iacr.org/2024/1108

[74] Nibesh Shrestha, Rohan Shrothrium, Aniket Kate, and Kartik Nayak. 2025. Sailfish: Towards Improving the Latency of DAG-Based BFT . In *2025 IEEE Symposium on Security and Privacy (SP)*. IEEE Computer Society, Los Alamitos, CA, USA, 1928–1946. https://doi.org/10.1109/SP61157.2025.00021

[75] Alexander Spiegelman, Neil Giridharan, Alberto Sonnino, and Lefteris Kokoris-Kogias. 2022. Bullshark: DAG BFT Protocols Made Practical. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security* (Los Angeles, CA, USA) *(CCS '22)*. Association for Computing Machinery, New York, NY, USA, 2705–2718. https://doi.org/10.1145/3548606.3559361

[76] Alexander Spiegelman, Neil Giridharan, Alberto Sonnino, and Lefteris Kokoris-Kogias. 2022. Bullshark: The Partially Synchronous Version. *arXiv preprint arXiv:2209.05633* (2022).

[77] Alexander Spiegelman, Arik Rinberg, and Dahlia Malkhi. 2021. ACE: Abstract Consensus Encapsulation for Liveness Boosting of State Machine Replication. In *24th International Conference on Principles of Distributed Systems (OPODIS 2020)*. https://drops-dev.dagstuhl.de/entities/document/10.4230/LIPIcs.OPODIS.2020.9

[78] Chrysoula Stathakopoulou, Tudor David, Matej Pavlovic, and Marko Vukolić. 2019. Mir-bft: High-throughput robust bft for decentralized networks. *arXiv preprint arXiv:1906.05552* (2019).

[79] Diem Team. 2021. *DiemBFT v4: State Machine Replication in the Diem Blockchain*. Technical Report. Diem. https://tinyurl.com/pt2e4sfr/2021-08-17.pdf (last accessed on 09/23/2024).

[80] Pasindu Tennage, Cristina Basescu, Lefteris Kokoris-Kogias, Ewa Syta, Philipp Jovanovic, Vero Estrada-Galinanes, and Bryan Ford. 2023. QuePaxa: Escaping the tyranny of timeouts in consensus. In *Proceedings of the 29th Symposium on Operating Systems Principles*. 281–297.

[81] Michael Whittaker, Neil Giridharan, Adriana Szekeres, Joseph Hellerstein, and Ion Stoica. 2021. SoK: A Generalized Multi-Leader State Machine Replication Tutorial. In *Journal of Systems Research - Mar 2021*. https://openreview.net/forum?id=4Xo8nv5DNS

[82] Lei Yang, Seo Jin Park, Mohammad Alizadeh, Sreeram Kannan, and David Tse. 2022. DispersedLedger: High-Throughput Byzantine Consensus on Variable Bandwidth Networks. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*. USENIX Association, Renton, WA, 493–512. https://www.usenix.org/conference/nsdi22/presentation/yang

[83] Maofan Yin, Dahlia Malkhi, Michael K. Reiter, Guy Golan Gueta, and Ittai Abraham. 2019. HotStuff: BFT Consensus with Linearity and Responsiveness. In *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing* (Toronto ON, Canada) *(PODC '19)*. Association for Computing Machinery, New York, NY, USA, 347–356. https://doi.org/10.1145/3293611.3331591

A Retry Protocol

Recovery Input Selection. To determine whether a value may have been committed in prior views, replicas broadcast their local state to others.

1: P → R: Replica *P* broadcasts START-VIEW.

When *P* enters view *v*, it broadcasts its local understanding of whether the previously elected lane committed (it sets $cert\text{-}data = qcs[E]$ or $\langle \perp_{elect} \rangle_P$ if it did not receive an *elected* recovery non-equivocation certificate): if $cert\text{-}data = \langle \perp_{elect} \rangle_P$, no operation committed in view $v-1$ according to *P* (note that, unlike traditional BFT protocols, this says nothing about whether a value committed in a prior view). Otherwise, it forwards $cert\text{-}data = qcs[E]$ to all other replicas as part of a START-VIEW $:= \langle v, cert\text{-}data \rangle$ message.

2: P: *P* receives a quorum of valid START-VIEW messages.

Replica *P* decides what input to recover based on the $n-f$ valid START-VIEW messages it has received. From these, *P* distinguishes between two cases: 1) a value could have been committed in view $v-1$ 2) a value definitely did not commit in view $v-1$, but may have committed in a prior view.

Case 1: *P* receives a recovery non-equivocation certificate in view $v-1$.

Upon receiving a recovery non-equivocation certificate (*recovery-non-equiv-cert*), *P* checks that is well-formed and from the elected lane in view $v-1$. Receiving an recovery non-equivocation certificate signals that a correct replica *may have committed* on the recovery path for view $v-1$. This follows from quorum intersection: if an *elected* recovery confirm certificate exists, *P* is guaranteed to receive at least one corresponding recovery non-equivocation certificate. To preserve safety, *P* must therefore set its recovery input to the *elected* recovery non-equivocation certificate: $r\text{-}input = recovery\text{-}non\text{-}equiv\text{-}cert$.

Case 2: *P* receives $n-f$ $\langle \perp_{elect} \rangle$.

Receiving $n-f$ $\langle \perp_{elect} \rangle$ messages definitively indicates that no correct replica committed on the recovery path in view $v-1$. By quorum intersection, if an *elected* recovery confirm certificate had existed, *P* would necessarily have received an recovery non-equivocation certificate.

However, *P* still cannot safely recover an arbitrary value. Receiving $n-f$ $\langle \perp_{elect} \rangle$ says nothing about what was decided in *prior views*. If a value was previously committed, *P* must recover the value and set it as its recovery input. We distinguish between two cases. A correct replica may have committed a value 1) on the racing path 2) on the recovery path in any earlier view $v' < v-1$.

If a value *x* committed on the racing path, all replicas would have received a sports car certificate in view 0, and thus set this as their recovery input. From then onwards, all recovery confirm certificate exchanged as part of higher views will always be for value *x*. All recovery confirm certificate received as part of view $v-1$ will thus necessarily be for value *x*. *P* can select any as its recovery input.

If a value *x* instead committed in view $v' < v-1$, all replicas would have received a REC-PREPARE for *x* in view $v'+1$, and would have thus set their recovery input to *x*. From then onwards, all recovery confirm certificate exchanged as part of higher views will always be for value *x*. In both cases, *P* can thus set its recovery input to any recovery confirm certificate from $v-1$: $r\text{-}input = rec\text{-}confirm\text{-}cert$, and aggregates the $n-f$ signatures on $\langle \perp_{elect} \rangle$ into a no-elected commit certificate, *nec-cert*.

After selecting its recovery input, *P* enters the race exclusion phase, using the no-elected commit certificate in place of the no-commit certificate (if it exists). From this point forward, *P* executes the persistence and lane-election phases exactly as in view 0, with no further differences.

B Proofs

B.1 Safety

For safety, correct replicas enforce the following validity rules for messages for a given slot.

Protocol validity rules. All rules below are for a fixed consensus instance/slot. Correct replicas only sign or vote for messages that are valid according to the rules below.

1. **Fast-prepare uniqueness.** A correct replica sends at most one SC-PREP-VOTE message.
2. **No-leader-proposal exclusion.** A correct replica never both signs $\langle \perp_{prep} \rangle$ and sends a SC-PREP-VOTE message.
3. **Fast-commit validity.** A correct replica sends SC-COMMIT for digest *d* only after verifying and storing a valid sports car certificate for the same digest *d*.
4. **No-leader-lock exclusion.** A correct replica never both sends SC-COMMIT and signs $\langle \perp_{sc} \rangle$. This exclusion is permanent in both directions.
5. **Slow-prepare uniqueness.** For each proposer *i* in view 0, a correct replica sends at most one TRUCK-PREP-VOTE message.
6. **Recovery-prepare vote uniqueness.** For each proposer *i* and view *v*, a correct replica sends at most one REC-PREP-VOTE message.
7. **Recovery-prepare vote validity.** A correct replica sends a REC-PREP-VOTE for digest *d* only after verifying a valid REC-PREPARE message for the same proposer, view, and digest *d*.
8. **View-0 recovery-prepare validity.** In view 0, a REC-PREPARE message from proposer *i* for digest *d* is valid only if it contains one of the following:

   (a) a valid sports car certificate for digest *d*; or

(b) a valid truck certificate from proposer i for digest d together with a valid no-commit certificate.

In particular, the digest of the REC-PREPARE must match the digest of the supporting certificate.

9. **Later-view recovery-prepare validity.** For view $v>0$, a REC-PREPARE message for digest d is valid only if it adopts one of the following certificates from view $v-1$:
   (a) an *elected* recovery non-equivocation certificate from view $v-1$ with digest d; or
   (b) a non-*elected* recovery confirm certificate from view $v-1$ with digest d together with a valid no-elected commit certificate for view v.

   Thus, the digest proposed in view v must equal the digest of the certificate adopted from view $v-1$.
10. **Recovery-confirm validity.** A correct replica sends a REC-CONFIRM-VOTE for digest d only after verifying a valid REC-CONFIRM message containing a valid recovery non-equivocation certificate for the same proposer, view, and digest d.
11. **Recovery-confirm vote uniqueness.** For each proposer i and view v, a correct replica sends at most one REC-CONFIRM-VOTE message.
12. **Election uniqueness.** In every view v, all correct replicas agree on at most one elected proposer. A certificate is *elected* in view v only if it originates from that proposer.
13. **No-election exclusion.** A correct replica never both sends a REC-CONFIRM-VOTE for an *elected* recovery non-equivocation certificate in view v and signs $\langle\perp_{elect}\rangle f$or view $v+1$. This exclusion is permanent in both directions.
14. **Commit validity.** A correct replica commits on the fast path only after receiving a valid leader commit certificate. A correct replica commits on the recovery path only after receiving a valid *elected* recovery confirm certificate.
15. **Embedded validity.** Whenever a correct replica verifies a message containing an embedded certificate, the embedded certificate must be valid and must match the same consensus instance, proposer where applicable, view, and digest claimed by the outer message.

Lemma 1. *For any two sports car certificate, P and P', $P.dig=P'.dig$*

Proof. Suppose for the sake of contradiction, $P.dig\neq P'.dig$. P and P' consist of $n-f$ SC-PREP-VOTE messages. Two quorums of size $n-f$ intersect in at least one correct replica, which means at least one correct replica sent conflicting SC-PREP-VOTE messages with different digests, a contradiction since correct replicas only send one SC-PREP-VOTE message. □

Lemma 2. *For any two leader commit certificate, C and C', $C.dig=C'.dig$*

Proof. Suppose for the sake of contradiction, $C.dig\neq C'.dig$. C and C' consist of $n-f$ SC-COMMIT messages. Two quorums of size $n-f$ intersect in at least one correct replica, which means at least one correct replica sent conflicting SC-COMMIT messages for different digests. A correct replica only sends a SC-COMMIT message for digest, dig, if it receives a sports car certificate also for dig, meaning it received two conflicting sports car certificate, a contradiction to lemma 1. □

Lemma 3. *For any sports car certificate, L, and any leader commit certificate, C, $L.dig=C.dig$.*

Proof. Suppose for the sake of contradiction, $L.dig\neq C.dig$. By lemma 1 any two sports car certificate must have the same digest, $L.dig$. C consists of $n-f$ SC-COMMIT messages, of which at least $n-2f$ are from correct replicas. This means that $n-2f$ correct replicas sent a SC-COMMIT message for a digest that does not correspond to a valid sports car certificate, a contradiction. □

Lemma 4. *For any two truck certificate, P and P', from proposer i in view 0, $P.dig=P'.dig$*

Proof. Suppose for the sake of contradiction, $P.dig\neq P'.dig$. P and P' consist of $n-f$ TRUCK-PREP-VOTE messages. Two quorums of size $n-f$ intersect in at least one correct replica, which means at least one correct replica sent conflicting TRUCK-PREP-VOTE messages with different digests, a contradiction since correct replicas only send one TRUCK-PREP-VOTE message for a given proposer in view 0. □

Lemma 5. *If a no-lock certificate forms, then no sports car certificate can exist.*

Proof. Suppose for the sake of contradiction there exists both a no-lock certificateand a sports car certificate. This means a quorum Q_1 of size at least $n-f$ signed $\langle\perp_{prep}\rangle$. And a quorum Q_2 of size at least $n-f$ sent a SC-PREP-VOTE message. Q_1 and Q_2 intersect in at least one correct replica, which signed $\langle\perp_{prep}\rangle a$nd sent a SC-PREP-VOTE message. A correct replica will not do both, a contradiction. □

We say that a recovery non-equivocation certificate is either $n-f$ REC-PREP-VOTE messages, or a tuple consisting of $n-f$ TRUCK-PREP-VOTE messages and a no-lock certificate (upgraded).

Lemma 6. *For any two recovery non-equivocation certificate, P and P', in view 0 from proposer i, $P.dig=P'.dig$*

Proof. Suppose for the sake of contradiction, $P.dig\neq P'.dig$. There are three cases: (*i*) P and P' consist of $n-f$ REC-PREP-VOTE messages (*ii*) P and P' consist of $n-f$ TRUCK-PREP-VOTE messages and a no-lock certificate or (*iii*) WLOG P consists of $n-f$ REC-PREP-VOTE messages and P' consists of $n-f$ TRUCK-PREP-VOTE messages and a no-lock certificate.

Case (*i*). P and P' consist of $n-f$ REC-PREP-VOTE messages. Two quorums of size $n-f$ intersect in at least one correct replica, which means at least one correct replica sent conflicting REC-PREP-VOTE messages with different digests, a contradiction since correct replicas only send one REC-PREP-VOTE message for a given proposer in view 0.

Case (*ii*). By lemma 4 any two truck certificate must be for the same digest, a contradiction.

Case (*iii*). By lemma 4, any truck certificate from proposer i must have digest = $P'.dig$, and by lemma 5, there does not exist a sports car certificate. This means that at least $n-2f$ correct replicas sent a REC-PREP-VOTE for a REC-PREPARE that did not contain a supporting truck certificate or a sports car certificate, a contradiction since a valid REC-PREPARE must contain either a truck certificate or a sports car certificate. □

Lemma 7. *For any two recovery confirm certificate, C and C′, in view* 0 *from proposer i, C.dig* $= C'.dig$

Proof. Suppose for the sake of contradiction, $C.dig \neq C'.dig$. C and C' consist of $n-f$ REC-CONFIRM-VOTE messages. Two quorums of size $n-f$ intersect in at least one correct replica, which means at least one correct replica sent conflicting REC-CONFIRM-VOTE messages for different digests. A correct replica only sends a REC-CONFIRM-VOTE message for digest, *dig*, if it receives a recovery non-equivocation certificate also for *dig* (contained in a REC-CONFIRM message), meaning it received two conflicting recovery non-equivocation certificate, a contradiction to lemma 6. □

Lemma 8. *For any recovery non-equivocation certificate, P, and recovery confirm certificate, C in view* 0 *from proposer i, P.dig* $= C.dig$.

Proof. Suppose for the sake of contradiction, $P.dig \neq C.dig$. By lemma 6 any recovery non-equivocation certificate must have digest equal to $P.dig$. This means that $n-2f$ correct replicas sent a REC-CONFIRM-VOTE message for a digest not corresponding to a valid recovery non-equivocation certificate, a contradiction. □

Lemma 9. *For any two recovery non-equivocation certificate P and P′, from proposer i in view* $v > 0$, $P.dig = P'.dig$.

Proof. Suppose for the sake of contradiction, $P.dig \neq P'.dig$. P and P' consist of $n-f$ REC-PREP-VOTE messages. Two quorums of size $n-f$ intersect in at least one correct replica, which means at least one correct replica sent conflicting REC-PREP-VOTE messages with different digests, a contradiction since correct replicas only send one REC-PREP-VOTE message per proposer per view. □

Lemma 10. *For any recovery non-equivocation certificate, P, and recovery confirm certificate, C in view* $v > 0$ *from proposer i, P.dig* $= C.dig$.

Proof. Suppose for the sake of contradiction, $P.dig \neq C.dig$. By lemma 9 any recovery non-equivocation certificate from proposer i in view v must have digest equal to $P.dig$. This means that $n-2f$ correct replicas sent a REC-CONFIRM-VOTE message for a digest that does not correspond to a valid recovery non-equivocation certificate, a contradiction. □

Lemma 11. *If there exists a leader commit certificate, C, then a no-commit certificate cannot exist.*

Proof. Suppose for the sake of contradiction, there exists both a leader commit certificate and a no-commit certificate. The existence of C implies that at least $n-2f$ correct replicas sent a SC-COMMIT message. These correct replicas must have stored a sports car certificate locally (in *sportscar-cert*) before sending a SC-COMMIT message. However, the existence of a no-commit certificate implies that at least $n-f$ replicas signed the message $\langle \perp_{sc} \rangle$. A quorum of $n-2f$ correct replicas intersects any quorum of $n-f$ replicas, in at least one correct replica. This correct replica must have sent a SC-COMMIT message and signed the message $\langle \perp_{sc} \rangle$, a contradiction since correct replicas never both send SC-COMMIT and sign the message $\langle \perp_{sc} \rangle$. □

Lemma 12. *If there exists a leader commit certificate, C, then any recovery non-equivocation certificate, P, from any proposer in view* 0 *must have P.dig* $= C.dig$

Proof. Suppose for the sake of contradiction, $P.dig \neq C.dig$. There are two cases: (*i*) P is a tuple consisting of a truck certificate and a no-lock certificate (upgraded) or (*ii*) P consists of $n-f$ REC-PREP-VOTE messages.

Case (*i*). The existence of C implies that at least $n-2f$ correct replicas sent a SC-COMMIT message. A correct replica only sends a SC-COMMIT message if it received a sports car certificate, thus a sports car certificate exists. By lemma 5, there cannot exist a sports car certificate since P contains a no-lock certificate, a contradiction.

Case (*ii*). By lemma 11 there cannot exist a no-commit certificate and by lemma 3, any sports car certificate must have the same digest as a leader commit certificate. However, the existence of P implies that at least $n-2f$ correct replicas sent a REC-PREP-VOTE message for $P.dig \neq C.dig$. A correct replica will only send a REC-PREP-VOTE message for $P.dig \neq C.dig$ if the REC-PREPARE message contains a no-commit certificate, a contradiction. □

We say an *elected* certificate in view v is a certificate originating from the randomly elected proposer in view v. We say a no-elected commit certificate is $n-f$ $\langle \perp_{elect} \rangle$*m*essages.

Lemma 13. *If there exists an elected recovery confirm certificate in view v, then there cannot exist a no-elected commit certificate in view* $v+1$.

Proof. Suppose for the sake of contradiction there exists both an *elected* recovery confirm certificate in view v and a no-elected commit certificate in view $v+1$. The existence of an *elected* recovery confirm certificate, means that at least $n-2f$ correct replicas sent a REC-CONFIRM-VOTE message for an *elected* recovery non-equivocation certificate. However, the existence of a no-elected commit certificate implies that at least $n-f$ replicas signed $\langle \perp_{elect} \rangle$. $n-2f$ correct replicas intersects a quorum of $n-f$ replicas in at least one correct replica. This means that a correct replica sent both a REC-CONFIRM-VOTE message indicating it received an *elected* recovery non-equivocation certificate in view v, and also signed $\langle \perp_{elect} \rangle$, a contradiction, since correct replicas never do both. □

Lemma 14. *If a correct replica commits dig in view v (recovery path), then any recovery non-equivocation certificate, P, in view $v' > v$ must have $P.dig = dig$*

Proof. We prove the lemma by induction.

Base Case: $v' = v+1$**.** Suppose for the sake of contradiction $P.dig \neq dig$. The existence of a recovery non-equivocation certificate in view $v+1$ implies that at least $n-2f$ correct replicas sent a REC-PREP-VOTE message for a REC-PREPARE message containing either (*i*) an *elected* recovery non-equivocation certificate from view v or (*ii*) a non-*elected* recovery confirm certificate and a no-elected commit certificate for view $v+1$. For case (*i*), by lemmas 10 and 8 any *elected* recovery non-equivocation certificate in view v must have a digest of dig, a contradiction since the recovery non-equivocation certificate must also be for digest dig. For case (*ii*), by lemma 13, a no-elected commit certificate cannot exist, a contradiction.

Induction Step. Assume the lemma holds for all $v'-1$, now consider view v'. Suppose for the sake of contradiction $P.dig \neq dig$. The existence of a recovery non-equivocation certificate in view v' implies that at least $n-2f$ correct replicas sent a REC-PREP-VOTE message for a REC-PREPARE message containing either (*i*) an *elected* recovery non-equivocation certificate from view $v'-1$ or (*ii*) a non-*elected* recovery confirm certificate from view $v'-1$ and a no-elected commit certificate for view v'. For case (*i*), by the induction assumption any recovery non-equivocation certificate in view $v'-1$ must have $P.dig = dig$, a contradiction. For case (*ii*), by the induction assumption and lemma 10 any recovery confirm certificate in view $v'-1$ must have digest $= dig$, a contradiction. □

Lemma 15. *If there exists a leader commit certificate for digest dig, then any recovery non-equivocation certificate, P, in view $v' > 0$ must have $P.dig = dig$.*

Proof. We prove the lemma by induction.

Base Case: $v' = 1$**.** Suppose for the sake of contradiction $P.dig \neq dig$. The existence of a recovery non-equivocation certificate in view 1 implies that at least $n-2f$ correct replicas sent a REC-PREP-VOTE message for a REC-PREPARE message containing either (*i*) an *elected* recovery non-equivocation certificate from view 0 or (*ii*) a non-*elected* recovery confirm certificate from view 0 and a no-elected commit certificate. By lemma 12 any recovery non-equivocation certificate (including *elected*) in view 0 must have digest $= dig$. By lemma 8 any recovery confirm certificate must have digest $= dig$. Thus a correct replica sent a REC-PREP-VOTE for neither (*i*) or (*ii*), a contradiction.

Induction Step. Assume the lemma holds for all $v'-1$, now consider view v'. Suppose for the sake of contradiction $P.dig \neq dig$. The existence of a recovery non-equivocation certificate in view v' implies that at least $n-2f$ correct replicas sent a REC-PREP-VOTE message for a REC-PREPARE message containing either (*i*) an *elected* recovery non-equivocation certificate from view $v'-1$ or (*ii*) a non-*elected* recovery confirm certificate from view $v'-1$ and a no-elected commit certificate for view v'. For case (*i*), by the induction assumption any recovery non-equivocation certificate in view $v'-1$ must have $P.dig = dig$, a contradiction, since the REC-PREPARE must adopt the same digest dig. For case (*ii*), by the induction assumption and lemma 10 any recovery confirm certificate in view $v'-1$ must have digest $= dig$, a contradiction, since the REC-PREPARE must adopt the same digest dig. □

Theorem 1 (Safety)**.** *No two correct replicas commit different values.*

Proof. Suppose for the sake of contradiction there exists two correct replicas, R and R', which commit dig and dig', where $dig \neq dig'$ respectively. There are three cases: (*i*) R and R' both committed from receiving a leader commit certificate, (*ii*) R and R' both committed from receiving an *elected* recovery confirm certificate, and (*iii*) WLOG R committed from receiving a leader commit certificate while R' committed from receiving an *elected* recovery confirm certificate.

For case (*i*), by lemma 2 any leader commit certificate must be for the same value, a contradiction. For case (*ii*), let view v and view v' be the views in which R and R' committed respectively. If $v = v'$ and $v = 0$, then by lemma 7, $dig = dig'$, a contradiction. If $v = v'$ but $v \neq 0$ then by lemmas 9 and 10, $dig = dig'$. Otherwise, WLOG let $v < v'$. By lemma 14, any recovery non-equivocation certificate in view v' must have digest $= dig$, and by lemma 10 any *elected* recovery confirm certificate must have digest $= dig$, a contradiction. For case (*iii*), let view v' be the view in which R' committed. If $v' > 0$ by lemma 15, any recovery non-equivocation certificate certificate in view v' must have digest $= dig$, and by lemma 10 any *elected* recovery confirm certificate in view v' must also have digest $= dig$, a contradiction. Otherwise, if $v' = 0$, the by lemmas 12 and 8 have digest $= dig$, a contradiction. □

B.2 Liveness

The elected lane in each view is produced by a common coin. For any set S of lanes fixed before the election output

is known, $\Pr[elected(v) \in S] = |S|/n$. In particular, if a replica has recovery confirm certificate certificates for $n-f$ distinct lanes before the elected lane is known, then the elected lane is among those lanes with probability $(n-f)/n > 2/3$.

Termination certificates are globally verifiable. If a correct replica terminates after receiving either a leader commit certificate or an elected recovery confirm certificate, then before terminating it forwards that certificate to all replicas. Any correct replica that receives a valid termination certificate terminates, regardless of its current view. We also assume eventual delivery of messages between correct replicas.

Lemma 16. *If no correct replica ever terminates and a correct replica, R is in view v, then all correct replicas will eventually advance to view $v+1$.*

Proof. We prove the lemma by induction on v.

Base Case: $v = 0$**.** Suppose for the sake of contradiction some correct replica never enters view 1. All correct replicas must have entered view 0 and started the race. Since there are at least $n-f$ correct replicas, and correct replicas only propose valid values, eventually all correct replicas will form a *race-cutoff cert*. This will trigger sending a STATUS message, so all correct replicas will eventually receive $n-f$ STATUS messages. After receiving $n-f$ STATUS messages, a correct replica either obtains a no-lock certificate and skips race exclusion, or enters race exclusion and eventually receives at least $n-f$ REC-PREP-VOTE messages. It then proceeds to persistence phase. It will eventually receive $n-f$ REC-CONFIRM-VOTE messages and complete the persistence phase. All correct replicas will eventually complete the persistence phase, so all correct replicas will eventually receive $n-f$ recovery confirm certificate and send an ELECT-LANE message. Thus every correct replica eventually receives at least $n-f \geq 2f+1$ ELECT-LANE messages. Since no correct replica ever terminates, every correct replica advances to view 1, a contradiction.

Induction Step. Assume the lemma holds for all views up to $v-1$, and consider view v. Since R is in view v, it previously entered view $v-1$. By the induction hypothesis applied to view $v-1$, all correct replicas eventually advance to view v. All correct replicas will send a START-VIEW message at the start of view v and eventually receive $n-f$ START-VIEW messages. All correct replicas will then eventually complete the race exclusion and persistence phases in view v since there are at least $n-f$ correct replicas. All correct replicas will thus receive $n-f$ recovery confirm certificate and send an ELECT-LANE message. Thus every correct replica eventually receives at least $n-f \geq 2f+1$ ELECT-LANE messages. Since no correct replica ever terminates, every correct replica advances to view $v+1$. □

Lemma 17. *If no correct replica terminates before view v and some correct replica enters view v, then some correct replica terminates in view v with probability at least $(n-f)/n > 2/3$.*

Proof. If some correct replica terminates in view v before the election step, then the claim already holds. Otherwise, no correct replica terminates before the election step of view v.

By the same progress argument as in Lemma 16, unless some correct replica terminates earlier in view v, correct replicas eventually complete the required phases of view v. Hence, some correct replica R eventually receives $n-f$ recovery confirm certificate certificates for $n-f$ distinct lanes before the elected lane is known. Let S be this set of lanes.

By the common-coin assumption, $\Pr[elected(v) \in S] = (n-f)/n > 2/3$. If $elected(v) \in S$, then R has a valid recovery confirm certificate for the elected lane, so R terminates in view v. Therefore, some correct replica terminates in view v with probability at least $(n-f)/n > 2/3$. □

Lemma 18. *If a correct replica, R, terminates in view v, then all correct replicas eventually terminate.*

Proof. Suppose for contradiction, a correct replica R' never terminates. If R received a leader commit certificate it forwards it to all correct replicas before terminating. R' will thus receive it and also terminate, a contradiction. If R received an *elected* recovery confirm certificate in view v, it will also forward it to all correct replicas before terminating. R' will thus receive it and also terminate, a contradiction. □

Theorem 2 (Liveness)**.** *Ambulance terminates with probability* 1.

Proof. By Lemma 16, if no correct replica terminates, then correct replicas keep entering higher views. Let $p = (n-f)/n > 2/3$. By Lemma 17, conditioned on no correct replica terminating before a view, if no correct replica has terminated before that view, some correct replica terminates in that view with probability at least p. Therefore, by a geometric tail bound, the probability that no correct replica terminates in the first k views is at most $(1-p)^k$, which converges to 0 as $k \to \infty$. Hence some correct replica terminates with probability 1. By Lemma 18, once one correct replica terminates, all correct replicas eventually terminate. Therefore, Ambulance terminates with probability 1. □

C Additional Technical Discussion

By default, the race gives the leader a one-message-delay advantage over the other replicas: the leader can commit in two message delays, whereas non-leader replicas require three. In some deployments, such as LAN environments, this advantage may be insufficient because communication delays between replicas are extremely small. As a result, a non-slow leader may be mistakenly classified as slow, causing replicas to trigger recovery more often than necessary.

To increase the leader's advantage, Ambulance can insert a configurable number of dummy phases before non-leader replicas enter the race. In each dummy phase, replicas send

a message containing the phase number and wait for $n - f$ matching messages before advancing to the next phase. Alternatively, Ambulance can introduce a time-based delay, similar to a hedging delay, to further bias the race in favor of the leader. Although dummy phases or time-based delays may may seem at odds with our productivity requirement, replicas still use this time to generate non-equivocation certificates for recovery. By contrast, traditional timeouts and hedging delays do not perform useful protocol work while waiting.

Dummy phases are particularly useful when motorizing Ambulance. Since coverage (new data layer proposals) to start a slot may be achieved at different times across replicas, the leader may begin a slot later than other replicas, and therefore be detected as slow even when it is not. To mitigate this, replicas can forward their proposals to the leader, allowing the leader to satisfy coverage as well. Because this forwarding step takes one message delay, Ambulance can add one corresponding dummy phase to compensate for the added delay.

SDCs in the Wild: Characterizing and Diagnosing SDC-defective GPUs in Production LLM Training (Operational Systems)

Wenxin Zheng[1,2,*], Wenxiao Wang[2], Yun Zhang[2], Mingcong Han[2], Bin Xu[1], Jinyu Gu[1,✉], Xingda Wei[1], Haibo Chen[1], Zuquan Song[2], Gaohong Liu[2], Yucheng Nie[2], Zhe Nan[2], Zhuolin Zheng[2], Huan Yu[2], Shuguang Wang[2], Ziming Zhou[2], Hang Zhu[2], Wencong Xiao[2,✉], Xin Liu[2]

[1]*Shanghai Jiao Tong University* [2]*ByteDance Seed*

Abstract

Silent Data Corruption (SDC) has emerged as a critical reliability bottleneck in Large Language Model (LLM) training, where hardware faults are frequently indistinguishable from software anomalies. While standard industry practice relies on synthetic microbenchmarks for fault isolation, our experience shows these methods miss over 60% of defective devices. To understand this gap, we present a comprehensive characterization of 23 SDC-defective GPUs harvested from a large-scale production cluster. Our analysis reveals three key insights: (1) SDCs are not confined to new hardware but often arise later due to aging; (2) SDCs are highly data-dependent and unit-specific, meaning devices that pass general stress tests often fail under specific training input data; and (3) standard ECC and thermal protections fail to capture these logic-level bit flips. Driven by these findings, we propose SDCHUNTER, an automated diagnosis system for detecting SDC-defective GPUs in a large-scale training cluster. Instead of relying on generic benchmarks, SDCHUNTER employs execution replay with the exact training workload and input data that triggered the failure. Deployed at ByteDance, SDCHUNTER successfully mitigated 40 SDC incidents in production.

1 Introduction

Silent Data Corruption (SDC), the unintended alteration of data without error signals, is a persistent reliability challenge in large-scale computing [17, 20, 27, 75]. Although SDC events are statistically rare on one device, the massive scale of training Large Language Model (LLM) amplifies their impact, making them a visible and frequent disruption. With training clusters scaling to tens of thousands of GPUs running for months, low-probability hardware faults become inevitable. Recent reports highlight the frequency of hardware faults in large-scale training environments. Meta reports that during training runs on 16,000 GPUs, hardware failures occur approximately every 2.78 hours, with roughly 1.4% of these attributed to GPU SDC [1]. Similarly, Google observes SDC events every one to two weeks within their TPU clusters [3], while ByteDance records 6,096 implicit errors over a three-month operational period in their GPU clusters [73].

However, the most agonizing challenge for training engineers is not the frequency of these errors, but their *ambiguity*. GPU SDCs typically manifest as computational anomalies such as an unexpected fail-stop error or sudden loss spikes that appear identical to software bugs or numerical instabilities. This creates a costly dilemma: engineers often waste weeks debugging phantom code issues, unaware that the root cause is a silent hardware defect. Prior research has primarily focused on detecting and mitigating CPU-side SDCs [16, 18, 29, 40, 44]. On GPUs, numerous studies have investigated SDC detection and fault tolerance in model inference workloads [35, 36, 42, 82]. Other lines of work propose more reliable algorithms, such as self-check GEMM variants [47, 48], or techniques that monitor specific numerical ranges during training to detect SDCs [21, 26]. However, these approaches are not effective for large-scale LLM training, as the increased numerical robustness of LLM models and the significantly more complex software stack make SDCs harder to detect and localize.

In our production environment at ByteDance, we find that most SDC incidents stem from **SDC-defective GPUs**. These devices contain subtle hardware faults that intermittently corrupt computation. Although we occasionally observe SDCs originating from CPUs, they can be diagnosed with the same approach. In these cases, both CPU and GPU memory are protected by ECC and are not the source of SDCs. Unlike transient soft errors, SDC-defective GPUs pose a persistent risk because the same device can repeatedly corrupt training jobs. When engineers observe suspicious behavior during LLM training, such as unexpected errors, the key challenge is to identify which GPU or GPUs in the cluster caused the corruption. The common industry practice is to run hardware diagnostic tools and synthetic stress tests, such as intensive GEMM loops across all GPUs [52, 54, 77]. However, in our experience, these synthetic tests miss more than 60% of defective GPUs, leaving a substantial detection gap.

*Work done while interning at ByteDance.

Insights from Real-World GPU SDCs. To understand why standard benchmarks fail, we experimentally analyzed 23 SDC-defective GPUs harvested from our production fleet. Our characterization reveals three critical insights. First, SDCs are not limited to new hardware. They often appear later due to aging, requiring continuous lifecycle monitoring. Second, SDCs depend heavily on the specific hardware units (e.g., complex math units fail more often than Tensor Cores) and data inputs. A GPU might pass a general test but fail on specific training tasks. Third, these errors are missed by standard protections like ECC or thermal sensors. Consequently, effective diagnosis requires execution replay with the exact production model, checkpoint, workload, and data distribution that triggered the failure. Despite its promise, production replay faces two fundamental challenges.

Challenge 1: Training Replay Determinism. To identify a defective GPU, we compare its output against a "gold standard" from healthy hardware. Because no single GPU can be assumed reliable, this comparison must span replicas and requires identical inputs to produce bit-wise identical outputs. Yet LLM training is rife with non-determinism [25, 79]. System-level noise, such as non-deterministic reduction orders in AllReduce [62] or random operators, prevents direct cross-replica comparison and makes hardware SDCs indistinguishable from normal stochasticity. Our production measurements show that deterministic training has no measurable throughput loss (within 0.01% step-time difference) and reduces normalized debug time by 70% through bit-wise replay.

Challenge 2: Diagnosis Efficiency. Localizing an SDC requires capturing the corruption before it is masked by nonlinear operators or normalization layers [43]. Theoretically, this requires monitoring every intermediate tensor, but global full-state inspection across thousands of GPUs is prohibitively slow. Because intermittent SDCs may take hours to reproduce, such instrumentation would extend downtime from hours to days and delay the critical training job. We therefore need to isolate the fault quickly without paying the cost of full-state monitoring everywhere.

To address these challenges, we propose SDCHUNTER, an automated diagnosis system that uses hierarchical replay to balance speed and precision. SDCHUNTER first screens the cluster to isolate a small faulty GPU group, then investigates that group to pinpoint the defective device.

Phase 1: Lightweight Grouping via Homogeneous Replay. Upon detecting an anomaly, SDCHUNTER partitions the cluster along the Data Parallel (DP)[1] dimension into two logical replicas that consume the identical input batch. In this controlled "dual-replica" setting, outputs should be bit-wise identical. SDCHUNTER verifies this by hashing tensors only at the Pipeline Parallel (PP)[2] boundaries, incurring only 3% runtime overhead and a few extra training steps. Comparing these signatures across DP groups identifies the divergent stage within hours, which is enough to remove the affected replica and resume training while device-level localization continues offline.

Phase 2: Precise Localization via Full-State Comparison. Once the anomaly is confined to a parallel group[3], SDCHUNTER deterministically replays the problematic iteration on the *suspicious* group and a *healthy* reference group. It collects layer-wise signatures for all intermediate tensors, finds the first divergent tensor and kernel through differential analysis, and maps the error back to the defective GPU.

After the training job resumes on healthy machines, SDCHUNTER confirms the localized device offline by repeatedly replaying the alert-triggering trace and checking the result with hardware tools. This verification does not block the online job and provides a hardware-actionable decision within one hour in our production cases.

Both phases rely on system-wide determinism. SDCHUNTER eliminates noise by locking RNG seeds, enforcing deterministic GPU kernels, and standardizing communication order, so discrepancies across DP groups (Phase 1) or against the reference (Phase 2) indicate hardware defects.

We deployed SDCHUNTER in ByteDance's production environment managing a large GPU cluster. It has mitigated 40 SDC incidents to date. Most importantly, SDCHUNTER decouples cluster recovery from hardware debugging: training resumes within an hour after isolating the faulty stage, while the defective GPU is later pinpointed and confirmed offline within one hour for hardware repair.

Our contributions can be summarized as follows:

- We share our first-hand experience managing SDC incidents in large-scale production clusters at ByteDance, showcasing the problem of SDCs in real-world systems.
- We present a characterization study of 23 SDC-defective GPUs, revealing key insights about their manifestation.
- We quantitatively identify the critical challenges in reproducing SDCs in LLM training.
- We propose SDCHUNTER, an automated diagnosis system for localizing SDC-defective GPUs and have identified 40 SDC incidents in production, reducing diagnosis time from days to under one hour.

To the best of our knowledge, this is the first large-scale study of GPU SDCs in production-level LLM training workloads.

[1]The data-parallel dimension is the axis of the training layout where workers run replicated model copies on different input shards. A DP group denotes one set of workers that participate in this replicated computation and synchronize updates.

[2]Pipeline parallelism splits model layers into sequential stages placed on different workers. PP boundaries are the interfaces between adjacent stages where activation or gradient tensors are exchanged.

[3]A parallel group is a subset of distributed workers that cooperate along one parallelism axis, such as data, pipeline, tensor, or expert parallelism.

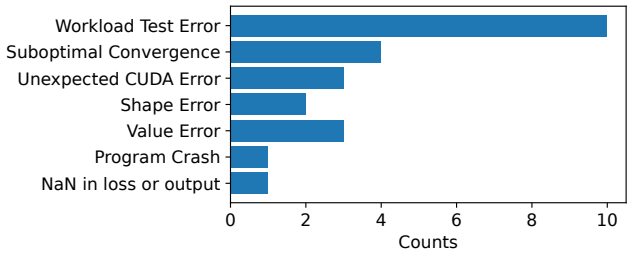


Figure 1: First detected errors of all confirmed SDC errors.

```
1 indices = torch.tensor([0, 1, 0, 1])
2 plan = torch.bincount(ideal_indices, minlength=2)
3 plan = plan.tolist()
4
5 mask = torch.zeros(4, 2, dtype=torch.int)
6 mask.scatter_(1, indices.view(-1, 1), 1) # wrong!
7
8 # Shape mismatch with input and mask
9 experts_input = torch.split(routed_data, plan)
```

Figure 2: Code snippet of a shape mismatch error caused by SDC during MoE training.

```
1 ids %= self.vocab_size # wrong!
2 ids += self.vocab_offset
3 ids = ids[:, -s:]
4
5 # Out-of-bound caused by ids
6 embeds = self.embedding_list[0](ids)
```

Figure 3: Code snippet of an out-of-bound access caused by SDC during MoE training.

2 GPU Silent Data Corruption in Production

2.1 Silent Data Corruption

The correctness of large model training is very important. In practice, training can yield incorrect data in three characteristic ways: crash [73], detected unrecoverable error (DUE) [6, 13, 14, 18, 24, 49], and silent data corruption (SDC) [24, 26, 43, 44]. The crash indicates that the program terminates abnormally due to an unrecoverable error, such as segmentation faults or illegal instructions. DUE refers to errors that are detected by the hardware or software, leading to the termination of the program to prevent further propagation of errors. Both crash and DUE provide immediate feedback to developers, allowing them to identify and address issues promptly.

Different from the crash and DUE, SDC does not produce any general immediate indication of an error. It refers to errors that occur silently without any immediate indication, leading to incorrect results or data corruption. SDC can manifest in various ways, such as bit flips in memory, or incorrect computations. The typical reasons for SDC include environmental factors, hardware bugs, and hardware degradation. Environmental factors such as cosmic rays and radiation can induce transient faults in hardware components [56, 74, 78, 82]. Hardware bugs, such as design flaws or manufacturing defects, can also lead to SDC [44]. Over time, hardware components can degrade due to circuit wear-out and aging [44]. All these factors can contribute to the occurrence of SDC, making it a significant concern in large-scale computing systems.

2.2 SDC-Induced Failures in LLM Training

In this paper, we analyze 23 GPUs with confirmed SDCs identified in our production clusters. Figure 1 classifies the detection sources of these defective devices. Specifically, 10 GPUs were identified during offline stress test using online workloads where they failed numerical validation checks. The remaining 13 GPUs surfaced during active LLM training tasks, causing task interruptions. These production failures manifest in two primary ways: (1) fail-stop errors, where the SDC triggers a system exception (e.g., shape mismatch or out-of-bound access) causing an immediate crash; and (2) implicit anomalies, where the training continues but exhibits incorrect behavior (e.g., loss spikes), leading engineers to manually halt the job.

Confronted with these interruptions, developers instinctively prioritize software debugging over hardware verification. Given the rarity of hardware faults, engineers typically attribute these anomalies to logical bugs in the training software framework or numerical instabilities inherent to the model architecture [30]. Consequently, engineers often invest days or weeks isolating pipeline components and attempting to reproduce the error, but these efforts are frequently wasted. The identification of a hardware fault usually emerges only as a diagnosis of exclusion after all software hypotheses have been exhausted. This characterizes the insidious nature of SDCs: while their probability of occurrence is low, the incurred cost in engineering productivity and wasted computing resources is disproportionately high. We share two detailed cases derived from our real-world production experience to illustrate these deceptive patterns of SDC-induced failures.

Case 1: Explicit Fail-stop Error (Shape Mismatch or Out-of-bound Access). Megatron [69] decouples capacity planning from tensor dispatching in its implementation of MoE operators to maximize device utilization. Under this pipeline, the framework first aggregates token counts to pre-allocate contiguous memory buffers, and subsequently invokes a scatter kernel to populate these buffers. We observed a case in our production environment where an SDC occurred during the token count aggregation, silently corrupting the buffer size calculation. As shown in Figure 2, the error comes from the corrupted `scatter` operation (line 6) and eventually triggers a shape mismatch error at line 9. As shown in Figure 3, the out-of-bound access is caused by the corrupted modulo operation (line 1) and eventually triggers a segmentation fault at line 6. This manifested error creates a debugging trap: the exception did not trigger immediately, but it indicates a fail-stop error, causing developers to suspect a bug in the code. However, the

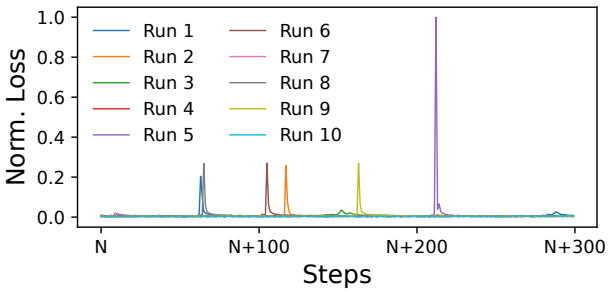


Figure 4: Loss spike caused by SDC during model training with different runs. *The loss spike is confirmed to be caused by SDC, but the spike is not always the same.*

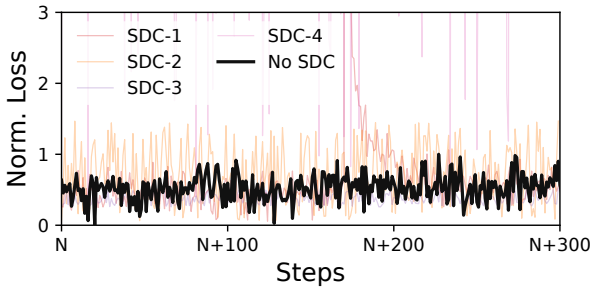


Figure 5: Loss curves of training runs with and without SDC collected from the same checkpoint in production. The spike problem observed in production is shown in Figure 4.

code at the crash site is correct. The SDC had already altered the program state in a previous kernel, rendering the error message a misleading indicator of the true failure source.

Case 2: Implicit Performance Degradation (Anomalous Loss Spikes). In contrast to fail-stop errors, SDCs can manifest as implicit model degradation where the training process continues but produces corrupted checkpoints. The loss comparison in Figure 5 shows the practical consequence: without SDC, the training loss remains within the expected narrow range, while SDC-affected executions create visible excursions that can invalidate subsequent checkpoints. The spike problem is shown in Figure 4. We encountered this scenario in 10 independent training runs executed on the same cluster and initiated from the identical checkpoint. The anomaly first surfaced in Run 1, where the loss spiked dramatically. A distinct re-run, Run 2, reproduced the failure, exhibiting a similar spike. However, when engineers attempted Runs 3 and 4, the training proceeded without incident, showing no anomalies. This led engineers to suspect the occurrence of SDCs. Continued monitoring validated this hypothesis: Runs 5 and 6 (and subsequently Runs 8 and 9) exhibited loss spikes again, but crucially, these occurred at completely different steps compared to the earlier failures. Unlike a code error that would typically fail at a consistent logical point, these random, high-magnitude gradient values indicated underlying hardware malfunctions in the floating-point units. Consequently, the training was halted, and the affected nodes were taken offline to confirm the existence of a defective GPU.

2.3 SDC Impact in LLM Training

Both failure modes impose substantial operational cost in production-level LLM training. Fail-stop errors (Case 1) immediately interrupt the job, leaving allocated GPUs idle until the training state is restored from a valid checkpoint and the defective hardware is isolated. Implicit anomalies (Case 2) are more difficult to handle because the training job may continue to produce invalid checkpoints. Engineers must then determine whether the observed loss spike originates from software bugs, numerical instability, or hardware corruption since the loss spike can degrade the model performance [5, 37, 60, 71, 72]. Without deterministic execution, rerunning the same checkpoint cannot guarantee bit-wise alignment, which further obscures the boundary between benign training variance and SDC-induced model corruption.

The difficulty of mitigation is amplified by the large variation in how frequently a defective GPU manifests an error. We define the *observed SDC occurrence rate* as the fraction of deterministic replays of an alert-triggering workload that produce an incorrect tensor. Across the confirmed defective GPUs, this replay occurrence rate ranges from 100% down to 10^{-6}%. This variation explains why generic hardware stress tests frequently miss SDC-defective GPUs and why diagnosis must replay the production workload that exposed the anomaly. To quantify the practical impact, we measure the optimistic estimation of retraining overhead and the debugging and diagnosis time observed in our production workflow in Table 1. **Actually, the overhead is a lot higher than the estimation in our production workflow.**

Table 1: Overhead over a full training run across varying SDC occurrence rates. *The **SDC Rate** means the visible difference in the loss curve. This trace is collected from manual oncall services with round-the-clock coverage.*

SDC Rate	Expected Steps	Polluted Ckpts	Overhead	
			Retrain	Debug
100%	~ 1	≤ 1	0.011%	0.029%
10%	~ 10	≤ 1	0.020%	0.29%
1%	~ 100	~ 1	0.110%	2.90%
0.1%	$\sim 1,000$	≥ 1	1.510%	29.0%
0.01%	$\sim 10,000$	≥ 10	1.510%	2.9×

Utilizing the industry-standard software diagnostic method [73], the required validation compute scales inversely with p. At $p = 0.01\%$, diagnosing a single defective GPU consumes parallel compute equivalent to training approxi-

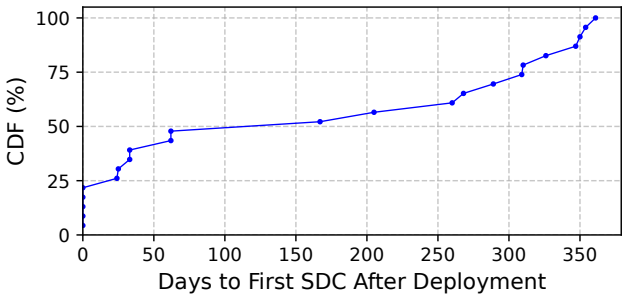


Figure 6: Temporal distribution of SDC emergence across hardware lifecycle.

mately three full LLM models (≈ 290%). This mathematically negates the viability of purely software-replay-based debugging for low-frequency SDCs in hyperscale clusters. This motivates a system to diagnose SDC-defective GPUs at scale.

3 Characterizing SDC-defective GPUs

Silent data corruption originates from hardware degradation that manifests in ways fundamentally different from hard failures. To characterize these differences, we first analyze hardware fault behaviors across four dimensions: temporal patterns, observability, triggering conditions, and visibility.

Temporal Patterns. Unlike prior studies that primarily observe CPU SDCs during early stress testing [75], our long-term analysis reveals that GPU SDCs pose a persistent threat throughout the entire hardware lifecycle. By tracking the timelines of all confirmed SDC cases in our production environment, we find that these faults are not limited to the early time of GPUs. Instead, they manifest continuously due to cumulative hardware degradation. Specifically, only 25% of SDCs were detected during pre-deployment burn-in, whereas the remaining incidents surfaced gradually: 25% within the first two months, 10% within six months, and a substantial 40% around one year post-deployment (see Figure 6). Furthermore, once a GPU begins to exhibit SDC, the error rate typically escalates over time, signaling progressive deterioration that can lead to permanent failure. This temporal dispersion indicates cumulative degradation rather than manufacturing defects. Vendor analysis suggests the underlying mechanism involves resistance drift in computational circuits, specifically, repeated intensive computation causes electrical stress that gradually alters resistance values, leading to arithmetic errors. This finding is applicable to all long-running GPU fleets that execute sustained high-utilization training workloads, while the exact timeline depends on GPU generation, cooling conditions, workload intensity, and maintenance policy.

> The emergence of SDC covers the entire life cycle of hardware and is caused by GPU hardware degradation, which is irreversible once it occurs.

Table 2: Distribution of kernels where the first SDC was detected on SDC-defective GPUs. Some SDCs are only triggered by specific kernels.

Kernel	Proportion
GEMM	17.3%
InplaceMoE	17.3%
FlashAttention	13.0%
RMSNorm	8.69%
InplaceLinear	8.69%
Scatter	8.69%
FusedKernel	8.69%
Mean	8.69%
InternalImplemented	8.69%

Table 3: Characterization of SDC sources in μ-archs for GEMM kernels. ✓ *indicates correct computation, while* ✗ *indicates erroneous computation.*

GPU	CUDA Core		Tensor Core			
	FP32	FP64	BF16	FP16	TF32	INT8
1	✗	✓	✓	✓	✓	✓
2	✗	✗	✓	✓	✗	✗
3	✓	✗	✓	✓	✓	✓
4	✗	✗	✓	✓	✗	✗

SDC Sources in μ-archs. Unlike memory errors that can be detected via ECC or execution exceptions triggered by instruction faults, SDCs come from diverse hardware components [44]. We conducted in-depth experiments to find the sources of SDC by replaying the same workload and kernels on victim GPUs and monitoring error rates. Our analysis reveals that SDC comes from diverse hardware components. Some of them may have specific component failures, while others may have unclear failures. We've listed the first-detected SDC kernels and GPUs in Table 2. For those with GEMM kernel SDCs, we conduct a further analysis of different data types and precision in Table 3. The results indicate that FP64 and FP32 CUDA cores exhibit a higher propensity for SDC. A plausible explanation is that high-precision ALUs are substantially larger and more complex than their lower-precision counterparts: they require wider datapaths and more transistors to implement the corresponding logic. The increased physical area correspondingly elevates the likelihood of fault manifestation [8, 65]. Our findings further suggest that certain SDCs are strongly tied to specific microarchitectural features and data types. For other cases, we posit that SDCs arise from intricate and insufficiently understood interactions within the microarchitecture. This finding is applicable to all GPUs because arithmetic units, memory paths, and fused execution units can all become fault sources, while the exact vulnerable unit and precision mode are implementation dependent.

SDCs originate from multiple GPU components, with susceptibility patterns highly specific to microarchitecture implementations and precision modes.

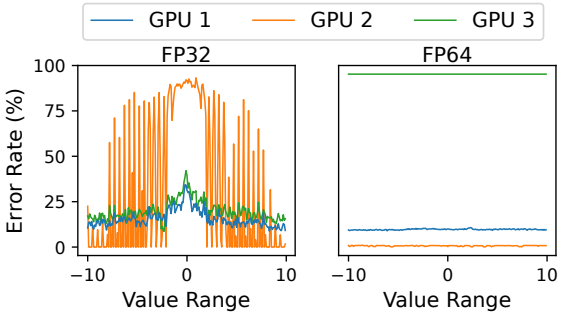


Figure 7: Error rate for a single operator (GEMM) across different input ranges and data types on SDC-affected GPUs.

Triggering Conditions. Triggering SDC is not straightforward. As listed in Table 2, some SDCs become visible only when specific kernels are executed. For these complex cases, errors often manifest as element-wise differences rather than simple numerical errors.

To investigate the triggering patterns of SDC, we executed repeated GEMM kernels on three distinct SDC-defective GPUs, denoted as GPU 1 through GPU 3, varying the input numerical ranges to observe stability boundaries. We define an error as any bit-wise deviation from the standard golden result. Unlike transient soft errors which are stochastic, the errors observed in these devices exhibit strong correlations with input patterns. As shown in Figure 7, the error rate is highly sensitive to both data type and input magnitude. Specifically, GPU 1 and GPU 3 yield incorrect results across all ranges for FP32, whereas GPU 2 exhibits errors only within specific value bands. As for FP64, GPU 3 produces 100% incorrect results with a calculation latency 50× higher than nominal, indicating severe hardware degradation, while GPU 2 maintains an error rate below 1%.

Overall, these findings highlight that SDCs are extremely hard to trigger because they depend on specific operators, data types, and values. Furthermore, even under identical workloads, the error manifests with only a relatively low probability. This finding is applicable to other LLM training workloads that rely on complex fused kernels and data-dependent operators, although the concrete triggering kernels and value ranges must be identified for each model and hardware platform.

SDC occurrence is highly dependent on specific workloads (e.g., kernels, data types and input values), exhibiting strong operator and workload affinity.

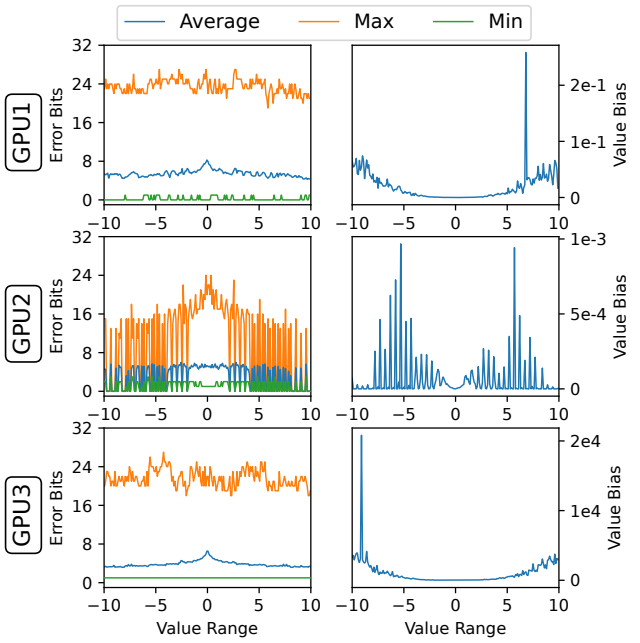


Figure 8: Number of error bits and average value bias for a single operator (GEMM, FP32) across different input ranges on SDC-affected GPUs.

SDC Visibility. We further dissect how SDCs manifest at the numerical level by analyzing the bit-level discrepancies and value biases across different devices. As illustrated in Figure 8, the manifestation of SDCs varies significantly across varying devices. While some defects cause massive deviations, others manifest as sparse bit flips affecting only specific significance levels. For instance, GPU 2 exhibits extremely subtle value biases (magnitude $< 10^{-3}$). Such minor errors are particularly insidious as they are prone to being masked or absorbed by subsequent computational operations (e.g., accumulations or activation functions), propagating silently without triggering obvious outliers.

More importantly, apart from these numerical discrepancies, the hardware faults remain invisible to standard system monitoring metrics. We compared the execution traces between golden hardware and SDC-defective devices. Despite the observable computational errors, standard indicators such as machine check exceptions, performance counters, and device status registers remained indistinguishable from the healthy baseline. Furthermore, the ECC error count shows no correlation with the SDC occurrences, and there are no observable anomalies in clock frequencies or thermal limits between the golden hardware and the defective GPUs. This lack of hardware-level observability implies that error detection must rely solely on verifying the output correctness (e.g., double-checking), as no side-channel signals are available to flag the degradation. This finding is expected to apply to logic-level computational corruptions across all GPUs, while the numerical magnitude and masking behavior depend on

the model architecture and downstream operators.

> SDCs often manifest as subtle value biases without triggering hardware signals, making them difficult to detect through traditional monitoring mechanisms.

4 Challenges of Diagnosing GPU SDCs

Given that SDCs are highly context-sensitive, our characterization reveals that synthetic benchmarks fail because they lack the specific data patterns and instruction sequences required to trigger the faults. This limitation compels us to shift our diagnosis strategy from *simulation* to *reproduction*. Instead of relying on pre-defined test vectors, we adopt an execution replay approach: restoring the training state from a recent checkpoint and re-executing the exact iterations that led to the failure (or loss spikes). By subjecting the suspected GPU to the real production workload, we maximize the probability of recreating the precise micro-architectural state that triggers the silent corruption.

The core principle of this approach is differential testing based on the premise of hardware determinism. Ideally, for a deterministic computational kernel, identical inputs must yield bit-wise identical outputs regardless of the device executing them. Therefore, to confirm a GPU is defective, we can compare its replay output against a golden reference generated by a healthy peer GPU running the same workload. Any discrepancy in the output can be definitively attributed to a hardware fault.

While conceptually straightforward, implementing this replay-based diagnosis in a production LLM training stack is non-trivial. The complexity of distributed training frameworks introduces two fundamental challenges that prevent straightforward comparison and localization: replay determinism and corruption observability. In this section, we quantitatively demonstrate the severity of these two challenges.

4.1 Training Replay Determinism

The prerequisite for differential testing is strict bit-wise reproducibility. Any deviation from the golden reference must stem uniquely from hardware faults. However, achieving such determinism in a distributed training cluster is inherently difficult. The complex interaction between hardware parallelism and software scheduling introduces significant execution noise, which can easily drown out the subtle signals of an SDC. In practice, we identify three primary sources of non-determinism.

Operator Randomness. GPU kernel implementations of some operators often exhibit inherent randomness during execution (e.g., scatter add). Consequently, without explicitly enabling deterministic algorithms, the computation results are inconsistent across runs, as shown in Figure 9. This inconsistency comes from nondeterministic execution order rather than floating-point instability, since a fixed deterministic order produces bit-wise identical results for the same inputs.

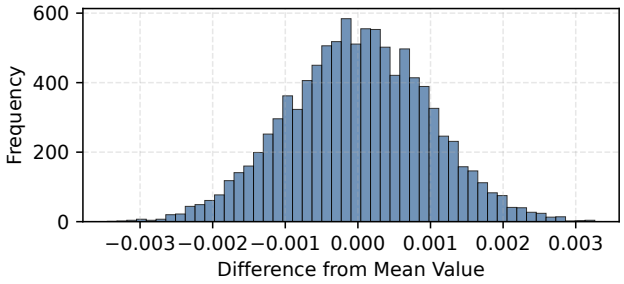


Figure 9: Distribution of output values for the scatter add operator, illustrating the non-determinism on GPU.

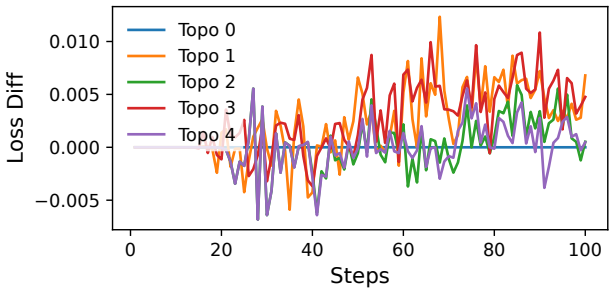


Figure 10: Differences in loss values of different network topologies (Topo 0 to Topo 4) relative to Topo 0.

Network Topology Influence. In distributed training, collective communication operations (e.g., AllReduce) are essential for gradient synchronization. Due to the non-associativity of floating-point arithmetic, ensuring bit-wise reproducibility requires a strictly fixed order of accumulation during reduction. However, the effective network topology in a production cluster is often volatile. NIC failures or bandwidth fluctuations can change the topology observed by communication libraries such as NCCL, causing them to adjust channel counts or algorithms. These changes alter the reduction structure and floating-point addition order, producing numerical divergence even under the same training configuration. To illustrate this effect, we execute the same training configuration across multiple network topologies and collect loss-level metrics. The results, presented in Figure 10, show that the loss variations introduced solely by differing network topologies can be non-trivial. This complicates the diagnosis of SDCs, as it becomes difficult to distinguish whether observed discrepancies stem from actual data corruption or from inherent software-level nondeterminism induced by network topology differences.

Scaling Non-equivalence. GPU resources are extremely precious. When a suspected SDC is detected in a training job involving thousands or even tens of thousands of GPUs, performing a direct replay on a cluster of the same scale is impractical. Doing so would consume a massive amount of

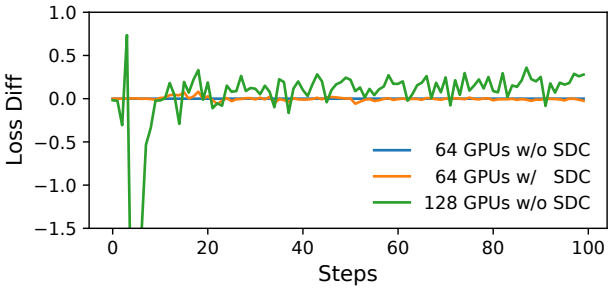


Figure 11: Differences in loss values relative to the setup of 64 GPUs without SDC when scaling down training configurations.

GPU hours and incur huge costs; furthermore, it monopolizes critical resources, blocking other training jobs from being scheduled. Consequently, it is necessary to scale the workload down to a smaller cluster to verify whether a specific GPU is faulty. However, the training result is sensitive to the scale. As stated in previous work [41], keeping consistency between large-scale and small-scale training configurations is challenging. The training framework will select different parallel strategies and kernel fusion strategies for different scales. The training result is different for different scales. To demonstrate this, we use different scales to run the same training configuration on both normal GPUs and SDC GPUs and collect the loss-level training metrics. The results are shown in Figure 11. We observed that the loss difference introduced by scaling down the training configuration is even larger than the deviation caused by a faulty GPU. Consequently, when observing a loss discrepancy in a scaled-down environment, it becomes impossible to distinguish whether the divergence originates from the SDC or the inherent software behavior changes due to scaling.

4.2 Data Corruption Observability

Even if a deterministic replay environment is established, capturing the manifestation of an SDC remains a significant hurdle due to the lack of appropriate observability. We face a fundamental dilemma between *sensitivity* and *efficiency*. High-level metrics (e.g., training loss) are non-intrusive but inherently mask errors due to the robustness of deep learning algorithms. Conversely, low-level inspection (e.g., instruction instrumentation) offers high sensitivity but extremely high overhead, making the SDC reproduction time impractical. We experimentally explore this trade-off in this section.

Loss Observability. Large-model training can mask SDCs in loss-level metrics. To illustrate this, we randomly perturb 10% of tensor values by 1% and 10% during training and compare the resulting loss and gradient-norm differences against an unmodified run as shown in Figure 12. The loss difference is initially small (absolute value below 0.1 and relative value below 5%) and becomes increasingly obscured as training proceeds.

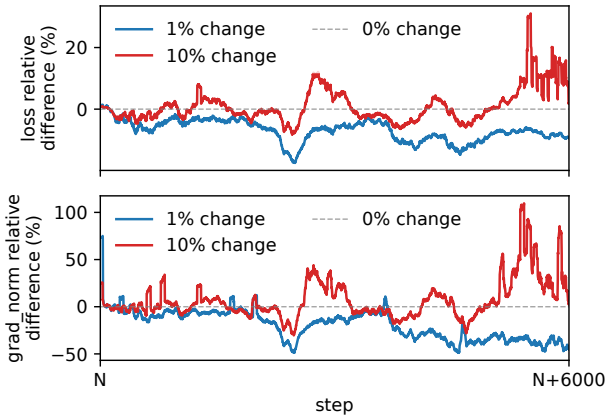


Figure 12: Difference in loss and gradient norm values for changing the value of 10% of tensors by 1% and 10% relative to no value change.

Even if the loss value is different, the first tensor value error appears significantly earlier than the loss value error. To show this lagging effect, we instrumented all tensors on the SDC GPUs and collected the loss-level metrics and tensor-level values. The result is shown in Figure 13. We found that the first tensor value error appears significantly earlier than the loss value error and the loss difference is usually smaller than the value difference. The magnitude of the loss difference and the value difference are related. These conclusions show that even if the loss value is different, it is challenging to determine which step is reliable and which step is caused by the SDC.

Intermediate Result Observability. Since loss values are often insensitive to transient faults, a theoretical alternative is to instrument the execution flow to capture intermediate states. However, deep instrumentation creates a Heisenbug effect: it fundamentally alters the micro-architectural state (e.g., pipeline timing and thread scheduling), potentially masking the very race conditions that trigger SDCs. We observed this phenomenon when applying heavy instrumentation to the defective GPUs identified in Figure 7. As shown in Figure 14, the error rates for GPU 1 and GPU 2 paradoxically decreased compared to the baseline, suggesting that the overhead altered the execution path enough to hide the corruption. Furthermore, we observed new anomalies (e.g., NaNs) absent in the original run, confusing the diagnosis.

A more practical approach involves verifying the checksums (hashes) of tensor outputs at different execution boundaries, yet this introduces a critical trade-off between diagnostic resolution and reproduction cost, as quantified in Table 4. To achieve precise localization, one must perform fine-grained inspection by checksumming outputs after every kernel execution. While this allows us to pinpoint the exact corrupted tensor and its corresponding kernel, it incurs a prohibitive

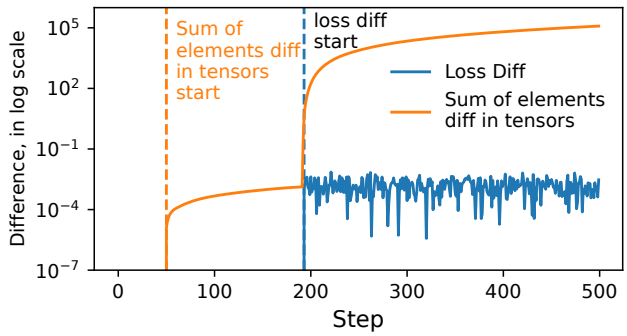


Figure 13: Timestep difference between the loss difference and value difference in log scale.

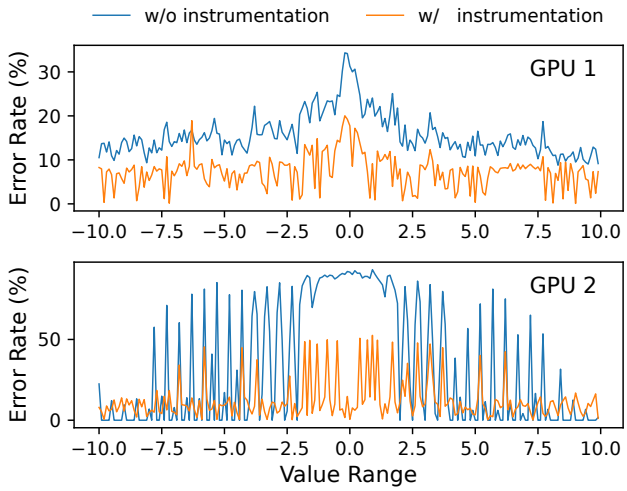


Figure 14: Error rate differences between instrumented and non-instrumented execution.

683.9% overhead. Since diagnosis requires iterative replay, such a 7× slowdown extends the debugging cycle from hours to days, rendering global fine-grained monitoring impractical for production timelines. Conversely, applying coarse-grained checks solely at communication boundaries offers a lightweight alternative with negligible overhead (4.3%). However, this approach suffers from limited resolution: while it can efficiently identify the faulty GPU, it fails to isolate the specific kernel or tensor responsible for the SDC.

5 Diagnosing SDC-defective GPUs at Scale

When an abnormal signal occurs in the online task (e.g., loss decreasing less than expected, Shape Error, etc.), we first determine whether it is a hardware issue. Specifically, we introduce a two-phase diagnosing mechanism. The first phase is a lightweight grouping of all GPUs to identify the suspicious parallel group by replaying the training workload. The second phase is a full tensor comparison within the suspicious GPU groups to pinpoint the specific device. After diagnosis, we confirm the SDC-defective GPUs using offline diagnostics. Before implementing such a two-phase mechanism, we need

Table 4: The overhead and coverage of different instrumentation granularity. ***R**: Rank, **K**: Kernel, **T**: Tensor*. The filled proportion of each circle indicates the coverage achieved by the corresponding instrumentation granularity at the Rank, Kernel, and Tensor levels.

Granularity	Overhead	Coverage	Locating		
			R	K	T
Base	+0%	0%	❍	❍	❍
Comm	+4.3%	5.65%	●	❍	❍
Layer	+36.6%	22.6%	●	◗	❍
Kernel	+683.9%	73.8%	●	●	●

to ensure the training procedure is deterministic.

SDCHUNTER diagnoses the affected job with the exact same model, checkpoint, input shards[4], operator implementations, and communication schedule[5] used by the original training run. The key insight is that deterministic training makes each healthy DP group a bitwise reproducible replica of the same computation, so SDCHUNTER can preserve execution equivalence when it reduces or repartitions DP groups for diagnosis. In the first phase, the parallel configuration[6] is the same as the original training run except that DP groups are organized into replay replicas. In the second phase, SDCHUNTER changes only the DP groups used for comparison while keeping the model, checkpoint, input shards, PP configuration, TP configuration, operator choices, and communication order unchanged. We intentionally avoid a separate lightweight diagnostic model because our characterization shows that GPU SDCs are strongly tied to the production kernel sequence, input value ranges, and communication order. A smaller diagnostic model would improve isolation from the production job, but it would lose the workload fidelity needed to reproduce rare, data-dependent corruptions. Therefore, SDCHUNTER isolates diagnosis by replaying only a bounded window from the latest checkpoint and by moving fine-grained device confirmation off the training critical path, rather than by changing the model under test.

5.1 Pre-requisites: Deterministic Training

To ensure the training procedure is deterministic, we add several deterministic constraints on the computation and communication.

Definition of Deterministic Training. Deterministic training does not eliminate or exclude the numerical randomness in the learning algorithm, such as randomized data order, dropout masks, or other pseudo-random choices. Instead, it

[4]Input shards are partitions of the training data assigned to different workers during distributed training.

[5]A communication schedule specifies the order and grouping of messages exchanged among distributed workers.

[6]A parallel configuration specifies how the training job maps model computation and data across workers, including DP, PP, TP, and EP settings.

makes these sources of randomness reproducible by recording or regenerating the same random sequences from the same checkpoint and configuration. Under this definition, a healthy training system should produce bitwise aligned execution traces when replaying the same checkpoint, input shards, model code, operator implementations, and communication schedule. The purpose is therefore not to eliminate stochastic learning behavior, but to preserve a reproducible training trace.

Such a reproducible training trace is essential for diagnosis. Several industry technical reports indicate that bitwise alignment and deterministic training are becoming standard practice for large-scale model training [1–3]. From the ByteDance production perspective, this property is particularly useful because it separates data issues, model issues, infrastructure issues, operator bugs, and SDC-induced hardware corruption with only several steps of replay under the same replay framework. Our production measurements show that deterministic training preserves normal step time, with less than 0.01% difference from non-deterministic training, while reducing normalized debugging time from 1.00× to 0.30×.

Setup Constraints. To support reproducible training, our training framework setup leverages the deterministic features provided by modern training frameworks and GPU drivers. In practice, we activate every deterministic toggle exposed by the software stack, including disabling `torch.backends.cudnn.benchmark` [10], enabling `torch.backends.cudnn.deterministic` [11] and `torch.use_deterministic_algorithms` [12]. By setting up these configurations, we ensure that computation executes identically in every iteration when the underlying hardware is healthy.

Computation Constraints. The primary source of non-determinism in GPU computation is the non-associativity of floating-point arithmetic, which causes parallel accumulation to vary across executions [7, 25, 68, 79, 80]. To achieve bitwise reproducibility, our framework applies strict constraints to core kernels: we disable optimization heuristics in operator libraries [50, 51, 53], deactivate GEMM autotuning to lock launch configurations, and serialize atomic reductions where needed. Dynamic control flows introduce additional challenges. For elementwise kernels, we enforce a static SM-to-row mapping. For operators with data-dependent routing (e.g., MoE gating), we re-implement kernels to serialize accumulation and scatter steps, ensuring deterministic processing regardless of thread scheduling. For kernels whose outputs can still depend on thread interleavings, we manually implement deterministic kernels whose reduction and write rules produce the same result under any valid execution order. All other standard operators retain their native deterministic behavior to preserve system-wide consistency.

Communication Constraints. As noted regarding network topology influence, the non-associativity of floating-point arithmetic renders collective operations (e.g., AllReduce) sensitive to the underlying reduction structure. Standard communication libraries (e.g., NCCL[7]) typically perform runtime topology detection to optimize bandwidth, dynamically selecting algorithms (e.g., Ring vs. Tree) and adjusting the number of parallel channels based on available hardware health. To eliminate this source of non-determinism, our framework enforces a static communication configuration regardless of the physical topology state. We explicitly lock the collective algorithms and fix the number of communication channels to prevent the library from adapting the reduction tree to hardware fluctuations. Additionally, for dynamic architectures like Mixture-of-Experts (MoE), we enforce a deterministic dispatch order to ensure that gradient aggregation sequences remain invariant across different runs. During training, SDCHUNTER records the communication trace and replays communication according to an immutable deterministic order, so diagnosis uses the same message ordering as the original execution.

These deterministic constraints are enabled during production training, not only during replay. This choice ensures that the communication order and kernel choices in the diagnostic run match those that produced the anomaly. Since the same deterministic configuration is used during both normal execution and diagnosis, SDCHUNTER can replay the alert-triggering window without changing the workload behavior being tested.

5.2 Phase 1: Lightweight Grouping

Figure 15 illustrates the overview of Phase 1. The objective of the first phase is to rapidly narrow the scope of anomalies from the entire cluster to a specific logical group with minimal overhead. This phase should not compromise the SDC characterization. To achieve this, we partition the cluster along the data-parallelism (DP) dimension, forming two replicas, each comprising half of the DP groups. Both replicas are provided with the identical training data used before as input. Because deterministic training fixes random sequences, operator behavior, and communication order, healthy replicas should produce the same signatures even after this DP-level scaling, which preserves the execution equivalence needed to expose SDC-induced divergence.

For each replica, we collect the signatures of the intra-DP-group pipeline parallelism (PP) communication tensors and compare these signatures across the two replicas. Here, the signature refers to a hash of the tensor values designed to satisfy associativity. By aggregating and comparing communication signatures across replicas, we can identify potentially anomalous DP groups. The novelty of this detection step is that it is implemented as a training framework ability, where the framework already knows the checkpoint

[7] NVIDIA Collective Communications Library, a GPU communication library commonly used for collectives such as AllReduce in distributed training.

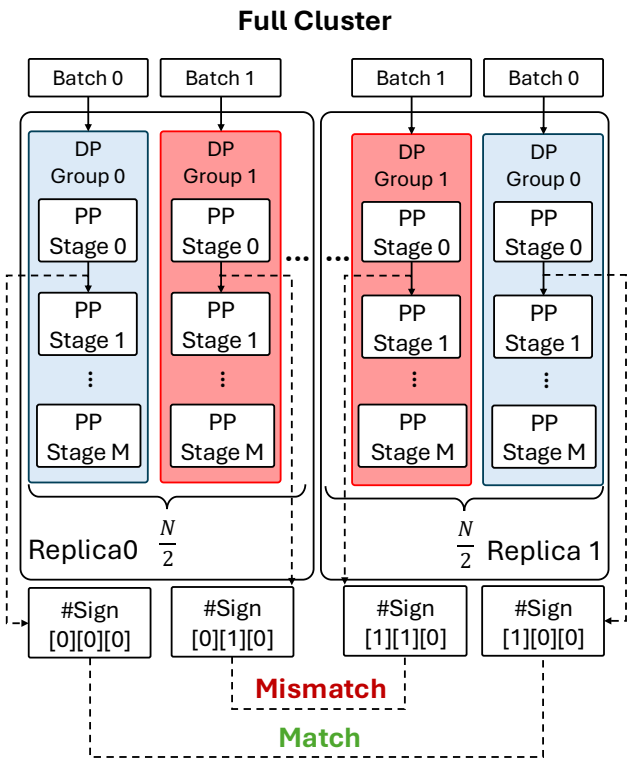


Figure 15: Phase 1: Lightweight Grouping. The indices in the signature label `#Sign [·][·][·]` denote the Replica ID, DP Group ID, and PP Stage ID, respectively.

boundary, DP groups, PP communication tensors, and replay window. As a result, SDCHUNTER only records compact signatures at framework-visible communication boundaries instead of dumping full tensors or inserting external kernel-level monitors, which keeps the online overhead very small. If only one DP group diverges from the majority signature, SDCHUNTER marks the machines in that group as suspicious and removes them from subsequent scheduling. The training job then resumes from the latest verified checkpoint on the remaining healthy machines. This resumption does not wait for the device-level diagnosis in Phase 2 because it only requires isolating the anomalous DP group so that corrupted ranks are removed from the restarted job. The blocking time on the training critical path is therefore the sum of the time to reload the checkpoint, replay the workload, compare the signatures, and restart the scheduler for the next step. Checkpoint reload and scheduler restart are the normal recovery costs, Phase 1 replays fewer than 100 steps in our deployment, and signature comparison takes seconds because only compact communication-boundary hashes are compared.

Ideally, comparing signatures derived from additional parallelism dimensions (e.g., expert parallelism, EP) would further localize the suspicious nodes. However, in modern training architectures that employ EP for Mixture-of-Experts (MoE) models, computation and communication kernels are frequently fused to maximize efficiency [81]. As a result, internal signatures of communication-related tensors within these fused kernels are inaccessible, preventing us from extracting EP-specific signatures.

Fault Model. In our deployment experience, we assume that, within one diagnosis window, at least one replay replica remains healthy and can serve as a reference. When two replicas disagree and no majority exists, SDCHUNTER introduces an additional replay replica on spare healthy machines and compares all three signatures to determine which DP group is faulty. If multiple DP groups are defective, SDCHUNTER can identify multiple outliers as long as they do not all produce the same corrupted signature. The pathological case where every replay replica is simultaneously corrupted in the same way cannot be resolved by consensus. In that case, SDCHUNTER falls back to offline confirmation with known good machines and vendor diagnostics. In practice, this is consistent with the observed production pattern where SDC-defective GPUs are rare and independent, so simultaneous identical corruptions across all DP replicas are unlikely.

5.3 Phase 2: Precise Localization

Once the suspicious parallel group is identified, we perform a comprehensive tensor comparison to pinpoint the specific defective device. This involves collecting tensor signatures from both the suspicious group and a healthy reference group during a deterministic replay of the problematic iteration. By comparing these signatures layer-by-layer, we can detect the exact point of divergence. This process isolates the first corrupted tensor and its producing kernel, allowing us to definitively identify the SDC-defective GPUs. The localization step also relies on framework-level knowledge of tensor ownership, rank[8] placement, and operator order, so SDCHUNTER can map the first divergent tensor back to a physical GPU without globally instrumenting every kernel during normal training. This framework integration is the reason SDCHUNTER can combine accurate device-level localization with much lower overhead than full tensor dumping or always-on fine-grained instrumentation.

5.4 Post-Diagnostic Verification

To verify suspicious GPUs, we employ a hybrid strategy to isolate the minimal reproducible execution trace. First, we utilize Hardware tools [52,55] to screen for permanent hardware faults, any failure triggers immediate repair. However, since SDC often evades standard stress tests, GPUs that pass these hardware tools undergo offline analysis using SDCHUNTER. We subject execution traces from the alert-triggering workload to intensive iterative replay to perform full-scale tensor verification, as extensive repetition of these traces maximizes the likelihood of reproducing the SDC symptoms. Through

[8] A rank is one distributed training process, usually mapped to one accelerator in GPU training.

this process, we progressively isolate the consistently reproducible kernel error, thereby confirming the GPU as SDC-defective.

6 Evaluation

To show the effectiveness of SDCHUNTER, we evaluate it by answering the following questions:

- How much overhead does SDCHUNTER introduce to the training process?
- How does SDCHUNTER perform in alarming SDC?
- How fast does SDCHUNTER perform in locating SDC?

Setup. We evaluate SDCHUNTER on a cluster with 128 and 512 GPUs. Each host contains 168 CPUs, 2TB memory, and 8 GPUs. All the GPUs are connected with high-speed inter-GPU communication and all hosts are connected with high-speed RDMA for inter-host communication.

The evaluation is conducted using ByteDance's production training traces. We use LLM models with 50B and 150B parameters. For baselines, we choose dumping all the tensors and comparing, and dumping all tensors signatures and comparing as our baseline. The dumped tensors will be stored in the GPU memory. If GPU memory is not enough, it will be stored in the host memory. We use the same training configuration as the original training process.

Baseline. We select two types of baselines, offline and online. Offline baselines include offline stress tests and log analysis, including hardware tools [52], GPU-Burn [54, 77] and PEPPA-X [61]. Online baselines include online metrics tracing including loss or gradient norm metric, and He et al. [26]. Also, online baselines contain real-time detection including HWSentinel [18]. Within the online baselines, we add a gold standard, Full Data Dump, which dumps all the tensors and compares them.

6.1 Production Deployment

SDCHUNTER is now deployed in a production GPU-cluster environment and is a part of base infrastructure for large model training in ByteDance. By the time of paper submission, SDCHUNTER has detected 40 SDC-defective GPUs in the production cluster. It has become the first step to check hardware issues when loss spikes occur and the offline debug time is reduced from several days to under one hour. Our investigation revealed that the SDC-defective GPUs detected by SDCHUNTER have occurrence rates ranging from 100% to 0.000001%.

6.2 SDCHUNTER Overhead

We evaluate the end-to-end overhead of SDCHUNTER on the production cluster in Figure 16.

Analysis. For both the 50B and 150B models, SDCHUNTER introduces minimal overhead, which remains below 4% relative to the original training process. In contrast, computing full tensor signatures introduces approximately a

Table 5: The detection ability of SDCHUNTER in detecting SDC GPUs in real-world tracing.

		Coverage (↑ is better)
Offline	Hardware tools [52]	30.0%
	GPU-Burn [54, 77]	20.0%
	PEPPA-X [61]	37.5%
Online	Loss/Grad Norm Metric	25.0%
	He et al. [26]	50.0%
	HWSentinel [18]	17.5%
Full Data Dump		100.0%
SDCHUNTER (Ours)		**100.0%**

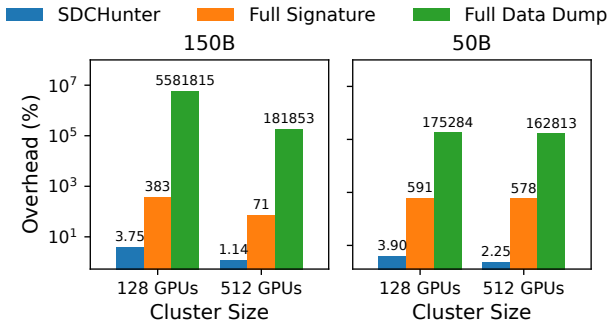


Figure 16: The end-to-end overhead comparison of SDCHUNTER and other tensor value observation methods on 128 and 512 GPUs with model size 50B and 150B.

4× slowdown, and performing full tensor data comparisons results is totally unacceptable. The substantial overheads of these two baselines arise primarily from transferring tensors from GPU to host due to the limited capacity of GPU memory, and also from the need to perform many of these transfers synchronously because the recorded tensors must remain unmodified before they are saved. In practical large-scale training scenarios, GPU memory is typically close to saturation, which further amplifies the overhead incurred by the baseline methods.

6.3 SDCHUNTER Detection Ability

To evaluate the effectiveness of SDCHUNTER in detecting SDCs, we conduct both real-world trace based and fault injection based evaluations. The fault injection is conducted by changing the input or output tensor values during the training process. For the real-world evaluation, we measure the coverage in detecting SDC-defective GPUs since it is impractical to manually replicate the training process to determine the golden baseline. For the fault injection evaluation, we measure the detection ability of SDCHUNTER in overhead, the percentage of SDC GPUs successfully identified, and the mean gap between the SDC occurrence and its first detection by the system. The real-world trace based evaluation results

Table 6: The detection ability of SDCHUNTER in detecting SDC GPUs in fault injection.

		Coverage (↑ is better)	Avg Δ Steps (↓ is better)	Overhead (↓ is better)
Metrics Based	Loss	16%	127	0%
	Grad Norm	17%	109	0%
	He et al. [26]	33%	25	1%
Duplication Based	Full Data Dump	100%	0	>1000%
SDCHUNTER (Ours)		**91%**	**3**	**3.9%**

Table 7: The localization ability of SDCHUNTER in detecting SDC GPUs in real-world trace and fault injection.

		Rank Correctness (↑ is better)	
		Real World	Fault Injection
Offline	Hardware tools [52]	60%	N/A
	GPU-Burn [54, 77]	100%	N/A
	PEPPA-X [61]	60%	57%
Online	Loss	0%	0%
	He et al. [26]	0%	0%
	HWSentinel [18]	0%	0%
Full Data Dump		100%	100%
SDCHUNTER (Ours)		**100%**	**100%**

are presented in Table 5 and the fault injection based results are shown in Table 6.

In the real-world trace evaluation, SDCHUNTER covers all confirmed SDC-defective GPUs, and it reaches 91% coverage under synthetic fault injection. For the fault injection evaluation, SDCHUNTER localizes SDC-defective GPUs in fewer than 5 steps, with 3 steps on average. For the real-world evaluation, SDCHUNTER only needs to replay fewer than 100 steps of the training process in the first stage, which allows the affected training job to resume within one hour after the suspicious DP group is isolated. The offline debugging stage then pinpoints the defective GPU in under one hour and does not block the online job because the training job has already moved away from the suspicious machines. Naive offline stress tests [52,54,77] can only achieve a coverage lower than 40%. PEPPA-X [61] improves coverage by searching for inputs that maximize SDC vulnerability based on the sensitivity of the input to SDC. Although this approach achieves certain gains compared with simple stress-testing methods, the coverage attainable during offline tests remains fundamentally limited. He et al. [26] detects SDCs by monitoring whether the optimizer's gradient history terms and the normalization moving-variance values exceed mathematically derived bounds. This method maintains a low overhead, but the coverage does not reach 50%, and the number of steps from SDC occurrence to metric out-of-bound is still relatively high. We also apply HWSentinel [18] to GPU SDC detection. However, once GPU logs reflect abnormal behavior, it is already too late to detect SDC.

6.4 SDCHUNTER Locating Ability

We assess the localization capability of device-level identification, which pinpoints the specific faulty accelerator, among all the baselines. We define localization accuracy as the percentage of SDC cases where SDCHUNTER correctly identifies the specific device ID. The result of localization accuracy evaluation is shown in Table 7, also including both real-world trace based and fault injection based evaluations.

SDCHUNTER achieves perfect device-level localization, offering 100% accuracy in attributing the SDC to the specific faulty GPU, even in the presence of complex collective communication primitives. This precise localization enables operators to isolate faulty hardware or numeric instabilities swiftly, significantly reducing the debug turnaround time compared to standard bisection methods.

7 Related Work

Silent Data Corruption on CPU. Extensive research targets CPU hardware defects [15, 22, 24, 31, 38, 39, 66, 67], detection [16, 34, 58, 59], and mitigation [17, 23, 27, 44, 76], particularly in HPC environments where computational density amplifies SDC frequency [9, 19, 20, 29, 46, 61]. Mission-critical systems often employ spatial or temporal redundancy [28, 29, 45, 63, 64], but high overheads frequently necessitate specialized processor support [45, 63, 64]. To reduce validation costs, Orthus [40] exploits control-data path separation, using lightweight checksums for control and versioned-memory re-execution for data. While system-level replication guarantees consistency in databases [57] and has been adapted for broader error detection [19, 20, 70], its utility is often limited by synchronization costs and latency in long-running jobs.

Silent Data Corruption on GPU. Recent studies extensively characterize GPU-induced SDC in AI workloads [4, 14, 21, 26, 32, 43]. Research has largely focused on software-based mitigation for inference: SAVE [82] employs bit-level robustness analysis to shield critical memory, while distribution-based methods [35, 42] detect errors by monitoring activa-

tion value bounds. Additionally, Agarwal et al. [4] analyze how transient faults propagate through LLM architectures to impact final outputs. While SDC in training poses unique challenges. Ma et al. [43] reveal that faults can induce catastrophic parameter drift silently, even when loss curves remain unaffected. To counter this, He et al. [26] propose checking optimizer states against derived statistical bounds and recovering by re-executing recent iterations. In proprietary systems, Google reports using deterministic replay to isolate incorrect computations [2, 3], though specific implementation details remain undisclosed.

GPU Determinism. Achieving determinism is critical for debugging and reliability [7, 25, 68, 79, 80]. Nondeterminism often stems from floating-point non-associativity in atomic operations; while some approaches focus on request-level batch invariance [25], they often overlook distributed communication noise. Although newer hardware supports deterministic in-switch reductions [62], communication over RDMA remains inherently non-deterministic. TBIK [80] bridges this gap by designing fully deterministic MatMul and AllReduce kernels for different tensor-parallel configurations.

8 Discussion

Deployment Experience. SDCHUNTER has been deployed as part of the production training infrastructure at ByteDance in cooperation with SDC detection tools [33] and OpGuard [83], where it is used as the first hardware-diagnosis step when training jobs exhibit suspicious symptoms such as loss spikes or unexplained fail-stop errors. By the time of submission, the system had identified 40 SDC-defective GPUs in production. These devices cover a wide range of manifestation frequencies, from faults that reproduce in every replay to faults that appear only rarely, which explains why hardware tools and generic stress tests alone cannot serve as a reliable production gate. In operation, SDCHUNTER first isolates the suspicious DP group so the online training job can move away from the affected machines and resume from a checkpoint. The subsequent offline debugging process pinpoints the defective GPU in under one hour and does not block the online job.

Deployment Lessons. Our experience suggests three practical lessons for operating large-scale LLM training clusters under SDC risk. First, SDC diagnosis must be integrated with the training control plane rather than treated as a standalone hardware test. The most useful action during an incident is to remove suspicious machines from the training path quickly and defer device-level confirmation to offline analysis. Second, exact workload replay is more effective than synthetic stress testing because many defective GPUs only fail for the model kernels, data ranges, and communication schedules that triggered the original anomaly. Third, deterministic training is useful beyond SDC detection. It turns ambiguous symptoms such as loss spikes into reproducible comparisons, which also helps engineers distinguish data issues, model issues, infrastructure problems, operator bugs, and hardware corruption. Finally, SDC occurrence rates vary widely across defective GPUs, so a fixed-duration stress test is not a reliable acceptance criterion. Production systems should retain the alert-triggering trace and replay it when confirming rare faults.

Analysis Ability of Other Problems. The core of SDCHUNTER is to use deterministic operators and instrument the operators to limit the error to within a single GPU, and then locate the faulty GPU by comparing different replicas. SDCHUNTER can be used not only for SDC detection but also for the analysis of other problems. In our production environment, the tool has also helped detect slow nodes, nodes with runtime jitter, and machine issues that cause training jobs to idle.

Towards Real-time Online SDC Detection. While SDCHUNTER currently reacts to explicit anomalies, we envision proactive, always-on SDC monitoring. The main bottleneck is the overhead of frequent tensor signature computation. Future work can reduce this cost through operator fusion. By embedding the checksum arithmetic into the memory load/store of the communication primitives (e.g., fusing hash calculation into the communication Send/Recv kernels), we can compute signatures on-the-fly as data flows through the interconnect. This design can mask the computation cost behind the communication latency or memory bandwidth, enabling zero-overhead, real-time SDC shielding for every training iteration.

9 Conclusion

We characterize GPU Silent Data Corruption (SDC) in production LLM training, showing that SDCs often mimic software bugs and evade synthetic benchmarks. We develop SDCHUNTER, which localizes defective devices through bitwise deterministic replay of the triggering workload. In production, SDCHUNTER identified 40 SDC-defective GPUs, reducing job-blocking recovery to under one hour and confirming each device offline within one hour.

Acknowledgments

We sincerely thank our shepherd and anonymous reviewers of OSDI 2026, whose reviews, feedback, and suggestions have significantly strengthened our work. This research was supported in part by New Generation Information Technology Program from Shanghai Committee of Science and Technology (No. 25511104100) and National Natural Science Foundation of China (No. 62432010). Corresponding author: Jinyu Gu (gujinyu@sjtu.edu.cn, Shanghai Jiao Tong University), Wencong Xiao (hanli.hl@bytedance.com, ByteDance Seed).

References

[1] The llama 3 herd of models, 2024.

[2] Gemini 2.5: Pushing the frontier with advanced reasoning, multimodality, long context, and next generation agentic capabilities, 2025.

[3] Gemini: A family of highly capable multimodal models, 2025.

[4] Udit Kumar Agarwal, Abraham Chan, and Karthik Pattabiraman. Resilience assessment of large language models under transient hardware faults. In *2023 IEEE 34th International Symposium on Software Reliability Engineering (ISSRE)*, pages 659–670, 2023.

[5] Anton Altenbernd, Philipp Wiesner, and Odej Kao. Exploring silent data corruption as a reliability challenge in llm training, 2026.

[6] AMD. Open-field-health-check (ofhc). https://github.com/amd/Open-Field-Health-Check, 2025. MIT license, accessed 2025-12-10.

[7] Berk Atil, Sarp Aykent, Alexa Chittams, Lisheng Fu, Rebecca J. Passonneau, Evan Radcliffe, Guru Rajan Rajagopal, Adam Sloan, Tomasz Tudrej, Ferhan Ture, Zhe Wu, Lixinyu Xu, and Breck Baldwin. Non-determinism of "deterministic" llm settings, 2025.

[8] Pedro Martins Basso, Fernando Fernandes dos Santos, and Paolo Rech. Impact of tensor cores and mixed precision on the reliability of matrix multiplication in gpus. *IEEE Transactions on Nuclear Science*, 67(7):1560–1565, 2020.

[9] Eduardo Berrocal, Leonardo Bautista-Gomez, Sheng Di, Zhiling Lan, and Franck Cappello. Lightweight silent data corruption detection based on runtime data analysis for hpc applications. In *Proceedings of the 24th International Symposium on High-Performance Parallel and Distributed Computing*, HPDC '15, page 275–278, New York, NY, USA, 2015. Association for Computing Machinery.

[10] PyTorch Contributors. torch.backends.cudnn.benchmark — pytorch documentation. https://docs.pytorch.org/docs/stable/backends.html#torch.backends.cudnn.benchmark, 2025. Accessed: 2025-12-09.

[11] PyTorch Contributors. torch.backends.cudnn.deterministic — pytorch documentation. https://docs.pytorch.org/docs/stable/backends.html#torch.backends.cudnn.deterministic, 2025. Accessed: 2025-12-09.

[12] PyTorch Contributors. torch.use_deterministic_algorithms — pytorch documentation. https://docs.pytorch.org/docs/stable/generated/torch.use_deterministic_algorithms.html, 2025. Accessed: 2025-12-09.

[13] NVIDIA Corporation. Nvidia gpu memory error management. Technical Report DA-09826-002_v001, NVIDIA Corporation, Apr 2025. Accessed: 2025-12-09.

[14] Shengkun Cui, Archit Patke, Hung Nguyen, Aditya Ranjan, Ziheng Chen, Phuong Cao, Brett Bode, Gregory Bauer, Catello Di Martino, Saurabh Jha, Chandra Narayanaswami, Daby Sow, Zbigniew T. Kalbarczyk, and Ravishankar K. Iyer. Characterizing gpu resilience and impact on ai/hpc systems, 2025.

[15] Peter W. Deutsch, Vincent Quentin Ulitzsch, Sudhanva Gurumurthi, Vilas Sridharan, Joel S. Emer, and Mengjia Yan. Delayavf: Calculating architectural vulnerability factors for delay faults. In *2024 57th IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 231–245, 2024.

[16] Harish Dattatraya Dixit, Laura Boyle, Gautham Vunnam, Sneha Pendharkar, Matt Beadon, and Sriram Sankar. Detecting silent data corruptions in the wild, 2022.

[17] Harish Dattatraya Dixit, Sneha Pendharkar, Matt Beadon, Chris Mason, Tejasvi Chakravarthy, Bharath Muthiah, and Sriram Sankar. Silent data corruptions at scale, 2021.

[18] Rhea Dutta, Harish Dattatraya Dixit, Rik Van Riel, Gautham Vunnam, and Sriram Sankar. *Hardware Sentinel: Protecting Software Applications from Hardware Silent Data Corruptions*, page 482–497. Association for Computing Machinery, New York, NY, USA, 2025.

[19] James Elliott, Kishor Kharbas, David Fiala, Frank Mueller, Kurt Ferreira, and Christian Engelmann. Combining partial redundancy and checkpointing for hpc. In *2012 IEEE 32nd International Conference on Distributed Computing Systems*, pages 615–626, 2012.

[20] David Fiala, Frank Mueller, Christian Engelmann, Rolf Riesen, Kurt Ferreira, and Ron Brightwell. Detection and correction of silent data corruption for large-scale high-performance computing. In *SC '12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, pages 1–12, 2012.

[21] Nishant George, Harish Dattatraya Dixit, Emel Goksu, Bharath Parthasarathy, Amber Huffman, Sudhanva Gurumurthi, Vilas Sridharan, Thiago Macieira, Arani Sinha,

Lisa Minwell, Dean Liberty, and Rob Chappell. Silent data corruption in ai. White paper, Open Compute Project, August 2025. Version 1.0, Creative Commons Attribution-ShareAlike 4.0 International License.

[22] Nick Hagerty, Jordan Webb, Veronica Melesse Vergara, and Matt Ezell. Experiences detecting defective hardware in exascale supercomputers. In *Proceedings of the SC '23 Workshops of the International Conference on High Performance Computing, Network, Storage, and Analysis*, SC-W '23, page 619–626, New York, NY, USA, 2023. Association for Computing Machinery.

[23] Siva Kumar Sastry Hari, Sarita V. Adve, and Helia Naeimi. Low-cost program-level detectors for reducing silent data corruptions. In *IEEE/IFIP International Conference on Dependable Systems and Networks (DSN 2012)*, pages 1–12, 2012.

[24] Siva Kumar Sastry Hari, Paolo Rech, Timothy Tsai, Mark Stephenson, Arslan Zulfiqar, Michael Sullivan, Philip Shirvani, Paul Racunas, Joel Emer, and Stephen W. Keckler. Estimating silent data corruption rates using a two-level model, 2020.

[25] Horace He and Thinking Machines Lab. Defeating nondeterminism in llm inference. *Thinking Machines Lab: Connectionism*, 2025. https://thinkingmachines.ai/blog/defeating-nondeterminism-in-llm-inference/.

[26] Yi He, Mike Hutton, Steven Chan, Robert De Gruijl, Rama Govindaraju, Nishant Patil, and Yanjing Li. Understanding and mitigating hardware failures in deep learning training systems. In *Proceedings of the 50th Annual International Symposium on Computer Architecture*, ISCA '23, New York, NY, USA, 2023. Association for Computing Machinery.

[27] Peter H. Hochschild, Paul Turner, Jeffrey C. Mogul, Rama Govindaraju, Parthasarathy Ranganathan, David E. Culler, and Amin Vahdat. Cores that don't count. In *Proceedings of the Workshop on Hot Topics in Operating Systems*, HotOS '21, page 9–16, New York, NY, USA, 2021. Association for Computing Machinery.

[28] Ted Hong, Yanjing Li, Sung-Boem Park, Diana Mui, David Lin, Ziyad Abdel Kaleq, Nagib Hakim, Helia Naeimi, Donald S. Gardner, and Subhasish Mitra. Qed: Quick error detection tests for effective post-silicon validation. In *2010 IEEE International Test Conference*, pages 1–10, 2010.

[29] Yafan Huang, Shengjian Guo, Sheng Di, Guanpeng Li, and Franck Cappello. Mitigating silent data corruptions in hpc applications across multiple program inputs. In *SC22: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–14, 2022.

[30] Yuxuan Jiang, Ziming Zhou, Boyu Xu, Beijie Liu, Runhui Xu, and Peng Huang. Training with confidence: Catching silent errors in deep learning training with automated proactive checks. In *Proceedings of the 19th USENIX Symposium on Operating Systems Design and Implementation*, OSDI '25, Boston, MA, USA, July 2025. USENIX Association.

[31] Nikos Karystinos, Odysseas Chatzopoulos, George-Marios Fragkoulis, George Papadimitriou, Dimitris Gizopoulos, and Sudhanva Gurumurthi. Harpocrates: Breaking the silence of cpu faults through hardware-in-the-loop program generation. In *2024 ACM/IEEE 51st Annual International Symposium on Computer Architecture (ISCA)*, pages 516–531, 2024.

[32] Jack Kosaian and K. V. Rashmi. Arithmetic-intensity-guided fault tolerance for neural network inference on gpus. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '21, New York, NY, USA, 2021. Association for Computing Machinery.

[33] Kinman Lei, Liyan Zheng, Xiang Li, Hongmin Chen, Yun Zhang, Gaohong Liu, Zuquan Song, Zixuan Ma, Zhiyu Xue, Minghui Yu, Shuguang Wang, Wencong Xiao, Haibin Lin, Yuyang Jin, Jidong Zhai, Bo Liu, and Xin Liu. Safeguarding LLM training at scale: Online SDC detection and insights from 35 million GPU hours. In *20th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2026, Seattle, WA, USA, July 13–15, 2026*. USENIX Association, 2026.

[34] David P. Lerner, Benson Inkley, Shubhada H. Sahasrabudhe, Ethan Hansen, Luis D. Rojas Munoz, and Arjan van de Ven. Optimization of tests for managing silicon defects in data centers. In *2022 IEEE International Test Conference (ITC)*, pages 578–582, 2022.

[35] Guanpeng Li, Siva Kumar Sastry Hari, Michael Sullivan, Timothy Tsai, Karthik Pattabiraman, Joel Emer, and Stephen W. Keckler. Understanding error propagation in deep learning neural network (dnn) accelerators and applications. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '17, New York, NY, USA, 2017. Association for Computing Machinery.

[36] Sihuan Li, Jianyu Huang, Ping Tak Peter Tang, Daya Khudia, Jongsoo Park, Harish Dattatraya Dixit, and Zizhong Chen. Efficient soft-error detection for low-precision deep learning recommendation models. In *2022 IEEE International Conference on Big Data (Big Data)*, pages 1556–1563, 2022.

[37] Xiaolong Li, Zhi-Qin John Xu, and Zhongwang Zhang. Loss spike in training neural networks, 2024.

[38] Fan Lin, Matt Beadon, Harish Dattatraya Dixit, Gautham Vunnam, Amol Desai, and Sriram Sankar. Hardware remediation at scale. In *2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks Workshops (DSN-W)*, pages 14–17, 2018.

[39] Fred Lin, Antonio Davoli, Imran Akbar, Sukumar Kalmanje, Leandro Silva, John Stamford, Yanai Golany, Jim Piazza, and Sriram Sankar. Predicting remediations for hardware failures in large-scale datacenters. In *2020 50th Annual IEEE-IFIP International Conference on Dependable Systems and Networks-Supplemental Volume (DSN-S)*, pages 13–16, 2020.

[40] Chenxiao Liu, Zhenting Zhu, Quanxi Li, Yanwen Xia, Yifan Qiao, Xiangyun Deng, Youyou Lu, Tao Xie, Huimin Cui, Zidong Du, Harry Xu, and Chenxi Wang. Orthrus: Efficient and timely detection of silent user data corruption in the cloud with resource-adaptive computation validation. In *Proceedings of the ACM SIGOPS 31st Symposium on Operating Systems Principles*, SOSP '25, page 286–304, New York, NY, USA, 2025. Association for Computing Machinery.

[41] Yunchi Lu, Youshan Miao, Cheng Tan, Peng Huang, Yi Zhu, Xian Zhang, and Fan Yang. Trainverify: Equivalence-based verification for distributed LLM training. In Youjip Won, Youngjin Kwon, Ding Yuan, and Rebecca Isaacs, editors, *Proceedings of the ACM SIGOPS 31st Symposium on Operating Systems Principles, SOSP 2025, Lotte Hotel World, Seoul, Republic of Korea, October 13-16, 2025*, pages 237–253. ACM, 2025.

[42] Dongning Ma, Fred Lin, Alban Desmaison, Joel Coburn, Daniel Moore, Sriram Sankar, and Xun Jiao. Dr. dna: Combating silent data corruptions in deep learning using distribution of neuron activations. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, ASPLOS '24, page 239–252, New York, NY, USA, 2024. Association for Computing Machinery.

[43] Jeffrey Ma, Hengzhi Pei, Leonard Lausen, and George Karypis. Understanding silent data corruption in llm training, 2025.

[44] Subhasish Mitra, Subho S. Banerjee, Martin Dixon, Mike Fuller, Rama Govindaraju, Peter Hochschild, Eric X. Liu, Bharath Parthasarathy, and Parthasarathy Ranganathan. Silent data corruption by 10× test escapes threatens reliable computing. *IEEE Des. Test*, 42(6):40–53, 2025.

[45] S.S. Mukherjee, M. Kontz, and S.K. Reinhardt. Detailed design and evaluation of redundant multi-threading alternatives. In *Proceedings 29th Annual International Symposium on Computer Architecture*, pages 99–110, 2002.

[46] S.S. Mukherjee, C. Weaver, J. Emer, S.K. Reinhardt, and T. Austin. A systematic methodology to compute the architectural vulnerability factors for a high-performance microprocessor. In *Proceedings. 36th Annual IEEE/ACM International Symposium on Microarchitecture, 2003. MICRO-36.*, pages 29–40, 2003.

[47] Chandra Sekhar Mummidi, Sandeep Bal, Brunno F. Goldstein, Sudarshan Srinivasan, and Sandip Kundu. A highly-efficient error detection technique for general matrix multiplication using tiled processing on simd architecture. In *2022 IEEE 40th International Conference on Computer Design (ICCD)*, pages 529–536, 2022.

[48] Chandra Sekhar Mummidi, Victor C. Ferreira, Sudarshan Srinivasan, and Sandip Kundu. Highly efficient self-checking matrix multiplication on tiled amx accelerators. *ACM Trans. Archit. Code Optim.*, 21(2), February 2024.

[49] NVIDIA. Nvidia data center gpu manager (dcgm). https://github.com/NVIDIA/DCGM, 2025. Apache-2.0 license, accessed 2025-12-10.

[50] NVIDIA Corporation. cuBLAS: Gpu-accelerated basic linear algebra subprograms. https://developer.nvidia.com/cublas, 2025. Accessed: 2025-12-11.

[51] NVIDIA Corporation. CUTLASS: Cuda templates and python dsls for high-performance linear algebra. https://github.com/NVIDIA/cutlass, 2025. Accessed: 2025-12-11.

[52] NVIDIA Corporation. *Extended Utility Diagnostics (EUD) — NVIDIA DCGM User Guide*. NVIDIA, 2025. Accessed: 2025-12-11, https://docs.nvidia.com/datacenter/dcgm/latest/user-guide/dcgm-eud.html.

[53] NVIDIA Corporation. NVIDIA CUDA Deep Neural Network Library (cuDNN). https://developer.nvidia.com/cudnn, 2025. Accessed: 2025-12-11.

[54] NVIDIA Corporation. *Targeted Stress Plugin — DCGM User Guide*. NVIDIA, 2025. Accessed: 2025-12-11, https://docs.nvidia.com/datacenter/dcgm/latest/user-guide/diag-targeted-stress-plugin.html.

[55] NVIDIA Corporation. *XID Errors*. NVIDIA Corporation, Santa Clara, CA, USA, release r590, v590 edition, December 2025. Online; accessed 11 December 2025.

[56] Ataberk Olgun, Majd Osseiran, Abdullah Giray Yaglikci, Yahya Can Tugrul, Haocong Luo, Steve Rhyner, Behzad Salami, Juan Gomez Luna, and Onur Mutlu. Read disturbance in high bandwidth memory: A detailed experimental study on hbm2 dram chips, 2024.

[57] Diego Ongaro and John Ousterhout. In search of an understandable consensus algorithm. In *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference*, USENIX ATC'14, page 305–320, USA, 2014. USENIX Association.

[58] George Papadimitriou and Dimitris Gizopoulos. Silent data corruptions: Microarchitectural perspectives. *IEEE Transactions on Computers*, 72(11):3072–3085, 2023.

[59] George Papadimitriou, Dimitris Gizopoulos, Harish Dattatraya Dixit, and Sriram Sankar. Silent data corruptions: The stealthy saboteurs of digital integrity. In *2023 IEEE 29th International Symposium on On-Line Testing and Robust System Design (IOLTS)*, pages 1–7, 2023.

[60] Hengzhi Pei, Leonard Lausen, and George Karypis. Connecting the Impact of Silent Data Corruption With Different Training Characteristics: An Empirical Study . *IEEE Micro*, 46(01):44–51, January 2026.

[61] Md Hasanur Rahman, Aabid Shamji, Shengjian Guo, and Guanpeng Li. Peppa-x: Finding program test inputs to bound silent data corruption vulnerability in hpc applications. In *SC21: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–14, 2021.

[62] Rainlin007 and other contributors. same data all reduce on h20, but results are different — issue #1497 comment on nvidia/nccl github repository. https://github.com/NVIDIA/nccl/issues/1497#issuecomment-3210819243, 2024. Comment posted 2024-10-28, accessed 2025-12-09.

[63] S.K. Reinhardt and S.S. Mukherjee. Transient fault detection via simultaneous multithreading. In *Proceedings of 27th International Symposium on Computer Architecture (IEEE Cat. No.RS00201)*, pages 25–36, 2000.

[64] E. Rotenberg. Ar-smt: a microarchitectural approach to fault tolerance in microprocessors. In *Digest of Papers. Twenty-Ninth Annual International Symposium on Fault-Tolerant Computing (Cat. No.99CB36352)*, pages 84–91, 1999.

[65] Fernando Fernandes dos Santos, Siva Kumar Sastry Hari, Pedro Martins Basso, Luigi Carro, and Paolo Rech. Demystifying gpu reliability: Comparing and combining beam experiments, fault simulation, and profiling. In *2021 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 289–298, 2021.

[66] Bianca Schroeder, Eduardo Pinheiro, and Wolf-Dietrich Weber. Dram errors in the wild: A large-scale field study. In *SIGMETRICS*, 2009.

[67] Kostya Serebryany, Maxim Lifantsev, Konstantin Shtoyk, Doug Kwan, and Peter Hochschild. Silifuzz: Fuzzing cpus by proxy, 2021.

[68] Sanjif Shanmugavelu, Mathieu Taillefumier, Christopher Culver, Oscar Hernandez, Mark Coletti, and Ada Sedova. Impacts of floating-point non-associativity on reproducibility for hpc and deep learning applications, 2024.

[69] Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. Megatron-lm: Training multi-billion parameter language models using model parallelism. *arXiv preprint arXiv:1909.08053*, 2019.

[70] J. R. Sklaroff. Redundancy management technique for space shuttle computers. *IBM Journal of Research and Development*, 20(1):20–28, 1976.

[71] Sho Takase, Shun Kiyono, Sosuke Kobayashi, and Jun Suzuki. Spike no more: Stabilizing the pre-training of large language models, 2025.

[72] Abhishek Tyagi, Saurabh Hukerikar, Nirmal Saxena, Yanxiang Huang, Philip Shirvani, Chung-Hsuan Tung, and Yuhao Zhu. Llm-prism: Characterizing silent data corruption from permanent gpu faults in llm training, 2026.

[73] Borui Wan, Gaohong Liu, Zuquan Song, Jun Wang, Yun Zhang, Guangming Sheng, Shuguang Wang, Houmin Wei, Chenyuan Wang, Weiqiang Lou, Xi Yang, Mofan Zhang, Kaihua Jiang, Cheng Ren, Xiaoyun Zhi, Menghan Yu, Zhe Nan, Zhuolin Zheng, Baoquan Zhong, Qinlong Wang, Huan Yu, Jinxin Chi, Wang Zhang, Yuhan Li, Zixian Du, Sida Zhao, Yongqiang Zhang, Jingzhe Tang, Zherui Liu, Chuan Wu, Yanghua Peng, Haibin Lin, Wencong Xiao, Xin Liu, and Liang Xiang. Robust llm training infrastructure at bytedance. In *Proceedings of the ACM SIGOPS 31st Symposium on Operating Systems Principles*, SOSP '25, page 186–203, New York, NY, USA, 2025. Association for Computing Machinery.

[74] Meiqi Wang, Han Qiu, Longnv Xu, Di Wang, Yuanjie Li, Tianwei Zhang, Jun Liu, and Hewu Li. A case for application-aware space radiation tolerance in orbital computing, 2024.

[75] Shaobu Wang, Guangyan Zhang, Junyu Wei, Yang Wang, Jiesheng Wu, and Qingchao Luo. Understanding silent data corruptions in a large production CPU population. In Jason Flinn, Margo I. Seltzer, Peter Druschel,

Antoine Kaufmann, and Jonathan Mace, editors, *Proceedings of the 29th Symposium on Operating Systems Principles, SOSP 2023, Koblenz, Germany, October 23-26, 2023*, pages 216–230. ACM, 2023.

[76] Shaobu Wang, Guangyan Zhang, Junyu Wei, Yang Wang, Jiesheng Wu, and Qingchao Luo. Understanding silent data corruptions in a large production cpu population. In *Proceedings of the 29th Symposium on Operating Systems Principles*, SOSP '23, page 216–230, New York, NY, USA, 2023. Association for Computing Machinery.

[77] wilicc. gpu-burn: Multi-gpu cuda stress test. GitHub repository, 2025. Accessed: 2025-12-11, https://github.com/wilicc/gpu-burn.

[78] Blaise Agüera y Arcas, Travis Beals, Maria Biggs, Jessica V. Bloom, Thomas Fischbacher, Konstantin Gromov, Urs Köster, Rishiraj Pravahan, and James Manyika. Towards a future space-based, highly scalable ai infrastructure system design, 2025.

[79] Jiayi Yuan, Hao Li, Xinheng Ding, Wenya Xie, Yu-Jhe Li, Wentian Zhao, Kun Wan, Jing Shi, Xia Hu, and Zirui Liu. Understanding and mitigating numerical sources of nondeterminism in llm inference, 2025.

[80] Ziyang Zhang, Xinheng Ding, Jiayi Yuan, Rixin Liu, Huizi Mao, Jiarong Xing, and Zirui Liu. Deterministic inference across tensor parallel sizes that eliminates training-inference mismatch, 2025.

[81] Chenggang Zhao, Shangyan Zhou, Liyue Zhang, Chengqi Deng, Zhean Xu, Yuxuan Liu, Kuai Yu, Jiashi Li, and Liang Zhao. Deepep: an efficient expert-parallel communication library. https://github.com/deepseek-ai/DeepEP, 2025.

[82] Wenxin Zheng, Bin Xu, Jinyu Gu, and Haibo Chen. Save: Software-implemented fault tolerance for model inference against gpu memory bit flips. In *Proceedings of the 2025 USENIX Annual Technical Conference (USENIX ATC '25)*. USENIX Association, 2025.

[83] Ziming Zhou, Yinjie Zhao, Hang Zhu, Wenxiao Wang, Zhihao Bai, Yun Zhang, Shuguang Wang, Haibin Lin, and Peng Huang. OpGuard: Bitwise alignment for precise and general debugging of production LLM training. In *20th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2026, Seattle, WA, USA, July 13–15, 2026*. USENIX Association, 2026.

Safeguarding LLM Training at Scale: Online SDC Detection and Insights from 35 Million GPU Hours

Kinman Lei[†,*] Liyan Zheng[*] Xiang Li Hongmin Chen Yun Zhang Gaohong Liu
Zuquan Song Zixuan Ma Zhiyu Xue[†] Minghui Yu Shuguang Wang Wencong Xiao
Haibin Lin Yuyang Jin[†] Jidong Zhai[†] Bo Liu Xin Liu

[†]*Tsinghua University* *ByteDance*

Abstract

Silent Data Corruption (SDC) poses a critical threat to large-scale LLM training. Existing offline tests and online detection methods provide solutions for large-scale systems, yet they suffer from high overhead or low detection accuracy in LLM training. This paper presents AEGIS, an online SDC detection framework for large-scale LLM training. We introduce a two-stage cSensor-cVerifier abstraction that decouples SDC detection into lightweight corruption sensing and definitive corruption verification. Based on this abstraction, AEGIS co-designs new detection techniques by integrating the *inherent features* of LLM training with GPU characteristics, enabling practical online SDC detection. In a production deployment spanning 3.5×10^7 GPU-hours, AEGIS identified 18 *real-world* SDC incidents and 13 faulty GPUs while incurring only 0.86% performance overhead, enabling a systematic empirical characterization of SDCs in large-scale LLM training.

1 Introduction

Large language models (LLMs) have demonstrated impressive capabilities across a wide range of domains [2, 9, 15]. To further explore the potential, many efforts have been devoted to scaling model training to hundreds of billions or even trillions of parameters, using datasets with comparable numbers of tokens. While this scaling delivers gains in performance and generalization, it requires extraordinary computational resources, with training often lasting weeks or months on large-scale GPU clusters. For example, Llama 3 405B [15] is trained on up to 16K H100 GPUs. MegaScale [19] reported employing around 10,000 GPUs to train LLMs.

With the continuous scaling-up of LLM training tasks, a class of low-frequency faults, namely silent data corruption (SDC), has emerged as one of the most challenging and devastating system failures. An SDC occurs when hardware produces incorrect results without triggering any hardware or software alarms. Such faults can destabilize training, degrade model quality, and even cause failures in LLM training tasks that cost millions of dollars. Figure 1 provides an example of GPU SDC incident in our in-house LLM training task involving around 10,000 GPUs. It causes pronounced gradient norm and loss spikes, indicating the model degrades significantly under the influence of SDC. Despite the severe consequences of SDC, the current understanding of SDC in LLM training—including its frequency, types, and root causes—remains extremely limited. This knowledge gap persists because the silent, low-frequency, and non-deterministic nature of SDC makes its detection and study exceedingly difficult.

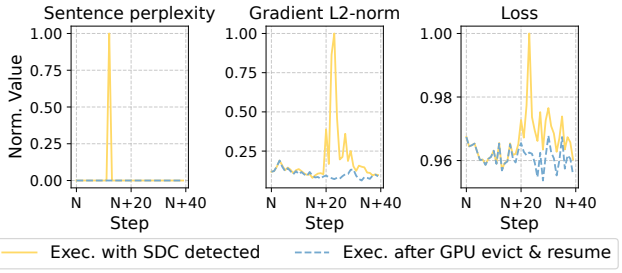


Figure 1: The impact of an SDC incident detected in a training task with around 10,000 GPUs.

Existing approaches fall into three broad categories, none of which is satisfactory for production LLM training. Offline diagnostics suites [3, 32] interrupt training and validate only a limited set of test workloads, leaving correctness during real production runs uncertain. Rerunning-based approaches [41] can provide strong confirmation, but typically require substantial duplicated computation and are therefore too expensive at LLM-training scale. Algorithm-based online detection [5, 6, 8, 21, 45] is attractive in principle, but directly applying it to modern low-precision training [23] is difficult: checksum mismatches can be dominated by ordinary floating-point error, causing false positives or triggering too many costly verifications [27]. As a result, existing methods either demand prohibitive computational resources or achieve unsatisfying detection efficacy. This makes them impractical

[*] Equal contribution.

for LLM training, where correctness is paramount but hardware resources (both computation and memory) are tightly constrained. Consequently, *how to effectively detect SDCs and safeguard large-scale LLM training* remains a significant open problem.

To address this challenge, we design AEGIS, an online SDC detection framework tailored for large-scale LLM training. Our key idea is a two-stage cSensor-cVerifier abstraction that reconciles two conflicting requirements: suspicious computations must be sensed immediately on the critical path, yet definitive confirmation must remain off the critical path. cSensor runs inline with training, performs lightweight detector-specific sensing, captures the minimal verification context needed for later confirmation, and emits compact verification tasks before the training framework releases or overwrites the relevant state. cVerifier then consumes these tasks to perform definitive confirmation. Because replay-based confirmation can be expensive, AEGIS schedules verification into naturally occurring idle periods such as pipeline bubbles [16, 30], improving assurance without perturbing training throughput.

The cSensor-cVerifier abstraction turns online SDC detection into a common system interface rather than a collection of ad hoc detectors. Based on this abstraction, AEGIS instantiates two categories of SDC detection methods that exploit complementary opportunities exposed by modern LLM training systems and accelerator hardware.

Mixed-precision-aware Algorithmic Detection. While low-precision arithmetic undermines classical algorithmic detection, we observe that the tensor processing units in modern GPUs feature high-precision accumulation. Based on this characteristic, AEGIS adopts a mixed-precision checksum method to significantly distinguish SDCs from numerical errors. Beyond protecting `Matmul`, AEGIS leverages the *inherent algebraic property* of attention mechanism to enable a novel, low-overhead algorithmic detection. During cSensor, AEGIS records checksums and a small slice of inputs/outputs; during cVerifier, it lazily replays the highly suspicious computation and performs a bitwise comparison between two outputs, enabling deterministic SDC detection.

Self-equivalence-based Deterministic Detection. AEGIS also exploits *inherently redundant computations* in LLM training to enable low-overhead SDC detection. This self-equivalence arises both at the operator level (e.g., FlashAttention [10, 11, 35] recomputing forward intermediates in backward) and at the framework level (e.g., activation recomputation [42]). During cSensor, AEGIS records a compact fingerprint of outputs for each self-equivalent segment; during cVerifier, it cross-checks fingerprints in the two executions to flag mismatches as SDCs.

We implement AEGIS as a lightweight system on top of our in-house Megatron-LM [36] implementation, requiring minimal changes to the training pipeline. In a deployment spanning 3.5×10^7 GPU-hours on our in-house LLM training jobs, AEGIS identified 18 SDC incidents and 13 faulty GPUs while incurring only 0.86% performance overhead on training tasks of around 10,000 GPUs. After the deployment, only one additional SDC was observed through other methods in training tasks (e.g., periodical task replay test), indicating the effectiveness of AEGIS.

To expedite efforts toward improving the correctness and stability of LLM training, we present a systematic study of SDCs in large-scale LLM training based on AEGIS. Our research yields several intriguing and crucial observations (§3.1 and §8), such as 1) The majority of SDC incidents silently corrupted training correctness and escaped alarms in training frameworks (e.g., `NaN` and `inf` detection). 2) All observed SDC incidents occurred non-deterministically even with the same program. 3) All detected SDCs were identified as permanent SDCs, which stem from GPU faults and recur on the same devices.

In summary, this paper makes the following contributions.

- We propose an online SDC detection system for large-scale LLM training, which decouples detection into time-critical corruption sensor and lazily-executable corruption verifier for both detection efficiency and accuracy.
- We share our experiences in deploying AEGIS in real-world large-scale LLM training and present insights gleaned from 35 million GPU hours, aiming to advance research in safeguarding the scaling of LLM training.
- We identify mathematically equivalent and self-equivalent features inherent to LLM training and leverage them for SDC detection. This substantially reduces the overhead of the primary sensing stage.
- We design and implement AEGIS based on the proposed techniques. We evaluate AEGIS on production-level training jobs over 3.5×10^7 GPU hours and detect 18 real-world SDCs.

2 Background

2.1 LLM Training at Scale

Transformer Architecture. The Transformer architecture serves as the foundational backbone for the majority of LLMs, consisting of stacked layers each with an attention and a feed-forward network (FFN) module. Computation is dominated by matrix multiplications (`Matmul`): the attention block involves six `Matmul` operations (projections, scoring, and aggregation), while the FFN mainly includes two large up- and down-projections.

Distributed Infrastructure. Training state-of-the-art models involves hundreds of billions to trillions of parameters, typically requiring thousands of GPUs operating in parallel for weeks or even months. Such jobs are orchestrated by large-scale distributed training frameworks that employ a variety of parallelism strategies to efficiently utilize cluster resources, including data parallelism [34], tensor parallelism [36], pipeline

parallelism [16, 30, 31, 33], and sequence parallelism [17, 25]. Workloads are dominated by linear algebra kernels, and typically executed in `bfloat16` alongside memory optimizations like activation recomputation [42].

SDCs in LLM Training. Despite the prohibitive cost of LLM training, reliability has received less attention than throughput optimization. SDCs pose a severe threat at scale; rare hardware faults can happen frequently in a large cluster and then cause catastrophic loss spikes or `NaN` values. However, SDCs can be masked by normal loss fluctuations, making loss monitoring unreliable. Moreover, distributed collective communication can rapidly propagate a single corrupted value across the cluster, jeopardizing the training run. In this work, we focus on detecting such infrequent, hardware-induced errors within large-scale GPU clusters.

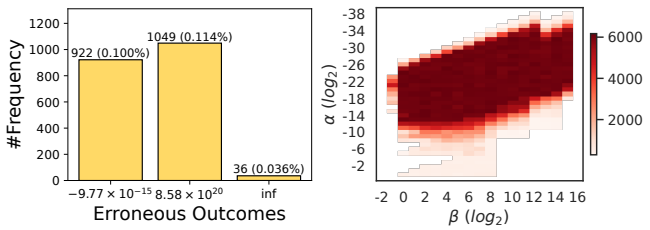
2.2 Existing Approaches to Detect SDCs

Hardware Diagnostic Suites. One primary approach to fault localization is hardware diagnostic suites [3, 32], which relies on offline stress testing. This approach involves running expert-designed, stop-time test suites, such as NVIDIA Extended Utility Diagnostics (EUD) [32], to validate hardware health. However, this method has critical limitations for SDC detection. These test suites often miss non-deterministic SDCs, yielding low recall (about 70% in production [41]), and are costly and slow, taking over 8 hours and substantial cluster resources to identify a single faulty GPU.

Rerunning-based Testing. Another approach is rerunning-based testing, which attempts to locate faults within the real training environment [41]. The core idea is to compare two identical, deterministic executions. ByteRobust [41] proposes a dual-phase replay mechanism. This approach partitions the cluster into two dimensions (horizontally and vertically), and replays the deterministic workloads on each set to locate the faulty machine via intersection. While these methods are more efficient at reproducing live faults, they still incur significant extra overhead at scale.

Algorithm-based Online Detection. As an alternative to the offline approach, algorithm-based online detection [5, 6, 8, 21, 45] is a prominent online detection technique that utilizes algorithm-specific properties (e.g., checksums) for online SDC detection in matrix multiplication. Specifically, prior work [6] proposes adding a checksum to matrix multiplication to flag whether SDCs have occurred during computation by comparing the results of two equivalent computation paths, e.g., $C\mathbf{1} = (AB)\mathbf{1} = A(B\mathbf{1})$, where $\mathbf{1} = [1, 1, \ldots, 1]^{\mathrm{T}}$, A, B are matrices. However, these classical techniques face a significant challenge in LLM training. As low-precision data types like `bfloat16` are common to LLM training, accumulated floating-point errors can easily exceed the error bound E designed for `float32`, resulting in a high rate of false positives [27] or requiring prohibitively many computation replays for verification.

As a result, existing approaches either require prohibitive computational resources or deliver unsatisfying detection accuracy. This limits their practicality in LLM training, where correctness is crucial but hardware resources—especially compute and memory—are severely constrained.



(a) (b)

Figure 2: Some observations on SDC incidents. (a) Frequency of erroneous outcomes when replaying a faulty `Matmul` workload (9.2×10^5 replays; expected correct result -8.58×10^{20}); and (b) SDC occurrence probability in matrix multiplication with varying inputs, simulated as $C = \alpha A \times \beta B$.

3 Observations and Motivation

3.1 Observations on SDC Incidents

Observation 1: Non-deterministic Reproducibility. In all SDC incidents we observed, their behavior was inherently non-deterministic: executing the same workload on the same faulty hardware did not always result in SDC. For instance, on a GPU where SDC had previously been observed, we repeatedly executed the nanoGPT [1] 50 times under identical settings; only one run showed divergent loss. Moreover, the numerical errors induced by SDC are also non-deterministic. By investigating this incident, we traced the fault to the `Matmul` operator. We re-executed this `Matmul` kernel 9.2×10^5 times and observed only three distinct erroneous outcomes, corresponding to an overall estimated error rate of 2×10^{-9}. Figure 2(a) summarizes these erroneous outcomes. When the correct result should have been -8.58×10^{20}, the faulty GPU instead returned either a near-zero value, a large but incorrect value with flipped sign, or `inf`. These observations indicate that a single execution of stress testing is insufficient to uncover all faulty GPUs, as SDCs may not be deterministically reproducible.

Observation 2: Workload Sensitivity. We observe that many SDC incidents are also input-sensitive, depending on both workload and numerical values, and often appearing only under specific conditions. For the SDC reported in Observation 1, we adjusted the input tensors A and B of the `Matmul` operation. Specifically, by introducing scaling factors α and β, we measured the probability of triggering SDC under different inputs of $C = \alpha A \times \beta B$. As shown in Figure 2(b), SDCs occur only within certain numerical ranges, while

other ranges remain unaffected. Consequently, standardized diagnostic tools such as EUD and NCCL tests, or even bitwise alignment tests tailored to match the structure of the target training job, cannot exhaustively cover all input conditions and may therefore fail to identify all faulty GPUs. As a result, machines that pass offline stress testing may still exhibit SDC during training.

Observation 3: Diverse Fault Patterns. Our analysis of production data (Table 2) shows that SDC incidents do not expose a uniform signature across faulty GPUs. Different incidents can surface in different computations and expose different observable evidence, making it difficult for any single checker to cover them all reliably. Although offline diagnostic tools (e.g., NVIDIA EUD) achieve a recall of approximately 70%, each run typically requires several hours for a GPU. Moreover, the inherent non-determinism and input-sensitive nature of SDC make it impossible for offline stress testing to exhaustively cover the computations and input regimes that arise in production. Consequently, neither offline pre-screening nor any single online checker can reliably preclude SDCs in large-scale GPU clusters.

3.2 Motivation

The three observations above have an important implication for system design. Observation 1 and Observation 2 show that SDCs cannot be reliably excluded by offline pre-screening, because whether a fault manifests depends on the exact execution and numerical regime encountered during real training. Observation 3 further shows that no single online checker can cover all incidents, because different faults surface in different computations and expose different forms of evidence.

Practical SDC detection therefore must satisfy three requirements. First, it must perform *lightweight in-situ sensing* on the critical path, because only online monitoring can observe the real computations and inputs that trigger SDCs. Second, it must support *multiple complementary sensing mechanisms* under a common interface, because different parts of the training stack expose different low-cost checking opportunities and produce different evidence. Third, it must provide *definitive confirmation* before reporting incidents, because lightweight online signals alone are not trustworthy enough in low-precision training. Together, these requirements suggest a sense-then-verify architecture that separates fast sensing from definitive confirmation while allowing multiple sensing strategies to share the same detection workflow. Realizing such a design in modern LLM training raises the following key challenges:

Challenge 1: Reliable Sensing under Low Precision. Low-precision data types such as `bfloat16` [20, 29] and even `float8` [23] are now standard in modern LLM training. In these regimes, ordinary floating-point round-off can obscure SDC-induced perturbations [27], causing naive checksum-based sensing to either flag too many benign computations as

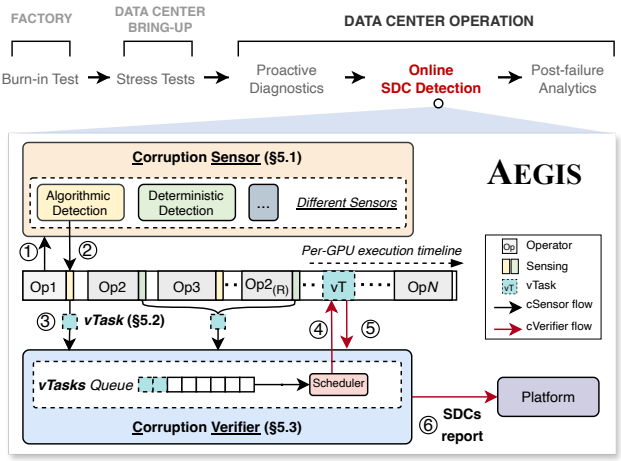


Figure 3: AEGIS Overview.

suspicious or miss real corruptions. The challenge is therefore to generate trustworthy online signals while keeping sensing lightweight on the critical path. The key question is whether modern training systems and hardware expose opportunities for reliable, low-cost sensing.

Challenge 2: Unifying Diverse Detection Mechanisms. Large-scale LLM training does not expose a single reusable checking pattern [21, 26]. Practical detection must instead combine multiple mechanism-specific sensing strategies that produce evidence with different semantics and confidence. The challenge is to unify these heterogeneous signals under a common abstraction so they can share the same buffering, scheduling, and confirmation workflow, rather than requiring custom system support for each new mechanism.

Challenge 3: Efficient Verification. Definitive confirmation is expensive in large-scale LLM training because establishing a trusted reference often requires replaying suspicious computations. Such replay consumes GPU cycles, can accumulate backlog, and may introduce stragglers [22] unless its cost is carefully bounded and scheduled. The challenge is therefore to keep verification off the critical path and scale it with suspicious-event volume without degrading end-to-end training throughput.

4 Overview

To address the above challenges, we propose AEGIS, a two-stage online SDC detection system for large-scale LLM training. As shown at the top of Figure 3, SDC detection is performed throughout the entire production and operational lifecycle of a GPU. AEGIS therefore targets the online stage with a sense-then-verify architecture that meets the three requirements identified in §3.2. Its *lightweight corruption Sensor* **cSensor** (§5.1) performs lightweight in-situ sensing on the critical path and unifies heterogeneous sensing outputs through the vTask abstraction (§5.2). Its *lazy corruption*

Verifier **cVerifier** (§5.3) performs definitive confirmation off the critical path. By decoupling fast sensing from expensive verification, AEGIS achieves low-overhead, high-coverage online protection without compromising training efficiency.

Workflow. Figure 3 illustrates AEGIS's workflow using a snapshot of the operator timeline on one GPU during an LLM training run. AEGIS instruments selected computations with cSensor (step ①). Immediately after a protected computation completes, cSensor applies the corresponding sensing strategy and records compact sensing evidence together with the minimal metadata needed for later verification (step ②). If cSensor flags a computation as potentially corrupted, it creates a verification task (vTask) and enqueues it into cVerifier 's vTask queue (step ③). cVerifier 's scheduler then dispatches vTasks to GPUs during naturally occurring idle periods to perform verification (step ④). cVerifier then performs definitive confirmation: depending on the vTask type, it either compares fingerprints from self-equivalent executions or replays the suspicious computation and checks it against the recorded evidence (step ⑤). When verification confirms an SDC, AEGIS reports the incident, together with the associated device and execution context, to the platform and users, enabling rapid restart and rollback to the previous checkpoint so training can resume with minimal wasted computation (step ⑥).

5 AEGIS Design

This section details how AEGIS realizes the cSensor-cVerifier architecture introduced in §4. The design relies on a system-level prerequisite: a protected computation must be deterministic, producing the same result under the same inputs and execution configuration. This prerequisite gives cVerifier a deterministic reference for confirmation, obtained either by deferred replay or by self-equivalent execution. cSensor therefore focuses on capturing compact evidence immediately after protected computations, while cVerifier performs definitive confirmation off the critical path. We first describe the inline sensing mechanisms, then the vTask interface, and finally the verifier and its scheduling policy.

5.1 cSensor: Corruption Sensor

cSensor is the inline sensing stage of the cSensor-cVerifier framework. Its goal is to observe real training computations at the moment SDCs can manifest, while retaining only the evidence needed for later confirmation. To address the reliable-sensing challenge under low precision, AEGIS exploits opportunities exposed by modern LLM training systems and accelerator hardware. Based on this idea, AEGIS instantiates cSensor with two complementary strategies: (1) mixed-precision-aware algorithmic detection and (2) self-equivalence-based deterministic detection. cSensor also applies sensing-control policies that decide when an inline signal becomes a vTask and bound the resulting verification load.

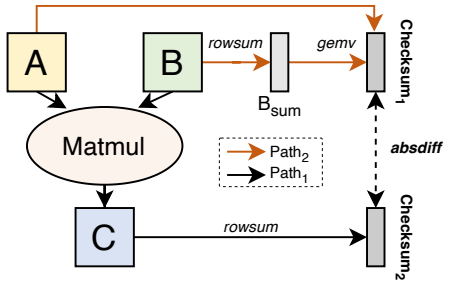


Figure 4: Classic matrix multiplication checksum workflow.

5.1.1 Mixed-precision-aware Algorithmic Detection

Algorithmic detection is AEGIS's answer to the low-precision reliability challenge: it keeps algebraic checks cheap enough for inline use by exploiting precision already present inside modern accelerator kernels. These sensors compress high-dimensional tensors into compact checksum values and compute a checksum difference Δ that indicates potential corruption. When Δ exceeds the current runtime tolerance estimate $\widehat{E}_t$, cSensor packages the suspicious computation into a vTask for later confirmation by cVerifier. We instantiate this approach for two dominant operators in LLM training. The key design point is to leverage *the mixed-precision design feature of modern accelerators* to overcome the floating-point error barrier in low-precision arithmetic.

Matrix Multiplication Checksum. AEGIS adopts the classic algorithmic scheme for matrix multiplication (`Matmul`), a key operation in LLM training. Specifically, this scheme validates `Matmul` by comparing two computationally equivalent paths encoded via checksums. As a simple example, we use a row checksum. Let $\mathbf{1}$ be a column vector of ones $[1,1,\ldots,1]^{\mathrm{T}}$; right-multiplying by $\mathbf{1}$ performs a row-wise sum. We obtain two checksum values: $C\mathbf{1} = (AB)\mathbf{1}$ and $A(B\mathbf{1})$. In the classic formulation, a significant mismatch is indicated when $\Delta_{\texttt{Matmul}} = \|(AB)\mathbf{1} - A(B\mathbf{1})\| > E$, where E is the theoretical round-off error bound [5]. A concrete example is illustrated in Figure 4.

However, algorithmic detection for `Matmul` has been deemed unreliable for low-precision LLM training (e.g., `bfloat16`) [27], as this method often suffers from large checksum differences caused by floating-point round-off noise. To better understand this issue, we conduct a fault injection study on a healthy machine. We consider a `Matmul` with `bfloat16` inputs and `bfloat16` outputs, and apply a row checksum over output C. Specifically, we inject faults by randomly flipping a single bit of the `bfloat16` input A.

Figure 5(a) shows the checksum differences obtained when checksums are accumulated from the final `bfloat16` outputs. Here, a checksum difference is the absolute difference between the two checksums, and the max discrepancy

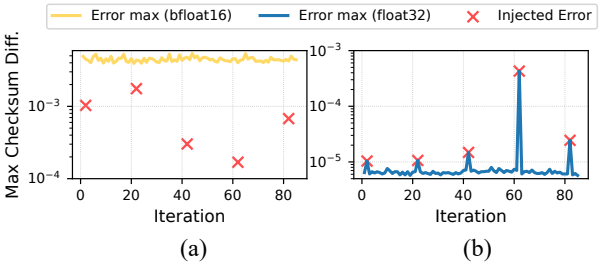


Figure 5: Fault injection study of checksum accumulation using `bfloat16` outputs versus `float32` accumulators.

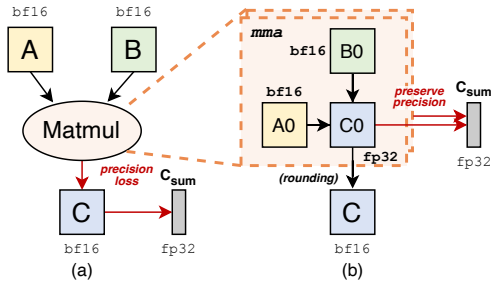


Figure 6: (a) Directly accumulating the final `bfloat16` output inherently introduces precision loss through truncation. (b) Accumulating directly from the `float32` accumulator preserves precision.

is the maximum over all such values. The red cross in the figure marks the fault-induced discrepancy caused by the injected fault. In Figure 5(a), it is clear that the fault-induced discrepancy is masked by larger round-off-induced discrepancies from `bfloat16` floating-point noise, making it difficult to distinguish a genuine fault from ordinary round-off noise.

To address this issue, our key insight is that `Matmul` operations in LLM training are typically executed using mixed-precision kernels (e.g., `bfloat16` compute paired with `float32` accumulation), a pattern inherent in modern accelerators (e.g., NVIDIA Tensor Core units). This creates an opportunity to compute the checksum of C in the `float32` accumulator, preserving higher precision and avoiding the final `bfloat16` truncation. In this way, we repeat the same fault injection experiment using the `float32` accumulator to validate this idea, as shown in Figure 5(b). The difference in checksum difference between (a) and (b) stems solely from round-off effects. Figure 5(b) clearly shows that using `float32` accumulator significantly reduces round-off-induced discrepancy, allowing each injected fault to produce a clear, pronounced peak that is easy to distinguish from the background numerical noise.

Based on this insight, AEGIS proposes a novel matrix multiplication checksum scheme tailored to low-precision computation. Specifically, AEGIS does not accumulate the result directly from the final `bfloat16` outputs, which would

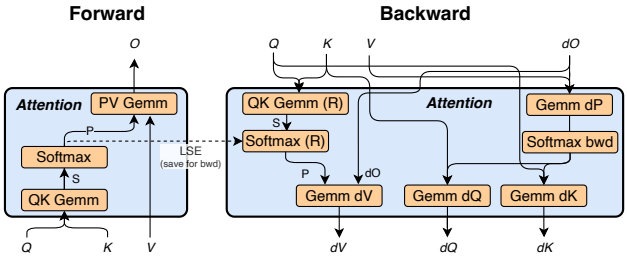


Figure 7: The forward and backward pass of FlashAttention.

inherently introduce precision loss through truncation from the kernel, as depicted in Figure 6(a). Instead, it leverages the mixed-precision execution provided by modern accelerators to accumulate checksums on-chip in `float32`, as shown in Figure 6(b). This higher precision accumulation significantly improves robustness to numerical noise, enabling reliable SDC detection for matrix multiplication in LLM training. In this case, the resulting vTask records the suspicious checksum together with the minimal replay context needed to re-execute the flagged computation during verification. The implementation details of the matrix multiplication checksum in AEGIS are described in §6.

FlashAttention Checksum. AEGIS proposes a novel algorithmic detection method for the attention operation, tailored to FlashAttention [10, 11, 35]. This method is grounded in an inherent algebraic property in the backward computation of dV. By exploiting this property, AEGIS can verify the correctness of the attention operation without performing checksum checks within each internal matrix multiplication. The forward and backward passes of FlashAttention are illustrated in Figure 7.

Specifically, the backward pass computes dV as:

$$\mathrm{d}V = P^{\mathrm{T}}\mathrm{d}O$$

Each row of the attention matrix P sums to 1 since it is the output of a row-wise softmax function, thus representing the attention distribution for each token. Consequently, its transpose P^T has columns that sum to 1.

Let $\mathbf{1}^T$ be a row vector of ones. By left-multiplying by $\mathbf{1}^T$ (which performs a column-wise sum), we observe:

$$\mathbf{1}^T\mathrm{d}V = \mathbf{1}^T(P^{\mathrm{T}}\mathrm{d}O) = (\mathbf{1}^T P^{\mathrm{T}})\mathrm{d}O$$

Given that P^T has columns that sum to 1, the identity $\mathbf{1}^T P^{\mathrm{T}} = \mathbf{1}^T$ holds. Therefore, we derive the mathematical invariant:

$$\mathbf{1}^T\mathrm{d}V = \mathbf{1}^T\mathrm{d}O$$

This relation asserts that the column-wise sum of the input gradient dO must equal the column-wise sum of the output gradient dV, and computing these sums incurs very low computational cost.

Furthermore, this novel checksum approach offers broad detection coverage. Beyond directly checking the invariant

itself, the invariant implicitly protects a larger portion of the computation. Because it is conditional on $\mathbf{1}^{\mathrm{T}}P^{\mathrm{T}} = \mathbf{1}^{\mathrm{T}}$, any error that violates this condition will, in turn, cause the assertion to fail. Specifically, the matrix P in the backward pass is recomputed within the `Softmax(R)` operator using a recomputed S (which is calculated by the `QKGemm(R)`) and the LSE saved from the forward pass. The LSE itself is computed across the forward `QKGemm` and `Softmax` operators. Therefore, an error in any of these preceding computations will violate the condition $\mathbf{1}^{\mathrm{T}}P^{\mathrm{T}} = \mathbf{1}^{\mathrm{T}}$ and consequently invalidate the invariant. Accordingly, AEGIS flags a potential SDC whenever $\Delta_{\text{attn}} = \|\mathbf{1}^{\mathrm{T}}\mathrm{d}V - \mathbf{1}^{\mathrm{T}}\mathrm{d}O\| > \widehat{E}_t$. In particular, the checksum in FlashAttention is also accumulated from the `float32` accumulators enabled by the mixed-precision feature of modern accelerators. As with `Matmul`, a violated invariant is converted into a vTask that carries the compact evidence and replay metadata required by cVerifier.

5.1.2 Self-equivalence-based Deterministic Detection

Self-equivalence-based deterministic detection exploits redundancy that already exists in LLM training. When a computation is executed twice deterministically, such as an original execution and a later recomputation, the two outputs should be bitwise identical. AEGIS captures this property by recording a compact xorsum fingerprint over each output during cSensor and attaching the fingerprints to vTasks. cVerifier then verifies the task by comparing the fingerprints from the two executions; any mismatch indicates an SDC. This differs from algorithmic detection, which reasons about approximate arithmetic through algebraic invariants, whereas deterministic detection relies on exact equality created by inherent recomputation.

Framework-level. Deterministic redundant computations are common in LLM training frameworks. For instance, activation recomputation [7, 37] mitigates the heavy memory footprint of activations by trading additional computation for memory footprint. These optimization techniques, while essential for the efficiency of large-scale LLM training, naturally create a detection opportunity.

Specifically, AEGIS computes each fingerprint by applying an xorsum reduction to compress high-dimensional tensors. As illustrated in Figure 8, the `Input` is processed by the operator (`Op`) during the forward pass to produce the initial output (`Output`), which is later recomputed as the recomputed output (`Output(R)`) in the backward pass. The detection mechanism computes two fingerprints, $\texttt{xorsum}_1$ and $\texttt{xorsum}_2$, by applying the `xorsum_reduce` function to the forward output `Output` and the recomputed output `Output(R)`, respectively. cVerifier (§5.3) subsequently compares $\texttt{xorsum}_1$ and $\texttt{xorsum}_2$, and any mismatch between the two fingerprints is confirmed by cVerifier as an SDC incident. Furthermore, this approach is applicable to a wide range of operators, extending beyond common operations such as `Matmul` and attention

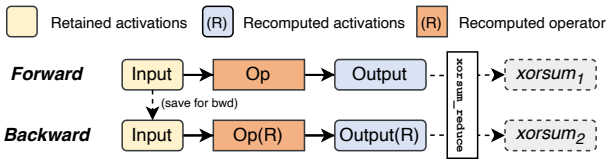


Figure 8: The mechanism of deterministic detection.

to fault-prone operators such as RMSNorm and GeLU, as detailed in §7.

Operator-level. Moreover, we identify and exploit similar inherent opportunities within individual operators. In AEGIS, we instantiate this detection mechanism in FlashAttention, and use it as a concrete example:

In FlashAttention, one primary optimization is selective recomputation in the backward pass, which avoids storing large intermediate matrices (S and P of size $N \times N$) from the forward pass, thereby reducing activation memory usage and I/O overhead (Figure 7). Specifically, S and P are recomputed when input blocks (Q, K, V) are loaded into SRAM. While this technique is fundamentally an optimization to improve computational efficiency and minimize memory footprint, it also creates a natural opportunity for deterministic self-checking. Under deterministic execution, the recomputed values (e.g., S) in backward must be bitwise identical to the original forward results. Accordingly, AEGIS applies an xorsum reduction in both the forward and backward passes to obtain two fingerprints for later comparison.

Deterministic detection complements algorithmic detection for attention. While it does not cover the $\mathrm{d}V = P^{\mathrm{T}}\mathrm{d}O$ path protected by our algorithmic method, it provides coverage for other self-equivalent segments. Together, the two methods offer complementary coverage for FlashAttention.

5.1.3 Sensing Control

Sensing control keeps cSensor useful as an online component rather than an unbounded generator of verification work. It decides when an inline signal is strong enough to become a vTask and how much evidence cSensor should retain for later confirmation. These policies bound verification workload while leaving the confirmation semantics of cVerifier unchanged.

Adaptive Threshold. For algorithmic detection, AEGIS employs a lightweight adaptive thresholding mechanism, rather than relying on the ideal tolerance E, to catch SDCs with small verification overhead, as illustrated in Figure 9. The core idea is to maintain a runtime tolerance estimate $\widehat{E}_t$ for the checksum difference Δ and dynamically calibrate it based on the historical distribution of Δ, which exhibits stability in training tasks. A key reason for this is that Δ, which essentially represents the numerical errors, is tightly correlated to the magnitude of the activations. Moreover, activation magnitudes do not change rapidly during training. Therefore, when the

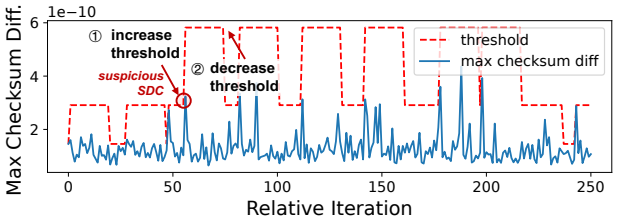


Figure 9: The adaptive thresholding mechanism.

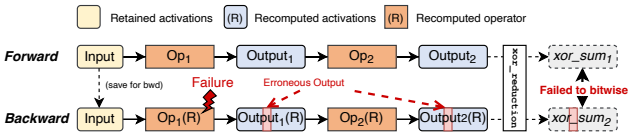


Figure 10: Example of backward recomputation and fingerprint mismatch under an operator fault.

observed Δ exceeds the current $\widehat{E}_t$ (indicating a potential SDC), AEGIS temporarily increases $\widehat{E}_t$ (e.g., by $2\times$) to adapt to the current error variance (step ①). Conversely, if no potential SDCs are detected over a subsequent window of checks, AEGIS decays $\widehat{E}_t$, gradually decreasing it (e.g., by halving) back toward its baseline (step ②). This heuristic yields a practical, low-overhead and adaptive mechanism that tolerates round-off fluctuations while still preserving reliable SDC detection.

Selective Operator Instrumentation. Under deterministic computation, repeated executions of the same computation graph always produce bitwise-identical results. Therefore, on a deterministic computation graph, any computational error in an operator inevitably leads to a different result. As illustrated in Figure 10, we consider a deterministic computation graph with two operators, $\texttt{Op}_1$ and $\texttt{Op}_2$, whose outputs are recomputed during the backward pass (denoted as $\texttt{Op}_1\,\texttt{(R)}$ and $\texttt{Op}_2\,\texttt{(R)}$). A fault in $\texttt{Op}_1\,\texttt{(R)}$ during the backward pass in turn corrupts the output of its successor $\texttt{Op}_2\,\texttt{(R)}$, causing the final fingerprint of $\texttt{Op}_2\,\texttt{(R)}$ to not be bitwise-identical with that of $\texttt{Op}_2$. Based on this insight, AEGIS's deterministic detection only focuses on the final operator in a deterministic computation graph, rather than instrumenting every operator individually. This policy reduces the number of vTasks generated by deterministic detection while preserving coverage of the entire recomputation chain.

5.2 vTask Abstraction

To address the challenge of unifying diverse detection mechanisms, AEGIS uses vTasks as the common contract between heterogeneous cSensor sensors and the shared cVerifier backend. Every suspicious computation that survives cSensor's local control is converted into a vTask that can be queued, scheduled, and processed uniformly by cVerifier. For algorithmic detection, a vTask records the minimal replay context and checksum evidence; for deterministic detection, it records

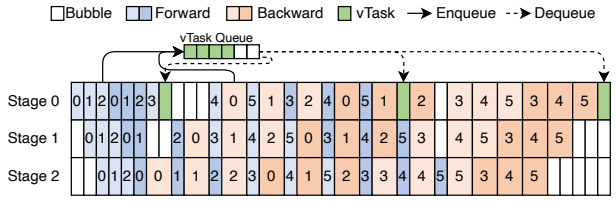


Figure 11: An example of vTask scheduling with interleaved 1F1B pipeline.

the two fingerprints for later cVerifier comparison. This unified abstraction lets AEGIS reuse one shared buffering and scheduling path across detectors. To minimize memory overhead, cSensor retains only the minimal payload required for cVerifier to perform verification, as detailed in §6. In addition to the minimal context required for verification, AEGIS also retains auxiliary metadata such as operator names, row/column indices, and verification type to facilitate scheduling and subsequent diagnosis.

5.3 cVerifier: Corruption Verifier

To address the efficient-verification challenge, AEGIS separates inline sensing from definitive confirmation. cVerifier consumes queued vTasks and turns inline evidence into confirmed SDC reports. Deterministic vTasks are checked by comparing fingerprints from self-equivalent executions, while algorithmic vTasks are checked by replaying the suspicious computation with the saved minimal context. An SDC is reported only after definitive confirmation is obtained, filtering out spurious alarms caused by benign numerical noise.

5.3.1 Lazy Verification and Scheduling

Lazy scheduling is the mechanism that makes definitive confirmation practical in production training. Because verification can be expensive, cVerifier treats vTasks as deferred work and executes them off-path by leveraging existing idle periods such as pipeline bubbles [16, 30].

As illustrated in Figure 11, we consider three-stage pipeline parallelism using the interleaved 1F1B pipeline schedule introduced in [31]. The cSensor creates and enqueues vTasks for both algorithmic and deterministic detection during the forward and backward passes. These vTasks are subsequently dequeued and processed by the cVerifier when workers are stalled waiting for tensor data from preceding or subsequent pipeline stages, i.e., during pipeline bubbles (shown as white cells in Figure 11). This scheduling strategy allows the cVerifier to perform the necessary verification without introducing additional synchronization stalls, thereby preserving the overall throughput of the LLM training process. In practice, scheduling follows a best-effort policy: vTasks are opportunistically executed during idle periods. Therefore, at the end of each training step, AEGIS reserves a small, fixed

time slice to process vTasks that could not be handled during idle time.

5.4 Supplementary Outlier Warning

Outlier warning provides a separate reporting channel for useful signals that would otherwise be ignored, as they are not yet strong enough to trigger a vTask. As discussed earlier, AEGIS uses a set of mechanisms to selectively verify only the most suspicious computations. This design balances coverage and performance, but may leave tiny SDC-induced deviations unconfirmed. To address this gap, AEGIS also performs an outlier-based warning analysis that is intentionally kept separate from definitive confirmation.

Specifically, AEGIS performs a cross-rank statistical analysis of the checksum difference at the end of each training step within each data-parallel (DP) group. The key insight is that, for the same batch and operator, the distribution of checksum differences across DP ranks should be relatively stable; a genuine SDC can manifest as an out-of-distribution deviation. To establish a collective baseline for each operation, AEGIS computes the $P90$ (90th percentile) of the checksum differences across all DP ranks. This statistic is collected via an efficient `allgather` over compact checksum-difference data, incurring only modest communication overhead.

If the checksum difference of a specific operation on a particular DP rank significantly exceeds the $P90$ baseline by a large margin (e.g., 10^6 times), AEGIS raises an outlier warning. Unlike confirmed SDC reports, such warnings are heuristic signals rather than definitive proof; they indicate a high likelihood of genuine SDCs that may have been missed by selective verification. Our production experience further confirms that outlier analysis is a practical and effective complement to the detection pipeline.

6 Implementation

AEGIS is integrated as a lightweight system on top of our in-house Megatron-LM [36] implementation. This integration requires minimal changes to the training pipeline and follows the proposed design to support online SDC detection for large-scale LLM training.

Verification Task Packing. Considering the severe memory constraints of LLM training, AEGIS retains only the minimal context required for verification. Specifically, only algorithm-based detection requires an additional recomputation step. Therefore, we describe the vTask packing design for algorithm-based detection in more detail.

For matrix multiplication, AEGIS adopts two checksum schemes: row checksums and row-and-column checksums. We apply them to different `Matmul` categories in LLM training, namely activation forward (*fwd*), input-gradient backward (*dgrad*), and weight-gradient backward (*wgrad*). For *fwd* and *dgrad*, AEGIS uses row checksums. The packing strategy is shown in Figure 12(a). In these cases, the input B corresponds to model weights that typically persist in GPU memory, whereas A and C are intermediate tensors that may be released before verification. Thus, the cSensor only needs to retain the row of A with the maximum numerical error and the corresponding row of C for later verification. For *wgrad*, AEGIS adopts row-and-column checksums to reduce the saved context, the packing strategy is shown in Figure 12(b). Since both inputs in *wgrad* are activations that may be released later by the framework, the cSensor retains the row of A and the column of B with the maximum numerical error, along with the corresponding elements of C needed for verification. For attention operation, verifying $\mathrm{d}V = P^{\mathrm{T}}\mathrm{d}O$ would require replaying the entire backward kernel and retaining additional intermediates (e.g., LSE, $\mathrm{d}O$, Q, and K).

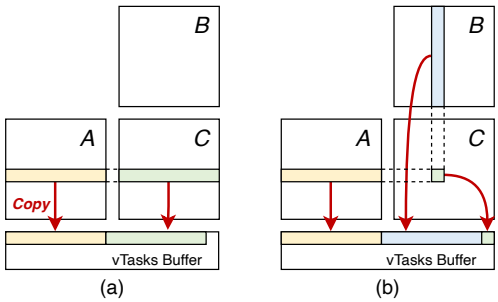


Figure 12: Different matrix multiplication checksum vTask packing strategy (a) row checksum and (b) row-and-column checksum.

Dynamic Sampling Strategy. In addition to the aforementioned mechanisms, we further employ a dynamic sampling strategy during the production deployment to reduce system overhead. Such a strategy is particularly practical for our algorithm-based detection mechanisms, as the computational overhead of checksum calculation in cSensor is non-negligible. Specifically, we assign a sampling rate to an operator, meaning that the operator will execute the checksum validation with a certain probability.

7 Evaluation

7.1 Experimental Setup

TestBed. All experiments are conducted on production GPU clusters. Each node is equipped with 8 GPUs unless otherwise specified.

Model Configurations. The configuration of the models used in our overhead analysis is listed in Table 1, detailing the number of layers, hidden size (h), number of attention heads, FFN intermediate size (h_{ffn}), number of experts, and top-k values.

Name	#layers	h	#heads	h_{ffn}	#experts	top-k
70B	24	4096	32	1024	256	8
268B	32	6144	48	2048	256	8

Table 1: Model configurations in evaluation.

7.2 AEGIS in Production

We first validate AEGIS's effectiveness in production through a deployment in our production clusters. During this time, AEGIS continuously monitors and detects SDCs in online large-scale LLM training jobs.

Detection Effectiveness. We first summarize the online SDC incidents detected by AEGIS. Over a deployment of 3.5×10^7 GPU-hours, AEGIS identified 18 SDC incidents involving 13 distinct faulty GPUs while incurring only 0.86% performance overhead. Among these incidents, only three incidents manifested as observable training failures (e.g., unexpected `NaN` loss); the rest were silent. This low overhead is achieved by enabling the dynamic sampling strategy described in §6, which balances detection coverage and performance in production.

Figure 13 plots the cumulative number of SDC incidents observed in our production environment after deploying AEGIS. For clarity, the figure shows only outlier warnings from GPUs that were repeatedly flagged and later confirmed faulty, reporting only the last warning for each such GPU. Initially, SDC incidents occur more frequently. Quantitatively, 13 SDC incidents were detected in the first half of the deployment period, whereas only 5 incidents occurred in the latter half. As faulty GPUs are detected and removed from the resource pool, the rate of SDC incidents gradually decreases. This marked reduction in SDC rate demonstrates the critical necessity of deploying online SDC detection in large-scale LLM training to protect training reliability and also validates the effectiveness of the AEGIS. Outlier warnings also serve as an early signal of latent risks; for example, a warning from a specific machine in Figure 13 was subsequently confirmed as a genuine SDC by AEGIS. We discuss outlier warning, false negative, and false positive problems in more detail in the discussion section (§8).

Coverage. The deterministic detection method detects more SDC incidents (12 times) than algorithmic detection and its outlier warnings (6 times). This observation can be attributed to two main factors: First, the broad operator coverage of deterministic approaches extends beyond `Matmul` and attention to include other components such as normalizations. Second, algorithmic detection is only sparingly sampled in production tasks to minimize overhead. Crucially, algorithmic detection remains essential as it offers a distinct detection scope and hence targets different SDCs. In particular, in training scenarios where recomputation is limited, algorithmic detection provides effective protection for computationally intensive operators that dominate the overall runtime. This

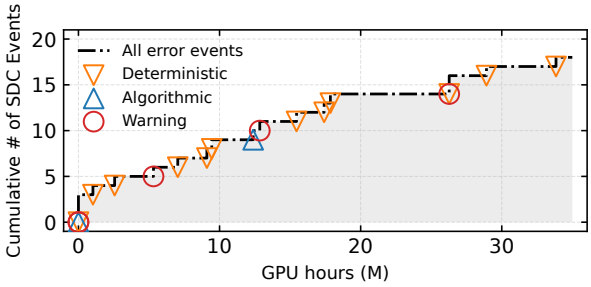


Figure 13: Cumulative SDC count over time (in million GPU-hours) in the production environment, as detected by AEGIS.

complementarity further justifies AEGIS's multi-sensor design for maximizing SDC detection coverage in production.

7.3 Analysis of Detection Capability

We conduct a capability study on eight available machines that had previously exhibited SDCs, as detailed in Table 2. For each machine, we run the training task 10 times to account for the inherent non-determinism of SDC manifestation during LLM training.

Compared with Offline Method. We compare AEGIS against the vendor-provided offline diagnostics tests. As shown in Table 2, AEGIS achieves a recall of 8 out of 8 faulty machines. In contrast, thorough diagnostic tests, which require hours to complete, detect only 25% of them (2 out of 8). We note that this recall is lower than the 70% reported in [41]. We attribute this discrepancy primarily to pre-screening bias in our operational environment. Prior to production training, all GPUs undergo strict offline stress testing to filter out visibly unhealthy machines. These eight GPUs were initially certified as healthy and then admitted into the LLM training cluster. Therefore, the low recall of offline stress testing underscores the difficulty of detecting such silent corruptions with conventional methods. This result highlights the practical value of our online approach in identifying hard-to-detect SDCs that escape offline screening.

Different Method Recall. Table 2 also breaks down the recall of individual methods across the eight faulty machines. A key observation is that no single method achieved 100% recall across all machines, indicating that online SDC detection in large-scale LLM training requires multiple complementary methods. This phenomenon aligns with our analysis in §7.2.

7.4 Overhead

We then evaluate the runtime overhead during training. We conduct experiments using two model sizes, 70B and 268B (as illustrated in Table 1), and scale up to 1,024 GPUs. Unless otherwise stated, the sequence length and global batch size are fixed at 8,192 and 768, respectively. We compare the per-iteration training time of AEGIS against a baseline without

ID	Fault Type / Description	Vendor-provided Diagnostics Tests	Algorithmic Detection		Deterministic Detection	
			Matmul	Attention	Framework	Attention
1	RMSNorm execution fault				✓	✓
2	`Matmul` execution fault on EU `id=81`	✓	✓			
3	`wmma` instruction execution fault on EU `id=94`		✓			
4	HBM memory error (single-bit flips)				✓	✓
5	`Matmul` execution fault	✓	✓	✓	✓	✓
6	FlashAttention execution fault				✓	✓
7	FlashAttention execution fault					✓
8	FlashAttention execution fault		✓			✓

Table 2: Summary of SDC detection capability of AEGIS and vendor-provided diagnostics tests on eight real-world faulty machines, with detailed fault descriptions.

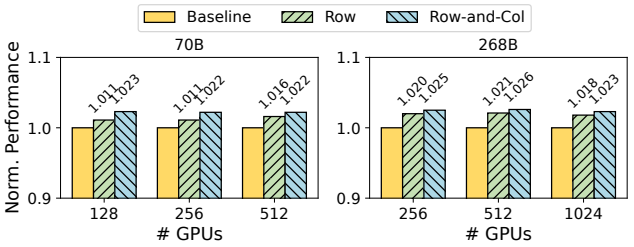


Figure 14: Overhead of AEGIS with different checksum techniques.

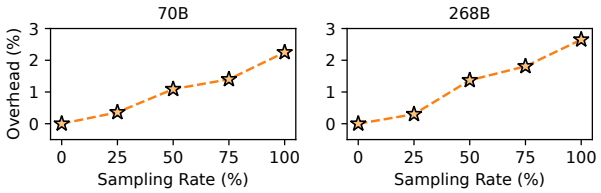


Figure 15: Overhead of different dynamic sampling rate.

SDC detection enabled to demonstrate its efficiency. All reported performance numbers are averaged over 10 stable iterations to ensure measurement reliability.

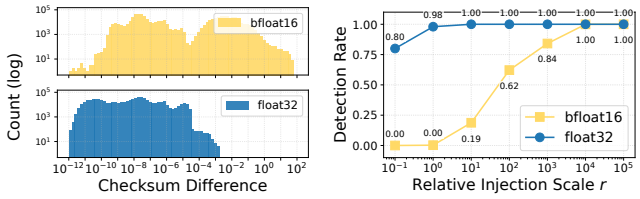
End-to-end Overhead. We evaluate two AEGIS configurations. The first enables only row checksums; in this configuration, *wgrad* is not fully covered. The second enables row-and-column checksums, providing coverage for all `Matmul` categories in LLM training. In both configurations, deterministic detection is enabled. Figure 14 summarizes the runtime overhead of AEGIS. Across all tested workloads, AEGIS incurs low overhead throughout training, averaging below 1.61% with row checksums and 2.35% with row-and-column checksums. The maximum slowdown is 2.1% for row checksums and 2.6% for row-and-column checksums. The primary sources of overhead can be divided into two stages: cSensor and cVerifier. In the cSensor stage, overhead mainly comes from checksum computation and minor memory operations needed to retain the minimal context for potential replay. In the cVerifier stage, overhead comes from executing vTasks to confirm whether it is a genuine SDC. The higher overhead of row-and-column checksums is primarily due to the extra memory copies (the column of input B) required to retain verification context compared to row checksums. In return, this configuration enables SDC verification for *wgrad*, trading minor overhead for broader coverage.

The results also indicate that AEGIS scales well to 1,024 GPUs and to larger model sizes. Because AEGIS does not introduce additional communication, it remains effective as cluster size grows. We expect this design to scale to larger deployments, providing reliable SDC detection for future ultra-large-scale LLM training with minimal impact on throughput.

Dynamic Sampling. To evaluate AEGIS 's dynamic sampling strategy (§6) and its impact on overhead, we conduct an ablation study under settings that mirror our end-to-end evaluation, except that both the 70B and 268B models are run on 128 GPUs. As shown in Figure 15, varying the sampling rate provides a simple and effective way to tune AEGIS 's overhead, which scales approximately linearly with the sampling rate. In our production deployment, we use this strategy to select a sweet spot that balances overhead and online SDC detection capability. Specifically, AEGIS controls the sampling rate of algorithmic detection to meet a target overhead budget (e.g., 0.86% overhead in our production).

7.5 Ablation Study

Checksum Difference. We next evaluate the contribution of algorithmic detection during training, focusing on how checksum accumulation precision affects detection sensitivity. Because operators differ in output scale and numerical-noise profile, we collect checksum-difference distributions for `bfloat16` and `float32` accumulation across operators over 500 training iterations without injected faults. As shown



(a) (b)

Figure 16: Ablation study of checksum precision in algorithmic detection. (a) Checksum difference distribution. (b) Detection rate vs. relative injection scale.

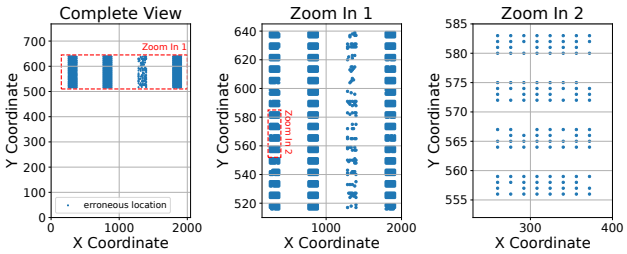


Figure 17: The erroneous locations of output tensor.

in Figure 16(a), `float32` produces substantially smaller checksum differences, making it more sensitive to small SDC-induced deviations.

Detection Rate. We then perform a calibrated fault-injection analysis. Because operators have different checksum-difference scales, injecting the same absolute perturbation, or an uncalibrated random fault such as a bit flip, would confound the comparison across operators. We therefore normalize each injected perturbation by the corresponding operator's `float32` checksum-difference scale. For each operator, we use the median `float32` checksum difference from this 500-iteration run as its reference scale. For a relative injection scale r, we inject a fault into a randomly selected element during the operator computation with probability 0.02%. The faulty element receives a signed perturbation with magnitude $r \cdot m_o$, where m_o is the corresponding operator's median `float32` checksum difference. Each setting is repeated 100 times. Figure 16(b) shows that `float32` checksums detect injected faults at the median reference scale with a near-100% detection rate. In contrast, `bfloat16` checksums remain largely insensitive at this scale and require roughly $10^4\times$ larger perturbations to reach comparable detection. This gap demonstrates that the higher-precision checksum design of our method provides stronger detection capability and enables finer-grained error detection.

7.6 Case Studies

Case 1: Detection of Matrix Unit Fault. In this case, the faulty GPU was detected during a matrix multiplication operation (corresponding to ID 3 in Table 2). We reproduced the SDC fault and analyzed its root cause.

We locate erroneous positions by comparing the faulty GPU's output with the expected results, as shown in Figure 17. The output tensor from the faulty GPU, of size 768×2048, exhibits errors only at specific positions with a regular pattern, as illustrated on the right. Zooming in on this region reveals a clearer structure, with errors recurring at periodic intervals along both the x and y axes (middle). Further zooming in (left) shows that the errors predominantly appear as four consecutive incorrect elements within localized regions.

Given this spatial clustering and regular structure, we hypothesize that the SDC originates from a specific execution unit. By inserting the assembly instruction to retrieve the `id` at erroneous locations, we identify the faulty execution unit (EU), which in our experiments consistently reports `id=94`. For identical inputs, we observe computation errors only when the Matrix Multiply-Accumulate instruction (`mma`, e.g., `wmma_f32_16x16x16_bf16_w32`) is executed on the matrix cores of this EU, whereas the same inputs produce correct results on the vector cores within the same EU. These observations indicate that the SDC is localized to the matrix unit and is specifically triggered by `mma` execution.

This case underscores the importance of integrating SDC detection into matrix multiplication, given the heavy reliance of LLM training on matrix units for efficient acceleration across the execution stack.

Case 2: Detection of HBM Fault. Our second case involves an SDC caused by an HBM fault (corresponding to ID 4 in Table 2). This SDC was detected by our deterministic detection. In this case, a `float32` intermediate value was saved during the forward pass and later read in the backward pass. Its value changed by a one-bit flip, from `10111111`**`0`**`11010100100000111110111` to `10111111`**`1`**`11010100100000111110111`. This single-bit error causes the fingerprint of the recomputed result to be not bitwise-aligned with that of the original forward pass, and is therefore detected by AEGIS. An interesting observation is that this fault does not trigger the ECC protection mechanism, suggesting a potential ECC failure that exposes GPU memory to the risk of undetected errors.

8 Discussions and Experiences

Permanent vs. Transient SDC Incidents. SDC incidents can be categorized as permanent or transient. Permanent SDCs stem from persistent hardware faults, whereas transient SDCs are rare, occasional events on healthy hardware, often attributed to external factors such as cosmic rays [28] and temperature [43]. In our production environment, all SDC incidents detected by AEGIS were reproducible on the corresponding GPUs. The time between re-triggered incidents during training varied widely, ranging from seconds to days. We further performed a reboot experiment on one such GPU. Although the GPU had previously been flagged

for a VRAM fault by the diagnostic tool, it passed all offline test suites immediately after a system reboot, suggesting that rebooting can temporarily mitigate SDC symptoms. However, one month later, the SDC reappeared; AEGIS detected it, and the diagnostic tool again reported the same underlying VRAM fault, indicating that *reboot does not resolve the underlying hardware faults*. These findings are similar with our in-house study [46], indicating that the observed SDC incidents predominantly stem from persistent hardware faults rather than sporadic transient events. Therefore, online SDC detection provides an efficient mechanism to identify and isolate faulty GPUs, preventing long-term degradation of LLM training caused by underlying hardware issues.

Software-induced SDCs. The correctness of AEGIS relies critically on bitwise comparison during computation replay. This property, however, can be violated not only by hardware faults but also by software bugs. During our deployment, AEGIS caught an SDC-like incident caused by a training framework bug. Specifically, the framework read illegal input data, which led to an out-of-bounds write in a kernel and triggered a recomputation mismatch alarm. This failure case highlights an additional benefit of AEGIS integrated into the training framework stack. Beyond hardware-induced SDCs typically targeted by offline testing, AEGIS can also identify software bugs within the training framework that manifest as SDCs, thereby offering further protection for training jobs.

False Negatives. To detect SDCs escaping from AEGIS, we screen for SDCs through several other methods. We periodically performed task-level replays using bitwise-deterministic, cluster-level training tasks to probe for undiscovered SDC issues. Across repeated runs, the losses remained aligned, providing additional confidence that no faulty GPUs were missed during the deployment. We also analyze unexpected task failures, including segmentation faults, `NaN` and `inf` values in the training task. Only one additional faulty GPU is detected, which caused an SDC that occurred outside the backbone of the Transformer network and hence beyond the protection region of AEGIS. This retrospective analysis demonstrates that AEGIS can detect most SDC incidents in LLM training, indicating its effectiveness. Nevertheless, we cannot completely rule out undetected SDCs, given their potentially low frequency and the inherent difficulty of observing silent corruptions in production LLM training.

Operator Coverage. The current AEGIS prototype focuses on dominant compute-intensive Transformer kernels, including `Matmul` and FlashAttention, while some lightweight or outside-backbone patterns, such as standalone element-wise operations, remain uninstrumented. This gap explains the remaining production false negative; extending coverage mainly requires adding the corresponding operator-specific sensor to the existing detection pipeline.

False Positives. Beyond these confirmed SDC reports, we observed 21 warning alerts from the outlier-based analysis (§5.4) during deployment. Among them, 11 alerts were attributed to reproducible faulty GPUs. Six alerts were repeatedly reported by a highly suspicious GPU over one month, yet could not be reproduced by existing offline diagnostic tools, highlighting the limitations of offline testing for identifying hard-to-reproduce faults. Three alerts in the remainder occurred on different GPUs during the model warm-up phase, suggesting that outlier-based numerical checks can be sensitive to out-of-distribution values in early LLM training.

Observing Only Compute- and Memory-Induced SDCs. In our deployment, we reproduced SDC incidents attributable to computation and memory, but did not observe communication-induced SDCs. A plausible explanation is that the communication stack is protected by multiple layers of error detection and correction (e.g., ECC and link-level checks), whereas compute paths on accelerators expose fewer built-in, end-to-end integrity checks. This asymmetry reinforces the importance of integrating lightweight SDC detection into compute-intensive operators in LLM training.

High Precision Accumulation Enables SDC Localization. During deployment, we did not successfully locate SDC incidents using checksums accumulated from `bfloat16` outputs. Instead, all confirmed detections were achieved by accumulating checksums from `float32` accumulators.

9 Related Work

SDC Detection in Training Systems. Existing research has explored various techniques to detect and mitigate the impact of SDC in AI training systems. [27] analyzes the effect of SDC in LLM training. ATTNChecker [21] uses ABFT to provide fault tolerance for the attention mechanism. Dr.DNA [26] proposes early-stage SDC detection and mitigation based on activation distributions. Gemini 2.5 [9] uses numerical analysis and rerun to identify suspicious SDCs. LongCat [38] introduces additional recomputation to verify the correctness of attention backward computation. However, these approaches often suffer from high overhead or limited detection efficacy. In contrast, AEGIS offers practical online SDC detection with low overhead and broad coverage for production-scale LLM training.

Fault-tolerant LLM Training. Some orthogonal efforts enhance the fault-tolerance of LLM training. GEMINI [44] and ByteCheckpoint [40] optimize checkpointing for faster recovery and reduced stall time. Other systems [14, 18, 39] exploit training redundancy to tolerate failures without disrupting training. Yet they do not study or address SDC issues. In contrast, AEGIS provides online SDC detection that promptly flags errors, which can then be combined with these fault-tolerance mechanisms to perform rapid rollbacks when necessary, thereby complementing these techniques.

SDC Detection in Other Applications. In other applications and hardware, [4] provides ultralight, application-agnostic

data dynamic monitoring to detect SDC in extreme-scale MPI. [12] studies SDC incidence and survey hardware/software mitigation strategies; Farron [43] mitigates CPU SDCs via prioritized testing and temperature control; Orthrus [24] detects SDCs in the cloud through resource-adaptive validation; and Hardware Sentinel [13] monitors large data-center fleets using software failure indicators. However, these CPU- and application-centric techniques do not directly address the distinctive computational and communication characteristics of GPU-accelerated LLM training; while AEGIS provides a specialized solution tailored to this setting.

10 Conclusion

We present the online SDC detection system AEGIS for large-scale LLM training, along with insights gleaned from 35 million GPU hours. Our findings demonstrate that online SDC detection should be established as a standard component of training frameworks to safeguard the scaling of LLM training.

11 Acknowledgment

We sincerely thank the anonymous reviewers and our shepherd for their valuable feedback on this paper. This work is supported by New Generation Artificial Intelligence-National Science and Technology Major Project under Grant 2025ZD0123801, NSFC for Distinguished Young Scholar under Grant 62225206, National Natural Science Foundation of China under Grants 62532006, 62495062, 62302251, and Beijing Natural Science Foundation under Grant L242017. Jidong Zhai and Xin Liu are the corresponding authors of this paper.

References

[1] nanoGPT. https://github.com/karpathy/nanoGPT.

[2] Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, et al. Gpt-4 technical report. *arXiv preprint arXiv:2303.08774*, 2023.

[3] AMD. ROCm validation suite (RVS), 2025. https://rocm.docs.amd.com/projects/ROCmValidationSuite.

[4] Leonardo Bautista-Gomez and Franck Cappello. Detecting silent data corruption for extreme-scale mpi applications. In *Proceedings of the 22nd European MPI Users' Group Meeting*, pages 1–10, 2015.

[5] Jieyang Chen, Hongbo Li, Sihuan Li, Xin Liang, Panruo Wu, Dingwen Tao, Kaiming Ouyang, Yuanlai Liu, Kai Zhao, Qiang Guan, and Zizhong Chen. Fault tolerant one-sided matrix decompositions on heterogeneous systems with gpus. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis*, SC '18. IEEE Press, 2019.

[6] Jieyang Chen, Xin Liang, and Zizhong Chen. Online algorithm-based fault tolerance for cholesky decomposition on heterogeneous systems with gpus. In *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 993–1002. IEEE, 2016.

[7] Tianqi Chen, Bing Xu, Chiyuan Zhang, and Carlos Guestrin. Training deep nets with sublinear memory cost. *arXiv preprint arXiv:1604.06174*, 2016.

[8] Zizhong Chen and Jack Dongarra. Algorithm-based fault tolerance for fail-stop failures. *IEEE Transactions on Parallel and Distributed Systems*, 19(12):1628–1641, 2008.

[9] Gheorghe Comanici, Eric Bieber, Mike Schaekermann, Ice Pasupat, Noveen Sachdeva, Inderjit Dhillon, Marcel Blistein, Ori Ram, Dan Zhang, Evan Rosen, et al. Gemini 2.5: Pushing the frontier with advanced reasoning, multimodality, long context, and next generation agentic capabilities. *arXiv preprint arXiv:2507.06261*, 2025.

[10] Tri Dao. Flashattention-2: Faster attention with better parallelism and work partitioning. *arXiv preprint arXiv:2307.08691*, 2023.

[11] Tri Dao, Dan Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. Flashattention: Fast and memory-efficient exact attention with io-awareness. *Advances in neural information processing systems*, 35:16344–16359, 2022.

[12] Harish Dattatraya Dixit, Sneha Pendharkar, Matt Beadon, Chris Mason, Tejasvi Chakravarthy, Bharath Muthiah, and Sriram Sankar. Silent data corruptions at scale. *arXiv preprint arXiv:2102.11245*, 2021.

[13] Rhea Dutta, Harish Dattatraya Dixit, Rik Van Riel, Gautham Vunnam, and Sriram Sankar. Hardware sentinel: Protecting software applications from hardware silent data corruptions. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, pages 482–497, 2025.

[14] Swapnil Gandhi, Mark Zhao, Athinagoras Skiadopoulos, and Christos Kozyrakis. Recycle: Resilient training of large dnns using pipeline adaptation. In *Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles*, pages 211–228, 2024.

[15] Aaron Grattafiori, Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Alex Vaughan, et al. The llama 3 herd of models. *arXiv preprint arXiv:2407.21783*, 2024.

[16] Yanping Huang, Youlong Cheng, Ankur Bapna, Orhan Firat, Dehao Chen, Mia Chen, HyoukJoong Lee, Jiquan Ngiam, Quoc V Le, Yonghui Wu, et al. Gpipe: Efficient training of giant neural networks using pipeline parallelism. *Advances in neural information processing systems*, 32, 2019.

[17] Sam Ade Jacobs, Masahiro Tanaka, Chengming Zhang, Minjia Zhang, Shuaiwen Leon Song, Samyam Rajbhandari, and Yuxiong He. Deepspeed ulysses: System optimizations for enabling training of extreme long sequence transformer models. *arXiv preprint arXiv:2309.14509*, 2023.

[18] Insu Jang, Zhenning Yang, Zhen Zhang, Xin Jin, and Mosharaf Chowdhury. Oobleck: Resilient distributed training of large models using pipeline templates. In *Proceedings of the 29th Symposium on Operating Systems Principles*, pages 382–395, 2023.

[19] Ziheng Jiang, Haibin Lin, Yinmin Zhong, Qi Huang, Yangrui Chen, Zhi Zhang, Yanghua Peng, Xiang Li, Cong Xie, Shibiao Nong, Yulu Jia, Sun He, Hongmin Chen, Zhihao Bai, Qi Hou, Shipeng Yan, Ding Zhou, Yiyao Sheng, Zhuo Jiang, Haohan Xu, Haoran Wei, Zhang Zhang, Pengfei Nie, Leqi Zou, Sida Zhao, Liang Xiang, Zherui Liu, Zhe Li, Xiaoying Jia, Jianxi Ye, Xin Jin, and Xin Liu. MegaScale: Scaling large language model training to more than 10,000 GPUs. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*, pages 745–760, Santa Clara, CA, April 2024. USENIX Association.

[20] Dhiraj Kalamkar, Dheevatsa Mudigere, Naveen Mellempudi, Dipankar Das, Kunal Banerjee, Sasikanth Avancha, Dharma Teja Vooturi, Nataraj Jammalamadaka, Jianyu Huang, Hector Yuen, et al. A study of BFLOAT16 for deep learning training. *arXiv preprint arXiv:1905.12322*, 2019.

[21] Yuhang Liang, Xinyi Li, Jie Ren, Ang Li, Bo Fang, and Jieyang Chen. Attnchecker: Highly-optimized fault tolerant attention for large language model training. In *Proceedings of the 30th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming*, pages 252–266, 2025.

[22] Jinkun Lin, Ziheng Jiang, Zuquan Song, Sida Zhao, Menghan Yu, Zhanghan Wang, Chenyuan Wang, Zuocheng Shi, Xiang Shi, Wei Jia, Zherui Liu, Shuguang Wang, Haibin Lin, Xin Liu, Aurojit Panda, and Jinyang Li. Understanding stragglers in large model training using what-if analysis. In *19th USENIX Symposium on Operating Systems Design and Implementation (OSDI 25)*, pages 483–498, Boston, MA, July 2025. USENIX Association.

[23] Aixin Liu, Bei Feng, Bing Xue, Bingxuan Wang, Bochao Wu, Chengda Lu, Chenggang Zhao, Chengqi Deng, Chenyu Zhang, Chong Ruan, et al. Deepseek-v3 technical report. *arXiv preprint arXiv:2412.19437*, 2024.

[24] Chenxiao Liu, Zhenting Zhu, Quanxi Li, Yanwen Xia, Yifan Qiao, Xiangyun Deng, Youyou Lu, Tao Xie, Huimin Cui, Zidong Du, et al. Orthrus: Efficient and timely detection of silent user data corruption in the cloud with resource-adaptive computation validation. In *Proceedings of the ACM SIGOPS 31st Symposium on Operating Systems Principles*, pages 286–304, 2025.

[25] Hao Liu, Matei Zaharia, and Pieter Abbeel. Ring attention with blockwise transformers for near-infinite context. *arXiv preprint arXiv:2310.01889*, 2023.

[26] Dongning Ma, Fred Lin, Alban Desmaison, Joel Coburn, Daniel Moore, Sriram Sankar, and Xun Jiao. Dr. dna: Combating silent data corruptions in deep learning using distribution of neuron activations. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, pages 239–252, 2024.

[27] Jeffrey Ma, Hengzhi Pei, Leonard Lausen, and George Karypis. Understanding silent data corruption in llm training. *arXiv preprint arXiv:2502.12340*, 2025.

[28] Sarah E Michalak, Andrew J DuBois, Curtis B Storlie, Heather M Quinn, William N Rust, David H DuBois, David G Modl, Andrea Manuzzato, and Sean P Blanchard. Assessment of the impact of cosmic-ray-induced neutrons on hardware in the roadrunner supercomputer. *IEEE Transactions on Device and Materials Reliability*, 12(2):445–454, 2012.

[29] Paulius Micikevicius, Sharan Narang, Jonah Alben, Gregory Diamos, Erich Elsen, David Garcia, Boris Ginsburg, Michael Houston, Oleksii Kuchaiev, Ganesh Venkatesh, and Hao Wu. Mixed precision training. In *International Conference on Learning Representations*, 2018.

[30] Deepak Narayanan, Aaron Harlap, Amar Phanishayee, Vivek Seshadri, Nikhil R Devanur, Gregory R Ganger, Phillip B Gibbons, and Matei Zaharia. Pipedream: Generalized pipeline parallelism for dnn training. In *Proceedings of the 27th ACM symposium on operating systems principles*, pages 1–15, 2019.

[31] Deepak Narayanan, Mohammad Shoeybi, Jared Casper, Patrick LeGresley, Mostofa Patwary, Vijay Korthikanti, Dmitri Vainbrand, Prethvi Kashinkunti, Julie Bernauer, Bryan Catanzaro, Amar Phanishayee, and Matei Zaharia. Efficient large-scale language model training on gpu clusters using megatron-lm. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '21, New York, NY, USA, 2021. Association for Computing Machinery.

[32] NVIDIA. Extended utility diagnostics (EUD), 2025. https://docs.nvidia.com/datacenter/dcgm/latest/user-guide/dcgm-eud.html.

[33] Penghui Qi, Xinyi Wan, Guangxing Huang, and Min Lin. Zero bubble (almost) pipeline parallelism. In *The Twelfth International Conference on Learning Representations*, 2024.

[34] Samyam Rajbhandari, Jeff Rasley, Olatunji Ruwase, and Yuxiong He. Zero: memory optimizations toward training trillion parameter models. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '20. IEEE Press, 2020.

[35] Jay Shah, Ganesh Bikshandi, Ying Zhang, Vijay Thakkar, Pradeep Ramani, and Tri Dao. Flashattention-3: Fast and accurate attention with asynchrony and low-precision. *Advances in Neural Information Processing Systems*, 37:68658–68685, 2024.

[36] Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. Megatron-lm: Training multi-billion parameter language models using model parallelism. *arXiv preprint arXiv:1909.08053*, 2019.

[37] Zhenbo Sun, Huanqi Cao, Yuanwei Wang, Guanyu Feng, Shengqi Chen, Haojie Wang, and Wenguang Chen. Adapipe: Optimizing pipeline parallelism with adaptive recomputation and partitioning. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, pages 86–100, 2024.

[38] Meituan LongCat Team, Bei Li, Bingye Lei, Bo Wang, Bolin Rong, Chao Wang, Chao Zhang, Chen Gao, Chen Zhang, Cheng Sun, et al. Longcat-flash technical report. *arXiv preprint arXiv:2509.01322*, 2025.

[39] John Thorpe, Pengzhan Zhao, Jonathan Eyolfson, Yifan Qiao, Zhihao Jia, Minjia Zhang, Ravi Netravali, and Guoqing Harry Xu. Bamboo: Making preemptible instances resilient for affordable training of large DNNs. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 497–513, Boston, MA, April 2023. USENIX Association.

[40] Borui Wan, Mingji Han, Yiyao Sheng, Yanghua Peng, Haibin Lin, Mofan Zhang, Zhichao Lai, Menghan Yu, Junda Zhang, Zuquan Song, Xin Liu, and Chuan Wu. ByteCheckpoint: A unified checkpointing system for large foundation model development. In *22nd USENIX Symposium on Networked Systems Design and Implementation (NSDI 25)*, pages 559–578, Philadelphia, PA, April 2025. USENIX Association.

[41] Borui Wan, Gaohong Liu, Zuquan Song, Jun Wang, Yun Zhang, Guangming Sheng, Shuguang Wang, Houmin Wei, Chenyuan Wang, Weiqiang Lou, et al. Robust llm training infrastructure at bytedance. In *Proceedings of the ACM SIGOPS 31st Symposium on Operating Systems Principles*, page 186–203, 2025.

[42] Linnan Wang, Jinmian Ye, Yiyang Zhao, Wei Wu, Ang Li, Shuaiwen Leon Song, Zenglin Xu, and Tim Kraska. Superneurons: Dynamic gpu memory management for training deep neural networks. In *Proceedings of the 23rd ACM SIGPLAN symposium on principles and practice of parallel programming*, pages 41–53, 2018.

[43] Shaobu Wang, Guangyan Zhang, Junyu Wei, Yang Wang, Jiesheng Wu, and Qingchao Luo. Understanding silent data corruptions in a large production cpu population. In *Proceedings of the 29th Symposium on Operating Systems Principles*, pages 216–230, 2023.

[44] Zhuang Wang, Zhen Jia, Shuai Zheng, Zhen Zhang, Xinwei Fu, TS Eugene Ng, and Yida Wang. Gemini: Fast failure recovery in distributed training with in-memory checkpoints. In *Proceedings of the 29th Symposium on Operating Systems Principles*, pages 364–381, 2023.

[45] Kai Zhao, Sheng Di, Sihuan Li, Xin Liang, Yujia Zhai, Jieyang Chen, Kaiming Ouyang, Franck Cappello, and Zizhong Chen. Ft-cnn: Algorithm-based fault tolerance for convolutional neural networks. *IEEE Transactions on Parallel and Distributed Systems*, 32(7):1677–1689, 2020.

[46] Wenxin Zheng, Wenxiao Wang, Yun Zhang, Mingcong Han, Bin Xu, Jinyu Gu, Xingda Wei, Haibo Chen, Zuquan Song, Gaohong Liu, Yucheng Nie, Zhe Nan, Zhuolin Zheng, Huan Yu, Shuguang Wang, Ziming Zhou, Hang Zhu, Wencong Xiao, and Xin Liu. SDCs in the wild: Characterizing and diagnosing SDC-defective GPUs in production LLM training. In *20th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2026, Seattle, WA, USA, July 13–15, 2026*. USENIX Association, 2026.

OpGuard: Bitwise Alignment for Precise and General Debugging of Production LLM Training

Ziming Zhou[1], Yinjie Zhao[1], Hang Zhu[2], Wenxiao Wang[2], Zhihao Bai[2], Yun Zhang[2], Shuguang Wang[2], Haibin Lin[2], Peng Huang[1]

[1]University of Michigan [2]ByteDance Seed

Abstract

Large-scale LLM training runs on many GPUs for weeks atop rapidly evolving software stacks. Bugs or hardware glitches can silently corrupt the computation and only surface much later. Debugging becomes finding a needle in a haystack across time. Developers often use another training run and compare their loss, gradient norms, *etc.* But these aggregate signals are noisy and easily diluted across millions of operations, offering little guidance on why the divergence occurs.

This paper introduces *bitwise alignment* as a correctness oracle and debugging primitive for LLM training, and OpGuard, a practical system that realizes it at production scale. OpGuard discovers semantic-stable operator boundaries across heterogeneous training stacks, and wraps them with lightweight fingerprinting. A schedule-tolerant mapper computes the longest prefix where two executions produce bitwise-identical tensors. The first mismatching point becomes a pivot for debugging and is presented with rich context. By carefully controlling benign nondeterminism, OpGuard makes the first mismatch strong evidence of error. OpGuard has been deployed at ByteDance across pre-training and post-training workloads. It diagnosed over twenty production issues, including subtle kernel races and silent data corruptions missed by existing checks, reducing debugging time from days to minutes.

1 Introduction

Large language model (LLM) training runs on a large number of GPUs for weeks [19, 29] on top of a complex software stack composed of user code, compilers, libraries, CUDA kernels, distributed runtime, *etc.* This stack also undergoes frequent updates [28], so bugs and hardware glitches are inevitable [17, 22]. Their symptoms are often subtle, such as loss spikes, gradient-norm drift, degraded model quality [35, 43, 77, 87].

When such an anomaly arises, developers need to determine (1) is the run truly buggy, and if so, (2) what the root cause is. This is challenging for several reasons. First, rerunning a training job consumes substantial resources, limiting the number of diagnostic reruns developers can afford. Second, the observed signals, such as loss and gradient norms, are only composite metrics aggregated over millions of operations and devices, so an error is easily diluted. Third, LLM training is prone to numerical variations [12, 19, 27, 80], so an anomaly may be harmless noise or early sign of a serious fault.

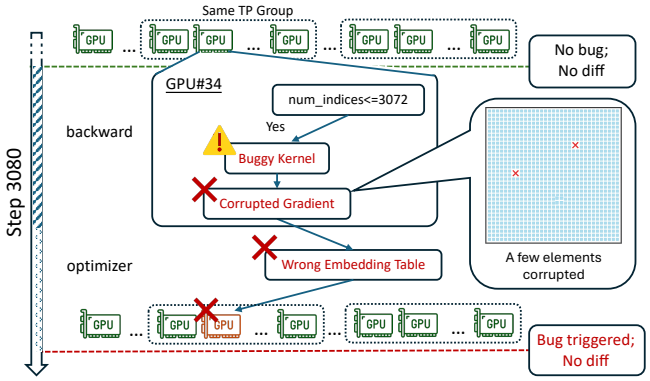


Figure 1. A tiny race in embedding backward affects one GPU's embedding shard. The next forward step reads the corrupt weight and propagates the error through its TP group and downstream layers. By the time the loss diverges, most GPUs already hold bad tensors.

A production incident illustrates the challenge. In one of our thousand-GPU vision-language model training runs, an alert on the gradient-norm curve was raised after over 3,000 steps. Expert developers spent five days running repeated experiments, toggling flags, and swapping kernels, but made no progress. The root cause turned out to lie in an embedding backward kernel, which had a tiny race condition that perturbed a few rows under rare token patterns (Figure 1).

To cope with the challenge, developers often compare a suspect run with a reference run. The reference run may come from an older release, a branch before a suspected feature, with different flags, with an alternative CUDA kernel, or even on another framework. This comparative strategy has been well studied in conventional debugging: delta debugging and related techniques compare executions, changes, or program states to isolate the minimal failure-inducing input or code [38, 95, 96, 99]. However, in LLM training, today's comparisons still rely on composite signals. Thus, they inherit the same ambiguity that can hide real bugs or raise false alarms. Even when two loss curves clearly differ, they offer little guidance on what causes the divergence. In the above case, developers repeatedly compared signals across runs (Figure 2), but this alone was insufficient to localize the bug.

Our insight is that effective comparisons for LLM training must be unambiguous and early enough. Composite signals have high ambiguity because they mix the effects of many operations and surface long after the original error; low-level code execution traces expose irrelevant details, creating many

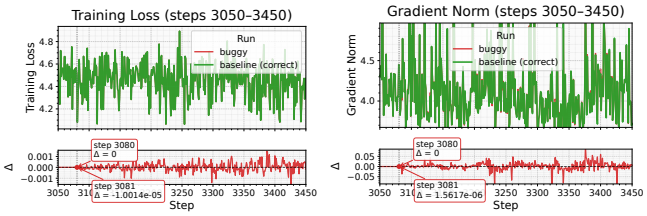


Figure 2. Loss and gradient-norm comparisons between a good run and the buggy run around Steps 3050–3450 for Figure 1. The bottom plot highlights where the buggy run first diverges and by how much.

benign differences when compared; tolerance-based tensor comparisons still leave ambiguity with ad-hoc thresholds.

Based on this insight, this paper proposes **bitwise alignment** as a new correctness oracle and debugging primitive for LLM training. Given two comparable training executions, we view each as a sequence of model-level computation boundaries and compute the *longest prefix* over which the tensors at corresponding boundaries match *bit-for-bit*. The first boundary failing this check is then the earliest point where the two runs diverge, regardless of how similar their aggregate metrics might still appear. This provides precise correctness criterion and the first diff becomes the debugging pivot.

At first glance, bitwise alignment seems infeasible for LLM training, which contains numerous sources of nondeterminism such as multi-threading and non-associative reductions. Production pipelines add further complexity through operator fusion, recomputation, graph capture, and backend-specific rewrites. These behaviors suggest that two training runs would diverge quickly, making bitwise alignment too fragile.

This intuition overlooks an opportunity: modern stacks provide substantial determinism control. Randomness can be seeded; kernel libraries expose deterministic modes [18, 24, 50, 51, 82]; data pipelines can be made reproducible[67, 83]; deterministic algorithms exist [20, 65, 73]. However, current determinism settings are scattered and used ad-hocly to improve *reproducibility*, not to systematically localize bugs.

We design OpGuard, a system that realizes the idea for production LLM training. OpGuard leverages existing determinism but organizes it to support debugging, and adds new mechanisms to make bitwise alignment feasible.

In practice, we find the space of bitwise-alignable runs is much broader than expected. With modest care, we can align runs that are several commits apart, runs that enable new features (*e.g.*, activation recomputation [11]), and even runs from different training stacks (*e.g.*, DeepSpeed vs. our custom runtime), as long as they implement the same model computation. This observation underpins our design of OpGuard and its applicability in complex, evolving LLM training pipelines.

Realizing this versatility requires addressing several challenges. Full-stack determinism is too costly, if not infeasible, for production environments. OpGuard does not seek absolute determinism. Instead, it identifies and neutralizes *avoidable* sources of variation for bitwise comparison to be well-posed.

Even so, unavoidable nondeterminism and complexities in production pipelines still pose problems. *How to make bitwise alignment robust even when the underlying stack is not fully deterministic and contains aggressive optimizations?* Production jobs routinely enable kernel fusion, recomputation, compute–communication overlap, evolving sharding layouts, and graph-captured executions. These optimizations reorder and regroup kernels, so any alignment scheme tied to a particular execution plan is brittle. Large organizations like ByteDance also use multiple training frameworks——PyTorch, DeepSpeed, Megatron, and internal runtimes——whose APIs differ substantially. Correctness checks defined in terms of specific framework APIs are therefore too restrictive and limits available reference runs. Finally, tensors are enormous, making full tensor dumps prohibitively expensive. Their representations can also change under sharding or reshuffling.

To address these challenges, OpGuard identifies and instruments a small set of *semantic-stable* boundaries—the lowest-level Python operators that correspond to model semantics, such as linear projections, layer normalization, and attention blocks. These points represent the same mathematical transformation across frameworks. Because these boundaries sit above kernel choices and scheduler decisions, they also become *execution-stable*: even if one run fuses kernels, overlaps communication differently, or uses a different CUDA Graph capture, the same semantic boundary is still reached and observed. OpGuard uses a short preflight phase that runs each stack to efficiently discovers these stable boundaries.

OpGuard automatically inserts a lightweight device-side *fingerprint kernel* around these boundaries, which emits a constant-size, *scale-stable* fingerprint. During the main run, OpGuard records only these fingerprints. This reduces the trace size and ensures the comparison does not depend on the size, layout, or sharding of the underlying tensor. Afterward, OpGuard uses a *schedule-tolerant mapping* algorithm to reliably align the fingerprint streams from two executions. It computes their longest bitwise-identical prefix, and reports the first boundary that differs, together with its immediate context. OpGuard further includes UI support that presents this boundary as a pivot in a unified timeline so developers can visually inspect how the discrepancy propagates.

OpGuard has been deployed in production in ByteDance for 8 months. It seamlessly supports different pre-training and post-training frameworks and heterogeneous hardware backends. This ease of integration is because OpGuard observes model-level tensors rather than framework APIs or internals.

To date, OpGuard has helped diagnose 20 production bug cases, including resolving 11 long-standing difficult cases like the motivating example. They cover a variety of subtle issues, such as nondeterministic races in communication-overlap paths, one-bit routing errors in MoE routing kernels that later explode into shape mismatches, and long-standing cross-stack mismatches such as different loss masking policies between Megatron and FSDP. Across these cases, OpGuard reduces

diagnosis time *from days to minutes*. In a separate online SDC-detection mode, OpGuard has also detected more than 21 silent-data-corruption (SDC) [44] cases. We also evaluated OpGuard in open-source training ecosystems. It enabled the diagnosis of 4 new reproducible issues and 6 long-standing bugs whose cumulative resolution time exceeded fifteen days.

Besides diagnosis, teams have used OpGuard as a reliable correctness oracle in scenarios where system changes introduce subtle correctness regressions, including new overlap schedules, recompute policies, sharding layouts, heterogeneous backends such as GPUs and NPUs, and compiler-generated execution paths. These workflows routinely alter execution order, precision, kernel fusion, or parallel boundaries—changes that are notoriously hard to validate and historically have caused silent divergences in production training. OpGuard provides a principled way to confirm correctness of low-level optimizations. This allows teams to safely innovate on scheduling, parallelism, and compilation.

In summary, this paper makes the following contributions:

- We formulate bitwise alginment as a new correctness oracle and practical debugging primitive for LLM training.
- We design and implement OpGuard, a system that realizes this primitive into a practical and scalable workflow.
- We evaluate OpGuard across production pre-training and post-training pipelines in ByteDance as well as open-source stacks. The results demonstrate its impact.

2 Background and Motivation

LLM training requires extensive computation distributed across many devices. A single training run traverses layers of software and hardware: user code that defines the model, optimizer, and configuration; distributed framework that implements data, tensor, and pipeline parallelism [79] across accelerators; runtime that schedules operators and manages multiple CUDA streams [52]; compiler and kernel-generation framework that lower operators into fused CUDA kernels [91].

2.1 Correctness Challenges in Large Training Stacks

Our experience in operating large training pipelines suggests that faults can originate from any part of this stack. A small deviation in user code can change the intended computation. Mistakes in the parallelism partitioning, or microbatch ordering can cause different devices to see inconsistent parameters or activations. Races between transfers and computation can corrupt intermediate state. A collective may reuse a stale buffer or misroute a shard. At lower levels, corner-case kernel fusion bugs, nondeterministic kernels, and hardware-level issues such as silent data corruption, can all produce wrong activations or gradients. Even the systems surrounding a ML framework, such as dataloaders and caching service, may supply incorrect or reordered batches that appear valid.

Moreover, these faults seldom produce an observable symptom at the moment they occur. They often initially affect only a single device or a handful of tensor elements. As computation intensifies and training state synchronizes, this local error gradually propagates. Eventually it manifests as some visible anomaly, *e.g.*, a drift in gradient norms. By that time, the original fault is often many layers and operations away. Worse still, the same symptom can be caused by many possible faults.

These characteristics makes correctness debugging in LLM training notoriously challenging. An apparently abnormal signal may turn out to be harmless noise, while a subtle deviation may be the first sign of a serious underlying fault. Localizing the true root cause is akin to finding a needle in a haystack spread across time.

2.2 Existing Solutions and Their Gaps

Substantial tools have been created for debugging individual training components. Framework-level debuggers and anomaly modes (*e.g.*, TensorFlow Debugger [84], PyTorch Autograd Anomaly [69]) focus on the training program and catch obvious mis-specifications. GPU correctness and sanitization tools (*e.g.*, cuCatch, NVIDIA Compute Sanitizer, GPUBurn) target kernel behavior and memory errors [8, 39, 54, 81]; MPI and communication checkers such as MUST [86] focus on collectives; hardware-focused tools like DCGM diagnostics [57] watch for device faults; and cluster monitoring systems track service and data anomalies. These tools are useful for identifying local issues, but each tool focuses on a narrow slice of the system. They do not answer the core questions developers face during correctness incidents of an end-to-end training task: (1) *is this run still performing the computation as intended?* (2) *where in this long execution did the misbehavior first occur?*

In practice, engineers routinely rely on a pragmatic strategy: they run a candidate training job next to a known-good one with the same checkpoint and seed, then compare their loss curves, gradient norms, activations, and related metrics. This approach is popular because it provides a concrete baseline for an otherwise fuzzy notion of correctness, while fitting naturally into existing rollout workflows. When the two runs match, engineers gain confidence; when they differ, there might be a bug that warrants investigation.

However, these comparisons only operate at composite signals that aggregate the effect of millions of operations, masking subtle discrepancies for long periods while amplifying harmless numeric fluctuations. Across real incidents, we observe that they help reveal two runs have diverged, but they offer little insight about why. Debugging remains a tedious trial-and-error by toggling flags, adjusting settings, swapping kernels, *etc.* Each rerun consumes costly cluster resources.

What is missing is a *precise* comparison mechanism to augment this practice. It must surface faults soon as they occur, while pinpointing what causes the divergence. It should remain robust under production training stacks and only reveal true differences in model computation semantics. In addition, it should not require extensive changes to a training pipeline, disable core optimizations, or incur severe slow-down.

3 Abstraction: Bitwise Alignment

To address the challenges and gaps described in §2, we advocate for *bitwise alignment* as a correctness oracle and debugging primitive for LLM training.

Definition 1 (Bitwise alignment)**.** *Consider two training executions E_1 and E_2 that are expected to compute the same model. Each execution produces a sequence of observable boundaries: $E_1 = \langle b_1^1, \ldots, b_m^1 \rangle$ and $E_2 = \langle b_1^2, \ldots, b_n^2 \rangle$. A boundary b has an identifying model-level operator $\sigma(b)$ with an input/output tensor state $T(b)$. An alignment between E_1 and E_2 is an order-preserving partial matching $A = \langle (b_{i_1}^1, b_{j_1}^2), \ldots, (b_{i_k}^1, b_{j_k}^2) \rangle$, where $\sigma(b_{i_r}^1) = \sigma(b_{j_r}^2)$. Bitwise alignment is the longest prefix of A such that every matched pair satisfies $T(b_{i_r}^1) \equiv_{\text{bit}} T(b_{j_r}^2)$.*

3.1 Alignment Spectrum

A useful way to think about correctness in LLM training is to examine it along a spectrum of alignment strengthen.

At the weakest level, *behavior alignment* judges whether the overall behavior of a model, *e.g.*, its accuracy, BLEU score [60], or loss curve, remains statistically consistent across runs. This level suffices for high-level monitoring and regression testing but offers no visibility into internal correctness: two runs can exhibit identical accuracy yet silently diverge in intermediate activations or gradients.

A stronger level is *numerical alignment*, which compares two tensors and treat them as equal if they are close enough, as in `torch.allclose` [68]. This appears robust, since it tolerates small differences, but in practice it is an unreliable oracle. A 10^{-4} difference in logits may be acceptable but may indicate a serious bug for a loss or normalization constant. Choosing a single tolerance that works across layers, data regimes, and models is difficult. In addition, due to floating-point arithmetic's non-associativity, small discrepancies arising from reduction trees, fusion patterns, or accumulation paths will change the results even when the underlying computation is mathematically the same. Raising the tolerance reduces false alarms but will also mask small-but-systematic drifts caused by actual bugs (*e.g.*, losing a few rows in a gradient, missing an update, or corrupting a subset of parameters) [40]. Numerical alignment is inherently ambiguous.

At the strongest level lies *bitwise alignment*, which demands that the tensors match exactly, *bit for bit*, across operations in both forward and backward passes. This *binary predicate* removes ambiguity entirely: two runs are either identical or not. When they differ, the first mismatched operation marks the precise boundary where equivalence breaks, exposing the responsible kernel, reduction order, race, *etc.* Therefore, it provides strong diagnostic power.

3.2 Make Bitwise Alignment a Precise Oracle

For bitwise alignment to serve as a precise oracle, mismatches should indicate a real difference in computation logic rather than incidental noise. This requires three key conditions.

Comparable replay. The two executions should be comparable, *i.e.*, implement the same mathematical logic. They need not be identical binaries. Many useful comparisons intentionally differ in implementation. A related assumption is that an execution can be reproduced given the same data, seeds, and environment. This is often feasible. Unlike conventional systems where reproduction is hard [59, 98], LLM training is fundamentally more reproducible. Its computation follows a mostly static graph of tensor operations, and its inputs are drawn from fixed datasets instead of dynamic, unknown user requests. If two pipelines intentionally read or preprocess data differently, they should first align the input, e.g., using the same checkpoint or captured post-preprocessing tensors. Reproducing a faulty run is generally feasible even when the underlying bug is a data race. LLM training is computation-intensive and repetitive: an operator is often invoked thousands of times per training step and millions of times over a run, which creates abundant opportunities for the race to manifest.

Feasible, however, does not mean cheap. Replaying a large training job consumes substantial resources and time, so a debugging system should require as few reruns as possible. OpGuard uses one captured faulty run and a reference run to expose the first boundary divergence; the faulty execution itself need not be deterministic.

Controlled nondeterminism. LLM training stacks contain various sources of nondeterminism that can introduce variation between runs even when they do not change the intended computation. We distinguish two categories. *Controlled nondeterminism* is benign variation in the execution environment, *e.g.*, RNG streams, data order, library kernel choices, collective topologies, and numeric modes. *Residual nondeterminism* is variation that remains after these controls are fixed, typically because user code, third-party kernels, or hardware behavior is schedule-sensitive. The former should be stabilized so that bitwise comparison is well-posed; the latter is intentionally left visible and treated as a bug signal. §4.2 describes how OpGuard enforces this split.

Model-level boundaries. Bitwise comparison should be applied at a granularity that yields stable and precise outputs. Comparing at full training steps or whole layers would be too coarse to localize faults. Comparing at every Python-level callsite would be too noisy and inconsistent: many functions do not operate on tensors, and those that do may be reordered, fused, or eliminated by tracing and compilation passes. Conversely, if the comparison is too low-level, *e.g.*, framework internals or temporary buffers, it becomes highly sensitive to implementation details. Small changes in heuristics, autotuning, or vendor libraries can add, rename, or remove temporary tensors without affecting the model semantics.

We find the right granularity for bitwise alignment are *model-level operators whose input and output tensors are materialized in every valid execution of the model*, *e.g.*, a linear projection, a layer-normalization call, an attention block,

or an MLP submodule. We call those comparison points **model-level boundaries**. By materialized tensors we mean those that any faithful implementation should produce and pass to the next model-level operator as opposed to transient buffers, which a compiler may create, rename, or eliminate. These tensors define the *contract* between adjacent model components. Backends may change kernel schedules, fusion plans, or temporary buffers, but changing a boundary tensor changes the model computation. A match at such a boundary implies that the proceeding components' transformations are equivalent for that input; a mismatch marks the first component that has divergent transformation. This granularity is stable across framework and backend because different optimizations should preserve model logic; yet it remains fine-grained to localize the divergence to a small region of model code.

4 Design of OpGuard

We design OpGuard, a system to realize the bitwise alignment abstraction. Our goal is to make OpGuard practical for production training pipelines. OpGuard also aims to make bitwise alignment broadly applicable as a general and powerful debugging primitive so it can support comparing executions across frameworks, compilers, libraries, hardware backends, and distributed runtimes, as long as both runs represent the same model computation. Otherwise, bitwise alignment would only be a narrow regression check for nearly identical binaries, which would greatly limit its utility.

Achieving these goals requires turning the insights discussed in §3.2 into concrete mechanisms. Production training pipelines are highly complex and contain aggressive optimizations such as operator fusion, and graph capture and rewriting. These complexities introduce several design challenges for OpGuard. First, the stable comparison points required by bitwise alignment are not directly exposed by single API surface. OpGuard needs to efficiently discover the boundaries without requiring users to rewrite models or disable optimizations. Second, production training runs generate enormous tensors. Dumping full tensors is infeasible. OpGuard should extract lightweight, scale-independent observations that are still informative. Third, even after benign nondeterminism is neutralized, two executions likely will not emit outputs in exactly the same order. OpGuard should align partially ordered observations and recover the longest prefix.

4.1 Overview

OpGuard addresses these challenges with a three-phase workflow that compares a suspect execution against a reference run (Figure 3) for post-failure root cause diagnosis. OpGuard has a separate deployment mode for online SDC detection, which is discussed in §6.4. The Preflight phase (§4.3) runs a few iterations in eager mode to discover model-level boundaries (§3.2) shared by both runs, and produces an alignment plan. The Guarded Execution phase (§4.4) wraps the discovered boundaries with input/output fingerprinting and runs the full

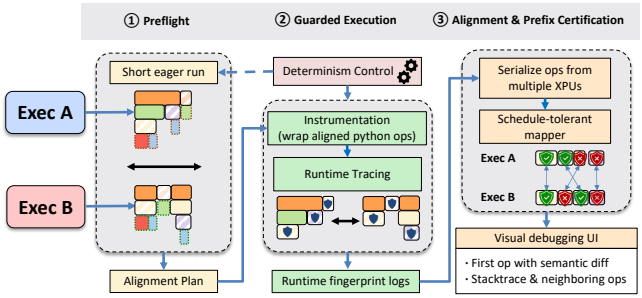


Figure 3. OpGuard workflow.

training job, producing compact fingerprint logs. Determinism control (§4.2) is applied in this phase as well as in Preflight. The Alignment phase (§4.5, §4.6) processes the two logs offline. A schedule-tolerant mapper first identifies anchors to partition each trace into windows; within each window, the mapper pairs the remaining boundaries despite fusion, overlap, and asynchrony. Prefix certification then walks the matched boundary pairs to compute the longest bitwise-identical prefix and report the earliest divergent boundary. A visual debugging UI presents this boundary along with surrounding context for engineers to quickly triage and pinpoint the root cause.

4.2 Determinism Control

As discussed in §3.2, bitwise alignment is only well-defined if nondeterminism is controlled. Conventional deterministic multithreading systems [6, 7, 15, 16, 42, 46, 47] force reproducible execution by controlling thread schedules, system-call results, and external inputs. One extreme for LLM training is full-stack bitwise determinism, exemplified by PaLM [12]. PaLM's training pipeline was engineered so that, starting from any checkpoint, resuming training produces bitwise identical results. Achieving this required carefully constraining the entire hardware and software stack. Outside such bespoke stacks, this level of control is rare. While mainstream frameworks such as PyTorch and TensorFlow offer deterministic modes, they are best-effort and do not eliminate nondeterminism. Enforcing bitwise determinism across heterogeneous fleets (mixed GPU generations, different drivers/compilers, evolving libraries) is also fragile. A single library upgrade or kernel change can break the assumption.

OpGuard's goal differs from deterministic runtimes and from PaLM's full-stack discipline. It does *not* aim to make every run absolutely bitwise deterministic or to impose a deterministic scheduler, which can severely slow down production workloads. Instead, it identifies and fixes *controlled nondeterminism*: avoidable sources of variation that would otherwise make two runs with the same intended computation perform different arithmetic and make bitwise alignment meaningless. Many numerical differences between runs arise from sources that have nothing to do with model correctness: different shuffling orders, RNG streams, collective algorithms or bucketizations, kernel variants, or numeric modes. If these

Surface	Controlled?	What remaining nondeterminism means
Randomness & data order (dropout, shuffling, sampling)	Yes	We fix all RNG seeds and sampler settings so that both executions see the same mini-batch sequence and the same stochastic draws. If runs still diverge here, it reflects data/config drift.
Framework / library kernels (torch ops, cuBLAS/cuDNN, fused kernels)	Yes	We enable deterministic algorithms and pin kernel choices so that, given the same inputs, the library computes in the same arithmetic order. After this, mismatches usually indicate version/flag skew or a latent library bug, not benign nondeterminism.
Hardware collective / reduction order (NCCL tree/ring, chunking, topology)	Yes	We pin NCCL's algorithm and topology so all partial sums follow the same reduction tree. Different trees lead to different FP accumulation orders, so we treat those as different configurations and should not compare them. If the hardware produces different bits despite identical inputs and kernel, this constitutes *silent data corruption (SDC)*.
User / third-party code & scheduling (custom CUDA kernels, host concurrency, warp/SM timing)	**No**	We do not fix CPU thread schedules, warp/SM timing, or user kernel design. Correct kernels must be schedule-invariant. If scheduling changes bits (racy atomics, missing sync, uninitialized memory), that is a bug, not benign nondeterminism, so OpGuard is meant to expose it.

Table 1. OpGuard distinguishes controlled nondeterminism from residual nondeterminism. Controlled nondeterminism inside the stack (RNG, framework kernels, hardware reduction order) is stabilized so bitwise comparison is well-posed. Residual nondeterminism after these controls is treated as a bug signal when it changes output.

choices differ between runs, they no longer perform the same arithmetic, making bitwise comparison meaningless.

As Table 1 lists, these are exactly the categories OpGuard controls. The controls are lightweight and do not disable core optimizations or alter user-visible training logic.

Specifically, OpGuard applies nondeterminism controls at five surfaces. First, it makes randomness and input order replayable by fixing CPU/CUDA RNG state, dropout streams, dataloader workers, sampler state, and distributed initialization, so both executions consume the same mini-batches and stochastic draws. Second, it pins arithmetic choices inside framework and vendor libraries: deterministic algorithm modes, fixed cuDNN/cuBLAS workspace behavior, disabled autotuning, and consistent numeric modes such as TF32 settings keep kernels using the same implementation and accumulation order. Third, it fixes distributed reduction order by pinning collective algorithm, protocol, topology, and bucketization; otherwise two correct runs can differ solely because floating-point partial sums follow different trees. Fourth, when comparing from a checkpoint, it restores all state that affects future execution, including RNG streams, optimizer state, dataloader position, and scheduler counters. Finally, for comparisons across tensor-parallel configurations, OpGuard can use a TP-simulator mode that executes the full unsharded arithmetic while preserving the logical parallel configuration, preventing sharding-induced reduction-order changes from being mistaken for faults.

Importantly, OpGuard does *not* suppress residual nondeterminism in user/third-party code or fix scheduling. Schedule-sensitive custom or third-party kernels (*e.g.*, CUDA/Triton kernels with race conditions or nondeterministic reductions) are treated as part of the system under test rather than stabilized or reasoned about internally. If such a kernel is not schedule-invariant, OpGuard surfaces the effect as a divergence. Thus controlled nondeterminism is noise that OpGuard removes, while residual nondeterminism is evidence that the tested system has a latent correctness problem. The exhaustive knob-level recipe is detailed in Appendix A.5.

4.3 Preflight: Discovering Stable Model Boundaries

OpGuard splits debugging into two phases: a short Preflight phase that determines *where* bitwise comparison should occur and whether the two executions share those points, followed by Guarded Execution that instruments only those points. This separation enables low overhead for long training runs.

Preflight's goal is to discover a shared set of boundaries across two executions that are expected to compute the same model-level transformation. These boundaries should define a stable comparison grid: regardless of differences in compiler passes, fusion strategies, or kernel schedules, the same boundaries in two runs should see the same logical tensors. Thus, same computation is a contract on boundary values, not source code or kernel identity.

We use the MoE block in Figure 4 as a running example throughout the design. The block dispatches tokens with `all_to_all_dispatch`, normalizes activations with `fused_rmsnorm`, runs expert computation with `grouped_gemm_experts`, and combines expert outputs with `moe_combine`. One execution may fuse adjacent kernels or schedule communication on different streams, but if it implements the same MoE block, it should expose the same logical tensors at the boundaries.

Preflight assumes the runs start from equivalent checkpoints, consume the same training examples in the same microbatch order, make the same seeded random choices, and use the same numerical contract for each boundary (*e.g.*, precision policy, normalization formula, optimizer update rule, and collective reduction order). They may still use different kernels, fusion plans, recomputation strategies, graph-capture schedules, or physical sharding layouts, as long as those choices implement the same boundary-to-boundary tensor transformation.

As discussed in §3.2, choosing the right boundary granularity is critical. OpGuard chooses *model-level operators* whose input and output tensors are materialized in every valid

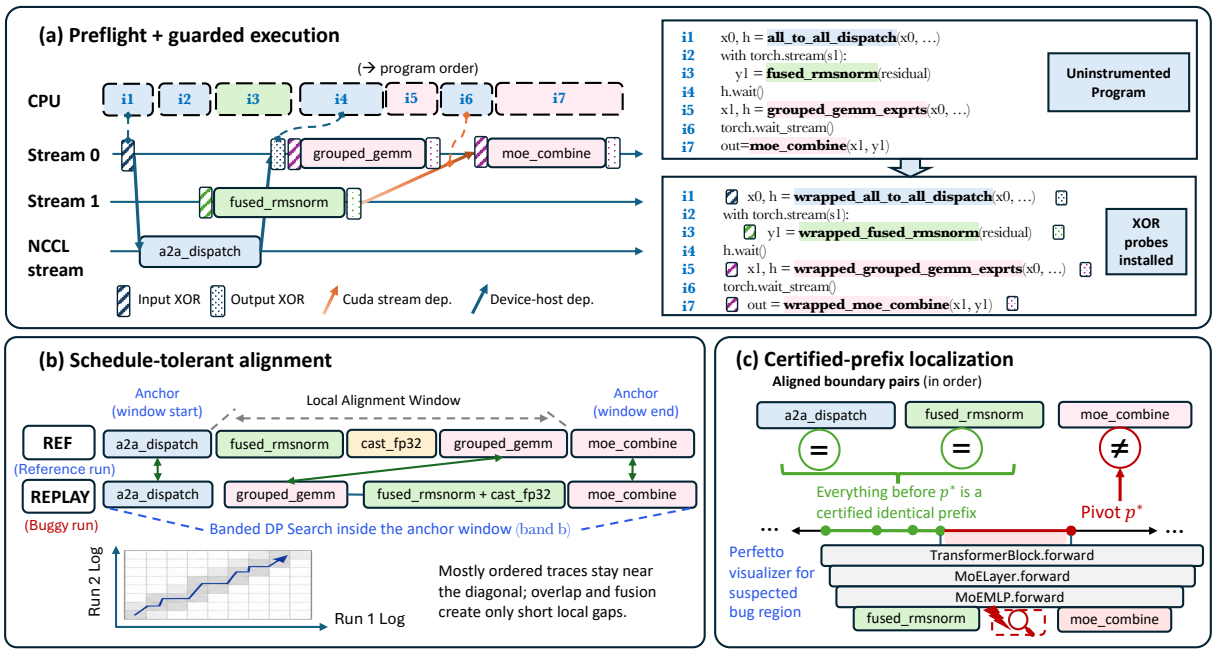


Figure 4. OpGuard components on one running MoE-block example. (a) Preflight discovers stable model-level boundaries and guarded execution installs input/output XOR probes at those boundaries. (b) Schedule-tolerant alignment pairs boundary logs despite fusion, overlap, and local reordering. (c) Prefix certification reports the first mismatching aligned boundary as the debugging pivot.

execution of the model, specifically the lowest Python-level operators that directly initiate device work (through C++/CUDA kernels), *e.g.*, `linear`, `layer_norm`, attention blocks, the four MoE operators in Figure 4. These operators are explicit in model code and define the model computation, while existing in every major training stack. They form a consistent logical order despite backend differences. Bugs must eventually perturb a tensor passed through one boundary.

OpGuard discovers these boundaries automatically. It executes each pipeline for a few iterations, and enables Python- and device-level tracing for tensor-producing operations. It then inspects the stack frames of these operations and filters out frames originating from JIT internals, compilation runtimes, and vendor libraries. From the remaining frames, OpGuard selects the nearest user-defined operator as a candidate.

OpGuard then identifies the device work caused by each candidate operator. This is needed to (1) filter Python callsites to operators that actually launch device work; (2) provide runtime context (streams, launch order, timing, *etc.*) for later stages. This attribution is non-trivial. A single model-level operator may trigger multiple CUDA or Triton kernels, call vendor libraries, issue collectives, or runtime-scheduled launches. The device trace alone cannot name the model operator that caused these launches, while the Python trace alone does not expose their concrete runtime behavior.

OpGuard therefore uses a layered attribution scheme. Its Python tracer records the operator intent (function names, callsite, argument summaries). Device-level tracer records kernel launches, memory operations, collectives, streams, and timestamps using the device profiling interface [55] and kernel-launch interception. OpGuard merges the two views into a unified stream that associates each Python-level operator with the set of device activities it triggers. This provides both the stable operator identifiers for instrumentation and concrete context for attaching fingerprints and later debugging.

4.4 Guarded Execution: Fingerprinting at Boundaries

After preflight identifies the boundaries to observe, OpGuard instruments only those sites during guarded full training execution. The instrumentation is done with framework operators instead of a separate profiling runtime. At each boundary, the original callsite is replaced with a small wrapper that launches a pre-op fingerprint kernel, invokes the original operator, and then launches a post-op fingerprint kernel. All other model code is untouched. Since the wrappers are normal framework operators, they preserve the original execution mode (eager, `torch.compile`, CUDA Graphs). In Figure 4a, the callsites for `all_to_all_dispatch`, `fused_rmsnorm`, `grouped_gemm_experts`, and `moe_combine` are replaced with wrappers.

Each wrapper emits a compact trace entry containing the boundary identifier, tensor metadata (shape, dtype, device, stream), rank id, and a local monotonic timestamp. The fingerprint kernels read the input/output tensors and compute a lightweight XOR hash; they add no tensor copies or control logic. Appendix A.4 analyzes the fingerprint's sensitivity to tensor precision, value distributions, and corruption modes.

Correct placement is essential. Modern training stacks routinely overlap computation and communication across

multiple streams, and many operators execute asynchronously. Fingerprints must run *exactly* where tensors become valid and on the same execution stream as the operator they guard. Thus, each wrapper launches its fingerprint kernels on the operator's own stream and attaches them to the same synchronization points (completion handles, `wait_stream` sites). This placement embeds fingerprinting into the existing stream graph without introducing additional synchronization or reordering.

Figure 4a illustrates this placement rule on the MoE block. The output probe for `all_to_all_dispatch` attaches after the collective handle completes; probes around `fused_rmsnorm` run on its stream; and the probe before `grouped_gemm_experts` respects the wait edge from the collective stream. In each case, the trace records the tensor at the device-time point where the program consumes or produces it, together with the callsite and local stream-order constraints. The runtime appends these entries to one log in device-dependency order, reusing existing handles and streams so logging adds no cross-stream synchronization and does not alter scheduling.

4.5 Schedule-Tolerant Alignment

Even when two executions implement the same computation, their fingerprint logs rarely line up position by position. Stream overlap, fusion, graph rewrites, and backend-specific kernels can introduce local reordering or extra events without changing the boundary values. A positional diff would confuse such schedule skew with real tensor drift.

The mapper looks for high-confidence, order-preserving partial matching. A pair of events are only matched when they share the same boundary identifier, tensor shape and dtype, and rank. A positive score would be given if they have relatively close device-time and operator index. Matching events should preserve relative order: if event a precedes event b in one log, their counterparts should appear in the same order in the other log. Gaps are allowed but penalized. They represent inserted or fused work that remain unmatched.

The mapper's alignment algorithm runs in three stages. First, it constructs anchors from boundaries that are unique in a local region, appear in both logs, and preserve relative order. A boundary is used as an anchor only when its symbolic operator name and tensor metadata are unambiguous among nearby events. These anchors split the logs into small windows, preventing repeated kernels or ambiguous metadata from pulling the mapper into a globally inconsistent alignment.

Second, within each anchor window, OpGuard runs a banded monotone dynamic programming algorithm (listed in Appendix A.3). Production traces are mostly ordered: stream overlap, fusion, and graph rewrites usually create short local displacements rather than arbitrary permutations. The band keeps the search near the diagonal, covering these local skews while avoiding the cost and instability of unconstrained global matching. If a true counterpart falls outside the window or is ambiguous, OpGuard leaves it as a gap.

Third, OpGuard performs a conservative rescue pass for missed matches that move slightly across anchor boundaries, such as repeated reduction events emitted at different times. The pass scans nearby unmatched events using device-time windows and fingerprint metadata, and adds a pair only when it is consistent with the established monotone alignment.

In Figure 4b, for example, one MoE run records `all_to_all_dispatch`, `fused_rmsnorm`, `grouped_gemm`, and `moe_combine`; the other reorders `grouped_gemm` around normalization and fuses `fused_rmsnorm` with a cast. Anchors keep the shared boundaries in order, local DP absorbs the short displacement, and fused work simply remains unmatched.

4.6 Prefix Certification and Fault Localization

Given two aligned traces, what engineers want is not a long list of discrepancies but a single, trustworthy pivot that separates equivalent behavior from a real tensor divergence. In the running example (Figure 4c), the two runs agree through dispatch and normalization, but the first mismatching aligned boundary is `moe_combine`; this boundary becomes the debugging pivot.

OpGuard therefore computes a *certified prefix*: the longest run of matched boundary pairs whose fingerprints agree on both sides. It walks the aligned sequence of boundaries and, at each one, checks whether both executions produced the same tensor. For most boundaries, this is a comparison of the attached fingerprints; suspected mismatches are confirmed with a one-time byte-level tensor comparison. The longest run of successful checks is the certified prefix.

This prefix yields a strong invariant: everything before that point is bitwise identical; everything after is downstream and allowed to differ for benign or faulty reasons. If both runs are supposed to implement the same computation at the reported boundary, a mismatch is strong evidence of a real fault. If instead it reflects an intentional change in the computation (*e.g.*, a different masking convention), the certified prefix still tells engineers exactly how far the two runs behave the same.

OpGuard reports the first divergent boundary together with its context, including the Python operator, stream, tensor metadata, rank, and timing information. Not all boundary differences are important. Some boundaries legitimately disagree on performance or configuration-dependent values, e.g., XPU utilization [10], peak-memory metrics, profiling buffers, RNG state [71], sliced views, padded regions, or scratch workspaces. These differences are difficult to eliminate completely with static rules, but easy for engineers to discard in the visualizer. In one major pre-training framework, engineers found only five such cases: three were padded output tensors allocated inside an operator, where the extra elements were never consumed, and two were RNG-state records whose differences stayed in framework bookkeeping while neighboring model tensors still matched. OpGuard shows the certified prefix as a marker in the timeline, the matching operators around the first divergence, the argument or output that differs, and the corresponding callstack/source location. If the mismatch is isolated

in the middle of otherwise matching neighboring operators, it is usually a false positive rather than a propagated model-state difference. Engineers can then discard the harmless tensor and, when appropriate, add a small filter rule.

5 Implementation

We implement OpGuard with 25.6K lines of code (86.7% Python, 11.3% C++, and 1.1% CUDA).

Identifying Instrumentation Sites in Source Code. Once preflight (§4.3) determines which operators to observe, OpGuard must locate the corresponding source-level sites. This step should tolerate the small, routine edits engineers make during debugging (reordering expressions, inserting print logs, extracting helpers) without new preflight run. OpGuard therefore uses a pattern-driven matcher rather than line-number tracking. For each traced operation, it re-parses the surrounding source using both abstract syntax tree (AST) and concrete syntax tree (CST). The AST captures the operator's program role (receiver, arguments, operation type), while the CST preserves syntactic structure and keeps instrumentation transparent: engineers may inspect the generated patches when desired, but routine use requires no manual confirmation.

This approach reliably identifies tensor-producing sites across diverse syntactic patterns, including fused expressions, helper-function refactorings, triton kernel launches, and in-place indexed writes. Because instrumentation is inserted statically, OpGuard preserves the framework's native dispatch and scheduling behavior, avoiding the perturbations of runtime monkeypatching. The code block in Figure 4a illustrates the resulting code-level transformations. Appendix A.2 details the matching rules and algorithms.

Runtime Trace Batching. During tracing, each rank accumulates trace records in an in-memory buffer that batches thousands of entries before flushing them to disk. This batching amortizes queue synchronization, JSON conversion, and file I/O costs. A background consumer thread on each rank then finalizes and writes each batch, offloading all heavyweight CPU work (serialization, I/O, and tensor-to-bytes conversion) from the critical GPU path. As a result, batching eliminates nearly all GPU–CPU synchronization points and reduces tracing overhead to negligible levels. Keeping `dump_mode="summary"` transfers only XOR signatures, while full tensor dumps are reserved for explicit debugging.

Distributed Trace Collection and Ordering. OpGuard collects traces on each GPU rank independently and should later assemble them into a consistent global order. Each instrumented operator thus carries two ordering metadata:

(1) *Intra-rank ordering*. Operations within rank may execute on multiple CUDA streams are completed out of submission order. OpGuard uses the GPU `%globaltimer` values recorded around each XOR kernel to derive a monotonic device-time order that respects inter-stream dependencies.

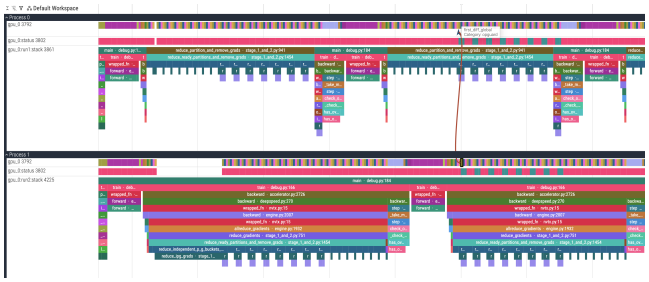


Figure 5. Perfetto visualization for Bug 21.

(2) *Inter-rank ordering*. Device clocks are not synchronized across ranks. OpGuard thus orders operators using their Python-level launch timestamps. These reflect global program order and are consistent across nodes; rank ID and per-rank sequence numbers break ties deterministically.

This lightweight scheme yields a coherent global timeline without clock synchronization or CUDA barriers. It assumes that inter-node clock skew is negligible relative to operator latencies, which in our workloads span 10^{-4}–10^{-1} seconds. Unlike Spanner [13], which uses bounded clock uncertainty to enforce consistency, our goal is diagnosis. A skewed clock can at worst misorder a few near-concurrent events and does not obscure the divergence OpGuard is designed to identify.

Trace Visualization. OpGuard exports its aligned traces to Perfetto [63], the timeline viewer used by `torch.profiler`, so engineers can inspect divergences without learning a new UI. Each GPU rank is shown as a separate lane with tracks for Python operators, kernel launches, and callstacks. Because OpGuard reconstructs a *logical* timeline, all slices are rendered with uniform width, focusing the view on ordering rather than noisy kernel durations. Aligned operators are color-coded and flow links connect corresponding slices across the two executions. This makes the first divergent boundary immediately visible and allows engineers to examine its neighborhood, jump to the source, and check whether later operations re-converge. By embedding alignment metadata and tensor signatures into Perfetto, OpGuard turns distributed tensor drift into a familiar profile-style visualization (Figure 5).

6 Evaluation

Our evaluation answers: (1) how effectively does OpGuard diagnose production failures in LLM training? (2) how accurately does it localize the first divergent operator under weakened configurations? (3) while designed for software debugging, does OpGuard help with hardware-induced SDC? (4) how robust is alignment when two executions differ in trace shape? (5) what are the coverage and runtime overheads of OpGuard across workloads and scales?

6.1 Deployment Status and Scale

OpGuard is deployed in ByteDance's production training environment and is actively used by more than **15** engineering teams across pre-training, post-training, vision–language training, heterogeneous hardware, platform, and compiler

groups. It is exercised on end-to-end model training jobs as well as compiler and kernel pipelines that introduce new fused operators and hardware-specific optimizations.

Debugging Workflow. When a run shows loss drift, nondeterminism, routing instability, or shape errors, engineers first try lightweight triage such as toggling flags, switching kernels, or bisecting the model. Cases unresolved by these approaches are escalated to OpGuard: engineers pick a reference run or configuration, replay both executions from a shared checkpoint with instrumentation, and inspect the first divergent operator in the visualizer. In this workflow, OpGuard is the *second-line* and often *final* localization tool for difficult cases.

Rollout Timeline. OpGuard was piloted within pre-training groups from Aug.–Oct. 2025, during which it was validated on real production failures and integrated with job orchestration tools. It was rolled out the broader infrastructure organizations in Nov. 2025. Adoption has since expanded across training, compiler, runtime, and platform teams. OpGuard is now routinely invoked in production debugging.

Scale and Deployment Modes. OpGuard has been applied on production jobs of up to 512 XPUs, covering diverse training pipelines. It exposes three deployment modes. In *trusted mode*, OpGuard skips a small allowlist of high-confidence primitives and instruments the remaining operators. In *full mode*, OpGuard instruments every operator boundary. Engineers use trusted mode by default for reactive two-run debugging and switch to full mode when they need complete coverage. The same debugging workflow can run either as a smaller-scale replay using fewer machines than the original job or as a targeted rerun from a shared checkpoint. The third mode, *online SDC detection* (§6.4), runs continuously alongside long-running production jobs and checks only the communication boundaries needed to identify suspect machines.

6.2 Production Failure Diagnosis

Following deployment, OpGuard has been applied on **20 production failures** escalated by engineers. We evaluate *all* cases, without filtering by reproducibility, complexity, or expected outcome. Across all cases, OpGuard quickly localized the culprit operators, reducing triage time from multi-day investigations to *minutes*. Engineers shared positive feedback after using OpGuard during incident response, *e.g.*,

> *"We had been chasing the wrong subsystem for almost a week. OpGuard showed the exact kernel in under fifteen minutes."*

> *"Without OpGuard, we would never have noticed that the drift originated in a single-row race. The loss precision is too low and gives us no clue where to start."*

Table 2 summarizes a representative subset of 11 failures with reliable engineering logs. "Manual triage" measures the time from when a ticket was opened to when engineers identified the root cause and proposed a fix. The fixes themselves are small (typically < 10 LOC), so this interval is dominated by diagnosis. The OpGuard times are coarse-grained estimates provided by engineers: the time to open OpGuard's UI, inspect the surfaced first-difference operator, and confirm its correspondence to the actual faulty region. The remaining cases are similar but lack precise timestamps. Table 3 lists the full set and the first diff OpGuard identifies.

Case Study: (Bug 1). The motivating example (Figure 1) illustrates the challenges of manual diagnosis. The underlying bug is a small race condition in a fused embedding-backward kernel that only corrupts a few floats, which were quickly diluted by gradient aggregation. We observe that the diagnosis was misled due to the lack of spatial and temporal localization. Logs offered little clue about which kernel, device, or communication path was at fault. When symptoms (gradient-norm spike) appeared, they surfaced in different training steps and downstream layers across reruns. Engineers focused debugging efforts on the forward step where the symptom showed up, while the race occurred in the previous backward step.

OpGuard pinpoints the earliest divergent operator. Across all evaluated failures, this operator lied inside the faulty kernel or its first consumer, which is precise enough for engineers to finish localization with a few quick checks such as selective gradient overrides or microbenchmarks.

In this case, when the same bug reappeared in a later run around Step 1535 instead of Step 3081, engineers did not need to identify that step manually. OpGuard surfaced the earliest boundary divergence, independent of where the symptom manifested. An early version of OpGuard did not wrap autograd-launched fused kernels, so it reported the next forward read of the corrupted embedding weights. Even this one-hop signal was decisive: adding a one-line gradient hook restored determinism, confirming `embedding_backward` as the origin. Pinpointing the fused kernel required only one additional targeted test. In current deployments, where OpGuard wraps backward kernels, it would directly report the `embedding_backward` kernel. This PyTorch kernel introduced the bug seven years ago and is in active use [74].

Case Study: (Bug 12). This case illustrates a different challenge: the first-difference boundary is precise, but the causal mechanism is non-obvious. OpGuard isolated the earliest divergence to a single `torch.scatter_` inside the expert-group repacking logic, where a one-bit SDC misrouted a token to the wrong expert group. Yet the externally visible symptom (a row-level shape mismatch in the *backward* pass) appeared unrelated to a forward-pass dispatch error, making it difficult for engineers to mentally connect the two.

What made the case tractable is that OpGuard wraps more than the first diff. After OpGuard reported the divergent scatter, engineers simply walked the trace downstream and found a shape mismatch in the expert-aggregation op roughly 30 operators later. This hop-by-hop propagation made the causal chain explicit and ruled out alternative hypotheses such

Case	Symptom	Root Cause	Manual	OpGuard
Bug 1	Loss diverges after ~3080; clean grad-norm	Racy torch.embedding backward kernel	5 days	15 min
Bug 4	Stream race in FP32 RMSNorm	CUDA stream race	5 days	5 min
Bug 9	Undeterministic backward kernel	Deterministic flag unset	3 days	5 min
Bug 10	Reduce-scatter mismatch by different nccl topology	Topology-dependent FP associativity	3 days	10 min
Bug 12	Shape mismatch on XPU pool A	Boolean scatter SDC	14 days	30 min
Bug 14	Routing drift in MoE	Histogram bin mismatch	4 hours	5 min
Bug 16	Fused embedding backward drift	Kernel race condition	8 days	5 min
Bug 17	Loss mismatch under activation offload	CUDA stream misuse	2 days	20 min
Bug 18	Backward attention drift	Triton kernel missing barrier	4 days	10 min
Bug 19	Incorrect QKV projections	Implicit contiguity assumption	5 days	5 min
Bug 20	Divergence after resume	Optimizer step var overwrite	3 days	5 min

Table 2. Subset of evaluated production bugs (full set in Table 3). OpGuard numbers are estimated debugging time provided by engineers.

as scheduling races or misconfigured parallelism. Most of the 30 min debugging time was therefore spent stress-testing the operator to confirm SDC and validating the propagation path.

False Positives. Across all 20 failures, OpGuard surfaced no false positives after initial adaptation. When first applied to a new model or backend, engineers typically encounter 2–10 benign differences arising from predictable patterns: unused padded regions [2], pinned-memory `copy_` operations that cannot be wrapped as normal kernels [72] (*e.g.*, host-to-device copies from dataloader buffers), internal helper buffers, GPU-side performance counters, or scratch workspaces with undefined contents. None affects model outputs.

Distinguishing these from real corruptions is quick: engineers confirm an FP in 2–3 minutes using OpGuard's visualizer. Each FP corresponds to a one-line filter rule; once added, the pattern never reappears for future runs on the same workflow. Because these rules accumulate over time, the FP rate quickly converges to zero.

Research Baselines. Besides the debugging tools engineers used in manual debugging, we evaluate four research solutions: DeepLocalize [89], DeepDiagnosis [88], DeepFD [9], and TTrace [34]. The first three systems are state-of-the-art for single-device architectural or hyperparameter bugs [32, 48], but they detect only 2 of 20 cases (Bug 3 and Bug 15). These are the only incidents whose root causes manifest as deterministic, single-layer faults in a standard computational graph (the failure model these tools are designed for).

TTrace is the closest prior system performing differential *numerical* comparison. It targets distributed training but relies on (1) a trusted single-device reference, (2) threshold-based numerical comparison, and (3) deterministic operator replays. Only 4 of our 20 cases have a native single-GPU path; for the others, we supply a multi-device reference via our deterministic replay to give TTrace the most faithful possible environment.

Under this adapted setup, TTrace successfully flags 11 cases. Its blame windows, however, remain coarse (on average 6.91 modules and 45.64 kernels), and it fails on all cases involving extremely small corruptions, nondeterministic kernels, or intermittent multi-stream races. These limitations stem from TTrace's perturbation-based calibration: numerical deviations smaller than the learned perturbation envelope are treated as benign, causing small drifts from bugs to be missed or attributed far downstream. In cases such as Bug 9 and Bug 16, only a few floating-point entries in a gradient shard were corrupted before being attenuated by reductions-well below TTrace's detection threshold. Moreover, TTrace's heavy tensor-dump instrumentation perturbs execution enough that some bug races become harder to reproduce.

6.3 Open-source Bug Diagnosis

We further evaluate OpGuard on 10 real open-source issues drawn from Megatron-LM, DeepSpeed, GPT-Neox, and HuggingFace Transformers. [4, 21, 79, 90] These cases span kernel nondeterminism, stream-ordering races, MoE dispatcher inconsistencies, and cache-management faults, etc. For each issue, we build the comparison from the smallest reproducer available in the public report: the reference is either a self-replay, a known-good configuration, a single-GPU/TP-simulator run, a stable library version, or an equivalent API or framework path. Both executions use the same inputs, seeds, and checkpoint when applicable, and OpGuard establishes alignment by retaining only model-level boundaries observed in both traces. OpGuard successfully produces a precise first-difference operator for 8 of 10 cases, including four long-standing unresolved issues where prior discussions had converged on incorrect or incomplete explanations. The surfaced operators, as summarized in Table 4, directly identify the corrupted tensor or boundary and map cleanly to the underlying faulty kernel, cache rule, or collective implementation. The remaining cases, where OpGuard does not yet apply cleanly, are discussed in §7.

6.4 Silent Data Corruption (SDC) Detection

OpGuard initially targets software-level correctness faults. Once deployed at scale, we observed a useful side effect: the same tensor-consistency checks that localize software bugs also surface silent data corruptions in hardware. When a single device begins producing inconsistent tensors, OpGuard flags

that divergence just like any other bug, and the first-difference report naturally attributes the fault to a specific computation or communication operator on that device. This operator-level attribution is valuable for subsequent vendor diagnostics, which typically only report that a card is "unhealthy" without identifying which workload or kernel exposes the fault.

We evaluate this emergent capability of OpGuard in its online and offline SDC-detection modes. In *online* mode, it runs alongside long-running production jobs and isolates suspect machines as soon as inconsistencies are observed. In *offline* mode, it replays training traces and pinpoints the exact operation where divergence first occurs. Throughout our evaluation, we compare OpGuard against the existing production baseline of vendor-provided device health checks (*e.g.*, NVIDIA's EUD [53], AMD's ROCm Validation Suite (RVS) [1], and analogous pre-flight diagnostics on other accelerators). These checks are executed *before* a training job is officially launched, and a machine is removed from service only if one of these diagnostics reports an error. Machines that pass these vendor diagnostics are considered healthy and are eligible to run large-scale training workloads.

To date, OpGuard has detected *21 distinct SDC machines* that all successfully passed these pre-flight health checks. In other words, these 21 machines were invisible to the baseline but were flagged by OpGuard during actual training. Each detection was subsequently validated through targeted stress testing and on-device EDC (error detection/correction) verification [100], and the confirmed faulty hardware was reported to engineers for further diagnosis and remediation.

6.5 Ablation Study

We evaluate how OpGuard components affect localization. For each production case, we compare the full configuration's localization index i_{full} with an ablated variant i_{abl}, reporting $\Delta\text{ops} = i_{\text{abl}} - i_{\text{full}}$. Thus $\Delta\text{ops} = 0$ means identical localization, while positive or negative values indicate later or earlier detections. Operators are globally ordered using machine and GPU timestamps.

We study five settings: (1) reducing instrumentation to 40 manual probes; (2–3) removing determinism controls and applying numerical tolerances of 10^{-5} and 10^{-3}; (4) replacing the XOR signature with a scalar `sum`; and (5) using a synchronous tracer that inserts global device synchronizations.

Figure 6 summarizes the results. Each column shows the twenty bugs under a different ablation, with markers indicating Δops clipped to $[-20, 20]$. The value above each column reports how many bugs remain perfectly localized ($\Delta\text{ops} = 0$).

We make three findings. First, reducing instrumentation severely degrades localization: only 2/20 bugs remain pinpointed, and many detections drift by tens of operators. Second, removing determinism controls substantially increases drift, especially at tolerance 10^{-3}, which yields only 7/20 exact localizations and several large outliers. Third, sum-based checking appears to work on easier cases but misses

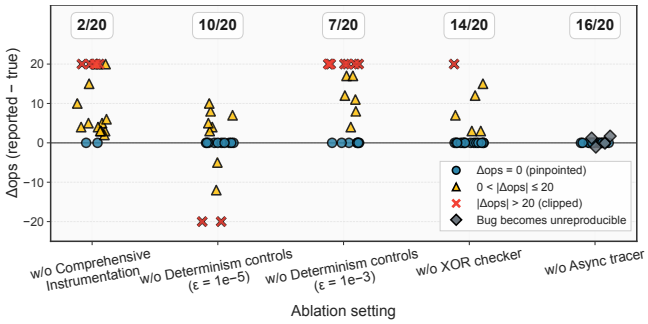


Figure 6. Localization regression under ablations.

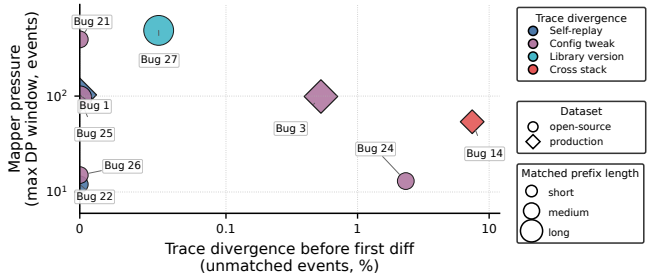


Figure 7. Alignment remains robust under real trace divergence.

subtle one-token or one-bit corruptions, leading to delayed detections; only 14/20 bugs remain precisely localized. The synchronous tracer preserves 16/20 cases but causes four bugs to become unreproducible due to perturbing concurrency.

6.6 Alignment Robustness to Trace Divergence

The mapper (§4.5) targets aligning executions that are comparable but typically do not produce identical traces due to fused kernels, helper operations, or local schedules. We evaluate whether the mapper can tolerate these differences and still identify the expected first-difference operator.

We sample nine paired traces: three production replays and six open-source replays. The sample covers self-replay, configuration tweaks, library-version changes, and cross-stack comparisons. For each pair, we evaluate alignment only over the comparable prefix preceding the expected first difference. This excludes subsequent divergence caused by the bug itself.

We measure alignment difficulty in three ways: (1) *trace-length ratio* captures the difference in event counts between the two comparable prefixes; (2) *unmatched-event fraction* measures the fraction of boundary events that the mapper classifies as insertions or deletions; (3) mapper pressure, measured as *maximum DP window*, the largest region that the mapper must align after partitioning the traces with anchors. A large window indicates that anchors are sparse or ambiguous, forcing the mapper to solve a harder local alignment problem.

Figure 7 relates the unmatched-event fraction to the maximum DP window. Marker shape distinguishes production and open-source traces, color denotes the source of trace divergence, and marker size denotes matched-prefix length.

The mapper reaches the expected first-difference operator in *all nine trace pairs*. Figure 7 shows that real trace divergence

is present but remains alignable. The median trace-length ratio is 1.005×, and the largest is 1.048×. The median unmatched-event fraction is 0; seven of nine cases are below 2.4%. Bug 14 has the largest fraction, 7.43%, but the mapper still identifies the expected first difference.

Anchor construction keeps the alignment problem local. The median anchor density is 15.1% (minimum 9.5%), and the median maximum DP window is 96 events. Two open-source pairs require larger windows (390 and 484 events), but these remain small relative to traces containing thousands of events.

The traces show two trends. First, production traces require smaller DP windows than open-source traces (54–103 events versus wider outliers). This matches our experience: their custom kernels and fused model-level operators often serve as distinctive anchors, whereas open-source framework traces contain longer sequences of generic operations such as `view`, `copy_`, `linear`, and helper bookkeeping calls. Second, unmatched events tend to increase as the two executions' implementations differ more: self-replay has little, configuration and library changes add localized gaps, and the cross-stack case has the largest unmatched fraction.

6.7 Aligned Coverage

A runtime tracer is only useful if it *sees* the kernels where failures occur. If a CUDA kernel is missing from the trace, any bug inside that kernel may be incorrectly attributed to its neighbors, sending engineers to debug the wrong part of the program. We therefore evaluate *aligned kernel coverage*: the fraction of profiler-recorded CUDA kernels that OpGuard associates with a model-level operator.

To measure this, we run each workload with OpGuard under `torch.profiler`, which provides the full set of kernels launched during the step; OpGuard's trace contains kernel slices grouped by high-level ops. Coverage is the fraction of baseline kernels that appear inside a OpGuard operator boundary. We evaluate eight training pipelines spanning LLM pretraining (with and without `torch.compile` [5]), VLM pretraining [75], RL actor/critic/rollout training [36], and two open-source systems (Megatron [79] and VERL [78]). These capture both eager and graph-based execution modes.

Across all workloads (Figure 8), OpGuard achieves at least 95% aligned kernel coverage, with many workloads reaching 98%–100%. The remaining small gaps mainly occur when a single Python source line launches multiple CUDA ops; OpGuard attaches one probe per line, so only the first launch is tagged. These launches occur contiguously in practice and do not affect localization: the divergence boundary still lands on the correct kernel or its immediate consumer.

6.8 Practicality and Overhead

Figure 9 reports the runtime overhead of OpGuard across pretraining and RL workloads from 8 to 512 XPUs. Online SDC detection adds effectively no cost (≈ 1.00×–1.01×); trusted mode incurs ≈ 1.25×–1.45× overhead; and full mode costs

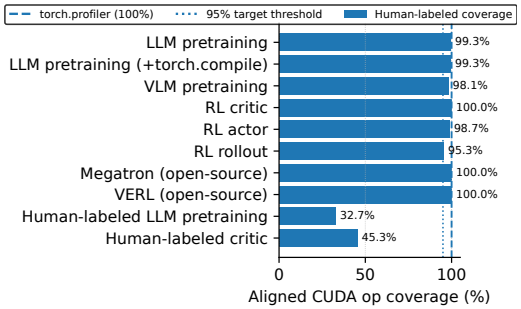


Figure 8. Aligned kernel coverage of OpGuard across workloads.

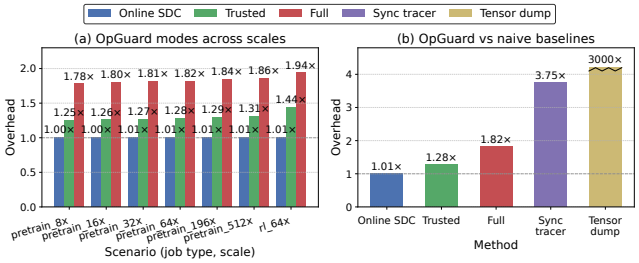


Figure 9. Runtime overhead under different deployment modes.

≈ 1.8×–1.95×. RL workloads show slightly higher slowdowns because they contain more inference- and control-heavy paths where checker kernels amortize less, but overhead is essentially flat with scale. Figure 9(b) further shows that even OpGuard's full mode is far cheaper than conventional debugging: a global-sync tracer slows training by ≈ 3.75×, and full tensor dumping would impose an infeasible ∼ 3000× cost.

The overhead gap follows from the amount of instrumentation each mode enables. Online SDC detection checks sparse communication boundaries, so its added XOR kernels are tiny compared with the data already in flight. Trusted mode avoids many checker launches on high-confidence primitives such as GEMMs and basic elementwise ops, which would otherwise dominate runtime; this balance makes it our standard configuration for production debugging. Full mode pays more for exhaustive operator coverage and is mainly used for stubborn or deeply latent bugs.

7 Discussions

Engineers consistently emphasized that precise, operator-level evidence is what accelerates debugging. Ambiguous signals rarely justify deep investigation. OpGuard's concrete divergence boundary provides such evidence. While fully automated debugging is ideal, engineers also noted that the first-diff localization is typically sufficient: once OpGuard surfaces the earliest corrupted tensor or kernel, they can reason about the root cause using their own domain expertise.

OpGuard relies on the two executions sharing a comparable model prefix: they must reach the same model-level boundaries with the same logical inputs, RNG choices, numerical policies, and distributed reduction rules. If a bug occurs before this prefix, or a version change intentionally changes one of these contracts, OpGuard can only report the earliest

downstream operator both executions reach in a comparable way. In Bug 29, for example, an early preprocessing regression changed model inputs, so the first alignable model operator was already affected. Such cases arise mainly from implementation differences in high-level preprocessing rather than faults in the model or runtime layers OpGuard targets.

Downscaled replay also changes the execution environment. Reducing a job from thousands of machines to hundreds may remove the faulty device that exposed an SDC, and smaller runs often create different memory pressure, allocator behavior, and scheduling. We observed this in one SDC incident: the failure appeared in a 1024-machine production run but did not reproduce in a 512-machine replay because the replay excluded the faulty machine. Thus, downscaled replay is best viewed as a debugging accelerator when the bug still reproduces, not a proof that a bug is absent. If the replay no longer reaches a comparable divergent execution, OpGuard cannot localize or explain that particular fault.

8 Related Work

Bug Detection and Monitoring for ML Training. Empirical studies have shown the prevalence of bugs and reliability issues in ML training [26, 31, 37, 93, 94, 97], which motivate work to detect bugs. Differential-testing tools [41, 62, 64, 92] expose inconsistencies in models, libraries, and compilers. Specialized analyzers such as PyTea [33], RANUM [37], and CUDASanitizer [54] aim to catch shape, numerical, or concurrency defects. TrainCheck [35] learns invariants to monitor training and detect silent errors at runtime. These solutions focus on detection and not diagnosis.

Fault Localization in ML Programs. Other work aims to diagnose faults by identifying specific model components or code regions responsible for erroneous behavior. DeepLocalize [89] correlates runtime metrics with suspicious layers. DeepDiagnosis [88] infers fault types and localizes them to model components or configuration issues. DeepFD [9] learns classifiers over rich runtime features to predict fault types and link them to faulty source statements. UMLAUT [76], Theia [45], and NeuraLint [49] combine structural analyses with behavioral signals to detect architectural defects or hyperparameter issues, while neuron/operator ranking methods [25] inspect trained models for internal inconsistencies. These approaches operate primarily at the framework or Python-operator level and assume single-node training environments.

Equivalence Checking. TrainVerify [43] uses symbolic reasoning to verify a parallel execution plan is mathematically equivalent to its single-device logical definition. It targets statically eliminating parallelization related bugs, and does not concern implementation or runtime behavior. TTrace [34] records full intermediate tensors and compares them numerically against a trusted single-device reference implementation. Such recording incurs high overhead and the tolerances its comparisons rely on are brittle. OpGuard targets debugging large-scale LLM training. It uses two production executions rather than requiring a single-device reference, which is often unavailable for large models. OpGuard introduces bitwise alignment and controls nondeterminism, while designing techniques to make the alignment robust across production-grade optimizations such as fusion, asynchrony, and graph capture.

Comparative Debugging. OpGuard builds on a long tradition of comparative debugging. Delta debugging [95] and related techniques isolate the root cause to a failure-inducing input or change, a cause-effect chain [96], a statistically suspicious predicate [38], or instruction sequence [99]. OpGuard shares the same high-level principle. Its contribution is to make this strategy precise for LLM training, where executions are expensive, tensor states are enormous, benign nondeterminism is pervasive, and implementation traces differ significantly. It introduces bitwise alignment with stable model-level boundaries, exact tensor equality, and certified-prefix localization.

Deterministic Replay. Deterministic multithreading [6, 7, 14–16, 42, 46, 47] and replay [3, 23, 30, 58, 61, 85] make conventional software executions reproducible by controlling thread schedules, synchronization, system-call results, and external inputs. These systems inspired OpGuard's use of controlled nondeterminism, but their goals differ. They seek to constrain whole executions by imposing strict scheduling discipline. In production LLM training, full-stack deterministic scheduling is rarely practical: it can impose high overhead and is fragile across heterogeneous hardware, libraries, drivers, compilers, *etc.* OpGuard instead controls only the sources of variation that would make bitwise alignment ill-posed while leaving residual schedule-sensitive behavior visible as a bug signal.

9 Conclusion

Debugging correctness issues in LLM training is notoriously difficult. We propose bitwise alignment as a precise correctness oracle and debugging primitive for addressing this challenge. We realize the ideas in OpGuard, which controls avoidable non-determinism, discovers semantic-stable boundaries, fingerprints them, and uses schedule-tolerant alignment with prefix certification to surface the divergence. OpGuard has been deployed across production pre-training and post-training pipelines, and helped engineers debug various difficult failures, reducing debugging time from days to minutes.

Acknowledgments

We thank our shepherd and anonymous reviewers for feedback that improved the paper, and Chandler Nie, Zuquan Song, Dongyu Xu, and Weiqi Feng for valuable contributions and feedback. This work was supported in part by NSF grants CNS-2317698, CNS-2317751, and CCF-2318937.

References

[1] Advanced Micro Devices, Inc. 2025. *ROCm Validation Suite (RVS) Documentation*. Advanced Micro Devices, Inc. https://rocm.docs.

amd.com/projects/ROCmValidationSuite/en/latest/ RVS 1.3.0 Documentation, accessed June 6, 2026.

[2] Osayamen Jonathan Aimuyo, Byungsoo Oh, and Rachee Singh. 2025. FlashMoE: Fast Distributed MoE in a Single Kernel. arXiv:2506.04667 [cs.DC] https://arxiv.org/abs/2506.04667 See Section 3.2.1 "In-place Padding for Payload Efficiency"; accessed June 6, 2026.

[3] Gautam Altekar and Ion Stoica. 2009. ODR: Output-Deterministic Replay for Multicore Debugging. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles* (Big Sky, Montana, USA) *(SOSP '09)*. Association for Computing Machinery, New York, NY, USA, 193–206. https://doi.org/10.1145/1629575.1629594

[4] Alex Andonian, Quentin Anthony, Stella Biderman, Sid Black, Preetham Gali, Leo Gao, Eric Hallahan, Josh Levy-Kramer, Connor Leahy, Lucas Nestler, Kip Parker, Michael Pieler, Jason Phang, Shivanshu Purohit, Hailey Schoelkopf, Dashiell Stander, Tri Songz, Curt Tigges, Benjamin Thérien, Phil Wang, and Samuel Weinbach. 2023. *GPT-NeoX: Large Scale Autoregressive Language Modeling in PyTorch*. EleutherAI. https://doi.org/10.5281/zenodo.5879544

[5] Jason Ansel, Edward Yang, Horace He, Natalia Gimelshein, Animesh Jain, Michael Voznesensky, Bin Bao, Peter Bell, David Berard, Evgeni Burovski, Geeta Chauhan, Anjali Chourdia, Will Constable, Alban Desmaison, Zachary DeVito, Elias Ellison, Will Feng, Jiong Gong, Michael Gschwind, Brian Hirsh, Sherlock Huang, Kshiteej Kalambarkar, Laurent Kirsch, Michael Lazos, Mario Lezcano, Yanbo Liang, Jason Liang, Yinghai Lu, C. K. Luk, Bert Maher, Yunjie Pan, Christian Puhrsch, Matthias Reso, Mark Saroufim, Marcos Yukio Siraichi, Helen Suk, Shunting Zhang, Michael Suo, Phil Tillet, Xu Zhao, Eikan Wang, Keren Zhou, Richard Zou, Xiaodong Wang, Ajit Mathews, William Wen, Gregory Chanan, Peng Wu, and Soumith Chintala. 2024. PyTorch 2: Faster Machine Learning Through Dynamic Python Bytecode Transformation and Graph Compilation. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2* (La Jolla, CA, USA) *(ASPLOS '24)*. Association for Computing Machinery, New York, NY, USA, 929–947. https://doi.org/10.1145/3620665.3640366

[6] Amittai Aviram, Shu-Chun Weng, Sen Hu, and Bryan Ford. 2010. Efficient system-enforced deterministic parallelism. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation* (Vancouver, BC, Canada) *(OSDI '10)*. USENIX Association, USA, 193–206. https://www.usenix.org/conference/osdi10/efficient-system-enforced-deterministic-parallelism

[7] Tom Bergan, Owen Anderson, Joseph Devietti, Luis Ceze, and Dan Grossman. 2010. CoreDet: a compiler and runtime system for deterministic multithreaded execution. In *Proceedings of the Fifteenth International Conference on Architectural Support for Programming Languages and Operating Systems* (Pittsburgh, Pennsylvania, USA) *(ASPLOS XV)*. Association for Computing Machinery, New York, NY, USA, 53–64. https://doi.org/10.1145/1736020.1736029

[8] Adam Betts, Nathan Chong, Alastair Donaldson, Shaz Qadeer, and Paul Thomson. 2012. GPUVerify: a verifier for GPU kernels. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications* (Tucson, Arizona, USA) *(OOPSLA '12)*. Association for Computing Machinery, New York, NY, USA, 113–132. https://doi.org/10.1145/2384616.2384625

[9] Jialun Cao, Meiziniu Li, Xiao Chen, Ming Wen, Yongqiang Tian, Bo Wu, and Shing-Chi Cheung. 2022. DeepFD: automated fault diagnosis and localization for deep learning programs. In *Proceedings of the 44th International Conference on Software Engineering* (Pittsburgh, Pennsylvania) *(ICSE '22)*. Association for Computing Machinery, New York, NY, USA, 573–585. https://doi.org/10.1145/3510003.3510099

[10] Adam Casson. 2023. Transformer FLOPs — Adam Casson's Blog. (2023). https://adamcasson.com/posts/transformer-flops online; accessed June 6, 2026.

[11] Tianqi Chen, Bing Xu, Chiyuan Zhang, and Carlos Guestrin. 2016. Training Deep Nets with Sublinear Memory Cost. arXiv:1604.06174 [cs.LG] https://arxiv.org/abs/1604.06174

[12] Aakanksha Chowdhery, Sharan Narang, Jacob Devlin, Maarten Bosma, Gaurav Mishra, et al. 2023. PaLM: Scaling Language Modeling with Pathways. *Journal of Machine Learning Research* 24, 240 (2023), 1–113. https://www.jmlr.org/papers/v24/22-1144.html See §5 on "Bitwise determinism" for engineering practices at 8B, 62B, and 540B scales.

[13] James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, J. J. Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, Wilson Hsieh, Sebastian Kanthak, Eugene Kogan, Hongyi Li, Alexander Lloyd, Sergey Melnik, David Mwaura, David Nagle, Sean Quinlan, Rajesh Rao, Lindsay Rolig, Yasushi Saito, Michal Szymaniak, Christopher Taylor, Ruth Wang, and Dale Woodford. 2013. Spanner: Google's Globally Distributed Database. *ACM Trans. Comput. Syst.* 31, 3, Article 8 (Aug. 2013), 22 pages. https://doi.org/10.1145/2491245

[14] Heming Cui, Jiri Simsa, Yi-Hong Lin, Hao Li, Ben Blum, Xinan Xu, Junfeng Yang, Garth A. Gibson, and Randal E. Bryant. 2013. Parrot: A Practical Runtime for Deterministic, Stable, and Reliable Threads. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles* (Farminton, Pennsylvania) *(SOSP '13)*. Association for Computing Machinery, New York, NY, USA, 388–405. https://doi.org/10.1145/2517349.2522735

[15] Heming Cui, Jingyue Wu, John Gallagher, Huayang Guo, and Junfeng Yang. 2011. Efficient deterministic multithreading through schedule relaxation. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles* (Cascais, Portugal) *(SOSP '11)*. Association for Computing Machinery, New York, NY, USA, 337–351. https://doi.org/10.1145/2043556.2043588

[16] Heming Cui, Jingyue Wu, Chia-Che Tsai, and Junfeng Yang. 2010. Stable deterministic multithreading through schedule memoization. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation* (Vancouver, BC, Canada) *(OSDI '10)*. USENIX Association, USA, 207–221. https://www.usenix.org/conference/osdi10/stable-deterministic-multithreading-through-schedule-memoization

[17] Daniel and Michael Han. 2024. Fixing All Gemma Bugs. https://unsloth.ai/blog/gemma-bugs Accessed: June 6, 2026.

[18] Dao-AILab. 2025. FlashAttention. https://github.com/Dao-AILab/flash-attention See README §2.4 "ALiBi attention with linear bias—deterministic backward pass"; accessed June 6, 2026.

[19] DeepSeek-AI, Aixin Liu, Bei Feng, Bing Xue, Bingxuan Wang, Bochao Wu, et al. 2025. DeepSeek-V3 Technical Report. arXiv:2412.19437 [cs.CL] https://arxiv.org/abs/2412.19437

[20] DeepSpeed Team. 2025. Megatron-DeepSpeed: Reproducibility. https://github.com/deepspeedai/Megatron-DeepSpeed/blob/3e1da1fbb226fd4d19aad33afcb33c2f6ed0eb26/README.md#reproducibility Commit 3e1da1f.

[21] deepspeedai. 2020–2025. DeepSpeed: Deep Learning Optimization Library. https://github.com/deepspeedai/DeepSpeed GitHub repository, accessed December 11, 2025.

[22] Harish D. Dixit. 2022. Detecting silent errors in the wild: Combining two novel approaches to quickly detect silent data corruptions at scale. https://engineering.fb.com/2022/03/17/production-engineering/silent-errors/ Accessed June 6, 2026.

[23] George W. Dunlap, Samuel T. King, Sukru Cinar, Murtaza A. Basrai, and Peter M. Chen. 2002. ReVirt: Enabling Intrusion Analysis Through Virtual-Machine Logging and Replay. In *Proceedings of the Fifth Symposium on Operating Systems Design and Implementation (OSDI '02)*. Boston, MA, 14 pages. https://www.usenix.org/conference/osdi-02/revirt-enabling-intrusion-analysis-through-virtual-machine-logging-and-replay

[24] edubart. 2025. machine-kernels-llama2.c. https://github.com/edubart/machine-kernels-llama2.c GitHub repository; minimal LLaMA2 kernels with deterministic variants; accessed June 6, 2026.

[25] Hasan Ferit Eniser, Simos Gerasimou, and Alper Sen. 2019. DeepFault: Fault Localization for Deep Neural Networks. In *Fundamental Approaches to Software Engineering*, Reiner Hähnle and Wil van der Aalst (Eds.). Springer International Publishing, Cham, 171–191. https://link.springer.com/chapter/10.1007/978-3-030-16722-6_10

[26] Yanjie Gao, Yichen He, Xinze Li, Bo Zhao, Haoxiang Lin, Yoyo Liang, Jing Zhong, Hongyu Zhang, Jingzhou Wang, Yonghua Zeng, Keli Gui, Jie Tong, and Mao Yang. 2024. An Empirical Study on Low GPU Utilization of Deep Learning Jobs. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering* (Lisbon, Portugal) *(ICSE '24)*. Association for Computing Machinery, New York, NY, USA, Article 96, 13 pages. https://doi.org/10.1145/3597503.3639232

[27] Yanjie Gao, Ruiming Lu, Haoxiang Lin, and Yueguo Chen. 2025. An Empirical Study of Issues in Large Language Model Training Systems. In *FSE 2025*. ACM. https://dl.acm.org/doi/10.1145/3696630.3728538 The ACM International Conference on the Foundations of Software Engineering, Industry Track.

[28] gottbrath. 2025. Meta PyTorch Team 2025 H2 Roadmaps. https://dev-discuss.pytorch.org/t/meta-pytorch-team-2025-h2-roadmaps/3184 Accessed: June 6, 2026.

[29] Aaron Grattafiori, Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, et al. 2024. The Llama 3 Herd of Models. arXiv:2407.21783 [cs.AI] https://arxiv.org/abs/2407.21783

[30] Zhenyu Guo, Xi Wang, Jian Tang, Xuezheng Liu, Zhilei Xu, Ming Wu, M. Frans Kaashoek, and Zheng Zhang. 2008. R2: An Application-Level Kernel for Record and Replay. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation* (San Diego, California) *(OSDI '08)*. USENIX Association, USA, 193–208. https://www.usenix.org/conference/osdi-08/r2-application-level-kernel-record-and-replay

[31] Qinghao Hu, Zhisheng Ye, Zerui Wang, Guoteng Wang, Meng Zhang, Qiaoling Chen, Peng Sun, Dahua Lin, Xiaolin Wang, Yingwei Luo, Yonggang Wen, and Tianwei Zhang. 2024. Characterization of large language model development in the datacenter. In *Proceedings of the 21st USENIX Symposium on Networked Systems Design and Implementation* (Santa Clara, CA, USA) *(NSDI'24)*. USENIX Association, USA, Article 39, 21 pages. https://www.usenix.org/conference/nsdi24/presentation/hu

[32] Nargiz Humbatova, Jinhan Kim, Gunel Jahangirova, Shin Yoo, and Paolo Tonella. 2025. An empirical study of fault localisation techniques for deep neural networks. *Empirical Software Engineering* 30 (06 2025). https://doi.org/10.1007/s10664-025-10657-7

[33] Ho Young Jhoo, Sehoon Kim, Woosung Song, Kyuyeon Park, DongKwon Lee, and Kwangkeun Yi. 2021. A Static Analyzer for Detecting Tensor Shape Errors in Deep Neural Network Training Code. arXiv:2112.09037 [cs.LG] https://arxiv.org/abs/2112.09037

[34] Haitian Jiang, Shaowei Zhu, Zhen Zhang, Zhenyu Song, Xinwei Fu, Zhen Jia, Yida Wang, and Jinyang Li. 2025. TTrace: Lightweight Error Checking and Diagnosis for Distributed Training. arXiv:2506.09280 [cs.DC] https://arxiv.org/abs/2506.09280

[35] Yuxuan Jiang, Ziming Zhou, Boyu Xu, Beijie Liu, Runhui Xu, and Peng Huang. 2025. Training with confidence: catching silent errors in deep learning training with automated proactive checks. In *Proceedings of the 19th USENIX Symposium on Operating Systems Design and Implementation* (Boston, MA, USA) *(OSDI '25)*. USENIX Association, USA, Article 18, 313-329 pages. https://www.usenix.org/conference/osdi25/presentation/jiang

[36] Vijay R. Konda and John N. Tsitsiklis. 1999. Actor-critic algorithms. *Advances in Neural Information Processing Systems* 12 (1999), 1008–1014. https://proceedings.neurips.cc/paper_files/paper/1999/file/6449f44a102fde848669bdd9eb6b76fa-Paper.pdf

[37] Linyi Li, Yuhao Zhang, Luyao Ren, Yingfei Xiong, and Tao Xie. 2023. Reliability Assurance for Deep Neural Network Architectures against Numerical Defects. In *Proceedings of the 45th International Conference on Software Engineering* (Melbourne, Victoria, Australia) *(ICSE '23)*. IEEE Press, 1827–1839. https://doi.org/10.1109/ICSE48619.2023.00156

[38] Ben Liblit, Mayur Naik, Alice X. Zheng, Alex Aiken, and Michael I. Jordan. 2005. Scalable Statistical Bug Isolation. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation* (Chicago, IL, USA) *(PLDI '05)*. Association for Computing Machinery, New York, NY, USA, 15–26. https://doi.org/10.1145/1065010.1065014

[39] William Lin. 2019. GPU Burn: A Multi-GPU CUDA Stress Test Tool. https://github.com/wilicc/gpu-burn GitHub repository, Accessed: June 6, 2026.

[40] Jiacai Liu, Yingru Li, Yuqian Fu, Jiawei Wang, Qian Liu, and Yu Shen. 2025. *When Speed Kills Stability: Demystifying RL Collapse from the Training–Inference Mismatch*. https://yingru.notion.site/RL-Collapse-271211a558b7808d8b12d403fd15edda Accessed June 6, 2026.

[41] Jiawei Liu, Jinkun Lin, Fabian Ruffy, Cheng Tan, Jinyang Li, Aurojit Panda, and Lingming Zhang. 2023. NNSmith: Generating Diverse and Valid Test Cases for Deep Learning Compilers. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2* (Vancouver, BC, Canada) *(ASPLOS 2023)*. Association for Computing Machinery, New York, NY, USA, 530–543. https://doi.org/10.1145/3575693.3575707

[42] Tongping Liu, Charlie Curtsinger, and Emery D. Berger. 2011. Dthreads: Efficient Deterministic Multithreading. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles* (Cascais, Portugal) *(SOSP '11)*. Association for Computing Machinery, New York, NY, USA, 327–336. https://doi.org/10.1145/2043556.2043587

[43] Yunchi Lu, Youshan Miao, Cheng Tan, Peng Huang, Yi Zhu, Xian Zhang, and Fan Yang. 2025. TrainVerify: Equivalence-Based Verification for Distributed LLM Training. In *Proceedings of the ACM SIGOPS 31st Symposium on Operating Systems Principles* (Lotte Hotel World, Seoul, Republic of Korea) *(SOSP '25)*. Association for Computing Machinery, New York, NY, USA, 237–253. https://doi.org/10.1145/3731569.3764850

[44] Jeffrey Jian Ma, Hengzhi Pei, Leonard Lausen, and George Karypis. 2025. Understanding Silent Data Corruption in LLM Training. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)* (Vienna, Austria) *(ACL '25)*. Association for Computational Linguistics, 20372–20394. https://doi.org/10.18653/v1/2025.acl-long.996

[45] Ruchira Manke, Mohammad Wardat, Foutse Khomh, and Hridesh Rajan. 2025. Leveraging Data Characteristics for Bug Localization in Deep Learning Programs. *ACM Trans. Softw. Eng. Methodol.* 34, 6, Article 157 (July 2025), 29 pages. https://doi.org/10.1145/3708473

[46] Timothy Merrifield, Sepideh Roghanchi, Joseph Devietti, and Jakob Eriksson. 2019. Lazy Determinism for Faster Deterministic Multithreading. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems* (Providence, RI, USA) *(ASPLOS '19)*. Association for Computing Machinery, New York, NY, USA, 879–891. https://doi.org/10.1145/3297858.3304047

[47] Omar S. Navarro Leija, Kelly Shiptoski, Ryan G. Scott, Baojun Wang, Nicholas Renner, Ryan R. Newton, and Joseph Devietti. 2020. Reproducible Containers. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems* (Lausanne, Switzerland) *(ASPLOS '20)*. Association for Computing Machinery, New York, NY, USA, 167–182.

https://doi.org/10.1145/3373376.3378519

[48] Thanh-Dat Nguyen, Haoye Tian, Bach Le, Patanamon Thongtanunam, and Shane McIntosh. 2025. A Systematic Survey on Debugging Techniques for Machine Learning Systems. arXiv:2503.03158 [cs.SE] https://arxiv.org/abs/2503.03158

[49] Amin Nikanjam, Houssem Ben Braiek, Mohammad Mehdi Morovati, and Foutse Khomh. 2021. Automatic Fault Detection for Deep Learning Programs Using Graph Transformations. *ACM Trans. Softw. Eng. Methodol.* 31, 1, Article 14 (Sept. 2021), 27 pages. https://doi.org/10.1145/3470006

[50] NVIDIA 2024. *NVIDIA cuDNN Documentation*. NVIDIA. https://docs.nvidia.com/deeplearning/cudnn/archives/cudnn-892/api/index.html#cudnnDeterminism_t See Section 3.1.2.7 "cudnnDeterminism_t"; accessed June 6, 2026.

[51] NVIDIA 2025. *cuBLAS Documentation: Results Reproducibility*. NVIDIA. https://docs.nvidia.com/cuda/cublas/index.html#results-reproducibility See Section 2.1.4 "Results Reproducibility" for toolkit, architecture, SM-count, and multi-stream workspace constraints; accessed June 6, 2026.

[52] NVIDIA Corporation 2023. *CUDA C Programming Guide*. NVIDIA Corporation. https://docs.nvidia.com/cuda/cuda-c-programming-guide/ Describes CUDA streams, kernel scheduling, and asynchronous execution.

[53] NVIDIA Corporation 2023. *DCGM Diagnostics: End User Diagnostics (EUD)*. NVIDIA Corporation. https://docs.nvidia.com/datacenter/dcgm/3.1/user-guide/dcgm-diagnostics.html#end-user-diagnostics-eud NVIDIA DCGM Documentation, version 3.1, accessed June 6, 2026.

[54] NVIDIA Corporation 2025. *Compute Sanitizer Documentation*. NVIDIA Corporation. https://docs.nvidia.com/compute-sanitizer/index.html Accessed June 6, 2026.

[55] NVIDIA Corporation 2025. *CUDA Profiling Tools Interface (CUPTI)*. NVIDIA Corporation. https://developer.nvidia.com/cupti Accessed: June 6, 2026.

[56] NVIDIA Corporation 2025. *NCCL_ALGO — NCCL Algorithm Environment Variables*. NVIDIA Corporation. https://docs.nvidia.com/deeplearning/nccl/user-guide/docs/env.html#nccl-algo Accessed June 6, 2026.

[57] NVIDIA Corporation 2025. *NVIDIA DCGM Diagnostics: GPU Memory Plugin — NVIDIA DCGM Documentation*. NVIDIA Corporation. https://docs.nvidia.com/datacenter/dcgm/latest/user-guide/dcgm-diagnostics.html

[58] Robert O'Callahan, Chris Jones, Nathan Froyd, Kyle Huey, Albert Noll, and Nimrod Partush. 2017. Engineering Record and Replay for Deployability. In *Proceedings of the 2017 USENIX Conference on Usenix Annual Technical Conference* (Santa Clara, CA, USA) *(USENIX ATC '17)*. USENIX Association, USA, 377–389. https://www.usenix.org/conference/atc17/technical-sessions/presentation/ocallahan

[59] Jia Pan, Haoze Wu, Tanakorn Leesatapornwongsa, Suman Nath, and Peng Huang. 2024. Efficient Reproduction of Fault-Induced Failures in Distributed Systems with Feedback-Driven Fault Injection. In *Proceedings of the 30th Symposium on Operating Systems Principles* (Austin, TX, USA) *(SOSP '24)*. Association for Computing Machinery, New York, NY, USA, 18 pages. https://doi.org/10.1145/3694715.3695979

[60] Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. 2002. BLEU: a method for automatic evaluation of machine translation. In *Proceedings of the 40th Annual Meeting on Association for Computational Linguistics* (Philadelphia, Pennsylvania) *(ACL '02)*. Association for Computational Linguistics, USA, 311–318. https://doi.org/10.3115/1073083.1073135

[61] Soyeon Park, Yuanyuan Zhou, Weiwei Xiong, Zuoning Yin, Rini Kaushik, Kyu H. Lee, and Shan Lu. 2009. PRES: Probabilistic Replay with Execution Sketching on Multiprocessors. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles* (Big Sky, Montana, USA) *(SOSP '09)*. Association for Computing Machinery, New York, NY, USA, 177–192. https://doi.org/10.1145/1629575.1629593

[62] Kexin Pei, Yinzhi Cao, Junfeng Yang, and Suman Jana. 2019. DeepXplore: automated whitebox testing of deep learning systems. *Commun. ACM* 62, 11 (Oct. 2019), 137–145. https://doi.org/10.1145/3361566

[63] Perfetto Project. 2025. Perfetto: System profiling, app tracing and trace analysis. https://perfetto.dev/ Accessed June 6, 2026.

[64] Hung Viet Pham, Thibaud Lutellier, Weizhen Qi, and Lin Tan. 2019. CRADLE: Cross-backend validation to Detect and Localize bugs in Deep learning libraries. In *Proceedings of the 41st International Conference on Software Engineering* (Montreal, Quebec, Canada) *(ICSE '19)*. IEEE Press, 1027–1038. https://doi.org/10.1109/ICSE.2019.00107

[65] PyTorch Core Team 2025. *Distributed Communication Package — torch.distributed*. PyTorch Core Team. https://docs.pytorch.org/docs/stable/distributed.html See notes on "globally consistent execution order of collectives across ranks"; accessed June 6, 2026.

[66] PyTorch Core Team 2025. *Distributed Communication Package — torch.distributed*. PyTorch Core Team. https://docs.pytorch.org/docs/stable/distributed.html#synchronous-and-asynchronous-collective-operations Accessed June 6, 2026.

[67] PyTorch Core Team 2025. *Reproducibility — PyTorch: Controlling Sources of Randomness*. PyTorch Core Team. https://docs.pytorch.org/docs/2.8/notes/randomness.html#controlling-sources-of-randomness Accessed June 6, 2026.

[68] PyTorch Core Team 2025. *torch.allclose — PyTorch Documentation*. PyTorch Core Team. https://docs.pytorch.org/docs/stable/generated/torch.allclose.html Accessed June 6, 2026.

[69] PyTorch Core Team 2025. *torch.autograd.set_detect_anomaly — PyTorch Documentation*. PyTorch Core Team. https://docs.pytorch.org/docs/stable/autograd.html#torch.autograd.set_detect_anomaly

[70] PyTorch Core Team 2025. *torch.backends.cuda.matmul.allow_tf32 — PyTorch Documentation*. PyTorch Core Team. https://docs.pytorch.org/docs/stable/backends.html#torch.backends.cuda.matmul.allow_tf32 Accessed June 6, 2026.

[71] PyTorch Core Team 2025. *torch.get_rng_state — PyTorch Documentation*. PyTorch Core Team. https://docs.pytorch.org/docs/stable/generated/torch.get_rng_state.html Accessed June 6, 2026.

[72] PyTorch Core Team 2025. *torch.Tensor.copy_ — PyTorch Documentation*. PyTorch Core Team. https://docs.pytorch.org/docs/stable/generated/torch.Tensor.copy_.html Accessed June 6, 2026.

[73] PyTorch Core Team 2025. *torch.use_deterministic_algorithms — PyTorch Documentation*. PyTorch Core Team. https://docs.pytorch.org/docs/stable/generated/torch.use_deterministic_algorithms.html Accessed June 6, 2026.

[74] PyTorch developers. 2018. More efficient kernels that avoid deprecated shuffles in Embedding and LookupTable. https://github.com/pytorch/pytorch/commit/db14f3f33c8ddc9c910ca2188f8787dd81b97b52 File `aten/src/ATen/native/cuda/Embedding.cu`, lines 74–79.

[75] Alec Radford, Jong Wook Kim, Chris Hallacy, Aditya Ramesh, Gabriel Goh, Sandhini Agarwal, Girish Sastry, Amanda Askell, Pamela Mishkin, Jack Clark, Gretchen Krueger, and Ilya Sutskever. 2021. Learning Transferable Visual Models From Natural Language Supervision. In *Proceedings of the 38th International Conference on Machine Learning (Proceedings of Machine Learning Research, Vol. 139)*, Marina Meila and Tong Zhang (Eds.). PMLR, 8748–8763. https://proceedings.mlr.press/v139/radford21a.html

[76] Eldon Schoop, Forrest Huang, and Bjoern Hartmann. 2021. UMLAUT: Debugging Deep Learning Programs using Program Structure and Model Behavior. In *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems* (Yokohama, Japan) *(CHI '21)*. Association for Computing Machinery, New York, NY, USA, Article

310, 16 pages. https://doi.org/10.1145/3411764.3445538

[77] Shreya Shankar, Labib Fawaz, Karl Gyllstrom, and Aditya G. Parameswaran. 2023. Moving Fast With Broken Data. arXiv:2303.06094 [cs.DB] https://arxiv.org/abs/2303.06094

[78] Guangming Sheng, Chi Zhang, Zilingfeng Ye, Xibin Wu, Wang Zhang, Ru Zhang, Yanghua Peng, Haibin Lin, and Chuan Wu. 2025. HybridFlow: A Flexible and Efficient RLHF Framework. In *Proceedings of the Twentieth European Conference on Computer Systems* (Rotterdam, Netherlands) *(EuroSys '25)*. Association for Computing Machinery, New York, NY, USA, 1279–1297. https://doi.org/10.1145/3689031.3696075

[79] Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. 2020. Megatron-LM: Training Multi-Billion Parameter Language Models Using Model Parallelism. arXiv:1909.08053 [cs.CL] https://arxiv.org/abs/1909.08053

[80] Sho Takase, Shun Kiyono, Sosuke Kobayashi, and Jun Suzuki. 2025. Spike No More: Stabilizing the Pre-training of Large Language Models. In *Second Conference on Language Modeling (COLM '25)*. https://openreview.net/forum?id=52YBEzcl0l

[81] Mohamed Tarek Ibn Ziad, Sana Damani, Aamer Jaleel, Stephen W. Keckler, and Mark Stephenson. 2023. cuCatch: A Debugging Tool for Efficiently Catching Memory Safety Violations in CUDA Applications. *Proc. ACM Program. Lang.* 7, PLDI, Article 111 (June 2023), 24 pages. https://doi.org/10.1145/3591225

[82] TensorFlow 2025. `tf.config.experimental.enable_op_determinism` — *TensorFlow Documentation*. TensorFlow. https://www.tensorflow.org/api_docs/python/tf/config/experimental/enable_op_determinism Accessed June 6, 2026.

[83] TensorFlow Datasets Team 2021. *TFDS and Determinism*. TensorFlow Datasets Team. https://www.tensorflow.org/datasets/determinism Accessed June 6, 2026.

[84] TensorFlow Team 2025. *TensorFlow Debugger v2 — TensorFlow Documentation*. TensorFlow Team. https://www.tensorflow.org/tensorboard/debugger_v2

[85] Kaushik Veeraraghavan, Dongyoon Lee, Benjamin Wester, Jessica Ouyang, Peter M. Chen, Jason Flinn, and Satish Narayanasamy. 2011. DoublePlay: Parallelizing Sequential Logging and Replay. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems* (Newport Beach, California, USA) *(ASPLOS XVI)*. Association for Computing Machinery, New York, NY, USA, 15–26. https://doi.org/10.1145/1950365.1950370

[86] VI-HPS. 2024. MUST: MPI Usage Error Detection Tool. https://www.vi-hps.org/tools/must.html

[87] Borui Wan, Gaohong Liu, Zuquan Song, Jun Wang, Yun Zhang, Guangming Sheng, Shuguang Wang, Houmin Wei, Chenyuan Wang, Weiqiang Lou, Xi Yang, Mofan Zhang, Kaihua Jiang, Cheng Ren, Xiaoyun Zhi, Menghan Yu, Zhe Nan, Zhuolin Zheng, Baoquan Zhong, Qinlong Wang, Huan Yu, Jinxin Chi, Wang Zhang, Yuhan Li, Zixian Du, Sida Zhao, Yongqiang Zhang, Jingzhe Tang, Zherui Liu, Chuan Wu, Yanghua Peng, Haibin Lin, Wencong Xiao, Xin Liu, and Liang Xiang. 2025. Robust LLM Training Infrastructure at ByteDance. In *Proceedings of the ACM SIGOPS 31st Symposium on Operating Systems Principles* (Lotte Hotel World, Seoul, Republic of Korea) *(SOSP '25)*. Association for Computing Machinery, New York, NY, USA, 186–203. https://doi.org/10.1145/3731569.3764838

[88] Mohammad Wardat, Breno Dantas Cruz, Wei Le, and Hridesh Rajan. 2022. DeepDiagnosis: automatically diagnosing faults and recommending actionable fixes in deep learning programs. In *Proceedings of the 44th International Conference on Software Engineering* (Pittsburgh, Pennsylvania) *(ICSE '22)*. Association for Computing Machinery, New York, NY, USA, 561–572. https://doi.org/10.1145/3510003.3510071

[89] Mohammad Wardat, Wei Le, and Hridesh Rajan. 2021. DeepLocalize: Fault Localization for Deep Neural Networks. In *Proceedings of the 43rd International Conference on Software Engineering* (Madrid, Spain) *(ICSE '21)*. IEEE Press, 251–262. https://doi.org/10.1109/ICSE43902.2021.00034

[90] Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Clara Ma, Yacine Jernite, Julien Plu, Canwen Xu, Teven Le Scao, Sylvain Gugger, Mariama Drame, Quentin Lhoest, and Alexander M. Rush. 2020. Transformers: State-of-the-Art Natural Language Processing. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*. Association for Computational Linguistics, 38–45. https://www.aclweb.org/anthology/2020.emnlp-demos.6

[91] Mengdi Wu, Xinhao Cheng, Shengyu Liu, Chunan Shi, Jianan Ji, Man Kit Ao, Praveen Velliengiri, Xupeng Miao, Oded Padon, and Zhihao Jia. 2025. Mirage: a multi-level superoptimizer for tensor programs. In *Proceedings of the 19th USENIX Conference on Operating Systems Design and Implementation* (Boston, MA, USA) *(OSDI '25)*. USENIX Association, USA, Article 13, 18 pages. https://dl.acm.org/doi/10.5555/3767901.3767914

[92] Chenyuan Yang, Yinlin Deng, Jiayi Yao, Yuxing Tu, Hanchi Li, and Lingming Zhang. 2023. Fuzzing Automatic Differentiation in Deep-Learning Libraries. In *Proceedings of the 45th International Conference on Software Engineering* (Melbourne, Victoria, Australia) *(ICSE '23)*. IEEE Press, 1174–1186. https://doi.org/10.1109/ICSE48619.2023.00105

[93] Yilin Yang, Tianxing He, Zhilong Xia, and Yang Feng. 2022. A comprehensive empirical study on bug characteristics of deep learning frameworks. *Information and Software Technology* 151 (2022), 107004. https://doi.org/10.1016/j.infsof.2022.107004

[94] Xiao Yu, Haoxuan Chen, Feifei Niu, Xing Hu, Jacky Wai Keung, and Xin Xia. 2025. Towards Understanding Bugs in Distributed Training and Inference Frameworks for Large Language Models. arXiv:2506.10426 [cs.SE] https://arxiv.org/abs/2506.10426

[95] Andreas Zeller. 1999. Yesterday, My Program Worked. Today, It Does Not. Why?. In *Proceedings of the 7th European Software Engineering Conference Held Jointly with the 7th ACM SIGSOFT International Symposium on Foundations of Software Engineering* (Toulouse, France) *(ESEC/FSE-7)*. Springer-Verlag, Berlin, Heidelberg, 253–267. https://dl.acm.org/doi/10.1145/318774.318946

[96] Andreas Zeller. 2002. Isolating Cause-Effect Chains from Computer Programs. In *Proceedings of the 10th ACM SIGSOFT Symposium on Foundations of Software Engineering* (Charleston, South Carolina, USA) *(SIGSOFT '02/FSE-10)*. Association for Computing Machinery, New York, NY, USA, 1–10. https://doi.org/10.1145/587051.587053

[97] Ru Zhang, Wencong Xiao, Hongyu Zhang, Yu Liu, Haoxiang Lin, and Mao Yang. 2020. An empirical study on program failures of deep learning jobs. In *Proceedings of the 42nd International Conference on Software Engineering* (Seoul, South Korea) *(ICSE '20)*. Association for Computing Machinery, New York, NY, USA, 1159–1170. https://doi.org/10.1145/3377811.3380362

[98] Yongle Zhang, Serguei Makarov, Xiang Ren, David Lion, and Ding Yuan. 2017. Pensieve: Non-Intrusive Failure Reproduction for Distributed Systems using the Event Chaining Approach. In *Proceedings of the 26th Symposium on Operating Systems Principles* (Shanghai, China) *(SOSP '17)*. Association for Computing Machinery, New York, NY, USA, 19–33. https://doi.org/10.1145/3132747.3132768

[99] Yongle Zhang, Kirk Rodrigues, Yu Luo, Michael Stumm, and Ding Yuan. 2019. The Inflection Point Hypothesis: A Principled Debugging Approach for Locating the Root Cause of a Failure. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles* (Huntsville, Ontario, Canada) *(SOSP '19)*. Association for Computing Machinery, New York, NY, USA, 131–146. https://doi.org/10.1145/3341301.3359650

[100] Wenxin Zheng, Wenxiao Wang, Yun Zhang, Mingcong Han, Bin Xu, Jinyu Gu, Xingda Wei, Haibo Chen, Zuquan Song, Gaohong Liu,

Yucheng Nie, Zhe Nan, Zhuolin Zheng, Huan Yu, Shuguang Wang, Ziming Zhou, Hang Zhu, Wencong Xiao, and Xin Liu. 2026. SDCs in the Wild: Characterizing and Diagnosing SDC-defective GPUs in Production LLM Training. In *20th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2026, Seattle, WA, USA, July 13–15, 2026*. USENIX Association. https://www.usenix.org/conference/osdi26/presentation/zheng

A Implementation Details

A.1 Preflight Operator Capturing

This appendix describes the mechanisms used by OpGuard to capture both semantic operator intent and concrete device execution.

Python-Level Monkeypatching. At the semantic layer, OpGuard instruments high-level tensor operations by wrapping relevant Python callsites through dynamic monkeypatching. This unified mechanism subsumes multiple pathways, including `__torch_dispatch__` and Triton's `JITFunction.run`, ensuring that all user-visible tensor operators—including custom Triton kernels—are intercepted before reaching the runtime.

Each intercepted call records the operator name, tensor metadata, argument summaries, and the Python callstack. This provides lightweight, framework-agnostic attribution to user code while introducing negligible overhead.

CUPTI: Comprehensive Device-Level Visibility. CUPTI (CUDA Profiling Tools Interface) supplies a stream of GPU activity records, including kernel launches, memory transfers, and synchronization events. These callbacks reveal exactly which kernels execute on the device, independent of Python semantics, fusion decisions, or scheduling.

However, CUPTI callbacks run asynchronously inside CUDA driver threads, which makes Python callstack collection unreliable—the interpreter may have already advanced beyond the callsite that triggered the kernel. Moreover, CUPTI callbacks can occasionally interact poorly with internal driver threads, creating rare reentrancy or deadlock hazards.

Despite these limitations, CUPTI is the default backend in OpGuard because it provides nearly exhaustive device-level visibility with minimal runtime interference.

CUDA API Interception: Safe Fallback Path. When CUPTI is unsafe or unavailable, OpGuard falls back to intercepting CUDA runtime and driver APIs such as `cudaLaunchKernel`, `cuLaunchKernel`, and `cublasLtMatmul`. This mechanism captures every kernel launch along with its stream and device association, and is highly robust even in applications that create custom CUDA contexts or employ heavy multithreading.

The tradeoff is wrapper completeness: new kernels introduced by vendor or third-party libraries must be added to the wrapper registry to maintain coverage.

Portability. For NPUs and other emerging XPU architectures, we are implementing analogous backends that expose kernel-launch activity streams comparable to the CUDA mechanisms above. All backends conform to the same operator–execution attribution model, keeping the upper layers of OpGuard hardware-agnostic.

A.2 Instrumentation Recognition

Extraction of Instrumentation Entries. For each operator recorded during preflight tracing, OpGuard re-parses the corresponding source file using both the abstract syntax tree (AST) and the concrete syntax tree (CST). The recognizer identifies the smallest AST node representing the operator and emits an `InstrumentationEntry` containing:

- operator form (call, assignment, augmented assignment, subscript write);
- symbol and attribute names;
- structural tags (receiver, argument shapes, keyword structure);
- metadata propagated from runtime traces (kernel symbols, stream IDs).

Entries represent all potential instrumentation sites surfaced during preflight, including user-defined operators, Triton kernels, and implicit tensor writes.

Signature-Based Matching. To locate instrumentation sites robustly across minor code edits, OpGuard performs tag-based signature matching. Rather than relying on fixed line numbers or function names, the recognizer compares extracted syntax tags against a library of operator templates using weighted-overlap scores. Tags include operator category, argument structure, receiver patterns, and indicator variables associated with device-kernel launches. The highest-scoring template determines which operator is being instrumented.

This matching model tolerates refactorings such as helper-function extraction, expression rewrites, and reordering of arguments, while still identifying the semantically correct operator.

Specialized Structural Patterns. Some operator forms require dedicated handling. Here are the two main cases:

- Triton kernels: The recognizer identifies Triton-style launches of the form `kernel[grid](...)` using structural cues such as grid blocks, pointer-style arguments, Triton-specific meta-arguments, and runtime correlation with kernel symbols.
- Indexed writes: Assignments such as `x[i] = v` are treated as in-place updates (e.g., `index_put`). The recognizer rewrites these sites into small shims that route through controlled OpGuard instrumentation while preserving evaluation semantics.

A.3 Window Alignment Algorithm

Algorithm 1 gives the dynamic program used inside each anchor-bounded alignment window. Anchor construction and the rescue pass are described in Section 4.5; the algorithm below focuses on the local monotone matching step.

Algorithm 1: AlignWindowDP($F_1[i_0..i_1], F_2[j_0..j_1]$)

Input: Two sequences of boundary events $F_1[i_0..i_1]$ and $F_2[j_0..j_1]$.
Output: Matches, a monotone, non-crossing set of pairs.
Initialize DP table $D[u, v]$ for $u \in [i_0..i_1], v \in [j_0..j_1]$.
Constrain the band: only fill entries with $|(u - i_0) - (v - j_0)| \le b$.
for $u \leftarrow i_0$ ***to*** i_1 **do**
  for $v \leftarrow j_0$ ***to*** j_1 **do**
    if $|(u - i_0) - (v - j_0)| > b$ **then**
      continue
    `// Skip, match, or gap`
    $s_{\text{match}} \leftarrow D[u{-}1, v{-}1] + \text{score}(F_1[u], F_2[v])$
    $s_{\text{gap1}} \leftarrow D[u{-}1, v] + \text{gap}$
    $s_{\text{gap2}} \leftarrow D[u, v{-}1] + \text{gap}$
    $D[u, v] \leftarrow \max\{s_{\text{match}}, s_{\text{gap1}}, s_{\text{gap2}}\}$
Backtrack from (i_1, j_1) to (i_0, j_0), collecting pairs (u, v) where the transition chose score$(F_1[u], F_2[v])$.
return Matches

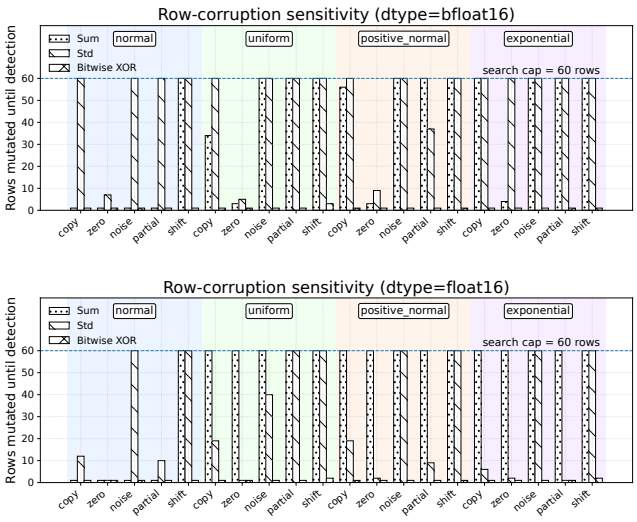


Figure 10. Sensitivity of the XOR fingerprint under row-level corruptions in `bfloat16` and `float16`.

A.4 XOR Fingerprints

To determine whether two tensors are bitwise identical, OpGuard employs a constant-size *XOR fingerprint*: a deterministic reduction over the raw byte stream of the tensor. Given a contiguous byte array $\{b_1, \ldots, b_N\}$, the fingerprint is

$$\text{XOR}(b_1, \ldots, b_N) \;=\; b_1 \oplus b_2 \oplus \cdots \oplus b_N.$$

Kernel implementation. At each instrumented Python or CUDA op, OpGuard launches a custom CUDA reduction on the same stream as the operator. The kernel processes the tensor in coalesced 4-byte lanes, reducing to a 32-bit signature in a single pass. If the total byte count is not a multiple of four, the remaining 1–3 bytes are padded with zeros to form the final word, ensuring identical semantics to the CUDA kernel. XOR is order-insensitive, precision-agnostic, and flips on *any* bit change. Runtime is on par with native `torch.sum/mean` and typically below `torch.std`, imposing negligible overhead. The CPU fallback follows the same rule by packing the byte stream into 4-byte words (zero-padding any leftover bytes) and XOR-reducing them.

Sensitivity. Across all tested tensor precisions (FP32, BF16, FP16), value distributions (normal, uniform, positive-normal, exponential), and corruption modes (full-row overwrite, partial overwrite, small-noise injection, circular shifts), fingerprints diverge after only *1–2 corrupted rows* (Figure 10). A single-bit flip always flips the fingerprint. In contrast, numeric summaries (e.g., sum, mean, variance) frequently require tens to hundreds of corrupted rows to manifest detectable changes, particularly under low-precision formats where numeric cancellation is common.

Known blind spots. XOR is intentionally a lightweight fingerprint rather than a cryptographic hash, and it inherits XOR's algebraic blind spots. In particular, it is insensitive to byte order: a pure permutation of tensor contents, such as swapping two rows, preserves the fingerprint. More generally, structured changes can cancel when the byte-level differences XOR to zero, for example two identical corruptions introduced an even number of times. We consider this acceptable for OpGuard's operating point for three reasons. First, the production failures we target are silent value corruptions, stale reads, missed writes, precision/path regressions, and communication inconsistencies, all of which typically introduce non-canceling byte changes at the first faulty boundary. Second, OpGuard records shape, dtype, device, rank, stream, and boundary identity alongside the fingerprint, so XOR is not used to validate semantic reordering or layout changes in isolation. Third, the constant-size XOR kernel is cheap enough to run continuously at many boundaries; replacing it with a stronger order-sensitive hash would reduce coverage or increase perturbation. When a bug hypothesis is specifically permutation-sensitive, OpGuard can rerun that small region with full tensor dumps or a stronger checker, but this has not been needed for the cases in our study.

Collision probability. As a linear map $\{0,1\}^{8N} \to \{0,1\}^{32}$, XOR admits collisions in principle, but only when the two tensors differ by a byte vector whose XOR reduces exactly to zero. For random corruptions, the probability is 2^{-32} (2^{-64} for a 64-bit variant), negligible relative to typical sources of training noise. No collision was observed in millions of trials.

False positives and negatives. False positives cannot occur because the fingerprint is deterministic and purely byte-based. False negatives require two distinct tensors to reduce to the same XOR value, which for random corruptions occurs with the same 2^{-32} probability noted above. A more subtle structural false negative may arise only when a tensor contains a permutation or *deliberate, perfectly duplicated byte sequences*

that appear an even number of times, allowing their contributions to cancel. We observed the latter once while diagnosing Bug 29, where the model carried two identical embedding tables for interface uniformity. Such duplication does not arise in optimized production or open-source training pipelines.

Precision-independence and practicality. Because the fingerprint operates directly on raw bytes, its sensitivity is invariant to the tensor's numeric precision and underlying distribution. The overhead consists of two small reduction kernels per operator boundary ($\approx 0.8\times$–$\approx 1.1\times$ the cost of `torch.sum`), making the method suitable for continuous use over multi-day runs.

A.5 Practical Determinism Controls

The main paper summarizes the control surfaces that make bitwise alignment well-defined. This section records the concrete knobs we use in practice. These controls are lightweight and do not alter model semantics or disable core optimizations.

Controlled randomness and initialization. We set consistent seeds for CPU and CUDA RNGs [71], dropout streams, and dataloader workers [67], and use deterministic distributed initialization so that all ranks construct identical parameters and mini-batch sequences [66].

Deterministic kernel and library behavior. Reproducible kernel selection is achieved with `torch.use_deterministic_algorithms`, deterministic cuDNN modes, disabled autotuning, and TF32 disabled (`torch.backends.cuda.matmul.allow_tf32 = False`) [70]. For cuBLAS, setting a fixed workspace configuration (e.g., `CUBLAS_WORKSPACE_CONFIG=:16:8`) prevents stream-dependent workspace choices and keeps accumulation order stable. cuBLAS reproducibility also assumes the same toolkit version, GPU architecture, and SM count across runs; multi-stream use requires either a user-provided workspace or `CUBLAS_WORKSPACE_CONFIG` [51].

Deterministic attention and MoE operations. Certain fused attention and MoE kernels vary in reduction order or token grouping. We select deterministic variants using internal control flags that enforce stable indexing and stable accumulation patterns, preventing kernels that compute the same math but use unstable internal scheduling or reduction orders from causing bitwise drift.

Deterministic collectives. Floating-point collectives must follow the same reduction tree. We therefore pin the collective algorithm, protocol, and topology so that partial sums are accumulated in an identical order across runs. [56] Residual divergence under fixed topology indicates a true correctness fault or silent data corruption (SDC) (Bug 10 and Bug 12).

Checkpoint-resume determinism. To resume bitwise-identically, all parallel ranks restore their RNG states, optimizer state, dataloader position, and scheduler counters. With complete restoration, a resumed run continues bitwise from saved prefix.

Bitwise Comparability Across Parallelism. We also achieve bitwise comparability across different parallelism degrees, including internal and open-source Megatron-style tensor parallelism (TP). Because standard TP changes arithmetic order via tensor sharding and parallel reductions, we introduce a *TP-simulator mode*:

- The logical TP size remains > 1, but each rank executes the full, unsharded computation: embeddings, attention projections, activations, and logits are replicated locally.
- Standard TP modules (vocab-parallel embedding, column-/row-parallel linear layers, dot-product attention, and parallel cross-entropy) are replaced with virtual TP layers that avoid scattering, reconstruct full inputs on each rank, and use collectives only as numerical no-ops.
- Parameter shapes, padded vocabulary sizes, and RNG streams are overridden to match the TP$= 1$ configuration.
- Optimizer-side reductions (e.g., gradient-norm all-reduce and distributed parameter synchronization) are routed through degenerate single-rank paths.

Under this mode, all ranks observe identical inputs, parameters, gradients, and updates, yielding training trajectories that are bitwise identical between TP$= 1$ and virtual TP. This enables OpGuard to compare executions across different parallelism settings without conflating arithmetic reordering with faults (Bug 23).

B Diagnosed Cases

B.1 Production Cases

Detailed diagnoses for the 20 production failures are summarized in Table 3, highlighting symptoms, root causes, and the precise operator boundary where OpGuard first detects divergence.

B.2 Open-source Cases

Detailed diagnoses for the 10 open-source issues are summarized in Table 4, showing how OpGuard isolates the earliest semantic boundary that diverges across implementations.

ID	Category	Symptom	Root cause	First diff	Reference run	Scale	Step at 1st diff
Bug 1	Kernel race	Loss jump after ~3080; clean grad-norm	Racy fused embedding-bwd kernel	Embedding-bwd grad	Self	Full	3080
Bug 2	Kernel nondet	NaNs; fwd/recompute mismatch	Nondet varlen Flash-Attn kernel	Attn fwd/recompute	Self	Full	1
Bug 3	Recompute bug	Mismatch on long sequences	Missing transpose in recompute path	Post-recompute act.	Config tweak (`recompute=False`)	Down	1
Bug 4	Stream ordering	Run-to-run expert drift	Input format conversion races default stream	Expert input	Self	Full	35
Bug 5	RoPE impl mismatch	Early attention drift	Mixed PyTorch/CUDA RoPE impls	RoPE output	Config tweak (`rope_backend=pytorch`)	Down	1
Bug 6	Dtype drift	Small reproducible loss drift	Different bf16 cast points in MTP norm	MTP norm	X-stack (>10k LOC)	Down	1
Bug 7	Optimizer impl	Loss drift across stacks	PyTorch vs. fused AdamW accumulate differently	Post-optimizer weights	X-stack (>10k LOC)	Down	1
Bug 8	State overwrite	Loss drift after EMA validation	Validation overwrites tensors reused by training	Next-step params	Config tweak (`ema_validation=False`)	Down	1
Bug 9	Kernel nondet	Inconsistent attn gradients	Deterministic flag disabled in bwd kernel	Attn bwd output	Self	Down	1
Bug 10	Collective topology	Node-type-specific loss drift	Different NCCL tree leads to different FP reduction order	Reduce-scatter	Self (same Python stack, NCCL/top differs)	Full	1
Bug 11	Recompute bug	Recompute ≠ forward (bitwise)	Wrong microbatch index + mismatched cast order	Early recompute act.	Config tweak (`recompute=False`)	Down	2
Bug 12	Hardware SDC	Rare MoE shape crash (pool A only)	Boolean scatter flips 1 bit on some XPUs	Router mask	Self	Full	4232
Bug 13	Hardware SDC	Same MoE crash on pool B	Same scatter op shows low-freq SDC	Router mask	Self	Full	4955
Bug 14	Config mismatch	Divergence from MoE layer 2	Histogram bin count inconsistent across stacks	Histogram output	X-stack (>10k LOC)	Down	1
Bug 15	Init bug	Immediate drift	Pad-token embedding not zeroed	Padded-token embedding	X-stack (>10k LOC)	Down	1
Bug 16	Kernel race	Deterministic drift at fixed step	Shared-mem read before sync in custom kernel	Embedding-bwd grad	Self	Down	220
Bug 17	Offload misuse	Small persistent drift	GPU upload races default stream	Reloaded activation	Stable (310 LOC)	Down	1
Bug 18	Kernel race	Occasional bwd NaNs / drift	Triton bwd uses shared mem before barrier	Attn bwd output	Self	Full	95
Bug 19	Layout assumption	Attention drift after minor schedule change	Fused QKV kernel assumes contiguous input	QKV projection	Stable (260 LOC)	Down	1
Bug 20	Checkpoint bug	Resume diverges instantly	Name collision zeroes optimizer step tensor	Optimizer step tensor	Config tweak (`checkpoint=False`)	Full	1

Table 3. Summary of 20 production failures diagnosed using OpGuard. The table reports the first divergent tensor, the reference run used for comparison, replay scale, and the training step where the first mismatch appeared. Reference runs are grouped as self-replay (Self), last stable commit (Stable), configuration tweak, or cross-stack comparison (X-stack); parenthesized LOC counts are approximate code differences.

ID	Category	Symptom	Root cause	Reference run	OpGuard first diff
Bug 21	Stream race	Inconsistent loss when overlap-comm is set to true	Race between default stream and reduction stream	Config tweak (`overlap_comm=False`)	Modified tensor on the reduction stream
Bug 22	Kernel bug	Non-deterministic model outputs with context length 1	Kernel implicitly use the cached states	Self-replay (same input)	Wrong kernel output
Bug 23	TP consistency	Tensor parallelism produces divergent loss	Diff padded vocab size in diff TP size	Single-GPU TP simulator	Padded vocab
Bug 24	Collective op impl	Precision issue caused by different token dispatchers in MoE training	Diff calculation impl between alltoall and allgather	Config tweak (AllGather dispatcher)	Backward output grad
Bug 25	Stream race	Deepspeed overlap-comm data race	Race between default stream and reduction stream	Config tweak (`overlap_comm=False`)	Corrupted tensors caused by race
Bug 26	Kernel bug	Non-deterministic inference with customized injected kernel	Incorrect offset when slicing the workspace	Config tweak (kernel inject off)	Diff tensor directly after the wrong kernel
Bug 27	Impl mismatch	Different vision embedding outputs across different library versions	Numerical implementation lead to diff rounding order	Stable library version (>5K LOC)	Patch coordinate grid
Bug 28	Cache bug	Qwen-2-VL generates inconsistent logits between generate and call	Cached RoPE deltas not reset when the cache is None	API path (`mode=generate`)	Get-rope-index output
Bug 29	Init bug	Accuracy regression across diff versions	Position id prepared wrong	Stable library version (>5K LOC)	Vision patch emd output
Bug 30	Init bug	Inconsistent loss between DeepSpeed and DDP on single GPU	Perform an extra step when init optimzier	Cross-stack DDP	Diff loss

Table 4. Summary of 10 open-source issues evaluated using OpGuard. The reference column reports the run or configuration used for alignment; the first eight cases produced precise first-difference operators, while the last two expose current alignment limits.

RobustRL: Role-based Fault Tolerance System for RL Post-Training

Zhenqian Chen[1], Baoquan Zhong[1], Xiang Li[2], Qing Dai[1], Xinkui Zhao[1†],
Miao Ye[1], Ren Cheng[2], Lufei Zhang[3], Jianwei Yin[1]
[1]Zhejiang University [2]Unaffiliated
[3]State Key Laboratory of Mathematical Engineering and Advanced Computing, China

Abstract

RL post-training for LLMs has been widely scaled to enhance reasoning and tool-using capabilities. However, RL post-training interleaves training and inference workloads, exposing the system to faults from both sides. Existing fault tolerance frameworks for LLMs target either training or inference, leaving the optimization potential in the asynchronous execution unexplored for RL. Our key insight is role-based fault isolation so the failure in one machine does not affect the others. We treat trainer, rollout, and other management roles in RL training as distinct distributed sub-tasks. Instead of restarting the entire task in pretrain robust system ByteRobust, we recover only the failed role and reconnect it to living ones, thereby eliminating the full-restart overhead including rollout replay and initialization delay.

We present RobustRL, the first comprehensive robust system to handle GPU machine errors for RL post-training ETTR (Effective Training Time Ratio) improvement via a *Detect-Restart-Reconnect* paradigm. (1) *Detect*. We implement role-aware monitoring to distinguish actual failures from role-specific behaviors to avoid the false positive and delayed detection. (2) *Restart*. For trainers, we implement a non-disruptive recovery where rollouts persist state and continue trajectory generation, while the trainer is rapidly restored via rollout warm standbys. For rollout, we perform isolated machine replacement without interrupting the RL task. (3) *Reconnect*. We replace static collective communication with dynamic, UCX-based (Unified Communication X) point-to-point communication, enabling immediate weight synchronization between recovered roles. In an RL training task on a 256-GPU cluster with Qwen3-8B-Math workload under 10% failure injection frequency, RobustRL can achieve an ETTR of over 80% compared with the 60% in ByteRobust and achieves 8.4%-17.4% faster in end-to-end training time.

1 Introduction

Reinforcement Learning (RL) has emerged as a transformative paradigm for post-training LLMs (Large Language Models) [2, 6, 32], where the week-level [24] and 200K GPUs large-scale [80] RL training has been deployed to enhance their reasoning and tool-using capabilities through iterative policy optimization [57,59,85]. State-of-the-art models [7,24,52,63,64,80] all leverage RL post-training to achieve superior performance in complex tasks [91], such as mathematics [50], code generation [30,42], and search [31] *et al.*.

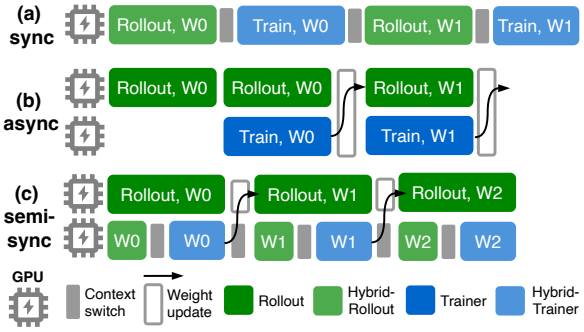


Figure 1: Different RL training architectures. The GPU role includes trainer, rollout and hybrid (colocate the both).

Unlike traditional LLM pre-training [19, 29, 73] or serving [34,62,75,81] workloads, RL post-training comprises two phases: rollout (generation and tool interaction) and training, where they contribute close to 100% and 70% post-training time in reasoning tasks and tool learning tasks [60]. To mitigate the long-tail latency inherent in the rollout phase, RL training frameworks have evolved from synchronous architectures that co-locate rollout and trainers [26, 43, 61, 84, 90] in Figure 1(a) to asynchronous designs [16, 25, 60, 77, 89, 93] in Figure 1(b) and (c). These include async mode with dedicated standalone rollouts and trainers, as well as semi-sync mode [7, 65] that combine co-located workers with additional standalone rollouts.

The efficiency of large-scale RL training is therefore critically dependent on the reliability of both the trainer and rollout because they consume the majority of training time [60]. When scaling to thousands of GPUs, machine failures become the dominant source of task interruption [11, 38, 73].

†Corresponding author.

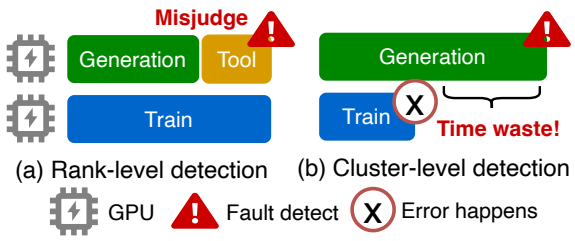


Figure 2: Fault detection of pre-train applied in RL. (a) Rank-level leads to false positive. (b) Cluster-level leads to delay.

However, because rollout and training are tightly interleaved in a single RL task, fault detection and recovery must handle failures stemming from both phases. A naive solution is applying different fault tolerance strategies to the specific roles of the failed components. Unfortunately, they lack the fine-grained mechanisms required for the unique demands of RL. Existing fault tolerance solutions are tailored to pure pre-training [4, 13, 17, 27, 36, 70, 71, 73] or pure inference workloads [39, 46, 54]. In current RL systems, the failure of a single critical worker would trigger a RL task restart, discarding training progress and wasting computation resources. Ideally, a fault in a machine hosting one distributed RL role should not disrupt the others. Only the ***detected*** failed role should ***restart*** and subsequently ***reconnect*** with others while the remaining roles continue to run. However, achieving this ideal state presents three significant challenges.

Challenge #1: Accurate and efficient fault detection. The rank-level detection for training in ByteRobust does not work for RL. It flags a process as suspect when its behavior deviates from peers or it exhibits zero GPU or network activity [11, 29, 38, 73] because each process runs identical forward-backward passes. As shown in Figure 2(a), however, rollouts can idle their GPUs while awaiting external tool responses, causing false positives. Conversely, cluster-level detects the error when all ranks have no GPU activity. It masks these idle periods and introduces significant detection delays as shown in Figure 2. For instance, a trainer network failure might remain undetected for hours until the long-tail rollout phase completes, wasting valuable cluster resources.

Challenge #2: Fine-grained and efficient restart. We need to guarantee both machine failures in trainer and rollout would not lose the trajectory due to the task restart. Furthermore, for the trainer, we need rollout to continue the generation during the recovery of the trainer. Besides, the gang-scheduling leads to the long blocking time until the new machine initializes, while the extra warm standby machine leads to the resource waste. An efficient and resource-friendly system is needed for efficient trainer restart. For rollout, we want the recovered rollout to soon get the latest weight for trajectories generation.

Challenge #3: Dynamic and efficient weight synchronization. The restart trainer or rollout needs to build reconnection for subsequent weight update as shown in Figure 1(b) and (c). Collective communication based on NCCL [1] needs to build the fixed communication group before transfer, so it cannot connect with the recovered role in the new machine and does not support fault tolerance. Point-to-point communication like UCX is available for dynamic reconnection, but needs asynchronous and parallel optimization to be efficient.

To address these challenges, we designed RobustRL, the first comprehensive role-based fault tolerance system for RL post-training in the three RL modes of Figure 1. It mitigates the impact of the machine failures for the rollout and trainer. Our contributions are as follows.

1. Role-phase-aware fault detection. It decides which roles should restart. We introduce a phase-aware and extensible detection strategy for trainer and rollout. Faults in corresponding roles are handled using appropriate robust strategies. (§4).

2. Role-based and resource-efficient restart. It decides how the roles restart. We propose a decoupled recovery strategy to avoid the RL task restart and preserve the rollout progress. In semi-sync and async mode, for trainer failures, we implement an efficient recovery that dynamically takes rollout as warm standbys, bypassing gang-scheduling delays without requiring extra idle resources. For rollout failures, the recovered rollout would pull the latest weight from other rollouts for trajectory generation immediately. (§5.1)

3. Efficient and reliable reconnection. It decides how the roles reconnect. We implement UCX-based (Unified Communication X) dynamic communication to replace NCCL for weight synchronization. It can achieve efficient weight update through asynchronous point-to-point communication and relay server design. In addition, it can handle the different failure cases during the recovery. (§5.2)

4. End-to-end benefits. In the scenario with frequent failures where a trainer fault is injected every 10% steps, RobustRL still maintains an 80% ETTR on the Qwen3-8B-Math training task with 20% higher than ByteRobust. The RL training time is 8.4%-17.4% faster than ByteRobust. (§7)

2 Background & Motivation

2.1 RL System for LLM Post-Training

Post-training using Reinforcement Learning (RL), which evolved from initial human feedback (RLHF) techniques [5, 55], is now critical for enhancing the reasoning [24, 59, 85] and tool-using capabilities [14, 30] of LLMs. This process consists of two phases, rollout and training.

Rollout. The rollout model generates trajectories based on prompts and the results of environment interaction, a process typically completed by an inference engine [35, 88]. For example, mathematical reasoning problems [50] can be solved using chain-of-thought (CoT) [76]. The rollout model can further achieve tool learning through multi-turn tool invocations. In software engineering [30] scenarios, trajectories include multi-turn interactions with a code sandbox to fix a bug or implement a specific function. In search scenarios [31, 33, 83],

trajectories involve multi-turn interactions with a search engine to retrieve target answers.

Training. The actor updates its policy by trajectory rewards evaluated in the training engine [49, 87]. The score can be obtained via various reward sources such as rule-based functions [24, 85] or learned reward models [5, 53]. Subsequently, the reference and critic models convert per-trajectory scores and advantages into training experience. Finally, the actor model consumes this experience through a specific policy update strategy [3, 57, 59, 85] to complete this iteration.

RL training system. RL frameworks like verl [22] adopt a synchronous mode. After the rollout phase is completed, they switch the inference engine to a training engine by resharding the model weights. However, the multi-turn interactions and decode latency in rollout leads to a significant long-tail latency. Therefore, existing async RL frameworks [16, 25, 60, 89, 93] allow the trainer to train within a certain range of offline steps without waiting for all rollout trajectories to complete. It greatly improves training efficiency while maintaining an acceptable loss in precision. Compared to the hybrid deployment of the sync mode, the async paradigm independently deploys the trainer and standalone rollouts, with weight synchronization through network pulling. The semi-sync [58, 65] mode replaces the async trainer with a hybrid one, thereby avoiding idle waiting caused by an improper ratio of trainer to standalone rollouts as shown in Figure 1.

2.2 LLM Fault Tolerance System

Like pre-training, the RL post-training of large models can last for several weeks or even months [80]. During this period, machine failures inevitably occur, leading to a loss of training progress. In RL, a GPU machine where a failure occurs could be executing either a training or an inference task. Prior works, ByteRobust [73] and EaaS [39], have respectively discussed machine failure scenarios during training and inference. In the pre-training process, at the scale of a 100K GPU cluster, machine-induced failures (e.g., GPU, network, CPU, memory, disk) cause over 3,000 interruptions per week on average (30/k-machine/week). For inference services, at the scale of a 4K GPU cluster, 8.8 failures also occur weekly (2.2/k-machines/week), with the main faults manifesting as GPU disconnection and ECC (Error-Correcting Code) errors.

On top of training and inference workload, RL introduces additional stages such as weight synchronization, Ray [48] management, and tool calls. It further increases the possibility of machine failures. Furthermore, by analyzing issues from open-source RL frameworks with keyword matching, we have found the similar problems caused by the machine error as shown in Table 1. Unfortunately, there is no specific robust system for the RL system as we know and the ByteRobust is applied since RL post-training is a kind of training task. Restarting the task when an error happens in ByteRobust is not the best optimization in RL.

Project	verl [61]	Roll [21]	OpenRLHF [20]	Slime [93]
Total issues	1830	127	719	162
CUDA error	189	7	74	14
Timeout	152	6	60	9
OOM	117	7	66	12
Job Hang	65	2	20	4

Table 1: Issues of machine error and job hang in open source RL training framework.

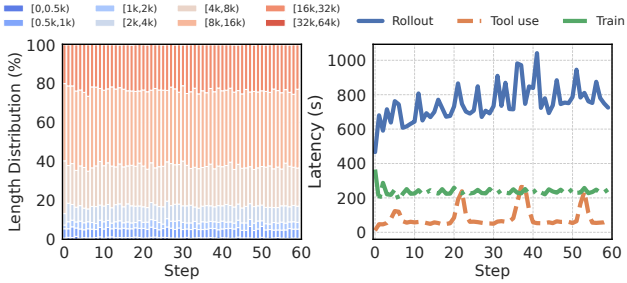


Figure 3: Trajectory length distribution (a) and time cost of each step (b) of Search R1 [31] training in HotPotQA dataset [83] with Qwen2.5-7B [67].

For instance, the pre-train robust system ByteRobust [73] achieves warm standby machines to enable fast recovery from a task restart. But it restarts the whole task whatever the error happens, leading to the large progress loss especially in RL training. A better solution is fine-gained restart by role and rollout can be the warm standby for the trainer. We can apply ByteRobust only when the role-based robust strategy in RL cannot handle the case like human-made code or configuration error. For the inference system, they mainly focus on keeping the service running [39,46] or avoid re-prefilling [18,54] when one inference machine fails. Current RL robust systems follow the inference design [60, 79] to achieve the robust rollout because it only needs to extra consider how to reconnect with the trainer in async mode. However, the trainer process has much higher failure frequency compared with inference. The robust trainer is important and the robust rollout should take the trainer failure into consideration.

2.3 Opportunity and Challenge

RL robust benefits. Role-based fault tolerance in RL has two benefits. First, it ensures the failure isolation of different roles. The task can continue to run, which guarantees that the current rollout and agent environment states are maintained. This is important under the long-tail phenomenon, where the rollout phase constitutes the main overhead. In Figure 3 of search agent training, rollout phase plays the major overhead in one step. Because the response length is long and the tool calling time contributes to the extra overhead. Second, it reduces the recovery time. We only need to restart the trainer instead of the RL task. In the case of async training, the failure of a rollout

does not affect the training task, as requests are forwarded to other rollouts. A trainer failure does not affect the rollouts from generating trajectories. To achieve so, we need to handle the following issues.

Per-step checkpoint. If weights are not saved each step, a weight inconsistency between the recovered trainer and the rollouts will occur when a failure happens. This leads to trajectory deviations and affects the training results. In this situation, it is necessary to roll back to the last checkpoint saved step to retrain and re-rollout. Per-step checkpoint can avoid the re-execution of the time consuming rollout phase.

We believe that a per-step checkpoint is available in the RL scenario based on the following insights. First, in pre-training scenarios, checkpoints are typically saved every ten or hundred steps because a single training step takes only seconds. In contrast, an RL step can last for minutes or even hours [91], which makes the checkpoint overhead ratio small. Second, in async RL training, the checkpoint process only blocks the trainer for a short time, while rollouts can continue inference during this period. Third, to further reduce the overhead brought by a per-step checkpoint, we apply the ByteCheckpoint [72]. The blocking time for saving weights is mainly the GPU to memory time within 5 seconds.

Efficient point-to-point communication. In asynchronous RL training, when trainer or rollout has faults, it needs to re-establish connections after recovery for later weight synchronization shown in Figure 1(b) and (c). The existing collective communication protocol NCCL only supports the static communication member so it cannot support robust cases. For example, the trainer would send its weight by layer to the rank-0 of the rollout machine by NCCL. Then the rollout machine boards the weight to other ranks through NVLink. When the number of communication members increases, the transfer overhead would increase linearly.

We solve this by the UCX-based point-to-point communication. It supports dynamic connection and asynchronous rank-level transmission from the trainer GPU to rollout GPU directly. It solves the traffic bottleneck of single NICs (Network Interface Cards) in NCCL. In addition, the rollout which finishes the weight update can be the relay server for the outdated or recovered rollout pulling. Finally, We further support the fault tolerance case during weight synchronization.

3 System Overview

The design goal of RobustRL is to ensure reliable RL training by using a role-based fault detection mechanism and applying different fault recovery strategies to trainer and rollout. RobustRL consists of two core components: control plane and data plane. The architecture overview is shown in Figure 4.

Control Plane. It mainly consists of the phase aware analyzer and the runtime controller. The analyzer is responsible for interacting with the RL robust runtime. It tailors fault-detection strategies to each RL role and training stage. The controller,

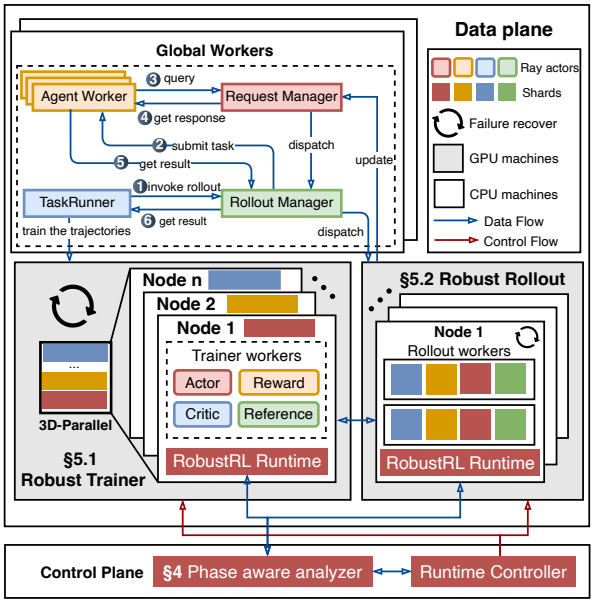


Figure 4: System overview of RobustRL.

responsible for machine scheduling, receives analysis results from the analyzer and adopts the appropriate fault tolerance and recovery strategy (§4).

Data Plane. The GPU machines are the primary failure sources and they all have fault tolerance capabilities (§5) for machine errors with RobustRL. All trainers would restart whenever any trainer machine fails, while rollouts would only restart or replace the corresponding faulty machine.

The CPU machine includes management roles, such as the `AgentWorker` for tool interaction, the `RolloutManager`, and the `RequestManager` for managing inference instances and requests. They are placed on CPU machines, and the number of machines is determined by the workload. There can be multiple `AgentWorkers`, while the other roles default to one instance each. The `TaskRunner` invokes the `RolloutManager` to get the trajectories (①). Then `RolloutManager` invokes the `AgentWorker` (②) to call the inference engine and tools to store the result in the `RequestManager` (③). This process can be a multi-turn conversation and the `AgentWorker` get the trajectories (④ and ⑤) to the trainer (⑥).

A failure in these roles triggers a RL task restart [73]. We use affinity scheduling to ensure that management roles are not scheduled on trainer or rollout machines. This eliminates RL task restarts that would otherwise be triggered by the termination of a management role when a trainer or rollout machine is replaced after a failure.

4 Role-based Fault Detection

We illustrate the semi-sync RL workflow in Figure 5 since both sync and async training are special cases of it. Our goal

is to detect which trainer or rollout machine has failed fast and apply the corresponding robust mechanism in §5.

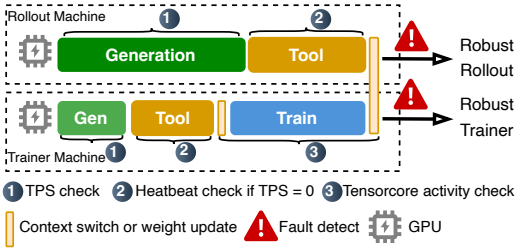


Figure 5: Role-phase-aware fault detection in semi-sync RL training. Row 2 is the hybrid. TPS: Throughput Per Second.

Trainer fault detection. As introduced in §2.1, the training phase is divided into the forward-only computation of the reference/critic/reward models and the forward-backward computation of the actor/critic policy update. The detection strategy during this process is consistent with that of pre-training. Failures manifest as zero GPU TensorCore activity for a five-minute window in RobustRL. When this occurs, we switch to the robust trainer workflow (§5.1.2).

Trainer detection is only applied during the training phase on the trainer machines (Figure 5 ③). Although phases such as context switching, weight updates, or advantage computation exhibit no GPU activity, their duration is short enough relative to the training phase. Thus, monitoring TensorCore activity is sufficient to detect failures without false positives from short GPU-idle phases. Users can also extend their own detection strategy for this case. For example in large scale RL training, the advantage computation can be over 5 minutes and we can set a large GPU idle threshold. The trainer fault detection method is orthogonal to both pre-training [10,11] and machine diagnostic methods [73].

Rollout fault detection. Rollout workloads comprise inference, weight synchronization, and awaiting requests or tool returns. TensorCore activity occurs solely during inference. In the other two stages, the worker exhibits no GPU activity yet remains healthy. We first periodically collect the throughput of each rollout from the `RolloutManager` (①). If the rollout exhibits zero throughput for a specified time interval (60 seconds in our system), we mark it as suspect and trigger heartbeat detection to further verify its status (②). No response within the timeout confirms a rollout machine failure and we will handle it with a robust rollout strategy (§5.2). The Rollout detection adopts throughput detection prior to heartbeat, avoiding misjudgment caused by unprocessed heartbeat requests and excessively long TTFT (time-to-first-token) in high-load Rollout scenarios. For the hybrid-rollout of sync and semi-sync mode, this detection method can also be applied. Since the error happens at the trainer machine, we switch to the robust trainer workflow (§5.1.2).

Extensibility to complex faults. It is worth noting that the role separation architecture proposed in RobustRL naturally facilitates the integration of advanced fault detection methods. By decoupling the execution contexts, our system becomes inherently compatible with specialized detection strategies designed for complex issues like silent data corruption (SDC) [28,40,41] and stragglers [38,78] to further harden the system.

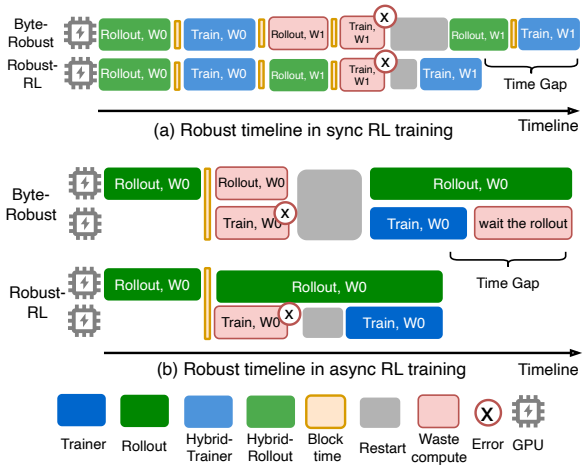


Figure 6: Timeline comparison between ByteRobust and RobustRL in (a) sync and (b) async mode when the trainer machine error happens.

5 Role-based Fault Tolerance

5.1 Robust Trainer

This section illustrates the benefit of robust trainers in different RL architectures (§5.1.1), appropriate restart strategy (§5.1.2) and reduction of restart overhead by rollout warm standbys (§5.1.3).

5.1.1 Robust Trainer Timeline

Figure 6 illustrates the difference between the ByteRobust and the RobustRL in both sync and async training mode. The semi-sync mode is the special case of sync and async mode because it contains both hybrid and standalone rollout. The yellow part denotes the time that blocks the RL training, which includes weight synchronization, per-step checkpoint and weight reshard between training and inference engine.

Robust trainer benefit. In sync training mode of Figure 6(a), ByteRobust restarts the entire task, with the rollout progress of the current iteration lost. RobustRL only restarts the affected hybrid role and preserves the rollout progress. After recovery, it simply resumes rollout and thus completes the iteration sooner than ByteRobust. In async training mode of Figure 6(b), the extra benefit is that rollout can continue running during the trainer recovery. Under the same failure case, the rollout in RobustRL ends earlier, it avoids the trainer to wait for the rollout to end in ByteRobust. In addition, compared

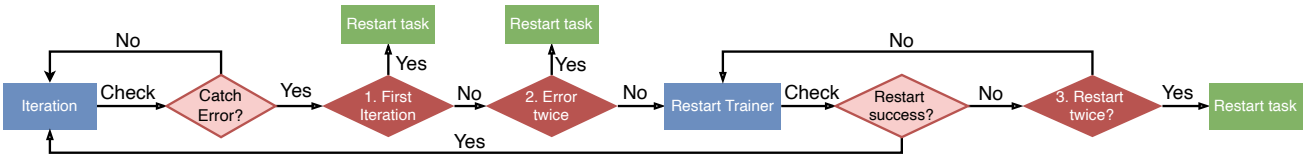


Figure 7: Robust trainer workflow. The blue part is the operation in RL training. The red part is the judge for the robust case. The green part is the policy to handle the robust case.

with restarting the RL task, we eliminate the overhead for container initialization, ray cluster initialization *et al.*.

Low blocking overhead. Due to the long-tail phenomenon of rollout, one step in RL is long shown in Figure 3(b), making the blocking time relatively short. The efficiency of rollout weight update has been solved in §5.2.1. In addition, we introduce ByteCheckpoint [72] to improve the checkpoint saving efficiency. The blocking time is only the GPU-to-memory, while the transfer from memory to disk or other persistent storage is executed asynchronously.

5.1.2 Robust Trainer Workflow

Robust trainer targets restarts from machine failures and we must avoid an infinite loop of trainer restarts by the human-induced fault. When a human-induced fault occurs, we restart the task and check the code or roll back the code version instead of trainer role restart as soon as possible. This operation "Restart task" is similar to ByteRobust's machine diagnostics and code rollback procedures. We present the robust trainer workflow in Figure 7.

Iteration. One iteration includes the step ① to ⑥ in Figure 4. The RL training is completed through the iteration of multiple steps. Robust trainer wraps the training step function with a `try-catch` block to capture exceptions during training. The sources of exceptions include both explicit and implicit failures, which are detected by the method in §4. When the exception happens it turns to the robust phase.

Trainer restart. First, `TaskRunner` terminates all trainer-related processes. Then, it re-executes the trainer's initialization phase and finally reloads the weights from the last saved model checkpoint. Since the model weights are saved each step, the reloaded weights are consistent with the rollout. Information that binds rollouts to the trainer, such as communication addresses for pulling weights, is updated concurrently with the trainer loading the weights.

When we restart to iterate, we skip to rollout a new batch. Instead, when the current batch finishes rollout, `TaskRunner` fetches the trajectories from the `RequestManager` to the trainer. Otherwise, if we use the sync or semi-sync mode for RL training, we switch the context to the inference engine to rollout the trajectories.

In the following situations, we restart the RL task instead of only the trainers shown in Figure 7.

1. First-iteration exception detected. If the model fails on the first step of training or on the first step after resuming training, this indicates an explicit machine or code error. [2]

2. Repeated exception detected. If it is the second time the fault has been detected in the current step, we indicate a reproducible fault exists in this step. This situation is also similar to a code error and we restart the task. For example, developers introduce a code error in the validation process, which is executed each 5 steps.

3. Repeated restart failure detected. The restart process itself may fail. For example, if resources are not fully released during the trainer restart or if connection establishment fails due to the restart. In such cases, one restart failure is permitted. If restart failure persists, we also consider it to be a reproducible machine or code error.

5.1.3 Trainer Warmup by Rollout

The trainer schedule follows the gang scheduling, meaning the trainer's initialization cannot begin until a new machine is scheduled and the environment is initialized, which is time-consuming as shown in Figure 8(a). ByteRobust prepares the extra warm standby machines to replace faulty ones, thereby saving scheduling and container initialization time. However, using redundant machines leads to potential resource waste, making this strategy limited to large-scale training.

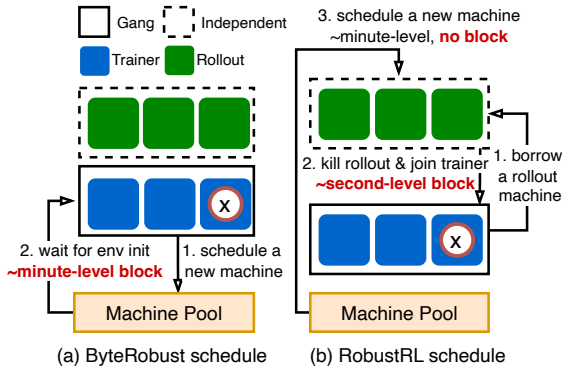


Figure 8: Schedule without extra warmup machines when trainer fails. (a) ByteRobust scheduled a new one from the machine pool. (b) RobustRL borrows a machine from rollout and rollout schedules a new one from the pool.

This design does not work for sync-mode training. In semi-

[2]Code errors in rollout trigger this process as well.

sync and async mode, we can use rollouts as warm standbys for the trainer as shown in Figure 8(b). When a failure occurs, one rollout is killed, and it will join the trainer's initialization. Since rollouts use independent scheduling, requests being processed by the rollout on the replaced machine are handled through the robust rollout strategy (§5.2).

The prerequisite for warmup rollout is that rollouts and trainers are in the same data center and rollout and trainer machines are homogeneous. In deployment, similar to ByteRobust, one can reserve a quantity of max(DP, TP, PP, EP, CP) rollouts as homogeneous machines in the same data center, enabling hot-swapping when a failure or suspected fault occurs in any parallel group. Since warmup rollouts have the same environment as the trainer, they can be quickly substituted when a trainer machine fails.

Our scheme is compatible with existing Trainer-Rollout heterogeneous deployment [89]. For example, we can add an H800 warm Rollout in cluster A for heterogeneous training with cluster A (10×H800 Trainer) and cluster B (10×H20 Rollout)). In addition, Rollout warmup requires no weight resharding because one Rollout machine is corresponding to one DP Group, which means recrimination only reduces inference concurrency. Trainer parallelism remains unchanged after borrowing a machine.

5.2 Robust Rollout

The design goals for robust rollout are to ensure that after a rollout fails, the recovered rollout can **1)** re-establish a connection with the trainer. **2)** Achieve high pulling efficiency for weight from the trainer or other relay rollouts. **3)** Update weights and resume inference as quickly as possible after recovery. To achieve these goals, we have introduced UCX-based weight synchronization (§5.2.1) and discuss the situation of the failure handling during rollout recovery (§5.2.2).

5.2.1 UCX-based Weight Update

The naive-UCX baseline transfers weights from GPU memory on machine 1 to host memory via PCIe, then across machines through a single network interface to machine 2's host memory, and finally from machine 2's memory to individual ranks via PCIe. The path is: PCIe -> NIC -> PCIe.The NCCL baseline uses RDMA GPU zero-copy (IB/RoCE v2) across nodes and intra-node NVLink for rank-to-rank communication. The path is: NIC -> NVLink. Compared to naive UCX, NCCL has a shorter data path and benefits from higher NVLink bandwidth than PCIe. Therefore, we need to shorten the communication path and improve the concurrency of UCX so as to combine the advantages both on the efficiency and elasticity. Previous work like Owl [15] discusses the data relay transfer mechanism to achieve the data-ready object sending to another object. But in RL weight synchronization, we need to further support GPU-to-GPU zero-copy transmission

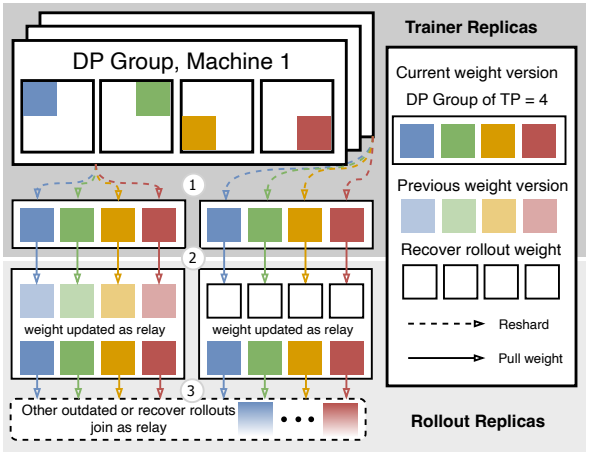


Figure 9: UCX-based weight update.

and weight shard management, and handle fault scenarios throughout the RL training lifecycle. The weight synchronization includes three parts.

① **Reshard.** As shown in Figure 9 step ①, we reshard the model to align with the sharding of the rollout. The implementation is based on verl [22], and this process takes under ten seconds.

② **UCX weight synchronization.** To ensure efficient weight synchronization, we implement point-to-point communication from the trainer's multiple data-parallel (DP) groups to the rollout replicas based on their corresponding ranks shown in Figure 9 ②. Each transfer initiates by placing the model layer by layer into a buffer for transmission. We ensure the buffer size is large enough to saturate the RDMA bandwidth. Besides, we also prevent OOM error on the GPU by preserving a buffer for transfer. The object pulling weights include both outdated and recovered rollouts awaiting weight updates. We ensure that at any given time, each weight is pulled by only one rollout to prevent network bandwidth contention.

Compared with current point-to-point communication design [60], the entire weight synchronization process is executed on the GPU. However, since torch tensors do not directly support UCX transfer, we convert them to cupy arrays [51], as cupy supports byte-level transfers. We use DLPack [12] to support a unified tensor memory layout, achieving zero-copy overhead for the conversion from torch tensors to cupy arrays.

③ **Relay transfer.** A weight updated rollout will act as a relay server, allowing other outdated and recovered rollouts to pull weights from it. Once weight pulling is complete, they join the relay-server set. The pulling process in this stage is the same as the one described in Figure 9 step ③. If a rollout recovers outside of a weight update stage, it directly pulls weights from a relay server, as shown in step ③.

The transfer efficiency between trainers and rollouts from different DP groups can vary due to network fluctuations. We make the entire weight synchronization asynchronous. It

allows the outdated and recovered rollouts to immediately pull from any rollout having the latest weights. A more extreme asynchronous strategy would be to allow asynchronous transfers at the tensor parallelism (TP) granularity, taking into account that stragglers can exist among different NICs [60]. However, the async DP transfer for a 235B model on 4×200Gbps NICs is under 10s. This overhead is sufficiently small compared to a single RL step in minutes or even hours level. Therefore, to maintain the simplicity of the weight synchronization design, we did not pursue further optimization.

In semi-sync RL training, the rollout phase does not begin until all rollouts have finished updating. In this case, the hybrid-rollout in the trainer machine also acts as a relay. In async RL training, the trainer resumes its shard once the first batch of relay rollouts finish their weight update, minimizing training stalls.

5.2.2 Rollout Failure Cases

Preserve the trajectories. We save the trajectories of the prompt of each tool iteration in the `RequestManager` to avoid the rollout progress lost when the rollout machine fails. After each tool iteration, the `RolloutManager` checks the liveness status of the corresponding rollouts. When a failure occurs on a rollout, the previous results are reassigned to other living rollouts. If it is a rollout machine failure, such as an ECC error, the machine is replaced with a new one. Since the `AgentWorker` and the sandbox reside outside the rollout machine, the rollout progress can be preserved.

Relay server failure during pulling. When resuming the weight pull, the rollout will first update its set of target addresses to the living rollouts. If a weight pull fails, it will find another relay server to pull from. The outdated or recovered rollout records their successful pulling progress and pulls the remaining one from other living relay servers.

Trainer failure during pulling. Failure may happen when the trainer prepares to be pulled or being pulled by rollout. In the first case, rollouts close their established connections with the trainer and wait for its recovery before re-establishing connections. In the case where a pull is in progress, the process is interrupted, and the partially updated weights are cleared. Finally, rollouts wait for the trainer to recover before re-initiating the weight pull.

6 Implementation

The implementation of the RL fault tolerance system at the framework level is 8k LoCs of Python for the semi-sync support, fault detection, and robust roles. RobustRL is built on verl [22] and its asynchronous RL extension [44]. We extend this foundation with a semi-sync training mode, which retains the trainer's resharding operation to the inference engine. The implementation of the training scheduler that supports the training architectures in Figure 1.

```
1.  class ElasticRayWorkerGroup
2.      # init function
3.      def __init__(self, cls, args):
4.      # support scale up/down for worker group
5.      def create_worker(...): # and destroy worker
6.      # create RayClass with init_args
7.      def scale_up(num_worker): # and scale down
8.      # invoke the create_worker
9.      def liveness_probe(...):
10.     # detect the liveness of worker groups
11. class ElasticPolicy:
12.     # define when to scale up/down
13.     def should_scale_up(...): # and scale down
14.     # define the scale up condition (recover phase)
15.     def _scaling_loop(...):
16.     # scale up if reinit, scale down if error
17.
18. rollout_cls = RayClassWithInitArgs(RolloutClass, args)
19. - rollout = RayWorkerGroup(rollout_cls, ...)
19. + elastic_rollout = ElasticRayWorkerGroup(rollout_cls, ...)
20. + rollout_policy = ElasticPolicy(elastic_rollout, ...)
```

Figure 10: The replicated ray worker group and scaling policy for the robust and elastic ray worker in RobustRL.

Robust API. In verl, the initialization of workers for the trainer or rollout roles uses the `RayWorkerGroup` (RWG) in line 19 of Figure 10. We further encapsulate this abstraction to handle fault tolerance and recovery scenarios (line 19-20). The `ElasticRayWorkerGroup` (ERWG) supports expansion and destruction, while a `ElasticPolicy` determines the timing for these actions. The ERWG supports monitoring the liveness status of the RWG and defines the initialization and destruction functions, `create_worker` and `destroy_worker`, along with their corresponding pre- and post-processing hooks. The `scale_up` is used to determine the number of workers to expand. For example, multiple rollouts fail or are taken to warm up the trainer (§5.1.3) or scales [79]. The policy defines the conditions for elastic scaling with a polling thread to check the liveness of the RWG. For instance, when the platform detects a machine failure during training, it is captured by this policy and scales up the failure worker. This abstraction avoids the need to repeatedly implement creation and destruction logic for different RWGs.

7 Evaluation

Our evaluation tries to answer the following questions.

Q1. What is the end-to-end training time and ETTR of RobustRL in scenarios with training failures? (§7.2)

Q2. What are the enhancements in RobustRL's fault recovery in terms of restart efficiency and state preservation? (§7.3)

Q3. What is the extra overhead introduced by RobustRL to implement role-specific fault tolerance? (§7.4)

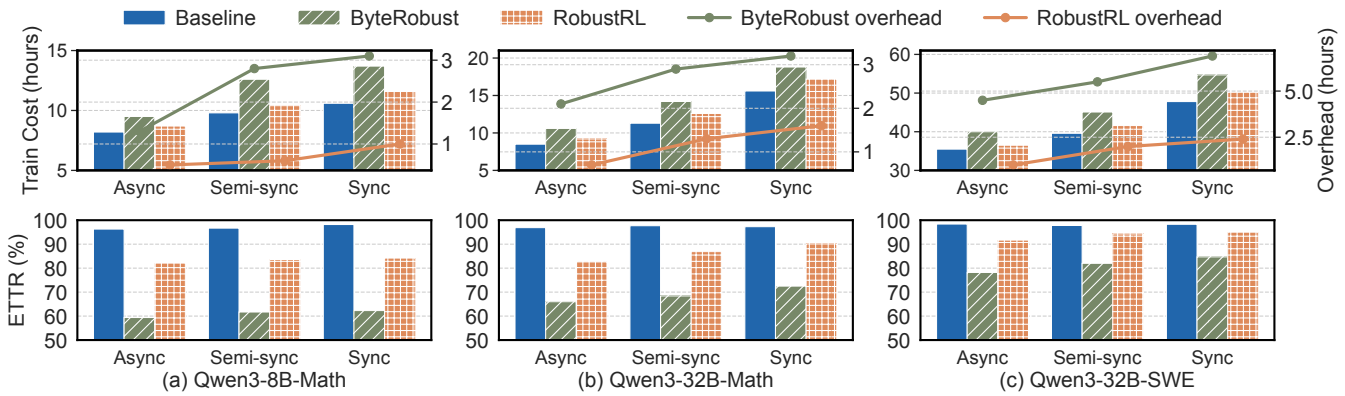


Figure 11: End to end evaluation for RL post-training of Qwen3 models in math [85] and SWE-bench [30]. The first row is the job completion time and the second row is the ETTR. The overhead of the first row is the time gap with Baseline.

7.1 Setup

Testbed. We deploy RobustRL on 32 GPU machines with 256 GPUs. Each machine is equipped with eight NVIDIA H20 96GB GPUs. The intra-machine and inter-machine bandwidths are configured with 900 GB/s NVLink and 4×200Gbps NICs, respectively. Each machine has 2TB of memory. Our software versions are CUDA 12.4, PyTorch 2.4.1, and NCCL 2.21.5. We build our system on verl 0.5.x [22], using Pytorch FSDP2 [87] and vLLM [35] as the training and inference backend. Our checkpointing uses ByteCheckpoint [72] with HDFS as the storage backend. We selected Qwen3-8B, 32B, and 235B-A22B [82] to evaluate the performance because they are the popular models for RL post-training research in academia and industry. We use the 8B and 32B models to test training performance and provide results from the 235B model to test weight saving and weight synchronization costs. The TP size of rollout is 8 to guarantee each rollout occupies a machine.

Baselines. The methods we compare are ByteRobust and our RobustRL. The baseline system is RL training without any injected faults. In ByteRobust and RobustRL, we inject a fault for the trainer at a random time within every 10% interval steps. Since the trainer follows the gang-scheduling, it would trigger the `TaskRunner` to automatically restart all the trainers. This failure frequency setting is higher than in practical production environments if the machine error rate is *i.i.d.* For example, the 100K GPU training in Meta [56] the error frequency is 18min, corresponding to 117 hours in a 256 GPU cluster. Our experiments are designed as stress tests to quantify the benefits of our fault-tolerant design under the extreme, large-scale training scenarios that will emerge in future workloads.

For ByteRobust, we retain only the in-place restart part of a failure, without machine rescheduling. The RL training includes the three architectures in Figure 1. Semi-sync switches from rollout mode to train mode when 50% of the prompts in a batch are completed and the sync mode is 100%. Async is 0% with a one-step warmup. For semi-sync and async mode, the number of GPUs for the trainer and rollouts is equal. We do not compare with the current robust rollout RL system [37, 60] because existing RL systems do not support trainer fault tolerance. Under trainer failures, their behavior is equivalent to ByteRobust. For rollout fault tolerance, the failure or recovery of rollout would not lead to the training bottleneck with further evaluation. For the weight synchronization efficiency, we directly compare with the time of ideal network bandwidth.

Dataset. For reasoning tasks on math with "DAPO-Math-17K" [85], we use a rule-based reward function to score the trajectories. For the multi-turn tool-learning task, we generate the trajectories by obtaining answers through multi-turn interactions with a sandbox in SWE-bench [30]. We use GRPO [59] for both tasks. RbustRL is independent of the specific RL algorithm and the effectiveness of RobustRL can generalize to the others. Our global batch size is set to 512, with 64 prompts per batch, and 8 responses generated for each prompt following the previous research [60]. We set the maximum text length to 65536 and the maximum number of tool interaction turns to 50. We train for 100 steps by default.

Metrics. Our primary evaluation metrics for fault tolerance are the end-to-end training time and the ETTR. We further evaluate the restart time overhead, additional fault tolerance overhead, and the time saved due to fault tolerance for different methods.

7.2 End to End Evaluation

We compare the end-to-end training time and ETTR for different training architectures under various GPU workloads on math and SWE post-training tasks on 128 GPUs.

End-to-End Training Time. As shown in the first row of Figure 11, the Qwen3-8B-Math takes the shortest time because the model is small and the post-training task is relatively simple. The SWE task takes the longest time because the introduction of tool interaction significantly lengthens the rollout time. Taking the Qwen3-8B-Math task as an example, the training time varies for different training architectures,

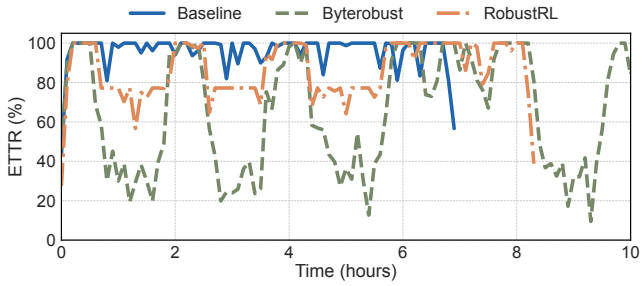


Figure 12: Sliding ETTR of the training process Qwen3-8B-Math. The sampling interval is 5 minutes.

where the async mode is faster than the sync one by mitigating long-tail overhead. Therefore, the end-to-end training time continuously increases from the async to the sync mode.

Since we inject a fault every 10 training steps, the end-to-end training times for ByteRobust and RobustRL are longer than the baseline. The overhead introduced by a fault restart is mainly caused by the restart initialization and the loss of training progress, especially the loss of rollout progress (§5.1.1). In contrast, RobustRL can restart only the trainers, allowing the rollouts to continue executing inference and environment interaction processes. Therefore, the overhead of RobustRL is smaller than that of ByteRobust. As we shift from async to sync training mode, the impact of the long-tail phenomenon becomes progressively more severe, thus the overhead of ByteRobust gradually increases. In sync mode, the rollout and training phases are executed sequentially as shown in Figure 1. By leveraging the role-based fault tolerance strategy, RobustRL explicitly preserves the rollout results. Consequently, if a failure occurs during the training phase, the system can resume directly from the training step, skipping the redundant rollout phase as shown in Figure 6(a). This stands in contrast to approaches like ByteRobust, which necessitate restarting the entire RL step. Lastly, the fault tolerance benefits of using RobustRL on complex tasks like SWE are greater than on math reasoning, because the rollout phase is more time-consuming. The benefit of preserving the rollout progress is more obvious.

In summary, under the extreme condition of a 10% failure rate, RobustRL can save 0.8-2.1h, 1.4-1.6h, and 3.5-4.5h in end-to-end training time compared to ByteRobust on the 8B-Math, 32B-Math and 32B-SWE tasks respectively. The restart overhead of RobustRL for fault tolerance accounts for less than 5% of the total time, while the ByteRobust introduces 20% overhead.

ETTR. As shown in the second row of Figure 11, computation stages such as actor policy updates and rollout are all counted towards the effective training time in RL post-training. In RobustRL, there is a phase where the trainer restarts for fault tolerance while the rollouts continue running. During this process, the recovery ETTR is calculated with the ratio $ETTR_{ratio} = \frac{\#Rollout}{\#Rollout+\#Trainer}$. Since RobustRL significantly reduces restart overhead compared to ByteRobust, its ETTR is better than ByteRobust's, and the advantage is more pronounced in simpler tasks like Qwen3-8B-Math. The reason is that the longer training time is, the more negligible the restart time. In addition, although ByteRobust would lose the progress during the restart, the re-execution of rollout is also counted towards ETTR. So the gap of average ETTR between ByteRobust and RobustRL is not large.

We further show the sliding ETTR for Qwen3-8B-Math in Figure 12 with semi-sync training mode. We sample the training process every 5 minutes. We take the process of 0.8-2h as an example. Since ByteRobust restarts both the trainer and rollouts when a fault occurs, its ETTR is significantly lower at 20%. During the same period, RobustRL's rollouts continue to execute, so the influence of failure is less. Compared to ByteRobust, RobustRL improves ETTR by 18-24% on the Qwen3-8B-Math task.

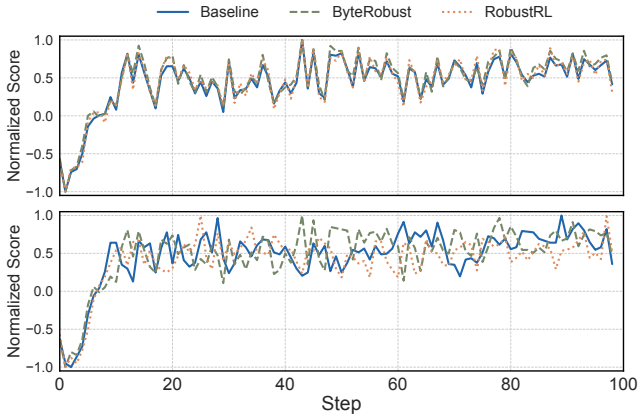


Figure 13: Training consistency in sync (row 1) and semi-sync (row 2) mode. The reward of the answer is normalized.

Training trend. We use the training trend of Qwen3-8B-Math with sync and semi-sync to verify the correctness of training in the presence of failures. We execute 100 training steps and obtain curves with similar training trends and the result is shown in Figure 13. In sync training, the three methods has the similar score tendency because we can make sure the training order of the prompt by batch mode instead of the streaming pipeline mode [89][3]. When the error happens, we can get the prompt and response from `RequestManager` to continue the training in RobustRL. However, the result in sync does not get fully alignment because we do not enable the deterministic inference [69] with FlashAttention [9] and CUDA atomic operations in our evaluation. In semi-sync situation, the results are still not deterministic and the trend difference is larger because the async schedule mode must use the streaming schedule, which cannot guarantee the order of the prompt. But generally, the three methods in semi-sync training shows the similar tendency.

[3]Batch mode is waiting for all the prompts in this batch to finish. Streaming mode is adding a new prompt to rollout if one prompt in this batch finishes.

Comparison under different error injection frequency. We further evaluate the performance of RobustRL and ByteRobust under diverse fault frequencies shown in Figure 14. All experiments are conducted on a 64-GPU in semi-sync mode with the same number of trainers and rollout for Qwen3-8B-Math workload and each experiment runs for 50 training steps. Notably, ByteRobust adopts a fixed checkpoint interval of one snapshot per 5 steps, and faults are uniformly injected throughout the entire training timeline.

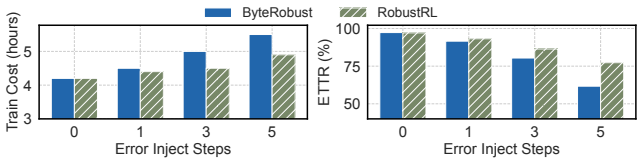


Figure 14: End-to-end training time and ETTR comparison of ByteRobust under different error injection frequency.

In the fault-free scenario, RobustRL and ByteRobust achieve comparable end-to-end training time and ETTR. Although RobustRL adopts a one-step checkpoint, the overall time for a small proportion of the full RL training §7.4. We provide an in-depth breakdown of checkpoint overhead in the following overhead analysis section. As the injected fault frequency increases from 1 to 5 (2%-10% in 50 steps), RobustRL outperforms ByteRobust by 2.2%–12.2% in end-to-end completion time reduction and 2.7%–15.7% in ETTR improvement because the time gain of RobustRL compared to ByteRobust is highly depends on the fault frequency. Considering that the performance advantage of RobustRL approaches saturation at a 2% fault rate, we omit additional evaluations for even lower fault frequencies.

7.3 Robust Benefits Analysis

Trainer restart benefit. Figure 15 shows a comparison of restart time breakdown between ByteRobust and RobustRL for different model sizes and numbers of GPUs. We use the semi-sync case as an example because its startup time is the longest, as the trainer machines need both the training and inference engines, in addition to the rollout roles. The main bottlenecks in starting an RL training task consist of four stages: instance restart, rollout initialization, checkpoint loading, and worker initialization. The "restart instance" stage includes container startup, installation of third-party libraries after startup, Kubernetes scheduling *et al.*. The "worker init" stage is primarily the initialization of the training engine. The "rollout init" stage is the initialization process for the standalone rollout inference engine, not including weight synchronization. The "load checkpoint" is the stage to load from memory to the GPU. When we detect a model anomaly, we asynchronously load the corresponding shards from HDFS to the local memory. The overhead from "rollout init" and "restart instance" phases does not exist in RobustRL, making it faster than ByteRobust. Benefited by the warmup by rollout

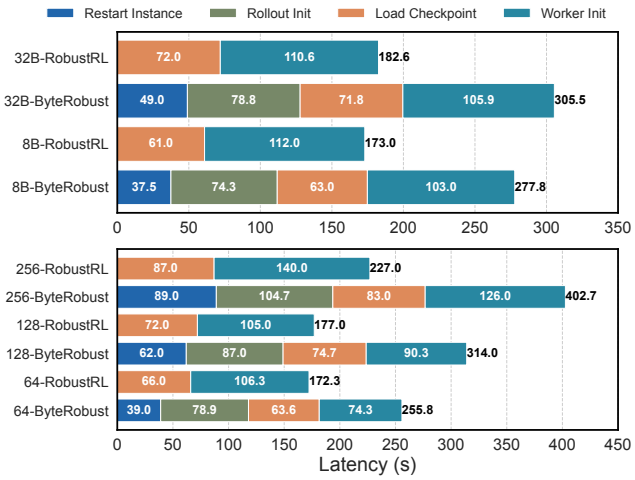


Figure 15: Restart cost comparison of ByteRobust and RobustRL. Row 1 is the result with different model size and row 2 is the result of 8B with different number of GPUs.

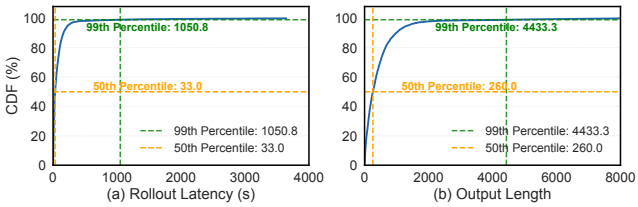


Figure 16: Benefit from preserving the rollout progress (a) Rollout cost CDF and (b) Rollout length CDF of all prompts in Qwen3-32B-SWE.

mechanism in §5.1.3, we avoid the trainer's gang scheduling process when the trainer has machine error because we can schedule the rollout machines to replace.

The second row of Figure 15 shows the results of restart latency of ByteRobust and RobustRL as the number of GPUs increases, where the duration of all initialization stages increases accordingly. The "worker init" overhead in RobustRL is larger than in ByteRobust. Compared to ByteRobust's startup, the trainer's restart includes a destruction phase, which introduces additional overhead for destroying network connections. Overall, RobustRL improves restart efficiency by a factor of 1.5-1.7× compared to ByteRobust.

Preserving rollout progress benefit. In addition to the restart benefit, RobustRL can preserve rollout progress compared to ByteRobust. We recorded the output time distribution and output length distribution for 50k prompts (batch size × steps = 512 × 100) on the Qwen3-32B-SWE task shown in Figure 16. The tail latency reaches 1050s, with an output length of over 4k tokens. RobustRL can avoid the overhead of re-executing this inference. When we extend to larger RL training scenarios involving more conversation turns and more complex tool invocation logic, the benefit can be more significant.

Rollout fault tolerance benefit. A machine failure on a rollout or a machine being borrowed by the trainer does not

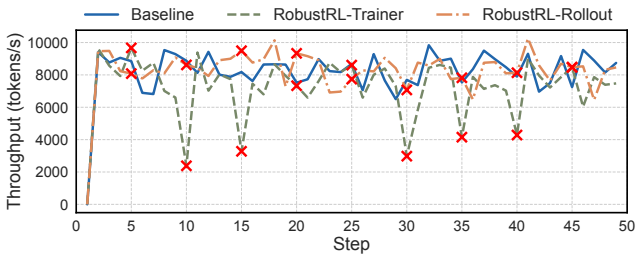


Figure 17: Rollout token throughput of Qwen3-8B-Math with 50 steps. We inject the error each five steps.

influence the RL task. There are multiple rollout instances, and in semi-sync mode, the trainer can also provide rollout capabilities. Compared to the over 300s required for a RL task restart shown in Figure 15, the impact of a rollout restart would not influence the throughput of token decode as shown in Figure 17. In the event of a rollout machine failure in 32B model, the time for a single rollout to start up and begin providing service includes machine scheduling (30s), container startup (under 30s), inference engine startup (49s), and weight synchronization (10s), for a total of 119s.

Throughput under the trainer and rollout failure. We visualize the rollout throughput of semi-sync training in Figure 17. The failure in trainer machines can lead to the throughput decrease because the error can happen when rollout phase finishes. In this phase, the trajectories wait for the consumption by the trainer and continue the next step rollout. The token throughput of rollout failure is not affected because we have multiple rollout replicas.

Detection benefit. ByteRobust [73] applies the 30 seconds interval for network-related fault detection and 10 seconds for the GPU ones (setup in its Table 1). As discussed in §1, it would lead to endless restart of the task since the RL task can have idle time waiting for the response from the tools. The RobustRL can continue the running process. Compared with the cluster-level detection in Figure 2(b), it can save at most about 1000 seconds as shown in Figure 16(a) since we set the hang detection interval for rollout as 60 seconds.

7.4 Robust Overhead Analysis

7.4.1 Weight Synchronization of Trainer and Rollout

Weight synchronization efficiency. Figure 18 shows the weight synchronization efficiency for an equal number of trainers and rollouts. The 235B model requires more than 64 GPUs for the trainer to start as shown in Figure 18(b). The NCCL communication method first gathers all weight to rank 0 and then broadcasts the weight to all the rollouts. The UCX uses point-to-point communication to the corresponding ranks in rollouts. Since UCX can fully utilize all the bandwidth of the NICs, it can have close performance to the NCCL. In addition, UCX allows for dynamic connections, providing fault tolerance capabilities. For a 235B model with FP16 precision, the model size is 470GB. With the 4×200Gbps NICs, the theoretical transfer cost is $\frac{235\times2}{4\times200/8} = 4.7$s. Our UCX-based weight synchronization cost is about 6s. The extra overhead can be introduced by the network bandwidth fluctuations and the async bubbles between NICs in one machine.

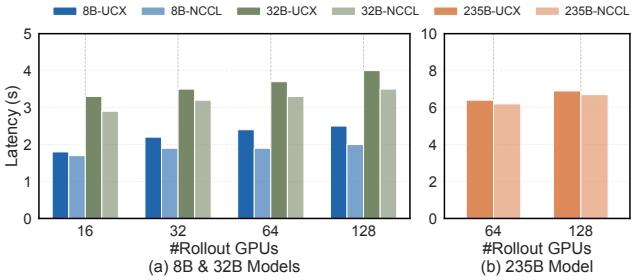


Figure 18: Weight synchronization latency with different number of GPUs and model size. The number of GPUs of trainer and rollout is equal. (a) 8B and 32B and (b) 235B.

We further evaluate the pulling efficiency of our weight synchronization strategy in Figure 19. Our UCX-based transfer can achieve a linear increase in cost even when the number of rollouts grows exponentially. Benefit by the joining as relay design (§5.2.1), the outdated or recovered rollout can pull the weight from the relay server. On the contrary NCCL-based weight synchronization strategy does not support dynamic connection with relay server and all the rollout must pull the weight from the trainer. When the number of rollouts is larger than the trainer, its efficiency decreases.

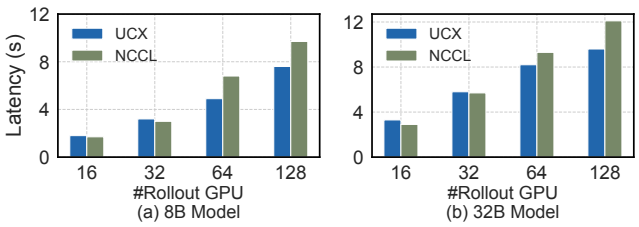


Figure 19: Weight synchronization latency with 16 trainer GPUs and different number of rollout GPUs. (a) 8B and (b) 32B models.

7.4.2 Per-step Checkpoint

Checkpoint efficiency. We evaluate the checkpoint overhead of both FSDP2 and Megatron in Figure 20. It shows the GPU to memory and memory to disk latency with different model size and number of GPUs for checkpoint. The GPU to memory and memory to disk are independent of the model size and the number of GPUs because the saving process only needs to save the shard corresponding to each rank. The 3s blocking time is 1% of the minutes or hours required for one step in the RL task. It guarantees that a per-step GPU-to-memory checkpoint would not lead to the OOM during the process of checkpoint from GPU to memory because each step takes

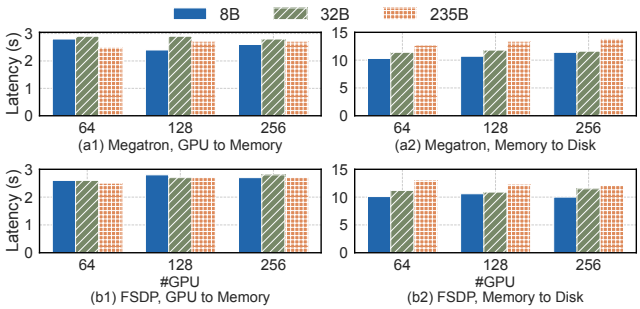


Figure 20: Checkpoint latency of ByteCheckpoint. (a) GPU to memory and (b) memory to disk with different model size and number of GPUs.

over minutes. We have enough time to offload the checkpoint from memory to the disk for about 10s. The disk writing stage is non-blocking, which further minimizes the overhead of checkpoint for the RL training.

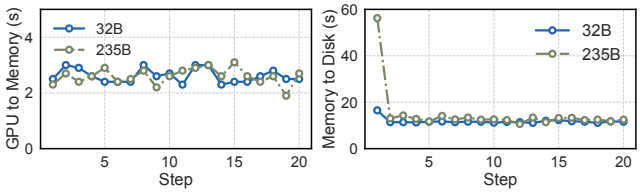


Figure 21: Checkpoint overhead of each step in GPU to memory (left) and memory to HDFS (right).

Continuous checkpoint. In Figure 21, we analyze the latency of GPU-to-memory and memory-to-disk across the first 20 training steps for the 32B model. For the 235B model, checkpoint saving is executed at a consistent 10-minute interval. Consistent with the observations, the GPU-to-memory latency is independent of model scale and training steps. This is because the GPU-to-memory address mapping in ByteCheckpoint [72] is pinned during the initialization phase.

In contrast, for the memory-to-disk persistence phase in Bytecheckpoint, the first weight saving needs extra full traversal of all model tensors, global shard metadata generation, distributed execution planning. Larger models introduce more tensors, larger metadata footprints, and greater volumes of data to be written, leading to a strong correlation between persistence overhead and model scale. Starting from the second checkpoint, however, distributed planning and metadata are fully cached. Redundant operations including tensor traversal, metadata parsing, cross-node communication, and structural construction are eliminated, leaving only raw data copying and disk I/O. This process is no longer affected by model size, but bounded by storage bandwidth for each shard.

Checkpoint frequency > 1. We further evaluate the reward trends under different checkpoint frequencies with a failure injected every 10 steps shown in Figure 22. All experiments are conducted in semi-synchronous mode on the math task using

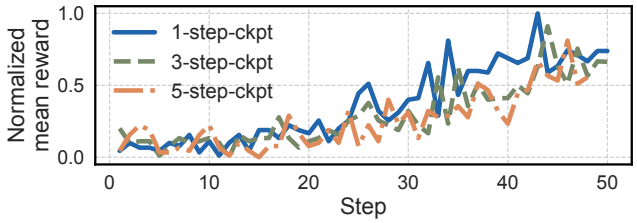


Figure 22: Reward under different checkpoint frequency with 10-step-frequency failures.

the Qwen3-8B-Base model. For a 3-step checkpoint interval, if a failure occurs at the 10th step, the rollback checkpoint is restored from the 9th step, resulting in 1 offline step. Similarly, a 5-step checkpoint interval introduces 5 offline steps when a failure happens at step 10. The experimental results demonstrate that the number of offline steps directly affects the upward trend of reward growth. Such degradation can be more severe for large-scale, highly complex reinforcement learning training workloads like search [31] and coding [30]. Given that the overhead of per-step checkpoint is negligible, we prioritize preserving algorithmic fidelity and training effectiveness in our design.

8 Discussion and Limitation

Limitation of Test Scaling. Frequent failures only emerge in large-scale training scenarios. Under small-scale settings with infrequent failures like 2%, RobustRL exhibits no substantial performance gains over ByteRobust because the benefit is highly dependent on the failure frequency and end-to-end training time. Even so, our experiments simulate large-scale frequent failure conditions via small-scale cluster testbeds, which adequately demonstrates the inherent advantages of RobustRL for future large-scale workloads.

Limitations of Diagnosis Tools. Current fault diagnosis tools are primarily designed for training [11, 38, 66, 73, 74], lacking specific capabilities for RL post-training scenarios. In RL, the presence of multiple roles and complex data and control dependencies makes root cause localization challenging. For instance, an OOM error in one role might be caused by a memory leak in another co-located role in the same machine. Similarly, a role hang event could be due to abnormal metrics from other roles. More precise RL training system diagnostic tools would help developers pinpoint root causes faster during failures, thereby further improving RL training efficiency.

Role-based Hot Updates. Adjusting parameters during training is a common requirement. For example, increasing the batch size when GPU memory utilization is found to be low. ByteRobust [73] proposes a hot update mechanism that allows instances to restart and update in-place, avoiding machine rescheduling. Building upon role-based fault tolerance for hot updates, it helps developers achieve faster role-specific parameter updates. RobustRL reduces restart overhead and

prevents the loss of rollout.

Elastic RL Training. Existing work supports elasticity for the rollout [79, 89]. Since the rollout performs inference tasks and uses independent scheduling, elastic scaling does not affect task execution. With the support of RobustRL, the trainer's elasticity in the data-parallel dimension can adopt a similar approach to previous work [8, 23, 36, 86], requiring only a trainer restart, thus reducing the overhead of elastic scaling.

9 Related Work

Fault tolerance systems for LLM training. For training systems, their main goal is to locate system failures and recover as soon as possible through fault detection [10, 11, 29, 38, 73, 78], checkpoint [47, 72], and elastic recovery [8, 36] to ensure the correctness of model training and reduce task restart overhead. Some works have further extended to elastic training scenarios, focusing on reducing the overhead of scaling due to resource changes. Examples include pipeline redundancy [27] and parallelism adjustment prediction [70] or in spot instance [4]. However, the fault detection strategies for pre-train are not incompatible in RL. For the trainer robust work like Torch-FT [45, 56], it is orthogonal to us since it can accelerate the trainer restart efficiency. But it does not consider the recovery of the trainer from partial failures in conjunction with rollouts, nor the design opportunities presented by the asynchronous execution of rollout.

Fault tolerance systems for LLM inference Existing inference fault tolerance systems primarily operate at the token, and rank levels. The token level refers to utilizing the KV Cache [18, 34, 54] to avoid re-prefilling requests. The rank level refers to scenarios with AF (attention-FFN) disaggregation [39, 68, 92], where the failure of a sub-ranks is prevented from causing a full service crash. For example, EaaS considers fault tolerance and recovery after an expert machine fails. However, async RL training scenarios also need to consider the interaction between rollouts and the trainer. Additionally, as an offline training task, we just need to guarantee the prompt can be generated.

RL Systems and Fault Tolerance. RL involves both training and inference stages. Considering the long-tail phenomenon of rollout, RL training paradigms have shifted from sync [43, 61, 90] to async mode [16, 60, 89, 93] to improve training efficiency. Some RL training works have considered the fault tolerance and elasticity of rollouts [60, 79]. However, the probability of failure in the rollout phase is much lower than in the training ones because the communication management of training is more complex. RobustRL further considers the case of trainer fault tolerance, utilizing rollouts as warm standbys and allowing them to continue inference during a trainer failure.

10 Conclusion

We have implemented RobustRL, the first fault tolerance system for RL training that supports all GPU roles against machine failures. Through techniques in *detect-restart-reconnect*, RobustRL can detect the fault by roles quickly, isolate the failure and minimize the restart overhead with rollout progress preserving. RobustRL can achieve 80% ETTR on the 8B-Math training task with 20% higher than ByteRobust on the Qwen3-8B-Math task in an extremely robust case.

Acknowledgements

We thank OSDI anonymous shepherd and reviewers for their insightful and constructive feedback. This work was supported in part by the National Science Foundation of China under Grants (62472375, 62125206), and in part by Zhejiang Provincial Natural Science Foundation of China under Grant No. LD24F020014 and No. LD25F020002, and in part by the Zhejiang Pioneer (Jianbing) Project (2024C01032). The corresponding author is Xinkui Zhao.

References

[1] Nvidia collective communications library (nccl), 2023.

[2] Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, et al. Gpt-4 technical report. *arXiv preprint arXiv:2303.08774*, 2023.

[3] Arash Ahmadian, Chris Cremer, Matthias Gallé, Marzieh Fadaee, Julia Kreutzer, Olivier Pietquin, Ahmet Üstün, and Sara Hooker. Back to basics: Revisiting reinforce style optimization for learning from human feedback in llms, 2024.

[4] Sanjith Athlur, Nitika Saran, Muthian Sivathanu, Ramachandran Ramjee, and Nipun Kwatra. Varuna: Scalable, low-cost training of massive deep learning models. In *Proceedings of the Seventeenth European Conference on Computer Systems*, EuroSys '22, page 472–487, New York, NY, USA, 2022. Association for Computing Machinery.

[5] Yuntao Bai, Andy Jones, Kamal Ndousse, Amanda Askell, Anna Chen, Nova DasSarma, Dawn Drain, Stanislav Fort, Deep Ganguli, Tom Henighan, et al. Training a helpful and harmless assistant with reinforcement learning from human feedback. *arXiv preprint arXiv:2204.05862*, 2022.

[6] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. Language models are few-shot learners. *Advances in neural information processing systems*, 33:1877–1901, 2020.

[7] ByteDance Seed. Technical Introduction to the Seed1.6 Model Series. https://seed.bytedance.com/en/seed1_6, 2025.

[8] Zhenqian Chen, Xinkui Zhao, Chen Zhi, and Jianwei Yin. Deepboot: Dynamic scheduling system for training and inference deep learning tasks in GPU cluster. *IEEE Trans. Parallel Distributed Syst.*, 34(9):2553–2567, 2023.

[9] Tri Dao, Daniel Y. Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. Flashattention: Fast and memory-efficient exact attention with io-awareness. In Sanmi Koyejo, S. Mohamed, A. Agarwal, Danielle

Belgrave, K. Cho, and A. Oh, editors, *Advances in Neural Information Processing Systems 35: Annual Conference on Neural Information Processing Systems 2022, NeurIPS 2022, New Orleans, LA, USA, November 28 - December 9, 2022*, 2022.

[10] Yangtao Deng, Xiang Shi, Zhuo Jiang, Xingjian Zhang, Lei Zhang, Zhang Zhang, Bo Li, Zuquan Song, Hang Zhu, Gaohong Liu, Fuliang Li, Shuguang Wang, Haibin Lin, Jianxi Ye, and Minlan Yu. Minder: Faulty machine detection for large-scale distributed model training. In *22nd USENIX Symposium on Networked Systems Design and Implementation (NSDI 25)*, pages 505–521, Philadelphia, PA, April 2025. USENIX Association.

[11] Yangtao Deng, Lei Zhang, Qinlong Wang, Xiaoyun Zhi, Xinlei Zhang, Zhuo Jiang, Haohan Xu, Lei Wang, Zuquan Song, Gaohong Liu, et al. Mycroft: Tracing dependencies in collective communication towards reliable llm training. *arXiv preprint arXiv:2509.03018*, 2025.

[12] DLPack. DLPack: Open In Memory Tensor Structure. https://github.com/dmlc/dlpack, 2024.

[13] Jiangfei Duan, Ziang Song, Xupeng Miao, Xiaoli Xi, Dahua Lin, Harry Xu, Minjia Zhang, and Zhihao Jia. Parcae: Proactive, liveput-optimized dnn training on preemptible instances. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*, pages 1121–1139, 2024.

[14] Jiazhan Feng, Shijue Huang, Xingwei Qu, Ge Zhang, Yujia Qin, Baoquan Zhong, Chengquan Jiang, Jinxin Chi, and Wanjun Zhong. Retool: Reinforcement learning for strategic tool use in llms. *arXiv preprint arXiv: 2504.11536*, 2025.

[15] Jason Flinn, Xianzheng Dou, Arushi Aggarwal, Alex Boyko, Francois Richard, Eric Sun, Wendy Tobagus, Nick Wolchko, and Fang Zhou. Owl: Scale and flexibility in distribution of hot content. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 1–15, Carlsbad, CA, July 2022. USENIX Association.

[16] Wei Fu, Jiaxuan Gao, Xujie Shen, Chen Zhu, Zhiyu Mei, Chuyi He, Shusheng Xu, Guo Wei, Jun Mei, Jiashu Wang, Tongkai Yang, Binhang Yuan, and Yi Wu. AReaL: A Large-Scale Asynchronous Reinforcement Learning System for Language Reasoning, June 2025.

[17] Swapnil Gandhi, Mark Zhao, Athinagoras Skiadopoulos, and Christos Kozyrakis. Recycle: Resilient training of large dnns using pipeline adaptation. In *Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles*, pages 211–228, 2024.

[18] Bin Gao, Zhuomin He, Puru Sharma, Qingxuan Kang, Djordje Jevdjic, Junbo Deng, Xingkun Yang, Zhou Yu, and Pengfei Zuo. Cost-efficient large language model serving for multi-turn conversations with cachedattention. In Saurabh Bagchi and Yiying Zhang, editors, *Proceedings of the 2024 USENIX Annual Technical Conference, USENIX ATC 2024, Santa Clara, CA, USA, July 10-12, 2024*, pages 111–126. USENIX Association, 2024.

[19] Hao Ge, Junda Feng, Qi Huang, Fangcheng Fu, Xiaonan Nie, Lei Zuo, Haibin Lin, Bin Cui, and Xin Liu. Bytescale: Efficient scaling of llm training with a 2048k context length on more than 12,000 gpus. *arXiv preprint arXiv:2502.21231*, 2025.

[20] Github. OpenRLHF: An Easy-to-use, Scalable and High-performance RLHF Framework. https://github.com/OpenRLHF/OpenRLHF, 2022.

[21] Github. ROLL: Reinforcement Learning Optimization for Large-Scale Learning. https://github.com/alibaba/ROLL, 2022.

[22] Github. verl: Volcano Engine Reinforcement Learning for LLMs. https://github.com/volcengine/verl, 2022.

[23] Diandian Gu, Yihao Zhao, Yinmin Zhong, Yifan Xiong, Zhenhua Han, Peng Cheng, Fan Yang, Gang Huang, Xin Jin, and Xuanzhe Liu. Elasticflow: An elastic serverless training platform for distributed deep learning. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, pages 266–280, 2023.

[24] Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Ruoyu Zhang, Runxin Xu, Qihao Zhu, Shirong Ma, Peiyi Wang, Xiao Bi, et al. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning. *arXiv preprint arXiv:2501.12948*, 2025.

[25] Zhenyu Han, Ansheng You, Haibo Wang, Kui Luo, Guang Yang, Wenqi Shi, Menglong Chen, Sicheng Zhang, Zeshun Lan, Chunshi Deng, Huazhong Ji, Wenjie Liu, Yu Huang, Yixiang Zhang, Chenyi Pan, Jing Wang, Xin Huang, Chunsheng Li, and Jianping Wu. Asyncflow: An asynchronous streaming rl framework for efficient llm post-training, 2025.

[26] Jian Hu, Xibin Wu, Zilin Zhu, Xianyu, Weixun Wang, Dehao Zhang, and Yu Cao. Openrlhf: An easy-to-use, scalable and high-performance rlhf framework, 2024.

[27] Insu Jang, Zhenning Yang, Zhen Zhang, Xin Jin, and Mosharaf Chowdhury. Oobleck: Resilient distributed training of large models using pipeline templates. In *Proceedings of the 29th Symposium on Operating Systems Principles*, pages 382–395, 2023.

[28] Yuxuan Jiang, Ziming Zhou, Boyu Xu, Beijie Liu, Runhui Xu, and Peng Huang. Training with confidence: Catching silent errors in deep learning training with automated proactive checks. In Lidong Zhou and Yuanyuan Zhou, editors, *19th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2025, Boston, MA, USA, July 7-9, 2025*, pages 313–329. USENIX Association, 2025.

[29] Ziheng Jiang, Haibin Lin, Yinmin Zhong, Qi Huang, Yangrui Chen, Zhi Zhang, Yanghua Peng, Xiang Li, Cong Xie, Shibiao Nong, Yulu Jia, Sun He, Hongmin Chen, Zhihao Bai, Qi Hou, Shipeng Yan, Ding Zhou, Yiyao Sheng, Zhuo Jiang, Haohan Xu, Haoran Wei, Zhang Zhang, Pengfei Nie, Leqi Zou, Sida Zhao, Liang Xiang, Zherui Liu, Zhe Li, Xiaoying Jia, Jianxi Ye, Xin Jin, and Xin Liu. MegaScale: Scaling large language model training to more than 10,000 GPUs. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*, pages 745–760, Santa Clara, CA, April 2024. USENIX Association.

[30] Carlos E. Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. Swe-bench: Can language models resolve real-world github issues? *arXiv preprint arXiv: 2310.06770*, 2023.

[31] Bowen Jin, Hansi Zeng, Zhenrui Yue, Jinsung Yoon, Sercan Arik, Dong Wang, Hamed Zamani, and Jiawei Han. Search-r1: Training llms to reason and leverage search engines with reinforcement learning. *arXiv preprint arXiv:2503.09516*, 2025.

[32] Komal Kumar, Tajamul Ashraf, Omkar Thawakar, Rao Muhammad Anwer, Hisham Cholakkal, Mubarak Shah, Ming-Hsuan Yang, Phillip HS Torr, Fahad Shahbaz Khan, and Salman Khan. Llm post-training: A deep dive into reasoning large language models. *arXiv preprint arXiv:2502.21321*, 2025.

[33] Tom Kwiatkowski, Jennimaria Palomaki, Olivia Redfield, Michael Collins, Ankur Parikh, Chris Alberti, Danielle Epstein, Illia Polosukhin, Jacob Devlin, Kenton Lee, Kristina Toutanova, Llion Jones, Matthew Kelcey, Ming-Wei Chang, Andrew M. Dai, Jakob Uszkoreit, Quoc Le, and Slav Petrov. Natural questions: A benchmark for question answering research. *Transactions of the Association for Computational Linguistics*, 7:452–466, 2019.

[34] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph Gonzalez, Hao Zhang, and Ion Stoica. Efficient memory management for large language model serving with pagedattention. In Jason Flinn, Margo I. Seltzer, Peter Druschel, Antoine Kaufmann, and Jonathan Mace, editors, *Proceedings of the 29th Symposium on Operating Systems Principles, SOSP 2023, Koblenz, Germany, October 23-26, 2023*, pages 611–626. ACM, 2023.

[35] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph Gonzalez, Hao Zhang, and Ion Stoica. Efficient memory management for large language model serving with pagedattention. In *Proceedings of the 29th Symposium on Operating Systems Principles*, pages 611–626, 2023.

[36] Mingzhen Li, Wencong Xiao, Hailong Yang, Biao Sun, Hanyu Zhao, Shiru Ren, Zhongzhi Luan, Xianyan Jia, Yi Liu, Yong Li, et al. Easyscale: Elastic training with consistent accuracy and improved utilization on gpus. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–14, 2023.

[37] Nándor Licker, Kevin Hu, Vladimir Zaytsev, and Lequn Chen. RDMA point-to-point communication for LLM systems. *CoRR*, abs/2510.27656, 2025.

[38] Jinkun Lin, Ziheng Jiang, Zuquan Song, Sida Zhao, Menghan Yu, Zhanghan Wang, Chenyuan Wang, Zuocheng Shi, Xiang Shi, Wei Jia, Zherui Liu, Shuguang Wang, Haibin Lin, Xin Liu, Aurojit Panda, and Jinyang Li. Understanding stragglers in large model training using what-if analysis. In Lidong Zhou and Yuanyuan Zhou, editors, *19th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2025, Boston, MA, USA, July 7-9, 2025*, pages 483–498. USENIX Association, 2025.

[39] Ziming Liu, Boyu Tian, Guoteng Wang, Zhen Jiang, Peng Sun, Zhenhua Han, Tian Tang, Xiaohe Hu, Yanmin Jia, Yan Zhang, He Liu, Mingjun Zhang, Yiqi Zhang, Qiaoling Chen, Shenggan Cheng, Mingyu Gao, Yang You, and Siyuan Feng. Expert-as-a-service: Towards efficient, scalable, and robust large-scale moe serving. *CoRR*, abs/2509.17863, 2025.

[40] Chang Lou, Dimas Shidqi Parikesit, Yujin Huang, Zhewen Yang, Senapati Diwangkara, Yuzhuo Jing, Achmad Imam Kistijantoro, Ding Yuan, Suman Nath, and Peng Huang. Deriving semantic checkers from tests to detect silent failures in production distributed systems. In Lidong Zhou and Yuanyuan Zhou, editors, *19th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2025, Boston, MA, USA, July 7-9, 2025*, pages 19–38. USENIX Association, 2025.

[41] Yunchi Lu, Youshan Miao, Cheng Tan, Peng Huang, Yi Zhu, Xian Zhang, and Fan Yang. Trainverify: Equivalence-based verification for distributed LLM training. In Youjip Won, Youngjin Kwon, Ding Yuan, and Rebecca Isaacs, editors, *Proceedings of the ACM SIGOPS 31st Symposium on Operating Systems Principles, SOSP 2025, Lotte Hotel World, Seoul, Republic of Korea, October 13-16, 2025*, pages 237–253. ACM, 2025.

[42] Michael Luo, Sijun Tan, Roy Huang, Ameen Patel, Alpay Ariyak, Qingyang Wu, Xiaoxiang Shi, Rachel Xin, Colin Cai, Maurice Weber, Ce Zhang, Li Erran Li, Raluca Ada Popa, and Ion Stoica. Deepcoder: A fully open-source 14b coder at o3-mini level. https://www.together.ai/blog/deepcoder, 2025. Notion Blog.

[43] Zhiyu Mei, Wei Fu, Kaiwei Li, Guangju Wang, Huanchen Zhang, and Yi Wu. Real: Efficient rlhf training of large language models with parameter reallocation. *arXiv preprint arXiv: 2406.14088*, 2024.

[44] Meituan search. Recipe: Fully Async Policy Trainer. https://verl.readthedocs.io/en/latest/advance/fully_async.html, 2024.

[45] Meta. Easy per step fault tolerance for pytorch. https://meta-pytorch.org/torchft, 2026.

[46] Xupeng Miao, Chunan Shi, Jiangfei Duan, Xiaoli Xi, Dahua Lin, Bin Cui, and Zhihao Jia. Spotserve: Serving generative large language models on preemptible instances. In Rajiv Gupta, Nael B. Abu-Ghazaleh, Madan Musuvathi, and Dan Tsafrir, editors, *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2, ASPLOS 2024, La Jolla, CA, USA, 27 April 2024- 1 May 2024*, pages 1112–1127. ACM, 2024.

[47] Jayashree Mohan, Amar Phanishayee, and Vijay Chidambaram. CheckFreq: Frequent, Fine-Grained DNN checkpointing. In *19th USENIX Conference on File and Storage Technologies (FAST 21)*, pages 203–216. USENIX Association, February 2021.

[48] Philipp Moritz, Robert Nishihara, Stephanie Wang, Alexey Tumanov, Richard Liaw, Eric Liang, Melih Elibol, Zongheng Yang, William Paul, Michael I Jordan, et al. Ray: A distributed framework for emerging {AI} applications. In *13th USENIX symposium on operating systems design and implementation (OSDI 18)*, pages 561–577, 2018.

[49] Deepak Narayanan, Mohammad Shoeybi, Jared Casper, Patrick LeGresley, Mostofa Patwary, Vijay Korthikanti, Dmitri Vainbrand, Prethvi Kashinkunti, Julie Bernauer, Bryan Catanzaro, et al. Efficient large-scale language model training on gpu clusters using megatron-lm. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–15, 2021.

[50] Mathematical Association of America. Aime 2024, 2024.

[51] Ryosuke Okuta, Yuya Unno, Daisuke Nishino, Shohei Hido, and Crissman Loomis. Cupy: A numpy-compatible library for nvidia gpu calculations. In *Proceedings of Workshop on Machine Learning Systems (LearningSys) in The Thirty-first Annual Conference on Neural Information Processing Systems (NIPS)*, 2017.

[52] OpenAI. Introducing OpenAI o1. https://openai.com/o1/, 2024.

[53] Long Ouyang, Jeffrey Wu, Xu Jiang, Diogo Almeida, Carroll Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, Alex Ray, et al. Training language models to follow instructions with human feedback. *Advances in Neural Information Processing Systems*, 35:27730–27744, 2022.

[54] Ruoyu Qin, Zheming Li, Weiran He, Jialei Cui, Feng Ren, Mingxing Zhang, Yongwei Wu, Weimin Zheng, and Xinran Xu. Mooncake: Trading more storage for less computation—a {KVCache-centric} architecture for serving {LLM} chatbot. In *23rd USENIX Conference on File and Storage Technologies (FAST 25)*, pages 155–170, 2025.

[55] Scott Reed, Konrad Zolna, Emilio Parisotto, Sergio Gomez Colmenarejo, Alexander Novikov, Gabriel Barth-Maron, Mai Gimenez, Yury Sulsky, Jackie Kay, Jost Tobias Springenberg, Tom Eccles, Jake Bruce, Ali Razavi, Ashley Edwards, Nicolas Heess, Yutian Chen, Raia Hadsell, Oriol Vinyals, Mahyar Bordbar, and Nando de Freitas. A Generalist Agent, November 2022.

[56] Omkar Salpekar, Rohan Varma, Kenny Yu, Vladimir Ivanov, Yang Wang, Ahmed Sharif, Min Si, Shawn Xu, Feng Tian, Shengbao Zheng, Tristan Rice, Ankush Garg, Shangfu Peng, Shreyas Siravara, Wenyin Fu, Rodrigo de Castro, Adithya Gangidi, Andrey Obraztsov, Sharan Narang, Sergey Edunov, Maxim Naumov, Chunqiang Tang, and Mathew Oldham. Training llms with fault tolerant HSDP on 100,000 gpus. *CoRR*, abs/2602.00277, 2026.

[57] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017.

[58] ByteDance Seed, Jiaze Chen, Tiantian Fan, Xin Liu, Lingjun Liu, Zhiqi Lin, Mingxuan Wang, Chengyi Wang, Xiangpeng Wei, Wenyuan Xu, et al. Seed1. 5-thinking: Advancing superb reasoning models with reinforcement learning. *arXiv preprint arXiv:2504.13914*, 2025.

[59] Zhihong Shao, Peiyi Wang, Qihao Zhu, Runxin Xu, Junxiao Song, Xiao Bi, Haowei Zhang, Mingchuan Zhang, YK Li, Y Wu, et al. Deepseekmath: Pushing the limits of mathematical reasoning in open language models. *arXiv preprint arXiv:2402.03300*, 2024.

[60] Guangming Sheng, Yuxuan Tong, Borui Wan, Wang Zhang, Chaobo Jia, Xibin Wu, Yuqi Wu, Xiang Li, Chi Zhang, Yanghua Peng, Haibin Lin, Xin Liu, and Chuan Wu. Laminar: A scalable asynchronous RL post-training framework. *CoRR*, abs/2510.12633, 2025.

[61] Guangming Sheng, Chi Zhang, Zilingfeng Ye, Xibin Wu, Wang Zhang, Ru Zhang, Yanghua Peng, Haibin Lin, and Chuan Wu. Hybridflow: A flexible and efficient rlhf framework. In *Proceedings of the Twentieth European Conference on Computer Systems*, EuroSys '25, page 1279–1297, New York, NY, USA, 2025. Association for Computing Machinery.

[62] Biao Sun, Ziming Huang, Hanyu Zhao, Wencong Xiao, Xinyi Zhang, Yong Li, and Wei Lin. Llumnix: Dynamic scheduling for large language model serving. In *18th USENIX symposium on operating systems design and implementation (OSDI 24)*, pages 173–191, 2024.

[63] Gemini Team, Petko Georgiev, Ving Ian Lei, Ryan Burnell, Libin Bai, Anmol Gulati, Garrett Tanzer, Damien Vincent, Zhufeng Pan, Shibo Wang, et al. Gemini 1.5: Unlocking multimodal understanding across millions of tokens of context. *arXiv preprint arXiv:2403.05530*, 2024.

[64] Kimi Team, Angang Du, Bofei Gao, Bowei Xing, Changjiu Jiang, Cheng Chen, Cheng Li, Chenjun Xiao, Chenzhuang Du, Chonghua Liao, et al. Kimi k1. 5: Scaling reinforcement learning with llms. *arXiv preprint arXiv:2501.12599*, 2025.

[65] Meituan LongCat Team. Longcat-flash-thinking technical report. *CoRR*, abs/2509.18883, 2025.

[66] NVIDIA Team. NVIDIA DCGM. https://developer.nvidia.com/dcgm, 2021.

[67] Qwen Team, An Yang, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chengyuan Li, Dayiheng Liu, Fei Huang, Haoran Wei, Huan Lin, Jian Yang, Jianhong Tu, Jianwei Zhang, Jianxin Yang, Jiaxi Yang, Jingren Zhou, Junyang Lin, Kai Dang, Keming Lu, Keqin Bao, Kexin Yang, Le Yu, Mei Li, Mingfeng Xue, Pei Zhang, Qin Zhu, Rui Men, Runji Lin, Tianhao Li, Tianyi Tang, Tingyu Xia, Xingzhang Ren, Xuancheng Ren, Yang Fan, Yang Su, Yichang Zhang, Yu Wan, Yuqiong Liu, Zeyu Cui, Zhenru Zhang, and Zihan Qiu. Qwen2.5 technical report. *arXiv preprint arXiv: 2412.15115*, 2024.

[68] Stepfun Team. Step-3 is large yet affordable: Model-system co-design for cost-effective decoding. *CoRR*, abs/2507.19427, 2025.

[69] Thinking Machines. Defeating Nondeterminism in LLM Inference. https://thinkingmachines.ai/blog/defeating-nondeterminism-in-llm-inference/, 2025.

[70] John Thorpe, Pengzhan Zhao, Jonathan Eyolfson, Yifan Qiao, Zhihao Jia, Minjia Zhang, Ravi Netravali, and Guoqing Harry Xu. Bamboo: Making preemptible instances resilient for affordable training of large dnns. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 497–513, 2023.

[71] Marcel Wagenländer, Guo Li, Bo Zhao, Luo Mai, and Peter Pietzuch. Tenplex: Dynamic parallelism for deep learning using parallelizable tensor collections. In *Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles*, pages 195–210, 2024.

[72] Borui Wan, Mingji Han, Yiyao Sheng, Yanghua Peng, Haibin Lin, Mofan Zhang, Zhichao Lai, Menghan Yu, Junda Zhang, Zuquan Song, Xin Liu, and Chuan Wu. ByteCheckpoint: A unified checkpointing system for large foundation model development. In *22nd USENIX Symposium on Networked Systems Design and Implementation (NSDI 25)*, pages 559–578, Philadelphia, PA, April 2025. USENIX Association.

[73] Borui Wan, Gaohong Liu, Zuquan Song, Jun Wang, Yun Zhang, Guangming Sheng, Shuguang Wang, Houmin Wei, Chenyuan Wang, Weiqiang Lou, Xi Yang, Mofan Zhang, Kaihua Jiang, Cheng Ren, Xiaoyun Zhi, Menghan Yu, Zhe Nan, Zhuolin Zheng, Baoquan Zhong, Qinlong Wang, Huan Yu, Jinxin Chi, Wang Zhang, Yuhan Li, Zixian Du, Sida Zhao, Yongqiang Zhang, Jingzhe Tang, Zherui Liu, Chuan Wu, Yanghua Peng, Haibin Lin, Wencong Xiao, Xin Liu, and Liang Xiang. Robust llm training infrastructure at bytedance, 2025.

[74] Wandb Team. AI is Easy to Productionize. https://wandb.ai/site/, 2025.

[75] Jiahao Wang, Jinbo Han, Xingda Wei, Sijie Shen, Dingyan Zhang, Chenguang Fang, Rong Chen, Wenyuan Yu, and Haibo Chen. Kvcache cache in the wild: Characterizing and optimizing kvcache cache at a large cloud provider. *arXiv preprint arXiv:2506.02634*, 2025.

[76] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, Denny Zhou, et al. Chain-of-thought prompting elicits reasoning in large language models. *Advances in neural information processing systems*, 35:24824–24837, 2022.

[77] Bo Wu, Sid Wang, Yunhao Tang, Jia Ding, Eryk Helenowski, Liang Tan, Tengyu Xu, Tushar Gowda, Zhengxing Chen, Chen Zhu, Xiaocheng Tang, Yundi Qian, Beibei Zhu, and Rui Hou. LlamaRL: A Distributed Asynchronous Reinforcement Learning Framework for Efficient Large-scale LLM Training, June 2025.

[78] Tianyuan Wu, Wei Wang, Yinghao Yu, Siran Yang, Wenchao Wu, Qinkai Duan, Guodong Yang, Jiamang Wang, Lin Qu, and Liping Zhang. GREYHOUND: hunting fail-slows in hybrid-parallel training at scale. In Deniz Altinbüken and Ryan Stutsman, editors, *Proceedings of the 2025 USENIX Annual Technical Conference, USENIX ATC 2025, Boston, MA, USA, July 7-9, 2025*, pages 731–747. USENIX Association, 2025.

[79] Yongji Wu, Xueshen Liu, Haizhong Zheng, Juncheng Gu, Beidi Chen, Z. Morley Mao, Arvind Krishnamurthy, and Ion Stoica. Rlboost: Harvesting preemptible resources for cost-efficient reinforcement learning on llms. *CoRR*, abs/2510.19225, 2025.

[80] xAI. Grok 4. https://x.ai/news/grok-4, 2025.

[81] Yuxing Xiang, Xue Li, Kun Qian, Wenyuan Yu, Ennan Zhai, and Xin Jin. Servegen: Workload characterization and generation of large language model serving in production. *arXiv preprint arXiv:2505.09999*, 2025.

[82] An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengen Huang, Chenxu Lv, et al. Qwen3 technical report. *arXiv preprint arXiv:2505.09388*, 2025.

[83] Zhilin Yang, Peng Qi, Saizheng Zhang, Yoshua Bengio, William W. Cohen, Ruslan Salakhutdinov, and Christopher D. Manning. Hotpotqa: A dataset for diverse, explainable multi-hop question answering. *CoRR*, abs/1809.09600, 2018.

[84] Zhewei Yao, Reza Yazdani Aminabadi, Olatunji Ruwase, Samyam Rajbhandari, Xiaoxia Wu, Ammar Ahmad Awan, Jeff Rasley, Minjia Zhang, Conglong Li, Connor Holmes, Zhongzhu Zhou, Michael Wyatt, Molly Smith, Lev Kurilenko, Heyang Qin, Masahiro Tanaka, Shuai Che, Shuaiwen Leon Song, and Yuxiong He. Deepspeed-chat: Easy, fast and affordable rlhf training of chatgpt-like models at all scales, 2023.

[85] Qiying Yu, Zheng Zhang, Ruofei Zhu, Yufeng Yuan, Xiaochen Zuo, Yu Yue, Weinan Dai, Tiantian Fan, Gaohong Liu, Lingjun Liu, Xin Liu, Haibin Lin, Zhiqi Lin, Bole Ma, Guangming Sheng, Yuxuan Tong, Chi Zhang, Mofan Zhang, Wang Zhang, Hang Zhu, Jinhua Zhu, Jiaze Chen, Jiangjie Chen, Chengyi Wang, Hongli Yu, Yuxuan Song, Xiangpeng Wei, Hao Zhou, Jingjing Liu, Wei-Ying Ma, Ya-Qin Zhang, Lin Yan, Mu Qiao, Yonghui Wu, and Mingxuan Wang. Dapo: An open-source llm reinforcement learning system at scale, 2025.

[86] Xinyi Zhang, Hanyu Zhao, Wencong Xiao, Xianyan Jia, Fei Xu, Yong Li, Wei Lin, and Fangming Liu. Rubick: Exploiting job reconfigurability for deep learning cluster scheduling. *Proceedings of Machine Learning and Systems*, 7, 2025.

[87] Yanli Zhao, Andrew Gu, Rohan Varma, Liang Luo, Chien-Chin Huang, Min Xu, Less Wright, Hamid Shojanazeri, Myle Ott, Sam Shleifer, et al. Pytorch fsdp: Experiences on scaling fully sharded data parallel. *arXiv preprint arXiv:2304.11277*, 2023.

[88] Lianmin Zheng, Liangsheng Yin, Zhiqiang Xie, Jeff Huang, Chuyue Sun, Cody Hao Yu, Shiyi Cao, Christos Kozyrakis, Ion Stoica, Joseph E. Gonzalez, Clark Barrett, and Ying Sheng. Efficiently programming large language models using sglang. *arXiv preprint arXiv: 2312.07104*, 2023.

[89] Yinmin Zhong, Zili Zhang, Xiaoniu Song, Hanpeng Hu, Chao Jin, Bingyang Wu, Nuo Chen, Yukun Chen, Yu Zhou, Changyi Wan, Hongyu Zhou, Yimin Jiang, Yibo Zhu, and Daxin Jiang. StreamRL: Scalable, Heterogeneous, and Elastic RL for LLMs with Disaggregated Stream Generation, April 2025.

[90] Yinmin Zhong, Zili Zhang, Bingyang Wu, Shengyu Liu, Yukun Chen, Changyi Wan, Hanpeng Hu, Lei Xia, Ranchen Ming, Yibo Zhu, et al. Rlhfuse: Efficient rlhf training for large language models with inter-and intra-stage fusion. *arXiv preprint arXiv:2409.13221*, 2024.

[91] Jiecheng Zhou, Qinghao Hu, Yuyang Jin, Zerui Wang, Peng Sun, Yuzhe Gu, Wenwei Zhang, Mingshu Zhai, Xingcheng Zhang, and Weiming Zhang. RL in the wild: Characterizing RLVR training in LLM deployment. *CoRR*, abs/2509.25279, 2025.

[92] Ruidong Zhu, Ziheng Jiang, Chao Jin, Peng Wu, Cesar A Stuardo, Dongyang Wang, Xinlei Zhang, Huaping Zhou, Haoran Wei, Yang Cheng, et al. Megascale-infer: Serving mixture-of-experts at scale with disaggregated expert parallelism. *arXiv preprint arXiv:2504.02263*, 2025.

[93] Zilin Zhu, Chengxing Xie, Xin Lv, and slime Contributors. slime: An llm post-training framework for rl scaling. https://github.com/THUDM/slime, 2025. GitHub repository. Corresponding author: Xin Lv.

ARTIFACT EVALUATED
USENIX
AVAILABLE

Oxbow: A Coordinated Architecture for Multi-component File Systems

Jongyul Kim[+] Jaehwan Lee[*] Inhoe Koo[*] Peizhe Liu[+]
Jiyuan Zhang[+] Junho Ahn[*] Tianyin Xu[+] Youngjin Kwon[*]

[+]University of Illinois Urbana-Champaign [*]KAIST

Abstract

Fast storage hardware and computational SSDs have outpaced the traditional kernel-centric or kernel-bypass file system designs, fragmenting modern storage stacks across library file systems, kernel subsystems, and in-device file systems. Each architecture offers only a subset of desired properties: user-level designs deliver low latency but lose kernel services and isolation, kernel file systems retain rich functionality but become CPU-bound and slow, and device-resident logic reduces host load but suffers from PCIe latency and wimpy processors.

This paper presents Oxbow, a coordinated storage architecture that composes kernel, user-space, and device components to achieve all four goals simultaneously: high performance, strong kernel interoperability, low CPU consumption, and fast development velocity. Oxbow combines a kernel-based read path with a kernel-bypassing write path, shared-ownership metadata, and Split Journaling, a host–device journaling mechanism that decouples `fsync` from background commits using staging areas and DMA-based snapshots. We demonstrate that Oxbow delivers kernel-level protection and sharing semantics while matching or exceeding the performance of state-of-the-art user-level file systems and substantially reducing host CPU use through offload to computational SSDs.

1 Introduction

The storage landscape is undergoing a fundamental transformation. Storage devices have raced past 14 GB/s [23], turning the traditional kernel-centric architecture—once the bedrock of stability—into a performance bottleneck. To bypass this bottleneck, the industry has diverged into two competing directions. On one side, user-level file systems [36, 39] implement lightweight I/O data paths and eliminate system-call overhead to maximize host throughput. On the other, computational storage devices (CSDs) [15, 22] embed ARM cores to offload tasks entirely from the host, saving host CPU cycles [35, 42]. Yet, despite these advances, modern systems face an uncomfortable truth: no single architecture delivers high performance, efficient resource utilization, and robust kernel integration simultaneously.

The root cause is fragmentation. Storage functionality is now fractured across application libraries, user-space servers, kernel components, and in-device logic. This fragmentation forces systems into rigid trade-offs: User-level file systems achieve low latency but lose the kernel's rich functionality—page cache, readahead, access control, and zero-copy primitives such as `sendfile()`. Device-resident file systems [33, 56, 57] reduce host CPU load but are constrained by PCIe latency and weak device-side processors, making them unsuitable for latency-sensitive or metadata-heavy operations. Each architecture adopts a different subset of functionality, but none can satisfy all system goals at once. The prevailing wisdom dictates a choice: user space for performance, the kernel for integration, or the device for resource efficiency. However, this either-or framing ignores a deeper opportunity.

We present Oxbow, a file system embodying a *coordinated architecture* that harmonizes these fragmented pieces. Oxbow distributes operations across four components: a user-level library (oxLib) that exposes fast, application-aware interfaces, a trusted user-level file server (H-Server) that executes core file-system logic, a thin kernel shim (illuFS) that integrates with VFS, page cache, and kernel-managed file abstractions, and a device-side server (D-Server) that offloads crash-consistency work such as background journaling and checkpointing. The central idea behind Oxbow is to partition responsibilities strategically and minimize shared mutable state, allowing each component to excel at the tasks it is naturally suited for. This coordination enables Oxbow to combine high user-level performance, rich kernel interoperability, and efficient device offloading—without introducing excessive synchronization or communication overhead.

Realizing this architecture raises three coordination problems. *First*, the kernel must be engaged selectively: bypassing it forfeits the page cache, readahead, and zero-copy paths such as `sendfile()`, while routing all I/O through it reintroduces system-call and data-movement overhead. Oxbow applies *semi-kernel-bypassing I/O*, exploiting the kernel's asymmetric value for reads versus writes—reads traverse the kernel page-fault path to reuse the page cache and readahead, whereas writes bypass the kernel and stream from the application through H-Server to the device at user-level speed. *Second*, components must share metadata without the component-crossing synchronization that becomes a bottleneck under contention. Oxbow uses *shared-ownership metadata*, partitioning inode attributes by the component that updates them so that each attribute has a single writer (e.g., oxLib owns `size` and `mtime`, the kernel owns `uid` and `gid`), removing synchronization from the common path. *Third*, crash-consistency work—

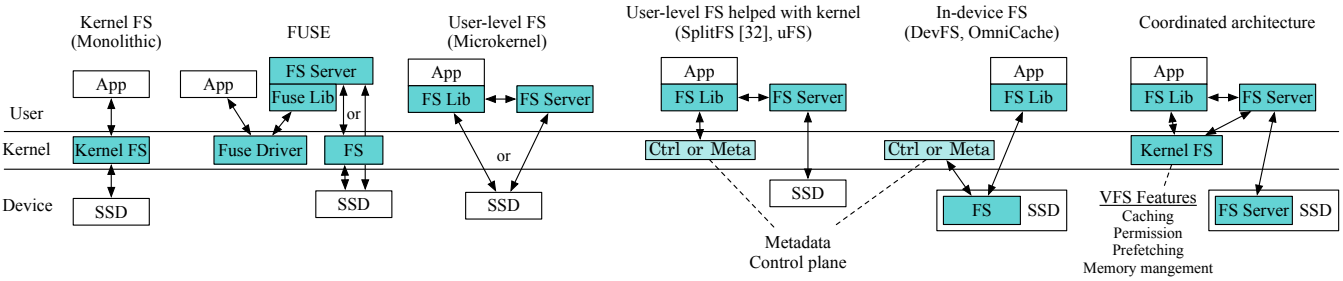


Figure 1: **Comparison of file system architectures.** Shaded boxes denote file-system components.

CPU-intensive I/O that runs in the background, making it a strong offloading candidate—should move to the device without stalling foreground `fsync` on slow device-side commits. Oxbow introduces *split journaling*, which stages `fsync` data on a fast host path while the CSD journals on a separate background path, letting `fsync` complete at host speed while reducing host CPU consumption.

We implement Oxbow with four components totaling 53K lines of code, emulating a CSD using a BlueField-2 DPU [22] and an SR-IOV-capable SSD [13] to enable concurrent host-device access. We evaluate Oxbow against Ext4, µFS [39], and OmniCache [57] using microbenchmarks and real-world workloads. Across microbenchmarks, Oxbow delivers up to 4.8× the write throughput of Ext4 and up to 86% higher throughput than µFS, while reducing host CPU consumption by up to 55% relative to µFS and improving throughput-per-CPU efficiency by up to 3.9× over µFS (4.7× over Ext4). In a LevelDB YCSB benchmark, Oxbow achieves the best throughput across read- and range-heavy workloads, improving throughput by up to 89% over µFS and 41% over Ext4. By retaining the kernel's VFS layer, Oxbow can transparently leverage `sendfile`, achieving 3.3× higher Nginx throughput than the same configuration without it.

We have made Oxbow publicly available at `https://github.com/xlab-uiuc/oxbow`.

2 Trends in Modern Storage Systems

2.1 The Fragmented Landscape

Modern file systems span a widening spectrum of architectures, each making fundamentally different design tradeoffs as illustrated in Figure 1. Monolithic kernel file systems [1, 25, 43] integrate tightly with VFS, page cache, and memory management, offering robust protection and strong sharing semantics. However, their reliance on the complex kernel I/O stack incurs kernel-crossing costs and software overheads. FUSE-based systems preserve kernel services while enabling user-space development, but frequent boundary crossings and data copies constrain them to non–performance-critical workloads. User-level file systems bypass the kernel entirely—either as application-linked libraries (e.g., Exokernel [29]) or user-space servers in microkernel designs [38], achieving impressive throughput by eschewing kernel mediation. Yet this performance comes at the cost of protection, global coordination, and transparent sharing. Even modern hybrids such as µFS [39] must combine user-level and kernel mechanisms in ad hoc ways that complicate correctness and maintenance. Finally, in-device file systems [33, 47, 56, 57] execute directly on CSDs, reducing host CPU usage and benefiting from near-data processing. Their shortcomings lie elsewhere: limited device compute, PCIe round trips, and difficulty integrating with host-side abstractions.

The result is a fragmented landscape: each architecture excels along one axis but none strikes a balanced design.

2.2 The Promise and Cost of User-level I/O

Fast storage hardware [2, 10, 11] has amplified the relative cost of kernel I/O paths and mode switches. To remove these bottlenecks, many modern systems adopt user-level drivers such as DPDK [3] and SPDK [7], which bypass kernel mediation entirely [36, 39, 42, 46, 52].

2.2.1 Promise

User-level I/O offers three advantages: (1) a fast, predictable data path, (2) scalable resource allocation by simply adjusting user-level thread counts [39], and (3) rapid development with rich user-space tooling. These benefits allow developers to quickly adapt file systems to emerging hardware such as CSDs, Compute Express Link (CXL) SSDs [49], Zoned Namespace (ZNS) SSDs [14], and Flexible Data Placement (FDP) SSDs [16].

2.2.2 Cost of Abandoning the Kernel

Bypassing the kernel, however, forfeits its mature, well-engineered services and system-wide management, forcing user-level designs to re-engineer kernel functionality in user space or forgo it entirely.

Isolation. Without kernel scheduling or memory reclamation, user-level file systems often require dedicated resources or entire machines to avoid interference. Many file systems partially delegate resource control back to the kernel [4, 33, 39, 47], complicating the architecture.

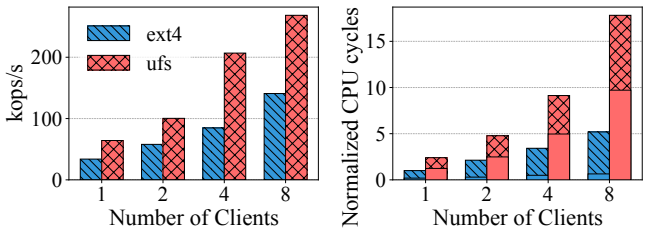


Figure 2: **Filebench (Varmail) throughput and CPU cycles.** CPU cycles are normalized to `Ext4` with one client; solid portions denote cycles in the file system layer. Both systems perform metadata journaling on the testbed (§6.1).

Performance. The kernel page cache, inode and dentry caches, prefetching, and memory management integration accelerate the I/O path. User-level systems that bypass the VFS cannot exploit these services and must build their own in user space—duplicating decades of kernel engineering effort and adding significant development overhead. Some user-level systems even lack essential features such as prefetching [39].

File Sharing. User-level implementations cannot leverage kernel page tables or VFS permission checks, so they must reimplement sharing and serialization in user space to enforce correctness. To emulate kernel-level sharing, systems such as Strata [36], µFS [39], and OmniCache [57] rely on leases [31]. However, lease revocation and renewal incur significant overhead under concurrent access [39, 41], and maintaining consistency in user space raises development costs.

Kernel Interoperability. Kernel interoperability enables file systems to support kernel-managed services transparently. For example, zero-copy system calls such as `splice` and `sendfile` move data directly between kernel buffers and are critical for high-performance web servers and file servers. If a file descriptor belongs to a user-level file system that does not interoperate with the kernel, data must traverse user space, incurring extra copies and mode switches.

2.3 The CPU Bottleneck and Device Offload

As storage speeds increase (e.g., 63 GB/s with PCIe 5.0 [12]), the bottleneck shifts from I/O to host CPU. Two trends exacerbate this: (1) Polling-based I/O—SPDK busy-waits for low latency, consuming 100% of a CPU core. (2) Massive parallelism—systems like µFS [39] and optimized Ext4 [26, 34] scale throughput but linearly increase CPU usage. Figure 2, with a mail server workload, shows the trade-off: µFS achieves 2.4× Ext4 throughput but consumes 3.4× CPU cycles, with over 50% spent in the file system layer. CPU-heavy operations such as journaling, checksums, and encryption steal cycles from applications.

Device Offloading. CSDs [15, 19, 22] now include ARM cores, DRAM, and accelerators. While prior work uses CSDs for I/O scheduling [33, 47] or near-data processing [54, 56, 57], we argue that they suit *background tasks* more than latency-critical data paths. CSD processors are weaker than host CPUs (e.g., 2.0 GHz ARM vs. 4.0 GHz x86), and PCIe crossings add latency; on the other hand, they excel at asynchronous, parallel processing. Offloading journaling, garbage collection, or compression to the device reduces host CPU load without penalizing foreground operations.

2.4 Requirements vs. Current Designs

These observations expose major tensions in modern file systems in the following dimensions:

- **High performance:** scalable, low-overhead I/O paths.
- **Kernel interoperability:** leveraging mature VFS services and interfaces.
- **Low CPU consumption:** minimizing interference with co-located workloads by device offloading.
- **Fast development velocity:** enabling rapid evolution in user space.

Existing architectures typically center on one primary placement: user-level designs place functionality in user space for performance and agility, kernel file systems keep it in the kernel for integration, and device-resident designs move it to the device to reduce host CPU load. Each choice satisfies a different subset of requirements, leaving open how to compose their strengths without inheriting their coordination costs.

3 Coordinated Architecture

Our key insight is that user-space, kernel, and device components are not adversaries to be chosen between, but complementary resources that can be orchestrated. We therefore propose a *coordinated architecture* that deliberately distributes file system responsibilities across components, assigning each task to the component best suited to perform it.

At a high level, the kernel remains responsible for caching, protection, and sharing; user space hosts flexible file-system logic and fast I/O; and the device handles CPU-intensive background tasks that would otherwise interfere with application performance. Crucially, unlike prior device-centric systems (e.g., DevFS [33], CrossFS [47], and OmniCache [57]) that migrate substantial foreground logic into the device, the coordinated architecture retains foreground file-system operations on the host to avoid PCIe round trips and preserve low latency.

Placement across User Space and Kernel. A key principle of the coordinated architecture is to place *dynamic, fast-evolving* features in user space, while reusing *stable, heavily optimized* functionality in the kernel. Kernel services such as the page cache, readahead, write-back, protection, and permission checking have decades of engineering investment and evolve slowly relative to hardware trends. Reusing these services avoids reimplementation, reduces complexity, and ensures transparent interoperability with other applications.

User space, in contrast, offers rapid iteration and flexible mapping to fast I/O paths, enabling adaptation to emerging device interfaces and application demands.

Composing multiple components into a coherent file system introduces several fundamental challenges beyond the limitations of any single component:

- **Balancing kernel involvement.** Fully bypassing the kernel forfeits its services, while routing all I/O through it reintroduces kernel-crossing overheads. The architecture must decide which operations genuinely benefit from kernel involvement.
- **Metadata synchronization.** If multiple components update file system metadata, every access risks cross-component synchronization. For example, both the kernel (through the VFS and memory subsystem) and the user-level file server may need to access or mutate inode state. Naively granting either component full ownership forces costly synchronization—e.g., fetching the latest version from the other component, invalidating cached state, or acquiring shared locks or leases that quickly become contended and add complexity.
- **Cross-component data movement.** Distributing functionality across user space, the kernel, and the device can amplify communication overheads. Data or metadata may traverse multiple boundaries: IPC between the application and user server, kernel crossings, PCIe round trips to a device, and memory copies within the host. Without careful structuring, these costs can overshadow the benefits of specialized components.
- **Crash-consistency dependencies.** Offloading journaling to a CSD is appealing because background journaling consumes substantial host CPU cycles in user-level I/O stacks, but traditional journaling schemes couple foreground `fsync` operations with background commit work. In-place data updates and POSIX ordering constraints can force the fast host CPU to wait for slower device-side commits, while the system still must preserve ordering and recovery semantics across host and device state.

To address these challenges, the coordinated architecture adopts three techniques that selectively place operations on the kernel path, minimize synchronization, reduce data movement, and restrict device offload to background work:

- **Semi-kernel-bypassing I/O.** The coordinated architecture exploits the asymmetric benefits of kernel involvement. Operations that depend on kernel services or state (e.g., page faults, eviction decisions, access-control checks) traverse the kernel, while others bypass the kernel to achieve user-level latency and throughput. In particular, reads benefit from kernel caching and readahead, whereas writes benefit from bypassing the kernel on the persistence path.
- **Shared-ownership metadata.** Instead of assigning an inode or metadata structure exclusively to a single component, the coordinated architecture partitions metadata fields and assigns each partition to the component best suited to maintain it. This single-writer structure avoids coarse-grained invalidation or locking while still allowing multiple components to read shared metadata.
- **File-semantics-agnostic device offloading.** The coordinated architecture decouples crash consistency from POSIX file semantics, allowing journaling and other maintenance tasks to run entirely on the device without host–device coordination on the foreground path. Because device CPUs are slower and PCIe adds latency, offloaded work is restricted to background operations (e.g., journaling, garbage collection, and compression) that tolerate latency but benefit from parallel, near-data execution.

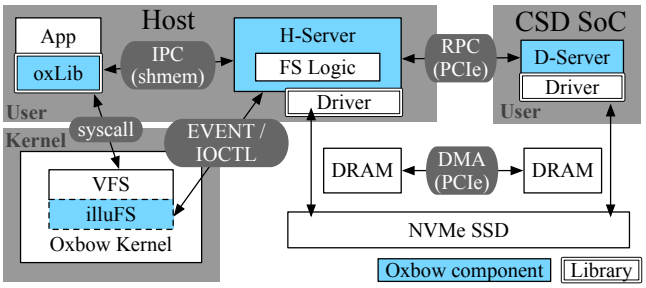


Figure 3: **Oxbow components and their communication.**

4 Oxbow File System

Oxbow instantiates the coordinated architecture of §3 with four components (Figure 3): a per-application user-level library (oxLib) and a trusted user-level file server (H-Server) on the fast user-space I/O path, a thin in-kernel shim (illuFS) that reuses kernel VFS services, and a device-resident file server (D-Server) that runs background work on a CSD.

The three techniques of §3 map onto these components as follows. *Semi-kernel-bypassing I/O* is realized by the oxLib/illuFS/H-Server split and its read/write paths (§4.1, §4.3, and §4.6); *shared-ownership metadata* partitions inode fields across oxLib, Oxbow Kernel, and H-Server (§4.2); and *file-semantics-agnostic device offloading* is embodied by Split Journaling, which keeps `fsync` on the host while D-Server performs background journaling (§4.4 and §4.5).

4.1 Components and Overall Architecture

Figure 3 shows the four Oxbow components and communication channels that realize semi-kernel-bypassing I/O and device-side journaling.

oxLib. oxLib is a user-space library linked into each application. It exposes the POSIX API, intercepts file system calls, and replaces read and write system calls with `mmap`-based loads and stores. For operations that require durable persistence or coordination, such as `fsync` and namespace operations (e.g., file and directory creation), oxLib forwards

requests to H-Server. oxLib also maintains per-file dirty-page and page-lock bitmaps in shared memory (§4.2), which allow H-Server to identify dirty pages and enforce page-granularity atomicity without involving the kernel in block management.

Oxbow Kernel and illuFS. To leverage the kernel's VFS services, Oxbow adds a lightweight in-kernel file system, illuFS. From the VFS's perspective, illuFS behaves like a normal kernel file system: it participates in mount/unmount, cooperates with the page cache, and integrates with readahead and eviction. Instead of implementing its own file system layer, however, illuFS forwards I/O and metadata requests to H-Server via an event-driven interface. Oxbow therefore reuses the kernel's page cache, permission checks, and sharing semantics without modifying kernel subsystems or routing data through the kernel block layer.

H-Server. H-Server is a trusted user-level file server that owns the storage device. It implements the core file system logic, including on-disk layout, block and inode allocation, indexing, and cooperation with journaling. Conceptually, H-Server plays the role of a generic "VFS for user-space file systems": it offers a pluggable interface for file-system-specific logic (such as Ext4-style layouts) while providing generic mechanisms for staging, journaling, and communication with oxLib and illuFS. All device I/O from the host passes through H-Server and is issued via a user-level driver, providing a fast data path and enabling kernel bypass for writes. H-Server also prepares transactions for Split Journaling, stages `fsync` data, and coordinates background journaling with D-Server (§4.4).

D-Server. D-Server runs on the CSD and executes background journaling and checkpointing. It receives prepared transactions from H-Server, pulls their contents into device DRAM via DMA, and persists them to an on-disk journal area. By running on the CSD, D-Server offloads the CPU and I/O work of background journaling from the host while exploiting device-level parallelism. D-Server implements data journaling and checkpointing, but it does not understand file semantics; instead, it operates purely on block addresses and extents provided by H-Server (§4.4).

Communication Paths. Oxbow adds explicit communication channels between components beyond system calls. The kernel communicates with H-Server using an event-driven mechanism based on `epoll`: illuFS enqueues I/O and metadata requests for H-Server, and H-Server responds via `ioctl` once operations complete. oxLib and H-Server share metadata and control information through shared memory and lightweight IPC, avoiding copies and unnecessary mode switches. H-Server and D-Server communicate over PCIe using an RPC mechanism for control (e.g., "transaction ready") and DMA for bulk data transfer. When H-Server prepares a transaction, it copies metadata and dirty pages into a contiguous DMA buffer and notifies D-Server. D-Server then pulls this DMA buffer into device memory using DMA, which does not consume host CPU cycles. These channels make coordination explicit while keeping latency-sensitive communication on the host and restricting PCIe traffic to background paths.

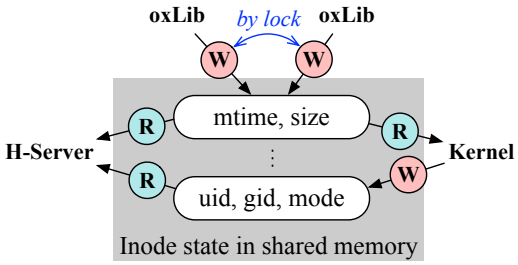


Figure 4: **Shared-ownership of states.** R: read, W: update.

4.2 State Ownership and Sharing

Oxbow adopts a shared-nothing design for most metadata, and shared-ownership only where components must cooperate. On disk, H-Server hosts arbitrary file system logic, using the underlying layout (e.g., Ext4 superblock, inode table, and allocation bitmaps); Oxbow adds its own structures for a dedicated journal area and a staging area (§4.3), each with a superblock. This design decouples crash consistency from base file system semantics and allows Split Journaling to operate independently.

In memory, each metadata object has a primary owner: H-Server manages global file system state such as free block and inode bitmaps and the inode cache as part of its core logic, while D-Server exclusively owns crash-consistency metadata.

Shared-ownership of Metadata. For metadata accessed by multiple host components, Oxbow uses *shared-ownership*. Inode state is split into small attribute groups with a single writer per group (Figure 4): permission fields such as `uid/gid` are updated only by the kernel, while file size and timestamps are updated exclusively by oxLib. H-Server reads these fields when updating on-disk metadata but does not modify them after initial load. All shared attributes are stored in shared memory so the kernel, oxLib, and H-Server can access them without copying. This single-writer, multi-reader pattern avoids global locks for synchronization in the I/O path, with locks needed only when multiple oxLib instances concurrently update the same inode.

Ownership is enforced rather than advisory: Oxbow treats the kernel and H-Server as trusted and oxLib as untrusted, and H-Server exposes each attribute group as a separate POSIX shared-memory region whose permissions derive from the file and are enforced by the kernel. Attributes that oxLib does not own (e.g., `uid/gid`) are mapped read-only, so a compromised oxLib can modify only the fields it owns (e.g., size and `mtime`), and only for files it may already write; it cannot read or corrupt metadata of files it is not authorized to access. H-Server also re-validates owned fields on use, so fabricated values stay contained—an inflated size, for instance, fails at block resolution instead of exposing unallocated blocks.

Dirty and Page-lock Bitmaps. File data is shared by all

host components. Applications write to `mmap`-ed pages via oxLib, the kernel manages those pages in the page cache, and H-Server writes them back to disk. Oxbow avoids data copies by sharing page-cache memory across components. Per-file dirty-page and page-lock bitmaps, maintained by oxLib in shared memory, let H-Server identify dirty data precisely without exposing page-to-block mappings to the kernel. These bitmaps are accessed only by oxLib and H-Server, not by the kernel: Oxbow delegates dirty-page tracking and page locking—tasks traditionally handled by the kernel as part of page-cache management—to user-level components, avoiding the mode switches and communication overhead that kernel-managed bookkeeping would otherwise incur.

When an application modifies a page, oxLib sets the corresponding dirty bit. H-Server consumes these bits when preparing `fsync` staging or background journaling, and clears each bit only after the page's contents have been copied into the DMA buffer. Page-lock bits provide page-granularity synchronization by guaranteeing exclusive access to a page-cache page, so different pages of the same file can be updated concurrently. A lock is held in two cases: when oxLib writes to a page-cache page, and when H-Server copies a page from the page cache into the DMA buffer. Because both writers acquire the same lock, an application write and H-Server's copy never race on the same page, preserving the kernel's page-lock semantics in user space.

In-memory Caches. H-Server maintains the authoritative in-memory inode cache and on-disk allocation structures, in a form compatible with `struct inode`. Selected attributes are shared with the kernel and oxLib via the shared-ownership mechanism, while the kernel's VFS layer continues to serve reads via the page cache. H-Server caches directory entries so namespace operations can avoid kernel involvement. Oxbow does not rely on the kernel's native VFS inode cache for on-disk updates. On the device, D-Server keeps journal metadata, including its superblock and commit/checkpointing state. Journal transactions store only block addresses, extents, and staging descriptors, keeping D-Server file-system-agnostic and reinforcing the separation between crash consistency and file system logic.

4.3 End-to-End Operation Flow

Open Path. On `open`, oxLib calls into the kernel to obtain a file descriptor. illuFS forwards the lookup to H-Server, which performs path resolution, allocates an inode number, and initializes per-file structures in both H-Server and shared memory. The kernel returns to oxLib once the inode number is known, while H-Server continues initialization in parallel. oxLib then maps the file for direct load/store access via `mmap` and the shared-memory region.

Read Path—Through the Kernel Page Cache. As shown in Figure 5, oxLib translates a `read` into loads from the `mmap`-ed region. A first access causes a page fault; the kernel allocates a

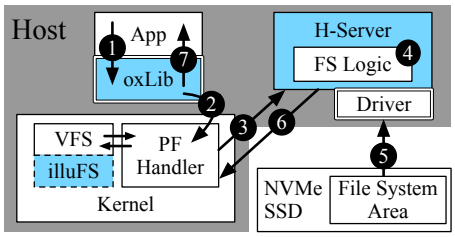


Figure 5: **The read path.** Numbers denote the order of steps.

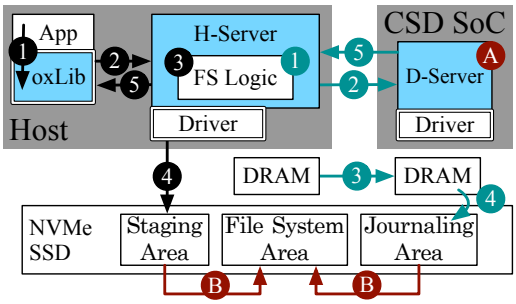


Figure 6: **The writeback path.** (X: `fsync`, X: background commit, X: checkpointing)

page and invokes illuFS, which issues a user-level read request to H-Server. H-Server computes block numbers using its file-system logic, submits I/O through its user-level driver, and fills the page cache upon completion. The kernel then resolves the page fault and resumes the application. Subsequent reads hit in the page cache. Because Oxbow uses the VFS layer, it also benefits from kernel readahead: asynchronous read-ahead requests are issued by the kernel and served by H-Server the same way, reducing read latency for sequential workloads.

Write Path and `fsync`—Bypassing the Kernel. Applications write by storing into `mmap`-ed pages (Figure 6, ❶). If the page is present, the store updates it directly; otherwise, a page fault loads it as in the read path. oxLib marks the page dirty and sets its page-lock bit while the page is being modified.

On `fsync`, oxLib forwards the request to H-Server (❷), which persists the file durably on the host using Split Journaling (§4.4). H-Server snapshots the file's metadata, resolves its dirty pages to block addresses (allocating blocks if needed) (❸), and writes the file's data and inode into a dedicated *staging area*, a persistent on-disk region (❹). `fsync` returns as soon as this staging write is durable (❺); it does *not* wait for the device-side background journaling. Background journaling then proceeds asynchronously on D-Server, batching many files into the journal area without blocking the application. §4.4 details how Split Journaling decouples these two paths so that `fsync` need not wait.

4.4 Split Journaling: Host–Device Journaling

Conventional journaling, as in Ext4 and other journaling file systems, persists updates in two phases: applications' data and metadata modifications are grouped into transactions and

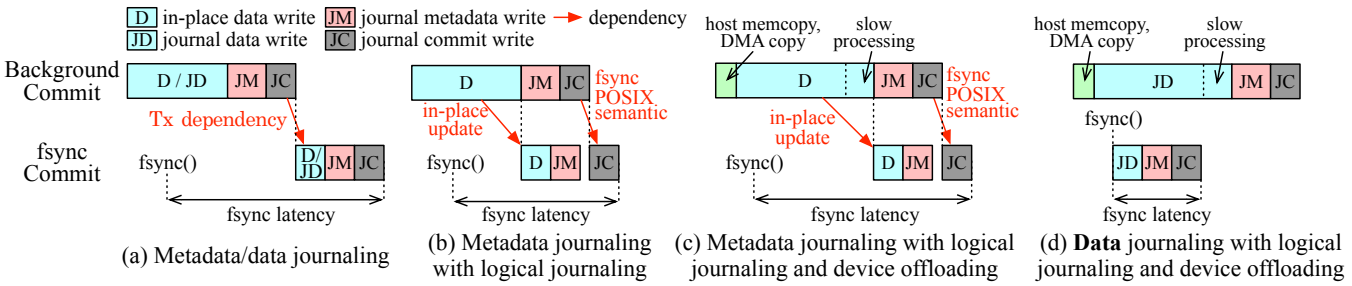


Figure 7: **Dependencies in different journaling schemes.** Host memcopy indicates copying dirty pages to a DMA buffer within the host. DMA copy represents data copy from the host DMA buffer to a device memory. The only dependency in (d) is host memcopy—`fsync` must wait for memcopy from the page cache to the DMA buffer.

written to a journal area, where a commit record marks each transaction durable; a later checkpointing phase applies those changes to the main file system area and frees journal space. Metadata journaling writes only metadata to the journal and updates data in place, reducing write amplification but weakening data consistency, while data journaling logs both data and metadata, providing strong crash consistency at the cost of more I/O. By utilizing a CSD, Oxbow offloads data journaling costs to D-Server.

Split Journaling keeps `fsync` latency on the host while offloading background commits to the device. Achieving this requires removing the dependencies that tie foreground `fsync` to background journaling in traditional designs. In conventional metadata journaling, foreground `fsync` and background commits share a single journal and are committed in order, so `fsync` is serialized behind outstanding background commits (Figure 7a). Logical journaling techniques [45, 48] break this coupling by separating `fsync`-induced transactions from background commits. Doing so, however, leaves two finer dependencies behind, each of which still introduces a delay (Figure 7b). First, `fsync`'s in-place write must wait for the completion of any in-progress background in-place update to the same file (*in-place update dependency*). Second, `fsync` must wait for all prior transactions modifying the file to be committed (*POSIX ordering dependency*).

Device offload exacerbates these residual dependencies: checking them requires frequent host–device coordination, and any slowdown in background commits directly impacts `fsync` (Figure 7c). Split Journaling eliminates both dependencies by using data journaling for its two paths and introducing a staging area that decouples `fsync` from background commits (Figure 7d).

Decoupling Staging from Background Commits. Split Journaling provides two data-journaling paths that target two on-disk areas: a fast *staging path* driven by the host for `fsync`, and a slower *background path* driven by D-Server. The background path runs as periodic *background commits*—triggered automatically by the file system—that batch updates for many files into a single journal transaction. On `fsync`, H-Server writes the file's data and inode metadata directly into the staging area through the DMA buffer of its user-level driver, forming a *staging transaction* that is independent of background commit progress. For background commits, H-Server instead fills a separate host-memory DMA buffer (Figure 6, ❶) and signals D-Server (❷), which pulls the data into device memory (❸) and writes it out to the journal area (❹). Because staged data is logged rather than updated in place, the in-place update dependency disappears.

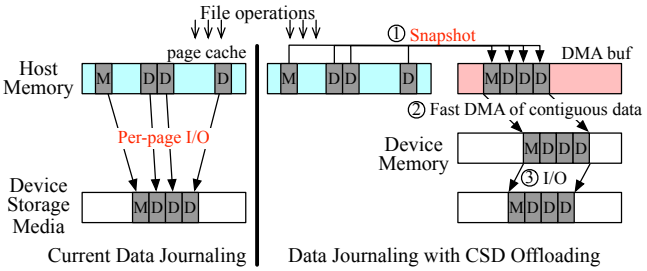


Figure 8: **DMA buffer for shadow copy.** A write to a page is blocked while it is being persisted (left) vs. only while it is copied into the DMA buffer (right). D: data, M: metadata.

To eliminate POSIX ordering dependencies, staging transactions are self-contained. If a file is part of any in-flight journal transaction, H-Server re-logs the data and metadata relevant to that file into the staging area along with new `fsync` data. Each staging transaction also records the ID of the latest committed journal transaction for recovery (§4.5). As a result, `fsync` no longer waits for background commits, and host and device can operate in parallel (Figure 7d).

DMA Snapshots as Shadow Copies. Offloading to CSD requires transferring data over PCIe, and issuing per-page DMA operations is too costly. Split Journaling copies all pages of a transaction into a contiguous DMA buffer and performs a single DMA transfer. This extra copy is repurposed into a snapshot: the DMA buffer becomes a shadow copy of the page cache (Figure 8).

The snapshot copying brings two benefits. First, it removes interference between page updates and background commits. H-Server briefly locks pages, copies them into the DMA buffer, and immediately unlocks them, allowing subsequent

writes to proceed while the device processes the snapshot. Existing journaling (e.g., JBD2 of Ext4) also relies on shadow copies (frozen buffers) for this, but creates them only lazily: once a page's journal I/O is in flight, a conflicting write must block until that I/O completes [50]. Second, it raises the granularity at which the global lock is acquired from pages to files when adding metadata to a transaction. In traditional journaling, each page inserted into the running transaction—whether data or metadata—must acquire a single global lock. Since data journaling journals data as well, it performs far more insertions and contends for this lock heavily [50]. In Oxbow, snapshots reduce the blocking time to the duration of the memory copy, making metadata insertion with file-granularity locks practical at lower contention. Note that data pages remain synchronized using per-page locks (§4.2).

Parallel and Aggressive Journaling. Background commits exploit both pipeline and thread-level parallelism. The commit pipeline consists of: (1) copying metadata and dirty data into the DMA buffer in H-Server, (2) DMA transfer to D-Server, and (3) device-side journal writes. Once the DMA buffer is handed off, H-Server begins preparing the next transaction.

Oxbow also triggers background commits more aggressively than kernel journaling. Shorter intervals reduce the amount of data written during `fsync`, lowering foreground latency. Because background commits run on CSD CPUs and rely on DMA for transfer, this aggressiveness reduces host CPU consumption and improves perceived `fsync` cost. D-Server further parallelizes journal I/O across its cores, maximizing device bandwidth.

Checkpointing. D-Server reclaims journal and staging space by checkpointing committed transactions (Figure 6, Ⓐ) into the main file-system area when either the journal or the staging area runs low on free space (Ⓑ). Because a journal transaction commits after the staging transactions it records, D-Server checkpoints those staging transactions first and the journal transaction last, preserving commit order.

4.5 Crash Consistency

Oxbow's crash consistency reduces to a single invariant: every `fsync`-ed file is recoverable from its self-contained staging transaction together with the journal prefix whose ID the transaction records. The transaction contents below maintain this invariant, which recovery relies on directly.

Background Journaling. Journal transactions contain snapshots of file-system state (superblock fields, allocation bitmaps), per-file metadata, and data. They are indexed purely by block addresses and extents so that D-Server can process commits without interpreting file semantics. Each journal transaction also records, for every staging transaction committed since the previous journal commit, the address of its *staging descriptor* (the staging transaction's on-disk header). This enables D-Server to reconcile staging and journal state during checkpointing and recovery.

Staging for `fsync`. A staging transaction contains only the inode and data for the `fsync`-ed file, plus the ID of the most recently committed journal transaction. To guarantee recoverability, Oxbow re-logs all uncommitted data for that file into the staging transaction, even if a background commit is in progress. As a result, each staging transaction is self-contained: after a crash, the file can be reconstructed solely from its staging transaction and the journal prefix up to the recorded ID.

Recovery. On initialization, D-Server checks whether the journal is clean. If not, it replays all committed but uncheckpointed journal transactions in order, applying any staging transactions referenced by them. It then scans for "dangling" staging transactions whose associated journal commit was lost or never issued. Because a staging transaction captures all modifications since the last committed journal transaction, D-Server can safely apply these remaining staging transactions on top of the replayed journal state, ensuring durability for all `fsync`-ed files. Since H-Server allocates blocks and inodes after snapshotting the free bitmaps, snapshots lag behind actual allocations. During recovery, D-Server scans all transactions and reconstructs free block and inode bitmaps and superblock counters based on the extents stored in transactions, avoiding reliance on stale bitmap images.

Partial Failures. Oxbow assumes fail-stop components. Durability is established at `fsync` via a self-contained staging transaction persisted to the on-disk staging area, so a host-side failure (oxLib, illuFS, or H-Server) loses only data not yet `fsync`-ed, exactly as under POSIX, while `fsync`-ed files remain durable in the staging area. Such failures need no recovery: since H-Server is a client of D-Server and the device retains this durable state, H-Server can simply be relaunched and reconnect without restarting D-Server. A D-Server failure, by contrast, invalidates H-Server's session, so both are relaunched and D-Server runs the recovery procedure on restart. If the CSD's compute capability is lost (e.g., due to a hardware failure) while its storage remains reachable as a block device, D-Server can run on the host without losing availability.

4.6 File Sharing and Protection

Oxbow preserves the kernel's protection and sharing model by relying on the page cache and permission checks in the VFS layer. When a file is opened, the kernel enforces discretionary access control (DAC) and refuses to `open` or `mmap` the file for unauthorized processes. Once a page is in the page cache, the kernel maps it into the address spaces of all processes that have opened the file, enabling efficient sharing without additional mechanisms in user space. This avoids the lease-based file sharing and custom coherence protocols that user-level file systems must implement [24, 36, 39, 57].

The only additional mechanism Oxbow requires for safe sharing is the arbitration of concurrent writes. The page-lock bitmap maintained by oxLib substitutes for the kernel's in-

ternal page locks: oxLib and H-Server consult the bitmap to avoid concurrent updates to the same page. Because the bitmap lives in shared memory, all oxLib instances share a consistent view of page ownership.

Device access is restricted to H-Server and D-Server. No other process can issue I/O to the storage device directly, which prevents malicious applications from bypassing Oxbow 's protection mechanisms. Shared-memory regions such as dirty and page-lock bitmaps inherit the file's permissions, so an application that modifies oxLib code can corrupt only files that it already has permission to write.

5 Implementation

We use lwext4 [5] as H-Server's file system logic with minor modifications for integration. lwext4 originally targets embedded systems, providing the essential Ext4 logic and on-disk structures such as extents and block groups. We add multi-threading support and disable its built-in journaling because Oxbow provides crash consistency through Split Journaling. Oxbow is implemented in C/C++ totaling 53K lines of code excluding lwext4.

We emulate a CSD using a SmartNIC and an SSD supporting SR-IOV and NVMe Namespace Sharing [18]. Emulation requires (1) an in-device SoC and (2) the ability for the device and host to access the same storage media concurrently. For (1), we use an NVIDIA BlueField-2 DPU [17] which provides 8 ARM cores and 16 GB DRAM within the PCIe power budget, making it a realistic proxy for a CSD processor. For (2), the SmartNIC must access the host SSD. Because BlueField-2 communicates with the host only via RDMA, we expose the SSD to the SmartNIC using NVMe-oF. RDMA transfers place data directly into host DRAM and flush it to the SSD without host CPU involvement, closely matching the host-CPU savings of true CSD offload.

SR-IOV and Namespace Sharing allow both host and SmartNIC to operate on the same SSD namespace. The SSD exposes multiple NVMe controllers—e.g., a Physical Function (PF) for the SmartNIC (D-Server) and a Virtual Function (VF) for the host (H-Server)—each appearing as an independent PCIe device. Oxbow's coordination prevents H-Server and D-Server from accessing the same block concurrently.

6 Evaluation

We evaluate Oxbow in terms of performance, host CPU consumption, and interoperability with the kernel's subsystems. We compare Oxbow with µFS [39], OmniCache [57], and Ext4 [43]. µFS is the state-of-the-art user-level file system designed for high scalability on SSDs. We evaluate the coordinated architecture by comparing Oxbow with OmniCache. Unlike Oxbow, OmniCache operates a whole file system on CSD and selectively caches data between host and device memory (Figure 1). Ext4 is the most widely used journaling file system in the Linux kernel.

Our evaluation proceeds as follows. We first use latency and throughput microbenchmarks to characterize basic file system operations under append, sequential, and random patterns, sweeping the I/O size (§6.2.1) and the number of concurrent clients (§6.2.2). We then run an ablation study to isolate each design choice (§6.3), followed by real-world workloads—LevelDB under YCSB, RAG retrieval, LLM checkpointing, and Nginx with `sendfile`—to assess end-to-end benefits (§6.4). We additionally evaluate Oxbow with metadata-intensive workloads in the appendix (Appendix B).

6.1 Experimental Setup

Testbed. We run all the experiments on a dual-socket server equipped with two Intel(R) Xeon(R) Gold 5218 CPUs @ 2.30GHz, 128GB of DRAM, and a 3.2 TB Samsung PM1735 NVMe SSD [6] that supports SR-IOV and NVMe Namespace Sharing. We emulate a CSD with this device as described in §5. For the emulation, we use an NVIDIA BlueField-2 DPU [17] which is equipped with eight 2.0 GHz ARMv8 A72 cores and 16 GB of DRAM, and 64 GB eMMC for the storage. It supports 100 Gbps Ethernet; hence, PCIe bandwidth is not a bottleneck.

System Configuration. As we implement illuFS in the kernel version 6.2, we use the Ext4 data journaling mode in the same kernel. We disable the lazy initializations of Ext4 to get consistent and stable performance after mounting. We configure a sufficiently large journal (39 GB) and adjust the checkpointing threshold to avoid Ext4 performance degradation caused by journal space exhaustion.

µFS performs user-level I/O using SPDK and guarantees crash consistency with its own metadata journaling. µFS has two modes: a POSIX mode that is transparent to applications and a mode with a custom memory allocator that avoids memory copies between applications and the user-level file server. We use the latter mode to represent the best performance of µFS. We set the number of µFS server threads equal to the number of clients, since we observed having fewer server threads degrades performance [39]. We scale the number of clients up to 10, which is the maximum number µFS supports.

The OmniCache artifact runs only with persistent memory instead of SSDs and emulates CSD using a kernel module. We extend it to operate with a real CSD. The custom NVMe commands are replaced with RPC calls as in Oxbow. We run Ext4 on CSD as OmniCache's NearStorageFS as described in its paper [57]. Here, Ext4 runs in metadata journaling mode to enable direct I/O because OmniCache bypasses the kernel page cache and manages its own user-level cache.

In H-Server, we pin each I/O worker thread to an SSD I/O queue. Unless otherwise stated, H-Server is configured with eight I/O worker threads. Similar to Ext4, we provision Oxbow with sufficient journal space to avoid journal-space exhaustion during the experiments.

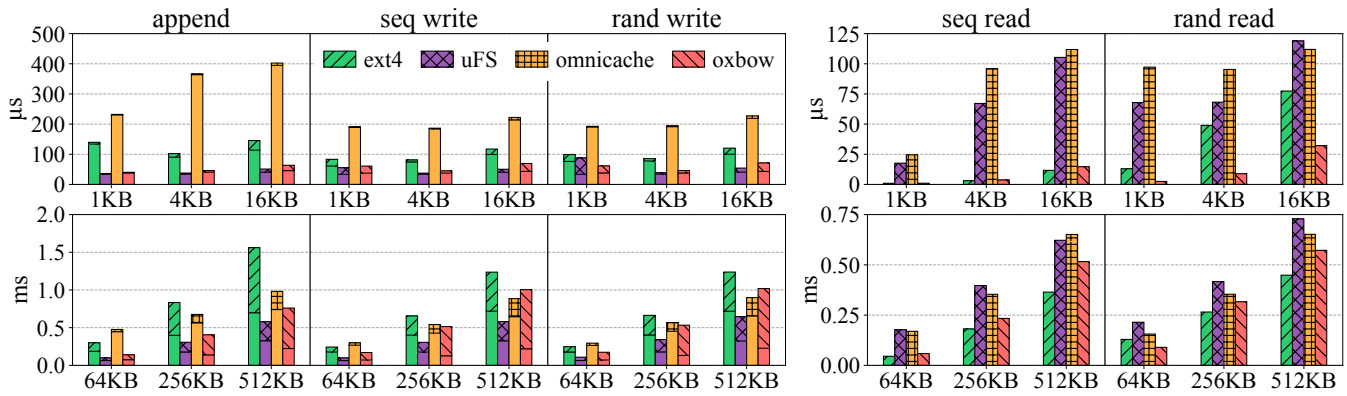


Figure 9: **Latency microbenchmark.** Average per-operation latency of five workload patterns across I/O sizes. The top row shows small I/Os and the bottom row shows large I/Os. The solid portion of each bar denotes the `fsync` latency. Lower is better.

6.2 Microbenchmarks

To evaluate basic file system operations of Oxbow we use the microbenchmark from the µFS paper [39].

6.2.1 Latency

We evaluate the performance of foreground I/O operations using the latency benchmark. A total of 1 GB of data is written to or read from a file with different I/O sizes, and the latency of individual file operations is measured. In particular, in write benchmarks, each `write` call is followed by a `fsync`, and the total latency is measured. The bars in Figure 9 represent the average total latency, and the solid portion of each bar represents the average `fsync` latency.

Write Latency. The two user-level file systems, µFS and Oxbow, achieve lower latency than Ext4 because they have fast foreground paths. Ext4's latency suffers due to its heavier kernel software stack and transaction commit mechanism. Compared with Ext4, Oxbow achieves 2.1–3.5×, 1.2–1.8×, and 1.2–1.9× lower latency for append, sequential, and random writes, respectively. However, µFS achieves up to 43% lower latency than Oxbow because Oxbow performs two additional memory copies: from the application buffer to the kernel page cache during a `write`, and from the page cache to an SPDK buffer during `fsync`. In contrast, µFS exposes SPDK buffers to applications to achieve zero-copy I/O, sacrificing application transparency. This overhead is the cost of leveraging the kernel's VFS.

OmniCache suffers from PCIe-crossing overhead inherent to its architecture. On `fsync`, data always traverses the device-side file system logic over PCIe, requiring copies from the user buffer to a DMA buffer and then to device memory. In contrast, we place the file system logic in H-Server, which enables host-side I/O without interacting with the device. Additionally, running a complex file system on the CSD's slow cores increases the latency further.

Read Latency. The sequential read latency of Oxbow is 18.2× lower than µFS for the 4KB I/O size, which is attributed to the kernel's readahead mechanism. While Ext4 achieves the lowest latency based on the kernel's readahead, it is 3–29% lower than Oxbow because of Oxbow's longer read path that involves two context switches between the kernel and H-Server and one additional memory copy (i.e., SPDK buffer to page cache). For 4KB random read, Oxbow delivers 5.5× and 7.7× lower latency than Ext4 and µFS, respectively. This is due to a faster I/O than Ext4 through a user-level driver and a prefetching effect within the 1 GB file relative to µFS. As the I/O size increases, this benchmark increasingly represents sequential reads, where Oxbow achieves lower throughput than Ext4 (§6.2.2). Compared to µFS, Oxbow's additional memory copies worsen the latency with larger I/O sizes. OmniCache shows high latency for the same reasons as in the write latency cases.

6.2.2 Throughput and Scalability

In the throughput microbenchmark (Figure 10), each thread writes 2 GB of data to its private file with 4KB I/O size and calls `fsync` after every 512 MB is written, i.e., many writes to a file followed by a single `fsync`. We increase the number of application processes from 1 to 32 (64 for reads) to measure the scalability of file systems. For this benchmark, we configure H-Server with 16 I/O worker threads. Because each client keeps at most one `fsync` (writes) or readahead (reads) request outstanding, the number of active I/O worker threads is bounded by the number of clients.

Write Throughput. For write workloads, Oxbow outperforms µFS by up to 86%. Fundamentally, Oxbow bypasses the kernel as µFS does, which enables faster I/O than Ext4. In addition, the background commit by D-Server reduces the amount of data written during `fsync`. This offsets the overheads from two more memory copies than µFS, resulting in higher throughput. OmniCache also achieves comparable throughput to Oxbow and µFS for the append and sequential write workloads. Different from the latency benchmark, the data is batched in 512 MB chunks and sent to the device. It en-

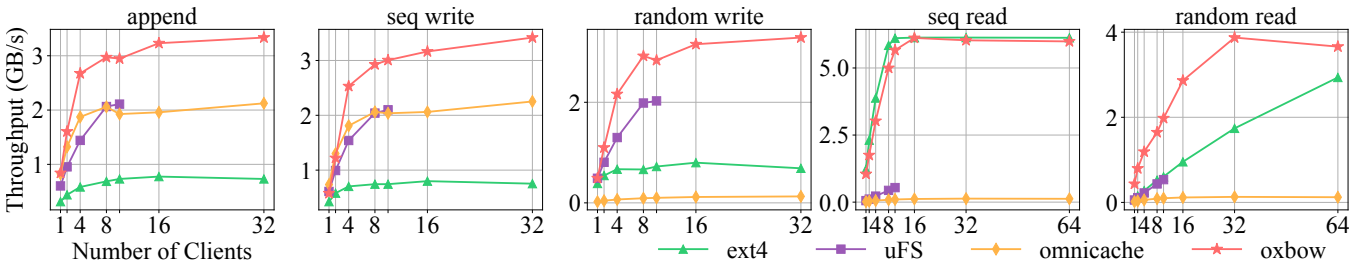


Figure 10: **Throughput microbenchmark.** The maximum SSD read and write bandwidths are 6.1 and 3.6 GB/s, respectively.

Clients	Workload	Ext4	µFS	OmniCache	Oxbow no-bg-jnl	Oxbow
1	Append	0.602 (34.21×)	0.339 (19.23×)	0.454 (25.76×)	0.155 (8.79×)	0.018 (1.00×)
	Seq write	0.628 (31.59×)	0.333 (16.77×)	0.527 (26.52×)	0.156 (7.86×)	0.020 (1.00×)
	Rand write	0.723 (28.73×)	0.442 (17.59×)	20.825 (828.12×)	0.176 (6.98×)	0.025 (1.00×)
10	Append	5.196 (4.55×)	2.866 (2.51×)	2.395 (2.10×)	1.205 (1.06×)	1.141 (1.00×)
	Seq write	5.329 (5.12×)	2.518 (2.42×)	2.255 (2.17×)	0.962 (0.92×)	1.041 (1.00×)
	Rand write	5.455 (5.57×)	1.957 (2.00×)	49.686 (50.74×)	1.030 (1.05×)	0.979 (1.00×)

Table 1: **Throughput microbenchmark `fsync` time (sec).** Numbers in parentheses are relative to Oxbow.

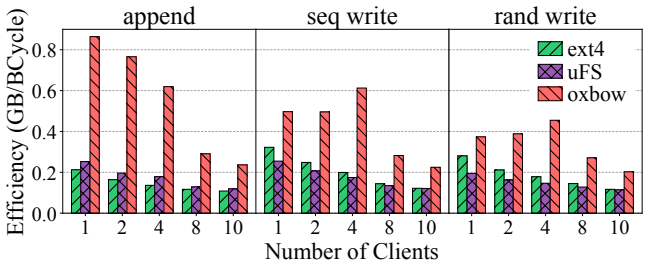
ables OmniCache to fully utilize PCIe lanes and available I/O bandwidth, hiding its high latency. On the other hand, random writes issue a large number of I/O requests for fragmented block chunks, creating a bottleneck in CSD's CPU resources. It prevents OmniCache from scaling beyond four processes. Ext4 suffers from contention in the block-based transaction composition of JBD2, resulting in inefficient use of I/O bandwidth and low throughput. In contrast, Oxbow composes a transaction at file granularity, which significantly reduces contention. As a result, Oxbow achieves 2.7–4.6×, 1.4–4.5×, and 1.3–4.8× the throughput of Ext4 for the append, sequential write, and random write workloads, respectively.

Fsync Time Reduction. We evaluate how effectively Oxbow reduces the foreground `fsync` time. Oxbow's `fsync` times are the lowest among all the other file systems (Table 1). With a single client, its `fsync` times are 16.8–19.2× lower than the second-best system, since D-Server commits most data in the background and leaves little to persist on `fsync`. With 10 clients, the gap narrows to 2.0–2.2× as I/O bandwidth saturation leaves less headroom for background commits to run ahead of `fsync`. This demonstrates Oxbow effectively reduces the foreground `fsync` latency by aggressively committing data with D-Server.

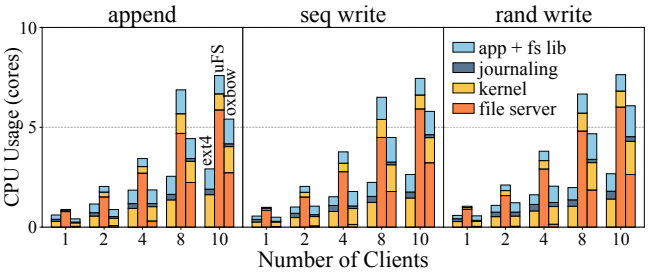
Read Throughput. For sequential read, Oxbow's throughput is 10.5–18.5× that of µFS thanks to the kernel's readahead prefetching, but 0.2–24% lower than Ext4 due to its longer read path. The kernel triggers the next readahead batch (e.g., 32 pages) when the application reaches a marked page in the current window; in Oxbow this cycle is longer because of kernel–H-Server context switches and an extra memory copy, which lowers the block-layer request rate and bandwidth utilization. Enlarging the readahead window compensates—e.g., raising it from 32 to 128 pages via a mount option lets Oxbow outperform Ext4 in all cases. Like µFS, OmniCache lacks a prefetching mechanism and achieves low sequential-read throughput; implementing custom prefetching would raise their performance but also their development cost. OmniCache further suffers from CSD CPU bottlenecks when 4 KB requests are issued one by one, as in random writes.

For random read, Oxbow achieves the highest throughput. Oxbow and Ext4 issue identical kernel readahead requests, so the difference comes from the lower layers: Oxbow uses a faster user-level driver and a lighter block layer (avoiding the kernel's I/O scheduler and interrupt-based stack), batches requests to reduce block-layer crossings, and pipelines across I/O worker threads. Even for random reads, kernel readahead prefetches some pages, disadvantaging µFS; still, its fast user-level I/O keeps it on par with Ext4, while OmniCache again suffers from CSD resource deficiency.

CPU Utilization. To demonstrate how effectively Oxbow reduces the host CPU resource consumption of user-level file systems by offloading journaling to CSD, we measure the CPU utilization of a whole system while running the throughput microbenchmark. For a fair comparison, we exclude the CPU cycles that µFS spends busy-waiting for client-server communication—but not those for I/O completion polling—when reporting its CPU usage; this is generous to µFS, since it would lose performance if such busy-waiting were removed. We introduce a metric, *efficiency*, to represent how much throughput a file system achieves given a certain amount of CPU resource, i.e., the number of bytes processed per CPU cycle. Figure 11a compares efficiency of Oxbow, µFS, and Ext4. For write operations, Oxbow achieves 1.8–3.9× the efficiency of µFS. This gain comes from D-Server's background commits, which reduce the amount of data written by the host and, in turn, lower the host CPU time spent polling for I/O completion. The efficiency of Oxbow decreases as the number of clients increases, because bursts of `fsync` calls reduce the opportunity for I/O offloading. Ext4 consumes less host CPU

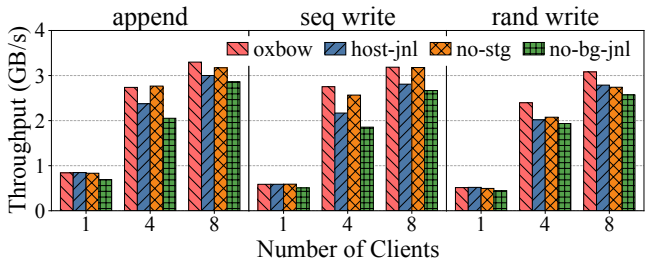


(a) Efficiency (throughput / host CPU usage). Higher is better.

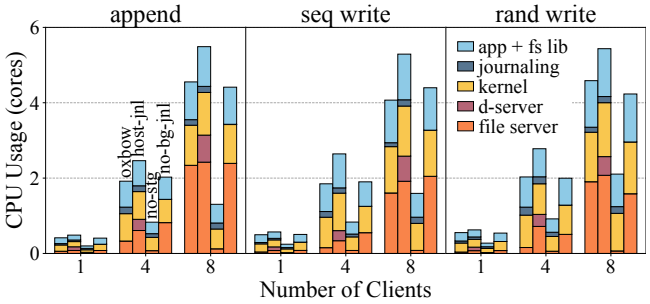


(b) Absolute host CPU usage and its breakdown. Lower is better. Bars are ordered as `ext4`, `µFS`, and `oxbow`.

Figure 11: **Throughput microbenchmark host CPU usage.**



(a) Throughput. Higher is better.



(b) Absolute host CPU usage and its breakdown. Lower is better. Bars are ordered as `oxbow`, `host-jnl`, `no-stg`, and `no-bg-jnl`.

Figure 12: **Throughput microbenchmark ablation studies.**

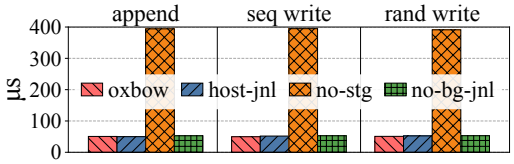


Figure 13: **Latency microbenchmark ablation study** with 4 KB I/O size. Lower is better.

resources than Oxbow, due to its interrupt-driven I/O. However, its performance is much lower than Oxbow, resulting in Oxbow achieving 1.3–4.7× the efficiency of Ext4.

Figure 11b breaks the absolute host CPU usage—reported in cores, i.e., consumed cycles/s normalized to one 2.3 GHz core—down into the benchmark application, the kernel, journaling, and the user-level file server. µFS demands the most CPU (e.g., 7.6 cores at 10 clients): it runs one busy-waiting file server thread per client to poll for I/O completion, so its file server dominates (around 80%) and grows with the client count, while its small kernel time mainly reflects memory-management overhead from its user-space processes. Ext4 instead spends most of its cycles in the kernel—where its entire storage stack, including JBD2 journaling, resides—but its interrupt-driven I/O yields the lowest overall consumption of the three (e.g., 2.9 cores at 10 clients). For Oxbow, the file server (H-Server) dominates as clients increase, since more worker threads busy-wait for user-level I/O completion once I/O becomes the bottleneck (e.g., H-Server uses 2.7 of 5.4 cores at 10 clients for append). In contrast, the background journaling thread uses only 0.1 core because it offloads journaling I/O to the CSD, and the kernel portion here covers both illuFS and the VFS.

6.3 Ablation Study

We perform ablation studies to evaluate the contribution of each major design choice of Oxbow. We compare the default Oxbow against three variants: *host journaling* (`host-jnl`), *no staging* (`no-stg`), and *no background journaling* (`no-bg-jnl`). Figure 12 reports the throughput and host CPU usage of the throughput microbenchmark (§6.2.2), and Figure 13 reports the results of the latency microbenchmark with 4 KB I/O size (§6.2.1).

Host Journaling. In this configuration, D-Server runs on the host, so Oxbow has no device component. Instead of using DMA, D-Server communicates with H-Server through shared memory, which eliminates one data copy compared to the default Oxbow. Under less contention, i.e., with fewer clients, the throughput degradation of host journaling is marginal. As the number of clients increases, host-journaling throughput becomes up to 21% lower than that of Oxbow due to host resource contention. For example, the host memory bandwidth is shared between H-Server and D-Server, and D-Server runs on the remote NUMA node because the local NUMA node lacks sufficient cores. Figure 12b shows that host journaling consumes up to 44% more host CPU resources than the default Oxbow even though it achieves lower throughput. The performance gap decreases as the number of clients grows, because of I/O bandwidth saturation.

No Staging. Staging is disabled, so `fsync` persistence requests are handled by the background-journaling path. With this configuration, all I/O operations are performed by the device, resulting in a drastic reduction of host CPU consumption (Figure 12b). However, its latency is about 7.8× that of

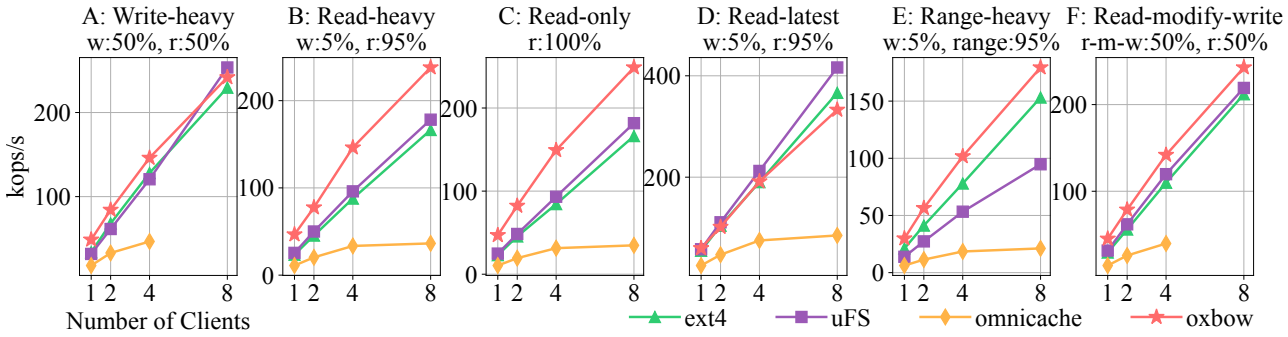


Figure 14: **LevelDB throughput with YCSB workloads A–F.** Subplot titles show the operation mix of each workload (w: write, r: read, range: short-range scan, r-m-w: read-modify-write). Higher is better.

the default Oxbow (Figure 13), because this path incurs two overheads for every `fsync`: waiting for the ongoing background commit to complete and crossing PCIe. The other configurations show similar latency to the default Oxbow.

No Background Journaling. Background journaling is disabled, so there is no periodic journaling and data is persisted only by `fsync`. In the throughput microbenchmark, this configuration shows up to 33% lower throughput because it does not utilize CSD resources, whereas the other configurations benefit from them. In particular, its `fsync` time is up to 8.8× that of the default Oxbow (Table 1), because it does not persist data ahead of `fsync` calls. Note that, even without device offloading, this configuration still outperforms Ext4, demonstrating that Oxbow's kernel-bypass write path is effective.

Type	File system	Avg	P50	P90	P99	P99.9
End-to-end (ms)	Oxbow	40	33	71	97	240
	Ext4	40	33	71	95	260
Per-probe I/O (μs)	Oxbow	61	41	150	290	530
	Ext4	122	110	210	330	960

Table 2: **RAG retrieval latency.** A probe is a single vector read from the on-disk store during a retrieval request.

File system	Throughput (MB/s)	Avg. latency (s)	Max. latency (s)
Ext4	200.47	23.87	39.72
Oxbow	316.72	12.93	13.00
OmniCache	309.42	13.24	14.34

Table 3: **LLM checkpointing throughput and latency.**

6.4 Real-World Benchmarks

LevelDB. We run LevelDB [28] with the YCSB [9] workloads to evaluate the end-to-end performance of Oxbow. We configure it to have 10M entries of 16B keys and 80B values for each client and run 100K operations [39]. Each run starts cold, so the reported run-phase throughput reflects on-device I/O rather than pure in-memory cache hits. Because the working set is spread across the 10M-key space and exceeds LevelDB's in-memory block cache, the read-dominated workloads, except for workload D, issue many reads that reach the device. Oxbow's high random-read performance makes it the best for the random-read-dominated workloads, B and C (Figure 14). It outperforms µFS, the second best, by 83% and 89% with 1 process and 34% and 37% with 8 processes for workloads B and C, respectively. Oxbow achieves the best throughput for workload E as well. In workload E, both sequential and random reads matter, as each short-range scan performs a random seek across LSM levels followed by a short sequential read. Oxbow and Ext4 outperform µFS with the kernel's readahead, while Oxbow's higher random-read performance keeps it ahead of Ext4. Oxbow has 41% and 17% higher throughput than the second best, Ext4, with 1 and 8 processes, respectively. For workload D with eight processes, where most reads are cache hits, Oxbow performs 20% worse than µFS. The gap is attributed to µFS's zero-copy between the user and file-system buffers. Workloads A and F mix reads with updates that rarely trigger `fsync` under LevelDB's default settings, so Oxbow's read-path advantage and µFS's zero-copy writes partly offset each other, narrowing Oxbow's lead as concurrency grows. Meanwhile, OmniCache's high latency limits its performance on the latency-sensitive LevelDB.

RAG Retrieval. We evaluate how Oxbow performs in the trendy AI domain. We develop a RAG (Retrieval-Augmented Generation) Retrieval benchmark. It models an SSD-backed vector search service that generates many small random reads to file systems. We construct a 20GB on-disk vector store from synthetic embeddings with a vector size of 768 (3072B) and a 1024B payload. We generate 400-client loads using a Locust [21]-based HTTP client. Table 2 shows the end-to-end and I/O latencies of Oxbow and Ext4. Although the end-to-end latency is not dominated by I/O, Oxbow has 50% lower I/O latency than Ext4 thanks to its high random read performance. Similarly, it reduces tail I/O latency by up to 45% relative to Ext4.

LLM Checkpointing. Next, we benchmark the disk I/O patterns during the checkpointing phase of large LLM (Large Language Model) training. We assume a single process that checkpoints a large tensor iteratively. In each iteration, the process writes a whole tensor to SSD and calls `fsync` at the

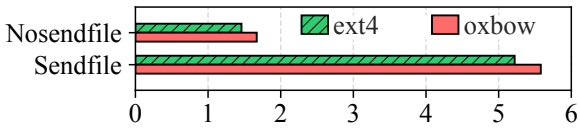


Figure 15: **Nginx web server throughput (GB/s)** with the `sendfile` enabled vs. disabled. Higher is better.

end. We configure the tensor size to 4 GB, run 5 iterations of checkpointing, and measure the average throughput and latency (Table 3). Oxbow achieves 58% higher throughput and 46% lower latency than Ext4. It is mainly due to the higher sequential write performance of Oxbow over Ext4. Ext4's high maximum latency is attributed to its heavier transaction composition, compared to Oxbow's Split Journaling and OmniCache's device-side metadata journaling. OmniCache achieves throughput comparable to Oxbow. This single-threaded workload writes a large tensor sequentially, so each checkpoint is transferred with only a few large DMA operations, incurring little PCIe-crossing overhead and no CSD CPU contention. Oxbow retains only a slight edge over OmniCache, as its background journaling persists a small amount of data ahead of `fsync`.

6.5 Kernel Interoperability

We run Nginx web server version 1.26.2 on Ext4 and Oxbow to demonstrate the benefit of leveraging the kernel's VFS layer. `sendfile` is a system call that accelerates data transfer from one file descriptor to another. It eliminates data copying between user and kernel buffers by directly copying the data between two kernel buffers within the kernel context. As Oxbow interoperates with the kernel's VFS layer, it can leverage the `sendfile` feature at no development cost.

We configure the server to run with 8 threads and compare the throughput with the `sendfile` option enabled and disabled. On clients' request, the server reads the file and sends it to them. When `sendfile` is enabled, the server copies data directly from the file descriptor to the network socket descriptor. When disabled, it reads the file into user space and writes back to the socket, incurring additional copy and context switch overheads. We use `wrk` [8] version 4.2.0 as a client load generator. It launches 10 clients, each of which sends requests to fetch a file from the server for a fixed duration.

Figure 15 shows that Oxbow exploits the kernel's `sendfile` interface as effectively as Ext4. With `sendfile` enabled, Oxbow's throughput is 3.3× higher than with it disabled, similar to Ext4's 3.6×. Note that after reading the file once, the data is copied from the page cache, which decreases the throughput difference between Oxbow and Ext4.

7 Related Work

User-level File Systems. FUSE [4, 27] forwards requests to user-space daemons via kernel modules, but this design incurs multiple user-kernel switches per operation [51]. Several studies [32, 36, 37, 39, 53] intercept system calls to avoid these overheads and achieve high performance with user-level file systems. Prior work [32, 39] considers a hybrid architecture by delegating privileged operations to the kernel. TriCache [30] moves caching out of the kernel to maximize out-of-core data-path performance, but trades away features such as cross-process data sharing, global memory management, and in-kernel optimizations. No prior work considers a cooperative user-kernel design that jointly optimizes both data and metadata paths.

File System Operation Offloading to SoC-based CSD. Various offloading techniques in the file system context have been explored [33, 47, 54, 56, 57]. Prior work [33, 47] proposes a firmware-level file system running on the CSD. Another work [57] proposes a selective data-caching scheme in addition to offloading compound operations. Other studies [54, 56] offload small tasks to the CSD and implement near-data processing. These studies focus on enabling near-data processing by using CSD resources, whereas we instead focus on orchestrating CSD offloading with other file system components.

Logical Journaling. Recent work [39, 45, 48] proposes logical journaling techniques to improve `fsync` latency in metadata journaling. We are inspired by these ideas and retrofit them to data journaling to enable offloading journaling operations to the CSD. Furthermore, we propose a new scheme that eliminates the remaining dependencies between transactions.

Development Cost Reduction. Existing work [20, 40, 42, 44, 55] highlights the importance of development cost for in-kernel file systems and proposes techniques such as microkernel architectures, layered abstractions, and safety guarantees to reduce development complexity. We also reduce file-system development cost through an architectural design that is distinct from prior work.

8 Conclusion

Fast storage and computational SSDs have fragmented the storage stack, forcing file systems to trade performance, kernel interoperability, and CPU efficiency against one another. We argue that this tension is an artifact of granularity: committing an entire file system to user space, the kernel, or the device inherits that tier's weaknesses along with its strengths. Oxbow instead places each operation where its properties are best served, with reads in the kernel page cache, writes on a user-level bypass path, and background crash-consistency work on the device. The challenge then becomes coordination cost, which Oxbow keeps low through single-writer state and by severing foreground operations from background ones. Oxbow suggests that the path beyond today's fragmented storage stack lies not in choosing one architecture, but in coordinating user space, the kernel, and the device.

Acknowledgments

We thank our shepherd and the anonymous reviewers for their insightful comments. This work was supported in part by NSF CNS-2145295 and CNS-1956007. This work was also supported by the Institute of Information & Communications Technology Planning & Evaluation (IITP) grant funded by the Korea government (MSIT) under the project (No. RS-2023-00221040 and No. RS-2025-02263378). This work was further supported by Samsung Electronics (HiPER SCOUT).

A Artifact Appendix

A.1 Abstract

Oxbow is a coordinated file system that distributes file-system operations across a user-level library (oxLib), a trusted user-level server (H-Server), a thin kernel shim (illuFS), and a device-side server (D-Server) on a computational SSD. This artifact provides the complete source code of Oxbow—including the custom kernel—together with build and run scripts, the benchmarks used in the paper, and documentation.

A.2 Scope

The artifact lets reviewers validate the experiments in the main paper by reproducing the reported results and analyzing the system's performance. Beyond reproduction, it can serve as a foundation for file-system researchers and developers who wish to build their own user-level file-system logic on top of the services Oxbow provides—namely the kernel's VFS functionality and file-system-agnostic journaling.

A.3 Content

The artifact contains the following:

- Source code of Oxbow, including oxLib, H-Server, and D-Server, along with the custom kernel that hosts illuFS.
- Scripts to build and run the system.
- Benchmarks for reproducing the performance results reported in the paper.
- Documentation listing the hardware and software requirements and a step-by-step guide to building and running the system.
- License files for the source code.

A.4 Hosting

We host the artifact on GitHub at `https://github.com/xlab-uiuc/oxbow`, on the `osdi26-ae` branch and tagged `artifact`.

A.5 Requirements

The detailed software and hardware requirements are listed in the top-level `README.md` of the artifact repository. The development and test platform matches the testbed described in the main paper; in particular, reproducing the device-offload results requires a BlueField-2 DPU and an SR-IOV-capable NVMe SSD.

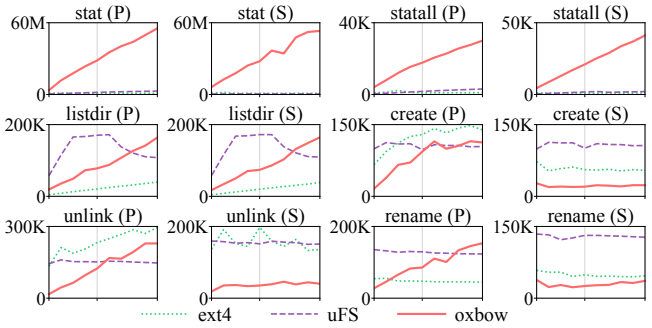


Figure 16: **Metadata microbenchmark throughput** in ops/s across 1–10 clients. P and S denote private and shared metadata operations, respectively; all operations run in memory.

Operation	Description
`stat`	`stat` a single file.
`statall`	`stat` all files in a directory.
`listdir`	`opendir`, `readdir` all entries, then `closedir`.
`create`	`mkdir`.
`unlink`	`unlink`.
`rename`	`rename`.

Table 4: **Metadata microbenchmark operations.**

B Metadata Benchmarks

We run the metadata operations from µFS's microbenchmark suite [39], comparing Oxbow against µFS and Ext4 (Figure 16). It exercises six operations (Table 4), each in a *private* variant (every client uses a separate file or directory) and a *shared* variant (all clients contend on a common one). All operations run in memory, so the measured differences stem from the file systems' software architectures.

Oxbow scales linearly on `stat` and `statall` because oxLib reads inode state directly from shared memory (§4.2) under a read lock (no contention), without kernel or H-Server crossing, whereas µFS and Ext4 pay a context switch per operation. At 10 clients, Oxbow reaches up to 19× and 10× the throughput of µFS and 57× and 30× that of Ext4. `listdir` scales likewise because oxLib reads the directory-entry cache directly from shared memory. µFS leads at low client counts by prefetching entries on `opendir`—an optimization that Oxbow could also adopt—but flattens because a single file server thread serves all directory operations by design. Consequently, Oxbow overtakes µFS at higher client counts, ending at 1.5× µFS and 4.1× Ext4. Because these lookups are read-only, Oxbow scales in the shared case as well.

The `create` and `unlink` operations go through both H-Server and the kernel; hence, Oxbow's single-client throughput is the lowest of the three systems despite an asynchronous design. User-level parallelism nonetheless lets it scale with clients, surpassing µFS by 10 clients (up to 1.6× on `unlink`) but still trailing Ext4. `rename` also enters the kernel, but Oxbow bypasses the kernel's coarse-grained VFS rename mutex, which is typically a scalability bottleneck as clients increase, and instead handles synchronization with fine-grained locks in H-Server. As a result, despite the component-crossing overhead, Oxbow scales 5.5× from 1 to 10 clients and leads in the private case. This is an example of the performance optimizations made possible by Oxbow's flexible multi-component design. In shared configurations, parent-directory lock contention prevents all three file systems from scaling across `create`, `unlink`, and `rename`.

References

[1] BTRFS. https://btrfs.readthedocs.io.

[2] Compute Express Link. https://computeexpresslink.org.

[3] Data Plane Development Kit. https://www.dpdk.org.

[4] libfuse. https://github.com/libfuse/libfuse.

[5] LWEXT4. https://github.com/gkostka/lwext4.

[6] Samsung PM1733/PM1735 NVMe SSD. https://semiconductor.samsung.com/ssd/enterprise-ssd/pm1733-pm1735.

[7] Storage Performance Development Kit. http://www.spdk.io.

[8] wrk - a http benchmarking tool. https://github.com/wg/wrk/tree/master.

[9] Yahoo! Cloud Serving Benchmark (YCSB). https://github.com/brianfrankcooper/YCSB.

[10] Samsung Z-SSD SZ985. https://semiconductor.samsung.com/news-events/tech-blog/samsung-z-ssd-sz985, Aug. 2018.

[11] Intel Optane DC Persistent Memory Product Brief, 2019. https://www.intel.com/content/dam/www/public/us/en/documents/product-briefs/optane-dc-persistent-memory-brief.pdf.

[12] PCI Express Base Specification Revision 5.0, Version 1.0. https://pcisig.com/specifications, May 2019.

[13] Samsung Brings Revolutionary Software Innovation to PCIe Gen4 SSDs for Maximized Storage Performance. https://news.samsung.com/global/samsung-brings-revolutionary-software-innovation-to-pcie-gen4-ssds-for-maximized-storage-performance, Sept. 2019.

[14] NVM Express Zoned Namespace Command Set Specification, revision 1.1b. https://nvmexpress.org/specifications, Jan. 2022.

[15] Samsung SmartSSD 2nd Generation. https://news.samsung.com/global/samsung-electronics-develops-second-generation-smartssd-computational-storage-drive-with-upgraded-processing-functionality, July 2022.

[16] TP4146 Flexible Data Placement. https://nvmexpress.org/specifications, Nov. 2022.

[17] NVIDIA BlueField-2 DPU. https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/documents/datasheet-nvidia-bluefield-2-dpu.pdf, 2024.

[18] NVM Express Base Specification Revision 2.0d. https://nvmexpress.org/wp-content/uploads/NVM-Express-Base-Specification-2.0d-2024.01.11-Ratified.pdf, Jan. 2024.

[19] ScaleFlux SFX 3000 Storage Processor. https://scaleflux.com/library/sfx-3000-storage-processor, 2024.

[20] eBPF. https://ebpf.io/, 2025.

[21] Locust. https://locust.io, 2025.

[22] NVIDIA BlueField Data Processing Unit. https://www.nvidia.com/en-us/networking/products/data-processing-unit, Dec. 2025.

[23] Samsung 9100 PRO PCIe 5.0 NVMe SSD Datasheet. https://download.semiconductor.samsung.com/resources/data-sheet/Samsung_NVMe_SSD_9100_PRO_Datasheet_Rev.1.0.pdf, Jan. 2025.

[24] ANDERSON, T. E., CANINI, M., KIM, J., KOSTIĆ, D., KWON, Y., PETER, S., REDA, W., SCHUH, H. N., AND WITCHEL, E. Assise: Performance and Availability via Client-local NVM in a Distributed File System. In *Proceedings of 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI'20)* (Nov. 2020).

[25] ANDERSON, T. E., DAHLIN, M. D., NEEFE, J. M., PATTERSON, D. A., ROSELLI, D. S., AND WANG, R. Y. Serverless Network File Systems. In *Proceedings of the*

15th ACM Symposium on Operating Systems Principles (SOSP'95) (Dec. 1995).

[26] BHAT, S. S., EQBAL, R., CLEMENTS, A. T., KAASHOEK, M. F., AND ZELDOVICH, N. Scaling a file system to many cores using an operation log. In *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP'17)* (Oct. 2017).

[27] CHO, K.-J., CHOI, J., KWON, H., AND KIM, J.-S. RFUSE: Modernizing Userspace Filesystem Framework through Scalable Kernel-Userspace Communication. In *Proceedings of the 22nd USENIX Conference on File and Storage Technologies (FAST'24)* (Feb. 2024).

[28] DEAN, J., AND GHEMAWAT, S. LevelDB: A Fast Persistent Key-Value Store. https://opensource.googleblog.com/2011/07/leveldb-fast-persistent-key-value-store.html, July 2011.

[29] ENGLER, D. R., KAASHOEK, M. F., AND O'TOOLE, J. Exokernel: An Operating System Architecture for Application-level Resource Management. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles (SOSP'95)* (Dec. 1995).

[30] FENG, G., CAO, H., ZHU, X., YU, B., WANG, Y., MA, Z., CHEN, S., AND CHEN, W. TriCache: A User-Transparent Block Cache Enabling High-Performance Out-of-Core Processing with In-Memory Programs. In *Proceedings of the 16th USENIX Symposium on Operating Systems Design and Implementation (OSDI'22)* (July 2022).

[31] GRAY, C., AND CHERITON, D. Leases: An Efficient Fault-tolerant Mechanism for Distributed File Cache Consistency. In *Proceedings of the 12th ACM Symposium on Operating Systems Principles (SOSP'89)* (Dec. 1989).

[32] KADEKODI, R., LEE, S. K., KASHYAP, S., KIM, T., KOLLI, A., AND CHIDAMBARAM, V. SplitFS: Reducing Software Overhead in File Systems for Persistent Memory. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP'19)* (Oct. 2019).

[33] KANNAN, S., ARPACI-DUSSEAU, A. C., ARPACI-DUSSEAU, R. H., WANG, Y., XU, J., AND PALANI, G. Designing a True Direct-Access File System with DevFS. In *Proceedings of the 16th USENIX Conference on File and Storage Technologies (FAST'18)* (Feb. 2018).

[34] KIM, J., CAMPES, C., HWANG, J.-Y., JEONG, J., AND SEO, E. Z-Journal: Scalable Per-Core Journaling. In *Proceedings of the 2021 USENIX Annual Technical Conference (USENIX ATC'21)* (July 2021).

[35] KIM, J., JANG, I., REDA, W., IM, J., CANINI, M., KOSTIĆ, D., KWON, Y., PETER, S., AND WITCHEL, E. LineFS: Efficient SmartNIC Offload of a Distributed File System with Pipeline Parallelism. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles (SOSP'21)* (Oct. 2021).

[36] KWON, Y., FINGLER, H., HUNT, T., PETER, S., WITCHEL, E., AND ANDERSON, T. Strata: A Cross Media File System. In *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP'17)* (Oct. 2017).

[37] LI, C., YI, R., ZHANG, Z., LIU, J., MIN, C., ZHANG, J., LUO, Y., WANG, X., WANG, Z., AND ZHOU, D. Aeolia: A Fast and Secure Userspace Interrupt-Based Storage Stack. In *Proceedings of the ACM SIGOPS 31st Symposium on Operating Systems Principles (SOSP'25)* (Oct. 2025).

[38] LIEDTKE, J. On Micro-Kernel Construction. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP'95)* (Dec. 1995).

[39] LIU, J., REBELLO, A., DAI, Y., YE, C., KANNAN, S., ARPACI-DUSSEAU, A. C., AND ARPACI-DUSSEAU, R. H. Scale and Performance in a Filesystem Semi-Microkernel. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles (SOSP'21)* (Oct. 2021).

[40] LOGAN, L., GARCIA, J. C., LOFSTEAD, J., SUN, X.-H., AND KOUGKAS, A. Labstor: A modular and extensible platform for developing high-performance, customized i/o stacks in userspace. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (SC'22)* (Nov. 2022).

[41] LV, W., LU, Y., ZHANG, Y., DUAN, P., AND SHU, J. InfiniFS: An Efficient Metadata Service for Large-Scale Distributed Filesystems. In *Proceedings of the 20th USENIX Conference on File and Storage Technologies (FAST'22)* (Feb. 2022).

[42] MARTY, M., DE KRUIJF, M., ADRIAENS, J., ALFELD, C., BAUER, S., CONTAVALLI, C., DALTON, M., DUKKIPATI, N., EVANS, W. C., GRIBBLE, S., KIDD, N., KONONOV, R., KUMAR, G., MAUER, C., MUSICK, E., OLSON, L., RUBOW, E., RYAN, M., SPRINGBORN, K., TURNER, P., VALANCIUS, V., WANG, X., AND VAHDAT, A. Snap: a Microkernel Approach to Host Networking. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP'19)* (Oct. 2019).

[43] MATHUR, A., CAO, M., BHATTACHARYA, S., DILGER, A., TOMAS, A., AND VIVIER, L. The new ext4 filesystem: current status and future plans. In *Proceedings of the Linux Symposium* (June 2007).

[44] MILLER, S., ZHANG, K., CHEN, M., JENNINGS, R., CHEN, A., ZHUO, D., AND ANDERSON, T. High Velocity Kernel File Systems with Bento. In *Proceedings of the 19th USENIX Conference on File and Storage Technologies (FAST'21)* (Feb. 2021).

[45] PARK, D., AND SHIN, D. iJournaling: Fine-Grained Journaling for Improving the Latency of Fsync System Call. In *Proceedings of the 2017 USENIX Conference on USENIX Annual Technical Conference (USENIX ATC'17)* (July 2017).

[46] PETER, S., LI, J., ZHANG, I., PORTS, D. R. K., WOOS, D., KRISHNAMURTHY, A., ANDERSON, T., AND ROSCOE, T. Arrakis: The Operating System is the Control Plane. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation (OSDI'14)* (Oct. 2014).

[47] REN, Y., MIN, C., AND KANNAN, S. CrossFS: A Cross-layered Direct-Access File System. In *Proceedings of the 14th USENIX Conference on Operating Systems Design and Implementation (OSDI'20)* (Dec. 2020).

[48] SHIRWADKAR, H., KADEKODI, S., AND TSO, T. FastCommit: resource-efficient, performant and cost-effective file system journaling. In *Proceedings of the 2024 USENIX Annual Technical Conference (USENIX ATC'24)* (July 2024).

[49] SOLTANIYEH, M., SUN, G., YAO, X., BEYGI, A., KACHARE, R., ZHAO, D., HUEN, H., CHANG, A., MURUGESAPANDIAN, S., AND KAHN, C. Revisiting Memory Hierarchies with CMM-H: Use Device-side Caching to Integrate DRAM and SSD for a Hybrid CXL Memory. In *Proceedings of the 17th ACM Workshop on Hot Topics in Storage and File Systems (HotStorage'25)* (July 2025).

[50] SON, Y., KIM, S., YEOM, H. Y., AND HAN, H. High-performance transaction processing in journaling file systems. In *Proceedings of the 16th USENIX Conference on File and Storage Technologies (FAST'18)* (Feb. 2018).

[51] VANGOOR, B. K. R., TARASOV, V., AND ZADOK, E. To FUSE or not to FUSE: performance of user-space file systems. In *Proceedings of the 15th Usenix Conference on File and Storage Technologies (FAST'17)* (Feb. 2017).

[52] VOLOS, H., NALLI, S., PANNEERSELVAM, S., VARADARAJAN, V., SAXENA, P., AND SWIFT, M. M. Aerie: Flexible File-system Interfaces to Storage-class Memory. In *Proceedings of the 9th European Conference on Computer Systems (EuroSys'14)* (Apr. 2014).

[53] YADALAM, S., ALVERTI, C., KARAKOSTAS, V., GANDHI, J., AND SWIFT, M. BypassD: Enabling fast userspace access to shared SSDs. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'24)* (Apr. 2024).

[54] YANG, Z., LU, Y., LIAO, X., CHEN, Y., LI, J., HE, S., AND SHU, J. λ-IO: A Unified IO Stack for Computational Storage. In *Proceedings of the 21st USENIX Conference on File and Storage Technologies (FAST'23)* (Feb. 2023).

[55] ZHANG, J., KIM, J., ALVERTI, C., LIU, P., JIA, W., AND XU, T. Rethinking Tiered Storage: Talk to File Systems, Not Device Drivers. In *Proceedings of the 20th Workshop on Hot Topics in Operating Systems (HotOS-XX)* (May 2025).

[56] ZHANG, J., REN, Y., AND KANNAN, S. FusionFS: Fusing I/O Operations using CISCOps in Firmware File Systems. In *Proceedings of the 20th USENIX Conference on File and Storage Technologies (FAST'22)* (Feb. 2022).

[57] ZHANG, J., REN, Y., NGUYEN, M., MIN, C., AND KANNAN, S. OmniCache: Collaborative Caching for Near-storage Accelerators. In *Proceedings of the 22nd USENIX Conference on File and Storage Technologies (FAST'24)* (Feb. 2024).

ARTIFACT EVALUATED
USENIX
AVAILABLE

Scaling the IO wall with Declarative IO

Sanjith Athlur[#*] Sara McAllister[#†‡*] Theo Gregersen[#] Timothy Kim[#]
Yiwei Chen[#] Sarvesh Tandon[#] Lucy Wang[#]
Daniel S. Berger[#§] Saurabh Kadekodi[‡] Arif Merchant[‡] Benjamin Berg[¶]
Nathan Beckmann[#] Rashmi Vinayak[#] George Amvrosiadis[#] Gregory R. Ganger[#]

[#]*Carnegie Mellon University* [†]*University of Wisconsin, Madison* [‡]*Google*
[§]*Microsoft Azure and University of Washington* [¶]*UNC Chapel Hill*[*]

Abstract

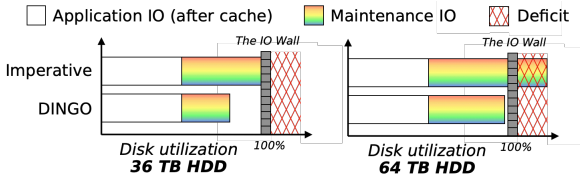
HDD capacities will greatly increase over the next ten years, lowering cost-per-TB in large-scale storage systems. Unfortunately, device bandwidth will not grow proportionally to device capacity. Hence, storage systems will face an *IO wall* where the demand for HDD IO will outstrip supply.

We find that, surprisingly, between 45% and 70% of after-cache HDD IO demand for 6 hyperscalers comes from crucial maintenance tasks that ensure data reliability and efficiency (e.g. scrubbing, garbage collection). Unfortunately, caching maintenance tasks is ineffective — individual tasks have little reuse and inter-task reuse is too far apart in time. Fortunately, maintenance tasks are flexible in the timing, ordering of data accesses, and even which data they access. However, the current imperative storage interface (e.g., read/write) hides maintenance tasks' flexible nature. We propose *Declarative IO*, a new interface for distributed storage systems that allows developers to expose tasks' flexibility to the storage system. This interface allows tasks to send a *declaration* to our distributed storage system, DINGO, specifying sets of data and their associated deadlines, such as "process all blocks of this device within 7 days". In processing declarations, DINGO coordinates IO across different tasks to create timely data reuse. DINGO achieves a 26–51% IO savings for maintenance task mixes corresponding to real hyperscalers, enabling the deployment of 1.7× larger HDDs than in imperative systems.

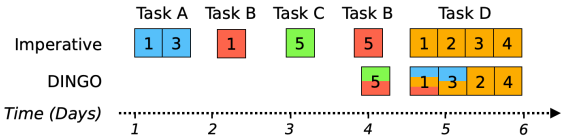
1 Introduction

Large distributed storage systems store exabytes of data over hundreds of thousands of disks [22, 33, 55, 61, 70]. Mechanical disks (HDDs) remain more cost-effective for most high-capacity workloads than flash (SSDs) due to new technologies that increase HDD density, such as heat-assisted magnetic recording (HAMR) [6, 59]. Thus, HDDs continue to be an integral piece of datacenter storage systems. Most vendors expect 40TB drives in the near-term and 100TB drives as soon as 2030, with the cost-per-TB decreasing 6× [60]. These high-capacity HDDs will allow hyperscalers to accommodate data growth while minimizing the power usage and physical footprint of their storage systems.

Storage IO wall. Unfortunately, HDD accesses are not scaling proportionally to their capacities. In particular, the IO supply — i.e., IOPS and bandwidth per device — has remained roughly constant as capacity has sharply increased over time (Sec. 2.2). Put differently, the IO supply per TB of HDD storage is trending precipitously downward. Hence, as HDDs grow denser, we are approaching a point where the total IO demand of datacenter workloads on HDDs exceeds IO supply — a phenomenon we refer to as the *IO wall*. Beyond this wall, storage systems will be unable to use higher-capacity HDDs, forgoing their power, footprint, and cost advantages.



(a) The IO wall limits HDD density in imperative storage systems.



(b) DINGO finds data reuse between different maintenance tasks, reducing disk IO.

Fig. 1: (a) Storage clusters are running into an IO wall. If maintenance tasks cumulatively use 50% of disk IO (commonly observed in hyperscalers), imperative IO uses all of a 36 TB disks' time (or has 100% IO utilization). Since DINGO reduces maintenance IO by 40%, it enables the deployment of 64 TB disks under the same IO requirements. (b) DINGO leverages Declarative IO and each maintenance task's flexibility to enable IO reuse between tasks that imperative storage systems would schedule independently over time.

To deploy higher-capacity HDDs, storage systems need to reduce IO demand. Systems have thus far adapted to decreasing IO supply by deploying flash caches that absorb application requests before they reach HDDs [16, 26, 30, 38, 49, 71]. While these caches absorb a large fraction of IO, the remaining disk IO is increasingly hard to cache. Hence, there are diminishing marginal returns to adding caching capacity.

Maintenance tasks dominate IO demand. Analysis of storage cluster workloads across six hyperscalers[1] surprisingly indicates that, on average across a cluster, 45–70% of HDD IO comes from various *data maintenance tasks* (Sec. 3).

[*]These authors contributed equally.

[1]We use the term hyperscaler to refer to companies that run large-scale, distributed datacenters, delivering extremely scalable compute and storage for many applications including cloud, AI, batch analytics, and web.

Maintenance tasks pervade datacenters, from integrity checking [39,53,58], to reliability maintenance [25,34], to capacity compaction [2,3], to adaptive efficiency enhancement [40–42]. Maintenance tasks are crucial to the durability and availability of stored data, but — importantly — *are not* latency-sensitive. They also generally access large amounts of data with little reuse, rendering caches ineffective and demanding HDD IO.

Opportunity: Maintenance IO is flexible. While maintenance tasks are individually hard to cache, we observe that *there is significant data overlap between different maintenance tasks*. For instance, while scrubbing exhibits no reuse as it scans over all blocks in the cluster, these scans will overlap with every other maintenance task. However, because maintenance tasks are not generally synchronized, most inter-task reuse is also too far apart in time to result in cache hits. Fortunately, *maintenance tasks are generally flexible in access order, access time, and even which data they access*. A single maintenance task (e.g., "scrub a given disk to check data integrity") is composed of several lower-level *requests* (e.g. "scrub a given block"). A maintenance task must complete all of its requests within a certain timeframe, but the timing of individual requests is flexible. Our goal is to leverage maintenance task flexibility to cause overlapping requests from different maintenance tasks to execute at similar times. For instance, we can change the order and timing for scrubbing accesses to mirror those from a capacity-balancing task.

Unfortunately, the current *imperative* distributed storage interfaces (e.g., `GET`/`PUT`, `read`/`write`) do not expose flexibility — each request is for a *specific* data unit to be accessed *now*. While it is possible in principle to coordinate imperative IO, in practice maintenance tasks are generated by disparate teams working on different applications or different layers of the system (e.g., block-level scrubbing vs. garbage collection from a database application). Explicit coordination of imperative IO across teams, software layers, and even organizational boundaries is impractical.

Our solution: Declarative IO. We introduce *Declarative IO*, a new distributed storage interface that allows tasks to *declare* upcoming IO reads along with a deadline for those requests. Our system, DINGO[2], processes declarations to find overlapping IO across maintenance tasks, leveraging data reuse to reduce aggregate IO demand (Fig. 1b).

DINGO uses its *IO Planner* to track IO declarations, plan which IO to schedule, and then dispatch callbacks to maintenance tasks so that they know to access their data. The IO Planner enables DINGO to reduce disk IO *without changes to maintenance tasks' requested work*. The IO Planner respects task deadlines, and is feasible to deploy for large, single-datacenter storage clusters. We accomplish this by developing a rate-based scheduler for the IO Planner (Sec. 5.1) and an efficient dispatcher (Sec. 5.2) that quickly finds and exploits overlap between outstanding maintenance tasks.

In small-scale experiments on a HDD cluster, we find that Declarative IO decreases disk reads by 26%. To evaluate DINGO at scale on a broader set of maintenance tasks, we simulate 100 PB clusters. While imperative IO is limited to 36 TB HDDs, DINGO enables the deployment of 1.7× higher capacity devices, or 64 TB, on maintenance task mixes based on real hyperscalers. Finally, we show that adding additional work to DINGO comes with only a small IO penalty. This lowers the barrier to increasing maintenance activities and improving the performance of storage clusters (e.g. more garbage collection to reduce fragmentation and save capacity).

Contributions. This paper contributes the following:

- We characterize the IO wall for HDDs and demonstrate why caching alone will not avoid this wall (Sec. 2.2).
- We identify that data maintenance tasks account for 45-70% of HDD time in storage clusters at six hyperscalers and present per-task IO breakdowns for three of these hyperscalers (Sec. 3).
- We introduce the Declarative IO storage interface, allowing tasks to declare their order, time, and data flexibility to the storage system (Sec. 4).
- We present DINGO, a storage system implementing Declarative IO. DINGO's IO Planner finds IO overlap while meeting declarations' deadlines, scales to large clusters and minimizes caching overheads (Sec. 5).
- We show that DINGO achieves 26–51% maintenance IO savings, enabling deployment of 1.7x larger HDDs (Sec. 6).

2 The impending IO wall

Distributed storage systems are the backbone of modern datacenters, containing exabytes of data across 100Ks of disks and supporting applications from ML training [21] to the internet [33]. This section describes how data flows in these storage systems (Sec. 2.1) and how HDDs' increasing capacity is leading to the IO wall (Sec. 2.2).

2.1 IO in distributed storage systems

Datacenters store and process data through a complex, interconnected collection of data storage, caching, management, and serving systems. Fig. 2 illustrates this architecture.

Datacenter architecture. The *bulk storage tier* combines 100Ks of HDDs into a single distributed storage system [22, 32, 44, 55, 62, 70]. The *caching tier* absorbs IO on DRAM or SSDs to reduce latency and handle IO that would otherwise burden HDDs [16, 26, 49, 50]. The *data management tier* consists of services such as object stores or databases that organize, index and process content in the lower storage tiers. The *application tier* consists of applications on compute clusters that read/write data from the storage tier directly or via the data management tier.

IO at the caching or bulk storage tier is requested via *im-*

[2]A dingo is Australian canine. DINGO is the Declarative INterface for Global storage Optimization. Both rely on packs: packs of dingoes to hunt kangaroos and packs of maintenance tasks in DINGO to find IO reuse.

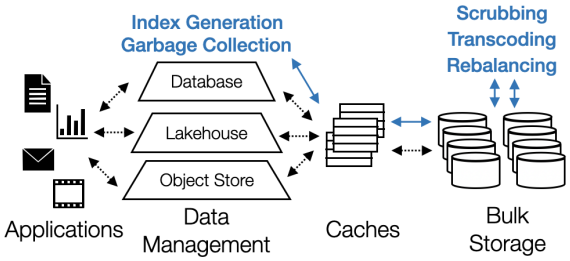


Fig. 2: Datacenter architecture has four tiers: application, data management, caching, and bulk storage. IO requests come from both applications and maintenance tasks, but maintenance tasks' IO is less cache-friendly so more of results in disk IO. In some hyperscalers, the caching tier and bulk storage tier may be combined into one distributed storage system.

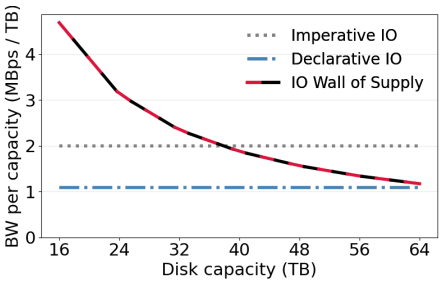


Fig. 3: As disks become larger, their IO supply decreases. Therefore, storage clusters are running into an IO wall where they will no longer have enough IO supply to fulfill their IO demand. Assuming maintenance tasks are allowed to use only 50% of disk bandwidth, imperative IO can only use 36 TB disks.

perative interfaces, interfaces that only allow read/write or get/put requests for data now. These interfaces are typical for everything from key-value stores [3] to caches [16] to distributed file systems and their distributed block-level interfaces [22, 33, 55].

Quantifying disk IO. Each tier ultimately fetches all non-cached data from disk. We measure disk IO with two metrics: (1) bandwidth and (2) *disk-head time* [15, 26, 57], the sum of the disk's seek latency and transfer time for the requested data. Since disk-head time measures active time of the disk, we can divide it by wall-clock time to get disk utilization[3].

2.2 HDDs are IO bound

While HDDs have seen significant increases in capacity, their seek latencies and transfer rates have not improved proportionately [28, 59]. Thus, the disk-head time of a given request has remained roughly constant as HDD capacities have grown. For a given unit of time, a smaller percentage of the disk's data is accessible as its capacity increases. This trend will accelerate over the next ten years as HAMR devices, which have recently come to market, lead to continued increases in HDD capacity [1, 4, 5, 42].

We describe both the IO supplied by the storage system and the IO demanded by maintenance reads in terms of *Bandwidth-per-TB*. Fig. 3 shows the projected IO supply (in MB/s-per-TB) for maintenance reads (Sec. 3) as HDD capacity increases. Given a constant IO demand, storage systems will not have enough IO to deploy HDDs larger than 36 TB.

We term this IO-induced capacity limit the *IO wall*. Unless system designers find a way to decrease IO demand, *distributed storage systems will hit an IO wall, where IO demand will significantly outstrip IO supply*, and prevent larger HDDs' cost, power, and footprint advantages.

Caches cannot climb the IO wall. The main tool to address decreasing IO supply on HDDs has been caching. In particular, datacenters deploy increasingly large SSD caching tiers to absorb an outsized fraction of IO demand [16, 19, 26, 71] for reasonable cost. So far, scaling the caching tier to offset decreases in HDD IO supply has been feasible.

Unfortunately, we are now reaching the end of this scaling regime. The caching tier already absorbs the easily cacheable parts of the IO workload. The IO that currently passes through the caching tier to the bulk storage tier tends to be composed of scanning access patterns with little data reuse (Sec. 3). Hence, adding additional capacity to the caching tier will not significantly reduce this HDD IO demand, closing the gap between the decreasing IO supply and static IO demand.

Key insight. Latency-sensitive application IO dominates the pre-cache IO workload, and is largely absorbed by the caching tier. On the contrary, flexible maintenance IO tasks access large amounts of data *with little reuse* and are hard to cache. After years of scaling the caching tier to absorb more application IO, maintenance IO now accounts for a significant portion of disk IO. As adding additional cache capacity does little to absorb maintenance IO, storage systems need a new approach to scale the IO wall.

3 Maintenance tasks

To overcome the IO wall, we need to reduce maintenance IO. This section discusses why maintenance tasks are necessary (Sec. 3.1), analyzes the IO demand of maintenance tasks (Sec. 3.2), describes the challenges in reducing maintenance IO (Sec. 3.3), and discusses how flexibility provides an opportunity to reduce maintenance IO (Sec. 3.4).

3.1 Maintenance tasks are essential

Distributed storage systems need to ensure data durability and efficient accesses. Thus, storage systems are designed to provide fault tolerance and reliability while minimizing request latency, maximizing throughput, and limiting the total cost of the system. To meet these goals, storage systems run essential maintenance tasks, such as integrity checking (reads data and metadata to check fidelity, e.g., scrubbing [39, 53, 58], fsck [29, 47, 51]), reconstruction (rebuilds data from erasure coding after detecting data loss), and rebalancing (copies data to enable fault tolerance, load balancing, capacity distribution, etc.). Some maintenance tasks may be even more obscure, like moving data for "short stroking" (places hotter data on the

[3] See Sec. A.1 for details on calculating disk-head time.

outer HDD tracks for faster transfer times and moves colder data to inner tracks).

While the distributed storage system generates some maintenance tasks itself, much originates from other tiers (Fig. 2). Shingled disks internally garbage collect to reclaim overwritten tracks [28]. Log-structured stores and LSM-tree-based databases perform garbage collection or compaction to minimize space overhead and reduce read amplification [2, 3, 23, 45, 54]. Data lakehouses transcode data as it ages to save space while storing newer data in narrower encodings to lower tail latency [42] and geo-replicate data to ensure redundancy to datacenter-scale failures [67, 69]. Databases create indexes [14, 43, 48], materialized views [18, 35], and summary statistics [20] for better performance[4].

3.2 Maintenance tasks use most HDD IO

Although data maintenance activities play a critical role throughout the datacenter, prior literature does not address their cumulative impact on HDD utilization. Based on communication with six hyperscalers and analysis of their storage workloads, we find that maintenance tasks account for *at least 45-70%* of disk IO. This number is a lower bound for each hyperscaler — the remaining IO is from both applications and unaccounted maintenance (e.g., database-tier maintenance tasks running on top of cloud storage). For simplicity, we refer to the remaining IO as application IO.

Breakdown of maintenance IO. To understand the breakdown of maintenance IO between tasks, we examine data from three hyperscalers: Meta, Microsoft, and Google (listed here in a random order for anonymity). For these hyperscalers, we present each maintenance tasks' disk IO normalized to that hyperscaler's total known maintenance disk IO.

Hyperscaler A. We present measured bandwidth, IOPS, and estimated disk-head time from a storage cluster over 3 months in 2025. These results highlight why we favor disk-head time as a metric for IO: IOPS understates the impact of large requests and bandwidth understates the impact of per-request overheads.

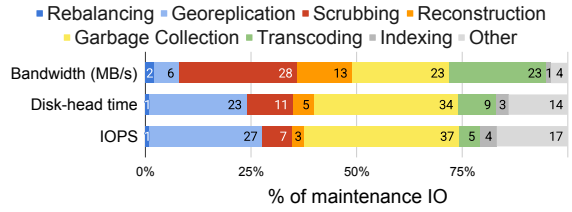


Fig. 4: The average IO from different maintenance tasks over 3 months at hyperscaler A, broken down by bandwidth, estimated disk-head time, and IOPS.

Hyperscaler B. We present data on disk-head time for reads from maintenance tasks over 2024 from three general-purpose clusters at hyperscaler B (Fig. 5). Rebalancing comes from 3 independent maintenance tasks since 3 different tiers in the distributed architecture control different data placement decisions. Other tasks include metadata operations, fsck, and short stroking. We note that measuring maintenance IO throughout the datacenter architecture is important. For example, maintenance IO generated by the bulk storage tier comprises only 43.9% of the total maintenance IO of Cluster 1.

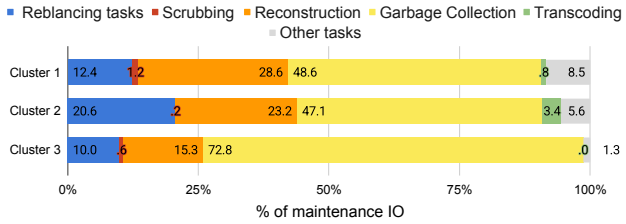


Fig. 5: For hyperscaler B, we show the maintenance task split of 2024 disk-head read time from three different clusters.

Hyperscaler C. We finally present data on disk-head time from the perspective of hyperscaler C's an object store over the course of a week in 2024. Migration reads data to move it to a separate cold storage system. Other includes object-level scrubbing and other tasks. Hyperscaler C's final task is a programmatically-combined compaction and transcoding task. This combination is possible because both tasks are generated by the same system, but trying to replicate this coordination across different tiers is prohibitively complicated. This IO breakdown also does not include the copious amounts of maintenance IO generated by the bulk storage system itself upon which the object store depends, highlighting the importance of considering maintenance IO across tiers.

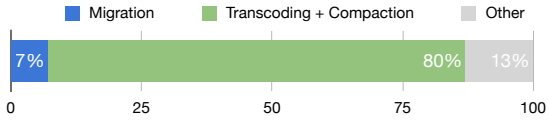


Fig. 6: Disk-head time spent for each maintenance task in hyperscaler C's object store. Hyperscaler C combines transcoding and compaction into one operation to reduce IO.

Observations: Based on this data, we make three main observations that will shape how we reduce maintenance IO. (1) Maintenance tasks account for the majority of after-cache disk IO across many hyperscalers and their different storage architectures. (2) Both hyperscaler A and hyperscaler B's maintenance IO comes from many different tasks. (3) Maintenance tasks occur across layers of the datacenter stack — such as rebalancing in hyperscaler B occurring at 3 separate tiers and object-level scrubbing in hyperscaler C's object store. Sec. 3.3 discusses how these observations complicate reducing maintenance IO.

3.3 Challenges in reducing maintenance IO

We identify two crucial challenges to reducing maintenance IO: (1) maintenance tasks are hard to synchronize at scale and (2) maintenance tasks' IO requirements scale with capacity.

[4] More details on maintenance tasks in Sec. A.2.

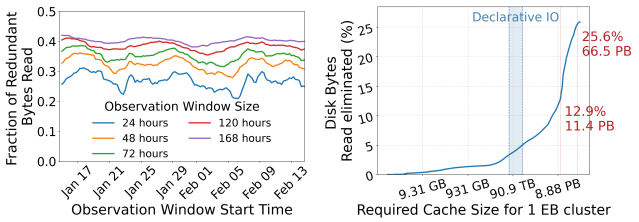


Fig. 7: Fraction of redundant reads in a cluster at Google over varying windows of time flexibility over a month in 2024 (left), and cache size required to eliminate redundant reads in 1 EB cluster with imperative IO in one 24 hour window (right).

Reuse between maintenance tasks is hard to cache. Even though maintenance tasks are not cache friendly (Sec. 2.2), there is data overlap *between* maintenance tasks. For instance, scrubbing touches all data, overlapping with every maintenance task. Hyperscaler B has 3 different rebalancing tasks that independently choose data to prevent adding unnecessary complexity, but should have overlap in which data to move.

More concretely, our analysis of publicly released Google I/O traces from 2025 [56] reveal substantial data reuse across applications, but with reuse distances too long to be effectively captured by conventional caches. Fig. 7 shows the fraction of redundant disk reads in a Google cluster over one week in different periods of observational windows. We find that approximately 24% of after-cache disk read traffic is redundant when requests may be delayed or reordered within a 24-hour window. This redundancy increases to approximately 42% when reads are observed over 7 days [5].

Using a Miss Ratio Curve (MRC) computed from the LRU stack distance of 4 KB blocks, we find that for capturing all of the reuse over a 24-hour window using caching alone would require increasing the cache tier by at least 6% of cluster capacity (66.5 PB in a 1 EB cluster). This is an order-of-magnitude more caching than hyperscalers typically have [26, 55]. Thus, we need to find a different way to capture reuse other than just caching. With Declarative IO, we show that we eliminate significant redundancy in bytes read with two orders-of-magnitude less cache space (Sec. 5.2).

Maintenance tasks are hard to synchronize at scale. If only a few maintenance tasks dominated maintenance IO, hyperscalers could effectively reduce IO demand by manually synchronizing overlapping maintenance tasks. However, across both hyperscalers, we observed a large number of maintenance tasks account for the aggregate maintenance IO (Sec. 3.2). Hence, many possible groups of tasks could be synchronized, with no single maintenance task representing a clear target for optimization. Additionally, the manual synchronization approach does not scale well to combining a large number of tasks. This development effort is made even more difficult by the need to combine tasks from different tiers of the datacenter architecture that different teams or even different organizations manage. To reduce maintenance IO, we need a solution that allows us to use maintenance task overlap at scale, and across many tasks.

[5] Although the provenance of IO in the trace is anonymized, and some requests may correspond to inflexible or latency-sensitive application I/O, we expect such requests to constitute only a small fraction of the workload because our analysis focuses on *after-cache* disk traffic, which is typically dominated by maintenance workloads.

Many maintenance tasks grow with capacity. One might hope that, as the overall capacity of the storage system grows, data would become on average colder and the disk IO-per-TB demand would decrease. Then, the reduced IO demand would allow device capacity to grow despite its lower IO supply.

Unfortunately, we find that this is not the case because maintenance tasks' IO grows with capacity. For example, assuming a constant disk failure rate, the total IO of reconstruction over time scales with the cluster's capacity. Our analysis of hyperscaler B's data suggests that maintenance tasks will use an additional 0.9% of a disk's time for each additional TB of HDD capacity. Hence, counterintuitively, we cannot amortize the impact of data maintenance across larger storage capacities and we need to address maintenance tasks' IO.

3.4 Opportunity: Maintenance IO is flexible

While data reuse across maintenance tasks typically occurs too far apart in time to be captured by traditional caches, maintenance tasks exhibit significant flexibility. Maintenance tasks often have *order-* and *time-flexibility*: target data must be read by a *deadline*, but these reads can occur in any order and at any time within said deadline. For instance, scrubbing aims to validate every data block within a specified period, but it does not care about the order of blocks read. We could leverage this flexibility to schedule different tasks' overlapping accesses close together, increasing cache hits and decreasing disk IO. Some maintenance tasks also exhibit *data-flexibility*: they are flexible about *which* exact data they need. For instance, rebalancing typically has many data blocks to choose from when moving data. We could carefully select blocks that other maintenance tasks were already planning to read, reducing IO through rebalancing's data flexibility.

Imperative IO impedes flexibility. Unfortunately, there is no way to express flexibility in today's imperative storage interface (e.g., read/write). The imperative interface only allows a task to specify that it needs a *specific* piece of data *now*, imposing order, time, and data constraints. Thus, data reuse can only be found through explicitly combining task implementations and manually aligning IO patterns.

Distributed storage needs a new interface. To exploit maintenance task time-, order-, and data-flexibility in a scalable manner, we need a new interface for distributed storage systems. We propose *Declarative IO*, a *declarative* interface that allows maintenance tasks to specify sets of data blocks that must be read before a deadline. With this information, the storage system can delay or reorder IO to increase temporal locality. While declarative interfaces are found across many

areas of computing [9, 24, 52, 72], we believe we are the first to introduce the concept to distributed storage's IO interface. The Declarative IO interface gives us a tool to confront the IO wall by reducing maintenance tasks' growing share of IO.

4 Declarative IO

In this section, we define the Declarative IO interface (Sec. 4.1, Sec. 4.2), discuss how maintenance tasks can adopt a declarative paradigm (Sec. 4.3), and explore the correctness guarantees that Declarative IO provides (Sec. 4.4).

4.1 Interface Overview

The Declarative IO interface centers on a `declare` function, which sends a *declaration* to the storage system describing a set of flexible read requests. The caller (e.g., maintenance task) passes a description of the data it wants to read, a deadline by which the data is needed, and a callback to notify when parts of the declared data should be read. Upon callback invocation, the caller reads the specified data through the standard imperative interface. The storage system will invoke the callback, potentially multiple times, such that all declared data can be read before the deadline.

Unlike imperative IO, Declarative IO does not guarantee when or in what order data is read, only that the appropriate callbacks will be called before the deadline. This requires: (1) targeting append-only storage systems where sealed blocks are immutable, (2) tasks handling the possibility that declared blocks may be deleted before callback, and (3) tasks using the block IDs specified in each callback to issue imperative reads for the data they need.

Declarative IO allows tasks to express flexibility. Expressing a declaration's maximum flexibility is vital to reducing IO. `declare` allows tasks to express order-, time-, and data-flexibility. A single call to `declare` specifies groups of data that the task wants to read. These groups can be read in any order (order-flexibility) as long as they are completed by the deadline (time-flexibility). While the system's default behavior is to make *all* declared data available, the caller can specify that only a subset of data is necessary (data-flexibility).

Asynchronous callbacks in `declare` provide a mechanism for the storage system to shift, reorder, and select declarations' data based on this flexibility. By shifting and reordering these callbacks, the system turns data reuse that originally spanned weeks (ineffective to cache) into overlapping reads that occur within minutes (easy to cache). Furthermore, data-flexibility creates reuse that is difficult to express imperatively by giving the storage system choice over data selection.

By changing the storage system interface, we can aggregate flexible declarations from throughout the system architecture. Any task that calls `declare` exposes its flexibility to the storage system. This allows us to find, create, and exploit data reuse between tasks ranging from a node-local scrubbing task to a file-system transcoding task to an object-store garbage-collection task (Sec. 6).

```
1 class BlockSet { set<BlockId> block_ids };
2 void declare(list<BlockSet> read_sets,
3   size_t sets_needed, time_t deadline,
4   void callback(list<BlockSet> sets_selected,
      bool overloaded));
```

Fig. 8: Maintenance tasks use the `declare` *call to explicitly communicate their flexibility to the storage system. Maintenance tasks specify three main components: what data they need (via* `BlockSet`*s and* `sets_needed`*, when they need that data by (via the deadline), and how to notify the task to read the data (via the callback).*

4.2 Interface Details

We formalize the interface for `declare` in Fig. 8. A declaration must identify the data to be read, how much of that data must be read, a deadline, and a callback.

Declaring data. Maintenance tasks need to specify which data they want to read. Importantly, many tasks need to read data in specific groups (e.g. stripes, files, objects), but have order and flexibility between these groups. Therefore, `declare` takes the `read_sets` argument, which represents a list of `BlockSets`, the group of data that a maintenance task needs at the same time. Specifically, each `BlockSet` in `read_sets` is a set of `BlockIds`, each representing a unique block of data in the distributed storage system[6].

To support data-flexibility, `declare` optionally takes a `sets_needed` argument, allowing maintenance tasks to specify how many `BlockSets` must be read from the declaration. If `sets_needed` is specified, the system will schedule some subset of *exactly* `sets_needed` elements of `read_sets`. Otherwise, all elements of `read_sets` will be scheduled.

Deadline. Although maintenance tasks are not latency-critical, they still have performance requirements. Maintenance tasks use the `deadline` argument to specify the time by which a declaration must be completed. In practice, deadlines can range from hours (e.g., for high-priority reconstruction) to multiple weeks (e.g., for scrubbing). Specifying the longest feasible deadlines improves time flexibility and thus enables more IO savings. If deadlines cannot be met due to overload, the callback is invoked with `overloaded=true` and tasks may fall back to imperative reads.

Callback. Maintenance tasks need to know when to read data that they declared. Therefore, they pass in the `callback` argument to `declare`. When a maintenance task should read one or more of its declared `BlockSets`, the storage system will invoke the callback and pass back a list of `BlockSets` to read. The storage system can invoke the callback multiple times depending on how the storage system schedules IO for the declaration. Thus, the callback's list of `BlockSets` may be a subset of `sets_needed`.

The callback does not include `BlockSet`'s data. Rather, the

[6]These blocks are *not* the same as LBAs in a local file system. They are the unit of data tracked in the distributed file system and are typically both much larger (>8 MB) [42] and stored as a file in a bulk storage node's local file system. They are also immutable (Sec. 4.4).

callback serves as a prompt from the storage system to a task that reading the specified data *now* will result in IO savings. Leaving the task in charge of reading the data simplifies consistency, failure handling, and access control by leveraging existing storage system consistency mechanisms (e.g., leases) when reading (Sec. 4.4). Additionally, Declarative IO may perform *callback elision* and not callback maintenance tasks about `BlockSets` that are completely deleted.

Extending `declare` to files. Some maintenance tasks may express their data needs in higher-level abstractions such as objects, database tables, or files. To support these tasks, we extend Declarative IO to file declarations. Specifically, maintenance tasks can declare sets of file segments (tuples of file path, offset, and length)[7]. These calls are translated into `declare` calls at declaration time using the metadata service. This means that declarations are snapshots at declaration time; new blocks appended to files after declaration are not included (Sec. 4.4). A similar translation strategy would allow tasks to declare other data abstractions.

4.3 Converting tasks to Declarative IO

While many developers are well-versed in writing IO-efficient code using an imperative interface, there are different design considerations when using Declarative IO. Declarations should remain simple, while exposing as much flexibility as possible. We show two examples of building effective declarations that (1) express all time- and order-flexibility and (2) hunt for data-flexibility[8].

Express all time- and order-flexibility. Consider scrubbing, a task that periodically checks that each block in the system is still valid over a long time horizon (e.g., monthly). Imperative scrubbing iterates over all blocks at a fixed rate and in a fixed order. A naïve declarative scrubbing might instead generate declarations when imperative scrubbing would have issued a read, with a deadline before the next declaration. This naïve approach exposes limited time-flexibility (Fig. 9) and no order-flexibility. A better approach would use a single declaration to read all blocks with a one-month deadline, giving the storage system freedom to decide the order and timing of scrubbing's reads and creating massive overlap with other tasks. While scrubbing is a simple example, it illustrates how a small change in declaration pattern can drastically change the flexibility afforded to the storage system.

Hunt for data-flexibility. Now, consider capacity balancing, which moves data to ensure that all disks have about the same amount of data. Capacity balancing needs to move *some* amount of data, not any *specific* data, making it well-suited to using Declarative IO to express data-flexibility. Even though it suffices to move a small set of blocks, capacity balancing should declare all blocks that *could* be moved (potentially all

[7]The exact interface is in Sec. A.3.

[8]An example of applying these concepts to a more complex task is in Sec. A.4.

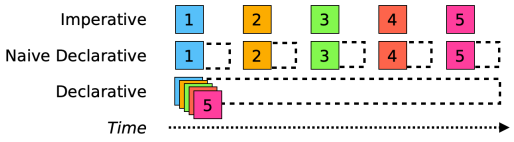


Fig. 9: Comparison of how scrubbing changes with Declarative IO. Declarative scrubbing expresses both order- and time-flexibility by declaring all blocks at once with a long deadline.

blocks on a disk) and specify how many *need* to be moved (Fig. 10). The storage system can then create data reuse by selecting blocks also needed by other declarations. Notably, larger HDDs tend to have more candidate blocks for capacity balancing, resulting in more data-flexibility.

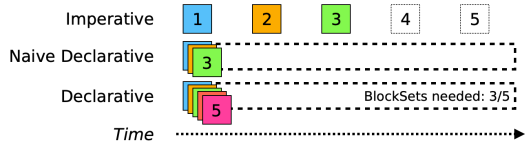


Fig. 10: Comparison of how capacity balancing requests blocks over time. Leveraging the balancing task's data-flexibility maximizes the chance for overlap with other maintenance tasks.

4.4 Consistency with Declarative IO

Declarative IO changes what data a task reads when. The following section describes our assumptions about distributed storage system behavior, how Declarative IO provides consistency based on these assumptions, and what happens if a declarative storage system cannot meet deadlines.

Storage system assumptions. We target append-only distributed storage systems. Append-only storage is the standard approach in hyperscalers [31, 33, 36, 44, 55, 70] due to its relative simplicity enabled from its block-level immutability guarantees. Append-only file systems enforce block-level immutability by sealing fully-written blocks. The only way to change a sealed block is to delete it.

Declarative IO only operates on sealed blocks. Given that these blocks are immutable, we only need to reason about the correctness properties of certain file operations: deletions, creations, appends, and metadata operations.

Deletions. A declaration can refer to a block that will be deleted before the declaration's deadline (e.g. *Block 4* in Fig. 11). We have to assume that Declarative IO callbacks may include deleted blocks, since blocks can be deleted during callback invocation. Fortunately, maintenance tasks' use imperative reads and thus inherit their guarantees. Existing distributed storage systems already handle similar problems (e.g. cached block metadata can become stale by the time of the actual request). Thus, Declarative IO relies on the pre-existing storage system's consistency to ensure it operates correctly when declared blocks are deleted.

Declarative IO may also perform *callback elision* for `BlockSets` composed entirely of deleted blocks. That is, to prevent the storage system from processing repeated requests

for deleted blocks, the system may forgo callbacks for any deleted `BlockSets`. Since these reads would have failed, callback elision does not change the data a maintenance task sees, but may require updating some error-handling in the task.

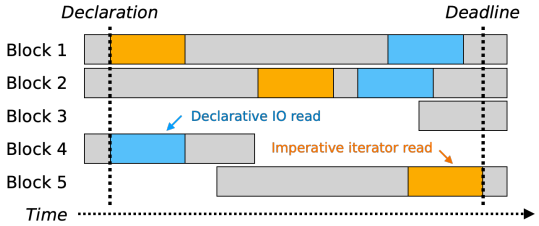


Fig. 11: An imperative vs a declarative rebalancing implementation read different blocks even if rebalacing is implemented in a naive manner (Sec. 4.2). Declarative rebalancing would not read blocks 3 or 5 because they were written after the declaration.

Metadata and data consistency. Declarative IO guarantees that file declarations will be a snapshot of blocks that exist at declaration time. It does not know about subsequent file appends, creations, or metadata changes (e.g., renaming, changes to access permissions). For instance, after a maintenance task makes a file declaration, new data can be appended to the file that would have been in the declared segment. The resulting new blocks will not be part of that declaration. Thus, tasks must create additional declarations if they need to read these newly appended blocks or any newly created files. This is not a new problem — most maintenance tasks already have to handle new files and blocks in their imperative iterators.

Any metadata changes, like access permissions changes, must be handled by the maintenance tasks themselves. After receiving a callback, a task will reobtain metadata needed for its imperative reads, both to check read permissions and file mappings. Hence, while metadata changes can reduce IO savings, they do not affect correctness.

Corner cases with Declarative IO vs imperative interfaces. When introducing a new interface to distributed storage systems like Declarative IO, it can introduce risk of failures. We find that many of these issues in Declarative IO are simply new instances of typical failures in imperative distributed storage or they are easily remedied with a slight reduction in IO reuse. To show this, we discuss three scenarios: coordinated disk failures causing large amount of reconstruction, compute unavailability, and IO load spikes.

Correlated disk failures. When many disks fail at once, reconstruction becomes urgent for all stripes that have lost multiple blocks due to a low mean-time-to-data-loss (MTTDL). Since this reconstruction is high priority due to the risk of data loss, it should use imperative requests instead of declarations. In most systems that can lose multiple blocks without data loss, this is both a rare event (thus not expected to significantly affect Declarative IO's overall IO reduction) and a potentially catastrophic event (thus already requiring extra attention to avoid data loss). Thus, these disk failures do not introduce new challenges in declarative vs imperative systems.

Compute unavailability. While imperative requests come from a scheduled compute job, Declarative IO may send its callback to tasks that do not have enough compute resources at the time of the callback, a new type of failure. While maintenance tasks generally require little compute and we expect compute to usually be available, compute unavailability, at worst, means the task cannot use the data upon callback[9], resulting in additional IO when the task is able to respond to the callback. This is still correct behavior, even if it will decrease IO reuse if its a frequent occurrence.

IO load spikes. Both of the other failures hint at a different type of failure, a spiking IO load that overwhelms the disks' ability to respond to IO. This is not a new problem and can be caused by higher than average user loads, cache failures, or unsustainable maintenance loads. In many of these cases, Declarative IO gives system owners new tools to reduce (Sec. 6.2) and handle IO spikes due to centralizing maintenance IO. Declarative IO allows owners to monitor whether maintenance IO will likely complete before the deadlines appear and to lower the maintenance IO rate in the event of too much IO from other sources (Sec. 5.1).

However, if Declarative IO ends up in a state where it is failing to meet most deadlines resulting in imperative IO from maintenance tasks and the system's IO is provisioned expecting a certain level IO reuse, the system can enter a metastable failure [37] analogous to caching. While this is an important edge case to be aware of in a declarative system, we do not believe Declarative IO adds additional risk when compared to the main alternative — *much* larger caches with their own metastable failure patterns.

5 DINGO Design

We introduce DINGO, a distributed storage system that implements Declarative IO. DINGO consists of an append-only file system with a metadata service and storage nodes (Fig. 12). Like other distributed storage systems, DINGO supports an imperative interface for both maintenance tasks and foreground applications. DINGO's main contribution is its *IO Planner*, which enables the Declarative IO interface. We first give an overview of how imperative requests, metadata requests, and declarations work in DINGO, and then discuss the design and optimizations in the IO Planner.

Imperative and metadata requests. Imperative and metadata requests (read, write, file creation, etc.) operate the same as in other distributed storage systems. Imperative requests first go to the metadata service to translate file requests into node-specific block requests and to get a lease on the requested data. The imperative task then queries this data from the appropriate data nodes, traversing the cache.

[9] Due to caching, a declarative system can handle a reasonable amount of latency in accessing IO as discussed in Sec. 5.2)

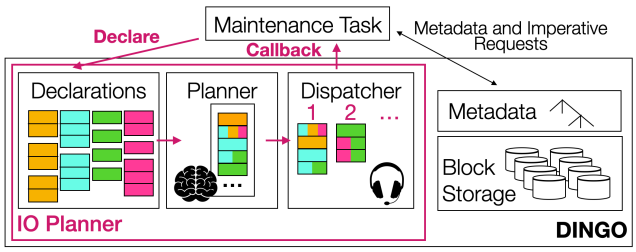


Fig. 12: DINGO *is a storage system that implements Declarative IO in addition to an imperative interface. Declarations go to the IO Planner, which keeps track of outstanding declarations, schedules them each quantum, and then dispatches them based on reuse.*

Declarations. Unlike imperative requests, declarations go to and are stored by the IO Planner. Each scheduling quantum, the IO Planner decides which `BlockSets` (a set of blocks needed by a declaration, Sec. 4.1) to fulfill, aiming to meet all declaration deadlines while adhering to a per-quantum IO limit. For each scheduled `BlockSet`, the IO Planner dispatches callbacks, prompting the maintenance tasks to read the associated data. A good scheduling algorithm aims to maximize overlap in `BlockIds` between scheduled `BlockSets` as this translates to data reuse over a short time horizon, making the maintenance workload more cacheable. DINGO uses a rate-based scheduler to optimize IO while making progress towards deadlines (Sec. 5.1). We also optimize DINGO with a dispatcher that reduces the required cache space, minimize memory overheads in the IO Planner, and explicitly manage the cache for DINGO to ensure reuse is captured (Sec. 5.2).

5.1 Scheduling in DINGO's IO Planner

The IO Planner seeks to minimize disk IO needed to fulfill all declarations by their deadlines. For each scheduling quantum, it must decide which of the outstanding `BlockSets` to schedule given that it can only schedule a limited amount of total disk IO. To minimize disk IO, an optimal schedule will maximize block reuse, but finding the optimal schedule is NP-hard[10] and we need a scheduling solution that can run periodically as tasks create new declarations. Thus, we develop a rate-based scheduling heuristic.

Rate-based scheduling. For DINGO's scheduling heuristic, we prioritize meeting deadlines over finding data reuse because (1) scheduling for reuse is NP-hard, (2) we find that this approach still produces significant data overlap in practice, and (3) missing deadlines leads to more imperative reads. To bias the scheduler towards meeting deadlines, we implement a rate-based scheduling heuristic.

For each declaration, the IO Planner calculates the IO rate needed to complete the request by its deadline. Until the quantum's total disk IO limit is reached, the IO Planner schedules declarations from highest to lowest IO rate. Any blocks already scheduled are considered "saved" and do not count towards the total disk IO limit (i.e., `BlockSet` 1 in Fig. 13 only causes one disk IO, leading to 2 saved reads). Declarations that consist of entirely saved blocks will also be scheduled. Within each declaration, we choose `BlockSets` that have the most free blocks in the quantum. Thus, the IO Planner finds more free blocks for declarations with more data-flexibility.

[10]The problem even without deadlines reduces to VM packing [63] as described in Sec. A.5.

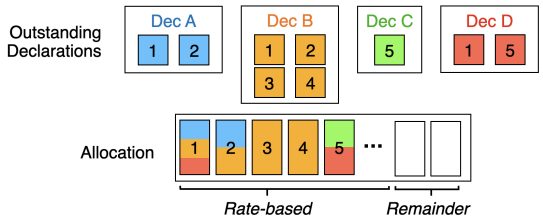


Fig. 13: DINGO*'s IO Planner must decide which blocks to schedule each quantum within its disk IO allocation. It uses a rate-based scheduler which biases toward completing all declarations. When multiple declarations access the same block (e.g., declarations A, B, and D read block 1), only one disk read is needed for that block because these reads are now close together in time.*

Dispatching decouples cache space and scheduling quanta. While the rate-based scheduling heuristic accomplishes its goal of prioritizing deadlines while finding reuse, we cannot naïvely deploy it in the IO Planner. To maintain the assumption that each unique block in a scheduling quantum only incurs one disk IO, the cache needs enough space for all saved blocks in that period. Therefore, to reduce cache resources, the ideal scheduling quantum is short. However, this is not practical as the scheduler's runtime is O(`BlockSets`).

Instead, the IO Planner uses dispatching logic to divide scheduled IO into smaller *dispatching groups* of overlapping `BlockSets`. By invoking each group's callbacks at even intervals during the quantum, we break the dependency between cache overhead and scheduling period: the cache now only needs enough space for saved blocks in a single dispatching group. Furthermore, since dispatching only considers scheduled `BlockSets` rather than all declared `BlockSets`, it can be optimized for frequent execution (Sec. 5.2).

Handling leftover IO allocation. The IO Planner may meet all the declarations' rates and still have remaining disk IO (white boxes in Fig. 13). One option is to schedule more `BlockSets` and fully use the allocation. However, this is not always ideal as it can lower future overlap. The default policy in DINGO is to leave this remaining IO unscheduled and available for other higher-priority IO. We find that in practice we do not violate deadlines due to this policy even when the per-quantum disk IO limit changes over time (Sec. 6.4).

5.2 Optimizing the IO Planner

We now discuss optimizations in DINGO's IO Planner: (1) a dispatcher leverage erasure coding for efficient overlap identification and (2) how DINGO uses explicit cache management to ensure IO savings.

EC-group based dispatcher to minimize runtime Optimally finding dispatching groups has a naive runtime com-

plexity of $O(|\text{scheduled } \texttt{BlockSet}|^2)$. In a cluster processing millions of outstanding declarations, executing this search within a scheduling quantum becomes intractable — defeating the goal of dispatching to minimize cache space. To make dispatching tractable, we introduce an erasure-coding (EC) group based dispatcher which bounds the search space to a handful of `BlockSets` accessing a given set of EC groups. For every EC group in the system, we maintain a list of outstanding `BlockSets` accessing any subset of its blocks. In order to find the dispatching group for a scheduled `BlockSet`, DINGO identifies its EC-groups and searches for overlapping `BlockSets` exclusively within this metadata, lowering the complexity to $O(|\text{scheduled } \texttt{BlockSet}|)$.

Explicit cache management to ensure overlap. DINGO saves disk IO by assuming the cache will absorb multiple reads for the same block. However, we found that in practice this approach results in less saved IO than the IO Planner predicts since blocks can be evicted between reads or not placed in the cache at all. Additionally, DINGO causes multiple reads to the same data in a short period of time, potentially causing the cache to think the blocks are popular and useful to store.

To help ensure that scheduled free IO is actually saved and to avoid cache pollution, the IO Planner issues cache directives for scheduled blocks each dispatching round. We dedicate a small amount of cache space for DINGO. The cache places blocks contained in the cache directive in this reserved space upon a read of the block. To account for delays in maintenance task read requests, blocks are kept in this dedicated cache space for the entire dispatch round. In the rare event that the cache runs out of dedicated space, the cache applies its eviction policy to the reserved space.

We find that the cache space required by DINGO is minimal. The required capacity is primarily a function of (i) the effective IO rate after eliminating redundant reads and (ii) the time window over which blocks must remain pinned until application IO requests are actually issued following dispatch. For a maintenance IO rate of 2 MB/s per TB, even in the extreme scenario where no overlap is found, and conservatively assuming it takes one minute for applications to issue the corresponding IO after dispatch, the required cache allocation would be only 120 MB per TB of disk. If data blocks need to remain pinned to the cache for two minutes, we would still only need 240 MB of cache per TB of disk. This translates to a pessimistic upper bound of 240 TB of cache spread across many servers for a 1 EB storage cluster (Fig. 7).

6 Evaluation

We evaluate Declarative IO through both a small-scale prototype and a simulation of a datacenter-scale storage system. We find that for a standard set of HDFS maintenance tasks, DINGO reduces disk reads by 26%. With datacenter-scale workloads, these savings increase: DINGO decreases disk reads by 28-51%, enabling devices that are 1.7× larger. This section discusses experimental setup (Sec. 6.1), DINGO's IO savings (Sec. 6.2), an attribution of these IO savings (Sec. 6.3), and DINGO's sensitivity to both workload characteristics and IO availability (Sec. 6.4).

Task	Frequency	Deadline	Logical Bytes Read
Rebalancing	On node addition	14hr	750GB
Reconstruction	On node failure	3hr	4.9TB
Scrubbing	Every 4 days	81hr	18TB
Transcoding	Every 40 mins	3hr	160GB

Table 1: Maintenance task parameters for HDFS experiments. We run a 3/24 scaled down version in both time and capacity.

6.1 Experimental Setup

DINGO Prototype. We build a DINGO prototype on top of HDFS [61] due to HDFS being an open-source append-only distributed storage system like most deployed systems (Sec. 4.4). The IO Planner supports both block-level and file-level declarations. We evaluate DINGO against a version of HDFS with no modifications other than the addition of transcoding support (baseline). Both storage systems run on 10 SuperMicro 4042G-6RF nodes, each with 64 cores, 128 GB RAM and a 3TB HDD connected with 40 GbE InfiniBand. We limit both systems to 10 GB of memory per disk to better emulate the ratio of cache to disk in larger capacity systems with large SSD caches. We would limit the memory usage further to replicate smaller caches, but the HDFS Datanode process requires sufficient memory. Thus, our prototype uses 2 GB per TB of disk. The cluster starts with ∼80% capacity used (18TB of data with 25% in 3-way replication and 75% in RS-3-2-1024k). We measure disk IO with `iostat` [7].

The maintenance workload follows the distribution from hyperscaler A (Sec. 3.2), but isolated to four maintenance tasks pertinent to HDFS: rebalancing, reconstruction, scrubbing, and transcoding. We scale down both capacity and time by an eighth to reflect our cluster's 3TB HDDs compared to a 24TB datacenter HDD and to run for the full length of our longest declaration. The parameters of the scaled maintenance tasks are described in Table 1. We configure DINGO with a scheduling quota of 120 MB/s and run the experiment for 81 hours (i.e., a month at our experiment scale).

Datacenter Simulator. To evaluate IO savings at scale, we build a time-driven cluster storage simulator. It models IO activity from real-world datacenter maintenance workloads in discrete 3-hour steps over a 30-day period on a 100 PB cluster. The simulated cluster stores data in 7.5 GB files, distributed in 256 MB blocks [22, 61] and maintains 80% storage capacity utilization. Blocks are either 3-way replicated or part of 6-of-9 or 30-of-33 erasure-coded stripes [41, 42]. We assume an IO demand of 2.0 MBps/TB for maintenance reads and show how results vary with different demands in Sec. 6.4.

We simulate two cluster workloads, one based on each of the two hyperscalers described in Sec. 3.2 (exact workload breakdowns are in Fig. 17). We simulate eight workloads: reconstruction, scrubbing, garbage collection, transcoding,

Maintenance task	Event frequency	Deadline
Reconstruction	On disk failure	36 hours
Scrubbing	Once in 30 days	30 days
Garbage Collection	Once in 6 hours	5 days
File transcoding	Once in 12 hours	5 days
Rebalancing	Once in a day	5 days
Georeplication	Once in 30 days	30 days
File Scrubbing	Once in 30 days	30 days
FSCK	Once in 15 days	7 days

Table 2: Maintenance task parameters for simulator experiments.

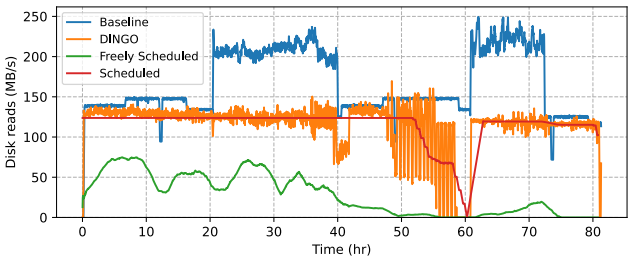


Fig. 14: Maintenance task disk reads for DINGO *(orange) and unmodified HDFS (blue). The red and green lines represent the logical IO scheduled by the IO Planner for* DINGO*, with the red line being IO that counts toward the disk IO limit and the green line being IO that is free due to overlap. For the baseline, periods of increased IO occur around hours 20 and 60 due to reconstruction events.*

FSCK, georeplication, and rebalancing.[11] Maintenance task frequency and deadlines are listed in Table 2. We validated our simulator, finding that its IO savings are within 5% of those measured in the analogous HDFS experiment.

We open-source both the HDFS prototype and the simulator at `www.github.com/Thesys-lab/declarativeIO-osdi26-artifact`.

6.2 DINGO enables higher capacity HDDs

We first explore DINGO's IO savings in our prototype and in our datacenter simulator.

Takeaway 1: *Our prototype shows* DINGO *works in a storage cluster, reducing bytes read by 26% for the four maintenance tasks.*

We evaluate the IO reduction benefits of DINGO in our storage cluster. Fig. 14 compares the maintenance disk reads in DINGO to that of unmodified HDFS (baseline). The baseline has to read a byte from disk per every logical byte needed by each maintenance task. By comparison, DINGO completes the maintenance workload while only reading 0.74 bytes from disk per logical byte, a 26% reduction in total bytes read.

Additionally, DINGO enables a steadier maximum rate of maintenance IO. While DINGO sometimes has more IO than scheduled due to cache misses, metadata operations, and other noise (scheduled IO savings are 28% compared to DINGO's observed 26%), DINGO is aware of all outstanding maintenance tasks and schedules their combined IO rate. In fact, DINGO's largest deviations from a constant rate only occur

[11] We describe how the workloads were simulated in detail in Suppl. A.2.

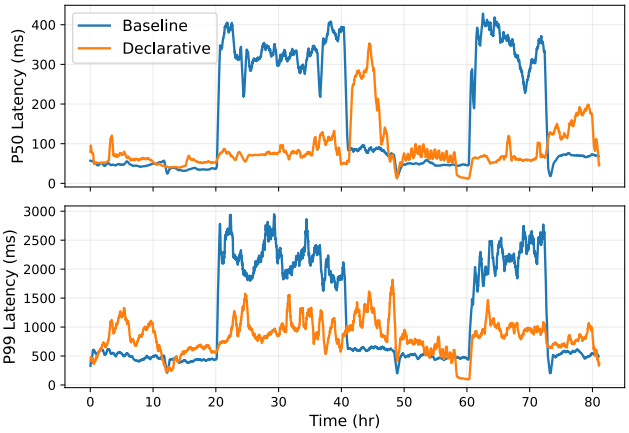


Fig. 15: Max of clients' P50 and P99 read latencies per minute over time. Ongoing maintenance IO impacts latency. For instance, reconstruction events around hours 20 and 60 increase latency in the baseline due to lack of a central rate limiter for maintenance read IO, and dense scheduling of transcoding events around hour 40 in DINGO *increases P50 latency due to elevated write IO.*

when it finishes most of its outstanding work ahead of schedule, such as around hour 50. At such points, disk reads oscillate based on when new declarations arrive.

With the imperative paradigm, even when each maintenance task rate limits itself independently, the combined IO shifts significantly depending on which tasks are running. For example, the baseline in Fig. 14 has periods of higher disk IO starting around hours 20 and 60 due to reconstruction.

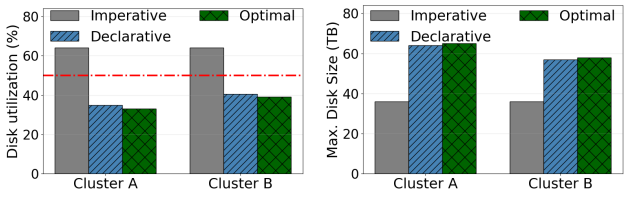
Takeaway 2: DINGO *has similar or better client P50 and P99 read latencies than the baseline.*

To determine whether DINGO affects client performance, we repeat the experiment with a read workload based on analysis of Alibaba block storage traces [73]. We configure this foreground workload to contribute around 30% of total disk read IO (Sec. 3.2). The average client throughput is 6.36 MB/s for the baseline and 6.82 MB/s for DINGO. As seen in Fig. 15, we find that the median client read latencies are better than the baseline both on average (86 vs 167 ms) and at worst (423 vs 587 ms). The 99th percentile has an even larger difference, 832 vs 1181 ms on average and 4969 vs 6877 ms at worst, due to its more stable read IO from maintenance tasks.

However, as seen in Fig. 15, for certain time intervals, the client latency differs significantly (up to about 6× lower or higher) due to increases in maintenance IO. The baseline's latency spikes when it is reconstructing data. Similarly, although DINGO respects the read bandwidth limit, bursts of write activity from clustered transcoding can cause temporary P50 read latency increases. Future work can improve robustness by integrating other metrics into scheduling (e.g. write IO, CPU and network utilization, etc.).

Takeaway 3: DINGO *enables 1.7× larger drive adoption on petabyte-scale clusters in our* DINGO *simulator.*

For a representative maintenance read demand of 2 MBp-



(a) Disk-head utilization. *(b) Max. Drive Sizes supported*

Fig. 16: (a) DINGO's IO savings in terms of disk-head utilization across two clusters with a cumulative maintenance IO demand of 2 MBps/TB and 48 TB drives. The red line represents the maximum disk utilization maintenance tasks can use and still allow deployment of 48 TB drives in a cluster where half of disk IO is from applications. (b) Maximum drive sizes that can be supported in the two clusters for a total demand of 4 MBps/TB, half of which is maintenance.

s/TB, Fig. 16a shows the percentage of disk head utilization required in a cluster with 48 TB drives. With this IO demand, the imperative approach consumes 64% of disk time to fulfill all reads for the maintenance workload. Declarative IO allows DINGO to identify up to 45% of bytes read as redundant and reduce disk utilization to as low as 34.8% in Cluster A.

DINGO's significant reductions to disk head utilization allow the adoption of higher capacity drives in storage clusters. Fig. 16b shows how Declarative IO enables 64TB drives in Cluster A (assuming maintenance reads can only use 50% of disk bandwidth), a 1.7× increase in the maximum supported drive size compared to an imperative system. Cluster B sees similar benefits with a 1.5× drive capacity increase to 58TB.

Takeaway 4: DINGO *achieves near-optimal IO reduction.*

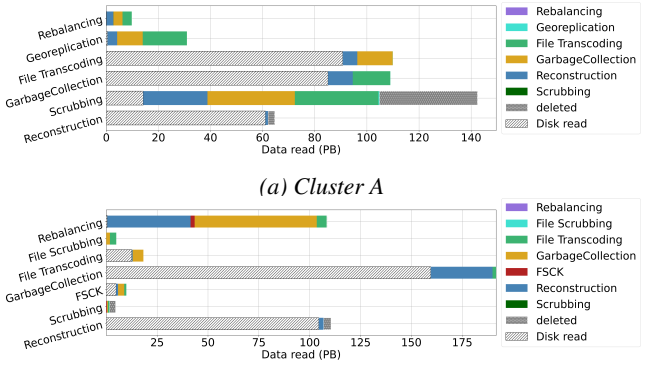
We also compare DINGO's savings against `Optimal`, an infeasible theoretical lower bound. `Optimal` knows all future declarations, does not consider per-quanta disk IO limitations, and does not consider deadlines. `Optimal` reads a block only when it is not in any future declarations to avoid missing IO reuse. Even under many more constraints, DINGO only incurs slightly more IO than `Optimal`. DINGO stays within 5% of this lower bound for both Cluster A and Cluster B.

Takeaway 5: *Even for exabyte storage clusters, the IO Planner can run on a single server.*

We assess the scalability of DINGO by measuring the time and memory overheads of the IO Planner in our simulator on a Dell PowerEdge R640 server with 240 GB of RAM. For a 1 EB storage cluster performing 2 MBps/TB of maintenance reads, our IO Planner was able to find all dispatching groups for a one-hour scheduling quanta in under 15 minutes with a peak memory usage of 205 GB. With more powerful and memory-rich compute nodes typically available in hyperscale environments, DINGO can be directly adopted for multi-exabyte storage clusters.

6.3 Where are DINGO's IO savings from?

Fig. 17 shows the breakdown of IO savings for the two simulated clusters. For each workload in each cluster, we report the amount of data read directly from disk, and the



(a) Cluster A

(b) Cluster B

Fig. 17: Volume of data read by different workloads across two clusters. White hatched bars represent data that had to be read directly from disk. The solid colored bars represent data that was read for free due to cache hits. The colors represent the workload with which these free bytes overlapped. The dotted gray bars represent data that was deleted (so the IO was avoided altogether).

amount of IO saved, either because of cache hits (the data was recently read by another task) or because the corresponding data was deleted, eliminating the need for IO. When tasks are scheduled with overlapping IO, we attribute disk reads to the tasks with the nearest deadlines as those are the first to decide which data to read in our scheduling heuristic (Sec. 5.1).

Takeaway 6: DINGO *allows more flexible tasks to piggyback on IO from more urgent and less-flexible tasks.*

Since reconstruction tasks have the shortest deadlines, reconstruction often drives the IO Planner to choose data in a scheduling quantum, though this changes when other declarations are close to their deadlines. Other tasks are then able to exploit these reads when overlapped by the IO Planner.

Another interesting example is rebalancing, which becomes completely free. While rebalancing has equal or worse time-flexibility than file transcoding, garbage collection, or index checking, it has much better data-flexibility. Rebalancing declares more blocks than it needs and only requires a single block at a time. This pattern greatly increases the odds of finding overlap with other tasks and shows why finding data-flexibility is important to saving IO in a declarative paradigm.

Takeaway 7: *Imperative storage systems often do unnecessary maintenance IO.*

In a traditional storage system with no additional information about the remaining lifetime of data blocks, scrubbing indiscriminately reads and verifies all data in the system. But DINGO's results reveal that a large portion of this work is unnecessary. Maintenance tasks such as garbage collection and file transcoding write new file contents and then delete the original file. By simply allowing a flexible time window to accomplish scrubbing, we find that ∼26% of reads declared for scrubbing were unnecessary because the data was subsequently deleted.

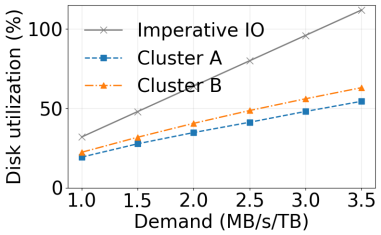


Fig. 18: Disk utilization by maintenance tasks with different demands. Utilization with imperative IO increases proportionally with demand. DINGO *is able to reduce the utilization by effectively scheduling tasks requesting common bytes together.*

6.4 Sensitivity studies

We now explore how DINGO responds to a variety of different situations — particularly when IO demand, task deadlines, and IO supply change.

Takeaway 8: DINGO *is effective across different maintenance IO demands.*

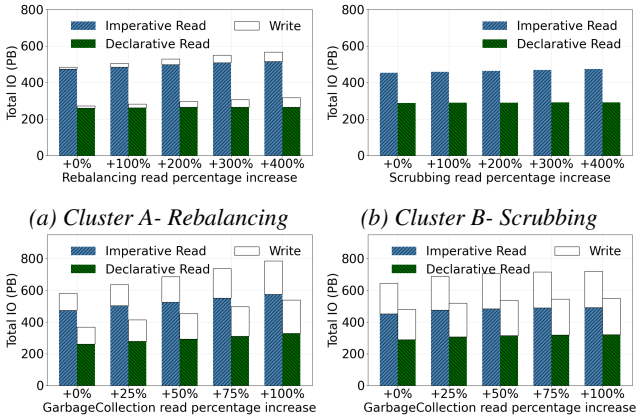
Depending on the nature of data stored and the users being serviced, the IO demand for maintenance reads can vary across clusters, even within a hyperscaler. Fig. 18 shows savings with a demand of maintenance reads ranging from 1 MBps/TB to 3.5 MBps/TB for two cluster workloads using 48 TB drives. To scale the IO demand, we increase the frequency of each maintenance task. For scrubbing, we also shorten the deadline since otherwise it would overlap with itself. Disk utilization grows proportionally with the requested rate of maintenance reads as expected.

Disks spend 22% of their time on maintenance reads at 1 MBps/TB, but exceed 90% utilization at 3 MBps/TB with imperative IO (we set imperative's IO demand to be the same for both clusters). If the demand exceeds 3 MBps/TB, it would be physically impossible for the disk to service even just the maintenance reads, leaving no room for foreground IO. Declarative IO reduces the disk-head time from disk by 39-51% in Cluster A and 28-46% in Cluster B, saving more at higher IO demands.

Takeaway 9: DINGO *enables each maintenance task to perform more work with less additional IO than imperative IO.*

Due to the lack of IO supply, different hyperscalers prioritize some maintenance workloads and throttle others based on their trade-offs. For example, many clusters dynamically adjust the rate of garbage collection based on when available capacity or IO is needed more. Fig. 19 illustrates the IO overheads incurred when the rate or the IO demand of maintenance tasks increases.

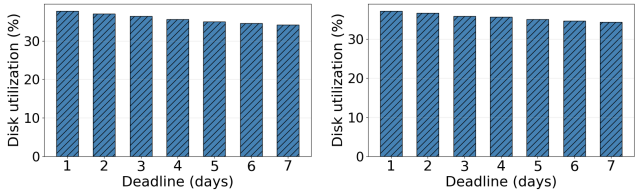
By leveraging data reuse, DINGO can perform additional maintenance work for substantially less IO than an imperative system. For example, Cluster B is able to scrub 4× more data with virtually no extra IO, whereas an imperative-only cluster performs proportionately more reads. DINGO can also perform more garbage collection and rebalancing by consuming less read bandwidth. Note that DINGO only reduces the bandwidth needed for *reading* data; maintenance tasks still need to write their data to disk. Despite this, in Cluster A, DINGO can perform 4× more rebalancing and still consume 45% less IO than the imperative baseline. Similarly, for both clusters, DINGO is able to perform 2× more garbage collection for the same IO as the imperative IO baseline. That is, if these clusters adopted declarative IO today, they could garbage collect twice as aggressively without any additional disk IO.



(a) Cluster A- Rebalancing *(b) Cluster B- Scrubbing*

(c) Cluster A- GC *(d) Cluster B- GC*

Fig. 19: Declarative IO allows for increasing maintenance tasks' work while incurring the same, or less, disk IO than the imperative baseline, reducing the need to throttle maintenance tasks. Total IO is from all maintenance IO in the cluster, not just that of the task being adjusted.



(a) Garbage Collection *(b) File transcoding*

Fig. 20: In cluster A, DINGO*'s disk utilization decreases as garbage collection and file transcoding increase their time-flexibility through extending their deadlines.*

Takeaway 10: *Exposing more time-flexibility with longer deadlines saves IO.*

We investigate how the time-flexibility of a workload, as defined by its completion deadline, impacts achievable savings with the IO Planner. Fig. 20a and Fig. 20b show IO savings in Cluster A as deadlines vary from 1 to 7 days for garbage collection and file transcoding. Longer deadlines allow DINGO to find more overlap for the declared tasks. For instance, moving from a 1-day deadline to a 7-day deadline for file transcoding resulted in ∼7% lower disk utilization in the cluster. Note that a realistic deadline will depend on the SLOs that the workload is required to meet. Depending on IO traffic, garbage collection tasks may mandate shorter deadlines to limit the accumulation of garbage, reduce storage overheads and improve IO performance.

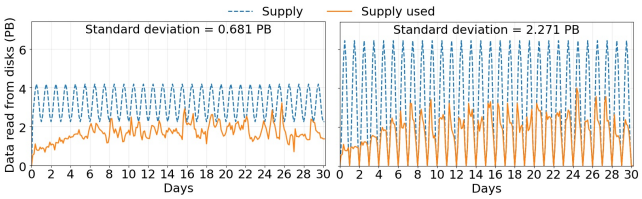


Fig. 21: Bytes read over a month by maintenance tasks with declarative IO following a diurnal IO supply. DINGO *rate-limits the IO schedule based on the quota allocated during both low and high supply variance.*

Takeaway 11: DINGO *adapts to changes in IO supply, such as those driven by foreground workload variations.*

Maintenance tasks are often throttled in order to prioritize user-facing application IO. One approach to throttling is to allocate a dynamic IO quota to each maintenance task. This IO supply typically follows a diurnal pattern, reflecting user demand throughout a day. DINGO is able to dynamically rate-limit the amount of scheduled maintenance work in order to stay within the allocated IO quota. Fig. 21 shows that the IO used by declarative IO is always below the allocated threshold, even with a high variance in IO supply.

7 Related Work

This section discusses additional related work with similar techniques and goals to Declarative IO.

Duet and Quartet. Duet [10] uses page events to inform local maintenance tasks about new in-cache data. Quartet [27] builds on Duet to schedule subtasks of map-reduce jobs with overlapping IO requests close in time to each other. DINGO instead introduces an interface that allows a large variety of different maintenance tasks to declare multiple types of IO flexibility across large distributed storage systems.

Disk optimizations. Disk-level optimizations can re-order IO. For instance, freeblock scheduling [46] re-orders low-priority IO to exploit disk-head time spent rotating to the right disk location. This can be extended to local maintenance reads [68]. These optimizations are orthogonal to Declarative IO since the maintenance reads will still go through these disk-level optimizations.

Semantically-smart disks [11, 64, 65] intuit the structure of the file system or database to make better data placement decisions or secure deletion of data, increasing disk functionality without changing the disk interface. Declarative IO both addresses a different problem — the impending IO-wall in distributed storage system on HDDs — and also takes a different approach — embracing interface change as necessary.

Database optimizations. The database community has tried to combine data requests, such as through shared scans [8, 12, 66]. These optimizations rely on having database queries — a higher-level abstract than many maintenance tasks operate at. Declarative IO instead targets the file and block level since all maintenance tasks that perform disk IO must eventually go through the bulk storage tier. Thus, we need an interface specifically at that tier and we find that we do not need one as verbose for databases.

Reducing IO from individual maintenance tasks. There has been work trying to reduce IO from individual maintenance tasks such as transcoding [13, 42]. This work is orthogonal. Any improvements in reducing this IO will just contribute to the end goal of overcoming the IO wall and deploying higher-capacity hard-disk drives.

8 Conclusion and Discussion

We show that HDDs are running into an IO wall. Because application IO is cacheable and maintenance IO is not, we find that maintenance IO now dominates disk IO demand. We introduce Declarative IO to allow maintenance tasks to express their IO flexibility to the storage system. DINGO uses this flexibility to find reuse between maintenance tasks, enabling deployment of 1.7x larger HDDs.

Declarative IO has broader implications than just reducing disk IO to save cost, emissions, and footprint from using larger-capacity devices. Datacenters subject to bursty application workloads or dynamic power constraints (e.g., power grid limits) [17] now have a powerful tool to throttle cluster-wide maintenance IO. Using less IO supply for maintenance tasks also allows more low-priority opportunistic workloads to run, since there is more likely a surplus in available IO even with larger-capacity devices. Finally, while we target maintenance tasks, there are many higher-level applications, such as batch analytics or machine learning training, where Declarative IO could help further reduce disk IO.

9 Acknowledgements

We thank our shepherd Erci Xu and the anonymous reviewers for their invaluable feedback. We extend special thanks to Larry Greenfield, Kim Keeton, and Richard McDougall at Google; Rohit Bhat, Dario Marasco, Cornel Rat, and Venkatraghavan Srinivasan at Meta; and Ricardo Bianchini at Microsoft for helpful discussions and insights into maintenance IO in hyperscalers. This research is generously supported in part by NSF grants CNS-2402838, CNS1956271, CNS1901410, CAREER 1943409, CCF-2403195, IIS-2322974, as well as a Google Junior Faculty Award, Sloan Foundation Fellowship, VMware Systems Research Award, and a NDSEG Fellowship. We also thank the members and companies of the PDL consortium (Amazon, Bloomberg, Datadog, Google, Honda, Intel, Jane Street, LayerZero Research, Meta, Microsoft, Oracle, Oracle Cloud Infrastructure, Everpure, Salesforce, Samsung, Western Digital) for their interests, insights, feedback, and support.

References

[1] How Lasers Could Unlock Hard Drives With 10 Times More Data Storage. https://www.popularmechanics.com/technology/a20078/heating-magnets-lasers-could-be-the-key-magnetic-recording/.

[2] Leveldb. https://github.com/google/leveldb.

[3] Rocksdb. http://rocksdb.org.

[4] Seagate: HAMR is nailing it – no looming 20TB to 30TB capacity problem. https://blocksandfiles.com/2021/09/24/seagate-hamr-on-course-no-looming-20-to-30tb-capacity-problem/.

[5] Seagate Reveals HAMR HDD Roadmap: 32TB First, 40TB Follows. https://www.tomshardware.com/news/seagate-reveals-hamr-roadmap-32-tb-comes-first.

[6] Heat assisted marnetic recording HAMR. https://www.seagate.com/innovation/hamr/, 2024.

[7] iostat(1) - linux manual page. https://man7.org/linux/man-pages/man1/iostat.1.html, 2025.

[8] Parag Agrawal, Daniel Kifer, and Christopher Olston. Scheduling shared scans of large data files. *Proc. VLDB Endow.*, 1(1):958–969, August 2008.

[9] Tyler Akidau, Robert Bradshaw, Craig Chambers, Slava Chernyak, Rafael J Fernández-Moctezuma, Reuven Lax, Sam McVeety, Daniel Mills, Frances Perry, Eric Schmidt, et al. The dataflow model: a practical approach to balancing correctness, latency, and cost in massive-scale, unbounded, out-of-order data processing. *Proceedings of the VLDB Endowment*, 8(12):1792–1803, 2015.

[10] George Amvrosiadis, Angela Demke Brown, and Ashvin Goel. Opportunistic storage maintenance. In *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP '15, page 457–473, New York, NY, USA, 2015. Association for Computing Machinery.

[11] Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, Lakshmi N. Bairavasundaram, Timothy E. Denehy, Florentina I. Popovici, Vijayan Prabhakaran, and Muthian Sivathanu. Semantically-smart disk systems: past, present, and future. *SIGMETRICS Perform. Eval. Rev.*, 33(4):29–35, March 2006.

[12] Subi Arumugam, Alin Dobra, Christopher M. Jermaine, Niketan Pansare, and Luis Perez. The datapath system: a data-centric analytic processing engine for large data warehouses. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, SIGMOD '10, page 519–530, New York, NY, USA, 2010. Association for Computing Machinery.

[13] Sanjith Athlur, Timothy Kim, Saurabh Kadekodi, Francisco Maturana, Xavier Ramos, Arif Merchant, KV Rashmi, and Gregory R Ganger. Okapi: Decoupling data striping and redundancy grouping in cluster file systems. In *19th USENIX Symposium on Operating Systems Design and Implementation (OSDI 25)*, pages 897–914, 2025.

[14] David F. Bacon, Nathan Bales, Nico Bruno, Brian F. Cooper, Adam Dickinson, Andrew Fikes, Campbell Fraser, Andrey Gubarev, Milind Joshi, Eugene Kogan, Alexander Lloyd, Sergey Melnik, Rajesh Rao, David Shue, Christopher Taylor, Marcel van der Holst, and Dale Woodford. Spanner: Becoming a sql system. In *Proceedings of the 2017 ACM International Conference on Management of Data*, SIGMOD '17, page 331–343, New York, NY, USA, 2017. Association for Computing Machinery.

[15] Michael A Bender, Alex Conway, Martín Farach-Colton, William Jannen, Yizheng Jiao, Rob Johnson, Eric Knorr, Sara McAllister, Nirjhar Mukherjee, Prashant Pandey, et al. Small refinements to the dam can have big consequences for data-structure design. In *The 31st ACM Symposium on Parallelism in Algorithms and Architectures*, pages 265–274, 2019.

[16] Benjamin Berg, Daniel S. Berger, Sara McAllister, Isaac Grosof, Sathya Gunasekar, Jimmy Lu, Michael Uhlar, Jim Carrig, Nathan Beckmann, Mor Harchol-Balter, and Gregory G. Ganger. The CacheLib caching engine: Design and experiences at scale. In *USENIX OSDI*, 2020.

[17] Ricardo Bianchini, Christian Belady, and Anand Sivasubramaniam. Data center power and energy management: Past, present, and future. *IEEE Micro*, 44(5):30–36, 2024.

[18] Jose A. Blakeley, Per-Ake Larson, and Frank Wm Tompa. Efficiently updating materialized views. In *Proceedings of the 1986 ACM SIGMOD International Conference on Management of Data*, SIGMOD '86, page 61–71, New York, NY, USA, 1986. Association for Computing Machinery.

[19] Netflix Technology Blog. Application data caching using ssds. https://netflixtechblog.com/application-data-caching-using-ssds-5bf25df851ef, 2016.

[20] Nicolas Bruno and Surajit Chaudhuri. Exploiting statistics on query expressions for optimization. In *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data*, SIGMOD '02, page 263–274, New York, NY, USA, 2002. Association for Computing Machinery.

[21] Maximilian Böther, Xiaozhe Yao, Tolga Kerimoglu, Dan Graur, Viktor Gsteiger, and Ana Klimovic. Mixtera: A data plane for foundation model training, 2025.

[22] Brad Calder, Ju Wang, Aaron Ogus, Niranjan Nilakantan, Arild Skjolsvold, Sam McKelvie, Yikang Xu, Shashwat Srivastav, Jiesheng Wu, Huseyin Simitci, et al. Windows azure storage: a highly available cloud storage service with strong consistency. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, pages 143–157, 2011.

[23] Badrish Chandramouli, Guna Prasaad, Donald Kossmann, Justin Levandoski, James Hunter, and Mike Barnett. Faster: an embedded concurrent key-value store for state management. *VLDB*, 2018.

[24] Gohar Irfan Chaudhry, Esha Choukse, Íñigo Goiri, Rodrigo Fonseca, Adam Belay, and Ricardo Bianchini. Towards resource-efficient compound ai systems. In *Proceedings of the 2025 Workshop on Hot Topics in Operating Systems*, HotOS '25, page 218–224, New York, NY, USA, 2025. Association for Computing Machinery.

[25] Peter M Chen, Edward K Lee, Garth A Gibson, Randy H Katz, and David A Patterson. Raid: High-performance, reliable secondary storage. *ACM Computing Surveys (CSUR)*, 26(2):145–185, 1994.

[26] Carson Molder Sathya Gunasekar Jimmy Lu Snehal Khandkar Abhinav Sharma Daniel S. Berger Nathan Beckmann Greg Ganger Daniel Lin-Kit Wong, Hao Wu. Baleen: Ml admission and prefetching for flash caches. In *FAST*, 2024.

[27] Francis Deslauriers, Peter McCormick, George Amvrosiadis, Ashvin Goel, and Angela Demke Brown. Quartet: harmonizing task scheduling and caching for cluster computing. In *Proceedings of the 8th USENIX Conference on Hot Topics in Storage and File Systems*, HotStorage'16, page 1–5, USA, 2016. USENIX Association.

[28] Western Digital. Shingled magnetic recording (smr) hdd technology, October 2022. White Paper.

[29] David Domingo and Sudarsun Kannan. pFSCK: Accelerating file system checking and repair for modern storage. In *19th USENIX Conference on File and Storage Technologies (FAST 21)*, pages 113–126. USENIX Association, February 2021.

[30] Gil Einziger, Roy Friedman, and Ben Manes. Tinylfu: A highly efficient cache admission policy. volume 13, New York, NY, USA, November 2017. Association for Computing Machinery.

[31] Apache Software Foundation. HDFS Erasure Coding. `https://hadoop.apache.org/docs/r3.0.0/hadoop-project-dist/hadoop-hdfs/HDFSErasureCoding.html`, 2017.

[32] S. Ghemawat, H. Gobioff, and S.T. Leung. The Google File System. In *ACM SIGOPS Operating Systems Review*, volume 37, pages 29–43. ACM, 2003.

[33] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The google file system. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles*, 2003.

[34] Garth A. Gibson. *Redundant Disk Arrays: Reliable, Parallel Secondary Storage*. PhD thesis, EECS Department, University of California, Berkeley, 12 1990.

[35] Ashish Gupta, Inderpal Singh Mumick, et al. Maintenance of materialized views: Problems, techniques, and applications. *IEEE Data Eng. Bull.*, 18(2):3–18, 1995.

[36] Cheng Huang, Huseyin Simitci, Yikang Xu, Aaron Ogus, Brad Calder, Parikshit Gopalan, Jin Li, Sergey Yekhanin, et al. Erasure Coding in Windows Azure Storage. In *ACM Trans. Storage*, 2012.

[37] Lexiang Huang, Matthew Magnusson, Abishek Bangalore Muralikrishna, Salman Estyak, Rebecca Isaacs, Abutalib Aghayev, Timothy Zhu, and Aleksey Charapko. Metastable failures in the wild. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 73–90, Carlsbad, CA, July 2022. USENIX Association.

[38] Sai Huang, Qingsong Wei, Dan Feng, Jianxi Chen, and Cheng Chen. Improving flash-based disk cache with lazy adaptive replacement. volume 12, New York, NY, USA, February 2016. Association for Computing Machinery.

[39] Ilias Iliadis, Robert Haas, Xiao-Yu Hu, and Evangelos Eleftheriou. Disk scrubbing versus intra-disk redundancy for high-reliability raid storage systems. *ACM SIGMETRICS Performance Evaluation Review*, 36(1):241–252, 2008.

[40] Saurabh Kadekodi, Francisco Maturana, Suhas Jayaram Subramanya, Juncheng Yang, KV Rashmi, and Gregory Ganger. Pacemaker: Avoiding heart attacks in storage clusters with disk-adaptive redundancy. In *Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation*, 2020.

[41] Saurabh Kadekodi, K V Rashmi, and Gregory R Ganger. Cluster storage systems gotta have HeART: improving storage efficiency by exploiting disk-reliability heterogeneity. 2019.

[42] Timothy Kim, Sanjith Athlur, Saurabh Kadekodi, Francisco Maturana, Dax Delvira, Arif Merchant, Gregory R. Ganger, and K. V. Rashmi. Morph: Efficient file-lifetime redundancy management for cluster file systems. In *Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles*, SOSP '24, page 330–346, New York, NY, USA, 2024. Association for Computing Machinery.

[43] Philip L Lehman and S Bing Yao. Efficient locking for concurrent operations on b-trees. *ACM Transactions on Database Systems (TODS)*, 6(4):650–670, 1981.

[44] Qiang Li, Qiao Xiang, Yuxin Wang, Haohao Song, Ridi Wen, Wenhui Yao, Yuanyuan Dong, Shuqi Zhao, Shuo Huang, Zhaosheng Zhu, Huayong Wang, Shanyang Liu, Lulu Chen, Zhiwu Wu, Haonan Qiu, Derui Liu, Gexiao Tian, Chao Han, Shaozong Liu, Yaohui Wu, Zicheng Luo, Yuchao Shao, Junping Wu, Zheng Cao, Zhongjie Wu, Jiaji Zhu, Jinbo Wu, Jiwu Shu, and Jiesheng Wu. More than capacity: Performance-oriented evolution of pangu in alibaba. In *21st USENIX Conference on File and Storage Technologies (FAST 23)*, pages 331–346, Santa Clara, CA, February 2023. USENIX Association.

[45] Lanyue Lu, Thanumalayan Sankaranarayana Pillai, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Wisckey: Separating keys from values in ssd-conscious storage. In *USENIX FAST*, 2016.

[46] Christopher R. Lumb, Jiri Schindler, Gregory R. Ganger, and David F. Nagle. Towards hiugher disk head utilization: Extracting free bandwidth from busy disk drives. In *Proceedings of the 4th Symposium on Operating Systems Design and Implementation*, OSDI '00, USA, 2000. USENIX Association.

[47] Ao Ma, Charlotte Dragga, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. ffsck: The fast file system checker. In *11th USENIX Conference on File and Storage Technologies (FAST 13)*, pages 1–15, San Jose, CA, February 2013. USENIX Association.

[48] Yannis Manolopoulos, Yannis Theodoridis, and Vassilis Tsotras. *Advanced database indexing*, volume 17. Springer Science & Business Media, 2012.

[49] Sara McAllister, Benjamin Berg, Julian Tutuncu-Macias, Juncheng Yang, Sathya Gunasekar, Jimmy Lu, Daniel S. Berger, Nathan Beckmann, and Gregory R. Ganger. Kangaroo: Caching billions of tiny objects on flash. In *ACM SOSP*, 2021.

[50] Sara McAllister, Yucong "Sherry" Wang, Benjamin Berg, Daniel S. Berger, George Amvrosiadis, Nathan Beckmann, and Gregory R. Ganger. FairyWREN: A sustainable cache for emerging Write-Read-Erase flash interfaces. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, pages 745–764, Santa Clara, CA, July 2024. USENIX Association.

[51] Marshall Kirk McKusick, Willian N Joy, Samuel J Leffler, and Robert S Fabry. 'fsck-the unix file system check program. *Unix System Manager's Manual-4.3 BSD Virtual VAX-11 Version*, 1986.

[52] Rishiyur S Nikhil et al. Executing a program on the mit tagged-token dataflow architecture. *IEEE Transactions on computers*, 39(3):300–318, 2002.

[53] Alina Oprea and Ari Juels. A Clean-Slate Look at Disk Scrubbing. 2010.

[54] James O'Toole and Liuba Shrira. Opportunistic log: Efficient installation reads in a reliable storage server. In *USENIX OSDI*, 1994.

[55] Satadru Pan, Theano Stavrinos, Yunqiao Zhang, Atul Sikaria, Pavel Zakharov, Abhinav Sharma, Mike Shuey, Richard Wareing, Monika Gangapuram, Guanglei Cao, et al. Facebook's tectonic filesystem: Efficiency from exascale. In *19th USENIX Conference on File and Storage Technologies (FAST 21)*, pages 217–231, 2021.

[56] Phitchaya Mangpo Phothilimthana, Saurabh Kadekodi, Soroush Ghodrati, Selene Moon, and Martin Maas. Thesios: Synthesizing accurate counterfactual i/o traces from i/o samples. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, ASPLOS '24, page 1016–1032, New York, NY, USA, 2024. Association for Computing Machinery.

[57] Chris Ruemmler and John Wilkes. An introduction to disk drive modeling. *Computer*, 27(3):17–28, 1994.

[58] Thomas JE Schwarz, Qin Xin, Ethan L Miller, Darrell DE Long, Andy Hospodor, and Spencer Ng. Disk scrubbing in large archival storage systems. In *The IEEE Computer Society's 12th Annual International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunications Systems, 2004.(MASCOTS 2004). Proceedings.*, pages 409–418. IEEE, 2004.

[59] Anton Shilov. Seagate's HAMR Update: 32 TB in Early 2024, 40+ TB Two Years Later. `https://www.anandtech.com/show/21125/seagates-hamr-update-32-tb-in-early-2024-40-tb-two-years-later`, 2023.

[60] Anton Shilov. 60tb hard drives arriving in 2028 according to industry roadmap — hdd capacity forecast to double in four years. https://www.tomshardware.com/pc-components/storage/60tb-hard-drives-arriving-in-2028-according-to-industry-roadmap-hdd-capacity-forecast-to-double-in-four-years, 2024.

[61] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. The hadoop distributed file system. In *2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*, pages 1–10, 2010.

[62] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, Robert Chansler, et al. The Hadoop distributed file system. 2010.

[63] Michael Sindelar, Ramesh K. Sitaraman, and Prashant Shenoy. Sharing-aware algorithms for virtual machine colocation. In *Proceedings of the Twenty-Third Annual ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '11, page 367–378, New York, NY, USA, 2011. Association for Computing Machinery.

[64] Muthian Sivathanu, Lakshmi N. Bairavasundaram, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Database-aware semantically-smart storage. In *Proceedings of the 4th Conference on USENIX Conference on File and Storage Technologies - Volume 4*, FAST'05, page 18, USA, 2005. USENIX Association.

[65] Muthian Sivathanu, Vijayan Prabhakaran, Florentina I. Popovici, Timothy E. Denehy, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Semantically-smart disk systems. In *Proceedings of the 2nd USENIX Conference on File and Storage Technologies*, FAST '03, page 73–88, USA, 2003. USENIX Association.

[66] Craig A.N. Soules, Kimberly Keeton, and Charles B. Morrey. Scan-lite: enterprise-wide analysis on the cheap. In *Proceedings of the 4th ACM European Conference on Computer Systems*, EuroSys '09, page 117–130, New York, NY, USA, 2009. Association for Computing Machinery.

[67] Yair Sovran, Russell Power, Marcos K. Aguilera, and Jinyang Li. Transactional storage for geo-replicated systems. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, page 385–400, New York, NY, USA, 2011. Association for Computing Machinery.

[68] Eno Thereska, Jiri Schindler, John Bucy, Brandon Salmon, Christopher R. Lumb, and Gregory R. Ganger. A framework for building unobtrusive disk maintenance applications. In *Proceedings of the 3rd USENIX Conference on File and Storage Technologies*, FAST '04, page 213–226, USA, 2004. USENIX Association.

[69] Kaushik Veeraraghavan, Justin Meza, Scott Michelson, Sankaralingam Panneerselvam, Alex Gyori, David Chou, Sonia Margulis, Daniel Obenshain, Shruti Padmanabha, Ashish Shah, Yee Jiun Song, and Tianyin Xu. Maelstrom: Mitigating datacenter-level disasters by draining interdependent traffic safely and efficiently. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 373–389, Carlsbad, CA, October 2018. USENIX Association.

[70] Sage A. Weil, Scott A. Brandt, Ethan L. Miller, Darrell D. E. Long, and Carlos Maltzahn. Ceph: A scalable, High-Performance distributed file system. In *7th USENIX Symposium on Operating Systems Design and Implementation (OSDI 06)*. USENIX Association, 2006.

[71] Tzu-Wei Yang, Seth Pollen, Mustafa Uysal, Arif Merchant, and Homer Wolfmeister. CacheSack: Admission Optimization for Google Datacenter Flash Caches. page 17.

[72] Yunzhao Yang, Runhui Wang, Xuanqing Liu, Adit Krishnan, Yefan Tao, Yuqian Deng, Kuangyou Yao, Peiyuan Sun, Henrik Johnson, Davor Golac, et al. Declarative data pipeline for large scale ml services. *arXiv preprint arXiv:2508.15105*, 2025.

[73] Ying Zhang, Zhenyuan Ruan, Yiying Zhang, Yu Hua, Ming Zhao, Youtao Zhang, and Mengjia Yan. Characterizing, modeling, and benchmarking i/o workload of a production cloud scale block storage system. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP)*, pages 763–778. ACM, 2021.

Umap: Revisiting Memory-mapped I/O on Distributed File Systems for Efficient Matrix Access (Operational Systems)

Yongchao He[†] *Unaffiliated*
Guangyan Zhang[*] *Tsinghua University*
Zane Cao *ScitiX AI*
Wenfei Wu[*] *Peking University*

Abstract

Our production experience in large-scale data-processing pipelines shows that, despite its attractive programming model, memory-mapped I/O (*mmap-IO*) on distributed file systems (DFS) exhibits severe operational pathologies—livelocks during write-heavy phases, chronically low multi-threaded throughput, and high memory footprints that may trigger out-of-memory kills in containerized environments. Our measurements further show that mmap-IO on DFS is 3×–10× slower than on local file systems (LFS) for matrix random-access workloads, primarily because page-granularity network I/O underutilizes high-speed networks and deferred write-back behavior incurs expensive distributed flushes and metadata operations. We present *umap*, a DFS-agnostic runtime that delivers near-in-memory matrix-access performance on DFS via network-efficient communication, a concurrency-aware cache protocol with linear scalability, and lazy-expansion cache management. Deployed in production for over 18 months, *umap* has eliminated livelocks and out-of-memory-induced job failures while improving throughput by up to 6.7× across diverse matrix-access workloads. Our experience shows that rethinking mmap-IO's interaction with DFS is essential for robust, predictable performance in modern large-scale clusters.

1 Introduction

Matrix access underlies many modern data center workloads, including machine learning (ML) [27,66], quantitative finance pipelines such as large-scale backtesting [31,47], LLM inference [40,71], and scientific computing [15,18]. For example, in quantitative finance, matrices encode per-asset state over long historical windows and are intensively accessed in offline backtesting and factor-research pipelines; In serverless LLM inference, matrices hold model parameters that must be repeatedly loaded into CPU memory (e.g., via *mmap*) and transferred to GPUs [29,65]. As these workloads grow, matrix access increasingly dominates overall cost and can even exceed the FLOP cost itself [51,72].

To balance performance and scalability, practitioners commonly adopt *file-backed matrices* (FBM), implemented via *mmap-IO*[1]. FBM offers a compelling abstraction: data appears memory-resident, while the kernel transparently fetches pages on demand. This abstraction is commonly used to enable out-of-core access without loading entire datasets into memory, which is essential when working with matrices that exceed physical memory capacity. This model is widely adopted in practice and is supported by widely used systems. For example, NumPy provides `memmap` for out-of-core array access [13], PyTorch data pipelines commonly use disk-backed and memory-mapped loading mechanisms to stream large training datasets [17], and systems such as vLLM use mmap to load multi-GB model weights into CPU memory during initialization [20]. Across these systems, FBM is often the primary mechanism for accessing datasets and model parameters that exceed memory capacity, rather than an optional optimization.

Yet our large-scale deployment reveals that this long-standing abstraction *breaks down* under today's disaggregated storage architectures. Modern clusters increasingly adopt high-performance DFS [9, 22, 38, 45] to separate compute from storage, enabling independent scaling and higher aggregate bandwidth. This shift fundamentally disrupts assumptions baked into the virtual memory (VM) subsystem.

Abstraction mismatch under disaggregated storage. *mmap-IO* was designed for local, low-latency storage and page-granularity VM, whereas DFSes export block-oriented semantics with distributed metadata, locking, and cache coherence. This structural mismatch causes each page fault to incur fragmented remote I/O, heavy metadata traffic, and cross-node synchronization. In our production cluster that supports large-scale financial backtesting, scientific computing, and

[†] Work done while at ScitiX AI.
[*] Corresponding author.

[1] We use *mmap-IO* to denote the full Linux memory-mapped I/O subsystem, including the `mmap` syscall, page cache, page fault path, and filesystem integration.

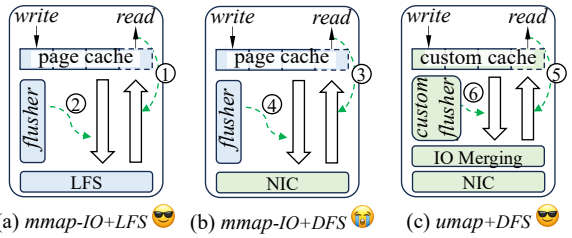
AI workloads, we observed substantial FBM slowdown after migrating from direct-attached storage to a disaggregated architecture (i.e., replacing LFS with DFS). Despite the DFS providing over *25 GB/s* of per-node remote bandwidth—an order of magnitude higher than a local SSD—FBM workloads still suffer *3×–10× lower throughput*, poor bandwidth utilization, and extreme tail latency. Increasing page granularity via hugepages further exacerbates amplification for irregular access patterns [50]. These findings highlight a deeper issue: the traditional VM loading model no longer aligns with the disaggregated storage substrate.

Implicit VM "magic" undermines predictability and operations. A more serious operational issue arises from *mmap-IO*'s *implicit* I/O behavior. On LFS, deferred write-back and on-demand paging are fast and predictable. On DFS, these mechanisms become opaque and failure-prone: asynchronous write-back competes for distributed locks, triggering prolonged stalls; page faults inflate into multi-millisecond network operations; blocked threads accumulate in `iowait`, and under pressure they may be misclassified as deadlocked. In our production cluster, such misbehavior caused tasks to remain idle for tens of minutes, cascade into *livelock*, and even trigger system-level miskill events—directly affecting SLAs and service availability. The core problem is not merely low performance, but the absence of observability and control: performance anomalies disappear into the kernel's page-fault machinery, hidden from operators, turning performance debugging into a black-box guessing game.

Greedy caching harms multi-tenant stability. Finally, the kernel's default caching policy—designed for single-node, single-tenant environments—does not translate to multi-tenant clusters. Linux aggressively expands the page cache to retain remote data, with little regard for reuse or isolation. For FBM workloads on DFS, this leads to memory pressure, noisy-neighbor interference, and occasional out-of-memory (OOM) kills, even when applications exhibit modest working sets. In contrast, our measurements show that an application-aware, lazy-expansion strategy can achieve higher throughput while using *less than 10%* of the memory consumed by vanilla *mmap-IO*. This highlights a growing need for user-level control over caching behavior in shared clusters.

Recent work aims to accelerate *mmap-IO* on fast local storage through prefetching [30, 41, 44, 52], per-core caches [54], and access pattern-specific optimizations [46, 56, 57]. In the era of disaggregated storage, where bandwidth is plentiful but metadata and consistency dominate, these assumptions collapse: their optimizations amplify page-granular I/O, stalls, and livelock. FBM workloads therefore need a new abstraction that preserves memory-style access while exposing coarse-grained, explicit control for predictable high performance on DFS.

This paper presents *umap*, a high-performance, drop-in compatible runtime for FBM on DFS. *umap* addresses the abstraction mismatch and operational challenges above without requiring changes to applications or DFSes. It introduces: (1) a network-friendly communication manager that merges page-level requests into DFS-native transfers; (2) a concurrency-aware, lock-minimizing cache protocol that adapts to parallel random access; and (3) a lazy-expansion cache manager that enforces predictable memory usage in multi-tenant settings.



(a) *mmap-IO+LFS* (b) *mmap-IO+DFS* (c) *umap+DFS*

Figure 1: Comparison of mmap and umap runtimes on LFS and DFS. Page-granularity network I/O and deferred write-back significantly degrade "mmap-IO+DFS" performance.

Deployed for 18 months in production, *umap* eliminates livelock and writeback-induced stalls observed with *mmap-IO*, stabilizes tail latency, and improves availability. Experiments show that *umap* achieves up to 2.8× faster reads and 8.3× faster writes than state-of-the-art (SOTA) NVMe-backed *mmap-IO*, and accelerates real-world training, inference, scientific, and quantitative workloads by up to 6.7×.

2 Motivation, Challenges and Solutions

2.1 Background

***mmap-IO* enabled file-backed matrix (FBM).** A FBM stores data on the file system while providing transparent in-memory access via *mmap()* [10]. The example below maps a file into the process's address space, exposes it as a memory buffer, and accesses it as a $ROWs \times COLs$ matrix.

```
1 float* mat = mmap(... /*fd=*/open("~/mat.dat"));
2 for (size_t i = 0; i < ROWs; ++i)
3   for (size_t j = 0; j < COLs; ++j)
4     mat[i * COLS + j] = i + j;
```

Figure 1(a) illustrates how *mmap-IO* constructs the FBM *mat* on LFS. It maps the file from LFS into the process's virtual memory, allowing applications to access file as if it were in RAM. Upon a page fault (①), the OS loads the data into the *page cache* [16] via IO bus, potentially evicting existing pages (dirty pages are flushed to the LFS by a *deferred, asynchronous* flusher thread (②)) if memory is limited.

Unlike memory-backed matrices that consume physical RAM and risk OOM errors, FBMs utilize the file system and *mmap-IO* to access datasets beyond physical memory efficiently and scalably. Compared to direct I/O, which bypasses the page cache and requires manual management, *mmap-IO* benefits from OS-level caching and demand paging, reducing redundant I/O and improving performance.

FBM behavior changes in LFS-to-DFS transition. In traditional direct-attached storage architecture, storage is physically tied to compute nodes (e.g., LFS in Figure 1(a)), whereas disaggregated storage architectures consolidate storage into DFS, accessible via dedicated storage networks (Figure 1(b)). Unlike LFS, which relies on local storage devices (e.g., SSDs), DFS distributes data across multiple *storage nodes* to support concurrent access by many clients. Despite data distribution across nodes, DFS retains the same API as LFS, enabling compute nodes to access files as local. As a result, migrating from LFS to DFS does not change *mmap-IO*'s behavior; the primary difference is that *mmap-IO* now issues network I/O via NIC to access DFS (Figure 1(b)).

2.2 Performance Issues of *mmap-IO* on DFS

Key characteristics of FBM-based workloads. FBM-based workloads present unique challenges that influence both application performance and deployment stability:

C1: Random access patterns. Depending on the algorithm, threads may access matrix elements sequentially or randomly. Random access reduces spatial locality and can significantly increase I/O overhead, especially on DFS, directly impacting user-perceived performance.

C2: High concurrency. FBM applications often use multiple threads, each accessing different rows or columns. While enabling parallelism, such concurrency can strain file systems when threads generate overlapping I/O.

C3: Large scale. FBMs often exceed physical memory, making *mmap-IO* essential to scale workloads beyond RAM limits. However, this can lead to frequent page faults and high memory pressure, potentially threatening deployment stability—especially in containerized environments where excessive memory use may trigger out-of-memory kills.

These characteristics reveal that *mmap-IO* is not an ideal intermediary between applications and DFS, due to a fundamental mismatch in access patterns, which is further amplified by optimizations designed for LFS. We next validate these observations through experiments.

1) Storage underutilization in random small I/O. To better understand the limitations of *mmap-IO* as an intermediary for FBM workloads, we first measure single-threaded random access performance of *mmap-IO*-based FBM on a LFS (ext4 [19] over an Optane NVMe SSD [7]) and on two DFSes (GPFS [5] and IBM NFSv4 [12]). Using `fio` [4], we benchmark FBM by varying request size to observe its effect, and also measure random *direct I/O* performance—where reads and writes bypass the page cache—for reference.

Figure 2 presents the results (lines without the "*mmap-IO*" prefix indicate direct I/O). On LFS, *mmap-IO* and direct I/O achieve similar throughput, while on DFS, *mmap-IO* performs significantly worse, with the gap widening as request size increases: for example, DFS throughput triples from 4KB to 64KB I/Os under direct I/O, whereas *mmap-IO* remains flat.

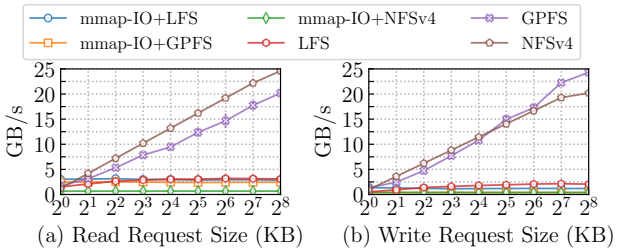


Figure 2: Single-threaded performance of direct I/O versus memory-mapped I/O (setup in §6.1).

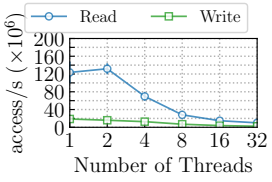


Figure 3: Concurrent random access to a *single* FBM stored on a **DFS** (GPFS).

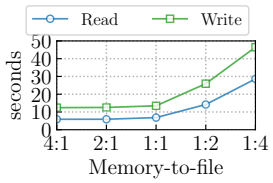


Figure 4: Concurrent random access to *multiple* FBMs stored on an **LFS**.

The reasons for *mmap-IO*'s poor performance are twofold. On LFS (Figure 1(a)), reads and writes go directly over the I/O bus to the LFS, allowing efficient access with minimal overhead. On DFS (Figure 1(b)), each *mmap-IO* operation is fragmented into 4KB network I/Os and incurs additional latency from RPCs, metadata lookups, and locks—orders of magnitude higher than LFS access. *Write* throughput suffers even more, as the Linux *flusher* thread performs deferred, asynchronous write-backs (④), limiting effective throughput.

2) Non-scalable I/O throughput in multi-threaded scenarios. Figure 3 extends the workload from Figure 2 by scaling from single-threaded to multi-threaded execution. The results on GPFS and NFSv4 are similar; for simplicity, we only present those on GPFS. As shown, increasing thread count leads to worse performance. Kernel profiling attributes this to growing time in *iowait*—mainly queued for distributed locks—and in spin locks within the system call *native_queue_spin_lock_slowpath* in Linux. Moreover, as the thread count grows, time spent in both *lock* and *iowait* increases. At 32 threads, the task spends 88.9% of its time in *iowait*, and 76.1% of the remaining time waiting on locks.

The scalability of *mmap-IO* does not align with the I/O performance gains offered by DFS. Two primary factors limit *mmap-IO*'s scalability on DFS: (i) One is that the spin lock (e.g., *tree_lock* in Linux) is used to enforce the atomic page replacement. Consequently, concurrent access to the memory-mapped region incurs three primary overheads, ranked in descending order: file I/O (>4000 clock cycles in LFS [7]), lock overhead (750∼2500 clock cycles [11]), and memory access

(144∼192 clock cycles [6, 8]). These overheads can overlap, so the actual performance is determined by the slowest among them. After migrating to DFS, using the RDMA-enabled network reduces file I/O overhead to levels close to memory access. However, this improvement shifts the bottleneck to lock overhead, which becomes dominant in data access performance. (ii) Another is that an increase in concurrent threads leads *mmap-IO* to generate numerous parallel I/O requests to DFS, escalating contention for distributed locks and exacerbating performance degradation.

3) Performance impact of excessive caching in *mmap-IO*. *mmap-IO* employs a *greedy-expansion* strategy, aggressively caching file data into memory to minimize access latency. While effective for moderate workloads, this approach becomes counterproductive in data-intensive FBM workloads—particularly with many FBMs—where the total mapped size exceeds available memory. The resulting memory pressure leads to performance degradation.

To validate this, we run experiments on a node with memory restricted to 64GB using *cgroup*, with other configurations as described in §6.1. Each test consists of multiple concurrent tasks, where each task uses *fio* [4] to perform 4 million single-threaded random accesses to a 16GB FBM (*float32*). We vary the number of concurrent tasks from 1 to 16, resulting in total mapped sizes from 16GB to 256GB—corresponding to memory-to-file size ratios of 4:1 to 1:4. As shown in Figure 4, where the y-axis denotes the completion time of each test, performance remains stable while the data fits in memory but degrades sharply once it exceeds capacity, highlighting the cost of excessive caching under memory pressure.

2.3 Challenges and Solutions

The slowdown arises not from flaws in *mmap-IO* or DFS individually, but from a fundamental semantic mismatch between them. *mmap-IO* assumes a local, page-granular backing store and issues demand-driven 4 KB page faults. DFS, however, provides remote, block-granular storage, where each access traverses network stacks and metadata services. Consequently, page faults fragment even large sequential accesses into thousands of tiny remote I/Os, amplifying metadata contention and serialization, which explains the severe bandwidth underutilization and long-tail stalls observed under concurrency.

To address these limitations, the core is to design DFS-friendly cache and write-back mechanisms. Figure 1(c) illustrates the key idea behind *umap*: reshaping implicit page-level access into explicit, network-efficient operations to restore system predictability and observability, while preserving the *mmap-IO* abstraction and high-performance. Achieving this requires overcoming the following challenges.

Challenge 1: Small-I/O amplification prevents DFS from reaching full bandwidth. *mmap-IO* generates bursty streams of 4 KB requests, which are inefficient for DFS. Larger I/Os (e.g., through hugepages) reduce metadata pressure but introduce I/O amplification [50], wasting network bandwidth.

Solution 1: *umap* preserves the 4 KB page size but introduces *network-friendly communication management* that merges page requests using a rank-based, time-aware policy and fairly schedules them across I/O channels. This reduces request volume and mitigates network burstiness and overhead. (§4.1)

Challenge 2: Lock contention limits scalability in multi-threaded workloads. *mmap-IO* relies on locks to ensure atomic cache replacement [10, 54]. In data-intensive multicore workloads, frequent replacements trigger high lock costs—e.g., acquiring a lock costs ∼2500 cycles vs. 144–192 cycles to load a page [11]. While negligible on LFS due to slow disk access [7], these costs dominate with fast DFS networks (200 Gbps [38]), capping throughput.

Solution 2: *umap* employs a *concurrency-aware cache protocol* that detects runtime access parallelism and uses a state-machine–based mechanism to perform lock-free cache lookups and replacements. This enables near-linear scalability and efficient random access. (§4.2)

Challenge 3: Unbounded cache growth inflates memory usage in multi-job environments. Existing memory-mapping systems [10, 54] rely on *greedy expansion*, where page caches grow opportunistically until memory pressure forces eviction. This reduces memory available for applications and exacerbates interference across concurrent jobs. However, limiting cache size risks increased cache misses and costly DFS I/O.

Solution 3: *umap* introduces *lazy-expansion cache management*, which dynamically adjusts cache size based on observed access patterns. It prioritizes replacement over expansion, estimates runtime memory demand for FBM workloads, and reclaims memory when pressure rises—achieving near-peak performance with substantially lower memory usage. (§4.3)

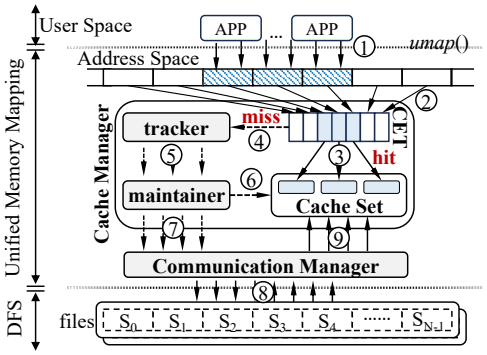


Figure 5: *umap* architecture. Files are divided into cache-block–sized segments and mapped on demand.

3 Overview of *umap*

umap is a memory mapping runtime that constructs a full-stack I/O path over DFS. By replacing implicit OS paging with user-space orchestration, it bridges application-level memory access and low-level data movement while ensur-

ing predictable performance and fault observability. From the user's perspective, *umap* provides a drop-in replacement interface, *umap()*, which is compatible with *mmap()* and returns a memory buffer that supports direct read/write operations. As a result, applications written using the *mmap* APIs can be easily migrated to the *umap* APIs with minimal modifications.

Architecture. As illustrated in Figure 5, *umap* comprises two main components: the *Cache Manager (CaM)* and the *Communication Manager (CoM)*. The CaM manages two core data structures. The *Cache Set* is a global pool of fixed-size cache blocks (CB) that temporarily store file segments fetched from the DFS. The *Cache Entry Table (CET)* is a collection of index arrays, each corresponding to a mapped file. CaM also includes two submodules: the *tracker*, which makes decisions about cache allocation and replacement based on reuse policies, and the *maintainer*, a background thread that enacts the *tracker's* decisions by allocating new CBs or selecting existing ones to load data. Mapping a new file with *umap()* just adds a new index array to the CET. This array holds α pointers, where α equals the file size divided by the CB size. Each entry represents a file segment and indicates whether it has been loaded. In our configuration (4 bytes per pointer and a 4 KB CB size), the memory overhead is approximately 1/1024 of the file size. The CoM merges I/O requests, manages concurrency, and performs asynchronous data transfers between the DFS and the local cache set.

Execution flow. Once initialized, *umap* intercepts memory accesses from applications (① in Figure 5). Given a virtual address within a mapped file, it computes the corresponding segment index and accesses the associated entry in CET's index array. If the entry points to a valid CB, it indicates a *cache hit*, and the data is returned directly from the cache set (②, ③). Otherwise, a *cache miss* occurs (④): the *tracker* selects a CB (⑤) —potentially evicting an existing one—and the *maintainer* prepares it for reuse (⑥ ⑦). The CoM then asynchronously fetches the required segment from the DFS (⑧ ⑨). Once the data is loaded, the CET entry is updated to reference the new CB, allowing execution to resume.

umap prioritizes throughput over per-access latency. By batching and merging requests, it may increase individual access latency, but significantly improves overall system throughput under concurrent workloads. This tradeoff is aligned with the characteristics of the target workloads.

Consistency model. *umap* follows a mmap-style weak consistency model with explicit synchronization. Each node maintains a local cache and does not enforce implicit cross-node coherence. This design targets data-parallel workloads where cross-node concurrent writes are rare, and avoids the excessive coordination overhead that arises from enforcing strong consistency at page granularity in distributed file systems.

4 Design

4.1 Network-friendly I/O Management

The communication manager (CoM) transforms fine-grained page faults into coarse-grained I/O requests. It achieves this by merging adjacent requests and batching them through a queueing mechanism, thereby improving network efficiency and reducing I/O amplification.

I/O Merging and Prioritization. A natural way to improve network throughput is to merge consecutive requests that are contiguous on DFS. While random matrix accesses exhibit spatial locality, naïve merging only captures strictly adjacent requests, leaving much of the locality unused. Caching requests can create additional merging opportunities, but delays may underutilize the network.

To this end, we design a *Push-In Admission-Out (PIAO)* queue to facilitate I/O request merging. Elements in the PIAO queue are ordered by a unique *rank*. When an element with rank r is inserted, it is placed at position i such that r is greater than the rank of the $(i-1)_{th}$ element and less than that of the $(i+1)_{th}$ element. Adjacent elements with consecutive ranks are then merged into a single, larger network I/O request.

Merging via PIAO may starve higher-rank, earlier-arriving requests. To prevent this, each request is also enqueued into a *FIFO* queue. When a merge occurs in PIAO, the corresponding request in FIFO is updated rather than enqueued again. Dequeueing always begins with the FIFO queue; the CoM retrieves the request from FIFO and removes the corresponding elements from PIAO.

umap associates a pair of PIAO and FIFO queues with each mapped file. At runtime, each request is tagged with a tuple $(timestamp, rank)$ (index of the file segment is used as *rank*) and sent to the CoM. Figure 6 gives an example. The CoM receives two requests with ranks r_8 and r_4 at times t_5 and t_6, respectively. The request (t_5, r_8) is inserted at the tail of both the PIAO queue and the FIFO queue because r_8 is greater than the ranks of all existing requests, and t_5 is later than all existing timestamps. In contrast, the request (t_6, r_4) can be merged with (t_1, r_5) and (t_4, r_6), so it is inserted at the third position in the PIAO queue. Meanwhile, the CoM updates the merged I/O request in the FIFO queue. During dequeueing, the request $(t_1, r_4 - r_6)$ will first be removed from the FIFO queue. Meanwhile, the corresponding requests (t_6, r_4), (t_1, r_5), and (t_4, r_6) will also be removed from the PIAO queue.

Communication Scheduling. Even with I/O merging, single-threaded communication may underutilize DFS bandwidth [34]. To address this, CoM leverages modern multi-queue NICs [35], which handle packets in parallel to sustain line-rate throughput. By establishing multiple I/O channels and distributing FIFO requests across them, CoM improves network utilization.

To ensure fair scheduling of each FIFO queue, the CoM utilizes a *least-first* mechanism to select a queue. At runtime,

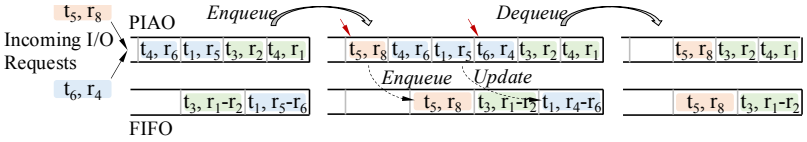


Figure 6: Example of I/O merging and prioritization.

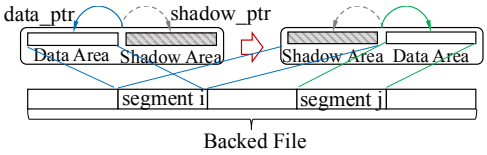


Figure 7: The shadow-copy mechanism.

it maintains a *min-heap* to manage all FIFO queues, and employs T_r to track the transmitted data amount of the heap's root node, while each FIFO queue uses T_q to record its own transmitted data amount. The heap starts empty with $T_r = 0$. When a queue is added to the heap, its T_q is initialized to T_r. All I/O channels proactively fetch requests from this heap. Upon fetching the root node, if the corresponding queue is not empty, the CoM dequeues a request from the queue, updates T_q ($T_q \leftarrow T_q +$ request_size) and T_r accordingly ($T_r \leftarrow T_q$). Otherwise, the queue will be removed from the heap.

Data Synchronization. Accessing a DFS involves distributed locking and coordination across multiple computing nodes [24, 60, 67], introducing significantly higher overhead than LFS. To address this, CoM introduces a *shadow-copy* mechanism [34] that overlap communication with computation. Each CB in CoM maintains two equal-sized buffers (Figure 7): one serving as the *data* buffer for application access and the other as the *shadow* buffer for background synchronization. At runtime, CoM alternates the roles of the two buffers, enabling continuous access while synchronization proceeds in parallel.

(1) Data fetching: Whenever a CB is recycled for caching another segment, CoM fetches data from the corresponding file segment to the CB's data area. Usually, only one data fetching operation occurs for each CB. Before a write-back, regardless of whether other threads are mapped to this CB, data fetching will not be triggered again.

(2) Write back: Figure 7 outlines *umap*'s write-back process. When a CB's data area changes association with the underlying file segment, *umap* initiates several steps. First, it removes the mapping between the CB and index array entry. Then, it atomically swaps the pointers *data_ptr* and *shadow_ptr*, respectively pointing to the data area and shadow area. Subsequently, an I/O request is generated to flush the shadow area back to the file (i_{th} segment). CoM uses atomic state transitions to ensure exclusive access during write-back, preventing *Write-after-Read (WAR)* hazards. Concurrently, it repopulates the CB's data area from the j_{th} segment.

4.2 Concurrency-Aware Cache Protocol

The concurrency-aware cache protocol determines the system's concurrency in real-time and sets CBs to different states based on thread access behavior. By leveraging the system's thread scheduling characteristics, it partitions the CBs into different sets, achieving lockless cache replacement and lockless data access, thus scaling the performance linearly.

4.2.1 Thread Tracking for Concurrency Awareness

Using locks to enforce data consistency when multiple threads access the same CB would hinder scalable throughput in memory-mapped I/O. Instead, CaM detects the concurrency degree at runtime and distinguishes threads accessing the same data, preventing one thread from preempting resources used by another. This ensures correctness and performance without locking, avoiding the following WAR hazards.

Consider a *false eviction* scenario with threads t_1 and t_2 accessing a mapped file concurrently. Thread t_1 accesses a loaded CB at the 10th entry E[10], while t_2 accesses an unloaded CB via E[100], initially pointing to a *sentinel CB* (a special marker with a NULL data area). Before t_1 returns, a cache replacement updates E[100] to the CB previously at E[10] and sets E[10] to the sentinel CB. Consequently, t_1 would encounter a segmentation fault when accessing a NULL data area.

A naive approach to prevent WAR hazard is to use a lock (e.g., *tree_lock* in *mmap-IO*) to protect cache replacement process on each access. With this method, in the aforementioned example, t_1 will update the status of CaM before the function returns, preventing thread t_2 from mistakenly setting E[10] to the sentinel CB during cache eviction. However, this approach is inefficient: even a single Compare-and-Swap (CAS) costs 15–30 cycles without contention and up to 2500 cycles under contention. Accessing an element with locks can thus cost hundreds of cycles. In contrast, *umap* requires only ∼12 cycles per access (§6.3).

The challenge arises because CaM aims to enhance throughput with limited cache capacity (§4.3). Unlike *mmap-IO*, which aggressively allocates new pages on cache misses and may pause threads during page faults, CaM reuses existing CBs to avoid thread suspension and reduce overhead.

In the concurrency-aware method, CaM attaches a unique identifier (*tid*) to each running thread to identify the current concurrency degree. This is accomplished by allocating a thread-local storage variable using the C keyword *__thread*. The *tid* is initialized when a thread triggers a cache replacement in *umap* for the first time, which is typically the first time it accesses data. Additionally, as shown in Figure 8(a), each CB has a reference map *rmap* to record which threads are referring to it. For a file, the *rmap* of CBs is non-overlapping. This means when a thread triggers a cache replacement, the *tid* should be removed from the *rmap* of the thread's previous CB and added to the *rmap* of its newly assigned CB. A CB with a non-empty *rmap* will never be evicted, thus avoiding

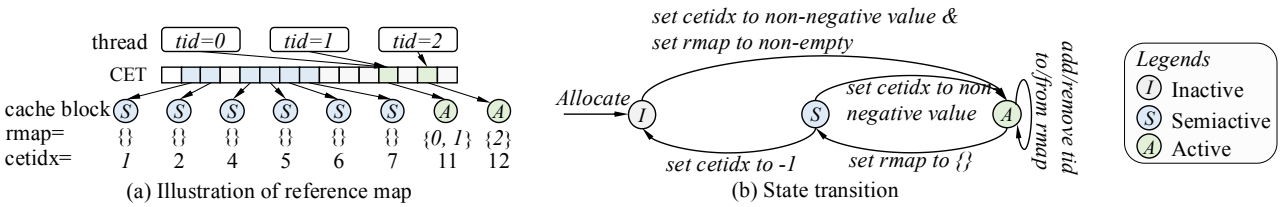


Figure 8: Overview of the concurrency-aware mechanism and its state machine.

false eviction. The reference counting [36] is not applicable here because a CB's lifetime may exceed that of a thread. A corner case is that when a thread terminates, it cannot modify the state of the last CB it accessed, which would result in some CBs never being released; *umap* solves this issue by checking the status of *rmap*'s associated threads to determine whether a CB can be released.

4.2.2 FSA-Based Cache Management Protocol

With only the *concurrency-aware* mechanism, multi-threaded random access causes *cache thrashing* [42,61,69,70], degrading performance. Assuming zero lock overhead, let t_a be the cost to access a CB element and t_r the cost of a replacement ($t_a \ll t_r$). For a task with x accesses and y replacements, the total overhead is $xt_a + yt_r$, dominated by y since x is fixed. If a CB is replaced once every k accesses on average, $y = x/k$, so higher randomness (smaller k) makes yt_r the dominant cost. Frequent loading and eviction of CBs also increase network overhead, reducing throughput and causing unpredictable performance.

The CaM maintainer employs a finite-state machine (FSA) to implement the concurrency-aware cache protocol to manage CBs. This protocol enables CBs of a file to transition between specific states without requiring data synchronization and triggering cache replacement, thereby eliminating the cache thrashing problem. Combined with *rmap* in the concurrency-aware mechanism, each normal CB has three states (Figure 8(b)):

(1) *Active*: referred to by threads and pointed to by a CET index array. CaM will return data directly upon accessing the corresponding entries. This allows CaM to bypass the cache replacement phase, enhancing performance by reducing unnecessary cache operations.

(2) *Semiactive*: not referred to by any thread but still pointed to by some index array entries. When accessing the corresponding entries, CaM returns data directly. These CBs are ready to be evicted to the inactive list but can also be re-activated.

(3) *Inactive*: not referred to by any thread and not pointed to by any index array entry. When a cache miss occurs, CaM picks one from the inactive list, sets it to active, binds it to a entry, and returns it. Additionally, newly allocated CBs are also added to the inactive list. This approach ensures efficient reuse of CBs, reducing the overhead of frequent allocations and evictions while maintaining performance.

Figure 8(b) illustrates the workflow of the FSA-based protocol. We elaborate on the state transition as follows:

Case 1 (Ⓘ → Ⓐ): An uninitialized CET index array entry is accessed, and then *umap* initializes an inactive CB, loads data and updates CB's metadata.

Case 2 (Ⓢ → Ⓐ): An initialized but not referred CB is accessed, and then *umap* just updates metadata.

Case 3 (Ⓐ → Ⓐ): If a referred CB is accessed by a new thread, or if a thread associated with the CB is detected to be dead or is accessing another entry, and then *umap* just updates the metadata.

Case 4 (Ⓐ → Ⓢ): All threads associated with this CB have either been detected dead or are accessing other CBs.

Case 5 (Ⓢ → Ⓘ): When inactive CBs are insufficient, *umap* flushes data to the backed file, sets the associated CET entry uninitialized, and converts a semiactive CB into inactive.

In CaM, inactive CB is used for cache replacement, and other states are used for data access. Since no data synchronization is involved when the semiactive and active states transition and CB has been initialized, the entire process is lockless. And the number of cache replacements y can be significantly reduced. Moreover, since the CB that can be used for cache replacement has been placed in the inactive state, there is no need to worry that cache replacement will evict the CB being used. Thus, we can use lockless techniques (e.g., concurrency queue [49]) to retrieve a CB from the inactive CB list and associate it with the target index array entry in the cache replacement process.

Except for case 5 (Ⓢ → Ⓘ), other state transition conditions are clear. CaM minimizes the number of inactive CBs to ensure that as many CBs as possible are used for caching data while ensuring that a CB being set to inactive is not accessed by other threads before the status modification is completed. Since the cost of state switching is less than that of associating a CB to an index array entry, and the number of threads scheduled for execution does not exceed the number of CPU logical cores (denoted as N_{core}), the number of simultaneous accesses in the inactive list is less than N_{core}. Therefore, CaM only needs to trigger Ⓢ → Ⓘ once when the number of CBs in the inactive list is less than N_{core}.

CaM's cache protocol can use cache efficiently. For **sequential access**, semi-active CBs are reused along the path of Ⓢ → Ⓘ → Ⓐ → Ⓢ → ⋯ without need to allocate new CBs. For workloads with repeated random access [53], semi-active CBs are reactivated to reduce data synchronization overhead.

Algorithm 1: Lazy-expansion cache management

```
1  Interface: CB* get_CB(idx)
2  Stateful Variables: counter, lru_list, lru_map,
     last_appearance
3  if last_appearance == NULL then
4      last_appearance ← (int*) malloc(4 × ⌈S_f / S_c⌉)
5      Initialize all elements in last_appearance to -1
6  N_c' ← 1
7  if last_appearance[idx] ≠ -1 then
8      N_c' ← counter - last_appearance[idx]
9  if lru_map.find(idx) ≠ lru_map.end() then
10     CB* cb ← get_CB_pointer(lru_map.find(idx))
11     lru_list.erase(lru_map.find(idx))
12     lru_list.push_front((cb))
13 else
14     if N_c' ≥ N_c then
15         Allocate N_new CBs and add them to inactive_list
16         N_c ← N_c + N_new
17     if inactive_list.size() > 0 then
18         lru_list.push_front(get_CB_pointer(inactive_list.front()))
19         inactive_list.pop_front()
20     else
21         CB* cb ← get_CB_pointer(lru_list.back())
22         lru_list.pop_back()
23         lru_list.push_front(cb)
24 last_appearance[idx] ← counter++
25 lru_map[idx] ← lru_list.front()
26 return get_CB_pointer(lru_list.front())
```

4.3 Lazy-Expansion Cache Management

The *Lazy-expansion Cache Management* module (tracker in Figure 5) decides whether to allocate a new CB or reuse an existing one on a cache miss. Allocating more CBs can improve performance, but mapping too many files increases pressure on the kernel's memory subsystem [53], potentially causing burst I/O [26] and degraded performance in multi-tenant environments (§2.2). To mitigate this, the tracker minimizes the number of CBs N_c of a mapped file while maximizing the cache hit ratio.

Obviously, N_c depends on the user's access pattern and must be determined dynamically. Accesses may be sequential or random, but due to spatial locality, data accessed within a short period is usually nearby in storage. Given an access sequence $X = \{\cdots, x_i, \cdots\}$, we define the degree of randomness at the i-th element as $R_i = n - i$, where $n > i$, *and* $x_i = x_n$ and $\forall j \in (i,n), x_j \neq x_i$. We then have the following result.

Theorem 1 $N_c = \max(R_i)$, $i \in \mathbb{N}$, is the optimal number of CBs for minimizing the cache capacity in an Least Recently Used (LRU) cache.

Proof. From the definition, $R_i \leq N_c$ for the i-th element e, meaning the distance to its next occurrence cannot exceed N_c. In an LRU cache with a capacity of N_c, when an element is added to the cache, it is placed at the head of the LRU list. In the worst case, subsequent evictions will push it one position back in the LRU list until it is removed. Thus, the element will remain in the cache for at least $N_c - 1$ evictions. Given that $R_i \leq N_c$, the element e will be accessed again and moved back to the head of the LRU cache before it is evicted, specifically during the R_i-th cache replacement. This process repeats indefinitely. Consequently, each data segment is loaded into the cache only once during its first access. Subsequent accesses to the same data will not result in a cache miss. □

Lazy-expansion: umap develops a variant of LRU cache to achieve the lazy-expansion of the capacity at runtime. Algorithm 1 shows the process to find optimal N_c. Based on Theorem 1, N_c should be set to be $max(R_i)$, $i \in [1, I]$ when processing I_{th} element. When processing the next element, there are three cases

- cache hit: then update the state of LRU cache and return (line 9-12).
- cache miss: If the element has never appeared, perform a regular LRU replacement (line 21-23).
- (*expansion*) cache miss: Otherwise, let i denote the index of the last appearance, expand the LRU cache to $N_c' = I + 1 - i$, and allocate $N_c' - N_c$ new CBs. Then update N_c and associate the element with a new CB (line 7-8 & line 14-19).

Note that in line 15-16 of Algorithm 1, when new CBs need to be allocated, *umap* will allocate N_{new} CBs instead of $N_c' - N_c$. This ensures the robustness of the algorithm by preventing elements with low frequency but large R_i from causing excessive CB allocations. With this algorithm, for file scanning workloads [46] where each element is accessed only once, such as in training [59], only one CB is needed regardless of the dataset size. For random-access workloads [53] that may repeatedly access the same elements, the memory required for the cache is minimized to achieve optimal performance.

Dynamic Cache Reclamation: The cache tracker maintains a virtual capacity N_v for each file, allowing the cache to shrink dynamically. Whenever N_c is updated, N_v is set to $N_c/2$, and Algorithm 1 is executed to update N_v without triggering actual allocation or replacement. Under memory pressure signaled by the Linux *kswapd* thread, the tracker instructs the cache maintainer to free $N_c - N_v$ CBs. Dynamic cache reclamation is needed because N_c is optimal only for the previous access sequence and may be too large for future accesses. If a smaller capacity can be sustained without cache misses, memory can be safely released.

5 Implementation

Seamless Integration. *umap* is implemented as a drop-in replacement for legacy `mmap`. We supply a lightweight shim library supporting two integration modes. (1) Source-

compatible mode: Applications can redirect `mmap` to *umap* through a small wrapper or macro-level substitution, requiring minimal code changes (see Appendix B of the supplementary material). (2) Binary-compatible mode: For closed-source or large distributed frameworks, *umap* can be injected via standard LD_PRELOAD–based interposition, intercepting libc `mmap` to transparently accelerate file-backed mappings on DFS paths. This design enables existing workloads to adopt disaggregated storage with low friction while preserving their original architecture.

Consistency model and operational semantics. Rather than enforcing cross-node coherence through distributed locking, *umap* adopts the conventional mmap-style consistency model where updates are cached locally and synchronized only when explicitly requested. We intentionally avoid implicit strong coherence across nodes for overlapping writes, which can cause false-sharing storms in distributed settings. This design follows common practice in distributed shared-memory systems and favors predictable performance for data-parallel workloads, where cross-writer conflicts are rare.

umap further improves robustness and performance through the following enhancements.

(1) DFS-agnostic implementation. *umap* abstracts away the differences between specific DFS implementations by focusing only on their common characteristics—such as relying on network I/O. As a result, it adopts a DFS-agnostic design that interacts with the file system through POSIX APIs, capturing the generic behavior of DFS. This approach ensures that *umap* remains compatible with any POSIX-compliant DFS without requiring system-specific adaptations.

(2) Optimize the branch hit rate for the fast path. There is a special *Sentry* CB, a globally unique single CB node whose status is always inactive. In practice, *umap* uses this node as the default pointer for the CET entry. Setting this node reduces the number of operations when the cache hits in the fast path. With this node, in the case of cache hits, *umap* needs only two steps: (i) check whether the CB's state is non-inactive and (ii) return the result. Without this node, *umap* would need an additional step to verify whether the CET entry is NULL before performing the two steps.

6 Performance Evaluation

Our evaluation tries to answer the following questions: (1) How much speedup does *umap* achieve in real-world workloads (§6.2)? (2) How does *umap* perform under various settings (§6.3)? (3) In *umap*, how much benefit does each key technique bring (§6.4)? (4) What is the cost of *umap* 's performance gains in terms of system overhead (§6.5)?

6.1 Methodology

Testbed. We evaluate *umap* on a cluster where each node has two 200 Gbps ConnectX-6 NICs (one for storage traffic and one for compute), 1.82 TB DRAM, dual Intel® Xeon® 8260 CPUs (128 logical cores), a 3.84 TB Optane NVMe SSD [7], and eight NVIDIA A100 GPUs. We compare two configurations that are identical except for the DFS backend: one uses GPFS [5] and the other uses NFSv4 [12]. Both DFS deployments run as shared services across the cluster.

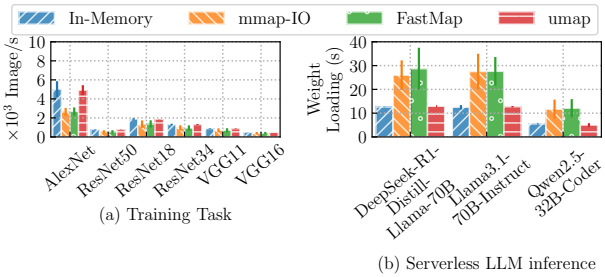


Figure 9: Impact of the FBM backend on AI workload performance.

Baselines. We use *mmap-IO* and FastMap [2, 54] over the SSD-backed LFS as baselines. *mmap-IO* is the de facto memory-mapped I/O mechanism in production systems, and FastMap is the leading academic design. Both are evaluated using their default configurations. We omit *mmap-IO*/FastMap on DFS because their performance is orders of magnitude lower, making them impractical for comparison.

Workloads. We evaluate four classes of FBM-based workloads, each with distinct access patterns and runtime behaviors. Unless otherwise stated, all real workloads run on GPFS; both GPFS and NFSv4 are used to evaluate *umap*'s DFS-agnostic design.

(i) AI workloads, where *PyTorch* and vLLM [40] use *umap*, *mmap-IO*, FastMap, or in-memory variants to load large FBMs—either as training datasets or LLM model weights. These workloads stress large-granularity, partially sequential accesses.

(ii) Cache-intensive scientific computing, represented by *OpenBLAS* [15], where we run all six FBM-backed matrix kernels using their default parallel configuration. These kernels repeatedly access small, high-locality regions of FBMs, placing pressure primarily on the cache manager rather than the DFS.

(iii) I/O-intensive financial analytics, represented by *backtrader* [1], where we run offline backtesting on 2020–2025 historical stock data across multiple markets. This workload scans multi-GB FBMs with low spatial locality, placing pressure primarily on sustained read/write throughput.

(iv) Microbenchmarks. We map 1–128 GB FBMs and use fio [4] with 1–32 threads to generate controlled random, sequential, and strided accesses.

Metrics. FBM workloads are mostly offline; we measure throughput and Job Completion Time (JCT) to capture large-matrix processing efficiency rather than per-request latency.

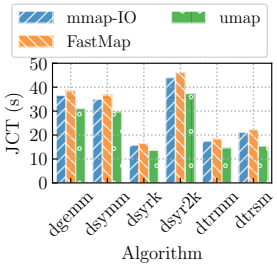


Figure 10: JCT comparison across different memory-mapped I/O runtimes.

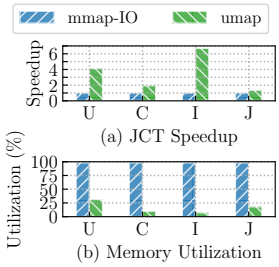


Figure 11: Improved JCT and lower memory usage achieved by *umap*.

6.2 Performance on Real-world Workloads

Performance in AI workloads. We evaluate the impact of different PyTorch [23] *Dataset* loaders on training performance: an in-memory loader, *mmap-IO*/FastMap loaders on LFS-backed FBMs, and a *umap* loader on DFS. To isolate FBM access, we preprocess ImageNet into a matrix-based format, allowing samples to be fetched without decompression overhead. We train AlexNet [39], ResNet [33], and VGG [63] using up to eight GPUs on a single node (Figure 9(a)). Although computation dominates end-to-end time, *umap* still provides 1.2–1.9× speedups. FastMap performs similarly to *mmap-IO* because ImageNet samples are relatively large (200–300 KB) and accessed single-threadedly, limiting the benefit of its lock-free parallel I/O optimizations.

In serverless LLM inference, model initialization time directly impacts request latency. Figure 9(b) shows the weight-loading time when vLLM [40] loads various LLMs via different backends, each implemented as a dedicated *model loader*. *umap* achieves near–in-memory performance and outperforms *mmap-IO* by 2.3×. LLM weight loading issues much larger I/O requests (often several megabytes per read) and uses only 1–8 threads, producing an access pattern that favors *umap* 's DFS-aware caching while reducing the benefit of FastMap's multi-threaded optimizations.

Performance in scientific computing. We further evaluate *umap* on cache-intensive workloads characterized by frequent reuse of cached FBM data. Specifically, we benchmark OpenBLAS [15] on six preset matrix routines using synthetic matrices of size $2^{15} \times 2^{15}$ (float64), filled with random values. On LFS, FBMs are accessed via *mmap-IO* and FastMap; on DFS, they are served by *umap*.

Across all benchmarks, *umap* outperforms *mmap-IO* and FastMap by 13%–28% (Figure 10). This is expected for cache-intensive workloads: OpenBLAS matrix operations are $O(n^3)$ and dominated by computation with frequent cache reuse, leaving little room for I/O optimizations. Even though OpenBLAS fully utilizes all CPU cores (up to 128 threads), FastMap shows no advantage over *mmap-IO*, consistent with prior observations [46] that its lockless optimizations mainly benefit small-file workloads (e.g., < 1 GB).

Performance in financial workloads. Financial backtesting is I/O-intensive, involving frequent random reads and writes on large historical market data files that cannot fit in memory, making I/O efficiency and cache management critical. To evaluate such workloads in practice, we run *backtrader* [1] on five years of historical stock data from four countries (U/I/C/J), using FBMs for market data. FastMap is omitted in this experiment due to deployment restrictions on the production market data servers; however, comparison with *mmap-IO* alone is sufficient to highlight the benefits of *umap*.

Figure 11(a) shows normalized JCT, with *mmap-IO* normalized to 1 for cross-dataset comparability. *umap* delivers up to 6.7× speedup by mitigating DFS I/O bottlenecks and enabling high-throughput FBM access on high-bandwidth storage. Due to the large size of market data, full in-memory loading is infeasible and also imposes significant overhead on the Linux memory subsystem. Figure 11(b) shows that *mmap-IO* leads to nearly 100% memory utilization, whereas *umap* maintains higher performance while keeping memory usage between 8% and 31%, thanks to efficient cache management.

Operational Resilience. In large-scale DFS environments, frequent *livelocks* arise from *mmap-IO* 's deferred-write mechanism. While this mechanism is effective on LFS, it interacts poorly with DFS consistency, which is typically enforced at coarse block sizes (hundreds of KBs to several MBs, e.g., 256 KB–16 MB in GPFS and Lustre). As a result, even when data resides in the local page cache, reads may trigger forced write-back and remote coherence checks, causing compute nodes to queue for distributed locks. In PB-scale systems, flushing 1 TB could take up to 3 hours, and in our production cluster, financial jobs experienced ≥10 livelock incidents per day, during which threads spent up to 90% of execution time in iowait without progress. These stalls were often misclassified as deadlocks by the batch scheduler, leading to forced termination and ≥100 core-hours lost per incident. After deploying *umap*, which bypasses *mmap-IO* 's asynchronous writeback via explicit POSIX writes and synchronous DFS metadata updates, livelocks were eliminated, with zero job terminations observed over 18 months of operation (§6.4).

6.3 Micro-benchmarks

In this section, we take a closer look at how *umap*'s performance varies across different configurations.

Throughput gains of *umap*. Figures 12(a) and 12(b) show the random parallel *read* and *write* performance of *umap* versus baselines on both LFS and DFS. The results are consistent across FBM sizes; here we focus on the 128GB FBM.

In single-threaded mode, *umap* on DFS shows a modest 20% drop in *read* performance compared to *mmap-IO* and FastMap on LFS (worst-case small random I/O; AI workloads use large I/O and thus see positive improvements). This is due to the lack of internal prefetching and lock-free optimizations for single-threaded access. Nevertheless, *umap* narrows the

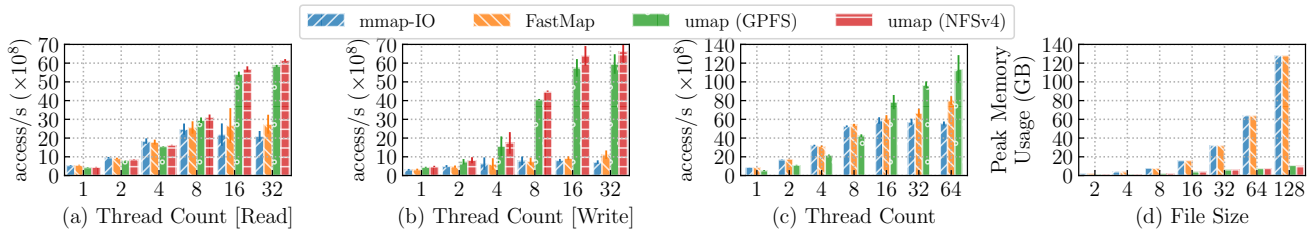


Figure 12: End-to-end performance comparison. In (c), only one bar is shown for *umap*, since it is not backed by a file system.

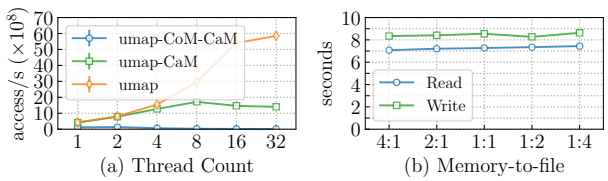


Figure 13: Ablation study results for *umap*.

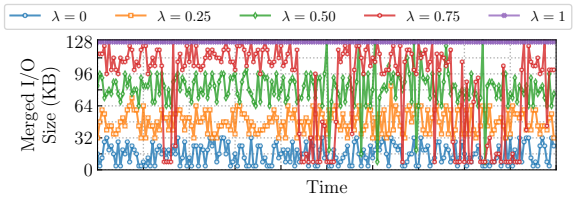


Figure 14: Impact of spatial locality on merged I/O sizes. A larger λ indicates stronger spatial locality.

performance gap between DFS and LFS used with *mmap-IO* (see Figure 2).

For single-threaded *write* operations, *umap* outperforms the baselines. Its network-friendly communication management and lazy-expansion cache strategy minimize overhead by reusing existing cache blocks and performing parallel write-back of the shadow copy. In contrast, baselines incur page faults and allocate new pages, incurring extra overhead.

Throughput scalability with *umap*. Figures 12(a) and 12(b) show that *umap* scales consistently and outperforms baselines. At 32 threads, *umap*+GPFS achieves 2.8× higher *read* throughput and 8.3× higher *write* throughput than *mmap-IO*+LFS. In contrast, *mmap-IO* performance degrades beyond 8–16 threads due to contention on the global *tree_lock*, which limits atomic page replacement during page faults. *umap*'s gains are more pronounced in *write* tasks because *mmap-IO* and FastMap benefit from prefetching on reads. Disk I/O is asymmetric; for example, the Optane SSD [7] has 5 μs read latency and 6 μs write latency. In contrast, *umap* 's performance over DFS shows near-parity between *read* and *write*, as both are limited by NIC bandwidth.

Beyond 32 threads, *umap*+DFS throughput plateaus around 190 Gbps, approaching the 200 Gbps NIC ceiling. To explore further scalability, we simulate a bandwidth-unconstrained environment where *umap* bypasses data synchronization and *mmap-IO* uses anonymous mappings. Figure 12(c) shows

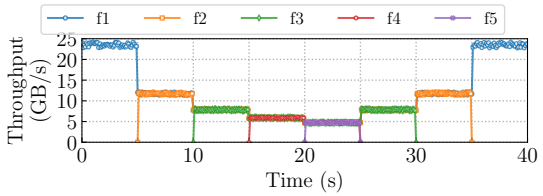


Figure 15: Inter-process fairness under *umap*.

that *umap* scales to 64 threads, while *mmap-IO* stalls at 16 threads, demonstrating the superior scalability of *umap* 's concurrency-aware caching protocol.

FastMap scales better than *mmap-IO*, but its overall performance gain is limited due to memory-stealing overhead: it divides system memory into 32 free-page lists, making it suitable only for small files. *umap* exhibits similar performance on GPFS and NFSv4; throughput is slightly higher on NFSv4, because NFSv4's weaker consistency model reduces the complexity of coherence management.

Memory efficiency of *umap* with sustained performance. Figure 12(d) shows the peak memory consumption of the *write* benchmark under varying file sizes with 32 threads. We observe that *mmap-IO* and FastMap consumes nearly as much memory as the file size itself, whereas *umap* uses less than 10.4% of the memory required by *mmap-IO* under the same conditions. This indicates that *umap* achieves higher performance with significantly lower memory usage, validating the effectiveness of its cache management strategy.

6.4 Effects of Individual Techniques

Communication management and cache protocol. We demonstrate that *umap*'s performance improvement is attributed to network-friendly communication management within the CoM and the concurrency-aware cache protocol within the CaM. In Figure 13(a), we compare the *read* performance of *umap* with and without CoM and CaM. Disabling both CoM and CaM (line *umap*-CoM-CaM) shows that *umap* performs poorly without these components. With only CoM enabled (bar *umap*-CaM), *umap*'s performance improves by 3.5× in single-threaded mode but does not scale beyond 8 threads. The inclusion of CoM and CaM (line *umap*), however, scales the performance linearly.

To analyze the effectiveness of CoM, we evaluate its I/O merging mechanism PIAO under varying access locality.

Table 1: Breakdown of execution progress for *mmap-IO* and *umap* over GPFS.

Test Type	context-switches	CPU-migrations	CPU cycles	page-faults	branches	branch-misses	$\frac{lockcost}{JCT-iowait}$	$\frac{iowait}{JCT}$
mmap-IO+LFS	7.6×10^3	7.1×10^1	2.5×10^{11}	2.0×10^6	1.2×10^{10}	3.6×10^7	68.0%	1.7%
mmap-IO+DFS	1.1×10^8	8.6×10^5	3.4×10^{12}	4.0×10^6	2.7×10^{11}	1.5×10^9	76.1%	88.9%
umap+DFS	2.1×10^3	3.1×10^1	6.3×10^{10}	6.3×10^5	1.4×10^{10}	3.1×10^7	1.2%	15.0%

We use a synthetic model, inspired by stochastic locality studies [48] and fio [4], that interpolates between random and sequential access via a parameter $\lambda \in [0,1]$. Each file has N blocks, and requests evolve as $x_t = (x_{t-1} + \Delta_t) \bmod N$, $\Delta_t \sim \text{Uniform}(0, R(\lambda))$, where $R(\lambda) = \lceil \texttt{iodepth} \cdot \lambda \rceil + jitter$ bounds the run length and `iodepth` is the number of concurrent outstanding I/O requests. Thus, $\lambda = 0$ yields fully random access, $\lambda = 1$ fully sequential access, and intermediate values partial locality. As shown in Figure 14, the average merged size grows from 17.2 KB ($\lambda = 0$) to 127.9 KB ($\lambda = 1.0$), confirming that PIAO effectively exploits locality and achieves DFS-friendly communication.

Cache Management. Figure 13(b) replicates the experiment from Figure 4 to evaluate *umap* 's lazy-expansion cache strategy under multi-task workloads. We run tests on one node, but restrict the available memory to 64 GB using *cgroup* limits, launching up to 16 concurrent tasks. Each task uses *umap*-backed *fio* [4] to issue 4 million single-threaded random accesses to a 16 GB FBM stored on DFS. This yields total mapped sizes from 16 GB to 256 GB, with memory-to-file ratios from 4:1 to 1:4. As shown, task performance remains stable across all ratios—even when the total mapped size far exceeds physical memory. By contrast, *mmap-IO* suffers severe slowdowns once memory falls short (Figure 4). These results highlight *umap* 's ability to sustain performance under memory pressure via its efficient cache management.

Fairness. We evaluate the fairness of *umap* 's CoM across concurrent processes, focusing on network bandwidth utilization. Fairness here refers specifically to network bandwidth sharing rather than memory usage, which we do not enforce, following the semantics of *mmap-IO*and FastMap; memory isolation can still be achieved via *cgroup* limits if needed. We launch multiple sequential *read* processes with staggered start/end times to ensure continuous bandwidth demand. As shown in Figure 15, a single process nearly saturates the available bandwidth (>95%), while N concurrent processes share it roughly equally. When a process finishes, its bandwidth is immediately redistributed to the remaining processes. These results demonstrate that *umap* 's CoM layer enforces fair bandwidth sharing while maintaining high efficiency in multi-tenant DFS deployments.

Resolving Livelock. We further profile kernel-level behavior to explain the poor performance of *mmap-IO*+DFS. Table 1 shows the execution breakdown of each *write* task under 32 threads. Several insights emerge. First, *mmap-IO*+DFS suffers from severe *iowait*, where CPUs stall on flushing dirty pages to DFS, causing a form of *livelock* [21] in which threads remain active but make no progress. Second, *umap* significantly improves CPU efficiency, cutting CPU usage by 98.2% and CPU migrations by 56.3% by eliminating the lock contention and wasted cycles that dominate *mmap-IO*. Third, lock contention accounts for most non-iowait time in *mmap-IO* (68.0% on LFS, 76.1% on DFS), largely due to spin locks (*native_queued_spin_lock_slowpath* in Linux). *umap* reduces this overhead to 1.2%. Although it shows a higher proportion of iowait than *mmap-IO*+LFS, the absolute time is lower because of its shorter JCT.

6.5 Latency Trade-offs

One concern is that *umap* introduces higher per-access latency due to its request-merging and scheduling logic (Figure 16). Although *mmap-IO* shows lower latency for individual 4KB reads, this benefit is largely irrelevant for FBM workloads, whose performance is governed by aggregate throughput rather than single-access latency. Even for online serverless services such as vLLM, the key metric is model loading time (cold-start latency), not single-byte access time. As shown in Figure 9, *umap* speeds up bulk loading by 2.3×, directly improving service availability and elasticity. For latency-critical sub-4KB random I/Os, *mmap-IO over local NVMe SSD remains ideal*; however, in the throughput-bound regimes typical of disaggregated data processing, the added latency by *umap* is a necessary trade-off to fully utilize the network.

7 Related Work

Page cache management. Many studies have examined techniques for high-performance page cache management for emerging fast storage devices like NVMe [14] SSDs. These approaches include designing concurrency-friendly page cache indexes [53, 54, 58], isolating memory mapping performance across applications [55, 62], and reducing cache allocation overhead through batch allocation or dedicated cache management units [43, 54, 64, 68]. *umap* is the first memory mapping runtime specifically tailored for DFS, fully accounting for the network behavior inherent in DFS—a factor that has been overlooked in previous studies.

Prefetching. Beyond the Linux kernel's built-in prefetching, many techniques have been proposed to enhance storage performance. OS-level approaches such as Lynx [41], ATS [37], HoPP [44], FastMap [54], and MMap [52] improve memory mapping via learning-based strategies or optimized disk layouts. Other studies [25, 32] target workload-specific prefetching at the application layer, while cross-layer designs [30] al-

low applications to influence system-level prefetching. These techniques are orthogonal to *umap*, which does not implement prefetching. Integrating them could further boost *umap*'s performance in read-heavy scenarios.

Workload-specific cache management. Some studies [46, 56, 57] optimize memory mapping for specific workloads such as key-value stores, databases, and distributed training by tailoring caching strategies. In contrast, *umap* targets random matrix access on network-attached DFS and remains agnostic to any particular workload. Prior work on user-space caching (e.g., Tricache [28]) focuses on improving local cache efficiency, while *umap* targets coordination and I/O amplification issues specific to DFS. User-space file systems based on FUSE [3] have gained popularity in recent AI systems. However, since they rely on the OS page cache, they inherit the limitations of mmap-IO in distributed settings. In contrast, *umap* redesigns the mmap path to be aware of distributed file system characteristics.

8 Discussion

In our 18-month deployment, *umap* eliminated previously observed livelock scenarios and significantly reduced instability such as OOM-induced failures, leading to more stable large-scale job execution. Our large-scale deployment of *umap* reveals broader implications for memory and storage system designs in disaggregated architectures. We summarize two *conceptual observations* and three *practical lessons* from our deployment.

Observation #1: Rethinking VM for Disaggregated Storage. Compute–storage decoupling exposes a structural mismatch between page-granularity VM and the block-oriented semantics of DFSes. Page-fault-driven I/O fragments remote accesses, amplifies metadata coordination, and prevents effective bandwidth utilization. As workloads increasingly operate over remote datasets (exceeding 95% in our cluster), future systems must move beyond traditional VM-based loading models toward coarser-grained, storage-aware abstractions.

Observation #2: Workload Semantics Favor Coarse-Grained Access. Across data analytics, model training, and financial computing, we observe that matrix- and tensor-dominated workloads rarely benefit from fine-grained paging or generic read-ahead heuristics. Their access patterns are coarse, structured, and explicitly known by the application. Exposing these semantics to the memory system enables much better performance than page-level mechanisms designed for general-purpose locality.

These observations motivate deeper cross-layer co-design between runtimes and DFSes. DFSes increasingly require semantic hints to schedule I/O efficiently, while applications require predictable interaction with remote memory. *umap*'s chunk-oriented abstraction suggests a path toward unified compute–storage interfaces that preserve memory-like convenience while embracing DFS-native granularity.

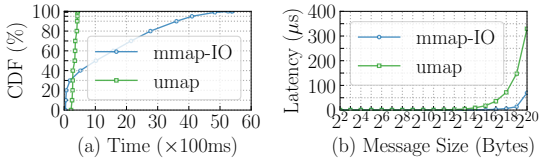


Figure 16: Message synchronization latency.

Lessons Learned. Operating *umap* in production reveals additional operational insights that go beyond the abstraction mismatch.

Lesson #1: Implicit page-based I/O undermines observability. On remote storage, *mmap-IO* turns network delays, DFS contention, and metadata bottlenecks into opaque page stalls, obscuring both performance anomalies and root causes. Coarse-grained, explicit I/O operations expose system behavior and produce far more predictable failure modes.

Lesson #2: Kernel-level greedy caching is harmful in multi-tenant clusters. Linux's page cache aggressively retains remote data with limited reuse, amplifying noisy-neighbor effects and triggering avoidable OOM events. Effective caching must shift to the application or runtime layer, where reuse semantics are known and can be explicitly controlled.

Lesson #3: Predictability outweighs micro-optimizations at scale. Eliminating livelock paths, bounding long-tail behavior, and simplifying retry logic improved job completion rates far more than median-performance tuning. In large clusters, robustness and isolation—not marginal throughput gains—dominate overall efficiency.

Applicability. The issues addressed by *umap* arise primarily in network file systems that involve distributed metadata management and locking (e.g., NFSv4 and GPFS). In contrast, block storage systems (e.g., EBS) or locally mounted file systems do not exhibit the same *mmap-IO* pathologies, as page faults do not trigger distributed coordination. *umap* is therefore designed for the former setting.

9 Conclusion

The rise of resource-disaggregated architectures presents both challenges and opportunities for optimizing memory mapping on DFS in FBM workloads. We present *umap*, a memory mapping runtime tailored for random matrix access in FBM workloads on DFS. By integrating network-aware I/O scheduling with dynamic cache management, *umap* improves data access efficiency while reducing memory overhead. Evaluation across synthetic benchmarks and production environments demonstrates consistent advantages over existing approaches.

Acknowledgments

We thank the shepherd and anonymous reviewers for their comments. This work was supported by the National Natural Science Foundation of China under Grant 62025203.

References

[1] backtrader. https://github.com/kilasuelika/backtradercpp.git.

[2] FastMap. https://github.com/CARV-ICS-FORTH/FastMap.git.

[3] Filesystem in Userspace. https://sourceforge.net/projects/fuse/.

[4] Flexible I/O tester. https://fio.readthedocs.io/en/latest/fio_doc.html.

[5] IBM General Parallel File System. https://www.ibm.com/docs/en/gpfs.

[6] Infographics: Operation Costs in CPU Clock Cycles. http://ithare.com/infographics-operation-costs-in-cpu-clock-cycles/.

[7] Intel® Optane™ SSD DC D4800X Series. https://www.intel.com/content/www/us/en/products/sku/129970/intel-optane-ssd-dc-d4800x-series-1-5tb-2-5in-pcie-2x2-3d-xpoint/specifications.html.

[8] Intel® Xeon® Platinum 8260 Processor. https://www.intel.com/content/www/us/en/products/sku/192474/intel-xeon-platinum-8260-processor-35-75m-cache-2-40-ghz/specifications.html.

[9] JuiceFS Adopters. https://juicefs.com/docs/community/adopters.

[10] Linux mmap API. https://linux.die.net/man/2/mmap.

[11] Lockless Programming Considerations for Xbox 360 and Microsoft Windows. https://learn.microsoft.com/en-us/windows/win32/dxtecharts/lockless-programming.

[12] Network File System. https://www.ibm.com/docs/en/i/7.4.0?topic=systems-network-file-system-nfs.

[13] NumPy memmap API. https://numpy.org/doc/stable/reference/generated/numpy.memmap.html.

[14] NVM Express. https://nvmexpress.org/.

[15] OpenBLAS. https://github.com/OpenMathLib/OpenBLAS.git.

[16] Page Cache. https://en.wikipedia.org/wiki/Page_cache.

[17] PyTorch MemoryMappedTensor. https://docs.pytorch.org/tensordict/main/reference/generated/tensordict.MemoryMappedTensor.html.

[18] SuiteSparse: A Suite of Sparse matrix packages. https://github.com/DrTimothyAldenDavis/SuiteSparse.

[19] The fourth extended file systems. https://man7.org/linux/man-pages/man5/ext4.5.html.

[20] vLLM Model Loader. https://github.com/vllm-project/vllm/tree/main/vllm/model_executor/model_loader.

[21] Franklin Abodo, Robert Rittmuller, Brian Sumner, and Andrew Berthaume. Detecting work zones in shrp 2 nds videos using deep learning based computer vision. In *2018 17th IEEE International Conference on Machine Learning and Applications (ICMLA)*, pages 679–686. IEEE, 2018.

[22] David Abramson, Chao Jin, Justin Luong, and Jake Carroll. A BeeGFS-based caching file system for data-intensive parallel computing. In *Asian Conference on Supercomputing Frontiers*, pages 3–22. Springer International Publishing Cham, 2020.

[23] Paszke Adam, Gross Sam, Chintala Soumith, Chanan Gregory, Yang Edward, DeVito Zachary, Lin Zeming, Desmaison Alban, Antiga Luca, and Lerer Adam. Automatic differentiation in PyTorch. In *In NIPS 2017 Autodiff Workshop: The Future of Gradient-based Machine Learning Software and Techniques*.

[24] Dhruba Borthakur. The Hadoop Distributed File System: Architecture and Design, 2008.

[25] Benjamin Cassell, Tyler Szepesi, Jim Summers, Tim Brecht, Derek Eager, and Bernard Wong. Disk prefetching mechanisms for increasing http streaming video server throughput. *ACM Transactions on Modeling and Performance Evaluation of Computing Systems (TOMPECS)*, 3(2):1–30, 2018.

[26] Jungsik Choi, Jiwon Kim, and Hwansoo Han. Efficient Memory Mapped File I/O for In-Memory File Systems. In *9th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 17)*, 2017.

[27] Marco Dantas, Diogo Leitão, Peter Cui, Ricardo Macedo, Xinlian Liu, Weijia Xu, and João Paulo. Accelerating deep learning training through transparent storage tiering. In *2022 22nd IEEE International Symposium on Cluster, Cloud and Internet Computing (CCGrid)*, pages 21–30. IEEE, 2022.

[28] Guanyu Feng, Huanqi Cao, Xiaowei Zhu, Bowen Yu, Yuanwei Wang, Zixuan Ma, Shengqi Chen, and Wenguang Chen. Tricache: A user-transparent block cache enabling high-performance out-of-core processing with in-memory programs. *ACM Transactions on Storage*, 19(2):1–30, 2023.

[29] Yao Fu, Leyang Xue, Yeqi Huang, Andrei-Octavian Brabete, Dmitrii Ustiugov, Yuvraj Patel, and Luo Mai. {ServerlessLLM}:{Low-Latency} serverless inference for large language models. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, pages 135–153, 2024.

[30] Shaleen Garg, Jian Zhang, Rekha Pitchumani, Manish Parashar, Bing Xie, and Sudarsun Kannan. Crossprefetch: Accelerating i/o prefetching for modern storage. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1*, pages 102–116, 2024.

[31] Campbell R Harvey and Yan Liu. Backtesting. *Available at SSRN 2345489*, 2015.

[32] Brandon Haynes, Maureen Daum, Dong He, Amrita Mazumdar, Magdalena Balazinska, Alvin Cheung, and Luis Ceze. Vss: A storage system for video analytics. In *Proceedings of the 2021 International Conference on Management of Data*, pages 685–696, 2021.

[33] K. He, X. Zhang, S. Ren, and J. Sun. Deep residual learning for image recognition. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 770–778, 2016.

[34] Yongchao He, Wenfei Wu, Yanfang Le, Ming Liu, and ChonLam Lao. A Generic Service to Provide In-Network Aggregation for Key-Value Streams. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, pages 33–47, 2023.

[35] Mohammad Hedayati, Kai Shen, Michael L Scott, and Mike Marty. Multi-Queue fair queuing. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 301–314, 2019.

[36] Paul Hudak. A semantic model of reference counting and its abstraction (detailed summary). In *Proceedings of the 1986 ACM Conference on LISP and Functional Programming*, pages 351–363, 1986.

[37] Song Jiang, Xiaoning Ding, Yuehai Xu, and Kei Davis. A prefetching scheme exploiting both data layout and access history on disk. *ACM Transactions on Storage (TOS)*, 9(3):1–23, 2013.

[38] Ziheng Jiang, Haibin Lin, Yinmin Zhong, Qi Huang, Yangrui Chen, Zhi Zhang, Yanghua Peng, Xiang Li, Cong Xie, Shibiao Nong, Yulu Jia, Sun He, Hongmin Chen, Zhihao Bai, Qi Hou, Shipeng Yan, Ding Zhou, Yiyao Sheng, Zhuo Jiang, Haohan Xu, Haoran Wei, Zhang Zhang, Pengfei Nie, Leqi Zou, Sida Zhao, Liang Xiang, Zherui Liu, Zhe Li, Xiaoying Jia, Jianxi Ye, Xin Jin, and Xin Liu. MegaScale: Scaling large language model training to more than 10,000 GPUs. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*, pages 745–760, Santa Clara, CA, April 2024. USENIX Association.

[39] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. Imagenet classification with deep convolutional neural networks. *Commun. ACM*, 60(6):84–90, May 2017.

[40] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph Gonzalez, Hao Zhang, and Ion Stoica. Efficient memory management for large language model serving with pagedattention. In *Proceedings of the 29th Symposium on Operating Systems Principles*, pages 611–626, 2023.

[41] Arezki Laga, Jalil Boukhobza, Michel Koskas, and Frank Singhoff. Lynx: A learning linux prefetching mechanism for ssd performance model. In *2016 5th Non-Volatile Memory Systems and Applications Symposium (NVMSA)*, pages 1–6. IEEE, 2016.

[42] Chengtao Lai, Zhongchun Zhou, Akash Poptani, and Wei Zhang. Lcm: Llm-focused hybrid spm-cache architecture with cache management for multi-core ai accelerators. In *Proceedings of the 38th ACM International Conference on Supercomputing*, pages 62–73, 2024.

[43] Viktor Leis, Adnan Alhomssi, Tobias Ziegler, Yannick Loeck, and Christian Dietrich. Virtual-memory assisted buffer management. *Proceedings of the ACM on Management of Data*, 1(1):1–25, 2023.

[44] Haifeng Li, Ke Liu, Ting Liang, Zuojun Li, Tianyue Lu, Hui Yuan, Yinben Xia, Yungang Bao, Mingyu Chen, and Yizhou Shan. Hopp: Hardware-software co-designed page prefetching for disaggregated memory. In *2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 1168–1181. IEEE, 2023.

[45] Qiang Li, Lulu Chen, Xiaoliang Wang, Shuo Huang, Qiao Xiang, Yuanyuan Dong, Wenhui Yao, Minfei Huang, Puyuan Yang, Shanyang Liu, Zhaosheng Zhu, Huayong Wang, Haonan Qiu, Derui Liu, Shaozong Liu, Yujie Zhou, Yaohui Wu, Zhiwu Wu, Shang Gao, Chao Han, Zicheng Luo, Yuchao Shao, Gexiao Tian, Zhongjie Wu, Zheng Cao, Jinbo Wu, Jiwu Shu, Jie Wu, and Jiesheng Wu. Fisc: A large-scale cloud-native-oriented file system. In *21st USENIX Conference on File and Storage Technologies (FAST 23)*, pages 231–246, Santa Clara, CA, February 2023. USENIX Association.

[46] Zhiyue Li and Guangyan Zhang. StreamCache: Revisiting Page Cache for File Scanning on Fast Storage

Devices. In *2024 USENIX Annual Technical Conference (USENIX ATC 24)*, pages 1119–1134, Santa Clara, CA, July 2024. USENIX Association.

[47] Tian Ma, Cunfei Liao, and Fuwei Jiang. Factor momentum in the chinese stock market. *Journal of Empirical Finance*, 75:101458, 2024.

[48] Richard L. Mattson, Jan Gecsei, Donald R. Slutz, and Irving L. Traiger. Evaluation techniques for storage hierarchies. *IBM Systems journal*, 9(2):78–117, 1970.

[49] Maged M Michael and Michael L Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *Proceedings of the fifteenth annual ACM symposium on Principles of distributed computing*, pages 267–275, 1996.

[50] Jayashree Mohan, Rohan Kadekodi, and Vijay Chidambaram. Analyzing io amplification in linux file systems. *arXiv preprint arXiv:1707.08514*, 2017.

[51] Francisco Muñoz-Martínez, Raveesh Garg, Michael Pellauer, José L Abellán, Manuel E Acacio, and Tushar Krishna. Flexagon: A multi-dataflow sparse-sparse matrix multiplication accelerator for efficient dnn processing. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, pages 252–265, 2023.

[52] Anastasios Papagiannis, Manolis Marazakis, and Angelos Bilas. Memory-mapped I/O on steroids. In *Proceedings of the Sixteenth European Conference on Computer Systems*, pages 277–293, 2021.

[53] Anastasios Papagiannis, Giorgos Saloustros, Pilar Gonzalez-Ferez, and Angelos Bilas. An efficient memory-mapped key-value store for flash storage. In *Proceedings of the ACM Symposium on Cloud Computing*, pages 490–502, 2018.

[54] Anastasios Papagiannis, Giorgos Xanthakis, Giorgos Saloustros, Manolis Marazakis, and Angelos Bilas. Optimizing memory-mapped I/O for fast storage devices. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pages 813–827. USENIX Association, July 2020.

[55] Jonggyu Park and Young Ik Eom. Weight-aware cache for application-level proportional i/o sharing. *IEEE Transactions on Computers*, 71(10):2395–2407, 2021.

[56] Ivy Peng, Marty McFadden, Eric Green, Keita Iwabuchi, Kai Wu, Dong Li, Roger Pearce, and Maya Gokhale. Umap: Enabling application-driven optimizations for page management. In *2019 IEEE/ACM Workshop on Memory Centric High Performance Computing (MCHPC)*, pages 71–78. IEEE, 2019.

[57] Ivy B Peng, Maya B Gokhale, Karim Youssef, Keita Iwabuchi, and Roger Pearce. Enabling scalable and extensible memory-mapped datastores in userspace. *IEEE Transactions on Parallel and Distributed Systems*, 33(4):866–877, 2021.

[58] Kiet Tuan Pham, Seokjoo Cho, Sangjin Lee, Lan Anh Nguyen, Hyeongi Yeo, Ipoom Jeong, Sungjin Lee, Nam Sung Kim, and Yongseok Son. Scalecache: A scalable page cache for multiple solid-state drives. In *Proceedings of the Nineteenth European Conference on Computer Systems*, pages 641–656, 2024.

[59] Amedeo Sapio, Marco Canini, Chen-Yu Ho, Jacob Nelson, Panos Kalnis, Changhoon Kim, Arvind Krishnamurthy, Masoud Moshref, Dan Ports, and Peter Richtárik. Scaling distributed machine learning with In-Network aggregation. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*, pages 785–808, 2021.

[60] Frank Schmuck and Roger Haskin. GPFS: A Shared-Disk file system for large computing clusters. In *Conference on file and storage technologies (FAST 02)*, 2002.

[61] Zhibing Sha, Jun Li, Fengxiang Zhang, Min Huang, Zhigang Cai, Francois Trahay, and Jianwei Liao. Visibility Graph-based Cache Management for DRAM Buffer Inside Solid-state Drives. *ACM Trans. Storage*, 19(3), jun 2023.

[62] Prateek Sharma, Purushottam Kulkarni, and Prashant Shenoy. Per-vm page cache partitioning for cloud computing platforms. In *2016 8th International Conference on Communication Systems and Networks (COMSNETS)*, pages 1–8. IEEE, 2016.

[63] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.

[64] Brian Van Essen, Henry Hsieh, Sasha Ames, Roger Pearce, and Maya Gokhale. Di-mmap—a scalable memory-map runtime for out-of-core data-intensive applications. *Cluster Computing*, 18:15–28, 2015.

[65] vLLM Team. vLLM Implementation, 2025. https://github.com/vllm-project/vllm.git.

[66] Chen Wang, Kathryn Mohror, and Marc Snir. File system semantics requirements of hpc applications. In *Proceedings of the 30th International Symposium on High-Performance Parallel and Distributed Computing*, pages 19–30, 2021.

[67] Yiduo Wang, Yufei Wu, Cheng Li, Pengfei Zheng, Biao Cao, Yan Sun, Fei Zhou, Yinlong Xu, Yao Wang, and

Guangjun Xie. Cfs: Scaling metadata service for distributed file system via pruned scope of critical sections. In *Proceedings of the Eighteenth European Conference on Computer Systems*, pages 331–346, 2023.

[68] Lars Wrenger, Florian Rommel, Alexander Halbuer, Christian Dietrich, and Daniel Lohmann. LLFree: Scalable and Optionally-PersistentPage-Frame Allocation. In *2023 USENIX Annual Technical Conference (USENIX ATC 23)*, pages 897–914, 2023.

[69] Jun Xiao, Yaocheng Xiang, Xiaolin Wang, Yingwei Luo, Andy Pimentel, and Zhenlin Wang. Floria: A fast and featherlight approach for predicting cache performance. In *Proceedings of the 37th International Conference on Supercomputing*, pages 25–36, 2023.

[70] Juncheng Yang, Yazhuo Zhang, Ziyue Qiu, Yao Yue, and Rashmi Vinayak. Fifo queues are all you need for cache eviction. In *Proceedings of the 29th Symposium on Operating Systems Principles*, pages 130–149, 2023.

[71] Takeshi Yoshimura, Tatsuhiro Chiba, Manish Sethi, Daniel Waddington, and Swaminathan Sundararaman. Speeding up model loading with fastsafetensors. *arXiv preprint arXiv:2505.23072*, 2025.

[72] Guowei Zhang, Nithya Attaluri, Joel S Emer, and Daniel Sanchez. Gamma: Leveraging gustavson's algorithm to accelerate sparse matrix multiplication. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 687–701, 2021.

ARTIFACT EVALUATED
USENIX
AVAILABLE

CoPilotIO: CPU as a Co-pilot for GPU I/O to Free GPU Compute

Guanyi Chen[†] Qi Chen[†] Shu Yin[‡] Jian Zhang[†]

[‡]*ShanghaiTech University* [†]*The Hong Kong University of Science and Technology (Guangzhou)*

Abstract

Limited GPU memory increasingly forces modern AI and data analytics workloads to access terabyte-scale datasets and model states from storage, making efficient GPU I/O critical. Existing GPU I/O engines are either CPU-centric or GPU-centric. CPU-centric approaches avoid consuming GPU resources but often fail to provide high-throughput, on-demand GPU access due to kernel overheads and limited parallelism. GPU-centric approaches enable fine-grained on-demand I/O but require intensive I/O polling that consumes valuable GPU resources and introduces intra-warp, inter-warp, and inter-SM I/O stalls. We present CoPilotIO[1], a novel GPU I/O engine that delivers high-throughput, on-demand storage access without sacrificing GPU compute resources. CoPilotIO adopts an asynchronous GPU I/O architecture in which GPUs initiate I/O while CPU cores act as I/O co-pilots responsible for completion polling. To enable efficient coordination, CoPilotIO introduces a split SQ/CQ architecture, hardware barrier-based synchronization, a lock-free barrier-table, and adaptive CPU–GPU co-polling. Across microbenchmarks and real applications, including GoFS, LLM Mixture-of-Experts (MoE) inference, and Deep Learning Recommendation Models (DLRM), CoPilotIO reduces I/O-induced stalls by up to 55.5%, requires 50% fewer SMs to saturate the GPU PCIe bandwidth, accelerates GoFS by up to 17.4%, and improves application performance by up to 85%.

1 Introduction

Graphics Processing Units (GPUs) have become the primary compute device powering today's most demanding applications, including large-scale graph analytics [20, 36], scientific simulations [4, 13], high-performance computing (HPC) [8, 16], and the rapid emergence of generative AI, such as large language models (LLMs) [6, 9, 32]. These workloads routinely operate on multi-terabyte datasets, trillion-token corpora, and model checkpoints exceeding hundreds of gigabytes [6, 34, 38]. However, despite substantial advances in compute capability, GPU memory capacity remains fundamentally restricted: modern high-bandwidth memory (HBM) typically offers only 40–192 GB [1, 2, 24] per accelerator. This mismatch between massive data footprints and limited on-device memory means GPUs must continuously fetch data at extremely high throughput to keep their compute units fully utilized. As a result, the design of an efficient, scalable GPU I/O stack becomes critical.

Existing GPU I/O engines *exhibit a fundamental tradeoff among I/O performance, support for on-demand access, and GPU compute efficiency*. Broadly, prior designs fall into two categories: CPU-centric I/O and GPU-centric I/O.

CPU-centric I/O engines, exemplified by NVIDIA GPUDirect Storage (GDS) [23], enable peer-to-peer (P2P) data transfer between storage and GPU memory, but the entire control path remains on the CPU and relies heavily on the mature kernel I/O stack. This design offers high GPU compute efficiency because no GPU cores are consumed in the I/O path. However, limited CPU parallelism and substantial kernel storage stack overhead prevent GDS from saturating the available GPU PCIe bandwidth (e.g., PCIe 4.0 ×16 at ~25 GB/s) [28, 39]. Moreover, GDS lacks true on-demand access for GPU kernels: GPU threads cannot directly initiate I/O, so applications typically rely on CPU-side preloading, which introduces significant data amplification and unnecessary data movement [28].

To overcome these limitations, recent work [28] introduces GPU-centric I/O engines that empower the GPU to directly initiate I/O and manage the entire I/O path, including NVMe command submission, completion-queue polling, and request processing. This design achieves high throughput and allows GPU kernels to fetch data just-in-time, providing true on-demand access. However, shifting the full I/O stack onto the GPU introduces a critical drawback: *valuable GPU compute resources are consumed by the I/O stack, particularly by intensive I/O polling on NVMe completion queues.* This polling interferes with normal GPU execution and leads to substantial I/O-induced stalls, preventing compute kernels from making forward progress and ultimately degrading overall application

[1] Artifact available at https://github.com/HKUST-GZ-SYSTEM/OSDI26-CoPilotIO

performance. Our analysis (Section 3) shows that three types of stalls manifest at multiple levels, including intra-warp, inter-warp, and inter-SM stalls. For example, an intra-warp stall occurs when a GPU warp issues an on-demand I/O request to prefetch data; subsequent independent computation stalls because the warp must continuously poll the completion queue to detect I/O completion. This busy-waiting prevents the warp from making forward progress and reduces available GPU compute throughput by up to 87%. In effect, GPU-centric I/O can inadvertently convert highly parallel GPU SMs into inefficient I/O controllers.

Therefore, existing designs face a fundamental limitation: CPU-centric approaches offer high GPU compute efficiency but suffer from low I/O performance and inefficient CPU-mediated I/O control or data movement [19, 23, 30], whereas GPU-centric approaches provide high I/O performance and on-demand capability but consume substantial GPU compute cycles [18, 27, 28, 39]. This tension raises a central design question: **Can we achieve high-performance, on-demand GPU I/O without sacrificing valuable GPU compute resources?**

In this paper, we answer this question affirmatively. We argue that GPU compute resources are extremely valuable and should be prioritized for application computation rather than I/O servicing. Therefore, we advocate for a new GPU I/O architectural direction in which the CPU and GPU cooperate as peers rather than one replacing the other, delivering high-throughput I/O while preserving GPU compute availability.

We present CoPilotIO, a GPU I/O engine that treats the CPU as an I/O co-pilot for GPU applications. CoPilotIO introduces a new asynchronous I/O architecture across CPU and GPU in which the GPU initiates fine-grained on-demand I/O, while the CPU handles I/O queue polling. By offloading polling to the CPU, CoPilotIO fully eliminates GPU-side I/O stalls while preserving true on-demand access for GPU kernels. Designing such an architecture raises several key challenges. First, how can we achieve both high I/O throughput and low latency when the I/O stack resides across two devices (CPU and GPU), especially given that traditional CPU-side I/O stacks suffer from substantial OS kernel overheads? Second, how do we design an efficient synchronization and notification mechanism when the I/O submission and completion paths span both the CPU and GPU? Third, given the limited parallelism available on CPUs, how can we saturate the high PCIe bandwidth of modern GPUs while avoiding I/O-induced GPU stalls?

CoPilotIO addresses the above challenges with the following contributions: (1) A split SQ/CQ architecture across CPU–GPU memory, combined with a user-level SPDK-like I/O library on the CPU side [15], enabling fast GPU-initiated I/O submission and efficient CPU polling without OS kernel software overheads. (2) A hardware barrier–based synchronization mechanism [22] and a lock-free barrier-table, providing efficient, contention-free coordination and notification between the CPU and GPU. (3) A CQ-based adaptive CPU–GPU co-polling mechanism that dynamically migrates the polling role between CPU and GPU based on runtime I/O load, ensuring high throughput under high I/O load, while still avoiding GPU-side polling stalls.

We implement CoPilotIO in approximately 5K lines of code. We evaluate CoPilotIO using a comprehensive suite of microbenchmarks that measure reductions in I/O-induced stalls, pure I/O performance, the effectiveness of CQ-based adaptive polling, and the scalability and GPU compute efficiency of CoPilotIO. Our results show that CoPilotIO reduces I/O stalls by up to 55.5% compared to the state-of-the-art GPU-centric I/O engine, BaM. Enabled by CQ-based adaptive polling, CoPilotIO consistently sustains high I/O throughput even under dynamically changing I/O loads. In addition, CoPilotIO requires 50% fewer SMs to saturate four SSDs (i.e., the GPU's PCIe bandwidth) than BaM. We further demonstrate the practicality of CoPilotIO by integrating it into the state-of-the-art GPU filesystem GoFS [18], where CoPilotIO delivers up to 17.4% performance improvement. Finally, we evaluate CoPilotIO on real-world applications, including LLM Mixture-of-Experts (MoE) inference [12, 29] and Deep Learning Recommendation Models (DLRM) [21], both of which show up to 85% performance gains.

2 Background

2.1 GPU Architecture

Modern GPUs consist of many streaming multiprocessors (SMs), each acting as an independent parallel compute unit equipped with hundreds of CUDA cores. When a CUDA kernel launches, its threads are grouped into thread blocks, which are assigned to SMs. Within each SM, threads are further partitioned into warps of 32 threads, which execute instructions in lock-step. All SMs share a global L2 cache and the off-chip DRAM memory. A warp is the fundamental execution and scheduling unit in the SIMT model: all active threads in a warp issue the same instruction but operate on different data. Each SM has four warp schedulers, allowing up to four warps to issue instructions (active) simultaneously. Each scheduler selects the next ready warp to issue to the CUDA cores whenever the currently executing warp goes to sleep or completes, ensuring high concurrency. If a warp continuously occupies the cores without making progress, it limits scheduling opportunities and reduces overall GPU parallelism. For instance, an NVIDIA A100 GPU [22] contains 108 SMs with a total of 40-80 GB of memory, and each SM can track up to 64 resident warps (2,048 threads). Across the whole device, the A100 can therefore have over 200,000 in-flight threads, providing massive hardware-managed multi-threading.

In addition, GPUs also support directly reading from and writing to CPU memory over PCIe using zero-copy access [25]. In this mode, GPU threads can access host memory without an explicit DMA transfer, allowing the GPU to

operate on data resident in CPU memory with low latency. Conversely, GDRCopy [10] enables CPU threads to directly read and write GPU VRAM by creating user-space mappings of GPU memory through GPUDirect RDMA, providing a low-latency path for CPU-initiated accesses to GPU-resident data structures.

2.2 GPU I/O Engines

We categorize the state-of-the-art GPU I/O engines into two approaches: CPU-centric I/O and GPU-centric I/O.

CPU-centric I/O: GPUs rely on the host CPU to initiate I/O. As discussed above, CPU-centric approaches offer high GPU compute efficiency but suffer from low I/O performance and lack true on-demand GPU access. The standard solution is NVIDIA GPUDirect Storage (GDS) [23], which enables peer-to-peer PCIe DMA transfers between NVMe SSDs and GPU memory, allowing data to be written directly into GPU buffers without staging through host memory. However, its control plane remains CPU-orchestrated, which prevents true GPU on-demand access [28]. Data must be pre-loaded from storage into memory before GPU computation can proceed, leading to unnecessary data movement and read amplification. In addition, GDS suffers from limited I/O performance because its control plane resides in the OS kernel, incurring substantial kernel software overheads [28] that make it difficult to fully saturate the GPU's PCIe bandwidth (e.g., ~25 GB/s). Although GDS enables direct SSD-to-GPU DMA transfers, each I/O request still requires CPU-side I/O submission, completion handling, and synchronization through the kernel I/O stack, introducing significant control-path overheads such as system calls, kernel traps, buffer registration, DMA mapping, and NVMe queue management [23]. While asynchronous I/O interfaces (e.g., `cuFileReadAsync`) can overlap the data-transfer latency of individual requests, they cannot eliminate the per-request control-path overhead that CPU cores must process. As a result, the aggregate control-path overhead can exceed what the available CPU cores can sustain, even when using asynchronous I/O.

Additionally, several CPU-centric GPU I/O designs have been proposed to provide on-demand access for GPU applications. GPUfs [33] introduces a wrapper library that exposes familiar filesystem APIs directly to GPU code. However, all I/O operations are ultimately executed by the CPU through the host kernel's filesystem stack, resulting in limited I/O performance due to kernel overheads and CPU mediation. Moreover, GPUfs incurs substantial GPU compute overhead: the GPU issues I/O requests via an RPC mechanism to the CPU, causing excessive GPU-side communication and synchronization costs during I/O operations. ActivePointers [30] introduces a memory-mapped file abstraction for GPUs by moving the control path of page-fault handling onto the GPU. However, the data movement path remains entirely CPU-centric: for every major page fault, the OS still reads the page from the SSD into CPU memory and then copies it to the GPU. As a result, substantial GPU compute resources are wasted on complex page-fault handling logic, leading to significant overhead during I/O-intensive workloads.

GPU-centric I/O: To provide GPU on-demand I/O, BaM [28] allocates submission/completion queues and I/O buffers directly in GPU memory, maps NVMe doorbells into the GPU address space, and allows GPU threads to prepare commands, ring doorbells, and poll completions without invoking CPU-side I/O stacks. BaM provides high I/O performance and on-demand capability, but wastes substantial GPU compute cycles as the entire control and data paths of each I/O operation are initiated and completed on the GPU. This design forces GPU threads to actively poll for I/O completion, introducing substantial I/O-induced stalls into GPU applications, as detailed in Section 3. AGILE [39] builds on top of BaM to provide asynchronous I/O by dedicating additional SMs for polling, but its control and data planes remain entirely GPU-driven, which cannot avoid inter-warp and inter-SM stalls mentioned in Section 3. CAM [35] offloads SSD management to the CPU, but adopts a two-level submission path where GPU threads write request metadata into CPU memory and CPU-side threads issue the actual NVMe commands, adding a CPU-mediated indirection on the submission path.

Additionally, several GPU-centric filesystems have been proposed [18, 27], all of which build on BaM as their underlying GPU I/O engine. GeminiFS [27] adopts a companion filesystem design that embeds essential metadata into files, retrieves this metadata into GPU memory, and, together with a GPU-friendly page cache and NVMe queues, exposes simplified file-based direct storage access to the GPU. GoFS [18] further pushes the entire filesystem into the GPU to fully bypass the CPU. However, offloading the full filesystem and I/O management to the GPU introduces substantial interference and I/O stalls for regular GPU compute workloads. All filesystem tasks, including metadata lookup, caching, traversal, and I/O queue management, now compete with AI kernels for SM cycles. In modern AI workloads where GPU compute resources are extremely precious and increasingly scarce, dedicating GPU threads to non-compute tasks directly undermines utilization. Every GPU cycle spent on the filesystem I/O stack is a cycle not spent on model training or inference, leading to degraded performance, reduced parallelism, and poor cost efficiency at cluster scale.

3 Motivation

Tradeoff between I/O performance and GPU compute efficiency: For CPU-centric I/O systems such as GDS, GPU compute resources are not consumed for I/O operations. However, GDS suffers from limited I/O throughput due to substantial OS kernel software overheads and restricted CPU parallelism. As shown in Figure 3a, with 16 CPU threads, GDS fails to saturate a single SSD's bandwidth across a range of I/O sizes. Achieving full GPU PCIe link bandwidth (approximately 25

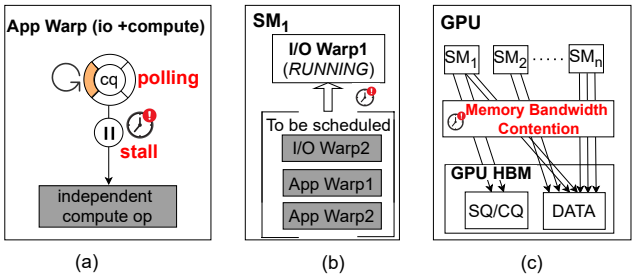
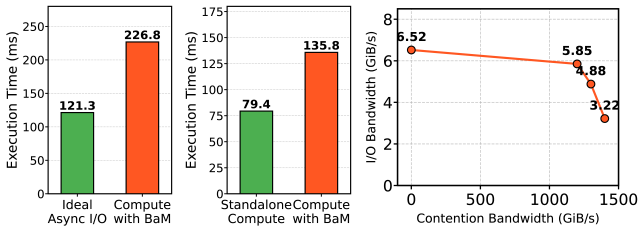


Figure 1: (a) Intra-warp stall: independent compute operation is stalled by I/O polling; (b) Inter-warp stall: compute warp is delayed from being scheduled as I/O polling warps keep occupying the GPU cores; (c) Inter-SM stall: I/O polling warps and regular application memory accesses contend for the global GPU memory.



(a) Intra-warp **(b)** Inter-warp **(c)** Inter-SM

Figure 2: I/O Stall Analysis of the GPU-Centric Design.

GB/s for PCIe 4.0 ×16) requires saturating four SSDs concurrently, which is difficult for GDS because its kernel-level overhead and limited core parallelism prevent it from issuing and completing I/Os at a sufficiently high rate. On the other hand, BaM can saturate the bandwidth of four SSDs because it moves the entire I/O path onto the GPU, eliminating CPU-side software overheads and exploiting the GPU's massive parallelism. However, achieving this level of throughput requires more than 80 SMs (Figure 10a) on an NVIDIA A100 for I/O, which results in extremely poor compute efficiency. Moreover, when I/O and compute tasks are co-located on the GPU, they introduce multiple levels of stalls, including intra-warp, inter-warp, and inter-SM stalls as discussed next.

Lesson 1: Modern GPU I/O designs face a fundamental trade-off: *CPU-centric approaches preserve GPU compute capacity but deliver low I/O throughput, while GPU-centric approaches achieve high throughput but severely degrade GPU compute efficiency.*

Co-locating I/O and compute tasks on the GPU causes severe I/O stalls: Although GPU-centric I/O provides high throughput and supports on-demand access while fully bypassing the CPU, we observe that designs such as BaM incur substantial I/O polling stalls when I/O tasks are co-located with compute tasks on the GPU. Because BaM must continuously poll the NVMe completion queue to detect I/O completions, polling warps interfere with compute warps, leading to significant stalls and reduced overall GPU efficiency. As shown in Figure 1, we identify three polling-induced stalls.

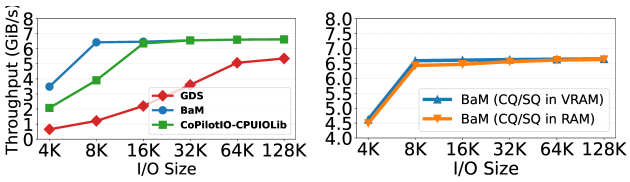
(1) Intra-warp stall: A stall inside a warp. When an application warp polls for data (e.g., waiting for the completion queue), it cannot execute subsequent instructions until that data arrives. Even if later instructions are independent of the polling data, the GPU's in-order warp execution model prevents them from being issued, causing the entire warp to stall.

(2) Inter-warp stall: A stall between warps but within an SM. When an I/O warp aggressively polls the completion queue, the GPU hardware warp scheduler is unaware that this warp makes no forward progress, as it remains marked as "ready". Consequently, it reduces the opportunities for other ready compute warps to execute and wastes valuable SM execution cycles while the requested data has not yet arrived. Since each SM can issue only one warp instruction per cycle, these unproductive polling instructions suppress useful compute work, reducing parallelism and overall SM efficiency.

(3) Inter-SM stall: A stall across SMs. In GPU-centric I/O designs, both application data and the NVMe SQ/CQ structures reside in global GPU memory. Unlike the previous two types of stalls, here the I/O performance itself is directly affected by application memory traffic (Figure 2c). Because application kernels concurrently access global memory, they introduce contention on memory channels and bandwidth, which interferes with BaM's frequent CQ polling. This contention delays CQ detection, slows down I/O completion handling, and ultimately degrades overall I/O performance.

Recent work, AGILE [39], proposes an asynchronous GPU-centric I/O model by dedicating specific SMs to poll the completion queue and overlap I/O with computation. However, this design requires allocating additional SMs exclusively for polling, consuming GPU execution resources that would otherwise be available for computation. For example, to saturate the bandwidth of 4 SSDs (limited by GPU PCIe bandwidth), AGILE must reserve 32 SMs purely for polling on an A100, which has only 108 SMs in total. Furthermore, AGILE continues to suffer from the inter-warp and inter-SM stalls discussed above. The application warp (i.e., the warp that initiates the I/O) must repeatedly poll a software lock flag to detect completion, which directly contributes to inter-warp stalls. In addition, the dedicated SMs used for CQ polling generate substantial memory traffic, further exacerbating inter-SM contention.

Analysis: We conduct a set of experiments to analyze the three types of stalls in detail. For intra-warp stalls, we launch multiple application warps, with each warp running computation alongside I/O operations that the computation does not depend on. We compare BaM against our CoPilotIO I/O engine. As shown in Figure 2a, using BaM increases intra-warp stall time by up to 1.87× due to synchronous polling. For inter-warp stalls, we co-run multiple compute warps with I/O warps. As illustrated in Figure 2b, compared to running the compute warp alone, co-running with BaM's polling warp increases the total execution time by up to 1.71×, showing that a spinning I/O warp competes aggressively with ready compute warps for SM scheduling slots. Finally, for inter-SM memory contention, we evaluate how regular GPU memory



(a) User-space I/O Polling **(b)** SQ/CQ on CPU vs GPU

Figure 3: (a) User-space CPU I/O polling performance comparing GDS and BaM; (b) Impact of placing Completion Queue (CQ) and Submission Queue (SQ) in CPU DRAM versus GPU VRAM.

traffic interferes with GPU-driven I/O. As shown in Figure 2c, increasing the memory bandwidth consumed by the application significantly degrades BaM's I/O bandwidth, demonstrating that GPU-side queue polling contends with application memory accesses across SMs.

Lesson 2: There is a fundamental flaw in current GPU-centric I/O: as long as the GPU is responsible for polling NVMe queues, the architecture inevitably imposes intra-warp, inter-warp, and inter-SM stalls. This motivates the adoption of a *fundamentally new asynchronous I/O engine that removes GPU-side polling and decouples I/O progress from GPU execution, eliminating the root cause of all three stall types.*

CPU can be used as a copilot for GPU I/O: Prior works [7, 18, 27, 28] often assume that CPU-centric I/O (e.g., GDS) suffers prohibitive OS kernel overhead, leading to the conclusion that its performance is substantially slower than GPU-centric designs such as BaM. This assumption motivates a series of GPU filesystems [18, 27] that rely on BaM to fully bypass the CPU. However, such bypassing overlooks CPU-side capabilities and may leave significant performance opportunities unexploited.

First, user-level CPU I/O polling can achieve performance comparable to GPU-centric I/O by removing the kernel software overheads. We implement a user-level SPDK-like system (CoPilot-CPUIOLib) by moving the NVMe queue pair into user space and using the CPU to handle SQ/CQ polling and directly moving data from storage to the GPU. As shown in Figure 3a, CoPilot-CPUIOLib can saturate SSD bandwidth with 16 CPU threads once the I/O size exceeds 16 KB.

However, **naive user-level CPU I/O polling alone cannot meet our goals.** First, it cannot provide fine-grained on-demand I/O initiation from the GPU, as all requests must originate on the CPU side. Second, the limited CPU parallelism is insufficient to sustain the high polling rate required for small I/O sizes (e.g., <16 KB), causing throughput to fall short of device capabilities. Moreover, saturating a GPU PCIe link would require more than 64 CPU cores dedicated solely to I/O polling, which is an unrealistic requirement in real server environments where CPU resources are shared with other applications. Therefore, an efficient adaptive polling design is required between CPU and GPU (Section 4.5).

In addition, **placing the SQ/CQ in CPU memory achieves I/O throughput comparable to GPU-resident queues.** This is because GPU accesses to pinned CPU memory through PCIe provide sufficiently high bandwidth for the small, cacheline-sized SQ/CQ entry performed during I/O. The GPU's polling only reads or writes a few bytes (4B) per command (e.g., command ID, status fields), so the control-path traffic is negligible relative to the large datapath DMA transfers. As a result, the overhead of using the GPU to poll the CQ in CPU memory is minimal and does not limit end-to-end NVMe throughput. As shown in Figure 3b, our evaluation of both configurations, SQ/CQ placed in CPU memory versus GPU memory and both using the GPU for polling, reveals that GPU-polling of CPU-resident queues can sustain throughput close to the optimal GPU-VRAM polling case while offering significantly lower interference with GPU compute workloads.

Properties	CPU Centric	GPU Centric		
Systems	GDS [23]	BaM [28]	AGILE [39]	CoPilotIO
On-demand GPU I/O	✗	✓	✓	✓
Saturate GPU PCIe bandwidth	✗	✓	✓	✓
High GPU compute efficiency	✓	✗	✗	✓
Avoid intra-warp stall	✓	✗	✓	✓
Avoid inter-warp stall	✓	✗	✗	✓
Avoid inter-SM stall	✓	✗	✗	✓

Table 1: Capabilities and limitations of state-of-the-art GPU I/O approaches. The last column shows our proposed CoPilotIO.

Lesson 3: *CPU cores and CPU memory are essential co-pilots for GPU I/O.* However, naive user-level CPU polling cannot sustain high throughput under limited CPU parallelism. Meanwhile, GPU polling of SQ/CQ placed in CPU memory still delivers sufficiently high bandwidth. These observations motivate the need for an efficient adaptive polling design that dynamically coordinates between CPU and GPU.

4 CoPilotIO Design

4.1 Design Goals and Principles

Goal 1: Eliminate I/O-induced stalls. CoPilotIO introduces a split SQ/CQ design across CPU-GPU memory and utilizes an asynchronous I/O architecture that leverages CPU cores as I/O co-pilots for CQ polling and uses GPU hardware barriers (*cuda::barrier*) to coordinate cross-device data synchronization. This design avoids GPU-side I/O polling stalls, including intra-warp, inter-warp, and inter-SM stalls.

Goal 2: Support high-performance on-demand I/O access from the GPU side. First, CoPilotIO provides a scalable, lock-free barrier table that allows the GPU to efficiently communicate with the CPU. Second, CoPilotIO implements a user-level SPDK-like design (CoPilot-CPUIOLib) that enables high-performance I/O polling on the CPU side.

Goal 3: Sustain high I/O bandwidth when CPU cores are insufficient under high I/O loads. CoPilotIO introduces the CQ-based adaptive polling. At runtime, CoPilotIO dynamically selects the most suitable polling agent based on I/O load. This adaptive design maintains high I/O bandwidth while simultaneously avoiding GPU I/O polling stalls.

Goal 4: Free more GPU cores for computation rather

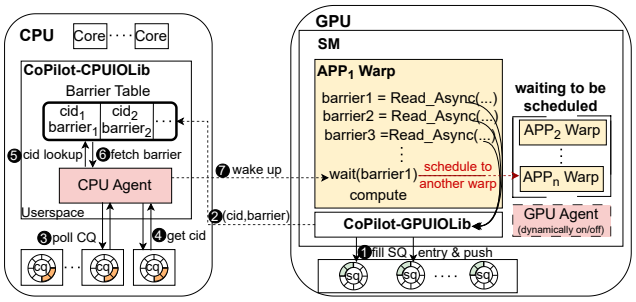


Figure 4: Design Overview of CoPilotIO. Application warps initiate I/O through CoPilot-GPUIOLib, after creating `cuda::barrier` and `cid`, ❶ CoPilot-CPUIOLib submits the I/O requests to SQs and immediately returns control to the application. ❷ CoPilot-GPUIOLib writes the `cid` and the `cuda::barrier` to barrier-table that resides on the CPU memory. ❸ and ❹ CPUAgent polls the CQ until it receives a CQ entry and extracts the `cid`. ❺ and ❻ CoPilot-CPUIOLib looks up the `cuda::barrier` using the `cid`. At the end, CoPilot-CPUIOLib uses ❼ to notify the application warp on the GPU.

than I/O. CoPilotIO enables CPU–GPU co-polling, allowing the system to saturate the full GPU PCIe bandwidth while using significantly fewer GPU cores for I/O.

4.2 CoPilotIO Design Overview

CoPilotIO is a new asynchronous GPU I/O engine that leverages the CPU as a co-pilot to deliver high on-demand I/O throughput while minimizing I/O-induced stalls and maximizing GPU compute efficiency. CoPilotIO consists of two libraries. On the GPU side, CoPilot-GPUIOLib enables GPU applications to initiate I/O requests and handles I/O submission operations. On the CPU side, CoPilot-CPUIOLib provides a high-performance, SPDK-like user-level library responsible for polling I/O completions. Both CPU and GPU have their own polling agents. CPUAgent is enabled by default, as one of the design goals is to free GPU cores for computation. GPUAgent is disabled initially but can be dynamically activated when needed. Its activation is governed by our CQ-based adaptive polling mechanism, which determines when GPU-side polling is necessary to sustain high I/O throughput under increasing load.

4.3 Split SQ/CQ Across CPU–GPU Memory

Traditional GPU-centric I/O designs co-locate both the NVMe submission queue (SQ) and completion queue (CQ) in GPU memory. While this enables GPU-initiated commands, it also introduces I/O polling stalls when co-located with GPU compute applications. To overcome this limitation, CoPilotIO adopts a split-queue architecture that places the SQ in GPU memory and the CQ in CPU memory. This split design preserves the key benefit of GPU-centric I/O that the GPU can directly fill the SQ with low latency, while shifting I/O completion detection (polling) to a CPU memory region that can be efficiently polled by both the CPU and GPU (Figure 3b).

First, placing the CQ in CPU memory naturally eliminates inter-SM polling stalls. In our design, the GPU accesses CPU memory directly over PCIe using zero-copy loads/stores without invoking explicit `cudaMemcpy()`. This approach ensures that I/O polling bypasses GPU memory, preventing interference with other compute kernel memory accesses and preserving SM parallelism.

However, naively splitting the SQ/CQ across CPU and GPU memory is not sufficient. Although this design removes inter-SM stalls and improves I/O performance, it cannot eliminate intra-warp and inter-warp stalls. Regardless of where the SQ resides, a GPU warp must still continuously poll the CQ to determine whether an I/O request has completed. As a result, any independent computation within the same warp is forced to wait for the I/O to finish. Similarly, other ready warps on the same SM cannot make forward progress because the polling warp remains "ready" and repeatedly preempts the scheduler, preventing efficient warp-level parallelism. Fortunately, placing the CQ in CPU memory enables the CPU to take over CQ polling. This observation motivates us to design a novel asynchronous I/O architecture built upon the split SQ/CQ design, which we discuss in the next section.

4.4 CoPilotIO Asynchronous I/O Architecture

At the core of CoPilotIO is a fully asynchronous I/O architecture that spans both the CPU and GPU. The key idea is to offload I/O polling to the CPU while using NVIDIA's hardware barrier to notify the GPU when data becomes ready. By eliminating GPU-side polling entirely, CoPilotIO effectively removes both intra-warp and inter-warp stalls. Next, we first describe the end-to-end asynchronous I/O flow by going through the main components of CoPilotIO.

Issuing I/O from the GPU: As mentioned, CoPilotIO provides a lightweight GPU-side library, *CoPilot-GPUIOLib*, which enables GPU applications to initiate I/O requests on demand. Applications interact with CoPilot-GPUIOLib through familiar POSIX-like interfaces (e.g. `async_read` and `async_write`) as mentioned above. For each I/O request, CoPilotIO creates a corresponding `cuda::barrier` instance, implemented using NVIDIA's hardware barrier support in the CUDA runtime. Once the application receives a `cuda::barrier` handle, it can simply place the barrier before dependent compute operations to ensure data availability. Crucially, unlike conventional polling, a `cuda::barrier` allows the GPU to suspend the waiting warp and let the hardware scheduler issue other ready warps, effectively eliminating inter-warp stalls during I/O wait periods while maintaining correctness. Before returning the `cuda::barrier` to the user, CoPilot-GPUIOLib allocates an NVMe Command ID (`cid`) [37] for the I/O request and saves the `cid` to `cuda::barrier` mapping in a *barrier-table* (described shortly). Each `cid` is uniquely associated with a `cuda::barrier`. After that, CoPilot-GPUIOLib immediately submits the I/O request to the SQ and returns control to the application.

I/O completion detection on the CPU: On the CPU side, *CoPilot-CPUIOLib* is the CPU-side user-level I/O engine of CoPilotIO and plays the role of an I/O "co-pilot" for the GPU. CoPilot-CPUIOLib uses an SPDK-like architecture, which moves NVMe queue management into user space and dedicates CPU cores to handle SQ/CQ submission, completion polling, and DMA coordination without kernel involvement. This design enables CoPilotIO to deliver high I/O throughput while serving as the control plane that drives GPU-initiated I/O. CoPilot-CPUIOLib continuously polls the CQ using dedicated CPU cores. Once the SSD updates a CQ entry to indicate completion, the CPU detects it, looks up the corresponding `cuda::barrier` in the shared barrier-table using the I/O request identifier (`cid`), and signals the GPU by triggering the associated `cuda::barrier`. When the barrier is signaled, the waiting GPU warp wakes up and the dependent computation resumes. On platforms that do not support the hardware barrier, the CPU can alternatively use GDRCopy [10] to write a completion flag directly into GPU memory.

Synchronization between CPU and GPU: To enable efficient CPU–GPU coordination, CoPilot-CPUIOLib preallocates *a lock-free barrier-table* in CPU memory, which is directly accessible by both processors. The table maintains a one-to-one mapping between each NVMe command ID (`cid`) and its corresponding `cuda::barrier`. First, on the GPU side, upon receiving an I/O request, CoPilot-GPUIOLib performs a GPU zero-copy write to insert the (`cid`, `cuda::barrier`) pair into the barrier-table. Each barrier-table is assigned to a private warp instance, eliminating lock contention between warps. During insertion, CoPilot-GPUIOLib uses the `cid` as the index. Since each entry is only 8 bytes, the GPU-to-CPU-memory write latency remains low. On the CPU side, once CoPilot-CPUIOLib observes a completion queue (CQ) entry, it retrieves the associated `cid` and directly indexes into the same barrier-table to fetch the stored `cuda::barrier`. The CPU then signals this `cuda::barrier` to notify the GPU that the requested data has arrived. In this way, the barrier-table provides a lightweight, lock-free communication channel that efficiently connects CQ polling on the CPU with barrier-based synchronization on the GPU.

CID reclamation: When the CPU observes a CQ completion entry, it extracts the `cid`, signals the corresponding `cuda::barrier`, and immediately marks the `cid` as available for reuse. Because each `cid` is uniquely bound to a single in-flight I/O request and the barrier-table is indexed by `cid`, reclamation is a simple index reset with no synchronization required. If an NVMe-level I/O error occurs (indicated by the status field in the CQ entry), the CPU still reclaims the `cid` and signals the `cuda::barrier`, but sets an error flag in the barrier table. Upon waking up, the GPU warp checks this flag and handles the failure (e.g., retries or propagates the error to the application). This design ensures that `cid` resources are never leaked, even under I/O failures.

Memory overhead of barrier-table: In the NVMe protocol, the `cid` is a 16-bit field, allowing up to 65,536 outstanding commands per SQ/CQ pair. Each table entry (`cid` + `cuda::barrier` object) is 8B. To saturate a modern NVMe SSD typically requires 128 SQ/CQ pairs, each configured with a queue depth of 1024. To fully utilize the GPU's PCIe bandwidth, four SSDs are enough to saturate it. Thus, the total memory footprint is 8 B × 1024 × 128 CQs × 4 SSDs = 4 MB. As a result, the barrier-table imposes negligible memory overhead while supporting the full bandwidth required to saturate the GPU PCIe link.

This asynchronous handshake between CPU, GPU, and NVMe SSD eliminates GPU-side I/O stalls and saves more GPU compute for application execution. First, there is no GPU I/O polling, thereby eliminating intra-warp stalls. Once an I/O request is issued, the application kernel can continue executing independent work without waiting for data. Second, even with dependent operations, `cuda::barrier` suspends the waiting warps, allowing the scheduler to issue other ready warps, effectively avoiding inter-warp stalls. In addition, by separating the SQ and CQ across devices, CoPilotIO eliminates contention on GPU memory accesses during I/O polling, thereby further reducing inter-SM stalls.

Challenges: GPU I/O polling can only be fully avoided when the system has sufficient CPU cores to sustain high CQ polling throughput. As shown in Figure 3a, our evaluation uses 16 CPU cores for I/O polling; however, in a real server environment where other applications compete for CPU resources, fewer cores may be available, making it difficult for CPU-side polling to saturate the full GPU PCIe bandwidth (25 GB/s). Moreover, for small I/O sizes, even 16 polling cores are insufficient: small requests require high I/O depth and many active SQ/CQ pairs to reach the device's peak bandwidth, but limited CPU parallelism cannot provide enough polling throughput. In contrast, the GPU offers thousands of hardware threads capable of providing ample parallelism to saturate storage bandwidth. Yet shifting polling back to the GPU would reintroduce the very GPU-side stalls we aim to eliminate. Thus, **the central challenge is achieving high I/O bandwidth when CPU cores are insufficient, while simultaneously avoiding GPU I/O polling stalls.** Therefore, we propose CQ-based adaptive CPU-GPU co-polling.

4.5 CQ-based Adaptive CPU-GPU Co-Polling

We propose CQ-based Adaptive CPU-GPU Co-Polling, which addresses two questions: (1) When should GPUAgent be turned on or off? (2) How to minimize I/O stalls when I/O polling is on the GPU?

As shown in Figure 5, GPUAgent is disabled by default. Initially, only CPUAgent is responsible for polling. We decide when to turn on the GPUAgent based on the number of pending entries in the CQ. The number of outstanding CQ entries reflects the CPU's ability to keep up with I/O processing: if many entries remain unconsumed, it indicates that the CPU

Algorithm 1: CQ-based adaptive polling algorithm

Input: For each queue CQ_i: current number of pending entries p_i;
Upper migration threshold T_i^{high} and lower migration threshold T_i^{low} ($T_i^{\text{low}} < T_i^{\text{high}}$);
Decision batch size R_i (trigger a migration decision every R_i I/O requests);
Current polling mode $mode_i \in \{\text{CPU}, \text{GPU}\}$;
Current I/O submission binding $bind_i \in \{\text{CPU_CQ}, \text{GPU_CQ}\}$
State: For each CQ_i, a request counter req_i since the last decision, initialized to 0
Output: For each CQ_i, an adaptive choice of polling location and I/O binding

```
1  Function OnIORequest(CQ_i)
2      req_i ← req_i + 1;
3      if req_i ≥ R_i then
4          req_i ← 0;
5          DecideMigration(CQ_i);
6      end
7  end
8  Function DecideMigration(CQ_i)
9      p_i ← ReadPendingCount(CQ_i);
10     if mode_i = CPU and p_i > T_i^high then
11         mode_i ← GPU;
12         WakePollingWarpOnSameSM(CQ_i);
13         bind_i ← GPU_CQ;
14         UpdateQueueBindingToGPU(CQ_i);
15     else if mode_i = GPU and p_i < T_i^low then
16         mode_i ← CPU;
17         bind_i ← CPU_CQ;
18         UpdateQueueBindingToCPU(CQ_i);
19         OptionallySleepPollingWarp(CQ_i);
20     end
21     else
22         Keep current mode unchanged; return;
23     end
24 end
```

does not have enough cores to poll and process CQ entries in time.

In our design, each CQ is associated with a threshold. As Algorithm 1 shows, when CoPilot-CPUIOLib observes that the number of pending CQ entries exceeds this threshold, CoPilotIO stops inserting new CQ entries into the CPU-polled queue and immediately activates the GPUAgent. Newly generated CQ entries are then redirected to the queue polled by the GPU. Conversely, when the pending count falls below the threshold, CoPilotIO stops inserting new CQ entries into the GPU-polled queue, and the GPUAgent is disabled after draining all remaining pending CQ entries. At that point, polling

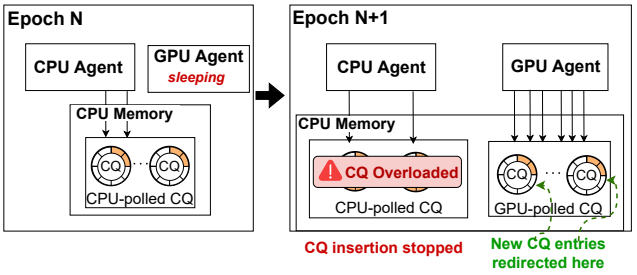


Figure 5: CQ-based Adaptive Polling. At epoch N, CPUAgent is enabled by default to poll the CQ, while GPUAgent remains disabled. At epoch N + 1, CoPilotIO detects that the CQ is becoming overloaded and the CPU can no longer sustain high throughput, GPUAgent is activated to assist with polling.

responsibility shifts back to the CPU as new CQ entries are inserted into the CPU-polled queue.

Importantly, first, the CQ itself always resides in CPU memory to avoid inter-SM stalls. When the GPUAgent is active, the GPU directly polls the CPU-resident CQ using zero-copy reads. As shown in Figure 3b, GPU polling achieves similar throughput regardless of where the CQ is placed, making CPU-resident CQs preferable for reducing GPU-side contention. Second, CoPilotIO does not migrate polling threads between CPU and GPU; instead, it simply controls which CQ receives newly inserted entries, thereby steering the polling load to the appropriate device. This design avoids costly thread migration or context switching while enabling fast, lightweight transitions between CPU-side and GPU-side polling. Third, the CPUAgent is not disabled when the GPUAgent is enabled. Instead, CoPilotIO allows true CPU–GPU co-polling, where each agent polls its own dedicated CQ without interfering with the other. Because the CPU and GPU poll separate queues, there is no contention or shared-state synchronization overhead between them, enabling smooth load balancing and higher sustained I/O throughput under bursty or high-concurrency workloads.

Moreover, when I/O polling is moved back to the GPU, we need to minimize the I/O stalls reintroduced by GPU-side polling. BaM requires each warp that issues I/O to poll for completion itself, leading to severe inter-warp stalls as multiple polling warps contend for the warp scheduler. AGILE improves upon this by assigning one agent warp to each NVMe queue and consolidating all agent warps onto a few dedicated SMs that do not run any application computation. However, this design has two drawbacks: first, it wastes GPU compute resources since those dedicated SMs cannot be used for application workloads; second, because multiple polling warps are consolidated onto the same SM, inter-warp interference among them still exists. Furthermore, AGILE's polling warps cannot be disabled once launched and continue to occupy GPU resources even when the I/O load is low.

In contrast, GPUAgent takes a different approach. For each SM, GPUAgent is initialized at kernel launch time and activates a single dedicated warp to perform I/O polling only

when polling responsibility is transferred from the CPU to the GPU. Unlike BaM, GPUAgent uses only one warp per SM for polling rather than having each I/O warp poll individually, thereby significantly reducing inter-warp stalls. Unlike AGILE, GPUAgent runs alongside the application kernel on the same SM rather than consolidating polling warps onto dedicated SMs, thus avoiding inter-warp interference among multiple polling warps. Since only a single warp serves as the GPUAgent per SM, the impact on application performance is minimal. Moreover, GPU application kernels can seamlessly switch to having GPUAgent handle polling without any code changes. Furthermore, the polling warp of GPUAgent can enter a sleep state when not polling and be woken up on demand, and can be completely disabled when all I/O polling is moved back to the CPU, allowing all warps on each SM to be fully available for computation.

GPUAgent operates similarly to CoPilot-CPUIOLib: it shares the barrier-table located in shared memory with the application kernel, polls the CPU-resident CQ via zero-copy reads, and upon detecting a completed CQ entry, uses the corresponding `cid` to locate the associated `cuda::barrier` in the table and signals it to wake up the waiting application warp.

5 Evaluation

We evaluate CoPilotIO to answer the following questions:

- How effective is the asynchronous I/O of CoPilotIO in avoiding I/O-induced stalls when I/O and compute tasks are co-located on the GPU?
- What I/O throughput and latency does CoPilotIO achieve when running I/O tasks alone?
- How effective is the CQ-based adaptive CPU-GPU co-polling mechanism?
- Can CoPilotIO saturate the GPU's PCIe bandwidth?
- What is the overall impact of CoPilotIO on real-world applications?

5.1 Experimental Setup and Methodology

Testbed: Our testbed consists of a dual-socket Intel Xeon Gold 6530 system, with 32 cores per socket, 256 GB of DDR5 DRAM (4 × 64 GB), and we evaluate on both an NVIDIA A100 GPU with 40 GB of memory and an NVIDIA H800 GPU with 80 GB of memory. The system runs Ubuntu 22.04.4 with Linux kernel 5.15.0, CUDA 12.4, and NVIDIA driver version 550.54.14. We use four 1 TB Samsung 990 Pro NVMe SSDs (PCIe 4.0 x4) for storage.

Methodology: We compare CoPilotIO against state-of-the-art GPU-centric I/O systems, including BaM (synchronous I/O) and AGILE (asynchronous I/O). We evaluate GDS in Figure 8, but exclude it from the other experiments as it does not support on-demand GPU-initiated I/O, and prior work has already demonstrated its limited I/O performance (Section 3).

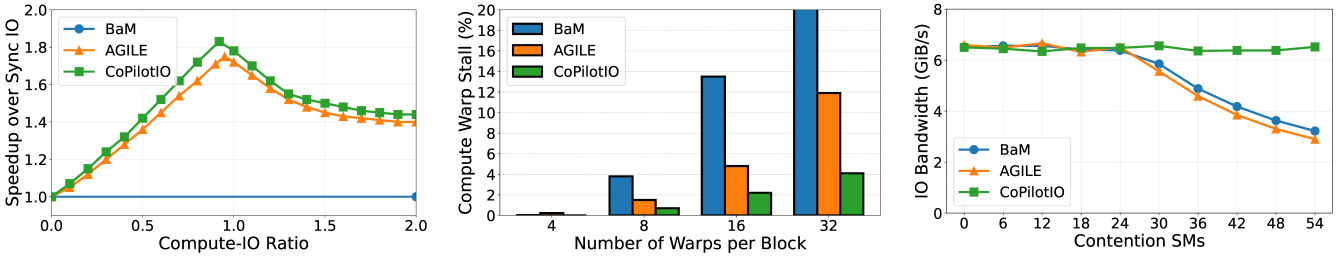
5.2 Microbenchmarks

5.2.1 Reduction of I/O Stalls

First, in Figure 6, we design three microbenchmarks by co-locating I/O and compute tasks on the GPU to evaluate the effectiveness of CoPilotIO in reducing I/O-induced stalls.

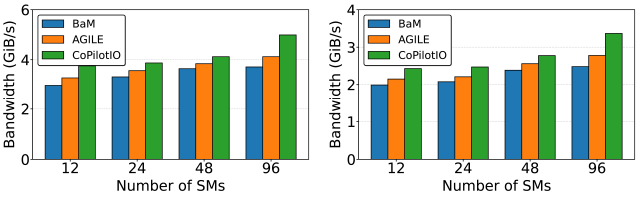
Intra-warp stalls: We launch 32 warps, where each warp performs computation (Fused Multiply-Add) interleaved with random read operations. The computation is independent of the data returned by the reads. For the x-axis, we vary the compute-to-I/O ratio by fixing the number of I/O operations while progressively increasing the amount of computation. For the y-axis, we report the speedup relative to synchronous I/O (BaM), which cannot overlap I/O with computation. As shown in Figure 6a, before the compute-to-I/O ratio reaches 1.0, both AGILE and CoPilotIO continue to improve performance. In this region, the workload is I/O-dominated, so avoiding intra-warp stalls directly increases throughput. Once the ratio exceeds 1.0, the amount of computation becomes larger than the amount of I/O, causing the workload to become compute-dominated, and the relative speedup naturally decreases. Overall, both AGILE and CoPilotIO effectively reduce intra-warp stalls, but CoPilotIO achieves better performance. The key reason is that AGILE dedicates an entire SM to I/O polling, yet the application's I/O warp must still repeatedly poll a software lock flag to detect I/O completion, leaving residual intra-warp stall overhead. In contrast, CoPilotIO completely eliminates intra-warp stalls by offloading all I/O polling to the CPU and leveraging a hardware barrier to provide fully asynchronous I/O–compute overlap.

Inter-warp stalls: We co-run I/O warps that issue purely random reads alongside compute warps that perform pure computation (Fused Multiply-Add). We fix four compute warps and progressively increase the number of I/O warps. The x-axis represents the total number of active warps. The y-axis reports the compute warp's stall ratio. As shown in Figure 6b, as the number of I/O warps increases, the compute warp under using BaM experiences stall rates of up to 20%. This is because BaM performs I/O polling on the GPU, causing I/O warps to repeatedly occupy GPU execution resources and delay compute warps. As mentioned, AGILE still relies on polling a software lock flag to detect I/O completion, which introduces substantial overhead. This overhead is particularly severe for inter-warp stalls because the application's I/O warp cannot sleep and therefore continues occupying the GPU cores, preventing compute warps from being scheduled to run. Moreover, the dedicated SM wastes GPU compute capacity. In contrast, CoPilotIO offloads I/O polling entirely to the CPU and relies on hardware barriers to notify the GPU. This allows I/O warps to sleep and enables the hardware GPU scheduler to issue ready compute warps without interference. Consequently, compared with BaM, CoPilotIO reduces inter-warp stalls by up to 18.6%.



(a) Intra-warp stalls **(b)** Inter-warp stalls **(c)** Inter-SM contention

Figure 6: Stall-elimination microbenchmarks across three dimensions.



(a) Random read **(b)** Random write

Figure 7: Pure-I/O performance.

Inter-SM stalls: We co-run I/O warps with compute warps, where the compute warps frequently access global GPU memory, generating substantial memory traffic. For the x-axis, we vary the number of contending SMs (i.e., SMs running compute warps), and for the y-axis, we report the achieved I/O bandwidth. As shown in Figure 6c, for both BaM and AGILE, I/O bandwidth begins to drop once memory contention exceeds 24 SMs and continues to decline as additional SMs generate memory traffic. Under heavy contention, the I/O bandwidth can drop by up to 50.6%. This degradation occurs because both designs use the GPU to poll the CQ located in GPU memory, which exacerbates on-chip memory contention. For AGILE, the need to poll additional software lock flags further increases memory contention, resulting in even worse performance than BaM. In contrast, CoPilotIO maintains high I/O bandwidth regardless of the number of contending SMs. This is due to its split SQ/CQ design, which places the CQ in CPU memory and therefore naturally avoids inter-SM stalls caused by GPU global-memory contention.

System	Read Latency (us)	Write Latency (us)
BaM	47.72	43.11
AGILE	38.77	34.28
CoPilotIO	37.75	33.14

Table 2: Average latency comparison with 96 SMs, each utilizing 32 warps for 4 KB I/O operations.

5.2.2 Pure-I/O Performance

Next, we evaluate CoPilotIO with pure-I/O workloads. We run random reads and random writes with a 4 KB I/O size and vary the number of SMs. Figure 7 reports the I/O throughput, and Table 2 shows the corresponding latency. CoPilotIO outperforms both BaM and AGILE across workloads in terms of both throughput and latency. First, these results demonstrate that the split SQ/CQ design does not compromise I/O performance: GPU-resident SQs enable fast submission, while CPU-resident CQs maintain high throughput through efficient user-level CPU polling. Second, compared to BaM and AGILE, the performance gains of CoPilotIO arise because inter-warp stalls and inter-SM stalls also occur even in pure-I/O workloads. For instance, under inter-warp interference, a pending I/O warp cannot be scheduled if a previously issued I/O warp continues polling, leading to reduced parallelism.

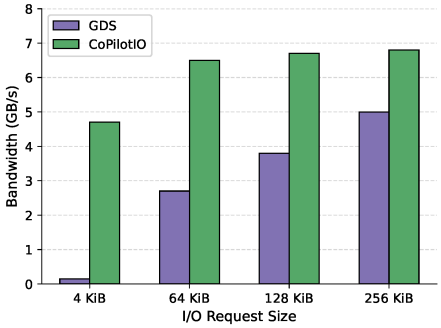


Figure 8: Random read bandwidth: GDS vs. CoPilotIO.

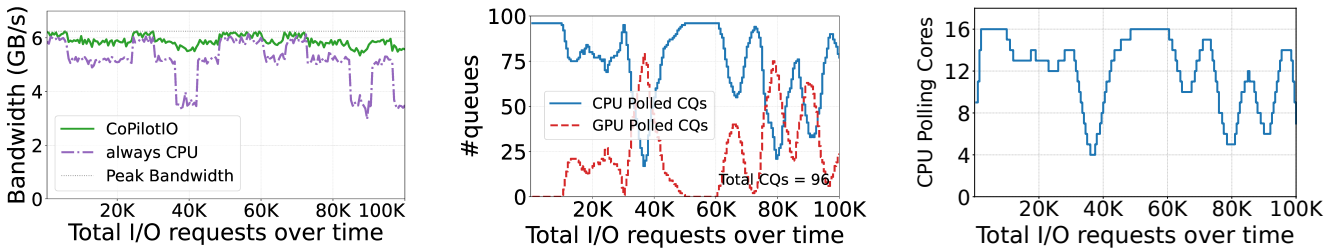
5.2.3 Comparison with GDS

We compare CoPilotIO against NVIDIA GDS with 16 CPU threads, performing random reads on an SSD. Figure 8 shows the achieved bandwidth across different I/O request sizes.

CoPilotIO consistently outperforms GDS across all I/O sizes. The performance advantage comes from avoiding the high kernel software overheads in GDS, including system calls, buffer registration, and DMA address mapping in the kernel I/O path. As the I/O size increases, these overheads become less dominant because the data transfer time increasingly amortizes the per-request kernel software cost. In contrast, with the benefit of CoPilot-CPUIOLib, CoPilotIO bypasses the kernel entirely: the GPU submits NVMe commands directly through user-space queues, while the CPU performs only lightweight CQ polling without kernel involvement. This design enables CoPilotIO to sustain near-peak I/O bandwidth even for small 4 KB I/O requests.

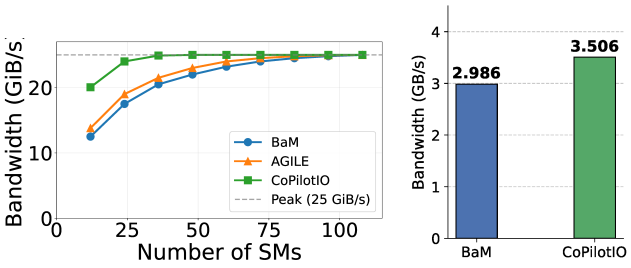
5.3 Effectiveness of Adaptive Polling

Next, we validate the effectiveness of GPUAgent-based adaptive CPU-GPU co-polling. We design a microbenchmark that emulates an AI application whose IOPS fluctuate over time.



(a) Bandwidth **(b)** CQ Distribution **(c)** CPU Resource Analysis

Figure 9: Analysis of CPU and GPU polling: (a) Bandwidth curves, (b) CQ polling distribution, and (c) CPU resource analysis.



(a) **(b)**

Figure 10: (a) The scalability of CoPilotIO; (b) GoFS random read performance over different I/O engines: BaM vs. CoPilotIO.

We use CoPilotIO-cpu-only to denote a configuration where CoPilotIO always uses the CPU for polling, and CoPilotIO-adaptive-polling to denote CoPilotIO with our CQ-driven adaptive polling mechanism enabled.

Figure 9a shows how I/O throughput evolves over time. CoPilotIO-adaptive-polling consistently sustains high throughput, while CoPilotIO-cpu-only exhibits significant variation as the IOPS load changes. This occurs because, under very high IOPS, CPU-only polling cannot keep up due to limited CPU-side parallelism, causing delays in processing CQ entries. In contrast, with adaptive polling, CoPilotIO detects CQ overload and gradually activates GPUAgent to offload polling to the GPU, allowing the system to maintain stable high throughput. Figure 9b shows the corresponding breakdown of CQ entries processed by the CPU and GPU over time. Initially, most CQs are handled by the CPU, but as the IOPS increase, the number of CQs processed by the CPU gradually decreases while the number handled by the GPU increases. This trend directly validates the behavior of our adaptive polling algorithm, demonstrating that CoPilotIO automatically shifts polling responsibility to the GPU when CPU-side polling becomes a bottleneck.

CPU Resource Analysis: Figure 9c shows the number of active CPU polling cores over time under adaptive polling. When I/O load is low and the GPU is not polling, CoPilotIO uses up to 16 CPU cores for CQ processing. As I/O pressure increases and the adaptive mechanism activates GPU co-polling, the CPU polling load decreases, and the number of active CPU cores drops to as few as 3 or 4 cores. This demonstrates that the adaptive mechanism dynamically reduces CPU core usage when GPU co-polling is engaged, freeing CPU resources for other system tasks. In steady state with moderate I/O load, CoPilotIO typically requires only 4 to 8 dedicated CPU cores.

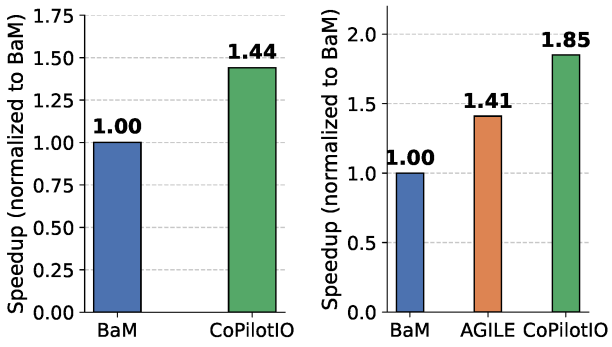
5.4 The Scalability of CoPilotIO

Next, we evaluate the scalability of CoPilotIO with multiple SSDs. Figure 10a shows the aggregate bandwidth achieved by BaM, AGILE, and CoPilotIO when performing 8 KB random reads across four SSDs. Each SM runs 16 warps, and we configure 108 NVMe queue pairs with a queue depth of 1024.

The results demonstrate two key advantages of CoPilotIO. First, CoPilotIO exhibits excellent scalability as the number of SMs increases, with bandwidth growing smoothly until saturating the PCIe 4.0 x16 limit of 25 GB/s. Second, and more importantly, CoPilotIO saturates the PCIe bandwidth with only 24 SMs, whereas BaM and AGILE require more than 72 SMs to achieve the same throughput. This 3× reduction in SM usage stems from CoPilotIO CPU-GPU co-polling design: by offloading CQ polling to the CPU, CoPilotIO eliminates GPU-side polling overhead and allows each SM to sustain higher effective I/O throughput. Furthermore, when the I/O load is high, the CPU and GPU can poll CQs simultaneously, further improving completion queue processing capacity. In contrast, BaM and AGILE rely solely on the GPU for polling and must dedicate significant GPU resources to this task, requiring substantially more SMs to reach peak bandwidth. This efficiency gain is particularly valuable for mixed compute–I/O workloads, where the saved SMs can be allocated to application computation rather than I/O management.

5.5 Performance with GPU Filesystem

We further evaluate CoPilotIO using the state-of-the-art GPU filesystem GoFS [18], which is built on the BaM GPU-centric I/O engine and fully offloads filesystem management to the GPU, bypassing the CPU entirely. GoFS is a mixed compute–I/O workload: its filesystem operations involve multiple compute tasks such as data indexing, block management, and metadata manipulation. We integrated CoPilotIO into GoFS with fewer than 20 lines of code changes. Although AGILE does not natively support GoFS, we attempted to extend AGILE for this experiment, but it encountered unknown runtime errors; therefore, AGILE is excluded from this evaluation. Figure 10b presents the random-read (4 KB I/O size)



(a) MoE SSD offloading. **(b)** DLRM inference.

Figure 11: Real-world application performance.

performance of GoFS when using BaM versus CoPilotIO. CoPilotIO achieves up to 17.4% performance improvement over BaM due to our efficient asynchronous I/O architecture, which removes all I/O stalls.

5.6 Performance on Real-world Applications

LLM Mixture-of-Experts (MoE) inference: We retrofitted the FlashMoE [3] megakernel MoE operator to support SSD offloading of expert weights, enabling an evaluation of CoPilotIO in MoE inference scenarios. FlashMoE was originally a fully GPU-resident distributed MoE operator that fuses gate computation, expert FFNs, and token dispatch/combine communication into a single persistent GPU kernel. It uses an actor model (Processor, Scheduler, Subscriber) to implement tile-level pipelined scheduling, eliminating multi-kernel launch overhead and hiding communication latency. We preserved this in-kernel scheduling framework while moving expert-weight storage from GPU memory to NVMe SSDs: when a Processor needs to execute a given expert's GEMM, it first issues an NVMe read request directly from the GPU to load the corresponding up-projection and down-projection weight matrices from the SSD into the GPU page cache, then executes the fused GEMM-activation-GEMM computation.

Experiments use a synthetic workload with 2,048 experts, hidden and intermediate dimensions of 2,048, Top-2 routing, sequence length 8,192, and capacity factor 1.0. Each expert's weights occupy 32 MB (16 MB each for up- and down-projections, TFloat32 format), totaling 64 GB for all experts. All experts are stored contiguously on the SSD. At the I/O layer, we configure 108 NVMe queue pairs, queue depth 128, and a 64 KB page size. Every expert corresponds to 512 pages, which are read cooperatively by 32 threads within a warp. After 32 warm-up iterations, we measure the average latency of 32 forward passes, comparing end-to-end inference performance between CoPilotIO and BaM. Similar to the GoFS case, AGILE does not natively support MoE workloads with unknown runtime access patterns.

Figure 11a shows the performance speedup of CoPilotIO over BaM. CoPilotIO achieves up to a 1.44× speedup. The improvement arises because MoE inference is a mixed compute–I/O workload: while some warps execute expert GEMMs, others must fetch the next expert's weights from the SSD. Under BaM's synchronous model, I/O warps stall and occupy GPU resources during NVMe polling, delaying compute warps and reducing overall SM utilization. CoPilotIO eliminates these stalls by offloading CQ polling to the CPU, enabling I/O warps to yield immediately after issuing requests. This allows the GPU scheduler to overlap expert computation with weight prefetching, thereby reducing end-to-end latency.

DLRM: We evaluate CoPilotIO on DLRM inference using the Criteo 1TB Click Logs dataset [11], following the experimental setup used in AGILE [39]. We construct the categorical feature vocabulary from the first three days of data and use cuBLAS for all dense matrix multiplications. Embedding vectors are fetched from the SSD through CoPilotIO, BaM, or AGILE, and integrated into the same CUDA stream pipeline as in AGILE for fair comparison. We adopt the standard DLRM architecture from [21], where the bottom MLP consists of three fully connected layers of size 512, while the top MLP consists of three fully connected layers of size 1,024 (referred to as Config-1 in AGILE). We use a batch size of 2,048 and measure the end-to-end inference time over 10,000 iterations. Unless otherwise specified, we use a 2 GB software cache with the clock replacement policy, 128 NVMe queue pairs, and a queue depth of 256 per queue, consistent with AGILE's configuration.

Figure 11b reports the end-to-end inference speedup normalized to BaM. CoPilotIO achieves a 1.85× speedup over BaM, outperforming AGILE's 1.41× speedup. The performance gain of CoPilotIO over AGILE stems from two factors. First, AGILE dedicates 4 SMs exclusively to I/O polling, which wastes GPU compute resources that could otherwise be used for MLP computation. Second, AGILE's I/O warps must still poll software lock flags to detect completion, introducing residual intra-warp stalls during embedding lookups. In contrast, CoPilotIO offloads all polling to the CPU and uses hardware barriers for completion notification, allowing I/O warps to fully overlap with cuBLAS kernels without occupying GPU execution resources.

6 Discussion

This section discusses the workload regimes where CoPilotIO provides the greatest benefit, as well as scenarios where its advantages are limited.

6.1 Beneficial Regimes

Compute-to-I/O ratio: The speedup from CoPilotIO scales with the fraction of execution time spent on GPU-side polling. In MoE, GEMM computation and GPU polling overhead account for approximately 32% of the total execution time. The remaining time is spent on storage I/O. CoPilotIO achieves a 1.44× speedup by eliminating the polling overhead. In contrast, DLRM exhibits a more balanced compute-to-I/O ratio:

embedding lookups are interleaved with MLP computation of comparable duration, allowing CoPilotIO to pipeline I/O with useful GPU computation. The freed GPU cycles are immediately consumed by MLP layers that would otherwise wait behind polling warps, yielding a higher 1.85× speedup.

GPU Resource Contention: GPU applications operate under high SM occupancy and massive warp concurrency, leading to intense contention for shared GPU resources, including warp slots, registers, shared memory, and memory bandwidth. In BaM, polling warps remain active on the SM while waiting for I/O completions, continuously occupying these shared resources and reducing the resources available for application kernels. For example, the additional GPU memory traffic generated by polling further contends with application memory accesses. In contrast, CoPilotIO significantly reduces GPU-side polling overhead due to our async I/O and split SQ/CQ design, leaving more SM resources and memory bandwidth available for concurrent application computation.

I/O concurrency and SSD utilization: CoPilotIO asynchronous completion model allows GPU warps to have multiple I/O requests in flight simultaneously, since warps do not block on CQ polling between submissions. This increases the effective NVMe queue depth seen by the SSD, enabling its internal NAND dies to operate in parallel. The benefit is most pronounced for workloads issuing many fine-grained requests (4 to 64 KB): in our microbenchmarks (Figure 7a), CoPilotIO sustains higher IOPS than BaM precisely because it maintains deeper queues without dedicating GPU threads to polling.

Application Domains: Beyond the applications evaluated in this paper, the architecture of CoPilotIO benefits a range of emerging GPU applications that combine I/O with computation. GPU-accelerated database engines (e.g., HeavyDB [14] and BlazingSQL [5]) perform table scans from SSD while executing filter, join, and aggregation operators on the GPU, a pattern where I/O and compute naturally overlap. Graph analytics frameworks (e.g., Emogi [20] and GIDS [26]) traverse out-of-core graphs by fetching neighbor lists from storage while processing vertices on the GPU. Retrieval-augmented generation (RAG) [17] systems fetch embedding vectors from SSD-resident indices during LLM inference, interleaving I/O with attention computation. KV cache offloading frameworks (e.g., FlexGen [31]) swap KV cache tensors between GPU memory and SSDs to serve long-context or large-batch LLM inference under limited GPU memory, generating bursty fine-grained I/O that competes with ongoing attention computation on the GPU. In each case, the application workload exhibits the characteristics identified above: mixed I/O and compute, high SM pressure from I/O management, and fine-grained access patterns requiring high queue depth.

6.2 Limited-Benefit Regimes

In contrast, CoPilotIO provides limited benefit in the following scenarios. First, purely compute-bound workloads spend little time waiting for storage operations. Since GPU resources are already dominated by application kernels rather than polling threads, removing polling contributes little to end-to-end performance. Second, workloads dominated by large sequential transfers typically require only modest queue depths to saturate SSD bandwidth. In these cases, storage efficiency is already high, and GPU polling overhead constitutes only a small fraction of execution time. As a result, the advantages of CoPilotIO asynchronous completion model and split SQ/CQ design are less pronounced. Finally, CoPilotIO relies on CPU threads for completion polling. When CPU resources are scarce or heavily utilized by other applications, CPU-side polling may become a bottleneck under high IOPS workloads. While adaptive co-polling can partially mitigate this issue by shifting polling work to the GPU, the resulting behavior increasingly resembles GPU-centric polling, thereby reducing the relative advantage of CoPilotIO.

7 Conclusion

In conclusion, CoPilotIO demonstrates that high-performance, on-demand GPU I/O does not require sacrificing GPU compute resources. By treating the CPU as an I/O co-pilot and introducing a fully asynchronous cross-device architecture, CoPilotIO eliminates GPU-side polling stalls while sustaining high throughput. Our evaluation shows that CoPilotIO significantly improves GPU compute efficiency, storage utilization, and end-to-end application performance.

Acknowledgments

We thank the OSDI '26 shepherd and reviewers for their insightful feedback, which helped improve the quality of this paper. We also thank the members of the HKUST(GZ) Systems Lab for their valuable comments and discussions. This research was supported by HKUST(GZ) startup funding. Jian Zhang is the corresponding author.

References

[1] Nvidia a100 tensor core gpu product brief (40gb hbm2). https://www.nvidia.com/en-us/data-center/a100/, 2020. 40 GB HBM2 memory configuration, accessed 2025-12-11.

[2] Nvidia h100 tensor core gpu product specifications. https://www.nvidia.com/en-us/data-center/h100/, 2022. 80 GB HBM3 memory, accessed 2025-12-11.

[3] Osayamen Jonathan Aimuyo, Byungsoo Oh, and Rachee Singh. FlashMoE: Fast distributed MoE in a single kernel. *arXiv preprint arXiv:2506.04667*, 2025.

[4] Joshua A Anderson, Chris D Lorenz, and Alex Travesset. General purpose molecular dynamics simulations fully implemented on graphics processing units. *Journal of Computational Physics*, 227(10):5342–5359, 2008.

[5] BlazingDB. BlazingSQL: A lightweight, GPU accelerated, SQL engine for python. https://github.com/BlazingDB/blazingsql, 2021. Accessed 2026-05-15.

[6] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language models are few-shot learners. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2020.

[7] Chia-Hao Chang, Vikram Sharma Mailthody, Jihoon Han, Zaid Qureshi, Anand Sivasubramaniam, and Wen-mei Hwu. GMT: GPU orchestrated memory tiering for the big data era. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, pages 464–478, 2024.

[8] Shuai Che, Michael Boyer, Jiayuan Meng, et al. Rodinia: A benchmark suite for heterogeneous computing. In *Proceedings of the IEEE International Symposium on Workload Characterization (IISWC)*, 2009.

[9] Aakanksha Chowdhery, Sharan Narang, Jacob Devlin, Maarten Bosma, Gaurav Mishra, Adam Roberts, Paul Barham, Hyung Won Chung, Charles Sutton, Sebastian Gehrmann, et al. Palm: Scaling language modeling with pathways. *Journal of Machine Learning Research*, 24(240):1–113, 2023.

[10] NVIDIA Corporation. Gdrcopy: A low-latency gpu memory copy library based on nvidia gpudirect rdma technology. 2023.

[11] Criteo AI Lab. Criteo 1tb click logs dataset. https://ailab.criteo.com/download-criteo-1tb-click-logs-dataset/, 2015. Accessed 2026-05-15.

[12] William Fedus, Barret Zoph, and Noam Shazeer. Switch transformers: Scaling to trillion parameter models with simple and efficient sparsity. *Journal of Machine Learning Research*, 23(120):1–39, 2022.

[13] Salman Habib, Joe Insley, David Daniel, Patricia Fasel, Zarija Lukic, Vitali Morozov, Nicholas Frontiere, Hal Finkel, Adrian Pope, Katrin Heitmann, Kalyan Kumaran, Venkatram Vishwanath, and Tom Peterka. HACC: Extreme scaling and performance across diverse architectures. *Communications of the ACM*, 60(1):97–104, 2017.

[14] HEAVY.AI. HeavyDB: Open source sql-based analytical database. https://github.com/heavyai/heavydb, 2026. Accessed 2026-05-15.

[15] Intel. Storage Performance Development Kit. http://www.spdk.io/.

[16] Thorsten Kurth, Sean Treichler, Joshua Romero, Mayur Mudigonda, Nathan Luehr, Everett Phillips, Ankur Mahesh, Michael Matheson, Jack Deslippe, Massimiliano Fatica, and Prabhat. Exascale deep learning for climate analytics. In *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 649–660, 2018.

[17] Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen-tau Yih, Tim Rocktäschel, et al. Retrieval-augmented generation for knowledge-intensive NLP tasks. In *Advances in Neural Information Processing Systems*, volume 33, pages 9459–9474, 2020.

[18] Shaobo Li, Yirui Eric Zhou, Yuqi Xue, Yuan Xu, and Jian Huang. Managing scalable direct storage accesses for GPUs with GoFS. In *Proceedings of the 31st ACM Symposium on Operating Systems Principles (SOSP '25)*, pages 979–995, Seoul, Republic of Korea, 2025.

[19] Pak Markthub, Mehmet E. Belviranli, Seyong Lee, Jeffrey S. Vetter, and Satoshi Matsuoka. DRAGON: Breaking GPU memory capacity limits with direct NVM access. In *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 414–426, 2018.

[20] Seung Won Min, Vikram Sharma Mailthody, Zaid Qureshi, Jinjun Xiong, Eiman Ebrahimi, and Wen-mei Hwu. Emogi: Efficient memory-access for out-of-memory graph-traversal in gpus. *Proceedings of the VLDB Endowment*, 14(2):114–127, 2021.

[21] Maxim Naumov, Dheevatsa Mudigere, Hao-Jun Michael Shi, Jianyu Huang, Narayanan Sundaraman, Jongsoo Park, Xiaodong Wang, Udit Gupta, Carole-Jean Wu, Alisson G. Azzolini, et al. Deep learning recommendation model for personalization and recommendation systems. *arXiv preprint arXiv:1906.00091*, 2019.

[22] NVIDIA Corporation. Nvidia a100 tensor core gpu architecture. https://resources.nvidia.com/en-us-tensor-core/nvidia-ampere-architecture-whitepaper, 2020. Whitepaper, accessed 2025-12-11.

[23] NVIDIA Corporation. Gpudirect storage: A direct path between storage and gpu memory. https://developer.nvidia.com/blog/gpudirect-storage/, 2022. Accessed 2026-05-15.

[24] NVIDIA Corporation. Nvidia blackwell architecture. https://resources.nvidia.com/en-us-blackwell-architecture, 2024. Accessed 2025-12-11.

[25] NVIDIA Corporation. Cuda runtime api documentation: Mapped host memory and zero-copy access. https://docs.nvidia.com/cuda/cuda-runtime-api/, 2025. Mapped Host Memory (cudaHostAllocMapped), accessed 2025-12-11.

[26] Jeongmin Brian Park, Vikram Sharma Mailthody, Zaid Qureshi, and Wen-mei Hwu. Accelerating sampling and aggregation operations in gnn frameworks with gpu initiated direct storage accesses. *Proceedings of the VLDB Endowment*, 17(6):1227–1240, 2024.

[27] Shi Qiu, Weinan Liu, Yifan Hu, Jianqin Yan, Zhirong Shen, Xin Yao, Renhai Chen, Gong Zhang, and Yiming Zhang. GeminiFS: A companion file system for GPUs. In *23rd USENIX Conference on File and Storage Technologies (FAST '25)*, pages 221–236, Santa Clara, CA, USA, February 2025.

[28] Zaid Qureshi, Vikram Sharma Mailthody, Isaac Gelado, Seung Won Min, Amna Masood, Jeongmin Park, Jinjun Xiong, C. J. Newburn, Dmitri Vainbrand, I-Hsin Chung, Michael Garland, William Dally, and Wen mei Hwu. GPU-initiated on-demand high-throughput storage access in the BaM system architecture. In *ASPLOS 2023 - Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, pages 325–339, 2023.

[29] Samyam Rajbhandari, Conglong Li, Zhewei Yao, Minjia Zhang, Reza Yazdani Aminabadi, Ammar Ahmad Awan, Jeff Rasley, and Yuxiong He. DeepSpeed-MoE: Advancing mixture-of-experts inference and training to power next-generation AI scale. In *Proceedings of the 39th International Conference on Machine Learning*, volume 162 of *Proceedings of Machine Learning Research*, pages 18332–18346. PMLR, 2022.

[30] Sagi Shahar, Shai Bergman, and Mark Silberstein. ActivePointers: A case for software address translation on GPUs. In *Proceedings of the 43rd International Symposium on Computer Architecture (ISCA)*, pages 596–608, 2016.

[31] Ying Sheng, Lianmin Zheng, Binhang Yuan, Zhuohan Li, Max Ryabinin, Beidi Chen, Percy Liang, Christopher Ré, Ion Stoica, and Ce Zhang. FlexGen: High-throughput generative inference of large language models with a single GPU. In *Proceedings of the 40th International Conference on Machine Learning*, 2023.

[32] Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. Megatron-lm: Training multi-billion parameter language models using model parallelism. *arXiv preprint arXiv:1909.08053*, 2019.

[33] Mark Silberstein, Bryan Ford, Idit Keidar, and Emmett Witchel. GPUfs: Integrating a file system with GPUs. In *Proceedings of the 18th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 485–498, 2013.

[34] Luca Soldaini, Rodney Kinney, Akshita Bhagia, Dustin Schwenk, David Atkinson, Russell Authur, Ben Bogin, Khyathi Chandu, Jennifer Dumas, Yanai Elazar, Valentin Hofmann, Ananya Harsh Jha, Sachin Kumar, Li Lucy, Xinxi Lyu, Nathan Lambert, Ian Magnusson, Jacob Morrison, Niklas Muennighoff, Aakanksha Naik, Crystal Nam, Matthew E. Peters, Abhilasha Ravichander, Kyle Richardson, Zejiang Shen, Emma Strubell, Nishant Subramani, Oyvind Tafjord, Evan Walsh, Luke Zettlemoyer, Noah A. Smith, Hannaneh Hajishirzi, Iz Beltagy, Dirk Groeneveld, Jesse Dodge, and Kyle Lo. Dolma: an open corpus of three trillion tokens for language model pretraining research. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (ACL)*, 2024.

[35] Ziyu Song, Jie Zhang, Jie Sun, Mo Sun, Zihan Yang, Zheng Zhang, Xuzheng Chen, Fei Wu, Huajin Tang, and Zeke Wang. CAM: Asynchronous GPU-initiated, CPU-managed SSD management for batching storage access. In *2025 IEEE 41st International Conference on Data Engineering (ICDE)*, pages 2309–2322, 2025.

[36] Yangzihao Wang, Andrew A. Davidson, Yuechao Pan, Yuduo Wu, Andy Riffel, and John D. Owens. Gunrock: A high-performance graph processing library on the GPU. In *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 11:1–11:12, 2016.

[37] NVM Express Workgroup. NVMExpress Specification. https://nvmexpress.org/resources/specifications/.

[38] Lingyun Yang, Yongchen Wang, Yinghao Yu, Qizhen Weng, Jianbo Dong, Kan Liu, Chi Zhang, Yanyi Zi, Hao Li, Zechao Zhang, Nan Wang, Yu Dong, Menglei Zheng, Lanlan Xi, Xiaowei Lu, Liang Ye, Guodong Yang, Binzhang Fu, Tao Lan, Liping Zhang, Lin Qu, and Wei Wang. GPU-Disaggregated serving for deep learning recommendation models at scale. In *22nd USENIX*

Symposium on Networked Systems Design and Implementation (NSDI 25), pages 847–863, Philadelphia, PA, April 2025. USENIX Association.

[39] Zhuoping Yang, Jinming Zhuang, Xingzhen Chen, Alex K. Jones, and Peipei Zhou. AGILE: Lightweight and efficient asynchronous GPU-SSD integration. In *The International Conference for High Performance Computing, Networking, Storage and Analysis (SC '25)*, pages 1028–1042, St. Louis, MO, USA, November 2025.

ARTIFACT EVALUATED
USENIX
AVAILABLE

RoCE BALBOA: Service-Enhanced RDMA Offload Engine for Data Center SmartNICs

Maximilian Jakob Heer, Benjamin Ramhorst, Yu Zhu, Luhao Liu, Zhiyi Hu, Jonas Dann, Gustavo Alonso
{maximilian.heer, benjamin.ramhorst, yu.zhu}@inf.ethz.ch
{luhliu, zhiyihu}@student.ethz.ch
{jonas.dann, alonso}@inf.ethz.ch
Systems Group, Dept. of Computer Science, ETH Zurich, Switzerland

Abstract

Remote Direct Memory Access (RDMA) has become the de facto standard for high-performance data center networking. However, current deployments rely heavily on fixed-function, commercial NICs. These "black box" commercial hardware implementations prevent researchers and system architects from modifying the transport layer for specialized tasks. In parallel, research on NICs often lacks offloaded networking stacks or uses simplified protocol implementations, limiting insight into novel networking solutions in realistic settings. In this paper, we bridge this gap by introducing BALBOA, an open-source, 100 Gbps RDMA offload engine designed for research on networking and fully compatible with commercial RNICs. Unlike prior stack implementations which lack scalability and bandwidth, or struggle with data center interoperability and miss strict protocol compliance, BALBOA supports hundreds of Queue Pairs in switched network environments and allows for line-rate offloads, making it a viable platform for realistic data center research. We describe the system architecture, detailing how BALBOA overcomes FPGA memory and timing bottlenecks through a decoupled state architecture and streaming control-data separation. We evaluate BALBOA on a hardware cluster with FPGAs, RNICs, and switches, showing that it matches the performance of commercial ASICs while offering full customization. Finally, we showcase BALBOA's potential through novel case studies: protocol enhancements for infrastructure purposes (encryption, deep packet inspection) and an offloaded preprocessing pipeline for deep learning recommender systems, which applies streaming transformations to the incoming data before feeding it directly to a GPU for model serving.

1 Introduction

With the ever-increasing scale of modern applications, such as large language models (LLMs) and big data analytics, computer systems have become increasingly constrained by network performance. This network bottleneck [101] has driven the widespread adoption of Remote Direct Memory Access (RDMA), which accounts for up to 70% of cloud network traffic [17]. By offloading the network stack to the NIC and bypassing the host operating system (OS), RDMA achieves low latency, high throughput, and reduced CPU utilization. These properties facilitate the implementation of novel platforms for memory disaggregation [5–7], distributed key-value stores [52], and data analysis [47]. However, such deployments often rely on commercial, fixed-function NICs, limiting customization for researchers. For example, having been adopted from trusted HPC domains, RDMA lacks access control and cryptographic policies [78, 85]. Furthermore, research has proposed a wide range of customized congestion control schemes for RDMA [19, 57, 71, 108], but commercial NICs typically only include one or two standard solutions. More generally, modifying the transport protocol or the packet-processing pipeline to address these security and architectural limitations or adapt it to a given application is nearly impossible on commercial NICs, limiting the options for experimental evaluation and prototyping.

To further address the network bottleneck and improve NIC programmability, recent work has shifted focus to SmartNICs and data processing units (DPUs), moving computation closer to the network. Commercial platforms such as AWS Nitro [82], NVIDIA BlueField [22], and Meta FBNIC [16] achieve high performance and improve programmability by enabling application offloads using NIC-hosted CPUs and on-datapath accelerators. However, such offloads often fall short of peak performance and require careful tuning [24]. Additionally, like traditional NICs, SmartNICs and DPUs implement the packet processing pipeline in hardware, remaining "black boxes" at the transport layer, thus limiting possibilities for customization.

FPGA-based platforms, both academic [33, 37, 62, 63, 65] and commercial [23, 32, 70], improve programmability and customizability. However, they suffer from low performance, limited scalability, and a lack of data center interoperability. For example, FPGA-based SmartNICs often lack an offloaded network stack [33, 98], relying on the host OS for the

packet processing pipeline, leading to increased latency and CPU utilization. Similarly, they also often fail to achieve high throughput (less than 100G) [68, 99] and compatibility with the specification [67, 84, 99], due to missing components that are complex to implement and affect performance such as the cyclic redundancy check (CRC) or retransmission logic.

In this paper, we bridge this gap with BALBOA, an open-source[1], 100G, RoCEv2-compatible RDMA stack designed for research flexibility without compromising performance or interoperability. As such, BALBOA enables the exploration of novel RDMA extensions and in-network computation, while additionally being usable as a plug-and-play component for SmartNICs. The paper makes the following contributions:

1. **High-performance architecture:** We propose RoCE BALBOA, an open-source, 100G, switched network-tested, and RoCE-v2 compatible RDMA stack for FPGAs that offers direct memory access to server-grade CPUs and GPUs [92]. It matches commercial NICs in both throughput and latency while offering flexibility for extensions and modifications on the hardware level. This demonstrates that the properties prior work treats as fundamentally in tension — 100G line rate, strict protocol compliance, and datapath extensibility — can be reconciled at the architecture level through decoupled state management and deep pipelining.
2. **Protocol evolution:** We demonstrate BALBOA's extensibility by implementing two line-rate protocol enhancements impossible on fixed-function NICs: inline Advanced Encryption Standard (AES) Counter Mode (CTR) encryption and ML-based deep packet inspection (DPI) for NIC-level access control. BALBOA thus enables the implementation of RDMA protocol modifications at line-rate, on a switch-compatible platform.
3. **Application acceleration:** We illustrate the performance benefits of on-datapath acceleration by deploying a commercial ML preprocessing pipeline for recommender models [109]. By utilizing BALBOA to preprocess and stream model data directly to the GPU, bypassing the host CPU, we achieve significant latency reductions and throughput improvements compared to the baseline.

2 Background and Motivation

This section reviews the RDMA protocol and motivates the need for an extensible stack with support for protocol enhancements and application acceleration at line rate.

2.1 The Rigidity of Commercial RDMA

RDMA achieves high throughput and low latency by offloading the transport layer to the NIC and utilizing OS-bypassing DMA [53]. Flows between remote nodes are managed via Queue Pairs (QP) as a connection abstraction and used for remote read and write accesses through either one- (`RDMA WRITE` and `RDMA READ`) or two-sided (`RDMA SEND` and `RDMA RECEIVE`) transactions [15, 66]. The central problem of commercial RDMA NICs (RNICs) is that the key logic is hardwired. Modifying the protocol pipeline to, for example, add a flow-tracking state machine, collect telemetry, or encrypt data to address some of the well-known security flaws of RDMA [78, 85, 89], requires modifying the transport logic. Because commercial silicon cannot be modified, these modifications have to be implemented either in software, thus reducing the performance benefits of RDMA and increasing CPU utilization, or on slow, off-datapath coprocessors [24] that add considerable latency.

2.2 The Case for Application Logic Offload

With ever-increasing amounts of data and improved accelerator performance, the gap between compute performance and network bandwidth is becoming more acute. At the same time, network virtualization is becoming increasingly expensive, consuming significant CPU cycles [32]. These shifts have motivated the offload of both network functions and application-specific logic [80] onto specialized accelerators. Recent examples range from data analytics [73] to ML preprocessing tasks [55], which can be implemented on both FPGA-based platforms [50, 63] as well as commodity SmartNICs and DPUs. The combination of SmartNICs and GPUs for ML-related use cases, such as offloaded object storage clients [51] or shared memory implementations [39], is currently of particular interest. However, as explained in Section 3, most of these examples suffer from critical shortcomings:

- SoC-based SmartNICs rely on embedded ARM cores for programmability in the off-datapath compute model. While flexible, they suffer from the "look-aside" bottleneck: processing a packet requires moving it to the core's memory hierarchy, incurring significant latency penalties, making line-rate processing at 100Gbps infeasible for complex applications [24].

- Existing FPGA prototypes for stream-oriented in-network compute either lack an offloaded network stack and rely on the host CPU, sacrificing performance and CPU utilization, or build on simplified network stacks that are not compatible with common data center NICs and accelerators.

In this paper, we demonstrate how BALBOA bridges this gap through an open and extensible architecture, allowing it to serve as a critical piece of research infrastructure and a building block for future SmartNICs and DPUs.

[1] https://github.com/fpgasystems/BALBOA_AE

Table 1: Feature overview and comparison of various networking stacks and RDMA NICs with BALBOA as a reference.

System	RoCE-v2 & NIC-compatible	100G-capable	Open source	Works with switches	Custom-izable stack	DMA to host	DMA to GPU	On-datapath acceleration	Protocol enhancements
PROP [99]	☐	☐	☐	☑	☑	☑	☐	☐	☐
Mansour [67]	⊞	☑	☐	☐	☑	☑	☐	☐	☐
StRoM [84]	☐	☑	☑	☐	☑	☑	☐	☑	☐
Schelten [26, 81]	☑	☑	☐	☑	☑	☐	☐	☑	☐
Sun et al. [88]	☑	☑	☐	☐	☑	☑	☐	☐	☐
Marini et al. [29, 68]	☑	⊞	⊞	⊞	☑	☑	☐	☐	☐
ETRNIC [97]	☐	☑	☐	☐	☐	☐	☐	⊞	☐
ERNIC [11, 107]	☑	☑	☐	☐	☐	☑	☐	☑	☐
ConnectX-5 [91]	☑	☑	☐	☑	☐	☑	☑	☐	☐
BlueField 3 [2]	☑	☑	☐	☑	☐	☑	☑	⊞	⊞
BALBOA	☑	☑	☑	☑	☑	☑	☑	☑	☑

☑ - satisfied, ⊞ - partially satisfied, ☐ - not satisfied

3 Related Work

RDMA is extensively used in HPC and data centers to implement systems such as distributed key-value stores [52], disaggregated memory [110], data analytics platforms [47], among others. These platforms are orthogonal to our efforts and can benefit from the stack offloading we propose. The same is true for FPGA-based SmartNICs [61, 62] which focus on offload management rather than the stack itself. We thus focus only on RDMA implementations and efforts that overlap with the use cases we explore in the paper.

On the commercial side, several products support RDMA. The Mellanox ConnectX-5 [91] is a standard 100G-capable RDMA NIC. A SmartNIC version of the system is NVIDIA's BlueField 3 [2], combining the network functionality with both flexible and hardened compute cores for on- and off-datapath offloads. Such commercial systems are useful, but limiting: the programmable on-datapath offloads are limited by access to cache and memory [24], while the hardened decompression core fails to meet line-rate throughput while entirely lacking support for compression [60]. Being a closed system, there is no way to either extend the system or address its limitations. This makes it difficult to explore logic suitable for offload and how to best implement it since the bottlenecks are inherent to the platform and cannot be addressed.

Table 1 summarizes RDMA implementations on FPGAs from academia and industry, highlighting their network capabilities, open-source availability, support for on-datapath acceleration, and communication with GPUs. One of the earliest approaches of RDMA on FPGAs [99] adopts the general principles of RDMA data transport with NIC-offloaded networking and DMA, but neither follows the actual protocol nor achieves 100G throughput. There are also solutions developed for very concrete use cases (e.g., high-energy physics or telescope data readouts) that demonstrate throughput performances of 100G [67] and 10G [29, 68] respectively, but do not implement `RDMA READ` and focus on embedded devices rather than the cloud and data center. For that purpose, several other solutions have been developed [26, 81, 84]. StRoM [84] is open-source, supports 100G, and has been used extensively for RoCE-enabled FPGA projects [27, 35, 42, 56, 77, 87], but misses key features. Due to its lack of high-performance checksum logic, it does not provide interoperability with commercial RNICs. Since its implementation of retransmission and handling of packet loss lack robustness, StRoM also does not support communication via switches. These two shortcomings eliminate StRoM as a platform for realistic data center network prototyping. Similarly, the lack of support for GPU-direct RDMA diminishes its applicability for modern AI-driven networks. Furthermore, the stack presented in [26, 81] provides a protocol-adherent RDMA implementation only for `RDMA WRITE` between FPGAs. Being closed-source and thus not extensible, the system makes it impossible to add missing functionality for `RDMA READ`, access to host memory, and RoCEv2 compliance. As such, the platform can enable data exchange between FPGAs but has limited applicability for network research in realistic data centers. Both StRoM [84] and the stack presented in [26, 81] explore offloading of application logic to the FPGA as we do here, but are limited due to the missing GPU-DMA-compatibility and the lack of demonstrated enhancements in the packet processing pipeline itself. Commercially, AMD's original implementation of RDMA [97] targeted mobile applications. The current version, ERNIC [11], is better suited for data center applications with 100G throughput. In [107], it was used to implement on-datapath offloads. However, the stack is oriented towards FPGA-to-FPGA communication and is closed-source, not allowing changes to the packet processing.

RoCE BALBOA aims to provide a comprehensive coverage of RDMA functionality and compatibility with commercial data center switches and NICs. Furthermore, it enables researchers to extend the protocol and add application offloads with direct access to GPU memory. Adding the missing functionality, while achieving high performance and enabling arbitrary offloads involves non-trivial design decisions that we discuss in this paper. A survey of existing network studies using FPGAs and custom SmartNICs reveals the gap that BALBOA tries to fill: many prominent FPGA-based NIC platforms, such

Table 2: Overview of the implemented features of BALBOA

Feature	BALBOA-implementation
Networking Speed	100G (200G theoretically possible)
RDMA Mode	Reliable Connection (RC)
Supported RDMA-verbs	RDMA WRITE and READ
Flow Control	ACK-clocked sliding window of static size
Retransmission Logic	Go-back-N, backed by on-device HBM
Multi-QP support	500 QPs by default, build-configurable

as Corundum [33], PANIC [61] and OpenNIC [98] do not include an offloaded networking stack, thus relying on the host CPU, while others, such as DUA [83], SuperNIC [62] and ACCL+ [42], implement parts of the packet processing pipeline or simplified protocols while focusing on function offloads. BALBOA is orthogonal to these works, aiming to offer a plug-and-play data center-compatible RDMA stack that provides connectivity and interoperability to these existing frameworks. Similarly, BALBOA could be used for the exploration of alternative transport stack principles as frequently discussed in research [46, 76, 96] through the customizability of its packet processing logic. The same is true for recent work focused on adding programmability and configurability to transport stacks [14, 21] and specifically RDMA [102]. Finally, BALBOA could be integrated with higher-level network management tools [59] due to its interoperability with commercial NICs.

4 RoCE BALBOA Architecture

4.1 Design Requirements and Challenges

Given the task to develop a widely usable RDMA stack for network research in data centers, we identify four key requirements that guide our design:

(R1) 100G Throughput: The stack should be able to satisfy the 100G networking throughput on current FPGA boards and offer the potential to be upgraded to 200G for the next generation of boards.

(R2) Protocol Adherence: To support integration into data center networks with commodity NICs and switches, the stack must adhere to the RoCEv2 standard and be interoperable with other networking cards. This adds significant complexity to the design, requiring, e.g., robust and fast retransmission, and the computationally complex CRC, which are both often ignored in existing systems.

(R3) Resource Efficiency: To provide space for application offloads and protocol enhancements, the resource and power consumption of the stack itself should be minimal.

(R4) Openness and Extensibility: Integrating offloads and protocol enhancements into the packet-processing pipeline calls for a degree of openness that fundamentally complicates the NIC design. To support such offloads, the stack must adopt a modular structure and expose well-defined, standardized interfaces between its components.

The design of a research-grade RDMA stack faces a fundamental trade-off: it must provide the transparency and modularity required for experimentation **(R4)** while still achieving high performance **(R1)** and protocol compliance **(R2)** available in commercial NICs. Standard open-source stacks often sacrifice **(R1)** and **(R2)** for **(R4)**. The core research contribution of BALBOA lies in demonstrating how openness and protocol-adherent performance can be satisfied at the same time through architectural design choices.

To achieve protocol compliance **(R2)** while maintaining a manageable development workload, we decided to limit BALBOA to a relevant subset of RoCEv2 features, deemed sufficient for data center deployment and research (Table 2). We implement the Reliable Connection (RC) mode with focus on one-sided RDMA verbs (`RDMA WRITE` and `RDMA READ`). As shown by prior works [8, 49, 93, 100, 111], the RC mode, together with one-sided verbs, is the most commonly used configuration in RDMA networks, enabling on-datapath offloads with minimal CPU involvement. Furthermore, RC mode demands the most complex hardware support among RoCEv2 transport types, encompassing retransmission, timeout handling, and in-order delivery guarantees. Supporting UC mode would follow as a straightforward extension, gating the reliability logic on QP type and extending the opcode decode tables to cover the UC opcode range — demonstrating that the design cleanly separates the reliability layer from the core datapath.

4.2 Architecture Overview

The core architecture of BALBOA (Figure 1) consists of several distinct building blocks for: ④ header (UDP, TCP, IB) processing, ③ checksum calculation, ② retransmission buffering, ① flow control, and ⑤ arbitration. These blocks communicate through well-defined, industry-standard AXI4-Stream interfaces for both data and control. From a development perspective, the modular design simplifies testing of the complex network stack by enabling stand-alone simulation and verification of individual components. At the same time, it improves accessibility, as components can be modified or extended independently, making the design suitable for future research **(R4)**. And from a performance point of view, the modular design lends itself to deep pipelining between components, helping to achieve timing closure with limited routing complexity **(R3)**.

Internally, BALBOA uses 512-bit–wide AXI data streams that operate at a system-wide clock frequency of 250 MHz. This configuration is sufficient to sustain an end-to-end throughput of 100 Gbps (250 MHz × 512 bit = 128 Gbps), satisfying the performance requirement **(R1)**. The chosen bus width and clock frequency create a balance between routing complexity and resource overhead compared to designs that would rely on wider datapaths or higher frequencies, addressing **(R3)**. The design also provides headroom for future

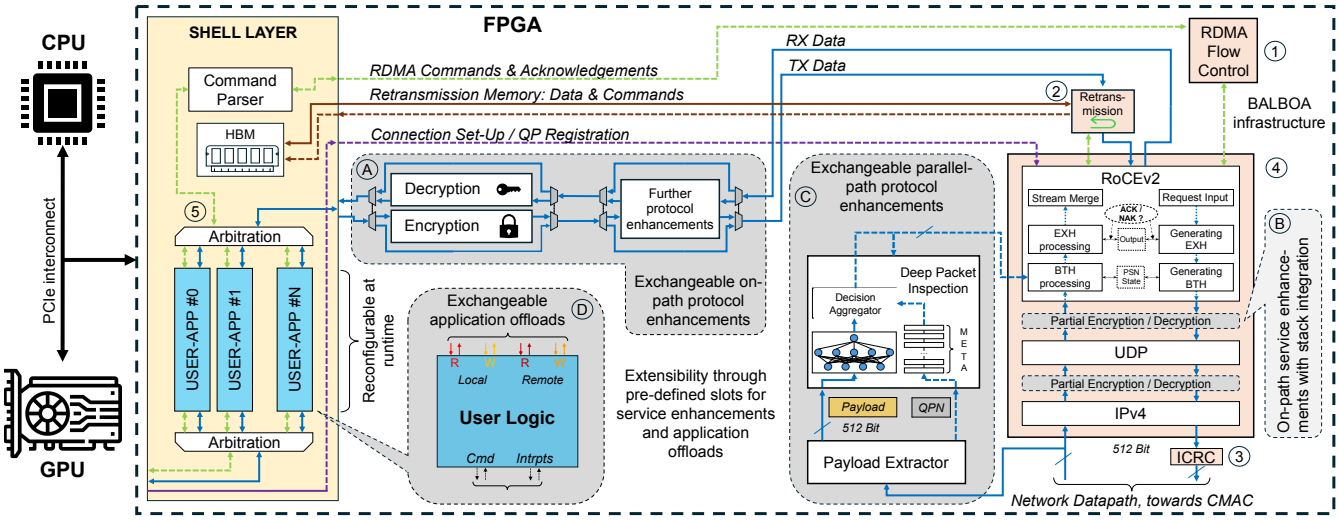


Figure 1: Overview of RoCE BALBOA, consisting of the customizable 100G RoCE packet processing pipelines ④ with surrounding infrastructure ①-③ and stack-adjacent slots for exchangeable on-datapath (Example: Ⓐ - AES CTR) and parallel-path service enhancements (Example: Ⓒ - ML-DPI). The open BALBOA-architecture also enables on-path enhancements within the packet processing pipeline (Ⓑ). Allows for DMA to both CPUs and GPUs with data routed through reconfigurable logic slots Ⓓ in the FPGA shell.

networking standards. The complete stack successfully synthesizes and meets timing at clock frequencies of up to 400 MHz, enabling a clear upgrade path towards 200 Gbps as suitable FPGA platforms become available.

By exposing industry-standard AXI4-Stream interfaces, RoCE BALBOA enables plug-and-play integration of custom offloads **(R4)** and allows it to be deployed in different FPGA platforms (shells). While this paper evaluates the stack in the open-source Coyote v2 shell [77] with the XDMA-core [12] for direct memory access to the host and the 100G CMAC-core [13] for network access, a port to the AMD RecoNIC/ONIC platform [107] has been realized in the past [69], demonstrating BALBOA's portability across platforms.

4.3 Packet Processing Pipeline

The core packet processing pipeline is implemented in AMD Vitis HLS using a vertically (per protocol-header) and horizontally (for RX / TX) partitioned architecture (Figure 1 - ④). To achieve 100Gbps throughput **(R1)** while ensuring maintainability and extensibility **(R4)**, we map the RoCEv2 header stack (IP, UDP, IB) to individual, pipelined HLS functions. This design pattern ensures sufficient readability and maintainability of the entire stack with the option of easily adding input parameters and processing logic in high-level description (as exemplified in subsubsection 5.2.2) to fulfill the goal of an open and extensible networking stack **(R4)**. In comparison, an RTL-based (e.g., Verilog, VHDL) design would make the effort for development, testing, and subsequent offload integration too demanding, especially for non-hardware engineers. Lastly, this separated design pattern enforces an a-priori pipeline architecture and allows the use of HLS auto-pipelining, helping with both the timing constraints for a 100G throughput design **(R1)** and the optimization of resource consumption **(R3)**.

A critical challenge in bidirectional 100G RDMA is maintaining QP-state consistency without stalling the pipeline. BALBOA solves this through a decoupled state architecture with physical state tables (Connection, PSN, MSN) implemented in dual-port BRAM and initialized during QP-setup. This design allows simultaneous access by both RX and TX pipelines **(R1)**, enabling independent line rate processing of incoming and outgoing network packets. By default, these tables support up to 500 QPs, but can be scaled depending on resource restrictions and use case (Table 3). A shared transport timer also measures packet timeouts to trigger retransmissions and potentially generate round-trip-time (RTT) based congestion signals for future extensions of the flow control.

Throughout the packet processing pipeline, BALBOA implements a versatile, three-fold interface with data, control, and completion streams: built around the notion of function offloads **(R4)**, split data and control streams allow placing processing pipelines directly on the datapath while manipulating and synchronizing the control commands in parallel, as opposed to a unified transmission where the command-manipulation, i.e. for size-expanding data operations, would be much more complicated. Building on this set of interfaces, the RX- and TX-path process traffic independently as follows:

- RX Path: Incoming headers are stripped and processed by dedicated finite state machines (FSMs) which

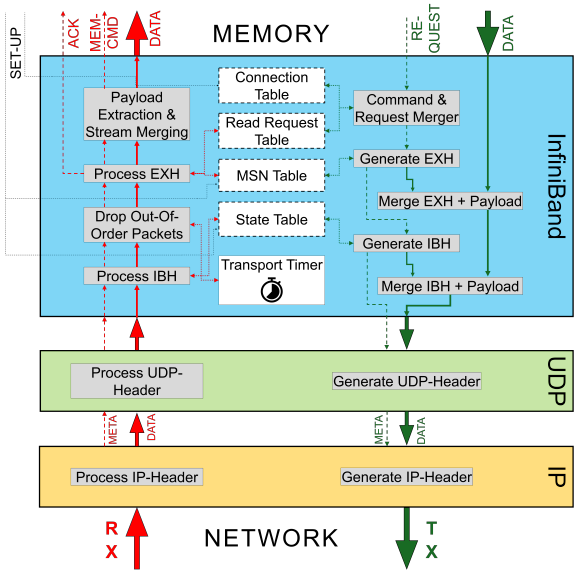


Figure 2: Packet processing pipeline for the sequence of IP, UDP, and InfiniBand headers of RoCE-v2 packets.

generate lightweight meta-commands for state retrieval and checks on parallel control buses. This isolates the main payload stream from the transport logic for improved timing and resource consumption (**R1, R3**). Crucially, BTH and RETH processing stages generate completion events on a dedicated bus, allowing the flow control logic to operate as a stand-alone, replaceable module (subsection 4.5).

- TX Path: The transmission pipeline accepts independent data and command streams from the host. This decoupling allows the architecture to absorb arbitration latency or retransmission fetches (Section 4.4) without stalling header generation. The command flow retrieves QPN/PSN state to generate headers (RETH/BTH), which are stepwise interleaved with the payload stream in the pipeline before the finished packet is forwarded to the checksum calculation (ICRC) and CMAC-unit for transmission.

4.4 Retransmission Buffers and Stream Multiplexer

The retransmission and stream multiplexer module (Figure 3 - ②) manages data delivery to the stack and buffers packets for potential retransmissions. A core challenge in RDMA stack implementations is the inherently opposing nature of local `RDMA WRITE` and remote `RDMA READ` operations. Delivering data for both over a single stream can cause stalls, preventing full throughput saturation. By separating data delivery for `RDMA WRITE`s and `RDMA READ`s onto distinct AXI streams, BALBOA manages to achieve bidirectional 100G throughput **(R1)** while also improving extensibility **(R4)**. More specifically, different offloads for `READ` and `WRITE` can be deployed on separate streams, enabling advanced enhancements and experimentation with the stack design.

Within the retransmission module, the two incoming buses for `READ RESPONSE`s and `WRITE`s are demultiplexed into a single stream via arbitration, while all incoming payloads are buffered in a dedicated, directly exposed HBM channel. Payloads, whether incoming from host or reloaded from HBM, are picked up and released by incoming commands from the packet processing pipeline.

If retransmissions are required due to packet timeouts or PSN sequence errors, the required payload cannot be refetched from the host, since an additional PCIe transfer would lead to unacceptable latency overheads. At the same time, on-chip SRAM is insufficient for buffering the bandwidth-delay product of a 100G link. Instead, all transmitted payloads need to be buffered in directly accessible fast card memory (HBM) until the remote side acknowledges receipt. This design principle is also backed up by the initial idea of adhering to the RoCE v2 standard **(R2)**: BALBOA implements Go-Back-N retransmission as defined in the protocol for RC-mode, and is compatible with off-the-shelf NICs, which means that up to N outstanding packets at any point in time need to be buffered for potential retransmission.

To explore the efficiency of the design, we analyze the resource implications for the Alveo U55C platform used for the evaluation: To saturate the bandwidth-delay-product, the flow control allows up to $N = 16$ outstanding packets per flow, which amounts to a retransmission buffer footprint of 64kB given the standard 4k MTU of RoCE. The FPGA provides 32 GB of fast off-chip HBM, offering a theoretical capacity for up to 500,000 QPs and thus preventing buffer overflows even for very large-scale deployments. In fact, this buffer capacity provides an advantage for an FPGA-based evaluation platform compared to commercial RNICs, which are typically limited to a few MB of on-chip SRAM [96, 106]. The practicability of this HBM-based approach to retransmission is further demonstrated under packet loss: The design only requires 1.732 μs to fully retrieve a buffered MTU-sized payload from memory, with an overall latency of 1.86 μs from the first indication of required retransmission to transmitting the fully formed packet towards the Ethernet-block of the design. This timespan is only a fraction of the round-trip time for packets of this size.

4.5 Flow Control Logic

BALBOA uses an ACK-based flow control over a fixed-size sliding window of outstanding packets, which are identified by their PSNs. This design enables timeout- and PSN-triggered Go-back-N retransmission while remaining compatible with commodity NICs. The flow control logic resides in the con-

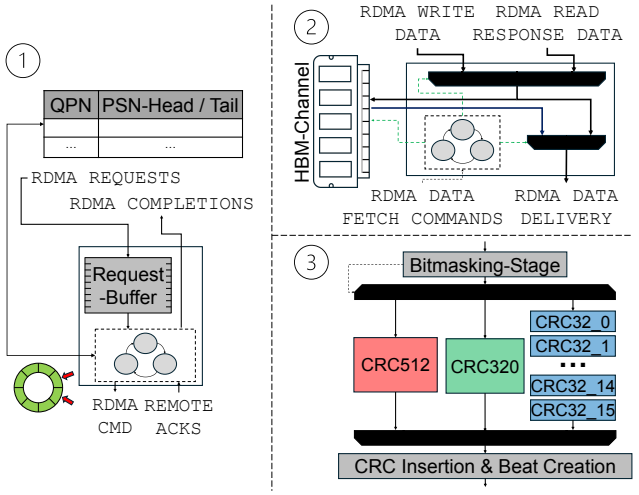


Figure 3: Building blocks of BALBOA: ① - ACK-clocked flow control, ② - Retransmission logic, ③ - ICRC pipeline

trol path (Figure 3 - ①), where incoming RDMA requests are either forwarded to the packet-processing pipeline or queued to limit the number of outstanding packets. The remaining packet budget per QP is stored in BRAM and updated; decreased when new requests are sent and increased upon receipt of RDMA ACKs.

By implementing flow control as a standalone block separate from the packet-processing pipeline (Figure 1 - ④) with standardized interfaces, BALBOA makes it easy to integrate alternative congestion control algorithms **(R4)**. Algorithms like DCQCN [108] or TIMELY [71] can simply replace the current implementation and throttle the command flow to the stack. Another reason for the modular design is to leverage partial reconfiguration on FPGAs, which, in the future, would enable programmable congestion control by dynamically exchanging the flow control module during runtime.

In addition to BALBOA's flow control, the underlying CMAC core on the target platforms natively supports Priority Flow Control (PFC) as specified by the RoCE standard [13], providing compatibility with this aspect of the protocol definition **(R2)**.

4.6 ICRC Pipeline

As described in [81], efficient calculation of the ICRC checksum is performance-critical for achieving high throughput **(R1)** and a well-known challenge for any RoCE implementation on FPGAs due to its combinatorial complexity. Since off-chip memory access to pre-calculated values would incur a significant latency penalty, impacting end-to-end throughput, RoCE BALBOA reimplements the concepts presented in [81]. More specifically, we calculate and concatenate the partial checksum of every incoming AXI beat within a packet. As a further trade-off between processing speed and resource consumption, we follow the referenced work in implementing separate, parallel pipelines for the full 512-bit and the partial 320-bit AXI beats, which are the most common fill-levels in RoCE packets on a 512-bit bus. A third, multi-stage pipeline for chunks of 32 bits length allows one to calculate the required checksum for any possible AXI beat, given the guaranteed 4-byte alignment of all RoCE messages.

4.7 Traffic Sniffer as a Debugging Utility

Developing a full FPGA network stack is challenging, especially when debugging faults that appear only during long-running jobs. To address this and maintain sustainable development, we built a performance-preserving traffic sniffer for bidirectional 100G RX/TX paths. It captures all packets and outputs standard PCAP files suitable for analysis with tools such as Wireshark. A header-based filter at the CMAC-level allows users to capture only specific protocols (e.g., RoCEv2) or omit payloads to reduce memory usage, which is crucial for tracing lengthy flows. This tool proved invaluable during development and supports our goal of comparability with off-the-shelf NICs and deployment in realistic networks **(R2)**, where traffic capture is essential for tuning and debugging.

5 Compute Offloads in BALBOA

The open design of BALBOA enables two types of offloads: protocol enhancements and application-specific offloads in configurable logic slots on the datapath.

5.1 Design Methodology for Offloads

For both types of offloads, developers can choose from a diverse toolbox of development strategies with specific trade-offs. Thus, we now discuss the design methodologies available in the framework and their implications for design effort, latency, and resource footprint. BALBOA's implementation-agnostic stream interfaces open a wide design space around the tradeoffs between development effort versus control. At the lowest level, RTL (VHDL / (System)Verilog) as showcased in Section 5.2.1 gives maximum control over timing and resources but requires hardware expertise and typically weeks of development per module. HLS (e.g., Vitis HLS) allows the direct synthesis of C/C++ expressions, reducing the development effort to days and enabling non-hardware engineers to leverage BALBOA's capabilities. The usefulness of this approach is demonstrated in Section 8. Finally, tools like hls4ml [30], showcased in Section 5.2.2, or P4-to-hardware compilers (e.g., Vitis Networking P4 [3]) take a cross-disciplinary approach and eliminate the manual creation of hardware by compiling pretrained ML-models or P4-scripts directly into FPGA-images, reducing the development time.

Regardless of the implementation methodology, coordination with the packet processing pipeline for on-datapath

offloads reduces to a backpressure signal on the AXI4-stream signal as a consequence of the realized design goal **(R4)**. As laid out in Section 5.3, RX-path offloads must be non-stalling, while TX-path offloads benefit from the decoupled command/data architecture. Parallel-path enhancements need to interact with the processing pipeline through HLS-hooks, requiring more attention to potential pipeline stalls.

5.2 Protocol Enhancements as a Service

The RDMA protocol enhancements are located close to the stack and interact directly with both data- and control flow. Compared to application-specific offloads, protocol enhancements implement network-specific tasks applicable to the entire BALBOA-stack. Function offloads as user-defined applications for selected QPs (e.g., data preprocessing) can select to utilize these protocol enhancements as a service of the network stack, just like the general RDMA transport mechanism. In our example, we focus on the aforementioned security flaws of RoCE and demonstrate the unique capability of BALBOA to deploy enhancements that close important security gaps of the protocol. In general, two forms of protocol enhancements are possible:

On-datapath protocol enhancements are located right before or after the packet processing pipeline and consume the incoming AXI data streams, without interfering with the actual protocol logic of the packet processing pipeline (Figure 1 - Ⓐ). While we only show host-facing offloads located after the BALBOA-stack, future work could also place additional logic on the network streams leading to the packet processing pipeline, such as a network impairment model for congestion control testing.

Contrary to this, *parallel-path protocol enhancements* (Figure 1 - Ⓒ) are placed in parallel to the core packet processing pipeline. These enhancements operate on a multiplexed copy of the incoming or outgoing packets and can directly interact with the packet processing logic via interfaces to the pipeline. The latency of these protocol enhancement services is hidden by the partial latency of the parallel packet processing pipeline. Parallel-path protocol enhancements are unique to an open stack design as demonstrated with BALBOA **(R4)**, since they require direct integration into the protocol logic.

5.2.1 AES Encryption

To demonstrate secure, line-rate stream processing, we integrated an open-source AES-CTR engine[2] (Figure 1 - Ⓐ). The module operates as a 'bump-in-the-wire' on the AXI datapath, adding a deterministic latency of only 11 cycles (44ns) while sustaining full 100Gbps throughput.

BALBOA's architecture provides a critical advantage for cryptographic correctness: unlike fixed-function NICs where internal state is hidden, our stack exposes the transport state machine to the encryption logic. This gives us the opportunity to construct a cryptographically unique counter (IV) for AES-CTR by combining the Queue Pair Number (QPN) with per-packet counters. This state-aware synchronization ensures semantic security without requiring software intervention or side-channel signaling.

Furthermore, BALBOA's modular AXI-based design allows for flexible placement of encryption blocks. Beyond standard payload encryption, the engine can be repositioned within the packet processing pipeline (Figure 1 - Ⓑ) to obfuscate transport headers (e.g., BTH, UDP) for stealth communication within trusted subnets, or extended to protect specific metadata fields. Similar concepts have been discussed in research before [90] and are currently seeing deployment in production environments (e.g., NVIDIA / Google PSP [36, 75, 86]).

5.2.2 ML-based Deep Packet Inspection

As an example of parallel-path protocol enhancements (Figure 1 - Ⓒ), RoCE BALBOA deploys an open-source machine learning-based deep packet inspection module [44]. It leverages low latency inference of ML on FPGAs to offload packet checking to the NIC and identify, at line-rate, potentially malicious executables embedded in RDMA flows, countering the well-known shortcomings of the protocol in terms of access control. This maintains host bypassing while marking and rejecting potentially malicious packets before they ever reach host memory and possibly inflict damage. The ML model was trained to distinguish between common payloads, such as CSVs, PNGs, and TXTs, and compiled malware executables. The trained and quantized model was converted to a hardware module suitable for FPGA inference through the hls4ml compiler [30]. The end-to-end inference latency for DPI decisions is only 44ns per AXI beat. This is shorter than the latency of the packet processing pipeline and, therefore, DPI does not lead to performance degradation. Utilizing the open stack design, the aggregated DPI decision is communicated as an input value to the BTH-processing function of the packet processing pipeline and included as a flag in the resulting command to the host. This allows the stack to raise an interrupt, upon which the host OS could decide to run additional checks. In more conservative setups, packets can also be immediately dropped by the NIC, before reaching the host.

The integration of the DPI model into the packet processing logic is a prime example of a protocol enhancement only possible with a customizable and open-source design like BALBOA **(R4)**. Following this design principle, other modules can be installed in a similar position for, e.g., data analytics on incoming packets to control the further processing in the stack.

[2] https://github.com/fpgasystems/hw-acceleration-of-compression-and-crypto

5.3 On-datapath Application-specific Offload

Thanks to its generic AXI stream interfaces, RoCE BALBOA allows users to deploy arbitrary logic on the RX- and TX-datapath to manipulate payloads. Multiple design criteria must be met for such offloads:

- For both directions, special care is required for any data expansion operation on the payloads. Backpressure is exerted if an offload exceeds the available bandwidth.
- Similarly, all operations placed on the RX datapath for incoming packets must not stall at any point to avoid causing backpressure to the network stack as it would lead to packet drops. For offloads below line rate, this requires an end-to-end approach with adapted data sending rate from the remote node.

At the same time, the user finds powerful design tools for customized data processing in the application-specific on-datapath offloads:

- The user has access to both the data and control flow for incoming packets. This allows manipulating the local data forwarding, e.g., scatter-gather operations to multiple host memory locations for arriving data. On top of that, payloads can also be forwarded to local NIC-attached memory for buffering.
- Finally, also stateful offload operators can be configured by the user: By using memory-mapped control registers, the user is able to provide context or state for processing to such hardware modules.

To facilitate development against these design constraints, we provide a comprehensive logic simulation framework that tests any on-datapath user-application against the realistic network behavior of RoCE BALBOA and helps to easily spot throughput bottlenecks, data dependencies, and race conditions while hiding the actual complexity of the implementation. In fact, the workflow for deployment and simulations only differs by a single compile flag and does not require any framework or code adaptation. This makes it possible to design and test advanced on-datapath compute offloads. As a consequence, BALBOA has been proven useful and easy to handle for developers in many collaborative projects with both industry and academia in the past, where it was deployed in research and production systems.

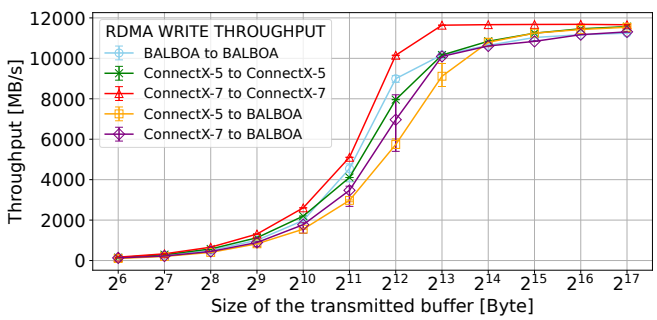
6 Experimental Evaluation

We evaluate RoCE BALBOA along multiple dimensions in a public research cluster [72] that resembles a typical data center network. The 100G subnet connects servers with Mellanox ConnectX-5 (MCX515A-CCAT) and NVIDIA ConnectX-7 (integrated in a BlueField-3 B3220-DPU), as commodity NICs, and BALBOA deployed on AMD Alveo U55C FPGA accelerators as PCIe cards over CISCO Nexus 9000 Series switches, tuned to 4K maximum transmission units (MTU). We evaluate the performance between RoCE BALBOA on the Alveo FPGAs and the commercial NICs through a single layer of switches, but also verify the same results in a 2-tier fat-tree network topology more common for HPC-oriented cloud setups. Additionally, we have successfully deployed and tested BALBOA on other AMD Alveo accelerators with CMAC blocks (U250, U280).

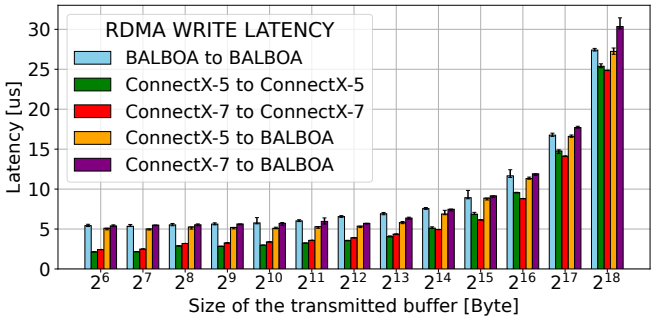
6.1 Basic Performance & Cross-compatibility

RoCE BALBOA is evaluated in the described setup for latency and throughput at various buffer sizes, both in FPGA-to-FPGA flows as well as heterogeneous FPGA-to-Mellanox connections (Figure 4). Additionally, Mellanox-to-Mellanox results are plotted for comparison. Following the general practice of network performance testing, latency values are obtained through repeated single buffer transmissions and completion-polling, while throughput performance is measured with repeated batched transmissions.

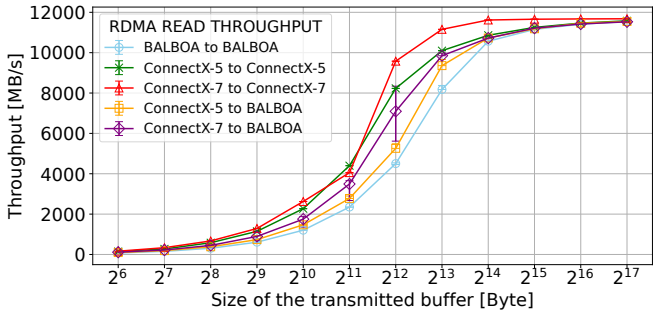
When first comparing only the BALBOA-to-BALBOA performance with the NIC-to-NIC performance, it becomes evident that both BALBOA and the commodity NICs perform similarly, especially for `RDMA WRITE` operations. As shown in Figure 4a, all three devices reach the 100G saturation point for 32kB buffers at the latest, with a marginal advantage for the newer generation ConnectX-7. In terms of latency (Figure 4b), the higher internal clock frequency of the ASIC-based NIC results in a latency advantage, especially for small buffers, with diminishing differences for larger messages. All three devices have a steady latency performance with a negligible tail latency as shown by the 95th-percentile lines. The same general observations hold true for the `RDMA READ` behavior. A more pronounced throughput difference for medium-sized buffers becomes visible for `RDMA READ` (Figure 4c). Unlike `RDMA WRITE` operations which utilize posted PCIe transactions (fire-and-forget), `RDMA READ` relies on non-posted memory read requests. This requires the NIC to manage the round-trip latency of fetching data from the host root complex. To saturate the link, the NIC must maintain a high volume of outstanding split-transactions (PCIe tags) to mask this latency. The commercial ASIC, utilizing a custom controller clocked at $\approx$1GHz, can issue and track these requests at a higher rate than the FPGA. In BALBOA, the lower logic frequency (250 MHz) and the interface overhead of the generic PCIe hard-block introduce slight serialization delays in the request-issue loop, preventing full saturation for medium-sized transfers where the per-transaction overhead is most pronounced. Heterogeneous connections between BALBOA and an ASIC-based NIC exhibit characteristics of both devices as described before. The measurements demonstrate that RoCE BALBOA is capable of saturating a 100G link for both `RDMA READ` and



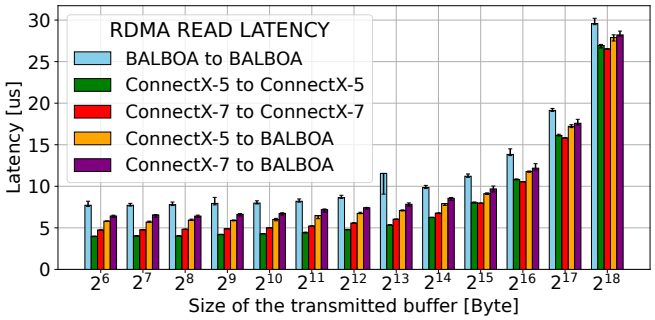
(a) Throughput evaluation for `RDMA WRITE`.



(b) Latency evaluation for `RDMA WRITE`.



(c) Throughput evaluation for `RDMA READ`.



(d) Latency evaluation for `RDMA READ`.

Figure 4: Performance evaluation of RoCE BALBOA in a switched 100G data center network, 100 repetitions per measurement.

`WRITE` in connections to both another FPGA and a commodity NIC. The adaptability across different generations of ASIC-based RNICs appears even more crucial, as such connections are known to be troublesome even in commercial deployments [59]. The experiment validates the design choices for achieving the network bandwidth **(R1)** and compatibility of the stack with commodity hardware **(R2)**.

6.2 Multi-QP Performance Scaling

RoCE BALBOA can serve up to hundreds of independent QPs in parallel from independent threads. By intertwining batched transmissions of multiple QPs, we demonstrate that the arbiters used in RoCE BALBOA are capable of evenly distributing the available bandwidth between the competing flows (Figure 5), while maintaining full aggregated throughput saturation. Results obtained with a Mellanox card show similar behavior to BALBOA.

6.3 Protocol Enhancements: Encryption and DPI in BALBOA and Other Platforms

We showcase the performance of encryption and DPI and, if applicable, compare them to software-based implementations of the same functionalities on the host CPU (AMD EPYC 7302P, 16 cores, 64GB RAM) coupled to a Mellanox ConnectX-5. While the processing steps in RoCE BALBOA

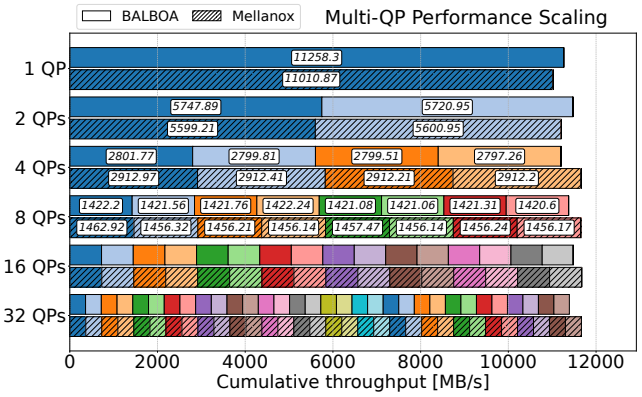
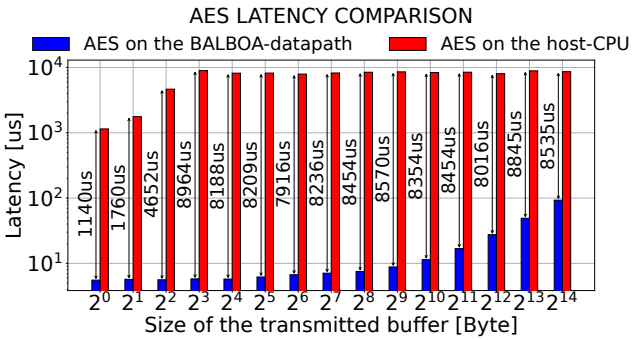


Figure 5: Bandwidth distribution and aggregation across multiple QPs for interleaved `RDMA READ`s of 32k size.

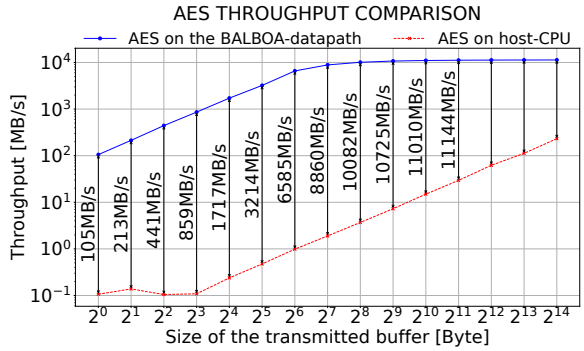
reside on the datapath and are therefore executed automatically, the execution on the CPU is triggered via a remotely written doorbell register with host-polling.

6.3.1 AES-CTR Encryption on RDMA Traffic

The direct comparison of latency and throughput for both the on-datapath BALBOA implementation and the realization of AES CTR on the host CPUs demonstrates a stark performance difference (Figure 6). Even with all 16 cores of the server-grade CPU (AMD EPYC 7302P) and with



(a) Latency comparison for AES-CTR on the BALBOA datapath vs. on the host CPU with a Mellanox NIC for RDMA WRITEs.



(b) Throughput comparison for AES-CTR on the BALBOA datapath vs. on the host CPU with a Mellanox NIC for RDMA WRITEs.

Figure 6: Comparison of AES implementations residing on either the BALBOA-datapath or the host CPU.

hardware support for AES via the standard OpenSSL library, the software-based implementation is only able to saturate a small fraction of the 100G network bandwidth (Figure 6b). Conversely, the hardware-based implementation in BALBOA achieves full throughput and only adds negligible additional latency - the AES pipeline has an end-to-end latency of 11 clock cycles/44 ns (Figure 6a). This performance difference is further backed up by existing research on accelerator-based encryption, where these solutions consistently outperform CPUs [25], while also reducing the "data center tax" of continuously thrashing CPU cores for repetitive tasks.

For completeness, we also tried to compare to the performance achievable with the hardware encryption accelerators in NVIDIA BlueField-3 DPUs, but ran into difficulties: The officially documented path for hardware-accelerated IPsec on BlueField-3 relies on a VXLAN+IPsec tunnel architecture via a patched strongSwan, which is not directly comparable to BALBOA's per-flow RDMA payload encryption. We were unable to build the patched strongSwan instance due to its sparse documentation and dependency as well as timestamp issues. There is no evidence that another per-flow RDMA encryption is supported in any documented configuration. Requests for assistance on the NVIDIA developer forum went

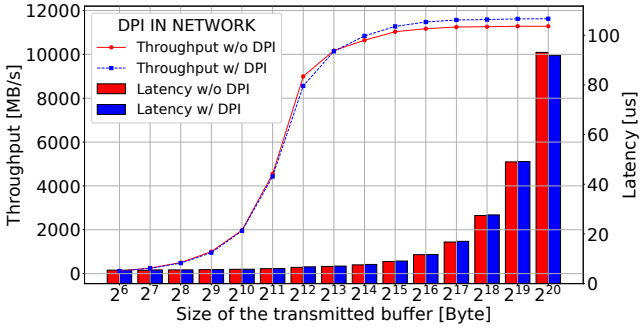


Figure 7: Throughput and latency with and without deployed ML model for deep packet inspection.

unanswered beyond a referral to enterprise support, and independent community reports confirm the same unresolved issues as recently as late 2025.

6.3.2 Deep Packet Inspection

Analyzing the performance of the DPI enhancement, we first observe that the deployed fully-connected ternary neural network provides an excellent detection rate of executables with 97.83% for full payloads and 89.35% for partially embedded executables. Compared to the low false positive rate of flagging acceptable payloads of the model, a highly effective and fine-grained differentiation policy is possible based on the ML-decisions. More details on the accuracy and performance of this model can be found in [44].

At the same time, Figure 7 shows that the deployment of the ML model alongside the BALBOA stack does not have a negative impact on either end-to-end latency or throughput, since the inference latency of DPI is completely hidden by the parallel packet processing pipeline.

7 Hardware Resource Evaluation

We analyze the FPGA resource utilization and power consumption with post-routing hardware and power estimation results for an AMD Alveo U55C FPGA (Table 3). The design aims to minimize the "transport tax" - the resources consumed by the network stack - to maximize the area available for offloads and enhancements:

- **Resource Efficiency (R3):** The base BALBOA stack utilizes only 3.4% of the Look-Up-Tables (LUTs) and 5.1% of Block RAM (BRAM). Notably, the complex transport logic fits entirely within on-chip memory, ensuring deterministic access times without consuming external HBM bandwidth for state management. Even with the AES-Encryption and ML-DPI offloads enabled, the total LUT utilization remains below 12.15%. This confirms the successful combination of resource-efficient and still extensible stack architecture achieved with BALBOA.

Table 3: Resource and power estimation breakdown for RoCE BALBOA, and for the AES and DPI service offloads.

	LUTs	BRAM	FFs	Power
BALBOA	43732 [3.4%]	101 [5.1%]	102988 [4%]	1.745 W
ICRC	12462 [0.96%]	0 [0%]	23942 [0.9%]	0.233 W
Retrans-Logic	587 [0.05%]	34 [1.7%]	1157 [0.04%]	0.073 W
Flow Control	787 [0.06%]	1 [0.05%]	2836 [0.1%]	0.022 W
Packet Process.	29897 [2.3%]	66 [3.4%]	75053 [2.9%]	1.416 W
- Conn Table	64 [0.01%]	4 [0.2%]	230 [0.01%]	0.014 W
- State Table	202 [0.02%]	6 [0.3%]	265 [0.01%]	0.014 W
- Trans. Timer	209 [0.02%]	2 [0.1%]	248 [0.01%]	0.022 W
- MSN Table	92 [0.01%]	5 [0.25%]	373 [0.01%]	0.016 W
AES Crypto	65662 [5%]	0 [0%]	11669 [0.5%]	0.356 W
ML DPI	54404 [3.75%]	0 [0%]	40508 [1.6%]	3.231 W
Payload Ext.	6252 [0.09%]	0 [0%]	1135 [0.04%]	0.072 W
ML Model	48152 [3.66%]	0 [0%]	39373 [1.5%]	3.159 W

- **Power Consumption:** The estimated power draw for the full 100Gbps design is $1.745W$, validating that FPGA-based transport is power-efficient enough for dense data center deployment.

- **Scalability:** Given the low LUT utilization, the design would theoretically allow for multiple parallel BALBOA instances of a single data center FPGA. This would enable multi-port 200G or 400G deployments in future work, limited only by the available PCIe-bandwidth for host communication.

To contextualize these numbers, a direct comparison with an established open-source TCP stack is instructive: Limago [41, 79] implements simplified 100G TCP and has been widely used in the community for network research [43] and distributed applications [9, 40, 48, 54, 94, 95]. Despite implementing full transport-layer reliability — including Go-Back-N retransmission and flow control as required by RoCEv2 — BALBOA consumes 18% fewer LUTs than the Limago 100G TCP stack under equivalent conditions (capacity for 500 flows, targeting AMD Alveo U55C), the primary scarce resource for co-located applications as evidenced by the AES and DPI offloads in Table 3. BALBOA uses more FFs (102,988 vs. 72,974) and BRAM (101 vs. ∼78 BRAM36-equivalent), though both remain comfortably within the available budget of the U55C and do not constrain placement of additional logic.

8 Use Case Example: ML Preprocessing Pipelines for RDMA-to-GPU

Deep learning recommendation models (DLRM) comprise a large fraction of all ML workloads in data centers [38]. To maintain accuracy of the recommendations during data drift and evolution, frequent online re-trainings are necessary [20, 31]. While the multi-modal input data consists of images, audio, or text, the training process itself operates on vector embeddings [105], so a preprocessing step for conversion is required. Traditionally, such training systems utilize a hybrid hardware configuration with CPUs and GPUs [104], as the GPU utilization should not be bloated with frequent kernel launches for preprocessing. When transferring data from disaggregated storage to the GPUs for DLRM-training, preprocessing becomes a critical bottleneck, as more and more CPU cores are required to saturate the increasing bandwidth of GPUs, leading to more than 60% share of total power consumption for just this initial data preparation step [18, 104]. This motivates an accelerated approach which combines network and compute using the capabilities of BALBOA. We demonstrate how to significantly increase the efficiency and the processing speed of DLRM preprocessing by offloading it onto the BALBOA datapath in deep pipelines operating at line-rate and combining it with the direct-to-GPU capability. Thus, the CPU as a potential bottleneck is completely bypassed.

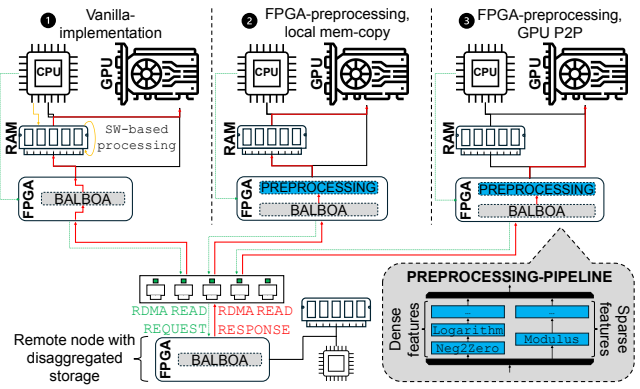


Figure 8: Comparison of three setups: ① - CPU-based preprocessing, ② - preprocessing on the FPGA, mem-copy to GPU, ③ - preprocessing on the FPGA and DMA to the GPU

8.1 Functional and Design Overview

For a realistic evaluation, we deploy operators from Meta's training toolkit [1, 74] implemented in deep pipelines for non-stalling line-rate execution in hardware [109]. We select the following three stateless operators and concatenate them to a preprocessing pipeline, as they are deemed relevant for optimization in hardware [58] and software [103]: `Neg2Zero` and `Logarithm` operate on dense input features by clipping negative values to zero and compressing very large values. `Modulus` restricts the range of sparse feature values. With an initiation interval of one clock cycle each, pipelines composed of these operators are able to maintain network speed and do not bottleneck the BALBOA datapath.

8.2 Performance Benchmarking

To explore the performance of the preprocessing offload and the direct memory access to the GPU, we conduct three experiments (Figure 8). In all three cases, the local node reads

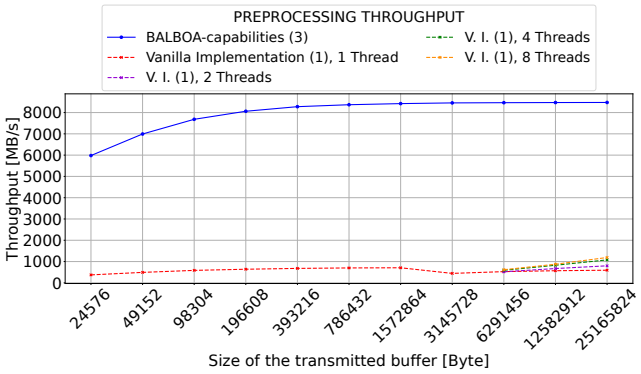


Figure 9: Throughput of preprocessing in BALBOA with direct-to-GPU vs. CPU preprocessing and copy to GPU for different levels of multithreading.

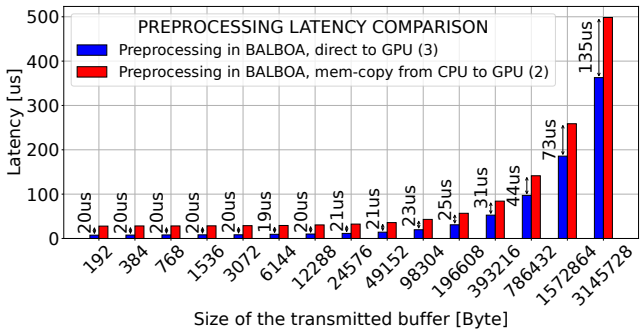


Figure 10: Latency of ML preprocessing in BALBOA with direct-to-GPU vs. an additional step through CPU memory.

the input data from remote memory via `RDMA READs`. In the vanilla setup, without offloaded processing or the direct-to-GPU feature, the received data is written to local host memory, preprocessed in software on the CPU and then locally copied to GPU memory. In a more optimized setup, the preprocessing is implemented in BALBOA's datapath, while still relying on intermediate buffer copies from host to GPU memory. The final setup utilizes both on-datapath preprocessing and direct memory access to the GPU. Again, all experiments are conducted in the same public research cluster with Alveo U55C FPGAs, AMD MI210 GPU accelerators and AMD EPYC 7V13 CPUs. However, a PCIe-switch on the GPU-FPGA path limits the maximum throughput between FPGA and GPU to 70 Gbps (8500 MB/s).

Comparing the vanilla implementation to the BALBOA-optimized design reveals the throughput and utilization benefit of on-datapath pipelines. Even when using multithreading on the CPU and using up to 8 cores for preprocessing tasks, the achievable throughput is limited to 1190 MB/s compared to the FPGA throughput of 8500 MB/s (Figure 9). At the same time, the latency comparison between the GPU-direct utilization of BALBOA compared to the additional copy through host memory shows latency savings of around $20-135\ \mu s$ (Figure 10). Additionally, the direct transfer to GPU remains compliant with RDMA principles of bypassing the CPU.

In summary, the combination of DMA-to-GPU mechanisms with on-datapath preprocessing in the NIC is a promising way to fully use the bandwidth of modern GPUs, avoiding running into CPU bottlenecks and blocking valuable cores.

9 Future Work

In future work, we plan to further leverage the openness and extensibility **(R4)** of BALBOA for research in network infrastructure and application acceleration, similar to the previously demonstrated ML-preprocessing in Section 8: On the application level, the on-datapath enhancements provide an exploration space for operator push-down in distributed database systems [28] to eliminate CPU-passes in query execution, content-aware packet steering as a natural extension of the demonstrated DPI-example or hardware-offloaded gradient compression at line rate [4].

Another potential line of research focuses on network-level transport characteristics: A major problem in RoCE-dependent AI-networks is congestion control [34], which is hard to investigate with commercial NICs due to hardwired congestion controllers. BALBOA maintains an open interface for congestion algorithms deployed in hardware, thus forming an ideal platform to investigate and prototype programmable congestion control, ML-driven congestion control or test other algorithms in hardware instead of simulation. Similar to this, an extensible and open stack allows modifying key RoCE behavior and testing alternative approaches to retransmission [45], packet reordering for per-packet load balancing [64] or custom verbs for interaction with remote memory [10]. Finally, the open nature of BALBOA allows us to further improve its performance in the future, e.g., scaling the throughput to 200G on newer FPGA platforms.

10 Conclusions

In this paper, we present RoCE BALBOA, an open-source, RoCE v2 capable and data center network-ready RDMA stack for exploring and developing in-network accelerators and SmartNICs that allows for far-reaching customization with protocol-enhancing services and on-datapath acceleration of function offloads. We demonstrate the protocol enhancements with AES encryption and ML-based deep packet inspection while also showcasing a realistic use-case scenario of on-datapath function offloads for realistic ML-preprocessing pipelines for recommendation models.

11 Acknowledgements

The authors thank AMD for the donation of the Heterogeneous Accelerated Compute Cluster (HACC), which was used for the development and testing of this project, and Geert Roks for the support and help with the cluster set-up. Also, thanks to the AMD Research team in Dublin for useful feedback and discussions on the design of RoCE BALBOA. We thank Heejae Kim from Seoul National University for his support with Limago.

References

[1] Deep learning recommendation model for personalization and recommendation systems:, 2024. GitHub, https://github.com/facebookresearch/dlrm.

[2] Nvidia bluefield-3 dpu controller user manual, 2024. https://docs.nvidia.com/networking/display/nvidia-bluefield-3-dpu-controller-user-manual.pdf.

[3] Advanced Micro Devices, Inc. *Vitis Networking P4 User Guide (UG1308)*. Advanced Micro Devices, Inc., 2024. Accessed: April 23, 2026.

[4] Agarwal, S., Wang, H., Venkataraman, S., and Papailiopoulos, D. On the utility of gradient compression in distributed training systems. In *Proceedings of Machine Learning and Systems* (2022), D. Marculescu, Y. Chi, and C. Wu, Eds., vol. 4, pp. 652–672.

[5] Aguilera, M. K., Amaro, E., Amit, N., Hunhoff, E., Yelam, A., and Zellweger, G. Memory disaggregation: why now and what are the challenges. *SIGOPS Oper. Syst. Rev. 57*, 1 (June 2023), 38–46.

[6] Aguilera, M. K., Amit, N., Calciu, I., Deguillard, X., Gandhi, J., Novaković, S., Ramanathan, A., Subrahmanyam, P., Suresh, L., Tati, K., Venkatasubramanian, R., and Wei, M. Remote regions: a simple abstraction for remote memory. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)* (Boston, MA, July 2018), USENIX Association, pp. 775–787.

[7] Aguilera, M. K., Amit, N., Calciu, I., Deguillard, X., Gandhi, J., Subrahmanyam, P., Suresh, L., Tati, K., Venkatasubramanian, R., and Wei, M. Remote memory in the age of fast networks. In *Proceedings of the 2017 Symposium on Cloud Computing* (New York, NY, USA, 2017), SoCC '17, Association for Computing Machinery, p. 121–127.

[8] Aguilera, M. K., Keeton, K., Novakovic, S., and Singhal, S. Designing far memory data structures: Think outside the box. In *Proceedings of the Workshop on Hot Topics in Operating Systems* (New York, NY, USA, 2019), HotOS '19, Association for Computing Machinery, p. 120–126.

[9] Alonso, T., Petrica, L., Ruiz, M., Petri-Koenig, J., Umuroglu, Y., Stamelos, I., Koromilas, E., Blott, M., and Vissers, K. Elastic-df: Scaling performance of dnn inference in fpga clouds through automatic partitioning. *ACM Trans. Reconfigurable Technol. Syst. 15*, 2 (Dec. 2021).

[10] Amaro, E., Luo, Z., Ousterhout, A., Krishnamurthy, A., Panda, A., Ratnasamy, S., and Shenker, S. Remote memory calls. In *Proceedings of the 19th ACM Workshop on Hot Topics in Networks* (New York, NY, USA, 2020), HotNets '20, Association for Computing Machinery, p. 38–44.

[11] AMD. Amd embedded rdma enabled nic v4.2, 2024. https://docs.amd.com/viewer/book-attachment/pALa6_dFBrEQ5VQoJ9ygg/gP6nwK9Yi85wq3xZ4KyqZw-pALa6_dFBrEQ5VQoJ9ygg.

[12] AMD. Dma/bridge subsystem for pci express product guide (pg195), 2024. https://docs.amd.com/r/en-US/pg195-pcie-dma.

[13] AMD. Ultrascale+ devices integrated 100g ethernet subsystem logicore ip product guide (pg203), 2024. https://docs.amd.com/r/en-US/pg203-cmac-usplus.

[14] Arashloo, M. T., Lavrov, A., Ghobadi, M., Rexford, J., Walker, D., and Wentzlaff, D. Enabling programmable transport protocols in high-speed nics. In *Proceedings of the 17th Usenix Conference on Networked Systems Design and Implementation* (USA, 2020), NSDI'20, USENIX Association, p. 93–110.

[15] Association, I. T. Roce v2 specification, 2014. https://www.infinibandta.org/ibta-specification/.

[16] Bagga, J., Fang, T., Khare, S., Moeller, O., Provine, J., Sunkad, R., Wang, X., Wu, L., and Zhou, R. Ocp summit 2024: The open future of networking hardware for ai, 2024. Accessed on April 8, 2025.

[17] Bai, W., Abdeen, S. S., Agrawal, A., Attre, K. K., Bahl, P., Bhagat, A., Bhaskara, G., Brokhman, T., Cao, L., Cheema, A., Chow, R., Cohen, J., Elhaddad, M., Ette, V., Figlin, I., Firestone, D., George, M., German, I., Ghai, L., Green, E., Greenberg, A. G., Gupta, M., Haagens, R., Hendel, M., Howlader, R., John, N., Johnstone, J., Jolly, T., Kramer, G., Kruse, D., Kumar, A., Lan, E., Lee, I., Levy, A., Lipshteyn, M., Liu, X., Liu, C., Lu, G., Lu, Y., Lu, X., Makhervaks, V., Malashanka, U., Maltz, D. A., Marinos, I., Mehta, R., Murthi, S., Namdhari, A., Ogus, A., Padhye, J., Pandya, M., Phillips, D., Power, A., Puri, S., Raindel, S., Rhee, J., Russo, A., Sah, M., Sheriff, A., Sparacino, C., Srivastava, A., Sun, W., Swanson, N., Tian, F., Tomczyk, L., Vadlamuri, V., Wolman, A., Xie, Y., Yom, J., Yuan, L., Zhang, Y., and Zill, B. Empowering azure storage with RDMA. In *20th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2023, Boston, MA, April 17-19, 2023* (2023), M. Balakrishnan and M. Ghobadi, Eds., USENIX Association, pp. 49–67.

[18] Basant, A. Scaling data ingestion for machine learning training at meta, Oct 2022. https://engineering.fb.com/2022/09/19/ml-applications/data-ingestion-machine-learning-training-meta/.

[19] Bonato, T., Kabbani, A., Sensi, D. D., Pan, R., Le, Y., Raiciu, C., Handley, M., Schneider, T., Blach, N., Ghalayini, A., Alves, D., Papamichael, M., Caulfield, A., and Hoefler, T. Fastflow: Flexible adaptive congestion control for high-performance datacenters, 2024.

[20] Böther, M., Robroek, T., Gsteiger, V., Holzinger, R., Ma, X., Tözün, P., and Klimovic, A. Modyn: Data-centric machine learning pipeline orchestration. *Proceedings of the ACM on Management of Data 3*, 1 (2025), 1–30.

[21] Brunella, M. S., Belocchi, G., Bonola, M., Pontarelli, S., Siracusano, G., Bianchi, G., Cammarano, A., Palumbo, A., Petrucci, L., and Bifulco, R. hxdp: efficient software packet processing on fpga nics. In *Proceedings of the 14th USENIX Conference on Operating Systems Design and Implementation* (USA, 2020), OSDI'20, USENIX Association.

[22] Burstein, I. Nvidia data center processing unit (dpu) architecture. In *2021 IEEE Hot Chips 33 Symposium (HCS)* (2021), pp. 1–20.

[23] Caulfield, A., Chung, E., Putnam, A., Angepat, H., Fowers, J., Haselman, M., Heil, S., Humphrey, M., Kaur, P., Kim, J.-Y., Lo, D., Massengill, T., Ovtcharov, K., Papamichael, M., Woods, L., Lanka, S., Chiou, D., and Burger, D. A cloud-scale acceleration architecture. In *Proceedings of the 49th Annual IEEE/ACM International Symposium on Microarchitecture* (October 2016), IEEE Computer Society.

[24] CHEN, X., ZHANG, J., FU, T., SHEN, Y., MA, S., QIAN, K., ZHU, L., SHI, C., ZHANG, Y., LIU, M., AND WANG, Z. Demystifying datapath accelerator enhanced off-path smartnic. In *2024 IEEE 32nd International Conference on Network Protocols (ICNP)* (2024), pp. 1–12.

[25] CHIOSA, M., MASCHI, F., MÜLLER, I., ALONSO, G., AND MAY, N. Hardware acceleration of compression and encryption in sap hana. *Proc. VLDB Endow. 15*, 12 (Aug. 2022), 3277–3291.

[26] CHRISTGAU, S., EVERINGHAM, D., MIKOLAJCZAK, F., SCHELTEN, N., SCHNOR, B., SCHROETTER, M., STABERNACK, B., AND STEINERT, F. Enabling communication with fpga-based network-attached accelerators for hpc workloads. In *Proceedings of the SC '23 Workshops of the International Conference on High Performance Computing, Network, Storage, and Analysis* (New York, NY, USA, 2023), SC-W '23, Association for Computing Machinery, p. 530–538.

[27] COCK, D., RAMDAS, A., SCHWYN, D., GIARDINO, M., TUROWSKI, A., HE, Z., HOSSLE, N., KOROLIJA, D., LICCIARDELLO, M., MARTSENKO, K., ACHERMANN, R., ALONSO, G., AND ROSCOE, T. Enzian: an open, general, cpu/fpga platform for systems software research. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems* (New York, NY, USA, 2022), ASPLOS '22, Association for Computing Machinery, p. 434–451.

[28] DANN, J., AND ALONSO, G. Should I hide my duck in the lake? *CoRR abs/2602.18775* (2026).

[29] DATENLORD. blue-rdma: Rocev2 hardware implementation in bluespec systemverilog. https://github.com/datenlord/blue-rdma, 2025.

[30] DUARTE, J., ET AL. Fast inference of deep neural networks in FPGAs for particle physics. *JINST 13*, 07 (2018), P07027.

[31] EGG, A. Online learning for recommendations at grubhub. In *Proceedings of the 15th ACM Conference on Recommender Systems* (New York, NY, USA, 2021), RecSys '21, Association for Computing Machinery, p. 569–571.

[32] FIRESTONE, D., PUTNAM, A., MUNDKUR, S., CHIOU, D., DABAGH, A., ANDREWARTHA, M., ANGEPAT, H., BHANU, V., CAULFIELD, A. M., CHUNG, E. S., CHANDRAPPA, H. K., CHATURMOHTA, S., HUMPHREY, M., LAVIER, J., LAM, N., LIU, F., OVTCHAROV, K., PADHYE, J., POPURI, G., RAINDEL, S., SAPRE, T., SHAW, M., SILVA, G., SIVAKUMAR, M., SRIVASTAVA, N., VERMA, A., ZUHAIR, Q., BANSAL, D., BURGER, D., VAID, K., MALTZ, D. A., AND GREENBERG, A. G. Azure accelerated networking: Smartnics in the public cloud. In *15th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2018, Renton, WA, USA, April 9-11, 2018* (2018), S. Banerjee and S. Seshan, Eds., USENIX Association, pp. 51–66.

[33] FORENCICH, A., SNOEREN, A. C., PORTER, G., AND PAPEN, G. Corundum: An open-source 100-gbps nic. In *2020 IEEE 28th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)* (2020), pp. 38–46.

[34] GANGIDI, A., MIAO, R., ZHENG, S., BONDU, S. J., GOES, G., MORSY, H., PURI, R., RIFTADI, M., SHETTY, A. J., YANG, J., ZHANG, S., FERNANDEZ, M. J., GANDHAM, S., AND ZENG, H. Rdma over ethernet for distributed training at meta scale. In *Proceedings of the ACM SIGCOMM 2024 Conference* (New York, NY, USA, 2024), ACM SIGCOMM '24, Association for Computing Machinery, p. 57–70.

[35] GIANTSIDI, D., PRITZI, J., GUST, F., KATSARAKIS, A., KOSHIBA, A., AND BHATOTIA, P. Tnic: A trusted nic architecture: A hardware-network substrate for building high-performance trustworthy distributed systems. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2* (New York, NY, USA, 2025), ASPLOS '25, Association for Computing Machinery, p. 1282–1301.

[36] GOOGLE. PSP Security Protocol. https://github.com/google/psp, 2022. Accessed: 2025-12-09.

[37] GUO, A., GENG, T., ZHANG, Y., HAGHI, P., WU, C., TAN, C., LIN, Y., LI, A., AND HERBORDT, M. Fcsn: A fpga-centric smartnic framework for neural networks. In *2022 IEEE 30th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)* (2022), pp. 1–2.

[38] GUPTA, U., WU, C.-J., WANG, X., NAUMOV, M., REAGEN, B., BROOKS, D., COTTEL, B., HAZELWOOD, K., HEMPSTEAD, M., JIA, B., LEE, H.-H. S., MALEVICH, A., MUDIGERE, D., SMELYANSKIY, M., XIONG, L., AND ZHANG, X. The architectural implications of facebook's dnn-based personalized recommendation. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)* (2020), pp. 488–501.

[39] HAMIDOUCHE, K., VENKATESH, A., AWAN, A. A., SUBRAMONI, H., CHU, C.-H., AND PANDA, D. K. Exploiting gpudirect rdma in designing high performance openshmem for nvidia gpu clusters. In *2015 IEEE International Conference on Cluster Computing* (2015), pp. 78–87.

[40] HARTMANN, M., WEBER, L., WIRTH, J., SOMMER, L., AND KOCH, A. Optimizing a hardware network stack to realize an in-network ml inference application. In *2021 IEEE/ACM International Workshop on Heterogeneous High-performance Reconfigurable Computing (H2RC)* (2021), pp. 21–32.

[41] HE, Z., KOROLIJA, D., AND ALONSO, G. Easynet: 100 gbps network for HLS. In *31st International Conference on Field-Programmable Logic and Applications, FPL 2021, Dresden, Germany, August 30 - Sept. 3, 2021* (2021), IEEE, pp. 197–203.

[42] HE, Z., KOROLIJA, D., ZHU, Y., RAMHORST, B., LAAN, T., PETRICA, L., BLOTT, M., AND ALONSO, G. Accl+: an fpga-based collective engine for distributed applications. In *Proceedings of the 18th USENIX Conference on Operating Systems Design and Implementation* (USA, 2024), OSDI'24, USENIX Association.

[43] HE, Z., PARRAVICINI, D., PETRICA, L., O'BRIEN, K., ALONSO, G., AND BLOTT, M. Accl: Fpga-accelerated collectives over 100 gbps tcp-ip. In *2021 IEEE/ACM International Workshop on Heterogeneous High-performance Reconfigurable Computing (H2RC)* (2021), pp. 33–43.

[44] HEER, M. J., RAMHORST, B., AND ALONSO, G. Machine learning-based deep packet inspection at line rate for rdma on fpgas. In *Proceedings of the 5th Workshop on Machine Learning and Systems* (New York, NY, USA, 2025), EuroMLSys '25, Association for Computing Machinery, p. 148–155.

[45] HUANG, P., CHEN, G., ZHANG, X., LIU, C., WANG, H., SHEN, H., BIAN, Y., LU, Y., RUAN, Z., LI, B., ZHANG, J., LIU, Y., AND CHEN, Z. Fast and scalable selective retransmission for rdma. In *IEEE INFOCOM 2025 - IEEE Conference on Computer Communications* (2025), pp. 1–10.

[46] IBANEZ, S., MALLERY, A., ARSLAN, S., JEPSEN, T., SHAHBAZ, M., KIM, C., AND MCKEOWN, N. The nanopu: A nanosecond network stack for datacenters. In *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)* (July 2021), USENIX Association, pp. 239–256.

[47] ISLAM, N. S., SHANKAR, D., LU, X., WASI-UR-RAHMAN, M., AND PANDA, D. K. Accelerating i/o performance of big data analytics on hpc clusters through rdma-based key-value store. In *2015 44th International Conference on Parallel Processing* (2015), pp. 280–289.

[48] ISTVÁN, Z. Let's add transactions to fpga-based key-value stores! In *Proceedings of the 16th International Workshop on Data Management on New Hardware* (New York, NY, USA, 2020), DaMoN '20, Association for Computing Machinery.

[49] JASNY, M., ZIEGLER, T., NELSON-SLIVON, J., LEIS, V., AND BINNIG, C. Synchronizing disaggregated data structures with one-sided rdma: Pitfalls, experiments and design guidelines. *ACM Trans. Database Syst. 50*, 1 (Mar. 2025).

[50] JIA, C., LI, C., LI, Y., HU, X., AND LI, J. Facl: A flexible and high-performance acl engine on fpga-based smartnic. In *2022 IFIP Networking Conference (IFIP Networking)* (2022), pp. 1–9.

[51] JIN, Z., CHEN, Y., LIANG, M., WANG, Y., FANG, G., ZHOU, A., ZHANG, K., XU, J., LIN, W., LIN, Y., ZHAO, S., SHI, W., HE, Z., CAI, S., AND CHEN, W. Os2g: A high-performance dpu offloading architecture for gpu-based deep learning with object storage. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2* (New York, NY, USA, 2025), ASPLOS '25, Association for Computing Machinery, p. 750–765.

[52] KALIA, A., KAMINSKY, M., AND ANDERSEN, D. G. Using rdma efficiently for key-value services. *SIGCOMM Comput. Commun. Rev. 44*, 4 (Aug. 2014), 295–306.

[53] KENNY, J. P., AND ULMER, C. D. Roce: Promising technology for ethernet as a high performance networking fabric. Tech. rep., Sandia National Lab. (SNL-CA), Livermore, CA (United States), 11 2019.

[54] KHAN, B., HEINZ, C., AND KOCH, A. The open-source deliba2 hardware/software framework for distributed storage accelerators. *ACM Trans. Reconfigurable Technol. Syst. 17*, 2 (Mar. 2024).

[55] KIANPISHEH, S., AND TALEB, T. A survey on in-network computing: Programmable data plane and technology specific applications. *IEEE Communications Surveys & Tutorials 25*, 1 (2023), 701–761.

[56] KOROLIJA, D., ROSCOE, T., AND ALONSO, G. Do OS abstractions make sense on FPGAs? In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)* (Nov. 2020), USENIX Association, pp. 991–1010.

[57] KUMAR, G., DUKKIPATI, N., JANG, K., WASSEL, H. M. G., WU, X., MONTAZERI, B., WANG, Y., SPRINGBORN, K., ALFELD, C., RYAN, M., WETHERALL, D., AND VAHDAT, A. Swift: Delay is simple and effective for congestion control in the datacenter. In *Proceedings of the Annual Conference of the ACM Special Interest Group on Data Communication on the Applications, Technologies, Architectures, and Protocols for Computer Communication* (New York, NY, USA, 2020), SIGCOMM '20, Association for Computing Machinery, p. 514–528.

[58] LEE, Y., KIM, H., AND RHU, M. PreSto: An In-Storage Data Preprocessing System for Training Recommendation Models . In *2024 ACM/IEEE 51st Annual International Symposium on Computer Architecture (ISCA)* (Los Alamitos, CA, USA, July 2024), IEEE Computer Society, pp. 340–353.

[59] LI, Q., GAO, Y., WANG, X., QIU, H., LE, Y., LIU, D., XIANG, Q., FENG, F., ZHANG, P., LI, B., DONG, J., TANG, L., LIU, H. H., LIU, S., LI, W., MIAO, R., WU, Y., WU, Z., HAN, C., YAN, L., CAO, Z., WU, Z., TIAN, C., CHEN, G., CAI, D., WU, J., ZHU, J., WU, J., AND SHU, J. Flor: An open high performance RDMA framework over heterogeneous RNICs. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)* (Boston, MA, July 2023), USENIX Association, pp. 931–948.

[60] LI, Y., KASHYAP, A., GUO, Y., AND LU, X. Compression Analysis for BlueField-2/-3 Data Processing Units: Lossy and Lossless Perspectives . *IEEE Micro 44*, 02 (Mar. 2024), 8–19.

[61] LIN, J., PATEL, K., STEPHENS, B. E., SIVARAMAN, A., AND AKELLA, A. PANIC: A High-Performance programmable NIC for multi-tenant networks. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)* (Nov. 2020), USENIX Association, pp. 243–259.

[62] LIN, W., SHAN, Y., KOSTA, R., KRISHNAMURTHY, A., AND ZHANG, Y. Supernic: An fpga-based, cloud-oriented smartnic. In *Proceedings of the 2024 ACM/SIGDA International Symposium on Field Programmable Gate Arrays* (New York, NY, USA, 2024), FPGA '24, Association for Computing Machinery, p. 130–141.

[63] LIU, J., DRAGOJEVIĆ, A., FLEMING, S., KATSARAKIS, A., KOROLIJA, D., ZABLOTCHI, I., NG, H.-C., KALIA, A., AND CASTRO, M. Honeycomb: Ordered key-value store acceleration on an fpga-based smartnic. *IEEE Transactions on Computers 73*, 3 (2024), 857–871.

[64] LIU, X., LI, W., AND CHEN, K. Enabling packet spraying over commodity rnics with in-network support. In *Proceedings of the 9th Asia-Pacific Workshop on Networking* (New York, NY, USA, 2025), APNET '25, Association for Computing Machinery, p. 51–58.

[65] MA, R., GEORGANAS, E., HEINECKE, A., GRIBOK, S., BOUTROS, A., AND NURVITADHI, E. Fpga-based ai smart nics for scalable distributed ai training systems. *IEEE Computer Architecture Letters 21*, 2 (2022), 49–52.

[66] MACARTHUR, P., LIU, Q., RUSSELL, R. D., MIZERO, F., VEERARAGHAVAN, M., AND DENNIS, J. M. An integrated tutorial on infiniband, verbs, and mpi. *IEEE Communications Surveys & Tutorials 19*, 4 (2017), 2894–2926.

[67] MANSOUR, W., JANVIER, N., AND FAJARDO, P. Fpga implementation of rdma-based data acquisition system over 100-gb ethernet. *IEEE Transactions on Nuclear Science 66*, 7 (2019), 1138–1143.

[68] MARINI, F., BELLATO, M., BERGNOLI, A., CORTI, D., GRIGGIO, A., ISOCRATE, R., MODENESE, L., TOFFANO, M., ARCARO, C., PIERRO, F. D., MARIOTTI, M., MI, M., AND WANG, P. Fpga-based rocev2-rdma readout electronics for the ctao-lst advanced camera. *IEEE Transactions on Nuclear Science* (2025), 1–1.

[69] MARQUART, R. Porting the roce-balboa rdma-stack from coyote to amd reconic. Master's thesis, ETH Zurich, 2024.

[70] MCINNES, M., HOLLINGSHED, M., NOTTINGHAM, C., MALIS, S., PLANK, A., AND LEE, D. Microsoft azure boost. https://learn.microsoft.com/en-us/azure/azure-boost/overview, 2025. Accessed on April 8, 2025.

[71] MITTAL, R., LAM, V. T., DUKKIPATI, N., BLEM, E., WASSEL, H., GHOBADI, M., VAHDAT, A., WANG, Y., WETHERALL, D., AND ZATS, D. Timely: Rtt-based congestion control for the datacenter. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication* (New York, NY, USA, 2015), SIGCOMM '15, Association for Computing Machinery, p. 537–550.

[72] MOYA, J., GABATHULER, M., RUIZ, M., AND ALONSO, G. fpgasystems/hacc: Ethz-hacc. Zenodo, Sept. 2023. https://doi.org/10.5281/zenodo.8340448.

[73] MUSTARD, C., RUFFY, F., GAKHOKIDZE, A., BESCHASTNIKH, I., AND FEDOROVA, A. Jumpgate: In-Network processing as a service for data analytics. In *11th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 19)* (Renton, WA, July 2019), USENIX Association.

[74] NAUMOV, M., MUDIGERE, D., SHI, H. M., HUANG, J., SUNDARAMAN, N., PARK, J., WANG, X., GUPTA, U., WU, C., AZZOLINI, A. G., DZHULGAKOV, D., MALLEVICH, A., CHERNIAVSKII, I., LU, Y., KRISHNAMOORTHI, R., YU, A., KONDRATENKO, V., PEREIRA, S., CHEN, X., CHEN, W., RAO, V., JIA, B., XIONG, L., AND SMELYANSKIY, M. Deep learning recommendation model for personalization and recommendation systems. *CoRR abs/1906.00091* (2019).

[75] NVIDIA CORPORATION. Nvidia doca psp gateway application guide. https://docs.nvidia.com/doca/sdk/doca-psp-gateway-application-guide/index.html. Accessed: 2025-12-09.

[76] PRASOPOULOS, K., KOSTA, R., BUGNION, E., AND KOGIAS, M. Sird: a sender-informed, receiver-driven datacenter transport protocol. In *Proceedings of the 22nd USENIX Symposium on Networked Systems Design and Implementation* (USA, 2025), NSDI '25, USENIX Association.

[77] RAMHORST, B., KOROLIJA, D., HEER, M. J., DANN, J., LIU, L., AND ALONSO, G. Coyote v2: Raising the level of abstraction for data center fpgas. In *Proceedings of the ACM SIGOPS 31st Symposium on Operating Systems Principles* (New York, NY, USA, 2025), SOSP '25, Association for Computing Machinery, p. 639–654.

[78] ROTHENBERGER, B., TARANOV, K., PERRIG, A., AND HOEFLER, T. ReDMArk: Bypassing RDMA security mechanisms. In *30th USENIX Security Symposium (USENIX Security 21)* (Aug. 2021), USENIX Association, pp. 4277–4292.

[79] RUIZ, M., SIDLER, D., SUTTER, G., ALONSO, G., AND LÓPEZ-BUEDO, S. Limago: An fpga-based open-source 100 gbe TCP/IP stack. In *29th International Conference on Field Programmable Logic and Applications, FPL 2019, Barcelona, Spain, September 8-12, 2019* (2019), I. Sourdis, C. Bouganis, C. Álvarez, L. A. T. Díaz, P. Valero-Lara, and X. Martorell, Eds., IEEE, pp. 286–292.

[80] SAPIO, A., ABDELAZIZ, I., ALDILAIJAN, A., CANINI, M., AND KALNIS, P. In-network computation is a dumb idea whose time has come. In *Proceedings of the 16th ACM Workshop on Hot Topics in Networks* (New York, NY, USA, 2017), HotNets '17, Association for Computing Machinery, p. 150–156.

[81] SCHELTEN, N., STEINERT, F., KNAPHEIDE, J., SCHULTE, A., AND STABERNACK, B. A high-throughput, resource-efficient implementation of the rocev2 remote dma protocol and its application. *ACM Trans. Reconfigurable Technol. Syst. 16*, 1 (Dec. 2022).

[82] SHALEV, L., AYOUB, H., BSHARA, N., AND SABBAG, E. A cloud-optimized transport protocol for elastic and scalable hpc. *IEEE Micro 40*, 6 (2020), 67–73.

[83] SHU, R., CHENG, P., CHEN, G., GUO, Z., QU, L., XIONG, Y., CHIOU, D., AND MOSCIBRODA, T. Direct universal access: Making data center resources available to FPGA. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)* (Boston, MA, Feb. 2019), USENIX Association, pp. 127–140.

[84] SIDLER, D., WANG, Z., CHIOSA, M., KULKARNI, A., AND ALONSO, G. Strom: smart remote memory. In *Proceedings of the Fifteenth European Conference on Computer Systems* (New York, NY, USA, 2020), EuroSys '20, Association for Computing Machinery.

[85] SIMPSON, A. K., SZEKERES, A., NELSON, J., AND ZHANG, I. Securing RDMA for High-Performance datacenter storage systems. In *12th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 20)* (July 2020), USENIX Association.

[86] SINGHVI, A., DUKKIPATI, N., CHANDRA, P., WASSEL, H. M. G., SHARMA, N. K., REBELLO, A., SCHUH, H., KUMAR, P., MONTAZERI, B., BANSOD, N., THOMAS, S., CHO, I., SEIBERT, H. L., WU, B., YANG, R., LI, Y., HUANG, K., YIN, Q., AGARWAL, A., VADUVATHA, S., WANG, W., MOSHREF, M., JI, T., WETHERALL, D., AND VAHDAT, A. Falcon: A reliable, low latency hardware transport. In *Proceedings of the ACM SIGCOMM 2025 Conference* (New York, NY, USA, 2025), SIGCOMM '25, Association for Computing Machinery, p. 248–263.

[87] SU, W., AND SHRIVASTAV, V. Edm: An ultra-low latency ethernet fabric for memory disaggregation. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1* (New York, NY, USA, 2025), ASPLOS '25, Association for Computing Machinery, p. 377–394.

[88] SUN, Z., GUO, Z., MA, J., AND PAN, Y. A high-performance fpga-based roce v2 rdma packet parser and generator. *Electronics 13*, 20 (2024).

[89] TARANOV, K., ROTHENBERGER, B., DE SENSI, D., PERRIG, A., AND HOEFLER, T. Nevermore: Exploiting rdma mistakes in nvme-of storage applications. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security* (New York, NY, USA, 2022), CCS '22, Association for Computing Machinery, p. 2765–2778.

[90] TARANOV, K., ROTHENBERGER, B., PERRIG, A., AND HOEFLER, T. sRDMA – efficient NIC-based authentication and encryption for remote direct memory access. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)* (July 2020), USENIX Association, pp. 691–704.

[91] TECHNOLOGIES, M. Mellanox connectx®-5 ex ethernet single and dual qsfp28 port adapter cards user manual, 2018. https://gzhls.at/blob/ldb/9/a/3/2/073d2b59ddaec6a5a00744e21c93a3459529.pdf.

[92] VENERE, M., SORRENTINO, G., RAMHORST, B., HEER, M. J., PETRICA, L., KOROLIJA, D., SANTAMBROGIO, M. D., CONFICCONI, D., ALONSO, G., AND O'BRIEN, K. Ropeerto: A datacenter-scale architecture for peer-to-peer dma between gpus and fpgas. In *Proceedings of the 21st European Conference on Computer Systems* (New York, NY, USA, 2026), EUROSYS '26, Association for Computing Machinery, p. 1829–1846.

[93] WANG, Q., LU, Y., AND SHU, J. Designing an efficient tree index on disaggregated memory. *Commun. ACM* (Apr. 2025). Online First.

[94] WANG, W., PENG, B., YAO, J., AND GUAN, H. Rehss: Optimizing latency for cloud hybrid storage systems using in-network placement. In *2025 IEEE/ACM 33rd International Symposium on Quality of Service (IWQoS)* (2025), pp. 1–6.

[95] WANG, Z., HUANG, H., ZHANG, J., WU, F., AND ALONSO, G. FpgaNIC: An FPGA-based versatile 100gb SmartNIC for GPUs. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)* (Carlsbad, CA, July 2022), USENIX Association, pp. 967–986.

[96] WANG, Z., LUO, L., NING, Q., ZENG, C., LI, W., WAN, X., XIE, P., FENG, T., CHENG, K., GENG, X., WANG, T., LING, W., HUO, K., AN, P., JI, K., ZHANG, S., XU, B., FENG, R., DING, T., CHEN, K., AND GUO, C. SRNIC: A scalable architecture for RDMA NICs. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)* (Boston, MA, Apr. 2023), USENIX Association, pp. 1–14.

[97] XILINX. Xilinx embedded target rdma enabled nic v1.1, 2018. https://docs.amd.com/v/u/en-US/pg294-etrnic.

[98] XILINX. open-nic: Open-source Ethernet/Networking stack for Xilinx devices. https://github.com/Xilinx/open-nic, 2025. Accessed: 3 December 2025.

[99] ZANG, D., CAO, Z., LIU, X., WANG, L., WANG, Z., AND SUN, N. Prop: Using pcie-based rdma to accelerate rack-scale communications in data centers. In *2015 IEEE 21st International Conference on Parallel and Distributed Systems (ICPADS)* (2015), pp. 465–472.

[100] ZHANG, M., HUA, Y., ZUO, P., AND LIU, L. FORD: Fast one-sided RDMA-based distributed transactions for disaggregated persistent memory. In *20th USENIX Conference on File and Storage Technologies (FAST 22)* (Santa Clara, CA, Feb. 2022), USENIX Association, pp. 51–68.

[101] ZHANG, Z., CHANG, C., LIN, H., WANG, Y., ARORA, R., AND JIN, X. Is network the bottleneck of distributed training? In *Proceedings of the 2020 Workshop on Network Meets AI & ML, NetAI@SIGCOMM, Virtual Event, USA, August 14, 2020* (2020), B. Arzani and X. Jin, Eds., ACM, pp. 8–13.

[102] ZHAO, C., MIN, J., LIU, M., AND KRISHNAMURTHY, A. Whiteboxing rdma with packet-granular software control. In *Proceedings of the 22nd USENIX Symposium on Networked Systems Design and Implementation* (USA, 2025), NSDI '25, USENIX Association.

[103] ZHAO, H., YANG, Z., CHENG, Y., TIAN, C., REN, S., XIAO, W., YUAN, M., CHEN, L., LIU, K., ZHANG, Y., LI, Y., AND LIN, W. Goldminer: Elastic scaling of training data pre-processing pipelines for deep learning. *Proc. ACM Manag. Data 1*, 2 (June 2023).

[104] ZHAO, M., AGARWAL, N., BASANT, A., GEDIK, B., PAN, S., OZDAL, M., KOMURAVELLI, R., PAN, J., BAO, T., LU, H., NARAYANAN, S., LANGMAN, J., WILFONG, K., RASTOGI, H., WU,

C.-J., KOZYRAKIS, C., AND POL, P. Understanding data storage and ingestion for large-scale deep recommendation model training: industrial product. In *Proceedings of the 49th Annual International Symposium on Computer Architecture* (New York, NY, USA, 2022), ISCA '22, Association for Computing Machinery, p. 1042–1057.

[105] ZHAO, X., WANG, M., ZHAO, X., LI, J., ZHOU, S., YIN, D., LI, Q., TANG, J., AND GUO, R. Embedding in recommender systems: A survey. *arXiv preprint in arXiv:2310.18608* (2023).

[106] ZHAO, Y., SHU, R., AND XIONG, Y. Src: A scalable reliable connection for rdma with decoupled qps and connections. In *Proceedings of the 9th Asia-Pacific Workshop on Networking* (New York, NY, USA, 2025), APNET '25, Association for Computing Machinery, p. 44–50.

[107] ZHONG, G., KOLEKAR, A., AMORNPAISANNON, B., CHOI, I., JAVAID, H., AND BALDI, M. A primer on reconic: Rdma-enabled compute offloading on smartnic. *CoRR abs/2312.06207* (2023).

[108] ZHU, Y., ERAN, H., FIRESTONE, D., GUO, C., LIPSHTEYN, M., LIRON, Y., PADHYE, J., RAINDEL, S., YAHIA, M. H., AND ZHANG, M. Congestion control for large-scale rdma deployments. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication* (New York, NY, USA, 2015), SIGCOMM '15, Association for Computing Machinery, p. 523–536.

[109] ZHU, Y., JIANG, W., AND ALONSO, G. Efficient tabular data preprocessing of ml pipelines. *arXiv preprint arXiv:2409.14912* (2024).

[110] ZUO, P., SUN, J., YANG, L., ZHANG, S., AND HUA, Y. One-sided RDMA-Conscious extendible hashing for disaggregated memory. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)* (July 2021), USENIX Association, pp. 15–29.

[111] ZUO, P., ZHOU, Q., SUN, J., YANG, L., ZHANG, S., HUA, Y., CHENG, J., HE, R., AND YAN, H. Race: One-sided rdma-conscious extendible hashing. *ACM Trans. Storage 18*, 2 (Apr. 2022).

ARTIFACT EVALUATED
USENIX
AVAILABLE

DPA-STORE: An Ordered Network Data Path Key-Value Store

Frederic Schimmelpfennig
Johannes Gutenberg University Mainz

Jan Sass
Saarland University Saarbrücken[*]

Reza Salkhordeh
Johannes Gutenberg University Mainz

Martin Kröning
RWTH Aachen University

Stefan Lankes
RWTH Aachen University

André Brinkmann
Saarland University Saarbrücken[*]

[*]Work was done while affiliated with *Johannes Gutenberg University Mainz*.

Abstract

Remote in-memory key–value (KV) stores are fundamental to a wide range of applications, many of which depend on efficient range operations. However, existing designs fall short of simultaneously providing high performance, low complexity, and full range-query support. Host-based systems like REDIS and MEMCACHED are constrained by the kernel network stack and NIC–host interactions. Implementations of hash-based structures that bypass the OS kernel or serve directly from SmartNICs demonstrate upper performance limits but cannot support range queries. Distributed RDMA systems offer high throughput and range functionality when using stateful clients, but these clients increase the risk of faults and complicate scaling. Finally, SmartNICs traversing host-memory trees suffer from high numbers of DMA round-trips.

This paper presents *DPA-Store*, which uses the on-path *Data Path Accelerators* (DPAs) of the BlueField-3 SmartNIC to circumvent OS overheads while supporting stateless clients and range queries. The DPAs fetch incoming requests directly from the NIC buffers and traverse a lock-free learned index tree within the DPA memory. Values are fetched from a host-side replica of the tree when reaching the leaf level. Writes are buffered in DPA memory and transferred in batches to the host. Compute-heavy structural operations are executed on the host and transactionally *stitched* back to the SmartNIC. Complemented with a read cache directly on the NIC, DPA-Store sustains 33 million operations per second (MOPS) for lookups and 13 MOPS for range queries. Our evaluation shows that DPA-Store is already faster than or competitive with state-of-the-art solutions, and we demonstrate how small changes to the BlueField-3 hardware could additionally increase performance.

1 Introduction

Many modern workloads rely on remote key-value (KV) stores that support lookups, inserts, and range queries. However, remote performance is often limited by transport bottlenecks. Traditional remote KV stores such as REDIS [38] and MEMCACHED [33] are accessed over TCP/IP sockets, commonly over Ethernet. Their reliance on the operating system (OS) kernel network stack, together with PCIe bottlenecks between the network interface card (NIC) and the host, limits throughput and increases latencies.

To mitigate OS overhead, some approaches use the Data Plane Development Kit (DPDK) [12] to bypass the network stack via direct user-space access. MICA reaches nearly 100 million operations per second (MOPS) on a single server node [28]. Other works like KV-DIRECT achieve even higher throughput by removing host involvement and serving requests directly from an FPGA SmartNIC [23]. This reduces latencies because the KV store is closer to the network. However, to achieve these results, KV-DIRECT is constrained in the complexity and capacity of the underlying data structures. Both MICA and KV-DIRECT use hash-based point lookups and lack range queries. Recent works apply similar offload designs to newer SmartNIC generations and are affected by the same shortcomings [15, 37]. Approaches like HONEYCOMB [29] traverse data structures in host memory and support range queries. However, their performance falls behind because they issue frequent DMA round-trips.

Clients of KV stores that use remote direct memory access (RDMA) can access host memory directly [25, 26, 39, 47–49, 57]. These approaches introduce architectural challenges because every client eventually needs to know the remote addresses of the data on the server to access it. SHERMAN minimizes round-trips for writes but requires client caches for tree traversals [25]. Similarly, ROLEX [26] maintains metadata indexes on the clients and optimizes accesses using learned indexes. These systems can deliver high performance but their dependence on client-side logic and resources raises scaling complexities, introduces consistency concerns, and requires additional failure handling.

In this paper, we present *DPA-Store*, an ordered in-memory KV store that uses the BlueField-3 SmartNIC [36] and its *Data Path Accelerators* (DPAs) [35] to bypass OS overheads and to allow stateless clients. DPAs are highly parallel, programmable compute units embedded in the network data path.

DPAs have direct access to NIC buffers and NIC-side DRAM, and can perform DMA operations to access host memory, enabling low-latency and flexible request processing [7].

DPA-Store is built around the high level of concurrency of the DPA subsystem, which consists of 16 cores, each with 16 threads, for a total of 256 threads. When receiving requests, these threads traverse a learned index tree [20], stored in NIC memory, without involving the host. To circumvent memory capacity restrictions of the NIC, values are stored in a replica of the tree on the host. As a result, point and range lookups, at most, need to issue DMA operations at the leaf level. This keeps the number of DMA operations, and thus PCIe crossings, to a minimum.

When an insert request is received, the responsible DPA thread appends the entry to an insert buffer at the leaf level on the NIC side. These entries are immediately visible to subsequent lookups. The insert buffers are transferred in batches to the host when they are full. The host, with its greater compute capabilities, performs expensive structural updates to its tree replica using concurrent *patcher* threads that apply the inserts to the host-side tree, retrain affected subtrees, and prepare new nodes. *Stitcher* threads then copy the new nodes into the NIC-side tree and make them available via pointer swaps. This guarantees consistency while keeping the NIC-side traversal path lock-free.

We employ a NIC-side read cache for hot entries to reduce the number of tree traversals and DMA operations. Requests are routed to DPA threads based on a hash of their key, so that small Bloom filters that guard cache accesses fit within each thread's unused cache-line space. These cache-lines are rarely evicted from the L1 caches of the DPA threads, allowing efficient caching of hot key-value pairs.

Our learned index allows fine-grained optimizations for the DPA subsystem's memory characteristics, scheduling, and concurrency. We design the traversal logic around optimal cache-line accesses and employ fixed-point calculations to compensate for the lack of floating-point units on the DPAs. Our contributions are as follows:

- We show that it is possible to build a KV store supporting lookups, inserts, and range queries on a BlueField-3 SmartNIC that is already competitive, for most operations, with state-of-the-art RDMA-based KV stores *without* relying on stateful clients. DPA-Store achieves 33 MOPS for `GET` and 13 MOPS for `RANGE` operations.
- Our extensive experiments to evaluate DPA-Store reveal in-depth performance characteristics of the BlueField-3, going beyond what prior works have reported. These insights also explain DPA-Store's low `INSERT` performance of only 1.7 MOPS.
- We demonstrate how modest modifications to the BlueField-3 hardware, without changing the overall programming model, could unlock substantial performance gains, which would easily surpass the performance of today's fastest RDMA-based KV stores.

The paper is structured as follows. Section 2 provides background and surveys related work. Section 3 details the DPA-Store architecture and implementation. Section 4 evaluates DPA-Store with a sensitivity analysis and comparison benchmarks. Finally, Section 5 concludes the paper.

2 Background & Related Work

In this section, we summarize the background on learned indexes and the BlueField-3 DPU, and position DPA-Store relative to kernel-bypass, NIC-assisted, and RDMA-based remote KV systems.

2.1 Classical & Learned Index Structures

Key-value (KV) stores expose a compact interface, providing `GET`, `INSERT`, `UPDATE`, `DELETE`, and optionally `RANGE` operations. Typically, KV stores rely on either hash tables or, if range operations are required, ordered tree structures. Modern, non-remote in-memory KV stores serve node-local processes using optimized variants of traditional data structures such as B$^+$-trees, tries, or hash maps, with highly optimized traversals and concurrency control [3, 21, 22, 30, 52]. However, traditional data structures often suffer from suboptimal memory access patterns [20].

To address this issue, *learned indexes* [9, 11, 13, 19, 20, 24, 27, 32, 51, 54] capture the key distribution using lightweight models. This allows lookup operations to jump directly to the relevant region inside a node, followed by a short scan, and requires only a few cache-line accesses per tree level (see Figure 1). A common choice for approximating the key distribution is a piecewise linear approximation (PLA). Short linear segments are fitted to the cumulative key distribution, enabling a linear prediction $p = a \cdot k + b$ of the target position of key k using parameters a and b with an error of at most ε.

Thus, instead of $O(\log n)$ memory accesses per node of size n, with potentially many independent cache-line accesses (e.g., in a B$^+$-tree), lookup operations in a learned index are aggregated in a small contiguous window $[p-\varepsilon, p+\varepsilon]$, trading relatively slow memory accesses for fast CPU cycles.

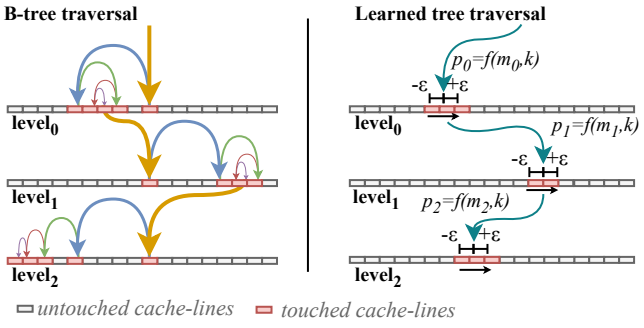


Figure 1: Access patterns in B$^+$-trees vs. learned index trees.

The error bound ε balances capacity efficiency against scan effort. Smaller bounds reduce the scan range but increase the number of models, whereas larger bounds keep more data in continuous segments, requiring larger regions to be scanned. Therefore, choosing a suitable parameter ε is important when mapping learned indexes to accelerators like SmartNICs with their limited memory access performance.

Insert and model rebuild strategies further impact runtime behavior. ALEX [9], for example, stores keys in gapped arrays with small buffers and does not require relearning until gaps are full. HYPER [54] combines bottom-up and top-down strategies and uses overflow buffers to limit memory overheads, while XINDEX [43] targets highly concurrent workloads. In all cases, buffer layout, size, and update policies govern retraining cost and concurrency characteristics.

2.2 Remote KV Stores

Remote KV stores introduce additional complexity by allowing access over network protocols. Widely deployed Ethernet-based KV stores [10, 17, 33, 38] serve general-purpose workloads. However, their request throughput is limited by the host networking stack and related OS components, such as network sockets and context switches, as well as the NIC-host PCIe boundary [4, 34, 40].

To address these issues, RDMA-based KV stores bypass the OS and networking stack [6, 14, 57]. They are based on either traditional tree architectures [25, 47, 48] or learned indexes [26, 49] and offer the full set of KV operations, including range queries.

SHERMAN [25] is a write-optimized distributed B$^+$-tree over disaggregated memory that employs hierarchical NIC on-chip locking, client-side index caching, and lock-free reads to increase write throughput. SMART [39] optimizes this approach by addressing RDMA-NIC scale-up bottlenecks. XSTORE [49] couples a server-side B$^+$-tree with a learned cache on the client, which the client uses to predict the location of the KV pair on the server, saving RDMA round-trips. This allows range queries to complete with as few as two RDMA operations. The server retrains models in the background. ROLEX [26] addresses dynamic workloads by strictly controlling data movement within a learned tree, allowing asynchronous retraining of models. A leaf-atomic shift scheme keeps leaves sorted and minimizes interference.

These systems demonstrate that low-latency serving of ordered range queries can achieve high throughput. However, they either rely on stateful client logic or on multiple RDMA round-trips per request.

The discussed bottlenecks can also be overcome by minimizing the impact of the operating system or by using SmartNICs. MICA combines DPDK user-space networking with parallel data partitioning to bypass kernel overheads, allowing a single server to sustain nearly 100 MOPS [12, 28]. KV-DIRECT offloads the KV store to FPGA-programmable NICs by extending RDMA primitives [23]. It serves NIC-resident point lookups that scale with multiple NICs per server and reach microsecond tail latencies even at high throughput, albeit with constrained capacity. Both MICA and KV-DIRECT use hash-based data structures that achieve very high throughput but do not support range queries.

Some recent works [15, 37, 50, 55] also lack support for range operations but contribute detailed insights into the capabilities of current SmartNICs, which can offload increasingly complex KV-store mechanisms [16, 18, 41, 42, 45].

HONEYCOMB [29] implements a B-tree traversal within an FPGA on a SmartNIC that accesses a tree in host memory via DMA. It provides a KV-store interface, including ranges, but requires multiple expensive DMAs for uncached KV pairs. HIDPU [56] performs address translation for disaggregated storage and uses the Huawei Hi1823 SmartNIC's DPAs to map partially continuous address areas. HIDPU stores those mappings in specialized segments, accessed through learned models within the 4 MB NIC-side memory. This work is orthogonal to a general-purpose KV store in terms of features and requirements, relies on hashed segments, and does not support range queries across multiple mappings. Nevertheless, its performance benefits motivate offloading KV stores to SmartNICs. DALDEX, on the other hand, uses SmartNICs to offload the training of a learned index for persistent memory to save host overheads, rather than for network serving [46].

2.3 BlueField-3 and DPA Subsystem

The BlueField-3 SmartNIC is a ConnectX-based network adapter. In addition to high-speed network functions, it offers an off-path ARM CPU and an on-path DPA cluster, as well as 16 GB DDR5 memory (see Figure 2). The ARM CPU allows the BlueField-3 to run a dedicated operating system, e.g., for control plane management, and to access accelerator engines on the SmartNIC (e.g., de/encryption engines).

The DPA subsystem consists of 16 physical RISC-V cores, each running at 1.8 GHz and featuring 16 threads. Currently, only 189 of the 256 available threads can be used by application code [7]. DPA threads are hardware-scheduled at fine granularity, enabling high concurrency and packet-processing

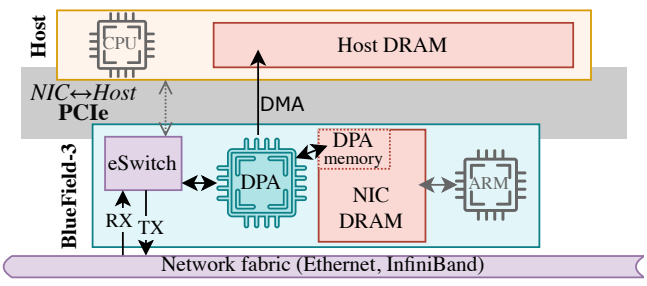


Figure 2: NVIDIA BlueField-3 architecture configured in NIC mode, allowing DMA access to host memory.

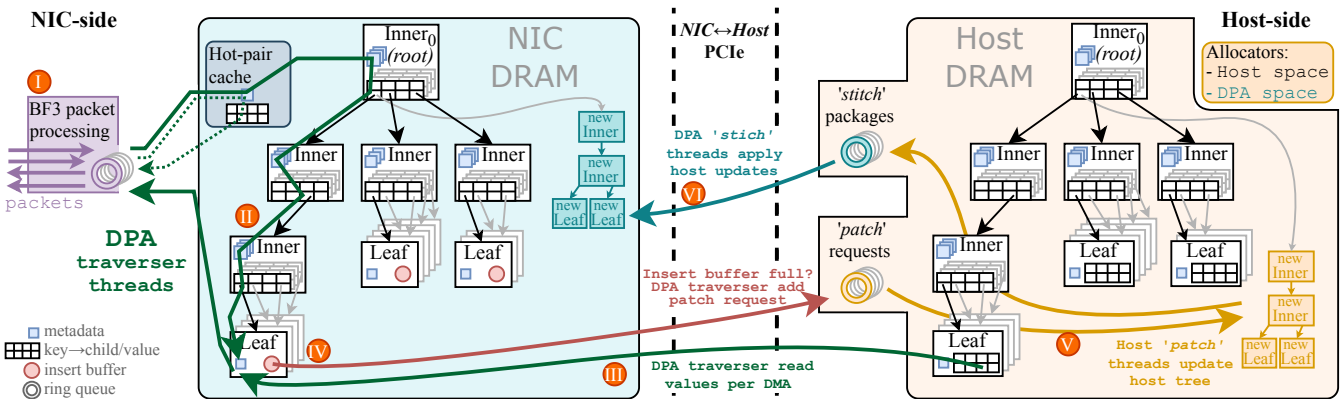


Figure 3: DPA-Store architecture: (I) Request packets are assigned on-path to a DPA traverser thread, which (II) walks a NIC-side learned index. (III) Values are fetched per DMA using leaf-level models, and (IV) inserts are placed in leaf-level insert buffers. (V) The host performs structural updates that are (VI) transactionally *stitched* back to the NIC without interrupting traversers.

pipelining.

Each DPA thread has a private L1 cache and shares 1.5 MiB L2 and 3 MiB L3 caches. The cache hierarchy is backed by a dedicated 1 GiB region of the BlueField-3 DDR5 memory, termed *DPA memory*. Additionally, the DPAs can access either the remainder of the BlueField-3 memory or the host memory, depending on the configuration. If the BlueField-3 is configured for DPA DMA access to the host, the ARM CPU is disabled [7].

DPA thread invocation can be controlled by applications on the host or on the ARM, or by defining a Transport Interface Receive (TIR) target for incoming network packets. For the latter, packet matching rules on hardware-defined fields (e.g., source or destination addresses, ports, or VLAN tags) can be used to filter relevant packets at line rate. Matched packets are placed directly into the L2 cache of the DPAs, allowing low-latency access to packet data. Unmatched packets are transparently placed on the default network path.

The BlueField-3 aims for a broad range of applications by featuring on-path and off-path compute, specialized accelerator engines, and dedicated memory. However, developing high-performance applications requires an in-depth understanding of the limitations of the architecture. For example, memory accesses to DPA memory induce a latency of nearly 500 ns and are significantly slower than DRAM accesses of a standard CPU [7]. In the context of this work, it is therefore important to minimize the number of memory accesses to the tree-based data structure, motivating the use of learned indexes with their limited number of memory accesses per tree level.

3 DPA-Store: Architecture & Implementation

In this section, we present DPA-Store, our remote KV store that supports range queries and allows stateless clients. DPA-Store uses the DPA subsystem of the BlueField-3 DPU to process incoming requests, avoiding all OS overheads and leveraging the high degree of parallelism of the DPAs.

We selected a learned index tree to reduce the number of accesses to the latency-constrained DPA memory. A DPA-memory access averages 465 ns [7], which is about five times the latency of a host DRAM access from a standard CPU. Therefore, the number of cache-line accesses per level dominates compute time and impacts performance as much as tree depth. A B$^+$-tree with the same fan-out as a learned-index tree uses binary search inside each node and accesses $O(\log n)$ cache-lines per level (see Figure 1). In contrast, a piecewise-linear learned index with error bound ε searches only a contiguous window $[p-\varepsilon, p+\varepsilon]$, which we bound to at most two DPA-memory cache-lines in inner nodes and at most three DMA-accessed cache-lines at the leaf level. We quantify this trade-off relative to a B$^+$-tree baseline in Section 4.2.5 and with a closed-form memory model in Section 4.2.6.

Given the restricted 1 GiB capacity of DPA memory, DPA-Store uses the high-capacity host memory to store values in a tree replica. Each lookup retrieves a value with a minimum number of DMA operations between NIC and host. Furthermore, the host is used for compute-heavy or blocking tree operations, such as model retraining and node splits. The host propagates tree updates to the NIC side via read-copy-update (RCU) semantics, enabling the DPAs to traverse the index tree without locks and maintaining consistency at all times.

DPA-Store employs thread-local caches for hot KV pairs. Insert and update operations are batched in a leaf-level insert buffer, reducing the number of tree updates propagated to the host. Figure 3 shows the overall architecture, which is discussed in detail in this section. We start with the NIC-side request path (Sec. 3.1), including the learned index and caching details, and then continue with the update cycle involving the host (Sec. 3.2).

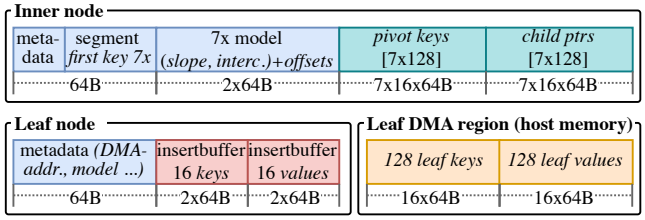


Figure 4: Layout of NIC-side nodes and leaf DMA regions.

3.1 Request Processing in the DPA Subsystem

DPA-Store uses the User Datagram Protocol (UDP) over Ethernet for transporting requests, with each request consisting of a single UDP packet. All requests are terminated on the DPA threads of the BlueField-3. We have defined packet matching rules to map each of the listening UDP ports to a designated DPA thread, effectively utilizing hardware steering rules to distribute requests over available DPAs. While every request can be served from every DPA thread, the port selection can be used by clients to facilitate load balancing or to improve cache utilization. By default, clients use a shared key hashing function so that the load is distributed uniformly across all DPA threads.

Incoming requests are consumed by the responsible *traverser* threads (176 in total), which descend the learned index tree until reaching the appropriate leaf node. Inner nodes contain seven learned models, where each model consists of a PLA over its interval, pivot keys, and pointers to the respective children (see Figure 4). We limit the number of pivot keys and child pointers to 128 per segment, chosen to match our selected ε values.

Partitioning inner nodes into seven segments allows us to fit the segments' first keys and node metadata into a single cache-line. While the traverser thread performs a binary search, comparing the operation key with the segment keys, the segment models are prefetched, overlapping compute with memory access. Having found the target segment for a key, the traverser thread evaluates the segment's model to obtain the predicted position. Again, during the computation, we prefetch the cache-lines containing the pivot keys of the current segment. The prediction is used to scan the pivot array to find the index and the pointer to the designated child node before the traverser thread moves to the next level. We store pivots and child pointers separately to compare more pivots per cache-line before incurring a single access to the child.

Once a traverser thread reaches a leaf node, it performs the requested operation. Leaf nodes contain a single learned model and an insert buffer with space for 16 key-value pairs. `INSERT`, `UPDATE`, and `DELETE` requests operate only on the insert buffer. The key-value pair (or key-delete marker for deletes) is appended using two atomic counters: one atomic increment takes place before and the other after the data write, allowing concurrent writers. We ensure that concurrent lookups read values before the corresponding key to guarantee correct mappings.

When a traverser writes into the last slot of the insert buffer, it sends a *patch* request to the host via DMA. The host can then perform the required tree operations and propagate the tree updates via *stitcher* threads back to the NIC-side tree (see Section 3.2). During this period, the NIC-side still has read-only access to the corresponding insert buffer. If a traverser cannot append to an insert buffer because it is full, the traverser re-enqueues the request for later processing.

`GET` requests must first scan the insert buffer, since it holds the most recent version of any updated or inserted KV pair. Each leaf node has a separate insert buffer, which is unlikely to be cached by the DPA thread. We prefetch it in parallel with the model computation, so a predicted position is ready for the DMA fallback in case of a miss. If the insert buffer contains the requested KV pair, the traverser exits early. Otherwise, the traverser scans the leaf keys array in host memory around the predicted position. To further reduce DMA latency, we also prefetch the keys before final scan range clamping. Once the key position in the array is found, the value is read from host memory via DMA.

`RANGE` requests reach the leaf node with the target key $k_{\min}$ in the same way as `GET` requests. Then, the insert buffer is scanned, and all values within the requested range are added to a temporary buffer, which later becomes the response. The traverser then scans the leaf keys array in host memory. When the scan reaches the end of a leaf, the traverser resumes at the next leaf by re-descending for the smallest key strictly larger than the last key returned.

Once the DPA thread finishes the request, it sends a response packet to the client. Response packets mirror the request layout but extend the type field with a status code and populate the value or range fields. For range responses, each packet carries at most 64 KV pairs to stay within a 1500-byte maximum transmission unit (MTU).

Tree traversal by DPAs is always lock-free because the tree structure is only changed on the host side, with changes being *stitched* back to the NIC via RCU semantics (see Section 3.2). Therefore, DPA threads do not stall during tree traversal, and the tree maintains consistency at all times.

3.1.1 Learned Index Parameters

We employ PLAs with slope a, intercept b, and prediction $p(k) = a \cdot k + b$. For inner nodes, each segment's model predicts the index of the child node at the next level. For leaf nodes, the model predicts the location of the value. After prediction, the linear search visits at most 2ε keys in $[p-\varepsilon, p+\varepsilon]$. We enforce an error bound of $\varepsilon_{\text{inner}} = 4$ for inner nodes (resulting in at most two cache-line accesses) and $\varepsilon_{\text{leaf}} = 8$ for leaf nodes (resulting in at most three DMA-accessed cache-lines).

We use the greedy algorithms provided by PGM [11] to compute model parameters from sorted keys. The selected

ε values are fixed and enforced during training. To optimize for node allocation overheads during updates, we restrict the size of inner node segments and leaf nodes equally to hold at most 128 pivot keys and child pointers.

BlueField-3 DPAs do not offer floating-point operations [35]. Hence, all arithmetic operations use fixed-point numbers for slopes and intercepts. To maintain precision over the full 64-bit key space, we temporarily expand operands to 128-bit during the calculations (see also [56]).

3.1.2 Hot Entry Caching on the NIC

Accesses to KV stores often follow a skewed distribution [1,5,8,53], so caching hot entries can save tree traversals and DMA fetches. Therefore, each DPA traverser thread maintains a cache (see Figure 5) consisting of (i) a three-way Bloom filter and (ii) a hash table composed of an array of buckets containing four KV pairs each. Each bucket is cache-line sized for better memory utilization. In this section, we detail the cache implementation.

Key-based per-thread caching. To ensure that the same keys are always processed by the same DPA thread, clients send packets to a designated DPA thread by selecting its UDP port via shared key hashing. Each key, therefore, has a single *home* traverser that owns its cache slot, so no cross-thread invalidation traffic is required when a key is mutated. The client also adds data needed for cache lookups, such as the Bloom hash index and the hash bucket index, to reduce the effort required for DPA computations. Each Bloom filter contains 256 bits, which is the maximum size that fits into the remaining cache-line of the traverser's thread context. The same cache-line also contains important metadata for controlling RX/TX queues and is therefore unlikely to be evicted from the thread's L1 cache. Thus, the Bloom filter incurs no additional memory access cost.

We set the cache capacity to 96 entries per traverser thread, leading to an average false-positive rate of 31%. A false positive only triggers one additional cache-line probe in the per-thread hash table and can never return an incorrect value because both keys and values are stored in the cache. With 176 traverser threads, a total of 16,896 entries can be cached. Assuming a dataset of 200M entries and an access pattern following a Zipfian distribution for $\alpha = 1$, these cached entries make up more than 50% of all requests.

Mutations and working-set shifts. `UPDATE` and `DELETE` requests on a cached key invalidate the corresponding cache slot before the response is sent. We deliberately do not update the Bloom filter on invalidation as removing a key would require either re-hashing all remaining keys in the affected bits or maintaining a counting Bloom filter. Both techniques would undermine our performance goals. We tolerate the resulting residual false-positive rate because its cost is bounded by a

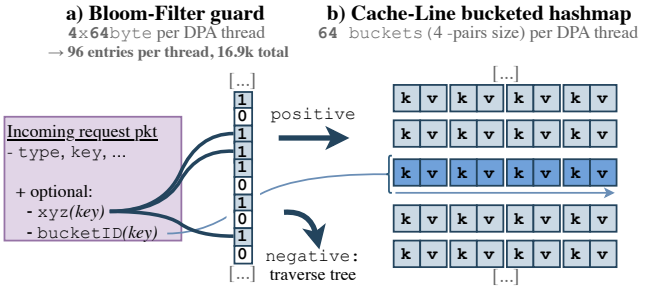


Figure 5: DPA-side hot-entry cache.

single extra hash-table probe. `INSERT`s of new keys are not actively admitted to the cache. A new key enters the cache only after a `GET` operation. This approach keeps `INSERT`s from polluting the cache with cold data. Since admission and replacement are local to the home traverser, working-set shifts are handled opportunistically, since new `GET` misses may replace old entries. Furthermore, a reset of a single traverser's cache and Bloom filter is safe and requires no cross-thread coordination.

In the case of rerouting overloaded keys to a non-home traverser (see Section 3.1.3), `GET` requests bypass the cache, whereas `UPDATE` and `DELETE` requests, as an exception, perform the required cross-thread invalidations.

Cache admission strategy. On receiving a `GET` request, the DPA thread randomly decides to admit the requested key, yielding an overall hit ratio of 25% on the workload with $\alpha = 1$. The chance of selection can be adjusted based on the workload characteristics to adjust adoption speed for new values. We treat this as a deliberate lightweight design choice under DPA constraints, leaving more sophisticated admission strategies for future work. We note that the Bloom filter and bucket array are highly optimized for cache-line access and any non-trivial admission policy must pay the cost of additional DPA-memory accesses on the hot path. To emphasize this, we evaluated a lightweight variant of the Space-Saving heavy-hitter algorithm [44] per traverser to drive cache admission. We scaled it to achieve hit ratios of up to 50% under a test workload with $\alpha = 1.1$. Due to the additional read and write per request, the overall `GET` throughput could not exceed that of the random admission strategy. With wider per-thread caches in future BlueField generations, the ability to use the full range of the 256 DPA threads, or a shared DPA-cluster admission service, we expect more sophisticated admission strategies and hotness tracking to become attractive.

3.1.3 Maximizing Delivery Rate

DPA-Store receives requests and sends responses as distinct UDP packets via Ethernet. We do not implement a connection-oriented transport such as TCP on the DPA. DPA threads are

invoked on packet arrival and run with a tight per-invocation compute budget. Additionally, they have only a few cache-lines of effective per-thread context (see Section 3.1.2). A full TCP proxy implementation would require per-flow session windows, retransmission buffers, sequence tracking, and time-driven retransmissions, all of which fit poorly into the event-driven execution model. It would also multiply per-thread state well beyond the L1 cache capacity of the DPAs. We consider the implementation of such a flow and congestion control framework overall unfit for DPA-Store and out of the scope of this work. We note that network-intercepting designs such as KV-DIRECT and MICA [23, 28] utilize UDP for similar reasons.

Retries and Idempotence. We assume that NIC-based KV stores are ill-suited to ensure reliable request transport. For example, DPA-Store has no knowledge about requests that were lost in transit due to the UDP-based transport. Therefore, DPA-Store expects a client to resend a request if it does not receive a response within a configurable time frame.

DPA-Store does not keep information about duplicate requests since doing so would require extensive metadata at every traverser and would work against the stateless-client design. We argue that the resulting transport semantics therefore permit duplicates. `GET` and `RANGE` requests are always safe to retry. Repeated `DELETE`s, `UPDATE`s, or `INSERT`s with identical arguments are handled by the host-side patcher, which ignores them in the insert buffers when recreating the respective leaf (Section 3.2). However, DPA-Store does not provide exactly-once semantics for write requests: if a write is delayed and interleaves with a conflicting write to the same key, the duplicate may be ordered after the conflicting write. If the application requires a specific ordering, including between `UPDATE`s or `INSERT`s on different keys, the client must serialize them.

A write is acknowledged to the client as soon as it has been appended to the leaf-level insert buffer, which is its linearization point in the delivered-request history. The written KV pair is then immediately visible to all subsequent `GET` and `RANGE` requests, irrespective of whether the updated subtree has already been stitched into the NIC-side index. Because every read consults the insert buffer prior to traversing the tree, and because stitches are atomic, the design of DPA-Store leaves no observable consistency gap for delivered requests.

Per-thread queues and overload. Each traverser has a 256-packet DPA receive queue, giving the DPA an aggregate capacity of 45,056 queued requests across 176 traverser threads. Whether queues fill up is a function of offered load relative to per-thread service time, not of the absolute number of clients: a single client that issues more in-flight requests than a thread can drain produces the same effect as many clients that collectively over-subscribe the same thread. In practice, we observe two regimes in which the queues fill up: (i) extreme key skew, where a single hot key is routed to its home traverser and a single thread becomes the bottleneck, and (ii) aggregate bursts that briefly exceed the average per-thread service rate of $\sim$187,500 `GET`s/s (33 MOPS / 176 threads).

To reduce loss in these regimes, clients can monitor their own response-to-request ratio as a local overload heuristic. For `GET` requests whose responses time out, a client may retry with an alternative hash that maps the affected key to a non-home traverser. The client may then throttle the rate of its outgoing requests. Each key has a single *home* thread that owns its cache slot (Section 3.1.2), so requests served by a non-home thread deliberately bypass the hot-entry cache. We understand this as a lightweight mitigation and leave the implementation of full congestion control for future research.

3.2 Update Cycle

This subsection explains how buffered writes become structural updates without blocking traverser threads. We introduce concurrent *patcher* threads on the host and *stitcher* threads in the DPA subsystem. DPA-Store guarantees that the NIC-side tree is consistent at all times by employing RCU semantics, so nodes are never modified in-place but exchanged via atomic pointer swaps. More complex tree updates are executed bottom-up until the procedure reaches the last node to be modified. Its address is updated in its parent via a pointer swap. This stitching procedure is transparent to all traverser threads.

The cache consistency protocols of the DPA subsystem guarantee that a node's address is always consistent over all DPA threads [35]. However, a traverser may descend an outdated subtree. To maintain overall consistency, epoch-based garbage collection reclaims outdated nodes and host-side DMA locations only after every traverser has moved on to its next request.

3.2.1 Host-Side Patching

The DPA traverser threads append insert, delete, and update operations to the per-leaf insert buffers. When an insert buffer becomes full, the corresponding traverser enqueues a patch request to its dedicated host-memory queue via DMA. Only the traverser that fills an insert buffer is allowed to emit a patch request, ensuring at most one patch request per buffer. On the host, a small number of *patcher* threads (four by default) process the patch requests and update the host-side tree. Patcher threads are not pinned to specific host CPU cores. Under sustained update load, they poll their queues to minimize patch-to-stitch latency. Under low update activity, the patcher threads `sleep` and are therefore subject to the OS scheduler. For read-only workloads, no patch requests are emitted, and host CPU involvement is therefore negligible.

If the patch request contains only `UPDATE` operations, the patcher modifies the values accordingly. In this case, the

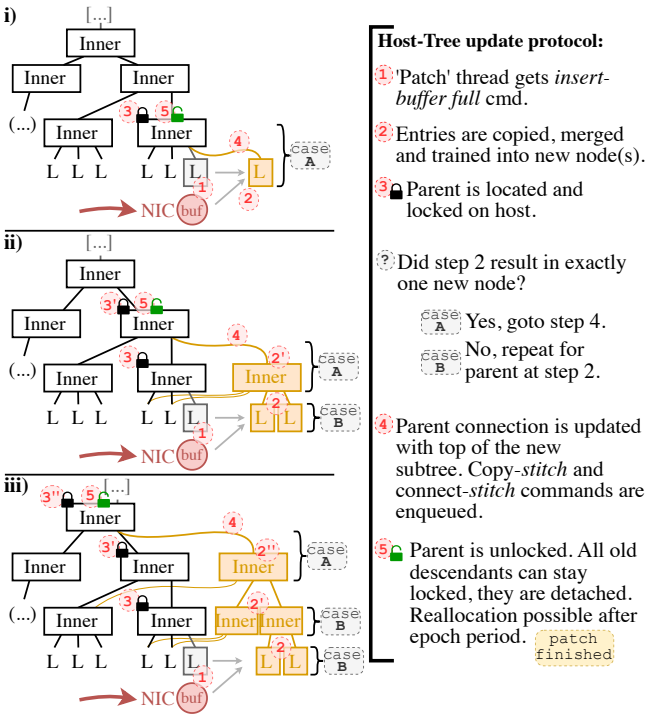


Figure 6: Host-side *patching* protocol.

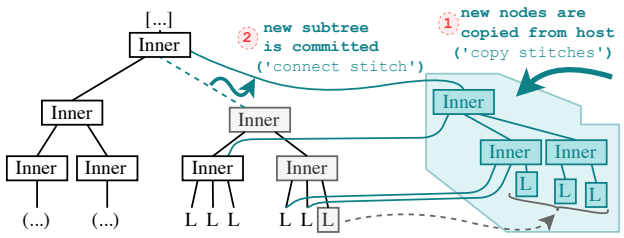


Figure 7: The NIC-tree stays valid while *stitches* are applied.

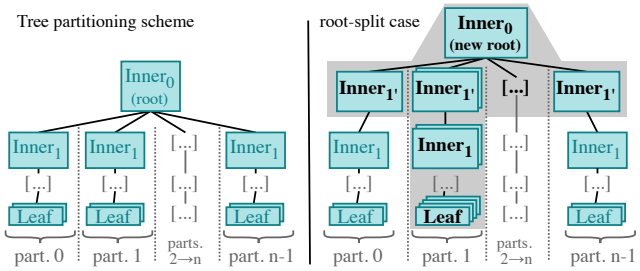


Figure 8: Tree partitioning to allow concurrent *stitches*.

patcher thread notifies the DPA-side stitcher threads to clear the insert buffer and perform no further action. If the patch contains `INSERT` or `DELETE` operations, the patcher first merges the key-value pairs with the existing leaf contents into a temporary, sorted array. The resulting array is then partitioned using PLA segmentation with error bound $\varepsilon_{\text{leaf}}$, where each segment becomes a new leaf node. This results in either a single segment that fits within the leaf capacity or multiple segments. When splitting becomes necessary, we limit segment sizes by a *retrain bound* (0.25×capacity). The resulting leaf nodes are sparsely populated, and future patch requests may be absorbed without another split, reducing the overall number of split operations.

After the leaf nodes are updated on the host, the patcher thread locates the parent of the original leaf by descending from the root. It then locks the parent for exclusive access, so that concurrent updates from sibling nodes must wait for the current one to finish. We avoid explicit parent pointers because maintaining bidirectional references under concurrency requires complex synchronization schemes.

If the retraining produced exactly one leaf, a single pointer swap in the parent is sufficient. Otherwise, the parent must also be rebuilt. We merge the new child pivots with the parent entries and apply PLA segmentation with $\varepsilon_{\text{inner}}$. PLA ranges in this step represent the inner-node segments. If the segmentation yields more ranges than the maximum allowed per node, multiple inner nodes are created, each containing a balanced number of segments. Similarly to splitting leaf nodes, we limit the number of segments within inner nodes using the *retrain bound*. Overall, this process is iterated bottom-up for every level of the tree, stopping only when the parent node does not need to be split or the root node has been reached.

Once the host tree contains the new nodes, we propagate the changes to the NIC-side tree using our stitcher threads. We enqueue the individual node updates to the stitcher queues as `COPY` stitches. Afterward, a single `CONNECT` stitch command is issued that translates to a single pointer swap, making previous `COPY` stitches effective (see Figure 6).

3.2.2 NIC-Side Stitching

The NIC-side DPA stitcher threads apply stitch commands submitted through stitcher thread queues. Stitcher threads process incoming `COPY` and `CONNECT` stitches in order, redoing the host-tree changes on the NIC side (Figure 7). The host has pre-calculated every destination address in the DPA memory space, and the stitcher threads do not have to allocate any memory. Instead, they execute the provided commands on the pointers passed from the host. To enable concurrent stitching, we partition the tree beneath the first level of inner nodes (Figure 8, left). Each parallel stitcher thread (four by default) works on a dedicated partition of the tree. The host is aware of the mapping of stitcher threads to partitions and assigns stitch commands accordingly.

Ordering of stitches and single-thread ownership eliminate most race conditions. However, root-level updates can produce conflicting stitches: (i) a `CONNECT` could reference a node that has not yet been installed by a corresponding `COPY` stitch, or (ii) newly copied and connected nodes could refer-

ence child nodes that are not yet available on the NIC-side tree. Therefore, we have implemented two safeguards. First, we delay the execution of stitches targeting nodes originating directly from a root split until their destination is available. We do this by using unique identifiers (UIDs) to probe whether the target node of a `CONNECT` stitch is already in place. Second, the stitch that installs an updated root node is blocked by a queue fence until earlier updates have completed. To preserve the partitioning scheme, the host ensures that root splits maintain balanced partitions and distribute new top-level nodes across all partitions (Figure 8, right-hand side).

3.2.3 Memory Reclamation

NIC-side subtrees become obsolete after a new version is installed using `COPY` and `CONNECT` stitches, in which case the leaf memory must eventually be freed. However, traverser threads may still descend into subtrees containing the affected nodes. Therefore, we perform epoch-based reclamation of obsolete nodes, using all DPA threads' incoming and outgoing packet counters to compute a global epoch value.

3.2.4 Bulk Loading

Bulk loading partitions a set of sorted KV pairs into PLA segments for the configured leaf error bound $\varepsilon_{\text{leaf}}$ on the host. Each PLA segment becomes a leaf node. Once all leaf nodes are available, their first keys form the scaffolding for building upper levels. We then apply the same PLA construction bottom-up. The first keys of child nodes serve as input for constructing the nodes at the next-higher level, with an error bound of $\varepsilon_{\text{inner}}$ to regulate tree fan-out. This recursive process continues until a single inner node remains, which becomes the root node.

Throughout bulk loading, the host enqueues `COPY` and `CONNECT` stitch commands to the stitcher queues. The stitcher threads assemble the initial tree following the enqueued host commands, ending with a final root-pointer stitch that makes the structure visible to traverser threads. As the host implicitly determines the tree partitions by selecting a stitcher queue for every tree update, no further actions on the NIC side are necessary.

4 Evaluation

In this section, we first provide an overview of the experimental setup before presenting an in-depth analysis of DPA-Store. Finally, we compare DPA-Store with ROLEX [26], a state-of-the-art RDMA-based KV store.

4.1 Experimental Setup

Unless stated otherwise, our test environment consists of one server and six client machines connected through a 100 Gb/s Dell PowerSwitch S5232F. The switch is Ethernet-based and supports RDMA over Converged Ethernet (RoCE). The clients use dual-socket AMD EPYC 7301 CPUs (32 cores) with Mellanox ConnectX-5 NICs. Clients use DPDK [12] for request transmission, bypassing OS bottlenecks. The server uses a BlueField-3 B3140L on a system with a 32-core AMD EPYC 9354P and 128 GB of DDR5 memory. In all experiments, including DPA-Store, the BlueField-3 operates in NIC mode, i.e., with DPA DMA access to the host and disabled ARM cores. When the DPA subsystem is not in use, the BlueField-3 behaves like a ConnectX-7 NIC. This allows us to evaluate related work on a conventional host/NIC setup. At the same time, DPA-Store executes on the same hardware and software stack, ensuring a fair comparison.

Many learned indexes are sensitive to key distributions. We ensure rigorous testing of our learned index by using the common SOSD datasets [2, 31]. They consist of the synthetic *sparse* and *dense* datasets, as well as real-world datasets derived from Facebook (*face*), Amazon (*amzn*), Wikipedia (*wiki*), and OpenStreetMap (*osmc*) workloads. The *sparse* dataset randomly selects keys from the full 64-bit range, while our *dense4x* selects 50M keys from a consecutive range keys four times that large. Unless stated otherwise, we bulk load 25M entries before each experiment and use a value of $\alpha = 0.99$ for skewed key popularities according to Zipf's Law. Similar to most related work, we use 64-bit keys and 64-bit values.

We averaged throughput values over four runs for long-running benchmarks (e.g., `GET`) and eight runs for short-running ones (e.g., inserts of new keys). Latency measurements were gathered from all clients and combined. Client start times were synchronized via MPI. Measurements showed a standard deviation of less than 5% for throughput and less than 9% for latencies.

4.2 DPA-Store Evaluation

In this section, we first investigate the memory consumption of DPA-Store. We then analyze the effects of the learned-index parameters, client parameters, and DPA/host thread counts on throughput and latency. Using our findings, we deduce default values for DPA-Store. Furthermore, we analyze whether different BlueField-3 models or changes to the BlueField-3 hardware influence performance.

4.2.1 Memory Consumption

We evaluated the overheads of the index structure compared to the raw KV data on the host side for 50M inserted KV pairs (Table 1). Since *face* and *osmc* show the highest memory consumption and overheads for values of $\varepsilon_{\text{inner}} = 4$ and $\varepsilon_{\text{leaf}} = 8$, we demonstrate that their memory impact can be significantly reduced with $\varepsilon = 16$ for both, inner and leaf nodes. In the following benchmarks, we select the larger values for ε for

Table 1: Relative overhead and NIC-side memory consumption for 64-bit key distributions and 50M entries.

Dataset (50M)	Rel. overhead of index	Overall DPA memory req.
sparse	32%	207 MB
dense4x	26%	169 MB
wiki	23%	147 MB
amzn	54%	346 MB
osmc	74%	472 MB
face	104%	672 MB
osmc ($\varepsilon_{in} = \varepsilon_{lf} = 16$)	35%	228 MB
face ($\varepsilon_{in} = \varepsilon_{lf} = 16$)	52%	332 MB

those datasets.

Compared to other learned indexes, DPA-Store chooses small ε values to minimize the cost of searching for the key around the predicted position in latency-constrained DPA memory. For example, ROLEX uses $\varepsilon \in \{128, 256\}$, allowing it to maintain much larger nodes and thus reducing per-node memory overheads. ROLEX reports a metadata overhead of 6.5% for cache data on a 500 M dataset, assuming 16 B KV pairs [26]. However, it requires all clients to hold a separate cache, increasing memory overhead proportionally to the number of clients in addition to the number of keys.

4.2.2 DPA and Host Thread Counts

In this subsection, we evaluate the impact of a varying number of traverser, patcher, and stitcher threads on the performance of DPA-Store. Figure 9 (left) shows the throughput and latencies of a `GET`-only workload with uniform key popularity on the sparse dataset. We observe that the throughput scales proportionally to the number of traverser threads. Latencies exhibit higher variance with few traverser threads but stabilize with a larger number of threads. This is because these few threads are overwhelmed by the number of incoming packets, causing them to fail to process requests in a timely manner. For values larger than 16, multiple DPA hardware cores are used, resulting in lower latencies.

Throughput increases slightly beyond 176 threads, which fully occupy eleven hardware DPA cores. The remaining usable 13 DPA threads reside on the next, partially-used physical core. We found that using the last available hardware core for both traverser and stitcher threads leads to worse throughput for `INSERT` operations. This is a result of the hardware scheduler that prioritizes NIC doorbell events. These events invoke traverser threads and therefore stall the execution of stitchers. We observed a 14% degradation in throughput for `INSERT` when mixing both types of DPA threads on one core. For both `INSERT` and `UPDATE` workloads, throughput flattens beyond four patcher threads and four stitcher threads. We note that the throughput of host-side patcher threads is limited by the NIC-side stitcher threads (see Section 4.2.8 for details).

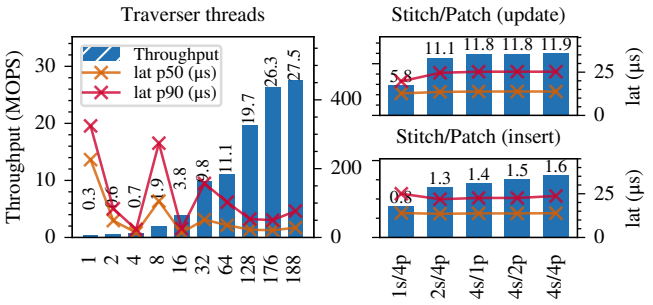


Figure 9: DPA-store throughput and latencies for different numbers of traverser, patcher, and stitcher threads. The left plot shows the number of traverser threads for `GET` requests. The right plot varies patcher/stitcher thread count for `INSERT`- and `UPDATE`-only workloads with 176 traversers.

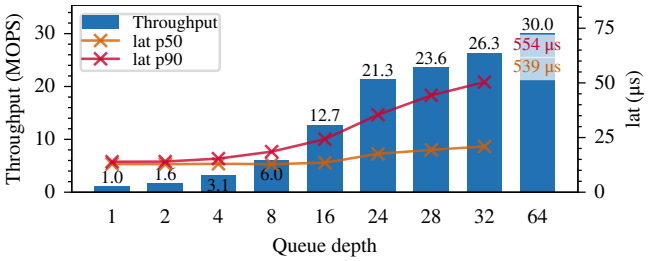


Figure 10: DPA-Store `GET` throughput and latency using different client-side queue depths.

Lessons learned: Prioritizing NIC doorbell events by the BlueField-3 requires partitioning the 189 available DPA threads into 176 traverser threads on 11 physical cores and four stitcher threads on a dedicated core, leaving nine threads unused. Furthermore, we configure DPA-Store to run four host-side patcher threads.

4.2.3 Client-Side Queue Depth

In the following, we investigate the impact of the client-side queue depth and the resulting total number of in-flight requests on throughput and latencies. Each of the six client nodes is running 31 threads. Threads issue between 1 and 64 concurrent requests. NIC-side hash tables ensure that outgoing and corresponding incoming packets are handled by the same thread and that requests lacking an acknowledgment are resent after a timeout. Figure 10 shows how client-side queue depth affects `GET` performance under a uniform key distribution.

Throughput increases significantly up to a client queue depth of 32 with a maximum of 5,952 in-flight requests. After this point, throughput continues to rise; however, latency increases beyond acceptable levels because the processing rate of DPA-Store cannot keep pace with the request submission rate. Accordingly, a queue depth of 32 is used for `GET`

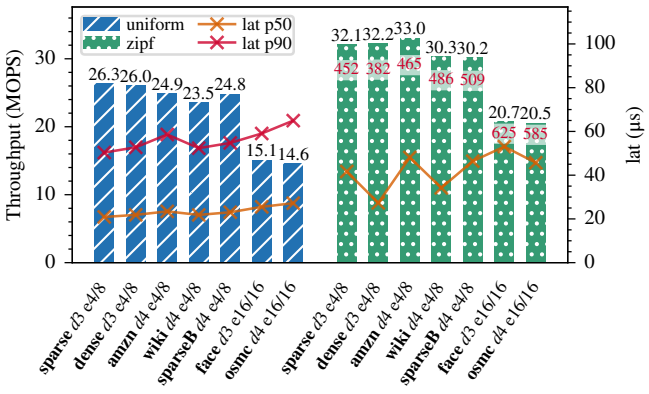


Figure 11: Results of DPA-Store using `GET`-only workloads.

requests throughout all subsequent experiments. We similarly set a queue depth of 18 for `INSERT` and `RANGE` workloads. Integrating adaptive flow control is left as future work.

4.2.4 Effect of Tree Depth and Index Error Bound

We evaluate the effect of tree depth and the error bound ε on the throughput and latency of DPA-Store. To demonstrate the effect of the tree depth, we include the *sparseBig* dataset in this section. Bulk loading *sparseBig* (consisting of 50M KV pairs) results in a tree depth of four, whereas *sparse* has a depth of three. Figure 11 shows `GET` performance for different datasets under uniform and skewed key popularity. Under uniform access, we observe a slight reduction in throughput for deeper trees because of the extra accesses to an additional inner node. For the datasets *face* and *osmc*, choosing a larger ε value causes a more significant performance degradation. This illustrates the impact of additional cache-line accesses required when verifying the model predictions (see also the DPA-Store performance model in Section 4.2.6).

With skewed key popularity, the hot-entry cache increases throughput by up to 30%, which meets our expectations in Section 3.1.2. However, tail latencies increase because some traverser threads receive a disproportionate share of requests, leading to longer wait times in the DPA queues.

4.2.5 B$^+$-tree Comparison

We chose a learned index as the fundamental data structure behind DPA-Store. To motivate this decision, we compare the `GET` throughput and latencies of the learned index tree against a B$^+$-tree baseline, measured directly after bulk-loading the data. For the B$^+$-tree we set $\varepsilon = \infty$ during construction, leading to fully packed 2 KiB nodes with 128 entries each. For lookups inside nodes, the B$^+$-tree uses standard binary search starting from the middle of the node; it does not extrapolate the start position from split keys in the parent. We deliberately keep the baseline at this canonical form because adding parent-key interpolation would itself constitute a partial learned index

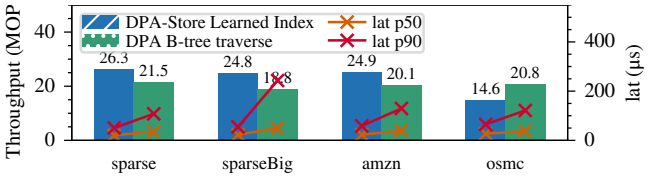


Figure 12: Comparing B$^+$-tree and learned DPA traversals.

and blur the comparison. We use 176 traverser threads for both variants, matching the DPA-Store configuration. The corresponding B$^+$-tree metadata overhead is about 145%.

Figure 12 shows that latencies are mostly higher for the B$^+$-tree than for the learned index. With the default ε, the learned index achieves higher throughput on the *sparse*, *sparseBig*, and *amzn* datasets, even though the more densely populated nodes in the B$^+$-tree yield lower tree depths. The *osmc* dataset is, in our default configuration, trained with $\varepsilon = 16$ to keep the metadata footprint manageable (see Table 1). At this setting the B$^+$-tree wins on raw throughput because the cost of scanning a 32-key learned window approaches that of binary search over a packed node, although median latency remains slightly worse due to a suboptimal balance between compute and memory accesses. Using $\varepsilon_{\text{inner}} = 4$ and $\varepsilon_{\text{leaf}} = 8$ here increases NIC-side metadata substantially. The trade-off is therefore governed by ε and the resulting in-node scan length. Small ε exploits the contiguous-window property of the learned index, while larger ε reduces the effect, making a compact B$^+$-tree competitive.

4.2.6 Analysis of `GET` Operations and Memory Accesses

The previous two subsections have shown that DPA-Store depends heavily on the performance of the memory subsystem, particularly the access latency of the DPA-addressable DDR5 memory. Prior work [7] reports average memory access times of 465 ns for DPA memory, compared with 910 ns for DMA accesses to host memory. In the following, we model the minimal duration of a full tree traversal, assuming our default parameters of $\varepsilon_{\text{inner}} = 4$ and $\varepsilon_{\text{leaf}} = 8$.

The first cache-line accessed in every inner node includes the node's metadata and the segments' first keys. The second cache-line contains each segment's model. The next one or two cache-lines (depending on the node fullness) contain the pivots, and the fifth cache-line contains the child pointer. Assuming nodes are 50% full on average, this results in an average of 4.5 cache-lines per inner node. Similarly, leaf nodes require one cache-line for metadata, up to three DMA operations to fetch the key, and one DMA operation to fetch the value. Note that we assume insert buffers to be empty in this analysis, requiring no additional memory accesses.

For a tree of depth 3, we access two inner nodes and one leaf node, resulting in overall memory access times of

$$2 \cdot \delta_{\text{inner}} + \delta_{\text{leaf}} = 2 \cdot 4.5 \cdot 0.465\,\mu\text{s} + 0.465\,\mu\text{s} + 2 \cdot 0.91\,\mu\text{s} = 6.47\,\mu\text{s}$$

Note that the cache-lines for the leaf key are sequential on the host and collapse into a single DMA. Assuming DPA scheduling overlaps one thread's computation with another thread's memory accesses, the maximum throughput is $176/(6.47\,\mu\text{s}) = 27.2$ MOPS, provided none of the relevant cache-lines are cached.

If we further assume the root node's first cache-lines (node metadata, segment keys, and models) are cached for all DPA threads, and the two memory accesses are replaced by L3 access times of 64 ns each, we compute $\delta_{\text{root}} = 1.2905\,\mu\text{s}$ and obtain a maximum throughput of 31.05 MOPS. While this model is an approximation, our evaluation of DPA-Store achieves similar results (e.g., Figure 9).

As detailed in Section 3.1, we optimized the traverser path to prefetch cache-lines optimistically, overlapping memory access with compute. Due to the relatively weak per-thread compute capabilities, we measured an improvement of 19% for this optimization.

Lessons learned: `GET` performance is limited by the access latencies to DPA memory and could increase significantly if memory access latencies were comparable with latencies of standard CPUs. For memory latencies of 100 ns, `GET` latencies could decrease to less than 2.82 μs and throughput could increase to more than 62 MOPS. Decreased memory latencies would also allow increasing ε values and becoming more memory efficient.

4.2.7 Bulk Load Performance

We evaluated the bulk load throughput using a 50M *sparse* dataset and four stitcher threads. Traverser threads are disabled during bulk loading. The full bulk load is finished on the host after an average runtime of 1,643 ms. The copying and processing of stitch requests accumulates to 1,605 ms. The bulk load copies 192 MB of tree data into DPA memory, resulting in a host-to-DPA memory bandwidth of only 120 MByte/s. This low bandwidth arises because fine-grained writes targeting scattered addresses cannot be issued efficiently from the host directly to DPA memory. Instead, the stitcher threads have to poll for incoming stitches via DMA and then first load the data into the stitcher thread's local context before writing it into DPA memory.

Lessons learned: Bulk load performance and, as shown in the following section, insert performance could be greatly increased if the BlueField-3 supported efficient transfers from host memory to DPA memory.

4.2.8 `INSERT`/`UPDATE` Performance

We evaluate throughput and latency for `INSERT` and `UPDATE` operations, mixed varying proportions of `GET` operations, using the *sparse*, *amzn*, and *osmc* datasets (see Figure 13).

Stitch requests for `UPDATE`-only patches require no data to be copied to DPA memory. Instead, only the host-side values are updated, and a stitch command resets the original DPA leaf's insert buffer. Here we reach up to 12.1 MOPS. We observe particularly low throughput for `INSERT` operations across all three datasets, with at most 1.7 MOPS. As soon as nodes are retrained on the host side from the leaf upward, `COPY` stitches are transferred. For leaves, only model parameters and DMA addresses are transferred, while inner nodes copy their complete pivots and child pointers. As we already saw for the bulk load performance, which uses the same stitching method as runtime updates, we are heavily constrained by the BlueField-3's inability to efficiently write data from host to device. The stitching bandwidth during runtime in an `INSERT`-heavy scenario cannot exceed the low rate measured for bulk loading. Although the polling stitchers perform below our expectations, future SmartNIC generations with a stronger host-to-DPA write path could address this limitation.

To rule out that the host-to-DPA bandwidth limit is an artifact of our DPA-pull stitching design, we also investigated a host-initiated copy-stitching path implemented with NVIDIA's FlexIO APIs [35, 36]. In this variant, the host directly pushes stitch payloads into DPA memory rather than enqueueing pointers that DPA stitcher threads pull via DMA. This removes the polling stitcher and one DPA-side memory hop from the critical path. The host-pushed path delivered comparable, but not better, host-to-DPA throughput, and `INSERT` performance did not improve. This suggests that the limiting factor is the underlying host-to-DPA write path on BlueField-3 rather than the direction of the copy, so we retain the simpler DPA-pulled stitching protocol.

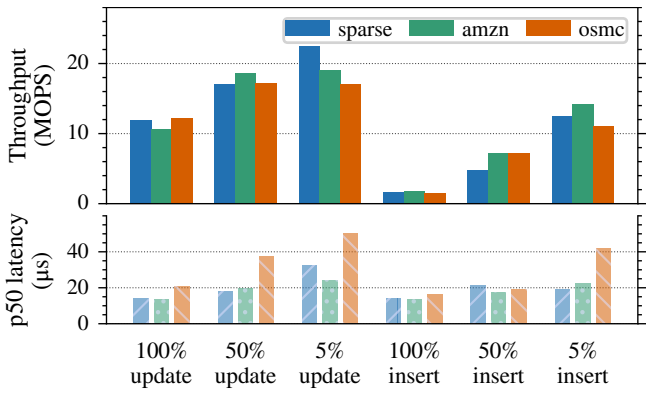


Figure 13: DPA-Store performance of `UPDATE`s and `INSERT`s.

4.2.9 BlueField-3 Model Comparison

We ran tests on an additional setup to evaluate the B3220 model of the BlueField-3 family. Unlike the B3140L with single-channel memory and one network port, the B3220 features dual-channel DPA memory and two network ports. The number and type of DPA cores, however, remain the same. Unfortunately, comparable client nodes were unavailable, and we had to generate requests with another BlueField-3 B3220 NIC via DPA programs. This allowed us to saturate the throughput of DPA-Store, but the client card lacked the compute performance to evaluate response latencies. We also disabled the restriction on the number of in-flight requests for the B3140L. Therefore, the throughput of DPA-Store was saturated for both models, enabling a fair comparison.

Figure 14 shows `GET`-only workloads on different datasets with uniform and skewed key popularities, for both NICs. We observe that throughput for the *sparse* dataset with a uniform key distribution is nearly identical for both NICs. This shows that `GET` throughput is dominated by DPA memory latencies and that moving from single- to dual-channel memory has no direct effect on DPA-Store. DPA-Store's `INSERT`, `UPDATE`, and `RANGE` throughputs also do not differ significantly.

However, with a skewed key distribution for `GET`-only workloads, The B3220 shows higher throughput than the B3140L, reaching 48.5 MOPS compared to the B3140L's 39.9 MOPS. Additionally, we ran ping tests in which the receiving DPA thread returns packets without any memory access. For ping, the B3140L model reached 44.9 MOPS, whereas the B3220 delivered 69% higher throughput. Since cached requests trigger only a single memory access and uniform accesses remain unchanged, these results demonstrate that the dual-port B3220 features stronger packet-matching hardware and therefore can process small requests faster.

4.3 Comparison with ROLEX

We compared DPA-Store against ROLEX [26], the fastest RDMA KV store that runs on modern hardware and is available for comparison, to highlight the effects of different architectural choices. We evaluated ROLEX in the same configuration as DPA-Store. This led to better throughput results for ROLEX compared to its own published results but also increased its latencies. We evaluated both solutions using YCSB workloads [8] consisting of six scenarios: A (50% reads, 50% updates), B (95% reads, 5% updates), C (100% reads), D (95% reads, 5% inserts), E (95% range, 5% inserts), and F (50% reads, 50% read-modify-update). Additionally, we include experiments for 100% inserts and 100% range queries covering 10 adjacent keys. The experiments were executed using *sparse*, *amzn*, and *osmc* datasets, which resulted in different tree depths and error-bound configurations for DPA-Store (see Figure 11). We measured with a uniform key distribution to reduce the effect of the hot-entry cache.

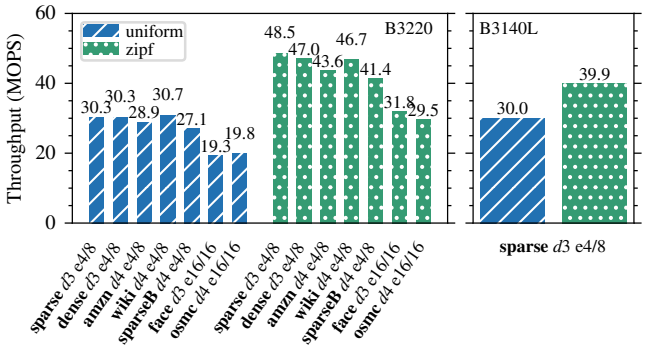


Figure 14: `GET` throughput for different BlueField-3 models.

Figure 15 shows that DPA-Store exceeds the throughput of ROLEX for the *amzn* and *osmc* datasets for YCSB-A and for all `RANGE`-only workloads. For `GET`-only workloads, DPA-Store achieves higher throughput and lower latencies compared to ROLEX on *sparse* and *amzn*. However, ROLEX achieves better results on *osmc*, where its larger values of ε are more suitable.

ROLEX shows higher throughput for `INSERT` workloads. Its clients can issue `INSERT` requests via one-sided RDMA writes directly into large server memory buffers and lazily decouple model retraining, so the structural update cost is spread over the client side instead of being concentrated in a host-to-DPA copy path. This is an effective design for insert-heavy workloads, but it relies on stateful clients that maintain learned-cache metadata and issue one-sided RDMA operations. The client-side cache also scales with the number of clients. DPA-Store makes the opposite trade-off. Clients only select a DPA traverser by hashing the key and sending a request packet, while the SmartNIC and host maintain the index. As a result, DPA-Store's `INSERT` throughput is limited by copying new node contents from the host back into DPA memory (see Section 4.2.8). The `INSERT`-only workload is therefore a worst-case comparison.

In workloads where inserts of new keys are infrequent compared to `GET` or `RANGE` requests, such as YCSB-D and YCSB-E, DPA-Store nearly reaches ROLEX's performance. Updating existing keys is less restricted by the host-to-DPA bottleneck because raw updates do not change inner nodes, and no additional copy stitches are sent. Consequently, even for high update ratios, as in YCSB-A, DPA-Store can surpass ROLEX for *amzn* and *osmc*.

Figure 15 shows lower latencies for DPA-Store compared to ROLEX in all experiments. While DPA-Store may cross the relatively slow PCIe bus to the host for DMA access during `GET` operations, the hot cache and insert buffer reduce the number of DMA operations on the lookup path. ROLEX requires RDMA operations for every request and, in uncached or mispredicted cases, additional round-trips. This difference is particularly visible for higher numbers of in-flight requests, where the extra RDMA traffic causes contention delays.

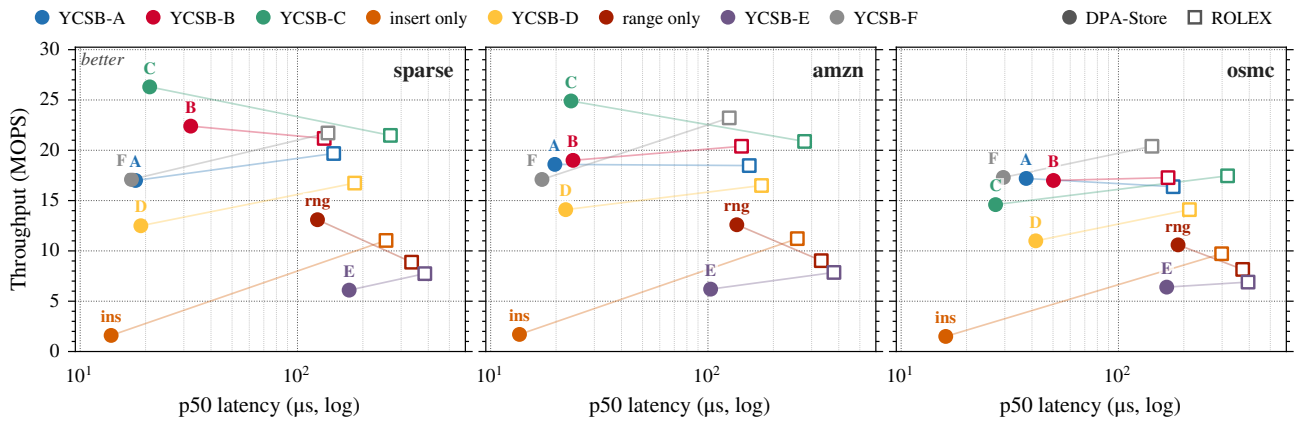


Figure 15: YCSB workloads on *sparse*, *amzn*, and *osmc* datasets. Each of the three subplots shows the throughput and latency for all workloads as 2D scatter. Connected points represent DPA-Store's and ROLEX's performance for the same workload.

Lessons learned: DPA-Store is, at the cost of more complex NIC-side hardware, faster for most workloads than the state-of-the-art ROLEX KV store while keeping clients stateless. Its main weakness is lower `INSERT` throughput, which is bounded by the BlueField-3 host-to-DPA write path. A future BlueField generation with a better DPA memory interface and efficient host-to-DPA transfers could remove this bottleneck without changing DPA-Store's client model.

5 Conclusion

We have proposed DPA-Store, a KV store residing on a SmartNIC that removes OS latencies from all operations while keeping clients stateless. By carefully analyzing the internal hardware of the BlueField-3, we have designed DPA-Store to maximize operation throughput. We have offloaded computation-heavy learned-index maintenance to the host while keeping tree traversal lock-free and highly concurrent. We have shown that DPA-Store is competitive with ROLEX and that a better DPA memory interface and host-to-DPA transfer path could significantly improve these results.

Acknowledgement

This work has been supported by the project *Big Data in Atmospheric Physics (BINARY)* (funded by the Carl Zeiss Foundation, P2018-02-003) and the project *ScalNEXT: Optimierung des Datenmanagements und des Kontrollflusses von Rechenknoten für Supercomputing* (16ME0688). This work was partly performed in the framework of the PUNCH4NFDI consortium supported by DFG fund "NFDI 39/1", Germany. Further, this research was conducted using hardware of the Institute for Automation of Complex Power Systems at RWTH Aachen University.

References

[1] Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. Workload analysis of a large-scale key-value store. In *ACM SIGMETRICS/PERFORMANCE Joint International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS), London, United Kingdom, June 11-15*, pages 53–64, 2012. doi:10.1145/2254756.2254766.

[2] Lorenzo Bellomo, Giuseppe Cianci, Luca de Rosa, Paolo Ferragina, and Mattia Odorisio. A comparative study of compressed, learned, and traditional indexing methods for integer data. In *23rd International Symposium on Experimental Algorithms (SEA), Venice, Italy, July 22-24*, pages 5:1–5:23, 2025. doi:10.4230/LIPICS.SEA.2025.5.

[3] André Brinkmann, Reza Salkhordeh, Florian Wiegert, Peng Wang, Xin Yao, Renhai Chen, Keji Huang, and Gong Zhang. Hln-tree: A memory-efficient b+-tree with huge leaf nodes and locality predictors. *ACM Trans. Storage*, 21(2):14:1–14:27, 2025. doi:10.1145/3707641.

[4] Qizhe Cai, Shubham Chaudhary, Midhul Vuppalapati, Jaehyun Hwang, and Rachit Agarwal. Understanding host network stack overheads. In *ACM SIGCOMM Conference, Virtual Event, USA, August 23-27*, pages 65–77, 2021. doi:10.1145/3452296.3472888.

[5] Zhichao Cao, Siying Dong, Sagar Vemuri, and David H. C. Du. Characterizing, modeling, and benchmarking rocksdb key-value workloads at facebook. In *18th USENIX Conference on File and Storage Technologies (FAST), Santa Clara, CA, USA, February 24-27*, pages 209–223, 2020.

[6] Haibo Chen, Rong Chen, Xingda Wei, Jiaxin Shi, Yanzhe Chen, Zhaoguo Wang, Binyu Zang, and Haibing Guan. Fast in-memory transaction processing using RDMA and HTM. *ACM Trans. Comput. Syst.*, 35(1):3:1–3:37, 2017. doi:10.1145/3092701.

[7] Xuzheng Chen, Jie Zhang, Ting Fu, Yifan Shen, Shu Ma, Kun Qian, Lingjun Zhu, Chao Shi, Yin Zhang, Ming Liu, and Zeke Wang. Demystifying datapath accelerator enhanced off-path smartnic. In *32nd IEEE International Conference on Network Protocols (ICNP), Charleroi, Belgium, October 28-31*, pages 1–12, 2024. doi:10.1109/ICNP61940.2024.10858560.

[8] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing (SoCC), Indianapolis, Indiana, USA, June 10-11*, pages 143–154, 2010. doi:10.1145/1807128.1807152.

[9] Jialin Ding, Umar Farooq Minhas, Jia Yu, Chi Wang, Jaeyoung Do, Yinan Li, Hantian Zhang, Badrish Chandramouli, Johannes Gehrke, Donald Kossmann, David B. Lomet, and Tim Kraska. ALEX: an updatable adaptive learned index. In *Proceedings of the International Conference on Management of Data (SIGMOD), online conference, June 14-19*, pages 969–984, 2020. doi:10.1145/3318464.3389711.

[10] Dragonfly. Dragonfly - An In-Memory Data Store without Limits. Accessed: 2025-08-14. URL: https://www.dragonflydb.io/.

[11] Paolo Ferragina and Giorgio Vinciguerra. The pgm-index: a fully-dynamic compressed learned index with provable worst-case bounds. *Proc. VLDB Endow.*, 13(8):1162–1175, 2020. doi:10.14778/3389133.3389135.

[12] Linux Foundation. Data Plane Development Kit (DPDK), 2015. URL: http://www.dpdk.org.

[13] Jiake Ge, Huanchen Zhang, Boyu Shi, Yuanhui Luo, Yunda Guo, Yunpeng Chai, Yuxing Chen, and Anqun Pan. SALI: A scalable adaptive learned index framework based on probability models. *Proc. ACM Manag. Data*, 1(4):258:1–258:25, 2023. doi:10.1145/3626752.

[14] Anuj Kalia, Michael Kaminsky, and David G. Andersen. Using RDMA efficiently for key-value services. In *ACM SIGCOMM Conference, Chicago, IL, USA, August 17-22*, pages 295–306, 2014. doi:10.1145/2619239.2626299.

[15] Arjun Kashyap, Yuke Li, and Xiaoyi Lu. DPU-KV: on the benefits of DPU offloading for in-memory key-value stores at the edge. In *Proceedings of the 34th International Symposium on High-Performance Parallel and Distributed Computing (HPDC), Notre Dame, IN, USA, July 20-23*, pages 11:1–11:14, 2025. doi:10.1145/3731545.3731571.

[16] Arjun Kashyap, Yuke Li, Darren Ng, and Xiaoyi Lu. Understanding the idiosyncrasies of emerging bluefield dpus. In *Proceedings of the 39th ACM International Conference on Supercomputing (ICS), Salt Lake City, UT, USA, June 8-11*, pages 807–821, 2025. doi:10.1145/3721145.3725780.

[17] KeyDB. KeyDB - The Faster Redis Alternative. Accessed: 2025-11-22. URL: https://docs.keydb.dev.

[18] Elie F. Kfoury, Samia Choueiri, Ali Mazloum, Ali AlSabeh, Jose Gomez, and Jorge Crichigno. A comprehensive survey on smartnics: Architectures, development models, applications, and research directions. *IEEE Access*, 12:107297–107336, 2024. doi:10.1109/ACCESS.2024.3437203.

[19] Andreas Kipf, Ryan Marcus, Alexander van Renen, Mihail Stoian, Alfons Kemper, Tim Kraska, and Thomas Neumann. Radixspline: a single-pass learned index. In *Proceedings of the Third International Workshop on Exploiting Artificial Intelligence Techniques for Data Management (aiDM@SIGMOD), Portland, Oregon, USA, June 19*, pages 5:1–5:5, 2020. doi:10.1145/3401071.3401659.

[20] Tim Kraska, Alex Beutel, Ed H. Chi, Jeffrey Dean, and Neoklis Polyzotis. The case for learned index structures. In *Proceedings of the International Conference on Management of Data (SIGMOD), Houston, TX, USA, June 10-15*, pages 489–504, 2018. doi:10.1145/3183713.3196909.

[21] Viktor Leis, Alfons Kemper, and Thomas Neumann. The adaptive radix tree: Artful indexing for main-memory databases. In *29th IEEE International Conference on Data Engineering (ICDE), Brisbane, Australia, April 8-12*, pages 38–49, 2013. doi:10.1109/ICDE.2013.6544812.

[22] Viktor Leis, Florian Scheibner, Alfons Kemper, and Thomas Neumann. The ART of practical synchronization. In *Proceedings of the 12th International Workshop on Data Management on New Hardware (DaMoN), San Francisco, CA, USA, June 27*, pages 3:1–3:8, 2016. doi:10.1145/2933349.2933352.

[23] Bojie Li, Zhenyuan Ruan, Wencong Xiao, Yuanwei Lu, Yongqiang Xiong, Andrew Putnam, Enhong Chen, and Lintao Zhang. Kv-direct: High-performance in-memory key-value store with programmable NIC. In *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP), Shanghai, China, October 28-31*, pages 137–152, 2017. doi:10.1145/3132747.3132756.

[24] Pengfei Li, Yu Hua, Jingnan Jia, and Pengfei Zuo. Finedex: A fine-grained learned index scheme for scalable and concurrent memory systems. *Proc. VLDB Endow.*, 15(2):321–334, 2021. doi:10.14778/3489496.3489512.

[25] Pengfei Li, Yu Hua, Pengfei Zuo, Zhangyu Chen, and Jiajie Sheng. A high-performance rdma-oriented learned key-value store for disaggregated memory systems. *ACM Trans. Storage*, 19(4):30:1–30:30, 2023. doi:10.1145/3620674.

[26] Pengfei Li, Yu Hua, Pengfei Zuo, Zhangyu Chen, and Jiajie Sheng. ROLEX: A Scalable RDMA-oriented Learned Key-Value Store for Disaggregated Memory Systems. In *21st USENIX Conference on File and Storage Technologies (FAST), Santa Clara, CA, USA, February 21-23*, pages 99–114, 2023.

[27] Pengfei Li, Hua Lu, Rong Zhu, Bolin Ding, Long Yang, and Gang Pan. DILI: A Distribution-Driven Learned Index. *Proc. VLDB Endow.*, 16(9):2212–2224, 2023. doi:10.14778/3598581.3598593.

[28] Hyeontaek Lim, Dongsu Han, David G. Andersen, and Michael Kaminsky. MICA: A Holistic Approach to Fast In-Memory Key-Value Storage. In *Proceedings of the 11th USENIX Symposium on Networked Systems Design and Implementation (NSDI), Seattle, WA, USA, April 2-4*, pages 429–444, 2014.

[29] Junyi Liu, Aleksandar Dragojevic, Shane T. Fleming, Antonios Katsarakis, Dario Korolija, Igor Zablotchi, Ho-Cheung Ng, Anuj Kalia, and Miguel Castro. Honeycomb: Ordered Key-Value Store Acceleration on an FPGA-Based SmartNIC. *IEEE Trans. Computers*, 73(3):857–871, 2024. doi:10.1109/TC.2023.3345173.

[30] Yandong Mao, Eddie Kohler, and Robert Tappan Morris. Cache craftiness for fast multicore key-value storage. In *European Conference on Computer Systems, Proceedings of the Seventh EuroSys Conference 2012 (EuroSys), Bern, Switzerland, April 10-13*, pages 183–196, 2012. doi:10.1145/2168836.2168855.

[31] Ryan Marcus, Andreas Kipf, Alexander van Renen, Mihail Stoian, Sanchit Misra, Alfons Kemper, Thomas Neumann, and Tim Kraska. Benchmarking learned indexes. *Proc. VLDB Endow.*, 14(1):1–13, 2020. doi:10.14778/3421424.3421425.

[32] Ryan Marcus, Emily Zhang, and Tim Kraska. Cdfshop: Exploring and optimizing learned index structures. In *Proceedings of the International Conference on Management of Data (SIGMOD), online conference, June 14-19*, pages 2789–2792, 2020. doi:10.1145/3318464.3384706.

[33] Memcached. Memcached - A distributed memory object caching system. Accessed: 2025-08-14. URL: https://memcached.org.

[34] Rolf Neugebauer, Gianni Antichi, José Fernando Zazo, Yury Audzevich, Sergio López-Buedo, and Andrew W. Moore. Understanding pcie performance for end host networking. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication (SIGCOMM), Budapest, Hungary, August 20-25*, pages 327–341, 2018. doi:10.1145/3230543.3230560.

[35] NVIDIA. NVIDIA BlueField-3 data-path accelerator (DPA). Accessed: 2025-08-14. URL: https://docs.nvidia.com/doca/sdk/dpa+subsystem/index.html.

[36] NVIDIA. NVIDIA BlueField Networking Platform. Accessed: 2025-08-14. URL: https://www.nvidia.com/en-us/networking/products/data-processing-unit/.

[37] Boris Pismenny, Liran Liss, Adam Morrison, and Dan Tsafrir. The benefits of general-purpose on-nic memory. In *27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), Lausanne, Switzerland*, pages 1130–1147, 2022. doi:10.1145/3503222.3507711.

[38] Redis. Redis - The Real-time Data Platform. Accessed: 2025-08-14. URL: https://redis.io.

[39] Feng Ren, Mingxing Zhang, Kang Chen, Huaxia Xia, Zuoning Chen, and Yongwei Wu. Scaling up memory disaggregated applications with SMART. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1 (ASPLOS), La Jolla, CA, USA*, pages 351–367, 2024. doi:10.1145/3617232.3624857.

[40] Henry N. Schuh, Arvind Krishnamurthy, David E. Culler, Henry M. Levy, Luigi Rizzo, Samira Manabi Khan, and Brent E. Stephens. CC-NIC: a cache-coherent interface to the NIC. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating*

Systems, Volume 1 (ASPLOS), La Jolla, CA, USA, pages 52–68, 2024. doi:10.1145/3617232.3624868.

[41] Shangyi Sun, Chunpu Huang, Rui Zhang, Lulu Chen, Yukai Huang, Ming Yan, and Jie Wu. A comprehensive study on optimizing systems with data processing units. *CoRR*, abs/2301.06070, 2023. doi:10.48550/ARXIV.2301.06070.

[42] Shangyi Sun, Rui Zhang, Ming Yan, and Jie Wu. SKV: A smartnic-offloaded distributed key-value store. In *IEEE International Conference on Cluster Computing (CLUSTER), Heidelberg, Germany, September 5-8*, pages 1–11, 2022. doi:10.1109/CLUSTER51413.2022.00016.

[43] Chuzhe Tang, Youyun Wang, Zhiyuan Dong, Gansen Hu, Zhaoguo Wang, Minjie Wang, and Haibo Chen. Xindex: a scalable learned index for multicore data storage. In *25th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP), San Diego, California, USA, February 22-26*, pages 308–320, 2020. doi:10.1145/3332466.3374547.

[44] Justin Thaler, Michael Mitzenmacher, and Thomas Steinke. Hierarchical heavy hitters with the space saving algorithm. In *14th Meeting on Algorithm Engineering & Experiments (ALENEX), The Westin Miyako, Kyoto, Japan, January 16*, pages 160–174, 2012. doi:10.1137/1.9781611972924.16.

[45] Lasse Thostrup, Daniel Failing, Tobias Ziegler, and Carsten Binnig. A dbms-centric evaluation of bluefield dpus on fast networks. In *International Workshop on Accelerating Analytics and Data Management Systems Using Modern Processor and Storage Architectures (ADMS@VLDB), Sydney, Australia, September 5*, pages 1–10, 2022.

[46] Aoyang Tong, Yu Hua, and Menglei Chen. Daldex: A dpu-accelerated persistent learned index via incremental learning. In *Proceedings of the 39th ACM International Conference on Supercomputing (ICS), Salt Lake City, UT, USA, June 8-11*, pages 535–549, 2025. doi:10.1145/3721145.3725750.

[47] Jing Wang, Qing Wang, Yuhao Zhang, and Jiwu Shu. Deft: A scalable tree index for disaggregated memory. In *Proceedings of the Twentieth European Conference on Computer Systems (EuroSys), Rotterdam, The Netherlands*, pages 886–901, 2025. doi:10.1145/3689031.3696062.

[48] Ruihong Wang, Jianguo Wang, Prishita Kadam, M. Tamer Özsu, and Walid G. Aref. dlsm: An lsm-based index for memory disaggregation. In *39th IEEE International Conference on Data Engineering (ICDE), Anaheim, CA, USA, April 3-7, 2023*, pages 2835–2849, 2023. doi:10.1109/ICDE55515.2023.00217.

[49] Xingda Wei, Rong Chen, Haibo Chen, and Binyu Zang. Xstore: Fast rdma-based ordered key-value store using remote learned cache. *ACM Trans. Storage*, 17(3):18:1–18:32, 2021. doi:10.1145/3468520.

[50] Xingda Wei, Rongxin Cheng, Yuhan Yang, Rong Chen, and Haibo Chen. Characterizing off-path smartnic for accelerating distributed systems. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI), Boston, MA, USA, July 10-12*, pages 987–1004, 2023.

[51] Jiacheng Wu, Yong Zhang, Shimin Chen, Yu Chen, Jin Wang, and Chunxiao Xing. Updatable learned index with precise positions. *Proc. VLDB Endow.*, 14(8):1276–1288, 2021. doi:10.14778/3457390.3457393.

[52] Xingbo Wu, Fan Ni, and Song Jiang. Wormhole: A fast ordered index for in-memory data management. In *Proceedings of the Fourteenth EuroSys Conference (EuroSys), Dresden, Germany, March 25-28*, pages 18:1–18:16, 2019. doi:10.1145/3302424.3303955.

[53] Juncheng Yang, Yao Yue, and K. V. Rashmi. A large-scale analysis of hundreds of in-memory key-value cache clusters at twitter. *ACM Trans. Storage*, 17(3):17:1–17:35, 2021. doi:10.1145/3468521.

[54] Shunkang Zhang, Ji Qi, Xin Yao, and André Brinkmann. Hyper: A high-performance and memory-efficient learned index via hybrid construction. *Proc. ACM Manag. Data*, 2(3):145, 2024. doi:10.1145/3654948.

[55] Yiwen Zhang, Guokuan Li, Jiguang Wan, Junyue Wang, Jun Li, Ting Yao, Huatao Wu, and Daohui Wang. Dowkv: A dpu-offloaded and write-optimized key-value store on disaggregated persistent memory. In *IEEE International Conference on Cluster Computing (CLUSTER), Santa Fe, NM, USA, October 31 - Nov. 3*, pages 271–283, 2023. doi:10.1109/CLUSTER52292.2023.00030.

[56] Wenbin Zhu, Zhaoyan Shen, Qian Wei, Renhai Chen, Xin Yao, Dongxiao Yu, and Zili Shao. Hidpu: A dpu-oriented hybrid indexing scheme for disaggregated storage systems. In *23rd USENIX Conference on File and Storage Technologies (FAST), Santa Clara, CA, February 25-27*, pages 271–285, 2025.

[57] Tobias Ziegler, Sumukha Tumkur Vani, Carsten Binnig, Rodrigo Fonseca, and Tim Kraska. Designing distributed tree-based index structures for fast rdma-capable networks. In *Proceedings of the International*

Conference on Management of Data (SIGMOD), Amsterdam, The Netherlands, June 30 - July 5, pages 741–758, 2019. doi:10.1145/3299869.3300081.

FARLock: Asymmetric RDMA Locking Made Fair

Yuehao Hu Jiatang Zhou Tianzheng Wang Keval Vora

School of Computing Science
Simon Fraser University
British Columbia, Canada

Abstract

Distributed locking is essential for coordinating access to shared resources in modern RDMA-based distributed systems. While state-of-the-art RDMA locks can deliver high-performance by introducing asymmetry (*i.e.*, treating requests that are local and remote to the lock differently), they often trade off fairness as they fail to grant locks in the expected first-come first-serve manner. This can lead to long delays for critical tasks, missing service level objectives.

We present FARLock, a fast and fair RDMA lock to solve this problem. Drawing inspiration from ticket and MCS locks, FARLock employs tickets and MCS-style handover to ensure that locks are granted strictly by arrival order. Through careful coordination between request queues and the ticket, FARLock provides strong fairness semantics with high performance. Our evaluation on a range of workloads shows that FARLock guarantees fairness and achieves lower latencies compared to prior state-of-the-art. Incorporating FARLock in a recent RDMA-based distributed indexing solution improves its query tail latencies.

1 Introduction

Distributed locks are a fundamental building block of distributed systems to coordinate accesses to shared memory across server boundaries. Importantly, they must be carefully designed to deliver both strong performance (i.e., low-latency lock acquire/release operations) *and* fairness among requesters (*i.e.*, locks are granted in a first-come-first-serve manner [15]). While both are critical and desirable, offering strong fairness is particularly important for a broad range of systems—such as database systems [8, 11, 24, 33, 35, 38], key-value stores/indexes [3, 20, 22, 23, 32, 34, 42], and distributed file systems [6, 25, 36]—to reach their service level objectives. For example, it would be unacceptable if a mission-critical database query is delayed due to unfair granting of locks that govern shared buffer accesses in a multi-tenanted database system [27]. As emerging applications [5, 19, 21, 29, 31, 39] continue to pose stringent deadline and real-time requirements, the need for fair distributed locking has become crucial to prevent biases and delays.

RDMA for Efficiency. Modern distributed systems are increasingly built around remote direct memory access (RDMA) that enables ultra-low latency and high throughput. For example, RDMA over InfiniBand HDR can deliver 200Gb/s of bandwidth and μs-level latency [2]. With kernel bypassing and zero-copy data transfer, RDMA allows machines to communicate directly with each other's memory without involving their CPUs. RDMA also provides one-sided atomics such as RDMA compare-and-swap (CAS) that offer similar interfaces to their CPU-based counterparts. These make RDMA atomics an attractive choice for designing distributed locks which can be hosted on a server and accessed via one-sided RDMA by a remote server without involving the host's CPU.

Problem: Efficiency at the Cost of Fairness. RDMA atomics guarantee atomicity only when concurrent atomic operations execute on the same RDMA-enabled NIC (RNIC) [30] as they bypass the CPU. As a result, atomicity is not guaranteed when RDMA atomics contend with local CPU memory accesses. To preserve RDMA atomicity, even local accesses must be routed through the host's RNIC [41], but this indirection is expensive than a direct local access. For RDMA-based locks, using loopback for local lock acquisitions introduces unnecessary latency and becomes a poor choice for performance-critical coordination.

Recent work [4] mitigates this issue by introducing asymmetric RDMA locking where lock requests targeting a lock that is local to the requesting thread are handled separately from those that target a lock stored on another server. This way, local requests can perform local CPU-based atomics without involving expensive RDMA operations, which are only required by remote requests. Given the large latency gap between local and RDMA memory accesses ($\sim$100ns vs. μs-level), this can greatly improve overall lock operation performance. As we elaborate in Section 2.3, however, it sac-

rifices fairness as the lock requests get reordered, leading to biased access patterns that no longer reflect true arrival order.

Our Solution. In this paper, we present FARLock, an efficient and fair RDMA lock. FARLock is the first RDMA lock that ensures lock requests are granted in the expected first-come first-served order while eliminating loopback for local requests. FARLock's design is inspired by two classic designs and combines the advantages of both: (1) ticket-based acquisition found in classic ticket locks [15] and (2) the queue-based handover strategy of the MCS lock [26]. Similar to prior work [4], local and remote requests are handled differently in FARLock, allowing them to be separately optimized and using RDMA operations only for remote requests. However, each request additionally draws a ticket from a global counter, and locks are granted in ticket order. By globally ordering all requests (local and remote) based on their arrival time, FARLock ensures fairness.

However, designing FARLock correctly with queue-based handover between requesters and global ticket ordering is challenging. Naïvely merging the two strategies could lead to inconsistency issues where the ordering of requests in queues is different from their ticket ordering. This is problematic as granting locks in ticket order would require searching the right requester in queue and handling the resulting queue manipulation properly.

To overcome this challenge, we identify the *ordering consistency* property that dictates a correlation between the ticket ordering of requests and their respective queue ordering. The property provides flexibility between local and remote requests, allowing each type to be managed in a customized manner. FARLock ensures the property is maintained throughout execution. This is achieved by reversing the order of ticket acquisition and enqueuing, and coordinating ticket assignment based on each request's position in the queues. The benefit of this approach is that it requires no additional atomicity or coordination to maintain ordering consistency.

The next challenge is coordinating local and remote requests as they safely acquire their tickets, which is non-trivial because they access the ticket differently (the local requester does not use RDMA). With ordering consistency, this challenge reduces to mutual exclusion between just two requesters, *i.e.*, the head of the local and remote queues. This allows us to leverage the Peterson's lock [28] which requires no atomics and can be implemented entirely with simple reads and writes, making it an ideal fit.

FARLock is scalable, offering low-latency in less contention scenarios thanks to RDMA, and maintaining low-latency under high contention by avoiding ordering biases. In addition, we propose a *fairness-preserving grouping* mechanism to reduce the overhead of accessing the Peterson's lock during high-contention workloads. Our mechanism groups same-type requests by sharing the Peterson's lock, allowing multiple requests to acquire their tickets together. FARLock then ensures ordering consistency by granting locks in ticket order, similar to the usual case.

On top of the mutual exclusion support provided by FARLock, we further present an optimistic reader lock extension that only requires minor changes.

We evaluated FARLock's performance on a cluster of 10 RDMA-enabled machines using various microbenchmarks and RDMA locking strategies. Our results show that FARLock achieves lower tail latencies than all other RDMA locks. This comes from its efficient handling of local requests compared to designs that rely on the expensive loopback mechanism, and from avoiding the reordering delays seen in unfair locks that eliminate loopback but sacrifice fairness.

To understand FARLock's effectiveness in practice, we further incorporated FARLock in Sherman [32], a recent high-performance RDMA-based distributed B+-tree. Our deployment showed that Sherman benefits from FARLock's unbiased lock granting especially during write-heavy scenarios as the tail latencies improve by up to 14×.

2 Background and Motivation

In this section, we give the necessary background and motivate our work. To set the stage, we first introduce shared-memory locking and RDMA which today's distributed locks are built upon. We then expand on the implications of RDMA on locking to motivate our work.

2.1 Shared-Memory Locking

Locking mechanisms for shared-memory multicore systems have motivated the design of scalable distributed locks. The simplest is the (test-and)test-and-set (TTS) [15] locks, which rely on hardware-supported primitives such as compare-and-swap (CAS) [16] to provide mutual exclusion across multiple threads. However, TTS locks do not scale with core count as contending threads need to repeatedly poll the lock word at central memory location, generating heavy inter-thread communication and interconnect traffic. TTS locks also do not provide any ordering guarantees and is known to be vulnerable to starvation. Ticket locks [15] provide strong fairness guarantees by introducing a global ticket counter. Each requester atomically obtains a unique ticket via a single fetch-and-add (FAA) or CAS operation, and the lock is granted in ticket order. However, threads still have to spin on a central location (*i.e.*, the ticket counter) while waiting for their turn, leading to performance bottlenecks. The MCS lock [26] addresses both issues (centralized contention and ordering) by organizing requesters in a FIFO queue and allowing each requester thread to spin locally in its queue node. The lock itself always points to the tail of the queue and is only modified by an atomic swap (exchange) operation. The lock is then handed over directly from each holder to its successor in the queue. This eliminates global contention and naturally

preserves fairness, since the lock handover follows the order of the queue.

2.2 Remote Direct Memory Access

Remote Memory Direct Access (RDMA) allows one machine to directly access the main memory of another machine without involving the remote CPU. On top of a recent high-bandwidth, low-latency interconnect, such as InfiniBand [1], RDMA can enable competitive latency profiles at the μs level, making it possible now to build distributed and disaggregated in-memory applications that span multiple nodes. Different from traditional TCP/IP, RDMA exhibits a verbs interface that operates around queue pairs. Applications can perform two-sided operations (send/receive) or one-sided operations (read/write) by posting an RDMA request to a queue pair. The main difference between one- and two-sided operations is that the former does not require the remote end to coordinate, thus freeing up CPU cycles. Distributed locks also mostly rely on one-sided RDMA operations. The RDMA protocol also supports atomics such as RDMA CAS and RDMA FAA, which are also one-sided and operate on single 8-byte memory words, like their x86 counterparts.

Notably, RDMA atomics provide atomicity when they are executed on the same RDMA-enabled network interface card (RNIC) [30]. At the hardware level, atomic operations and other memory accesses are performed by RNIC via the PCIe on the remote machine [41]. Since the RNIC operations are independent of those from the remote CPU, the atomicity of RDMA atomics is *not* guaranteed when they are performed concurrently with local memory accesses by the remote CPU. This leads to asymmetric performance between local and remote accesses [4] that ideally should be leveraged by RDMA locks, which is one of the motivations of our work as we elaborate next.

2.3 Fairness and Latency of RDMA Locks

Several RDMA-based lock designs achieve high efficiency by reducing network round trips through one-sided RDMA [12, 13, 37, 40, 41]. In these designs, the lock is exposed via the RNIC. This allows lock operations to bypass the CPU of the server hosting the lock and enables low-latency lock operations. Beyond adaptations of the aforementioned centralized TTS or ticket locks, the MCS lock has also been adapted to support distributed locking using RDMA. Distributed requesters directly manipulate the remote tail pointer to enqueue themselves using one-sided RDMA, and the lock is handed over along the queue whose nodes are also accessed using one-sided RDMA. This approach preserves MCS lock's fairness properties by maintaining FIFO handover ordering based on the arrival order of distributed requesters.

Due to the aforementioned consistency issue in Section 2.2 between remote accesses from an RNIC and from the local

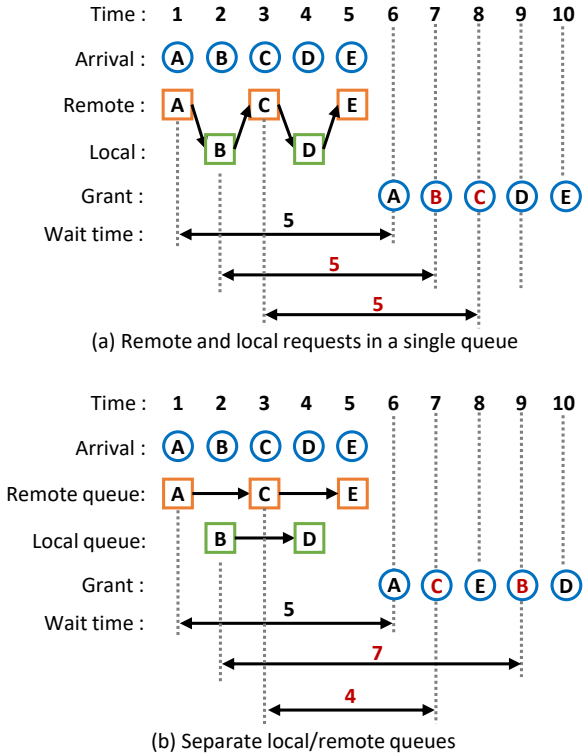


Figure 1: Performance vs. fairness guarantees in prior work. (a) Using a single queue (e.g., MCS) ensures fairness across all requests but sacrifices performance without considering asymmetry. (b) Existing asymmetry-aware locks can grant locks without considering arrival order, leading to long wait times for certain requests (e.g., B and D).

CPU, however, it is important that regardless of where the requesters are located relative to the lock (*i.e.*, whether on the same or different servers), the requester always needs to use RDMA operations to operate on the lock. This leads to high latency especially for local requesters (*i.e.*, threads on the server hosting the lock). Because RDMA does not guarantee atomicity between local and remote accesses, local requesters must interact with the lock through a loopback mechanism [4, 41], meaning their accesses are routed through the local RNIC which is unnecessarily expensive. A potential way to eliminate the expensive loopback is to treat local and remote requesters differently while enforcing correct coordination between them. However, doing so while preserving fair lock acquisition order is challenging. Maintaining fairness in terms of lock acquisition order is important because it directly impacts task latency, as discussed next.

Lock Acquisition Order. Lock acquisition order is the order in which waiting requesters acquire the lock. The common notion of fairness in lock acquisition requires that *requesters obtain locks in the order of their arrival* [15]. Such fairness is important because it prevents the additional delays that arise when requests are served out of order.

Figure 1 shows an example where 5 lock requests arrive one after the other from different tasks. For simplicity, we assume each task takes 1 unit of time to execute its critical section and release the lock. When the lock becomes available at time $t = 6$, the first scenario (a) maintains a single queue which preserves arrival order, so each request's latency directly reflects its position in the queue. This is how an RDMA-adapted MCS lock would work. In the second scenario (b), however, local and remote requests are maintained in separate queues (*e.g.*, in ALock [4]) and the lock is granted out of order as lock granting alternates between two queues. This reduces the wait times for some requests that arrive later (*e.g.*, C) but increases wait times for the earlier ones (*e.g.*, B). When considering end-to-end task latencies, such reordering can inadvertently increase latency for some requesters relative to others. Under highly contended workloads, these effects become especially pronounced, manifesting as high tail latencies.

We verified this by measuring tail latencies of RDMA-based lock implementations. We ran the RDMA implementation of the MCS lock, which guarantees fairness but incurs high overhead due to its loopback mechanism, and a recent RDMA-based lock, ALock [4], which removes this overhead by treating local and remote requesters differently but at the cost of not enforcing fairness between them. We used a cluster of 10 machines over an RDMA network (detailed setup in Section 5), with one of the machines hosting 10 locks and all machines running 12 threads that randomly (uniformly) access those locks. As shown in Figure 2, the MCS lock yields similar tail latencies for both local and remote requests. In contrast, ALock shows a gap between its local and remote latencies, which stems from its asymmetric treatment of local versus remote requests. ALock effectively prioritizes local requests and disregards arrival order across the two classes, which causes its remote tail latencies to become significantly higher than those of MCS lock.

To confirm this impact on ALock latencies, we conducted an experiment where we controlled the order of lock requests. In this experiment, we used two machines and a single lock on one of the machines. We orchestrated the lock acquisition requests such that 12 local threads issued requests before the 12 remote threads. With this setup, we captured the lock acquisition order as well as the waiting time for each request to acquire the lock. Since ALock reshuffles the requests to alternate between local and remote, as shown in Figure 3(a), the wait times to acquire the lock are not correlated to the arrival order. Moreover, the constant bouncing over the network increases the wait times even more; when considering tail latencies in this example, the last local request (arrival index 11) contributes to the highest tail latency.

The issue with shuffled processing becomes more pronounced at scale. We sampled lock requests for ALock over a longer run and captured the difference between the arrival order and the processing order. Figure 3(b) shows the standard deviation of this difference as number of threads increase.

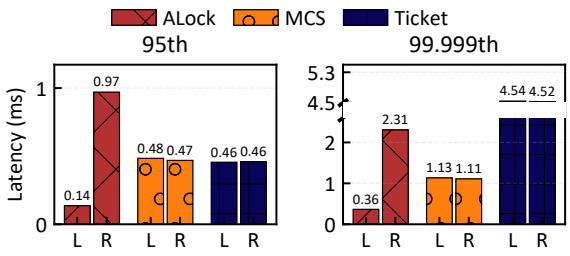


Figure 2: Tail latencies for local (L) and remote (R) requests for ALock, MCS and ticket locks.

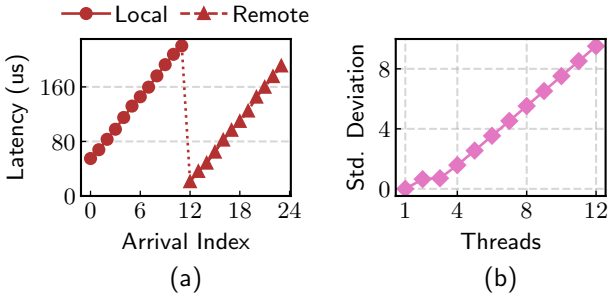


Figure 3: (a) Lock acquisition wait times versus arrival index under controlled arrival order for ALock. (b) Standard deviation of acquisition order relative to arrival order in ALock.

Effectively, with more threads, lock contention increases and more requests remain queued up waiting. This causes more requests to be shuffled which would result in longer delays and higher tail latencies.

Maintaining fairness is important to avoid arbitrary delays that increase tail latencies under high contention. Both the MCS and ticket locks provide fairness, with MCS outperforming ticket locks as expected (shown in Figure 2) because of its handover mechanism. This raises a crucial question: *how to design an RDMA lock that preserves fairness while also eliminating the costly loopback mechanism for local requesters?*

3 FARLock: Fair Asymmetric RDMA Locks

We present FARLock, an efficient RDMA lock that guarantees fairness and considers asymmetry for high performance. FARLock maintains fair lock acquisition order and does not slow down the local requests for coordination. This not only reduces lock latencies but also eliminates reordering delays that impact tail latencies.

FARLock's design is inspired from the queue-based handover strategy of the MCS lock [26] and strict lock grant ordering in ticket locks [15]. FARLock combines the benefits of both while also treating the local requests and remote requests differently. To avoid having local requests perform RDMA operations during lock acquisition, we place local and remote requests in separate queues, as shown in Figure 4.

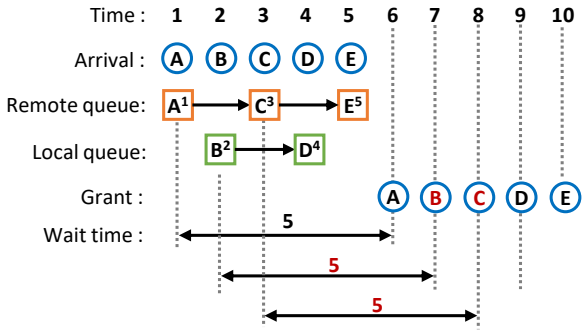


Figure 4: FARLock employs separate remote and local queues to accommodate asymmetry, and assigns requests a ticket following their arrival order to guarantee fairness.

Local requests can then enqueue using purely local operations without contending with remote requests. As requests arrive, each request obtains a ticket from a global ticket counter, and when the lock is released, the request with the next ticket in sequence acquires it. This ensures all the waiting requests, regardless whether local or remote, are globally ordered based on their arrival time and thus treated fairly. Compared to existing designs in Figure 1(b), in Figure 4, FARLock attaches a ticket to each requester according to arrival order using the global ticket counter. The lock is acquired in order shown by the ticket numbers, giving a fair wait time (5) for all requests in this example.

While the idea is straightforward, achieving it is challenging given the lock needs to maintain several separate structures—two queues and a ticket counter—that cannot be atomically updated. (1) The ordering in each queue needs to be consistent with the ordering of ticket values. (2) Accessing the ticket counter itself must be atomic. Additionally, (3) the ticketing mechanism adds more roundtrips which FARLock should try to reduce. Our solutions as summarized below constitute the major building blocks of FARLock.

Consistent Ordering. Consistency between queue ordering and ticket values entails the following invariant to hold: *any two local (remote) requests A and B with ticket of A smaller than ticket of B must be enqueued in the local (remote) queue such that A is enqueued before B.* This invariant does not enforce any requirement on requests of different types as they end up in different queues. FARLock maintains this invariant by having the requests enqueued in the queue before acquiring the ticket. By doing so, requests acquire the ticket one after the other based on their order in the queue. This is achieved similarly to the handover mechanism in the MCS lock, except that consecutive requests hand over the chance to acquire the ticket value. This way, global contention on the ticket counter is eliminated as each requester waits for its turn based on its position in the queue.

Atomic Global Ordering. Consistent local ordering orchestrates requests of same type, reducing the atomicity problem down to a single local and global request (*i.e.*, the heads of each queue). We then employ the Peterson's lock [28] which ensures mutual exclusion between two processes, similar to prior work [4]. The Peterson's lock uses only local or RDMA-based reads and writes without relying on any atomics. This allows local requesters to safely perform ticket-related lock operations locally on remotely-visible memory, while remote requesters use RDMA reads and writes. Moreover, local requests update the ticket counter directly while remote requesters use RDMA, allowing local requests to complete lock acquisition entirely without accessing RNIC.

Fairness-Preserving Grouping. Each request needs to acquire and release the Peterson's lock to obtain a ticket, adding additional roundtrips. However, we observe that this is not always necessary, especially under high contention when many concurrent requests arrive at the same time on each queue. Requests always queue up first, allowing later concurrent requests in the queue to piggyback on their predecessors to obtain the ticket (*i.e.*, passing the ticket along the queue). Since the queue already maintains ordering among same-type requests, concurrent requests could then share the same ticket. Subsequently, only the first requester in a group would need to access the Peterson's lock to allocate a new ticket, greatly reducing the lock and unlock operations on the Peterson's lock and speeding up the ticket acquisition process. Moreover, a useful side-effect of our grouping strategy is that it reduces the bouncing of lock over the RDMA network. Our grouping strategy preserves the fairness as requests of the same type are still granted the lock in the order of their queue position. We elaborate the detailed design of FARLock next.

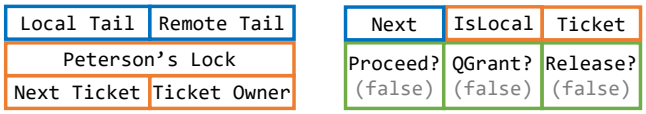
4 FARLock Design

Now we describe FARLock's design in detail, including its data structures, lock acquire/release protocols, optimizations and extensions for supporting readers.

4.1 FARLock Structures

Similar to prior work [4], FARLock employs two queues—one for local requests and another for remote requests—to optimize for asymmetric accesses. The lock itself, as Figure 5(a) shows, then includes two pointers to respectively the tail of the queues of local (`local_tail`) and remote (`remote_tail`) requesters. FARLock's lock structure also carries the necessary fields for ticketing (`next_ticket` and `ticket_owner`) protected by a Peterson's lock. As we elaborate later, this allows FARLock to enforce both consistent local ordering and atomic global ordering for fairness among lock requesters based on arrival time.

Each requester is represented by a queue node whose reference is passed in as a parameter of the lock acquire/release protocols (details later). As Figure 5(b) shows, a queue node includes fields that are similar to those typically found in an



(a) Lock structure (b) Queue node

Figure 5: FARLock structures. (a) The lock uses two queues whose tails are identified by two separate RDMA-aware pointers, a Peterson's lock and ticket (orange) related fields for ordering guarantees. (b) Each lock requester is represented by a queue node which carries a ticket and boolean fields (green, with default values) for ticket and lock handover.

MCS lock's queue node such as a `next` pointer to the immediate successor of the same type (either remote or local) and a `qgrant` field that indicates whether the requester has been "queue-granted." That is, whether the requester can be granted the lock if the lock's `ticket_owner` matches the requester's own ticket. Each queue node also carries additional fields that will be used to support the ticketing mechanism and lock handover between local and remote requesters in the locking protocols, which we describe next.

4.2 Basic FARLock

With the data structures described above, now we discuss how FARLock supports lock acquire and release operations. Here we start with the basic protocols for mutual exclusion; we discuss further optimizations including ways to reduce ordering overhead and supporting (optimistic) readers later.

Acquire. The lock acquisition protocol resembles that of the MCS lock [26], but with additional operations to accommodate for asymmetric accesses and ordering across remote and local requests. From a high level, a requester needs to (1) join the local or remote queue, (2) obtain a ticket and (3) wait for its turn to acquire the lock. Algorithm 1 shows the details. To acquire the lock, the requester first initializes its queue node with the `next` pointer set to null and `ticket` set to a sentinel `Invalid` value. Using an RDMA-aware exchange operation (`XCHG`),[1] the requester first lines up at the corresponding local/remote queue (line 3) identified by the `get_tail_ptr` function which based on the addresses of `qnode` and `lock` to return the pointer (address) of the target queue. The RDMA-aware `XCHG`'s API follows its single-node counterpart [16] to return the previous value that was stored on the target memory word. If the return value is not null, i.e., a predecessor of the same type exists, the requester then links itself with the predecessor (lines 6–7) and then needs to ensure that it will not acquire a ticket that is smaller than the predecessor's. This is done in a two-step, *consume-release* handshake protocol that is respectively executed by the requesting successor and releasing predecessor. The first step is for the requester to

[1]Could be implemented as an RDMA CAS loop or masked CAS [30]. The latter is no longer available in the latest RDMA drivers, so our implementation uses the former, following prior work [4].

Algorithm 1 FARLock's lock acquire protocol.

```
1  def acquire(lock, qnode):
2    # Install queue node to the target queue
3    pred = XCHG(get_tail_ptr(lock, qnode), qnode)
4
5    # Ensure predecessor has acquired a ticket
6    if pred != null:
7      pred.next = qnode
8      t = pred.ticket
9      pred.release = true
10     if t == INVALID:
11       spin_until qnode.proceed == true
12
13   ... acquire Peterson's lock ...
14   my_ticket = lock.ticket++ # Obtain ticket
15   ... release Peterson's lock ...
16
17   qnode.ticket = my_ticket
18
19   # Unblock my potential successor
20   if qnode.next != null:
21     qnode.next.proceed = true
22
23   if pred != null:
24     spin_until qnode.qgrant == true
25
26   # Wait for my turn
27   spin_until my_ticket == lock.ticket_owner
```

"consume" the ticket at line 8. Note that it is important to ensure that the successor (current requester) reads a stable copy of the predecessor's ticket value (i.e., the queue node memory should not be recycled until the current requester has finished using it). We therefore introduce a `release` boolean to facilitate this: the predecessor can only finish its release protocol (line 14 of Algorithm 2, more details later) after its `release` is set to true. After consuming the predecessor's ticket t, if t turns out to be invalid, the requester spin-waits for the predecessor to obtain a valid ticket (lines 10–11) before proceeding to obtain its own ticket.

After ensuring the predecessor-successor ticket ordering, now the requester moves on to acquire its own ticket at lines 13–15 which is protected by the Peterson's lock. Note that it is important for the requester to avoid assigning the ticket to its `ticket` field until the Peterson's lock is released at line 17: assigning it earlier (e.g., at line 14), however, would make it possible for a third requester to start competing for the Peterson's lock which only supports two requesters (as a result of executing lines 8–10), leading to undefined behavior. The requester then waits for the predecessor (if any) to finish its release handshake protocol by toggling the `qgrant` field (lines 23–24) and waits for its turn (line 27) to eventually be granted the lock.

Release. The release protocol starts by atomically incrementing the lock's ticket owner (line 3 of Algorithm 2) which

Algorithm 2 FARLock's lock release protocol.

```
1  def release(lock, qnode):
2    # Announce the next ticket (lock) owner
3    lock.ticket_owner = qnode.ticket + 1
4
5    # Make sure there is no successor
6    if qnode.next == null:
7      if CAS(get_tail_ptr(lock, qnode), qnode, null):
8        return
9
10   # Wait for the successor to gather my ticket
11   spin_until qnode.release == true
12
13   # Grant successor (pending ticket ordering)
14   qnode.next.qgrant = true
```

would unblock the requester spinning on line 27 of Algorithm 1, passing the lock to the next requester (if any). Algorithm 2 then follows the same approach as MCS lock's to make sure whether there is a successor using an RDMA-aware CAS and return if the CAS succeeded (i.e., there is indeed no successor, lines 6–8). The remaining steps facilitate the aforementioned consume-release handshake by (1) waiting for the `release` field to be toggled to true (by another requester at line 9 of Algorithm 1), and (2) setting the successor's `qgrant` field to true. The first step in essence waits for the acknowledgment of a successor's consumption of the releasing requester's ticket, while the second step allows the lock to be passed along the queue of the same type.

4.3 Fairness-Preserving Grouping

The use of tickets allowed the basic FARLock protocols to provide fairness, enabling requesters be granted the lock at arrival order. But this at the cost of additional roundtrips. We observe that FARLock's queue-based design can help remove most of such additional roundtrips. (1) Concurrently arriving requests could be combined to share a single ticket. (2) The queue nodes provide additional storage space for passing along information on ordering. Importantly, as we will elaborate soon, neither breaks fairness guarantees.

Based on these observations, we introduce fairness-preserving grouping to solve this problem. Akin to the classic lock cohorting idea [10], fairness-preserving grouping allows concurrently arriving requests to form a group (of a predefined maximum size) which will share the same ticket. Instead of for each requester to obtain a different ticket (hence contending on the ticket and subsequently spinning on it), only the head of a group will contend for the Peterson's lock to obtain a ticket, which is then passed along the queue. Requesters that inherited a ticket from a predecessor will then only need to rely on the queue-local grant, without spinning on the central `ticket_owner` field of the lock, effectively mitigating contention. Fairness is also preserved because grouping only

Algorithm 3 Fairness-preserving grouping. Starting line number aligned with Algorithm 1, with overlapping code shaded.

```
-1 def acquire_fpg(lock, qnode):
 0   joined_group = false
 1
 2   # Install queue node to the target queue
 3   pred = XCHG(get_tail_ptr(lock, qnode), qnode)
 4
 5   # Ensure predecessor has acquired a ticket
 6   if pred != null:
 7     pred.next = qnode
 8     t = pred.ticket
 9     pred.release = true
10     if t == INVALID:
11       spin_until qnode.budget != 0
12       if qnode.budget != get_group_size():
13         my_ticket = qnode.group_ticket
14         joined_group = true
15         goto skip
16
17   ... acquire Peterson's lock ...
18   my_ticket = lock.ticket++ # Obtain ticket
19   qnode.budget = get_group_size()
20
21 skip:
22   succ = qnode.next
23   # Pass ticket to successor if budget allows
24   if succ && qnode.budget != 1:
25     succ.group_ticket = my_ticket
26     succ.budget = qnode.budget - 1
27   else: # End of this batch
28     ... release Peterson's lock ...
29     qnode.ticket = my_ticket
30     succ = qnode.next # Refresh the successor
31     if succ: # Tell successor it is in a new batch
32       succ.budget = get_group_size()
33
34   if pred != null:
35     spin_until qnode.qgrant == true
36
37   if joined_group:
38     return # Joined group, rely on qgrant only
39   spin_until my_ticket == lock.ticket_owner
40
41 def release_fpg(lock, qnode):
42   if qnode.ticket != INVALID:
43     lock.ticket_owner = qnode.ticket + 1
44   ... Same as lines 6-14 of Algorithm 2 ...
```

reorders the concurrently-arriving requesters on the same queue. In other words, these requesters would have received consecutively increasing tickets without grouping anyway.

Realizing this idea requires slight changes to the queue node design shown in Figure 5(b) to include in each queue node two additional fields (`batch_ticket` and `budget`) to allow passing along necessary ticket information on the queue. The lock is also augmented with two more parameters to re-

spectively indicate the maximum group size of the remote and local queues; we experimentally evaluate their effect later in Section 5. With these changes on lock structure, Algorithm 3 describes the idea in detail.

Acquire. Compared to Algorithm 1, the requester attempts to join an existing group after lining up after a predecessor (lines 11–15 of Algorithm 3). The requester starts this process by waiting for its `budget` field to be filled out by its predecessor (more later). The `get_group_size()` function returns the predefined maximum group size, which is then stored in the queue node's `budget` field. As more requesters join the group, the `budget` field of each requester tracks the remaining number of allowed members in the group. The `group_ticket` field in each queue node then denotes the ticket of the current group, which will be passed along the queue until the running budget is depleted. Here, at line 12, if the requester's budget is not the initial maximum (i.e., a predecessor has started a group), it will inherit the ticket from its predecessor and skip the process of acquiring the Peterson's lock (lines 17–19). Otherwise, it proceeds to acquire the Peterson's lock to obtain a ticket as the head of the group. The algorithm then continues to inspect its successor state to pass on the ticket (lines 21–32). Note that at line 21, the group head is still holding the Peterson's lock, which will only be released by the last member of the group.

Figure 6 shows an example where a group is formed at the remote and local queue, respectively. On either side, the group head acquires the Peterson's lock and the last member in the group releases it. The "last member" can be a successor (or the head itself if group size is one) when budget is depleted or when there is no successor (lines 28–29). In the latter case, it is necessary to re-read the `next` field of the queue node to ensure the (non)existence of a successor, in case a successor has finished executing line 10 before the predecessor finishes line 29. Finally, if a requester inherited a ticket as a group member (non-head), it only needs to rely on its queue-local grant signal at lines 34–35 and directly return at line 38 to finish acquiring the lock. Otherwise, the group head will still need to wait for its turn (line 39), same as in the basic protocol.

Release. In the above lock acquire protocol, the inherited ticket is stored in `group_ticket` and the `ticket` field in each queue node is only filled when the requester is the last to acquire the lock (i.e., it is the last member) at line 29 after releasing the Peterson's lock. Therefore, the field will remain `INVALID` for all but the last member, which should increment `ticket_owner` of the lock upon release (lines 42–43). Subsequent logic for the release protocol remains the same as the basic protocol's.

4.4 Supporting (Optimistic) Readers

Beyond mutual exclusion which we have focused on so far, modern distributed applications (e.g., data analytics) also desire shared reader support to enable more concurrency by allowing multiple read-only requests to enter the critical section when no writer is present. This is often done in pessimistic approaches that explicitly enforce (shared) reader or (exclusive) writer accesses using additional fields in the lock word and/or queue nodes to track the active reader count to support this coordination [13, 37]. Such designs are generally more challenging to implement and introduce additional roundtrips through the RDMA NIC, ultimately increasing latency, especially for local requests.

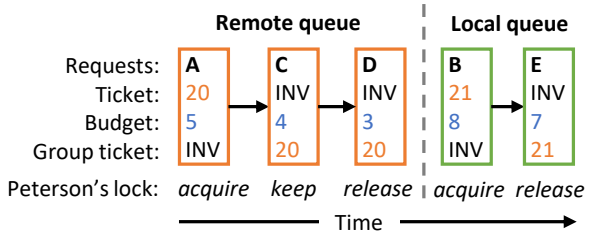


Figure 6: Fairness-preserving grouping example with five requests forming two groups on the remote ($A \rightarrow C \rightarrow D$) and local ($B \rightarrow E$) queues. The group heads (A and B) acquire the Peterson's lock, which is kept by intermediate members (e.g., C) and released by the last members (D and E). Upon releasing the FARLock, the remote group's last member D increments the ticket number from 20 to 21, which is picked up by the subsequent head request B and passed along to E.

FARLock instead advocates a simpler, optimistic approach that can be easily added on top of the protocol described in earlier sections. Classic optimistic locks [7, 14, 18] where (1) the lock carries a version number that records the number of times a writer has executed the critical section and (2) readers proceed to read without taking the lock but only need to verify that the lock is free and the version number remained the same before and after their accesses. FARLock's optimistic readers follow similar ideas, but offloads all the conflict checking that was done jointly by checking the lock state and version number (step 2) to a dedicated 0-initialized `version` field in the lock. With the dedicated version field, writers proceed by incrementing it both upon acquiring the lock (i.e., immediately after line 27 of Algorithm 1 or line 37/39 of Algorithm 3), and releasing the lock (e.g., at line 2 of Algorithm 2). Correspondingly, readers start by taking a snapshot of the version number. It then checks whether `version` is odd, and if so, that means a writer is in progress and retries until `version` is even. The reader then can start its work and will verify upon release that the version number did not change.

Such optimistic designs are inherently unfair between readers and writers, by favoring writers. However, we believe this is a reasonable tradeoff given the benefits: throughout the entire read process, optimistic readers perform no (RDMA or local) writes. Using a dedicated version field (instead of co-locating it with the lock word itself as in prior work) also frees us from handling complex interactions between readers and writers, and leads to a composable design; if optimistic readers are not needed, their support can be easily turned off.

Table 1: Locks under comparison.

Lock	Asymmetric	Fair	# Remote Spinners
TS	No	No	Unbounded
Ticket	No	Yes	Unbounded
MCS	No	Yes	0
ALock	Yes	No	1
FARLock	Yes	Yes	2

The write path incurs additional roundtrips, but can be merged with other steps (e.g., spinning or incrementing the ticket) to effectively hide the overhead, which we leave as future work.

5 Evaluation

We empirically evaluate FARLock through microbenchmarks to show the following:

- How much does tail latency improve with FARLock? (Section 5.2)
- Does FARLock deliver high throughput? (Section 5.3)
- How much does FARLock's fairness-preserving grouping strategy benefit performance? (Section 5.4)

5.1 Experiment Setup

Testbed. We perform experiments on a 10-node cluster from CloudLab's Apt Cluster.[2] Each node is equipped with two 2.6Ghz Xeon E5-2650v2 CPUs (16 cores in total), 64GB DRAM and a Mellanox FDR CX3 RNIC (similar as ALock [4]). The nodes are interconnected via a InfiniBand FDR switch, giving 56Gbps of bandwidth between nodes.

Compared Locks. Since FARLock mainly targets mutual exclusion, for fair comparuison we limit our comparison to classic and recent mutual exclusion locks. We evaluate FARLock against baselines shown in Table 1. Among them, TS, MCS and Ticket are the corresponding shared-memory locks but adapted to use RDMA atomics to work in distributed scenarios. ALock [4] is the asymmetric lock that misses fairness, maintaining two queues to serve local and remote requests separately. ALock also supports batching to reduce frequently bouncing over the network. We evaluate against the default ALock and its batched version; for this, we tested various batching parameters and found that ALock performs best when using batch sizes 20 and 5 for its local and remote queues. We use `ALock (20, 5)` to indicate this configuration of ALock with batching. Finally, FARLock denotes FARLock without grouping, and FARLock-G represents FARLock with grouping. Unless otherwise noted, we set group size to 5 and explore the impact of different group sizes later.

Benchmark Settings. All locks are hosted on a single node, and the rest of the nodes act as remote requesters. Each node runs 12 threads to issue lock requests concurrently. This defines the 12 threads from the hosting node as local requesters, and the 12 threads from each remaining node as remote requesters. We vary the number of locks on the hosting node between 1 (high lock contention) and 240 (low lock contention) to test different contention levels. Inside the critical section, each thread performs a busy-wait loop for a specified number of cycles to simulate work. Accesses to the CPU's last-level cache (LLC) typically complete within a few tens of nanoseconds [9], and RDMA routinely exhibits μs-level latency [11,17]. This presents a speed gap of 10–100$\times$ between CPU cache and RDMA accesses. We account for this gap by assigning different critical section sizes for local and remote requests to model realistic workloads where local operations are typically faster than remote operations. We set the remote critical section size to 15,000 cycles ($\sim$5μs) and the local critical section size to 300 cycles ($\sim$100ns).

Metrics. Each experiment runs for 20 seconds, and use the middle 10 seconds to collect results to rule out impacts of different thread progresses and stragglers. We measure the throughput and latency of jobs where each job includes a lock acquire operation, its critical section and the unlock operation. We present tail latencies to showcase delays experienced from (un)fairness by the slowest requests, measured as the 95th, 99th, 99.9th, and 99.999th percentile of a latency distribution. Each experiment is repeated for three times, and standard deviations across multiple experiments were 0.97-14.88% on average, with highest being 14.88% for 99.999th percentile latency measurements. We therefore report the averages of three runs for each experiment.

5.2 Latency

Figure 7 shows tail latencies when using 1 and 10 locks (representing high contention scenario) and Figure 8 shows tail latencies when using more than 10 locks. Across different percentiles, as we include more locks, contention becomes lower, leading to better performanc (lower latency) for all locks. FARLock exhibits the lowest local tail latencies compared to all other locks, and competitive remote latencies across all scenarios. By grouping (five) requests, FARLock-G gives very similar tail latencies as FARLock.

High Contention (1 – 10 locks / Figure 7). We observe up to 62$\times$ lower local tail latencies compared to other fair locks (MCS and Ticket), while 8$\times$ lower than ALock and 55-5898$\times$ lower than TS. In these scenarios, no delays are introduced for FARLock (benefit of fairness) and requests are processed in their arrival order. The other two fair locks, Ticket and MCS, exhibit much higher local tail latencies as their local requests are routed through RNIC (loopback) which is expensive. FARLock's latencies are lower than those of ALock because ALock ends up reordering requests, resulting in more delays and network bounces as locks get handed over between

[2] https://docs.cloudlab.us/hardware.html

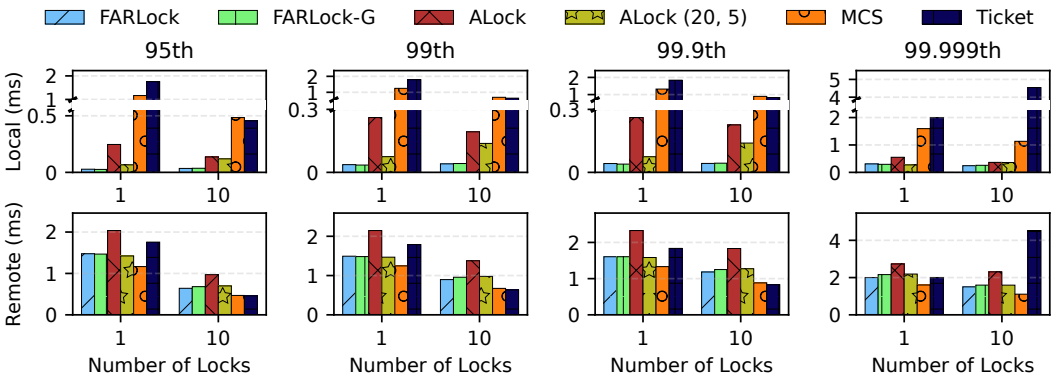


Figure 7: Tail latency for local (top) and remote (bottom) requests under high contention with 1 and 10 locks. Latencies for TS are not shown here since they are very high (2-171× higher than the next highest data point).

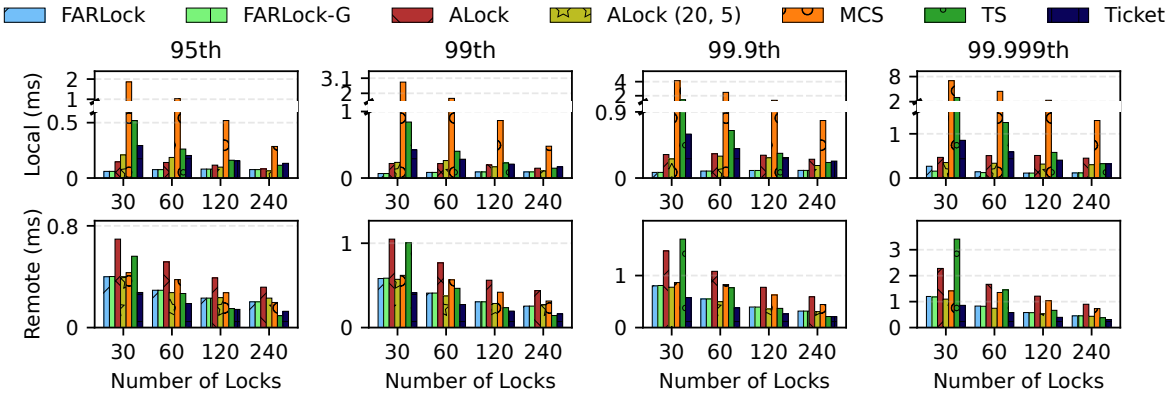


Figure 8: Tail latency for local (top) and remote (bottom) requests under decreasing contention.

different requests. ALock(20, 5) reduces the effect of such bouncing via batching (similar observation as the original paper [4]), but still suffers from reordering of requests which causes arbitrary delays and results in higher tail latencies compared to FARLock. MCS consistently provides competitive remote latencies with under high contention (i.e., low number of locks). This is because the MCS lock requires fewer instructions than FARLock to process a lock request. FARLock provides competitive remote latencies, often lower than the unfair ALock. Finally, TS shows the highest tail latencies due to its well-known starvation issue (resulting from unfairness).

Decreasing Contention (beyond 10 locks / Figure 8). FARLock still has lower tail latencies because of its advantages over MCS and Ticket (no loopback for local requests), and over TS and ALock (no reordering due to fairness). The impact of using more RDMA oprations in FARLock compared to Ticket is observed in their tail latencies as the number of locks increase beyond 30.

Comparing ALock with and without batching shows that its batching is effective in reducing remote tail latencies as batching processes more remote requests before bouncing over the network for handover. However, this benefit is marginal when compared to FARLock.

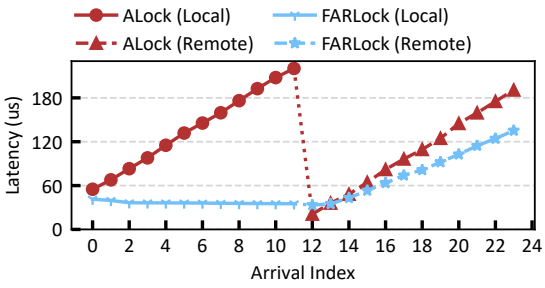


Figure 9: Latency for controlled arrival order.

Fairness Study. We perform the same experiment from Figure 3(a) where we controlled request arrival order. Along with measuring the acquisition latencies for ALock, this time we also measure for FARLock. We use two machines, and 12 local requests are submitted followed by 12 remote requests. As shown in Figure 9, latencies for FARLock are lower than that for ALock because FARLock processes requests in order, and hence, all local requests get processed followed by all remote requests. This eliminates the continuous bouncing

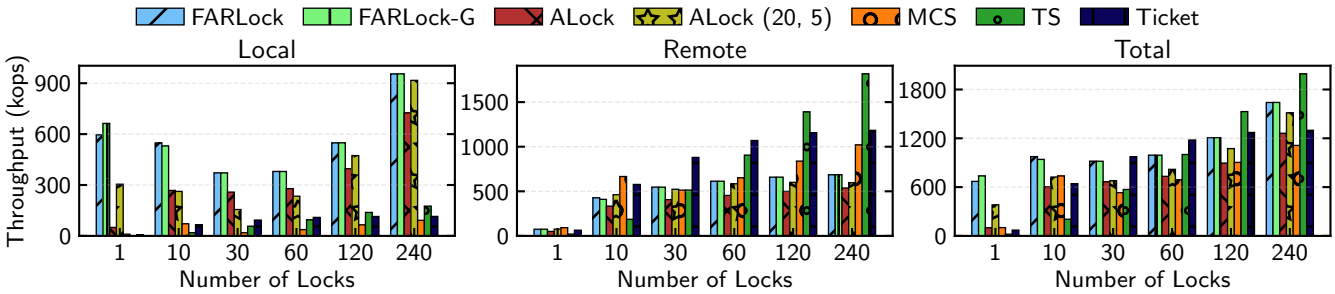


Figure 10: Throughput under different contention.

delays that ALock's ordering introduces, and results in lower latencies overall; the highest latency for the last remote request is 135.30μs which is much lower than highest latency with ALock (incurred by the last local request).

5.3 Throughput

While fairness benefits tail latency, we also assess FARLock's ability to deliver high throughput. Figure 10 compares the local, remote and total throughput of FARLock and other locks. FARLock's benefits for local requests are visible as it delivers the highest local throughput across the board with up to 891× higher than MCS, TS, and Ticket. MCS and Ticket show similar remote throughputs as FARLock under 1 lock, with both providing higher remote throughput as the number of locks increases; this is due to fewer instructions involved in remote paths of those two locks. While TS also enjoys high remote throughput under low contention for similar reasons, the expensive remote retries during high contention (with 1 and 10 locks) significantly limit its throughput compared to FARLock.

Compared to ALock, FARLock achieves 11.9× and 2× higher local throughput with 1 and 10 locks respectively, and this gap closes gradually with more locks. This is because ALock's reordering mechanism forces the lock to be handed over across nodes more often, which is less favorable when more requests line up on fewer locks. FARLock, on the other hand, is able to handle more lock requests on the same node depending on the arrival order which results in less bouncing. For instance, in the highest contention scenario (1 lock), we observed that FARLock frequently performs up to 12 consecutive local lock handovers before serving a remote request. In contrast, remote requests tend to hand over the lock on the same node far less often. Batching in ALock(20, 5) reduces its bouncing issue, with ALock(20, 5) providing much higher local throughput. However, it still consistently forces handover across nodes, albeit less often, which limits its throughput.

Finally, we observe a dip in FARLock's local throughput going from 1 lock to 30 locks which gets balanced by a sharp increase in remote throughputs. This is likely due to a side effect of fairness combined with efficiency in local requests. For the single lock scenario, local requests queue up faster than remote requests as the latter must go through RDMA with higher latency. As the number of locks increase, remote requests have more chance to queue up across different locks since local requests get spread out, resulting in remote requests being served more often (fair ordering in FARLock) compared with just a single lock.

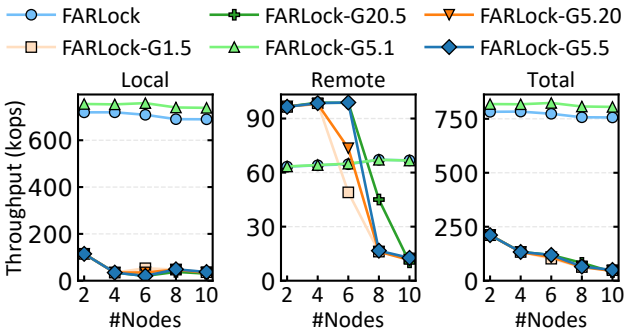


Figure 11: Throughput for FARLock with grouping. FARLock-GX.Y means at most X local and Y remote consecutive requests are grouped together while preserving fairness.

5.4 Fairness-Preserving Grouping

Grouping in FARLock is achieved by sharing the Peterson's lock across consecutive waiting requests. We evaluate its by measuring improvements in throughput. Since each group would contain requests of the same type, grouping in FARLock can be configured differently for local and remote requests. Figure 11 compares the throughput achieved by grouping 5 and 20 requests of each type, with a single lock and varying number of nodes to capture increasing remote requests.

Since FARLock optimizes local requests, grouping them improves local throughput. However, we observe that grouping remote requests leads to decreasing throughput as contention increases with more nodes. We hypothesize this is due to the remote queue getting emptied more quickly, causing more frequent bounces, which additionally cancels out the benefits brought by local grouping. The total throughput without remote grouping remains high, and we observe grouping 5 local requests gives higher total throughput. As observed in previous experiments, this grouping gives very similar tail latencies to when grouping is disabled, while there is a visible increase in throughput for high contention single lock scenarios (Figure 10).

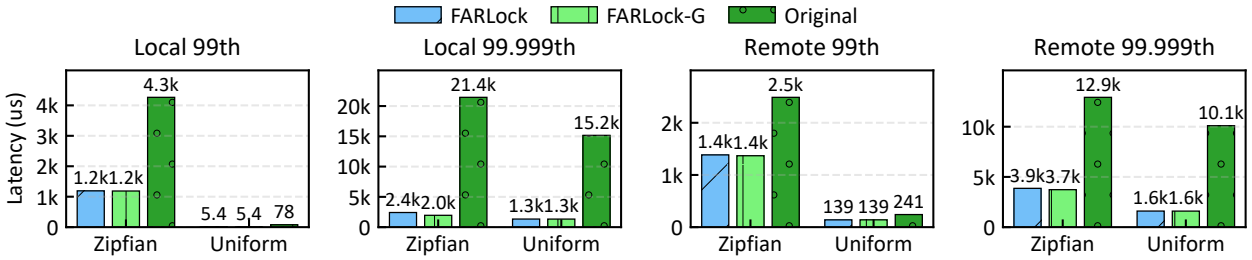


Figure 12: Tail latency of Sherman queries when using FARLock instead of its original lock.

6 FARLock in Practice

We demonstrate how FARLock can improve practical systems using a database indexing scenario. Sherman [32] is a recent RDMA-based distributed B+-tree that uses locks to coordinate access to leaf nodes. Its original lock implementation is a local ticket lock with a global spin lock for hashed memory locations. Sherman uses RDMA atomic operations to acquire locks on remote memory locations (*i.e.*, it does not use asymmetric RDMA locking), and passes the lock ownership locally with the local ticket lock in each node until a predetermined local budget. This is an unfair RDMA locking strategy as it reshuffles requests. We replaced this lock with FARLock while keeping Sherman's use of RDMA operation combination to hide remote access latencies.

We deployed Sherman on the same 10-node cluster used in Section 5. One of the nodes emulates as a memory server (MS) as well as compute server (CS), while the remaining 9 nodes act as compute servers. The MS hosts 40GB shared memory, and each CS maintains 500MB cache. The system is warmed up with 800 million key-value pairs (8-byte keys and 8-byte values). To stress test locks, we focus on update operations over random keys, emphasizing tail latencies that can impact service-level objectives under high contention. We vary the key distribution between uniform and Zipfian (θ = 0.99, skewed) to test different contention levels. Figure 12 shows improvements in tail latency when using FARLock. We observe 11-14$\times$ and 3.6-11$\times$ lower local tail latencies with uniform and Zipfian distributions respectively, and 1.7-6$\times$ and 1.8-3.5$\times$ lower remote tail latencies respectively. This is due to FARLock's efficient local request handling combined with its fair ordered processing which eliminates delays caused by reshuffling. We also observed similar throughput with and without FARLock, indicating that Sherman's high throughput is retained.

7 Related Work

Our work is most related to recent work on RDMA based locks and lock managers. Ziegler et al. [41] gives guidelines for correct and scalable synchronization using one-side RDMA which we also follow in this work. ShiftLock [13] is a reader-writer lock that also adopts the MCS-style handover mechanism, where the lock holder explicitly transfers ownership to the next requester before exiting the critical section. Similar to MCS and FARLock, ShiftLock strictly enforces FIFO ordering for writes, ensuring fair serialization among exclusive lockers. However, ShiftLock does not leverage asymmetry, so both local and remote requesters must use RDMA operations. ShiftLock further implements mechanisms to handle node failures; considering node failures is interesting future work for FARLock.

DSLR [37] is a decentralized distributed lock manager for reader-writer locks using one-sided RDMA. It adapts the principles of ticket locks and assigns monotonically increasing tickets to requesters. Although DSLR has a short lock acquire path and ensures fairness, DSLR also does not distinguish between local requester and remote requester. Local requesters still need to use RDMA to acquire locks. Citron [12] proposes a distributed range lock manager with one-sided RDMA. It leverages a segment tree to map dynamic range requests to tree nodes, allowing clients to secure specific data ranges without global serialization. However, their underlying locking mechanism is orthogonal to Citron's architecture. Beyond RDMA, FISSLOCK [40] reduces CPU overhead using programmable switches by implementing a high-performance, centralized in-network lock manager. By mediating all acquisition and release traffic, the switch maintains global visibility over the system's lock states. Programmable switches have limited memory and compute power and it remains to be seen how this approach can be combined with RDMA-based locks such as FARLock to overcome these limitations.

8 Conclusion

We presented FARLock, an efficient and fair RDMA lock. FARLock ensures lock requests are granted in the expected first-come first-served order while eliminating expensive loopback for local requests. FARLock enables efficient lock handover across different queues while adhering to a global ticket ordering. We developed a fairness-preserving grouping mechanism in FARLock to reduce operation overheads, and proposed optimistic reader lock extensions. Our evaluation shows that FARLock achieves lower tail latencies than other RDMA locks as it enables efficient local requests while avoiding reordering delays (effect of fairness). We also evaluated FARLock's practical effectiveness and observed that it significantly boosts query latencies in a state-of-the-art RDMA-based database indexing strategy.

References

[1] Infiniband architecture specification, 2025.

[2] Infiniband roadmap, 2025.

[3] Ahmed Alquraan, Sreeharsha Udayashankar, Virendra Marathe, Bernard Wong, and Samer Al-Kiswany. LoLKV: The logless, linearizable, RDMA-based Key-Value storage system. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*, pages 41–54, Santa Clara, CA, April 2024. USENIX Association.

[4] Amanda Baran, Jacob Nelson-Slivon, Lewis Tseng, and Roberto Palmieri. ALock: Asymmetric Lock Primitive for RDMA Systems. In *Proceedings of the 36th ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '24, page 15–26. ACM, June 2024.

[5] Soroush Bateni and Cong Liu. NeuOS: A Latency-Predictable Multi-Dimensional Optimization Framework for DNN-driven Autonomous Systems. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pages 371–385, 2020.

[6] Wei Cao, Zhenjun Liu, Peng Wang, Sen Chen, Caifeng Zhu, Song Zheng, Yuhui Wang, and Guoqing Ma. PolarFS: an ultra-low latency and failure resilient distributed file system for shared storage cloud database. *Proc. VLDB Endow.*, 11(12):1849–1862, August 2018.

[7] Sang K. Cha, Sangyong Hwang, Kihong Kim, and Keunjoo Kwon. Cache-conscious concurrency control of main-memory indexes on shared-memory multiprocessor systems. In *Proceedings of the 27th International Conference on Very Large Data Bases*, VLDB '01, page 181–190, 2001.

[8] Yanzhe Chen, Xingda Wei, Jiaxin Shi, Rong Chen, and Haibo Chen. Fast and general distributed transactions using RDMA and HTM. In *Proceedings of the Eleventh European Conference on Computer Systems*, EuroSys '16, New York, NY, USA, 2016. Association for Computing Machinery.

[9] Intel Corporation. Memory performance in a nutshell. Technical Article, 2023. Available online.

[10] David Dice, Virendra J. Marathe, and Nir Shavit. Lock cohorting: a general technique for designing NUMA locks. In *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '12, page 247–256, 2012.

[11] Aleksandar Dragojević, Dushyanth Narayanan, Miguel Castro, and Orion Hodson. FaRM: Fast remote memory. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, pages 401–414, Seattle, WA, April 2014. USENIX Association.

[12] Jian Gao, Youyou Lu, Minhui Xie, Qing Wang, and Jiwu Shu. Citron: Distributed range lock management with one-sided RDMA. In *21st USENIX Conference on File and Storage Technologies (FAST 23)*, pages 297–314, Santa Clara, CA, February 2023. USENIX Association.

[13] Jian Gao, Qing Wang, and Jiwu Shu. ShiftLock: mitigate one-sided RDMA lock contention via handover. In *Proceedings of the 23rd USENIX Conference on File and Storage Technologies*, FAST '25, USA, 2025. USENIX Association.

[14] Rachid Guerraoui and Vasileios Trigonakis. Optimistic Concurrency with OPTIK. In *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '16, 2016.

[15] Maurice Herlihy and Nir Shavit. *The Art of Multiprocessor Programming, Revised Reprint*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 2012.

[16] Intel Corporation. Intel 64 and IA-32 architectures software developer manuals. October 2016.

[17] Anuj Kalia, Michael Kaminsky, and David G. Andersen. Using RDMA efficiently for key-value services. In *Proc. ACM SIGCOMM*, Chicago, IL, August 2014.

[18] Viktor Leis, Michael Haubenschild, and Thomas Neumann. Optimistic lock coupling: A scalable and efficient general-purpose synchronization method. *IEEE Data Eng. Bull.*, 42:73–84, 2019.

[19] Peiliang Li, Xiaozhi Chen, and Shaojie Shen. Stereo R-CNN based 3D object detection for autonomous driving. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pages 7644–7652, 2019.

[20] Pengfei Li, Yu Hua, Pengfei Zuo, Zhangyu Chen, and Jiajie Sheng. ROLEX: A scalable RDMA-oriented learned Key-Value store for disaggregated memory systems. In *21st USENIX Conference on File and Storage Technologies (FAST 23)*, pages 99–114, Santa Clara, CA, February 2023. USENIX Association.

[21] Neiwen Ling. Time-sensitive AI System for Physical Agents. In *Proceedings of the 23rd Annual International Conference on Mobile Systems, Applications and Services*, pages 671–672, 2025.

[22] Yi Liu, Minghao Xie, Shouqian Shi, Yuanchao Xu, Heiner Litz, and Chen Qian. Outback: Fast and communication-efficient index for key-value store on disaggregated memory. *Proc. VLDB Endow.*, 18(2):335–348, October 2024.

[23] Baotong Lu, Kaisong Huang, Chieh-Jan Mike Liang, Tianzheng Wang, and Eric Lo. DEX: Scalable Range Indexing on Disaggregated Memory. *Proc. VLDB Endow.*, 17(10):2603–2616, June 2024.

[24] Haodi Lu, Haikun Liu, Yujian Zhang, Zhuohui Duan, Xiaofei Liao, Hai Jin, and Yu Zhang. Fast distributed transactions for RDMA-based disaggregated memory. In *Proceedings of the 2025 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC '25, USA, 2025. USENIX Association.

[25] Youyou Lu, Jiwu Shu, Youmin Chen, and Tao Li. Octopus: an RDMA-enabled distributed persistent memory file system. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, pages 773–785, Santa Clara, CA, July 2017. USENIX Association.

[26] John M. Mellor-Crummey and Michael L. Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Trans. Comput. Syst.*, 9(1):21–65, feb 1991.

[27] Vivek Narasayya, Ishai Menache, Mohit Singh, Feng Li, Manoj Syamala, and Surajit Chaudhuri. Sharing buffer pool memory in multi-tenant relational database-as-a-service. *Proc. VLDB Endow.*, 8(7):726–737, February 2015.

[28] Gary L. Peterson. Myths about the mutual exclusion problem. *Inf. Process. Lett.*, 12:115–116, 1981.

[29] Harry A Pierson and Michael S Gashler. Deep learning in robotics: a review of recent research. *Advanced Robotics*, 31(16):821–835, 2017.

[30] Hemal Shah, Felix Marti, Wael Noureddine, Asgeir Eiriksson, and Robert Sharp. Remote Direct Memory Access (RDMA) Protocol Extensions. RFC 7306, June 2014.

[31] Chunlin Tian, Xinpeng Qin, Kahou Tam, Li Li, Zijian Wang, Yuanzhe Zhao, Minglei Zhang, and Chengzhong Xu. CLONE: Customizing LLMs for Efficient Latency-Aware Inference at the Edge. *arXiv preprint arXiv:2506.02847*, 2025.

[32] Qing Wang, Youyou Lu, and Jiwu Shu. Sherman: A Write-Optimized Distributed B+Tree Index on Disaggregated Memory. In *Proceedings of the 2022 International Conference on Management of Data*, SIGMOD '22, page 1033–1048, New York, NY, USA, 2022. Association for Computing Machinery.

[33] Ruihong Wang, Jianguo Wang, Stratos Idreos, M. Tamer Özsu, and Walid G. Aref. The case for distributed shared-memory databases with rdma-enabled memory disaggregation. *Proc. VLDB Endow.*, 16(1):15–22, September 2022.

[34] Xingda Wei, Rong Chen, and Haibo Chen. Fast RDMA-based ordered Key-Value store using remote learned cache. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 117–135. USENIX Association, November 2020.

[35] Xingda Wei, Jiaxin Shi, Yanzhe Chen, Rong Chen, and Haibo Chen. Fast in-memory transaction processing using RDMA and HTM. In *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP '15, page 87–104, New York, NY, USA, 2015. Association for Computing Machinery.

[36] Jian Yang, Joseph Izraelevitz, and Steven Swanson. Orion: A Distributed File System for Non-Volatile Main Memory and RDMA-Capable Networks. In *17th USENIX Conference on File and Storage Technologies (FAST 19)*, pages 221–234, Boston, MA, February 2019. USENIX Association.

[37] Dong Young Yoon, Mosharaf Chowdhury, and Barzan Mozafari. Distributed Lock Management with RDMA: Decentralization without Starvation. In *Proceedings of the 2018 International Conference on Management of Data*, SIGMOD '18, page 1571–1586, New York, NY, USA, 2018. Association for Computing Machinery.

[38] Erfan Zamanian, Carsten Binnig, Tim Harris, and Tim Kraska. The end of a myth: distributed transactions can scale. *Proc. VLDB Endow.*, 10(6):685–696, February 2017.

[39] Fotios Zantalis, Grigorios Koulouras, Sotiris Karabetsos, and Dionisis Kandris. A review of machine learning and IoT in smart transportation. *Future Internet*, 11(4):94, 2019.

[40] Hanze Zhang, Ke Cheng, Rong Chen, and Haibo Chen. Fast and scalable in-network lock management using lock fission. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, pages 251–268, Santa Clara, CA, July 2024. USENIX Association.

[41] Tobias Ziegler, Jacob Nelson-Slivon, Viktor Leis, and Carsten Binnig. Design guidelines for correct, efficient, and scalable synchronization using one-sided rdma. *Proc. ACM Manag. Data*, 1(2), June 2023.

[42] Pengfei Zuo, Jiazhao Sun, Liu Yang, Shuangwu Zhang, and Yu Hua. One-sided RDMA-Conscious Extendible Hashing for Disaggregated Memory. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, pages 15–29. USENIX Association, July 2021.

ARTIFACT EVALUATED
USENIX
AVAILABLE

When DDIO Meets Page Coloring: Revisiting DDIO Performance with Sepia

Changwoo Song[†], Sanghyun Kim[†], Jinhyeok Oh[†], Qizhe Cai[*], Joonsung Kim[†], and Jaehyun Hwang[†]
[†]*Sungkyunkwan University*
[*]*University of Virginia*

Abstract

In modern host network stacks, Direct Cache Access (DCA) technology, such as Intel's Data Direct I/O (DDIO), plays a crucial role in packet processing by allowing received packet data to be accessed directly from the last-level cache (LLC) instead of host memory. However, DDIO performance is often constrained by the leaky DMA problem, where the packet data is evicted from the LLC before processing completes due to the limited capacity of the DDIO-reserved portion.

In this study, we revisit this issue and reveal that conflict misses, rather than capacity limitations alone, are the primary co-contributor to LLC misses in DDIO. Our in-depth analysis indicates that introducing *page coloring* to DDIO can increase the effective LLC capacity by 77.8–94.4% over Linux. Motivated by this insight, we present Sepia, a color-aware page allocator that reduces LLC misses during packet processing (including copy-to-user) by leveraging a deeper understanding of LLC behavior. Under configurations that jointly avoid capacity and conflict misses, our Linux kernel prototype saturates a 200Gbps link using only 3.5 CPU cores, 2.5 fewer than the default Linux stack, while maintaining low LLC miss rates (∼0.4%). This improves total throughput per unit of core utilization by ∼1.51× across diverse setups and real-world applications.

1 Introduction

Prior studies on Intel's Direct Cache Access (DCA) implementation, Data Direct I/O (DDIO) [33], have highlighted both its strengths and limitations [9, 14, 22, 24, 41, 58, 60]. On the positive side, for example, DDIO can improve network throughput by ∼1.4×, on top of well-established TCP optimizations such as TCP Segmentation Offload (TSO), Generic Receive Offload (GRO) and Jumbo Frames, by reducing CPU cycles required for data copy between the kernel and user space [14]. Tail latencies can also be improved by ∼21.5% with extended operations during DDIO [25]. On the other hand, the DDIO-reserved portion of the LLC is highly constrained, which limits DDIO's effectiveness, as packet-related cache lines are often evicted before processing completes. For example, Intel's default configuration reserves only two LLC ways for DDIO, corresponding to 3–8MB in Intel Ice Lake microarchitectures [35]. Our experiments show that CPU efficiency (i.e., total throughput per unit of CPU utilization) drops by ∼46% as the number of concurrent TCP connections contending the same LLC increases from 1 to 6 on a 200Gbps link, with ∼60.4% LLC miss rates (§3). This phenomenon, commonly known as the *leaky DMA* problem, has also been reported in prior studies [14, 24, 40, 51, 52, 58, 60, 64]. While conventional wisdom suggests that it stems from the limited DDIO-reserved capacity, we find this explanation to be incomplete. Our key findings are:

What "capacity" means depends on traffic load. Memory pages are loaded into the LLC via DDIO writes or CPU reads. On a DDIO write hit, DDIO updates the corresponding cache line; on a DDIO write miss, it allocates a new line into one of the DDIO-reserved ways. In contrast, CPU reads bring data into the LLC only on read misses, and newly allocated lines may be placed in arbitrary ways according to the cache's replacement policy. Therefore, when the LLC can hold the active page working set (i.e., the memory pages used for NIC DMA and network processing), DDIO tends to hit and update existing lines, even if they reside outside the DDIO-reserved ways. Thus, capacity misses occur when the working set exceeds the *full LLC capacity*. Under high load, however, when the LLC cannot hold the increased working set, DDIO write misses become more frequent, allocating new lines into the reserved ways. As a result, capacity misses occur when incoming traffic exceeds the *DDIO-reserved capacity* (§3.2).

Cache conflicts reduce the effective capacity. Our analysis further reveals that even when load stays within capacity, LLC miss rates can still increase. This is because the page working set constructed by the default Linux memory allocator is not evenly distributed across the LLC sets, leading to significant cache conflict misses (§3.3). Moreover, today's sliced LLC architecture (de facto standard in modern CPUs; Figure 2) can make this skewed distribution even worse. Our experiments with real page-address traces on our sliced LLC indicate that cache conflicts can occur even when the page working set

size is only 46.2% of the full LLC capacity! (§4.2)

Based on these observations, we adopt the page coloring technique to reduce conflict misses and increase the effective LLC capacity. While integrating page coloring into real systems has been viewed as challenging [12, 59], we find that applying it to DDIO scenarios presents new opportunities in the following aspects. First, prior studies [13, 42, 63, 66, 67] often rely on complex coloring as applications may have irregular memory access patterns, which significantly increases its logic complexity. In contrast, network stacks show highly deterministic behavior; for example, in the kernel, page sequences for DMA are dictated by device drivers, and payload pages are typically accessed during copy-to-user operations and then recycled (userspace stacks exhibit similar behavior). This predictability simplifies the coloring logic and enables a simple, uniform policy for all page working sets across cores. Second, coloring in virtual address space may suffer from poor efficiency as physical pages are scattered across memory. However, pages for NIC DMA can be pre-allocated over a large contiguous region in advance, enabling effective page coloring. In combination, these properties make DDIO-oriented, color-aware page allocations more effective. Building on these insights, we design *Sepia*, a color-aware page allocator that realizes practical page coloring for CPU-efficient network I/O processing. Sepia consists of two simple design components (§4.3):

Sepia Manager: The manager pre-allocates a large contiguous memory chunk and models it as a two-dimensional page array based on the cache structure, such that pages in the same row share the same color (i.e., the same cache sets). The manager then constructs per-core colored page pools[1] in a way that ensures page colors are evenly distributed and conflicts are avoided. This design also allows each core to operate independently without tracking other cores' status, minimizing management overhead while keeping colors balanced across cores on average. It also tracks page availability within each pool, as page-recycle timing naturally varies.

Sepia Allocator: Given that contiguous pages have different colors in our page pools, the Sepia Allocator adopts a simple *stride-1* allocation pattern, selecting colors sequentially (§4.1). Even when Sepia encounters unavailable pages (not yet recycled into the pool), it preserves sequential ordering to balance utilization across colors, selecting the next available page with the same color. This strategy enables low-conflict allocations from the pools. When wrapping around in the color sequence, it prefers the earliest available pages in each color, helping reduce the active working set size at low load.

We implement our Sepia prototype in the Linux kernel 6.6 (§5) and evaluate its performance across diverse experimental setups and real-world applications, including SPDK, Nginx, and Memcached (§6). Our implementation is available at https://github.com/skku-syslab/sepia. We outline several key benefits of Sepia as follows:

- Sepia increases the effective LLC capacity theoretically by 77.8% over Linux, utilizing 82.1% of the total LLC even under an imbalanced slice-hashing configuration.
- Sepia saturates a 200Gbps link using 3.5 cores while keeping near-zero LLC miss rates in the best-case scenario; the default Linux network stack requires 6 cores.
- Sepia is fully compatible with existing network applications, requiring no modifications. Our experiments demonstrate that Sepia improves CPU efficiency (throughput per core) by $\sim$1.51$\times$ compared to the default Linux network stack.

While Sepia is effective for applications that benefit from DDIO, our study comes with two caveats. First, building Sepia's page pools relies on LLC architectural details, such as the width of the set-index field (Figure 2). Although we provide a detailed methodology for color-aware page allocations through §4, we tested it on a limited set of recent Intel processors, including Intel Xeon Gold 6354 (Ice Lake) and Xeon 6526Y (Emerald Rapids). Thus future CPU models may require updates to this methodology. Nonetheless, our study focuses more on a comprehensive, end-to-end understanding of network processing, spanning NIC drivers, the LLC, the network stack, and applications, under DDIO. Second, Sepia's performance gains may depend on system configurations; for instance, the benefits diminish when excessive traffic from multiple cores contends for a small LLC capacity, as capacity misses are not the focus of our design. Instead, Sepia fills a missing piece by complementing prior work on capacity-miss issues—e.g., LLC architectural redesigns [24, 64], working-set reduction [51, 52], and traffic shaping [58]. Therefore, our goal is to show how Sepia, along with these solutions, enables further CPU-efficiency gains.

2 Background

This section provides a brief review of the page lifecycle in Rx packet processing, the modern LLC architecture, and DDIO operations. We focus on the ConnectX-series NIC driver, though most modern NICs follow a similar design.

Lifecycle of pages in Rx packet processing. Modern datacenter NICs expose multiple Rx queues, typically one per CPU core, so each queue is mapped to a particular core. Packet processing is therefore distributed across cores by steering packets to the corresponding queues using policies such as Receive Side Scaling (RSS) and accelerated Receive Flow Steering (aRFS). The Linux kernel maintains per-queue Rx ring buffers and page pools (see Figure 1), enabling lock-free per-core processing for multi-queue NICs. At initialization, ring-buffer entries (descriptors) are associated with pages and posted/synchronized with the NIC, allowing the device to locate available descriptors and the corresponding

[1] Throughout the paper, we assume a typical multi-queue NIC configuration in which NIC Rx queues, their corresponding Rx rings, and CPU cores have a one-to-one mapping. Thus, each core has a dedicated page pool for its Rx ring. Otherwise, page pools can be constructed per Rx queue.

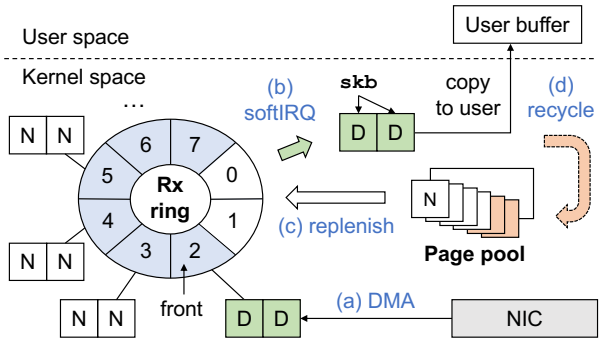


Figure 1: **Page lifecycle with Rx ring and page pool** (N=new pages, D=DMAed pages): **(a)** the NIC DMAs incoming packets into the corresponding ring-buffer pages; **(b)** the softIRQ handler allocates skbs, detaches the DMAed pages from the Rx ring, and hands them off to the skbs; **(c)** consumed ring entries are replenished with new pages from the page pool; **(d)** after network processing (including data copy), the pages are returned to the page pool for reuse. See §2 for more details.

page addresses for DMA writes when packets arrive. In ConnectX, Rx descriptors are implemented via multi-packet work queue elements (MPWQEs) [1], where each element consists of 64 pages to efficiently serve multiple packets arriving at high line rates. Network operators can tune the number of WQEs by adjusting the ring-buffer size and MTU using Ethtool [4]. Internally, drivers convert these two parameters into the WQE count proportionally. For example, in a default Linux setup with 1024 Rx ring entries and jumbo frames (9000-byte MTU), the driver initializes 64 WQEs per ring, corresponding to 16MB of pages per core.

After packets are DMAed into the descriptor pages, the softIRQ handler allocates socket buffers (skbs, the Linux packet representation) and maps the DMAed pages into those skbs for subsequent packet processing. Meanwhile, the driver refills the Rx ring by drawing new pages from the per-core page pool[2]. Once processing completes—including data copy to userspace—the pages are returned to the page pool.

Modern LLC architecture. With DDIO support [33], incoming Rx packets are steered to the LLC rather than to DRAM. Thus, a solid understanding of modern LLC architecture is essential for explaining DDIO behavior. Unlike the monolithic private caches (L1/L2), modern CPUs adopt a sliced LLC architecture [47, 61]. Each LLC slice is shared across multiple cores and connected via a high-speed interconnect, enabling parallel access and improving overall memory bandwidth. Figure 2 illustrates a high-level view of the typical LLC architecture and its physical address mapping mechanism. For each memory reference, a slice-hash function takes both the tag and set bits as input to select a target slice; once the slice is determined, the set index is derived from the set bits (as in a monolithic cache), and the remaining tag bits are used to identify the matching cache line. Note that LLC microarchitectural details (e.g., the slice-hash function, the width of the set bits, the set mapping policy) are undocumented and vary across CPU platforms, necessitating in-depth analysis and reverse engineering with carefully designed microbenchmarks. Following well-known methodologies [47], our experiments reveal that, for our Intel Ice Lake system, the LLC comprises 2048 sets (11-bit index), 12-way associativity, and 26 slices. We further confirm that the slice-hash function uses the entire tag and set bits, and that the set index is derived directly from the set bits without any additional hashing.

[2]In practice, pages are recycled from the memory-pool/page-pool and fall back to the default Linux page allocator when the page pool is empty; we omit these details in Figure 1 for the sake of simplicity.

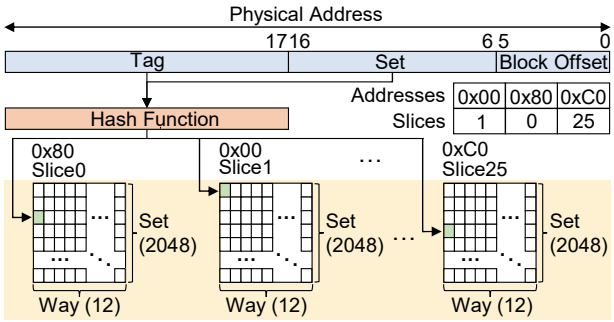


Figure 2: **LLC architecture and physical address mapping for Intel Ice Lake.** When address 0xC0 is accessed, the slice hash (from the full set of *tag* and *set* bits) maps the address to a slice (e.g., slice 25). Given a set index of 3, the access targets the cache line in set 3 of slice 25. See §2 for more details.

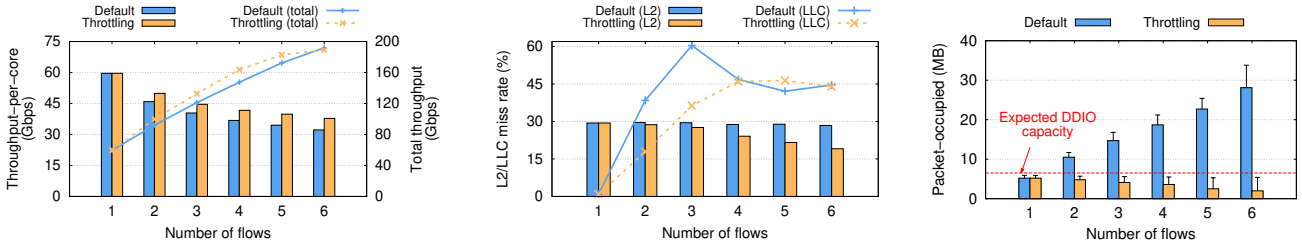
DDIO write hits and misses. DDIO uses way partitioning, reserving a subset of cache ways for DDIO writes (i.e., device-initiated writes to the LLC), typically two ways per set, with behavior similar to CPU store instructions: when the cache lines for the descriptor-page addresses are already present in the corresponding LLC slices (in any way, not only the DDIO-reserved ones), they are updated in place (**DDIO write hit**); otherwise, DDIO allocates new lines within the DDIO-reserved ways (**DDIO write miss**) [24, 33, 48, 60]. In either case, subsequent CPU reads (data copy) can hit in the LLC and avoid cold misses unless those lines have been evicted.

3 Revisiting Leaky DMA Problem

This section analyzes DDIO performance trends in modern Linux kernel stacks and identifies the underlying cause of the leaky DMA problem, focusing on Rx-side processing where CPU bottlenecks arise. We first describe our measurement setup (§3.1) and revisit the leaky DMA behavior with DDIO. We then present key observations and insights (§3.2), which lead to our motivation (§3.3).

3.1 Measurement Setup

Following the measurement methodology of prior studies [14, 15], we use a testbed consisting of two physical machines directly connected via a 200Gbps link to focus on CPU-bottleneck scenarios. Each machine has a 2-socket Intel Xeon Gold 6354 3.0GHz processor (18 cores per socket),



(a) Throughput-per-core (Gbps). (b) L2/LLC miss rate (%). (c) Total packet-occupied (MB).

Figure 3: **Leaky DMA performance with DDIO.** (Default setting) Traditionally, the leaky DMA problem has been attributed to cases where the volume of incoming packets exceeds the DDIO capacity, causing packets stored in DDIO-reserved cache ways to be evicted before processing completes. This results in reduced throughput-per-core and increased LLC miss rates. (Throttling setting) However, throttling incoming traffic to stay within the DDIO capacity is insufficient to fully resolve the issue. See §3.2 for more details.

1.25MB per-core L2, 12-way 39MB LLC, 256GB DRAM, and an NVIDIA ConnectX-6 (200Gbps) NIC. Both machines run Ubuntu 20.04 with the Linux kernel 6.6. We enable TCP Segmentation Offload (TSO), Generic Receive Offload (GRO), Jumbo frames (9000B), Dynamically-Tuned Interrupt Moderation (DIM), and aRFS to maximize receiver-side CPU efficiency during network processing. We disable hyper-threading, IOMMU, and `irqbalance`, since these settings maximizes performance in our measurements. Unless stated otherwise, we use default values for all system parameters.

We use iperf [21] to generate long-lived TCP flows and measure total throughput, overall CPU utilization, L2 and LLC miss rates. Throughput-per-core is calculated as total throughput divided by total CPU utilization, representing the throughput achievable by one fully utilized core. This metric serves as an indicator of CPU efficiency in network processing. In addition, we measure the packet-occupied memory, the total memory consumed by incoming packets that have not yet completed processing (i.e., in-flight bytes); this metric allows us to estimate whether total in-flight bytes exceed the expected DDIO-reserved LLC capacity (hereafter, DDIO capacity). In our testbed, the DDIO capacity is approximately 6.5MB (39MB$\times$2/12). We instrument the Linux kernel to measure packet-occupied memory in both the Rx socket buffer and the backlog queue, which holds packets when the corresponding socket is temporarily locked by user-level system calls [14]. To minimize overhead, we sample every 100 softIRQ events and record both the mean and the maximum with error bars (Figure 3(c)).

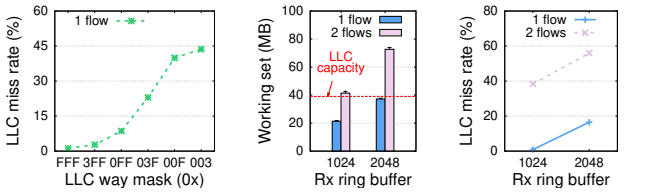
3.2 Understanding DDIO Miss Behaviors

We begin by reproducing the leaky DMA phenomenon observed with DDIO. Specifically, we measure throughput as the number of TCP flows increases, with each flow running on a separate core that shares the same LLC. Figure 3(a) shows that the Linux network stack (labeled "Default") benefits from the LLC with a single flow but experiences a ~46% drop in throughput-per-core as the number of flows increases from 1 to 6, primarily due to high LLC contention; correspondingly, the LLC miss rate rises to ~60.4%, as shown in Figure 3(b).

(1) Single-flow case: Near-zero miss rates due to a small working set (DDIO write-hit regime). In the single flow case (one core), the NIC driver holds 16MB of pages by default (§2). We further observe that ~6MB of additional pages are needed to replenish *page-consumed* Rx descriptors after the DMAed pages are handed to `skb`s (matching the maximum TCP Rx buffer memory size), so a single-flow replenishment cycle spans approximately 22MB in total. The key LLC behavior is that, even if DMAed pages are evicted from the DDIO-reserved ways (e.g., when more than two lines contend for the same set and slice), they are eventually reloaded into the LLC (likely into non-DDIO ways) by the CPU during the copy-to-user operation. Therefore, if the LLC can accommodate the entire page working set (~22MB), subsequent DDIO writes tend to hit in the LLC, as DDIO can leverage all LLC ways rather than only the two DDIO-reserved ways. This explains why the single-flow case sees near-zero miss rate (0.9%) in Figure 3(b). To support this observation, we rerun the single-flow experiment while reducing the number of LLC ways available to that core from 12 (LLC mask: 0xFFF) to 2 (LLC mask: 0x003), as shown in Figure 4(a). With fewer ways, the reloaded pages are more likely to be evicted from the LLC, resulting in more DDIO write misses.

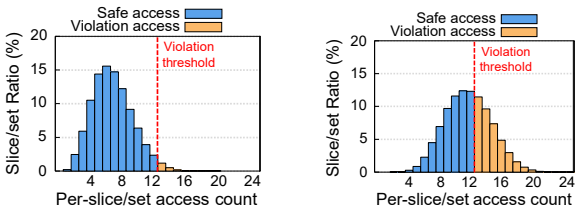
→Yet keeping the working set within the LLC capacity isn't enough. To clarify the impact of working set size, we increase the ring buffer to 2048 (32MB) in the single-flow experiment, making the total working set ~38MB, close to but below the LLC capacity (Figure 4(b)). Despite this, the LLC miss rate increases to 16.5% as shown in Figure 4(c) (while throughput drops by 11%). This indicates that simply keeping the page working set under the LLC capacity is insufficient to suppress LLC misses.

(2) Multi-flow case: High miss rates due to small DDIO capacity (DDIO write-miss regime). With two or more flows, the LLC cannot hold the entire page working set, so even CPU-reloaded pages are likely evicted before DDIO can reuse them on later writes. This shifts DMA traffic into a DDIO write-miss regime, where new lines are allocated in the DDIO-reserved ways. However, because the aggregate incoming traffic (i.e., packet-occupied memory in Figure 3(c)) far exceeds



(a) Limiting LLC ways. (b) Working set (MB). (c) LLC miss rate (%).

Figure 4: **DDIO write behavior.** (a) Limiting the number of available LLC ways increases DDIO write misses even in the single-flow case, indicating that DDIO write hits drive low LLC miss rates. (b–c) The write-hit regime can collapse even with working sets near the LLC capacity, raising LLC miss rates. See §3.2 for more details.



(a) Working set size: 22MB. (b) Working set size: 38MB.

Figure 5: **Violation ratio comparison (22MB vs. 38MB).** For given page footprints: (a) when per-slice/set access counts stay at or below the violation threshold (i.e., ≤associativity), accesses are safe (blue bars), so evictions are unlikely; (b) counts above this threshold are violations (orange bars), indicating potential conflict misses and LLC underutilization. See §3.3 for more details.

the DDIO capacity in multi-flow cases, those new lines are rapidly displaced by subsequent DDIO writes, which in turn increases the LLC miss rate during the CPU reads.

→**Yet throttling traffic below the DDIO capacity isn't enough.** Prior work suggests limiting in-flight bytes to fit within the DDIO capacity [58], so that CPU reads ideally hit in the LLC. Surprisingly, we find this is not sufficient by itself to keep LLC miss rates low. In Figure 3(c), we throttle the incoming rates of individual TCP connections (labeled "Throttling") by adjusting the TCP receive buffer size, ensuring that total packet-occupied memory remains within the DDIO capacity even with multiple flows. However, Figure 3(b) reveals that Throttling does not suppress LLC misses; instead, the miss rate rises to as high as 46.4%.

Despite similarly high LLC miss rates at 4–6 flows (Figure 3(b)), Throttling still offers a slight improvement compared to Default (Figure 3(a)); our measurements indicate that, beyond three flows, its L2 miss rate decreases while Default's remains unchanged, which is evidence of the improvement. This behavior is likely due to increased L2 hits on frequently recycled objects (e.g., `skb`) as the per-core throttled traffic (i.e., packet-occupied memory) approaches the L2 capacity. We also note that the average packet-occupied memory under Throttling decreases as the number of flows increases in Figure 3(c). This is because, given the same total incoming traffic, increasing the number of cores accelerates packet processing, which in turn reduces the average amount of packet-occupied memory before subsequent packets arrive.

3.3 Motivation: Conflict Misses Matter

To dive deeper into DDIO miss behavior, we analyze how the page working set is distributed across the LLC. Specifically, we compare the 1-flow and 2-flow cases under Throttling in Figure 3, configured so that both the Rx ring buffer and in-flight bytes fit within the LLC capacity (22MB vs. 38MB), yet the LLC miss rates differ (0.9% vs. 18%). Our methodology is as follows: First, by leveraging well-known reverse-engineering techniques [47], we identify the LLC details of our processor, including its slice hash function (§2). This allows us to compute the slice and set for any physical memory address. Second, we collect page-address traces while rerunning the two cases and examine the slice/set distribution over a working-set window. To quantify cache contention, we define a *violation* when the number of mappings to a particular slice/set exceeds the LLC associativity (i.e., 12 ways); the *violation ratio* is the fraction of such violations over all slice/set combinations.

Skewed page addresses lead to conflict misses and LLC underutilization. Figure 5(a) shows that, in the 22MB working-set case, mappings to any slice/set mostly stay below the associativity limit, yielding a violation ratio (orange bars) of only 1.97%. In contrast, the 38MB working-set case in Figure 5(b) jumps to 39.3% even though the working set still fits within the LLC. This indicates that parts of the LLC are underutilized because page addresses are skewed across slice/set mappings, which can lead to conflict misses. We note that a high violation ratio does not necessarily result in a high LLC miss rate; since tracing precise CPU access timing is prohibitively expensive, our analysis in Figure 5(b) does not reveal whether evictions occur before or after CPU reads. Nonetheless, a high violation ratio is necessary for high miss rates, whereas near-zero violation implies conflict misses are unlikely. In our measurements, we observe a clear correlation between violations and actual LLC misses.

Avoiding conflict misses is key. Revisiting the leaky DMA problem, we draw two key insights. First, expanding the DDIO allocation beyond two LLC ways [24, 64] is insufficient: in the DDIO write-hit regime, DDIO already exploits LLC ways outside the reserved portion. Although this provides DDIO with access to more LLC space, *conflict misses* still prevent full utilization of that space (Figure 5(b)). Second, throttling traffic to stay within the DDIO capacity [58] also falls short: our analysis reveals that capacity pressure is driven more by the page working-set size than by the amount of incoming traffic (Figure 3), while *conflict misses* further reduce the effective LLC capacity available to that working set (Figure 5(b)). These findings motivate a new design that actively reduces conflict misses to make DDIO effective, particularly for the applications that benefit from DDIO.

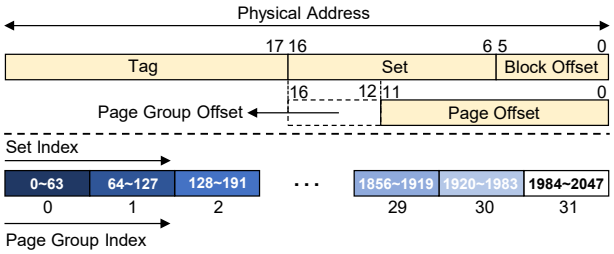


Figure 6: **Page group allocation.** The set index and block offset fields span 17 bits in total. Of these, 12 bits are used as the 4KB page offset. Thus, five upper set index bits remain available for page grouping, which we refer to as the page group offset. Accordingly, the 2048 LLC sets in the system can be partitioned into 32 page groups, each consisting of 64 sets. See §4.1 for more details.

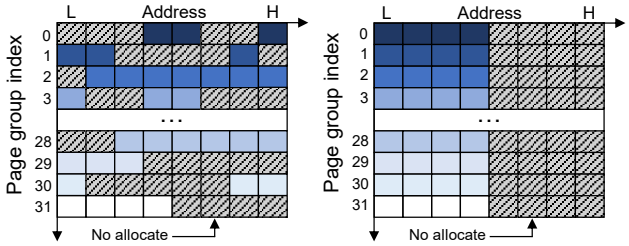


Figure 7: **Memory allocation behavior.** (a) The default Linux buddy allocator ignores set indexing, resulting in uneven page distribution across page groups and set-level imbalance. (b) Stride-1 enforces set balance by allocating an equal number of pages to each of the 32 page groups. See §4.1 for more details.

4 Sepia: DDIO Meets Page Coloring

To tackle conflict misses, we bring a classical yet effective technique—"page coloring"—into the DDIO context, adapting it to the packet I/O path. In this section, we introduce Sepia, a color-aware page allocator for DDIO. We first outline our page-coloring methodologies (§4.1). We then evaluate how coloring improves the effective capacity of modern sliced LLCs in both DDIO write-hit and write-miss regimes (§4.2). Finally, we present our Sepia design that makes page coloring practical in real DDIO-enabled systems (§4.3).

4.1 Page Coloring Methodology

As modern processors employ a sliced LLC architecture (recall §2), we must design our coloring mechanism by considering its structural characteristics. Figure 6 shows the bit-level structure of a physical address. Under page-granularity allocation, the lower set index bits overlap with the page offset, so we can use only the upper five bits (e.g., bits 12–16 in Figure 6) for page coloring. We use these specific bits to partition memory into 32 *page groups*, where each group consists of pages sharing the same upper set index (i.e., page group offset). Accordingly, we employ this page group as the basic unit of our page coloring technique.

Baseline: Stride-1. We first observe that the default Linux page allocator (i.e., buddy allocator) operates in both set- and slice-agnostic ways, leading to significant conflict misses in the LLC. As illustrated in Figure 7(a), the allocator does not consider the set-index mapping, resulting in imbalanced page-group usage: it allocates more pages on some page groups, increasing pressure on the corresponding sets and causing frequent conflict misses. Likewise, the allocator ignores the slice mapping, leading to imbalanced slice usage as well. Thus, the default allocator suffers from both set- and slice-level imbalance, directly degrading effective LLC capacity.

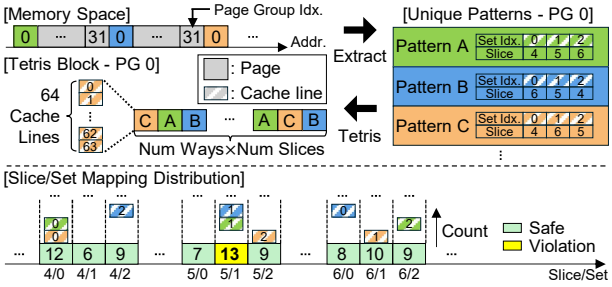


Figure 8: **Construction of a Tetris Block.** First, unique slice/set mapping patterns are extracted from the physical memory space for each page group (e.g., PG 0). Then, these patterns are assembled (i.e., *Tetris*) into a *Tetris block* to maximize the balance of the slice/set mapping. **However, due to the finite number of unique patterns, violations inevitably occur in the mapping distribution.**

To address this issue, we introduce Stride-1, a practical page-coloring technique that improves effective LLC capacity by balancing both set and slice usage (Figure 7(b)). Stride-1 first achieves set balance by allocating the same number of pages to each of the 32 page groups. Within each page group, it then allocates physical pages sequentially in ascending address order, leveraging an inherent behavior of modern LLC slice-hash functions (i.e., contiguous physical addresses tend to be distributed across slices), which is shown in prior work [46] and also confirmed by our own reverse engineering. By combining set-balanced allocation with this intrinsic slice-spreading behavior, Stride-1 successfully eliminates set conflicts and substantially reduces slice conflicts.

We observe that, while Stride-1 substantially reduces slice conflicts, it cannot fully eliminate slice-level imbalance due to an inherent hardware constraint. Modern processors often employ a non–power-of-two number of LLC slices (e.g., 26 in our system), which introduces modulo bias in the slice-hash function [47]. As a result, slice mappings are not uniformly distributed, creating hotspots on specific slices even when pages are allocated contiguously. This slice imbalance represents a fundamental limitation that prevents Stride-1 from achieving perfectly balanced slice usage.

Tetris: Slice-balanced Stride-1. To quantify how much performance potential remains beyond Stride-1, we further introduce *Tetris*, a theoretically optimal page-allocation sequence designed to minimize slice conflicts. Unlike Stride-1's sequential allocation, Tetris leverages reverse-engineered slice/set mappings to construct an allocation order that achieves near-perfect slice balance. As illustrated in Figure 8, Tetris first

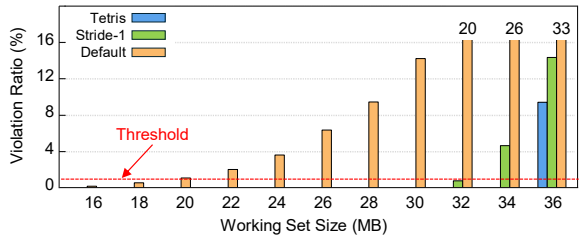
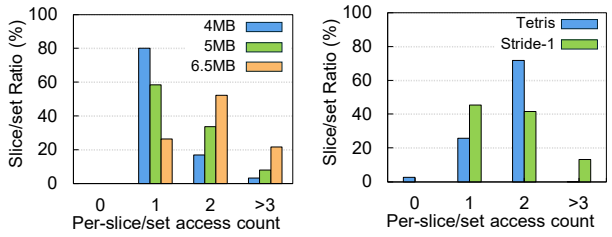


Figure 9: **Evaluation of effective LLC capacity.** Comparison of *violation ratio* across working set sizes. We define the effective LLC size as the working set size at which the violation ratio remains negligible ($< 1\%$) (Default: 18MB, Stride-1: 32MB, Tetris: 35MB (0.04%, not shown)). See §4.2 for more details.



(a) Violation analysis of Stride-1. (b) Tetris vs Stride-1 (5.5MB).

Figure 10: **Comparison of slice/set distributions.** (a) The effective DDIO capacity of Stride-1 is limited to 3MB (not shown), where the violation ratio remains 0%. (b) Tetris extends the effective DDIO capacity up to 5.5MB while maintaining a 0% violation ratio through optimized slice/set mapping. See §4.2 for more details.

extracts all unique patterns (i.e., distinct per-page slice/set mappings). For each page group, it then assembles these patterns into a *Tetris block*, a carefully arranged sequence that balances slice usage within that group. We use this Tetris block as the allocation unit, allowing us to achieve an optimally balanced slice distribution under page-granularity constraints. Tetris therefore serves as an upper bound, revealing how much headroom remains if Stride-1 were to run under a perfectly slice-balanced distribution in a system with a non-power-of-two number of LLC slices. In the following section, we rigorously evaluate the effective LLC capacity of three allocation mechanisms: Default (slice/set-imbalanced), Stride-1 (set-balanced and partially slice-balanced), and Tetris (set-balanced and fully slice-balanced).

4.2 Validation of Coloring Effectiveness

This subsection quantifies the gain in effective LLC capacity achieved by coloring compared to Default.

Experimental setup. We evaluate the three schemes across a range of allocation sizes to examine their sensitivity under different working set sizes. We use the *violation ratio* (defined in §3.3) as our metric, as we observe that it strongly correlates with cache miss rates. To control page allocation patterns, we use a large physical memory chunk for all three schemes. To quantify their effectiveness, we employ the *effective LLC size*, defined as the maximum working set size where the violation ratio remains negligible ($< 1\%$).

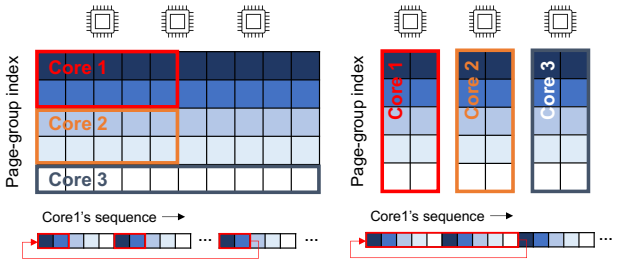
Analysis of effective LLC size (Write-hit regime). Figure 9 illustrates how the violation ratio changes as the working set size increases. First, we observe that Default fails to utilize the cache efficiently, limiting its effective LLC size to only 18MB (46.2% of capacity). In contrast, Stride-1 significantly shifts the point at which the violation threshold (1%) is reached, extending the effective LLC size to 32MB. This represents a 77.8% expansion in effective capacity compared to the baseline, allowing Stride-1 to utilize 82.1% of the total LLC. Tetris further pushes this limit to 35MB (89.7% of capacity). However, although Tetris represents the theoretical optimum, it does not reach 100%. As shown in Figure 8, this gap arises from the finite number of unique mapping patterns; since memory allocation operates at page granularity, the reuse of patterns inevitably causes set collisions on specific slices, thereby bounding the maximum achievable capacity.

We note that Tetris's violation ratio spikes from 0.04% to 9.43% beyond 35MB, exposing a critical dependency on a perfectly predefined allocation order. Given that this order is inevitably disrupted in real systems (detailed in §4.3), Tetris is practically infeasible.

Analysis of effective DDIO capacity (Write-miss regime). To verify whether the benefits of coloring observed in the full LLC carry over to the restricted DDIO region, we repeat the analysis with 2-way DDIO associativity. Figure 10 shows that coloring effectiveness is notably reduced compared to the full LLC results. While Stride-1 and Tetris utilized 82.1% and 89.7% of the total LLC, their efficiency in the 6.5MB DDIO region drops significantly to 46.2% (3MB) and 84.6% (5.5MB), respectively. This degradation highlights the impact of non-uniform mapping (i.e., modulo bias detailed in §4.1); the restricted 2-way associativity provides insufficient buffer to tolerate skewed load distributions among slices. Therefore, DDIO performance can be limited if we rely on coloring exclusively within such a restricted LLC region.

4.3 Sepia Design

We now present Sepia, which realizes the Stride-1 coloring over DDIO-enabled CPU cores; as discussed in §1, applying page coloring to DDIO scenarios creates new opportunities for CPU-efficient packet processing. However, two design challenges remain: inter-core and intra-core interference. First, since CPU cores share the LLC, colors must be balanced across cores to maximize effective LLC capacity while minimizing coordination overhead among per-core page pools. Second, although network processing behavior is deterministic, page recycling times can still vary within the same page pool, paced by application read calls. Thus, the allocator must choose the best color from the (recycled) pages currently available in the pool when replenishing the Rx ring. Once pages are submitted to the NIC, the corresponding ring entries are fixed and cannot be recolored until those entries are



(a) Shared pool. (b) Per-core pool.

Figure 11: **Sepia Manager** constructs per-core page pools, which is the right design that reduces working-set reconstruction overhead. See §4.3 for more details.

replenished. From these insights, our design goal is simple: *keep a per-core, color-aware page pool robust to cross-core conflicts and uneven page recycling speeds.* To this end, we introduce two design components: (1) Sepia Manager, which builds per-core colored page pools and tracks page availability; and (2) Sepia Allocator, which selects the best available page at allocation time.

Sepia Manager. The Sepia Manager organizes the Rx-ring working set as a two-dimensional page array (see Figure 7(b)), which enables the Stride-1 allocation strategy. This page array is modeled after the structure of the LLC: vertically, there are N page-groups, and horizontally, M pages per group; pages within the same group share the same color. The value of N is determined by the LLC specification (§4.1); M is then set by the total number of pages required by each Rx ring. For example, if we provision 16MB per ring with 32 page groups, then $M = 128$ pages per group ($32 \times 128 \times 4\text{KB} = 16\text{MB}$).

For scenarios where multiple cores contend for the LLC via DDIO writes, we consider two options for constructing the colored page pool: (1) assign a subset of page groups to each core while sharing a single pool (Figure 11(a)); or (2) provision a separate per-core pool that permits access to all page groups (Figure 11(b)). The first option reduces cross-core conflicts but raises management costs: we must track which cores are active to assign page groups preferentially to busy cores for better LLC utilization. Moreover, since pages already submitted to the NIC cannot be recolored, any working-set rebuild triggered by changes in active-core counts must wait for a full recycle turn, consuming the previously colored pages in the meantime. Finally, color distribution can become unfair depending on the number of active cores and the value of N. For example, in Figure 11(a), Core 3 is assigned a single page group while others receive two. Sepia avoids this overhead by building per-core colored page pools (Figure 11(b)), allowing each core to operate independently without tracking other cores' status. While the per-core pool design accepts some cross-core interference as a trade-off, each core can draw from the full set of page groups, which helps keep per-group page counts balanced across cores on average.

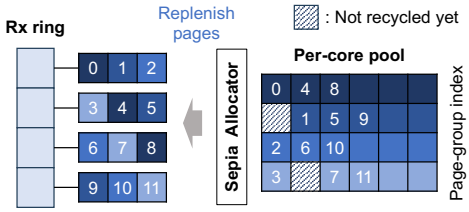


Figure 12: **Sepia Allocator** selects pages one by one from each page-group index, choosing the earliest page within the group. The numbers inside the pool illustrate an example allocation order. When previously unavailable pages are recycled, they become eligible for selection in subsequent rounds. See §4.3 for more details.

Sepia Allocator. The Sepia Allocator is responsible for providing pages from the per-core page pools. Ideally, allocation cycles through page groups sequentially in a wraparound manner—i.e., the Stride-1 pattern over a contiguous memory chunk, as illustrated in Figure 11(b) (see "Core 1's sequence"). This ideal ordering is attainable at initialization; however, as pages are recycled, the order in which they return for Rx-ring replenishment may diverge from the original allocation, creating gaps—or "page holes"—in the pool, as shown in Figure 12. Given that the recycle timing of individual pages is outside our control, Sepia prioritizes preserving sequential color ordering during Rx-ring replenishment. If the Sepia Allocator encounters a page hole during sequential allocation, it selects the next available page within the same group (and thus the same color). This ensures that Rx descriptor pages are filled in page-group order and preserves the low-conflict allocation pattern. A simplified example of this scenario is illustrated in Figure 12. Additionally, the allocator prefers the earliest available page per group, which helps minimize the active working set at light load. Lastly, if a particular page group reaches the right boundary (M) earlier than others due to page holes, the Sepia Allocator skips that group until pages become available. Once the pool is exhausted, it falls back to the standard Linux allocator. Although this may temporarily introduce non-colored pages into the pool, Sepia continues to manage and prioritize colored pages. As the load decreases and colored pages are recycled, Sepia can naturally restore the effectiveness of page coloring (§6.3).

5 Sepia Implementation

We implement the Sepia prototype in the Linux kernel 6.6. Although our current implementation is integrated with a specific network device driver (ConnectX), our goal is to provide a generic API for network drivers, thereby minimizing the effort required to port Sepia to other network drivers. This portability is enabled by the fact that Sepia's implementation is decoupled from any driver-specific logic.

Page working set management. Sepia requires a carefully constructed working set of well-aligned, consecutive pages to minimize cache conflicts. Since the default Linux memory allocator does not provide this level of control, we leverage the Contiguous Memory Allocator (CMA) [18] to obtain large,

physically contiguous memory regions suitable for coloring-aware allocation. Specifically, we pre-allocate 16MB of pages per core (4MB for the Rx ring and 12MB for descriptor replenishment). Given that our testbed has 18 DDIO-enabled cores, this results in a total allocation of 288MB of contiguous memory, comparable to the default Linux footprint for Rx rings (excluding packet-occupied memory). When additional pages are further needed, Sepia falls back to the standard Linux allocator as discussed in §4.3.

Sepia interfaces. To facilitate driver integration, we introduce Sepia interfaces, for example, `sepia_init()` to initialize per-core colored page pools. The function `sepia_alloc()` can be invoked by network drivers when new pages are needed for Rx descriptor replenishment. Initially, the network drivers populate Rx descriptors using the default Linux allocator. However, once Sepia detects that a core is actively consuming Rx descriptors and a colored page pool has been created, Sepia seamlessly switches the replenishment source to the Sepia Allocator. This implementation design allows Sepia to operate transparently alongside existing kernel infrastructure while optimizing memory placement and minimizing cache conflicts during high-throughput packet processing. Note that Sepia recognizes colored pages by verifying that their addresses lie within the CMA-allocated range, which ensures they are correctly returned to Sepia's pools during recycling.

Handling capacity misses in Sepia. While Sepia focuses on reducing conflict misses, CPU efficiency is typically maximized when both capacity and conflict misses are effectively avoided. In the write-hit regime, Sepia expands the effective working set by 77.8% over Linux (Figure 9); thus, if the aggregate working set is kept beneath this effective LLC budget, both types of misses can be avoided. Prior proposals shrink the working set by cross-core Rx-ring sharing [51, 52], but they operate in user space [2] or on the device. We instead tune the per-core Rx ring (we plan to integrate the prior approach [51] into Sepia as future work). Considering that, in our default setup, the NIC's DIM [3] is configured to coalesce up to 128 packets (∼1.1MB at 9000B MTU) before triggering softIRQ and ring sizes are powers of two, we set Sepia's default to 4MB per core (ring size 256) as our baseline configuration to constrain the working set. In our evaluation, this setup successfully replenishes the consumed ring entries before the next DMA arrivals; for comparison, we also evaluate the default ring in §6.2. In addition, reducing TCP Rx buffer further helps shrink the working set; for example, combining 4MB per-core rings with 4MB TCP Rx buffer (`tcp_rmem`) per core keeps Sepia's working set within the effective LLC capacity (32MB; Figure 9) up to 4 cores (∼8MB per core), saturating 200Gbps with ∼4 cores (§6.1). We use this configuration for Sepia in our microbenchmarks to evaluate the best-case regime.

In the write-miss regime (aggregate working set > LLC capacity), the key is to keep total in-flight bytes below the effective DDIO capacity. On our platform, two DDIO ways yield only ∼3MB effective capacity (Figure 10). Although system-wide TCP tuning (as in §3) could enforce this, throttling every connection under such a tight budget is impractical: per-flow traffic monitoring and rate regulation typically incur high overhead as flow loads vary over time [30][3]. Accordingly, we use the write-hit regime to represent the best case (we also evaluate excessive working-set scenarios).

6 Evaluation

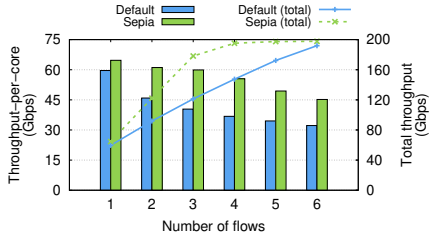
This section evaluates Sepia's effectiveness across diverse experimental setups and real-world applications. We first perform microbenchmarks, comparing it with default Linux (§6.1). Next, we provide a performance breakdown of Sepia in terms of throughput-per-core to understand the performance gains in the best-case regime (§6.2). Finally, we evaluate Sepia with real-world applications, including SPDK, Nginx, and Memcached (§6.4). We describe each setting in place. Unless stated otherwise, we use the same evaluation setup described in §3.1.

6.1 Microbenchmarks

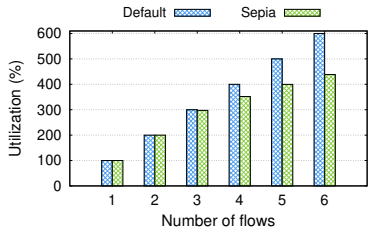
We use iperf [21] for our microbenchmarks, varying the number of flows with one flow per core as we did in §3. Specifically, we use two configurations of Sepia discussed in §5 to evaluate it under (1) the write-hit regime, where capacity and conflict misses are avoided (best case), and (2) the write-miss regime, where capacity misses occur (worst case).

Write-hit regime. Our goal is to understand how increasing the effective LLC capacity translates into improved CPU efficiency. In Figure 13(a), we first observe that Sepia improves throughput-per-core by ∼1.51× compared to Default, saturating the 200Gbps link with 3.5 cores, whereas Linux requires 6 cores to reach link saturation (Figure 13(b)). This is because DDIO's benefits are maximized with near-zero LLC miss rates (∼0.4%) up to 6 flows (cores) (Figure 13(c)). To further investigate how Sepia achieves this, we plot its working set size corresponding to Figure 13, as shown in Figure 14. As discussed earlier, each working set consists of per-core Rx descriptor pages and packet-occupied memory (i.e., in-flight bytes controlled by TCP). Consequently, the Rx descriptor portion of the working set increases linearly with the number of cores (i.e., 4MB per core), as shown in Figure 14. On top of this, the packet-occupied memory increases to ∼7.8MB up to 3 cores, and then decreases as the link becomes saturated (we discussed this phenomenon in §3.2). Since Sepia increases the effective LLC capacity from 18MB to 32MB via the color-aware allocations (Figure 14), it can avoid both capacity and conflict misses up to 6 cores (and also 7 cores;

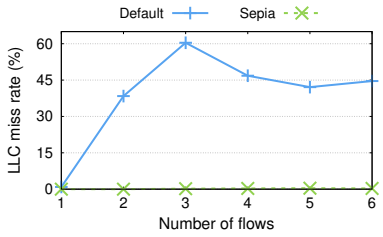
[3] One alternative is to increase the DDIO way budget, making DDIO write-missed pages more likely to remain in the LLC until CPU reads [64], so coarse TCP controls suffice. We thoroughly followed existing open methodologies [10, 16, 23, 34] to adjust the DDIO-allocated portion, which worked on earlier generations of Intel CPUs, but none succeeded on our recent Intel Ice Lake and Emerald Rapids machines. In any case, Sepia remains effective because its design is independent of the DDIO-associativity configuration.



(a) Throughput (Gbps).



(b) CPU utilization (%).



(c) LLC miss rate (%).

Figure 13: **DDIO write-hit regime: Sepia's benefits are maximized as both capacity and conflict misses are avoided.** (a) Sepia achieves up to 1.51× improvement in terms of throughput-per-core, compared to Default, reaching the link saturation (200Gbps) using (b) 2.5 fewer cores than Default, (c) while keeping near-zero LLC miss rates (∼0.4%). See §6.1 for more details.

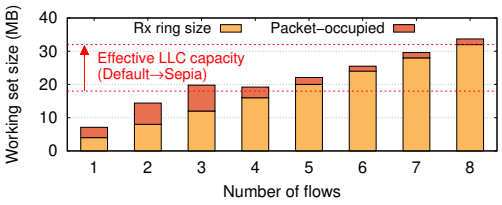
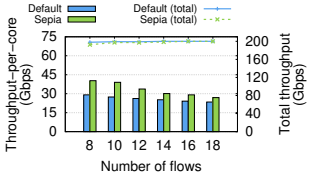
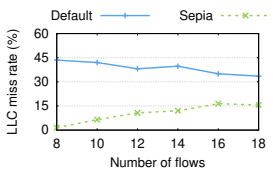


Figure 14: **Sepia's working set trend (write-hit regime).** Our color-aware allocation expands the effective LLC capacity from 18MB to 32MB, accommodating the page working sets of up to 7 cores.



(a) Throughput (Gbps).



(b) LLC miss rate (%).

Figure 15: **DDIO write-miss regime: Sepia's benefits shrink as capacity misses become unavoidable.** (a) Even in this worst-case scenario, Sepia still improves throughput-per-core, and (b) reduces LLC miss rates compared to Default. See §6.1 for more details.

results not shown). These results also confirm that our analysis in §4.2 matches real-world experiments. Note that if our LLC were fully slice-balanced (so that the effective capacity would become 35MB in our system), LLC misses would be suppressed up to 8 cores.

Write-miss regime. In Figure 15, we further increase the number of flows from 8 to 18 using all available DDIO-enabled cores. Sepia's working set starts to exceed the effective LLC capacity at 8 flows (Figure 14), thus shifting to the write-miss regime and showing a 1.5% LLC miss rate (> 1%). As the number of flows grows to 18, Sepia's LLC miss rates increase up to 16.4% (Figure 15(b)). We find that after 8 flows, Sepia's packet-occupied memory fluctuates between 2.6–3.7MB; given that the effective DDIO capacity for Sepia is around 3MB (Figure 10), this can lead to high LLC miss rates. Nonetheless, Sepia still shows lower LLC miss rates than Default, achieving better CPU efficiency overall (Figure 15(a)) even in the worst-case scenarios, where capacity misses are unavoidable.

6.2 Understanding Performance Gains

To understand how Sepia improves CPU efficiency and to quantify the contribution of each technique, we perform an ablation study by enabling our mechanisms one at a time under the best-case scenario shown in Figure 13. Starting from the baseline Linux allocator (Default), we first replace Linux's page allocator with our Sepia Allocator (Stride-1). Next, we reduce the default Linux Rx ring size by *throttling* the number of ring entries to 256 while retaining the baseline Linux allocator (Ring Throttling). This setup evaluates the effect of working set reduction, following a similar approach to SHRing [51]. Lastly, We evaluate Sepia, which incorporates both Stride-1 and our working-set throttling techniques (i.e., limiting the number of ring entries and the TCP buffer size). Figure 16 shows the performance breakdown across different configurations, where the x-axis denotes the number of concurrent flows and the y-axis reports throughput-per-core.

Stride-1. Overall, Stride-1 improves per-core throughput by up to 11.4% (8.62% on average) by increasing the effective LLC capacity through reduced conflict misses compared to the default Linux memory allocator. With a single flow, the total working set size (21.2MB on average), including Rx ring and TCP buffers, exceeds the baseline's effective LLC capacity (18MB) but fits within that of Stride-1 (32MB). Here, Stride-1 reduces unnecessary LLC conflicts and lowers the LLC miss rate (from 0.9% to 0.1%), yielding a modest but measurable throughput improvement. As the number of flows increases, the working set exceeds the effective capacity of both Default and Stride-1, making both cases operate in the DDIO write miss scenario. In this case, efficient use of the DDIO-reserved ways becomes critical. Note the leaky DMA problem becomes even more severe with fewer available ways, making it difficult for the baseline to utilize the DDIO-reserved ways efficiently. On the other hand, Stride-1 can mitigate this problem, so DDIO write misses are more uniformly distributed across LLC slices and sets, improving overall efficiency even under high flow counts.

Ring Throttling. The results show that throttling the Rx ring also provides meaningful performance benefits. With a single flow, its performance is comparable to Default because the Default's working set size is already small and fits within

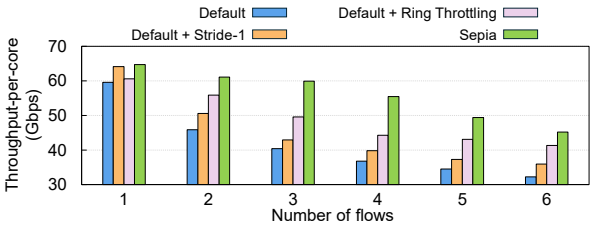


Figure 16: **Sepia performance breakdown.** This result indicates that Sepia's color-aware allocation (together with Ring Throttling) allows the network stack to utilize the LLC more efficiently during DDIO. See §6.2 for more details.

Table 1: **Memory bandwidth consumption** for the 4-flow case in Figure 16. This result indicates that Sepia is more cache-friendly, requiring less memory bandwidth than the other configurations.

Mechanism	Default	Default w/Stride-1	Default w/Ring-Thrtt.	Sepia
BW (GB/s)	31.29	30.01	16.59	0.96

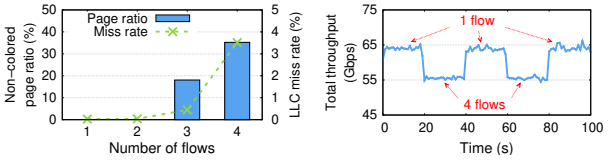
the effective LLC capacity. However, as the number of flows increases, Rx ring throttling provides substantial performance improvements over both Default and Stride-1. This is because its reduced working set improves the LLC efficiency, resulting in lower LLC miss rate. To quantify this effect, we additionally measure the memory bandwidth consumption for each configuration at 4 flows (Table 1). The results show that Ring Throttling reduces memory bandwidth usage by 47.0% and 44.7% compared to Default and Stride-1, respectively.

Sepia. On top of Stride-1, Sepia adopts working-set throttling techniques: limiting both the number of active ring entries and the TCP buffer size. For lower flow counts (1–4 flows), the working set can fit within the effective LLC capacity because Sepia reduces the working set size compared to the baseline. This allows additional flows to continue operating in the DDIO write hit scenario. Here, the combination of Stride-1 (larger effective capacity) and memory throttling (smaller working set) produces a clear synergistic effect, yielding substantial throughput improvements by up to 50.8%. For higher flow counts (5–6 flows), the working set again exceeds the LLC effective capacity, placing all cases in the DDIO write miss scenario. Here, Sepia shows consistent performance improvement over Default and Stride-1. This is because Sepia can improve L2 cache efficiency by constraining the number of active descriptor pages per core. This reduces the overall load-miss stall cycles, leading to higher throughput.

In summary, Sepia delivers significant performance improvement across all flow counts by reducing the working set size, amplifying the benefits of color-aware page allocation, and improving L2 efficiency.

6.3 Performance under Bursty Workloads

We now evaluate Sepia under an extreme scenario where its page pool is exhausted by bursty DMA requests, causing the default Linux allocator to introduce *non-colored* pages into the pool. To generate this bursty workload, we increase



(a) Page ratio and miss rate (%). (b) Total throughput (Gbps).

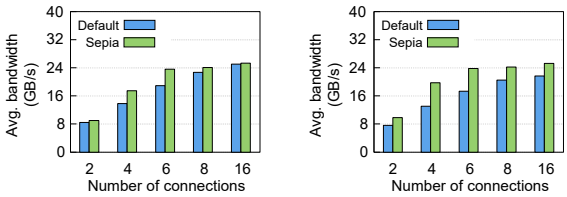
Figure 17: **Sepia performance under bursty workloads.** (a) Page-pool exhaustion can introduce non-colored pages from the default allocator, increasing LLC miss rates. (b) Sepia quickly recovers once enough colored pages become available. See §6.3 for more details.

the number of flows on a single core, each using a 6MB TCP Rx buffer, to consume Sepia's per-core page pool more aggressively. Figure 17(a) shows the non-colored page ratio (i.e., the fraction of non-colored pages in the pool), along with the corresponding average LLC miss rate for different numbers of flows. With 4 flows, non-colored pages account for 35% of the pool, increasing the LLC miss rate to 3.5% and reducing total throughput from 64Gbps to 55Gbps compared to the 1-flow case. Based on these static measurements, we then alternate between 1 and 4 flows every 20 seconds, as shown in Figure 17(b). The results demonstrate that Sepia effectively recovers from colored-page depletion as soon as enough colored pages become available.

6.4 Real-world Applications

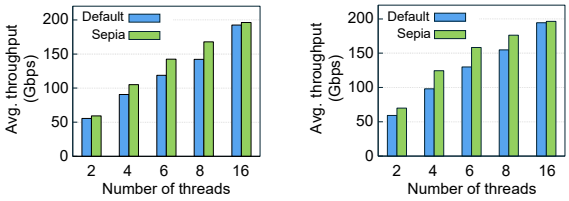
SPDK. We evaluate Sepia using SPDK, a high-performance userspace storage stack. Specifically, we use NVMe-over-TCP, which connects host and target machines through the Linux network stack [28, 29, 62]. The host NVMe initiator generates SPDK read requests with two block sizes (64KB, 128KB) and a queue depth of 16. For the NVMe remote target, we deploy 36 SPDK polling threads pinned to 36 cores, so that the initiator with Sepia becomes the bottleneck. We use a NULL block device [8] for a backend storage medium to eliminate the storage bottlenecks. We measure the average I/O bandwidth across different numbers of NVMe-over-TCP connections (five runs for each configuration), as shown in Figure 18. The results show that Sepia improves I/O bandwidth by up to 26.7% for 64KB blocks and 51.1% for 128KB blocks. Notably, Sepia shows huge performance improvement over the default Linux kernel when the working set is kept within the effective LLC capacity. The benefit becomes much larger with 128KB blocks, where each request spans more packets and enlarges the working set size. In this case, the default Linux quickly encounters conflict misses and LLC capacity pressure (due to higher portion of packet occupied), leading to significant performance degradation. On the other hand, Sepia achieves stable link-saturating throughput without any performance degradation at higher connection counts (>6). These results demonstrate that Sepia substantially improves real-world I/O performance.

Nginx. We evaluate Sepia using Nginx [7], a widely used web server, running all worker processes over DDIO-enabled cores.



(a) Block size: 64KB. (b) Block size: 128KB.

Figure 18: **Evaluation results with SPDK.** Sepia achieves up to 1.27× and 1.51× improvements in average bandwidth over the default Linux with 64KB and 128KB block sizes, respectively.

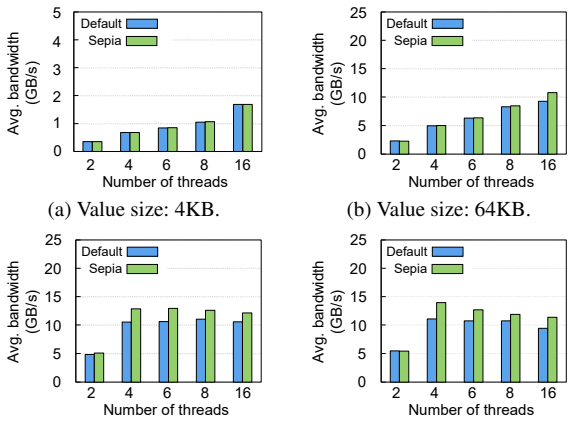


(a) Web page size: 2MB. (b) Web page size: 4MB.

Figure 19: **Evaluation results with Nginx.** Sepia achieves up to 1.2× and 1.27× improvements in average throughput over default Linux with 2MB and 4MB web page sizes, respectively.

For input traffic, we use the wrk benchmark [27] to generate HTTP POST requests, so that the Nginx server with Sepia becomes the bottleneck on the receiver side. We configure one TCP connection per wrk thread and increase the number of threads to scale the request concurrency. We compute upload bandwidth (requests/sec × payload size) and report the average upload bandwidth over five runs for each configuration (Figure 19). Following prior work that reports the average size of modern webpages to be around 2MB [32], we evaluate two payload (web page) sizes: 2MB (Figure 19(a)) and 4MB (Figure 19(b)). With 2MB payload, Sepia improves the upload bandwidth by up to 20% at six wrk threads. With 4MB payload, Sepia achieves up to 27.1% bandwidth improvement at four wrk threads. Larger payloads increase the number of packets per HTTP request and enlarge the descriptor working set, making the default Linux allocator more susceptible to LLC capacity pressure. In contrast, Sepia can keep their working set compact, allowing Nginx to sustain higher upload bandwidth over Default across all configurations.

Memcached. We evaluate Sepia using Memcached, one of the most widely used in-memory key-value applications [5]. We use four Memcached worker threads and pin them to four CPU cores to evaluate how efficiently each scheme can handle incoming requests under constrained resources. On the client side, we use Memtier benchmark [6] configured to issue 100% SET requests to ensure the Memcached server with Sepia becomes the bottleneck. Similar to Nginx evaluation, we create one TCP connection per client thread, and use a pipeline depth of four. We measure the average throughput by varying the number of client threads (Figure 20). Here, we use four different value sizes: 4KB, 64KB, 512KB, and 1MB. The 4KB value size represents a workload with limited



(a) Value size: 4KB. (b) Value size: 64KB.
(c) Value size: 512KB. (d) Value size: 1MB.

Figure 20: **Evaluation results with Memcached.** For workloads with limited DDIO benefit (e.g., 4KB values), Sepia matches the performance of default Linux. As the value size increases to 512KB and 1MB, where DDIO efficiency becomes more important, Sepia improves average bandwidth by up to 1.26× over default Linux.

DDIO benefits, because each request touches only a small amount of data. As shown in Figure 20(a), Sepia does not degrade performance compared to Default even for this workload, indicating that it can remain safely enabled when DDIO benefit is limited. Starting from 64KB, Sepia begins to show modest benefits (Figure 20(b)). For larger value sizes, where DDIO efficiency becomes more important, Sepia provides clearer improvements: up to 22% with 512KB values (Figure 20(c)) and up to 25.9% with 1MB values (Figure 20(d)). Note that Sepia sustains higher throughput once the server becomes core-saturated (more than four threads), indicating that Sepia processes requests more efficiently under resource-constrained environment. This throughput improvement is also aligned with latency results. With 512KB, Sepia reduces average latency from 0.74ms to 0.60ms. With 1MB, Sepia provides more latency reduction from 1.40ms to 1.12ms.

7 Discussion

This section further discusses how Sepia complements existing solutions and can be extended to future architectures.

Sepia's benefits with shared ring buffer schemes. Our work shows that conflict misses are another key contributor to high LLC miss rates in DDIO operations, making Sepia complementary to existing shared ring buffer schemes such as SHRing [51]. SHRing reduces cache capacity misses by sharing DMA buffers across flows, while Sepia reduces conflict misses by allocating colored pages. Two aspects of the cache footprint are particularly important. First, the TCP Rx buffer contributes to the total cache footprint in addition to Rx ring pages, so the total working set size can increase beyond the ring size, imposing higher cache capacity pressure. By increasing effective cache capacity, Sepia can also improve the effectiveness of SHRing. Second, under heavy incoming traffic, SHRing's single shared ring may need to allocate more

pages to accommodate concurrent DMA requests. In this case, SHRing can benefit from a larger effective ring size because Sepia reduces conflicts among DMA target pages.

Sepia's benefits with zero-copy receive. DDIO efficiency remains important even when zero-copy is enabled, because applications may still read or write network data after it has been received. Sepia can therefore benefit such applications by allowing them to access received data from the LLC instead of main memory. One possible extension is to expose an application-level allocation interface that returns pages from the Sepia Manager rather than from the standard memory allocation datapath. This would ensure that the NIC can directly DMA into page-colored application buffers.

Sepia's benefits with different and future architectures. While Stride-1 can utilize 82.1% of the total LLC capacity in our hardware configuration (§4.2), our additional simulations suggest that power-of-two slice counts (e.g., 16 slices [47]), would allow Stride-1 to utilize 100% of the LLC capacity by eliminating slice-hash imbalance. However, this architectural property alone is not sufficient: without color-aware allocations, cache utilization remains similar to Figure 5(b), even under power-of-two slice configurations. We also found that, for non-power-of-two slice counts, using only tag bits in the slice hash function can make Tetris more feasible, as all sets within the same page map to the same slice, making it easier to find alternative Tetris blocks for unrecycled pages. These observations provide useful insights for hardware vendors designing future architectures that better support cache-friendly network processing with Sepia, while applying Sepia to other CPU architectures, such as AMD, remains future work.

8 Related Work

Improving cache efficiency. Prior work has found that direct cache access (e.g., DDIO) becomes inefficient as network bandwidth increases across both user-space and kernel network stacks [14, 23, 58]. Intel Cache Allocation Technology (CAT) can isolate cache usage across applications; however, it cannot isolate I/O traffic, so I/O operations still contend for DDIO-reserved cache ways [58]. To reduce this contention, prior work either limits the buffer size to match the DDIO cache capacity [58] or ensures that the I/O working set fits within the LLC capacity [51, 52]. Nevertheless, we show that avoiding capacity misses alone is insufficient. NeBuLa reduces network queuing and improves RPC tail latency by pushing RPC payloads directly into the processor's L1 cache [57]. Finally, NanoPU [31] extends NeBuLa's scheme to transfer data directly from the NIC into CPU registers.

Page/cache coloring. Prior work has shown that page/cache coloring can improve cache performance by reducing cache contention and conflict misses [42, 59, 63, 66, 67], but these benefits have primarily been explored for application memory accesses. For example, Memstrata [67] treats Compute Express Link (CXL) memory as a second tier and local memory as a cache to extend memory capacity with Intel Flat Memory mode; to avoid inter-virtual machine (VM) interference in local memory, it uses page coloring to provide performance isolation across VMs. With DCA support, NIC-induced cache conflicts can also degrade performance, while Sepia mitigates them with page coloring to improve CPU efficiency (§6).

Improving CPU efficiency. Hardware-based solutions [17, 55] achieve zero-copy processing by allowing applications to provide their buffer addresses directly to the NIC, so the NIC can DMA data straight into application memory. Recently, Linux has added support for TCP zero-copy send and receive [19, 20], which may require hardware features such as header–data split. Cornflakes leverages the NIC's scatter-gather capability to aggregate small messages into a single packet, achieving zero-copy processing to reduce latency [54]. Beyond zero-copy, there are works aiming to reduce other packet processing overhead. For example, NetChannel reduces CPU load by recycling pages—after data copies—into Rx descriptor memory pools, thus avoiding expensive slow-path memory allocations [15]. Sepia is orthogonal to these approaches and can further boost CPU efficiency alongside them as discussed in §7.

Network stacks for scaling/scheduling resources. NetChannel [15] disaggregates network processing into multiple layers, enabling each layer to scale and schedule independently; this allows the Linux network stack to saturate terabit network links. Similarly, FlexToE [56] parallelizes the TCP processing pipeline into multiple processing units within smartNICs to scale the performance. And also, there has been a lot of recent work on userspace network stacks [11, 26, 36, 37, 38, 39, 43, 44, 45, 49, 50, 53, 65]. Integrating Sepia with these userspace stacks would be interesting future work.

9 Conclusion

In this paper, we revisit DDIO's leaky DMA problem and show that conflict misses are a missing piece in understanding high LLC miss rates in DDIO. By introducing page coloring into DDIO, we demonstrate that the effective LLC capacity can increase by up to 77.8–94.4% over Linux, substantially improving cache utilization. Building on this insight, we design Sepia, a color-aware page allocator that reduces LLC conflict misses during packet processing. Our Sepia prototype implemented in the Linux kernel saturates a 200Gbps link with 3.5 cores (2.5 fewer cores than Linux), while maintaining near-zero LLC miss rates in the best-case scenarios.

Acknowledgments

We would like to thank the anonymous OSDI reviewers for their insightful feedback. This work was supported by Institute of Information & communications Technology Planning & Evaluation (IITP) grant funded by the Korea government (MSIT) (RS-2024-00459026, RS-2024-00405128, RS-2024-00349594, RS-2024-00395134, RS-2025-02217106). The corresponding authors are Jaehyun Hwang and Joonsung Kim.

References

[1] net/mlx5e: Support RX multi-packet WQE (Striding RQ). https://patchwork.ozlabs.org/project/netdev/patch/1460928725-18741-6-git-send-email-saeedm@mellanox.com/, 2016.

[2] SHRing - DPDK. https://github.com/BorisPis/shRing-dpdk, 2025.

[3] Dynamically-Tuned Interrupt Moderation (DIM). https://enterprise-support.nvidia.com/s/article/dynamically-tuned-interrupt-moderation--dim-x, 2026.

[4] Ethtool. https://linux.die.net/man/8/ethtool, 2026.

[5] Memcached - a distributed memory object caching system. https://memcached.org, 2026.

[6] memtier_benchmark. https://github.com/redis/memtier_benchmark, 2026.

[7] nginx. https://nginx.org/en/, 2026.

[8] Null block device driver. https://docs.kernel.org/block/null_blk.html, 2026.

[9] Mohammad Alian, Yifan Yuan, Jie Zhang, Ren Wang, Myoungsoo Jung, and Nam Sung Kim. Data Direct I/O Characterization for Future I/O System Exploration. In *IEEE ISPASS*, 2020.

[10] Tom Barbette. DDIOTune Element Documentation. https://github.com/tbarbette/fastclick/wiki/DDIOTune, 2022.

[11] Adam Belay, George Prekas, Ana Klimovic, Samuel Grossman, Christos Kozyrakis, and Edouard Bugnion. IX: A Protected Dataplane Operating System for High Throughput and Low Latency. In *USENIX OSDI*, 2014.

[12] Jacob Bramley. Page Colouring on ARMv6 (and a bit on ARMv7). https://developer.arm.com/community/arm-community-blogs/b/architectures-and-processors-blog/posts/page-colouring-on-armv6-and-a-bit-on-armv7, 2013.

[13] Edouard Bugnion, Jennifer M. Anderson, Todd C. Mowry, Mendel Rosenblum, and Monica S. Lam. Compiler-Directed Page Coloring for Multiprocessors. In *ACM ASPLOS*, 1996.

[14] Qizhe Cai, Shubham Chaudhary, Midhul Vuppalapati, Jaehyun Hwang, and Rachit Agarwal. Understanding host network stack overheads. In *ACM SIGCOMM*, 2021.

[15] Qizhe Cai, Midhul Vuppalapati, Jaehyun Hwang, Christos Kozyrakis, and Rachit Agarwal. Towards μs tail latency and terabit ethernet: Disaggregating the host network stack. In *ACM SIGCOMM*, 2022.

[16] Intel Community. tuning 0xc8b register by wrmsr command. https://community.intel.com/t5/Intel-Xeon-Processor-and-Server/tuning-0xc8b-register-by-wrmsr-command/m-p/1641012, 2024.

[17] RDMA Consortium. Architectural specifications for RDMA over TCP/IP. http://www.rdmaconsortium.org/.

[18] Jonathan Corbet. A reworked contiguous memory allocator. https://lwn.net/Articles/447405/, 2011.

[19] Jonathan Corbet. Zero-copy networking. https://lwn.net/Articles/726917/, 2017.

[20] Jonathan Corbet. Zero-copy TCP receive. https://lwn.net/Articles/752188/, 2018.

[21] Jon Dugan, Seth Elliott, Bruce A. Mah, Jeff Poskanzer, and Kaustubh Prabhu. iPerf - The ultimate speed test tool for TCP, UDP and SCTP. https://iperf.fr/, 2026.

[22] Alireza Farshin, Tom Barbette, Amir Roozbeh, Gerald Q. Maguire Jr., and Dejan Kostić. PacketMill: Toward Per-Core 100-Gbps Networking. In *ACM ASPLOS*, 2021.

[23] Alireza Farshin and Adrien Mahieux. ddio-bench: Understanding Intel Data Direct I/O Technology. https://github.com/aliireza/ddio-bench, 2021.

[24] Alireza Farshin, Amir Roozbeh, Gerald Q. Maguire Jr., and Dejan Kostić. Reexamining Direct Cache Access to Optimize I/O Intensive Applications for Multi-hundred-gigabit Networks. In *USENIX ATC*, 2020.

[25] Alireza Farshin, Amir Roozbeh, Gerald Q. Maguire, and Dejan Kostić. Make the Most out of Last Level Cache in Intel Processors. In *ACM Eurosys*, 2019.

[26] Joshua Fried, Zhenyuan Ruan, Amy Ousterhout, and Adam Belay. Caladan: Mitigating Interference at Microsecond Timescales. In *USENIX OSDI*, 2020.

[27] Will Glozer. wrk HTTP benchmark. https://github.com/wg/wrk, 2021.

[28] Jaehyun Hwang, Qizhe Cai, Ao Tang, and Rachit Agarwal. TCP $\approx$ RDMA: CPU-efficient Remote Storage Access with i10. In *USENIX NSDI*, 2020.

[29] Jaehyun Hwang, Midhul Vuppalapati, Simon Peter, and Rachit Agarwal. Rearchitecting Linux Storage Stack for μs Latency and High Throughput. In *USENIX OSDI*, 2021.

[30] Jaehyun Hwang, Joon Yoo, and Nakjung Choi. Deadline and Incast Aware TCP for cloud data center networks. *Computer Networks*, 68:20–34, 2014.

[31] Stephen Ibanez, Alex Mallery, Serhat Arslan, Theo Jepsen, Muhammad Shahbaz, Changhoon Kim, and Nick McKeown. The nanoPU: A Nanosecond Network Stack for Datacenters. In *USENIX OSDI*, 2021.

[32] Jamie Indigo, Dave Smart, Chris Steele, Danielle Rohe, and Barry Pollard. *Page Weight*, chapter 21. HTTP Archive, 2022.

[33] Intel. Intel Data Direct I/O Technology (Intel DDIO): A Primer. https://www.intel.com/content/dam/www/public/us/en/documents/technology-briefs/data-direct-i-o-technology-brief.pdf, 2012.

[34] Intel. User space software for Intel(R) Resource Director Technology. https://github.com/intel/intel-cmt-cat, 2025.

[35] Intel. 3rd Gen Intel(R) Xeon(R) Scalable Processors. https://www.intel.com/content/www/us/en/ark/products/series/204098/3rd-gen-intel-xeon-scalable-processors.html, 2026.

[36] EunYoung Jeong, Shinae Woo, Muhammad Asim Jamshed, Haewon Jeong, Sunghwan Ihm, Dongsu Han, and KyoungSoo Park. mTCP: a highly scalable user-level TCP stack for multicore systems. In *USENIX NSDI*, 2014.

[37] Anuj Kalia, Michael Kaminsky, and David Andersen. Datacenter rpcs can be general and fast. In *USENIX NSDI*, 2019.

[38] Rishi Kapoor, George Porter, Malveeka Tewari, Geoffrey M. Voelker, and Amin Vahdat. Chronos: Predictable Low Latency for Data Center Applications. In *ACM SoCC*, 2012.

[39] Antoine Kaufmann, Tim Stamler, Simon Peter, Naveen Kr. Sharma, Arvind Krishnamurthy, and Thomas Anderson. TAS: TCP Acceleration as an OS Service. In *ACM Eurosys*, 2019.

[40] Reese Kuper, Ipoom Jeong, Yifan Yuan, Ren Wang, Narayan Ranganathan, Nikhil Rao, Jiayu Hu, Sanjay Kumar, Philip Lantz, and Nam Sung Kim. A Quantitative Analysis and Guidelines of Data Streaming Accelerator in Modern Intel Xeon Scalable Processors. In *ACM ASPLOS*, 2024.

[41] Michael Kurth, Ben Gras, Dennis Andriesse, Cristiano Giuffrida, Herbert Bos, and Kaveh Razavi. NetCAT: Practical Cache Attacks from the Network. In *IEEE Symposium on Security and Privacy*, 2020.

[42] Haifeng Li, Tianyue Lu, Yuhang Liu, and Mingyu Chen. Make Page Coloring more Efficient on Slice-Based Three-Level Cache. In *IEEE ICPADS*, 2019.

[43] Ilias Marinos, Robert N.M. Watson, and Mark Handley. Network Stack Specialization for Performance. In *ACM SIGCOMM*, 2014.

[44] Ilias Marinos, Robert N.M. Watson, Mark Handley, and Randall R. Stewart. Disk|Crypt|Net: Rethinking the Stack for High-Performance Video Streaming. In *ACM SIGCOMM*, 2017.

[45] Michael Marty, Marc de Kruijf, Jacob Adriaens, Christopher Alfeld, Sean Bauer, Carlo Contavalli, Michael Dalton, Nandita Dukkipati, William C. Evans, Steve Gribble, Nicholas Kidd, Roman Kokonov, Gautam Kumar, Carl Mauer, Emily Musick, Lena Olson, Erik Rubow, Michael Ryan, Kevin Springborn, Paul Turner, Valas Valancius, Xi Wang, and Amin Vahdat. Snap: a Microkernel Approach to Host Networking. In *ACM SOSP*, 2019.

[46] Clémentine Maurice, Nicolas Scouarnec, Christoph Neumann, Olivier Heen, and Aurélien Francillon. Reverse Engineering Intel Last-Level Cache Complex Addressing Using Performance Counters. In *RAID*, 2015.

[47] John D. McCalpin. Mapping addresses to l3/cha slices in intel processors. 2021.

[48] Rolf Neugebauer, Gianni Antichi, José Fernando Zazo, Yury Audzevich, Sergio López-Buedo, and Andrew W. Moore. Understanding PCIe performance for end host networking. In *ACM SIGCOMM*, 2018.

[49] Amy Ousterhout, Joshua Fried, Jonathan Behrens, Adam Belay, and Hari Balakrishnan. Shenango: Achieving High CPU Efficiency for Latency-sensitive Datacenter Workloads. In *USENIX NSDI*, 2019.

[50] Simon Peter, Jialin Li, Irene Zhang, Dan R. K. Ports, Doug Woos, Arvind Krishnamurthy, Thomas Anderson, and Timothy Roscoe. Arrakis: The operating system is the control plane. In *OSDI*, 2014.

[51] Boris Pismenny, Adam Morrison, and Dan Tsafrir. ShRing: Networking with Shared Receive Rings. In *USENIX OSDI*, 2023.

[52] Boris Pismenny, Adam Morrison, and Dan Tsafrir. Disentangling the Dual Role of NIC Receive Rings. In *USENIX OSDI*, 2025.

[53] George Prekas, Marios Kogias, and Edouard Bugnion. ZygOS: Achieving Low Tail Latency for Microsecond-scale Networked Tasks. In *ACM SOSP*, 2017.

[54] Deepti Raghavan, Shreya Ravi, Gina Yuan, Pratiksha Thaker, Sanjari Srivastava, Micah Murray, Pedro Henrique Penna, Amy Ousterhout, Philip Levis, Matei Zaharia, and Irene Zhang. Cornflakes: Zero-copy serialization for microsecond-scale networking. In *ACM SOSP*, 2023.

[55] Hugo Sadok, Nirav Atre, Zhipeng Zhao, Daniel S. Berger, James C. Hoe, Aurojit Panda, Justine Sherry, and Ren Wang. Enso: A Streaming Interface for NIC-Application Communication. In *USENIX OSDI*, 2023.

[56] Rajath Shashidhara, Tim Stamler, Antoine Kaufmann, and Simon Peter. FlexTOE: Flexible TCP offload with Fine-Grained parallelism. In *USENIX NSDI*, 2022.

[57] Mark Sutherland, Siddharth Gupta, Babak Falsafi, Virendra Marathe, Dionisios Pnevmatikatos, and Alexandros Daglis. The NEBULA RPC-optimized architecture. In *ACM/IEEE ISCA*, 2020.

[58] Amin Tootoonchian, Aurojit Panda, Chang Lan, Melvin Walls, Katerina Argyraki, Sylvia Ratnasamy, and Scott Shenker. ResQ: Enabling SLOs in Network Function Virtualization. In *USENIX NSDI*, 2018.

[59] Linus Torvalds. Page coloring and cache coloring on linux. https://yarchive.net/comp/linux/cache_coloring.html, 2003.

[60] Minhu Wang, Mingwei Xu, and Jianping Wu. Understanding I/O Direct Cache Access Performance for End Host Networking. *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, 6(1):1–37, 2022.

[61] Zhipeng Wei, Zehan Cui, and Mingyu Chen. Cracking Intel Sandy Bridge's Cache Hash Function. *arXiv preprint arXiv:1508.03767*, 2015.

[62] Ziye Yang, James R. Harris, Benjamin Walker, Daniel Verkamp, Changpeng Liu, Cunyin Chang, Gang Cao, Jonathan Stern, Vishal Verma, and Luse E. Paul. SPDK: A Development Kit to Build High Performance Storage Applications. In *IEEE CLOUDCOM*, 2017.

[63] Ying Ye, Richard West, Zhuoqun Cheng, and Ye Li. COLORIS: A Dynamic Cache Partitioning System Using Page Coloring. In *PACT*, 2014.

[64] Yifan Yuan, Mohammad Alian, Yipeng Wang, Ren Wang, Ilia Kurakin, Charlie Tai, and Nam Sung Kim. Don't Forget the I/O When Allocating Your LLC. In *ACM/IEEE ISCA*, 2021.

[65] Irene Zhang, Amanda Raybuck, Pratyush Patel, Kirk Olynykr, Jacob Nelson, Omar S. Navarro Leija, Ashlie Martinez, Jing Liu, Anna Kornfeld Simpson, Sujay Jayakar, Pedro Henrique Penna, Max Demoulin, Piali Choudhury, and Anirudh Badam. The Demikernel Datapath OS Architecture for Microsecond-scale Datacenter Systems. In *ACM SOSP*, 2021.

[66] Xiao Zhang, Sandhya Dwarkadas, and Kai Shen. Towards Practical Page Coloring-based Multi-core Cache Management. In *ACM EuroSys*, 2009.

[67] Yuhong Zhong, Daniel S. Berger, Carl Waldspurger, Ryan Wee, Ishwar Agarwal, Rajat Agarwal, Frank Hady, Karthik Kumar, Mark D. Hill, Mosharaf Chowdhury, and Asaf Cidon. Managing Memory Tiers with CXL in Virtualized Environments. In *USENIX OSDI*, 2024.

ARTIFACT EVALUATED
USENIX
AVAILABLE

Disentangling Graph Dependencies for Efficient Billion-Scale GPU Vector Search

Haoru Zhao*, Jingkai He*, Jingyao Zeng, Mingkai Dong, Dong Du
Institute of Parallel and Distributed Systems, Shanghai Jiao Tong University

Abstract

Graph-based approximate nearest neighbor search (ANNS) drives high-performance vector search for AI systems. Nowadays, GPU becomes the emerging ANNS platform for its high performance and cost efficiency. However, GPU's limited memory capacity hinders graph ANNS systems from scaling to billion-level, due to graph's high memory consumption (239–334 GB). Existing efforts mitigate this by offloading graph to CPU memory; however, this incurs severe performance penalties due to data transfer overhead and GPU stalls.

We identify the root cause of this inefficiency: a strict *step-level dependency* in graph search, where each step relies on the traversal and computations of all nodes in the previous step. Our key insight is that *this monolithic step-level dependency can be disentangled into a more flexible, fine-grained node-level dependency*. Specifically, for each node, it is first accessed as a neighbor via an edge (i.e., *discovery*), and later selected as a parent to traverse its neighbors (i.e., *expansion*). These two stages are typically separated by many steps, exposing a sufficient discovery-expansion window. Leveraging this time window, the edge fetching to access some neighbors can be deferred and overlapped with computation. Based on this insight, we propose FlowANN, a graph-based ANNS system that efficiently supports billion-scale search on a single GPU. FlowANN employs a tiered graph structure, offloading the edges connected to neighbors that have sufficient time windows to the CPU. It effectively pipelines GPU computation with edge fetching via optimized asynchronous transfer and dynamic coordination. Evaluations show that FlowANN outperforms state-of-the-art systems by 4.08–45.7× on average (up to 172.6×), without compromising search accuracy.

1 Introduction

Approximate nearest neighbor search (ANNS) serves as a foundational component of modern AI systems, and is widely used in retrieval-augmented generation (RAG) [1, 2], recommendation systems [3, 4], image retrieval [5], and LLM serving [6, 7]. These systems embed data into high-dimensional vectors and employ ANNS to identify nearest vectors to the query vector, enabling efficient retrieval of relevant information. Among all ANNS indexes, graph-based approaches [8–18] are favored for their superior search efficiency.

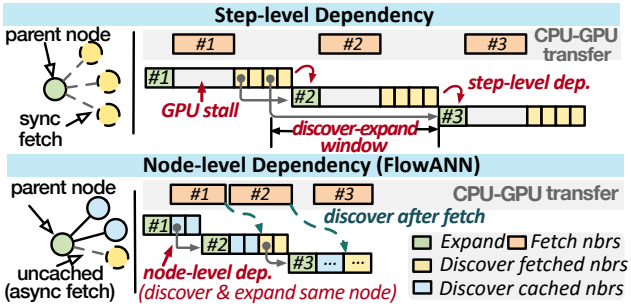


Figure 1: Step-level dependency vs. node-level dependency. *"nbr" is short for neighbor; "dep" is short for dependency.*

The rapid growth of data scales and the explosive computational demands of ANNS has driven the emerging adoption of GPUs for ANNS [19–23], leveraging their high computational efficiency and cost efficiency. GPU's parallel computing capabilities naturally align with ANNS' vector computation requirements, making a single mid-tier GPU achieve over 200× the throughput of a high-end CPU server[1] (Fig. 2). Thus, GPU ANNS has been widely adopted by the community [24, 25] and industry (Meta [26], NVIDIA [27], etc. [28–30]).

However, the limited memory capacity of GPUs poses significant challenges for deploying large-scale graph-based ANNS on GPUs. For instance, even after quantization [31, 32], billion-scale graph-based ANNS indexes [33–36] (datasets in Table 1) still require 258–350 GB memory, with the graph alone consuming 239–334 GB. This far exceeds the 80–96 GB memory capacity of mainstream GPUs.

Alternatively, offloading graph to host memory and fetching data on demand suffers from high data transfer overhead due to the strict dependencies in the search process. Specifically, graph ANNS employs a *best-first search* algorithm, which proceeds iteratively. In each step, it selects the node closest to the query from the *candidate pool* as the parent, fetches the parent's edges from the graph to retrieve its neighbors, computes all neighbors' distances to the query, and adds them into the candidate pool for subsequent steps. As the selection of the parent requires that all neighbors from the previous step

*Both authors contributed equally to this work and should be considered as co-first authors.

[1] GPU: NVIDIA H20, with 15% of computing capability (TFLOPS) of H100; CPU: 2x Intel Platinum 8457C. All experiments follow settings in §7.

complete the computations and are inserted into the candidate pool, this rigid step-level dependency results in significant GPU stalls to wait for edge fetching, as Fig. 1 shows.

However, this conventional view of step-level dependency is overly strict, and the associated waiting may be unnecessary. We disentangle the step-level dependency into a finer-grained *node-level dependency*. From a node's perspective, during the search process, it is first reached as a neighbor (i.e., *discovery*) and then used to find its neighbors (i.e., *expansion*). The best-first search algorithm ensures that, in each step, multiple nodes are *discovered* and only one node is *expanded*. Therefore, for most nodes, there exists a time window (i.e., several steps) between their *discovery* and *expansion* (**Insight #1**). For instance, ~95.6% of expanded nodes have an average window > 5 steps (§ 3).

Such discovery-expansion windows suggest an opportunity to offload some edges to CPU with (almost) zero overhead: when expanding a node, its offloaded edges can be asynchronously fetched to discover the neighbors. As long as these neighbors have sufficient discovery-expansion windows, their deferred discoveries will not disrupt the nodes' expansion order in search process (i.e., preserving best-first search's search path). Moreover, such deferred discovery effectively hides data transfer by overlapping it with computations.

However, though the discovery-expansion window presents an opportunity for zero-overhead offloading, identifying suitable edges for offloading remains challenging, since predicting the window is difficult. We find that the expanded nodes' nearby neighbors (i.e., connected via short edges) are expanded sooner, leaving insufficient windows for deferred discovery. This is because the node and its nearby neighbors are close in space, so once one is expanded (i.e., being near the query), the other is also likely to be close to the query and expanded soon. Therefore, generally, long edges are more suitable for offloading, while short edges should reside on the GPU to ensure timely discovery (**Insight #2**).

Based on these insights, we propose **FlowANN**, a GPU graph ANNS system that efficiently supports billion-scale search on a single GPU. As Fig. 1 shows, at each step, it discovers the parent's neighbors residing on the GPU and the nodes deferred from previous steps, while asynchronously fetching other neighbors from the CPU. We have mathematically proven that deferred discoveries do not compromise search accuracy, even if FlowANN might (sometimes) yield a different search path compared to synchronous edge fetching.

FlowANN further addresses the following challenges.

Challenge #1: Imbalanced graph tiering. While edge length provides a macroscopic criterion for graph tiering, selecting on-GPU edges based on a static edge length threshold leads to critical imbalances. Due to the uneven distribution of data points in high-dimensional space, edges in dense regions are typically short, while nodes in sparse regions have few short edges. Simply using a length threshold causes queries located in sparse regions to suffer from excessive uncached edges.

Challenge #2: Lagging edge fetching. FlowANN has to fetch edges to GPU asynchronously. However, despite many methods to copy data between CPU and GPU (e.g., `cudaMemcpy`, unified memory) [37], current GPU ecosystems lack support for GPU-initiated async transfers. Existing methods either require returning control to CPU to initiate async copies, or rely on GPU-initiated synchronous copies. Furthermore, existing CPU-GPU transfer methods typically rely on DMA. But DMA's high initialization overhead severely degrades the performance of edge fetching, whose size is relatively small.

Challenge #3: Uncoordinated pipelining. Although the discovery-expansion windows enable deferred discovery, excessive deferrals lead the search to suboptimal search paths, thereby sacrificing efficiency. Thus, FlowANN has to wait for some discoveries if their deferred steps exceed their discovery-expansion windows. However, estimating the window is challenging due to its dynamic evolution throughout the search. Furthermore, GPU computational resources (e.g., CUDA cores, shared memory) are statically allocated for the GPU kernel, but deferred discovery leads to dynamically varying workloads in each step, resulting in resource under-utilization.

To address the challenges, we propose the following designs. Firstly, we design the *grouping-based graph tiering*, which partitions the graph by grouping spatially proximate nodes and places edges between nodes within the same group on the GPU. Building on recent theoretical advances [38], we employ a multi-level label propagation scheme for grouping, which considers both edge lengths and node spatial distribution. Furthermore, we introduce a compact matrix layout to store the tiered graph, which reduces memory waste from padding and enables more edges to reside in GPU memory.

Secondly, we design *xCopier* to enable GPU-initiated asynchronous data transfers. xCopier coordinates GPU-side ring queues and CPU-side data-moving threads, and provides GPU-optimized programming primitives for GPU kernels. To facilitate edge fetching (i.e., small copies), xCopier employs MMIO-based data transfer, bypassing the overhead of conventional DMA-based approaches. xCopier makes edge fetching in FlowANN both trap-less and asynchronous, enabling effective overlap of data transfer and computation.

Finally, we design a coordinated pipeline for efficient execution. FlowANN employs the *adaptive synchronization* to prevent excessively deferred discoveries, which holistically considers global search convergence and local fluctuations. To efficiently utilize GPU resources, FlowANN introduces *cross-step balancing*, which adaptively balances workloads across steps to align computing demands with hardware resources.

We evaluate FlowANN on billion-scale datasets [39–41], and compare it with SOTA GPU ANNS systems from industry and academia. FlowANN outperforms cluster-based ANNS systems like cuVS [27], Faiss [24], Rummy [19], and FusionANNS [22] by 4.08–8.41× on average, while maintaining the same accuracy. It outperforms graph-based systems like BANG [20] and FlashANNS [42] by 45.7× and 14.3×

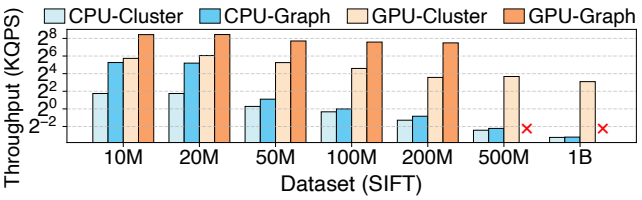


Figure 2: ANNS performance across indexes on GPU/CPU with different dataset sizes. *Recall@10 = 0.9. Recall@α means the proportion of true neighbors found within the top-α results. We use Faiss [24] (w/ HNSW [10]) and cuVS [27] for CPU/GPU ANNS. For CPU ANNS, all (160) cores are used. Detailed settings in §7.*

on average, and achieves comparable throughput (using one GPU) with multi-GPU graph system GGNN [43] using 8 GPUs. FlowANN consistently achieves improvements across different accuracies, GPUs, and algorithm configurations.

FlowANN is fully open-sourced at https://github.com/SJTU-IPADS/GPU-Graph-ANN.

2 GPU Graph ANNS: Fast yet Memory-bound

2.1 GPU Graph ANNS Primer

Vector search underpins modern AI systems, such as RAG [1, 2], recommendation systems [3, 4, 30], image retrieval [5], and LLM inference [6]. These systems embed data (e.g., text, images) into *vectors*, and use vector search to retrieve data relevant to the embedded query vector. The core operation of vector search is *k-nearest neighbor search*, which tries to find the k nearest vectors (*top-k*) to a query. As exact nearest neighbor search via brute-force is infeasible at large data scales (e.g., billion-level [33–35, 41]), modern vector search adopts *approximate nearest neighbor search (ANNS)*, which leverages pre-built indexes for efficient approximate retrieval.

Graph-based ANNS. Mainstream ANNS indexes generally fall into two broad categories: partition-based [44–49] and graph-based [8–12]. Partition-based indexes divide the dataset into multiple *partitions*, such as clusters of spatially proximate points or discrete buckets mapped via hashing.[2] Subsequently, the search process scans the vectors within the partitions that are close to the query via brute force. In contrast, graph-based methods construct a proximity graph to capture the relationships between vectors, and the search proceeds by traversing the graph.

Graph-based ANNS systems offer superior search efficiency over cluster-based ones. As shown in Fig. 2, they deliver 1.6–10.5× higher performance and superior scalability. This advantage stems from the graph structure, which allows the search to navigate directly toward nearest neighbors by graph traversal, significantly reducing distance computations compared to the brute-force scans of cluster-based methods.

[2]While partition-based indexes encompass various approaches (e.g., hash-based and cluster-based ones), our subsequent discussion mainly focuses on the cluster-based ones.

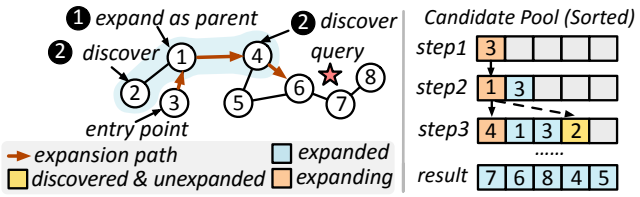


Figure 3: Best-first graph search procedure.

KNN graph and best-first search. The graph structure for ANNS is the *K-Nearest Neighbor (KNN)* graph, a graph where each node (vector) is connected to its approximately K nearest neighbors in the dataset, as determined by distance metrics like Euclidean distance. The graph is stored as a 2D matrix, where each row contains the K neighbor IDs for a node.

The graph search algorithm utilizes a *best-first search* strategy. As illustrated in Fig. 3, it maintains a candidate pool that contains a fixed number of the closest nodes found so far, sorted by their distance to the query. The algorithm begins from an entry point (e.g., node 3) and proceeds iteratively. At each iteration, it selects the nearest unexpanded node from the candidate pool as the *parent* and traverses its edges to obtain its neighbors (❶).[3] It then computes the distances between these neighbors and the query, and inserts them into the pool (❷). This process repeats until all nodes in the pool have been expanded, and the algorithm returns the top-k nearest vectors in the pool as the final result. During the search, each node undergoes two main stages: ❶ it is reached as a neighbor via an edge and is inserted into the candidate pool after distance computation (termed *discovery*); ❷ it is selected as the parent to further *discover* its neighbors (termed *expansion*).

GPU-based ANNS. Given the growing demand for large-scale vector processing (e.g., billions of vectors), GPU-based ANNS is increasingly favored for its parallel capabilities and cost-efficiency [22]. Currently, GPU-based ANNS is widely adopted by industry leaders such as Meta [26, 50], NVIDIA [27, 51], etc. [28–30]. With both data and indexes (e.g., graph) placed on the GPU, SOTA systems accelerate key ANNS operations (e.g., distance computation, sorting) through parallel execution [27, 51, 52], significantly boosting search speed and throughput. As Fig. 2 shows, GPU-based ANNS achieves a 9.0–222.0× speedup over CPU-based ones across various dataset sizes. Notably, graph-based GPU ANNS outperforms cluster-based ones by 5.2–15.1×, highlighting the superiority of graph-based ANNS on GPU.

2.2 Memory Strain in GPU Graph ANNS

Although exhibiting computational efficiency, the substantial memory footprint of the graph limits the applicability of GPU graph ANNS at billion-scale. For example, even after

[3]A common variant of best-first search selects multiple parents in each step (i.e., beam search). We also consider it in our evaluations (§7.4.2).

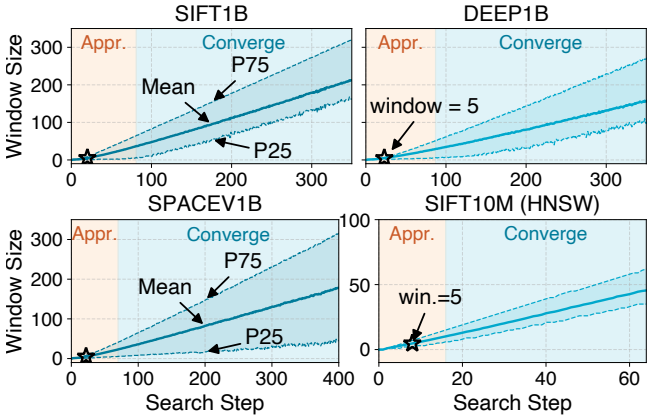


Figure 4: Discovery-expansion windows over steps. *For a node being expanded, its window size = current step − the step when it was discovered − 1. We also validate other KNN graph variant (i.e., HNSW) exhibits similar results. The HNSW index is from FAISS [24].*

quantization[4], which reduces the vector memory footprint to 12.5% of the original, graph ANNS on SIFT1B dataset [39] demands ~258 GB memory. This far exceeds the 80-96 GB capacity of mainstream GPUs, making it undeployable on a GPU (Fig. 2). Notably, the graph alone consumes ~93% of the total memory, constituting the key bottleneck. Such memory consumption is unavoidable to preserve adequate graph connectivity and ensure search quality. Thus, the core challenge in scaling GPU graph ANNS to billion-scale lies in *breaking the memory bottleneck while maintaining high performance.*

2.3 Existing Efforts and Possible Solutions

Multiple GPUs. Although employing multiple GPUs [23, 43] (e.g., through graph sharding) presents a theoretical solution, it faces non-trivial challenges. Firstly, efficient graph partitioning for ANNS remains challenging, as the search algorithm's step-by-step discovery-oriented nature makes the traversal path for a given query inherently unpredictable. Secondly, it introduces frequent inter-GPU communication, which consumes computational resources, increases search latency (several ms), and degrades throughput, as demonstrated in §7.1.

Unified memory (UM). UM extends memory capacity by creating a unified address space across CPU and GPU. However, it is ill-suited for graph traversal, which exhibits virtually *no locality*. Consequently, the traversal frequently triggers page faults and migrations (62 μs), which are exceedingly costly.

CPU-GPU tiering. Another approach is to store the graph primarily in CPU memory and transfer edges of parent nodes to GPU on demand (e.g., BANG [20]). However, the strict *step-level dependency* in best-first search severely limits computation-transfer overlap: each step's new parent selection depends on all the neighbors of the parent being fetched and *discovered* in the previous step. Consequently, it is nearly impossible to hide transfer latency by overlapping it with computation (< 10% overlapped). Additionally, this approach leads to severe GPU memory under-utilization (~50 GB idle on an 80 GB HBM GPU for the quantized SIFT1B dataset), as it treats GPU memory merely as a transient staging area.

[4]To enhance computational efficiency, SOTA systems search with lower-dimensional, quantized vectors, and return a candidate set with more than k items, which is *re-ranked* using the full-dimensional vectors to produce the final top-k results. Product Quantization (PQ) [32] is the most widely used.

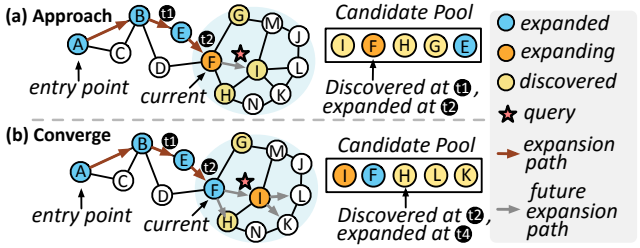


Figure 5: Two phases in the graph search process. *The upper figure shows approach phase, and the lower shows converge phase.*

3 Disentangling Dependency in Graph Search

Given the conventional strict step-level dependency, CPU-GPU tiering for graph ANNS seems infeasible due to high data transfer overhead. However, we find that *this step-level dependency is overly rigid and can be disentangled into* ***node-level dependency***. Specifically, at each step of a given best-first search path, only the node to be expanded needs to have been discovered. Based on this fine-grained dependency, we analyze the graph search traces for tens of thousands of queries on three billion-scale datasets (§7).[5] From the analysis, we identify opportunities to defer certain discoveries, thereby overlapping data transfer with computation without sacrificing accuracy.

Observation #1: Discovery-Expansion window. We discover that during the search process, *there exists a time window between a node's discovery and expansion* (**discovery-expansion window**). As Fig. 4 shows, for 95.6% of the search steps, the average discovery-expansion window exceeds 5 steps (6–14 μs per step, depending on the batch size). Moreover, the window size consistently grows throughout the search process.

Insight #1: Deferrable discovery. The discovery-expansion window presents an opportunity to defer the discovery of certain nodes without compromising accuracy. Specifically, since this window can typically overlap the edges' host-to-GPU transfer latency (~8 μs, Fig. 17b), we can offload the edges connecting to neighbors with sufficient windows to CPU memory, and defer the discovery of these neighbors until the edges are fetched to GPU. When fetching the edges, GPU can continue executing subsequent steps' computations without blocking.

Observation #2: Short edges indicate non-deferrable dis-

[5]We deploy CAGRA [51] (NVIDIA) using UM to capture node-level traces.

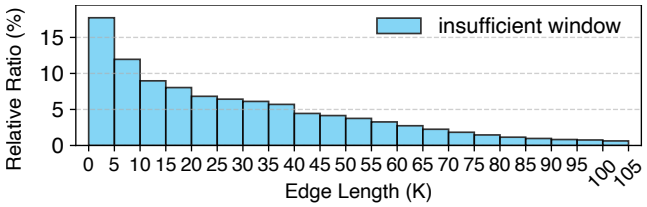


Figure 6: Proportion of edges with insufficient windows (≤ 5 steps) for each edge length bucket (SIFT1B). *E.g., in the shortest-edge bucket (0–5), 17.7% of the edges connect to neighbor nodes with insufficient windows. Other datasets exhibit similar results.*

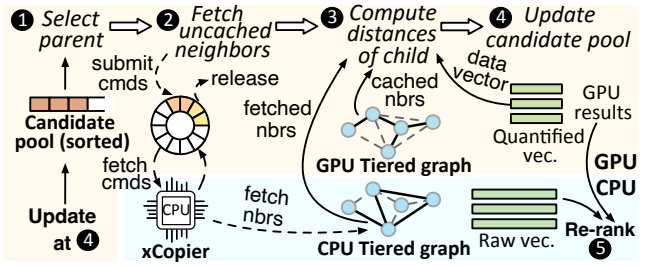


Figure 7: Overview of FlowANN workflow. *Currently, the tiered graph is constructed and placed offline. "nbr" stands for neighbor.*

coveries. While the discovery-expansion window presents a general opportunity for deferrable discovery, not all nodes are equally suitable for deferral. As Fig. 5 (a) shows, when expanding node F at t_2, its neighbor I is less suitable for deferral than G, as I will be expanded soon. To systematically identify non-deferrable discoveries, we analyze the correlation between edge length and window size. Specifically, for each expanded node, we collect the length of the edge used to discover it, and analyze edges whose corresponding discoveries have insufficient windows (conservatively ≤ 5 steps).

Results reveal that *discoveries through short edges are less likely to be deferrable.* As Fig.6 shows, shorter edges are more frequently associated with discoveries that have insufficient windows. The proportion of edges with insufficient windows in the shortest-edge bucket (17.7%) is much higher than that in the longest-edge bucket (0.63%). We also find that short edges have higher access frequency. Nevertheless, most discoveries (~90.6%) have sufficient windows (>5 steps) for deferral, demonstrating broad applicability of deferrable discovery.

Insight #2: Length-based graph tiering. In the graph, since discoveries via short edges are less deferrable and more frequently accessed, short edges should be retained in GPU memory. This effectively utilizes the remaining GPU memory after storing the quantized vectors. Conversely, long edges can be safely offloaded to host memory and fetched to GPU on demand, thus maximizing memory efficiency.

Root cause and generality. These observations are derived from the nature of best-first search, which aims to find the query's top-k closest nodes, specifically from its surrounding region (e.g., shaded area in Fig. 5). This process is naturally divided into two phases: *approach* and *converge* [11, 53].

During the approach phase, the search rapidly advances from the entry point toward the vicinity of the query (e.g., node A→F in Fig. 5 (a)). Each discovery tends to find nodes closer to the query, which are then quickly expanded, resulting in small windows. As Fig.4 shows, the first quartile of window size (*P25*) remains close to zero during the first ~22% steps.

However, upon entering the converge phase, the search has reached the nodes near the query (e.g., node F in Fig. 5 (b)). During this phase, the search generally follows a query-centric, outward-expanding trend [53] to collect the actual top-k nearest neighbors. As the search progresses, the likelihood of discovering a closer node diminishes (e.g., expansion of node I at t_3 cannot find a closer node than I). Consequently, the node selected for expansion (i.e., nearest unexpanded node) is typically the one that was discovered several steps earlier (e.g., node H at t_4), resulting in the presence of a window.

Meanwhile, since the nodes expanded during this phase are already close to the query, their nearby neighbors (connected by short edges) are also likely to be close to the query and thus expanded sooner, resulting in smaller windows. This pattern is reflected in the overall search behavior (Fig. 6) due to the dominance of the converge phase in the search process.

Correctness and accuracy. Some may concern that precisely predicting the discovery-expansion windows is difficult, potentially leading to over-deferral of some nodes' expansion, and thus compromising accuracy. We mathematically prove that, given the greedy nature of graph search, even in the worst case (i.e., all discoveries are over-deferred), FlowANN can achieve identical accuracy within a bounded and deterministic number of steps, which is proved in Appendix A. We further design the synchronization mechanism to proactively prevent excessive deferral of discoveries (§ 6.2).

4 FlowANN Overview

We design FlowANN, a graph-based ANNS system that efficiently handles billion-scale vector search on a single GPU.

Workflow. As Fig. 7 shows, FlowANN tiers the graph and offloads the edges with low access frequency and large discovery-expansion windows to the CPU. It fuses the entire GPU search process (multiple steps) into a single GPU kernel. Each step can be divided into four phases. ❶ First, it selects the closest unexpanded node to the query from the candidate pool as the parent. ❷ Subsequently, it asynchronously fetches the parent's offloaded edges (i.e., uncached neighbors) from the CPU-tiered graph to the GPU using xCopier. ❸ Meanwhile, for the parent's cached neighbors and those neighbors already fetched to the GPU, FlowANN calculates their distances to the query using their corresponding data vectors stored on the GPU. Like mainstream systems [24, 27], for better computational efficiency, the vectors stored on the GPU are quantized. ❹ Finally, the candidate pool is updated with these discovered neighbors and re-sorted. ❺ After the completion of the whole GPU search phase, the nodes in the

Algorithm 1 FlowANN Search

Require: Query q, tiered graph $(G_{\text{gpu}}, G_{\text{cpu}})$, pool size L, top-k

Initialize candidate pool C with the selected entry points for q
Deferred neighbors $D \leftarrow \emptyset$
while C has unexpanded nodes **do**
    $v \leftarrow$ nearest unexpanded node in C
    ▷ *Async neighbor fetch via xCopier (§6.1)*
    Async fetch v's neighbors in G_{cpu}; add to D
    ▷ *GPU-cached neighbors from graph tiering (§5)*
    $E \leftarrow$ neighbors of v in G_{gpu}
    $E \leftarrow E \cup \{u \in D \mid \text{transfer completed}\}$; update D;
    ▷ *Adaptive synchronization (§6.2)*
    Sync-wait overdue nodes in D; move to E
    for each $u \in E$ **do**
        compute distance between q and u; insert u into C
    end for
end while
return Rerank the candidates in C on CPU and return the top-k

candidate pool (i.e., GPU results) are re-ranked on CPU using the original full-precision vectors to ensure final result accuracy. The CPU re-ranking and GPU search are also pipelined across batches. Algorithm 1 provides the precise pseudocode of the search procedure.

GPU-CPU graph tiering. FlowANN prioritizes storing short edges on the GPU. To avoid distributional imbalance of cached edges caused by a static length threshold, it utilizes grouping to select edges for GPU residency, considering both edge length and graph spatial distribution. Specifically, we employ *multi-level label propagation* (§5.1) to partition the graph into balanced groups while maximizing the retention of short edges within groups, and then place intra-group edges on the GPU. Moreover, on GPU, FlowANN replaces each node's global ID from a billion-scale namespace with a shorter local ID within each group, and designs a compact matrix graph layout (§5.2), allowing more edges to reside on GPU.

Trap-less async data transfer. We design *xCopier*, a system service that enables GPU-initiated asynchronous data transfer (§6.1). xCopier further enhances transfer performance by leveraging hardware capabilities: it employs Memory-Mapped I/O (MMIO) via BAR mapping to replace DMA-based `cudaMemcpy`, reducing transfer latency by 40× and ensuring the overlap of data movement and computation.

Coordinated pipeline. FlowANN pipelines GPU computation and CPU-GPU transfer. To preserve search efficiency, it prevents excessively deferred expansion through *adaptive synchronization*, which considers both global search convergence progress and local fluctuations. FlowANN employs *cross-step balancing* to align the dynamically varying per-step workload with hardware resources, and selects suitable entry points to shorten the approach phase (§6.2).

Discussions. We discuss FlowANN's scope and generality.

• **Generality for graph structures.** FlowANN is based on the widely-used CAGRA [51] (i.e., NN-Descent [54]) graph.

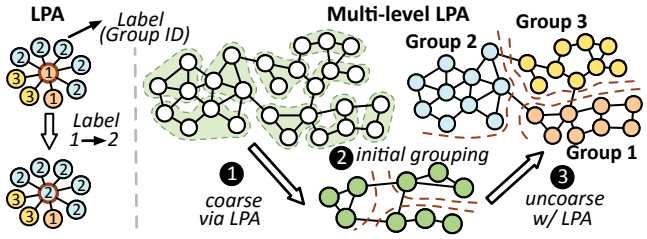


Figure 8: Multi-level LPA. *The node colors represent groups.*

Its motivations and designs are general and adaptable to various KNN graphs [9, 10].

• **Graph updates.** Like prior efforts [19, 20, 22, 55], FlowANN focuses on the search process. It can use existing online or offline graph update methods [56–59], and update the tiered graph with minimal effort, e.g., via a single round of label propagation (§5.1) to assign new nodes to groups.

• **Cluster serving and multi-GPU.** FlowANN aims to maximize single-GPU capability for billion-scale ANNS. It can be deployed with multiple replicas in clusters. It can be integrated into multi-GPU systems for prospective larger datasets.

• **Hardware generality.** While FlowANN is implemented on NVIDIA GPUs, which represent cutting-edge architectures, it can be adapted to other SIMT-based GPUs (e.g., AMD).

5 Deferrable Discovery with Graph Tiering

Guided by *Insight #2* (§3), graph tiering should prioritize placing short edges on GPU. However, applying a fixed length threshold to select on-GPU edges causes imbalance due to the non-uniform distribution of data points in high-dimensional space. Specifically, in dense regions, many edges are short and would be stored on GPU, while nodes in sparse regions have few edges cached. This leads to queries located in sparse regions benefiting little from tiering. Thus, graph tiering should consider both edge length and spatial node distribution.

We find that after grouping spatially proximate nodes into the same group (i.e., *grouping*), the intra-group edges are suitable for GPU placement: (1) Nodes connected by short edges are often close to each other and naturally cluster within the same group, which satisfies the goal of prioritizing short edges; (2) Grouping holistically considers the spatial distribution of all nodes, preventing nodes in sparse regions from being excluded by a single edge length threshold.

Challenges. However, designing a high-quality graph grouping and tiering scheme is still challenging due to two issues. First, partitioning a billion-scale graph to maximize the retention of short edges within groups is an NP-Complete problem [60, 61]. Simply applying common grouping methods (e.g., K-means) often causes poor grouping quality, as they only consider global node distribution while ignoring the graph's structural information (e.g., edge connections). For instance, when grouping via K-means, only ~60% of edges

Algorithm 2 Multi-level LPA Graph Grouping

Require: KNN graph $G = (V, E)$, max group size $S_{\max}$
Assign edge weight $w(e) \leftarrow mapping(len(e))$
Assign each node with an initial group label
▷ *Coarsening (❶ in Fig. 8)*
while graph is not sufficiently small **do**
    ▷ *Weighted LPA*
    For each node v, sum edge weights per neighbor label l:
        $totalWeight(l) = \sum_{u \in neighbors(v), label(u)=l} w(v,u)$
    $label(v) \leftarrow$ the label l with maximum $totalWeight(l)$
    Merge nodes sharing a label into a super-node
end while
▷ *Initial grouping (❷ in Fig. 8)*
Partition the coarsest graph via recursive bipartitioning
▷ *Uncoarsening + refinement (❸ in Fig. 8)*
for each level from coarsest back to original graph **do**
    Copy each super-node label to its finer-level nodes
    Run weighted LPA to adjust boundary nodes
end for
return the final group label for each node

that have insufficient windows are retained within groups (Fig. 17a). This necessitates a grouping scheme tailored to KNN graphs to maximize intra-group short edge retention, while keeping computational complexity tractable. Second, after grouping, the number of intra-group neighbors per node may vary. Continuing to store each node's neighbors in a 2D matrix results in non-trivial memory waste due to padding.

5.1 Spatial Locality-aware Graph Grouping

For high intra-group short edge retention, the graph grouping scheme in FlowANN should consider both edge connections and node distribution. To achieve this, we employ *Multi-level label propagation (multi-level LPA)* for graph grouping, which is inspired by a theoretical graph grouping scheme [38].

Procedure. Multi-level LPA is based on the *size-constrained weighted label propagation*, which proceeds as follows: First, each edge is assigned a weight, where shorter edges receive higher weights. Then, each node is initialized with a unique group label. In each subsequent iteration, every node calculates the sum of weights from its neighbors for each distinct label and updates its own label to the one with the highest total weight [62] (Fig. 8).[6] To ensure balanced group sizes, we introduce a size constraint to limit the maximum group size during propagation.[7] Notably, the edge weight efficiently incorporates edge length information, ensuring that shorter edges have a higher likelihood of being retained within groups.

As Fig. 8 shows, multi-level LPA involves a coarsen-uncoarsen process: The graph is recursively coarsened by grouping nodes into super-nodes through LPA until it is sufficiently small (❶); Then initial groups are generated on the coarsened graph via recursive bipartitioning [63] (❷); Finally, the groups are projected back to the original graph, with LPA applied during uncoarsening to refine group boundaries (❸). The full procedure is given in Algorithm 2.

[6] If all neighbor labels' summed weights are tied, one is randomly chosen.

[7] FlowANN chooses the maximum group size that retains the most intra-group short edges while staying within the current GPU memory budget.

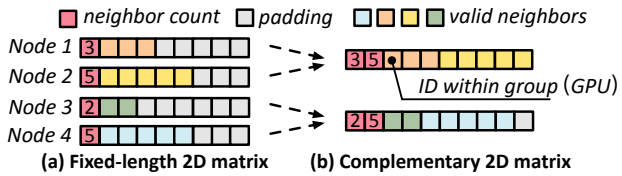


Figure 9: Graph layout with complement.

Benefits. Multi-level LPA can effectively preserve short edges within groups and obtain balanced, high-quality groups from a global perspective. This is mainly because: (1) *local structure awareness*: it captures local graph structure through label propagation based on edge connections and edge lengths, ensuring that nodes connected by short edges are grouped together; (2) *global distribution consideration*: by generating groups from a global view of the coarsened graph and refining them locally during uncoarsening, it holistically takes the nodes' distribution into account. Moreover, since LPA mainly relies on local label propagation, it achieves near-linear time complexity, making it well-suited for billion-scale graphs.

5.2 Compact Graph Layout with Complement

To ensure quick random access to each node's neighbors, existing graph ANNS systems store the graph on GPU using a 2D matrix with fixed-length rows, where each row corresponds to a node's neighbor list, as shown in Fig. 9 (a). However, after grouping, the number of intra-group neighbors for each node becomes variable. Continuing to use this 2D matrix would lead to significant memory waste due to padding (Fig. 9 (a)).

Through the analysis of intra-group neighbor count distributions, we identify a potential memory-saving opportunity: the distribution exhibits a quasi-symmetric shape, indicating that the space saved by the half of nodes with fewer neighbors effectively complements the extra space required by the half with more neighbors. Thus, as Fig. 9 (b) shows, we employ a new graph layout for the subgraphs on the GPU and CPU.

For the subgraph on the GPU, nodes are first sorted by their intra-group neighbor counts. They are then paired by iteratively matching the node with the most intra-group neighbors to the one with the fewest. Each pair is stored in a single row. The case for CPU is similar. This method not only reduces space waste from padding (§ 7.3) but also preserves efficient random access, as the graph is still stored in a 2D matrix with a fixed number of neighbor lists per row.

Moreover, leveraging the property that nodes in the GPU-tiered graph are only connected within the same group, we use shorter group-local IDs (e.g., 20–24 bits) rather than global graph-wide IDs (e.g., 32 bits) to represent each node. This

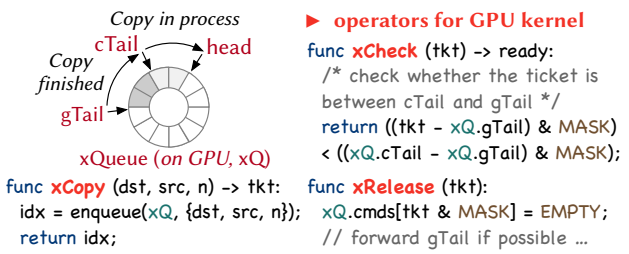


Figure 10: xQueue operators. *The atomic operations are simplified.*

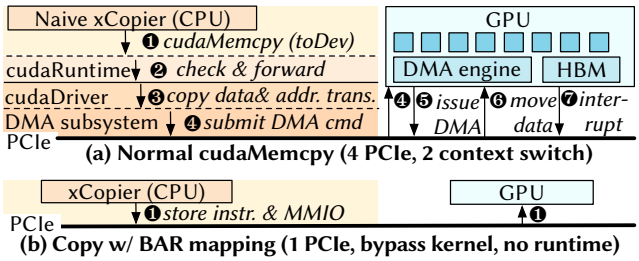


Figure 11: Copying process of xCopier and cudaMemcpy.

enables more edges to reside on the GPU (Fig. 17a), thereby allowing more computation to overlap with data transfer.

6 Execution with Coordinated Pipeline

The tiered graph enables deferred discovery. However, efficient transfer-computation pipelining remains challenging. We design xCopier to support efficient trap-less asynchronous transfer (§6.1), and further develop the coordinated pipeline that ensures search efficiency and hardware utilization (§6.2).

6.1 Streaming Data Movement with xCopier

During search, FlowANN employs an asynchronous data pipeline to fetch non-cached edges from the CPU to the GPU, overlapping transfer cost with on-GPU computation.

Challenges. This design, however, is impeded by two fundamental constraints in GPU architecture: (1) To minimize kernel launch overhead, FlowANN fuses all search steps into a monolithic kernel. However, no existing mechanism permits a GPU kernel to initiate asynchronous copies between device and host. (2) The performance of standard CUDA copy operations (e.g., *cudaMemcpy*) [37, 64] is poor for small data blocks (typically 64–256 bytes per node's edges), as they are bottlenecked by the inherent DMA engine launch overhead.

CPU-GPU data movement. We begin by recapturing existing data movement mechanisms between the CPU and GPU. On CPU side, transfers are initiated using *cudaMemcpy* [37], which can be asynchronous with GPU computation. On GPU side, data access to CPU memory is facilitated through two mechanisms: Unified Memory (UM) and Zero-copy (Pinned) Memory [37]. UM provides a unified address space, migrating pages (triggered by GPU page faults) upon access. Zero-copy Memory enables direct GPU access to CPU memory. Critically, both are synchronous and block GPU kernel execution.

For FlowANN, each transfer mechanism presents a critical limitation. CPU-initiated transfers require the search flow to return to the CPU after each step (i.e., trap) [20], thereby incurring substantial kernel termination and launch overhead. Conversely, Unified Memory and Zero-copy Memory forfeit the ability to construct an asynchronous copy pipeline.

Trap-less async copy with xCopier. We introduce xCopier, a system-level service for asynchronous data movement initiated directly from GPU threads, bridging the architectural gap in the GPU ecosystem. xCopier is built around a GPU-side ring queue (xQueue) and a CPU-side data-moving thread (xThread). The xQueue, featuring one head and two tails, exposes three primitives: xCopy, xCheck, and xRelease.

Specifically, as Fig. 10 shows, a GPU thread initiates an async transfer via xCopy, which enqueues a transfer command into xQueue and returns its queue index as the ticket. The xThread polls xQueue's head, detects new commands, executes the data transfers, and finally advances the cTail (cpu_tail). The GPU thread checks the status of an async transfer with xCheck, which compares the ticket against current cTail. Once the transfer is confirmed completed, the thread can reclaim the command slot using xRelease. The release is a two-step process, which allows out-of-order release: it first marks the command as invalid, then attempts to advance gTail (gpu_tail) if there are invalid commands at the tail.

These primitives are highly optimized for GPU parallelism. We employ batched invocation and warp-level aggregation, so that all xCopy/xRelease in a warp require just one atomic operation to update head/gTail. xCopier further mitigates atomic operation contention by supporting multiple xQueues. The xThread can be scaled to multiple threads under high load.

Efficient data transfer via BAR mapping. In FlowANN, every data transfer command involves copying the uncached edges of a parent node (64–256 bytes). Even when batched, the transfers remain relatively small. However, conventional DMA-based mechanisms (e.g., cudaMemcpy) perform poorly for such fine-grained transfers. As Fig. 11 (a) shows, this inefficiency stems from the DMA overhead, which involves four PCIe transactions, two context switches into/from GPU driver, and runtime overhead. A cudaMemcpy-based xCopier transfer for 64B–1KB data incurs a latency as high as 21–22 μs (Fig. 12), which cannot be overlapped with GPU computation.

To solve this challenge, we employ a novel hardware strategy: GPU's Base Address Register (BAR) mapping. Leveraging the open-source driver [65][8], xCopier maps the GPU memory regions directly into CPU's physical address space. Subsequent accesses to the regions are then performed via Memory-Mapped I/O (MMIO). With hardware write-combining, consecutive writes are transparently coalesced into fewer, larger MMIOs. As Fig. 11 (b) shows, this reduces CPU-to-GPU copy

[8] BAR mapping is also supported by other GPUs, e.g., AMD ROCm [66].

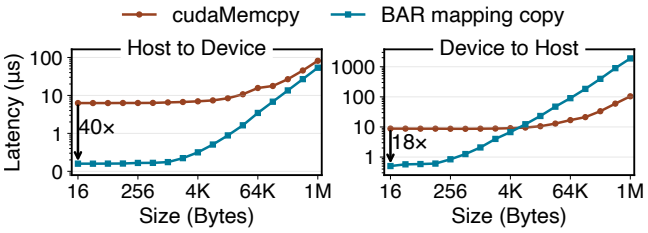


Figure 12: Copy latency of BAR mapping and cudaMemcpy. *Each (batched) xCopier transfer requires one D2H copy (xQueue head read) and two H2D copies (data write, xQueue cTail update).*

to a single PCIe transaction, which is executed by CPU SIMD instructions, eliminating context switches and runtime overhead, and accelerating small transfers by up to 40× (Fig. 12).

In FlowANN, each xQueue is associated with a pre-allocated (and BAR-mapped) GPU buffer pool for the fetched edges. As a general-purpose system service [67], xCopier is readily applicable to other GPU workloads [68–70].

6.2 Pipelining with Adaptive Coordination

Leveraging xCopier, FlowANN pipelines GPU computation with CPU-GPU edge fetching. At each step, the search kernel first selects the unexpanded node closest to the query from the candidate pool as the parent, then asynchronously fetches its uncached neighbors to GPU (xCopy). Subsequently, it checks the completion status of previously issued transfer commands (xCheck), and discovers both the parent's cached neighbors and the neighbors that have already been transferred.

Challenges. Efficient pipelining faces non-trivial challenges. ❶ Although discovery-expansion windows enable deferred discovery, excessive deferrals lead the search to suboptimal paths, sacrificing efficiency (§6.2.1). ❷ Due to deferred discoveries, the discovery workload per step varies dynamically, resulting in a mismatch with pre-determined hardware resources (§ 6.2.2). ❸ During the approach phase (§ 3), the windows are too small to hide edge transfers (§6.2.3).

6.2.1 Near-optimal Search with Adaptive Synchronization

Excessive deferral of discoveries leads the search to suboptimal paths, sacrificing efficiency. To avoid unnecessary computation and transfer, FlowANN synchronously waits for the completion of some discoveries if their deferred steps exceed their discovery-expansion windows. However, estimating the window is non-trivial, as it varies with the search process.

Estimating the window size. We find *the expanded node's position in the candidate pool* (P_e) can be an effective indicator to estimate the window size. Specifically, during the approach phase (§ 3), the search rapidly advances toward the query, and both the window size and P_e are usually zero. As the search converges, the window size increases (Fig. 4). Meanwhile, the expanded node's position shifts rightwards (i.e., P_e increases), as nodes near the query gradually accumulate at the head of the candidate pool and are successively expanded. Our analysis across tens of thousands of query traces confirms the strong correlation between P_e and the window sizes of the expanded node's neighbors, with a Pearson correlation[9] = 0.77.

FlowANN thus uses P_e to estimate the window size (W) of the expanded node's neighbors in each step, i.e., $W = \alpha * P_e$, where α is derived from offline profiling with linear regression. Relying on offline profiling is sufficient here because α depends primarily on the dataset's underlying data distribution. Furthermore, this formulation robustly adapts to dynamic variations in the approach phase's length. Specifically, since P_e typically remains zero during the approach phase, the estimated window W will also be zero, ensuring accurate synchronization without depending on offline statistics.

Beyond the normal cases, we observe that *fluctuations* during search convergence can lead to unexpectedly small windows, which manifest as a sharp drop in P_e. This happens when the search in the converge phase unexpectedly discovers a node much closer to the query (i.e., a sharp drop in P_e). Once such a node is found, its neighbors become the prime targets for subsequent expansion, thus their windows are small. FlowANN needs to handle such cases specially.

Adaptive synchronization based on window size. To handle the fluctuation cases, when FlowANN detects a sharp drop in P_e, it waits for all neighbors of the parent node to be fetched. While this incurs a performance cost, such fluctuations occur with very low probability (<3%). In our evaluation, FlowANN sets the threshold for a sharp drop in P_e to 10% of the candidate pool size. Meanwhile, in each step, if some neighbors from previous steps have been deferred for more than their estimated windows, FlowANN also waits for their discoveries to complete.

Stall-less synchronization. To avoid wasting computational resources, FlowANN yields the hardware threads to schedule other queries' computations during the wait. This scheduling is very lightweight, as it is handled by the hardware scheduler.

6.2.2 Regularizing Workload with Cross-Step Balancing

In previous synchronous execution, the number of nodes discovered per step (i.e., the parent node's neighbor count) is deterministic. However, as FlowANN defers some nodes' discoveries, this count becomes dynamic. This causes severe GPU under-utilization because computational resources (e.g., CUDA cores, shared memory) are statically predetermined.

FlowANN introduces a cross-step balancing mechanism to align the discovery workload with hardware resources. Specifically, when each step begins, FlowANN identifies three categories of nodes to discover: ❶ the parent's neighbors that reside on the GPU, ❷ deferred nodes that must be discovered

[9] Pearson correlation measures the linear relationship between two variables, with values > 0.7 indicating strong correlation [71, 72].

due to synchronization constraints, and ❸ deferred nodes that have already been transferred to the GPU but can be further deferred. If the total number of these nodes does not align with the hardware resources, FlowANN defers the discovery of some nodes from category ❸ to the next step. For example, if 32 thread teams (each team discovers the nodes collaboratively) are allocated for each query, FlowANN tries to align the number of nodes discovered per step to a multiple of 32. The extra nodes are passed to the next step via shared memory.

6.2.3 Shortening Approach Phase with Entry Points

In the approach phase (§ 3), the small discovery-expansion windows necessitate frequent synchronization, leading to non-negligible overhead. To reduce the proportion of this phase, FlowANN selects suitable entry points for each query, instead of starting from random nodes. Inspired by the principles of cluster-based ANNS, we select a set of representative nodes from the dataset (e.g., K-means centroids) as candidate entry points, and choose the ones closest to the query through parallel computation. This approach quickly identifies suitable entry points by leveraging the GPU's parallel capabilities. Thanks to the selected nodes' representativeness, only a very small number of candidates (one in a million) are required to reduce the approach phase to ~5% of total search steps.

7 Evaluation

Implementation. We build FlowANN with 14,700 lines of CUDA/C++ based on CAGRA [51] (i.e., graph-based ANNS in cuVS [27]), a SOTA system open-sourced by NVIDIA.

Hardware setup and datasets. All experiments run on a server with 2 Intel Xeon Platinum 8457C CPUs (2.60 GHz, 160 cores), 2 TB DRAM, and NVIDIA H20 GPUs. We also evaluate FlowANN on other GPUs (§ 7.4.2). We mainly use three widely-used billion-scale datasets [39–41]. The specifications of GPUs and datasets are summarized in Table 1.

Baselines. We compare with 8 GPU ANNS systems.

• **Quantized cluster-based ANNS.** SOTA cluster-based ANNS systems first search on GPU using quantized vectors (e.g., PQ [32]), then re-rank the results with full-precision vectors on CPU. We evaluate *cuVS-cluster* [27] (NVIDIA) and *faiss-cluster* [24] (Meta). We carefully implement in-memory *FusionANNS* [22] based on cuVS as it is not open-source.

• **Full-precision cluster-based ANNS.** *Rummy* [19] stores all full vectors on CPU and transfers them to GPU on demand, overlapping data transfer with computation via pipelining.

• **Graph-based ANNS.** *BANG* [20] supports in-memory billion-scale graph ANNS on a single GPU. It stores PQ-quantized vectors on GPU, while the graph and full vectors are stored in CPU memory and transferred to GPU on-demand.

• **Graph ANNS with related dependency.** The latest work, *FlashANNS* [42] relaxes step-level dependency for better pipelining. Its step *n* depends on step *(n - 2)*. We implement its in-memory version based on its available artifact.

	Dataset	Dimension	Type	#Vector	#Queries	PQ dim
Datasets	**SIFT1B**	128	uint8	1B	10,000	16
	DEEP1B	96	float32	1B	10,000	24
	SPACEV1B	100	int8	1.4B	29,316	13

	GPU & Arch.	Memory	GPU & Arch.	Memory
GPUs	**H20 (Hopper)**	96 GB HBM3	**A800 (Ampere)**	80 GB HBM2
	L20 (Ada)	48 GB GDDR6	**V100 (Volta)**	32 GB HBM2

Table 1: Datasets and GPUs used in evaluation.

• **Unified memory (UM).** UM unifies CPU and GPU memory. We extend CAGRA [51] with UM (*CAGRA-UM*).

• **Multi-GPU.** *GGNN* [43] is the only billion-scale multi-GPU graph ANNS system known to us.[10] It shards data across multiple GPUs, with each GPU performing searches on its corresponding subgraph, and then merges the results.

Configurations. Given the characteristics of the datasets and the constraints of CAGRA, we quantize the vectors to the dimensions in Table 1 (*PQ dim*) for GPU search, and apply the same quantization settings to cuVS-cluster and faiss-cluster. However, since BANG cannot achieve high accuracy with these quantization parameters, we use BANG's default quantization settings (PQ dim = 74) to ensure it attains comparable accuracy. For FlowANN and all graph-based baselines, we set the graph degree to 32, following CAGRA [51]'s official practice. We also measure FlowANN's performance under different quantization settings and degrees in § 7.2.

Metrics. We measure search accuracy using *recall@k* [32]: the fraction of true *k* nearest neighbors in the returned results. Without explicit mention, we measure the throughput (i.e., queries per second, QPS) and latency at 0.9 recall, a commonly used accuracy target in ANNS [11, 36, 73]. We also evaluate FlowANN under different accuracy targets in § 7.1.

7.1 Overall Performance

We evaluate FlowANN and the baseline systems under various batch sizes (from 16 to 2048) to reflect different application scenarios, such as online serving (small batch size) and offline processing (large batch size), similar to prior works [19].

Throughput. As Fig. 13 shows, FlowANN outperforms all baselines across batch sizes and datasets. Compared to cluster-based baselines, it achieves 8.67× and 8.15× higher average throughput for batches of 2048 and 16, respectively. The performance gain improves with larger batches, as the increased per-step computation time better hides data transfer. The cluster-based methods underperform FlowANN and scale poorly with batch size as their brute-force intra-cluster search

[10] A recently proposed system (PathWeaver [23]) cannot support billion-scale datasets on our testbed due to its high memory consumption.

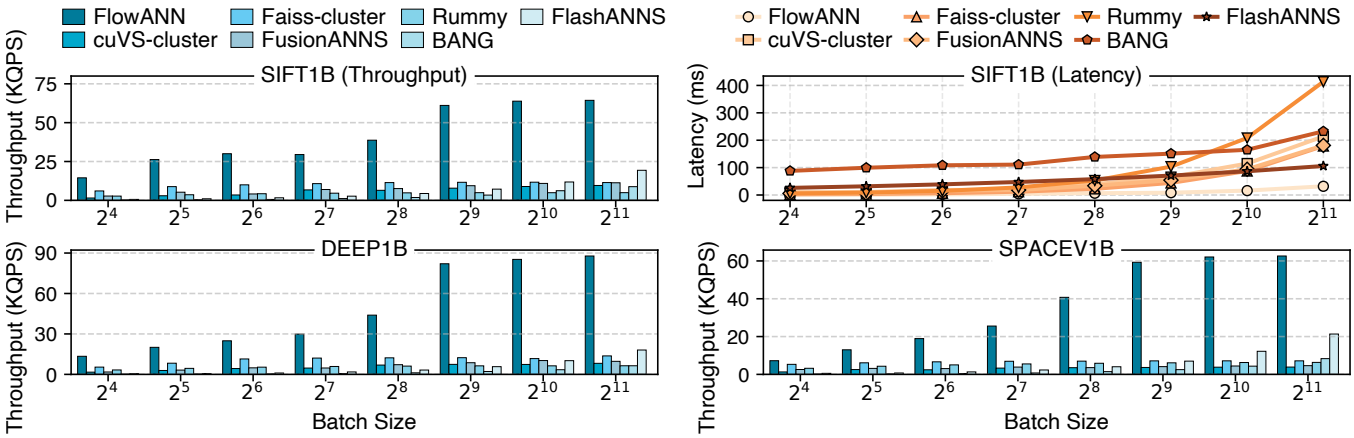


Figure 13: Throughput and average latency of FlowANN and baselines. *Recall@10 = 0.9. Due to space constraints, we only present the latency results on SIFT1B; trends on DEEP1B and SPACEV1B are similar. We omit CAGRA-UM in the figure for its low performance.*

saturates GPU resources earlier than graph traversal. FusionANNS outperforms cuVS-cluster because of its heuristic re-ranking.

FlowANN achieves 9.52× and 78.8× higher throughput than BANG on average for batch sizes 2048 and 16. This is because BANG only overlaps edge fetching with limited computation (i.e., pool updates) and relies on costly *cudaMemcpy*, causing GPU stalls. Smaller batch sizes further worsen BANG's overlapping efficiency. In contrast, FlowANN effectively hides data transfer by leveraging the discovery-expansion window and ensuring efficient data transfer via xCopier.

FlowANN outperforms FlashANNS by 3.71× and 21.8× on average for batch sizes 2048 and 16. FlashANNS' relaxation of step-level dependency is limited and inflexible. It fails to overlap I/O effectively for small batches. It leads to suboptimal search paths during the approach phase, and excessive waiting in the converge phase.

Compared to CAGRA-UM, FlowANN achieves 110.8–1888.8× higher throughput. Since UM is unaware of the graph traversal's random access patterns, frequent page faults and migrations are triggered, severely hampering performance.

Latency. FlowANN achieves lower latency than all baselines. It reduces latency by 83.8% and 81.6% on average compared to baselines for batch sizes of 2048 and 16. Notably, at batch size 16, it only takes 0.962 ms to process a query. This efficiency stems from its accurate graph traversal and efficient pipeline, which preserves high accuracy while effectively hiding transfer overhead. It reduces the P99 tail latency by 75.4% and 86.1% on average for batch sizes of 2048 and 16.

Cluster-based baselines exhibit higher latency than FlowANN, especially in larger batches. Among them, Rummy has the highest latency, as its full-precision GPU computation increases processing overhead. BANG and FlashANNS exhibit high latency at small batches as the non-overlapped data transfer becomes more pronounced with fewer queries.

Comparison with multi-GPU systems. We compare single-GPU FlowANN with GGNN [43]. As Fig.14 shows, even with 8 GPUs, single-GPU FlowANN outperforms GGNN by 2.22–15.3× at batch sizes of 8–1024. Only at batch size 2048 does FlowANN's throughput slightly lag behind 8-GPU GGNN (92.2%). GGNN requires 8 GPUs and a large batch to match FlowANN, because of its high communication overhead and GPU under-utilization at small batches.

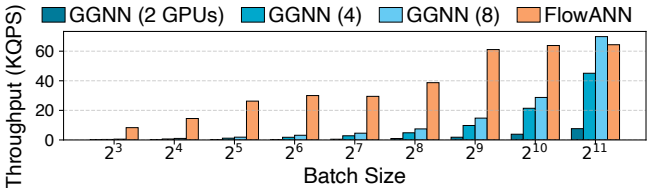


Figure 14: Throughput of FlowANN and GGNN on SIFT1B. *We use GGNN's default configurations. When running on 2 GPUs, GGNN offloads part of the graph and data to CPU memory.*

Different accuracy targets. As shown in Fig.15, we measure FlowANN's throughput across recall@10 from 0.8 to 0.995 under small (64) and large (2048) batch sizes. We choose cuVS-cluster as the representative of cluster-based methods, as it shares fundamental operations (e.g., distance computation) with FlowANN, enabling a more direct comparison. FlowANN outperforms baselines across all accuracy levels, and the performance gap widens at higher accuracy. At recall = 0.8, FlowANN outperforms cuVS-cluster and BANG by 5.4× and 28.9× on average; At recall = 0.995, these gains increase substantially to 29.1× and 111.8×, respectively.

Impact of deferred discovery on search steps. We compare search steps to reach the same accuracy with and without deferred discovery (via CAGRA-UM). Across three billion-scale datasets at accuracies from 0.8 to 0.995, FlowANN introduces no extra steps in most cases (~96%), with only a few showing a marginal increase of ~0.7–2.1% in steps. This shows that deferred discovery preserves search correctness and has a negligible impact on convergence speed in practice.

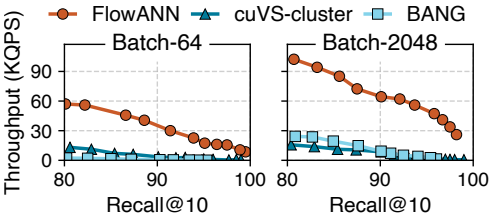
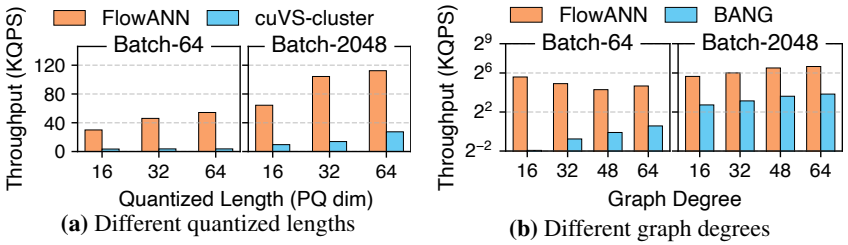


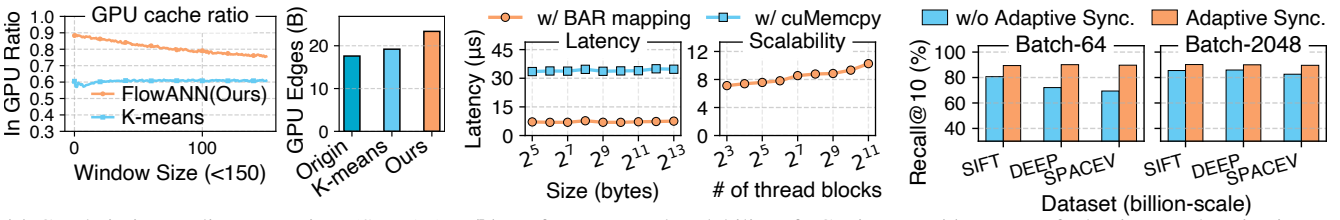
Figure 15: Throughput under different accuracy (SIFT1B).



(a) Different quantized lengths

(b) Different graph degrees

Figure 16: Throughput on SIFT1B dataset with different configurations. *(a) These three lengths are the only supported lengths by CAGRA given the dimensionality of SIFT.*



(a) Graph tiering quality comparison (SIFT1B)

(b) Performance and scalability of xCopier

(c) Impact of adaptive synchronization

Figure 17: Breakdown analysis. *For Fig. (a), all methods use the same GPU memory budget (66 GB).*

7.2 Sensitivity Analysis

We evaluate FlowANN under different configurations, i.e., various quantization settings and graph degrees.

Quantization configurations. We evaluate FlowANN under varying quantized vector lengths, using batch sizes of 64 and 2048 (Fig. 16 (a)). Across different quantization configurations, FlowANN achieves average throughput improvements of 8.47× (small batch) and 5.12× (large batch) over cuVS-cluster, demonstrating its adaptability to different quantization settings. Longer quantized vectors improve traversal accuracy, reducing search steps to reach target accuracy.

Graph degree. We evaluate FlowANN under varying graph degrees (i.e., neighbor counts). It outperforms BANG in all cases, achieving 136.37× (small batch) and 14.72× (large batch) higher throughput on average (Fig. 16 (b)). A higher graph degree discovers more edges, improving accuracy and reducing the number of steps, but raising transfer and computational overhead. This degrades FlowANN's throughput with small batches; whereas under large batches, the transfer is overlapped by computation, resulting in performance gains.

7.3 Breakdown Analysis

Graph tiering. To evaluate multi-level LPA, we measure the GPU edge cache ratio for each discovery-expansion window size of all queries (Fig. 17a). For edges with insufficient windows (i.e., window ≤ 5, § 3), it keeps ~87.9% of them on the GPU (89.1% for window = 0), a 29.6% improvement over K-means. Moreover, Fig. 17a reveals that FlowANN improves memory efficiency via group-local node IDs, storing 18.2% and 30.2% more edges on GPU than K-means and the original (no grouping) method. This is because multi-level LPA generates smaller, more balanced groups than K-means. The complement-based graph layout reduces space waste, leaving only ~0.506% padding for billion-scale datasets, reducing memory waste by ~98.5% compared to the original layout.

xCopier. We evaluate xCopier on different data sizes and concurrency. As Fig. 17b shows, compared to using cudaMemcpy, xCopier with BAR mapping reduces the end-to-end fetching latency by 78–80% for 32–8192 B data. xCopier achieves high scalability to serve 2048 concurrent thread blocks (1024 threads per block) with 64 xQueues during search, thanks to its lock-free design and thread block-level aggregation.

Adaptive synchronization. We compare the search accuracy with and without adaptive synchronization, using the same number of search steps. As Fig. 17c shows, adaptive synchronization improves accuracy by 21.7% and 6.2% on average at batch sizes of 64 and 2048, enabling high-accuracy search of recall@10 > 0.9. Smaller batches lead to shorter computation time per step, demanding more proper synchronizations.

7.4 Extensive Studies

7.4.1 Performance Upper Bound and Lower Bound

Gap to ideal upper bound. We compare FlowANN with CAGRA on small datasets. It caches only 50% of the edges, while CAGRA retains the entire graph on the GPU. As Fig. 18 shows, at batch sizes of 64 and 2048, FlowANN achieves average throughput equal to 67.9% and 85.4% of CAGRA. This confirms the efficiency of its tiered graph and edge fetching pipeline. Its performance nearly matches CAGRA's at large batches, which enhances GPU utilization and extends per-step execution time, providing a wider window for async transfers.

Performance without GPU edge cache. We measure FlowANN's throughput when all edges reside on the CPU (i.e., w/o tiering). As Fig. 19 shows, the throughput of FlowANN

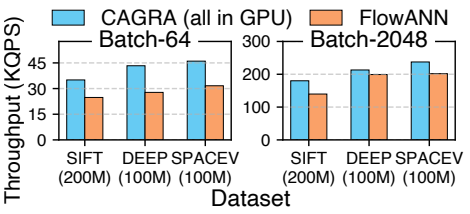


Figure 18: FlowANN's throughput on 100M and 200M datasets and the upper bound.

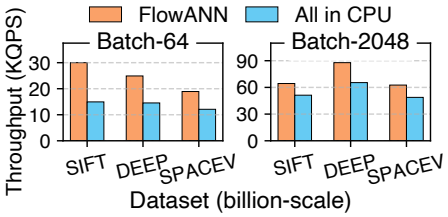


Figure 19: Throughput of FlowANN w/o GPU cache (representing lower bound).

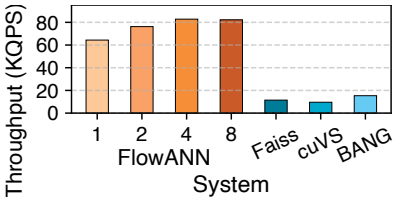


Figure 20: Throughput under different search widths (1-8). *SIFT1B dataset.*

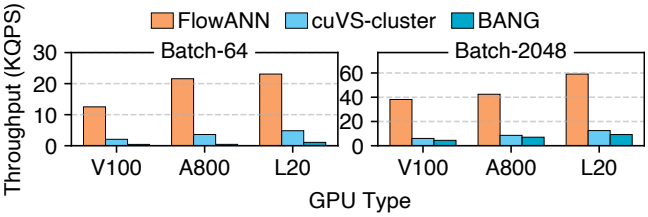


Figure 21: Throughput across GPUs.

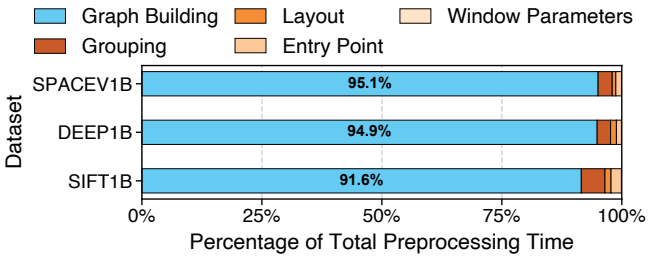


Figure 22: Preprocessing time breakdown of FlowANN.

w/o tiering reaches 58.4% (batch size 64) and 78.1% (batch size 2048) of standard FlowANN. Larger batches narrow this gap, as they better overlap transfer overhead. It still outperforms the baselines (Fig. 13) by 1.5–50.6× (batch 64) and 4.5–12.6× (batch 2048). This shows that even without GPU edge caching, FlowANN's optimizations (e.g., xCopier, adaptive synchronization) still yield non-trivial performance gains.

7.4.2 Generality for Algorithms and Hardware

Applicability to best-first search variants. A common variant of best-first search is to expand multiple nodes per step [8] (i.e., search width > 1) to improve efficiency. We evaluate FlowANN's throughput with different search widths (1–8). As Fig. 20 shows, FlowANN outperforms the baseline across all search widths. Larger search widths allow FlowANN to find high-quality neighbors faster, reducing total search steps and improving throughput up to a width of 4. Beyond that, throughput declines due to higher per-step computation cost and diminishing improvements in result quality.

Applicability to other GPU architectures. Besides NVIDIA H20, we also evaluate FlowANN on V100, A800, and L20, covering mainstream GPUs [74], as Table 1 details. As Fig. 21 shows, under different architectures and memory capabilities, FlowANN consistently outperforms the baselines (by 4.97–29.4×), demonstrating its generality. For small batches, its gains over baselines grow with larger memory. For large batches, the gains increase with GPU compute power, as longer per-step computation time already overlaps data transfer, amplifying the impact of computational capability.

7.4.3 Preprocessing Cost

FlowANN's additional preprocessing steps beyond graph building (offline) include graph tiering, entry point selection, and parameter acquisition for window estimation. Fig. 22 shows the proportion of each step in the total preprocessing time. The results indicate that the extra overhead introduced by these new steps is relatively small: only 4.9%, 5.1%, and 8.4% on the three datasets, respectively.

8 Discussion

Offloading strategy and future adaptability. FlowANN primarily offloads graph connectivity because it constitutes the main memory bottleneck in current billion-scale workloads (e.g., SIFT1B). This rationale aligns with mainstream tiered ANNS designs [8, 11]. To accommodate the massive high-dimensional vectors typical of emerging LLM workloads, FlowANN's tiering design can naturally support offloading both graph edges and data vectors. In such a case, the neighbor's data vector is fetched simultaneously during the asynchronous neighbor fetch phase.

Compatibility with dynamic updates. While FlowANN mainly focuses on optimizing the search process, its preprocessing mechanisms do not inherently hinder dynamic graph updates. FlowANN can readily integrate with existing online or offline graph update methods. When newly inserted points are added to the graph, the system does not require a complete, global re-grouping. For example, a newly inserted node can be grouped by simply calculating its neighbors' weights and joining the valid group with the highest total weight. Only when the number of groups exceeds a certain threshold, a global re-grouping is required.

Compatibility with variant graph structures. FlowANN is compatible with variant graph structures that support best-first search. The core optimization of FlowANN is disentangling

step-level dependencies to exploit the time windows between node discovery and expansion. This mechanism is rooted in the iterative nature of best-first search. Thus, FlowANN does not require an exact KNN graph structure and can seamlessly adapt to diverse proximity graphs.

Cost-effectiveness of GPU for ANNS. Due to the massive parallelism of GPUs, GPU-based ANNS can yield superior cost-efficiency (i.e., QPS/$) compared to CPU-based ones [19, 22]. This economic advantage has driven the adoption of GPU-based ANNS in industry [75], further amplified by hardware repurposing. While older or lower-tier GPUs (e.g., NVIDIA V100) are no longer suitable for demanding LLM workloads, they remain highly efficient for ANNS and can be repurposed for cost-effective deployment.

Scalability of xCopier with MMIO-based data transfer. xCopier utilizes MMIO-based data transfer (§ 6.1), which maintains its performance advantage over DMA even under high concurrency (i.e., large batch sizes). For instance, with batch size 10 k, the total data transferred in one step is merely ~1 MB, allowing MMIO to maintain its advantage (Fig. 12). As a result, xCopier does not encounter bandwidth bottlenecks even when all SMs are fully saturated. xCopier will fall back to DMA-based copying when MMIO is unavailable.

Necessity of deferred expansion. A seemingly straightforward alternative to deferred expansion is to rely purely on hardware yielding (i.e., swapping to other queries during data transfers). However, yielding alone is insufficient to hide latency due to GPU resource limits and batch size constraints. Because GPU computational resources (e.g., shared memory and registers) strictly bound the maximum number of queries simultaneously residing on SMs, yielding during data transfers quickly stalls all active queries, leaving the GPU idle. Furthermore, pure yielding requires massive batch sizes to successfully interleave queries, rendering it ineffective for latency-sensitive online serving. FlowANN's deferred expansion overcomes both limitations by overlapping transfer overhead within each individual query, ensuring high performance without requiring massive concurrency.

9 Related Work

SSD-based ANNS. Prior efforts [76–78] employ pipelining for fetching data from SSD. Although FlashANNS [42] a one-step relaxation in the search process, it still maintains the strict step-level dependency. PipeANN [11] relaxes compute-I/O order to better saturate SSD I/O bandwidth. However, it treats all nodes' expansions equally, which may cause sub-optimal search paths when some important expansions are delayed. Additionally, unlike the CPU-SSD setting, the compute-I/O gap for GPU-CPU is relatively narrow (similar latency), which limits the degree of I/O parallelism that can be exploited. FlowANN accounts for the variability in the discovery-expansion window across nodes, keeping a near-optimal search path with minimal computational overhead, improving both latency and throughput.

Moreover, we believe that SSD is unsuitable for GPU-based graph ANNS' second-tier storage: (1) SSD's latency (~100 μs [79]) far exceeds graph search's per-step execution time (6–14 μs), and is hard to be overlapped, leading to low GPU utilization; (2) GPU servers typically have TB-level memory [80–82], which is sufficient to hold billion-scale datasets and graphs.

Distributed ANNS. Distributed ANNS systems [43, 83, 84] focus on horizontal scaling to support ANNS on multiple nodes. They are complementary to FlowANN, as FlowANN primarily focuses on maximizing the search capability of a single GPU. Additionally, to support the multi-node scenario, FlowANN can be deployed as multiple replicas.

GPU ANNS' computational optimizations. Previous works explore parallelism [21, 51, 85, 86], quantization [22, 55], and multi-GPU collaboration [23, 43]. They can be (or already have been) integrated with FlowANN.

Graph-based ANNS on CPU. Many works optimize graph-based ANNS on the CPU, such as parallelism [87, 88], enhancing graph structures [9, 10, 57, 89–92], improving quantization methods [12, 31, 93, 94], and tuning parameters [46, 95–99]. These efforts are complementary to FlowANN.

10 Conclusion

We present FlowANN, a graph ANNS system enabling efficient billion-scale vector search on a single GPU. FlowANN is built on the key insight that the rigid step-level dependency in graph search can be disentangled into a fine-grained node-level dependency. Guided by this insight, FlowANN adopts a tiered graph structure that offloads the edges with sufficient discovery-expansion windows to the CPU. Leveraging these windows, FlowANN defers the discovery of some nodes and overlaps their edge fetching with GPU computation. Evaluations on billion-scale datasets show that FlowANN outperforms state-of-the-art systems by 4.08–45.7× on average (up to 172.6×) without compromising search accuracy.

Acknowledgments

We thank our shepherd and the anonymous reviewers for their insightful comments and feedback. We sincerely thank Haibo Chen for his guidance throughout this work, and Jingwei Xu and Weidong Zhang for their helpful suggestions. This work is supported in part by the Fundamental and Interdisciplinary Disciplines Breakthrough Plan of the Ministry of Education of China (JYB2025XDXM113) and the National Natural Science Foundation of China (No. 62132014, No.62302300). Corresponding authors: Mingkai Dong (mingkaidong@sjtu.edu.cn) and Dong Du (dd_nirvana@sjtu.edu.cn).

References

[1] Zhengding Hu, Vibha Murthy, Zaifeng Pan, Wanlu Li, Xiaoyi Fang, Yufei Ding, and Yuke Wang. HedraRAG: Co-optimizing generation and retrieval for heterogeneous RAG workflows. In *Proceedings of the ACM SIGOPS 31st Symposium on Operating Systems Principles*, SOSP '25, page 623–638, New York, NY, USA, 2025. Association for Computing Machinery.

[2] Siddhant Ray, Rui Pan, Zhuohan Gu, Kuntai Du, Shaoting Feng, Ganesh Ananthanarayanan, Ravi Netravali, and Junchen Jiang. METIS: Fast quality-aware RAG systems with configuration adaptation. In *Proceedings of the ACM SIGOPS 31st Symposium on Operating Systems Principles*, SOSP '25, page 606–622, New York, NY, USA, 2025. Association for Computing Machinery.

[3] Jie Li, Haifeng Liu, Chuanghua Gui, Jianyu Chen, Zhenyuan Ni, Ning Wang, and Yuan Chen. The design and implementation of a real time visual search system on JD e-commerce platform. In *Proceedings of the 19th International Middleware Conference Industry*, Middleware '18, page 9–16, New York, NY, USA, 2018. Association for Computing Machinery.

[4] Sen Li, Fuyu Lv, Taiwei Jin, Guli Lin, Keping Yang, Xiaoyi Zeng, Xiao-Ming Wu, and Qianli Ma. Embedding-based product retrieval in Taobao search. In *Proceedings of the 27th ACM SIGKDD Conference on Knowledge Discovery & Data Mining*, KDD '21, page 3181–3189, New York, NY, USA, 2021. Association for Computing Machinery.

[5] Liang Zheng, Liyue Shen, Lu Tian, Shengjin Wang, Jingdong Wang, and Qi Tian. Scalable person re-identification: A benchmark. In *2015 IEEE International Conference on Computer Vision (ICCV)*, pages 1116–1124, 2015.

[6] Di Liu, Meng Chen, Baotong Lu, Huiqiang Jiang, Zhenhua Han, Qianxi Zhang, Qi Chen, Chengruidong Zhang, Bailu Ding, Kai Zhang, Chen Chen, Fan Yang, Yuqing Yang, and Lili Qiu. Retrievalattention: Accelerating long-context LLM inference via vector retrieval. *arXiv preprint arXiv:2409.10516*, 2024.

[7] Fu Bang. GPTCache: An open-source semantic cache for LLM applications enabling faster answers and cost savings. In Liling Tan, Dmitrijs Milajevs, Geeticka Chauhan, Jeremy Gwinnup, and Elijah Rippeth, editors, *Proceedings of the 3rd Workshop for Natural Language Processing Open Source Software (NLP-OSS 2023)*, pages 212–218, Singapore, December 2023. Association for Computational Linguistics.

[8] Suhas Jayaram Subramanya, Devvrit, Rohan Kadekodi, Ravishankar Krishaswamy, and Harsha Vardhan Simhadri. *DiskANN: fast accurate billion-point nearest neighbor search on a single node*. Curran Associates Inc., Red Hook, NY, USA, 2019.

[9] Cong Fu, Chao Xiang, Changxu Wang, and Deng Cai. Fast approximate nearest neighbor search with the navigating spreading-out graph. *Proc. VLDB Endow.*, 12(5):461–474, January 2019.

[10] Yu. A. Malkov and D. A. Yashunin. Efficient and robust approximate nearest neighbor search using hierarchical navigable small world graphs. *arXiv preprint arXiv:1603.09320*, 2018.

[11] Hao Guo and Youyou Lu. Achieving low-latency graph-based vector search via aligning best-first search algorithm with SSD. In *Proceedings of the 19th USENIX Conference on Operating Systems Design and Implementation*, OSDI '25, USA, 2025. USENIX Association.

[12] Yutong Gou, Jianyang Gao, Yuexuan Xu, and Cheng Long. SymphonyQG: Towards symphonious integration of quantization and graph for approximate nearest neighbor search. *Proc. ACM Manag. Data*, 3(1), February 2025.

[13] Shulin Zeng, Zhenhua Zhu, Jun Liu, Haoyu Zhang, Guohao Dai, Zixuan Zhou, Shuangchen Li, Xuefei Ning, Yuan Xie, Huazhong Yang, and Yu Wang. DF-GAS: a distributed FPGA-as-a-Service architecture towards billion-scale graph-based approximate nearest neighbor search. In *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '23, page 283–296, New York, NY, USA, 2023. Association for Computing Machinery.

[14] Zhenhua Zhu, Jun Liu, Guohao Dai, Shulin Zeng, Bing Li, Huazhong Yang, and Yu Wang. Processing-in-hierarchical-memory architecture for billion-scale approximate nearest neighbor search. In *Proceedings of the 60th Annual ACM/IEEE Design Automation Conference*, DAC '23, page 1–6. IEEE Press, 2025.

[15] Ji-Hoon Kim, Yeo-Reum Park, Jaeyoung Do, Soo-Young Ji, and Joo-Young Kim. Accelerating large-scale graph-based nearest neighbor search on a computational storage platform. *IEEE Transactions on Computers*, 72(1):278–290, 2023.

[16] Bing Tian, Haikun Liu, Zhuohui Duan, Xiaofei Liao, Hai Jin, and Yu Zhang. Scalable billion-point approximate nearest neighbor search using SmartSSDs. In *Proceedings of the 2024 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC'24, USA, 2024. USENIX Association.

[17] Yitu Wang, Shiyu Li, Qilin Zheng, Linghao Song, Zongwang Li, Andrew Chang, Hai "Helen" Li, and Yiran Chen. NDSearch: Accelerating graph-traversal-based approximate nearest neighbor search through near data processing. In *Proceedings of the 51st Annual International Symposium on Computer Architecture*, ISCA '24, page 368–381. IEEE Press, 2025.

[18] Junhyeok Jang, Hanjin Choi, Hanyeoreum Bae, Seungjun Lee, Miryeong Kwon, and Myoungsoo Jung. CXL-ANNS: Software-Hardware collaborative memory disaggregation and computation for Billion-Scale approximate nearest neighbor search. In *2023 USENIX Annual Technical Conference (USENIX ATC 23)*, pages 585–600, Boston, MA, July 2023. USENIX Association.

[19] Zili Zhang, Fangyue Liu, Gang Huang, Xuanzhe Liu, and Xin Jin. Fast vector query processing for large datasets beyond GPU memory with reordered pipelining. In *Proceedings of the 21st USENIX Symposium on Networked Systems Design and Implementation*, NSDI'24, USA, 2024. USENIX Association.

[20] Karthik V., Saim Khan, Somesh Singh, Harsha Vardhan Simhadri, and Jyothi Vedurada. BANG: Billion-scale approximate nearest neighbor search using a single GPU. *arXiv preprint arXiv:2401.11324*, 2025.

[21] Yuntao Gui, Peiqi Yin, Xiao Yan, Chaorui Zhang, Weixi Zhang, and James Cheng. PilotANN: Memory-bounded GPU acceleration for vector search. *arXiv preprint arXiv:2503.21206*, 2025.

[22] Bing Tian, Haikun Liu, Yuhang Tang, Shihai Xiao, Zhuohui Duan, Xiaofei Liao, Hai Jin, Xuecang Zhang, Junhua Zhu, and Yu Zhang. Towards high-throughput and low-latency billion-scale vector search via CPU/GPU collaborative filtering and re-ranking. In *Proceedings of the 23rd USENIX Conference on File and Storage Technologies*, FAST '25, USA, 2025. USENIX Association.

[23] Sukjin Kim, Seongyeon Park, Si Ung Noh, Junguk Hong, Taehee Kwon, Hunseong Lim, and Jinho Lee. PathWeaver: a high-throughput multi-GPU system for graph-based approximate nearest neighbor search. In *Proceedings of the 2025 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC '25, USA, 2025. USENIX Association.

[24] Jeff Johnson, Matthijs Douze, and Hervé Jégou. Faiss: A library for efficient similarity search and clustering of dense vectors. https://github.com/facebookresearch/faiss, 2017.

[25] Jianguo Wang, Xiaomeng Yi, Rentong Guo, Hai Jin, Peng Xu, Shengjun Li, Xiangyu Wang, Xiangzhou Guo, Chengming Li, Xiaohai Xu, Kun Yu, Yuxing Yuan, Yinghao Zou, Jiquan Long, Yudong Cai, Zhenxiang Li, Zhifeng Zhang, Yihua Mo, Jun Gu, Ruiyi Jiang, Yi Wei, and Charles Xie. Milvus: A purpose-built vector data management system. In *Proceedings of the 2021 International Conference on Management of Data*, SIGMOD '21, page 2614–2627, New York, NY, USA, 2021. Association for Computing Machinery.

[26] Jeff Johnson, Matthijs Douze, and Hervé Jégou. Billion-scale similarity search with GPUs. *arXiv preprint arXiv:1702.08734*, 2017.

[27] RAPIDS AI. cuVS: a library for vector search and clustering on the GPU. https://github.com/rapidsai/cuvs, 2025.

[28] Meituan (the largest online food delivery company in the world). Practice of meituan's GPU-based vector retrieval system. https://tech.meituan.com/2024/04/11/gpu-vector-retrieval-system-practice.html, 2024.

[29] Zilliz cloud named a leader in the forrester wave. https://zilliz.com, 2025.

[30] Chuangxian Wei, Bin Wu, Sheng Wang, Renjie Lou, Chaoqun Zhan, Feifei Li, and Yuanzhe Cai. AnalyticDB-V: a hybrid analytical engine towards query fusion for structured and unstructured data. *Proc. VLDB Endow.*, 13(12):3152–3165, August 2020.

[31] Jianyang Gao and Cheng Long. RaBitQ: Quantizing high-dimensional vectors with a theoretical error bound for approximate nearest neighbor search. *Proc. ACM Manag. Data*, 2(3), May 2024.

[32] Herve Jégou, Matthijs Douze, and Cordelia Schmid. Product quantization for nearest neighbor search. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 33(1):117–128, 2011.

[33] NeurIPS'21. Billion-scale approximate nearest neighbor search challenge: NeurIPS'21 competition track. https://big-ann-benchmarks.com/neurips21.html, 2021.

[34] Artem Babenko Yandex and Victor Lempitsky. Efficient indexing of billion-scale datasets of deep descriptors. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 2055–2063, 2016.

[35] Anton Voronov, Denis Kuznedelev, Mikhail Khoroshikh, Valentin Khrulkov, and Dmitry Baranchuk. Switti: Designing scale-wise transformers for text-to-image synthesis. *arXiv preprint arXiv:2412.01819*, 2025.

[36] Harsha Vardhan Simhadri, George Williams, Martin Aumüller, Matthijs Douze, Artem Babenko, Dmitry Baranchuk, Qi Chen, Lucas Hosseini, Ravishankar Krishnaswamy, Gopal Srinivasa, Suhas Jayaram Subramanya, and Jingdong Wang. Results of the NeurIPS'21 challenge on billion-scale approximate nearest neighbor search, 2022.

[37] NVIDIA. Nvidia CUDA C++ programming guide. https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html, 2025.

[38] Lars Gottesbüren, Tobias Heuer, Peter Sanders, Christian Schulz, and Daniel Seemaier. Deep Multilevel Graph Partitioning. In Petra Mutzel, Rasmus Pagh, and Grzegorz Herman, editors, *29th Annual European Symposium on Algorithms (ESA 2021)*, volume 204 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 48:1–48:17, Dagstuhl, Germany, 2021. Schloss Dagstuhl – Leibniz-Zentrum für Informatik.

[39] The TEXMEX Project Team. Corpus-TEXMEX: Datasets for approximate nearest neighbor search. http://corpus-texmex.irisa.fr/, 2011.

[40] Artem Babenko Yandex and Victor Lempitsky. Efficient indexing of billion-scale datasets of deep descriptors. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 2055–2063, 2016.

[41] Microsoft. SPACEV1B: A billion-scale vector dataset for text descriptors. https://github.com/microsoft/SPTAG/tree/master/datasets/SPACEV1B, 2023.

[42] Yang Xiao, Mo Sun, Ziyu Song, Bing Tian, Jie Zhang, Jie Sun, and Zeke Wang. Breaking the storage-compute bottleneck in billion-scale ANNS: A GPU-driven asynchronous I/O framework. *arXiv preprint arXiv:2507.10070*, 2025. Accepted by the 2026 ACM SIGMOD/PODS Conference (SIGMOD 2026).

[43] Fabian Groh, Lukas Ruppert, Patrick Wieschollek, and Hendrik P. A. Lensch. GGNN: Graph-based GPU nearest neighbor search. *IEEE Transactions on Big Data*, 9(1):267–279, 2023.

[44] Aristides Gionis, Piotr Indyk, and Rajeev Motwani. Similarity search in high dimensions via hashing. In *Proceedings of the 25th International Conference on Very Large Data Bases*, VLDB '99, page 518–529, San Francisco, CA, USA, 1999. Morgan Kaufmann Publishers Inc.

[45] Qi Chen, Bing Zhao, Haidong Wang, Mingqin Li, Chuanjie Liu, Zengzhong Li, Mao Yang, and Jingdong Wang. SPANN: highly-efficient billion-scale approximate nearest neighbor search. In *Proceedings of the 35th International Conference on Neural Information Processing Systems*, NIPS '21, Red Hook, NY, USA, 2021. Curran Associates Inc.

[46] Jason Mohoney, Devesh Sarda, Mengze Tang, Shihabur Rahman Chowdhury, Anil Pacaci, Ihab F. Ilyas, Theodoros Rekatsinas, and Shivaram Venkataraman. Quake: adaptive indexing for vector search. In *Proceedings of the 19th USENIX Conference on Operating Systems Design and Implementation*, OSDI '25, USA, 2025. USENIX Association.

[47] Ruiqi Guo, Philip Sun, Erik Lindgren, Quan Geng, David Simcha, Felix Chern, and Sanjiv Kumar. Accelerating large-scale inference with anisotropic vector quantization. In *Proceedings of the 37th International Conference on Machine Learning*, ICML'20. JMLR.org, 2020.

[48] Philip Sun, David Simcha, Dave Dopson, Ruiqi Guo, and Sanjiv Kumar. SOAR: improved indexing for approximate nearest neighbor search. In *Proceedings of the 37th International Conference on Neural Information Processing Systems*, NIPS '23, Red Hook, NY, USA, 2023. Curran Associates Inc.

[49] Yuming Xu, Hengyu Liang, Jin Li, Shuotao Xu, Qi Chen, Qianxi Zhang, Cheng Li, Ziyue Yang, Fan Yang, Yuqing Yang, Peng Cheng, and Mao Yang. SPFresh: Incremental in-place update for billion-scale vector search. In *Proceedings of the 29th Symposium on Operating Systems Principles*, SOSP '23, page 545–561, New York, NY, USA, 2023. Association for Computing Machinery.

[50] Junjie Qi, Gergely Szilvasy, Michael Norris, and Vishal Gandhi. Accelerating GPU indexes in faiss with NVIDIA cuVS. *Engineering at Meta*, May 2025.

[51] Hiroyuki Ootomo, Akira Naruse, Corey Nolet, Ray Wang, Tamas Feher, and Yong Wang. CAGRA: Highly parallel graph construction and approximate nearest neighbor search for GPUs. *arXiv preprint arXiv:2308.15136*, 2024.

[52] Jingrong Zhang, Akira Naruse, Xipeng Li, and Yong Wang. Parallel top-k algorithms on GPU: A comprehensive study and new methods. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '23, New York, NY, USA, 2023. Association for Computing Machinery.

[53] Qianxi Zhang, Shuotao Xu, Qi Chen, Guoxin Sui, Jiadong Xie, Zhizhen Cai, Yaoqi Chen, Yinxuan He, Yuqing Yang, Fan Yang, Mao Yang, and Lidong Zhou. VBASE: Unifying online vector similarity search and relational queries via relaxed monotonicity. In *17th*

USENIX Symposium on Operating Systems Design and Implementation (OSDI 23), pages 377–395, Boston, MA, July 2023. USENIX Association.

[54] Wei Dong, Charikar Moses, and Kai Li. Efficient k-nearest neighbor graph construction for generic similarity measures. In *Proceedings of the 20th International Conference on World Wide Web*, WWW '11, page 577–586, New York, NY, USA, 2011. Association for Computing Machinery.

[55] Zihan Liu, Wentao Ni, Jingwen Leng, Yu Feng, Cong Guo, Quan Chen, Chao Li, Minyi Guo, and Yuhao Zhu. JUNO: Optimizing high-dimensional approximate nearest neighbour search with sparsity-aware algorithm and ray-tracing core mapping. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, ASPLOS '24, page 549–565, New York, NY, USA, 2024. Association for Computing Machinery.

[56] Hao Guo and Youyou Lu. OdinANN: Direct insert for consistently stable performance in billion-scale graph-based vector search. In *24th USENIX Conference on File and Storage Technologies (FAST 26)*, Santa Clara, CA, 2026. USENIX Association.

[57] Xi Zhao, Yao Tian, Kai Huang, Bolong Zheng, and Xiaofang Zhou. Towards efficient index construction and approximate nearest neighbor search in high-dimensional spaces. *Proc. VLDB Endow.*, 16(8):1979–1991, April 2023.

[58] Cong Fu, Changxu Wang, and Deng Cai. High dimensional similarity search with satellite system graph: Efficiency, scalability, and unindexed query compatibility. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 44(8):4139–4150, 2022.

[59] Shuo Yang, Jiadong Xie, Yingfan Liu, Jeffrey Xu Yu, Xiyue Gao, Qianru Wang, Yanguo Peng, and Jiangtao Cui. Revisiting the index construction of proximity graph-based approximate nearest neighbor search. *Proc. VLDB Endow.*, 18(6):1825–1838, February 2025.

[60] Laurent Hyafil and Ronald L. Rivest. Graph partitioning and constructing optimal decision trees are polynomial complete problems. Rapport de Recherche 33, IRIA – Laboratoire de Recherche en Informatique et Automatique, 1973.

[61] M. R. Garey, D. S. Johnson, and L. Stockmeyer. Some simplified NP-complete problems. In *Proceedings of the Sixth Annual ACM Symposium on Theory of Computing*, STOC '74, page 47–63, New York, NY, USA, 1974. Association for Computing Machinery.

[62] Usha Nandini Raghavan, Réka Albert, and Soundar Kumara. Near linear time algorithm to detect community structures in large-scale networks. *Physical Review E*, 76(3), September 2007.

[63] George Karypis and Vipin Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM Journal on Scientific Computing*, 20(1):359–392, 1998.

[64] Taeyoon Kim, ChanHo Park, Mansur Mukimbekov, Heelim Hong, Minseok Kim, Ze Jin, Changdae Kim, Ji-Yong Shin, and Myeongjae Jeon. FusionFlow: Accelerating data preprocessing for machine learning with CPU-GPU cooperation. *Proc. VLDB Endow.*, 17(4):863–876, December 2023.

[65] NVIDIA. Nvidia GDRCopy. https://github.com/NVIDIA/gdrcopy, 2025.

[66] AMD. BAR configuration for AMD GPUs. https://rocm.docs.amd.com/en/latest/how-to/Bar-Memory.html, 2025.

[67] Jingkai He, Yunpeng Dong, Dong Du, Mo Zou, Zhitai Yu, Yuxin Ren, Ning Jia, Yubin Xia, and Haibo Chen. How to copy memory? coordinated asynchronous copy as a first-class OS service. In *Proceedings of the ACM SIGOPS 31st Symposium on Operating Systems Principles*, SOSP '25, page 1062–1081, New York, NY, USA, 2025. Association for Computing Machinery.

[68] Zheng Wang, Anna Cai, Xinfeng Xie, Zaifeng Pan, Yue Guan, Weiwei Chu, Jie Wang, Shikai Li, Jianyu Huang, Chris Cai, Yuchen Hao, and Yufei Ding. WLB-LLM: Workload-balanced 4D parallelism for large language model training. In *19th USENIX Symposium on Operating Systems Design and Implementation (OSDI 25)*, pages 785–801, Boston, MA, USA, July 2025. USENIX Association.

[69] Yeonhong Park, Jake Hyun, Hojoon Kim, and Jae W. Lee. DecDEC: A systems approach to advancing low-bit LLM quantization. In *19th USENIX Symposium on Operating Systems Design and Implementation (OSDI 25)*, pages 803–819, Boston, MA, USA, July 2025. USENIX Association.

[70] Shiwei Gao, Qing Wang, Shaoxun Zeng, Youyou Lu, and Jiwu Shu. Weaver: Efficient multi-LLM serving with attention offloading. In *2025 USENIX Annual Technical Conference (USENIX ATC 25)*, pages 587–595, Boston, MA, USA, July 2025. USENIX Association.

[71] James D. Evans. *Straightforward statistics for the behavioral sciences*. Brooks/Cole Publishing Company, Pacific Grove, Calif., 1996.

[72] National Library of Medicine. As a rule of thumb, correlation coefficients greater than 0.7 or less than -0.7 are considered strong. https://www.nlm.nih.gov/oet/ed/stats/02-300.html, 2025.

[73] Chaoji Zuo, Miao Qiao, Wenchao Zhou, Feifei Li, and Dong Deng. SeRF: Segment graph for range-filtering approximate nearest neighbor search. *Proc. ACM Manag. Data*, 2(1), March 2024.

[74] NVIDIA Developer. CUDA GPU Compute Capability. https://developer.nvidia.com/cuda-gpus, 2025.

[75] Karthik Bharathy, Shasank Chavan, Ikroop Dhillon, and Manas Singh. Oracle ai database + nvidia collaboration advances enterprise ai at nvidia gtc 2026, March 2026. Accessed: 2026-04-15.

[76] Mengzhao Wang, Weizhi Xu, Xiaomeng Yi, Songlin Wu, Zhangyang Peng, Xiangyu Ke, Yunjun Gao, Xiaoliang Xu, Rentong Guo, and Charles Xie. Starling: An I/O-efficient disk-resident graph index framework for high-dimensional vector similarity search on data segment. *Proc. ACM Manag. Data*, 2(1), March 2024.

[77] Jiongkang Ni, Xiaoliang Xu, Yuxiang Wang, Can Li, Jiajie Yao, Shihai Xiao, and Xuecang Zhang. DiskANN++: Efficient page-based search over isomorphic mapped graph index using query-sensitivity entry vertex. *arXiv preprint arXiv:2310.00402*, 2023.

[78] Joobo Shim, Jaewon Oh, Hongchan Roh, Jaeyoung Do, and Sang-Won Lee. Turbocharging vector databases using modern SSDs. *Proc. VLDB Endow.*, 18(11):4710–4722, July 2025.

[79] Zaid Qureshi, Vikram Sharma Mailthody, Isaac Gelado, Seungwon Min, Amna Masood, Jeongmin Park, Jinjun Xiong, C. J. Newburn, Dmitri Vainbrand, I-Hsin Chung, Michael Garland, William Dally, and Wen-mei Hwu. GPU-initiated on-demand high-throughput storage access in the BaM system architecture. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, ASPLOS 2023, page 325–339, New York, NY, USA, 2023. Association for Computing Machinery.

[80] NVIDIA Corporation. NVIDIA DGX A100 Datasheet. https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/nvidia-dgx-a100-datasheet.pdf, 2020.

[81] Amazon Web Services, Inc. Amazon EC2 P4d Instances. https://aws.amazon.com/ec2/instance-types/p4/, 2025.

[82] Microsoft Corporation. NDm A100 v4-series – Azure Virtual Machines. https://learn.microsoft.com/en-us/azure/virtual-machines/sizes/gpu-accelerated/ndma100v4-series?tabs=sizebasic, 2025.

[83] Philip Adams, Menghao Li, Shi Zhang, Li Tan, Qi Chen, Mingqin Li, Zengzhong Li, Knut Risvik, and Harsha Vardhan Simhadri. Distributedann: Efficient scaling of a single diskann graph across thousands of computers, 2025.

[84] Yuming Xu, Qianxi Zhang, Qi Chen, Baotong Lu, Menghao Li, Philip Adams, Mingqin Li, Zengzhong Li, Jing Liu, Cheng Li, and Fan Yang. Scalable distributed vector search via accuracy preserving index construction, 2025.

[85] Weijie Zhao, Shulong Tan, and Ping Li. SONG: Approximate nearest neighbor search on GPU. In *2020 IEEE 36th International Conference on Data Engineering (ICDE)*, pages 1033–1044, 2020.

[86] Yuanhang Yu, Dong Wen, Ying Zhang, Lu Qin, Wenjie Zhang, and Xuemin Lin. GPU-accelerated proximity graph approximate nearest neighbor search and construction. In *2022 IEEE 38th International Conference on Data Engineering (ICDE)*, pages 552–564, 2022.

[87] Magdalen Dobson Manohar, Zheqi Shen, Guy Blelloch, Laxman Dhulipala, Yan Gu, Harsha Vardhan Simhadri, and Yihan Sun. ParlayANN: Scalable and deterministic parallel graph-based approximate nearest neighbor search algorithms. In *Proceedings of the 29th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming*, PPoPP '24, page 270–285, New York, NY, USA, 2024. Association for Computing Machinery.

[88] Zhen Peng, Minjia Zhang, Kai Li, Ruoming Jin, and Bin Ren. iQAN: Fast and accurate vector search with efficient intra-query parallelism on multi-core architectures. In *Proceedings of the 28th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming*, PPoPP '23, page 313–328, New York, NY, USA, 2023. Association for Computing Machinery.

[89] Ziqi Yin, Jianyang Gao, Pasquale Balsebre, Gao Cong, and Cheng Long. DEG: Efficient hybrid vector search using the dynamic edge navigation graph. *Proc. ACM Manag. Data*, 3(1), February 2025.

[90] Zengyang Gong, Yuxiang Zeng, and Lei Chen. Accelerating approximate nearest neighbor search in hierarchical graphs: Efficient level navigation with shortcuts. *Proc. VLDB Endow.*, 18(10):3518–3530, June 2025.

[91] Benjamin Coleman, Santiago Segarra, Alex Smola, and Anshumali Shrivastava. Graph reordering for cache-efficient near neighbor search. In *Proceedings of the 36th International Conference on Neural Information Processing Systems*, NIPS '22, Red Hook, NY, USA, 2022. Curran Associates Inc.

[92] Lars Gottesbüren, Laxman Dhulipala, Rajesh Jayaram, and Jakub Łącki. Unleashing graph partitioning for large-scale nearest neighbor search. *Proc. VLDB Endow.*, 18(6):1649–1662, February 2025.

[93] Jianyang Gao, Yutong Gou, Yuexuan Xu, Yongyi Yang, Cheng Long, and Raymond Chi-Wing Wong. Practical and asymptotically optimal quantization of high-dimensional vectors in euclidean space for approximate nearest neighbor search. *Proc. ACM Manag. Data*, 3(3), June 2025.

[94] Ziming Yuan, Lei Dai, Wen Li, Jie Zhang, Shengwen Liang, Ying Wang, Cheng Liu, Huawei Li, Xiaowei Li, Jiafeng Guo, Peng Wang, Renhai Chen, and Gong Zhang. NeuVSA: A unified and efficient accelerator for neural vector search. In *2025 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 790–805, 2025.

[95] Conglong Li, Minjia Zhang, David G. Andersen, and Yuxiong He. Improving approximate nearest neighbor search through learned adaptive early termination. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, SIGMOD '20, page 2539–2554, New York, NY, USA, 2020. Association for Computing Machinery.

[96] Vo Ngoc Anh, Owen de Kretser, and Alistair Moffat. Vector-space ranking with effective early termination. In *Proceedings of the 24th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, SIGIR '01, page 35–42, New York, NY, USA, 2001. Association for Computing Machinery.

[97] Zili Zhang, Chao Jin, Linpeng Tang, Xuanzhe Liu, and Xin Jin. Fast, approximate vector queries on very large unstructured datasets. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 995–1011, Boston, MA, April 2023. USENIX Association.

[98] Manos Chatzakis, Yannis Papakonstantinou, and Themis Palpanas. DARTH: Declarative recall through early termination for approximate nearest neighbor search. *Proc. ACM Manag. Data*, 3(4), September 2025.

[99] Minjia Zhang, Wenhan Wang, and Yuxiong He. GraSP: Optimizing graph-based nearest neighbor search with subgraph sampling and pruning. In *Proceedings of the Fifteenth ACM International Conference on Web Search and Data Mining*, WSDM '22, page 1395–1405, New York, NY, USA, 2022. Association for Computing Machinery.

A Proof of Correctness and Convergence

We provide a theoretical analysis regarding: (1) **Search correctness**, proving that the asynchronous fetching strategy does not compromise the termination and accuracy of the best-first search; (2) **Convergence speed lower bound**, proving that the worst-case search steps are strictly bounded, ensuring the system efficiently converges to the target; and (3) **Search budget bound**, proving that the required candidate pool size to guarantee accuracy remains within practical limits.

A.1 Search Correctness

The core concern is whether deferring the discovery of host-side neighbors prevents the algorithm from finding the true nearest neighbors. We model the search process as a traversal on a connected graph $G = (V, E)$.

Lemma 1 (Lossless Candidate Availability)**.** *In FlowANN, for any node u visited at step t, its full neighbor set $N(u)$ is eventually added to the candidate pool Q. Let $N(u) = N_{GPU}(u) \cup N_{Host}(u)$.*

- *$N_{GPU}(u)$ is added to Q at step t.*
- *$N_{Host}(u)$ is added to Q at step $t+\delta$, where δ represents the asynchronous fetch latency (in steps).*

Since δ is finite ($\delta < \infty$), no candidate is permanently discarded.

Theorem 1 (Top-k Convergence Consistency)**.** *Given a monotonic distance measure and a connected graph, for any query q, the deferred discovery mechanism ensures that the final candidate set $\mathcal{R}$ converges to the same set of top-k nearest neighbors as the synchronous baseline (i.e., the original best-first search), provided that the search budget is sufficient.*

Proof. Let $\mathcal{C}_t$ denote the candidate pool at step t, maintaining the best candidates found so far, sorted by distance to q. The capacity of $\mathcal{C}_t$ is bounded by the search budget L ($L \geq k$). Let $d_L(t)$ be the distance of the worst candidate in $\mathcal{C}_t$ (i.e., the L-th nearest). If $|\mathcal{C}_t| < L$, we define $d_L(t) = \infty$.

In the synchronous baseline, a node v is added to $\mathcal{C}$ if $\mathrm{Dist}(q, v) < d_L(t)$.

In FlowANN, consider a potential top-k neighbor v^* residing on the host side. Due to the fetch latency, v^* arrives at step $t+\delta$ instead of t.

- **Persistence of entry condition:** Since v^* is a true top-k neighbor (or a node on the path to one), its distance $\mathrm{Dist}(q, v^*)$ is inherently small.
- **Insertion logic:** When v^* becomes available at $t+\delta$, *provided that the search budget is sufficient* (i.e., L is large enough), the system compares v^* against the current state $\mathcal{C}_{t+\delta}$. Even if the pool has evolved during the delay δ, since v^* is optimal (one of the true top-k) and L is sufficient, it holds that:

$$\mathrm{Dist}(q, v^*) < d_L(t+\delta) \tag{1}$$

  This inequality holds because a sufficient budget L ensures that the queue is not filled with L candidates *strictly better* than a true top-k neighbor (by definition of top-k).

Therefore, v^* is successfully inserted into $\mathcal{C}_{t+\delta}$ and correctly sorted. The delay δ only shifts the moment of insertion but does not prevent the node from displacing a suboptimal candidate. Thus, under the sufficient budget assumption, the final set $\mathcal{R}$ retains the same top-k accuracy as the strict synchronous execution. □

A.2 Bound of Search Convergence Speed

While Theorem 1 establishes the convergence to top-k neighbors under the assumption of a sufficient search budget, it is imperative to ensure that this required budget remains within practical limits. If asynchronous fetch latency were to cause the search path to deviate indefinitely, the system would require an unrealistically large budget to avoid premature termination. Therefore, to validate the practicality of the sufficient budget assumption, we must prove that the additional search cost induced by asynchronous fetching is strictly bounded.

We assume the goal is to limit the "detour" caused by the asynchronous fetch latency, and we define the search efficiency by bounding the total traversal steps.

Definitions:

- Let $\mathcal{P}_{base}$ be the search path of the strict synchronous baseline (best-first search). Let $|\mathcal{P}_{base}| = T$.
- Let $\mathcal{P}_{sys}$ be the search path of FlowANN.
- Let τ be the constant fetch latency (in steps) for host-resident edges.
- Let $W(e)$ be the discovery-expansion window for an edge $e = (u, v)$ in the best-first search, defined as $W(e) = step_{expand}(v) - step_{discover}(u)$.

Theorem 2 (Bounded Step Deviation)**.** *The number of steps in FlowANN, denoted as $|\mathcal{P}_{sys}|$, is bounded by:*

$$|\mathcal{P}_{sys}| \leq T + \sum_{i=1}^{K} \max(0, \tau - W(e_i)) \tag{2}$$

where $\{e_1, \ldots, e_K\}$ denotes the set of K offloaded edges (i.e., neighbors) encountered along the critical path $\mathcal{P}_{base}$.

Proof. Consider a critical node v on the optimal path $\mathcal{P}_{base}$ reached via edge $e = (u, v)$. We analyze the impact of fetching latency in three cases:

Case 1: $e \in E_{GPU}$ **(Cached Edge).** The node v is available immediately at step t. No delay is introduced. The deviation $\Delta = 0$.

Case 2: $e \in E_{Host}$ **and** $W(e) \geq \tau$ **(Successful Prediction).** Node u is expanded at step t, and the fetch request for v is issued. Due to the window property, the baseline algorithm would not have popped (expanded) v from Q until step $t' = t + W(e)$. In FlowANN, v arrives at Q at step $t + \tau$. Since $\tau \leq W(e)$, it follows that $t + \tau \leq t'$. Thus, v arrives *before* or *exactly when* it is needed by the best-first search. The availability of v matches the baseline's requirement, so $\Delta = 0$.

Case 3: $e \in E_{Host}$ **and** $W(e) < \tau$ **(Prediction Miss).** This is the only scenario causing a detour. The algorithm needs v at step $t + W(e)$, but v arrives at $t + \tau$. The system is forced to discover suboptimal nodes for a duration of $\Delta_{step} = \tau - W(e)$. Once the τ steps pass, v enters Q and corrects the path.

Conclusion: The total extra cost is the sum of these "waiting gaps" ($\tau - W(e)$). Since the fetch latency τ is effectively overlapped with computation in most steps (Cases 1 and 2), and the candidate pool ensures automatic path correction, the search complexity of FlowANN remains linear $O(T)$, providing a guaranteed lower bound on convergence speed comparable to the synchronous baseline. □

Quantitative Worst-Case Analysis: We now instantiate the bound to demonstrate the limits of search expansion under extreme conditions. We assume a catastrophic scenario where two worst-case conditions occur simultaneously: (1) **Zero cache hit**, implying that every node on the search path resides on the host and requires a fetch operation (i.e., $K = T$); and (2) **Zero discovery-expansion window**, where window prediction fails completely for every step, meaning data is needed immediately (i.e., $W(e_i) \approx 0$ for all i).

Substituting these conditions ($K = T, W = 0$) into Equation (2), we obtain the general upper bound:

$$|\mathcal{P}_{sys}|_{worst} \leq T + \sum_{i=1}^{T} (\tau - 0) = T + T \cdot \tau = T(1 + \tau) \quad (3)$$

In practice, xCopier's asynchronous fetch latency is ~8 μs (Figure 19, Section 7.3), which does not exceed the duration of two search steps (6–14 μs per step, Section 3). Therefore, we set the latency penalty $\tau \approx 2$ steps. Substituting this into the general formula yields:

$$|\mathcal{P}_{sys}|_{worst} \leq T(1 + 2) = 3T \quad (4)$$

This derivation explicitly quantifies that even in the absolute worst-case scenario—where no caching or window optimization works—the search complexity remains strictly linear ($O(T)$), bounded by a small constant factor of the baseline steps.

A.3 Bound of Search Budget

While the step bound established in Equation (2) guarantees time efficiency, it is equally critical to verify the search budget constraints (i.e., candidate pool capacity). A potential risk introduced by deferred discovery is *candidate overflow*: the intermediate nodes visited during the waiting period might populate the candidate pool with numerous suboptimal candidates. To rule out this possibility, we explicitly derive the upper bound of the required candidate pool size.

Theorem 3 (Bounded Candidate Pool Size)**.** *Let U_{base} be the worst-case upper bound of the candidate pool size for the synchronous baseline. The worst-case candidate pool capacity U_{sys} required by FlowANN is bounded by:*

$$U_{sys} \leq U_{base} \cdot (1 + \tau) \quad (5)$$

where τ is the latency factor defined in Theorem 2.

Proof. The maximum number of candidates stored in the pool is determined by the number of expanded nodes and the maximum graph degree. For the synchronous baseline, which executes T steps (where $T = |\mathcal{P}_{base}|$), the worst-case pool size is bounded by the cumulative number of added neighbors:

$$U_{base} \approx T \cdot D_{max} \quad (6)$$

where D_{max} represents the maximum node degree in graph G.

For FlowANN, Theorem 2 proves that the search process expands at most $T(1 + \tau)$ nodes in the worst case to recover the optimal path. Since the node expansion logic remains identical (adding neighbors of visited nodes), the required pool capacity scales linearly with the maximum number of visited steps:

$$U_{sys} \leq [T \cdot (1 + \tau)] \cdot D_{max} \quad (7)$$

By rearranging the terms, we can express the bound of FlowANN in terms of the baseline's bound:

$$U_{sys} \leq (T \cdot D_{max}) \cdot (1 + \tau) = U_{base}(1 + \tau) \quad (8)$$

Conclusion: This result proves that the space complexity of FlowANN (i.e., required search budget) remains in the same order of magnitude as the baseline. Given that $\tau \approx 2$ in our system, a modest linear increase in the candidate pool capacity is sufficient to prevent valid candidates from being evicted, ensuring high recall without unbounded memory consumption. □

ARTIFACT EVALUATED
USENIX
AVAILABLE

Efficient GPU-centric Evolving Graph Processing at Scale

Yunmo Zhang
City University of Hong Kong

Jiacheng Huang
City University of Hong Kong

Xizhe Yin
Independent Researcher

Junqiao Qiu
City University of Hong Kong

Hong Xu
The Chinese University of Hong Kong

Chun Jason Xue
Mohamed bin Zayed University of Artificial Intelligence

Abstract

Large-scale evolving graph analytics (EGA), which evaluates graph queries over sequences of snapshots, is facing growing demands for real-time insight extraction. While GPUs offer immense potential for accelerating graph workloads, they suffer from the memory capacity wall and poor hardware utilization when applied to EGA.

To bridge this gap, this work presents POEGA, a GPU-centric framework for efficient large-scale EGA. The core idea of POEGA is to leverage *proxy graphs* to minimize out-of-memory IO. It achieves this by first analyzing a compact in-memory graph abstraction to drive approximate results, thereby guiding the out-of-memory refinement. Although this approach incurs more computations, our key insight is that this cost can be amortized by exploiting the GPU's massive parallelism to process multiple snapshots concurrently. This concurrency is supported by a carefully designed fused kernel and a novel bound-based pruning technique. Furthermore, we address a commonly overlooked memory bottleneck caused by multi-version vertex states, which arises when scaling concurrent analysis to a large number of snapshots, by introducing an adaptive state compaction format. Evaluation shows that POEGA yields 3.7–23.5× speedups over the state-of-the-art EGA solutions across a range of real-world datasets.

1 Introduction

Graph processing serves as the backbone for unstructured data analysis across diverse domains. Since real-world graphs naturally expand and evolve, Evolving Graph Analytics (EGA), which extracts insights from a sequence of snapshots in a specific time window, has become a cornerstone of many modern decision-making systems. Crucially, these applications face an imperative demand for *real-time* responsiveness to the continuously changing real-world data. For instance, financial systems rely on evolving connectivity trends for instant fraud detection [31, 81]; network infrastructure requires evolving reachability analysis to pinpoint and recover from failures [29]; cybersecurity analysts track changing path properties to identify malicious DNS domains [69]; and epidemiological platforms leverage EGA for real-time influenza forecasting during COVID-19 pandemics [79].

A straightforward approach for EGA is to apply the query to snapshots individually, yet this yields significant overhead. A series of incremental graph analytics approaches, such as Tornado [66], Kickstarter [70], Graphbolt [46], CommonGraph [1], and others [16, 83], are proposed for resolving the urgent need for evaluating queries on graphs under dynamic environments. They have proven the superior efficiency of incremental computation over re-computing from scratch, given that evolving graphs typically exhibit high similarity (often exceeding 99% [12, 60]) between consecutive snapshots.

In recent decades, GPUs have been an attractive computing platform for graph applications [6, 24, 27, 35, 36, 49, 67, 71, 75]. State-of-the-art GPU-based incremental analysis systems, such as Grapin [74], have demonstrated over 10× speedups compared to CPU-based counterparts. However, applying these GPU-accelerated systems to large-scale graphs faces a fundamental challenge of limited GPU-resident memory, known as the GPU *memory oversubscription* problem [43, 56]. While general GPU data management techniques, ranging from explicit transfer [22, 62, 63] to implicit unified memory [58] or zero-copy [51], have been proposed, they often incur prohibitive data transfer overheads when applied to the incremental EGA (§2.3). Consequently, the I/O bottleneck fundamentally prevents existing GPU-based EGA systems from scaling to massive evolving graphs.

To bridge this gap, we introduce POEGA, an efficient evolving graph processing framework that tackles the memory wall problem. The main idea of POEGA is to leverage proxy graph, a concept that has recently been proven feasible for retaining the computation-critical structures of original graphs within a compact footprint [32]. By confining the initial analysis to a resident-memory proxy graph, POEGA derives approximate results to guide the subsequent exact refinement on the full graph, thereby significantly reducing the I/O traffic required. However, this two-phase paradigm (approximate-

then-refine) inherently introduces extra computations. They make previous proxy graphs yield marginal gains or even performance degradation because the extra compute overhead outweighs the I/O savings [32]. Our insight is that this computation overhead can be effectively *amortized* by exploiting the GPU's massive parallelism to process multiple snapshots in the evolving graph concurrently. We propose a snapshot-level concurrent execution model that fuses computations across snapshots into a single kernel to maximize instruction throughput and coalesce memory accesses. Furthermore, we mitigate redundant work across snapshots via a runtime bound-based pruning technique.

Finally, scaling this concurrent analysis introduces a new bottleneck: the memory pressure from maintaining vertex states across numerous snapshots, a challenge largely overlooked by existing concurrent models [10, 17]. To address this, we design an adaptive multi-version format that efficiently compacts vertex states, enabling POEGA to sustain high concurrency without exhausting GPU memory. To confirm the effectiveness of the proposed designs in POEGA, we conduct extensive experiments with real-world large-scale graphs and commonly used query benchmarks. The results show that POEGA outperforms the state-of-the-art EGA solutions, including EGraph [86], Grapin [74] and state-of-the-art incremental analysis methods, Kickstarter [70], and CommonGraph [1], combined with diverse GPU data management techniques [51, 58, 62] by 8.9× on average and up to 23.5×.

In summary, this paper makes the following contributions:

- Proposes POEGA, a framework for large-scale EGA that utilizes proxy graphs to shift the bottleneck from expensive I/O operations to GPU-friendly compute-intensive workflows;
- Presents a concurrent analysis method that fuses the execution and coalesces the data transfer, as well as a runtime pruning technique to reduce redundant computations across snapshots;
- Identifies the memory pressure of multi-version vertex states as a key impediment to scaling concurrency and proposes an adaptive multi-version format to address it;
- Provides an efficient implementation of POEGA and compares it with the state-of-the-art solutions through systematic evaluations. The results demonstrate its significant performance improvements. POEGA is open-sourced at https://github.com/YunMoZhang/POEGA.

2 Background and Motivation

2.1 Evolving Graphs Analytics

Analytics over dynamic graphs can be generally classified into three categories [18, 84], including: (1) *Streaming graph analytics* [14, 16, 46, 48, 70] which mainly explores the latest version of the graph as it continually changes; (2) *Temporal graph analytics* [8, 13, 28, 44, 47, 89], which exploits the timestamps of edges (and optionally vertices) to analyze fine-grained time-constrained behaviors, such as investigating general temporal path problems [52, 78, 89]; and (3) *Evolving graph analytics*, which is also called *snapshot sequence analytics* [1, 17, 23, 30, 37, 40]. Formally, EGA evaluates a query Q (such as single-source-shortest-path) on N snapshots of an evolving graph $[G_1, \cdots, G_N]$ (usually taken periodically) over a specified time period $[t_1, t_N]$, where $G_i = \{V, E_i\}$, and thus offers precise insights or information drawn from discrete historical data. Changes between consecutive snapshots[1] consist of a batch of edge additions Δ_i^+ and a batch of edge deletions Δ_i^-. This paper focuses on evolving graph analytics. An example is shown in Figure 1.

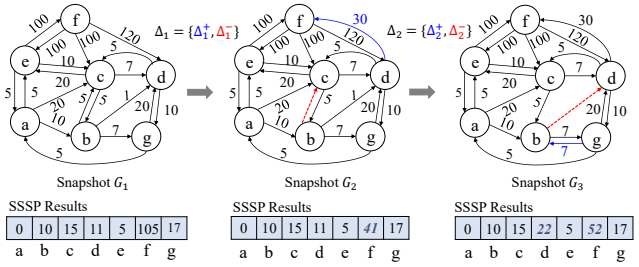


Figure 1: An EGA example which evaluates a query *SSSP(a)* on 3 snapshots.

Incremental EGA. The naive EGA approach evaluates the query independently on each snapshot from scratch, known as *re-evaluation*. To reduce redundant computations during re-evaluation, incremental algorithms have been proposed [14, 16, 46, 66, 70]. This paper focuses on the widely studied monotonic path-based graph algorithms [11, 70, 82]. Specifically, to process Δ_i^+, the incremental algorithm detects impacted vertices by propagating the value through the edges in Δ_i^+, and then adds updated vertices to the frontier. It reuses evaluation results on the previous snapshot and then propagates computation from affected vertices forward through the graph until re-convergence. For example, when the edge (d, f, 30) is inserted during $G_1 \rightarrow G_2$ in Figure 1, we apply the edge function from d to its out-neighbor f based on d's current value 11, and thus detect a new best value 41 for f.Then we put f to the frontier and resume the iterative evaluation until all values converge again. To process Δ_i^-, incremental algorithms usually maintain one-to-one dependencies that record which in-neighbor contributed to each vertex's computed value. When a deleted edge belongs to a dependency, incremental algorithms first perform *dependency tracing* to identify affected vertices. Tracing is also iterative, starting from directly impacted vertices and traversing along the cas-

[1]This work considers changes in the form of edge addition/deletion. Vertex or edge weight updates can be transformed to edge addition/deletion [83].

cading relationships till the end. All the affected vertices need to reset their values, pull the new one, and re-converge. As this kind of incremental algorithm [46, 70], such as representative Kickstarter [70], requires maintaining evaluation results in the previous snapshot, it is referred to as **streaming-based** incremental analysis.

As handling edge deletions in streaming-based approaches is generally more expensive (typically 2-3×) than processing the same number of additions [1], **batch-based** incremental analysis, such as pioneer work CommonGraph [1], has been proposed in recent years [1, 17, 88]. This approach first evaluates the query over a subgraph shared across all targeted snapshots. It then identifies per-snapshot Δ (only containing additions) relative to the common subgraph, and applies deletion-free incremental analyses to each snapshot independently. As it removes the dependency between snapshots, concurrent query evaluation is enabled in this approach.

2.2 GPU-based Large-scale Graph Processing

GPUs have become a popular platform for accelerating graph processing due to their high compute throughput and memory bandwidth [24, 27, 36, 49, 75]. However, GPU-based graph processing frameworks remain constrained by limited device memory, facing challenges when processing today's ever-growing graph datasets. This constraint becomes even more pronounced in EGA, where queries must be evaluated over multiple snapshots, further increasing memory demand.

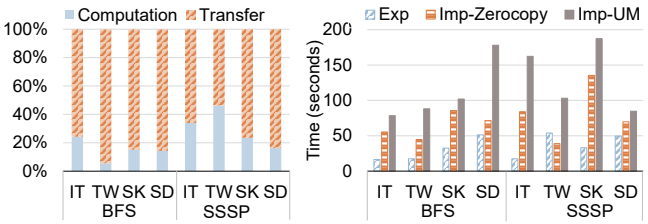
Handling GPU memory oversubscription. To enable graph processing when the graph size exceeds the GPU memory capacity (i.e., oversubscription), existing efforts can be mainly classified into two categories: (1) ***Explicit management***. A coarse-grained approach [40, 86, 93] partitions the oversized graph into subgraphs that can fit into GPU memory, and then transfers partitions containing active edges in each iteration. However, this often incurs unnecessary data movement, as an entire partition may be transferred even when only a few edges are active. To address this inefficiency, finer-grained design [62, 91], including state-of-the-art Subway [62], adopts a dynamic partitioning strategy that generates subgraphs on demand per iteration. It first applies data compaction after knowing the active frontier, and then transfers compacted subgraphs. While minimizing transfer volume, it requires additional computation overhead. (2) ***Implicit management***. To ease the engineering efforts of explicit data movement, recent studies explore leveraging Unified Virtual Memory (UVM) [3, 4, 19, 34], which maps host and GPU memory into the same address space and triggers on-demand page migration (typically 4KB pages) during graph processing. However, this may incur non-negligible overhead due to costly page faults. A finer-grained method investigated recently is zero-copy access [51, 59, 73], which allows GPUs to directly read from host memory via transaction layer packets of PCIe. According to the PCIe spec [51], up to 256 outstanding memory requests can be in flight concurrently, each accessing 32~128 bytes. This enables fine-grained and parallel data access. However, zero-copy may cause redundant transfers upon repeated access to the same data [55, 73].

Proxy Graphs. In recent years, utilizing proxy graphs has emerged as a promising alternative for processing large-scale graphs [32]. A proxy graph is a compact graph abstraction that retains computation-critical edges to preserve topological semantics of the original graph. This idea has been explored across several contexts, including reducing graph complexity for analytics and visualization [32, 38, 41, 76, 92], optimizing partition navigation in distributed environments [25, 53, 72], and relieving memory constraints in out-of-core systems [32, 85]. Existing proxy graph construction methods primarily follow three routes: (1) ***Structure-preserving Reduction***. Early methods in this direction used edge contraction, i.e., merging connected vertices, to generate coarse-grained graph representations [25, 53, 72]. While effective for regular meshes, such methods may abstract away individual vertex details. Input Reduction [38] instead removes graph components that do not affect query results in irregular graphs, e.g., vertices with no incoming or outgoing edges. While preserving exact correctness, its reduction ratio is typically constrained by the inherent connectivity of the graph (e.g., around 50% [38]). (2) ***Graph Sampling***. This line of work utilizes edge sampling techniques, such as random walk [41] and priority-based selection [92] (e.g., selecting low-weight edges for shortest-path queries), to bound proxy graph size. Wonderland [85], an out-of-core graph system, further enhances connectivity by incorporating edges that bridge weakly connected components. These sampling-based abstractions prioritize a small memory footprint; yet their lossy nature often necessitates an intensive refinement phase to ensure result accuracy. (3) ***Query-based Sketch***. To better align the abstraction with query semantics, recent methods propose constructing proxy graphs based on offline pre-evaluated queries [32, 76]. For instance, CoreGraph [32] evaluates a set of queries with sources at high-degree vertices and includes all edges on the resulting critical paths. This approach yields high accuracy (often exceeding 95%), though it involves a non-trivial generation phase.

2.3 Motivation

While state-of-the-art incremental EGA techniques achieve promising efficiency on GPUs [17, 18, 74], their performance collapses when the graph scale exceeds GPU memory capacity. To better understand these limitations and guide the design of a more efficient solution, we profile representative incremental EGA systems and identify two critical challenges and one key opportunity.

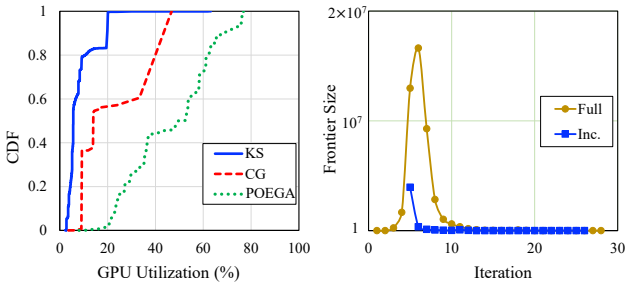
Challenge #1: The I/O Bottleneck in Data Transfer. We first break down the execution time of batch-based incremental EGA under memory oversubscription to evaluate existing data transfer strategies. As shown in Figure 2 (a), the overhead



(a) Breakdown of execution time using Explicit data transfer.

(b) Performance comparison between Explicit and Implicit data transfer.

Figure 2: Data transfer overhead in evolving graph incremental analysis (settings in §6).



(a) GPU utilization in different systems

(b) Frontier size distribution.

Figure 3: Profiling of incremental *SSSP* on graph *IT*. For better comparison, the curve about frontier sizes during incremental analysis has been right-shifted in (b).

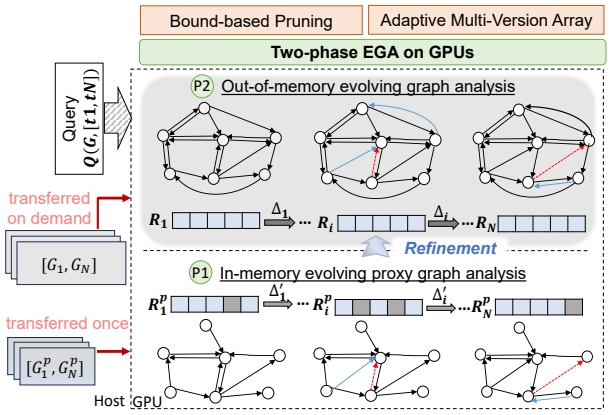


Figure 4: Workflow of EGA in POEGA.

of explicit data transfer dominates the execution, accounting for 77.6% of the total analysis time on average, and peaking at 94.4%. Implicit management strategies, zero-copy and unified memory (UM), are even worse, exhibiting significantly higher latency as shown in Figure 2 (b). These results underscore a fundamental barrier: even if an ideal policy transfers only the exact on-demand data per iteration, the heavy I/O overhead remains prohibitive for incremental analysis. This suggests that simply optimizing data transfer is insufficient; a paradigm shift to reduce the volume of required I/O is necessary.

Challenge #2: The Memory Bottleneck from Multi-Version States. While vertex-centric data is traditionally assumed to fit within device memory [51, 62, 63], this assumption breaks down under multi-snapshot concurrency. For instance, the value arrays for the Subdomain graph (used in our evaluation) across 40 snapshots consume approximately $102\text{M} \times 4\text{B} \times 40 \approx 16\text{GB}$, exceeding our GPU capacity (see §6). In practice, when co-located with graph structure data, the system encounters out-of-memory errors even earlier, typically when $N > 30$. Crucially, this scalability hurdle is overlooked by existing concurrent approaches [10, 17, 82], which assume all concurrent state arrays are memory-resident. A naive workaround involves offloading value arrays to host memory and transferring them on demand, as for graph data. Yet given the high frequency of fine-grained, atomic state updates in graph analytics, this approach incurs prohibitive transfer overheads, severely throttling the performance.

Opportunity: Exploiting Computational Underutilization. We observe that current GPU-based EGA methods suffer from severe hardware underutilization. Figure 3(a) presents the Cumulative Distribution Function (CDF) of thread utilization (defined as the ratio of active threads to total threads) when evaluating an *SSSP* query over 32 snapshots. We compare a streaming-based method, KS [70], and a batch-based approach, CG [1], both using fine-grained explicit memory management [62]. KS exhibits extreme inefficiency, with GPU usage remaining below 10% for over 60% of the runtime, primarily due to its serial snapshot processing scheme. While CG improves utilization, it still remains below 45% overall. The root cause is the sparsity of incremental workloads. As shown in Figure 3(b), the incremental analysis of a single snapshot typically activates significantly fewer vertices (one to two orders of magnitude) compared to a full re-evaluation. This creates a massive utilization gap, presenting a prime opportunity to amortize the cost of our proposed proxy-based approach by processing multiple snapshots concurrently.

3 Overview of POEGA

To enable efficient incremental EGA on GPUs, we propose POEGA, a holistic system that orchestrates a Proxy-Guided Concurrent Analysis execution model. Figure 4 provides an overview of the workflow of POEGA. Inheriting the two-phase methods of proxy graph approaches, POEGA is tailored for evolving graphs, including: (P1) *In-Memory Evolving Proxy Graph Analysis*, and (P2) *Concurrent Out-of-Memory Incremental Analysis for Snapshots*.

POEGA system accepts an evolving graph query on snapshots within a time interval $[t_1, t_N]$ and operates as follows. At P1, POEGA first transfers the *evolving proxy graph* to the GPU and then evaluates the query on it entirely within the memory, generating the approximate results $R_1^p, \ldots, R_N^p$. Guided by these results, POEGA at P2 performs exact incremental analy-

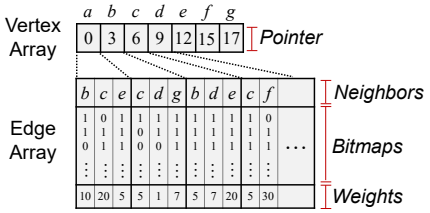


Figure 5: UnionCSR format of an evolving graph in POEGA.

sis on the full snapshots, which are outside the GPU memory and transferred only *on demand*. Thanks to our incremental-analysis-oriented evolving proxy graph (§4.1), the analysis at this phase is *deletion-free*, requiring minimal data transfers. In this phase, POEGA fuses the execution of multiple snapshots into a single kernel, which enforces data access of neighbors coalesced across snapshots (§4.2). Furthermore, the computations that can also be coalesced are identified and pruned by our novel runtime technique based on checking the bounds of values across snapshots (§4.3). Finally, to enable more snapshots to enjoy the benefits of concurrent analysis, POEGA manages the values of snapshots using Adaptive Multi-Version Array, a novel format for resolving the memory bottleneck in concurrent analysis on GPUs (§5).

Format of Evolving Graphs. We represent the evolving graph using a *snapshot-union* Compressed Sparse Row (CSR) format. Any edge that appears in at least one snapshot is stored once in this *UnionCSR* structure. Figure 5 shows this representation of the evolving graph in Figure 1. Like classical CSR format, adjacency information of the graph is maintained by (1) the vertex array, which records the start offsets of each vertex's neighbors, and (2) the edge array, which concatenates all neighbors. To differentiate edge presence across snapshots, UnionCSR associates each edge with a bitmap, where the i-th bit indicates its presence in snapshot G_i. By default, a 64-bit word is used for alignment, while additional words can be added to support more snapshots. This design enables fast GPU thread data access, akin to compact formats in prior works [12, 17]. The difference is that UnionCSR supports diverse data access interfaces by $O(1)$ bit operations, including view(G_i) for a single snapshot, view_shared(G_i, G_j) for shared subgraph, and view_diff(G_i, G_j) for the delta $G_j - G_i$. As this work focuses on enabling efficient GPU-based analytics, we assume graph updates are ingested offline and treat storage optimizations [7, 21, 65, 77] as orthogonal to ours. In fact, UnionCSR can be seamlessly generated using the standard parallel CSR computation from the durable graph storage, such as the multi-version adjacency list in GraphOne [37].

4 Proxy Graph-based EGA

The main idea of POEGA is shifting the out-of-memory irregular large-scale evolving graph processing into a two-phase processing, and solving the extra computation introduced by refinement via our efficient concurrent analysis on GPUs.

4.1 Incremental Analysis Oriented Evolving Proxy Graphs

Prior studies [32, 76] have demonstrated the feasibility of building high-fidelity approximations with a condensed structure for *static* graphs. For example, the state-of-the-art Core Graph [32] achieves over 95% accuracy with only 10% of the original edges. Starting from the approximate results on the proxy graph, the subsequent phase analysis refines the results on the full graph. Fundamentally, proxy-based approaches *trade* reduced memory I/O (by confining the bulk of access to the refinement phase) for increased total computation (processing both proxy and full graphs).

However, applying existing proxy designs to evolving graphs faces a fundamental dilemma between construction cost and approximation fidelity. On one hand, lightweight heuristics like Wonderland [85] yield low accuracy (e.g., around 50%), less effective in reducing the out-of-memory accesses in the second-stage analysis due to the iterative nature of graph processing. On the other hand, high-fidelity proxies like Core Graph are designed for static snapshots; naive reconstruction for every snapshot is computationally prohibitive (e.g., 7–14 minutes per snapshot [32]), outweighing any analytical benefits. More critically, reusing a single static proxy for an evolving sequence is inferior to structural divergence. After a series of updates, the proxy graph is no longer the subgraph of snapshots. Consequently, the second-phase analysis that processes the delta between the snapshot and the proxy graph needs to handle deletions, which are expensive to process, e.g., 2× slower than processing additions [1].

Maintained Evolving Proxy Graphs (EPG). Notice that *evolving graph changes slowly*, a well-known belief that has been used in optimizing evolving graph analytics [12, 30, 46, 70] and storage [12], we employ a single high-fidelity base proxy to capture the critical graph structure of recent snapshots and use a lightweight maintenance heuristic to adapt it computation-friendly to the second-phase refinement.

The complete EPG construction design is summarized in Figure 6. Following standard graph storage practices [42], we periodically construct proxy graph checkpoints offline. These proxy graphs are generated by evaluating queries on the top-K highest-degree vertices. For each vertex reached by a query, different from existing methods [32, 76], we select only one edge on its critical path (e.g., the shortest path from the source), thus keeping the proxy graph size complexity of $O(|V|)$. In our implementation, we empirically set K to 10 by default. To ensure sufficient structural coverage, if the resulting proxy remains below a predefined threshold (e.g., 10%) of the original snapshot size, we further augment it by evaluating other high-degree vertices that appear at a few hops from the source of the existing critical paths for the K queries.

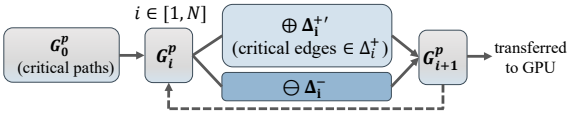


Figure 6: Design of Maintained Evolving Proxy Graph.

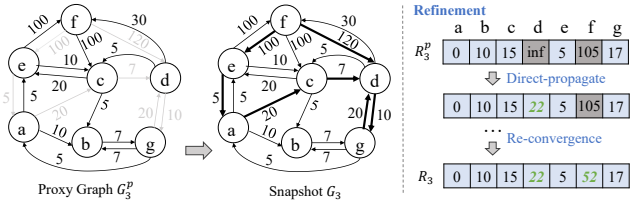


Figure 7: An illustrated example of applying refinement.

After the arrival of query Q on window $[t_1, t_N]$, we retrieve the nearest preceding checkpointed proxy graph at t_0 as the *base proxy* graph for EGA, denoted by G_0^p, where $t_0 \leq t_1$. We then apply the maintenance heuristics to map original deltas (Δ_i^+, Δ_i^-) onto G_0^p for generating subsequent proxy graphs:

$$G_{i+1}^p = G_i^p \oplus \Delta_i^+ \ominus \Delta_i^-, \quad i \in [0, N). \tag{1}$$

When the addition deltas are large, we selectively choose two types of edges from Δ_i^+ to keep the proxy graph compact while preserving fundamental structure: (1) bridge edges that connect two high-degree vertices, and (2) backup edges that connect any low-degree vertex to others.

Beyond the differences from existing work in pre-evaluation query and critical edge selection, the maintained EPG is specifically designed for dynamic scenarios with two key benefits. First, its update logic can be integrated seamlessly with the original graph's update procedure, and can even be executed concurrently to hide EPG generation latency. Second, by explicitly applying deletions, EPG remains a strict subgraph of the original graph ($G_i^p \subset G_i$). As a result, the evolving proxy graphs can effectively drive the *deletion-free* incremental analysis in the second phase, while keeping the high structural fidelity.

4.2 Two-stage Evolving Graph Analysis

P1: *In-memory Evolving Proxy Graph Analysis*. In this phase, the query is evaluated on the proxy graphs G_i^p to derive approximate results $\{R_i^p\}$ for the corresponding snapshots. After transferring the UnionCSR format of evolving proxy graphs, which is generated when creating the format of the original evolving graph, to the GPU once, this entire phase executes entirely within the GPU memory. As a result, evolving proxy graphs can be evaluated efficiently using existing in-memory EGA *incremental analysis* engines, such as streaming-based [46, 70] and batch-based [1] mentioned in §2.3. We adopt the batch-based incremental analysis method, as it aligns more with the GPU's massive parallelism.

By first analyzing the shared subgraphs of a series of proxy graphs G_c^p (via view_shared), batch-based analysis first obtains R_c^p and then incrementally processes additional edges of the proxy graphs to the shared graphs $\Delta(G_i^p - G_c^p)$ (via view_diff) to obtain R_i^p. Note that only the edge additions over the subgraph are processed in this process, where each edge e in the delta is tried once in relaxation operations (called *tracing*), e.g., in SSSP, push the value of source $e.src$ to the destination $e.dst$ if $val(e.src) + e.weight < val(e.dst)$. The affected vertices are added to the frontier to finish the convergence as the normal graph processing. Since these incremental analyses start from the shared subgraph that has no inter-dependency, we thus parallelize them using the same concurrent analysis kernel in the second phase below.

P2: *Concurrent Out-of-memory Incremental Analysis for Snapshots*. In the second phase, POEGA refines the approximate results R_i^p into precise results R_i for each snapshot G_i. This is achieved by treating the graph difference, $\Delta(G_i - G_i^p)$, as a batch of edge additions and performing an incremental analysis. Benefit from our evolving proxy graph construction ($G_i^p \subset G_i$), the incremental analysis performed during refinement is deletion-free. This process consists of two primary steps: (1) *direct-propagate* the addition delta to identify the effects of $\Delta(G_i - G_i^p)$ to R_i^p and (2) *re-convergence*, which resumes iterative processing until convergence. As introduced in existing incremental analysis designs (§2), the direct-propagate step performs a single-pass traversal over the delta edges, applying the edge function to each. An illustrated example is shown in Figure 7. After getting the approximate results for the proxy graph of snapshot G_3 in Figure 1, the direct-propagate step processes the delta edges (bold in G_3). Specifically, it applies the edge function for the delta edge (c, d) and then updates the value of vertex d to 22. Subsequently, during the re-convergence phase, this updated value is further propagated through the graph iteratively, ultimately refining the result of f. Since this refinement aligns with traditional incremental analysis [23, 66, 70], their correctness in reaching the precise value of G_i is inherently guaranteed.

Different from P1, both steps in P2 require out-of-memory graph data accesses. For the direct-propagate, since the delta size is relatively large, we use the explicit data management method [62] to transfer the data to the GPU memory to avoid the throttling performance. For the following convergences, we employ the implicit data transfers, since the incremental analysis involves a relatively small frontier size, where the implicit transfers outperform the explicit ones as they avoid the data preparation (e.g., compaction).

Coalesced Computation. Since the processing logic is the same across snapshots (for both proxy and full graph analysis), we *fuse* the analysis of multiple snapshots into a single kernel to concurrently execute them, as shown in Algorithm 1. Thanks to the UnionCSR format, this concurrent analysis

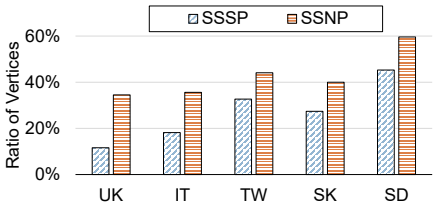


Figure 8: Inter-snapshot Redundant Computations in EGA.

Table 1: Bounds in the conditions for pruning over edge (u,v).

Alg.	Push conditions	$\bowtie$	$\mathcal{B}_{src}(val(u))$	$\mathcal{B}_{dst}(val(v))$
SSSP	$val(u)+w<val(v)$	$<$	$val^{\downarrow}(u)$	$val^{\uparrow}(u)$
BFS	$val(u)+1<val(v)$	$<$	$val^{\downarrow}(u)$	$val^{\uparrow}(u)$
SSWP	$min(val(u),w)>val(v)$	$>$	$val^{\uparrow}(u)$	$val^{\downarrow}(u)$
Viterbi	$val(u)/w>val(v)$	$>$	$val^{\uparrow}(u)$	$val^{\downarrow}(u)$
SSNP	$max(val(u),w)<val(v)$	$<$	$val^{\downarrow}(u)$	$val^{\uparrow}(u)$
WCC	$val(u)<val(v)$	$<$	$val^{\downarrow}(u)$	$val^{\uparrow}(u)$

coalesces the access to the neighbors of a vertex. Instead of issuing individual memory requests for each snapshot, threads access the neighbor union of a vertex in a single transaction (Lines 3-4). The kernel then determines edge presence in a snapshot via bitwise operations (Line 6). This reduces GPU global and out-of-memory traffic regardless of explicit or implicit data transfers. Analyzing vertex v in N snapshots in an iteration requires $|neigh(v)|$ times memory access, outperforming $\sum_{i\in[1,N]}|neigh_i(v)|$ required by separate kernels. Besides, the access to the active vertex set, i.e., frontiers of snapshots, is coalesced (Line 2) by maintaining the union of frontiers (Line 10), similar in spirit to prior static graph multi-query concurrent analysis [10,82]. Using this technique, the memory footprint is reduced, while the correctness of monotonic algorithms is not affected [82]. But since this method introduces unnecessary computations, we optionally also maintain separate frontiers if the memory budget allows.

Finally, thanks to the fused processing of multiple snapshots in one kernel, as of one snapshot, existing GPU thread scheduling schemes, such as the vertex-centric [36], warp-centric, or block-centric [75] processing, can be seamlessly applied to our concurrent analysis kernel to solve the thread mapping efficiency of graph algorithms on GPUs.

Algorithm 1 Coalesced Concurrent Analysis (G,$[1,N]$)

```
__global__ SSSP_CoalConcur(...)
1:  /* id is thread ID, warp ID, or block ID. */
2:  vid = frontier[id];
3:  for eid = G.vertexPtr[vid] to G.vertexPtr[vid + 1] do
4:      e = G.neighbors[eid];
5:      for sid = 1 to N do /* a thread or a warp of threads */
6:          if e.bmp[sid] then
7:              new_val = value[vid][sid] + e.weight;
8:              if new_val < val[e.dst][sid] then
9:                  CASMIN(val[e.dst][sid], new_val);
10:                 frontier_n ← e.dst;
```

4.3 Bound-based Pruning

In the concurrent analytics frameworks discussed in previous sections and prior works [10, 82, 87, 90], computations for neighbors, including arithmetic operations and atomic updates, are executed individually for each snapshot. This occurs even when the graph data accesses are coalesced.

However, we identify that this distinct per-snapshot computation incurs significant redundancy due to the inherent temporal locality of evolving graphs. We define *redundant computation* as edge operations that yield identical values or trigger the same control-flow outcomes (e.g., effective updates) on the same vertex across all snapshots. Our profiling shown in Figure 8 reveals that such redundancies constitute 18.2% to 59.6% of the total computations, highlighting a substantial potential for optimization.

Eliminating these redundant computations is challenging because vertices with identical converged results cannot be identified a priori. To address this, we propose a runtime pruning technique based on the **bounds of values**. This approach leverages *necessary conditions*, rather than exact conditions, to rapidly identify and prune redundant operations.

Definition 1 (Value Bounds)**.** *Let $val_i(u)$ be the converged or intermediate value of vertex u in snapshot i, where $i\in[1,N]$. We define its lower bound as $val^{\downarrow}(u)=\min_{i=1}^{N}(val_i(u))$ and the upper bound as $val^{\uparrow}(u)=\max_{i=1}^{N}(val_i(u))$.*

We now detail the bound-based pruning mechanism applied to the computations for an edge (u,v) across snapshots $G_1,\ldots,G_N$. First, we abstract the generic relaxation criterion that serves as a prerequisite for updating $val(v)$, denoted as $[val(u)\otimes w(u,v)]\bowtie val(v)$. Here, $\otimes$ represents the numerical operator that generates a candidate value, and $\bowtie$ denotes the relational operator that determines whether the candidate yields a superior result. While we formulate this using the *push* model, this logic applies symmetrically to the *pull-based* model (where v checks incoming neighbors). The specific operators for the graph algorithms evaluated in this work (§6) are listed in Table 1.

Definition 2 (Pruning Criteria)**.** *For an algorithm Q, the computations over edge (u,v) across all snapshots can be safely pruned if the following necessary condition **fails to hold**: $[\mathcal{B}_{src}(val(u))\otimes w(u,v)]\bowtie\mathcal{B}_{dst}(val(v))$ is False, where $\mathcal{B}_{src}$ and $\mathcal{B}_{dst}$ represent the specific bounds (upper or lower) determined by $\bowtie$.*

Table 1 specifies the bounds for different algorithms. The logic is as follows: if $\bowtie$ is $<$, a successful update requires the candidate value to be smaller than the current value. Therefore, we use the *lower bound* for the source u (best-case can-

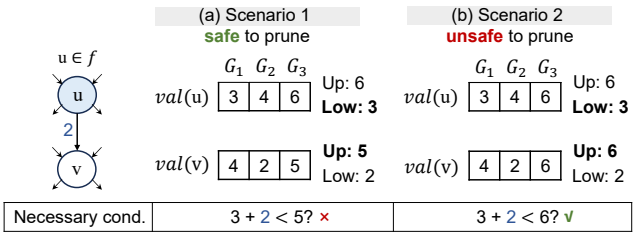


Figure 9: Pruning the push operations over edge (u, v) when concurrently serving an *SSSP* query in three snapshots.

didate) and the *upper bound* for the destination v (worst-case target). If this "best-case" fails to trigger an update, computations for all snapshots can be skipped. The inverse logic applies when $\bowtie$ is $>$.

An example of this pruning technique for SSSP is shown in Figure 9. In Case (a), the source's lower bound (after adding weight) is 5, which is not smaller than the destination's upper bound (5). This mathematical guarantee ensures that no snapshot can trigger a valid update, allowing the kernel to safely skip all N computations/comparisons for this edge. Case (b) highlights the necessary nature of this check. Although the bound condition allows the computation to proceed (i.e., no pruning), specific values in individual snapshots may still fail to trigger updates. Thus, passing the bound check implies a potential, rather than guaranteed, update.

Algorithm 2 Coalesced Concurrent Analysis with Pruning

```
__global__ SSSP_CoalConcur_Prune(...)
1:  vid = ...
2:  for eid = G.vertexPtr[vid] to G.vertexPtr[vid + 1] do
3:      e = G.neighbors[eid];
4:      if not Q.necessary(vid, e.dst, Upper, Lower) then
5:          continue; /* Bypass Redundant Computation */
6:      for sid = 1 to N do
    ... /* Lines 6-10 in Algorithm 1 */
__global__ Update_Bound(...)
14: if vid in frontier then
15:     Upper = max(val[vid][...]);
16:     Lower = min(val[vid][...]);
```

Lazy Update of Bounds. In implementation, we store the bounds of the vertices in two arrays: `Upper` and `Lower`. However, maintaining exact bounds in real-time on the GPU is expensive, as it necessitates frequent atomic updates to the global `Upper/Lower` arrays, leading to high memory contention. A key insight is that *out-of-date* bounds can still serve as a valid basis for safe pruning, even if they miss some optimization opportunities. We thus introduce a lazy update runtime technique for the bounds. As shown in Algorithm 2, instead of updating bounds immediately after the edge relaxation within each iteration, the lower and upper bounds of a vertex's values are updated only at the end of each iteration, via a separate lightweight kernel (Lines 14 to 16).

Correctness of the runtime lazy update. Using lazy bounds preserves the algorithmic correctness by analyzing the effects of out-of-date bounds of the destination ($\mathcal{B}_{dst}^{stale}$) and source ($\mathcal{B}_{src}^{stale}$) vertices separately. Without loss of generality, we assume a monotonically decreasing algorithm (e.g., SSSP), where both upper and lower bounds can only decrease or remain static.

- For the destination vertex, the pruning checks its upper bound. Since values decrease monotonically, the lazily updated (stale) upper bound is always a *conservative overestimation* of the true state (i.e., $\mathcal{B}_{dst}^{stale} \geq \mathcal{B}_{dst}^{true}$). If the pruning condition (e.g., $Candidate \geq \mathcal{B}_{dst}^{stale}$) holds for the looser stale bound, it guarantees that it also holds for the tighter true bound. Thus, a stale $\mathcal{B}_{dst}$ only leads to missed pruning opportunities, but never incorrectly prunes a valid update.

- For the source vertex, the pruning checks its lower bound. A stale lower bound implies $\mathcal{B}_{src}^{stale} \geq \mathcal{B}_{src}^{true}$, which may result in an inflated candidate value. Theoretically, this could trigger a "false positive" pruning (i.e., skipping a computation that should be performed). However, correctness is maintained by the *iterative nature* of the graph algorithm: if $\mathcal{B}_{src}^{true}$ has decreased, the source vertex u must have been updated and added to the active frontier. Consequently, this edge will be re-evaluated in the next iteration with the updated bounds. While this may slightly delay convergence, it ensures no valid updates are permanently lost.

5 Divergence-Aware Vertex State Management

While the concurrent execution design in POEGA effectively harnesses GPU compute resources, the achievable degree of concurrency remains bottlenecked by the $O(|V| \times N)$ space complexity required to maintain query states (e.g., distance values) across N snapshots. To address this issue without resorting to expensive host-device transfers or concurrency-limited execution, POEGA introduces the **Adaptive Multi-Version Array (AMVA)**, a divergence-aware storage backend that dynamically manages a hybrid scalar-vector layout.

5.1 Dynamic Data Layout

Design Goals. Existing dynamic graph formats on GPUs, including those based on Packed Memory Arrays (PMAs) [65, 94] and linked lists [21, 77], are predominantly optimized for topological mutability (e.g., structural edge insertions and deletions). However, these designs are ill-suited for the vertex state management required by incremental graph analysis, which involves massive, concurrent read-modify-write operations. Consequently, a high-performance state management design should satisfy three requirements:

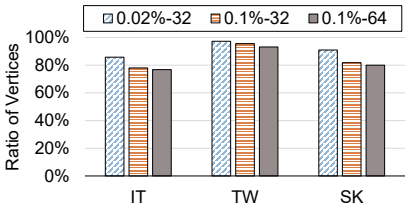


Figure 10: Inter-snapshot Equivalent Values in EGA ($|\Delta|$-N).

- **Deterministic $O(1)$ access complexity**. Data structures requiring traversal (e.g., pointer chasing in linked lists) or search (e.g., binary search in PMAs) to locate value data introduce significant latency, and exacerbate thread divergence and stall execution pipelines on GPUs, leading to severe performance degradation.
- **Spatial locality for memory coalescing**. Conventional dense arrays (as shown in Figure 11(a)) offer excellent spatial locality. Conversely, adopting formats with poor locality (e.g., hash-based structures [5]) would break memory coalescing and waste GPU memory bandwidth.
- **Minimal data movement overhead**. Dynamic structures like PMAs often require shifting contiguous memory blocks to accommodate element insertion. Such cascading data movement triggers severe write amplification and requires costly global synchronization [88]. Therefore, the proposed state management design must avoid physical data relocation or re-allocation [77].

As shown in Figure 10, empirical analysis reveals that a substantial portion of vertices (65% to 95%) maintain invariant state values across all snapshots during incremental EGA. This indicates that statically provisioning a full-version value array for every vertex is prohibitively wasteful. However, since the set of stable vertices is input and query dependent and unknown *a priori*, exploiting this redundancy requires a design that performs adaptive compaction at runtime. To satisfy the design goals, we propose the Adaptive Multi-Version Array. Specifically, AMVA employs a *dual-format* storage strategy: (i) *Scalar format*, which stores a single value shared across all snapshots for stable vertices; and (ii) *Full-version format*, which maintains a dense vector of N values for divergent vertices. Compression schemes (e.g., storing only unique values) are deliberately avoided for deterministic O(1) addressing: the current binary design ensures that states of a vertex in any snapshot can be accessed via either a direct read or a single-level redirection.

Layout. The AMVA consists of two core components:

(i) The Vertex Directory (VD). The VD acts as the primary lookup table, implemented as a linear array where the i-th element corresponds to vertex v_i. To minimize memory footprint, a *tagged pointer* scheme is employed to multiplex data and metadata within a single word (extensible to larger data types). As shown in Figure 11(b), the two Most Significant Bits (MSBs) serve as the divergence flag, while the remaining bits constitute the payload. The flag bits encode the current mode of the vertex. Across different modes, the payload stores either the shared value directly or an index offset pointing to the Expansion Buffer. More details are shown in §5.2.

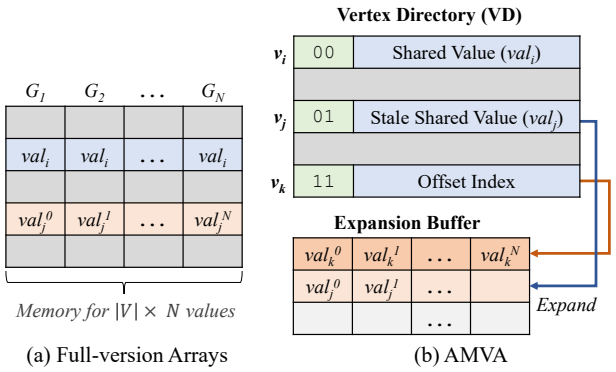


Figure 11: Vertex State Representations.

(ii) The Expansion Buffer. To accommodate divergent vertices, we pre-allocate a contiguous memory block on the GPU, organized as B_{max} rows where each row contains N elements. The j-th element of a row stores the value for snapshot G_j. When a vertex transitions from scalar format to full-version format, a free row from this buffer is assigned to that vertex, i.e., *expansion*. To ensure high-throughput expansion, we utilize a **lock-free bump-pointer allocator** [64] design with managing a global atomic counter that tracks the index of the next available row. Consequently, assigning a buffer row requires only a single atomic update (e.g., `atomicAdd`).

Memory Management. POEGA adopts a flexible provisioning policy. Prior to execution, it checks if the residual GPU memory suffices for full state materialization. In unconstrained scenarios (e.g., small datasets or low concurrency), POEGA reverts to the flat full-value arrays. However, when memory bottleneck comes from vertex state management, AMVA is instantiated with a total footprint bounded by $(|V| + B_{max} \times N)$. We determine the buffer capacity B_{max} based on the remaining memory budget after allocating essential graph structures (e.g., the proxy graph, frontiers, etc.). Due to the high stability of vertex values in EGA, this pre-allocated buffer typically accommodates all divergent vertices. In cases where extreme divergence exhausts the buffer, POEGA enforces *concurrency throttling*, which restricts the batch size. We consider this an acceptable trade-off, because such high-divergence scenarios typically imply sufficient workload to saturate GPU compute resources.

5.2 Lock-Free Access Protocols

To support lock-free expansion of a vertex value, three operational modes for elements in the VD are defined using

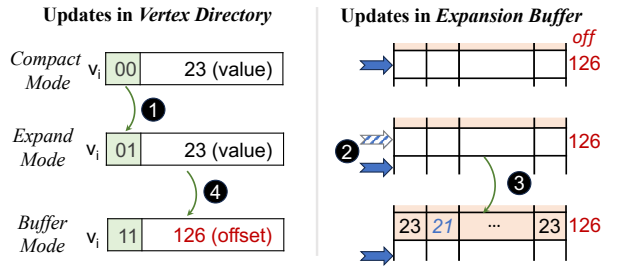


Figure 12: An illustrated example of Write-with-Expansion in AMVA. Assume that `atomicMin`(v_i, 21) is executed in the edge function during an iteration of incremental EGA.

the divergence flag (i.e., tag bits): (1) *Compact* mode (`00`), where the lower bits store the scalar shared value; (2) *Expand* mode (`01`), which indicates an ongoing expansion, where the lower bits still hold the valid but potentially stale value; and (3) *Buffer* mode (`11`), where the payload stores the offset index pointing to the expansion buffer. Based on these modes, AMVA supports three core operations on the vertex states: write(), write_expand(), and read().

Direct Write. When a thread attempts to update the state of a vertex v in snapshot G_i after verifying the push condition of edge (u,v) (e.g., $val_i[u]+w < val_i[v]$ in SSSP), the write() operation handles two specific scenarios: (i) *Buffer Mode Detected*. If the thread finds v is in the Buffer mode (tag `11`), it follows the payload pointer and directly writes the new value into the expansion buffer using an atomic operation. (ii) *Expand Mode Detected*. If v is in Expand mode (tag `01`), it indicates another thread is currently processing an expansion. The current thread waits until the mode transitions to Buffer Mode, after which it completes the write update.

Write with Expansion. When a thread attempts to update the state of a vertex v in the Compact mode, it initiates the expansion procedure and finishes the update via write_expand(). To ensure data consistency under high concurrency, we design a four-step atomic expansion protocol, as shown in Figure 12. ❶ Ownership Acquisition. The thread attempts to acquire exclusive rights to expand the values of v using an atomic Compare-and-Swap (CAS) instruction. It tries to transition the state mode from Compact to Expand while preserving the original value in the payload. Only one thread succeeds; failing threads infer that another writer has either claimed the vertex or completed the expansion, thus triggering a retry of write(). ❷ Low-overhead Allocation. The owner thread allocates a new row in the expansion buffer by performing an atomic addition of the global `counter` with the old `offset` recorded. ❸ Data Migration and Update. The owner thread expands the old scalar value (carried in the CAS payload) to all snapshot entries in the newly allocated buffer row. It then writes the new value (e.g., $val_i[u]+w$) directly into the target snapshot column i. A memory fence is issued after this step to ensure these writes are visible to all threads. ❹ Atomic Publishing. Finally, the owner commits the expansion by atomically updating the entry in the VD. The tag is flipped to Buffer mode (`11`), and the payload is replaced with the buffer index `offset`. This atomic instruction acts as the linearization point: once executed, all subsequent accesses are seamlessly redirected to the buffer.

Direct Read. When accessing a vertex state, read() first inspects the MSB of the tag: If MSB is 1 (i.e., Buffer mode), the reader thread follows the offset payload to retrieve the latest value from the buffer. If MSB is 0 (Compact or Expand mode), the value stored in the VD is consumed directly. Note that in Expand mode, the value retrieved may technically be "stale", but this will not affect the final results of incremental EGA when the target graph algorithm exhibits monotonicity.

Correctness of Stale Reads. Let d_{stale} denote the stale state read from a vertex currently in Expand mode and d_{new} denote the finalized state being written to the buffer. This vertex can be a *source* or a *destination*. (i) For source access: propagating updates derived from a source d_{stale} corresponds to processing a valid previous state of the graph. This may trigger *loose* updates (e.g., a path distance that will later be shortened), which are valid relaxations respecting the algorithmic invariants. The algorithm will eventually converge to the correct fixed point once the *tighter* d_{new} is propagated. (ii) For destination access: If a candidate update fails to improve upon d_{stale} , it is insufficient. Since the candidate update is worse than d_{stale}, the update is transitively worse than d_{new}, considering the monotonicity property (e.g., $d_{stale} \leq d_{new}$ in minimization problems). Thus, discarding the update based on d_{stale} yields the same logical outcome as checking d_{new}. In sum, the correctness of this read() operation is guaranteed.

6 Evaluation

6.1 Evaluation Methodology

Implementation. We prototype PoEGA as a graph framework written with C++ and CUDA. We integrate our framework with an open-source in-memory GPU-based static graph processing framework [57, 75]. To make a fair comparison, we incorporate this framework and UnionCSR format into all the compared systems.

Baselines. We compare PoEGA with the following systems:

- *Re-evaluation-based Approach:* **EGraph** [86], which employs a re-evaluation analytics engine and reuses the transferred shared graphs among snapshots to reduce data transfer overhead.
- *Streaming-based Incremental Analysis Approach*: including **Grapin** [74], which employs the state-of-the-art streaming-based analytics engine [20, 70] and uses zero-copy for out-of-memory accesses. We also integrate the Kickstarter engine with unified memory (**KS-UM**), which allocates the

Table 2: Graph Statistics.

Graph	V	E	$Size_{org}$	$Size_{union}$
UK-2005 (UK) [61]	40M	1.6B	12GB	18GB
IT-2004 (IT) [61]	41M	2.1B	16GB	24GB
Twitter-2010 (TW) [39]	62M	2.4B	18GB	27GB
SK-2004 (SK) [61]	51M	3.7B	28GB	41GB
Subdomain (SD) [50]	102M	3.9B	30GB	44GB

Table 3: Graph Algorithms and their Edge Functions.

Alg. (Q)	$EdgeFunction(e(s,d))$
SSSP	*CASMIN*(dist(d), dist(s) + w(s,d))
BFS	*CASMIN*(level(d), level(s) + 1)
SSWP	*CASMAX*(wide(d), *min*(wide(s), w(s,d)))
Viterbi	*CASMAX*(vite(d), vite(s) / w(s,d))
SSNP	*CASMIN*(narrow(d), *max*(narrow(s), w(s,d))
WCC	*CASMIN*(id(d), id(s))

graph data with `cudaMallocManaged()`, and state-of-the-art explicit data transfer method Subway [62] (**KS-SW**).

- *Batch-based Incremental Analysis Approach*: we deploy **Mega** [17] on GPUs, which employs state-of-the-art *CommonGraph* [1] as analytics engines and reuses the delta transferred with Subway. We also integrate CommonGraph with unified memory (**CG-UM**) and zero-copy (**CG-ZC**).

Evaluation Platform. Experiments are mainly conducted on a server equipped with an NVIDIA RTX A4000 GPU with 16 GB memory and a 16-core Intel Xeon Gold 6426Y 2.50GHz processor with 256 GB RAM. The server runs on Ubuntu 20.04.6 with Linux kernel 5.15.0, and the codes are compiled with CUDA 12.4 with the highest optimization level. For large datasets, we extend the evaluation onto an NVIDIA A6000 Ada GPU with 48 GB memory, while keeping the remaining system configuration unchanged.

Graph Datasets. Evaluations were conducted with a set of real-world graphs from different domains. Table 2 lists these graphs with their original graph CSR sizes ($Size_{org}$) and their UnionCSR sizes ($Size_{union}$) for EGA, both with weights. All these graphs have their UnionCSR sizes exceeding the GPU memory, requiring out-of-GPU-memory graph processing. Following prior studies [45, 66, 70], we initialize and warm up the system by preloading 50% of each dataset as the base snapshot (G_0). Unless otherwise specified, all experimental results presented in this section are obtained over 32 snapshots; and each snapshot involves a delta batch of 0.1% updates, including 0.05% edge additions (randomly selected from the remaining dataset) and 0.05% edge deletions (randomly selected from the loaded portion).

Graph Algorithms. We use six types of graph queries to evaluate the graph analysis performance, including BFS (breadth-first search), SSSP (single source shortest path), SSWP (single source widest path), SSNP (single source narrowest path), Viterbi, and Weakly Connected Component (WCC). Table 3 lists their edge functions.

Table 4: Performance Results (Seconds).

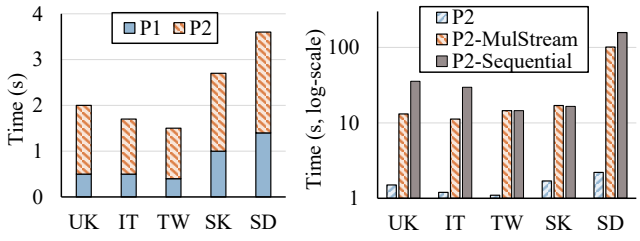
Alg.	**G.**	EGraph	KS-UM	KS-SW	Grapin	CG-UM	CG-ZC	Mega	PEGA
BFS	UK	172	5.8	14.9	8.6	32.4	33.7	18.0	1.4
	IT	205	22.5	20.9	13.1	78.3	55.3	16.4	2.1
	TW	164	78.5	25.4	21.2	88.1	44.8	17.5	1.8
	SK	537	80.4	25.3	18.2	102	85.7	32.5	2.3
	SD	859	194	24.3	12.7	178	71.7	51.3	3.2
SSSP	UK	446	17.7	40.3	18.8	58.9	63.2	17.9	2.0
	IT	341	38.8	45.7	13.2	162	84.0	17.5	1.7
	TW	182	92.8	39.6	15.6	103	39.0	53.9	1.5
	SK	767	97.2	49.3	33.0	187	135	33.2	2.7
	SD	1097	297	44.3	18.9	339	69.7	49.6	3.6
SSWP	UK	476	10.0	32.8	10.6	35.8	30.5	19.8	1.3
	IT	948	23.4	53.3	13.1	54.0	46.1	25.6	1.7
	TW	271	89.4	30.8	13.7	82.0	38.5	71.3	1.6
	SK	1749	96.4	66.6	27.9	123	83.5	70.1	2.8
	SD	3545	208	73.1	18.0	222	66.9	119	5.9
Viterbi	UK	474	17.1	8.5	25.9	63.8	34.3	16.3	1.1
	IT	923	37.9	41.6	23.9	15.9	52.4	27.1	1.8
	TW	276	94.6	28.0	21.2	11.3	38.1	71.7	1.5
	SK	1909	109	88.5	31.9	178	174	92.7	4.6
	SD	3473	274	50.7	19.1	276	66.8	127	4.7
SSNP	UK	288	4.0	17.4	8.6	30.6	30.5	14.0	1.2
	IT	994	22.5	51.6	13.9	56.2	45.3	27.2	1.8
	TW	274	89.1	22.5	14.3	83.0	38.4	72.7	1.6
	SK	1479	99.5	65.2	36.3	124	86.7	87.5	2.5
	SD	4318	202	65.4	15.9	201	74.3	125	3.7
WCC	UK	288	5.9	14.3	9.9	30.6	29.5	9.3	1.6
	IT	994	19.0	13.2	7.8	48.6	39.5	12.8	1.9
	TW	274	60.8	17.6	14.6	87.0	37.4	35.1	1.5
	SK	1479	68.3	22.2	19.5	92.1	69.0	32.5	2.7
	SD	4318	126	17.9	11.0	14.6	60.2	63.6	3.0
GMean Speedup (×)		**10.6**	**17.2**	**33.4**	**7.1**	**9.9**	**15.0**	**253.9**	

6.2 Overall Performance

Table 4 reports the overall performance of all the methods, as well as the geometric mean speedup over the baseline, EGraph. The results show that PEGA outperforms other systems across all datasets and algorithms, achieving 253.9× speedup over the baseline method EGraph. When compared with SOTA (the fastest system other than PEGA in each case), PEGA has the speedup of 6.6× on average, and at most 14.3× for SSNP on the SK dataset.

Specifically, against the streaming-based method (KS) with different GPU memory management, PEGA achieves 23.9×, 7.6×, 14.7× speedups over unified memory (UM), Grapin, and Subway (SW) in geometric mean, respectively. Besides, against the batch-based method (CG), PEGA achieves 35.8×, 25.5×, 17.0× speedups over UM, zero-copy (ZC), and Mega.

We observe that the streaming-based methods generally outperform batch-based methods, particularly when using zero-copy. This is due to the trade-off between parallelism and data efficiency. KS exploits the exact dependency tracing and result bootstrapping to minimize the data required for analysis. In contrast, CG requires much more data to support its parallel incremental analysis [17], which is even more expensive for



(a) P1 and P2 performance. (b) P2 performance under different execution strategies.

Figure 13: Two Phases Breakdown (SSSP).

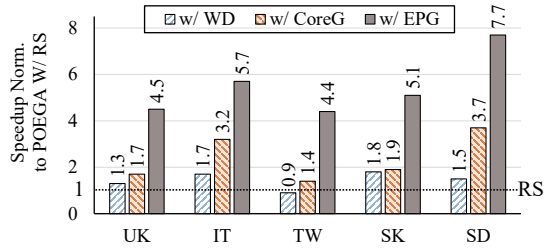


Figure 14: Effect of evolving proxy graph. Compared with POEGA integrating random sampling (RS), WD [85], or CoreGraph [32] (SSSP).

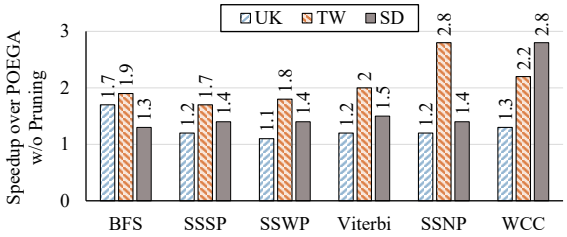


Figure 15: Effect of bound-based pruning.

the explicit data transfer (SW). The consistent speedups of POEGA over both categories confirm the effectiveness of its holistic approach for GPU memory management and parallel computation efficiency. Finally, all incremental systems show consistent speedups over recomputation-based EGraph, validating the fundamental advantage of result reuse in EGA.

6.3 Breakdown Analysis

To assess the contribution of the proposed components to the final results of POEGA, we conduct a breakdown analysis.

Breakdown of the Two Phases and Effect of Coalesced Concurrent Analysis. Figure 13(a) decomposes the execution latency of POEGA. The P1 phase, which performs in-memory EPG analysis, accounts for 24% to 39% of the total runtime, with an average of 31%. Although the P2 refinement phase remains the dominant cost, its latency is reduced to a level comparable to in-memory processing through our coalesced concurrent execution design. To further evaluate this, Figure 13(b) compares POEGA against two alternative strategies for P2: concurrent execution with CUDA stream [58] and sequential execution. Note that several techniques in POEGA, e.g., bound-based pruning, cannot be supported under a multi-stream execution. The coalesced strategy in POEGA achieves 13.6× and 21.9× speedups over these two baselines, respectively. These results validate the efficiency of data access optimization and computation reduction discussed in §4.3, and indicate that naive concurrency alone is insufficient to fully utilize GPU resources.

Effect of Evolving Proxy Graph. To evaluate the benefits of evolving proxy graph (EPG), we benchmark POEGA against three alternative proxy graph strategies: evolving proxy graph generated for each snapshot with (1) the random sampling (RS), (2) the heuristics in Wonderland (WD) [85] (e.g., edges with the least weights), both sized at 10% of E, and (3) a static Core Graph [32]. Results in Figure 14 show that EPG helps POEGA achieve average speedups of 5.4× over RS and 3.8× over WD. This is due to the higher accuracy of EPG (from 87.1% to 98.8%) compared to that of RS (10% to 26%) and WD (14% to 48%). Besides, EPG in POEGA outperforms the static core graph by 2.4×, despite the latter's high accuracy for the initial snapshot. This verifies the benefits of incremental analysis-oriented adaptation for updates. Besides, the proxy graphs in POEGA account for 12.1% to 15.9% of the original snapshot sizes, with a maintenance and generation cost of less than 0.3 seconds per snapshot. Note that this cost can be covered by the graph storage update procedure.

Effect of Bound-based Pruning. To evaluate the impact of the bound-based pruning, we disable it and then benchmark POEGA. The results in Figure 15 show diverse benefits of pruning on different graph datasets and algorithms. It improves the performance of POEGA by up to 2.8×, and by 1.6× on average. These results highlight the room for further improving the concurrent analysis by reducing redundant computations across snapshots, underscoring the effectiveness of our approach, including the lazy updates of the bounds.

Effect of Using AMVA. We evaluate the effectiveness of AMVA in scenarios where allocating full value arrays out of GPU memory (e.g., 32 snapshots of TW/SD). AMVA is compared against two alternative strategies: (1) batch-by-batch, which trades reduced concurrency to memory, and (2) zero-copy for managing out-of-memory value arrays. Results in Figure 16 show that AMVA outperforms batch-by-batch concurrent analysis up to 3.3×. This contributes to the benefits of snapshot coalescing with a higher concurrency. Notably, both AMVA and batch-by-batch analysis consistently outperform zero-copy, demonstrating the need for explicit runtime state management on GPUs.

The results in Figure 16 also present the impact of AMVA on two phases and yield two observations. First, AMVA delivers more gains at higher snapshot numbers (32 and 64), yet

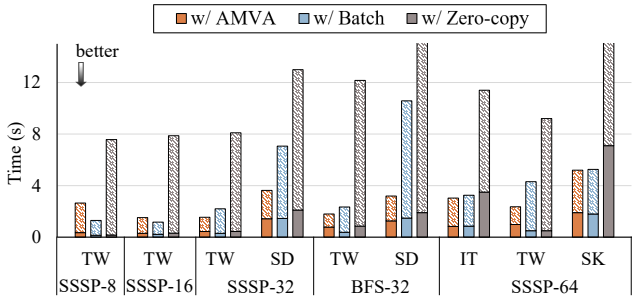


Figure 16: Effect of AMVA in two phases. Compared against reduced concurrency (batch-by-batch) and zero-copy for out-of-memory value array. (P1: Solid; P2: Patterned.)

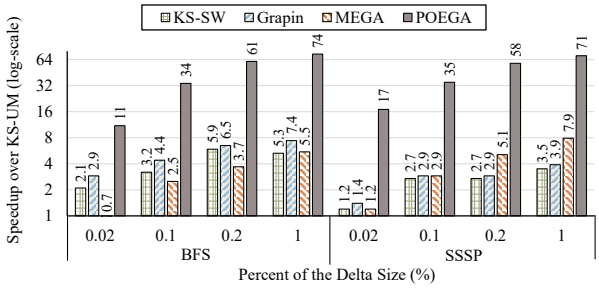


Figure 17: Effect of delta size (*SK*).

exhibits a performance degradation compared to the batch-by-batch method at smaller scales (8 and 16). This indicates that when potential concurrency is limited, the runtime overheads of AMVA outweigh its memory-saving advantages. Second, AMVA introduces a trade-off between the two phases. While it boosts P2 by up to 4.7× and 12.9× compared with batch-based and zero-copy, it results in only marginal gains, or even slight slowdowns for P1. This is because in-memory processing benefits relatively less from increased concurrency, which makes AMVA's computational overhead more visible.

6.4 Scalability

We also examine the scalability of POEGA. First, the delta batch size is set from 0.02% (0.01% add. and 0.01% del.) to 1% (0.5% add. and 0.5% del.) while keeping 32 snapshots. Speedups of POEGA and two state-of-the-art systems, Grapin and Mega, as well as the KS-SW, over KS-UM are reported in Figure 17. POEGA and batch-based incremental system Mega have consistently increasing speedups with larger delta sizes. Mega outperforms Grapin when the delta is 1% for SSSP, showing the scalability of batch-based incremental methods in terms of delta size.

We then vary the snapshot number from 8 to 64 while keeping the delta size at 0.1%. The results in Figure 18 show that POEGA generally has more benefits than other approaches across different N. Streaming-based methods, KS-SW and

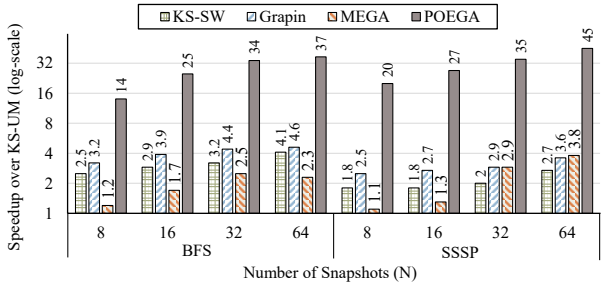


Figure 18: Effect of number of snapshots (*SK*).

Table 5: Performance results (seconds) for running over *SD* dataset on a 48GB GPU.

Systems	BFS	SSSP	SSWP	Viterbi	SSNP	WCC
EGraph	212.6	254.1	759.4	620.2	966.4	182.9
Grapin	10.9	16.0	14.1	16.1	11.3	9.7
Mega	12.0	21.4	52.9	55.4	60.5	18.6
POEGA	2.7	2.4	2.2	3.0	2.7	2.6

Grapin, have stable performance with different N, which is due to their sequential processing of delta for each new snapshot, regardless of the total snapshot numbers. KS-SW shows a fluctuating performance because it is sensitive to the effect of deletion deltas.

Results on a GPU with 48 GB memory. We further evaluate the scalability of POEGA on a GPU platform with increased memory capacity. Table 5 reports the performance of POEGA, the baseline, and the SOTA systems over the graph dataset *SD* on a 48 GB GPU. Note that despite the larger GPU memory, the runtime state and graph data of SD exceed 48 GB.

The results show that while increasing GPU memory to 48 GB improves performance for all systems, i.e., 4.2× for EGraph, 1.2× for Grapin, 2.8× for Mega and 1.6× for POEGA, the gains are sub-linear. These suggest that these systems are generally memory-bound, yet simply scaling the GPU memory size cannot fully mitigate the overhead of irregular, fine-grained out-of-memory graph data accesses typical in incremental analysis. Notably, POEGA maintains its benefits, achieving a 4.9× speedup over Grapin on SD, surpassing the 4.0× speedup observed in the 16 GB scenario.

7 Discussion and Future Work

Monotonic Graph Algorithm. POEGA currently supports graph algorithms that exhibit monotonicity, where *vertex values only change monotonically across iterations before convergence.* This property is common for a wide spectrum of iterative graph algorithms: beyond the fundamental algorithms presented in §6, it encompasses complex tasks such as graph radii estimation and minimum spanning tree. The significance of monotonicity has been extensively recognized and exploited in prior graph systems. It serves as the theoretical

foundation for for asynchronous iterative processing [62, 82], result reuse in dynamic graph processing [11, 16, 70, 83], automated parallelization [15], and iteration pruning [9, 32, 33, 80]. Furthermore, as existing studies have provided automated condition checkers to confirm whether a graph algorithm is monotonic [20], users can directly leverage them to verify whether an algorithm can be deployed in POEGA.

For non-monotonic graph algorithms, such as PageRank and Betweenness Centrality, POEGA could integrate the corresponding incremental approach, such as Graphbolt [46], to its analytics engine, while needing to adapt the bound-based pruning technique, which currently relies on the monotonicity. The efficiency of POEGA for non-monotonic graph algorithms is an open problem, and we leave the investigation of general incremental graph analysis on GPUs as our future work.

Extending POEGA to Multi-GPU Environments. Beyond single GPU, POEGA can be extended to multi-GPU scaling. In this scenario, snapshots can be distributed across multiple GPUs by batch, with each batch still using the coalesced concurrency analysis. Since the resident-memory proxy graph is compact, it can be replicated on each device to enable independent, zero-communication approximate analysis in P1. Besides, future work could exploit the distinct bottleneck profiles of POEGA's two phases to overlap P1's compute-intensive tasks with P2's I/O-intensive transfers across devices, further mitigating the PCIe/NVLink overhead.

8 Related Work

Evolving Graph Systems. Various incremental computation frameworks have been proposed. Early approaches such as Tornado [66] and Naiad [54] explore the incremental analysis concept, but only support edge additions. Kickstarter [70] designs the runtime support for the analysis to enable the capability of handling deltas with deletions, while Graphbolt [46] further supports the Bulk Synchronous Parallel semantics on the basis of Kickstarter. Tegra [30] extends streaming analysis to arbitrary time windows of the graph. CommonGraph [1] proposes to transform the expensive deletion processing into addition processing by first analyzing the common subgraphs and then incrementally reaching the full snapshots. MEGA [17] is an evolving graph accelerator that improves data reuse in CommonGraph by optimizing the scheduling of the incremental analysis. More recently, UVV [2] further reduces computations in CommonGraph by identifying vertices with stable values across snapshots. However, UVV relies on *offline* evaluation over the union of snapshots, which incurs significant overhead for large-scale, out-of-core processing. POEGA departs from this by introducing *online* stable value identification. Through a lightweight (yet safe) approximation for bound-based pruning, it achieves similar computational savings without heavy pre-processing.

A few works have explored improving the data representation of Kickstarter. Risgraph [16] optimizes for the per-update incremental analysis scenario, proposing the hash index for the adjacency list of graph data to enhance access efficiency. IncBoost [83] discusses the efficiency of dependency data representations in Kickstarter. All these systems and formats are built on multi-core or distributed CPUs.

GPU-based Graph Processing. Many GPU-based systems are designed for a specific graph algorithm [6, 26, 67, 71]. Among the general frameworks, Gunrock [75] introduces the frontier-centric model and implements a number of graph primitives. Maximum warp [27], CuSha [36], warp segmentation [35] and Tigr [57] focus on addressing GPU thread divergence issues for irregular graphs. These methods are orthogonal to the techniques of this work.

To address GPU memory oversubscription, early work Graphie [22] and GraphReduce [63] adopt the partition-based method and actively track the active vertices on partitions to avoid unnecessary partition transfers. EMOGI [51] explores efficiently employing zero-copy for graph analytics. Subway [62] proposes to transfer fine-grained partitions of the active vertices. All of these works focus on the memory oversubscription issue on the *static graphs*.

GPU-based *dynamic graph* systems are built to meet the real-time response requirement for the applications on rapidly evolving graphs. EGraph [86] proposes to share partitions among snapshots in dynamic graphs to reduce the redundant data transfer to devices. Grapin [74] adopts zero-copy for its streaming incremental engine (based on Kickstarter) and caches the hot vertices' subgraph in the GPU to reduce potential data transfer overhead. CGCGraph [68] transfers active shared partitions to GPUs while processing unshared active parts on the CPU concurrently. Aside from analytics, a parallel line of work has explored dynamic graph data structures or storage on GPUs [5, 7, 21, 65, 77].

9 Conclusion

This work presents POEGA, a framework designed to enable efficient large-scale evolving graph analytics on GPUs. It adopts a proxy-guided incremental analysis model to reduce out-of-memory IO, which first analyzes the proxy graphs and then executes global refinement. To solve the extra computations, POEGA employs an efficient concurrent analysis on GPUs, including bound-based pruning and adaptive state format to enable high concurrency. A comprehensive evaluation of POEGA demonstrates its effectiveness.

Acknowledgment

We are grateful to the anonymous reviewers and our shepherd for their constructive comments. This work was supported in part by the Research Grants Council of Hong Kong (No. 11216925 and 14212425), by City University of Hong Kong internal and donation fundings (No. 9610598,

No. 9220148, and No. 7005991), and by CUHK (4937007, 4937008, 5501329, 5501517).

References

[1] Mahbod Afarin, Chao Gao, Shafiur Rahman, Nael Abu-Ghazaleh, and Rajiv Gupta. Commongraph: Graph analytics on evolving data. In *Proceedings of the ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2023.

[2] Mahbod Afarin, Chao Gao, Xizhe Yin, Zhijia Zhao, Nael Abu-Ghazaleh, and Rajiv Gupta. Uvvs: Identifying unchanged vertex values in evolving graphs via intersection-union analysis. In *40th IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 2026.

[3] Neha Agarwal, David Nellans, Mark Stephenson, Mike O'Connor, and Stephen W Keckler. Page placement strategies for gpus within heterogeneous memory systems. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 607–618, 2015.

[4] Rachata Ausavarungnirun, Joshua Landgraf, Vance Miller, Saugata Ghose, Jayneel Gandhi, Christopher J Rossbach, and Onur Mutlu. Mosaic: a gpu memory manager with application-transparent support for multiple page sizes. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 136–150, 2017.

[5] Muhammad A Awad, Saman Ashkiani, Serban D Porumbescu, and John D Owens. Dynamic graphs on the gpu. In *2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 739–748. IEEE, 2020.

[6] Federico Busato and Nicola Bombieri. An efficient implementation of the bellman-ford algorithm for kepler gpu architectures. *IEEE Transactions on Parallel and Distributed Systems*, 2015.

[7] Federico Busato, Oded Green, Nicola Bombieri, and David A Bader. Hornet: An efficient data structure for dynamic sparse graphs and matrices on gpus. In *2018 IEEE High Performance extreme Computing Conference (HPEC)*, pages 1–7. IEEE, 2018.

[8] Jaewook Byun, Sungpil Woo, and Daeyoung Kim. Chronograph: Enabling temporal graph traversals for efficient information diffusion analysis over time. *IEEE Transactions on Knowledge and Data Engineering*, 32(3):424–437, 2019.

[9] Hongtao Chen, Mingxing Zhang, Ke Yang, Kang Chen, Albert Zomaya, Yongwei Wu, and Xuehai Qian. Achieving sub-second pairwise query over evolving graphs. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 1–15, 2023.

[10] Hongzheng Chen, Minghua Shen, Nong Xiao, and Yutong Lu. Krill: a compiler and runtime system for concurrent graph processing. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–16, 2021.

[11] Zheng Chen, Feng Zhang, Yang Chen, Xiaokun Fang, Guanyu Feng, Xiaowei Zhu, Wenguang Chen, and Xiaoyong Du. Enabling window-based monotonic graph analytics with reusable transitional results for pattern-consistent queries. *Proceedings of the VLDB Endowment*, 17(11):3003–3016, 2024.

[12] Yongli Cheng, Yan Ma, Hong Jiang, Lingfang Zeng, Fang Wang, Xianghao Xu, and Yuhang Wu. Tgstore: An efficient storage system for large time-evolving graphs. *IEEE Transactions on Big Data*, 2024.

[13] Ariel Debrouvier, Eliseo Parodi, Matías Perazzo, Valeria Soliani, and Alejandro Vaisman. A model and query language for temporal graph databases. *The VLDB Journal*, 30(5):825–858, 2021.

[14] Laxman Dhulipala, Guy E Blelloch, and Julian Shun. Low-latency graph streaming using compressed purely-functional trees. In *Proceedings of the ACM SIGPLAN conference on Programming Language Design and Implementation (PLDI)*, 2019.

[15] Wenfei Fan, Jingbo Xu, Yinghui Wu, Wenyuan Yu, Jiaxin Jiang, Zeyu Zheng, Bohan Zhang, Yang Cao, and Chao Tian. Parallelizing sequential graph computations. In *Proceedings of the 2017 ACM International Conference on Management of Data*, pages 495–510, 2017.

[16] Guanyu Feng, Zixuan Ma, Daixuan Li, Shengqi Chen, Xiaowei Zhu, Wentao Han, and Wenguang Chen. Risgraph: A real-time streaming system for evolving graphs to support sub-millisecond per-update analysis at millions ops/s. In *Proceedings of the International Conference on Management of Data (SIGMOD)*, 2021.

[17] Chao Gao, Mahbod Afarin, Shafiur Rahman, Nael Abu-Ghazaleh, and Rajiv Gupta. Mega evolving graph accelerator. In *IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2023.

[18] Hongru Gao, Xiaofei Liao, Zhiyuan Shao, Kexin Li, Jiajie Chen, and Hai Jin. A survey on dynamic graph

processing on gpus: concepts, terminologies and systems. *Frontiers of Computer Science*, 18(4):184106, 2024.

[19] Prasun Gera, Hyojong Kim, Piyush Sao, Hyesoon Kim, and David Bader. Traversing large graphs on gpus with unified memory. *Proceedings of the VLDB Endowment*, 13(7):1119–1133, 2020.

[20] Shufeng Gong, Chao Tian, Qiang Yin, Wenyuan Yu, Yanfeng Zhang, Liang Geng, Song Yu, Ge Yu, and Jingren Zhou. Automating incremental graph processing with flexible memoization. *Proceedings of the VLDB Endowment*, 14(9):1613–1625, 2021.

[21] Oded Green and David A Bader. custinger: Supporting dynamic graph algorithms for gpus. In *2016 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–6. IEEE, 2016.

[22] Wei Han, Daniel Mawhirter, Bo Wu, and Matthew Buland. Graphie: Large-scale asynchronous graph traversals on just a gpu. In *2017 26th International Conference on Parallel Architectures and Compilation Techniques (PACT)*. IEEE, 2017.

[23] Wentao Han, Youshan Miao, Kaiwei Li, Ming Wu, Fan Yang, Lidong Zhou, Vijayan Prabhakaran, Wenguang Chen, and Enhong Chen. Chronos: a graph engine for temporal graph analysis. In *Proceedings of the Sixteenth European Conference on Computer Systems (EuroSys)*, 2014.

[24] Pawan Harish and Petter J Narayanan. Accelerating large graph algorithms on the gpu using cuda. In *International conference on high-performance computing*, pages 197–208. Springer, 2007.

[25] Bruce Hendrickson and Robert Leland. A multilevel algorithm for partitioning graphs. In *Proceedings of the 1995 ACM/IEEE conference on Supercomputing*, pages 28–es, 1995.

[26] Changwan Hong, Laxman Dhulipala, and Julian Shun. Exploring the design space of static and incremental graph connectivity algorithms on gpus. In *Proceedings of the ACM International Conference on Parallel Architectures and Compilation Techniques*, pages 55–69, 2020.

[27] Sungpack Hong, Sang Kyun Kim, Tayo Oguntebi, and Kunle Olukotun. Accelerating cuda graph algorithms at maximum warp. In *Proceedings of the 16th ACM symposium on Principles and practice of parallel programming*, pages 267–276, 2011.

[28] Chengying Huan, Shuaiwen Leon Song, Santosh Pandey, Hang Liu, Yongchao Liu, Baptiste Lepers, Changhua He, Kang Chen, Jinlei Jiang, and Yongwei Wu. Tea: A general-purpose temporal graph random walk engine. In *Proceedings of the Eighteenth European Conference on Computer Systems*, pages 182–198, 2023.

[29] Anand Iyer, Li Erran Li, and Ion Stoica. CellIQ: Real-Time Cellular Network Analytics at Scale. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2015.

[30] Anand Padmanabha Iyer, Qifan Pu, Kishan Patel, Joseph E Gonzalez, and Ion Stoica. TEGRA: Efficient Ad-Hoc Analytics on Evolving Graphs. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2021.

[31] Jiaxin Jiang, Yuan Li, Bingsheng He, Bryan Hooi, Jia Chen, and Johan Kok Zhi Kang. Spade: A real-time fraud detection framework on evolving graphs. *Proceedings of the VLDB Endowment*, 16(3):461–469, 2022.

[32] Xiaolin Jiang, Mahbod Afarin, Zhijia Zhao, Nael Abu-Ghazaleh, and Rajiv Gupta. Core graph: Exploiting edge centrality to speedup the evaluation of iterative graph queries. In *Proceedings of the Nineteenth European Conference on Computer Systems*, pages 18–32, 2024.

[33] Xiaolin Jiang, Chengshuo Xu, Xizhe Yin, Zhijia Zhao, and Rajiv Gupta. Tripoline: generalized incremental graph processing via graph triangle inequality. In *Proceedings of the Sixteenth European Conference on Computer Systems (Eurosys)*, pages 17–32, 2021.

[34] Jens Kehne, Jonathan Metter, and Frank Bellosa. Gpuswap: Enabling oversubscription of gpu memory through transparent swapping. In *Proceedings of the 11th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, pages 65–77, 2015.

[35] Farzad Khorasani, Rajiv Gupta, and Laxmi N Bhuyan. Scalable simd-efficient graph processing on gpus. In *2015 International Conference on Parallel Architecture and Compilation (PACT)*, pages 39–50. IEEE, 2015.

[36] Farzad Khorasani, Keval Vora, Rajiv Gupta, and Laxmi N Bhuyan. Cusha: vertex-centric graph processing on gpus. In *Proceedings of the 23rd international symposium on High-performance parallel and distributed computing*, 2014.

[37] Pradeep Kumar and H Howie Huang. GRAPHONE: a data store for real-time analytics on evolving graphs. In *USENIX Conference on File and Storage Technologies (FAST)*, 2019.

[38] Amlan Kusum, Keval Vora, Rajiv Gupta, and Iulian Neamtiu. Efficient processing of large graphs via input reduction. In *Proceedings of the 25th ACM International Symposium on High-Performance Parallel and Distributed Computing*, pages 245–257, 2016.

[39] Haewoon Kwak, Changhyun Lee, Hosung Park, and Sue Moon. What is Twitter, a social network or a news media? In *Proc. ACM World Wide Web (WWW)*, 2010.

[40] Aapo Kyrola, Guy Blelloch, and Carlos Guestrin. GraphChi: Large-Scale graph computation on just a PC. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2012.

[41] Jure Leskovec and Christos Faloutsos. Sampling from large graphs. In *Proceedings of the 12th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 631–636, 2006.

[42] Changji Li, Hongzhi Chen, Shuai Zhang, Yingqian Hu, Chao Chen, Zhenjie Zhang, Meng Li, Xiangchen Li, Dongqing Han, Xiaohui Chen, et al. Bytegraph: a high-performance distributed graph database in bytedance. *Proceedings of the VLDB Endowment*, 15(12):3306–3318, 2022.

[43] Chen Li, Rachata Ausavarungnirun, Christopher J Rossbach, Youtao Zhang, Onur Mutlu, Yang Guo, and Jun Yang. A framework for memory oversubscription management in graphics processing units. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 49–63, 2019.

[44] Yunkai Lou, Chaokun Wang, Tiankai Gu, Hao Feng, Jun Chen, and Jeffrey Xu Yu. Time-topology analysis on temporal graphs. *The VLDB Journal*, 32(4):815–843, 2023.

[45] Peter Macko, Virendra J Marathe, Daniel W Margo, and Margo I Seltzer. LLAMA: Efficient Graph Analytics using Large Multiversioned Arrays. In *IEEE International Conference on Data Engineering (ICDE)*, 2015.

[46] Mugilan Mariappan and Keval Vora. Graphbolt: Dependency-driven synchronous processing of streaming graphs. In *Proceedings of the Sixteenth European Conference on Computer Systems (EuroSys)*, 2019.

[47] Maria Massri, Zoltan Miklos, Philippe Raipin, and Pierre Meye. Clock-g: A temporal graph management system with space-efficient storage technique. In *2022 IEEE 38th International Conference on Data Engineering (ICDE)*, pages 2263–2276, 2022.

[48] Frank McSherry, Derek Gordon Murray, Rebecca Isaacs, and Michael Isard. Differential Dataflow. In *CIDR*, 2013.

[49] Duane Merrill, Michael Garland, and Andrew Grimshaw. Scalable gpu graph traversal. In *Proceedings of the 17th ACM SIGPLAN symposium on Principles and Practice of Parallel Programming*, pages 117–128, 2012.

[50] Robert Meusel, Sebastiano Vigna, Oliver Lehmberg, and Christian Bizer. The graph structure in the web–analyzed on different aggregation levels. *The Journal of Web Science*, 1, 2015.

[51] Seung Won Min, Vikram Sharma Mailthody, Zaid Qureshi, Jinjun Xiong, Eiman Ebrahimi, and Wen-mei Hwu. Emogi: efficient memory-access for out-of-memory graph-traversal in gpus. *Proc. VLDB Endow.*, 14(2):114–127, 2020.

[52] Srikanth Mithinti and Prashant Singh. Gpu algorithms for fastest path problem in temporal graphs. In *Proceedings of the 53rd International Conference on Parallel Processing*, pages 587–596, 2024.

[53] Irene Moulitsas and George Karypis. Multilevel algorithms for generating coarse grids for multigrid methods. In *Proceedings of the 2001 ACM/IEEE conference on Supercomputing*, pages 45–45, 2001.

[54] Derek G Murray, Frank McSherry, Rebecca Isaacs, Michael Isard, Paul Barham, and Martín Abadi. Naiad: a timely dataflow system. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 439–455, 2013.

[55] Nurlan Nazaraliyev, Elaheh Sadredini, and Nael Abu-Ghazaleh. Dream: Device-driven efficient access to virtual memory. In *Proceedings of the 39th ACM International Conference on Supercomputing*, pages 1190–1205, 2025.

[56] Abdun Nihaal and Madhu Mutyam. Selective memory compression for gpu memory oversubscription management. In *Proceedings of the 53rd International Conference on Parallel Processing*, pages 189–198, 2024.

[57] Amir Hossein Nodehi Sabet, Junqiao Qiu, and Zhijia Zhao. Tigr: Transforming Irregular Graphs for GPU-Friendly Graph Processing. In *Proceedings of the ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2018.

[58] NVIDIA. CUDA C++ Programming Guide. https://docs.nvidia.com/cuda/cuda-c-programming-guide/.

[59] NVIDIA. Cuda c++ best practices guide. https://docs.nvidia.com/cuda/cuda-c-best-practices-guide/index.html, 2020.

[60] Chenghui Ren, Eric Lo, Ben Kao, Xinjie Zhu, and Reynold Cheng. On querying historical evolving graph sequences. *Proceedings of the VLDB Endowment*, 2011.

[61] Ryan Rossi and Nesreen Ahmed. The network data repository with interactive graph analytics and visualization. In *Proceedings of the AAAI conference on artificial intelligence*, volume 29, 2015.

[62] Amir Hossein Nodehi Sabet, Zhijia Zhao, and Rajiv Gupta. Subway: Minimizing data transfer during out-of-gpu-memory graph processing. In *Proceedings of the Fifteenth European Conference on Computer Systems*, 2020.

[63] Dipanjan Sengupta, Shuaiwen Leon Song, Kapil Agarwal, and Karsten Schwan. Graphreduce: processing large-scale graphs on accelerator-based systems. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2015.

[64] Sangmin Seo, Junghyun Kim, and Jaejin Lee. Sfmalloc: A lock-free and mostly synchronization-free dynamic memory allocator for manycores. In *2011 International Conference on Parallel Architectures and Compilation Techniques*, pages 253–263. IEEE, 2011.

[65] Mo Sha, Yuchen Li, Bingsheng He, and Kian-Lee Tan. Accelerating dynamic graph analytics on gpus. *Proceedings of the VLDB Endowment*, 11(1):107–120, 2017.

[66] Xiaogang Shi, Bin Cui, Yingxia Shao, and Yunhai Tong. Tornado: A system for Real-time Iterative Analysis over Evolving Data. In *Proceedings of the International Conference on Management of Data (SIGMOD)*, 2016.

[67] Jyothish Soman, Kothapalli Kishore, and PJ Narayanan. A fast gpu algorithm for graph connectivity. In *2010 IEEE International Symposium on Parallel & Distributed Processing, Workshops and Phd Forum (IPDPSW)*. IEEE, 2010.

[68] Yiming Sun, Jie Zhang, Huawei Cao, Yuan Zhang, Xuejun An, Junying Huang, and Xiaochun Ye. Cgcgraph: Efficient cpu-gpu co-execution for concurrent dynamic graph processing. *ACM Transactions on Architecture and Code Optimization*, 2025.

[69] Hau Tran, An Nguyen, Phuong Vo, and Tu Vu. Dns graph mining for malicious domain detection. In *2017 IEEE International Conference on Big Data (Big Data)*, pages 4680–4685. IEEE, 2017.

[70] Keval Vora, Rajiv Gupta, and Guoqing Xu. Kickstarter: Fast and Accurate Computations on Streaming Graphs via Trimmed Approximations. In *Proceedings of the ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2017.

[71] Kai Wang, Don Fussell, and Calvin Lin. A fast work-efficient sssp algorithm for gpus. In *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 133–146, 2021.

[72] Kai Wang, Guoqing Xu, Zhendong Su, and Yu David Liu. {GraphQ}: Graph query processing with abstraction {Refinement—Scalable} and programmable analytics over very large graphs on a single {PC}. In *2015 USENIX Annual Technical Conference (USENIX ATC 15)*, pages 387–401, 2015.

[73] Qiange Wang, Xin Ai, Yanfeng Zhang, Jing Chen, and Ge Yu. Hytgraph: Gpu-accelerated graph processing with hybrid transfer management. In *2023 IEEE 39th International Conference on Data Engineering (ICDE)*, pages 558–571. IEEE, 2023.

[74] Qiange Wang, Yongze Yan, Hongshi Tan, Cheng Chen, Cheng Zhao, Jiaming Tian, Jiaxin Jiang, Xiaoliang Cong, Yanfeng Zhang, Ge Yu, et al. Efficient graph data access for out-of-memory gpu streaming graph processing. *Proceedings of the VLDB Endowment*, 18(11):3854–3867, 2025.

[75] Yangzihao Wang, Andrew Davidson, Yuechao Pan, Yuduo Wu, Andy Riffel, and John D Owens. Gunrock: A high-performance graph processing library on the GPU. In *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 2016.

[76] Ye Wang, Qing Wang, Henning Koehler, and Yu Lin. Query-by-sketch: Scaling shortest path graph queries on very large networks. In *Proc. International Conference on Management of Data (SIGMOD)*, 2021.

[77] Martin Winter, Daniel Mlakar, Rhaleb Zayer, Hans-Peter Seidel, and Markus Steinberger. faimgraph: High performance management of fully-dynamic graphs under tight memory constraints on the gpu. In *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 754–766. IEEE, 2018.

[78] Huanhuan Wu, James Cheng, Silu Huang, Yiping Ke, Yi Lu, and Yanyan Xu. Path problems in temporal graphs. *Proceedings of the VLDB Endowment*, 7(9):721–732, 2014.

[79] Mincheng Wu, Chao Li, Zhangchong Shen, Shibo He, Lingling Tang, Jie Zheng, Yi Fang, Kehan Li, Yanggang Cheng, Zhiguo Shi, et al. Use of temporal contact graphs

to understand the evolution of COVID-19 through contact tracing data. *Communications Physics*, 5(1):270, 2022.

[80] Chengshuo Xu, Keval Vora, and Rajiv Gupta. Pnp: Pruning and prediction for point-to-point iterative graph analytics. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 587–600, 2019.

[81] Chang Ye, Yuchen Li, Bingsheng He, Zhao Li, and Jianling Sun. Gpu-accelerated graph label propagation for real-time fraud detection. In *Proceedings of the 2021 International Conference on Management of Data*, pages 2348–2356, 2021.

[82] Xizhe Yin, Zhijia Zhao, and Rajiv Gupta. Glign: Taming misaligned graph traversals in concurrent graph processing. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2023.

[83] Xizhe Yin, Zhijia Zhao, and Rajiv Gupta. Incboost: Scaling incremental graph processing for edge deletions and weight updates. In *Proceedings of the 2024 ACM Symposium on Cloud Computing*, 2024.

[84] Aya Zaki, Mahmoud Attia, Doaa Hegazy, and Safaa Amin. Comprehensive survey on dynamic graph models. *International Journal of Advanced Computer Science and Applications*, 7(2), 2016.

[85] Mingxing Zhang, Yongwei Wu, Youwei Zhuo, Xuehai Qian, Chengying Huan, and Kang Chen. Wonderland: A novel abstraction-based out-of-core graph processing system. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2018.

[86] Yu Zhang, Yuxuan Liang, Jin Zhao, Fubing Mao, Lin Gu, Xiaofei Liao, Hai Jin, Haikun Liu, Song Guo, Yangqing Zeng, et al. Egraph: efficient concurrent gpu-based dynamic graph processing. *IEEE Transactions on Knowledge and Data Engineering*, 35(6):5823–5836, 2022.

[87] Yu Zhang, Xiaofei Liao, Hai Jin, Lin Gu, Ligang He, Bingsheng He, and Haikun Liu. {CGraph}: A correlations-aware approach for efficient concurrent iterative graph processing. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 441–452, 2018.

[88] Yunmo Zhang, Jiacheng Huang, Xizhe Yin, Junqiao Qiu, Hong Xu, and Chun Jason Xue. Pie: Enabling fast and scalable incremental evolving graph analytics on persistent memory. In *Proceedings of the 39th ACM International Conference on Supercomputing*, pages 564–579, 2025.

[89] Jin Zhao, Qian Wang, Ligang He, Yu Zhang, Sheng Di, Bingsheng He, Xinlei Wang, Hui Yu, Hao Qi, Longlong Lin, et al. Tempgraph: An efficient chain-driven temporal graph computing framework on the gpu. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, pages 230–246, 2025.

[90] Jin Zhao, Yu Zhang, Xiaofei Liao, Ligang He, Bingsheng He, Hai Jin, Haikun Liu, and Yicheng Chen. Graphm: an efficient storage system for high throughput of concurrent graph processing. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–14, 2019.

[91] Long Zheng, Xianliang Li, Yaohui Zheng, Yu Huang, Xiaofei Liao, Hai Jin, Jingling Xue, Zhiyuan Shao, and Qiang-Sheng Hua. Scaph: Scalable {GPU-Accelerated} graph processing with {Value-Driven} differential scheduling. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pages 573–588, 2020.

[92] Fang Zhou, Sebastien Malher, and Hannu Toivonen. Network simplification with minimal loss of connectivity. In *2010 IEEE international conference on data mining*, pages 659–668. IEEE, 2010.

[93] Xiaowei Zhu, Wentao Han, and Wenguang Chen. {GridGraph}:{Large-Scale} graph processing on a single machine using 2-level hierarchical partitioning. In *USENIX Annual Technical Conference (ATC)*, 2015.

[94] Lei Zou, Fan Zhang, Yinnian Lin, and Yanpeng Yu. An efficient data structure for dynamic graph on gpus. *IEEE Transactions on Knowledge and Data Engineering*, 35(11):11051–11066, 2023.

A Artifact Appendix

Abstract

This artifact provides the source code, benchmarks, and evaluation scripts for POEGA, a high-performance framework designed to enable efficient large-scale evolving graph analytics on GPUs. It includes the implementation of proxy-guided two-phase analysis, bound-based pruning, and adaptive multi-version array.

Scope

This artifact enables validation of POEGA's performance against seven compared systems mentioned in §6. Refer to the artifact's README for specific experiments.

Contents

The artifact includes core source code (`src/`), dataset preprocessing utilities (`tools/`), and the script for running all experiments (`run.sh`). The README details the full directory structure and contents.

Hosting

The artifact of POEGA is hosted at https://github.com/YunMoZhang/POEGA. Use the latest commit on the `master` branch.

Requirements

CUDA Toolkit ($\geq$ 12.4) and CMake ($\geq$ 3.18) are required. Evaluated on an NVIDIA RTX A4000 GPU and NVIDIA A6000 Ada GPU. Refer to the README for comprehensive requirements.

Pluto: High-Performance, Memory-Efficient Distributed Graph Analytics Through Advanced Mirroring

Ying-Wei Wu
University of Texas at Austin

Christopher J. Rossbach
University of Texas at Austin
Microsoft

Mattan Erez
University of Texas at Austin

Abstract

Inter-host communication poses a significant performance bottleneck in distributed graph analytics due to synchronization overheads. To mitigate this, state-of-the-art systems typically employ the full mirroring technique to replicate all potentially needed remote data and adopt a bulk-synchronous parallel execution model for coarse-grained communication. While effective for reducing network traffic, these approaches substantially increase memory footprint and constrain system parallelism. This paper introduces Pluto, a memory-efficient distributed graph analytics system based on two advanced mirroring designs: static partial mirroring and a mirror-free architecture. Nonproductive data duplication is avoided to reduce memory usage, while the work migration mechanism allows communication-computation overlap for performance improvement. For homogeneous graphs, Pluto achieves up to 3.8× speedup (harmonic mean 1.75×) compared to a full mirroring baseline and delivers up to 12× speedup (harmonic mean 1.75×) over existing open-source systems. For labeled property graphs, Pluto achieves up to 2.6× speedup (harmonic mean 1.37×) and lowers the minimum host requirement to 50%–90% of the baseline.

1 Introduction

The widespread adoption of social media and search engines has driven significant interest in graph-based analysis of social networks and web crawls. These datasets are massive, often ranging from terabytes to petabytes, making single-node storage and processing impractical. As a result, distributed frameworks are developed to partition graphs across clusters of nodes and execute in parallel.

A fundamental challenge in distributed systems is inter-node communication, which is frequently constrained by network latency and limited interconnect bandwidth. To mitigate communication costs, state-of-the-art systems [11, 14, 19, 20, 48] commonly combine the *mirroring* approach with bulk-synchronous parallel (BSP) execution. Mirroring replicates remote data to reduce communication during execution, and the BSP model combines multiple updates for coarse-grained synchronization.

While mirroring reduces communication volume, it increases memory usage. Given historically abundant memory, the prevailing practice has been to replicate all potentially required remote data, an approach we call *full mirroring*. However, with the slowing of Moore's law [33] and memory capacity scaling lagging behind data growth [6], optimizing solely for performance is no longer a sustainable solution as memory becomes the limiting resource. In our study, full mirroring incurs up to 4× memory overhead and can be even larger on highly connected graphs, questioning the long-term viability of this conventional dogma under emerging trends.

We address this challenge through two advanced mirroring designs: *static partial mirroring* and a *mirror-free* architecture. Static partial mirroring leverages *mirror heterogeneity*, the observation that not all data replication contribute productively to execution. By identifying and eliminating the nonproductive duplications, the memory usage is reduced while maintaining or even improving performance. The mirror-free architecture targets algorithmic variants that operate efficiently without any mirroring, and thus completely removes the memory overhead of data duplication. Together, the two designs cover a broad range of common graph applications.

Building on these ideas, we present Pluto, a memory-efficient distributed graph analytics system. By lowering memory requirements, Pluto enables the same applications to run on fewer nodes. In addition, Pluto migrates work to the nodes that hold the relevant data while continuing local computation, overlapping communication with computation to hide network latency and improve performance.

This work makes the following contributions:

- We identify *mirror heterogeneity* and propose a *static partial mirroring* approach that detects and avoids nonproductive data replication.
- We analyze algorithmic variants that operate efficiently without remote data replicas and design a *mirror-free*

architecture to remove duplication overhead entirely.

- We introduce an *on-demand work migration* mechanism that initiates network transfers early and overlaps communication with computation, amortizing communication costs.

- We implement these techniques on top of D-Galois [14] to build Pluto, a high-performance and memory-efficient distributed graph analytics system. Across large-scale homogeneous graphs and standard benchmarks, Pluto outperforms state-of-the-art systems by up to 12× (harmonic mean 2.5×) and delivers 3.8× speedup (harmonic mean 1.75×) over our own full mirroring baseline. For labeled property graphs, Pluto achieves up to 2.6× speedup (harmonic mean 1.37×) while enabling the same graphs to be stored on only 50%–90% of the hosts required by existing systems.

2 Background

2.1 Graph Data

A basic graph is defined as $G = (V, E)$, where V is a set of vertices and E is a set of edges. This topology-only representation, often called a *plain* or *homogeneous* graph, is widely used in academic research for algorithmic studies that focus on geometry.

The *labeled property graph (LPG)* [4], which is the most dominant data model in modern graph databases [1, 24, 31, 34, 42], augments G by associating labels and properties with both vertices and edges. Labels classify graph elements into meaningful categories, and properties attach attributes with arbitrary data structures to each element. This combination of flexibility and expressiveness allows the LPG model to represent real-world domains in a rich and intuitive manner.

2.2 Graph Analytics

This work focuses on graph analytics, a class of algorithms that analyzes the topology of graph-structured data to uncover patterns and insights.

2.2.1 Programming Models

Most graph analytics systems [11, 14, 19, 20, 28, 48] adopt a vertex-centric model: each vertex maintains a state, and each algorithm defines a local computational rule to update the vertex state with connecting edges and immediate (one-hop) neighbor values. An alternative line of work [2, 7, 8, 32] operates on the adjacency matrix representation of graphs through linear algebra. Hochan et al. report that the vertex-centric paradigm outperforms the linear-algebra-based approach across a range of graph analytics workloads [26]. Accordingly, all five benchmarks in our evaluation are implemented using the vertex-centric model.

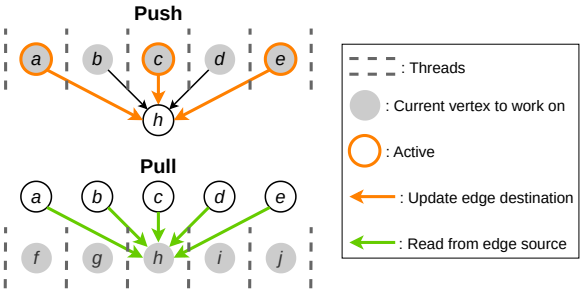


Figure 1: The push and pull update models.

2.2.2 Update Models

Fig. 1 illustrates the two primary update models used in vertex-centric programs: push and pull. In the push model, a vertex becomes active when its own state satisfies conditions specified by the algorithm and then propagates updates along outgoing edges to destinations. In the pull model, a vertex collects values from all sources of incoming edges and jointly decides whether to update its own state. Execution proceeds iteratively according to the chosen model until the graph reaches a global quiescent state with no further updates.

The push model only touches outgoing edges of active vertices, but the pull model scans incoming neighbors of all vertices uniformly. In multithreaded execution, push requires atomic operations at edge destinations because multiple threads may concurrently update the same vertex, whereas pull aggregates incoming neighbor values for a vertex sequentially and applies the corresponding update within a single thread. Overall, push can reduce memory operations when the number of active vertices is small, while pull can avoid contention and benefit from contiguous reads when memory activity is dense. The optimal choice typically depends on the sparsity of active vertices.

Typical graph analytics systems employ a fixed update model for the duration of execution. Ligra [39] and Gemini [48] implement hybrid strategies that adaptively switch between push and pull based on the active vertex set size per round. However, a practical implication is the memory overhead: supporting both directions efficiently requires maintaining outgoing and incoming edges simultaneously, which doubles the edge list storage.

2.3 Distributed Graph Processing

2.3.1 Graph Partitioning

The first step in distributed graph processing is to partition the graph across hosts. We follow conventions by using the term “host” to refer to each compute node and retain this

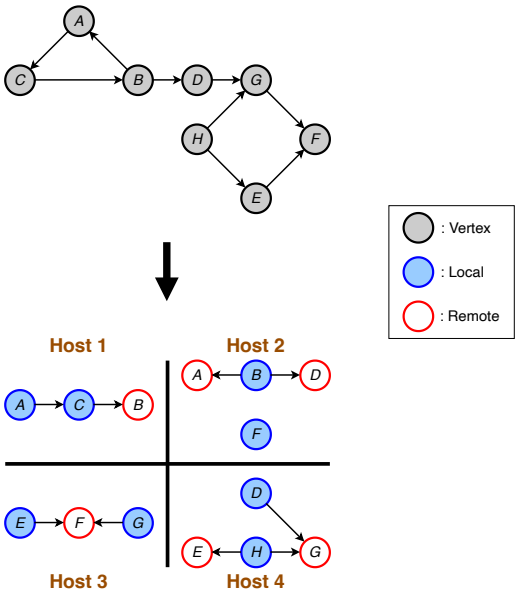


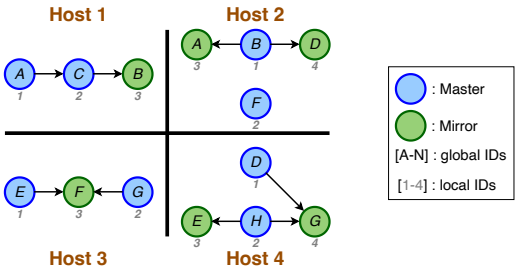
Figure 2: Example of outgoing edge-cut partitioning.

terminology throughout the paper. A wealth of partitioning schemes has been proposed and are categorized into edge-cuts [23,29,30,40,41,43], vertex-cuts [35,46], and 2D-cuts [5, 17, 18]. Evaluations in prior work [18, 21] show that no single scheme is universally optimal due to diverse graph topologies and application requirements. This paper thus focuses on the widely used outgoing edge-cut (OEC) policy.

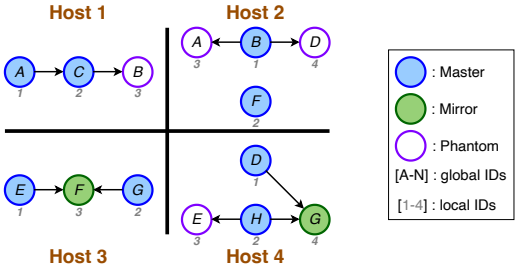
In OEC, each vertex is first assigned to a host based on a predetermined mapping, often a simple hash function. All outgoing edges of a vertex are then co-located with it on the same host. As illustrated in Fig. 2, many local edges will naturally point to remote vertices. Unlike vertex-cuts or 2D-cuts, which may split a single vertex across multiple hosts, edge-cuts maintain single ownership per vertex. This avoids ambiguity for reads and writes to remote edge endpoints, providing reproducible behavior and consistent performance across independent runs.

2.3.2 Full Mirroring

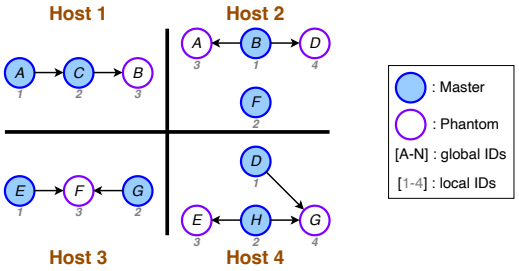
Direct execution on partitioned graphs frequently incurs remote reads and writes, which are expensive under high network latency and limited interconnect bandwidth. Modern distributed graph analytics systems [11, 14, 19, 20, 48] mitigate this with the *mirroring* technique, which replaces remote communication with local memory accesses. Each host allocates memory to replicate the vertex properties of remote edge destinations, allowing edges to read and update locally. These local replicas are commonly known as *mirrors* or *mirror nodes*, while the originally designated vertices are called *masters* or *master nodes*. To maximize communication reduction, the dogma has been to replicate all potentially required



(a) Full mirroring



(b) Static partial mirroring



(c) Mirror-free

Figure 3: Example of different mirroring techniques.

remote vertex data. We refer to this approach as *full mirroring*, which is illustrated in Fig. 3a.

2.3.3 Bulk-Synchronous Parallel

As computation proceeds, replicated mirror properties may become stale and diverge from their true values. Synchronization between mirrors and masters is required, and therefore the bulk-synchronous parallel (BSP) model [44] is commonly employed with full mirroring. Program execution is organized into rounds, and the BSP model further divides each round into two distinct phases:

- *Computation:* each host operates on its local partition using potentially stale mirror values, while multiple updates targeting the same mirror are combined.
- *Communication:* the accumulated mirror updates are sent to masters for reduction, and then reduced values are

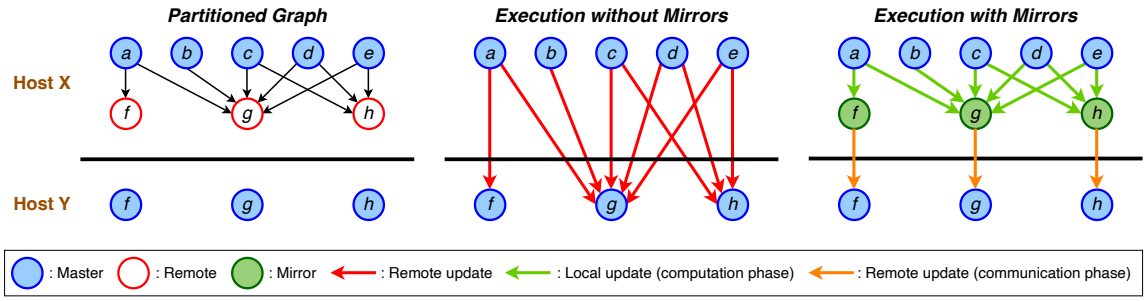


Figure 4: Example of mirror heterogeneity.

broadcast back to mirrors, restoring consistency before the next round.

This approach enables coarse-grained synchronization, minimizing network traffic while maintaining correctness.

2.3.4 Identifiers

Each vertex in the input graph is assigned a unique identifier to ensure distinction across the distributed system. These identifiers, referred to as *global IDs*, are recognized on all hosts and serve as the standard means of addressing vertices during inter-host communication. However, global IDs are often sparse, non-consecutive, and may require large data types due to their wide address space.

Under full mirroring, every master and mirror on a host maintains a piece of local memory. To optimize memory utilization, vertex data is laid out in a contiguous block using a compressed sparse row (CSR) representation. Within this structure, each vertex is assigned an offset into the local memory chunk, known as the *local ID*. Local IDs are host-specific and can be reused across hosts without conflict. Their compact and dense nature enables more efficient indexing than sparse global IDs, making them well suited for use during the computation phase.

3 Advanced Mirroring Design

We present two advanced mirroring designs that reduce memory overhead while preserving high performance under OEC: *static partial mirroring* and *mirror-free*. This section details their rationale and the supporting Pluto mechanisms.

3.1 Push Model

3.1.1 Mirror Heterogeneity

In the push model, each outgoing edge represents a potential update to a destination vertex. A mirror replicates the remote destination's properties so that updates along local edges are applied without immediate remote accesses during computation. A fixed synchronization cost is then paid during communication to propagate the accumulated local changes to the master. This reduces inter-host communication volume, alleviating network bottlenecks in distributed frameworks and thereby improving performance.

Fig. 4 illustrates the key insight of mirror heterogeneity: not all mirrors contribute equally to reducing remote communication. Each incoming edge to a mirrored destination corresponds to a potential conversion of a remote update into a local one, so the benefit generally increases with higher incoming degree. In practice, the access patterns also matter because workloads that activate only a limited subset of the graph may leave even high-degree destinations inactive. Critically, mirrors with only one incoming edge (e.g., mirror f) replace one remote update with one local update plus one remote synchronization, adding local work without reducing network traffic. Since the number of updates increases without any communication saving, these mirrors are considered *nonproductive*. Fig. 5 quantifies the fraction of nonproductive mirrors across our datasets; the tall bars highlight the importance of addressing mirror heterogeneity in system design.

3.1.2 Static Partial Mirroring

Static partial mirroring leverages mirror heterogeneity by removing nonproductive mirrors. After partitioning, candidates with a single incoming edge are identified statically and no mirror storage is allocated for them. These remote destinations without local replicas are referred to as *phantoms* or *phantom nodes*. Fig. 3b illustrates this approach: on host 4, vertex G has two incoming edges and is mirrored, whereas vertex E has a single incoming edge and is treated as a phantom. By excluding phantoms from data replication, static partial mirroring reduces memory usage and avoids unnecessary overhead while retaining the performance benefits for productive mirrors.

3.2 Pull Model

3.2.1 Mirror-Free Architecture

In the pull model, each thread pivots on one vertex at a time and sequentially reads all incoming edges. Source values are

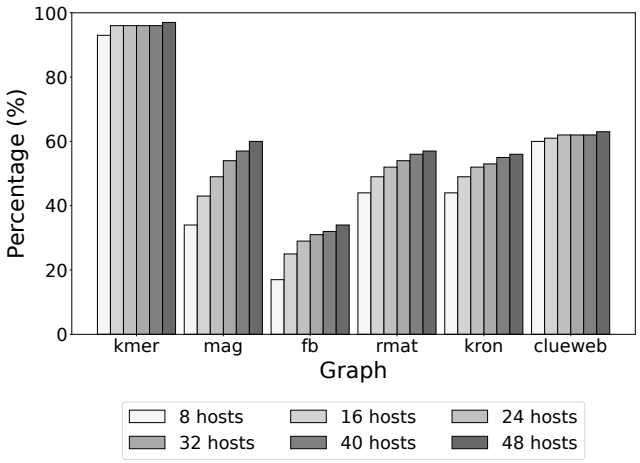


Figure 5: Fraction of nonproductive mirrors.

combined in a temporary accumulator, and the pivot's state is updated once with the accumulated result. If the pivot is a mirror, execution performs a local update during computation and later issues a remote update to the master during communication. Without the mirror, only a single remote update is issued, indicating that mirrors provide no benefit and introduce extra local work. Under OEC, outgoing edges reside with their masters on the same host, so all edge sources are masters and mirror values are never read in the pull scenario. As a result, removing mirrors completely eliminates local update overhead without increasing remote reads, which motivates the proposed mirror-free design (Fig. 3c).

3.2.2 Vertex Traversal Order

The mirror-free architecture enforces a fixed vertex traversal order that processes all masters before any phantoms in each round. This deterministic ordering allows each thread to implicitly detect the transition and switch modes exactly once: it writes updates locally while visiting masters, and then sends remote update requests while visiting phantoms. In contrast, static partial mirroring pushes updates to edge destinations that may be masters, mirrors, or phantoms. Because these destination types differ in how updates are applied, explicit vertex-type checks are required to select the appropriate interface, adding computational overhead.

3.3 Inter-Host Communication

3.3.1 Work Migration

Since phantoms do not maintain replicated data, any computation that involves a phantom requires a remote data fetch and may be followed by a remote update. This pattern substantially increases communication overhead because it introduces two-way traffic with request and response messages. To mitigate this, we introduce a work migration mechanism in which computation associated with phantoms is relocated to the owner host where the data resides [37]. This design converts bidirectional request–response exchanges into unidirectional work messages, avoiding the performance penalties of conventional remote data fetches.

In both advanced mirroring designs, only a single update and thus a single work message is associated with each phantom. All phantoms have exactly one incoming edge in static partial mirroring, while local contributions are combined in temporary accumulators before being sent in the mirror-free architecture. As a result, work migration does not introduce additional payload, and the total communication volume remains unchanged compared to full mirroring.

3.3.2 Communication-Computation Overlap

Pluto dispatches work migration messages on demand, effectively shifting part of the communication activity into the computation phase. To preserve BSP semantics, updates originating from remote hosts can only become visible to local masters after the current computation round completes. Therefore, only the network transmission is overlapped with computation, while the migrated work itself is deferred until the subsequent communication phase. By initiating communication earlier and spreading it over a longer interval, Pluto reduces instantaneous pressure on the interconnect bandwidth. Otherwise idle network resources are used for message dispatch and reception during computation, shortening the communication phase without extending computation time. Overall, this overlap between communication and computation allows Pluto to deliver lower end-to-end execution time over fully mirrored systems.

3.3.3 Address Translation

Given that all participating hosts execute the same program in a distributed system, synchronizing between masters and mirrors requires only a vertex identifier and the corresponding update. To ensure consistent interpretation across hosts, the vertex ID must be globally unique. As such, the sender translates its local ID to a global ID before transmission, while the receiver performs a global-to-local ID lookup upon message arrival. These translations can introduce non-negligible overhead.

D-Galois [14] reduces this overhead via memoization of address translation. Under BSP, synchronization between masters and mirrors occurs only during the communication phase, and all updates from a given sender to a given receiver are aggregated into a single message to improve communication efficiency. The aggregation order is fixed for each host pair, and both sender and receiver maintain an array that records the corresponding local vertex IDs in this order. This arrangement allows communication without attaching global vertex

IDs, since local IDs can be recovered by indexing the array sequentially, avoiding translation table lookups.

However, this technique does not apply to phantoms in Pluto because work migration messages are dispatched on demand during the computation phase, and their order can vary from round to round. To alleviate the high translation overhead, hosts exchange the local IDs corresponding to the masters of phantom nodes. The sender can then translate its local ID into the target host's local ID before constructing the message, allowing the receiver to bypass the global-to-local lookup and use the local ID contained in the message directly.

3.4 Dirty Phantom Identification

Full mirroring systems synchronize only dirty mirrors to minimize network traffic and achieve high performance. When a mirror receives an update, the new value is compared with the stored value to decide whether the mirror should be modified and flagged as dirty. In advanced mirroring, phantoms do not maintain persistent states, so this comparison-based flagging cannot be applied directly. Although temporary accumulators are used during computation, their contents are discarded between rounds.

In the push model, this limitation is resolved naturally because only active masters propagate updates through outgoing edges. Under static partial mirroring, each phantom has exactly one incoming edge and derives its state entirely from that source. Whenever the source master changes value and pushes an update, the destination phantom is inherently dirty and immediately migrates the associated computation to the remote owner host. No additional state is required on the phantom to determine whether the migration is necessary.

The pull model is more challenging because phantom nodes in the mirror-free architecture may receive inputs from multiple masters and must combine them. Fig. 6 shows an example where a phantom computes the minimum among its incoming masters, as in algorithms like Connected Components or Breadth-First Search. If the resulting minimum remains unchanged across consecutive rounds, the ideal behavior is to synchronize it only once, since resending the same value does not advance the global reduction. However, the lack of a replicated value prevents a phantom from directly comparing the current minimum with the previous one.

To address this, Pluto introduces an alternative mechanism to identify dirty phantoms without storing full replicas. At the beginning of each communication phase, all vertices reset a dirty bit before any remote work is applied. These bits are then set whenever a vertex's value changes, whether during the processing of migrated work or during the subsequent computation phase. A phantom transmits its temporary accumulator value for reduction only if the source vertex that currently determines this accumulated value is marked dirty. Since the update function is deterministic, unchanged inputs imply an unchanged output. By tracking which inputs have

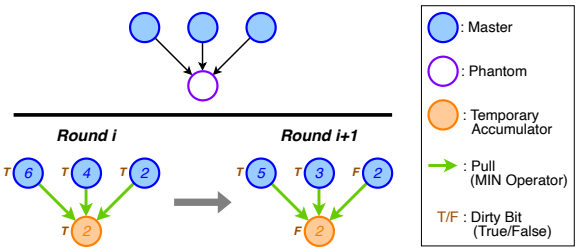


Figure 6: Example of dirty phantom identification.

changed since the last round, Pluto can infer when a phantom's output will change, regardless of the specific update operator (e.g., minimum, maximum, sum, logical operations). This design allows the identification of necessary synchronization targets while avoiding the storage overhead associated with maintaining replicated mirror states.

3.5 Backward Data Dependence

Most graph analytics applications perform only forward passes, where a destination vertex depends on the values of its sources along incoming edges. Some algorithms, most notably Betweenness Centrality, also require a backward pass, in which a source vertex is updated based on its outgoing edge destinations. In these cases, the absence of local state at phantom nodes becomes problematic because outgoing edges are co-located with their source vertices, but the computation depends on destination properties that are not replicated locally.

To support such workloads, Pluto falls back to full mirroring for the backward pass. In the current implementation, this transition is triggered on demand immediately before the backward phase begins. At that point, Pluto reinstates data replication for all phantom nodes by allocating the necessary memory and broadcasting the corresponding values from their masters. This mechanism allows Pluto to remain applicable to the full range of graph analytics benchmarks while still preserving the benefits of advanced mirroring on forward-only workloads. In our benchmarks, only Betweenness Centrality performs a backward pass and transitions exactly once from forward to backward, making this on-demand reinstatement of mirroring both practical and efficient.

3.6 Application Programming Interface (API)

Pluto follows the D-Galois parallel programming abstractions and supports a vertex-centric programming model. Fig. 7 shows push and pull implementations of Breadth-First Search using Pluto APIs.

During the computation phase, `doAll` distributes the specified node set across worker threads and invokes the given function on each node in parallel. In the push version, outgoing edges are traversed and `isPhantom` checks whether

the destination is a phantom. Computation associated with phantoms is migrated to the owner host via `sendWork`, while masters and mirrors are updated atomically using `atomicMin`.

In the pull version, masters are processed first using the non-atomic `min` operation, and dirty bits in `bitset_dist` are set to support dirty phantom identification. Phantoms then accumulate updates from incoming edges, track the source that determines the current value, and issue `sendWork` only when the contributing source is marked dirty.

After local computation finishes, the system enters the communication phase. Migrated work received from remote hosts is applied via `applyRemoteWork`, and `syncMirror` synchronizes remaining mirrors with their corresponding masters.

3.7 Memory Overhead

Prior work introduced the replication factor [14] to quantify the memory overhead of mirroring. Defined as the average number of mirrors per master, this metric captures vertex-level duplication but does not account for edges or metadata used during program execution. To directly reflect total memory usage, we propose the *aggregated memory footprint (AMF)*. As shown in (1), AMF includes both graph storage M_G and peak buffer usage M_{buf}, providing a more comprehensive measure of memory impact across different mirroring strategies.

$$AMF = M_G + M_{buf} \tag{1}$$

Pluto stores graphs in CSR format, which requires memory for vertex data M_{V_data}, edge data M_{E_data}, the edge list array M_{E_dest}, and the index array M_{E_index}. Outgoing edges of the same vertex are grouped together, with their destinations stored sequentially in the edge list array, while the index array records the position of the last outgoing edge for each vertex. These components are given in (2), where D_{vertex} and D_{edge} denote the property data size per vertex and edge in the LPG model, D_{vertex_id} and D_{index} are the sizes of the data types for vertex identifiers and edge list array indices, and N_{master}, N_{mirror}, and N_{edge} are the total numbers of masters, mirrors, and edges in the graph.

$$\begin{aligned} M_G &= M_{V_data} + M_{E_data} + M_{E_dest} + M_{E_idx} \\ \text{where } &\begin{cases} M_{V_data} = D_{vertex} \times (N_{master} + N_{mirror}) \\ M_{E_data} = D_{edge} \times N_{edge} \\ M_{E_dest} = D_{vertex_id} \times N_{edge} \\ M_{E_index} = D_{index} \times N_{master} \end{cases} \end{aligned} \tag{2}$$

Peak buffer usage is further decomposed into buffers for mirror synchronization M_{sync_buf} and buffers for work migration M_{work_buf}. The size of M_{sync_buf} is determined once the graph is partitioned, while M_{work_buf} is obtained from runtime measurements:

$$M_{buf} = M_{sync_buf} + M_{work_buf} \tag{3}$$

```
void BFS() {
  doAll(active_nodes, push);
  applyRemoteWork();
  syncMirror();
}

void push(Node src) {
  NodeData& src_dist = getData(src);
  for (Edge edge: outEdges(src)) {
    Node dst = getEdgeDst(edge);
    if (isPhantom(dst)) {
      sendWork(dst, src_dist+1);
    } else {
      NodeData& dst_dist = getData(dst);
      atomicMin(dst_dist, snode_dist+1);
    }
  }
}
```

(a) Push version

```
void BFS() {
  doAll(all_masters, pullMaster);
  doAll(all_phantoms, pullPhantom);
  bitset_dist.reset();
  applyRemoteWork();
}

void pullMaster(Node dst) {
  NodeData& dst_dist = getData(dst);
  for (Edge edge: inEdges(dst)) {
    Node src = getEdgeSrc(edge);
    NodeData& src_dist = getData(src);
    bool update = min(dst_dist, src_dist+1);
    if (update) {
      bitset_dist.set(dst);
    }
  }
}

void pullPhantom(Node dst) {
  NodeData dst_dist = UINT32_MAX;
  bool send = false;
  for (auto edge: inEdges(dst)) {
    Node src = getEdgeSrc(edge);
    NodeData& src_dist = getData(src);
    if (src_dist+1 < dst_dist) {
      dst_dist = src_dist+1;
      send = bitset_dist.test(src);
    }
  }
  if (send) {
    sendWork(dst, dst_dist);
  }
}
```

(b) Pull version

Figure 7: Examples of Breadth-First Search in Pluto.

4 Implementation

Pluto is implemented on top of the D-Galois GitHub repository forked in May 2023 [13]. Implemented in C++, the integration of both advanced mirroring designs required approximately 4000 lines of code. Key implementation details are summarized below.

Communication Substrate. Inherited from D-Galois, Pluto leverages Gluon [14] as its communication substrate and uses MPI for message transport across the network. To ensure forward progress, it utilizes asynchronous MPI primitives so that multiple outstanding requests can be in flight concurrently, and the MPI handles are reclaimed once the corresponding non-blocking operations complete. Additionally, Pluto limits the number of active MPI tags in each phase of execution to improve the efficiency of message probing and handling.

Communication Isolation. Each host employs a dedicated communication thread responsible for sending and receiving messages. This thread is pinned to the core physically closest to the NIC, as specified by the system architecture, to reduce latency and improve inter-host communication efficiency. All worker threads are likewise pinned to other individual cores, and Pluto enforces a hard limit on the total number of threads to avoid oversubscription and the associated context-switching overhead.

Message Aggregation. Given the heavy network traffic generated by work migration, effective message aggregation is critical for achieving coarse-grained communication and high performance. In Pluto, work messages are continuously aggregated into buffers during the computation phase. Each worker thread independently aggregates messages destined for the same host, and the partially aggregated buffers from all workers are further consolidated before transmission. The default setting of Pluto aggregates 2^{15} messages per buffer.

Buffer Pool. Profiling reveals that frequent calls to `malloc` and `free` introduce significant overhead and can noticeably increase execution time. To address this inefficiency, Pluto preallocates a pool of fixed-size aggregation buffers that are reused throughout execution, eliminating most repeated memory allocations. When the available buffers are insufficient to meet runtime demand, the pool dynamically doubles in size to ensure scalability.

5 Experimental Setup

Configuration. Pluto is evaluated on the Skylake cluster of the Stampede3 system at the Texas Advanced Computing Center. Each compute node has an Intel Xeon Platinum 8160 processor with 48 cores running at 2.1 GHz and 192 GB of DDR4 RAM. Nodes are interconnected through an Intel Omni-Path network with a peak bandwidth of 100 Gb/s. The system runs Rocky Linux 9.5 with kernel version 5.14.0. All code is compiled with gcc 15.1.0 and the Intel MPI library 21.15.

Table 1: Graph datasets.

Graph G	Vertices $\|V\|$	Edges $\|E\|$	Edge Factor $\|E\|/\|V\|$	Max Edge Degree D_{out}	D_{in}
kmer	214M	233M	1.1	8	5
mag	244M	1.7B	7	1.3K	0.4M
fb	100M	9.4B	94	4.3K	3.8K
rmat	268M	4.3B	16	4M	0.3M
kron	1073M	17.2B	16	3.2M	3.2M
clueweb	978M	42.6B	44	7.4K	75M

Datasets. Table 1 summarizes the large-scale plain graphs used in our evaluation. The kmer dataset is a protein k-mer graph from the Graph Challenge competition hosted by the IEEE HPEC conference. The mag dataset is sourced from the Open Graph Benchmark [22] and originates from the Microsoft Academic Graph [45]. The fb dataset is generated using the social network benchmark suite from the Linked Data Benchmark Council (LDBC) [15], designed to reflect structural properties of the Facebook web crawl. The rmat and kron datasets are synthetic graphs generated using randomized generators [9, 27], with weights configured according to the Graph500 benchmark specifications [12]. Finally, the clueweb [36] dataset is one of the largest publicly available real-world web crawls.

Benchmarks. We evaluate Pluto using five representative graph analytics applications: PageRank (PR), Connected Components (CC), Breadth-First Search (BFS), K-Core Extraction (KCore),[1] and Betweenness Centrality (BC). PR runs for a fixed 50 iterations; the other applications run until convergence. Following prior work [14], the source vertex for BFS and BC is chosen as the highest-degree vertex to ensure comprehensive traversals, and the k value for KCore is set to the graph's average degree. Based on active vertex set sizes, BFS and BC use the push model, while PR, CC, and KCore use the pull model to maximize performance on Pluto. Reported results are the median of nine independent runs.

Properties. Existing LPG datasets are too small to meaningfully stress distributed systems and are therefore unsuitable for our evaluation. To approximate LPG characteristics, we assign properties to vertices and edges in the large-scale plain graphs. Property data sizes are derived from the LDBC social network graph [3, 15], which annotates entities (vertices) and relations (edges) with multiple attributes to model a real social network. We record the label distribution of masters, mirrors, and edges independently when distributing this social network graph with the CuSP partitioner [21] employed by Pluto. The collected property layouts for each configuration are then applied to the plain graphs for evaluation. Fig. 8 shows the distribution of property data sizes for the 8-host configuration.

[1] KCore does not perform the full k-core decomposition but only computes the k-core subgraph for a given k.

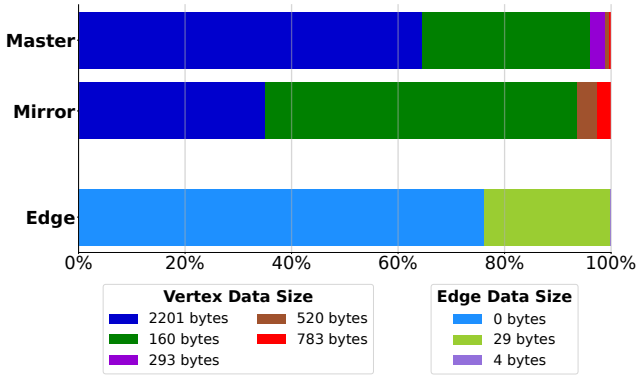


Figure 8: Distribution of property data sizes in the LDBC social network graph for the 8-host configuration.

To further study system behavior when properties are accessed, we conduct a separate set of experiments in which the traversal algorithms are extended to include predicate checks on vertex and edge properties. These setups, including property assignment and benchmark extension, enable us to capture the impact of property storage on memory locality and evaluate the effectiveness of our advanced mirroring designs on LPGs.

Systems. Pluto is compared against three state-of-the-art distributed graph analytics systems: Graphite [32], D-Galois [14], and Gemini [48]. We follow each system's recommended configuration and tune parameters to achieve the optimal performance. GraphScope [16] is excluded because it depends on Vineyard and Kubernetes for resource management, which are incompatible with the Slurm-based Stampede3 environment. We also enhanced D-Galois with the communication isolation optimization and present D-Galois$^+$ as an additional full-mirroring comparison point.

6 Evaluation

6.1 Memory Overhead

6.1.1 Aggregated Memory Footprint

Fig. 9 shows the average aggregated memory footprint (AMF) per host for Pluto under the worst peak buffer usage observed across all benchmarks. Results are normalized to a full mirroring baseline, whose average per-host memory usage is shown above the dotted line for each data model. Most data points fall below 1, indicating lower memory requirements. As expected, the mirror-free architecture generally achieves larger savings than static partial mirroring because it removes all mirrors rather than only a subset. The few exceptions arise in benchmarks such as CC and KCore that converge in very few rounds, where most progress occurs early and work migration incurs relatively high buffer usage. For other algorithms, these trends hold consistently.

The memory benefits of Pluto are substantially greater under the LPG data model. As properties are added, M_G grows sharply while M_{buf} remains roughly constant, making M_{buf} negligible in the AMF calculation in (1). The much larger vertex property sizes compared to edge properties reported in Fig. 8 cause reductions in the number of mirrors to contribute more heavily to M_{V_data} in (2). Conversely, graphs with larger edge factors in Table 1 (e.g., fb and clueweb) exhibit smaller savings because their M_G is dominated by edge-related terms (M_{E_data} and M_{E_dest}), weakening the impact of advanced mirroring that primarily reduces vertex storage. Finally, memory savings for static partial mirroring are strongly correlated with the number of removed mirrors (i.e., phantoms). As more hosts are added, the increasing fraction of phantoms in Fig. 5 aligns with the clear downward trend of normalized AMF in Fig. 9.

6.1.2 Minimum Number of Hosts

While AMF captures the total memory requirement, an important aspect of reduced memory usage in distributed processing is the ability to run on fewer hosts, which often translates to lower cluster cost. Average AMF per host (Fig. 9), however, does not reflect load imbalance due to imperfect graph partitioning. Under outgoing edge-cut, even if masters are evenly distributed, the number of co-located edges can vary significantly across hosts.

To obtain a more realistic view, we apply the LPG setup described in Section 5 to both Pluto and D-Galois$^+$ before execution. We then sweep over a range of host counts and record the minimum number of hosts required to run all LPG-extended benchmarks, as shown in Fig. 10. This procedure incorporates the load imbalance produced by the CuSP partitioner [21] and ensures that the host with the maximum memory requirement does not exceed node memory capacity and trigger out-of-memory errors. Note that the results in Fig. 9 are obtained analytically from (1), (2), and (3) without enforcing node memory capacity limits, whereas Fig. 10 reports actual feasibility on the Skylake cluster. Across our input graphs, Pluto requires only 50% to 90% of the hosts needed by a fully mirrored system.

6.2 Performance

The performance gains of Pluto shown in this section are attributed to the advanced mirroring design and its associated implementation features introduced in Sections 3 and 4. Pluto inherits optimizations from D-Galois because it is directly implemented on top of it, but does not apply optimizations from other existing systems otherwise.

6.2.1 Optimal Configuration

Table 2 reports the best end-to-end execution time achieved by each system across all inputs and applications. We evaluate

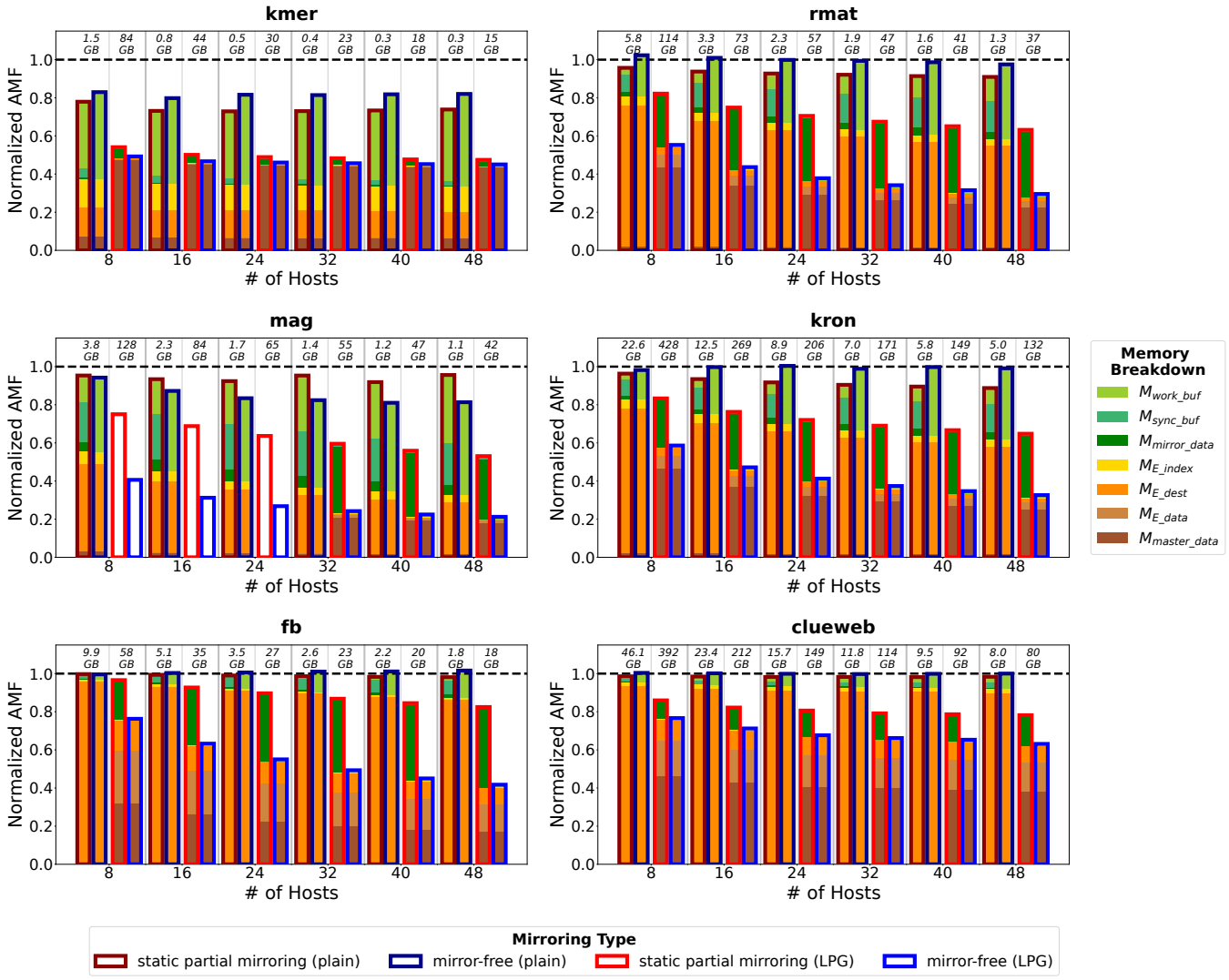


Figure 9: Average memory usage per host for advanced mirroring under the worst peak buffer requirement. The aggregated memory footprint is normalized to a full-mirroring baseline, whose average per-host usage is shown above the dotted line.

each system on clusters of 8, 16, 24, 32, 40, and 48 hosts and select the configuration that yields the optimal result. Overall, Pluto outperforms all prior works and delivers a 2.5× harmonic-mean speedup.

Graphite exhibits roughly a 20× slowdown compared to Pluto mainly because the underlying linear-algebra-based programming model requires substantially more iterations to converge than our vertex-centric approach.[2] Since D-Galois is our implementation baseline, Pluto consistently outperforms it as expected. Gemini is the fastest prior system due to its adaptive hybrid update models but still falls short of Pluto without advanced mirroring designs and the corresponding communication optimizations.

6.2.2 Strong Scaling

Since Gemini delivers the best performance among prior systems in most cases, we collect strong-scaling execution times for Pluto and Gemini to enable a direct comparison. Fig. 11 presents a subset of the results, omitting smaller graphs and some benchmarks due to space constraints. Overall, Gemini reaches peak performance at smaller cluster sizes, while Pluto exhibits better scalability as the number of hosts increases.

Gemini's advantage at small scales stems from its dual update model, which allows adaptively switching between push and pull during runtime. Many graph analytics workloads favor pull-based processing in early iterations when active vertex sets are large, and benefit from push-style execution

[2]We confirmed the setup and configuration with the authors, but the measured performance is worse than the results reported in the original paper [32]. We have also evaluated on various clusters with diverse processors, and the slowdown trend persists.

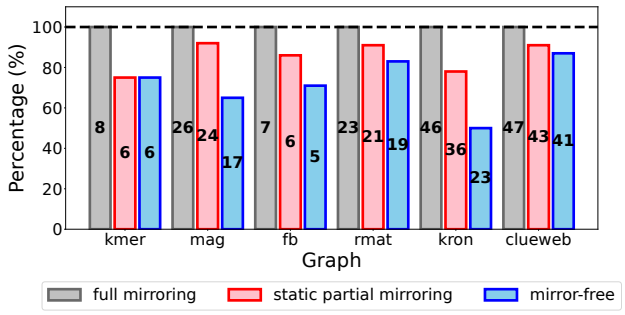


Figure 10: Minimum number of hosts required for LPG workloads.

near convergence when the active set shrinks. Pluto is thus limited by its fixed update model throughout execution. Although Pluto could adopt a hybrid approach, doing so would sacrifice memory efficiency: supporting both push and pull requires maintaining outgoing and incoming adjacency structures simultaneously, effectively doubling edge list storage (M_{E_dest} and M_{E_index}). In addition, the memory benefit of removing mirrors entirely in the mirror-free architecture would no longer apply while supporting static partial mirroring.

The high memory requirements of Gemini limit its applicability, leading to out-of-memory failures at smaller host counts for large graphs. Furthermore, Gemini exhibits weaker scalability primarily because its communication substrate is less optimized. As additional hosts are added and communication becomes the dominant cost, Gemini's performance degrades and execution time increases. In contrast, Pluto scales efficiently and outperforms Gemini as cluster sizes grow.

6.3 Advanced Mirroring Impact

Comparisons across systems depend on many factors and should not be attributed to a single feature. To isolate the contributions of advanced mirroring, we compare Pluto against D-Galois$^+$, a full mirroring baseline with communication isolation, for each graph data model.

6.3.1 Plain Graph

Under optimal configurations, Table 2 shows that advanced mirroring delivers a 1.75× harmonic-mean speedup over full mirroring across all inputs and benchmarks. Fig. 11 demonstrates strong-scaling performance for Pluto and D-Galois$^+$. The two versions converge at different rates and may reach their minimum execution time at different host counts. Fig. 12 summarizes the strong-scaling speedup of our advanced mirroring designs up to the smallest host count at which either version achieves its minimum execution time. Configurations beyond that point are excluded because adding more hosts degrades performance on one or both systems, making them neither a fair comparison nor a practical choice.

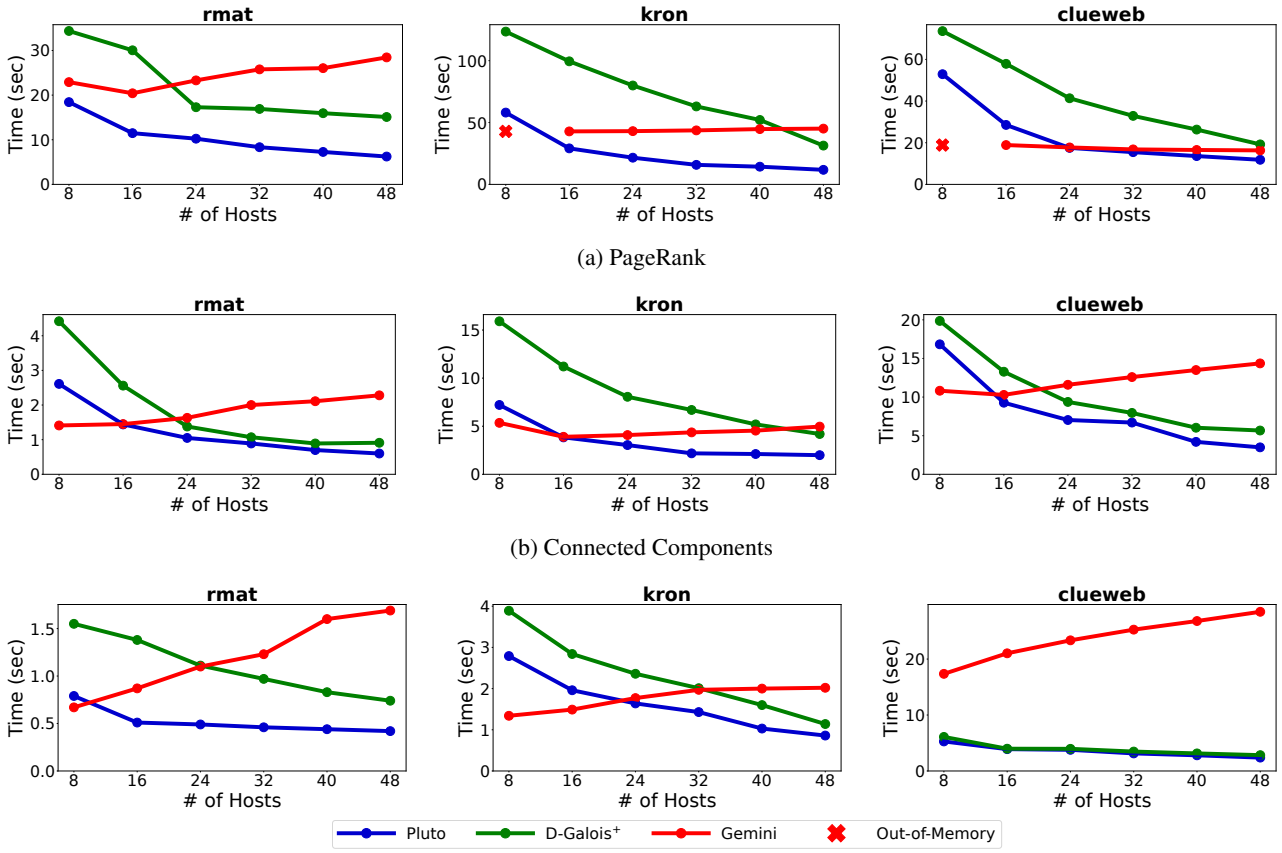
Table 2: Fastest execution time (sec) under optimal configuration. Minimum time across all systems is highlighted in yellow.

Benchmark	*Graph*	*Graphite*	*D-Galois*	*Gemini*	*D-Galois$^+$*	*Pluto*
PR	**kmer**	17.61	8.35	7.10	0.84	0.65
	mag	42.05	14.02	16.72	3.32	1.70
	fb	102.1	10.08	8.64	4.89	3.02
	rmat	100.5	31.70	20.41	15.10	6.35
	kron	180.5	98.03	42.83	31.36	11.56
	clueweb	480.2	36.45	15.52	19.21	11.75
CC	**kmer**	44.98	41.68	17.42	3.26	1.51
	mag	20.17	18.11	5.61	3.64	2.41
	fb	18.61	1.18	0.92	0.48	0.31
	rmat	12.74	2.08	1.40	0.89	0.63
	kron	47.26	9.11	3.90	4.19	2.03
	clueweb	165.1	14.05	10.28	5.67	3.53
BFS	**kmer**	14.83	0.34	0.04	0.04	0.01
	mag	7.40	5.82	2.59	0.64	0.31
	fb	0.63	1.69	5.66	0.30	0.24
	rmat	6.72	1.31	0.67	0.74	0.46
	kron	14.64	3.47	1.34	1.14	0.84
	clueweb	49.31	25.13	17.36	2.83	2.42
KCore	**kmer**	—	4.47	—	3.45	2.08
	mag	—	5.98	—	2.75	1.93
	fb	—	2.68	—	0.99	0.73
	rmat	—	1.32	—	0.91	0.46
	kron	—	3.69	—	1.89	0.86
	clueweb	—	247.5	—	167.7	123.1
BC	**kmer**	—	1.60	0.10	0.23	0.08
	mag	—	17.80	8.45	7.94	3.72
	fb	—	4.22	1.62	2.13	0.56
	rmat	—	6.37	1.46	1.61	0.66
	kron	—	13.39	3.32	4.93	2.49
	clueweb	—	155.2	25.13	23.99	17.38

*— means the benchmark is not implemented by the original authors.

The speedup of both advanced mirroring designs follows a consistent concave-downward trend as the number of hosts increases. At small cluster sizes, computation phases are relatively long and provide ample time to cover the network traffic associated with work migration. The fraction of total execution time saved by shifting part of the communication workload into computation is therefore modest, and the speedup is limited. As additional hosts are introduced, increased parallelism shortens computation phases, improving communication–computation overlap and increasing speedup. This trend continues until an inflection point where the computation duration aligns optimally with the amount of communication that can be overlapped, yielding maximal speedup. Beyond this point, further scaling shortens computation phases to the extent that they can no longer fully hide remote work message transmission, causing spillover into the communication phase and reducing overall performance gains.

When the number of mirrors per host is small, the communication phase of full mirroring and static partial mirroring must still pay for dirty-mirror collection and the inherent network latency of inter-host synchronization. These fixed costs do not diminish with additional hosts and increasingly dominate communication time as total execution time approaches its minimum. In contrast, the mirror-free architecture can still



(a) PageRank

(b) Connected Components

(c) Breadth-First Search

Figure 11: Strong-scaling execution time.

realize slight improvements because work messages are sent during computation, allowing a portion of the communication latency to be hidden. This effect explains the slight uptick in mirror-free speedup at larger cluster sizes.

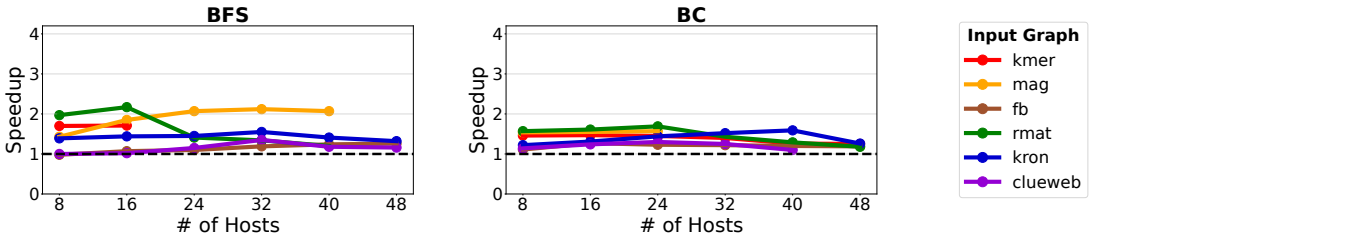
Overall, the mirror-free architecture achieves higher speedup than static partial mirroring, primarily due to reduced computation slowdown and more effective communication–computation overlap. Static partial mirroring incurs extra cost from explicit vertex-type checks, whereas the mirror-free architecture avoids this overhead via a fixed vertex traversal order. In addition, since work migration applies only to phantoms, removing mirrors entirely increases the fraction of communication that can be overlapped with computation. In the ideal case, the full communication workload can be executed in parallel with the computation phase.

6.3.2 Labeled Property Graph

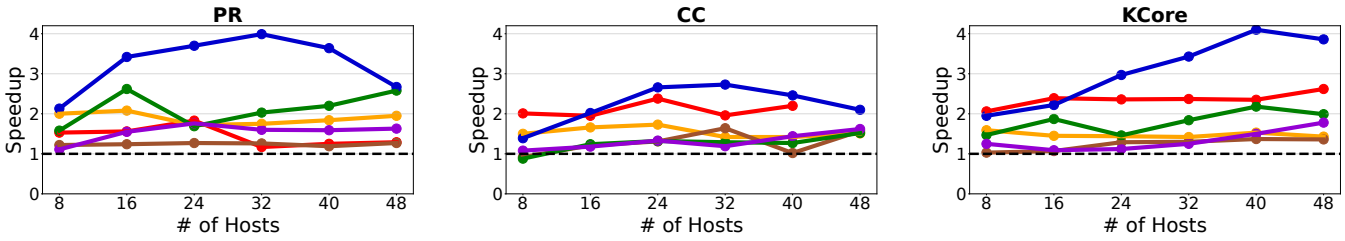
Fig. 13 shows the speedup of our advanced mirroring designs for LPG workloads; missing data points indicate configurations where the graph does not fit in memory according to Fig. 10. The same concave-downward trend observed for plain graphs persists under strong scaling on LPGs, but the overall speedup shrinks.

The performance benefits of the advanced mirroring designs manifest differently for LPGs due to the changed compute–communication balance. Once property predicates are added to our workloads, as described in Section 5, most of our benchmarks spend a larger fraction of time in local computation, so the portion of execution time that any communication optimization (including work migration and mirroring) can affect is smaller. Consequently, the relative speedups shrink, even though the communication phase is still significantly shortened and the underlying mechanism remains effective. This phenomenon is particularly pronounced for PR, where nearly all vertices and edges are accessed in each round. Overall, the advanced mirroring designs still deliver speedups of up to 2.6× with a harmonic mean of 1.37×.

Only kmer and mag exhibit relatively larger LPG speedups for CC and KCore. These two graphs have smaller edge factors, and both benchmarks converge within a few rounds. Fast convergence makes most progress in early iterations when work migration volume is high and spillover is more likely, where remote work message transmission cannot be fully

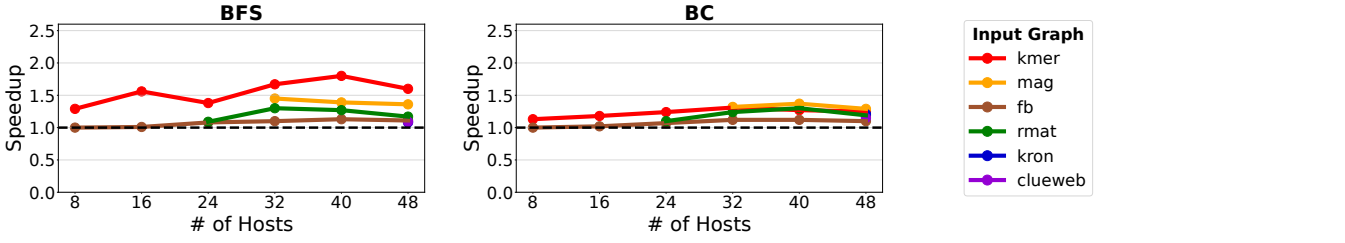


(a) Static partial mirroring

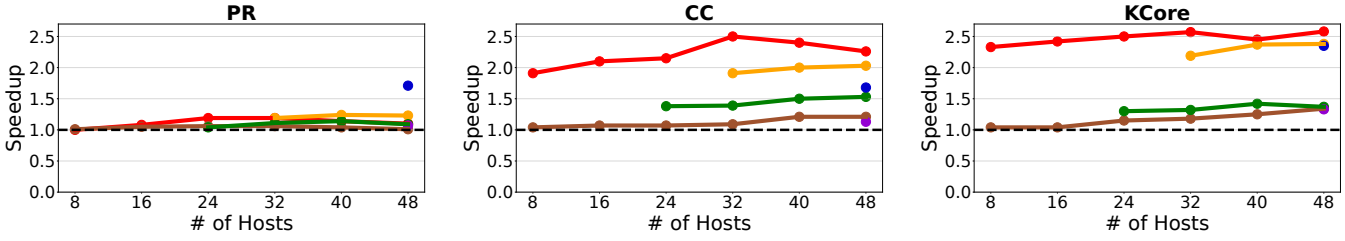


(b) Mirror-free architecture

Figure 12: Speedup of advanced mirroring designs for plain graph.



(a) Static partial mirroring



(b) Mirror-free architecture

Figure 13: Speedup of advanced mirroring designs for LPG workloads.

hidden by the computation phase and extends into the communication phase. At the same time, the lower edge factor keeps predicate-check overhead moderate. This balance allows the additional computation to compensate for spillover without dominating total execution time.

6.4 Design Features

The design features described in Sections 3 and 4 are necessary to support the high communication volume induced by advanced mirroring. Fig. 14 reports the execution time of Pluto with each feature selectively disabled (while all others remain enabled), illustrating its impact on performance. Execution times are normalized to Pluto with all features enabled, and results are reported as the harmonic mean across all inputs and benchmarks.

Buffer pools amortize memory allocation and deallocation costs by avoiding frequent `malloc` and `free` calls. Pluto benefits most from rounds with intensive work migration and continuous message aggregation, but the overall effect is diluted by near-convergence rounds with low compute intensity. The fixed vertex traversal order has limited impact because

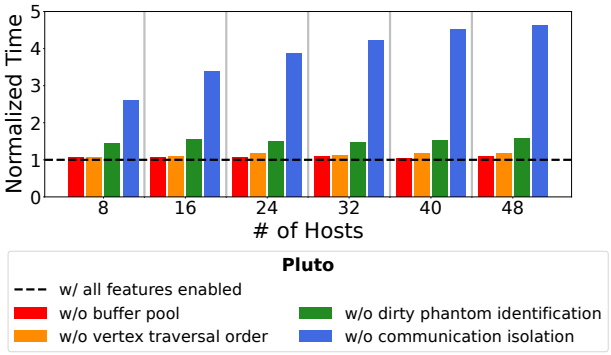


Figure 14: Impact of different design features. Execution time is normalized to Pluto with all features enabled. Results are the harmonic mean across all inputs and benchmarks.

outgoing edges are sorted based on destination local IDs to form the order of masters first, mirrors next, and phantoms last. Modern branch predictors take advantage of it to make the additional vertex-type check relatively inexpensive even without this optimization.

Since inter-host communication is the main bottleneck in distributed systems, communication-related features are the most critical for efficiency. Dirty phantom identification provides a moderate improvement by reducing the total communication volume. Disabling communication isolation causes substantial performance degradation because context switching overhead slows down the communication thread significantly, and it may be scheduled on a core farther from the NIC. Message aggregation has the largest impact on performance but is omitted from the figure: without aggregation, network traffic explodes and Pluto exceeds the 1000-second timeout limit on all experiments.

7 Related Work

7.1 Memory-Efficient Data Duplication

Pregel+ [47] also creates mirrors selectively based on vertex degree, but uses a threshold derived from a simplified cost model rather than a fixed threshold of one. Unlike Pluto, which overlaps communication and computation via work migration, Pregel+ defers all message handling (send, receive, and work processing) to the communication phase. Instead of aggregating messages on demand, it stores work messages in a large buffer during computation and scans this buffer later to identify aggregation opportunities. Because the cost model typically yields thresholds larger than one, multiple updates to the same vertex can accumulate in the buffer, leading to significant memory overhead. We also evaluated Pregel+ with our experimental setup, and Pregel+ is more than 100× slower than Pluto while sometimes fails to complete within a 1000-second timeout.

Khuzdul [10] incorporates a static data cache that selectively replicates remote data at runtime. The cache follows a *first-accessed-first-cached-with-threshold* policy and does not support eviction. Once the cache is full, subsequent requests do not incur replacement and no additional data can be duplicated. This design risks caching early-accessed vertices that may not be representative of the true hot set when the work set exceeds cache capacity. In addition, the threshold is empirically fixed at 64 without justification or sensitivity analysis. Unlike mirrors constructed deterministically after partitioning, the dynamic duplication of the data cache prevents it from undergoing the necessary preprocessing to index with compact local IDs. As a result, all vertices are referenced by their sparse global IDs in Khuzdul to avoid expensive ID translations.

7.2 Performance Optimizations Based on Vertex Degree

While static partial mirroring in Pluto targets low-degree mirrors, several prior systems apply optimizations that reduce the overheads associated with high-degree vertices to improve performance. These existing techniques are complementary to Pluto, but we do not incorporate and evaluate them due to differences in graph representations and underlying frameworks.

GPS [38] partitions adjacency lists of high-degree vertices across hosts to avoid repetitive messages and reduce communication volume. LUX [25] splits high-degree vertices across multiple hosts and uses edge partitioning primarily to balance load on multi-GPU systems. SympleGraph [49] extends Gemini with compiler-assisted enforcement of loop-carried dependencies, removing redundant computation and communication by propagating updates of high-degree mirrors in a host-to-host fashion until their masters are reached.

8 Conclusion

As real-world graph datasets continue to grow in scale, distributed graph processing systems face increasing challenges related to high memory requirements and performance ceilings. In this work, we propose static partial mirroring and the mirror-free architecture to reduce memory usage while also improving performance. Both designs are implemented in Pluto, our system which integrates a suite of optimization techniques to address the increased communication volume, preserving high scalability and throughput. Our comprehensive evaluation demonstrates that Pluto consistently outperforms state-of-the-art systems, offering a promising direction for the future of scalable distributed graph analytics.

Acknowledgments

This research is partially supported by the NSF (CISE "Expedition" Grant Number 2326576 and SHF-2006943) and by the U.S. Department of Energy, National Nuclear Security Administration Award Number DE-NA0003969. The authors acknowledge the Texas Advanced Computing Center (TACC) at The University of Texas at Austin for providing computational resources that have contributed to the research results reported within this paper. URL: http://www.tacc.utexas.edu

References

[1] Amazon Web Services. Amazon neptune: Graph database service. https://aws.amazon.com/neptune, 2024.

[2] Michael J. Anderson, Narayanan Sundaram, Nadathur Satish, Md. Mostofa Ali Patwary, Theodore L. Willke, and Pradeep Dubey. Graphpad: Optimized graph primitives for parallel and distributed platforms. In *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 313–322, 2016.

[3] Renzo Angles, János Benjamin Antal, Alex Averbuch, Altan Birler, Peter Boncz, Márton Búr, Orri Erling, Andrey Gubichev, Vlad Haprian, Moritz Kaufmann, Josep Lluís Larriba Pey, Norbert Martínez, József Marton, Marcus Paradies, Minh-Duc Pham, Arnau Prat-Pérez, David Püroja, Mirko Spasić, Benjamin A. Steer, Dávid Szakállas, Gábor Szárnyas, Jack Waudby, Mingxi Wu, and Yuchen Zhang. The ldbc social network benchmark. https://arxiv.org/abs/2001.02299, 2024.

[4] Renzo Angles and Claudio Gutierrez. Survey of graph database models. *ACM Comput. Surv.*, 40(1), February 2008.

[5] Erik G. Boman, Karen D. Devine, and Sivasankaran Rajamanickam. Scalable matrix computations on large scale-free graphs using 2d graph partitioning. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '13, New York, NY, USA, 2013. Association for Computing Machinery.

[6] Adrian Bridgwater. Why unstructured data is sorting itself out. https://www.forbes.com/sites/adrianbridgwater/2025/06/16/why-unstructured-data-is-sorting-itself-out/, June 2025.

[7] Aydin Buluç, Tim Mattson, Scott McMillan, José Moreira, and Carl Yang. Design of the graphblas api for c. In *2017 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 643–652, 2017.

[8] Aydın Buluç and John R Gilbert. The combinatorial blas: design, implementation, and applications. *The International Journal of High Performance Computing Applications*, 25(4):496–509, 2011.

[9] Deepayan Chakrabarti, Yiping Zhan, and Christos Faloutsos. R-mat: A recursive model for graph mining. In *Proceedings of the 2004 SIAM International Conference on Data Mining (SDM)*, pages 442–446. SIAM, 2004.

[10] Jingji Chen and Xuehai Qian. Khuzdul: Efficient and scalable distributed graph pattern mining engine. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, ASPLOS 2023, page 413–426, New York, NY, USA, 2023. Association for Computing Machinery.

[11] Rong Chen, Jiaxin Shi, Yanzhe Chen, Binyu Zang, Haibing Guan, and Haibo Chen. Powerlyra: Differentiated graph computation and partitioning on skewed graphs. *ACM Trans. Parallel Comput.*, 5(3), January 2019.

[12] Graph500 Committee. Graph500 benchmarks. https://graph500.org, 2010.

[13] D-Galois developers. D-galois github repository. https://github.com/IntelligentSoftwareSystems/Galois, 2018.

[14] Roshan Dathathri, Gurbinder Gill, Loc Hoang, Hoang-Vu Dang, Alex Brooks, Nikoli Dryden, Marc Snir, and Keshav Pingali. Gluon: a communication-optimizing substrate for distributed heterogeneous graph analytics. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2018, page 752–768, New York, NY, USA, 2018. Association for Computing Machinery.

[15] Orri Erling, Alex Averbuch, Josep Larriba-Pey, Hassan Chafi, Andrey Gubichev, Arnau Prat, Minh-Duc Pham, and Peter Boncz. The ldbc social network benchmark: Interactive workload. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, page 619–630, New York, NY, USA, 2015. Association for Computing Machinery.

[16] Wenfei Fan, Tao He, Longbin Lai, Xue Li, Yong Li, Zhao Li, Zhengping Qian, Chao Tian, Lei Wang, Jingbo Xu, Youyang Yao, Qiang Yin, Wenyuan Yu, Jingren Zhou, Diwen Zhu, and Rong Zhu. Graphscope: a unified engine for big graph processing. *Proc. VLDB Endow.*, 14(12):2879–2892, July 2021.

[17] Gurbinder Gill, Roshan Dathathri, Loc Hoang, and Keshav Pingali. A study of partitioning policies for graph

analytics on large-scale distributed platforms. *Proc. VLDB Endow.*, 12(4):321–334, December 2018.

[18] Gurbinder Gill, Roshan Dathathri, Loc Hoang, and Keshav Pingali. A study of partitioning policies for graph analytics on large-scale distributed platforms. *Proc. VLDB Endow.*, 12(4):321–334, December 2018.

[19] Joseph E. Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. Powergraph: distributed graph-parallel computation on natural graphs. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, OSDI'12, page 17–30, USA, 2012. USENIX Association.

[20] Joseph E. Gonzalez, Reynold S. Xin, Ankur Dave, Daniel Crankshaw, Michael J. Franklin, and Ion Stoica. Graphx: graph processing in a distributed dataflow framework. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, OSDI'14, page 599–613, USA, 2014. USENIX Association.

[21] Loc Hoang, Roshan Dathathri, Gurbinder Gill, and Keshav Pingali. Cusp: A customizable streaming edge partitioner for distributed graph analytics. In *2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 439–450, 2019.

[22] Weihua Hu, Matthias Fey, Marinka Zitnik, Yuxiao Dong, Hongyu Ren, Bowen Liu, Michele Catasta, and Jure Leskovec. Open graph benchmark: datasets for machine learning on graphs. In *Proceedings of the 34th International Conference on Neural Information Processing Systems*, NIPS '20, Red Hook, NY, USA, 2020. Curran Associates Inc.

[23] Jiewen Huang and Daniel J. Abadi. Leopard: lightweight edge-oriented partitioning and replication for dynamic graphs. *Proc. VLDB Endow.*, 9(7):540–551, March 2016.

[24] JanusGraph Contributors. Janusgraph: an open-source, distributed graph database. https://janusgraph.org/, 2024. Version 1.1.0.

[25] Zhihao Jia, Yongkee Kwon, Galen Shipman, Pat McCormick, Mattan Erez, and Alex Aiken. A distributed multi-gpu system for fast graph processing. *Proc. VLDB Endow.*, 11(3):297–310, November 2017.

[26] Hochan Lee, David Wong, Loc Hoang, Roshan Dathathri, Gurbinder Gill, Vishwesh Jatala, David Kuck, and Keshav Pingali. A study of apis for graph analytics workloads. In *2020 IEEE International Symposium on Workload Characterization (IISWC)*, pages 228–239, 2020.

[27] Jure Leskovec, Deepayan Chakrabarti, Jon Kleinberg, Christos Faloutsos, and Zoubin Ghahramani. Kronecker graphs: An approach to modeling networks. *J. Mach. Learn. Res.*, 11:985–1042, March 2010.

[28] Grzegorz Malewicz, Matthew H. Austern, Aart J.C Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel: a system for large-scale graph processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, SIGMOD '10, page 135–146, New York, NY, USA, 2010. Association for Computing Machinery.

[29] Claudio Martella, Dionysios Logothetis, Andreas Loukas, and Georgos Siganos. Spinner: Scalable graph partitioning in the cloud. In *2017 IEEE 33rd International Conference on Data Engineering (ICDE)*, pages 1083–1094, 2017.

[30] Christian Mayer, Ruben Mayer, Muhammad Adnan Tariq, Heiko Geppert, Larissa Laich, Lukas Rieger, and Kurt Rothermel. Adwise: Adaptive window-based streaming edge partitioning for high-speed graph processing. In *2018 IEEE 38th International Conference on Distributed Computing Systems (ICDCS)*, pages 685–695, 2018.

[31] Memgraph Ltd. Memgraph: Real-time graph streaming platform. https://memgraph.com, 2024. Version 2.x.

[32] Mohammad Hasanzadeh Mofrad, Rami Melhem, Yousuf Ahmad, and Mohammad Hammoud. Graphite: a numa-aware hpc system for graph analytics based on a new mpi * x parallelism model. *Proc. VLDB Endow.*, 13(6):783–797, February 2020.

[33] Gordon E. Moore. *Cramming more components onto integrated circuits*, page 56–59. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2000.

[34] Neo4j, Inc. Neo4j graph database. https://neo4j.com, 2024. Version 5.x.

[35] Fabio Petroni, Leonardo Querzoni, Khuzaima Daudjee, Shahin Kamali, and Giorgio Iacoboni. Hdrf: Stream-based partitioning for power-law graphs. In *Proceedings of the 24th ACM International on Conference on Information and Knowledge Management*, CIKM '15, page 243–252, New York, NY, USA, 2015. Association for Computing Machinery.

[36] The Lemur Project. The clueweb12 dataset. https://lemurproject.org/clueweb12/, 2013.

[37] A. Rogers and K. Pingali. Process decomposition through locality of reference. In *Proceedings of the ACM SIGPLAN 1989 Conference on Programming Language Design and Implementation*, PLDI '89, page 69–80, New

York, NY, USA, 1989. Association for Computing Machinery.

[38] Semih Salihoglu and Jennifer Widom. Gps: a graph processing system. In *Proceedings of the 25th International Conference on Scientific and Statistical Database Management*, SSDBM '13, New York, NY, USA, 2013. Association for Computing Machinery.

[39] Julian Shun and Guy E. Blelloch. Ligra: a lightweight graph processing framework for shared memory. In *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '13, page 135–146, New York, NY, USA, 2013. Association for Computing Machinery.

[40] George M. Slota, Sivasankaran Rajamanickam, Karen Devine, and Kamesh Madduri. Partitioning trillion-edge graphs in minutes. In *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 646–655, 2017.

[41] Isabelle Stanton and Gabriel Kliot. Streaming graph partitioning for large distributed graphs. In *Proceedings of the 18th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '12, page 1222–1230, New York, NY, USA, 2012. Association for Computing Machinery.

[42] TigerGraph, Inc. Tigergraph: Scalable graph database for the enterprise. https://www.tigergraph.com, 2024. Version 3.x.

[43] Charalampos Tsourakakis, Christos Gkantsidis, Bozidar Radunovic, and Milan Vojnovic. Fennel: streaming graph partitioning for massive scale graphs. In *Proceedings of the 7th ACM International Conference on Web Search and Data Mining*, WSDM '14, page 333–342, New York, NY, USA, 2014. Association for Computing Machinery.

[44] Leslie G. Valiant. A bridging model for parallel computation. *Commun. ACM*, 33(8):103–111, August 1990.

[45] Kuansan Wang, Zhihong Shen, Chiyuan Huang, Chieh-Han Wu, Yuxiao Dong, and Anshul Kanakia. Microsoft academic graph: When experts are not enough. *Quantitative Science Studies*, 1(1):396–413, 02 2020.

[46] Cong Xie, Ling Yan, Wu-Jun Li, and Zhihua Zhang. Distributed power-law graph computing: theoretical and empirical analysis. In *Proceedings of the 28th International Conference on Neural Information Processing Systems - Volume 1*, NIPS'14, page 1673–1681, Cambridge, MA, USA, 2014. MIT Press.

[47] Da Yan, James Cheng, Yi Lu, and Wilfred Ng. Effective techniques for message reduction and load balancing in distributed graph computation. In *Proceedings of the 24th International Conference on World Wide Web*, WWW '15, page 1307–1317, Republic and Canton of Geneva, CHE, 2015. International World Wide Web Conferences Steering Committee.

[48] Xiaowei Zhu, Wenguang Chen, Weimin Zheng, and Xiaosong Ma. Gemini: a computation-centric distributed graph processing system. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI'16, page 301–316, USA, 2016. USENIX Association.

[49] Youwei Zhuo, Jingji Chen, Qinyi Luo, Yanzhi Wang, Hailong Yang, Depei Qian, and Xuehai Qian. Symplegraph: distributed graph processing with precise loop-carried dependency guarantee. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2020, page 592–607, New York, NY, USA, 2020. Association for Computing Machinery.

ARTIFACT EVALUATED
USENIX
AVAILABLE

The Clustering Strikes Back: Building Cost-Effective and High-Performance ANNS at Scale with Helmsman (Operational Systems)

Yuchen Huang[1,3], Baiteng Ma[1,3], Yiping Sun[3], Yang Shi[3], Xiao Chen[3],
Xiaocheng Zhong[3], Zhiyong Wang[3], Yao Hu[3], Erci Xu[2], Chuliang Weng[1]
[1]East China Normal University [2]Shanghai Jiaotong University [3]Xiaohongshu Inc

Abstract

RedNote (a.k.a., Xiaohongshu, a global-scale social network platform) widely adopts approximate nearest neighbor search (ANNS) to power its search, recommendation, and advertising services. Due to the demanding Service Level Agreements (SLAs), we have to rely on in-memory graph-based ANNS (i.e., HNSW) to provide high throughput and low latency.

However, the ever-growing user base and content volume have led to an explosive increase in memory footprint and consequently huge CapEx and OpEx. After exploring various alternatives, we find that building a clustering-based ANNS on top of all-flash servers can be promising. Yet, we still experience severe overheads from the kernel I/O stack, a fixed pruning strategy, and slow index construction.

We present HELMSMAN, a high-performance and cost-effective clustering-based ANNS system, which combines an ANNS-oriented userspace storage stack, a leveling-learned pruning module, and GPU-accelerated pipelines of construction. HELMSMAN saves over 90% of hardware costs and enables billion-scale index (re)builds within hours. In the current production deployment, operating stably for several months, 40 machines now host ANNS workloads that previously required about 35,000 cores and 0.35 PB DRAM.

1 Introduction

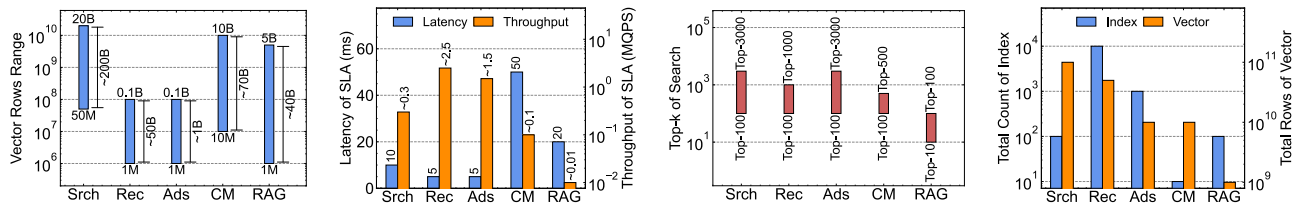
RedNote (a.k.a., Xiaohongshu) [8] is a global social media platform for users to share and interact with photos, videos, and text in the form of "notes". By 2025, the platform hosts over 300 million monthly active users and more than 80 million creators [5]. As the core infrastructure behind search, recommendation, and advertising [22, 41, 43], our approximate nearest neighbor search (ANNS) service manages hundreds of billions of embedding vectors and sustains millions of queries per second (QPS) with strict latency service-level agreements (SLAs) (e.g., 5-10 ms for real-time services).

To meet the quality of services (QoS), we previously relied on large in-DRAM graph indexes [20, 40] for high throughput and low latency. However, the scaling of users and content drastically increases the memory footprint, pushing the ANNS infrastructure at RedNote to reach the petabyte level. This incurs prohibitively high CapEx and OpEx for the existing in-DRAM ANNS infrastructure [2, 9]. We are therefore motivated to explore more cost-efficient alternatives.

The recent advancement in NVMe SSDs has made them a promising candidate given their high bandwidth and low cost per GB [18, 25, 34]. For example, a 12-drive PCIe-Gen5 SSD array [12] can deliver roughly 30% of the bandwidth of a 12-channel DDR5 memory [10], yet SSD costs about 1/40 of DRAM (i.e., 0.2 $/GB vs. 8 $/GB). While we have successfully deployed graph-based hybrid (SSD+DRAM) ANNS systems like DiskANN [49] for the more relaxed offline workloads (e.g., content moderation), using them for online services (e.g., search) remains impracticable in terms of latency and throughput SLAs. The main reason is the greedy graph traversals maintain long candidate lists (e.g., lengths can be up to 1.5× top-k), which incur many serialized I/Os and thus fail to utilize high bandwidth of SSDs.

The above observation motivates us to rethink the possibility of using clustering-based ANNS. With dependency-free I/Os issued in batches, clustering-based ANNS, such as SPANN [17], can be expected to achieve low latency that meets our SLA and also be scaled for throughput by adding more SSDs. Yet, to apply them in production remains challenging. First, under the traditional I/O stack, SPANN uses only about 20–60% of SSD bandwidth, resulting in a significant throughput gap to in-DRAM HNSW. Second, given the varying top-k of real workloads, the existing fixed pruning approach cannot sufficiently reduce redundant scans and leads to unstable per-query performance. Third, the existing single-node and CPU-only construction can take tens of hours or even days, failing to support fast rebuilds needed by frequent updates of embedding models and vectors.

Hence, to overcome these limitations, we develop HELMSMAN, a high-performance and cost-effective ANNS system using clustering-based indexes atop all-flash servers. There are three main techniques. First, we build an ANNS-oriented userspace storage backend that bypasses the kernel I/O stack, minimizing software overhead and directly orchestrating devices to match the ANNS-specific I/O patterns. Second, we develop a leveling-learned pruning module that adapts to both top-k and query distributions, while remaining compatible with SSD-friendly batched I/O. Third, we leverage GPU acceleration and dynamically allocate CPUs for index construction, enabling sub-hour builds for common-scale (e.g., 0.1B)



(a) *Total rows of vector datasets.* (b) *SLAs of performance.* (c) *Top-k value of requests.* (d) *Frequency of construction.*

Figure 1: *Characterizing ANNS of RedNote along 4 aspects: scale, SLA, top-k, and construction. Search is denoted as Srch, and others are recommendation (Rec), advertising (Ads), content moderation (CM), and retrieval-augmented generation (RAG).*

and hour-level builds for ultra-large-scale (e.g., 10B) indexes.

Our evaluations show that HELMSMAN delivers 2–16× higher throughput than existing DRAM–SSD ANNS and up to 85% of the throughput of in-DRAM ANNS, while preserving latency SLAs. We are actively rolling out HELMSMAN to our production environment. Currently, HELMSMAN can use only 40 machines with about 30-40 TB DRAM in total to sustain online traffic that previously consumed roughly 35,000 CPU cores and ∼0.35 PB of in-DRAM ANNS infrastructure, saving more than 90% device costs. The open-source proof-of-concept version of HELMSMAN and datasets fitted to real-world distributions are available at https://github.com/Red-EAD/helmsman.

2 Background

2.1 ANNS-based Services in RedNote

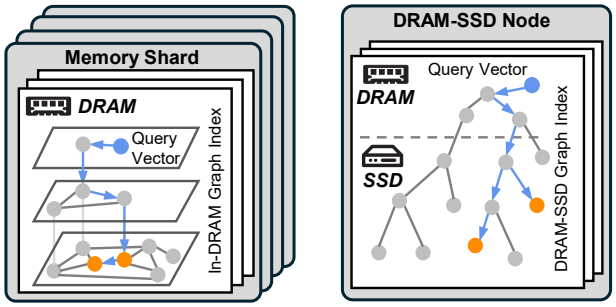
RedNote (a.k.a., Xiaohongshu) [8] is a global social platform with hundreds of millions of active users sharing and interacting with various types of content including pictures, comments and short videos on a daily basis. The proper functioning of RedNote relies heavily on the search engine, recommendation system, advertising platform, content moderation, and emerging AI services (e.g., retrieval-augmented generation for large language models). All of the above are supported by approximate nearest neighbor search (ANNS) over hundreds of billions of high-dimensional vectors with millions of queries per second. Despite all being ANNS-based, these services can have different characteristics as shown in Figure 1, such as scale of dataset (Figure 1a), the SLA of performance (Figure 1b), the top-k of requests (Figure 1c), and the index building (Figure 1d). We now further discuss them in detail.

Search. First, our search targets the full corpus (i.e., all user-generated content and web-wide data) and the index can reach up to tens of billions of vectors. In addition, the search services also follow a multi-path workflow where different recall paths cover various subsets of the corpus (e.g., texts, images, videos, and user behaviors) to ensure high accuracy. Therefore, search has the largest overall vector volume, totaling up to twenty billion vectors in practice. Second, search is directly exposed to end users, which requires an SLA of ∼10 millisecond average latency [16] under ∼300K QPS, our typical peak traffic. Third, ANNS is the first stage of the ranking pipeline in search. Thus, it requires a relatively large top-k (e.g., around 100 to 3,000) to provide sufficient candidates for downstream re-ranking models [21, 37]. Finally, because our embedding models are continuously trained with daily collected metrics (e.g., user behaviors and content popularity), we wish to rebuild (instead of in-place updating such as SPFresh [58], Quake [43] and OdinANN [23]) the entire indexes on a daily basis.

Recommendation. Unlike search, recommendation only focuses on the popular subsets of full corpus. As a result, the total vector volume is significantly smaller than that of search, ranging from 1 to 100 million vectors per index. Recommendation directly serves end users but involves more recall paths. Thus, the online request throughput can reach ∼2.5 million QPS while still requiring millisecond-level latency SLAs [33]. Moreover, post-filtering based on user-specific constraints is common in recommendation. To ensure sufficient results after filtering [63], the top-k from ANNS can be up to 1,000. Finally, embedding models are updated in batches (aggregated from minutes to hours [42, 47]) based on real-time feedback (e.g., click-through rates), leading to up to ten thousand index rebuilds per day and a cumulative rebuilt volume on the order of tens of billions of vectors.

Advertising. Embedded from products, advertising has fewer vectors, around one billion in total. Similar to search and recommendation, advertising also directly serves users and thus requires millisecond-level latency SLAs under millions of QPS [59]. In addition, advertising follows a post-filtering pipeline. ANNS retrieval is followed by scalar filtering on multiple attributes (e.g., exposure constraints and billing status), so the required top-k can be large as well (e.g., up to 3,000) [61]. Finally, for index construction, since the embedding models are modified incrementally by rates of clicks or purchases on the order of minutes, this requires that index rebuilds keep up with these real-time updates as well.

Content moderation. This maintains two types of indexes, the large allow-list corpus and the smaller block-list corpus. The allowlists consist of all content conforming to policies and values, so their indexes can reach up to 10B vectors. The block-list stores only restricted content and is typically much



(a) *In-DRAM graph indexes on distributed nodes for achieving low latency.* (b) *DRAM-SSD graph indexes on a single node for saving memory footprint.*

Figure 2: *Two solutions employed in services of RedNote.*

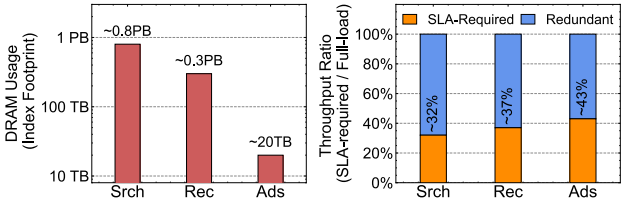
smaller (e.g., 10M vectors). Content moderation is mainly an offline workload, and new content is compared against both allow-list and block-list, then assessment models select high-risk candidates that are forwarded to manual checking. This pipeline results in more relaxed latency SLAs and lower overall throughput than the above-mentioned services [45]. Additionally, to capture restricted content with best effort, we also use relatively large top-k (up to ∼500). Since these collections change slowly, block-list indexes are rebuilt on a daily basis and the allow-list indexes are refreshed weekly.

Retrieval-augmented generation. Our ANNS also supports emerging AI applications like RAG. In RAG, ANNS is mainly used to index task-specific knowledge bases with sizes varying widely from about one million to several billion vectors depending on the applications. Since LLM inference dominates resource consumption and current models have limited context length, the SLA requirements for the ANNS stage are relatively relaxed (e.g., ∼20 ms) [29] and the required top-k is much smaller (e.g., 10-100) than other services' [44, 62]. Moreover, the underlying knowledge bases evolve slowly and the embedding models change infrequently, so index rebuilds are needed far less often than in other online workloads.

2.2 Solutions for ANNS-based Services

Graph-based ANNS offers high throughput with low latency, and is therefore widely deployed in RedNote's production services. To satisfy the strict performance SLAs of search, recommendation, and advertising, we run a large fleet of distributed memory shards that host in-DRAM graph indexes [20, 40]. The aggregated count of CPU cores already exceeds 10^5 among 4,000 nodes, spanning ∼50 clusters as of today. For scenarios with slightly more relaxed latency and throughput requirements (e.g., content moderation and RAG), we instead employ hybrid DRAM–SSD nodes with SSD-backed graph indexes [22, 49, 56] to trade some latency for a much smaller memory footprint.

In-DRAM Graph. In our production, HNSW [40], one of the state-of-the-art in-DRAM indexes, is commonly adopted.



(a) *PB-level DRAM usage.* (b) *Throughput overkill.*

Figure 3: *In our online services, latency SLAs force massive DRAM for pure in-DRAM indexes, while throughput SLAs are modest to the full-load maximum of the in-DRAM deployment.*

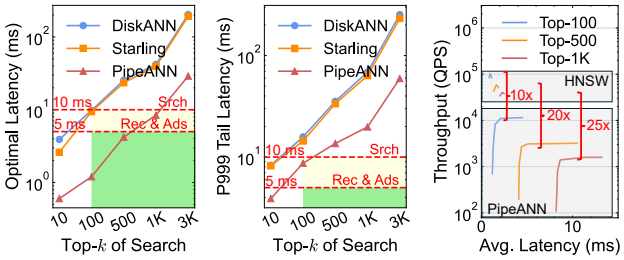
And it is widely deployed for search, recommendation, and advertising. Figure 2a shows, for a given query, the HNSW search starts from an entry node on the top layer and performs greedy best-neighbor descent layer by layer, until it reaches the bottom, where a best-first search over a candidate pool produces the final top-k results. All graph data, including neighbor lists and raw vectors, are kept in DRAM so that the critical path consists only of in-memory accesses. When a single shard cannot hold the entire index, we partition the dataset across multiple memory shards and execute search independently on each HNSW shard. The frontend then merges the partial results from all shards to obtain the global top-k answers. This design can deliver consistently low search latency and high QPS, guaranteeing QoS.

Hybrid DRAM-SSD Graph. For scenarios with more relaxed performance SLAs, such as content moderation and RAG, we employ hybrid DRAM–SSD graph indexes, as illustrated in Figure 2b. These indexes significantly reduce DRAM consumption, allowing a large-scale graph to be hosted on a single node. Representative designs include DiskANN [49] and its recent optimized variants such as Starling [56] and PipeANN [22]. They retain PQ-compressed vectors and a small set of frequently accessed vectors cached in DRAM, while storing the full-precision vectors and graph edges on SSDs to cut memory footprint. During query processing, the graph is traversed with the best-first search strategy and a beam-search procedure. The search path always advances along the edges towards the neighbor that is closest to the query. The beam-search width is translated into an I/O width that controls how many candidates are fetched from SSDs in batches, thereby improving bandwidth utilization for higher performance.

3 Motivation

3.1 Can't Afford In-memory ANNS to Scale

Over the past few years, the fast growth of RedNote's user base has driven the data volume of our stored vector corpus to be doubled annually. The more than billions of new vectors persisted each year exacts a heavy toll on the CapEx of our in-memory ANNS deployment. Figure 3a illustrates that, as



(a) *Average latency.* (b) *P999 latency.* (c) *Throughput gap.*

Figure 4: *All graph-based systems fail in replacing in-DRAM solution. DiskANN and Starling can never meet the latency SLAs. PipeANN shows an insuperable gap in throughput. Srch refers to the search business, with latency SLAs of 10 ms. And Rec & Ads refers to the recommendation and advertising, with latency SLAs of 5 ms.*

Hardware	DRAM [10]	Gen5 SSD [12]	Gen4 SSD [11]
Price ($/GB)	8 (***100%***)	0.2 (***2.5%***)	0.15 (***1.9%***)
BW (GB/s)	12 × 38 (***100%***)	12 × 12 (***32%***)	12 × 6.5 (***17%***)

Table 1: *Price and bandwidth comparisons as of Dec. 2025.*

of 2025, the in-DRAM HNSW indexes for search, recommendation, and advertising have already consumed PB-level DRAM in practice. Even just the ANN indexes of a single service (e.g., search) can now consume hundreds of terabytes of DRAM.

However, the massive DRAM usage is mostly provisioned to maintain the QoS (e.g., 10 ms average latency SLA of search). Specifically, Figure 3b shows that, out of the 100% throughput provided by in-DRAM deployments, only ∼32–43% is actually needed to maintain the required latency SLAs. Nevertheless, we still need the rest 57–68% of DRAM, not for throughput, but for the capacity to host the entire set of indexes to avoid latency spikes under workload bursts [39].

3.2 Can't Replace with Hybrid Graph

Recall that our CM and RAG services (see §2) have adopted a hybrid DRAM-SSD architecture for the graph-based DiskANN to reduce memory consumption. Given that SSDs are becoming increasingly affordable and high-performance as Table 1, it thus motivates us to consider integrating SSDs into other services (e.g., search) as well. Here, we explore the feasibility by conducting comprehensive evaluations of existing graph-based DRAM-SSD ANNS systems, including DiskANN, Starling, and PipeANN, on our standard 96-core server equipped with 12 PCIe-5.0 SSDs. Using the SIFT100M dataset [13], we target a 90% recall rate and evaluate top-k values from 10 to 3,000, which is consistent with both the BigAnnBenchmark [48] and our online services' requirements (§2.1). Unfortunately, the results are disappointing in terms of both latency and throughput.

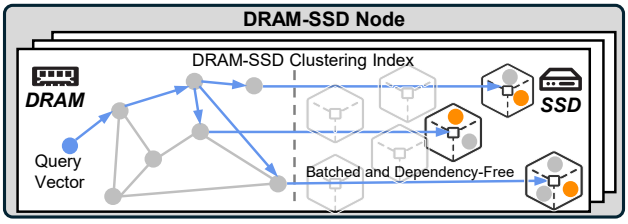


Figure 5: *DRAM-SSD clustering-based index follows the batched and dependency-free I/O pattern on the NVMe SSDs.*

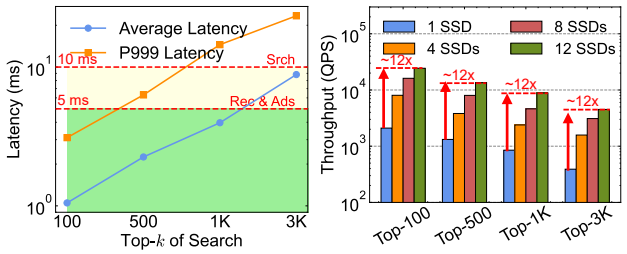
Latency. Figure 4a and Figure 4b report the mean and 99.9th percentile tail latency of the three candidates. We test them with a latency-friendly strategy by executing single-threaded queries (i.e., lowest concurrency). We can observe both DiskANN and Starling consistently fail to meet both our average and tail latency SLAs in the field (the green shaded area). PipeANN, by leveraging intra-query parallelism (i.e., multi-threaded beam-search traversal) to better utilize SSD bandwidth, achieves the lowest latency among the three but still struggles to satisfy the online SLAs at high top-k values, especially for tail latency.

Throughput. To further explore whether PipeANN can serve as a (partial) replacement, we evaluate its throughput against in-DRAM HNSW under the latency SLAs. As shown in Figure 4c, even with 12 PCIe-Gen5 SSDs providing over 30% of DDR5 DRAM bandwidth (Table 1), PipeANN's peak throughput remains 10–25× lower than that of HNSW. The root cause is that the graph-based DRAM-SSD ANNS inherently suffers from strongly serialized, dependency-chain I/Os during search (i.e., each expansion step depends on the results of previous reads). This serialized access pattern magnifies the raw latency gap between SSDs and DRAM (around 10^2-10^3). In our low-latency and high-throughput online services, this limited performance rules out the possibility of graph-based DRAM-SSD systems as a viable replacement for in-DRAM solutions at scale.

3.3 Clustering-Based ANNS Brings Hope

Failing to port in-memory ANNS with graph-based hybrid designs motivates us to reconsider another design, clustering-based DRAM-SSD ANNS. At a high level, Figure 5 shows that SPANN [17], a representative clustering-based ANNS, uses a clustered layout. The clusters' centroids are kept in DRAM, while vectors are stored on SSD as clustered posting lists. During search, it first searches the in-memory graph of centroids to identify the *nprobe* number of (e.g., 3 in Figure 5) closest partitions, then issues batched reads to fetch the corresponding posting lists, and finally computes the top-k neighbors from the loaded candidates of these lists.

Historically, this line of research was known to be latency-friendly but largely dismissed in the field due to the limited throughput [22, 51, 56]. However, the recent advance-



(a) *Near-SLAs latency.* (b) *Scalable throughput.*

Figure 6: *Based on high-bandwidth modern SSDs, the clustering-based SPANN shows nearly qualified search latency and scalable throughput under the large top-k values.*

ment in high-bandwidth NVMe SSDs (e.g., 12 GB/s from Gen5 SSD) has changed this landscape. As shown in Figure 6, using SPANN and up to 12 PCIe-Gen5 SSDs on SIFT100M, our evaluations demonstrate that, with the batched and dependency-free I/O pattern, this clustering-based design can already approach latency SLAs of online services under processing single-threaded query and achieve scalable throughput by adding SSDs. In addition, Figure 6a shows that, even when we increase top-k up to 3×10^3, SPANN nearly keeps the average and tail latency within or close to the SLAs of search, recommendation, and advertising. Moreover, Figure 6b illustrates that the throughput scales almost linearly with the number of SSDs and reaches $\sim 12\times$ speedup when using 12 SSDs, reinforcing our assumption that modern SSDs with clustering-based index can be a promising direction.

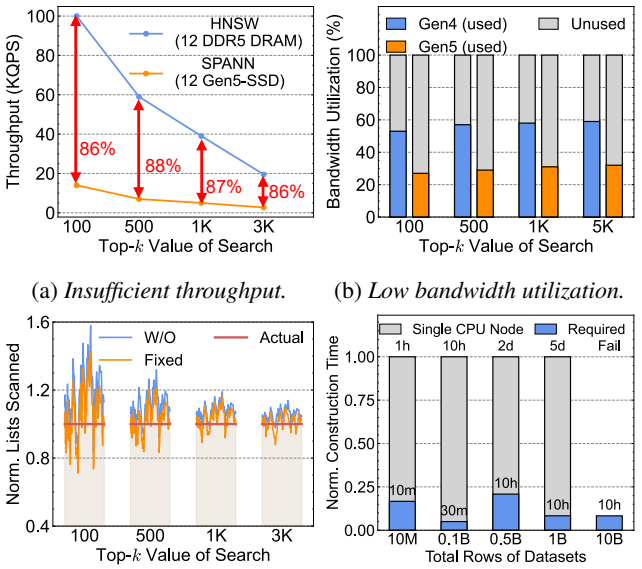
3.4 Challenges in Deploying SPANN at Scale

However, even when we pair SPANN with the array of NVMe SSDs, such a solution still cannot be directly deployed in the field. Challenges persist in both online search and offline construction. For online serving, the achieved search throughput is well below our production SLAs. Moreover, in the offline stage, the construction of indexes incurs prohibitive time overheads that must be substantially reduced.

First, SPANN cannot yet serve as a scalable replacement for the in-DRAM HNSW indexes deployed in our online stack. Under the latency SLAs of online services, even on a 96-core node with 12 PCIe 5.0 SSDs, the throughput remains insufficient. As shown in Figure 7a, there is still a throughput gap of about 86-88% between SPANN and the in-DRAM HNSW baseline.[1] This gap is largely due to insufficient utilization of NVMe SSDs' bandwidth. Figure 7b shows that SPANN uses only ~26-59% of the available bandwidth on PCIe Gen5 and Gen4 arrays, highlighting a primary opportunity to scale throughput by driving SSD bandwidth actually utilized closer to the hardware limits.

Second, SPANN's current strategy for determining the

[1]Referring to our production index-building settings, `edges` and `efConstruction` of HNSW are set as 24 and 120. SPANN is set as §5.1



(a) *Insufficient throughput.* (b) *Low bandwidth utilization.*

(c) *Un-adapted range of scans.* (d) *Long-time index building.*

Figure 7: *Clustering-based SPANN still has deficiencies. Its performance also fails to fully meet the throughput SLAs (~12%-14% of in-DRAM HNSW), due to the low bandwidth utilization and un-adapted range of scans. Furthermore, its single-node CPU-only construction of indexes incurs a heavy expenditure of time, violating the freshness of services.*

scan range (i.e., the number of loaded clusters *nprobe*) impacts both throughput and search quality. It relies on a fixed distance-based pruning rule shown in Equation 1. After locating the closest centroid c_{i1}, SPANN includes clusters $\mathbf{X}_{ij}$ in the search if distances from $\mathbf{q}$ to their centroids c_{ij} are within a $(1+\varepsilon)$ factor of the distance to c_{i1}, where ε is chosen empirically and does not adapt to query difficulty or data distribution. Figure 7c presents statistics for ~100 queries under varying top-k. The scanned range after pruning is only slightly smaller than the no-pruning (W/O) baseline, yet for each individual query, it often overshoots the range needed to reach the target recall (e.g., 90%), wasting I/O and reducing throughput; or undershoots it, causing large variance in recall and resulting in unstable search quality.

$$\mathbf{q} \xrightarrow{\text{search}} \mathbf{X}_{ij} \iff \text{Dist}(\mathbf{q}, c_{ij}) \le (1+\varepsilon)\,\text{Dist}(\mathbf{q}, c_{i1}),$$
$$\text{Dist}(\mathbf{q}, c_{i1}) \le \text{Dist}(\mathbf{q}, c_{i2}) \le \cdots \le \text{Dist}(\mathbf{q}, c_{iK}) \quad (1)$$

Finally, SPANN also faces significant challenges in offline index (re)building. As shown in Figure 7d, the existing implementation constructs the clustering structures on a standalone CPU node. When the dataset grows from ten million to tens of billions of vectors, the construction time escalates from several hours to multiple days. These construction windows are far longer than what is acceptable in our production scenarios, where embedding models and vector data are refreshed frequently. Consequently, index building also becomes a major bottleneck hindering the practical deployment of SPANN.

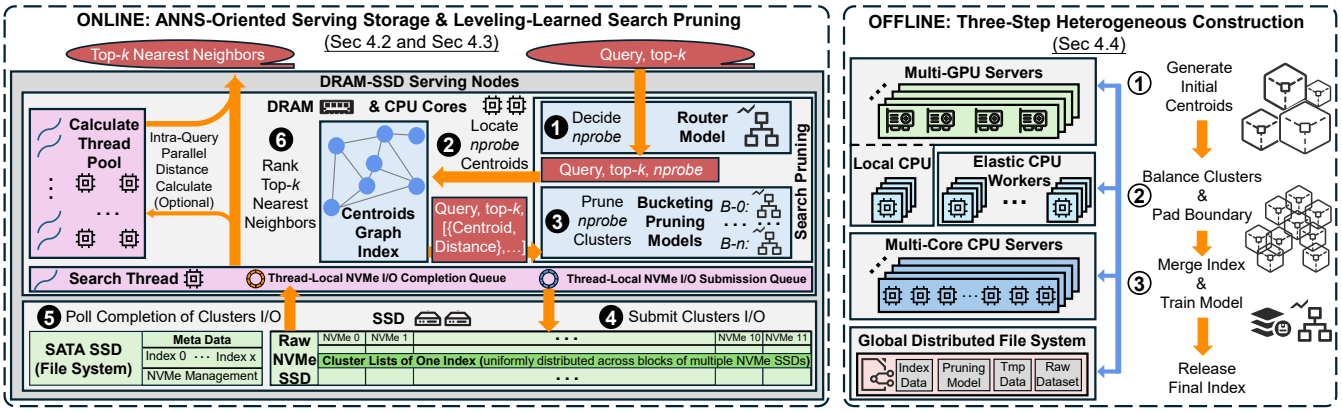


Figure 8: *Design overview of* HELMSMAN *with workflows of online search and offline index construction.*

4 HELMSMAN Design

4.1 Overview

We present HELMSMAN, a high-performance and cost-effective ANNS system using clustering-based indexes on all-flash servers. Figure 8 presents an overview of HELMSMAN, including online serving (i.e., the left half of Figure 8) and offline index construction (i.e., the right half).

Online serving. We employ all-flash servers as ANNS serving nodes where each is equipped with 12×2 TB PCIe Gen5 NVMe SSDs, 12×96 GB DDR5 DRAM, and a 96-core CPU. Recall that existing solutions such as SPANN fall short in throughput due to bandwidth utilization and pruning efficiency. We therefore redesign the storage stack and the pruning module. The key idea is eliminating overhead from I/O software and introducing adaptive learning-based models. At a high level, for each index released to serving, we use DRAM to store the centroid graph for locating nearest clusters and weights of the search pruning module (router and pruning). Meanwhile, the cluster lists are striped across raw NVMe SSDs as the granularity of logical blocks, bypassing the traditional Linux software stack. Since a single node can usually host multiple vector indexes, we use an extra SSD to store the metadata of all indexes and management of raw SSDs.

For each query with target top-k, ❶ the router model decides the *nprobe* value, ❷ then HELMSMAN uses the centroids graph index to locate the *nprobe* closest centroids, and ❸ the leveling pruning model routed at the first step further prunes the candidate clusters. ❹ The search threads submit asynchronous NVMe I/O commands for the selected cluster lists, ❺ poll completions of cluster reads on the hardware completion queue, and ❻ calculate the loaded vectors in the local thread or dispatch them to the calculate thread pool to compute distances and finally rank the top-k nearest neighbors as the result of ANNS.

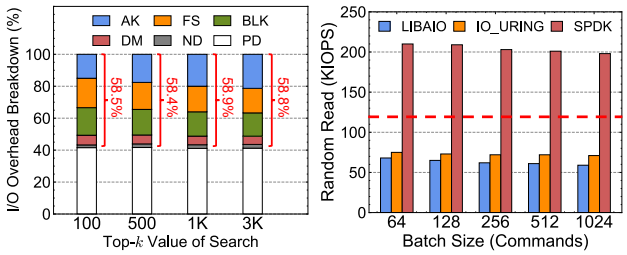
Offline index construction. To facilitate frequent index building, HELMSMAN employs multi-GPU servers to form a three-stage pipeline. Note that a major drawback in previous builds by a single node's CPUs is limited computing power and the inability to scale. Hence, we introduce GPUs to generate initial centroids and distributed nodes to accelerate balancing and padding. Backed by a global distributed file system holding the final index, pruning models, temporary data (e.g., checkpoints), and raw datasets, we orchestrate the entire construction pipeline end-to-end, achieving minute-level index building for datasets up to million-scale vectors and hour-level construction for billion-scale datasets.

In offline construction, ① multi-GPU servers run k-means to generate initial centroids from the raw dataset and save these centroids to the distributed storage. ② Then, owning initial centroids, local CPUs of multi-GPU servers or elastic CPU workers split and rebalance clusters from the initial centroids, pad cluster boundaries, and write intermediate index shards to the global distributed file system. ③ Multi-core CPU servers merge the shards, build the graph for all final centroids, train leveling pruning models, and finally materialize the index files for releasing them on serving nodes.

Design advantages. With these designs, HELMSMAN effectively solves the aforementioned challenges. First, as ❹ and ❺ of the online serving in the left of Figure 8, it bypasses the kernel on the throughput-critical I/O path (i.e., cluster-list reads) completely to exploit the high bandwidth of modern multi-SSD arrays, minimizing software overhead by directly steering NVMe SSDs to match ANNS access patterns (§4.2).

Second, as ❶ , ❷ , and ❸ of the online serving, it adaptively chooses the search range (i.e., probed clusters) on a per-query basis, achieving target recall while avoiding unnecessary I/O and computation: a trained router model predicts the recall-sufficient *nprobe*, and leveling pruning models further eliminate unnecessary scans (§4.3).

Third, as ① and ② of the offline building, to support frequent index building for incremental embedding model training and continuously evolving vector data, we utilize heterogeneous acceleration and make support for elastic scaling. Multi-GPU servers accelerate centroid generation on large-



(a) *I/O breakdown of SPANN.* (b) *Ideal IOPS per core.*

Figure 9: *SPANN's I/O path relies on the traditional Linux I/O software stack, including AK (application-kernel switching), FS (file system), BLK (block layer), DM (device mapper for RAID), and ND (NVMe driver). By issuing fixed-size 12 KB reads in batches—a new batch only after the previous one completes—we show the I/O breakdown of SPANN, and compare the performance of* `libaio`, `io_uring`, *and SPDK.*

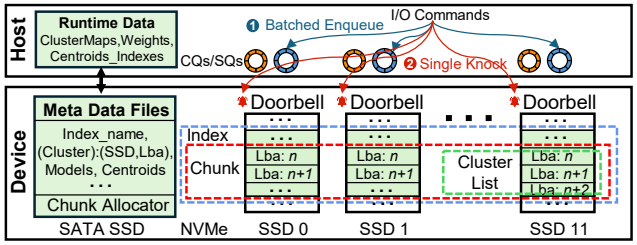


Figure 10: *Meta data as files, clusters' lists as raw blocks.*

scale datasets and we can dynamically allocate CPUs to speed up the following fine-grained index construction flexibly according to requirements (§4.4).

4.2 ANNS-oriented Storage Stack

To better serve the clustering-based ANNS indexes, we study the existing storage stack and propose an ANNS-oriented storage design. We start by analyzing the I/O behavior of clustering-based search and then discuss the corresponding design choices which take advantage of multiple NVMe SSDs with the help of the user-space SPDK driver [14].

Understanding I/O patterns. Clustering-based ANNS has two outstanding I/O patterns during online search. First, the search process tends to generate a large number of batched reads. For example, a single query can generate up to $\sim 10^3$ cluster-list loading requests when searching the top-3,000 nearest neighbors, a typical case in online search and advertising. Such massive reads can lead to a demandingly high IOPS requirement. Taking SIFT100M as an example, under a 10 ms latency SLA, computation (e.g., finding the nearest centroids, distance calculations on vectors loaded, and ranking top-k results) can already cost ~2-4 ms with only ~6-8 ms left for I/Os, which in return translates to ~120-170 KIOPS per search thread on a single core. However, the I/O stack used in SPANN can often only achieve ~30-40 KIOPS.

Second, the clusters of an index have the same size which is also the size of the issued reads. This is because, to avoid long-tail latency and the boundary error during search [17], the clustering-based index usually balances clusters' sizes below a threshold and pads clusters with boundary vectors to the same size (e.g., 12 KB per cluster for SIFT in SPANN). As a result, each cluster can have same count of pages, and each probe typically issues fixed-size reads (e.g., 12 KB from three 4KB pages) when loading the nearest clusters.

Exploring design choices. We next profile the I/O overhead of SPANN (Figure 9a) and three popular I/O stacks including `libaio`, `io_uring`, and SPDK (Figure 9b) under workloads of batched and fixed-size read patterns (i.e., the same I/O behavior as the clustering-based search) as shown in Figure 9. Results indicate that the overhead of the traditional I/O stack with `libaio` dominates the end-to-end path (up to 58% of the total) across all top-k settings, even exceeding that of accessing the SSD (i.e., *PD*, physical devices). `io_uring` shows moderate improvements over `libaio` since it does not have system calls [25]. SPDK achieves the highest IOPS by bypassing the traditional kernel stack completely. Note that these throughput numbers are all measured without considering the computation cost in Figure 9b. Given that `io_uring` and `libaio` have already failed to meet the expected IOPS, we therefore choose to build our serving storage over SPDK.

Customizing the storage stack. Adopting SPDK means dropping the support from the kernel, and we hence need to manage the data layout of indexes, the I/O control of raw devices, and the multi-device space allocation. We demonstrate our serving storage stack, as shown in Figure 10.

- *Data layout.* The index data consists of metadata and cluster lists. Metadata includes the index name, mappings of each cluster and its physical location (SSD id and LBA), pruning models, and the centroid index. Since these structures are small and can reside in memory without external reads at runtime, we simply store them as regular files on a local SSD. In contrast, the efficiency-critical cluster lists are placed directly on the logical blocks of the NVMe SSD array. Each cluster list occupies a contiguous block range of a single SSD (e.g., a cluster list uses three consecutive LBAs of SSD 11 in Figure 10), so that reading one cluster only requires a single I/O command, avoiding multiple I/Os due to crossing devices or blocks.
- *I/O control.* For the only runtime external I/O–loading cluster lists–we bypass system calls and directly submit NVMe commands to hardware queues while polling for completions. With this direct control of devices, we optimize submission at hardware granularity. I/O commands are enqueued to host-side NVMe queues in batches, and each NVMe device is notified with a single PCIe doorbell knock per batch. This eliminates hundreds of per-command PCIe round-trips (at the microsecond scale) and significantly reduces CPU overhead on the critical read path.
- *Space allocation.* Since all clusters are padded to a fixed

size, we can pre-allocate cluster-aligned regions on SSDs, avoiding fragmentation and complex file-system allocators. Hence, on raw NVMe devices, we exploit this property with a unified chunk-based free-list allocator (e.g., 64 MB per chunk) that manages SSD space for all indexes. It allocates and recycles fixed-size regions for deploying and deleting indexes at the chunk granularity. Each index then partitions its chunks into consecutive block ranges sized to a cluster list and assigns each range to a single cluster.

4.3 Leveling-learned Search Pruning

Problem and purpose. In clustering-based search, the number of probed clusters (i.e., *nprobe*) determines recall and performance. The trade-off is that an overly large *nprobe* significantly increases the I/O and computation cost due to redundant scans, while an overly small *nprobe* hurts recall.

Adapting nprobe by pruning. Many prior works such as Quake [43], Auncel [64], and LAET [36] propose early termination in pruning, which seems suitable for our adaptive adjustment of *nprobe*. However, they in fact cannot be applied to our scenario due to the lack of support for changing top-*k* of requests and the strong dependence on intermediate results. First, existing methods typically make early-termination rules for a fixed top-*k* while varying target recall. In contrast, in our services, the recall target can be predetermined from scenarios (e.g., 90% in online search) but top-*k* can vary across requests (e.g., 100-3,000 in online search).

Second, LAET also relies on intermediate features such as the distances to the current 1st and 10th neighbors after probing a subset of the nearest candidate clusters located. Similarly, Quake and Auncel need to check after each newly scanned cluster whether the recall target is met. Such iterative probe–compute–decide loops serialize cluster I/Os, and thus are unfriendly to the batched SSD reads for cluster lists, leading to reduced bandwidth utilization.

Leveling-learned search pruning. We propose the leveling-learned search pruning (LLSP) based on gradient boosting decision trees (GBDT) [2] [19,32]. First, to address varying top-*k* values, LLSP has a router model for choosing the maximum range according to the different search difficulty (i.e., the distribution of query vector and the top-*k* value) of requests. Second, it has a group of multi-level learning-based models for pruning redundant scans. At this stage, to keep batched cluster reads, we avoid using intermediate search results as features, and instead use only pre-search information: the query vector, the top-*k* value, and statistics of its distances to all nearest candidate centroids [3].

[2] GBDT is an effective architecture for pruning of online search because of its fast training (e.g., minute-level overhead for 1 million entries), low-overhead inference (e.g., ~10-30 us per prediction), and small memory footprint (e.g., only hundreds of KB per model).

[3] Feature-importance analysis shows query vectors and centroid-distance ratios can contribute nearly 50–70% of predictive power [36].

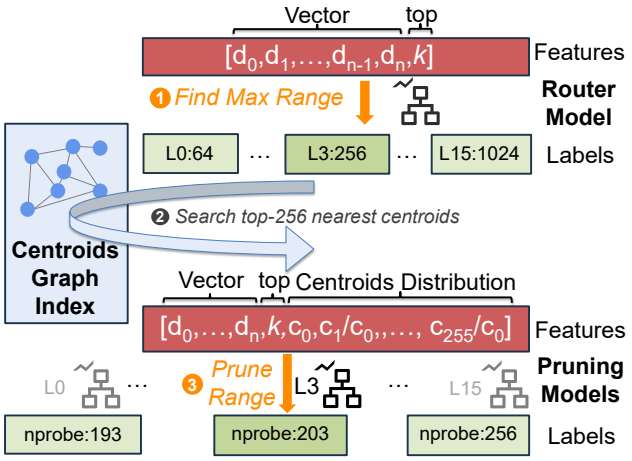


Figure 11: *Pruning workflow of search during online serving.*

Online-offline workflow. In detail, we first present the pruning process of search in online serving and then describe how the models are trained in offline building.

- *Online serving workflow.* As shown in Figure 11, we first feed the query vector and its requested top-*k* into the router model, which predicts a coarse maximum search range's level (e.g., level 3 with 256 as *nprobe* here, L3:256) as the upper bound of *nprobe*. Then the centroids graph index returns the top-*nprobe* nearest centroids. Finally, LLSP constructs pruning features from the query, top-*k*, and the centroid–distance distribution (the nearest centroid-query distance and relative ratios of the following 255 centroids' to the 1st centroid's), and applies the corresponding level-specific (i.e., level 3 here) pruning model to refine *nprobe* for batched loading nonredundant clusters from SSDs.
- *Offline training workflow.* We first set a series of range levels with increasing upper bounds on *nprobe* (e.g., 64 to 1,024 with a step of 64). From a recent time window (e.g., the previous 1 day for search, 1 hour for recommendation and ads), we uniformly sample about 1% of logged items as the training supervision. This is because most production traces show up to 90% duplication in short windows [15, 53, 54]. To avoid the overhead of brute-force ground truth, we approximate labels by running non-pruning search with a large *nprobe* (e.g., 4,096). We then train the router model by the procedure that, for each query and business-required top-*k*, finding the smallest level whose range meets the target recall and using the pair (query, top-*k*) as features and the level as label. Afterward, within each level, we derive labels for training pruning models by gradually decreasing *nprobe* from the level's maximum until recall drops to the threshold; the resulting *nprobe* is taken as the label. And the query, top-*k*, and centroid-distance distributions under the maximum *nprobe* form the feature vectors together. With the labels (i.e., actual *nprobe* minimized) and these features, we train the pruning model of each level.

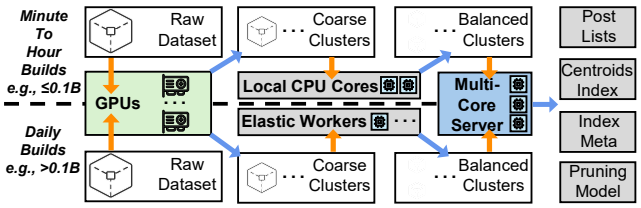


Figure 12: *Pipelines of the three-step index construction.*

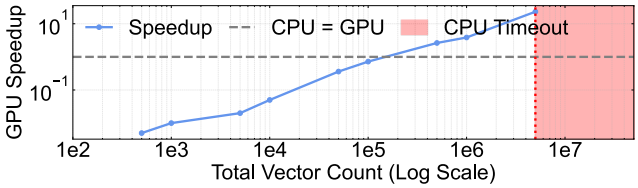


Figure 13: *The speedup ratio of GPU clustering across varying vector counts with an NVIDIA L20 vs. 48 CPU cores.*

4.4 GPU-accelerated and Elastic Building

To reduce the index construction time for million- to billion-scale datasets from days to tens of minutes, we leverage GPU-based acceleration and further design an elastic pipeline to scale the computation from a distributed CPU pool.

Procedure and issues of existing index construction. Again, we first examine the status quo (i.e., the index construction in SPANN). There are three steps. First, it applies hierarchical k-means to partition vectors into many size-bounded clusters, avoiding oversize and unbalanced partitions. Then, it performs closure multi-cluster assignment that duplicates boundary vectors, using RNG rules [52] to control redundant copies. Finally, it builds an in-DRAM graph on the centroids of all clusters for locating the nearest clusters to the query during online serving.

We can see that there are two main drawbacks in the first clustering step, which incurs the main overhead (up to ∼60-80%) of existing constructions. First, k-means on large-scale datasets incurs repeated distance calculations on high-dimensional vectors and thus places a heavy load which is beyond the capability of a CPU. Second, when vectors increase to billions from millions, it still requires multiple hours of building, requiring paralleling about 10^2-$10^3\times$ resources as data scaling. However, the current SPANN construction can only use one CPU in building and thus takes days to finish building a billion-scale index, which is unacceptable for production services.

GPU acceleration and elastic scaling. Hence, we exploit GPUs to accelerate centroid initialization and iterative balancing, and employ the elastic CPU pool to address the scaling of datasets. Figure 12 shows the workflow of our three-step index construction pipeline.

Generating coarse centroids with GPUs. First, for datasets of all scales, we use GPUs to perform coarse clustering. In this step, we do not directly split clusters down to their final size, because we found that GPU acceleration is not always effective[4]. As shown in Figure 13, GPUs provide up to orders-of-magnitude speedups on multi-million–scale and more vectors, while for small jobs (e.g., clustering on $<10^5$ 128-dim vectors), extra host–device transfers can dominate, making GPUs slower than the CPUs.

Constructing balanced posting lists. After the early clustering phase on GPUs, most clusters are already smaller than the threshold (e.g., 10^5 by default). We then use CPUs to perform fine-grained splitting and redundant padding of boundary vectors. Depending on the application scenario and dataset scale, we adopt two execution schemes.

For minute-to-hour builds of common-scale datasets (e.g., $<$ 0.1B vectors from recommendation and advertising), we employ local CPU cores to immediately finish the fine splitting and duplication with RNG checking, avoiding network transfers and extra scheduling overhead.

For search services that require daily rebuilds on larger corpora, elastic workers in online CPU clusters perform the fine-grained clustering and padding during off-peak hours, opportunistically harvesting idle resources from existing clusters under diurnal load patterns (i.e., lighter utilization at night and heavier loads during the day).

In addition, to avoid impacting real-time services, we enforce a QoS policy: whenever online and offline jobs contend for resources, online traffic always has priority and the index-building task on that node is preempted and terminated, and would retry later. To control the resulting tail latency, we further introduce task re-assignment and node eviction: once a task exceeds a retry threshold, it is reassigned to another node, and the original node is temporarily removed from the resource pool, preventing a few unstable nodes from dominating the overall construction time.

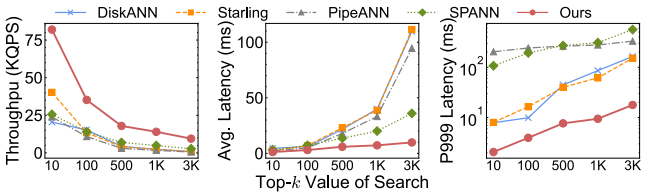
Building final index data. Finally, balanced and replicated clusters generated by both local CPUs of GPU servers and elastic CPU workers are uniformly prepared for releasing. They are consolidated on the multi-core servers to produce deployable index files, including posting lists, centroid index, metadata, and the pruning model.

5 Evaluation

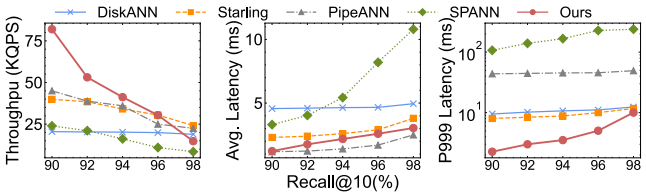
In this section, we conduct extensive experiments to evaluate the practical impacts of HELMSMAN, summarized as follows:

- HELMSMAN achieves 2-16× throughput of baselines, and can meet latency SLAs to replace in-DRAM HNSW (§5.2).
- HELMSMAN achieves 1.6-7.5× SSD bandwidth utilization of other DRAM-SSD systems, yielding up to 87% improvement by simply upgrading SSDs (§5.3).
- Under the same average recall, HELMSMAN ensures that >80% of queries individually meet the target recall by near 30 percentage points over SPANN (§5.4).

[4] At present, the development our GPU K-means is based on Faiss.



(a) *Performance across various top-k of search, under recall = 90%.*



(b) *Performance as the top-10 search, under various recall rates.*

Figure 14: *Comparisons about throughput, average and tail latency under different top-k and recall, using SIFT100M.*

- GPU acceleration enables constructing 0.1B-vector indexes within 1 hour. And by elastically scaling resources, 10B vectors can be built within 10 hours (§5.5).
- HELMSMAN can reduce DRAM usage by over 90% and improve throughput-per-dollar by 8.3× for the serving deployment (§5.6).

5.1 Setup

Dataset	Scale	Dim.	Total size	Top-*k* Range
SIFT	0.1B&10B	128	12 GB&1.2 TB	10–3,000
RedSrch	0.5B&10B	64	30 GB&0.6 TB	100–3,000
RedRec	0.1B	64	6 GB	100–1,000
RedAds	20M	128	2.5 GB	100–3,000
RedCM	0.1B	64	6 GB	100–500
RedRAG	4M	1024	4 GB	10–100

Table 2: Statistics of the evaluated datasets.

Workloads. We evaluate on the public SIFT and five production datasets, as shown in Table 2. SIFT10B is created by 10× replicating SIFT1B. Real datasets span 4M–10B vectors from our main services mentioned. Similarity metric is the L2 distance. For SIFT, we uniformly generate query top-*k* values ranging from 10 to 3,000, while for other datasets, top-*k* values are sampled from our real production traces.

Baselines. We compare HELMSMAN with DiskANN, Starling, PipeANN, SPANN, and HNSW. For DRAM-SSD baselines, we use their released implementations. We tune beam widths of graph-based systems (16 for DiskANN and Starling, 32 for PipeANN, unless otherwise specified) to achieve their optimal performance, since multi-SSD setups offer sufficient bandwidth. We set DRAM budget as 25% of the size of raw datasets [22]. For HELMSMAN and SPANN, we reduce the replication factor to 4, with centroids accounting for 8% of the total scale, thereby setting the DRAM:SSD ratio to 1:20 to match the hardware capacity. We include in-DRAM HNSW that is widely used in production for low-latency services, which serves as a strong performance reference for purely memory-resident deployments but with higher DRAM cost.

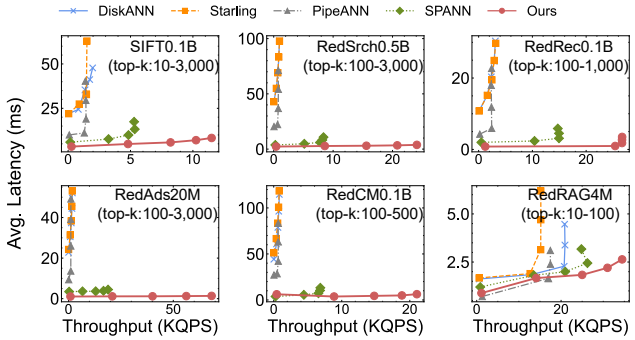


Figure 15: *Trends of throughput and average latency.*

Specifications. All DRAM–SSD tests are conducted on our testbed equipped with the 96-core AMD EPYC 9654 CPU, 12×96 GB DDR5 DRAM, 12×1.92 TB PCIe-5.0 NVMe SSDs. In-DRAM distributed HNSW uses our standard configuration with 32-core CPUs and 256 GB DDR5 DRAM to hold shards, a configuration commonly adopted in production.

5.2 Search Performance

End-to-end performance. Overall, HELMSMAN achieves the best end-to-end efficiency across a wide range of top-*k* and recall targets, as shown in Figure 14.

Varying top-k (Figure 14a). With recall fixed at 90%, HELMSMAN sustains the highest throughput on SIFT0.1B across all top-*k* values. As top-*k* increases from 10 to 3,000, its throughput degrades more slowly than that of other DRAM-SSD systems, while average latency stays within 10 ms and P99.9 latency remains well below 20 ms. For graph-based baselines (DiskANN, Starling, PipeANN), larger top-*k* forces longer greedy walks due to larger candidate lists. Starling improves top-10 performance via optimized layout but is less effective for large top-*k*. The reason is that, as top-k increases, the search depth of graph-based search grows significantly. For example, for top-1000 on the SIFT0.1B, both the candidate queue length and the number of search hops increase substantially to around 1,500 or more. The number of accessed SSD pages also rises significantly, which leaves little room for gains from layout reordering of Starling. Meanwhile, SPANN is constrained by the traditional I/O software stack. Consequently, both throughput and latency for baselines deteriorate rapidly.

Varying recall (Figure 14b). Fixing top-*k* at 10, HELMSMAN delivers competitive throughput at 90-96% recall and keeps clear advantages when recall enters the high-accuracy regime. Both average and tail latency grow smoothly with the recall, and HELMSMAN maintains sub-10 ms average latency

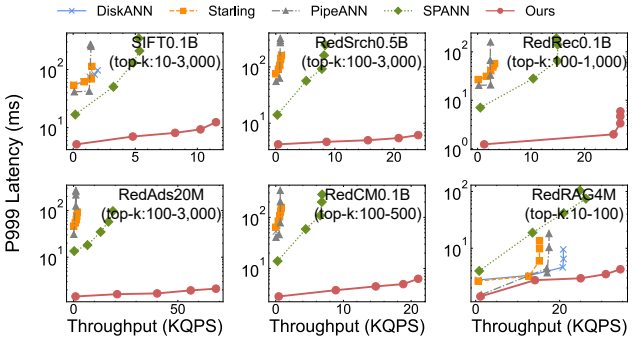


Figure 16: *Trends of throughput and P999 tail latency.*

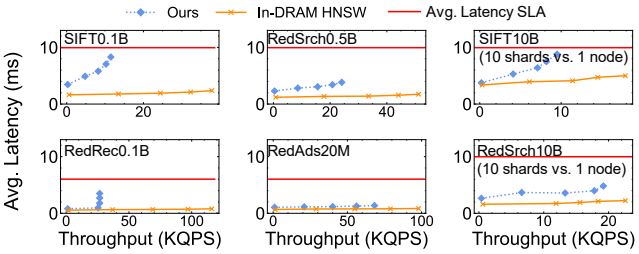


Figure 17: *Comparison with in-DRAM HNSW deployments.*

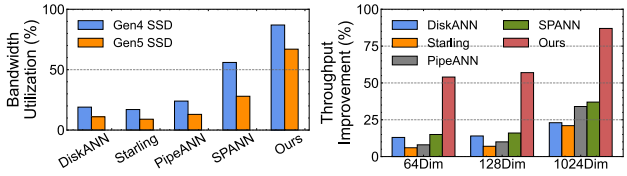
and the lowest P99.9 latency across the entire recall range.

Throughput vs. latency. We evaluate workloads from generated SIFT and productions with service-specific top-*k* and a 90% recall target, ramping threads until all systems saturate.

Throughput vs. average latency. As shown in Figure 15, graph-based systems can incur ∼120 ms average latency at only 0.5-3 KQPS because large top-*k* requires long greedy walks for longer length of candidate lists (e.g., 4,000 for top-3,000). SPANN benefits from high-bandwidth SSDs, and HELMSMAN further pushes the frontier, achieving up to 30× throughput while keeping average latency within 5-10 ms.

Throughput vs. tail latency. For P99.9 latency (Figure 16), graph-based systems and SPANN exhibit steep tail blow-ups once saturated. By clustering-based search and our customized I/O stack, HELMSMAN maintains up to an order-of-magnitude lower P99.9 latency, keeping it around ∼10 ms.

Comparisons with in-DRAM deployments. Figure 17 shows results. When HNSW fits in the same 96-core node (20M–0.5B datasets), HELMSMAN achieves about 25–70% of the throughput of in-DRAM HNSW deployments at 90% recall, while always satisfying the average-latency SLAs. For the 10B-scale SIFT and RedSrch, the production HNSW uses 10 shards with a total of 2.5 TB DRAM and 320 CPU cores (standard 32-core, 256-GB nodes). In contrast, HELMSMAN reaches roughly 47-85% of their throughput under the same SLA using a single 96-core machine with only 160–330 GB of DRAM. HELMSMAN meets SLAs with modestly higher latency, but reduces CPU usage by about 3–4× and DRAM consumption by nearly an order of magnitude.



(a) *I/O Bandwidth utilization on PCIe-4.0 and PCIe-5.0 SSDs.* (b) *Improvements by upgrading SSDs from PCIe-4.0 to PCIe-5.0.*

Figure 18: *Impacts of different generations of NVMe SSDs.*

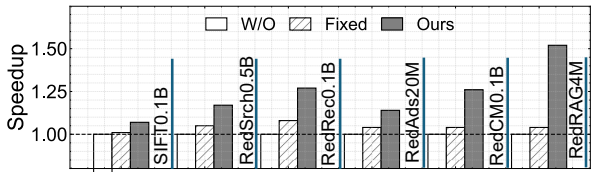


Figure 19: *Performance speedup by the pruning module.*

5.3 Storage I/O Impact

Bandwidth utilization. As shown in Figure 18a, on RedSrch0.5B, we measure the bandwidth utilization of a 12-SSD array with both PCIe-4.0 [11] and PCIe-5.0 drives [12]. Graph-based systems utilize less than 20% of SSD bandwidth, while clustering-based SPANN also merely reaches about 55% (Gen4). In contrast, HELMSMAN pushes utilization to ∼85% on Gen4 and ∼70% on Gen5, showing that its batched, device-direct I/O path is much closer to the device limits.

Gains from hardware upgrades. In Figure 18b, we then report throughput gains when upgrading SSDs from PCIe 4.0 to 5.0 on 64-dim RedSrch, 128-dim RedAds, and 1024-dim RedRAG workloads. Graph-based systems (DiskANN, Starling, and PipeANN) improve by only 10-30%, and SPANN by up to 40%, indicating they are largely I/O-latency- or software-stack-bound. In contrast, HELMSMAN gains about 55% at 64- and 128-D and 87% at 1024-D, demonstrating that its performance scales more efficiently with available bandwidth.

5.4 Pruning Efficiency

For the public SIFT dataset, we randomly sample 1M vectors as the training set and use 10% of them as queries; for production datasets, we extract 110K consecutive queries from online logs, use the first 100K as the training set, and reserve the last 10K as queries, consistent with our production scenario. By default, we set the number of iterations to 500 and the learning rate to 0.2.

Performance gains. Our pruning module consistently accelerates search across all datasets, as shown in Figure 19. Compared with the non-pruning baseline, it yields 1.1–1.6× throughput, and still provides 5–25% higher throughput than the fixed pruning policy. The benefit is most pronounced on RedRAG4M because its workloads use small top-k values (i.e., 10–100), where the optimal search range (i.e., *nprobe*) tends to fluctuate more, leading to more redundant scans.

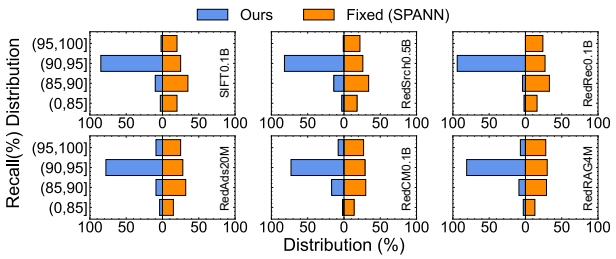
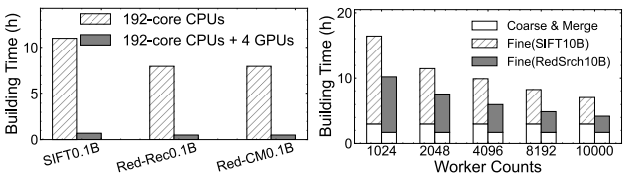


Figure 20: *Search quality improved by the pruning module.*



(a) *Comparison of single node.* (b) *Speedup from elastic scaling.*

Figure 21: *Acceleration of GPUs and distribution.*

Accuracy gains. As shown in Figure 20, LLSP also leads to a more stable search quality under the same average recall. In contrast to the fixed policy, which fails to meet the target recall for over 40% of queries, our method ensures that over 80% of queries meet the target recall (i.e., 90%), effectively reducing low-recall outliers across all six datasets.

Dataset	Model	Feature Importance
RedSrch	Router	*Query*:67.3% *k*:32.7%
	Pruning	*Query*:34.3% *k*:15.2% *Centroids*:50.5%
RedRAG	Router	*Query*:74.1% *k*:25.9%
	Pruning	*Query*:48.3% *k*:7.8% *Centroids*:43.9%

Table 3: Feature importance of leveling-learned pruning.

Feature importance. We further show the importance of different features in the leveling-learned pruning model in Table 3. RedSrch0.5B and RedRAG4M are chosen as representatives of production datasets with different dimensions and top-*k* ranges. RedSrch has a larger top-*k* range (100–3,000) and smaller dimension (64), while RedRAG has a smaller top-*k* range (10–100) and larger dimension (1024). Results show that the query features (e.g., query coordinates and distances to centroids used during pruning) are important for both router and pruning models, indicating that the local distribution of queries is a key factor for determining the search range and pruning ratio. Meanwhile, the importance of top-*k* is higher for the router model than the pruning model, which is reasonable since the router determines the search range (i.e., *nprobe*) based on the target top-*k*, while the pruning model further prunes clusters within the smaller selected search range.

5.5 Construction Evaluation

GPU acceleration. On a single 192-core node, CPU-only construction of 0.1B-scale indexes takes about 9–12 hours. Offloading the coarse clustering stage to 4 NVIDIA L20 GPUs reduces the total build time to within an hour, as shown in Figure 21a (up to ∼10× speedup), transforming hours-long offline jobs into minute-level builds for high-freshness services (e.g., recommendation and advertising).

Elastic scaling. For 10B-scale datasets, using GPUs for coarse clustering and single-node merging for final indexes (i.e., Coarse & Merge), we further parallelize the fine-grained balancing stage (i.e., Fine) across an elastic CPU cluster. Increasing the number of CPU workers from 1,024 to 10^4 cores reduces end-to-end building time from over 16 hours to about 4–7 hours, as shown in Figure 21b, enabling hours-level reconstruction of billion-scale indexes.

5.6 Cost Efficiency

System	DRAM (GB, $)	Gen5 SSD (GB, $)	Throughput (KQPS)	Storage Eff. (QPS/$)
HNSW	123, 1K	N/A	51	51
PipeANN	8, 64	260, 52	0.8	7
SPANN	8, 64	162, 32	8.4	88
Ours	8, 64	162, 32	24	**250**

Table 4: Comparison of cost efficiency on RedSrch0.5B.

System	DRAM (TB, $)	Gen5 SSD (TB, $)	Throughput (KQPS)	Storage Eff. (QPS/$)
HNSW	2.5, 20K	N/A	23	1.2
Ours	0.16, 1.3K	3.2, 0.6K	19	**10**

Table 5: Comparison of cost efficiency on RedSrch10B.

System	Cloud Price (norm. price / hour)	Build Time (hour)	Offline Cost (norm. cost)
HNSW	1 (96 Core)	1.5	1.5
Ours	1.3 (96 Core + 4 GPU)	1.3	1.7

Table 6: Comparison of construction cost on RedSrch0.5B.

Cost efficiency of online serving. We compare storage cost and throughput on search workloads to show online serving costs. On RedSrch0.5B (Table 4), under production top-*k* workloads, PipeANN sustains only 0.8 KQPS, yielding 7 QPS/$, even lower than in-DRAM HNSW (51 QPS/$). SPANN reaches 88 QPS/$, while HELMSMAN further improves to 250 QPS/$ (5.4× over HNSW, 2.9× over SPANN). On RedSrch10B (Table 5), by replacing a 10-shard HNSW deployment using ∼2.5 TB DRAM with each single node using 32 cores and 256 GB DRAM, HELMSMAN boosts cost efficiency from 1.2 to 10 QPS/$ (8.3×) by saving 90% of DRAM usage at an additional SSD cost of only $0.6K.

Cost efficiency of offline building. In addition to comparing our offline building time with the CPU-only SPANN, we also estimate the single-node CPU-GPU construction cost of HELMSMAN and CPU-based HNSW we deployed previouly by comparing the normalized cloud price of CPU-only and CPU-GPU instances [4]. As shown in Table 6, our GPU-accelerated construction achieves similar cost compared with CPU-only HNSW building, because of higher price of GPU instances but shorter build time. And as the GPU-based clustering implementation is developing [60], we expect the cost of clustering-based indexes' construction to further reduce, making it more cost-effective.

6 Deployment

6.1 Deployment Status

At the time of publication, HELMSMAN has been rolling out as a unified ANN layer in RedNote's production. We use HELMSMAN to support all services, and all-flash servers are gradually replacing previous in-DRAM deployments across search, recommendation, advertising, and other vector services in the coming months. Currently, with only ∼40 all-flash servers-each with 0.7–1.1 TB DRAM and 12 NVMe SSDs-HELMSMAN already sustains online workloads that previously required about 35,000 CPU cores and ∼0.35 PB DRAM in in-DRAM deployments. In the offline construction phase, HELMSMAN can leverage up to 10,000 CPU cores from online clusters during low-traffic periods (e.g., late-night hours) to accelerate large-scale builds. Looking forward, RedNote plans to migrate the majority of in-DRAM deployments onto HELMSMAN, with projected infrastructure cost savings on the order of tens of millions of dollars per year.

6.2 Operational Lessons and Future Work

Performance bottlenecks stem from local hotspots of queries and SSDs' die-level conflicts. In most indexes, the overall deployment remains smooth and stable. When SSD bandwidth and CPU utilization remain unsaturated, latency stays within SLAs and throughput scales with additional search threads. However, in the early stage of trial operation, for a few recommendation indexes, adding CPU cores failed to increase throughput, even though SSD bandwidth remained below 20%. Log replay shows that transient query bursts can target the same nearest clusters and logical blocks, causing internal chip conflicts [31] and high-latency SSD accesses. To mitigate this, we place a few redundant copies of the cluster lists on NVMe SSDs, which reduces conflicts and raises the throughput ceiling by 1.5–2×, while incurring only a minor extra SSD space overhead.

Memory bandwidth becomes the bottleneck before external I/O bandwidth is fully utilized. In our all-flash deployment, 12 Gen5 SSDs provide roughly 140 GB/s aggregate external I/O bandwidth, but only about 70% can be utilized in practice. The bottleneck is the limited effective memory bandwidth of modern 12-channel DDR5 servers, typically around 300–350 GB/s, which must serve SSD-to-DRAM transfers, DRAM-to-CPU rereads, and nearest-centroid search. Consequently, adding more SSDs brings limited benefit once memory bandwidth is saturated, as observed in our early more SSDs equipments (e.g., 20 Gen5 SSDs experiments). This imbalance is likely to worsen as modern CPUs expose enough PCIe lanes (e.g., 128-160 lanes [1]) to attach over 32 Gen5 SSDs, providing external bandwidth beyond what current memory systems can sustain. For the commercially available devices, the direct I/O-to-cache techniques such as Intel DDIO [6] and AMD SDCI [3] may mitigate this issue, but currently these methods only work for network devices and their SSD ecosystem remains immature. We think that extending such support to SSD-based systems might be a promising future direction. At the same time, recent studies [27, 28, 57] have used GPUs to overcome the limitations of CPU compute power and host memory bandwidth, while also taking advantage of the increasingly abundant external I/O bandwidth provided by NVMe SSDs. We plan to explore their optimizations in our future deployments.

Large top-k can be more important than extremely high recall. In production search, recommendation, and advertising systems, first-stage ANNS is typically used as a candidate generator rather than the final decision maker. Since single-vector similarity only provides a compressed semantic signal, downstream stages usually apply post-filtering based on scalar attributes, or rescore the retrieved candidates using richer features and the original content, such as text, images, and videos [21, 37, 63]. Therefore, in our workloads, retrieving a large candidate pool, e.g., top-k from 100 to 1000 at around 90% recall, is often more valuable than pursuing extremely high recall such as 98–99.9% for small top-k requests. The latter can substantially increase ANNS search cost but brings limited end-to-end business gain, because downstream stages mainly benefit from having enough valid and diverse candidates to filter and rescore, rather than from marginal improvements in the vector recall of a small candidate set. This suggests that future ANNS systems should be evaluated not only by recall@10 [48], but also by performance under large top-k, which better matches multi-stage pipelines.

In-place updates remain challenging. Dynamic updates are important for freshness in vector data management, especially to keep up with the near-real-time and streaming training of embedding models. In a typical 0.1B-scale recommendation workload with hourly rebuilding and ∼25–30 KQPS of search, fully replacing periodic rebuilds with in-place updates would require roughly 25–30 KOPS of updates (i.e., insertions and deletions) while concurrently serving online search. This is beyond the update and search throughputs of recent state-of-the-art large-scale dynamic ANNS systems [23, 58]. Therefore, supporting both high-rate updates

and high-performance search remains a challenging direction. Our current deployment adopts a hybrid design [24,26,50,55]: the main SSD-resident index is periodically rebuilt, recent insertions are maintained in an auxiliary in-memory index e.g., (HNSW and IVF), and deletions are tracked by a tombstone bitmap. Queries search both indexes, merge candidates, and filter tombstoned vectors. This preserves freshness with modest complexity, but still incurs extra memory usage and unavoidable rebuild costs.

7 Related Work

DRAM-SSD ANNS. DRAM–SSD ANNS includes graph-based systems (e.g., DiskANN [49], Starling [56], PipeANN [22]) and clustering-based designs (e.g., SPANN [17, 58]). Though graph-based schemes are popular for issuing fewer I/Os than clustering-based search, our findings show that SSD latency and serial access can make them slower than clustering on modern SSDs with the near-DRAM bandwidth [25,34,35], particularly for top-k up to the thousands in production [21,37,63]. Hence, we integrate clustering methods with high-bandwidth NVMe SSD arrays.

Pruning for ANNS. Quake [43], LAET [36], Auncel [64], and ANSMET [38] study pruning and early stopping for in-DRAM ANNS, where the search can iteratively check intermediate results and terminate once convergence-like conditions are met. However, such fine-grained, stateful control conflicts with SSD-friendly batched I/Os, which prefer issuing large sequential reads without per-step feedback. Therefore, instead of these intermediate-result-based methods, we propose our pruning module trained on historical traces, which predicts search ranges without relying on intermediate states.

Acceleration of index building. CAGRA [46], RAFT [7], ParlayANN [41], and Faiss [30] accelerate ANNS index construction and clustering using highly parallel graph builders and heterogeneous acceleration. We adapt several of their techniques (e.g., GPU-based k-means and parallel graph construction) in our offline index-building pipelines.

8 Conclusion

Our experience at RedNote shows that, under industrial workloads with large top-k queries, graph-based ANNS on SSDs suffers from long, latency-bound search paths and loses competitiveness. In contrast, pairing high-bandwidth NVMe arrays with the clustering-based search allows DRAM–SSD designs to approach in-DRAM performance (both throughput and latency) while greatly reducing hardware cost. HELMSMAN demonstrates this path in practice, combining a custom storage stack, adaptive pruning, and fast elastic construction to deliver cost-effective, high-performance ANNS at scale.

Acknowledgment

We thank all anonymous reviewers and our shepherd, Asaf Cidon, for their valuable feedback and helping us improve our paper significantly. We also thank engineers at Xiaohongshu Inc. for their efforts in deploying HELMSMAN in production. This work was conducted by Yuchen Huang and Baiteng Ma during their internship at RedNote Engine Architecture Department. Erci Xu is the corresponding author. Yuchen Huang and Chuliang Weng are supported by the National Natural Science Foundation of China (Grant No.62272171).

References

[1] 5th Generation AMD EPYC™ Server CPUs. https://www.amd.com/en/products/processors/server/epyc/9005-series.html.

[2] Accelerating Vector Database Performance through Disk-Based Storage. https://americas.kioxia.com/en-us/business/resources/performance-brief/cm7-vector-db-r6615-performance-brief.html.

[3] AMD Smart Data Cache Injection. https://docs.amd.com/api/khub/documents/gLSrfVtcWNt~1fzExUSiIg/content.

[4] AWS EC2 Pricing. https://aws-pricing.com/.

[5] Everything you need to know about xiaohongshu. Explainer piece; positions Xiaohongshu as a lifestyle social commerce platform with 300M+ monthly active users. https://restofworld.org/2025/rednote-xiaohongshu-what-to-know/.

[6] Intel Data Direct I/O Technology. https://www.intel.com/content/www/us/en/io/data-direct-i-o-technology.html.

[7] RAPIDS RAFT: Reusable Accelerated Functions and Tools for Vector Search and More. https://github.com/rapidsai/raft.

[8] RedNote (Xiaohongshu Inc). https://rednotes.co/.

[9] Reduce costs with disk-based vector search. https://opensearch.org/blog/reduce-cost-with-disk-based-vector-search/.

[10] Samsung DDR5 Data-centric DRAM Memory. https://semiconductor.samsung.com/dram/ddr/ddr5/.

[11] Samsung PCIe-Gen4.0 PM9A3 Data-centric NVMe SSD. https://semiconductor.samsung.com/ssd/datacenter-ssd/pm9a3/.

[12] Samsung PCIe-Gen5.0 PM9D3A Data-centric NVMe SSD. https://semiconductor.samsung.com/ssd/datacenter-ssd/pm9d3a/.

[13] SIFT dataset. http://corpus-texmex.irisa.fr/.

[14] Storage Performance Development Kit (SPDK). https://github.com/spdk.

[15] Mozhdeh Ariannezhad, Sami Jullien, Ming Li, Min Fang, Sebastian Schelter, and Maarten de Rijke. ReCANet: A Repeat Consumption-Aware Neural Network for Next Basket Recommendation in Grocery Shopping. In *Proceedings of the 45th International ACM SIGIR Conference on Research and Development in Information Retrieval*, SIGIR, 2022.

[16] Grant Ayers, Jung Ho Ahn, Christos Kozyrakis, and Parthasarathy Ranganathan. Memory Hierarchy for Web Search. In *2018 IEEE International Symposium on High Performance Computer Architecture*, HPCA, 2018.

[17] Qi Chen, Bing Zhao, Haidong Wang, Mingqin Li, Chuanjie Liu, Zengzhong Li, Mao Yang, and Jingdong Wang. SPANN: Highly-efficient Billion-scale Approximate Nearest Neighbor Search. In *Advances in Neural Information Processing Systems 34*, NeurIPS, 2021.

[18] Xiaopeng Fan, Song Yan, and Chuliang Weng. Structured storage for ubiquitous operating systems. *SCIENTIA SINICA Informationis*, 54, 2024.

[19] Jerome H. Friedman. Greedy Function Approximation: A Gradient Boosting Machine. *Annals of Statistics*, 29(5), 2001.

[20] Cong Fu, Chao Xiang, Changxu Wang, and Deng Cai. Fast Approximate Nearest Neighbor Search With Navigating Spreading-out Graphs. *Proceedings of the VLDB Endowment*, 12, 2019.

[21] Luyu Gao, Zhuyun Dai, Tongfei Chen, Zhen Fan, Benjamin Van Durme, and Jamie Callan. Complement Lexical Retrieval Model with Semantic Residual Embeddings. In *Advances in Information Retrieval (Proceedings of ECIR 2021)*, ECIR, 2021.

[22] Hao Guo and Youyou Lu. Achieving Low-Latency Graph-Based Vector Search via Aligning Best-First Search Algorithm with SSD. In *Proceedings of the 19th USENIX Symposium on Operating Systems Design and Implementation*, OSDI, 2025.

[23] Hao Guo and Youyou Lu. OdinANN: Direct Insert for Consistently Stable Performance in Billion-Scale Graph-Based Vector Search. In *24th USENIX Conference on File and Storage Technologies*, FAST, 2026.

[24] Rentong Guo, Xiaofan Luan, Long Xiang, Xiao Yan, Xiaomeng Yi, Jigao Luo, Qianya Cheng, Weizhi Xu, Jiarui Luo, Frank Liu, Zhenshan Cao, Yanliang Qiao, Ting Wang, Bo Tang, and Charles Xie. Manu: a cloud native vector database management system. *Proceedings of the VLDB Endowment*, 15, 2022.

[25] Gabriel Haas and Viktor Leis. What Modern NVMe Storage Can Do, And How To Exploit It: High-Performance I/O for High-Performance Storage Engines. *Proceedings of the VLDB Endowment*, 2023.

[26] Yuchen Huang, Xiaopeng Fan, Song Yan, and Chuliang Weng. Neos: A NVMe-GPUs Direct Vector Service Buffer in User Space. In *Proceedings of the 40th IEEE International Conference on Data Engineering*, ICDE, 2024.

[27] Yuchen Huang, Baiteng Ma, Erci Xu, and Chuliang Weng. Don't Surrender to Low QPS/$: Fast and Cost-Efficient ANNS with TridentANN. In *Proceedings of the 53rd Annual International Symposium on Computer Architecture*, ISCA, 2026.

[28] Haodi Jiang, Hao Guo, Minhui Xie, Jiwu Shu, and Youyou Lu. High-Throughput, Cost-Effective Billion-Scale Vector Search with a Single GPU. In *Proceedings of the 2026 ACM SIGMOD International Conference on Management of Data*, SIGMOD, 2026.

[29] Wenqi Jiang, Suvinay Subramanian, Cat Graves, Gustavo Alonso, Amir Yazdanbakhsh, and Vidushi Dadu. RAGO: Systematic Performance Optimization for Retrieval-Augmented Generation Serving. In *Proceedings of the 52nd Annual International Symposium on Computer Architecture*, ISCA, 2025.

[30] Jeff Johnson, Matthijs Douze, and Hervé Jégou. Billion-Scale Similarity Search with GPUs. *IEEE Transactions on Big Data*, 7, 2021.

[31] Yuhun Jun, Shinhyun Park, Jeong-Uk Kang, Sang-Hoon Kim, and Euiseong Seo. We ain't afraid of no file fragmentation: causes and prevention of its performance impact on modern flash SSDs. In *Proceedings of the 22nd USENIX Conference on File and Storage Technologies*, FAST, 2024.

[32] Guolin Ke, Qi Meng, Thomas Finley, Taifeng Wang, Wei Chen, Weidong Ma, Qiwei Ye, and Tie-Yan Liu. LightGBM: A Highly Efficient Gradient Boosting Decision Tree. In *Advances in Neural Information Processing Systems 30*, NeurIPS, 2017.

[33] Barrie Kersbergen, Olivier Sprangers, and Sebastian Schelter. Serenade - Low-Latency Session-Based Recommendation in e-Commerce at Scale. In *Proceedings*

of the 2022 International Conference on Management of Data, SIGMOD, 2022.

[34] Maximilian Kuschewski, Jana Giceva, Thomas Neumann, and Viktor Leis. High-Performance Query Processing with NVMe Arrays: Spilling without Killing Performance. In *Proceedings of the 2025 ACM SIGMOD International Conference on Management of Data*, SIGMOD, 2025.

[35] Baptiste Lepers, Oana Balmau, Karan Gupta, and Willy Zwaenepoel. KVell: The Design and Implementation of a Fast Persistent Key-Value Store. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP, 2019.

[36] Conglong Li, Minjia Zhang, David G. Andersen, and Yuxiong He. Improving Approximate Nearest Neighbor Search through Learned Adaptive Early Termination. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, SIGMOD, 2020.

[37] Sen Li, Fuyu Lv, Taiwei Jin, Guli Lin, Keping Yang, Xiaoyi Zeng, Xiao-Ming Wu, and Qianli Ma. Embedding-based Product Retrieval in Taobao Search. In *Proceedings of the 27th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*, KDD, 2021.

[38] Yiwei Li, Yuxin Jin, Boyu Tian, Huanchen Zhang, and Mingyu Gao. ANSMET: Approximate Nearest Neighbor Search with Near-Memory Processing and Hybrid Early Termination. In *Proceedings of the 52nd Annual International Symposium on Computer Architecture*, ISCA, 2025.

[39] Kejing Lu, Mineichi Kudo, Chuan Xiao, and Yoshiharu Ishikawa. HVS: Hierarchical Graph Structure Based on Voronoi Diagrams for Solving Approximate Nearest Neighbor Search. *Proceedings of the VLDB Endowment*, 15, 2021.

[40] Yu A. Malkov and D. A. Yashunin. Efficient and Robust Approximate Nearest Neighbor Search Using Hierarchical Navigable Small World Graphs. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 42, 2020.

[41] Magdalen Dobson Manohar, Zheqi Shen, Guy E. Blelloch, Laxman Dhulipala, Yan Gu, Harsha Vardhan Simhadri, and Yihan Sun. ParlayANN: Scalable and Deterministic Parallel Graph-Based Approximate Nearest Neighbor Search Algorithms. In *Proceedings of the 29th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming*, PPoPP, 2024.

[42] Kiran Kumar Matam, Hani Ramezani, Fan Wang, Zeliang Chen, Yue Dong, Maomao Ding, Zhiwei Zhao, Zhengyu Zhang, Ellie Wen, and Assaf Eisenman. QuickUpdate: a Real-Time Personalization System for Large-Scale Recommendation Models. In *Proceedings of the 21st USENIX Symposium on Networked Systems Design and Implementation*, NSDI, 2024.

[43] Jason Mohoney, Devesh Sarda, Mengze Tang, Shihabur Rahman Chowdhury, Anil Pacaci, Ihab F. Ilyas, Theodoros Rekatsinas, and Shivaram Venkataraman. Quake: Adaptive Indexing for Vector Search. In *19th USENIX Symposium on Operating Systems Design and Implementation*, OSDI, 2025.

[44] Aashiq Muhamed, Mona T. Diab, and Virginia Smith. CoRAG: Collaborative Retrieval-Augmented Generation. In *Proceedings of the 2025 Conference of the Nations of the Americas Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume 2: Short Papers)*, NAACL, 2025.

[45] Yuto Oikawa, Yuki Nakayama, and Koji Murakami. A Stacking-based Efficient Method for Toxic Language Detection on Live Streaming Chat. In *Proceedings of the 2022 Conference on Empirical Methods in Natural Language Processing*, EMNLP, 2022.

[46] Hiroyuki Ootomo, Akira Naruse, Corey Nolet, Ray Wang, Tamas Feher, and Yong Wang. CAGRA: Highly Parallel Graph Construction and Approximate Nearest Neighbor Search for GPUs. In *Proceedings of the 40th IEEE International Conference on Data Engineering*, ICDE, 2024.

[47] Chijun Sima, Yao Fu, Man-Kit Sit, Liyi Guo, Xuri Gong, Feng Lin, Junyu Wu, Yongsheng Li, Haidong Rong, Pierre-Louis Aublin, and Luo Mai. Ekko: A Large-Scale Deep Learning Recommender System with Low-Latency Model Update. In *Proceedings of the 16th USENIX Symposium on Operating Systems Design and Implementation*, OSDI, 2022.

[48] Harsha Vardhan Simhadri, George Williams, Martin Aumüller, Matthijs Douze, Artem Babenko, Dmitry Baranchuk, Qi Chen, Lucas Hosseini, Ravishankar Krishnaswamy, Gopal Srinivasa, Suhas Jayaram Subramanya, and Jingdong Wang. Results of the NeurIPS'21 Challenge on Billion-Scale Approximate Nearest Neighbor Search. In *Proceedings of the NeurIPS 2021 Competitions and Demonstrations Track*, NeurIPS, 2022.

[49] Suhas Jayaram Subramanya, Devvrit, Harsha Vardhan Simhadri, Ravishankar Krishnawamy, and Rohan Kadekodi. DiskANN: Fast Accurate Billion-point Nearest Neighbor Search on a Single Node. In *Advances in Neural Information Processing Systems 32*, NeurIPS, 2019.

[50] Yiping Sun, Yang Shi, and Jiaolong Du. A Real-Time Adaptive Multi-Stream GPU System For Online Approximate Nearest Neighborhood Search. In *Proceedings of the 33rd ACM International Conference on Information and Knowledge Management*, CIKM, 2024.

[51] Bing Tian, Haikun Liu, Yuhang Tang, Shihai Xiao, Zhuohui Duan, Xiaofei Liao, Xuecang Zhang, Junhua Zhu, and Yu Zhang. FusionANNS: An Efficient CPU/GPU Cooperative Processing Architecture for Billion-scale Approximate Nearest Neighbor Search. In *23rd USENIX Conference on File and Storage Technologies*, FAST, 2025.

[52] Godfried T. Toussaint. The Relative Neighborhood Graph of a Finite Planar Set. *Pattern Recognition*, 12, 1980.

[53] Kosetsu Tsukuda and Masataka Goto. Explainable Recommendation for Repeat Consumption. In *Proceedings of the 14th ACM Conference on Recommender Systems*, RecSys, 2020.

[54] Chenyang Wang, Min Zhang, Weizhi Ma, Yiqun Liu, and Shaoping Ma. Modeling Item-Specific Temporal Dynamics of Repeat Consumption for Recommender Systems. In *Proceedings of the World Wide Web Conference*, WWW, 2019.

[55] Jianguo Wang, Xiaomeng Yi, Rentong Guo, Hai Jin, Peng Xu, Shengjun Li, Xiangyu Wang, Xiangzhou Guo, Chengming Li, Xiaohai Xu, Kun Yu, Yuxing Yuan, Yinghao Zou, Jiquan Long, Yudong Cai, Zhenxiang Li, Zhifeng Zhang, Yihua Mo, Jun Gu, Ruiyi Jiang, Yi Wei, and Charles Xie. Milvus: A Purpose-Built Vector Data Management System. In *Proceedings of the 2021 ACM SIGMOD International Conference on Management of Data*, SIGMOD, 2021.

[56] Mengzhao Wang, Weizhi Xu, Xiaomeng Yi, Songlin Wu, Zhangyang Peng, Xiangyu Ke, Yunjun Gao, Xiaoliang Xu, Rentong Guo, and Charles Xie. Starling: An I/O-Efficient Disk-Resident Graph Index Framework for High-Dimensional Vector Similarity Search on Data Segment. In *Proceedings of the 2024 ACM SIGMOD International Conference on Management of Data*, SIGMOD, 2024.

[57] Yang Xiao, Mo Sun, Ziyu Song, Bing Tian, Jie Sun, Jie Zhang, Zeke Wang, Zonghui Wang, Wenzhi Chen, and Fei Wu. FlashANNS: GPU-Driven Asynchronous I/O Pipelining for Eliminating Storage-Compute Bottlenecks in Billion-Scale Similarity Search. In *Proceedings of the 2026 ACM SIGMOD International Conference on Management of Data*, SIGMOD, 2026.

[58] Yuming Xu, Hengyu Liang, Jin Li, Shuotao Xu, Qi Chen, Qianxi Zhang, Cheng Li, Ziyue Yang, Fan Yang, Yuqing Yang, Peng Cheng, and Mao Yang. SPFresh: Incremental In-Place Update for Billion-Scale Vector Search. In *Proceedings of the 29th Symposium on Operating Systems Principles*, SOSP, 2023.

[59] Zhiqiang Xu, Dong Li, Weijie Zhao, Xing Shen, Tianbo Huang, Xiaoyun Li, and Ping Li. Agile and Accurate CTR Prediction Model Training for Massive-Scale Online Advertising Systems. In *Proceedings of the 2021 International Conference on Management of Data*, SIGMOD, 2021.

[60] Shuo Yang, Haocheng Xi, Yilong Zhao, Muyang Li, Xiaoze Fan, Jintao Zhang, Han Cai, Yujun Lin, Xiuyu Li, Kurt Keutzer, Song Han, Chenfeng Xu, and Ion Stoica. Flash-KMeans: Fast and Memory-Efficient Exact K-Means. In *arXiv*, 2026.

[61] Xinyang Yi, Ji Yang, Lichan Hong, Derek Zhiyuan Cheng, Lukasz Heldt, Aditee Ajit Kumthekar, Zhe Zhao, Li Wei, and Ed H. Chi. Sampling-Bias-Corrected Neural Modeling for Large Corpus Item Recommendations. In *Proceedings of the 13th ACM Conference on Recommender Systems*, RecSys, 2019.

[62] Donghan Yu, Chenguang Zhu, Yuwei Fang, Wenhao Yu, Shuohang Wang, Yichong Xu, Xiang Ren, Yiming Yang, and Michael Zeng. KG-FiD: Infusing Knowledge Graph in Fusion-in-Decoder for Open-Domain Question Answering. In *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, ACL, 2022.

[63] Qianxi Zhang, Shuotao Xu, Qi Chen, Guoxin Sui, Jiadong Xie, Zhizhen Cai, Yaoqi Chen, Yinxuan He, Yuqing Yang, Fan Yang, Mao Yang, and Lidong Zhou. VBASE: Unifying Online Vector Similarity Search and Relational Queries via Relaxed Monotonicity. In *Proceedings of the 17th USENIX Symposium on Operating Systems Design and Implementation*, OSDI, 2023.

[64] Zili Zhang, Chao Jin, Linpeng Tang, Xuanzhe Liu, and Xin Jin. Fast, Approximate Vector Queries on Very Large Unstructured Datasets. In *20th USENIX Symposium on Networked Systems Design and Implementation*, NSDI, 2023.

WiseCode: Breaking the Scalability Barriers of Wide-Stripe Vector Codes

Sijie Cai, Guangyan Zhang*, Xiao Niu
Tsinghua University

Abstract

Wide-stripe erasure codes, with tens to over a hundred data chunks per stripe, offer high reliability at low storage overhead. Existing wide-stripe designs that are based on scalar codes (e.g., LRCs in Google and Azure) reduce *repair traffic* but increase *storage overhead*. Although vector codes are theoretically optimal in both metrics, they face severe scalability barriers in wide-stripe deployments.

We present WiseCode, *the first practical and scalable wide-stripe vector-coding approach* that achieves both efficient repair and ultra-low storage overhead. WiseCode overcomes three key scalability barriers through innovations in coding structure, coefficient selection, and coding algorithms. It introduces a template-unfold structure design that avoids sub-packetization blowup, a repetition-minimized search strategy that reduces coefficient search cost, and a two-stage coding algorithm that enables efficient encoding and decoding.

Evaluations on Ceph with ∼100-wide stripes and 1.04–1.06 storage overhead show that WiseCode increases repair throughput by 1.41×–2.18× compared to Google's UCLRCs at equal storage overhead, and also delivers higher throughput even at 2% lower storage overhead. WiseCode retains this advantage when combined with advanced repair-scheduling methods, consistently outperforming UCLRCs.

1 Introduction

Storage systems rely on redundancy mechanisms to tolerate routine device failures [1–4]. As data volumes grow exponentially [5], reducing storage overhead while preserving data reliability becomes increasingly important. In large-scale clusters managing exabytes of data, even a 1% storage overhead reduction saves millions of dollars in capital, operational, and energy costs [6]. This demand has driven the shift from replication to erasure codes [7–16] and now motivates *wide-stripe erasure codes*, or "wide codes" [6, 17, 18]. An (n,k) erasure code divides data into k data chunks and encodes them into $m = n-k$ parity chunks, forming a stripe with storage overhead of n/k (typically ≤ 1.5). Wide codes further reduce overhead by keeping m small while enlarging k. For example, moving from $(11,8)$ to $(106,100)$ reduces storage overhead from 1.375 to 1.06, without sacrificing reliability (§2.1).

Since data recovery is performed online, wide codes should quickly restore lost data [19–21] and minimize interference with foreground workloads [22–24]. Network and I/O resources are often the primary bottlenecks during recovery [24, 25], so reducing *repair traffic*—the amount of data accessed and transferred during recovery—is essential.

Achieving both *low repair traffic* and *ultra-low storage overhead* remains challenging in wide-code designs. Erasure codes fall into two categories: scalar and vector. Scalar codes, such as RS codes [18, 26] and Locally Recoverable Codes (LRCs) [6, 17, 27–30], operate at chunk granularity. RS codes satisfy the *Maximum Distance Separable (MDS)* property, achieving the optimal storage efficiency for a given reliability, but require reading k chunks to repair one; LRCs reduce repair traffic using *local parity* but increase storage overhead. Thus, **scalar codes cannot simultaneously optimize repair traffic and storage overhead.**

Vector codes [31–38], including Clay [34] and RS-ET [36], divide each chunk into α sub-chunks and organize them into α sub-stripes, where α is referred to as the *sub-packetization level*. Each sub-stripe defines a *coding equation* that specifies the relationship among its sub-chunks, allowing it to decode multiple sub-chunks independently. This enables repairing a lost chunk by decoding a small subset of sub-stripes, substantially reducing repair traffic while preserving the MDS property. As shown in Figure 1, implementing a vector code involves three key steps: (1) designing the *coding structure* that specifies sub-chunk variables in each sub-stripe's equation (with unknown coefficients), (2) selecting coefficients that ensure decodability under any 1 to m-chunk failures, and (3) developing efficient encoding/decoding algorithms.

In wide-stripe deployments (e.g., $n \approx 100$), vector codes face three **scalability barriers**:

1. *Sub-packetization blowup.* $(104, 100, \alpha)$ Clay codes require $\alpha = 4^{26}$ to achieve low repair traffic, making them

*Corresponding author: gyzh@tsinghua.edu.cn

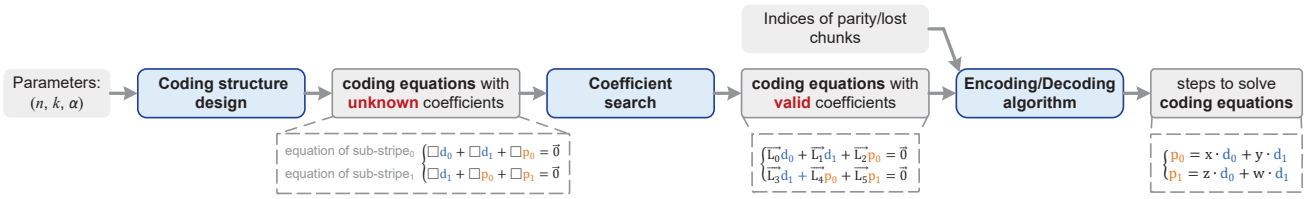


Figure 1: Three steps to realize vector codes—from coding structure design to coefficient search and coding algorithm.

impractical for deployments. RS-ET codes use smaller α but incur over 50% higher repair traffic than Clay codes.

2. *Prohibitive coefficient search.* For $(106, 100, 216)$ RS-ET codes, verifying one coefficient combination takes over 1,000 hours to check 1.7B failure cases. Moreover, many combinations must be tested before finding a valid one.

3. *High computational overhead.* Standard generator-matrix-based encoding achieves only 97 MB/s throughput for $(106, 100, 216)$ vector codes on our testbed, versus several GB/s for $(106, 100)$ scalar codes.

We propose WiseCode, *the first practical and scalable wide-stripe vector-coding approach* that achieves both low repair traffic and ultra-low storage overhead. WiseCode breaks the three scalability barriers via innovations in coding structure design, coefficient selection, and coding algorithms.

Template-unfold structure design. WiseCode takes a narrow MSR (Minimum Storage Regenerating [39]) stripe as a *template* and repeatedly instantiates it to form a wide-stripe structure while preserving the template's α and m. This allows each lost chunk to be recovered by reading sub-chunks from only a subset of sub-stripes. WiseCode selects the widest MSR template under a device-dependent α limit (higher on SSDs than HDDs for the same chunk size) to simultaneously avoid excessive data fragmentation and minimize instantiations that inflate repair traffic. This design achieves near-optimal repair traffic and minimum storage overhead without incurring sub-packetization blowup.

Repetition-minimized coefficient search. WiseCode reduces redundant work in coefficient search using two complementary techniques. *Divide-and-conquer verification* partitions chunks into disjoint groups and verifies decodability only for intra-group failure cases (i.e., all lost chunks lie within the same group), greatly reducing the verification scope while guaranteeing decodability for all failure cases. *Neighborhood-prioritized retry* generates a new candidate for retry by modifying only a small subset of coefficients in a rejected combination, preserving the decodability of most previously-verified failure cases. This significantly reduces per-candidate verification cost and accelerates retries, enabling n up to several hundred with $4 \le m \le 8$ over GF(2^{16}), whereas existing vector codes typically limit n to a few tens.

Two-stage encoding. Motivated by the observation that the generator matrix is much denser (up to 21×) than the sparse coefficients in the coding equations, WiseCode employs a *two-stage encoding* design. The *data aggregation* stage multiplies data by the sparse coefficient matrix to produce an intermediate vector; the *parity solving* stage then computes parity from the intermediate vector by solving the coding equations. WiseCode further optimizes both stages by extracting common operations during data aggregation and decomposing parity solving into computation-efficient steps using a divide-and-conquer strategy. This reduces multiplications from $O(km\alpha^2)$ to $km\alpha + O(m^3\alpha)$, cutting up to 94.6% of multiplications and improving throughput by 5.5×–22.4× in our experiments.

We integrated WiseCode into Ceph [8] as an erasure-code plugin. Evaluations on Ceph with stripe widths near 100 and storage overheads of 1.04–1.06 show that WiseCode delivers a strictly better tradeoff between repair performance and storage overhead than Google's UCLRCs [6], the state-of-the-art wide codes. At equal storage overhead, WiseCode increases repair throughput by 1.41×–2.18× compared to UCLRC; even with 2% lower storage overhead, it reduces repair traffic by 14.4% and improves repair throughput by 6.8%. When combined with the RepairBoost [21] repair-scheduling framework, WiseCode also increases repair throughput by 1.56×–2.38× compared to UCLRC, demonstrating consistent advantages under advanced scheduling.

2 Background and Challenges

2.1 Background

In an (n, k) erasure code, user data are divided into k equal-sized data chunks and encoded into $m = n - k$ parity chunks, forming an n-chunk *stripe* with storage overhead n/k. These n chunks are placed across independent failure domains (e.g., disks, hosts, racks) to tolerate multiple failures. A stripe with m parity chunks satisfies the MDS property [40] if any m lost chunks can be recovered. MDS codes have the minimum storage overhead for a given fault tolerance requirement.

Wide codes. Wide codes increase n while keeping m small (e.g., $n \ge 80$ and $m \le 8$ [6]) to reduce storage overhead without sacrificing reliability. For example, expanding RS codes

from $(11,8)$ to $(106,100)$ reduces storage overhead from 1.375 to 1.06, while increasing MTTDL (mean time to data loss, analyzed via a Markov model; see Appendix A.1) from 3.0×10^{11} to 5.4×10^{16} years. In large-scale storage clusters, degraded stripes are reported to be dominated by single-chunk failures (99.2% at Google [6], 98.08% at Facebook [22]). Moreover, modeling with a Binomial distribution [41, 42]—assuming an independent failure rate of 1/(4 years) and a repair rate of 1/(30 minutes) [17, 27, 28]—demonstrates that a stripe with $n = 100$ resides in a single-chunk failure state for over 99% of its total degraded duration. Therefore, existing wide-code designs (e.g., LRCs) primarily optimize single-chunk repair [6, 17].

Scalar and vector codes. Scalar codes treat each of the k data chunks as one variable and generate m coded variables, each corresponding to a parity chunk. Vector codes divide each chunk into α equal-sized sub-chunks and treat them as independent variables, increasing the number of variables in the encoding operation by a factor of α.

Limitations of wide-stripe scalar codes. Wide-stripe scalar codes (e.g., LRCs) reduce repair traffic by adding *local parities* on top of global parities, but this increases storage overhead. Consequently, they cannot achieve efficient repair and ultra-low storage overhead simultaneously.

2.2 Scalability Barriers of Vector Codes

In wide-stripe deployments, vector codes face three scalability barriers.

Sub-packetization blowup for minimal repair traffic. Minimizing repair traffic in vector codes comes at the cost of rapidly increasing α. MSR codes achieve optimal repair traffic [39] with $\alpha \geq m^{\lceil \frac{n}{m} \rceil}$ [43], causing severe and often impractical data fragmentation in wide stripes. For example, $(104,100,\alpha)$ Clay codes [34] require $\alpha = 4^{26}$. In practice, storage systems must cap α to limit I/O overhead from data fragmentation, which inevitably increases repair traffic. For example, when $\alpha = 64$, which corresponds to 64 MB chunks and 1 MB sub-chunks, RS-ET [36] and Hashtag+ [37] codes incur over 50% more repair traffic than the optimal bound. It is therefore challenging to design a coding structure that reduces repair traffic without incurring sub-packetization blowup.

Prohibitive search cost for valid coefficients. Two factors make coefficient search intractable: (1) *Expensive per-candidate verification*. A valid coefficient combination must ensure that every $m\alpha \times m\alpha$ coefficient matrix corresponding to any m-chunk failure case is invertible. In a $(106,100,216)$ RS-ET stripe, this requires checking 1.7 billion 1296×1296 matrices, each taking 2.4 ms on our testbed, totaling 1,130 hours. (2) *Frequent rejects*. Any failed invertibility check discards a candidate, and wide stripes have many independent failure cases, causing extremely high reject rates. With coefficients randomly selected from GF(2^{16}), $(76,72,\alpha)$ or wider RS-ET stripes fail to find any valid combination within 10 million trials in our experiments. The candidate space grows exponentially with the number of coefficients, making exhaustive enumeration infeasible.

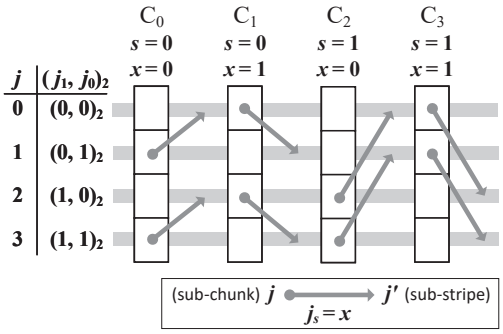


Figure 2: Coding structure of a $(4,2)$ MSR code with $\alpha = 4$. Columns correspond to chunks, each containing 4 sub-chunks. Rows correspond to sub-stripes, each annotated by index j. Rows depict Rule #1 and arrows depict Rule #2.

Excessive encoding cost using the standard method. Standard encoding, multiplying data by a generator matrix, works well for scalar codes [44, 45] but is too costly for vector codes. In the $(106,100,216)$ vector code (§5.1), each column of the generator matrix has up to 126 nonzero entries (vs. 6 in a $(106,100)$ scalar code), dramatically increasing the number of multiplications and lowering encoding throughput to 97 MB/s on our testbed (vs. several GB/s for the scalar code). This motivates developing encoding algorithms that restore computational efficiency.

2.3 MSR Coding Structure and Repair

We introduce the state-of-the-art MSR codes with the lowest sub-packetization levels $\alpha = m^{\lceil \frac{n}{m} \rceil}$ [34, 43, 46], which form the basis of WiseCode.

Coding structure. Each chunk (data or parity, determined by storage systems) is divided into α sub-chunks, and the stripe is organized into α sub-stripes. The sub-chunk to sub-stripe assignment follows two coding rules:

Rule #1. *Sub-Chunk$_j$ of each chunk belongs to Sub-Stripe$_j$ ($0 \leq j < \alpha$).*

Rule #2. *Sub-Chunk$_j$ of Chunk$_i$ also belongs to Sub-Stripe$_{j'}$ when $j' \neq j$. To compute j', express j in the base-m representation as $j = (\ldots, j_2, j_1, j_0)_m$. Let $s = \lfloor \frac{i}{m} \rfloor$, and $x = i \bmod m$. Then replace j_s with x to obtain $j' = (\ldots, j_{s+1}, x, j_{s-1}, \ldots)_m$.*

In Figure 2 for a $(4,2)$ MSR code, rows depict Rule #1 and arrows depict Rule #2. Take Sub-Chunk$_1$ of Chunk$_0$ as an example. Rule #1 assigns this sub-chunk to Sub-Stripe$_1$. Rule #2 sets $i = 0$, $j = (0,1)_2$, $s = 0$, and $x = 0$. This gives $j' = (j_1, x)_2 = (0,0)_2 \neq j$, so the sub-chunk also belongs to Sub-Stripe$_0$.

Coding equations and coefficients. Each sub-stripe defines a linear relation among its sub-chunks, expressed as a

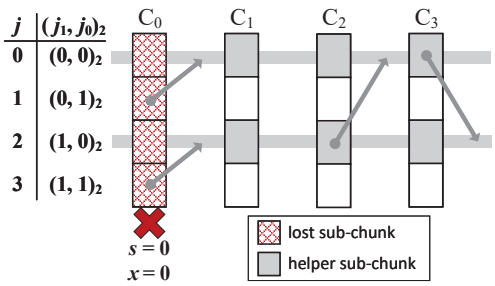


Figure 3: Repair pattern of Chunk_0 in a $(4,2)$ MSR code. Only Sub-Stripe_0 and Sub-Stripe_2 are decoded during repair.

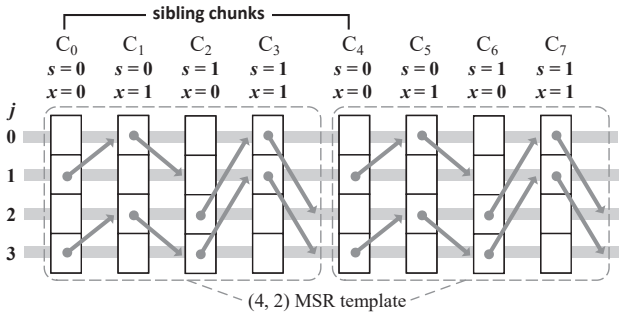


Figure 4: Constructing an $(8,6,4)$ WiseCode stripe using a $(4,2)$ MSR code as the template.

coding equation: $\vec{L}_0 c_0 + \vec{L}_1 c_1 + \cdots = \vec{0}$, where c_i represents a sub-chunk, and $\vec{L}_i$ is an m-element *coefficient vector* over a finite field (e.g., GF(2^8) [34]).

Single-failure repair pattern. Each sub-stripe can recover up to m lost sub-chunks. A lost chunk i is recovered by decoding α/m sub-stripes whose indices j satisfy $j_s = x$, where s and x are defined in Rule #2. Each such sub-stripe contains m lost sub-chunks. Each of the $n-1$ healthy chunks contributes one helper sub-chunk to each selected sub-stripe, incurring the total repair traffic (normalized by α) of $(n-1) \times \frac{\alpha}{m} \times \frac{1}{\alpha} = \frac{n-1}{m}$, which is optimal [39].

Figure 3 shows the repair pattern for Chunk_0 in the $(4,2)$ MSR code. With $s = 0$, and $x = 0$, Sub-Stripe_0 and Sub-Stripe_2 are the two sub-stripes whose indices satisfy $j_s = x$. Sub-Stripe_0 recovers Sub-Chunk_0 and Sub-Chunk_1 of Chunk_0, while Sub-Stripe_2 recovers the remaining two lost sub-chunks. In total, 6 healthy sub-chunks across these two sub-stripes are accessed as helper sub-chunks.

3 Template-Unfold Structure Design

This section presents the template-unfold structure design (§3.1), explains how it achieves near-optimal repair traffic for single-chunk failures (§3.2), shows how it reduces multi-chunk repair traffic (§3.3), and presents the dedicated chunk placement strategy (§3.4).

3.1 Code Construction via Template Unfolding

An (n,k,α) WiseCode constructs the coding structure in two steps: (1) It first selects a (n_{msr}, k_{msr}) MSR code ($n_{msr} < n$) as a *template*, which has the same parity count m and sub-packetization level α with the target coding structure; (2) It then *unfolds the template* by repeatedly instantiating it into multiple *instances* and concatenating sub-stripes with the same index across instances to scale the stripe width, while preserving m and α of the template.

Figure 4 illustrates the coding structure of an $(8,6,4)$ WiseCode where $m = 2$ and $\alpha = 4$, using a $(4,2)$ MSR code as the template. The template is instantiated twice, once for $\text{Chunk}_{0\sim3}$ and once for $\text{Chunk}_{4\sim7}$, with identical sub-chunk to sub-stripe assignments. We refer to chunks from different instances that occupy the same position in the template as *sibling chunks* (e.g., Chunk_0 and Chunk_4). Each chunks inherits the parameters s and x specified by Rule #2 (§2.3) from the template, and sibling chunks share identical s and x.

To generate the n chunks, the (n_{msr}, k_{msr}) template is instantiated $\lceil \frac{n}{n_{msr}} \rceil$ times—so each chunk has $\lceil \frac{n}{n_{msr}} \rceil - 1$ sibling chunks—producing a total of $\lceil \frac{n}{n_{msr}} \rceil \cdot n_{msr}$ chunks. WiseCode supports flexible n by retaining only the first n chunks and treating any remaining chunks as logical zero chunks, which are omitted from physical storage.

While n and k can take arbitrary integer values, α is discrete because the underlying (n_{msr}, k_{msr}) MSR code only supports α in the form of $\alpha = m^{\lceil \frac{n_{msr}}{m} \rceil}$. In practice, WiseCode imposes an upper bound on α by dividing the chunk size (e.g., 64 MB$\sim$1 GB [9, 11]) by an appropriate sub-chunk size (e.g., 1$\sim$4 MB on HDDs and 128 KB on SSDs [47]). WiseCode then selects the MSR template with the largest n_{msr} among those whose α remains within the bound, and sets $k_{msr} = n_{msr} - m$. This design mitigates the I/O overhead from data fragmentation and minimizes instantiations that inflate repair traffic (§3.2).

At this stage, (n,k,α) WiseCode has specified the sub-chunks in each of the α sub-stripes. Each sub-stripe defines a coding equation, where sub-chunks in this sub-stripe appear as variables and the coefficients are specified in Section 4.

3.2 Single-Chunk Failure Repair

An (n,k,α) WiseCode repairs a lost chunk by decoding α/m sub-stripes, each recovering m lost sub-chunks. Their indices j satisfy $j_s = x$, where s and x are the parameters of the lost chunk, inherited from the template (§3.1).

Figure 5 illustrates the repair pattern of Chunk_0 in the $(8,6,4)$ WiseCode. Both Sub-Stripe_0 and Sub-Stripe_2 contain two lost sub-chunks, and decoding them suffices to reconstruct all lost sub-chunks. Each healthy chunk, except Chunk_4, contributes only two helper sub-chunks, while Chunk_4 must

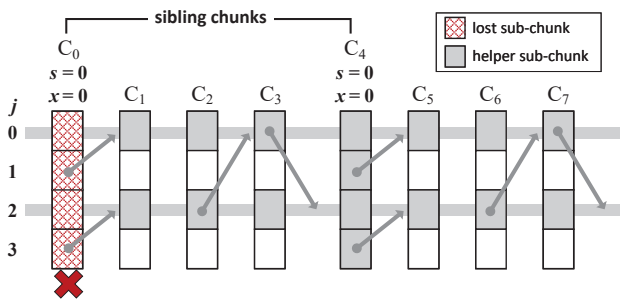


Figure 5: Repair pattern of Chunk_0 in the $(8,6,4)$ WiseCode. Only Sub-Stripe_0 and Sub-Stripe_2 are decoded during repair.

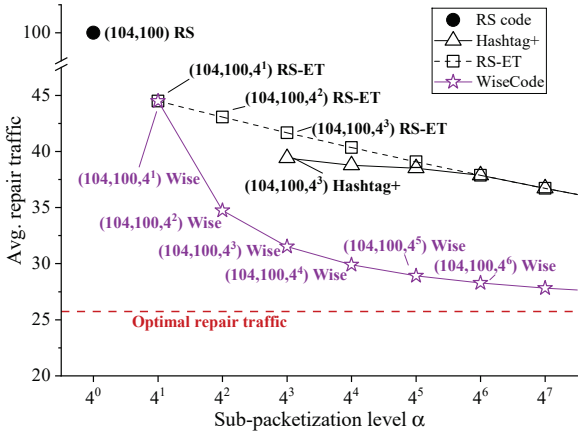


Figure 6: Trade-off between α and repair traffic for $(104,100,\alpha)$ WiseCode, HashTag+, and RS-ET codes.

contribute all 4 sub-chunks as helper sub-chunks. This is because Chunk_4 is a sibling chunk of Chunk_0, which means sub-stripes used to recover Chunk_0 also span the entire Chunk_4.

Generally, the repair pattern of Chunk_i in (n,k,α) WiseCode consists of two parts (proof in Appendix A.4):

❶ Non-sibling part: each healthy chunk that is not a sibling chunk of Chunk_i contributes 1 helper sub-chunk per sub-stripe, totally α/m helper sub-chunks whose indices i satisfy $i_s = x$, where s and x are the parameters of Chunk_i.

❷ Sibling part: each healthy chunk that is a sibling chunk of Chunk_i contributes m helper sub-chunks per sub-stripe. In total, all α sub-chunks serve as helper sub-chunks. □

Traffic analysis. Assuming Chunk_i has z sibling chunks, the repair traffic (normalized by α) is calculated as:

$$\left((n-1-z)\frac{\alpha}{m}+z\alpha\right)\cdot\frac{1}{\alpha}=\frac{n-1}{m}+\frac{m-1}{m}z$$

where the first term $\frac{n-1}{m}$ matches the optimal repair traffic [39], while the second term $\frac{m-1}{m}z$ reflects the overhead introduced by template unfolding.

Since each chunk has $z=\lceil\frac{n}{n_{msr}}\rceil-1$ sibling chunks (§3.1), increasing n_{msr} decreases z and thus reduces repair traffic. However, a larger n_{msr} also increases $\alpha=m^{\lceil\frac{n_{msr}}{m}\rceil}$, exacerbating data fragmentation. To balance this trade-off, WiseCode maximizes n_{msr} under a given upper bound of α (§3.1), thereby *achieving near-optimal repair traffic while mitigating data fragmentation*.

Figure 6 illustrates the trade-off between sub-packetization level (α) and repair traffic in $(104,100,\alpha)$ WiseCode under varied α. With a 64 MB chunk size and 1 MB sub-chunk size, $(104,100,64)$ WiseCode incurs only 22.3% higher repair traffic than the optimal, while with a 512 MB chunk size and 128 KB sub-chunk size, $(104,100,4^6)$ WiseCode incurs only 9.9% higher repair traffic.

Compared with two representative vector codes, HashTag+ [37] and RS-ET [36] codes, WiseCode converges to the optimal repair traffic more rapidly as α increases, achieving 19.3-25.3% lower repair traffic when $4^2\le\alpha\le4^7$ (noting that HashTag+ codes do not support $\alpha=m$ or $\alpha=m^2$). This advantage stems from coding design differences: HashTag+ requires decoding more sub-stripes, while RS-ET accesses more helper sub-chunks per sub-stripe, both of which increase repair traffic.

3.3 Multi-Chunk Failure Repair

Although multiple-chunk failures are rare, there is a growing concern that their likelihood increases in wide stripes in a certain degree [6]. When multiple-chunk failure occurs, WiseCode selects the *minimum subset of sub-stripes* to recover all lost sub-chunks. It uses a greedy strategy, iteratively selecting sub-stripes that maximize *recovery capability*—the number of lost sub-chunks they can recover—until all lost sub-chunks are recoverable.

In each iteration, WiseCode first calculates the recovery capability of the currently selected sub-stripes using a flow model. In this model, each sub-stripe sends up to m flows to the its lost sub-chunks, and any lost sub-chunks that receives at least one flow is considered recoverable. Thus, computing the recovery capability reduces to a max-flow problem. After obtaining the current recovery capability, WiseCode enumerate unselected sub-stripes and adds the sub-stripe that yields the largest increase in recovery capability.

In this way, only a subset of sub-stripes are selected for recovery, thereby significantly reducing the number of helper sub-chunks. For example, in the case of double-chunk failures, $(104,100,64)$ WiseCode incurs an average repair traffic of 66.68, compared to 100 in $(104,100)$ RS codes.

3.4 Fault-Tolerance and Placement Strategy

WiseCode designs its chunk placement policy in two steps: first, it enforces a hard durability constraint to guarantee data persistence against correlated failures; subject to this con-

straint, it then optimizes chunk co-location to minimize cross-rack network traffic.

Uniform fault-tolerance guarantee. Unlike LRCs—where local parities provide less protection than global ones and certain failure patterns involving no more than m chunks remain unrecoverable—WiseCode is designed to achieve the maximum fault-tolerance level, i.e., the MDS property. This property ensures WiseCode can recover from the loss of any m chunks (see §4), thereby provides a uniform fault-tolerance guarantee. This uniformity significantly simplifies chunk placement against correlated failures. For instance, to protect against rack-level correlated failures, a placement policy only needs to ensure that no more than m chunks of the same stripe are placed within a single rack, regardless of their specific chunk indices.

Rack-aware placement. Subject to the constraint that no more than m chunks reside within a single rack, WiseCode leverages its template-unfold design to minimize cross-rack traffic in hierarchical network topologies, where cross-rack bandwidth is typically much more constrained than intra-rack bandwidth [48]. Specifically, when repairing a lost chunk, sibling chunks contribute their entire data, while non-sibling chunks contribute only partial data. WiseCode exploits this asymmetry by grouping sibling chunks within the same rack whenever possible, thereby maximizing intra-rack data exchange and minimizing cross-rack transfers. For example, compared to random placement, rack-aware placement can save 18.9% cross-rack repair traffic for $(106,100,216)$ WiseCode.

4 Repetition-Minimized Coefficient Search

This section presents the repetition-minimized coefficient search, which combines two techniques: divide-and-conquer verification to reduce per-candidate verification cost (§4.1), and neighborhood-prioritized retry to minimize both the overhead of each retry and the number of retries (§4.2).

4.1 Divide-and-Conquer Verification

Given a candidate coefficient combination, WiseCode verifies the decodability of all failure cases in a *divide-and-conquer* manner. It first partitions chunks into disjoint groups and classifies all failure cases into two categories: *intra-group* failure cases where missing chunks lie within the same group, and *inter-group* failure cases where missing chunks span multiple groups. It then verifies decodability only for the intra-group ones, while safely skipping verification for the inter-group ones. This strategy is enabled by the coefficient design and the chunk-partitioning rule, as detailed below.

Coefficient design. In an (n,k,α) WiseCode, coefficients are generated from mn distinct parameters, denoted λ_0 to λ_{mn-1}. Each Chunk$_i$ is associated with m parameters, specifically λ_{im} to $\lambda_{(i+1)m-1}$. For Sub-Chunk$_j$ of Chunk$_i$, WiseCode

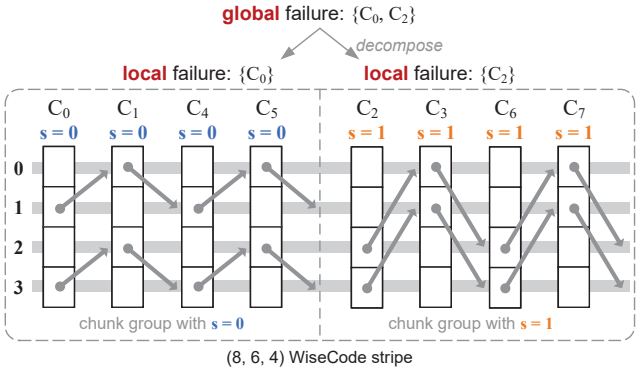


Figure 7: The chunk-partitioning rule groups chunks by s, decomposing a global failure into multiple local failures.

selects the parameter $\lambda = \lambda_{im+j_s}$, where s and j_s are defined in Rule #2 (§2.3). The coefficient vector is then constructed as $\vec{\lambda} = [\lambda^0, \lambda^1, \cdots, \lambda^{m-1}]^T$ and applied to all variables corresponding to this sub-chunk in the coding equations.

Chunk-partitioning rule. WiseCode partitions chunks by their parameters s, forming groups where all chunks share the same s value. Since $s = \lfloor i/m \rfloor$ for $0 \le i < n_{msr}$ (§2.3) takes $\lceil n_{msr}/m \rceil$ distinct values, the chunks are thus partitioned into $g = \lceil n_{msr}/m \rceil$ groups. As shown in Figure 7, in an $(8,6,4)$ WiseCode where $n_{msr} = 4$ and $m = 2$, chunks are partitioned into $g = 2$ groups corresponding to $s = 0$ and $s = 1$.

Below, we explain the correctness of divide-and-conquer verification. An inter-group failure case, referred to as a *global failure case*, can be decomposed into multiple *local failure cases*, each confined to a single group. As shown in Figure 7, in an $(8,6,4)$ WiseCode stripe, a global failure case {Chunk$_0$, Chunk$_2$} is decomposed into two local failure cases, {Chunk$_0$} and {Chunk$_2$}. We prove that global failure cases can be safely omitted from verification through Theorem 4.1 and Theorem 4.2.

Theorem 4.1. *A global failure case is decodable if and only if all corresponding local failure cases are decodable.*

Proof sketch. The proof follows a similar approach to prior theoretical work on MSR codes [49]. The key idea is to decompose the coefficient matrix of a global failure case into multiple sub-matrices, each corresponding to a local failure case, such that the overall matrix is invertible if and only if all sub-matrices are invertible (Appendix A.6). □

Theorem 4.2. *Every local failure is a subset of some m-chunk intra-group failure. If this m-chunk failure case is decodable, the local failure case is also decodable.*

Proof. Every local failure involves fewer than m lost chunks, and by adding chunk(s) from the same group we obtain the corresponding m-chunk intra-group failure. If that m-chunk failure is decodable, then any subset of its lost chunks—including the current local failure—is also decodable. □

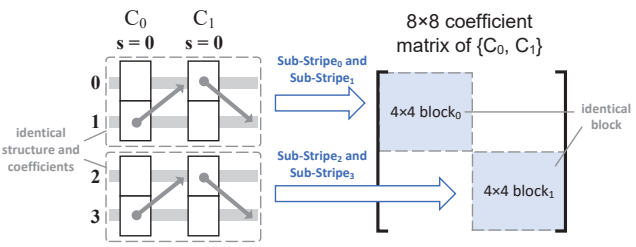


Figure 8: Coefficient matrices for chunks within a group exhibit a matrix-block-repetition pattern.

Consequently, WiseCode only needs to verify all m-chunk intra-group failure cases, totaling $g \times \binom{n/g}{m}$ cases—a small fraction of all $\binom{n}{m}$ possible failure cases. For example, in a $(106, 100, 216)$ WiseCode where $g = 3$, only 5.2 million failure cases ($\sim$0.3% of all failure cases) require verification.

Matrix size reduction. WiseCode also reduces the size of matrices to be verified according to a useful structural property under chunk partitioning: the coefficient matrix associated with chunks within a single group exhibits a *matrix-block-repetition pattern*. Specifically, it contains α/m identical diagonal matrix blocks, each corresponding to a set of m sub-stripes, with all sub-stripe sets sharing identical coding structures and coefficient assignments (Appendix A.5). As shown in Figure 8, the coefficient matrix of $\{\text{Chunk}_0, \text{Chunk}_1\}$ in the $s = 0$ group consists of two identical matrix blocks, one for Sub-Stripe$_0$ and Sub-Stripe$_1$, and another for the remaining sub-stripes. Therefore, WiseCode verifies invertibility by checking only a single matrix block rather than the entire matrix, reducing the computation cost to m/α of the original.

Overall, WiseCode significantly accelerates verification by reducing both the number of failure cases and the size of matrices to be verified. For example, in a $(106, 100, 216)$ stripe, WiseCode requires verifying only 5.2 million matrices of size 36×36, which can be completed within 3 minutes, achieving a speed-up of over $20,000\times$ compared to 1130 hours required by RS-ET code.

4.2 Neighborhood-Prioritized Retry

If a candidate coefficient combination fails verification, the search continues exploring new candidates until a valid one is found. To reduce retry overhead and accelerate convergence to valid solutions, WiseCode employs a *neighborhood-prioritized retry* strategy. Instead of resampling all coefficients, it updates only a small subset of coefficients while retaining the verified ones, generating a *neighborhood* of the rejected candidate. This localized adjustment preserves the decodability of most previously-verified failure cases, thereby avoiding redundant verification work.

To track which coefficients are retained, WiseCode treats each chunk's coefficients as a chunk-level *entry*, managed

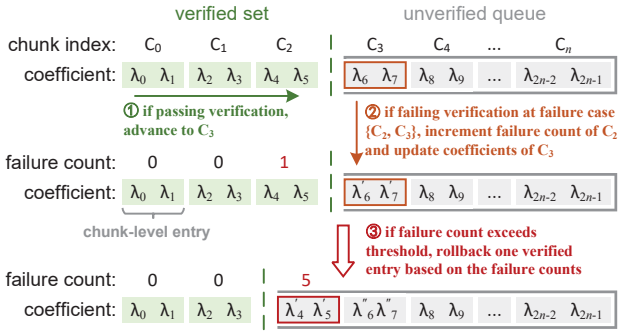


Figure 9: Three actions in each neighborhood-prioritized retry step: (1) advance, (2) update, and (3) rollback.

through a *verified set* and an *unverified queue*. A chunk is considered (un)verified if its coefficients are (un)verified. The unverified queue, initially containing all entries ordered by chunk index, is manipulated through *advance* and *rollback* operations described later. In each step, WiseCode updates only the coefficients of the entry at the front of the unverified queue, marking its corresponding chunk as the current one.

To determine which failure cases require verification, WiseCode groups chunks using the chunk-partitioning rule (§4.1) and identifies the group containing the current chunk. It then verifies only failure cases that involve the current chunk and any $m-1$ verified chunks within the same group. Failure cases involving only m verified chunks require no repeated verification since their coefficients remain unchanged. WiseCode also maintains a *failure count* for each verified entry, reflecting its contribution to failed verifications and guiding rollback decisions.

As shown in Figure 9, each search step operates on entries parameterized by λ (§4.1) and performs one of three actions:

1. **Advance**. If all required failure cases pass verification, WiseCode moves the current entry to the verified set and proceeds to the next unverified entry (or terminates if none remain);

2. **Update**. If verification fails, WiseCode increments the failure counts of involved verified entries and randomly re-samples the λ values of the current entry for retry;

3. **Rollback**. If any failure count exceeds a threshold, WiseCode randomly selects a verified entry (weighted by its failure count), updates its coefficients, moves it to the front of the unverified queue, and resets all failure counts.

Search overhead. WiseCode reduces both per-candidate verification cost and retry count, since failure cases composed entirely of verified chunks require no verification and never trigger retry. We conduct repeated experiments to compare the search overhead of the neighborhood-prioritized strategy with a naive baseline, *exhaustive enumeration*, which re-samples λ values for all entries and retry the verification of all failure cases whenever a candidate is rejected.

Table 1: Comparison of exhaustive enumeration (Baseline) and neighborhood-prioritized retry (Ours) in pre-candidate verification time, retry count, and total search time.

Stripe configuration	Verification time		# Retry		Total search time	
	Baseline	Ours	Baseline	Ours	Baseline	Ours
(104, 100, 16)	36.0 ms	2.49 ms	279	117	5.93 s	3.78 s
(104, 100, 64)	22.3 ms	0.78 ms	42	34	1.10 s	1.14 s
(105, 100, 25)	861 ms	66.0 ms	> 1.6e6	2047	> 24 h	131 s
(105, 100, 125)	819 ms	16.5 ms	> 1.6e6	312	> 24 h	21.0 s
(106, 100, 216)	2113 ms	163 ms	> 6.3e5	2403	> 24 h	341 s

Table 2: Maximum stripe width achievable from coefficient search over GF(2^{16}), for given m and α.

Code configuration		m=4	m=5	m=6	m=7	m=8
WiseCode	$\alpha = m$	232	80	48	35	27
	$\alpha = m^2$	464	160	96	70	54
	$\alpha = m^3$	696	240	144	105	81
HashTag+	$\alpha = m^3$	87	39	25	19	17
RS-ET	$\alpha = m$	72	46	30	28	24
	$\alpha = m^2$	65	40	34	25	26
	$\alpha = m^3$	68	45	31	20	15

Both strategies use 20 threads that verify different candidates in parallel. The neighborhood-prioritized retry synchronizes the candidate across threads whenever any thread advances or the failure count (aggregated across threads) of any verified entry exceeds the rollback threshold, which is set to 1000. All coefficients and λ parameters are defined over the finite field GF(2^{16})[1]. We measure the per-candidate verification time, retry count, and the total search time, averaged over 10 independent runs (Table 1). For $(104, 100, 16)$ and $(104, 100, 64)$ stripes, our method accelerates per-candidate verification by up to 28.6× and reduces retry count by up to 58%, while the total search time remains comparable due to additional synchronization overhead. For stripes with larger m—$(105, 100, 25)$, $(105, 100, 125)$, and $(106, 100, 216)$—our method finishes the search within 6 minutes, whereas the baseline fails to find valid solutions within 24 hours due to excessive retries.

Scalability in stripe width. We evaluate WiseCode's scalability by targeting a large stripe width of 1000 and measuring the maximum number of verified chunks—denoted as n^*—achieved with 20 threads within a 24-hour search period. At this point, the first n^* chunks have been verified, enabling any stripe of width n' ($n' \leq n^*$) to be constructed by the first n' chunks. For comparison, HashTag+ and RS-ET codes adopts a similar neighborhood-prioritized retry strategy, but without divide-and-conquer verification.

As shown in Table 2, WiseCode enables stripe widths up to several hundred with $4 \leq m \leq 8$. The achievable stripe width increases with α, because a higher α enlarges both n_{msr} and the group count $g = \lceil n_{msr}/m \rceil$, reducing intra-group failures and thereby accelerating the search. In contrast, HashTag+ and RS-ET are restricted to stripe widths of only a few tens, since the absence of divide-and-conquer verification forces them to check too many failure cases, significantly impeding the search process.

Existence of valid coefficients. Prior studies [35, 36] address the finite field size required to guarantee the existence of valid coefficients. For example, to ensure a valid coefficient assignment, a $(70, 66, 4)$ RS-ET code requires a field size $\geq 2(\binom{n-1}{k-1} - \binom{\lceil n/\alpha \rceil - 1}{\lceil k/\alpha \rceil - 1})$, which equals 1.7e6. However, as shown in Table 2, $(70, 66, 4)$ RS-ET code is successfully supported by GF(2^{16}) with a field size of 65536, suggesting that existing theoretical bounds are overly conservative. While determining the maximum stripe width supported by a given finite field remains an open problem, the results in Table 2 strongly suggest that the theoretical ceiling for WiseCode's scalability is substantially higher; we provide further empirical analysis in the Appendix A.2 to validate this inference.

5 Two-Stage Encoding Framework

This section presents the two-stage encoding framework that requires much fewer multiplications than standard generator matrix encoding (§5.1) and explains how it extracts common operations during *data aggregation* (§5.2) and decomposes *parity solving* into computation-efficient steps, followed by extending these optimizations to the decoding process (§5.3).

5.1 Design Overview

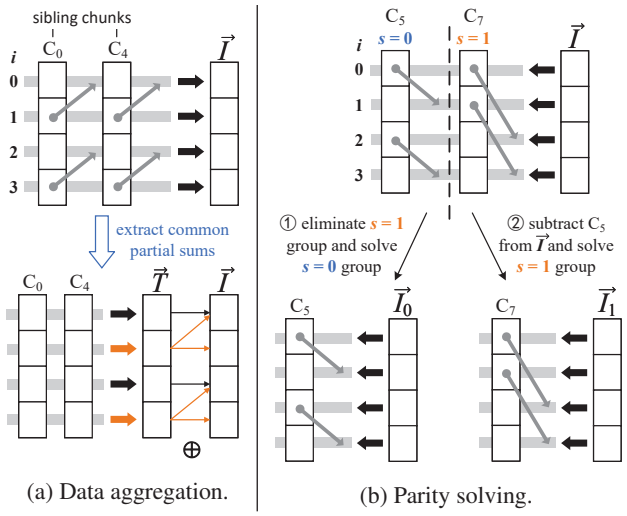
Once the parity chunks are designated, the coding equations can be expressed in matrix form as $C_D \times D + C_P \times P = \vec{0}$, where D and P denote the $k\alpha$ data sub-chunks and $m\alpha$ parity sub-chunks, and C_D and C_P are the corresponding $m\alpha \times k\alpha$ and $m\alpha \times m\alpha$ coefficient matrices, with C_P guaranteed invertible (§4). The standard encoding approach [45, 50] constructs the generator matrix $G = -{C_P}^{-1} \times C_D$ and computes $P = G \times D$.

However, the standard encoding approach is costly for WiseCode. In a $(106, 100, 216)$ WiseCode stripe, encoding with G requires 17–126 multiplications per data sub-chunk (depending on which chunks are designated as parity), while each coefficient vector in the coding equations contains only $m = 6$ elements. We make the following observation.

Observation 5.1. *The number of multiplications incurred by the nonzero entries in G far exceeds the number of coefficients in the coding equations.*

Motivated by this observation, WiseCode reduces multiplications—the dominant cost relative to additions—by multiplying each data sub-chunk only by its own coefficient vector and introducing a two-stage encoding framework:

[1]This choice balances coefficient design requirements and arithmetic complexity: GF(2^8) limits the stripe width to $256/m$ because coefficient design requires mn unique parameters (§4.1) but GF(2^8) provides only 256; GF(2^{32}) decreases multiplication throughput by 51.6% compared to GF(2^{16}).



(a) Data aggregation. (b) Parity solving.

Figure 10: Encoding scheduling: (a) extracting common data sums to eliminate redundant aggregation operations; (b) solving parity chunks using a divide-and-conquer strategy.

1. **Data aggregation**. Within each sub-stripe, data sub-chunks are multiplied by their m-element coefficient vectors and summed position-wise, producing an m-element aggregated-result vector. Concatenating the aggregated-result vectors from all α sub-stripes yields an $m\alpha$-element intermediate vector $\vec{I}$ that satisfies $\vec{I} = C_D \times D$.
2. **Parity solving**. After replacing $C_D \times D$ by $\vec{I}$, the parity sub-chunks P are then solved from the resulting coding equations $C_P \times P = -\vec{I}$.

This framework enables WiseCode to extract common operations across sub-stripes during data aggregation and follow a divide-and-conquer manner to process parity solving.

5.2 Extraction of Common Data Operations

During data aggregation, when a data sub-chunk contributes to multiple sub-stripes, it is multiplied by the same coefficient vector across those sub-stripes, as dictated by the coefficient selection (§4.1), yielding identical multiplication-result vectors. Moreover, sibling chunks follow the same coding structure, so sub-chunks with the same index across sibling chunks map to the same set of sub-stripes. Leveraging these properties, WiseCode extracts common partial sums from the multiplication-result vectors of sibling chunks, enabling their reuse across sub-stripes.

We illustrate the process of data aggregation using an $(8,6,4)$ WiseCode stripe (Figure 10a), where Chunk_0 and Chunk_4 are data chunks and form a pair of sibling chunks. Sub-chunks in each row are multiplied by their respective m-element coefficient vectors and summed to obtain an m-element common partial sum $\vec{T}$ for this row. The resulting $\vec{T}$ is then accumulated into the corresponding positions of $\vec{I}$ according to the sub-chunk to sub-stripe mapping. To minimize memory traffic, each element of $\vec{T}$ is added to $\vec{I}$ immediately after it is computed, avoiding temporary storage of $\vec{T}$ in memory. The remaining chunks are processed together with their sibling chunks (if any) in the same manner to complete the computation of $\vec{I}$.

Cost analysis. By extracting common partial sums, each of the $k\alpha$ data sub-chunks is multiplied by its m-element coefficient vector only once, leading to a total of $km\alpha$ multiplications. From the coding rules (§2.3), we observe that a fraction $(m-1)/m$ of all sub-chunks participate in two sub-stripes instead of one. Consequently, this method reduces the number of multiplications during data aggregation by $(m-1)/(2m-1)$.

5.3 Divide-and-Conquer Parity Solving

During parity solving, directly computing $P = -C_P^{-1} \times \vec{I}$ is prohibitively expensive because C_P^{-1} can be up to $10\times$ denser than C_P. Instead, WiseCode uses a divide-and-conquer strategy that operates on sub-matrices with sparse inverses. Parity chunks are partitioned into groups using the same chunk-partitioning rule as §4.1, which correspondingly partitions C_P into sub-matrices. As discussed in Section 4.1, each sub-matrix exhibits a matrix-block-repetition structure with nonzero entries confined to the matrix blocks, and its inverse preserves this sparsity[2]. WiseCode therefore performs matrix elimination and inversion at the sub-matrix granularity, using each sparse inverse to independently and efficiently solve the corresponding parity group.

Figure 10b illustrates this process for an $(8,6,4)$ WiseCode stripe. Chunk_5 and Chunk_7 are parity chunks and belong to groups with $s=0$ and $s=1$, respectively. To solve the $s=0$ group, WiseCode first eliminates the $s=1$ group via linear transformations (Appendix A.6), leaving only the $s=0$ sub-matrix and a transformed vector $\vec{I}_0$ (from $\vec{I}$). It then inverts this sub-matrix and multiplies it with $\vec{I}_0$ to obtain the $s=0$ group $\{\text{Chunk}_5\}$. After subtracting the contribution of Chunk_5 from $\vec{I}$ to produce $\vec{I}_1$, only the $s=1$ group remains unknowns. WiseCode applies the same procedure to solve the $s=1$ group $\{\text{Chunk}_7\}$.

In general, when parity chunks span multiple groups, WiseCode recursively eliminates groups until one remains, solves that group, and continues solving the remaining groups sequentially as the recursion unwinds.

Cost analysis. The divide-and-conquer parity solving incurs $O(m^3\alpha)$ multiplications (Appendix A.7), compared to $O(m^2\alpha^2)$ for directly computing $P = -C_P^{-1} \times \vec{I}$. Overall, the two-stage encoding process performs $km\alpha + O(m^3\alpha)$ multiplications, whereas the standard approach that directly computes $P = G \times D$ requires $O(km\alpha^2)$. WiseCode processes sub-chunks at a default 4 KB granularity (e.g., an 8 KB sub-chunk is processed as two 4 KB units), and therefore requires

[2]Since each matrix block appears independently and identically, we can invert them block-wise while preserving the repetition structure.

an additional $m\alpha\times$4KB of memory (within 5 MB in our experiments) to store the $m\alpha$-element intermediate vector $\vec{I}$ and other temporary results.

Extending to Decoding. During decoding, healthy chunks are treated as data chunks and aggregated using the same data-aggregation stage. Lost chunks are then partitioned into groups and recovered using the same divide-and-conquer parity-solving strategy.

6 Implementation

To demonstrate WiseCode's portability, we integrate it into Ceph [8, 51], a widely used open-source distributed storage system (§6.1). Because Ceph's native workflow is not adaptable to advanced repair scheduling, we also combine WiseCode with several representative scheduling methods to illustrate its compatibility with prior techniques (§6.2).

6.1 Ceph Erasure-Code Plugin

Although WiseCode introduces several coding-level designs, deploying it in an existing storage system requires only two essential operations: (1) identifying helper sub-chunks during repair, and (2) performing encoding or decoding using its dedicated algorithms. The coefficients obtained (§4) are stored in the system configuration, avoiding any runtime search.

To simplify integration with existing storage systems, we package WiseCode as an erasure-code plugin with two core functions: one identifies the required helper sub-chunks for given lost chunks, and the other performs encoding or decoding over available chunks. The finite field arithmetic for encoding and decoding is implemented with AVX-512 instructions [52], following GF-Complete [50]. We integrate this plugin into Ceph v17.2.5 with only ∼400 lines of C++ glue code, enabling Ceph's OSDs (Object Storage Daemons) to invoke the wrapped functions.

6.2 Integration with Repair Scheduling

Degraded read. We accelerate degraded reads using partial-parallel repair [19] and repair pipelining [20], two *intra-stripe scheduling* methods that select relay nodes to perform partial repairs[3] in parallel. With WiseCode, a node storing a non-sibling chunk of the lost chunk cannot serve as a relay because it would need to transmit a full-chunk partial result rather than a subset of sub-chunks (§3.2), increasing repair traffic. We therefore restrict relay selection to nodes storing sibling chunks, while keeping each method's original scheduling logic between the relays and the target node unchanged. Non-relay nodes simply distribute their helper data evenly across the relays and the target node.

[3]Partial repair produces a partial result using a subset of helper data (treating other helper data as zeros); summing all partial results recovers the lost chunk. WiseCode, being linear, supports this naturally.

Full-node repair. We accelerate full-node repair using RepairBoost [21], an *inter-stripe scheduling* method that reorders transmissions across stripes to improve network utilization and runs atop any intra-stripe scheduling method—partial-parallel repair, repair pipelining, or conventional repair (without relays). With WiseCode, RepairBoost only needs to account for the lower traffic contribution of non-sibling chunks than sibling chunks of the lost chunk, estimating network utilization using weighted rather than uniform traffic.

All methods are implemented in a ∼6.9K-line C++ prototype comprising a coordinator that issues repair tasks and multiple agents that execute repair operations, with gRPC [53] for inter-node communication.

7 Evaluation

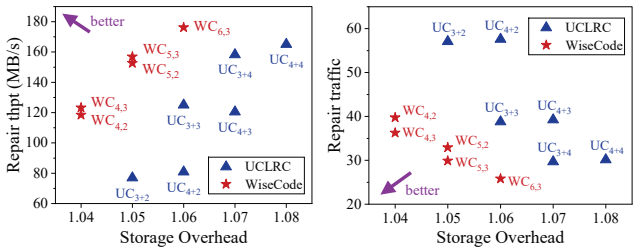
7.1 Methodology

We evaluate WiseCode against UCLRC (Uniform Cauchy LRC [6]), the state-of-the-art wide codes. We first use Ceph as the development platform, implementing both codes as erasure-code plugins for fairness, and examine offline and online repairs triggered by manually disabling one storage node. We measure repair throughput (repaired-data size divided by repair time), foreground I/O latency during online repair, and normal-write and degraded-read latencies without background repair. We then evaluate both codes with representative repair-scheduling techniques: RepairBoost [21] for full-node repair, and three strategies for single-chunk degraded reads—conventional repair (CR), partial-parallel repair (PPR) [19], and repair pipelining (RP) [20].

Code configurations. We configure UCLRC with (n,k,r,p), where r and p denote the numbers of global and local parities, to control storage overhead, repair traffic, and reliability (measured via MTTDL; see Appendix A.1). Both WiseCode and UCLRC use $k=100$, with remaining parameters varied to explore design trade-offs. Table 3 summarizes all configurations. UC_{r+p} denotes a $(100+r+p,100,r,p)$ UCLRC, and $\mathrm{WC}_{m,b}$ denotes a $(100+m,100,m^b)$ WiseCode.

Testbed. Experiments run on a cluster of 161 ecs.hfg6.large instances [54] on Alibaba Cloud. Each instance has 2 vCPUs and 8 GB RAM. Among them, 120 serve as storage nodes, each with a 100 GB elastic SSD (∼170 MB/s bandwidth), one is the monitor, and 40 act as clients. All instances connect via a 1 Gbps network.

System and workload settings. On Ceph, we use 512 placement groups [55] for balanced chunk distribution. Storage is prefilled using YCSB [56] with 4000 records (1.46 TB). During online repair, each client issues YCSB requests with read:write ratios of 50:50 (YCSB-a), 100:0 (YCSB-c), and 0:100 (YCSB-w), as summarized in Table 4. We emulate light and heavy loads using 10 and 40 clients. For evaluations with repair scheduling, 1000 stripes are randomly placed across the cluster before repair. RP is configured with a 32 KB slice



(a) Offline repair throughput. (b) Measured repair traffic.

Figure 11: Repair throughput and measured repair traffic during offline repair on Ceph.

size following prior work [20].

Chunk-size settings are listed in Table 5. In Ceph experiments, UCLRC uses a 4 MB chunk size; we avoid larger chunks (e.g., 8 MB) because Ceph performs client I/O at full-stripe granularity, which becomes very large under wide stripes and can trigger Out-Of-Memory failures on Ceph's OSDs in our testbed, particularly when multiple requests target the same OSD concurrently. For repair-scheduling experiments, UCLRC follows prior work [20, 21] and uses 64 MB chunks. WiseCode derives its chunk size from the corresponding UCLRC configuration, rounding up to the nearest multiple of $\alpha \cdot 4$ KB to satisfy 4 KB sub-chunk alignment, and this adjustment does not bring any unfair performance advantage.

7.2 Overall Performance with Ceph

Offline repair. Figures 11a and 11b show offline repair throughput and repair traffic (total network traffic divided by size of repaired chunks) versus storage overhead, averaged over 5 independent runs. The results show that WiseCode achieves a strictly better throughput-overhead tradeoff than UCLRC. With storage overhead 1.06, $WC_{6,3}$ achieves 2.18× and 1.41× the repair throughput of UC_{4+2} and UC_{3+3}, respectively. With storage overhead 1.05, $WC_{5,3}$ outperforms UC_{3+2} by 2.04× in the repair throughput. These throughput gains stem from WiseCode's reduced repair traffic, which enables faster recovery under the same network bandwidth.

Multi-dimensional Pareto Improvement. Figure 12 illustrates the joint improvements in storage overhead, reliability, and repair throughput. Unlike MTTDL analysis in Table 3, which assumes full network bandwidth utilization during repair, the repair rate used in the reliability model for this figure is derived from the observed repair throughput and node capacity in our testbed. Results show that, with a consistent overhead of 1.05, $WC_{5,2}$ offers two orders of magnitude higher MTTDL and 1.98× higher repair throughput compared to UC_{3+2}. $WC_{6,3}$ outperforms UC_{4+4} by offering 2% lower overhead, 1.81× higher MTTDL, 14.4% lower repair traffic, and 6.8% higher repair throughput against UC_{4+4}. We further conduct an iso-MTTDL comparison by interpolating UCLRC results along log-linear curve (mapping log MTTDL to linear throughput) within the same overhead configurations. Our analysis shows that with the same MTTDL, $WC_{5,2}$ offers 1.53× higher repair throughput with 1% lower overhead, or 1.11× higher repair throughput with 2% lower overhead compared to UCLRC.

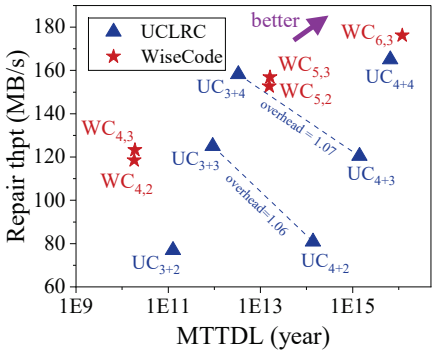


Figure 12: Joint comparison of storage overhead, MTTDL and repair throughput between WiseCode and UCLRC.

Online repair. Figure 13 plots average foreground I/O latency versus background repair throughput. At equal storage overhead, WiseCode improves both metrics: $WC_{6,3}$ delivers 36%–102% higher repair throughput and 11%–27% lower I/O latency than UC_{4+2}. Even against UC_{4+4}, which has 2% higher storage overhead than $WC_{6,3}$, $WC_{6,3}$ attains 27.7%–45.6% higher repair throughput with comparable I/O latency (within ±5%), except under YCSB-c with 40 clients, where it yields 38% higher throughput with 10% higher latency.

Normal-write and degraded-read latencies. Figure 14 shows normal-write (YCSB-w) and degraded-read (YCSB-c) latencies, each averaged over a 3-minute run. All codes exhibit similar performance (within ±5%), indicating that WiseCode introduces negligible read/write overhead, except a 6% latency increase for $WC_{5,3}$ (YCSB-c, 40 clients) and 7% for $WC_{6,3}$ (YCSB-c, 10 clients). These deviations stem from WiseCode's chunk-size rounding (Table 5), which causes Ceph to pad zeros and inflate I/O. In practice, this effect can be eliminated via precise chunking techniques [9, 27].

7.3 Performance with Repair Scheduling

Full-node repair. Figure 15 shows full-node repair throughput versus storage overhead, averaged over 5 independent runs. We present results using CR as the intra-stripe scheduling strategy for RepairBoost, as it consistently outperforms PPR and RP for both WiseCode and UCLRC in our experiments. The results show that WiseCode achieves a strictly better throughput-storage tradeoff than UCLRC, exhibiting the same performance advantages observed in the Ceph experiments. For example, with storage overhead 1.06, $WC_{6,3}$ achieves 2.38× and 1.56× the repair throughput of UC_{4+2} and UC_{3+3}, respectively. Even against UC_{4+4}, $WC_{6,3}$ achieves 13.6%

Table 3: Evaluated codes with their storage overhead, theoretical repair traffic for a single-chunk failure, and MTTDL. UC_{r+p} denotes a $(100+r+p, 100, r, p)$ UCLRC, and $WC_{m,b}$ denotes a $(100+m, 100, m^b)$ WiseCode.

Code	UC_{3+2}	UC_{3+3}	UC_{3+4}	UC_{4+2}	UC_{4+3}	UC_{4+4}	$WC_{4,2}$	$WC_{4,3}$	$WC_{5,2}$	$WC_{5,3}$	$WC_{6,3}$
Storage overhead	1.05	1.06	1.07	1.06	1.07	1.08	1.04	1.04	1.05	1.05	1.06
Repair traffic	51.50	34.33	25.75	52.00	34.67	26.00	34.75	31.52	28.42	25.60	21.59
MTTDL (years)	2.51×10^{10}	1.69×10^{11}	6.38×10^{11}	2.59×10^{13}	2.64×10^{14}	1.20×10^{15}	3.47×10^{9}	3.82×10^{9}	2.83×10^{12}	3.14×10^{12}	2.46×10^{15}

Table 4: Workloads used in the Ceph experiments.

Workload	R/W ratio	# Records	# KV pairs	Value size	Total
YCSB-a	50%/50%				
YCSB-c	100%/0%	4000	400	10^6 B	1.46 TB
YCSB-w	0%/100%				

Table 5: Chunk sizes used in the two sets of experiments.

Chunk size (KB)	UCLRCs	$WC_{4,2}$	$WC_{4,3}$	$WC_{5,2}$	$WC_{5,3}$	$WC_{6,3}$
Ceph	4096	4096	4096	4100	4500	4320
Repair-scheduling	65536	65536	65536	66000	66000	65664

higher repair throughput, despite 2% lower storage overhead.

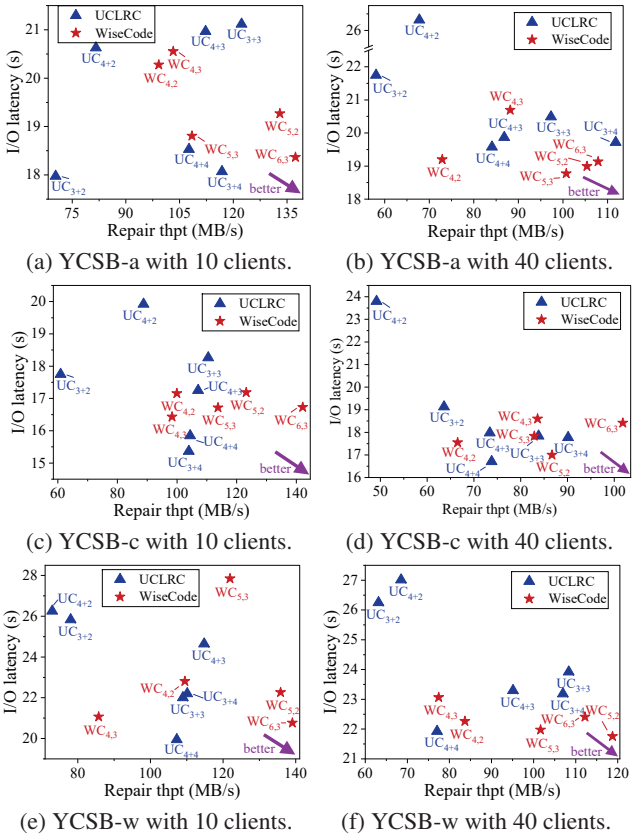
Single-chunk degraded read. Figure 17 shows single-chunk degraded-read latency, averaged over 10 independent runs. The results show that WiseCode's improvements vary across the three methods. For example, at equal storage overhead, WiseCode reduces degraded-read latency by 32%–55% compared to UCLRC under CR, due to its 37%–58% lower repair traffic. Under PPR, WiseCode and UCLRC achieve comparable latencies. In contrast, under RP, WiseCode incurs 117%–200% higher latency than UCLRC because the relay-selection constraints limit scheduling flexibility (§6.2).

The potential increase in pipelining latency represents an acceptable tradeoff for two primary reasons. First, this overhead only impacts foreground degraded reads, while WiseCode significantly enhances background repair throughput, which shortens the degraded window and directly improves MTTDL. Second, the performance impact is minimal given the rarity of failure states. Modeling with either Binomial distribution or Markov chain indicates that a stripe resides in degraded states for less than 1% of its total lifetime. This translates to a negligible impact on aggregate read performance, especially for warm and cold storage where accesses are infrequent and latency-insensitive [57].

7.4 Ablation Study of Two-Stage Coding

We measure WiseCode's computational overhead by encoding and decoding a stripe from k in-memory chunks using a single thread. Coding throughput is defined as the stripe size divided by the coding time, averaged over 10 independent runs, with chunk sizes matching the Ceph experiments.

We evaluate four variants to isolate each optimization: (1) **baseline**: generator-matrix encoding; (2) **+framework**: two-stage encoding framework only; (3) **+stage-1 opt**: framework plus data-aggregation optimization; (4) **+full opts**: all optimizations enabled (parity-solving optimization added).



(a) YCSB-a with 10 clients. (b) YCSB-a with 40 clients.

(c) YCSB-c with 10 clients. (d) YCSB-c with 40 clients.

(e) YCSB-w with 10 clients. (f) YCSB-w with 40 clients.

Figure 13: Foreground I/O latency and background repair throughput during online repair on Ceph.

As shown in Figure 16, the two-stage framework and the data-aggregation optimization increase throughput by 2.6×–3.0× and 2.0×–5.0×, respectively. The parity-solving optimization provides an additional 1.5×–1.8× speedups for $WC_{5,3}$ and $WC_{6,3}$. Overall, the two-stage encoding approach achieves 5.5×–22.4× throughput over the baseline.

7.5 Applicability to Diverse Hardware

Applicability to HDDs. While our cloud testbed uses SSDs, as in production wide-stripe deployments such as VastData [58], we also evaluate WiseCode's applicability to HDDs, which remain prevalent for their lower cost per byte [59]. We examine HDD read bandwidth during repair, where read operations dominate (95%–97%) but may be slowed by data fragmentation incurred by sub-packetization.

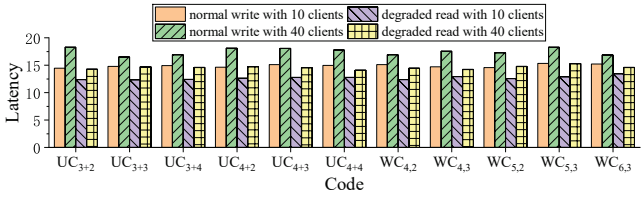


Figure 14: Normal-write and degraded-read latencies on Ceph.

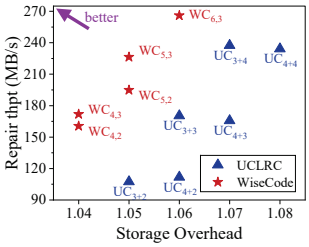


Figure 15: Full-node repair throughput with RepairBoost.

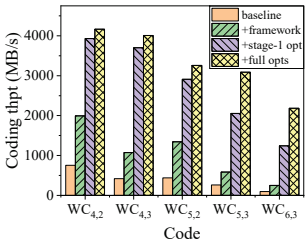


Figure 16: Coding throughput.

For example, during the repair process of $WC_{4,2}$, each helper chunk exhibits one of three read patterns with different percentages: repeatedly reading 1 sub-chunk and skipping 3 (44.2%), reading 4 consecutive sub-chunks (44.2%), or reading all 16 sub-chunks (11.6%).

We emulate these read patterns on a 7200 RPM HDD using *fio* [60] (iodepth = 8), measure per-pattern read bandwidth, and compute the percentage-weighted average, as shown in Figure 18. The results show that fragmentation reduces read bandwidth, but WiseCode still match raw network bandwidth with practical chunk sizes (e.g., 12 MB or 64 MB), ensuring that HDDs are not the bottleneck in the read-transfer pipeline [21]. Under the 1 Gbps (∼125 MB/s) network in our testbed, $WC_{4,2}$ requires 12 MB or larger chunks, and $WC_{6,3}$—despite its higher sub-packetization level—requires 64 MB or larger, both easily satisfied by the 64 MB–1 GB chunk sizes used in production [9, 11].

Applicability to faster network. Production nodes typically pair high-bandwidth network (e.g. 10+ Gbps) with high-density disk arrays (e.g., 36–216 HDDs [61]) to maximize network utilization. For instance, the Amazon D3.8xlarge instance [62] pairs 25 Gbps network with 24 HDDs, while the Tencent Cloud DA4m.32XLARGE512 instance [63] pairs 100 Gbps (50 Gbps full-duplex) network with 48 HDDs. For these big-data-oriented instances, WiseCode's repair improvement remains highly applicable because the aggregate storage bandwidth typically exceeds the network capacity. Specifically, as WiseCode achieves over 130 MB/s per disk (with 64 MB chunks), a disk array with 24 HDDs can provide over 25 Gbps of aggregate I/O, which matches the 25 Gbps network and does not become a bottleneck. Furthermore, given its high coding throughput (> 2000 MB/s), WiseCode can process 10+ Gbps network traffic using only a single dedicated CPU thread without incurring computational bottleneck.

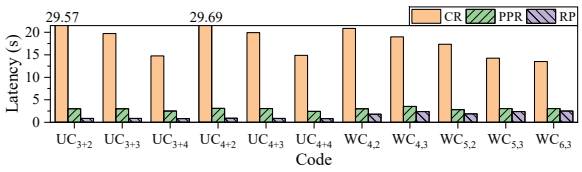


Figure 17: Single-chunk degraded-read latency with repair scheduling.

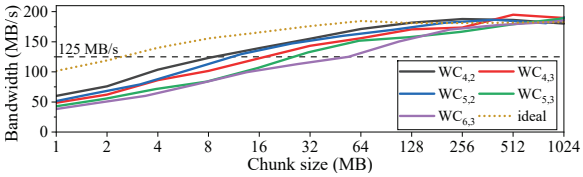


Figure 18: Average HDD read bandwidth during repair.

8 Related Work

Wide codes. Most wide-stripe solutions are based on LRCs [6, 17, 27, 29, 30], which reduce repair traffic but sacrifice the MDS property. WiseCode is the first wide-stripe vector code that achieves near-optimal repair traffic while preserving the MDS property. VAST [58] designs a wide-stripe locally decodable code, but its design details remain undisclosed.

Vector codes. Clay codes [34], Butterfly codes [32], and other MSR codes suffer from sub-packetization blowup in wide stripes. Zigzag codes [64] alleviate this only for $m = 2$, offering insufficient reliability in wide stripes. HashTag+ [37] and RS-ET [36] codes avoid sub-packetization blowup but incur higher repair traffic than WiseCode and face the scalability barrier in coefficient search. WiseCode cannot yet support the flexible sub-packetization levels of RS-ET codes due to limitations of current MSR templates; future work will explore new template designs to enable this.

Repair scheduling. Prior work [19–21, 24] optimizes repair transmission schedules to better utilize network resources and accelerate repair. WiseCode is complementary: it reduces repair traffic itself and can be combined with these scheduling techniques through appropriate adaptions, as discussed in Section 6.

9 Conclusion

By breaking three scalability barriers, we present WiseCode, the first practical and scalable wide-stripe vector-coding approach that achieves near-optimal repair traffic and ultra-low storage overhead (e.g., 1.04–1.06) in large-scale storage clusters. Experimental results show that WiseCode delivers a strictly better tradeoff between repair performance and storage overhead than Google's UCLRCs.

Acknowledgments

We sincerely thank our shepherd for helping us improve the paper. We are also grateful to all reviewers of this paper for their helpful comments and feedback. This work was supported by the National Natural Science Foundation of China under Grant 62025203.

References

[1] Iyswarya Narayanan, Di Wang, Myeongjae Jeon, Bikash Sharma, Laura Caulfield, Anand Sivasubramaniam, Ben Cutler, Jie Liu, Badriddine Khessib, and Kushagra Vaid. SSD failures in datacenters: What? when? and why? In *Proceedings of the 9th ACM International on Systems and Storage Conference*, pages 1–11, 2016.

[2] Bianca Schroeder and Garth A Gibson. Understanding disk failure rates: What does an MTTF of 1,000,000 hours mean to you? *ACM Transactions on Storage (TOS)*, 3(3):8–es, 2007.

[3] Eduardo Pinheiro, Wolf-Dietrich Weber, and Luiz André Barroso. Failure trends in a large disk drive population. In *5th USENIX Conference on File and Storage Technologies (FAST 07)*, pages 17–29, 2007.

[4] Stathis Maneas, Kaveh Mahdaviani, Tim Emami, and Bianca Schroeder. A study of SSD reliability in large scale enterprise storage deployments. In *18th USENIX Conference on File and Storage Technologies (FAST 20)*, pages 137–149, 2020.

[5] David Reinsel-John Gantz-John Rydning, J Reinsel, and J Gantz. The digitization of the world from edge to core. *Framingham: International Data Corporation*, 16, 2018.

[6] Saurabh Kadekodi, Shashwat Silas, David Clausen, and Arif Merchant. Practical Design Considerations for Wide Locally Recoverable Codes (LRCs). In *21st USENIX Conference on File and Storage Technologies (FAST 23)*, pages 1–16, 2023.

[7] Apache Software Foundation. HDFS Erasure Coding. https://hadoop.apache.org/docs/current/hadoop-project-dist/hadoop-hdfs/HDFSErasureCoding.html, 2025.

[8] Ceph Community. Ceph Git Repository. https://github.com/ceph/ceph, 2025.

[9] Weidong Zhang, Erci Xu, Qiuping Wang, Xiaolu Zhang, Yuesheng Gu, Zhenwei Lu, Tao Ouyang, Guanqun Dai, Wenwen Peng, Zhe Xu, et al. What's the Story in EBS Glory: Evolutions and Lessons in Building Cloud Block Store. In *22nd USENIX Conference on File and Storage Technologies (FAST 24)*, pages 277–291, 2024.

[10] Qiang Li, Qiao Xiang, Yuxin Wang, Haohao Song, Ridi Wen, Wenhui Yao, Yuanyuan Dong, Shuqi Zhao, Shuo Huang, Zhaosheng Zhu, et al. More than capacity: performance-oriented evolution of Pangu in Alibaba. In *21st USENIX Conference on File and Storage Technologies (FAST 23)*, pages 331–346, 2023.

[11] Subramanian Muralidhar, Wyatt Lloyd, Sabyasachi Roy, Cory Hill, Ernest Lin, Weiwen Liu, Satadru Pan, Shiva Shankar, Viswanath Sivakumar, Linpeng Tang, et al. f4: Facebook's warm BLOB storage system. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 383–398, 2014.

[12] Su Zhou, Erci Xu, Hao Wu, Yu Du, Jiacheng Cui, Wanyu Fu, Chang Liu, Yingni Wang, Wenbo Wang, Shouqu Sun, et al. SMRSTORE: A storage engine for cloud object storage on HM-SMR drives. In *21st USENIX Conference on File and Storage Technologies (FAST 23)*, pages 395–408, 2023.

[13] Michael Ovsiannikov, Silvius Rus, Damian Reeves, Paul Sutter, Sriram Rao, and Jim Kelly. The quantcast file system. *Proceedings of the VLDB Endowment*, 6(11):1092–1101, 2013.

[14] Youngmoon Lee, Hasan Al Maruf, Mosharaf Chowdhury, Asaf Cidon, and Kang G Shin. Hydra: Resilient and highly available remote memory. In *20th USENIX Conference on File and Storage Technologies (FAST 22)*, pages 181–198, 2022.

[15] Qiang Li, Lulu Chen, Xiaoliang Wang, Shuo Huang, Qiao Xiang, Yuanyuan Dong, Wenhui Yao, Minfei Huang, Puyuan Yang, Shanyang Liu, et al. Fisc: a large-scale cloud-native-oriented file system. In *21st USENIX Conference on File and Storage Technologies (FAST 23)*, pages 231–246, 2023.

[16] Yanjing Ren, Yuanming Ren, Xiaolu Li, Yuchong Hu, Jingwei Li, and Patrick PC Lee. ELECT: Enabling erasure coding tiering for {LSM-tree-based} storage. In *22nd USENIX Conference on File and Storage Technologies (FAST 24)*, pages 293–310, 2024.

[17] Yuchong Hu, Liangfeng Cheng, Qiaori Yao, Patrick PC Lee, Weichun Wang, and Wei Chen. Exploiting Combined Locality for Wide-Stripe Erasure Coding in Distributed Storage. In *FAST*, pages 233–248, 2021.

[18] Meng Wang, Jiajun Mao, Rajdeep Rana, John Bent, Serkay Olmez, Anjus George, Garrett Wilson Ransom, Jun Li, and Haryadi S Gunawi. Design considerations and analysis of multi-level erasure coding in large-scale

data centers. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–13, 2023.

[19] Subrata Mitra, Rajesh Panta, Moo-Ryong Ra, and Saurabh Bagchi. Partial-parallel-repair (PPR): a distributed technique for repairing erasure coded storage. In *Proceedings of the eleventh European conference on computer systems*, pages 1–16, 2016.

[20] Runhui Li, Xiaolu Li, Patrick PC Lee, and Qun Huang. Repair pipelining for Erasure-Coded storage. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, pages 567–579, 2017.

[21] Shiyao Lin, Guowen Gong, Zhirong Shen, Patrick PC Lee, and Jiwu Shu. Boosting Full-Node Repair in Erasure-Coded Storage. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, pages 641–655, 2021.

[22] Korlakai Vinayak Rashmi, Nihar B Shah, Dikang Gu, Hairong Kuang, Dhruba Borthakur, and Kannan Ramchandran. A solution to the network challenges of data recovery in erasure-coded distributed storage systems: A study on the facebook warehouse cluster. In *Presented as part of the 5th USENIX Workshop on Hot Topics in Storage and File Systems*, 2013.

[23] Zhufan Wang, Guangyan Zhang, Yang Wang, Qinglin Yang, and Jiaji Zhu. Dayu: Fast and low-interference data recovery in very-large storage systems. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 993–1008, 2019.

[24] Yuhui Cai, Shiyao Lin, Zhirong Shen, Jiahui Yang, and Jiwu Shu. Chameleonec: Exploiting tunability of erasure coding for low-interference repair. In *2025 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 15–28. IEEE, 2025.

[25] Zhirong Shen, Yuhui Cai, Keyun Cheng, Patrick PC Lee, Xiaolu Li, Yuchong Hu, and Jiwu Shu. A survey of the past, present, and future of erasure coding for storage systems. *ACM Transactions on Storage*, 21(1):1–39, 2025.

[26] Irving S Reed and Gustave Solomon. Polynomial codes over certain finite fields. *Journal of the society for industrial and applied mathematics*, 8(2):300–304, 1960.

[27] Cheng Huang, Huseyin Simitci, Yikang Xu, Aaron Ogus, Brad Calder, Parikshit Gopalan, Jin Li, and Sergey Yekhanin. Erasure coding in windows azure storage. In *Presented as part of the 2012 USENIX Annual Technical Conference (USENIX ATC 12)*, pages 15–26, 2012.

[28] Maheswaran Sathiamoorthy, Megasthenis Asteris, Dimitris Papailiopoulos, Alexandros G Dimakis, Ramkumar Vadali, Scott Chen, and Dhruba Borthakur. XORing elephants: novel erasure codes for big data. *Proceedings of the VLDB Endowment*, 6(5):325–336, 2013.

[29] Itzhak Tamo and Alexander Barg. A family of optimal locally recoverable codes. *IEEE Transactions on Information Theory*, 60(8):4661–4676, 2014.

[30] Oleg Kolosov, Gala Yadgar, Matan Liram, Itzhak Tamo, and Alexander Barg. On fault tolerance, locality, and optimality in locally repairable codes. *ACM Transactions on Storage (TOS)*, 16(2):1–32, 2020.

[31] KV Rashmi, Preetum Nakkiran, Jingyan Wang, Nihar B Shah, and Kannan Ramchandran. Having your cake and eating it too: Jointly optimal erasure codes for I/O, storage, and network-bandwidth. In *13th USENIX Conference on File and Storage Technologies (FAST 15)*, pages 81–94, 2015.

[32] Lluis Pamies-Juarez, Filip Blagojevic, Robert Mateescu, Cyril Gyuot, Eyal En Gad, and Zvonimir Bandic. Opening the chrysalis: On the real repair performance of MSR codes. In *14th USENIX conference on file and storage technologies (FAST 16)*, pages 81–94, 2016.

[33] Yuchong Hu, Henry CH Chen, Patrick PC Lee, and Yang Tang. NCCloud: applying network coding for the storage repair in a cloud-of-clouds. In *FAST*, volume 21, 2012.

[34] Myna Vajha, Vinayak Ramkumar, Bhagyashree Puranik, Ganesh Kini, Elita Lobo, Birenjith Sasidharan, P Vijay Kumar, Alexandar Barg, Min Ye, Srinivasan Narayanamurthy, et al. Clay codes: Moulding MDS codes to yield an MSR code. In *16th USENIX Conference on File and Storage Technologies (FAST 18)*, pages 139–154, 2018.

[35] Katina Kralevska, Danilo Gligoroski, Rune E Jensen, and Harald Øverby. Hashtag erasure codes: From theory to practice. *IEEE Transactions on Big Data (TBD)*, 4(4):516–529, 2017.

[36] Kaicheng Tang, Keyun Cheng, Helen H. W. Chan, Xiaolu Li, Patrick P. C. Lee, Yuchong Hu, Jie Li, and Ting-Yi Wu. Balancing repair bandwidth and subpacketization in erasure-coded storage via elastic transformation. In *IEEE INFOCOM 2023 - IEEE Conference on Computer Communications*, pages 1–10, 2023.

[37] Katina Kralevska and Danilo Gligoroski. An explicit construction of systematic mds codes with small subpacketization for all-node repair. In *2018 IEEE 16th Intl Conf on Dependable, Autonomic and Secure Computing,*

16th Intl Conf on Pervasive Intelligence and Computing, 4th Intl Conf on Big Data Intelligence and Computing and Cyber Science and Technology Congress (DASC/PiCom/DataCom/CyberSciTech), pages 1080–1084. IEEE, 2018.

[38] Chuang Gan, Yuchong Hu, Leyan Zhao, Xin Zhao, Pengyu Gong, and Dan Feng. Revisiting network coding for warm blob storage. In *23rd USENIX Conference on File and Storage Technologies (FAST 25)*, pages 139–154, 2025.

[39] Alexandros G Dimakis, P Brighten Godfrey, Yunnan Wu, Martin J Wainwright, and Kannan Ramchandran. Network coding for distributed storage systems. *IEEE transactions on information theory*, 56(9):4539–4551, 2010.

[40] R. Singleton. Maximum distanceq-nary codes. *IEEE Transactions on Information Theory*, 10(2):116–118, 1964.

[41] John E Angus. On computing mtbf for a k-out-of-n: G repairable system. *IEEE Transactions on Reliability*, 37(3):312–313, 2002.

[42] Saurabh Kadekodi, Francisco Maturana, Sanjith Athlur, Arif Merchant, KV Rashmi, and Gregory R Ganger. Tiger:{Disk-Adaptive} redundancy without placement restrictions. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 413–429, 2022.

[43] SB Balaji, Myna Vajha, and P Vijay Kumar. Lower bounds on the sub-packetization level of MSR codes and characterizing optimal-access MSR codes achieving the bound. *IEEE Transactions on Information Theory*, 68(10):6452–6471, 2022.

[44] James S Plank, Scott Simmerman, and Catherine D Schuman. Jerasure: A library in C/C++ facilitating erasure coding for storage applications-Version 1.2. *University of Tennessee, Tech. Rep. CS-08-627*, 23, 2008.

[45] Intel, Inc. Intelligent Storage Acceleration Library. https://github.com/intel/isa-l, 2025.

[46] Min Ye and Alexander Barg. Explicit constructions of optimal-access mds codes with nearly optimal sub-packetization. *IEEE Transactions on Information Theory*, 63(10):6307–6317, 2017.

[47] Jonggyu Park and Young Ik Eom. FragPicker: A new defragmentation tool for modern storage devices. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles (SOSP 21)*, pages 280–294, 2021.

[48] Theophilus Benson, Aditya Akella, and David A Maltz. Network traffic characteristics of data centers in the wild. In *Proceedings of the 10th ACM SIGCOMM conference on Internet measurement*, pages 267–280, 2010.

[49] Guodong Li, Ningning Wang, Sihuang Hu, and Min Ye. MSR codes with linear field size and smallest sub-packetization for any number of helper nodes. *IEEE Transactions on Information Theory*, 2024.

[50] James S Plank, Ethan L Miller, Kevin M Greenan, Benjamin A Arnold, John A Burnum, Adam W Disney, and Allen C McBride. GF-Complete: A Comprehensive Open Source Library for Galois Field Arithmetic Version 1.02, 2014.

[51] Sage A Weil, Scott A Brandt, Ethan L Miller, Darrell DE Long, and Carlos Maltzahn. Ceph: A scalable, high-performance distributed file system. In *Proceedings of the 7th symposium on Operating systems design and implementation*, pages 307–320, 2006.

[52] Intel, Inc. AVX-512 Overview. https://www.intel.com/content/www/us/en/architecture-and-technology/avx-512-overview.html. 2025.

[53] gRPC Authors. gRPC, A high performance, open source universal RPC framework. https://grpc.io/, 2025.

[54] Alibaba Cloud Company. User guide of Instance families with high clock speeds (hf series). https://www.alibabacloud.com/help/en/ecs/user-guide/instance-families-with-high-clock-speeds, 2025.

[55] Ceph authors and contributors. Placement Groups in Ceph. https://docs.ceph.com/en/quincy/rados/operations/placement-groups/, 2025.

[56] YCSB Community. Yahoo! Cloud Serving Benchmark (YCSB) Git Repository. https://github.com/brianfrankcooper/YCSB, 2025.

[57] Timothy Kim, Sanjith Athlur, Saurabh Kadekodi, Francisco Maturana, Dax Delvira, Arif Merchant, Gregory R Ganger, and KV Rashmi. Morph: Efficient file-lifetime redundancy management for cluster file systems. In *Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles*, pages 330–346, 2024.

[58] VastData, Inc. Breaking Resiliency Trade-offs With Locally Decodable Erasure Codes. https://vastdata.com/blog/breaking-resiliency-trade-offs-with-locally-decodable-erasure-codes, 2019.

[59] Sanjith Athlur, Timothy Kim, Saurabh Kadekodi, Francisco Maturana, Xavier Ramos, Arif Merchant, KV Rashmi, and Gregory R Ganger. Okapi: Decoupling data striping and redundancy grouping in cluster file systems. In *19th USENIX Symposium on Operating Systems Design and Implementation (OSDI 25)*, pages 897–914, 2025.

[60] Jens Axboe. FIO - Flexible I/O Tester. https://github.com/axboe/fio, 2025.

[61] Chunqiang Tang. Meta's hyperscale infrastructure: Overview and insights. *Communications of the ACM*, 68(2):52–63, 2025.

[62] Amazon Web Service. Amazon EC2 D3 and D3en Instances. https://aws.amazon.com/cn/ec2/instance-types/d3/, 2026.

[63] Tencent Cloud. Big data DA4m instance specifications. https://www.tencentcloud.com/document/product/213/59879, 2026.

[64] Itzhak Tamo, Zhiying Wang, and Jehoshua Bruck. Zigzag codes: MDS array codes with optimal rebuilding. *IEEE Transactions on Information Theory*, 59(3):1597–1616, 2012.

[65] Kashi Venkatesh Vishwanath and Nachiappan Nagappan. Characterizing cloud computing hardware reliability. In *Proceedings of the 1st ACM symposium on Cloud computing*, pages 193–204, 2010.

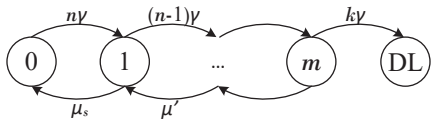
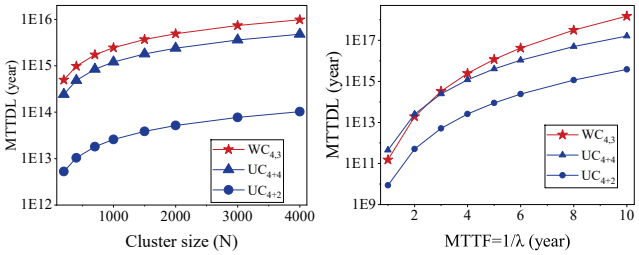


Figure 19: Markov model used for reliability analysis.



(a) Varying cluster size N. (b) Varying MTTF $1/\lambda$.

Figure 20: Sensitivity analysis on MTTDL.

A Appendix

A.1 MTTDL and Markov Model

We evaluate data reliability using the mean-time-to-data-loss (MTTDL) metric, modeled via continuous-time Markov chains following prior works [6, 27]. As shown in Figure 19, each circles corresponds to a health states of a stripe: state i indicates that i chunk(s) are lost, while the Data Loss (DL) state represents unrecoverable data loss. A stripe transitions between adjacent states at specified rates, and MTTDL is defined as the expected time for the stripe to transition from state 0 to the DL state.

The state transition rate from state i to state $i+1$ is $(n-i)\gamma$, where γ is the failure rate of each node and n is the stripe width. A stripe enters the DL state once fewer than k healthy chunks remain. For UCLRC, local parity does not fully protect the stripe: when $i \geq r$, an additional failure may directly cause data loss with probability p_i, which is empirically measured via simulation [6]. The transition rate μ_s from state 1 back to state 0 is $\mu_s = \frac{1}{T_s}$, where $T_s = \frac{CS}{\varepsilon(N-1)B}$ is the mean time to repair a single-failure stripe. Here, S is the node capacity, C is the repair traffic normalized by chunk size, CS is the total network traffic during repair, N is the cluster size, B is the per-node network bandwidth, and ε is the fraction of network bandwidth allocated to repair. The term $\varepsilon(N-1)B$ represents the total available bandwidth assuming the repair traffic is evenly distributed across nodes. Since multi-chunk failures are rare and prioritized, the repair rate μ', i.e. the transition rate from state $i(>1)$ to state $i-1$, is basically dominated by $\mu' = \frac{1}{T'}$, where T' denotes the time to detect multi-failure stripes and trigger repair [6, 27].

We configure the parameters as follows: the mean-time-to-failure (MTTF) $1/\gamma = 4$ years, $N = 1000$ nodes, $S = 16$ TB, $B = 10$ Gbps, $T' = 30$ minutes, $\varepsilon = 0.1$. The MTTDL results of evaluated codes in our experiments are shown in Table 3. Figure 20 further illustrates the sensitivity of MTTDL to variations in N and $\frac{1}{\gamma}$ for $WC_{6,3}$, UC_{4+2}, and UC_{4+4}. The results indicate that while the MTTDL of both WiseCode and UCLRC scales at a similar rate with N, WiseCode exhibits a faster growth rate as $1/\gamma$ increases. Although $WC_{4,3}$ underperforms UC_{4+4} under an extremely low MTTF ($\leq$2 years), WiseCode demonstrates superior reliability in more realistic MTTF ranges (3–5 years) [17, 28, 65].

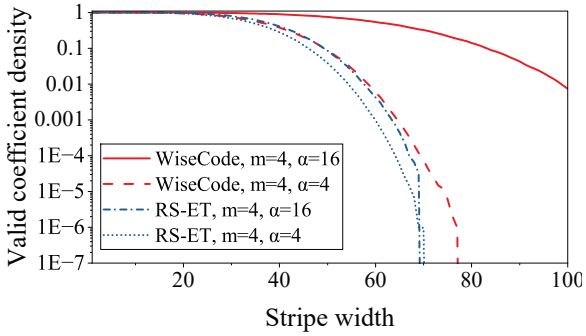


Figure 21: Density of valid coefficient combinations.

A.2 Existence of Valid Coefficients

As Table 2 summarizes the maximum stripe width achieved in our experiments, we believe that these results suggest that the theoretical ceiling for WiseCode's scalability is substantially higher. To validate this, we randomly choose up to 1,000,000 coefficient combination from GF(2^{16}) for RS-ET codes and WiseCode, calculating the density of valid assignments for each. And we observe that WiseCode has a higher probability to find valid coefficient under stripe-width configurations. As shown in Figure 21, the density of valid combinations for RS-ET codes with $m = 4$ decreases precipitously when the stripe width approaches 70. While WiseCode with $m = 4$ and $\alpha = 4$ exhibits a slightly better density curve, the density improves significantly as α increases. This improvement is driven by divide-and-conquer verification, which effectively reduces the number of coefficient constraints (§4.1) and thus render less coefficient combinations invalid.

A.3 Revisiting the Construction of WiseCode

Section 3 illustrates the template-based construction of WiseCode. Here we offer a straightforward matrix-based construction to simplify the discussions and proofs. We focus on how to directly obtain the parity-check matrix P, i.e., the entire coefficient matrix of coding equations, such that the coding equations can be collectively defined as $P \times C = \vec{0}$, where C represents all chunks.

In an (n,k,α) WiseCode built atop the (n_{msr},k_{msr}) MSR template, we have $m = n-k = n_{msr}-k_{msr}$. Define $b = n_{msr}/m$, then $\alpha = m^b$. If the $m \times 1$ coefficient vector of each sub-chunk is considered as a element, the parity-check matrix (i.e., coeffi-

cient matrix) P corresponding to all n chunks can be regarded as a $\alpha \times n\alpha$ matrix, where each of the n chunks corresponds to a $\alpha \times \alpha$ sub-parity-check-matrix (*SPCM*), i.e., Chunk_i corresponds to $SPCM_i$. Basically, each row of $SPCM_i$ corresponds to a sub-stripe, and each column corresponds to a sub-chunk. The elements of $SPCM_i$ are given as follows.

As discussed in Section 3.1, Chunk_i's parameters x and s are inherited from the template and are originally defined in Rule #2 (§2.3). The position of each element in the *SPCM* is denoted by the coordinate (r,c) starting from the top left corner, r for row and c for column. Given the base-m representation of c as $c = (c_{b-1}, c_{b-2}, \ldots, c_0)_m$, we define $c(x,s) = (c_{b-1}, \ldots, c_{s+1}, x, c_{s-1}, \ldots, c_0)_m$. We also define $\vec{\lambda} = [1, \lambda, \lambda^2, \ldots, \lambda^{m-1}]^T$. With m parameters λ_{im} to $\lambda_{(i+1)m-1}$ as specified in Section 4, the elements in the $SPCM_i$ of chunk_i are:

- $\overrightarrow{\lambda_{im+c_s}}$, if $c = r$ or $c(x,s) = r$,
- $\vec{0}$, otherwise.

We note that the conditions "$c = r$" and "$c(x,s) = r$" match the Rule #1 and Rule #2 in Section 2.3, respectively.

Lemma A.1. *Row_r of $SPCM_i$ contains 1 or m non-zero elements: If $r_s = x$, Row_r contains m non-zero elements at columns with indices $\{r(t,s) \mid 0 \le t < m\}$, with the element values specified as $\{\overrightarrow{\lambda_{im+t}} \mid 0 \le t < m\}$. If $r_s \neq x$, Row_r contains one non-zero element only at Column_r, with the element value specified as λ_{im+r_s}.*

A.4 WiseCode Repair Pattern

We prove the correctness of WiseCode repair pattern shown in Section 3.2. The key is to answer two questions: (1) For the lost chunk i, which sub-stripes are selected for decoding; (2) In a helper chunk i', which sub-chunks belong to the selected sub-stripes.

Answer for question (1). With the parameters x and s of chunk i, the α/m sub-stripes whose indices j satisfy $j_s = x$ are selected for decoding. According to Lemma A.1, each Row_j in the $SPCM_i$ contains m non-zero elements at columns with the indices $\{j(t,s) \mid 0 \le t < m\}$, therefore the Sub-Stripe_j can recover the m corresponding sub-chunks by solving the equation on Row_j. It is obvious that for any two different selected sub-stripes, their j vary at some digit j_p ($p \neq s$), therefore the two sets of $\{j(t,s) \mid 0 \le t < m\}$ have no intersection, and traversing all selected α/m sub-stripes can recover all α lost sub-chunks.

Answer for question (2). If $\text{Chunk}_{i'}$ is a sibling chunk of Chunk_i, it is obvious that it needs to contribute all α sub-chunks as helper sub-chunks, because $SPCM_{i'}$ is identical to $SPCM_i$ regarding the positions of non-zero elements, and these positions span all columns (sub-chunks).

If $\text{Chunk}_{i'}$ is not a sibling chunk of Chunk_i, we prove that in $SPCM_{i'}$, any non-zero element of Row_j only occurs within columns whose indices c satisfy $c_s = x$, i.e., having the same value range as the indices of the selected α/m rows (sub-stripes). Let the parameters of $\text{Chunk}_{i'}$ to be s' and x', then we discuss the position of non-zero elements in each Row_j based on Lemma A.1. If $j_{s'} = x'$, then the Column_c having a non-zero elements satisfies $c \in \{j(t,s') \mid 0 \le t < m\}$. In this case we have $s' \neq s$, otherwise $s' = s$ and $x' = j_{s'} = j_s = x$, which contradicts the assumption that $\text{Chunk}_{i'}$ is not a sibling chunk of Chunk_i. Therefore, any $c \in \{j(t,s') \mid 0 \le t < m\}$ satisfies $c_s = j_s = x$ because the sth digit of $j(t,s')$ is identical to j. If $j_{s'} \neq x'$, then the Column_c having a non-zero elements satisfies $c = j$, which also means $c_s = x$ because $j_s = x$. In conclusion, any non-zero element of Row_j only occurs within columns whose indices c satisfy $c_s = x$, which means that the indices c of helper sub-chunks also satisfy $c_s = x$ since each column corresponds to a sub-chunk with the same index.

A.5 Matrix-block-repetition pattern

Recall that each element of $SPCM_i$ corresponds to a coefficient vector of a sub-chunk, and there are at most m non-zero elements in each row of $SPCM_i$ (Lemman A.1). The m^b sub-chunks of chunk_i can be partitioned into m^{b-1} mutually disjoint subsets, each containing m sub-chunks, such that any sub-stripe only contains sub-chunks within a single subset. Using the sub-chunk index to describe each subset, sub-chunk_c belongs to the subset $\{c(t,s) \mid 0 \le t < m\}$, and these m sub-chunks sequentially corresponds to coefficients $\{\overrightarrow{\lambda_{im+t}} \mid 0 \le t < m\}$. The indices j of sub-stripes related to this subset of sub-chunks satisfy $j \in \{c(t,s) \mid 0 \le t < m\}$, i.e., having the same value range of the sub-chunk indices. Therefore, $SPCM_i$ of size $m^b \times m^b$ can be partitioned into m^{b-1} small matrices of size $m \times m$, each corresponding to one sub-chunk subset and its sub-stripe subset.

These small matrices have identical elements and cover all non-zero elements of $SPCM_i$. We denote each of these small matrices as elementary-parity-check-matrix (EPCM), and therefore $SPCM_i$ consists of m^{b-1} $EPCM_i$. An $EPCM_0$ in the case of $m = 4$ is as follows:

$$EPCM_0 = \begin{bmatrix} \overrightarrow{\lambda_0} & \overrightarrow{\lambda_1} & \overrightarrow{\lambda_2} & \overrightarrow{\lambda_3} \\ & \overrightarrow{\lambda_1} & & \\ & & \overrightarrow{\lambda_2} & \\ & & & \overrightarrow{\lambda_3} \end{bmatrix} \quad (1)$$

Corresponding to chunk partitioning rule in Section 4.1, concatenating SPCMs of chunks within group_s produces grouped-$SPCM_s$ ($GSPCM_s$). For each SPCM within group_s, there is an EPCM with the rows and columns indexed within the same value range $\{c(t,s) \mid 0 \le t < m\}$, and these EPCMs can be also concatenated together to produce grouped-$EPCM_s$ ($GEPCM_s$). Therefore, assuming group_s include m_s chunks, $GSPCM_s$ can be partitioned into m^{b-1} identical $GEPCM_s$, exhibiting the matrix-block-repetition pattern.

A.6 The Parity Solving and Matrix Elimination

Here are the mathematical details of Section 5.3, and we also prove Theorem 4.1 along this process. The objective of matrix elimination is to eliminate *GSPCM* of other groups, such that computing each sub-chunks within group$_s$ only needs to invert $GSPCM_s$. As $GSPCM_s$ exhibits the matrix-block-repetition pattern, the inversion of $GSPCM_s$ is further reduced to the inversion of $GEPCM_s$.

Details of matrix elimination. Define $\vec{\lambda}^{[h,t)} = [\lambda^h, \lambda^{h+1}, \ldots, \lambda^{t-1}]^T$, i.e., performing head and tail truncation on the original $\vec{\lambda}$. Then we define that substituting each element $\vec{\lambda}$ by $\vec{\lambda}^{[h,t)}$ in $GSPCM_s$ and $GEPCM_s$ produces $GSPCM_s^{[h,t)}$ and $GEPCM_s^{[h,t)}$, respectively. Now we start to separate $GSPCM_0$ from the other GSPCMs and eliminate $GSPCM_0$. Note that we may logically permute matrix rows between steps to facilitate the demonstration, which has no actual impacts on computations. Without loss of generality, we assume that the m chunks to be computed are parity chunks with indices $\{p_1, \ldots, p_m\}$, while any m other chunks follow the same procedure to perform divide-and-conquer solving. We start with the entire coefficient matrix $[P_{p_1}, \ldots, P_{p_m}]$, corresponding to parity chunks $\{C_{p_1}, \ldots, C_{p_m}\}$. The matrix elimination consists of two key operations: *row transformation* and *sub-chunk transformation*. Equation 2 demonstrates the row transformation process, with the row transformation matrix RT_0 described later:

$$
\begin{aligned}
&[P_{p_1}, \ldots, P_{p_m}] \\
&= [GSPCM_0^{[0,m)} \,|\, GSPCM_1^{[0,m)} \,|\, \cdots \,|\, GSPCM_{b-1}^{[0,m)}] \\
&\quad \text{(Group } SPCM\text{s within each group}_s \ (0 \le s \le b-1).) \\
&= \left[\begin{array}{c|c|c|c} GSPCM_0^{[0,m_0)} & GSPCM_1^{[0,m_0)} & \cdots & GSPCM_{b-1}^{[0,m_0)} \\ GSPCM_0^{[m_0,m)} & GSPCM_1^{[m_0,m)} & \cdots & GSPCM_{b-1}^{[m_0,m)} \end{array}\right] \\
&\quad \text{(Define } m_s \text{ as the number of parity chunks in group}_s.) \\
&= \left[\begin{array}{c|c} GSPCM_0^{[0,m_0)} & GSPCM_{[1,b)}^{[0,m_0)} \\ GSPCM_0^{[m_0,m)} & GSPCM_{[1,b)}^{[m_0,m)} \end{array}\right] \\
&\quad \text{(Focus on only group}_0.) \\
&\Rightarrow \left[\begin{array}{c|c} GSPCM_0^{[0,m_0)} & GSPCM_{[1,b)}^{[0,m_0)} \\ 0 & RT_0 \times GSPCM_{[1,b)}^{[0,m_0)} + GSPCM_{[1,b)}^{[m_0,m)} \end{array}\right] \\
&\quad \text{(Eliminate } GSPCM_0^{[m_0,m)} \text{ using } RT_0 \times GSPCM_0^{[0,m_0)}. \\
&\quad RT_0 \text{ represents row transformations to eliminate group}_0.) \\
&= \left[\begin{array}{c|c} GSPCM_0^{[0,m_0)} & GSPCM_{[1,b)}^{[0,m_0)} \\ 0 & A \end{array}\right] \\
&\quad \text{(Denote the bottom-right part by } A.)
\end{aligned}
\tag{2}
$$

Note 1 for proof of Theorem 4.1. Now we have decomposed the coefficient matrix $[P_{p_1}, \ldots, P_{p_m}]$ into $GSPCM_0^{[0,m_0)}$ and A, such that the coefficient matrix is invertible if and only if $GSPCM_0^{[0,m_0)}$ and A are invertible. The remaining of the proof lies in proving that A is equivalent to $GSPCM_{[1,b)}^{[0,m-m_0)}$ regarding the invertibility (See Note 2). If we can establish this equivalence, then we can continuously decompose $GSPCM_s^{[0,m_s)}$ from $GSPCM_{[1,b)}^{[0,m-m_0)}$, such that $[P_{p_1}, \ldots, P_{p_m}]$ is invertible if and only if all $GSPCM_s^{[0,m)}$ are invertible, matching Theorem 4.1.

Row transformation matrix. Since $GSPCM_0$ can be partitioned into m^{b-1} identical $GEPCM_s$, RT_0 can also be partitioned into m^{b-1} identical elementary-RT_0 (ERT_0), each eliminating $GEPCM_0^{[m_0,m)}$ using $ERT_0 \times GEPCM_0^{[0,m_0)}$, i.e., $ERT_0 \times GEPCM_0^{[0,m_0)} = GEPCM_0^{[m_0,m)}$. According to linear algebra, $ERT_0 = GEPCM_0^{[m_0,m)} \times (GEPCM_0^{[0,m_0)})^{-1}$. Each ERT_0 is of size $(m-m_0)m \times mm_0$ and one can verify that it has at most $(mm_0 + m - m_0)m_0(m-m_0)$ non-zero coefficients.

Given the original parity-check matrix equation:

$$
\begin{aligned}
&[P_{p_1}, \ldots, P_{p_m}] \times \{\text{all parity chunks}\} = \vec{R} \\
&\text{i.e., } GSPCM_{[0,b)}^{[0,m)} \times \{\text{parity chunks in group}_{[0,b)}\} = \vec{R}
\end{aligned}
$$

RT_0 also transforms the intermediate results $\vec{R}$ (see §5.3) of size ml to $\vec{R}'$ of size $(m-m_0)l$. That is,

$$
A \times \{\text{parity chunks in group}_{[1,b)}\} = \vec{R}' \tag{3}
$$

where $A = RT_0 \times GSPCM_{[1,r)}^{[0,m_0)} + GSPCM_{[1,r)}^{[m_0,m)}$.

Sub-chunk transformations. To maintain the original coding coefficients of the remaining parity sub-chunks (i.e., to transform A back to $GSPCM_{[1,b)}^{[0,m-m_0)}$), we adopt sub-chunk transformations (ST_0) such that

$$
GSPCM_{[1,b)}^{[0,m-m_0)} \times ST_0 \times \{\text{parity chunks in group}_{[1,b)}\} = \vec{R}' \tag{4}
$$

Comparing Equation (3) and (4), we have

$$
RT_0 \times GSPCM_{[1,b)}^{[0,m_0)} + GSPCM_{[1,b)}^{[m_0,m)} = GSPCM_{[1,b)}^{[0,m-m_0)} \times ST_0 \tag{5}
$$

To determine the required ST_0, we analyze the left side of Equation 5 at the granularity of ERT_0. As each ERT_0 performs the row transformation on a $GEPCM_0$, it also transforms coefficients of parity sub-chunks within other groups. We take group$_1$ for an example. In $GSPCM_1$, we select the m rows corresponding to ERT_0 and select the m sub-chunks with the same indices as these rows. One can verify that the coefficient matrix of the m selected sub-chunks within the m selected rows is a $m \times m$ diagonal matrix, whose elements are identical coefficient vectors. The coefficient vector can be

determined as in Appendix A.3, and here we assume its value to be $\vec{\lambda}$, such that coefficient matrix, denoted by $Diag_\lambda$, can be written as follows:

$$Diag_\lambda = \begin{bmatrix} \vec{\lambda} & & & \\ & \vec{\lambda} & & \\ & & \ddots & \\ & & & \vec{\lambda} \end{bmatrix}$$

Therefore, for the m selected sub-chunks and m selected rows, Equation 5 turns into

$$ERT_0 \times Diag_\lambda^{[0,m_0)} + Diag_\lambda^{[m_0,m)} = Diag_\lambda^{[0,m-m_0)} \times EST_{0,\lambda} \tag{6}$$

where $EST_{0,\lambda}$ denotes an elementary-ST_0 corresponding to $Diag_\lambda$. Recall that $ERT_0 \times GEPCM_0^{[0,m_0)} = GEPCM_0^{[m_0,m)}$, we define ERT_0^t $(0 \le t < m-m_0)$ to be the sub-matrix of ERT_0 that satisfies

$$\begin{aligned} &ERT_0^t \times GEPCM_0^{[0,m_0)} = GEPCM_0^{[m_0+t,m_0+t+1)} \\ \Rightarrow &ERT_0^0 \times GEPCM_0^{[0,m_0)} = GEPCM_0^{[m_0,m_0+1)} \\ \Rightarrow &ERT_0^0 \times GEPCM_0^{[t,m_0+t)} = GEPCM_0^{[m_0+t,m_0+t+1)} \\ \Rightarrow &ERT_0^t \times GEPCM_0^{[0,m_0)} = ERT_0^0 \times GEPCM_0^{[t,m_0+t)} \end{aligned}$$

That is to say, performing ERT_0^t on $\vec{\lambda}^{[0,m_0)}$ is equivalent to performing ERT_0^0 on $\vec{\lambda}^{[t,m_0+t)}$. Therefore, we have

$$\begin{aligned} &ERT_0^t \times Diag_\lambda^{[0,m_0)} + Diag_\lambda^{[m_0+t,m_0+t+1)} \\ =&ERT_0^0 \times Diag_\lambda^{[t,m_0+t)} + Diag_\lambda^{[m_0+t,m_0+t+1)} \\ =&(ERT_0^0 \times Diag_\lambda^{[0,m_0)} + Diag_\lambda^{[m_0,m_0+1)}) \times Diag_\lambda^{[t,t+1)} \\ =&Diag_\lambda^{[t,t+1)} \times (ERT_0^0 \times Diag_\lambda^{[0,m_0)} + Diag_\lambda^{[m_0,m_0+1)}) \end{aligned}$$

traversing t from 0 to $m-m_0$, we have

$$\begin{aligned} &ERT_0 \times Diag_\lambda^{[0,m_0)} + Diag_\lambda^{[m_0,m)} \\ =&Diag_\lambda^{[0,m-m_0)} \times (ERT_0^0 \times Diag_\lambda^{[0,m_0)} + Diag_\lambda^{[m_0,m_0+1)}) \end{aligned} \tag{7}$$

comparing Equation 6 and Equation 7 we have $EST_{0,\lambda} = ERT_0^0 \times Diag_\lambda^{[0,m_0)} + Diag_\lambda^{[m_0,m_0+1)}$. And sub-chunk transformation ST_0 is determined by traversing all ERT_0 and getting all $EST_{0,\lambda}$. Now we successfully eliminate group$_0$ and maintain the original coefficient matrix (with necessary truncation) of sub-chunks in other groups. Each $EST_{0,\lambda}$ is of size $m \times m$, and one can verify that $(EST_{0,\lambda})^{-1}$ has at most $mm_0 + m - m_0$ non-zero elements.

Note 2 for proof of Theorem 4.1. Now we have given the form of $EST_{0,\lambda}$. Recall that $A = GSPCM_{[1,b)}^{[0,m-m_0)} \times ST_0$, then A is equavilent to $GSPCM_{[1,b)}^{[0,m-m_0)}$ regarding invertibility as long as each $EST_{0,\lambda}$ is invertible. One can refer to [49] for the proof of a similar problem in the MSR-code version.

A.7 The Computational Overhead of Parity Solving and Matrix Elimination

We count the number of multiplications imposed by eliminating group$_s$. Denote the number of remaining chunks after eliminating group$_s$ as r_s. Firstly, each of m^{b-1} ERT_s imposes at most $(mm_s + m - m_s)m_s r_s$ multiplications on $\vec{R}$, totaling $(mm_s + m - m_s)m_s r_s m^{b-1}$. Secondly, eliminating group$_s$ introduces sub-chunk transformations ESTs that need to be inverted, imposing at most $mm_s + m - m_s$ multiplications for each of $r_s m^{b-1}$ ESTs (m^{b-1} ESTs for each of r_0 remaining chunks), totaling $(mm_s + m - m_s)r_s m^{b-1}$. Thirdly, when the remaining $r_s m^b$ chunks are solved, substituting them into the equation of group$_s$ takes m_s multiplications for each sub-chunk, totaling $m_s r_s m^b$. Finally, we need to solve the following equation for group$_s$:

$$GSPCM_s^{[0,m_s)} \times \{\text{parity chunks in group}_s\} = \vec{U}$$

where $\vec{U}$ is the remaining intermediate results after substituting solved sub-chunks. One can verify that each $(GEPCM_s^{[0,m_s)})^{-1}$ has at most $(mm_s + m - m_s)(m_s)^2$ non-zero coefficients, totaling $(mm_s + m - m_s)(m_s)^2 m^{b-1}$ for $(GSPCM_s^{[0,m_s)})^{-1}$, and the number of multiplications for solving this equation does not exceed this number. Note that $\sum m_s = m$, $r_s + m_s \le m$, and $mm_s + m - m_s \le m^2$. It can be easily verified that the total multiplication number is $O(m^{b+3}) = O(m^3\alpha)$.

The LogDrive: Composable Durability for Cloud-Based Shared Logs

Gardner Vickers, Lucas Bradstreet, Mahesh Balakrishnan†, Prince Mahajan, David Mao†
Xavier Léauté, Ismael Juma, Nikhil Bhatia, Jack Vanlightly, Prateek Jindal, Sumit Arrawatia
Andrew Grant, Dhruvil Shah, Dimitar Dimitrov, Gaurav Badoni, Shimiao Zhang, Yang Yu
Confluent

Abstract

A growing class of systems leverages inexpensive cloud object storage as a disaggregated data plane, offloading durability and scaling to the cloud. Storing the metadata for such systems in a cloud database is too expensive; while self-managed databases are complex and fragile. Conflux provides a third option by storing a shared log on cloud storage and using it to replicate state across VMs. A key innovation in Conflux is the separation of durability from sequencing. Durability is provided solely by the novel LogDrive abstraction: a simple, low-level substrate that can be layered above arbitrary cloud storage, striped for throughput, and – unlike shared logs – composed via quorum-based replication. Sequencing is provided by the AtomicLog, which implements a conventional shared log over any LogDrive. This design allows Conflux to replicate arbitrary state machines while using RAID-like compositions of cloud storage for durability. Conflux is deployed in production at Confluent as the metadata service for an S3-based publish-subscribe system called K2. We show that Conflux can run on diverse storage services (e.g., DynamoDB, S3, S3Express) with just a few hundred extra LOC per service; as well as augment the durability of these services (e.g., with synchronous cross-region replication). Conflux unlocks new cost vs. latency trade-offs over cloud storage: for our representative workloads and latency SLAs (compared to using DynamoDB directly) Conflux-over-DynamoDB slashes metadata cost by 10X and overall cost by 3X.

1 Introduction

Cloud storage services such as S3 [8] offer scalable, elastic, and fault-tolerant data storage at inexpensive prices. As a result, stateful distributed services with rich APIs – ranging from analytical databases to publish-subscribe systems – are increasingly designed as stateless / soft-state layers over disaggregated cloud storage. In such services, the data plane resides directly on cloud storage (e.g., S3), while metadata (e.g., an index) is stored on a separate metadata service; for example, Snowflake stores data on S3 and metadata in FoundationDB [45]. The resulting services have data planes that

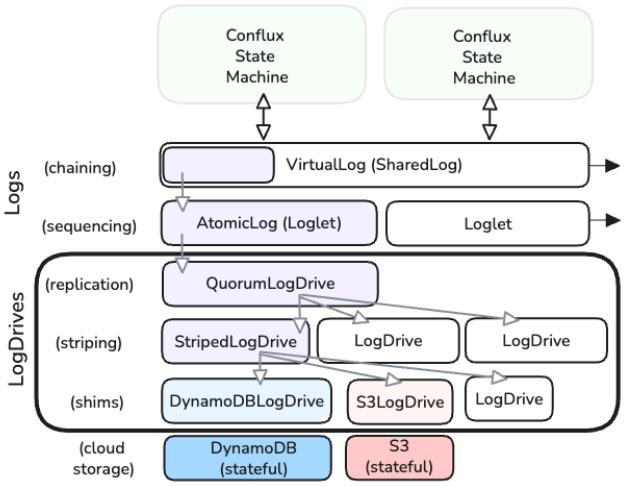


Figure 1. *LogDrives are **easy to build** as shims over cloud storage, and via RAID-like composition; they are **useful** for implementing shared logs (and SMR).*

are cheaper and simpler to build and operate, which translates to cost savings and higher reliability for end-customers.

At Confluent, we set out to build a new publish-subscribe service called K2 directly over S3, leveraging its inexpensive large writes to enable a high-throughput, low-cost product (called Freight [11]) for customers storing bulk data (e.g., observability traces) at sub-second latencies. In the K2 design, data is written directly to S3 in large, batched blobs; and the unit of metadata is a fine-grained index (an ordered list of pointers for each pub-sub topic). To store this metadata, we needed a durable yet easy-to-operate database that could support small writes (and reads) at low cost (but not necessarily low latency); and expose a convenient API for storing per-topic indexes.

Unfortunately, we found our options limited:

- Operating a stateful distributed database to store metadata is complex and difficult, typically requiring a collection of individual machines (e.g., EC2 VMs deployed via K8s [5]) to ensure durability in the presence of failures. Existing options such as FoundationDB [45], KRaft [6], ZooKeeper [37], and etcd [4] are known to be difficult to operate in production [17,

†These authors are no longer at Confluent; their contributions to this work were made while at Confluent.

28]. This complexity is exacerbated by the hybrid nature of the overall system, since the data and metadata planes can each fail in different ways; as well as the outsized impact of metadata loss on overall durability.
- Conversely, storing metadata in a cloud database (e.g., DynamoDB [32]) is simpler and safer, but prohibitively expensive for the small, frequent writes and reads found in metadata workloads. We estimated that a DynamoDB-based metadata layer would incur roughly 75% of K2's overall cost. Further, new and cheaper options can emerge and prices can fluctuate (e.g., S3Express [9] launched in 2024 and changed pricing in 2025 [12]), necessitating difficult, risky migrations between storage services to optimize cost.

In this paper, we present a new metadata store called Conflux. Conflux uses cloud storage as a shared log for State Machine Replication [49]. We apply research on shared logs [2, 18, 19, 22, 31, 44, 52] to construct a replicated state machine over a total order of commands, which in turn is stored on a shared log implemented directly on cloud storage. In this design, the shared log is a consensus engine that delegates durability entirely to cloud storage; drives down write costs by batching log-structured writes; and lowers read cost via materialized local views. By storing the shared log on passive cloud storage, Conflux aims for the benefits of RAID-like [47] composition over different storage services: striping for throughput; replication for higher availability and durability; chaining for dynamic switching of new writes; as well as tiering for migrating older writes.

However, we encountered an unexpected challenge in realizing our goal of delegating durability to arbitrary, diverse, and composable cloud storage. While shared logs can be composed via chaining and striping [18], they cannot be easily or efficiently composed via replication. The reason is the narrow, opinionated *append* interface, which allows each shared log to decide the timestamps of its new entries: when an entry is appended to multiple logs, each one can assign a different, potentially conflicting timestamp. As a result, we were unable to compose single-region DynamoDB logs into a multi-region log; or one-AZ S3Express logs into a multi-AZ or multi-region log. The absence of replication severely hobbles the value of composition (imagine RAID without mirroring!). Ironically, the property that makes shared logs compelling – the delegation of sequencing and durability to a single abstraction – also hinders their composition.

One solution is to *separate sequencing from durability*: we can introduce a low-level abstraction for durability (such as a shared address space) that is easy to compose via layering, striping, and replication; and layer the shared log above it as a shim, with a soft-state counter (as in Corfu [19]) for tracking the tail. We borrow this principle from other areas of systems: e.g., filesystems layered over block devices composed via RAID [47]. Unfortunately, this moves the burden to recovery: if the shared log layer experiences failures (e.g., if the counter appears to fail), recovering its soft state – specifically, the linearizable tail – from an address space can be difficult and expensive, particularly in a distributed setting with partial failures and concurrent writes. Adding special APIs to the address space (e.g., to fence concurrent writes and return the first unwritten entry, as in Corfu's custom storage servers) hinders its implementation over passive cloud storage and easy composition. Required is an abstraction for shared log durability that is composable, yet enables recovery.

Accordingly, we propose a new abstraction called the LogDrive. Like a conventional address space, the LogDrive exposes a numbered, random-write address space of durable registers. However, the LogDrive expects to be written (almost) in sequence: accordingly, it provides an extra *weakTail* API. The LogDrive has unusually weak semantics: e.g., individual addresses are only linearizable as long as a single value can possibly be written to them, and hence weaker than Write-Once, Conditional, or Atomic Registers [16, 24]. In addition, the *weakTail* call (as its name implies) is not necessarily linearizable; instead, it is only required to correspond to a deterministic function over a non-atomic, unordered scan. The LogDrive abstraction is *easy to implement*: the compact API and its weak semantics allow layering over existing cloud storage APIs (even without write-once or conditional-write support); and composition via single-round quorums and striping.

At the same time, the LogDrive abstraction is *powerful and useful*. We describe a new shared log design called the AtomicLog that combines the LogDrive with a soft-state counter and a windowed write discipline. The result is a full-fledged, linearizable shared log, which in turn can be used for State Machine Replication. By composing LogDrives via replication, we can augment the existing intra-region durability of cloud stores with cross-region quorums. For example, we can run a single DynamoDBLogDrive per region; and then construct a QuorumLogDrive that writes to two out of three regions before responding.

Conflux breaks new ground in a number of ways. The LogDrive (in concert with the AtomicLog) is the first solution for composing shared logs via replication; prior work such as Delos [18] stopped at composition via striping and concatenation. Earlier systems [7, 18, 19, 31] have explored disaggregated shared logs implemented over collections of custom servers, whereas Conflux uses common-denominator cloud storage APIs. Prior work such as LazyLog [44] *temporally* separates ordering and durability within a shared log for performance, whereas Conflux *spatially* separates these concerns into different layers for composability. Scalog [31] separates sequencing from durability for performance, but its sequencing layer is a stateful SMR-based metadata layer in itself.

In this paper, we make the following contributions:

- We describe the design and implementation of Conflux, the metadata service for a new S3-based publish-subscribe system called K2. Conflux uses cloud storage as a shared log, reducing the metadata cost by 10X compared to using DynamoDB directly, within a p99 write latency of 130ms; and the overall K2 cost by 3X.
- We propose a novel LogDrive abstraction that allows Conflux to easily run on diverse cloud stores (S3, DynamoDB, S3Express) and switch dynamically between them. New LogDrives are easy to build: for example, a single engineer was able to build a production-quality DynamoDBLogDrive in just a week.
- The LogDrive can be composed via replication (as well as striping): we show that this lets us construct multi-region quorums over DynamoDB, as well as multi-AZ quorums over S3Express.

2 Motivation

At Confluent, our goal was to build a Kafka [40] publish-subscribe (or pub-sub) implementation directly over cloud storage. Kafka is the *de facto* API for modern pub-sub systems, with thousands of production use cases. Applications can publish data to a topic and fetch data from it (in practice, Kafka refers to a topic as a partition and a bundle of partitions as a topic; but we match terminology with the pub-sub literature [33] by referring to a topic as a single total order). Existing implementations of Kafka are typically complex distributed systems that run directly on hardware for low latency workloads, with a resulting operational cost that is passed on to customers.

Our goal in building a new Kafka implementation was two-fold. First, we wanted to exploit high-latency but low-cost cloud storage to offer an inexpensive, bulk variant of Kafka to our customers (called Confluent Freight [11]). In particular, S3 provides cross-AZ durability for writes at a fixed cost regardless of payload size (in contrast to EC2 cross-AZ networking, which is charged per-byte), creating an opportunity to slash cost via big writes. Second, we hoped to get the ancillary benefits of running stateless logic above cloud storage, such as simpler operations and better reliability.

In this Kafka implementation (called K2), clients connect to stateless brokers (i.e., servers that implement the Kafka protocol). To publish data to a topic, a client sends it to a broker; once the broker has a large enough batch that typically spans writes to multiple topics, it issues a single write to S3 for the batch and then updates the metadata for all those topics. In this architecture, much of the heavy lifting is done by the metadata layer. Our default option for storing metadata was to build or buy a distributed database, deploy it on cloud VMs using K8s, and operate it with high reliability (e.g., matching Snowflake's use of FoundationDB [45]). However, this option would require a massive investment in time and resources, and considerably delay our time to market. Could we do better by using cloud storage for metadata as well?

Why not store Metadata directly in a cloud DB? Cloud databases like DynamoDB are powerful, reliable, and capable of handling our metadata workload: each K2 topic's metadata can simply be a key in DynamoDB. However, existing databases turned out to be prohibitively expensive: for example, our estimates suggested that we would spend 75% of our cloud cost just on metadata storage if we used DynamoDB. One reason for high cost is that the metadata workload consists of small writes to different keys (e.g., if an K2 broker combined data for 10 topics in a single S3 batch, we'd have to issue 10 separate writes to the metadata layer). A different concern relates to reads; metadata is often accessed repeatedly as clients and background processes replay the topic state. In the absence of (potentially complex) caching infrastructure, these accesses blow up read costs.

In addition, we needed fast reaction times to changes in cloud pricing and features. Unlike the data plane, which requires a minimal put/get interface from cloud storage, metadata can benefit from rich APIs for indexing. However, using such APIs would limit the portability of our code to other cloud providers, or to other storage options at the same provider if prices changed; in addition, adding new metadata APIs would require careful implementation on each backend.

Could we somehow obtain the benefits of storing metadata in cloud storage (i.e., delegating durability entirely to the cloud provider); while at the same time keep our cost low?

2.1 The Cloud is the Log!

In a shared log design, we execute multiple copies of a database on individual servers; and keep these copies synchronized via an external shared log. In principle, the log is the source of consistency as well as durability (storing the linearizable order of commands); the database is simply a materialized cache over the log. This clean, API-driven separation between the soft-state database and the durable shared log allows us to easily place the latter on cloud storage, entirely delegating durability to the cloud.

Using cloud storage as a shared log eliminates costly small writes: we can batch metadata updates for multiple topics and issue them in a single write to the shared log. Since each database server effectively has a full, strongly consistent copy of the database, we can serve reads with low cost (and low latency) without unnecessarily accessing cloud storage. In addition, we access cloud storage through a narrow API (the API of the shared log does not change over time even as we add new APIs to our database).

Unfortunately, we ran into a fundamental limitation of the shared log abstraction's *append* interface: composition via replication. Consider the task of composing a replicated shared log R over two individual shared logs A and B. Trivially, we could implement the *append* operation on R by

```
interface LogDrive {
  void write(long address, ByteBuf payload);
  ByteBuf read(long address);
  TailDesc weakTail(int K);
  struct TailDesc {
    long nonContiguousTail;
    Set<Long> holes;
  }
}
```

Figure 2. *The LogDrive API*

forwarding it to both A and B; however, there is no guarantee in the shared log API that we will obtain the same timestamp from both (such a guarantee would be difficult to obtain, given that the implementation could choose to embed internal protocol details such as membership epochs in its timestamps [7]). Further, if one of the two appends is delayed, retrying it incurs the risk of duplicate entries with different timestamps. In addition, concurrency poses a challenge: we have to issue the appends one-at-a-time, else they could arrive at the logs in different orders.

These issues can be solved if R entirely ignores the timestamps / ordering provided by A or B and instead embeds application-level timestamps into entry headers. However, playback now requires re-ordering of entries, negating the simplicity benefit of the shared log abstraction. In addition, the replicated log needs a source of these external timestamps; using an explicit sequencer turns composition into a complex, heavy-weight layer (in contrast to a light-weight, RAID-like shim that can be nested arbitrarily).

Why is composition via replication so important? Our use cases for K2 spanned the durability gamut: some applications were okay with data loss if a single AZ failed; others required multi-AZ durability within a single region; while some mission-critical use cases required zero data loss even if an entire region failed. These requirements complicated our path to using existing services like S3, DynamoDB, or S3Express for storing metadata; since S3 and DynamoDB do not currently support synchronous cross-region replication; while S3Express does not support cross-AZ replication. The ability to compose shared logs via replication would allow us to leverage the existing machinery of these services within regions (in the case of DynamoDB or S3) and AZs (with S3Express) while augmenting it with cross-region or cross-AZ replication, respectively. Even if these systems provide synchronous cross-region replication as an option (e.g., DynamoDB enabled support in mid-2025), we would have more flexibility in a compositional layer (e.g., to use different policies or placements) and retain the ability to migrate between clouds or even span clouds.

3 The LogDrive Abstraction

The LogDrive is a novel abstraction that can be easily implemented on cloud storage and composed via replication and striping. We now describe the LogDrive API and its various implementations. In later Sections, we explain the utility of the LogDrive for State Machine Replication.

3.1 The LogDrive Abstraction

The LogDrive is an address space of registers with a random *read*/*write* interface (see Figure 2). Addresses are initialized to an *unwritten* state; a *write* switches them to a *written* state. In Figure 3, we show the anatomy of a LogDrive: the *contiguous tail* ($\mathcal{T}$) is the first unwritten address before which all addresses are written; the *non-contiguous tail* ($\mathcal{N}$) is the first unwritten address after which all addresses are unwritten; and in between is a "swiss cheese" area of intermixed written and unwritten addresses. We call the unwritten addresses less than $\mathcal{N}$ the *hole-set* ($\mathcal{H}$).

Two features distinguish the LogDrive from a conventional address space. The first is the *weakTail* API, which returns a tuple with ($\mathcal{N}$, $\mathcal{H}$): the non-contiguous tail and the hole-set. Note that the caller can compute the contiguous tail $\mathcal{T}$ from the returned tuple: it is guaranteed to be the lowest address in $\mathcal{H}$, or $\mathcal{N}$ if $\mathcal{H}$ is empty. Informally, we will refer to $\mathcal{T}$ as returned by *weakTail* even though it is a derived value. Importantly, the application provides a parameter K to *weakTail*, which is the maximum possible distance between $\mathcal{T}$ and $\mathcal{N}$. The application typically knows and ensures this bound by using a windowing discipline to write to the LogDrive. As we see later, this bound is critical for implementing *weakTail* efficiently, giving the implementation license to ignore the prefix of the address space while determining $\mathcal{H}$. As long as the K bound is correct, the *weakTail* response entirely characterizes the written/unwritten status of each address in the LogDrive. Note that $|\mathcal{H}|$ is bounded by K: this API is only practical for small K.

The second distinguishing characteristic of the LogDrive is the unusually weak semantics it provides, on two dimensions. First, individual addresses are *single-value registers*: i.e., they are only linearizable for *write*/*read* operations if a single value is written to them (though the same value can be written multiple times). We rely on the application to never write multiple values to an address. In this sense, they are similar to SWMR registers [41] or write-once registers [34] but with significantly weaker semantics. Second, the *weakTail* is not linearizable; instead, it is merely equivalent to a deterministic function over an unordered, non-atomic scan (e.g., executed via parallel point reads) on the address space. This guarantee has other parallels in the literature: e.g., non-atomic collects [15] on SWMR registers and interval-linearizability [27].

Lemma (**[LD.1]** Tail-Range Guarantee)**.** *Let a weakTail operation run over a linearization span* $[t_{\text{start}}, t_{\text{stop}}]$*, such that*

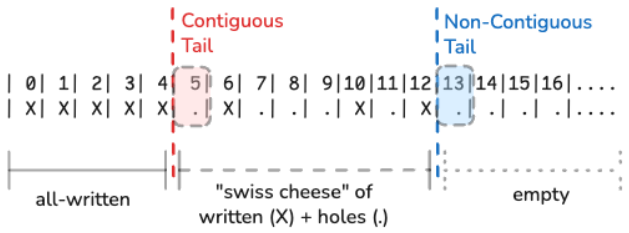


Figure 3. *The Anatomy of a LogDrive.*

$\mathcal{T}_{\text{start}}$ and $\mathcal{T}_{\text{stop}}$ denote the actual contiguous tail at those times. Then the value $\mathcal{T}_{\text{obs}}$ returned by weakTail satisfies:

$$\mathcal{T}_{\text{obs}} \in [\mathcal{T}_{\text{start}}, \mathcal{T}_{\text{stop}}].$$

Proof sketch. Consider a *weakTail* implementation that literally scans for holes by issuing point reads in parallel to the entire address space. During the scan, different addresses are sampled at different times in $[t_{\text{start}}, t_{\text{stop}}]$. Suppose $\mathcal{T}_{\text{obs}} < \mathcal{T}_{\text{start}}$. Then the scan must have observed address $\mathcal{T}_{\text{obs}}$ unwritten at some time $\tau \geq t_{\text{start}}$, which contradicts the definition of $\mathcal{T}_{\text{start}}$ as the first unwritten slot at t_{start}. Similarly, if $\mathcal{T}_{\text{obs}} > \mathcal{T}_{\text{stop}}$, then the scan must have observed all addresses $< \mathcal{T}_{\text{obs}}$ as written strictly before t_{stop}. But then $\mathcal{T}_{\text{stop}}$ cannot be the first unwritten address at t_{stop}, a contradiction. Thus $\mathcal{T}_{\text{obs}}$ must lie within $[\mathcal{T}_{\text{start}}, \mathcal{T}_{\text{stop}}]$. *Full Proof in Appendix.* □

However $\mathcal{T}_{\text{obs}}$ is not necessarily linearizable, since writes on the LogDrive can linearize out of address order: e.g., let us say we write to 1 and then 0 (causing the contiguous tail to jump from 0 to 2). A *weakTail* concurrent with both writes might return 1 if it first reads 1 as unwritten and then 0 as written. But this execution does not correspond to any total order of the two writes and the *weakTail*.

Lemma ([LD.2] Window Scan Property**).** *To satisfy weakTail semantics, it is sufficient for an implementation to first determine the non-contiguous tail linearizably; and then issue an unordered, non-atomic scan to the K-sized window of preceding addresses. The value returned by this window scan is equal to the value returned by some unordered, non-atomic scan over the entire address space.*

Proof sketch. Reading the non-contiguous tail $\mathcal{N}$ linearizably tells us that, at some instant t^* during the operation, all written addresses lie in $[0, \mathcal{N}^*)$ and all addresses $\geq \mathcal{N}^*$ are still unwritten. Under the K-window discipline, there can be at most K holes before $\mathcal{N}^*$, so the hole-set at t^* must lie entirely inside $[\mathcal{N}^* - K, \mathcal{N}^*]$, and every address $< \mathcal{N}^* - K$ is definitely written. We can therefore construct an equivalent full, unordered, non-atomic scan by (i) using the issued window reads in $[\mathcal{N}^* - K, \mathcal{N}^*]$, and (ii) using hypothetical reads issued at time t^* that returned written below the window and unwritten above it. The returned non-contiguous tail and hole-set from this hypothetical full scan exactly match those found by the window scan. *Full Proof in Appendix.* □

3.2 Primitive LogDrive Implementations

The compact API and weak semantics of the LogDrive simplify its implementation on existing cloud storage. Single-value registers are trivial to implement over any existing key-value store (even ones that do not provide conditional put operations, such as S3 prior to 2024) or even shared disks.

As a result, the only complexity in implementing a LogDrive on an existing cloud store has to do with the *weakTail* call to find its contiguous tail. In practice, we find that existing cloud stores give us efficient ways of finding the non-contiguous tail; and can scan back K entries to find the contiguous tail, as we describe:

S3: Each LogDrive is assigned a particular prefix in an S3 bucket (the combination of bucket and prefix uniquely identifies the LogDrive). Within that prefix, we reverse-encode addresses as S3 keys (e.g., so that the 0th address is lexicographically last). S3 gives us the ability to list keys in lexicographic order. Accordingly, we can list the first K keys to obtain the last K written slots, which we then invert to return the list of holes in the last K slots. By Lemma LD.2 (Window Scan Property), this is sufficient to return a correct *weakTail* response.

DynamoDB: We use composite keys to encode LogDrive entries: each LogDrive to a particular partition key, and each address is encoded as the sort key. Unlike S3, we don't need to reverse-encode, since DynamoDB lets us query items in either lexicographical order. By Lemma LD.2 (Window Scan Property), this is sufficient to return a correct *weakTail* response.

S3Express: This storage service is nearly identical to S3, except for its single-AZ durability and some differences in API semantics. One difference is that we can no longer list keys in lexicographical order [10]. To get around this, we create a directory-like hierarchical structure on the S3Express key space. We then enumerate all entries at each level of this hierarchy to route to the non-contiguous tail; and then scan backwards K entries to find the contiguous tail. By Lemma LD.2 (Window Scan Property), this is sufficient to return a correct *weakTail* response.

3.3 The StripedLogDrive

Striping the LogDrive is straightforward for writes and reads: we implement RAID-0-style striping to map from *global* addresses (on the StripedLogDrive's address space) to *local* addresses on S individual stripes. Note that a stripe here is a LogDrive instance, implemented over a cloud store as described above.

To implement *weakTail* on the StripedLogDrive, we invoke it on all stripes in parallel and wait for all responses. We pass in K/S as the parameter to the per-stripe *weakTail* calls; we require K to be a multiple of S. We convert all returned local addresses to global addresses; and then pick the maximum

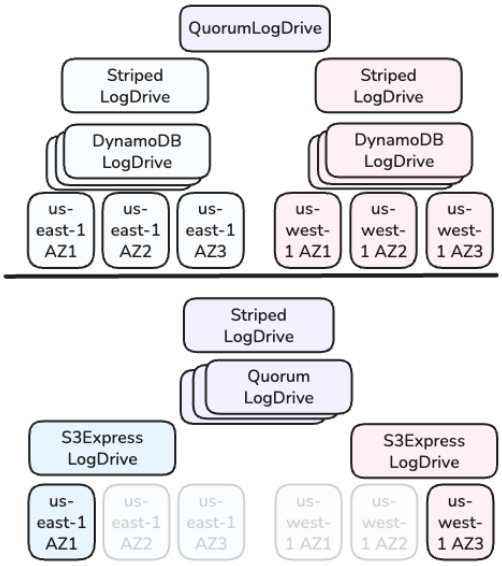


Figure 4. *Replication over Striping, similar to RAID-10 (Top); Striping over Replication, similar to RAID-01 (Bottom).*

returned global non-contiguous tail, along with the union of all the returned hole-sets.

Theorem ([SLD.1] Striped LogDrive Preserves *weakTail* Semantics)**.** *The weakTail implementation of the StripedLogDrive, described above, satisfies the LogDrive weakTail semantics: it is equivalent to the result of some unordered, non-atomic scan over the global address space within the linearization span.*

Proof sketch: The *weakTail* call on each stripe is equivalent to an unordered, non-atomic scan on its own local address space (corresponding to a stripe of the global address space). We can construct an unordered, non-atomic scan on the global address space by combining these per-stripe scans, translating each local address to its global equivalent. Picking the maximum non-contiguous tail returned across all stripes is equivalent to choosing the non-contiguous tail from that global scan; the union of all the returned hole-sets is equal to the hole-set on the global address space. *Full Proof in Appendix.* □

3.4 The QuorumLogDrive

The QuorumLogDrive (or QuorumDrive for short) replicates data across a quorum of N LogDrive replicas. Note that a replica here is a LogDrive instance, implemented over a cloud store as described above. The QuorumDrive is configured with a write quorum size Q_w, which can range from $[1, N]$.

On a *write*, the QuorumDrive forwards the command to all replicas. As an optimization, we can also hedge [29] by sending to a specific Q_w-sized subset and contact more replicas if responses are delayed. Once Q_w nodes ack, we can ack the write; as a result, writes ack in one round-trip to a write quorum. Since only one value can possibly be written to a single-value register, we avoid a two-round protocol like ABD [16] for Atomic Registers.

The *weakTail* call needs a conventional read quorum to intersect with write quorums: $Q_r = N - Q_w + 1$. At the same time, it has to "repair" writes [16] to enforce a write quorum for any observed partial writes. Practically, the *weakTail* accesses a quorum of size $Q_f = max(Q_w, Q_r)$ to invoke *weakTail* locally on each one. However, determining the global state from the responses is more involved than in striping: just because we observe a local hole at a single replica does not mean that corresponding global address was a hole at any time during the *weakTail*'s linearization span. The replica could have merely missed that write by being temporarily inaccessible.

Accordingly, we examine each slot to determine if it is a global hole. For each slot in the address space, we can determine the global written / unwritten status as follows.

1. We see that the slot is unwritten on Q_r of replicas. We mark it as globally unwritten.
2. We see that the slot is written on Q_w of replicas. We mark it as globally written.
3. We cannot determine if the slot is globally written or unwritten (i.e., we do not see Q_w written or Q_r unwritten replicas). This happens if we can only access Q_f replicas and we are unable to determine the slot's global status without the inaccessible replicas. In this case, we repair the write by copying over the value to Q_w replicas and mark the slot as globally written.

Based on our determination of the global status of each slot, we can then return a *weakTail* response.

Windowing discipline on the global address space has two implications for this algorithm. First, we can run the algorithm in $O(Q_f * K)$: we start from the highest returned non-contiguous tail and examine only the K global addresses prior to it, since everything prior to that is guaranteed (by the application, via the *weakTail* parameter) to be lower than the global contiguous tail, and hence globally written. Second, relaying K to the per-replica *weakTail* calls instructs them (in their own *weakTail* implementations) to ignore anything prior to N-K: even though they may have local holes at these addresses, the application (in this case, the QuorumDrive) knows these to be globally written.

Theorem ([QLD.1] QuorumLogDrive preserves *weakTail* semantics)**.** *The weakTail implementation of the QuorumLogDrive, described above, satisfies the LogDrive weakTail semantics: it is equivalent to the result of some unordered, non-atomic scan over the global address space within the linearization span.*

Proof sketch: A *weakTail* call issues a *weakTail* to a quorum of replica LogDrives. By the *weakTail* spec, each replica responds with a full (unordered, non-atomic) characterization of its address space. In effect, we observe the status of every address in each responding LogDrive at some (different) instant within the linearization span. Any slot that we mark

```
interface Loglet {
  Pair<LogPos, SealStatus> checkTail();
  LogPos append(ByteBuf buf); //throws if sealed
  void seal();
  LogPos prefixTrim(LogPos trimPos);
  ByteBuf readNext(LogPos min, LogPos max);
}
```

Figure 5. *The AtomicLog API (== Delos Loglet API [18]).*

as globally unwritten must have been globally unwritten at the start of the linearization span. Any slot that we mark as globally written must be globally written by the end of the linearization span. As a result, we can construct a full (unordered, non-atomic) scan of the entire global address space with a linearization point for each individual written or unwritten entry within the linearization span. *Full Proof in Appendix.* □

4 Implementing a Shared Log over the LogDrive: The AtomicLog

Thus far, we have established that the LogDrive abstraction is *easy to implement and compose*. We now show that the LogDrive is also a *useful* abstraction that can support a conventional shared log.

In particular, we implement the Loglet API from Delos [18] (see Figure 5) verbatim. The Loglet API supports linearizable *append*, *checkTail*, and *readNext* calls. It also provides a linearizable *seal* call, which ensures that subsequently linearized *append*s do not acknowledge successfully. The Loglet is required to be highly available for all calls except *append*. In particular, *checkTail* has to be highly available and linearizable with appends, effectively returning the first unwritten log position. As shown by Delos, this API can be layered under a virtualization layer that provides high availability for *append*s; and support efficient, safe, and highly available State Machine Replication in production systems. Accordingly, we start by describing the AtomicLog in this Section, which implements the Loglet API over a LogDrive; and then describe how we virtualize it for high availability.

The AtomicLog is a stateless client-side library that interacts with two components: the LogDrive and a shared, soft-state sequencer object. The sequencer supports a simple *acquireSlot* / *completeSlot* API. It does not see any data payloads and does not have to be durable or highly available. There can only be one sequencer in an AtomicLog instance; if it fails or reboots, the AtomicLog becomes permanently unavailable for appends. Accordingly, it can be implemented as an RPC service collocated on one of the AtomicLog clients.

To *append*, a client first calls *acquireSlot* on the sequencer to obtain the next slot; *writes* to the LogDrive; and then calls *completeSlot*; before acking the *append*. The state of the sequencer is a combination of a counter and a window of in-flight writes. If an incoming *acquireSlot* finds that the window is full — i.e., there are more than K slots between the contiguous tail and the non-contiguous tail – it blocks until the window opens up (i.e., a write completes that moves the contiguous tail moves forward). In turn, each *completeSlot* blocks until all prior slots are completed.

In practice, this protocol ensures that the AtomicLog writes to the LogDrive within a K-sized sliding window. For example, in Figure 3, the AtomicLog (configured with $K = 8$) has to block before writing to address 13 since address 5 has not been written yet. As a result, it enforces a bound of K on the distance between the contiguous and non-contiguous tails of the LogDrive.

In this protocol, the linearization point of an append is determined by the *completeSlot*. In other words, appends linearize in strict address order: an append returning position 5 must linearize after an append that returns position 4 and after an append returning position 6. As per the Loglet contract, the *append* is not highly available (e.g., the sequencer can fail; or clients can crash after acquiring a slot); we discuss soon how unavailability is handled at a higher layer. However, the Loglet API does require a highly available, linearizable *checkTail* that returns the first unwritten log position.

Implementing *checkTail* is trivial: we simply invoke *weakTail* on the underlying LogDrive. Surprisingly, the returned contiguous tail gives us a linearizable *checkTail* for the AtomicLog, even though the LogDrive's *weakTail* is not itself linearizable.

Theorem (**[AL.1]** *checkTail* Linearizability)**.** *The AtomicLog checkTail operation, implemented via a single weakTail on the underlying LogDrive, is linearizable with respect to append.*

Proof sketch: Appends linearize in strict address order. A *checkTail* invoked over interval $[t_{\text{start}}, t_{\text{stop}}]$ returns $\mathcal{T}_{\text{obs}} \in [\mathcal{T}_{\text{start}}, \mathcal{T}_{\text{stop}}]$ (since the enclosed *weakTail* satisfies these bounds, by Lemma LD.1). Placing the *checkTail* immediately after all appends to addresses $< \mathcal{T}_{\text{obs}}$ yields a total order consistent with real-time and with the returned value. Thus *checkTail* is linearizable wrt *append*. □

Intuitively, the LogDrive *write* / *weakTail* API is not linearizable since writes do not commit in address order, and hence it is not always possible to generate a candidate total order that satisfies returned values as well as a sequential specification; whereas *append* / *checkTail* is linearizable due to the blocking behavior of appends on prior unwritten slots.

Finally, we provide a highly available *seal* to satisfy the Loglet API. For this, the AtomicLog stores a seal bit in a designated location on cloud storage (e.g., as address 0 on a second LogDrive). Each *append* checks this bit after *completeSlot* and throws if the log is sealed, while *checkTail* checks it after the *weakTail* and returns the seal status. To *seal* the AtomicLog, we simply write to this location. As per Delos, the seal checks are not required to be atomic with the *append* or *checkTail*,

though they must not be linearized before [35]. We omit describing *readNext* and *prefixTrim* for lack of space.

Optimizing *checkTail*: The protocol described above can be inefficient since *weakTail* requires accessing cloud storage. We can drastically reduce the common-case cost and latency of *checkTail* by using a *fast-path checkTail* that goes to the sequencer rather than the LogDrive. **We only use the *weakTail* on the LogDrive as a *slow-path* once the log is sealed.** If the sequencer is unavailable, we first seal the log; and then switch to the slow-path.

We do have to be careful in this case to avoid having two sources of truth for the tail. For example, a slow-path response might see a tail X on the LogDrive; but a subsequent, non-overlapping fast-path *checkTail* invocation can return $X-1$ as the tail, resulting in a linearizability violation (for example, if the writer of entry $X-1$ went to sleep after writing to the LogDrive but before calling *completeSlot* on the sequencer).

To avoid such linearizability violations, the *checkTail* uses the fast-path sequencer-based tail check before accessing the seal bit; if the seal bit is set, the client retries the tail check on the slow-path. As a result, once the seal bit is set, all subsequently linearized *checkTail* operations return values from the slow-path.

Handling Failures via Virtualization As in Delos, we layer a VirtualLog above the AtomicLog to convert our Loglet (which does not provide high availability for appends) into a highly available shared log that can drive State Machine Replication (see Figure 6 for the full stack). Our VirtualLog is not novel: it is identical to the Delos design and implementation, storing membership (i.e., the mapping from the VirtualLog's address space to individual Loglets) in a conditional register on cloud storage. For each Loglet, we store the configuration of the corresponding AtomicLog instance (including the underlying LogDrive – or multiple layered LogDrives – as well as the endpoint of the sequencer). Above the VirtualLog, we use a conventional SMR layer similar to Delos, which stores durable snapshots in cloud storage; we describe this layer in the next section.

When the Loglet (i.e., the AtomicLog) becomes unresponsive for *append*s, we simply *seal* it; call *checkTail* to determine the switchover point; and then write a mapping to the conditional register so we can switch to a new Loglet (either another AtomicLog or an entirely different implementation) for new *append*s. Within the AtomicLog, there are two failure cases of interest. First, the sequencer can fail. Second, a client can fail after acquiring a slot (effectively making the sequencer unavailable, since *acquireSlots* can no longer complete). We treat both these cases identically; in both cases, the Loglet becomes unavailable for *append* calls. Our choices here are identical to those made in prior work (e.g., the Delos NativeLoglet).

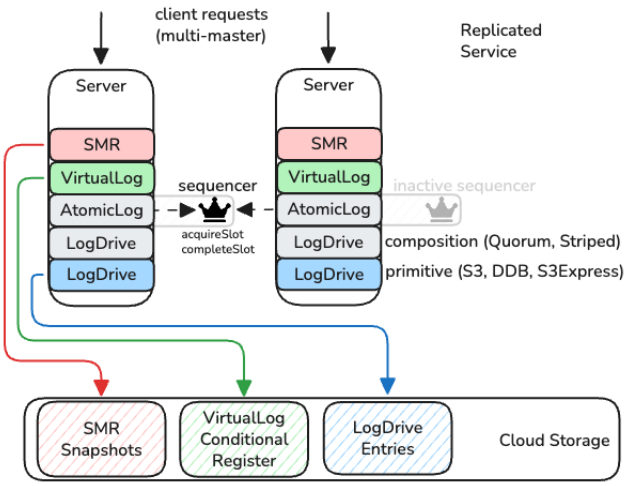


Figure 6. *SMR via LogDrives: each SMR-replicated server stores all its hard state in cloud storage. Servers do not interact via RPCs at any layer, with the exception of the soft-state sequencer in the AtomicLog.*

4.1 Discussion

Test-Driven Design: We created an extensive simulation testing framework for our shared log implementations. In this framework, we spin up a range of concurrent workloads against the shared log API in Figure 5; execute the workloads against a given target implementation of the API (e.g., the VirtualLog on an AtomicLog on a QuorumLogDrive) with failure and delay injection; and then check whether the resulting execution is linearizable.

Interestingly, this simulation testing drove many of our design insights. For example, we had initially assumed that the LogDrive would have to provide a linearizable *weakTail*; it was only after a pagination-induced linearizability violation in our S3LogDrive went uncaught in the AtomicLog sim-test that we realized we did not need *weakTail* linearizability: the AtomicLog was linearizable even if LogDrive was not. As an aside, our simulator also independently discovered a seal check ordering bug in the VirtualLog protocol described subsequently by the Delos team [35].

Low-Latency AtomicLog: As described, the AtomicLog uses the LogDrive abstraction to extract, augment, and scale durability (e.g., via quorums and striping) from external cloud storage. In one extreme, we use a primitive LogDrive (e.g., S3LogDrive) directly under AtomicLog, without further stacking. In principle, Conflux should also be able to support the other extreme, where we rely entirely on stacked LogDrives to provide durability over a non-durable primitive LogDrive.

To test this hypothesis, we implemented another primitive LogDrive: a simple, single-node LogServer that implements the LogDrive API via RPC, storing state on a local SSD

without further replication or striping. We obtain replicated durability by layering a QuorumLogDrive on top (which writes to a quorum of LogServers); and scale throughput via a StripedLogDrive. This resulting AtomicLog deployment is equivalent to a conventional shared log implemented on servers, allowing us to trade off operational overhead for lower latency within the same codebase.

5 Conflux and K2

We now describe how we built a metadata store called Conflux over the AtomicLog / LogDrive abstractions; and how we used it to power K2, a new Kafka implementation that stores data in S3 and metadata in Conflux.

5.1 The Conflux Metadata Store

Conflux has a shared log design, nearly identical to Delos [18] and borrowing from other shared log systems from research and industry. A Conflux service consists of one or more servers, each of which stores a local, single-node database (in production we use RocksDB [26]); and a single VirtualLog that each server can append to and read from. Conflux implements State Machine Replication [49] over the VirtualLog (as shown in Figure 6): updates to the service can be sent to any server; which *append*s an entry describing the update to an external shared log; and then synchronizes its local state with the shared log by calling *readNext* and applying new updates (until it executes the one it just added). Read-only queries can be served by any server, which calls *checkTail* on the shared log to establish a linearization point; and then synchronizes its local state until that tail position; before serving the query from local state. This simple protocol results in a replicated, linearizable database where both ordering and durability are delegated entirely to the external shared log, which is essentially the linearizable or strictly serializable total order of updates to the service. In principle, we can reconstruct state by replaying the shared log from the beginning on a new server; in practice, we frequently checkpoint state to cloud storage and trim the shared log, so that only a small suffix of the log has to be replayed on recovery.

By default, Conflux is a multi-master database that stores input commands (prior to execution) in the shared log and executes them (deterministically) on playback; but also supports modes where servers execute commands and append speculative outputs to the log [21]; or a single server (elected via the log itself) acts as a primary and stores non-speculative outputs in the log. In multi-master mode, reads have to wait for at least one round-trip to check the tail of the shared log; we use a "bus-stand" optimization [17] to amortize the bandwidth cost of this check, with many reads queueing up behind a single "bus" that checks the tail periodically.

Above the shared log, Conflux uses a stack of SMR layers [20]. These layers include logic for time/size-based batching and observability. The top-most SMR layer is the application, which can expose any arbitrary API. Later in the paper, we describe the specific API we implement on Conflux to support its primary use case as the K2 metadata store.

As described earlier, Conflux implements the VirtualLog abstraction from Delos, which separates reconfiguration from sequencing and durability (Figure 1 showed this separation of concerns earlier in the paper). The VirtualLog stitches together a number of Loglets into a single shared log. As a result, Conflux can switch from one Loglet to another without downtime, dynamically changing the underlying storage backend. The membership of the VirtualLog (i.e., the sequence of Loglets and the address ranges they cover) is stored in an external conditional register, which is the ultimate (and only) source of consensus in the system; in Conflux, this register is stored directly on DynamoDB (cost is not a concern since the state is small and only accessed on node reboots and reconfigurations).

5.2 The K2 Pub-Sub System

We now describe how we used Conflux to power K2, a new Kafka implementation that stores data in S3 and metadata in Conflux. K2 consists of a collection of servers called *brokers* (mirroring Kafka parlance) that implement the Kafka protocol. Clients can connect to brokers to *publish* data to a topic; and *fetch* data from the topic. As mentioned previously, Kafka uses the term 'partition' to refer to topics; but we use the classical definition of a topic as a single total order. Each broker collects incoming publish requests (across different topics) into a large batch to write to S3; and then updates metadata stored in Conflux. On a fetch request, the broker consults metadata in Conflux to retrieve the location of the requested data, and serves the data by reading S3.

Over time, brokers can access and rewrite state in S3: to switch it from the initial write-optimized format to a read-optimized one; to compact topics (i.e., remove overwritten updates to the same fine-grained key); or to build additional indexes over the topic for data lake formation. This background activity can be isolated on a separate pool of background brokers, or be collocated on the client-facing brokers.

In principle, any broker can respond to a publish or consume request for any topic. In practice, we shard the space of hashed topic IDs into ordered ranges [14], and assign different ranges to sets of brokers (typically one per AZ, to enable zone-aware routing from clients that minimizes inter-AZ data transfer costs). Affinitizing brokers to topics enables better locality within the initial write batches, which in turns results in more efficient background processing; and simplifies caching for reads. Brokers are not required to be durable – we can reconstruct the system entirely from S3 and Conflux – but do maintain soft-state caches of data.

```
interface K2MetaStore {
 long appendDataRef(TopicID id, DataRef newData);
 DataRef fetchDataRef(TopicID id, long address);
}
```

Figure 7. *Simplified subset of API exposed by Conflux to K2.*

5.3 Conflux in K2

As noted earlier, Conflux is a general-purpose SMR system that can replicate any deterministic service with an arbitrary API. To support K2, we implement a particular service called the K2MetaStore. In the remainder of this discussion, we use Conflux interchangeably with K2MetaStore (until we introduce other APIs). Figure 7 shows the subset of this API used in the critical path of publish-subscribe requests.

For each topic, Conflux stores an ordered list of pointers called DataRefs. To a first approximation, a DataRef is a pointer to some external piece of data. Accordingly, when a broker stores data in S3, it creates an S3DataRef – a pointer to a portion of an S3 object – and then sends it to Conflux, which adds it to the ordered list.

Logs on Logs: Each ordered list in Conflux can be viewed as a fine-grained shared log within a state machine, which in turn is replicated over a coarse-grained shared log. We exploit this log-on-log structure by using a common API for both layers: the ordered list of DataRefs implements a read-only subset of the API in Figure 5, as does each individual DataRef. Using a common API at different layers has a number of practical implications. For example, we were able to augment our test coverage for Conflux by reusing our sim-test framework from Section 4.1 to run against a single topic, testing the entire stack end-to-end for linearizability. Practically, a common base API for both the fine-grained application-level construct and the coarse-grained replication log allowed us to reuse code: oncall tooling, wrappers for observability, composition, and interception; reusable benchmarking logic; etc. In making this choice, we were inspired by a familiar pattern in OS design: e.g., virtual disks and physical disks.

In addition, DataRefs are not necessarily just passive pointers to S3 blobs; as shared logs, they can be backed by complex systems. This allows Conflux to delegate indexing for a prefix of the topic: e.g., to a prior Kafka deployment (for migration) or to a data lake format on S3 (for tiering cold metadata).

Sharding in Conflux: In production, we compose multiple Conflux instances into a larger, sharded instance, for scaling throughput and capacity. We use an identical sharding scheme to the K2 brokers, to maximize locality between brokers and Conflux shards (to minimize the fan-out between a broker and the number of Conflux shards it interacts with).

Re-sharding is currently done via a conventional transaction protocol that locks and moves sub-ranges of hashed topic IDs from one shard to another. We are currently exploring ways to exploit the log-like nature of the state machine

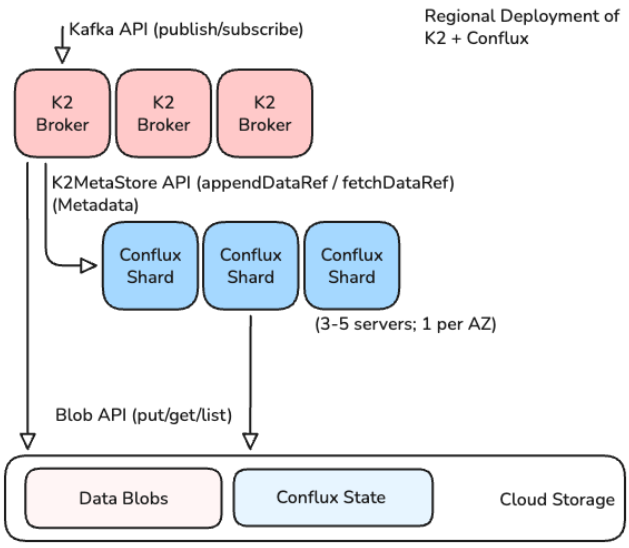


Figure 8. *K2 servers store data blobs in cloud storage and metadata in Conflux. (See Figure 6 for the structure of a single Conflux shard).*

for faster moves: e.g., we can virtualize a topic's index to span physical segments on different shards. This is similar to the shared log virtualization we use in the consensus layer, echoing Lampson's principle of "use a good idea twice" [43].

Other APIs: To support other parts of the Kafka protocol, we implemented two other separate APIs on Conflux in addition to the K2MetaStore. The first is a K2GroupService which implements the group coordinator abstraction in Kafka, responsible for managing groups of consumers fetching from a bundle of individual total orders (i.e., which Kafka calls a topic; or equivalently a bundle of conventional pub-sub topics), mapping consumers to total orders and tracking cursor offsets. The second is a K2TxService, which implements a coordinator for implementing Kafka transactions.

In production, we deploy a single Conflux service for each of our three APIs within a region. Each such regional Conflux deployment consisting of multiple shards. Each shard is replicated across multiple Conflux servers (typically one per AZ, such that we have between 3 to 5 servers, depending on the AWS region).

6 Evaluation

We present an evaluation of Conflux using a combination of production and benchmarking data. All our deployments run on AWS EC2 instances deployed via K8s [5], using EBS drives for network-attached storage. In all experiments, we have production-grade logging and monitoring enabled via Datadog [3]. In our single-region tests, each Conflux deployment consists of one server per AZ (for example, in us-west-2 we run four servers), while clients are distributed across all AZs. Since Conflux servers store only soft state, the number

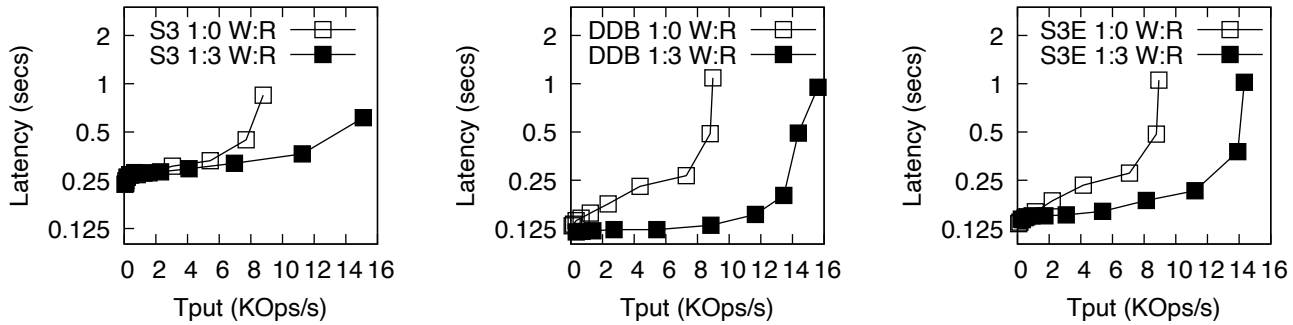


Figure 9. *Conflux can run on S3 (Left), DynamoDB with Striping (Middle), and S3Express with Quorums (Right).*

of servers has no durability implications. In our multi-region tests, we continue to run our deployment entirely on us-west-2, but synchronously mirror data to two other regions: us-east-1 (~60ms away) and eu-east-1 (~120ms away).

In our cost and latency comparisons, we include a hypothetical DynamoDB strawman for the Conflux API. We assume that each *appendDataRef* results in one write to DynamoDB; and each *fetchDataRef* is one read. In practice, implementing a production-ready shim on DynamoDB would require non-trivial engineering effort (e.g., concurrency control between competing *appendDataRef* calls); however, this simple model gives us a lower bound on achievable latency and cost.

In our production workloads, the average write size to Conflux is 165 bytes; and the write:read ratio is typically 1:3. In our benchmarking setup, we use m6g.xlarge EC2 instances; we omit details of our production servers. We use three cloud storage backends in our experiments. S3 and DynamoDB provide durability against AZ failures, while S3Express can lose data with one AZ failure.

In our setups, we vary the LogDrive used by Conflux, but use S3 to store snapshots every 10 minutes (so that the shared log does not have to be replayed from the beginning). By default, we set a batching timeout in Conflux to 100ms; and a size threshold to 60KB. The Conflux AtomicLog is configured with a windowsize of $K = 16$. All reported latencies are p99s.

6.1 Pluggability and Composability

Conflux can leverage the LogDrive abstraction to easily run on and *extract durability* from different cloud stores with minimal effort. Figure 9 shows the throughput-latency curve for different Conflux deployments, running over S3 (Left), DynamoDB (Middle), and S3Express (Right), respectively.

Interestingly, this graph also illustrates that the LogDrive can *augment the durability* of existing cloud storage: to provide an apples-to-apples comparison with S3Express (which is natively single-AZ), we layered a QuorumLogDrive over three S3ExpressLogDrives, each one in a different AZ.

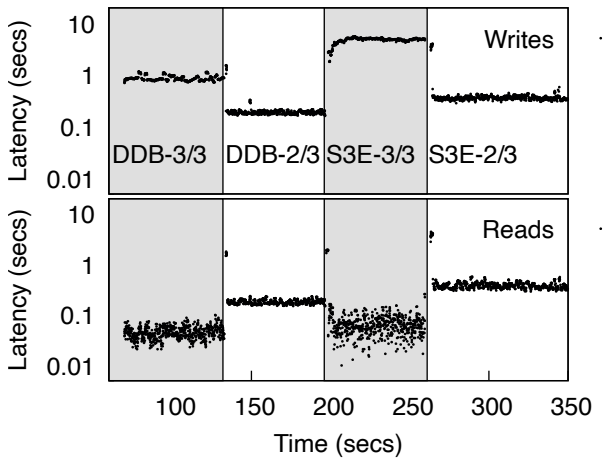


Figure 10. *Conflux reconfig: DynamoDB (3,3) quorum to DynamoDB (2,3); to S3Express (3,3); to S3Express (2,3). Replicas in us-west-2, us-east-1, eu-east-1; clients in us-west-2.*

Conflux can switch its VirtualLog between different AtomicLogs which are backed by different LogDrives. In Figure 10, we start running Conflux on a QuorumLogDrive running on a (3,3) quorum of DynamoDBLogDrive instances running in us-west-2, us-east-1, and eu-east-1; writes have to wait for all regions, but reads are cheap. We then switch to a (2,3) quorum: writes and reads are now equally fast, since they both have to reach a majority. After that we switch to (3,3) quorum of S3Express single-AZ instances (one AZ per region); and finally we switch to a (2,3) quorum of S3Express. Finally, in Figure 11, we show this (2,3) quorum S3Express configuration seamlessly tolerate an S3Express outage in us-west-2 (emulated by disable access control). These graphs illustrates that Conflux can reconfigure without downtime between different backends; obtain different durability / latency profiles via quorum sizes; augment DynamoDB (one-region) and S3Express (one-AZ) to provide multi-region durability; and tolerate regional storage failures. Table 12 shows the LOC for the various LogDrive implementations involved.

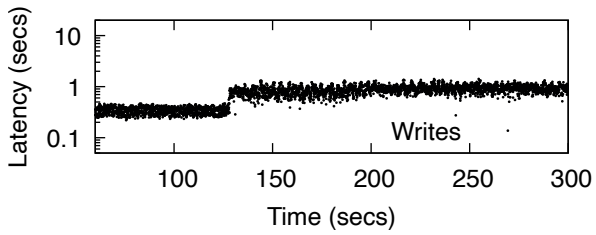


Figure 11. *Conflux S3Express (2,3): S3Express fails in us-west-2.*

Implementation	LOC
S3LogDrive	519
DynamoDBLogDrive	528
S3ExpressLogDrive	525
StripedLogDrive	194
QuorumLogDrive	325

Figure 12. *LOCs for LogDrive implementations.*

Hourly cost ($)	DDB	Conflux (DDB)	Conflux (S3)	Conflux (S3Express)
Instance+disk	0	0.47	0.47	0.47
Write API	4.50	0.15	0.54	0.37
Read API	2.70	0.07	0.09	0.01
Metadata cost	7.20	0.69	1.10	0.85
Total cost	9.57	3.06	3.47	3.22

Figure 13. *Conflux-over-DDB is 10X cheaper than DDB for metadata and over 3X cheaper overall (data plane = $2.37/hr).*

6.2 Cost

We provide a detailed cost estimate of running Conflux over the DynamoDBLogDrive, compared with using DynamoDB directly. We pick a record size of 165 bytes, matching the average observed in our production workloads. For Conflux, we include the cost of 3 EC2 instances for the Conflux deployment and the associated EBS disk cost. Our modeled workload issues 2K writes/s and 6K reads/s. We use the latest available cost information from AWS: for DynamoDB, we used on-demand capacity pricing [1] in the us-west-2 region ($0.625/M Write Request Units / WRUs, and $0.125/M Read Request Units).

Table 13 shows that Conflux-over-DynamoDB is an order of magnitude cheaper than the direct-over-DynamoDB strawman. This cost differential arises for a number of reasons. First, Conflux collects small writes in a batch and issues a single large write to DynamoDB (i.e., the append to the shared log). Since each write (regardless of size, up to 1KB) is charged independently as a WRU, batching lowers our cost significantly even though we are writing the same amount of data in both Conflux and the strawman design. Second, batching writes allows us to compress writes more efficiently (we model a factor of 5X, which is what we observe in our production workloads). Third, Conflux serves reads from a strongly consistent, full copy of the database (i.e., by playing the shared log), avoiding redundant reads to DynamoDB. We also include cost numbers for Conflux on S3 and S3Express (replicated over 3 AZs); both are more expensive than DynamoDB.

The Conflux setup we model here is identical to the one we evaluated earlier in Figure 9; accordingly, the corresponding p99 latency is roughly ~130ms. As a result, our $10X$ cost win comes with a $6.5X$ write slowdown; but the resulting latency is acceptable for Conflux's use case.

Separately, we validate that metadata is indeed a significant fraction of overall cost. To do so, we model the corresponding data plane cost assuming: (a) each data blob referenced by a metadata record is $128KB$; (b) data blobs are batched together into $5MB$ objects; (c) a $1:3$ ratio of publish to fetch. Under these assumptions, we obtain a data plane cost of $2.37 per hour. As a result, replacing a strawman DynamoDB in K2 cuts cost by more than $3X$ (see Table 13).

6.3 K2 in Production

This section shares empirical data from a production K2 deployment that shows the performance and reconfigurability benefits of Conflux. This deployment spans three AZs in us-west-2 region and uses LogDrive-on-DynamoDB with a batching latency of 100ms. The deployment consists of 114 K2 brokers, of which 108 are serving requests. We use 20 Conflux instances as shards with three servers each.

Figure 14 shows a screenshot of our production dashboard for a run with $9GB/s$ produce and $27GB/s$ fetch. Latencies are reported over 1-min intervals from each K2 broker. Produce p99 latency is about 600ms whereas fetch p99 latency is about 300ms; the 500ms numbers are due to a blocking timeout on client fetches for empty topics. Figure 15 (Left) compares data plane and metadata latency on the production run (plotting the p99 across per-server p99s). The produce latency is dominated by the latency to batch and upload data to S3 and then to sequence it via an *appendDataRef* RPC to Conflux (which has a p99 of 130ms, in line with our micro-benchmarks for Conflux-on-DynamoDB). Fetch latency is dominated by the latency to retrieve data from S3; sometimes serving a single client fetch might require reading a number of different S3 objects, which can aggravate the tail latency. In this run, the average metadata record size is around 165 bytes and we get an effective compression of around $6-7X$.

Finally, we show Conflux change backends mid-flight. Our initial deployment used S3LogDrive under Conflux. However, we discovered that S3LogDrive had suboptimal performance (as we showed in Section 6.1) as well as cost (as we saw in Section 6.2) compared to DynamoDBLogDrive. Accordingly, we decided to switch our early access service to use Conflux over DynamoDB. A single engineer implemented and

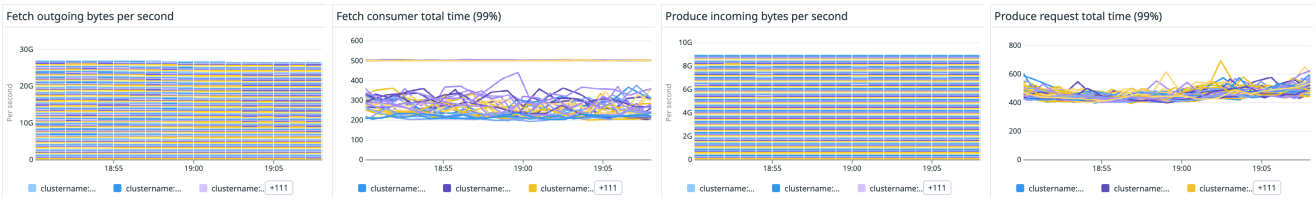


Figure 14. *Screenshot of the production dashboard for a K2 cluster serving 9GB/s produce and 27GB/s fetch throughput.*

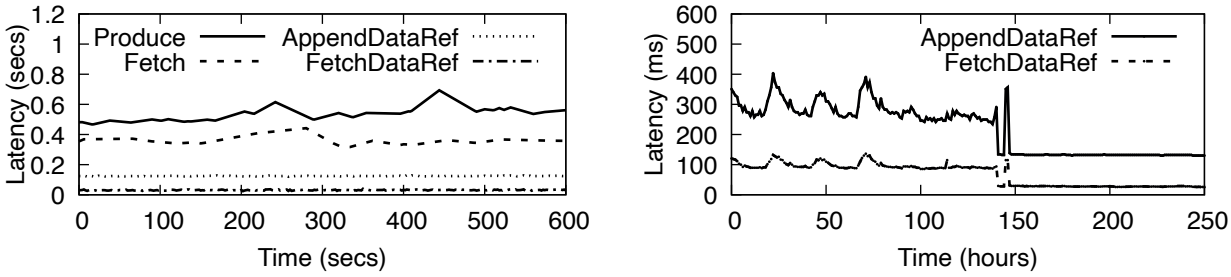


Figure 15. *(Left) Latency on K2 9GB/s produce / 27GB/s fetch production cluster (p99 across per-server p99s); (Right) switchover on an early production cluster from S3 to DynamoDB.*

deployed the DynamoDBLogDrive in about 1 week, leveraging existing simulation tests and observability wrappers. Early tests hit rate limits on DDB partition key accesses; the engineer was able to work around that by using the StripedLogDrive. Figure 15 (Right) shows a running Conflux service make the switch without downtime. Interestingly, we first switched to an incorrect table; switched immediately back to S3 (hence the spike); and then made the correct switch.

7 Related Work

As described, K2 implements large numbers of fine-grained shared logs over an S3-based data plane and the Conflux metadata layer; and Conflux is itself a database designed over a shared log. Accordingly, both K2 and Conflux are shared log designs which intersect with the literature in different ways.

Early group communication systems [23, 30, 51] provided flexible placement of sequencing and durability: e.g., sequencing could be out-of-band or in-band, fixed or moving, etc. While the theory of Paxos retained this flexibility [42, 50, 53], production-ready MultiPaxos systems [25] and later Raft [46] adopted inflexible leader-driven designs where a specific node acted as a sequencer, a storage acceptor, and a database learner.

Scalable shared logs: Initial shared logs like Corfu [19] reintroduced sequencing flexibility by separating the location of sequencing from durable storage, in a "sequence-first, write-later" design; in doing so, they scaled the throughput of a single log. Later shared log designs like Scalog [31] reversed this order to "write-first, sequence-later", shifting the burden of storing the durable sequence to a stateful metadata layer. In the same timeline, industry systems such as Kafka [40] and BookKeeper [39] focused instead on scaling to large numbers of fine-grained leader-driven logs. Given this history, K2 can be viewed as a "write-first, sequence-later" system that scales to many fine-grained logs (with Conflux acting as the metadata layer storing the durable sequence for each log).

Shared logs for SMR: At the same time, shared logs have also played a different role by providing an API for consensus, simplifying SMR-based production databases that need extreme reliability, arbitrary APIs, composability, and zero-downtime upgrades, but not massive scale or low latency [2, 18]. Conflux extends this literature by enabling these properties over cloud storage at low cost. In addition, the AtomicLog / LogDrive can be used underneath any existing SMR-based database.

Shared logs have been recently extended to serverless functions [36, 38, 48] and stream processing [54]. The ideas in Conflux could potentially allow these systems to operate directly over cloud storage. In addition, the LazyLog [44] applies the same principle of separating sequencing from durability in a slightly different way. In a sense, the LazyLog changes *when* a timestamp is assigned to an appended entry, whereas in this work we focus on *which layer* assigns the timestamp.

Streaming on cloud storage: K2 fits into a trend towards constructing high-level APIs by combining cloud storage with a metadata layer. Snowflake's database architecture [45] over S3 and FoundationDB is an early and significant example of this design pattern. Warpstream [13] is another

example of a Kafka implementation that uses S3 as its durability layer. While K2 shares this design pattern, our primary contribution is the LogDrive abstraction and its use within Conflux to provide a simple, robust, and inexpensive metadata layer over cloud storage.

8 Conclusion

Commodity cloud storage has the potential to unlock innovation at higher layers by providing durability as a service. At the same time, existing cost structures make it prohibitively expensive to build metadata layers that require small, frequent accesses. In this paper, we proposed a new system called Conflux that uses cloud storage as a shared log for replicating updates, extracting durability while slashing cost. Conflux introduces a novel shared log design that separates sequencing (via the AtomicLog implementation) and durability (via the LogDrive abstraction). The LogDrive supports arbitrary RAID-like stacking, allowing us to combine and augment existing cloud durability in new ways. Conflux is deployed at Confluent as the metadata service for K2 (which in turn powers the Freight product), driving down metadata cost by 10X and overall system cost by 3X.

9 Acknowledgments

We would like to thank Chad Verbowski, Jun Rao, and Jay Kreps for funding the K2 project and shaping its goals. Anil Sharma and Kamal Gupta were responsible for managing early iterations of the team. Hakan Hacigumus significantly impacted the project in its later stages. We would also like to thank our anonymous OSDI shepherd. Finally, this paper would not have been written without CQ Tang's advice and encouragement.

References

[1] [n. d.]. Amazon DynamoDB Pricing for On-Demand Capacity. https://aws.amazon.com/dynamodb/pricing/on-demand/.
[2] [n. d.]. CorfuDB. https://github.com/corfudb.
[3] [n. d.]. Datadog. https://www.datadoghq.com/.
[4] [n. d.]. etcd. https://etcd.io/.
[5] [n. d.]. K8s. https://kubernetes.io/.
[6] [n. d.]. KRaft. https://docs.confluent.io/platform/current/kafka-metadata/kraft.html.
[7] [n. d.]. LogDevice. https://logdevice.io/.
[8] [n. d.]. S3. https://aws.amazon.com/s3/.
[9] [n. d.]. S3 Express. https://aws.amazon.com/s3/storage-classes/express-one-zone/.
[10] [n. d.]. S3Express API Differences. https://docs.aws.amazon.com/AmazonS3/latest/userguide/s3-express-differences.html.
[11] 2025. Confluent Freight. https://www.confluent.io/blog/freight-clusters-are-generally-available/.
[12] 2025. S3Express Price Reduction. https://aws.amazon.com/blogs/aws/up-to-85-price-reductions-for-amazon-s3-express-one-zone/.
[13] 2026. Warpstream. https://www.warpstream.com/
[14] Atul Adya, Daniel Myers, Jon Howell, Jeremy Elson, Colin Meek, Vishesh Khemani, Stefan Fulger, Pan Gu, Lakshminath Bhuvanagiri, Jason Hunter, et al. 2016. Slicer: Auto-Sharding for datacenter applications. In *USENIX OSDI 2016*.
[15] Yehuda Afek, Hagit Attiya, Danny Dolev, Eli Gafni, Michael Merritt, and Nir Shavit. 1993. Atomic snapshots of shared memory. *Journal of the ACM (JACM)* 40, 4 (1993), 873–890.
[16] Hagit Attiya, Amotz Bar-Noy, and Danny Dolev. 1995. Sharing Memory Robustly in Message-Passing Systems. *Journal of the ACM (JACM)* 42, 1 (1995), 124–142.
[17] Mahesh Balakrishnan. 2024. Taming Consensus in the Wild (with the Shared Log Abstraction). In *ACM SIGOPS OSR 2024*.
[18] Mahesh Balakrishnan, Jason Flinn, Chen Shen, Mihir Dharamshi, Ahmed Jafri, Xiao Shi, Santosh Ghosh, Hazem Hassan, Aaryaman Sagar, Rhed Shi, Jingming Liu, Filip Gruszczynski, Xianan Zhang, Huy Hoang, Ahmed Yossef, Francois Richard, and Yee Jiun Song. 2020. Virtual Consensus in Delos. In *USENIX OSDI 2020*.
[19] Mahesh Balakrishnan, Dahlia Malkhi, Vijayan Prabhakaran, Ted Wobber, Michael Wei, and John D. Davis. 2012. CORFU: A Shared Log Design for Flash Clusters. In *USENIX NSDI 2012*.
[20] Mahesh Balakrishnan, Chen Shen, Ahmed Jafri, Suyog Mapara, David Geraghty, Jason Flinn, Vidhya Venkat, Ivailo Nedelchev, Santosh Ghosh, Mihir Dharamshi, et al. 2021. Log-structured protocols in Delos. In *ACM SOSP 2021*.
[21] Philip A Bernstein, Sudipto Das, Bailu Ding, and Markus Pilman. 2015. Optimizing Optimistic Concurrency Control for Tree-Structured, Log-Structured Databases. In *ACM SIGMOD 2015*.
[22] Shreesha G Bhat, Tony Hong, Xuhao Luo, Jiyu Hu, Aishwarya Ganesan, and Ramnatthan Alagappan. 2025. Low End-to-End Latency atop a Speculative Shared Log with Fix-Ante Ordering. In *USENIX OSDI 2025*.
[23] Kenneth P Birman and Thomas A Joseph. 1987. Reliable Communication in the Presence of Failures. *ACM Transactions on Computer Systems (TOCS)* 5, 1 (1987), 47–76.
[24] Matthew Burke, Audrey Cheng, and Wyatt Lloyd. 2020. Gryff: Unifying consensus and shared registers. In *USENIX NSDI 2020*.
[25] Mike Burrows. 2006. The Chubby lock service for loosely-coupled distributed systems. In *USENIX OSDI 2006*.
[26] Zhichao Cao, Siying Dong, Sagar Vemuri, and David HC Du. 2020. Characterizing, Modeling, and Benchmarking RocksDB Key-Value Workloads at Facebook. In *USENIX FAST 2020*.
[27] Armando Castañeda, Sergio Rajsbaum, and Michel Raynal. 2015. Specifying concurrent problems: beyond linearizability and up to tasks. In *International Symposium on Distributed Computing*. Springer, 420–435.

[28] Tushar D Chandra, Robert Griesemer, and Joshua Redstone. 2007. Paxos made live: an engineering perspective. In *ACM PODC 2007*.
[29] Jeffrey Dean and Luiz André Barroso. 2013. The tail at scale. *Commun. ACM* 56, 2 (2013), 74–80.
[30] Xavier Défago, André Schiper, and Péter Urbán. 2004. Total order broadcast and multicast algorithms: Taxonomy and survey. *ACM Computing Surveys (CSUR)* 36, 4 (2004), 372–421.
[31] Cong Ding, David Chu, Evan Zhao, Xiang Li, Lorenzo Alvisi, and Robbert van Renesse. 2020. Scalog: Seamless Reconfiguration and Total Order in a Scalable Shared Log. In *USENIX NSDI 2020*.
[32] Mostafa Elhemali, Niall Gallagher, Bin Tang, Nick Gordon, Hao Huang, Haibo Chen, Joseph Idziorek, Mengtian Wang, Richard Krog, Zongpeng Zhu, et al. 2022. Amazon {DynamoDB}: A scalable, predictably performant, and fully managed {NoSQL} database service. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*. 1037–1048.
[33] Patrick Th Eugster, Pascal A Felber, Rachid Guerraoui, and Anne-Marie Kermarrec. 2003. The many faces of publish/subscribe. *ACM computing surveys (CSUR)* 35, 2 (2003), 114–131.
[34] Michael J Fischer, Nancy A Lynch, and Michael S Paterson. 1985. Impossibility of distributed consensus with one faulty process. *Journal of the ACM (JACM)* 32, 2 (1985), 374–382.
[35] David Geraghty, Mahesh Balakrishnan, Suyog Mapara, and David Devecsery. 2025. Erratum to "Virtual Consensus in Delos". https://maheshba.bitbucket.io/papers/delos-erratum.pdf
[36] Dimitra Giantsidi, Emmanouil Giortamis, Nathaniel Tornow, Florin Dinu, and Pramod Bhatotia. 2023. Flexlog: A shared log for stateful serverless computing. In *ACM HPDC 2023*.
[37] Patrick Hunt, Mahadev Konar, Flavio Paiva Junqueira, and Benjamin Reed. 2010. ZooKeeper: Wait-free Coordination for Internet-scale Systems. In *USENIX ATC 2010*.
[38] Zhipeng Jia and Emmett Witchel. 2021. Boki: Stateful Serverless Computing with Shared Logs. In *ACM SOSP 2021*.
[39] Flavio P Junqueira, Ivan Kelly, and Benjamin Reed. 2013. Durability with bookkeeper. *ACM SIGOPS OSR* 47, 1 (2013), 9–15.
[40] Martin Kleppmann and Jay Kreps. 2015. Kafka, Samza and the Unix Philosophy of Distributed Data. *IEEE Data Engineering Bulletin, 38 (4)* (2015).
[41] Leslie Lamport. 1986. On interprocess communication: part I: basic formalism. *Distributed computing* 1, 2 (1986), 77–85.
[42] Leslie Lamport. 1998. The Part-Time Parliament. *ACM Transactions on Computer Systems (TOCS)* 16, 2 (1998), 133–169.
[43] Butler W Lampson. 1983. Hints for computer system design. In *ACM SOSP 1983*.
[44] Xuhao Luo, Shreesha G Bhat, Jiyu Hu, Ramnatthan Alagappan, and Aishwarya Ganesan. [n. d.]. LazyLog: A New Shared Log Abstraction for Low-Latency Applications. In *ACM SOSP 2024*.
[45] Ashish Motivala. 2018. How FoundationDB Powers Snowflake Metadata Forward. https://www.snowflake.com/en/blog/how-foundationdb-powers-snowflake-metadata-forward/
[46] Diego Ongaro and John K Ousterhout. 2014. In Search of an Understandable Consensus Algorithm. In *USENIX ATC 2014*.
[47] David A Patterson, Garth Gibson, and Randy H Katz. 1988. A case for redundant arrays of inexpensive disks (RAID). In *ACM SIGMOD 1988*.
[48] Sheng Qi, Xuanzhe Liu, and Xin Jin. 2023. Halfmoon: Log-optimal fault-tolerant stateful serverless computing. In *ACM SOSP 2023*.
[49] Fred B Schneider. 1990. The state machine approach: A tutorial. In *Fault-tolerant distributed computing*. Springer, 18–41.
[50] Robbert Van Renesse and Deniz Altinbuken. 2015. Paxos made moderately complex. *ACM Computing Surveys (CSUR)* 47, 3 (2015), 42.
[51] Robbert Van Renesse, Kenneth P Birman, and Silvano Maffeis. 1996. Horus: A Flexible Group Communication System. *Commun. ACM* 39, 4 (1996), 76–83.
[52] Michael Wei, Amy Tai, Christopher J Rossbach, Ittai Abraham, Maithem Munshed, Medhavi Dhawan, Jim Stabile, Udi Wieder, Scott Fritchie, Steven Swanson, et al. 2017. vCorfu: A Cloud-Scale Object Store on a Shared Log. In *USENIX NSDI 2017*.
[53] Michael Whittaker, Ailidani Ailijiang, Aleksey Charapko, Murat Demirbas, Neil Giridharan, Joseph M Hellerstein, Heidi Howard, Ion Stoica, and Adriana Szekeres. 2021. Scaling replicated state machines with compartmentalization. *Proceedings of the VLDB Endowment* 14, 11 (2021), 2203–2215.
[54] Zhiting Zhu, Zhipeng Jia, Newton Ni, Dixin Tang, and Emmett Witchel. 2025. Impeller: Stream Processing on Shared Logs. In *EuroSys 2025*.

A LogDrive Proofs

A.1 Specification of the LogDrive

A LogDrive is a bounded array of $N+1$ registers, indexed by the integers $A = \{0, 1, \ldots, N\}$. Each address $a \in A$ holds either a payload from a value set $\mathcal{V}$ or the distinguished value $\perp$ indicating *unwritten*. At time t, the system state is

$$S(t) : A \to \mathcal{V} \cup \{\perp\}.$$

For simplicity, we use the Nth register as a sentinel: $S(t, N) = \perp$ for all t. We assume that this sentinel register is never written to.

Read/write semantics. Operations $\textsc{write}(a, v)$ and $\textsc{read}(a)$ are *linearizable*. That is, there exists a total order of all completed reads and writes, consistent with real-time precedence, such that:

- A write $\textsc{write}(a, v)$ takes effect at its linearization point, setting $S(t, a) = v$.
- A read $\textsc{read}(a)$ returns $S(t, a)$ at its linearization point.
- **Write-once monotonicity:**

$$S(t_1, a) \neq \perp \;\Rightarrow\; S(t_2, a) \neq \perp \qquad \text{for all } t_1 \leq t_2.$$

  Once written, an address is never unwritten.
- **Single-value assumption.** If two writes to the same address propose *different* values, the behavior is unspecified. All correctness arguments assume that executions are *well-formed*: for each address a, at most one value v is ever written to a.

Contiguous tail. At any time t, the contiguous tail is:

$$\mathcal{T}(t) = \min\{a \mid S(t, a) = \perp\}.$$

Non-contiguous tail and K-window discipline. At time t, define the *non-contiguous tail* as

$$\mathcal{N}(t) = \min\Big\{ a \in A \;\Big|\; S(t, a) = \perp \text{ and } \forall b \in A,\ b > a \Rightarrow S(t, b) = \perp \Big\}.$$

That is, $\mathcal{N}(t)$ is the first unwritten address after which all addresses are unwritten.

We say that an execution satisfies the *K-window discipline* if, for all times t,

$$\mathcal{N}(t) - \mathcal{T}(t) \leq K.$$

Thus at most K addresses in the prefix $A_{<\mathcal{N}(t)}$ are unwritten. Intuitively, this captures that the application never has more than K *in-flight* writes.

weakTail semantics. Let a weakTail(K) operation o execute during the real-time interval $[t_{\text{start}}(o),\ t_{\text{stop}}(o)]$. During this interval, o determines (via direct reads or implicit deduction) the written/unwritten status of every address. Formally, for each $a \in A$, let

$$\tau_o(a) \in [\ t_{\text{start}}(o),\ t_{\text{stop}}(o)\ ]$$

denote the time at which o observes the status of address a. Define

$$w_o(a) = \begin{cases} \text{written} & \text{if } S(\tau_o(a), a) \neq \bot, \\ \text{unwritten} & \text{if } S(\tau_o(a), a) = \bot. \end{cases}$$

The return value of weakTail(K) is produced by a deterministic function

$$F : w_o \longmapsto (\mathcal{N}_{\text{obs}}, \mathcal{H}_{\text{obs}}),$$

where:

- $\mathcal{N}_{\text{obs}} = \min\{\, a \in A \mid w_o(a) = \text{unwritten and } \forall b \in A,\ b > a \Rightarrow w_o(b) = \text{unwritten} \,\}$
- $\mathcal{H}_{\text{obs}} = \{\, a \in A \mid a < \mathcal{N}_{\text{obs}} \text{ and } w_o(a) = \text{unwritten} \,\}$ is the *observed hole set*, i.e., the unwritten addresses between the contiguous and non-contiguous observed tails.

Further, the caller can derive the contiguous tail $\mathcal{T}_{\text{obs}}$ from the return value of weakTail. Given $(\mathcal{N}_{\text{obs}}, \mathcal{H}_{\text{obs}})$, the observed contiguous tail is

$$\mathcal{T}_{\text{obs}} = \begin{cases} \min \mathcal{H}_{\text{obs}} & \text{if } \mathcal{H}_{\text{obs}} \neq \emptyset, \\ \mathcal{N}_{\text{obs}} & \text{if } \mathcal{H}_{\text{obs}} = \emptyset. \end{cases}$$

For brevity, in the remainder of the proof we will speak of weakTail as "returning" $\mathcal{T}_{\text{obs}}$, with the understanding that this value is computed in the above manner.

Intuitively, weakTail corresponds to an *unordered, non-atomic scan* in which each address a is read at least once at some time $\tau_o(a)$ within the operation interval, and the operation returns the first unwritten address beyond which all addresses are unwritten; as well as the list of holes prior to it.

A.2 Derived Properties of LogDrive

Lemma ([LD.1] Tail-Range Guarantee**).** *Let a* weakTail(K) *operation o execute during the real-time interval $[t_{\text{start}},\ t_{\text{stop}}]$, and let*

$$\mathcal{T}_{\text{start}} = \mathcal{T}(t_{\text{start}}), \qquad \mathcal{T}_{\text{stop}} = \mathcal{T}(t_{\text{stop}})$$

be the true contiguous tails at the start and end of o. Let $\mathcal{T}_{\text{obs}}$ be the observed tail returned by o, as defined in the weakTail *semantics. Then*

$$\mathcal{T}_{\text{obs}} \in [\mathcal{T}_{\text{start}},\ \mathcal{T}_{\text{stop}}].$$

Proof. By the weakTail semantics, for each address $a \in A$ there is an observation time

$$\tau_o(a) \in [t_{\text{start}}, t_{\text{stop}}],$$

and $w_o(a)$ is determined from the state at that time as written or unwritten.

Lower bound. Suppose, for contradiction, that $\mathcal{T}_{\text{obs}} < \mathcal{T}_{\text{start}}$. By definition of $\mathcal{T}_{\text{start}}$, every address $a < \mathcal{T}_{\text{start}}$ is written at time t_{start}, i.e. $S(t_{\text{start}}, a) \neq \bot$. By write-once monotonicity, once written an address never becomes unwritten, so for all $t \geq t_{\text{start}}$ we have $S(t, a) \neq \bot$. In particular,

$$S\big(\tau_o(\mathcal{T}_{\text{obs}}), \mathcal{T}_{\text{obs}}\big) \neq \bot,$$

since $\tau_o(\mathcal{T}_{\text{obs}}) \in [t_{\text{start}}, t_{\text{stop}}]$. Hence $w_o(\mathcal{T}_{\text{obs}}) = \text{written}$, contradicting the assumption that $\mathcal{T}_{\text{obs}}$ is the smallest address with $w_o(a) = \text{unwritten}$. Therefore $\mathcal{T}_{\text{obs}} \geq \mathcal{T}_{\text{start}}$.

Upper bound. Suppose instead that $\mathcal{T}_{\text{obs}} > \mathcal{T}_{\text{stop}}$. By definition of $\mathcal{T}(t_{\text{stop}})$, $\mathcal{T}_{\text{stop}}$ is unwritten at time t_{stop}, i.e. $S(t_{\text{stop}}, \mathcal{T}_{\text{stop}}) = \bot$. By write-once monotonicity, if a location is unwritten at some time, it must have been unwritten at all earlier times. Thus for all $t \leq t_{\text{stop}}$,

$$S(t, \mathcal{T}_{\text{stop}}) = \bot.$$

In particular, since $\tau_o(\mathcal{T}_{\text{stop}}) \in [t_{\text{start}}, t_{\text{stop}}]$, we have

$$S\big(\tau_o(\mathcal{T}_{\text{stop}}), \mathcal{T}_{\text{stop}}\big) = \bot,$$

and hence $w_o(\mathcal{T}_{\text{stop}}) = \text{unwritten}$. But this contradicts the fact that $\mathcal{T}_{\text{obs}}$ is the *smallest* address with $w_o(a) = \text{unwritten}$, since $\mathcal{T}_{\text{stop}} < \mathcal{T}_{\text{obs}}$. Therefore $\mathcal{T}_{\text{obs}} \leq \mathcal{T}_{\text{stop}}$.

Combining the two bounds yields $\mathcal{T}_{\text{obs}} \in [\mathcal{T}_{\text{start}},\ \mathcal{T}_{\text{stop}}]$. □

Lemma ([LD.2] Window-Scan Refinement**).** *Assume the K-window discipline holds, and that the storage layer provides a linearizable operation to read the non-contiguous tail $\mathcal{N}$. Consider a weakTail operation with real-time interval $[t_{\text{start}}, t_{\text{stop}}]$ implemented as follows:*

1. *Read $\mathcal{N}$ once, yielding $\mathcal{N}^*$, with linearization point $t^* \in [t_{\text{start}}, t_{\text{stop}}]$.*
2. *Perform an unordered, non-atomic scan only over the window $[max(0, \mathcal{N}^* - K),\ \mathcal{N}^*]$, reading each address in this window at least once at some time in $[t_{\text{start}}, t_{\text{stop}}]$, and return the set of holes in that window.*

Then the pair $(\mathcal{N}_{\text{obs}}, \mathcal{H}_{\text{obs}})$ returned by the window scan is equal to the pair produced by some full unordered, non-atomic scan whose observations all lie within $[t_{\text{start}}, t_{\text{stop}}]$.

Proof. Let t^* be the linearization point of the read of $\mathcal{N}$, so we observe $\mathcal{N}^* = \mathcal{N}(t^*)$. By the K-window discipline at t^*, the true contiguous tail $\mathcal{T}(t^*)$ lies in $[\mathcal{N}^* - K,\ \mathcal{N}^*]$, all addresses $< \mathcal{N}^* - K$ are written at t^*, and all addresses $> \mathcal{N}^*$ are unwritten at t^*. If $\mathcal{N}^* < K$, we interpret the window $[\mathcal{N}^* - K,\ \mathcal{N}^*]$ as $[max(0, \mathcal{N}^* - K),\ \mathcal{N}^*]$, since addresses below 0 are outside the array.

Construct a hypothetical full unordered, non-atomic scan as follows: for each address in $[\mathcal{N}^* - K,\ \mathcal{N}^*]$, use the actual read (time and result) performed by the window-scan implementation; for each address $< \mathcal{N}^* - K$, imagine a read at time t^* returning "written"; and for each address $> \mathcal{N}^*$, imagine a read at time t^* returning "unwritten". All these reads occur within $[t_{\text{start}}, t_{\text{stop}}]$ since t^* does, so this is a valid full unordered, non-atomic scan in the sense of our *weakTail* specification.

By construction, the non-contiguous tail in this hypothetical full scan is exactly the non-contiguous tail observed by the window-based implementation; and the set of holes observed in the full scan is also equal to that returned by the window-based implementation. Thus there exists a full unordered, non-atomic scan whose observations all lie within $[t_{\text{start}}, t_{\text{stop}}]$ and whose $(\mathcal{N}_{\text{obs}}, \mathcal{H}_{\text{obs}})$, as defined by the WEAKTAIL semantics, coincide exactly with those returned by the window-based implementation.

□

A.3 StripedLogDrive Implementation

Let $S \geq 1$ be the number of stripes. Each stripe D_i, for $i \in \{0, \ldots, S-1\}$, is an independent LogDrive instance with local address set

$$A^{\text{loc}} = \{0, 1, \ldots, N\},$$

where the local address N is a sentinel that is never written ($S_i(t, N) = \perp$ for all t), and addresses $0, \ldots, N-1$ are the real data slots. We write

$$A^{\text{real}} = \{0, 1, \ldots, N-1\}$$

for the set of real local addresses.

The global address space of a striped LogDrive is the finite set

$$A^{\text{glob}} = \{0, 1, \ldots, SN\},$$

where addresses $0, \ldots, SN-1$ are striped across the S underlying instances and address SN is a global sentinel that is never written.

For a global address $a \in \{0, \ldots, SN-1\}$ we define

$$\text{stripe}(a) = a \bmod S, \qquad \text{local}(a) = \left\lfloor \frac{a}{S} \right\rfloor.$$

Then $\text{stripe}(a) \in \{0, \ldots, S-1\}$ and $\text{local}(a) \in A^{\text{real}}$ for all $a < SN$. At time t, the global state is

$$S^{\text{glob}}(t, a) = \begin{cases} S_{\text{stripe}(a)}\big(t, \text{local}(a)\big) & \text{if } a < SN, \\ \perp & \text{if } a = SN. \end{cases}$$

Read/write semantics. A striped LogDrive supports global operations $\text{WRITE}(a, v)$ and $\text{READ}(a)$ only for $a \in \{0, \ldots, SN-1\}$, by delegating to the corresponding stripe:

$$\text{WRITE}(a, v) = \text{WRITE}_{\text{stripe}(a)}\big(\text{local}(a), v\big).$$

$$\text{READ}(a) = \text{READ}_{\text{stripe}(a)}\big(\text{local}(a)\big).$$

The global sentinel address SN is never written and is never the target of a read.

Algorithm 1 StripedLogDrive Implementation

```
1:  procedure StripedWrite(a, v)                          ▷ 0 ≤ a < SN
2:      i ← a mod S;    ℓ ← ⌊a/S⌋
3:      D_i.write(ℓ, v)
4:  end procedure

5:  procedure StripedRead(a)                              ▷ 0 ≤ a < SN
6:      i ← a mod S;    ℓ ← ⌊a/S⌋
7:      return D_i.read(ℓ)
8:  end procedure

9:  procedure StripedWeakTail(K)
10:     assert K mod S = 0
11:     K_i ← ⌊K/S⌋ for all i ∈ {0, ..., S − 1}
12:     for all i ∈ {0, ..., S − 1} in parallel do
13:         (N_obs^(i), H_obs^(i)) ← D_i.weakTail(K_i)
14:     end for
15:     T_cand ← {SN}
16:     for i ← 0 to S − 1 do
17:         if N_obs^(i) < N then
18:             T_cand ← T_cand ∪ { S · N_obs^(i) + i }
19:         end if
20:     end for
21:     N_obs^glob ← min T_cand
22:     H_obs^glob ← ∅
23:     for i ← 0 to S − 1 do
24:         for all a ∈ H_obs^(i) with a < N do
25:             H_obs^glob ← H_obs^glob ∪ { S · a + i }
26:         end for
27:     end for
28:     return (N_obs^glob, H_obs^glob)
29: end procedure
```

Disjointness Assumption. Each stripe D_i is an independent LogDrive instance with no shared state.

Lemma (**[SLD.0]** Linearizability of StripedLogDrive)**.** *StripedLogDrive READ/WRITE operations are linearizable: there exists a total order of all completed operations, consistent with real-time precedence, whose behavior matches that of an abstract LogDrive.*

Proof. By specification, each global READ/WRITE on address a is routed to a single stripe and stripes do not interfere. Since each stripe is a linearizable LogDrive, its operations admit a linearization order. Because the per-stripe histories are independent, these orders can be merged while preserving real-time precedence. The resulting total order satisfies the abstract LogDrive semantics, establishing linearizability. □

Theorem (**[SLD.1]** Striped LogDrive Refines the LogDrive Specification)**.** *Assume K is divisible by S, and that each stripe D_i is a correct LogDrive satisfying the abstract specification*

of §A.1, including the WEAKTAIL(K/S) *semantics. Then the striped LogDrive defined above:*

1. *provides linearizable* READ/WRITE *operations; and*
2. *provides a correct* WEAKTAIL(K) *implementation whose return value coincides with that of some unordered, non-atomic scan over* A^{glob} *in the sense of §A.1.*

Thus the striped LogDrive is a correct implementation of the abstract LogDrive.

Proof. Linearizability of reads and writes follows from A.3.

***Correctness of* WEAKTAIL.** Consider a striped WEAKTAIL(K) operation o with real-time interval $[t_{\text{start}}, t_{\text{stop}}]$. By the StripedLogDrive implementation, for each stripe $i \in \{0, \ldots, S-1\}$ the client invokes

$$D_i.\text{WEAKTAIL}(K/S),$$

where K/S is an integer by assumption. Since D_i is a correct LogDrive, the abstract WEAKTAIL(K/S) specification guarantees that there exists a (hypothetical) local unordered, non-atomic scan o_i over the local address space $A^{\text{loc}} = \{0, \ldots, N\}$ whose per-address observations all occur within $[t_{\text{start}}, t_{\text{stop}}]$, and such that applying the unstriped WEAKTAIL semantics to o_i yields exactly the pair $(\mathcal{N}_{\text{obs}}^{(i)}, \mathcal{H}_{\text{obs}}^{(i)})$ returned by D_i .WEAKTAIL(K/S).

We now construct a global unordered, non-atomic scan o^{glob} over the global address space $A^{\text{glob}} = \{0, 1, \ldots, SN\}$. For each stripe $i \in \{0, \ldots, S-1\}$ and each real local address $a \in A^{\text{real}} = \{0, \ldots, N-1\}$, the scan o^{glob} performs a read of the global address $A = S \cdot a + i$ at the same time and with the same return value as in the local scan o_i for address a. For the global sentinel address SN, the scan o^{glob} performs a single read at any time in $[t_{\text{start}}, t_{\text{stop}}]$, returning $\perp$. All per-address observations of o^{glob} thus lie within the interval $[t_{\text{start}}, t_{\text{stop}}]$ and are consistent with the striped LogDrive state, so o^{glob} is a valid global unordered, non-atomic scan in the sense of §A.1.

By construction, for every i and $a \in A^{\text{real}}$, the global address $A = S \cdot a + i$ is unwritten in o^{glob} if and only if the corresponding local address a is unwritten in the local scan o_i. Applying the unstriped WEAKTAIL semantics *globally* to o^{glob} therefore yields:

- a global observed non-contiguous tail

$$\mathcal{N}_{\text{obs}}^{\text{glob}} = \min\Big(\big\{S \cdot a + i \;\big|\; i \in \{0, \ldots, S-1\}, \; a = \mathcal{N}_{\text{obs}}^{(i)} < N\big\} \;\cup\; \{SN\}\Big),$$

- and a global observed hole set

$$\mathcal{H}_{\text{obs}}^{\text{glob}} = \bigcup_{i=0}^{S-1} \left\{ S \cdot a + i \;\middle|\; a \in \mathcal{H}_{\text{obs}}^{(i)} \cap A^{\text{real}} \right\}.$$

Algorithm 2 QuorumLogDrive Implementation: Write and Read

```
1:  procedure QuorumWrite(a, v)
2:      for all j ∈ {0, ..., N − 1} in parallel do
3:          send write(a, v) to D_j
4:      end for
5:      wait until ack received from at least Q_w replicas
6:      return
7:  end procedure

8:  procedure QuorumRead(a)
9:      choose any set R_r ⊆ {0, ..., N − 1} with |R_r| = Q_r
10:     for all j ∈ R_r in parallel do
11:         x_j ← D_j.read(a)
12:     end for
13:     if some x_j ≠ ⊥ then
14:         pick any non-⊥ value v among {x_j}_{j∈R_r}
15:         send write(a, v) to enough replicas to ensure at
        least Q_w written
16:         return v
17:     else
18:         return ⊥
19:     end if
20: end procedure
```

These are exactly the values computed and returned by the StripedWeakTail algorithm. Hence for this striped WEAKTAIL(K) operation o, there exists a global unordered, non-atomic scan o^{glob} whose observed non-contiguous tail and hole set coincide with the striped implementation's return value $(\mathcal{N}_{\text{obs}}^{\text{glob}}, \mathcal{H}_{\text{obs}}^{\text{glob}})$.

Conclusion. The striped LogDrive provides linearizable read/write semantics and a correct WEAKTAIL(K) implementation, and therefore refines the abstract LogDrive specification. □

A.4 QuorumLogDrive Implementation

A.5 The QuorumLogDrive

We implement a replicated LogDrive over N independent LogDrive replicas $D_0, \ldots, D_{N-1}$. For each replica D_j we assume the abstract LogDrive specification of §A.1 holds, including linearizable READ/WRITE and WEAKTAIL(K) semantics. The QuorumLogDrive is configured with:

- a write quorum size Q_w, where $1 \le Q_w \le N$;
- a read quorum size $Q_r = N - Q_w + 1$;
- a fault-tolerance quorum size $Q_f = \max(Q_w, Q_r)$.

Lemma (**[QLD.0]** Linearizability of QuorumLogDrive Reads / Writes)**.** *Under the assumptions of §A.5, the QuorumLogDrive* READ/WRITE *operations are linearizable: there exists a*

total order of all completed operations, consistent with real-time precedence, whose behavior matches that of an abstract LogDrive.

Proof. Fix an address a. Because replicas $D_0, \ldots, D_{N-1}$ are independent LogDrive instances, each replica's local READ/WRITE operations on a are linearizable with respect to its own state.

A global WRITE(a, v) sends WRITE(a, v) to all N replicas and waits for acknowledgments from a write quorum of size Q_w. We linearize this write at the real-time instant when the Q_w-th replica durably stores v at address a. After that instant, at least Q_w replicas contain v at a and, by the single-valued assumption, no replica ever stores a different value at a.

A global READ(a) selects a read quorum R_r of size $Q_r = N - Q_w + 1$ and performs a local READ(a) at each replica in R_r, obtaining responses $\{x_j\}_{j \in R_r}$. There are two cases:

- If all $x_j = \perp$, then no completed write to a can exist: any completed write would have updated a write quorum of size Q_w, and since $Q_w + Q_r > N$, the read quorum R_r would intersect that write quorum at a replica returning $v \neq \perp$. Thus the abstract LogDrive state at a is $\perp$ throughout the read's interval, and we can linearize the read at any time within its interval, returning $\perp$.
- Otherwise, some $x_j = v \neq \perp$. Let WRITE(a, v) be the latest completed write to a in the write linearization order. At its linearization point, at least Q_w replicas store v at a, and by the single-valued assumption no later write stores a different value. Since $Q_w + Q_r > N$, the read quorum R_r intersects this write quorum at some replica that stores v, so the read observes v. The QuorumLogDrive picks some non-$\perp$ value v among the x_j and optionally performs a read-repair step that sends WRITE(a, v) to additional replicas until at least Q_w replicas store v. We linearize the read at any time between the linearization point of WRITE(a, v) and the time when it has collected its Q_r responses. The subsequent read-repair writes occur after this linearization point and write the same value v, so they do not change the value of a in the abstract LogDrive and do not affect correctness.

In both cases the read's return value matches the value stored at a in the abstract LogDrive at its linearization point, and real-time precedence is preserved. Since operations on different addresses are independent and each replica is linearizable, we can take the union of these per-address linearizations to obtain a global linearization order over all READ/WRITE operations.

Thus the QuorumLogDrive provides linearizable writes and reads. □

Theorem ([QLD.1] QuorumLogDrive Refines the LogDrive Specification**).** *Under the assumptions of §A.5, the QuorumLogDrive provides linearizable READ/WRITE operations and a correct* WEAKTAIL(K) *implementation. Hence it refines the abstract LogDrive.*

Algorithm 3 QuorumLogDrive Implementation: WeakTail

```
procedure QuorumWeakTail(K)
    choose any set R_f ⊆ {0, ..., N − 1} with |R_f| = Q_f
    for all j ∈ R_f in parallel do
        (N_obs^(j), H_obs^(j)) ← D_j.weakTail(K)
    end for
    N_max ← max_{j∈R_f} N_obs^(j)
    L ← max{0, N_max − K}                ▷ window: only K slots before N_max
    ▷ Classify global written/unwritten status for slots in [L, N_max)
    for a ← L to N_max − 1 do
        cntWritten ← 0;   cntUnwritten ← 0
        for all j ∈ R_f do
            if a ≥ N_obs^(j) or a ∈ H_obs^(j) then
                cntUnwritten ← cntUnwritten + 1
            else
                cntWritten ← cntWritten + 1
            end if
        end for
        if cntUnwritten ≥ Q_r then
            globalStatus[a] ← unwritten
        else if cntWritten ≥ Q_w then
            globalStatus[a] ← written
        else      ▷ ambiguous: repair and treat as written
            pick any replica j* ∈ R_f with local written at a
            v ← value of slot a at replica j*
            replicate v to enough replicas to ensure at least Q_w written
        (e.g., send write(a, v) to further replicas)
            globalStatus[a] ← written
        end if
    end for
    ▷ Derive global non-contiguous tail and hole set from globalStatus
    N_obs^glob ← min{a | a ≥ L, ∀b > a : globalStatus[b] = unwritten}
    H_obs^glob ← { a | L ≤ a < N_obs^glob, globalStatus[a] = unwritten }
    return (N_obs^glob, H_obs^glob)
end procedure
```

Proof. Linearizability of READ/WRITE follows immediately from Lemma QLD.0. We prove correctness of WEAKTAIL(K).

Let operation o run during $[t_{\text{start}}, t_{\text{stop}}]$. It queries a quorum R_f of size Q_f and obtains from each replica $j \in R_f$ a valid local WEAKTAIL(K) observation $(\mathcal{N}_{\text{obs}}^{(j)}, \mathcal{H}_{\text{obs}}^{(j)})$, each corresponding to some local unordered scan o_j whose observations fall within o's interval.

Let $\mathcal{N}_{\max} = \max_{j \in R_f} \mathcal{N}_{\text{obs}}^{(j)}$ and $L = \max\{0, \mathcal{N}_{\max} - K\}$. By the global K-window discipline, all $a < L$ are globally written throughout o.

For each $a \in [L, \mathcal{N}_{\max})$, classify the status of a using the quorum rule:

$$\text{status}(a) = \begin{cases} \text{unwritten} & \text{if at least } Q_r \text{ replicas in } R_f \text{ report } a \text{ unwritten,} \\ \text{written} & \text{if at least } Q_w \text{ replicas in } R_f \text{ report } a \text{ written,} \\ \text{written} & \text{otherwise (after performing repair to ensure } Q_w \text{ replicas store the unique value).} \end{cases}$$

Because the address space is single-valued and $Q_w + Q_r > N$, this classification is consistent with the linearizable write-once semantics: a completed write to a implies at least Q_w replicas store the value v, so no read quorum can fail to observe v; conversely, if a read quorum observes a as unwritten, then no write quorum has completed. Repair writes only propagate existing values and occur within o's interval, so they preserve linearizability.

Define a hypothetical global unordered scan o^{glob} by assigning each address a a written/unwritten observation consistent with $\text{status}(a)$ at some time in $[t_{\text{start}}, t_{\text{stop}}]$; this is possible since each case of $\text{status}(a)$ corresponds to a quorum condition that holds at some time during o. For $a < L$, mark a written; for $a \geq \mathcal{N}_{\max}$, mark a unwritten, consistent with all local observations.

Applying the abstract weakTail semantics to o^{glob} yields exactly

$$\mathcal{N}_{\text{obs}}^{\text{glob}} = \min\{a \mid \forall b > a : \ \text{status}(b) = \text{unwritten}\},$$

and

$$\mathcal{H}_{\text{obs}}^{\text{glob}} = \{a < \mathcal{N}_{\text{obs}}^{\text{glob}} \mid \text{status}(a) = \text{unwritten}\},$$

which coincide with the QuorumLogDrive output.

Thus the implementation refines the abstract weakTail(K) specification, completing the proof. □

ARTIFACT EVALUATED
USENIX
AVAILABLE

Timelock Drive: Isolated Time-Based Defense for Storage Systems

Jonah Rosenblum
University of Michigan

Juechu Dong
University of Michigan

Peter M. Chen
University of Michigan

Satish Narayanasamy
University of Michigan

Abstract

Data is one of the most critical assets for organizations and individuals, yet its integrity is increasingly threatened by ransomware, data tampering, sabotage, and intentional data vandalism. While most organizations rely on backups, backups themselves are also vulnerable. Even a fully secure software system may be insufficient, as nearly two-thirds of attacks exploit human errors to steal access credentials.

We present Timelock Drive (TD), which allows users to timelock a physical disk block for a duration during which the stored data cannot be modified by anyone, including credentialed users. We show that this enables a secure backup system where the versioning system itself is not part of the TCB. Only a small, isolated checker remains in the TCB, and we formally verify it. A critical challenge we address is maintaining metadata without overwriting prior state using a pure append-only design. We solve the performance problem of scanning logs to retrieve metadata by offloading metadata management to the untrusted host while ensuring security through integrity checks.

Our experiments show that TD incurs negligible space, performance, and storage I/O overheads compared to conventional versioning systems.

1 Introduction

Data is one of the most precious commodities owned by businesses and consumers, and safeguarding it is paramount. Unfortunately, the integrity[1] of critical data is under attack by ransomware [39, 51], data tampering and sabotage [2], and data vandalism. Ransomware in particular has become pervasive: in 2024, 67% of hospital organizations were hit [51], and ransomware attacks are projected to cost victims $265 billion annually by 2031 [39].

The most common way to defend against integrity attacks is to keep regular backups using a versioning system (VS) [8, 9, 14, 33]. Unfortunately, this defense has proven insufficient in practice because the backup system itself often shares the same vulnerabilities as the host: bugs in backup software, compromised administrator credentials, and insider attacks all allow adversaries to corrupt or delete backups [51]. In fact, recent surveys show that 95% of ransomware-hit organizations reported that attackers attempted to compromise their backups, and two-thirds of those attempts succeeded [51].

To defend against these vulnerabilities, researchers have proposed adding retention policies that prevent recently created versions from being deleted for some fixed period of time [18, 45]. Similar to the timelocked safes used in banks, such policies force attackers to wait before damaging recently written data. While a patient attacker could wait out the retention interval, forcing such delays increases the chance of detecting the intrusion before damage occurs and makes opportunistic attacks far less attractive.

Unfortunately, adding a timelock policy to a backup or versioning system is insufficient, as the VS itself may be compromised. Bugs or stolen administrator credentials allow an attacker to override or bypass the retention policy—an issue made concrete by recent trimming attacks that bypass timelock logic inside storage-side VS systems [36].

In this paper, we propose a more fundamental approach: a Timelock Drive (TD) that enforces timelocks directly on physical disk blocks. The TD exposes a simple storage interface but provides a strong security guarantee called **transient immutability**: each block becomes immutable for a fixed period after it is written to the drive, and no software component, including a fully compromised VS or administrator, can override this protection.

All accesses to TD pass through a small, isolated microcontroller, called the TD checker, which enforces the timelock constraints. The TD provides a simple storage interface that supports reads, writes, and timelock operations, while the versioning system (VS) runs on the untrusted host and is entirely outside the trusted computing base. This separation provides a clear division of responsibility: the TD guarantees that blocks cannot be modified for the timelocked duration, while the VS is free to manage data in any way as long as it does not attempt to overwrite data before its timelock expires (as such attempts will fail). Decoupling the two components greatly reduces the amount of trusted code required on the TD, and as a result, the TD checker is only about 400 lines of code. Its small size allowed us to formally verify the correctness of

[1] Confidentiality is also important, but is outside the scope of this paper.

the timelock mechanism.

A critical constraint of the TD is that it forbids overwriting any physical block during its timelock. Yet existing file systems and versioning systems depend on metadata overwrites to maintain version indices, free space, and consistency state. For example, LFS periodically updates its checkpoint region [41], and systems such as FlashGuard and S4 overwrite version metadata [18,45]. A VS running on top of TD cannot perform these updates, since any metadata overwrites would violate timelock immutability and be rejected, creating a fundamental mismatch between traditional VS design and the non-overwriting constraint imposed by a TD.

In fact, even the TD system itself is not allowed to overwrite any timelocked physical blocks; it must instead maintain and update its own metadata state without overwriting timelocked physical blocks. This leads to the central question of this paper: **Can we build both a usable TD checker and a performant versioning system on top of a storage device that forbids all overwrites for a fixed period, including overwrites of their own metadata?**

Our solution is to store TD metadata as a purely append-only log, the TD-log, on the drive. Every update creates a new log entry, which are coalesced and written at the block granularity. Each TD-log entry includes a timelock field that protects the physical addresses referenced in the entry, including the log block itself, eliminating the need for additional metadata about the metadata. While this design is non-overwriting, retrieving TD metadata from the drive requires scanning the entire log – an approach with an unacceptable overhead.

To address this, we use a **delegate-but-verify** strategy. The untrusted host maintains a full, in-DRAM cache of TD metadata, along with a cryptographic BLAKE3 hash generated by the TD checker. On each write, the host supplies the relevant metadata and hash; the TD checker verifies integrity before proceeding. The checker itself stores only a small set of freshness counters (~ 2 MB per 1 TB disk), enabling a scalable design. Because the host provides all metadata during normal operation, the TD checker never needs to read TD-log entries from disk. The on-disk TD-log is read only during recovery, which is infrequent and can tolerate a full scan. This approach yields an efficient metadata solution by trading rare recovery-time cost for non-overwrite property.

We build a secure, non-overwriting versioning system (VS) on top of an isolated TD. The VS runs entirely on the untrusted host; even if it is fully compromised, the system can always recover to a pre-intrusion state. This is because TD's transient immutability ensures that checkpoints created before the intrusion remain unmodified until their timelocks expire.

To respect TD's no-overwrite constraint, the VS records all of its metadata in TD as append-only logs. For performance, it maintains an in-memory logical-to-physical map on the host. From the TD's perspective, VS metadata and data are indistinguishable and subject to the same timelock protection.

During recovery, the only trusted input is the content stored on the TD. For each logical address, the system retrieves the most recent version written before the suspected attack time and discards those written after, thereby neutralizing spoofed entries. To make this possible, the TD provides a **time-of-lock** security guarantee: recovery can reliably determine the time-of-lock for every block, as the TD checker records each block's time-of-lock in the TD metadata log.

We implement the VS as a block device driver that can be mounted by ext4, formally verify the TD checker in Dafny [24], and prototype the TD on a Raspberry Pi interposed between the host and a storage device. Reads incur no overhead, and writes benefit from secure host-side metadata caching. Across a variety of storage traces and filesystem workloads [46], TD introduces negligible overhead (0.4% execution, 0.5% throughput for SSDs).

Our paper makes the following contributions:

- We design and implement a secure Timelock Drive (TD) that guarantees transient immutability for a physical block. All information (data and its metadata) are timelocked for the duration specified by the host (user).
- We design and evaluate a TD and VS system that does not require overwrites to timelocked data and metadata.
- We isolate the TD checker from the rest of the system, including the versioning system. It has a narrow interface and a small TCB (∼400 LoC), which enables formal verification.
- We demonstrate that TD incurs negligible performance and space overhead by securely caching TD metadata on the untrusted host.

2 Background and Threat Model

In this section, we describe a threat model for data integrity attacks and the requirements for secure backups.

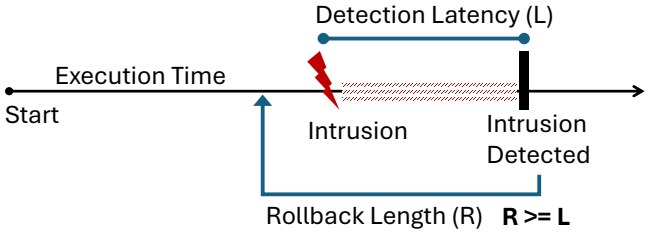


Figure 1: Recover by restoring state before the intrusion.

Figure 1 illustrates a simplified timeline of a data integrity attack and recovery using a continuous backup system. Before an intrusion, a VS periodically checkpoints data on a storage device. After intrusion, we assume that an attacker has complete control over the host software stack, including privileged users, the OS, filesystem, drivers, and the versioning system.

At some point, the intrusion is detected (in the worst case, after the attack is completed). We define the latency between

intrusion and detection as intrusion detection latency (L), as shown in Figure 1.

The fundamental property of secure backups is the ability to roll back, restoring the storage state to *before* the time of the intrusion. We refer to this as rollback length (R). To defend against intrusions, the rollback length must exceed the expected intrusion-detection latency (L).

Need for time-based defenses: A timelock adds a significant hurdle to an attacker who compromises the host system. The value of a timelock is threefold: (1) attackers are forced to wait much longer before asking for a ransom, and during this waiting period they risk being detected before they can damage stored data [3, 28, 32], (2) a safety window provides administrators with more time to discover exploits and deploy patches, (3) expensive deep scans that detect intrusions, invalid/corrupted data, etc. can run less frequently and analyze changes in system state over time. Timelocks allow administrators to trade extra storage space for greater security guarantees. With unlimited storage capacity, an infinite timelock will indefinitely prevent overwrites of protected data; more realistically, having enough capacity to store six months of old data poses a massive challenge for ransomware. According to a recent report [1] from Google, the median intrusion detection latency in 2024 was only five days, and $\geq 99\%$ of intrusions are detected within 5 months.

Isolation for Security: Isolation is a well-known security concept that reduces attack surfaces and minimizes the impact of software bugs. For example, in OpenSSH, isolating authentication logic from pseudo-terminal management and running them at different privilege levels prevents privilege escalation attacks [35]. Another example is Page Table Isolation, which separates user and kernel page-table mappings to protect against Meltdown attacks [26, 47]. Isolation is a key design principle in microkernels [5, 25], web browsers [37, 38], and hypervisors/VMs [4, 42].

Prior S4 systems [18, 45, 50] integrate time-based security checks into a VS on the storage device. Recent work has shown that vulnerabilities in version management can bypass timelock checks and delete old versions, violating the time-based defenses [36]. Our work aims to simplify the timelock primitive to the smallest TCB possible and isolate it from all other components of the secure backup system.

Orthogonal threats: Data exfiltration happens when an attacker can siphon off sensitive data, which they can use for their own gain or threaten to release for a ransom. Such attacks only require read permission. The proposed defenses [15, 27, 40] against data exfiltration are orthogonal to Timelock Drive and thus can be used in conjunction with each other. For example, time-release encryption schemes [27, 40] can prevent plaintext leakage while Timelock Drive guarantees the ciphertext can't be overwritten.

Another threat model includes physical access to the storage system. These powerful adversaries can steal the physical storage device and/or damage it beyond repair. Our work focuses on defending against digital security threats.

TD Command	Inputs	Outputs/Impact	ATA Command
read	addr	Data @ addr	(0x25) READ DMA EXT
write	addr, data, md	Write data @ addr if free.	(0x35) WRITE DMA EXT
timelock-update	addr, TD log block, TD hash	Check validity of timelock/unfreeze operations in TD log using hash. Write TD log block at specified addr.	(0x49) Unused opcode
get-next-td-hash	start-bit	Reads next block in TD metadata log, updates freshness counters, returns secure hash. Returns null at the end	(0x28) Unused opcode
identify	N/A	Disk model, serial number, geometry, etc.	(0xEC) IDENTIFY DEVICE
sync	N/A	Flushes drive buffers (write barrier)	(0xE7) FLUSH CACHE

Table 1: TD-checker Controller's Modified ATA Interface

Clock Assumptions: In this work, we assume our TD-checker has a monotonically non-decreasing clock, even across restarts/crashes. This is similar to non-volatile secure counters in trusted platform modules (TPMs) [30]. If the checker loses power, the clock stops, so all timelock mutability countdowns simply pause until power is restored. This does not weaken the immutability guarantee; it only delays deletion. This assumption is not unique to our work; all S4 systems [18, 45, 50] make the same assumptions about clocks.

3 Timelock Drive

This section presents the Timelock Drive (TD), a standard SSD or HDD whose accesses pass through a lightweight, formally verified, TD-checker controller. The TD augments traditional credential-based security with a temporal protection mechanism: once a user writes to an address and issues a timelock, that block cannot be modified by anyone, including privileged software, until the lock expires.

This section addresses the core challenge of building a performant TD checker on a storage device that blocks all overwrites for a fixed period, even of its own metadata. Section 4 describes how we build a secure backup system on top of Timelock Drive.

3.1 Timelock Drive Interface

A conventional storage device is enhanced with a formally verified TD-checker controller (Fig. 2a), which guards and mediates all access. We refer to this augmented device as the Timelock Drive (TD). It presents a narrow command interface to the host, listed in Table 1.

The **read** and **identify** commands behave like their conventional counterparts and carry no additional TD constraints.

The TD-checker permits a **write** command to a block only when the timelock constraint is satisfied. The additional parameters (md, md-hash) enable the TD-checker to validate this constraint (described later in Section 3.4).

The **timelock** operation allows users to protect a block for a duration δ, enforcing a **transient immutability** guarantee that prevents modification of the block for at least δ time.

A key interface question is when the timelock countdown should begin. If the countdown starts at the moment of timelock, the user must somehow predict how long the data will remain useful. In practice, users rarely know this. For instance,

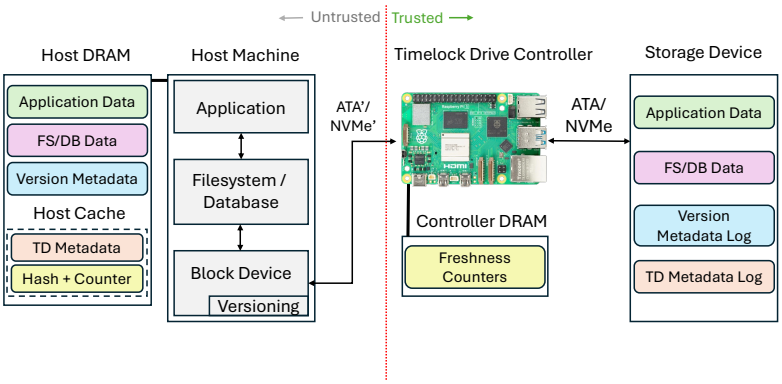


Figure 2a. Timelock Drive system overview.

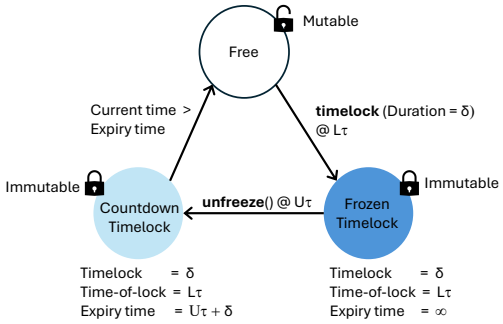


Figure 2b. States of a storage block.

the lifetime of a versioned block lasts until the next version is written; a timing that cannot be known in advance.

A simple workaround is to refresh timelocks for all live blocks before they expire, but doing so would impose heavy performance overhead. Instead, TD uses two states for a timelocked block. After a block is timelocked, it enters a frozen state in which it is protected indefinitely. Later, the user may explicitly **unfreeze** the block. Once unfrozen, TD prevents writes for the duration specified at the original timelock (δ).

This design preserves transient immutability: the block remains protected for the full timelock interval *after* being released. Even if an adversary gains control immediately after a write and attempts to unfreeze the block, they must still wait out the timelock before making any modification.

Figure 2b illustrates the finite-state machine for a disk block. A timelocked block transitions from free to frozen, disallowing further writes. When unfrozen, it moves to the countdown state, where writes remain disallowed. Once the timelock duration expires, the block returns to the free state and can be written again. The timelock operation records a time-of-lock timestamp in the block's metadata, which helps prevent adversarial spoofing during recovery (Section 4.7).

The **timelock-update** command applies timelock/unfreeze operations on a host-specified list of addresses. The motivation for bulk update is further explained in Section 3.3.

Finally, the **get-next-td-hash** command rebuilds the host's TD metadata cache after a checker reset or intrusion recovery. With the `start` bit set, the checker begins scanning the TD metadata log from the beginning; each subsequent call processes the next log block, updates the checker freshness counters, and returns the corresponding secure TD hash, returning `null` at the end of the log. The purpose of this command is discussed in Section 3.4.

3.2 Verified Trusted TD-Checker Controller

The TD-Checker controller is a small microcontroller that natively executes the timelock logic. Its inputs are storage-interface commands; in our prototype, we use the "TD-ATA" command set shown in Table 1. The controller approves or rejects each command based on the addressed block's timelock state, allowing only those that satisfy the current timelock constraints to pass through. TD is agnostic to the underlying storage medium (HDD or SSD). We adopt the ATA command interface for its simplicity and widespread use, though the design can be extended to other interfaces, including NVMe.

To support TD commands, the controller has a clock that retains its state across power cycles. The assumptions about this clock are stated in Section 2.

Small and Isolated TCB: The controller and its state (clock and metadata) are isolated from the host system, interacting only through the verified TD-ATA interface. We formally verify (Section 6) that our controller implementation satisfies TD's security guarantees, ensuring that the timelock constraint holds even if the host is compromised. All higher-system components—privileged users, the filesystem, versioning software, drivers, and the OS—are excluded from the TCB. Our verified implementation in Dafny consists of roughly ∼400 lines of code, excluding ghost statements used solely to assist verification.

3.3 Non-Overwrite TD Metadata

TD-checker relies on per-block metadata—timelock state, timelock duration, time-of-lock, and expiry time—to enforce its checks. If no metadata exists for a block, we assume it is in the free state, as is true for all blocks initially.

When a block is timelocked, its metadata is updated and persisted on the drive. This metadata must itself be timelocked; otherwise, an adversary could tamper with it and subvert future TD checks.

A natural design is to store TD metadata along with data within a disk block to timelock it. However, the TD metadata must change when the block transitions from the frozen state to the countdown state, since the recorded state and expiry time change. The controller does not continuously update metadata during the countdown; it only checks, on a later write, whether the recorded expiry time has passed. Still, the frozen-to-countdown transition requires a metadata update, which would be forbidden if the metadata block itself were

timelocked. Thus, TD needs a way to update metadata state without overwriting timelocked metadata.

We address this challenge by representing TD metadata as an append-only log. When a block is timelocked, a metadata entry is appended to the log; when the block is unfrozen, a new entry is appended rather than modifying the old one. Logically, the latest valid entry for a block determines its current TD metadata state. During recovery, the log can be replayed to reconstruct this state; during normal operation, TD avoids expensive log scans by delegating metadata caching to the untrusted host and verifying the supplied metadata, as explained in Section 3.4.

Each log entry stores the TD metadata for a timelocked disk block, and the entry is protected under the same timelock duration as the block it corresponds to. Since writes to disk are at block granularity, each log entry would require an entire block, which is space-inefficient. We solve this problem by extending the TD interface to support **bulk timelock-update**. This allows the user to apply the timelock or unfreeze transition to a list of addresses rather than a single address. While the list size could be 1, for a versioning system, bulk operations work well, as they timelock/unfreeze a set of blocks at the end of a checkpoint interval.

3.4 Delegating TD Metadata Management to Untrusted Host

TD metadata for a block must be read on every write to perform the timelock check. A pure append-only log incurs high latency overhead, since locating the latest metadata for a block requires an expensive scan of the entire log.

We can reduce this overhead by adding a metadata cache inside the TD-checker, but TD metadata exhibits poor temporal locality, so the cache would need to hold nearly all metadata to be effective. This significantly increases controller complexity, and any cache miss still triggers an expensive log scan. Storing all metadata (one metadata block per 512 addresses) consumes roughly $\frac{1}{513}$ of total device capacity – impractical for a small controller.

Instead, we let the untrusted host maintain the TD metadata cache and secure it using cryptographic mechanisms. The TD-checker computes a BLAKE3 HMAC over the metadata, the TD address, a freshness counter, and a 256-bit secret key, and gives it to the host. When writing to a block, the host (user) supplies its corresponding metadata and HMAC to the controller, which verifies the hash and freshness counter.

In fact, we let the host propose metadata updates when issuing a timelock or unfreeze command. The TD-checker's role is to validate the integrity of old metadata and the correctness of new/updated metadata, both supplied by the host.

To maintain freshness, the controller maintains a 4-byte counter for each TD metadata block and includes it in the HMAC. The host locally caches the latest TD metadata and corresponding hashes and supplies them with TD commands. This cache is not trusted: if the host supplies stale metadata or an old hash, the checker recomputes the HMAC using its current freshness counter and rejects the command.

The controller's memory requirement remains modest: it stores only the freshness counters, requiring approximately $\frac{1}{525{,}312}$ of disk capacity, or about 2 MB per 1 TB of storage. Assuming a disk write latency of 10 ms, nonstop writes to a single block would take 1.36 years to overflow a 4-byte counter, making overflows rare.

The TD metadata log is read-only when the cache must be rebuilt, such as after a power reset, a freshness-counter overflow, or intrusion recovery. In these cases, the checker generates a new secret key, disables writes, and reconstructs its freshness counters by scanning the append-only TD metadata log. The host initiates this scan by issuing `get-next-td-hash` with the `start` bit set. Each subsequent `get-next-td-hash` command causes the checker to read the next TD metadata log block, apply the metadata updates in that block to its freshness counters, and return the resulting secure TD hash to the host. When the end of the log is reached, the checker returns `null`; at that point, the host has rebuilt its cache of TD metadata and corresponding hashes.

If multiple log entries refer to the same address, the scan may return multiple hashes for that address. Only the hash corresponding to the latest processed metadata state remains valid, because each subsequent metadata update advances the checker freshness counter. Thus, a host that later supplies an intermediate metadata entry and hash will be rejected as stale.

A random-access command, such as `get-td-hash(addr)`, would be inefficient: the checker would have to scan the TD log to find the latest metadata for that address. Rebuilding all hashes after a reset would therefore require one log scan per address, yielding quadratic recovery time. In contrast, `get-next-td-hash` streams through the log once and rebuilds both the checker's freshness counters and the host's valid hashes in linear time.

3.5 Timelock Drive Policy and Use Cases

Prior solutions that embed timelock policy within the versioning system (VS) [18, 45, 50] offer weaker security guarantees due to a large TCB and lack user-controlled timelocking to prevent overwrites of a physical block. Because the timelock mechanism is intertwined with the VS, users are bound to predefined policies, and any change requires modifying an already large TCB—risking new vulnerabilities.

TD is fundamentally different. It enforces timelocks directly on physical disk blocks and exposes this capability to users through a minimal interface. The VS is built on top of this abstraction and is therefore excluded from the TCB. This separation allows users to implement their own versioning policies without affecting the TCB.

We outline several use cases for Timelock Drive, beyond conventional versioning, that are enabled only due to TD abstraction. Users with highly sensitive data (e.g., cryptocurrency keys) can store them in TD with an infinite timelock. An

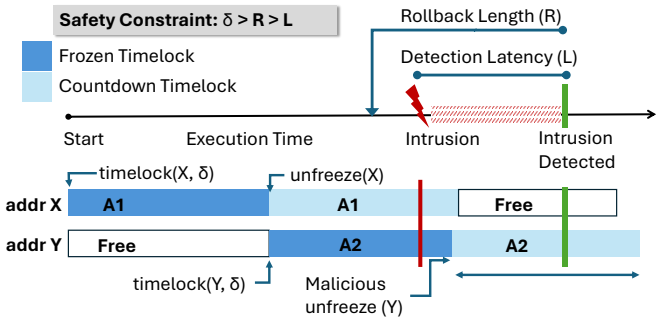


Figure 3: Secure backups using Timelock Drive

OS or middleware can automatically identify important files (based on content or type) and timelock them with durations tailored to importance. Organizations can timelock records to ensure regulatory compliance. TD can also protect logs during an attack, enabling accurate forensic reconstruction and intrusion analysis [10, 11].

While many applications are possible, we focus on one crucial use case for Timelock Drive: secure backups, which we discuss in the next section.

4 Secure Backups Using Timelock Drive

We build a secure, non-overwriting versioning system (VS) atop the isolated Timelock Drive, with the VS running entirely on the untrusted host. Even if the VS is fully compromised, TD's transient immutability preserves all pre-intrusion versions long enough to guarantee recovery to a pre-intrusion state (Section 2).

4.1 Overview

The TD-checker controller exposes an unchanged view of the physical address space to the VS (Figure 4). The VS, running on the untrusted host, then exposes a subset of the physical address space to the client filesystem[2] as the logical address space. The VS manages the placement of TD-metadata, VS-metadata, and the filesystem's data. The TD-checker does not distinguish between VS data and VS-metadata.

When the filesystem writes to a logical address, the VS allocates a free physical block, writes the new version, and timelocks it. It then unfreezes the old version so that, once its countdown timelock expires, the corresponding physical block can be reclaimed.

The VS maintains metadata mapping each logical address to its sequence of physical blocks. Conventional VS designs overwrite metadata on every disk write, but in an untrusted VS, such overwrites could allow an adversary to corrupt or erase version history during an intrusion. Thus, VS metadata must also be timelocked, requiring a design that avoids overwriting during the timelock interval. This section discusses how our VS achieves this using a pure append-only, non-overwriting log; conceptually similar to the TD-log.

A trusted recovery process reconstructs a pre-intrusion system state using only the contents of TD. Finally, we show how standard optimizations such as incremental checkpointing can still be implemented despite the constraints imposed by TD.

[2]Any system that uses a block device—filesystems, databases, object stores, etc.—can run atop TD; we use a filesystem for illustration.

4.2 Secure Untrusted Version System

When the filesystem issues a write, the VS stores the data in a free block in TD and secures it with a timelock set to the rollback length (R). Even if an intrusion happens immediately after the write, the worst an adversary can do is attempt to unfreeze the data. However, due to the timelock, TD ensures the data are protected for the duration of R. Therefore, if the intrusion is detected in less than R, the data can be recovered.

Once data are written to a physical address and timelocked, that physical address remains immutable until unfrozen. VS unfreezes that physical address only when the filesystem writes a new version to the corresponding logical address.

Figure 3 shows an example. When the filesystem writes to a logical address (A), the VS stores the data in a free block (X) in the TD, and applies a timelock of duration R. On a subsequent write to the same logical address (A), the VS writes the new version (A2) to another free physical address (Y), timelocks it, and immediately unfreezes the old version (A1) stored in physical address X. The unfrozen physical address then enters a countdown timelock state, and stays protected for the duration of R.

Version Integrity Guarantee: To establish that we will always be able to safely recover the latest pre-intrusion version for a logical address, let us consider all possible storage states at the time of intrusion for an address.

First, for physical addresses in a free state, there is no valid data and therefore no guarantee is needed.

Second, we consider addresses in a countdown timelocked state. For instance, in Figure 3, physical address X is in this state at the time of an intrusion. This physical block was unfrozen by our VS before the intrusion. While the VS is untrusted, its behavior before an intrusion is benign. Therefore, it would have issued an unfreeze command for block X only after writing a newer version (A2) to another physical block (Y). As a result, even if block X expires after the attack and loses protection, a more recent version (A2 in Y) can still be recovered for the corresponding logical address.

Finally, a physical address (Y) could be in the frozen timelock state, implying that it contains the most up-to-date version for a logical address and was not unfrozen before the intrusion. In this state, the worst an adversary can do is unfreeze the physical address shortly after the intrusion, as shown in Figure 3. This would only transition the physical address into a countdown timelock state, where it remains protected for the duration of the timelock (R). Therefore, as long as the timelock duration is set long enough to outlast the time needed to detect the intrusion, we can recover a valid version for A.

4.3 Incremental Checkpoint on TD

Creating a version for every write can greatly increase storage overhead. A conventional solution is to divide execution into epochs and take a copy of the checkpoint at the end of each epoch. An incremental checkpoint [16, 34] typically only maintains the most recent version of a logical address written within its epoch. If an address was not written within an epoch, then the checkpoint has no versions for it.

Our VS implements incremental checkpoints as follows. For each write to a logical address within an epoch, the VS allocates a new TD block and stores the version without a timelock. At the end of the epoch, the VS timelocks (for duration R) the set of blocks containing the most recent versions of all logical addresses written during that epoch. It also unfreezes old versions corresponding to these logical addresses.

The only limitation of incremental checkpointing is that, rather than always recovering to the last write before the intrusion, we roll back to the most recent checkpoint/epoch before the intrusion. We use small epoch intervals (one hour), which is negligible compared to the intrusion detection latency (weeks). Given that we may lose all versions produced after an intrusion, incremental checkpointing would additionally lose at most one epoch's worth of data, which is again negligible.

4.4 Non-Overwriting Version Metadata

We assume that an attacker can gain full control over the VS after an intrusion. To enable safe recovery, we must apply the same timelock protection to both data and the version metadata. This is a novel challenge, as all prior VS we are aware of, including "append-only" systems [9, 14, 33, 41], modify version metadata. For example, LFS periodically overwrites checkpoint regions every 30 seconds [41].

Our VS metadata maps logical addresses to physical addresses. Since we cannot overwrite the map during updates, a naive solution is to track changes in version metadata by adding additional versioning. However, to do this, we would need a second layer of metadata on top of our version metadata. Future updates encounter the same issues, necessitating the addition of a third layer of metadata, followed by a fourth, and so on. This becomes an infinite recursion problem.

We solve this recursive problem by using an append-only log that tracks the chronological updates of versions. When the filesystem writes to a logical address, the VS will create a new version and store it in a free physical address. At the same time, a new metadata entry is added to the end of the log that describes the mapping between the logical and physical addresses. When enough log entries are created (or enough time has passed), a block of log entries is stored in TD with a timelock greater than or equal to the underlying data. Each log block is linked together through a series of forward pointers with the head of the log stored at a predetermined location in the physical address space. This data structure allows us to track all changes to our metadata (versions of versions) without triggering the recursive update problem [41].

4.5 Bulk Metadata Update

Each VS metadata log entry is small, so we aggregate multiple log entries into a metadata disk block during an epoch. At the end of an epoch, the VS metadata are written to the drive. When the VS is under heavy write load, the metadata block may fill up quickly before an epoch ends. In that case, more than one metadata disk block would be written at the end of that epoch.

At the end of an epoch, VS timelocks and unfreezes a set of blocks in bulk (Section 3.3). This aggregates TD metadata and writes it to the disk only at the end of each epoch.

4.6 Crash Safety

Users rely on crash safety to recover a valid system state after sudden crashes, such as power loss [29]. Prior systems that embed timelocks into the VS do not provide explicit crash-safety guarantees or require a battery backup for each device [18, 19, 50]. This means that even with a crash-safe filesystem, prior solutions can still lose data and/or timelock protections after a crash.

A write barrier, implemented through synchronization operations such as `fsync` or disk flushes, provides a persistence-ordering point: writes issued before the barrier must reach stable storage before writes issued after the barrier are allowed to become durable. TD preserves the crash-safety guarantees of the client filesystem by forwarding filesystem writes and write barriers (ATA sync) to the underlying disk in the same order in which they are received. However, the VS also creates additional persistent state, including version metadata and TD metadata, and this state must be ordered carefully with respect to the data it describes.

For each committed epoch, the VS persists updates in three ordered steps. First, it writes the data blocks. It then inserts a write barrier before writing the corresponding version metadata. This ordering is required in any versioning system: recovery must never observe version metadata that points to data blocks that were not durably written. Second, after writing the version metadata, the VS inserts another write barrier before writing the TD metadata and timelock records. This ordering is specific to TD: recovery must not observe timelock metadata for an epoch unless the version metadata for that epoch is also durable. Otherwise, a crash could leave TD metadata durable while the corresponding version metadata is missing or stale, causing recovery to associate timelock state with stale data. Finally, after writing the TD metadata, the VS issues a final write barrier to commit the epoch. If a crash occurs before this final barrier completes, the incomplete epoch is discarded during recovery.

4.7 Spoof-free Guarantee for Recovery

Prior systems [18, 45, 50] implicitly trust their versioning software to behave correctly after an intrusion. We cannot make this assumption, as our VS is untrusted. Consequently, recovery must remain correct even under adversarial VS behavior,

including spoofing attacks carried out after the intrusion.

We assume that the intrusion time can be identified through forensic analysis, which is outside the scope of this work. Prior research provides techniques for reconstructing and analyzing intrusions [10, 11, 53]. Alternatively, one may attempt recovery under several candidate intrusion times and validate each resulting system state.

The only input our trusted recovery software depends on is the content of the timelock drive. The first step in recovery is to identify data written after the intrusion, so that it can be discarded. This immediately defeats spoofing attempts in which an intruder fabricates a new state after compromise.

We considered adding per-write "time-of-write" metadata to the TD. However, this would force a metadata update on every write, which is unnecessary because only one version per checkpoint interval is protected by a timelock.

Instead, we use time-of-lock as a proxy for determining whether a version predates the intrusion. Time-of-lock is recorded when the timelock command is issued. Any version protected by that timelock must have been written before its time-of-lock. Thus, if a block's time-of-lock precedes the intrusion time, its stored version must also precede the intrusion and is therefore valid for recovery.

As discussed in Section 4.2, versioning integrity is guaranteed for the most recent version of every logical address. During recovery, we scan the version metadata linked list (Section 4.4) to identify the most recent version for each logical address. Each log entry stores a logical address and a pointer to a physical block containing one of its versions. We determine the latest valid log entry by reading its time-of-lock. This entry provides the physical address of the most recent pre-intrusion version of that logical address.

After fully resetting all untrusted components, the VS is reinitialized using the logical-to-TD mapping constructed during recovery. The system can then resume operation safely.

4.8 Versioning System with Multiple Drives

Our work constructs a versioning system (VS) using a single Timelock Drive (TD). However, deploying this system across multiple drives managed by a single host requires additional design considerations. Because each TD maintains its own unique, unsynchronized clock, there is no universal timestamp for system-wide recovery.

To overcome this, the host VS must periodically log "recovery barriers" across all drives. To safely log a barrier, the host first issues a synchronization barrier (write barrier) to each drive. Next, it writes a special sentinel block containing a unique identifier to each drive and timelocks it. Finally, the host issues a second synchronization barrier to all drives before resuming normal execution.

With recovery barriers in place, multi-drive recovery proceeds securely. First, the intrusion time is determined with respect to a specific drive, say D. The recovery process then subtracts a small guard window from this timestamp to account for the maximum possible clock drift across the system. It then locates the last recovery barrier, B, written to drive D prior to this adjusted intrusion time. Finally, the process retrieves B's time-of-lock on every other drive. Since B is a synchronized event, its time-of-lock on each drive provides a safe, localized pre-intrusion timestamp, allowing the host to accurately roll back the state for every drive in the system.

5 Implementation

5.1 System Organization

Users create and modify files through a standard Linux ext4 filesystem. To persist these updates, ext4 issues block I/O requests to a block device driver. We implement a new block device driver using the BDUS framework [13]. This driver implements our untrusted VS and manages all version metadata. It communicates with the TD controller over sockets using the interface shown in Table 1.

We implement the TD controller logic in Dafny and formally verify the properties described in Section 6. The verified Dafny specification is transpiled to Rust and compiled with `rustc` to produce the executable TD-checker.

Our target design is a hardware microcontroller that runs the TD-checker natively and intercepts storage-interface commands directly. Because we were unable to find a commercial device that exposes this programmable interception layer, we prototype the TD controller on a Raspberry Pi.

5.2 TD Interface

TD exposes a modified ATA command interface, TD-ATA, derived from the ATA specification. TD-ATA (Table 1) extends ATA by adding fields for timelock durations to write commands—reusing reserved and obsolete bits—and by introducing two additional operations: `timelock-update` and `scan-td-log`. We include only the commands needed for our experiments, though additional ATA commands (e.g., power-management, SMART, and streaming-configuration operations) can be incorporated once proven safe.

The controller interprets each TD-ATA command, along with its supplied parameters and the current timelock state of the addressed block, to determine whether to accept or reject the command. Accepted commands are forwarded unchanged to the underlying storage device.

5.3 Metadata Layout

Our VS issues sequential writes to storage, preserving spatial locality: each new write is placed in the next free disk block, typically adjacent to the previous one. VS and TD metadata are persisted at the end of each epoch. To maintain spatial write locality, we interleave VS and TD metadata blocks as shown in Figure 4.

TD metadata log: Each metadata entry is 8 bytes. Four bytes are dual-purpose, storing either the timelock (for frozen

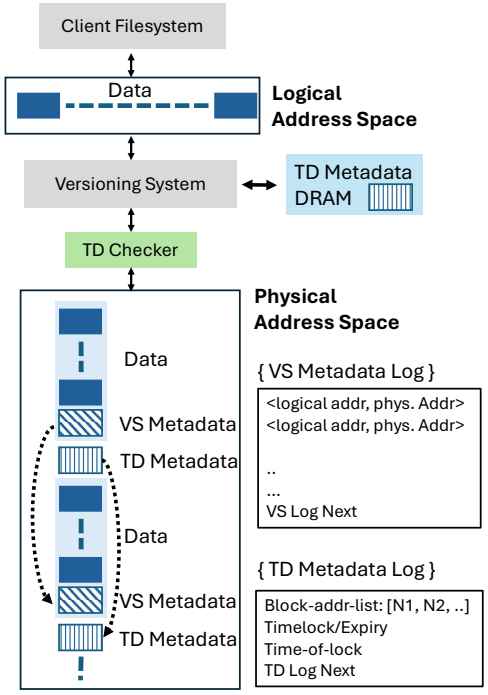


Figure 4: Address space layout.

state) or the expiry time (for countdown state). The remaining four bytes encode the time-of-lock in the upper 31 bits, with the lowest bit indicating whether frozen or unfrozen. We store metadata for adjacent addresses together, allowing $\frac{\text{BlockSize}}{8}$ entries per metadata block. With a 4KB block size, exactly 512 TD metadata entries fit in one block. Each block contains a forward pointer to the next TD metadata block.

VS metadata log: Each log entry is also 8 bytes. Thus, a 4KB metadata block stores $\frac{\text{BlockSize}-8}{8}$ entries (511 entries), reserving 8 bytes for a forward pointer to the next VS metadata block.

5.4 VS Append-Only Metadata Log

VS metadata maps each logical address to a set of TD addresses that contain its versions. We maintain this mapping using an append-only log (Section 4).

VS reserves a physical block (the *home node*) that stores the head pointer of a linked list of VS metadata blocks. Within each metadata block, we reserve 8 bytes for a pointer to the next block in the list. Before writing a metadata block, the VS preallocates a TD address for the next metadata block and records this pointer in the block currently being written.

This design introduces a subtle challenge: the current metadata block points to a future block that has not yet been written, and thus contains no valid metadata. If a crash or power failure occurs at this point, naïvely following the pointer chain would encounter an invalid block. We address this issue by relying on the time-of-lock. Metadata blocks in the linked list are written in chronological order, except for the final preallocated (but unwritten) block. During recovery, if a metadata block's time-of-lock indicates it was written *before* its parent, we conclude that we have reached the true end of the list.

5.5 Garbage Collecting VS Metadata Logs

We must periodically garbage-collect the VS metadata log. The VS performs this operation infrequently (user-defined, e.g., once per month). Before unfreezing the old list, the VS constructs a condensed log that contains only the most recent version entry for each logical address. It computes this condensed log by scanning its in-memory copy of the metadata. The VS then stores the new log in TD and timelocks it with the same duration as its underlying data. The head of the new list is written to a second predefined head-pointer location in the home node. The garbage collector alternates between these two locations.

Note that TD is agnostic to whether a block contains data or version metadata; it is isolated from versioning logic and protects both using the same mechanisms. The garbage collector responsible for freeing the old VS metadata list is itself part of the untrusted VS. To free the old list, it issues `timelock-update` commands to the TD addresses that store the old metadata, but only after writing the new log to a separate location. The old list remains immutable during its countdown timelock state. Consequently, even a fully compromised host cannot corrupt VS metadata stored on TD until its timelock expires.

5.6 Garbage Collecting TD Metadata Logs

TD metadata logs are maintained using a solution similar to the VS metadata log. The mechanism for garbage-collecting TD metadata logs is also similar to that for VS logs, with one key difference. Unlike VS metadata, a TD metadata log entry cannot be unfrozen until its corresponding data block is also unfrozen. The VS unfreezes a data block only after the application creates a new version for its corresponding logical address.

To address this constraint during garbage collection, the VS issues a write that copies the current version of a logical address to a new physical block. This action enables the VS to treat the original version as the "old version" and unfreeze its TD block. Once the data block is unfrozen, the corresponding TD metadata entry can also be unfrozen, allowing the TD metadata log to be reclaimed safely.

5.7 TD-aware Disk Block Management in VS

The VS maintains a list of physical blocks that are in the free state. Initially, this list contains all blocks. When the filesystem writes to a logical address, the VS allocates a free TD block to store the new version. The TD block holding the previous version for that logical address is then unfrozen and moved to a countdown list. Periodically, the VS scans this countdown list and returns any TD blocks whose timelocks have expired back to the free list.

Defragmentation: The VS preserves high write performance by issuing sequential writes; however, over time, the available sequential region becomes fragmented, reducing opportunities for long, efficient writes—a well-known limitation of log-structured filesystems [41]. Periodic cleaning restores sequential space by relocating live data and reclaiming fragmented segments. Unlike LFS, our VS must wait for each segment's timelock to expire before reclaiming it. A timelock-aware defragmentation daemon performs this cleaning while preserving future sequential write performance. Because all reclamations in the VS must respect timelocks, and the TDs clock remains idle while not powered on (Section 2), we note that defragmentation and garbage collection cannot make progress unless the TD is powered on.

6 Formally Verified Security Guarantees

We formally verified the following two strong guarantees for the metadata state of TD: (1) There is no way to bypass the immutability guarantee of a timelock. (2) In the event of a power outage, replaying the TD log of transactions correctly reconstructs the metadata state. To formally prove properties about TD we create a model of it in Dafny. Dafny is a programming language that supports proofs about functional correctness of code [24].

6.1 Timelock Guarantees

To guarantee timelock protections cannot be bypassed, we prove that our code exactly implements the state machine in Figure 2b. We first create an abstract model of the state machine, then prove our executable code (which operates on metadata encoded as bytes) correctly follows all states and transitions. This process, known as refinement, guarantees our code is free of exploits that allow an attacker to bypass states (e.g. skipping from Frozen Timelock directly to Free).

Proof sketch: (1) We model abstract states A and transitions from Figure 2b with the function $\texttt{transition}(A)$. (2) We create a function $\texttt{abstract}(B)$ that converts a sequence of bytes B to an abstract state. (3) We implement $\texttt{transition}(B)$ which performs byte-level operations to encode the next state. Our proof shows that transitioning at the byte level and then abstracting is equivalent to transitions at the abstract level. Formally, for any byte sequence B: $\texttt{abstract}(\texttt{transition}(B)) = \texttt{transition}(\texttt{abstract}(B))$.

6.2 Recovery Guarantees

If the system loses power, we must correctly reconstruct the timelock metadata state from the log stored on the drive. We achieve this by persisting each metadata transaction to form a transaction log. Our proof demonstrates that scanning and replaying this log produces a metadata state equivalent to the original execution.

Proof sketch: assume that replaying the log up to transaction n produces an identical metadata state to performing those n transactions. When we perform transaction n+1 during normal execution, we update the metadata state and append the transaction to the end of the log. Since (1) the replay is correct through transaction n, and (2) the replay and real-time execution of a transaction apply the same updates, appending transaction n+1 to the end of the log preserves correctness. Thus, given the same initial state, both execution paths will always produce the same final state.

Table 2: Workloads Used for Evaluating TD.

Workload	Description
VM-Enterprise [23]	Traces collected from VMs running enterprise software.
Exchange [43]	Traces collected from production mail server at Microsoft.
TPCC [20]	Traces of TPCC benchmark.
TPCE [20]	Traces of TPCE benchmark.
Finance [49]	Traces from OLTP applications at large financial institutions.
Web-Research [22]	Traces from Apache web-server managing FIU research projects.
filebench [46]	Filesystem benchmarking tool. Benchmarks used simulate a file server, a mail server, and database transactions.

6.3 Proof Assumptions

Since Dafny does not natively support raw ATA commands, we use a trusted but unverified Rust function to pass commands through to the drive. We model the transaction log on disk as a data structure containing a sequence of 4KB blocks. Each time the Rust function is invoked, Dafny updates the transaction log. As such, we assume this model accurately reflects blocks as persisted on disk. As with all computer-aided proofs, we also rely on the correctness of the theorem provers verifying our Dafny code.

7 Evaluation

Our evaluation demonstrates the efficiency of TD across a variety of workloads. We show that, for latency, bandwidth, and I/O, TD incurs negligible overhead compared to a baseline solution. Additionally, we demonstrate the value of our host-side cache by evaluating an alternative design that scans the TD log to compute the current metadata state. Lastly, we compare our solution to the widely used backup/snapshotting solution, LVM [17], to explore the storage overhead and recovery-time implications of TD .

7.1 Experimental Setup

To measure TD performance, we use the workloads in Table 2. These include a mixture of traces and benchmarking workloads that capture the behavior of enterprise software [23],

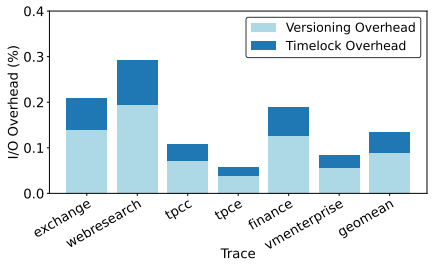


Figure 5a. Disk IO overhead.

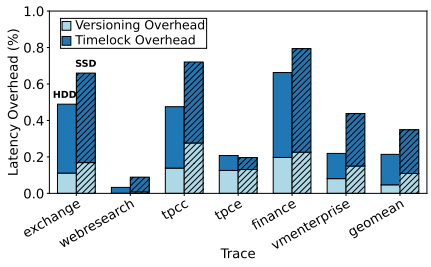


Figure 5b. Latency overhead.

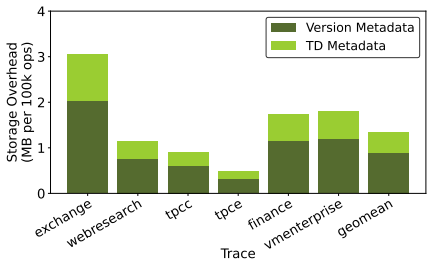


Figure 5c. Space overhead.

mail servers [43,46], web servers [22], file servers [46], and database OLTP workloads [20,46,49]. We compare runtime performance against a baseline log-structured block device. This baseline has no security or versioning features and uses the log-structure to improve random-write performance.

In production, we envision the checker logic integrated directly into the drive's controller silicon, with a negligible footprint. For our experiments, we use two configurations: one for security testing and one for performance evaluation.

The first configuration involves a prototype implementation of the TD checker on a low-end controller (a Raspberry Pi 5). The host driver communicates with the controller via Ethernet. This emulates the physical isolation between the host and checker and is used for the experiment in Section 7.2.

The second configuration collates the controller logic with the host machine. The host driver communicates with the controller logic via shared memory IPC. This is done to eliminate any potential bandwidth or latency limitations of the Ethernet connection, thus more closely measuring the overhead of TD. We use this configuration for all other experiments.

The primary storage device used in our evaluation is a 4TB Seagate ST4000DMZ04 hard disk. In experiments that explicitly mention SSD performance, we use a 1 TB Samsung MZ-77E1T0B drive.

7.2 Security Analysis

Our formal verification of the TD controller in Section 6 proves our transient immutability guarantee. To validate our secure backup system, we deploy 18 ransomware samples from well-known "families" of ransomware attacks [21]. TD successfully recovered the filesystem state for every sample.

7.3 Performance Analysis

7.3.1 Trace Evaluation:

We evaluate the latency and I/O overhead of TD via I/O traces collected from a variety of workloads found in Table 2. Before each workload, we write 1GB of data to the driver to warm up the system. We run the first 100,000 commands from each trace as quickly as possible to stress-test the I/O latency of TD. We also measure the performance of just versioning, with no timelock protections, to learn the performance breakdown.

I/O Overhead: I/O overhead is due to VS and TD metadata reads and writes. As noted earlier, these metadata are also delegated to the host, and therefore, they are only read in uncommon scenarios during recovery.

In our experiments, we conservatively version and timelock all writes. Even so, the I/O overhead stays low as seen in Figure 5a. Write-heavy traces, such as Web-Research and Exchange, incur the highest I/O overhead due to additional version and timelock metadata that must be persisted to protect the data.

Because the metadata cache is managed by the host, we do not incur additional metadata reads at runtime. Without this optimization, the I/O overhead of TD grows considerably, which we explore in Section 7.3.4.

Latency Overhead: Compared to the baseline system, the latency overhead of TD is composed of the I/O overhead, the ordering of new I/O, the cost of managing the host metadata cache (computing hashes and transferring cache blocks), and the overhead of checking the state of each block address before it is written.

As seen in Figure 5a, the I/O overhead is negligible, which is due to host-side caching. The order of I/O (contiguous versus random) can have significant performance implications. As explained in Section 5.3, VS writes version and timelock metadata contiguously with data during write-heavy periods. As a result, the additional I/O incurred by TD has minimal performance impact. Looking at the breakdown in Figure 5b, we see that the bulk of overhead in TD is due to timelock checks (ensuring blocks are free) and managing the host metadata cache. Even with the cost of timelocks/metadata management, TD achieves negligible overhead over the baseline.

We found that TD performs significantly better than LVM in terms of latency across all benchmarks, except Web-Research. LVM does not structure its data writes sequentially in a log [41], and thus suffers from random-write latency. The Web-Research benchmark is write-heavy; however, most of its writes are naturally sequential, which neutralizes the performance gap between TD and LVM.

7.3.2 Filesystem Benchmarks

To evaluate the performance of real-time workloads, we mount an ext4 filesystem onto the VS driver and run several write-intensive filebench [46] workloads. The filesystem

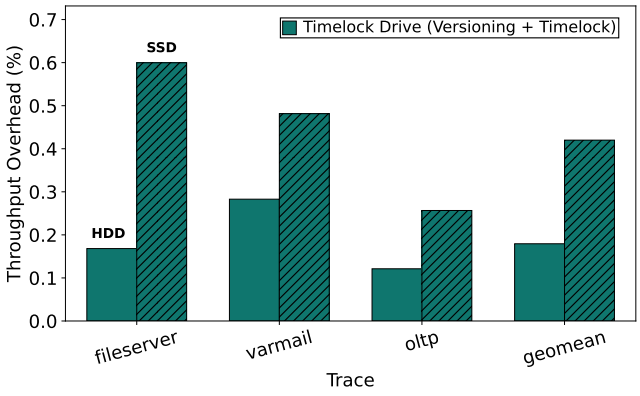


Figure 6: Throughput overhead for filebench workloads.

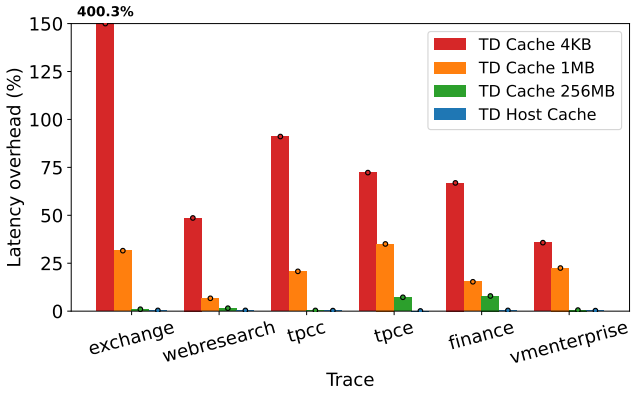


Figure 7: Cache sensitivity analysis for traces in Table 2.

used is the same for both TD and the baseline, so it does not add new overhead compared to the trace evaluation. We find, as shown in Figure 6, that, much like latency, TD incurs negligible throughput overhead.

7.3.3 Performance on HDD vs SSD

We perform all trace and filesystem benchmark experiments twice, once on an HDD and once on an SSD. Read and write operations are considerably faster on an SSD. While the absolute cost of managing versions and doing timelock related computations stays the same, they become a larger proportion of the overall runtime. As a result we see increased overhead of TD, especially in workloads with a large volume of writes. While TD is less performant on SSDs, it still maintains a modest overhead (<1%) in all workloads.

7.3.4 Delegating Caching Metadata to Host

To motivate our host-managed metadata cache design from Section 3.4, we explore an alternative design for TD that manages its own cache on the gatekeeper. In a gatekeeper-managed cache design, every cache miss requires scanning the entire timelock metadata log to compute the most up-to-date metadata state. To evaluate the performance of this design, we perform a sensitivity study with three different cache sizes (small/4K, medium/1M, and large/256M).

Due to the timelock constraint, there is no temporal locality

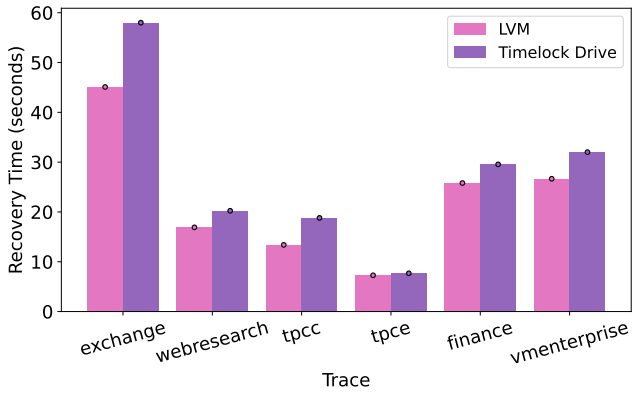


Figure 8: Recovery time for traces in Table 2 compared to LVM.

for TD metadata. As a result, for write-intensive traces, we observe frequent cache misses, each of which requires an expensive log scan. This is true even for a medium-sized cache (1 MB), resulting in over 30% overhead. Even for non-write-intensive workloads, triggering a scan moves the seek head away from the most recently read/written data. As a result, cache misses increase the amount of random disk I/O. To avoid missing the cache entirely, we need 2GB of cache for every 1 TB of storage. This is prohibitive for a microcontroller design, which is why we opt for the host-managed cache.

7.4 Efficiency of Data Recovery

We run each trace to completion, take a snapshot, overwrite every address in the trace to simulate an integrity attack, and measure recovery time for both TD and LVM. We compared our performance against LVM's snapshot feature.

Recovery in TD is proportional to the length of the version log, which is proportional to the number of writes. This scanning method is modestly slower than LVM, as shown in Figure 8. This is one of the two costs of TD's append-only log design compared to a conventional backup system.

7.5 Storage Space Overhead

TD introduces two sources of storage space overhead. The first is from TD metadata. The TD log grows dynamically as more data is timelocked. One TD metadata log block can track updates to 1020 data blocks. A second form of space overhead is version metadata. Like TD metadata, the version log grows over time as more versions are created. One version log block can track 511 logical-to-physical mappings. Both logs grow proportionally to the number of writes.

As explained in Sections 5.5 and 5.6, we perform delayed defragmentation of the log and reclaim space once the timelock protecting it expires. Looking at Figure 5c, we can see that per 100k operations, the space overhead incurred by TD is modest. The highest space overhead comes to just over 3MB for both version and TD metadata combined. We observe that for most traces, one-quarter to one-third of the log can be reclaimed, freeing up more space in the future.

LVM, on the other hand, tracks a constant amount of metadata per snapshot, which we observed to be about 2MB. Most of our traces stay under this size for the 100k operations; however, for long-running workloads with frequent writes, LVM's constant overhead is lower than ours. This is a cost we pay for a non-overwriting pure append-only system.

8 Related Work

This paper is the first to propose a versioning system while using a storage device that forbids overwrites for a fixed period. TD constructs a secure backup solution on top of an isolated timelock primitive, which was not previously known to be possible. This is challenging as the timelock drive forbids all overwrites, including metadata, for a fixed period. We also formally verify the security guarantees of TD.

Backups and Versioning. The traditional ransomware defense is data backups. Common policies include versioning [9, 14, 33], snapshotting [7], and delta backups [6, 8, 54]. These solutions are implemented in software and are thus vulnerable to common malware attacks that compromise the OS [18, 31], and are thus inadequate for ransomware defense. However, the versioning policies and optimizations they use can be implemented in TD's untrusted VS.

Several prior works use the intrinsic out-of-place write structure of an SSD to implement data versioning that defends against ransomware without trusting the OS [18, 19, 50]. These works modify garbage collection in the flash translation layer (FTL) to retain old versions of data and enable rollbacks. BVSSD [19] retains all versions indefinitely, which implements a continuous data protection policy [52, 55] inside an SSD. Storage on an SSD is finite, which is why RSSD [36] modifies FTL garbage collection to back up flash blocks to the cloud via NVMe over Ethernet before erasing them, providing the illusion of infinite capacity. To truly address capacity limitations, we must eventually reclaim space. FlashGuard [18] and Project Almanac [50] allow for garbage collection and modify the FTL to add time-delayed versioning.

The use of time as a defense in FlashGuard and Project Almanac is similar to our work, however, we differ in several key ways. Both works use time retention as a security policy in their versioning logic. Notably, these prior works include a versioning system in their TCB and overwrite metadata, whereas we do not. We are the first to show that time retention can be decoupled from versioning, and that untrusted versioning software can use the TD interface to build a secure backup system. Additionally, because both prior works implement versioning in the FTL, the entire modified firmware is included in the TCB. Verifying that changing the FTL does not introduce vulnerabilities is a significant challenge. Tripathy et al. formally verify several, *but not all*, properties of a ransomware-resistant FTL [48], improving confidence in the correctness of FTL-based solutions.

RSSD shows that prior S4 works, including FlashGuard and Project Almanac, are vulnerable to a "trimming attack" that bypasses security checks in the FTL to erase data [36]. In contrast, our work decouples versioning from the TCB, which is a non-trivial feat as explained in Section 4.4. Our isolated TD is only responsible for enforcing timelock guarantees and is separated from the FTL, making it easier to prove strong security guarantees. We prove that TD has no equivalent to the "trimming attack" [36].

Integrating versioning and timelock checks into SSD firmware makes versioning and timelock policies less flexible, thereby restricting their use cases. By excluding versioning from the TCB, our approach enables easy updates to versioning policies and optimizations in untrusted software, without altering TCB code. As a result, TD enables several uses beyond encryption ransomware, as detailed in Section 3.5. Commercial products, such as ExaGrid and Oracle, implement "retention timelocks" by enforcing retention periods in application- or service-level backup logic [12, 44]. In contrast, our solution timelocks physical blocks on disk. It isolates timelocks from versioning logic, resulting in a minimal TCB that improves security and flexibility.

Ransomware Detection and Prevention. Another line of defense is ransomware detection, which monitors I/O and data patterns to flag malicious behavior [3, 28, 32]. But even when attacks are caught early, some data is already damaged [36], and attackers can adapt to evade known heuristics. Detection accelerates recovery, but it cannot prevent data loss; it must be paired with secure backups.

9 Conclusion

While significant advancements have been made to address hardware and software security vulnerabilities, solutions targeting human errors remain limited. Given that nearly two-thirds of ransomware attacks exploit human weaknesses, there is a pressing need for defenses that do not rely solely on managing data access through user credentials.

Timelock Drive (TD) is a significant step toward achieving this goal. It is the first work to completely isolate the timelock defense, enabling the construction of a secure backup system with the smallest known TCB — one that doesn't trust the user or the VS. The TD interface is simple enough to allow formal verification, yet flexible enough to support a wide range of versioning optimizations and data retention policies. These benefits come with negligible performance and storage overhead, making TD an efficient and robust solution.

Acknowledgments

We thank our shepherd and the anonymous OSDI reviewers for their constructive feedback. This work was supported by the National Science Foundation grant 2403119.

Availability

Our artifact is publicly available on GitHub under the GNU Affero General Public License v3.0. We have also registered our artifact on Zenodo with the DOI 10.5281/zenodo.20647469.

References

[1] M-trends 2024 special report. Technical report, Mandiant, 2024.

[2] Steve Alder. Ascension ransomware attack hurts financial recovery. 2024. https://www.hipaajournal.com/ascension-cyberattack-2024/.

[3] SungHa Baek, Youngdon Jung, Aziz Mohaisen, Sungjin Lee, and DaeHun Nyang. Ssd-insider: Internal defense of solid-state drive against ransomware with perfect data recovery. In *2018 IEEE 38th International Conference on Distributed Computing Systems (ICDCS)*, pages 875–884. IEEE, 2018.

[4] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. *ACM SIGOPS operating systems review*, 37(5):164–177, 2003.

[5] Simon Biggs, Damon Lee, and Gernot Heiser. The jury is in: Monolithic os design is flawed: Microkernel-based designs improve security. In *Proceedings of the 9th Asia-Pacific Workshop on Systems*, pages 1–7, 2018.

[6] Randal C Burns and Darrell DE Long. Efficient distributed backup with delta compression. In *Proceedings of the fifth workshop on I/O in parallel and distributed systems*, pages 27–36, 1997.

[7] Hoi Chan and Trieu Chieu. An approach to high availability for cloud servers with snapshot mechanism. In *Proceedings of the industrial track of the 13th ACM/IFIP/USENIX international middleware conference*, pages 1–6, 2012.

[8] Ann Chervenak, Vivekenand Vellanki, and Zachary Kurmas. Protecting file systems: A survey of backup techniques. In *Joint NASA and IEEE Mass Storage Conference*, volume 99. Citeseer, 1998.

[9] Brian Cornell, Peter A Dinda, and Fabián E Bustamante. Wayback: A user-level versioning file system for linux. In *Proceedings of Usenix Annual Technical Conference, FREENIX Track*, pages 19–28, 2004.

[10] David Devecsery, Michael Chow, Xianzheng Dou, Jason Flinn, and Peter M Chen. Eidetic systems. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 525–540, 2014.

[11] George W Dunlap, Samuel T King, Sukru Cinar, Murtaza A Basrai, and Peter M Chen. Revirt: Enabling intrusion analysis through virtual-machine logging and replay. *ACM SIGOPS Operating Systems Review*, 36(SI):211–224, 2002.

[12] ExaGrid. Retention time-lock for ransomware recovery. https://www.exagrid.com/wp-content/uploads/ExaGrid-Retention_Time-Lock_for_Ransomware_Recovery_DS.pdf, 2023.

[13] Alberto Faria, Ricardo Macedo, José Pereira, and João Paulo. Bdus: implementing block devices in user space. In *Proceedings of the 14th ACM International Conference on Systems and Storage*, pages 1–11, 2021.

[14] Michail Flouris and Angelos Bilas. Clotho: Transparent data versioning at the block i/o level. In *MSST*, pages 315–328. Citeseer, 2004.

[15] Roxana Geambasu, Tadayoshi Kohno, Amit A Levy, and Henry M Levy. Vanish: Increasing data privacy with self-destructing data. In *USENIX security symposium*, volume 316, pages 10–5555, 2009.

[16] Roberto Gioiosa, Jose Carlos Sancho, Song Jiang, and Fabrizio Petrini. Transparent, incremental checkpointing at kernel level: a foundation for fault tolerance for parallel computers. In *SC'05: Proceedings of the 2005 ACM/IEEE conference on Supercomputing*, pages 9–9. IEEE, 2005.

[17] Michael Hasenstein. The logical volume manager (lvm). *White paper*, 2001.

[18] Jian Huang, Jun Xu, Xinyu Xing, Peng Liu, and Moinuddin K Qureshi. Flashguard: Leveraging intrinsic flash properties to defend against encryption ransomware. In *Proceedings of the 2017 ACM SIGSAC conference on computer and communications security*, pages 2231–2244, 2017.

[19] Ping Huang, Ke Zhou, Hua Wang, and Chun Hua Li. Bvssd: Build built-in versioning flash-based solid state drives. In *Proceedings of the 5th Annual International Systems and Storage Conference*, pages 1–12, 2012.

[20] Swaroop Kavalanekar and Bruce Worthington. Microsoft enterprise traces (SNIA IOTTA trace set 131). In Geoff Kuenning, editor, *SNIA IOTTA Trace Repository*. Storage Networking Industry Association, October 2007.

[21] Mohsen Khashei. Ransomware-samples. https://github.com/kh4sh3i/Ransomware-Samples, 2022.

[22] Ricardo Koller and Raju Rangaswami. I/o deduplication: Utilizing content similarity to improve i/o performance. *ACM Trans. Storage*, 6(3), September 2010.

[23] Chunghan Lee, Tatsuo Kumano, Tatsuma Matsuki, Hiroshi Endo, Naoto Fukumoto, and Mariko Sugawara. Understanding storage traffic characteristics on enterprise virtual desktop infrastructure. In *Proceedings of the 10th ACM International Systems and Storage Conference*, pages 1–11, 2017.

[24] K Rustan M Leino. Dafny: An automatic program verifier for functional correctness. In *International conference on logic for programming artificial intelligence and reasoning*, pages 348–370. Springer, 2010.

[25] Jochen Liedtke. On micro-kernel construction. *ACM SIGOPS Operating Systems Review*, 29(5):237–250, 1995.

[26] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. Meltdown. *arXiv preprint arXiv:1801.01207*, 2018.

[27] Jia Liu, Tibor Jager, Saqib A Kakvi, and Bogdan Warinschi. How to build time-lock encryption. *Designs, Codes and Cryptography*, 86:2549–2586, 2018.

[28] Donghyun Min, Donggyu Park, Jinwoo Ahn, Ryan Walker, Junghee Lee, Sungyong Park, and Youngjae Kim. Amoeba: An autonomous backup and recovery ssd for ransomware attack defense. *IEEE Computer Architecture Letters*, 17(2):245–248, 2018.

[29] Edmund B Nightingale, Kaushik Veeraraghavan, Peter M Chen, and Jason Flinn. Rethink the sync. *ACM Transactions on Computer Systems (TOCS)*, 26(3):1–26, 2008.

[30] Justin D Osborn and David C Challener. Trusted platform module evolution. *Johns Hopkins APL Technical Digest (Applied Physics Laboratory)*, 32(2):536–543, 2013.

[31] Harun Oz, Ahmet Aris, Albert Levi, and A Selcuk Uluagac. A survey on ransomware: Evolution, taxonomy, and defense solutions. *ACM Computing Surveys (CSUR)*, 54(11s):1–37, 2022.

[32] Jisung Park, Youngdon Jung, Jonghoon Won, Minji Kang, Sungjin Lee, and Jihong Kim. Ransomblocker: A low-overhead ransomware-proof ssd. In *Proceedings of the 56th Annual Design Automation Conference 2019*, pages 1–6, 2019.

[33] Zachary Peterson and Randal Burns. Ext3cow: A time-shifting file system for regulatory compliance. *ACM Transactions on Storage (TOS)*, 1(2):190–212, 2005.

[34] James S Plank, Jian Xu, and Robert HB Netzer. Compressed differences: An algorithm for fast incremental checkpointing. Technical report, Citeseer, 1995.

[35] Niels Provos, Markus Friedl, and Peter Honeyman. Preventing privilege escalation. In *12th USENIX Security Symposium (USENIX Security 03)*, 2003.

[36] Benjamin Reidys, Peng Liu, and Jian Huang. Rssd: Defend against ransomware with hardware-isolated network-storage codesign and post-attack analysis. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 726–739, 2022.

[37] Charles Reis and Steven D Gribble. Isolating web programs in modern browser architectures. In *Proceedings of the 4th ACM European conference on Computer systems*, pages 219–232, 2009.

[38] Charles Reis, Alexander Moshchuk, and Nasko Oskov. Site isolation: Process separation for web sites within the browser. In *28th USENIX Security Symposium (USENIX Security 19)*, pages 1661–1678, 2019.

[39] John Riggi. A look at 2024's health care cybersecurity challenges. 2024. https://www.aha.org/news/aha-cyber-intel/2024-10-07-look-2024s-health-care-cybersecurity-challenges.

[40] Ronald L Rivest, Adi Shamir, and David A Wagner. Time-lock puzzles and timed-release crypto. 1996.

[41] Mendel Rosenblum and John K Ousterhout. The design and implementation of a log-structured file system. *ACM Transactions on Computer Systems (TOCS)*, 10(1):26–52, 1992.

[42] Reiner Sailer, Enriquillo Valdez, Trent Jaeger, Ronald Perez, Leendert Van Doorn, John Linwood Griffin, Stefan Berger, Reiner Sailer, Enriquillo Valdez, Trent Jaeger, et al. shype: Secure hypervisor approach to trusted virtualized systems. *Techn. Rep. RC23511*, 5, 2005.

[43] Vishal Sharda, Swaroop Kavalanekar, and Bruce Worthington. Exchange server traces (SNIA IOTTA trace 134). In Geoff Kuenning, editor, *SNIA IOTTA Trace Repository*. Storage Networking Industry Association, December 2007.

[44] Kelly Smith. Retention lock your cloud database backups for increased ransomware protection. https://blogs.oracle.com/maa/post/retention-lock-for-increased-ransomware-protection, 2023.

[45] John D Strunk, Garth R Goodson, Michael L Scheinholtz, Craig AN Soules, and Gregory R Ganger. Self-securing storage: Protecting data in compromised systems. In *OSDI*, pages 165–180, 2000.

[46] Vasily Tarasov, Erez Zadok, and Spencer Shepler. Filebench: A flexible framework for file system benchmarking. *login Usenix Mag.*, 41, 2016.

[47] The Linux Kernel Documentation. *Page Table Isolation (PTI)*. Documentation for PTI (Page Table Isolation) on x86.

[48] Shivani Tripathy, Debiprasanna Sahoo, Manoranjan Satpathy, and Madhu Mutyam. Formal modeling and verification of security properties of a ransomware-resistant ssd. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 42(8):2766–2770, 2022.

[49] University of Massachusetts. OLTP trace from UMass trace repository. https://traces.cs.umass.edu/docs/traces/storage/, 2002.

[50] Xiaohao Wang, Yifan Yuan, You Zhou, Chance C Coats, and Jian Huang. Project almanac: A time-traveling solid-state drive. In *Proceedings of the Fourteenth EuroSys Conference 2019*, pages 1–16, 2019.

[51] Sophos Whitepaper. The state of ransomware in healthcare 2024. Technical report, 2024. https://assets.sophos.com/X24WTUEQ/at/4bk9xt4h7gsm4xs6mfzh3k/sophos-state-of-ransomware-healthcare-2024.pdf.

[52] Weijun Xiao, Jin Ren, and Qing Yang. A case for continuous data protection at block level in disk array storages. *IEEE Transactions on Parallel and Distributed Systems*, 20(6):898–911, 2008.

[53] Yulai Xie, Dan Feng, Zhipeng Tan, and Junzhe Zhou. Unifying intrusion detection and forensic analysis via provenance awareness. *Future Generation Computer Systems*, 61:26–36, 2016.

[54] Yucheng Zhang, Ye Yuan, Dan Feng, Chunzhi Wang, Xinyun Wu, Lingyu Yan, Deng Pan, and Shuanghong Wang. Improving restore performance for in-line backup system combining deduplication and delta compression. *IEEE Transactions on Parallel and Distributed Systems*, 31(10):2302–2314, 2020.

[55] Ningning Zhu and Tzi-cker Chiueh. Portable and efficient continuous data protection for network file servers. In *37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN'07)*, pages 687–697. IEEE, 2007.

High Fidelity Models for Large Scale Stateful Services (Operational Systems)

Nouraldin Jaber Dongyun Jin Bernhard Kragl Enrico Magnago
Gustavo Petri Thorsten Tarrach Serdar Tasiran
{noujab, dongyuj, kraglb, emagnago, gfpetri, tarracht, tasirans}@amazon.com
Amazon Web Services

Abstract

The Amazon Simple Storage Service (S3) is 20 years old, hosts more than 500 trillion objects, and averages over 200 million requests per second. Its API must continuously evolve and adapt to new workloads and use cases, sometimes resulting in full re-implementations like the S3 Express One Zone storage class. This highlights two conflicting requirements: ensuring that every change preserves backward compatibility and maintaining a high-paced development environment. We report on our experience combining the use of *reference models*, in a model-based testing (MBT) style, with *abstraction* to systematically explore all behaviors of the S3 API. Our approach is grounded in the abstraction of API requests and state elements, enabling systematic validation of all abstract inputs and system states. Through rigorous validation, we produce models that are sufficiently accurate to be used as de facto specifications of S3. Our tool continuously runs in the S3 development and deployment (CI/CD) cycle, validating changes, preventing regressions, and supporting the development of new features.

1 Introduction

Over the past couple of decades, cloud services have evolved from simple web-based APIs to elaborate services with dozens of features, configurations, and input and output parameters. In this work we focus on the Amazon Simple Storage Service (S3), which supports a wide variety of customer use cases and workloads that have co-evolved with S3 over its 20 years of existence. This evolution requires new code to be added and existing code to be refactored by hundreds of developers continuously. To provide for different performance, price, and availability trade-offs, the S3 API has been re-implemented multiple times, sometimes using entirely different code bases, programming languages, and hardware. Yet, to the customers, the API behavior must remain the same. We refer to the challenge of preserving the API behavior under continuous change as the *API sameness* or conformance checking problem. This challenge is the main focus of our work. We aim at validating the functional input-output behavior of the S3 API under normal sequential behavior. We focus on the sequential behavior thanks to the consistency guarantees of S3[1], which are validated through dedicated mechanisms (e.g. [BJA+21]) and are beyond the scope of this paper. In this work we execute each operation "in isolation" and focus exclusively on validating the correctness of the sequential behavior of the API.

There are two key aspects to API sameness: (i) defining what constitutes correct API behavior, and (ii) ensuring that the service implements the correct behavior. We address the first challenge by building an executable reference model for each API. Then, we propose a systematic model-based testing (MBT) approach to address the second challenge. In order to get sufficient confidence in the correctness of the service, we require the tests to achieve sufficient and measurable coverage of the behaviors of the system. For this reason we design a systematic test generation strategy to provide the inputs on which we check conformance between the model and the API implementation. While the techniques we present are widely applicable to other stateful services, for the rest of the paper we will focus on S3. Our model and input generation techniques are now used in the continuous integration and deployment (CI/CD) pipelines of S3, preventing unexpected behaviors from reaching production. We run this validation after the execution of traditional unit and integration tests, and it has so far prevented more than 300 potential regressions from being deployed.

Amazon S3. Amazon S3 is a highly available and durable cloud storage service that provides a REST (Representational State Transfer) API. It supports 96 API operations [S3D24c]. Beyond the `PutObject` and `GetObject` operations, over its evolution S3 has added support for conditional read and write operations, multiple checksum algorithms, object versioning, tagging, replication configurations, several server-side encryp-

[1] https://aws.amazon.com/s3/consistency/

tion configurations, and many more features. To illustrate the complexity of S3 APIs and the types of edge-cases involved, consider `GetObject` [S3D24b]. `GetObject` appears to be simple: it takes a bucket name and a key name, and returns the object stored in the bucket for that key name. Consider the following `GetObject` request:

```
GetObject( bucket="bucket", key="abc",
           range="bytes=0-5242880",
           checksumMode="ENABLED")
```

`range` is a standard HTTP header described in RFC-9110 [RFC22], while `bucket`, `key` and `checksumMode` are specific to the S3 API. The RFC states that an HTTP server *may* support a sequence of ranges on `GET` requests. S3 supports a single byte range and implements the behavior described in the RFC. Depending on the state, S3 returns `206` for successful cases, `404` status code if the object does not exist, or `416` if the object is empty. Successful responses will contain the requested range of bytes of the object. In S3, the headers in the success response depend on the target object. In our example: a) if the object has 5242881 bytes or less, the response will contain the full object and its checksum, b) if the object has more than 5242881 bytes, only the first 5242881 bytes will be returned with no checksum, except, c) if the object was created with a multi-part upload API, and the first part is exactly 5242881 bytes, the response will include the checksum of the part.

We now consider a different `GetObject` request to highlight how client code often depends on specific S3 error responses, not just success responses.

```
GetObject( bucket="bucket", key="abc",
           If-Unmodified-Since="2026-02-02")
```

The `If-Unmodified-Since` header is another standard HTTP header. In `GetObject`, clients use this header to retrieve the object only if it has not been modified since the specified date. S3 will reply with `200 OK` and the object data if the content has not been modified since `2026-02-02`, and with `412 Precondition Failed` otherwise. A client will likely handle these two responses differently. Client code typically uses this header as a safety mechanism to detect changes in the object content. In this case, the error response clearly signals that the assumption has been violated. Preserving the exact behavior of both success and error responses is critical to the correctness of S3's client code. More generally, clients often rely on all observable behaviors of the API; from the service provider's viewpoint, this means *every* behavior must be preserved across changes to the service.

These two examples use only five distinct parameters and already highlight the many details that affect the full behavior of each API and their importance to customers. And yet, `GetObject` has 21 input parameters and `GetObject` responses have 36 output parameters plus the content payload. In addition, there are many configurations [S3D24a] of the underlying target bucket which affect the behavior of each request, such as bucket encryption, versioning, and access policies. As we shall see in Sec. 6, there are at least 10^{25} qualitatively distinct combinations of input parameters and states for `GetObject`. Covering all these combinations is not possible without novel reductions and automation.

Our approach to conformance. This paper addresses the conformance checking problem by combining *model-based testing* (MBT) [UPL12, BJA+21] with *predicate abstraction* [GS97]. We incrementally build an executable reference model to be the de-facto specification of the underlying API. This is motivated by the need to: a) make the model accessible to developers and tools with minimal effort, b) run conformance tests between the model and API implementation in CI/CD pipelines, and c) have reproducible diagnostics for issues identified in development.

Building a reference model takes effort and one might consider using an alternative/previous version of the service as the reference model. However, this is impractical and error prone. In response to a request, an API can allow multiple valid responses (e.g. different pagination of results, errors being reported in different order). An implementation may pick one of these possibilities, and another implementation might pick a different one. As a result, two versions of the service driven by the same test inputs can diverge while both being correct. A reference model that encapsulates all allowed behaviors of the service allows us to keep independence between the implementation under test and the reference model. In fact, for S3 Express One Zone, having access to the model was instrumental in the design and development of the implementation. The majority of S3 Express One Zone APIs match the behavior of regional S3, but getting a comprehensive specification of the APIs' behavior was not easy using only the existing regional S3 implementation. S3 Express One Zone developers used our model and test generation to validate conformance with regional S3. In specific cases where S3 Express One Zone intentionally deviates from regional S3 we extended our model with the new behavior.

Ensuring conformance is practically challenging due to the complexity of the S3 APIs with their large number of parameters and dependencies on the service state. In order to ensure API conformance for all behaviors, we need comprehensive coverage of these combinations of parameters and states. We use predicate abstraction to lift the universe of concrete requests and states to *abstract requests and states*, and to explore this much-reduced abstract space in a systematic way. Developers use this approach to comprehensively validate code changes over a well-defined (sub-)space of requests and states at development and pre-deployment time. The approach identifies regressions even in subtle edge cases, within a reasonable time budget. As a result, the approach improves development velocity and gives developers confidence that all behaviors will be tested before deployment.

To summarize, we use our validation approach for:

Preventing Regressions. A difficult challenge in maintaining a large decoupled code base is ensuring that changes to one component (e.g. a bug fix) do not introduce unintended changes to the behavior of the API. We use our executable specification as an oracle to *drive input generation strategies* aimed at maximally covering the set of behaviors that may be affected by changes. We then employ differential testing [McK98] between model and implementation to detect unintended changes.

Conformance Checking. A given API might have multiple implementations, such as the S3 Express One Zone storage class launched in 2023. With S3 Express One Zone, data is stored in a different bucket type called *directory buckets*. Directory buckets support the core APIs of regular S3 buckets, but with some intentional differences (e.g., entries in `ListObjectsV2` responses are not sorted). We used our model to guide and inform the development of S3 Express One Zone for the APIs that match regular S3. Similarly, we use it in other implementations and refactors, reducing the synchronization overhead across teams.

Model-based validation. Another challenging aspect in a large stateful application is generating tests that drive server-side behaviors to deep and non-obvious paths. We build a *custom input generation strategy*, which intentionally drives the model into the necessary pre-states (and since the model is accurate, also the system state). We use knowledge about the model state to generate inputs exercising a combination of service paths that target specific behaviors.

Validation Coverage. In order to achieve these goals, we need a cost-effective way of exploring the possible inputs of the system. Generating new inputs is only helpful if they drive the service to unexplored behaviors. In the `GetObject` example, the number of combinations of 21 input parameters and numerous state configurations makes exhaustive exploration intractable. Therefore, it is critical to carefully identify the key features and scenarios that need to be validated. We address this challenge using abstraction at the level of individual request and state features.[2] An important consequence of this approach is that we can quantify the portion of the request space that a particular input generation campaign covers.

2 Overview

Unit and integration testing are widespread mechanisms used to validate software [LW90, AO17]. Unfortunately, unit tests are only as good as the quality of the scenarios and assertions that developers consider [ZHM97]. In a highly decentralized development environment, it becomes hard to establish quality metrics from individual tests in each of the components. For web services, the overall end-to-end behavior of the service API represents yet another top-level testing target, commonly known as integration-testing [MLSRC21]. In general, assessing the quality of an integration test suite is hard, and it is seldom a subject of precise and actionable metrics.

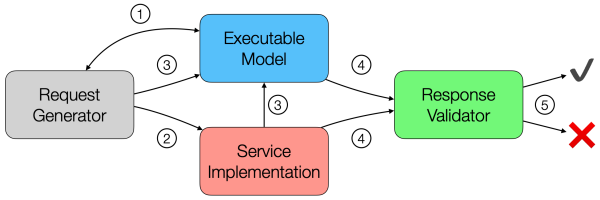


Figure 1: Model-based testing flow.

In our approach, we argue that the starting point of any effective validation technique (formal or semi-formal) is a specification of the system. Alas, when working with legacy systems, the correct level of abstraction or language for a specification is not always clear. Therefore, we make the pragmatic choice of defining a reference model as a specification, which we then use to validate the evolution or even different implementations of the system. Following the model-based testing (MBT) [UPL12] literature, we refer to the system implementation under consideration as the *System Under Test* (SUT), and use the model as an oracle in the SUT validation campaigns. Each validation campaign compares the SUT's behavior against the model on a set of inputs and highlights any discrepancies. To assess the quality of our model, and to later validate the SUT against it, we need automatic input generators that cover measurable portions of the API's input and state space. We develop a systematic input generation approach based on the model (Sec. 5). This helps us bootstrap the model when starting with a legacy system and, once the model is deemed accurate, validate evolutions of the system.

The systematic input generation is designed around three key properties:

No assertion left behind. Unlike unit testing, we do not need to write assertions. We compare every element of the responses generated by the SUT and the model. This is the strongest possible assertion on the input-output relation: equality between service and model responses.

No scenario left behind. We pair the model with automated test generation to solve the problem of test coverage and quality of integration testing. We employ input-output conformance [Tre08] validation to compare the SUT and the model (see Sec. 3).

No redundant scenario. Given the massive space of possible input requests to the API, we need to avoid redundancy. Since execution time is of the essence, our methodology avoids redundant inputs.

[2] Roughly, we consider a feature to be any of the request arguments, configurations of the bucket, or state pre-conditions that can influence the behavior of the operation.

```
while not metric.is_covered():
    test = metric.next_test(model) ①
    s_resp = service.run(test) ②
    m_resp = model.run(test, s_resp) ③
    valid = validator(test,s_resp,m_resp) ④
    report(valid) ⑤
    metric.update(model, test, m_resp)
```

Listing 1: Coverage-driven testing (pseudo).

Fig. 1 depicts the MBT strategy. There are three main components to our validation besides the implementation of the service itself. Firstly, the executable model (Sec. 3) is a simple implementation of the API. It preserves all the functional aspects of the service and ignores all the non-functional ones. This is shown in the blue box in Fig. 1. Secondly, the green box represents the response validator. This component implements the logic to *assert* whether the response of the SUT is consistent with the response generated by the model. It also allows us to select between the multiple behaviors allowed by the reference model and keep its state in sync with the SUT. Finally, the gray box corresponds to the request generator. The generator implements the algorithms that aim at maximizing our coverage metric (Sec. 5). This component ensures that all scenarios are exercised, and it is the cornerstone of our coverage-driven validation architecture.

To understand the execution flow of our tool, consider the pseudo-code in Listing 1. In the first step, the request generator queries the current state of the model and the desired coverage metric to generate the next request to execute. This is marked with bullet ① in the picture and the pseudo code. Next, in step ②, the request is sent for execution to the SUT and its response is recorded in `s_resp`. Because of the possible non-determinism in the server, the server response is sent to the model alongside with the request in step ③. To understand this, consider a simple `PutObject` interaction in the request/response excerpts shown in Listing 2. At the top we have the request issued, in this case a `PUT` request. While all fields of the requests are calculable at request generation time, that is not true for all the response fields at model execution time. In fact, the model state doesn't have the necessary information to compute the values marked in red: `x-amz-id-2`, `x-amz-request-id`, `Date` and `ETag`. The model considers these values to be non-deterministic and treats them as opaque values. While some of these values are not important to the model (for example `x-amz-id-2` and `x-amz-request-id` are used only for logging on the server side), some of them are critical for the functionality of the service, and need to be captured into the model. An example is the `Etag` value returned for stored objects. The object `Etag` is created when the object is uploaded, and can later be used in conditional operations via headers like `If-Match` and `If-None-Match`. In other cases, like the response `Date`, the model cannot compute the exact value (e.g. predict the exact time at which the server will process the request), but expects the value to conform to

```
PUT /main.pdf HTTP/1.1
Host: bucket-for-paper.s3.us-east-1.amazonaws.com
Accept-Encoding: identity
User-Agent: aws-cli/2.17.34 ...
Content-MD5: /JcyfUI6ZR8eKZuf6t5M8w==
X-Amz-Date: 20250108T111733Z
X-Amz-Security-Token: (redacted)
X-Amz-Content-SHA256: UNSIGNED-PAYLOAD
Authorization: (redacted)
Content-Length: 621629

...payload...
%%EOF

HTTP/1.1 200 OK
x-amz-id-2: 7Ze3etUJjG02zh3Moft+Iv5MgyDSrFQjzhEyorQ...=
x-amz-request-id: XN5HVNKEYT6W7EZB
Date: Wed, 08 Jan 2025 11:17:35 GMT
x-amz-server-side-encryption: AES256
ETag: "fc97327d423a651f1e299b9feade4cf3"
Content-Length: 0
Server: AmazonS3
```

Listing 2: Request/Response example. Prophecy values are in red, blue values are edited for readability.

a specific format. To simplify the response comparison, we propagate these opaque values into the model, and we take them as input for model evaluation. This is shown at step ③ where the observed response is sent as an input to the model evaluation. Care must be taken when identifying which of these values should be used in the model. For that reason, we distinguish them as *prophecies*, and prophecy values are treated specially in our model. Moving on with our execution flow, in step ④ both the model and service response are sent to the validator for checking. Finally, in step ⑤, the result of the validation is reported to the user of our tool before restarting with the next request if any. There are two possible outcomes: either the responses agree and we continue, or they don't agree, and we report them as a *deviation* to the user. The rest of the paper uses the term deviation to refer to discrepancies between model and SUT. A deviation represents an error in the model, in the SUT or in both.

2.1 Technical Choices and Assumptions

Black-box Modeling. We take a *black-box* approach to modeling and validation. We cannot afford to make assumptions about the implementation which may change in the future, or may be invalid w.r.t. other implementations of the same API. Thus, our model is agnostic to the internal implementation choices of a specific SUT, and all of our modeling is based on input/output behaviors of the service. One caveat of this approach is that the model is not able to differentiate between different *internal service states* leading to identical input/output behavior, for example those resulting from components such as caches.

Stateful REST API. We consider services that provide a REST API where the result of an operation can be influenced by preceding operations. For instance, the result of a `GetObject` is affected by a successful prior `PutObject` on the same key. Therefore, the model of the service must also be stateful and retain sufficient information about the previous operations to be able to correctly compute the response for the next request. However, in the interest of simplicity, our model is: 1. *in-memory*, meaning that the state is not durable across instantiations, 2. *single-threaded*, meaning that API calls are executed one at a time in the order that they are received by the model, and 3. *performance agnostic*, meaning it does not reflect performance requirements of the implementation, such as latency or throughput.

Non-determinism. Our target systems are mostly deterministic: given the same state, if we issue the same API call, we observe the same behavior, and the same resulting updated state. Some exceptions to this rule are: *service-generated identifiers*, which depend on internal state we cannot observe; *timestamps*, which depend on when the request is received and how long processing takes; and *transient errors*, such as `500 InternalServerError` responses triggered by hardware failures. A second form of non-determinism arises across implementations. Each implementation is free to choose the order in which to process independent request attributes, and this order affects which error is returned. The model must accommodate this variation: for each request, it returns the set of valid error responses, and a correct implementation must return one of them. We additionally require each SUT to consistently return the same error. Therefore, we learn the choices made by each SUT and then the validation steps for the SUT fail if a different choice is observed. Sec. 3 exemplifies this.

2.2 End to End Simplified Example

We illustrate our approach end-to-end through a simplified version of the S3 `GetObject` API with only three request fields: `bucket`, `key`, and `checksumMode`. We use the simplest form of bucket: a key-value store mapping keys to object blobs. Each object has an associated checksum, which can be retrieved via `GetObject` by setting `checksumMode` to `ENABLED`.

To validate that an implementation of `GetObject` behaves correctly for every distinguishable combination of request and state, we decompose the validation task into four subgoals:

(a) enumerate the set of scenarios (abstract state-request pairs) necessary to cover all API behaviors,

and for each of these scenarios:

(b) bring both SUT and model into the state required by the scenario,

(c) issue a concrete request representing the scenario against SUT and model, and

Response	Condition
`400 InvalidBucketName`	syntactically invalid `bucket`
`400 InvalidKey`	syntactically invalid `key`
`404 NoSuchBucket`	valid `bucket` but it does not exist in account
`404 NoSuchKey`	valid `bucket` and `key`, but `key` does not exist in `bucket`
`400 InvalidArgument`	`checksumMode` is a string other than `ENABLED`
`200 OK` + payload	otherwise

Table 1: Simplified `GetObject` behaviors

(d) compare the response received from the SUT with the responses admitted by the model.

We describe subgoal (d) first, since it defines what *correct* means, and then revisit (a)–(c) in turn.

Expected outcome and comparison (d). For our simplified `GetObject`, the model's behaviors are described in Tab. 1. To illustrate the non-determinism of error responses, notice that the conditions can overlap on some requests. For example, a request with an invalid `bucket` and an invalid `key` admits both `400 InvalidBucketName` and `400 InvalidKey` as valid model responses. The validator declares success whenever the SUT response is among those allowed by the model, and reports a deviation otherwise.

Input scenarios enumeration (a). Enumerating all concrete request/state pairs is intractable: bucket and key names alone range over all possible strings. The main insight of our input generation strategy is that we only need to consider inputs that lead to distinct behaviors. For instance, S3 rejects all bucket names longer than 63 characters and a few such names suffice to validate the SUT on this behavior. To formalize this, we partition the values of each API request and state feature into equivalence classes, which we call *categories*. Each category is defined via predicates over the request fields and S3 state (see Sec. 4).

To make the explanation concrete, let us denote by V_Ω the model state, and let b, k and c represent the values for the `bucket`, `key`, and `checksumMode` fields respectively. In this example, we define the abstraction function via six Boolean predicates: (i) valid_bucket_name(b) holds iff b is a syntactically valid bucket name; (ii) bucket_exists(V_Ω, b) holds iff V_Ω contains a bucket named b; (iii) valid_key_name(k) holds iff k is a syntactically valid key name; (iv) key_exists(V_Ω, b, k) holds iff V_Ω contains an object with key k in bucket b; (v) checksum_mode_present(c) holds iff the request contains the `checksumMode` field (i.e. $c \neq$ `None`); and (vi) valid_checksum_mode(c) holds iff c is a valid value for the `checksumMode` (i.e. $c \in \{$`None`, `ENABLED`$\}$).

The predicates (i), (iii), (v) and (vi) are request predicates and do not depend on the state; the remaining state-dependent

predicates relate the request with the model state. An *input scenario* is a truth assignment over the six predicates, and there are $2^6 = 64$ such assignments. However, most of these are *spurious*: existing objects cannot have an invalid bucket name or an invalid key name, a key cannot exist in an unknown bucket, and if the `checksumMode` is not present, it must be valid. We capture these constraints as a conjunction of four implications, which we call the model invariant inv_Ω (Sec. 5.1):

$$\begin{aligned}
\text{bucket_exists}(V_\Omega, b) &\rightarrow \text{valid_bucket_name}(b) \\
\text{key_exists}(V_\Omega, b, k) &\rightarrow \text{valid_key_name}(k) \\
\text{key_exists}(V_\Omega, b, k) &\rightarrow \text{bucket_exists}(V_\Omega, b) \\
\neg\text{checksum_mode_present}(c) &\rightarrow \text{valid_checksum_mode}(c)
\end{aligned}$$

We enumerate non-spurious scenarios by computing all satisfying assignments of inv_Ω. For our example, this reduces the 64 scenarios to 21: 2 success scenarios with an existing bucket, key, and checksum mode either `None` or `ENABLED`; 1 error scenario with an existing bucket, key, and invalid checksum mode; 3 error scenarios with an existing bucket and valid but non-existent key; 3 error scenarios with an existing bucket and invalid key; 6 error scenarios with a valid but non-existent bucket; 6 error scenarios with an invalid bucket.

Most input scenarios lead to error responses, and many will produce the same error on the SUT. In practice, deterministic implementations check arguments in some order and short-circuit on the first error. Suppose our SUT first performs the syntactic checks on `bucket`, `key`, and `checksumMode` in this order, and then looks for the object with the given bucket and key names in the state. Under this evaluation order, several of the 21 scenarios map to the same SUT behavior. For example, the 6 input scenarios with an invalid bucket name all produce the same response (`400 InvalidBucketName`) regardless of the state or the other field values. We call this the *first-error hypothesis* and use it to reduce the number of scenarios further (Sec. 5.2).

State setup (b). Given a non-spurious scenario, we have to bring both the SUT and model into a state that satisfies all state-dependent predicates of the scenario. For instance, every scenario where bucket_exists holds requires the state to contain at least one bucket. We address this problem with a component called API-planner (Sec. 5.3). Given the state-dependent predicates of a scenario, the API-planner returns a sequence of requests that lead from the current state to a state that satisfies the input predicates.[3] Consider a success scenario with both predicates bucket_exists and key_exists set to true. The scenario requires V_Ω to contain a bucket with a valid name, and in it an object with a valid key. Assuming the current state (V_Ω) is empty, the API-planner produces two preparatory requests: 1. `CreateBucket`(b) with a freshly synthesized valid bucket name b, and 2. `PutObject`(b, k, *payload*) to write an object with freshly synthesized valid key k and some content *payload* into bucket b. Each preparatory request is itself a test and it is processed end-to-end through our validation pipeline as described in Fig. 1. Once the planner reports that the target state is reached, we proceed to test the `GetObject` input scenario request of interest.

[3] Since we only exercise non-spurious scenarios, we only ever call the API-planner with non-contradictory predicates.

Request concretization (c). With the state in place, the last step is to pick concrete values for the request features that are consistent with the input scenario. For request predicates we sample from the category dictated by the scenario: e.g. if valid_bucket_name must be false, we sample from a pool of syntactically invalid names (too short, too long, illegal characters, ...). Instead, for state-dependent predicates we query the model state for values that satisfy them. For example, if both bucket_exists and key_exists must hold, we ask the model for an existing (bucket, key) pair, and reuse the names synthesized during state setup. Finally, the `checksumMode` is: 1. omitted from the request if checksum_mode_present is false, 2. set to `ENABLED` if both checksum_mode_present and valid_checksum_mode hold, and 3. sampled from a pool of invalid values (e.g. empty string, `DISABLED`, or some other invalid string) if valid_checksum_mode is false. The resulting `GetObject` request is then executed according to our description of step (d) above.

3 The Model

Our model is intended to serve as the de facto specification of the S3 API. As a result, service owners need to be familiar with the model code base and extend it in anticipation of changes in the service. Practically, the choice of programming language and development methodology for the model follows the same standards as the majority of our service code bases. Consequently, we implement our model in Java, and we use the standard builder tools available to engineers.

Our model implements a key-value store with additional metadata to represent details about the stored objects. For simplicity, we only store information in the model if it can be observed by subsequent API calls. For example, the model records the object's `ETag`. The `ETag` is computed when the object is created, and later `GetObject` operations for the object will return the same `ETag`. However, the model does not store the operation's request identifier (`x-amz-request-id`) since there is no way of retrieving it via API calls. Some examples of common metadata that we store in the model alongside the object key and payload are: object creation time, object tags, server-side encryption mode, encryption key for the object (if any). Each of these can affect the behavior of subsequent operations, and hence the model must have access to this information. As a concrete example, the encryption key in a `GetObject` request needs to match the one used at

creation time for the request to succeed; otherwise a `403 Access Denied` error code is returned.

We strive to reuse model code and functionality when implementing similar features for different APIs. As an example, consider the conditional operation features present in the `GetObject`, `PutObject`, `CopyObject` and `DeleteObject` APIs. These features validate the same syntactic and semantic conditions on the arguments and state of the system. Therefore, our model encapsulates the behavior of each feature in a single *trait*. This reduces the size and complexity of our model, and it makes reasoning about each individual feature uniform across all APIs that use them. As we shall see in Sec. 5, this also enables us to abstract the input generation strategy for each individual feature, which then generalizes for all the APIs that use the feature.

Non-determinism and Prophecy Variables. The model exhibits non-determinism in two ways: incorporating prophecy values from the SUT response and allowing different input validation orders. For example, consider the following `GetObject` request:

```
GetObject(bucket="non-existing", PartNumber="abc")
```

This request leads to an error response since: 1. the bucket name, while syntactically valid, does not correspond to an existing bucket and 2. the part number is not a positive integer. Different implementations can perform these checks in different orders. For instance, Amazon S3 performs the syntactic checks on the part number before checking whether the bucket exists, and replies with the error `400 InvalidPartNumber`. However, a different implementation of `GetObject` could choose to report `404 NoSuchBucket` instead. Our model generates all the acceptable error responses and the SUT response is correct if it is among the ones generated by the model. This raises the question of checking a mostly-deterministic service with a non-deterministic model. A specific SUT must be consistent in its responses and always pick the same answer among the ones allowed by the model. This property is important for customers who may have error handling routines for errors they observed during testing. If the SUT suddenly reports a different error for the same request, customers' code may break in production. We validate that each SUT keeps returning the same error by learning a partial order for each SUT. The partial order describes which error the SUT returns among the ones allowed by the model. We then use the precedence order to specialize the model for the specific SUT. This enables our tool to validate that the SUT retains the same ordering without having to build a separate model.

4 Request and State Abstraction

The effectiveness of our model-based validation rests on its ability to exercise tests that show whether the model and implementation coincide (step ① in Fig. 1). Our goal is to explore all SUT behaviors that are distinguishable from a black-box viewpoint. As anticipated in Sec. 2.2, we use predicate abstraction [GS97] to focus our validation efforts on meaningful input scenarios and, consequently, on tests that exercise distinct behaviors.

Terminology and definitions. Tab. 2 reports some terms that we use throughout the section. We use *feature* generically to refer to either an input parameter (*request feature*), or a configuration of a logical component of the state of the system (*state feature*). For example, in `GetObject` the bucket name of the request is a request feature. In contrast, state features correspond to elements of the state that influence the behavior of the API in an observable way. For example, whether a bucket name is in-use (i.e. a bucket that has been successfully created and has not been deleted yet) is a state feature.

We group together values of a feature that result in similar behavior when considering specifically that feature.[4] We define *categories* (equivalence classes) of values for a feature, with the expectation that values in the same category behave similarly. Considering bucket name as an example, a bucket name can fall into one of three equivalence classes: 1. syntactically invalid names, 2. syntactically valid names for non-existent buckets, or 3. valid names for existing buckets. Each class triggers distinct service behaviors and outputs. The equivalence classes partition the possible values of a feature. Therefore, we can exercise the relevant behaviors by selecting one value from each category instead of trying all possible values. An *input scenario* maps each feature to a corresponding category. Each input scenario is a truth assignment for a set of predicates over the request and state features of the API. Input scenarios reduce the set of all possible requests that need to be exercised to a much smaller finite set that still exercises all meaningful behaviors. This transforms an intractable exploration problem (exploring all possible values) into one amenable to algorithmic automation. We apply the abstraction across all API features and dramatically reduce the universe of scenarios to exercise while preserving comprehensive behavioral coverage. Consider the categories for bucket names as shown in Fig. 2. The left diagram in the figure shows how the space of all possible bucket names is partitioned according to the three categories. Whether a bucket name is well-formed is strictly a property of the bucket name string. Therefore, it is a request feature and no information about the state is needed to decide it. On the other hand, the existence of a bucket depends on the state and it is a state feature. We assume that for each feature, we can categorize its values such that if two values are in the same category, they drive the same SUT behavior.

For each category we introduce a corresponding predicate that holds iff the value is in the particular category. We use

[4] It is sometimes useful to think of a feature as a type, limiting the kind of values that it can accept.

Request / concrete request		Request readily executable against the service.
State / concrete state		Current state of the model / SUT.
Deviation		Request and state where the model and SUT do not agree on the response.
Feature	Request	Argument or contextual parameter that affects strictly the request. E.g. the bucket name.
	State	Contextual parameter or condition on the state that affects the request. E.g. if a bucket with a specific name exists.
Predicates	Input	Set of boolean functions / formulae over model state and action. E.g. "is_valid_bucket_name" or "bucket_exists".
	Output	Set of boolean functions / formulae over response / output. E.g. "status_code_200" or "no_such_bucket".
Input scenario		Total truth assignment over the input predicates. Denotes set of states and requests that are *equivalent* with respect to the predicates.
Projections	Abstraction	Maps a concrete request and state to the corresponding input scenario. E.g. the bucket name "ab" maps to the category of invalid buckets.
	Concretization	Function that maps an input scenario to concrete request and states in its denotation. E.g. the category of invalid bucket names maps to the concrete bucket name "bc" and others.
Category	Request	Set of values for a request feature that are equivalent w.r.t. the input predicates.
	State	Set of values for a state feature that are equivalent w.r.t. the input predicates.

Table 2: Terminology

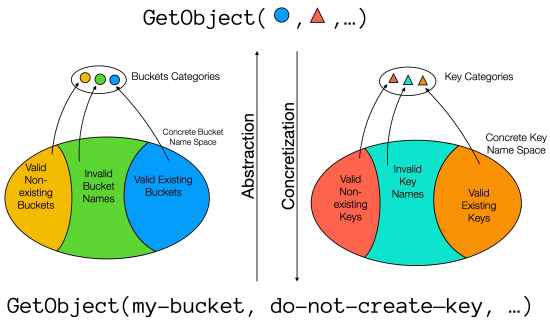


Figure 2: Request abstraction

predicates on inputs, outputs, and state elements to characterize the space of behaviors of the system. A predicate may depend on more than one request feature. S3 buckets contain objects identified by a key. Therefore, the predicate key_exists, holding only when an object with the given key exists, depends on the state of the system, as well as the request's bucket and key names.

The *abstraction* function evaluates the predicates on a particular request and state pair. The *concretization* function does the inverse: given an input scenario, it generates concrete request and state pairs, producing an executable test case. Fig. 2 depicts the process of mapping a concrete request to an input scenario and vice versa. Notice that sampling concrete requests from input scenarios is a critical building block to obtain a concrete test suite that achieves our coverage goals. We generate a set of input scenarios that achieves our coverage goal; concretization then turns each scenario into a concrete request executable on both SUT and model, thereby producing a test suite that realizes the coverage goal.

More formally, let $\mathbb{P}_I$ be the set of input and state predicates and $\mathbb{P}_O$ be the set of predicates on the output / responses. Then, $2^{\mathbb{P}_I}$ is the set of all the possible assignments to the input predicates, hence the set of all input scenarios. We say the abstraction is *adequate* iff every concretization of a given input scenario leads to outputs that correspond to the same assignment to the predicates in $\mathbb{P}_O$. In this case we say that the set of predicates $\mathbb{P}_I \cup \mathbb{P}_O$ is *fine*. Therefore, the input predicates are sufficiently precise to distinguish request-state pairs leading to distinct replies (w.r.t. $\mathbb{P}_O$). The abstraction is adequate if two request-state pairs that lead to responses that correspond to different evaluations of the output predicates $\mathbb{P}_O$ are mapped into distinct assignments to the input predicates $\mathbb{P}_I$.

To add structure to our abstract scenario space we take into account the *kind* of response (success or error) that the scenario will lead to on the SUT. This is simplified because web services reply to every request with either a success response or an error response.[5] For instance, ¬bucket_exists establishes a priori that the `GetObject` request cannot succeed, and moreover, if no other error-inducing predicates are true of the input scenario, it defines the exact error case that we will observe (in this case `404 NoSuchBucket`). This addition allows us to 1. use the model to predict whether the requests should succeed or not, and therefore focus our input generation for requests that should succeed first, and 2. define metrics on error-inducing requests, based on their proximity to a successful request. We define the distance between an error-inducing request and a success request as the number of parameters whose evaluation leads to error. We refer to this distance function as num_errors. For instance, `GetObject(bucket="1b", key="", PartNumber="abc")` induces at least three errors: the bucket name is too short, the key name is empty, and the `PartNumber` is not a number. num_errors counts the total number of error preconditions of an input scenario. We include it in our abstraction to distinguish between requests that are expected to succeed (num_errors $= 0$), requests that hit exactly one error condition (num_errors $= 1$), requests that can trigger two distinct error preconditions (num_errors $= 2$), and so on. Sec. 5.2 describes how this enables us to reduce the number of inputs that need to be validated to achieve coverage of the input scenarios via the *first-error hypothesis*.

4.1 Predicate Definition

Consider the example in Fig. 3. The picture shows three HTTP requests for key "`key`" in bucket "`bucket`". The requests specify different `If-Unmodified-Since` dates and

[5] For simplicity, we assume that 2XX HTTP responses [RFC22] are successful, and other responses (errors or informational) are considered errors.

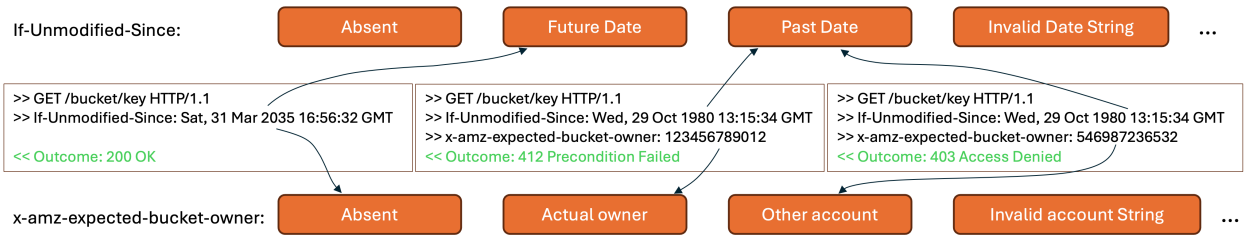


Figure 3: Abstraction example.

`x-amz-expected-bucket-owner` values. The response received for each request is shown below the request in green. The semantics of `If-Unmodified-Since` is such that: 1. if the date is in the future the header is ignored; 2. if the date is before the creation date of the object we will receive a `412 Precondition Failed`; 3. otherwise the request can succeed with a `200 OK` response if no error happens elsewhere. The `x-amz-expected-bucket-owner` header is used to ensure we only operate against a bucket owned by the specified account. If the account does not own the bucket a `403 Access Denied` error is returned. This example illustrates the importance of the abstraction: the exact choice of future date or exactly which "wrong" owner we specify does not matter. We only need to test one future date for this scenario, not all possible ones. Similarly, for the `x-amz-expected-bucket-owner` field we only need to test the actual owner, any other account as owner, and at least one other value for each of the remaining categories shown at the bottom of the figure. Importantly, the choice of category for a feature can drive different outcomes for the request, while different concrete values in the same category will produce the same responses.

Identifying Meaningful Categories. A natural question is how to determine the appropriate set of categories of each feature. For instance, should we refine the "invalid bucket names" category further? The choice of categories is driven by the following observations. 1. Whether all the values in the category result in the same behavior for the feature in question. 2. Whether a category represents values that are potentially difficult to handle, even if it may not change the observable behavior. Typical examples are control characters in strings, empty or null values, boundary conditions, etc. 3. Whether the category contains values that are relevant for the specific optimizations and design choices of the SUT. While in this case, the category may not change the observable behavior, developers with information about the SUT can provide knowledge about inputs that trigger a different SUT path.

While some of these categories are immediately derivable from observations of the system and documentation, others need expert knowledge of the system. Over time, engineers add more information to our abstraction that relates to server paths that are not distinguishable by the model, but nonetheless deserve to be validated separately. For example, the object size of a request may affect the path, while the observable behavior is unaffected. Crucially, this has no effect on the model, but in the validation strategy we may be explicitly interested in covering such cases.

5 Automated Input Scenario Generation

Our test generation strategy uses the request and state abstraction introduced in Sec. 4 to exercise a configurable set of input scenarios. A test configuration defines the predicates that we are interested in and bounds the minimum and maximum num_errors of the input scenarios. The configuration is a boolean formula over $\mathbb{P}_I$ and num_errors. The test generation procedure explores all input scenarios in the configuration by enumerating the truth assignments that satisfy the formula. When the configuration includes all the features of the API type (the formula is valid), the test generation strategy will cover all the input scenarios for that API. Since input scenarios are equivalence classes of the observable behavior of the system, executing one request for each input scenario gives us the best black-box approximation of the service execution paths according to our abstraction predicates $\mathbb{P}_I$ and $\mathbb{P}_O$.

While this goal is intuitively simple, in many cases covering the entire space of input scenarios makes the procedure exercise scenarios that are *redundant* or *spurious* as discussed in Sec. 2.2. An input scenario is *redundant* if it covers behaviors of the SUT that have been already exercised by other scenarios. An input scenario is *spurious* if there are no concrete tests that can satisfy the category combination required by the input scenario. For example, the set of input scenarios $2^{\mathbb{P}_I}$ contains inputs that combine invalid buckets with values in all categories for key names: including the category of existing keys! In S3, there cannot exist keys in a bucket with an invalid bucket name, hence the input scenario cannot be concretized. There exists no state and request pair that meets the requirements of the scenario.

Recall the goals introduced in Sec. 2.2. We want to exercise all non-spurious input scenarios in a configurable portion of the space. To that end, we first need to enumerate the relevant (non-spurious) input scenarios. Then, for each scenario, we must reach a state and generate a request that exercises it on the SUT. Therefore, for every input scenario we execute a sequence of requests. The prefix of the sequence contains the requests necessary to reach the necessary state. The last request of the sequence is the one that actually exercises the

input scenario we are interested in. We send every request to the SUT first and then we check whether the responses are admissible by sending the same requests to the model (Fig. 1). We call the process of computing the sequence of requests to exercise an input scenario *concretization*.

5.1 Removing Spurious Input Scenarios

As exemplified in Sec. 2.2, we introduce a propositional formula inv_Ω (for state invariant) to distinguish spurious and non-spurious input scenarios. We first show what inv_Ω looks like in the case of Amazon S3 and then give a more general description of its shape and how it is employed in the scenario enumeration procedure.

For this example, consider the `bucket`, `key` and `version id` features of the `GetObject` API. In this setting we define inv_Ω as the conjunction of seven implications. Implications 1 and 2 express the fact that in S3 keys exist only inside buckets, and a key can have multiple versions. Implication 3 states that every key has the null version. Then, implications 4, 5 and 6 state that only buckets, keys and version ids that are syntactically correct can exist in the state. Finally, implication 7 encodes the fact that a version id cannot be both named and null at the same time. More formally, let V_Ω be the model state and let b,k,v represent the values of `bucket`, `key`, and `version id` respectively. Consider the predicates valid_bucket_name, valid_key_name, valid_named_version, null_version, bucket_exists, key_exists, version_exists $\in \mathbb{P}_I$. Then, inv_Ω is the conjunction of the following formulae:

$$\text{key_exists}(V_\Omega,b,k) \rightarrow \text{bucket_exists}(V_\Omega,b) \quad (1)$$

$$\text{version_exists}(V_\Omega,b,k,v) \rightarrow \text{key_exists}(V_\Omega,b,k) \quad (2)$$

$$\text{key_exists}(V_\Omega,b,k) \wedge \text{null_version}(v) \rightarrow \text{version_exists}(V_\Omega,b,k,v) \quad (3)$$

$$\text{bucket_exists}(V_\Omega,b) \rightarrow \text{valid_bucket_name}(b) \quad (4)$$

$$\text{key_exists}(V_\Omega,b,k) \rightarrow \text{valid_key_name}(k) \quad (5)$$

$$\text{version_exists}(V_\Omega,b,k,v) \rightarrow \text{valid_named_version}(v) \vee \text{null_version}(v) \quad (6)$$

$$\text{valid_named_version}(v) \rightarrow \neg\text{null_version}(v) \quad (7)$$

More generally, we require the atoms of inv_Ω to be the abstraction predicates $\mathbb{P}_I$ and we require inv_Ω to denote the set of non-spurious input scenarios. An input scenario is a truth assignment over the atoms of inv_Ω, hence it makes the formula either true or false. In other words, inv_Ω is satisfied by an input scenario iff the scenario is non-spurious. Therefore, we can generate the non-spurious input scenarios by enumerating the satisfying assignments of inv_Ω. In fact, the core of the input scenario generation procedure is an *all-sat* engine that enumerates the truth assignments to the predicates $\mathbb{P}_I$. The *all-sat* procedure can be implemented using any off-the-shelf SAT solver [CKHM25]. The input scenario generation procedure takes as input the invariant inv_Ω and the test configuration. It explores all non-spurious scenarios in the given configuration by enumerating all satisfying assignments (*all-sat*) of the formula obtained by the conjunction of inv_Ω and the test configuration. The conjunction of the test configuration with inv_Ω prevents the procedure from generating scenarios that are spurious and cannot be concretized.

5.2 Removing Input Scenario Redundancy

APIs often involve many, sometimes optional, input parameters. Unfortunately, the number of input scenarios is exponential in the number of features of an API. This implies that in several cases it may be neither practical nor cost-effective to explore all input scenarios. Instead, we are interested in exploring specific aspects or features of the API. Generally, the time available for validation is bounded (e.g. in a CI/CD approval step). Therefore, we need to limit our search to fit the time budget of the development cycles. We must focus our validation on features and scenarios that are more likely to exhibit new behaviors, potentially deviating from the model. For example, consider the `GetObject` API with its 21 input parameters. For simplicity, assume we define only 2 independent predicates for each feature (i.e. 4 categories), then we have 4^{21} input scenarios for `GetObject`. Since our tool executes around 1.5×10^5 requests per hour, it would take about 2.9×10^7h to execute all `GetObject` scenarios. This simple computation shows that the predicates $\mathbb{P}_I$ alone do not sufficiently focus our exploration for all APIs. For this reason, we exploit num_errors to reduce, in a safe manner, the cases that need to be covered to identify deviations.

As exemplified in Sec. 2.2, covering all input scenarios would exercise the same behavior multiple times. This is especially true in the case of requests leading to error responses. Consider again the `GetObject` request from Sec. 3:

```
GetObject(bucket="non-existing", PartNumber="abc")
```

In Amazon S3, this request will result in a `400 InvalidPartNumber` error, indicating that the part number check took precedence over the bucket existence check. The SUT stops processing the request and immediately returns once the first error is found. The values of all other fields are ignored and don't affect the SUT behavior. We assume that error responses short-circuit the service execution, and dub this property the SUT *first-error hypothesis*. We use the hypothesis to focus our exploration policy further.

num_errors, defined in Sec. 4, helps us exploit the short-circuiting mechanism. We use it to reduce the number of requests we generate, while still exploring all SUT behaviors under the first-error hypothesis. For instance, input scenarios where more than one input would lead to error induce redundancy: one of the two inputs will get evaluated first and the other will be ignored. However, our goal is to ensure conformance between model and SUT. Therefore, we cannot just stop after exercising all requests with up to 1 error. We need to

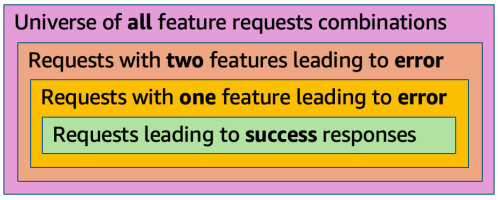


Figure 4: Space of requests by errors.

exercise requests with 2 errors to ensure the ordering in which the features are evaluated by the SUT is compatible with the model expectation. For example, all authentication and authorization checks must occur before all state-dependent checks (e.g. existence of the key). We use this strategy to organize our space exploration as illustrated in Fig. 4, where each set contains all the scenarios with up to n errors.

0-errors campaigns. These include all input scenarios leading to success responses.

1-error campaigns. These are all input scenarios that contain exactly one feature leading to error. These campaigns validate that our error reporting is accurate, and in combination with the 0-errors campaigns above, if the first-error hypothesis holds, evaluate all SUT paths for the API.

2-errors campaigns. These cover all input scenarios with exactly 2 features that should lead to an error. They validate that our model predicts the parameter evaluation precedence when more than one parameter leads to error.

Under the first-error hypothesis, any campaign with at least 3 errors exercises the exact same behaviors observed in the 1-error campaigns, and contributes nothing to the validation of the error ordering already achieved by the 2-errors campaigns.

In some instances, accurately calculating the exact num_errors for an input scenario is quite expensive. Intuitively, this requires identifying the closest input scenario leading to success, which implies considering all alterations for all predicates. While we generally keep the num_errors definition precise, for larger numbers n of num_errors we accept some requests with $m < n$ errors, if that speeds up the search. Notice that this is a safe overapproximation of the space of requests with n errors, since it only adds cases which may have less than n errors. The set will contain all scenarios with n errors.

5.3 Input Scenario Concretization

The previous sections described how we enumerate the input scenarios that achieve the desired behavior coverage. Unfortunately, we cannot directly execute an input scenario on the SUT. We must first set up the state required by the scenario and then create a corresponding request. For example, consider a `GetObject` input scenario where both bucket_exists and key_exists hold. To exercise the scenario, we first need a state with an existing bucket b and an object in b with some key k. Then, we can exercise the scenario by executing the request `GetObject(bucket="b", key="k", ...)`.

Listing 3 details the process of exercising an input scenario on the SUT and performs four main steps. First, we analyze the input scenario and extract the description of the required target state (`make_target_state`). Second, we execute the (possibly empty) sequence of operations necessary to reach the target from the current state (the while loop in `execute`). Third, we inspect the state and generate the request we need to execute (`concretize_request`). Finally, we execute this request on both model and SUT, exercising the input scenario (the last call to `test` in `execute`).

We now describe how the steps are implemented. The state required by an input scenario depends on the assignment prescribed by the scenario to the input predicates that depend on the model state. Therefore, given an input scenario we analyze its truth assignment to all predicates in $\mathbb{P}_I$ that depend on the model state and compile it into a structure we call *target state*. The target state is an abstract description of the state necessary to execute the input scenario. In the case of S3, the target state defines the number of buckets needed and, for each bucket it specifies the required configuration, keys and objects it must contain.

Given the target state, we must compute the sequence of operations necessary to set it up. As we mentioned in Sec. 2.2, that is the task of the *API-planner* component. Given the target state, the API-planner produces the sequence of actions necessary to reach the target from the current state. API-planner is a function that, given the current state of the model and the target state, generates the next action we need to perform or `None` if the current state meets the target. Recall the example from Sec. 2.2. Starting from an empty state, we want to exercise a `GetObject` input scenario where both bucket_exists and key_exists hold. The API-planner will first synthesize a valid bucket name and create a bucket with such name via the `CreateBucket` API. After successful execution of the `CreateBucket` request, the API-planner synthesizes a key name and object payload. These values are used to generate a `PutObject` request to write an object in the bucket we just created. Upon successful execution of the `PutObject`, the state has both a bucket and a key. We've now reached our desired state and the API-planner returns `None`. In order to exercise the scenario, we need to synthesize the corresponding request. In our example, we need to generate a `GetObject` request that uses the bucket and key names that we just created. We achieve this by querying the current model state for an existing bucket and key.

Importantly, the API-planner assumes the SUT and the model agree on the behavior of all *write* APIs (`CreateBucket` and `PutObject` in our example). We validate this by comparing the model and SUT responses for all requests generated by the API-planner and by running test campaigns specific for `PutObject` and the other APIs. In general, if the SUT and

```
def test(request: Request, model: Model) -> bool:
  sut_response = execute_sut(request)
  prophecies = make_prophecies(sut_response)
  model_response = model.execute(request, prophecies)
  return sut_response == model_response

def execute(model: Model, scenario: Scenario) -> bool:
  target: TargetState = make_target_state(scenario)
  request: Request = api_planner(model, target)
  while request is not None:
    if not test(request):
      raise StateSetupFailure()
    request = api_planner(model, target)
  request = concretize_request(model, scenario)
  return test(request)
```

Listing 3: Scenario concretization and execution (pseudo).

model do not agree, our validation campaigns will highlight defects either during the state-setup sequence, if SUT and model reply differently to a request generated by the API-planner, or when validating the response to the final request that exercises the desired scenario.

Listing 3 decomposes the scenario concretization process in two methods: `test` and `execute`. The `test` method simply executes the given `request` on both the model and SUT, and it returns true iff they generate the same response. As described in Sec. 3, the `prophecies` structure is used to enable the model to replicate the decisions made by the implementation (e.g. set the `Date` in the response correctly). The `execute` method sets up the state required to exercise the input scenario, generates the request corresponding to the scenario, and executes the request. The method returns true iff both SUT and model agree on the response to such request. The `execute` method first extracts the state requirements from the scenario into the corresponding target state. Then, it keeps calling `api_planner` to retrieve the next request to be executed in order to set up the state. It relies on the `test` method to execute the request on both model and SUT. If model and SUT reply differently, it notifies the caller that there has been a failure during the state setup. Once the desired state has been reached, `concretize_request` generates a request that exercises the given scenario in the current model state. Finally, it calls `test` one last time to run the request on both model and SUT and returns false iff a deviation was found.

5.4 Configuring Validation Campaigns

The input scenario generator takes as input a configuration that defines the testing campaign. The configuration defines the input scenarios of interest, explicitly defining the features and predicates that must be completely explored. We strategically define test campaigns and the corresponding configurations with the help of engineers. We define configurations that exploit knowledge about un/correlated features in the SUT. The configurations limit the combinatorial complexity of the validation campaigns and make them fit in the computational budget. For example, conditional features (`If-Modified-Since`, `If-Unmodified-Since`, `If-Match`, `If-None-Match`) are related to each other, and are also related to the freshness of the object. However, conditionals are mostly independent from other features, such as bucket configurations, and fields like `x-amz-expected-bucket-owner`. For this reason, for time budgeted campaigns, we restrict the features to be tested according to their correlation.

6 Results

Our model-based validation infrastructure is currently employed in CI/CD pipelines of S3. Tab. 3 reports the number of findings of our tool on three major initiatives within S3. Each of these findings resulted in proactive fixes and prevented the deployment of model-breaking code changes. Each finding has been manually analyzed and assigned a severity score. For each initiative the table reports the total number of findings and the number of findings with high severity.

The S3 Express One Zone storage class launched in 2023. Prior to launch, we applied our approach to all the S3 APIs that require strict compatibility with the original regional S3 implementation. For the APIs with intentional changes, we adapted our model to reflect the expected behavior on directory buckets. Using our methodology we prevented 171 deviations. Of these, 12 were of high severity. All these defects were fixed before the service launch in 2023.

Our tool is used in an ongoing multi-year rewrite of S3's frontend API service. As part of this effort, developers run it locally in their development environments, and it is also integrated into the CI/CD pipeline. Local runs typically exercise a small, specifically defined, subset of the overall tests that are run in the CI/CD steps. Over the course of the rewrite, the CI/CD steps identified 92 issues that were missed by other unit and integration tests. This enabled developers to address all of them before deploying to production.

Finally, as part of the CI/CD process in S3 we continuously check for errors and prevent them from releasing into production. To date, we identified 109 deviations, of which 24 have high severity. In the past two years, our tool helped in validating several major new feature launches such as: enabling server-side encryption by default, support for full object checksums, and conditional write operations (e.g. the `If-Match` field for `PutObject` requests).

6.1 Validation Reductions

Tab. 5 reports an overapproximation of the number of input scenarios for the `GetObject` API obtained from the number of features and categories in Tab. 4. The number of scenarios is still too large to be executed in the CI/CD pipeline. Each approval step in the CI/CD pipeline must provide feedback within a reasonable amount of time. As a starting point, we allocate a budget of three hours per validation campaign and run multiple campaigns in parallel. For this reason, in our testing setup each campaign can execute up to approximately

Implementation	# Issues	High Sev
S3	109	24
S3 Express One Zone	171	12
S3 Refactor	92	10

Table 3: Issues prevented to date.

Feature Type	Features	Categories
Request	21	102
State	16	33
Total	37	135

Table 4: Number of features and categories for the S3 `GetObject` API.

# errors	Scenarios (approx)
0 Errors	10^{12}
1 Errors	10^{14}
2 Errors	10^{15}
Any	10^{25}

Table 5: Approximate number of input scenarios for `GetObject` by number of errors.

Groups	Features	0 Errors	1 Errors	2 Errors
Ranges	`Bucket(C)`, `Key(C)`, `VersionId(C)`, `Size`, `PartNumber`, `Range`, `ChecksumMode`	27000	92700	92700
Conditionals	`Bucket(C)`, `Key(C)`, `VersionId(C)`, `If-Match`, `If-None-Match`, `If-Modified-Since`, `If-Unmodified-Since`	640	4707	3984
Encryption	`Bucket(C)`, `Key(C)`, `VersionId(C)`, `Size`, `SSECustomerKey`, `SSECustomerAlgorithm`, `SSECustomerKeyMD5`	86400	527040	1270080
Response*	`Bucket(C)`, `Key(C)`, `VersionId(C)`, `ResponseExpires`, `ResponseCacheControl`, `ResponseContent*`	279936	0	0

Table 6: Sample grouping of correlated features for validation and approximation of number of input scenarios.

432000 requests. For the features of the `GetObject` API, we defined an average of 2.4 equivalence classes that can be used in successful requests. Therefore, in an approval step we can cover the success-case combinations of about 15 features ($\approx \log_{2.4}(432000)$). Developers define the combination of features to consider based on their knowledge of the SUT and the changes they are interested in validating. We then rely on asynchronous long-running validation tasks to explore a rotating subset of the features with the goal of highlighting defects resulting from unexpected interactions between features.

We also have predefined validation campaigns, built alongside senior developers, that couple together highly correlated features. Tab. 6 reports a sample of common groupings of interest. For each group it reports the number of input scenarios containing 0, 1, and 2 errors that cover the feature combinations. Consider the first group (Ranges) in Tab. 6. We have, as in most validation configurations: `Bucket(C)`, `Key(C)`, `VersionId(C)`, where the `(C)` stands for correct categories for the group since we are interested in the object structure in this case, not bucket and key names, which are validated elsewhere. We also have features related to the size and structure of the stored object: `Size`, `PartNumber`, `Range`, and `ChecksumMode`. This results in 212400 requests, which we can execute in about 1.5 hours.

While we still don't cover all the input scenarios on every change, we highlight that the number of cases being covered is well beyond what can be achieved with manually defined tests. In addition, our approach is flexible and the scenarios being exercised can be changed via simple configuration updates.

6.2 Comparison to Property Based Testing

We compared our test generation approach to property-based testing (PBT). In the PBT setup the input generator samples each parameter from a weighted distribution and runs `GetObject` requests against randomly generated objects in an S3 bucket. We then apply our abstraction function to each request's input and output parameters and count the unique input scenarios covered. In a run of 28457 `GetObject` requests, the PBT tests exercise only 9040 unique scenarios, meaning that 19417 requests were scenarios already covered.

We obtain a similar result when targeting a specific scenario of interest. In this case we selected 3 features of `GetObject`: `bucket`, `key`, and `range`, and we allowed 2 categories for each, totaling 8 non-spurious input scenarios. The PBT generator eventually covers all 8 scenarios. However, across 10 different runs it required 3200 requests on average to do so. Instead, our scenario generation procedure directly enumerates the distinct input scenarios and deterministically generates the 8 required requests.

These simple experiments highlight the additional cost incurred to achieve the same coverage goals without a systematic way of exploring the input space.

7 Related Work

The literature more closely related to our work is on the topic of automated test generation and model-based testing. Tab. 7 organizes tools in this large research field along 4 orthogonal dimensions. 1. The *testing approach* is white-box if the tool leverages the source code of the system, and black-box otherwise. 2. The *test-generation policy* is the algorithm used to select the next test scenario. 3. The *input generation* represents the logic used to select the input value for each scenario. 4. An approach is *stateful* if the test generation considers the current state of the SUT, stateless otherwise. 5. The *oracle* is the component that performs the assertions to validate the

Tool	Testing approach	Test generation policy	Input Generation	Statefulness	Oracle
APIFuzzer [api]	Black-Box	Random	Mutation	Stateless	Status Code
EvoMaster [Arc19] (BB)	Black-Box	Random	Random	Stateful	Status Code
EvoMaster [Arc19] (WB)	White-Box	AI, Evolutionary	Mutation, Dynamic	Stateful	Status Code
Fuzz4All [XPT+24]	Black-Box	AI	AI	Stateless	Status Code
Grammarinator [HKG18]	Black-Box	Grammar-based	Random	Stateless	Status Code
RestTestGen [VDC20]	Black-Box	Heuristic Dependency-based	Mutation, Example, Random, Dynamic	Stateful	Status Code
Schemathesis [HD22]	Black-Box	Property-based	Random, Example	Stateful	Response validation
SibylFS [RST+15]	Black-Box	Model-based	Combinatorial, Manual	Stateful	Full Response
Tcases [tca]	Black-Box	Combinatorial	Only Scenario Generation	Only Scenario Generation	Only Scenario Generation
VIVAS [FFP+23]	Black-Box	Model-based	Symbolic	Stateful	Abstract Response

Table 7: Classification of API testing tools

correctness of the observed behavior.

Grammar-based approaches, such as Grammarinator [BCD23, HKG18], exploit an explicit grammar to define the set of inputs. The space of all possible scenarios is defined by the possible combinations of derivation rules in the grammar, and the tools use randomization to explore the productions of the grammar. Unfortunately, grammar-based input generation is not well suited for stateful systems.

There is extensive literature on automated test generation for REST APIs [GZA24, KXSO22, CZPC21, MASR21, HD22]. Tools such as APIFuzzer [api], EvoMaster [Arc19], RestTestGen [VDC20, CZPC22], Schemathesis [HD22], and Fuzz4All [XPT+24] aim at testing an arbitrary REST API. Therefore, they often use comparatively weak oracles that don't enforce any specific behavior and perform only shallow checks on the response. Moreover, most of them rely on randomization or best-effort GenAI approaches for test generation, and do not provide conclusive coverage metrics on the covered space of requests.

The VIVAS framework [GGT25, FFP+23] relies on the definition of an abstraction of the system and the coverage goal is defined via (temporal) properties in the abstract space. The oracle is defined by runtime monitors [CTT19] synthesized from a set of desirable properties of the system on the abstract space. VIVAS focuses on ensuring that all traces of the system satisfy a set of properties. Instead, we focus on a simpler setting and consider the validation of a single action at a time executed in different SuT states. Our oracle is the full system model, defined directly on the concrete space, while in VIVAS the oracle is given by abstract properties.

Combinatorial Interaction Testing (CIT) [CDS08, NMT12, HWKK20, DMDS24] and tools like Tcases [tca] also organize the input space in features and categories. The tool Tcases simply generates the input scenarios. How to incorporate the scenarios into a practical testing campaign for a SuT is left to the user. Tcases employs a generate-and-filter logic to enumerate the non-spurious input scenarios. This approach does not scale to the number of features and conditions of the APIs we are interested in. Works on CIT focus on how to select a "minimal", fixed set of input scenarios to cover as many feature interactions as possible. Some of these works (e.g. [CDS08]) also account for constraints between features to eliminate spurious scenarios. Their goal is to maximize the likelihood of exercising interesting behaviors of an arbitrary system, while minimizing the size of the test suite. Instead, we target a specific system and API and ask the developers to select which feature combinations need to be validated. The combinations to cover are chosen dynamically based on the portion of the system that is under active development.

Ridge et al. propose SibylFS [RST+15] for the verification of file systems. File system operations have only a few input arguments, while we deal with tens of input request parameters. In fact, 21k tests are sufficient for SibylFS, while the simplest of our individual APIs requires upwards of 100k tests to explore its behaviors. They also identify equivalence classes to obtain a finite set of scenarios to be tested. The test generator explores the combinations of all commands where the combinatorial testing is feasible and covers all "static real-world behaviors". The generated *test scripts* are then supplemented by manually defined ones to cover remaining cases. Instead, we are interested in conclusive configurable testing campaigns that can quantify which portion of the space of input scenarios has been covered.

8 Conclusions and Future Work

We described the approach we adopted to define a high-fidelity model for the S3 API. The same approach is used to increase the confidence in the correctness of both the model and the implementation itself. The tool is under active development to support new features and implementations. It has been instrumental in preventing deployment of over 300 regressions. The tool validates code deployments in the CI/CD pipelines of S3, S3 Express One Zone, and internal rewrites.

Our approach relies on manually defined predicates that abstract the set of requests and states into abstract scenarios. We are currently working on learning/mining these predicates from service interactions and from the source code to reduce the effort required to build and maintain the model.

Acknowledgments

We thank James Bornholt and Rajeev Joshi for valuable discussions and feedback on this work.

References

[AO17] Paul Ammann and Jeff Offutt. *Introduction to software testing*. Cambridge University Press, 2017.

[api] APIFuzzer — HTTP API testing framework. https://github.com/KissPeter/APIFuzzer. Accessed: 2025-11-16.

[Arc19] Andrea Arcuri. Evomaster: Evolutionary multi-context automated system test generation. *CoRR*, abs/1901.04472, 2019.

[BCD23] Bachir Bendrissou, Cristian Cadar, and Alastair F. Donaldson. Grammar mutation for testing input parsers (registered report). In *Proceedings of the 2nd International Fuzzing Workshop, FUZZING 2023, Seattle, WA, USA, 17 July 2023*, pages 3–11. ACM, 2023.

[BJA+21] James Bornholt, Rajeev Joshi, Vytautas Astrauskas, Brendan Cully, Bernhard Kragl, Seth Markle, Kyle Sauri, Drew Schleit, Grant Slatton, Serdar Tasiran, Jacob Van Geffen, and Andrew Warfield. Using lightweight formal methods to validate a key-value storage node in amazon S3. In *SOSP '21: ACM SIGOPS 28th Symposium on Operating Systems Principles, Virtual Event / Koblenz, Germany, October 26-29, 2021*, pages 836–850. ACM, 2021.

[CDS08] Myra B Cohen, Matthew B Dwyer, and Jiangfan Shi. Constructing interaction test suites for highly-configurable systems in the presence of constraints: A greedy approach. *IEEE Transactions on Software Engineering*, 34(5):633–650, 2008.

[CKHM25] Codel Cayden, Fazekas Katalin, Heule Marijn J. H., and Iser Markus. Proceedings of SAT competition 2025 : Solver and benchmark descriptions. 2025.

[CTT19] Alessandro Cimatti, Chun Tian, and Stefano Tonetta. NuRV: a nuXmv extension for runtime verification. In *Runtime Verification - 19th International Conference, RV 2019, Porto, Portugal, October 8-11, 2019, Proceedings*, volume 11757 of *Lecture Notes in Computer Science*, pages 382–392. Springer, 2019.

[CZPC21] Davide Corradini, Amedeo Zampieri, Michele Pasqua, and Mariano Ceccato. Empirical comparison of black-box test case generation tools for RESTful APIs. In *21st IEEE International Working Conference on Source Code Analysis and Manipulation, SCAM 2021, Luxembourg, Sept. 27-28, 2021*, pages 226–236. IEEE, 2021.

[CZPC22] Davide Corradini, Amedeo Zampieri, Michele Pasqua, and Mariano Ceccato. RestTestGen: An extensible framework for automated black-box testing of RESTful APIs. In *IEEE International Conference on Software Maintenance and Evolution, ICSME 2022, Limassol, Cyprus, October 3-7, 2022*, pages 504–508. IEEE, 2022.

[DMDS24] Swaroopa Dola, Rory McDaniel, Matthew B Dwyer, and Mary Lou Soffa. Cit4dnn: Generating diverse and rare inputs for neural networks using latent space combinatorial testing. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, pages 1–13, 2024.

[FFP+23] Simone Fratini, Patrick Fleith, Nicola Policella, Alberto Griggio, Stefano Tonetta, Srajan Goyal, Thi Thieu Hoa Le, Jacob Kimblad, Chun Tian, Konstantinos Kapellos, et al. Verification and validation of autonomous systems with embedded AI: The VIVAS approach. *ASTRA, ESA*, 2023.

[GGT25] Srajan Goyal, Alberto Griggio, and Stefano Tonetta. System-level simulation-based verification of autonomous driving systems with the VIVAS framework and CARLA simulator. *Sci. Comput. Program.*, 242:103253, 2025.

[GS97] Susanne Graf and Hassen Saïdi. Construction of abstract state graphs with PVS. In *Computer Aided Verification, 9th International Conference, CAV '97, Haifa, Israel, June 22-25, 1997, Proceedings*, volume 1254 of *Lecture Notes in Computer Science*, pages 72–83. Springer, 1997.

[GZA24] Amid Golmohammadi, Man Zhang, and Andrea Arcuri. Testing RESTful APIs: A survey. *ACM Trans. Softw. Eng. Methodol.*, 33(1):27:1–27:41, 2024.

[HD22] Zac Hatfield-Dodds and Dmitry Dygalo. Deriving semantics-aware fuzzers from web API schemas. In *44th IEEE/ACM International Conference on Software Engineering: Companion Proceedings, ICSE Companion 2022, Pittsburgh, PA, USA, May 22-24, 2022*, pages 345–346. ACM/IEEE, 2022.

[HKG18] Renáta Hodován, Ákos Kiss, and Tibor Gyimóthy. Grammarinator: a grammar-based open source fuzzer. In *Proceedings of the*

9th ACM SIGSOFT International Workshop on Automating TEST Case Design, Selection, and Evaluation, A-TEST@SIGSOFT FSE 2018, Lake Buena Vista, FL, USA, November 05, 2018, pages 45–48. ACM, 2018.

[HWKK20] Linghuan Hu, W Eric Wong, D Richard Kuhn, and Raghu N Kacker. How does combinatorial testing perform in the real world: an empirical study. *Empirical Software Engineering*, 25(4):2661–2693, 2020.

[KXSO22] Myeongsoo Kim, Qi Xin, Saurabh Sinha, and Alessandro Orso. Automated test generation for REST APIs: no time to rest yet. In *ISSTA '22: 31st ACM SIGSOFT International Symposium on Software Testing and Analysis, Virtual Event, South Korea, July 18 - 22, 2022*, pages 289–301. ACM, 2022.

[LW90] H.K.N. Leung and L. White. A study of integration testing and software regression at the integration level. In *Proceedings. Conference on Software Maintenance 1990*, pages 290–301, 1990.

[MASR21] Alberto Martin-Lopez, Andrea Arcuri, Sergio Segura, and Antonio Ruiz-Cortés. Black-box and white-box test case generation for RESTful APIs: Enemies or allies? In *32nd IEEE International Symposium on Software Reliability Engineering, ISSRE 2021, Wuhan, China, October 25-28, 2021*, pages 231–241. IEEE, 2021.

[McK98] William M. McKeeman. Differential testing for software. *Digit. Tech. J.*, 10(1):100–107, 1998.

[MLSRC21] Alberto Martin-Lopez, Sergio Segura, and Antonio Ruiz-Cortés. Restest: automated black-box testing of restful web apis. In *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 682–685, 2021.

[NMT12] Cu D Nguyen, Alessandro Marchetto, and Paolo Tonella. Combining model-based and combinatorial testing for effective test case generation. In *Proceedings of the 2012 International Symposium on Software Testing and Analysis*, pages 100–110, 2012.

[RFC22] HTTP RFC 9110. https://datatracker.ietf.org/doc/html/rfc9110, 2022. Accessed: 2026-06-10.

[RST+15] Tom Ridge, David Sheets, Thomas Tuerk, Andrea Giugliano, Anil Madhavapeddy, and Peter Sewell. Sibylfs: formal specification and oracle-based testing for POSIX and real-world file systems. In *Proceedings of the 25th Symposium on Operating Systems Principles, SOSP 2015, Monterey, CA, USA, October 4-7, 2015*, pages 38–53. ACM, 2015.

[S3D24a] Amazon S3: Creating buckets. https://docs.aws.amazon.com/AmazonS3/latest/userguide/creating-buckets-s3.html, 2024. Accessed: 2026-06-10.

[S3D24b] Amazon S3: GetObject. https://docs.aws.amazon.com/AmazonS3/latest/API/API_GetObject.html, 2024. Accessed: 2026-06-10.

[S3D24c] Amazon S3: REST API operations. https://docs.aws.amazon.com/AmazonS3/latest/API/API_Operations_Amazon_Simple_Storage_Service.html, 2024. Accessed: 2026-06-10.

[tca] Tcases: A model-based test case generator. https://github.com/Cornutum/tcases. Accessed: 2025-11-16.

[Tre08] Jan Tretmans. Model based testing with labelled transition systems. In *Formal Methods and Testing, An Outcome of the FORTEST Network, Revised Selected Papers*, volume 4949 of *Lecture Notes in Computer Science*, pages 1–38. Springer, 2008.

[UPL12] Mark Utting, Alexander Pretschner, and Bruno Legeard. A taxonomy of model-based testing approaches. *Softw. Test. Verification Reliab.*, 22(5):297–312, 2012.

[VDC20] Emanuele Viglianisi, Michael Dallago, and Mariano Ceccato. RestTestGen: Automated black-box testing of RESTful APIs. In *13th IEEE International Conference on Software Testing, Validation and Verification, ICST 2020, Porto, Portugal, October 24-28, 2020*, pages 142–152. IEEE, 2020.

[XPT+24] Chunqiu Steven Xia, Matteo Paltenghi, Jia Le Tian, Michael Pradel, and Lingming Zhang. Fuzz4all: Universal fuzzing with large language models. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering, ICSE 2024, Lisbon, Portugal, April 14-20, 2024*, pages 126:1–126:13. ACM, 2024.

[ZHM97] Hong Zhu, Patrick A. V. Hall, and John H. R. May. Software unit test coverage and adequacy. *ACM Comput. Surv.*, 29(4):366–427, December 1997.

M3U: Scalable Kernel Memory Management for Efficient Post-copy Live Migration of High-end Virtual Machines

Yizhe Xu[1*], Yuan Tao[2*], Zhibin Zhang[2*], Kang Yan[2], Chao Zhang[2◇], Shuo Shi[1], Zongpu Zhang[1], Xu Huan[1], Yibin Shen[2], Xudong Zheng[2], Jiesheng Wu[2], Jian Li[1◇], Haibing Guan[1]

[1]*Shanghai Jiao Tong University*, [2]*Alibaba Cloud Computing*

Abstract

High-end virtual machines (VMs) have become essential in public clouds for supporting large-scale guest applications. While conventional pre-copy live migration cannot reliably migrate such high-end VMs due to convergence problems, post-copy migration offers a viable alternative. However, high-end post-copy migration suffers from significant performance penalties, including extended downtime, prolonged post-copy duration and degraded guest performance, which stem from scalability bottlenecks in kernel memory management.

We identify the unnecessary overuse of lock protection as the root cause of scalability bottlenecks in existing approaches, which can be effectively resolved through strategic lock relaxation. To address this challenge, we propose M3U, a scalable kernel memory management approach for post-copy live migration of high-end VMs. M3U customizes lock-protected memory management operations, reducing operation overhead and minimizing critical sections. Additionally, M3U employs lock-reduced parallelism to decrease dirty page registration overhead. M3U also implements a decoupled fault handling pipeline to maximize page transfer efficiency, and utilizes fault-aware page size determination to meet minimum fault latency requirements. For pass-through devices, M3U further provides proactive identification and pre-transmission of device states, effectively eliminating 98.5% of hardware I/O page faults. Our evaluation demonstrates that M3U achieves a 47.0% reduction in downtime, an 89.6% reduction in post-copy duration, and a 4.1× improvement in guest performance.

1 Introduction

Today's public cloud infrastructure faces growing demand for deploying large-scale, resource-intensive applications—such as cloud gaming, deep learning, and large language model serving—driving the proliferation of high-end virtual machines (VMs). In this paper, we define high-end VMs as instances with substantially aggregated resources: at least 64 vCPUs, 256 GB of memory, up to 100 Gbps network bandwidth, and 600k IOPS disk performance. According to Google Cloud Platform's 2025 documentation [9], 85.7% of currently available machine types already support high-end VM configurations.

Live migration is an essential feature for cloud management, enabling transparent and seamless relocation of VMs from one physical server to another. This capability is particularly valuable for routine operations such as hardware maintenance and dynamic workload rebalancing [2, 7, 25, 28, 39, 51]. However, the increasing adoption of high-end VMs introduces significant challenges to live migration processes, as migrating these resource-intensive, high-end VMs without service interruption requires enhanced technical approaches.

Pre-copy [8, 26, 37], the predominant live migration approach in public clouds, is prone to failure when migrating high-end VMs due to the convergence problem [29, 38, 49]. The convergence problem occurs when vCPUs write dirty pages to VM memory at a rate exceeding the network transfer speed, preventing pre-copy from satisfying its stop condition. If the convergence problem persists, pre-copy live migration will fail. In fact, this problem is particularly prevalent among high-end VMs. Our analysis of over 50,000 pre-copy live migration samples from high-end VMs collected over 12 months in our cloud infrastructure reveals a migration success rate of merely 81%, with the convergence problem identified as the root cause of migration failures.

To ensure successful live migration of high-end VMs, employing a post-copy scheme [22] is essential. Post-copy addresses the convergence problem inherent in pre-copy by allowing the VM to resume execution on the target host before all memory has been transferred. The remaining dirty pages are then synchronized gradually in the background between the source and target hosts. If the VM accesses a page that has not yet been transferred, a page fault is triggered, causing the VM to block until that page arrives. Importantly, because the source VM is suspended during this process, the dirty memory set ceases to grow, which ensures that the migration

[*] Yizhe Xu, Yuan Tao and Zhibin Zhang equally contributed to this work.
[◇] Jian Li and Chao Zhang are the corresponding authors.

process will eventually converge and complete successfully.

However, post-copy live migration incurs two significant performance penalties when applied to high-end VMs—penalties stemming from their extreme resource density (e.g., large memory footprints, numerous vCPUs) and workload intensity (e.g., high memory-dirtying rates, bursty I/O). As VM resource allocations continue to scale, high-end VMs increasingly expose fundamental scalability limitations in the post-copy model. These limitations primarily manifest in two ways. **(1) Unacceptable VM switching downtime.** The post-copy model requires marking dirty memory as missing before the VM resumes on the target system. Consequently, the target system unmaps all dirty memory in page-sized units during downtime—a process termed *dirty page registration*. This operation accounts for 57–66% of the total downtime and results in excessive downtime lasting multiple seconds (§2.2). **(2) Heavy performance degradation during prolonged post-copy duration.** In post-copy migration, missing memory pages are fetched remotely and reconstructed on the target system. However, this reconstruction process introduces significant lock contention, as multiple tasks concurrently attempt to maintain cross-page-table consistency. This lock contention reduces page transfer throughput to only 9.2% of the available network bandwidth. Consequently, both per-page-fault latency and post-copy duration are substantially prolonged, leading to severe guest performance degradation—e.g., network interruption lasting over 15 seconds (§2.2).

Few prior studies have examined the performance penalties associated with post-copy live migration of high-end VMs. Recent work on Two-Dimensional Paging (TDP) Memory Management Unit (MMU) [18] attributes this challenge to scalability bottlenecks within the kernel MMU. In this paper, we define MMU as the software memory management subsystem provided by the host kernel and hypervisor, rather than the conventional hardware unit for address translation. Specifically, TDP MMU reveals severe lock contention arising from exclusive locking of the entire kernel MMU during each page fault in post-copy migration. However, as we demonstrate in §2.2, TDP MMU addresses only a limited subset of kernel MMU scalability bottlenecks and provides minimal improvements in post-copy performance.

In this paper, we present a critical observation: lock protection in standardized kernel MMU capabilities is *unnecessarily overused* in the context of post-copy live migration. This overuse of lock protection introduces scalability bottlenecks across page table management, consistency maintenance and fault handling, which collectively result in significant performance penalties in high-end VM scenarios. By strategically applying lock relaxation techniques to kernel MMU capabilities, we can minimize the critical sections protected by locks, thereby maximizing system parallelism and improving the overall efficiency of post-copy migration.

We propose Migration Memory Management Unit (M3U), a novel kernel MMU design that addresses the scalability challenges inherent in high-end post-copy migration. M3U manages VM memory and interacts with the original kernel MMU through standard system abstractions while implementing its own page-fault handler and page table management logic (e.g., page mapping/unmapping). To achieve this, M3U employs three key optimizations: **(1) First,** M3U fully preallocates VM physical memory, and maintains it statically by employing lightweight permission-bit flagging to replace costly page mapping/unmapping operations. This approach reduces critical sections and operation overhead, thereby enabling efficient parallel dirty page registration to minimize VM switching downtime. **(2) Second,** M3U implements a decoupled userfault pipeline to achieve lock-free parallel page data copying and mixed page table updating. These mechanisms collectively minimize fault latency and reduce the duration of the post-copy phase. **(3) Third,** to mitigate performance degradation caused by I/O page fault handling for pass-through devices, M3U eliminates nearly all I/O page faults through device state identification and pre-transmission.

In summary, our contributions in this paper are as follows.

- We provide detailed analyses and empirical observations on the performance penalties and kernel MMU scalability bottlenecks inherent in high-end post-copy migration.
- We introduce M3U, a scalable kernel MMU approach designed to address these scalability bottlenecks and mitigate the performance penalties associated with high-end post-copy migration.
- We present a comprehensive evaluation of M3U, demonstrating significant performance improvements over existing approaches: reducing downtime by up to 47.0%, decreasing post-copy duration by 89.6%, and improving guest service performance by 2.6–4.1×.

2 Background and Motivation

2.1 High-end Post-copy and Kernel MMU

High-end post-copy. The post-copy scheme [22] represents an appropriate choice for live migration of high-end VMs, as it effectively avoids the convergence problem inherent in pre-copy migration. Post-copy migration generally comprises two distinct phases.

- **VM switching phase.** During this phase, the VM is first suspended on the source host, and the target system restores the VM's state information (including CPU and device states) from the source. Second, *dirty page registration* marks the remaining dirty pages on the source VM as not present for the target VM. Finally, once the target system completes the restoration of all VM states, the VM is resumed on the target host. The VM downtime is incurred during this phase, as the VM remains suspended throughout the switching process.
- **Post-copy phase.** Following the VM switching phase, the remaining dirty pages are transferred to the target system

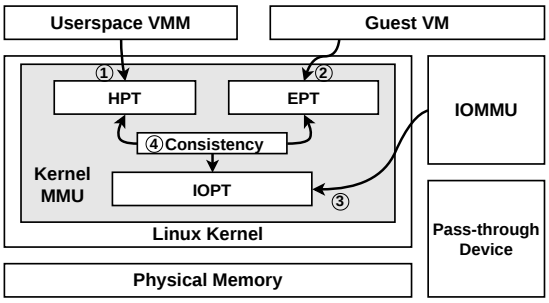


Figure 1: Kernel MMU capabilities for virtualization.

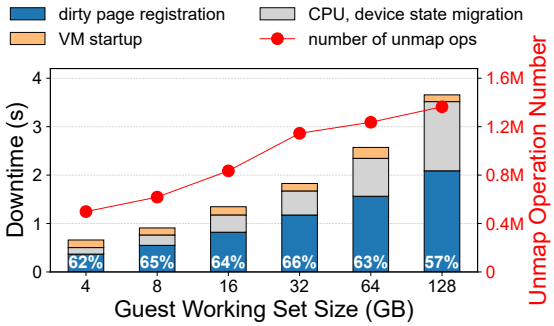


Figure 2: Downtime breakdown collected on the target during post-copy live migration of a VM with 64 vCPUs, 256 GB memory. (workload: memory random write, block size 1 KB)

through two complementary mechanisms: *demand paging* and *active pushing* [22]. Demand paging operates by immediately pulling a page from the source to the target when the target VM accesses a registered page and triggers a page fault. Concurrently, active pushing maintains a continuous background transfer of dirty pages from the source, ensuring that pages are available before they are accessed by the target VM.

Three critical migration metrics—*downtime* (the duration of VM switching), *post-copy completion time* (PCT; that is, the duration of the post-copy phase), and *guest performance degradation*—are commonly used to evaluate both performance and guest impact during post-copy live migration. For convenience, we hereafter refer to post-copy live migration of high-end VMs as *high-end post-copy*.

Kernel MMU. The core operation performed during VM migration involves identifying dirty memory pages, transferring them, and integrating them into the target VM's address space. These tasks are typically implemented using a suite of functions provided by the kernel Memory Management Unit (MMU).

As shown in Figure 1, the kernel MMU supports post-copy live migration through the following key capabilities:

① *guest memory manageability* through the Host Page Table (HPT), which enables the userspace Virtual Machine Monitor (VMM), such as QEMU [41], to allocate, deallocate, map, and unmap VM physical memory;

② *guest memory virtualization* through the Extended Page Table (EPT), which facilitates VM execution by managing Host Physical Address (HPA) to Guest Physical Address (GPA) mapping and handling vCPU page faults;

③ *Direct Memory Access (DMA) support* through the I/O Page Table (IOPT), which enables memory virtualization for pass-through devices;

④ *cross-table consistency*, which coordinates updates across the HPT, EPT, and IOPT to ensure mapping coherence, for instance, reallocating a guest physical memory page triggers synchronized updates to all corresponding page table entries.

2.2 High-end Post-copy Challenges MMU Scalability

However, these kernel MMU capabilities confront significant scalability challenges for high-end VM post-copy, which lead to unacceptable performance penalties in migration metrics.

In this section, we provide both theoretical and experimental analyses on these performance penalties. The test configuration is identical to that in §6.1. To improve address translation efficiency and overall guest performance, we utilize huge pages (hereafter referring to 2 MB pages) to manage guest memory by default. We support 1 GB huge pages but do not use them as the default option because 1 GB huge page requires large, contiguous physical memory regions, which is less practical in our deployment scenarios. The results of migration metrics are shown in Figures 2 and 3.

Unacceptable VM switching downtime. As shown in Figure 2, the migration downtime increases rapidly with the workload intensity and exceeds 3 seconds (*s*) when we increase the working set size of memory random write workload to 128 GB in the high-end VM being migrated. In our cloud, downtime at such a level counts against most of our committed Service Level Objectives (SLOs) with customers, which can lead to immediate customer complaints when their real-time workloads are interrupted.

We found that the overhead of dirty page registration accounts for 57–66% of the total downtime as a major contributor. This overhead comes from the costly unmap operations to register and mark every dirty page in guest memory as missing on the target system. As a userspace process, the migration thread cannot directly modify its own page tables. Consequently, it must rely on the kernel-provided unmap interface, which simultaneously releases physical memory and updates the HPT, as the only mechanism for HPT management (① in Figure 1). However, the unmap operation is lock-protected and mutually-exclusive in execution, which leads to scalability bottlenecks. As illustrated in Figure 2, the total cost of unmap operations scales almost linearly with total operation number. While it is acceptable for normal VMs, for high-end VMs with an extremely large quantity of unmap

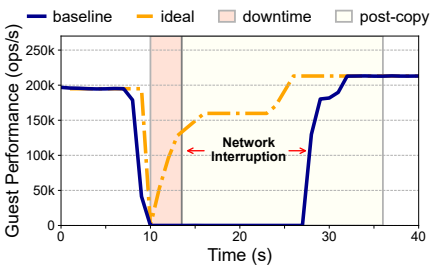


Figure 3: Guest service undergoes network interruption for over 15 *s*. (Redis, 1 thread, 24 GB working set, random SET)

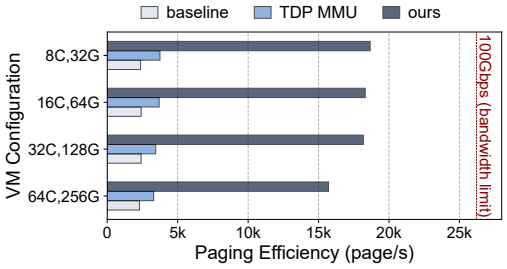


Figure 4: Paging efficiency comparison (quantified in pages per second from source to target; page size is 4 KB).

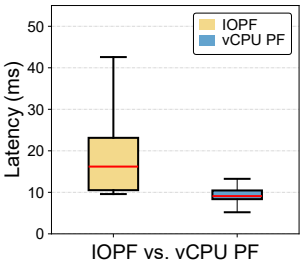


Figure 5: Demand paging latency comparison.

operations required, the overhead of dirty page registration becomes non-trivial. For instance, a VM with a 128 GB working set can trigger up to 32 million unmap operations to be done before the VM can be resumed on the target. Note that these unmapped pages must be reallocated when they are later recovered—either via active pushing or demand paging.

Heavy performance degradation and long PCT. Ideally, as shown in Figure 3, we expect post-copy live migration to bear almost no downtime and the guest to suffer minimum performance before all its remaining memory has been transferred. Contrary to our expectation, in the test shown in Figure 3, guest service experiences a network interruption for over 15 *s* even after the VM has resumed on the target host. Moreover, the PCT, defined as the duration for the VM to recover to its peak throughput after migration, exhibits significant divergence from the theoretical memory-transfer bound. As shown in Figure 3, the observed PCT is 24 *s*, whereas transferring the remaining 24 GB of memory over a 50 Gbps bandwidth (half of the host available bandwidth, explained in §6.1) would take only 3.84 *s* under ideal conditions. This results in a significantly longer effective downtime compared to that of pre-copy.

We observe that heavy guest degradation and long PCT are both caused by the low efficiency of demand paging and active pushing in dealing with the dirty pages and guest page faults at high-end scale. We show in Figure 4 that the baseline efficiency of demand paging and active pushing merely achieves 9.2% utilization of the total physical network bandwidth. With further analysis, we discover that such low efficiency is primarily caused by two conflicts between demand paging and active pushing.

Lock contention for cross-table consistency. Post-copy migration leverages the Linux userfault mechanism to handle the transferred page data for demand paging and active pushing. To safely integrate these newly arrived pages into the target's guest memory, the userfault handler must ensure cross-table consistency among the HPT, EPT, and IOPT (④ in Figure 1). The cross-table consistency is achieved in the following exclusively locked steps: a new host physical page is allocated, its content is copied from userspace to kernel, and all the three page tables (HPT, EPT, and IOPT) are updated atomically under their memory locks. These locks raise significant contention among demand paging, active pushing and vCPU page fault handling. Consequently, the userfault-driven page-in path becomes a scalability bottleneck in high-end post-copy scenarios, where a large quantity of vCPUs may concurrently fault on missing pages—leading to severe lock contention and serialized page-in operations.

Page size dilemma. In contrast to demand paging, which favors the smallest page size (e.g., 4 KB) to minimize per-fault latency due to the data movement through the Ethernet, active pushing operates as a background, bulk-transfer mechanism and thus benefits significantly from larger memory chunks. To maximize throughput and reduce overhead, we advocate using large pages (e.g., 2 MB) for actively pushed regions. This reduces the frequency of userfault invocations, page-copy operations, and page-table installations—collectively lowering control-plane overhead and improving overall paging efficiency. However, memory virtualization (e.g., by QEMU/KVM) provided by the kernel MMU enforces a *single, fixed page size* to manage the entire guest address space across the HPT, EPT and IOPT. The single page size enforcement raises conflicts between low per-fault latency and high paging efficiency required by demand paging and active pushing respectively.

A recent work, TDP MMU [18], partially resolves the lock contention in cross-table consistency by enabling parallelized EPT update. It replaces the original spinlock on the entire kernel MMU with a shared read lock, and uses atomic instructions to guarantee synchronous changes on EPT entries. However, TDP MMU cannot resolve the lock contention in the remaining operations of the userfault handler, e.g., the updating of the HPT and IOPT. In addition, TDP MMU does not provide a solution to the page size dilemma we mentioned above. Consequently, TDP MMU gains very limited improvement in kernel MMU scalability and paging efficiency as shown in Figure 4.

2.3 I/O Page Fault Performance

To meet high-end VMs' demands for high-performance networks, storage, and heterogeneous accelerators, cloud service providers adopt pass-through as a primary I/O virtualization mechanism. This approach enables guest VMs to communicate directly with devices using DMA (③ in Figure 1), bypassing the CPU and host software to achieve bare-metal level I/O performance. During the post-copy phase, DMA operations on non-present pages trigger hardware I/O page faults (IOPFs). To investigate IOPF performance, we constructed a real IOPF-enabled platform based on an existing IOPF solution [60] (also used in [21, 59]) that leverages PCI hardware support [40].

However, demand paging triggered by IOPF imposes a greater impact on guest performance than page faults triggered by vCPUs. Figure 5 shows that demand paging between these two types of page faults incurs approximately 2× average latency differences (3.21× for maximum). This performance degradation occurs because IOPF undergoes a PCIe transaction and a host interrupt before being handled by the host CPU, resulting in significantly higher latency than normal page faults (e.g., 3–80×) [56, 57]. Furthermore, IOPF handling occurs within the critical path of device DMA. Consequently, the device must suspend processing of the faulted I/O queue until the IOPF is resolved. This suspension can further lead to queue blocking and packet loss, particularly under bursty I/O workloads common in modern NICs and storage controllers. As a result, a single unresolved IOPF may stall the entire I/O queue, inducing delays that extend beyond the IOPF latency alone and directly impact guest service quality [21, 24, 59].

Our further investigation reveals that the IOPF pattern in post-copy migration exhibits a distinct temporal and structural pattern. Almost all IOPFs occur within the first few seconds after the VM resumes on the target host. These IOPFs fall into two categories: 1) Descriptor faults, triggered when the device driver accesses I/O descriptors (e.g., VirtIO virtqueue headers/descriptors) that record buffer metadata (address, length); 2) Buffer faults, caused by missing I/O data buffers. Crucially, I/O descriptors reside in driver-initialized, fixed-location control structures (e.g., virtqueue vrings) and are reused circularly during I/O processing—meaning each descriptor page faults at most once per migration. In contrast, I/O buffers are dynamically allocated by the driver per request; however, since the driver always accesses them first via CPU (e.g., to populate headers or issue commands), buffer pages are typically recovered via regular vCPU page faults—not IOPFs. As a result, descriptor faults dominate the IOPF footprint, while buffer-induced IOPFs are rare in practice.

2.4 Challenges and Insights

Kernel MMU scalability challenges. The performance analysis reveals that high-end post-copy migration introduces significant scalability challenges to the kernel MMU. These challenges stem from three primary sources:

- **HPT manageability** (① in Figure 1): lock-serialized updates to the HPT create a bottleneck, resulting in prolonged downtime.
- **Cross-table consistency maintenance** (④ in Figure 1): lock-serialized userfault handlers enforce strict synchronization requirements, which causes substantial guest performance degradation and extends PCT.
- **IOPT fault handling** (③ in Figure 1): hardware constraints in IOPF handling mechanisms limit fault processing latency, further degrading guest performance.

Key insight. The kernel MMU inherently enforces strict lock protection during HPT updates and cross-table consistency maintenance to ensure data correctness and consistency. However, we observe that *lock protection is unnecessarily overused* in numerous operations during post-copy migration (which will be elaborated in the next paragraph). Consequently, it is feasible to strategically relax lock protection in these operations to achieve better kernel MMU scalability in high-end post-copy scenarios.

The overused lock protection is primarily concentrated in three areas: physical memory allocation and deallocation, page content and table updating. **(1) First,** physical memory allocation and deallocation are essential to native kernel MMU capabilities such as map and unmap operations; however, they are entirely unnecessary for post-copy migration. We can therefore statically maintain the placement of VM physical memory during post-copy to eliminate both physical memory allocation/deallocation and the corresponding lock-protection overhead. **(2) Second,** page content updating is tightly coupled with the userfault handler through lock protection. By decoupling page data copy from the userfault handler, we can effectively eliminate this lock protection and achieve higher copy efficiency through parallel execution.

Based on these insights, we address the aforementioned challenges by designing strategic lock relaxation in kernel MMU capabilities, which comprises three components: (1) lock-reduced parallelism for HPT updates to achieve scalable HPT manageability; (2) decoupled and pipelined userfault handling to minimize lock contention during cross-table consistency maintenance; and (3) minimized IOPF occurrence (reduced to nearly zero IOPFs) rather than laboriously mitigating IOPT fault handling performance.

3 M3U Overview

We propose the Migration Memory Management Unit (M3U), a novel kernel MMU design for high-end post-copy to tackle the scalability challenges. As shown in Figure 6, M3U is a modular plugin that interacts with the original kernel MMU through standard system MMU operation abstractions. Therefore, M3U does not include invasive changes to other VMM or kernel MMU functionalities. M3U preserves the memory

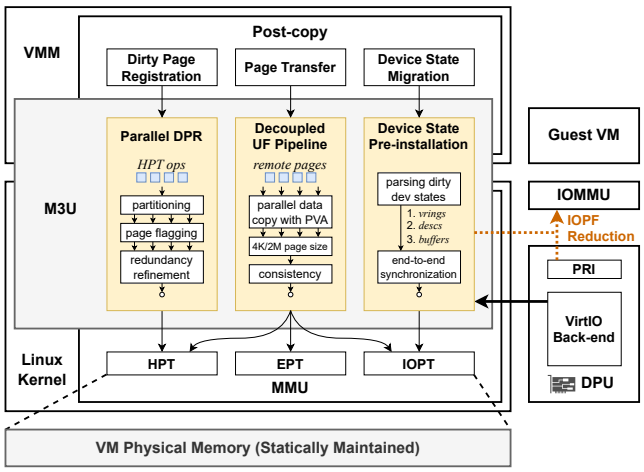


Figure 6: M3U overview.

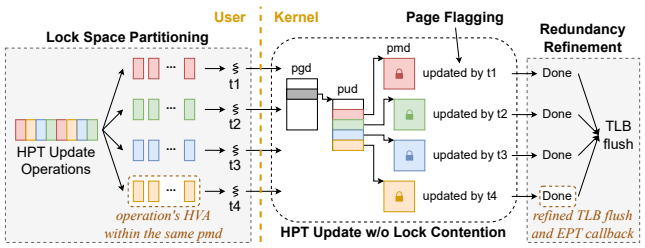


Figure 7: Lock-reduced parallelism for dirty page registration.

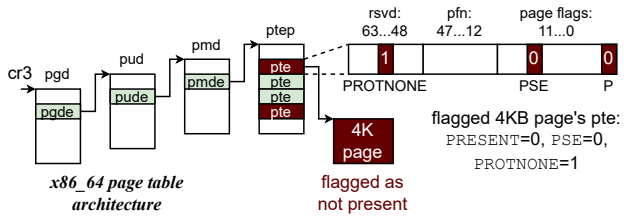


Figure 8: Illustration of page flagging.

mappings across the HPT, EPT and IOPT unchanged throughout post-copy migration, thereby significantly minimizing MMU modification overhead. As a result, MMU operations become lightweight, and their lock contention is drastically reduced. Specifically, M3U incorporates the following design components.

Parallel Dirty Page Registration. To minimize VM switching downtime, we introduce a lightweight, parallel dirty-page registration mechanism. By leveraging two key properties: (1) physical memory remains allocated throughout post-copy migration, and (2) the guest is fully suspended during VM switching, we replace heavyweight HPT operations (e.g., page mapping/unmapping) with ultra-lightweight permission flag updates, such as toggling the Present bits in HPT entries. This eliminates redundant MMU operations that incur high synchronization overhead, including physical page allocation/deallocation and cross-CPU Translation Look-aside Buffer (TLB) shootdowns, thereby dramatically reducing lock contention and enabling highly parallel memory discarding.

Decoupled Userfault Pipeline. To mitigate performance degradation and reduce PCT during VM resumption, we decouple the underlying address dependencies between active pushing and demand paging. This separation simultaneously alleviates lock contention for cross-table consistency and resolves the page-size trade-off inherent to these two paths. Specifically, we assign distinct Host Virtual Address (HVA) spaces for active pushing and demand paging. This design eliminates costly user/kernel memory copies and enables batched, pipelined HPT updates, thereby dramatically reducing lock contention. Moreover, this address space decoupling between active pushing and demand paging allows us to support mixed page sizes: large pages (e.g., 2 MB) for active pushing to maximize bandwidth efficiency, and 4 KB pages for demand paging to minimize fault latency, optimizing each path independently.

Device State Pre-installation. Rather than laboriously optimizing the inherently complex and latency-sensitive IOPF handling path, we eliminate nearly all IOPFs with simple device state pre-installation. Since descriptor faults dominate IOPFs in post-copy migration, and because I/O descriptors along with their associated I/O buffers occupy only a small memory footprint, we proactively transmit this device state to the target before VM resumption. As a result, DMA operations during the critical first few seconds after resumption proceed without triggering any actual IOPFs.

4 M3U Detailed Design

In this section, we present the M3U design details that address the scalability challenges and achieve efficient post-copy live migration for high-end VMs.

4.1 Parallel Dirty Page Registration

To improve the throughput of dirty page registration during the critical VM switching window, we propose parallel dirty page registration. The key to enabling efficient parallelism is eliminating or simplifying redundant MMU operations that cause lock contention.

Our approach rests on three core design principles: (1) Replace heavyweight HPT update operations with lightweight permission-bit flagging (e.g., clearing the Present/Writable bits), thereby keeping physical memory allocations static throughout the process; (2) Eliminate redundant MMU operations—such as page allocation/deallocation and TLB maintenance—that incur significant locking overhead; (3) Partition the VM's HPT address space to allow concurrent dirty page registration across disjoint regions, thereby avoiding inter-thread lock contention. We illustrate this design in Figure 7 and explain the details as follows.

Page Flagging. We propose page flagging (illustrated in Figure 8) to replace the heavy HPT updates (e.g., page mapping or unmapping), to make the dirty page registration

more lightweight. Page flagging refers to replacing physical memory deallocation during dirty-page registration with a lightweight 2 MB page-table splitting operation, followed by marking all 512 resulting 4 KB page-table entries with the non-present bit. Once flagged as non-present, a page becomes inaccessible to the guest VM; any subsequent VM access triggers a standard page fault, which notifies the migration process to fetch the page from the source host. Meanwhile, the discard process creates an HVA address space dirty bitmap [15] to record the absence of memory pages, with each bit representing a present/non-present 4 KB memory page.

M3U also leverages this page-flagging mechanism to preserve physical memory allocations throughout the migration procedure. This stability allows M3U to exploit a key property: the VM's physical memory (in HPTs) is fully allocated during the pre-copy phase. Consequently, when pages are later recovered via demand paging, their mappings can be restored efficiently—without requiring new physical allocations or complex remapping logic.

Redundancy Refinement. As VM stops running during VM switching and physical memory allocation remains, we can further refine two redundant locks in the HPT update to reduce lock contention: (1) the lock-intensive MMU notifier callback for erasing invalidated EPT mappings, which is unnecessary because the EPT is not constructed until VM resumption; (2) the TLB flush for every unmap operation. Specifically, since EPT is not fully constructed until userfault handling, the EPT callback in MMU notifier can be safely removed from the unmap operation without affecting system behavior. Meanwhile, VM does not access guest memory in downtime, so that we can aggregate all TLB flushes into one flush when dirty page registration is completed. To this end, the refined HPT update operation in M3U fully eliminates contention caused by redundant locks.

Lock Space Partitioning. To eliminate lock contention in parallelized HPT page flagging, we partition HPT update operations by address space. By replacing complex memory unmapping with lightweight page flagging, the only locks required during flagging are the page-table locks (e.g., PGD, PUD, and PMD in the x86_64 page table hierarchy). Other locks, such as the VMA lock, are no longer needed, since the underlying HPT mappings of HVA-to-HPA remain unchanged. As shown in Figure 7, M3U partitions the entire HVA space into disjoint 1 GB regions. Consequently, the PUD and PMD locks required to update the 512 PMD entries within each region are independent across regions.

M3U employs a worker thread pool to parallelize dirty page registration. The page-grouping worker receives dirty pages from the source system, which are transmitted in strictly increasing order of HVAs. Leveraging this ordering, the worker groups all pages that fall within the same 1 GB address boundary and assigns each group to a dedicated discarding worker. Because HPT updates within a single 1 GB region require only the corresponding PMD and PUD locks, discarding workers operating on disjoint 1 GB address ranges do not contend for the same locks, enabling effectively lock-contention-free parallelism. When all the VM pages are discarded, the page-grouping worker triggers a TLB flush to update the physical TLB buffer.

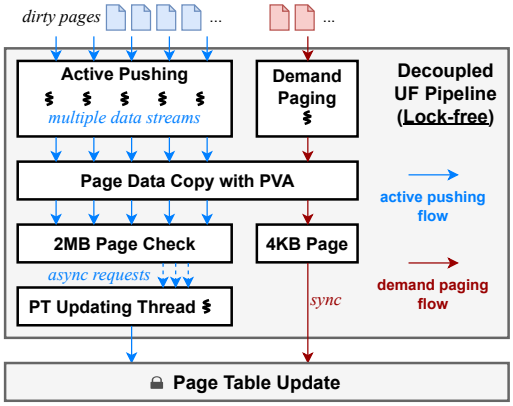


Figure 9: Decoupled userfault pipeline.

4.2 Decoupled Userfault Pipeline

To mitigate performance degradation and reduce PCT during VM resumption, we separate the address space to reduce contention and enable mixed page granularity to speed up active pushing and demand paging. Figure 9 depicts the overall design of decoupled userfault pipeline.

4.2.1 Separate Address Space

To improve the efficiency of active pushing, M3U builds an exclusive post-copy virtual address (PVA) space for statically maintained physical memory to straightforwardly recover dirty pages by direct memory copying. This direct mapping allows active pushing to recover multiple dirty pages concurrently across multiple data streams to enable high-throughput, parallel page restoration.

Post-copy Virtual Address (PVA). M3U incorporates a PVA mapping interface to directly remap the VM memory in another address space distinct from the origin HVA. With PVA, active pushing breaks the original dirty-page recovery procedure into two stages: memory data copying and cross-table consistency maintenance. First, when the active pushing on the target fetches a remote page, it directly copies the page data to guest memory with PVA. This eliminates the original costly user/kernel memory copies and lock holding during updates to the page table and page content in kernel space. Second, the active pushing clears the corresponding bit in the dirty bitmap set by page flagging and submits an asynchronous update request to a dedicated page-table-updating thread to perform cross-table consistency operations, such as synchronizing HPT, EPT, and IOPT entries. As these operations are handled exclusively by a dedicated single thread, lock contention among data stream threads is eliminated.

Multi-pushing. By leveraging an isolated PVA address space that enables concurrent memory copying, M3U integrates parallel data streams into active pushing to allow simultaneous page transfers—significantly improving paging efficiency. Each data stream is established over a dedicated socket connection and managed by migration threads on both the source and target sides. The source side dispatches dirty pages across multiple data streams for parallel transmission, while the target side concurrently receives and restores the contents of these pages. Meanwhile, a dedicated page-table-updating thread asynchronously maintains the cross-table consistency (e.g., across the HPT, EPT, and IOPT) required by all data stream threads, thereby eliminating lock contention among them. Through this design, M3U achieves highly efficient page transfer via batched, pipelined active pushing.

For demand paging, which is extremely latency-sensitive, M3U employs a dedicated data stream with a 20 Gbps throughput upper bound to handle page-fault requests, ensuring they are not blocked behind active-pushing traffic.

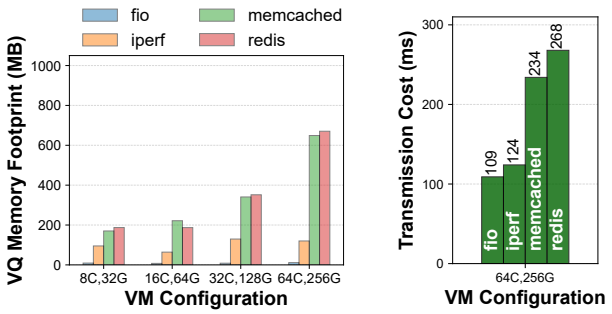
4.2.2 Mixed Page Table Updating

The decoupling of PVA from HVA simplifies the cross-table consistency maintenance so that M3U is capable of supporting mixed page sizes to balance the trade-off between active pushing and demand paging.

Demand Paging in 4 KB Granularity. M3U enforces a 4 KB page size for demand paging to minimize page-fault handling latency. As described in Section 4.1, any 2 MB HPT entry containing missing pages is split into 512 individual 4 KB page-table entries, each initially marked as non-present. When the VM accesses a missing 4 KB page, the demand-paging handler immediately fetches the page from the source and clears the corresponding bit in the dirty bitmap set by page flagging. The vCPU fault handler then sets the Present bit in the corresponding 4 KB HPT entry and installs the matching EPT entry, allowing the VM to resume execution seamlessly. By using 4 KB pages for demand paging, M3U avoids unnecessary overfetching of dirty memory and minimizes fault latency, ensuring rapid recovery of only the required data.

Active Pushing in 2 MB Granularity. M3U provides a batched HPT page-table rebuild interface that enables active pushing to recover discarded (missing) pages at 2 MB granularity. When active pushing finishes copying data and submits an asynchronous update request to the page-table-updating thread, the thread checks the dirty bitmap to determine whether all memory within the corresponding 2 MB region has been successfully retrieved from the source to the target. This 2 MB page check is performed using atomic bitmap operations and does not necessitate lock protection. Once all 512 constituent 4 KB pages of a 2 MB region are confirmed as recovered, the thread coalesces them into a single 2 MB HPT entry, replacing the individual 4 KB mappings. This optimization reduces what would otherwise require 512 separate page-table updates to just one, thereby dramatically decreasing the number of locks acquired (e.g., PMD/PUD locks) and significantly alleviating lock contention during HPT maintenance. In addition, M3U eliminates redundant transfer of 4 KB pages with an existing send-once guarding mechanism in QEMU—during post-copy, the source leverages its dirty bitmap to ensure that each page is transferred at most once [43]—thereby preventing transfer amplification.

Notably, M3U eliminates all intermediate page copies that the original QEMU must rely on when guest memory uses 2 MB page size. For the origin QEMU on the target system, it has to wait for the 512 × 4 KB pages to be transferred before it finally maps this 2 MB page. However, there is no guarantee that all 4 KB page data within a 2 MB page can be transferred strictly in order and without any loss. Consequently, each 2 MB page uses a temporary buffer to store the 4 KB page data until the 2 MB page is completely transferred, which causes an additional page copy through this temporary buffer. With PVA and mixed page size support, M3U no longer needs such intermediate page copies.



(a) Virtqueue Memory Footprint (b) Transmission Cost

Figure 10: Small total memory footprint and lightweight transmission cost of virtqueue structures, making one-time transmission of these virtqueues a feasible alternative to demand paging by IOPF handling.

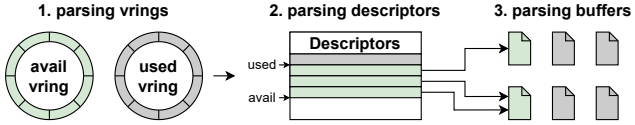


Figure 11: Parsing the VirtIO virtqueue structures.

4.3 Device State Pre-installation

Device state pre-installation eliminates nearly all IOPFs during post-copy. A critical observation on our platform is that the device states, e.g., the virtqueue structures for the VirtIO driver, remain in a small total memory footprint, regardless of the I/O pressure. Figure 10a shows various workloads (including intensive workloads for network, storage, CPU and memory) we tested for diverse I/O pressures, and the total

memory footprint of virtqueues is at most ∼671 MB. Directly transferring these virtqueues incurs very lightweight overhead considering the network bandwidth assigned for live migration. Generally, for production-level live migration practices, we assign 20 Gbps for pre-copy and 100 Gbps (the host available bandwidth, explained in §6.1) for transmission during downtime. With such network bandwidth, the transmission cost of virtqueues achieves no more than 300 milliseconds (*ms*) for the tested workloads as illustrated in Figure 10b.

As a result, transmitting all device state in a single batch is significantly more efficient than fetching it on demand via IOPFs. When all DMA-related missing pages are recovered during VM switching, the I/O buffers already present in the virtqueue become non-faulting. Subsequent I/O buffers produced after VM resumption will typically trigger a vCPU page fault, which retrieves the required memory from the source system on demand.

End-to-end Synchronization of Dirty Device States. M3U implements end-to-end functionality for capturing, parsing, exporting, and importing all dirty virtqueue structures by leveraging the VirtIO backend driver. As shown in Figure 11, M3U begins by locating the vring metadata and descriptor tables within each virtqueue. The physical addresses of these structures are recorded by QEMU during device initialization. M3U then traverses each descriptor table between the available and used ring indices to identify all active I/O buffers associated with pending or in-flight requests. The resulting dirty virtqueue state—including vrings, descriptors, and referenced buffers—is exported to the migration thread, which handles its transmission over the network to the target host. Upon receipt, the target VMM uses M3U to reconstruct and install the complete virtqueue state, completing end-to-end synchronization of dirty device state. Critically, this entire process is fully transparent to the guest OS and requires no modifications to the guest—preserving compatibility while enabling efficient post-copy device resumption.

The concrete procedure of device state migration is now as follows. After the VM and the devices are suspended on the source, the VMM performs a device draining operation that flushes all in-flight packets in devices and makes the device states consistent. During the device state migration in VM switching, the VMM leverages M3U to achieve this synchronization of device states between the source and target.

4.4 Concurrency Safety

M3U ensures concurrency safety throughout post-copy migration in two phases: (1) Externally across the source and target hosts. To perform critical VM switching, the source suspends the vCPUs and drains the devices during the downtime phase, where no concurrent VM execution is allowed, thereby ensuring data consistency. After the VM resumes on the target, non-present pages remain inaccessible to the VM until they are transferred, preventing data corruption. (2) Internally on the target host. Technically, the only new race introduced by M3U is the concurrent update of the same page by both demand paging and active pushing. To eliminate this race, the source leverages its dirty bitmap [43] to ensure that each page is transferred at most once; the target uses its received bitmap to ensure that each page and the corresponding mapping are restored at most once. This coordination mechanism across both migration hosts ensures such races never occur during post-copy, effectively preventing page data from being overwritten or corrupted. As for other inherent race conditions—such as those between concurrent faults and demand-paging requests—the hypervisor and VMM already handle them safely, eliminating any risk of data corruption.

5 Implementation

We implemented a prototype of M3U in QEMU 8.2 and Alios with Linux kernel 4.19. Our prototype supports all VirtIO driver versions; we use version 0.95 in our evaluation. The M3U prototype implementation consists of over 2,000 lines of code (LoC) in userspace and over 4,000 LoC in the kernel.

Applicability. M3U's design is applicable to both Type I and Type II hypervisors. The implementation follows the deployment setting adopted in our cloud environment, which utilizes the QEMU-KVM virtualization stack under a Type II model. KVM-based Type II hypervisors remain a mainstream solution in modern cloud infrastructures (e.g., Google Cloud [10] and AWS Nitro [54]), making this deployment setting representative of real-world practice.

M3U is highly compatible with the Linux kernel because it relies on stable standard interfaces provided by the kernel MMU for external management modules [32–35], and the kernel preserves the compatibility of these interfaces across internal feature evolutions. Therefore, M3U can be integrated into recent kernels with minimal adaptation. We have verified M3U's compatibility with Linux kernels 4.19, 5.15 and 6.6 in our cloud environment; we use the implementation with Linux kernel 4.19 in our evaluation.

Device memory management. M3U manages the VM memory separately through Linux `/dev/mmap` interfaces, so that it can interact with the original kernel MMU through standard system abstractions, while implementing its own page-fault handler and page-table management logic (e.g., mapping/unmapping).

Demand paging. Similar to QEMU's post-copy preempt feature [46], we employ a single data stream for demand paging that transfers dirty pages immediately without batching. Unlike batched transfer in active pushing for higher throughput, this allows a faulted page to be transferred with minimum latency. In addition, we refine the function calls irrelevant to post-copy during vCPU page fault handling to reduce demand paging latency.

Parallelism tuning. For parallel dirty page registration (§4.1), we assign one worker thread in the thread pool per 8 GB guest memory. The thread count is no more than 16, where it

achieves the maximum efficiency improvement.

For parallel data streams (§4.2), we implement the page transfer framework based on the QEMU multifd [44]. We employ 6-8 data streams for active pushing, while the demand paging always employs a single data stream. This practical optimal configuration (verified in our evaluation) enables the paging efficiency to reach over 70% of the physical network bandwidth on a single host server, which is generally beyond the bandwidth limit for live migration.

6 Evaluation

6.1 Platform and Workload

The evaluation platform comprises a pair of source and target nodes for live migration, with guest VMs deployed as clients on these nodes. Each node is equipped with dual-socket, 32-core Intel Xeon 8369B CPUs, 512 GB DDR4 memory, and a DPU card (PCIe Gen3 connection with 8 lanes, supporting a maximum physical network bandwidth of 200 Gbps).

M3U supports all major Linux-based guest OSes and uses Ubuntu 20.04.6 with Linux kernel 5.4 in evaluation. Our high-end guest VM is configured by default with 64 vCPUs, 256 GB of memory, and a 100 Gbps virtual network interface controller (vNIC) enabled via I/O pass-through[1]. To obtain stable performance measurements, we pin each vCPU to a separate hyper-threaded core co-located on a single NUMA socket. The host OS is statically configured to use 2 MB huge pages by default, with Transparent Huge Page (THP) [5] disabled to optimize guest performance. Correspondingly, the guest OS has huge pages disabled by default.

Baseline. For our first baseline, we begin with the community version of QEMU 8.2 and Linux kernel 4.19. We also implement pass-through I/O virtualization in the baseline using a state-of-the-art DPU-based VirtIO offloading approach [60]. To handle IOPFs, we adopt the same hardware solution employed in recent works [21, 59], which relies on the Address Translation Service (ATS) and Page Request Interface (PRI) as defined by the PCI standard [40].

For our second baseline, to demonstrate M3U's advantages in paging efficiency and PCT, we introduce the TDP MMU, a state-of-the-art approach for kernel MMU scalability (explained in §2.2). TDP MMU adoption requires a recent Linux kernel version that is incompatible with our DPU offloading virtualization platform [60]. Consequently, we deployed TDP MMU on a bare-metal machine with an identical configuration to M3U's setup, except without the DPU device, using QEMU 8.2 and Linux kernel 5.15. Since this baseline deployment does not include modifications to dirty page registration and lacks support for pass-through devices during post-copy live migration, we limit our comparison of TDP MMU and our approach to paging efficiency and PCT metrics, excluding downtime, guest performance, and I/O-related measurements.

[1]The same configuration is used in the `n4-standard-64` VM type on Google Cloud Platform, which costs 2.2K USD per month per node.

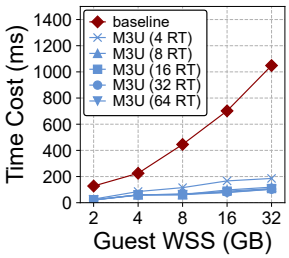


Figure 12: Dirty page registration overhead. "RT" denotes registration thread.

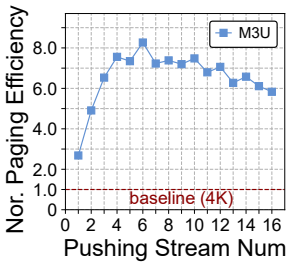


Figure 13: Normalized paging efficiency using baseline with 4 KB page size as 1.0×.

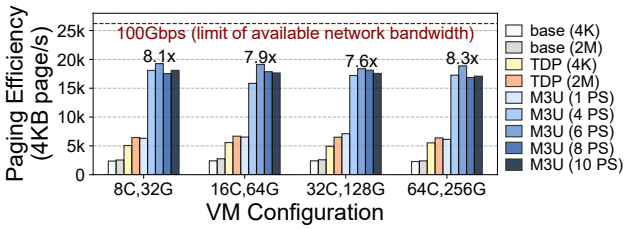


Figure 14: Paging efficiency of post-copy under different VM configurations. "PS" denotes the number of pushing streams.

To perform practical live migration, we adopt a "*hybrid*" method [23, 50]—a state-of-the-art approach that combines pre-copy and post-copy phases—which has become the standard in practice [49]. We enforce only one round of hybrid pre-copy before the live migration enters the downtime phase to highlight the performance differences between M3U and the baseline approaches during post-copy. The available network bandwidth for live migration is 100 Gbps (max 200 Gbps; 100 Gbps for guest VM, 100 Gbps for host).

Workload. In high-end VMs, we run Liblinear [14] and Llama.cpp [17] as the compute-intensive workloads. We run Graph500 [19], as well as YCSB [11] and Memtier [48] benchmarks on Memcached [36] and Redis [47] as the memory-intensive workloads. We use iperf3 [20] and FIO [6] for network and block I/O intensive workloads, respectively.

6.2 Micro-benchmark

Dirty page registration. Figure 12 demonstrates that M3U reduces the time cost of dirty page registration significantly by up to 90.2%. For all downtime-related tests, we employed memory random write operations with 1 KB block size as the workload and progressively increased the workload intensity. The test results show that M3U effectively reduces the time cost of dirty page registration by 60.0–90.2%, demonstrating better scalability across varying workload intensities compared to the baseline. Additionally, we observed that the time cost of dirty page registration plateaus at 16 worker threads; therefore, we configured this value as the default thread count for the worker thread pool.

Management of stream number. To study the scalability

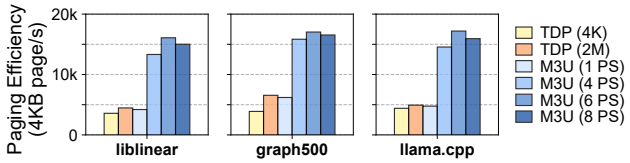


Figure 15: Comparison of paging efficiency under varying guest workloads.

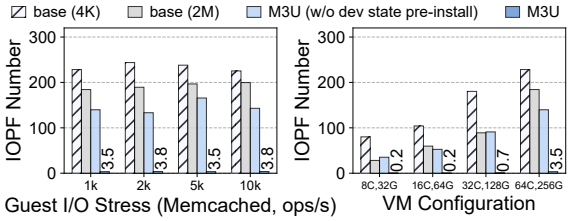


Figure 16: IOPF number under varying guest I/O stress levels and VM configurations.

of multi-pushing (§4.2) and how the number of concurrent streams should be managed, we adopt a practical evaluation on paging efficiency while systematically increasing the number of pushing streams from 1 to 16. The results in Figure 13 show that M3U achieves the highest paging efficiency with 6 pushing streams (7.6× improvement over the 4 KB baseline due to the throughput gains enabled by multi-pushing). Since the available host bandwidth is 100 Gbps, M3U with 6 streams already achieves the maximum migration bandwidth of approximately 80 Gbps. In addition, we can adjust the number of streams from 1 to 6 to almost linearly increase M3U's paging efficiency to better utilize the available migration bandwidth. As future work, we can further utilize advanced network solutions (e.g., RDMA or CXL) to reduce transmission overhead.

Paging efficiency. Figure 14 shows that M3U achieves superior paging efficiency compared to both the baseline approach and TDP MMU. The paging-efficiency upper bound derived from the physical network limit (dotted line in Figure 14) is calculated by dividing the maximum network bandwidth of 100 Gbps by the base page size of 4 KB.

Compared to the baseline approach using 4 KB page size, M3U achieves a paging efficiency improvement of 7.6–8.3× in the best case when utilizing 6 data streams for active pushing. Even with a single data stream for active pushing, M3U outperforms the baseline by 1.6–2.1× due to the performance gains enabled by asynchronous decoupling between page data copying and cross-table consistency maintenance.

Compared to TDP MMU, M3U with 6 active pushing streams delivers the most substantial improvement. Using a 4 KB page size, TDP MMU achieves a 1.4–1.6× improvement compared to the baseline, which is less than M3U with 1 active pushing stream compared to the baseline. As expected, larger page sizes enhance paging efficiency. When employing a larger 2 MB page size, paging efficiency improves by up to 1.2× for the baseline and 2.7× for TDP MMU.

In addition, to evaluate paging efficiency across diverse guest workloads, we select several resource-intensive workloads and run them continuously in the high-end VM during post-copy: (1) Liblinear (50% CPU, 20% memory), (2) Graph500 (66% memory), and (3) Llama.cpp (benchmarking llama-3-8B, 75% CPU, 20% memory). The results in Figure 15 show that, consistent with the prior results under idle conditions, using 6 active-pushing streams remains the optimal configuration across all workloads. Under this configuration, M3U outperforms TDP MMU by 3.9–4.5× with a 4 KB page size and by 2.6–3.6× with a 2 MB page size.

IOPF reduction. Figure 16 illustrates that M3U reduces the total IOPF number to 0.2–3.8 IOPFs per post-copy live migration of high-end VMs, across various guest I/O stress levels (using the Memcached benchmark) and with different VM configurations. Compared to the baseline, M3U achieves up to 98.5% reduction in IOPFs during post-copy migration by leveraging device state pre-installation.

First, as shown in Figure 16 (left), the total IOPF number for M3U remains consistently low (3.5–3.8 IOPFs per post-copy live migration) across different guest I/O stress levels. Next, Figure 16 (right) shows that M3U achieves 0.2–3.5 IOPFs per live migration across all tested VM configurations, whereas the baseline approaches' IOPF counts (with two page size configurations) scale proportionally with VM configuration due to the high-end VM's elevated memory dirty rates and larger total dirty page volumes. Lastly, even without device state pre-installation, M3U achieves fewer IOPFs overall owing to its improved paging efficiency. Similarly, the baseline using 2 MB pages incurs fewer IOPFs than the variant using 4 KB pages for the same reason.

6.3 Migration Metrics

Downtime. Figure 17 shows that M3U achieves up to 47.0% downtime reduction compared to the baseline. With lock-reduced parallelism, the overhead of dirty page registration during downtime decreases from 40.2–64.4% to 6.1–15.4%. The device state pre-installation incurs a time cost of 94–304 *ms*, which accounts for 12.7–29.0% of the total downtime. Although device state migration in M3U requires an additional 131.6 *ms* on average due to the device state pre-installation, this overhead remains marginal relative to the total downtime.

Apart from the costs associated with dirty page registration and device state migration, the remaining downtime costs between M3U and the baseline are relatively similar, primarily due to CPU state migration (0.2–1.4 *s*) and VM startup (121–169 *ms*). We attempted to reduce the cost of CPU state migration for further improvement; however, this approach proved highly challenging because CPU state migration strongly depends on the hardware-specific interfaces in QEMU-KVM.

PCT. Figure 18 illustrates that M3U delivers the shortest PCT

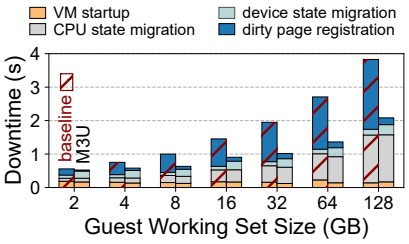


Figure 17: Downtime comparison.

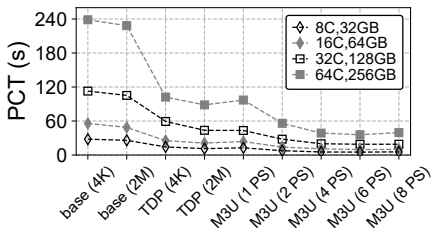


Figure 18: PCT comparison under varying VM configurations.

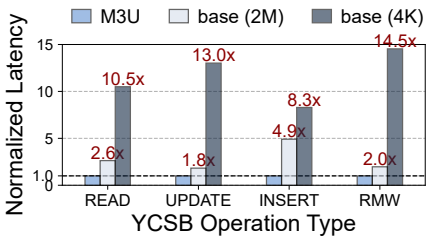


Figure 19: YCSB benchmark on guest Memcached (normalized latency).

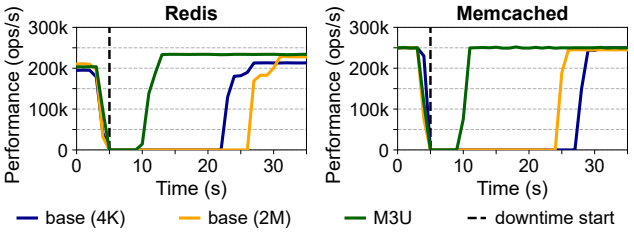


Figure 20: Memtier benchmark on guest Redis and Memcached services (throughput).

compared to TDP MMU and the baseline. We varied the total number of dirty pages across all test suites to demonstrate PCT differences. The results reveal three key findings. First, due to improved paging efficiency, M3U achieves an 85.8–89.6% reduction in PCT compared to the baseline. Even with only a single data stream, M3U reduces PCT by 33.4–55.6%. Second, TDP MMU also significantly reduces PCT by up to 55.7% compared to the baseline; however, this reduction is less than M3U's reduction with ≥ 2 data streams. Third, page size emerges as a deterministic factor for PCT reduction: TDP MMU and the baseline achieve additional PCT reductions of 25.7% and 12.6%, respectively, when using 2 MB pages. This result also indicates that M3U's fault-aware page size determination is highly effective.

Guest performance. Figure 19 shows the normalized operation latency of the YCSB benchmark on the guest Memcached service, demonstrating a substantial reduction. This evaluation encompasses four operation types that constitute all YCSB workloads (a–f) in different ratios, with the exception of the "Scan" operation, which is unsupported in Memcached. The "Read-Modify-Write" operation is abbreviated as "RMW" for clarity. The results show that M3U achieves 1.8–4.9× lower operation latency than the baseline configuration with 2 MB pages, and 8.3–14.5× lower latency than the baseline using 4 KB pages.

Figure 20 shows the real-time performance curves of the Memtier benchmark for guest Redis and Memcached services executing SET operations. To evaluate M3U's effectiveness, we quantify the service throughput degradation during the post-copy phase by the performance curve valley. The results show that M3U achieves a 2.6–4.1× reduction in throughput loss compared to the baseline approach, thereby significantly improving guest performance.

7 Discussions

IOPF elimination. M3U eliminates up to 98.5% of the IOPFs in post-copy, as tested in §6.2. The remaining IOPFs stem from non-atomic virtqueue processing in the guest VirtIO front-end (FE). Specifically, a timing window exists between (a) when the VirtIO FE allocates a buffer, and (b) when it posts this buffer into a descriptor. If live migration enters the downtime phase while virtqueue processing occurs within this interval, an IOPF will occur if the buffer is dirty, even though M3U has already performed device state pre-installation.

To completely eliminate IOPFs in post-copy migration, one straightforward approach would be to design a para-virtualized mechanism that guarantees atomic virtqueue buffer processing during the downtime phase of post-copy live migration. However, this approach introduces significant complexity and requires substantial modifications to the guest VirtIO FE. Moreover, it would make the guest VM fully aware of the ongoing post-copy live migration, thereby violating the principle of guest transparency. Consequently, M3U does not currently pursue this approach and takes this challenge as future work.

Failure tolerance. Post-copy is vulnerable to failures because the VM's memory state is bifurcated across both the source and target hosts during post-copy, and failure on either host will make the VM unrecoverable. Recently, a failure-tolerance approach for post-copy has been proposed based on the checkpoint/restore technique [16]. It is a software-based approach that M3U can adopt for failure tolerance in post-copy.

Generalizability. In brief, M3U is a fully generalizable approach for the community solutions. We discuss the generalizability by dividing the designs into two separate parts: pass-through device related and the others. For the latter part, M3U's designs do not rely on any new hardware features, which can be applied to community solutions like QEMU-KVM on commercial off-the-shelf servers as we evaluated. For the pass-through device related part, M3U's solution, i.e., the one-time transmission of device states during downtime

is device-independent, which can also be applied to diverse pass-through devices.

Overcommitment. Live migration naturally coexists with memory overcommitment, an essential property for cloud elasticity. Since the entire VM memory must eventually be transferred during migration, reclaimed memory is restored before migration proceeds. During restoration, the host hypervisor swaps in reclaimed pages and reconstructs the corresponding host-side HVA-to-HPA mappings in the background. Guest execution remains isolated from the host-side restoration because guest GVA-to-GPA mappings remain unchanged. The only interaction occurs when guest applications access reclaimed pages, where restoration introduces neither additional interference nor guest-visible overhead because both behaviors invoke the same underlying swap-in path. Consequently, migration executes with fully resident VM memory and without overcommitment, as commonly adopted in existing live migration systems. Based on this invariant, M3U prevents the target host from swapping migrated memory until migration completes, thereby eliminating additional complexity and overhead. Outside the migration period, overcommitment remains fully supported on our platform.

Applicability to Type I hypervisors. The core techniques in M3U (e.g., static memory allocation, page flagging, separate address space, etc.) also extend to Type I hypervisors, particularly to lock-protected memory management paths in the hypervisor and userspace migration components. Therefore, the design principles in M3U remain applicable across both Type I and Type II hypervisors.

8 Related Work

Post-copy live migration. The post-copy scheme [22] was introduced to reduce the total migration time and transferred data compared to the conventional pre-copy scheme. By directly switching the VM to the target host and lazily transferring all remaining memory data, post-copy effectively eliminates the pre-copy convergence problem [31, 52]. In contrast, other optimized pre-copy approaches (e.g., compression [27, 53] and CPU throttling [42]) can only mitigate, rather than eliminate, the convergence problem while simultaneously introducing host resource over-consumption or negatively impacting guest performance.

Post-copy migration is well-known for delivering poor guest service performance during extended PCT, especially when migrating high-end VMs. Previous studies have attempted to address this challenge through two primary approaches: (a) incorporating a hybrid pre-copy phase [23, 30, 50], or (b) actively and adaptively pushing dirty pages during the post-copy phase [22, 45, 52]—both we have adopted as our baseline strategy. Abe et al. [1] propose a para-virtualized post-copy approach that enables coordination between the host VMM and guest OS to identify VM's working sets for prioritized transfer. However, we do not adopt this approach because it violates the guest transparency principle. Note that the existing literature does not adequately address how this performance penalty amplifies when migrating high-end VMs, nor does it identify the root cause: specifically, the limitations of demand paging and active page pushing mechanisms, as well as the underlying scalability bottlenecks in kernel MMU.

I/O page fault. Resolving IOPFs caused by pass-through device DMA on non-present I/O pages—a common challenge in overcommitted cloud infrastructures—is a well-established research area [3, 4, 12, 13, 21, 56, 58]. Most existing studies focus on reducing the number of IOPFs or minimizing associated latency; however, they are generally constrained by performance bottlenecks or compatibility limitations. A recent approach, VIO [59], introduces an elastic pass-through mechanism that converts most IOPFs into CPU-side page faults. However, this approach does not further reduce the frequency or the cost of these page faults.

Regarding post-copy live migration, Tian et al. [55] present a VFIO-based solution that extends post-copy support to pass-through devices. Nevertheless, this approach suffers from limited device compatibility—it is specifically optimized for Intel XXV710—and introduces significant overhead from the VFIO stack. In contrast, our work adopts an IOPF handling solution based on the PRI/ATS hardware features, which ensures broad device compatibility. Notably, our work is the first study to successfully eliminate the vast majority of the IOPF handling costs during post-copy live migration.

9 Conclusion

M3U is a scalable kernel MMU design that eliminates the unacceptable performance penalties of high-end post-copy live migration. It addresses kernel MMU scalability bottlenecks in HPT manageability, cross-table consistency maintenance, and IOPT fault handling through three key techniques: (1) lock-reduced parallelism for dirty page registration, which reduces total downtime by up to 47.0%; (2) a decoupled userfault handling pipeline based on parallel data copying with a separate PVA and mixed page table updates, which reduces PCT by 89.6% and improves guest performance by up to 4.1×; and (3) identification and pre-transmission of pass-through device states, which eliminates up to 98.5% of IOPFs.

The design principles of M3U are fully generalizable to existing open-source solutions, providing clear guidance to the community on adding high-end VM support for post-copy live migration.

Acknowledgments

We thank the shepherd and anonymous reviewers for their valuable comments. This paper was supported in part by the Alibaba Innovative Research Program and the National Natural Science Foundation of China (Grant No. 62232012). Jian Li and Chao Zhang are the corresponding authors.

References

[1] Yoshihisa Abe, Roxana Geambasu, Kaustubh R. Joshi, and Mahadev Satyanarayanan. Urgent virtual machine eviction with enlightened post-copy. *ACM SIGPLAN Notices*, 51(7):51–64, 2016.

[2] Raja Wasim Ahmad, Abdullah Gani, Siti Hafizah Ab Hamid, Muhammad Shiraz, Abdullah Yousafzai, and Feng Xia. A survey on virtual machine migration and server consolidation frameworks for cloud data centers. *Journal of Network and Computer Applications*, 52:11–25, 2015.

[3] Nadav Amit, Muli Ben-Yehuda, Dan Tsafrir, and Assaf Schuster. vIOMMU: Efficient IOMMU emulation. In *USENIX ATC 2011*, 2011.

[4] Nadav Amit and Michael Wei. The design and implementation of hyperupcalls. In *USENIX ATC 2018*, pages 97–112, 2018.

[5] Andrea Arcangeli. Transparent hugepage support. In *KVM Forum*, volume 9, 2010.

[6] Jens Axboe. GitHub – axboe/fio: Flexible I/O Tester. https://github.com/axboe/fio, 2008. [Accessed 2026-06-01].

[7] Anton Beloglazov and Rajkumar Buyya. OpenStack Neat: a framework for dynamic and energy-efficient consolidation of virtual machines in OpenStack clouds. *Concurrency and Computation: Practice and Experience*, 27(5):1310–1333, 2015.

[8] Christopher Clark, Keir Fraser, Steven Hand, Jacob Gorm Hansen, Eric Jul, Christian Limpach, Ian Pratt, and Andrew Warfield. Live migration of virtual machines. In *NSDI 2005*, pages 273–286, 2005.

[9] Google Cloud. Machine families resource and comparison guide | Compute Engine | Google Cloud Documentation. https://cloud.google.com/compute/docs/machine-resource, 2025. [Accessed 2026-06-01].

[10] Google Cloud. Compute Engine overview | Google Cloud Documentation. https://docs.cloud.google.com/compute/docs/overview, 2026. [Accessed 2026-06-01].

[11] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with YCSB. In *SoCC 2010*, pages 143–154, 2010.

[12] Yiyuan Dong and Zeyu Mi. IOGuard: Software-based I/O page fault handling with one CPU core. In *Proceedings of the 15th Asia-Pacific Symposium on Internetware*, pages 337–346, 2024.

[13] Alexander Duyck. mm / virtio: Provide support for free page reporting. https://lwn.net/Articles/808807, 2020. [Accessed 2026-06-01].

[14] Rong-En Fan, Kai-Wei Chang, Cho-Jui Hsieh, Xiang-Rui Wang, and Chih-Jen Lin. LIBLINEAR: A library for large linear classification. *Journal of Machine Learning Research*, 9:1871–1874, 2008.

[15] Fedora Project. Dirty Bitmaps and Incremental Backup. https://kashyapc.fedorapeople.org/QEMU-Docs-v4.0.0-143-g1cb2ca0415/_build/html/docs/interop/bitmaps.html, 2019. [Accessed 2026-06-01].

[16] Dinuni Fernando, Jonathan Terner, Kartik Gopalan, and Ping Yang. Live migration ate my VM: recovering a virtual machine after failure of post-copy live migration. In *INFOCOM 2019*, pages 343–351, 2019.

[17] Georgi Gerganov. GitHub – ggml-org/llama.cpp: LLM inference in C/C++. https://github.com/ggml-org/llama.cpp, 2023. [Accessed 2026-06-01].

[18] Google. Introduce the TDP MMU. https://lwn.net/Articles/832835/, 2020. [Accessed 2026-06-01].

[19] Graph500 Project. GitHub – graph500/graph500: Graph500 reference implementations. https://github.com/graph500/graph500.git, 2010. [Accessed 2026-06-01].

[20] Vivien Gueant. iPerf – The TCP, UDP and SCTP network bandwidth measurement tool. https://iperf.fr/, 2009. [Accessed 2026-06-01].

[21] Kaijie Guo, Dingji Li, Ben Luo, Yibin Shen, Kaihuan Peng, Ning Luo, Shengdong Dai, Chen Liang, Jianming Song, Hang Yang, Xiantao Zhang, and Zeyu Mi. VPRI: efficient I/O page fault handling via software-hardware co-design for IaaS clouds. In *SOSP 2024*, pages 541–557. ACM, 2024.

[22] Michael R Hines and Kartik Gopalan. Post-copy based live virtual machine migration using adaptive pre-paging and dynamic self-ballooning. In *VEE 2009*, pages 51–60, 2009.

[23] Liang Hu, Jia Zhao, Gaochao Xu, Yan Ding, and Jianfeng Chu. HMDC: Live virtual machine migration based on hybrid memory copy and delta compression. *Applied Mathematics*, 7(2L):639–646, 2013.

[24] Benoit Hudzia. On allowing shorter timeout on Mellanox cards and other tips and tricks. https://www.reflectionsofthevoid.com/2014/02/on-allowing-shorter-timeout-on-mellanox.html, 2014. [Accessed 2026-06-01].

[25] Lung-Hsuan Hung, Chih-Hung Wu, Chiung-Hui Tsai, and Hsiang-Cheh Huang. Migration-based load balance of virtual machine servers in cloud computing by load prediction using genetic-based methods. *IEEE Access*, 9:49760–49773, 2021.

[26] Khaled Z Ibrahim, Steven Hofmeyr, Costin Iancu, and Eric Roman. Optimized pre-copy live migration for memory intensive applications. In *SC 2011*, pages 1–11, 2011.

[27] Hai Jin, Li Deng, Song Wu, Xuanhua Shi, and Xiaodong Pan. Live virtual machine migration with adaptive, memory compression. In *2009 IEEE International Conference on Cluster Computing and Workshops*, pages 1–10. IEEE, 2009.

[28] Vincent Kherbache, Eric Madelaine, and Fabien Hermenier. Planning live-migrations to prepare servers for maintenance. In *Euro-Par 2014 Workshops, Revised Selected Papers, Part II*, pages 498–507. Springer, 2014.

[29] Tuan Le. A survey of live virtual machine migration techniques. *Computer Science Review*, 38:100304, 2020.

[30] Chunguang Li, Dan Feng, Yu Hua, and Leihua Qin. Efficient live virtual machine migration for memory write-intensive workloads. *Future Generation Computer Systems*, 95:126–139, 2019.

[31] Handong Li, Guangrong Xiao, Yulei Zhang, Ping Gao, Qiumin Lu, and Jianguo Yao. Adaptive live migration of virtual machines under limited network bandwidth. In *VEE 2021*, pages 98–110, 2021.

[32] Linux Kernel Developers. Memory Management APIs. https://www.kernel.org/doc/html/latest/core-api/mm-api.html, 2026. [Accessed 2026-06-01].

[33] Linux Kernel Developers. Page Tables. https://docs.kernel.org/mm/page_tables.html, 2026. [Accessed 2026-06-01].

[34] Linux Kernel Developers. Process Addresses. https://docs.kernel.org/mm/process_addrs.html, 2026. [Accessed 2026-06-01].

[35] Linux Kernel Developers. The x86 kvm shadow mmu. https://docs.kernel.org/virt/kvm/x86/mmu.html, 2026. [Accessed 2026-06-01].

[36] Memcached. Memcached – a distributed memory object caching system. https://memcached.org/, 2003. [Accessed 2026-06-01].

[37] Michael Nelson, Beng-Hong Lim, Greg Hutchins, et al. Fast transparent migration for virtual machines. In *USENIX ATC 2005*, pages 391–394, 2005.

[38] Mostafa Noshy, Abdelhameed Ibrahim, and Hesham Arafat Ali. Optimization of live virtual machine migration in cloud computing: A survey and future directions. *Journal of Network and Computer Applications*, 110:1–10, 2018.

[39] Shingo Okuno, Fumi Iikura, and Yukihiro Watanabe. Maintenance scheduling for cloud infrastructure with timing constraints of live migration. In *2019 IEEE International Conference on Cloud Engineering (IC2E)*, pages 179–189. IEEE, 2019.

[40] PCI-SIG. Address Translation Services Revision 1.1. https://pcisig.com/PCIExpress/Specs/Base/AddressTranslationServices_1.1, 2009. [Accessed 2026-06-01].

[41] QEMU. QEMU. www.qemu.org, 2003. [Accessed 2026-06-01].

[42] QEMU. ChangeLog/1.6. https://wiki.qemu.org/ChangeLog/1.6, 2013. [Accessed 2026-06-01].

[43] QEMU. Migration — Postcopy — Source side page maps. https://qemu.readthedocs.io/en/v7.2.19/devel/migration.html#source-side-page-maps, 2014. [Accessed 2026-06-01].

[44] QEMU. Features/Migration-Multiple-fds. https://wiki.qemu.org/Features/Migration-Multiple-fds, 2015. [Accessed 2026-06-01].

[45] QEMU. Source behaviour; QEMU documentation. https://www.qemu.org/docs/master/devel/migration/postcopy.html#source-behaviour, 2015. [Accessed 2026-06-01].

[46] QEMU. Postcopy preemption mode; QEMU documentation. https://www.qemu.org/docs/master/devel/migration/postcopy.html#postcopy-preemption-mode, 2023. [Accessed 2026-06-01].

[47] Redis. Redis – Real-time data for agents & apps. https://redis.io/, 2009. [Accessed 2026-06-01].

[48] Redis. GitHub – redis/memtier_benchmark: NoSQL Redis and Memcache traffic generation and benchmarking tool. https://github.com/redis/memtier_benchmark, 2013. [Accessed 2026-06-01].

[49] Adam Ruprecht, Danny Jones, Dmitry Shiraev, Greg Harmon, Maya Spivak, Michael Krebs, Miche Baker-Harvey, and Tyler Sanderson. VM live migration at scale. *ACM SIGPLAN Notices*, 53(3):45–56, 2018.

[50] Shashank Sahni and Vasudeva Varma. A hybrid approach to live migration of virtual machines. In *2012 IEEE International Conference on Cloud Computing in Emerging Markets (CCEM)*, pages 1–5. IEEE, 2012.

[51] Sultan Mahmud Sajal, Luke Marshall, Beibin Li, Shandan Zhou, Abhisek Pan, Konstantina Mellou, Deepak Narayanan, Timothy Zhu, David Dion, Thomas Moscibroda, et al. Kerveros: Efficient and scalable cloud admission control. In *OSDI 2023*, pages 227–245, 2023.

[52] Bin Shi and Haiying Shen. Memory/disk operation aware lightweight VM live migration across data-centers with low performance impact. In *INFOCOM 2019*, pages 334–342. IEEE, 2019.

[53] Petter Svärd, Benoit Hudzia, Johan Tordsson, and Erik Elmroth. Evaluation of delta compression techniques for efficient live migration of large virtual machines. In *VEE 2011*, pages 111–120, 2011.

[54] The Linux Kernel Documentation. Nitro Enclaves. https://www.kernel.org/doc/html/v6.0/virt/ne_overview.html, 2022. [Accessed 2026-06-01].

[55] Kevin Tian. Post-copy Live Migration on Pass-through Devices. https://gitlab.com/qemu-project/kvm-forum/-/raw/main/_attachments/2019/kvm-forum-postcopy-final.pdf, 2019. [Accessed 2026-06-01].

[56] Kun Tian, Yu Zhang, Luwei Kang, Yan Zhao, and Yaozu Dong. coIOMMU: A virtual IOMMU with cooperative DMA buffer tracking for efficient memory management in direct I/O. In *USENIX ATC 2020*, pages 479–492, 2020.

[57] Jan Vesely, Arkaprava Basu, Mark Oskin, Gabriel H Loh, and Abhishek Bhattacharjee. Observations and opportunities in architecting shared virtual memory for heterogeneous systems. In *2016 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 161–171. IEEE, 2016.

[58] Carl A Waldspurger. Memory resource management in VMware ESX server. *ACM SIGOPS Operating Systems Review*, 36(SI):181–194, 2002.

[59] Yun Wang, Liang Chen, Jie Ji, Xianting Tian, Ben Luo, Zhixiang Wei, Zhibai Huang, Kailiang Xu, Kaihuan Peng, Kaijie Guo, et al. To PRI or not to PRI, that's the question. In *OSDI 2025*, pages 75–89, 2025.

[60] Xiantao Zhang, Xiao Zheng, Zhi Wang, Hang Yang, Yibin Shen, and Xin Long. High-density multi-tenant bare-metal cloud. In *ASPLOS 2020*, pages 483–495, 2020.

Compaction-Free Memory Defragmentation for Virtualization via Infinite Guest Physical Address Space

Peixin Zeng[†]*, Hao Huang*[†]*, Yanqi Pan, Wen Xia*[✉]*, Darong Yang, Jiahao Chen, Nan Zhang*
Harbin Institute of Technology, Shenzhen

Abstract

In virtualized environments, guest memory defragmentation is essential for exploiting huge-page benefits and improving application performance. However, existing approaches directly reuse host-side defragmentation and assume a limited guest physical address (GPA) space. As a result, they rely on memory compaction to defragment this constrained space, causing throughput to drop by up to 51% and latency to rise by up to 102% in YCSB-Redis workloads.

This paper proposes INFINIDEFRAG, a compaction-free memory defragmentation technique for virtualization. Our key insight is that the GPA space can be regarded as (nearly) infinite by controlling the mapping between GPA and host physical address (HPA), thereby eliminating the need for guest-side memory compaction. To realize this insight, we introduce (1) *Infinite Address Manager* that expands the GPA space while reclaiming free fragment pages; (2) *Host Memory Guard* that maintains the GPA–HPA mapping and constrains HPA usage within each VM's quota; and (3) *Scalability Optimizer* that scales GPA/HPA space management to multi-thread and multi-VM environments. Experiments on micro-benchmarks and real-world applications show that INFINIDEFRAG outperforms state-of-the-art approaches and can achieve ideal, fragmentation-free performance.

1 Introduction

Virtualization has become a foundational technology to enable resource sharing, workload isolation, and flexible deployment [32, 54]. Therefore, virtual machines (VMs) power a wide range of services, including cloud platforms and enterprise IT environments, which require efficient and scalable system designs. Among these, memory access is one of the most important optimizations to improve VM performance.

Huge page is a widely adopted technique to improve memory access performance [18, 21, 33, 35, 36, 39, 45, 47, 57]. By increasing the page size from 4 KB to 2 MB, and even to 1 GB, huge page significantly lowers the translation lookaside buffer (TLB) miss rates and page table walk (PTW) cycles. In virtualized environments, huge pages are particularly important because they substantially reduce the overhead of two-level address translation, where each memory access traverses from the guest virtual address (GVA) to the guest physical address (GPA), and then to the host physical address (HPA).

However, the effectiveness of the huge page highly depends on the contiguous physical memory, whereas memory fragmentation is a notorious problem that makes the memory discontinuous [28, 42, 48, 56]. Specifically, fragmentation arises when the memory is allocated and freed in a non-deterministic pattern, making it increasingly difficult for the page allocator to find large contiguous regions for huge page allocation.

To solve the problem, existing fragmentation mitigation works primarily focus on two dimensions: (1) *Anti-fragmentation*: optimizing allocation policies to proactively avoid fragmentation [31, 37, 40, 48, 53]. However, such static strategies are tailored to specific scenarios and fail to adapt across diverse workloads. (2) *Defragmentation*: Invoking memory compaction when contiguous regions are unavailable [2, 33, 39, 50, 57]. Yet frequent compaction incurs substantial latency spikes. Our preliminary experiments show that these approaches can reduce throughput by up to 51% and increase latency by up to 102% in YCSB-Redis workloads (§3), thus squandering the performance benefits of huge pages.

This work aims to eliminate the costly guest-side memory compaction in virtualized environments, thereby fully exploiting the benefits of huge pages. Our insight is that *the GPA space can be regarded as (nearly) infinite*[1] *by manipulating the mapping between GPA and HPA*, thereby eliminating the need for guest-side memory compaction. Existing guest systems require memory compaction because they adopt a host-like abstraction that assumes the GPA space is fixed and limited. In reality, GPA is a form of virtual memory, as the guest OS behaves like a host-side user process. Consequently, expensive compaction and page migration are unnecessary. Instead, contiguous memory can be obtained simply by expanding the GPA space and updating the GPA-HPA mapping.

We incarnate this idea by building INFINIDEFRAG, a compaction-free memory defragmentation technique. The primary challenge is how to efficiently manage the infinite GPA space while keeping the host physical memory usage within the VM's quota. To address this, we introduce an array of techniques. First, *Infinite Address Manager* provides new contiguous GPA memory by reclaiming free fragment GPA pages and expanding the new GPA region using memory trade (*i.e.,* trades fragmented pages for contiguous memory), fast reclaim, and reserved region. Second, *Host Memory Guard* maintains the GPA-HPA mapping and constrains the actual

[†] Co-first author.
✉ Corresponding Author: Wen Xia (xiawen@hit.edu.cn).

[1] Modern processors support a PB-scale address space enabled by 57-bit address width, which almost never exhausts (more details are in §3.3, §7).

host memory usage within a VM's quota using self-hosted remap and batch unmapping. Finally, *Scalability Optimizer* further improves GPA/HPA space management efficiency and scales INFINIDEFRAG to multi-thread and multi-VM environments with lockless page-bitmap updates, in-kernel remap with delayed TLB flushes, and hybrid paging.

We implement INFINIDEFRAG[2] as a module of Linux's memory management subsystem, invoked when the standard high-order allocation path cannot satisfy the new huge-page allocation requests. We evaluate INFINIDEFRAG using both micro-benchmarks and real-world applications. The results show that INFINIDEFRAG substantially outperforms state-of-the-art approaches and achieves ideal, fragmentation-free performance. For example, on YCSB-Redis workloads, INFINIDEFRAG improves throughput by 21%–105% compared to LLFREE [53], Linux with transparent huge page (THP) enabled [17], and Linux using the default 4KB pages.

In a nutshell, we make the following contributions:

- We reveal that existing guest-side defragmentation techniques incur significant overhead. The limited GPA space is the primary cause of their overhead, even though GPA can in fact be treated as an infinite virtual address space (§3).
- We propose INFINIDEFRAG, a compaction-free memory defragmentation technique to achieve lightweight defragmentation with three designs: *Infinite Address Manager*, *Host Memory Guard*, and *Scalability Optimizer* (§4).
- We implement and evaluate INFINIDEFRAG. The results show that it substantially outperforms existing approaches with ideal, fragmentation-free performance (§5, §6).

2 Background

2.1 Memory Virtualization

Memory virtualization is a fundamental abstraction in modern VMs, allowing guest operating systems (OSes) to share physical memory efficiently and transparently. By decoupling the guest's view of memory from the underlying hardware, the hypervisor provides each VM with an isolated, contiguous address space, ensuring both safety and flexibility.

Address translation in virtualization. Memory virtualization relies on a two-level address translation: guest virtual addresses (GVAs) are first mapped to guest physical addresses (GPAs), which are then translated to host physical addresses (HPAs). This design provides strong isolation and flexible management but introduces additional overhead.

Mitigating address translation overhead. Modern processors support two-level paging through *nested page tables* (NPT), such as Intel Extended Page Tables (EPT) and AMD Rapid Virtualization Indexing (RVI). EPT, in particular, maintains a second-level page table for GPA → HPA translation, supports multiple page sizes (*i.e.,* 4KB, 2MB, and 1GB), and enforces per-page read, write, and execute permissions. On each memory access, the CPU first translates GVA → GPA via the guest page table, then GPA → HPA via EPT. The results are cached in the TLB and Page Walk Cache [44], allowing fast lookups. However, on a cache miss, both guest page table and EPT should be traversed, imposing significant overhead for workloads with poor TLB locality or frequent memory turnover [39]. *In this paper, we use NPT to represent the two-level paging* (i.e., *GVA → GPA → HPA), while EPT to represent the specific GPA → HPA translation.*

[2] INFINIDEFRAG refers to the compaction-free memory Defragmentation technique using Infinite guest physical address space.

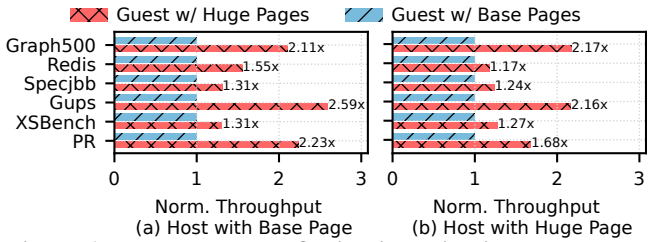


Figure 1: Huge page benefits in virtualization. *We evaluate four configurations combining huge and base pages on both the guest and the host. Experiments are conducted on Linux THP [17] without host/guest fragmentation, and the results show that guest-side huge pages yield substantial performance improvements.*

2.2 Huge Page and Fragmentation

Huge Pages. To reduce the overhead of two-level address translation and improve TLB efficiency, modern systems support huge pages (e.g., 2 MB and 1 GB on x86-64), which decrease TLB misses, shorten page walks, and boost performance—particularly for memory-intensive workloads. Specifically, in virtualized environments, hardware features such as Intel EPT allow huge pages at both the guest and host levels, reducing nested page table entries and improving translation efficiency to near-native levels. As shown in Figure 1, employing huge pages on the guest can significantly enhance performance, regardless of the host-side configuration.

However, huge pages require physically contiguous regions, but long-running systems often suffer from external fragmentation. Consequently, even with sufficient free memory, allocating huge pages becomes increasingly difficult, highlighting the need for an efficient huge page management strategy.

Huge page management in Linux. Linux originally introduces hugetlb pages [13] to support huge pages, but they are ad-hoc and non-transparent to applications. To address this limitation, the Linux kernel provides transparent huge pages (THP) [17], which automatically promotes base pages into huge pages. THP relies on a background kernel thread (*i.e., khugepaged*) to coalesce contiguous 4 KB base pages into 2 MB huge pages and may additionally promote pages during page faults when conditions permit. This design enables applications to benefit from reduced TLB misses and shorter page walks without modification. However, THP critically depends on memory compaction to obtain physically contiguous memory, which can introduce substantial latency spikes, particularly under memory pressure or fragmentation.

The fragmentation problem. As systems run, the continual allocation and release of memory inevitably create external

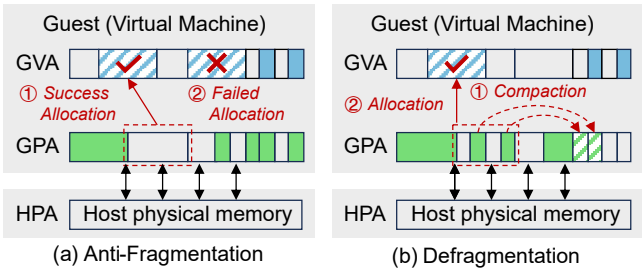


Figure 2: Deficiency of existing fragmentation mitigation techniques. *(a) Anti-fragmentation is tailored to dedicated scenarios and fails to scale across diverse workloads, while (b) defragmentation incurs substantial overhead due to page compaction.*

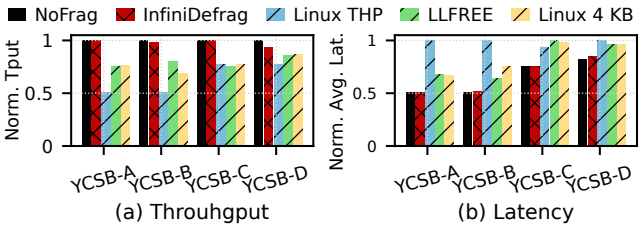


Figure 3: Performance analysis under fragmentation. *The host uses huge pages, NoFrag denotes the non-fragmented baseline using Linux THP, and experiments are conducted under the extreme fragmentation setting (§6.1). The results suggest that existing fragmentation mitigation approaches substantially deteriorate throughput and latency, sometimes even fall below Linux 4 KB.*

fragmentation, substantially reducing the availability of contiguous 2 MB pages. Consequently, THP often falls back to 4 KB pages without extra mechanisms even when the total free memory remains sufficient. Although the buddy allocator [15] attempts to maintain memory contiguity, fragmentation still prevents huge page allocations in practice. Moreover, under memory pressure, background compaction often fails to keep pace with fragmentation, leading to substantial performance degradation for workloads that depend on huge pages [28, 42].

2.3 Existing Fragmentation Mitigation Works

Existing fragmentation mitigation works fall into two categories: (1) *Anti-fragmentation*: optimizing memory allocation to avoid fragmentation; (2) *Defragmentation*: invoking memory compaction when contiguous regions are unavailable.

Anti-fragmentation. These approaches [31, 37, 40, 41, 48, 53] mitigate memory fragmentation by employing specialized allocation policies that carefully control page placement decisions to preserve large contiguous regions. For example, as shown in Figure 2a, several approaches [37, 40] co-locate objects or pages with similar lifetimes or sizes to improve spatial locality and maintain memory contiguity.

Although such proactive strategies can delay fragmentation, they suffer from inherent limitations. Specifically, (1) their effectiveness depends on heuristics or assumptions about future allocation patterns, which often fail under dynamic and heterogeneous workloads, leading to inevitable fragmentation. (2) They may also trade short-term allocation efficiency for preserving contiguous regions, leading to unnecessary memory waste for no-fragmentation scenarios. (3) They typically require modifications to allocators or applications, significantly limiting their deployability in production environments that demand transparency and backward compatibility.

Defragmentation. These techniques [2, 39, 47, 50, 56] rely on relocating movable pages to coalesce scattered free pages, thereby creating large contiguous regions (*i.e.,* Figure 2b). For example, Linux THP performs both background and on-demand compaction to facilitate 2 MB huge page allocation even under severe fragmentation, as discussed in §2.2.

However, reactive strategies introduce substantial runtime overhead. Specifically, (1) relocating pages requires data copy, page-table updates, and TLB shootdowns, which disrupt execution and often cause noticeable latency spikes. (2) Pages that are pinned, shared, or hardware-mapped are inherently unmovable [34, 48, 56], fundamentally limiting how much contiguous space compaction can recover. (3) Compaction competes for CPU cycles and memory bandwidth, and is frequently throttled under memory pressure, making it less effective when fragmentation is extremely severe.

Overall, prior works primarily mitigate memory fragmentation at the host level, with benefits indirectly inherited by guests. To our knowledge, this is the first work to design a dedicated guest-side defragmentation technique.

3 Observations and Motivations

3.1 Limitations of Existing Approaches

In this section, we begin with preliminary experiments to examine whether existing fragmentation mitigation approaches can improve the performance of foreground applications.

Methodology. We evaluate throughput and latency using YCSB [23] on Redis [16]. The host uses huge pages, and experiments are conducted under the *extreme fragmentation setting* (see details in §6.1). Moreover, we select LLFREE [53] and Linux THP [17] to represent anti-fragmentation and defragmentation approaches, respectively. We also compare against Linux with 4 KB pages and *NoFrag*, which represents Linux THP without fragmentation. Specifically, LLFREE is a memory allocator designed to replace the Linux buddy system [15], which applies heuristics to proactively reduce potential future fragmentation. Linux THP, on the other hand, relies on background huge-page promotion or foreground synchronous page compaction to defragment memory (§2.2).

Observation #1: State-of-the-art fragmentation mitigation techniques exhibit poor performance. As shown in Figure 3, both anti-fragmentation (*LLFREE*) and defragmentation (*Linux THP*) achieve only 49%–72% of *NoFrag* throughput and incur 30%–102% higher latency. In some cases, their performance even falls below that of *Linux 4 KB*, which does not exploit huge pages. These results suggest that the overhead imposed by existing fragmentation mitigation techniques outweighs the performance gains provided by the additional huge pages they produce, as further studied in the next subsection.

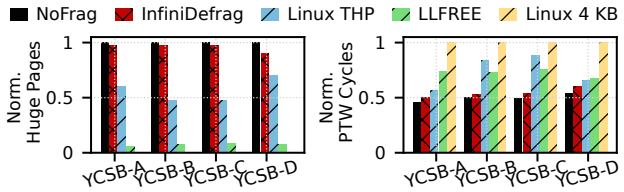


Figure 4: Huge page and PTW cycle analysis under fragmentation. *The host uses huge pages, NoFrag denotes the non-fragmented baseline using Linux THP, and experiments are conducted under the extreme fragmentation setting (§6.1). Results in subfigure (a) suggest that existing approaches fail to provide sufficient huge pages, which result in high PTW cycles in subfigure (b).*

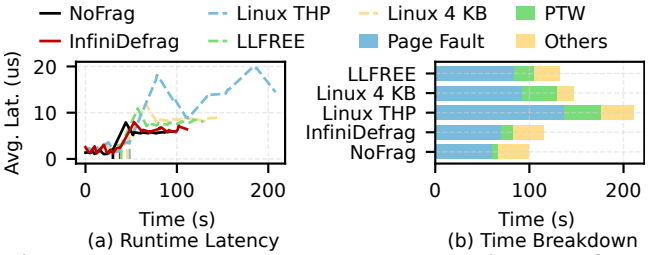


Figure 5: Latency and breakdown analysis under fragmentation. *The host uses huge pages, NoFrag denotes the non-fragmented baseline using Linux THP, and experiments are conducted under the extreme fragmentation setting (§6.1). (a) The vertical line indicates the end of the load phase, after which is the run phase. (b) Memory compaction increases the page fault time, while huge pages reduce the PTW cycles and the page fault time.*

3.2 Deficiency Analysis

We then conduct detailed experiments to demystify the deficiency of existing fragmentation mitigation approaches.

Methodology. The experimental setup follows that described in §3.1. Specifically, we: (1) measure the number of allocated huge pages to assess the ability to maintain contiguous memory regions (Figure 4a); (2) record page table walk (PTW) cycles to evaluate the efficiency of two-level address translation (Figure 4b); (3) measure real-time request latency to capture performance fluctuations over time (Figure 5a); and (4) perform time-breakdown to analyze how compaction and huge page contribute to overall performance (Figure 5b).

Observation #2: Anti-fragmentation performs poorly due to limited number of huge pages. As shown in Figure 4a, when memory becomes fragmented, anti-fragmentation (*i.e., LLFREE*) fails to provide sufficient huge pages compared to the ideal case (*i.e., NoFrag*). Although LLFREE is designed to preserve contiguous regions, it ends up allocating the fewest huge pages (*Linux 4 KB* does not have huge pages), resulting in longer *Page Fault* and *PTW* time, as shown in Figure 5b.

Observation #3: Defragmentation performs poorly due to substantial compaction overhead. Although defragmentation (*i.e., Linux THP*) is able to allocate more huge pages and reduce PTW cycles (Figure 4b), it still delivers the longest average latency—even worse than *Linux 4 KB* without huge pages (Figure 5a). The reason is illustrated in Figure 5b, where

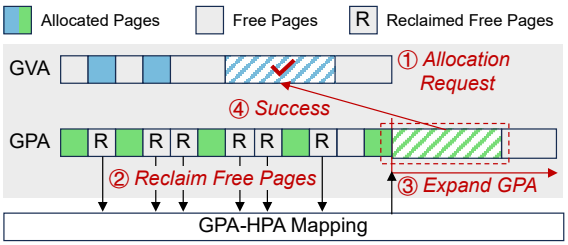


Figure 6: The Infinite GPA space. *It enables efficient defragmentation without any compaction or page migration, by simply controlling the mapping between the GPA and HPA pages.*

Linux THP exhibits much higher *Page Fault* time, either because synchronous compaction increases page-fault handling time or because background page migration increases the number of page faults. These compactions directly interfere with foreground request processing, leading to pronounced latency inflation despite the presence of more huge pages.

Thus, existing approaches face a dilemma: *increasing huge-page availability requires defragmentation effort, yet the resulting overhead can outweigh the performance benefits that huge pages are meant to deliver.*

3.3 Motivation: The Infinite GPA Space

Based on the above analysis, we pose a fundamental question: Is it possible to combine the low overhead of anti-fragmentation with the huge page ability of defragmentation?

Opportunity #1: The GPA space is (nearly) infinite. Modern systems now feature terabytes of physical memory, substantially expanding the physical address space. To accommodate this growth, recent x86 processors, including Intel Ice Lake and Sapphire Rapids, implement 5-level paging [5], thereby increasing the physical address width from 48 to 57 bits. This expansion highlights both the need for scalable memory management and the challenge of system software to efficiently exploit such a vast address space [4].

We quantify the write intensity of guest memory using the dirty-page rate, a standard metric used in prior live-migration studies [22, 30]. Existing work reports that cloud workloads often dirty memory at tens of MB/s, and 32 MB/s is the point that pre-copy migration becomes ineffective [30]. At this rate, a VM produces roughly 2.76 TB of modified memory per day. Given a 57-bit address space (*i.e.,* $\sim 1.44 \times 10^{17}$ bytes), the system can sustain about 4.7×10^4 days—over a century before exhaustion. Moreover, even under higher dirty-rate assumptions, the lifetime remains on the order of decades.

Opportunity #2: GPA-to-HPA indirection enables flexible remapping. By decoupling a VM's view of physical memory (*i.e.,* GPA) from the underlying real machine layout (*i.e.,* HPA), the virtualization layer allows the host to transparently reorganize memory pages without impacting the guest. This indirection enables defragmentation strategies that are both low-overhead and non-intrusive to guest workloads.

Insight: Compaction is unnecessary with the infinite GPA space. With the two opportunities, a VM's physical memory

(*i.e.,* GPA) can be treated as an infinite address space, mapped to host physical memory (*i.e.,* HPA) on demand. This enables compaction-free defragmentation by controlling the mapping between guest and host memory. We use Figure 6 to illustrate this approach: ① when a huge page allocation request cannot find a contiguous GPA, ② free fragment GPA pages are reclaimed, and ③ the GPA space is expanded with the updated GPA-HPA mapping, ④ thereby reserving a contiguous GPA region to succeed the request without compaction.

Note that maintaining the GPA–HPA mapping during GPA space expansion may introduce host-side fragmentation, especially in multi-VM environments. We regard it as a challenge.

Challenges. However, leveraging this insight to build a practical defragmentation mechanism raises several challenges: (1) How can the VM's infinite physical address space (*i.e.,* GPA) be efficiently managed? (2) How can actual physical memory (*i.e.,* HPA) be constrained within the VM's quota? (3) How can the system scale efficiently under multi-thread and multi-VM environments? We address them in Section 4.

4 INFINIDEFRAG Design

We propose INFINIDEFRAG, a compaction-free memory defragmentation technique. However, designing INFINIDEFRAG is non-trivial, and we first introduce our design goals:

- *Low overhead*. It should be lightweight without affecting the performance of the foreground applications (§4.2).
- *High HPA utilization*. It should ensure that the VM's actual HPA usage does not exceed the resource quota (§4.3).
- *High scalability*. It should provide high scalability under multi-thread and multi-VM environments (§4.4).

4.1 INFINIDEFRAG Overview

Components. Figure 7 presents the major components. First, *Infinite Address Manager* manages the infinite GPA space and performs compaction-free defragmentation, thereby achieving the *low-overhead goal* and addressing *Challenge 1* (§4.2). Second, *Host Memory Guard* maintains the GPA–HPA mappings to ensure that the HPA consumption stays within the VM's quota, thereby fulfilling the *high HPA utilization goal* and addressing *Challenge 2* (§4.3). Finally, *Scalability Optimizer* improves the efficiency of GPA/HPA management in multi-thread and multi-VM environments, thereby achieving the *high scalability goal* and addressing *Challenge 3* (§4.4). Moreover, INFINIDEFRAG is compatible with existing virtualization architectures, allowing traditional VMs and our VMs to run concurrently within the same environment.

Workflow. We use Figure 7 to illustrate the workflow of INFINIDEFRAG: ① When a huge-page allocation request cannot find contiguous GPA space, ② the *Page Allocator* invokes the INFINIDEFRAG module to perform memory defragmentation. First, ③ the *Infinite Address Manager* reclaims free fragment GPA pages and expands the GPA space to the same size; Second, ④ the *Host Memory Guard* reallocates HPA pages to maintain the new GPA–HPA mappings; Finally, ⑤ the newly extended contiguous GPA region is allocated to satisfy the

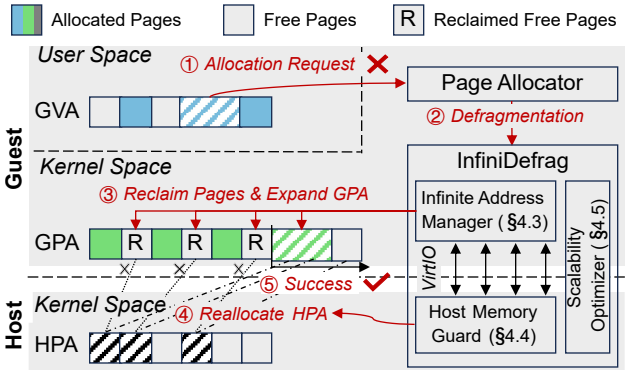


Figure 7: INFINIDEFRAG overview. *The major components are Infinite Address Manager, Host Memory Guard, and Scalability Optimizer, corresponding to our three challenges and design goals.*

huge-page allocation request. Note that INFINIDEFRAG falls back to regular memory compaction (*i.e.,* Linux THP [17]) when the GPA space can no longer be expanded. However, given the petabyte-scale address space and its century-level sustainability (§3.3), this situation rarely happens.

4.2 Infinite Address Manager

Infinite Address Manager leverages *memory trade (§4.2-1)* to manage the infinite GPA space, which consists of two components: *guest reclaimer* (§4.2-2) and *guest extender* (§4.2-3).

1. Managing GPA space through memory trade. INFINIDEFRAG adopts a simple yet efficient idea—*memory trade*, to fully exploit the infinite GPA space while minimizing system overhead, design complexity, and engineering effort.

Specifically, when the GPA space becomes fragmented and the guest OS cannot satisfy a huge page allocation request, it initiates a *memory trade* with the host via *VirtIO* [51]. *Memory trade* refers to the trade of fragmented base pages for a newly contiguous GPA region. In practice, the guest reclaims free fragment pages (via *Guest Reclaimer*) and uses them to extend the GPA space (via *Guest Extender*). The contiguous region is allocated in 1 GB block granularity. This process avoids page migration and compaction; it simply updates the host's page tables (*i.e.,* EPT). Moreover, the reclaimed pages and the expanded memory should satisfy the following equation:

$$N_{frag} * Size_{base_page} = N_{expand} * Size_{mem_block} \quad (1)$$

where N_{frag} is the total number of reclaimed pages; $Size_{base_page}$ is the size of the reclaimed base pages (*i.e.,* 4KB); N_{expand} is the number of memory blocks that will be expanded; and $Size_{mem_block}$ is the size of each memory block.

Note that *memory trade* is performed asynchronously without blocking allocation, so INFINIDEFRAG will fall back to base-page allocation until the *memory trade* completes.

2. Reclaiming GPA pages via guest reclaimer. *Guest Reclaimer* leverages the buddy-system allocator's page-allocation interface [15] to identify free fragment physical pages and forwards their page-frame numbers to the host. Specifically, it reserves these fragmented pages by invoking the allocation interface, preventing regular workloads from

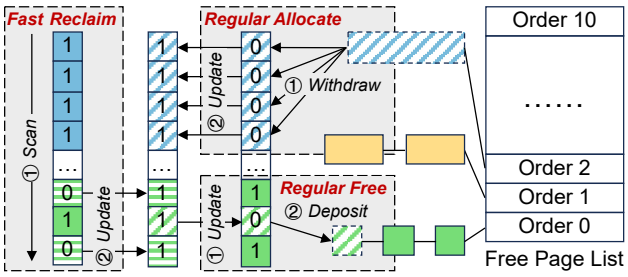


Figure 8: Fast Reclaim vs. Regular Allocation/Free. *Each dashed box is executed atomically. Note that Fast Free is unnecessary, and its workflow (if used) is similar to that of Fast Reclaim.*

acquiring them, and thus achieving logical reclamation. Moreover, *Guest Reclaimer* should coordinate with *Host Memory Guard* to unmap these pages from the QEMU process's (*i.e.,* guest OS) page table and return them to the host's page-frame allocator (§4.3), which is similar to memory ballooning [29]. These operations expose the reclaimed pages to the host, allowing them to be reassigned to other processes.

Optimizing reclamation via fast reclaim. Unfortunately, directly applying the (memory ballooning-like) Guest Reclaimer fails to meet our latency and overhead requirements because the buddy allocator has two limitations [52, 53]:

- *Low Scalability*: Allocating pages requires acquiring the zone lock, which is the bottleneck during burst allocations.
- *High Complexity*: Allocating pages requires complex bookkeeping to maintain buddy allocator free lists and update multiple data structures, increasing engineering overhead.

Therefore, we propose *Fast Reclaim* to minimize page-reclamation overhead. Inspired by Wrenger et al. [52], *Fast Reclaim* uses a bitmap to efficiently track page states. Unlike Wrenger's approach, which requires a custom allocator with extensive modifications, our design remains fully compatible with the guest OS's default buddy allocator [53].

We analyze both *Fast Reclaim* and *Regular Allocation/Free*, as shown in Figure 8. *Fast Reclaim* is triggered to reclaim free fragment pages when the GPA space needs expansion, which introduces a global bitmap to track the state of each page. Specifically, *Fast Reclaim* ① scans the entire bitmap and ② updates each bit accordingly. Moreover, INFINIDEFRAG falls back to *Regular Allocation/Free* for normal page requests. During *Regular Allocation*, it first checks whether a page in the *Free Page List* has already been reclaimed by *Fast Reclaim*. If so, INFINIDEFRAG selects another free page. If the allocation succeeds and the page was not previously marked as allocated, the guest kernel sets the corresponding bitmap bit to 1. Similarly, during *Regular Free*, INFINIDEFRAG clears the bitmap bits to 0 and deposits the pages back to the list.

Note that maintaining consistency between the bitmap and the buddy allocator does not degrade regular allocation/free performance. Newly extended pages are placed at the head of *Free Page List*, making bitmap checks rarely fail in practice (*i.e.,* only check once). Moreover, we further improve *Fast Reclaim*'s scalability by optimizing the synchronization between the bitmap and the allocator, as shown in §4.4.

3. Expanding the GPA space via guest extender. *Guest Extender* first requests additional memory resources and forwards the corresponding page-frame numbers to *Host Memory Guard* as a memory-expansion request. Once *Host Memory Guard* fulfills the request, *Guest Extender* initializes the necessary bookkeeping structures and brings the newly allocated memory online. This guest-host coordination ensures that subsequent EPT faults triggered by accesses to the newly allocated GPA space are legal and correctly handled.

Optimizing expansion via reserved region. When guest memory expands, the VM must invoke the memory hotplug interface to bring newly added pages online. This process requires initializing per-page metadata inside the guest and allocating a corresponding memory-backend device in QEMU, both of which introduce nontrivial overhead. As a result, frequent expansions can noticeably degrade performance.

To address this, we employ two optimizations. First, we expand the GPA space in coarse-grained chunks (*i.e.,* 4 GB, four memory blocks), which balances the trade-off between the cost of bringing blocks online and the frequency of expansions. Second, during system boot, the guest kernel proactively reserves a large memory region (*i.e.,* 96 GB), initializes associated metadata, and keeps it offline. In practice, the reserved region is large enough for most workloads to run for an extended period (at the hour level, §3.3), while preparing this region takes less than 1 s, making expansion rarely happens.

4.3 Host Memory Guard

Host Memory Guard performs *page reallocation* (§4.3-1) to maintain GPA–HPA mappings when the GPA space expands. We then introduce *self-hosted remap* (§4.3-2) and *batch unmapping* (§4.3-3) for the host with base and huge pages.

1. Guarding host resources during page reallocation. *Host Memory Guard* coordinates with the *Infinite Address Manager* to support free GPA-page reclamation and GPA-space expansion, which in turn introduces host-side page reallocation. To ensure that a VM's actual memory usage remains within its quota, *Host Memory Guard* should carefully control memory consumption during this process. Specifically, *page reallocation* maintains the GPA–HPA mapping while ① *unmapping* HPA pages corresponding to reclaimed GPA pages and ② *allocating* new HPA pages for the expanded GPA space.

2. Optimizing the host with base pages via self-hosted remap. When host memory is scarce, the host is unable to immediately allocate enough physical pages to satisfy a guest's memory expansion request. In this case, the host should first unmap the reclaimed pages and release them back to the host's buddy-system allocator. Later, when the guest accesses the expanded GPA space, the corresponding accesses trigger host-side page faults, and the host allocates physical pages from its allocator to resolve them. This workflow introduces two challenges to squander memory expansion efficiency:

- *Long latency*: The memory expansion request cannot be

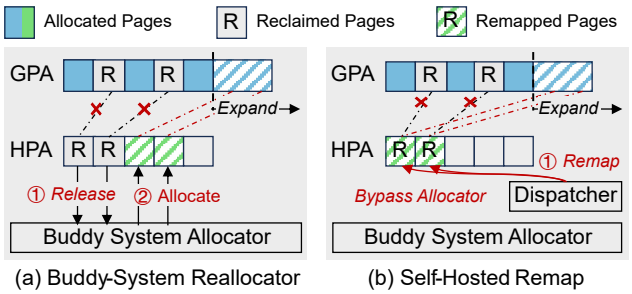


Figure 9: Self-hosted remap. *Compared to buddy system reallocator, self-host remap directly remaps fragmented pages from their original GPA locations into expanded GPA space with low overhead.*

satisfied until page reallocation completes. Because this work occurs inside the VM-exit handler, it incurs substantial latency and stalls the generation of contiguous memory.

- *Excessive complexity*: Each reallocated page goes through multiple steps—① being unmapped from the QEMU process page table, ② returned to the host buddy allocator, and ③ reallocated to serve the newly expanded GPA region.

As shown in Figure 9b, we introduce *self-hosted remap* to address this issue. The key idea is to remap reclaimed fragment pages from their original GPA locations directly into the newly expanded GPA region, bypassing the host buddy allocator and amortizing expensive unmapping operations. Specifically, when the host receives fragment pages from the guest, it no longer blocks in the VM-exit handler until unmapping completes. Instead, it records the corresponding page-frame numbers in *Page Dispatcher*. Later, when an EPT violation occurs for accesses to the expanded GPA region, *Page Dispatcher* fetches one of the recorded page frames and uses it to satisfy the page fault. Note that this technique is enabled when GPA memory is backed by host base pages.

3. Optimizing the host with huge pages via batch unmapping. When the host uses huge pages, INFINIDEFRAG falls back to regular page reallocation (*i.e.,* Figure 9a) because *self-hosted remap* performs in a 4 KB-granularity and will trigger numerous system calls to remap a huge page. Therefore, when the host receives fragmented huge pages, it unmaps them from the QEMU process's page table and returns them to the host page-frame allocator. However, reclaim requests typically arrive in a scattered pattern, and unmapping pages individually incurs substantial overhead: every operation requires a system call (*e.g., madvise* or *munmap*), acquisition of the process's address-space lock, a page-table walk, and cross-core TLB shootdowns [20]. To address this, we introduce batch unmapping. Specifically, all scattered unmapping requests are first recorded in a bitmap; the host then scans this bitmap and issues a single unmapping system call for each contiguous page range, significantly reducing unmapping overhead.

4.4 Scalability Optimizer

Scalability Optimizer scales INFINIDEFRAG to multi-thread and multi-VM environments. Specifically, we first introduce *optimized fast reclaim* (§4.4-1) to scale *Infinite Address Manager*; we then introduce *optimized self-hosted remap* (§4.4-2) and *hybrid paging* (§4.4-3) to scale *Host Memory Guard* for the host with base and huge pages, respectively.

```
KVM_MMU_LOCK(kvm);
// TLB shootdown for EPT
kvm_flush_remote_tlbs(kvm); // <-- Time cousuming
KVM_MMU_UNLOCK(kvm);
...
ptl_lock(src_ptl); // Lock src page table lock
ptl_lock(dst_ptl); // Lock dst page table lock
// TLB shootdown for host page table
flush_tlb_page(vma, addr); // <-- Time cousuming
...
set_pte_at(mm, dst_addr, dst_pte, origi_dst_pte);
ptl_unlock(src_ptl); // Unlock src page table lock
ptl_unlock(dst_ptl); // Unlock dst page table lock
```

Listing 1: Naive in-kernel remap.

1. Scaling infinite address manager via optimized fast reclaim. As shown in Figure 8, the core idea of fast reclaim is to bypass the poorly scalable kernel page allocation path by leveraging a page bitmap. Consequently, the page bitmap should be updated in several situations: (1) whenever a page is allocated or freed through the regular page allocation interface, and (2) whenever the fast reclaim interface is invoked. A straightforward way to maintain consistency between the page bitmap and the buddy allocator is to use a coarse-grained lock to protect the entire bitmap. However, such coarse-grained locking severely limits scalability and introduces substantial overhead to regular memory allocation/free operations.

Therefore, we propose a transaction-based mechanism that relies solely on hardware atomic instructions to update page-bitmap entries. (1) When different threads modify different bitmap entries, no contention or cache-line bouncing occurs. (2) For regular allocation requests (via buddy allocator) targeting compound pages of order ≤ 6, which contain at most 64 base pages, we use an atomic instruction (*i.e., atomic64_try_cmpxchg*) to test and update the corresponding bitmap entry. If it succeeds, the page is allocated; otherwise, the allocator retries with another free-list entry. Note that fast reclaim also relies on the same procedure to ensure consistency. (3) For larger allocations (order > 6), we apply a coarse-grained transaction-based approach: the allocator acquires each order-6 subpage sequentially. If any sub-allocation attempt fails, it then aborts and rolls back the entire request.

2. Scaling host memory guard with base pages via optimized self-hosted remap. INFINIDEFRAG leverages page remapping to bypass the host buddy allocator (Figure 9). A straightforward and non-intrusive way to implement it is to invoke *mremap* [9]. Unfortunately, *mremap* does not scale well with increased threads. As shown in Figure 10, its latency grows sharply with thread count, reaching $\sim$ 270 µs at 16 threads. The poor scalability stems from its need to manipulate multiple virtual memory areas (*i.e., vm_area_struct*) and frequently split the source virtual memory area, which is costly and requires mutual exclusion in concurrent scenarios.

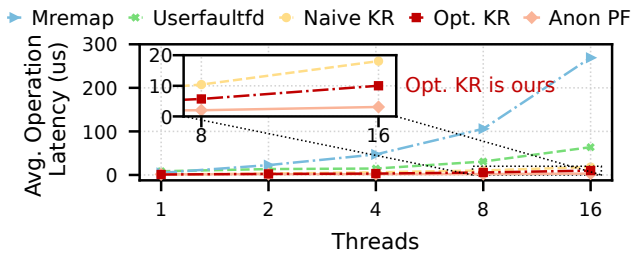


Figure 10: Page remap latency for different techniques. *The host uses base pages, and experiments are conducted under the extreme fragmentation setting (§6.1). By optimizing page remapping—from mremap, to userfaultfd, to naive in-kernel remap (KR), and finally to optimized KR — the page remap overhead is close to that of a regular anonymous page fault (i.e., Anon PF).*

We therefore explore a more efficient remapping mechanism based on *userfaultfd* [11], which is a user-space paging policy and implemented in host-side QEMU/KVM. The key advantage is that it updates only page-table entries and avoids manipulating kernel virtual-memory metadata. As shown in Figure 10, *Userfaultfd* is more scalable than *Mremap*. However, at 16 threads, remapping a single page still consumes 63 µs, which is too slow to meet SLOs. Moreover, kernel–user context switches introduce additional overhead, causing even single-thread remap operations to take up to 9 µs, which is much higher than 1 µs of a regular anonymous page fault.

Given the high kernel–user context-switch overhead of *userfaultfd*, we next introduce a lightweight kernel-based mechanism for page remapping. A naive design is to migrate the source physical page directly to the destination physical page by invoking *move_pages*, as shown in Listing 1. Nevertheless, as the thread count increases, the latency of page remapping remains high (*Naive KR* in Figure 10). Our profiling shows that with eight threads, locking and TLB-flush operations account for a substantial fraction of the remapping cost (§6.7). Each TLB flush is guarded by either the KVM MMU lock or the page-table lock, which are expensive in multi-thread environments [19, 20, 26, 38] because they require *TLB shootdown*. The shootdown further exacerbates lock contention, as all threads spin while attempting to acquire these locks.

We therefore propose ***in-kernel remap with delayed TLB flushes***. Our key insight is that *once the reclaimed GPA pages have been allocated inside the guest OS, they are logically marked as inaccessible* (§4.2). Consequently, there is no need to perform synchronous TLB flush. Instead, we can defer it asynchronously (as they will never be accessed by others), thereby avoiding lock contention. As shown in Figure 10, it scales efficiently across all thread counts, achieving performance close to regular anonymous page faults (*Anon PF*).

3. Scaling host memory guard with huge pages via hybrid paging. When the host backs VM memory using base pages, *self-hosted remap* is fairly efficient. However, when the host allocates VM memory using huge pages, reclaiming free GPA pages may introduce either internal (*i.e.,* lazily freeing fragmented base pages from a host huge page) or external (*i.e.,* directly splitting host huge pages) fragmentation on the host side, particularly in the multi-VM environment.

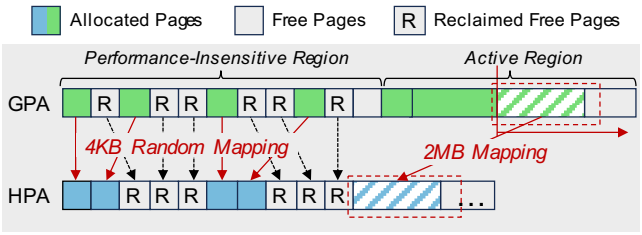


Figure 11: Hybrid paging. *Memory allocated during subsequent expansions is backed by huge pages on the host, whereas the previously allocated fragment pages are mapped using base pages.*

We introduce *hybrid paging* to address this challenge, as shown in Figure 11. Specifically, we first align GPA-HPA mappings with huge pages to maximize system performance and reduce TLB misses. For old fragment pages that are still valid and cannot be reclaimed by *guest reclaimer*, we use base pages to maintain their GPA-HPA mappings. Because these fragmented pages typically reside in performance-uncritical regions (*e.g.,* kernel objects, page cache, and other cold memory areas), we let the host perform background compaction. In contrast, performance-critical pages are allocated from the newly expanded GPA regions and backed by host-huge pages, which benefit from large GPA-HPA mappings. Therefore, *hybrid paging* avoids both host-side and guest-side fragmentation to achieve the near-ideal performance (Figure 23). Note that host-side compaction is necessary when the host uses huge pages, whereas the guest remains compaction-free during defragmentation, consistent with our insight (§3.3).

5 Implementation

INFINIDEFRAG requires coordinated changes to three layers of the virtualization stack: the guest kernel, the host kernel, and QEMU/KVM, with ∼ 7K lines of code (LoC) modified. Nevertheless, since INFINIDEFRAG operates as separate modules at each layer, it is still compatible with existing virtualization architectures, allowing traditional VMs and our modified VMs to run concurrently within the same environment. This section summarizes where each component in §4 is implemented and how the three layers interact.

Guest-side implementation. The *Infinite Address Manager* (§4.2) and the guest-side portion of *Scalability Optimizer* (§4.4-1) are implemented inside the guest kernel and extend the regular higher-order allocation path. Specifically, *Guest Reclaimer* and *Fast Reclaim* are integrated with the guest buddy allocator to identify and reserve free fragment GPA pages, while *Guest Extender* manages GPA-space expansion and memory hotplug. Moreover, the guest also maintains the page bitmap used by *Fast Reclaim* and initializes per-page metadata for newly onlined pages. From the guest's perspective, INFINIDEFRAG remains transparent to applications: it is triggered only when regular huge-page allocation fails.

Host-side implementation. The *Host Memory Guard* (§4.3) and the host-side portion of *Scalability Optimizer* (§4.4-2

Workload	Mem	Details
Gups	9.0 GB	A benchmark that updates in randomly selected locations in large memory region. [1]
Specjbb	10.0 GB	A benchmark that models a business application, stressing with compute, memory and GC. [3]
Redis	20.0 GB	An in-memory key-value store widely used for caching, real-time analytics and serving. [16]
Graph500	4.5 GB	A benchmark that evaluates graph tasks, featuring irregular memory access patterns. [6]
PR	11.0 GB	A benchmark from GAPBS, which computes node importance in large graphs. [8]
BC	11.0 GB	A benchmark from GAPBS, which measures the centrality of nodes. [8]
XSBench	5.8 GB	A Monte Carlo-based proxy application that simulates neutron cross-section lookups. [12]
Liblinear	4.5 GB	A machine learning library commonly used for training support vector machines (SVMs). [14]
CG.d	20.0 GB	A benchmark from the NAS Parallel Benchmarks, solving large, sparse linear systems. [10]
Random	7.8 GB	Issuing randomized read/write operations to produce irregular memory access patterns. [7]

Table 1: Evaluated workloads. *'Mem' is the working set size. These workloads are widely used in prior studies [39, 43, 46].*

and §4.4-3) are implemented in the host kernel. Their responsibilities are to (1) reclaim HPA pages corresponding to freed GPA pages, (2) enforce the VM's memory quota during GPA expansion, and (3) maintain GPA–HPA mappings during page reallocation. The host kernel further implements the optimized remapping path used by *self-hosted remap*, the bitmap-driven *batch unmapping*, and the host-side *hybrid paging*. When the host uses huge pages, it may still perform background compaction to recover host huge pages; However, guest-side compaction is removed from the critical path.

QEMU/KVM integration. QEMU/KVM serves as the coordination point between the guest and host implementations. QEMU provisions the memory-backend device used for reserved GPA expansion, while KVM handles the resulting EPT faults and updates the corresponding second-level mappings. Therefore, the optimized remapping (§4.4-2) and delayed TLB-flush (§4.4-3) are also implemented in the KVM-assisted host path, while QEMU provides the userspace memory backend needed to expose the expanded GPA region.

6 Evaluation

Our evaluation aims to answer the following questions:

- Does INFINIDEFRAG outperform competitors? (§6.2)
- Does INFINIDEFRAG reduce request latency? (§6.3)
- Can INFINIDEFRAG defragment memory efficiently? (§6.4)
- Can INFINIDEFRAG supply sufficient huge pages? (§6.5)
- Can INFINIDEFRAG scale to complex environments? (§6.6)
- Does INFINIDEFRAG introduce additional overhead? (§6.7)
- Does INFINIDEFRAG consume more resources? (§6.8)

6.1 Experimental Setup

Testbed. We conduct experiments on QEMU 8.2.94 with KVM enabled, using a Linux 6.10.0 host kernel. Specifically, (1) The host machine is a dual-socket server, where each socket contains a 28-core Intel Xeon Gold 6330 CPU (2.0 GHz) and 128 GB of DRAM, running Ubuntu 22.04. (2) The guest VM runs Ubuntu 20.04 and is configured with 16 vCPUs and 64 GB of memory. All modifications are applied to both QEMU 8.2.94 and the Linux 6.10.0 kernel.

Competitors. We compare INFINIDEFRAG with several state-of-the-art approaches for mitigating huge-page fragmentation (guest side only): CBMM [43], LLFREE [53], Linux Transparent Huge Pages (Linux THP) under multiple configurations, Linux using only 4 KB base pages (Linux 4 KB), and Linux THP in non-fragment settings (NoFrag). Here, *NoFrag* denotes the non-fragmented baseline under the corresponding host-page configuration used in each figure. CBMM applies adaptive policies to decide when to defragment and form huge pages, while LLFREE introduces a new kernel page allocator optimized for 2 MB page allocation. For Linux THP, we evaluate three variants: the default policy (promotion rate 1.6 MB/s), aggressive policy (16 MB/s, THP Aggr), and synchronous huge-page promotion on page faults (THP Sync).

Workloads and methodology. Table 1 summarizes the workloads used in our experiments. We choose them because (1) their working-set sizes reliably induce and expose fragmentation, and (2) they are widely used in prior studies. Moreover, we evaluate host configurations using both 4 KB base pages (THP is disabled) and 2 MB huge pages (THP is set to *always*). Due to the page limit, we omit (some) detailed analysis of the *host-side huge-page configuration*, as part of the results have already been shown in §3, and these results follow a trend similar to those of host-side base-page configuration.

Fragmentation setup. To emulate guest-side fragmentation, we follow [46] (see details in §8) with additional steps. Specifically, for each guest, we first load large files that exceed the guest's total memory capacity into the page cache, and then randomly access them for 10 minutes to disrupt LRU ordering. We further reclaim a memory region equal to the target workload's working set, triggering page replacement to evict cold pages. Meanwhile, we evaluate three levels of fragmentation.

- ***Extreme fragmentation setting.*** We logically divide the file-backed memory into 2 MB regions and randomly access 256 × 4 KB pages within each region before reclaim. As a result, the reclaimed holes are densely interleaved, so the buddy allocator can coalesce free memory only into order-0 or order-1 chunks (i.e., 4 KB or 8 KB free extents), creating severe fragmentation with almost no high-order pages.
- ***Moderate (50%) fragmentation setting.*** We access only 1–2 random 4 KB pages in each 2 MB region before reclaim. This leaves one or two "pinned" pages in every 2 MB-aligned region, preventing order-9 coalescing while still allowing the buddy allocator to merge the remaining free space into 512 KB or 1 MB blocks. We refer to this setting as *50% fragmentation* because, after reclaim, each huge-page-sized region remains coalescible only up to about half of its original size (i.e., at most an order-8 block, or roughly 1 MB), rather than a full 2 MB order-9 block.

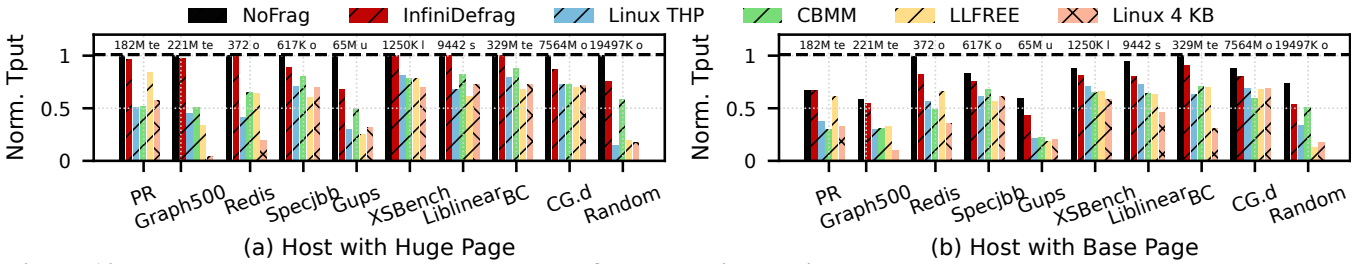


Figure 12: End-to-end throughput under extreme fragmentation setting. *All bars are normalized to* NoFrag *with host-huge configuration, enabling direct comparison between the host-base and host-huge configurations. The dashed line marks this upper bound. Moreover, the numbers above the bars report the throughput of* NoFrag *with host-huge configuration for each workload. The terms te, o, u, l, and s denote Traversed Edges, Operations, Updates, Lookups, and Samples Per Second, respectively.*

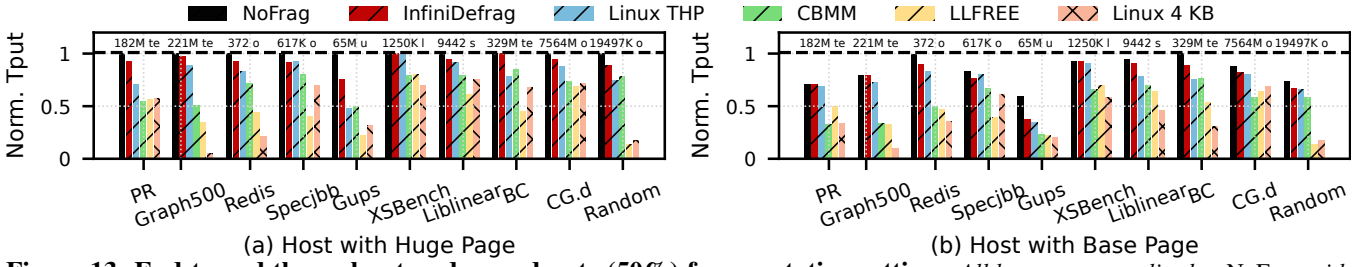


Figure 13: End-to-end throughput under moderate (50%) fragmentation setting. *All bars are normalized to* NoFrag *with host-huge configuration, enabling direct comparison between the host-base and host-huge configurations. The dashed line marks this upper bound. Moreover, the numbers above the bars report the throughput of* NoFrag *with host-huge configuration for each workload.*

- ***Light/no fragmentation setting.*** We skip the fragmentation-injection procedure above to avoid severe fragmentation. Instead, we start from a freshly initialized VM, warm up the workload, and measure after the memory layout reaches a stable state. In this setting, contiguous 2 MB regions remain widely available and all systems can obtain huge pages normally, allowing us to check whether INFINIDEFRAG introduces overhead on the normal allocation path.

Finally, we start the workload with both background compaction and THP promotion enabled. Unless otherwise stated, the main results use the extreme fragmentation setting. Note that competitors in the anti-fragmentation category (i.e., CBMM and LLFREE) are expected to mitigate the fragmentation problem during this procedure, enabling subsequent workloads to acquire more huge pages during execution.

6.2 Performance Improvement

Evaluation setup. In this section, we compare the end-to-end throughput of INFINIDEFRAG under extreme, moderate, and light/no-fragmentation settings, with all the workloads summarized in Table 1, and the host with huge/base page configurations. Moreover, all results are normalized to *NoFrag* with the host-huge configuration. This unified normalization enables direct comparison between the host-base and host-huge configurations. Note that for INFINIDEFRAG, we enable *self-hosted remap* (§4.3-2 and §4.4-2) when the host uses base pages, and *batch unmapping* (§4.3-3) plus *hybrid paging* (§4.4-3) when the host uses huge pages, separately.

Throughput under extreme fragmentation. The results in Figure 12 show that INFINIDEFRAG consistently achieves the highest throughput under all the configurations. Specifically, when the host uses huge pages, INFINIDEFRAG approaches the upper bound in most workloads, indicating that it preserves most of the benefit of 2 MB guest and host mappings. When the host uses base pages, INFINIDEFRAG still outperforms all baselines, but a visible gap to the upper bound remains. This gap reflects the cost of shattered EPT mappings: although INFINIDEFRAG preserves 2 MB guest pages, scattered 4 KB host pages may prevent the host from using 2 MB EPT entries, requiring up to 512 EPT entries for a 2 MB guest region. However, even with this penalty, INFINIDEFRAG still reliably supplies 2 MB huge pages on demand and avoids the wasted compaction overhead incurred by *Linux THP*.

Throughput under moderate (50%) fragmentation. Figure 13 reports the moderate-fragmentation results under both host-huge/base configurations, which suggests that INFINIDEFRAG remains the best-performing approach in both of the two configurations. However, the throughput gap over Linux THP becomes smaller than that in Figure 12. The reason is that moderate fragmentation still leaves sizable contiguous regions (e.g., 512 KB and 1 MB blocks), making huge-page recovery easier for the baseline and therefore reducing the benefit of compaction-free defragmentation. Moreover, compared to the extreme fragmentation setting, INFINIDEFRAG with moderate fragmentation remains closer to the upper bound (i.e., NoFrag with host-huge pages) under the host-huge configuration, whereas the host-base configuration still exhibits a moderate gap due to the lack of 2 MB EPT mappings.

Throughput under light/no-fragmentation. In this setting, since all systems can obtain host-side huge pages normally and easily, all approaches achieve similar performance.

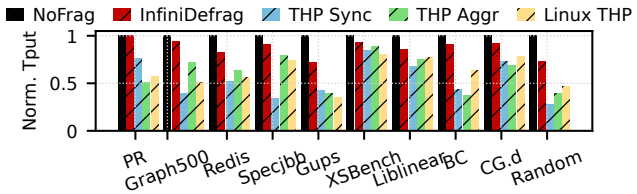


Figure 14: Throughput of different THP settings. *The host uses base pages, and experiments are conducted under extreme fragmentation. Note that other configurations have a similar trend.*

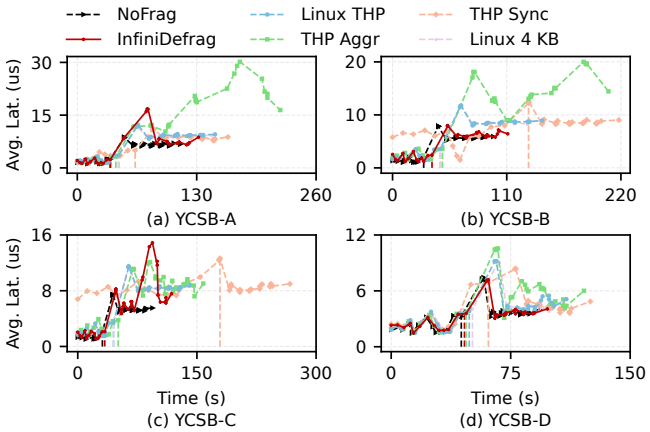


Figure 15: Latency analysis under YCSB-Redis. *We first build the database (*i.e.*, the load phase), and the vertical dashed line marks the transition to the run phase. We omit LLFREE and CBMM because their results are similar to those of Linux 4 KB. Moreover, the host uses base pages, experiments are conducted under extreme fragmentation, and other configurations have a similar trend.*

Therefore, we do not report the results under the light/no-fragmentation setting. Specifically, when fragmentation is mild, INFINIDEFRAG rarely needs to expand the GPA space, while Linux THP can recover huge pages through direct compaction with little overhead. As a result, the benefit of compaction-free defragmentation becomes small.

Throughput of different THP settings. As shown in Figure 14, INFINIDEFRAG consistently outperforms all variants of *Linux THP* across evaluated workloads. Notably, there is no single THP configuration that achieves optimal performance for every workload; both conservative (*i.e., Linux Sync*) and aggressive (*i.e., Linux Aggr*) strategies can introduce performance regressions. The reasons are further analyzed in §6.7.

6.3 Latency Analysis

Figure 15 shows the real-time latency of the YCSB workloads during both the load and run phases. We report only the results for the host using base pages under the extreme fragmentation setting, while other configurations exhibit a similar trend. The results suggest that INFINIDEFRAG consistently exhibits the lowest latency, whereas all Linux THP variants (*i.e., Linux THP*, *THP Aggr*, and *THP Sync*) exhibit poor latency and can even underperform the 4KB-only baseline (*i.e., Linux 4 KB*). Interestingly, *THP Sync* and *THP Aggr* hit their latency bottlenecks at different stages: *THP Sync* shows substantially higher latency during the load phase, while *THP Aggr* suffers from severe latency spikes during the run phase. The reason is that most memory allocations occur in the load phase; *THP Sync* triggers synchronous compaction on the allocation path, introducing high allocation latency. By contrast, *THP Aggr* performs system-wide scanning and page migrations in the run phase, heavily interfering with the YCSB-Redis workload.

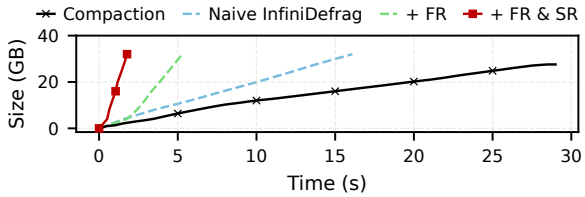


Figure 16: Defragmentation efficiency analysis. *Experiments are conducted without foreground applications. "FR" refers to fast reclaim (§4.2), and "SR" refers to self-hosted remap (§4.3). The host uses base pages, experiments are conducted under extreme fragmentation, and other configurations have a similar trend.*

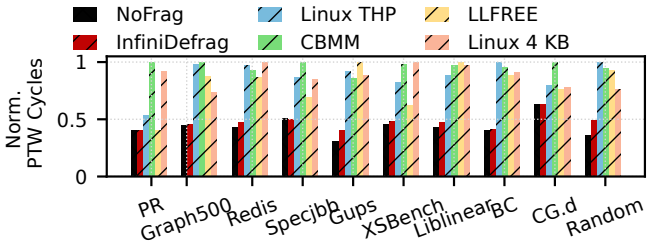


Figure 17: Comparison of page table walk cycles. *The host uses base pages, and experiments are conducted under extreme fragmentation. Although using huge pages only at the guest cannot reduce TLB miss rate, it still significantly reduces PTW cycles.*

6.4 Defragmentation Efficiency

This section measures the raw bandwidth of different defragmentation mechanisms, where bandwidth is defined as the amount of contiguous physical memory generated per second. Similarly, we report only the results for the host using base pages under the extreme fragmentation setting, while other configurations exhibit a similar trend. Moreover, we exclude those anti-fragmentation approaches (*i.e., LLFREE* and *CBMM*) because they cannot produce contiguous memory. The results are shown in Figure 16, *Compaction* (*i.e., Linux THP*) achieves 0.91 GB/s, while INFINIDEFRAG + *FR & SR* reaches nearly 20 GB/s (*i.e.,* about 19× improvement). These results demonstrate that INFINIDEFRAG can defragment memory with substantially higher efficiency.

6.5 Huge Page Analysis

Page table walk cycles. The workloads we evaluate (Table 1) exhibit highly random memory access patterns, which lead to frequent TLB misses and expensive page table walks. INFINIDEFRAG outperforms other approaches primarily because it significantly reduces the cycles spent on page table walks. Figure 17 compares PTW cycles across all approaches and, together with Figure 12, demonstrates a strong correlation between reduced PTW cycles and improved performance. The remaining gap between INFINIDEFRAG with host base

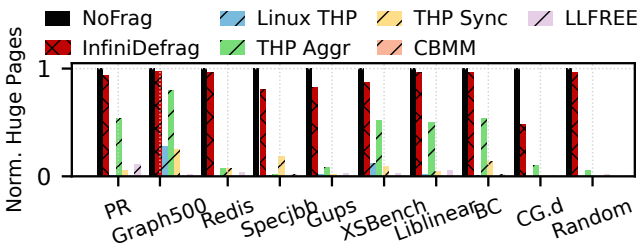


Figure 18: Total allocated huge pages. *The host uses base pages, and experiments are conducted under extreme fragmentation. CBMM/LLFREE cannot allocate any huge pages in many workloads.*

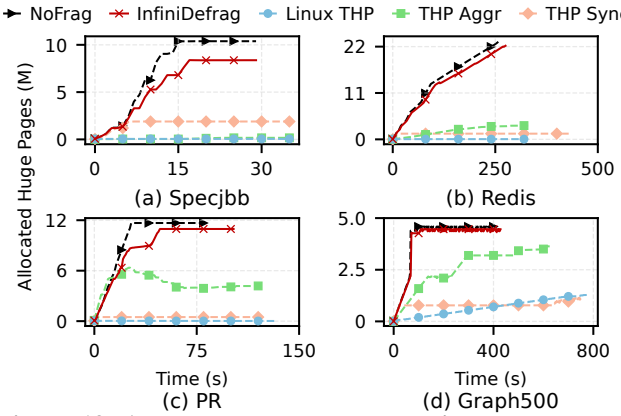


Figure 19: Allocated huge pages over time. *The host uses base pages, and experiments are conducted under extreme fragmentation. We do not evaluate CBMM and LLFREE because they cannot allocate any huge pages in many workloads (i.e., Figure 18).*

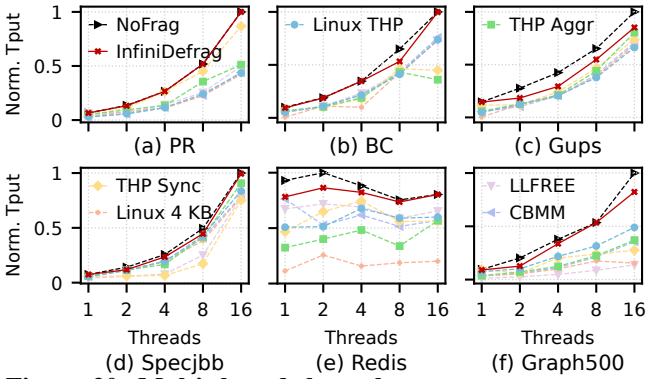


Figure 20: Multi-thread throughput. *The host uses base pages, and experiments are conducted under the extreme fragmentation setting. Note that the Redis server is a single-thread model; therefore, increasing thread counts does not improve throughput and may even degrade performance due to lock contention.*

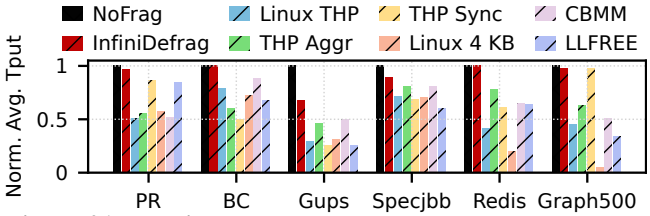


Figure 21: Multi-VM throughput. *The host uses huge pages, and experiments are conducted under extreme fragmentation. Each bar reports the average throughput. We do not evaluate the host with base pages because its performance is similar to the single-VM case (i.e., without host-side fragmentation and inter-VM interference).*

pages and the upper bound in Figure 12 is consistent with this result, as host base pages prevent the use of 2 MB EPT mappings and thus increase nested page-walk overhead.

Allocated huge pages. By mapping larger memory regions with each TLB entry, huge pages substantially reduce TLB pressure and the number of memory accesses required during a TLB miss. This reduction in PTW cycles makes huge pages a critical optimization for efficient address translation.

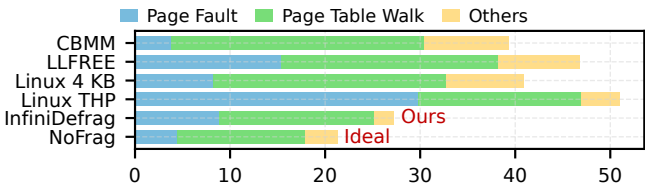
Figure 18 shows the total allocated huge pages. Under severe fragmentation, both anti-fragmentation (*i.e., CBMM* and *LLFREE*) and defragmentation (*i.e., Linux THP*, *THP Aggr*, and *THP Sync*) techniques allocate far fewer huge pages than the total working set size. In contrast, INFINIDEFRAG nearly reaches this upper bound, with the small remaining gap arising from its fallback to base-page allocation when memory trade (§4.2-1) is performed asynchronously. Although *THP Aggr* can increase the number of allocated huge pages, they still suffer from substantial throughput degradation (Figure 14) due to the high overhead of memory compaction (§6.7).

We further analyze the temporal behavior of huge page allocations across the workload's execution, as shown in Figure 19. Most workloads allocate physical memory gradually rather than all at startup, consistent with the OS's lazy allocation policy. Consequently, memory allocation continues throughout the workload's lifetime, making these workloads highly sensitive to the memory compaction overhead. Because memory compaction lies on the critical path of the page fault handler—*i.e.,* during physical page allocation—it can introduce substantial runtime overhead.

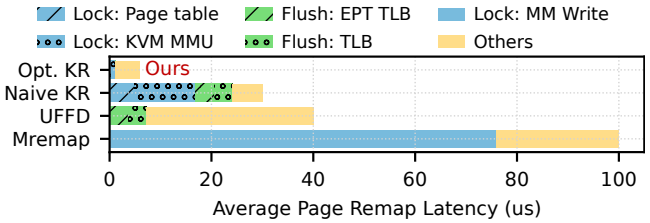
6.6 Scalability Analysis

Throughput under multi-threads. As shown in Figure 20, INFINIDEFRAG consistently outperforms competitors at all concurrency levels, demonstrating the effectiveness of *optimized fast reclaim* (§4.4-1) and *optimized self-hosted remap* (§4.4-2). Specifically, *THP Sync* fails to scale for Redis and Graph500 because synchronous compaction in the page-fault path becomes a bottleneck at high concurrency. *Linux THP* also fails to scale, as its background compaction depends on expensive page migrations, including TLB shootdowns. Note that the huge page configuration (on the host) and the moderate (50%) fragmentation setting exhibit a similar trend, and we evaluate only six workloads due to the page limit.

Throughput under multi-VMs. Figure 21 shows the average performance of three VMs, where INFINIDEFRAG outperforms competitors in most cases, demonstrating the effectiveness of *optimized fast reclaim* (§4.4-1) and *hybrid paging* (§4.4-3). Note that the multi-VM results are similar to those of the single-VM results (*i.e.,* Figure 12), indicating that there is no noticeable performance interference between VMs. Moreover, we omit the configuration where the host uses base pages or suffers from moderate (50%) fragmentation, because these



(a) Time breakdown of Gups.



(b) Time breakdown of remap under 8-thread Gups.

Figure 22: Overhead analysis of INFINIDEFRAG. *The host uses base pages, and experiments are conducted under the extreme fragmentation setting. (a) Page Fault includes memory compaction or fast reclaim. (b) Opt. KR is our optimized in-kernel remap, Naive KR is the in-kernel remap without deferred TLB flush, UFFD is userfaultfd, and Mremap is the mremap syscall (§4.4-2).*

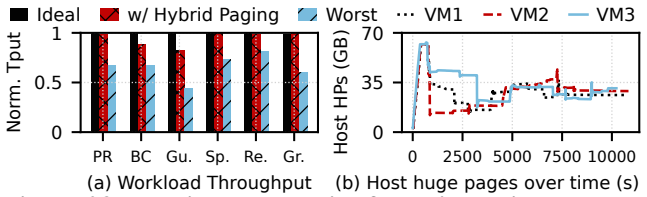
settings do not introduce meaningful host-side fragmentation or inter-VM interference; therefore, their overall performance is expected to be similar to the single-VM case.

6.7 Overhead Analysis

Infinite address manager. We first analyze *Gups* to evaluate whether *infinite address manager* introduces additional overhead, as shown in Figure 22a. The results show that INFINIDEFRAG reduces the time spent in the page-fault path and achieves near-optimal performance thanks to *optimized fast reclaim* (§4.4-1). In contrast, *Linux THP* reduces PTW cycles but spends considerable time in memory compaction (*i.e.,* increased *Page Fault*). As a result, the runtime is even longer than that of *Linux 4 KB*. Note that the host-side *Page Fault* introduced in Figure 22a is different from the guest-side anonymous page fault introduced in Figure 10.

Host memory guard with base pages. We further evaluate whether the optimized kernel remap (*i.e., Opt. KR*, §4.4-2) improves the efficiency of *self-hosted remap* and thereby minimizes the overhead of *host memory guard* when the host uses base pages. The results in Figure 22b indicate that, by safely deferring TLB flushes, *Opt. KR* eliminates nearly all lock contention overheads (*i.e.,* only reserve *Lock: KVM MMU*), thus achieving the best performance with minimal overheads.

Host memory guard with huge pages. We next evaluate whether *hybrid paging* mitigates host-side fragmentation and thereby reduces the overhead of *Host Memory Guard* when the host uses huge pages (§4.4-3). As shown in Figure 23a, *w/ Hybrid Paging* achieves performance close to *Ideal*, which represents the case where the host has sufficient free memory to provide huge pages. This prevents INFINIDEFRAG from degrading to *Worst*, which occurs when the host becomes fully fragmented and cannot provide any huge pages.



(a) Workload Throughput (b) Host huge pages over time (s)

Figure 23: Multi-VM analysis of hybrid paging. *The host uses huge pages (i.e., only this configuration has hybrid paging), and experiments are conducted under the extreme fragmentation setting. Moreover, (a) Gu., Sp. Re., and Gr. represent Gups, Specjbb, Redis, and Graph500. Ideal and Worst denote Linux THP running atop the host using all huge and base pages. (b) Each VM sequentially executes the six workloads listed in subfigure (a).*

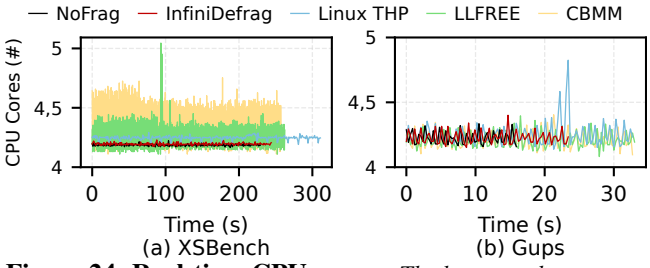


Figure 24: Real-time CPU usage. *The host uses base pages, and experiments are conducted under extreme fragmentation.*

The small remaining gap observed in *BC* and *Gups* workloads arises from two aspects: (1) reclaiming fragmented pages may require splitting nearby huge pages into 4 KB mappings; and (2) the host performs asynchronous compaction on these 4 KB pages, introducing background overhead. Because most of these 4 KB pages reside in performance-insensitive regions, their overheads have only a minor impact.

Figure 23b shows that each VM requires only 30–40 GB of huge pages. This is because host-side huge pages are primarily applied to the expanded GPA region, which largely covers workloads' working set. In addition, VMs finish with a similar time, indicating a small interference between them.

6.8 Consumed Resources

CPU usage. Figure 24 shows that INFINIDEFRAG consumes fewer CPU cores with negligible fluctuations than competitors. Under fragmented memory, Linux THP incurs compaction overhead from page faults and the background daemon. Moreover, *LLFREE* and *CBMM* consume more CPU cores to carefully control page placement decisions and preserve large contiguous regions for each allocation. In contrast, INFINIDEFRAG eliminates compaction and introduces only negligible CPU overhead when expanding contiguous memory.

Memory usage. The *fast reclaim* (§4.2-2) introduces a page bitmap that tracks page states using only 2 bits per page (one bit for free/allocation state and one bit for online/offline state). In addition, as the guest physical address space grows, the memory footprint of per-page metadata increases proportionally: each 4 KB page requires 64 bytes of metadata, amounting to roughly 1.6% of its size. Moreover, the memory overheads of all other kernel data structures are negligible.

7 Discussion

Host-side huge pages vs base pages. Using huge pages on the host reduces TLB pressure and page-table footprint, thereby improving VM performance. However, huge pages limit fine-grained control: they complicate memory reclamation, increase internal fragmentation, and disable page-level deduplication mechanisms such as KSM [27, 39, 55]. In contrast, base pages provide greater flexibility for reclamation, compaction, and deduplication, but incur higher TLB overhead. Overall, host-side huge pages deliver high performance, whereas host-side base pages offer high flexibility, and INFINIDEFRAG performs well under both configurations.

Giant page support. 1 GB giant pages provide substantial benefits for memory-intensive workloads [50]. However, Linux currently lacks transparent support for 1GB THP, and memory fragmentation makes such allocations nearly impossible in practice [50, 56]. In contrast, INFINIDEFRAG can efficiently construct large contiguous physical regions, making 1 GB page allocation far more feasible.

Unbounded GPA space expansion. We currently apply no specialized optimizations to the unbounded GPA expansion. Therefore, there are two possible problems: (1) page-metadata memory grows linearly with GPA size; one possible optimization is to recycle metadata from reclaimed pages to avoid new allocations during future expansions. (2) unrestricted expansion could eventually exhaust the 57-bit physical address space, although this would take more than a century (§3.3). A potential optimization is to retrieve earlier fragmented pages from the host while returning the expanded contiguous regions. We leave these optimizations to our future work.

Page cache reclamation. The page cache is a major contributor to guest-side memory fragmentation [42]. Therefore, we plan to leverage our *fast reclaim* mechanism (§4.2-2) to reclaim fragmented page-cache pages. By integrating page-cache reclamation into the defragmentation pipeline, we expect to further improve the availability of contiguous memory and enhance the overall effectiveness of INFINIDEFRAG.

8 Related Work

Object-level and allocator-level defragmentation. Object-based defragmentation techniques classify objects by predicted lifetimes and colocate those with similar lifetimes [24, 31, 40, 41]. Recent allocator-level techniques such as TCMalloc [31] are also hugepage-aware and improve huge-page utilization in production by steering allocations into already-backed huge pages and adaptively releasing unused subregions. Because the kernel ultimately manages memory at page granularity, these approaches operate primarily in user space and do not directly address guest physical memory fragmentation inside a VM. In contrast, INFINIDEFRAG operates at the guest-physical level and is orthogonal to such methods; the two could be combined to further improve performance.

Page-level defragmentation. Illuminator [48] targets permanent fragmentation caused by unmovable pages but does not optimize regular memory compaction, making it complementary to our approach. LLFREE [53] revisits page-frame allocation from the kernel level and provides a scalable allocator with low metadata overhead and built-in anti-fragmentation behavior for huge pages. However, it requires replacing the Linux buddy allocator, making it less compatible with existing kernel features such as compaction and NUMA balancing. Moreover, under the highly fragmented VM settings (*e.g.,* the *extreme fragmentation* setting in §6.1), its anti-fragmentation benefits become limited and it underperforms INFINIDEFRAG in end-to-end throughput (*i.e.,* Figure 12 and Figure 13).

NUMA-aware defragmentation. vMitosis [46] addresses remote page-table accesses in virtualized NUMA systems by improving translation locality, either by migrating page tables to the local node or replicating them across sockets. We do not compare against it because the two works are orthogonal: vMitosis focuses on NUMA-aware page-table placement, whereas our work targets memory defragmentation. By the way, our proposed *memory trade* (§4.2-1) can achieve NUMA-aware support by implementing at a per-NUMA-node granularity, ensuring that the GPA space expansion allocates and reclaims pages only from the corresponding NUMA node.

Architectural support for defragmentation. Some prior works [25, 49, 56] address huge-page allocation under fragmentation by introducing architectural-level optimizations, which require specialized hardware support. For example, Contiguitas [56] targets host-side physical memory contiguity. It separates movable and unmovable allocations and uses hardware support to migrate unmovable pages, thereby increasing the availability of contiguous HPA. In contrast, INFINIDEFRAG provides contiguous GPA without guest-side compaction. The two approaches are therefore complementary: a host running Contiguitas could provide more contiguous HPA memory, allowing INFINIDEFRAG's host-huge page mode and *hybrid paging* to use more 2 MB-2 MB mappings.

9 Conclusion

We propose INFINIDEFRAG, a novel defragmentation technique for virtualized environments that efficiently generates huge pages. The key insight is to treat the guest physical address space as (nearly) infinite by controlling the mapping between guest and host physical addresses, thereby eliminating the need for costly memory compaction. Extensive experiments demonstrate that INFINIDEFRAG outperforms state-of-the-art approaches, achieving near-optimal performance comparable to non-fragmented memory conditions.

Acknowledgment

We sincerely thank our shepherd, Emmett Witchel, and reviewers for their constructive suggestions. This research was partly supported by the National Natural Science Foundation of China under Grant 62472127, Shenzhen Science and Technology Program under Grant RCYX20210609104510007.

References

[1] Gups: Hpcc randomaccess benchmark. https://github.com/alexandermerritt/gups, 2006.

[2] Memory compaction. https://lwn.net/Articles/368869/, 2010.

[3] Specjbb® 2015. https://www.spec.org/jbb2015/, 2015.

[4] Five-level page tables. https://lwn.net/Articles/717293/, 2017.

[5] Intel 5-level paging. https://cdrdv2-public.intel.com/671442/5-level-paging-white-paper.pdf, 2017.

[6] Graph 500: large-scale benchmarks. https://graph500.org/, 2019.

[7] Randomaccess. https://github.com/csl-iisc/nuKSM-pact21-artifact/tree/main/benchmarks/randomAccess, 2021.

[8] Gap benchmark suite. http://gap.cs.berkeley.edu/benchmark.html, 2024.

[9] mremap(2) — linux manual page. https://man7.org/linux/man-pages/man2/mremap.2.html, 2024.

[10] Nas parallel benchmarks. https://www.nas.nasa.gov/software/npb.html, 2024.

[11] userfaultfd(2) — linux manual page. https://man7.org/linux/man-pages/man2/userfaultfd.2.html, 2024.

[12] Xsbench: The monte carlo macroscopic cross section lookup benchmark. https://github.com/ANL-CESAR/XSBench, 2024.

[13] Hugetlb pages. https://docs.kernel.org/admin-guide/mm/hugetlbpage.html, 2025.

[14] Liblinear – a library for large linear classification. https://www.csie.ntu.edu.tw/~cjlin/liblinear/, 2025.

[15] Physical page allocation. https://www.kernel.org/doc/gorman/html/understand/understand009.html, 2025.

[16] Redis - the real-time data platform. https://redis.io/, 2025.

[17] Transparent hugepage support. https://docs.kernel.org/admin-guide/mm/transhuge.html, 2025.

[18] Chloe Alverti, Stratos Psomadakis, Vasileios Karakostas, Jayneel Gandhi, Konstantinos Nikas, Georgios Goumas, and Nectarios Koziris. Enhancing and exploiting contiguity for fast memory virtualization. In *Proceedings of the 47th Annual International Symposium on Computer Architecture (ISCA)*, pages 515–528, 2020.

[19] Nadav Amit. Optimizing the TLB shootdown algorithm with page access tracking. In *Proceedings of the 2017 USENIX Annual Technical Conference (USENIX ATC)*, pages 27–39, 2017.

[20] Nadav Amit, Amy Tai, and Michael Wei. Don't shoot down TLB shootdowns! In *Proceedings of the 15th European Conference on Computer Systems (EuroSys)*, pages 1–14, 2020.

[21] Arkaprava Basu, Jayneel Gandhi, Jichuan Chang, Mark D Hill, and Michael M Swift. Efficient virtual memory for big memory servers. *ACM SIGARCH Computer Architecture News*, 41(3):237–248, 2013.

[22] Christopher Clark, Keir Fraser, Steven Hand, Jacob Gorm Hansen, Eric Jul, Christian Limpach, Ian Pratt, and Andrew Warfield. Live migration of virtual machines. In *Proceedings of the 2nd Symposium on Networked Systems Design and Implementation (NSDI)*, 2005.

[23] Brian F Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with YCSB. In *Proceedings of the 1st ACM symposium on Cloud computing (SoCC)*, pages 143–154, 2010.

[24] Tamar Domani, Elliot K. Kolodner, and Erez Petrank. A generational on-the-fly garbage collector for Java. In *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation (PLDI)*, pages 274–284, 2000.

[25] Yu Du, Miao Zhou, Bruce R Childers, Daniel Mossé, and Rami Melhem. Supporting superpages in non-contiguous physical memory. In *Proceedings of the 21st International Symposium on High Performance Computer Architecture (HPCA)*, pages 223–234. IEEE, 2015.

[26] Bin Gao, Qingxuan Kang, Hao-Wei Tee, Kyle Timothy Ng Chu, Alireza Sanaee, and Djordje Jevdjic. Scalable and effective page-table and TLB management on NUMA systems. In *Proceedings of the 2024 USENIX Annual Technical Conference (USENIX ATC)*, pages 445–461, 2024.

[27] Fan Guo, Yongkun Li, Yinlong Xu, Song Jiang, and John CS Lui. SmartMD: A high performance deduplication engine with mixed pages. In *Proceedings of the*

2017 USENIX Annual Technical Conference (USENIX ATC), pages 733–744, 2017.

[28] Alexander Halbuer, Illia Ostapyshyn, Lukas Steiner, Lars Wrenger, Matthias Jung, Christian Dietrich, and Daniel Lohmann. The New Costs of Physical Memory Fragmentation. In *Proceedings of the 2nd Workshop on Disruptive Memory Systems (DIMES)*, pages 33–40, 2024.

[29] Jingyuan Hu, Xiaokuang Bai, Sai Sha, Yingwei Luo, Xiaolin Wang, and Zhenlin Wang. HUB: Hugepage ballooning in kernel-based virtual machines. In *Proceedings of the International Symposium on Memory Systems (MEMSYS)*, pages 31–37, 2018.

[30] Wenjin Hu, Andrew Hicks, Long Zhang, Eli M. Dow, Vinay Soni, Hao Jiang, Ronny L. Bull, and Jeanna N. Matthews. A quantitative study of virtual machine live migration. In *ACM Cloud and Autonomic Computing Conference (CAC)*, pages 11:1–11:10, 2013.

[31] Andrew Hamilton Hunter, Chris Kennelly, Paul Turner, Darryl Gove, Tipp Moseley, and Parthasarathy Ranganathan. Beyond malloc efficiency to fleet efficiency: a hugepage-aware memory allocator. In *Proceedings of the 15th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 257–273, 2021.

[32] Nancy Jain and Sakshi Choudhary. Overview of virtualization in cloud computing. In *Proceedings of the 2016 Symposium on Colossal Data Analysis and Networking (CDAN)*, pages 1–4, 2016.

[33] Weiwei Jia, Jiyuan Zhang, Jianchen Shan, and Xiaoning Ding. Making dynamic page coalescing effective on virtualized clouds. In *Proceedings of the 18th European Conference on Computer Systems (EuroSys)*, pages 298–313, 2023.

[34] Sudarsun Kannan, Yujie Ren, and Abhishek Bhattacharjee. KLOCs: kernel-level object contexts for heterogeneous memory systems. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 65–78, 2021.

[35] Vasileios Karakostas, Jayneel Gandhi, Furkan Ayar, Adrián Cristal, Mark D Hill, Kathryn S McKinley, Mario Nemirovsky, Michael M Swift, and Osman Ünsal. Redundant memory mappings for fast access to large memories. *ACM SIGARCH Computer Architecture News*, 43(3S):66–78, 2015.

[36] Vasileios Karakostas, Jayneel Gandhi, Adrián Cristal, Mark D Hill, Kathryn S McKinley, Mario Nemirovsky, Michael M Swift, and Osman S Unsal. Energy-efficient address translation. In *Proceedings of the 2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 631–643, 2016.

[37] Sang-Hoon Kim, Sejun Kwon, Jin-Soo Kim, and Jinkyu Jeong. Controlling physical memory fragmentation in mobile systems. *ACM SIGPLAN Notices*, 50(11):1–14, 2015.

[38] Mohan Kumar Kumar, Steffen Maass, Sanidhya Kashyap, Ján Veselỳ, Zi Yan, Taesoo Kim, Abhishek Bhattacharjee, and Tushar Krishna. Latr: Lazy translation coherence. In *Proceedings of the 23rd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 651–664, 2018.

[39] Youngjin Kwon, Hangchen Yu, Simon Peter, Christopher J Rossbach, and Emmett Witchel. Coordinated and efficient huge page management with ingens. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 705–721, 2016.

[40] Martin Maas, David G Andersen, Michael Isard, Mohammad Mahdi Javanmard, Kathryn S McKinley, and Colin Raffel. Learning-based memory allocation for C++ server workloads. In *Proceedings of the 25th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 541–556, 2020.

[41] Martin Maas, Chris Kennelly, Khanh Nguyen, Darryl Gove, Kathryn S McKinley, and Paul Turner. Adaptive huge-page subrelease for non-moving memory allocators in warehouse-scale computers. In *Proceedings of the 2021 ACM SIGPLAN International Symposium on Memory Management (ISMM)*, pages 28–38, 2021.

[42] Mark Mansi and Michael M Swift. Characterizing physical memory fragmentation. *arXiv preprint arXiv:2401.03523*, 2024.

[43] Mark Mansi, Bijan Tabatabai, and Michael M Swift. CBMM: Financial Advice for Kernel Memory Managers. In *Proceedings of the 2022 USENIX Annual Technical Conference (USENIX ATC)*, pages 593–608, 2022.

[44] Timothy Merrifield and H Reza Taheri. Performance implications of extended page tables on virtualized x86 processors. In *Proceedings of the 12th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE)*, pages 25–35, 2016.

[45] Theodore Michailidis, Alex Delis, and Mema Roussopoulos. Mega: Overcoming traditional problems with

os huge page management. In *Proceedings of the 12th ACM International Conference on Systems and Storage (SYSTOR)*, pages 121–131, 2019.

[46] Ashish Panwar, Reto Achermann, Arkaprava Basu, Abhishek Bhattacharjee, K Gopinath, and Jayneel Gandhi. Fast local page-tables for virtualized numa servers with vmitosis. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 194–210, 2021.

[47] Ashish Panwar, Sorav Bansal, and K Gopinath. Hawkeye: Efficient fine-grained os support for huge pages. In *Proceedings of the 24th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 347–360, 2019.

[48] Ashish Panwar, Aravinda Prasad, and K Gopinath. Making huge pages actually useful. In *Proceedings of the 23th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 679–692, 2018.

[49] Chang Hyun Park, Sanghoon Cha, Bokyeong Kim, Youngjin Kwon, David Black-Schaffer, and Jaehyuk Huh. Perforated page: Supporting fragmented memory allocation for large pages. In *Proceedings of the 47th Annual International Symposium on Computer Architecture (ISCA)*, pages 913–925. IEEE, 2020.

[50] Venkat Sri Sai Ram, Ashish Panwar, and Arkaprava Basu. Trident: Harnessing architectural resources for all page sizes in x86 processors. In *Proceedings of the 54th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 1106–1120, 2021.

[51] Rusty Russell. virtio: towards a de-facto standard for virtual I/O devices. *ACM SIGOPS Operating Systems Review*, 42(5):95–103, 2008.

[52] Lars Wrenger, Kenny Albes, Marco Wurps, Christian Dietrich, and Daniel Lohmann. HyperAlloc: Efficient VM Memory De/Inflation via Hypervisor-Shared Page-Frame Allocators. In *Proceedings of the 20th European Conference on Computer Systems (EuroSys)*, pages 702–719, 2025.

[53] Lars Wrenger, Florian Rommel, Alexander Halbuer, Christian Dietrich, and Daniel Lohmann. LLFree: Scalable and Optionally-Persistent Page-Frame Allocation. In *Proceedings of the 2023 USENIX Annual Technical Conference (USENIX ATC)*, pages 897–914, 2023.

[54] Yuping Xing and Yongzhao Zhan. Virtualization and cloud computing. In *Future Wireless Networks and Information Systems*, pages 305–312. 2012.

[55] Zhehua Zhang, Suzhen Wu, Wenyan You, Chunfeng Du, and Bo Mao. Gemina: A Coordinated and High-Performance Memory Deduplication Engine. In *Proceedings of the 2025 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 1603–1617, 2025.

[56] Kaiyang Zhao, Kaiwen Xue, Ziqi Wang, Dan Schatzberg, Leon Yang, Antonis Manousis, Johannes Weiner, Rik Van Riel, Bikash Sharma, Chunqiang Tang, et al. Contiguitas: The pursuit of physical memory contiguity in datacenters. In *Proceedings of the 50th Annual International Symposium on Computer Architecture (ISCA)*, pages 1–15, 2023.

[57] Weixi Zhu, Alan L Cox, and Scott Rixner. A comprehensive analysis of superpage management mechanisms and policies. In *Proceedings of the 2020 USENIX Annual Technical Conference (USENIX ATC)*, pages 829–842, 2020.

ARTIFACT EVALUATED
USENIX
AVAILABLE

Inside Out: A Paradigm Shift In Live VM Introspection

Dufy Teguia[1,2], Louis Duval[1], Teo Pisenti[3], Kahina Lazri[2],
Daniel Hagimont[3], Thomas Pasquier[4], Renaud Lachaize[1], Alain Tchana[1]

[1]*Univ. Grenoble Alpes, CNRS, Inria, Grenoble INP, LIG, 38000 Grenoble, France*
[2]*Orange Research, France*
[3]*IRIT, Université de Toulouse, CNRS, Toulouse INP, UT3 Toulouse, France*
[4]*University of British Columbia, Vancouver, British Columbia, Canada*

Abstract

We present GOODKIT, a new framework for live virtual machine introspection (LVMI) designed for performance, scalability, and safe integration in modern cloud environments. Unlike existing approaches—such as LibVMI—which rely on heavy VM pausing, GOODKIT executes observers as lightweight VMs colocated with the VMM, enabling native-speed access to the target state while preserving strong isolation. GOODKIT introduces fine-grained, lock-aware memory–coherence mechanisms, a configurable probing subsystem for I/O and kernel-level events, and a mutualization layer that allows multiple observers to operate concurrently without degrading target performance. Across 21 real world use cases, including rootkit detection, ransomware monitoring, and scheduler introspection, GOODKIT delivers high performance (compared to LibVMI), strong isolation, and broad applicability.

1 Introduction

Virtual machines (VMs) are the basic abstraction for modern cloud computing. Tenants and providers must routinely leverage *VM introspection (VMI)* techniques to observe VM behavior, in order to detect attacks, maintain availability, and understand performance problems. A VM observer should perform the above tasks while remaining safe for the platform, and also (in many use cases) transparent to the guest. This requires three properties: *low overhead* to track frequent events, *strong isolation* from both the guest and the cloud provider, and the ability to *bridge the semantic gap* between low-level VM state and high-level system abstractions. Our goal is to design a *live VMI (LVMI)* framework that achieves all three properties in a multi-tenant (IaaS) cloud, and that can be leveraged by tenants without constraints imposed by the cloud provider.

Existing VMI systems place observers in three locations: in a separate VM, inside the hypervisor or VMM, or inside the target VM's guest OS. Hypervisor-based systems [13, 22, 23, 25, 41, 48, 50, 52] such as LibVMI [8] run the observer in a separate VM and improve isolation from the guest, but they require hypervisor changes, often pause the VM for a consistent view, and incur extra kernel to userspace crossings. Observers embedded in the VMM [24, 41] have direct access to guest memory and VMM events, but they enlarge the trusted computing base (TCB). Besides, this approach does not provide isolation guarantees between the different observers (deployed by tenants and/or the cloud provider) attached to a target VM. In-guest observers [7, 18, 32] offer native performance and no semantic gap, but a compromised guest can disable or mislead them, and they share its fault domain. No existing design provides live introspection that is fast, strongly isolated from both guest and provider, and deployable without modifying the hypervisor.

We present GOODKIT, a LVMI framework that takes a different approach. It runs one or several observers as regular guest VMs *under the same VMM process as their target VM* and maps selected regions of the target's memory into these observers under explicit policy. From an observer's perspective, introspection looks like native memory access, while still benefiting from VM-level isolation and standard hypervisor accounting. All introspection mechanisms live in user space inside the VMM and in the observer VMs. Moreover, GOODKIT does not require modifying the hypervisor (enabling quick adoption by cloud providers) nor the target guest, and is agnostic to the observers' guest OSes. To support multiple observers efficiently and safely, GOODKIT combines fine-grained access control with a mutualization layer that shares common introspection work and reduces contention on kernel data structures.

Our evaluation shows that GOODKIT retains the benefits of LibVMI-style introspection while avoiding its costs. On 14 Phoronix workloads, GOODKIT observers introduce at most a 1.06× slowdown on the target, whereas LibVMI induces slowdowns between 5.15× and 37.6× under comparable conditions. On micro-benchmarks, GOODKIT's lock-based introspection is up to 17× faster than LibVMI's pause/resume mechanism, and overall introspection latency improves by up

to 110× for some policies. Across 21 real-world use cases, including rootkit detection, ransomware monitoring, liveness monitoring, and scheduler analysis, GOODKIT provides strong isolation, stable capture rates at high frequencies, and modest implementation effort; observer code is typically 3-6× shorter than equivalent LibVMI-based implementations.
Overall, we make the following contributions:

- We identify requirements for live VMI in multi-tenant clouds: flexibility, isolation, consistency, low overhead, precise accounting, and easy deployment (§3.1).
- We present GOODKIT, an introspection framework that shares the same VMM between a target VM and its observer VM(s) and maps the target memory into each observer VM under explicit policy, without changing the hypervisor (§3.4.
- We design and implement mechanisms for lock-aware, consistent introspection, shared monitoring across observers, and VMM-side probing of VMM and hypervisor events (§4).
- We develop a GOODKIT prototype supporting Linux-based target and observer VMs on x86 machines. We evaluate GOODKIT on 21 real-world use cases and show low overhead, strong isolation, and broad applicability (§5).
- We release GOODKIT as an open-source project (Appendix A).

2 Background and Motivations

This section presents the necessary background on virtualization (§2.1), motivates the need for live VM introspection (§2.2), and discusses the limitations of state-of-the-art approaches (§2.3, §2.4).

2.1 Virtualization

Generalities. Several popular virtualization platforms (e.g., KVM [2] and Oracle VirtualBox [5]) follow a two-layer design with a *hypervisor* and a *virtual machine monitor (VMM)*. Hereafter, we focus our concrete discussion on KVM-based platforms, as they underpin some of the main public (e.g., AWS, GCP) and private (e.g., OpenStack, Proxmox) cloud infrastructures. The hypervisor (KVM in our case) runs in the host kernel and provides core virtualization primitives: vCPU execution, EPT page-fault handling, and VM isolation. It exposes a userspace API (the KVM API) but does not encode any notion of a VMM instance. Because it is part of the kernel, modifying the hypervisor is undesirable.

The VMM (e.g., Firecracker [1, 12] or QEMU [6]) runs as a userspace process and builds full virtual machines using this API. It reserves part of its address space as guest "physical" memory (see below), configures virtual devices, loads the guest OS, and manages paravirtualized I/O during execution. Since VMMs are regular userspace programs, they are the natural place to add mechanisms such as VMI support without touching the hypervisor.[1]

Memory virtualization. It involves several coexisting address spaces, each interpreted at a different layer of the virtualization stack. The *host virtual address (HVA) space* corresponds to the virtual memory of the VMM process (e.g., QEMU or Firecracker). Processes inside the guest OS use the *guest virtual address (GVA) space*. The guest OS maps GVAs to *guest physical addresses (GPAs)* using its own page tables, which represent the physical memory abstraction exposed to the guest. Finally, *host physical addresses (HPAs)* denote the actual machine memory managed by the host kernel and used by the hypervisor.

When creating a VM, the VMM allocates one or more regions in its own virtual address space (HVA), typically via `mmap()`, to hold the guest's RAM. These HVA regions do not need to be contiguous, but the VMM can present them to the guest (GPA) as a contiguous physical address space. It does so by configuring KVM's *memory slots*, each mapping a range of HVAs to a corresponding range of GPAs. For example, an HVA region *(hva1, size)* in the VMM may be exposed to the guest as *(gpa=0, size)*, creating the illusion of a contiguous physical memory layout for the guest regardless of how the VMM arranged the underlying HVAs.

KVM, in turn, maintains for each VM a second-level page table, the *Extended Page Table (EPT)* on x86, that translates GPAs to HPAs. Inside the guest, each process has its own page table maintained by the guest OS, which translates GVAs to GPAs. Thus, when the processor executes a memory access inside the guest, it performs a two-dimensional translation: a guest page walk (GVA to GPA), followed by an EPT walk (GPA to HPA). The resulting HPA identifies the actual physical page on the host. Notably, after memory slots are configured, HVAs play no role in guest execution (in the hardware translation performed on behalf of the guest). The HVA is involved only when the VMM itself accesses the guest's memory (e.g., for snapshots, migrations, or to read VirtIO ring buffers).

2.2 Live VM Introspection

VM introspection (VMI) gives cloud tenants and providers visibility into the internal state of their virtual machines. It supports three main activities: (1) *security operations*, which ensure that a guest does not execute or prepare to execute malicious code (e.g., ransomware); (2) *liveness monitoring*,

[1] Note that the terminology of the domain is not fully homogeneous. Some research papers and existing tools use the expressions *VMM* and *hypervisor* interchangeably to denote the complete virtualization stack. Some other works use *hypervisor* for the entire vitualization system and *VMM* for the subsystem virtualizing the CPU and memory [16]. In the present document, like some other authors, [12, 17, 51], we use *hypervisor* for the host kernel component (here, KVM), and *VMM* for the user-space helper process (here, Firecracker) involved in CPU, memory and I/O virtualization.

which detects crashes, deadlocks, or livelocks and triggers restart; and (3) *forensic analysis*, which examines a VM's memory after a failure to identify root causes. The first two activities typically run on live systems and often precede forensic investigation. We focus on live VMI (LVMI). §5.2 presents several representative real-world LVMI use cases.

We call an *introspection mechanism* any component that can access the internal state of a VM. An *introspection framework* is a coherent stack of mechanisms, libraries, and runtimes that together provide efficient and convenient access to VM state. An *observer* is a system built on top of such a framework to implement high-level LVMI services (for example, liveness monitoring).

2.3 LibVMI-based Observers

LibVMI [8] underpins most industry [4, 9] and academic VMI systems [23, 25, 38, 39, 41, 48, 50]. We first review how it works and then explain why it is a poor fit in live-monitoring scenarios.

Description. LibVMI is a C library that builds on KVM-VMI (KVMI) [37], Bitdefender's extension [15] of KVM for introspection. KVMI adds a kernel subsystem and a userspace library, `libkvmi`, that communicate over a UNIX socket. KVM then spawns a dedicated kernel thread that handles commands from the observer.

LibVMI offers three main classes of commands: (1) *vCPU commands*, to get/set registers, query `cpuid`/`xsave`, and pause or resume vCPUs; (2) *memory commands*, to read/write guest physical memory, adjust page permissions, and translate GVA to GPA; (3) *configuration commands*, to fetch VM/hypervisor metadata (e.g., memory size, KVM version), register events (e.g., CR3 traps), and configure monitoring at VM or vCPU level.

Impracticability for LVMI. LibVMI introduces two major costs for live monitoring:

(1) LibVMI-based tools typically *pause the target VM* to obtain a consistent snapshot of kernel data structures. These global pauses are extremely expensive in LVMI: prior work reports slowdowns up to 9x [31], and we observe similar behavior. The cost grows roughly linearly with the polling rate, which makes LibVMI unsuitable for use cases involving live VM monitoring, such as process scheduling analysis or memory overcommitment.

(2) LibVMI *never accesses guest state directly*. Every operation is routed through KVMI over a socket, causing extra user–kernel transitions and data copies. For example, writing a buffer to guest memory involves copying it from the observer to a KVMI kernel buffer, then again into guest memory.

LibVMI provides an optional cache in the observer to hide part of these costs, but caching trades freshness for speed. It is therefore unusable for most LVMI scenarios that require up-to-date state, such as the use cases in §2.2.

2.4 Limitations of Existing Designs

Existing VMI approaches adopt different placement strategies, each with distinct trade-offs in observability, isolation, performance, and deployability.

A first class of designs, often called out-of-VMM approaches [13, 22, 23, 25, 41, 48, 50, 52], runs the observer in a VM managed by a different VMM instance than the target. Most of these systems rely on LibVMI [8], which exposes a high-level API for inspecting guest state but requires hypervisor modifications. This placement provides strong isolation and clean per-VM accounting. Still, it incurs expensive cross-VMM interactions, frequent VM pauses, and high memory overhead, as each observer requires its own VMM process. Since the observer runs outside the target's VMM, it also cannot see VMM-level events such as device I/O or VM exits. `mmap()`-based variants [21, 28, 30, 40, 60] reduce memory-access cost by mapping the target's memory into the observer VMM, yet they retain key drawbacks: per-observer VMM bloat, dependence on hypervisor changes for memory sharing, and lack of visibility into VMM-level activity.

A second class of systems [24, 41] embeds the observer inside the VMM that hosts the target. The observer is compiled into the VMM process and thus has direct access to guest memory and VMM events. This yields high observability and low-latency introspection. However, it removes isolation: observer and VMM share privileges and address space, so a bug or compromise in the introspection code can bring down the entire VMM and all its guests. This design is also hard to modularize (e.g., to several, dynamically-attached observers) and prevents separate accounting of monitoring overhead.

A third class, in-target approaches [7, 18, 32], places the observer directly inside the target guest OS as an agent or kernel extension. This mitigates the semantic gap and gives very efficient access to guest state but completely forfeits isolation and trust, since compromising the target also compromises the monitor.

GOODKIT adopts a different design: an in-VMM, multi-guest architecture in which observers run as independent guest OSes under the same VMM instance as the target. This preserves the modularity and isolation benefits of out-of-VMM observers while avoiding per-observer VMM overhead, and it still allows access to VMM-level events without compiling monitoring code into the VMM. By extending the VMM (rather than the hypervisor), GOODKIT enables controlled, direct mapping of the target's memory into observer guests while keeping the hypervisor unchanged. As a result, it combines high-performance introspection with strong isolation, precise accounting, and compatibility with existing cloud deployments.

3 Overview

This section details the design requirements of GOODKIT (§3.1), explains the threat model (§3.2), and presents an overview of GOODKIT (§3.4).

3.1 Requirements

We design GOODKIT to meet the following requirements simultaneously:

(R1) ***Flexibility:*** GOODKIT must support a wide range of monitoring use cases, potentially requiring events from the guest OS, the VMM, and the hypervisor. Achieving this goal is particularly challenging because we explicitly avoid modifying the hypervisor (see R6).

(R2) ***Isolation:*** Observers must be isolated from the target guest OS, the VMM, and the hypervisor. The system must provide (i) fault isolation (a misbehaving observer must not disrupt target services) and (ii) privilege isolation (untrusted tenant observation code must be isolated from the VMM).

(R3) ***Consistency:*** Observer data must be accurate and internally consistent. For example, traversing the process list must yield a coherent snapshot even if the target modifies the list concurrently.

(R4) ***Speed and Overhead:*** The system must observe the target in (near) real time, missing no relevant events while imposing low performance overhead on the target.

(R5) ***Resource Efficiency and Accountability:*** Each observer's resource consumption must be small, accurately measurable, and attributable to the corresponding tenant for accounting and billing.

(R6) ***Adoption:*** The system must integrate seamlessly with existing cloud deployments and avoid hypervisor changes, since updating hypervisors at scale is a time-consuming, resource-intensive and security-sensitive endeavor.

Non-goals. There are several valuable directions that lie beyond the scope of GOODKIT. Although GOODKIT enables the development of introspection policies, our objective is not to propose new *policies*, for example, aimed at improving kernel-compromise detection. Similarly, enhancing detection accuracy through novel analysis algorithms is also outside the scope of this work.

3.2 Threat Model

The VMM and hypervisor are part of the Trusted Computing Base (TCB) and are assumed to be correct and uncompromised. The target guest VM may be fully compromised, for example by a rootkit or ransomware. Each observer guest is trusted by its tenant to perform legitimate introspection, but may still be faulty or malicious. A faulty or malicious observer can lie to its tenant, disrupt the services provided by the target or more generally, compromise the integrity of the (target) guest state due to the ability to modify locks or other memory regions. From the cloud provider's perspective, an observer is untrusted: it is a regular VM with the same isolation guarantees as any other tenant VM. A faulty or malicious observer thus cannot compromise other VMs.

3.3 Design Trade-Offs

In GOODKIT's design, an observer runs as an independent guest VM managed by the same VMM instance as the corresponding target VM. Compared to LibVMI, where the monitoring code executes outside of the VMM, the isolation layer is thinner but, in return, we do gain live access to the target VM memory (mapped into the observer), and also direct access to the target VM devices. Modifying the VMM rather than the hypervisor is preferable because the latter is a software layer with a complete set of privileges that must necessarily be under the control of the cloud provider, whereas the ownership of the VMM layer can be more easily offloaded to a cloud tenant. Indeed, unlike the hypervisor, the VMM runs in user mode on the host machine and there is one distinct VMM instance per (traditional) tenant VM. Besides, in real-world scenarios, modern virtualization stacks already make use of multiple sandboxing facilities as an additional level of protection against a potentially misbehaving (buggy or compromised) VMM, including limited (user) privileges and contained domains (built with features like namespaces, seccomp-bpf filters, and control groups) [12]. In our approach, such sandboxing facilities may also be leveraged for mitigating the impact of a misbehaving observer on the host and the other guests.

3.4 Architecture

We now describe how GOODKIT's design (Fig. 1) satisfies the requirements introduced in §3.1. We begin by explaining how GOODKIT runs observers as guest OSes (R2), before detailing the programming model, including the introspection API and the mechanisms for controlling and deploying observers (R1). We then present our memory–mapping scheme, which provides isolation, accurate accounting, and consistent access to the target state (R2–R5). Finally, we show how shared introspection and a purely userspace implementation of GOODKIT reduces overhead and eases adoption in existing cloud environments (R4–R6).

Observer as guest OS, enabling isolation and precise accountability (R2 & R5). In GOODKIT, an observer (A) can run either as a unikernel [36] or a full Linux guest. In our current prototype, we opted for the latter, as it provides a more general and flexible execution environment. (B) GKBUILDER launches observers as separate guests using the same KVM interfaces as for the target (1 and 2). For each observer, it sets up memory slots in the VMM process that KVM uses to build the observer's EPT (C): some slots map selected regions of the target's memory, while others are private. The

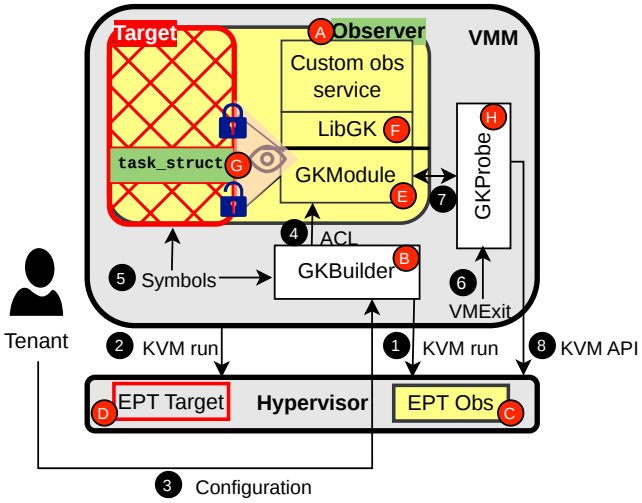


Figure 1: GOODKIT Architecture. The gray boxes correspond to existing Hypervisor and VMM layers. The white boxes correspond to new VMM subcomponents introduced by GOODKIT. The yellow boxes denote the observer's context. In this example, we use a single observer, whose purpose is to introspect a subset of the target VM's internal state, such as the list of existing processes.

target itself is assigned its own independent EPT (D). From the hypervisor's perspective, observers are ordinary guests with their own isolation and resource accounting, yet they retain controlled access to the target's memory where needed.

Introspection API, enabling flexibility (R1). GOODKIT provides a two-layer introspection API (§4.1) that supports a broad range of monitoring and analysis services (§5.2). In the observer VM, the low-level API is implemented in GKMODULE (E) and is used by kernel-space code, while the high-level API is provided by LIBGK (F) for userspace. Observation services can be implemented in the kernel, in userspace, or split across both. Userspace observers can integrate heavyweight analysis components, such as machine-learning models (§5.2, ransomware detection). Kernel-space observers, in contrast, have direct access to the target memory and to the full low-level API, including VMM communication primitives. Userspace observers interact with GOODKIT through the high-level API listed in Table 1, and can also issue requests to the VMM. This makes userspace suitable for complex code that depends on external libraries or convenient device APIs, while kernel space is better suited to low-level memory introspection. When a userspace observer needs information not exposed by the high-level API, it can delegate the low-level target-memory access to a kernel component and consume the resulting data in userspace.

Controlled introspection, enabling flexibility (R1). Tenants specify, in a YAML manifest file (3), which operations each observer is allowed to perform (§4.2). GOODKIT enforces these permissions along four axes: memory protection, introspection-API function access, symbol-table access, and probe control. GKBUILDER applies the memory-protection rules at observer boot time, and (4) GKMODULE enforces function and symbol access at runtime. These act more as guidelines than strong isolation barriers and may help building sound observers. Depending on the use cases, observers may access the target with write accesses essentially limited to the locks protecting the monitored data structures (e.g., for a rootkit detector traversing the list of processes), or also require write accesses to additional parts of the target memory (e.g., for a synchronous/active VMI scenario that places breakpoints or hooks into the target [21, 34]).

Address translation, enabling consistency (R3). Each observer receives the target's kernel symbol table (5) and uses it to locate kernel data structures. Because the target and observer use distinct page tables, these addresses must be translated. GOODKIT relies on the Linux kernel memory layout and implements seven dedicated translation functions, each covering a specific kernel memory region (§4.4). In our current implementation, the target and observer use the same kernel image so that the observer code can more easily interact with the target's kernel data-structure layout during introspection. This is not a fundamental limitation of GOODKIT; the support of decoupled kernels is left for future work. Note that, in any case (i.e., using identical or distinct kernel images for the target and observer VMs), GOODKIT, like many other VMI approaches, relies on the assumption that the observation logic is aware of the precise memory layout and intricacies of the OS kernel running in the target. Therefore, for all these approaches (including GOODKIT), a cleverly-compromised guest might evade detection by an observer if it manages to bypass some of its assumptions.

Fine-grained suspension, enabling consistency and low overhead (R3 & R4). GOODKIT preserves consistency by locking the data structure being inspected (G), instead of suspending the whole VM as LibVMI-based systems do. It assumes the OS already protects shared structures with locks. With the target's memory mapped, the observer acquires the same lock before accessing a structure, as if it were a thread within the target; concurrent updates then block on that lock. GOODKIT provides built-in support for lock acquisition (§4.5).

Shared monitoring, enabling low overhead and resource efficiency (R4 & R5). When several observers perform similar introspection tasks, such as scanning the process list, redundant work and lock contention increase overhead. GOODKIT introduces a mutualizer observer (§4.7) that centralizes these operations. Domain-specific observers register their requests with the mutualizer, which performs each operation once and distributes the results.

Only userspace components, enabling quick adoption (R6). GKBUILDER and GKPROBE (H) are implemented and run in the VMM process in userspace. GKPROBE (§4.6) monitors target-related VM exits, I/O, etc. (6) and runs provider-developed probes that can be enabled according to tenant

Low-level API (GKMODULE)		High-level API (LIBGK)	
ID	**Operation**	**ID**	**Operation**
1	Traverse Linked List	1	Traverse Process List
2	Traverse RB-Tree	2	Traverse Module List
3	Acquire Queued Spinlock	3	List Open Files
4	Acquire Rlock	4	List VM Areas
5	Acquire Wlock	5	List Syscall Routines Addresses
6	Release Queued Spinlock	6	List Process Credentials
7	Release Rlock	7	List `/proc` Entries
8	Release Wlock	8	Check `/proc` File Operations
9	Lookup Symbol By Name	9	Check Present TTY
10	Translate TGVA → OGVA	10	Check System Call Change
11	Send VMM Request	11	Check `AF_INFO` Operations
12	Poll Bulk Data	12	Check Keylogger presence
13	Create Memory Dump	13	Check Netfilter Hijacking

Table 1: A subset of GOODKIT's programming API: low-level (by GKMODULE) and high-level functions (by LIBGK).

needs, then forwards their outputs to observers (❼). Provider-developed probes are capable of implementing arbitrary operations, including (❽) invoking existing hypervisor API to capture hypervisor-level events. Its goal is to cover introspection operations that cannot be performed directly from inside observers. All changes are confined to the VMM and guest; the hypervisor remains unmodified.

4 Design and Implementation

This section presents the design and implementation of GOODKIT. While GOODKIT's design is generic, this section is based on our implementation using Firecracker as the VMM and Intel x86-64 for the CPU architecture.

4.1 Programming API

GOODKIT exposes a two-layer introspection API split between GKMODULE and LIBGK. GKMODULE provides low-level primitives in the observer kernel, including address translation, lock-aware access to target data structures, and interfaces for issuing requests to GKPROBE. LIBGK builds on these primitives to offer a higher-level userspace API for common monitoring tasks. Together, they let developers implement observation services in kernel space, userspace, or both, without handling the underlying mapping and synchronization details. A sample in Appendix B illustrates how to use this API.

Low-level API (GKMODULE). The low-level API is implemented in GKMODULE and provides basic building blocks for introspection. Its functions fall into three categories. First, it offers helpers to traverse common kernel data structures such as linked lists (line 1, first column in Table 1) and red-black trees (line 2, first column). Second, it provides memory-access utilities for address translation (line 10, first column), memory-coherence control (lines 3–8, first column), and raw memory inspection (line 13, first column). Third, it exposes system-wide control functions, including resolving kernel symbols by name (line 9, first column) and sending or receiving requests to or from GKPROBE (lines 11–12, first column).

High-level API (LIBGK). Building on the low-level functions, the high-level API implemented in LIBGK offers ready-to-use implementations of common introspection tasks, drawn from prior work [29, 40, 48]. For example, listing operations (lines 1–7, second column in Table 1), such as traversing the process list or the kernel-module list, internally rely on the linked-list traversal primitive. Several high-level functions also help validate the integrity of the target kernel code (lines 8–13, second column). In particular, in Table 1, all functions whose description begins with the verb *"Check"* are designed to detect malicious hooks or other kernel tampering.

4.2 Policy Enforcement

GOODKIT enforces fine-grained policies that limit what each observer can see and do. Some controls are implemented in the VMM (GKBUILDER, GKPROBE) and cannot be bypassed by observer code. Others are enforced inside the observer by GKMODULE; when the observer comes from a third party, its requests can instead be routed through the mutualizer observer, which applies the same checks on its behalf.

Memory protection. The memory regions visible to an observer depend on its owner and use case. For example, a memory-overcommitment observer (e.g., running Wang et al.'s V-Probe algorithm [58]) deployed by the cloud provider needs to touch only the `vmemmap` region that holds struct page entries; it should not access other parts of the guest. The YAML manifest therefore lists allowed regions and their (read/write) permissions. GKBUILDER enforces these policies when configuring shared mappings for each observer.

Probe control. Probes (described in §4.6) expose VMM- and hypervisor-level information about the target. GKPROBE treats each probe as a configurable unit: the manifest specifies which probes are enabled and, optionally, how far they may inspect guest state. For instance, `blk_probe` can either collect disk I/O request types only or also inspect read/write buffers; its configuration can restrict it to pattern collection. Because probes run in the VMM, these limits hold even if the observer is compromised.

API–function access. GOODKIT maintains an access-control list of API functions that each observer may invoke. The YAML manifest contains the allowed GKMODULE/LIBGK functions; a GKMODULE macro checks at runtime whether the caller is authorized before executing the operation. For third-party observers, the same checks can be applied in the mutualizer, which filters their requests before they reach GKMODULE.

Symbol-table access. Symbol resolution can be confined in a similar way. When needed, the manifest specifies a whitelist of kernel symbols. At runtime, the observer (or mutualizer) may resolve only these symbols; all other lookups fail with a `Symbol Not Found` error.

If no security policy is specified, GOODKIT falls back to a permissive configuration: the observer runs without restrictions, except that probes obey the default settings defined by the cloud provider.

4.3 Memory Mapping

GKBUILDER constructs the address spaces of the target and its observers and wires them together through shared memory. It allocates a contiguous region in the VMM process to back both target and observer memory, then configures KVM memory slots and EPT entries so that each observer sees its own guest-physical space with controlled mappings to the target. In this way, observers obtain direct access to target state while remaining isolated as separate VMs.

GKBUILDER first `mmap`s a large contiguous region in the VMM's virtual address space to back both target and observer memory, laid out as $\texttt{observer}_1 \mid \texttt{observer}_2 \mid \cdots \mid \texttt{target}$. For each observer, it then configures KVM memory slots that, together, form the guest-physical address space: some slots cover the observer's own core memory (allocated by the tenant), others map selected regions of the target. For instance, with 2 GB for the observer and 10 GB for the target, the observer sees 12 GB of physical memory. The precise configuration of the target memory viewed by the observer depends on the aforementioned settings but also of architecture-dependent constraints. For example, on x86, an MMIO hole typically appears near 3.25 GB, so mapping the target into the observer may require up to three slots, depending on how their ranges intersect this hole. We identify three distinct mapping scenarios, which we detail in Appendix C due to space limitations.

4.4 Address Translation

GKMODULE resolves target kernel pointers through a two-step translation: from a *Target Guest Virtual Address* (TGVA) to a *Target Guest Physical Address* (TGPA), and then to an *Observer Guest Virtual Address* (OGVA). Given an address in the target's kernel space (a TGVA), it translates it to a TGPA based on the Linux kernel memory layout, and finally maps that TGPA into the observer's own address space. This pipeline lets the observer access target kernel objects through regular in-guest pointers while preserving isolation.

TGVA → TGPA: Although one could continuously traverse the target kernel's page tables to perform address translation, this would incur substantial overhead. Fortunately, for several Linux kernel memory regions, the layout is sufficiently structured to allow more efficient translations. The Linux kernel divides its virtual address space into seven major regions [3], each with its own data types and allocation policy. GOODKIT therefore chooses the TGVA → TGPA translation method based on the region in which the address lies. The boundaries of these regions are defined by the kernel's memory layout.

(1) Text segment. This region is stored in a contiguous region in the TGPA space. Addresses in this region are translated by simple arithmetic: GKMODULE subtracts the fixed offset `__START_KERNEL_map` from the TGVA.

(2) Direct mapping region. Here, a similar optimization can take place. GKMODULE subtracts the `PAGE_OFFSET` constant from the TGVA. This region includes all contiguous allocations produced by functions such as `kmalloc()`, `kzalloc()`, `kcalloc()`, `alloc_pages()`, and `__get_free_pages()`.

(3) Vmalloc region. Addresses in this region fall within `VMALLOC_START`–`VMALLOC_END` or `MODULES_VADDR`–`MODULES_END`. These ranges cover the kernel modules area, *ioremap* mappings, and memory allocated via the `vmalloc_*` family. For such addresses, GKMODULE performs a page-table walk in the target kernel. It starts from the swapper[2] process' page table, whose page global directory (PGD) base address is exported via the `init_top_pgt` symbol, walks down the page table levels to the page table entry (PTE), and returns the corresponding physical address.

(4) Vmemmap region. Addresses in this region lie within the `VMEMMAP_START`–`VMEMMAP_END` range, which stores the `struct page` array describing physical memory. As for the `vmalloc` area, translation requires a page-table walk starting from the page table of the target's idle process. However, `vmemmap` is backed by huge pages, so the walk stops at the page middle directory (PMD) level instead of the PTE level. While `vmalloc` translation uses the PTE to obtain the physical frame, the PMD entry for `vmemmap` yields the base physical address of a memory section. The final `struct page` address (TGPA) is obtained by adding the offset of the target page within that section.

(5) Per-CPU region. These addresses correspond to per-CPU variables such as `current` (the currently executing `task_struct`) and `runqueues` (per-CPU scheduler queues). Translation uses the `__per_cpu_offset` array: GKMODULE adds the offset for the target CPU to the base address of the per-CPU variable to obtain the complete TGVA, and then translates it as in the direct mapping region.

(6) Fixmap region. This region contains statically mapped kernel addresses. Their translation uses a predefined lookup table that directly resolves virtual-to-physical mappings.

(7) Local Descriptor Table (LDT) regions. LDT-based addresses are process-specific and, in modern Linux kernels, rarely used. They can be treated as a legacy case and are typically ignored in practice.

TGPA → OGVA: At the start of the observer, the physical memory regions of the target are made available to the observer. GKBUILDER uses this information to reserve, in the observer's address space, the ranges that correspond to the target. For each of the regions, the observer allocates an in-

[2] The existence of the swapper process (also known as "idle" process), which spawns one task (i.e., idle loop) per CPU, is an invariant in the Linux kernel. This process also provides us with a stable entry point to access the kernel page table of the target.

memory descriptor. Each descriptor records the region's base address in the target (`target_base`), the corresponding virtual base in the observer (`virt_start`), and the region size.

TGPA translation to OGVA is then performed by locating the descriptor that covers the requested guest address and computing: $\texttt{OGVA} = \texttt{virt_start} + (\texttt{TGPA} - \texttt{target_base})$.

4.5 Fine-Grained Suspension

GKMODULE preserves memory coherence during introspection without the high cost of suspending the entire target VM, as in LibVMI. Instead of stopping the guest, it joins the target kernel's locking protocol: for each data structure it inspects, GKMODULE acquires the lock that protects it, so concurrent updates serialize naturally with observer accesses. Read-only objects (for example, those marked `__ro_after_init` or `const`) are accessed directly, while frequently updated structures are accessed only after taking the corresponding lock, as a regular kernel thread in the target would.

Basic idea. For any observer action that must appear as if it were performed inside the guest, the observer reuses an existing guest execution context whose participation does not affect normal execution. For example, when acquiring spinlocks, it impersonates the context of `vCPU0` (more precisely, it impersonates the target's idle task attached to `vCPU0`.) in the target and enters the same queuing and arbitration logic as a regular thread. This only adds one extra contender to the lock, while the locking protocol and scheduling behavior remain unchanged.

The Linux kernel provides two main classes of locking primitives [57]: spinning locks (spinlocks, bit spinlocks, `rwlock_t`) and sleeping locks (mutex, `rt_mutex`, semaphore, `rw_semaphore`, `ww_mutex`, `percpu_rw_semaphore`). We illustrate our approach with Spinlock, RWlock and Mutex by describing their implementations since they are the most widely used primitives in the kernel [35, 43, 54, 56].

Spinlocks. Our solution builds on the standard *queued spinlock* algorithm and reuses the per-CPU `qnodes` to place the observer in the lock queue. In Linux's queued spinlock, each waiter is represented by an MCS node [45] taken from a per-CPU `qnodes` array. Each CPU has four such nodes, one for each context: *Normal Task*, *Hardware Interrupt*, *Software Interrupt*, and *Non-Maskable Interrupt*. Normally, a CPU runs only one of these contexts at a time, so at most a few `qnodes` are in use. It is extremely unlikely that all four contexts contend for the same lock simultaneously. If this happens, the last context simply spins until a `qnode` becomes free.

In our design, an observer thread behaves like one more nested context. It uses a `qnode` in the same way as an interrupt handler, joins the MCS queue, and eventually acquires the lock in turn. Appendix D illustrates our approach through a simple illustrative scenario.

RWLocks. RW locks follow the usual semantics: at most one writer may hold the lock, whereas multiple readers may hold it concurrently. A writer blocks all other holders; when readers hold the lock, a writer sets the write-pending bit and waits for exclusive access. In Linux, contending readers and writers spin on an internal spinlock embedded in `rwlock_t`. Our observer simply uses the same queued spinlock algorithm: on contention, it joins the queue and participates in arbitration like any other entity, ensuring safe and consistent access to the protected kernel structure.

Mutexes. Mutexes are sleeping locks that can only be used in a preemptible task context. In the classic mutex algorithm, the contender is the `current` task. In GOODKIT, the observer instead impersonates the `idle` thread of the target's CPU0 as the mutex owner, since manipulating the `idle` task cannot disrupt normal service.

Lock-free data structures. Besides these lock-based data structures, Linux also uses Read-Copy-Update (RCU) protected structures, which are common in recent kernel versions. GOODKIT does not currently support RCU-specific synchronization; we discuss possible extensions in §7.2.

4.6 External Event Probing

GKPROBE is a VMM-side helper that gives observers controlled visibility and actuation at the VMM/hypervisor boundary. It lets GOODKIT offer LibVMI-style capabilities (e.g., seeing I/O activity or triggering actions) without running code in the hypervisor or modifying it.

Because many relevant events occur outside the target's memory, along the hypervisor–VMM path or inside the VMM itself, observers cannot access them directly. GKPROBE fills this gap: it runs entirely inside the VMM as a set of probes and actuators that only the cloud provider can configure. For operations already exposed by the KVM API, a probe or actuator often reduces to a simple `ioctl()` call; for events already visible in the VMM (such as VirtIO I/O operations), GKPROBE hooks into existing code paths and reports them to observers as introspection events.

Observer guests communicate with GKPROBE using VirtIO channels called *virtqueues*. An observer can produce one or several messages into a virtqueue and then trigger a hypercall, which allows notifying the corresponding VMM-level probe/actuator for processing them. Symmetrically, a probe/actuator managed by GKPROBE within the VMM can also produce one or several messages into a virtqueue and send an interrupt request to trigger the execution of a VirtIO callback inside the observer (as a response to an observer request), registered by GKMODULE, which can forward the notification to a user-space agent. The precise interaction patterns between an observer and the VMM depends on the characteristics of a given introspection scenario and the chosen probe(s). In many use cases, it is possible to leverage asynchronous communications and message batching to reduce the occurrence of hypercalls. For example, in the case of a disk I/O probe used for ransomware detection (detailed in

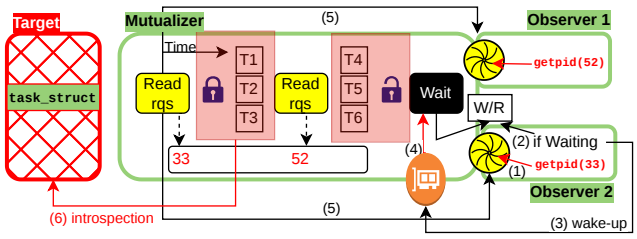


Figure 2: Mutualized introspection.

§5.2), hypercalls are only used sporadically by the observer and the VMM, respectively for submitting a set of buffers and for signaling when they are full.

4.7 Shared Monitoring

GOODKIT reduces redundant work and overhead on the target by delegating common introspection tasks to a dedicated observer, the *mutualizer*, illustrated in Fig. 2. Instead of having each observer traverse the same kernel data structures (such as list or trees of processes, VMAs, modules, etc.), client observers submit requests to the mutualizer. It checks each request against the configured policy, performs the traversal once, and shares the results, reducing CPU overhead, lock contention, and simplifying access control for third-party observers.

The mutualizer offers a configurable, iterator-based interface that treats different types of data structures uniformly. Each request specifies (i) the data structure to traverse (for example, process or module list) and (ii) the item to search for in that structure.

For communication, the mutualizer maintains a per-client ring buffer for requests and completions, and a small shared region that stores its state (`Running` or `Waiting`). When a client issues a request, it writes it into its ring buffer (1) and then checks the mutualizer state (2). If the state is `Running`, the client simply waits for the result. If the state is `Waiting`, it wakes the mutualizer by connecting to a sleeping socket via VirtIO (3). Checking the state avoids unnecessary VirtIO notifications and thus extra hypercalls and VM-Exits.

When woken, the mutualizer scans all ring buffers, groups pending requests by data structure type, and starts one worker thread per type. Each worker first acquires the corresponding lock in the target, then walks the list in fixed-size batches (6). A batch is a fixed number of structure elements (e.g., list nodes), configured at startup. After processing a batch, the worker briefly "spins and waits": it re-examines the ring buffers for new requests of the same type, merges them into its work set, and then continues from the next batch. When a requested item is found, the worker writes the result into the completion ring buffer of the requesting observer (5).

A worker stops if it scans the data structure R consecutive times without seeing any new request. Between two iterations, the mutualizer sleeps for S μs to give the target a chance to acquire the same lock (experimentally, an approximate value of 20 μs provides the best balance between responsiveness and minimization of lock contention). Once all workers have stopped, the mutualizer's main thread switches back to the `Waiting` state.

5 Evaluation

The implementation of GOODKIT consists of two parts: the VMM side and the observer side. The VMM side corresponds to 3,803 LoC (written in Rust) added to Firecracker (1,867 for GKBUILDER and 1,936 for GKPROBE), which represents 5.1% of its full code base. On the observer side, there are an additional 4,480 LoC for GKMODULE and 1,039 for LIBGK, all written in C.

Our evaluation of GOODKIT centers on the following main questions: **(Q1)** How effective and flexible is GOODKIT in a variety of use cases? (§5.2–§5.3) **(Q2)** How does GOODKIT perform compared to the state-of-the-art competitor, LibVMI (§5.4–§5.5)? **(Q3)** What overheads does GOODKIT impose on the target (§5.6)? and **(Q4)** How does the mutualizer observer improve scalability when multiple observers are running (§5.7)?

5.1 Experimental Setup

Testbed. Our experiments run on a 16-core Intel Xeon Silver 4215 at 2.50 GHz with 256 GB of RAM, running Ubuntu 20.04 and Linux 5.4.24+. All guests run Alpine Linux with a 5.10.198 kernel and the VMM is Firecracker 1.5. Unless otherwise specified, each target or observer VM is configured with 1 vCPU.

Baseline. §2.4 gave a qualitative comparison between GOODKIT and existing approaches. For the quantitative evaluation, we compare GOODKIT against LibVMI (with KVMI v7), following prior established methodology [40, 60]. State-of-the-art VMI systems are broadly based on LibVMI. We were unable to reproduce the two in-VMM instrumentation approaches [24, 41] because their source code is not publicly available.

LibVMI targets QEMU and does not natively support Firecracker. We ported LibVMI to Firecracker by reusing its QEMU logic and verified that the resulting performance matches the QEMU-based implementation (§5.4). We publicly release this port (along with the GOODKIT source code) and we provide further details in Appendix E.

Workloads. We use two dimensions of workloads: where they run (inside the target vs. inside observers) and their granularity (micro vs. macro benchmarks). Target-side workloads measure the interference of GOODKIT on guest execution; observer-side workloads act as monitoring services.

ID	Name	ID	Name
P1	Process List	P6	`/proc` File Operations
P2	System Calls	P7	TTY Check
P3	Credentials List	P8	VM Area Check
P4	Keylogger Check	P9	Open File List
P5	Netfilter Check	Idle	Lock/Unlock or Pause/Resume

(a) Micro-benchmarks for observers.

ID	Name & metric	ID	Name & metric
DB	RocksDB 10.0.1 (Ops/sec)	NAS	NAS Parallel Bench. (Mop/sec)
Nginx	Nginx 3.0.1 (Req/sec)	Zstd-C	Zstd Compression (MB/sec)
FIO-MB	Flexible IO Tester (MB/s)	Zstd-D	Zstd Decompression (MB/sec)
FIO-IOPS	Flexible IO Tester (IOPS)	7z-C	7z Compression (MB/s)
iPerf	iPerf (Mbit/s)	7z-D	7z Decompression (MB/s)
GCC	GCC compilation (sec)	Kern	Kernel compilation (sec)
LLVM	LLVM compilation (sec)	OSSL	Open SSL (Bytes/sec)

(b) Phoronix applications used as macro-benchmarks for the target.

ID	Name	ID	Name
Dia	Diamorphine [42]	Ado	Adore-ng [11]
Spy	Spy [49]	Sut	Sutekh [53]
Def	Deftpunk [59]	Liv	Liveness monitor
RQM	Runqueue monitoring	-	-

(c) Macro-benchmarks for observers.

Table 2: Workloads and benchmarks.

Micro-benchmarks are custom workloads that isolate specific GOODKIT features, inspired by prior work [40]. We introduce each target-side micro-benchmark when it first appears in the evaluation; observer-side micro-benchmarks are summarized in Table 2.a. Macro-benchmarks evaluate GOODKIT under realistic conditions. On the target side, we use fourteen applications from the Phoronix benchmark suite [47] (Table 2.b). On the observer side, we implement seven services (Table 2.c, §5.2).

5.2 Real-world Use Cases

We evaluate the effectiveness and flexibility of GOODKIT with realistic use cases. As discussed in §3.1, the novelty of GOODKIT does not lie in these use cases; they simply illustrate the range of applications it can support.

Rootkit detection. Using GOODKIT, we developed four observers to detect four open-source Linux rootkits: *Diamorphine* [42], *Adore-ng* [11], *Spy* [49], and *Sutekh* [53].

Diamorphine hooks three syscalls by overwriting their table entries with pointers to its own module. It hooks `kill()` (used to receive commands such as hide/unhide a process) and `getdents()` / `getdents64()` (used to hide processes from directory listings, including `/proc`). *Sutekh* also hooks syscalls, namely `umask()` and `execve()`, to grant root privileges to a designated process. *Spy* is a keylogger that attaches to the kernel keyboard notification chain by registering a `struct notifier_block`. *Adore-ng* modifies function pointers in the file operations structure for the `/proc` directory to hide processes and files.

Detection focuses on the kernel objects that these rootkits modify: syscall table entries (*Diamorphine*, *Sutekh*), the `struct atomic_notifier_head` for keyboard notifications (*Spy*), and the `/proc` file-operations table (*Adore-ng*). The observer checks whether these pointers reference code outside the kernel text segment, which usually means module memory. It then traverses the module list to find the module whose address range contains the suspicious function. In our experiments, the observers correctly detected all four rootkits.

Ransomware detection. Behavior-based ransomware detection either hooks filesystem syscalls in the guest [19] or inspects disk I/O at the VMM level [14, 27, 59]. We implement the second approach to showcase GKPROBE. Concretely, we reimplement *DefPunk*, a recent ML-based detector from Alibaba [59].

Our observer has two parts: a kernel module and a userspace process. The kernel module configures GKPROBE to capture disk I/O requests issued by the target. Using the probe's configurability, the observer selects which fields to collect and at what frequency. GKPROBE forwards only raw VirtIO metadata (no payload) asynchronously to the observer, keeping the probe lightweight along the guest's critical I/O path. The C structure of these records is shown below:

```
struct __attribute__((packed)) virtio_blk_probe_response {
    uint32_t type;// Request type (e.g., read, write)
    uint64_t sector;// Target disk sector
    uint32_t size;// Request size in bytes
    uint32_t disk_id;// Virtio block device id
};
```

From this stream, the kernel module derives higher-level metrics such as the total number of requests, reads, writes, write-after-read patterns (WAR), the fraction of WARs, and the number of distinct block addresses (working-set size) per time unit. Every 10 s, the userspace component classifies the target's behavior using these metrics. Because the original DefPunk model is not available, we train a linear SVM on the RanSAP dataset [26], which includes benign traces (e.g., Firefox) and several ransomware families such as WannaCry [10].

The resulting model reaches an F1 score of 0.76, comparable to the original [59]. In our tests, it raises no alarms on benign workloads and triggers at least one alert for every ransomware sample.

Liveness Monitoring. The observer plays a role similar to `mysql_safe`: it monitors the `mysqld` process inside the target. Periodically, it walks the target's process list and searches for the `task_struct` whose `comm` field matches the MySQL executable. If no such entry is found, the observer instructs GKPROBE to reboot the target.

CPU runqueue monitoring. The CPU runqueue (`struct rq`) holds the tasks about to run on a given

Use Case	Locks Types	Mem. Types	LibVMI LoC	GoodKit LoC
P1	RWLock(R)	Direct-map, per-CPU	82 (5.4×)	15
P2	-	Text segment	53 (4.8×)	11
P3	RWLock(R)	Direct-map, per-CPU	107 (4.4×)	24
P4	Spinlock	Direct-map, per-CPU	60 (3.3×)	18
P5	Spinlock	Direct-map, per-CPU	76 (4×)	19
P6	-	Text segment	87 (6.2×)	14
P7	Mutex	Direct map, per-CPU	73 (2×)	35
P8	RWLock(R)	Direct map, per-CPU	100 (6.2×)	16
P9	RWLock(R)	Direct map, per-CPU	92 (2.1×)	42
Dia	Mutex	Vmalloc, Text	172 (7.8×)	22
Ado	Mutex	Vmalloc, Text	131 (14.5×)	19
Spy	Mutex	Vmalloc, Direct-map	104 (4.5×)	23
Sut	Mutex	Vmalloc, Text	172 (7.8×)	22
Def	-	-	142 (1.6×)	86
Liv	RWLock(R)	Direct-map, per-CPU	140 (3.2×)	43
RQM	Spinlock	Per-CPU	212 (4.2×)	50

Table 3: Summary of locks, memory translations, and implementation effort for each observer use case. The LoCs correspond to the introspection code without includes. An example of such a code can be seen in Appendix B.

CPU. Unlike the previous use cases on global kernel structures, this one shows that GoodKit can also introspect per-CPU state. Inspecting it reveals CPU pressure and the currently running task. We use this observer to detect if a malicious or buggy process is monopolizing a CPU [55].

5.3 Internal Metrics

We analyze the observer code for our use cases to understand which kernel regions and synchronization primitives they exercise. All the GoodKit basic operations consist of either accessing the target memory (which may involve address translation and lock acquisition steps) or sending requests to the VMM probes through a VM-exit as explained in §4.6. Across all policies, the most frequently translated regions are the *direct-mapped* and *per-CPU* areas, followed by the *text segment* and *Vmalloc* region. This indicates that most introspection tasks focus on kernel code and per-task metadata. For synchronization, mutexes and read-side RWLocks are the most common, each used in 5 of the 15 policies.

During the experiments presented above, we also measured the cost of address translation and lock acquisition operations. For the former, depending on the memory type, latencies range from 45 ns in the best case (direct mapping) to 85 ns in the worst case (`vmalloc` region). The acquisition of locks without contention incurs the same overhead as within the guest: acquiring a spinlock takes approximately 29 ns, while RWLocks and mutexes take respectively approximately 26 and 36 ns.

As Table 3 shows, the implementation effort also differs markedly between GoodKit and LibVMI. Using GoodKit reduces code size by 3-6× across all policies, since the framework provides low-level introspection primitives, address-translation helpers, and coherence mechanisms.

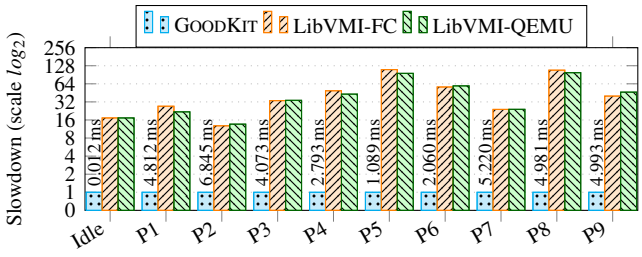


Figure 3: Introspection Turnaround Time.

5.4 Introspection Turnaround Time

We measure how long GoodKit takes to introspect the target and compare it to LibVMI. For both systems, we time the end-to-end cost to (i) resolve target kernel addresses, (ii) apply the synchronization mechanism (lock/unlock in GoodKit, pause/resume in LibVMI), (iii) collect the required information, and (iv) release the synchronization mechanism.

We use P1–P9 and the idle observers from Table 2.a. The idle observer isolates the raw cost of synchronization management. Both observer and target run with a single vCPU, and the target is idle.

Fig. 3 reports the speed up of GoodKit relative to LibVMI (all GoodKit values are normalized to 1). GoodKit is faster in every case. The idle bar shows the cost of LibVMI's `pause/resume` mechanism: acquiring and releasing a lock in GoodKit is 17× faster than pausing and resuming the guest. Locking is not the only factor. Because GoodKit observers are colocated with the VMM, they benefit from *native-speed memory access* (§5.4). This is visible in P2, where no synchronization is needed and both observers simply read the (stable) syscall table; LibVMI is still much slower. LibVMI's slowdown reaches up to 110× for P5. This is because tied to its nested loop structure that multiplies the number of `vmi-read` requests of P5.

These experiments also validate our LibVMI port for Firecracker. The slowdowns for Firecracker and QEMU versions are nearly identical, confirming that our Firecracker integration preserves LibVMI's performance characteristics.

5.5 Capture Rate

We evaluate the introspection speed of GoodKit against LibVMI using the cat-and-mouse experiment of Hongyi Liu et al. [40]. In this scenario, a kernel rootkit repeatedly corrupts the credential structures of selected processes and quickly restores them. By varying the modification frequency, we measure how often the observer catches these transient inconsistencies.

Fig. 4 shows results for attack rates up to 500 modifications per second. Above 150 modifications per second, LibVMI's capture rate drops to about 50%, and at 200 modifications per

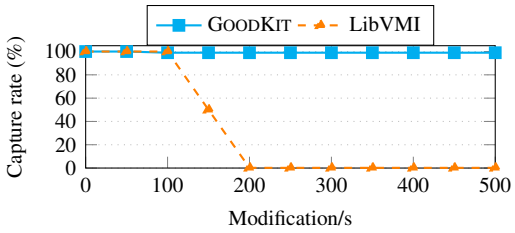


Figure 4: Rootkit capture rate of GoodKit and LibVMI in a cat-and-mouse scenario [40].

second it falls to 0.16%, showing that pause/resume cannot keep up with fast-changing kernel state. In contrast, GoodKit maintains a capture rate of 99% across the same range, thanks to its low-latency, lock-based coherence mechanism. Even at 10 million modifications per second (not shown in Fig. 4), GoodKit still achieves a capture rate of 80%.

5.6 Introspection Overhead

We now evaluate the overhead GoodKit imposes on the target, upon boot and during execution.

5.6.1 Boot Time

GoodKit starts observers inside the same VMM as the target. This increases cold-start time, which is critical in settings such as serverless computing. We evaluate three scenarios: *Parallel*, where GKBuilder launches the observer while the vanilla code builds the target; *Before Target*, where GKBuilder launches the observer first; and *After Target*, where GKBuilder launches the observer just after the target is created. In the *Parallel* configuration, we vary the number of observers from one to five.

With one observer in *Parallel*, the target boot time increases by about 1.08x-1.11x compared to the vanilla (no-observer) setup. This comes from doubling the number of `ioctl()` calls to KVM and the associated kernel-side synchronization. The overhead grows with the number of observers, reaching 1.57x for five observers; each additional observer adds roughly 1.13x-1.14x to the target build time. Thus, instantiating many observers in parallel can noticeably impact VM boot time.

In the *After Target* configuration, the target's boot time increases only by about 1.02x-1.04x. This small cost is due to parsing both configuration files and allocating memory for both guests at initialization. Importantly, this overhead does not grow with the number of observers. In the *Before Target* configuration, the target's boot time essentially doubles, since the observer is fully initialized before the target's own boot begins.

Given this overhead, users should carefully choose how they instantiate observers so they meet their application needs while minimizing overhead.

5.6.2 Execution Time

We now measure the runtime overhead imposed on the target. The target uses 14 vCPUs, and each observer runs with a single vCPU. The target executes the 14 Phoronix applications listed in Table 2.b. The baseline, normalized to 1, corresponds to running the applications without any observation. We evaluate two configurations: (i) a *single observer* and (ii) *multiple observers*. For the first configuration, we consider two representative observers—P3 (memory introspection) and *DefPunk* (VMM-level I/O events)—and we vary their observation frequency. For the second configuration, we launch the target alongside four observers: *DefPunk*, a *rootkit detector*, a *keylogger detector* (P4), and a `/proc` *file operations checker* (P6).

Single observer (P3). The second bar in Fig. 5 reports the slowdown induced by P3 on the target. Across all workloads, the P3-GoodKit observer introduces at most a 1.06× slowdown (observed on OpenSSL), which is negligible. In contrast, P3-LibVMI incurs slowdowns ranging from 5.15× to 37.6×. This substantial gap is primarily due to LibVMI's use of the *pause/resume* mechanism to ensure memory coherence, whereas GoodKit relies on lightweight, fine-grained locking.

Single observer (Deftpunk). For *Deftpunk*, we evaluate both the ransomware trace replayer from §5.2 and the `FIO-IOPS` benchmark. With GoodKit, the replayer achieves 874 I/O requests per second, matching the vanilla throughput. Under LibVMI [3], throughput drops to 747 I/O requests per second, corresponding to a **1.17× slowdown** compared to GoodKit. For `FIO-IOPS`, which is more I/O intensive than the ransomware trace, the slowdown under LibVMI is even higher, reaching **1.87**×.

Heterogeneous observers and scalability. The third bar reports the slowdown observed on the target when running the four observers simultaneously. The slowdown is approximately **1.16x** on average, which is comparable to the degradation observed with the single P3 observer earlier in §5.6.2. This experiment demonstrates that GoodKit enables efficient multi-observer introspection without significantly increasing overhead on the target.

We did not conduct the same experiment with LibVMI because LibVMI does not support multiple independent observers. If observers run as distinct processes, a pause request issued by one observer may conflict with an unpause request from another, leading to inconsistent states. Achieving coordinated behavior would require colocating all observers in a single program or implementing an external synchronization mechanism, both of which prevent running legacy observers. In contrast, GoodKit naturally supports multiple isolated observers without requiring any coordination or code modifications.

[3] We added I/O introspection support to LibVMI (see Appendix F).

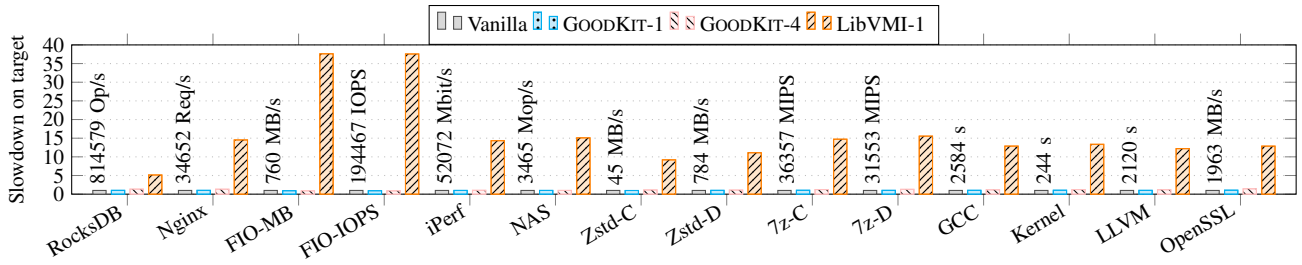


Figure 5: Overhead of GoodKit (using 1 and 4 observers) and LibVMI on the Phoronix benchmark suite.

5.7 Mutualizer Benefits

We evaluate the benefits of the *mutualizer*, which makes sense when several observers monitor the same data structures in the target. We consider both the target's and the observers' perspectives. The target runs a multi-threaded program that continuously creates POSIX threads using the `pthread` library. Each thread executes an empty function, and the main thread waits for it to terminate before creating the next one. This sustained thread creation keeps updating the kernel's thread list. Each observer repeatedly searches for a given PID in the target's thread list and performs this lookup N times.

Without the mutualizer, all observers complete their workload, but the target freezes as soon as more than one observer runs. Traversing the process list requires acquiring an RWLock in read mode, while the kernel needs the same lock in write mode to update the list when creating or terminating threads. Because observers traverse the list in a tight loop with no pause, at least one of them almost always holds the read-lock. RWLocks allow concurrent readers and do not enforce fairness between readers and writers, so the writer (the target) starves and never acquires the write-lock. The target can no longer update the thread list and eventually stalls.

With the mutualizer, this problem disappears. Only the mutualizer takes the read-lock, and it releases it after each traversal. Writers (running in the target) regularly obtain the write-lock, preventing starvation. Performance remains stable and independent of the number of observers: the target sustains approximately 35K thread creations per second, and each observer performs about 38K iterations per second. We observe the same performance when a single observer runs without the mutualizer. For comparison, a target without observers creates about 40k threads in our configuration.

6 Discussion

This section explains the position of an observer developer regarding the locks acquisition order. It also talks about how atomic variables are managed with GoodKit mechanisms.

6.1 Target's Locks Acquisition Order

The Linux kernel has a well-established locking discipline. When analyzing a complex kernel state, an observer must decide whether to follow the full locking protocol. Not following the target kernel's lock discipline may leave the system in an inconsistent state. Observer developers must carefully consider the locking strategy based on the kernel data structures they inspect and their invariants.

6.2 Atomic Variables

Atomic variables are variables that are protected by the hardware while they are being modified. This means that the coherence between two VMs that are dealing with an atomic data item is not guaranteed by the operating system, but by the hardware; this makes GoodKit-based introspection straightforward for these.

7 Limitations and Future Work

In this section, we consider some questions raised by our approach regarding the observers fault tolerance and RCU-based synchronization between an observer and its target VM.

7.1 Observers Fault Tolerance

Running observers may be malicious or faulty. While malicious observers are out of scope of this work, faulty observers should be handled to mitigate service disruption. Faulty observers can disrupt normal target execution if they acquire a target lock and then crash. To handle crashes, GoodKit may record each lock operation in a buffer shared between the observer and Firecracker, allowing the VMM to track the locks currently held by each observer. When an observer crashes, the VMM would check this shared state. If the lock map is stable, the observer holds no target lock. Otherwise, the VMM will restart the observer in recovery mode. The recovery observer would release all locks held by the failed observer, ending the disruption on the target. After recovery, the observer can either resume normal execution or shut down. A watcher may also be implemented to monitor observers that hold locks for too long. This can be done by recording the

timestamp alongside the lock map mentioned before, and have a pre defined time limit for locks holding. This has been planned as future work.

7.2 RCU-protected Data Structures

RCU (read-copy-update) [44] supports both read and write synchronization. For reads, a thread enters an RCU-protected section with `rcu_read_lock()`, obtains the current version of the object with `rcu_dereference()`, and exits with `rcu_read_unlock()`. For writes, a thread initializes a new version of the object, publishes it with `rcu_assign_pointer()`, and waits for a grace period with `synchronize_rcu()` (or `call_rcu()` for the asynchronous variant) before reclaiming old versions.

GOODKIT's current proof of concept does not implement RCU synchronization. When the observer needs to access data structures in read mode (which is likely the most dominant case), it can mimic a target reader by entering and leaving RCU reader-side sections. The main challenge comes when a reader observer needs to interact with the writer. According to the RCU protocol, a writer in the target waits until all CPUs that were executing between `rcu_read_lock()` and `rcu_read_unlock()` have left their critical sections before reclaiming old versions. An external observer is not naturally part of this protocol, so data stability is not guaranteed without additional mechanisms. Moreover, an observer performing writes cannot track the exact state of the target vCPUs in the target code execution, hindering the synchronization process.

8 Related Work

Note that we already discussed a portion of the related work in the motivation section (§2.4).

Most existing systems that address memory coherence during VMI rely on *pause-based* consistency mechanisms. Many tools depend on LibVMI [8] as their backend, and consequently adopt the strategy of pausing the target vCPUs before inspecting memory. Prior work [41] highlights several drawbacks of such techniques: latency-sensitive services may experience noticeable disruptions, and frequent pauses increase overall system jitter. Cosseron et al. [20] further demonstrated that pausing a VM induces a measurable drift between the VM's internal clock and real time, a side effect that can be detected by malware and used as a trigger to conceal or disable malicious behavior.

A different perspective is offered by TxIntro [41], which leverages Intel's Hardware Transactional Memory (HTM) [33]. TxIntro monitors the state of synchronization variables protecting kernel data structures and performs introspection within a hardware transaction. As long as the lock variables remain unchanged, the transaction proceeds, ensuring that the observed structure is read consistently; if a concurrent modification is detected, the transaction aborts and partial results are discarded. While this approach avoids explicit VM pauses, it has several limitations. First, it depends on HTM support, which is not always available or enabled on cloud platforms, e.g., due to mitigations against side-channel attacks. Second, it can only provide consistency guarantees when observing guest state that fits within the maximum read-/write set capacity of the HTM. Third, it requires hypervisor modifications, unlike GOODKIT.

Some studies, such as Virtuoso [23] and VM-Space Traveler [25], address the semantic gap by facilitating or automating the generation of introspection operations over guest kernel data structures. These efforts are orthogonal to GOODKIT and can enrich or automate its introspection API while preserving GOODKIT's isolated and efficient execution model.

BlueGuard [46] targets both hypervisor-level and VM-level introspection, whereas GOODKIT focuses exclusively on introspecting guest VMs. To achieve this, BlueGuard isolates its observer on a dedicated accelerator (e.g., a SmartNIC) and relies on DMA to access memory. However, BlueGuard exhibits two main limitations. First, DMA-based access can lead to data-coherence issues, since memory may be concurrently modified by the CPU. Second, BlueGuard requires specialized hardware accelerators, which significantly restricts its deployability in real-world cloud environments.

9 Conclusion

In this work, we introduced GOODKIT, a new LVMI framework that enables efficient, scalable, and safe introspection without relying on intrusive hypercalls or VM pausing. By colocating observers with the VMM, enforcing lightweight coherence mechanisms, and supporting flexible, modular probes, GOODKIT significantly reduces overhead while improving responsiveness and accuracy. Our evaluation shows that GOODKIT outperforms LibVMI by large margins across all scenarios, offers strong scalability thanks to the mutualizer design, and supports a wide range of real-world use cases. Overall, GOODKIT bridges the gap between practicality and performance, enabling cloud providers and tenants to deploy LVMI at scale. The GOODKIT prototype is publicly available as described in Appendix A.

Acknowledgments

We thank our anonymous shepherd and the anonymous reviewers for their insightful comments. This work was supported by the French *Agence Nationale de la Recherche* (ANR) under the DiVa project of "PEPR Cloud" (ANR-23-PECL-0004), by Inria "Défi OS", and by the CNRS MLNS2 International Research Project (IRP). Finally, we thank Baptiste Lepers and André Freyssinet for their feedback.

References

[1] Firecracker. https://firecracker-microvm.github.io/. Accessed June 8, 2026.

[2] Kernel Virtual Machine. https://linux-kvm.org/page/Main_Page. Accessed June 8, 2026.

[3] Linux kernel memory layout for x86. https://www.kernel.org/doc/Documentation/x86/x86_64/mm.txt. Accessed June 8, 2026.

[4] NoGap: New Security Monitoring Technology in the Cloud. https://hellofuture.orange.com/en/nogap-new-security-monitoring-technology-in-the-cloud/. Accessed June 8, 2026.

[5] Oracle VirtualBox. https://www.virtualbox.org/. Accessed June 8, 2026.

[6] QEMU. https://www.qemu.org/. Accessed June 8, 2026.

[7] QEMU GuestAgent. https://wiki.qemu.org/Features/GuestAgent. Accessed June 8, 2026.

[8] The LibVMI API. https://libvmi.com/api/. Accessed June 8, 2026.

[9] Volatility. https://github.com/volatilityfoundation/volatility. Accessed June 8, 2026.

[10] Wannacry. https://fr.wikipedia.org/wiki/WannaCry. Accessed June 8, 2026.

[11] Adore-Ng Rootkit Sources. https://github.com/yaoyumeng/adore-ng. Accessed June 8, 2026.

[12] Alexandru Agache, Marc Brooker, Alexandra Iordache, Anthony Liguori, Rolf Neugebauer, Phil Piwonka, and Diana-Maria Popa. Firecracker: Lightweight Virtualization for Serverless Applications. In *Symposium on Networked Systems Design and Implementation (NSDI'20)*. USENIX, 2020.

[13] Kurniadi Asrigo, Lionel Litty, and David Lie. Using VMM-Based Sensors to Monitor Honeypots. In *International Conference on Virtual Execution Environments (VEE'06)*. ACM, 2006.

[14] SungHa Baek, Youngdon Jung, Aziz Mohaisen, Sungjin Lee, and DaeHun Nyang. SSD-Insider: Internal Defense of Solid-State Drive against Ransomware with Perfect Data Recovery. In *International Conference on Distributed Computing Systems (ICDCS'18)*. IEEE, 2018.

[15] Bitdefender. Bitdefender Website. https://www.bitdefender.com/. Accessed June 8, 2026.

[16] Edouard Bugnion, Jason Nieh, and Dan Tsafrir. *Hardware and Software Support for Virtualization*. Morgan & Claypool Publishers, 2017.

[17] Jiahao Chen, Dingji Li, Zeyu Mi, Yuxuan Liu, Binyu Zang, Haibing Guan, and Haibo Chen. Security and Performance in the Delegated User-level Virtualization. In *Symposium on Operating Systems Design and Implementation (OSDI'23)*. USENIX, 2023.

[18] Tzi-cker Chiueh, Matthew Conover, and Bruce Montague. Surreptitious Deployment and Execution of Kernel Agents in Windows Guests. In *International Symposium on Cluster, Cloud and Grid Computing (CCGRID'12)*. IEEE/ACM, 2012.

[19] Andrea Continella, Alessandro Guagnelli, Giovanni Zingaro, Giulio De Pasquale, Alessandro Barenghi, Stefano Zanero, and Federico Maggi. ShieldFS: A Self-Healing, Ransomware-Aware Filesystem. In *Annual Conference on Computer Security Applications (ACASC'16)*. ACM, 2016.

[20] Léo Cosseron, Louis Rilling, Matthieu Simonin, and Martin Quinson. Simulating the Network Environment of Sandboxes to Hide Virtual Machine Introspection Pauses. In *European Workshop on Systems Security (EuroSec'24)*. ACM, 2024.

[21] Thomas Dangl, Benjamin Taubmann, and Hans P Reiser. RapidVMI: Fast and Multi-core Aware Active Virtual Machine Introspection. In *International Conference on Availability, Reliability and Security (ARES'21)*. ACM, 2021.

[22] Xuhua Ding. How to Resuscitate a Sick VM in the Cloud. In *International Conference on Dependable Systems and Networks (DSN'23)*. IEEE/IFIP, 2023.

[23] Brendan Dolan-Gavitt, Tim Leek, Michael Zhivich, Jonathon Giffin, and Wenke Lee. Virtuoso: Narrowing the Semantic Gap in Virtual Machine Introspection. In *Symposium on Security and Privacy (S&P'11)*. IEEE, 2011.

[24] Pavel Dovgalyuk, Natalia Fursova, Ivan Vasiliev, and Vladimir Makarov. QEMU-Based Framework for Non-Intrusive Virtual Machine Instrumentation and Introspection. In *Joint Meeting on Foundations of Software Engineering (FSE'17)*. ACM, 2017.

[25] Yangchun Fu and Zhiqiang Lin. Bridging the Semantic Gap in Virtual Machine Introspection via Online Kernel Data Redirection. *ACM Transactions on Information and System Security*, 2013.

[26] Manabu Hirano, Ryo Hodota, and Ryotaro Kobayashi. RanSAP: An Open Dataset of Ransomware Storage Access Patterns for Training Machine Learning Models. *Forensic Science International: Digital Investigation*, 2022.

[27] Manabu Hirano and Ryotaro Kobayashi. Machine Learning Based Ransomware Detection Using Storage Access Patterns Obtained From Live-forensic Hypervisor. In *International Conference on Internet of Things: Systems, Management and Security (IOTSMS'19)*. IEEE, 2019.

[28] Jennia Hizver and Tzi-cker Chiueh. Real-Time Deep Virtual Machine Introspection and Its Applications. In *International Conference on Virtual Execution Environments (VEE'14)*. ACM, 2014.

[29] Owen S. Hofmann, Alan M. Dunn, Sangman Kim, Indrajit Roy, and Emmett Witchel. Ensuring Operating System Lernel Integrity with OSck. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'11)*. ACM, 2011.

[30] Jiaqi Hong and Xuhua Ding. A Novel Dynamic Analysis Infrastructure to Instrument Untrusted Execution Flow Across User-Kernel Spaces. In *Symposium on Security and Privacy (S&P'21)*. IEEE, 2021.

[31] Guangyuan Hu, Tianwei Zhang, and Ruby B. Lee. Position Paper: Consider Hardware-enhanced Defenses for Rootkit Attacks. In *International Workshop on Hardware and Architectural Support for Security and Privacy (HASP'20)*. ACM, 2020.

[32] Amani S. Ibrahim, James Hamlyn-Harris, John Grundy, and Mohamed Almorsy. CloudSec: A Security Monitoring Appliance for Virtual Machines in the IaaS Cloud Model. In *International Conference on Network and System Security (NSS'11)*. IEEE, 2011.

[33] Intel. Exploring Intel® Transactional Synchronization Extensions with Intel® Software Development Emulator. https://www.intel.com/content/www/us/en/developer/articles/community/exploring-tsx-with-software-development-emulator.html. Accessed June 8, 2026.

[34] Bhushan Jain, Mirza Basim Baig, Dongli Zhang, Donald E Porter, and Radu Sion. Sok: Introspections on Trust and the Semantic Gap. In *2014 IEEE symposium on security and privacy*, pages 605–620. IEEE, 2014.

[35] Kernel Locking Techniques. https://dl.acm.org/doi/fullHtml/10.5555/563953.563964. Accessed June 8, 2026.

[36] Simon Kuenzer, Vlad-Andrei Bădoiu, Hugo Lefeuvre, Sharan Santhanam, Alexander Jung, Gaulthier Gain, Cyril Soldani, Costin Lupu, Ştefan Teodorescu, Costi Răducanu, Cristian Banu, Laurent Mathy, Răzvan Deaconescu, Costin Raiciu, and Felipe Huici. Unikraft: Fast, Specialized Unikernels the Easy Way. In *European Conference on Computer Systems (EuroSys'21)*. ACM, 2021.

[37] KVM-based Virtual Machine Introspection. https://github.com/KVM-VMI/kvm-vmi. Accessed June 8, 2026.

[38] Tamas K. Lengyel, Steve Maresca, Bryan D. Payne, George D. Webster, Sebastian Vogl, and Aggelos Kiayias. Scalability, Fidelity and Stealth in the DRAKVUF Dynamic Malware Analysis System. In *Annual Computer Security Applications Conference (ACSAC'14)*, 2014.

[39] Tamas K Lengyel, Justin Neumann, Steve Maresca, Bryan D Payne, and Aggelos Kiayias. Virtual Machine Introspection in a Hybrid Honeypot Architecture. In *Workshop on Cyber Security Experimentation and Test (CSET'12)*. USENIX, 2012.

[40] Hongyi Liu, Jiarong Xing, Yibo Huang, Danyang Zhuo, Srinivas Devadas, and Ang Chen. Remote Direct Memory Introspection. In *Security Symposium (USENIX Sec'23)*. USENIX, 2023.

[41] Yutao Liu, Yubin Xia, Haibing Guan, Binyu Zang, and Haibo Chen. Concurrent and Consistent Virtual Machine Introspection with Hardware Transactional Memory. In *International Symposium on High Performance Computer Architecture (HPCA'14)*. IEEE, 2014.

[42] M0nad. Diamorphine Rootkit. https://github.com/m0nad/Diamorphine. Accessed June 8, 2026.

[43] John Madieu. *Linux Device Drivers Development*. Packt Publishing, 1st edition, October 2017.

[44] Paul E. McKenney. Is Parallel Programming Hard, And, If So, What Can You Do About It? https://mirrors.edge.kernel.org/pub/linux/kernel/people/paulmck/perfbook/perfbook-1c.2025.12.18a.pdf. Accessed June 8, 2026.

[45] MCS Locks and Qspinlocks. https://lwn.net/Articles/590243/. Accessed June 8, 2026.

[46] Meni Orenbach, Rami Ailabouni, Nael Masalha, Thanh Nguyen, Ahmad Saleh, Frank Block, Fritz Alder, Ofir Arkin, and Ahmad Atamli. BlueGuard: Accelerated Host and Guest Introspection Using DPUs. In *Security Symposium (USENIX Sec'25)*. USENIX, 2025.

[47] Phoronix Benchmark Suite. https://github.com/phoronix-test-suite/phoronix-test-suite/blob/master/documentation/phoronix-test-suite.md. Accessed June 8, 2026.

[48] Fabian Schwarz and Christian Rossow. 00SEVen – Re-enabling Virtual Machine Forensics: Introspecting Confidential VMs Using Privileged in-VM Agents. In *Security Symposium (USENIX Sec'24)*. USENIX, 2024.

[49] Spy Rootkit Sources. https://github.com/jarun/spy. Accessed June 8, 2026.

[50] Deepa Srinivasan, Zhi Wang, Xuxian Jiang, and Dongyan Xu. Process Out-Grafting: An Efficient "Out-of-VM" Approach for Fine-Grained Process Execution Monitoring. In *Conference on Computer and Communications Security (CCS'11)*. ACM, 2011.

[51] Udo Steinberg and Bernhard Kauer. NOVA: a Microhypervisor-Based Secure Virtualization Architecture. In *European Conference on Computer Systems (EuroSys'10)*. ACM, 2010.

[52] Sahil Suneja, Canturk Isci, Vasanth Bala, Eyal de Lara, and Todd Mummert. Non-intrusive, Out-of-band and Out-of-the-box Systems Monitoring in the Cloud. In *International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS'14)*. ACM, 2014.

[53] Sutekh Rootkit Sources. https://github.com/PinkP4nther/Sutekh. Accessed June 8, 2026.

[54] The Linux Kernel Locking API and Shared Objects. https://medium.com/geekculture/the-linux-kernel-locking-api-and-shared-objects-1169c2ae88ff. Accessed June 8, 2026.

[55] Dan Tsafrir, Yoav Etsion, and Dror G. Feitelson. Secretly Monopolizing the CPU without Superuser Privileges. In *Security Symposium (USENIX Sec'07)*. USENIX, 2007.

[56] Two Main Types of Kernel Locks: Spinlocks and Semaphores. https://www.kernel.org/pub/linux/kernel/people/rusty/kernel-locking/c93.html. Accessed June 8, 2026.

[57] Locks Types and Their Rules. https://docs.kernel.org/locking/locktypes.html. Accessed June 8, 2026.

[58] Yaohui Wang, Ben Luo, and Yibin Shen. Efficient Memory Overcommitment for I/O Passthrough Enabled VMs via Fine-grained Page Meta-data Management. In *Annual Technical Conference (ATC'23)*. USENIX.

[59] Zhongyu Wang, Yaheng Song, Erci Xu, Haonan Wu, Guangxun Tong, Shizhuo Sun, Haoran Li, Jincheng Liu, Lijun Ding, Rong Liu, Jiaji Zhu, and Jiesheng Wu. Ransom Access Memories: Achieving Practical Ransomware Protection in Cloud with DeftPunk. In *Symposium on Operating Systems Design and Implementation (OSDI'24)*. USENIX, 2024.

[60] Siqi Zhao, Xuhua Ding, Wen Xu, and Dawu Gu. Seeing Through The Same Lens: Introspecting Guest Address Space At Native Speed. In *Security Symposium (USENIX Sec'17)*. USENIX, 2017.

A Artifact Appendix

Abstract

This artifact is the prototype of GOODKIT. The code corresponds to the implemented mechanisms mentioned in the present paper. This concerns the Firecracker modification, GKMODULE and LIBGK. We also make available the Firecracker LibVMI port. This includes the Firecracker, LibVMI, LibKVMI and KVMI modifications.

Scope

This prototype allows you to make observations on a target VM. To run existing use cases, and build new live VMI (LVMI) use cases based on the existing tools. It can also serve as a basis for extended research on live VMI.

Contents

The repository contains the following:

- `firecracker-main`: A fork of Firecracker (written in Rust) with the framework's VMM integration. It allows creating/starting observers, executing probes, and managing VM–VMM communication.
- `images_builder`: Dockerfiles and scripts for building VM images, including those used in the evaluation.
- `ioctl_injector`: Userspace LIBGK code (written in C) that implements the high-level API.
- `libVMI`: Firecracker's LibVMI plugin, plus our KVMi patch adding VirtIO support for LibVMI.
- `linux-5.10.198`: Linux kernel sources (written in C) used to build the target and observer vmlinux images.
- `usecase`: Source code for the implemented use cases. The repository root also includes scripts to run experiments, a Makefile to build the project and images, and a `flake.nix` file for Nix-assisted reproducibility.

Hosting

The public link for our artifact is https://doi.org/10.5281/zenodo.20069006. It links to a Git repository.

Requirements

This artifact requires an x86 machine with KVM support capable of running Firecracker microVMs, along with the standard toolchain required to configure and compile a Linux kernel. The prototype was primarily evaluated on an Ubuntu 20.04 host running Linux kernel 5.4.24+, but it is also compatible with Ubuntu 24.04 running kernel 6.8.0. Our experimental evaluation was conducted on a 16-core Intel Xeon Silver 4215 CPU, 2.50 GHz with 256 GB of RAM. However, a 2-core CPU is sufficient to run the prototype. Finally, reproducing the experiments detailed in §5.6.2 requires at least 90 GB of RAM and 60 GB of disk space.

B Code Sample

Listing 1 is a snippet that depicts the observation of a process list. The introspection starts with a symbol lookup. The observer first gets the addresses related to the corresponding symbols (lines 2–4). Those are: the head of the tasks list `init_task`, the qnodes that reside in the per-CPU area `qnodes`, the tasks list lock that protects tasks list modifications `tasklist_lock` and the CPU areas offset array, which is subsequently used to resolved per-CPU addresses `_per_cpu_offset`.

The observer then translates the addresses to OGVA, for direct access (lines 6–8). The observer then takes the read lock on the `rwlock_t` (line 10). The `for_each_process_in_target` traverse the list of processes (lines 11–13) and translate the relevant addresses under the hood, and gives a pointer `p` on the current entry for usage. The observer then prints the process name (line 12) and releases the lock afterwards (line 14).

Listing 1: Process list traversal from an observer using GoodKit's API. The program resolves the symbol adresses using the symbol table, then grabs the protecting spinlock before walking through the list.

```
1 void process_list(struct context *vo){
2   rwlock_t *tasklock; struct task_struct *p; unsigned
      long *pcpu_off; struct qnode *qnode0; struct
      symbol_entry **syms; char *names[] = { "init_task",
      "qnodes", "tasklist_lock", "__per_cpu_offset" };
3   /* Lookup kernel symbols */
4   syms = symbols_multilookup(names, ARRAY_SIZE(names));
5   /* resolve into Observer virtual address space */
6   pcpu_off = tgva_to_ogva(syms[3]->addr, vo);
7   qnode0 = tgva_to_ogva(syms[1]->st_addr + pcpu_off[0],
      vo);
8   tasklock = tgva_to_ogva(syms[2]->st_addr, vo);
9   /* Lock, walk task list and unlock */
10  obs_read_lock(vo, tasklock, qnode0);
11  for_each_process_in_target(p, syms[0]->st_addr, vo) {
12    pr_info("Process name : %s",p->comm)
13  }
14  obs_read_unlock(vo, tasklock);
15 }
```

C Memory Slots Scenarios

KVM slots are structures that describe the physical memory layout of the virtual machine. A slot contains a range of guest physical addresses, their corresponding host virtual addresses in the VMM, and region-specific flags. Slots are used to declare the complete VM's physical memory layout to KVM. The number of KVM slots per observer depends on (i) architecture-specific physical layout constraints and (ii) the relative sizes of observer and target. On x86, an MMIO hole typically appears near 3.25 GB, so mapping the target into the observer may require up to three slots, depending on how their ranges intersect this hole. We identify three distinct mapping scenarios.

Scenario 1. Either (i) the sum of target memory and observer core memory is below 3.25 GB, or (ii) the observer core memory alone is above 3.25 GB while the target memory remains below this threshold. In both cases, the target memory occupies a single contiguous region in the observer's physical address space, so GKBuilder creates *one* KVM memory slot for the target.

Scenario 2. Either (i) the observer core memory is below 3.25 GB and the combined observer+target size exceeds 3.25 GB, or (ii) both observer and target are individually larger than 3.25 GB. Here, adding the target crosses the MMIO hole and introduces a discontinuity in the observer's physical address space. GKBuilder therefore creates *two* KVM memory slots for the target.

Scenario 3. If the observer core memory is below 3.25 GB and the target memory itself spans beyond the MMIO hole, mapping the target into the observer creates *two* distinct holes: the observer's MMIO hole and the target's own MMIO hole. In this case, GKBuilder creates *three* KVM memory slots for the target.

D Illustration of Spinlock Acquisition by a GoodKit Observer

Figure 6 shows how the observer thread W competes with two target threads, T1 and T2, for the same spinlock. (1) T2, on vCPU1, acquires the lock and enters the critical section. (2) T1, on vCPU0, tries to acquire the lock; because it is held, T1 spins and sets the pending bit. (3) T1 is added to the lock's waiting list, stored in a per-CPU (vCPU0) variable. (4) The observer thread W also tries to take the lock. (5) W spins and is added to the same waiting list for vCPU0.

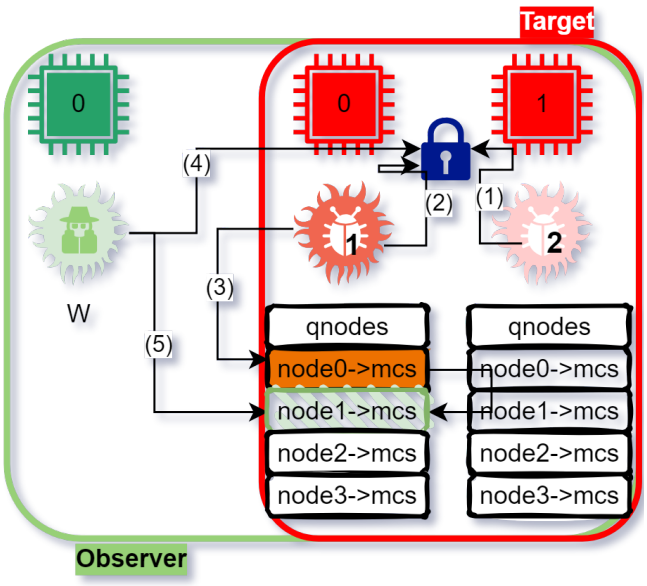


Figure 6: Remote spinlock.

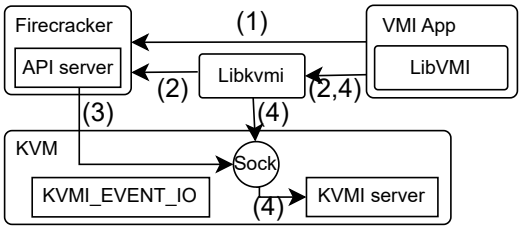


Figure 7: LibVMI usage with Firecracker.

E Porting LibVMI to Firecracker

LibVMI relies on the KVM-VMI project to provide general-purpose introspection capabilities. KVM-VMI is an extension of KVM that enables fine-grained introspection mechanisms. To perform introspection, LibVMI communicates with KVM (over the KVMI extension) through a dedicated socket.

Figure 7 illustrates how we integrated LibVMI with Firecracker. When a VMI application starts, it invokes the LibVMI initialization function, which connects to the VMM and requests the creation of an introspection socket (❶). With QEMU, LibVMI relies on libvirt's `qemu` driver for this step; however, to the best of our knowledge, no libvirt plugin exists for Firecracker. To overcome this limitation, we extended the Firecracker API to allow introspection-socket creation via HTTP requests. We also implemented a LibVMI Firecracker driver to handle (dis)connection with Firecracker.

Once the socket is created, LibVMI calls a `libkvmi` function to connect to the VMM introspection socket (❷). The KVMI protocol performs a three-step handshake to establish the connection. After the handshake completes, the VMM issues an `ioctl()` to KVM (❸) to pass the VM identifier and the socket file descriptor to KVMI. KVMI then spawns a kernel thread responsible for serving userspace KVMI requests.

From this point onward, the VMM is no longer involved in the introspection process: all communication occurs directly between LibVMI and KVM via the LibKVMI API (❹). This design ensures fairness in our evaluation.

When introspection ends, the VMI application sends a message to the VMM to terminate the KVMI session through a final `ioctl()` call.

F I/O Introspection with LibVMI

Concretely, we added a new command (`KVMI_EVENT_IO`) to KVMI, enabling a LibVMI-based observer to request that KVMI forward events corresponding to the page faults generated inside KVM when the guest attempts to access MMIO memory regions (i.e., reads or writes to memory-mapped I/O). When such an event occurs, KVMI returns to the observer the destination address. In the case of VirtIO-based devices, these addresses correspond to the in-guest VirtIO ring buffers. Using these addresses, the observer can rely on existing LibVMI memory read primitives to read and analyze the content of these buffers in order to identify the underlying disk read or write requests.

vBOIDs: Taming Chaos via Coarse-grained Scheduling Abstraction for Containers

Kaesi Manakkal[⋆], Nathan Daughety[†], Yu Sun[*], Marcus Pendleton[†], Hui Lu[⋆]
[⋆]*The University of Texas at Arlington,* [†]*Air Force Research Laboratory (AFRL),* [*]*Binghamton University*

Abstract

Today's high-density container deployment often descends into scheduling chaos, where fine-grained, per-thread scheduling decisions lead to thrashing and unpredictable performance. We present vBOIDs, a container scheduling system that tames this chaos with two key techniques: a coarse-grained BOID abstraction and a two-level balancing scheme. The BOID abstraction groups tasks into larger units to dramatically reduce scheduling churn, while two-level balancing coordinates global and local scheduling to ensure efficient resource utilization without oscillation. Our evaluation shows that vBOIDs improves the throughput of containerized microservices with thousands of threads by up to 3× under highly dynamic workloads compared with existing approaches, while incurring minimal overhead and yielding negligible performance impact even in high parallel workloads. vBOIDs effectively restores order to container scheduling, delivering high performance even under chaotic conditions.

1 Introduction

Cloud runtimes now deploy thousands of containers per machine [9, 18, 33, 50]. Production systems like Alibaba's RunD host over 2,500 containers on a single node [33], and kernel-bypass runtimes like Junction pack more than 3,000 instances on a 128 GB node [18]. This growth reflects the trend of finer-grained microservices, expanding core counts, and increasingly aggressive multi-tenant consolidation [14, 20, 29, 48].

However, despite being "lightweight", container-based deployments often perform even worse than full-fledged VMs – e.g., by more than 80% ($2). This counter-intuitive result reveals a fundamental abstraction breakdown: Containers leak their internal concurrency directly to the host kernel – a single containerized microservice can expand into hundreds of *host-visible* scheduling entities, forcing Linux's Completely Fair Scheduler (CFS) [41] to manage large, dynamic runqueues. For instance, the Hotel Reservation application [21], comprising 24 microservices, generates over 500 processes/threads [1] when containerized, compared to only 50 vCPUs for VMs.

To understand the consequences, we analyze container and VM deployments under identical loads ($2). We find that while the rates of intra-core switching (i.e., time-sharing a single core) are comparable, high-density containers incur substantially higher rates of inter-core migration (i.e., moving tasks across cores for load balancing) – by an order of magnitude. Because every container thread appears directly on the host's runqueues, CFS sees frequent queue-length imbalances and performs aggressive load-balancing routines that constantly scan cores and migrate tasks to equalize state.

Each inter-core switch forces the kernel and hardware to reconstruct execution state on a new core – by issuing TLB shootdowns (via IPIs), invalidating L1/L2 caches, perturbing LLC locality, and disrupting branch-predictor and prefetching. These penalties quickly accumulate when containers generate large numbers of runnable tasks, causing the scheduler to move threads faster than hardware locality can converge – a phenomenon we term *scheduling chaos*, where a modest increase in migration rates can trigger a breakdown of hardware locality, leading to severe system inefficiency.

Existing approaches mitigate this issue through resource partitioning and smarter scheduling. Techniques such as thread pinning, cache partitioning, and CPU frequency tuning [17, 31, 32, 35, 54] carve up shared resources, while advanced scheduling and learning-based systems infer sensitivity or demand at runtime and place tasks accordingly [13, 15, 19, 26, 34, 34, 43, 53]. These techniques often reduce tail latency and noisy-neighbor effects but do not reduce the number of host-visible scheduling entities. At large scales, thousands of container threads still compete for CPU, and the resulting scheduling complexity remains a key bottleneck.

These findings reveal a core tension in high-density container deployments: Containers achieve lightweight isolation

[1] For brevity, we use "thread" thereafter to refer to both Linux processes and threads, scheduled identically by the kernel.

DISTRIBUTION STATEMENT A. Approved for public release: Distribution unlimited. Case Number AFRL-2026-0584. Dated 09 Feb 2026

by exposing internal threads to the host, forcing fine-grained per-thread scheduling, but per-thread scheduling triggers aggressive load balancing, destroying hardware locality at scale.

In this paper, we introduce vBOIDs, a coarse-grained container scheduling abstraction paired with an intermediate balancing layer embedded in the kernel scheduler. Rather than exposing hundreds of threads to the host scheduler, vBOIDs bundles each container's threads into a small number of scheduling units, i.e., BOIDS, that act as virtual CPUs. By allowing the kernel to schedule only a few BOIDS per container, vBOIDs restores a stable scheduling hierarchy, sharply reducing cross-core thrashing and preserving cache locality. In addition, a lightweight, two-level balancing mechanism coordinates load at both the global and intra-container levels: the host scheduler migrates BOIDS across cores based on aggregate pressure, while an internal balancer redistributes threads among a container's BOIDS to avoid internal hotspots. vBOIDs operates entirely within the kernel and is fully compatible with existing container frameworks, requiring no changes to applications or orchestration systems (e.g., Kubernetes [5]). As a result, containers retain their agility while gaining the stability of coarse-grained scheduling.

We implemented vBOIDs in the Linux kernel and evaluated it on representative cloud-native microservices [21, 22]. For workloads that suffer from severe scheduling chaos, vBOIDs sharply reduces migration storms and cache disruption seen under the default Linux scheduler, achieving up to 3× higher throughput while still meeting SLOs. It yields stable throughput and latency on par with the ideal – yet impractical – pinned-core setup. In highly parallel workloads that demand rapid execution across many cores, vBOIDs incurs negligible overhead: it remains lightweight while matching the throughput of the fastest unpinned configuration, demonstrating that coarse-grained scheduling does not compromise parallelism. Overall, vBOIDs restores VM-like stability to containerized execution while preserving container efficiency.

2 Motivation

Modern cloud applications increasingly consist of dozens of fine-grained microservices communicating over lightweight RPCs [8, 23, 27, 38]. This decomposition enables independent deployment, rapid iteration, and elasticity at scale, and has fueled widespread adoption of container-based execution across platforms such as Borg [52], Kubernetes [5], and ECS [11]. Containers have thus become the dominant unit of computation: they package application code, dependencies, and runtime state while sharing the host OS for isolation, scheduling, and resource management.

However, this shift toward microservice granularity creates an unprecedented amount of host-visible concurrency. Even a modest service graph can expand into thousands of runnable threads once containerized. Crucially, containers expose this internal concurrency directly to the host scheduler. Unlike VMs, which virtualize CPU resources behind a small, stable set of vCPUs, containers allow the kernel to see every runnable thread. At high density, this forces the CPU scheduler to manage thousands of scheduling entities, exercise aggressive load balancing, and migrate tasks across cores at a rate far beyond its intended operating regime.

This pattern is widespread in production. Meta operates over 18,500 active microservices across 12M+ service instances, with deep service graphs that fan out into many short RPC handlers per user request [28]. Alibaba's microservice trace covers ∼20,000 services across 10,000+ bare-metal nodes [38], and its RunD runtime hosts over 2,500 secure containers per 384 GB node, starting more than 200 per second [33]; Junction similarly packs 3,000+ kernel-bypass instances on a single 128 GB node [18]. Standard runtime stacks further inflate these counts. Microservices commonly adopt thread-per-request or thread-pool models on top of gRPC, where each blocking RPC handler occupies an entire worker for its lifetime and concurrent fan-out forces pool sizes to scale with request concurrency [49]; service-mesh sidecars (e.g., Envoy/Istio) further add a multi-threaded proxy to every pod [27]. The result is that even a single co-located service contributes tens to hundreds of host-visible threads to the kernel scheduler, independent of application logic. In addition, Function-as-a-Service (FaaS) workloads exhibit the same pattern at a finer granularity – Azure Functions' production trace reports that 50% of functions execute in under one second, with invocation rates spanning eight orders of magnitude [48]; Caladan reports up to 230,000 core reallocations/second under microsecond-scale RPCs [19], and the same scheduling chaos has independently motivated the "Wasted Cores" study [36], Google's ghOSt [26], and the upstreaming of sched_ext [24].

2.1 Scheduling Chaos

We first illustrate the scheduling chaos problem using the Hotel Reservation application from DeathStarBench [21] that models a simplified online hotel booking platform. It consists of 24 interconnected microservices that collectively implement key functionalities such as user authentication, room availability search, reservation booking, payment processing, and customer account management. Each microservice is deployed independently and communicates with others via lightweight RPCs, making the application representative of real-world, latency-sensitive, multi-tier microservices.

We compared two deployment setups: Docker containers [1] and Firecracker micro-VMs [10]. In both setups, we ensured sufficient CPU resources per microservice. For containers, we constrained CPU allocations using `cgroups` [6], assigning 1 to 4 cores based on utilization; for micro-VMs, we similarly allocated 1 to 4 vCPUs. Note that the number of

DISTRIBUTION STATEMENT A. Approved for public release: Distribution unlimited. Case Number AFRL-2026-0584. Dated 09 Feb 2026

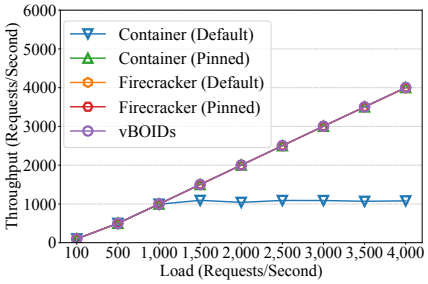


Figure 1: The throughput of Hotel Reservation as the load increases.

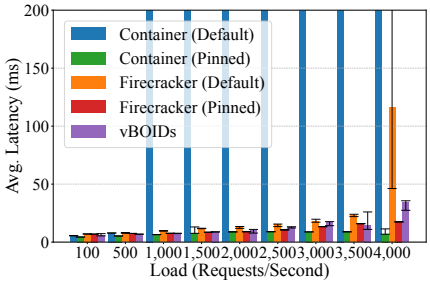


Figure 2: Average latency of Hotel Reservation under varying workloads.

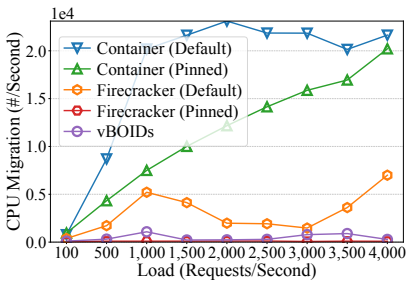


Figure 3: Process migrations of Hotel Reservation under varying workloads.

allocated CPU cores or vCPUs for each microservice was determined by (pre-)measuring its CPU utilization and rounding up to the nearest whole number to ensure sufficient resources.

We evaluated two scheduling schemes. In the `Default` scheme, the CPU scheduler, i.e., Linux CFS, freely schedules container threads or vCPUs on any available core, enabling system-wide load balancing. In the `Pinned` scheme, we manually pinned each thread or vCPU to a specific core or group of cores using CPU affinity. Although static pinning hinders load balancing, our manual pinning made sure that CPU resources remained sufficient for all microservices. It is important to note that manually pinning hundreds or thousands of short-lived threads is impractical in real deployments due to the highly dynamic nature of real-world workloads. The application ran on a server machine, equipped with two NUMA nodes, each with 32 cores (more details in §6).

Figure 1 and Figure 2 present the throughput and average latency of the Hotel Reservation application under increasing request rates. Under `Default` scheduling, the container-based deployment exhibits sharp latency spikes even at moderate load, eventually failing to meet SLOs (e.g., 200ms) as request rates increase – beyond 1k requests/second (RPS). Applying `Pinned` scheduling to containers substantially reduces latency growth and preserves stable performance across the full load range, enabling much higher sustainable throughput. In contrast, both micro-VM configurations (`Default` and `Pinned`) maintain consistently low and stable latencies across nearly all loads, with divergence appearing only at the highest request rates (around 4k RPS) – where the `Default` micro-VM configuration eventually violates SLOs.

These results highlight a clear and interesting contrast that containers are highly sensitive to CPU placement and benefit significantly from explicit core pinning, whereas micro-VMs maintain stable latency even under the default scheduler. Note that, in this paper, we chose two publicly available, widely-used microservices benchmarks – DeathStarBench [21] and Online Boutique [22] – to study the scheduling chaos problem. Both serve as conservative proxies for the production deployment – as discussed earlier, real-world deployments expose even higher host-visible concurrency, with correspondingly stronger migration storms, than what we report here.

2.2 Root Cause Analysis

To understand why containerized microservices exhibit unstable latency under `Default` scheduling while micro-VMs remain stable, we analyze microarchitectural metrics, context-switch behavior, and migration patterns at a load of 1k requests/second (Table 1), as this is the highest load on our system where all cases meet the SLO reliably.

First, Table 1 shows that `Default`-scheduled containers incur substantially higher TLB, L1-dcache, and LLC miss rates than any other configuration. In contrast, `Pinned` containers reduce misses across all levels of the cache hierarchy (e.g., L1 and LLC) as well as both dTLB and iTLB MPKI (Misses Per Kilo Instructions). Yet, micro-VMs exhibit a different pattern: although their virtualization stack introduces overhead (reflected in slightly higher LLC and TLB activity), the gap between `Default` and `Pinned` cases is small.

To further narrow down the cause, we examine context-switch behavior. An *intra-core switch* occurs when the scheduler time-slices runnable tasks on the same physical core. Microservice workloads naturally generate many such switches: short, bursty RPCs frequently block on I/O or remote calls and quickly wake up again, producing constant turnover among runnable tasks. Table 1 shows that both containers and micro-VMs exhibit similarly high intra-core switch rates, reflecting the fact that intra-core switching is driven primarily by the application's I/O-driven blocking and wake-up patterns; even in micro-VMs, the hypervisor delivers these events to the vCPUs (via VM Exits/Entries), resulting in comparable on-core scheduling activity at the host. Because these switches occur on the same core, they preserve L1/L2 cache warmth and TLB residency and avoid cross-core coordination. As a result, while intra-core switches contribute to overall scheduling volume, they impose low locality overhead.

In stark contrast, Figure 3 shows that `Default` scheduling causes containerized workloads to incur a disproportionately high number of inter-core migrations, and the rate grows

DISTRIBUTION STATEMENT A. Approved for public release: Distribution unlimited. Case Number AFRL-2026-0584. Dated 09 Feb 2026

Table 1: Running statistics for the Hotel Reservation application under the load of 1,000 requests/second. Note that cache and TLB miss numbers are reported as values normalized to the Container (Default) case.

Load (1,000 requests/s)	scheduling unit #	intra-core switches (# /second/core)	inter-core migr (# /second/core)	L1-dcache misses	LLC load/store misses	dTLB load/store misses	iTLB MPKI	CPU%
Container (Default)	495	2,677.26	315.47	1.00	1/1	1/1	1	26.25%
Container (Pinned)	495	2,698.85	117.13	0.84	0.33/0.18	0.22/0.35	0.82	13.17%
Firecracker (Default)	52	3,377.32	81.42	1.29	0.78/0.66	0.59/0.98	0.82	10.97%
Firecracker (Pinned)	52	3,558.76	1.50	1.31	0.31/0.21	0.45/0.86	0.86	9.21%
vBOIDs	52	2,535.97	16.85	0.86	0.28/0.20	0.28/0.44	0.75	10.71%

rapidly with load. Each migration moves execution to a different core (or NUMA node in the worst-case scenario), scattering working sets across cores and repeatedly discarding accumulated locality. This behavior directly elevates L1/L2 cache misses, perturbs LLC residency, and triggers TLB shootdowns, as reflected in Table 1. In addition, every cross-core move requires coordination between cores (e.g., updating runqueue state, enforcing memory-protection changes, and acquiring shared scheduler locks), further amplifying overhead. As load increases, the frequency of these migrations outpaces the hardware's ability to re-establish locality, producing the unstable execution behavior we term *scheduling chaos*. Under this, even moderate increases in request rate lead to rising tail latency and SLO violations when the system CPU utilization remains low – less than 30%.

As expected, `Pinned` scheduling substantially reduces inter-core migrations for containers – by roughly 60% at a load of 1k RPS (Figure 3) – stabilizing hardware locality and lowering cache and TLB miss rates (Table 1). This reduction manifests directly in the improved latency observed in Figure 2. However, because a container can expose hundreds of runnable threads and threads, it is difficult and impractical to pin each one to a specific physical core. Note that the existing affinity-based approach instead pins a container's all threads to a designated subset of cores. While this restricts scheduling freedom, it does not eliminate inter-core migrations: imbalances within the pinned core subset still trigger load balancing, particularly under heavy load.

Finally, `micro-VMs` exhibit different behavior. Rather than exposing hundreds of runnable entities to the host, micro-VMs present a small number of long-lived vCPUs (e.g., 52 total for this workload). Because the host scheduler sees far fewer host-visible scheduling units, runqueues remain stable and load-balancing routines have little opportunity, or need, to migrate vCPUs between cores. Consequently, both `Default` and `Pinned` micro-VM configurations incur very few migrations in Figure 3, and their cache/TLB profiles remain stable across scheduling modes. Without migration-induced disruption, micro-VMs sustain lower latencies than containerized workloads under `Default`, despite additional virtualization overhead of the guest kernel and hypervisor.

2.3 Insights

The analysis shows that the primary source of performance degradation across different configurations is excessive inter-core migrations (rather than the necessary, workload-driven intra-core switches). Like many existing approaches attempting to mitigate interference [17, 31, 32, 35, 54], thread pinning reduces migrations and restores locality but sacrifices flexibility or requires manual tuning. In contrast, micro-VMs expose only a handful of long-lived vCPUs, allowing runqueues to remain stable and maintaining hardware locality, but at the cost of full virtualization overhead.

This exposes a long-standing tension: stable placement is essential for maintaining low and consistent latency, but enforcing stability restricts dynamic load balancing, which is necessary for high resource utilization. It further raises a research question: can we decouple the granularity of thread-level execution from that of host-level migrations, thereby enabling containers to present only a handful of migration-stable units to the host while retaining their inherent flexibility and avoiding the costs of full virtualization?

3 Background

We focus on Linux's CFS, the default and widely deployed scheduler in production datacenters [37]. The core idea behind our approach – interposing a virtual scheduling abstraction between threads and physical cores – is general and can extend to emerging frameworks such as EEVDF [2] or BPF-based sched_ext [24]. However, we center our design, implementation, and evaluation on CFS, as it remains the practical substrate for most containerized cloud workloads today [52]. In the following, we provide the necessary background on CFS before introducing our solution.

Thread-granularity mismatch. The fundamental unit of execution in Linux is the thread (`task_struct`), represented in the scheduler as a scheduling entity. CFS operates on a strictly 1:1 threading model: It schedules individual threads onto physical cores based on their virtual runtime (`vruntime`). While effective for general-purpose computing, this model ignores the logical boundaries of modern containerized cloud applications. A container is simply a namespace grouping

DISTRIBUTION STATEMENT A. Approved for public release: Distribution unlimited. Case Number AFRL-2026-0584. Dated 09 Feb 2026

of hundreds of threads [6]. To the scheduler, these threads are independent actors; CFS has no structural concept of a "container" as an executable unit and cannot naturally enforce concurrency bounds – it cannot dynamically limit a container to executing on exactly N cores simultaneously (only static pinning), nor can it prevent the "split-brain" scattering of a container's threads across disjoint cores or NUMA nodes.

Task groups. In the Linux kernel, the abstract concept of a container is materialized as a `task_groups` (`cgroups` [6]). Container runtimes like Docker [1] map each container to a distinct `task_groups`. The `task_groups` aggregates the weights of its member threads to ensure proportional fairness at the group level. To limit CPU usage, the `task_groups` utilizes a "bandwidth control" mechanism. This is a temporal throttle: the kernel tracks aggregate usage and throttles all threads in the group once a global quota is exceeded. Crucially, the `task_groups` enforces temporal isolation (how much time a group gets) but lacks spatial isolation (where and how many cores a group occupies). Accordingly, `cfs_quota_us` is not a direct substitute for vBOIDs: it bounds CPU time over an interval, but not the number of cores a container can occupy concurrently. For example, a container with a quota of 2.0 CPUs can legally execute on 100 cores for 20ms (a massive burst) before being throttled. This bursty behavior degrades cache locality and induces tail latency for neighbors as we have discussed in §2. While a shorter quota period could approximate some of vBOIDs behavior, doing so would require workload-specific tuning; unless otherwise noted, our experiments use the default CFS quota and period settings.

Load balancing and PELT. For multicore systems, Linux invokes the load balancer periodically on scheduler ticks (e.g., every 1ms), whenever a CPU becomes idle, during task wakeups, or constraint changes to maintain fairness and minimize load imbalance. Migration decisions are driven by Per-Entity Load Tracking (PELT) [51], a geometric decay algorithm that implements an Exponentially Weighted Moving Average (EWMA) with a 32 ms half-life. PELT tracks the time each entity spends executing or waiting (runnable load) and aggregates these signals hierarchically – summing the load of individual threads into their parent `task_group`. During load balancing, the scheduler selects pairs of source and destination cores beyond certain load imbalances (e.g., ∼15% within a NUMA node and ∼30–50% across NUMA nodes). Then, the scheduler pulls threads from the source core runqueue to the destination core runqueue – a process called *migration*. While PELT provides an accurate view of container load, the standard load balancer still iterates over individual threads to resolve imbalances. In high-density environments, this $O(N)$ search space (i.e., total number of threads) becomes costly.

4 Design of vBOIDs

The design goal of vBOIDs is to address the fundamental mismatch between the massive thread-level concurrency exposed by containers and the much coarser physical parallelism of modern multicore hardware. Our design is guided by three principles: (1) *Restore placement stability through coarse granularity:* The root cause of scheduling chaos is that containers expose too many independent migration candidates. Restoring locality requires substantially reducing the scheduler's decision space with coarser-grained placement. (2) *Minimize hot-path interference:* The Linux scheduler's hot path must remain fast and wait-free; any design that introduces per-thread coordination, shared locking, or complex data-structure traversals in this path is untenable. (3) *Preserve work conservation and fairness:* Coarse-grained placement must not strand resources. The design must maintain work conservation while avoiding static pinning (underutilization) and bandwidth throttling (temporal idling).

vBOIDs satisfies these principles by introducing a new scheduling abstraction, the BOID, along with a new scheduling layer that decouples a container thread-level execution from the kernel's core-level placement. In the reminder of this section, we first formalize the BOID abstraction (§4.1), then present the scheduling hierarchy (§4.2), and finally describe the dual-path load balancing mechanisms (§4.3).

4.1 BOID Abstraction

As shown in Figure 4, we introduce BOID, Bound Object Integrated Dispatch [2], as a coarse-grained scheduling abstraction for containers. Conceptually, a BOID functions as a container-level vCPU, aggregating the container's threads into a single, logical entity that the host scheduler treats as a schedulable unit. To operate on BOIDS, vBOIDs adds an intermediate layer to the scheduling hierarchy. Instead of scheduling individual threads, the host scheduler (CFS) schedules BOIDS onto physical cores, while each BOID acts as a vehicle for the execution of its associated threads. Structurally, a BOID is a *first-class* kernel scheduling entity (i.e., backed by `struct boid` in Figure 4) that maintains a runqueue of its assigned threads. While this resembles the hierarchical scheduling of Linux task groups (cgroups), BOIDS enforce a strict serial execution invariant: regardless of how many threads are queued in a BOID, at most one may execute on a physical core.

This serialization makes the BOID the basic unit of concurrency for a container. The number of BOIDS assigned to a container dictates its maximum degree of parallelism. A container provisioned with N BOIDS is therefore limited to N simultaneous execution contexts – mirroring the behavior of

[2] The BOID term is borrowed from Craig Reynolds' BOIDS algorithm [46], which simulates the flocking behavior of birds. We adopt this to capture BOID's flock-like migration across cores.

DISTRIBUTION STATEMENT A. Approved for public release: Distribution unlimited. Case Number AFRL-2026-0584. Dated 09 Feb 2026

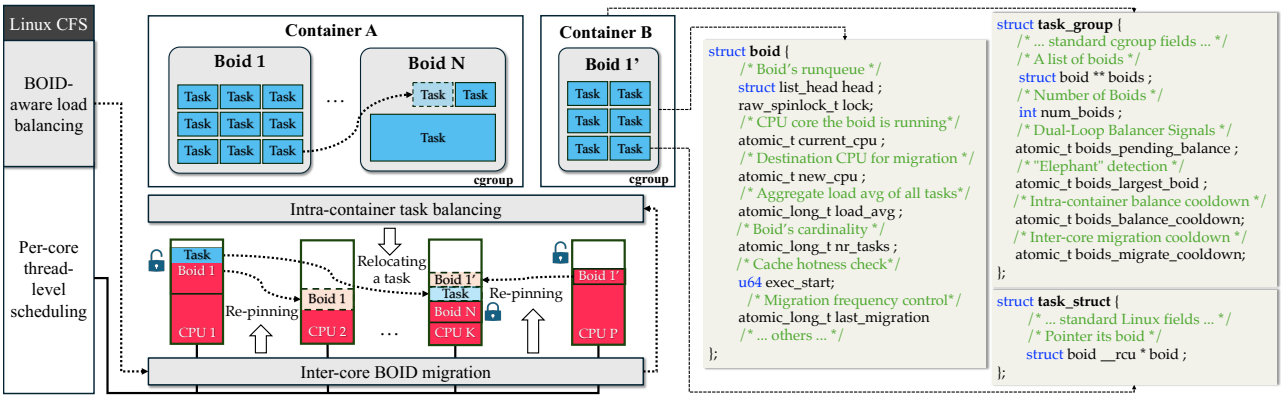


Figure 4: Architecture overview of vBOIDs (left). Key kernel data structures materializing the `boid` abstraction (right). The `boid` acts as the virtual cores (vCPUs) of containers, while `task_group` and `task_struct` are augmented to support the hierarchy.

a VM with N vCPUs. This yields a deterministic concurrency bound that eliminates the need for reactive, latency-inducing mechanisms such as CPU bandwidth quotas. Consequently, vBOIDs provides strong performance isolation by construction. In an unconstrained environment, a container with N BOIDs can utilize up to N physical cores when it has sufficient work. Under contention, the host scheduler balances load across runnable BOIDs, ensuring proportional CPU sharing at the granularity of the "virtual core" (i.e., BOIDs) rather than individual threads. Note that, vBOIDs preserves CFS's existing fairness machinery and applies it at BOID granularity: The load balancer treats BOIDs as the proportional-share entities, and per-core `vruntime` ordering still governs scheduling among threads sharing the same core. Two-level fairness properties therefore continue to hold under vBOIDs: (i) *inter-container fairness* – containers receive CPU shares proportional to their cgroup weights, summed across their BOIDs; and (ii) *intra-*BOID *fairness* – threads sharing a BOID's core compete via standard CFS `vruntime` ordering. By default BOIDs carry equal weight; they can be assigned heterogeneous weights to support fractional reservations or priorities.

4.2 Scheduling Hierarchy

While standard Linux CFS handles per-core execution and cross-core load balancing over the same set of threads, vBOIDs separates these concerns through a two-level hierarchy (Figure 4): the per-core scheduler time-shares threads assigned to each BOIDs (i.e., intra-BOID execution) on a single core, while vBOIDs's load balancer places BOIDs across cores to equalize system load (i.e., inter-BOID placement).

Intra-BOIDs execution (thread-level): At the lowest level, a BOID forms a strict execution boundary. Each BOID enforces this boundary through *dynamic affinity binding*: When a thread is assigned to a BOID (via `boids_add_task`), the kernel restricts its allowed CPU only to the physical core currently hosting that BOID. All threads whose BOIDs are mapped to a given core still compete in that core's runqueue under standard scheduling policies (e.g., `vruntime` ordering). This ensures that the hardware parallelism of a container is bounded by the number of BOIDs it is provisioned with – regardless of how many runnable threads it contains – while still preserving proportional fairness among threads sharing the same core. Notably, enforcing this invariant requires *no* changes to the scheduler's hot path; the per-core scheduler continues to select the next runnable entity on the core, guaranteed to belong to the resident BOID [3].

Inter-BOID placement (system-level): The core innovation of vBOIDs lies at the inter-BOID level, where the kernel's load balancer is extended to treat BOIDs as the atomic units of migration. vBOIDs first introduces a hierarchical load-tracking mechanism that aggregates the PELT signals (§3) of all threads within a BOID into a single, cohesive load metric (its aggregated `load_avg`). These aggregated metrics are then fed into the scheduler's existing load-balancing logic (§3), allowing the balancer to evaluate system-wide imbalance at the granularity of BOIDs rather than individual threads.

When the scheduler determines that migration is needed to equalize load, it invokes vBOIDs's *atomic migration* approach (§4.3). Instead of peeling off individual threads, the scheduler migrates the entire BOID by first atomically updating its target CPU. This operation informs the scheduler that all threads belonging to the BOID should execute on the destination core. Such BOID-level *re-pinning* is extremely lightweight and does not stall the scheduler's hot path. Threads are migrated lazily: they are removed from the source

[3] By updating each thread's allowed-CPU mask – when a thread is created or migrated – to reflect the core hosting its BOID, vBOIDs relies on the scheduler's existing CPU-selection logic. The built-in per-task CPU-admission check (`is_cpu_allowed()`), invoked during task placement (e.g., in `select_task_rq_fair()`), naturally enforces the execution boundary without requiring any changes to scheduler internals.

DISTRIBUTION STATEMENT A. Approved for public release: Distribution unlimited. Case Number AFRL-2026-0584. Dated 09 Feb 2026

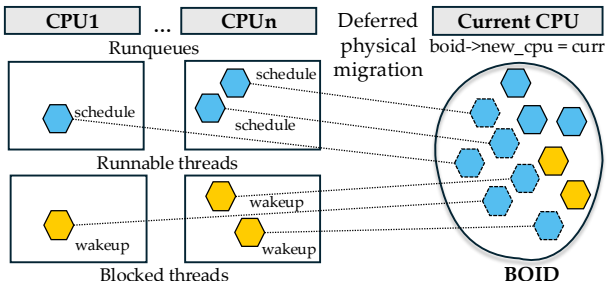


Figure 5: BOID flock-like migration.

core's runqueue and enqueued on the destination core only when they are subsequently scheduled or context-switched. Because the threads of a BOID are executed sequentially by design, the overhead of migrating the entire BOID is naturally constrained and amortized across its constituent threads.

This group-wise, atomic relocation is what gives BOIDS their flock-like behavior: placement decisions act on the collective, and individual threads simply follow the group's movement without independent negotiation. By treating the BOID as the indivisible unit of migration, the scheduler avoids the scheduling chaos of split-brain execution. The resulting hierarchy aligns the scheduler's placement decisions with the application's logical structure (i.e., BOIDS): the scheduler positions a container's BOIDS, and each BOID, once placed, carries its instruction stream on a single core.

4.3 Load Balancing

The introduction of BOIDS transforms the Linux load balancing problem from an $O(T)$ search over global threads to an $O(N)$ search over aggregate entities, i.e., BOIDS, where N represents the number of BOIDS and T denotes the number of threads – i.e., $N \ll T$. This reduction in dimensionality simplifies the scheduler's state space and yields more stable global placement decisions. vBOIDS augments Linux's default balancer with a complementary two-part migration mechanism: the *inter-core balancer* (§4.3.1) distributes BOIDS across cores to balance system-wide load, while the *intra-container balancer* (§4.3.2) distributes threads across a container's BOIDS to balance load within that container.

4.3.1 Inter-Core BOID Migration

Algorithm 1 illustrates the inter-core BOID migration logic. Traditionally, when the kernel load balancer detects a load imbalance between a source CPU (CPU_{src}) and a destination CPU (CPU_{dst}) – i.e., when `load(`CPU_{src}`) - load(`CPU_{dst}`)` exceeds the imbalance limit determined by PELT (§3) – the balancer begins scanning the runnable threads on CPU_{src}. It then migrates each movable thread (i.e., by relocating it to CPU_{dst}'s runqueue) to reduce the difference, stopping when

Algorithm 1 Inter-core BOID migration.

Require: X: source CPU; Y: destination CPU
Require: $L(X)$, $L(Y)$: load of CPUs X and Y
Require: $\theta_{\text{imbalance}} = L(X) - L(Y)$: load imbalance
Require: $\theta_{\text{imbalance}} > \theta_{\text{imbalance_t}}$ (determined by PELT §3)
Require: $L(b)$: load of BOID b
Require: $NRTasks(b)$: # tasks associated with BOID b
Require: $NRBoids$: # BOIDS in the taskgroup
Require: $R(b,X)$: residency time of BOID b on X
Require: θ_{hot}: cache-hot residency threshold
1: **for all** BOIDS b on CPU X **do**
2: **if** $\theta_{\text{imbalance}} \leq 0$ **then**
3: **break**
4: **if** $R(b,X) < \theta_{\text{hot}}$ **then**
5: **continue** ▷ Skip cache-hot BOIDS
6: **if** $L(b) > \theta_{\text{imbalance}}$ **then**
7: **if** $NRTasks(b) \geq 2$ **and** $NRBoids \geq 2$ **then**
8: CALL INTRA-CONTAINER-BALANCE
9: **if** $L(X) - L(b) > L(Y) + L(b)$ **then**
10: b.new_cpu $\leftarrow$ Y ▷ If imbalance is improved, perform migration asynchronously
11: $\theta_{\text{imbalance}} -= L(b)$

the loads converge or no additional movable threads remain. vBOIDS augments this path by evaluating BOIDS as the basic migration units, involving three main steps.

(1) *Candidate selection and locality* (line 4). The balancer first filters BOIDS based on execution history to avoid unnecessary cache disruption. For each BOID b, vBOIDS computes its residency time, $t_{res} = t_{now} - b.exec_start$. If t_{res} is below the configured hotness limit θ_{hot} (e.g., 500 μs), the BOID is considered cache-hot – its threads have recently populated the local L1/L2 caches. This BOID is thus skipped. The search is also topology-aware: a BOID is considered for migration only when CPU_{dst} lies within the same NUMA node as CPU_{src} or within the BOID's preferred node (a configurable parameter). This constraint enforces locality-preserving placement and avoids migrations that would incur unnecessary cache invalidation and NUMA penalties.

(2) *The "elephant" condition and delegation* (lines 6–8). A key challenge in coarse-grained balancing is the "elephant" BOID, i.e., an entity whose aggregate load $L(b)$ exceeds the imbalance gap between CPU_{src} and CPU_{dst} (i.e., $L(b) > \theta_{imbalance}$). Migrating such a BOID would only invert the imbalance, rather than resolve it, triggering the ping-pong effect. Instead of simply skipping these candidates, vBOIDS interprets them as a signal of *internal skew*. If an elephant BOID contains multiple runnable threads, the scheduler delegates the task-level balancing to the intra-container balancer (§4.3.2), which redistributes the BOID's internal threads across other, less-loaded BOIDS in the same container. This delegation enables vBOIDS to address fine-grained imbalance alongside the global BOID placement decisions.

(3) *Atomic actuation and convergence* (line 9-10). If a BOID satisfies the anti-thrashing condition – which guarantees that migrating it will reduce, rather than invert, the load

DISTRIBUTION STATEMENT A. Approved for public release: Distribution unlimited. Case Number AFRL-2026-0584. Dated 09 Feb 2026

Algorithm 2 Intra Container Load Re-balancing.

Require: Taskgroup T ▷ Current task group
Require: BOID E: Elephant BOID
Require: BOID S: Smallest BOID in taskgroup
Require: task t: Current task in BOID E
Require: $L(t)$: Load of task t
Require: $NRTasks(E)$: tasks associated with BOID E
Require: $IB(E,S)$: Imbalance between BOIDS E, S
Require: $\theta_{\text{imbalance}}$: Minimum load gap to trigger balancing

1: **if** $(E == S)$ **or**
2: $(IB(E,S) < \theta_{\text{imbalance}})$ **or**
3: $(L(t) > IB(E,S))$ **then**
4: return
5: **boids_remove_task**(E,t)
6: **boids_add_task**(S,t) ▷ Move task t from BOID E to S

gap – the scheduler commits to the move. The commitment is achieved using a *lazy, state-based* migration protocol that avoids taking multiple runqueue locks: Instead of immediately relocating all threads to the destination runqueue, the load balancer performs a single atomic CAS updating the BOID's target CPU state (`boid->new_cpu`). This completes the logical migration without touching per-task state.

Figure 5 illustrates this flocking-like behavior. The physical migration of threads is deferred to the context-switch path: runnable threads stay on their current runqueues, and blocked threads remain in their wakeup queues until they execute again. Upon scheduling or wakeup, each thread observes its BOID's new CPU state and migrates, causing the BOID to converge and ensuring that a thread always executes on the CPU assigned to its BOID. This two-phase mechanism allows the BOID to move atomically from the scheduler's perspective while its threads migrate asynchronously and incrementally, amortizing the cost and avoiding heavy runqueue locking. This behavior closely mirrors a VM's vCPU model, where threads execute only on the vCPU's assigned core, but differs in that BOID threads migrate lazily and independently rather than being moved as a preemptible virtual core.

Finally, the algorithm re-evaluates the load difference between CPU_{src} and CPU_{dst} (for each BOID on CPU X). If the imbalance has been neutralized ($\theta_{imbalance} \leq 0$), the search terminates (line 2), ensuring that the system performs the minimum number of migrations to restore equilibrium.

4.3.2 Intra-Container Task Balancing

By default, VBOIDS distributes threads evenly across BOIDS of a container at its initialization and when new threads are created during execution. However, as threads block, wake, and execute with varying intensity, the runnable load across BOIDS can drift over time. This imbalance breaks *work conservation* at the container level – an overloaded BOID stalls progress while underloaded BOIDS remain idle.

To address this internal skew, VBOIDS incorporates an *intra-container balancer*. Unlike VMs, which rely on a guest scheduler, VBOIDS performs this directly within the host scheduler. Unlike the default balancer, which migrates threads across arbitrary cores, VBOIDS confines redistribution strictly within the container's BOIDS. Hence, it does not reintroduce the "scheduling chaos" (§2). This intra-container balancer operates periodically or opportunistically during lifecycle events (e.g., task creation or resizing), and more importantly, cooperates with the inter-core BOID balancer.

Specifically, when a BOID is too "heavy" to migrate atomically, the inter-core balancer invokes intra-container redistribution (line 8, Algorithm 1). As Algorithm 2 further shows, once triggered, the intra-container balancer attempts to move a runnable task from the elephant BOID to the least-loaded BOID, provided two key conditions hold: (a) the load imbalance between the two BOIDS is significant enough to warrant intervention ($IB > \theta_{\text{imbalance}}$); and (b) the task is not too large, ensuring that migrating it reduces the gap rather than overshooting it ($L(t) \leq IB$). When these guards are met, the scheduler updates the task's BOID assignment (i.e., `p->boid`); the task physically migrates to the runqueue of the target BOID's core on its next wakeup or schedule event.

This pairing lets VBOIDS make progress on two fronts at once: it smooths/balances runnable load across a container's BOIDS while also reducing the size of an elephant BOID that the inter-core balancer cannot move. By shedding just enough tasks to another BOID – without overshooting – the mechanism both improves internal parallelism and turns an elephant BOID back into a migratable unit. This way, containers use their allocated BOIDS more efficiently, and BOID-level migration remains effective even under skewed workloads.

4.4 Putting It All Together

In summary, the VBOIDS architecture yields four key properties. *First*, it decouples thread execution from core placement: thread-level scheduling is handled locally preserving L1/L2 cache warmth, while expensive cross-core migrations are restricted to low-frequency and efficient structural load shifts. *Second*, it compresses the balancing state space from $O(T)$ threads to $O(N)$ BOIDS ($N \ll T$), dampening the "noise" of massive host-visible scheduling tasks and making placement decisions tractable. *Third*, its dual-path balancing approach – the inter-core balancer handles coarse-grained load drift by migrating BOIDS across cores, while the intra-container balancer corrects fine-grained skew by redistributing tasks within a container – forms a feedback system that maintains balanced utilization with minimal disruption to locality. *Finally*, it resolves the tension between isolation and utilization: unlike static pinning, which strands resources (i.e., low utilization), BOIDS remain *work-conserving* through the dual-path balancing mechanism; unlike bandwidth quotas, which enforce fairness through throttling (i.e., idling runnable tasks and in-

DISTRIBUTION STATEMENT A. Approved for public release: Distribution unlimited. Case Number AFRL-2026-0584. Dated 09 Feb 2026

creasing bursty latency), vBOIDs's concurrency bounding enforces fairness without throttling.

It is worth noticing that existing Linux task-group mechanisms control how much CPU time a container receives, but not the scheduler-visible unit of placement and migration. In contrast, BOIDS are introduced as a distinct abstraction to provide that unit: a small set of container-level virtual cores that bound concurrency and prevent thread-granularity balancing from reintroducing scheduling chaos.

5 vBOIDs Implementation

We implemented vBOIDs as a patch to the Linux 6.15 kernel, adding ∼2,000 lines of code [4] that touch the core kernel scheduler components (e.g., fair.c, pelt.c, and cgroup.c).

As depicted in Figure 4 (right), we integrate BOIDS directly into the kernel by augmenting the `task_group` structure. Each `task_group` maintains a dynamically allocated array of `struct boid` pointers. The BOID itself is a refcounted kernel object that tracks its aggregate load, current CPU, and a linked list of member tasks. To map threads to BOIDS, we modify `task_struct` to include a direct pointer to its assigned BOID. This pointer is managed via Read-Copy-Update (RCU) to allow lockless dereferencing during the scheduler's hot loop (`is_cpu_allowed`). Further, the management plane is exposed via a new `cgroup` interface file, `cpu.boids`. Writing an integer N to this file triggers a resizing operation that reallocates the BOID array and redistributes existing tasks round-robin – the existing processes within the container are distributed evenly across the new BOIDS. We determine the initial CPU affinity for a BOID based on the CPU where its first assigned process was previously resident. While integrating these data structures establishes the structural hierarchy of vBOIDs, enforcing the BOID abstraction within the scheduler's highly concurrent hot path required overcoming several challenges related to concurrency, complexity, and stability.

(1) Tunable hysteresis and locality preservation. Workloads often exhibit conflicting scheduling requirements, e.g., high-parallel tasks benefit from immediate dispatch to idle cores to minimize queuing delay, whereas latency-sensitive applications often require preserving cache locality and minimizing rebalancing overhead. To allow administrators to optimize for these conflicting goals, we introduce two tunable parameters. First, `boids_migrate_cooldown` prevents tasks from bouncing between cores during rapid interactions (by enforcing a minimum interval between migrations), which preserves cache locality. Further, `boids_balance_cooldown` stops the scheduler from overreacting to short load spikes (by throttling the frequency of load balancing events), ensuring we do not waste cycles rebalancing BOIDS for work that finishes before the migration is even complete. These parameters are enabled at the per-container level (`struct task_group`), enabling the scheduler to enforce custom migration and balancing for different workloads running on the same machine. By default, they are set to zero (i.e., no delays), which works well in most cases observed in our experiments (§6).

(2) Lazy, state-based BOID migration. A major hurdle was enforcing atomic migration without holding the runqueue locks of multiple cores simultaneously, which would induce prohibitive contention. We solved this by implementing a two-phase, flock-like lazy migration. In the load balancing path (`detach_tasks`), we do not physically move tasks. Instead, we perform a lightweight atomic CAS on the BOID's metadata (`boid->new_cpu`). The physical migration is deferred to the context switch path: we inserted a hook, `boids_post_migrate`, into `finish_task_switch`. When a thread is scheduled in, it checks its BOID's state; if the BOID has moved, the thread updates its own runqueue (`cfs_rq`) and the current cpu pointers (`current_cpu`) immediately. This allows the BOID to logically move instantly while its threads physically catch up asynchronously, eliminating the need for global runqueue locking.

(3) Zero-cost hierarchical PELT. Accurate load balancing requires aggregating the load of all threads in a BOID, but iterating over thread lists during balancing is $O(N)$. We modified the PELT update in the current PELT (§3) to perform differential aggregation (`__update_load_avg_se`). Whenever a thread's load average changes, the delta is atomically propagated to its parent BOID (`atomic_long_add`). This ensures a mathematically consistent view of aggregate demand in $O(1)$ time, without ever traversing the member threads.

(4) Lockless affinity and RCU synchronization. Enforcing strict BOID affinity (`is_cpu_allowed`) requires accessing shared BOID state during the scheduler's most performance-sensitive loops. We used RCU heavily to manage the lifecycle of BOID structures (`rcu_dereference(p->boid)`), ensuring that task placement (`select_task_rq_fair`) remains wait-free. Further, updating BOID affinity from atomic contexts (e.g., during `detach_tasks`) prevents the use of locks (`boid->lock`). Instead, we use an atomic integer to specify a BOID's current core. This primitive allows us to update a BOID's affinity atomically, ensuring no lock contention, and that the operation cannot stall the CPU.

(5) Provisioning and lightweight resizing. Provisioning BOIDS is conceptually analogous to provisioning vCPUs for a VM: the number N of BOIDS per container is workload- and deployment-dependent, and we set N to each container's expected concurrent CPU demand – the same value an operator would pick for a VM's vCPU count. Unlike VM vCPUs, however, vBOIDs allows N to be resized at runtime: the BOID array is reallocated in place, threads are re-pinned via RCU, and the operation requires neither a container restart nor thread recreation. As a result, N becomes a tunable runtime parameter rather than a deployment-time constant, and vertical

[4] Project website: https://github.com/kmanakk1/vboids

DISTRIBUTION STATEMENT A. Approved for public release: Distribution unlimited. Case Number AFRL-2026-0584. Dated 09 Feb 2026

Table 2: Characteristics of the four microservices in DeathStarBench [21] and Google Online Boutique [22].

Applications	CPU intensity	Latency sensitivity	Fan-out/ graph depth	# of services
Hotel Reserv	moderate	moderate	low	24
Social Network	low	high	high	27
Media Services	high	low	high	33
Online Boutique	moderate	moderate	moderate	11

scaling [45] – i.e., growing or shrinking a container's effective parallelism in place – becomes a lightweight operation. This makes N amenable to dynamic, load-driven adaptation; we leave a full study of such policies to future work.

6 Evaluation

Our evaluation answers a key question: *to what extent does* vBOIDs *restore placement stability and improve end-to-end performance for high-density containerized microservices?* We compared vBOIDs against various Linux configurations across a broad suite of microservice benchmarks.

Testbed. All experiments were conducted on a two-socket server equipped with two Intel Xeon Gold 6430 processors, 64 GB of DDR5 memory, and a Micron 7450 NVMe SSD formatted with ext4. Hyperthreading was disabled to avoid SMT-induced interference with a total of 64 physical cores, 32 each NUMA node. All system resources were provisioned to eliminate non-CPU bottlenecks. The machine ran Ubuntu 22.04 and our modified Linux 6.15.0 kernel, with and without vBOIDs enabled. Container experiments used Docker [1] as the container runtime; comparisons to micro-VM isolation employed Firecracker [10] configured with matching constraints for CPU and memory resources.

Workloads. We used both DeathStarBench [21] and Google Online Boutique [22] – widely used benchmarks for studying end-to-end performance in cloud-native microservices applications. DeathStarBench captures realistic service-graph compositions, request fan-out, and cross-service communication patterns typical of large-scale online services. Our experiments focus on three representative applications – Hotel Reservation, Social Network, and Media Services. Boutique [22], on the other hand, features 10 stateless services connected via synchronous RPCs and multiple aggregation points in the frontend. They collectively exercise a spectrum of CPU intensity, branching factors, and latency sensitivities, as summarized in Table 2.

Baselines for comparison. We evaluated vBOIDs against four baselines. (1) *Containers (Default):* each microservice runs as a Docker container, with CPU allocations constrained using `cgroups` and typically assigned 1–4 cores based on measured utilization; threads can execute on any available physical core (no pinning). (2) *Firecracker (Default):* each microservice runs inside a Firecracker micro-VM [10], provisioned with 1–4 vCPUs to match the container allocations; vCPUs can freely migrate across cores. (3) *Containers (Pinned)* and (4) *Firecracker (Pinned):* container threads or Firecracker vCPUs are manually pinned to specific cores or core groups using CPU affinity, ensuring the same CPU resources as those in (1) and (2). (5) vBOIDs*:* our approach, configured with the same number of BOIDS as the number of cores allocated to each container or vCPU count in the Firecracker configurations. These baselines are referenced throughout the evaluation for brevity. We collected workload- and system-level performance metrics – e.g., throughput, average and tail latency, CPU utilization, scheduling frequency, and cache activity.

6.1 End-to-End Performance

We first demonstrate the end-to-end performance using Hotel Reservation, Social Network, and Media Services under various configurations. We generated load to them using `wrk2` configured with an exponential inter-arrival distribution for requests, `20` threads, and a total of `60` connections. Each experiment ran for 60 seconds after a brief warm-up. Unless otherwise noted, we set `boids_balance_cooldown` and `boids_migrate_cooldown` to 0 to allow unthrottled balancing activity; cooldown effects are explored separately in our sensitivity analysis. To ensure fairness, resource provisioning is normalized across all configurations: the number of BOIDS, vCPUs, and physical cores is matched for vBOIDs, micro-VMs, and Docker containers, respectively.

Hotel Reservation was our motivating case (§2), where Container (Default) triggered the migration storms and locality destruction. We return to this workload to evaluate whether vBOIDs eliminates these effects in a real deployment.

Figure 1 reports end-to-end throughput under increasing RPS load (requests per second), measured only within the range where latency satisfies a 200ms SLO. As load increases, *Container (Default)* collapses early, saturating before 1k RPS. *Container (Pinned)* and both Firecracker configurations extend beyond 4k RPS due to reduced inter-core migration. vBOIDs sustains high throughput up to 4k RPS. Figure 2 shows the corresponding latency trends. *Container (Default)* exhibits sharp spikes around 1k RPS, while vBOIDs remains close to the manually pinned case, maintaining low latency through 4k RPS.

These performance trends align with the inter-core migration data in Figure 3 and micro-architectural data in Table 1, collected using `perf` over a 10-second steady-state window (for 1k RPS). First, *Container (Default)* incurs the highest inter-core migrations and cache/TLB miss rates, whereas vBOIDs dramatically reduces migration frequency and increases cache efficiency. Further, vBOIDs matches the migration behavior of *Firecracker (Default)*, recreating vCPU-like

DISTRIBUTION STATEMENT A. Approved for public release: Distribution unlimited. Case Number AFRL-2026-0584. Dated 09 Feb 2026

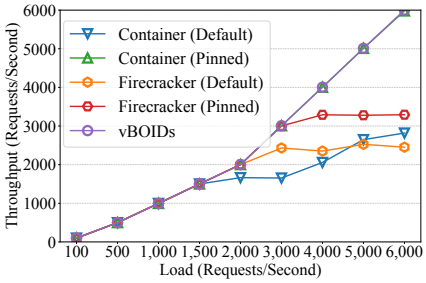


Figure 6: Throughput of Social Network as the load increases.

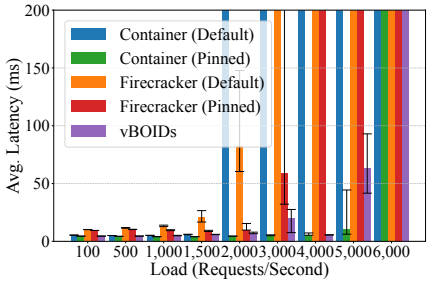


Figure 7: Average latency of Social Network under varying workloads.

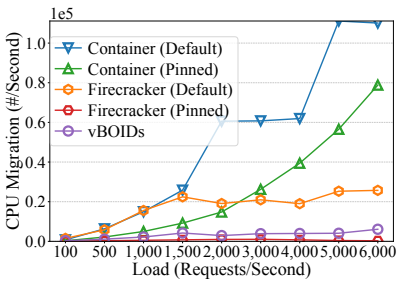


Figure 8: Process migrations of Social Network under varying workloads.

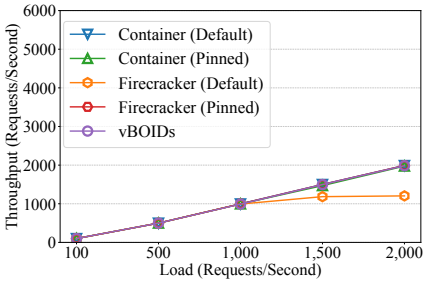


Figure 9: Throughput of Media Microservices as the load increases.

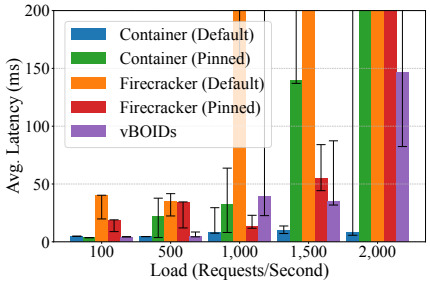


Figure 10: Average latency of Media Microservices under varying workloads.

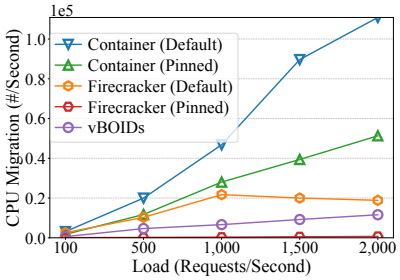


Figure 11: CPU Migrations for Media Microservices under varying workloads.

execution using BOIDS. Last, vBOIDs incurs even fewer inter-core migrations than *Container (Pinned)*, where threads remain confined to a subset of cores and still migrate frequently within that group, especially when load is high.

Overall, vBOIDs improves throughput by *over 3*× compared to *Container (Default)* and matches the performance of *Container (Pinned)* and *Firecracker (Pinned)* – all without manual pinning and while preserving full container flexibility. vBOIDs is lightweight: its overall CPU% (Table 1) is 40% of *Container (Default)* and 80% of *Container (Pinned)*.

Social Network, compared to Hotel Reservation, features a deeper, more highly branched service graph and generates a larger runnable thread set when containerized.

For throughput, as reported in Figure 6, *Container (Default)* saturates early, flattening near 2k RPS. Both vBOIDs and *Container (Pinned)* perform substantially better, sustaining close to 5k RPS, still meeting the SLO, while both Firecracker configurations follow the pinned curve till up to 3k RPS. Figure 7 captures the latency behavior. Consistent with its early saturation, *Container (Default)* experiences sharp latency spikes around 2k RPS while both Firecracker configurations can sustain till 3k RPS. In contrast, vBOIDs and *Container (Pinned)* remain stable and low-latency up to 5k RPS. Figure 8 and Table 3 explain this behavior: At 1k RPS, *Container (Default)* exhibits extremely high inter-core migrations and elevated cache/TLB miss rates, whereas vBOIDs cuts migrations by 85% and with lower cache/TLB misses.

This demonstrates that vBOIDs's coarse-grained scheduling remains effective even for deep, fan-out–heavy service graphs – not only outperforming *Container (Default)* but also surpassing both pinned and default Firecracker configurations. vBOIDs behaves just like *Container (Pinned)* in terms of performance (both throughput and latency), yet achieves this without any manual pinning. Meanwhile, vBOIDs maintains low CPU utilization, again underscoring its efficiency.

Media Services differs from the previous two workloads in that it is substantially more CPU-intensive and issues far fewer fine-grained RPCs. During execution, it rapidly spawns many short-lived threads that aim to run immediately on any available core. Hence, this workload benefits from maximizing hardware parallelism, making *Container (Default)* – with its unconstrained access to all cores – particularly effective.

Figure 9 shows that all, except *Firecracker (Default)*, scale up to 2k RPS. In Figure 10, as Media Services benefits from maximal hardware parallelism, the unpinned cases, *Container (Default)* and vBOIDs, outperform the pinned ones, *Container (Pinned)* and *Firecracker (Pinned)*, reflected in the lower latencies. vBOIDs achieves this by adaptively issuing more inter-core migrations (Figure 11 and Table 3) than those in Hotel Reservation and Social Network, enabling it to better exploit available hardware parallelism. These results show that vBOIDs delivers performance on par with the most ef-

DISTRIBUTION STATEMENT A. Approved for public release: Distribution unlimited. Case Number AFRL-2026-0584. Dated 09 Feb 2026

Table 3: Running statistics for Social Network and Media Services under the load of 1,000 requests/second.

Load (1,000 requests/s)	scheduling unit #	intra-core switches (# /second/core)	inter-core migr (# /second/core)	L1-dcache misses	LLC load/store misses	dTLB load/store misses	iTLB MPKI	CPU%
Social Network								
Container (Default)	950	2,273.91	233.44	1.00	1/1	1/1	1	9.73%
Container (Pinned)	950	2,214.03	77.72	0.94	0.62/0.39	0.36/0.54	0.61	7.22%
Firecracker (Default)	61	6,437.88	243.14	2.12	1.46/1.08	1.66/2.94	0.57	15.86%
Firecracker (Pinned)	61	6,407.99	8.21	2.05	0.84/0.57	0.85/2.26	0.59	12.23%
vBOIDs	61	2,011.22	34.22	1.01	0.69/0.47	0.50/0.72	0.65	7.98%
Media Services								
Container (Default)	936	6,031.39	731.21	1.00	1/1	1/1	1	17.07%
Container (Pinned)	936	5,617.54	437.88	1.02	0.81/0.56	0.27/0.53	0.58	14.58%
Firecracker (Default)	59	7,454.12	339.25	1.43	1.77/1.29	1.50/2.84	0.49	18.29%
Firecracker (Pinned)	59	8,883.83	5.18	1.62	0.63/0.44	0.92/2.45	0.42	18.82%
vBOIDs	59	5,439.03	103.55	0.99	0.94/0.60	0.26/0.50	0.58	14.02%

fective configurations, even when high inherent parallelism reduces the advantage of migration control.

In summary, for workloads dominated by scheduling chaos (Hotel Reservation and Social Network), vBOIDs' coarse-grained BOID abstraction sharply reduces destructive inter-core migrations and preserves locality, achieving performance comparable to pinned configurations. In workloads with high inherent parallelism (Media Services), vBOIDs remains lightweight while matching the fastest unpinned configuration. These results show that vBOIDs delivers VM-like stability while retaining container flexibility and efficiency.

6.2 CPU Balancing and Stability

In Figure 12 (bottom), because the CPU scheduler now schedules at BOID granularity, vBOIDs exhibits VM-like per-core average CPU utilization – naturally uneven across cores due to the smaller number of schedulable units. Figure 12 (top) shows that vBOIDs's flocking-like migration behavior – where BOIDS move logically atomically but physically asynchronously – produces container-like total CPU stability, maintaining smooth, steady utilization comparable to thread-level container scheduling and avoiding the sharp oscillations seen in VMs. Together, it shows that vBOIDs combines the most desirable behaviors of VMs and native containers.

Figure 15 further presents a per-core view of CPU utilization over time under vBOIDs, with each line representing a single physical core. Individual cores exhibit natural variability driven by BOID migrations and intra-container task balancing, yet most cores remain stably clustered around 50% utilization, showing that vBOIDs maintains healthy load diffusion and remains fully *work-conserving* rather than creating persistent hotspots or underutilized cores. Consistent with this pattern, vBOIDs performs only a modest number of BOID migrations (e.g., roughly 600 over 60 s), relying instead on more frequent fine-grained intra-container task redistributions (e.g., about 16k events). This confirms that vBOIDs preserves global stability through coarse-grained BOID placement while using intra-container balancing to smooth local fluctuations without inducing system-wide oscillation.

6.3 Sensitivity and Scalability

We evaluated the impact of the inter-container balancing cooldown (`boids_balance_cooldown`) and inter-core migration cooldown (`boids_migrate_cooldown`) on two distinct workload profiles: Social Network (RPC-heavy) and Media Services (CPU-bound), as shown in Figure 13.

Sensitivity. For RPC-heavy workloads like Social Network, active intra-container balancing is essential. Because the workload is not fork-heavy, tasks are initially well distributed across BOIDS but remain long-lived, causing substantial load drift over time. Disabling balancing (`boids_balance_cooldown=-1`) or delaying it via a large cooldown degrades performance as BOIDS gradually become imbalanced. In contrast, the migration cooldown has little effect: the workload's moderate CPU intensity results in fewer inter-core migrations, leaving system behavior largely insensitive to migration throttling. Consequently, a low balancing cooldown is ideal, ensuring that vBOIDs aggressively corrects internal skew as it emerges.

In contrast, Media Services exhibit high-frequency forking of short-lived, CPU-intensive threads. This workload is largely insensitive to parameter tuning for two reasons. First, vBOIDs's placement policy assigns new forks to the least-loaded BOID, ensuring the container remains inherently balanced without requiring reactive intervention from the intra-container balancer. Second, because the workload is CPU-bound, BOIDS often remain heavy ("elephants"), which naturally suppresses inter-core migration regardless of the cooldown setting. Consequently, performance variance in this test is driven more by the high load intensity (`1.5k` RPS) and run-to-run noise than by specific scheduling parameters.

Scalability. We subjected the Social Network workload (fixed at 2k RPS) to increasing levels of background containers, simulating a high container deployment environment. We intro-

DISTRIBUTION STATEMENT A. Approved for public release: Distribution unlimited. Case Number AFRL-2026-0584. Dated 09 Feb 2026

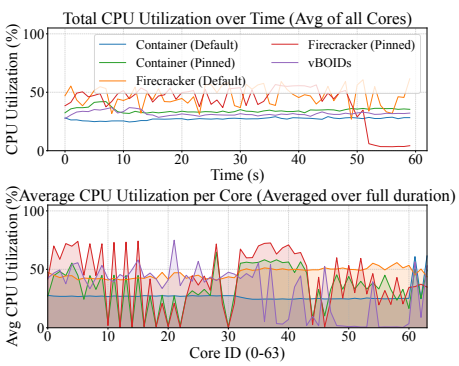


Figure 12: CPU% for Social Network under 5k RPS: total CPU% over time (top) and average CPU% per core (bottom).

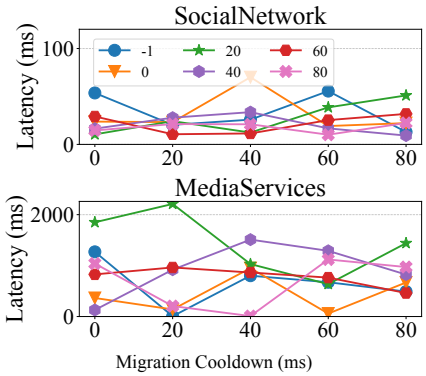


Figure 13: Sensitivity. Media Services at 1.5 RPS. Social Network at 4.5 RPS. Each line is a different balance cooldown.

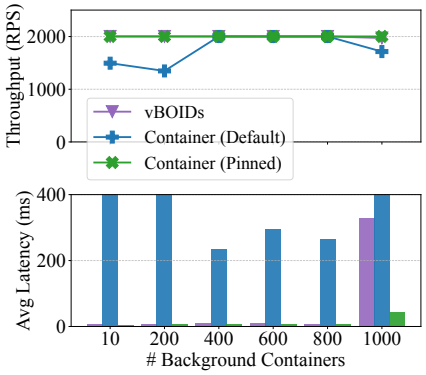


Figure 14: Scalability test using Social Network (at 2k RPS) with increasing number of background containers.

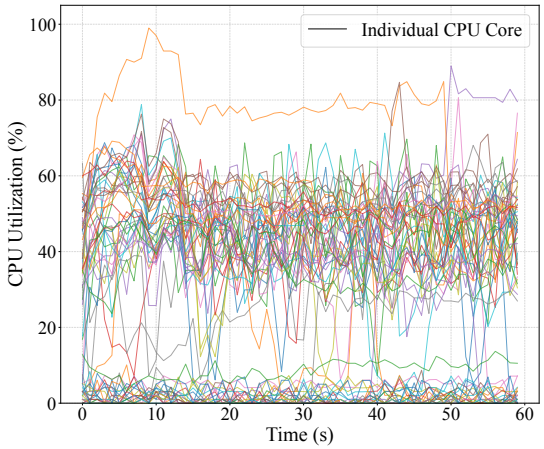


Figure 15: Per-core CPU% over time for Social Network under load of 5k RPS.

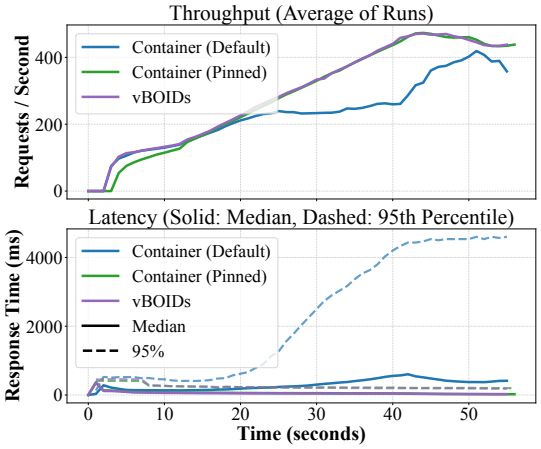


Figure 16: Performance of Boutique with linear ramp-up of 50 users per second.

duced up to 1,000 "noise" containers, each provisioned with a single BOID and running a light workload (1–2% single core utilization). We compared VBOIDS against both default and pinned container baselines (i.e., we only pinned Social Network BOIDS, not the background container ones). As shown in Figure 14, both VBOIDS and the pinned configuration successfully maintain the target throughput even as background noise scaled to 1,000 containers, where the average latency under VBOIDS starts to increase.

6.4 Kubernetes

To evaluate VBOIDS in a production setting, we deployed the Google Online Boutique benchmark on Kubernetes [5], ramping load to 2k concurrent users via Locust [7]. The Locust workload emulates randomized user actions in the store, including product browsing, currency changes, cart views, add-to-cart operations, and checkout; these actions generate a realistic mix of read-heavy requests (e.g., `/product/<id>`, `/cart`) and state-mutating requests (e.g., `/cart`, `/cart/checkout`). In our setup, Online Boutique exposes 359 scheduler-visible units in the standard containerized deployment, which VBOIDS reduces to 31 BOIDS. We compared VBOIDS (2 BOIDS/pod) against standard unpinned pods and manually pinned pods (2 CPUs/pod). While average latencies were comparable, the unpinned baseline suffered severe tail latency degradation (p95 exceeding 4000ms). Conversely, VBOIDS matched the throughput and tail latency (under 500 ms) of the pinned configuration, achieving pinning-grade stability without manual intervention.

7 Related Work

Resource partitioning and QoS-aware scheduling. To mitigate interference in co-located environments, prior work [13, 15–17, 31, 32, 35, 39, 55] has largely focused on resource partitioning and sensitivity-aware placement. Systems such as Heracles [35] and PARTIES [13] rely on feedback loops to

DISTRIBUTION STATEMENT A. Approved for public release: Distribution unlimited. Case Number AFRL-2026-0584. Dated 09 Feb 2026

dynamically partition resources and protect latency-critical services from best-effort batch jobs. Similarly, Paragon [15] and Quasar [16] use collaborative filtering to classify workloads and avoid placing contentious applications on the same node. While these approaches reduce "noisy neighbor" interference via hardware isolation, they do not address the *scheduling overhead* itself. In contrast, vBOIDs fundamentally alters how the scheduler perceives container concurrency to prevent scheduling thrashing.

User-space and extensible scheduling. A growing body of work [18, 19, 24, 26, 29, 30, 42–44] argues for moving scheduling logic out of the kernel to support the high-frequency requirements of microservices. Systems such as Caladan [19], Shinjuku [30] and Arachne [44] implement user-space schedulers that bypass the kernel entirely to achieve microsecond-scale tail latency. More recently, Google's GhOSt [26] and the Linux `sched_ext` [24] framework utilize eBPF to delegate scheduling policies to userspace agents, allowing for rapid policy modifications. While effective, these approaches impose significant adoption barriers: they often require recompiling applications against custom libraries (e.g., Arachne and Caladan) or managing complex userspace agents (e.g., GhOSt). vBOIDs acts as a middle-ground: it retains the transparent, drop-in nature of the standard kernel scheduler, while introducing schemes to mitigate *scheduling chaos*.

Lightweight virtualization. Micro-VMs, such as Firecracker [10] and Kata Containers [4] address multi-tenancy by enforcing strong isolation boundaries [3, 4, 10, 12, 25, 33, 40, 47]. They expose a fixed set of vCPUs to the host scheduler, rather than all guest processes/threads, which naturally mitigates *scheduling chaos* and stabilizes load balancing. However, they also introduce full virtualization overhead and reduce flexibility in resource over-subscription. vBOIDs effectively extends the vCPU abstraction to native containers, providing the stability of a vCPU without the performance penalty of full virtualization or the rigidity of fixed resource constraints.

8 Discussions

vBOIDs bridges the gap between spatial partitioning (pinning) and temporal throttling (CFS quotas). vBOIDs adopts *spatial concurrency bounding* – by mapping threads onto a fixed set of migratable BOIDs, it bounds a container's parallelism – while preserving *temporal flexibility* through a two-level scheduling hierarchy. This provides pinning-like isolation benefits and quota-like fairness benefits.

However, *short-lived* microservices that rely on rapid forking pose unique placement challenges. Placing child threads on the parent's BOID maximizes locality but can temporarily bottleneck execution on a single core. Because these tasks are short-lived, they may complete before the kernel's load balancer can detect and correct the imbalance. vBOIDs mitigates this issue by adopting a parallelism-first placement policy: newly created threads are assigned to the least-loaded BOID in the taskgroup, enabling immediate load spreading and reducing transient hotspots. This design is particularly effective for bursty workloads with enough available BOIDS to absorb short-term parallelism. However, vBOIDs may be less beneficial for extremely short-lived, highly parallel workloads whose execution time is too brief to amortize placement and cache-warming effects, or when the number of provisioned BOIDS is significantly lower than the workload's instantaneous parallelism. In such cases, load spreading can be constrained by BOID availability, and the latency benefit may be reduced.

This can be further addressed via custom policies. The stable execution provided by vBOIDs offers an ideal substrate for delegating fine-grained scheduling decisions to userspace frameworks, like GhOSt [26] or sched_ext [24]. Future work could leverage this to implement custom, application-specific policies atop their assigned BOIDS, effectively insulating their internal logic from the host placement. While we currently implemented vBOIDs by modifying the kernel code (5), we are also exploring the use of eBPF or sched_ext for a more modular and easily deployable implementation.

9 Conclusion

We presented vBOIDs, a container scheduling system that restores stability to high-density container deployments by raising the scheduler's placement granularity from individual threads to BOIDS. By introducing BOIDS as container-level virtual scheduling units, vBOIDs decouples thread-level execution from host-level placement, allowing the kernel to make fewer, more stable, and more locality-preserving scheduling decisions. vBOIDs combines this coarse-grained abstraction with a two-level balancing mechanism. The inter-core balancer migrates BOIDS as aggregate units to reduce global imbalance, while the intra-container balancer redistributes tasks across a container's BOIDS to preserve work conservation and avoid internal hotspots. This design provides VM-like placement stability without requiring full virtualization, manual pinning, or rigid CPU partitioning, meanwhile preserving the flexibility and efficiency of containers that are attractive for cloud-native deployment. Our evaluation shows that vBOIDs substantially reduces migration-induced locality disruption and improves end-to-end performance for chaos-prone microservice workloads. vBOIDs's abstraction provides a practical foundation for future adaptive policies, including dynamic BOID provisioning and user-extensible scheduling.

10 Acknowledgments

We thank our shepherd and the anonymous reviewers for their helpful feedback. This work was supported by the Air Force Research Laboratory (AFRL) under Award FA8750-25-C-B038, and by the National Science Foundation (NSF) under Awards CCF-2415473 and CNS-2415774.

DISTRIBUTION STATEMENT A. Approved for public release: Distribution unlimited. Case Number AFRL-2026-0584. Dated 09 Feb 2026

References

[1] The docker container. Accessed: 2025-11-04. URL: https://www.docker.com/.

[2] Eevdf scheduler. Accessed: 2025-12-03. URL: https://www.kernel.org/doc/html/latest/scheduler/sched-eevdf.html.

[3] Google gvisor. Accessed: 2025-11-15. URL: https://github.com/google/gvisor.

[4] Kata containers. Accessed: 2025-12-03. URL: https://github.com/kata-containers.

[5] Kubernetes. Accessed: 2025-11-15. URL: https://kubernetes.io/.

[6] Linux control groups. Accessed: 2025-12-05. URL: https://www.kernel.org/doc/Documentation/cgroup-v1/cgroups.txt.

[7] Locust: An open source load testing tool. Accessed: 2025-11-15. URL: https://locust.io/.

[8] Microservices adoption statistics. Accessed: 2025-11-15. URL: https://codeit.us/blog/microservices-use-cases.

[9] Overview of azure service fabric. Accessed: 2025-11-15. URL: https://learn.microsoft.com/en-us/azure/service-fabric/service-fabric-overview.

[10] Alexandru Agache, Marc Brooker, Alexandra Iordache, Anthony Liguori, Rolf Neugebauer, Phil Piwonka, and Diana-Maria Popa. Firecracker: Lightweight virtualization for serverless applications. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 419–434, Santa Clara, CA, February 2020. USENIX Association. URL: https://www.usenix.org/conference/nsdi20/presentation/agache.

[11] Amazon Web Services. Amazon elastic container service developer guide. Accessed: 2025-04-26. URL: https://docs.aws.amazon.com/AmazonECS/latest/developerguide/Welcome.html.

[12] Adam Belay, Andrea Bittau, Ali Mashtizadeh, David Terei, David Mazières, and Christos Kozyrakis. Dune: Safe user-level access to privileged CPU features. In *10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*, pages 335–348, Hollywood, CA, October 2012. USENIX Association. URL: https://www.usenix.org/conference/osdi12/technical-sessions/presentation/belay.

[13] Shuang Chen, Christina Delimitrou, and José F. Martínez. Parties: Qos-aware resource partitioning for multiple interactive services. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '19, page 107–120, New York, NY, USA, 2019. Association for Computing Machinery. doi:10.1145/3297858.3304005.

[14] Yue Cheng, Ali Anwar, and Xuejing Duan. Analyzing alibaba's co-located datacenter workloads. In *2018 IEEE International Conference on Big Data (Big Data)*, pages 292–297. IEEE, 2018.

[15] Christina Delimitrou and Christos Kozyrakis. Paragon: Qos-aware scheduling for heterogeneous datacenters. *SIGPLAN Not.*, 48(4):77–88, March 2013. doi:10.1145/2499368.2451125.

[16] Christina Delimitrou and Christos Kozyrakis. Quasar: resource-efficient and qos-aware cluster management. *SIGARCH Comput. Archit. News*, 42(1):127–144, February 2014. doi:10.1145/2654822.2541941.

[17] Christina Delimitrou and Christos Kozyrakis. Bolt: I know what you did last summer... in the cloud. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 599–613, 2017. doi:10.1145/3037697.3037743.

[18] Joshua Fried, Gohar Irfan Chaudhry, Enrique Saurez, Esha Choukse, Inigo Goiri, Sameh Elnikety, Rodrigo Fonseca, and Adam Belay. Making kernel bypass practical for the cloud with junction. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*, pages 55–73, Santa Clara, CA, April 2024. USENIX Association. URL: https://www.usenix.org/conference/nsdi24/presentation/fried.

[19] Joshua Fried, Zhenyuan Ruan, Amy Ousterhout, and Adam Belay. Caladan: Mitigating interference at microsecond timescales. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 281–297. USENIX Association, November 2020. URL: https://www.usenix.org/conference/osdi20/presentation/fried.

[20] Alexander Fuerst, Stanko Novaković, Íñigo Goiri, Gohar Irfan Chaudhry, Prateek Sharma, Kapil Arya, Kevin Broas, Eugene Bak, Mehmet Iyigun, and Ricardo Bianchini. Memory-harvesting VMs in cloud platforms. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2022.

DISTRIBUTION STATEMENT A. Approved for public release: Distribution unlimited. Case Number AFRL-2026-0584. Dated 09 Feb 2026

[21] Yu Gan, Yanqi Zhang, Dailun Cheng, Ankitha Shetty, Priyal Rathi, Nayan Katarki, Ariana Bruno, Justin Hu, Brian Ritchken, Brendon Jackson, Kelvin Hu, Meghna Pancholi, Yuan He, Brett Clancy, Chris Colen, Fukang Wen, Catherine Leung, Siyuan Wang, Leon Zaruvinsky, Mateo Espinosa, Rick Lin, Zhongling Liu, Jake Padilla, and Christina Delimitrou. An open-source benchmark suite for microservices and their hardware-software implications for cloud & edge systems. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '19, page 3–18, New York, NY, USA, 2019. Association for Computing Machinery. doi:10.1145/3297858.3304013.

[22] Google Cloud Platform. Online boutique (microservices demo), 2020. Accessed: 2025-04-26. URL: https://github.com/GoogleCloudPlatform/microservices-demo.

[23] Carlos Henríquez, Jarol Derley Ramón Valencia, and Germán Sánchez Torres. Architectural evolution at netflix: A case study on microservices and the transformation from monolithic to scalable systems. *Prospectiva*, 23(1), 2025.

[24] Daniel Hodges. Scheduling at scale: eBPF schedulers with Sched_ext. Dublin, October 2024. USENIX Association.

[25] Hang Huang, Jiangshan Lai, Jia Rao, Hui Lu, Wenlong Hou, Hang Su, Quan Xu, Jiang Zhong, Jiahao Zeng, Xu Wang, Zhengyu He, Weidong Han, Jiang Liu, Tao Ma, and Song Wu. Pvm: Efficient shadow paging for deploying secure containers in cloud-native environment. In *Proceedings of the 29th Symposium on Operating Systems Principles*, SOSP '23, page 515–530, New York, NY, USA, 2023. Association for Computing Machinery. doi:10.1145/3600006.3613158.

[26] Jack Tigar Humphries, Neel Natu, Ashwin Chaugule, Ofir Weisse, Barret Rhoden, Josh Don, Luigi Rizzo, Oleg Rombakh, Paul Turner, and Christos Kozyrakis. ghost: Fast & flexible user-space delegation of linux scheduling. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, SOSP '21, page 588–604, New York, NY, USA, 2021. Association for Computing Machinery. doi:10.1145/3477132.3483542.

[27] Darby Huye, Yuri Shkuro, and Raja R. Sambasivan. Lifting the veil on Meta's microservice architecture: Analyses of topology and request workflows. In *2023 USENIX Annual Technical Conference (USENIX ATC 23)*, pages 419–432, Boston, MA, July 2023. USENIX Association. URL: https://www.usenix.org/conference/atc23/presentation/huye.

[28] Darby Huye, Yuri Shkuro, and Raja R. Sambasivan. Lifting the veil on Meta's microservice architecture: Analyses of topology and request workflows. In *2023 USENIX Annual Technical Conference (USENIX ATC 23)*, pages 419–432, Boston, MA, July 2023. USENIX Association. URL: https://www.usenix.org/conference/atc23/presentation/huye.

[29] Zhipeng Jia and Emmett Witchel. Nightcore: efficient and scalable serverless computing for latency-sensitive, interactive microservices. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '21, page 152–166, New York, NY, USA, 2021. Association for Computing Machinery. doi:10.1145/3445814.3446701.

[30] Kostis Kaffes, Timothy Chong, Jack Tigar Humphries, Adam Belay, David Mazières, and Christos Kozyrakis. Shinjuku: preemptive scheduling for µsecond-scale tail latency. In *Proceedings of the 16th USENIX Conference on Networked Systems Design and Implementation*, NSDI'19, page 345–359, USA, 2019. USENIX Association.

[31] Harshad Kasture, Davide B. Bartolini, Nathan Beckmann, and Daniel Sanchez. Rubik: Fast analytical power management for latency-critical systems. In *Proceedings of the 48th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 598–610. IEEE Computer Society, 2015. doi:10.1145/2830772.2830797.

[32] Harshad Kasture and Daniel Sanchez. Ubik: Efficient cache sharing with strict qos for latency-critical workloads. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 729–742. ACM, 2014. doi:10.1145/2541940.2541943.

[33] Zijun Li, Jiagan Cheng, Quan Chen, Eryu Guan, Zizheng Bian, Yi Tao, Bin Zha, Qiang Wang, Weidong Han, and Minyi Guo. RunD: A lightweight secure container runtime for high-density deployment and high-concurrency startup in serverless computing. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, pages 53–68, Carlsbad, CA, July 2022. USENIX Association. URL: https://www.usenix.org/conference/atc22/presentation/li-zijun-rund.

[34] Lei Liu, Xinglei Dou, and Yuetao Chen. Intelligent resource scheduling for co-located latency-critical

DISTRIBUTION STATEMENT A. Approved for public release: Distribution unlimited. Case Number AFRL-2026-0584. Dated 09 Feb 2026

services: A Multi-Model collaborative learning approach. In *21st USENIX Conference on File and Storage Technologies (FAST 23)*, pages 153–166, Santa Clara, CA, February 2023. USENIX Association. URL: https://www.usenix.org/conference/fast23/presentation/liu.

[35] David Lo, Liqun Cheng, Rama Govindaraju, Parthasarathy Ranganathan, and Christos Kozyrakis. Heracles: Improving resource efficiency at scale. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture (ISCA)*, pages 450–462. IEEE Computer Society, 2015. doi:10.1145/2749469.2749475.

[36] Jean-Pierre Lozi, Baptiste Lepers, Justin Funston, Fabien Gaud, Vivien Quéma, and Alexandra Fedorova. The linux scheduler: a decade of wasted cores. In *Proceedings of the Eleventh European Conference on Computer Systems*, EuroSys '16, New York, NY, USA, 2016. Association for Computing Machinery. doi:10.1145/2901318.2901326.

[37] Jean-Pierre Lozi, Baptiste Lepers, Justin Funston, Fabien Gaud, Vivien Quéma, and Alexandra Fedorova. The linux scheduler: a decade of wasted cores. In *Proceedings of the Eleventh European Conference on Computer Systems*, EuroSys '16, New York, NY, USA, 2016. Association for Computing Machinery. doi:10.1145/2901318.2901326.

[38] Shutian Luo, Huanle Xu, Chengzhi Lu, Kejiang Ye, Guoyao Xu, Liping Zhang, Yu Ding, Jian He, and Chengzhong Xu. Characterizing microservice dependency and performance: Alibaba trace analysis. In *Proceedings of the ACM Symposium on Cloud Computing*, SoCC '21, page 412–426, New York, NY, USA, 2021. Association for Computing Machinery. doi:10.1145/3472883.3487003.

[39] Jonathan Mace, Peter Bodik, Rodrigo Fonseca, and Madanlal Musuvathi. Retro: Targeted resource management in multi-tenant distributed systems. In *Proceedings of the 12th USENIX Conference on Networked Systems Design and Implementation*, NSDI'15, page 589–603, USA, 2015. USENIX Association.

[40] Filipe Manco, Costin Lupu, Florian Schmidt, Jose Mendes, Simon Kuenzer, Sumit Sati, Kenichi Yasukata, Costin Raiciu, and Felipe Huici. My vm is lighter (and safer) than your container. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, page 218–233, New York, NY, USA, 2017. Association for Computing Machinery. doi:10.1145/3132747.3132763.

[41] Ingo Molnar. Modular Scheduler Core and Completely Fair Scheduler (CFS). https://www.kernel.org/doc/html/latest/scheduler/sched-design-CFS.html, 2007. Accessed: 2025-04-02.

[42] Amy Ousterhout, Joshua Fried, Jonathan Behrens, Adam Belay, and Hari Balakrishnan. Shenango: Achieving high CPU efficiency for latency-sensitive datacenter workloads. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 361–378, Boston, MA, February 2019. USENIX Association. URL: https://www.usenix.org/conference/nsdi19/presentation/ousterhout.

[43] George Prekas, Marios Kogias, and Edouard Bugnion. Zygos: Achieving low tail latency for microsecond-scale networked tasks. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, page 325–341, New York, NY, USA, 2017. Association for Computing Machinery. doi:10.1145/3132747.3132780.

[44] Henry Qin, Qian Li, Jacqueline Speiser, Peter Kraft, and John Ousterhout. Arachne: core-aware thread management. In *Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation*, OSDI'18, page 145–160, USA, 2018. USENIX Association.

[45] Gourav Rattihalli, Madhusudhan Govindaraju, Hui Lu, and Devesh Tiwari. Exploring potential for non-disruptive vertical auto scaling and resource estimation in kubernetes. In *2019 IEEE 12th International Conference on Cloud Computing (CLOUD)*, pages 33–40, 2019. doi:10.1109/CLOUD.2019.00018.

[46] Craig W. Reynolds. Flocks, herds and schools: A distributed behavioral model. In *Proceedings of the 14th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '87, page 25–34, New York, NY, USA, 1987. Association for Computing Machinery. doi:10.1145/37401.37406.

[47] Vasily A. Sartakov, Lluís Vilanova, David Eyers, Takahiro Shinagawa, and Peter Pietzuch. CAP-VMs: Capability-Based isolation and sharing in the cloud. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 597–612, Carlsbad, CA, July 2022. USENIX Association. URL: https://www.usenix.org/conference/osdi22/presentation/sartakov.

[48] Mohammad Shahrad, Rodrigo Fonseca, Inigo Goiri, Gohar Chaudhry, Paul Batum, Jason Cooke, Eduardo Laureano, Colby Tresness, Mark Russinovich, and Ricardo

DISTRIBUTION STATEMENT A. Approved for public release: Distribution unlimited. Case Number AFRL-2026-0584. Dated 09 Feb 2026

Bianchini. Serverless in the wild: Characterizing and optimizing the serverless workload at a large cloud provider. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pages 205–218. USENIX Association, July 2020. URL: https://www.usenix.org/conference/atc20/presentation/shahrad.

[49] Akshitha Sriraman and Thomas F. Wenisch. μTune: Auto-Tuned threading for OLDI microservices. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 177–194, Carlsbad, CA, October 2018. USENIX Association. URL: https://www.usenix.org/conference/osdi18/presentation/sriraman.

[50] Kun Suo, Yong Zhao, Wei Chen, and Jia Rao. An analysis and empirical study of container networks. In *IEEE INFOCOM 2018 - IEEE Conference on Computer Communications*, pages 189–197, 2018. doi:10.1109/INFOCOM.2018.8485865.

[51] Paul Turner, Ben Segall Rao, and Nikhil Levin. Per-entity load tracking, Jan 2013. Accessed: 2025-12-03. URL: https://lwn.net/Articles/531853/.

[52] Abhishek Verma, Luis Pedrosa, Madhukar Korupolu, David Oppenheimer, Eric Tune, and John Wilkes. Large-scale cluster management at google with borg. In *Proceedings of the Tenth European Conference on Computer Systems*, EuroSys '15, New York, NY, USA, 2015. Association for Computing Machinery. doi:10.1145/2741948.2741964.

[53] Bo Wang, Zhiguang Chen, and Nong Xiao. A survey of system scheduling for hpc and big data. In *Proceedings of the 2020 4th International Conference on High Performance Compilation, Computing and Communications*, HP3C 2020, page 178–183, New York, NY, USA, 2020. Association for Computing Machinery. doi:10.1145/3407947.3407977.

[54] Cong Xu, Karthick Rajamani, Alexandre Ferreira, Wesley Felter, Juan Rubio, and Yang Li. dcat: Dynamic cache management for efficient, performance-sensitive infrastructure-as-a-service. In *Proceedings of the Thirteenth EuroSys Conference*, pages 1–13, 2018.

[55] Xiao Zhang, Eric Tune, Robert Hagmann, Rohit Jnagal, Vrigo Gokhale, and John Wilkes. Cpi2: Cpu performance isolation for shared compute clusters. In *Proceedings of the 8th ACM European Conference on Computer Systems*, EuroSys '13, page 379–391, New York, NY, USA, 2013. Association for Computing Machinery. doi:10.1145/2465351.2465388.

DISTRIBUTION STATEMENT A. Approved for public release: Distribution unlimited. Case Number AFRL-2026-0584. Dated 09 Feb 2026

Efficient LLM Serving on Commodity GPU Clusters with Data-Reduced Cross-Instance Orchestration

Jiangsu Du* Hongbin Zhang* Taosheng Wei Zhenyi Zheng
Jiazhi Jiang Kaiyi Wu Zhiguang Chen Yutong Lu†

School of Computer Science and Engineering, Sun Yat-Sen University

Abstract

Existing LLM serving strategies can be categorized by whether prefill and decode phases are disaggregated: non-disaggregated (NoDG) or fully disaggregated (FuDG). However, they neither fit commodity GPU clusters, which remain widely deployed as mainstream AI infrastructure. NoDG suffers from severe prefill–decode interference, while FuDG depends heavily on high-performance interconnects that such clusters lack.

We present EcoServe, an LLM serving system tailored to commodity GPU clusters. It enables a data-reduced collaboration among inference instances to mitigate prefill-decode interference, termed the partially disaggregated (PaDG) strategy. Particularly, within a single instance, PaDG disaggregates the prefill and decode phases along the time dimension to mitigate interference and enhance throughput. Next, it coordinates multiple instances and cyclically activates them to ensure the continuous availability of prefill processing, thereby rescuing latency. Thus, EcoServe's basic serving unit is the macro instance, within which multiple instances collaborate. It further incorporates an adaptive scheduling algorithm to route requests in a macro instance and a mitosis scaling approach for fine-grained capacity adjustments in online scenario.

On a 32-GPU NVIDIA L20 cluster over Ethernet, EcoServe improves goodput by 1.96×, 1.99×, 2.51×, and 2.40× when serving 30B- and 70B-scale LLMs, compared to four representative NoDG and FuDG systems, vLLM, Sarathi, DistServe, and MoonCake. EcoServe remains competitive even on an NVIDIA H100 cluster with NVLink and InfiniBand. Our code is released at https://github.com/MLSysU/EcoServe.

1 Introduction

Large language models [10, 18, 46] (LLMs) have been widely adopted in various tasks. Handling these massive LLM inference workloads requires cluster-level deployment. Recent years have witnessed the emergence of increasingly advanced GPU architectures, from H100 to GB200, offering higher compute density, larger memory bandwidth, and significantly improved interconnect capabilities. Although such devices are widely promoted and actively used in LLM research, their high cost and the slow refresh cycle of production environments imply that, in practice, commodity GPU clusters without high-performance interconnects still account for a significant share [15, 39, 47]. This observation aligns with popular LLM serving works such as Sarathi [7], DistServe [55], and FlexGen [42], all of which evaluate on GPU nodes with weak or commodity interconnects. Moreover, our study is conducted on an NVIDIA L20 cluster that provides LLM serving for a major super app of a top-tier Internet company. Consequently, efficiently serving LLMs on commodity GPU clusters remains essential.

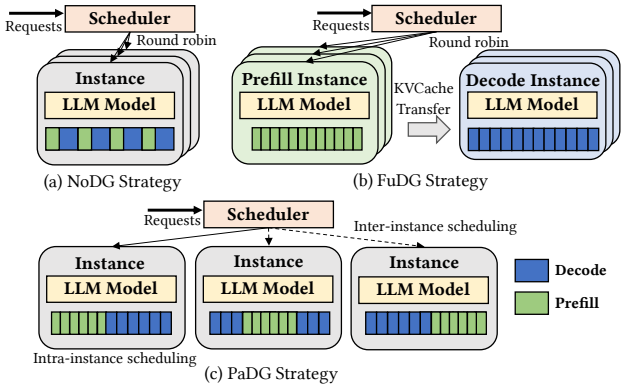


Figure 1: The NoDG, FuDG, and PaDG strategies.

LLM inference consists of two distinct phases, the prefill phase and the decode phase, each associated with a different latency SLO, time to first token (TTFT) and time per output token (TPOT). These two latency goals, together with throughput, form an inherent performance trade-off triangle. Existing cluster-level LLM inference solutions [2, 4, 7, 35, 37, 52, 55] can be categorized based on whether prefill and decode phases

*Jiangsu Du and Hongbin Zhang contributed equally.
†Corresponding author

are disaggregated: the non-disaggregated (NoDG) strategy and the fully disaggregated (FuDG) strategy. They both have severe limitations for commodity GPU clusters, either incurring severe interference between prefill and decode phases, or heavily depends on high-performance interconnects.

As shown in Figure 1(a), given that the prefill and decode phases share model weight and KV cache, the NoDG strategy [2,4,7,52], which colocates the two phases within a single instance and replicates multiple instances, appears to be a natural choice. Such colocation inevitably leads to significant inter-phase interference [35, 53, 55]. As two phases execute alternatively, prioritizing the scheduling of one phase risks violating the latency requirements of the other. This interference also harms throughput, as the decode phase cannot accumulate a sufficiently large batch size to saturate GPU resources.

As illustrated in Figure 1(b), FuDG [35, 37, 55] proposes to fully eliminate the prefill-decode interference by assigning the two phases to separate instances. However, since massive amounts of KV cache must be transferred between prefill and decode instances, it relies on hyper-clusters with powerful interconnects as the default hardware infrastructure, as exemplified by the FuDG approach in DistServe [55]. Unfortunately, high-performance interconnects like NVLINK and InfiniBand are not only costly and power-hungry, but also demand substantial technical expertise for construction and maintenance. Moreover, scaling FuDG involves adjusting both prefill and decode instances, which complicates load balancing [28].

EcoServe is designed to deliver cost-effective LLM serving on commodity clusters, without relying on high-performance interconnects. As shown in Figure 1(c), our key observation is that intra-instance scheduling, which determines when to execute prefills and decodes, should be coordinated with inter-instance scheduling, which decides when and where requests should be routed, to raise the upper bound of the trade-off triangle and fully utilize available resources. Thus, EcoServe incorporates temporal disaggregation and rolling activation to proactively schedule at intra- and inter-instance levels, so called the partially disaggregated (PaDG) strategy.

Within a single instance, the prefill and decode phases are disaggregated along the time dimension, with each phase lasting longer. By reducing prefill-decode switching frequency, it mitigates interference and achieves higher throughput. Moreover, since both phases are still in a single instance, PaDG avoids massive KV cache transmission. Next, to meet latency SLOs, it proactively coordinates multiple instances in a cyclic pattern. At any given time, at least one instance is activated to process prefills, thereby ensuring acceptable TTFT latency. A group of such cooperating instances is referred to as a macro instance.

EcoServe further includes an adaptive scheduling algorithm and the mitosis scaling approach. The adaptive scheduling algorithm guides request scheduling within the macro instance.

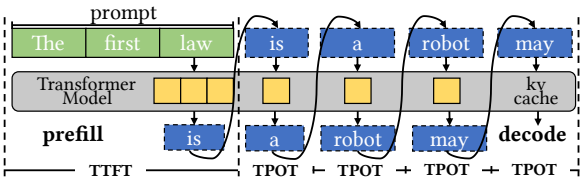


Figure 2: LLM autoregressive decoding process.

While prioritizing the maintenance of satisfactory TPOT, it identifies the most suitable instance for admitting new requests and determines the optimal number of prefill tokens that can be inserted into that instance. The mitosis scaling approach enables fine-grained control over system capacity by continuously adjusting the number of instances within multiple macro instances. A serializable proxy object is introduced to support logical migration between macro instances without interrupting execution. Our contributions are summarized:

- We present EcoServe for cost-effective LLM serving on clusters with commodity interconnects. It is based on the PaDG strategy, along with the adaptive scheduling algorithm and the mitosis scaling approach.
- We implement EcoServe in a hierarchical architecture.
- We evaluate EcoServe and compare it against four representative LLM inference systems, i.e. vLLM [4], Sarathi [7], DistServe [55], and MoonCake [37] on clusters equipped with either commodity or high-performance networks, demonstrating its superior efficiency.

2 Preliminary and Motivation

2.1 LLM Inference Characteristics

As illustrated in Figure 2, the LLM predicts the next token with the accumulated context iteratively until it encounters the end-of-sequence (EoS). By saving the key and value embedding in the memory (i.e. KV cache), redundant computations are avoided in subsequent steps, thus the inference process is divided into prefill and decode phases. A request consists of a single prefill followed by tens or even hundreds of decode steps. Modern LLMs primarily adopt the Transformer architecture [48], which leverages the self-attention mechanism to model complex dependencies in sequences. Here we conclude its computation and memory characteristics.

Distinct arithmetic intensity. Transformer's computation involves four main parts: QKV projection, QKV attention, output projection, and dim expansion and reduction. In these parts, matrix multiplications dominate computation time, while softmax and layer normalization contribute only marginally. Table 2 lists six major matrix multiplications,

Table 1: Notations.

Variable	Description	Notation
prompt_len	The length of prompt	S
generation_len	The length of generated tokens	G
batch_size	The number of batched requests	B
layer_num	The number of model layers	L
hidden_size	Input dimension of the hidden layer	H
heads	The number of attention heads	M
size_per_head	The hidden state per head	D

Table 2: Approximate arithmetic intensity (AI) of primary operations in LLMs. Negligible factors are omitted.

Operation	P/D	FLOPS	Memory Access	Approximate AI
QKV Projection	Prefill	$6BSH^2$	$6BSH+3H^2$	BS
	Decode	$6BH^2$	$6BH+3H^2$	B
Attention QK^T	Prefill	$2BS^2H$	$2BSH+BS^2M$	S
	Decode	$2BSH$	$2BSM+BH(S+1)$	1
Attention $(QK^T)V$	Prefill	$2BS^2H$	$2BSH+BS^2M$	S
	Decode	$2BSH$	$2BSM+BH(S+1)$	1
Output Projection	Prefill	$2BSH^2$	$2BSH+H^2$	BS
	Decode	$2BH^2$	$2BH+H^2$	B
Dim Expansion	Prefill	$8BSH^2$	$2BSH+4H^2$	BS
	Decode	$8BH^2$	$2BH+4H^2$	B
Dim Reduction	Prefill	$8BSH^2$	$2BSH+4H^2$	BS
	Decode	$8BH^2$	$2BH+4H^2$	B

whose arithmetic intensities are computed separately for prefill and decode using the hyperparameters in Table 1. Negligible terms (e.g., $1/H$) are omitted for approximation. Prefill intensity depends on both sequence length S and batch size B, while decode depends merely on B and requires additional KV cache loading. Thus, prefill is compute-bound, whereas decode is memory-bound.

Memory-compute trade-off. During LLM inference, limited memory capacity constrains computational parallelism, making parallel inference a potential avenue of superlinear speedup. Beyond model weights, the KV cache is the primary contributor to memory usage. In the decode phase, processing hundreds of requests simultaneously leads to substantial memory consumption. For example, in Llama-30B, a single token requires 1.52 MB of KV cache, so 128 requests with an average output of 300 tokens consume about 58.4 GB, comparable to the model weights. Moreover, since input and output lengths are stochastic and outputs remain unknown until completion, memory must be over-provisioned to avoid OOM, further complicating resource allocation.

2.2 LLM Batching Techniques

To fully utilize modern GPUs, batching is commonly adopted in deep learning workloads, where multiple samples are processed simultaneously to expose high parallelism. Regarding whether to pack prefill and decode together, modern LLM serving systems adopt either separate batching [4] or hybrid batching [7]. Separate batching handles prefills and decodes independently. Since prefills are compute-bound, even a batch size of one can saturate the GPU, whereas decodes

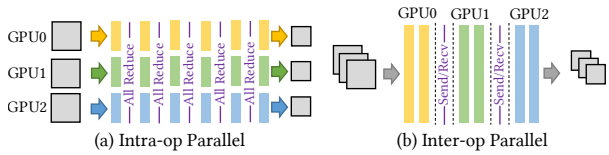


Figure 3: Intra-op and inter-op parallelism.

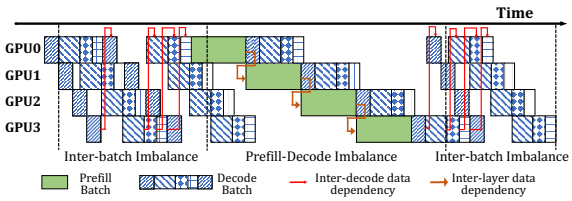


Figure 4: The inter-op parallelism with separate batching suffers from severe pipeline bubbles.

are memory-bound and require hundreds of requests to do so. By contrast, hybrid batching organizes requests of both prefill and decode phases in a hybrid batch.

2.3 Scale-up LLM Inference

To scale up both computational and memory capacity within a single inference instance, LLMs adopt several forms of parallelism, can be categorized to intra-op parallelism [12, 22, 30, 41, 43, 44] and inter-op parallelism [20, 27, 34, 53]. As shown in Figure 3, intra-op parallelism splits each layer across multiple GPUs, with model weights and KV cache evenly distributed among workers to accelerate single inference runs. It introduces frequent collective communication [13, 14].

Inter-op parallelism partitions a model layer-wise, with each device handling tens of consecutive layers. As a result, communication is limited to lightweight point-to-point transfers that occur only every few layers. The largely-reduced communication makes it a promising approach to support LLM serving, especially on commodity hardware that high-performance intra-node interconnects like NVLINK are unavailable. However, as illustrated in Figure 4, the varied input lengths and the inter-iteration dependencies in the decode lead to imbalanced and interdependent micro-batches [7, 53], causing severe pipeline bubbles.

2.4 Scale-out LLM Serving

The adoption of cluster-level infrastructure have become essential to meet the increasing demands of LLM workloads. There are NoDG and FuDG strategies.

Table 3: KV cache generation speed in the prefill instance. Here each node includes 8 GPUs and intra-op parallelism is used when a single GPU's memory capacity is insufficient.

Model	Device	Tokens/s	Data volume
Llama-30B	L20	6584.6	9.796 GB/s
Llama-30B	A800	26189.2	38.96 GB/s
Llama-30B	H100	64220.0	95.54 GB/s

2.4.1 Non-Disaggregated Strategy

NoDG [3, 4, 7] colocates prefills and decodes in a single instance, which manages the entire life-cycle of a request. In Figure 1(a), when one instance cannot handle the incoming load, the system scales out by replicating additional independent instances. NoDG supports both separate batching and hybrid batching. However, since prefills typically take much longer than a decoding step, joint scheduling causes decodes to be delayed, significantly increasing TPOT; conversely, the inclusion of decodes leads to non-trivial increases in TTFT. Thus, NoDG often exhibits low throughput as the decode phase struggles to accumulate a sufficiently large batch size to saturate the GPU under SLO constraints. To mitigate this, chunked prefill [7] divides a prefill into smaller chunks and processes over multiple iterations, but it incurs repeated KV cache accesses and heavily depends on the input-to-output length ratio. Moreover, NoDG is poorly compatible with pipeline parallelism [7, 53] due to load imbalance and interdependence.

2.4.2 Fully-Disaggregated Strategy

In Figure 1(b), FuDG [35, 37, 55] disaggregates the prefill and decode phases into separate instances, with the KV cache transferred between them. Although it avoids prefill-decode interference, it incurs data transmission [55], load imbalance [28], and complex engineering.

For data transmission, Table 3 reports the KV cache generation rate on GPU nodes (all prefill instances), indicating the volume of data to be transferred. For LLaMA-30B on a node with 8 NVIDIA A800 GPUs, the required output bandwidth reaches 39 GB/s, necessitating at least a 400 Gbps network. These represent only theoretical lower bounds; in practice, the small granularity of KV cache blocks and the complexity of interconnect topology further increase bandwidth demand. More critically, many commodity clusters were never provisioned with high-performance inter-node communication networks, making FuDG less suitable in such environments.

FuDG can be further classified into intra-node FuDG (DistServe [55]) and inter-node FuDG (MoonCake [37]), depending on whether prefill and decode instances are colocated within the same node or distributed across nodes. DistServe targets scenarios with insufficient inter-node interconnects by deploying prefills and decodes within a node and transferring KV cache via intra-node links. MoonCake designs a centralized KV cache pool and relies on InfiniBand to connect prefill and decode instances across nodes.

Besides large data transfer, FuDG suffers from load imbalance, limited parallelism compatibility, and high engineering complexity. 1) Load imbalance occurs in both computation and memory. On the computation side, continuously varying input and output lengths make it difficult to determine an appropriate number of prefill and decode instances. On the memory side, decode instances must retain large amounts of KV cache, whereas prefill instances require far less. 2) Limited compatibility with intra-op parallelism on nodes without GPU-direct links, as collective communication and KV-cache migration contend heavily for bandwidth. 3) Coordinating resources, communication, and memory across instances significantly increase engineering complexity, hindering adoption by small teams.

3 EcoServe Design

Figure 5 demonstrates EcoServe's architecture. It follows the partially disaggregated (PaDG) strategy and proactively orchestrates both intra- and inter-instance execution. Specifically, the prefill and decode phases are disaggregated along the time dimension in each instance (temporal disaggregation) to mitigate interference, while coordination is conducted across different instances to ensure continuous availability (rolling activation).

The system adopts a hierarchical architecture with three levels of scheduling, i.e. the instance scheduler, the macro-instance scheduler, and the overall scheduler. The instance scheduler manages execution within a single instance, including prefill–decode coordination, device orchestration, and compliance with higher-level directives. The macro-instance scheduler coordinates multiple instances by aggregating their execution states and dispatching requests according to profiling results and SLOs. The macro instance is a unique abstraction in EcoServe and serves as the basic unit. Finally, the overall scheduler assigns requests to macro-instances based on their capabilities. In this work, we focus primarily on the internal architecture of a macro-instance.

Furthermore, EcoServe integrates an adaptive scheduling algorithm and a mitosis scaling approach to enable the practical adoption of PaDG. The adaptive scheduling algorithm coordinates instance- and macro-instance schedulers in a master–worker pattern. The mitosis scaling approach enables capacity adjustments at the instance level rather than the coarser macro-instance level, enabling fine-grained resource utilization. To support this, a serializable proxy object is designed to seamlessly add or remove instances.

3.1 Partially Disaggregated Strategy

PaDG enables a data-reduced collaboration across instances to alleviate prefill–decode interference. It achieves this

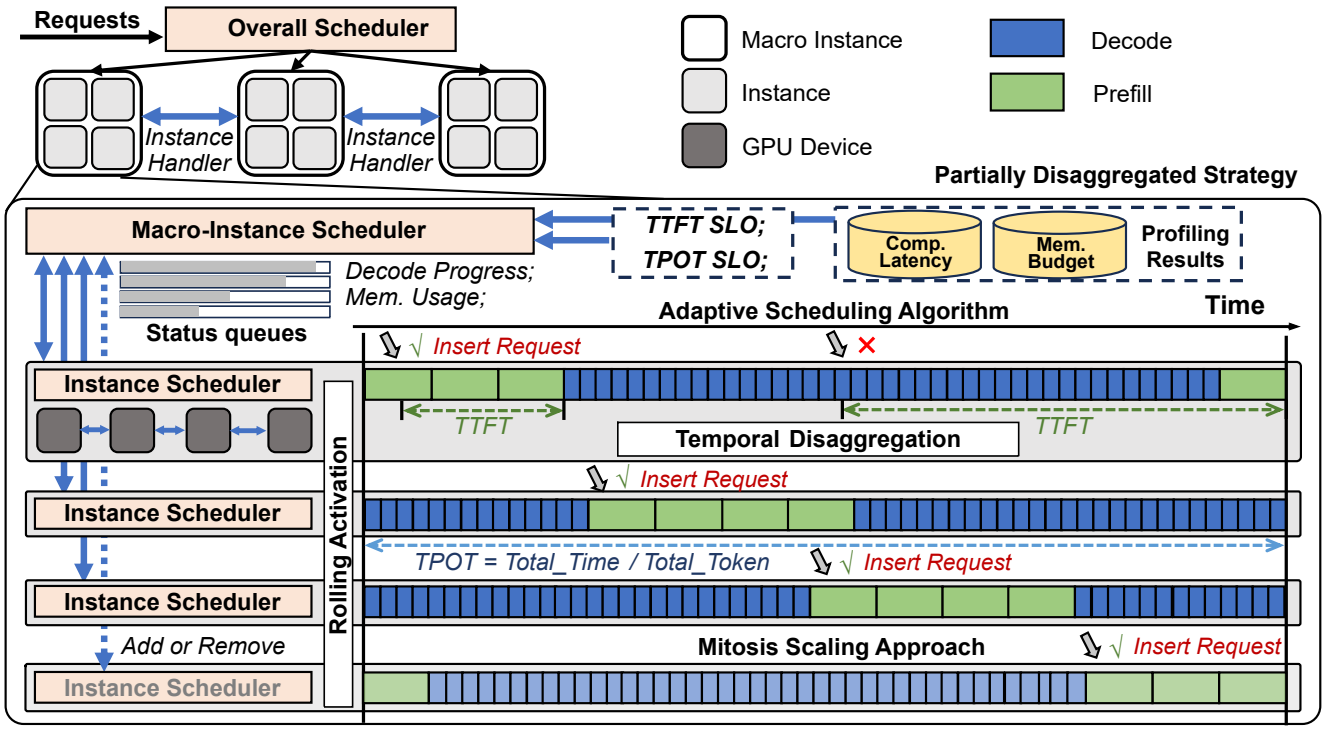


Figure 5: EcoServe adopts a hierarchical architecture in which the system consists of multiple macro-instances, and each macro-instance comprises multiple instances. The macro instance is the basic unit, within which multiple instances collaborate.

through employing temporal disaggregation and rolling activation to proactively schedule requests at both intra- and inter-instance levels.

3.1.1 Temporal Disaggregation

PaDG takes the proactive intra-instance scheduling to disaggregate the prefill and decode phases along the temporal dimension within each instance to mitigate prefill-decode interference. Unlike the FuDG strategy, which assigns dedicated prefill and decode roles to different instances, PaDG maps these roles to different time slots on the same instance. This design keeps the KV cache local and eliminates the need for explicit KV cache transmission, thereby improving on-device resource utilization and avoiding cross-instance data movement.

Since each phase occupies an instance for an extended period, intra-instance scheduling along risks degrading TTFT. As shown in Figure 5, when a new request arrives at an instance currently processing prefills, it can be immediately admitted, thereby meeting the TTFT SLO. However, if the instance is in the decode phase, the request must wait until the next prefill window, potentially resulting in unacceptable TTFT violations.

In contrast, modern LLM serving systems typically render outputs in a typewriter fashion, where the TPOT SLO can be satisfied once enough tokens are produced within a given time window. This allows decode execution, when faster than the TPOT constraint, to accumulate slack time (referred to as saved TPOT), which can then be leveraged to interrupt decodes for prefills without violating TPOT requirements.

3.1.2 Rolling Activation

Although a single instance can only remain in either the prefill or decode phase at any given time and therefore cannot immediately process newly arrived requests, rolling activation addresses this by proactively scheduling multiple instances and staggering their prefill phases over time, ensuring continuous prefill availability. As illustrated in Figure 5, instances enter the prefill phase in a cyclic pattern. From the perspective of individual requests, they are always routed to an instance currently processing prefills and can be admitted immediately. Thus, rolling activation reduces waiting time and preserves TTFT. Since output lengths are unknown, instances must continuously update their status, such as decode progress and memory usage, to the macro-instance scheduler for immediate coordination.

3.2 System Metrics for User Experience

Figure 6 illustrates the timing of both runtime and frontend. Once a token is generated, it is transmitted to the frontend, buffered, and then rendered. Classical metrics such as TTFT

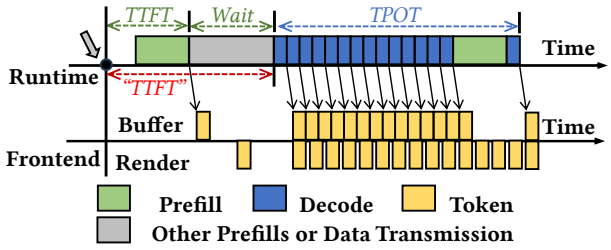


Figure 6: Runtime and frontend Timing.

and TPOT characterize latency from the runtime perspective, capturing the prefill and decode phases. However, these metrics are insufficient to capture service quality, as modern LLM serving systems handle massive concurrent requests and introduce a prefill–decode switching stage for each.

Before a request enters its decode phase, additional operations occur across all strategies, NoDG, PaDG, and FuDG. For the NoDG and PaDG strategies, prefills from other requests may occur before a given request can enter its decode phase. For the FuDG strategy, it incurs KV cache transmission overhead prior to the decode phase. Thus, the phase-switching waiting time should be recorded to provide a more comprehensive evaluation of LLM serving systems. This waiting overhead has implicitly appeared in previous work, and is frequently misrepresented.

To maintain consistency with prior work, we continue using TTFT to evaluate prefill latency. In our context, the reported TTFT consists of two components: the actual TTFT and the phase-switching waiting time. This definition represents **a stricter SLO**, as it incorporates additional overhead within the same latency bound. Accordingly, TPOT measurement begins only after the phase-switching.

3.3 Adaptive Scheduling Algorithm

The adaptive scheduling algorithm enables the macro-instance scheduler and the instance schedulers to operate in a master–worker manner, where the two levels of schedulers collaborate to coordinate scheduling decisions. From the instance scheduler's perspective, although the outcome involves disaggregating the prefill and decode phases along the temporal dimension, its scheduling policy prioritizes prefills. It continues processing active decodes, periodically updating its progress to the macro-instance scheduler, and switches to prefills upon receiving new requests from the upper scheduler.

From the macro-instance scheduler's perspective, it continuously collects status updates from all instances and dispatches requests accordingly. It traverses instances in a cyclic manner. For each incoming request, it first attempts to route it to the same instance that handled the previous request. If the instance satisfies the constraints verified by Algorithm 1, the request is forwarded. Otherwise, it checks the next available instance. To support this process, multiple public message queues are constructed on the same node as the macro-instance scheduler. Each instance periodically uploads its status to the corresponding queue, enabling the macro-instance scheduler to aggregate this information and make global scheduling decisions.

Algorithm 1: Constraint Checking Algorithm

Data: Latency constraints: S_{TTFT}, S_{TPOT}
Input: I: Statuses of all instances in a macro instance;
r: A new request

Function `CheckConstraints`(I, r)**:**
  `// 1. Check TTFT constraint`
  $t_{\text{switch}} \leftarrow$ last phase switching timestamp;
  $pending_prefills \leftarrow \{ r \in instance.reqs \mid r.arrival_time \geq t_{\text{switch}} \} \cup \{ req \}$;
  $t_{\text{total}} \leftarrow \sum(pending_prefills$ durations$)$;
  if $t_{\text{total}} > S_{TTFT}$ **then**
    return NotSatisfied;
  `// 2. Check TPOT constraint`
  $existed_decodes \leftarrow \{ r \in instance.reqs \mid r.arrival_time < t_{\text{switch}} \}$;
  $saved_tpots \leftarrow [\,]$;
  $current_time \leftarrow$ current timestamp;
  foreach $r \in existed_decodes$ **do**
    $L \leftarrow r.output_length$;
    $saved_tpot \leftarrow L \times S_{TPOT} - (current_time - r.first_token_time)$;
    append $saved_tpot$ to $saved_tpots$;
  $mean_saved_tpot \leftarrow mean(saved_tpots)$;
  if $mean_saved_tpot < t_{\text{total}}$ **then**
    return NotSatisfied;
  `// 3. Check memory capacity`
  if $req_kvcache_size > remain_memsize$ **then**
    return NotSatisfied;
  return Satisfied;

As displayed in Algorithm 1, the constraint checking algorithm is responsible for verifying that assigning an incoming request to an instance will not violate the TTFT/TPOT SLOs or exceed the instance's available memory capacity. First, the algorithm ensures that the total duration (denoted as t_{total}) will not exceed the TTFT constraint (S_{TTFT}) after adding a new request to the pending prefills during the prefill phase. The prefill duration of a single request can be predicted in advance by profiling sequences of various lengths. Next, by utilizing updated information from the instance, the algorithm calculates the saved TPOT by subtracting the required time from the achieved time. PaDG does not assume uniformly borrowing slack over the entire decoding process. Instead, slack is estimated and utilized over short decoding intervals. During each prefill–decode switch, the saved TPOT of each

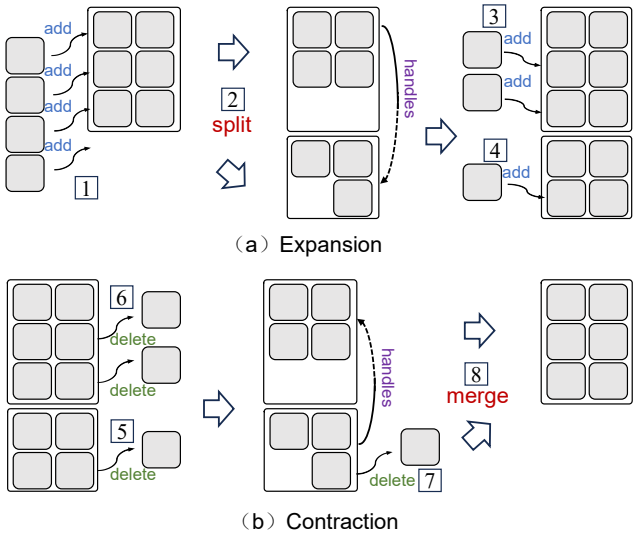


Figure 7: The illustration of the expansion and contraction processes. Here $N_l = 3$ and $N_u = 6$.

instance is locally accumulated and then consumed by incoming prefills. This localized and short-term design removes reliance on long-term prediction and improves robustness. As discussed in 3.1.1, provided that t_{total} does not exceed the saved TPOT, the TPOT constraint will be satisfied. Finally, the algorithm ensures that the total KV cache size does not exceed the remaining GPU memory capacity, preventing memory overflow.

3.4 Mitosis Scaling Approach

With the introduction of macro-instance abstraction, scaling at this coarse granularity, or relying on a single macro instance to manage all instances, would be both inefficient and inflexible. For example, as the request rate continuously increases, the macro instance scheduler must manage an increasing number of instances to meet SLOs, eventually becoming a bottleneck. Inspired by biological cell mitosis, the mitosis scaling approach adjusts capacity at the instance level, allowing EcoServe to adapt more precisely to workload fluctuations.

3.4.1 Expansion and Contraction

The mitosis scaling approach expands or contracts capacity in two steps, first by adding or removing instances within a macro-instance, and then by splitting or merging entire macro-instances. We define two hyperparameters, N_l and N_u, as the lower and upper bounds on the number of instances in a macro-instance. A small N_l causes frequent phase switching, whereas a large N_u incurs management overhead and may create scheduling bottlenecks.

Figure 7 illustrates an example of how the expansion and contraction processes are performed. The scaling is triggered either when the system fails to meet the defined SLOs or when there is sustained resource underutilization. New instances are incrementally added (step 1) until the number of instances exceeds the upper limit N_u, at which point a new macro instance containing N_l instances is split off from the original macro instance (step 2). If more instances are required, they are first added to the original macro instance until it again reaches N_u (step 3), and subsequent instances are then added to the new macro instance (step 4).

On the contrary, when the capacity becomes excessive and the contraction process is triggered, instances are firstly removed from the smallest macro instance until the number of instances in the macro instance reaches N_l (step 5). Next, instances are removed from a full macro instance (step 6). When the total number of instances across these two macro instances reaches N_u (step 7), they will be merged into a single macro instance (step 8). These removed instances will complete allocated requests before being freed. Generally, an inference system maintains multiple full macro instances, along with one or two partially filled macro instances.

3.4.2 Scaling Threshold

EcoServe decides when to expand or contract capacity by monitoring runtime signals. For expansion, the TTFT SLO attainment is adopted. When the system becomes overloaded, the TTFT SLO attainment is the first to decline and is more sensitive than TPOT SLO attainment. This is because the macro-instance scheduler routes requests only if an instance satisfies all constraints; when the system is overloaded and no instance meets these constraints, the requests become pending. These pending requests will immediately cause the TTFT SLO attainment to drop sharply, preceding degradation in TPOT SLO attainment. As demonstrated in Equation 1, EcoServe continuously tracks the average TTFTs and starts to add a new instance once it exceeds that the TTFT SLO, thereby preserving both TTFT and TPOT guarantees.

$$\frac{1}{n}\sum_{i=1}^{n} T_{\text{TTFT}}^{(i)} > T_{\text{TTFT}}^{\text{SLO}} \tag{1}$$

In contrast, saved TPOT is adopted as the signal for contraction. When the request arrival rate is low and system pressure decreases, the saved TPOT increases. For a macro-instance with n instances, we consider the reasonable range of saved TPOT to be $[T_{\text{TTFT}}, T_{\text{TTFT}} \cdot \frac{n+1}{n}]$, where the factor $(n+1)/n$ reflects the reduction in per-instance decode load if the macro-instance were hypothetically expanded from n to $n+1$ instances. Once the saved TPOT exceeds the upper bound of this interval, as shown in Equation 2, the system is over-provisioned, and EcoServe begins contracting instances.

$$T_{\text{savedTPOT}} > T_{\text{TTFT}} \cdot \frac{n+1}{n} \tag{2}$$

Table 4: Dataset Features and Corresponding SLOs.

DataSet	In_{Avg}	In_{Med}	Out_{Avg}	Out_{Med}	SLO_{TTFT}	SLO_{TPOT}
Alpaca-gpt4 [1]	20.63	17.00	163.80	119.00	1s	100ms
ShareGPT [5]	343.76	148.00	237.20	152	5s	100ms
LongBench [9]	2686.89	2736.50	101.78	19	15s	100ms

3.4.3 Flexible Instance Migration

To dynamically split or merge macro instances without reinitializing or interrupting individual instances, we design a serializable proxy object and transfer instance handles between different macro-instance schedulers. At the core of this design is the InstanceHandler metadata, which encapsulates essential information such as the instance entrypoint and its associated public message queues. When a handle is transferred between macro-instance schedulers (i.e., across processes), it is first serialized using the pickle library and then sent to the target macro-instance scheduler. The transmission process is coordinated by the overall scheduler. Upon deserialization, the target macro-instance connects with the entrypoint to obtain control of the instance, after which the instance is orchestrated into its rolling activation process. This design enables logical migration of instances across macro instances without interrupting their execution, thereby supporting more flexible and low-overhead scaling.

4 Evaluation

EcoServe adopts vLLM [4] as the single-device runtime, employs Ray to control multiple devices within each instance, and leverages ZeroMQ to synchronize instance status with the macro-instance scheduler. We evaluate EcoServe on LLMs of different scales across diverse datasets and clusters, showing its performance, scalability, and parallelism compatibility.

4.1 Experimental Setup

Cluster testbed. We conduct experiments on three clusters featuring commodity or high-performance interconnects. The primary cluster consists of 8 nodes with 64 NVIDIA L20-48GB GPUs, where GPUs are PCIe-connected and nodes communicate via 10 Gbps Ethernet. The second cluster has 2 nodes with 16 NVIDIA A800-80GB GPUs, also PCIe-connected but using a higher-bandwidth 25 Gbps RoCE interconnect. The third cluster contains 2 nodes with 16 NVIDIA H100-80GB GPUs, featuring NVLink for intra-node GPU interconnects and eight 400 Gbps InfiniBand links (one per GPU) for inter-node communication.

Model, dataset and workloads. We evaluate three representative LLM models, i.e. Llama-30B [46], CodeLlama2-34B [38], and Qwen2-72B [50]. LLaMA-30B adopts standard multi-head attention (MHA), whereas CodeLlama2-34B and Qwen2-72B adopt grouped-query attention (GQA) [8]. By sharing keys and values across query heads, GQA reduces KV cache size and mitigate FuDG transmission overhead. All experiments are conducted with BF16 precision. For applications and datasets, as shown in Table 4, we follow prior studies [24,37,55] and select three representative applications with diverse input–output length distributions.

- **Alpaca-gpt4:** [1] Instruction-following dataset with short inputs and long outputs.
- **ShareGPT:** [5] Chatbot dataset featuring relatively balanced input and output lengths.
- **LongBench:** [9] Summarization dataset with long inputs (articles) and short outputs (summaries).

We set TTFT and TPOT SLOs based solely on applications, independent of model size, and in most cases stricter than those in prior work [37,55]. Each model is paired with all datasets. To emulate realistic serving, requests arrive at a fixed rate with a Poisson distribution to introduce minor fluctuations.

Baseline. We compare EcoServe against four NoDG or FuDG systems. All baselines are built on or aligned to vLLM 0.7.3 [4] as the underlying runtime, ensuring fairness.

- **vLLM [4]:** A NoDG system with separate batching and prefill-priority scheduling, which is provided in the original vLLM implementation.
- **Sarathi [7]:** A NoDG system with hybrid batching, decode-priority scheduling, and the chunked prefill technique.
- **DistServe**[1] **[55]:** An intra-node FuDG system where prefill and decode instances are colocated within the same node. DistServe also provides a cross-node strategy that limits KV cache transmission within a node, but this is only applicable to pipeline parallelism and cannot satisfy SLOs in our setting.
- **MoonCake**[2] **[37]:** An inter-node FuDG system where prefill and decode instances may reside on different nodes. MoonCake introduces a centralized KV cache pool for transmission; even when colocated on the same node, KV cache must pass through this pool. To mitigate load imbalance, we experiment with different prefill/decode ratios and adopt the optimal one.

Metrics. Following prior work, we use SLO attainments as the evaluation metrics. Specifically, we measure system throughput at different attainment levels. Throughput is obtained by incrementally increasing the request rate until the system fails to satisfy the target attainment. Following AlpaServe [26] and vLLM [4], Poisson and Gamma distributions are used to model request inter-arrival times. The Poisson process is adopted as the default setting unless otherwise specified. It produces relatively smooth arrival patterns with

[1] https://github.com/LLMServe/DistServe
[2] https://github.com/kvcache-ai/Mooncake/tree/v0.3.2 (2025.03)

moderate variability. In contrast, the Gamma process enables controllable burstiness by adjusting the coefficient of variation and is used in bursty workload evaluations. Compared to Poisson arrivals, it generates more skewed inter-arrival times, leading to sharper fluctuations where many requests may arrive within short periods followed by idle intervals. Notably, we extend the request-sending program in the vLLM repository with multithreading support, allowing us to generate high-concurrency workloads at scale.

4.2 End-to-end Evaluation

We compare EcoServe with baselines across models, datasets, and clusters. On the L20 cluster, LLaMA-30B and CodeLlama2-34B use TP=4, and Qwen2-72B uses TP=8. To mitigate MoonCake's bandwidth limits, we deploy one instance per node, yielding 8 instances in total. On A800 and H100 clusters, LLaMA-30B and CodeLlama2-34B use TP=2 (8 instances), and Qwen2-72B uses TP=4 (4 instances). We present all evaluations with three discrete SLO attainments in Figure 8, and demonstrate two evaluation cases on the L20 cluster with continuously varying SLO attainments in Figure 9. Notably, MoonCake and DistServe fail to meet SLO attainments in some cases. In addition, DistServe cannot support Qwen2-72B with TP=8 on a single node. These cases are omitted from the results.

4.2.1 Overall Comparison

EcoServe outperforms all baselines across all evaluation cases on the L20 and A800 clusters with commodity networks, which are our primary focus. For NoDG systems, EcoServe achieves an average P90 goodput improvement of 2.01× over vLLM and 1.87× over Sarathi. As shown in Figure 9, EcoServe is more resilient at high request rates, with SLO attainment degrading more slowly than NoDG systems. For FuDG systems, EcoServe improves P90 goodput by 3.43× over DistServe and 3.41× over MoonCake. Figure 9 also shows FuDG exhibits a sharp bottleneck on commodity networks, as its interconnect bandwidth and load imbalance quickly lead to SLO violations.

On the H100 cluster with high-performance interconnects, EcoServe also outperforms almost all baselines, though with smaller margins. For NoDG systems, it gains a modest P90 improvement of 1.34× over vLLM and 1.25× over Sarathi, since the strong compute capability of H100 substantially reduces prefill–decode interference. Compared with FuDG, EcoServe still improves P90 goodput by 1.75× over DistServe and 1.24× over MoonCake, somewhat surprising, as high-bandwidth networks typically favor FuDG systems. This result is mainly due to load imbalance and engineering complexity in FuDG systems, as well as the relatively lenient latency SLOs on H100, which prevent PaDG and NoDG systems from exposing their usual bottlenecks.

DistServe colocates prefill and decode instances on the same node, causing severe imbalance, and its unmaintained prototype implementation further limits performance. MoonCake, still under active development, achieves performance comparable to EcoServe and even surpasses it on CodeLlama2-34B. Although the improved interconnect and reduced imbalance on the H100 cluster allow MoonCake to better realize its potential, we still observe intermittent pauses in prefill instances, indicating that load imbalance exists and transmission buffers quickly become insufficient.

4.2.2 Comparison across models

EcoServe demonstrates consistent benefits over NoDG systems across different model scales. In particular, it achieves average throughput improvements of 1.59×, 1.83×, and 1.76× when serving Llama-30B, CodeLlama2-34B, and Qwen2-72B, respectively. In contrast, its improvements over FuDG systems vary substantially, reaching 4.82×, 2.15×, and 1.79× on the same three models.

These differences can be explained by KV cache characteristics and model scaling effects. Llama-30B suffers the largest degradation in FuDG because its relatively larger KV cache incurs substantial inter-instance transmission overhead. In comparison, CodeLlama2-34B and Qwen2-72B adopt GQA [8], which significantly reduces KV cache size and alleviates transmission bottlenecks. Moreover, because computation grows faster than KV cache footprint as model size increases (e.g., for 72B), larger models have relatively lower KV-to-compute ratios and typically tighter latency SLOs, enabling FuDG to perform relatively better.

It is also worth noting that when serving the LongBench dataset, Llama-30B consistently outperforms CodeLlama2-34B because Llama-30B supports only a maximum sequence length of 2048, causing longer inputs to be truncated and resulting in lower actual workload.

4.2.3 Comparison across clusters

Comparing EcoServe with other systems, the gains over NoDG systems remain relatively consistent across clusters, whereas the improvements over FuDG systems vary substantially. Specifically, under P90 SLO attainment, EcoServe improves goodput over NoDG systems by an average of 1.97× on L20, 1.91× on A800, and 1.29× on H100. This mainly arises from the disparity in computational capability among the GPU devices. Since we apply unified latency SLOs determined only by the datasets, GPU devices with higher computational capability can satisfy these SLOs more easily and therefore experience less prefill–decode interference, making NoDG advantageous.

Against FuDG systems, the gains are 2.45× on L20, 4.21× on A800, and 1.50× on H100, illustrating how the relative advantages shift across clusters. This disparity can be attributed

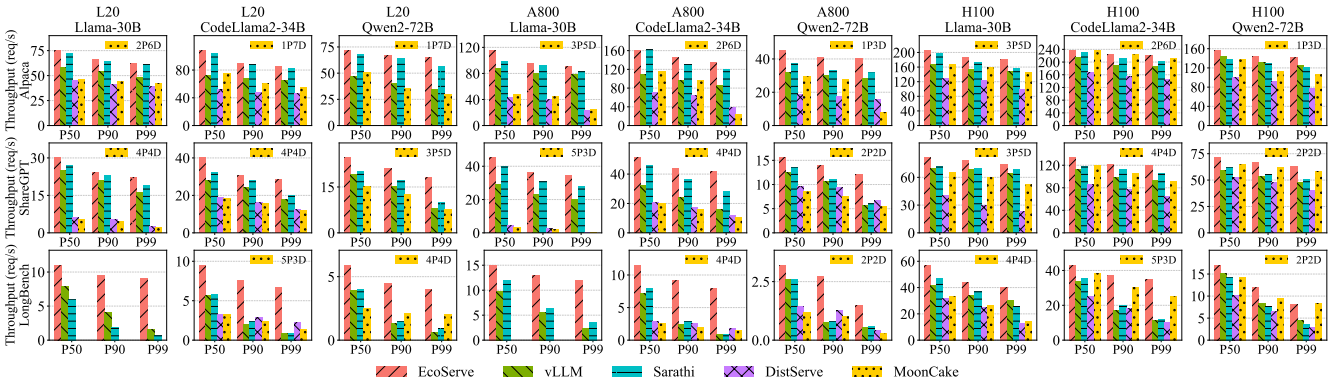
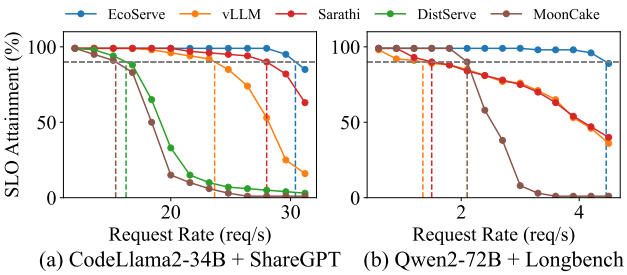


Figure 8: Evaluations across models, datasets, and clusters. The y-axis shows throughput under different SLO attainment levels.



(a) CodeLlama2-34B + ShareGPT (b) Qwen2-72B + Longbench

Figure 9: Two evaluation cases on the L20 cluster with continuously varying request rates. The vertical lines indicate the maximum rate that satisfies P90 latency SLOs.

to the ratio between computational capability and network bandwidth. A higher compute-to-bandwidth ratio generally leads to larger improvements over FuDG systems since KV cache transmission becomes the bottleneck. The A800 cluster exhibits such a ratio, as its network bandwidth increases by only 2.5× compared with L20, whereas its processing capability improves by more than 4× (Table 3), making the inter-node network an even more critical bottleneck.

4.2.4 Comparison across datasets

When comparing EcoServe with other systems, the gains vary significantly across datasets due to differences in input–output length ratios. Under P90 SLO attainment, EcoServe achieves average throughput improvements of 1.19×, 1.26×, and 2.72× over NoDG on the Alpaca, ShareGPT, and LongBench datasets, and outperforms FuDG by 1.70×, 4.00×, and 2.53×, respectively.

These differences arise from how input and output lengths affect system behavior. For NoDG systems, shorter inputs reduce prefill–decode interference and repeated KV cache accesses during chunked prefill, enabling EcoServe to gain larger advantages. For FuDG systems, datasets with longer inputs and shorter outputs require more prefill instances to

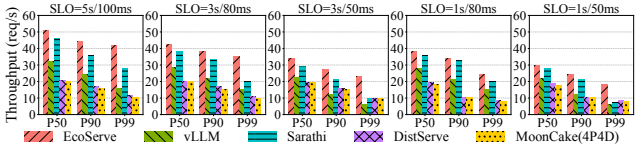


Figure 10: Evaluations under different SLOs.

generate KV cache, which increases network transmission pressure and exacerbates FuDG's bandwidth sensitivity, leading to greater improvements for EcoServe. It is also worth noting that the improvement over FuDG on LongBench would have been even higher had Llama-30B not been excluded due to execution failures.

4.3 SLO Sensitivity Study

We further evaluate SLO sensitivity by tightening the SLO from $(5s, 100ms)$ to four stricter settings: $(3s, 80ms)$, $(3s, 50ms)$, $(1s, 80ms)$, and $(1s, 50ms)$, using CodeLlama2-34B on A800 GPUs with ShareGPT.

As shown in Figure 10, as SLOs become stricter, all systems experience throughput degradation. Under the strictest $(1s, 50ms)$ SLO, EcoServe's P99 throughput decreases from 42 rps to 18 rps, a 57.1% reduction. In comparison, vLLM drops from 16 rps to 6.4 rps (60.0%), while Sarathi drops from 28 rps to 7.6 rps (72.9%), indicating that NoDG systems are more sensitive to tighter SLOs. FuDG systems, i.e., DistServe and MoonCake, show smaller relative degradation, with P99 reductions of 26.9% and 23.9%, respectively. This is because their throughput is already constrained by KV cache transmission and load imbalance under loose SLOs. Next, they show apparent robustness under stricter SLOs.

4.4 Bursty Workload Evaluation

In bursty workload evaluations, we compare throughput under both Poisson and Gamma arrival processes. To model increas-

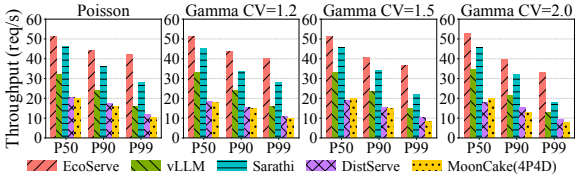


Figure 11: Evaluations under different burstiness.

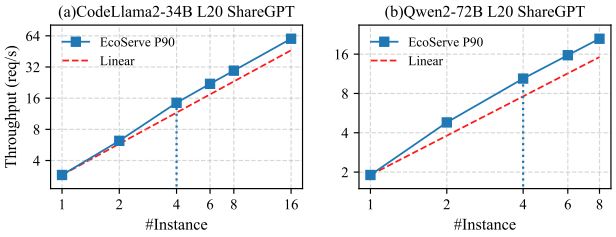


Figure 12: Static coarse-grained scaling.

ing levels of burstiness, we vary the coefficient of variation (CV) of the Gamma process as 1.2, 1.5 and 2.0, while keeping the average request rate constant. Experiments are conducted using the CodeLlama-34B model on NVIDIA A800 GPUs with the ShareGPT dataset.

In Figure 11, as workload burstiness increases, throughput degrades more significantly at higher percentiles (e.g., P90 and P99) across all systems due to burst-induced resource contention and scheduling instability, while showing almost no impact at lower percentiles (e.g., P50). Comparing these systems under P99 SLO attainment, throughput degrades by 17.3% (EcoServe), 19.5% (vLLM), 35.7% (Sarathi), 12.6% (DistServe), and 17.1% (MoonCake), respectively, as the CV increases from 1.2 to 2.0. This is because NoDG systems exhibit the weakest robustness to contention and thus suffer the largest degradation, while PaDG shows stronger resilience under resource contention caused by bursty workloads.

4.5 Scaling Capability

We evaluate the scaling capability of EcoServe from two aspects. First, we measure throughput by varying the number of instances in a macro-instance to examine static coarse-grained scaling. Second, we incrementally increase the request rate and analyze the mitosis scaling approach as the dynamic fine-grained scaling.

4.5.1 Static Coarse-grained Scaling

Static coarse-grained scaling uses CodeLlama2-34B (TP=4) and Qwen2-72B (TP=2) on the L20 cluster, with the ShareGPT dataset as the workload. Throughput under P90 SLO attainment is presented in Figure 12.

Both cases exhibit superlinear speedup initially, before reverting to linear and eventually sublinear scaling. When scaling from 1 instance (4 GPUs) to 4 instances (16 GPUs), CodeLlama2-34B achieves a 4.96× throughput increase, while Qwen2-72B reaches 5.47×. The superlinear speedup arises because EcoServe incurs minimal overhead when managing additional instances within a symmetric cluster topology. More importantly, adding instances provides greater capacity to mitigate inter-phase interference, increasing arithmetic intensity and improving GPU saturation. Assuming a macro instance contains only a single instance, the PaDG strategy actually degrades to NoDG, and two phases still switch frequently and interference severely in a single instance. This superlinear scaling effect will plateau once a sufficient number of instances is reached.

4.5.2 Dynamic Fine-grained Scaling

Fine-grained scaling uses CodeLlama2-34B (TP=4) on the L20 cluster, with the ShareGPT dataset as the workload. The request rate gradually increases from 20 to 50 requests per second in 2-minute intervals, with SLO attainments collected every 30 seconds. Based on the static scaling experiments, we set $N_l = 6$ and $N_u = 11$, since superlinear speedup disappears beyond this point and dividing a macro-instance no longer degrades performance. The system starts with 32 GPUs (8 instances) and finally uses up all 64 GPUs.

As shown in Figure 13, increasing the request rate initially results in a drop in SLO attainment, which is promptly restored once a new instance is added. This process continues until the 12th instance, where we can see that SLO attainment recovers slightly more slowly. Since the total number of instances exceeds the upper bound N_u, the system triggers a split into two macro-instances of 6 instances each. Thanks to the serializable proxy object, the split introduces only minor performance fluctuations and avoids model re-initialization. In contrast, interrupting and re-initializing an instance incurs much higher overhead. For a single instance, re-initializing CodeLlama2-34B from an L20 node's local storage takes about three minutes, and is even longer when loading weights from remote storage. Consequently, the mitosis scaling approach enables flexible, fine-grained scaling that effectively adapts to dynamic workload demands.

4.6 Parallelism Compatibility

We further design experiments to validate that FuDG is more compatible with pipeline parallelism (PP) than NoDG. As discussed in Section 2.3, PP is appealing for high-throughput inference due to its low communication overhead, but it becomes vulnerable to pipeline bubbles when prefill–decode interference is present. To evaluate this effect, we fix the TTFT SLO at 5 seconds, progressively relax the TPOT constraint, and measure throughput under different degrees of pipeline parallelism. Specifically, We evaluate CodeLlama2-34B on the ShareGPT dataset using the L20 cluster, configuring the

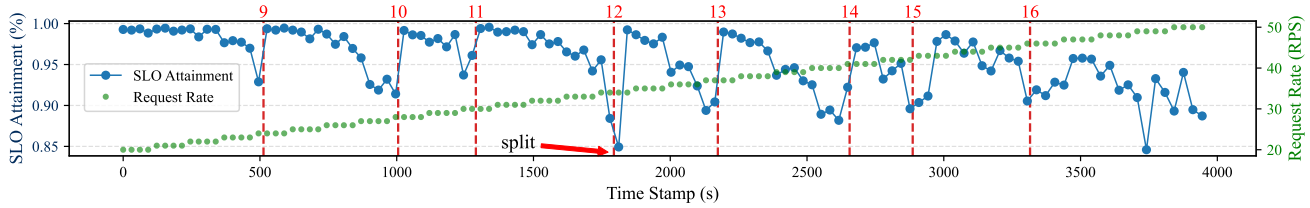


Figure 13: Dynamic fine-grained scaling experiment of CodeLlama2-34B on the L20 cluster. Here $N_l = 6$ and $N_u = 11$.

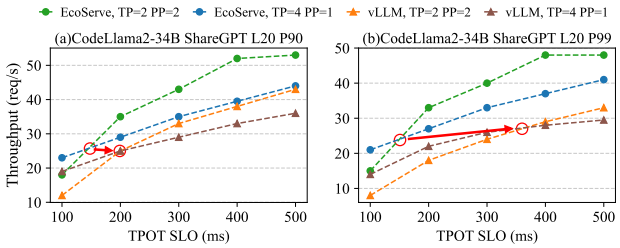


Figure 14: Pipeline parallel compatibility.

model with two settings: TP=2, PP=2 and TP=4, PP=1, which correspond to higher and lower PP degrees, respectively.

Figure 14 shows throughput as the TPOT SLO increases from 100 ms to 500 ms. The results show that EcoServe with a higher PP degree outperforms its lower-PP counterpart under tighter TPOT SLOs, surpassing that of vLLM. In other words, the performance intersection of different PP degrees occurs at a tighter TPOT constraint, and the throughput plateau achieved with PP is significantly higher than that of vLLM. These findings indicate that PaDG is substantially more compatible with pipeline parallelism, enabling PP to better reach its performance potential.

4.7 Ablation Study

We conduct an ablation study to evaluate the effectiveness of rolling activation and adaptive scheduling in EcoServe by selectively disabling them and comparing the performance.

4.7.1 Rolling Activation.

To evaluate the importance of rolling activation, we consider a variant where the next instance is selected randomly rather than following the cyclic rolling activation strategy. Experiments are conducted using all three models on NVIDIA A800 GPUs with the ShareGPT dataset. As shown in Figure 15, cyclic activation achieves higher throughput across all cases, with more pronounced gains in higher SLO attainments. This is because higher SLO attainments impose tigher scheduling constraints, leaving less slack for prefills. As a result, suboptimal scheduling strategies fail to allocate sufficient resources to meet the SLO targets. These results demonstrate that rolling activation is an effective design.

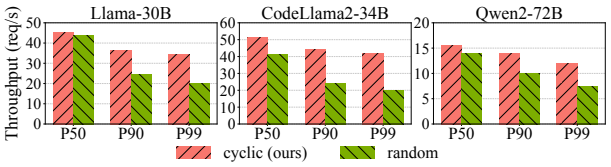


Figure 15: Ablation study of rolling activation.

Table 5: Large-scale LLM Serving Strategy Comparison.

	Goodput	Cost Effective	Load Balance	Hardware Cost	Parallelism Compatibility	Engineering Complexity
NoDG	✓	Good	Easy	Low	Low	Low
FuDG	✓✓	Poor	Hard	High	High	High
PaDG	✓✓	Excellent	Easy	Low	High	Low

4.7.2 Adaptive Scheduling Algorithm.

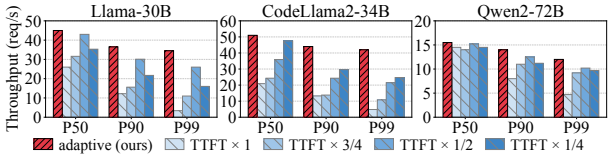


Figure 16: Ablation study of adaptive scheduling algorithm.

To evaluate the effectiveness of adaptive scheduling algorithm, We compare it with a variant that requests are routed at fixed intervals without estimating available prefill slots. Experiments are conducted across all models on NVIDIA A800 GPUs with the ShareGPT dataset. The fixed intervals are set to 1/4, 1/2, 3/4, and 1/1 of the TTFT SLO. As shown in Figure 16, while different fixed intervals lead to varying throughput, our adaptive algorithm consistently achieves the best performance, demonstrating its effectiveness.

5 Discussion

Commercial success in LLM serving hinges on adopting cost-effective strategies, which **require a careful trade-off between throughput, SLO attainment, infrastructure cost, parallelism compatibility, and even engineering complexity**. Table 5 presents a comparison between PaDG and the

existing NoDG and FuDG. In terms of goodput, PaDG is comparable to FuDG, while largely outperforming NoDG. While FuDG is designed for tight SLOs and relies on high-performance interconnects, PaDG is for cost-effective deployments that SLOs are relaxed or interconnects are limited.

Beyond hardware, PaDG also reduces load imbalance and engineering complexity. Unlike FuDG, which scales across two instance types, NoDG and PaDG scale at the granularity of individual instances, leading to simpler scaling and more balanced workloads. In addition, the lack of cross-instance KV cache transmission in PaDG and NoDG significantly lowers system complexity. From a parallelism compatibility perspective, PaDG offers further advantages. Its lower frequency of prefill-decode switching improves pipeline parallelism efficiency, while minimal data movement and reduced PCIe contention make it more suitable for tensor parallelism in commodity nodes.

NoDG, PaDG, and FuDG each have their own advantageous scenarios, and LLM serving vividly demonstrates the art of trade-offs in system optimization. NoDG is well-suited for small models, such as 7B and 13B. These models have lower computational demands, and their SLOs are easier to satisfy, making prefill-decode interference negligible. Larger models, such as 30B, 70B, and 130B, benefit more from PaDG. These models typically require parallel techniques to extend memory capacity and are still capable of meeting typical latency SLOs in a single instance. In extreme scenarios, such as ultra-large models or stringent SLOs, even minor interference can significantly degrade these metrics. In such cases, PaDG with commodity hardware becomes insufficient. To satisfy the TPOT SLO, it can be enhanced with lightweight add-on strategies (e.g., chunked prefill) or augmented with advanced hardware, while designs such as FuDG with specialized hardware may ultimately be required. More aggressive strategies are worth studying. For example, MegaScale-Infer [57] and Eaas [32] study ultra-large MoE model and disaggregates the attention and FFN modules into different instances. Moreover, these strategies incur incremental engineering costs, which also serve as a major factor.

6 Related Work

Scheduling in LLM serving. Based on whether prefill and decode phases are disaggregated, existing LLM serving approaches can be categorized into the NoDG strategy [3, 4, 7, 52] and the FuDG strategy [35, 37, 55], which are most relevant to EcoServe. Adrenaline [28] notices the load imbalance issue in FuDG and reschedules computation in prefill and decode instances.

Moreover, other studies address issues in specific inference scenarios. Flexgen [42], FastDecode [19] and Specinfer [33] enable LLM inference with limited memory capacity by employing offloading strategies. Loongserve [49] and Infinite-llm [29] targets long-context inference and optimize parallel strategy and memory utilization respectively. Moelightning [11], Pre-gated MoE [21] and Lina [25] focus on MoE models and optimize resource utilization by employing expert popularity. MegaScale-Infer [57] and Eaas [32] target ultra-large MoE model and accelerate the decode phase by disaggregating the attention and FFN modules. Liger [13] and NanoFlow [56] carefully schedules and overlaps GPU kernels from different requests to improve efficiency.

KV Cache Management. To reduce KV cache memory usage of standard MHA [48], MQA [40] and GQA [8] share key and value projections across query heads. PagedAttention [24] and vAttention [36] reduces memory fragmentation by organizing the KV cache into fixed-size blocks. To compress KV cache, H2O [54], Keyformer [6], and Liu et al. [31] find token similarity and removes redundant information. Next, Shadowkv [45], Prompt cache [17], and Ragcache [23] further explore KV cache compression and offloading strategies in long-context scenarios. AttentionStore [16] schedules KV cache across hierarchical storage tiers, while CacheBlend [51] introduces the pipelining loading with partial recomputation to use slower object stores.

7 Conclusion

EcoServe is built on the novel partially disaggregated (PaDG) strategy for cluster-level LLM serving. PaDG mitigates prefill–decode interference by enabling multiple instances to collaborate in a data-reduced manner, well suited for GPU clusters with the commodity network. EcoServe also excels in load balancing, hardware cost, parallelism compatibility, and engineering simplicity, making it a cost-effective choice.

Acknowledgments

This research is supported by the National Key R&D Program of China under Grant No.2025YFB3003501, the National Natural Science Foundation of China under Grant No.62402534, the GuangDong Basic and Applied Basic Research Foundation under Grant No.2023A1515110117, the Yunnan Provincial Major Science and Technology Special Plan Projects under Grant No.202502AD080009. We also thank the WeChat Group at Tencent for valuable discussions and for providing access to a commodity GPU cluster that reflects practical production settings.

References

[1] Alpaca-GPT4. https://github.com/tatsu-lab/alpaca_eval, 2023. Instruction-following dataset generated using GPT-4.

[2] Faster transformer. https://github.com/NVIDIA/FasterTransformer, 2024.

[3] Sglang. https://github.com/sgl-project/sglang, 2024.

[4] vllm: Easy, fast, and cheap llm serving for everyone. https://github.com/vllm-project/vllm, 2024.

[5] Hugginface sharegpt. https://huggingface.co/datasets/anon8231489123/ShareGPT_Vicuna_unfiltered, 2025.

[6] Muhammad Adnan, Akhil Arunkumar, Gaurav Jain, Prashant Nair, Ilya Soloveychik, and Purushotham Kamath. Keyformer: Kv cache reduction through key tokens selection for efficient generative inference. Proceedings of Machine Learning and Systems, 6:114–127, 2024.

[7] Amey Agrawal, Nitin Kedia, Ashish Panwar, Jayashree Mohan, Nipun Kwatra, Bhargav Gulavani, Alexey Tumanov, and Ramachandran Ramjee. Taming {Throughput-Latency} tradeoff in {LLM} inference with {Sarathi-Serve}. In 18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24), pages 117–134, 2024.

[8] Joshua Ainslie, James Lee-Thorp, Michiel de Jong, Yury Zemlyanskiy, Federico Lebrón, and Sumit Sanghai. Gqa: Training generalized multi-query transformer models from multi-head checkpoints, 2023.

[9] Yushi Bai, Xin Lv, Jiajie Zhang, Hongchang Lyu, Jiankai Tang, Zhidian Huang, Zhengxiao Du, Xiao Liu, Aohan Zeng, Lei Hou, Yuxiao Dong, Jie Tang, and Juanzi Li. Longbench: A bilingual, multitask benchmark for long context understanding, 2023. arXiv preprint.

[10] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. Language models are few-shot learners. Advances in neural information processing systems, 33:1877–1901, 2020.

[11] Shiyi Cao, Shu Liu, Tyler Griggs, Peter Schafhalter, Xiaoxuan Liu, Ying Sheng, Joseph E Gonzalez, Matei Zaharia, and Ion Stoica. Moe-lightning: High-throughput moe inference on memory-constrained gpus. In Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1, pages 715–730, 2025.

[12] Shenggan Cheng, Ziming Liu, Jiangsu Du, and Yang You. Atp: Adaptive tensor parallelism for foundation models. arXiv preprint arXiv:2301.08658, 2023.

[13] Jiangsu Du, Jinhui Wei, Jiazhi Jiang, Shenggan Cheng, Dan Huang, Zhiguang Chen, and Yutong Lu. Liger: Interleaving intra-and inter-operator parallelism for distributed large model inference. In Proceedings of the 29th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming, pages 42–54, 2024.

[14] Yangyang Feng, Minhui Xie, Zijie Tian, Shuo Wang, Youyou Lu, and Jiwu Shu. Mobius: Fine tuning large-scale models on commodity gpu servers. In Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2, pages 489–501, 2023.

[15] Futuriom. What's next for networking infrastructure for ai. Technical report, Futuriom Research, 2025. Industry analysis report.

[16] Bin Gao, Zhuomin He, Puru Sharma, Qingxuan Kang, Djordje Jevdjic, Junbo Deng, Xingkun Yang, Zhou Yu, and Pengfei Zuo. Attentionstore: Cost-effective attention reuse across multi-turn conversations in large language model serving. arXiv e-prints, pages arXiv–2403, 2024.

[17] In Gim, Guojun Chen, Seung-seob Lee, Nikhil Sarda, Anurag Khandelwal, and Lin Zhong. Prompt cache: Modular attention reuse for low-latency inference. Proceedings of Machine Learning and Systems, 6:325–338, 2024.

[18] Aaron Grattafiori, Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Alex Vaughan, et al. The llama 3 herd of models. arXiv preprint arXiv:2407.21783, 2024.

[19] Jiaao He and Jidong Zhai. Fastdecode: High-throughput gpu-efficient llm serving using heterogeneous pipelines. arXiv preprint arXiv:2403.11421, 2024.

[20] Yanping Huang, Youlong Cheng, Ankur Bapna, et al. Gpipe: Efficient training of giant neural networks using pipeline parallelism. In Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, December 8-14, 2019, Vancouver, BC, Canada, pages 103–112, 2019.

[21] Ranggi Hwang, Jianyu Wei, Shijie Cao, Changho Hwang, Xiaohu Tang, Ting Cao, and Mao Yang. Pregated moe: An algorithm-system co-design for fast and scalable mixture-of-expert inference. In 2024 ACM/IEEE 51st Annual International Symposium on Computer Architecture (ISCA), pages 1018–1031. IEEE, 2024.

[22] Sam Ade Jacobs, Masahiro Tanaka, Chengming Zhang, Minjia Zhang, Shuaiwen Leon Song, Samyam Rajbhandari, and Yuxiong He. Deepspeed ulysses: System optimizations for enabling training of extreme long sequence transformer models, 2023.

[23] Chao Jin, Zili Zhang, Xuanlin Jiang, Fangyue Liu, Xin Liu, Xuanzhe Liu, and Xin Jin. Ragcache: Efficient knowledge caching for retrieval-augmented generation. arXiv preprint arXiv:2404.12457, 2024.

[24] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph Gonzalez, Hao Zhang, and Ion Stoica. Efficient memory management for large language model serving with pagedattention. In Proceedings of the 29th Symposium on Operating Systems Principles, pages 611–626, 2023.

[25] Jiamin Li, Yimin Jiang, Yibo Zhu, Cong Wang, and Hong Xu. Accelerating distributed {MoE} training and inference with lina. In 2023 USENIX Annual Technical Conference (USENIX ATC 23), pages 945–959, 2023.

[26] Zhuohan Li, Lianmin Zheng, Yinmin Zhong, Vincent Liu, Ying Sheng, Xin Jin, Yanping Huang, Zhifeng Chen, Hao Zhang, Joseph E Gonzalez, et al. {AlpaServe}: Statistical multiplexing with model parallelism for deep learning serving. In 17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23), pages 663–679, 2023.

[27] Zhuohan Li, Siyuan Zhuang, Shiyuan Guo, Danyang Zhuo, Hao Zhang, Dawn Song, and Ion Stoica. Terapipe: Token-level pipeline parallelism for training large-scale language models. In International Conference on Machine Learning, pages 6543–6552. PMLR, 2021.

[28] Yunkai Liang, Zhangyu Chen, Pengfei Zuo, Zhi Zhou, Xu Chen, and Zhou Yu. Injecting adrenaline into llm serving: Boosting resource utilization and throughput via attention disaggregation. arXiv preprint arXiv:2503.20552, 2025.

[29] Bin Lin, Chen Zhang, Tao Peng, Hanyu Zhao, Wencong Xiao, Minmin Sun, Anmin Liu, Zhipeng Zhang, Lanbo Li, Xiafei Qiu, et al. Infinite-llm: Efficient llm service for long context with distattention and distributed kvcache. arXiv preprint arXiv:2401.02669, 2024.

[30] Hao Liu, Matei Zaharia, and Pieter Abbeel. Ring attention with blockwise transformers for near-infinite context. arXiv preprint arXiv:2310.01889, 2023.

[31] Shu Liu, Asim Biswal, Audrey Cheng, Xiangxi Mo, Shiyi Cao, Joseph E Gonzalez, Ion Stoica, and Matei Zaharia. Optimizing llm queries in relational workloads. arXiv preprint arXiv:2403.05821, 2024.

[32] Ziming Liu, Boyu Tian, Guoteng Wang, Zhen Jiang, Peng Sun, Zhenhua Han, Tian Tang, Xiaohe Hu, Yanmin Jia, Yan Zhang, et al. Expert-as-a-service: Towards efficient, scalable, and robust large-scale moe serving. arXiv preprint arXiv:2509.17863, 2025.

[33] Xupeng Miao, Gabriele Oliaro, Zhihao Zhang, Xinhao Cheng, Zeyu Wang, Zhengxin Zhang, Rae Ying Yee Wong, Alan Zhu, Lijie Yang, Xiaoxiang Shi, et al. Specinfer: Accelerating generative large language model serving with tree-based speculative inference and verification. arXiv preprint arXiv:2305.09781, 2023.

[34] Deepak Narayanan, Aaron Harlap, Amar Phanishayee, Vivek Seshadri, Nikhil R Devanur, Gregory R Ganger, Phillip B Gibbons, and Matei Zaharia. Pipedream: Generalized pipeline parallelism for dnn training. In Proceedings of the 27th ACM Symposium on Operating Systems Principles, pages 1–15, 2019.

[35] Pratyush Patel, Esha Choukse, Chaojie Zhang, Aashaka Shah, Íñigo Goiri, Saeed Maleki, and Ricardo Bianchini. Splitwise: Efficient generative llm inference using phase splitting. In 2024 ACM/IEEE 51st Annual International Symposium on Computer Architecture (ISCA), pages 118–132. IEEE, 2024.

[36] Ramya Prabhu, Ajay Nayak, Jayashree Mohan, Ramachandran Ramjee, and Ashish Panwar. vattention: Dynamic memory management for serving llms without pagedattention. arXiv preprint arXiv:2405.04437, 2024.

[37] Ruoyu Qin, Zheming Li, Weiran He, Jialei Cui, Feng Ren, Mingxing Zhang, Yongwei Wu, and Weimin Zheng. Mooncake: Trading more storage for less computation — a KVCache-centric architecture for serving LLM chatbot. In 23rd USENIX Conference on File and Storage Technologies (FAST 25), pages 155–170, Santa Clara, CA, February 2025. USENIX Association.

[38] Baptiste Rozière, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Romain Sauvestre, Tal Remez, Jérémy Rapin, Artyom Kozhevnikov, Ivan Evtimov, Joanna Bitton, Manish Bhatt, Cristian Canton Ferrer, Aaron Grattafiori, Wenhan Xiong, Alexandre Défossez, Jade Copet, Faisal Azhar, Hugo Touvron, Louis Martin, Nicolas Usunier, Thomas Scialom, and Gabriel Synnaeve. Code llama: Open foundation models for code, 2024.

[39] ServerMall. Infrastructure for machine learning and generative ai: Evolution and future directions. Medium article, 2024. Industry infrastructure analysis.

[40] Noam Shazeer. Fast transformer decoding: One write-head is all you need, 2019. URL https://arxiv. org/abs, 1911.

[41] Noam Shazeer, Youlong Cheng, Niki Parmar, Dustin Tran, Ashish Vaswani, Penporn Koanantakool, Peter Hawkins, HyoukJoong Lee, Mingsheng Hong, Cliff Young, et al. Mesh-tensorflow: Deep learning for supercomputers. Advances in neural information processing systems, 31, 2018.

[42] Ying Sheng, Lianmin Zheng, Binhang Yuan, Zhuohan Li, Max Ryabinin, Beidi Chen, Percy Liang, Christopher Ré, Ion Stoica, and Ce Zhang. Flexgen: High-throughput generative inference of large language models with a single gpu. In International Conference on Machine Learning, pages 31094–31116. PMLR, 2023.

[43] Mohammad Shoeybi, Mostofa Patwary, Raul Puri, et al. Megatron-lm: Training multi-billion parameter language models using model parallelism. CoRR, abs/1909.08053, 2019.

[44] Siddharth Singh, Olatunji Ruwase, Ammar Ahmad Awan, Samyam Rajbhandari, Yuxiong He, and Abhinav Bhatele. A hybrid tensor-expert-data parallelism approach to optimize mixture-of-experts training. In Proceedings of the 37th International Conference on Supercomputing, pages 203–214, 2023.

[45] Hanshi Sun, Li-Wen Chang, Wenlei Bao, Size Zheng, Ningxin Zheng, Xin Liu, Harry Dong, Yuejie Chi, and Beidi Chen. Shadowkv: Kv cache in shadows for high-throughput long-context llm inference. arXiv preprint arXiv:2410.21465, 2024.

[46] Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, et al. Llama: Open and efficient foundation language models. arXiv preprint arXiv:2302.13971, 2023.

[47] TrendForce. Ai server gpu market analysis. Technical report, TrendForce, 2024. Market analysis report.

[48] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. Advances in neural information processing systems, 30, 2017.

[49] Bingyang Wu, Shengyu Liu, Yinmin Zhong, Peng Sun, Xuanzhe Liu, and Xin Jin. Loongserve: Efficiently serving long-context large language models with elastic sequence parallelism. In Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles, pages 640–654, 2024.

[50] An Yang, Baosong Yang, Binyuan Hui, and et al. Qwen2 technical report, 2024.

[51] Jiayi Yao, Hanchen Li, Yuhan Liu, Siddhant Ray, Yihua Cheng, Qizheng Zhang, Kuntai Du, Shan Lu, and Junchen Jiang. Cacheblend: Fast large language model serving for rag with cached knowledge fusion. In Proceedings of the Twentieth European Conference on Computer Systems, pages 94–109, 2025.

[52] Gyeong-In Yu, Joo Seong Jeong, Geon-Woo Kim, Soojeong Kim, and Byung-Gon Chun. Orca: A distributed serving system for {Transformer-Based} generative models. In 16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22), pages 521–538, 2022.

[53] Hongbin Zhang, Taosheng Wei, Zhenyi Zheng, Jiangsu Du, Zhiguang Chen, and Yutong Lu. Td-pipe: Temporally-disaggregated pipeline parallelism architecture for high-throughput llm inference. arXiv preprint arXiv:2506.10470, 2025.

[54] Zhenyu Zhang, Ying Sheng, Tianyi Zhou, Tianlong Chen, Lianmin Zheng, Ruisi Cai, Zhao Song, Yuandong Tian, Christopher Ré, Clark Barrett, et al. H2o: Heavy-hitter oracle for efficient generative inference of large language models. Advances in Neural Information Processing Systems, 36:34661–34710, 2023.

[55] Yinmin Zhong, Shengyu Liu, Junda Chen, Jianbo Hu, Yibo Zhu, Xuanzhe Liu, Xin Jin, and Hao Zhang. {DistServe}: Disaggregating prefill and decoding for goodput-optimized large language model serving. In 18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24), pages 193–210, 2024.

[56] Kan Zhu, Yilong Zhao, Liangyu Zhao, Gefei Zuo, Yile Gu, Dedong Xie, Yufei Gao, Qinyu Xu, Tian Tang, Zihao Ye, et al. Nanoflow: Towards optimal large language model serving throughput. arXiv preprint arXiv:2408.12757, 2024.

[57] Ruidong Zhu, Ziheng Jiang, Chao Jin, Peng Wu, Cesar A. Stuardo, Dongyang Wang, Xinlei Zhang, Huaping Zhou, Haoran Wei, Yang Cheng, Jianzhe Xiao, Xinyi Zhang, Lingjun Liu, Haibin Lin, Li-Wen Chang, Jianxi Ye, Xiao Yu, Xuanzhe Liu, Xin Jin, and Xin Liu. Megascale-infer: Serving mixture-of-experts at scale with disaggregated expert parallelism, 2025.

Revisiting Pipeline Parallelism for LLM Serving

Soonjae Hwang Jeongseob Ahn

Korea University

Abstract

As the memory capacity of a single GPU is insufficient to accommodate large language models (LLMs), model parallelism has become the standard approach for serving LLMs across multiple GPUs. In online serving environments, tensor parallelism has become the de facto way in single-node multi-GPU systems because it can reduce the computation latency through parallel execution. Although pipeline parallelism can offer higher throughput, it suffers from pipeline imbalance that is exacerbated under online workloads, leading to resource underutilization and performance degradation.

In this study, we revisit pipeline parallelism for serving LLMs. Our analysis shows that computational imbalance between pipeline stages becomes exacerbated in online serving. To address these pipeline inefficiencies, we propose three techniques: two mechanisms, greedy and predictive schemes, that dynamically adjust the chunk size to mitigate prefill-induced bubbles, and a delay scheduling technique that dynamically rebalances decode workloads across pipeline stages to further reduce pipeline bubbles. We implement our techniques on top of SGLang and demonstrate that, for Qwen2.5 32B and 14B on four NVIDIA A100 40GB GPUs, pipeline parallelism with our mechanisms outperforms tensor parallelism.

1 Introduction

As large language models (LLMs) with billions of parameters have become increasingly prevalent, their inference workloads demand far more computation and memory than a single accelerator (e.g., GPU or NPU) can provide. For instance, the Llama3.1-70B [4] BF16 model requires at least four NVIDIA A100 40GB GPUs to retain its parameters. This necessitates parallel execution across multiple accelerators. In single-node multi-GPU inference, tensor parallelism has become the de facto standard [30]. By partitioning large weight matrices and attention operations across GPUs, tensor parallelism achieves low per-token latency [16], particularly with high-bandwidth interconnects (e.g., NVLink) that hide frequent all-reduce communication overhead [27]. In addition, every GPU executes the same set of operations at each step, making load balancing trivial. However, many commodity GPUs and NPUs still rely on traditional PCIe interconnects, which do not provide high-bandwidth peer-to-peer communication. In such settings, the substantial communication volume in tensor parallelism can quickly become a performance bottleneck, making it less suitable for online LLM serving.

In contrast, pipeline parallelism partitions the model along its depth, passing microbatches through sequential pipeline stages. This dramatically reduces inter-device communication and can achieve higher throughput than tensor parallelism by overlapping stage execution [11, 17], making it attractive for bandwidth constrained deployments such as PCIe.

This paper revisits pipeline parallelism to achieve high performance by effectively utilizing multiple GPUs for LLM serving. We first conduct an in-depth analysis of pipeline parallelism in the context of online serving. Our analysis identifies that computational imbalance between pipeline stages becomes exacerbated in online serving, as variability in request arrival times and input lengths leads to microbatches with different computational demands, resulting in *pipeline bubbles*. For example, if one microbatch consists entirely of decode-phase requests while another includes prefill-phase requests, the per-stage workloads differ significantly. Even when all microbatches across stages consist of decode-phase requests (e.g., decode-heavy), pipeline imbalance still occurs when one microbatch handles many decode-phase requests while another handles only a few.

To this end, we propose three techniques for pipeline parallelism to achieve high throughput while satisfying the latency requirements (e.g., SLO: Service Level Objective) for both TTFT (Time To First Token) and TPOT (Time Per Output Token). Our strategy is to leverage chunked-prefill [2], which splits long input sequences into smaller, fixed-size chunks. Since prefill computation is proportional to its input length, limiting the input with a fixed chunk size helps reduce latency imbalance across pipeline workers. However, smaller chunks lower computational throughput due to fewer tokens being

processed per iteration, leading to higher TTFT latency.

To balance this trade-off between minimizing pipeline bubbles and maximizing compute utilization, we propose two methods that *dynamically adjust the chunk size*. The first is a greedy approach, in which the LLM serving framework monitors the latency slack for both TTFT and TPOT at every iteration based on a given SLO, and incrementally increases or decreases the chunk size to balance the two phases. Specifically, when the observed $TPOT_{slack}$ is about to violate its SLO, we decrease the chunk size to reduce pipeline bubbles; when the $TTFT_{slack}$ is insufficient, we increase the chunk size to drain the waiting queue faster. This simple feedback-driven requires only minimal changes to existing serving systems.

However, the greedy approach is inherently reactive and thus limited in its ability to accurately determine the optimal chunk size under the system's current context. To overcome this limitation, we propose a *predictive dynamic chunked prefill* method that proactively selects the chunk size using a lightweight latency prediction model. The model combines offline profiling with online adaptation to account for both static and dynamic components of LLM iteration latency. Given the current system state (e.g., waiting queue, running batch, and request arrival rate), the inference serving scheduler uses this model to select the chunk size that minimizes pipeline bubbles, thereby improving goodput.

The last piece of this study is to dynamically rebalance workloads across pipeline stages, called *delay scheduling*. We find that existing implementations of pipeline parallelism typically rely on static microbatch scheduling. Once a request is assigned to a microbatch, it remains fixed for its entire lifetime [14, 37]. In decode-heavy workloads where many of the batches consist of only decode phases, such static scheduling amplifies pipeline imbalances and repeatedly causes stalls across iterations. To alleviate this, our delay scheduling examines load imbalances (i.e., # requests) at each scheduling iteration and migrates requests from the heaviest to the lightest microbatch in proportion to their computational demand.

We implement our three techniques, as well as the baseline pipeline parallelism with chunked prefill, on top of SGLang [37]. We evaluate the effectiveness of our techniques using two representative LLM models, Qwen2.5 32B and 14B, on a server equipped with four NVIDIA A100 40GB GPUs. To emulate real-world LLM request patterns, we use the `Conversation` trace from Azure [22], the `ShareGPT` trace from HuggingFace [24], and the `CNN` trace [9]. Since these are prefill-heavy workloads, we also construct a decode-heavy workload by reversing the input and output distribution of the CNN trace. We demonstrate that pipeline parallelism with a carefully tuned static chunk size already outperforms tensor parallelism in terms of goodput, TTFT, and TPOT. Our dynamic chunked prefill and delay scheduling techniques improve further: on the `Conversation` workload, they reduce TPOT and end-to-end latency by 35% and 31%, respectively, over this tuned pipeline parallelism.

2 Background

2.1 LLM Inference and Serving

LLMs typically execute inference in two distinct phases: *prefill* and *decode*. In the prefill phase, the input prompt tokens are processed through all decoder layers to initialize the attention key-value (KV) cache and produce the first output token. After this step, the request enters the decode phase, where tokens are generated autoregressively, one token at a time [29]. Each decode step reuses the KV cache and computes attention only for the newly generated token, resulting in a near-constant and significantly lower per-token cost compared to prefill. The number of decode steps (i.e., output tokens) depends on its input prompt.

Meanwhile, modern LLM serving systems commonly employ iteration-level scheduling [28, 35], where multiple requests are grouped into the same batch and processed at the granularity of iteration. The iteration-level scheduling allows new requests to join ongoing batches when GPU resources are available, and completed requests to exit early without waiting for the entire batch to finish. In online serving environments, requests often enter and exit the prefill and decode phases at different times. Batching at the iteration-level enables efficient GPU utilization by allowing requests in different phases to be processed together.

However, while iteration-level scheduling enhances GPU utilization, it can increase the decoding latency when long prefill requests are included in the same iteration. To mitigate this, Sarathi-Serve introduces the *chunked-prefill* technique, which splits the prefill phase into fixed-size chunks [2]. This approach distributes the compute load of long prompts across multiple iterations, preventing them from dominating a single iteration. The chunked-prefill approach has been widely adopted in modern LLM serving frameworks such as vLLM [14], SGLang [37], and Dynamo [19].

In online LLM serving, TTFT (Time to First Token) and TPOT (Time Per Output Token) are commonly used latency metrics to evaluate performance during the prefill and decode phases, respectively. According to the MLPerf inference benchmark [1], typical SLO targets are 2,000ms for TTFT and 200ms for TPOT. From the perspective of operating LLM-based services, it is crucial to achieve high throughput while satisfying these latency targets, referred to as goodput.

2.2 Parallelism in LLM Serving

Although each new GPU generation increases memory capacity, it still falls short of accommodating the weights of large models. For instance, serving the Llama 3.1 70B FP16 model, which requires approximately 140GB of memory, necessitates at least two H100 GPUs. A common practice is to employ model parallelism, which partitions the model across multiple GPUs. There are two primary forms of model parallelism:

tensor parallelism and *pipeline parallelism*.

Tensor Parallelism (TP) shards intra-layer tensors across multiple GPUs [25, 27]. In a transformer block, large weight matrices, such as those in attention projections or feedforward layers, are split across devices. Each GPU performs partial matrix multiplications (e.g., GEMMs) and attention operations on its shard. The partial outputs are then aggregated via collective communication primitives like all-reduce, preserving the logical semantics of the full layer. Since every GPU holds all layers in a sharded form, each forward pass involves all participating devices.

Pipeline Parallelism (PP) partitions the model by layer depth, assigning contiguous sets of layers (called `stages`) to different GPUs [11, 17]. During inference, a microbatch, a smaller split of the full input batch, is passed sequentially through these pipeline stages. While stage 0 processes microbatch i+1, stage 1 can process microbatch i, enabling concurrent execution and hiding latency through pipelining. Unlike TP, each GPU is only responsible for a subset of layers, reducing communication overhead but introducing the potential for pipeline bubbles (later discussed in detail).

In single-node multi-GPU environments, tensor parallelism has become the de facto standard for latency-sensitive online LLM serving [30]. While tensor parallelism introduces substantial communication overhead, especially compared to pipeline parallelism, this cost is largely mitigated by high-bandwidth interconnects such as NVLink [27]. However, NVLink is only available in a limited set of GPU platforms, such as NVIDIA DGX systems with A100 or H100 GPUs. Still, many GPU servers in public clouds rely on PCIe, which offers lower bandwidth. The same constraint applies to AMD GPUs and NPUs. In such environments, pipeline parallelism may offer better performance due to its lower communication requirements. Another reason tensor parallelism is often preferred is that it is generally simpler to implement and integrate, avoiding the scheduling complexity and microbatching challenges inherent in pipeline parallelism. As a result, modern LLM serving frameworks remain relatively immature in their support for efficient pipeline parallelism.

2.3 Tensor Parallelism vs. Pipeline Parallelism

To have a better understanding of the advantages and disadvantages of pipeline parallelism, we conduct a performance analysis comparing it with tensor parallelism on four NVIDIA A100 GPUs connected via PCIe 4.0 to the host. We evaluate TTFT and TPOT latency when serving Qwen2.5-32B with SGLang [37], using the Azure `Conversation` workload [22] under varying request rates. The detailed experimental setup is described in Section 5.1.

Figure 1 presents TTFT (top) and TPOT (bottom) performance for both pipeline parallelism and tensor parallelism. Pipeline parallelism achieves lower TTFT latency than tensor parallelism due to its pipelined execution across stages, which enables efficient overlap of computation during the prefill phase and accelerates the generation of the first token. However, pipeline parallelism exhibits relatively higher TPOT latency compared to tensor parallelism. This is primarily due to *pipeline bubbles* across different pipeline stages as depicted in Figure 2. We will discuss more details in the following section. As the request rate increases, these bubbles become more significant, causing pipeline parallelism to violate TPOT_{SLO}. The vertical dotted lines denote the maximum achievable goodput given latency budgets of 2,000 ms for TTFT and 200 ms for TPOT [1]. In terms of SLO attainment, tensor parallelism can sustain a 1.2× higher request rate than pipeline parallelism.

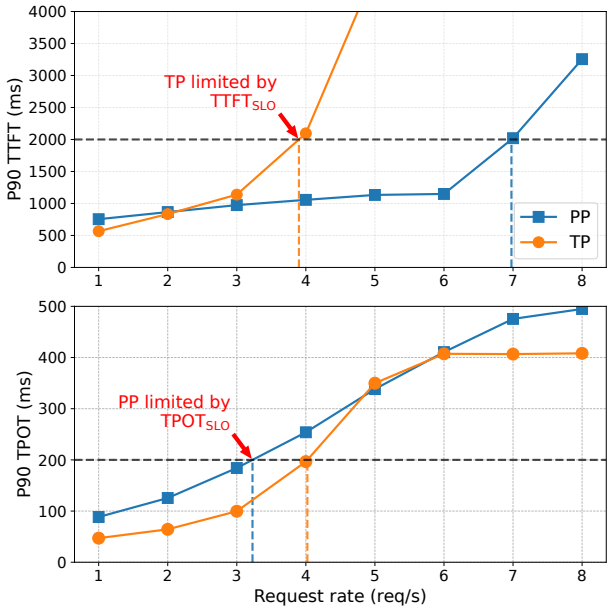


Figure 1: Performance comparison for pipeline and tensor parallelism in serving Qwen2.5-32B with the Azure Conversation workload on 4 NVIDIA A100 GPUs

3 Challenges and Proposed Approaches

This section describes challenges in serving LLMs with pipeline parallelism. First, we analyze the root cause of pipeline bubbles that lead to computational inefficiencies. Second, we provide an overview of our design approaches to reduce the pipeline bubbles.

3.1 Pipeline Bubbles in LLM Serving

The root cause of pipeline bubbles lies in computational imbalances between pipeline stages (i.e., GPU workers). Unlike model training, where inputs are fixed and predictable, online serving introduces inherent variability in request arrival

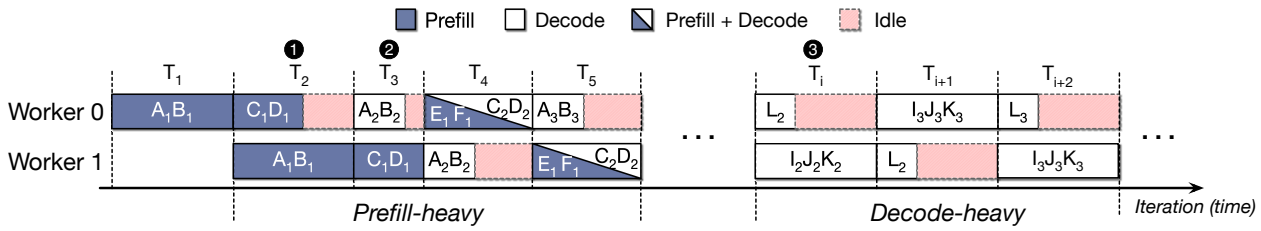


Figure 2: Three types of pipeline bubbles (The subscript indicates the iteration number.)

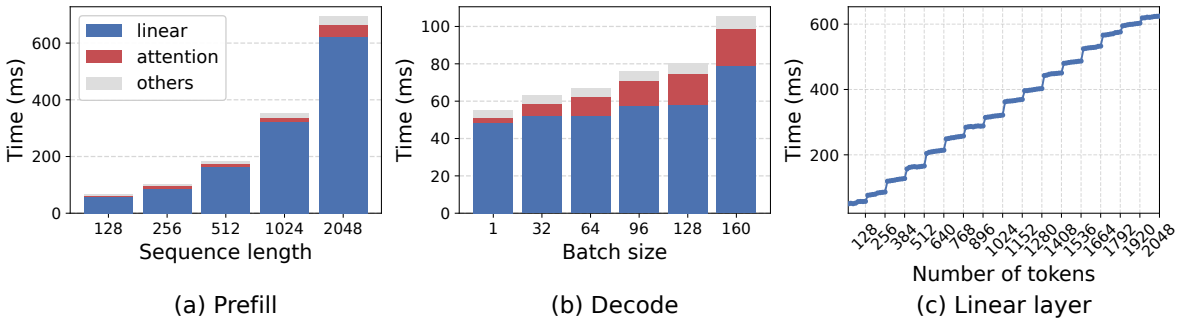


Figure 3: Performance characteristics: (a) prefill time when varying the sequence length, (b) decode time when increasing the batch size, and (c) linear operations when increasing the number of tokens

times and batch composition, making it difficult to maintain balanced compute across pipeline stages.

3.1.1 Types of Pipeline Imbalance

Figure 2 illustrates three representative cases that lead to pipeline bubbles, assuming a two-stage pipeline and a maximum batch size of three. First, the iteration-level scheduling composes batches of requests with varying input lengths, leading to imbalanced prefill latencies. For instance, ❶ at T_2, suppose a microbatch A_1B_1 with long prompt sequences and another microbatch C_1D_1 with shorter ones. Microbatches with longer prompts spend more computation cycles during the prefill phase. In this case, `Worker 0` completes the microbatch C_1D_1 earlier, while `Worker 1` continues executing the prefill for A_1B_1. As a result, `Worker 0` cannot start the next microbatch A_2B_2 until `Worker 1` finishes its stage, leading to idle cycles. We call this *Prefill-Prefill (P-P) imbalance*.

Second, ❷ at T_3, when one microbatch (A_2B_2) consists solely of decode-phase requests while another microbatch (C_1D_1) consists solely of prefill-phase requests, the computational workload diverges sharply across stages. We call this *Prefill-Decode (P-D) imbalance*. Here, `Worker 0` handling the decode-only microbatch finishes earlier and remains idle, while `Worker 1` handling the prefill-only microbatch continues execution, further exacerbating pipeline imbalance, increasing TPOT for microbatch A_2B_2. Another P-D imbalance case is shown at T_4, where microbatch $C_2D_2E_1F_1$, assigned to `Worker 0`, includes two prefill and two decode requests, while microbatch A_2B_2 on `Worker 1` consists only of decode requests. In this case, `Worker 0` becomes the bottleneck, delaying the initiation of microbatch A_3B_3.

Third, ❸ from T_i to T_{i+2}, when all microbatches across pipeline stages contain only decode phase requests, this leads to another form of pipeline imbalance where one microbatch handles many such requests while another handles only a few. We refer to this as *Decode-Decode (D-D) imbalance*.

3.1.2 Performance Implication of Pipeline Imbalance

To precisely quantify pipeline imbalance, we analyze the performance of prefill and decode phases by varying the sequence length and batch size, respectively. First, Figure 3a shows that the prefill latency depends largely on the input sequence length, indicating that balancing the compute density of prefill stages across workers is critical. For instance, when two prefills with sequence lengths of 2048 and 128 are issued sequentially, the resulting *P-P imbalance* generates a substantial pipeline bubble, around 627ms, which consequently stalls subsequent pipeline stages.

Second, during the decode phase, the batch size (i.e., number of requests) becomes the dominant factor determining execution time, given that only a single token is processed at a time. Figure 3b shows that the decode execution time increases as the batch size grows. Although the attention operation becomes more significant in the decode phase, linear operations remain the primary source of execution time, exhibiting a distinct step-function pattern as illustrated in Figure 3c. We observe that while the performance gap between batch sizes 96 and 128 is a relatively minor 5%, this gap sig-

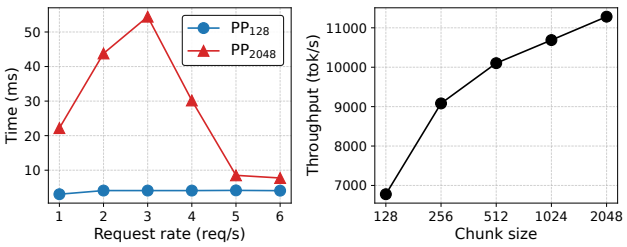


Figure 4: Pipeline bubbles and prefill token throughput according to load and chunk size (when serving Qwen2.5-32B with the Azure `Code` workload on 4 NVIDIA A100 GPUs)

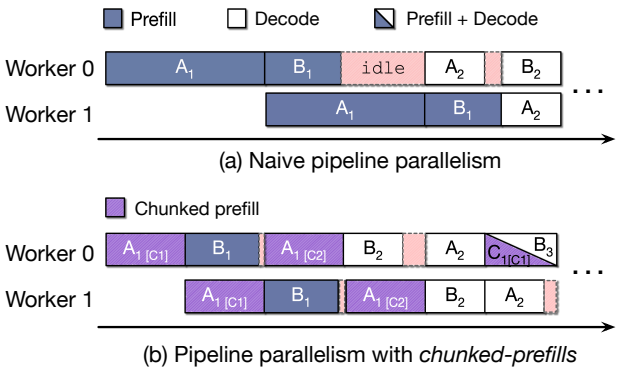


Figure 5: Reducing pipeline stalls from chunked-prefill [2]

nificantly widens to 31% for batch sizes between 128 and 160. Due to the step-function nature of linear operations [20], a batch size increase from 128 to 129 can abruptly trigger this about 31% idle time, resulting in substantial *D-D imbalance*. To mitigate such inefficiencies, microbatches for decode-only phases should be configured in multiples of 128 to align with the preferred efficiency of the underlying compute kernels.

3.2 Our Approaches

3.2.1 Chunked Prefills in Pipeline Parallelism

To reduce the pipeline bubbles caused by long prefills, a simple and straightforward approach is to leverage *chunked-prefill*, which splits the long input sequences into small, fixed-size chunks [2]. Since prefill computation is proportional to the input prompt length, limiting the prompt with a fixed chunk size helps reduce the latency imbalance across pipeline workers. Figure 4a shows the pipeline bubble time by increasing the request rate for two different chunk sizes. The large chunk size of 2,048 is determined by the compute capability of the NVIDIA A100 GPU used in this analysis. At low request rates, the chunk size of 128 results in fewer bubbles than 2,048. With a chunk size of 2,048, under low to moderate load (1~3 req/s), a few long prefill microbatches dominate the pipeline, increasing pipeline bubbles. Once the request rate reaches around 4 req/s, the pipeline is more consistently filled with incoming prefill requests, which reduces imbalance across stages. For request rates above 5 req/s, pipeline bubbles rarely occur for either chunk size because each pipeline stage remains fully utilized by the increased number of tokens.

While smaller chunk sizes help minimize pipeline stalls across the load, they sacrifice the computational throughput because the number of tokens to be processed per iteration is reduced. Figure 4b presents token throughput according to the chunk sizes. The chunk size of 128 shows significantly lower throughput than that of 2,048, leading to underutilization of GPU resources. This exacerbates TTFT performance, especially when the incoming request rate is high, as more requests are queued. Larger chunks are more effective for achieving high computational throughput.

To balance the trade-off between minimizing pipeline bubbles and maximizing GPU utilization, the chunk size must be dynamically adjusted according to system load.

3.2.2 Balancing Microbatches

Although adjusting the chunk size in prefill phases can mitigate pipeline bubbles caused by long prefills, it cannot make pipeline stages balanced in decode-heavy phases where most batches consist of only decode phases, as shown in Figure 2 ❸. In such decode-heavy phases, the computational load of each microbatch is primarily determined by the number of requests, since each request processes only a single input token. However, in existing LLM serving systems, once a request is assigned to a microbatch, it remains fixed for its lifetime [14, 37]. This static scheduling scheme magnifies imbalance and sustains pipeline stalls across multiple iterations. Note that tensor parallelism does not decompose input batches into microbatches; instead, it evenly partitions the computation of each batch across GPUs.

To address this issue, a straightforward approach is to dynamically rebalance the number of requests in microbatches across pipeline stages during iteration-level scheduling.

4 Retrofitting Pipeline Parallelism

4.1 Dynamic Chunked Prefill

Our design goal is to adaptively select a chunk size that minimizes pipeline bubbles while maintaining sufficient GPU utilization to process the current request load. Figure 5 shows that dividing the prompt (A_1) into two smaller chunks ($A_{1[C1]}$ and $A_{1[C2]}$) allows earlier stages in the pipeline to make progress, thus reducing idle time.

When determining the optimal chunk size, we treat the key performance metrics of online LLM serving, $TTFT_{SLO}$ and $TPOT_{SLO}$, as constraints. These two SLOs jointly define the feasible range of chunk sizes. Based on these constraints, our

Algorithm 1 Greedy chunk size adjustment

```
D_p ← Pipeline depth
C ← Set of possible chunk sizes (e.g., {128, 256, ...})
c ← Current chunk size
δ ← Chunk adjustment step size (e.g., 128)
KV_free ← Available KV cache memory ratio (∈ [0,1])
θ_mem ← KV memory watermark (∈ [0,1])
α, β ← Safe and critical boundaries for TTFT_slack (∈ [0,1])
γ ← Boundary for TPOT_slack (∈ [0,1])

for every D_p scheduling iterations do
    # ❶ Retrieve system metrics from the previous iterations
    retrieve_metrics (TTFT_slack, TPOT_slack, KV_free)
    # ❷ Check the available KV space
    if KV_free < θ_mem then
        c ← min{x ∈ C | x > N_run_req}
    # ❸ TPOT is violated or there is sufficient TTFT_slack
    else if TPOT_slack < 0 or TTFT_slack > α · TTFT_SLO then
        c ← c − δ
    # ❹ TTFT_slack is tight and there is sufficient TPOT_slack
    else if TTFT_slack < β · TTFT_SLO and TPOT_slack > γ · TPOT_SLO then
        c ← c + δ
```

primary objective is to minimize pipeline bubbles by selecting the smallest chunk size that can accommodate the current request load. As demand increases, the system adaptively enlarges the chunk size to improve utilization, but strictly limits this increase based on $TPOT_{SLO}$. When a TPOT violation is anticipated, our scheduler promptly reduces the chunk size to preserve responsiveness.

Meanwhile, $TTFT_{SLO}$ serves as an indicator of the current request load. When the waiting queue grows due to high request arrival rates, incoming requests are likely to violate $TTFT_{SLO}$. To mitigate this, the system increases the chunk size to increase throughput and drain the queue more quickly. However, this adjustment is allowed only as long as it does not violate $TPOT_{SLO}$. When TTFT is low, it indicates light load. In this case, the system actively reduces the chunk size. This adjustment can effectively mitigate the pipeline bubbles. This sophisticated scheduling improves resource efficiency by minimizing the idle time caused by pipeline imbalances. Based on these common optimization principles, we explore two approaches in the following sections: a greedy approach and a predictive approach.

4.1.1 Greedy approach

We first present a simple yet effective greedy method (Algorithm 1) that operates as a feedback control loop to dynamically adjust the chunk size. We maintain a discrete set of candidate chunk sizes $\mathcal{C}$ and keep the current chunk size $c \in \mathcal{C}$, initialized to the minimum chunk size (e.g., 128). We update c every D_p scheduling iterations, since a chunk size change takes effect only after a microbatch has traversed all D_p stages. We use three control parameters (α, β, and γ) that define tolerance boundaries for $TTFT_{SLO}$ and $TPOT_{SLO}$. An additional parameter, θ_{mem}, prevents out-of-memory (OOM) problems during inference.

For every D_p scheduling iterations, ❶ we compute $TTFT_{slack}$ and $TPOT_{slack}$, the remaining latency budgets for their respective SLOs, based on measured TTFT and TPOT from the previous iterations. Also, we collect the available KV memory ratio KV_{free}. Then, ❷ we perform a memory sanity check. When $KV_{free} < \theta_{mem}$, keeping a large chunk size can cause some stages to fail KV allocations. In this case, we immediately reset c to the smallest candidate that is larger than the current number of running requests. This decision prioritizes quickly draining in-flight requests, freeing KV memory, and minimizing additional bubbles.

If memory is sufficient, we then adjust the chunk size based on latency slack. ❸ When the observed $TPOT_{slack}$ indicates that the $TPOT_{SLO}$ is (or is about to be) violated, we decrease the chunk size. This reduces the pipeline bubbles caused by the prefill phase and helps ensure that the system continues to satisfy the target $TPOT_{SLO}$ for in-flight requests. In addition, when $TTFT_{slack}$ is sufficiently large, implying that the request load is low relative to the current chunk size and the system is far from the $TTFT_{SLO}$ violation. In this case, we also decrease the chunk size. This decision mitigates the *P-P* and *P-D* imbalances and significantly improves TPOT. Under light load, this adjustment has little impact on throughput because the system is not compute-saturated.

❹ When $TTFT_{slack}$ is insufficient while TPOT remains within a comfortable range, we increase the chunk size to improve throughput. This enlarges the amount of prefill computation per iteration, improving compute utilization and reducing TTFT, while preventing the violation of $TPOT_{SLO}$.

Limitation: Although our greedy approach performs reasonably well, it has structural limitations in finding the optimal chunk size. First, due to its step-wise adjustment mechanism based on reactive feedback, the greedy approach suffers from inherent latency in adaptation. It requires multiple iterations to converge on the optimal chunk size, so under rapidly changing workloads, the scheduler is constantly chasing a moving target. This inevitably leads to prolonged periods of operation in a suboptimal state.

Second, as the greedy approach relies only on latency statistics observed in previous iterations, $TTFT_{slack}$ and $TPOT_{slack}$, it leads to suboptimal chunk sizes under pipeline parallelism, where feedback is delayed by pipeline depth and skewed by larger chunks in other pipeline stages. These limitations motivate us to explore a predictive approach that estimates the iteration latency of LLM inference from the current system state, enabling accurate slack estimation.

4.1.2 Predictive approach

Overview: To overcome the limitations of the reactive greedy approach, we propose a predictive chunk size adjustment technique. This method leverages a lightweight latency prediction

model to evaluate candidate chunk sizes and select one that minimizes pipeline bubbles, thereby improving goodput.

To make this decision, given the current system state (e.g., the waiting queue and in-flight batches), our scheduler uses the prediction model to estimate the iteration latency and throughput of each candidate chunk size. The iteration latency is then multiplied by the pipeline depth to estimate the end-to-end latency under pipeline parallelism. The scheduler then establishes an upper bound on the chunk candidates by comparing the predicted latency against $TPOT_{SLO}$. Next, it derives a lower bound by comparing the predicted throughput against the minimum throughput required to satisfy $TTFT_{SLO}$. Finally, the scheduler selects the smallest feasible chunk size within these bounds to minimize pipeline bubbles.

Latency prediction model: Our latency prediction model consists of two components: offline profiling for linear layers and online adaptation for attention layers. As shown in Figure 3, iteration latency is dominated by linear layers, whose latency scales with the total number of batched tokens (i.e., chunk size), enabling accurate latency prediction through offline profiling. In chunked-prefill, the selected chunk size determines the number of tokens processed per iteration, making the input size to linear layers largely deterministic. On the other hand, attention layers account for a smaller and dynamic portion of the overall latency tied to the request composition. Since LLM inference is inherently auto-regressive, the composition of requests in the batch evolves gradually across iterations. This gradual evolution enables real-time calibration of dynamic attention latency, allowing the model to track system dynamics without introducing prediction instability.

We model the per-iteration latency, T_{iter}, as the sum of an offline component, $T_{offline}$, and an online component, T_{online}. Thus, $T_{iter} = T_{offline} + T_{online}$. The offline component models the latency of linear layers as a function of the total number of batched tokens, N_{tokens}, along with a constant runtime cost: $T_{offline} = \alpha_0 \cdot N_{tokens} + \alpha_1$. In the offline phase, we measure iteration latency across discrete batched tokens (N_{tokens}) from 128 to 2048 to compute the latency for linear layers.

For the online component, T_{online}, we model the latency of attention layers, which varies with the context lengths:

$$T_{online} = \beta_0 \cdot \sum(L_{past} \cdot L_{pre}) + \beta_1 \cdot \sum(L_{pre}^2) + \beta_2 \cdot \sum L_{dec} + \varepsilon$$

In chunked-prefill, the attention latency in each iteration depends on three quantities: the accumulated past context from prior chunks (denoted as L_{past}), the current prefill chunk (L_{pre}), and the context lengths of decode requests (L_{dec}). These contribute through three terms, prefill attention over past context ($L_{past} \cdot L_{pre}$), self-attention within the current chunk (L_{pre}^2), and decode attention (L_{dec}), with coefficients, β_0, β_1, and β_2, respectively. The residual ε accounts for unmodeled system noise.

To calibrate the parameters (β_0, β_1, and β_2), we compute the difference between the predicted and the measured iteration latency at each inference step. Based on this prediction error, we adaptively update the model parameters using the Recursive Least Squares (RLS) algorithm [8]. RLS is well-suited for our setting in two ways. First, unlike standard gradient descent, whose updates are sensitive to the scale of input features, RLS uses an internal covariance matrix to adapt to feature-wise scaling, mitigating the severe scale disparities between terms (e.g., L_{pre}^2 vs. L_{dec}). Second, a forgetting factor prioritizes recent observations, allowing the model to quickly adapt to dynamic changes such as prefill-decode interference.

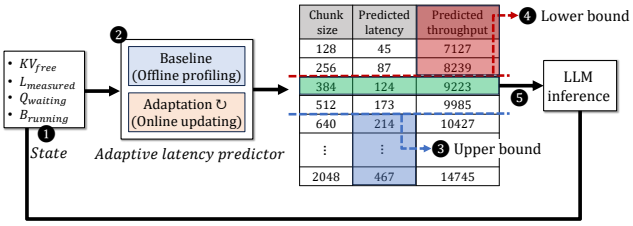


Figure 6: Workflow of predictive dynamic chunked-prefill

Online inference: Figure 6 presents how our scheduler leverages the prediction model to find the optimal chunk size. For each scheduling iteration, ❶ the scheduler first retrieves the current system state, specifically the available KV memory (KV_{free}), the measured iteration latency from the previous iteration ($L_{measured}$), the waiting queue ($Q_{waiting}$), and the running batch ($B_{running}$). It then performs a memory safety check, which is identical to that in the greedy algorithm. ❷ It updates the predictor using $L_{measured}$ and attention features (e.g., prefix and context lengths) measured in the previous iteration. The updated predictor then estimates the iteration latency for each candidate chunk size using features derived from $Q_{waiting}$ and $B_{running}$. These estimates are used to derive the predicted end-to-end latency, by scaling the iteration latency with the pipeline depth, and the expected throughput. ❸ Candidates whose predicted latency exceeds $TPOT_{SLO}$ are discarded, leaving a feasible search pool.

To determine the lower bound on the chunk size, ❹ the scheduler derives the minimum prefill throughput required to satisfy $TTFT_{SLO}$. It uses the incoming request rate and token distribution over a sliding window to compute an arrival-matching throughput, ensuring that newly arriving requests can be served in real time without growing the waiting queue. It further accounts for waiting requests exceeding a freshness threshold, calculated by dividing their total prompt length by their remaining time budget (i.e., $TTFT_{SLO}$ minus the average waiting time). The arrival-matching throughput and this additional component are summed to obtain the minimum required throughput. The lower bound is defined as the smallest chunk size whose predicted throughput meets this target.

Finally, ❺ it determines the optimal chunk size. If candidates satisfying both SLOs exist, the smallest one is chosen to minimize pipeline bubbles while satisfying the GPU uti-

lization required by the current request load. If no candidate meets the conditions, the system is considered overloaded. In this case, to prioritize draining the waiting queue, it selects the largest chunk that satisfies $TPOT_{SLO}$ to maximize throughput.

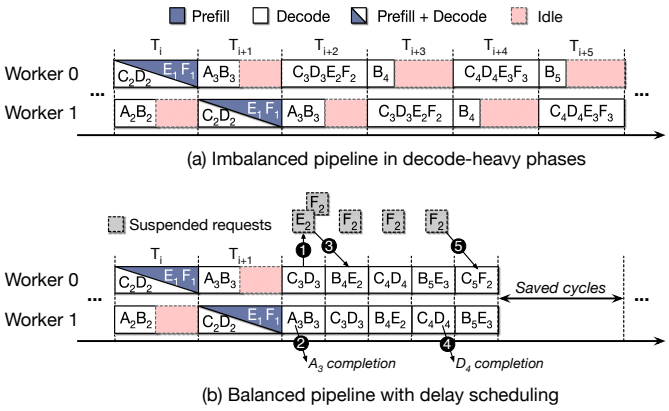
4.2 Dynamic Rebalancing

Although our dynamic chunked-prefill effectively reduces pipeline bubbles caused by prefill requests, it is ineffective when the running batch contains no prefill requests (i.e., decode-only iterations). To further reduce pipeline bubbles, we introduce a runtime scheme, called *delay scheduling*, that dynamically rebalances workloads across pipeline stages. We observe that in decode-only iterations, the number of requests is the dominant factor in computational imbalance. This arises because each request consumes only one input token at a time, resulting in nearly uniform per-request computation in the linear layers. As shown in Figure 3b, the decoding latency highly depends on the number of requests, and the linear layer takes most of the time.

When balancing microbatch sizes across pipeline stages, our delay scheduling policy aligns batch sizes with hardware-efficient batch size boundaries (e.g., 128, 256, 384). As shown in Figure 3c, processing 129 tokens costs almost the same as processing 256 tokens, so batch sizes that are slightly misaligned with these boundaries achieve little additional throughput while exacerbating *D–D imbalance*. To mitigate this, we apply a simple threshold-based preemption policy. If the number of requests in a microbatch exceeds a boundary but the overflow is within a configured threshold, we temporarily preempt the excess requests, reducing the batch size to the nearest boundary. If the overflow is larger than the threshold, we instead rebalance by aligning the batch size to the average request count across all microbatches, resuming previously suspended requests as needed to fill the gap.

The selection of which requests to preempt adapts to the current system state. When $TPOT_{slack}$ is insufficient, we preferentially preempt older requests, which can tolerate additional latency more readily. This preserves the TPOT of newer requests and reduces pipeline bubbles, lowering overall TPOT. When $TPOT_{slack}$ is ample, however, the primary risk becomes KV-cache exhaustion and $TTFT_{SLO}$ violations. In this case, we preempt newer requests instead, trading off $TPOT_{slack}$ to let older requests finish earlier and free KV memory, thereby improving goodput.

Figure 7 highlights the effectiveness of our delay scheduling approach. Although some decode-phase requests are deferred to subsequent iterations, this strategy reduces pipeline stalls, thereby improving overall completion time. At each scheduling iteration, we examine the load distribution across pipeline stages. In naive pipeline parallelism (Figure 7a), `Worker 0` executes a microbatch containing $C_3D_3E_2F_2$ while `Worker 1` processes a smaller microbatch A_3B_3 at T_{i+2}, leading to significant imbalance and idle periods. In contrast,

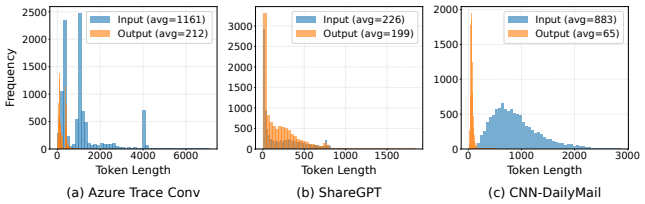


(a) Imbalanced pipeline in decode-heavy phases

(b) Balanced pipeline with delay scheduling

Figure 7: Reducing pipeline stalls with delay scheduling

with delay scheduling (Figure 7b), ❶ request E and F are temporarily suspended to rebalance the pipeline. Although their decoding phases E_2 and F_2 are postponed, this reduces pipeline stalls across stages. ❷ Once request A completes at T_{i+2}, ❸ the suspended decode-phase E_2 can be resumed, forming a microbatch B_4E_2. Similarly, ❹ after request D finishes, ❺ the suspended decode-phase F_2 is resumed. As the pipeline stages become more balanced, the overall latency is significantly improved.



(a) Azure Trace Conv (b) ShareGPT (c) CNN-DailyMail

Figure 8: Input and output length distributions

5 Evaluation

5.1 Experimental Setup

Environment: We run all experiments on a server with four NVIDIA A100 (40GB) GPUs, one 128-core AMD EPYC 9754 CPU (2.25 GHz), and 768 GB of RAM. Each GPU is connected to the host system via PCIe 4.0. We use PyTorch v2.5.1 [3], CUDA 12.4 [18], and FlashInfer [34]. We implement our proposed techniques as well as the baseline pipeline parallelism with chunked prefill on top of SGLang 0.4.1 [37].

Models and workloads: We select two popular LLM models, Qwen2.5 32B and 14B [32]. All the models are BF-16 formatted versions. To mimic real-world workloads, we use three real-world traces, including Azure Conversation [22], ShareGPT [24], and CNN [9], and a synthetic trace for evaluating decode-heavy workloads by reversing the input and output distribution of the CNN trace [9]. Figure 8 shows the input and output length distributions of the traces.

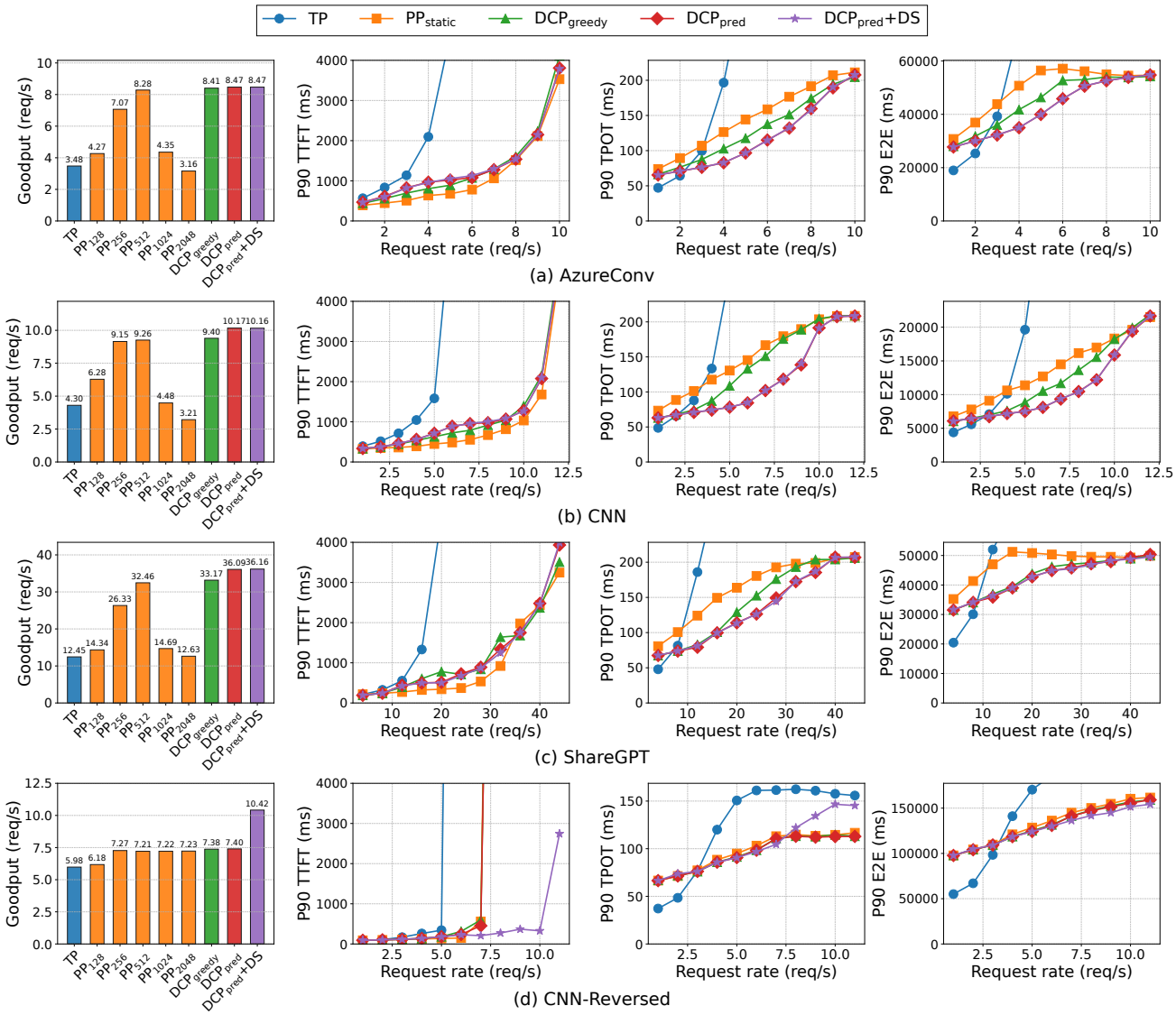


Figure 9: Goodput, TTFT, TPOT, and E2E latency for Qwen2.5-32B

Comparisons: Our baselines are tensor parallelism (**TP**) and pipeline parallelism (**PP**). We explore our three design options: dynamic chunked-prefill (**DCP**) with the greedy approach (**DCP$_{\text{greedy}}$**) and the predictive approach (**DCP$_{\text{pred}}$**), and the integration of the predictive DCP and delay scheduling (**DCP$_{\text{pred}}$+DS**).

Metrics: We use goodput as our primary evaluation metric, defined as the highest request rate at which both the 90th-percentile (P90) TTFT and TPOT satisfy their SLOs. In accordance with MLPerf guidelines [1], we set the latency budgets for TTFT and TPOT to 2,000ms and 200ms, respectively. Our primary objective is to maximize this goodput. Once we achieve comparable or sufficiently high goodput levels, we then shift our focus to latency minimization. We closely examine the P90 values for TTFT, TPOT, and end-to-end (E2E) latency to compare the latency behavior across designs.

5.2 Experimental Results

5.2.1 Goodput and latency

Figure 9 shows the goodput, TTFT, TPOT, and E2E latency as we increase the request rate for Qwen2.5-32B. To emulate realistic user traffic, we generate request arrivals according to a Poisson process, a standard model for interactive user-facing services. For tensor parallelism, we use the default chunk size of 2,048 [2, 31]. For the TTFT, TPOT, and E2E latency plots, we report only PP$_{\text{static}}$, configured with the chunk size that achieves the highest goodput for each workload. The `AzureConv`, `CNN`, and `ShareGPT` workloads exhibit prefill-heavy behavior, leading to *Prefill-Prefill* and *Prefill-Decode pipeline imbalances*, whereas `CNN-Reversed` is decode-heavy and thus suffers from *Decode–Decode pipeline imbalance*. On `AzureConv`, `CNN`, and `ShareGPT`,

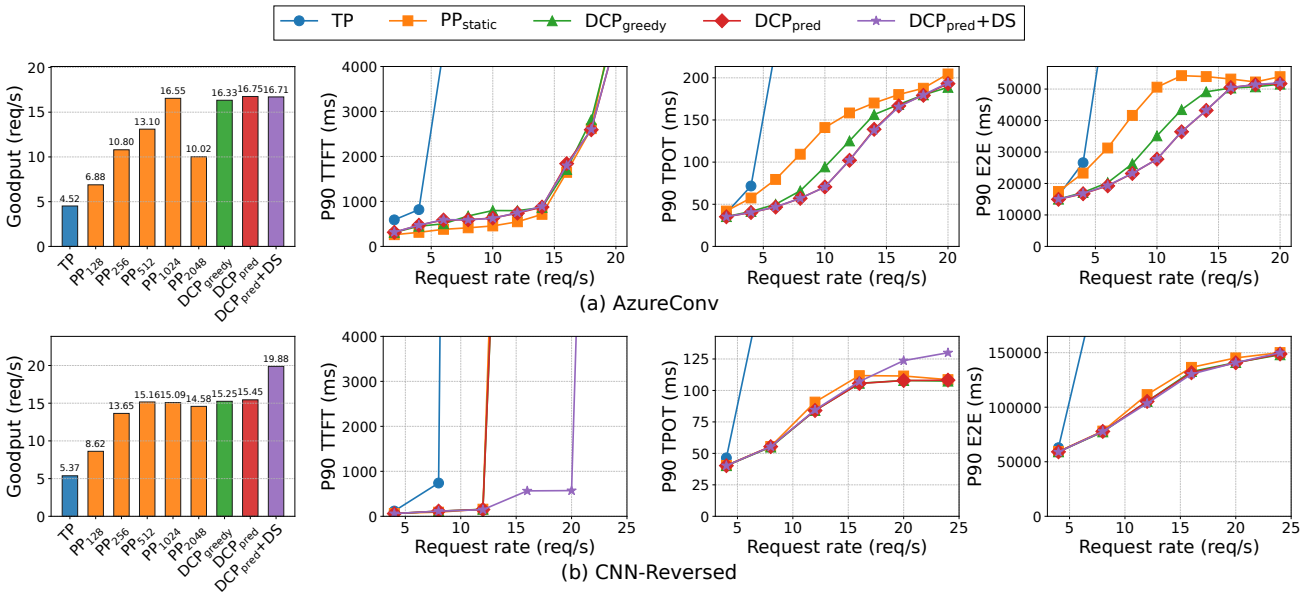


Figure 10: Goodput, TTFT, TPOT, and E2E latency for Qwen2.5-14B

a chunk size of 512 achieves the highest goodput, while `CNN-Reversed` reaches its peak goodput with a chunk size of 256, indicating that the optimal chunk size is workload-dependent. Also, this indicates that once we select the appropriate chunk size, pipeline parallelism delivers higher goodput than tensor parallelism because it substantially reduces pipeline imbalance.

However, PP_{static} cannot adjust the chunk size at runtime as the load (request rate) changes, and thus continues to suffer from pipeline bubbles. Our two dynamic chunked prefill (DCP) techniques address this limitation. Compared to PP_{static}, both DCP variants slightly increase TTFT but have a negligible impact on goodput, while substantially improving TPOT and E2E latency by reducing pipeline bubbles.

For `AzureConv`, DCP_{greedy} reduces TPOT and E2E latency by up to 19% and 18%, respectively, whereas DCP_{pred} achieves 35% and 31% reductions. The reactive nature of DCP_{greedy} fundamentally limits how aggressively it can exploit favorable operating points. Consequently, it nearly matches DCP_{pred} in goodput, but it shows a smaller latency reduction. We observe similar trends in the other prefill-heavy workloads. Compared to PP_{static}, on `CNN`, DCP_{greedy} reduces TPOT and E2E latency by 27% and 28%, respectively, whereas DCP_{pred} achieves larger reductions of 42% and 36%. On `ShareGPT`, DCP_{pred} achieves reductions of 36% and 24%. These results indicate that predictive chunk size selection provides more consistent reductions across workloads than the greedy one.

We also evaluate our delay scheduling (DS), which rebalances work across microbatches (DCP_{pred}+DS). For prefill-heavy workloads, *D–D pipeline imbalance* rarely occurs, so delay scheduling is rarely triggered. On the other hand, for decode-heavy workloads where input sequences are short but outputs are long, delay scheduling is frequently activated and further reduces E2E latency on top of DCP_{pred}. For `CNN-Reversed`, our delay scheduling can drastically reduce the TTFT latency, leading to higher goodput compared to our dynamic chunked prefill and the two baselines. Once requests are processed quickly, we can promptly reclaim their KV memory, which in turn reduces TTFT.

We also evaluate our techniques on Qwen2.5-14B. Due to space constraints, Figure 10 reports results for two representative workloads, `AzureConv` (prefill-heavy) and `CNN-Reversed` (decode-heavy). Overall, we observe similar performance trends to those of Qwen2.5-32B. For `AzureConv`, a chunk size of 1,024 achieves the highest goodput. Compared to Qwen2.5-32B, the desired chunk size is changed. Our dynamic chunked prefill techniques achieve the highest goodput and lower TPOT and E2E latency. Compared to PP_{static}, our greedy and predictive approaches reduce the TPOT latency by up to 40% and 50%. Although our delay scheduling (DCP_{pred}+DS) has little effect on the prefill-heavy workload, it reduces TTFT and median E2E latency for the decode-heavy workload. While the 32B model shows a P90 E2E latency reduction at high request rates, the 14B model shows no visible reduction. Nonetheless, the improved median latency shows the effectiveness of our delay scheduling.

5.2.2 Performance analysis

We analyze how effectively our techniques reduce pipeline stalls and imbalance across stages. Figure 11 shows the cumulative distributions of per-iteration latency for (a) `AzureConv`, (b) `CNN`, and (c) `CNN-Reversed`. For each workload, we de-

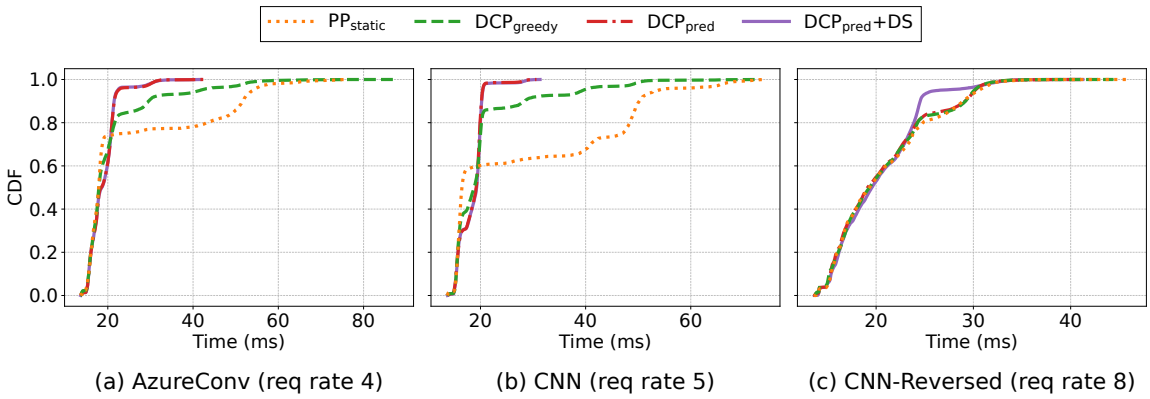


Figure 11: CDF of iteration latency for (a) AzureConv, (b) CNN, and (c) CNN-Reversed

rive the distributions from the runs in Figure 9 at average loads of roughly 4, 5, and 8 requests per second, respectively.

For the `AzureConv` workload, the 90th-percentile iteration latency drops from 51.68ms with PP_{static} to 29.56ms with DCP_{greedy} and further to 21.53ms with DCP_{pred}. At the 99th percentile, DCP_{pred} achieves 30.94ms compared to 65.34ms with PP_{static}, yielding a 2.11× improvement that significantly reduces tail latency. This suggests that DCP_{pred} more efficiently concentrates iterations within this lower bound and thereby minimizes pipeline bubbles. For `AzureConv`, *decode-decode pipeline imbalance* is negligible. Therefore, DCP_{pred}+DS is rarely triggered and results in behavior similar to DCP_{pred}.

The `CNN` workload exhibits a similar performance behavior to `AzureConv`. DCP_{pred} substantially shortens per-iteration latency, yielding a noticeable performance improvement over PP_{static}. Although DCP_{pred} shows slightly higher latency than PP_{static} below the 50th percentile, starting around the 60th percentile DCP_{pred} becomes faster and its advantage grows increasingly large toward the tail.

For the `CNN-Reversed` workload, the short input sequences and long outputs leave much less room for control via our DCP schemes. As a result, PP_{static}, DCP_{greedy}, and DCP_{pred} exhibit only marginal differences. On the other hand, with delay scheduling the 90th-percentile iteration latency drops to 24.53ms, whereas PP_{static} shows 28.94ms. This behavior confirms that decode-heavy workloads structurally suffer less from pipeline bubbles than prefill-heavy ones, but also that our DCP techniques alone cannot fully eliminate imbalance.

5.2.3 Goodput under varying SLOs

We evaluate how our proposed techniques perform under increasingly tight SLO budgets. Figure 12 shows goodput for Qwen2.5-32B and Qwen2.5-14B under `AzureConv` and `CNN-Reversed` workloads with varying SLO scales. On the x-axis, SLO Scale of 1.0 corresponds to 2,000 ms for TTFT and 200 ms for TPOT. The y-axis represents goodput, the maximum stable request rate satisfying these SLOs. Both TP and PP baselines use a default chunk size of 2048. PP generally outperforms TP, except for the 32B model under `AzureConv`, where the high compute costs dominate, diminishing PP's advantage in reducing communication overhead. We therefore focus on comparing our proposed approaches against the PP baseline. Since our two dynamic chunked prefill variants, DCP_{greedy} and DCP_{pred}, achieve similar goodput across SLO scales, we refer to them collectively as DCP unless otherwise specified. At the strictest SLO scale of 0.2 for the 32B model, even the smallest possible chunk violates the TPOT constraint, reducing the goodput of all PP-based systems to zero.

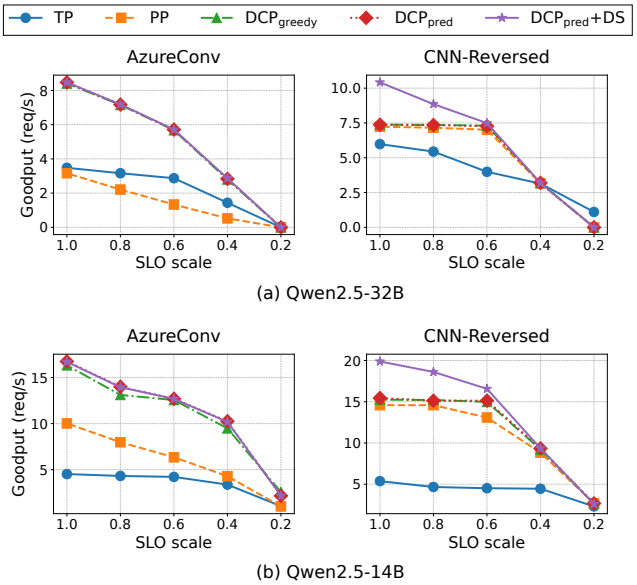


Figure 12: Goodput comparison of Qwen2.5 32B and 14B under various SLO constraints.

DCP effectively mitigates the *P-P* and *P-D* pipeline imbalances inherent in the prefill-heavy `AzureConv` workload, significantly outperforming the PP baseline. Notably, the rela-

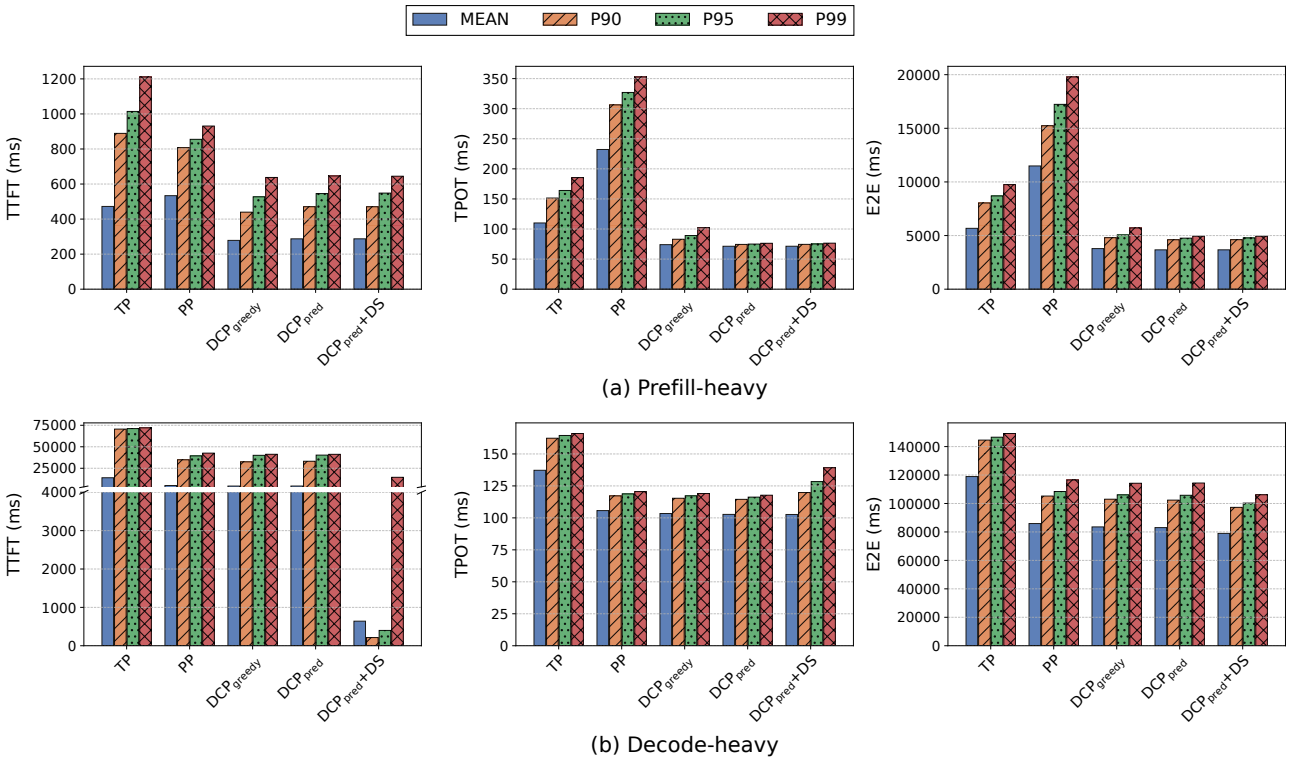


Figure 13: TTFT, TPOT, and E2E latency for (a) prefill-heavy and (b) decode-heavy synthetic workloads. Input and output lengths are uniformly sampled from [512, 1024] and [32, 64] tokens for (a), and from [32, 64] and [512, 1024] tokens for (b).

tive improvement of DCP over PP grows as SLO constraints tighten. As the SLO scale decreases from 1.0 to 0.4, the improvement increases from 2.7× to 5.4× for the 32B model, and from 1.7× to 2.4× for the 14B model. This highlights DCP's robustness under increasingly tight latency constraints. Meanwhile, delay scheduling (DS) provides no additional benefit for this prefill-heavy workload.

By contrast, in the decode-heavy `CNN-Reversed` workload, the goodput of the PP baseline remains nearly constant as the SLO scale decreases from 1.0 to 0.6. With minimal prefill tokens, violations are primarily driven by KV cache exhaustion that triggers abrupt TTFT spikes, making goodput insensitive to moderate SLO tightening. While DCP provides limited benefits due to the small prefill fraction, DS plays a crucial role by resolving the *D-D* pipeline imbalance inherent to decoding operations, yielding up to a 1.4× goodput improvement for both the 14B and 32B models over the PP baseline. However, as the SLO tightens below 0.6, DS effectiveness drops sharply since the system hits the TPOT limit before accumulating enough requests to meet its activation threshold.

5.2.4 Synthetic prefill-heavy and decode-heavy cases

To further analyze the performance benefits and limitations of each scheme, we construct synthetic workloads by uniformly sampling sequence lengths. For a prefill-heavy workload, input and output lengths are sampled from 512 to 1024 and 32 to 64 tokens, respectively; the ranges are swapped for a decode-heavy workload. Figure 13 presents TTFT, TPOT, and end-to-end latency when serving Qwen2.5-32B under prefill-heavy and decode-heavy workloads at 5 and 14 requests per second, respectively. These rates represent the point at which the PP baseline begins to violate the SLO constraints.

In the prefill-heavy workload, both TP and PP baselines achieve similar TTFT, but PP suffers much higher TPOT and end-to-end latency than TP due to severe pipeline imbalances. Meanwhile, our two DCP methods show lower latency than TP and PP across all metrics by reducing these imbalances. This indicates that mitigating *P-P* imbalance is more critical than using larger chunks to improve GPU utilization, even if smaller chunks reduce batching efficiency. The two DCP variants achieve nearly identical mean latency, while DCP_{pred} further reduces P99 TPOT and end-to-end latency by 25% and 13% over $\text{DCP}_{\text{greedy}}$, respectively, showing its advantage in reducing tail latency. Delay scheduling has little effect in this prefill-heavy case because the dominant bottleneck comes from the prefill-induced pipeline bubbles.

In the decode-heavy workload, PP achieves lower latency than TP across all metrics, reducing mean TTFT, TPOT, and end-to-end latency by 2.82×, 1.30×, and 1.38×, respectively. Both DCP variants perform similarly to PP, since *P–P* and *P–D* imbalances rarely occur in decode-heavy workloads.

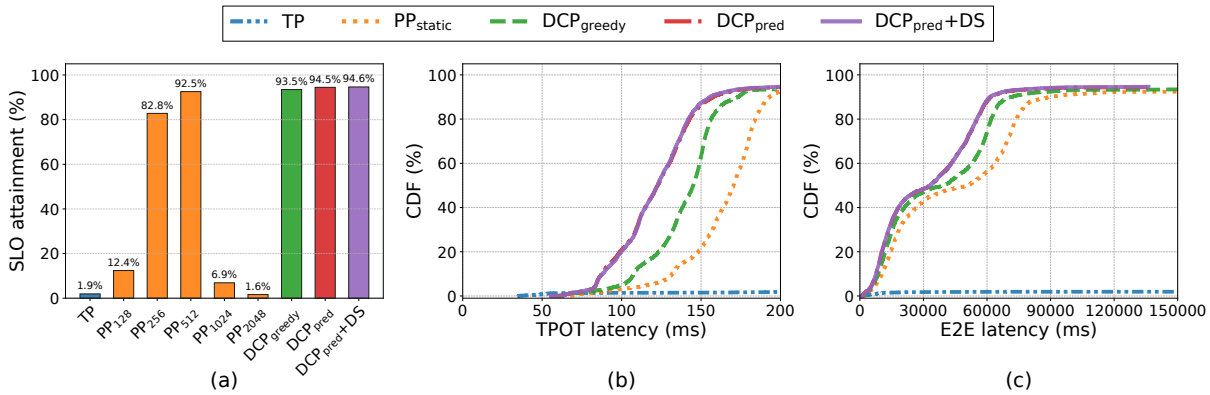


Figure 14: Performance of real-trace using Azure Conversation [22]: (a) SLO attainment, (b) TPOT latency CDF (conditional on $TTFT_{SLO}$), and (c) E2E latency CDF (conditional on all SLOs) for Qwen2.5-32B

Meanwhile, delay scheduling (DS) effectively reduces *D-D* imbalance, lowering TTFT and end-to-end latency. Specifically, DS significantly reduces the TTFT latency compared to DCP and PP, resulting in lower end-to-end latency. Once *D-D* imbalance becomes severe, DS preempts short-context requests in the batch to expedite the completion of long-context requests, reclaiming their KV memory earlier and allowing more pending requests to be served. As a result, DS reduces end-to-end latency by 4,062ms on average and 8,088ms at P99 compared to DCP_{pred}. However, these preempted requests incur higher TPOT, making P99 TPOT 15% worse than DCP_{pred}.

5.2.5 Performance of real-world trace

We evaluate our techniques using a real-world workload trace from `AzureConv`. Specifically, we replay the first 15 minutes of the trace. As the original load is already high, we scale the request arrival rate down to 0.9× its original value, so that the SLO attainment of $\text{PP}_{\text{static}}$ is approximately 90%.

Figure 14 presents (a) SLO attainment, (b) TPOT CDF for satisfying TTFT_{SLO}, and (c) E2E latency CDF for requests satisfying both TTFT_{SLO} and TPOT_{SLO}. TP achieves only 1.9% SLO attainment due to its substantial communication overhead. With the default chunk size of 2,048 tokens, pipeline parallelism (PP_{2048}) performs even worse, with SLO attainment of just 1.6%. On the other hand, $\text{PP}_{\text{static}}$, whose chunk size is selected via an offline search, achieves 92.5% SLO attainment. The workload is prefill-heavy, so our delay scheduling scheme is not activated.

While the overall SLO attainment of $\text{PP}_{\text{static}}$ and DCP is similar, DCP more effectively eliminates pipeline bubbles and shifts a larger fraction of requests into lower-latency regions. Among the two designs, $\text{DCP}_{\text{greedy}}$ yields slightly worse performance than DCP_{pred} due to the inherent limitations of its heuristic. In real-world traces, high load appears in short bursts rather than being sustained. Thus, choosing a fixed chunk size based on peak load (as in PP_{static}) is inefficient. When the load drops, the large chunk size underutilizes pipeline stages and reintroduces pipeline bubbles during normal low-load periods. If the TPOT_{SLO} is set to 150 ms, $\text{DCP}_{\text{greedy}}$ and DCP_{pred} achieve 3.06× and 4.14× higher attainment than PP_{static}, respectively. In terms of E2E latency, $\text{DCP}_{\text{greedy}}$ and DCP_{pred} complete 1.3× and 1.6× as many requests within 60,000ms as $\text{PP}_{\text{static}}$.

Model	MAPE (%)	Pred. time (ms)	Size (KB)
ALP (Baseline)	15.85	0.000136	0.363
+ Prefix Attn	12.21	0.001027	0.408
+ Prefill Attn	9.69	0.001570	0.474
+ Dec Attn	3.48	0.002101	0.552
+ ε (Residual)	3.26	0.002875	0.643
Linear Regression	4.26	0.000594	0.187
Decision Tree Ensemble	2.74	0.038217	1330
DNN (2-layer, BN)	2.56	0.046407	570

Table 1: Comparison of latency prediction models

5.2.6 Analysis of latency prediction models

To understand the effectiveness of the proposed adaptive latency prediction (ALP) model, we perform an in-depth analysis of its prediction accuracy and computational cost. As described in Section 4.1.2, ALP accounts for latency variations in attention layers, which are inherently dynamic and cannot be predicted by offline profiling alone. We first evaluate how each component of ALP contributes to prediction accuracy, starting from an offline profiling method that models only the latency spent in linear layers and progressively adding the prefix, prefill, and decode attention terms, as well as residual terms. We measure Mean Absolute Percentage Error (MAPE) on a real-world `AzureConv` workload.

Table 1 presents the effectiveness of each ALP component. Since the offline profiling method does not account for the latency spent in attention layers or runtime noise, it shows

a relatively large prediction error. As we add each attention component, the model better predicts the latency contributed by attention layers, and MAPE gradually decreases. Finally, adding the residual ε term compensates for runtime noise that the attention terms cannot explain, achieving the lowest prediction error among the variants. Although prediction time and model size grow as the model becomes more sophisticated, both remain negligible.

We also evaluate three ML-based techniques for iteration latency prediction: linear regression using NumPy's `linalg.lstsq` [7], a decision tree ensemble using LightGBM [13], and a simple DNN with two linear layers and batch normalization. We train linear regression with the same features used in ALP's online adaptation phase, and additionally provide request-level statistics, such as the number of requests per microbatch and the average and maximum number of tokens per request, to the decision tree and DNN models. Although the decision tree ensemble and DNN achieve slightly lower MAPE than ALP, the improvement is marginal: ALP's MAPE is only 0.70 percentage points higher than that of the DNN model. This small accuracy gap does not lead to meaningful end-to-end performance improvements in TTFT or TPOT. In addition, these ML-based techniques require an offline training phase, whereas ALP performs lightweight online adaptation without pre-training. Therefore, ALP provides a more practical choice for LLM serving systems.

6 Related Work

There have been significant efforts to develop efficient LLM serving frameworks on modern GPUs. ORCA was an early and influential system that introduced iteration-level scheduling and selective batching techniques to effectively utilize GPU cores, achieving substantial performance gains [35]. More recently, chunked-prefill [2, 10] was proposed to enhance iteration-level scheduling under prefill-heavy workloads by splitting long prompts into smaller chunks, thereby alleviating TPOT latency. These techniques have been widely adopted in production frameworks such as vLLM [14], SGLang [37], TensorRT-LLM [28], and Dynamo [19]. In addition to the scheduling enhancement, managing KV-cache on GPU, CPU, and storage has also been widely studied to avoid the recomputation cost of attention [5, 12, 33, 36].

For multi-GPU environments, prior research has examined a range of parallelism strategies, including tensor parallelism [25, 27], pipeline parallelism [11, 17], and expert parallelism [15, 26] in MoE models to maximize hardware efficiency. In practice, tensor parallelism has become the de facto choice for single-node, multi-GPU LLM serving due to its predictable load balancing and low per-token latency, especially when supported by high-bandwidth interconnects. AlpaServe [16] extended this by combining statistical multiplexing with model parallelism to dynamically allocate GPU resources according to workload characteristics, improving throughput in multi-tenant settings. ExeGPT [21] further refined resource allocation with constraint-aware scheduling, jointly optimizing for latency, throughput, and GPU utilization in LLM inference.

However, despite these efforts, the interference between the prefill and decode phases causes significant bubbles in pipeline parallelism. To mitigate this, phase-disaggregated serving [22, 23, 38] isolated these phases on separate GPUs. While this design reduces interference, it introduces nontrivial overheads, including model weight duplication and high-bandwidth requirements for KV cache transfer. To avoid such overheads, unified serving with chunked-prefill [2] remains a practical choice, yet static chunking yields non-negligible pipeline bubbles. While recent dynamic chunking approaches [6] prioritized SLO compliance, they overlooked the structural constraints of pipeline parallelism, leaving pipeline bubbles unresolved. This highlights the need for a pipeline-specific dynamic chunking strategy that effectively eliminates bubbles under varying workloads.

7 Conclusions

This paper investigated the inefficiencies of pipeline parallelism in LLM serving and proposed scheduling techniques to mitigate pipeline imbalances. Our greedy and predictive chunk size adjustment schemes effectively improved pipeline performance in prefill-heavy workloads, while a dynamic rebalancing scheme further alleviated imbalance in decode-heavy workloads. We evaluated both individual techniques and integrated designs on the Qwen2.5 32B and 14B models using several real-world workloads. Experimental results demonstrated that pipeline parallelism with the proposed techniques outperforms tensor parallelism in terms of goodput for single-node, multi-GPU environments.

Acknowledgments

We thank the anonymous reviewers for providing helpful feedback and suggestions to improve our work. This work was supported by the Institute of Information & communications Technology Planning & Evaluation (IITP) grant (RS-2024-00396013) and the Electronics and Telecommunications Research Institute (ETRI) grant [26ZS1100, Development of Large-Scale Parallel Computing Technology for Generative AI]. Both grants were funded by the Korean government (MSIT). This work was supported in part by SK hynix Inc. through the provision of GPU cloud resources for evaluating the performance of existing parallelism methods. Jeongseob Ahn is the corresponding author.

References

[1] Mlperf inference: Datacenter benchmark suite results, 2024. https://mlcommons.org/benchmarks/inference-datacenter/.

[2] Amey Agrawal, Nitin Kedia, Ashish Panwar, Jayashree Mohan, Nipun Kwatra, Bhargav Gulavani, Alexey Tumanov, and Ramachandran Ramjee. Taming Throughput-Latency tradeoff in LLM inference with Sarathi-Serve. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2024.

[3] Jason Ansel, Edward Yang, Horace He, Natalia Gimelshein, Animesh Jain, Michael Voznesensky, Bin Bao, Peter Bell, David Berard, Evgeni Burovski, Geeta Chauhan, Anjali Chourdia, Will Constable, Alban Desmaison, Zachary DeVito, Elias Ellison, Will Feng, Jiong Gong, Michael Gschwind, Brian Hirsh, Sherlock Huang, Kshiteej Kalambarkar, Laurent Kirsch, Michael Lazos, Mario Lezcano, Yanbo Liang, Jason Liang, Yinghai Lu, CK Luk, Bert Maher, Yunjie Pan, Christian Puhrsch, Matthias Reso, Mark Saroufim, Marcos Yukio Siraichi, Helen Suk, Michael Suo, Phil Tillet, Eikan Wang, Xiaodong Wang, William Wen, Shunting Zhang, Xu Zhao, Keren Zhou, Richard Zou, Ajit Mathews, Gregory Chanan, Peng Wu, and Soumith Chintala. PyTorch 2: Faster Machine Learning Through Dynamic Python Bytecode Transformation and Graph Compilation. In *29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2024.

[4] Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Amy Yang, Angela Fan, et al. The llama 3 herd of models. *arXiv e-prints*, pages arXiv–2407, 2024.

[5] Bin Gao, Zhuomin He, Puru Sharma, Qingxuan Kang, Djordje Jevdjic, Junbo Deng, Xingkun Yang, Zhou Yu, and Pengfei Zuo. Cost-Efficient large language model serving for multi-turn conversations with CachedAttention. In *2024 USENIX Annual Technical Conference (ATC)*, 2024.

[6] Kanishk Goel, Jayashree Mohan, Nipun Kwatra, Ravi Shreyas Anupindi, and Ramachandran Ramjee. Niyama: Breaking the silos of llm inference serving. *arXiv preprint arXiv:2503.22562*, 2025.

[7] Charles R Harris, K Jarrod Millman, Stéfan J van der Walt, Ralf Gommers, Pauli Virtanen, David Cournapeau, Eric Wieser, Julian Taylor, Sebastian Berg, Nathaniel J Smith, Robert Kern, Matti Picus, Stephan Hoyer, Marten H van Kerkwijk, Matthew Brett, Allan Haldane, Jaime Fernández del Río, Mark Wiebe, Pearu Peterson, Pierre Gérard-Marchant, Kevin Sheppard, Tyler Reddy, Warren Weckesser, Hameer Abbasi, Christoph Gohlke, and Travis E Oliphant. Array programming with NumPy. *Nature*, 585(7825):357–362, 2020.

[8] S.S. Haykin. *Adaptive Filter Theory*. Pearson, 2014.

[9] Karl Moritz Hermann, Tomas Kocisky, Edward Grefenstette, Lasse Espeholt, Will Kay, Mustafa Suleyman, and Phil Blunsom. Teaching machines to read and comprehend. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2015.

[10] Connor Holmes, Masahiro Tanaka, Michael Wyatt, Ammar Ahmad Awan, Jeff Rasley, Samyam Rajbhandari, Reza Yazdani Aminabadi, Heyang Qin, Arash Bakhtiari, Lev Kurilenko, et al. Deepspeed-fastgen: High-throughput text generation for llms via mii and deepspeed-inference. *arXiv preprint arXiv:2401.08671*, 2024.

[11] Yanping Huang, Youlong Cheng, Ankur Bapna, Orhan Firat, Mia Xu Chen, Dehao Chen, HyoukJoong Lee, Jiquan Ngiam, Quoc V. Le, Yonghui Wu, and Zhifeng Chen. Gpipe: efficient training of giant neural networks using pipeline parallelism. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2019.

[12] Jinwoo Jeong and Jeongseob Ahn. Accelerating llm serving for multi-turn dialogues with efficient resource management. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2025.

[13] Guolin Ke, Qi Meng, Thomas Finley, Taifeng Wang, Wei Chen, Weidong Ma, Qiwei Ye, and Tie-Yan Liu. Lightgbm: A highly efficient gradient boosting decision tree. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2017.

[14] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph Gonzalez, Hao Zhang, and Ion Stoica. Efficient memory management for large language model serving with pagedattention. In *Proceedings of the 29th Symposium on Operating Systems Principles (SOSP)*, 2023.

[15] Dmitry Lepikhin, HyoukJoong Lee, Yuanzhong Xu, Dehao Chen, Orhan Firat, Yanping Huang, Maxim Krikun, Noam Shazeer, and Zhifeng Chen. Gshard: Scaling giant models with conditional computation and automatic sharding. *arXiv preprint arXiv:2006.16668*, 2020.

[16] Zhuohan Li, Lianmin Zheng, Yinmin Zhong, Vincent Liu, Ying Sheng, Xin Jin, Yanping Huang, Zhifeng Chen, Hao Zhang, Joseph E. Gonzalez, and Ion Stoica. AlpaServe: Statistical multiplexing with model parallelism for deep learning serving. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2023.

[17] Deepak Narayanan, Aaron Harlap, Amar Phanishayee, Vivek Seshadri, Nikhil R. Devanur, Gregory R. Ganger, Phillip B. Gibbons, and Matei Zaharia. Pipedream: generalized pipeline parallelism for dnn training. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP)*, 2019.

[18] NVIDIA. CUDA, release: 12.4, 2024. https://developer.nvidia.com/cuda-toolkit.

[19] NVIDIA AI Dynamo Team. Nvidia dynamo: A datacenter scale distributed inference serving framework. https://github.com/ai-dynamo/dynamo, 2024. GitHub repository.

[20] NVIDIA Corporation. Deep learning performance documentation: Matrix multiplication background. https://docs.nvidia.com/deeplearning/performance/dl-performance-matrix-multiplication/index.html#tile-quant, 2024. Accessed: 2025-12-12.

[21] Hyungjun Oh, Kihong Kim, Jaemin Kim, Sungkyun Kim, Junyeol Lee, Du-seong Chang, and Jiwon Seo. Exegpt: Constraint-aware resource scheduling for llm inference. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2024.

[22] P. Patel, E. Choukse, C. Zhang, A. Shah, I. Goiri, S. Maleki, and R. Bianchini. Splitwise: Efficient generative llm inference using phase splitting. In *Proceedings of the 51st Annual International Symposium on Computer Architecture (ISCA)*, 2024.

[23] Ruoyu Qin, Zheming Li, Weiran He, Jialei Cui, Heyi Tang, Feng Ren, Teng Ma, Shangming Cai, Yineng Zhang, Mingxing Zhang, Yongwei Wu, Weimin Zheng, and Xinran Xu. Mooncake: A kvcache-centric disaggregated architecture for llm serving. *ACM Transactions on Storage*, 2024.

[24] ShareGPT, 2023. https://huggingface.co/datasets/anon8231489123/ShareGPT_Vicuna_unfiltered/tree/main.

[25] Noam Shazeer, Youlong Cheng, Niki Parmar, Dustin Tran, Ashish Vaswani, Penporn Koanantakool, Peter Hawkins, HyoukJoong Lee, Mingsheng Hong, Cliff Young, Ryan Sepassi, and Blake Hechtman. Mesh-tensorflow: Deep learning for supercomputers. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2018.

[26] Noam Shazeer, Azalia Mirhoseini, Krzysztof Maziarz, Andy Davis, Quoc Le, Geoffrey Hinton, and Jeff Dean. Outrageously large neural networks: The sparsely-gated mixture-of-experts layer. *arXiv preprint arXiv:1701.06538*, 2017.

[27] Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. Megatron-lm: Training multi-billion parameter language models using model parallelism, 2020.

[28] Neal Vaidya, Nick Comly, Joe DeLaere, Ankit Patel, and Fred Oh. NVIDIA TensorRT-LLM Supercharges Large Language Model Inference on NVIDIA H100 GPUs, September 2023. https://developer.nvidia.com/blog/nvidia-tensorrt-llm-supercharges-large-language-model-inference-on-nvidia-h100-gpus.

[29] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Ł ukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2017.

[30] vLLM Docs. Distributed inference and serving, 2025. https://docs.vllm.ai/en/stable/serving/distributed_serving.html.

[31] vLLM Team. vLLM engine argument utilities (version v0.12.0). https://github.com/vllm-project/vllm/blob/v0.12.0/vllm/engine/arg_utils.py, 2025. Describes the `max_num_batched_tokens` parameter, which dictates the default chunk size.

[32] An Yang, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, et al. Qwen2.5 technical report. *arXiv preprint arXiv:2412.15115*, 2024.

[33] Jiayi Yao, Hanchen Li, Yuhan Liu, Siddhant Ray, Yihua Cheng, Qizheng Zhang, Kuntai Du, Shan Lu, and Junchen Jiang. Cacheblend: Fast large language model serving for rag with cached knowledge fusion. In *Proceedings of the Twentieth European Conference on Computer Systems (EuroSys)*, 2025.

[34] Zihao Ye, Lequn Chen, Ruihang Lai, Yilong Zhao, Size Zheng, Junru Shao, Bohan Hou, Hongyi Jin, Yi fei Zuo, Liangsheng Yin, Tianqi Chen, and Luis Ceze. Accelerating self-attentions for llm serving with flashinfer, February 2024.

[35] Gyeong-In Yu, Joo Seong Jeong, Geon-Woo Kim, Soojeong Kim, and Byung-Gon Chun. Orca: A distributed serving system for Transformer-Based generative models. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2022.

[36] Lingfan Yu, Jinkun Lin, and Jinyang Li. Stateful large language model serving with pensieve. In *Proceedings of the Twentieth European Conference on Computer Systems (EuroSys)*, 2025.

[37] Lianmin Zheng, Liangsheng Yin, Zhiqiang Xie, Chuyue Sun, Jeff Huang, Cody Hao Yu, Shiyi Cao, Chris tos Kozyrakis, Ion Stoica, Joseph E. Gonzalez, Clark Barrett, and Ying Sheng. Sglang: Efficient execution of structured language model programs. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2024.

[38] Yinmin Zhong, Shengyu Liu, Junda Chen, Jianbo Hu, Yibo Zhu, Xuanzhe Liu, Xin Jin, and Hao Zhang. DistServe: Disaggregating prefill and decoding for goodput-optimized large language model serving. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2024.

ARTIFACT EVALUATED
USENIX
AVAILABLE

OpenTela: Unifying Decentralized Computing Resources for Heterogeneous LLM Serving

(Operational Systems)

Xiaozhe Yao
ETH Zurich

Youhe Jiang
University of Cambridge

Ilia Badanin
EPFL

Qinghao Hu
MIT

Robert Matthew Smith
ETH AI Center

Binhang Yuan
HKUST

Imanol Schlag
ETH AI Center

Eiko Yoneki
University of Cambridge

Ana Klimovic
ETH Zurich

Abstract

Large language models (LLMs) are becoming critical to a variety of public services, motivating sovereign AI initiatives that seek to serve models on infrastructure they control. Yet much of this infrastructure is built as HPC clusters optimized for batch jobs rather than interactive, always-on inference services. Existing LLM serving engines efficiently execute requests on GPUs, but rely on an external control plane (e.g., Kubernetes in cloud environments) for service discovery, routing, health monitoring, and load balancing. In HPC environments, these primitives are often unavailable: resources are managed by schedulers such as Slurm, allocations are transient, compute nodes may be unreachable from outside the cluster, and GPU capacity is fragmented across heterogeneous clusters and administrative domains.

We present OpenTela, a user-space orchestration overlay that turns existing fragmented HPC clusters into a unified, cross-institutional serving platform. OpenTela provides fault-tolerant service discovery via a CRDT-based gossip network, a unified API over heterogeneous cluster managers and serving engines, and a heterogeneity-aware scheduler, all in user-space without root privileges or cluster reconfiguration. OpenTela has been deployed for over 22 months, serving 13 million requests and 15 billion tokens across 142 models to over 1000 researchers across multiple institutions. We open-source the system and release an anonymized production trace to facilitate further research into real-world LLM serving workloads, and provide a replicable blueprint for other sovereign AI initiatives to harness their own federated GPU infrastructure.

1 Introduction

As large language models (LLMs) are increasingly being integrated into critical digital infrastructure for public services such as administration [48], healthcare [7, 61], and education [14], governments and organizations need more control over how these models are developed and served. While commercial AI providers like OpenAI and Anthropic provide convenient access to state-of-the-art models, organizations using these models lack insight into how the models were trained — and hence the biases they contain. Users also have no control over the availability of these models and how they evolve over time (e.g., OpenAI can suddenly decide to stop serving a particular version of a model). Hence, many governments and organizations are investing heavily in *sovereign AI infrastructure*: compute clusters, data pipelines, model development research, and operational platforms that they can control, inspect, and govern [4, 24, 49, 64].

This shift has triggered large-scale public investment in AI-oriented supercomputing infrastructure. For example, Switzerland has invested in the Alps supercomputer, a geo-distributed cluster of over 10,000 NVIDIA Grace Hopper GPUs for open science [44]. Japan's ABCI 3.0 provides 6,128 NVIDIA H200 GPUs for AI [49]. The European Commission plans to build 19 AI Factories and up to five AI Gigafactories backed by a €20 billion investment [16]. The UK government has committed up to £2 billion to deliver new AI supercomputers [28]. Canada, Singapore, South Korea, the United States and many others are also investing in building and broadening access to AI compute [24–27].

Making sovereign AI useful in practice requires more than training models on this infrastructure. These models must also be served efficiently, reliably, and continuously to users. Model serving requires substantial compute, often exceeding the cycles spent on training [17, 67].

Yet the infrastructure being built for sovereign AI often consists of HPC clusters, which are designed for throughput-oriented batch jobs rather than interactive, always-on services like LLM inference. Model serving requires load balancing, dynamic service discovery, persistent endpoints, fault-tolerant request routing, and elastic scaling. LLM serving engines such as vLLM [38] and SGLang [75] assume that an external orchestration layer (e.g., Kubernetes) will provide the surrounding service functionality. However, in HPC environments, resources are usually managed by batch schedulers, such as Slurm [73], that give users a set of nodes for a bounded time window (e.g., up to 12 hours), with no persistent endpoints across job re-invocations or control plane to route external traffic to model replicas. Replacing Slurm with Kubernetes

is non-trivial as HPC workloads depend on features such as topology awareness, MPI integration, and established operational policies. Running a separate Kubernetes partition avoids disrupting existing workflows, but rigidly partitions scarce GPUs between batch and serving workloads, which could leave idle capacity in one partition while there is demand for the other [1, 65].

Furthermore, to deal with highly dynamic GPU availability within each HPC cluster (e.g., see Figure 1) and the bursty load patterns of LLM inference, it is often beneficial to pool resources across clusters to smooth both effects: idle GPUs at one site can absorb demand from another, and aggregating requests across clusters can improve batching efficiency. However, cross-cluster serving is challenging as clusters are often administered separately and may run different schedulers, expose different network policies, contain heterogeneous GPUs, and differ in data sovereignty guarantees.

We present OpenTela, a thin user-space overlay that turns fragmented HPC resources into a unified LLM serving platform. OpenTela runs on top of existing cluster managers such as Slurm and Kubernetes, providing LLM serving primitives entirely in user space, such that it does not require root privileges, cluster reconfiguration, or changes to user workflows. OpenTela exposes a unified OpenAI-compatible API across heterogeneous serving engines, including vLLM and SGLang, that it can run as transient Slurm jobs, long-running Kubernetes services, and diverse GPU hardware. Its decentralized gossip network backed by a CRDT-based registry provides service discovery, health monitoring, and fault-tolerant routing without assuming a central coordinator. A heterogeneity-aware placement policy helps map models to available GPUs, while ensuring that requests are only routed to model instances that a user trusts.

We deploy OpenTela to provide a catalog of open-source and community-developed models to researchers in the Swiss AI Initiative, an initiative that trains and serves open foundation models, such as Apertus [30]. OpenTela unifies multiple geo-distributed clusters with different hardware and resource management stacks, including Slurm-managed and Kubernetes-managed resources. Researchers interact with a single API endpoint, while OpenTela handles authentication, usage tracking, routing, and node lifecycle management across the underlying infrastructure. OpenTela has been operational for over 22 months. During this period, it has served more than 13 million requests and 15 billion tokens across 142 models to over 1000 researchers across multiple institutions. We open-source OpenTela at https://github.com/eth-easl/opentela and release an anonymized production trace spanning July 2024 to October 2025 at https://huggingface.co/datasets/eth-easl/swissai-serving-trace. Unlike prior public traces that typically cover a small number of models or short time windows, our trace captures a long-running, multi-model, research-oriented LLM serving workload with both popular open-weight models and a long tail of community-developed models.

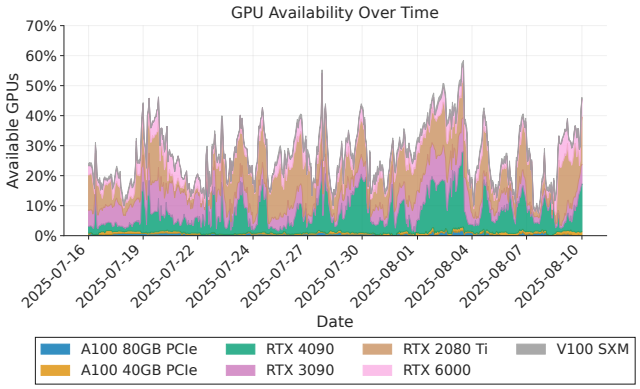


Figure 1: Percentage of idle GPUs from ETH HPC Cluster.

2 Background and Motivation

LLM serving engines are not serving platforms. Modern LLM serving engines, such as vLLM and SGLang, provide the execution layer for inference. They load model weights, manage KV caches, batch requests, schedule prefill and decode work, and expose an API endpoint for submitting prompts. These engines have made it substantially easier to run open-weight models efficiently on GPUs. However, an inference engine is not a complete serving platform. A shared LLM service must also provide a control plane around the engine: clients need a stable endpoint, the system must discover which replicas are currently serving which models, requests must be routed and load-balanced across replicas, failed replicas must be removed from the service, users must be authenticated, and operators need visibility into model availability and resource usage. In cloud deployments, these functions are often provided by Kubernetes or a proprietary orchestration system of a commercial provider. In contrast, most sovereign AI initiatives make GPU capacity available in HPC environments, where these features are lacking [41].

HPC schedulers. Slurm [73] is the most common workload manager across the top supercomputers [56]. Slurm's design reflects its origins in batch scientific computing workloads: it allocates nodes for a fixed wall-clock duration, prioritizes raw throughput over interactivity, and treats each job as a transient process with no expectation of persistent state. This works well for AI training jobs and batch data-processing pipelines. However, LLM serving requires clients to discover and route requests to model replicas. A Slurm job can launch an inference engine, but the resulting endpoint is tied to a particular allocation and compute node. When the job ends, is preempted, or is relaunched elsewhere, the endpoint changes or disappears. The scheduler does not provide a persistent service name, an external routing layer, a model registry, or a mechanism for replacing a failed replica with a healthy one.

Simply replacing Slurm with Kubernetes to leverage its orchestration primitives for LLM serving is often impractical

as many other jobs rely on Slurm features such as topology-aware scheduling and native MPI job launch [1, 65]. Another option is to partition HPC resources into a Kubernetes-managed component and a Slurm-managed component. However, this makes it more difficult to flexibly re-balance resources across the two clusters, which can lead to under-utilization and unnecessary queuing delays as the load on each cluster dynamically changes over time.

Federating HPC clusters. A single GPU cluster is often not enough to provide reliable and efficient LLM serving. GPU availability changes over time as resources are consumed by training jobs, released by completed jobs, and reserved for maintenance. Figure 1 shows this effect in an ETH GPU cluster: the fraction of idle GPUs varies over time and across GPU types. At the same time, our trace analysis in §6 shows that inference demand is bursty and model popularity shifts as users evaluate new models. Pooling resources across clusters can improve availability and utilization. Idle GPUs at one site can absorb demand from another, and aggregating requests across users and institutions can increase batching opportunities for popular models.

However, cross-cluster LLM serving introduces several challenges. The participating clusters may be administered by different organizations, run different schedulers, expose different network policies, contain different GPU types, and offer different data governance guarantees. Compute nodes may not be directly reachable from outside the cluster. Most importantly, there may be no single privileged coordinator that all sites are willing or able to operate. In the next section, we describe how to overcome these challenges.

3 OpenTela Design

To bridge the gap between HPC infrastructure and LLM serving workloads, we propose *OpenTela*: a user-space orchestration overlay that turns fragmented compute resources into a unified LLM serving platform. OpenTela *providers* contribute compute resources and deploy inference engine jobs while OpenTela *consumers* submit inference requests through an API frontend. OpenTela does not replace the underlying cluster manager in each cluster it federates. Instead, a provider launches OpenTela alongside an existing serving engine, such as vLLM or SGLang, on resources obtained from a local cluster manager (e.g., a Slurm allocation or a Kubernetes deployment). Once an OpenTela job is launched, the node running the job joins OpenTela 's peer-to-peer network, advertises its hardware and model metadata to other nodes, and becomes available to serve requests.

3.1 System Requirements

We design OpenTela around four key requirements:

R1: Non-invasive deployment. OpenTela must run with ordinary user permissions, on top of existing cluster managers, without requiring root access, scheduler modifications, or cluster reconfiguration. This allows researchers and operators to contribute resources from Slurm or Kubernetes environments without changing existing HPC workflows.

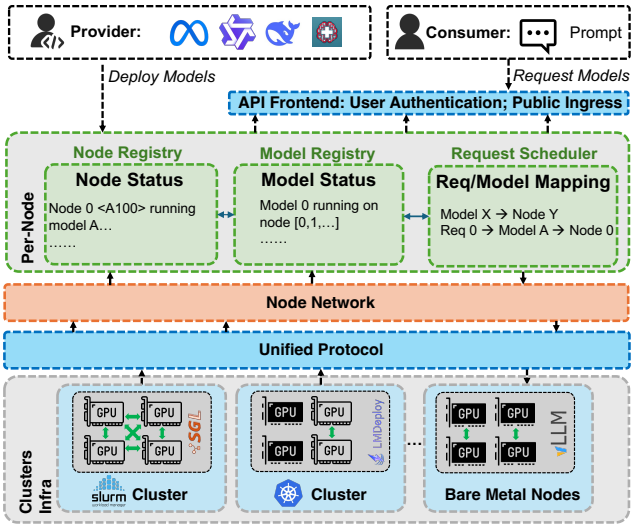


Figure 2: System architecture of OpenTela Serving.

R2: Service abstraction over transient resources. OpenTela must turn short-lived allocations and independently launched serving processes into a persistent service abstraction. It must provide service discovery, model and node registries, health monitoring, load balancing, and request routing even as nodes join, leave, fail, or are preempted.

R3: Federation without a centralized coordinator. OpenTela must operate across independently administered clusters with no shared privileged control plane. The system must maintain an eventually consistent view of available nodes and models, tolerate network partitions and node churn, and avoid making a central store a single point of failure.

R4: Heterogeneity- and trust-aware request routing. OpenTela must support diverse GPU types, serving engines, model variants, and deployment environments. It must also ensure that requests are routed only to providers authorized by the user or institution, preserving data-governance constraints even when other resources are available.

3.2 System Overview

Figure 2 shows the system architecture that OpenTela adopts to meet the above requirements.

Cluster infrastructure. OpenTela is designed to unify *geo-distributed* clusters that may consist of *heterogeneous* hardware. Each cluster may differ in hardware characteristics (GPU memory capacity and processing power, network bandwidth) and software environments (cluster manager, serving frameworks supported, CUDA versions).

Unified ML serving protocol. Most LLM serving engines expose an OpenAI-compatible API [50], which allows OpenTela to sit above them without modification. OpenTela adopts this interface as its foundation and extends it with infrastructure-level endpoints that the OpenAI API does not provide. Specifically, OpenTela exposes additional endpoints for querying real-time node status, model registry state, and

provider metadata, allowing operators and consumers to inspect the state of the serving platform directly without requiring access to individual nodes. For security, these endpoints are strictly read-only and expose the CRDT registry state without permitting external modification, preserving the integrity of the distributed states.

Node network. The node network is the transport layer that connects every OpenTela node into a single peer-to-peer fabric, regardless of which institution or cluster manager it belongs to. We build this layer with libp2p [39], which provides three key features our setting requires: peer addressing and discovery, encrypted peer-to-peer channels between any two peers, and connectivity establishment across NAT boundaries. Together, these make cross-institutional federation feasible: nodes on different clusters can locate one another, establish encrypted channels, and reach each other across private networks without any centralized broker. This is needed in our setting, as HPC nodes typically have only outbound connectivity and cannot accept inbound connections from outside the cluster. The node network establishes secure channels between two such nodes through hole punching when both can reach a common rendezvous peer, or through a circuit relay on a publicly reachable peer otherwise. Both modes use only the outbound connections that compute nodes are already permitted to make, and require no changes to institutional network policies. A new node joins the network by contacting one or more bootstrap peers. If all bootstrap peers are temporarily unreachable, the node retries with exponential backoff until connectivity is restored. Once connected, the node uses the Kademlia DHT [45] to discover other peers and exchanges state with them through a randomized gossip protocol layered on top of these peer-to-peer connections.

Registry. The registry is the state layer on top of the node network. It maintains a consistent global view of available nodes and models across institutional boundaries without a central coordinator. Our design of the registry considers that HPC nodes frequently join and leave as allocations expire or are preempted from scavenger queues, and that no central coordinator can be assumed to serialize state updates. OpenTela represents the registry as a Growth-only Map (G-Map) conflict-free Replicated Data Type (CRDT) [58], whose merge semantics are commutative, associative, and idempotent, guaranteeing convergence without coordination. Each entry in the map represents a node in the network: the key is a unique identifier assigned at node creation, and the value is a tuple containing the node's lifecycle state and its metadata payload. State updates propagate over the gossip protocol described above, and every node maintains a full local replica that it reads without further network round trips. To prevent unbounded registry growth as nodes join and leave over time, OpenTela applies a two-tier garbage collection policy: evicted nodes are first removed from the application-level view after a configurable retention window (24 hours by default), and CRDT-level tombstone markers are compacted periodically thereafter. The registry exposes two views derived from the same underlying state. The *Node Registry* tracks hardware availability and lifecycle state for each node, and the *Model Registry* maps models to the nodes currently serving them. Both views are queried locally on whichever node receives the query: every OpenTela node holds a full replica and answers from it without contacting any centralized catalog. In practice, consumers do not query the registry directly. The API frontend reads its local replica when routing each inference request, and the same replica backs the read-only HTTP endpoints described earlier, which expose registry contents for inspection by users and operators.

Service scheduler. While users can manually assign nodes to serve specific models, OpenTela also has a centralized *service scheduler* that can automatically optimize model placement to balance performance requirements with resource availability and cost efficiency. We describe the default policy in §3.5, but operators can substitute their own.

API frontend. OpenTela's API frontend runs as a regular OpenTela node that participates in the gossip network like any other but is reachable at a public IP address so that consumers can submit requests without running their own node. We call it an *ingress node* as it is the entry point through which external requests enter the gossip network. Alongside the standard OpenTela node functionality, the API frontend also runs a user-facing service that manages client connections and handles authentication, usage tracking, and the request-response lifecycle. The user-facing service and the ingress node can also run on separate hosts, as long as the former can forward authenticated requests to the latter. Because the ingress node holds a full local replica of the CRDT registry, it performs the candidate selection and routing described in §3.4 without contacting any external coordinator.

User interface. To ease access, OpenTela provides a web-based dashboard that allows users to browse the real-time catalog of actively served models alongside their hosting hardware, manage API keys, and validate model behavior through an interactive chat interface.

3.3 Trust Model

Allowing users to run arbitrary models and serve the requests of other users introduces security concerns. If any node can join the network and claim to serve any model, users have no guarantee that the node will execute inference correctly or keep the input data confidential. OpenTela addresses these concerns at two points in the serving path.

Membership admission control. Only signed OpenTela binaries can join the network. Each release is signed by maintainers, and unsigned or tampered binaries fail the join handshake and never appear in the CRDT registry.

Client-side filtering. When a user submits an inference request, they may specify a set of trusted provider IDs in the request metadata. The ingress node filters the candidate set from the CRDT registry to retain only nodes whose provider

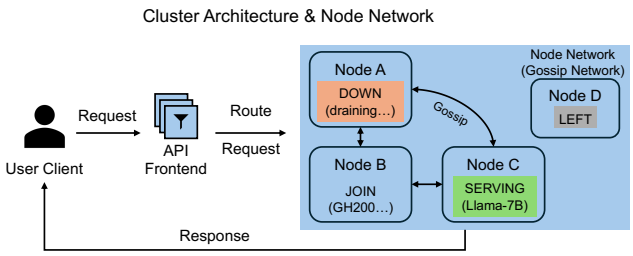


Figure 3: Workflow example of OpenTela.

IDs appear on this allowlist before applying load balancing. Requests are never dispatched to providers that the user has not explicitly allowed. This places trust management in the hands of the user, which is appropriate for a research environment where institutional relationships are known, but it requires users to maintain and update their allowlists correctly.

These mechanisms still have limitations. OpenTela attests binaries at the software level: peers verify a signature over the binary at load time, proving the file on disk matches what OpenTela maintainers signed. However, the provider, as the root user on the node they contribute, can replace the serving engine with a modified build that records every request or captures the prompts on the loopback interface between the OpenTela process and the serving engine. None of these produces externally observable artifacts that OpenTela can currently detect. Hardware-backed attestation through confidential computing would close this gap by running the serving code inside a hardware-encrypted enclave that the host root user cannot read, and by producing a remote attestation that cryptographically identifies the code executing inside the enclave. Clients could then require proof that their request was handled by a specific audited binary rather than trusting the provider's claim. We consider this a direction for future work.

3.4 Workflow Example

Life of a node. A node progresses through four strictly ordered states: `JOIN` $\prec$ `SERVING` $\prec$ `DOWN` $\prec$ `LEFT`. When a provider contributes a node to OpenTela, it initiates the `JOIN` process by contacting a known *bootstrap peer*, defined in the cluster configuration. The node registers its metadata (i.e., hardware specification) with the network. OpenTela uses a randomized gossip protocol for membership management: upon connection, the node exchanges state with the bootstrap peer, and this membership information propagates rapidly across the network. Within a few gossip rounds, the system converges and all peers update their local registries to include the new node.

After a node has joined the network, it begins the lifecycle of a model/service, entering the `SERVING` stage. The node deploys a specific model (either explicitly configured by the provider or dynamically assigned by the service scheduler). To announce availability, the node updates its local CRDT state with a service record containing the model identifier, the node's unique *session ID*, and optionally the *provider ID*. When the model is served with non-default configurations, such as lower precision or shorter context windows, it is the provider's responsibility to add this information to the model identifier. This metadata is automatically replicated to all peers, enabling other nodes to discover the new service endpoint and include it in their routing tables.

The `DOWN` stage is triggered if there is a local health check failure: when the per-node OpenTela process detects that the serving subprocess has terminated, it explicitly transitions to `DOWN` and propagates this state to all peers. `LEFT` marks permanent eviction and is triggered after the node exceeds the heartbeat grace period without recovery. The system applies a two-stage detection policy before escalating to `LEFT`: a node that fails liveness probes is first marked as *suspected* and excluded from the routing candidate set, but no irreversible state change is committed. Three independent probe paths must all fail before suspicion is declared, and a suspected node can self-refute by broadcasting a liveness update before the suspicion timeout expires. Only after this grace period elapses is the node permanently evicted. Since the merge function $\sqcup$ selects the entry with the highest lifecycle state, the network converges to the most recent fully-specified configuration for every node without any central reconciliation step. A node that recovers after eviction rejoins under a new session ID through the normal `JOIN` $\rightarrow$ `SERVING` lifecycle.

Life of an inference request. Figure 3 shows the life of a request. To schedule the request, the ingress node queries its local CRDT replica to identify candidate nodes that are serving the requested model. OpenTela routes requests based on the unique *session ID* and *provider ID* associated with each node. The system filters the initial candidate set against the user's specified *trusted providers*. Any node whose provider ID is not in the user's allowlist is pruned from the candidates pool. From the remaining authorized candidates, a specific session ID is selected via a load-balancing strategy, and the request is dispatched to that node's API endpoint. By default, the ingress node selects a session ID uniformly at random from the authorized candidates, with round robin and weighted random selection (e.g., weighted by reported GPU class) as alternative options. This decentralized approach ensures high availability and fault tolerance, as the system can dynamically route traffic across all compliant nodes, continuing to function even if individual nodes go offline.

Request-level fault tolerance. OpenTela performs best-effort retries by re-routing failed requests to an alternative node, up to a configurable maximum retry count. For non-streaming requests, this retry is transparent to the client. Streaming requests are handled differently: if a node fails or the network connection is interrupted after the stream has begun, OpenTela cannot resume generation on a different node easily, because LLM inference is non-deterministic and the partial response cannot be replayed from an arbitrary continuation point using the standard serving API. Clients issuing streaming requests must therefore implement their own retry

logic at the application level.

Scheduling policy customization. OpenTela decouples scheduling policy from request routing mechanics, allowing operators to integrate custom scheduling policies into the system. At the request-scheduling stage, a policy implements a three-method interface: `Pick` selects a worker from the trust-filtered candidate set for the current request, `OnRequestStart` and `OnRequestEnd` bracket each request so that load-aware policies can track in-flight work. Richer signals, such as the hardware of each candidate node, can be read from the local CRDT state. At the fleet level, OpenTela provides the building blocks for an external control loop: every node exposes the full status of the fleet, including the nodes in the network, their available hardware, the models they serve, and which nodes are idle. A controller can therefore poll any node, identify spare capacity, and decide which models to deploy where. For example, consider integrating Helix [46], which serves LLMs over heterogeneous GPUs and networks by routing each request along a path derived from a max-flow optimization. Its two components naturally map to OpenTela's design: the `Pick` method implements the max-flow-based selection logic, while the model-placement decision becomes a fleet-level controller that launches and terminates serving engines.

3.5 Model Placement and Scheduling

OpenTela allows users to manually specify which nodes in the cluster will serve which models. However, these decisions can be difficult for users to make. Users will often request the highest-end GPUs for their jobs to maximize performance, but these devices are usually also the most expensive and their availability is often scarce due to high demand. OpenTela therefore provides a scheduler that users can invoke when there are idle resources to fill and models waiting to be served. The scheduler decides which models should run on which nodes, and is a pluggable component so operators can swap in their own policy if needed. The scheduler runs whenever there is a model to place across more than one available hardware configuration, typically as new resources become available or as observed workload statistics drift.

In our current deployment, OpenTela does not reclaim compute nodes from running models during periods of low load, and does not repurpose them toward more popular models, because doing so makes the system less predictable for users. Resource reclamation happens only through the underlying cluster manager (e.g., Slurm timeouts or preemption). When unallocated resources do appear, OpenTela uses them to serve popular models, but resources that are already allocated to a running model are left in place even when underutilized. OpenTela also does not currently change parallelism configurations or other execution settings once the model is running. Reconfiguration requires the user to stop the model instance and redeploy it with the new settings.

Default scheduling policy. The default scheduling policy in OpenTela minimizes mean end-to-end latency over a heterogeneous resource pool. We formulate it as a constrained optimization problem. Consider a heterogeneous cluster $\mathbf{D} = \{d_1, \ldots, d_N\}$, where d_n denotes the count of available GPUs of type n. We serve M distinct model types, indexed by m, with incoming workloads $\mathbf{W} = \{w_1, \ldots, w_M\}$. Each workload w_m is defined by a tuple $\langle \lambda, \mu_{in}, \mu_{out} \rangle$ representing arrival rate and input/output length distributions, which are collected and monitored by the API frontend. The scheduler determines two key decision variables:

1. **Allocation matrix ($\mathbf{A} \in \mathbb{R}^{M \times N}$):** Where $a_{m,n}$ is the number of type-n GPUs allocated to model m.
2. **Parallelism strategy ($\mathbf{P} = \{p_1, \ldots, p_M\}$):** Where p_m defines the data and tensor parallelism configuration for model m.

The optimization objective is formulated as:

$$\min_{\mathbf{A},\mathbf{P}} \quad L(\mathbf{W}, \mathbf{A}, \mathbf{P}) \tag{1}$$

where $L(\cdot)$ represents the mean end-to-end latency over the workload $\mathbf{W}$ under allocation $\mathbf{A}$ and parallelism strategy $\mathbf{P}$.

Solving Eq. 1 requires an efficient method to evaluate $L(\cdot)$ for any candidate configuration. The typical approach of profiling jobs on different hardware configurations is accurate but not scalable, particularly for clusters that may have highly heterogeneous hardware, such as the cases we consider. Instead, we use a *serving simulator* to analytically estimate performance.

The serving simulator takes model specifications (e.g., architecture, parameter count), hardware characteristics (e.g., GPU type) and request workloads (e.g., arrival times, input/output lengths) as inputs. The simulation operates in two stages, extending the analytical method of `LLM-Viewer` [74] from single-request analysis to multi-request serving scenarios. (**1**) We first enumerate all operators in the model, such as the $\mathbf{q}, \mathbf{k}, \mathbf{v}$ linear projections, self-attention computations, feed-forward layers, etc. For each operator, we estimate its execution time on the target hardware. In our current design, we estimate execution time using the roofline model [6, 71], taking into account the operator's computational intensity and the hardware's peak performance and memory bandwidth. This analytical model, though not as accurate as empirical profiling, provides a unique advantage in our environment as it does not require offline profiling, making it adaptable to new models, new hardware, and diverse serving environments, such as model or KV cache quantization, varying sequence lengths and batch sizes, etc. (**2**) In the second stage, we simulate the concurrent execution of the input workload. Based on the operator latencies derived in the first stage, the simulator models system behaviors including continuous batching, model parallelism, and scheduling policies. The simulator outputs key metrics like latency and throughput, which drive the

service scheduler's deployment decisions. However, we acknowledge that the accuracy of the simulator is limited by the assumptions of the roofline model and the complexity of real-world serving optimizations. Our simulator does not currently model many optimizations, such as chunked prefill [3], prefill and decode disaggregation [2], prefix caching, etc., which can significantly impact performance. We consider improving the accuracy of the serving simulator a key direction for future work.

These simulation results parameterize our objective function, allowing us to evaluate the specific trade-offs inherent in Eq. 1. The scheduler uses *Constraint Programming* (CP) [55] to solve the optimization problem, which excels at exploring discrete combinatorial spaces through constraint propagation, making it ideal for our resource allocation problem. We enforce three key constraints to ensure feasible deployments:

C1: Global Resource Capacity. The total allocation for each GPU type n cannot exceed availability:

$$\sum_{m=1}^{M} a_{m,n} \leq d_n, \quad \forall n. \tag{2}$$

C2: Memory Feasibility. To prune the search space, we ensure any assigned GPU type n has sufficient memory (mem_n) to host the model weights and KV cache (M_m):

$$a_{m,n} \cdot \text{mem}_n \geq M_m, \quad \forall (m,n) \text{ where } a_{m,n} > 0. \tag{3}$$

C3: Parallelism compatibility. For each model type m, the parallelism strategy p_m specifies a configuration $\langle \psi_{\text{DP}}^{(n)}, \psi_{\text{TP}}^{(n)} \rangle$ for each GPU type n where $a_{m,n} > 0$, representing the data parallelism [40] and tensor parallelism [47, 60] degrees, respectively. The configuration must satisfy:

$$a_{m,n} = \psi_{\text{DP}}^{(n)} \times \psi_{\text{TP}}^{(n)}, \quad \forall (m,n) \text{ where } a_{m,n} > 0, \tag{4}$$

ensuring that the $a_{m,n}$ GPUs of type n allocated to model m are exactly partitioned into $\psi_{\text{DP}}^{(n)}$ data-parallel replicas, each employing $\psi_{\text{TP}}^{(n)}$-way tensor parallelism.

4 Implementation

Our implementation of OpenTela consists of approximately 7K lines of Go. We choose Go because of its strong support for concurrency, cross-platform compatibility, and a rich ecosystem of libraries for building distributed systems. OpenTela uses `libp2p` [39], a modular network stack, to construct the peer-to-peer network that underpins the decentralized architecture. OpenTela provides a `Gin` [19]-based API endpoint that exposes RESTful endpoints for accessing the internal status and functionalities of the system. The complementary API frontend is implemented using `FastAPI` [18].

OpenTela is distributed as a pre-built command-line tool. To deploy a model, users simply encapsulate their existing inference commands (e.g., `vllm` or `sglang`) within the OpenTela CLI. For instance, a user can start a node by running: `otela start --process vllm serve [model_name]`. Alternatively, users can start a node without a pre-defined model by running `otela start`. In this mode, the node registers its available hardware resources with the network, allowing the service scheduler to assign models to the node later.

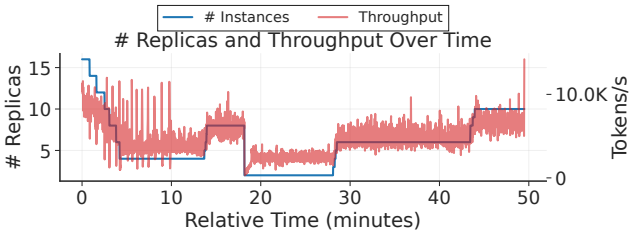


Figure 4: OpenTela's ability to recover from node failures over time.

5 System Evaluation

In this section, we evaluate the fault-tolerance and performance of OpenTela with benchmark workloads, while in §6, we will analyze the real workloads on the deployed system. Our evaluation aims to answer the following key questions:

Q1: Control plane overhead of OpenTela. How much latency does OpenTela's user-space control plane add to each request, and does this overhead remain stable as request load increases?

Q2: Elasticity and fault-tolerance. Can OpenTela maintain service continuity and scale throughput proportionally in a dynamic environment with node failures and additions?

Q3: Placement efficiency. How does our heterogeneity-aware placement strategy compare to standard heuristics in terms of aggregate cluster throughput?

Q4: Simulation accuracy. Does the serving simulator accurately capture relative hardware performance differences to guide scheduling decisions?

Experiment setup. We conduct our experiments on a heterogeneous environment composed of NVIDIA GPUs, ranging from datacenter-grade accelerators (GH200, H100, A100) to consumer-grade hardware (RTX 3090). This diversity allows us to evaluate OpenTela's ability to handle hardware heterogeneity. We use OpenTela to serve popular open-source LLMs with varying parameter sizes: `Qwen/Qwen3-1.7B`, `Llama-2-13b`, `CodeLlama-34b`, and `Llama-3.3-70B-Instruct`. Unless otherwise stated, request arrival times follow a Poisson process, with input and output token lengths sampled from Normal distributions derived from real-world traces collected from our deployment (see more details about traces in §6).

Control-plane overhead. We measure the per-request overhead introduced by OpenTela's control plane to quantify the cost of operating as a user-space overlay above the serving engines. We decompose each request's lifecycle into five control plane stages alongside the LLM inference performed by

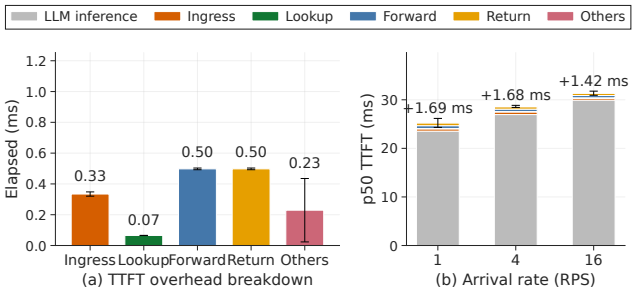


Figure 5: Control plane overhead of OpenTela. (a) Per-stage cost for a single request. (b) p50 TTFT decomposition at arrival rates of 1, 4, and 16 RPS with 256 input tokens.

the serving engine, and benchmark these components on a deployment serving `Qwen/Qwen3-1.7B` on 1 GH200 GPU, with Poisson arrival rates of 1, 4, and 16 RPS, and report the time spent in each stage.

Figure 5a reports the per-stage overhead breakdown for serving a 1.7B model. *Ingress*, which covers request reception and dispatch at the ingress node, takes 0.33 ms, and *Lookup*, the CRDT-based candidate selection at the ingress node, takes less than 0.1 ms, confirming that the local replica supports fast selection decisions without contacting a central coordinator. The dominant costs are *Forward* and *Return* at around 0.5 ms each, reflecting the network hops between the ingress and serving nodes. *Others* contributes a residual cost of less than 1 ms, which includes the handoff to the serving engine and other processing. Figure 5b shows how these costs contribute to the overall time-to-first-token (TTFT) at different arrival rates, with the control plane overhead remaining stable and contributing around 10% of the total TTFT, while the majority of the latency is due to LLM inference. As the model size grows, the relative impact of the control plane overhead shrinks, since prefill latency scales with model size while the control plane overhead remains fixed. For time-per-output-token (TPOT), only the *Return* hop contributes per token, and as the other stages are amortized over the lifetime of the request, the resulting per-token overhead is sub-millisecond, which is negligible compared to the decoding time of each token. In a wide-area deployment with the ingress node in Frankfurt and the serving node in Switzerland, OpenTela adds ~3.5 ms to TTFT over invoking the model directly on the serving node. The network round-trip accounts for ~3.1 ms, leaving only ~0.4 ms attributable to OpenTela's routing.

We also measure how quickly state changes propagate through OpenTela's gossip network. We trigger an event at one node (either a node joining the network or an existing node starting to serve a new model) and measure the elapsed time until each remaining node has applied the corresponding update to its local registry. At cluster sizes from 8 to 128 nodes, 50% of nodes converge in 7 to 13 ms, 75% within 26 ms, and 95% within 1 second. The slowest node shows a tail of several seconds, reaching around 10 seconds with 128 nodes in the network. This tail reflects how long full cluster convergence takes, but it does not block request routing: the ingress node dispatches requests to the new endpoint as soon as its own replica reflects the update, independent of when the rest of the cluster converges.

In addition, we measure the message bandwidth overhead introduced by OpenTela. On the data plane, forwarding a request from the ingress node to the serving node adds a fixed header cost of around 450 bytes on the request path and 120 bytes on the response path, independent of the payload size. On the control plane, the background traffic scales with the mesh size rather than the request rate: an idle node exchanges only liveness messages, and its per-node traffic grows roughly linearly from 1 KB/s at 10 nodes to 8 KB/s at 50 nodes. Updates to the shared registry are more expensive: each CRDT write (e.g., a node joining or registering a model) raises per-node traffic from 1 KB/s at idle to 40 KB/s when 10 nodes re-register every 3 seconds. However, these spikes are transient and do not affect request processing, since the control plane operates asynchronously in the background.

Cross-cluster overhead. The measurements above isolate per-request cost within a single cluster. OpenTela also adds overhead at the network layer through its tunneling fabric, which becomes relevant once requests cross cluster boundaries. HPC compute nodes typically sit behind login-node gateways with no inbound connectivity and tight egress policies, so any cross-cluster path requires a tunneling layer that OpenTela provides as part of its node fabric. To quantify this cost, we measure pairwise latency and throughput between our deployment clusters and representative client locations across Switzerland and Europe.

Figure 6 reports measurements through OpenTela. Parenthesized values give direct `ping` and `iperf` results for paths where the two endpoints can establish a direct connection. Entries without parentheses correspond to paths where direct communication is not possible, and OpenTela establishes connectivity through hole punching and circuit relay. On paths where direct measurement is possible, OpenTela's tunneling adds at most 15 ms above the 19 ms direct baseline (Clariden→JSC, from 19 ms to 34 ms) and reduces throughput by up to 16% (OCI-2→OCI-1, from 503 to 423 Mbps).

Elasticity and fault-tolerance. To evaluate OpenTela in a dynamic HPC-based environment, where nodes may join or leave the network at any time, we deploy OpenTela on a system with at most 64 GH200 GPUs, serving a continuous stream of inference requests while varying the available worker pool by randomly terminating nodes in the system during the experiment. Figure 4 reports the throughput (in tokens/s) alongside the number of active replicas over time. OpenTela's throughput (red line) adapts proportionally, on average, to the number of active replicas (blue line), showing that OpenTela can effectively adjust to changes in compute capacity without manual intervention. Furthermore, despite the abrupt termination of nodes, we observed zero user-facing HTTP errors during the entire experiment. This confirms

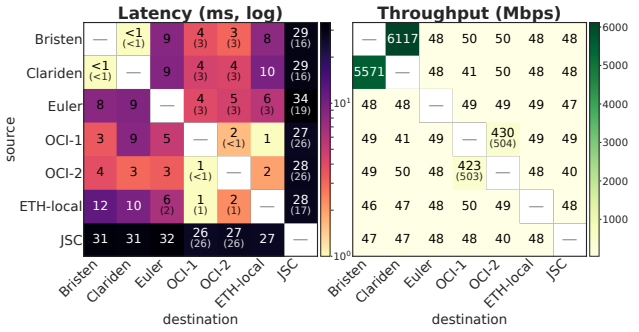


Figure 6: Network conditions across HPC clusters. Left: latency (ms, log scale), right: throughput (Mbps). Parenthesized values are direct `ping` or `iperf` baselines when reachable directly without hole punching. Unparenthesized entries use OpenTela's tunneling via hole punching and circuit relay.

Table 1: We compare how OpenTela vs. a memory-proportional heuristic policy allocate hardware for different models running in a cluster of 24 A100s and 32 GH200s.

	13B	34B	70B
MemP	4×A100	16×A100	32×GH200+4×A100
Ours	6×A100	16×GH200+10×A100	16×GH200+8×A100

that our decentralized registry maintains service continuity even under high churn rates, successfully routing requests to remaining healthy nodes while the cluster topology stabilizes.

Placement strategy. We evaluate the effectiveness of our placement strategy by comparing it to a memory-proportional policy, *memP*, inspired by HexGen [36], which represents a standard heuristic. It allocates GPU resources proportional to the memory demand of different models, defined as the product of the average request arrival rate and the model parameter size. We synthesized a workload modeled from real-world traces collected from OpenTela, consisting of three distinct models with varying sizes and load characteristics: `Llama-2-13b`, `CodeLlama-34b`, and `Llama-3.3-70B-Instruct`. Requests for each model follow a Poisson arrival process with average rates of 110, 185.5, and 221 req/s, respectively. We sample input and output sequence lengths from Normal distributions, with mean input lengths ranging from approximately 600 to 1170 tokens, and mean output lengths ranging from 64 to 530 tokens. Table 1 shows the placement allocation that each policy decides when given a budget of 24 A100 and 32 GH200 GPUs. Figure 7 compares the performance of the two placement decisions. The memP baseline prioritizes the largest model (70B) and overprovisioning it with GH200 and A100 GPUs. This does not leave enough resources for the 34B model, resulting in lower aggregate performance. In contrast, our heterogeneity-aware strategy balances allocation across model and GPU types. We do not claim that our policy is globally optimal. Our goal here is simply to highlight the importance of explicitly modeling system heterogeneity to improve resource utilization.

Simulator validation. Figure 8 compares real and simulated end-to-end latency on H100 and RTX 3090 GPUs across three configurations that vary input length, output length, and batch size. The simulator's absolute latency estimates deviate from real measurements by up to 10%, primarily due to accumulated errors in decode time estimation. The simulator captures relative performance more accurately, predicting the H100 speedup over RTX 3090 within 6% of measured values across all configurations. This shows the simulator can help OpenTela make heterogeneity-aware scheduling decisions.

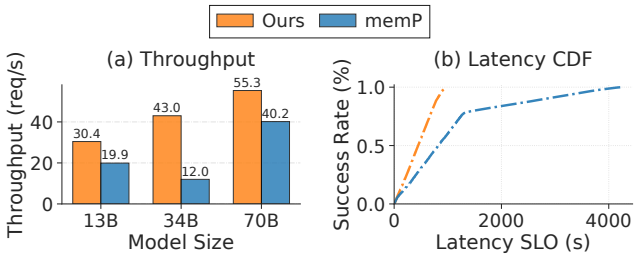


Figure 7: Throughput and latency comparison between our placement method and the memory-proportional approach.

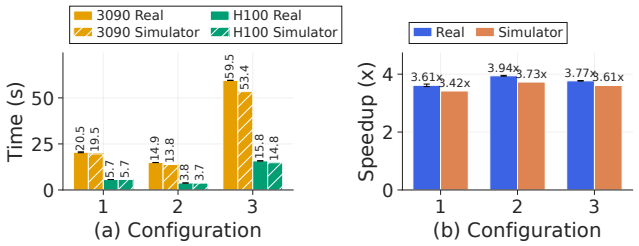


Figure 8: End-to-end performance of H100 and RTX 3090, measured on real hardware and predicted by our simulator. Each configuration is specified as (input length, output length, batch size), with lengths in tokens: **(1)** (1024, 1024, 1), **(2)** (2048, 512, 4), **(3)** (512, 2048, 8). (Left) Absolute latency in seconds. (Right) Speedup of H100 over RTX 3090.

6 Swiss AI Deployment and Trace Analysis

OpenTela is currently deployed as the serving backbone for the Swiss AI Initiative, a multi-institutional open-science research initiative for developing and providing access to open-source frontier AI models. At the time of writing, we have been operating OpenTela for over 22 months. It has served billions of tokens to over 1000 researchers across multiple institutions. We first discuss our cluster setup and then describe the anonymized trace dataset that we collected over a 16-month period (§6.1). We then characterize the LLM serving workloads from this trace (§6.2).

Cluster setup. The Swiss AI deployment spans three subclusters within the Alps supercomputer at CSCS, each with distinct hardware and management stacks. *Bristen* is an x86-64 cluster equipped with A100 GPUs, managed by Slurm, and used primarily for smaller-scale and development workloads. *Clariden* is a large-scale Slurm cluster equipped with GH200 superchips, used for production inference at scale. *Breithorn* is a small Kubernetes-managed cluster, also equipped with

GH200 superchips, intended for long-running, dedicated serving. Despite their administrative independence, all three sub-clusters participate in a single OpenTela node network. Compute nodes in all three sub-clusters sit behind network address translation (NAT) and cannot be reached directly from outside CSCS, so we run an OpenTela node on another CPU-only Kubernetes cluster as the public ingress point. Within CSCS, inter-cluster latency remains under 1 ms with multi-Gbps throughput between sub-clusters (Figure 6). Researchers and users interact with OpenTela's unified API endpoint without awareness of which sub-cluster handles their request.

Operational model. Researchers with Slurm access contribute nodes to the shared pool by wrapping their serving commands in the OpenTela CLI. The Breithorn Kubernetes partition hosts a small set of models that the community has designated for continuous availability (currently the Apertus family of open-weight models [30] developed within the Swiss AI initiative, along with a small number of other popular open models). The remaining models are served on-demand from Slurm allocations on Bristen and Clariden, with node lifecycle managed through the gossip network as described in §3. The API frontend runs as a persistent service on Breithorn and serves as the ingress node of the gossip network. It handles authentication via institutional credentials, usage tracking, and request routing across all three sub-clusters.

6.1 Trace Overview

We release the anonymized trace dataset collected from July 2024 to October 2025 to facilitate future research on AI serving system infrastructure. This dataset includes over 13 million requests, with over 15 billion tokens processed. For each request, we recorded its timestamp, model identifier, aggregate input and output token counts, and sampling parameters (e.g., temperature, maximum tokens). Due to privacy and compliance requirements, we neither use nor disclose the actual content of both input prompts and output responses in our traces. However, we do analyze prefix sharing and token reuse characteristics using the approach first described in Mooncake [53], which involves irreversibly hashing the input tokens and putting 16 consecutive tokens into a single hash bucket. A bucket ID will be equal to another if and only if the 16 tokens in both buckets are identical. We hash the bucket ID so that no information can be inferred from the ID. Hence, our trace does not include any user-identifiable information. We still allow users to bypass the usage tracking in cases where it is necessary to ensure data compliance.

In Table 2, we summarize key differences between our traces and existing public traces. Most existing traces focus on a single or very few models (often a popular closed-source LLM such as GPT-3.5 or GPT-4) while our trace includes a diverse set of real-world models, with over 142 different models being requested. Among those models, 46 are existing open-weight models that are publicly available, while the remaining 96 models are custom-trained models developed within our researchers and users community. We omit ShareGPT from this comparison because it exists in several unofficial versions and consists of conversation sessions from proprietary models without timestamps or other request-level metadata. Additionally, our traces span a longer period of time, allowing us to analyze long-term trends in model usage and request patterns.

Trace	Span	Schema	# Models	LLM
BurstGPT [69]	121 days	T, M, I/O	2	✓
Azure 1 [51]	days	T, I/O	1	✓
Azure 2 [62]	weeks	T, I/O	1	✓
Azure 3 [54]	weeks	T, I/O	1	✓
Mooncake [53]	-	-	1	✓
ChatLMSys [22]	days	M, I/O	4⋆	✓
Azure MAF [57]	days	T, M⋆	multiple⋆	✗
Ours	16 months	T, M, I/O	142	✓

Table 2: Trace characteristics summary. T: timestamp, M: model, I/O: input/output tokens, ⋆: not inference, just function invocations. Number of models only counts unique models.

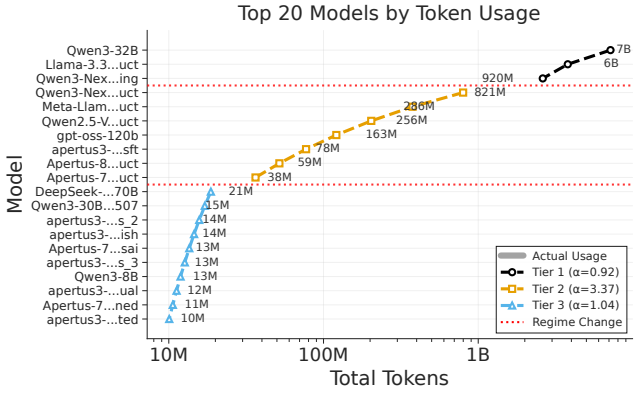


Figure 9: Overall model token usage. Each bar represents the total token consumption (input + output) for a specific model, ranked from highest to lowest usage.

6.2 Workload Characterization

Model usage distribution. Figure 9 shows the total token consumption of the top 20 models, including both input and output tokens per model, ranked from highest to lowest. We observe a multi-tier power-law distribution. The first tier consists of the top 3 models, which account for a significant portion of total token usage. These models are general-purpose LLMs that are used for various tasks. The second tier consists of multi-modal models (notably *Qwen-2.5-VL*) and also models that researchers in the Swiss AI initiative are particularly interested in evaluating (e.g., *Apertus* models, which are trained within the Swiss AI community [30]). The third tier consists of a long tail of models that are used infrequently, often for specialized tasks or by niche user groups.

Key Takeaway 1: Serving systems must efficiently support a skewed workload distribution, handling both popular general-purpose models and a long tail of diverse, niche models.

Per-model request burstiness. Next, we characterize

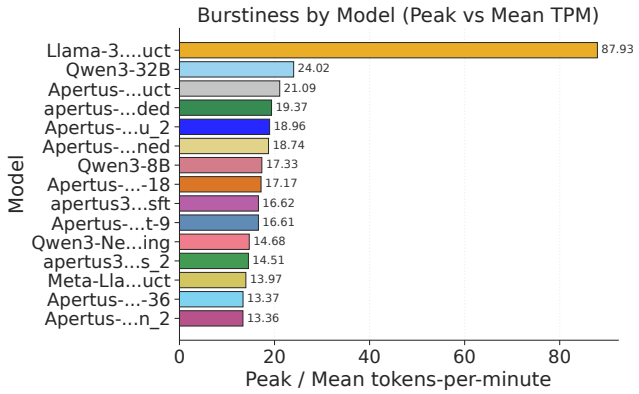


Figure 10: Peak-to-average request rate ratio per model during their active minutes.

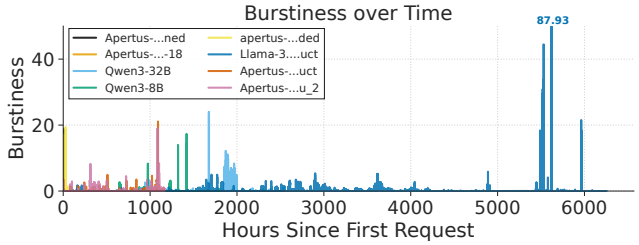


Figure 11: Burstiness over time for several representative models. Burstiness is defined as the ratio of instantaneous TPM to the model's average TPM over active periods. Timelines are aligned to the first request at $t = 0$.

model request burstiness along two dimensions: Magnitude and temporal dynamics. We define the *instantaneous burstiness* as the ratio of the instantaneous tokens per minute (TPM) to the model's average TPM. We then quantify the overall *burstiness magnitude* as the peak of this ratio. We compute the average TPM strictly over active minutes (minutes with at least one request) to avoid artificially inflating the ratio due to periods of inactivity. Figure 10 reports the magnitude, ranking models by their burstiness magnitude to highlight the most extreme bursty workloads. Overall, we observe extreme volatility across the board. Even among the top 15 most bursty models, the lowest burstiness exceeds 13, while the most bursty model reaches a ratio of over 80. This indicates that average request arrival rates underestimate peak loads. For example, Figure 11 shows the temporal request invocation dynamics for a variety of models, with timelines aligned to each model's first request ($t = 0$). This reveals when load spikes occur relative to the start of a model's operation.

Key Takeaway 2: Dynamic resource allocation and elastic scaling are essential, as static provisioning based on average rates does not satisfy peak demands while provisioning for peak would underutilize resources. Similar burstiness occurs in other real-world traces, such as ShareGPT and BurstGPT [69].

Input / Output Length Distribution. The distribution of input and output sequence lengths has significant implications for resource planning and management. We analyze the distribution of input and output token lengths of four models with highest request counts: *Llama-3.1-70B*, *Llama-3.3-70B*, *DeepSeek-R1-Distilled-Llama70B*,*Qwen-32B*. Figure 12 shows the empirical CDF of input (left) and output (right) token counts for these four models, with dotted vertical lines marking the aggregate P50, P95, and P99 across all requests. Input distributions are similar, with most prompts falling between 10^2 and 10^3 tokens, except for *Deepseek-R1-Distilled-Llama70B*, which shows a flat CDF below 10^3 tokens before a sharp jump near 10^4, indicating a dominant workload of long, fixed-length prompts. Output distributions diverge more sharply: *Llama-3.1-70B* and *Llama-3.3-70B* have a large fraction of requests producing only 1–2 tokens, consistent with classifier or guardrail traffic, while *Qwen-32B* follows a smooth sigmoid centered around 10^2–10^3 tokens.

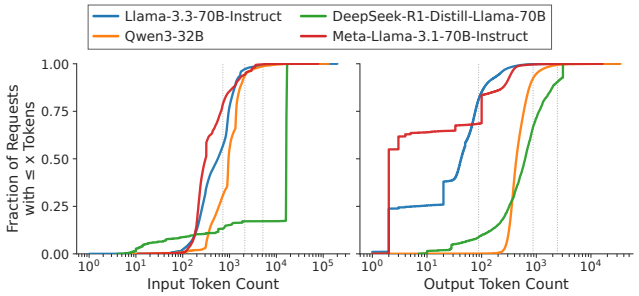


Figure 12: Cumulative distribution of token lengths for the analyzed models.

Key Takeaway 3: Output length distribution varies significantly across models. For non-reasoning models, a substantial fraction of requests produce only a few output tokens, particularly for classifier or guardrail traffic, motivating optimizations such as PrefillOnly [21]. Reasoning models, on the other hand, concentrate on longer output sequences with no comparable short output fraction. This highlights the importance of applying different optimizations for different model and workload characteristics.

Input / Output ratio characteristics. A request's resource footprint depends heavily on its structure: Input-heavy requests strain compute during the prefill phase, while output-heavy requests consume memory bandwidth during the decode phase. Therefore, we analyze how the output sequence lengths of requests compares to the total (input + output) sequence lengths. Figure 13 plots the output token ratio for `Qwen3-32B`, over three months. As shown in Figure 13, the workload is far from uniform: the ratio of output tokens to total tokens fluctuates over the three-month period.

Key Takeaway 4: Interference between compute-bound prefill and memory-bound decode stages is a critical issue in production LLM serving. This highlights the importance of optimizations such as chunked prefill [3] and prefill-decode disaggregation [34, 51, 63, 76] to mitigate this interference.

Prefix sharing characteristics. Beyond the structure of

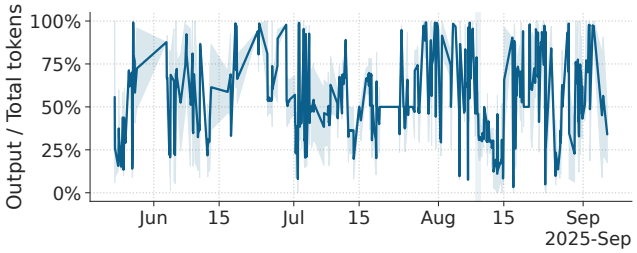


Figure 13: Output token ratio distribution for `Qwen3-32B` over three months.

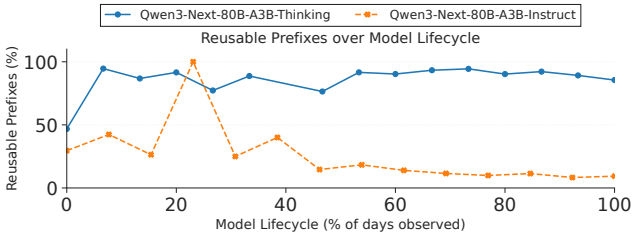


Figure 14: Reusable prefix ratio over model lifecycles for two representative models.

individual requests, we investigate the computational redundancy *across* requests. We analyze the token reuse ratio, defined as the fraction of input tokens overlapping with recent history, to identify opportunities for avoiding re-computation. Figure 14 shows how usage patterns differ between model types. The reasoning-heavy *Qwen3-Next-80B-A3B-Thinking* has a large reuse ratio exceeding 90%. This implies that the dominant workload for this reasoning model consists of deep, iterative multi-turn sessions where users build progressively on a shared context. In contrast, the standard *Instruct* variant shows a modest reuse ratio of $\sim$ 15%, reflecting the usage pattern of independent, single-turn queries. This suggests that much of the prefill computation in reasoning models may be redundant. This motivates KV-cache management optimizations like `LMCache` [11], which amortizes the quadratic attention costs inherent in long-context, iterative workflows.

Key Takeaway 5: Our analysis of *Qwen3-Next-80B-A3B-Thinking* suggests that reasoning models, particularly when used in iterative, multi-turn sessions, might exhibit high prefix reuse ratios. This suggests that optimizations focused on prefix caching and KV-cache management [11] could be particularly beneficial for amortizing the quadratic attention costs in these workloads.

Latency Distribution. Figure 15 shows the cumulative distribution of end-to-end latency (E2E, left) and output token counts for reasoning and non-reasoning requests. We observe that reasoning requests exhibit substantially higher E2E latency than non-reasoning ones, with a P95 of 173.58 s, more than 5.5$\times$ higher than the P95 of non-reasoning requests (30.77 s). The output token distribution shows the source of this gap: non-reasoning requests produce short outputs for a substantial fraction of the workload, with a median of 87 tokens and P95 of 828 tokens, while reasoning requests

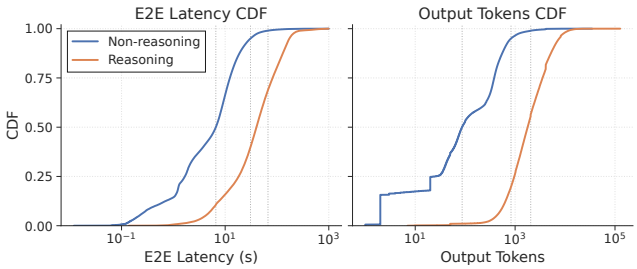


Figure 15: Cumulative distribution of end-to-end latency (left) and output token counts (right).

routinely generate much longer outputs, with a median of 1,782 tokens and P95 of 7,628 tokens, roughly an order of magnitude higher than non-reasoning requests.

Key Takeaway 6: Service Level Objectives (SLOs) must be differentiated to accommodate reasoning models, applying distinct timeout and retry policies that account for their significantly higher latency tails compared to standard models. To actively guarantee SLOs, systems could implement "reasoning budget" [43, 70] that dynamically caps the computational depth of reasoning traces to fit within time windows, or SLO-aware model selection [9, 35] where the system dispatches queries to the appropriate models, or SLO-aware model serving [10].

7 Discussion

We reflect on our experiences designing, implementing, and operating OpenTela for real users (§7.1). We then share some observations on emerging workload patterns driven by reasoning models (§7.2). Finally, we discuss the limitations of our current design and outline directions for future work (§7.3).

7.1 Lessons learned

Flexibility and interoperability as first-class citizens. We observed that researchers often have varying preferences for serving engines and hardware capabilities, and that the flexibility to adapt to different hardware and software environments is crucial for the deployment and adoption of OpenTela. In the context of LLM serving, the widespread adoption of the unified OpenAI-compatible interface across these serving engines provided a standardized abstraction layer. This allowed us to design OpenTela to be engine-agnostic by default. Furthermore, we evaluated whether Kubernetes-based orchestration would be a better fit for our use case. However, given our reliance on Slurm-based HPC clusters for GPU resources, we found that integrating with existing HPC infrastructure provided a more seamless experience for our users.

Need for more expressive abstractions. While the OpenAI-compatible interface simplifies interaction, it obscures critical execution details like model quantization levels and context limits. This creates ambiguity when users have strict expectations regarding numerical precision and other parameters critical for model quality. To mitigate this, we currently require model providers to adhere to strict naming conventions (e.g., explicitly tagging quantization bits in

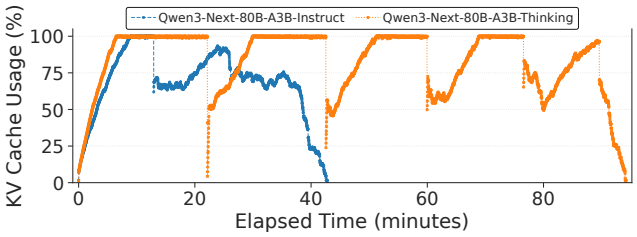


Figure 16: KV cache usage comparison between reasoning and non-reasoning models.

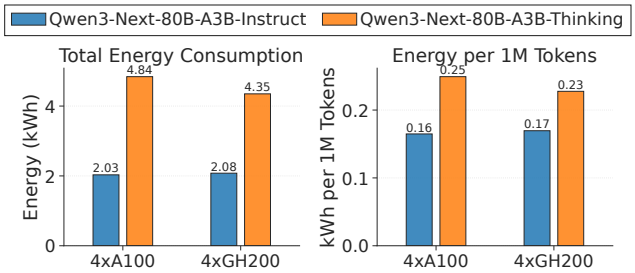


Figure 17: Power consumption of reasoning and non-reasoning models.

model names) to ensure transparency. This highlights a gap in current LLM serving protocols: the lack of a protocol for negotiating model quality of service beyond simple identifiers.

Importance of proper systematic modeling. Early iterations of OpenTela relied on a centralized database for tracking GPU worker availability. We found that enforcing strong consistency for metadata operations created a significant scalability bottleneck. The pivotal lesson was transitioning from ad-hoc state tracking to a formal algebraic model that represents the cluster state as a conflict-free replicated data type (CRDT). By modeling the system state as a monotonic lattice, we were able to decouple node interactions and adopt an eventually consistent model. This shift eliminated the central coordinator entirely, enabling the system to remain available even during network partitions.

Introspection, monitoring and human intervention. Debugging in a decentralized environment is inherently challenging due to the lack of direct access (e.g., SSH) to worker nodes. Consequently, we shifted our design philosophy from attempting to recover to fail-stop semantics. Instead of attempting complex runtime recovery, OpenTela relies on strict health checks, and upon detecting fatal anomalies (e.g., NCCL timeout), a node automatically evicts itself from the cluster to preserve global system integrity. To aid post-mortem analysis, we proactively collect logs to a sink before termination.

7.2 Implications of Reasoning for LLM Serving Systems

We also conduct some preliminary experiments to motivate future work on making serving systems more workload-aware and hardware-aware. In particular, since we notice that OpenTela often serves reasoning and non-reasoning variants of models, we explore the serving system implications of reasoning by running representative public benchmarks.

KV cache pressure. Reasoning models tend to produce longer outputs, which consumes additional memory compared to their non-reasoning ("Instruct") model counterparts. Figure 16 compares the KV cache usage for the *AIME 1983-2024* [68] benchmark. The non-reasoning model shows a dynamic usage pattern with rapid reclamation cycles, finishing in $\sim$ 40 minutes. In contrast, the reasoning model extends execution to $\sim$ 95 minutes and has a distinct sawtooth pattern: sharp, sustained increases represent the accumulation of transient state during chain-of-thought reasoning, followed by sharp drops upon sequence completion. *The high cost of regenerating transient "reasoning" states requires sophisticated memory management, such as offloading KV caches to secondary storage during preemption rather than discarding and recomputing them.*

Energy consumption. The longer execution time and higher memory usage of reasoning models inevitably increases energy costs. Figure 17 quantifies this impact on A100 and GH200 architectures for the *AIME 1983-2024* benchmark [68]. We find that reasoning models are not just slower but also consume more energy. The reasoning variant consumes over 2$\times$ the total energy of the non-reasoning baseline (rising from $\sim$ 2.0 kWh to $\sim$ 4.3 kWh). Crucially, the *energy cost per token* also jumps by $\sim$ 56% (from $\sim$ 0.16 to $\sim$ 0.25 kWh per million tokens). This non-linear scaling stems from a compound effect: The model generates more tokens (volume), and as the context window expands, the quadratic complexity of self-attention makes every subsequent decoding step more computationally intensive. Furthermore, we compare the energy consumption across two GPU architectures. While GH200s deliver superior raw performance, their energy consumption is on par with A100s for the same workload. We attribute this to two factors associated with this particular workload. First, the workload is memory-bound by the KV cache, restricting the batch sizes and preventing full saturation of GH200's compute units. Second, the sparse activation of MoE models exacerbates this by requiring more memory capacity while demanding relatively less compute. This aligns with previous findings [12, 15]. *Future serving systems could improve energy efficiency for reasoning models by applying memory optimizations, such as KV cache compression or offloading, as the workload is constrained by memory capacity rather than raw compute power.*

7.3 Limitations and Future Work

While OpenTela successfully bridges the gap between HPC infrastructure and model serving, operating a decentralized user-space overlay at this scale has revealed limitations.

Protocol expressiveness and quality of service (QoS). As discussed earlier, the adoption of a unified, OpenAI-compatible interface simplifies client integration but abstracts away critical configuration nuances; a client cannot easily request specific trade-offs, such as lower precision for lower

latency or guaranteed precision without relying on rigid naming conventions. We propose extending the serving protocol to support QoS negotiation. Future versions of OpenTela will allow clients to specify constraints (e.g., quantization $\geq$ 4-bit, specific GPU type) in the request metadata, and the system will route requests to nodes that satisfy these constraints.

Availability. OpenTela currently operates on a best-effort basis and does not provide strong availability guarantees. We plan to introduce tiered service levels. By categorizing resources based on their expected stability (e.g., differentiating between stably owned partitions and volatile scavenger queues), the scheduler could route critical workloads to more stable nodes, while batch processing tasks can exploit volatile, lower-priority capacity to maximize cluster throughput without compromising reliability for high-priority users.

Cross-node serving optimizations. Several recent serving optimizations that would benefit our workload are not yet supported in OpenTela. KV cache reuse works only when a request lands on a node already holding the relevant prefix, and prefill and decode disaggregation [51, 76] is unsupported because the OpenTela router treats each request as an atomic unit. A global KV cache layer [11] across the mesh and prefill-decode disaggregated deployment are both future work.

8 Related Work

Heterogeneous LLM serving. Heterogeneous LLM serving refers to systems that utilize a mix of GPU types with varying hardware characteristics—such as differing computational and memory capacities—to optimize resource utilization and cost-efficiency. Several recent works [29, 33, 37, 52, 66] have addressed this domain. Petals [5] is the first open-source system that allows users to serve LLMs on a set of decentralized hardware. HexGen [36] introduces asymmetric partitioning and designs advanced scheduling techniques to deploy generative inference in heterogeneous and decentralized settings. SkyServe [42] leverages spot instances across different regions and clouds to ensure availability for AI models serving, but does not focus on distributed HPC environments. LLM-ServingSim [13] adopts a trace-driven performance modeling approach and provides a simulation framework to evaluate the performance of heterogeneous LLM serving systems. ThunderServe [34] addresses the distinct computational characteristics of the prefill and decoding phases by deploying them on heterogeneous GPUs matched to their respective workload requirements. Similarly, Helix [46] formulates the optimization of heterogeneous GPU and network connections as a max-flow problem, utilizing mixed-integer linear programming to determine optimal model deployment, SageServe [32] formulates the optimization strategies as an optimization problem and solves it with integer linear programming (ILP). Collectively, these works demonstrate that the strategic utilization of heterogeneous resources significantly enhances the cost-efficiency of LLM serving. Unlike existing works on single-model serving, our service scheduler jointly addresses model heterogeneity (varying LLMs) and GPU heterogeneity (varying hardware specifications) for multi-model deployment.

Multi-tenant LLM serving. Another line of research focuses on serving multiple models concurrently to maximize resource utilization. AlpaServe [40] reveals that model parallelism can be used for statistical multiplexing of multiple models, and designs a serving system that determines efficient placing and parallelizing strategies. MuxServe [22] introduces a flexible spatial-temporal multiplexing system that identifies optimal colocation strategies for multiple LLMs. When the multiple models are fine-tuned from a shared base model, DeltaZip [72] exploits the shared weights and compresses the deltas to improve serving efficiency. If the fine-tuning is performed using parameter-efficient methods (e.g., LoRA [31]), Punica [8] and S-LoRA [59] design specialized serving systems for serving adapters efficiently. In contrast, OpenTela focuses on the challenges arising from both model and hardware heterogeneity, as well as administrative burdens in large-scale deployments, particularly on HPC environments.

Federated research infrastructure. OpenTela is similar in spirit to federated testbeds such as PlanetLab [20] and CloudLab [23], which pool resources across institutions for systems and networking research. OpenTela focuses on LLM serving and operates as an overlay on existing HPC clusters rather than dedicated testbed hardware.

9 Conclusion

As open-source models proliferate and organizations seek greater control over their AI stack, serving infrastructure needs to operate across HPC clusters with heterogeneous hardware and dynamic resource availability. Decentralized orchestration overlays such as OpenTela offer a convenient alternative to centralized cloud-native control planes. OpenTela unifies fragmented and heterogeneous compute resources across diverse clusters into a single LLM serving platform, providing service discovery, fault-tolerant routing, heterogeneity-aware scheduling, and a unified serving interface on top of existing LLM inference engines, all while operating entirely in user space. Our deployment in the Swiss AI Initiative shows that this approach is practical at scale: OpenTela has served more than 15 billion tokens across 142 models to over 1000 researchers from multiple institutions.

Acknowledgements

Beyond the authors, dozens of engineers and student assistants have contributed to the design, implementation, and deployment of OpenTela. We are grateful to Aryan Ahadinia, Elia Palme, Stefano Schuppli and the rest of the Swiss AI team and the Swiss Supercomputing Center (CSCS), in particular Joost VandeVondele and Thomas Schulthess, for their support. We thank Prof. Chaojian Li, Foteini Strati, Cedric Renggli, anonymous reviewers, and our shepherd for their insightful feedback and suggestions. This work was supported as part of the Swiss AI Initiative by a grant from the Swiss National Supercomputing Centre (CSCS) on Alps.

References

[1] Understanding slurm for ai/ml workloads. https://www.whitefiber.com/blog/understanding-slurm-for-ai-ml-workloads, November 2025.

[2] Amey Agrawal, Nitin Kedia, Ashish Panwar, Jayashree Mohan, Nipun Kwatra, Bhargav Gulavani, Alexey Tumanov, and Ramachandran Ramjee. Taming {Throughput-Latency} tradeoff in {LLM} inference with {Sarathi-Serve}. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, pages 117–134, 2024.

[3] Arney Agrawal, Nitin Kedia, Ashish Panwar, Jayashree Mohan, Nipun Kwatra, Bhargav S. Gulavani, Alexey Tumanov, and Ramachandran Ramjee. Efficient llm inference via chunked prefills. *SIGOPS Oper. Syst. Rev.*, 59(1):9–16, August 2025.

[4] Allen AI. Allen institute for artificial intelligence. https://allenai.org/, 2026. Accessed: 2026-05-17.

[5] Alexander Borzunov, Dmitry Baranchuk, Tim Dettmers, Maksim Riabinin, Younes Belkada, Artem Chumachenko, Pavel Samygin, and Colin Raffel. Petals: Collaborative inference and fine-tuning of large models. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 3: System Demonstrations)*, pages 558–568, 2023.

[6] Shiyi Cao, Shu Liu, Tyler Griggs, Peter Schafhalter, Xiaoxuan Liu, Ying Sheng, Joseph E Gonzalez, Matei Zaharia, and Ion Stoica. Moe-lightning: High-throughput moe inference on memory-constrained gpus. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1*, pages 715–730, 2025.

[7] Marco Cascella, Jonathan Montomoli, Valentina Bellini, and Elena Bignami. Evaluating the feasibility of chatgpt in healthcare: an analysis of multiple clinical and research scenarios. *Journal of medical systems*, 47(1):33, 2023.

[8] Lequn Chen, Zihao Ye, Yongji Wu, Danyang Zhuo, Luis Ceze, and Arvind Krishnamurthy. Punica: Multi-tenant lora serving. *Proceedings of Machine Learning and Systems*, 6:1–13, 2024.

[9] Lingjiao Chen, Matei Zaharia, and James Zou. Frugalgpt: How to use large language models while reducing cost and improving performance. *arXiv preprint arXiv:2305.05176*, 2023.

[10] Siyuan Chen, Zhipeng Jia, Samira Khan, Arvind Krishnamurthy, and Phillip B Gibbons. Slos-serve: Optimized serving of multi-slo llms. *arXiv preprint arXiv:2504.08784*, 2025.

[11] Yihua Cheng, Yuhan Liu, Jiayi Yao, Yuwei An, Xiaokun Chen, Shaoting Feng, Yuyang Huang, Samuel Shen, Kuntai Du, and Junchen Jiang. Lmcache: An efficient kv cache layer for enterprise-scale llm inference. *arXiv preprint arXiv:2510.09665*, 2025.

[12] Krishna Teja Chitty-Venkata, Siddhisanket Raskar, Bharat Kale, Farah Ferdaus, Aditya Tanikanti, Ken Raffenetti, Valerie Taylor, Murali Emani, and Venkatram Vishwanath. Llm-inference-bench: Inference benchmarking of large language models on ai accelerators. In *SC24-W: Workshops of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1362–1379. IEEE, 2024.

[13] Jaehong Cho, Hyunmin Choi, Guseul Heo, and Jongse Park. LLMServingSim 2.0: A Unified Simulator for Heterogeneous and Disaggregated LLM Serving Infrastructure. In *2026 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 1–14, 2026.

[14] Zhendong Chu, Shen Wang, Jian Xie, Tinghui Zhu, Yibo Yan, Jinheng Ye, Aoxiao Zhong, Xuming Hu, Jing Liang, Philip S Yu, et al. Llm agents for education: Advances and applications. *arXiv preprint arXiv:2503.11733*, 2, 2025.

[15] Jae-Won Chung, Jeff J Ma, Ruofan Wu, Jiachen Liu, Oh Jun Kweon, Yuxuan Xia, Zhiyu Wu, and Mosharaf Chowdhury. The ml. energy benchmark: Toward automated inference energy measurement and optimization. *arXiv preprint arXiv:2505.06371*, 2025.

[16] European Commission. Ai factories. https://digital-strategy.ec.europa.eu/en/policies/ai-factories, 2025. Accessed: 2025-12-11.

[17] McKinsey & Company. The cost of compute: A $7 trillion race to scale data centers. https://www.mckinsey.com/industries/technology-media-and-telecommunications/our-insights/the-cost-of-compute-a-7-trillion-dollar-race-to-scale-data-centers, 2025. Accessed: 2025-12-11.

[18] FastAPI Contributors. Fastapi framework, high performance, easy to learn, fast to code, ready for production. https://fastapi.tiangolo.com/, 2025. Accessed: 2025-12-10.

[19] Gin Contributors. Gin web framework. https://gin-gonic.com/, 2025. Accessed: 2025-12-01.

[20] David E. Culler. PlanetLab: An open, Community-Driven infrastructure for experimental Planetary-Scale services. Seattle, WA, March 2003. USENIX Association.

[21] Kuntai Du, Bowen Wang, Chen Zhang, Yiming Cheng, Qing Lan, Hejian Sang, Yihua Cheng, Jiayi Yao, Xiaoxuan Liu, Yifan Qiao, Ion Stoica, and Junchen Jiang. Prefillonly: An inference engine for prefill-only workloads in large language model applications. In *Proceedings of the ACM SIGOPS 31st Symposium on Operating Systems Principles*, SOSP '25, page 399–414, New York, NY, USA, 2025. Association for Computing Machinery.

[22] Jiangfei Duan, Runyu Lu, Haojie Duanmu, Xiuhong Li, Xingcheng Zhang, Dahua Lin, Ion Stoica, and Hao Zhang. Muxserve: flexible spatial-temporal multiplexing for multiple llm serving. In *Proceedings of the 41st International Conference on Machine Learning*, ICML'24. JMLR.org, 2024.

[23] Dmitry Duplyakin, Robert Ricci, Aleksander Maricq, Gary Wong, Jonathon Duerig, Eric Eide, Leigh Stoller, Mike Hibler, David Johnson, Kirk Webb, Aditya Akella, Kuangching Wang, Glenn Ricart, Larry Landweber, Chip Elliott, Michael Zink, Emmanuel Cecchet, Snigdhaswin Kar, and Prabodh Mishra. The design and operation of CloudLab. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 1–14, Renton, WA, July 2019. USENIX Association.

[24] U.S. National Science Foundation. National artificial intelligence research resource pilot. https://www.nsf.gov/focus-areas/ai/nairr, 2025. Accessed: 2025-12-09.

[25] Canada Government. Ai sovereign compute infrastructure program. https://ised-isde.canada.ca/site/ised/en/ai-sovereign-compute-infrastructure-program, 2026. Accessed: 2026-05-17.

[26] Singapore Government. National ai strategy. https://www.smartnation.gov.sg/initiatives/national-ai-strategy/, 2026. Accessed: 2026-05-17.

[27] South Korea Government. Korea's artificial intelligence action plan. https://aikorea.go.kr/upload/contents/files/Korea's_Artificial_Intelligence_Action_Plan_20260429.pdf, 2026. Accessed: 2026-05-17.

[28] UK Government. Uk compute roadmap. https://www.gov.uk/government/publications/uk-compute-roadmap/uk-compute-roadmap, 2026. Accessed: 2026-05-17.

[29] Tyler Griggs, Xiaoxuan Liu, Jiaxiang Yu, Doyoung Kim, Wei-Lin Chiang, Alvin Cheung, and Ion Stoica. Mélange: Cost efficient large language model serving by exploiting gpu heterogeneity. *arXiv preprint arXiv:2404.14527*, 2024.

[30] Alejandro Hernández-Cano, Alexander Hägele, Allen Hao Huang, Angelika Romanou, Antoni-Joan Solergibert, Barna Pasztor, Bettina Messmer, Dhia Garbaya, Eduard Frank Ďurech, Ido Hakimi, et al. Apertus: Democratizing open and compliant llms for global language environments. *arXiv preprint arXiv:2509.14233*, 2025.

[31] Edward J Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, Weizhu Chen, et al. Lora: Low-rank adaptation of large language models. *ICLR*, 1(2):3, 2022.

[32] Kunal Jain, A. Parayil, Ankur Mallick, Rujia Wang, Renee St. Amant, Chetan Bansal, Victor Ruehle, Saravan Rajmohan, Shashwat Jaiswal, Yogesh Simmhan, Anoop Kulkarni, and Steve Kofsky. Serving Models, Fast and Slow:Optimizing Heterogeneous LLM Inferencing Workloads at Scale. In *ACM Sigmetrics 2026*, June 2026.

[33] Youhe Jiang, Fangcheng Fu, Xiaozhe Yao, Guoliang He, Xupeng Miao, Ana Klimovic, Bin Cui, Binhang Yuan, and Eiko Yoneki. Demystifying cost-efficiency in llm serving over heterogeneous gpus. In *Proceedings of the 42nd International Conference on Machine Learning*, ICML'25. JMLR.org, 2025.

[34] Youhe Jiang, Fangcheng Fu, Xiaozhe Yao, Taiyi Wang, Bin Cui, Ana Klimovic, and Eiko Yoneki. Thunderserve: High-performance and cost-efficient llm serving in cloud environments. *arXiv preprint arXiv:2502.09334*, 2025.

[35] Youhe Jiang, Fangcheng Fu, Wanru Zhao, Stephan Rabanser, Nicholas D Lane, and Binhang Yuan. Cascadia: A cascade serving system for large language models. *arXiv preprint arXiv:2506.04203*, 2025.

[36] Youhe Jiang, Ran Yan, Xiaozhe Yao, Yang Zhou, Beidi Chen, and Binhang Yuan. Hexgen: Generative inference of large language model over heterogeneous environment. *arXiv preprint arXiv:2311.11514*, 2023.

[37] Youhe Jiang, Ran Yan, and Binhang Yuan. Hexgen-2: Disaggregated generative inference of llms in heterogeneous environment. *arXiv preprint arXiv:2502.07903*, 2025.

[38] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph Gonzalez, Hao Zhang, and Ion Stoica. Efficient memory management for large language model serving with pagedattention. In *Proceedings of the 29th symposium on operating systems principles*, pages 611–626, 2023.

[39] Protocol Labs. libp2p - the peer-to-peer network stack. https://libp2p.io/, 2025. Accessed: 2025-12-01.

[40] Zhuohan Li, Lianmin Zheng, Yinmin Zhong, Vincent Liu, Ying Sheng, Xin Jin, Yanping Huang, Zhifeng Chen, Hao Zhang, Joseph E Gonzalez, et al. {AlpaServe}: Statistical multiplexing with model parallelism for deep learning serving. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*, pages 663–679, 2023.

[41] Pedro Garcia Lopez, Daniel Barcelona Pons, Marcin Copik, Torsten Hoefler, Eduardo Quiñones, Maciej Malawski, Peter Pietzutch, Alberto Marti, Thomas Ohlson Timoudas, and Aleksander Slominski. Ai factories: It's time to rethink the cloud-hpc divide. *arXiv preprint arXiv:2509.12849*, 2025.

[42] Ziming Mao, Tian Xia, Zhanghao Wu, Wei-Lin Chiang, Tyler Griggs, Romil Bhardwaj, Zongheng Yang, Scott Shenker, and Ion Stoica. Skyserve: Serving ai models across regions and clouds with spot instances. In *Proceedings of the Twentieth European Conference on Computer Systems*, pages 159–175, 2025.

[43] Sara Vera Marjanović, Arkil Patel, Vaibhav Adlakha, Milad Aghajohari, Parishad BehnamGhader, Mehar Bhatia, Aditi Khandelwal, Austin Kraft, Benno Krojer, Xing Han Lù, et al. Deepseek-r1 thoughtology: Let's think about llm reasoning. *arXiv preprint arXiv:2504.07128*, 2025.

[44] Maxime Martinasso, Mark Klein, and Thomas Schulthess. Alps, a versatile research infrastructure. In *Proceedings of the Cray User Group*, pages 156–165. 2025.

[45] Petar Maymounkov and David Mazieres. Kademlia: A peer-to-peer information system based on the xor metric. In *International workshop on peer-to-peer systems*, pages 53–65. Springer, 2002.

[46] Yixuan Mei, Yonghao Zhuang, Xupeng Miao, Juncheng Yang, Zhihao Jia, and Rashmi Vinayak. Helix: Serving large language models over heterogeneous gpus and network via max-flow. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1*, pages 586–602, 2025.

[47] Xupeng Miao, Yujie Wang, Youhe Jiang, Chunan Shi, Xiaonan Nie, Hailin Zhang, and Bin Cui. Galvatron: Efficient transformer training over multiple gpus using automatic parallelism. *Proc. VLDB Endow.*, 16(3):470–479, November 2022.

[48] Erina Seh-Young Moon, Matthew Tamura, Angelina Zhai, Nuzaira Habib, Behnaz Shirazi, Altaf Kassam, Devansh Saxena, and Shion Guha. The promises and perils of using llms for effective public services. In *Proceedings of the 2026 CHI Conference on Human Factors in Computing Systems*, pages 1–21, 2026.

[49] National Institute of Advanced Industrial Science and Technology (AIST). Ai bridging cloud infrastructure (abci). https://abci.ai/, 2025. Accessed: 2025-12-09.

[50] OpenAI. Api overview | openai api reference. https://developers.openai.com/api/reference/overview, 2026. Accessed: 2026-05-25.

[51] Pratyush Patel, Esha Choukse, Chaojie Zhang, Aashaka Shah, Íñigo Goiri, Saeed Maleki, and Ricardo Bianchini. Splitwise: Efficient generative llm inference using phase splitting. In *2024 ACM/IEEE 51st Annual International Symposium on Computer Architecture (ISCA)*, pages 118–132. IEEE, 2024.

[52] You Peng, Youhe Jiang, Chen Wang, and Binhang Yuan. Hexgen-text2sql: Optimizing llm inference request scheduling for agentic text-to-sql workflow. *arXiv preprint arXiv:2505.05286*, 2025.

[53] Ruoyu Qin, Zheming Li, Weiran He, Jialei Cui, Feng Ren, Mingxing Zhang, Yongwei Wu, Weimin Zheng, and Xinran Xu. Mooncake: Trading more storage for less computation — a KVCache-centric architecture for serving LLM chatbot. In *23rd USENIX Conference on File and Storage Technologies (FAST 25)*, pages 155–170, Santa Clara, CA, February 2025. USENIX Association.

[54] Haoran Qiu, Anish Biswas, Zihan Zhao, Jayashree Mohan, Alind Khare, Esha Choukse, Íñigo Goiri, Zeyu Zhang, Haiying Shen, Chetan Bansal, Ramachandran Ramjee, and Rodrigo Fonseca. Modserve: Modality- and stage-aware resource disaggregation for scalable multimodal model serving. In *Proceedings of the 2025 ACM Symposium on Cloud Computing (SoCC 2025)*, New York, NY, USA, 2025. Association for Computing Machinery.

[55] Francesca Rossi, Peter Van Beek, and Toby Walsh. *Handbook of constraint programming*. Elsevier, 2006.

[56] SchedMD. Slurm scheduler by schedmd. https://www.schedmd.com/slurm/. Accessed: 2025-06-18.

[57] Mohammad Shahrad, Rodrigo Fonseca, Inigo Goiri, Gohar Chaudhry, Paul Batum, Jason Cooke, Eduardo Laureano, Colby Tresness, Mark Russinovich, and Ricardo Bianchini. Serverless in the wild: Characterizing and optimizing the serverless workload at a large cloud provider. In *2020 USENIX annual technical conference (USENIX ATC 20)*, pages 205–218, 2020.

[58] Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. Conflict-free replicated data types. In *Symposium on Self-Stabilizing Systems*, pages 386–400. Springer, 2011.

[59] Ying Sheng, Shiyi Cao, Dacheng Li, Coleman Hooper, Nicholas Lee, Shuo Yang, Christopher Chou, Banghua Zhu, Lianmin Zheng, Kurt Keutzer, Joseph E. Gonzalez, and Ion Stoica. Slora: Scalable serving of thousands of lora adapters. In P. Gibbons, G. Pekhimenko, and C. De Sa, editors, *Proceedings of Machine Learning and Systems*, volume 6, pages 296–311, 2024.

[60] Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. Megatron-lm: Training multi-billion parameter language models using model parallelism. *arXiv preprint arXiv:1909.08053*, 2019.

[61] Sina Shool, Sara Adimi, Reza Saboori Amleshi, Ehsan Bitaraf, Reza Golpira, and Mahmood Tara. A systematic review of large language model (llm) evaluations in clinical medicine. *BMC Medical Informatics and Decision Making*, 25(1):117, 2025.

[62] Jovan Stojkovic, Chaojie Zhang, Íñigo Goiri, Josep Torrellas, and Esha Choukse. Dynamollm: Designing llm inference clusters for performance and energy efficiency. In *2025 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 1348–1362. IEEE, 2025.

[63] Foteini Strati, Sara McAllister, Amar Phanishayee, Jakub Tarnawski, and Ana Klimovic. Déjàvu: Kv-cache streaming for fast, fault-tolerant generative llm serving. In *Proceedings of the 41st International Conference on Machine Learning*, ICML'24. JMLR.org, 2024.

[64] SwissAI. Swiss ai initiative. https://www.swiss-ai.org/, 2025. Accessed: 2025-12-09.

[65] Nebius Team. Slurm workload manager: The go-to scheduler for hpc and ai workloads. https://nebius.com/blog/posts/slurm-workload-manager, 2024.

[66] Chris Tong, Youhe Jiang, Gufeng Chen, Tianyi Zhao, Sibian Lu, Wenjie Qu, Eric Yang, Lynn Ai, and Binhang Yuan. Parallax: Efficient llm inference service over decentralized environment. *arXiv preprint arXiv:2509.26182*, 2025.

[67] UncoverAlpha. The next tectonic shift in ai: Inference. https://www.uncoveralpha.com/p/the-next-tectonic-shift-in-ai-inference, 2024. Accessed: 2025-12-11.

[68] Hemish Veeraboina. Aime problem set 1983-2024. https://www.kaggle.com/datasets/hemishveeraboina/aime-problem-set-1983-2024, 2024.

[69] Yuxin Wang, Yuhan Chen, Zeyu Li, Xueze Kang, Yuchu Fang, Yeju Zhou, Yang Zheng, Zhenheng Tang, Xin He, Rui Guo, Xin Wang, Qiang Wang, Amelie Chi Zhou, and Xiaowen Chu. Burstgpt: A real-world workload dataset to optimize llm serving systems. In *Proceedings of the 31st ACM SIGKDD Conference on Knowledge Discovery and Data Mining V.2*, KDD '25, page 5831–5841, New York, NY, USA, 2025. Association for Computing Machinery.

[70] Hao Wen, Xinrui Wu, Yi Sun, Feifei Zhang, Liye Chen, Jie Wang, Yunxin Liu, Yunhao Liu, Ya-Qin Zhang, and Yuanchun Li. Budgetthinker: Empowering budget-aware llm reasoning with control tokens. *arXiv preprint arXiv:2508.17196*, 2025.

[71] Samuel Williams, Andrew Waterman, and David Patterson. Roofline: an insightful visual performance model for multicore architectures. *Communications of the ACM*, 52(4):65–76, 2009.

[72] Xiaozhe Yao, Qinghao Hu, and Ana Klimovic. Deltazip: Efficient serving of multiple full-model-tuned llms. In *Proceedings of the Twentieth European Conference on Computer Systems*, pages 110–127, 2025.

[73] Andy B Yoo, Morris A Jette, and Mark Grondona. Slurm: Simple linux utility for resource management. In *Workshop on job scheduling strategies for parallel processing*, pages 44–60. Springer, 2003.

[74] Zhihang Yuan, Yuzhang Shang, Yang Zhou, Zhen Dong, Zhe Zhou, Chenhao Xue, Bingzhe Wu, Zhikai Li, Qingyi Gu, Yong Jae Lee, et al. Llm inference unveiled: Survey and roofline model insights. *arXiv preprint arXiv:2402.16363*, 2024.

[75] Lianmin Zheng, Liangsheng Yin, Zhiqiang Xie, Chuyue Livia Sun, Jeff Huang, Cody Hao Yu, Shiyi Cao, Christos Kozyrakis, Ion Stoica, Joseph E Gonzalez, et al. Sglang: Efficient execution of structured language model programs. *Advances in neural information processing systems*, 37:62557–62583, 2024.

[76] Yinmin Zhong, Shengyu Liu, Junda Chen, Jianbo Hu, Yibo Zhu, Xuanzhe Liu, Xin Jin, and Hao Zhang. {DistServe}: Disaggregating prefill and decoding for goodput-optimized large language model serving. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, pages 193–210, 2024.

ARTIFACT EVALUATED
USENIX
AVAILABLE

KAIROX: Adaptive GPU–CPU Hybrid LLM Inference via Online Neuron Balancing

Yapeng Jiang[1,3,5], *Minghao Gan*[1], *Zicong Hong*[2]*, *Wuhui Chen*[1,3], *Junyuan Liang*[1], *Yue Yu*[3]†, *Meng Guo*[4], *Zibin Zheng*[1]

[1]*Sun Yat-sen University,* [2]*École Polytechnique Fédérale de Lausanne,* [3]*Peng Cheng Laboratory,* [4]*Qilu University of Technology,* [5]*Zhuhai Key Laboratory of Trusted Large Language Models*

Abstract

Hybrid LLM inference systems exploit both GPU and CPU for computation and memory but are bottlenecked by the CPU's lower computational capabilities. Recent approaches leverage activation sparsity by offline partitioning FFN neurons into "hot" (frequently activated) and "cold" (rarely activated) sets. This approach retains critical computations on the GPU, yet static partitioning struggles to adapt to runtime activation changes, leading to suboptimal throughput.

We present KAIROX, an adaptive GPU–CPU hybrid inference system that addresses these limitations through online neuron balancing, a mechanism that dynamically redistributes neurons between the GPU and CPU based on activation patterns. To realize this, KAIROX introduces a Live Pipeline designed to prefetch neurons by predicting next-layer activation patterns. Furthermore, leveraging activation locality, we develop a Temporal Activation Momentum cache policy to prioritize neurons with sustained utility while minimizing transient, wasteful transfers. Finally, an Adaptive Neuron Balancer modulates the balancing intensity according to runtime resource conditions, maintaining an optimal equilibrium between competing system bottlenecks. For standard completion on consumer-grade PCs, KAIROX improves end-to-end throughput by up to 7.57×, 3.70×, 6.35×, and 3.76× over llama.cpp, PowerInfer, Neuralink, and Q-Infer, respectively. Across all evaluated settings, it achieves geomean speedups of 3.15× and 3.93× over llama.cpp on two representative PCs and around 2.1× over the three sparse baselines.

1 Introduction

Large Language Models (LLMs) have achieved remarkable success across diverse domains [10, 48, 59]; however, their substantial memory and computational requirements limit deployment primarily to cloud infrastructures [30]. While cloud-based inference remains dominant, deploying LLMs on edge devices such as personal computers is becoming increasingly important [6,34,57]. Local deployment offers significant advantages, including stronger data privacy protections [55] and reduced operational costs in practice [45].

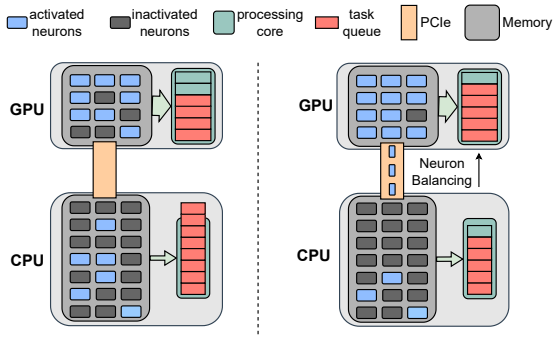


Figure 1: Two strategies for leveraging activation sparsity to compute only activated neurons (i.e., subsets of FFN weights stored in GPU or CPU memory). Left: static placement without runtime adjustment. Right: online neuron balancing with dynamic neuron transfers between the CPU and GPU.

However, the scale of modern LLMs continues to exceed the memory budgets of typical consumer hardware, making efficient on-device inference a fundamental challenge. For example, running a 13-billion-parameter model in standard FP16 precision requires more than 26 GB of VRAM—surpassing the 24 GB available even on high-end consumer GPUs such as the NVIDIA RTX 4090 [32].

To address this memory bottleneck, parameter offloading has become a widely adopted solution, wherein a portion of model weights is stored in host memory [2, 38]. In such architectures, the CPU typically performs computations on the offloaded weights, and the results are then merged with GPU-produced outputs. Frameworks such as llama.cpp [14] generally implement this using layer-wise offloading. However, because CPU computational capability is substantially weaker than that of the GPU, the CPU frequently becomes the system bottleneck, leading to increased inference latency.

Recent research provides a promising direction for alle-

*Work done while at Hong Kong University of Science and Technology.
†Corresponding author: yuy@pcl.ac.cn

viating this bottleneck through **activation sparsity** [28, 39, 43, 60]. For a given input, only a small subset of neurons—corresponding to specific rows or columns of the weight matrix—contributes to the output. These sparse activation patterns can be reliably predicted using lightweight predictors analogous to Mixture-of-Experts (MoE) gating [61]. By computing only the activated neurons, systems can substantially reduce both computation workload and memory footprint, thereby expanding the design space for hybrid inference. State-of-the-art activation-sparse inference frameworks, such as PowerInfer [41], leverage this property by statically placing frequently activated neurons on the GPU and rarely activated ones on the CPU based on offline profiling. As shown in Figure 1(left), the system computes only the neurons predicted to be active at runtime during inference.

However, static placement derived from offline profiling becomes inadequate when activation patterns shift or expand—such as during semantic drift or batch inference. These shifts cause many "cold" neurons (those residing on the CPU) to become active, turning the CPU into the dominant bottleneck. In practice, this limitation leads to early throughput saturation (e.g., at batch sizes of 4 or 8) and high variance in TPOT (ranging from 50 to 200 ms even at batch size 1), underscoring the inefficiencies of static placement in practice.

A natural solution is to shift from static placement to online neuron balancing, where neuron workloads are continuously redistributed across the GPU and CPU according to the activation patterns (see Figure 1(right)). While this approach can alleviate the CPU bottleneck, it introduces three tightly coupled system challenges: **how to achieve efficient balancing with computation overlap,** given that redistributing neurons across devices incurs substantial transfer latency and layer-wise dependencies provide limited opportunities to hide these transfers behind computation; **how to prioritize sustained neurons and avoid transient, wasteful balancing,** since activation patterns may fluctuate rapidly and simply redistributing each newly activated neuron risks repeatedly moving short-lived neurons, wasting scarce bandwidth and evicting more valuable ones; and **how to adaptively control balancing intensity under runtime resource constraints,** as aggressive balancing reduces CPU pressure but risks saturating the I/O path, whereas conservative balancing preserves bandwidth but leaves the CPU overloaded. These challenges make effective online neuron balancing fundamentally difficult.

To address these challenges, we propose KAIROX, an adaptive GPU–CPU hybrid LLM inference system for consumer-grade hardware. KAIROX enables efficient and adaptive online neuron balancing through three key designs:

Live Pipeline. The primary obstacle for dynamic redistribution is I/O latency: moving activated neurons from host memory to the GPU introduces delays, and strict layer-wise dependencies constrain opportunities for overlapping these transfers with computation. KAIROX mitigates this by predicting the activation pattern of layer $i+1$ immediately after the Attention block of layer i, allowing balancing actions to begin earlier and overlap data movement with ongoing computation. Neurons are further grouped and reordered based on co-activation patterns to improve PCIe utilization.

Locality-Aware Neuron Balancing. Even with prefetching, balancing bandwidth must be used judiciously. Activation patterns often contain short-lived spikes that trigger wasteful transfers and evict useful GPU-resident neurons. To avoid such thrashing, KAIROX leverages activation locality and introduces Temporal Activation Momentum (TAM), a lightweight policy that captures the temporal persistence of activations. TAM enables KAIROX to identify neurons with sustained utility, prioritizing their movement toward the GPU while suppressing transient ones that provide little benefit.

Adaptive Neuron Balancer. Effective online neuron balancing must reconcile CPU workload with balancing-induced I/O overhead—two competing bottlenecks during inference. KAIROX addresses this challenge through a runtime feedback mechanism that adjusts balancing intensity based on real-time resource availability. When the system becomes CPU-bound, unused I/O capacity emerges, creating an opportunity for KAIROX to migrate more computation to the GPU. Conversely, when the system becomes I/O-bound, excessive balancing saturates the transfer path; KAIROX then reduces balancing intensity to alleviate I/O pressure. This adaptive control strategy enables the system to sustain an efficient operating point under diverse runtime conditions.

Our contributions are summarized as follows:

- We design a hybrid inference pipeline that uses an adjacent predictor to relax sequential dependencies and overlap computation with data transfer, reducing exposed I/O latency during online neuron balancing.
- We propose Locality-Aware Neuron Balancing based on Temporal Activation Momentum, prioritizing sustained neurons while avoiding I/O waste on transient ones.
- We introduce an Adaptive Neuron Balancer that adjusts balancing intensity to real-time CPU and I/O availability, effectively managing the competing bottlenecks.
- We implement KAIROX on top of llama.cpp [14]. On diverse consumer-grade PCs, KAIROX improves standard-completion throughput by up to 7.57×, 3.70×, 6.35×, and 3.76× over llama.cpp [14], PowerInfer [41], Neuralink [50], and Q-Infer [29], respectively.

2 Background

2.1 LLM Inference & Activation Sparsity

The inference of LLMs consists of two stages: prefill and decoding. In prefill, the model processes the input sequence in parallel to generate the first output token. In decoding, the model operates auto-regressively, generating one token at a time until a termination token is produced or a maximum length is reached. LLMs adopt a stack of Transformer

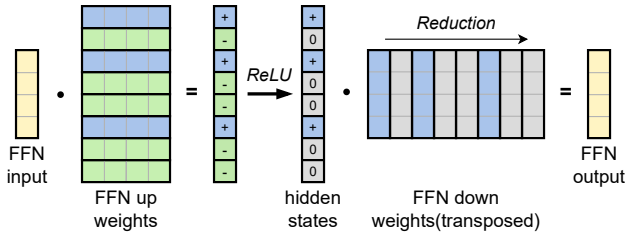


Figure 2: Illustration of activation sparsity in the up- and down-projection layers of a ReLU FFN. Activated neurons are shown in blue for visual clarity in the example below.

layers [48], each composed of an attention block and a Feed-Forward Network (FFN) [3]. While attention captures contextual dependencies, the FFN—implemented via an up-projection, an optional gated projection [36], a non-linear activation, and a down-projection—performs non-linear feature transformations. FFN blocks account for approximately 70% of parameters (e.g., 66.6% in OPT-6.7B [59] and 72.4% in LLaMA-3 8B [17], excluding embeddings), and contribute roughly 60% of prefill computation and 80% of decoding computation. Consequently, FFNs constitute the primary computational and memory bottleneck in LLM inference.

Activation Sparsity. Extensive prior work [26, 39, 40, 51] shows that LLMs exhibit substantial activation sparsity, most prominently in FFN blocks and primarily induced by the non-linear activation function (e.g., ReLU [1] or SiLU [11]). A large portion of neurons—corresponding to rows or columns of the weight matrices—produce negative or near-zero values after the first projection and are subsequently suppressed by the activation function. Figure 2 illustrates this sparsity pattern within an FFN block; for clarity, a standard projection (up- and down-projection) is shown, though the same behavior extends to gated variants. Recent studies reveal three key properties of activation sparsity: context specificity [28], where the set of activated neurons varies with the input and cannot be determined before inference; power-law distribution [41], where only a small subset of neurons is strongly activated while the majority remain inactive; and neuron co-activation [2], where neurons exhibit highly correlated activation patterns across architectures and datasets. Moreover, these works [2, 28, 41] show that FFN sparsity can be accurately predicted using lightweight predictors—analogous to MoE gating—enabling substantial reductions in computation and memory costs during LLM inference.

2.2 Resource-Constrained Inference

Deploying LLMs on edge devices (such as personal computers) has attracted increasing attention [6, 34, 57]. Compared with cloud-based solutions, local deployment offers advantages in data privacy [55], inference cost [45], and latency [20]. However, the large memory footprint of modern LLMs remains a fundamental bottleneck. For instance, a 13B model in FP16 requires at least 26 GB of memory for weights alone, exceeding the capacity of high-end consumer GPUs such as the NVIDIA RTX 4090 (24 GB) [32]. To address this constraint, recent work has explored model compression, especially quantization [22, 27], as well as system-level techniques such as offloading [18]. We categorize existing edge-inference systems by their execution strategies.

GPU-Only Inference. GPU-only frameworks primarily rely on the GPU for computation while alleviating memory pressure by offloading a portion of model weights to CPU memory or secondary storage and transferring them to the GPU on demand. Early systems such as FlexGen [38] adopt coarse-grained offloading strategies, using linear programming to schedule block-level weight swaps. While effective for high-throughput batch processing, these methods incur substantial latency for single-batch inference. In contrast, recent approaches including Deja Vu [28] and Neuralink [50] exploit activation sparsity and neuron co-activation to enable fine-grained parameter loading. By transferring only contextually relevant parameters, they significantly reduce I/O overhead and accelerate inference on memory-constrained GPUs.

Hybrid Inference. Heterogeneous frameworks leverage both GPUs and CPUs for inference. Modern CPUs support high-performance instruction sets (e.g., AVX-512, ARM NEON) that accelerate matrix operations, providing a computational resource for LLM workloads. Widely used systems such as llama.cpp [14] adopt coarse-grained layer partitioning, statically assigning selected layers to the CPU for storage and computation. This layer-wise split placement introduces synchronization barriers and often leads to GPU "bubbles"—idle periods caused by the slower execution of the CPU and the strict sequential dependencies inherent in LLMs. Empirical results from llama.cpp show that CPU computation can still account for more than 90% of total inference latency. In contrast, emerging frameworks such as PowerInfer [41] adopt fine-grained, sparsity-aware offloading based on the power-law distribution of activations. By assigning frequently activated neurons to the GPU while placing infrequently activated neurons on the CPU, these systems align computational tasks with device strengths and achieve substantial improvements in inference efficiency.

3 Motivation

While state-of-the-art activation-sparse inference methods [41] have significantly improved efficiency over traditional coarse-grained approaches, they remain constrained by inherent hardware bottlenecks. This motivates the need for a more adaptive execution model under resource constraints.

3.1 Bottlenecks in Resource-Constrained Inference

We analyze representative fine-grained approaches and identify two dominant bottlenecks: I/O latency in GPU-only systems and excessive CPU overhead in heterogeneous systems.

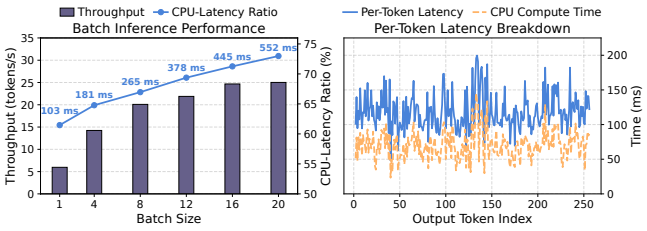


Figure 3: Illustration of the CPU bottleneck under static placement in heterogeneous systems. Left: throughput and CPU computation time under increasing batch size. Right: TPOT and CPU computation time across a 256-token decoding sequence at batch size 1. Both experiments were conducted on an NVIDIA RTX 3080 Ti.

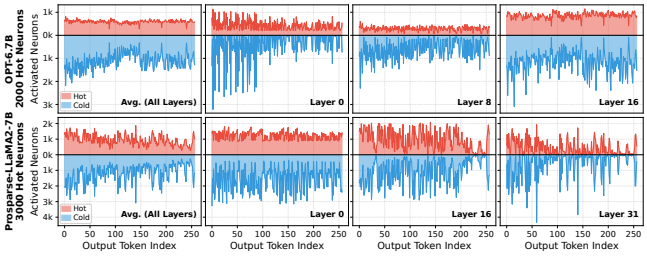


Figure 4: Examples of hot- and cold-neuron activation counts across output tokens under semantic drift, shown for OPT-6.7B and Prosparse-LLaMA2-7B at multiple layers.

Massive I/O Latency in GPU-Only Systems. In GPU-only systems with on-demand weight fetching (e.g., offloading FFN blocks to system memory), I/O latency becomes the primary bottleneck. Because computation must wait for weight transfers to complete, substantial computational bubbles form. Prior work shows that I/O can account for over 75% of generation time in sparsity-based systems such as LLM-Flash [2]. Although Neuralink improves access locality, the latency remains fundamentally constrained by physical bandwidth. As a result, prolonged I/O stalls severely underutilize the GPU, while modern CPUs remain largely idle, making this approach inefficient under resource constraints.

CPU Bottleneck of Static Weight Partitioning Systems. To improve hardware utilization, modern heterogeneous systems (e.g., PowerInfer [41]) statically assign neurons to either the CPU or GPU. The GPU executes frequently activated hot neurons, while the CPU handles rarely activated cold neurons. Although this eliminates runtime weight loading and reduces I/O overhead, it assumes that the hot–cold neuron split remains stable during inference. In practice, activation patterns drift, causing the workload to shift toward the slower CPU and resulting in significant performance degradation.

1) Workload Expansion in Batch Inference: In batch scenarios, concurrent prompts scale the workload linearly, overwhelming the CPU's limited parallelism. Crucially, while the CPU saturates, the GPU retains ample parallel headroom that could absorb this excess load, yet static placement fails to utilize it. When batch size grows from 1 to 20, throughput rises only modestly, while CPU latency increases sharply

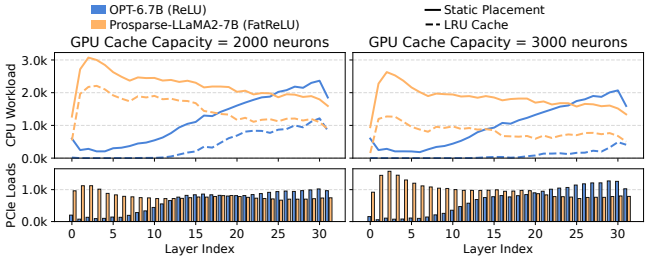


Figure 5: CPU workload and PCIe loads under static placement and an LRU neuron-balancing policy, shown for two sparse models under GPU cache capacities of 2,000 and 3,000 neurons. Solid lines denote static placement, and dashed lines denote the LRU policy.

from 102.8 ms to 551.7 ms. This static allocation turns the CPU into a dominant bottleneck, diminishing sparsity benefits and severely impacting tasks like speculative decoding that demand fast batch verification.

2) Activation Fluctuation in Generation: Activation sparsity also fluctuates substantially during token-by-token generation. Small shifts in semantic context can increase activations among cold neurons on the CPU, forcing the CPU to handle a larger portion of the computation. We quantify the semantic drift of two representative single-prompt decoding runs from two models in Figure 4. Under static placement, cold neuron activations fluctuate substantially per token (e.g., cold activations shift between adjacent tokens from nearly zero to over 3,000 in Layer 0 of OPT-6.7B). Across these representative runs, cold-neuron activations repeatedly form bursty intervals rather than isolated spikes, confirming that semantic drift is a persistent runtime phenomenon even at batch size 1. Although fluctuations also occur on the GPU side, its superior parallelism absorbs them with negligible overhead. In contrast, the same fluctuations saturate CPU workload due to its limited parallel compute capacity. As shown in Figure 3(right), TPOT varies from roughly 50 ms to nearly 200 ms within a single 256-token sequence, and CPU compute time ranges from around 30 ms to nearly 150 ms. These fluctuations persist throughout the sequence, revealing static placement's inability to follow dynamic activation patterns and resulting in unstable latency and degraded user experience.

3.2 A Strawman for Online Neuron Balancing

Both failures stem from the same root cause: static neuron placement. Once runtime activation patterns diverge from the offline profile, many computations are incorrectly routed to the CPU in most cases, creating unnecessary workload imbalance. A natural remedy is to migrate these computations back to the GPU. Our preliminary results (shown in Figure 5) show that even a simple LRU-style reload policy—loading activated neurons from the CPU to the GPU and evicting the least recently used ones—substantially reduces CPU workload with only modest PCIe overhead. For example, provisioning a small GPU cache of just 2,000–3,000 neurons yields

a notable decrease in CPU computation while keeping data transfers manageable. These observations demonstrate that limited, targeted neuron reloading can effectively trade I/O for reduced CPU computation, motivating a dynamic hybrid inference that we term online neuron balancing.

To make this idea concrete, we present a strawman system—a minimal yet illustrative design that operationalizes online neuron balancing. The strawman integrates frequency-aware static placement with simple runtime I/O control. During initialization, offline profiling identifies frequently activated neurons to pre-place on the GPU. During inference, an on-demand balancing policy detects activated cold neurons and migrates as many as the GPU cache can store, using a lightweight eviction policy (e.g., LRU). As shown in Figure 5, dynamically balancing even fewer than 1,000 neurons—despite each layer containing 11,008 neurons in Prosparse-LLaMA2-7B or 16,384 in OPT-6.7B—can substantially reduce CPU workload and stabilize execution efficiency. This strawman serves as a conceptual baseline that illustrates both the feasibility and the benefits of dynamic neuron placement.

Although this strawman theoretically overcomes the limitations of static placement by adapting to activation changes, it also reveals how the three high-level challenges of online neuron balancing manifest concretely in practice:

Challenge 1: How to achieve efficient balancing with computation overlap. Simply redistributing neurons at runtime introduces substantial transfer latency, but the strict sequential dependencies in LLM inference leave limited opportunity to hide this latency. In particular, the sparsity predictor depends on the output of the attention block to determine the activation pattern for the subsequent FFN block. As a result, the system must wait for transfers to complete after prediction, placing non-overlapped I/O directly on the pipeline. Meanwhile, integrating this dynamic balancing into a heterogeneous system—where CPU and GPU already execute concurrently—further increases pipeline complexity.

Challenge 2: How to prioritize sustained neurons and avoid transient, wasteful balancing. As shown in our analysis of activation fluctuation, activation patterns can shift rapidly within short context windows. A simple balancing policy risks repeatedly transferring cold neurons that are activated only momentarily ("one-hit wonders"), resulting in wasted I/O: scarce bandwidth is spent on neurons with minimal long-term utility, potentially evicting more valuable neurons already resident in the GPU cache.

Challenge 3: How to adaptively control balancing intensity under runtime resource constraints. I/O latency and CPU computation present fundamentally opposing bottlenecks: reducing CPU workload by balancing more neurons to the GPU increases I/O cost, while minimizing I/O leaves more computation to the slower CPU. Further, due to continuous fluctuations in resource availability and the dynamic I/O demand induced by the balancing policy, the optimal operating point shifts under different runtime conditions.

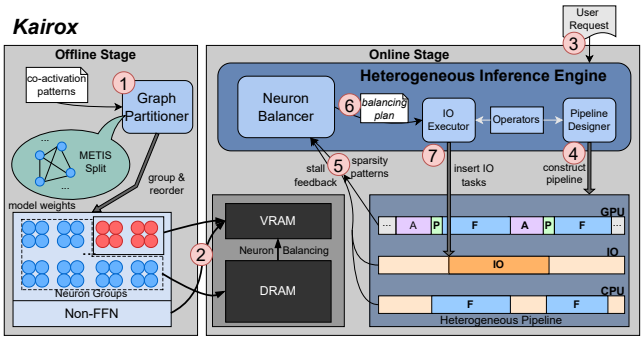


Figure 6: System overview of KAIROX.

Table 1: Summary of key notations and hyperparameters.

Symbol	Description
	Notations
n	Neuron index
g	Neuron group index
N	Total number of neurons per layer
M	Total number of neuron groups
GS	Group size (number of neurons per group)
w_{ij}	Co-activation weight between neurons i and j
$G=(V,E)$	Neuron co-activation graph
t	Generation step (token index)
$A_t^{(n)}, A_t^{(g)}$	Activation state of neuron n and normalized activation intensity of group g at step t
$S_t^{(n)}, S_t^{(g)}$	TAM momentum score of neuron n and group g at step t
	Hyperparameters
λ	TAM decay factor controlling balancing inertia
$\lambda_{\min}, \lambda_{\max}$	Lower and upper bounds of the decay factor
α	Adjustment rate for λ per decoding step
ε	Discrimination margin for loading threshold
τ_{load}	Neuron group loading threshold, $(1-\lambda)+\varepsilon$
K	GPU cache budget (top-K groups retained)

4 System Overview

To address the aforementioned challenges, we propose KAIROX, an adaptive GPU–CPU hybrid inference system. As illustrated in Figure 6, the system workflow operates in two stages. In the Offline Stage, the Graph Partitioner profiles co-activation patterns to group and reorder neurons into optimized clusters (①); we detail this process in subsection 5.2. These neuron groups are then initially distributed between VRAM and DRAM (②) to establish an efficient memory layout for runtime access.

The Online Stage is triggered by a user request (③). Upon initialization, the Pipeline Designer constructs a heterogeneous pipeline (④), as described in section 5. As inference proceeds, the engine executes an iterative optimization cycle: the Neuron Balancer continuously analyzes sparsity patterns and pipeline stall feedback (⑤) to synthesize real-time balancing plans (⑥) using the policies in section 6 and section 7. Acting on these directives, the IO Executor inserts asynchronous I/O tasks (⑦), driving a continuous loop that adapts resource allocation and overlaps data transfer with computation for each subsequent token. Table 1 lists the notation and hyperparameters.

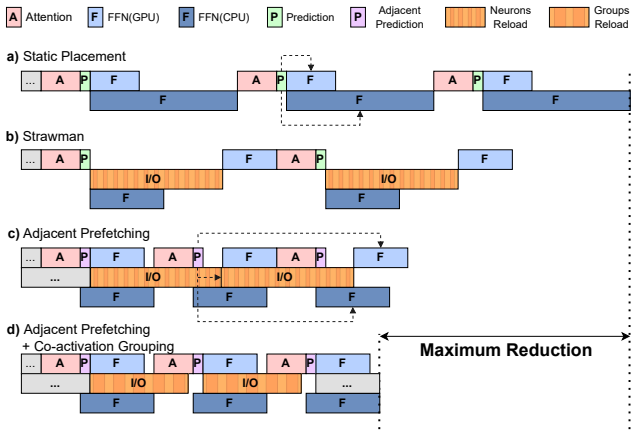


Figure 7: Comparison of heterogeneous pipelines for leveraging activation sparsity: (a) static placement; (b) the strawman design; (c) adjacent prediction; and (d) co-activation grouping & reordering.

5 Live Pipeline

We begin by constructing a basic heterogeneous pipeline similar to static neuron placement, where the CPU and GPU perform sparse computation concurrently. During online inference, the hybrid inference engine interprets computation tasks as a dependency graph and partitions them into backend-specific subgraphs. These dependencies, registered by the pipeline designer, form a Directed Acyclic Graph (DAG) that enables potential overlap across backends. A representative example is split-FFN execution: once the sparsity pattern is determined, the engine launches sparse kernels on both CPU and GPU to process their respective activated neurons, after which CPU outputs are merged back into the GPU, as illustrated in Figure 7(a).

In the strawman system, balancing-related I/O occurs strictly between sparsity prediction and FFN computation, as shown in Figure 7(b). This sequential dependency creates a substantial pipeline bubble, stalling computation on both CPU and GPU. To reduce this exposed I/O latency, we optimize balancing along two complementary dimensions: (1) hiding transfer latency via adjacent prediction and prefetching, and (2) improving bandwidth utilization through neuron co-activation grouping and reordering for transfer efficiency.

5.1 Adjacent Prediction

To hide I/O latency, KAIROX introduces adjacent prediction. This design is motivated by the empirical observation that intermediate hidden states in consecutive layers exhibit high cosine similarity, a stability that arises from the residual connections in Transformer blocks. Experiments show that the cosine similarity between adjacent layers reaches approximately 98%, indicating that the previous layer's hidden state can reliably predict the next layer's activation pattern with minimal accuracy loss. As illustrated in Figure 7(c), the predictor for layer i+1 uses the output of the attention block at layer i to forecast the activation sparsity pattern of the FFN block at layer i+1. This adjacent prediction creates a time window—spanning FFN layer i and the attention block of layer i+1—during which the I/O transfer for layer i+1 can be initiated without introducing stalls. We adopt a one-layer lookahead rather than a multi-layer lookahead because the computation and transfer durations for an individual layer are comparable. A multi-layer lookahead would yield redundant overlapping windows and significant accuracy degradation as similarity drops over larger layer gaps (e.g., 85% across two layers and 65% across four), while also introducing unnecessary complexity in boundary case management, as mentioned in the next paragraph. Therefore, single-layer adjacent prediction optimally balances overlapping efficiency with predictor accuracy, significantly reducing exposed I/O latency.

A special case arises at the first layer (Layer 0), which lacks a preceding layer for adjacent prediction. To avoid a pipeline stall at the beginning of each decoding step, we preload the required neurons during the CPU-side token sampling phase. This design leverages Activation Locality (section 6), which indicates that activation patterns remain stable across adjacent tokens. While processing the final layer for token t, we use the predicted activation pattern of Layer 0 for token t to identify and prefetch the neurons needed for Layer 0 of token t+1. By overlapping this prefetching with the final FFN computation and sampling of the current token, we eliminate the otherwise unavoidable stall at the start of the next decoding step.

A further advantage of adjacent prediction is its ability to overlap the prediction stage with CPU computation. In Figure 7(a), sparsity prediction is strictly serialized before sparse FFN execution and accounts for roughly 10% of total GPU computation time. In Figure 7(c), however, this overhead is hidden behind concurrent CPU workloads, rendering its contribution to end-to-end latency negligible.

Table 2: PCIe bandwidth (GB/s) under different hidden dimensions and neuron group sizes. The PCIe 3.0 and PCIe 4.0 results correspond to models with hidden sizes of 4096 (typical for 7B model) and 5120 (typical for 13B model), respectively.

GS	1	4	8	16	24	32	Theo.
PCIe 3.0	3.3	6.8	8.7	10.2	10.8	11.1	16
PCIe 4.0	5.8	13.7	17.8	20.9	22.3	22.8	32

5.2 Co-activation Grouping & Reordering

In LLM architectures, a single neuron occupies only a small memory footprint (e.g., $\approx$ 8 KB for a hidden size of 4096). Prior studies show that transferring weights at such fine granularity leads to severe underutilization of PCIe bandwidth due to small, fragmented reads and a large number of I/O operations. As shown in Table 2, single-neuron transfers achieve only 21% and 18% of theoretical peak bandwidth on PCIe 3.0 and PCIe 4.0, respectively. To overcome this limitation, we adopt a graph-partitioning strategy that clusters neurons with

high co-activation probability.

Problem Formulation. We formally model the neuron-grouping task as a weighted graph partitioning problem with a capacity constraint. Let $G = (V, E)$ be a dense undirected graph where V denotes a set of N neurons. Each edge (i, j) is associated with a weight $w_{ij} \geq 0$, representing the co-activation frequency (or correlation strength) between neurons i and j. Given a fixed group size GS (with GS dividing N), the objective is to partition the neurons into $M = N/GS$ disjoint groups of size GS. The goal is to maximize the total intra-group edge weight, ensuring that neurons frequently activated together are collocated and thus transferred as a unit. For descriptive purposes, we express this objective as the following binary linear programming for the grouping problem:

$$x_{ig} = \begin{cases} 1, & \text{if neuron } i \text{ is assigned to group } g, \\ 0, & \text{otherwise,} \end{cases} \tag{1}$$

$$\max \quad \sum_{g=1}^{M} \sum_{i<j} w_{ij} x_{ig} x_{jg} \tag{2}$$

$$\text{s.t.} \quad \sum_{g=1}^{M} x_{ig} = 1, \quad \forall i \in V, \tag{3}$$

$$\sum_{i=1}^{N} x_{ig} = GS, \quad \forall g \in \{1, \ldots, M\}. \tag{4}$$

This formulation corresponds to a variant of the graph partitioning problem, which is known to be NP-hard. Consequently, in the offline phase, we collect activation statistics to construct the weight matrix W and employ the METIS graph-partitioning tool [16] to efficiently approximate the optimal grouping strategy. In practice, this end-to-end grouping and reordering requires less than 10 minutes for a single layer.

Granularity Trade-offs. The group size GS serves as a critical hyperparameter. Larger groups improve bandwidth saturation by enabling more sequential PCIe transfers, but they introduce over-fetching because an entire group must be loaded even when only a subset of its neurons is activated. Conversely, smaller groups provide higher data efficiency but reduce PCIe utilization. We analyze the choice of GS in section 8. Furthermore, the mismatch between neuron-level sparsity prediction and coarse-grained group-level transfers necessitates a group-granularity scoring policy (section 6) to determine which groups to load. This optimized grouping strategy effectively reduces I/O latency (see Figure 7(d)) and mitigates pipeline stalls.

6 Locality-Aware Neuron Balancing

The efficacy of dynamic neuron balancing depends on a policy that minimizes thrashing while navigating the trade-off between I/O latency and CPU computation.

To design an effective balancing policy, we first conduct an empirical analysis of neuron activation traces. Although our system ultimately operates at the group level (as described in subsection 5.2), we present the analysis at the neuron granularity for clarity. We collected more than 400,000 activation sequences from multiple models across diverse datasets, with each sequence comprising 256 generation steps.

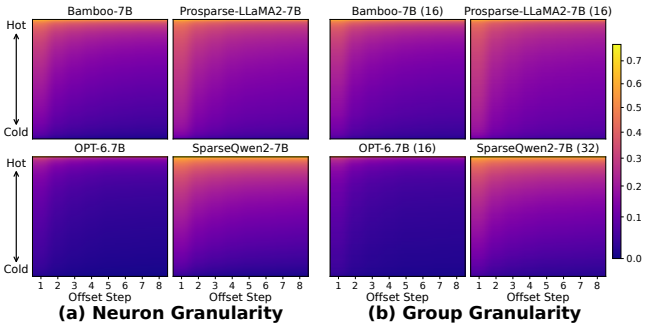


Figure 8: Heatmap of conditional reactivation probability for neuron and group granularity: (a) neuron granularity and (b) group granularity. The x-axis denotes the decoding-step offset δ (1 to 8 steps). The y-axis shows neuron or group indices sorted by activation frequency (hot to cold). The color intensity visualizes the conditional reactivation probability.

We formally define the event $A_t^{(n)}$ as the activation of neuron n at generation step t. To characterize the temporal recurrence of activations, we compute the conditional probability $P\left(A_{t+\delta}^{(n)} \mid A_t^{(n)}\right)$, which measures the likelihood that neuron n is reactivated at a temporal offset of δ steps after activation at step t. Figure 8 presents a heatmap visualizing this probability distribution of four representative sparse models. The horizontal axis denotes the time lag δ from 1 to 8, while the vertical axis lists neuron indices sorted by their global activation frequency in descending order.

Our analysis reveals an empirical observation of activation dynamics: a currently active neuron exhibits a high probability of being reactivated within the next few steps. This indicates that activations form continuous temporal clusters rather than isolated random events. These findings demonstrate the principle of Activation Locality: once a neuron is activated, it is highly likely to remain active for a subsequent sequence of tokens before becoming inactive again. We refer to this duration as the Activation Window. The window length varies substantially; although reactivation probability generally decreases with increasing δ, the decay rate is inversely correlated with global activation frequency. In particular, "cold" neurons (the bottom 70%) exhibit rapid decay, with their reactivation probability falling below 0.2 when $\delta > 2$.

Standard cache-inspired policies fail to adapt to these Activation Locality patterns. Least Recently Used (LRU) is overly sensitive to transient noise. As shown in our analysis, "cold" neurons often manifest as "one-hit wonders" with extremely short activation windows. Under LRU, a single incidental activation can promote a cold neuron to the most recently used position, potentially evicting a hot neuron that is only briefly

silent, leading to pollution and thrashing. Conversely, Least Frequently Used (LFU) suffers from history lag: it aggregates counts across the entire generation history, making it slow to respond to semantic transitions. When the context shifts, LFU tends to retain neurons associated with the previous context, failing to track the emergence of newly important neurons.

To reconcile long-term activation frequency with short-term recency effects, we introduce *Temporal Activation Momentum (TAM)*, a lightweight metric that quantifies the instantaneous "temperature" of a neuron. Inspired by the Hawkes process [23] in temporal point process modeling, TAM treats neuron importance not as a static count but as a time-decaying intensity function. Formally, the momentum score $S_t^{(n)}$ of neuron n at step t is updated recursively as follows:

$$S_{t+1}^{(n)} = \lambda \cdot S_t^{(n)} + (1-\lambda) \cdot A_t^{(n)}, \qquad \lambda \in [0,1), \tag{5}$$

TAM captures the Activation Window pattern observed in our analysis: when a neuron activates persistently, $S^{(n)}$ increases rapidly, raising its balancing priority and promoting its retention on the GPU. When the semantic context shifts and the neuron becomes inactive, $S^{(n)}$ decays exponentially, allowing the system to phase out outdated neurons smoothly.

However, directly relying on TAM may inadvertently promote "one-hit wonders"—cold neurons that activate once in isolation without forming a sustained Activation Window. To avoid unnecessary I/O, we impose a threshold constraint and, at the neuron level, treat only those neurons whose momentum scores exceed $\tau_{\text{load}} = (1-\lambda)+\varepsilon$ as balancing candidates. The term $1-\lambda$ corresponds to the theoretical momentum acquired by a cold neuron ($S_t^{(n)} \approx 0$) after a single isolated activation; thus, scores near this value typically indicate one-hit wonders. The hyperparameter ε provides an additional discrimination margin. By setting τ_{load} slightly above this baseline, isolated spikes are prevented from triggering balancing actions. The same thresholding principle is later applied to group-level scores in lines 10–15 of Algorithm 1.

Figure 8(b) shows that the principle of Activation Locality extends naturally to the co-activation level. To align with our grouping strategy (subsection 5.2), we generalize TAM from individual neurons to neuron groups by replacing the binary activation state $A_t^{(n)}$ with a normalized intra-group activation intensity $A_t^{(g)}$. The corresponding group score $S_t^{(g)}$ provides a unified scoring function for memory management. In practice, the system retains the top-K scoring groups in the GPU, where K is determined by the available GPU memory budget.

7 Adaptive Neuron Balancer

To address Challenge 3—the coupled bottleneck arising from the inverse relationship between I/O latency and CPU workload—we observe that static balancing policies are fundamentally insufficient. An aggressive policy reduces CPU workload but risks saturating PCIe bandwidth (I/O-bound), while a conservative policy alleviates I/O pressure but leaves substantial computation to the CPU (CPU-bound). Because the optimal balance varies with hardware characteristics and runtime I/O load, it cannot be predetermined.

Algorithm 1 Adaptive online neuron balancing.

Require: Group scores $\{S_t^{(g)}\}$, neuron activations $\{A_t^{(n)}\}$, current decay λ, adjustment rate α, margin ε, budget K, cache $\mathcal{C}_t$
Ensure: Updated $\mathcal{C}_{t+1}$, $\{S_{t+1}^{(g)}\}$, λ
Phase 1: Adaptive Balancing-Intensity Control
1: $Feedback \leftarrow$ GETSYSTEMBOTTLENECK()
2: **if** *Feedback* is `IO_BOUND` **then**
3: $\quad\lambda \leftarrow \min(\lambda \cdot (1+\alpha), \lambda_{\max})$ ▷ Increase inertia
4: **else if** *Feedback* is `CPU_BOUND` **then**
5: $\quad\lambda \leftarrow \max(\lambda \cdot (1-\alpha), \lambda_{\min})$ ▷ Decrease inertia
6: **end if**
Phase 2: Momentum Update & Group Filtering
7: $A_t^{(g)} \leftarrow$ ACTIVATIONGROUPING($\{A_t^{(n)}\}$) ▷ Map & Normalize
8: $\tau_{\text{load}} \leftarrow (1-\lambda)+\varepsilon$
9: $\mathcal{C}_{\text{cand}} \leftarrow \emptyset$
10: **for** each neuron group g **do**
11: $\quad S_{t+1}^{(g)} \leftarrow \lambda \cdot S_t^{(g)} + (1-\lambda) \cdot A_t^{(g)}$ ▷ TAM update
12: $\quad$ **if** $S_{t+1}^{(g)} > \tau_{\text{load}}$ **then**
13: $\quad\quad\mathcal{C}_{\text{cand}} \leftarrow \mathcal{C}_{\text{cand}} \cup \{g\}$ ▷ Filter "One-Hit Wonders"
14: $\quad$ **end if**
15: **end for**
16: $\mathcal{C}_{t+1} \leftarrow$ TOPK($\mathcal{C}_{\text{cand}}, K$) based on scores $S_{t+1}^{(g)}$
17: $\mathcal{G}_{\text{load}} \leftarrow \mathcal{C}_{t+1} \setminus \mathcal{C}_t$ ▷ Groups selected for loading
18: $\mathcal{G}_{\text{evict}} \leftarrow \mathcal{C}_t \setminus \mathcal{C}_{t+1}$ ▷ Groups selected for eviction
19: ISSUEASYNCIO($\mathcal{G}_{\text{evict}}, \mathcal{G}_{\text{load}}$)
20: **return** $\mathcal{C}_{t+1}, \{S_{t+1}^{(g)}\}, \lambda$

To overcome this limitation, we extend TAM into a feedback-driven balancing policy by incorporating a runtime control loop. In this design, the decay factor λ introduced in section 6 is no longer a fixed hyperparameter, but a dynamic control variable that modulates the system's balancing intensity according to real-time resource conditions.

In our momentum formulation, λ governs the inertia of the score updates and, consequently, the turnover rate of the GPU-resident neuron set. A high λ introduces strong inertia by emphasizing historical activations, making it difficult for cold neurons to accumulate sufficient momentum to exceed the loading threshold. This yields a conservative balancing policy that suppresses reload intensity. In contrast, a low λ reduces inertia, allowing momentum scores to respond quickly to recent activations. This heightened sensitivity increases the mismatch between currently high-scoring groups and the GPU-resident set, triggering more aggressive balancing actions that reduce CPU workload at the cost of increased I/O overhead. During inference, the hybrid inference engine monitors pipeline stalls to identify the dominant bottleneck. We incorporate a negative feedback loop (formalized in Phase 1 of Algorithm 1) to dynamically adjust the value of λ:

- **I/O-Bound:** If stalls are primarily caused by I/O delays,

the system detects excessive data-transfer pressure. In response, it increases λ by a factor of $(1+\alpha)$, strengthening momentum inertia and enforcing a more conservative balancing policy to reduce overall I/O load.

- **CPU-Bound:** if stalls originate from CPU computation, the system infers that the CPU is overloaded. It then decreases λ by a factor of $(1-\alpha)$, assigning greater weight to recent activations relative to historical momentum. This increases balancing intensity, reducing CPU workload by exploiting available I/O overlap.

We initialize λ to a moderate value (e.g., 0.5). Within the first few generation steps, the feedback-driven mechanism quickly converges λ toward—or maintains it near—an equilibrium that balances the two coupled bottlenecks. Because a static optimal configuration rarely exists under varying input contexts, this dynamic tuning enables KAIROX to adapt continuously to runtime fluctuations, including semantic shifts and changes in available hardware resources.

The complete balancing policy is formalized in Algorithm 1. It integrates Feedback-Driven Tuning and Temporal Activation Momentum within each layer's decode step once activation sparsity is predicted, automatically reconciling hardware constraints with Activation Locality.

The resulting balancing decisions are efficient ($O(1)$) and integrate seamlessly into the runtime system. We implement the updates with customized CUDA kernels (section 8), keeping end-to-end overhead negligible.

8 Implementation

We implemented KAIROX on top of `llama.cpp`, adding approximately 5,700 lines of C++ and CUDA code. In the offline stage, we collected over 400,000 activation patterns from the C4 dataset [35]. These patterns were used to train the adjacent predictors and to construct an undirected weighted graph for neuron grouping and reordering, which was subsequently partitioned using the METIS [16] graph-partitioning tool.

Priority-Aware Stream Management. To coordinate the heterogeneous pipeline, KAIROX employs three primary CUDA streams: a computation stream within the inference engine for kernel executions, and two I/O streams managed by an internal I/O Executor—one critical I/O stream for essential transfers (e.g., merging FFN results) and one reload stream for neuron-loading tasks. An asynchronous thread in the I/O Executor maintains a task queue for reload operations. When a critical I/O task arrives, the executor preempts queued reload tasks, allowing essential transfers to bypass the queue and execute immediately, thereby preventing pipeline stalls and ensuring stable end-to-end latency.

Customized Kernels. We developed optimized sparse CUDA and AVX kernels for activation-sparse inference. The feedback-driven TAM policy uses custom CUDA kernels that fuse `XOR` and `AND` operations for efficient group-level activation updates and set-difference computations. We also integrate a CUDA `argsort` kernel to rank neuron groups by momentum score. Because high-performance argsort is most efficient on inputs below 1024 elements, we constrain the neuron group count M to this threshold, keeping balancing latency negligible while integrating seamlessly with the online neuron balancing pipeline.

9 Evaluation

9.1 Experiment Setup

Hardware. We conduct all experiments on two representative consumer-grade PC configurations.

- **PC-Low**: NVIDIA RTX 3080 Ti (12 GB, 136 TFLOPs), Intel Xeon® E5-2680 v4 with execution capped at 12 threads, 64 GB host memory, and PCIe 3.0 (16 GB/s).
- **PC-High**: NVIDIA RTX 4090 (24 GB, 330 TFLOPs), AMD EPYC 7542 with execution capped at 16 threads, 128 GB host memory, and PCIe 4.0 (32 GB/s).

Models and Workloads. We evaluate a range of sparse LLMs, including OPT-6.7B/13B, OPT-30B-Q8/Q4, and OPT-66B-Q4 (ReLU); Prosparse-LLaMA2-7B/13B (derived from LLaMA-2; FatReLU); Bamboo-7B (derived from Mistral; dReLU); SparseQwen2-7B (derived from Qwen2; dReLU); and ReLUFalcon-40B-Q8 (derived from Falcon; ReLU) [39, 42, 43, 47, 59]. The suffixes "Q8" and "Q4" denote INT8 and INT4 quantization, respectively. All models are tested using a fixed subset of real-world user prompts randomly sampled from ShareGPT [46], ensuring consistent, diverse, and reproducible workloads. Additionally, we reuse the dense GEMM kernels from llama.cpp for the prefill stage, as the dense kernels are better optimized for large-batch computation.

Baselines. We compare KAIROX with four representative systems: llama.cpp [14], PowerInfer [41], Neuralink [50], and Q-Infer [29]. Llama.cpp serves as a widely used local inference baseline. PowerInfer is a state-of-the-art sparsity-aware framework; because it relies on an older version of llama.cpp, we port it to the same updated backend used by KAIROX for a fair comparison. Neuralink is an on-device activation-sparse system that eagerly loads all activated neurons onto the accelerators for computation. Because it is designed for smartphones and is not open source, we reimplement its core mechanism on top of KAIROX for evaluation. Q-Infer is a GPU–CPU hybrid inference system designed for high-end GPUs and large-batch LLM serving. Since it also uses runtime reloading, we evaluate it under the same setting as KAIROX for a direct comparison under identical constraints.

9.2 End-to-End Performance

We first evaluate throughput using a generation batch size of 1, reflecting a typical usage scenario for local LLM deployments. For consistency, we cap the maximum output length at 512 tokens and the maximum context size at 1024 tokens.

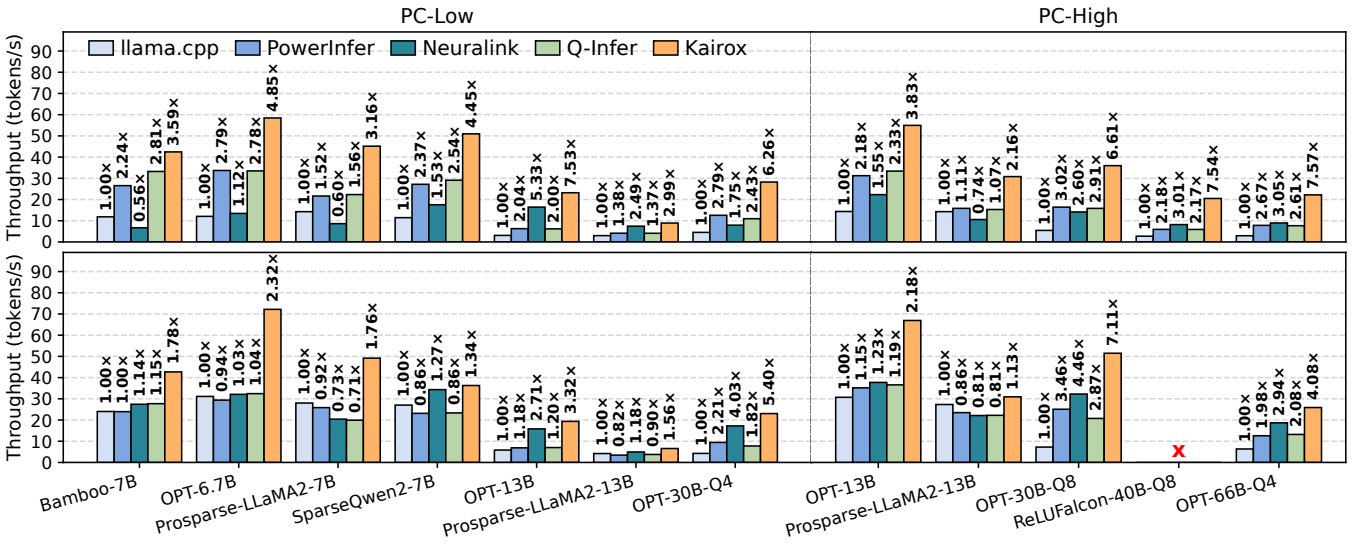


Figure 9: End-to-end performance of decoding. Top: standard decoding. Bottom: speculative decoding.

As shown in Figure 9, KAIROX consistently outperforms all baselines across models and hardware configurations. For standard completion, the speedup over llama.cpp reaches up to 7.53× on PC-Low and 7.57× on PC-High, with geomean gains of 4.45× and 5.00× across models, respectively. Relative to the sparse baselines, the geomean gains range from 2.08× to 3.06× on PC-Low and from 2.36× to 2.58× on PC-High, while the corresponding best-case gains reach 3.70×, 6.35×, and 3.76× on PC-Low and 3.46×, 2.91×, and 3.48× on PC-High over PowerInfer, Neuralink, and Q-Infer, respectively. The gains are largest on OPT models, whose stronger sparsity and shorter cold-neuron activation windows—reflected in the darker regions of Figure 8—allow TAM to filter transient one-hit-wonder activations more effectively. As a result, the GPU cache is reserved for neurons with genuinely long activation windows. Neuralink's relative performance ranges from near parity to multiple-fold gains because it is sensitive to cache hit rate, per-step activation volume, and GPU cache size. When the cache is large, it reloads a larger portion of activated neurons to make full use of the cache, introducing substantial I/O stalls (e.g., Bamboo-7B on PC-Low and Prosparse-LLaMA2-13B on PC-High). Q-Infer, despite using rebalancing, does not outperform PowerInfer in our setting. Because it is designed for large-batch serving, it focuses on current-round activations and may load short-lived cold neurons while missing persistently activated ones, leaving the system CPU-bound with extra scheduling overhead. By contrast, KAIROX leverages the activation locality property described in section 6 to ensure that high-value neuron groups consistently dominate the GPU cache, effectively reducing CPU workload and wasteful transfers.

To evaluate throughput scalability at higher batch sizes, we test Speculative Decoding—a widely used edge-acceleration technique in which a draft model proposes candidate tokens that are verified in parallel. All evaluations use 5 draft tokens, and ReLUFalcon-40B-Q8 is excluded because no compatible draft model is available. While we faithfully report cases where low acceptance rates reduce absolute gains, our analysis focuses on relative batch-processing efficiency against baseline systems. Under this setting, KAIROX continues to outperform all baselines, achieving geomean speedups of 2.23× and 2.91× over llama.cpp on PC-Low and PC-High, respectively. Relative to the sparse baselines, the geomean gains range from 1.53× to 2.11× on PC-Low and from 1.53× to 1.88× on PC-High. In this setting, llama.cpp occasionally outperforms PowerInfer and Neuralink because dense batched GEMMs can surpass sparse kernels when irregular memory access and index overhead outweigh sparsity benefits. Larger batch sizes also increase CPU workload, exposing bottlenecks in static systems. Neuralink can outperform other reloading-based baselines in some cases because its aggressive reloading absorbs part of this additional CPU workload. By contrast, Q-Infer still underperforms PowerInfer in most cases, as speculative-decoding batch sizes remain too small for its policy to reliably identify and retain high-value neurons. Nevertheless, KAIROX remains consistently superior; for example, on PC-Low with OPT-30B-Q4, it achieves speedups of 5.40×, 2.44×, 1.34×, and 2.97× over llama.cpp, PowerInfer, Neuralink, and Q-Infer, respectively.

9.3 Performance Analysis

We further examine the factors contributing to KAIROX's performance advantages over baseline systems.

Effectiveness of Adaptive Neuron Balancer. First, we compare CPU computation burden (measured by CPU latency), I/O pressure (measured by reload latency), and overall throughput under three balancing policies: a simple **Top-K**

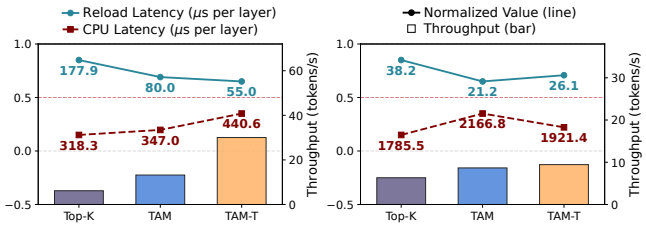


Figure 10: Comparison of CPU latency, I/O latency, and throughput under three reload policies. Left: Bamboo-7B on PC-Low. Right: Prosparse-LLaMA2-13B on PC-Low. CPU latency and reload overhead are normalized.

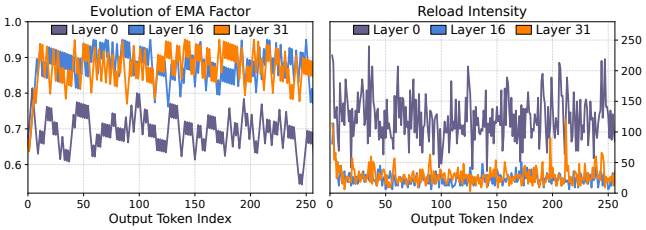


Figure 11: Evolution of the decay factor λ and the corresponding reload intensity, sampled from Bamboo-7B on PC-Low. Left: decay factor λ. Right: number of reload I/O tasks per step.

strategy that reloads the most activated neuron groups, a **TAM** scoring policy with a fixed λ, and **TAM-T**, i.e., TAM with feedback-driven tuning, as shown in Figure 10.

Under the Top-K policy, large numbers of cold neuron groups—isolated "one-hit wonders"—are indiscriminately transferred to the GPU. Although this approach nominally reduces CPU computation, the excessive reload latency induces severe stalls that negate any potential gains. Introducing a static TAM policy significantly filters out such one-hit wonders, reducing reload latency—the primary bottleneck—by approximately 1.8× to 2.2× and substantially improving throughput. Finally, the TAM-T policy dynamically navigates the inverse trade-off between I/O latency and CPU computation. In I/O-bound scenarios (e.g., Bamboo-7B on the PC-Low), the policy detects I/O-induced stalls and enforces a more conservative reloading intensity. Conversely, in CPU-bound scenarios (e.g., Prosparse-LLaMA2-13B on the PC-Low), it increases reloading intensity to alleviate CPU workload, thereby optimizing overall throughput.

Second, Figure 11 shows how the evolution of the decay factor λ regulates reloading intensity via the decoding process. Immediately after initialization, the balancer identifies the dominant bottleneck and converges toward a stable equilibrium. For example, during steps 1–7 at layer 31, λ increases monotonically, indicating an I/O-bound state that necessitates a conservative balancing. This adjustment rapidly reduces I/O volume and restores system harmony. Throughout generation, λ continues to adapt dynamically to real-time workload fluctuations. Distinct micro-oscillations emerge around specific thresholds—most notably during steps 65–85 at layer 0 and steps 130–150 at layer 16 & 31—signaling that the system has

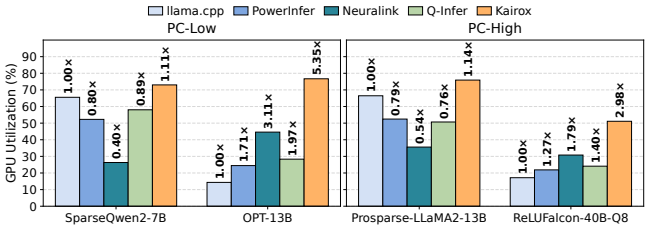


Figure 12: GPU utilization across baselines and KAIROX for representative models on PC-Low and PC-High.

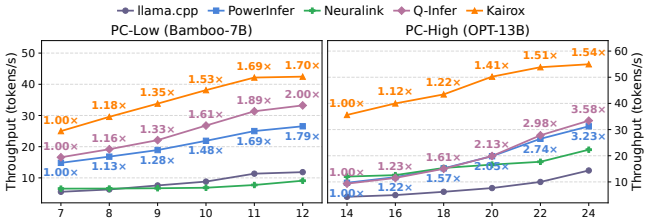


Figure 13: Throughput under varying GPU memory budgets. Left: PC-Low with Bamboo-7B. Right: PC-High with OPT-13B.

located a temporary optimal point where neuron transfers and CPU computation are effectively balanced. At layers 16 & 31, λ stabilizes around 0.85, reflecting a conservative balancing strategy. As shown in Figure 11(right), this corresponds to consistently low I/O intensity, remaining below 75 reloads per step. In contrast, the lower λ value observed at layer 0 triggers a more aggressive reloading policy, resulting in substantially higher I/O volume per decoding step.

GPU Utilization. We use NVIDIA Nsight Systems [31] to measure GPU utilization, a key indicator of pipeline efficiency and hardware stalls. As shown in Figure 12, KAIROX consistently outperforms all baselines, achieving up to 5.35× and 2.98× higher utilization on PC-Low and PC-High, respectively. This improvement reflects a more efficient execution pipeline in which the GPU remains active rather than idling. In contrast, llama.cpp exhibits pronounced utilization fluctuations determined by the degree of layer offloading. When most layers are offloaded to the GPU (e.g., Prosparse-LLaMA2-13B on PC-High), consistent kernel execution raises utilization to 68%; when memory constraints limit offloading (e.g., OPT-13B on PC-Low), utilization falls below 20%. Although activation-sparse inference frameworks generally outperform dense baselines in such settings, KAIROX still surpasses PowerInfer. This advantage stems from adaptive neuron balancing and a highly optimized pipeline that dynamically rebalances computation while overlapping CPU execution and I/O with GPU kernels to reduce stalls. GPU utilization is ultimately limited by residual non-overlappable I/O, cross-device synchronization, and system-level overheads such as CPU scheduling and token processing. In resource-constrained edge environments, these factors cap practical utilization at around 70% in our settings, which is already near saturation.

Performance under Various GPU Memory Budgets. Figure 13 shows that KAIROX consistently outperforms all base-

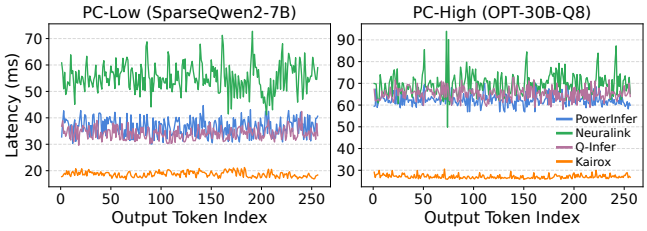


Figure 14: Per-token TPOT latency across models and systems. Left: SparseQwen2-7B on PC-Low. Right: OPT-30B-Q8 on PC-High.

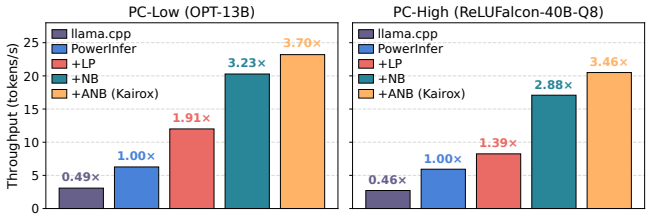


Figure 15: Contribution of each component in KAIROX, shown as throughput across incremental system variants.

lines across configurations, demonstrating strong resilience under constrained GPU memory budgets. As available GPU memory shrinks, static strategies leave more workload to the CPU, creating severe performance bottlenecks. Under these tighter budgets, KAIROX maintains a throughput of around 25 tokens/s on PC-Low with only 7 GB of VRAM when running Bamboo-7B (total memory footprint of approximately 14 GB). On PC-High, KAIROX under a 14 GB budget even surpasses all baselines operating at their full 24 GB budget, highlighting the effectiveness of its adaptive balancing under tight memory constraints. Q-Infer outperforms PowerInfer on Bamboo-7B but yields only marginal gains on OPT-13B, as its reloading policy struggles to identify high-value neurons in models such as OPT that exhibit a higher proportion of one-hit-wonder activations, as shown in Figure 8.

TPOT Latency Variability. We further evaluate per-token latency variability on a representative decoding sequence, as shown in Figure 14. Neuralink exhibits the most severe fluctuations: its TPOT trace closely follows the cold activation drift in Figure 4 because its aggressive reloading policy transfers all activated neurons without filtering, causing I/O latency to rise directly with cold activation volume. PowerInfer, as a static placement system, avoids these I/O-induced spikes but still suffers from CPU-side latency fluctuations as cold-neuron workload shifts across tokens. Q-Infer halts reloading once computation is required, avoiding the I/O-induced spikes seen in Neuralink. However, because it does not prioritize persistently activated neurons, high-value neurons may not be loaded before the I/O window closes, leaving the system CPU-bound with TPOT variability comparable to PowerInfer. KAIROX achieves the lowest and most stable TPOT across the sequence, demonstrating its robustness under dynamic activation patterns and different hardware configurations.

9.4 Ablation Study

We conduct an ablation study to quantify the contribution of each optimization component. Using llama.cpp as the primary baseline, we first observe that PowerInfer—which incorporates activation-sparse inference—achieves speedups of 1.43× and 1.61× under our evaluation settings. Building on this baseline, we incrementally integrate our dynamic optimizations in three stages and measure the resulting throughput improvements, as illustrated in Figure 15.

Table 3: End-to-end generation throughput (tokens/s) on non-ReLU architectures, comparing PowerInfer, Neuralink, Q-Infer and KAIROX. The suffixes "-Low" and "-High" indicate the hardware configurations used in each evaluation.

	Qwen2-7B (SiLU)-Low	LLaMA2-13B (SiLU)-Low	Falcon-40B (GeLU)-High
PowerInfer	16.46	3.97	9.86
Neuralink	12.95	3.02	7.83
Q-Infer	15.28	3.98	8.84
KAIROX	**21.96**	**5.07**	**10.53**

First, we apply the **Live Pipeline (+LP)** to maximize the overlap between I/O operations and heterogeneous execution. Second, we introduce online **Neuron Balancing (+NB)**, initially employing a fixed decay factor ($\lambda = 0.5$). Finally, we implement feedback-driven **Adaptive Neuron Balancing (+ANB)** to dynamically adjust λ in response to system constraints. The inclusion of LP yields throughput gains of 1.91× and 1.39× by effectively overlapping computation and communication. Notably, +NB contributes the largest performance gain, boosting throughput to 3.32× and 2.88×; this substantial leap underscores the critical importance of runtime dynamic neuron balancing in alleviating CPU bottlenecks. Ultimately, the constraint-aware tuning provided by ANB achieves peak speedups of 3.70× and 3.46×. These results demonstrate that while pipeline overlapping provides a solid foundation, dynamic neuron reloading acts as the primary driver of performance improvement, effectively unlocking the potential of hybrid inference.

9.5 Non-ReLU Models Performance

While activation-sparse inference methods typically achieve substantial acceleration on models with high activation sparsity, they often struggle with non-ReLU architectures (e.g., GeLU-based models). In non-ReLU models, a larger fraction of neurons are activated at each step due to reduced sparsity. Under static split strategies such as PowerInfer, this significantly increases the CPU workload. For Neuralink, the higher activation density triggers excessive indiscriminate reloads, rendering the system I/O-bound. Unlike competing systems, KAIROX retains its performance advantage even under high activation density. As shown in Table 3, KAIROX consistently outperforms all baseline systems across mainstream non-ReLU architectures. These results demonstrate the ro-

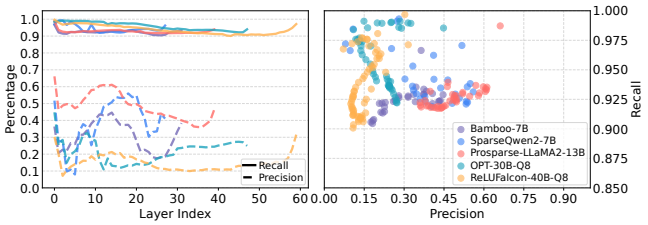


Figure 16: Predictor recall and precision across representative sparse models. Left: recall and precision across layers. Right: the corresponding precision–recall scatter plot for each predictor.

Table 4: Comparison of popular downstream task accuracy between the original models and their variants augmented with adjacent predictors (denoted as "model-P").

	PIQA	Wino	ARC	MMLU	GSM8K	Avg.
OPT-13B	76.22	66.69	32.76	24.58	1.67	40.38
OPT-13B-P	76.39	67.96	32.59	24.57	1.52	40.61
OPT-30B	77.69	69.61	34.22	25.67	1.82	41.80
OPT-30B-P	77.42	69.77	33.62	25.64	1.82	41.28
Prosparse-7B	75.90	67.96	27.39	42.47	14.40	45.62
Prosparse-7B-P	75.24	66.46	26.54	41.26	12.28	44.36
Prosparse-13B	76.06	69.93	34.47	50.10	31.01	52.31
Prosparse-13B-P	75.95	69.46	33.96	50.22	29.61	51.84
SparseQwen2	77.20	75.14	47.87	68.89	75.13	68.85
SparseQwen2-P	77.53	74.51	48.72	65.30	73.72	67.96

bustness of KAIROX and highlight potential opportunities for future research into inference optimization for less sparse or non-sparse model architectures on edge devices.

9.6 Sparse Inference Analysis

KAIROX employs lightweight predictors to skip computation for inactive neurons. Since only predicted neurons are computed, predictor accuracy directly affects downstream performance. In this section, we analyze predictor accuracy, its impact on the sparse inference system, and the underlying mechanisms behind this behavior.

Predictor Accuracy. Figure 16 illustrates predictor metrics across different layers. The recall ratio, defined as the fraction of ground-truth activated neurons that are correctly predicted, remains consistently high at 90%–99%. The precision ratio, defined as the fraction of predicted neurons that are confirmed as active in dense computation, is relatively lower. Specifically, the predictors are recall-prioritized: we deliberately sacrifice precision to maximize recall, reducing false negatives at the cost of additional false positives. This ensures that the vast majority of truly activated neurons are captured, preserving model performance.

Predictor Overhead. In our experiments, predictor execution accounts for approximately 5–10% of per-layer inference time when measured in isolation with NVIDIA Nsight Systems. However, the effective overhead introduced by the predictors is only 1–2% of end-to-end system latency, thanks to the dedicated Live Pipeline design described in section 5. Since the predictors execute on the GPU, the execution time of the predictor for layer $i+1$ is completely hidden behind the concurrent CPU-side MLP computation of layer i. The only unavoidable overhead occurs at the first layer, where the predictor and MLP computations must be serialized. As a result, predictor latency scales sub-linearly with the number of model layers, and its relative contribution diminishes in larger models. This overlap opportunity also allows us to use larger predictors for better accuracy than would be practical in systems where predictor latency is a critical bottleneck.

Model Downstream Performance. To evaluate the feasibility of KAIROX's sparse inference, we compare it against standard dense inference across several mainstream benchmarks, as shown in Table 4. The results indicate that KAIROX preserves model accuracy, with an average degradation of less than 0.5% across downstream tasks. This deviation falls within the expected statistical variance of the underlying models. Notably, our analysis shows that false negatives—neurons incorrectly classified as inactive—typically produce output magnitudes near zero. Consequently, omitting them has negligible impact on the layer's feature representation, maintaining the correctness of sparse inference.

10 Related Work

GPU–CPU Hybrid Inference. GPU–CPU hybrid inference has become a common strategy for running LLMs whose weights exceed GPU memory budgets. Representative systems such as llama.cpp [14] and PowerInfer [41] rely on static partitioning across the GPU and CPU, whereas more recent work explores runtime-adaptive reloading. Q-Infer [29], for example, is positioned toward server or high-end-GPU settings, where the goal is to maximize throughput under richer resources and larger-batch workloads. Accordingly, it focuses on current-round activations when deciding which CPU-resident weights to reload. This short-horizon policy can migrate transient "one-hit-wonder" cold neurons and incur extra transfers, but such overhead can often be amortized by the substantial computation available in large-batch serving. In contrast, KAIROX targets resource-constrained consumer PCs, where exposed transfer latency and CPU bottlenecks dominate once runtime activation drift invalidates static hot/cold placement. KAIROX therefore emphasizes not only which activated CPU-resident neurons should migrate back to the GPU, but also when such migration is worthwhile and how aggressively rebalancing should proceed under shifting CPU-bound and I/O-bound conditions.

Quantization. Quantization [8, 15] mitigates memory constraints by reducing model-weight precision to INT8 [9, 52], INT4 [13, 56], or even lower bit-widths [5, 49, 54] while maintaining acceptable accuracy. This technique is orthogonal to KAIROX, as we integrate INT8 and INT4 quantization support into KAIROX (section 9), enabling larger models to run

on limited hardware. Although quantization substantially reduces I/O volume by decreasing data-transfer size, it leaves the computational workload largely unchanged. Consequently, quantization often shifts the dominant system bottleneck from being I/O-bound to being computation-bound.

Mixture of Experts (MoE). MoE models introduce sparsity at the expert-selection level by routing each token to a small subset of FFN experts [4,61]. This differs from KAIROX, which targets fine-grained neuron-level sparsity inside FFN computation under GPU–CPU memory constraints. KAIROX does not directly replace MoE serving systems, and our implementation does not optimize expert routing or expert-level placement. However, the two forms of sparsity are complementary: once an expert is selected, its internal FFN computation can still exhibit activation sparsity, and KAIROX's neuron-level balancing could be applied within selected experts to reduce CPU-side work under limited GPU memory, presenting further opportunities for acceleration on edge devices [62]. Supporting full MoE inference would require integrating expert routing, expert-level placement, and neuron-level balancing, which we leave to future work.

Mobile Inference Systems. Deploying LLMs on mobile devices, such as smartphones, represents another important direction in edge computing. Unlike the consumer PC architecture targeted by KAIROX, mobile devices typically employ Systems-on-Chip (SoC) with Unified Memory (UM). UM enables the CPU and accelerators (e.g., NPUs, TPUs) to access a shared physical memory space, effectively eliminating explicit data-transfer overhead. Recent works such as llm.npu [53] and Smallthinker [44] exploit this property to optimize on-device inference.

LLM Serving Systems. LLM serving systems are typically deployed in cloud data centers and are designed to maximize throughput under high concurrency using abundant computational resources [30]. Research in this domain primarily focuses on efficient Key–Value (KV) cache management (e.g., vLLM [21], Mooncake [33], LMCache [7]) and distributed parallelism strategies [12, 19, 58]. LoRA-based serving systems such as CaraServe [24], S-LoRA [37], and TOPPINGS [25] represent another direction, focusing on efficient multi-adapter serving. However, these systems assume the base model fits entirely within GPU memory, with CPU involvement limited to lightweight adapter management. This makes them not directly applicable to the setting targeted by KAIROX, where the base model itself must be partitioned across the GPU and CPU under constrained GPU memory.

11 Conclusion

We presented KAIROX, an adaptive hybrid LLM inference system for consumer-grade PCs. KAIROX builds on activation sparsity to enable efficient **online neuron balancing** through a live pipeline, locality-aware neuron balancing, and feedback-driven tuning. Together, these designs navigate CPU and I/O bottlenecks, providing a foundation for low-latency hybrid LLM inference on resource-constrained hardware.

Acknowledgements

We thank our shepherd and the anonymous reviewers for their insightful comments and suggestions. This work was supported in part by the Major Key Project of Peng Cheng Laboratory under Grant No. PCL2025A11, the National Natural Science Foundation of China under Grant No. 62472459, and the Guangdong S&T Program (2025B0101080001).

References

[1] Abien Fred Agarap. Deep learning using rectified linear units (relu). *arXiv preprint arXiv:1803.08375*, 2018.

[2] Keivan Alizadeh, Seyed Iman Mirzadeh, Dmitry Belenko, S Khatamifard, Minsik Cho, Carlo C Del Mundo, Mohammad Rastegari, and Mehrdad Farajtabar. Llm in a flash: Efficient large language model inference with limited memory. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 12562–12584, 2024.

[3] George Bebis and Michael Georgiopoulos. Feed-forward neural networks. *Ieee Potentials*, 13(4):27–31, 2002.

[4] Weilin Cai, Juyong Jiang, Fan Wang, Jing Tang, Sunghun Kim, and Jiayi Huang. A survey on mixture of experts in large language models. *IEEE Transactions on Knowledge and Data Engineering*, 2025.

[5] Jerry Chee, Yaohui Cai, Volodymyr Kuleshov, and Christopher M De Sa. Quip: 2-bit quantization of large language models with guarantees. *Advances in Neural Information Processing Systems*, 36, 2024.

[6] Hao Chen, Cong Tian, Zixuan He, Bin Yu, Yepang Liu, and Jialun Cao. Inference performance evaluation for llms on edge devices with a novel benchmarking framework and metric. *arXiv preprint arXiv:2508.11269*, 2025.

[7] Yihua Cheng, Yuhan Liu, Jiayi Yao, Yuwei An, Xiaokun Chen, Shaoting Feng, Yuyang Huang, Samuel Shen, Kuntai Du, and Junchen Jiang. Lmcache: An efficient kv cache layer for enterprise-scale llm inference. *arXiv preprint arXiv:2510.09665*, 2025.

[8] Yu Cheng, Duo Wang, Pan Zhou, and Tao Zhang. A survey of model compression and acceleration for deep neural networks. *arXiv preprint arXiv:1710.09282*, 2017.

[9] Tim Dettmers, Mike Lewis, Younes Belkada, and Luke Zettlemoyer. Gpt3. int8 (): 8-bit matrix multiplication for transformers at scale. *Advances in Neural Information Processing Systems*, 35:30318–30332, 2022.

[10] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. In *Proceedings of the 2019 conference of the North American chapter of the association for computational linguistics: human language technologies, volume 1 (long and short papers)*, pages 4171–4186, 2019.

[11] Stefan Elfwing, Eiji Uchibe, and Kenji Doya. Sigmoid-weighted linear units for neural network function approximation in reinforcement learning. *Neural networks*, 107:3–11, 2018.

[12] Zhiyuan Fang, Yuegui Huang, Zicong Hong, Yufeng Lyu, Wuhui Chen, Yue Yu, Fan Yu, and Zibin Zheng. Klotski: Efficient mixture-of-expert inference via expert-aware multi-batch pipeline. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, pages 574–588, 2025.

[13] Elias Frantar, Saleh Ashkboos, Torsten Hoefler, and Dan Alistarh. Gptq: Accurate post-training quantization for generative pre-trained transformers. *arXiv preprint arXiv:2210.17323*, 2022.

[14] Georgi Gerganov. ggerganov/llama.cpp: Port of facebook's llama model in c/c++. https://github.com/ggerganov/llama.cpp, 2023.

[15] Amir Gholami, Sehoon Kim, Zhen Dong, Zhewei Yao, Michael W Mahoney, and Kurt Keutzer. A survey of quantization methods for efficient neural network inference. In *Low-Power Computer Vision*, pages 291–326. Chapman and Hall/CRC, 2022.

[16] Lars Gottesbüren, Tobias Heuer, Peter Sanders, Christian Schulz, and Daniel Seemaier. Deep multilevel graph partitioning. In *29th Annual European Symposium on Algorithms, ESA 2021*, volume 204 of *LIPIcs*, pages 48:1–48:17. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021.

[17] Aaron Grattafiori, Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Alex Vaughan, et al. The llama 3 herd of models. *arXiv preprint arXiv:2407.21783*, 2024.

[18] Ying He, Jingcheng Fang, F Richard Yu, and Victor C Leung. Large language models (llms) inference offloading and resource allocation in cloud-edge computing: An active inference approach. *IEEE Transactions on Mobile Computing*, 2024.

[19] Han-Byul Kim, Duc Hoang, Arnav Kundu, Mohammad Samragh, and Minsik Cho. Spd: Sync-point drop for efficient tensor parallelism of large language models. *arXiv preprint arXiv:2502.20727*, 2025.

[20] Minsu Kim, Pinyarash Pinyoanuntapong, Bongho Kim, Walid Saad, and Doru Calin. Edge vs cloud: How do we balance cost, latency, and quality for large language models over 5g networks? In *2025 IEEE Wireless Communications and Networking Conference (WCNC)*, pages 1–6. IEEE, 2025.

[21] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph Gonzalez, Hao Zhang, and Ion Stoica. Efficient memory management for large language model serving with pagedattention. In *Proceedings of the 29th symposium on operating systems principles*, pages 611–626, 2023.

[22] Jiedong Lang, Zhehao Guo, and Shuyu Huang. A comprehensive study on quantization techniques for large language models. In *2024 4th International Conference on Artificial Intelligence, Robotics, and Communication (ICAIRC)*, pages 224–231. IEEE, 2024.

[23] Patrick J Laub, Thomas Taimre, and Philip K Pollett. Hawkes processes. *arXiv preprint arXiv:1507.02822*, 2015.

[24] Suyi Li, Hanfeng Lu, Tianyuan Wu, Minchen Yu, Qizhen Weng, Xusheng Chen, Yizhou Shan, Binhang Yuan, and Wei Wang. Caraserve: Cpu-assisted and rank-aware lora serving for generative llm inference. *arXiv preprint arXiv:2401.11240*, 2024.

[25] Suyi Li, Hanfeng Lu, Tianyuan Wu, Minchen Yu, Qizhen Weng, Xusheng Chen, Yizhou Shan, Binhang Yuan, and Wei Wang. Toppings:{CPU-Assisted},{Rank-Aware} adapter serving for {LLM} inference. In *2025 USENIX Annual Technical Conference (USENIX ATC 25)*, pages 613–629, 2025.

[26] Zonglin Li, Chong You, Srinadh Bhojanapalli, Daliang Li, Ankit Singh Rawat, Sashank J Reddi, Ke Ye, Felix Chern, Felix Yu, Ruiqi Guo, et al. The lazy neuron phenomenon: On emergence of activation sparsity in transformers. *arXiv preprint arXiv:2210.06313*, 2022.

[27] Peiyu Liu, Zikang Liu, Ze-Feng Gao, Dawei Gao, Wayne Xin Zhao, Yaliang Li, Bolin Ding, and Ji-Rong Wen. Do emergent abilities exist in quantized large language models: An empirical study, 2023.

[28] Zichang Liu, Jue Wang, Tri Dao, Tianyi Zhou, Binhang Yuan, Zhao Song, Anshumali Shrivastava, Ce Zhang, Yuandong Tian, Christopher Re, et al. Deja vu: Contextual sparsity for efficient llms at inference time. In *International Conference on Machine Learning*, pages 22137–22176. PMLR, 2023.

[29] Kai Lu, Qiang Wei, Yier Lin, PengYu Liu, Haipeng Wang, Jiguang Wan, Ting Yao, Huatao Wu, and Daohui Wang. Q-infer: Towards efficient gpu-cpu collaborative llm inference via sparsity-aware dynamic scheduling. *ACM Transactions on Architecture and Code Optimization*, 2025.

[30] Xupeng Miao, Gabriele Oliaro, Zhihao Zhang, Xinhao Cheng, Hongyi Jin, Tianqi Chen, and Zhihao Jia. Towards efficient generative large language model serving: A survey from algorithms to systems. *ACM Computing Surveys*, 58(1):1–37, 2025.

[31] NVIDIA. https://developer.nvidia.com/nsight-systems, 2018.

[32] NVIDIA. https://www.nvidia.com/en-us/geforce/graphics-cards/40-series/rtx-4090/, 2024.

[33] Ruoyu Qin, Zheming Li, Weiran He, Jialei Cui, Heyi Tang, Feng Ren, Teng Ma, Shangming Cai, Yineng Zhang, Mingxing Zhang, et al. Mooncake: A kvcache-centric disaggregated architecture for llm serving. *ACM Transactions on Storage*, 2024.

[34] Guanqiao Qu, Qiyuan Chen, Wei Wei, Zheng Lin, Xianhao Chen, and Kaibin Huang. Mobile edge intelligence for large language models: A contemporary survey. *IEEE Communications Surveys & Tutorials*, 2025.

[35] Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J Liu. Exploring the limits of transfer learning with a unified text-to-text transformer. *Journal of machine learning research*, 21(140):1–67, 2020.

[36] Noam Shazeer. Glu variants improve transformer. *arXiv preprint arXiv:2002.05202*, 2020.

[37] Ying Sheng, Shiyi Cao, Dacheng Li, Coleman Hooper, Nicholas Lee, Shuo Yang, Christopher Chou, Banghua Zhu, Lianmin Zheng, Kurt Keutzer, et al. Slora: Scalable serving of thousands of lora adapters. *Proceedings of Machine Learning and Systems*, 6:296–311, 2024.

[38] Ying Sheng, Lianmin Zheng, Binhang Yuan, Zhuohan Li, Max Ryabinin, Beidi Chen, Percy Liang, Christopher Ré, Ion Stoica, and Ce Zhang. Flexgen: High-throughput generative inference of large language models with a single gpu. In *International Conference on Machine Learning*, pages 31094–31116. PMLR, 2023.

[39] Chenyang Song, Xu Han, Zhengyan Zhang, Shengding Hu, Xiyu Shi, Kuai Li, Chen Chen, Zhiyuan Liu, Guangli Li, Tao Yang, et al. Prosparse: Introducing and enhancing intrinsic activation sparsity within large language models. In *Proceedings of the 31st International Conference on Computational Linguistics*, pages 2626–2644, 2025.

[40] Jifeng Song, Kai Huang, Xiangyu Yin, Boyuan Yang, and Wei Gao. Achieving sparse activation in small language models. *arXiv preprint arXiv:2406.06562*, 2024.

[41] Yixin Song, Zeyu Mi, Haotong Xie, and Haibo Chen. Powerinfer: Fast large language model serving with a consumer-grade gpu. In *Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles*, pages 590–606, 2024.

[42] Yixin Song, Haotong Xie, Zeyu Mi, Li Ma, and Haibo Chen. Bamboo: Harmonizing sparsity and performance in large language models, 2024.

[43] Yixin Song, Haotong Xie, Zhengyan Zhang, Bo Wen, Li Ma, Zeyu Mi, and Haibo Chen. Turbo sparse: Achieving llm sota performance with minimal activated parameters. *arXiv preprint arXiv:2406.05955*, 2024.

[44] Yixin Song, Zhenliang Xue, Dongliang Wei, Feiyang Chen, Jianxiang Gao, Junchen Liu, Hangyu Liang, Guangshuo Qin, Chengrong Tian, Bo Wen, et al. Smallthinker: A family of efficient large language models natively trained for local deployment. *arXiv preprint arXiv:2507.20984*, 2025.

[45] Jovan Stojkovic, Esha Choukse, Chaojie Zhang, Inigo Goiri, and Josep Torrellas. Towards greener llms: Bringing energy-efficiency to the forefront of llm inference. *arXiv preprint arXiv:2403.20306*, 2024.

[46] ShareGPT Team. https://sharegpt.com/.

[47] SpaseLLM Team. Sparse large language models with relu activation, 2023.

[48] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. *Advances in neural information processing systems*, 30, 2017.

[49] Hongyu Wang, Shuming Ma, Li Dong, Shaohan Huang, Huaijie Wang, Lingxiao Ma, Fan Yang, Ruiping Wang, Yi Wu, and Furu Wei. Bitnet: Scaling 1-bit transformers for large language models. *arXiv preprint arXiv:2310.11453*, 2023.

[50] Tuowei Wang, Ruwen Fan, Minxing Huang, Zixu Hao, Kun Li, Ting Cao, Youyou Lu, Yaoxue Zhang, and

Ju Ren. Neuralink: Fast on-device llm inference with neuron co-activation linking. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, pages 147–162, 2025.

[51] Tuowei Wang, Kun Li, Zixu Hao, Donglin Bai, Ju Ren, Yaoxue Zhang, Ting Cao, and Mao Yang. Long exposure: Accelerating parameter-efficient fine-tuning for llms under shadowy sparsity. In *SC24: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–18. IEEE, 2024.

[52] Guangxuan Xiao, Ji Lin, Mickael Seznec, Hao Wu, Julien Demouth, and Song Han. Smoothquant: Accurate and efficient post-training quantization for large language models. In *International Conference on Machine Learning*, pages 38087–38099. PMLR, 2023.

[53] Daliang Xu, Hao Zhang, Liming Yang, Ruiqi Liu, Gang Huang, Mengwei Xu, and Xuanzhe Liu. Fast on-device llm inference with npus. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1*, pages 445–462, 2025.

[54] Yuzhuang Xu, Xu Han, Zonghan Yang, Shuo Wang, Qingfu Zhu, Zhiyuan Liu, Weidong Liu, and Wanxiang Che. Onebit: Towards extremely low-bit large language models. *arXiv preprint arXiv:2402.11295*, 2024.

[55] Biwei Yan, Kun Li, Minghui Xu, Yueyan Dong, Yue Zhang, Zhaochun Ren, and Xiuzhen Cheng. On protecting the data privacy of large language models (llms): A survey. *arXiv preprint arXiv:2403.05156*, 2024.

[56] Zhewei Yao, Reza Yazdani Aminabadi, Minjia Zhang, Xiaoxia Wu, Conglong Li, and Yuxiong He. Zeroquant: Efficient and affordable post-training quantization for large-scale transformers. *Advances in Neural Information Processing Systems*, 35:27168–27183, 2022.

[57] Gyeong-In Yu, Joo Seong Jeong, Geon-Woo Kim, Soojeong Kim, and Byung-Gon Chun. Orca: A distributed serving system for Transformer-Based generative models. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 521–538, Carlsbad, CA, July 2022. USENIX Association.

[58] Hongbin Zhang, Taosheng Wei, Zhenyi Zheng, Jiangsu Du, Zhiguang Chen, and Yutong Lu. Td-pipe: Temporally-disaggregated pipeline parallelism architecture for high-throughput llm inference. *arXiv preprint arXiv:2506.10470*, 2025.

[59] Susan Zhang, Stephen Roller, Naman Goyal, Mikel Artetxe, Moya Chen, Shuohui Chen, Christopher Dewan, Mona Diab, Xian Li, Xi Victoria Lin, et al. Opt: Open pre-trained transformer language models. *arXiv preprint arXiv:2205.01068*, 2022.

[60] Yuxin Zhang, Lirui Zhao, Mingbao Lin, Yunyun Sun, Yiwu Yao, Xingjia Han, Jared Tanner, Shiwei Liu, and Rongrong Ji. Dynamic sparse no training: Training-free fine-tuning for sparse llms. *arXiv preprint arXiv:2310.08915*, 2023.

[61] Zhengyan Zhang, Yankai Lin, Zhiyuan Liu, Peng Li, Maosong Sun, and Jie Zhou. Moefication: Transformer feed-forward layers are mixtures of experts. In *Findings of the Association for Computational Linguistics: ACL 2022*, pages 877–890, 2022.

[62] Yuxin Zhou, Zheng Li, Jun Zhang, Jue Wang, Yiping Wang, Zhongle Xie, Ke Chen, and Lidan Shou. Floe: On-the-fly moe inference on memory-constrained gpu. *arXiv preprint arXiv:2505.05950*, 2025.

A Artifact Appendix

Abstract. The artifact for KAIROX includes the modified `llama.cpp` source tree, the KAIROX implementation, comparison-system runners, and scripts for building, benchmarking, and summarizing results. It is designed to let artifact evaluators validate the released implementation and check selected end-to-end throughput trends from the paper under representative consumer-PC configurations.

Scope. The required artifact-evaluation path focuses on availability and selected performance validation. Specifically, the artifact checks that KAIROX builds and executes correctly; that the short `simple_validation` path runs `llama_cpp`, `kairox`, and `neuralink` on one validation model for the selected hardware profile; and that the full `e2e_performance` path reproduces selected Figure 9 trends by comparing KAIROX against the included baselines under matching models, hardware profiles, and generation modes. The paper also compares against PowerInfer and Q-Infer, but those external repositories are outside the required artifact-evaluation path.

Contents. The artifact repository contains the modified `llama.cpp` tree with KAIROX, the top-level driver `bench_models.sh`, the per-backend runner `test_kairox.sh`, the build helper `compile_kairox.sh`, the fixed prompt set `prompts.txt`, the log parser `parse_bench_logs.py`, and the detailed guide `ARTIFACTS_EVALUATION.md`. Model weights are not stored in the repository; evaluators download the required GGUF model files separately from the SPIF model release.

Hosting. The artifact repository is hosted publicly at https://github.com/Anhelor/kairox.cpp. The repository is intended to remain readable throughout artifact evaluation and contains the software path needed for the required runs, except for external model weights and standard container or dependency downloads described in the guide.

Requirements. The paper evaluates two consumer-grade profiles: PC-Low, using an RTX 3080 Ti with 12 GB VRAM, 12 benchmark CPU threads, and PCIe 3.0 x16; and PC-High, using an RTX 4090 with 24 GB VRAM, 16 benchmark CPU threads, and PCIe 4.0 x16. The closest reproduction uses matching GPUs, Docker, NVIDIA Container Toolkit, an NVIDIA driver compatible with CUDA 12.8, and sufficient disk space for source, build outputs, logs, summaries, and selected GGUF model files. Different CPUs, memory bandwidth, PCIe generations, drivers, or model subsets may change absolute throughput, so the artifact emphasizes matching comparisons within the same profile.

A.1 Installation

Evaluators clone the artifact repository, prepare a model directory, and enter a CUDA 12.8 development container with GPU access. Inside the container, they install the standard build dependencies, verify that the GPU is visible, and download either the full SPIF GGUF model set or a smaller validation-only subset for the selected profile. The exact host and container commands, including optional smaller downloads for PC-Low and PC-High validation, are provided in `ARTIFACTS_EVALUATION.md`. No separate upstream `llama.cpp` checkout is required because the artifact repository already contains the modified tree.

A.2 Experiment Workflow

The artifact provides two workflows. The short `simple_validation` workflow is a smoke test that builds the release binaries and runs completion and speculative decoding for `llama_cpp`, `kairox`, and `neuralink` on the profile's validation model: Prosparse-LLaMA2-7B for PC-Low and Prosparse-LLaMA2-13B for PC-High. It is invoked as `bash bench_models.sh low simple` or `bash bench_models.sh high simple`.

The full `e2e_performance` workflow runs the validation path followed by the selected throughput experiments. It uses a maximum context size of 1024 tokens and a maximum output length of 512 tokens, matching the paper's Figure 9 setup. It is invoked as `bash bench_models.sh low full` or `bash bench_models.sh high full`. Logs are written to `low_logs/` or `high_logs/`; evaluators can inspect each log's final `benchmark summary` block or run `parse_bench_logs.py` to generate Markdown and CSV summaries.

A.3 Expected Results

The `simple_validation` run passes if the build succeeds, all backends finish without runtime errors, logs are written, and each log ends with a `benchmark summary` block. The full `e2e_performance` run passes if the parser can summarize the logs and KAIROX reports a higher `decode mean` than the included comparison systems for matching profile, model, and generation mode. Because speculative decoding is sensitive to acceptance rate and hardware differences, exact throughput can vary; the expected result is the same relative trend under matched conditions rather than bit-for-bit reproduction of every number in the paper.

ADAngel: Accelerating Arbitrary-Precision Quantized LLMs with Adaptive Computing Mapping

Yao Liu, Wenjie Wang, Yifei Feng, Bo Peng, Jianguo Yao, Haibing Guan
Shanghai Jiao Tong University

Abstract

Arbitrary-Precision Quantization (APQ), which uses asymmetric bit-widths for weights and activations (e.g., W4A8), is a prevalent technique for LLM inference because of its excellent accuracy-performance balance. APQ transforms the general matrix multiplications (GEMM), the core of LLM computation, into mixed-precision GEMM (mpGEMM) whose two operand matrices have different quantization bit-widths. However, we identify that the computation paradigms of mpGEMM in current APQ LLM inference systems are suboptimal because the shapes and bit-widths of mpGEMM tasks in APQ LLM are highly variable, whereas existing static and workload-unaware paradigms can only accelerate mpGEMM tasks with the same or similar shapes and bit-widths.

Based on this finding, we propose ADAngel, a framework for creating a workload-adaptive mpGEMM computation core for target LLMs. The theoretical foundation of ADAngel is the DPR (Decomposition-Partial Product-Reconstruction) computation model, which enables systematic generation of a diverse portfolio of mpGEMM algorithms by specifying different bit-partition schemes. Guided by this model, ADAngel constructs a Computation Strategy Set comprising several highly optimized mpGEMM kernels, and exhaustively analyzes the strategy set to create an Oracle Policy Map, which enables a lightweight dispatcher to select and execute the optimal kernel for runtime mpGEMM tasks with negligible overhead. Our evaluation shows that the ADAngel-specialized engine achieves up to a 5.10× speedup in decode throughput over llama.cpp; while in the prefill stage, it demonstrates its adaptivity by delivering speedups ranging from 1.17× to 2.38× over TensorRT-LLM in Time-To-First-Token (TTFT).

1 Introduction

Large Language Models (LLMs) [9, 21, 30, 35] are increasingly being deployed on edge devices [19, 20, 25] to enhance the capabilities of edge intelligent tasks and agents. Unfortunately, edge devices are constrained by design limitations in chip area, power consumption, and cost, resulting in significantly inferior computing and memory/storage capabilities compared to cloud-scale infrastructure. Deploying full-scale LLMs on edge devices is inefficient or often infeasible because of limited hardware resources. Post-training quantization (PTQ) [15, 31, 33, 36] is a mainstream model compression technique that is widely adopted for edge-side LLM deployment. It quantizes the weights or activations of trained LLMs to reduce data bit-width and computational complexity, thereby enabling efficient inference on edge devices. Arbitrary-Precision Quantization (APQ) [3, 14, 26] quantizes weights and activations into different bit-widths, such as W4A8 (4-bit weights and 8-bit activations), thereby achieving a fine-grained trade-off between inference speed and quality. APQ results in LLMs' general matrix-matrix multiplication (GEMM) or general matrix-vector multiplication (GEMV) having operands with different bit-widths (mixed-precision bit-widths). However, existing edge computing units or accelerators lack native hardware support for operations with asymmetric bit-width operands.

To enable existing edge devices to support inference for APQ LLMs, padding [11, 17], lookup table (LUT) [18, 22, 32], and bit-disaggregation [39] techniques have been proposed. Padding techniques upcast low-bit-width weights to match the bit-width of activations, ensuring compatibility with existing edge computing units (e.g., executing W4A8 operations on INT8 Tensor Cores by upcasting weights to 8-bit). LUT-based approaches precompute the products of high-bit-width activations and low-bit-width weights, storing them in a lookup table. During inference, results are directly fetched from the LUT to avoid asymmetric GEMM and GEMV operations. Bit-disaggregation decomposes weights and activations of varying bit-widths into 1-bit representations, reconstructing the mixed-precision computation using 1-bit GEMM and GEMV kernels. Zeng et al. [39] leverage 1-bit Tensor Cores of NVIDIA Ampere GPUs to accelerate bit-level GEMM and GEMV. Due to the memory overhead and hardware support requirements, LUT-based methods exhibit lower efficiency than padding and bit-disaggregation on edge devices.

However, our investigation reveals that relying solely on

padding or bit-disaggregation fails to accommodate the computational heterogeneity in LLM inference, thereby hindering the inference speed of APQ LLMs on edge devices. Specifically, (1) **Computational heterogeneity between prefill and decode:** LLM inference consists of two phases, prefill and decode. Prefill processes long-sequence prompts, and compute-intensive GEMM is the primary workload. Decode generates tokens sequentially based on Key-Value Cache (KV Cache), and memory-bound GEMV is the dominant operation. Padding and bit-disaggregation techniques demonstrate suboptimal adaptability for the computational and memory patterns of these two stages, resulting in unstable inference performance.

(2) **Intra-Prefill computational heterogeneity:** The arithmetic intensity of GEMM in the prefill phase increases with the sequence length. Our analysis indicates that bit-disaggregation incurs lower overhead for short sequences, while padding demonstrates superior efficiency for long sequences. A static strategy fails to adapt to these dynamic workload variations.

(3) **Intra-Decode computational heterogeneity:** Batching is a typical method adopted to improve decode parallelism. We observe that with increasing decode batch size, the Linear layers transition towards higher arithmetic intensity, thus requiring adaptive strategy selection.

(4) **Computational heterogeneity in quantization bit-width:** The quantization bit-widths of weights and activations affect the GEMM's arithmetic intensity. Bit-disaggregation outperforms padding at ultra-low bit-widths. Conversely, padding proves more effective for higher bit-widths.

The root cause of these suboptimal behaviors of existing algorithms is the fundamental mismatch between the static, one-size-fits-all nature and the computational heterogeneity inherent in APQ LLM inference. Essentially, this heterogeneity stems from both runtime dynamics and intrinsic model structures. Most importantly, dynamic shifts in sequence length and batch size drastically alter the M dimension. As shown in Figure 1, for a given N and K, the optimal algorithm changes as M varies. Furthermore, variations in N and K also significantly affect the optimal strategy, as they change the arithmetic intensity and workload characteristics of GEMM. In practice, these effects are further shaped by the tiling mechanisms used in existing GEMM algorithms. Existing static paradigms fail because they cannot adapt to the shifting operational characteristics.

To this end, we propose ADAngel, an adaptive computing mapping system designed to accelerate APQ LLM inference on edge devices. At the heart of ADAngel lies the DPR (Decomposition-Partial Product-Reconstruction) computation model, a unified theoretical abstraction that formalizes diverse software adaptation algorithms (e.g., padding, bit-disaggregation) of mixed-precision GEMM (mpGEMM) into a single, generalized description. Guided by DPR, we systematically derive a diverse Computation Strategy Set to

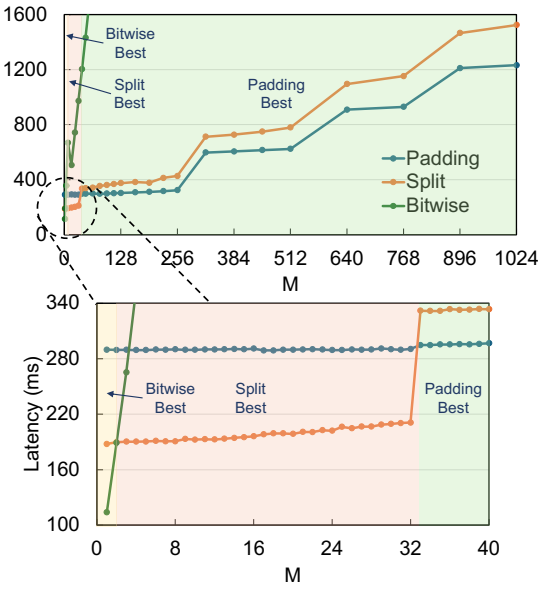


Figure 1: **The performance of three static computation strategies** (Split is our novel strategy, and Bitwise is based on bit-disaggregation) under different M for a typical W4A8 mpGEMM task in LLMs (N = 4096, K = 6144).

cover the spectrum of computational heterogeneity comprehensively. Then, in an offline phase, ADAngel exhaustively profiles these DPR-derived strategies across the target workload space defined by the target LLM to construct an Oracle Policy Map. This map is then embedded into a lightweight dispatcher, enabling the engine to dynamically select and execute the optimal kernel for every mpGEMM task at runtime with negligible overhead. For Arbitrary-Precision Quantized Models (e.g., W4A8), ADAngel natively accelerates the underlying mpGEMM operations, delivering maximum performance.

In summary, our main contributions are as follows:

(1) We are the first to perform a systematic analysis and characterization of the computational heterogeneity in APQ LLM inference, revealing the fundamental sub-optimality of static approaches.

(2) We propose the DPR (Decomposition-Partial Product-Reconstruction) computation model, a unified theoretical abstraction that establishes bit-level decomposition as a first principle to systematically generate a series of computation strategies for effective mpGEMM acceleration.

(3) We design and implement ADAngel, a methodology for constructing a workload-adaptive computation core for any given LLM through exhaustive offline profiling on several computation strategies derived by the DPR model.

(4) We conduct an extensive evaluation demonstrating that the ADAngel-specialized engine consistently outperforms state-of-the-art frameworks on both edge and data-center GPUs, with particularly large gains in long-prefill cases.

2 Background

2.1 LLM Quantization

The rapid advancement in LLMs (Figure 2, left) enhances model accuracy while substantially increasing computing and storage requirements. For example, deploying DeepSeek-R1-70B with FP16 precision consumes over 140GB of memory, requiring at least two modern high-performance NVIDIA H100 [4] GPUs with 80GB capacity. This poses significant challenges for deploying LLMs on low-cost edge devices with limited computing and memory/storage resources. Post-training quantization (PTQ) [15, 31, 33, 36] reduces memory/storage and computing costs by converting weights or activations from high-precision (FP32/FP16) to low-precision (INT8/INT4). It has become a mainstream approach for edge-side LLM deployment. Although quantizing model weights or activations enables LLM inference on edge devices, low-bit-width precision reduces model accuracy.

Existing studies [27, 37, 42] demonstrate that LLM weights exhibit high redundancy, making them amenable to low-bit-width quantization. In contrast, quantizing activations is more challenging, as low-bit-width activation representation significantly degrades LLM's quality. Arbitrary-precision quantization (APQ) [3, 14, 26] preserves LLM quality while minimizing resource demands by quantizing weights to low-bit-width representations and maintaining high-precision formats for activations, as shown in Figure 2 (Middle). In this paper, APQ refers to arbitrary weight-activation bit-width combinations, such as W4A4, W4A8, and W3A8. This usage differs from recent any-precision LLM approaches that support multiple model sizes or precision configurations from a shared representation [23]. However, APQ introduces asymmetric bit-widths between weights and activations. Current edge devices generally lack native or efficient support for asymmetric-bit-width GEMM/GEMV, requiring padding [11, 17], LUT [18, 22, 32], or bit-disaggregation [39].

Existing studies [27, 37, 42] demonstrate that LLM weights exhibit high redundancy, making them amenable to low-bit-width quantization. In contrast, quantizing activations is more challenging, as low-bit-width activation representation significantly degrades LLM's quality. In this paper, APQ refers to arbitrary-precision quantization, where weights and activations may use arbitrary and potentially asymmetric bit-widths within a deployed model, such as W4A8. This usage focuses on the execution of mixed-bit-width GEMM/GEMV operators. It differs from recent any-precision LLM approaches, which aim to support the deployment of multiple models or multiple bit-width configurations from a shared representation [23]. APQ preserves LLM quality while reducing resource demands by quantizing weights to low-bit-width representations and keeping activations at relatively higher precision, as shown in Figure 2 (Middle).

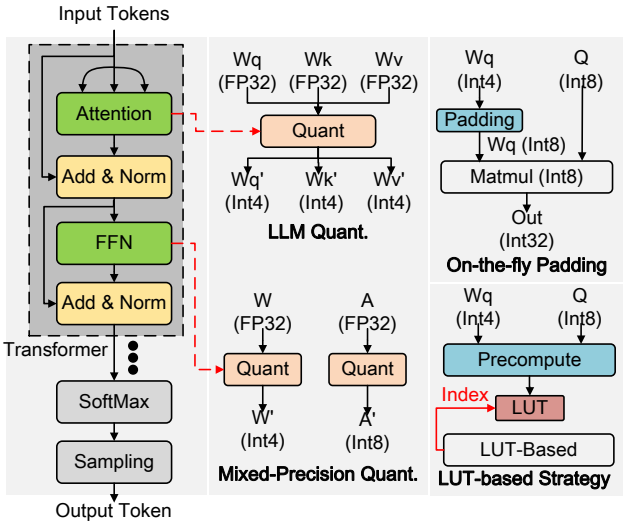


Figure 2: (1) Left: Decoder-only LLM architecture; (2) Middle: LLM quantization and Arbitrary-Precision Quantization (APQ); (3) Right: Padding and the LUT-based strategy in APQ LLM inference.

2.2 Padding and LUT in APQ LLM Inference

As shown in Figure 2 (Right), existing solutions [11, 17] widely adopt padding to implement APQ LLM inference on edge devices. It transforms low-bit-width weights to match activation bit-widths, then performs computation using conventional GEMM/GEMV operators. Although padding enables APQ LLM inference on edge devices, it introduces additional overheads: (1) the weight padding operation incurs additional cost: If it is performed online, it introduces runtime bit manipulation overhead that reduces the achievable effective compute throughput; If it is performed offline and the padded weights are stored in global memory, it leads to extra storage and memory access overhead; (2) it still executes computation on expanded high-bit-width weight representations rather than the original compact low-bit-width weights, thereby increasing computational cost.

Some researchers [18, 22, 32] propose the LUT-based strategy for direct asymmetric-bit-width GEMM/GEMV execution to avoid padding. Specifically, it precomputes the products of high-bit-width activations and multiple low-bit-width weights, and stores them in the LUT. Results are retrieved directly from the LUT instead of performing actual computations during LLM inference. However, constructing a LUT for each weight-activation bit-width combination introduces a significant memory footprint. Therefore, this paper focuses on padding and bit-disaggregation as the main software mapping strategies for GEMM/GEMV in APQ.

2.3 Bit-Disaggregation GEMM and GEMV

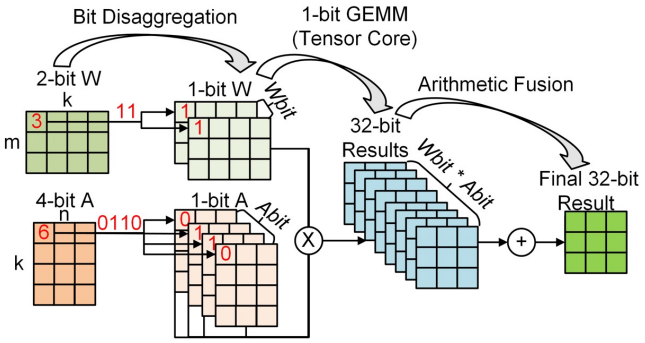


Figure 3: W2A4 GEMM bit-disaggregation workflow.

Bit-disaggregation is a mathematical approach for directly executing GEMM and GEMV with asymmetric bit-widths. As shown in Figure 3, it first disaggregates the weight matrix $[K,N]$ and the activation matrix $[M,K]$ into $WBit \times [K,N]$ and $ABit \times [M,K]$ 1-bit matrices (bitplanes) via bit-level disaggregation, where $WBit$ and $ABit$ represent the bit-width of weights and activations. Next, pairwise multiplications are performed between the 1-bit weight and activation matrices, yielding $WBit \times ABit$ result matrices. Finally, the $WBit \times ABit$ result matrices are combined into the final output. Equation 1 formally captures the bit-disaggregation process described above. It increases the number of GEMM and GEMV operations by a factor of $WBit \times ABit$, while also introducing additional combination overhead. Zeng et al. [39] leverage 1-bit Tensor Cores in NVIDIA Ampere architecture GPUs to accelerate 1-bit GEMM and GEMV, thereby optimizing the inference speed of APQ LLMs that adopt the bit-disaggregation strategy.

$$Result = \sum (2^0 W^{WBit-1} + ... + 2^{WBit-1} W^0) \times (2^0 A^{ABit-1} + ... + 2^{ABit-1} A^0) \quad (1)$$

3 Motivation

Although padding and bit-disaggregation enable APQ LLM inference on existing edge devices, our measurements show that using either alone fails to effectively capture the computational heterogeneity in LLM inference, which leads to suboptimal inference performance. To better understand this, we next analyze this heterogeneity along four dimensions.

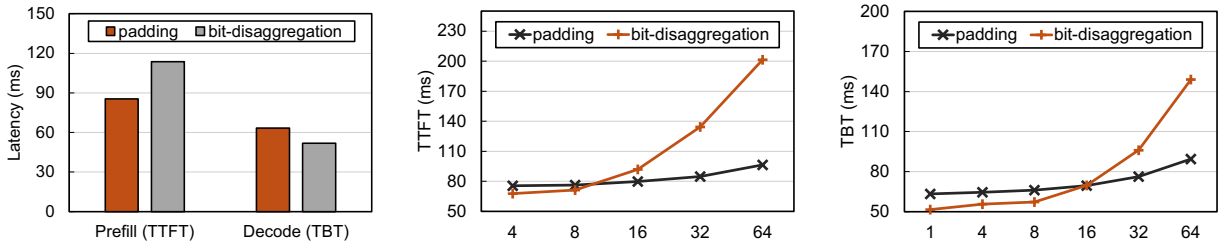
We deploy Llama-2-7B on the edge GPU of NVIDIA Jetson AGX Orin [20] using FasterTransformer [6]. The model adopts W4A8 quantization (4-bit weights and 8-bit activations). To support W4A8, FasterTransformer's GEMM and GEMV operators are reimplemented using padding and bit-disaggregation. In particular, to avoid the runtime overhead of on-the-fly weight format conversion, padded weights are precomputed and stored in global memory.

Computational Heterogeneity Between Prefill And Decode. Figure 4a presents the prefill and decode latency of the model when using padding and bit-disaggregation approaches, under the following configuration: prefill length = 32, generated sentence length = 10, and batch size = 1. Prefill latency is measured as Time To First Token (TTFT), while decode latency is recorded as Time Between Tokens (TBT).

The results show that TTFT with the padding method achieves lower TTFT than bit-disaggregation during prefill. In contrast, the decode TBT with bit-disaggregation is better than padding. The underlying reason is that prefill is typically more compute-intensive, where computation tends to dominate the performance. Bit-disaggregation significantly increases computational overhead, leading to poor performance. In contrast, decode corresponds to an I/O-intensive regime, where performance is primarily limited by memory traffic. In this case, bit-disaggregation achieves better performance mainly because it preserves a compact low-bit-width weight representation and reduces effective memory traffic. **Observation 1: Using only padding or bit-disaggregation fails to address the computational heterogeneity between prefill and decode, leading to suboptimal APQ LLM inference performance.**

Intra-Prefill Computational Heterogeneity. We increase the prefill length from 4 to 64 and measure the TTFT for Llama-2-7B's prefill. As shown in Figure 4b, the TTFT exhibits a monotonic increase with growing prefill lengths for both padding and bit-disaggregation approaches. We observe a crossover at a prefill length of 8: bit-disaggregation achieves a lower TTFT for short lengths (≤ 8), while padding demonstrates superior efficiency beyond this threshold. This crossover reflects the same compute-memory tradeoff observed between prefill and decode, but now within the prefill phase itself. For short prefills, the workload remains small, so the overhead introduced by loading expanded weights cannot be sufficiently amortized, making bit-disaggregation more favorable. As the prefill length increases, the GEMM becomes increasingly compute-intensive and the extra computation overhead of bit-disaggregation becomes its bottleneck. In this regime, padding achieves better TTFT because it incurs lower runtime computational overhead. Although the crossover point appears at prefill length = 8 in our experiments, it is not universal and can vary across model architectures, quantization bit-widths, and batch sizes. **Observation 2: Relying solely on padding or bit-disaggregation individually cannot fully account for the computational heterogeneity within prefill in APQ LLM inference.**

Intra-Decode Computational Heterogeneity. Next, we investigate the computational heterogeneity within decode. Figure 4c presents the TBT of Llama-2-7B decode with padding and bit-disaggregation for batch sizes 1∼64. The performance crossover point between padding and bit-disaggregation occurs at batch size = 16 in our experiments, although this point is not universal and can vary across model architectures

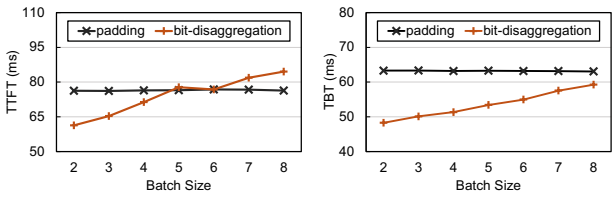


(a) Prefill and Decode (b) Prompt Length (c) Decode Batch Size

Figure 4: The computational heterogeneity between prefill and decode, intra-prefill computational heterogeneity, and intra-decode computational heterogeneity.

and quantization bit-widths. The underlying reason is the workload heterogeneity within decode. As the batch size increases, the decode workload gradually shifts from a memory-sensitive regime toward a more compute-intensive regime. For small batch sizes, performance is primarily limited by memory traffic, making bit-disaggregation more favorable because it preserves a compact low-bit-width weight representation and reduces memory access overhead. As the batch size increases, the computational overhead introduced by bit-disaggregation becomes increasingly significant, and padding achieves lower TBT due to its lower runtime processing overhead. **Observation 3: Using only padding or bit-disaggregation cannot fully adapt to the workload heterogeneity within decode in APQ LLM inference.**

Computational Heterogeneity in Quantization Bit-Width. Finally, with the activation bit-width fixed at 8 bits and the weight bit-width varying from 2 to 8 bits, we evaluate the prefill TTFT and decode TBT at prefill length = 32, generated sentence length = 10, and batch size = 1 for both padding and bit-disaggregation methods. Figure 5 demonstrates that TTFT and TBT remain stable with the padding method as the weight bit-width increases. This stability occurs because the padding method follows a similar execution pattern across different weight bit-widths in these settings. In contrast, bit-disaggregation progressively introduces more accesses to the weights and 1-bit matrix computations as weight bit-width increases, resulting in increased TTFT and TBT. **Observation 4: Neither padding nor bit-disaggregation alone can capture the computational heterogeneity introduced by the quantization bit-width in APQ LLMs.**



(a) Prefill (b) Decode

Figure 5: **Computational heterogeneity in quantization bit-width**. The activations are fixed at 8-bit quantization, while the weights vary from 2-bit to 8-bit. The x-axis represents the weight quantization bit-width.

4 Design

4.1 Overall Architecture

Our preceding analysis reveals that workload-unaware computation paradigms are fundamentally suboptimal for APQ LLM inference, failing to address the widespread computational heterogeneity of mpGEMM in LLM inference. To achieve high-performance inference, therefore, a new workload-adaptive paradigm for mpGEMM is needed, whose adaptivity is predicated on two core principles: (1) **Optionality**, the availability of a diverse portfolio of kernels that leverage hardware resources differently, and (2) **Optimality**, the capability to select the optimal kernel for each specific task.

To realize this paradigm, we propose **ADAngel**, a workload-aware framework, which is designed to accelerate APQ LLM inference, as shown in Figure 6. The theoretical foundation of ADAngel is the ❶ **DPR** (Decomposition–Partial Product–Reconstruction) computation model, which provides a unified form to both express and implement a diverse spectrum of mixed-precision GEMM strategies. Guided by the DPR model and hardware characteristics, we first define a set of three canonical strategies (Padding, Split, Bitwise) and perform deep, manual optimizations on their kernel implementations to form the high-performance ❷ **Computation Strategy Set**. Subsequently, the ❸ **Strategy Dispatcher** exhaustively evaluates these strategies offline on every potential workload within any given target LLM. This yields an oracle policy map which maps workloads to strategies, enabling the Strategy Dispatcher to perform zero-overhead, adaptive strategy selection at runtime when the LLM inference engine needs to execute mpGEMM/mpGEMV.

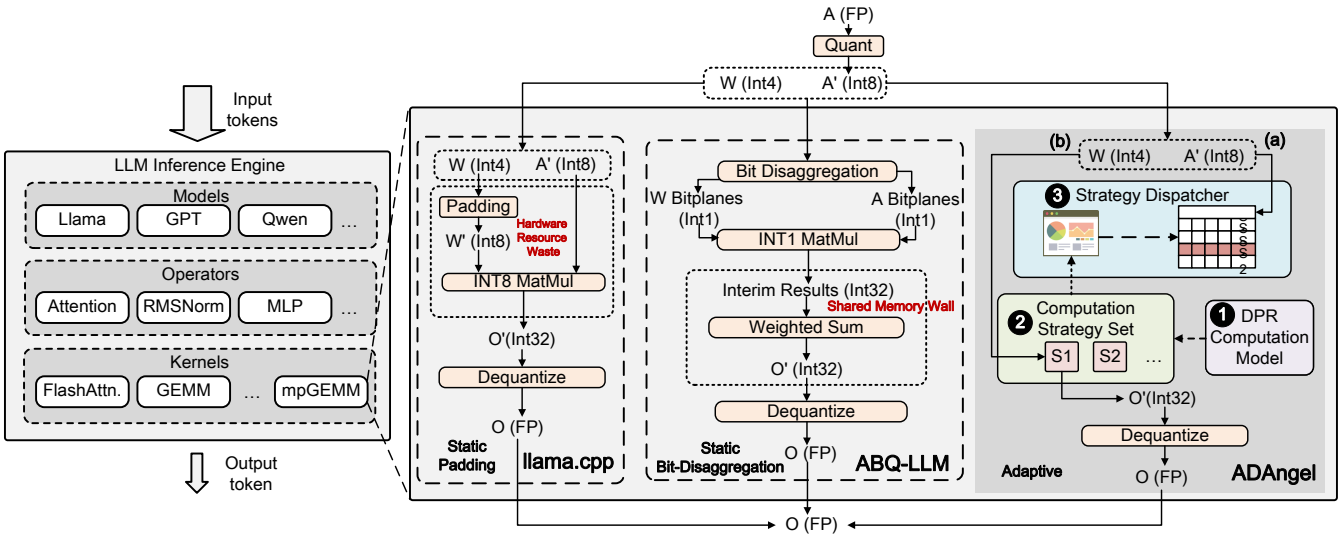


Figure 6: **The Overview of ADAngel**. In contrast to traditional static approaches, namely llama.cpp (which incurs hardware resource waste due to weight upcasting) and ABQ-LLM (which hits a shared memory wall during prefill due to bit-disaggregation, detailed in Section 6.2), ADAngel is built upon the ❶ DPR Computation Model. By analyzing hardware architecture, ADAngel constructs a ❷ Computation Strategy Set based on DPR constructs and employs a ❸ Strategy Dispatcher to map each mpGEMM task to the optimal computation strategy during APQ LLM inference. As indicated by the dashed paths (offline stage), ADAngel designs the strategy set under DPR guidance and evaluates these strategies on the target LLM to generate a dispatch table. As indicated by the solid arrows (runtime stage), ADAngel (a) retrieves the appropriate strategy based on the problem size and bit-width of the mpGEMM task, and (b) invokes the corresponding kernel to complete the computation.

4.2 DPR Framework

The performance upper bound of an adaptive system is fundamentally determined by the quality and diversity of its available strategies. Based on this principle, our framework requires a rich portfolio of high-performance computation strategies to be effective. The **DPR** (Decomposition–Partial Product–Reconstruction) computation model serves as the theoretical cornerstone for this task. That is, the strategies in our Computation Strategy Set are designed under the guidance of the DPR model, and the computation kernels dispatched and executed at runtime are concrete instantiations of it. DPR formally deconstructs any mixed-precision matrix multiplication into three logical stages: the framework first decomposes the input matrices into multiple components based on their bit-level representation; next, it performs matrix multiplication on these components to obtain a series of partial products; finally, it accurately reconstructs the final result via a weighted summation.

We now formally define this three-stage process. Given an activation matrix $X \in \mathbb{Z}^{M\times K}$ with bit-width $XBITS$ and a weight matrix $W \in \mathbb{Z}^{K\times N}$ with bit-width $WBITS$, our DPR framework computes the mpGEMM result $Y = XW \in \mathbb{Z}^{M\times N}$ as follows:

Decomposition. Its goal is to transform matrices X and W to corresponding physical tensors $\{X_{i,phys}\}$ and $\{W_{j,phys}\}$, which are uniformly typed and ready for computation by hardware accelerators like Tensor Cores in the next stage. It encapsulates two sub-steps: logical bit-partitioning and physical representation mapping.

We begin with defining an operation **logical bit-partition**: For a matrix A with $ABITS$ bit-width, based on a given bit-partitioning scheme:

$$P_A = (q_1, q_2, \ldots, q_m), \text{where} \sum_{i=1}^{m} q_i = ABITS, \tag{2}$$

the value of any element x in A is precisely expressed as a weighted sum of the corresponding elements x_i:

$$x = \sum_{i=1}^{m} x_i \cdot 2^{\text{shift}(i)}, \text{ where shift}(i) = \sum_{j=1}^{i-1} q_j. \tag{3}$$

Thus matrix A can be precisely expressed as a weighted sum of matrices A_i:

$$A = \sum_{i=1}^{m} A_i \cdot 2^{\text{shift}(i)}, \text{ where shift}(i) = \sum_{j=1}^{i-1} q_j, \tag{4}$$

then $\{A_i\}$ is the **logical bit-partition** of A based on the given bit-partitioning scheme P_A.

In the logical bit-partitioning step, respectively, $P_X = (q_{X1}, q_{X2}, \ldots, q_{Xm})$ and $P_W = (q_{W1}, q_{W2}, \ldots, q_{Wn})$ are applied to X and W, generating their logical partitions $\{X_s\}_{s=1}^{m}$ and

$\{W_t\}_{t=1}^{n}$. By applying different bit-partitioning schemes, this step provides a high degree of flexibility, allowing us to conceptually decompose matrices into logical components of arbitrary bit-widths. However, the underlying hardware acceleration units, particularly Tensor Cores, are not general-purpose multipliers. They are highly specialized computation engines, optimized for a fixed set of symmetric input precision pairs. This creates a mismatch between the flexibility of the logical decomposition at the software level and the rigid constraints of the hardware execution units. For instance, a 4-bit logical component and an 8-bit logical component cannot be multiplied directly by any single, native Tensor Core instruction.

To bridge this gap, the Physical Representation Mapping stage is an essential step. Its core mission is to unify all potentially heterogeneous logical matrices onto a single, homogeneous computation data type that is natively supported by the hardware. This is achieved through our Hardware-Aware Promotion Principle. The process first determines the maximum bit-width across all logical partitions, $b_{\max}$. It then promotes this value to the smallest, natively supported hardware precision that can accommodate it. Let H denote the set of natively supported hardware precisions on the target platform (e.g., $H = \{1,4,8\}$ on our target platform), then the final target bit-width can be expressed as

$$b_{\text{target}} = \min\{h \in H | h \geq \max\{q_{X1}, \ldots, q_{Xm}, q_{W1}, \ldots, q_{Wn}\}\}. \tag{5}$$

Subsequently, every logical matrix A_i (representing any W_s or X_t) is physically realized into a tensor $A_{i,phys}$ by promoting it to this target bit-width. This ensures that all tensors entering the computation stage are of a uniform, hardware-supported precision.

The resulting physical tensors from the decomposition stage are stored contiguously in memory, which maximizes bandwidth through coalesced accesses and allows us to fuse multiple partial product computations into a single kernel launch (e.g., computing $X_1 \times W_1$ and $X_2 \times W_1$ simultaneously), significantly reducing dispatch overhead.

Partial Product Computation. The output of the Decomposition stage is a set of hardware-ready physical tensors, $X_{s,phys}\}$ and $\{W_{t,phys}\}$, which all share a uniform bit-width, b_{target}. Therefore, each partial-product computation operates on operands with the same physical bit-width, allowing it to directly invoke symmetric-precision matrix-computation units, such as INT1, INT4, or INT8 Tensor Cores. The goal of the Partial Product Computation stage is to perform the actual matrix multiplication on these prepared tensors. Specifically, this stage executes $m \times n$ multiplications on the pairs of physical tensors to yield a series of partial product matrices, Y_{st}:

$$Y_{st} = \text{ComputeOp} < b_{target} > (X_{s,phys}, W_{t,phys}). \tag{6}$$

This stage's computational homogeneity ensures all operations are mapped to a single type of native hardware instruction (e.g., IMMA or BMMA) on the Tensor Cores, which facilitates mapping to a uniform hardware path.

It generates $m \times n$ partial product matrices, each of dimension $M \times N$. Crucially, the elements of these matrices are 32-bit integers (int32). This high precision is dictated by the architecture of the Tensor Cores, and exposes a fundamental performance trade-off: a more fine-grained decomposition (larger m or n) results in a proportionally larger memory footprint for these int32 intermediate results. This increased pressure on on-chip resources (especially shared memory) can constrain the achievable parallelism and occupancy of the GPU, a cost that can completely offset the inherent latency and throughput advantages of lower-precision Tensor Cores.

Reconstruction. In the final stage, our objective is to compute the result matrix Y. With the distributive property of multiplication, the product $Y = XW$ can be expanded from their logical decompositions into a weighted sum of partial products:

$$\begin{aligned} Y = XW &= (\sum_{s=1}^{m} X_s \cdot 2^{\text{shift}(s)})(\sum_{t=1}^{n} W_t \cdot 2^{\text{shift}(t)}) \\ &= \sum_{s=1}^{m}\sum_{t=1}^{n} X_s W_t \cdot 2^{\text{shift}(t)+\text{shift}(s)}. \end{aligned} \tag{7}$$

In summary, the DPR computation model provides a formal and principled foundation for mixed-precision GEMM. It establishes a clear methodology to deconstruct a complex, heterogeneous computation into a series of simple, hardware-aligned homogeneous operations.

4.3 Computation Strategy Set

The purpose of the Computation Strategy Set is to provide ADAngel with a broad and high-performance portfolio of computation strategies. Although the representational capacity of our DPR model provides a vast theoretical design space for this set, a significant portion of these strategies are practically inefficient. This inefficiency stems from the redundant promotion overhead introduced by our Hardware-Aware Promotion Principle when a strategy's logical bit-width does not align with the discrete precisions natively supported by the GPU's Tensor Cores.

To make this concrete, consider a W4A8 matrix multiplication where the activation partition is fixed at $P_A = (8)$. A weight decomposition of $P_W = (2,2)$ is provably suboptimal compared to the trivial $P_W = (4)$ scheme, because the $(2,2)$ scheme decomposes the work into two separate W8A8 GEMM operations, effectively doubling the computational workload.

Therefore, our method is to prune the design space by minimizing the promotion overhead, which is achieved by aligning the decomposed bit-widths with the native Tensor Core precisions. Based on this principle, we designed three families of computation strategies.

Throughput-Oriented: Padding. This strategy's philosophy is to achieve the highest possible computational throughput. It is formally expressed in our DPR model by a trivial partitioning scheme (i.e., $P_A = (ABITS)$) which maps the computation to the lowest-precision native hardware unit capable of accommodating the largest operand (e.g., INT8 Tensor Cores). Since it corresponds to a trivial partitioning scheme, the computation is not decomposed into multiple logical components, and no explicit reconstruction of partial products is required at runtime, resulting in lower overhead and higher effective throughput closer to the hardware peak.

Memory-Oriented: Bitwise. This strategy prioritizes memory efficiency. It corresponds to the finest-grained decomposition scheme in our DPR model (i.e., $P = (1,1,\ldots,1)$), replacing conventional arithmetic with a large number of simple bit-logic operations. This avoids the expansion overhead of padding, but introduces massive additional partial products and reconstruction overhead. Since it does not incur the extra weight memory traffic caused by padded operand representations, it is often more efficient in memory-bound scenarios.

Balanced: Split. On hardware platforms that provide native INT4 acceleration, there exists an intermediate alternative between Padding and Bitwise. Instead of using trivial partitioning or 1-bit decomposition, the weights and activations can be partitioned into a small number of hardware-aligned chunks that are as large as possible while not exceeding 4 bits (e.g., $P_W = (x), P_A = (4,4)$ for WxA8, where $x \leq 4$).

This strategy achieves a middle ground: compared to Padding, while both of them increase memory traffic by widening operand representations, Split incurs less overhead by using narrower widths (e.g., 4-bit vs. 8-bit in W2A8); compared to Bitwise, it reduces the number of partial products, and lowers both the reconstruction overhead and the intermediate-result memory footprint. The advantage of Split over Bitwise primarily stems from the reduced intermediate memory footprint. In Bitwise, a W_bA8 GEMM is decomposed into $8W_b$ bit-plane pairs, so the intermediate int32 storage for an output tile of size $T_M \times T_N$ scales as $4 \cdot (8W_b) \cdot T_M T_N$ bytes. In contrast, Split uses a much coarser partitioning (e.g., decomposing the activation into two 4-bit components), resulting in only $4 \cdot 2 \cdot T_M T_N$ bytes. Thus, Split reduces the intermediate-result footprint by a factor of $4W_b$; for W4A8, this corresponds to a 16× reduction, from 32 int32 partial products to 2. Since these intermediate int32 results typically need to be buffered (e.g., in shared memory) before the Reconstruction stage, this reduction directly lowers shared-memory pressure, improves resident CTA count and effective memory bandwidth, and thereby enables better performance than Bitwise under middle-M workloads.

To make the application of our three strategies concrete, we now instantiate them for a W4A8 mixed-precision task. The selection of the bit-partitioning scheme for each archetype, and the resulting hardware computation path dictated by our DPR model, are summarized in Figure 7.

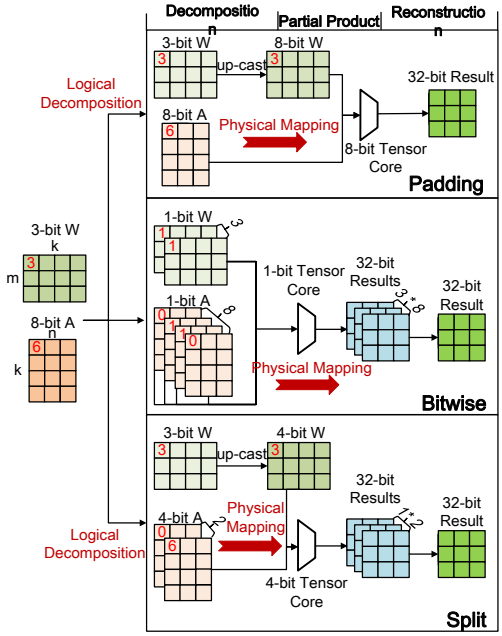


Figure 7: **Visualizing the DPR Framework**. For a canonical W4A8 task, how different Logical Decompositions and Physical Mappings, guided by our DPR model, result in three distinct hardware-aligned computation strategies.

For the above three strategy prototypes, high performance in practice also depends on efficient kernel-level implementations. We highly optimize all kernels in our strategy set, as detailed in Section 5.

In summary, guided by the theoretical principles of our DPR computation model, we have designed and implemented three distinct, highly optimized computation strategies that constitute our Computation Strategy Set. This portfolio of kernels provides the rich and diverse action space that is the essential foundation for ADAngel's adaptivity.

4.4 Strategy Dispatcher

The final component of the ADAngel framework is responsible for fulfilling the Optimality requirement: selecting the best-performing kernel from our Strategy Set for any given task. Our approach is grounded in a key insight: for a specific LLM, the variety of its constituent GEMM operations is highly constrained. This observation makes an exhaustive offline profiling approach not only feasible but also optimal. Therefore, our design consists of two phases: an offline policy construction phase that performs the analysis to generate an oracle policy map, and a lightweight online table-driven dispatch phase that executes mpGEMM tasks with the best strategy selected by the policy map.

We analyze the GEMM workload space within a target LLM. While the M dimension of a GEMM is dynamic, the N and K dimensions are static, determined by the model's archi-

tecture (e.g., hidden dimension). Consequently, the number of unique GEMM "types" (defined by static N and K dimensions) is small and finite. For instance, our analysis shows that an entire Llama-3-8B model inference involves only about 4 unique types of GEMM operations.

This bounded problem space makes an exhaustive, offline profiling approach conceivable, but its practical feasibility depends on the storage cost of the resulting lookup table. We quantify this cost as follows: storing function pointers for 4 unique GEMM types across a comprehensive range of $M = 1 \sim 8192$ (requiring 8 bytes per pointer) results in a total size of approximately 256 KB. This trivial memory footprint confirms that our empirical performance oracle approach is not only theoretically sound but also eminently practical.

The oracle policy map is constructed through an exhaustive, empirical profiling process executed once per target LLM and hardware platform. To achieve this, we first define the complete workload space by performing a static analysis of the target model's architecture to identify its finite set of unique GEMM "types", and then sweeping across a comprehensive range of the dynamic M dimension. For every point in this space, we benchmark each kernel from our Computation Strategy Set, using high-precision timers after a sufficient warm-up period to ensure accuracy. The kernel with the lowest average latency for each specific task is recorded, creating the oracle policy map: a simple key-value structure that maps a workload to a function pointer for its empirically proven optimal kernel.

The runtime behavior of the Strategy Dispatcher is simple and efficient. Upon receiving a GEMM task, it performs a direct lookup in the pre-computed oracle policy map using the task's shape as a key. This retrieves and immediately invokes a pointer to the empirically proven optimal kernel.

This dispatch mechanism, being a simple table lookup, is what provides ADAngel's two decisive advantages: guaranteed optimality (relative to our offline analysis) and negligible runtime overhead.

5 Implementation

We implemented the ADAngel framework in C++ and CUDA 12.6 with over 15k LOC. Our implementation first involved engineering the Computation Strategy Set: a portfolio of kernels where Padding and Split are built upon CUTLASS [8], and Bitwise is a custom BMMA-based implementation. With this optimized portfolio established, we then constructed the Strategy Dispatcher's policy through an automated, exhaustive offline profiling process. Finally, we integrated these components as an mpGEMM backend into FasterTransformer for end-to-end evaluation.

High-Performance Kernel Engineering. To ensure our strategy portfolio is highly competitive, we engineered each kernel family for maximum performance. The Padding and Split strategies were built on top of the CUTLASS 3.1.0 library. In our implementation, we restrict Split to weight bit-widths ≤ 4, since it no longer reduces weight memory traffic relative to Padding when the weight bit-width exceeds 4. In Split, weights are offline promoted to 4-bit and stored in a packed layout. At runtime, the 8-bit activation is decomposed into two 4-bit components and packed into a stacked $2M \times K$ matrix. Then, it is multiplied with the offline-packed weights via a single $(2M, N, K)$ CUTLASS GEMM, yielding a stacked $(2M, N)$ int32 intermediate. A lightweight merge kernel then completes the Reconstruction stage and dequantization. In contrast, the Bitwise strategy was implemented as a custom CUDA kernel based on BMMA instructions. We inherited the highly optimized weight layout from ABQ-LLM [39] to maximize memory coalescing, and applied selective kernel fusion (e.g., fusing the activation quantization and Decomposition stage) to reduce global memory traffic.

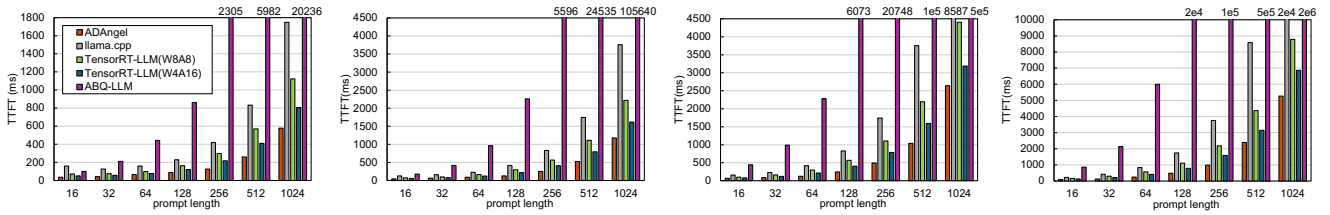
Offline Policy Construction. The Strategy Dispatcher's policy was constructed by an automated Python script that orchestrates the entire profiling workflow. It benchmarked every kernel in our strategy set across a comprehensive range of sequence lengths in the target LLM, using CUDA events for precise latency measurement. By selecting the lowest-latency kernel for each potential workload, this process generated the Oracle Policy Map. In FasterTransformer, we observed that Llama-3-8B involves 4 distinct (N, K) combinations. To construct the dispatch table on the NVIDIA Jetson AGX Orin for W4A8-quantized Llama-3-8B, we profiled these (N, K) combinations over $M \in [1, 8192]$ to cover the potential workloads. This incurs a one-time profiling cost of about 5.7 hours and a peak memory footprint of about 1.7 GB. This time cost can be further reduced by pruning the search space, since Figure 1 shows that Padding exhibits a clear advantage in the large-M regime. In addition, previously profiled entries can be reused across the same models or layers that share the same hardware platform.

End-to-End Framework Integration. To evaluate real-world performance, we integrated our ADAngel computation core into NVIDIA's FasterTransformer. We further implemented support for the widely used model Llama-3-8B within our FasterTransformer-based testbed. The integration is modular, requiring only redirecting the framework's standard mpGEMM calls to our ADAngel dispatcher.

6 Evaluation

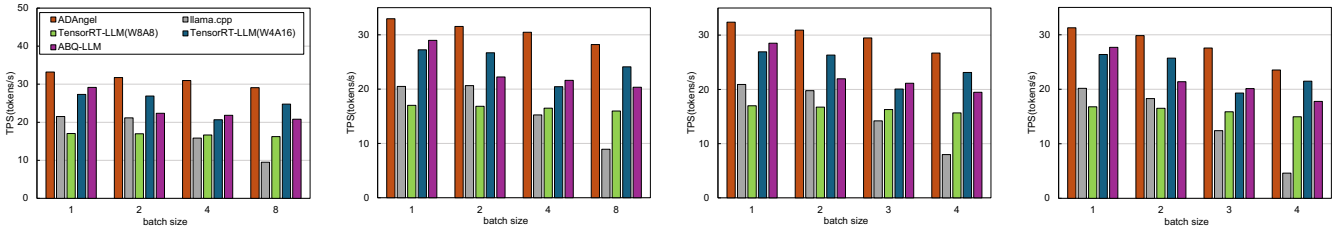
6.1 Experiment Setup

Platform. Experiments were conducted on an NVIDIA Jetson AGX Orin Development Kit 64GB, featuring an Ampere-architecture GPU (sm_87) and 64GB unified memory. We used the NVIDIA JetPack 6.1 [7] software stack (CUDA 12.6). The power mode of Orin is MODE_50W. Our entire testbed is built upon FasterTransformer, and we implemented W2A8,



(a) Llama-3-8B, batch size = 1 (b) Llama-3-8B, batch size = 2 (c) Llama-3-8B, batch size = 4 (d) Llama-3-8B, batch size = 8

Figure 8: **Prefill Stage Performance Comparison**. ADAngel significantly reduces the TTFT across various input sequence lengths, demonstrating its superiority in compute-bound scenarios.



(a) Llama-3-8B, pl = 128 (b) Llama-3-8B, pl = 256 (c) Llama-3-8B, pl = 512 (d) Llama-3-8B, pl = 1024

Figure 9: **Decode Stage Performance Comparison**. ADAngel achieves substantially higher TPS during the auto-regressive generation phase, showcasing its efficiency in memory-bound, low-latency operations.

W3A8, W4A8 and W5A8 versions of Llama-3 within this testbed. To validate the generality of the ADAngel methodology, we applied ADAngel to a data-center-class NVIDIA A100 GPU [5], generating a new, A100-specific engine.

Models. We used a representative modern LLM: Llama-3-8B.

Baselines and Our System: (1) ABQ-LLM [39], serves as a representative framework for bit-disaggregation-based mpGEMM. (2) llama.cpp [17]. This serves as an optimized, end-to-end baseline utilizing online padding mpGEMM routines. (3) TensorRT-LLM. As a recognized high-performance inference framework, TensorRT-LLM serves as a strong baseline. Since the current version for the Orin platform lacks native W4A8 support, we employ W8A8 (SmoothQuant) and W4A16 (AWQ) quantization methods for a comprehensive comparison. (4) QServe. Its W4A8KV4 configuration is a powerful solution for cloud inference but is currently unsupported on Orin. To address this, we compare ADAngel with QServe on an A100 80GB GPU to evaluate the scalability of our approach. (5) ADAngel. This is our main proposal. All mpGEMM operations in our testbed are handled by our ADAngel-specialized computation core.

Evaluation Metrics: We report (1) Time-To-First-Token (TTFT) to measure the latency of the Prefill stage, and (2) Tokens Per Second (TPS) per request to measure the throughput of the Decode stage. All presented results are the average of multiple runs following a warm-up period to ensure stable and accurate measurements.

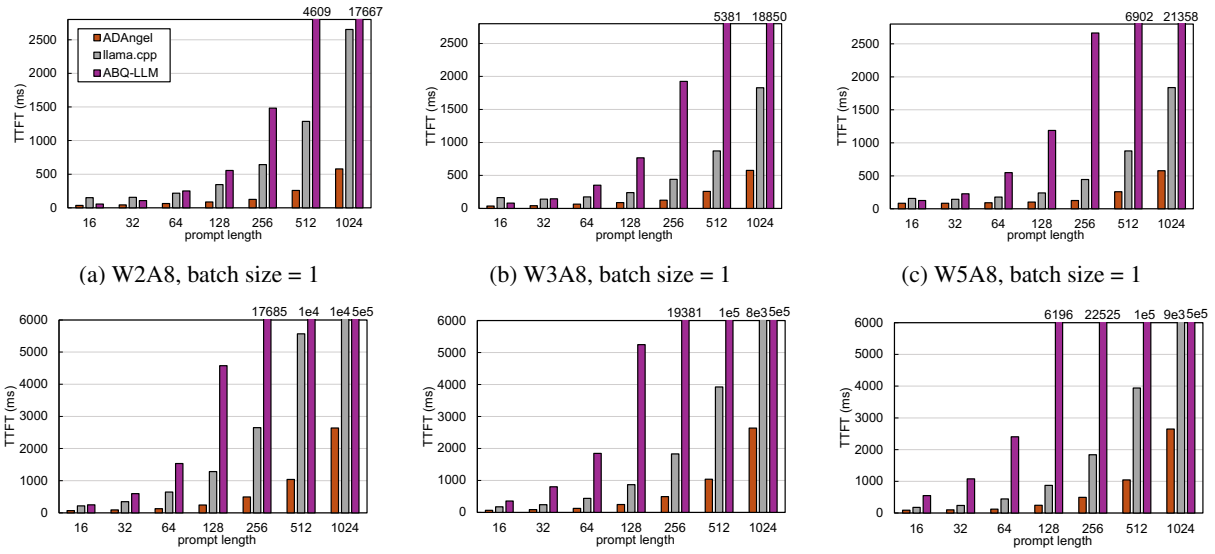
6.2 End-to-end Evaluation

Prefill. Figure 8 illustrates the end-to-end prefill latency (TTFT) across varying prompt lengths from 16 to 1024. ADAngel consistently establishes superior performance, achieving a peak speedup of 448.69× over ABQ-LLM and a speedup ranging from 1.17× to 2.38× over TensorRT-LLM.

Compared to TensorRT-LLM configurations, ADAngel surpasses the W4A16 baseline by leveraging the superior arithmetic throughput of INT8 Tensor Cores over FP16, and exceeds the W8A8 baseline by virtue of a lightweight system architecture that eliminates the significant runtime overheads inherent in the TensorRT-LLM framework.

In the short-prompt regime, the workload remains memory-sensitive. However, the mpGEMM kernel in llama.cpp exposes only limited CTA-level parallelism under its persistent stream-k scheduling, and does not implement an asynchronous global-to-shared-memory pipeline. As a result, memory latency cannot be effectively hidden, making the kernel substantially less efficient in this regime. In contrast, our Bitwise and Split kernels contain asynchronous pipelining to overlap memory accesses with computation, thereby achieving higher efficiency. For instance, at bs = 1 and pl = 32, ADAngel reduces latency by 65.9% compared to llama.cpp (43 ms vs. 126 ms).

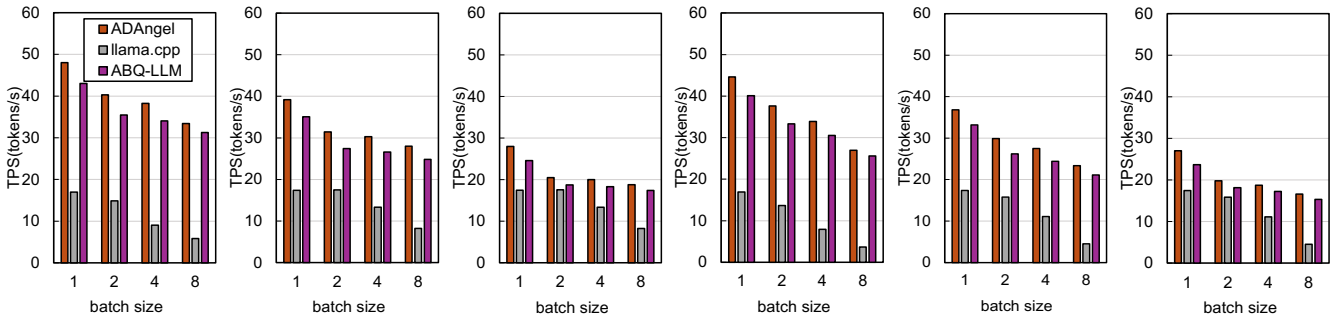
As the workload transitions to the compute-bound long-sequence regime, the bit-disaggregation approach (ABQ-LLM) consumes up to 32× more shared memory per thread block compared to padding-based approaches. This bottleneck, identified as the Shared Memory Wall, severely restricts



(a) W2A8, batch size = 1 (b) W3A8, batch size = 1 (c) W5A8, batch size = 1

(d) W2A8, batch size = 4 (e) W3A8, batch size = 4 (f) W5A8, batch size = 4

Figure 10: **Cross-precision prefill performance on Jetson AGX Orin**. TTFT comparison under W2A8, W3A8, and W5A8 across representative batch sizes.



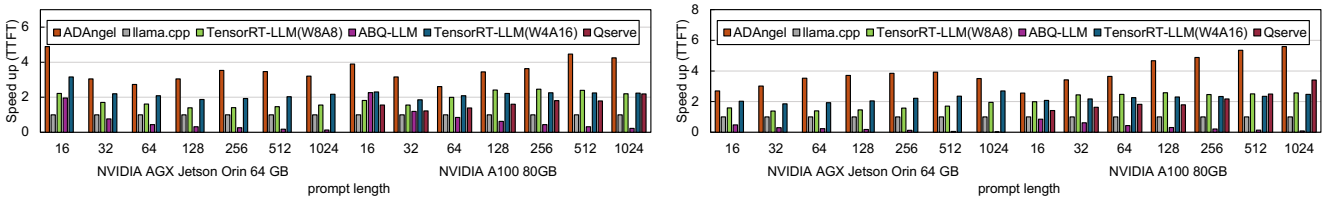
(a) W2A8, pl = 256 (b) W3A8, pl = 256 (c) W5A8, pl = 256 (d) W2A8, pl = 1024 (e) W3A8, pl = 1024 (f) W5A8, pl = 1024

Figure 11: **Cross-precision decode performance on Jetson AGX Orin**. TPS comparison under W2A8, W3A8, and W5A8 across representative prompt lengths.

GPU occupancy due to excessive intermediate accumulation states. Consequently, although ABQ-LLM is the SOTA framework for decoding and short-sequence prefill, it becomes practically unusable in this setting: at bs $= 8$ and pl $= 1024$, its TTFT surges to approximately 39.4 minutes. The root cause of the Shared Memory Wall is induced by the storage of int32 intermediate states. Our profiling reveals that this heavy shared memory consumption saturates the SM resources, restricting the hardware scheduler to launch only 3 thread blocks per SM, with each thread block computing only an 8×48 output tile. Such low occupancy effectively nullifies the massive parallelism advantage of the GPU. In contrast, ADAngel adaptively switches to the appropriate kernel to circumvent this latency explosion, maintaining the latency within seconds (5.27 s) and achieving a substantial 448.69× speedup.

ADAngel performs well across the evaluated representative sequence lengths. Since it covers all potential M values induced by sequence-length variations within the target range, we infer it to handle sequence length oscillations effectively.

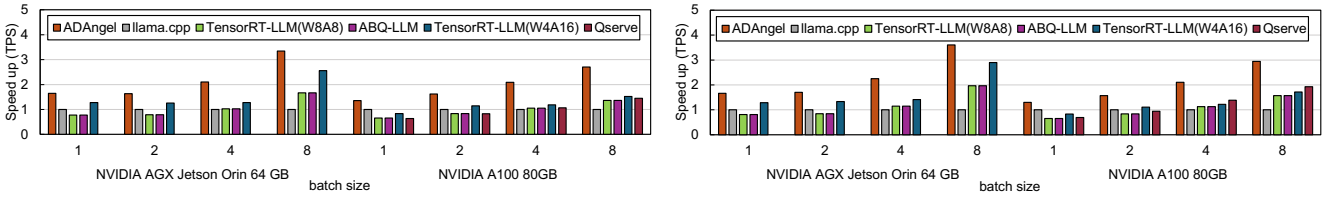
Decode. Figure 9 reports the decode throughput (TPS, tokens per second) across batch sizes from 1 to 8, where ADAngel consistently establishes superior performance. In the critical single-batch regime ($M = 1$), ADAngel achieves a 1.95× speedup over TensorRT-LLM W8A8 by eliminating two fundamental inefficiencies: it bypasses the physical bandwidth redundancy of using 8-bit weights, and simultaneously removes computational redundancy, where static padding kernels constrained by fixed INT8 Tensor Core shapes (typically $M = 16$) leave $\frac{15}{16}$ (93.75%) of compute cycles idle due to zero-padding. As the batch size scales ($M = 2 \sim 8$), while pure bit-disaggregation (ABQ-LLM) also circumvents these redundancies, it does so at the cost of excessive shared memory usage and instruction bloat; in contrast, ADAngel's Split strategy strikes an optimal balance between eliminating re-



(a) Llama-3-8B, batch size = 1

(b) Llama-3-8B, batch size = 4

Figure 12: **Prefill Stage Performance Speedup Across Hardware Platforms**. All results are normalized to the llama.cpp baseline on each respective platform (i.e., llama.cpp = 1.0×). ADAngel consistently outperforms both baselines.



(a) Llama-3-8B, prompt length = 64

(b) Llama-3-8B, prompt length = 512

Figure 13: **Decode Stage Performance Speedup Across Hardware Platforms**. All results are normalized to the llama.cpp baseline on each respective platform. ADAngel maintains its performance leadership on both the Orin and the A100 across all tested batch sizes.

dundancy and minimizing memory cost, thereby avoiding the resource bottlenecks encountered by bitwise methods and ultimately surpassing TensorRT-LLM W8A8 by 1.82× on average.

We further analyze the memory footprint of ADAngel to understand its resource trade-offs. Under a setting of bs = 8, prompt length = 1024, and generating 128 tokens per sequence, ADAngel achieves a peak memory usage of 18.80 GiB (≈ 31.5% of the device memory on Orin). This includes 14.96 GiB for model weights, 1.125 GiB for the KV cache, and 896 MiB for the ADAngel GEMM workspace, with the remaining memory attributed to runtime buffers and framework overhead. We materialize all decomposed weights in global memory to trade additional memory usage for inference performance, particularly in regimes where memory capacity is not the primary bottleneck.

6.3 Cross-Precision Evaluation

To demonstrate the generality of ADAngel across different precision combinations, we extend our evaluation to W2A8, W3A8, and W5A8 configurations. As shown in Figure 10, ADAngel consistently delivers superior prefill performance, outperforming both llama.cpp (by an average of 3.43× in TTFT) and ABQ-LLM (achieving up to 200.00× speedup in TTFT) across all evaluated bit-width combinations, confirming its robust adaptivity.

As weight bit-width increases, pure bit-disaggregation methods (ABQ-LLM) suffer from a severe TTFT expansion during long-sequence prefilling due to the proportional growth in bit-level matrix operations. ADAngel effectively curtails this latency by dynamically dispatching heavy compute workloads to the Padding strategy. Simultaneously, it maintains optimal decode throughput (TPS) by leveraging the Bitwise and Split strategies in memory-bound regimes, achieving a peak speedup of 7.35× over llama.cpp in TPS, representing the best end-to-end performance.

Notably, under the W5A8 configuration, the Split strategy becomes no longer effective. Consequently, ADAngel's TTFT speedup on short prompt lengths is less pronounced than in the W2A8 and W3A8 regimes (e.g., 92 ms in W5A8 vs. 65 ms in W2A8 at pl = 64). Nevertheless, ADAngel still yields significant performance gains over ABQ-LLM by adaptively switching away from Bitwise in large-M regimes.

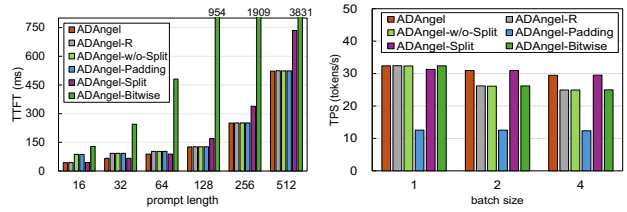
6.4 Generality Evaluation

Although ADAngel is primarily designed for resource-constrained edge scenarios, such as intelligent cockpits and robotics, the increasing adoption of arbitrary-precision quantization in cloud inference highlights its potential contribution to server-grade GPU efficiency. Figures 12 and 13 demonstrate this platform scalability by extending our evaluation to the NVIDIA A100 GPU (80GB). While QServe could not be executed on the Jetson AGX Orin due to a lack of support, we successfully deployed it on the A100 to enable a direct comparison. QServe employs strategies such as progressive group quantization tailored for high-throughput optimization; while

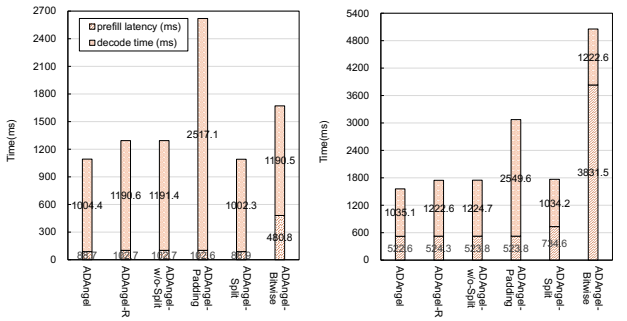
the associated overheads of these strategies are effectively hidden by massive parallelism in large-batch scenarios, they remain exposed in small-batch regimes. In contrast, ADAngel's adaptive approach successfully identifies performance bottlenecks in these specific scenarios and dispatches more suitable computation strategies, achieving speedups of 2.12× in TTFT and 1.72× in TPS relative to QServe. Furthermore, thanks to its hardware-aware adaptive mechanism, ADAngel maintains its performance advantage over TensorRT-LLM, llama.cpp, and ABQ-LLM on the A100 platform.

6.5 Ablation Study

To evaluate the specific contributions of our fine-grained dispatching mechanism and diverse strategy portfolio, we constructed three distinct ablation baselines:



(a) Prefill Stage, bs = 2 (b) Decode Stage, pl = 512



(c) Total Time, bs = 2, pl = 64 (d) Total Time, bs = 2, pl = 512

Figure 14: **Performance breakdown and ablation study of ADAngel compared with different variants**. (a) Time to First Token (TTFT) across varying prompt lengths with batch size bs = 2. Note that ADAngel-Bitwise suffers from high latency in longer sequences. (b) Generation throughput (TPS) across different batch sizes with pl = 512. (c) & (d) Breakdown of total end-to-end latency (prefill latency vs. decode time) for short sequences (pl = 64) and long sequences (pl = 512) at bs = 2, with 32 tokens generated. ADAngel achieves the lowest total latency by balancing prefill and decode efficiency.

(1) ADAngel-*: To disentangle the contribution of our core adaptive methodology from the benefits of low-level kernel optimizations, we evaluated systems using a single static strategy, denoted as ADAngel-*, where (*) represents the specific fixed strategy used (e.g., ADAngel-Padding). This serves as a control group to isolate the gains attributed solely to dynamic scheduling.

(2) ADAngel-R: To assess the necessity of our fine-grained dispatcher, we compared ADAngel against a strong rule-based baseline. This variant employs a static heuristic based solely on M: mapping tasks with $M \in [1,8]$ to Bitwise, $M \in (8,32]$ to Split, and $M > 32$ to Padding.

(3) ADAngel-w/o-Split: To evaluate the criticality of our diverse strategy portfolio, we implemented an ablated variant that excludes the Split strategy from the candidate pool.

As shown in Figure 14, ADAngel maintains a consistent performance advantage across all ablation settings. In extreme regimes where the optimal strategy is static such as long-sequence prefilling, ADAngel correctly identifies the optimal kernel with negligible dispatch overhead (total < 3 ms).

Specifically, ADAngel-R achieves gains over static Padding by selecting Split at pl = 16 during prefill and Bitwise during decode; however, due to ignoring the dimensional heterogeneity introduced by N and K, it underperforms the full ADAngel system in decode scenarios (bs $= 2 \sim 4$), highlighting the critical need for fine-grained kernel selection that considers the full (M,N,K) for every GEMM.

Furthermore, ADAngel-w/o-Split, operating on a restricted strategy set, defaults to Padding behavior in prefill to avoid the latency explosion of the Bitwise strategy, yet sacrifices the significant speedup opportunities at pl $= 16 \sim 64$ provided by Split. Similarly, in the decode phase (bs $= 2 \sim 4$), although it outperforms static Padding, it still lags behind ADAngel due to the absence of the Split strategy.

Since the benefits of ADAngel manifest across different phases, we conducted an end-to-end evaluation simulating typical intelligent cockpit scenarios (instruction parsing and short QA: generating 32 tokens with bs = 2 and pl $= 64/512$). The measured total latency demonstrates that ADAngel outperforms almost all ablation baselines across all scenarios, leveraging its effective fine-grained adaptive mechanism and a rich set of computational strategies.

7 Related Work

Post-Training Quantization (PTQ) is a key technique for compressing LLMs. Seminal methods like GPTQ [12] use second-order information for error compensation, a direction refined by works like HQQ [1]. To handle outliers, AWQ [16] protects salient weights via scaling, while other works preprocess weights via smoothing (MagR [40]) or orthogonal transformations (e.g., QuIP [2]), learn the error with adapters (e.g., QLoRA [10]), or dynamically correct errors at runtime (DecDEC [24]). To further enable integer-only hardware, another line of work performs weight-activation quantization. SmoothQuant [34] pioneered this by migrating activation difficulty to the weights, paving the way for advanced methods that reorder activations (RPTQ [38]) or reshape distributions with learnable transformations (FlatQuant [29]). While

these methods produce accurate quantized models, ADAngel addresses the critical downstream challenge of executing the resulting complex GEMM workloads with maximum, workload-adaptive efficiency.

Kernel-Level Acceleration for mixed-precision GEMM is the domain most directly related to our work. Prior art in this domain has primarily explored two major directions: software adaptation for general-purpose hardware, and the design of direct hardware accelerators. The majority of work focuses on software adaptation, which aims to optimally map these complex operations onto existing hardware. One prominent approach is to develop highly-specialized, static kernels for specific precision pairings, as exemplified by Marlin [13], which provides a state-of-the-art W4A16 GEMM kernel for NVIDIA GPUs. A different software approach, seen in frameworks like llama.cpp [11, 17], performs on-the-fly data transformation; its kernels employ techniques like online padding and native integer vectorization, where low-precision weights are unpacked and converted to a compatible format just-in-time for computation. Alternative software methods reframe the problem entirely, either leveraging bit-logic operations as in ABQ-LLM [39], or replacing multipliers with memory lookups as seen in LUT-based methods like T-MAC [32]. In contrast, the direct hardware design approach proposes new accelerator architectures. For instance, MixPE [41] designs specialized multiplier-less processing units using shift-and-add operations, while BitFusion [28] introduces a methodology to dynamically fuse bit-level processing elements.

8 Conclusion

In this paper, we identified and characterized the fundamental computation heterogeneity in arbitrary-precision LLM inference, revealing the inherent limitations of static, workload-unaware approaches. To address this, we proposed ADAngel, a framework that leverages our DPR computation model and an exhaustive offline analysis to construct a specialized, workload-adaptive computation core for any given LLM. Our extensive evaluation demonstrates that the ADAngel-specialized engine significantly outperforms state-of-the-art inference frameworks, establishing that principled, workload-adaptive specialization is an essential paradigm for unlocking the full performance potential of low-precision hardware in edge LLM inference.

Acknowledgements

We thank our shepherd and the anonymous reviewers for their valuable insights and constructive feedback. This research was supported by NSFC (No. 62402317) and STCSM (No. 24ZR1435500 and No. 25LN3200900). The corresponding author is Bo Peng (pengbo_michael@sjtu.edu.cn).

References

[1] Hicham Badri and Appu Shaji. Half-quadratic quantization of large machine learning models. *Dan Hendrycks, Collin Burns, Steven Basart, Andy Zou, Mantas Mazeika, Dawn Song, and Jacob*, 2023.

[2] Jerry Chee, Yaohui Cai, Volodymyr Kuleshov, and Christopher M De Sa. Quip: 2-bit quantization of large language models with guarantees. In A. Oh, T. Naumann, A. Globerson, K. Saenko, M. Hardt, and S. Levine, editors, *Advances in Neural Information Processing Systems*, volume 36, pages 4396–4429. Curran Associates, Inc., 2023.

[3] Huancheng Chen and Haris Vikalo. Mixed-precision quantization for federated learning on resource-constrained heterogeneous devices. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 6138–6148, June 2024.

[4] Jack Choquette. Nvidia hopper h100 gpu: Scaling performance. *IEEE Micro*, 43(3):9–17, May 2023.

[5] Jack Choquette and Wish Gandhi. NVIDIA A100 GPU: Performance & Innovation for GPU Computing . In *2020 IEEE Hot Chips 32 Symposium (HCS)*, pages 1–43, Los Alamitos, CA, USA, August 2020. IEEE Computer Society.

[6] NVIDIA Corporation. Fastertransformer. https://github.com/NVIDIA/FasterTransformer, 2022. Accessed: 2025-05-21.

[7] NVIDIA Corporation. Jetpack 6.1 release notes. https://docs.nvidia.com/jetson/archives/jetpack-archived/jetpack-61/release-notes/index.html, 2024. Accessed: 2025-03-18.

[8] NVIDIA Corporation. Cutlass 4.1.0. https://github.com/NVIDIA/cutlass, 2025. Accessed: 2025-07-18.

[9] DeepSeek-AI, Aixin Liu, Bei Feng, Bing Xue, Bingxuan Wang, Bochao Wu, Chengda Lu, Chenggang Zhao, Chengqi Deng, Chenyu Zhang, Chong Ruan, Damai Dai, Daya Guo, Dejian Yang, Deli Chen, Dongjie Ji, Erhang Li, Fangyun Lin, Fucong Dai, Fuli Luo, Guangbo Hao, Guanting Chen, Guowei Li, H. Zhang, Han Bao, Hanwei Xu, Haocheng Wang, Haowei Zhang, Honghui Ding, Huajian Xin, Huazuo Gao, Hui Li, Hui Qu, J. L. Cai, Jian Liang, Jianzhong Guo, Jiaqi Ni, Jiashi Li, Jiawei Wang, Jin Chen, Jingchang Chen, Jingyang Yuan, Junjie Qiu, Junlong Li, Junxiao Song, Kai Dong, Kai Hu, Kaige Gao, Kang Guan, Kexin Huang, Kuai Yu, Lean Wang, Lecong Zhang, Lei Xu, Leyi Xia, Liang Zhao, Litong Wang, Liyue Zhang, Meng Li, Miaojun Wang, Mingchuan Zhang, Minghua Zhang, Minghui Tang, Mingming Li,

Ning Tian, Panpan Huang, Peiyi Wang, Peng Zhang, Qiancheng Wang, Qihao Zhu, Qinyu Chen, Qiushi Du, R. J. Chen, R. L. Jin, Ruiqi Ge, Ruisong Zhang, Ruizhe Pan, Runji Wang, Runxin Xu, Ruoyu Zhang, Ruyi Chen, S. S. Li, Shanghao Lu, Shangyan Zhou, Shanhuang Chen, Shaoqing Wu, Shengfeng Ye, Shengfeng Ye, Shirong Ma, Shiyu Wang, Shuang Zhou, Shuiping Yu, Shunfeng Zhou, Shuting Pan, T. Wang, Tao Yun, Tian Pei, Tianyu Sun, W. L. Xiao, Wangding Zeng, Wanjia Zhao, Wei An, Wen Liu, Wenfeng Liang, Wenjun Gao, Wenqin Yu, Wentao Zhang, X. Q. Li, Xiangyue Jin, Xianzu Wang, Xiao Bi, Xiaodong Liu, Xiaohan Wang, Xiaojin Shen, Xiaokang Chen, Xiaokang Zhang, Xiaosha Chen, Xiaotao Nie, Xiaowen Sun, Xiaoxiang Wang, Xin Cheng, Xin Liu, Xin Xie, Xingchao Liu, Xingkai Yu, Xinnan Song, Xinxia Shan, Xinyi Zhou, Xinyu Yang, Xinyuan Li, Xuecheng Su, Xuheng Lin, Y. K. Li, Y. Q. Wang, Y. X. Wei, Y. X. Zhu, Yang Zhang, Yanhong Xu, Yanhong Xu, Yanping Huang, Yao Li, Yao Zhao, Yaofeng Sun, Yaohui Li, Yaohui Wang, Yi Yu, Yi Zheng, Yichao Zhang, Yifan Shi, Yiliang Xiong, Ying He, Ying Tang, Yishi Piao, Yisong Wang, Yixuan Tan, Yiyang Ma, Yiyuan Liu, Yongqiang Guo, Yu Wu, Yuan Ou, Yuchen Zhu, Yuduan Wang, Yue Gong, Yuheng Zou, Yujia He, Yukun Zha, Yunfan Xiong, Yunxian Ma, Yuting Yan, Yuxiang Luo, Yuxiang You, Yuxuan Liu, Yuyang Zhou, Z. F. Wu, Z. Z. Ren, Zehui Ren, Zhangli Sha, Zhe Fu, Zhean Xu, Zhen Huang, Zhen Zhang, Zhenda Xie, Zhengyan Zhang, Zhewen Hao, Zhibin Gou, Zhicheng Ma, Zhigang Yan, Zhihong Shao, Zhipeng Xu, Zhiyu Wu, Zhongyu Zhang, Zhuoshu Li, Zihui Gu, Zijia Zhu, Zijun Liu, Zilin Li, Ziwei Xie, Ziyang Song, Ziyi Gao, and Zizheng Pan. Deepseek-v3 technical report, 2025.

[10] Tim Dettmers, Artidoro Pagnoni, Ari Holtzman, and Luke Zettlemoyer. Qlora: Efficient finetuning of quantized llms. In A. Oh, T. Naumann, A. Globerson, K. Saenko, M. Hardt, and S. Levine, editors, *Advances in Neural Information Processing Systems*, volume 36, pages 10088–10115. Curran Associates, Inc., 2023.

[11] Boyuan Feng, Yuke Wang, Tong Geng, Ang Li, and Yufei Ding. Apnn-tc: accelerating arbitrary precision neural networks on ampere gpu tensor cores. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC'21, New York, NY, USA, 2021. Association for Computing Machinery.

[12] Elias Frantar, Saleh Ashkboos, Torsten Hoefler, and Dan Alistarh. Gptq: Accurate post-training quantization for generative pre-trained transformers, 2023.

[13] Elias Frantar, Roberto L. Castro, Jiale Chen, Torsten Hoefler, and Dan Alistarh. Marlin: Mixed-precision auto-regressive parallel inference on large language models. In *Proceedings of the 30th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming*, PPoPP '25, page 239–251, New York, NY, USA, 2025. Association for Computing Machinery.

[14] Ziyi Guan, Hantao Huang, Yupeng Su, Hong Huang, Ngai Wong, and Hao Yu. Aptq: Attention-aware post-training mixed-precision quantization for large language models. In *Proceedings of the 61st ACM/IEEE Design Automation Conference*, New York, NY, USA, 2024. Association for Computing Machinery.

[15] Wei Huang, Yangdong Liu, Haotong Qin, Ying Li, Shiming Zhang, Xianglong Liu, Michele Magno, and Xiaojuan Qi. Billm: Pushing the limit of post-training quantization for llms, 2024.

[16] Ji Lin, Jiaming Tang, Haotian Tang, Shang Yang, Wei-Ming Chen, Wei-Chen Wang, Guangxuan Xiao, Xingyu Dang, Chuang Gan, and Song Han. Awq: Activation-aware weight quantization for on-device llm compression and acceleration. In P. Gibbons, G. Pekhimenko, and C. De Sa, editors, *Proceedings of Machine Learning and Systems*, volume 6, pages 87–100, 2024.

[17] The llama.cpp team. llama.cpp: Llm inference in c/c++. https://github.com/ggml-org/llama.cpp, 2025. Accessed: 2025-06-18.

[18] Saeed Maleki. Look-up mai gemm: Increasing ai gemms performance by nearly 2.5x via msgemm, 2023.

[19] NVIDIA. Nvidia jetson nano. https://developer.nvidia.com/embedded/jetson-nano, 2025. Accessed: 2025-07-18.

[20] NVIDIA. Nvidia jetson orin. https://www.nvidia.com/en-us/autonomous-machines/embedded-systems/jetson-orin/, 2025. Accessed: 2025-07-18.

[21] OpenAI, Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, Red Avila, Igor Babuschkin, Suchir Balaji, Valerie Balcom, Paul Baltescu, Haiming Bao, Mohammad Bavarian, Jeff Belgum, Irwan Bello, Jake Berdine, Gabriel Bernadett-Shapiro, Christopher Berner, Lenny Bogdonoff, Oleg Boiko, Madelaine Boyd, Anna-Luisa Brakman, Greg Brockman, Tim Brooks, Miles Brundage, Kevin Button, Trevor Cai, Rosie Campbell, Andrew Cann, Brittany Carey, Chelsea Carlson, Rory Carmichael, Brooke Chan, Che Chang, Fotis Chantzis, Derek Chen, Sully Chen, Ruby Chen, Jason Chen, Mark Chen, Ben Chess, Chester Cho, Casey Chu, Hyung Won Chung, Dave Cummings, Jeremiah Currier, Yunxing Dai, Cory Decareaux, Thomas Degry, Noah Deutsch, Damien Deville, Arka Dhar, David Dohan, Steve Dowling, Sheila

Dunning, Adrien Ecoffet, Atty Eleti, Tyna Eloundou, David Farhi, Liam Fedus, Niko Felix, Simón Posada Fishman, Juston Forte, Isabella Fulford, Leo Gao, Elie Georges, Christian Gibson, Vik Goel, Tarun Gogineni, Gabriel Goh, Rapha Gontijo-Lopes, Jonathan Gordon, Morgan Grafstein, Scott Gray, Ryan Greene, Joshua Gross, Shixiang Shane Gu, Yufei Guo, Chris Hallacy, Jesse Han, Jeff Harris, Yuchen He, Mike Heaton, Johannes Heidecke, Chris Hesse, Alan Hickey, Wade Hickey, Peter Hoeschele, Brandon Houghton, Kenny Hsu, Shengli Hu, Xin Hu, Joost Huizinga, Shantanu Jain, Shawn Jain, Joanne Jang, Angela Jiang, Roger Jiang, Haozhun Jin, Denny Jin, Shino Jomoto, Billie Jonn, Heewoo Jun, Tomer Kaftan, Łukasz Kaiser, Ali Kamali, Ingmar Kanitscheider, Nitish Shirish Keskar, Tabarak Khan, Logan Kilpatrick, Jong Wook Kim, Christina Kim, Yongjik Kim, Jan Hendrik Kirchner, Jamie Kiros, Matt Knight, Daniel Kokotajlo, Łukasz Kondraciuk, Andrew Kondrich, Aris Konstantinidis, Kyle Kosic, Gretchen Krueger, Vishal Kuo, Michael Lampe, Ikai Lan, Teddy Lee, Jan Leike, Jade Leung, Daniel Levy, Chak Ming Li, Rachel Lim, Molly Lin, Stephanie Lin, Mateusz Litwin, Theresa Lopez, Ryan Lowe, Patricia Lue, Anna Makanju, Kim Malfacini, Sam Manning, Todor Markov, Yaniv Markovski, Bianca Martin, Katie Mayer, Andrew Mayne, Bob McGrew, Scott Mayer McKinney, Christine McLeavey, Paul McMillan, Jake McNeil, David Medina, Aalok Mehta, Jacob Menick, Luke Metz, Andrey Mishchenko, Pamela Mishkin, Vinnie Monaco, Evan Morikawa, Daniel Mossing, Tong Mu, Mira Murati, Oleg Murk, David Mély, Ashvin Nair, Reiichiro Nakano, Rajeev Nayak, Arvind Neelakantan, Richard Ngo, Hyeonwoo Noh, Long Ouyang, Cullen O'Keefe, Jakub Pachocki, Alex Paino, Joe Palermo, Ashley Pantuliano, Giambattista Parascandolo, Joel Parish, Emy Parparita, Alex Passos, Mikhail Pavlov, Andrew Peng, Adam Perelman, Filipe de Avila Belbute Peres, Michael Petrov, Henrique Ponde de Oliveira Pinto, Michael, Pokorny, Michelle Pokrass, Vitchyr H. Pong, Tolly Powell, Alethea Power, Boris Power, Elizabeth Proehl, Raul Puri, Alec Radford, Jack Rae, Aditya Ramesh, Cameron Raymond, Francis Real, Kendra Rimbach, Carl Ross, Bob Rotsted, Henri Roussez, Nick Ryder, Mario Saltarelli, Ted Sanders, Shibani Santurkar, Girish Sastry, Heather Schmidt, David Schnurr, John Schulman, Daniel Selsam, Kyla Sheppard, Toki Sherbakov, Jessica Shieh, Sarah Shoker, Pranav Shyam, Szymon Sidor, Eric Sigler, Maddie Simens, Jordan Sitkin, Katarina Slama, Ian Sohl, Benjamin Sokolowsky, Yang Song, Natalie Staudacher, Felipe Petroski Such, Natalie Summers, Ilya Sutskever, Jie Tang, Nikolas Tezak, Madeleine B. Thompson, Phil Tillet, Amin Tootoonchian, Elizabeth Tseng, Preston Tuggle, Nick Turley, Jerry Tworek, Juan Felipe Cerón Uribe, Andrea Vallone, Arun Vijayvergiya, Chelsea Voss, Carroll Wainwright, Justin Jay Wang, Alvin Wang, Ben Wang, Jonathan Ward, Jason Wei, CJ Weinmann, Akila Welihinda, Peter Welinder, Jiayi Weng, Lilian Weng, Matt Wiethoff, Dave Willner, Clemens Winter, Samuel Wolrich, Hannah Wong, Lauren Workman, Sherwin Wu, Jeff Wu, Michael Wu, Kai Xiao, Tao Xu, Sarah Yoo, Kevin Yu, Qiming Yuan, Wojciech Zaremba, Rowan Zellers, Chong Zhang, Marvin Zhang, Shengjia Zhao, Tianhao Zheng, Juntang Zhuang, William Zhuk, and Barret Zoph. Gpt-4 technical report, 2024.

[22] Gunho Park, Baeseong Park, Minsub Kim, Sungjae Lee, Jeonghoon Kim, Beomseok Kwon, Se Jung Kwon, Byeongwook Kim, Youngjoo Lee, and Dongsoo Lee. Lut-gemm: Quantized matrix multiplication based on luts for efficient inference in large-scale generative language models, 2024.

[23] Yeonhong Park, Jake Hyun, Sanglyul Cho, Bonggeun Sim, and Jae W. Lee. Any-precision LLM: Low-cost deployment of multiple, different-sized LLMs. In Ruslan Salakhutdinov, Zico Kolter, Katherine Heller, Adrian Weller, Nuria Oliver, Jonathan Scarlett, and Felix Berkenkamp, editors, *Proceedings of the 41st International Conference on Machine Learning*, volume 235 of *Proceedings of Machine Learning Research*, pages 39682–39701. PMLR, 21–27 Jul 2024.

[24] Yeonhong Park, Jake Hyun, Hojoon Kim, and Jae W Lee. Decdec: A systems approach to advancing low-bit llm quantization. In *19th USENIX Symposium on Operating Systems Design and Implementation (OSDI 25)*, pages 803–819, 2025.

[25] Inc. Qualcomm Technologies. Qualcomm® sa8155p product brief. https://www.qualcomm.com/content/dam/qcomm-martech/dm-assets/documents/qul7413_sa8155_productbrief_r4.pdf, 2019. Accessed: 2025-03-18.

[26] Mariam Rakka, Mohammed E. Fouda, Pramod Khargonekar, and Fadi Kurdahi. A review of state-of-the-art mixed-precision neural network frameworks. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 46(12):7793–7812, 2024.

[27] Rajarshi Saha, Naomi Sagan, Varun Srivastava, Andrea J. Goldsmith, and Mert Pilanci. Compressing large language models using low rank and low precision decomposition. In A. Globerson, L. Mackey, D. Belgrave, A. Fan, U. Paquet, J. Tomczak, and C. Zhang, editors, *Advances in Neural Information Processing Systems*, volume 37, pages 88981–89018. Curran Associates, Inc., 2024.

[28] Hardik Sharma, Jongse Park, Naveen Suda, Liangzhen Lai, Benson Chau, Joon Kyung Kim, Vikas Chandra, and Hadi Esmaeilzadeh. Bit fusion: Bit-level dynamically composable architecture for accelerating deep neural network. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, pages 764–775, 2018.

[29] Yuxuan Sun, Ruikang Liu, Haoli Bai, Han Bao, Kang Zhao, Yuening Li, Jiaxin Hu, Xianzhi Yu, Lu Hou, Chun Yuan, Xin Jiang, Wulong Liu, and Jun Yao. Flatquant: Flatness matters for llm quantization, 2025.

[30] Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, Aurelien Rodriguez, Armand Joulin, Edouard Grave, and Guillaume Lample. Llama: Open and efficient foundation language models, 2023.

[31] Changyuan Wang, Ziwei Wang, Xiuwei Xu, Yansong Tang, Jie Zhou, and Jiwen Lu. Q-vlm: Post-training quantization for large vision-language models. In A. Globerson, L. Mackey, D. Belgrave, A. Fan, U. Paquet, J. Tomczak, and C. Zhang, editors, *Advances in Neural Information Processing Systems*, volume 37, pages 114553–114573. Curran Associates, Inc., 2024.

[32] Jianyu Wei, Shijie Cao, Ting Cao, Lingxiao Ma, Lei Wang, Yanyong Zhang, and Mao Yang. T-mac: Cpu renaissance via table lookup for low-bit llm deployment on edge. In *Proceedings of the Twentieth European Conference on Computer Systems*, EuroSys'25, page 278–292, New York, NY, USA, 2025. Association for Computing Machinery.

[33] Junyi Wu, Haoxuan Wang, Yuzhang Shang, Mubarak Shah, and Yan Yan. Ptq4dit: Post-training quantization for diffusion transformers. In A. Globerson, L. Mackey, D. Belgrave, A. Fan, U. Paquet, J. Tomczak, and C. Zhang, editors, *Advances in Neural Information Processing Systems*, volume 37, pages 62732–62755. Curran Associates, Inc., 2024.

[34] Guangxuan Xiao, Ji Lin, Mickael Seznec, Hao Wu, Julien Demouth, and Song Han. Smoothquant: Accurate and efficient post-training quantization for large language models. In Andreas Krause, Emma Brunskill, Kyunghyun Cho, Barbara Engelhardt, Sivan Sabato, and Jonathan Scarlett, editors, *Proceedings of the 40th International Conference on Machine Learning*, volume 202 of *Proceedings of Machine Learning Research*, pages 38087–38099. PMLR, 23–29 Jul 2023.

[35] An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengen Huang, Chenxu Lv, Chujie Zheng, Dayiheng Liu, Fan Zhou, Fei Huang, Feng Hu, Hao Ge, Haoran Wei, Huan Lin, Jialong Tang, Jian Yang, Jianhong Tu, Jianwei Zhang, Jianxin Yang, Jiaxi Yang, Jing Zhou, Jingren Zhou, Junyang Lin, Kai Dang, Keqin Bao, Kexin Yang, Le Yu, Lianghao Deng, Mei Li, Mingfeng Xue, Mingze Li, Pei Zhang, Peng Wang, Qin Zhu, Rui Men, Ruize Gao, Shixuan Liu, Shuang Luo, Tianhao Li, Tianyi Tang, Wenbiao Yin, Xingzhang Ren, Xinyu Wang, Xinyu Zhang, Xuancheng Ren, Yang Fan, Yang Su, Yichang Zhang, Yinger Zhang, Yu Wan, Yuqiong Liu, Zekun Wang, Zeyu Cui, Zhenru Zhang, Zhipeng Zhou, and Zihan Qiu. Qwen3 technical report, 2025.

[36] Zhewei Yao, Xiaoxia Wu, Cheng Li, Stephen Youn, and Yuxiong He. Exploring post-training quantization in llms from comprehensive study to low rank compensation. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 38, pages 19377–19385, 2024.

[37] Lu Yin, Ajay Jaiswal, Shiwei Liu, Souvik Kundu, and Zhangyang Wang. Junk dna hypothesis: Pruning small pre-trained weights irreversibly and monotonically impairs "difficult" downstream tasks in llms, 2025.

[38] Zhihang Yuan, Lin Niu, Jiawei Liu, Wenyu Liu, Xinggang Wang, Yuzhang Shang, Guangyu Sun, Qiang Wu, Jiaxiang Wu, and Bingzhe Wu. Rptq: Reorder-based post-training quantization for large language models, 2023.

[39] Chao Zeng, Songwei Liu, Yusheng Xie, Hong Liu, Xiaojian Wang, Miao Wei, Shu Yang, Fangmin Chen, and Xing Mei. Abq-llm: Arbitrary-bit quantized inference acceleration for large language models. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 39, pages 22299–22307, 2025.

[40] Aozhong Zhang, Naigang Wang, Yanxia Deng, Xin Li, Zi Yang, and Penghang Yin. Magr: Weight magnitude reduction for enhancing post-training quantization. In A. Globerson, L. Mackey, D. Belgrave, A. Fan, U. Paquet, J. Tomczak, and C. Zhang, editors, *Advances in Neural Information Processing Systems*, volume 37, pages 85109–85130. Curran Associates, Inc., 2024.

[41] Yu Zhang, Mingzi Wang, Lancheng Zou, Wulong Liu, Hui-Ling Zhen, Mingxuan Yuan, and Bei Yu. Mixpe: Quantization and hardware co-design for efficient llm inference, 2024.

[42] Yilong Zhao, Chien-Yu Lin, Kan Zhu, Zihao Ye, Lequn Chen, Size Zheng, Luis Ceze, Arvind Krishnamurthy, Tianqi Chen, and Baris Kasikci. Atom: Low-bit quantization for efficient and accurate llm serving. In P. Gibbons, G. Pekhimenko, and C. De Sa, editors, *Proceedings of Machine Learning and Systems*, volume 6, pages 196–209, 2024.

Optimal Software Pipelining and Warp Specialization for Tensor Core GPUs

Rupanshu Soi*† *Stanford University*
Rohan Yadav*† *Stanford University*
Fredrik Kjolstad *Stanford University*
Alex Aiken† *Stanford University*
Maryam Mehri Dehnavi *NVIDIA*
Michael Garland *NVIDIA*
Michael Bauer *NVIDIA*

Abstract

GPU architectures have continued to grow in complexity, with recent incarnations introducing increasingly powerful fixed-function units for matrix multiplication and data movement to accompany highly parallel general-purpose cores. To fully leverage these machines, software must use sophisticated schedules that maximally utilize all hardware resources. Since realizing such schedules is complex, both programmers and compilers routinely employ program transformations, such as software pipelining (SWP) and warp specialization (WS), to do so in practice. However, determining how best to use SWP and WS in combination is a challenging problem that is currently handled through a mix of brittle compilation heuristics and fallible human intuition, with little insight into the space of solutions. To remedy this situation, we introduce a novel formulation of SWP and WS as a joint optimization problem that can be solved holistically by off-the-shelf constraint solvers. We reify our approach in Twill, the first system that automatically derives optimal SWP and WS schedules for a large class of iterative programs. Twill is heuristic-free, easily extensible to new GPU architectures, and guaranteed to produce optimal schedules. We show that Twill can rediscover, and thereby prove optimal, the SWP and WS schedules manually developed by experts for Flash Attention on both the NVIDIA Hopper and Blackwell GPU architectures.

1 Introduction

Driven by the insatiable appetite of machine learning for performance, recent GPUs continue to expand the scale and capabilities of fixed-function units for matrix multiplication (GEMM), called Tensor Cores (TCs) on NVIDIA GPUs, and bulk data movement. As these fixed-function units become more powerful, the relative performance ratios of data movement and FLOPS between general-purpose and fixed-function matrix units change dramatically, often by multiplicative factors. Additionally, the interfaces for targeting fixed-function units routinely undergo significant alterations across generations, ranging from modifications to where data can be placed, to how many threads are required to issue operations, and even to the model of asynchronous execution. Consequently, every program targeting TC GPUs may have a different optimal *schedule* for each architecture generation to fully leverage the available general-purpose and fixed-function units [26, 32, 38].

A common solution to this performance portability problem for iterative programs is *software pipelining* (SWP) [6, 7, 24]. SWP permits programmers to write a simple loop to describe a computation, which a compiler can then automatically transform to exploit instruction-level parallelism within and across iterations to derive a schedule that maximizes utilization of all the functional units. Importantly, when compiling the same loop for different machines, the compiler will tailor a custom schedule based on the underlying constraints for each architecture to ensure that the same program runs efficiently in all settings [6]. To realize a particular SWP schedule, the compiler must synthesize a new loop, but the requirements imposed by new TC GPUs, such as the need for many threads to cooperatively issue TC operations [2, 3], preclude the use of a standard sequential loop to express most schedules.

To realize SWP schedules for TC GPUs, both compilers [5, 13, 25, 35–37] and developers [27, 33] have adopted *warp specialization* (WS) [10, 11] as a programming paradigm. Instead of the standard data-parallel style of GPU programming that assigns the same computation to each thread, in WS, subsets of threads called *warps* collaboratively execute different parts of a program, exchanging data through common memories and synchronizing where necessary. While WS is a necessary program transformation for achieving SWP schedules for TC GPUs, it comes with performance trade-offs (e.g., extra communication and synchronization) that obscure how best to deploy it in practice.

Currently, approaches for both SWP and WS on TC GPUs are derived either by human intuition [27, 32, 38] or through compiler heuristics [13, 35, 37]. Moreover, the interaction

*Equal contribution.
†This work was done at NVIDIA.

between SWP and WS is poorly understood, with no technical framework for reasoning about the optimality of combined solutions. The hazards of this situation can be observed in the case of Flash Attention [17, 18], where a year transpired between the release of Hopper and the development of Flash Attention 3 [32], which proposed a custom SWP schedule and WS strategy for Hopper, wasting countless compute cycles on critical inference workloads in the interim.

To rectify this situation, we show that the problems of determining the best SWP schedule and WS strategy can and should be solved simultaneously. We observe that determining a SWP schedule to maximize resource utilization can be reduced to a traditional *modulo scheduling* problem [24,31], that can be solved *optimally* [21,34], yielding the maximum possible throughput for any given machine model. We then explain why WS is a dependent program transformation that must be jointly considered to compute realizable SWP schedules for recent TC GPU architectures. We extend the approach of modulo scheduling to formulate the problem of determining an optimal SWP schedule and WS strategy as a unified constraint satisfaction problem that can be solved holistically by off-the-shelf Satisfiability Modulo Theories (SMT) solvers [9].

To demonstrate our approach, we introduce Twill[1], a system that automatically discovers optimal SWP schedules and WS strategies from high-level, tile-based descriptions of loops with simple control flow. Twill generates optimal schedules for different GPU architectures simply by altering machine-specific constraints corresponding to the easily quantifiable costs of various GPU operations. In contrast to all existing systems that perform SWP and WS of which we are aware [13, 35–37], our implementation is heuristic-free, easily extensible to new GPU architectures, and guaranteed to yield an optimal schedule for a large and important class of iterative programs (i.e., singly-nested loops without additional control flow). The specific contributions of this work are:

- A mapping of the problem of constructing a SWP schedule for TC GPUs to modulo scheduling.
- A constraint-based formulation of SWP and WS as a joint optimization problem.
- Twill, a system that implements our approach and discovers optimal SWP schedules and WS strategies for real programs.

We evaluate Twill by applying it to the Flash Attention algorithm and derive schedules for both the NVIDIA Hopper and Blackwell architectures. We show that Twill can automatically generate schedules that match the proposed algorithms in Flash Attention 3 [32] and Flash Attention 4 [38] from a high-level description of attention. We then implement these schedules and show that the resulting programs can come within 1% of the performance of hand-tuned implementations from libraries like cuDNN and Flash Attention.

[1]Twill is a textile weave that interlaces warp and weft threads in a distinctive diagonal pattern.

2 Background on GPU Architecture

NVIDIA GPUs consist of a number of independent processors called streaming multiprocessors (SMs). Each SM contains independent functional units to support general-purpose floating-point or integer arithmetic, as well as load and store units for memory operations. SMs also contain several kinds of memory including a register file for storing thread-local data, a software-managed scratchpad called *shared memory* that can be accessed by all threads on an SM, and an off-chip *global memory* that can be accessed from any SM.

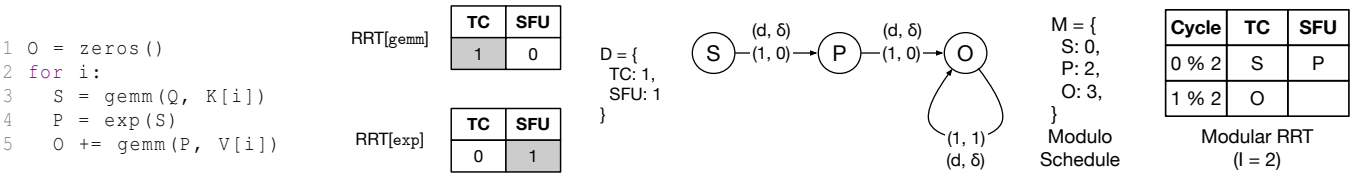
An SM executes groups of 32 threads called *warps*. Warps execute in a *single instruction, multiple threads* (SIMT) model: every thread in a warp has its own instruction stream, but only a subset of threads that agree on a common instruction to execute will issue in a given cycle, while the remaining threads are masked off and prevented from executing. Each thread executes *in-order*: if a thread's next instruction is blocked on an unresolved dependency or synchronization, it cannot issue any further instructions. Hopper and Blackwell SMs have four *execution contexts* that can host active warps, so at most four warps can issue instructions each cycle. When there are more warps than execution contexts, the SM's *warp scheduler* dynamically selects up to four ready warps from which to issue instructions.

Modern GPUs augment this architecture by incorporating asynchronous accelerators for operating on entire tiles of data. Tensor Cores (TCs) for accelerating the performance of GEMM operations were first introduced in the Volta architecture [4]. More recently, the Hopper architecture [3] introduced the Tensor Memory Accelerator (TMA) unit for asynchronously moving tiles of data between global and shared memory. Since they operate over large tiles, a single TMA or TC instruction may execute for thousands of clock cycles, in stark contrast to most floating-point or integer arithmetic instructions which often only execute for tens of cycles.

The Blackwell architecture [2] is similar to Hopper, but with two important enhancements. The Blackwell TC supports larger tile sizes with higher throughput, and each SM contains a new kind of memory called Tensor Memory where inputs to the TC may be sourced and accumulators stored. Performing general computations on Blackwell TC accumulators requires explicit data movement from the Tensor Memory into the register file. While conceptually straightforward, these modifications induce large perturbations in how programs must be structured for peak performance on Blackwell.

3 Scheduling for Tensor Core GPUs

Software pipelining (SWP) [7] reorders the operations in a sequential program to maximize utilization of the available functional units by exploiting instruction-level parallelism (ILP) within and across loop iterations. We use modulo scheduling [24,31] to compute pipelined SWP schedules, and

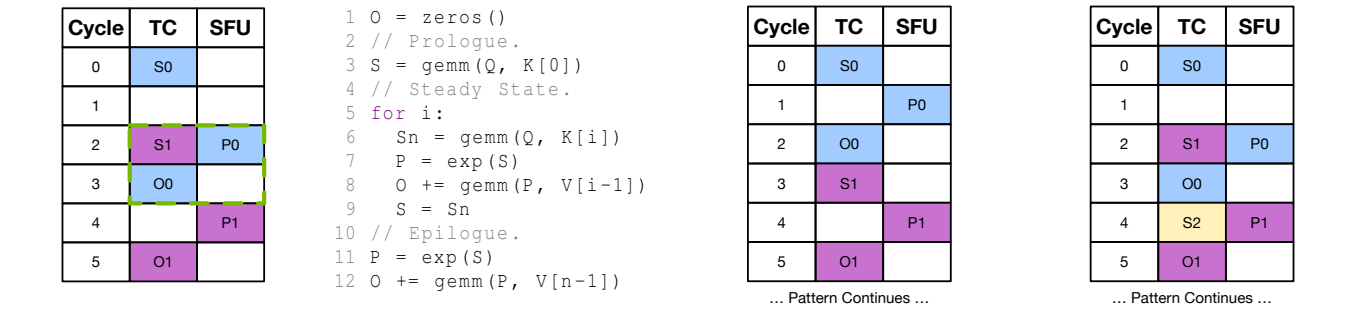


(a) Simplified Flash Attention pseudocode for 1 SM.

(b) Instruction RRTs and Hopper machine description.

(c) Loop dependence graph representation of Figure 1a.

(d) Valid modulo schedule and modular RRT with $I = 2, L = 4$.



(e) $\lceil L/I \rceil$ copies of Figure 1d's schedule placed I cycles apart.

(f) Code generated from Figure 1e with a steady-state loop.

(g) In-order execution, achieving $1/3$ its. per cycle.

(h) Pipelined execution, achieving $1/2$ its. per cycle after prologue.

Figure 1: Modulo scheduling a simplified Flash Attention expressed in a tile-based manner. The machine costs are for Hopper, where `GEMM` and `EXP` on a tile have roughly the same cost. Modulo scheduling recovers the Flash Attention 3 [32] pipeline.

Section 3.1 introduces standard terminology using the running example shown in Figure 1. The figure shows how to pipeline a simplified version of Flash Attention [18]. The code (Figure 1a), instruction cost information (Figure 1b), and a dependency graph (Figure 1c) go through two intermediate steps (Figures 1d and 1e) before yielding the pipelined code in Figure 1f. In this example, SWP is necessary for maximal TC utilization because the program exposes the latency of the exponential due to the data dependence on the result prior to the second matrix multiplication (Figure 1g). However, as we discuss in Section 3.2, it may not be possible on recent architectures for a single thread of execution to manifest this pipelined schedule. To address this issue, we describe in Section 3.3 how to realize an optimal pipelined schedule by distributing operations across multiple executing threads using WS, and the compilation challenges it induces.

3.1 Modulo Scheduling

Modulo scheduling [21,24,31] is a popular class of algorithms for SWP that transforms a given loop such that it achieves maximum throughput (iterations finishing per unit time) while simultaneously abiding by the data dependencies of the input program and capacities of the underlying machine. In modulo scheduling, the input loop is described as a dependence graph $G = (V, E)$ where V is the set of instructions in the loop and E defines dependencies between the instructions. We assume that G describes a singly-nested loop without control flow. In Twill, each instruction represents a tile-level operation that utilizes resources from the entire SM; dependence graphs of this form can be extracted from standard tile-based programming models (see Section 5).

Associated with each $v \in V$ is a *resource reservation table* (RRT) that indicates the functional unit(s) used during the execution of v. RRT$[v]$ is a 2-D integer array where each column corresponds to a kind of functional unit, each row corresponds to a clock cycle in the execution of v, and the entry gives the number of instances of f that are occupied by v at that clock cycle of its execution. The total number of instances of f on the machine, called the capacity of f, is given by a machine description D. RRTs for each operation in the running example and a machine description D are shown in Figure 1b; both instructions take a single cycle to execute, but they occupy different functional units. For simplicity we assume each functional unit can be scheduled and execute an instruction every cycle; on a real machine, functional units often take longer to execute some instructions than others and have limits on how often they accept new instructions. For TC GPUs that operate on large tiles of data, the abstraction of an RRT is still valid at a coarser granularity of computation, where a "cycle" might represent a coarser granularity of time. We describe how to cope with the problem of determining time granularity in Section 5.2.

Every edge $e \in E$ is a tuple (u, v, d, δ), where u is the source and v is the sink of the data dependence. The *clock cycle delay* $d \geq 0$ indicates that v must be issued at least d cycles after

u is issued. Finally, $\delta \geq 0$ is the *iteration delay*, indicating that the instance of v from loop iteration i must be issued at least d cycles after the instance of u from iteration $i-\delta$. For example, an edge with $\delta = 1$ indicates that the dependence between u and v is *loop carried* to the very next iteration. The dependence graph for the attention example is shown in Figure 1c, where the edges from S and P have $d = 1$ (from the corresponding RRTs) and $\delta = 0$, while the loop-carried dependence on O has $\delta = 1$. To simplify the exposition, we use variable names (which are unique) instead of instruction names in the figures. Together, the RRTs of each instruction (Figure 1b) and the dependence graph G (Figure 1c) are the inputs to modulo scheduling.

When run on a given loop, modulo scheduling finds an *initiation interval* I and *modulo schedule* M, from which the pipelined loop is constructed. I corresponds to the rate at which new loop iterations are issued (or finished), and is thus inversely related to the throughput; $I = 1$ is the minimum possible and means that a new loop iteration is launched every cycle. M maps each instruction to the clock cycle at which it must be issued, counting from the beginning of its iteration (hence the name "modulo"). If $M(v) = k$, then v will be issued at clock cycles $k, I+k, 2I+k, \ldots$ in the pipelined loop. A choice of M and I is valid when it ensures that the pipelined loop satisfies all dependence edges in E and keeps all functional units within capacity.

A valid modulo schedule for the running example is shown in Figure 1d. It is accompanied by the *modular RRT*, a data structure similar to the RRT that indicates the functional unit usage in the steady state of the pipelined loop. In the modulo schedule, both P and O were delayed one cycle later than an optimal schedule for just a single iteration. $M(P) = 1$ and $M(O) = 2$ would be valid per the data dependencies but invalid per the modulo RRT because O would conflict with S for the TC since $0 \equiv 2 \pmod 2$. A variety of algorithms have been developed to construct M and I for a given dependence graph G, including greedy [24, 31] algorithms and optimal [21, 34] algorithms that leverage Integer Linear Programming ($\mathbb{Z}$LP). Optimal algorithms yield modulo schedules with the smallest possible I, and thus the highest possible throughput.

Once a modulo schedule M and initiation interval I have been found, they can be used to synthesize a program that executes the schedule. A standard way [6] to realize a modulo schedule is with a sequential loop that consists of: 1) a *prologue* that primes the loop, 2) a *steady state* containing the repeatedly executing loop body, and 3) an *epilogue* that drains the loop. To construct these three components, we let L be the *length* of M, i.e., the number of cycles in M. The software-pipelined loop is constructed by overlapping $\lceil L/I \rceil$ copies of M, each copy offset by I cycles from the previous copy. The pipelined loop can then be read off from this staggered schedule: the first $(\lceil L/I \rceil - 1) \cdot I$ cycles correspond to the prologue, the next I cycles correspond to the looping steady state, and the remaining cycles are the epilogue that drains the pipeline. Placing the modulo schedule for the running attention example is shown in Figure 1e, where blue and purple correspond to different copies of the schedule, and the dashed box gives the steady state. Once the schedule has been constructed, it can be used to generate code. Generated pseudocode for the running example is shown in Figure 1f. An in-order execution as written in Figure 1a completes 1 loop iteration every 3 cycles (Figure 1g), while the pipelined execution completes 1 iteration every 2 cycles after the prologue (Figure 1h).

Applying modulo scheduling to the running Flash Attention example yields the exact pipeline developed by experts in Flash Attention 3 [32] from only a high-level description of the algorithm and machine.

3.2 Code Generation Challenges

For GPUs prior to Hopper, the modulo scheduled tile-level program in Figure 1f could be lowered to single-threaded code executing in a SIMT fashion, where each data-parallel thread would execute the same program on disjoint subsets of the tile of data. Unfortunately, most modulo schedules computed for the functional units of a Hopper or a Blackwell GPU cannot be realized as a single-threaded program in the standard style. There are four confounding factors that regularly thwart single-threaded code generation. First, and most commonly, the scale of TCs in recent GPUs requires multiple warps to cooperatively issue large GEMM computations. Most GPU compilers are built on sequential intermediate representations (e.g., LLVM and PTX) and are therefore ill-equipped to generate code for cooperative warps.

A second issue stems from the propensity of modulo schedules to increase the working set size of computations to keep live variables from multiple iterations of a program on-chip concurrently. The combination of this natural memory pressure of modulo schedules in conjunction with the larger quantities of data needed to feed bigger TCs makes it challenging to keep the working set of a computation on-chip. Most GPUs have a strict upper bound of 255 registers per thread and spilling data out of the register file often incurs a significant performance penalty as the spilled data is unlikely to remain in cache. Many single-threaded realizations of modulo schedules for recent GPUs struggle to achieve peak performance due to the cost of spilling.

A third complication is that some operations, such as transfers through the multi-tiered GPU memory hierarchy, have high variability in their execution times. While traditional modulo scheduling suggests using an upper bound on such costs [6], there is an implicit assumption that this upper bound is roughly the same order of magnitude as other operations in the schedule. However, operations like TMA transfers that move tiles of data between global and shared memory may have more than an order of magnitude difference between the fastest and slowest possible execution times for the same transfer. While these operations are performed asynchronously, the

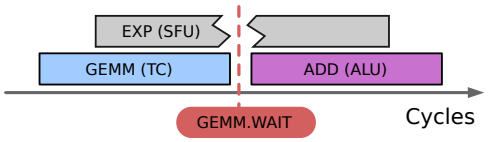


Figure 2: Visualization of three operations using different functional units scheduled on the same warp. The blocking sync after `GEMM` interrupts the concurrent issue of `EXP`.

high dynamic range of potential latencies can foil attempts to place synchronization instructions during code generation. In particular, overestimation results in under-utilization, while underestimation results in pipeline stalls during execution.

Finally, the fixed-function units on Hopper and Blackwell have asynchronous interfaces that require explicit blocking synchronization to consume results. Since GPU threads have *in-order* instruction issue, blocking synchronization interrupts the instruction issue of any concurrently scheduled operations on the same warp. An example is in Figure 2, where a modulo schedule requires an addition to consume the result of a `GEMM`, while executing an independent `EXP` concurrently. Note that all three occupy independent functional units. Executing all operations from the same warp suffers degraded performance as the `GEMM.WAIT` interrupts instruction issue for `EXP`.

Each of these four factors influence code generation for modulo schedules to differing degrees depending on the properties of the particular program. Combating them in practice necessitates a new approach to code generation.

3.3 Warp Specialization

Warp specialization [10,11] (WS) is a programming paradigm that offers a solution to the code generation challenges encountered when trying to realize a modulo schedule on GPUs. A warp-specialized program assigns operations to different warps while ensuring that the warps cooperatively execute the entire computation. WS is possible due to the hierarchical grouping of threads within an SM. Since warps are SIMT, a warp's performance is maximized when all threads in the warp issue the same instruction, and minimized when the threads *diverge*. However, unlike threads, warps are independent from each other and pay no penalty for control divergence.

While WS is commonly viewed as a separate optimization, our insight is that WS is useful precisely because it directly addresses all of the code generation problems for SWP presented in Section 3.2. First, WS naturally reasons about warps cooperating and can ensure that subsets of warps cooperatively issue TC operations. Second, by splitting the computation across many warps, the computation can access the register resources of many threads, yielding greater flexibility for fitting the large working sets demanded by SWP on-chip. Third, variable-latency operations can be separated onto dedicated warps, instead of needing to be statically scheduled alongside (and interfering with) fixed-latency operations. Finally, splitting operations across warps allows some warps to issue instructions while others are blocked on synchronization.

Since WS addresses the challenges of generating code for SWP, it has gained significant adoption by both compilers [5, 13, 25, 35–37] and programmers [27, 32, 38]. However, there are two complications. First, WS is not free and comes with trade-offs involving communication of data between warps and synchronizing access to both shared data and hardware resources. Currently, all automated approaches that we are aware of depend upon ad-hoc heuristics to decide these trade-offs [13,35,37]. These heuristics often involve canonical warp "roles" (e.g., loader and compute warps), or treat warps as parallel "agents" to whom work can be dispatched. Without a technical framework for understanding the optimality of these heuristics, it is difficult to know if programs generated using them are achieving peak performance.

The other significant complication to this approach is that, even with WS, it might not be possible to synthesize a program that actually achieves a modulo schedule. For example, despite spreading a computation across as many warps as possible in an SM, it still might be impossible to fit the working set demanded by a modulo schedule in the register file. Consequently, treating the computation of the modulo schedule and code generation using WS as two independent steps in compiling a program can lead to sub-optimal code in practice. To ensure that a modulo schedule can actually be realized, the constraints for code generation using WS need to be directly incorporated into the optimization process in conjunction with those for traditional modulo scheduling.

4 Joint Optimization Problem

We now demonstrate how to solve the optimization problem of finding a modulo schedule and a WS strategy simultaneously. The joint optimization problem is to take an input program $G = (V, E)$ and produce a modulo schedule with the minimum initiation interval along with a WS strategy capable of realizing the schedule. The result is a modulo schedule M^*, initiation interval I^* and warp assignment A^*, where A^* is an assignment of every $v \in V$ to a warp (or warps).

We approach this problem in two main steps. First, modulo scheduling is used to derive an initial modulo schedule M and initiation interval I that achieve the maximum throughput while respecting the data dependence and functional unit constraints. Then, M and I are used to seed a system of constraints that are supplemented with constraints for WS. The system of constraints can then be solved by an SMT solver to discover an M^* and A^* with the same I that respect the requirements of both modulo scheduling and WS.

We use M and I to define an initial straight-line program Q over which the constraint system is formulated. Q is obtained using the code-generation procedure of modulo scheduling from Section 3.2. Recall that $\lceil L/I \rceil$ copies of M are overlapped, each offset I cycles from the previous, resulting in a prologue, steady state and epilogue. Our insight is that we can

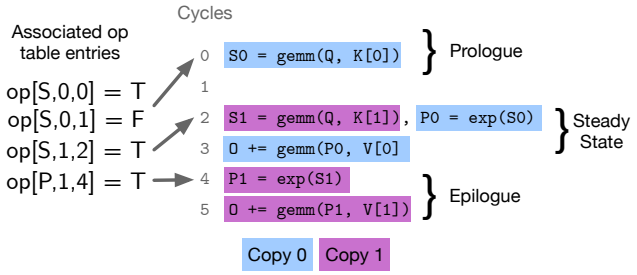


Figure 3: Straight-line code analyzed by Twill's joint formulation. Sample op table entries are to the left.

$$\forall v,i \quad \sum_t \mathrm{op}[v,i,t] = 1 \qquad \text{(UNIQUENESS)}$$

$$\forall v,i \in [1,\lceil L/I \rceil),t \quad \mathrm{op}[v,0,t] \Rightarrow \mathrm{op}[v,i,t+i\cdot I] \qquad \text{(CONSISTENCY)}$$

$$\forall v,i,t \quad t+\mathrm{cycles}(v) > T \Rightarrow \neg\mathrm{op}[v,i,t] \qquad \text{(COMPLETION)}$$

$$\forall i,t,(u,v,d,\delta) \in E,\ t' \in [0,t+d) \quad \mathrm{op}[u,i,t] \Rightarrow \neg\mathrm{op}[v,i+\delta,t'] \qquad \text{(DEPENDENCE)}$$

$$\forall t,f \quad \sum_{v,i,c \in [0,\mathrm{cycles}(v))} \mathrm{op}[v,i,t-c]\cdot \mathrm{RRT}[v][f,c] \le \mathrm{cap}(f) \qquad \text{(CAPACITY)}$$

Figure 4: Constraints enforcing a valid modulo schedule.

assume that the steady state executes exactly once, allowing us to treat the three parts as a straight-line program with length T. This assumption is sound, because it is permissible for the steady state to execute exactly once, and complete, because steady state executions are identical, and greatly simplifies the constraints by obviating the need for any loop analysis. An example program Q derived from the modulo scheduled example in Figure 1 is shown in the right half of Figure 3.

4.1 Modulo Scheduling with Constraints

The goal of the joint problem is to find a modulo schedule M^* that may be distinct from M but achieves the same initiation interval. To do so, Twill must be able to modify M, necessitating a *reimplementation* of modulo scheduling in the form of SMT constraints. Instead of modifying M directly, this reimplementation allows the modification of Q to Q^* such that Q^* has the same initiation interval and length as Q, but may be the result of a different modulo schedule. The constraints to do so are given in Figure 4. We define a 3-D boolean array such that $\mathrm{op}[v,i,t]$ indicates that operation $v \in V$ of iteration $i \in [0,\lceil L/I \rceil)$ was scheduled at clock cycle $t \in [0,T)$ in Q^*. The uniqueness constraint enforces that every operation is scheduled exactly once. The consistency constraint ensures that Q^* is obtainable from some modulo schedule. The completion constraint requires all operations to finish before the last clock cycle T of Q^*. Finally, the dependence and capacity constraints function identically to the corresponding constraints in modulo scheduling. We elide guards for brevity, but assume that constraints referencing $i \notin [0,\lceil L/I \rceil)$ are not emitted. Additionally, references to $t \notin [0,T)$ of op are mapped to false. A satisfying assignment of op induces a valid Q^* and M^*, where $M^*(v) = t$ iff $\mathrm{op}[v,0,t]$.

$$\forall t,m \quad \sum_{v,i} \mathrm{live}[v,i,t]\cdot \mathrm{footprint}(v,m) \le \mathrm{capacity}(m) \qquad \text{(MEMORY CAPACITY)}$$

$$\forall v \quad (\exists (v,u,d,\delta>0) \in E) \Leftrightarrow \mathrm{live}[v,\lceil L/I \rceil - 1,T] \qquad \text{(INIT)}$$

$$\forall v,i,t \quad (\mathrm{live}[v,i,t] \wedge \mathrm{op}[v,i,t]) \Rightarrow \neg\mathrm{live}[v,i,t-1] \qquad \text{(LIVEPROP-1)}$$

$$\forall v,i,t \quad (\mathrm{live}[v,i,t] \wedge \neg\mathrm{op}[v,i,t]) \Rightarrow \mathrm{live}[v,i,t-1] \qquad \text{(LIVEPROP-2)}$$

$$\forall v,i,t \quad (\neg\mathrm{live}[v,i,t] \wedge \bigvee_{(v,u,_,\delta)\in E} \mathrm{op}[u,i+\delta,t]) \Rightarrow \mathrm{live}[v,i,t-1] \qquad \text{(DEADPROP-1)}$$

$$\forall v,i,t \quad (\neg\mathrm{live}[v,i,t] \wedge \bigwedge_{(v,u,_,\delta)\in E} \neg\mathrm{op}[u,i+\delta,t]) \Rightarrow \neg\mathrm{live}[v,i,t-1] \qquad \text{(DEADPROP-2)}$$

Figure 5: Memory Allocation Constraints.

4.2 Memory Aware Constraints

To ensure that the working set of a modulo schedule can remain on-chip, we introduce additional constraints that enforce memory capacity, narrowing the set of valid instances of Q^*. These constraints assume that Q^* is in static single assignment form (SSA) [8, 16], where variables are defined once and have lifetimes that end at the last use. We define a second boolean array $\mathrm{live}[v,i,t]$, where $\mathrm{live}[v,i,t] = 1$ when the result of the i'th instance of v is live at time t. Using live, defining the memory capacity constraint is straightforward and shown in Figure 5, where the memory footprints and capacities are defined by the input graph and machine model. The non-trivial component of memory capacity is setting up the constraints defining $\mathrm{live}[v,i,t]$; this component cannot be discharged to an external analysis, because changes to Q^* affect when different values are live. These constraints are shown in Figure 5, and are derived from classic backwards dataflow algorithms for liveness [6]. There are three groups of liveness constraints: initial conditions, propagation of liveness, and propagation of deadness. Initially, only the loop-carried results of the last copy of each instruction are live at the last clock cycle of Q^*, i.e. at time T. Liveness is then propagated backwards through the straight-line program: if an operation's result is live at time t, it is live at time $t-1$ unless it was scheduled at time t. Deadness is propagated in a similar way, where if the result of an operation is dead at time t, it stays dead at time $t-1$ unless a user of the operation is scheduled at time t.

$$\forall v \quad \sum_{w} \text{opw}[v,w] = 1 \qquad \text{(WARP UNIQUENESS)}$$

$$\forall v \quad \text{variable_latency}(v) \Leftrightarrow \text{opw}[v, W_{vl}] \qquad \text{(VARIABLE LATENCY)}$$

$$\forall t, w \quad \sum_{v,i} \text{live}[v,i,t] \cdot \text{opw}[v,w] \cdot \text{regs}(v) \leq \text{reg_limit}() \qquad \text{(REGISTER LIMIT)}$$

$$\forall (u,v,d,\delta) \in E, t, i, w, w' \neq w, s \in [0, \text{spillcost}(u))$$
$$\text{op}[u,i,t] \wedge \text{opw}[u,w] \wedge \text{opw}[v,w'] \Rightarrow$$
$$\neg \text{op}[v, i+\delta, t+d+s] \qquad \text{(CROSS-WARP SPILLS)}$$

$$\forall (u,v,_,_) \in E, t, w, i, o \neq v$$
$$\text{op}[v,i,t] \wedge \text{opw}[v,w] \wedge \text{blocking}(u,v) \Rightarrow$$
$$\forall i', t' \in [t - (\text{cycles}(o) - 1), t],\ \neg(\text{op}[o,i',t'] \wedge \text{opw}[o,w]) \qquad \text{(CONCURRENCY)}$$

Figure 6: Warp Assignment Constraints.

4.3 Warp Assignment Constraints

Finally, we introduce constraints that influence WS decisions, shown in Figure 6. We define another boolean array $\text{opw}[v,w]$, indicating whether operation v has been assigned to warp w; opw directly defines the final warp assignments A^*. Similar to op, we define a uniqueness constraint, stating that every operation must be assigned to exactly one warp. This constraint is naturally extended to operations that span multiple warps, such as the warp-group level operations on Hopper and Blackwell.

The first constraint is the assignment of operations with variable and statically-unknown latencies. These operations are assigned to a designated warp W_{vl}, separating them from the scheduling and placement of instructions with statically-known and fixed latencies. The second constraint enforces the register limit on each warp; the constraint is similar to the capacity constraint in Figure 5, but scopes the summation to each warp. This constraint forces a partitioning of V to stay within register limits. However, data in the registers of warp w cannot be accessed on warp w' unless the data is transferred (or *spilled*) through the shared memory. When data is spilled across warps, a valid schedule must include the cost of the transfer when scheduling the consuming operation; this additional delay is captured by the *cross-warp spill* constraint. We assume the cost of cross-warp communication is provided as an annotation on nodes in G. This constraint forces a trade-off between memory capacity, latency and the costs of cross-warp spills. The final constraint is the *concurrency* constraint, which captures the effects of blocking synchronization interrupting the issue of concurrently scheduled operations. Certain edges (u,v) are designated as *blocking* edges, indicating that blocking synchronization is required by v before the results of u are accessible. The concurrency constraint states that if an operation v, assigned to warp w and scheduled at time t, requires blocking synchronization from an incoming dependence, then no other operation can be scheduled to run on w when v starts; this forces concurrent operations to either be placed onto different warps or scheduled at different times. While elided for brevity, the implementation includes a similar constraint for cross-warp spills, as transferring register data through shared memory requires blocking synchronization.

The entire system of constraints captures all the challenges presented in Section 3 that can make a desired modulo schedule impossible to implement in practice for recent GPUs. By describing the space of solutions holistically with a unified system of general constraints, we force the constraint solver to reckon with all aspects of the scheduling problem simultaneously. Furthermore, as new machines are released, constraints can easily be added, removed, and modified to reflect the challenges presented in future architectures. Therefore our approach provides a flexible framework for finding optimal solutions to the joint problems of modulo scheduling and WS.

5 Implementation

To reify our approach, we introduce Twill, a system that computes joint SWP and WS strategies for Triton [35] programs. Twill extracts dependence graphs from a mid-level Triton intermediate representation (IR) called TTGIR, which is a tile-based, SSA IR with arithmetic on tiles and explicit data movement. These graphs serve as the input program to Twill's optimization process as described in Section 4. Users of Twill must also inform it of the target GPU architecture. The target architecture is used to estimate costs of instructions and data movement (discoverable via documentation [1] or direct measurement), and to enable architecture-specific modeling, such as declaring available memories or denoting operations that require blocking synchronization.

As described in Section 4, Twill computes an initial modulo schedule by formulating it as an $\mathbb{Z}$LP problem [34] and dispatching it to the CBC solver [20]. Twill then uses the resulting modulo schedule and initiation interval to seed the system of SMT constraints. Twill discharges SMT queries to the quantifier-free linear integer arithmetic (QFLIA) theory of the Yices2 SMT solver [19]. Once a modulo schedule and warp assignment have been discovered, Twill uses standard code generation techniques to emit software-pipelined IR, annotating each instruction with the warp (or warps) that should execute it. The resulting IR can either be consumed by downstream compilers that implement a specified WS strategy (e.g., Tawa [13] or Cypress [37]), or used by an expert as reference for a manual implementation.

Algorithm 1 Twill's Search Procedure

```
1:  procedure TWILL(G)
2:      I ← 0
3:      while true do
4:          I ← I + 1
5:          M ← OPTIMAL-MODULO-SCHEDULE(G, I)
6:          if M = failure then
7:              continue
8:          L ← LEN(M)
9:          while ⌈L/I⌉ = ⌈LEN(M)/I⌉ do
10:             (M*, A*) ← SWP-AND-WS(G, M, I, L)
11:             if (M*, A*) = failure then
12:                 L ← L + 1
13:                 continue
14:             return (M*, I, A*)
```

5.1 Handling Unsatisfiability

Twill must handle the case when the constraint system is unsatisfiable. Unsatisfiability indicates that some resource limitation or concurrent interaction is preventing a modulo schedule with the initial initiation interval I from being achieved. Similar to standard approaches in modulo scheduling [24], if the constraints are unsatisfiable, Twill uses modulo scheduling to find a new candidate schedule M' with initiation interval $I+1$, which is then used to seed the system of constraints for another solution attempt. By searching monotonically from the smallest possible initiation interval, we ensure discovery of the highest throughput schedule that solves the constraints. In addition to searching over candidate values of I, Twill also searches over increasing values of L (the total schedule length) that do not affect $\lceil L/I \rceil$ at a given I. This search is depicted in Algorithm 1.

5.2 Cost Normalization

A critical component of Twill's implementation is *cost normalization*, which renders the optimization problems solved by Twill tractable. Consider the cycle counts computed from publicly available documentation [1] for a 128x128x128 GEMM on Hopper—roughly 1000 cycles. A reasonable dependence graph $G = (V, E)$ may contain many instructions with similar cycle counts. Optimal algorithms for modulo scheduling and Twill's joint formulation are not just exponential in $|G|$, but exponential in $\sum_{(u,v,d,\delta)\in E} d$. Therefore, directly using estimated cycle counts results in intractable $\mathbb{Z}$LP and SMT problems.

Our solution relies on the observation that multiplying every cycle count in a SWP problem by a positive integer results in a new problem isomorphic to the original. It is not the cycle counts that matter, but the *ratios* between cycle counts. Therefore, we want to obtain new, smaller cycle counts whose ratios are as close as possible to ratios of the original cycle counts. Cost normalization encodes this intuition as an $\mathbb{Z}$LP problem (separate from the one for modulo scheduling).

Let the list of integers C be the original cycle counts. We want to derive a new list of integers C' such that $C[i]/C[j] \approx C'[i]/C'[j]$ for all i, j. This is formalized as a constraint by introducing a variable F that bounds the change in ratios. We also bound the sum of C' from above and below, the latter to avoid the degenerate solution of all zeros:

$$\forall i, j \quad -F \leq C[i] \cdot C'[j] - C[j] \cdot C'[i] \leq F$$

$$1 \leq \sum_i C'[i] \leq U$$

U is a user-defined integer parameter that controls the exact trade-off between resolution of costs and running time of the subsequent algorithm that actually calculates a modulo schedule. Smaller values of U result in lower resolution (i.e., larger change in ratios) and lower running times. Finally, the $\mathbb{Z}$LP objective is to minimize F. Twill uses the SCIP solver [22] to find solutions because it was considerably faster than the CBC solver [20] at such problems. In our experiments, we pick $U = 300$ and find that SCIP is able to find the global minima in under 500 ms in all cases. Early in the development of Twill, we relied on an ad-hoc approach that divided each cycle count by a fixed integer, followed by rounding to obtain integer values. We found it difficult to keep this approach consistent, even for the same program on different GPUs. In contrast, framing cost normalization as an $\mathbb{Z}$LP problem offers a principled solution and is necessary for discovering optimal schedules.

5.3 Variable Latency Optimizations

As discussed in Section 3.2, approximating the execution time of variable-latency operations with a high dynamic range can cause imprecision in static scheduling. While Twill offloads variable-latency operations onto separate warps so they may be dynamically scheduled, further optimizations are possible for variable-latency operations in the target loop dependence graph that have no incoming data dependencies. We refer to these variable-latency operations as *streaming* operations. On separate warps, streaming operations can run ahead of the main software pipeline and complete several iterations before their results are needed. To reflect this, we assign streaming operations zero latency in our cost models, allowing dependent operations with statically-known latencies to be precisely scheduled. We then expose the pipeline depths for these streaming operations as parameters that may be tuned by an external auto-tuning system; exposing such parameters for dynamic tuning is commonplace among existing TC GPU programming systems [5, 15, 27, 33, 35, 37]. Many critical compute-bound kernels contain streaming variable-latency operations (like TMA loads of input tiles), warranting optimization of this important case.

5.4 Limitations and Future Work

Currently, our implementation of Twill only supports singly-nested loops without additional control flow. This limitation could be lifted through the use of hierarchical reduction techniques from the software pipelining literature [24], which we leave as future work. While Twill derives an optimal pipeline with respect to the constraints presented so far, the tile size is not automatically determined by Twill, and must therefore be selected by a human or a higher-level auto-tuning system. As we will see in Section 6.4, the solution times of Twill's joint problem range from tens of seconds to a few minutes. Fast solution times are a non-goal of our approach; we trade-off solution times for an optimality guarantee. We envision a system like Twill serving as a developer aid or as an offline optimization tool used prior to deployment, rather than being run continuously during interactive development. Despite this being a non-goal, in practice we find the solvers used by Twill to be highly optimized and capable of producing competitive search times as we will demonstrate.

6 Evaluation

We do a thorough evaluation on two important kernels from the machine learning literature, and show that Twill is able to derive SWP and WS strategies for *multiple generations* of NVIDIA GPUs automatically and from first principles. Twill is the only system we are aware of that is capable of doing so without heuristics tailored for each GPU generation. We then implement these strategies and show the achieved performance is competitive with hand-tuned implementations (Sections 6.2 and 6.3). Finally, we discuss the search times of Twill (Section 6.4).

6.1 Methodology

Kernel selection. We focus our evaluation efforts on compute-bound kernels that require utilizing both the TC and general-purpose functional units on the SM. Hence, we focus on the critically important forward and backward passes of Fused Multi-Head Attention, which have received significant human effort in developing SWP and WS strategies [32, 38]. The backward pass requires a significantly different loop structure than the forward pass and thus demonstrates Twill's flexibility in discovering different strategies.

Evaluation Platforms. We evaluate Twill on NVIDIA Hopper and Blackwell GPUs. We use a NVIDIA H100 SXM5 80 GB and a NVIDIA B200 180 GB. All experiments use CUDA 13.0. Search times were recorded on a single core of a Intel Xeon Platinum 8570.

Experimental Setup. Twill schedules dependence graphs extracted from Triton programs. Initially, we intended to use Twill within Triton and let Triton generate code from Twill's SWP and WS strategies. However, there are many additional (and orthogonal to Twill) critical decisions and optimizations that must be performed correctly to achieve high performance on modern GPUs. We found that Triton made many incorrect decisions during code generation, such as in memory allocation, data layout conversions or synchronization placement. As a result, Triton was either unable to successfully compile Twill's pipelines, or resulted in poorly performing code. To demonstrate the performance of schedules found by Twill, we instead "hand-compile" Twill's pipelines into CUDA C++. This process involved translating the pipelined and warp-annotated IR emitted by Twill into CUDA that implemented the specified strategy while allowing us to correctly make the remaining (and orthogonal) lowering steps. Automating all of these remaining steps in an optimal manner requires significant engineering and further research; as such, it is out of scope for this work.

6.2 Attention Forward Pass

The forward pass of FMHA has been the focus of significant optimization over multiple generations of GPU hardware through the Flash Attention (FA) algorithms [17, 18, 32, 38]. We focus on FMHA on Hopper and Blackwell, targeted by Flash Attention 3 (FA3) [32] and Flash Attention 4 (FA4) [38] respectively. For both architectures, novel SWP and WS strategies were required for peak performance and were discovered by humans months after each architecture was released. We demonstrate that Twill automatically rediscovers the same strategies across different architecture generations.

We started with the Triton tutorial [29] implementation of FA, a high-level implementation of FA without additional optimizations. A Triton program describes the decomposition of work for each SM. For Twill's joint approach to SWP and WS, we sub-tile the computation for the TC instruction sizes of the target hardware. This sub-tiling allows Twill to control scheduling and warp assignments at the appropriate level of granularity; Triton's IR only implicitly reasons about the decomposition of tiles onto multiple warps. We discuss these results in depth, focusing on each architecture in turn.

6.2.1 Hopper

FA3 [32] introduced two main techniques for optimizing FMHA on Hopper. The first technique was a software pipelining schedule for the main loop that extracts one GEMM into the loop's prologue so that latency of the exponential is not exposed (seen in Figure 1). The second technique was called *ping-pong scheduling*, which schedules the GEMM from one sub-tile while the exponential from another sub-tile executes, again increasing TC utilization. We show how Twill automatically derives both optimizations.

Figure 7 contains results for different implementations of

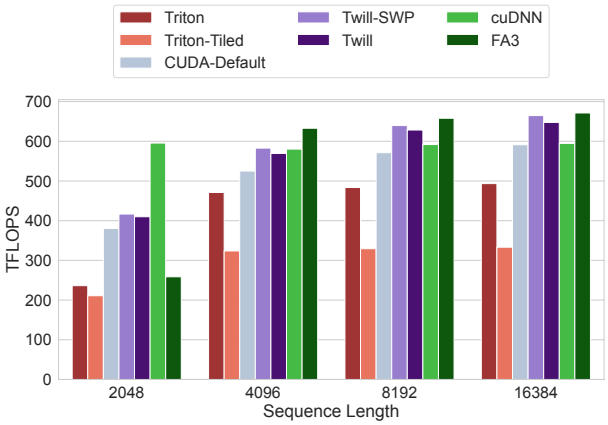


Figure 7: Hopper FP16 Non-Causal Forward Attention. (BATCH=4, NUM_HEADS=32, HEAD_DIM=128)

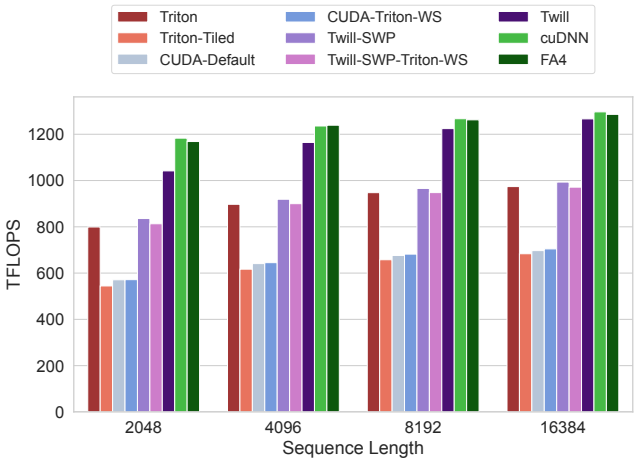


Figure 8: Blackwell FP16 Non-Causal Forward Attention. (BATCH=4, NUM_HEADS=32, HEAD_DIM=128)

forward FMHA on Hopper. The bars "Triton" and "Triton-Tiled" report the performance of Triton executing the tutorial implementation of FA and the sub-tiled version. As discussed in Section 6.1, Triton makes many suboptimal decisions during compilation. The "CUDA-Default" bar reports the performance of our "hand-compilation" of Triton IR into CUDA C++, which yields a moderate speedup over Triton. The first Twill result we discuss is the "Twill-SWP" bar, which just uses Twill's modulo scheduling (i.e., only running OPTIMAL-MODULO-SCHEDULE in Algorithm 1) to software pipeline the main loop and applies Triton's WS strategy, which puts loads from global memory on a separate warp than the rest of the computation. This results in a significant performance boost, coming within 1% of the official FA3 implementation at the 16384 sequence length. If algorithms like modulo scheduling were used within systems like Triton, this component of FA3 would not have needed human discovery. Finally, the "Twill" result uses Twill's joint formulation of SWP and WS (i.e., running the entirety of Algorithm 1). Twill's functional unit capacity constraints recover the ping-pong scheduling strategy described in FA3: since only one operation may use the TC at a time, Twill skews groups of warps so that one issues exponentials while the other uses the TC. Twill discovers the software pipeline used in FA3, but applies it only to one warp group, determining this to be sufficient to saturate the functional units. While the final implementation of this version performed slightly worse than Twill-SWP, the two initially performed equivalently (roughly 645 TFLOPS), but an orthogonal optimization (TMA multicasting) benefited the modulo-scheduled implementation more. On Hopper, multiple warps are interleaved dynamically to issue work into the TC and other functional units, naturally gaining ILP and lessening the burden on the quality of static instruction scheduling. We will see next that this is no longer true on Blackwell.

6.2.2 Blackwell

Blackwell requires substantially different SWP and WS strategies than Hopper due to a faster TC and a larger set of required synchronization operations (Tensor Memory loads and stores). The set of Blackwell results is shown in Figure 8.

As for Hopper, we include Triton results for the tutorial and sub-tiled versions. Our direct translation of Triton IR to CUDA performs worse than Triton on Blackwell because Triton's Blackwell backend heuristically applies the FA3 SWP strategy. When we apply modulo scheduling and Triton's WS strategy for Hopper, we match Triton's performance, but are far from the best implementations.

Twill's joint optimization of SWP and WS finds a strategy that performs significantly better than Triton and competitively with cuDNN and FA4. In fact, the discovered strategy is exactly the same as proposed by FA4 [38]! Twill's strategy combines a different software pipeline than Hopper with specific warp assignments and cross-warp communication. Twill's WS strategy is shown in Figure 9, using a visualization generated by Twill for the actual dependence graph obtained from Triton IR for this program. Twill places variable-latency operations (green) and TC GEMMs (pink) onto separate warps, softmax calculations for each sub-tile onto two different groups of warps (blue and orange), and accumulator rescaling operations for both sub-tiles onto a third group of warps (yellow).

This strategy does not comport to conventional warp roles like "loader" and "compute" warps. The reasoning behind this strategy is opaque without considering the associated software pipeline. The pipeline hoists the first GEMM for each sub-tile out of the loop like FA3. In the main loop, the pipeline schedules the exponential for one sub-tile during the GEMMs for the other sub-tile (like ping-pong scheduling), all while the sub-tile's accumulator is being rescaled. The rescaling is moved to a third group of warps because reading accumulators from Tensor Memory requires blocking

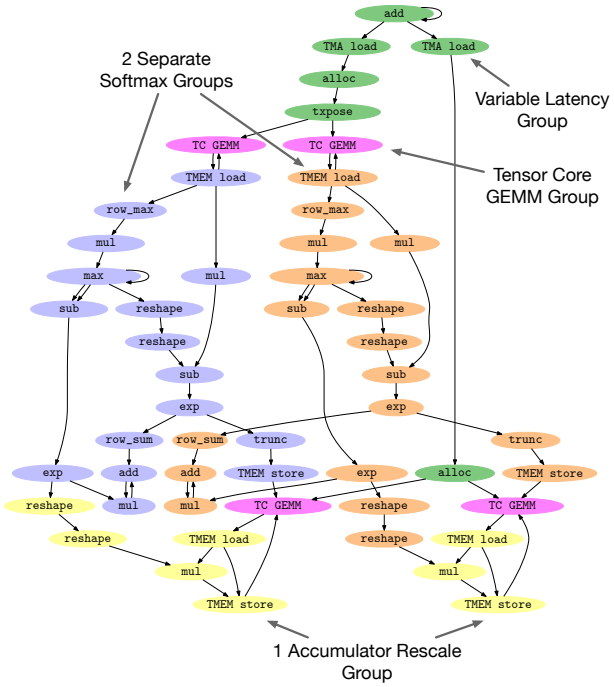


Figure 9: Twill's dependence graph showing the discovered forward WS strategy for Blackwell. Colors indicate instructions assigned to the same group of warps.

synchronization; placing it on the warp issuing exponentials disrupts the pipeline. However, moving the rescaling to a third group of warps incurs cross-warp communication, as it needs data produced by the softmax warps — Twill schedules around this latency and synchronization. Our implementation of this schedule comes within 2% of FA4 on the 16384 sequence length.

We now return to other kernel variants in Figure 8. Triton has heuristics to replicate the FA4 strategy by moving accumulator rescaling to a separate group of warps. Applying these heuristics to our purely software-pipelined implementation ("Twill-SWP-Triton-WS") or our default translation ("CUDA-Triton-WS") yields no benefit, showing that jointly considering SWP and WS is critical for peak performance.

The final compelling component of our Blackwell results are modifications which render the SMT problem unsatisfiable for the smallest I which makes the $\mathbb{Z}$LP succeed in Algorithm 1, forcing a search over larger values of I and L. These modifications include:

- Reducing the number of warps.
- Using the tutorial Triton implementation directly. Without sub-tiling, the instruction granularity is not fine enough for interleaving.
- Prohibiting cross-warp communication.

The variety of unsatisfiable configurations demonstrates the complexity of the search space for finding high performance SWP and WS strategies. Twill provides a technical framework for reasoning about and navigating this complex search space

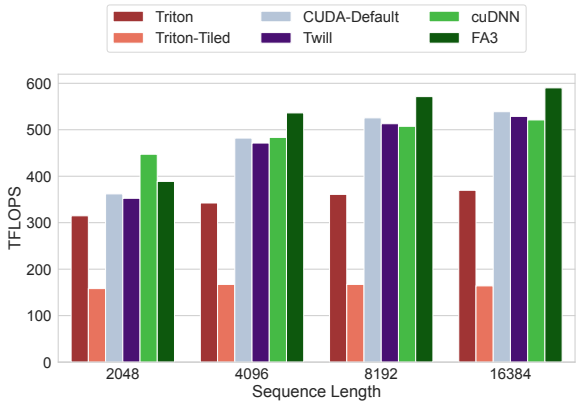


Figure 10: Hopper FP16 Non-Causal Backward Attention. `(BATCH=4, NUM_HEADS=32, HEAD_DIM=128)`

while guaranteeing optimal results.

6.3 Attention Backward Pass

We now move to the backward pass of FMHA. The backward pass is significantly different from the forward pass, requiring different SWP and WS strategies. We did not use the Triton tutorial implementation of the backward pass, as it uses a work-inefficient two-pass algorithm. Instead, we implemented the single-pass backward algorithm as described in FA3 [32] in Triton, allowing us to directly compare against cuDNN and the reference FA implementations. The single-pass algorithm consists of five GEMMs and one exponential, followed by an atomic reduction into global memory. Similar to the forward pass, we also implement a sub-tiled version of the algorithm.

6.3.1 Hopper

Hopper results are shown in Figure 10. Similar to forward on Hopper, our port of Triton IR into CUDA achieves a performance boost over Triton and performs competitively with cuDNN on the highest sequence length. Backward attention on Hopper is register constrained, and thus FA3 [32] does not construct a software pipeline that exploits ILP across iteration boundaries. Twill found a similar schedule: it exploited ILP within a loop iteration but could not do so across iterations due to register capacity. Twill's findings confirm that the developers of FA3 did not miss potential performance with their software pipelining strategy. We believe the reference FA3 implementation beats ours by 11% on the 16384 sequence length by using a slightly larger tile size (80x128), allowing for the use of larger GEMMs. Triton only supports power-of-two tile sizes, so we were not able to construct IR with the same tile size, but there is no fundamental reason that Twill cannot handle tile sizes that are not powers of two.

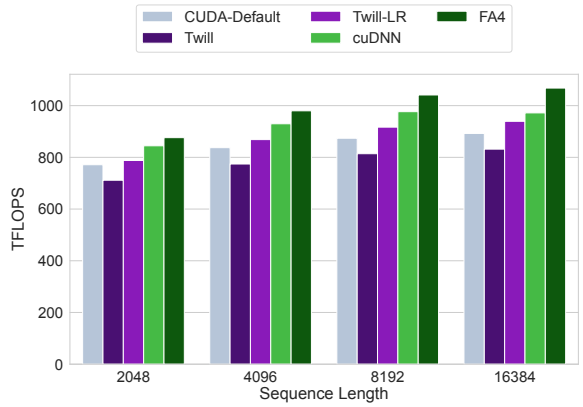


Figure 11: Blackwell FP16 Non-Causal Backward Attention. `(BATCH=4, NUM_HEADS=32, HEAD_DIM=128)`

6.3.2 Blackwell

Blackwell results are shown in Figure 11. Triton was unable to generate code for Blackwell, as it is currently unable to construct Tensor Memory allocation strategies that contain aliasing. The FA4 strategy for Blackwell [38] uses three groups of warps: two groups read accumulators from Tensor Memory to apply exponentials, and the last group stages accumulators in shared memory for the outgoing atomic reduction. Twill finds a different strategy using only two groups of warps which ping-pong between performing exponentials and staging reduction data ("Twill"). Despite our own inspection confirming the schedule should fit within the register-per-thread budget, `ptxas` was unable to register allocate this code without significant register spilling in the main loop, degrading performance. We then ran Twill with a reduced register-per-thread budget and three groups of warps, and it found the essentially same strategy as FA4. `ptxas` succeeded register allocating this version, yielding the small speedup over the default implementation ("Twill-LR"). Additionally, this configuration achieved a smaller initiation interval (Table 1) than the first strategy. This example shows the tightly-knit nature of the optimization space. It is difficult to reason about the usage of registers and warps in a separable manner; the necessary joint reasoning is best performed by automatic solvers.

The overall difference between Blackwell backward implementations is relatively small; the high throughput of the Blackwell TC renders the single-pass algorithm bandwidth-bound, limited by the completion of the atomic reduction at each iteration. However, Twill finds a similar SWP and WS strategy that hides some exposed latency present in a direct port of the Hopper backward strategy. We believe the remaining performance differences to be from orthogonal optimizations around memory layouts and instruction selection in the well-tuned reference implementations.

6.4 Search Time

While optimizing search time is not a primary goal of Twill, we measured it to demonstrate that solutions can be found in reasonable intervals of time that are comparable with existing solutions. Search times for all experiments are shown in Table 1 along with the minimum and final values of I and L. "SWP time" and "SWP-and-WS time" in the table refer to the total time spent in calls to OPTIMAL-MODULO-SCHEDULE and SWP-AND-WS in Algorithm 1 respectively. Twill's search times range from tens of seconds to a few minutes, which is reasonable for an offline system that provides an optimality guarantee for performance-critical kernels that will ultimately be deployed at scale. These times are competitive with existing solutions such as PipeThreader [14], which exhaustively searches a proposed space of SWP schedules. PipeThreader reports discovering the Hopper strategy for the forward pass in 315 seconds while Twill discovers the same strategy in 28 seconds, an order of magnitude faster. Our speedup over PipeThreader demonstrates that, despite their generality, modern ILP and SMT solvers are highly efficient at searching over and pruning large search spaces rapidly. We contrast PipeThreader with Twill in more detail in Section 7.2.

One final interesting observation is the difference in minimum and final values of I discovered by Twill for the backward B100 kernel. The larger final value of I demonstrates the existence of a kernel on a particular machine for which it was impossible to discover a WS strategy for the initiation interval predicted purely by modulo scheduling. This validates our claim from Section 3.3 that SWP and WS should always be handled jointly when determining an optimal schedule, as combinations of kernels and target machines clearly exist that necessitate a holistic approach and it is impossible to predict in advance when they might occur.

7 Related Work

7.1 Tensor Core GPU Programming

The combination of widely varying expertise in GPU programming and the desire for diverse and high-performance machine learning computations has led to a large ecosystem of programming systems for targeting TC GPUs. We provide a survey of this space of systems.

Below deep-learning frameworks like PyTorch [30] or JAX [12], domain-specific languages (DSLs) like Triton [35], Pallas [5] and CUDA Tile [15] provide abstractions for operating on tiles of data. The compilers for each language are then responsible for generating efficient code, often achieving moderate to high performance. Slightly lower-level DSL's like TLX [25] and TileLang [36] also expose tile-based programming models, but provide more control over optimizations. Gluon [28] is a Triton extension that exposes the Triton low-level intermediate representation (IR) to the program-

Benchmark	Total time (s)	SWP time (s)	SWP-and-WS time (s)	Min I, Min L	Final I, Final L
Fwd H100	28	3	25	40, 60	40, 60
Fwd B100	18	1	17	14, 27	14, 27
Bwd H100	84	2	81	30, 38	30, 41
Bwd B100 LR	48	15	32	15, 32	15, 33
Bwd B100	242	41	201	15, 32	20, 33

Table 1: Twill search time and results. Note: values of I, L are incomparable across programs due to cost normalization.

mer. Gluon programmers gain some automation from further lowering of this IR, but retain control over optimizations like memory allocation and WS. Cypress [37] is a task-based programming model that allows programmers to express the decomposition of computation and data across the GPU. Cypress provides a separate mapping interface to control performance-sensitive decisions without affecting correctness. Finally, systems like ThunderKittens [33] and CUTLASS [27] provide the developer complete control, leaving no performance-sensitive decisions to an automated system.

7.2 Software Pipelining

Pipelining for CPUs. Software pipelining is a classic and well-understood optimization for CPUs, critical for extracting enough instruction level parallelism to saturate the machine. Even for modern out-of-order processors, software pipelining can improve instruction throughput by filling the processor's reorder buffer with independent instructions. Compaction-based [7] pipelining algorithms logically unrolled the target loop until a fixed-point pipeline is discovered. Modulo scheduling [24, 31] pipelining algorithms model steady state resource usage with a modulo table data structure. Modulo scheduling algorithms are amenable to the derivation of optimal solutions through the use of Integer Linear Programming [21, 34]. Twill shows how to adapt traditional modulo scheduling techniques for Tensor Core GPUs.

Pipelining for Tensor Core GPUs. ALCOP [23] does SWP of loads with compute, and is therefore insufficient to derive pipelines that overlap compute operations, like FA3.

PipeThreader [14] finds an SWP strategy by defining a space of programs and then searching over it, profiling each candidate to find the fastest. In contrast, Twill takes a constraint-based approach where the space is implicitly defined by the constraints and the search is executed by highly-optimized solvers. Moreover, Twill does not rely on profiling—our approach is static and instead relies on publicly available specifications to estimate operation latencies. A comparison of the search times of Twill and PipeThreader is presented in Section 6.4. Apart from higher search times, because PipeThreader relies on the conventional, heuristic approach to WS involving the creation of producer and consumer warps, it cannot derive the Blackwell warp specialization strategy. Finally, perhaps the most important distinction between Twill and PipeThreader is that PipeThreader cannot provide any optimality guarantees for the schedules it finds.

7.3 Warp Specialization

The GPU programming technique of WS was introduced before NVIDIA GPUs contained TCs. CUDA-DMA [10] leveraged WS to separate data movement between global and shared memory from computation, achieving higher memory bandwidth. Singe [11] partitioned combustion chemistry computations across warps to shrink the register-level working set of each warp, allowing large chemical reactions to be simulated without spilling.

Since Hopper, WS has become ubiquitous in high-performance TC kernels. Programming systems for TC GPUs offer many different layers of control around WS. High-level systems like Triton [35], Cypress [37] or Tawa [13] heuristically define a WS *strategy*, and then automatically transform the source program to implement chosen strategy. Mid-level systems like TLX [25] allow specifying a WS strategy through source-level annotations, and then implement the strategy through transformation algorithms within the compiler. Finally, low-level systems like Gluon [28], ThunderKittens [33] or CUDA C++ require users to both define a WS strategy and realize it in low-level code. Twill focuses specifically on the problem of defining a WS strategy, which all other existing systems perform through machine-specific heuristics or by relying on programmer intuition.

8 Conclusion

We presented Twill, a system that discovers optimal SWP and WS strategies for Tensor Core GPUs. Twill presents a novel joint formulation of SWP and WS that can be offloaded to $\mathbb{Z}$LP and SMT solvers, deriving the complex strategies that experts have found by hand for multiple generations of GPUs. We additionally show that approaches that consider these optimization strategies separately are unable to achieve peak performance.

Acknowledgments

We thank Duane Merrill for discussions about GPU architecture and inspiring our view of warp specialization; Vinod Grover for discussions about SWP and WS; Evghenii Gaburov, Masahiro Masuda and Jason Knight for their support with Triton development and Triton's warp specialization infrastructure; Pradeep Ramani and Cameron Shinn for help with performance debugging of Hopper and Blackwell kernels; Benjamin Driscoll for suggesting the acronym ℤLP; Atharva Chougule, Chris Gyurgyik, Konstantinos Kallas, Katherine Mohr, Sai Gautham Ravipati, AJ Root and Shiv Sundram for providing feedback on a draft of this manuscript.

References

[1] CUDA C++ Best Practices Guide; CUDA C++ Best Practices Guide 13.0 documentation — docs.nvidia.com. https://docs.nvidia.com/cuda/cuda-c-best-practices-guide/index.html. [Accessed 11-26-2025].

[2] Inside NVIDIA Blackwell Ultra: The Chip Powering the AI Factory Era. https://developer.nvidia.com/blog/inside-nvidia-blackwell-ultra-the-chip-powering-the-ai-factory-era/. [Accessed 12-03-2025].

[3] NVIDIA H100 Tensor Core GPU Architecture. https://resources.nvidia.com/en-us-hopper-architecture/nvidia-h100-tensor-c. [Accessed 12-03-2025].

[4] NVIDIA Tesla V100 GPU Architecture. https://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf. [Accessed 12-03-2025].

[5] Pallas: a JAX kernel language; JAX documentation — docs.jax.dev. https://docs.jax.dev/en/latest/pallas/index.html. [Accessed 11-26-2025].

[6] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison-Wesley Longman Publishing Co., Inc., USA, 2006.

[7] A. Aiken and A. Nicolau. Optimal loop parallelization. *SIGPLAN Not.*, 23(7):308–317, June 1988.

[8] Andrew W. Appel. *Modern Compiler Implementation: In ML*. Cambridge University Press, USA, 1st edition, 1998.

[9] Clark Barrett and Cesare Tinelli. *Satisfiability Modulo Theories*, pages 305–343. Springer International Publishing, Cham, 2018.

[10] Michael Bauer, Henry Cook, and Brucek Khailany. CudaDMA: Optimizing GPU Memory Bandwidth via Warp Specialization. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '11, New York, NY, USA, 2011. Association for Computing Machinery.

[11] Michael Bauer, Sean Treichler, and Alex Aiken. Singe: Leveraging Warp Specialization for High Performance on GPUs. In *Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '14, page 119–130, New York, NY, USA, 2014. Association for Computing Machinery.

[12] James Bradbury, Roy Frostig, Peter Hawkins, Matthew James Johnson, Chris Leary, Dougal Maclaurin, George Necula, Adam Paszke, Jake VanderPlas, Skye Wanderman-Milne, and Qiao Zhang. JAX: composable transformations of Python+NumPy programs, 2018.

[13] Hongzheng Chen, Bin Fan, Alexander Collins, Bastian Hagedorn, Evghenii Gaburov, Masahiro Masuda, Matthew Brookhart, Chris Sullivan, Jason Knight, Zhiru Zhang, and Vinod Grover. Tawa: Automatic Warp Specialization for Modern GPUs with Asynchronous References. *arXiv preprint arXiv:2510.14719*, 2025.

[14] Yu Cheng, Lei Wang, Yining Shi, Yuqing Xia, Lingxiao Ma, Jilong Xue, Yang Wang, Zhiwen Mo, Feiyang Chen, Fan Yang, Mao Yang, and Zhi Yang. PipeThreader: Software-Defined Pipelining for Efficient DNN Execution. In *Proceedings of the 19th USENIX Conference on Operating Systems Design and Implementation*, OSDI '25, USA, 2025. USENIX Association.

[15] NVIDIA Corporation. NVIDIA CUDA Tile. https://developer.nvidia.com/cuda/tile, 2025. [Accessed 10-12-2025].

[16] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. An efficient method of computing static single assignment form. In *Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '89, page 25–35, New York, NY, USA, 1989. Association for Computing Machinery.

[17] Tri Dao. FlashAttention-2: Faster Attention with Better Parallelism and Work Partitioning. *arXiv preprint arXiv:2307.08691*, 2023.

[18] Tri Dao, Daniel Y. Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. FlashAttention: Fast and Memory-Efficient Exact Attention with IO-Awareness. In *Proceedings of the 36th International Conference on Neural Information Processing Systems*, NIPS '22, Red Hook, NY, USA, 2022. Curran Associates Inc.

[19] Bruno Dutertre. Yices 2.2. In Armin Biere and Roderick Bloem, editors, *Computer Aided Verification*, pages 737–744, Cham, 2014. Springer International Publishing.

[20] John Forrest and Robin Lougee-Heimer. CBC user guide. In *Emerging Theory, Methods, and Applications*, pages 257–277. INFORMS, September 2005.

[21] R. Govindarajan, E.R. Altman, and G.R. Gao. Minimizing register requirements under resource-constrained rate-optimal software pipelining. In *Proceedings of MICRO-27. The 27th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 85–94, 1994.

[22] Christopher Hojny, Mathieu Besançon, Ksenia Bestuzheva, Sander Borst, Antonia Chmiela, João Dionísio, Leon Eifler, Mohammed Ghannam, Ambros Gleixner, Adrian Göß, Alexander Hoen, Rolf van der Hulst, Dominik Kamp, Thorsten Koch, Kevin Kofler, Jurgen Lentz, Stephen J. Maher, Gioni Mexi, Erik Mühmer, Marc E. Pfetsch, Sebastian Pokutta, Felipe Serrano, Yuji Shinano, Mark Turner, Stefan Vigerske, Matthias Walter, Dieter Weninger, and Liding Xu. The SCIP Optimization Suite 10.0. Technical report, Optimization Online, November 2025.

[23] Guyue Huang, Yang Bai, Liu Liu, Yuke Wang, Bei Yu, Yufei Ding, and Yuan Xie. ALCOP: Automatic Load-Compute Pipelining in Deep Learning Compiler for AI-GPUs. In D. Song, M. Carbin, and T. Chen, editors, *Proceedings of Machine Learning and Systems*, volume 5, pages 680–694. Curan, 2023.

[24] M. Lam. Software pipelining: an effective scheduling technique for VLIW machines. In *Proceedings of the ACM SIGPLAN 1988 Conference on Programming Language Design and Implementation*, PLDI '88, page 318–328, New York, NY, USA, 1988. Association for Computing Machinery.

[25] Meta. TLX language. https://github.com/facebookexperimental/triton/tree/tlx, 2025. [Accessed 30-11-2025].

[26] NVIDIA. CUTLASS Blackwell Forward Attention Main Loop. https://github.com/NVIDIA/cutlass/blob/a2439551c765c5393aebe557ee75d3a0412d2211/examples/77_blackwell_fmha/collective/sm100_fmha_fwd_mainloop_tma_warpspecialized.hpp. [Accessed 20-11-2025].

[27] NVIDIA Corporation. CUTLASS: CUDA Templates for Linear Algebra Subroutines. https://docs.nvidia.com/cutlass/, 2025. Accessed: November 29, 2025.

[28] OpenAI. Gluon language. https://github.com/triton-lang/triton/blob/main/python/examples/gluon/01-attention-forward.py, 2025. [Accessed 30-11-2025].

[29] OpenAI. Triton tutorial attention implementation. https://github.com/triton-lang/triton/blob/fc9e3a637cd883a411a77459e15c2466aa9d3b7c/python/tutorials/06-fused-attention.py, 2025. [Accessed 30-11-2025].

[30] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Köpf, Edward Yang, Zach DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. PyTorch: An Imperative Style, High-Performance Deep Learning Library. *arXiv preprint arXiv:1912.01703*, 2019.

[31] B. Ramakrishna Rau. Iterative modulo scheduling: an algorithm for software pipelining loops. In *Proceedings of the 27th Annual International Symposium on Microarchitecture*, MICRO 27, page 63–74, New York, NY, USA, 1994. Association for Computing Machinery.

[32] Jay Shah, Ganesh Bikshandi, Ying Zhang, Vijay Thakkar, Pradeep Ramani, and Tri Dao. FlashAttention-3: Fast and Accurate Attention with Asynchrony and Low-precision, 2024.

[33] Benjamin F. Spector, Simran Arora, Aaryan Singhal, Daniel Y. Fu, and Christopher Ré. ThunderKittens: Simple, Fast, and Adorable AI Kernels. *arXiv preprint arXiv:2410.20399*, 2024.

[34] Artour Stoutchinin. An integer linear programming model of software pipelining for the MIPS R8000 processor. In Victor Malyshkin, editor, *Parallel Computing Technologies*, pages 121–135, Berlin, Heidelberg, 1997. Springer Berlin Heidelberg.

[35] Philippe Tillet, H. T. Kung, and David Cox. Triton: an intermediate language and compiler for tiled neural network computations. In *Proceedings of the 3rd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*, MAPL 2019, page 10–19, New York, NY, USA, 2019. Association for Computing Machinery.

[36] Lei Wang, Yu Cheng, Yining Shi, Zhengju Tang, Zhiwen Mo, Wenhao Xie, Lingxiao Ma, Yuqing Xia, Jilong Xue, Fan Yang, and Zhi Yang. TileLang: A Composable Tiled Programming Model for AI Systems, 2025.

[37] Rohan Yadav, Michael Garland, Alex Aiken, and Michael Bauer. Task-Based Tensor Computations on Modern Gpus. *Proc. ACM Program. Lang.*, 9(PLDI), June 2025.

[38] Ted Zadouri, Markus Hoehnerbach, Jay Shah, Timmy Liu, Vijay Thakkar, and Tri Dao. Flashattention-4: Algorithm and kernel pipelining co-design for asymmetric hardware scaling, 2026.

TileLoom: Automatic Dataflow Planning for Tile-Based Languages on Spatial Dataflow Accelerators

Wei Li[1,†], Zhenyu Bai[1,†,✉], Heru Wang[1,†], Pranav Dangi[1,†],
Zhiqiang Zhang[1], Cheng Tan[2], Huiying Lan[3], Weng-Fai Wong[1], Tulika Mitra[1]

[1]School of Computing, National University of Singapore
[2]Arizona State University and Google, [3]Lumai Ltd.

{liwei01, zhenyu.bai, heru.wang, dangi}@nus.edu.sg
t0937444@u.nus.edu, chengtan@asu.edu, huiying.lan93@gmail.com dcswwf,dcstm@nus.edu.sg,

Abstract

Spatial dataflow accelerators are a promising direction for next-generation computer systems because they can reduce the memory bottlenecks of traditional von Neumann machines such as CPUs and GPUs. They organize computation around explicit, compiler-managed data movement over on-chip networks, allowing operands to be forwarded directly between processing elements and reducing reliance on high-latency, bandwidth-limited global shared memory. However, their performance depends strongly on how workloads are mapped to hardware. Naive mappings can perform poorly, and most users rely on hand-tuned vendor libraries. Thus, despite their potential for high performance, energy efficiency, and cost efficiency, limited programmability remains a major barrier to wider adoption.

This paper presents *TileLoom*, an MLIR-based end-to-end framework that compiles tile-based programs, such as Triton kernels, onto spatial dataflow architectures. Unlike compiler frameworks that focus on optimizing code generation *within* a single tile, *TileLoom* distributes tile instances across spatially distributed cores and exploits the on-chip network and distributed memories to increase data reuse and reduce communication. *TileLoom* introduces a hardware representation that captures interconnect topology, memory hierarchy, and compute capabilities, enabling both architecture-specific optimizations and support for diverse spatial dataflow targets. In experiments on two generations of Tenstorrent systems, *TileLoom* achieves performance comparable to vendor libraries on various kernels.

[†]These authors contributed equally.
[✉]Zhenyu Bai is the corresponding author (zhenyu.bai@nus.edu.sg).

1 Introduction

Modern high-performance workloads, especially deep learning workloads, are highly data-intensive [5, 15]. For many of these workloads, the main bottleneck is not compute, but memory bandwidth [12, 27, 43, 70]. As process technology scales, we can place more arithmetic units on a chip, but off-chip memory bandwidth and capacity do not scale at the same rate [7, 46]. Each DRAM access costs much more energy than an arithmetic operation [22]. On-chip SRAM also becomes relatively more expensive in area and power as we move to smaller technology nodes [18, 22]. Together, these trends make it hard for conventional, memory-centric von Neumann architectures such as CPUs and GPUs to keep their compute units busy.

Spatial dataflow architectures are emerging as a strong alternative. Systems such as Tenstorrent [3], Cerebras [8], Graphcore [26], SambaNova [55, 58], Groq [1, 2], Meta's MTIA [14], AWS's Trainium [6], and Tesla's Dojo [19] organize computation around explicit, often software-controlled data movements over on-chip networks and buffers, reducing reliance on large shared caches and high-latency off-chip memories. When data is passed from core to core over short on-chip wires, the energy per bit and latency can be much lower than going back and forth to a shared cache or off-chip memory [4, 21, 22]. Figure 1 shows a representative spatial dataflow accelerator architecture from Tenstorrent: a 2D grid of cores, each typically a SIMD or vector engine, often with a matrix unit, plus a local scratchpad, connected by a packet-switched mesh NoC. 64 cores can concurrently access their local scratchpads, yielding an aggregate peak bandwidth of roughly 24.5 TB/s—substantially higher than the 6 TB/s L2 bandwidth of the NVIDIA H100 [60]. This abundant per-core bandwidth significantly alleviates memory bottlenecks for bandwidth-bound operators.

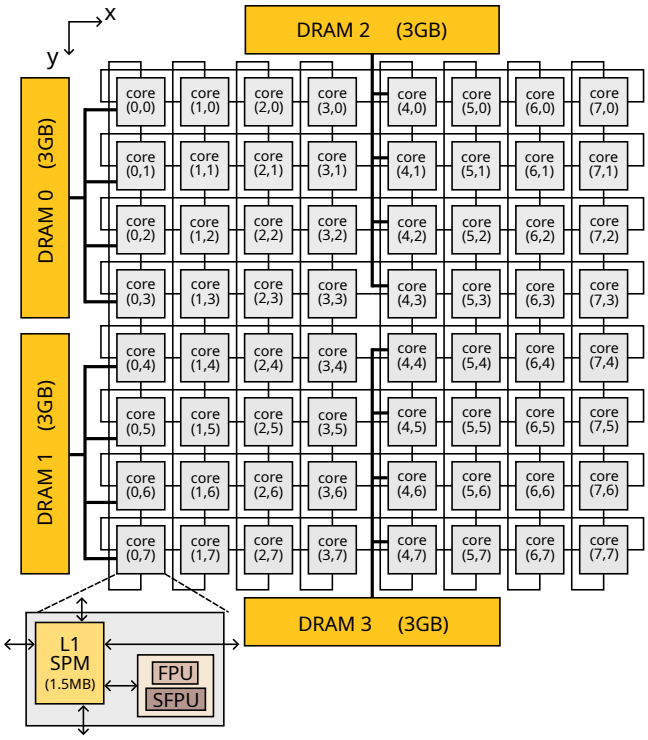


Figure 1: An example 2D-mesh spatial dataflow architecture, modeling Tenstorrent-Wormhole system.

While the structure of spatial dataflow accelerators can provide high efficiency, it also creates a serious programmability problem [16,48,56]. Performance depends heavily on how the workload is mapped: which cores execute which parts of the computation, how data is partitioned across local memories, and how traffic is scheduled on the network [17,38,56,71,73]. A naive mapping can cause severe load imbalance, network congestion, underutilized cores, or excessive off-chip memory traffic, resulting in poor performance and energy efficiency [17,38,71,73]. As a result, compiling high-level programs onto dataflow architectures is challenging, and non-experts struggle to write efficient low-level programs for them. Instead, users rely on vendor-provided, hand-optimized libraries by experts that implement a small set of popular kernels, such as matrix multiplication, convolutions, and attention, with carefully engineered mappings [11,28,34,36,45]. This dependence limits both portability across architectures and the ability to experiment with new kernels or model structures [11,34,45].

A common abstraction for parallel accelerators is the grid–block–thread model popularized by CUDA's grid-of-thread-blocks interface [30,49]. The programmer decomposes the problem into blocks and launches a grid of block instances across the device; each block, such as a CUDA thread block or OpenCL work-group, performs the same computation on a different region of the input or output tensor, and the grid covers the full problem domain [29,49]. Similar grid-based tiling models appear in high-level systems such as Halide and TVM [10,57], which treat tile shapes and launch configurations as schedule parameters and search over them to improve locality and parallelism.

On conventional, memory-centric architectures such as GPUs, the grid level is managed by hardware [30,47,49]: a hardware scheduler dynamically assigns blocks to Streaming Multiprocessors (SMs), and a shared cache hierarchy implicitly captures most data reuse across blocks [25,47]. This leaves little room for compilers to control grid-level placement, execution order, or communication. By contrast, the block–thread level is software-defined: the compiler or programmer decides how work within a block is mapped onto threads, warps, and intra-SM resources. CUDA exposes fine-grain control over thread organization, shared-memory layout, and synchronization, but exploiting it well requires careful tuning and detailed knowledge of GPU microarchitecture. To make this block-level programming more accessible, languages such as Triton [61], TileLang [66], CuTile [50], Tilus [13], and Taichi [23] let users express kernels in terms of high-level tile operators that define the work of a single block, while their compilers lower these operators onto intra-core or intra-SM resources.

Spatial dataflow accelerators change this picture. These architectures distribute compute and scratchpads across large arrays of processing elements connected by a high-bandwidth on-chip network, often without a large, unified cache that automatically exploits cross-core reuse [33,54,63]. Instead of offloading communication and placement decisions to hardware caches and schedulers, they expose richer inter-core programmability to software. Achieving good performance therefore requires a compiler or programmer to decide not only how to implement the tile program on a single core, but also how to place tile instances across cores and schedule them in time so that data can be forwarded or multicast efficiently over the NoC [33,51,63] and memory system. This inter-core programmability creates an enormous mapping design space: valid mappings are combinatorially large, and different mappings expose different reuse patterns and communication costs [51,54]. In today's systems, these decisions are baked into vendor-specific compilers and libraries [9, 54, 58, 63, 64, 75], which encode architecture-specific strategies for placement, routing, and pipelining, but are time-consuming to develop and do not generalize easily to new kernels, models, or hardware generations.

By giving more control to software, spatial dataflow architectures are harder to program but can simplify hardware and improve efficiency. Crucially, **as the hardware becomes more transparent to software by exposing explicit cores, networks, and memories instead of opaque caches and schedulers, dataflow architectures become more predictable than memory-centric architectures.** With sufficient hardware information, **a compiler can more easily**

reason about the performance of different schedules and deduce good static mappings. Building on this insight, we propose ***TileLoom*, a compiler framework that supports end-to-end execution of tile-based programs on spatial dataflow systems.** Our goal is to let users write kernels in a tile-wise language, such as Triton, while the compiler maps the logical grid onto the physical array of cores, schedules tile execution in time, and orchestrates data movement and reuse over the NoC and distributed memories. In other words, *TileLoom* takes on responsibilities typically handled by hardware schedulers and runtimes on GPUs, moving these decisions to compile time for spatial dataflow systems. To support a broad range of dataflow architectures, *TileLoom* introduces a hardware description that models interconnect topology, memory hierarchy, and compute resources. Given this description, the compiler searches for mappings that balance load, improve data reuse, and respect network and memory constraints across architectures. We evaluate our approach on Tenstorrent systems and show that it can match or outperform vendor-provided handwritten libraries on key kernels.

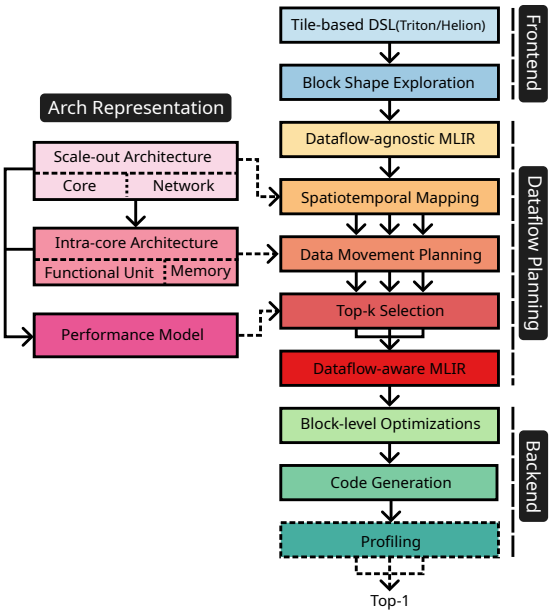


Figure 2: *TileLoom* framework overview.

2 Framework

2.1 Overview

TileLoom compiles a kernel written in a tile-based DSL (currently supporting Triton and Helion) into an executable for a target spatial dataflow architecture.[1] As shown in Figure 2, the compiler stack is organized around three main components: a front-end that lower tile-level kernels into a standard dataflow-agnostic MLIR representation that we propose; a dataflow planning stage that decides spatiotemporal mappings, data movements and generate candidates in a standard dataflow-aware MLIR representation; a back-end that generates hardware-specific executables for each core, all guided by the multi-level architecture representation and performance model.

The **front-end** takes as input a tile-level kernel and a description of how that kernel is scaled out over the full problem (the launch grid). It explores candidate *block shapes*: tile sizes and layouts, similar to the conventional auto-tunning process, and constructs corresponding programs. These candidate programs are then lowered into an MLIR-based intermediate representation and passed through normalization passes so that they share a common structure suitable for the dataflow planning pipeline. At this point, the computation is represented in a uniform, dataflow-agnostic IR affine + linalg + scf + arith): it encodes the tile and grid structure, but does not yet commit to any particular mapping onto the hardware.

The **dataflow planning** stage determines how this logical grid of tiles is realized on the target architecture. Guided by the hardware description, *TileLoom* explores spatiotemporal mappings of tile instances onto cores and time. For each candidate mapping, it performs data reuse analysis to identify which tiles can share data over the NoC or be reused over time via buffering, and it derives concrete data-movement plans: where each tensor tile is allocated, when it is copied, and how it is broadcasted or shared over the buffers. Together, the spatiotemporal mapping and the data-movement plan generate a design space of potential optimal dataflow planning candidates.

To model arbitrary dataflow architectures, the **architecture representation** is proposed, which provides the key inputs to both the dataflow planning search and the performance model. The scale-out (inter-core) description captures the spatial structure of the core array and interconnect, and guides spatiotemporal mapping decisions. The intra-core description captures the local memory hierarchy and compute resources, and guides decisions about where data is stored and how it is staged. *TileLoom* combines both levels of abstraction to generate **a performance model** that estimates the cost of different data-movement plans, taking into account memory bandwidths, NoC capabilities, and per-core compute throughput. The performance model is used to select the top candidates from the dataflow design space.

After dataflow planning stage, the IR is *dataflow-aware*: memory allocations, copies, and communication endpoints are concretized according to the chosen mapping. After chosing the top candidates from the design space, the **block-level optimization and code generation** stage compiles the dataflow-aware program that runs on each core into the vendor's existing back-end to generate an executable for each core. This corresponds to the existing compilers that compile tile-based

[1]The source code is available at https://github.com/ecolab-nus/loom-dataflow.

DSLs to block-level program.

TileLoom uses an optional two-step selection strategy to provide better support for various architectures: the automatically generated performance model (using the architectural information) first ranks candidate dataflow plans and selects the top-k mappings statically, and these k candidates are then profiled on the real hardware to choose the final mapping (which bridge the small details not covered by the architecture representation). This combination of model-guided search and hardware validation enables *TileLoom* to produce high-quality mappings while remaining portable across different spatial dataflow systems, especially when micro-architectural details are missing and hence causing inaccuracy in the hardware modeling.

2.2 Spatiotemporal Mapping

Input representation. A unified dataflow-agnostic MLIR representation is required before the dataflow planning pipeline. We assume this representation produced by the front-end, we will cover the implementation details of the Triton front-end in Section 3.1. An example of this MLIR for a matrix–multiplication kernel is shown in Listing 1. The block size and the strides for the input and output matrices are fixed by the front-end. The 2D output space is partitioned over two grid dimensions, `x` and `y`, which scale over the m and n dimensions of an $m \times k \times n$ matrix multiplication (an output-stationary tiling).

Scaling out across tiles is represented by an `affine.parallel` loop over `%block_id_x` and `%block_id_y`. Inside this loop, an `scf.for` loop iterates over the k dimension and accumulates into the same output tile, representing the sequential execution within one block over one output tile. The front-end is required to "affinize" the address arithmetic for memory operations, so every load and store address is an affine function of the tile indices and intra-tile indices. The tile-wise computation itself is expressed with `linalg` operations; this portion of the program is left unchanged during dataflow planning and is later lowered by the back-end.

The purpose of spatial–temporal mapping is to decide how the iteration space of the `affine.parallel` loop—the logical multidimensional tile grid—is assigned to physical cores and to time. To preserve locality, *TileLoom* uses *tiling-based* mappings: contiguous regions of the iteration space are mapped to contiguous spatial regions of the core array or to contiguous temporal regions in the execution schedule.

Schedule representation. On a 2D-mesh architecture like the one in Figure 1, spatial–temporal mapping produces the loop structure shown in Listing 2. The outermost `affine.parallel` loop now iterates over hardware spatial dimensions x and y, each of size 8. These indices correspond directly to the cores in an 8×8 2D mesh. After mapping, this loop represents code that actually runs in parallel across cores; its semantics have changed from a logical grid of parallelizable work-items to the physical parallel core indices.

The next `affine.for` loops iterate over `%tx` and `%ty`. These loops enumerate *waves* of tiles assigned to the same hardware array, i.e., temporal dimensions across blocks. Each wave assigns a batch of logical tiles to the available spatial cores, and the order of these waves determines the temporal schedule with which tiles traverse the array. The innermost `scf.for` remains a purely sequential loop within each core's program.[2]

Design space. The mapping from the original parallel dimensions to spatial and temporal dimensions defines the dataflow of the kernel. Under our tiling-based scheme, the design space is characterized by three coupled choices.

First, each original parallel dimension can be mapped to zero or more spatial dimensions. Mapping a parallel dimension to a spatial dimension corresponds to tiling the loop by the size of that spatial dimension and introducing a new outer `affine.parallel` loop over the hardware index.

Second, when a parallel dimension is tiled by multiple spatial dimensions (for example, by both x and y), the order in which tiling is applied matters. Different tiling orders induce different spatial layouts of tiles on the mesh and different execution schedules, and therefore expose different opportunities for spatial reuse and different communication costs.

Third, once all available spatial dimensions have been used, the remaining parallel dimensions become temporal dimensions implemented as loops over waves, (i.e. changing the rest of `affine.parallel` to `affine.for`), for instance the `%tx`↩ and `%ty` loops in Listing 2. These temporal loops and their order determine how tiles are batched onto the array over time and in what order they revisit each region of the global iteration space, which in turn affects temporal reuse and the shape of NoC traffic.

TileLoom enumerates candidate spatial–temporal mappings by exploring combinations of these choices. Each mapping fixes a concrete loop nest structure, which the subsequent analyses use to reason about data placement, reuse, and communication.

2.3 Data Reuse and Memory Operations

Different spatial–temporal mappings expose different opportunities to reuse data across time and across cores. *TileLoom* first analyzes these opportunities and then decides how to allocate data to memories and when and where to issue copies as broadcasts over the NoC or as loading from global memories.

Reuse analysis on affine accesses. For a fixed spatial–temporal mapping (Section 2.2), the loop nest contains: spatial loops (`affine.parallel`) over hardware core indices; temporal loops (`affine.for`) over waves of tiles, and sequential loops (`scf.for`) inside each core, as shown in Listing 2.

[2]This sequential loop is also temporal, but we use "sequential" to distinguish intra-core sequencing from inter-tile temporal dimensions.

Listing 1: MLIR representation before dataflow planning.

```
func.func @matmul(%A: memref<*xf32>, %B: memref<*xf32>, %C: memref<*xf32>, %grid_dim_x: index, %grid_dim_y: index, %grid_dim_z: index) {
    affine.parallel (%block_id_x, %block_id_y) = (0, 0) to (%grid_dim_x, %grid_dim_y) {
        %cst = arith.constant 0.000000e+00 : f32
        %0 = tensor.empty() : tensor<64x64xf32>
        %1 = linalg.fill ins(%cst : f32) outs(%0 : tensor<64x64xf32>) -> tensor<64x64xf32>
        %2 = scf.for %arg8 = 0 to 8 step 1 iter_args(%arg9 = %1) -> (tensor<64x64xf32>) {
            %4 = affine.apply affine_map<(d0, d1) -> (d1 * 32768 + d0 * 64)>(%arg8, %block_id_x)
            %reinterpret_cast_0 = memref.reinterpret_cast %arg0 to offset: [%4], sizes: [64, 64], strides: [512, 1] : memref<*xf32> to memref<64x64xf32, strided<[512,↩
        1], offset: ?>>
            %alloc = memref.alloc() : memref<64x64xf32>
            memref.copy %reinterpret_cast_0, %alloc : memref<64x64xf32, strided<[512, 1], offset: ?>> to memref<64x64xf32>
            %5 = bufferization.to_tensor %alloc restrict writable : memref<64x64xf32> to tensor<64x64xf32>
            %6 = affine.apply affine_map<(d0, d1) -> (d1 * 32768 + d0 * 64)>(%block_id_y, %arg8)
            %reinterpret_cast_1 = memref.reinterpret_cast %arg1 to offset: [%6], sizes: [64, 64], strides: [512, 1] : memref<*xf32> to memref<64x64xf32, strided<[512,↩
        1], offset: ?>>
            %alloc_2 = memref.alloc() : memref<64x64xf32>
            memref.copy %reinterpret_cast_1, %alloc_2 : memref<64x64xf32, strided<[512, 1], offset: ?>> to memref<64x64xf32>
            %7 = bufferization.to_tensor %alloc_2 restrict writable : memref<64x64xf32> to tensor<64x64xf32>
            // Tile-wise computation, omitted
            linalg.xxx;
            linalg.yyy;
            scf.yield %9 : tensor<64x64xf32>
        }
    }
}
```

Listing 2: Loop structure after spatial–temporal mapping.

```
affine.parallel (%x, %y) = (0, 0) to (8, 8) {
  affine.for %tx = 0 to %grid_dim_x ceildiv 8 {
    affine.for %ty = 0 to %grid_dim_y ceildiv 8 {
      scf.for %k {
        // tile-wise computation
      }
    }
  }
}
```

The front-end expresses all memory accesses as affine functions of these loop indices. For each access, *TileLoom* inspects which induction variables appear in its affine expression. If an access does not depend on a spatial index such as `%x`, then the accessed tile is identical for all cores along that dimension and is spatially reusable there. If an access does not depend on a temporal loop variable such as `%tx`, then the same tile is used across all iterations of that temporal loop and is temporally reusable there. If the access depends only on sequential indices, then reuse is purely intra-core. *TileLoom* records this information as reuse annotations on the memory operations.

Spatial reuse and broadcasts. We begin from a conservative baseline in which every core loads its tiles directly from global memory (for example, an L2 cache or DRAM) in the innermost loop, with no explicit sharing across cores.

If a load has no spatial reuse then the tile is unique to each core, so the load must remain a per-core global memory operation. If a load is spatially reusable along one or more spatial dimensions, *TileLoom* can reduce global traffic by replacing many per-core loads with a smaller number of global loads, followed by broadcasts over the NoC.

In the simplest case, if a tile is reusable only along a single spatial dimension, a designated producer core (or a small group of producers) loads it once from global memory and forwards it along that dimension (for example, along each row of the mesh), while receiving cores buffer their local copies. When a tile is reusable along multiple spatial dimensions, there are several concrete ways to realize that reuse. One option is to first duplicate the tile across all rows (or columns) and then perform independent one-dimensional broadcasts along the parallel dimensions; another is to propagate the tile in a wavefront-style pattern that sweeps across the array. These choices expose different tradeoffs between NoC traffic, latency, and local buffer usage. Listin 3 shows one example candidate of the matrix multiplication mapped to the 2D-mesh example architecture (Figure 1 with a 2D dataflow where the A tiles are broadcasted for each row of cores through the horizontal links of the NoC and the B tiles for each column of cores through the vertical links (we will present the notion of network resources later in Section 2.4). The broadcast information is associated to the load instructions as annotations.

TileLoom does not fix a single strategy: it uses the network description from the hardware representation to enumerate the broadcast patterns that are legal, for each spatially reusable load, a small set of candidate implementations ranging from direct per-core global loads to one-dimensional and multi-dimensional broadcasts. *TileLoom* enumerate all the possibile combinations of all memory operations that creates a design space. Their different hardware costs will be taken into account later by the performance model to select the best ones.

Listing 3: Spatial Reuse

```
affine.parallel (x,y) = (0,0) to (8,8) {
  affine.for tm = 0 to M_waves {
    affine.for tn = 0 to N_waves {
      scf.for tk = 0 to K_tiles {
        load A[tm*8+x, tk]  {type="broadcast", resource={%noc_h}}
        load B[tk, tn*8+y]  {type="broadcast", resource={%noc_v}}
        // ...
      }
    }
  }
}
```

Temporal reuse and loop hoisting. Temporal reuse is realized by choosing the loop level at which a load (or broadcast) is issued. At this point, the loop order is fixed by the spatial–temporal mapping; but we can hoist loads outward so that the same tile is reused across more iterations, at the cost of retaining it longer in a local buffer.

Consider the simplified GEMM-like loop nest below in Listing 4 left, treated as one candidate loop order, `tm` → `tn` → `tk`: the access `A[tm, tk]` depends on `tm` and `tk`, but not on `tn`, so tiles of `A` are temporally reusable across the `tn`↩ loop. If we hoist the loads of `A` outside the `tk` loop, we must buffer all tiles `A[tm, *]` for the current `tm`. Because the address depends on `tk`, hoisting across `tk` enlarges the buffered region from a single tile `A[tm, tk]` to the entire strip `A[tm, 0..K_tiles-1]`. Hoisting further outward, above `tn`, keeps the same buffered strip but reuses it across all values of `tn`, as shown in Listing 4 right:

```
affine.for tm = 0 to M_tiles {
  affine.for tn = 0 to N_tiles {
    scf.for tk = 0 to K_tiles {
      load A[tm, tk]
      load B[tk, tn]
      // ...
    }
  }
}
```

```
affine.for tm = 0 to M_tiles {
  load A[tm, *]
  affine.for tn = 0 to N_tiles {
    scf.for tk = 0 to K_tiles {
      load B[tk, tn]
      // ...
    }
  }
}
```

Listing 4: Loop structures before and after hoisting.

In general, hoisting a load across a loop that the access does not depend on increases reuse without increasing the size of the buffered region, because the accessed tile is the same for all iterations of that loop. Hoisting across a loop that the access does depend on expands the buffered region in proportion to the extent of that loop, because more distinct tiles must be kept live simultaneously. *TileLoom* applies these rules to enumerate, for each load or broadcast, all legal hoisting levels. For each level it computes the required buffer footprint and discards options whose footprint exceeds the capacity of the hardware model.

Temporal vs. Spatial. Temporal reuse and spatial reuse are orthogonal. A tile can be reused only temporally (loaded once per core and reused across iterations), only spatially (broadcast once and immediately consumed), or in both ways (broadcast once and then reused across several temporal iterations). In all cases, the decision can be viewed as picking when a tile is first loaded or received and how long it remains live in local storage under a fixed loop order.

Combining these choices for all loads yields a concrete allocation and copy mapping: a description of which memory each tile resides in at each point in time and which NoC transfers occur. This schedule can be represented by a loop sctructure with annotations as shown a simplified example in Listing 5. Each memory load is annotated with the target buffer, the type of load (using NoC for broadcasting or simple global load), and the NoC resources required. *TileLoom* prunes mappings that violate memory-capacity constraints and passes the remaining candidates to the performance model, which evaluates their compute, memory, and network costs and selects the top-k mappings for back-end profiling.

Listing 5: Example dataflow-friendly MLIR snippet before selection for matrix multiplication kernel on the example 2D-mesh architecture.

```
affine.for tm = 0 to M_tiles {
  alloc A {target_buffer=%L1, size=K_tiles*block_M*block_K}
  load A[tm, *] {type="global", resources={%noc_h, %noc_v}}
  affine.for tn = 0 to N_tiles {
    alloc C {target_buffer=%L1, size=block_M*block_N}
    scf.for tk = 0 to K_tiles {
      alloc B {target_buffer=%L1, size=block_K*block_N}
      load B[tk, tn] {type="broadcast", resources={%noc_h, %noc_v}}
      // tile-wise computations
      load C
      linalg.matmul ...
      linalg.exp ...
      linalg.sqrt ...
    }
    store C {type="global". resources={%noc_h, %noc_v}}
  }
}
```

2.4 Hardware Representation

TileLoom is designed to target multiple dataflow architectures. To make mapping decisions, the compiler needs a structured description of the hardware: how cores are arranged in space, where memories are placed, how components are connected, and what compute resources are available at each location. *TileLoom* captures this information in a multi-layer hardware representation stack. Different layers of this stack are consumed by different stages of the compiler passes.

We encode this representation in a custom MLIR dialect, `df`. The dialect provides operators that describe the scale-out structure of the machine (cores and interconnects), the memory hierarchy and its connectivity, and the intra-core compute units. The performance model and the mapping passes process the program and the hardware description written this dialect together, rather than hard-coding any particular architecture.

Scale-out architecture. At the top level, the `df` dialect describes the spatial layout of cores and the on-chip interconnect. The following operators are needed:

`df.spatial_dim(size)` declares an abstract spatial dimension, used to index and replicate hardware components. Spatial dimensions naturally represent arrays of parallel resources such as cores or memories.

`df.core(scaleout, scalein)` declares a set of cores indexed by the dimensions listed in `scaleout`; the `scalein` of the operation contains the compute components that live inside each core (optional argument, used at a lower abstraction level, described later).

`df.interconnects(components, map, bandwidth)` declares a network (set of links) that connects a set of components according to an affine map `map` and `bandwidth` specifies the bandwidth per link.

These operators are used to describe the scale-out architecture used in the spatial–temporal mapping. For our example 2D-mesh architecture (Figure 1), we can describe it with:

This fragment describes an 8×8 array of cores connected by horizontal and vertical rings (or a torus). The modulo in the affine maps encodes wrap-around links. The spatial–temporal

Listing 6: 2D mesh with abstract scale-out.

```
%x = df.spatial_dim 8
%y = df.spatial_dim 8
%cores = df.core { scaleout = (%x, %y) }
%noc_h = df.interconnects %cores, %cores { map = affine_map<(d0, d1) -> ((d0 + 1) mod 8, d1)>,bandwidth = 28 }
%noc_v = df.interconnects %cores, %cores { map = affine_map<(d0, d1) -> (d0, (d1 + 1) mod 8)>, bandwidth = 28 }
```

mapping pass uses this scale-out description to map the logical tile grid onto physical cores (Section 2.2), and the performance model uses the `df.interconnects` operators to estimate communication cost and traffic congestion of the memory operations (Section 2.5).

Memories and data movement. When planning data movements (Section 2.3), *TileLoom* must reason about concrete buffers and how the network feeds data into them. To do so, the scale-out description is refined with explicit memories.

`df.memory(scaleout, size, bandwidth)` declares a set of memories indexed by the `scaleout` dimensions. Each instance has the given capacity and per-port bandwidth.

`df.mux(dst, bandwidth, srcs, map)` declares a 1-to-N connectivity between `dst` components and `srcs`, with topology specified by an affine `map`. This operator captures fan-out connections such as "each core can access its local scratchpad" or "groups of cores share a DRAM channel".

A lowered version of the 2D-mesh description that includes L1 memories and DRAM is shown in Listing 7:

Listing 7: 2D mesh with scale-out cores, scratchpads, and DRAM.

```

%x = df.spatial_dim 8
%y = df.spatial_dim 8
%cores = df.core {scaleout=(%x, %y)}
// Per-core scratchpad memories.
%L1 = df.memory {scaleout=(%x, %y), size = 1499136, bandwidth = 60}
// Connect each core (x, y) to its local L1(x, y).
%core_to_L1 = df.mux %cores, %L1, {map = affine_map<(d0, d1) -> (d0, d1)>}
// On-chip NoC now connects L1 memories.
%noc_h = df.interconnects %L1, %L1, {map = affine_map<(d0, d1) -> ((d0 + 1) mod 8, d1)>, bandwidth = 28}
%noc_v = df.interconnects %L1, %L1, {map = affine_map<(d0, d1) -> (d0, (d1 + 1) mod 8)>, bandwidth = 28}
// Off-chip DRAM channels, indexed by a 1D spatial dimension.
%dram_idx = df.spatial_dim 4
%drams = df.memory {scaleout = %dram_idx, size = 12884901888, bandwidth = 267}
// Map each group of 4x4 edge cores to a DRAM channel.
%to_dram = df.interconnects %L1, %drams {map = affine_map<(d0, d1) -> (d0 ceildiv 4 + 2 * (d1 ceildiv 4))>, bandwidth = 30}
```

This representation now distinguishes the physical buffers that can hold tiles. For example, it specifies that each core `(x, y)` has an L1 scratchpad of around 1.5 MB, and that inter-core traffic flows between L1s rather than directly between cores. It also encodes DRAM channels are connected: every group of four adjacent cores along each edge shares the same DRAM bank, as in Figure 1. When *TileLoom* chooses where to buffer a tile and the costs of memory operations, it uses these `df.memory`, `df.mux`, and `df.interconnects` operators. As shown earlier in Listing 5, load instructions are annotated with bindings to these physical resources.

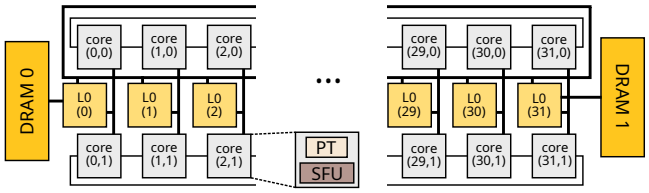


Figure 3: Example 1D triple-ring architecture modeled with the `df` dialect.

Intra-core compute model. To drive the performance model down to the level of individual cores, *TileLoom* needs a coarse description of the microarchitecture within each core. The `df` dialect provides operators for this purpose:

`df.mat(shape, throughput)` declares a matrix unit (for example, a tensor core) with a given input shape and sustained throughput.

`df.vec(shape, throughput)` declares a vector unit with the given vector width and throughput.

`df.scalar(latency)` declares a scalar unit with a given latency for scalar operations.

Each unit is assumed to accept operands of the specified shape and to produce results at the given throughput. These units are then attached to cores via the `scalein` argument of `df.core`, which describes the internal composition of each core. The lowest-level version of the 2D-mesh description therefore extends the previous listing with intra-core units:

Listing 8: Extra specifications of the intra-core architecture

```
%FPU = df.mat {shape=[32, 32, 32], throughput=98}
%SFPU = df.vec  {shape=[32], throughput=3}
%cores = df.core {scaleout = (%x, %y), scalein=(%FPU, %SFPU, [8, 1])}
```

With this information, the performance model can reason about the timing of both compute and memory. It can, for example, estimate how many cycles a particular tile-level matmul consumes on the matrix units and whether the NoC bandwidth and the L1 buffer are sufficient to keep them fed.

Expressiveness beyond 2D meshes. Although we have used the Tenstorrent-2D-mesh architecture as example, `df` can describe other spatial dataflow architectures. For instance, a 1D triple-ring topology similar to the IBM-Spyre accelerator as shown in Figure 3 can be described using `df` program as shown in Listing 9 .

Discussion We structure the hardware representation as a stack of layers of abstractions because each compiler pass should depend only on the level of detail it actually needs. Spatial–temporal mapping requires only the scale-out structure of cores and the topology and bandwidth of the interconnect. Data-movement planning additionally needs to know where memories are placed and how they are wired to compute and to DRAM. The fine-grain performance model, in

Listing 9: Example of the df dialect describing a 1D triple-ring architecture.

```
module {
    // functional units
    %PT = df.mat {shape = [128, 128, 128], throughput=16384}
    %SFP = df.vec {shape = [128], throughput=128}
    // scale-out
    %x = df.spatial_dim 32
    %y = df.spatial_dim 2
    %cores = df.core "cores" {scaleout=(%x, %y) , scalein=(%PT, %SFP, [1,1])}
    %L1 = df.memory {scaleout=(%x) , size = 2097152, bandwidth = 128}
    %core_to_mem = df.mux %cores, %L1, {map = affine_map<(d0, d1) -> (d0)>}
    %small_rings = df.interconnects %cores, %cores, {map = affine_map<(d0, d1)
        -> ((d0 + 1) mod 8, d1)>, bandwidth = 32}
    %big_ring = df.interconnects %L1, %L1, {map = affine_map<(d0) -> ((d0 + 1)
        mod 8)>, bandwidth = 258}
    // Global (DRAM/L2)
    %d = df.spatial_dim 2
    %dram = df.memory {scaleout=(%d) , size = 34359738368, bandwidth = 512}
    %to_dram = df.interconnects %dram, %L1, {map = affine_map<(d0)->(d0*31)>}
}
```

turn, needs a high-level view of intra-core compute units and their throughputs. This separation improves the reusability of the compiler. Changing the on-chip network, the memory hierarchy, or the per-core microarchitecture amounts to modifying the `df` description, without rewriting the optimization passes and the rest of hardware description. It also creates a bridge from software-level mapping decisions to hardware-level design trade-offs. Starting from the lowest abstraction level, the same representation can be refined further to include implementation-specific costs such as area and power, enabling combined design space exploration over both mappings and hardware configurations—an important capability for spatial dataflow architectures, where the architecture itself varies widely across generations and vendors.

2.5 Performance Modeling

After spatial–temporal mapping and data reuse / allocation decisions, *TileLoom* has a set of candidate dataflow schedules. Each candidate is represented as an MLIR program in which loop nests, memory operations, and data movements (global loads, broadcasts, buffered loads) are fully specified, and every memory or network operation is bound to concrete hardware resources described in the `df` dialect (Section 2.4). A simplified example is shown in Listing 5. The role of the performance model is to estimate the execution time of each candidate, using the compute units, memories, and interconnects defined in the `df` description, and then select the top-k candidates for downstream code generation and profiling.

Figure 4 illustrates how the performance model evaluates the overall execution time of the example in Listing 5. It evaluates from the innermost loop outward, aggregating compute, memory, and network costs hierarchically.

Compute cost per loop body. We first estimate the execution time of the innermost loop body, treating it as a block-level program running on a single core. For a given tile shape, every high-level operator (for example, a `linalg.matmul`) is decomposed into the core's low-level compute intrinsics. The available matrix, vector, and scalar units and their throughputs come from the `df.mat`, `df.vec`, and `df.scalar` units attached to that core via `df.core` (Section 2.4).

For each operator, *TileLoom* uses its `linalg` semantics to recover the parallel iteration space. This tells us, for each functional-unit type, how many intrinsic invocations of that type are independent and can, in principle, be issued in parallel. We then conceptually schedule these intrinsics onto the available parallel units of the same type: if there are N independent instances mapped to a unit type with U identical units, each capable of issuing r intrinsics per cycle, we approximate the operator's time contribution on that unit type as $N/(U \cdot r)$ cycles.

TileLoom then accounts for data dependencies and resource sharing among different unit types. Operators that are independent and target different unit types (for example, a matrix multiply on a matrix unit and a pointwise activation on a vector unit) can execute in parallel, whereas dependent operators or operators that compete for the same unit type must execute in sequence. The loop-body compute time is approximated as the sum over sequential segments, where each segment's time is the maximum over all operators that can run in parallel within that segment.

The potential parallelism exposed by this model is not necessarily fully achievable on a concrete microarchitecture, but the model does not attempt to exactly simulate the core's instruction scheduler. Instead, it is calibrated to be accurate enough to distinguish compute-bound from memory-bound mappings and to reason about overlap between compute and data movement. In our experiments, this coarse-grain modeling of compute is sufficient to discriminate between different dataflow schedules.

Compute–memory Overlap. Once we have an estimate for the loop body, we incorporate data movement. Let T_{load} be the time spent on all loads in one iteration of the loop body, T_{compute} the compute time (from the previous step), and T_{store} the time spent on all stores. We assume that each iteration executes as a pipelined load–compute–store sequence with double buffering: while iteration i is computing, the stores for iteration $i-1$ and the loads for iteration $i+1$ proceed in parallel whenever possible.

For an innermost loop with I iterations, the total execution time is approximated as:

$$T_{\text{loop}} \approx (I-2)\cdot \max\big(T_{\text{load}}+T_{\text{store}},\, T_{\text{compute}}\big) + \max\big(T_{\text{load}},\, T_{\text{compute}}\big) + \max\big(T_{\text{store}},\, T_{\text{compute}}\big) + T_{\text{load}} + T_{\text{store}}.$$

The first term accounts for the $I-2$ steady-state iterations, where load, compute, and store can overlap and the throughput is limited by the slower of T_{compute} and $T_{\text{load}}+T_{\text{store}}$. The remaining terms account for filling and draining the pipeline. This behavior is illustrated in Figure 4, where the k-body is executed in parallel with loading the next tile of B and storing the result tile of C from the previous iteration.

Concurrent data transfers and network traffic. Several

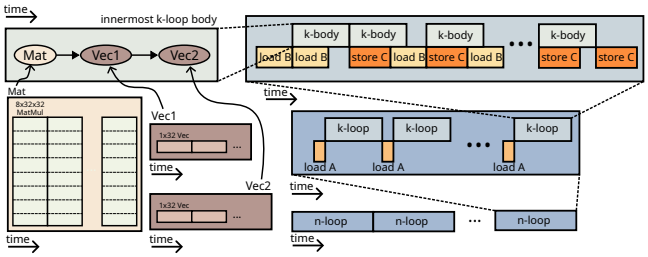


Figure 4: Example Evaluation of Pipelined execution.

memory operations may occur simultaneously and create traffic contention over the NoC. *TileLoom* estimates the effective bandwidth of each memory operation under this contention, using both the interconnect structure described in the hardware representation and the mapping annotations attached during memory operation mapping (Section 2.3). At that mapping step, each load or store is lowered either to a global load/-store or to a broadcast pattern, and the compiler records which subsets of the network it uses. For global loads, we assume accesses are sufficiently random that traffic is spread across the NoC links. For broadcasts, the resources used depend on the chosen pattern: on the 2D-mesh example, a broadcast that is performed independently along each row uses only the horizontal ring links (such as `%noc_h`), whereas a broadcast over the entire mesh may exercise both horizontal and vertical rings (both `%noc_h` and `%noc_v`). These choices are fixed during memory operation mapping and appear as annotations, as in Listings 3 and 5. Given these annotations, the performance model groups memory operations according to the network links and memory interfaces they occupy. For each group of operations that share a particular subset of links, it aggregates their offered traffic and derives an effective bandwidth per transfer by partitioning the nominal link bandwidth among them. Equivalently, the bandwidth seen by any one operation is reduced in proportion to the number of concurrent transfers using the same links or banks. The transfer time for each load or store is then computed from its tile size and effective bandwidth. The per-iteration load and store times, T_{load} and T_{store}, are obtained by combining these transfer times across operations, treating transfers on disjoint link sets as running in parallel and transfers on overlapping link sets as time-sharing the same resources. These T_{load} and T_{store} values are then plugged into the pipelined overlap model described above.

Candidate ranking through auto-profiling. *TileLoom*'s optimizations are highly dependent on the accuracy of the architectural models. When the architectural and micro-architecral information is not accurate enough, *TileLoom* supports a profiling-based autotuning step to maintain its performance. Concretely, for each candidate schedule, *TileLoom* combines the compute and data-movement estimates with the architectural model to obtain an approximate end-to-end execution time. This time reflects the balance between compute and communication, the benefit of spatial and temporal reuse, and the impact of NoC and memory contention on the concrete hardware described by `df`. *TileLoom* then ranks all candidates by this estimated time and keeps only the top-k dataflow mappings. Only these top-k candidates are handed to the back-end for full code generation and on-hardware profiling, where the final best-performing configuration (top-1) is selected. The choice of k controls the trade-off between compile time and the likelihood of including the true optimum: a larger k explores more mappings but costs more compilation and profiling time. We study this trade-off in Section 3.3.4.

3 Evaluation

3.1 Experimental Setup

Hardware platform. We conduct our experiments on Tenstorrent Wormhole and Blackhole cards. Table 1 summarizes the relevant hardware specifications. We compile *TileLoom* programs on a host machine equipped with dual 16-core Intel Xeon Gold 6326 CPUs and 512 GB of DRAM.

Table 1: Specifications of the Tenstorrent Wormhole and Blackhole used in our evaluation.

	TT-Wormhole	TT-Blackhole
Topology	8×8	12×10
On-chip SRAM	108 MB	180 MB
DRAM	12 GB GDDR6	32 GB GDDR6
Off-chip bw.	288 GB/s	512 GB/s
TFLOPS (FP16)	64	162
TBP	160 W	300 W
Technology Node	GF 12nm [52]	TSMC 6nm [53]

Architecture targets and modeling. Each Wormhole and Blackhole chip contains a 2D array of cores, with 8×8 cores on Wormhole and 12×10 cores on Blackhole, as shown in Figure 1. We model the Tenstorrent architecture in the `df` dialect, as described in Figure 1 and Section 2.4. Because the complete proprietary hardware specification is unavailable, we recover key parameters through isolated microbenchmarks, including matrix/vector-unit throughput and effective NoC and DRAM bandwidths. We instantiate these measured values in the `df` hardware representation used by *TileLoom*'s performance model.

Frontend. *TileLoom* currently supports Triton and Helion as kernel-development frontends. Although the core compiler is frontend-agnostic, our experiments use either Triton or Helion depending on kernel availability; the source code for all input kernels is available in our GitHub repository. For both frontends, we tune the tile, or block, shape using their existing Python-based autotuning stacks. For Triton, we use *triton-shared* [44] to lower kernels into MLIR, then apply a custom affinization pass that rewrites index arithmetic into affine expressions, followed by normalization into our dataflow-

agnostic MLIR format, as shown in Figure 2. For Helion, after Python-level tuning, we lower Helion Device IR into the same standard MLIR format using our custom lowering tool.

Backend. We lower our dataflow-aware MLIR to TT-Metalium, Tenstorrent's low-level C API, to generate the final executable, as shown in Figure 2. TT-Metalium exposes coarse-grained primitives for computation, synchronization, buffer allocation, and data movement, and handles most block- and core-level optimizations. *TileLoom* bridges the dataflow-level program and TT-Metalium by performing lifetime analysis over the block-level compute graph, using the resulting lifetimes to determine buffer allocation and synchronization among memory, compute, and data-movement operations. TT-Metalium then lowers these coarse-grained operations to hardware instructions.

3.2 End-to-End Performance

Table 2 reports the geometric mean of the relative performance of *TileLoom* over TTNN on four representative kernels: GEMM, FlashAttention, Flash Decode, and Mamba Chunk Scan linear attention. For each kernel, we evaluate a range of input shapes. The detailed shape-by-shape results are discussed in the corresponding subsections below. In this section, we report the performance of the top-ranked candidate selected by *TileLoom*'s performance model, without the final profiling-based tuning step. Therefore, the reported speedups come solely from *TileLoom*'s architectural modeling and dataflow-planning strategy.

Table 2: Relative performance of *TileLoom* over TTNN on GEMM, FlashAttention, Flash Decode, and Mamba Chunk Scan linear attention.

TileLoom vs. TTNN	Wormhole	Blackhole
vs. library implementation		
FlashAttention	1.94x	1.98x
Flash Decode	0.84x	0.87x
GEMM	0.95x	1.10x
vs. unfused implementation		
Mamba Chunk Scan	27.23x	16.27x

3.2.1 FlashAttention

We evaluate *TileLoom* on non-causal FlashAttention against the native TTNN implementation. The non-causal variant exposes more dataflow-optimization opportunities than the causal variant, making it a useful stress test for *TileLoom*'s spatial mapping strategy. We vary the number of attention heads between 64 and 128, sweep sequence length from 1024 to 16384, and adjust batch size to fit within DRAM capacity. As shown in Figure 5, *TileLoom* consistently achieves substantial speedups over TTNN, with 1.88–2.06× improvement in nearly every configuration. The gains come from exploiting reuse in the attention operands: *TileLoom* places tiles so that key tiles are reused on chip across multiple query and value tiles, reducing DRAM traffic compared with TTNN's default mapping, which repeatedly reloads these operands from DRAM.

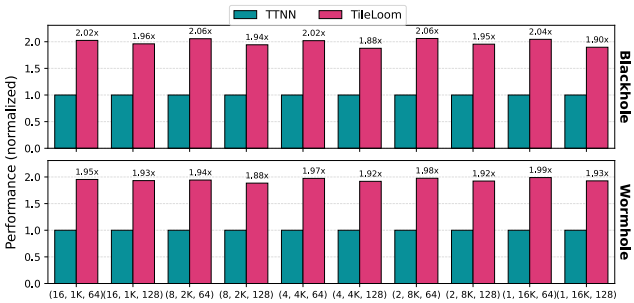


Figure 5: Relative Performance with *TileLoom* vs. TTNN on FlashAttention with parameters (batch size B, sequence length L, number of heads N).

3.2.2 FlashDecode

Flash Decode is algorithmically a special case of FlashAttention with query length one, but this changes the hardware mapping problem significantly. In full FlashAttention, *TileLoom* can exploit parallelism across query blocks and heads, while choosing mappings that improve key/value tile reuse. In Flash Decode, the query dimension no longer offers spatial parallelism, so parallelism mainly comes from the batch dimension and from splitting the key/value reduction across cores.

This yields a much smaller dataflow-planning space: there are fewer ways to distribute independent output tiles, and the main optimization is partitioning the key/value sequence and orchestrating the cross-core gather-reduce. Thus, this benchmark stresses *TileLoom*'s generated block-level code and reduction orchestration more than its ability to choose among diverse dataflow mappings.

As shown in Figure 6, *TileLoom* does not outperform the TTNN Flash Decode baseline, which is expected because TTNN uses scheduling and reduction optimizations specifically tuned for this operator. In contrast, *TileLoom* generates its implementation through a general compiler stack starting from tile-level kernel code. Even so, *TileLoom* achieves about 85% of TTNN performance on average, showing that it can produce competitive code even with limited dataflow optimization opportunity and a highly specialized vendor baseline.

3.2.3 Mamba Chunk Scan

Mamba Chunk Scan is a linear-attention kernel with substantial intra-block computation and nontrivial data movement.

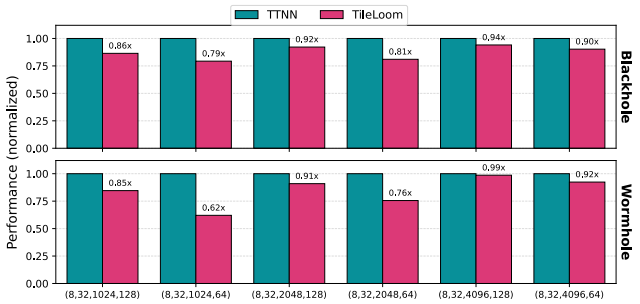


Figure 6: Relative performance of *TileLoom* compared with TTNN on Flash Decode with parameters (batch size B, head dimension H, sequence length L, hidden dimension D).

Since TTNN does not provide a fused implementation, we build the TTNN baseline by composing existing TTNN operations, yielding an unfused implementation. In contrast, *TileLoom* uses a fused tile-level Helion kernel.

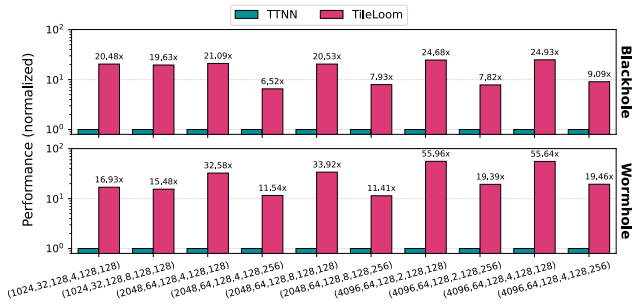


Figure 7: Relative performance of *TileLoom* vs. an unfused TTNN implementation on Mamba Chunk Scan with parameters sequence length L, number of heads N, head dimension H, number of groups G, hidden dimension D, and chunk size C.

This comparison shows a practical advantage of supporting tile-level kernel languages: fused kernels are often easier to obtain than highly optimized vendor-library implementations. Thus, *TileLoom* can generate efficient fused code even when the vendor library only exposes lower-level building blocks. As shown in Figure 7, *TileLoom* achieves 10x–55x speedup over the unfused TTNN baseline, reflecting both kernel fusion and *TileLoom*'s dataflow planning.

This large gap is typical between fused and unfused kernels, and similar trends appear in Tenstorrent's official technical reports. It is especially pronounced on dataflow architectures, which rely on on-chip NoC bandwidth for reuse and often have less off-chip bandwidth than HBM-based GPUs. Fusion and dataflow planning are therefore critical: without them, intermediate tensors are repeatedly written to and read from off-chip memory, causing severe slowdowns.

3.2.4 GEMM

We evaluate GEMM on over M, K, and N, ranging from 256 to 16384. GEMM is a key primitive, and the TTNN vendor implementation is already highly optimized, achieving about 70% of peak hardware throughput on average. Against this strong baseline, Table 2 shows that *TileLoom* delivers comparable performance using tile-level kernels while automatically generating low-level implementations.

For a more interpretable comparison, Figure 8 also reports two TTNN dataflow templates, TT-1D and TT-2D. TT-1D uses a 1D broadcast pattern: each core loads the smaller input matrix from global memory, while the other input is broadcast across the entire array. TT-2D uses a 2D broadcast pattern, streaming the two inputs across the mesh from the top and left. TTNN selects between these templates using a shape-dependent heuristic, with block size chosen by a separate strategy.

Figure 8 shows that *TileLoom* remains competitive with TTNN across the full GEMM sweep. This is notable because TTNN uses manually optimized library kernels, whereas *TileLoom* starts from tile-level kernels and searches the mapping space automatically. The TT-1D and TT-2D results show that fixed dataflows work well for regular GEMM shapes, where M, K, and N are similar, but can degrade on irregular shapes where the best mapping depends more sensitively on dimensions and hardware balance.

3.2.5 Shape Sensitivity on Irregular GEMM

To study shape sensitivity, we evaluate two irregular GEMM families. First, we fix $M = N = 32768$ and vary K from 256 to 2048. Second, we fix $M = K = 32768$ and vary N over the same range. Figure 9 shows the results.

When varying K (Figure 9a), *TileLoom* and TTNN follow trends similar to the 1D and 2D baselines. This is expected because K is mapped sequentially within each core, so changing it mainly affects intra-core compute cost and leaves limited room for inter-core dataflow optimization.

In contrast, varying N (Figure 9b) significantly changes the preferred dataflow. As N approaches M, the workload becomes more balanced across rows and columns, making 2D-like broadcast more attractive due to reuse along both mesh dimensions. When N is much smaller than M, 1D-like strategies are more effective, as reflected by the TT-1D and TT-2D results.

The transition between 1D- and 2D-favorable regimes depends on block size, compute-to-memory ratio, NoC overhead, and other hardware factors. This sensitivity motivates *TileLoom*'s cost-model-guided search, which accounts for spatial reuse, communication volume, and architectural constraints instead of relying on fixed heuristics.

These irregular-shape results also reveal a limitation of TTNN's heuristic selection: TTNN does not always choose

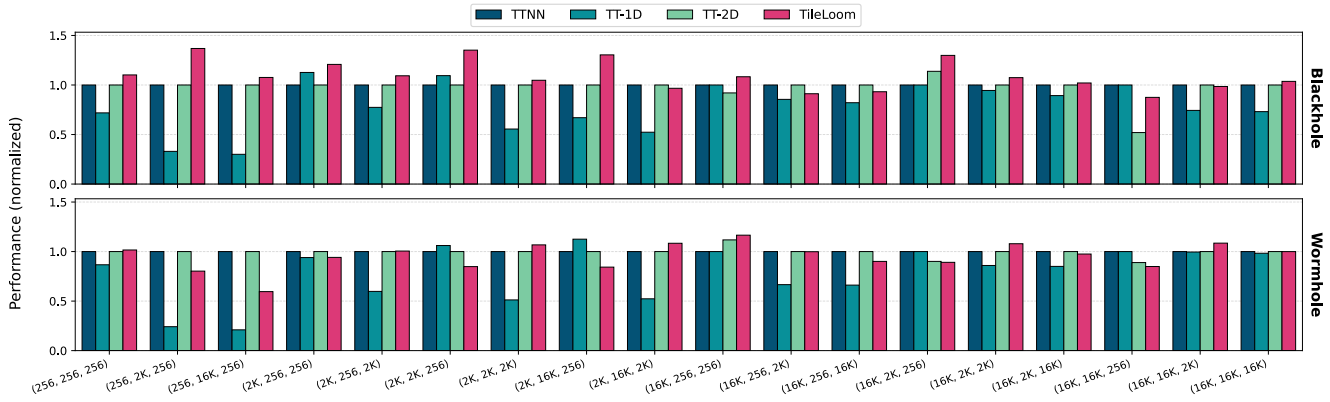


Figure 8: GEMM performance of *TileLoom* compared with TTNN. TT-1D and TT-2D are included as reference dataflow templates.

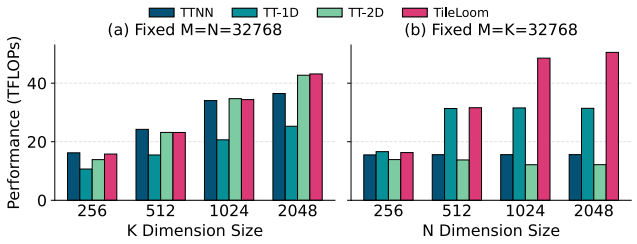


Figure 9: Performance comparison for GEMM under irregular input shapes.

the faster of its two templates. For example, in small-N, large-M cases, TTNN selects TT-2D even though TT-1D is faster, such as $M,K,N = 32K,32K,512/1024$. Our inspection suggests this comes from fixed thresholds over ratios such as M/N. Near these thresholds, the best strategy can also depend on tile size, bandwidth, and NoC cost, but the heuristic still makes a hard ratio-based decision.

In contrast, *TileLoom* searches beyond the two TTNN templates, exploring how logical dimensions map to spatial dimensions, where memory operations are hoisted in the loop nest, and which block sizes to use. This broader search allows *TileLoom* to find mappings that outperform both TTNN templates, as shown in Figure 8.

3.2.6 Wormhole vs. Blackhole

TileLoom generally achieves larger relative gains on Blackhole than on Wormhole. This is because Blackhole increases compute throughput more than off-chip bandwidth: peak FP16 throughput improves by 2.53×, while off-chip bandwidth improves by only 1.78× (Table 1). This shift makes kernels more likely to become memory-bound, increasing the value of *TileLoom*'s dataflow optimizations, which improve spatial reuse and reduce off-chip traffic.

GEMM shows this trend most clearly. On Wormhole, *TileLoom* reaches 0.95× TTNN performance. Profiling shows that GEMM is often compute-bound on Wormhole, so reducing off-chip traffic has limited end-to-end benefit. In addition, *TileLoom* starts from tile-level kernels and does not yet include all microarchitecture-specific optimizations used in TTNN's hand-tuned GEMM library, explaining the remaining 5% gap.

On Blackhole, the same GEMM evaluation improves to 1.10× over TTNN. Blackhole's higher compute-to-bandwidth ratio makes GEMM more sensitive to memory traffic and placement, allowing *TileLoom*'s dataflow planning to provide greater benefit. This shows that while *TileLoom* may not always beat a vendor GEMM library on a compute-limited device, its architectural modeling and spatial-reuse optimizations become more valuable as hardware shifts toward higher compute density.

3.3 Ablation Studies

3.3.1 Effect of Spatial Reuse

We quantify how much of *TileLoom*'s GEMM performance comes from spatial reuse by disabling the spatial-reuse pass and forcing all operands to be loaded from DRAM. Table 3 reports absolute performance with and without this optimization. Spatial reuse gives the largest gains on smaller GEMMs, where DRAM traffic is a major bottleneck. As size grows, the benefit decreases, consistent with the roofline model [69]: GEMM arithmetic intensity increases with matrix size, making larger problems more compute-bound. Once performance is limited by peak compute, reducing DRAM traffic has diminishing runtime impact.[3] Still, spatial reuse substantially reduces memory pressure: across these GEMM configurations, it cuts DRAM accesses by 70% on average. This does not

[3] Across all GEMM measurements, including configurations not shown here, effective throughput stabilizes around 45 TOP/s, indicating that many configurations operate near the compute roof.

always translate to proportional speedup in compute-bound regimes, but can improve memory headroom, reduce power, and leave more bandwidth for concurrent workloads.

Table 3: GEMM performance (TFLOP/s) on Wormhole with (*TileLoom*) and without (DRAM only) spatial reuse.

Configuration	DRAM only	*TileLoom*	Speedup
M=K=N=1024	11.15	23.70	2.12×
M=K=N=2048	16.77	33.96	2.03×
M=K=N=4096	22.96	40.44	1.76×
M=K=N=5120	29.19	41.61	1.42×
M=K=N=6144	28.22	42.47	1.51×

3.3.2 Effect of Temporal Reuse

Figure 10 shows the impact of temporal reuse across GEMM shapes. Temporal reuse buffers tiles locally so the same A or B tiles are reused across iterations instead of repeatedly loaded from DRAM.

We compare *TileLoom* with and without temporal reuse over different M and N values. As with spatial reuse, this optimization is most useful in memory-bound regimes, so we decrease K as M and N grow to keep the configurations memory-bound. In this setting, temporal reuse yields speedups up to 1.12×. The benefit increases with M and N, since larger dimensions create more opportunities to reuse the same A or B tiles. Thus, temporal reuse is most helpful when M or N is large and K is small. When it provides little benefit, *TileLoom*'s performance model deprioritizes those mappings, resulting in the same selected mapping and performance as the baseline without temporal reuse.

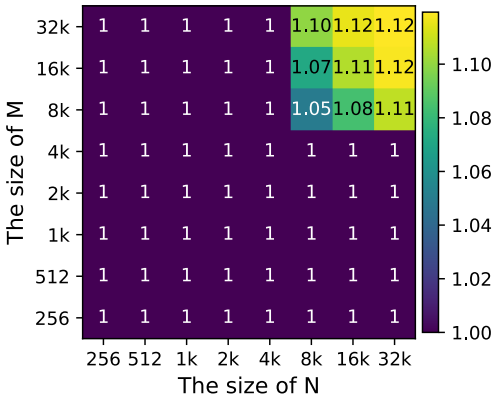


Figure 10: Normalized GEMM performance on Wormhole with and without temporal reuse.

3.3.3 Accuracy of *TileLoom*'s Performance Model Alone

We validate *TileLoom*'s performance model on Wormhole by comparing predicted throughput with measured GEMM performance across a wide range of (M,N,K) configurations. Figure 11 plots both estimates and measurements.

The predictions differ from measurements by 17% in geometric mean. However, the goal is not cycle accuracy, but reliable ranking and trend prediction, especially for transitions between memory- and compute-bound regimes as described by the roofline model [69]. As shown in Figure 11, the model captures these transitions well: it identifies when configurations become compute-bound and reflects the relative performance changes across shapes.

The end-to-end results in Table 2 and 4 further show that this error has limited impact on mapping selection. The model is accurate enough to rank candidates effectively, leaving the final optional profiling stage to choose among a small set of high-quality mappings.

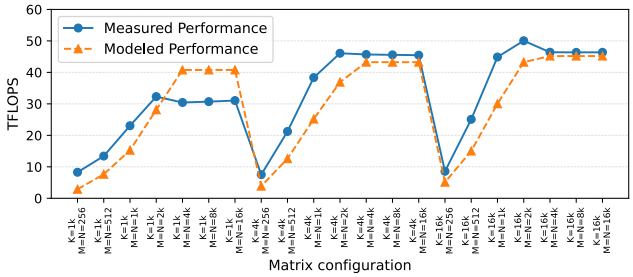


Figure 11: Validation of *TileLoom*'s performance model against measured GEMM performance.

3.3.4 Effect of top-k tuning and different topologies

As discussed in Section 2.5, *TileLoom* can optionally profile the top-k candidates generated by the compiler and select the best one. Thus, k trades compilation cost for final performance. We vary k from 1 to 5 and report geometric-mean normalized performance on Wormhole, along with compile time across hardware topologies, in Table 4. Here, top-1 means fully static compilation with only the best-predicted mapping, while larger k values add more profiled candidates.

We evaluate three topologies to test sensitivity to available parallelism and communication structure. The 8×8 mesh uses the full Wormhole core array, while the 4×8 mesh and 1×8 ring model smaller subsets. These reduced topologies change the balance between compute, NoC communication, and off-chip bandwidth, stressing whether mappings predicted for the full mesh remain effective under more constrained layouts.

Overall, k has a modest effect on performance. On the 8×8 mesh, top-5 outperforms top-1 by 7%, with most of the gain already achieved at top-2, which improves performance by 4.7%. The remaining gap mainly comes from occa-

sional model misrankings, where the predicted-best mapping is slightly worse than another candidate. In practice, a small k, such as 2 or 3, is usually enough to include the best mapping while keeping compile time moderate.

Similar trends hold on smaller topologies. The 1×8 ring is least sensitive to k because its restricted layout leaves fewer strong mappings to distinguish. The 4×8 and 8×8 meshes expose larger search spaces, so profiling a few extra candidates helps more. However, gains saturate quickly: beyond top-3, performance improves only marginally while compile time grows nearly linearly. This suggests that *TileLoom*'s static model effectively narrows the search to a small set of promising mappings, and a small profiling budget recovers most available performance across topology scales.

Table 4: Geometric-mean normalized performance (relative to TTNN) and compile time (seconds) of *TileLoom* under different top-k profiling settings. Top-1 corresponds to fully static compilation without additional profiling.

	1×8 Ring	4×8 Mesh	8×8 Mesh
top-1	-1.3% (6.36)	-5.4% (5.53)	-6.1% (5.75)
top-2	-1.3% (12.70)	-0.8% (11.29)	-1.4% (11.17)
top-3	-1.2% (19.07)	+0.5% (17.16)	-0.5% (16.65)
top-4	-0.4% (25.47)	+1.1% (23.03)	+0.4% (22.17)
top-5	-0.3% (31.82)	+1.3% (28.91)	+0.9% (27.66)

4 Related Works

Hardware modeling and co-design. There exist works on modeling and co-design of spatial accelerators: CGRAs, spatial FPGAs, and systolic arrays, where the computes and control are at a finer granularity than *TileLoom*. Languages and frameworks such as Spatial [31], Plasticine [56], T2S [59], Halide(-to-Hardware) [57], and HeteroCL [35] compile loop nests or functional pipelines into arrays of processing elements (PEs) and local memories, often co-designing the overlay itself. Polyhedral and tensor-centric systems such as AutoSA [65], TensorLib [24], and Rubick [39, 41] similarly start from affine loop nests or tensor expressions and derive space–time mappings that synthesize systolic or tensor arrays, buffers, and controllers. Co-design tools like Timeloop/Accelergy [51] and AMOS [74] explore dataflows, tilings, and memory hierarchies for DNN accelerators, while MAESTRO/-MAERI [32, 33] analytically model buffer and network usage for specialized systolic-style designs. In all of these systems, the modeling unit is a PE, buffer, or loop level in the memory hierarchy, and the goal is to search or synthesize hardware at that granularity. *TileLoom*, in contrast, assumes intra-core microarchitecture and local mapping are fixed, treats each core as the atomic unit, and models only the multi-level memory system and NoC *between* cores to decide how tile instances are distributed across space and time.

Software mapping and compilation for spatial architectures. On the software side, classical CGRA and FPGA flows treat mapping as a place-and-route problem on a PE-level dataflow graph: frameworks such as DSAGEN [67], ML-oriented CGRA compilers like ML-CGRA and MLIR-to-CGRA [40, 72], and architecture-agnostic mappers such as Morpher [68] and CaSMap [42] perform placement, routing, and modulo scheduling onto a fixed fabric. Loop- and polyhedral-based tools including Timeloop/Accelergy [51], and MAESTRO/MAERI [32, 33] reuse loop schedules or space–time mappings as the scheduling representation but are primarily design-space exploration tools: they evaluate mappings analytically. Research compilers for specific spatial architecture, such as AMOS [74], LISA [37], and system-level compilers for wafer-scale fabrics [20, 62], do generate per-core programs and communication schedules, but are typically tailored to a particular architecture family with baked-in mapping heuristics.

5 Conclusion

TileLoom demonstrates that compiler-driven mapping can deliver competitive performance on spatial dataflow accelerators while substantially reducing the need for hand-written, hardware-specific kernels. Starting from high-level tile-centric kernels, TileLoom automatically selects spatial and temporal mappings, data-reuse strategies, and communication patterns using a dataflow dialect that uniformly represents cores, memories, and interconnect.

Across Tenstorrent Wormhole and Blackhole, TileLoom achieves strong results on FlashAttention, Mamba Chunk Scan, GEMM, and Flash Decode. It outperforms TTNN when dataflow planning and fusion expose substantial opportunities, and remains competitive with highly optimized vendor implementations when the optimization space is more constrained. These results show that many decisions traditionally embedded in hand-engineered kernels can instead be handled by a reusable compiler stack, making optimized kernel development more accessible and providing a foundation for future spatial dataflow architectures.

Acknowledgment

This work is partially supported by the Advanced Research and Technology Innovation Centre (ARTIC), the National University of Singapore under Grant AFP-RP6, and by National Research Foundation, Singapore, under its Competitive Research Program Award NRF-CRP23-2019-0003 and the Ministry of Education, Singapore, under Tier 3 grant MOE-MOET32024-0003.

References

[1] Dennis Abts, Garrin Kimmell, Andrew C. Ling, John Kim, et al. A software-defined tensor streaming multiprocessor for large-scale machine learning. In *Proceedings of the 49th Annual International Symposium on Computer Architecture (ISCA 2022)*, pages 567–580, 2022.

[2] Dennis Abts, Jonathan Ross, Jonathan Sparling, Mark Wong-VanHaren, et al. Think fast: A tensor streaming processor (TSP) for accelerating deep learning workloads. In *Proceedings of the 47th Annual International Symposium on Computer Architecture (ISCA 2020)*, pages 145–158, 2020.

[3] Jenny Lynn Almerol, Elisabetta Boella, Mario Spera, and Daniele Gregori. Accelerating gravitational *N*-body simulations using the RISC-V-based tenstorrent wormhole™. *arXiv preprint*, arXiv:2509.19294, 2025.

[4] Luca Benini and Giovanni De Micheli. Networks on chips: A new SoC paradigm. *Computer*, 35(1):70–78, 2002.

[5] Randal E Bryant. Data-intensive supercomputing: The case for disc. 2007.

[6] Nafea Bshara. Aws trainium: the journey for designing and optimization full stack ml hardware. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, pages 4–4, 2024.

[7] Doug Burger, James R Goodman, and Alain Kägi. Memory bandwidth limitations of future microprocessors. *ACM SIGARCH Computer Architecture News*, 24(2):78–89, 1996.

[8] Cerebras Systems. Cerebras systems: Achieving industry best AI performance through a systems approach. Technical report, Cerebras Systems, 2021. Whitepaper 03.

[9] Cerebras Systems. The cerebras software development kit: A technical overview. Whitepaper, 2023.

[10] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Meghan Cowan, Haichen Shen, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. Tvm: An automated end-to-end optimizing compiler for deep learning. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 578–594, 2018.

[11] Sharan Chetlur, Cliff Woolley, Philippe Vandermersch, Jonathan Cohen, John Tran, Bryan Catanzaro, and Evan Shelhamer. cuDNN: Efficient primitives for deep learning. *CoRR*, abs/1410.0759, 2014.

[12] Zeshan Chishti and Berkin Akin. Memory system characterization of deep learning workloads. In *Proceedings of the International Symposium on Memory Systems*, pages 497–505, 2019.

[13] Yaoyao Ding, Bohan Hou, Xiao Zhang, Allan Lin, Tianqi Chen, Cody Hao Yu, Yida Wang, and Gennady Pekhimenko. Tilus: A tile-level GPGPU programming language for low-precision computation. *arXiv preprint arXiv:2504.12984*, 2025.

[14] Amin Firoozshahian, Joel Coburn, Roman Levenstein, Rakesh Nattoji, Ashwin Kamath, Olivia Wu, Gurdeepak Grewal, Harish Aepala, Bhasker Jakka, Bob Dreyer, et al. Mtia: First generation silicon targeting meta's recommendation systems. In *Proceedings of the 50th Annual International Symposium on Computer Architecture*, pages 1–13, 2023.

[15] Amir Gholami, Zhewei Yao, Sehoon Kim, Coleman Hooper, Michael W Mahoney, and Kurt Keutzer. Ai and memory wall. *IEEE Micro*, 44(3):33–39, 2024.

[16] Graphcore Ltd. Poplar graph framework software. https://www.graphcore.ai/products/poplar, 2022. Accessed: 2024-03-19.

[17] Sumanth Gudaparthi, Sarabjeet Singh, Surya Narayanan, Rajeev Balasubramonian, and Visvesh Sathe. CANDLES: Channel-aware novel dataflow-microarchitecture co-design for low energy sparse neural network acceleration. In *2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 876–891, 2022.

[18] Waqas Gul, Maitham Shams, and Dhamin Al-Khalili. Sram cell design challenges in modern deep sub-micron technologies: An overview. *Micromachines*, 13(8):1332, 2022.

[19] James Hamilton. Tesla project dojo overview. https://perspectives.mvdirona.com/2021/08/tesla-project-dojo-overview/, 2021. Blog post.

[20] Congjie He, Yeqi Huang, Pei Mu, Mike Wang, Ziming Miao, Jilong Xue, Lingxiao Ma, Fan Yang, and Luo Mai. Wafer-scale ai compute: A system software perspective.

[21] Ron Ho, Kenneth W. Mai, and Mark A. Horowitz. The future of wires. *Proceedings of the IEEE*, 89(4):490–504, 2001.

[22] Mark Horowitz. 1.1 computing's energy problem (and what we can do about it). In *2014 IEEE international solid-state circuits conference digest of technical papers (ISSCC)*, pages 10–14. IEEE, 2014.

[23] Yuanming Hu, Tzu-Mao Li, Luke Anderson, Jonathan Ragan-Kelley, and Frédo Durand. Taichi: A language for high-performance computation on spatially sparse data structures. *ACM Transactions on Graphics*, 38(6), 2019.

[24] Liancheng Jia, Zizhang Luo, Liqiang Lu, and Yun Liang. Tensorlib: A spatial accelerator generation framework for tensor algebra. In *2021 58th ACM/IEEE Design Automation Conference (DAC)*, pages 865–870. IEEE, 2021.

[25] Zhe Jia, Marco Maggioni, Benjamin Staiger, and Daniele P. Scarpazza. Dissecting the NVIDIA volta GPU architecture via microbenchmarking. *arXiv preprint*, arXiv:1804.06826, 2018.

[26] Zhe Jia, Blake Tillman, Marco Maggioni, and Daniele Paolo Scarpazza. Dissecting the graphcore IPU architecture via microbenchmarking. *arXiv preprint*, arXiv:1912.03413, 2019.

[27] Norman P Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, et al. In-datacenter performance analysis of a tensor processing unit. In *Proceedings of the 44th annual international symposium on computer architecture*, pages 1–12, 2017.

[28] Jehandad Khan, Paul Fultz, Artem Tamazov, Daniel Lowell, Chao Liu, Michael Melesse, Murali Nandhimandalam, Kamil Nasyrov, Ilya Perminov, Tejash Shah, Vasilii Filippov, Jing Zhang, Jing Zhou, Bragadeesh Natarajan, and Mayank Daga. MIOpen: An open source library for deep learning primitives. *CEUR Workshop Proceedings*, 2744, 2020.

[29] Khronos Group. *The OpenCL Specification, Version 3.0*, 2020. Available from the Khronos OpenCL Registry.

[30] David B. Kirk and Wen mei W. Hwu. *Programming Massively Parallel Processors: A Hands-on Approach*. Morgan Kaufmann, 2010.

[31] David Koeplinger, Matthew Feldman, Raghu Prabhakar, Yaqi Zhang, Stefan Hadjis, Ruben Fiszel, Tian Zhao, Luigi Nardi, Ardavan Pedram, Christos Kozyrakis, et al. Spatial: A language and compiler for application accelerators. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 296–311, 2018.

[32] Hyoukjun Kwon, Prasanth Chatarasi, Vivek Sarkar, Tushar Krishna, Michael Pellauer, and Angshuman Parashar. Maestro: A data-centric approach to understand reuse, performance, and hardware cost of dnn mappings. *IEEE micro*, 40(3):20–29, 2020.

[33] Hyoukjun Kwon, Ananda Samajdar, and Tushar Krishna. A communication-centric approach for designing flexible DNN accelerators. *IEEE Micro*, 38(6):25–35, 2018.

[34] Yongin Kwon, JooHyoung Cha, Sehyeon Oh, Misun Yu, Jeman Park, and Jemin Lee. Luthier: Bridging autotuning and vendor libraries for efficient deep learning inference. *ACM Transactions on Embedded Computing Systems*, 24(5s), 2025.

[35] Yi-Hsiang Lai, Yuze Chi, Yuwei Hu, Jie Wang, Cody Hao Yu, Yuan Zhou, Jason Cong, and Zhiru Zhang. Heterocl: A multi-paradigm programming infrastructure for software-defined reconfigurable computing. In *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 242–251, 2019.

[36] Jonathan S. Lew, Deval A. Shah, Suchita Pati, Shaylin Cattell, Mengchi Zhang, Amruth Sandhupatla, Christopher Ng, Negar Goli, Matthew D. Sinclair, Timothy G. Rogers, and Tor M. Aamodt. Analyzing machine learning workloads using a detailed GPU simulator. *CoRR*, abs/1811.08933, 2018.

[37] Zhaoying Li, Dan Wu, Dhananjaya Wijerathne, and Tulika Mitra. Lisa: Graph neural network based portable mapping on spatial accelerators. In *2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 444–459. IEEE, 2022.

[38] Andrea Lottarini, João P. Cerqueira, Thomas J. Repetti, Stephen A. Edwards, Kenneth A. Ross, Mingoo Seok, and Martha A. Kim. Master of none acceleration: A comparison of accelerator architectures for analytical query processing. In *Proceedings of the 46th Annual International Symposium on Computer Architecture (ISCA)*, pages 762–773, 2019.

[39] Liqiang Lu, Zizhang Luo, Size Zheng, Jieming Yin, Jason Cong, Yun Liang, and Jianwei Yin. Rubick: A unified infrastructure for analyzing, exploring, and implementing spatial architectures via dataflow decomposition. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 43(4):1177–1190, 2023.

[40] Yixuan Luo, Cheng Tan, Nicolas Bohm Agostini, Ang Li, Antonino Tumeo, Nirav Dave, and Tong Geng. Ml-cgra: An integrated compilation framework to enable efficient machine learning acceleration on cgras. In *2023 60th ACM/IEEE Design Automation Conference (DAC)*, pages 1–6. IEEE, 2023.

[41] Zizhang Luo, Liqiang Lu, Size Zheng, Jieming Yin, Jason Cong, Jianwei Yin, and Yun Liang. Rubick: A synthesis framework for spatial architectures via dataflow

decomposition. In *2023 60th ACM/IEEE Design Automation Conference (DAC)*, pages 1–6. IEEE, 2023.

[42] Xingchen Man, Jianfeng Zhu, Guihuan Song, Shouyi Yin, Shaojun Wei, and Leibo Liu. Casmap: agile mapper for reconfigurable spatial architectures by automatically c lustering intermediate representations a nd s cattering mapping process. In *Proceedings of the 49th Annual International Symposium on Computer Architecture*, pages 259–273, 2022.

[43] John D McCalpin et al. Memory bandwidth and machine balance in current high performance computers. *IEEE computer society technical committee on computer architecture (TCCA) newsletter*, 2(19-25), 1995.

[44] Microsoft. triton-shared: A shared middle-layer for the triton compiler. https://github.com/microsoft/triton-shared, 2025.

[45] Pengyu Mu, Yi Liu, Rui Wang, Guoxiang Liu, Hangcheng An, Qianhe Zhao, Hailong Yang, Chenhao Xie, Zhongzhi Luan, Chunye Gong, and Depei Qian. Deep learning operators performance tuning for changeable sized input data on tensor accelerate hardware. *IEEE Transactions on Computers*, 74(6):2101–2113, 2025.

[46] Onur Mutlu. Memory scaling: A systems architecture perspective. In *2013 5th IEEE International Memory Workshop*, pages 21–25. IEEE, 2013.

[47] NASA Advanced Supercomputing Division. Basics on NVIDIA GPU hardware architecture. https://www.nas.nasa.gov/hecc/support/kb/basics-on-nvidia-gpu-hardware-architecture_704.html, 2025. HECC Knowledge Base Article 704.

[48] Tim Noack, Louis Krüger, and Andreas Koch. Accelerating sparse linear solvers on intelligence processing units. In *Proceedings of the 39th IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 1023–1035, 2025.

[49] NVIDIA Corporation. *CUDA C Programming Guide*, 2017. PG-02829-001_v8.0.

[50] Nvidia Corporation. Nvidia cuda tile. https://developer.nvidia.com/cuda/tile, 2025. Accessed: 2025-12-6.

[51] Angshuman Parashar, Priyanka Raina, Yakun Sophia Shao, Yu-Hsin Chen, Victor A. Ying, Anurag Mukkara, Rangharajan Venkatesan, Brucek Khailany, Stephen W. Keckler, and Joel Emer. Timeloop: A systematic approach to DNN accelerator evaluation. In *2019 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 304–315, 2019.

[52] Dylan Patel. Tenstorrent wormhole analysis - a scale out architecture for machine learning that could put nvidia on their back foot, June 2021.

[53] Dylan Patel. Tenstorrent blackhole, grendel, and buda - a scale out architecture for sparsity, conditional execution, and dynamic routing, April 2022.

[54] Hongwu Peng, Caiwen Ding, Tong Geng, Sutanay Choudhury, Kevin Barker, and Ang Li. Evaluating emerging AI/ML accelerators: IPU, RDU, and NVIDIA/AMD GPUs. *arXiv preprint arXiv:2311.04417*, 2024.

[55] Raghu Prabhakar, Sumti Jairath, and Jinuk Luke Shin. Sambanova sn10 RDU: A 7nm dataflow architecture to accelerate software 2.0. In *2022 IEEE International Solid-State Circuits Conference (ISSCC)*, pages 350–352, 2022.

[56] Raghu Prabhakar, Yaqi Zhang, David Koeplinger, Matt Feldman, Tian Zhao, Stefan Hadjis, Ardavan Pedram, Christos Kozyrakis, and Kunle Olukotun. Plasticine: A reconfigurable architecture for parallel patterns. In *Proceedings of the 44th Annual International Symposium on Computer Architecture (ISCA)*, pages 389–402, 2017.

[57] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. Halide: A language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 519–530, 2013.

[58] SambaNova Systems. Accelerated computing with a reconfigurable dataflow architecture. Technical report, SambaNova Systems, 2021. Whitepaper.

[59] Nitish Srivastava, Hongbo Rong, Prithayan Barua, Guanyu Feng, Huanqi Cao, Zhiru Zhang, David Albonesi, Vivek Sarkar, Wenguang Chen, Paul Petersen, et al. T2s-tensor: Productively generating high-performance spatial hardware for dense tensor computations. In *2019 IEEE 27th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 181–189. IEEE, 2019.

[60] Moritz Thüning. Attention in sram on tenstorrent grayskull. *arXiv preprint arXiv:2407.13885*, 2024.

[61] Philippe Tillet, H. T. Kung, and David Cox. Triton: An intermediate language and compiler for tiled neural network computations. In *Proceedings of the 3rd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages (MAPL)*, 2019.

[62] Dirk Van Essendelft, Patrick Wingo, Terry Jordan, Ryan Smith, and Wissam Saidi. A system level compiler for massively-parallel, spatial, dataflow architectures. *arXiv preprint arXiv:2506.15875*, 2025.

[63] Dirk Van Essendelft, Patrick Wingo, Terry Jordan, Ryan Smith, and Wissam A. Saidi. A system level compiler for massively-parallel, spatial, dataflow architectures. *arXiv preprint arXiv:2506.15875*, 2025.

[64] Erwei Wang, Samuel Bayliss, Andra Bisca, Zachary Blair, Sangeeta Chowdhary, Kristof Denolf, Jeff Fifield, Brandon Freiberger, Erika Hunhoff, Phil James-Roxby, et al. From loop nests to silicon: Mapping ai workloads onto amd npus with mlir-air. *arXiv preprint arXiv:2510.14871*, 2025.

[65] Jie Wang, Licheng Guo, and Jason Cong. Autosa: A polyhedral compiler for high-performance systolic arrays on fpga. In *The 2021 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 93–104, 2021.

[66] Lei Wang, Yu Cheng, Yining Shi, Zhengju Tang, Zhiwen Mo, Wenhao Xie, Lingxiao Ma, Yuqing Xia, Jilong Xue, Fan Yang, and Zhi Yang. Tilelang: A composable tiled programming model for AI systems. *arXiv preprint arXiv:2504.17577*, 2025.

[67] Jian Weng, Sihao Liu, Vidushi Dadu, Zhengrong Wang, Preyas Shah, and Tony Nowatzki. Dsagen: Synthesizing programmable spatial accelerators. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, pages 268–281. IEEE, 2020.

[68] Dhananjaya Wijerathne, Zhaoying Li, Manupa Karunaratne, Li-Shiuan Peh, and Tulika Mitra. Morpher: An open-source integrated compilation and simulation framework for cgra. In *Fifth Workshop on Open-Source EDA Technology (WOSET)*, 2022.

[69] Samuel Williams, Andrew Waterman, and David Patterson. Roofline: an insightful visual performance model for multicore architectures. *Communications of the ACM*, 52(4):65–76, 2009.

[70] Wm A Wulf and Sally A McKee. Hitting the memory wall: Implications of the obvious. *ACM SIGARCH computer architecture news*, 23(1):20–24, 1995.

[71] Jiaqi Yang, Hao Zheng, and Ahmed Louri. DiTile-DGNN: An efficient accelerator for distributed dynamic graph neural network inference. In *Proceedings of the 52nd Annual International Symposium on Computer Architecture (ISCA)*, pages 1240–1253, 2025.

[72] Tianyi Yu, Omar Ragheb, Stephen Wicklund, and Jason Anderson. Mlir-to-cgra: A versatile mlir-based compileir framework for cgras. In *2024 IEEE 35th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, pages 184–192. IEEE, 2024.

[73] Jinming Zhang, Xi Fan, Yaoyao Ye, Xuyan Wang, Guojie Xiong, Xianglun Leng, Ningyi Xu, Yong Lian, and Guanghui He. INDM: Chiplet-based interconnect network and dataflow mapping for DNN accelerators. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 43(4):1107–1120, 2024.

[74] Size Zheng, Renze Chen, Anjiang Wei, Yicheng Jin, Qin Han, Liqiang Lu, Bingyang Wu, Xiuhong Li, Shengen Yan, and Yun Liang. Amos: enabling automatic mapping for tensor computations on spatial accelerators with hardware abstraction. In *Proceedings of the 49th Annual International Symposium on Computer Architecture*, pages 874–887, 2022.

[75] Jinming Zhuang, Shaojie Xiang, Hongzheng Chen, Niansong Zhang, Zhuoping Yang, Tony Mao, Zhiru Zhang, and Peipei Zhou. Aries: An agile mlir-based compilation flow for reconfigurable devices with ai engines. In *Proceedings of the 2025 ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, pages 92–102, 2025.

ARTIFACT EVALUATED
USENIX
AVAILABLE

MPK: A Compiler and Runtime for Mega-Kernelizing Tensor Programs

Xinhao Cheng[1,*] Zhihao Zhang[1,*] Yu Zhou[1,*] Jianan Ji[1,*] Jinchen Jiang[2] Zepeng Zhao[1]
Ziruo Xiao[1] Zihao Ye[3] Yingyi Huang[3] Ruihang Lai[1] Hongyi Jin[1] Bohan Hou[1]
Mengdi Wu[1] Yixin Dong[1] Anthony Yip[1] Zihao Ye[4] Songting Wang[1] Wenqin Yang[5]
Xupeng Miao[6] Tianqi Chen[1,3] Zhihao Jia[1]

Carnegie Mellon University[1] *Tsinghua University*[2] *NVIDIA*[3]
University of Michigan[4] *Independent Researcher*[5] *Peking University*[6]

Abstract

We introduce Mirage Persistent Kernel (MPK), the first compiler and runtime system that automatically transforms multi-GPU model inference into a single high-performance mega-kernel. MPK introduces an SM-level graph representation that captures data dependencies at the granularity of individual streaming multiprocessors (SMs), enabling cross-operator software pipelining, fine-grained overlap of computation and communication, and other optimizations that are infeasible under the conventional kernel-per-operator execution model. The MPK compiler lowers tensor programs into optimized SM-level task graphs and generates fast CUDA implementations for each task, while the MPK in-kernel parallel runtime executes these tasks within a single persistent mega-kernel using decentralized scheduling across SMs. Together, these components provide end-to-end kernel fusion with minimal developer effort, while preserving the flexibility of existing programming models. Our evaluation shows that MPK significantly outperforms existing kernel-per-operator LLM serving systems, achieving up to 1.7× lower end-to-end inference latency and pushing LLM inference performance close to the limits of the underlying hardware. MPK is publicly available at https://github.com/mirage-project/mirage.

1 Introduction

Enabling high-performance inference of ML models on GPUs is critical for modern AI applications, since inference latency directly affects both user experience and serving cost. Today's ML systems generally express model computation as a tensor program structured as a directed acyclic graph, whose nodes denote tensor algebra operators (e.g., matrix multiplication) and whose edges represent tensors, i.e., the n-dimensional arrays produced and consumed by these operators.

Most existing systems execute each operator using a dedicated GPU kernel, either hand-optimized by domain experts [18, 43] or generated automatically by ML compilers [15, 36, 41]. However, this *kernel-per-operator* execution model limits several key cross-operator GPU optimizations.

First, modern GPUs impose an implicit *kernel barrier* between consecutive launches on the same stream to ensure that all threads from the previous kernel complete before any thread from the next kernel begins. While this mechanism correctly enforces data dependencies, it prevents *cross-operator software pipelining* and forces dependent operators to execute strictly sequentially. NVIDIA recently introduced *programmatic dependent launch* (PDL) [11], which allows partial overlap between kernels on the same stream. However, adopting PDL requires significant engineering effort, as it fundamentally alters kernel structure and control flow.

Second, the kernel-per-operator execution model prevents fine-grained compute-communication overlap. Since dependencies are represented only at the coarse granularity of operators, the runtime must enforce full-operator completion before launching dependent communication or computation. For example, when a matrix multiplication is followed by an all-reduce in separate kernels, the all-reduce must wait for the entire matrix multiplication to complete, even though each fragment of the all-reduce depends only on a subset of the multiplication output. Exploiting such opportunities requires representing and enforcing dependencies at a granularity finer than individual kernels.

Finally, kernel-per-operator execution may require launching hundreds to thousands of kernels for each inference iteration. To reduce launch overhead, current systems rely heavily on *CUDA Graphs*, which capture a sequence of GPU operations and replay them with low overhead. However, CUDA Graphs are largely static: any changes to control flow, tensor shapes, or data dependencies require re-instantiating or modifying the captured graph, limiting their flexibility for the dynamic workloads commonly seen in model inference.

A promising approach to overcoming these limitations is to fuse all computation and communication of model inference into a single *mega-kernel*, also known as a *persistent kernel*. In this design, the system launches one GPU kernel to execute the entire model, including layer computations and inter-GPU

*Equal contribution.

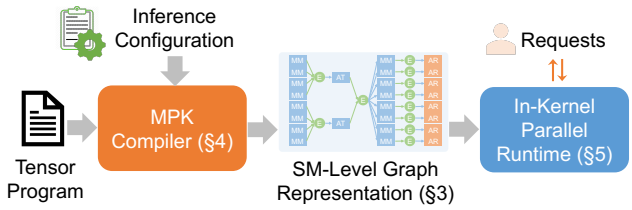


Figure 1: An overview of MPK.

communication, without interruption.

Mega-kernels address the limitations of kernel-per-operator execution in several ways. First, they eliminate repeated kernel launch overhead by replacing many operator-level launches with a single kernel invocation. Second, by fusing *all* operators into one kernel, they enable cross-operator software pipelining, allowing data for the next operator to be prefetched while computation for the current operator is still in progress. Third, they support fine-grained overlap of computation and inter-GPU communication, enabling concurrent execution that more effectively hides communication latency.

Despite these benefits, automatically transforming an ML model into a high-performance mega-kernel remains challenging. Existing ML systems—such as PyTorch [29], Triton [36], and TVM [15]—do not support end-to-end mega-kernel generation. Moreover, these systems rely on a fragmented ecosystem of specialized libraries: NCCL [4] or NVSHMEM [10] for communication, FlashInfer [43] or FlashAttention [18] for attention, and CUDA or Triton for custom computation. This fragmentation makes it difficult to unify the entire inference pipeline within a single kernel.

We present *Mirage Persistent Kernel* (MPK), the first compiler and runtime system that automatically transforms multi-GPU model inference into a high-performance mega-kernel. MPK enables end-to-end kernel fusion with minimal developer effort: users can mega-kernelize a PyTorch model with only a few lines of code while achieving significant performance improvements compared to running the model in vanilla PyTorch with CUDA Graphs and `torch.compile`. MPK combines the performance benefits of mega-kernels with the usability of existing ML frameworks.

A key idea in MPK is to represent computation and inter-GPU communication at the granularity of individual streaming multiprocessors (SMs), rather than at the granularity of an entire GPU. MPK introduces an *SM-level graph representation*, called *t*Graph, whose nodes denote *tasks* running on individual SMs and whose edges encode fine-grained dependencies between tasks. This representation exposes additional parallelism and enables optimizations such as cross-operator software pipelining and fine-grained kernel overlap, which are difficult to realize in conventional kernel-per-operator execution models. MPK realizes this idea using two key components shown in Figure 1.

The MPK compiler. The MPK compiler takes a tensor program and an inference configuration as input and automatically transforms the program's computation graph into an optimized SM-level *t*Graph tailored to the given inference configuration and GPU architecture. The compiler applies a range of optimizations, including event fusion, graph normalization, and graph linearization, to reduce synchronization overhead and improve the performance of generated *t*Graphs. In addition, MPK automatically generates fast CUDA implementations for individual tasks using existing superoptimization techniques [41], ensuring efficient SM-level execution.

In-kernel parallel runtime. MPK executes the SM-level *t*Graph using an in-kernel parallel runtime embedded entirely within a mega-kernel, enabling fine-grained control over task execution and scheduling *without* additional kernel launches during model execution. To achieve this goal, the runtime partitions a GPU's SMs into *workers* and *schedulers*. Each worker maintains a dedicated task queue and executes assigned tasks in a first-in-first-out order, while schedulers track dependencies across tasks and dispatch tasks once their prerequisites are satisfied. The MPK runtime uses an *event-driven, fully asynchronous* execution model to keep GPUs highly utilized. Finally, the runtime uses a *hybrid task-launch strategy* that combines just-in-time and ahead-of-time dispatch to minimize runtime overhead while preserving dynamic load balance across SMs.

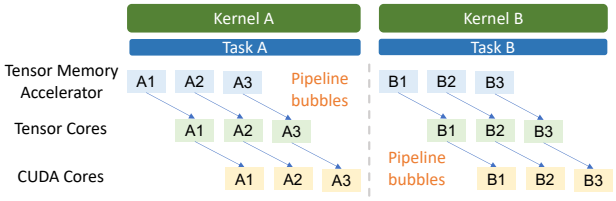
Evaluation results. We implement MPK as a PyTorch compiler backend: a PyTorch program can be compiled into an MPK mega-kernel with only a few lines of code changes. We evaluate MPK on five widely used models across three generations of NVIDIA GPUs: A100, H100, and B200. Even for workloads widely deployed and heavily optimized by existing kernel-per-operator systems, such as SGLang and vLLM for LLM serving, MPK outperforms current systems by 1.0–1.7× on both single- and multi-GPU deployments, pushing LLM inference performance close to hardware limits.

2 Background

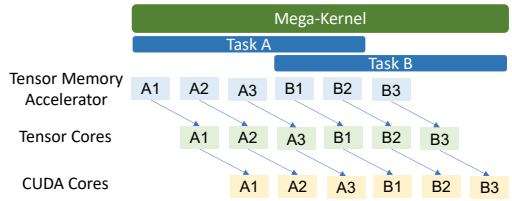
This section first reviews the kernel-oriented GPU programming model and its limitations (§ 2.1), and then presents kernel fusion and mega-kernel techniques (§ 2.2), which motivate the design of MPK.

2.1 GPU Programming Model

On GPUs, computations are organized as *kernels*, each representing a function executed concurrently across many cores in a single-program, multiple-data (SPMD) fashion. A kernel consists of a grid of *thread blocks*, where each block is scheduled on a streaming multiprocessor (SM) and contains mul-



(a) Kernel barriers prevent cross-task pipelining.



(b) MPK enables both intra- and cross-task pipelining.

Figure 2: Comparing how MPK and existing approaches support intra- and cross-task pipelining.

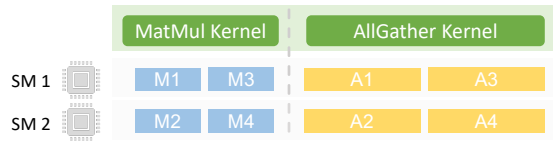
tiple *threads* that operate on individual data elements. Each thread has a private register file, while threads within the same block can cooperate through low-latency *shared memory* for data exchange and collective operations. All kernel inputs and outputs are stored in GPU *device memory*.

The conventional GPU programming model does not support direct synchronization across thread blocks within a kernel, because thread blocks are scheduled independently across SMs and may not all be resident simultaneously. As a result, cross-operator dependencies are enforced through *kernel barriers*, which are automatically inserted by the GPU runtime between consecutive kernels launched on the same stream.

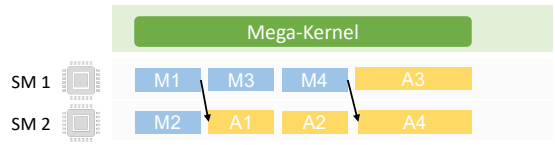
While kernel barriers simplify dependency management, they also prevent key GPU optimizations such as cross-kernel software pipelining and fine-grained operator overlap.

Software pipelining. GPU architectures are increasingly *heterogeneous*, integrating specialized accelerators such as tensor cores and tensor memory accelerators (TMAs). Since TMA load and store instructions execute asynchronously, data movement can proceed while tensor cores and CUDA cores perform computation. Fully exploiting these accelerators requires *software pipelining*—a technique that interleaves independent stages of computation and data movement across multiple iterations of tasks to maximize hardware utilization.

Existing systems implement *intra-task* pipelining, as shown in Figure 2a, where a single task is decomposed into multiple iterations. In this model, TMAs, tensor cores, and CUDA cores can simultaneously perform data transfer, matrix computation, and auxiliary operations for different iterations in a pipeline. However, kernel barriers restrict pipelining to within a single task, preventing *cross-task* pipelining and introducing pipeline bubbles that leave hardware resources underutilized.



(a) Kernel barriers prevent overlapping `MatMul` and `AllGather`.



(b) MPK enables fine-grained overlap of `MatMul` and `AllGather`.

Figure 3: Comparing how MPK and existing approaches support fine-grained kernel overlap between tasks. Data dependencies (black arrows in Figure 3b) ensure correctness.

Fine-grained kernel overlap. Kernel barriers also preclude opportunities to overlap kernels that utilize different hardware resources (e.g., compute and communication), as they enforce dependencies at the granularity of entire kernels rather than individual data units. Figure 3a illustrates a common pattern in large language models (LLMs), where a `MatMul` operator is followed by an `AllGather` operator. Existing systems generally launch these as two separate kernels, requiring all thread blocks of the `AllGather` kernel to wait until all thread blocks of the preceding `MatMul` kernel complete.

In practice, the data dependency between `MatMul` and `AllGather` is much finer-grained: since `AllGather` performs element-wise operations, each of its thread blocks only depends on the output of a single `MatMul` thread block. This dependency structure enables *fine-grained kernel overlap*, where different SMs can execute `MatMul` and `AllGather` in parallel, as long as fine-grained data dependencies are preserved. Such overlap allows the system to simultaneously utilize compute resources and communication bandwidth on modern GPUs, as identified in prior work [50]. Achieving this overlap, however, requires proper synchronization between SMs at sub-kernel granularity, as shown in Figure 3b, which is not supported by conventional kernel barriers.

2.2 Kernel Fusion

Kernel fusion eliminates kernel barriers by combining multiple GPU kernels that execute sequentially on the same data into a single, semantically equivalent kernel. Kernel fusion improves performance by avoiding materialization of intermediate results, reducing device memory accesses, and eliminating kernel launch overheads.

Kernel fusion has been widely adopted in tensor program compilers. Frameworks such as PyTorch, JAX, and TVM employ rule-based heuristics to fuse adjacent kernels [14, 15, 29], while systems such as Mirage and TASO automatically discover fusion rules through compiler super-

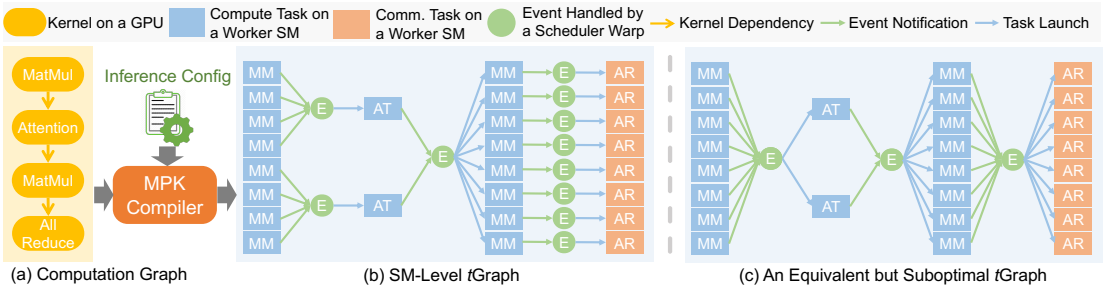


Figure 4: The MPK compiler transforms a kernel-level computation graph into an optimized SM-level *t*Graph. `MM`, `AT`, and `AR` denote `MatMul`, `Attention`, and `AllReduce` tasks, respectively.

optimization [26, 41]. However, existing compilers can only fuse small groups of local operators, as generating a single kernel that faithfully implements an entire complex tensor program is computationally difficult and often infeasible.

The *mega-kernel* paradigm pushes kernel fusion to the extreme by fusing all computation and communication of a tensor program into one persistent kernel, using device-memory synchronization primitives to coordinate execution across SMs. Despite its performance benefits, current ML compilers such as PyTorch, Triton, JAX, and TVM do not support mega-kernel compilation. Existing mega-kernels are instead handcrafted by GPU experts for specific models. For example, FlashDMoE fuses mixture-of-experts computation and inter-GPU communication into a single kernel [13], while Spector et al. manually designed and implemented a low-latency mega-kernel for LLAMA-1B [9, 37].

These manual approaches require substantial engineering effort and deep GPU expertise to mega-kernelize a tensor program. In contrast, MPK adopts a compiler-based approach that automatically transforms a tensor program into an optimized mega-kernel, eliminating the need for manual effort.

3 SM-Level Graph Representation

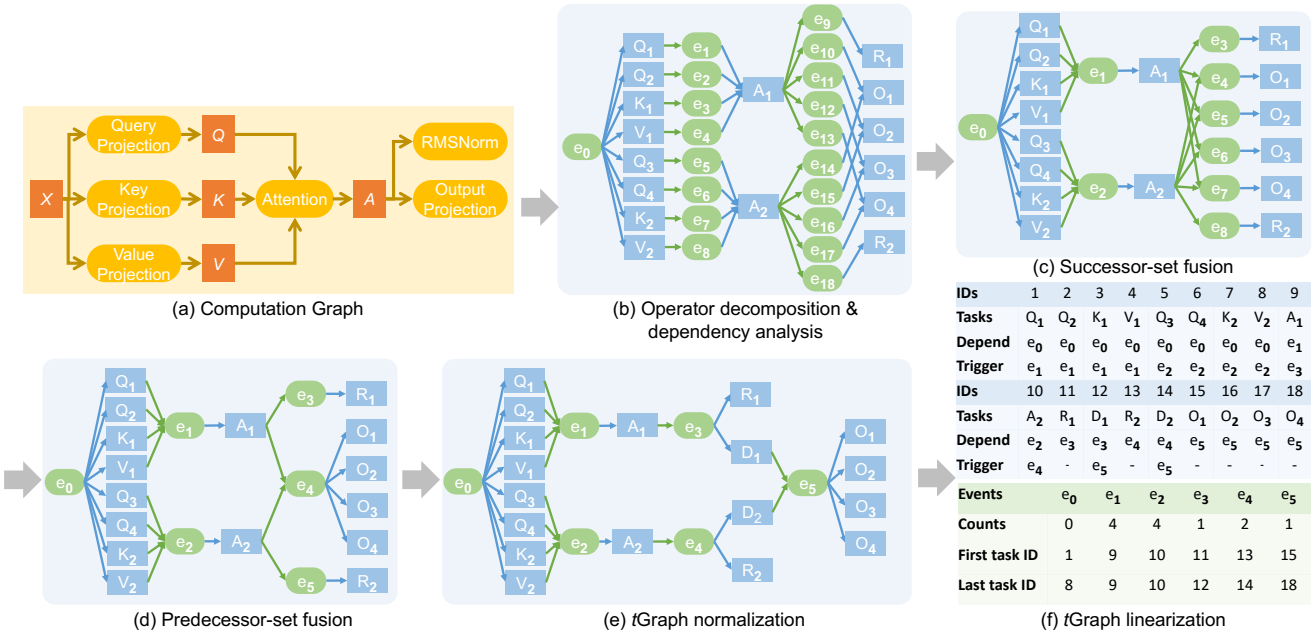
This section introduces *t*Graph, a representation that expresses the computation of a tensor program at the granularity of individual streaming multiprocessors (SMs). Unlike conventional computation graphs, which expose dependencies only between tensor operators, *t*Graph captures dependencies between SM-level units of work. This fine-grained representation exposes additional parallelism and enables optimizations such as cross-operator software pipelining and fine-grained kernel overlap, both of which are not supported by the existing kernel-per-operator execution model.

Figure 4 illustrates an example *t*Graph, where each node represents either a *task* or an *event*. Each task—shown as a blue (or orange) rectangle—denotes a unit of computation (or communication) executed on a single SM. Each event—shown as a green circle—represents synchronization across tasks. Tasks and events alternate in the graph: every task only has outgoing edges to *triggering events* and incoming edges from *dependent events*. A task is ready for execution when its dependent events are all *activated* and notifies its triggering event upon completion. An event is activated once it has received notifications from all tasks associated with it.

This structure captures dependencies at a much finer granularity than traditional computation graphs. For example, multi-GPU LLM serving often involves a `MatMul` operator followed by an `AllReduce` operator (Figure 4a). Existing systems generally execute these operators sequentially because coarse-grained kernel barriers synchronize entire kernels. In contrast, SM-level task graphs can represent precise task-level dependencies: since `AllReduce` performs element-wise communication and reduction, each of its tasks depends only on one corresponding `MatMul` task that produces its input tile. By inserting fine-grained events between dependent task pairs, MPK can overlap compute-intensive `MatMul` tasks with communication-intensive `AllReduce` tasks, improving overall GPU utilization.

Multiple *t*Graphs may represent the same computation graph. Figure 4c shows an alternative but suboptimal *t*Graph where events capture only operator-level dependencies, analogous to traditional kernel barriers. § 4 describes how MPK generates high-performance task graphs by inferring *precise* data dependencies to maximize concurrency and minimize synchronization overheads.

Comparison with CUDA Graphs. *t*Graphs can be viewed as a lower-level extension of CUDA Graphs, sharing several structural similarities. Like CUDA Graphs, *t*Graphs are statically instantiated and encode explicit dependencies among operations. However, while CUDA Graphs capture dependencies only at the kernel level, *t*Graphs operate at the granularity of individual SM tasks and sub-kernel events. CUDA Graphs primarily describe kernel launch order and rely on stream semantics for synchronization, which confines overlap and fusion to kernel boundaries. In contrast, *t*Graphs explicitly model both intra- and cross-operator dependencies, enabling fine-grained synchronization across SMs and overlap of computation and communication within a single kernel. This design allows MPK to exploit parallelism that is inaccessible to CUDA Graphs and other kernel-level execution models.



IDs	1	2	3	4	5	6	7	8	9
Tasks	Q_1	Q_2	K_1	V_1	Q_3	Q_4	K_2	V_2	A_1
Depend	e_0	e_0	e_0	e_0	e_0	e_0	e_0	e_0	e_1
Trigger	e_1	e_1	e_1	e_1	e_2	e_2	e_2	e_2	e_3
IDs	10	11	12	13	14	15	16	17	18
Tasks	A_2	R_1	D_1	R_2	D_2	O_1	O_2	O_3	O_4
Depend	e_2	e_3	e_3	e_4	e_4	e_5	e_5	e_5	e_5
Trigger	e_4	-	e_5	-	e_5	-	-	-	-

Events	e_0	e_1	e_2	e_3	e_4	e_5
Counts	0	4	4	1	2	1
First task ID	1	9	10	11	13	15
Last task ID	8	9	10	12	14	18

Figure 5: The MPK compiler workflow. In (b), Q, K, V, A, O, and R denote the sets of tasks produced by decomposing the query projection, key projection, value projection, attention, output projection, and RMSNorm, respectively. D_1 and D_2 in (e) are dummy tasks inserted during *t*Graph normalization to ensure that each task has a single triggering event. Finally, (f) shows how MPK linearizes the *t*Graph and stores the resulting structure, where tasks and events use a uniform, canonical representation.

4 The MPK Compiler

This section presents the MPK compiler, which takes a computation graph and an associated inference configuration as input and generates an optimized *t*Graph specialized for the target configuration and underlying GPU architecture. Figure 5 illustrates the end-to-end compilation workflow.

4.1 *t*Graph Generation

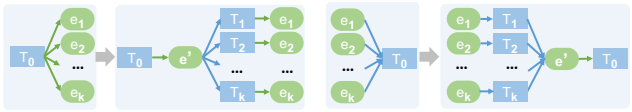
Operator decomposition. The MPK compiler decomposes each operator in the input computation graph into a set of tasks by *partitioning* the operator's output tensors. Each task computes a *disjoint* subset of the output, allowing tasks from the same operator to execute in parallel across SMs. Most tensor algebra operators can be partitioned along multiple output dimensions; for example, the output tensor of a matrix multiplication can be tiled along both the row and column dimensions to expose parallelism.

The performance of a partitioning strategy depends on both the problem shape and the target GPU architecture. To discover an effective strategy, MPK selects a partitioning strategy that minimizes data loading from device memory to shared memory, since device memory accesses are significantly more expensive than shared-memory accesses or computation on CUDA cores and tensor cores. By default, MPK generates a number of tasks proportional to the number of SMs to promote load balance across SMs during execution. MPK also provides an interface for users to specify custom partitioning strategies by setting the desired parallelization degree along each output dimension.

Dependency analysis. MPK uses *events* to capture dependencies between tasks. For any two operators sharing a tensor, MPK enumerates all pairs of tasks from the two operators and introduces an event e for a task pair (t_1, t_2) if and only if the output region produced by t_1 overlaps with the input region consumed by t_2. The event serves as a synchronization point indicating that t_2 cannot begin execution until t_1 has produced the required data. Accordingly, MPK inserts two edges, (t_1, e) and (e, t_2), into the resulting *t*Graph. This fine-grained dependency analysis preserves all producer-consumer dependencies while exposing parallelism across independent tasks.

Event fusion. MPK applies two complementary forms of event fusion—*successor-set fusion* and *predecessor-set fusion*—to eliminate redundant synchronization points and simplify the constructed *t*Graph. For an event e, we define two functions: $InTasks(e)$, the set of tasks that trigger e, and $OutTasks(e)$, the set of tasks that depend on e. These functions characterize when multiple events have identical dependency structure and can therefore be fused.

First, *successor-set fusion* merges events that serve as prerequisites for the same set of consumer tasks. Since these consumer tasks cannot begin execution until all such events are activated, representing the events separately provides no



(a) A transformation reducing fan-out of a task to one.

(b) A transformation reducing fan-in of a task to one.

Figure 6: MPK performs graph transformations to normalize an arbitrary *t*Graph, ensuring that every task has at most one dependent event and at most one triggering event.

additional scheduling flexibility.

Definition 4.1 (Successor-set fusion)**.** For any two events e_1 and e_2 in a *t*Graph, successor-set fusion applies if and only if $OutTasks(e_1) = OutTasks(e_2)$. MPK removes e_1 and e_2 from the *t*Graph and introduces a fused event e' with $InTasks(e') = InTasks(e_1) \cup InTasks(e_2)$ and $OutTasks(e') = OutTasks(e_1)$.

For example, successor-set fusion merges events e_{10} and e_{14} in Figure 5(b) into a new event, e_4 in Figure 5(c), because both events are prerequisites for task O_1.

Second, *predecessor-set fusion* merges events that depend on the same set of producer tasks. Since such events are triggered simultaneously, maintaining them as separate synchronization nodes introduces unnecessary graph complexity.

Definition 4.2 (Predecessor-set fusion)**.** For any two events e_1 and e_2 in a *t*Graph, predecessor-set fusion applies if and only if $InTasks(e_1) = InTasks(e_2)$. MPK removes e_1 and e_2 from the *t*Graph and introduces a fused event e' with $InTasks(e') = InTasks(e_1)$ and $OutTasks(e') = OutTasks(e_1) \cup OutTasks(e_2)$.

For example, predecessor-set fusion merges events e_4, e_5, e_6, and e_7 in Figure 5(c) into a single event, e_4 in the new *t*Graph, because all four events depend on tasks A_1 and A_2.

A core challenge MPK must address is representing dependencies between tasks and events efficiently. Because MPK executes tasks and updates events in parallel across SMs, the runtime requires a *uniform* and *cheap* representation that avoids costly indirect indexing. Two challenges arise. First, a task may depend on and trigger an arbitrary number of events. A straightforward approach to representing tasks is reserving space for the maximum number of dependent and triggering events per task. However, this approach leads to significant memory overhead. Second, after event fusion, an event may trigger an arbitrary number of tasks. Representing these outgoing edges by allocating space for the maximum fan-out per event is also expensive. MPK addresses these challenges using two techniques: *t*Graph normalization and *t*Graph linearization.

***t*Graph normalization.** Normalization bounds the dependency metadata stored for each task. When every task depends on and triggers at most one event, each task descriptor needs to store only one dependent-event identifier and one triggering-event identifier, rather than variable-length lists. MPK achieves this property through the two rewrites shown in Figure 6, which transform an input *t*Graph into a functionally equivalent form in which each task has at most one dependent event and at most one triggering event.

First, when a task T_0 triggers multiple events $e_1, \ldots, e_k$, MPK introduces a new event e' and k empty tasks $T_1, \ldots, T_k$, each of which performs no computation and depends on e'. After this transformation, T_0 triggers only e', and each newly introduced task T_i triggers exactly one of the original events e_i, as shown in Figure 6a. This transformation ensures that every task has at most one *triggering* event. Figure 5(e) shows how MPK applies this transformation to reduce the number of triggering events for A_1 and A_2 to one.

Second, when a task T_0 depends on multiple events $e_1, \ldots, e_k$, MPK introduces a new event e' and k empty tasks $T_1, \ldots, T_k$, each of which performs no computation and triggers e'. After this transformation, T_0 depends only on e', and each newly introduced task T_i depends on exactly one of the original events e_i, as shown in Figure 6b. This transformation ensures that every task has at most one *dependent* event.

*t*Graph normalization introduces additional tasks and events only when a *t*Graph contains tasks with multiple fan-in or fan-out events. This situation typically arises when the original computation graph contains operators that can execute in parallel. For example, tasks A_1 and A_2 in Figure 5(d) have two fan-out events because both the RMSNorm and output projection operators depend on attention and can therefore run in parallel. In practice, we observe negligible normalization overhead—always less than 1% in our evaluation—because real-world models are usually "deep" (with many sequential operators), rather than "wide" (with many parallel operators).

Algorithm 1 MPK's *t*Graph linearization algorithm. It is guaranteed that each task is enqueued into T once and that each event is enqueued into E once. Lines 5-7 ensure that all tasks depending on an event are consecutive in T.

Input: A normalized *t*Graph $\mathcal{G}$
Output: A list of tasks T such that for each event $e \in \mathcal{G}$, the tasks launched by e are consecutive in T.

```
1:  T ← ∅
2:  E ← {e ∈ G | e.counts = 0} ▷ Enqueue all events with no dependent tasks
3:  while E is not empty do
4:    e ← E.dequeue()
5:    for all task t ∈ G do
6:      if t.dependent_event = e then
7:        T.enqueue(t)
8:        e' ← t.trigger_event
9:        if all tasks triggering e' are in T then
10:         E.enqueue(e')
11:
12: return T
```

***t*Graph linearization.** Linearization complements normalization: while normalization bounds the number of events

associated with each task, linearization bounds the storage required to record the tasks associated with each event. *t*Graph normalization alone does not address the second representation challenge: after event fusion, an event may still need to trigger many tasks. For example, event e_5 in Figure 5(e) triggers four tasks, which would otherwise require explicit storage for all outgoing task indices.

MPK addresses this challenge using a breadth-first-search-based algorithm, shown in Algorithm 1, to linearize a *t*Graph. The algorithm assigns contiguous indices to all tasks triggered by the same event in the final task ordering. As a result, the fan-out of an event can be encoded compactly using only the first and last task indices, eliminating the need to store an explicit list of dependent tasks while preserving all dependencies.

Figure 5(f) illustrates how MPK stores the linearized *t*Graph in GPU device memory. For each task, MPK records only the indices of its dependent and triggering events. For each event, MPK stores the number of triggers required for activation; once activated, the runtime launches all tasks whose indices fall within the event's first and last task indices.

4.2 Task Implementation Generation

In addition to constructing a *t*Graph, MPK must generate a device function for each task to execute on a GPU SM. MPK leverages prior work and uses a *compiler superoptimization* approach to automatically generate a high-performance implementation for each task.

Specifically, MPK performs superoptimization at the thread block level instead of the kernel level. Each compute task is associated with a reference PyTorch implementation, and MPK uses the Mirage superoptimizer [41] to search for an optimized thread-block graph, which is sent to the Mirage compiler to generate a CUDA implementation. The CUDA implementation includes intra-SM optimizations such as software pipelining, register reuse, and layout optimizations to reduce shared-memory bank conflicts.

5 In-Kernel Parallel Runtime

MPK employs an *in-kernel parallel runtime* that executes the *t*Graph across all SMs within a single mega-kernel. This design eliminates repeated kernel launches and exposes fine-grained control over scheduling, synchronization, and execution order. Once launched, the mega-kernel continuously manages both computation and communication until the inference workload completes.

To support this execution model, MPK partitions a GPU's SMs into *workers* and *schedulers*. Each worker runs on one physical SM and maintains an independent *task queue*. Workers execute a lightweight loop that repeatedly dequeues tasks, performs the associated computation or communication, and signals task completion by notifying the task's triggering

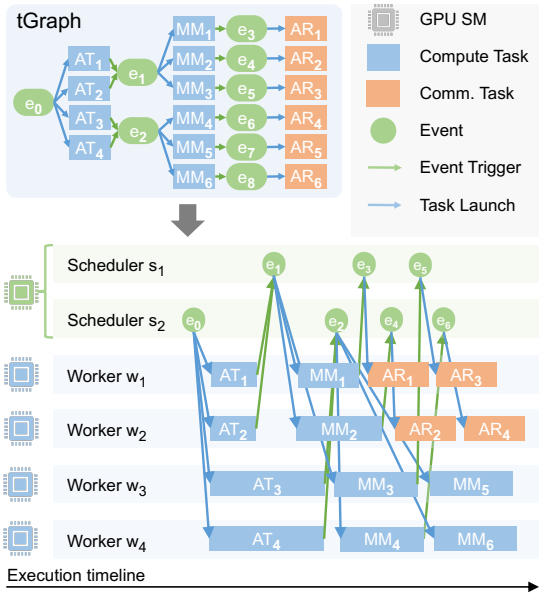


Figure 7: The MPK event-driven execution model. Circles denote events, and blue (or orange) rectangles denote compute (or communication) tasks, respectively. Edges from an event to a task correspond to task launches, while edges from a task to an event indicate that the task triggers the associated event upon completion. `AT`, `MM`, and `AR` refer to attention, matrix multiplication, and AllReduce, respectively.

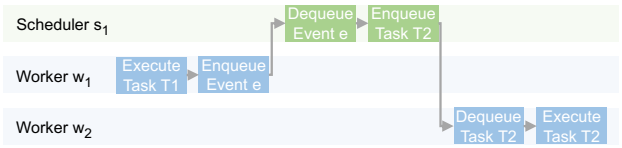
event. This design ensures that workers are fully utilized while enabling asynchronous execution across operators.

Schedulers are organized at *warp* granularity, with each SM hosting four scheduler warps. Each scheduler maintains an event queue and repeatedly polls for newly activated events, dispatching the corresponding tasks to workers. The allocation of workers and schedulers is fixed at kernel-launch time and matches the GPU's physical SM count, avoiding any dynamic role-switching overhead inside the kernel.

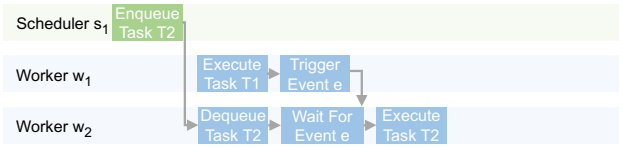
The remainder of this section details the in-kernel runtime architecture. § 5.1 describes MPK's event-driven execution model. § 5.2 introduces two complementary task-launch mechanisms, analyzes their trade-offs, and explains how MPK combines them to achieve low-latency and load-balanced execution. § 5.3 describes additional runtime optimizations that further reduce overhead and improve throughput.

5.1 Event-Driven Execution

MPK executes a *t*Graph using an *event-driven* model. Each *t*Graph begins with a designated *start* event (e.g., e_0 in Figure 7), which has no prerequisites. This event is initially enqueued into a scheduler's event queue. Upon dequeuing the event, the scheduler (e.g., s_1) launches all tasks that depend on it (e.g., $AT_1, \ldots, AT_4$). Each launched task is dispatched to a worker, which executes the task and, upon completion, notifies the triggering event associated with that task.



(a) Just-in-time task launch.



(b) Ahead-of-time task launch.

Figure 8: Comparing JIT and AOT task launches.

An event becomes *activated* once all of its prerequisites have completed and thus have collectively triggered the event the required number of times. When an event is activated, it is enqueued into a scheduler's event queue, allowing the runtime to continue propagating execution through the *t*Graph. In this way, events serve as the mechanism for driving task execution, enabling fine-grained, asynchronous scheduling.

This event-driven model also allows MPK to adapt to the underlying interconnect without requiring an explicit topology profile. Communication and computation are represented uniformly as tasks in the same *t*Graph and dispatched by the same scheduler. A dependent compute task launches as soon as the communication tasks producing its inputs trigger the relevant event, regardless of the latency of the links they traverse. A faster link triggers its event sooner and releases dependent work earlier, while a slower link delays only the tasks that depend on it. The scheduler therefore does not require explicit knowledge of link bandwidths or topology distances. The same *t*Graph and scheduler can thus handle heterogeneous interconnects, such as intra-node NVLink and inter-node networks, by reacting to data availability rather than relying on a static topology model.

5.2 Hybrid Task Launch

A task can be enqueued into a worker's task queue either *just-in-time* (JIT) or *ahead-of-time* (AOT). In JIT mode, a scheduler assigns a task to a worker only after its dependent event has been fully activated; the task can therefore begin execution immediately after assignment. In AOT mode, the runtime pre-enqueues the task into a worker's task queue before its predecessor event is activated. The worker cannot execute the task until the dependent event is fully activated, and thus waits locally for the dependency to be satisfied.

JIT and AOT approaches provide complementary advantages. On the one hand, JIT launch allows MPK to adapt to workload imbalance. For example, attention in LLMs involves highly variable execution times due to data-dependent sequence lengths—requests with long sequence lengths take longer to finish than those with short ones. This variance makes static assignment ineffective. Under JIT launch, MPK launches downstream tasks (e.g., `MatMul` or `AllReduce`) only after the attention tasks that trigger them have completed. Workers that finish their attention tasks earlier can execute more downstream tasks, improving end-to-end latency and balancing load across SMs, as illustrated in Figure 7.

On the other hand, JIT launch involves higher latency due to additional worker–scheduler communication. Figure 8 illustrates the difference when launching task T_2 after event e is triggered by task T_1. Under JIT launch (Figure 8a), the worker executing T_1 notifies a scheduler, which then dequeues event e, launches T_2, and enqueues it into a worker's task queue. The receiving worker must then dequeue T_2 before execution. This chain requires two synchronization steps (worker→scheduler and scheduler→worker). By contrast, under AOT launch (Figure 8b), T_2 has already been enqueued on a pre-assigned worker, which only needs to wait for event e to be activated. Thus, AOT launch requires only one synchronization step through the event trigger, reducing per-task launch latency.

MPK uses *hybrid task launch* to combine the advantages of both approaches. During *t*Graph construction, the compiler classifies each operator as JIT or AOT based on whether its execution time is data-dependent and likely to induce runtime imbalance. Operators with data-dependent durations (e.g., attention) are marked as JIT, and their downstream operators remain JIT until execution reaches a global barrier (i.e., an event that must be triggered by all upstream tasks). Such barriers eliminate accumulated imbalance, making subsequent operators suitable for AOT launch. All remaining operators are labeled AOT to minimize dispatch overhead. Labels apply at operator granularity: all tasks generated by the same operator share the same launch mode.

Workers maintain two queues, one for JIT tasks and one for AOT tasks. Workers always prioritize JIT tasks, as they are ready to execute immediately. When a worker exhausts its JIT queue, it checks whether the first AOT task's dependent event has been fully activated; if so, the worker executes that AOT task. This design ensures that a worker blocks only when no ready work is available.

Schedulers handle only events that launch JIT tasks. All AOT tasks are *pre-enqueued* before execution begins. Because operators typically produce a number of tasks proportional to the number of workers (§ 4), MPK distributes AOT tasks across workers in a round-robin fashion to maintain balanced load. Pre-enqueuing AOT tasks reduces scheduler load and amortizes dispatch overhead, while JIT launch dynamically balances work across SMs.

5.3 Runtime Optimizations

This subsection introduces runtime optimizations that minimize MPK's execution overhead.

Paged shared-memory abstraction. In conventional GPU programming models, shared memory is a fast on-chip memory private to each thread block. Shared memory exists only for the lifetime of a kernel: once the kernel finishes, its shared memory is automatically released. In the kernel-per-operator execution model, each kernel typically assumes *exclusive* access to the shared memory allocated to its thread block. However, this assumption prevents cross-task software pipelining (§ 2), which overlaps data loading for a subsequent task with computation in the current task, since both tasks may need shared memory at the same time.

To enable such pipelining, MPK introduces a *paged shared-memory* abstraction. Shared memory is partitioned into fixed-size pages, and task implementations are modified to operate on pages instead of assuming a monolithic allocation. A task may *acquire* one or more pages based on its shared-memory footprint and must *release* the pages when they are no longer needed. Once a task releases any page, it is no longer permitted to acquire additional shared-memory pages, enforcing a monotonic usage pattern that simplifies scheduling. When the current task releases pages, MPK can immediately allocate available pages for the next task and begin data prefetching. This design enables fine-grained, on-demand allocation of shared memory within the mega-kernel execution model.

Cross-task software pipelining. To enable software pipelining across tasks executed on the same worker (§ 2.1), MPK decomposes each task into a *pre-loading* phase and a *compute* phase. The pre-loading phase issues data transfer instructions to fetch a chunk of the required tensor from device memory into shared memory, initializing the intra-task software pipeline without performing computation.

MPK opportunistically overlaps the compute phase of the current task T_1 with the pre-loading phase of the subsequent task T_2 when two conditions hold: (1) T_1 has already issued all of its own data-transfer instructions, and (2) sufficient shared-memory pages are available for T_2's pre-loading phase. This pipeline does not interfere with T_1's execution because MPK inserts the necessary intra-SM synchronization barriers to ensure that T_2's memory transfers do not conflict with ongoing data transfers for T_1.

Pre-fetching task descriptions. Each worker maintains both JIT and AOT task queues in device memory. Every task is associated with a task description that encodes its input tensors, output tensors, and configuration parameters; in our current implementation, each description occupies 352 bytes (§ 6.1). To reduce enqueue/dequeue latency and hide device-memory access costs, MPK employs a lightweight prefetching mechanism that retrieves upcoming task descriptions into shared memory before they are needed.

6 Evaluation

This section evaluates MPK by answering four questions. First, § 6.3 compares MPK's mega-kernel execution model with state-of-the-art kernel-per-operator systems. Second, § 6.4 studies how MPK supports dynamic workloads. Third, § 6.5 evaluates the scalability and efficiency of MPK for multi-GPU execution. Finally, § 6.6 isolates the performance contributions of the main runtime optimizations in MPK.

We focus our evaluation on LLM serving for two reasons. First, LLM serving has several heavily optimized kernel-per-operator baselines, including SGLang and vLLM [27, 46]; comparing against them provides a stringent benchmark that highlights the benefits of MPK's mega-kernel approach. Second, LLM serving naturally exhibits dynamic execution behavior, as different serving iterations may vary in batch size, sequence length, and the mixture of prefill and decode work. This variability creates heterogeneous workloads that stress both the compiler and the runtime. Although our evaluation focuses on LLM serving, the MPK compiler and runtime are model-agnostic and can support general DNN architectures.

6.1 Implementation Details

We implement MPK as a PyTorch compiler backend. A PyTorch program can be compiled into an MPK mega-kernel via PyTorch's compilation interface by specifying MPK as the backend, i.e., `torch.compile(backend=MPK)`. This call invokes the MPK compiler, which generates a mega-kernel and returns it as a callable PyTorch function. Invoking this function issues a single launch of the generated mega-kernel.

The current MPK implementation consists of approximately 44K lines of C++, 42K lines of CUDA, and 10K lines of Python. The in-kernel parallel runtime is written in CUDA and uses semaphores in device memory to coordinate workers and schedulers. The MPK compiler, implemented in C++ and Python, automatically transforms an input tensor program into an optimized *t*Graph tailored to specific GPU types. For compute tasks, the compiler integrates the Mirage superoptimizer [41] to automatically generate optimized CUDA implementations and uses NVSHMEM [10] to support in-kernel inter-GPU communication.

Our implementation includes several optimizations to minimize runtime overhead and support dynamic workloads.

Task-launch overhead. Because MPK decomposes computation into tasks that are substantially finer-grained than traditional GPU kernels, minimizing per-task launch overhead is essential for performance. MPK uses several techniques to keep task-launch costs low. First, the runtime uses lightweight workers and schedulers: event queues and task queues are implemented as circular buffers in GPU device memory, and enqueue and dequeue operations rely only on

Table 1: MPK configuration in our evaluation.

GPU	# SMs	# Workers	# Schedulers
A100	108	104	16
H100	132	128	16
B200	148	144	16

low-cost `atomicAdd` instructions. Second, MPK uses decentralized scheduling, in which each scheduler assigns tasks using local state. This design avoids global coordination and eliminates the communication and synchronization overheads inherent to globally coordinated scheduling.

Although the current implementation uses decentralized scheduling, the runtime is designed to support alternative policies, including globally coordinated scheduling, with minor code changes. Exploring these policies and their performance trade-offs is an interesting direction for future work.

Supporting runtime dynamism. To demonstrate MPK's ability to support highly dynamic workloads, we extend the system with mechanisms required for LLM serving, including continuous batching [44] and paged attention [27]. When processing the start event of a *t*Graph, the scheduler prepares a new decoding iteration by (1) removing completed requests from the previous iteration, (2) admitting newly arrived requests, and (3) updating per-request KV-cache metadata. This logic executes as a single task, and the KV-cache metadata is stored in device memory for direct access by attention tasks.

To handle the dynamic batch sizes intrinsic to LLM serving, MPK generates multiple *t*Graphs specialized for representative batch sizes, using powers of two up to the maximum batch size. At runtime, the scheduler selects the appropriate *t*Graph based on the current batch size. This approach allows the compiler to generate *t*Graphs optimized for specific batch sizes while preserving flexibility for dynamic serving workloads.

6.2 Experimental Setup

We evaluate MPK on five widely deployed LLMs that span different parameter scales and architectural families, and on three generations of NVIDIA GPUs: A100, H100, and B200. Table 1 summarizes the MPK configuration for each GPU. In all experiments, MPK reserves four SMs for schedulers, allocating 16 scheduler warps in total because each SM can host up to four active scheduler warps. The remaining SMs are used as workers. We set the shared-memory page size to 32 KB on all GPUs, which yields 5, 7, and 7 shared-memory pages per SM on A100, H100, and B200, respectively. Figure 9 shows the evaluated LLMs, which include both dense and mixture-of-experts models across multiple model sizes.

To control for variability in request-arrival patterns, all experiments are conducted in an offline batched-inference setting while varying the maximum batch size. Each request uses a prompt length of 64 tokens and generates 1024 output tokens. This methodology eliminates server-side stalls caused by insufficient request concurrency and enables a controlled comparison of system-level performance.

6.3 End-to-end Results

We first compare the end-to-end serving performance of MPK with SGLang and vLLM, two state-of-the-art LLM serving systems. Both SGLang and vLLM use the kernel-per-operator approach and rely on specialized kernel libraries, including FlashInfer [43] and FlashAttention [7] for attention, cuBLAS and cuTLASS [2] for matrix multiplication, and CUDA or Triton [36] for other operators. All systems load model architectures from HuggingFace Transformers [6], use bfloat16 precision, and enable paged attention [27] and continuous batching [44]. The key architectural difference is that MPK integrates page allocation and request scheduling directly into the mega-kernel. In contrast, SGLang and vLLM perform these operations on the CPU, incurring additional host–device synchronization and dispatch overheads.

For each model, we evaluate all three systems on B200, H100, and A100 GPUs with maximum batch sizes from 1 to 16, and report serving throughput.

Figure 9 shows the end-to-end throughput results. For single-batch inference, MPK improves serving performance by 1.0–1.7× across models and hardware. The improvements are most significant for smaller models and newer GPU generations. This trend is expected because three overheads become more significant as the amount of computation per token decreases and GPU hardware becomes faster: (1) kernel-per-operator approaches incur kernel-switch overheads, even when using CUDA Graphs; (2) kernel boundaries introduce pipeline bubbles and prevent cross-task pipelining (Figure 2); and (3) SGLang and vLLM perform page allocation and request scheduling on the CPU, adding CPU–GPU synchronization delays. § 6.6 quantifies the impact of these optimizations.

These results show that MPK is well-suited for latency-sensitive serving scenarios, such as single-batch decoding, where reducing per-token latency is critical. For example, on Qwen3-8B running on an A100 GPU, MPK reduces per-token decoding latency from 14.5 ms, achieved by highly optimized systems such as vLLM and SGLang, to 12.5 ms. This approaches the approximate hardware lower bound of 10 ms, estimated by loading 16 GB of model parameters at 1.6 TB/s memory bandwidth.

Beyond performance, MPK also improves programmability. vLLM and SGLang require substantial engineering effort to optimize new models and integrate specialized kernels. In contrast, MPK takes a compiler-based approach that automatically mega-kernelizes a PyTorch model with only a few lines of code changes. As a result, MPK combines high performance with a familiar PyTorch development workflow, achieving more than 10× speedup over native PyTorch.

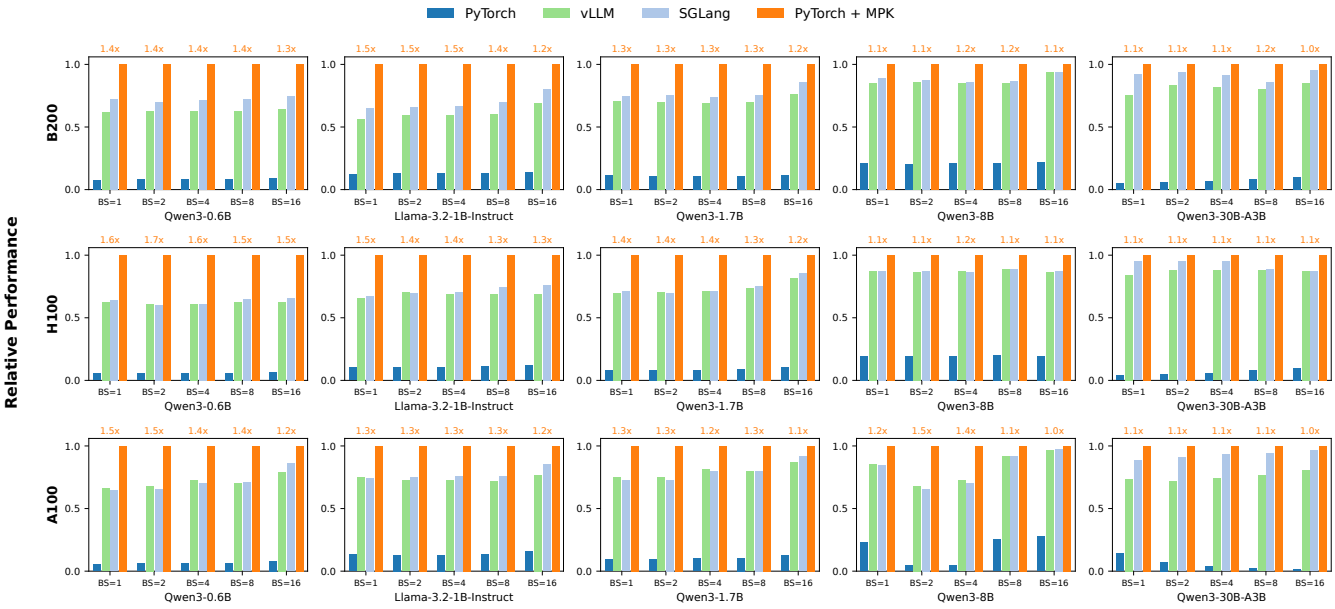


Figure 9: End-to-end comparison of MPK with existing systems across five models on A100, H100, and B200 GPUs. All results are normalized to MPK; higher is better. The value above each MPK bar reports its speedup over the best-performing baseline.

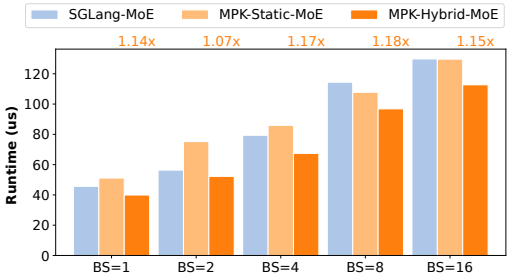


Figure 10: Comparing MPK with existing systems for Qwen3-30B-A3B on B200. Each value represents the actual MoE runtime in microseconds for each approach (lower is better), and the numbers above the bars indicate the speedup achieved by MPK-Hybrid-MoE over SGLang-MoE.

6.4 Case Study: Mixture-of-Experts

To efficiently serve dynamic workloads such as mixture-of-experts (MoE) models, MPK implements two MoE-specific optimizations: a hybrid workload balancer and a fused gather–GEMM implementation.

Representing expert parallelism. MPK represents expert parallelism using the same task abstraction used for other operators. An MoE block is lowered into a chain of SM-level tasks in the *t*Graph (§ 4), including routing, dispatch, expert computation, and combine. The dispatch and combine stages capture the all-to-all communication required by expert parallelism, and MPK lowers them into inter-GPU data-transfer tasks in the same way it lowers other collective operations (§ 6.5). Because communication tasks and expert-computation tasks reside in the same *t*Graph and are dispatched by the same event-driven scheduler, the runtime overlaps all-to-all communication with computation as soon as the relevant events are triggered, without relying on CUDA streams or host-side synchronization. The two MoE-specific optimizations described below operate within this task representation.

Hybrid workload balancer. Because the number of tokens routed to each expert is known only at runtime, choosing an effective workload partition statically is challenging. A naive static strategy assigns a fixed group of SMs to preassigned experts. However, under skewed routing distributions, this strategy can lead to severe load imbalance: some SM groups become oversubscribed while others remain underutilized. At the other extreme, a fully dynamic strategy based on persistent Grouped-GEMM [8] can balance work across SMs, but introduces fine-grained synchronization overheads.

MPK therefore uses a *hybrid* strategy that combines static structure with runtime adaptivity. At compile time, the compiler partitions the MoE computation into expert-specific tasks. At runtime, each task receives a meta-tensor produced by `topk-softmax` containing global MoE information, including the number of activated experts and tokens assigned to each expert. Using this metadata, tasks refine their workload allocation dynamically and split work more evenly while avoiding the overheads of fully dynamic scheduling. As shown in Figure 10, this hybrid strategy consistently outperforms purely static partitioning across all batch sizes.

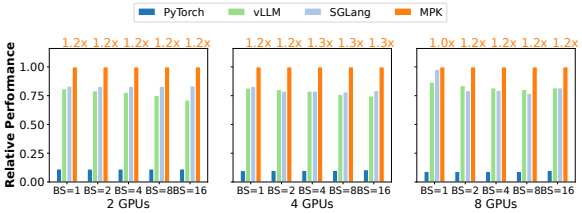


Figure 11: Multi-GPU comparison of MPK with existing systems for Qwen3-1.7B on H100 GPUs using tensor parallelism. All results are normalized to MPK; higher is better.

Fused gather-GEMM. To use tensor memory accelerators (TMAs) on Hopper and Blackwell GPUs, conventional MoE implementations first gather tokens routed to the same expert into a contiguous memory layout. For Qwen3-30B-A3B at batch size one on SGLang, this preprocessing step accounts for up to 11% of total MoE execution time. In MPK, implementing this step as a separate preprocessing task would also introduce additional scheduling overhead.

MPK addresses this issue by replacing the TMA-based gather with an asynchronous token-level copy integrated directly into the data-loading phase of the GEMM tasks. This fusion eliminates the standalone gather kernel and avoids additional scheduling points while preserving efficient memory movement. As a result, MPK with fused gather-GEMM achieves consistent speedups over SGLang's implementation.

6.5 Multi-GPU Results

We evaluate the scalability of MPK across multiple GPUs on an NVIDIA H100 DGX instance. As in the baseline systems, we use tensor model parallelism, following Megatron-LM [31]. Users specify tensor-parallel execution by inserting `AllReduce` after attention and gated MLP blocks. MPK then automatically compiles these collective operators into two types of tasks: inter-GPU data-transfer tasks, implemented using NVSHMEM's `nvshmem_signal_wait_until`, and local reduction tasks. This decomposition converts synchronous collective communication into asynchronous tasks that can be integrated into MPK's task-based, event-driven runtime.

Figure 11 shows the multi-GPU results. Compared with PyTorch, which uses a combination of hand-optimized kernels, CUDA Graphs, and `torch.compile`, MPK's mega-kernel execution improves throughput by up to 10×. Compared with highly optimized serving systems such as SGLang and vLLM, MPK achieves 1.1–1.4× speedups when scaling to 8 H100 GPUs. These gains come from three optimizations missing in kernel-per-operator baselines: (1) MPK integrates page allocation and request scheduling directly into the mega-kernel, eliminating CPU-side dispatch overheads; (2) MPK's asynchronous execution model overlaps compute tasks with collective communication; and (3) MPK eliminates kernel barriers and enables cross-task software pipelining. § 6.6 analyzes the latter two optimizations in detail.

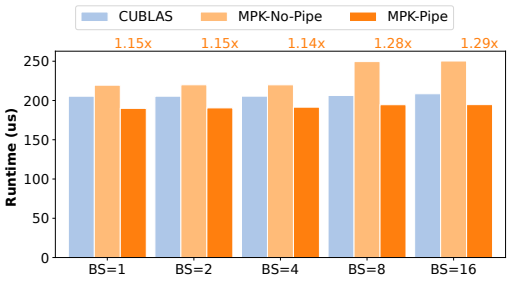


Figure 12: Ablation study of cross-task pipelining. We measure the runtime of the final linear layer in Qwen3-8B on an NVIDIA B200 GPU and report execution time in microseconds; lower is better. The value above each bar reports the speedup of MPK-Pipe over MPK-No-Pipe.

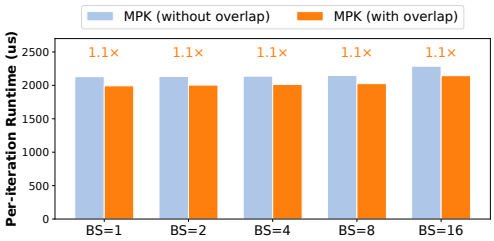


Figure 13: Ablation study on compute–communication overlap. We measure the runtime of Qwen3-1.7B on four H100 GPUs using tensor parallelism; lower is better. We compare MPK with compute–communication overlap enabled and disabled to quantify the benefit of fine-grained overlap.

6.6 Ablation Study

This section evaluates the impact of three runtime optimizations enabled by MPK: cross-task pipelining, compute–communication overlap, and kernel-launch reduction.

Cross-task pipelining. As described in § 5.3, MPK enables cross-task pipelining by preloading chunks of input tensors for the next task while the current task is executing. Figure 12 evaluates the impact of this optimization on the final linear layer in Qwen3-8B. Cross-task pipelining reduces task runtime by 1.2–1.3× and even outperforms cuBLAS-based compiled kernels.

Compute-communication overlap. MPK captures fine-grained dependencies between tasks (Figure 5), allowing the runtime to opportunistically overlap compute and communication. Figure 13 evaluates the impact of this optimization. To disable overlap, we restrict the *t*Graph to capture only coarse-grained dependencies between each collective operator, such as `AllReduce`, and its preceding computation using a single event, as illustrated in Figure 5c. Enabling fine-grained compute–communication overlap reduces per-iteration latency by 1.1×.

Kernel-launch reduction. MPK executes the entire model with a single kernel launch, whereas a kernel-per-operator execution of Qwen3-8B issues 293 kernel launches per token. On B200, each launch costs 3.8 μs in eager execution, totaling 1.1 ms per token, and 0.8 μs with CUDA Graphs, totaling 0.2 ms per token. MPK avoids this overhead; its in-kernel scheduler accounts for only 0.28% of total runtime (§ 6.1).

6.7 Compiler-Stage Ablation

The preceding ablations isolate two runtime mechanisms. We now isolate the contribution of each compiler stage described in § 4. Because these stages serve different purposes, we report the most appropriate metric for each stage rather than using a single common metric. Table 2 summarizes the results on three representative models.

Operator decomposition. Operator decomposition partitions each operator into independent SM-level tasks. As shown in Table 2, decomposition exposes substantial parallelism in real models: each operator is split into 32–47 tasks on average. For example, in Qwen3-8B, 293 operators are decomposed into 13,867 tasks. Compute-intensive operators, including linear layers, attention, and MoE experts, each expose tens of parallel tasks, while only a small number of pointwise operators, such as normalization at batch size one, map to a single task. This decomposition provides the runtime with sufficient independent work to keep all SMs busy.

Event fusion. Without fusion, dependency analysis emits a separate event for every overlapping producer–consumer task pair. Successor-set and predecessor-set fusion collapse events that share the same consumer set or producer set into a single synchronization point. This optimization is highly effective: across the evaluated models, the final *t*Graphs contain only 1,142–2,366 events, yet these events encode 69,000–162,000 producer–consumer task-pair dependencies. This corresponds to a 37–118× reduction in synchronization events (Table 2). The reduction is largest for the MoE model because expert routing creates many-to-many dependencies that are especially amenable to fusion.

***t*Graph normalization.** *t*Graph normalization inserts auxiliary tasks and events only when a task triggers or depends on more than one event, namely at forks and joins. The compiled LLM forward *t*Graphs in our evaluation are almost entirely sequential: we observe no fork/join groups in all three compiled *t*Graphs because operators that would otherwise fan out, such as the query/key/value projections, are emitted as fused operators. Normalization therefore leaves these *t*Graphs essentially unchanged, confirming the "deep, not wide" structure discussed in § 4. Nevertheless, normalization remains necessary for correctness when parallel branches do occur, such as in the unfused Q/K/V example in Figure 5. In those cases, normalization bounds each task's event fan-in and fan-out to one, preserving the fixed-size, indirection-free encoding.

Table 2: Per-compiler-stage statistics on B200. **Ops**: number of operators; **Tasks/op**: average number of tasks per operator after decomposition; **Events**: number of events in the final *t*Graph; **Fusion**: event-count reduction from event fusion; **Lin.**: device-memory footprint reduction from linearization.

Model	Ops	Tasks/op	Events	Fusion	Lin.
Qwen3-1.7B	229	35.6	1,870	37×	4.4×
Qwen3-8B	293	47.3	2,366	68×	5.9×
Qwen3-30B-A3B	533	32.2	1,142	118×	15.0×

***t*Graph linearization.** *t*Graph linearization encodes each event's successor tasks as a contiguous index range rather than an explicit task list. We measure the resulting on-device footprint as the size of the successor encoding with and without linearization. The contiguous-range encoding reduces this footprint by 4.4–5.9× on dense models, such as from 110,932 bytes to 18,928 bytes for Qwen3-8B, and by 15.0× on the MoE model. The MoE model benefits most because high-fan-out expert-routing events are particularly compact under the range-based encoding. Linearization achieves these reductions without losing dependency information.

7 Discussion

Resource footprint. A common concern with mega-kernels is that a single kernel must reserve, on every SM, the combined resources required by all fused operators. MPK avoids this issue: tasks are time-multiplexed across SMs rather than statically partitioned, so each SM uses only the resources required by the task it is currently executing (§ 5). Shared memory follows the same pattern: the paged abstraction (§ 5.3) acquires and releases pages at task granularity. The main exception is per-thread register usage, which is fixed for the mega-kernel at the maximum required across task types.

Integrating hand-tuned kernels and porting to new hardware. Each MPK task is a CUDA device function with a uniform calling convention defined by the *t*Graph. This interface gives MPK two forms of flexibility. First, a hand-tuned kernel can be incorporated into MPK by wrapping it as the device function for a task; the task then executes under the same event-driven schedule as compiler-generated tasks, without requiring changes to the runtime or the *t*Graph. MPK can thus reuse hand-optimized implementations when available while automatically generating the remaining tasks. Second, porting MPK to a new GPU architecture only requires updating the per-task code generators that emit these device

functions; the *t*Graph representation, the compiler's graph-level transformations, the runtime, and the scheduler remain unchanged. Therefore, per-task code generation is the only hardware-specific component of MPK (§ 4), while the rest of the system is shared across GPU architectures.

8 Related Work

Manually designed kernels. Existing ML frameworks such as TensorFlow XLA [1, 12], PyTorch [29], and TensorRT [34] adopt a kernel-per-operator approach and rely on GPU experts to manually design and implement kernels for individual operators. For attention alone, various specialized kernels have been developed, including FlashAttention [5, 18, 23], FasterTransformer [3], and FlashInfer [43], each targeting specific architectural features or deployment scenarios. Current systems rely on many specialized kernel libraries, making it hard to unify the entire inference pipeline into one mega-kernel.

ML compilers. A large body of work has explored *compiler-based* generation of high-performance kernels for tensor programs. Systems such as TVM [15, 16], Ansor [45], and Triton [36], alongside others [20, 22, 25, 49], build on the algorithm–schedule separation introduced by Halide [28, 30]. Another line of work employs *superoptimization* techniques to automatically search for efficient kernel implementations from high-level specifications [26, 38, 39, 42, 47]. However, these compilers are largely designed around operator- or graph-level optimization and do not support generating a unified mega-kernel or coordinating cross-operator execution.

Mega-kernels. Prior efforts on mega-kernels largely rely on manual design. For example, FlashDMoE fuses mixture-of-experts computation with inter-GPU communication into a single handcrafted mega-kernel [13]. Spector et al. manually developed a low-latency mega-kernel for LLaMA-1B [9, 37]. These approaches require extensive engineering effort and deep GPU expertise, and they do not generalize across models or GPUs. In contrast, MPK provides a compiler-based solution that automatically transforms a tensor program into a highly optimized mega-kernel, eliminating the need for manual fusion or hand-written mega-kernel implementations.

A related line of work simplifies the development of manually fused kernels. ThunderKittens [32] provides tile-based abstractions for high-performance kernels, and PipeThreader [17] exposes software-pipelined kernel execution to the programmer. These tools still require the developer to decide which operators to fuse, how to pipeline them, and how to coordinate their execution. MPK instead compiles an entire tensor program into SM-level tasks and executes them as one mega-kernel. These abstractions are therefore complementary to MPK: they could be used to implement individual task kernels within the MPK runtime.

TileRT [35] similarly decomposes operators into fine-grained tile-level tasks executed as a single persistent kernel, validating our core idea that persistent mega-kernel execution is key to low-latency inference.

Compute–communication overlap. A large body of work overlaps collective communication with computation to hide inter-GPU communication latency. DeepEP [19] provides expert-parallel all-to-all communication kernels for MoE models, and Triton-Distributed [48] extends the Triton compiler to express overlapping distributed kernels. TokenWeave [21], FlashOverlap [24], and ParallelKittens [33] fuse or co-schedule communication with dependent computation to reduce exposed communication latency. Earlier work from Google decomposes collective operators so that they overlap with dependent computation [40]. These systems realize overlap at the granularity of individual kernels or collective operators. In contrast, MPK represents communication and computation uniformly as tasks in one *t*Graph and schedules them using the same in-kernel runtime (§ 6.5). Compute–communication overlap therefore emerges from the global task schedule and applies uniformly to both tensor-parallel collectives and expert-parallel all-to-all communication.

9 Conclusion

This paper presents MPK, the first compiler and runtime system that automatically transforms multi-GPU model inference into a fully fused mega-kernel. By introducing SM-level task graphs and an in-kernel parallel runtime, MPK overcomes key limitations of the kernel-per-operator execution model, enabling inter-operator software pipelining, fine-grained overlap of computation and communication, and the elimination of kernel-launch and CPU-side scheduling overheads. Our evaluation shows that MPK brings LLM serving latency close to hardware limits and significantly improves throughput across models and GPU generations. By unifying execution within a single mega-kernel while preserving compatibility with existing ML frameworks, MPK opens a new path toward high-performance, compiler-driven inference systems.

Acknowledgment

We thank the anonymous OSDI reviewers and our shepherd for their valuable comments and suggestions. This work was partially supported by NSF awards CNS-2211882 and CNS-2239351, a Sloan Research Fellowship, and research awards from Amazon, Cisco, Google, Jane Street, Meta, NVIDIA, Oracle, Qualcomm, and Samsung. We also gratefully acknowledge NVIDIA for providing access to a DGX B200 system.

References

[1] Xla: Optimizing compiler for tensorflow. https://www.tensorflow.org/xla, 2017. 14

[2] Nvidia/cutlass: Cuda templates for linear algebra subroutines. https://github.com/NVIDIA/cutlass, 2019. 10

[3] Transformer related optimizations. https://github.com/NVIDIA/FasterTransformer, 2020. 14

[4] Nvidia nccl. https://developer.nvidia.com/nccl, 2021. 2

[5] Flash-decoding for long-context inference. https://crfm.stanford.edu/2023/10/12/flashdecoding.html, 2023. 14

[6] Huggingface Models. https://huggingface.co/models, 2023. 10

[7] A Triton implementation of the FlashAttention2 algorithm. https://triton-lang.org/main/getting-started/tutorials/06-fused-attention.html, 2023. 10

[8] Accelerating MoE's with a Triton Persistent Cache-Aware Grouped GEMM Kernel. Link, 2025. 11

[9] Designing a Low-Latency Megakernel for Llama-1B. https://hazyresearch.stanford.edu/blog/2025-05-27-no-bubbles, 2025. 4, 14

[10] NVIDIA OpenSHMEM Library (NVSHMEM) Documentation. https://docs.nvidia.com/nvshmem/api/index.html, 2025. 2, 9

[11] Programmatic Dependent Launch. https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html, 2025. 1

[12] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. Tensorflow: A system for large-scale machine learning. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI, 2016. 14

[13] Osayamen Jonathan Aimuyo, Byungsoo Oh, and Rachee Singh. Flashdmoe: Fast distributed moe in a single kernel, 2025. 4, 14

[14] James Bradbury, Roy Frostig, Peter Hawkins, Matthew James Johnson, Chris Leary, Dougal Maclaurin, George Necula, Adam Paszke, Jake VanderPlas, Skye Wanderman-Milne, and Qiao Zhang. JAX: composable transformations of Python+NumPy programs, 2018. 3

[15] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Haichen Shen, Eddie Q. Yan, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. TVM: end-to-end optimization stack for deep learning. *CoRR*, abs/1802.04799, 2018. 1, 2, 3, 14

[16] Tianqi Chen, Lianmin Zheng, Eddie Yan, Ziheng Jiang, Thierry Moreau, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. Learning to optimize tensor programs. In *Advances in Neural Information Processing Systems 31*, NeurIPS'18. 2018. 14

[17] Yu Cheng, Lei Wang, Yining Shi, Yuqing Xia, Lingxiao Ma, Jilong Xue, Yang Wang, Zhiwen Mo, Feiyang Chen, Fan Yang, Mao Yang, and Zhi Yang. Pipethreader: Software-defined pipelining for efficient dnn execution. In *19th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2025. 14

[18] Tri Dao, Daniel Haziza, Francisco Massa, and Grigory Sizov. Flash-decoding for long-context inference, 2023. 1, 2, 14

[19] DeepSeek-AI. DeepEP: An efficient expert-parallel communication library. https://github.com/deepseek-ai/DeepEP, 2025. 14

[20] Siyuan Feng, Bohan Hou, Hongyi Jin, Wuwei Lin, Junru Shao, Ruihang Lai, Zihao Ye, Lianmin Zheng, Cody Hao Yu, Yong Yu, and Tianqi Chen. Tensorir: An abstraction for automatic tensorized program optimization, 2022. 14

[21] Raja Gond, Nipun Kwatra, and Ramachandran Ramjee. TokenWeave: Efficient compute-communication overlap for distributed llm inference. In *Proceedings of Machine Learning and Systems (MLSys)*, 2026. 14

[22] Bastian Hagedorn, Bin Fan, Hanfeng Chen, Cris Cecka, Michael Garland, and Vinod Grover. Graphene: An ir for optimized tensor computations on gpus. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, ASPLOS 2023, page 302–313, New York, NY, USA, 2023. Association for Computing Machinery. 14

[23] Ke Hong, Guohao Dai, Jiaming Xu, Qiuli Mao, Xiuhong Li, Jun Liu, Kangdi Chen, Yuhan Dong, and Yu Wang. Flashdecoding++: Faster large language model inference on gpus, 2024. 14

[24] Ke Hong, Xiuhong Li, Minxu Liu, Qiuli Mao, Tianqi Wu, Zixiao Huang, Lufang Chen, Zhong Wang, Yichong Zhang, Zhenhua Zhu, Guohao Dai, and Yu Wang. FlashOverlap: A lightweight design for efficiently overlapping communication and computation. In *Proceedings of the European Conference on Computer Systems (EuroSys)*, 2026. 14

[25] Muyan Hu, Ashwin Venkatram, Shreyashri Biswas, Balamurugan Marimuthu, Bohan Hou, Gabriele Oliaro, Haojie Wang, Liyan Zheng, Xupeng Miao, Jidong Zhai, and Zhihao Jia. Optimal kernel orchestration for tensor programs with korch. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, ASPLOS '24, page 755–769, New York, NY, USA, 2024. Association for Computing Machinery. 14

[26] Zhihao Jia, Oded Padon, James Thomas, Todd Warszawski, Matei Zaharia, and Alex Aiken. Taso: Optimizing deep learning computation with automatic generation of graph substitutions. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP '19, page 47–62, New York, NY, USA, 2019. Association for Computing Machinery. 4, 14

[27] Woosuk Kwon, Zhuohan Li, Siyuan Liu, Xin Wu, Michael Zeng, Xiangru Zhang, Yuhao Zou, and Scott Moritz. Efficient memory management for large language models. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, 2023. 9, 10

[28] Ravi Teja Mullapudi, Andrew Adams, Dillon Sharlet, Jonathan Ragan-Kelley, and Kayvon Fatahalian. Automatically scheduling halide image processing pipelines. *ACM Trans. Graph.*, 35(4), 2016. 14

[29] Tensors and Dynamic neural networks in Python with strong GPU acceleration. https://pytorch.org, 2017. 2, 3, 14

[30] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. Halide: A language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '13, 2013. 14

[31] Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. Megatron-lm: Training multi-billion parameter language models using model parallelism. *CoRR*, abs/1909.08053, 2019. 12

[32] Benjamin F. Spector, Simran Arora, Aaryan Singhal, Arjun Parthasarathy, Daniel Y. Fu, and Christopher Ré. Thunderkittens: Simple, fast, and adorable ai kernels. In *International Conference on Learning Representations (ICLR)*, 2025. 14

[33] Stuart H. Sul, Simran Arora, Benjamin F. Spector, and Christopher Ré. ParallelKittens: Systematic and practical simplification of multi-gpu ai kernels. In *Proceedings of Machine Learning and Systems (MLSys)*, 2026. 14

[34] NVIDIA TensorRT: Programmable inference accelerator. https://developer.nvidia.com/tensorrt, 2017. 14

[35] Tile-AI. TileRT: Tile-based runtime for ultra-low-latency llm inference. https://github.com/tile-ai/TileRT, 2026. 14

[36] Philippe Tillet, H. T. Kung, and David Cox. Triton: an intermediate language and compiler for tiled neural network computations. In *Proceedings of the 3rd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*, MAPL 2019, page 10–19, New York, NY, USA, 2019. Association for Computing Machinery. 1, 2, 10, 14

[37] Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, et al. Llama: Open and efficient foundation language models. *arXiv preprint arXiv:2302.13971*, 2023. 4, 14

[38] Colin Unger, Zhihao Jia, Wei Wu, Sina Lin, Mandeep Baines, Carlos Efrain Quintero Narvaez, Vinay Ramakrishnaiah, Nirmal Prajapati, Patrick S. McCormick, Jamaludin Mohd-Yusof, Xi Luo, Dheevatsa Mudigere, Jongsoo Park, Misha Smelyanskiy, and Alex Aiken. Unity: Accelerating DNN training through joint optimization of algebraic transformations and parallelization. In *16th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2022, Carlsbad, CA, USA, July 11-13, 2022*, pages 267–284. USENIX Association, 2022. 14

[39] Haojie Wang, Jidong Zhai, Mingyu Gao, Zixuan Ma, Shizhi Tang, Liyan Zheng, Yuanzhi Li, Kaiyuan Rong, Yuanyong Chen, and Zhihao Jia. PET: Optimizing tensor programs with partially equivalent transformations and automated corrections. In *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*, pages 37–54. USENIX Association, July 2021. 14

[40] Shibo Wang, Jinliang Wei, Amit Sabne, Andy Davis, Berkin Ilbeyi, Blake Hechtman, Dehao Chen, Karthik Srinivasa Murthy, Marcello Maggioni, Qiao Zhang, Sameer Kumar, Tongfei Guo, Yuanzhong Xu,

and Zongwei Zhou. Overlap communication with dependent computation via decomposition in large deep learning models. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1 (ASPLOS)*, pages 93–106, 2023. 14

[41] Mengdi Wu, Xinhao Cheng, Shengyu Liu, Chunan Shi, Jianan Ji, Man Kit Ao, Praveen Velliengiri, Xupeng Miao, Oded Padon, and Zhihao Jia. Mirage: A {Multi-Level} superoptimizer for tensor programs. In *19th USENIX Symposium on Operating Systems Design and Implementation (OSDI 25)*, pages 21–38, 2025. 1, 2, 4, 7, 9

[42] Yichen Yang, Phitchaya Phothilimthana, Yisu Wang, Max Willsey, Sudip Roy, and Jacques Pienaar. Equality Saturation for Tensor Graph Superoptimization. *Proceedings of Machine Learning and Systems*, 3:255–268, March 2021. 14

[43] Zihao Ye, Lequn Chen, Ruihang Lai, Wuwei Lin, Yineng Zhang, Stephanie Wang, Tianqi Chen, Baris Kasikci, Vinod Grover, Arvind Krishnamurthy, and Luis Ceze. Flashinfer: Efficient and customizable attention engine for llm inference serving, 2025. 1, 2, 10, 14

[44] Gyeong-In Yu, Joo Seong Jeong, Geon-Woo Kim, Soojeong Kim, and Byung-Gon Chun. Orca: A distributed serving system for {Transformer-Based} generative models. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 521–538, 2022. 10

[45] Lianmin Zheng, Chengfan Jia, Minmin Sun, Zhao Wu, Cody Hao Yu, Ameer Haj-Ali, Yida Wang, Jun Yang, Danyang Zhuo, Koushik Sen, Joseph E. Gonzalez, and Ion Stoica. Ansor : Generating high-performance tensor programs for deep learning. *CoRR*, abs/2006.06762, 2020. 14

[46] Lianmin Zheng, Liangsheng Yin, Zhiqiang Xie, Chuyue Sun, Jeff Huang, Cody Hao Yu, Shiyi Cao, Christos Kozyrakis, Ion Stoica, Joseph E. Gonzalez, Clark Barrett, and Ying Sheng. Sglang: efficient execution of structured language model programs. In *Proceedings of the 38th International Conference on Neural Information Processing Systems*, NIPS '24, Red Hook, NY, USA, 2024. Curran Associates Inc. 9

[47] Liyan Zheng, Haojie Wang, Jidong Zhai, Muyan Hu, Zixuan Ma, Tuowei Wang, Shuhong Huang, Xupeng Miao, Shizhi Tang, Kezhao Huang, and Zhihao Jia. EINNET: Optimizing tensor programs with Derivation-Based transformations. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*, pages 739–755, Boston, MA, July 2023. USENIX Association. 14

[48] Size Zheng, Wenlei Bao, Qi Hou, Xuegui Zheng, Jin Fang, Chenhui Huang, Tianqi Li, Haojie Duanmu, Renze Chen, Ruifan Xu, Yifan Guo, Ningxin Zheng, Ziheng Jiang, Xinyi Di, Dongyang Wang, Jianxi Ye, Haibin Lin, Li-Wen Chang, Liqiang Lu, Yun Liang, Jidong Zhai, and Xin Liu. Triton-distributed: Programming overlapping kernels on distributed ai systems with the triton compiler. arXiv preprint arXiv:2504.19442, 2025. 14

[49] Size Zheng, Yun Liang, Shuo Wang, Renze Chen, and Kaiwen Sheng. Flextensor: An automatic schedule exploration and optimization framework for tensor computation on heterogeneous system. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '20, page 859–873, New York, NY, USA, 2020. Association for Computing Machinery. 14

[50] Kan Zhu, Yufei Gao, Yilong Zhao, Liangyu Zhao, Gefei Zuo, Yile Gu, Dedong Xie, Zihao Ye, Keisuke Kamahori, Chien-Yu Lin, et al. {NanoFlow}: Towards optimal large language model serving throughput. In *19th USENIX Symposium on Operating Systems Design and Implementation (OSDI 25)*, pages 749–765, 2025. 3

A Artifact Appendix

Abstract

The artifact is the MPK compiler and runtime together with scripts that reproduce the paper's end-to-end latency results. MPK transforms a PyTorch model into a single mega-kernel; the scripts measure per-token decoding latency for MPK, PyTorch, vLLM, and SGLang across five LLMs, five batch sizes, and three GPU generations.

Scope

The artifact validates the paper's main performance claims: (i) on single-batch serving, MPK reduces per-token latency by 1.0–1.7× over vLLM and SGLang, with the largest gains on smaller models and newer GPUs (Figure 9); (ii) under tensor parallelism, MPK improves throughput by up to 10× over PyTorch and 1.1–1.4× over vLLM/SGLang on 8 H100s (Figure 11); and (iii) cross-task pipelining and compute–communication overlap each contribute measurable speedups (Figures 12 and 13).

Contents

The repository holds the MPK compiler and in-kernel runtime, a runnable demo for each model under `demo/`, and one evaluation driver per GPU under `artifact_evaluation/`. A driver sweeps the five models over batch sizes 1 to 16 for all four systems and writes one result file per run, recording the system, GPU, model, batch size, and measured per-token latency. Collecting these files reproduces Figure 9 and the ablation figures. A `setup.sh` script installs the dependencies and builds MPK.

Hosting

The artifact is on the frozen branch `tgx-osdi26-ae` of https://github.com/mirage-project/mirage (commit `8b981a4`), archived on Zenodo at https://doi.org/10.5281/zenodo.20563064 (DOI: 10.5281/zenodo.20563064).

Requirements

Hardware. One NVIDIA A100, H100 (SXM), or B200 for the single-GPU sweeps, and a multi-GPU node (4–8 H100s) for the tensor-parallel experiments. The four smaller models fit on a 40 GB A100; Qwen3-30B-A3B needs at least 80 GB and runs on an 80 GB A100, an H100, or a B200.
Software. Linux with CUDA (12.8 in the paper), PyTorch 2.7, NVSHMEM and OpenMPI for the multi-GPU runs, and `transformers` 4.57.1. `setup.sh` installs these and builds MPK; vLLM and SGLang each run in a separate virtual environment.

Benchmarks. Qwen3-0.6B, Llama-3.2-1B-Instruct, Qwen3-1.7B, Qwen3-8B (dense), and Qwen3-30B-A3B (MoE), downloaded from HuggingFace (Llama-3.2 is gated; set `HF_TOKEN`).

Reproducing the results

```
git clone --recursive -b tgx-osdi26-ae \
   https://github.com/mirage-project/mirage
cd mirage && bash artifact_evaluation/setup.sh
bash artifact_evaluation/<gpu>/run_tgx.sh
bash artifact_evaluation/<gpu>/run_pytorch.sh
bash artifact_evaluation/<gpu>/run_vllm.sh
bash artifact_evaluation/<gpu>/run_sglang.sh
```

All runs use offline batched inference with a fixed prompt length of 64, 1024 decoded tokens, and greedy decoding; each reported number is the median of five runs after a four-iteration warmup. On a 40 GB A100 the sweep covers the four smaller models; reproducing the Qwen3-30B-A3B point additionally requires an 80 GB A100 (or an H100/B200).

ARTIFACT EVALUATED
USENIX
AVAILABLE

GraCE: Unlocking CUDA Graphs with Compiler Support for ML Workloads

Abhishek Ghosh[†], Ajay Nayak[†], Ashish Panwar[‡], and Arkaprava Basu[†]
[†]*Indian Institute of Science* [‡]*Microsoft Research India*

Abstract

Machine learning (ML) workloads launch hundreds of short-running GPU kernels per iteration. With GPU compute throughput growing rapidly, CPU-side launch latency of kernels is emerging as a bottleneck. CUDA Graphs promise to address this by replaying a set of kernels with a single dispatch of the graph, removing per-kernel launch costs. However, CUDA Graphs remain surprisingly difficult to deploy correctly and efficiently.

We present GraCE — a compiler framework to maximize the coverage and benefits of CUDA Graphs for ML workloads. It introduces three novel optimizations: it applies automatic code transformations to make ML applications amenable for CUDA Graphs; it eliminates the parameter copy overheads for kernels executing in CUDA Graphs, and it selectively deploys CUDA Graphs guided by a cost–benefit analysis. For 25 ML workloads from TorchBench, HuggingFace and TIMM, GraCE more than doubles the benefit from deploying CUDA Graph compared to the most popular and widely used ML compiler PyTorch2. GraCE is built atop PyTorch2's compilation framework and requires no programmer intervention.

1 Introduction

The demand for GPU compute continues to skyrocket due to a surge in ML applications [2, 8, 9, 34, 37, 69–71, 95]. Catering to this growing demand through hardware enhancement *alone* is challenging in the post-Moore era of slowing transistor technology [14]. Consequently, hardware vendors increasingly rely on hardware-software co-design to improve efficiency. For example, NVIDIA continues to introduce new performance features in GPUs that need software enablement. Recent examples include scoped synchronization [29, 30, 39] that allow programmers to selectively enable faster synchronization, tensor cores for fast matrix multiplication [27], lower- and mixed-precision computation [38], and CUDA Graphs [20, 90] for improving GPU utilization.

Surprisingly, GPUs are often underutilized despite high demand. Several hyperscalers, including Alibaba and Azure, report GPU utilization typically below 50% [18, 26, 79, 80]. A major contributing factor is that CPUs cannot submit kernels fast enough to keep modern GPUs busy. An ML application typically invokes hundreds of GPU kernels per iteration, and each kernel launched from the CPU adds 5-10 microseconds of latency [13]. As a result, GPUs stall when they finish executing kernels faster than the CPU can dispatch subsequent ones. This problem is amplified by recent trends in distributed ML, particularly Tensor Parallelism (TP) [68]. TP shards model layers across GPUs, reducing the runtime of per-GPU compute kernels while introducing additional kernel launches for inter-GPU collective operations. Together, these effects push launch latency deeper into the critical path.

CUDA Graphs can mitigate the kernel launch bottleneck. It allows software to *capture* kernel launches in a directed acyclic graph – a CUDA Graph. The CPU then launches the graph instead of individual kernels. This can significantly speed up applications that repeatedly execute the same set of kernels (e.g., training and inference). While useful in theory, CUDA Graphs come with several constraints. For example, the parameters to a kernel are hardcoded during a graph's capture. The same parameter values are passed to kernels during replays, posing a challenge to applications that compute over different data (tensors) across invocations. A CUDA Graph is also prohibited from using GPU-synchronizing operations.

Effectively harnessing capabilities of CUDA Graph is further made challenging by the fact that most ML applications are written in high-level frameworks such as PyTorch [54] and TensorFlow [1], and are semantically distant from their execution on the hardware. A key challenge, thus, is to effectively bridge the gap between the high-level program semantics and the low-level idiosyncrasies of hardware features.

Our goal is to enable high-level programs, e.g., in PyTorch, to effectively harness CUDA Graphs in modern GPUs without needing programmer intervention. Upon qualitatively and quantitatively analyzing many tens of PyTorch programs, we observed three key shortcomings that either prevent the deployment of CUDA Graphs or fail to harness its benefits.

① **CUDA Graph-oblivious programs:** Many important

ML programs, e.g., in TorchBench [22], are *not* written with CUDA Graph in mind. For example, it is common for programs, such as the text-to-image generator model DALLE-2 [62], to allocate a tensor on the CPU's DRAM (say, address `x`) and then copy its content to GPU memory during computation, e.g., in the `forward` pass. A CUDA Graph captured for such computation would hardcode the address of the DRAM-resident tensor (`x`). Its content must then be copied to GPU memory during the graph's replay. However, the host program may de-allocate the CPU tensor (`x`) after the graph's capture, causing the application to crash intermittently during replays. Thus, CUDA Graph must be avoided in such programs. We notice similar pattern in other programs e.g., Vision Mask R-CNN [23], Torch Multi-modal Clip [31, 61].

Further, in many cases one or more kernel parameters are passed as *scalar* variables e.g., attention kernel in speech transformer [55]. A CUDA Graph captured with such kernels would record the value of the scalar as the kernel parameter. If the variable's value should change across kernel invocations, it will receive *stale* values during graph replays. Thus, CUDA Graph must be avoided in such cases.

While a single or a few lines of code changes can unlock hundreds of kernels being captured in CUDA Graphs, the challenge is to automatically find such opportunities and update the code accordingly. It is unrealistic to expect developers to continue evolving their programs to make them friendlier to new hardware features. Thus, we posit that frameworks, such as PyTorch2, JAX [5], must step up to effectively harness advanced hardware features without burdening developers.

② **Parameter copy overheads in replaying CUDA Graphs:** By design, CUDA Graphs capture GPU kernel parameters, including addresses of tensors *by value*. However, different invocations of the same kernel across a graph's repeated replays often need to operate on different data. A simple strategy to enable this is to replace mutable parameters with *placeholder* parameters. Before each replay, actual parameter values for a given invocation are copied onto its placeholders.

While this strategy allows correct replay of graphs and is employed by widely used PyTorch2 ML compiler, it adds the overhead of copying the parameter (data) into the placeholder in the critical path of every graph replay. We empirically found overheads of copying parameters constitute up to 24% of a CUDA Graph's execution time, particularly when operating on large tensors (§2). Thus, there is a need to correctly deploy CUDA Graph but without slowing down graph replays.

③ **CUDA Graphs may hurt performance:** While CUDA Graph is a performance feature, we discovered that the overhead of replaying them can sometimes outweigh the benefits. Besides the latency of copying kernel parameters (tensors), the overhead of reusing/garbage collecting CUDA Graph's memory after replay and reset of the GPU's random-number generator (RNG) state before each replay adds latency to the critical path. In our analysis, nearly a quarter of the total CUDA Graphs degrade performance (up to 397%). While a performance feature is not expected to speed up every application, it must not severely slow down many applications.

GraCE: Driven by these observations, we created a CUDA Graph-aware compilation framework for PyTorch programs. GraCE embodies three key optimizations.

① It employs CUDA Graph-aware code transformations such as hoisting memcopies, changing data placement for a wider deployment of CUDA Graphs. It analyzes the intermediate representation (IR) of programs to root cause the failures to capture kernel(s) in a graph (e.g., scalar variables). It then updates the IR to emulate changes in the source that would have enabled CUDA Graph. ② It introduces an additional level of indirection in kernel parameters to convert tensor copies for parameters to *pointer* copies. While a tensor can be thousands of bytes, a pointer is only 8 bytes. For JIT-compiled kernels, GraCE uses Triton compiler [52] to automatically update kernels to de-reference their indirected parameters. For vendor-supplied immutable kernels (e.g., from cuBLAS), it introduces prelude kernels in the graph that perform de-reference of indirect-ed parameters on behalf of immutable kernels. ③ Finally, it adds automated profiling during graph capture (slow path) to analyze the cost-benefit of deploying CUDA Graphs. This guides GraCE to *selectively* deploy CUDA Graph during replay (fast path) only when beneficial. The judicious deployment of CUDA Graphs not only enables better end-to-end performance but, crucially, ensures that no application slows down. GraCE is integrated into PyTorch2 and requires *no* program modifications.

On a diverse set of 25 ML applications, GraCE provides geomean speedup of 29% over PyTorch2's CUDA Graph feature (and up to 3.36×) on an H100 GPU. Further, we studied GraCE's efficacy under distributed ML workloads by spreading a subset of applications over four GPUs using tensor parallelism. There, GraCE speeds up applications up to 3.56× over PyTorch2's CUDA Graph and by 75% on average.

The key contributions of our work are as follows:

- We demonstrate subtle programming idiosyncrasies in popular PyTorch programs, coupled with the constraints of CUDA Graphs, limit wider deployment of CUDA Graphs.
- We show that overheads due to copying kernel parameters can be limited by introducing an additional layer of indirection in passing the parameters to kernels.
- We demonstrate the need to selectively deploy CUDA Graphs to achieve better end-to-end application performance.
- We create GraCE atop PyTorch2's compilation framework that achieves the above-mentioned optimizations without needing any manual modifications.

2 Background and Motivation

This section elaborates on how CUDA Graphs have become indispensable for modern GPU platforms, along with the challenges involved in maximizing their utilization. We use the terms CUDA Graph(s) and CG(s) interchangeably.

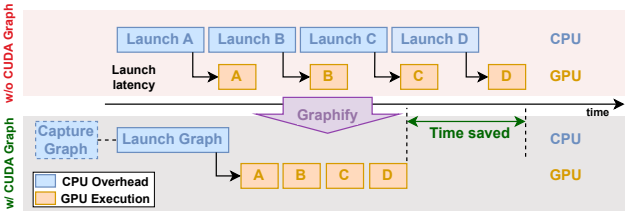


Figure 1: CUDA Graph reducing CPU launch overheads.

2.1 The need for CUDA Graphs

GPUs have become the workhorse accelerators for ML applications since they deliver massive compute throughput. NVIDIA's GPU compute scaling highlights this: peak FP16 tensor throughput jumped from 312 TFLOP/s on A100 [40] to ≈1 PFLOP/s on H100 [45], and ≈4 PFLOP/s on B200 [50]. In contrast, CPU frequency has stagnated, and the number of cores has only doubled during that period. At the same time, ML workloads have expanded dramatically, with each iteration often launching hundreds to thousands of kernels. Further, modern deployments increasingly shard a model across multiple GPUs. As GPU throughput improves and models scale out, kernels become smaller and faster.

A typical CUDA kernel launch takes ∼5-10 microseconds [13], while many kernels now run for only a few microseconds. This turns CPU launch overhead into a significant fraction of the total time. Reducing or amortizing this overhead is therefore critical to achieving high GPU utilization. For example, in DALLE2 [62] (inference, batch size 1), a segment of computation issues over 740 kernels whose combined GPU execution time is only 3.4 milliseconds on an H100 GPU, while the end-to-end time is 14 milliseconds—implying that 75% of the total latency is due to CPU launch delays.

A CG can address this bottleneck by obviating the need to individually launch each kernel from the CPU. The top part of Figure 1 shows traditional kernel launches, while the lower part shows steps involved in CG. Kernels are captured as part of a DAG (CG) once. The graph is launched from the CPU, and internally, the GPU hardware launches CG's constituent kernels without CPU intervention. Applications that repeatedly launch the same set of kernels, e.g., inference tasks, can benefit substantially by capturing the graph once and replaying it many times.

2.2 Deploying CUDA Graph and its challenges

CUDA supports automatic capture of CGs through *stream capture* mechanism. The developer designates a capture region. The runtime automatically records all GPU operations (e.g., kernel launch, memcopy) issued within the region, producing a computation DAG. Once capture completes, the developer instantiates the graph and launches it, causing the GPU to *replay* the recorded GPU operations. Owing to its simplicity, stream capture is the dominant way developers use CGs. However, even this approach requires care: the runtime records kernel parameters *by value*. If the application replays the CG without updating parameters that change across iterations, execution proceeds with stale arguments—leading to silent data corruption, incorrect results, or crashes. Developers must therefore identify which values remain stable across replays and update any mutable parameters in the graph before launching it. Furthermore, CUDA does not permit capturing synchronous copies (`cudaMemcpy`), memory allocation (`cudaMalloc`) or any other API that implicitly synchronizes the device [48]. These constraints force developers to rethink data movement and memory management, making it difficult to incorporate CGs—especially in legacy codebases.

```
1  import torch
2  import numpy as np
3
4  class ScaledDotProductAttention(nn.Module):
5    def __init__(self, temperature, attn_dropout):
6      ...
7      self.temperature = temperature
8      ...
9
10   def forward(self, q, k, v, mask=None):
11     attn = torch.bmm(q, k.transpose(1, 2))
12     # Divide operation below with a numpy scalar on CPU
13     attn = attn / self.temperature # offending line
14     ...
15
16 class MultiHeadAttention(nn.Module):
17   def __init__(self, n_head, d_model, d_k, d_v, dropout):
18     ...
19     temp = np.power(d_k, 0.5) # src of offending operand
20     self.attn = ScaledDotProductAttention(
21         temperature=temp, attn_dropout=dropout)
22     ...
23
24   def forward(self, q, k, v, mask=None):
25     ...
26     output, attn = self.attn(q, k, v, mask=mask)
27     ...
28
29 module = # Initialize the speech_tranformer model
30 module = torch.compile(module)
31 result = module(inputs)
```

Figure 2: Simplified code snippet from speech transformer model with CPU-resident scalar (marked red).

We posit that compilers and *not* the programmers should reason about program structure, automatically satisfy graph constraints and judiciously apply optimizations. This is the only way to harness advanced hardware features without burdening the developers. Unfortunately, existing compilers fall well short. To substantiate, we investigate PyTorch2 [3]—the most popular and widely used ML compiler.

2.3 CUDA Graphs are poorly harnessed

We analyze over 60 common ML and HPC workloads to identify key impediments to effective CG deployment. First, many programs fail to use CGs because their implementations are not aligned with CG requirements. Second, CG's deployment strategy today adds significant but avoidable overheads. Fur-

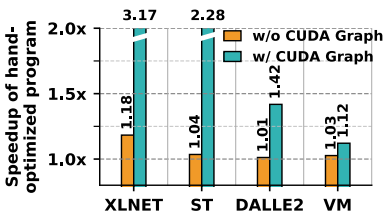
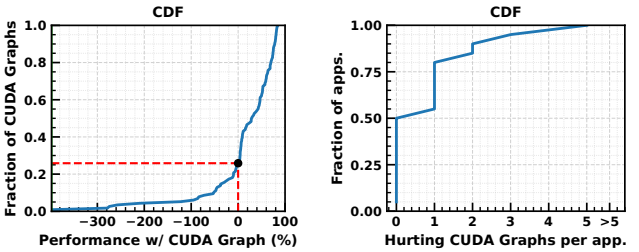


Figure 3: Performance impact of moving tensors from CPU (unoptimized programs) to GPU (optimized programs).



(a) 25% of the total CUDA Graphs hurt performance in TorchBench. **(b)** About 50% of the applications contain CUDA Graph(s) that hurt performance.

Figure 4: Cost-benefit analysis of CUDA Graphs.

ther, even when CGs can be deployed, their deployment is not always beneficial — yet current frameworks deploy them blindly. We present a detailed analysis below.

2.3.1 Obstacles in harnessing CUDA Graphs

We observe that CUDA Graphs are often not compatible with the way many ML workloads are written. For example, when a kernel is captured into a CG, all its arguments are recorded by value and baked into the graph. If those values change across iterations, replaying the captured graph would produce incorrect results. Consequently, a compiler would forgo CG deployment in these cases for correctness.

We highlight one such example with a simplified PyTorch code snippet of the speech transformer model [55] (Figure 2). In the constructor of `MultiheadAttention` (line 19), the variable `temperature` is assigned the value returned by a `numpy` method [74]. Note that `temperature` hosts a *scalar* value and is allocated on the CPU DRAM. This variable is passed to the constructor of `ScaledDotProductAttention` (line 21) and ultimately used to update `attn` in its forward method (line 13). This update of `attn` is performed by a GPU kernel whose parameters are `attn` and `temperature`. While the former is the address to a GPU tensor, the latter is a *scalar* variable. If this kernel is captured as part of a CUDA Graph, the scalar value of the parameter `temperature` will be hardcoded as a kernel parameter. Later, if the captured graph is replayed, the kernel would execute with a stale value of `temperature`.

We discovered other subtleties in PyTorch programs that limit CG deployment. A common case is when applications allocate a tensor on the DRAM (at address `x`) and copy its contents to GPU memory at the start of each iteration. A captured CG would hardcode the DRAM address `x`. However, CPU may de-allocate that tensor after capture, leaving the address invalid when the graph is replayed.

Quantifying the opportunity cost: The issues above stem from the fact that most ML programs are not written to meet CG constraints. Before the introduction of CG, the choice of where to allocate tensors had a limited impact. Tensors allocated on the CPU DRAM are copied to the GPU memory over PCIe for computation. If the tensor is large, the PCIe transfer time can dominate the kernel's execution time. Thus, a tensor placed in CPU DRAM can at most delay the execution of the corresponding kernel.

The same is not true in the context of CG. The choice of tensor placement for *one* kernel determines whether other surrounding kernels can be captured as part of a CG or not. This is because a single offending allocation for a kernel in an otherwise CG-amenable long chain of kernels can disallow any of the kernels from being captured in a CG. In short, how a tensor is allocated for a kernel can affect many *other* kernels in the context of CG.

To demonstrate this quantitatively, we manually rewrote small portions of several PyTorch models—XLNetLMHeadModel (XLNET) [85], Speech-Transformer (ST) [55], DALLE2 [62], and Vision Mask R-CNN (VM) [23]—specifically targeting code regions whose structure prevented CUDA Graph deployment. The edits were minor (e.g., relocating a CPU-allocated tensor to the GPU) but made the code CG-compatible. Figure 3 reports the results (all results use inference batch size 1). In the absence of CGs, optimizing tensor placement had a modest impact on performance, averaging 6.29% and reaching up to 18% for XLNET. However, the same code changes lead to a dramatic speedup when using CGs. For example, XLNET and ST sped up by 3.17× and 2.28×!

Furthermore, scaling out ML models to multiple GPUs makes CGs more critical for performance. For instance, in the XLNET model (inference at batch size 128), tensor-parallelism over four GPUs brings 48 additional collective kernels and reduces the runtime of more than 250 kernels by 50–75%. However, a few variable allocations in CPU DRAM render more than 450 kernels ineligible for capture in CGs, leaving significant performance on the table.

2.3.2 Cost-benefit analysis of CUDA Graphs

We also noticed that even when CGs can be deployed, they do not always improve performance. In fact, counterintuitively, we observed that CGs also hurt performance in many cases. This is unexpected; CGs are designed solely as a performance feature and should not slow down even a single application.

To understand this behavior, we quantitatively analyze 116 CGs drawn from 63 PyTorch applications in TorchBench [22].

We make two key observations. First, Figure 4a shows the CDF of performance changes when enabling CGs across all applications, capturing both cases where CGs help (right side of 0 on the x-axis) and where they hurt (left side of 0). Overall, 25% of the graphs (29 of 116) degrade performance. A deeper inspection reveals that the dominant source of this slowdown is the latency of copying kernel parameters (data) into the static placeholders (addresses) during a graph's replay.

The reason this copying is required is fundamental to how CGs operate: every kernel parameter is captured by value in a CUDA Graph, whereas different invocations of a kernel typically operate on different data. A simple strategy to manage this scenario is to replace mutable parameters with *place-holder* arguments. Before each replay, the runtime must overwrite these placeholders with the current input values of all mutable arguments. This copy happens on the critical path of every replay and scales with the number of kernels and the size of their argument lists, making it a nontrivial overhead. For instance, in CUDA Graphs of DALLE2 and NVIDIA Deep Recommender, parameter copying accounts for 24% and 17% of their end-to-end execution time, respectively. These overheads substantially erode the benefits of CG deployment and can even lead to net slowdowns.

Beyond parameter copy, replay of a CG suffers from other fundamental overheads. CGs cannot encode lifetimes of CPU-side objects. A language runtime that performs garbage collection, e.g., Python, must separate CG outputs from the host objects that originally produced them. The runtime snapshots only the GPU-side metadata of CG outputs (such as data pointers and layout) and discards the CPU-side tensor objects. This is necessary for the GPU memory to be safely reused across CG replays without memory bloat. However, on each replay, fresh CPU objects must be reconstructed from the saved metadata, adding further latency.

Our second insight is that many applications consist of both useful and harmful CGs. For example, Figure 4b shows the CDF of 20 applications that contain multiple CGs. Of these, nearly half of them exhibit a mix of performance-aiding and performance-hurting CGs. For a specific example, in DALLE2 (inference, batch size 1), one of the captured CG executes in 75 microseconds, while the kernel in it when run without CG takes 70 microseconds—implying an overhead of 7% in CG deployment. At the same time, a different CG in the same workload reduces the runtime of 743 kernels to 3.4 milliseconds, compared to 14 milliseconds without CGs, resulting in more than 75% performance gain. Unfortunately, CGs today are blindly deployed without a cost-benefit analysis.

In summary, these observations expose a deeper problem: today's ML software stack lacks both the semantic understanding needed to effectively harness CUDA Graphs and a cost model to decide when they are worthwhile. Applications often miss crucial opportunities to use CGs, or deploy them where they hurt performance. It begs for a compiler-driven approach to automatically enforce graph compatibility, reason about costs, and maximize the effective use of CGs.

3 GraCE: Goals and Optimizations

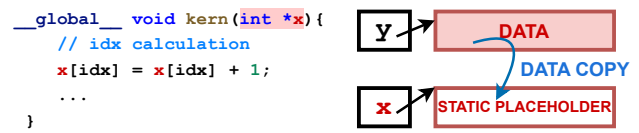
Motivated by the analysis, we present the goals of GraCE, a framework that enables effective use of CUDA Graphs in ML applications. We then outline the optimizations that help GraCE achieve the goals.

Goals: ① Expand the set of GPU kernels that can be captured in CGs. ② Minimize runtime overheads introduced during CG replay, especially those stemming from parameter copy. ③ Selectively deploy CGs only when they improve end-to-end performance. ④ Achieve all of the above without requiring program modifications or developers' intervention.

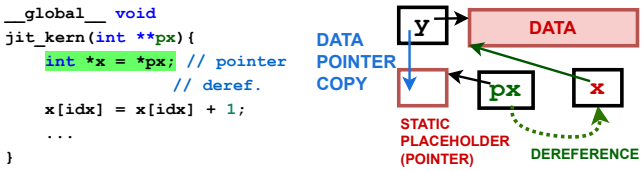
CUDA Graph-aware Code Transformation (CGCT): A large fraction of ML applications fail to utilize CUDA Graphs due to the presence of constructs—synchronous memory copies, CPU-resident scalars or tensors—that inhibit CG capture. These issues are often obscured by high-level frameworks and can easily escape a programmer's attention.

GraCE addresses this by restructuring the program such that large sequences of kernels become eligible for graph capture, without altering application semantics. It first analyzes the compiler's intermediate representation (IR) to identify code regions where CG capture fails and the root cause of the failure. It then performs targeted IR rewrites that make these regions compatible with CG capture. Conceptually, GraCE ensures that variables and operations in a candidate region satisfy the constraints imposed by CGs: computations and necessary inputs are placed on the GPU, and incompatible operations are hoisted prior to candidate graph regions. For example, if a scalar parameter prevents graph capture, GraCE alters the type of parameter as a GPU-resident tensor. During replays, the appropriate value of a parameter is copied into the tensor's placeholder, enabling safe capture of the surrounding kernels. Likewise, if a DRAM-to-GPU memory copy prevents CG deployment, GraCE hoists the copy above the sequence of kernels targeted for CG capture. This ensures that the copy is not captured as part of the CG and, thus, need not be replayed.

Parameter Indirection (PI): A dominant source of overhead in a CG replay is the copying of kernel parameters (data) into the placeholders that were recorded during graph capture. We eliminate this cost by removing data copies from the replay path entirely: instead of copying potentially large parameter data, we introduce a layer of indirection. Consequently, the replay updates only the corresponding pointers (eight bytes per parameter) rather than the data itself, which can be thousands of bytes. This optimization rests on a simple observation: with the input data changing on each replay, as long as a kernel can read from these new data tensors—different from those it



(a) Baseline behavior in current ML frameworks.



(b) After indirection.

Figure 5: Converting data (parameter value) copy to pointer copy through indirection in Parameter Indirection.

was captured with—the CG execution remains valid, without needing a parameter copy.

Figure 5 illustrates this mechanism with an example. Figure 5a shows the parameter (data) copy performed during the replay of a CG in the baseline. The kernel parameter recorded during graph capture (here, address x) is highlighted in red. During replay, the parameter for a given kernel invocation, residing at address y, must be copied into the memory location pointed to by x. Figure 5b shows how GraCE converts the kernel parameter to pointer-to-pointer via indirection. These pointer-to-pointers are then recorded during graph capture. Before each replay, the pointers to the parameter (y) for the given invocation of the kernel are copied onto the address (pointer-to-pointer, px) recorded during the graph capture. The modified kernel then de-references these pointer-to-pointers before performing any computation (green).

Our approach relies on the ability to rewrite kernels, which is feasible for JIT-compiled kernels (e.g., Triton [52]) but not for vendor-supplied libraries (e.g., CUTLASS [42], cuBLAS [41]) or pre-compiled kernels. In §4, we describe an alternative mechanism that extends PI to these kernels based on recently introduced CG management APIs [47], while meeting GraCE's goal of zero manual changes.

An additional benefit of our approach is that PI eliminates the need to keep static placeholders allocated for input tensors over a CG's lifetime. Instead, it requires a pointer-to-pointer whose contents are updated before each replay. This avoids duplicate storage, reducing the peak memory footprint by up to 15% during replay.

Selective CUDA Graphs (SCG): Since CGs can also hurt performance (§2), they should not be used blindly. GraCE employs automated profiling as part of the compilation and graph capture phase (slow path). Specifically, GraCE measures the aggregate time to execute individual kernels encapsulated in a CG. It also profiles the execution time of the corresponding CG with and without Parameter Indirection. The profiler then automatically chooses the best-performing configuration for the given set of kernels in an application. This decision is made independently for each of the CGs. Also, note that this profiling happens during compilation and CG capture phase (slow path) and not during the graph replay (fast path).

Taken together, these three optimizations push CUDA Graphs closer to their peak potential. CGCT broadens capture eligibility so that substantially more of the program can benefit from graph execution. PI removes a major replay bottleneck by eliminating parameter-copy overheads. SCG ensures that CGs are enabled only when they deliver end-to-end gains. The result is a system that captures larger kernel regions in graphs, executes them with lower overhead, and deploys CGs only where they improve performance.

4 Implementing GraCE

GraCE is built into PyTorch2's compilation workflow (slow path) that is responsible for CUDA Graph capture, enabling its optimizations to benefit fast-path execution (replay).

We chose PyTorch as the platform to build GraCE in order to seamlessly bring the benefits of CUDA Graph to a wide range of ML workloads. Arguably, PyTorch is the most popular framework for creating ML applications [6, 17, 35].

We begin with a brief discussion of the compilation workflow of the PyTorch2 compiler and then discuss how GraCE leverages it to apply its three optimizations.

Compilation workflow in GraCE: GraCE follows a structure commonly found in modern ML compilers—such as PyTorch2 [3], JAX/XLA [5]. The typical compilation pipeline consists of ① *Computation Graph Construction Layer (Frontend)*, where Python execution is symbolically traced, producing a computation graph in a high-level IR. ② *IR Lowering & Optimization Layer (Mid-Level Pass)*, where the graph is lowered into a more uniform IR, enriched with metadata such as tensor shapes, device placement, and debug mappings. ③ *Kernel Generation Layer (Backend)*, where primitive operations are either dispatched to optimized GPU libraries (cuBLAS [41], cuDNN [43], CUTLASS [42]) or implemented through just-in-time (JIT) kernel generators such as Triton [52]. ④ *Graph-Capture Layer*, where the final compiled module is executed, and if eligible, captured as a CG. This layer maintains parameter placeholders and bookkeeping logic required to replay CGs.

GraCE adds targeted extensions to each of these layers in PyTorch2. Figure 6 provides an overview of GraCE's implementation, depicting how it extends selected components of the PyTorch2's compilation workflow. The orange and purple background regions delimit the key components of the PyTorch2 compiler—its frontend (Torch Dynamo [57]) and backend (Torch Inductor [25]). Modifications for CGCT are shown in blue, while those for PI and SCG appear in yellow and green, respectively. GraCE's three optimizations —

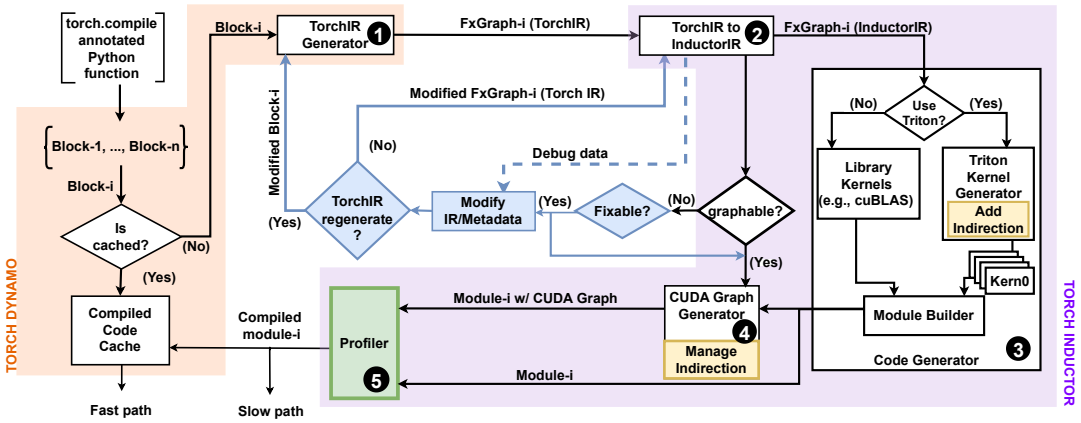


Figure 6: Implementation of GraCE within PyTorch2's compilation framework.

CGCT (blue), PI (yellow), and SCG (green) — extend different parts of the compilation workflow and are detailed in §4.1, §4.2, and §4.3, respectively.

4.1 CUDA Graph-aware Code Transformation

GraCE automatically finds and mitigates multiple factors that prevent deployment of CUDA Graphs. For ease of exposition, we describe the PyTorch2 components involved in deploying CGs first, and then GraCE's extensions to them.

4.1.1 Computation Graph Construction Layer

Torch Dynamo (Figure 6, orange block on the left) intercepts Python execution and constructs computation graphs (FxGraphs [63]) consisting of TorchIR — a simplified IR that exposes Python and PyTorch operations for optimization. Dynamo symbolically traces operations until it encounters one that cannot be safely traced (e.g., input-dependent control flow). It terminates the current FxGraph, compiles it, and splices the compiled result back into the source program via bytecode rewriting. It resumes tracing after the offending operation, effectively partitioning the program into blocks, each of which is lowered independently to TorchIR (① in Figure 6).

GraCE's extensions: Dynamo is extended to regenerate TorchIR from a modified code block (① in Figure 6), provided by the extensions to Torch Inductor described next.

4.1.2 IR-Lowering & Optimization Layer

Torch Inductor (Figure 6, purple region), lowers the TorchIR produced by Dynamo into a lower-level IR comprised of primitive Aten/Prim [56] operations. Inductor then augments this IR with operand metadata–including tensor shapes, device placement (CPU or GPU), and debug mappings from this lowered IR back to TorchIR–producing InductorIR (② in Figure 6). InductorIR serves as input to subsequent phases (§4.2) that apply further optimizations and code transformations.

Inductor also decides whether a FxGraph represented by an InductorIR instance can be captured as a CUDA Graph. If the metadata shows that an FxGraph's node has a CPU-resident scalar operand, performs a CPU–GPU memcopy, or involves an input mutation, Inductor avoids encapsulating it in a CG.

GraCE's extensions: To map more FxGraphs to CGs, GraCE enhances Inductor's decision routine for CG eligibility. When this routine determines that an FxGraph cannot be mapped to a CG, GraCE inspects the cause of failure. If the failure is due to the presence of one or more scalar variables (e.g., a NumPy scalar as discussed in §2), it uses bytecode rewrites to cast the variables as GPU tensors, effectively removing the CPU operand/operation from the graph capture path. Dynamo then regenerates TorchIR for the modified block, which is re-lowered to InductorIR, enabling the FxGraph to be mapped to a CG.

Figure 7 illustrates this transformation for the `temperature` scalar from the speech transformer example in Figure 2. Specifically, it shows the Dynamo-generated bytecode without (Figure 7a) and with CGCT (Figure 7b). GraCE analyzes the Inductor IR to notice that the scalar input

Without CGCT

```
graph_out_0 =
__compiled_fn_43(
 ..., torch.as_tensor(
  __import_torch.as_tensor(
   self.layer_stack[0]
    .slf_attn
    .attention
    .temperature
   )), ...)
```

(a) Original bytecode.

With CGCT

```
graph_out_0 =
__compiled_fn_43(
 ..., torch.as_tensor(
  __import_torch.as_tensor(
   self.layer_stack[0]
    .slf_attn
    .attention
    .temperature,
   device='cuda')), ...)
```

(b) Modified bytecode.

Figure 7: IR modification as part of CGCT.

to a kernel is resident on the CPU DRAM. It then uses debug metadata to backtrace the offending Dynamo-generated bytecode that creates a PyTorch tensor. GraCE then rewrites this bytecode to construct the tensor on the GPU (here, add `device='cuda'`). The rewritten IR then makes the surrounding FxGraph eligible for CG capture.

A CPU-to-GPU memcopy can also prevent Inductor from deploying a CG. Using debug metadata, GraCE backtraces the InductorIR node to the Python object owning the offending CPU tensor and copies it to the GPU. This effectively changes the variable's device placement, hoisting the memcopy into the object's constructor so that it precedes any computation. As a result, memcopy gets removed from the graph-capture path, and the corresponding module can be mapped to a CG. Similarly, a CG may not get deployed if the output of an InductorIR node is a CPU tensor. In such a case, GraCE updates the metadata of the corresponding TorchIR node to mark the output as a GPU tensor (i.e., setting `device='cuda'`). The updated TorchIR is then re-lowered to InductorIR (② in Figure 6), removing the CPU tensor from the graph-capture path and allowing the FxGraph to be mapped to a CG.

4.2 Parameter Indirection

We detail how GraCE achieves its goal of reducing parameter copy overheads through indirection. This needs modifying how kernels within CUDA Graphs obtain their parameters.

4.2.1 Kernel-Generation Layer

The code generator (③ in Figure 6) is responsible for generating an optimized executable (module) for an FxGraph. It uses different frameworks/libraries depending on the target hardware. Here, we focus on systems with NVIDIA GPUs.

The code generator either invokes kernels from vendor-supplied external libraries (e.g., cuBLAS [41]) or generates kernels just-in-time (JIT) using frameworks such as OpenAI's Triton [52]. It maintains a list of InductorIR ops (e.g., matmul, convolution) that use kernels from vendor-supplied libraries and emits direct calls to the relevant library routines. Remaining operations are implemented just-in-time using Triton. For such operations, the code generator constructs a Triton function in Triton's Python-based DSL, decorates it with `triton.jit` for JIT compilation. It then invokes the Triton compiler to generate CUDA binaries. A module created for an FxGraph may contain both Triton and vendor-supplied kernels. After resolving all operations, the module builder assembles these components into a single, optimized module.

GraCE's extensions: Recall that the goal of GraCE is to enable repeated replay of a captured CG with different parameter values, but without adding the overhead of copying the parameter. It makes two changes to kernels to turn parameter (data) copies into pointer copies. First, it replaces each pointer argument with a pointer-to-pointer that holds the address of the data the kernel will access. Second, before the kernel starts executing, it dereferences these pointers-to-pointers, allowing the rest of the kernel to access the actual data correctly.

GraCE employs two distinct implementation approaches to achieve the above objective, depending on whether the given kernel is generated just-in-time or is a vendor-supplied one. While the former can be updated during compilation, the latter *cannot* – requiring two different approaches.

Enabling PI in JIT-ed kernels: To apply indirection to eligible Triton-generated JIT kernels without manual intervention, GraCE identifies the parameters that must undergo indirection during the code generation phase [1] and attaches this information to the `torch.jit` decorator (yellow element in ④ in Figure 6). GraCE's custom LLVM [10] pass integrated into the Triton framework then retrieves this parameter list from the decorator and rewrites the PTX [46] code of the corresponding kernels. The pass updates the kernel signatures to use 'pointer-to-pointer' arguments for the designated parameters. Further, it inserts PTX instructions at the beginning of the kernel to de-reference these pointers before using them in computation. The rest of the kernel code remains unchanged.

Enabling PI in vendor-supplied kernels: The approach of rewriting kernels is not feasible for those from vendor-supplied external libraries. These kernels are often supplied as compiled binaries. To work around this fundamental limitation, GraCE introduces an additional *prelude* kernel in the CUDA Graph at the start of the graph. The prelude kernel updates the parameters of the vendor-supplied kernel to point to actual data for a given invocation on the fly during replays.

The prelude kernel is provided with pointer-to-pointers to the parameters of the vendor-supplied kernel, whose values change across graph replays. It is tasked with performing the dereferencing of the pointer-to-pointer argument for the immutable vendor-supplied kernel. GraCE compiles the prelude kernel on the fly using NVRTC [51], and re-records the CUDA Graph with this kernel inserted before any other nodes. To achieve the effect of dereferencing, the prelude kernel must be able to update parameters to the vendor-supplied kernel on the fly during replay.

Fortunately, CUDA runtime version 12.4 introduced new device-side APIs to update parameters of a node in CUDA Graph from within another kernel (`cudaGraphKernelNodeSetParam`) [47]. The prelude kernel uses this API to update the parameters of the vendor-supplied kernel during replay.

The device-side APIs to update kernel parameters take three arguments: ① the kernel-node handle, ② a device-memory address (pointer) that holds the replacement pointer value, and ③ the offset within the kernel node's parameter

[1] Not all kernels or parameters require copying parameter data into placeholders. Outputs produced by earlier kernels in a CG can be passed directly to later kernels. Only kernels whose parameters originate from external sources (e.g., application inputs) require indirection. GraCE selectively applies indirection only for such parameters.

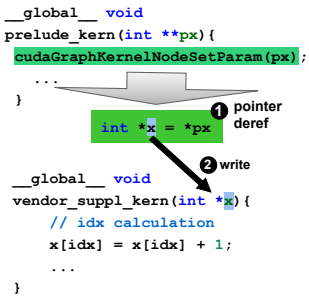


Figure 8: Indirection in vendor-supplied kernels.

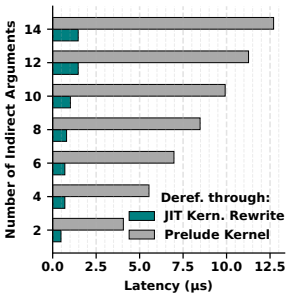


Figure 9: Latency comparison of PI approaches.

buffer. The API de-references the input pointer and writes the resulting value into the parameter buffer at the specified offset for the given kernel node. This operation performs a de-reference followed by a memory write.

The next challenge is to find the offset of the argument to be updated within the kernel's parameter buffer. For this purpose, GraCE uses another CUDA API `cuFuncGetParamInfo` to obtain a handle to the kernel's parameter buffer during compilation. Then, it performs a byte-pattern match with the known placeholder pointers (recorded during graph capture) within the buffer to identify the correct offsets. As with our earlier optimization, the transformation remains fully transparent to PyTorch programs and to the rest of the PyTorch2 compilation framework.

Figure 8 illustrates this technique. Recall GraCE's aim to convert a kernel parameter (pointer `x`) into a pointer-to-pointer (`px`). During graph capture (§4.2.2), these pointer-to-pointers are recorded as parameters of the prelude kernel. Before each replay of the captured CG, the invocation-specific pointers to parameters are written into the recorded pointer-to-pointer locations (`px`). During replay, the prelude kernel de-references these pointer-to-pointers (① in Figure 8) and writes the resulting values into the parameter fields of the vendor-supplied kernel node (②) using the aforementioned parameter update APIs.

A keen reader would note that this technique can also be used to achieve parameter indirection for JIT compiled kernels. While true, we empirically observed that it is noticeably slower than rewriting the kernel directly during compilation. Figure 9 compares the latency of de-reference through JIT kernel rewrite and through prelude kernel techniques as the number of indirected parameters increases. PI through the prelude kernel is substantially slower and scales poorly, exceeding 10 µs at higher parameter counts. Consequently, we employ JIT-based kernel update where possible, falling back to using prelude kernels only for vendor-supplied kernels.

4.2.2 Graph-Capture Layer

The CUDA Graph generator (④ in Figure 6) captures the compiled module as a CG whenever prior analysis on InductorIR determines that the entire FxGraph can be mapped to a CG. PyTorch2 captures CGs only at the granularity of a whole-FxGraph. Because a CG records kernel parameters (e.g., pointer addresses) by value, the generator creates static placeholders for these parameters and substitutes them for the original parameters before capture.

The graph generator must consider two types of kernel parameters: ① those supplied externally (e.g., application inputs), and ② those produced by preceding kernels within the CG. In the former, the placeholders do not initially contain the correct parameter value for each replay. It must be copied to the placeholders pointed to by the kernel parameters. In the second case, no copy is required, since the producer kernel writes directly into the placeholder. The CUDA Graph generator thus replaces the original parameters with static placeholders. It injects code that would perform the necessary parameter copies into the static placeholder before a kernel's invocation during replay. Finally, it uses the stream capture method (§2) to create a CG for the module.

GraCE's extensions: To take advantage of PI, GraCE enhances the CUDA Graph generator (yellow element in ④ in Figure 6). The enhanced generator receives the list of parameters that had undergone indirection by the code generator. For such parameters, GraCE: ① allocates static placeholders for *pointers* instead of data regions, and ② injects code to perform pointer copies for these parameters instead of parameter (data) copies. Note that these pointer copies are host-to-device (H2D) copies over the PCIe, as the list of pointers to parameters originally resides on the CPU. In contrast, original parameter copies are device-to-device within the GPU. The H2D pointer copy is issued on the same CUDA stream as the subsequent graph replay. Since operations queued in a stream are always executed sequentially, no additional host-device synchronization is needed for correctness. This extension also places the prelude kernel into the captured graph.

4.3 Selective CUDA Graphs

Recall that the overheads of a CUDA Graph can outweigh its benefits (§2). These overheads are due to parameter copies, as well as tasks such as garbage collection for graph replays (§2). While PI can reduce overheads due to parameter copies for kernels in a CG, the rest of the overheads remain.

GraCE introduces a profiler in the compilation phase (slow path) to judiciously decide whether to deploy a CG based on a cost-benefit analysis (colored green, ⑤ in Figure 6). The final outcome of the compilation is an optimized executable (module) corresponding to the given block of PyTorch code. It is then cached for use in the fast path.

PyTorch2 greedily deploys CGs wherever one can be

created. In contrast, GraCE considers up to three possible choices: ① A module *without* CG, ② a module with CG but *without* PI, and ③ a module with CG and PI. The profiler runs each of the available modules as part of the compilation phase, measures their execution time, and caches the module with the best performance.

One may wonder why GraCE distinguishes between ② and ③. The subtlety is that PI replaces device-to-device parameter (data) copies with host-to-device pointer copies. When working with many small objects, it is possible that the overhead of copying pointers over the PCIe outweighs that of copying parameters (data) within the GPU's memory. In addition, GPU memory bandwidth and PCIe transfer bandwidth vary across hardware. Consequently, the benefit from PI varies across hardware. This variability is captured by SCG through profiling conducted on the target hardware. GraCE selectively deploys CG only where it is beneficial.

Summary: GraCE extends Torch Dynamo and Torch Inductor to transparently update IRs for enabling more kernels to execute as part of CG (CGCT). It extends Triton's kernel generator and PyTorch2's CG generator to add indirections to convert data copies in CG to pointer copies (PI). Finally, it adds a profiler in the slow path that uses a cost-benefit analysis to decide when it is beneficial to deploy CG (SCG).

4.4 Applicability to other frameworks

GraCE addresses fundamental limitations arising from the CUDA Graph feature as defined by the hardware vendors. For example, the overhead of parameter copying exists because kernel parameters are recorded by value during graph capture – a constraint of NVIDIA's CUDA Graph feature. Such limitations affect not only PyTorch2 but also other frameworks that attempt to deploy CG. The optimizations proposed in this work, e.g., CUDA Graph-aware IR rewrite, parameter indirection, are also applicable to other frameworks such as JAX/XLA [5]. Importantly, these frameworks, like PyTorch2, embody features such as graph capture, JIT kernel generation, and metadata-rich IR, which enable GraCE's three optimizations to be implemented seamlessly.

Unlike PyTorch2 and JAX/XLA, which target general ML applications, specialized frameworks such as vLLM [33] and SGLang [11] are built atop PyTorch but are tailored for LLMs. For example, LLM's KV cache management is a primary focus of vLLM. In contrast, our goal is to bring the benefits of CG to any ML applications that rely on popular frameworks such as PyTorch. Importantly, vLLM is only partially compatible with PyTorch2's `torch.compile()`. Today, only parts of a model in vLLM can be compiled to benefit from PyTorch2. An ongoing community effort is actively working toward addressing this limitation [19]. In the meantime, vLLM employs a customized handwritten Torch's Inductor backend to leverage CG, albeit only partially [76, 77]. GraCE strives to avoid these custom and temporary fixes to individual frameworks. Once the community enables vLLM to fully use `torch.compile()`, we expect GraCE to benefit frameworks like vLLM automatically.

Suite	Application (Short Name, Type, Batch Size)
TorchBench	Equation of State (EOS, I 1048576), Speech Transformer (ST, I, 1), DALLE2_pytorch (DALLE2, I, 1), Multi-modal Clip (MMC, I, 1), Turbulent Kinetic Energy (TKE, I, 1048576), Deep Recommender (DR-T, T, 256), Deep Recommender (DR-I, I, 8192), Vision MaskRCNN (VM, I, 1), AlexNet (ALNET, I, 2), dlrm (DLRM, I, 2048), TIMM Efficientdet (TEFD, T, 1), dcgan (DCGAN, T, 32), BERT (BERT, T, 16), densenet121 (DNET, I, 64),
HuggingFace	XLNet LM-head Model (XLNet-I, I, 1), XLNet LM-head Model (XLNet-T, T, 8), MT5ForConditionalGeneration (MTCG-T, T, 16), MT5ForConditionalGeneration (MTCG-I, I, 16), MobileBertForQuestionAnswering (MBERT, I, 128), DebertaV2ForQuestionAnswering (DBV2, I, 1), BlenderbotSmallForConditionalGeneration (BSCG, T, 64), DistillGPT2 (DGPT2, T, 16), Speech2Text2ForCausalLM (STCLM, T, 256), DebertaForMaskedLM (DMLM, T, 8)
TIMM	lcnet_050 (LCNET, I, 256)

Table 1: Details of the workloads (T: Training, I: Inference).

5 Evaluation

Our evaluation seeks to answer the following questions:

- How well does GraCE exploit CUDA Graphs relative to the widely used PyTorch2 compiler?
- How much does each optimization in GraCE contribute to end-to-end performance?
- How robustly does GraCE generalize to distributed models and diverse GPUs?

Workloads and metric: We evaluate a broad mix of training and inference workloads from TorchBench [22], HuggingFace [24], and TIMM [81], spanning vision, NLP, and HPC. We focus on applications that are sensitive to CUDA Graph usage; Table 1 summarizes these workloads. Since ML workloads are inherently repetitive, we use per-iteration runtime as our primary performance metric for evaluation. An iteration consists of one forward pass for inference, while one forward and one backward pass for training. All reported numbers are averaged over 100 runs, each using a different input sample at the workload's configured batch size (Table 1).

Environment: We conduct experiments on an NVIDIA H100 NVL GPU with 94GB HBM3, paired with a 64-core Intel Xeon 8462Y+ CPU and 512 GB of DDR5 memory. The software stack includes PyTorch 2.4 [58], CUDA 12.8 [49], cuDNN 8.9.2 [44], and NVIDIA driver 575.57.08. For multi-GPU benchmarks, we use a system with four 80GB H100s connected via high-bandwidth NVLink, running CUDA 12.4.

Baselines: We compare GraCE against two baselines: PyTorch2 *without* CUDA Graphs, denoted PyTorch2-No-CG, and PyTorch2 with CUDA Graphs enabled, denoted

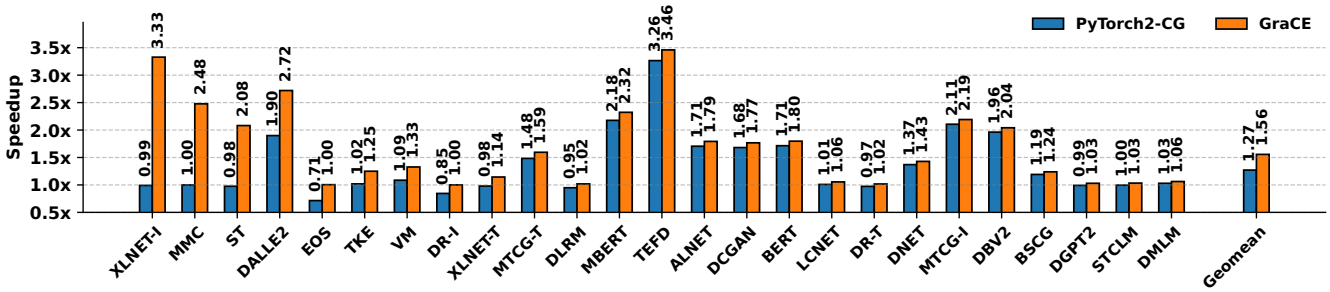


Figure 10: Performance improvement of PyTorch2-CG and GraCE against PyTorch2-No-CG.

PyTorch2-CG. All compilers are based on PyTorch v2.4.

We chose PyTorch2 as the baseline for two reasons: ① Our goal is to seamlessly bring the benefits of CG to a broad class of ML workloads. To the best of our knowledge, frameworks such as PyTorch2 and JAX/XLA are the only open-source ML programming frameworks that can automatically capture CG during compilation and deploy CG seamlessly in the fast path, and ② While it is possible to extend JAX/XLA to implement GraCE, we chose PyTorch2 because it is the most popular GPU backend [6, 17, 35]. In fact, popular LLM-specific frameworks such as vLLM [33] and SGlang [11] build on PyTorch2.

5.1 End-to-end performance

Figure 10 compares GraCE's performance against that of PyTorch2-No-CG and PyTorch2-CG. The figure displays two bars for each application, where the first bar indicates the speedup of PyTorch2-CG over PyTorch2-No-CG, and the second reports the speedup of GraCE over PyTorch2-No-CG.

First, we find that although CUDA Graph was introduced purely as a performance optimization, its current deployment in PyTorch2 can significantly hurt performance. PyTorch2-CG slows down 4 of the 25 applications by at least 3% compared to PyTorch2-No-CG (bars with height < 1), with worst-case regressions of 29% for EOS (HPC) and 15% for Deep Recommender (inference). These workloads consist of short kernels of tens of microseconds each for which the replay overhead (e.g., parameter copying and RNG management) dominates execution time (often around $\sim$ 50%). As a result, blindly deploying all CGs causes net slowdowns, and yet PyTorch2-CG deploys them anyway. In contrast, GraCE is at least as good as the best of PyTorch2-No-CG and PyTorch2-CG for every application. This is important as it absolves a user from having to decide whether to enable CG for each of their programs.

Second, even in cases where PyTorch2-CG improves performance over PyTorch2-No-CG, it still fails to effectively harness CUDA Graph as shown by the difference between bars for PyTorch2-CG and GraCE. GraCE improves performance over PyTorch2-CG by 29% on average and by up to 3.36× (XLNET-inference). While PyTorch2-CG fails to deploy any CUDA Graphs in XLNET-I, GraCE deployed 413 kernels through CGs, thanks to its CG-aware code transformation. In another example, PyTorch2-CG improves DALLE2 (inference) by 1.9× whereas GraCE improves it by 2.72×. In summary, GraCE effectively leverages CGs when there is an opportunity without hurting performance elsewhere.

5.2 Ablation of individual optimizations

Figure 11 quantifies individual contribution of each optimization in GraCE. Each application has three bars corresponding to three optimizations. The height of a bar is normalized to the performance of PyTorch2-No-CG. The total height of the first bar shows the speedup obtained by applying CUDA Graph-aware Code Transformation (CGCT) alone. The bottom stack (dark blue) of the first bar shows the speedup of PyTorch2-CG over PyTorch2-No-CG. The top stack captures the improvement due to CGCT over PyTorch2-CG. The second bar shows performance of PI in tandem with CGCT (+Parameter Indirection). The third bar shows the cumulative effect of three optimizations (+Selective CUDA Graphs).

CUDA Graph-aware Code Transformation. CGCT enables more kernels to be captured within CGs, amortizing the overhead associated with individual kernel launches. As a result, applications such as XLNET-I, MMC, and ST achieve speedups of 3.14×, 2.31×, 2× over PyTorch2-CG, while DALLE2, XLNET-T, and VM improve performance by 43%, 16%, and 11% over PyTorch2-CG, respectively.

Table 2 lists six applications that witness a significant increase in the percentage of their kernels launched as part of CG(s). For example, PyTorch2-CG fails to deploy any CUDA Graph for MMC and XLNET-I. In case of XLNET-I, its FxGraph with 413 kernels was not deployed via CG due to the presence of a *single* CPU tensor in the FxGraph. Similarly, one of the FxGraphs of DALLE2 with 314 kernels was not deployed as a CG due to a *single* memcopy operation from a CPU tensor to a GPU tensor. CGCT removes these constraints, deploying more than 99% of the kernels via CG(s). Even for ST and VM, CGCT increased CG coverage from 5.14% and 58.84% to 74.22% and 71.01%.

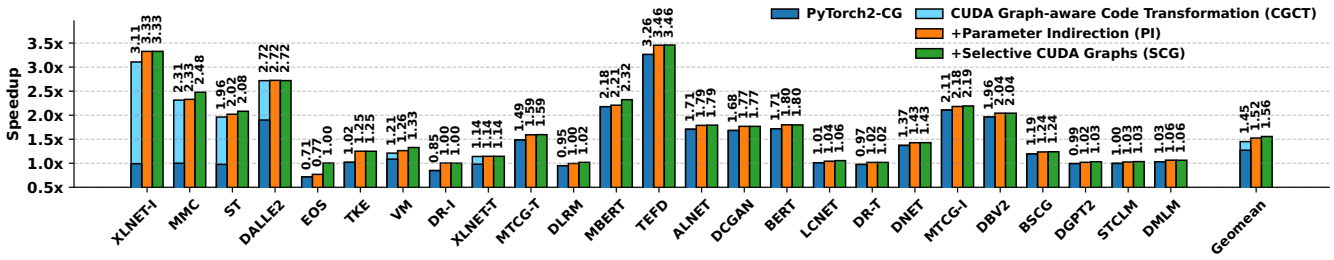


Figure 11: Ablation study on different optimizations of GraCE (performance normalized against PyTorch2-No-CG).

App.	% of Kernels in CUDA Graphs	
	PyTorch2-CG	w/ CUDA Graph-aware Code Transformation
ST	5.14	74.22
DALLE2	79.56	99.32
MMC	0	99.32
XLNET-T	1.53	98.06
XLNET-I	0	99.28
VM	58.84	71.01

Table 2: Enhanced coverage by CUDA Graphs due to CUDA Graph-aware Code Transformation.

App.	Bytes Copy Reduced (# CUDA Graphs)	App.	Bytes Copy Reduced (# CUDA Graphs)
XLNET-I	8.0 KB → 16.0 B (1)	MMC	314.0 KB → 16.0 B (1)
ST	1.3 MB → 336.0 B (3)	DALLE2	4.0 KB → 16.0 B (5)
EOS	4.1 MB → 24.0 B (0)	TKE	174.0 MB → 168.0 B (1)
VM	1.6 MB → 184.0 B (3)	DR-I	3.0 GB → 8.0 B (1)
XLNET-T	362.0 MB → 232.0 B (3)	MTCG-T	1.0 GB → 312.0 B (3)
DLRM	14.6 MB → 136.0 B (1)	MBERT	130.0 KB → 24.0 B (1)
TEFD	6.2 MB → 120.0 B (2)	ALNET	588.0 KB → 8.0 B (1)
DCGAN	1.5 MB → 16.0 B (1)	BERT	156.3 MB → 32.0 B (2)
LCNET	73.5 MB → 8.0 B (1)	DR-T	290.0 MB → 16.0 B (2)
DNET	18.4 MB → 8.0 B (1)	MTCG-I	32.0 KB → 16.0 B (1)
DBV2	4.0 KB → 24.0 B (1)	BSCG	1.6 GB → 56.0 B (30)
DGPT2	953.4 MB → 136.0 B (3)	STCLM	849.5 MB → 136.0 B (14)
DMLM	548.8 MB → 136.0 B (3)		

Table 3: Reduction in data copy and number of CUDA Graphs optimized by GraCE's parameter indirection.

Parameter Indirection. PI reduces overhead of copying kernel parameters while replaying CGs. This speeds up TKE by 23% and DR-I by 18%. Twelve others speed up by more than 4% (EOS, XLNET-I, MTCG-T, TEFD, DCGAN, DLRM, BERT, DR-T, ALNET, DNET, BSCG, VM).

Table 3 reports the reduction in bytes copied by PI. Most workloads see a drastic drop, with the maximum remaining copy size under GraCE at just 336 B. For example, it reduces the bytes copied in MTCG-T, DGPT2, STCLM, and DMLM from 1 GB, 953 MB, 850 MB, and 548 MB, respectively, to just 312 B, 136 B, 136 B, and 136 B, a reduction of over 99%. Table 3 also shows the number of CGs that GraCE optimizes via PI in parentheses.

Note that even after eliminating all data copy, PI may not always benefit the application. For example, for a small tensor, the PCIe transfer cost due to pointer copy may outweigh the benefit of avoiding a larger copy within GPU HBM. These are the cases where a CG does not benefit from PI. For instance, DALLE2 sees no benefit from PI. In contrast, the lone CG in TKE and DR-I gains significantly from reduced copy sizes, yielding speedups of 23% and 18%.

Selective CUDA Graphs. SCG judiciously deploys CGs driven by a cost-benefit analysis. It disables CGs where they could hurt performance, avoiding regression. This is particularly important for applications such as EOS, where blind deployment of CGs degrades performance by 29%.

In five applications, GraCE selectively disables CGs to achieve better overall performance. For instance, VM exposes 21 candidate CGs, but GraCE enables only four and disables the other 17 to improve end-to-end performance by 6%. Similarly, SCG disables the single CG in EOS to avoid performance loss. Overall, GraCE exercises selective deployment by enabling 97 of 123 possible CGs across the applications.

Overall, these results demonstrate that each of GraCE's optimizations makes a meaningful contribution across multiple applications, collectively enhancing overall performance.

5.3 Evaluating distributed ML models

We evaluated the efficacy of GraCE under tensor parallelism (TP) [68], a widely deployed strategy for distributing ML processing in a multi-GPU setting. We evaluated four workloads under TP on a server with 4×H100 GPUs connected over NVLink. Figure 12 summarizes the results: for each application, we report three groups of measurements corresponding to TP-1, TP-2, and TP-4. In each group, we compare the performance of PyTorch2-CG and GraCE against the PyTorch2-No-CG under the given TP setting. This allows us to observe the effectiveness of PyTorch2-CG and GraCE under TP.

Figure 12 demonstrates that GraCE significantly amplifies the benefits of CGs over PyTorch2-CG under TP. All results use an inference batch size of 1, except for XLNET and DALLE2, which use batch sizes of 128 and 2, respectively. As the degree of TP increases, GPU kernels finish faster, and CPU launch latency becomes a larger fraction of overall runtime. This, in turn, increases the importance of fully harnessing CUDA Graphs, which PyTorch2-CG fails to do. In contrast, GraCE yields substantially higher speedups for XLNET, DALLE2, and ST with progressively larger gains

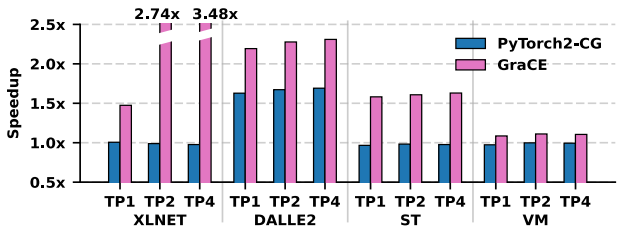


Figure 12: Speedup of PyTorch2-CG and GraCE over PyTorch2-No-CG across tensor parallel settings (TP=1,2,4).

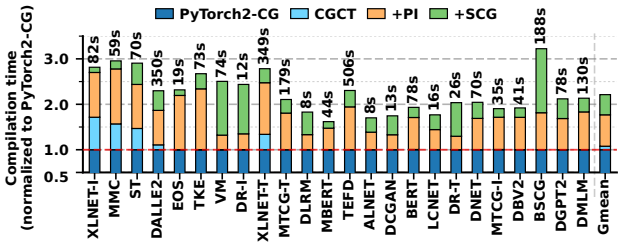


Figure 13: Breakdown of compilation time overheads of GraCE, normalized to PyTorch2-CG. The stacked bars show the cumulative contribution of CGCT, PI, and SCG. Numbers above the bars show absolute compilation time in seconds.

as parallelism increases, as it is able to deploy many kernels in CUDA Graphs. Its speedup reaches up to 3.48× over PyTorch2-No-CG at TP-4 for XLNET with 2.41× on average across different TP settings. This demonstrates that GraCE assumes further importance with increasingly wider deployment of distributed ML processing across multiple GPUs.

Evaluation on A6000 GPU. We also evaluated GraCE on a workstation-class A6000 GPU having less compute power than H100s. As the GPU becomes slower, the headroom for improvement with CGs reduces. Even then, GraCE more prudently utilizes CUDA Graph compared to PyTorch2-CG. For example, PyTorch2-CG degrades performance compared to PyTorch2-No-CG by up to 32% for EOS. In contrast, GraCE never degrades performance and achieves a maximum improvement of 3.25× over PyTorch2-No-CG with an average speedup of 1.18× compared to PyTorch2-CG's average speedup of 1.06 over PyTorch2-No-CG. This shows that GraCE's efficacy is not tied to a specific GPU architecture.

Analyzing overheads of GraCE. GraCE increases compilation time (slow path) for faster execution during graph replay (fast path). Note that PyTorch2 captures the CUDA Graph during compilation. Thus, GraCE's compilation overhead is incurred only once per application and hardware platform. This one-time cost is amortized over repeated execution of the application (fast path).

Figure 13 provides a breakdown of the compilation overhead of GraCE, normalized to PyTorch2-CG (y-axis; lower is better). Each application has a stacked bar. The number at the top of each bar reports the absolute compilation time. Each bar has four stacks. The bottom-most stack (PyTorch2-CG) represents the compilation time of PyTorch2-CG. The three stacks on top of it represent the overheads added by the three optimizations in GraCE, i.e., CGCT, PI, and SCG.

We observe that GraCE increases compilation time by 2.21× on average. In the worst case, the compilation time increased by 3.2× for the BSCG application. For BSCG, the increase comes from profiling a large number of candidate modules for CG deployment as part of SCG. On the other hand, applications such as TKE and MMC have many Triton-generated kernels that take longer to regenerate due to PI. Finally, CGCT contributes significantly to the compilation time of XLNET-I. It undergoes many TorchIR metadata updates and re-lowering to InductorIR to remove CPU-tensor outputs from InductorIR, enabling CG deployment. The remaining applications incur increased compilation time due to similar reasons. Overall, the compilation takes up to 506 seconds (highest for TEFD). For TEFD, PyTorch2 took 220 seconds to compile. In general, the increased time is a reasonable one-time cost for compilation and graph capture.

GraCE increases peak memory footprint during compilation by 12% on average. Like compilation time, this is also a one-time cost, though. The overhead arises from profiling multiple CG deployment choices in SCG. This temporarily retains captured tensors and allocation metadata in memory. However, GraCE does *not* introduce any memory overhead during replays (fast path). On the contrary, it actually reduces peak memory footprint during replays by up to 15% compared to PyTorch2-CG.

6 Related Work

ML compilers: Early ML compilers such as TensorFlow [1], Caffe [28], Theano [75], and CNTK [64] improve performance via graph-level optimizations, while later systems develop richer tensor-program IRs [4, 7, 12, 16, 21, 36, 52, 65, 78, 84, 89, 91–94] and target efficient code generation across diverse hardware and irregular workloads [15, 66, 73, 86]. Recent advances introduce whole-graph and memory-centric optimizations [67, 83], accelerate dynamic control flow [88], and perform multi-level super-optimization using unified IRs [82]. These efforts reduce kernel count and/or improve efficiency of kernels. GraCE complements them by mitigating CPU launch overhead and enhancing CUDA Graph's deployability.

PyTorch2 [3] showed that dynamic Python execution can coexist with graph compilation by using Torch Dynamo [57] to rewrite Python bytecode into FxGraphs [63] and Torch Inductor [25] to lower them into optimized GPU kernels. Earlier capture mechanisms–JIT tracing, TorchScript [60], lazy tensors [59, 72], and symbolic tracing [63]–were limited by unsoundness, incomplete Python coverage, or high recompilation overheads. GraCE is not tied to PyTorch2's

pipeline; it only assumes a compiler that produces a stable kernel-launch DAG and augments the backend that prepares and executes kernels. GraCE's approach can also be integrated with JAX [5] and XLA [53]. Source-level graph-repair techniques [32] are complementary to GraCE. They lengthen computation graphs, while GraCE ensures the resulting kernel DAGs are effectively captured as CUDA Graphs.

CUDA Graphs: Grape [90] is a close prior work, optimizing CUDA Graphs for DNNs through predication rewriting, metadata compression, and alias prediction, but these techniques require extensive manual kernel and CUDA-driver changes – e.g., over 400 lines of kernel edits for GPT beam search – and its alias predictor is incompatible with PyTorch2. In contrast, GraCE requires *no* manual modifications and works transparently for all PyTorch applications. The extensive manual changes required by Grape and its tight integration with PyTorch 1.12 make a quantitative comparison infeasible. Other systems use CUDA Graphs in specialized deployments, such as serverless LLM inference, by materializing and restoring pre-captured graphs [87]. These efforts focus on a graph's persistence, whereas GraCE enables broader and more efficient deployment of CUDA Graph and thus, is complementary.

7 Conclusion

We present GraCE — a compiler framework designed to enhance the deployment and performance of CUDA Graphs. GraCE addresses several key limitations of the PyTorch2 compiler, leading to substantial performance improvement across various ML benchmarks without requiring manual modifications to applications.

Acknowledgment

We thank anonymous reviewers and the shepherd for their constructive feedback. This work is partially supported by a research grant from Microsoft Research India. Ajay is supported by the Prime Minister's Fellowship Scheme for Doctoral Research, co-sponsored by the Confederation of Indian Industry, the Government of India, and Microsoft Research India.

References

[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. Tensorflow: a system for large-scale machine learning. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI'16, page 265–283, USA, 2016. USENIX Association.

[2] Amey Agrawal, Nitin Kedia, Ashish Panwar, Jayashree Mohan, Nipun Kwatra, Bhargav Gulavani, Alexey Tumanov, and Ramachandran Ramjee. Taming Throughput-Latency tradeoff in LLM inference with Sarathi-Serve. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, pages 117–134, Santa Clara, CA, July 2024. USENIX Association.

[3] Jason Ansel, Edward Yang, Horace He, Natalia Gimelshein, Animesh Jain, Michael Voznesensky, Bin Bao, Peter Bell, David Berard, Evgeni Burovski, Geeta Chauhan, Anjali Chourdia, Will Constable, Alban Desmaison, Zachary DeVito, Elias Ellison, Will Feng, Jiong Gong, Michael Gschwind, Brian Hirsh, Sherlock Huang, Kshiteej Kalambarkar, Laurent Kirsch, Michael Lazos, Mario Lezcano, Yanbo Liang, Jason Liang, Yinghai Lu, C. K. Luk, Bert Maher, Yunjie Pan, Christian Puhrsch, Matthias Reso, Mark Saroufim, Marcos Yukio Siraichi, Helen Suk, Shunting Zhang, Michael Suo, Phil Tillet, Xu Zhao, Eikan Wang, Keren Zhou, Richard Zou, Xiaodong Wang, Ajit Mathews, William Wen, Gregory Chanan, Peng Wu, and Soumith Chintala. Pytorch 2: Faster machine learning through dynamic python bytecode transformation and graph compilation. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, 2024.

[4] Riyadh Baghdadi, Jessica Ray, Malek Ben Romdhane, Emanuele Del Sozzo, Abdurrahman Akkas, Yunming Zhang, Patricia Suriana, Shoaib Kamil, and Saman Amarasinghe. Tiramisu: A polyhedral compiler for expressing fast and portable code. In *2019 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 193–205, 2019.

[5] James Bradbury, Roy Frostig, Peter Hawkins, Matthew James Johnson, Chris Leary, Dougal Maclaurin, George Necula, Adam Paszke, Jake VanderPlas, Skye Wanderman-Milne, and Qiao Zhang. JAX: composable transformations of Python+NumPy programs, 2018.

[6] Harald Carlens. State of machine learning competitions in 2024. *ML Contests Research*, 2025. https://mlcontests.com/state-of-machine-learning-competitions-2024.

[7] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Meghan Cowan, Haichen Shen, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin,

and Arvind Krishnamurthy. Tvm: an automated end-to-end optimizing compiler for deep learning. In *Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation*, OSDI'18, page 579–594, USA, 2018. USENIX Association.

[8] Seungbeom Choi, Sunho Lee, Yeonjae Kim, Jongse Park, Youngjin Kwon, and Jaehyuk Huh. Serving heterogeneous machine learning models on Multi-GPU servers with Spatio-Temporal sharing. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, pages 199–216, Carlsbad, CA, July 2022. USENIX Association.

[9] Arnab Choudhury, Yang Wang, Tuomas Pelkonen, Kutta Srinivasan, Abha Jain, Shenghao Lin, Delia David, Siavash Soleimanifard, Michael Chen, Abhishek Yadav, Ritesh Tijoriwala, Denis Samoylov, and Chunqiang Tang. MAST: Global scheduling of ML training across Geo-Distributed datacenters at hyperscale. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, pages 563–580, Santa Clara, CA, July 2024. USENIX Association.

[10] LLVM developer group. The llvm compiler infrastructure. https://llvm.org/, 2024.

[11] SGLang developers. SGLang: high-performance serving framework for large language models and multimodal models. https://github.com/sgl-project/sglang, 2026.

[12] Yaoyao Ding, Cody Hao Yu, Bojian Zheng, Yizhi Liu, Yida Wang, and Gennady Pekhimenko. Hidet: Task-mapping programming paradigm for deep learning tensor programs. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, ASPLOS 2023, page 370–384, New York, NY, USA, 2023. Association for Computing Machinery.

[13] dscerutti and NVIDIA. Any way to measure the latency of a kernel launch? https://forums.developer.nvidia.com/t/any-way-to-measure-the-latency-of-a-kernel-launch/221413, 2022.

[14] Babak Falsafi. Post-moore server architecture. In *Proceedings of the 34th ACM International Conference on Supercomputing*, ICS '20, New York, NY, USA, 2020. Association for Computing Machinery.

[15] Pratik Fegade, Tianqi Chen, Phillip B. Gibbons, and Todd C. Mowry. The cora tensor compiler: Compilation for ragged tensors with minimal padding, 2022.

[16] Siyuan Feng, Bohan Hou, Hongyi Jin, Wuwei Lin, Junru Shao, Ruihang Lai, Zihao Ye, Lianmin Zheng, Cody Hao Yu, Yong Yu, and Tianqi Chen. Tensorir: An abstraction for automatic tensorized program optimization. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, ASPLOS 2023, page 804–817, New York, NY, USA, 2023. Association for Computing Machinery.

[17] Linux Foundation. Shaping the future of generative ai. https://www.linuxfoundation.org/hubfs/LF%20Research/lfr_genai24_111924.pdf?hsLang=en, 2024.

[18] Yanjie Gao, Yichen He, Xinze Li, Bo Zhao, Haoxiang Lin, Yoyo Liang, Jing Zhong, Hongyu Zhang, Jingzhou Wang, Yonghua Zeng, Keli Gui, Jie Tong, and Mao Yang. An empirical study on low gpu utilization of deep learning jobs. In *ICSE 2024*. IEEE/ACM, April 2024. The 46th International Conference on Software Engineering.

[19] Luka Govedič, Richard Zou, Addie Stevens, Kaichao You, Michael Goin, and Saša Zelenović. Introduction to torch.compile and how it works with vllm. https://vllm-project.github.io/2025/08/20/torch-compile.html, 2025.

[20] Alan Gray. Getting started with cuda graphs. https://developer.nvidia.com/blog/cuda-graphs/, 2019.

[21] Bastian Hagedorn, Bin Fan, Hanfeng Chen, Cris Cecka, Michael Garland, and Vinod Grover. Graphene: An ir for optimized tensor computations on gpus. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, ASPLOS 2023, page 302–313, New York, NY, USA, 2023. Association for Computing Machinery.

[22] Yueming Hao, Xu Zhao, Bin Bao, David Berard, Will Constable, Adnan Aziz, and Xu Liu. Torchbench: Benchmarking pytorch with high api surface coverage, 2023.

[23] Kaiming He, Georgia Gkioxari, Piotr Dollár, and Ross Girshick. Mask r-cnn, 2018.

[24] HuggingFace. Transformers. https://github.com/huggingface/transformers.

[25] jansel. Torchinductor: a pytorch-native compiler with define-by-run ir and symbolic shapes. https://dev-discuss.pytorch.org/t/torchinductor-a-pytorch-native-compiler-with-define-by-run-ir-and-symbolic-shapes/747, 2024.

[26] Myeongjae Jeon, Shivaram Venkataraman, Amar Phanishayee, Junjie Qian, Wencong Xiao, and Fan Yang. Analysis of Large-Scale Multi-Tenant GPU clusters for DNN training workloads. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 947–960, Renton, WA, July 2019. USENIX Association.

[27] Appleyard Jeremy and Yokim Scott. Programming tensor cores in cuda 9. https://developer.nvidia.com/blog/programming-tensor-cores-cuda-9/, 2017.

[28] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. Caffe: Convolutional architecture for fast feature embedding, 2014.

[29] Aditya K. Kamath and Arkaprava Basu. iguard: In-gpu advanced race detection. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, SOSP '21, page 49–65, New York, NY, USA, 2021. Association for Computing Machinery.

[30] Aditya K. Kamath, Alvin A. George, and Arkaprava Basu. Scord: A scoped race detector for gpus. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, pages 1036–1049, 2020.

[31] Khandelwal Kartikay and De Ankita. Introducing torchmultimodal – a library for accelerating exploration in multimodal ai. https://pytorch.org/blog/introducing-torchmultimodal/, 2022.

[32] Savini Kashmira, Jayanaka Dantanarayana, Thamirawaran Sathiyalogeswaran, Yichao Yuan, Nishil Talati, Krisztian Flautner, Lingjia Tang, and Jason Mars. Graphmend: Code transformations for fixing graph breaks in pytorch 2, 2025.

[33] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph E. Gonzalez, Hao Zhang, and Ion Stoica. Efficient memory management for large language model serving with pagedattention. In *Proceedings of the ACM SIGOPS 29th Symposium on Operating Systems Principles*, 2023.

[34] Wonbeom Lee, Jungi Lee, Junghwan Seo, and Jaewoong Sim. InfiniGen: Efficient generative inference of large language models with dynamic KV cache management. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, pages 155–172, Santa Clara, CA, July 2024. USENIX Association.

[35] Matt Li. Pytorch vs tensorflow: Usage, popularity and performance in 2026. https://www.secondtalent.com/resources/pytorch-vs-tensorflow-usage-popularity-and-performance/, 2026.

[36] Tzu-Mao Li, Michaël Gharbi, Andrew Adams, Frédo Durand, and Jonathan Ragan-Kelley. Differentiable programming for image processing and deep learning in halide. *ACM Trans. Graph.*, 37(4), July 2018.

[37] Chaofan Lin, Zhenhua Han, Chengruidong Zhang, Yuqing Yang, Fan Yang, Chen Chen, and Lili Qiu. Parrot: Efficient serving of LLM-based applications with semantic variable. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, pages 929–945, Santa Clara, CA, July 2024. USENIX Association.

[38] Paulius Micikevicius, Sharan Narang, Jonah Alben, Gregory Diamos, Erich Elsen, David Garcia, Boris Ginsburg, Michael Houston, Oleksii Kuchaiev, Ganesh Venkatesh, and Hao Wu. Mixed precision training. In *International Conference on Learning Representations*, 2018.

[39] Ajay Nayak and Arkaprava Basu. Over-synchronization in gpu programs. In *2024 57th IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 795–809, 2024.

[40] NVIDIA. Nvidia a100 tensor core gpu: Unprecedented acceleration at every scale. https://www.nvidia.com/en-in/data-center/a100/, 2022.

[41] NVIDIA. Basic linear algebra on nvidia gpus. https://developer.nvidia.com/cublas, 2024.

[42] NVIDIA. Cutlass: Cuda c++ template abstractions for implementing high-performance matrix-matrix multiplication (gemm). https://github.com/NVIDIA/cutlass, 2024.

[43] NVIDIA. Nvidia cuda deep neural network library. https://developer.nvidia.com/cudnn, 2024.

[44] NVIDIA. Nvidia cudnn documentation. https://docs.nvidia.com/deeplearning/cudnn/archives/cudnn-892/index.html, 2024.

[45] NVIDIA. Nvidia h100 gpu:extraordinary performance, scalability, and security for every data center. https://www.nvidia.com/en-in/data-center/h100/, 2024.

[46] NVIDIA. Parallel thread execution isa. https://docs.nvidia.com/cuda/parallel-thread-execution/, 2024.

[47] NVIDIA. CUDA Graph Management APIs. https://docs.nvidia.com/cuda/cuda-runtime-api/group__CUDART__GRAPH.html, 2025.

[48] NVIDIA. Cuda programming guide: Cuda graphs. https://docs.nvidia.com/cuda/cuda-programming-guide/04-special-topics/cuda-graphs.html, 2025.

[49] NVIDIA. Cuda toolkit. https://developer.nvidia.com/cuda/toolkit, 2025.

[50] NVIDIA. Nvidia blackwell: The engine of the new industrial revolution. https://resources.nvidia.com/en-us-blackwell-architecture/datasheet, 2025.

[51] NVIDIA. Nvrtc. https://docs.nvidia.com/cuda/nvrtc/index.html, 2025.

[52] OpenAI. Triton: Open-source gpu programming for neural networks. https://openai.com/index/triton/, 2024.

[53] openxla. Xla (accelerated linear algebra). https://github.com/openxla/xla.

[54] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Köpf, Edward Yang, Zach DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library, 2019.

[55] PyTorch. speech transformer model in torchbenchmark. https://github.com/pytorch/benchmark/tree/main/torchbenchmark/models/speech_transformer.

[56] PyTorch. Core aten ir and prims ir. https://pytorch.org/docs/stable/torch.compiler_ir.html, 2024.

[57] PyTorch. Dynamo overview. https://pytorch.org/docs/stable/torch.compiler_dynamo_overview.html, 2024.

[58] PyTorch. Pytorch 2024-04-23 nightly release. https://github.com/pytorch/pytorch/commit/6d1c678c58d4fb9468aca3daac67ae7bcec8495d, 2024.

[59] PyTorch. Pytorch/xla. https://github.com/pytorch/xla, 2024.

[60] PyTorch. Torchscript. https://pytorch.org/docs/main/jit.html, 2024.

[61] Alec Radford, Jong Wook Kim, Chris Hallacy, Aditya Ramesh, Gabriel Goh, Sandhini Agarwal, Girish Sastry, Amanda Askell, Pamela Mishkin, Jack Clark, Gretchen Krueger, and Ilya Sutskever. Learning transferable visual models from natural language supervision, 2021.

[62] Aditya Ramesh, Prafulla Dhariwal, Alex Nichol, Casey Chu, and Mark Chen. Hierarchical text-conditional image generation with clip latents, 2022.

[63] James K. Reed, Zachary DeVito, Horace He, Ansley Ussery, and Jason Ansel. Torch.fx: Practical program capture and transformation for deep learning in python, 2022.

[64] Frank Seide and Amit Agarwal. Cntk: Microsoft's open-source deep-learning toolkit. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '16, page 2135, New York, NY, USA, 2016. Association for Computing Machinery.

[65] Junru Shao, Xiyou Zhou, Siyuan Feng, Bohan Hou, Ruihang Lai, Hongyi Jin, Wuwei Lin, Masahiro Masuda, Cody Hao Yu, and Tianqi Chen. Tensor program optimization with probabilistic programs. In Alice H. Oh, Alekh Agarwal, Danielle Belgrave, and Kyunghyun Cho, editors, *Advances in Neural Information Processing Systems*, 2022.

[66] Haichen Shen, Jared Roesch, Zhi Chen, Wei Chen, Yong Wu, Mu Li, Vin Sharma, Zachary Tatlock, and Yida Wang. Nimble: Efficiently compiling dynamic neural networks for model inference. *ArXiv*, abs/2006.03031, 2020.

[67] Yining Shi, Zhi Yang, Jilong Xue, Lingxiao Ma, Yuqing Xia, Ziming Miao, Yuxiao Guo, Fan Yang, and Lidong Zhou. Welder: Scheduling deep learning memory access via tile-graph. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*, pages 701–718, Boston, MA, July 2023. USENIX Association.

[68] Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. Megatron-lm: Training multi-billion parameter language models using model parallelism, 2020.

[69] Sudipta Saha Shubha, Haiying Shen, and Anand Iyer. USHER: Holistic interference avoidance for resource optimized ML inference. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, pages 947–964, Santa Clara, CA, July 2024. USENIX Association.

[70] Foteini Strati, Paul Elvinger, Tolga Kerimoglu, and Ana Klimovic. Ml training with cloud gpu shortages: Is cross-region the answer? In *Proceedings of the 4th Workshop on Machine Learning and Systems*, EuroMLSys '24, page 107–116, New York, NY, USA, 2024. Association for Computing Machinery.

[71] Foteini Strati, Xianzhe Ma, and Ana Klimovic. Orion: Interference-aware, fine-grained gpu sharing for ml applications. In *Proceedings of the Nineteenth European Conference on Computer Systems*, EuroSys '24, page

1075–1092, New York, NY, USA, 2024. Association for Computing Machinery.

[72] Alex Suhan, Davide Libenzi, Ailing Zhang, Parker Schuh, Brennan Saeta, Jie Young Sohn, and Denys Shabalin. Lazytensor: combining eager execution with domain-specific compilers, 2021.

[73] Shizhi Tang, Jidong Zhai, Haojie Wang, Lin Jiang, Liyan Zheng, Zhenhao Yuan, and Chen Zhang. Freetensor: a free-form dsl with holistic optimizations for irregular tensor programs. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, PLDI 2022, page 872–887, New York, NY, USA, 2022. Association for Computing Machinery.

[74] NumPy Team. Numpy: The fundamental package for scientific computing with python. https://numpy.org/, 2025.

[75] The Theano Development Team, Rami Al-Rfou, Guillaume Alain, Amjad Almahairi, Christof Angermueller, Dzmitry Bahdanau, Nicolas Ballas, Frédéric Bastien, Justin Bayer, Anatoly Belikov, Alexander Belopolsky, Yoshua Bengio, Arnaud Bergeron, James Bergstra, Valentin Bisson, Josh Bleecher Snyder, Nicolas Bouchard, Nicolas Boulanger-Lewandowski, Xavier Bouthillier, Alexandre de Brébisson, Olivier Breuleux, Pierre-Luc Carrier, Kyunghyun Cho, Jan Chorowski, Paul Christiano, Tim Cooijmans, Marc-Alexandre Côté, Myriam Côté, Aaron Courville, Yann N. Dauphin, Olivier Delalleau, Julien Demouth, Guillaume Desjardins, Sander Dieleman, Laurent Dinh, Mélanie Ducoffe, Vincent Dumoulin, Samira Ebrahimi Kahou, Dumitru Erhan, Ziye Fan, Orhan Firat, Mathieu Germain, Xavier Glorot, Ian Goodfellow, Matt Graham, Caglar Gulcehre, Philippe Hamel, Iban Harlouchet, Jean-Philippe Heng, Balázs Hidasi, Sina Honari, Arjun Jain, Sébastien Jean, Kai Jia, Mikhail Korobov, Vivek Kulkarni, Alex Lamb, Pascal Lamblin, Eric Larsen, César Laurent, Sean Lee, Simon Lefrancois, Simon Lemieux, Nicholas Léonard, Zhouhan Lin, Jesse A. Livezey, Cory Lorenz, Jeremiah Lowin, Qianli Ma, Pierre-Antoine Manzagol, Olivier Mastropietro, Robert T. McGibbon, Roland Memisevic, Bart van Merriënboer, Vincent Michalski, Mehdi Mirza, Alberto Orlandi, Christopher Pal, Razvan Pascanu, Mohammad Pezeshki, Colin Raffel, Daniel Renshaw, Matthew Rocklin, Adriana Romero, Markus Roth, Peter Sadowski, John Salvatier, François Savard, Jan Schlüter, John Schulman, Gabriel Schwartz, Iulian Vlad Serban, Dmitriy Serdyuk, Samira Shabanian, Étienne Simon, Sigurd Spieckermann, S. Ramana Subramanyam, Jakub Sygnowski, Jérémie Tanguay, Gijs van Tulder, Joseph Turian, Sebastian Urban, Pascal Vincent, Francesco Visin, Harm de Vries, David Warde-Farley, Dustin J. Webb, Matthew Willson, Kelvin Xu, Lijun Xue, Li Yao, Saizheng Zhang, and Ying Zhang. Theano: A python framework for fast computation of mathematical expressions, 2016.

[76] vLLM developers. Cuda graphs compatibility of attention backends. https://docs.vllm.ai/en/stable/design/cuda_graphs/#cuda-graphs-compatibility-of-attention-backends, 2025.

[77] vLLM developers. How to debug the vllm-torch.compile integration. https://docs.vllm.ai/en/stable/design/debug_vllm_compile/, 2025.

[78] Jian Weng, Animesh Jain, Jie Wang, Leyuan Wang, Yida Wang, and Tony Nowatzki. Unit: Unifying tensorized instruction compilation. In *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 77–89, 2021.

[79] Qizhen Weng, Wencong Xiao, Yinghao Yu, Wei Wang, Cheng Wang, Jian He, Yong Li, Liping Zhang, Wei Lin, and Yu Ding. MLaaS in the wild: Workload analysis and scheduling in Large-Scale heterogeneous GPU clusters. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, pages 945–960, Renton, WA, April 2022. USENIX Association.

[80] Qizhen Weng, Lingyun Yang, Yinghao Yu, Wei Wang, Xiaochuan Tang, Guodong Yang, and Liping Zhang. Beware of fragmentation: Scheduling GPU-Sharing workloads with fragmentation gradient descent. In *2023 USENIX Annual Technical Conference (USENIX ATC 23)*, pages 995–1008, Boston, MA, July 2023. USENIX Association.

[81] Ross Wightman. Pytorch image models. https://github.com/rwightman/pytorch-image-models, 2019.

[82] Mengdi Wu, Xinhao Cheng, Shengyu Liu, Chunan Shi, Jianan Ji, Man Kit Ao, Praveen Velliengiri, Xupeng Miao, Oded Padon, and Zhihao Jia. Mirage: a multi-level superoptimizer for tensor programs. In *Proceedings of the 19th USENIX Conference on Operating Systems Design and Implementation*, OSDI '25, USA, 2025. USENIX Association.

[83] Chunwei Xia, Jiacheng Zhao, Qianqi Sun, Zheng Wang, Yuan Wen, Teng Yu, Xiaobing Feng, and Huimin Cui. Optimizing deep learning inference via global analysis and tensor expressions. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1*, ASPLOS '24, page 286–301, New York, NY, USA, 2024. Association for Computing Machinery.

[84] Jiarong Xing, Leyuan Wang, Shang Zhang, Jack Chen, Ang Chen, and Yibo Zhu. Bolt: Bridging the gap between auto-tuners and hardware-native performance. In D. Marculescu, Y. Chi, and C. Wu, editors, *Proceedings of Machine Learning and Systems*, volume 4, pages 204–216, 2022.

[85] Zhilin Yang, Zihang Dai, Yiming Yang, Jaime Carbonell, Ruslan Salakhutdinov, and Quoc V. Le. Xlnet: Generalized autoregressive pretraining for language understanding, 2020.

[86] Zihao Ye, Ruihang Lai, Junru Shao, Tianqi Chen, and Luis Ceze. Sparsetir: Composable abstractions for sparse compilation in deep learning. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, ASPLOS 2023, page 660–678, New York, NY, USA, 2023. Association for Computing Machinery.

[87] Shaoxun Zeng, Minhui Xie, Shiwei Gao, Youmin Chen, and Youyou Lu. Medusa: Accelerating serverless llm inference with materialization. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1*, ASPLOS '25, page 653–668, New York, NY, USA, 2025. Association for Computing Machinery.

[88] Chen Zhang, Lingxiao Ma, Jilong Xue, Yining Shi, Ziming Miao, Fan Yang, Jidong Zhai, Zhi Yang, and Mao Yang. Cocktailer: Analyzing and optimizing dynamic control flow in deep learning. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*, pages 681–699, Boston, MA, July 2023. USENIX Association.

[89] Jie Zhao, Bojie Li, Wang Nie, Zhen Geng, Renwei Zhang, Xiong Gao, Bin Cheng, Chen Wu, Yun Cheng, Zheng Li, Peng Di, Kun Zhang, and Xuefeng Jin. Akg: automatic kernel generation for neural processing units using polyhedral transformations. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, PLDI 2021, page 1233–1248, New York, NY, USA, 2021. Association for Computing Machinery.

[90] Bojian Zheng, Cody Hao Yu, Jie Wang, Yaoyao Ding, Yizhi Liu, Yida Wang, and Gennady Pekhimenko. Grape: Practical and efficient graphed execution for dynamic deep neural networks on gpus. In *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '23, page 1364–1380, New York, NY, USA, 2023. Association for Computing Machinery.

[91] Lianmin Zheng, Chengfan Jia, Minmin Sun, Zhao Wu, Cody Hao Yu, Ameer Haj-Ali, Yida Wang, Jun Yang, Danyang Zhuo, Koushik Sen, Joseph E. Gonzalez, and Ion Stoica. Ansor: generating high-performance tensor programs for deep learning. In *Proceedings of the 14th USENIX Conference on Operating Systems Design and Implementation*, OSDI'20, USA, 2020. USENIX Association.

[92] Size Zheng, Renze Chen, Anjiang Wei, Yicheng Jin, Qin Han, Liqiang Lu, Bingyang Wu, Xiuhong Li, Shengen Yan, and Yun Liang. Amos: enabling automatic mapping for tensor computations on spatial accelerators with hardware abstraction. In *Proceedings of the 49th Annual International Symposium on Computer Architecture*, ISCA '22, page 874–887, New York, NY, USA, 2022. Association for Computing Machinery.

[93] Size Zheng, Yun Liang, Shuo Wang, Renze Chen, and Kaiwen Sheng. Flextensor: An automatic schedule exploration and optimization framework for tensor computation on heterogeneous system. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '20, page 859–873, New York, NY, USA, 2020. Association for Computing Machinery.

[94] Hongyu Zhu, Ruofan Wu, Yijia Diao, Shanbin Ke, Haoyu Li, Chen Zhang, Jilong Xue, Lingxiao Ma, Yuqing Xia, Wei Cui, Fan Yang, Mao Yang, Lidong Zhou, Asaf Cidon, and Gennady Pekhimenko. ROLLER: Fast and efficient tensor compilation for deep learning. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 233–248, Carlsbad, CA, July 2022. USENIX Association.

[95] Donglin Zhuang, Zhen Zheng, Haojun Xia, Xiafei Qiu, Junjie Bai, Wei Lin, and Shuaiwen Leon Song. MonoNN: Enabling a new monolithic optimization space for neural network inference tasks on modern GPU-Centric architectures. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, pages 989–1005, Santa Clara, CA, July 2024. USENIX Association.

A Artifact Appendix

Abstract

The artifact contains the GraCE implementation, Docker environment, benchmark drivers, workload configurations, and analysis scripts for reproducing the main results in the paper. The recommended path is to use the prebuilt Docker image, run the smoke test, and then run the provided scripts to regenerate the figures and tables.

Scope

The artifact supports validation of the following claims:

- GraCE improves CUDA Graph coverage compared with the PyTorch2-CG baseline on selected deep learning workloads.
- GraCE reduces data-copy movement required by CUDA Graph execution.
- GraCE improves single-GPU performance on selected TorchBench, HuggingFace, and TIMM workloads.
- GraCE improves performance on selected tensor-parallel workloads.
- The reported figures and tables can be regenerated using the provided workload lists, scripts, Docker images, and analysis pipeline.

The artifact is intended for artifact evaluation and reproducibility. Exact performance values may vary across machines because CUDA Graph experiments are sensitive to CPU-side launch overheads, CPU frequency scaling, deep idle states, GPU clocks, NUMA placement, and background host activity. The artifact repository therefore includes optional host-tuning scripts for low-noise benchmarking.

Contents

The artifact repository is organized as follows:

`benchmark_runner/`	Benchmark launcher
`configs/`	Workload lists
`docker/`	Docker run/build scripts
`host_tuning/`	Optional low-noise host tuning
`scripts/`	Experiments and analysis
`third_party/`	GraCE source modules
`results/`	Raw and processed outputs
`figures/, tables/`	Generated paper artifacts

The main source submodules are:

- `third_party/pytorch`: modified PyTorch implementation used by GraCE. This submodule contains separate `osdi26/grace-*` branches corresponding to the incremental GraCE optimizations used in the artifact.
- `third_party/triton`: Triton changes needed by GraCE, primarily on the `osdi26/grace-indirection` branch.
- `third_party/torchbenchmark`: benchmark-suite used by the evaluation.

Generated results are written under `results/`, figures under `figures/`, and tables under `tables/`.

Hosting

The artifact is hosted on GitHub: https://github.com/csl-iisc/GraCE-OSDI26-Artifact on the `master` branch. The prebuilt Docker image is hosted on Docker Hub: `abhishekghosh1998/grace-osdi26:cuda128-prebuilt`. Detailed setup, experiments and analysis instructions are provided in the artifact repository README.

Requirements

The artifact is Docker-based. The host system must have:

- Docker Engine.
- NVIDIA GPU driver.
- NVIDIA Container Toolkit for Docker GPU passthrough.
- One CUDA-capable NVIDIA GPU for smoke tests and single-GPU experiments; four visible GPUs are needed to reproduce the tensor-parallel experiments.

A full CUDA Toolkit installation is not required on the host for the prebuilt Docker workflow. CUDA user-space libraries, Python packages, PyTorch, Triton, TorchBench dependencies, and analysis tools are included in the Docker image. The host only needs an NVIDIA driver compatible with the CUDA runtime used by the selected image.

The paper uses two hardware setups. Single-GPU experiments were collected on an NVIDIA H100 NVL GPU with 94GB HBM3 and CUDA 12.8, paired with a 64-core Intel Xeon Platinum 8462Y+ CPU, 128 logical CPUs with hyperthreading, and 512GB DDR5 memory. Tensor-parallel experiments were collected on a separate four-GPU H100 system with high-bandwidth NVLink and CUDA 12.4.

The artifact can run on other recent x86-64 Linux systems with NVIDIA GPUs, but performance is hardware- and system-dependent. H100-class GPUs and the optional host-tuning workflow are recommended for closest reproduction. Tensor-parallel runs require at least as many visible GPUs as the selected tensor-parallel degree.

The host system should have at least 150GB of free disk space for the prebuilt Docker image. Building the prebuilt image from source takes roughly 2–3 hours on an H100-class host; the full experiment workflow should be treated as a long-running job, with a safe end-to-end allocation of about 24 hours.

Getting Started

The 30-minute validation path checks that the repository, Docker image, CUDA stack, conda environments, benchmark runner, and result-file writing path are functional.

To obtain the artifact:

```
git clone \
https://github.com/csl-iisc/GraCE-OSDI26-Artifact \
    grace-osdi26-artifact
cd grace-osdi26-artifact
git submodule sync --recursive
git submodule update --init --recursive
```

To pull the default prebuilt Docker image:

```
docker pull \
    abhishekghosh1998/grace-osdi26:cuda128-prebuilt
docker tag \
    abhishekghosh1998/grace-osdi26:cuda128-prebuilt \
    grace-osdi26:cuda128-prebuilt
```

From the host:

```
docker/run_cuda128.sh
```

Inside the Docker container:

```
bash scripts/smoke_test.sh
```

The expected final smoke-test output is:

```
[OK] Workload run completed.
```

Experiment Workflow

All commands in this section are run inside the Docker container unless stated otherwise.

Activate a conda environment available in the container:

```
conda activate grace-full
```

For single-GPU experiments (Figures 10–11):

```
bash scripts/run_single_gpu_experiments.sh
bash scripts/generate_figure10.sh
bash scripts/generate_figure11.sh
```

Expected outputs:

```
figures/figure10.pdf
figures/figure11.pdf
```

For tensor-parallel experiments (Figure 12):

```
bash scripts/run_tp_experiments.sh
bash scripts/generate_tp_figure.sh
```

Expected output:

```
figures/figure12.pdf
```

For Table 2, CUDA Graph coverage:

```
bash scripts/run_table2_cgct_coverage_nsys.sh
bash scripts/generate_table2_cgct_coverage.sh
```

Expected output:

```
tables/table2_cgct_coverage.csv
```

This step requires Nsight Systems inside the Docker image.

For Table 3, data-copy movement:

```
bash scripts/run_table3_pi_copy_debug.sh
bash scripts/generate_table3_pi_copy_debug.sh
```

Expected output:

```
tables/table3_pi_copy_debug.csv
```

Host Tuning

Host tuning is optional for functional validation but recommended for paper-quality performance reproduction. GraCE evaluates CUDA Graph execution, where host-side CPU launch overhead and scheduling variability can affect measurements. The repository includes optional scripts for CPU/GPU frequency control, deep C-state control, hyperthreading control, and Docker-aware CPU-set isolation.

Before launching Docker, user may run:

```
sudo host_tuning/tune_host_intel_h100.sh
```

After tuning, launch Docker normally:

```
docker/run_cuda128.sh
```

After all experiments finish, restore the host:

```
sudo host_tuning/restore_host_intel_h100.sh
```

VTC: DNN Compilation with Virtual Tensors for Data Movement Elimination

Muyan Hu[1]* Ahan Gupta[1] Jiachen Yuan[1] Vima Gupta[2] Taeksang Kim[1] Xin Xu[1]
Janardhan Kulkarni[3] Ofer Dekel[3] Vikram Adve[1] Charith Mendis[1]
[1]*University of Illinois Urbana-Champaign* [2]*Georgia Institute of Technology* [3]*Microsoft*

Abstract

With the widening gap between compute and memory operation latencies, data movement optimizations have become increasingly important for DNN compilation. Current optimizations such as layout transformations and operator fusion only target a subset of tensor operators and consequently miss important opportunities for reducing data movement in contemporary DNN workloads, including large language models.

We introduce VTC, a novel tensor compilation framework that for the first time eliminates all unnecessary data movement by targeting the full spectrum of data movement operators. VTC proposes the concept of *virtual tensors* to track data movement between compute operators via index mappings rather than expensive physical data transfers to and from global memory, which can seamlessly interoperate with existing computation kernels and handle arbitrary tensor operator compositions. We also introduce a novel data movement elimination algorithm to automatically identify a profitable virtual tensor creation strategy. Evaluation on a variety of DNNs shows that VTC can outperform existing ML compilers by up to 1.93× (1.28× on average) on NVIDIA GPUs with up to 60% (17.5% on average) inference memory savings.

1 Introduction

Deep neural network (DNN) workloads have gained popularity in recent years. Usually, DNN models are expressed as tensor (i.e. $n-$dimensional array) based computations. Consequently, DNN developers express these computations using tensor programming languages such as TensorFlow [3], JAX [7], and PyTorch [5], and then utilize tensor compilers like XLA [1], TorchInductor [5], and TVM [9] to generate highly performant executables targeting various hardware devices. These compilers use a multi-stage pipeline similar to general-purpose compilers [6]. During the *compiler frontend*, computations expressed in tensor programming languages are transformed into a compiler intermediate representation (IR), known as *computation graph*. Nodes of these graphs represent tensor operators (e.g. matrix multiplications, convolutions), and each edge represents a tensor flowing from the output of a producer node to the input of a consumer node. The *compiler middle-end* performs graph-level transformations (e.g. operator fusion, tiling) to optimize the computation graph. Finally, the *compiler backend* maps the computation graph onto a set of kernels, each of which is a program fragment written in single-program-multiple-data (SPMD) fashion on modern hardware accelerators.

To keep up with the computational demands of DNN models, specialized hardware accelerators, such as NVIDIA Tensor Cores, have been introduced. Each new generation of these accelerators has consistently improved the compute capacity to provide blazing speeds for compute instructions such as matrix multiplications (e.g., nearly 1 PFLOPS half precision on NVIDIA H100). Most tensor compilers already aim to maximize utilization on such hardware using dedicated compiler backends (e.g. Triton [30] for GPUs), achieving low latency and high throughput for compute operations.

However, memory technology in modern hardware has not kept pace with these advances in compute capabilities. Figure 1 illustrates the widening compute-to-memory ratio as newer accelerators are introduced, making memory instructions significantly more expensive than compute instructions. Moreover, an increasing number of DNN models are becoming memory bound, with performance dominated by memory access requests to and from the accelerator's global memory, which exacerbates the impact of slower memory. For example, in large language models (LLMs), the incremental decoding stage is memory-bound and can be the bottleneck of end-to-end inference [27]. Memory access requests are determined by the types and the composition of *data movement operations* in computation graphs. These operations only transfer data between global memory and the accelerator without performing any computation on tensor data with compute units. Critically, these inserted data movements between computational operators could introduce substantial extra latency. For example, Figure 2 demonstrates TensorRT's latency breakdown on a

*Part of the work done during an internship at Microsoft.

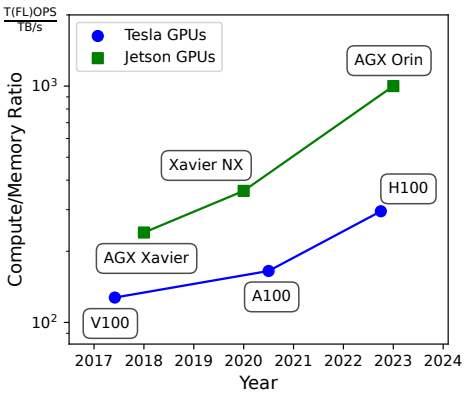


Figure 1: Trend of compute/memory ratio for NVIDIA GPUs over time. The ratio is calculated by dividing the computation power (half-precision performance in TFLOPS for Tesla GPUs and INT8 performance in TOPS for Jetson Edge GPUs) by the peak GPU memory bandwidth (in TB/s).

Llama 3 8B [14] decoder layer. Between the computation of QKV projection and FlashDecoding [11], the data movement operators take even more time than other computational operators. These findings underscore the critical importance of optimizing data movement operations in DNN compilers.

Prior works that propose optimizations for data movement operations in tensor compilers broadly fall under two categories: *layout optimizations* and *operator fusion*, both of which happen in the compiler middle end. These techniques are usually incomplete, targeting only a subset of the data movement operators and missing profitable data movement elimination opportunities that could lead to significant speedups, as detailed below.

Layout optimizations. The layout of a tensor determines in which order its dimensions are linearized into memory. When a producer operator generates an output tensor with a layout that differs from the expected input layout of the consumer operator, data layout conversion operators need to be inserted, resulting in excessive and avoidable data movement overhead. To find the optimal data layouts that minimize such overheads, previous work first identifies layout-sensitive tensor operators and optimally selects the adjacent layout operators (primarily `Reshape` and `Transpose` operators) to balance data movement overhead against computation efficiency [2, 16, 24]. However, the tensor operators considered in these works represent only a subset of data movement operations used in DNN models, overlooking optimization opportunities in other data movement operators (e.g., `ScatterND`) that are key for improved performance (see Section 3.1).

Operator fusion. During DNN execution, operators usually serve as boundaries of global memory data movement –

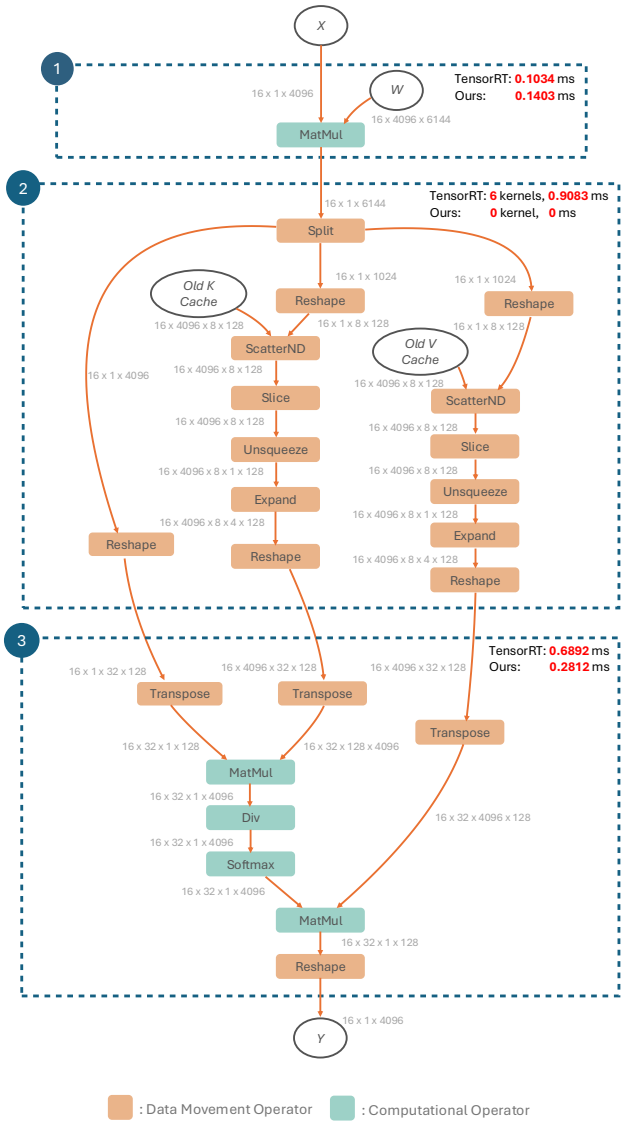


Figure 2: Motivating example: computation graph and latency breakdown for part of a Llama-3 decoder layer. The model size of Llama-3 is 8B, with batch size 16, context length 4096, query length 1 (decoding stage), `bfloat16` precision. Frame 3 denotes a single kernel fusing all the operators with FlashDecoding. VTC can eliminate all the data movement operators in frame 2 and achieve a 4× speed-up over TensorRT on this subgraph.

all input and output tensors are stored in global memory. For each operator, its corresponding kernel first loads inputs from global memory into on-chip buffers, performs computations, and then writes results back from on-chip buffers to global memory. To reduce this costly data movement, existing deep learning frameworks [2, 3, 5, 9] employ operator fusion, which enables intermediate results to be reused directly from fast on-chip buffers rather than being written to and read from

slower global memory. However, these systems perform fusion using hand-crafted rules that target specific data access patterns (e.g., [23]), making this optimization only available for operators that can be fused. As a result, similar to layout optimizations, operator fusion reduces data movement for only a subset of data movement operators, missing important optimization opportunities.

Our Solution. In contrast with these two popular techniques, in this paper, we introduce VTC, a compiler that eliminates unnecessary data movement across all data movement operators. For example, as shown in Figure 2, VTC can eliminate all the data movement operators between QKV projection and FlashDecoding. VTC is comprehensive and works for arbitrary composition of operators in computation graphs, making it generally applicable for all kinds of DNN models. Furthermore, it is complementary to both layout optimization and operator fusion.

The key idea of VTC is to eliminate data movement operators by using *virtual tensors* that store only a mapping function between physical tensors in global memory. This approach builds on the observation that modern hardware compute units require contiguity only in their local memory buffers, not in global memory. By relaxing the fully contiguous condition to partially contiguous without compromising memory coalescing, we can significantly eliminate data movement operators. Virtual tensors enable us to relax the contiguity condition by adding a level of indirection in the accesses to the tensors in global memory. However, realizing this virtual tensor representation requires addressing a few challenges to fully eliminate unnecessary data movement.

Challenge 1: Kernel creation that facilitates virtual tensors. With the introduction of virtual tensors, the current computational kernels cannot be used in their present form. First, these kernels assume contiguously laid out data in global memory, which is guaranteed by data movement operators. Second, virtual tensors can introduce extra overhead of mapping function and incontiguous memory access if used in current kernel implementations. We only modify the global memory I/O stages in existing computational kernels with block virtual tensor I/O (Section 4.2), resulting in minimal changes to the original kernel design and marginal runtime overhead.

Challenge 2: Determining a profitable virtual tensor construction strategy. Creating virtual tensors for all data movement operators is non-trivial. Different virtual tensor construction strategies lead to different profitabilities. We solve the problem of finding a profitable option in two parts. First, we introduce a *virtual tensor opportunity graph* to capture all virtual tensor creation options for a given computation graph (Section 5.1). Next, we design a global greedy algorithm that extracts a virtual tensor creation strategy with maximized latency savings (Section 5.2), which works empirically well in the evaluation.

In summary, we make the following contributions.

- We identify a new data movement optimization opportunity by allowing partially contiguous global memory storage in DNN compilation.
- We introduce the concept of *virtual tensors* for data movement elimination and efficiently support it in computational kernel generation.
- We propose *virtual tensor opportunity graph* and data elimination algorithm to automatically find an efficient strategy for creating virtual tensors.
- We implement these concepts in a compilation infrastructure, VTC. Our evaluations show up to 1.93× improvement (1.28× on average) over existing DNN compilers and up to 60% memory saving (17.5% on average) on NVIDIA GPUs.

2 Background

We first provide background about tensor stride, data movement operators at the graph-level, their necessity, and implementations of tensor operators at the kernel-level before explaining VTC.

2.1 Tensor Stride

To index an n-d tensor in the 1D memory, modern tensor compilers need to calculate an integer offset from the index vector $\vec{I} \in \mathbb{N}^n$. Tensor stride vector $\vec{S} \in \mathbb{N}^n$ indicates the number of elements we need to skip in memory to move from one element to the next in each dimension. For example, for a 3D tensor of shape (a,b,c), its stride vector $\vec{S} = (bc, c, 1)$. And the indexing will be simply the inner product of the stride vector and the index vector. In the same example, the offset of $\vec{I} = (i,j,k)$ is $\vec{I} \cdot \vec{S} = i \cdot bc + j \cdot c + k$.

2.2 Data Movement Operators

Data movement operators are a subset of tensor operators responsible for only moving data in global memory.[1] Examples include operators such as `Transpose`, `Split` and `ScatterND`.[2] As an example, the semantics of `Transpose` operator can be given as follows.

Definition 1 (`Transpose` Operator)**.** *`Transpose` operator takes a single input tensor and generates an output tensor by permuting the dimensions of the input with a given permutation `perm`:* $\forall i \in [0,n)$, output_dimensions[perm[i]] = input_dimensions[i].

[1] Conversely, computational operators are defined by their use of compute units to perform arithmetic operations on the data.

[2] In this paper, all mentioned operators are defined in ONNX (https://onnx.ai/onnx/operators/), a widely-used standard DNN format.

Figure 5 also shows a visualized example of `Split` operator. It is clear that `Transpose` and `Split` do not change values of individual elements, but rather reorganize their positions in the tensor. This property holds for any data movement operator and we formalize it as follows.

Definition 2 (Data Movement Operator)**.** *An operator with input tensors $I_1,\ldots,I_n$ and output tensors $O_1,\ldots,O_m$ is a data movement operator if and only if there exists a function $F(i,\vec{x}) = (j,\vec{y})$ such that $O_i[\vec{x}] = I_j[\vec{y}]$, for all $i \in [1,m], j \in [1,n]$, and $\vec{x},\vec{y}$ are subscripts of O_i, I_j, respectively.*

For the `Transpose` operator, $F(1,\vec{x}) = (1,\vec{y})$ where $y_i = perm[x_i]$. The mapping from outputs to inputs is one-to-one in data movement operators, ensuring that each output tensor element is derived from a unique input tensor element. Conversely, the mapping from inputs to outputs is one-to-many, indicating that an input tensor element may contribute to multiple output tensor elements.

2.3 Kernel Implementation of Operators

Graph-level tensor operators are typically implemented as hardware accelerator kernels with all the input and output data physically residing in global memory. The implementation can be divided into three distinct stages:

1. Transfer input data from global memory to on-chip buffers.
2. Perform computation with compute units, which read inputs from and write outputs to on-chip buffer (not present in data movement operators).
3. Transfer output from on-chip buffer to global memory.

For example, Listing 1 shows a matrix multiplication operator implemented in Triton [30] for GPU execution. In stage 1, two blocks of input matrices (2D-tensors) `a` and `b` are loaded as *contiguous* chunks from global memory into on-chip buffers. Stage 2 does the actual compute, in this case, it is an inner product of the loaded matrices. Stage 3 writes back the resultant `c` matrix from on-chip buffer to global memory also as contiguous chunks.

2.4 Necessity of Data Movement Operators

As seen in Section 2.3, tensor operators typically assume that the data is *contiguous* in memory during stages 1 and 3 of a kernel implementation. This serves two purposes: (1) satisfying the requirement of modern compute units for contiguous memory layout of inputs and outputs,[3] and (2) optimizing global memory access through memory coalescing and better locality. Due to the assumption of different kernel implementations, if a producer tensor operator uses a data layout or subsection that differs from its consumer tensor operator, data movement operators must be inserted to reconcile the difference. These data movement operators are necessary to maintain the correctness of the tensor computational graph, but they will introduce extra overhead.

[3]For example, NVIDIA tensor cores load two matrices of specific shapes from a contiguous memory fragment and then perform fused multiply-add (FMA) operations.

```
@triton.jit
def matmul_kernel(
    a_ptr, b_ptr, c_ptr, # Pointers to matrices
    ... # Elide some meta-parameters
):
    ... # Elide some pre-processing steps
    accumulator = tl.zeros(BLOCK_SIZE)
    for k in range(0, tl.cdiv(K, BLOCK_SIZE_K)):
        # Stage 1: global -> on-chip
        a = tl.load(a_ptrs, ...)
        b = tl.load(b_ptrs, ...)
        # Stage 2: computation
        accumulator = tl.dot(a, b, accumulator)
        # Advance the ptrs to the next K block
        a_ptrs += BLOCK_SIZE_K * stride_ak
        b_ptrs += BLOCK_SIZE_K * stride_bk
    ... # Elide some post-processing steps
    # Stage 3: on-chip -> global
    tl.store(c_ptrs, c, mask=c_mask)
```

Listing 1: An example of the three stages: a GPU matrix multiplication kernel written in Triton.

3 Motivation and Overview

3.1 Motivating Example: LLM Decoding

To illustrate the potential for data movement optimization in DNN inference, we examine the decoding stage of Llama 3 8B [14] as a motivating example. We profiled a single decoder layer using TensorRT [2], a state-of-the-art DNN compiler on NVIDIA GPUs. [4] TensorRT incorporates numerous expert-designed optimization rules and successfully identified the self-attention pattern in the decoder layer, triggering an ad hoc optimization for the entire layer. [5]

Figure 2 shows the computation graph and TensorRT's latency breakdown on an NVIDIA A100 GPU. TensorRT first applies graph-level transformations to merge the three query, key, and value (QKV) projections into a single matrix multiplication operation. It also employs the FlashDecoding algorithm [11] to fuse the computation of self-attention into one kernel. However, between these two computational kernels, there exists a large number of data movement operators, illustrated in frame 2 of Figure 2. The latency of these data movement operators in TensorRT even surpasses the combined latency of the two computational kernels.

[4]Similar latency distributions in this section were observed with other DNN compilers such as TorchInductor [5].

[5]This black-box optimization, called Myelin, fuses the entire transformer decoder layer into a single node and generates highly-optimized custom kernels for this node.

We now focus on the purpose of data movement operators and their role in the computation graph. TensorRT first executes a `Split` operator to extract the Q, K, V tensors from the result of merged QKV projection. Next, TensorRT uses `Reshape` and `ScatterND` operators to update the KV cache [25] with the new K and V tensors. Subsequently, TensorRT applies the `Slice` operator to keys and values to obtain a slice containing tokens up to the current sequence length. As Llama 3 8B employs grouped-query attention [14], TensorRT then uses `Unsqueeze` and `Expand` operators to align the number of KV heads with the number of query heads. Finally, TensorRT runs `Reshape` and `Transpose` operators to prepare the layout of keys and values for FlashDecoding. Although operator fusion is available for these data movement operators, TensorRT still requires 6 kernels to complete all the data movements above.

However, according to the analysis of layout requirements in Section 2, relaxing the fully contiguous condition of global memory storage may unlock opportunities for data movement operator elimination. For example, in stage 3 of the `MatMul` operator used for the merged QKV projection in frame 1, the `MatMul` kernel can directly write back the results to Q tensor and appropriate locations in KV cache, eliminating the need for `Split`, `Reshape` and `ScatterND` operators, as shown in Figure 3. Since the dimension per attention head (128) is larger than the GPU warp size (32), memory coalescing is not compromised, allowing for both good memory bandwidth in stage 3 and efficient utilization of tensor cores in stage 2. Similarly, by permitting stage 1 of FlashDecoding to read from not fully contiguous memory, the remaining data movement operators can be eliminated.

Based on these observations, VTC can eliminate *all* the data movement operators between QKV projection and FlashDecoding. For QKV projection, VTC writes to non-fully contiguous memory, resulting in a slightly slower `MatMul` kernel compared to TensorRT. However, VTC does not require any GPU kernel to explicitly execute the data movement operators shown in frame 2 of Figure 2, significantly reducing data movement overhead. Furthermore, after VTC's optimization, the FlashDecoding kernel can directly read from the KV cache instead of the duplicated data generated by the `Expand` operator. This optimization reduces global memory read overhead by a factor of 4 (the ratio of the `Expand` operator) and leads to a 2.5× speed-up of the FlashDecoding kernel.

3.2 Overview

Figure 4 shows VTC's overview. The input is a computation graph after operator fusion. To identify all the virtual tensor creation possibilities, VTC first runs an virtual tensor opportunity graph (VTOG) construction algorithm, based on VTC's virtual tensor definition and data movement optimization rules. Next, VTC runs a points-to graph construction algorithm to find a profitable virtual tensor strategy, represented by the resulting points-to graph. Finally, VTC generates an optimized executable according to this selected strategy.

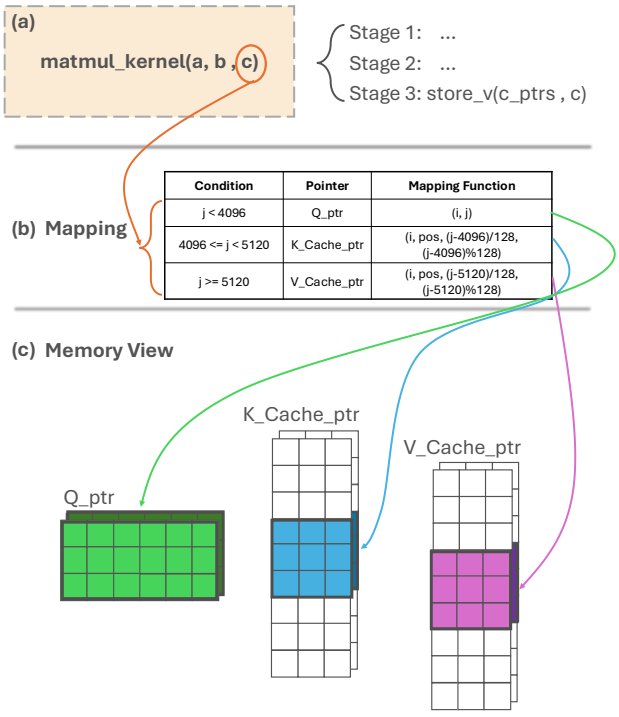


Figure 3: An example of data movement operator elimination with virtual tensors in QKV projection (frame 1 in Figure 2). The decoding stage is generating the *pos*-th token. Matrix $c[i, j]$ is a virtual tensor of shape (batch size, sum of QKV hidden dimensions), where Q has hidden dimension 4096, and K and V have hidden dimensions 1024 each. For the matrix multiplication kernel (Triton code is in Listing 1), we only need to modify stage 3, and the virtual tensor I/O function `store_v` will automatically write the data to proper physical tensor location according to the mapping function.

4 Virtual Tensor

Definition 2 reveals that it is possible to eliminate the need for actual data movement by storing only the mapping function F, which describes the relationship between the input and output tensors. By leveraging this insight, we can avoid the overhead associated with explicit data transfers in data movement operators. In this section, we introduce VTC's virtual tensor notation, which builds upon this observation to efficiently handle data movement operations. We will show how to handle virtual tensor I/O in Section 4.2 and analyze the profitability of the optimization in Section 4.3. Finally, we give several examples of how to use virtual tensors to eliminate data movement operators in Section 4.4.

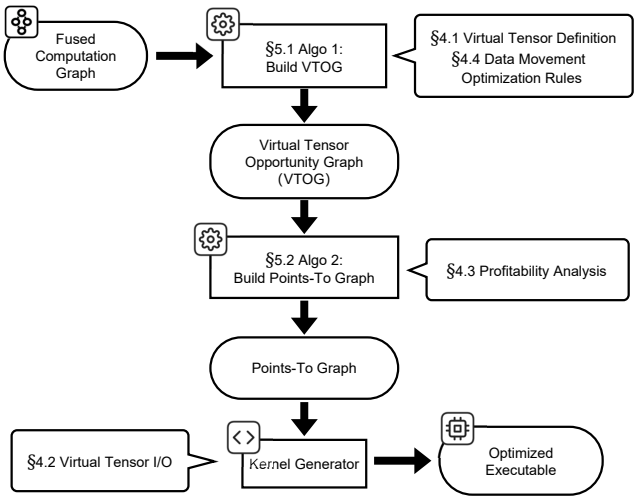


Figure 4: The overview of VTC.

4.1 Virtual Tensor Definition

The key idea behind the virtual tensor technique is to avoid redundant instantiation of tensor data in global memory. If a tensor is obtained by performing data movement operators on other tensors, we can directly represent it using a mapping function and a set of physical tensor pointers, instead of allocating global memory to store the tensor data.

Definition 3 (Virtual Tensor)**.** *A virtual tensor V is a tuple $(F, P_1, P_2, \ldots, P_n)$, where F is a mapping function and $P_1, P_2, \ldots, P_n$ is a set of physical tensor pointers. The mapping function $F : \vec{x} \rightarrow (j, \vec{y})$ describes the relationship between each virtual tensor index $\vec{x}$ of V and the corresponding physical tensor index $\vec{y}$ of P_j, where $j \in [1, n]$.*

Figure 3 shows an example of virtual tensor in QKV projection. Virtual tensor c has 3 physical tensor pointers Q_ptr, K_Cache_ptr, and V_Cache_ptr, with its mapping function illustrated in subfigure (b).

A virtual tensor provides a contiguous index space that references potentially non-contiguous locations within physical tensors. Mapping function F links each virtual tensor index to its corresponding physical tensor address. The virtual tensor technique eliminates data movement operators through two complementary approaches: (1) creating output virtual tensors for a producer data movement operation that map to input tensors of that operation, or (2) creating input virtual tensors for a consumer data movement operation that map to output tensors of that operation. While Definition 2 allows for one-to-many mappings, VTC restricts F to one-to-one mappings. This design choice simplifies implementation and avoids potentially expensive overhead of translating complex one-to-many mapping functions.

Mapping function composition. Virtual tensors can be nested: it is possible for a base tensor, which is used to create a virtual tensor, to be a virtual tensor itself. In such cases, the resulting virtual tensor's mapping function effectively applies the base tensor's mapping followed by the new mapping (e.g., $F_{\text{new}} = F_{\text{outer}} \circ F_{\text{base}}$), representing a chain of transformations.

4.2 Virtual Tensor I/O

VTC's virtual tensor interface seamlessly translates virtual tensor I/O operations into physical global memory I/O operations using the mapping function. In hardware kernels, global memory I/O is usually performed in a block-wise manner, where multiple threads simultaneously read a block of global memory data. For most real-world cases, the contiguity of mapping function conditions (e.g., 1024 in Figure 3) is larger than and a multiple of block size. Therefore, each block is likely to read from the same physical tensor, allowing us to directly apply a non-partitioned mapping function to a block of indices, which only introduces marginal overhead.

Virtual tensor provides a high-level abstraction that hides the underlying data movement complexities. Users can work with virtual tensors as if they were regular tensors, while VTC transparently manages the mapping and access to the physical tensor data behind the scenes. Since virtual tensors only impact the global memory I/O (stage 1 and 3 in Listing 1) of a hardware accelerator kernels, they leave the computation part (stage 2 in Listing 1) unchanged. This allows virtual tensors to be effortlessly integrated into existing optimized kernels with minimal modifications. For example, in Figure 3, we can directly pass a virtual tensor mapping to physical Q tensor and KV cache as the output parameter of the projection `MatMul` kernel. VTC then transforms the fully contiguous write to this virtual tensor into partially contiguous writes to the Q tensor and KV cache in `MatMul` kernel stage 3.

While virtual tensors have the potential to eliminate physical data transfers in data movement operators, the use of mapping functions and non-fully contiguous global memory access patterns still may introduce additional overhead. In the next subsection, we will delve into the profitability of this optimization technique, analyzing the trade-offs between the benefits of reduced data movement and potential extra costs.

4.3 Profitability Analysis

Since the indirect memory access patterns of virtual tensors are fully known at compile time, we can analyze the optimization profitability based on the characteristics of the mapping function. Similar to the classification of compiler optimizations into Type I (always profitable) and Type II (uncertain profitability) [22], we can categorize the profitability of virtual tensor optimizations by analyzing the mapping function. Without loss of generality, we assume each virtual tensor maps to a single base physical tensor here.

Remark 1. *In most DNNs, the final mapping function between the virtual tensor and a physical tensor can be expressed as*

$$f(\vec{I}) = \vec{S} \cdot \vec{I} + b(\vec{I}) \quad (1)$$

where $\vec{I}$ is the index vector, $\vec{S}$ is the stride vector (see Section 2.1) and $b : \mathbb{N}^n \to \mathbb{Z}$ is an index-dependent bias function.

Remark 2. *In ONNX operator set, most data movement operators preserve affine transformations on indexing, which can be achieved by adjusting stride and a* constant *bias. One exception is* `Expand`*. For instance, expanding a tensor from shape (a,b,c) to $(a,3b,c)$ and making the expanded tensor virtual yields the offset for index $\vec{I} = (i,j,k)$ as*

$$i \cdot bc + (j\%b) \cdot c + k = i \cdot bc + j \cdot c + k - \lfloor j/b \rfloor \cdot bc \quad (2)$$

Here for the mapping function in Equation (1), $\vec{S} = (bc, c, 1)$, $b(\vec{I}) = -\lfloor j/b \rfloor \cdot bc$.

Definition 4 (Contiguous dimension)**.** *For an n-d virtual tensor with shape $\vec{D} = (D_1, \ldots, D_n)$, dimension d is a* contiguous dimension *if its mapping function in Equation (1) satisfies*

1. *All strides from dimension d onward equal the suffix products of the shape vector: $\forall i \in [d,n]$, $S_i = \prod_{j=i+1}^{n} D_i$*
2. *Bias remains constant when the first $d-1$ dimensions are fixed: $\forall \vec{I_1} \in \mathbb{N}^{d-1}$, $\forall \vec{I_2} \in \mathbb{N}^{n-d+1}$, $b([\vec{I_1} \mid \vec{I_2}]) \equiv C$.*

Definition 5 (Partial contiguity and full contiguity)**.** *A virtual tensor with minimal contiguous dimension d and shape vector $\vec{D}$ retains* partially contiguous *memory access if $\prod_{i=d}^{n} D_i$ exceeds the minimal memory transfer unit size (e.g., memory coalescing size in GPUs). A virtual tensor maintains fully contiguous memory access if and only if its minimal contiguous dimension equals 1.*

Theorem 1. *Optimization is always profitable (Type I) for fully contiguous virtual tensors.*

Theorem 1 establishes the conditions for guaranteed profitability. When the mapping function preserves the original data access pattern with only a constant bias allowed, the virtual tensor optimization can be considered as Type I. Also, virtual tensors with partial contiguity are likely profitable since they do not break minimal memory transfer granularity and maintain high memory bandwidth. Otherwise, the profitability is uncertain. VTC will perform profiling (described in Section 5.2) to decide whether to proceed with the optimization according to actual latency impacts.

4.4 Virtual Tensor Optimization Rules for Data Movement Operators

The virtual tensor optimization is designed to eliminate data movement operators. In this subsection, we present several examples of data movement operators from ONNX, demonstrate how VTC eliminates them using virtual tensors, and discuss the profitability. All of the data movement operators presented in this subsection are outside the optimization space of previous layout optimizations.

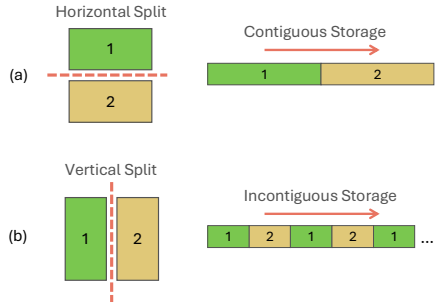


Figure 5: Comparison between contiguous `Split` (performed on the first axis) and incontiguous `Split` (performed on other axes) on a 2D matrix stored in row major.

Split When optimizing `Split` using virtual tensors, making input a virtual tensor of outputs is Type II, since the output tensors are stored separately and incontiguously in global memory. On the other hand, making outputs virtual tensors of input can be either a Type I or Type II optimization, depending on the contiguity of the split. If the split is performed along the first axis (axis 0), resulting in contiguous output tensors, it is a Type I optimization. Figure 5 shows an example of contiguous `Split` and incontiguous `Split`. When the incontiguous split size is large enough, all the two directions of virtual tensor optimizations have partial contiguity.

Expand The `Expand` operator broadcasts a single input tensor to a specified `shape`, producing a single output tensor with the same data as the input but with expanded dimensions. When optimizing `Expand` using virtual tensors, both directions are Type II. But if the number of duplicated elements exceeds memory coalescing size, it is almost always profitable.

ScatterND The `ScatterND` operator takes 3 inputs: `data`, `indices`, and `updates`. It generates a single output tensor by first initializing it as a clone of the `data` tensor and then updating the values at the specified indices in `indices` with the corresponding values from the `updates` tensor. There are two ways of virtual tensor optimization:

1. Making the output a virtual tensor of `data`. Since the output tensor is initially a clone of `data`, it is a Type I optimization.
2. Making `updates` a virtual tensor of the output tensor. This optimization is Type II because the `indices` can be incontiguous. In the LLM motivation example, `updates`

denotes all the locations in KV cache to be updated and is partially contiguous.

We can conduct analysis and define optimization rules similarly for all other data movement operators. As virtual tensor optimization rules are highly dependent on the operator semantics and the number of data movement operators is limited (e.g., 18 in the ONNX), VTC requires developers to specify the mapping function and possible virtual tensor optimization rules between inputs and outputs for each data movement operator. Then VTC can automatically analyze profitability and perform virtual tensor optimizations, which will be introduced in the next section.

5 Automatic Virtual Tensor Construction

This section introduces VTC's virtual tensor construction algorithm, which automatically makes intermediate tensors virtual to minimize unnecessary data transfers and improve overall performance. We first formalize the data movement elimination problem by building a virtual tensor opportunity graph (VTOG), and then introduce the global greedy algorithm to eliminate data movements.

5.1 Virtual Tensor Opportunity Graph

VTC's data movement eliminator takes a computation graph after operator fusion as input. In prior work, all the intermediate tensors, represented by edges in the computation graph, are physically stored in global memory. The data movement elimination problem aims to find a strategy to make certain intermediate tensors virtual and maximize the end-to-end latency savings.

VTC first builds a virtual tensor opportunity graph (VTOG) to analyze all the virtual tensor possibilities.

Definition 6 (Virtual Tensor Opportunity Graph)**.** *A virtual tensor opportunity graph (VTOG) is a directed graph* $G = (V, E)$*. Each node* $v \in V$ *represents a tensor in the computation graph. Each directed edge* $(u, v) \in E$ *represents a* direct *virtual tensor possibility, where tensor u can be made a virtual tensor of v by eliminating* only one *data movement operator.*

Figure 6 shows an example of a computation graph and its corresponding VTOG. `Split` determines the VTOG edges between `a` and `b`; `Reshape` determines the VTOG edges between `b` and `c`; and `ScatterND` determines the VTOG edges between `c`, `d` and `K Cache`.

For each node in the VTOG, some outgoing edges may conflict with each other, indicating mutually exclusive virtual tensor strategies. For example, in Figure 7, edges 1 and 3 conflict because designating `c` as a virtual tensor of both `a` and `d` simultaneously would map some indices of `c` to multiple physical memory locations, violating the one-to-one property

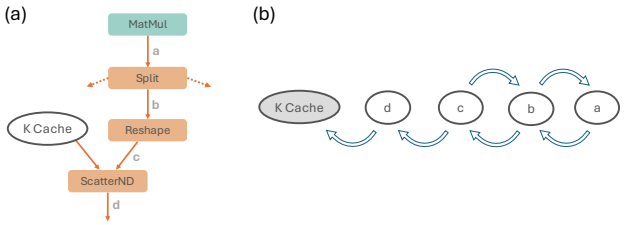


Figure 6: (a) The computation graph. (b) The corresponding virtual tensor opportunity graph. Other outputs of `Split` operator are ignored.

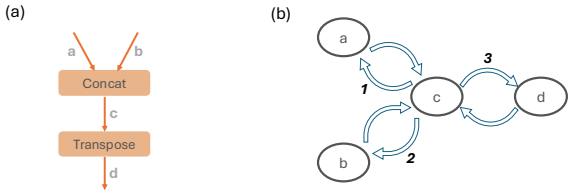


Figure 7: An example of a computation graph and conflicting edges in its VTOG. In (b), edge 1 and edge 3 conflict with each other. Edge 2 and edge 3 also conflict. However, edge 1 and edge 2 can be selected together.

of virtual tensor mapping functions. In contrast, edges 1 and 2 can coexist when preserving a one-to-one mapping. We capture all such conflicting edge pairs in a *conflict set*.

Algorithm 1 shows how to create a VTOG and the conflict sets from a computation graph. VTC constructs the VTOG by iterating through all data movement operators in the input computation graph, following the predefined virtual tensor rules for each operator (Section 4.4), and creating edges between operator inputs and outputs.

When all outgoing edges of each node in a VTOG are mutually compatible, the VTOG becomes a *points-to graph*, similar to that used in traditional compiler pointer analysis [4]. The points-to graph has one-to-one correspondence with a virtual tensor strategy for the entire computation graph. For example, Figure 8 shows several possible points-to graphs derived from the VTOG in Figure 6. Strategy 3 has no edges, indicating that all the tensors are physical. Strategy 2 has `b` and `K cache` as physical tensors, making `a` and `c` virtual tensors of `b`, and `d` a virtual tensor of `K cache`. Strategy 1 makes `a`, `b`, `c`, and `d` all virtual tensors of `K cache`. Note that `K cache` is an input tensor of the computation graph, so it must reside in physical memory.

Using an inductive argument, it is not hard to show the computational equivalence of the computation graph and the virtualization strategy corresponding to a points-to graph derived from the VTOG. As evident from the definition, a points-to graph can be derived from a VTOG by removing conflicting edges. However, different points-to graphs can lead to different end-to-end latency gains. The following subsection intro-

Algorithm 1 Build VTOG from a computation graph.

Input: A computation graph $G = (V, E)$
Output: The VTOG $= (V_{\text{VTOG}}, E_{\text{VTOG}})$ of G and a conflict set $S(v)\ \forall v \in V_{\text{VTOG}}$

$I = \{i \mid i \text{ is an input tensor of } G\}$
$O = \{o \mid o \text{ is an output tensor of } G\}$
$V_{\text{VTOG}} = I$
for $v \in V$ **do**
    $V_{\text{VTOG}} = V_{\text{VTOG}} \cup \{o \mid o \text{ is an output tensor of } v\}$
$E_{\text{VTOG}} = \{\}$
for $v \in V$ **do**
    if v is a data movement operator **then**
        for $(a, b) \in \text{VTRULES}(v)$ **do** ▷ a, b are input or output tensors of operator v and a can be made a virtual tensor of b
            if $a \notin I$ **and** $a \notin O$ **then** ▷ inputs and outputs of G must be physical tensors
                $E_{\text{VTOG}} = E_{\text{VTOG}} \cup \{(a, b)\}$
for $v \in V_{\text{VTOG}}$ **do**
    Construct $S(v)$ by enumerating outgoing edges of v
return $(V_{\text{VTOG}}, E_{\text{VTOG}})$, S

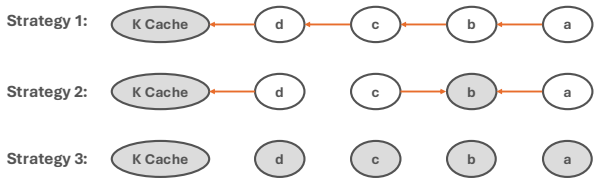


Figure 8: Several valid points-to graph of Figure 6, representing different virtual tensor strategies. Physical tensors are marked gray in each strategy.

duces VTC's points-to graph construction algorithm, which aims to obtain a points-to graph representing an efficient virtual tensor strategy.

5.2 Global Greedy Algorithm

The objective of our data movement elimination algorithm is to maximize the latency saving for each virtual tensor strategy, which can be formalized as follows. Let $(E_{\text{VTOG}}, \mathbb{F}, \ell)$ denote the virtual tensor search space corresponding to a VTOG. Here each edge in E_{VTOG} denotes a single virtualization opportunity, $\mathbb{F}$ denotes a set of all points-to graphs and $\ell : \mathbb{F} \to \mathbb{R}$ is the latency saving function that maps every points-to graph $C \in \mathbb{F}$ to the latency saving corresponding to virtual tensor strategy realized by C. Our goal is to find the optimal points-to graph

$$C^* = \arg\max_{\mathbb{F}} \{\ell(C)\} \tag{3}$$

Equation (3) is NP-hard as ℓ is a complex set function without any structure, which can only be measured through profiling on real hardware. In some situations we can expect that ℓ is a submodular function; for example, when all our operations are Type I (see Section 4.3). Unfortunately, it is not true for more general VTOGs.

Despite this, we take inspiration from the submodular function maximization literature, and design a global greedy algorithm to approximately solve our problem. It is well-known that the global greedy algorithm, which at each iteration adds the element that maximizes the discrete derivative of a submodular function ℓ, achieves a constant approximation to the optimal solution for a broad range of settings; we refer readers to [19] for an introductory survey.

Algorithm 2 shows VTC's points-to graph construction procedure. With all tensors stored in physical memory initially, it iteratively creates new virtual tensors. Specifically, the algorithm maintains an anchor node set A and a currently selected edge set C. In each iteration, it adds a new node v_c into the anchor node set by adding some non-conflicting edges between v_c and A into the currently selected edge set. For each edge $e \in E_{\text{VTOG}}$, we define its *discrete derivative with respect to* ℓ as

$$w(e) := \ell(C \cup \{e\}) - \ell(C), \tag{4}$$

which denotes the latency saving of merely adding edge e to C. The greedy strategy picks the node v_c that maximizes the sum of latency savings across its incident non-conflicting edges in the current iteration, then updates the sets A and C. Subsequently, since creating a new virtual tensor updates C, the algorithm recalculates $w(e)$ for each edge e pointing to v_c by profiling. If the maximum latency saving is negative or all the nodes are added to A, the algorithm terminates.

Complexity analysis. Algorithm 2 executes a maximum of $O(|V_{\text{VTOG}}|)$ iterations. In each iteration, it performs at most $O(|E_{\text{VTOG}}| + |V_{\text{VTOG}}|)$ calculations. Based on the VTOG definition 6 and its construction algorithm 1, $|E_{\text{VTOG}}|$ is linear with $|V|$ and $|V_{\text{VTOG}}| < |E|$. In typical DNN computation graphs, $O(|V|) = O(|E|)$, so $O(|V_{\text{VTOG}}|) = O(|E_{\text{VTOG}}|) = O(|V|)$. Consequently, the complexity of VTC's data movement elimination algorithm is the product of the maximum number of iterations and the cost per iteration, i.e. $O\left(|V|^2\right)$.

6 Implementation

We implemented VTC on top of Triton [30] and TorchInductor [5]. We first added virtual tensor support in Triton by overloading Triton's global memory I/O functions, `tl.load` and `tl.store`, enabling them to handle virtual tensor I/O operations. Mapping function is represented in the form of Equation (1). Our overloaded operators consume base physical tensor pointers along with mapping functions that determine how virtual tensors map to physical memory. Since all indirect memory access patterns through virtual tensors are fully known at compile time, we implemented overloaded `tl.load`

Algorithm 2 An iterative global greedy algorithm to build a points-to graph from a VTOG.

Input: VTOG $= (V_{\text{VTOG}}, E_{\text{VTOG}})$ and conflict set $S(v)$
Output: The points-to graph PTG $= (V_{\text{PTG}}, E_{\text{PTG}})$ of VTOG with maximized latency savings

1: $A = \{\}, C = \{\}$
2: **for** $v \in V_{\text{VTOG}}$ **do**
3: **if** v does not have outgoing edges **then**
4: $A = A \cup \{v\}$
5: **for** $e \in E_{\text{VTOG}}$ **do**
6: Profile and get $w(e)$
7: tot_saving $= 0$
8: **while** $A \neq V_{\text{VTOG}}$ **do**
9: max_saving $= -\infty$
10: **for** $v \in V_{\text{VTOG}} \setminus A$ **do**
11: $P, s =$ MaxEdges$(v, C(v), A)$ ▷ P is a set of non-conflicting edges between v and A with maximum sum of marginal latency savings s
12: **if** $s >$ max_saving **then**
13: max_saving $= s$, $P_c = P$, $v_c = v$
14: **if** max_saving < 0 **then**
15: **break**
16: $A = A \cup \{v_c\}$, $C = C \cup P_c$
17: tot_saving $=$ tot_saving $+$ max_saving
18: **for** $e = (t, v_c) \in E_{\text{VTOG}}$ **do**
19: Profile and update $w(e)$
return (V_{VTOG}, C), tot_saving

and `tl.store` by generating Python code specialized for each virtual tensor I/O, avoiding most of the indirection overheads.

We integrated VTC's data movement eliminator into TorchInductor through two passes: an analysis pass and a transformation pass. The analysis pass automatically generates a virtual tensor strategy with Algorithms 1 and 2. This entails identifying which physical tensors to promote to virtual tensors as well as computing mapping functions of promoted virtual tensors to their physical indices. The transformation pass then operates in two stages: (1) It mutates TorchInductor's IR nodes with virtual tensor awareness and removes any unnecessary data-movement operators. We modified TorchInductor's lowering strategy to correctly lower our overloaded `ops.load` and `ops.store` operators to Triton. (2) Once lowered to Triton, we generate the specialized Python code using the method mentioned earlier. The resulting Triton code is then compiled to produce an optimized hardware executable.

VTC's automatic virtual tensor construction employs a polynomial-time greedy algorithm rather than exhaustively searching in an exponential solution space. During compilation, profiling (lines 6 and 19 in Algorithm 2) dominates the time cost. Since VTC only modifies data movements in stages 1 and 3, no autotuning is needed for each kernel. The profiling overhead is less than 10 seconds per configuration, and the end-to-end compilation requires only a polynomial number of profiling runs, resulting in total compilation time under 10 minutes for all the models we have tested, which is on par with the compilation time of ML compilers [28, 37, 39].

As VTC focuses solely on optimizing data movement and does not alter the computation logic, it maintains end-to-end *numeric equivalence* with the original compiler, ensuring zero precision loss.

Table 1: Configurations of evaluated models.

Model	Component	Input Specification	Precision
Llama 3 8B Gemma 2 9B	Decoder Layer	Query Length: 1, Context Length: 4096	BF16
EfficientViT-Base YOLOv11n ShuffleNet	Attention Block C3K2 Block ShuffleUnit	Resolution: 4096 Resolution: 640 Resolution: 224	TF32

7 Evaluation

7.1 Experimental Setup

Platform. We conduct our evaluation on an A100 server and an H100 server. The A100 server is equipped with an Intel Xeon Platinum 8358 CPU and an NVIDIA A100 80GB PCIe GPU. The H100 server is equipped with an AMD EPYC 9454 Processor and an NVIDIA H100 NVL GPU. Both servers run Ubuntu 22.04, NVIDIA driver version 535.183.06 and CUDA version 12.1.

Workloads. We use five real-world DNNs from various domains to evaluate VTC. Llama 3 [14] and Gemma 2 [29] are transformer-based LLMs with grouped-query attention and local-global attention, respectively. EfficientViT [8] is a vision transformer backbone for high-resolution dense prediction with linear attention. YOLOv11 [18] is a powerful and efficient convolutional neural network (CNN) designed for a broad range of vision applications. ShuffleNet [38] is a CNN optimized for mobile devices. Table 1 provides detailed configurations for each model. We use PyTorch implementation for all the models. Except ShuffleNet, all the implementations are from official repositories. Following the common practice in previous ML compiler research [31, 40, 41], we evaluate all models with batch size 1 and 16.

7.2 End-to-end Performance

We first evaluate the end-to-end inference latency on a single A100 or H100 GPU and compare VTC with PyTorch 2.6.0 [5] (with `torch.compile` enabled), ONNX Runtime 1.21.1 [12], XLA[6] [1] and TensorRT 10 [2]. Figure 9 presents the end-to-end inference latency results with different batch sizes on A100 and H100, respectively. VTC achieves up to

[6] Since our benchmark workloads are implemented in PyTorch, we use `torch_xla` 2.6.0 package to run PyTorch models with XLA backend.

Table 2: Peak GPU memory consumption of PyTorch and VTC during end-to-end inference with batch size 1 and 16 on A100. All numbers are in Megabytes.

Model	Llama	Gemma	EfficientViT	YOLOv11	ShuffleNet
PyTorch (BS=1)	528.8	480.7	59.0	32.8	10.6
Ours (BS=1)	**461.8**	**436.1**	**55.6**	**26.7**	**10.4**
Savings (%)	**12.7%**	**9.3%**	**5.7%**	**18.5%**	**0.9%**
PyTorch (BS=16)	1787.3	1491.6	798.1	529.0	41.0
Ours (BS=16)	**714.5**	**776.4**	**773.9**	**450.9**	**39.9**
Savings (%)	**60.0%**	**48.0%**	**3.0%**	**14.8%**	**2.6%**

1.93× speed-up (1.28× on average) over the best of the four baselines (mostly TensorRT or PyTorch) in each benchmark.

During the evaluation, TensorRT successfully identifies the attention pattern for all Transformer-based models (Llama, Gemma and EfficientViT) and triggered highly optimized implementations for the entire model. [5] However, VTC still demonstrates an average speed-up of 1.36×. This is notably higher than the average speed-up of 1.15× observed for the CNN models (YOLOv11 and ShuffleNet), indicating that VTC effectively capitalizes on optimization opportunities inherent in newer model architectures like Transformers.

VTC achieves even greater speedup on H100 GPUs as compared to A100 GPUs for 7 cases out of the 10 benchmark cases (covering 5 models at 2 different batch sizes). This trend suggests that the optimization techniques employed by VTC are particularly effective on newer hardware accelerators, where the gap between compute power and memory bandwidth is getting larger (see Figure 1).

7.3 Memory Footprint

We also measure the inference GPU memory footprint after VTC's optimization compared to the PyTorch baseline (as VTC is built upon TorchInductor) using the PyTorch's built-in memory monitor. As shown in Table 2, even when optimizing for maximized latency reduction, VTC still achieves up to 60% peak memory saving (17.5% on average). This demonstrates that virtual tensor optimization inherently saves memory – it avoids storing full physical tensor data by representing virtual tensors with tensor pointers and mapping functions, and VTC handles virtual tensor I/O through code generation with no extra memory overhead. Analyzing the results in detail, we find that VTC successfully makes many large intermediate tensors virtual (e.g., the output of `ScatterND` in Figure 2), thereby reducing peak memory usage. Moreover, this highlights VTC's potential for more aggressive memory saving with relaxed latency constraints by reconfiguring the function ℓ in Section 5.2 to measure memory reduction, offering greater benefits for model training and deployment on memory-constrained hardware.

7.4 Latency Breakdown Analysis

Since TensorRT is the fastest baseline in most cases, we compare the latency breakdown of data movement operators and computational operators for VTC and TensorRT on A100. The results in Figure 10 reveal a substantial reduction in the latency proportion from data movement after VTC's optimization. In 7 out of 10 evaluated cases, VTC eliminates data movement operators entirely. Moreover, the analysis highlights that data movement operators account for a larger proportion of the overall latency in Transformer-based models compared to CNNs, enabling VTC to get better speed-ups.

However, in most of the benchmarks, VTC's latency on computational operators is higher than TensorRT's. There are two possible reasons: (1) VTC's computational kernels may perform virtual tensor I/O in non-contiguous physical addresses, introducing extra overhead in computational kernels (e.g., the `MatMul` kernel in frame 1 of Figure 2). (2) VTC is built upon Triton and TorchInductor, while TensorRT has more highly optimized implementations of computational kernels. This indicates that the performance gains from eliminating data movement significantly outweigh the slowdown incurred in computational operators.

7.5 Comparison with vLLM

As Figure 2 shows, VTC eliminates all data movement operators between QKV projection and attention in LLM decoder layer. Compared with existing compilers such as TensorRT and TorchInductor, the subsequent computational kernel FlashDecoding can launch directly after the previous computational kernel MatMul, improving the utilization rate of compute units without stalling on data movement operators.

To demonstrate the effectiveness of this optimization over a stronger LLM inference baseline beyond compilers, we compare VTC with a state-of-the-art LLM serving framework vLLM V1 [20] on Llama 3 8B inference. Since vLLM is PyTorch-based, we can seamlessly integrate VTC into it. We use similar settings (input sequence length 4096 and BF16 precision)[7] for evaluation on A100 and H100 GPUs.

Table 3 shows that vLLM has *manually* optimized many data movement operators from frame 2 of Figure 2 – the *KV Cache Update* row in Table 3 contributes to a small fraction of latency compared to compiler baselines. This poses a great challenge for VTC to compete, as vLLM is heavily optimized by domain experts on *specific* LLMs, while VTC is an *automated* compiler for *general* ML models. Nevertheless, VTC still identifies a redundant data movement involving the temporary tensor that stores QKV projection results, detailed in Table 4. This optimization yields a 1.043x speed-up on the

[7] vLLM V1 removes traditional notions of batch size and stage separation between prefill and decode through token-based iterative decoding. For our experiments, we construct an input batch of 4096 tokens and measure the corresponding attention latency.

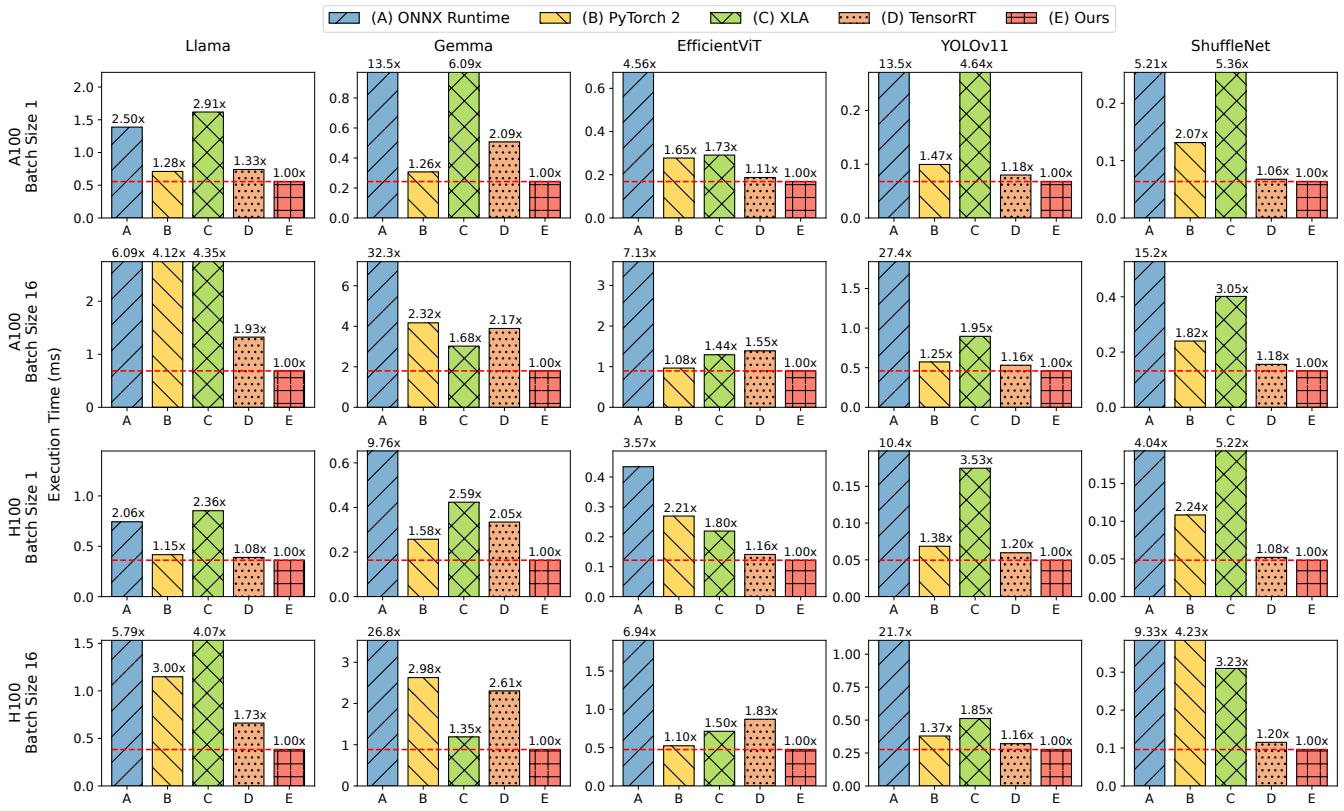


Figure 9: End-to-end inference latency comparison on a single NVIDIA A100 GPU and H100 GPU with batch sizes 1 and 16. Bars over 4× of VTC's latency are truncated.

Figure 2 computation graph and 1.011x end-to-end speed-up over vLLM on A100. However, no speed-up is observed on H100 since vLLM's cuBLAS MatMul kernel (QKV projection) significantly outperforms VTC's Triton kernel, and VTC's profile-guided virtual tensor construction automatically skips negative optimizations. Forcing the optimization from Table 4 on H100 causes an 8% performance degradation. We believe this gap could be bridged by integrating VTC into more optimized hardware-specific kernel libraries like CUTLASS in the future.

7.6 Case Studies

To gain a deeper understanding of how VTC eliminates data movement operators, we study some optimization cases found by VTC in detail.

EfficientViT. Figure 12 illustrates VTC's generated kernels and the points-to graph for EfficientViT attention block with batch size 16. All 5 kernels are computational kernels, and VTC eliminates data movement operators between these kernels by employing a virtual tensor strategy shown in the points-to graph. Among the tensors, only `a`, `e`, `f`, and `j` are physically stored. k_2 and k_3 directly read their inputs from the output tensor `a` of k_1. k_4 reads its inputs from the output tensor `f` of k_3 and writes to the input tensor `j` of k_5. VTC achieves only modest speed-up ($\sim 1.1\times$) and memory saving ($\sim 4.4\%$) on this model because most data movement elimination (e.g., optimizations related to `f`, `g`, `h`, `i` and `j`) can

Table 3: Comparison of Llama inference latency (ms) between VTC and vLLM. *QKV Projection*, *KV Cache Update* and *Attention* correspond to the 3 frames in Figure 2 in the same order. *Total* shows the latency on the Figure 2 subgraph. *End-to-end* shows latency for the entire decoder layer. *Speed-up* uses vLLM as the baseline. On H100, VTC defaults to identical behavior as vLLM, while enforcing optimization in Table 4 degrades performance.

	A100		H100		
	vLLM	VTC	vLLM	VTC	VTC w/ enforced opt.
QKV Projection	0.959	**0.914**	0.401	0.401	0.714
KV Cache Update	0.037	**0**	0.027	0.027	**0**
Attention	0.976	0.976	0.438	0.438	0.438
Total	1.972	**1.890**	0.866	0.866	1.152
Speed-up	1.000x	**1.043x**	1.000x	1.000x	0.752x
End-to-end	9.152	**9.055**	4.230	4.230	4.600
Speed-up	1.000x	**1.011x**	1.000x	1.000x	0.920x

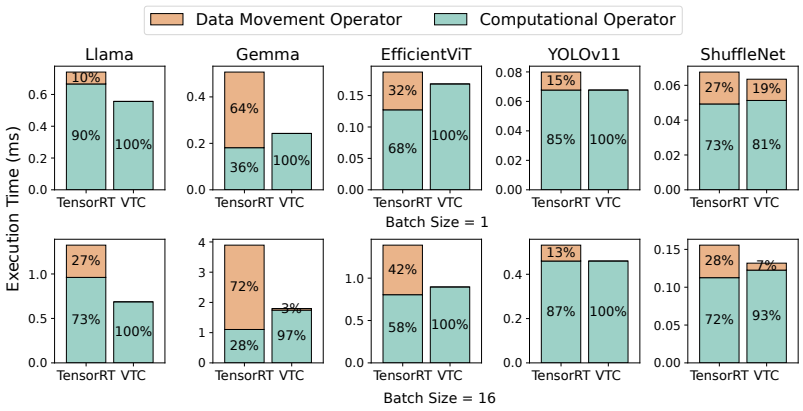


Figure 10: Breakdown of latency proportions for data movement and computational operators in VTC compared with TensorRT.

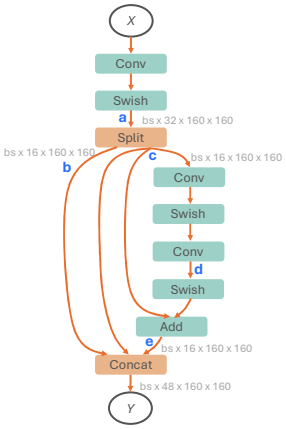


Figure 11: Computation graph of YOLOv11 C3K2 Block. VTC eliminates both data movement operators `Split` and `Concat` by making `a`, `b`, `c` and `e` virtual tensors of `Y`.

Table 4: Data movement optimization in VTC compared to vLLM. As defined in Section 2.3, *Stage 1 Input* and *Stage 3 Output* represent tensors stored in global memory that kernels read from and write to, respectively. vLLM allocates a temporary tensor to hold merged QKV projection values and splits it to update KV cache in an additional data movement kernel. VTC eliminates this overhead by making the temporary tensor virtual, directly writing results to Q tensor and KV cache.

vLLM	Stage 1 Input	Stage 2	Stage 3 Output
QKV Projection	Proj Input	MatMul	**Temporary Tensor**
KV Cache Update	**Temporary Tensor**	N/A	**Q, KV Cache**
Attention	Q, KV Cache	FlashAttn	Attn Output
VTC	**Stage 1 Input**	**Stage 2**	**Stage 3 Output**
QKV Projection	Proj Input	MatMul	**Q, KV Cache**
Attention	Q, KV Cache	FlashAttn	Attn Output

already be achieved by operator fusion in the baselines.

YOLOv11. Figure 11 shows the computation graph of C3K2 Block, a recurring component in YOLOv11. This block contains two data movement operators, `Split` and `Concat`. Traditional frameworks like TensorRT and PyTorch execute `Split` in an individual kernel and `Concat` in another kernel fused with `Swish` and `Add` (taking `b`, `c`, `d` as inputs and `Y` as the output). In contrast, VTC eliminates both data movement operators by making `a`, `b`, `c` and `e` virtual tensors of `Y`. Although `Split` and `Concat` are not fully contiguous at batch size 16 (Figure 5), each contiguous chunk ($16 \times 160 \times 160$ elements) is large enough to read and write in a coalesced manner, preserving high memory I/O bandwidth. Moreover, these two data movement operators constitute the primary memory footprint bottlenecks, as they operate on large tensors (`a`, `b`, `c`, `e` and `Y`). After VTC's optimization, all these tensors except `Y` become virtual, eliminating the need for physical memory allocation and reducing peak memory consumption.

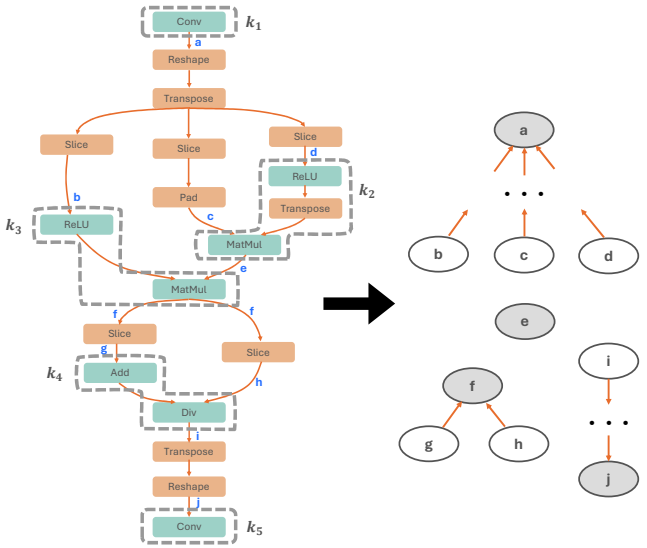


Figure 12: VTC generates 5 kernels (k_1–k_5) for EfficientViT attention block with batch size 16. All the unframed data movement operators are eliminated with virtual tensor. The figure on the right shows the virtual tensor strategy represented by a points-to graph.

8 Related Work

Virtual tensors. VTensor [34] is a programming framework which decouples tensor layouts from the programming interface, enabling developers to write layout-agnostic operations which significantly reduces lines of code. Separately, vLLM [20] and vTensor [33] are virtual memory management systems *specifically* designed for LLM serving. Unlike these systems, VTC optimizes *general* DNNs including LLMs with

a fundamentally different virtual tensor notation.

Layout optimizations. TensorRT [2] explores various layout options for 4D tensors and automatically applies the necessary layout transformations to determine the optimal layout. SmartMem [24] studies layout transformation elimination based on predefined rules and develops efficient memory layouts for 2.5D memory on mobile devices. In contrast to these layout optimizations, VTC proposes a more general optimization for data movements. Instead of focusing solely on layout operators (mostly `Reshape` and `Transpose`), VTC's optimization can be applied to all data movement operators. Furthermore, VTC's points-to graph construction algorithm automatically explores virtual tensor creation strategies, thereby expanding the optimization space beyond rule-based approaches.

Tensor graph optimizations. TensorFlow [3] and TensorRT [2] leverage manually designed rules to perform graph-level transformations. Recently, automated graph optimizers [17, 31, 35, 41] have emerged to automatically generate subgraph substitutions. VTC's optimization is orthogonal to these graph optimizations and can be used as an independent optimization pass after graph optimization in ML compilers.

Kernel orchestration is the process of mapping a computation graph to hardware-specific kernels [15]. This problem is commonly addressed by operator fusion [2, 3, 9, 23], which fuses multiple operators into a single kernel, thereby eliminating data movements of intermediate results. However, this type of work cannot optimize data movement operators that are hard to fuse with other operators. VTC takes a computation graph after operator fusion as input and further eliminates data movements between fused kernels.

Kernel generation. Numerous ML compilers [9, 40, 42] automatically generate hardware-specific kernels for DNN computation based on Halide's algorithm-schedule separation [26]. Rammer [21] and Hidet [13] approach kernel generation by utilizing a task abstraction of workloads. Mirage [32] employs superoptimization techniques on a multi-level compute hierarchy to generate efficient GPU kernels. As VTC only modifies the global memory I/O (stages 1 and 3) of a hardware-specific kernel, it can be easily integrated with virtual tensor support in these kernel generation frameworks.

Hand-optimized kernels crafted by domain experts can achieve state-of-the-art performance on specific DNN workloads. For instance, FlashAttention [10, 11] and FlashInfer [36] design highly optimized GPU kernels for self-attention calculation. VTC can also incorporate these manually optimized kernels by adding virtual tensor support.

9 Future Work

While our current virtual tensor construction algorithm provides performance guarantees through iterative profiling, this approach can incur substantial compilation overhead for time-sensitive applications. The VTOG formulation we present opens opportunities for developing more efficient algorithms that can compute high-quality virtual tensor strategies with reduced profiling requirements.

Our current implementation is based on Triton. However, Triton exhibits known performance limitations on newer GPU architectures, particularly NVIDIA Blackwell GPUs, where domain-specific languages such as CUTLASS demonstrate superior performance. Relatedly, the virtual tensor design does not explicitly consider emerging hardware-specific memory transfer mechanisms such as NVIDIA's Tensor Memory Accelerator (TMA), a hardware unit for asynchronous multi-dimensional memory transfers. While Triton handles some TMA patterns implicitly, explicit TMA awareness in the virtual tensor design could open additional optimization opportunities. Extending virtual tensor support to alternative DSLs and compilation frameworks, together with native handling of such bulk transfer mechanisms, is a promising direction for further performance gains on newer hardware architectures.

10 Conclusion

Data movement optimizations have become extremely important in tensor compilers to achieve good performance. VTC is the first compilation framework to eliminate all unnecessary data movement that spans across all data movement operators in a given tensor computational graph. We introduce the concept of a virtual tensor that allows VTC to track data movement between different physical tensors without instantiating intermediate tensors. VTC's data movement elimination algorithm allows us to achieve superior performance on a number of different DNN models including appreciable gains on large language model inference.

Acknowledgments

We thank Zhihao Jia, Hao Guo, Yuhao Ge and Yueying Li for their feedback and help on this work. This work was supported by PRISM and ACE, two of the seven centers in JUMP 2.0, a Semiconductor Research Corporation (SRC) program sponsored by DARPA, by NSF under the grant CCF-2316233, and by generous gifts from Qualcomm.

References

[1] Xla: Optimizing compiler for tensorflow. https://www.tensorflow.org/xla, 2017.

[2] NVIDIA TensorRT: An sdk with an optimizer for high-performance deep learning inference. https://developer.nvidia.com/tensorrt, 2024.

[3] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. {TensorFlow}: a system for {Large-Scale} machine learning. In *12th USENIX symposium on operating systems design and implementation (OSDI 16)*, pages 265–283, 2016.

[4] Lars Ole Andersen. *Program analysis and specialization for the C programming language*. PhD thesis, University of Copenhagen, 1994.

[5] Jason Ansel, Edward Yang, Horace He, Natalia Gimelshein, Animesh Jain, Michael Voznesensky, Bin Bao, Peter Bell, David Berard, Evgeni Burovski, et al. Pytorch 2: Faster machine learning through dynamic python bytecode transformation and graph compilation. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, pages 929–947, 2024.

[6] Marc Auslander and Martin Hopkins. An overview of the pl. 8 compiler. In *Proceedings of the 1982 SIGPLAN Symposium on Compiler Construction*, pages 22–31, 1982.

[7] James Bradbury, Roy Frostig, Peter Hawkins, Matthew James Johnson, Chris Leary, Dougal Maclaurin, George Necula, Adam Paszke, Jake VanderPlas, Skye Wanderman-Milne, and Qiao Zhang. JAX: composable transformations of Python+NumPy programs, 2018.

[8] Han Cai, Junyan Li, Muyan Hu, Chuang Gan, and Song Han. Efficientvit: Lightweight multi-scale attention for high-resolution dense prediction. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pages 17302–17313, 2023.

[9] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, et al. {TVM}: An automated {End-to-End} optimizing compiler for deep learning. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 578–594, 2018.

[10] Tri Dao, Dan Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. Flashattention: Fast and memory-efficient exact attention with io-awareness. *Advances in Neural Information Processing Systems*, 35:16344–16359, 2022.

[11] Tri Dao, Daniel Haziza, Francisco Massa, and Grigory Sizov. Flash-decoding for long-context inference. https://crfm.stanford.edu/2023/10/12/flashdecoding.html, 2023.

[12] ONNX Runtime developers. Onnx runtime. https://onnxruntime.ai/, 2024. Version: 1.20.1.

[13] Yaoyao Ding, Cody Hao Yu, Bojian Zheng, Yizhi Liu, Yida Wang, and Gennady Pekhimenko. Hidet: Task-mapping programming paradigm for deep learning tensor programs. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, pages 370–384, 2023.

[14] Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Amy Yang, Angela Fan, et al. The llama 3 herd of models. *arXiv preprint arXiv:2407.21783*, 2024.

[15] Muyan Hu, Ashwin Venkatram, Shreyashri Biswas, Balamurugan Marimuthu, Bohan Hou, Gabriele Oliaro, Haojie Wang, Liyan Zheng, Xupeng Miao, Jidong Zhai, et al. Optimal kernel orchestration for tensor programs with korch. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, pages 755–769, 2024.

[16] Andrei Ivanov, Nikoli Dryden, Tal Ben-Nun, Shigang Li, and Torsten Hoefler. Data movement is all you need: A case study on optimizing transformers. *Proceedings of Machine Learning and Systems*, 3:711–732, 2021.

[17] Zhihao Jia, Oded Padon, James Thomas, Todd Warszawski, Matei Zaharia, and Alex Aiken. Taso: optimizing deep learning computation with automatic generation of graph substitutions. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, pages 47–62, 2019.

[18] Rahima Khanam and Muhammad Hussain. Yolov11: An overview of the key architectural enhancements. *arXiv preprint arXiv:2410.17725*, 2024.

[19] Andreas Krause and Daniel Golovin. Submodular function maximization. *Tractability*, 3(71-104):3, 2014.

[20] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph Gonzalez, Hao Zhang, and Ion Stoica. Efficient memory management for large language model serving with pagedattention. In *Proceedings of the 29th Symposium on Operating Systems Principles*, pages 611–626, 2023.

[21] Lingxiao Ma, Zhiqiang Xie, Zhi Yang, Jilong Xue, Youshan Miao, Wei Cui, Wenxiang Hu, Fan Yang, Lintao Zhang, and Lidong Zhou. Rammer: Enabling holistic deep learning compiler optimizations with {rTasks}. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 881–897, 2020.

[22] Thirimadura Charith Yasendra Mendis. *Towards automated construction of compiler optimizations*. PhD thesis, Massachusetts Institute of Technology, 2020.

[23] Wei Niu, Jiexiong Guan, Yanzhi Wang, Gagan Agrawal, and Bin Ren. Dnnfusion: accelerating deep neural networks execution with advanced operator fusion. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, pages 883–898, 2021.

[24] Wei Niu, Md Musfiqur Rahman Sanim, Zhihao Shu, Jiexiong Guan, Xipeng Shen, Miao Yin, Gagan Agrawal, and Bin Ren. Smartmem: Layout transformation elimination and adaptation for efficient dnn execution on mobile. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, pages 916–931, 2024.

[25] Reiner Pope, Sholto Douglas, Aakanksha Chowdhery, Jacob Devlin, James Bradbury, Jonathan Heek, Kefan Xiao, Shivani Agrawal, and Jeff Dean. Efficiently scaling transformer inference. *Proceedings of Machine Learning and Systems*, 5:606–624, 2023.

[26] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. *Acm Sigplan Notices*, 48(6):519–530, 2013.

[27] Rya Sanovar, Srikant Bharadwaj, Renee St Amant, Victor Rühle, and Saravan Rajmohan. Lean attention: Hardware-aware scalable attention mechanism for the decode-phase of transformers. *arXiv preprint arXiv:2405.10480*, 2024.

[28] Yining Shi, Zhi Yang, Jilong Xue, Lingxiao Ma, Yuqing Xia, Ziming Miao, Yuxiao Guo, Fan Yang, and Lidong Zhou. Welder: Scheduling deep learning memory access via tile-graph. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*, pages 701–718, 2023.

[29] Gemma Team, Morgane Riviere, Shreya Pathak, Pier Giuseppe Sessa, Cassidy Hardin, Surya Bhupatiraju, Léonard Hussenot, Thomas Mesnard, Bobak Shahriari, Alexandre Ramé, et al. Gemma 2: Improving open language models at a practical size. *arXiv preprint arXiv:2408.00118*, 2024.

[30] Philippe Tillet, Hsiang-Tsung Kung, and David Cox. Triton: an intermediate language and compiler for tiled neural network computations. In *Proceedings of the 3rd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*, pages 10–19, 2019.

[31] Haojie Wang, Jidong Zhai, Mingyu Gao, Zixuan Ma, Shizhi Tang, Liyan Zheng, Yuanzhi Li, Kaiyuan Rong, Yuanyong Chen, and Zhihao Jia. {PET}: Optimizing tensor programs with partially equivalent transformations and automated corrections. In *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*, pages 37–54, 2021.

[32] Mengdi Wu, Xinhao Cheng, Shengyu Liu, Chunan Shi, Jianan Ji, Man Kit Ao, Praveen Velliengiri, Xupeng Miao, Oded Padon, and Zhihao Jia. Mirage: A {Multi-Level} superoptimizer for tensor programs. In *19th USENIX Symposium on Operating Systems Design and Implementation (OSDI 25)*, pages 21–38, 2025.

[33] Jiale Xu, Rui Zhang, Cong Guo, Weiming Hu, Zihan Liu, Feiyang Wu, Yu Feng, Shixuan Sun, Changxu Shao, Yuhong Guo, et al. vtensor: Flexible virtual tensor management for efficient llm serving. *arXiv preprint arXiv:2407.15309*, 2024.

[34] Jingling Xue. Vtensor: Using virtual tensors to build a layout-oblivious ai programming framework. *JOURNAL OF COMPUTER SCIENCE AND TECHNOLOGY*, 38(5):1074–1097, 2023.

[35] Yichen Yang, Phitchaya Phothilimthana, Yisu Wang, Max Willsey, Sudip Roy, and Jacques Pienaar. Equality saturation for tensor graph superoptimization. *Proceedings of Machine Learning and Systems*, 3:255–268, 2021.

[36] Zihao Ye, Lequn Chen, Ruihang Lai, Wuwei Lin, Yineng Zhang, Stephanie Wang, Tianqi Chen, Baris Kasikci, Vinod Grover, Arvind Krishnamurthy, et al. Flashinfer: Efficient and customizable attention engine for llm inference serving. *arXiv preprint arXiv:2501.01005*, 2025.

[37] Chen Zhang, Lingxiao Ma, Jilong Xue, Yining Shi, Ziming Miao, Fan Yang, Jidong Zhai, Zhi Yang, and Mao Yang. Cocktailer: Analyzing and optimizing dynamic control flow in deep learning. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*, pages 681–699, 2023.

[38] Xiangyu Zhang, Xinyu Zhou, Mengxiao Lin, and Jian Sun. Shufflenet: An extremely efficient convolutional neural network for mobile devices. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 6848–6856, 2018.

[39] Yifan Zhao, Hashim Sharif, Vikram Adve, and Sasa Misailovic. Felix: Optimizing tensor programs with gradient descent. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, pages 367–381, 2024.

[40] Lianmin Zheng, Chengfan Jia, Minmin Sun, Zhao Wu, Cody Hao Yu, Ameer Haj-Ali, Yida Wang, Jun Yang, Danyang Zhuo, Koushik Sen, et al. Ansor: Generating {High-Performance} tensor programs for deep learning. In *14th USENIX symposium on operating systems design and implementation (OSDI 20)*, pages 863–879, 2020.

[41] Liyan Zheng, Haojie Wang, Jidong Zhai, Muyan Hu, Zixuan Ma, Tuowei Wang, Shuhong Huang, Xupeng Miao, Shizhi Tang, Kezhao Huang, et al. {EINNET}: Optimizing tensor programs with {Derivation-Based} transformations. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*, pages 739–755, 2023.

[42] Size Zheng, Yun Liang, Shuo Wang, Renze Chen, and Kaiwen Sheng. Flextensor: An automatic schedule exploration and optimization framework for tensor computation on heterogeneous system. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 859–873, 2020.

Stop Pretending to be Busy: A Case for Serverless Paradigms in Co-located Batch Workloads (Operational Systems)

Xiaohu Chai[1,2], Jianfeng Tan[2], Congsi Yuan[2], Bowen Yang[2], Hao Dai[2], Tongkai Yang[2], Chao Huang[2], Dong Du[3], and Yu Chen[4,1]

[1]Tsinghua University, [2]Ant Group, [3]Shanghai Jiao Tong University, [4]Quan Cheng Laboratory

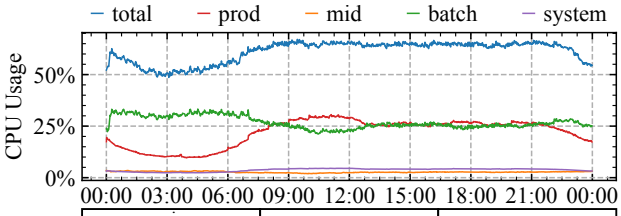
Abstract

High resource utilization is significant for cloud vendors. To achieve this, a common practice is to co-locate low-priority batch workloads (typically Spark-based analytics) with high-priority online services, while strictly maintaining Service Level Objectives (SLOs). This paper presents an empirical study of co-location and overcommitment in a production-scale datacenter. Specifically, within Ant Group, online services utilize only 22.0% of available CPU resources. By carefully overcommitting resources to deploy batch workloads, the system harvests an additional 26.8% of CPU capacity.

Despite this increased density, we observe that batch workloads remain inefficient, with a useful computation ratio of only 67%. We identify the root causes of this low "effective utilization" as four types of idleness: (1) *slot idle*, arising from coarse-grained resource management in Spark; (2) *gap idle*, caused by hardware heterogeneity and interference; and (3/4) *start/stop idles*, resulting from the high latency of launching and destroying analytic instances. To address these inefficiencies, we propose Quark, a novel framework that integrates serverless paradigms into batch analytics. Quark eliminates these idles through fine-grained resource allocation, heterogeneity and skew-aware scheduling, and rapid instance provisioning. Experimental results show that Quark increases cluster utilization by about 37.37% and reduces the proportion of long-tail jobs from 15% to 2%. Quark has been deployed at scale within Ant Group, processing 350,000 offline query jobs daily across a deployment footprint of 600,000 CPU cores, processing between 7,500 TB and 10,000 TB of data daily, and saving more than 100,000 CPU cores.

1 Introduction

Cloud providers strive to maximize resource utilization while strictly adhering to users' SLOs [56]. In modern cloud infrastructure, two primary strategies drive resource efficiency: *overcommitment* and *co-location*. Overcommitment enables providers to allocate aggregate resources exceeding a machine's physical capacity, leveraging the statistical probability that tenants will not reach peak usage simultaneously [13, 94, 100]. Complementing this, co-location schedules diverse workloads, combining latency-sensitive online services with resource-intensive offline analytics, onto the same physical nodes. This exploits complementary usage patterns to maximize utilization without a proportional increase in hardware, a common practice among major providers like Google [35, 78, 92], Alibaba [19, 40], and ByteDance [98].



	min	max	avg
total	48.6%	67.1%	61.2%
prod	9.59%	30.6%	22.0%
mid	1.92%	3.49%	2.72%
batch	21.2%	33.3%	26.8%
system	2.36%	4.59%	3.74%

Figure 1: **Co-location CPU Utilization in a Ant Group's Cloud Cluster.** Maintaining roughly 60% resource utilization achieves a practical trade-off between meeting online service SLOs and maximizing efficiency in clusters.

Ant Group also adopts these strategies to optimize its utilization. Our cloud overcommits resources such that total allocations exceed physical capacity, managing diverse workloads within a unified resource pool. As shown in Figure 1, a single node hosts a hierarchy of four distinct workload tiers. At the highest priority are *System* processes (cluster management daemonsets like etcd) and *Prod* services (latency-sensitive microservices or serverless functions [17, 39, 85]), which receive guaranteed resource allocations. Co-located alongside these are *Mid*-tier real-time tasks (not latency-critical but require stable and reserved resources, e.g., Flink [5]) and *Batch* workloads (opportunistic Spark jobs [12, 104]). *Mid* and *Batch* workloads run exclusively on overcommitted resources and are treated as best-effort; during contention, they are liable to

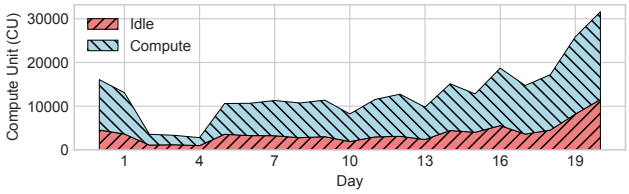


Figure 2: **Gap Between Allocated and Actual Compute Resources in Batch Workloads.** Light blue region represents utilization for actual compute tasks. Light coral region indicates idle time or non-effective computation. 1 CU indicates 1 CPU core and 4GB memory for a 1-hour period.

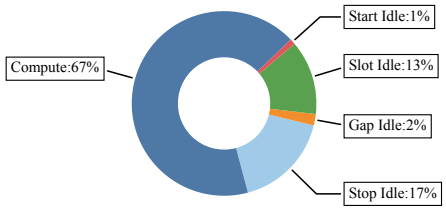


Figure 3: **Breakdown of Batch Workloads' Utilization.**

be throttled or evicted. Despite their lower priority, these workloads exhibit pronounced diurnal patterns, allowing effective co-location to significantly boost overall cluster utilization. However, based on large-scale production deployments and an in-depth analysis of existing solutions, we demonstrate that *the current coarse-grained job model [6, 8] for batch processing is structurally inefficient for co-location*.

Inefficiency in Batch Workloads. In Ant Group, the vast majority of batch workloads are processed using Apache Spark [104]. We execute over 350,000 Spark jobs daily, processing between 7,500 TB and 10,000 TB of data. This scale reflects a broader industry trend, where 80% of the Fortune 500 rely on Spark for large-scale data processing [11,26,29,97,102]. As shown in Figure 2, we analyze one month of resource usage in a production cluster, contrasting the resources requested by batch tasks against those effectively consumed (i.e., for computation). *Our analysis reveals that **33% of allocated resources were wasted** on non-effective states (e.g., waiting, initialization)*. Given the hyperscale of Ant Group, minimizing this hardware waste is critical.

We deconstruct this resource wastage into four types of idleness intrinsic to the current architecture. First, *Slot Idle* arises from Spark's coarse-grained allocation strategy, where an executor's resources remain fixed throughout its lifetime. This rigidity fails to adapt to the varying resource demands of different stages, leading to significant over-provisioning. Second, *Gap Idle* occurs when interference from high-priority workloads or hardware heterogeneity induces straggler tasks. Due to the bulk-synchronous nature of batch jobs, these stragglers delay the commit of entire stages, leaving other resources idle while waiting. Finally, *Start & Stop Idles* stem from the heavy initialization costs, such as JVM bootstrapping and SparkEnv [10] setup. To mitigate frequent startup overheads, the system retains executors for a timeout period before exiting, which inherently results in idle resource holding. Consequently, while overcommitment and co-location have improved cluster utilization in Ant Group, Figure 3 shows that a substantial portion of these resources is wasted.

Our Solution: Serverless Paradigm for Batch Workloads. We identify the root cause of the inefficiencies as a fundamental mismatch: traditional data analytics engines (like Spark) operate with long-running, pre-occupied, coarse-grained resource allocations, while co-located environments are dynamic, favoring fine-grained, on-demand provisioning. To bridge this gap, we advocate for ***applying serverless paradigms to batch workloads***. By transitioning from long-lived executors to task-level resource management, we align resource consumption with actual computational demand. While prior efforts have explored ephemeral executors [41, 43, 44, 95, 105], they often overlook the practical impediments to deploying such models at hyperscale, particularly regarding startup latency and scheduling overhead.

Realizing this vision in a production cluster presents three technical challenges. First, minimizing *slot idle* through task-level scheduling dramatically increases the pressure on the control plane; replacing thousands of coarse-grained executors with hundreds of thousands of individual tasks exacerbates scheduling complexity by orders of magnitude. Second, to address *gap idle*, the scheduler must account for the noise inherent in co-location. Online service interference and hardware heterogeneity create uneven computational power, causing stragglers that block stage completion. Finally, while on-demand provisioning resolves *stop idle*, it threatens to worsen *start idle*; without optimization, the cumulative latency of launching a fresh Java-based Spark instance for every task negates the efficiency gains.

We propose Quark, a novel analytics framework that fundamentally restructures batch processing by adopting a serverless paradigm where resources are provisioned and released at the granularity of individual tasks. Quark resolves the challenges through three key techniques. First, to ensure control plane scalability, we design a *scalable resource control* mechanism that regulates task parallelism via a *Slots Ring* and enforces global overcommitment limits through a *Quota Manager*, all orchestrated asynchronously to maximize throughput. Second, to mitigate hardware heterogeneity and interference, we introduce an *interference-aware scheduler* that converts raw node metrics into uniform capacity scores, enabling a variance-optimal placement algorithm that synchronizes task completion times. Finally, to resolve high cold start overheads, Quark employs *fast task provisioning*, which combines *state reuse* via `fork()` with *state pre-prepare* and *lazy-load* strategies, bypassing JVM initialization and removing non-critical dependencies (of Spark) from the critical path.

Deployment and Results. To the best of our knowledge,

Quark is the first reported commercial system to achieve near-zero resource waste for batch workloads in overcommitment and co-location scenarios. In the TPC-H benchmark, Quark reduced average resource consumption by 56.01%. In microbenchmark tests, the average task execution time decreased by 18% to 33%, the ratio of long-tail tasks dropped by 2.75×, and task startup overhead was reduced by 89.7%. In production environment tests, resource consumption decreased by 37.37%, the proportion of long-tail tasks fell from 15% to 2%, and the longest-to-shortest task time ratio dropped from 20× to 8×. The system has been deployed in a 600,000-core cluster, processing between 7,500 TB and 10,000 TB of data daily, and saving more than 100,000 CPU cores.

Research Contributions. First, it characterizes resource inefficiency in co-located batch workloads by identifying and quantifying distinct forms of idleness that prior work has not fully explored. Second, it proposes a serverless-based analytic framework that resolves critical challenges in scalability, heterogeneity, and latency through novel resource control and scheduling designs. Finally, it validates the system through a large-scale production deployment, offering practical optimizations and operational experiences that demonstrate its reliability and efficiency.

2 Demystifying the Reality of Cloud Utilization

We begin by profiling the overcommitment behavior and co-location workload characteristics in actual clusters. We then conduct an in-depth performance analysis of Spark, identifying critical bottlenecks that inform Quark's design.

2.1 Clusters and Resource Strategies

Clusters Characteristics. In Ant Group, millions of servers are deployed to host tens of thousands of services (different types), serving billions of users. To ensure data locality and stability, servers are distributed across 6 data center regions in different cities. As illustrated in Figure 4, the hardware within these clusters is highly heterogeneous, comprising multiple generations of servers from various hardware vendors. This diversity stems from staggered data center construction dates, the standard 4-year hardware warranty cycle, and the strategic need to ensure supply chain security. This practice of multi-generation server deployment is consistent with patterns reported in prior work [23, 25, 67].

Workload Placement. Ant Group supports a diverse ecosystem across finance, payments, and digital services. The applications of Ant Group include backend applications (microservices and serverless at 48.45%, AI agent at 3.95% and packages at 2.3%), web applications at 40.38%, big data at 4.43% (one platform supports all Spark SQL queries) and terminal applications. These applications are co-located in Ant Group and operate in accordance with a set of placement policies and specific allocate strategies.

(1) Placement Policy. As shown in Table 1, the placement policy follows a *hierarchical overcommitment* resource policy and *throttling and eviction* scheduling policies. Surplus capacity is allocated to lower-priority workloads when utilization of high-priority ones is low. Should their demand increase, lower-priority workloads may be throttled or evicted to guarantee Quality of Service (QoS) for higher-priority workloads.

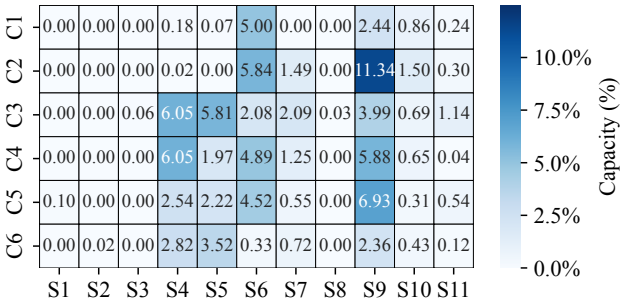


Figure 4: **Servers Distribution in Ant Group.** *C*1-*C*6 represents different clusters, and *S*1-*S*11 different types of servers.

Table 1: **Three-Layer Co-location of Workloads.**

Type	Resource Policy	Scheduling Policy
Prod	Strictly Guaranteed	Never throttled or evicted
Mid	Best-Effort	Can be throttled
Batch	No Guarantee	First to be throttled or evicted

(2) Allocation Strategies. When allocating resources, the scheduler determines the number and specifications of containers that can be deployed on each node based on the Placement Policy and a set of rules. Table 2 summarizes the allocation strategies for each workload. Intuitively, the ratio of prod and mid workloads is determined by pre-defined parameters, while batch is mainly determined by the remaining resources.

Table 2: **Allocation Strategies.**

Type	Allocate Ratio
Prod	min(NodeAllocatable * prodMaxCPUUsageRatio / prodCPUPredicatedUsage, HardLimit(Prod))
Mid	min(NodeAllocatable * midMaxCPUUsageRatio / midCPUPredicatedUsage, HardLimit(Mid))
Batch	min(NodeAllocatable * (1 - NodeUsageReserved) - NodeUsageExceptBatch, HardLimit(Batch))
NodeAllocatable: Total CPU cores in node. {prod, mid}MaxCPUUsageRatio: Max CPU utilization allowed. {prod, mid}CPUPredicatedUsage: Actual CPU usage. NodeUsageReserved: Reserved resources. *HardLimit*: Upper limit. NodeUsageExceptBatch: Per-node resource usage except batch.	

In Ant Group, the prodMaxCPUUsageRatio is 30%. Exceeding this threshold will incur two limitations: First, latency will increase and compromise the SLOs of online services. Second, it is essential to have capacity availability when needed during critical events such as cluster disaster recovery or unexpected workload spikes. There are few mid-type businesses, so the midMaxCPUUsageRatio is set to 10%.

According to these strategies, Figure 5 illustrates the CPU allocation ratios across different clusters. The ratios of *Prod* workload range from 94.6% to 121.3% and *Mid* workload

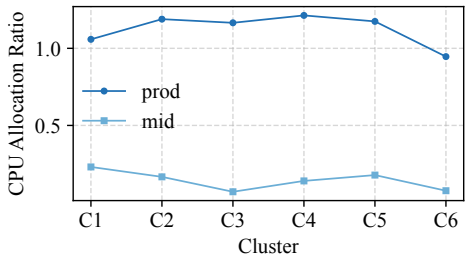


Figure 5: **CPU Allocation Ratios in Different Cluster.** Allocation ratio is defined as Allocated Cores/NodeAllocatable.

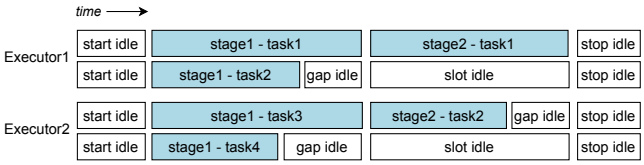


Figure 6: **A Spark Job's Execution Flow.** An example with two stages whose tasks are distributed across two executors.

range from 7% to 23.1%. *Batch* are not assigned a fixed allocation ratio; however, it is required that **the aggregate CPU utilization of all workloads on a node remain below 60% to avoid interference with *prod***. Figure 1 shows each type workload in Ant Group. In a co-located context with limited resource quotas, heterogeneity, and low priority, we need to consider how to efficiently run batch workloads.

2.2 Inefficient Co-location Job Model

We operate *batch* workloads on Spark [11], the de facto standard for big data processing. In Spark, each *job* is represented as a directed acyclic graph (DAG), and each DAG consists of multiple stages. A *stage* contains parallel *task*s that perform the same work, and those tasks are scheduled to run on *executor*s. In this paper, we will follow the same convention [8] to use the terms including *job*, *stage*, and *task*. Figure 6 illustrates an execution flow example across multiple nodes in the cluster. Through real-world deployments, we identify four types of idle or non-productive costs (uniformly referred to as *idle*) that significantly impact resource efficiency.

(1) Start Idle. Starting an executor is a complex, resource-intensive process in Spark. The startup procedure begins when the driver [8] requests executors from the cluster manager (e.g., Kubernetes), which launches pods on worker nodes. A pod is a basic scheduling unit in the cloud. After a pod is allocated, the JVM executor starts, initializes the RPC environment, and registers with the driver. It then creates SparkEnv [10], starts services such as the BlockManager and BlockTransferService, initializes memory and shuffle subsystems, and localizes driver-distributed jars and broadcast variables. Once BlockManager has registered, the driver marks the executor ready and begins sending LaunchTask RPCs.

(2) Slot Idle. An executor operates with fixed resource limits during its lifetime. When tasks at different stages require different resources, the executor must be provisioned to meet the peak demand for the entire job. This over-provisioning leads to resource wastage during less demanding stages. Adjusting resource allocation by replacing or restarting executors incurs significant performance penalties, primarily due to task rescheduling overhead and the potential loss of intermediate data. Previous work focuses mainly on predicting optimal resource configurations [18, 83] and reducing the granularity of resource allocation [97]. These approaches are hard to handle real-time resource requirement variation of different stages.

(3) Gap Idle. Spark adheres to the Bulk-Synchronous Parallel (BSP) model [46], requiring all tasks within a stage to complete before the next stage can commence. Consequently, slow tasks (stragglers) cause resource waste, with a few executors doing left work while others idle, delaying the entire job. Many studies have reported various straggler factors [2, 32, 53, 70]. In Ant Group, we focus on two cluster-related impacts. First is the *co-location*: batch workloads utilize overcommitted resources, and will be evicted when higher-priority workloads (e.g., system, production, mid-tier) contend for resources. Second, *hardware heterogeneity* within the cluster presents challenges. Servers in Ant Group follow 4-year replacement cycles and are sourced from multiple OEMs for supply-chain reasons. This heterogeneity results in uneven computing power across the cluster (Figure 4), causing unaligned tasks finish time within the same stage.

(4) Stop Idle. In Spark, an executor is a long-lived Java process, while a task runs as a thread within it. Executors do not terminate upon task completion. They persist until the driver requests their release from the cluster manager at application termination. When dynamic allocation feature is enabled, idle executors exceeding the configured executorIdleTimeout are decommissioned. Executor startup is slow (start idle), setting the executorIdleTimeout too low will increase latency and resource churn. The default executorIdleTimeout is 60 seconds, but it is often extended to several (or tens of) minutes in production environments.

2.3 Existing Approaches

Researchers have proposed a wide range of techniques, such as workload co-location and data analytics, to address performance and resource utilization issues in cloud infrastructures [3, 19, 23, 25, 29–31, 40, 41, 43, 44, 49, 66, 74, 75, 78, 84, 92, 97, 98, 105], as shown in Table 3. Some of these works are applicable and have inspired parts of Quark's design, which has been adopted in large-scale production environments. However, others neglect the critical issues discussed in §2.2 or rely on simplified assumptions that overlook the complexities of large-scale systems and cannot address the identified idles.

(1) Spark Optimizations. Apache Spark is the most widely used system for large-scale data processing and involves the coordinated work of many components. There are many efforts to improve its performance, including shuffle mechanisms [26, 59, 81], native compilation [29, 102], resource optimization [79], configuration tuning [50, 61, 99, 110] and so

Table 3: **Comparison of Existing Approaches and Quark.** (●is beneficial, ◐is partly beneficial, and ○is limited.)

Approaches	Related work	Slot Idle	Gap Idle	Start&Stop Idle
Spark Optimizations	Wu et al. [97], ResourceProfile [7]	◐	○	○
Cluster Management	Bubble-up [64], Quasar [25], Paragon [23], Gsight [109], Jiagu [54]	○	●	○
Serverless Analytics	Starling [74], Caerus [105], Ditto [41], Jiffy [44], MinFlow [49], MetaQ [95]	●	○	○
Quark(this paper)	-	●	●	●

on [22, 65, 72]. Some of the research efforts are orthogonal to, yet complementary with, the issues discussed in §2.2. Furthermore, prior work has also discovered that there is an urgent need to handle resource requirements variation of different stages (e.g., Wu et al. [97] and ResourceProfile [7]). However, these methods merely adjust resource configurations to align executors more closely with the average resource utilization. Although this yields certain improvements, the underlying two-layer *executor–tasks* job model remains unchanged, and the fundamental problem therefore persists.

(2) Cluster Management. Overcommitment and co-location may lead to performance degradation and QoS violation, it is a common practice to predict performance before scheduling an instance to avoid QoS violation. For example, previous works use modeling (e.g., Graph [33], Linear [60], RL [63], Random Forest Regression [54]) to determine optimal placement strategy [64, 73, 86, 101, 109]. Although this work offered valuable insights and inspirations that informed the design of Quark, there are still some issues that need to be addressed to optimize *gap idle*. First, existing works [54, 101, 109] focus on a single node, aiming to maximize instance deployment density without violating instances' QoS. However, batch workloads following the BSP model require that all tasks within the same stage have as uniform execution times as possible. Second, it fails to differentiate features among *prod*, *mid*, and *batch* workloads and the impact of heterogeneous hardware, prediction accuracy is limited [24, 25].

(3) Serverless Analytics. The elasticity, free-of management, and on-demand scalability of serverless have motivated the effort in deploying distributed data analytics applications to serverless platforms. We evaluate the potential of the existing work [41, 43–45, 49, 66, 74, 75, 84, 105] to provide fine-grained resource control and reduce slot idleness. Our analysis reveals two fundamental limitations in existing solutions: (1) *Coarse-grained resource allocation*. Some prior serverless analytic systems deploy function at executor scale [43, 45, 95], which cannot adapt to stage-level variations and leaves slot idle issues unresolved. (2) *Overlook the complexities of large-scale systems.* Ditto [41] allows each task to be assigned to a serverless function, but it neglects the task startup cost in real systems. MetaQ [95] reports that it takes $17.78s \sim 47.02s$ to start a Spark engine. When the granularity of serverless functions shifts from executor to task, the cumulative cold start penalty will increase enormously. Some systems [74, 105] remain in the prototype phase and are difficult to deploy at scale in practical scenarios, mainly due to challenges related to control plane scalability, heterogeneity, and cold starts.

3 Quark Overview

To address the inefficiencies of co-location (i.e., four idles), we propose Quark, a framework that re-architects batch workloads using a serverless paradigm. The core insight of Quark is to transition from a *coarse-grained and pre-allocated* resource model to a *fine-grained and on-demand* model. Specifically, a task could request one core (or sub-core) when it starts and release it immediately after completion. By utilizing the individual task, rather than the heavy-weight executor, as the basic unit of resource allocation, Quark allows the cluster to effectively harvest transient overcommitment resources.

3.1 Challenges

However, adapting a heavy-weight analytics engine like Spark to this fine-grained model introduces three challenges.

Challenge-1: Scheduling Scalability and Efficiency. The transition from coarse-grained executors to fine-grained tasks creates a massive surge in scheduling events, imposing severe pressure on the control plane. Figure 7 shows the CDF of task counts per stage; we observe that over 10% of stages exhibit parallelism exceeding 184, with peak parallelism reaching 52,383 tasks in a single stage. This explosion in scheduling introduces three specific issues. First, handling hundreds of thousands of individual task scheduling requests, as opposed to a few thousand executor requests, can saturate the cluster scheduler (e.g., Kubernetes), potentially disrupting the stability of co-located online services. Second, because native Spark lacks a global view of real-time overcommitted resources, it may aggressively issue scheduling requests that exceed available capacity, leading to a high volume of failed and wasteful invocations. Finally, the overhead of managing the lifecycle of short-lived tasks creates a throughput bottleneck in the control plane, increasing end-to-end latency.

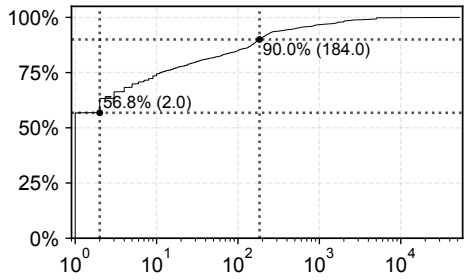


Figure 7: **Distribution of Tasks per Stage.**

Challenge-2: Hardware Heterogeneity and Interference. To effectively mitigate gap idle, the scheduler must ensure that

tasks within the same stage finish simultaneously. However, achieving this is difficult due to the significant variance in effective computing power across the cluster. As shown in Figure 4, servers in Ant Group follow a four-year replacement cycle and are sourced from multiple vendors, resulting in inherent hardware heterogeneity. Meanwhile, batch workloads run at low priority and are subject to being throttled or evicted by online service. Standard schedulers typically view resources nominally (e.g., CPU core counts) and overlook these performance variations. Without accounting for this heterogeneity and interference, tasks assigned to slower or noisier nodes become stragglers, delaying the completion of the entire stage and reducing cluster efficiency.

Challenge-3: Cold Start Overhead. The ephemeral nature of serverless tasks fundamentally clashes with the high cold start costs of the Spark runtime. While general-purpose serverless platforms utilize techniques like forking or checkpointing to optimize startup, Spark's initialization is uniquely complex, involving the setup of heavyweight components such as the JVM, RpcEnv, and RemoteDataManager. Currently, the startup latency for a standard Spark instance exceeds 6 seconds. In a coarse-grained model, this cost is amortized over the lifetime of a long-running executor. However, in a fine-grained model where instances are created and destroyed for specific tasks, this multi-second overhead becomes a dominant factor, potentially negating the benefits of on-demand allocation. Reducing this startup latency to a level compatible with short-lived tasks remains a critical open challenge.

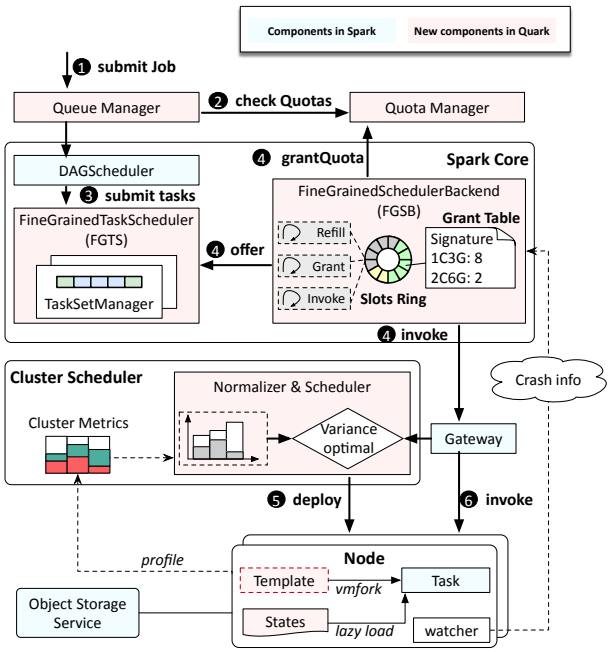


Figure 8: **Quark Overview.**

3.2 Design Overview

System Architecture. Figure 8 shows the overall architecture of Quark. Built atop the full Spark stack, Quark preserves complete compatibility with Spark-SQL while establishing a fine-grained resource model. The system operates across three logical layers. First, the **Spark Core** serves as the primary control plane; it receives analytic jobs and generates tasks via the *TaskScheduler*, which then coordinates with the *SchedulerBackend* to interact with the underlying infrastructure. Second, these components interface with the **Cluster Scheduler** within Ant Group's serverless platform, which handles the placement of individual tasks onto physical nodes. Finally, at the **Node Layer**, Quark utilizes lightweight, VM-based containers to execute these serverless instances, launching a fresh runtime for every scheduled task. These instances are stateless [74], driver/executor metadata objects that are traditionally pulled from executor memory (e.g., broadcast-related metadata) are stored in remote cloud object storage services and fetched on demand. Intermediate shuffle data are managed by vanilla Spark's ShuffleManager. This eliminates the need for rack-aware data locality during instance scheduling. Quark introduces three key techniques to address the challenges in §3.1.

Scalable Resource Control. To overcome the **Challenge-1**, we firstly design a *Slots Ring*-based mechanism within the Spark Core to regulate task parallelism and coordinate Spark scheduler components. Besides, we introduce a *quota-based resource management* to explicitly manage global overcommitment capacity, thereby reducing wasteful scheduling attempts. Finally, we adopt fully asynchronous principles in the control plane to decouple interdependent operations and maximize throughput.

Interference-aware Scheduler. To mitigate heterogeneity and interference (**Challenge-2**), Quark introduces a *Interference-aware Scheduler* situated within the serverless platform. Its primary goal is to normalize the effective computing power provided to tasks within the same stage. By monitoring real-time system metrics, the scheduler converts raw node resources into a uniform capacity score using a normalizer model. It then applies a variance-optimal placement algorithm to distribute tasks, significantly reducing the occurrence of stragglers and minimizing tail latency.When online services burst, a local agent detects contention signals (e.g., SLA degradation, LLC misses) and enforces an online-first policy by throttling batch resource entitlements and adjusting the node's reported available capacity.

Fast Task Provision. To resolve the high cold start overhead (**Challenge-3**), Quark optimizes the task initialization through three mechanisms. First, we employ *State Reuse*, where the system maintains pre-warmed template instances and creates new tasks via a fast *vmfork* operation, bypassing the initialization of JVM and parts Spark modules (e.g., executor-level state). Second, we use *State Pre-Prepare* to compute task-specific states (such as Codegen bytecode) in the driver and transfer them to the executor, avoiding redundant compilation. Last, we implement *State Lazy-Load* to remove non-critical components from the startup path, initializing them in parallel

only when needed.

4 Detailed Design

4.1 Scalable Resource Control

We first explain the techniques for scalable resource control.

Slots Ring-Based Task Management. In the traditional Spark architecture, a *TaskScheduler* decides which tasks should run, and a *SchedulerBackend* communicates with the underlying cluster manager (e.g., Kubernetes) to acquire and launch execution instances, as shown in Figure 8. To prevent the fine-grained model from leading to uncontrolled parallelism, Quark introduces the *Slots Ring* as a key data structure within the Spark Core. This ring buffer maintains the tasks currently allowed to be scheduled, helping to coordinate the *FineGrainedTaskScheduler (FGTS)* and the *FineGrainedSchedulerBackend (FGSB)*. The *FGTS* supplies new tasks (*refill*) to the ring only when idle slots are available, and the *FGSB* then attempts to acquire quotas and launch instances for these slotted tasks (*grant* and *invoke*). By configuring the size of this ring, Quark precisely controls the global parallelism of batch workloads, preventing scheduling pressure from overwhelming the cluster. Analogous to Spark's driver [9], *FGSB*/*FGTS* are centralized. They support HA via failover (restart/leader election depending on the deployment), but at any time only one instance is responsible for making scheduling decisions. Although centralized, the scheduler offer scalability because the slots-ring changes the critical path from O(M*N) tasks/executors match to O(1) enqueue/dequeue per scheduling event and avoids global contention.

Quota-Based Resource Control. Instead of relying on the cluster scheduler to reject scheduling requests due to lack of resources, which can lead to contention and wasted API calls, Quark introduces the *QuotaManager (QM)* to explicitly manage the global overcommitment capacity. *QM* is a distributed resource management service where a single leader instance is responsible for all allocation decisions, with follower nodes providing fault tolerance.When the leader fails, Quark enters a short failure-recovery window (<1 min) for failure detection and leader election. During this period, jobs that have already received a quota grant are not affected, and new submissions that require fresh quota decisions will be temporarily stalled until a new leader is ready. The leader maintains the current resource usage and schedulable tasks in an in-memory cache, which is periodically synchronized with a backend database. This design minimizes database load and accelerates scheduling decisions. Quota allocation is based on in-memory operations, so the leader will not suffer from scalability issues or performance bottlenecks.

As shown in Figure 9, operators define a total available quota for each *project* (representing a tenant or team). A submitted job registers its priority, and submission time within its project. Subsequently, the *FGSB* summarizes the resource needs of the tasks in the Slots Ring and requests a resource

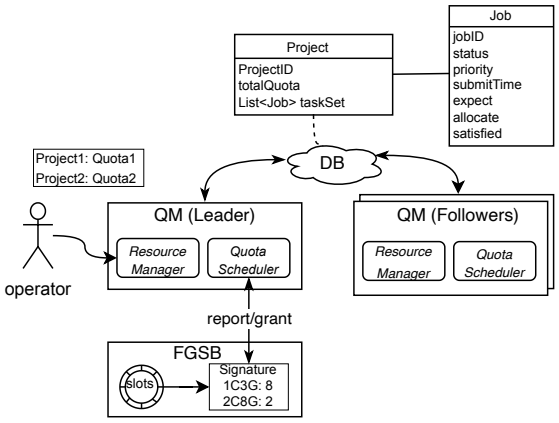


Figure 9: **Quota Manager.**

grant from the *QM*. Only upon receiving a successful grant does the *FGSB* proceed to invoke the underlying serverless platform. This explicit resource negotiation significantly reduces the number of failed and wasteful invocations.

Quota Allocation. The quota allocation procedure focuses on reclaiming under-utilized quota and prioritizing high-demand jobs. First, the *QM* reclaims resources by returning allocated quotas from completed jobs and adjusting the allocation of running jobs downward to their expected demand, recovering any surplus. Second, the reclaimed quota is re-distributed to jobs whose current allocation is below their expected level. Unfinished jobs are prioritized based on their explicit priority and submission time. The allocator first guarantees the required resource allocation for running jobs to minimize churn. If surplus resources remain, it allocates additional resources toward the job's expected demand on a best-effort basis, terminating the allocation cycle if the quota is exhausted. For jobs that receive a partial allocation, resources are preferentially assigned to smaller containers to maximize throughput.

Asynchronous Scheduling Framework. A key design to achieving control plane scalability is the full adoption of asynchronous principles. In a synchronous model, the three core operations of the SchedulerBackend: *refill*, *grant*, and *invoke*, are tightly coupled; a block in the *grant* operation, for instance, would stall the *invoke* of tasks that already possess granted quotas. To prevent this single point of blocking, Quark meticulously decouples these three operations into independent threads that communicate via the Slot Ring.

The *Refill* Thread generates available slots and presents them to the *FGTS*. It operates in a batched, asynchronous manner (e.g., every 500 ms), filling empty slots in the ring. The *FGTS* selects tasks based purely on priority, as the Slot Ring already controls parallelism (O(N) scheduling). The *Grant* Thread periodically (e.g., every 3 seconds) computes the aggregated resource demands of tasks in the Slot Ring and consults the *QM* for quota allocation. To reduce the load on the *QM*, it encodes the expected resource demand into a signature; if this signature remains unchanged from the previous

cycle, the expensive full allocation schedule is bypassed. Finally, the *Invoke* Thread handles the actual submission to the cluster gateway. It scans the Slot Ring every 500ms, identifies tasks that have received a valid quota grant, and dispatches the launch requests for parallel execution. This asynchronous design ensures that latency in one operation does not halt the entire task scheduling pipeline.

4.2 Interference-aware Scheduler

Quark introduces an interference-aware scheduler to overcome the heterogeneity and dynamic interference challenge, including two core techniques: the *Resource Normalizer* and the *Variance-Optimal Scheduler*.

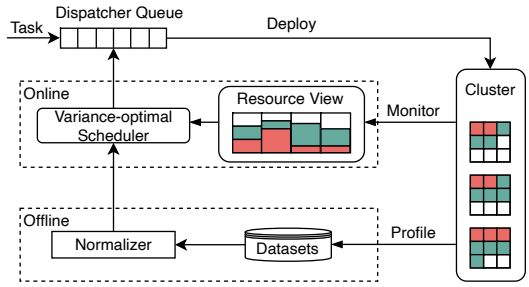


Figure 10: **Quark's Interference-aware Scheduler.**

Resource Normalizer. Since batch workloads have the lowest priority, they are susceptible to interference from both *prod* and *mid* workloads. The *Resource Normalizer* is designed to quantify the effective resource capacity available for batch on any given node. As shown in Figure 10, an agent continuously collects essential system metrics from kernel interfaces, including the CPU utilization of co-located workloads, NUMA topology, and the machine type. This data is fed into a linear model to calculate the effective capacity. Equation (1) quantifies the schedulable resource capacity on a node:

$$Batch_{\text{alloc}} = W_{\text{m}} \times \left(NodeAll - \sum_{i \in \{\text{sys,prod,mid,batch}\}} W_i \cdot Alloc_i \right) \quad (1)$$

Here, $NodeAll$ represents the total CPU cores of the node. W_{m} is a weight characterizing the intrinsic performance impact of the machine type (hardware heterogeneity). W_i is a weight representing the quantified impact of interference from workload type i (i.e., *sys*, *prod*, *mid*, and *batch*) on the batch execution, and $Alloc_i$ is the resources allocated to that workload. The weight-related parameters are determined offline through Bayesian optimization process, using a dataset built from comprehensive performance metrics collected across the cluster. The output, $Batch_{\text{alloc}}$ provides the scheduler with a unified metric of effective capacity. Cache contention is handled at the node via hardware QoS enforcement (e.g., Intel RDT, AMD QoS) with dynamically adjusted LLC quotas, which is orthogonal to scheduler-level interference model.

Variance-optimal Scheduler. After normalization, Quark then employs a variance optimization algorithm to make precise task placement decisions, ensuring optimal resource distribution across the cluster. Assume a cluster consists of N nodes, denoted as $node_1, node_2, \ldots, node_n$. The total resources of these nodes are $total_1, total_2, \ldots, total_n$, and at a given moment, their allocated resources are $alloc_1, alloc_2, \ldots, alloc_n$. The optimization goal is to minimize the variance of the nodes' allocation ratios after scheduling, The objective function is defined as:

$$Var = \frac{\sum_{i=1}^{n} \left(\frac{alloc_i}{total_i} - \overline{\frac{alloc}{total}} \right)^2}{N} \quad (2)$$

By minimizing this variance, the scheduler actively avoids creating load imbalances, ensuring that the workload is spread evenly relative to the effective capacity of each node.

4.3 Fast Task Provision

When a task is scheduled, the underlying execution platform must launch an instance to execute it, involving the startup of the *JVM*, the initialization of components within *SparkEnv*, and the preparation of the *ExecutorBackend*. In Spark's traditional model, the executor is a long-lived process, amortizing this multi-second startup cost across thousands of tasks. In contrast, when the execution granularity shifts to the task level, this startup delay must be incurred for every task, leading to a dramatic increase in cumulative cold-start overhead. To make the fine-grained model viable, Quark implements three mechanisms to achieve near-instantaneous task provisioning.

State Reuse via Forking. For states that are common across all tasks and independent of specific job logic, Quark employs a technique inspired by container optimization: it maintains a pre-initialized, paused template instance. This template instance completes the most expensive initializations in advance, including the JVM startup and the setup of core Spark components like RpcEnv and RemoteDataManager. When a new task instance is required, the system creates a live instance by forking from this template. Quark does not directly call fork() from an arbitrary multi-threaded JVM state. Instead, it leverages *sfork* (sandbox fork) design in previous work [17, 28], which introduces a transient single-thread mechanism: before forking, the template VM is brought into a controlled state where only one thread remains runnable and all other threads are quiesced. This avoids the well-known POSIX fork hazards in multi-threaded processes. After a new secure-container instance is forked, Quark will re-initialize the task-/application-specific Spark settings e.g., re-binding/re-initializing components such as RpcEnv endpoints and ShuffleManager connections. When new Spark versions are released, Quark periodically regenerates templates from the updated image. This mechanism allows the new instance to instantly inherit a initialized execution environment, eliminating seconds of overhead from the critical path, as detailed for the JVM and RemoteDataManager in Table 4.

State Pre-Prepare. The second mechanism addresses components that are task-specific but can be prepared outside

Table 4: **Cold Start Costs and Our Optimizations.**

Modules	Cost	Policy
WholeStageCodegen	70ms	Pre-Prepare
JVM	$2 \sim 3s$	Reuse
RemoteDataManager	600ms	Reuse
BlockManager	10ms	Reuse
RpcEnv	600ms	Reuse + Update
ShuffleManager	300ms	Reuse + Update
MetricsSystem	400ms	Lazy-Load
MapOutputTracker	300ms	Lazy-Load
SecurityManager	10ms	Lazy-Load
ExternalShuffleClient	100ms	Lazy-Load
NettyBlockTransferService	30ms	Lazy-Load

the critical launch path. A key target is Spark's Whole-Stage Code Generation (Codegen), which dynamically compiles optimized JVM bytecode at runtime to boost query performance. We leverage the insight that the code blocks generated by the driver for compilation are identical to those needed by the executor. The FGSB capitalizes on this by sending the compiled code from the driver directly to the executor instance via the serverless invocation payload. The task can then skip the costly process of code generation and compilation, directly loading the required classes, thereby reducing latency by an amount equivalent to the codegen time.

State Lazy-Load. Finally, Quark utilizes asynchronous loading and dependency removal for non-critical components. For states that can be initialized in parallel or are not immediately required for task execution, e.g., the security manager, Quark removes their initialization from the critical task startup path. As summarized in Table 4, this policy covers several heavyweight components like MetricsSystem, where initialization can be safely delayed, ensuring that task execution begins with minimal latency.

5 Implementation

Quark is implemented on top of Apache Spark. It remains fully compatible with Spark and requires no modifications to user code. To achieve Quark, there are approximately 10K lines of Java and Scala code for *FGTS* and *FGSB*, 5K lines for *QM*, 3K lines of Java for QueueManager, and 1500 lines of Go for cluster scheduler.

QueueManager. Beyond the design components, Quark introduces a *QueueManager (QMgr)* to control the maximum number of jobs that can be submitted. When a user submits a SparkSQL job, the system consults the *QM* to check whether the project's quota is sufficient. If the remaining quota is insufficient, the job is enqueued in the *QMgr*. Once enough quota becomes available, the *QMgr* dequeues and resubmits the job according to its priority and submission time.

Fault Tolerance. In Spark, when a task fails due to insufficient resources (e.g., out-of-memory), the Spark scheduler will retry it and reschedule it on another executor. However, because each executor has the same fixed resource configuration (e.g., CPU and memory), the retried task is likely to fail again. Quark's fine-grained job model can overcome the issues and enable precise fault-tolerance control: individual task failures do not affect other tasks, and the system can dynamically adjust resources and retry failed tasks as needed. As shown in Figure 11, Quark employs a three-level fault-tolerance mechanism tailored to the different stages at which failures may occur: (1) *Startup Monitor.* The *FGSB* monitors each task during startup. If a task fails to start within a predefined timeout, the *FGSB* marks the task as failed. (2) *Process Heartbeat.* While a task is running, it periodically sends heartbeat messages to the *FGSB*. If the *FGSB* does not receive a heartbeat within the expected interval, it treats the task as failed. (3) *Crash Callback.* If a task crashes, a node watcher detects the failure and reports the crash information to the *FGSB*, which then marks the task as failed.

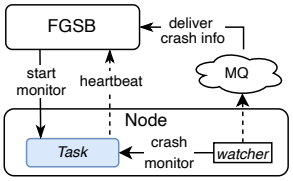


Figure 11: **Three Level Fault Tolerance.**

The *FGSB* then analyzes the failure context. If the failure is caused by insufficient resources (e.g., an out-of-memory error), it will apply an increased quota from the *QM*, and retries only the affected task, avoiding a full stage-wide retry.

6 Evaluation

6.1 End-to-end Performance

Testbeds. The evaluation is conducted on a 30-node cluster, with each node equipped with an Intel(R) Xeon(R) Platinum 8163 CPU (96 cores, 2.7 GHz). The operating system is Linux kernel 5.10. Each task instance gets 1 CPU core, 3 GB of memory, and 300 GB of storage. The evaluation takes the following systems as baselines.

1. **Spark** runs on Kubernetes (K8s) with a coarse-grained job model (version 3.2.0).
2. **Spark-F** represents an upgraded version of Spark; it adds a fine-grained resource model and runs in a serverless paradigm (include design in §4.1).
3. **Spark-S** adds the design of the interference-aware scheduler (§4.2) on the basis of Spark-F.
4. **Quark** is our final production platform that further optimizes task startup performance (§4.3) based on Spark-S.

Resource Efficiency. We first assess the resource utilization using the TPC-H benchmark [88], which includes 22 complex SQL queries covering a wide range of operations. Specifically, we measure the CU consumption of each SQL query over a 1 TB dataset (SF=1000) in four systems (1CU: 1CPU and 3GB memory for 1min in this context). As shown in Figure 12, Quark reduces resource consumption by 26.70%–87.86% compared to Spark across the 22 TPC-H queries, with an average reduction of 56.01% (20 trials, mean re-

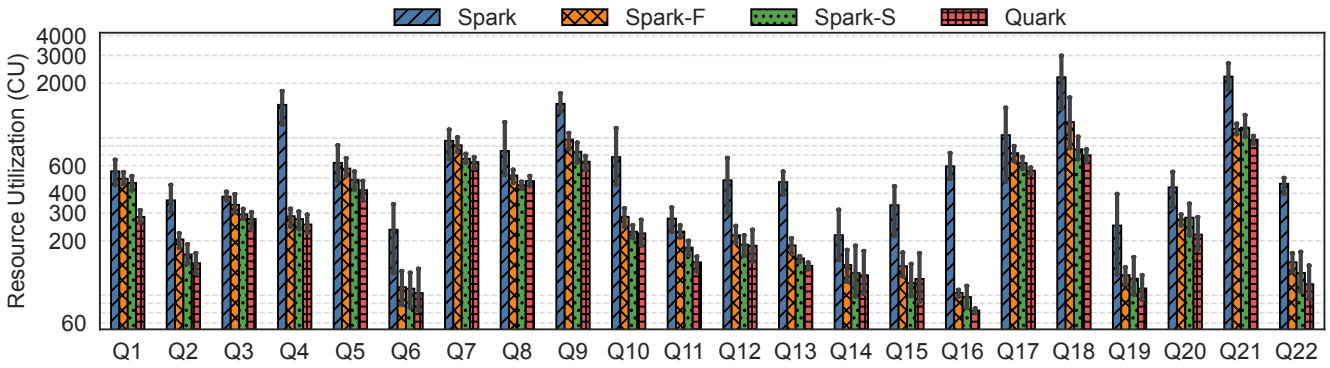


Figure 12: **Comparison of Resource Utilization in the TPC-H Benchmark.** Q1-Q22 are distinct TPC-H queries covering a broad range of SQL operations [89].

ported). This demonstrates the effectiveness of our approach. The smallest gain occurs in Q3, which involves joins and sorting on large tables (orders, lineitem) with heavy shuffle and memory use. Here, I/O is the main bottleneck, reducing the relative advantages of Quark's design. In contrast, Q16 shows the largest improvement, featuring filtering and group aggregation on small, highly parallelizable tables. For such workloads, Quark's scalable resource control, interference-aware scheduler, and fast task provisioning design are fully utilized, delivering significant gains.

Compared with Spark, Spark-F achieves a reduction in resource consumption ranging from 6.26% to 84.31%, with an average decrease of 42.68%. This improvement is attributed to fine-grained task scheduling, which minimizes resource waste caused by *Slot Idle* and *Stop Idle*. Similarly, Spark-S lowers resource consumption by 15.63% to 85.22%, averaging 50.26%, mainly through the adoption of an interference-aware scheduler that mitigates the *Gap Idle*.

Completion Time. To evaluate the job completion time, we design a compute-intensive job with no I/O operations. It consists of 800 tasks, each computing the Fibonacci sequence ($N = 50$) five times independently. Figure 13 compares the four systems in three aspects: resource utilization, job completion time, and task duration standard deviation.

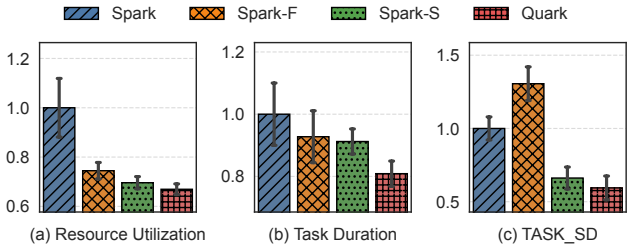


Figure 13: **Execution Time of Computation-Intensive Jobs.**

As shown in Figure 13(a), Spark-F, Spark-S, and Quark reduce resource consumption by 25.53%, 30.41%, and 33.06% compared to Spark. In terms of completion time, Figure 13(b) shows speedups of 7.24%, 8.78%, and 19.11%, respectively.

Notably, Figure 13(c) presents the standard deviation of task completion times for each system: native Spark exhibits an average standard deviation of 79.20s, while Spark-F, Spark-S, and Quark take 103.40s, 52.40s, and 47.07s, respectively. The slightly higher standard deviation of Spark-F compared to native Spark stems from its finer-grained scheduling, which operates at the task level rather than the executor level. In native Spark, each node allocates resources once at job startup. In contrast, Spark-F performs independent resource allocation for every task, increasing variability in startup overheads and thus leading to greater variation in task completion times. Spark-S and Quark substantially improve the consistency of task execution.

6.2 Micro-benchmarks

Interference-aware Scheduler. We construct a 21-node heterogeneous cluster, with machine models and counts listed in Table 5. To simulate realistic multi-service co-location, each node runs several online services: *Prod* services use 20%–30% of node resources, while *Mid* and *System* services each consume 5%–10%. To evaluate the effectiveness of interference-aware scheduler, a batch of computationally intensive tasks is submitted and their execution performance is assessed using two metrics: (1) *Average Task Time* and (2) *Tail Latency Ratio*, defined as the ratio of the longest to the shortest task run time within the same stage.

Table 5: **Cluster for Micro-benchmark.**

CPU Specifications	Number
Intel(R) Xeon(R) Platinum 8163 (24 cores, 2.5 GHz)	7
Intel(R) Xeon(R) Platinum 8269 CY (26 cores, 2.5 GHz)	7
AMD EPYC 7T83 (24 cores, 2.5 GHz)	7

Figure 14(a) presents the average task execution time within the cluster under varying load. After adopting an interference-aware scheduler, the average task execution time decreased from 250s to 191s (AR=70%) and from 306s to 286s (AR=100%), achieving a performance improvement of 18%–33%. Figure 14(b) illustrates the tail latency for the same stage of tasks. The tail duration ratio was reduced from 1.76× to 1.64× (AR=70%) and from 2.75× to 2.22×

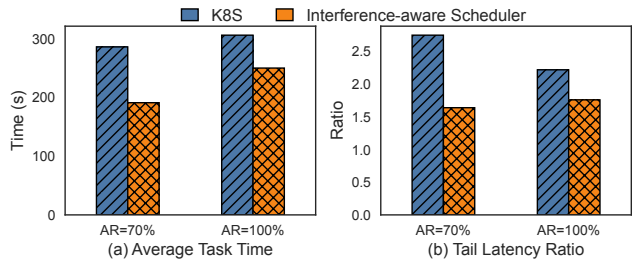


Figure 14: **Comparison of Task Execution Time and Tail Latency under Different Loads.** "AR" stands for allocation ratio. For example, a 100C cluster running 70 concurrent tasks (1C) has a 70% allocation ratio.

(AR=100%). This experiment demonstrates that interference-aware scheduler can significantly reduce both the average execution time and tail latency within a batch of tasks.

Fast Task Provision. Figure 15 compares the time to start an executor in Spark versus launching a task in Quark. Spark takes 6078*ms* to start an executor, while the corresponding task startup in Quark is only 626.53*ms* (100 trials, mean reported), attributed to the three startup optimization mechanisms in §4.3. Compared to SOTA results [17, 36, 37, 103], the remaining costs of Quark come from two aspects: first, the instance is launched under co-location with low-priority resources. In fact, Quark only takes around 100ms on a dedicated machine. Second, unlike typical simple serverless functions, Quark is monolithic and must initialize a lot of unique states, including driver/shuffle connection establishment, task-level code and plugin loading, and complicated structure initialization, which cannot be fully mitigated.

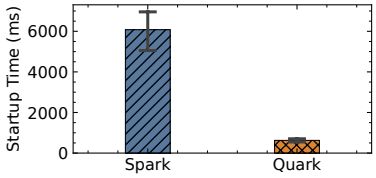


Figure 15: **Task Startup Performance.**

6.3 Production Workload Replay

To validate the effectiveness of Quark under real production workloads, we conduct a trace-driven replay experiment using 24-hour production traces captured from a project at Ant Group[1]. As shown in Figure 16, the traces comprise 22,532 Spark jobs exhibiting a typical diurnal pattern, with submissions peaking at 1,641 jobs/hour at 08:00 and declining to 630 jobs/hour at 21:00. The total I/O volume reaches 4,610 TB, dominated by warehouse reads (Tunnel I/O, 94.7%) with a 13.5:1 read-to-write ratio, while Shuffle I/O accounts for 5.3% with near-symmetric read/write.

To fairly compare Quark with unmodified Spark, we replay the same workloads on two equivalent-scale clusters, each

[1]The traces are available at https://github.com/antgroup/Quark

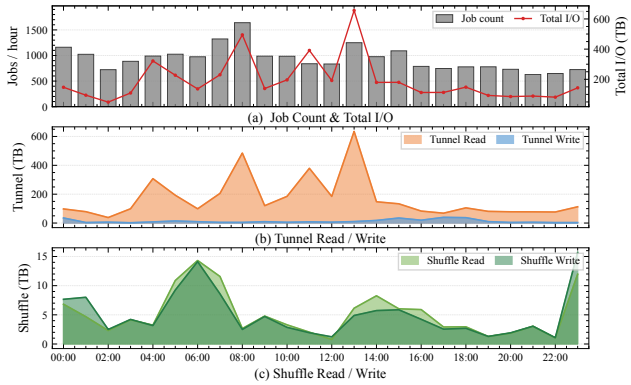


Figure 16: **24-hour Production Trace of Spark Workloads.** (a) Job count and total I/O volume. (b) Tunnel read/write throughput. (c) Shuffle read/write throughput.

consisting of 320 nodes equipped with Intel(R) Xeon(R) Platinum 8163 CPUs (96 cores, 2.7GHz), 512GB DRAM, and 3.52TB SSD. Figure 17(a) shows the CU distribution of each job, and Quark reduces the total CU consumption by 26.5% compared to Spark. Figure 17(b) further breaks down the total CU by cost tier. For jobs consuming 1K CU or above, Quark consistently achieves 22–29% savings across all tiers. A small fraction of jobs (3.4%) experience CU regression, primarily in the lightweight tier ($\leq$1K CU) due to the fixed overhead of function cold starts, which dominate the resource footprint for lightweight workloads. However, this tier accounts for only 1.0% of total CU, so its impact is negligible.

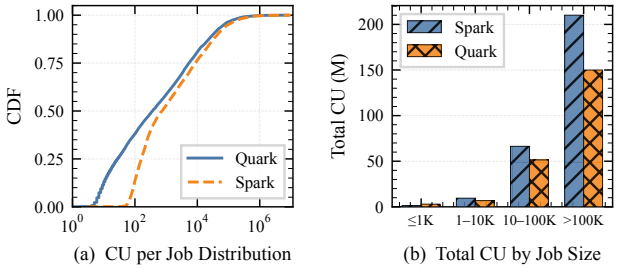
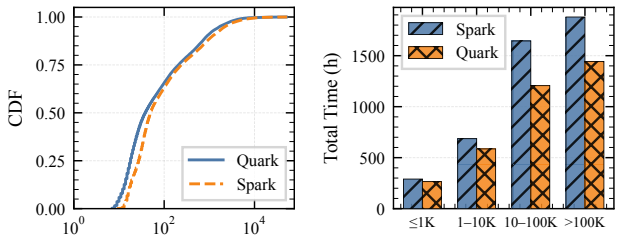


Figure 17: **Resource Consumption Comparison Between Spark and Quark.** (a) CDF of CU per job (log scale). (b) Total CU breakdown by job size.

Figure 18 shows that Quark reduces total execution time by 22.4% (3,501h vs. 4,501h). The time distribution mirrors CU consumption, except that the $\leq$1K tier has shorter execution time. This improvement stems from Quark's on-demand scheduling model. Unlike Spark, which stages jobs through a centralized scheduler and incurs queuing delays before resource allocation, Quark dispatches function invocations directly to available nodes, reducing scheduling cost.

6.4 In-Production Migration

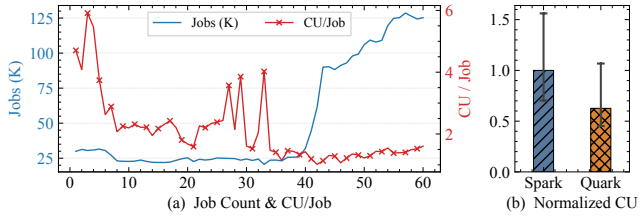
Following the successful replay experiment in §6.3, we proceed to migrate all production workloads from Spark to Quark.



(a) Execution Time per Job (s) (b) Total Time by Job Size

Figure 18: **Execution Time Comparison Between Spark and Quark.** (a) CDF of execution time per job (log scale). (b) Total execution time breakdown by job size.

This migration spans 57,000 tables and 350,000 jobs across 219 projects. This longitudinal deployment study captures real-world performance under actual user workloads, demonstrating Quark's operational effectiveness beyond controlled replay conditions.



(a) Job Count & CU/Job (b) Normalized CU

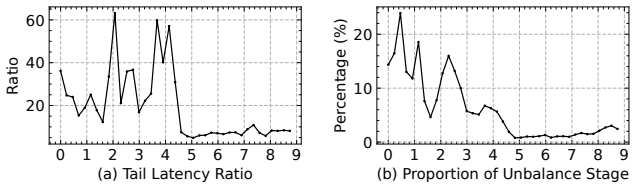
Figure 19: **Per-job Resource Consumption Trends During the Spark-to-Quark Migration.**

Per-job Resource Consumption. Figure 19(a) shows per-job resource consumption trends during this migration: Spark served production requests during days 0–40, followed by a complete cutover to Quark after day 40. The number of jobs increases (blue line) after migrating to Quark, while the average resource consumption per job (red line) declines throughout the migration. The normalized resource consumption in Figure 19(b) shows that Quark reduces resource idling during task execution, achieving 37.37% resource savings compared to Spark.

Job Tail Latency. We further evaluate the optimization effects in the production environment using two tail-performance metrics: (1) *Tail Latency Ratio*: within a stage, the ratio of the longest task runtime to the shortest; and (2) *Unbalanced Stage Proportion*: the fraction of stages where the longest task runtime exceeds 5× the average within that stage. As shown in Figure 20, compared to Spark, the average tail latency ratio was reduced from approximately 20× to 8× and the average unbalanced stage proportion decreased from approximately 15% to 2%. The interference-aware scheduler mitigates interference from co-located workloads and hardware heterogeneity, reducing latency variance and stage skew.

6.5 Operational Overview

Quark now operates as the batch workload processing engine at Ant Group. This section reports its operational metrics and

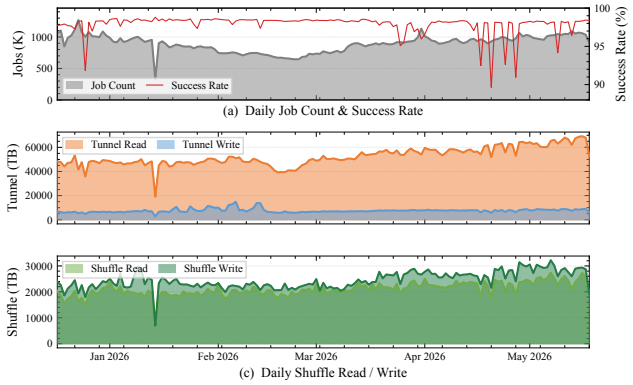


(a) Tail Latency Ratio (b) Proportion of Unbalance Stage

Figure 20: **Tail Latency Ratio and Unbalanced Stage Proportion Trends During the Spark-to-Quark Migration.**

production status over the past 6 months.

Scale and Reliability. As shown in Figure 21, Quark processes an average of 902K jobs per day (peaking at 1.27M) and achieves an overall success rate of 99.11%, meeting the strict SLA requirements of production batch workloads. The daily I/O volume averages 105.4 PB, dominated by tunnel reads (warehouse I/O) at 49.7% of total traffic (56.8 PB/day) and a 7.0:1 read-to-write ratio; shuffle I/O accounts for the remaining 43.2% (20.0% read, 23.2% write). These results demonstrate that Quark reliably meets production requirements at scale, sustaining high throughput and consistency over the entire 6-month period.



(a) Daily Job Count & Success Rate

(c) Daily Shuffle Read / Write

Figure 21: **Operational Overview of the Production Workload Over 6 Months.** (a) Daily job count and success rate. (b) Tunnel I/O. (c) Shuffle I/O.

Cluster Status. Quark is deployed on a production cluster comprising over 6,000 servers with approximately 600K CPU cores. The cluster employs resource overcommit with a ratio ranging from 1.592× to 1.803×, and achieves an overall utilization between 41.7% and 61.4%. Figure 22 shows the invocation rate over a representative 24-hour period. The system processes an average of 534 requests per second with a peak of 2,130 requests per second. The overall success rate (HTTP 200) is 98.55%. Error responses are dominated by rate limiting (HTTP 429), which spikes with traffic bursts.

We measure the overcommitted resource allocation ratio in the cluster to assess how evenly tasks are distributed across nodes. For example, if 100 cores are overcommitted in the cluster and 70 cores are in use, the allocation ratio is 70%. The black line in Figure 23(a) shows the mean overcommitted

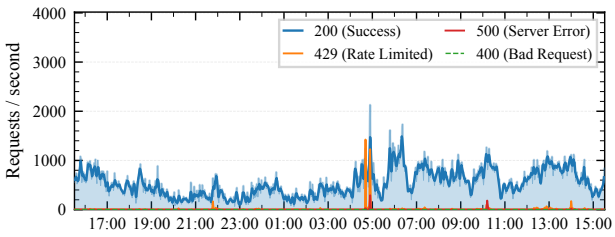


Figure 22: **Request Rate Over a 24-hour Period.**

resource allocation ratio over one day, with an average allocation ratio of 80.4%, red line shows the variance of allocation ratios across all nodes in the cluster, which remains stably low (the P95 is 22%), indicating good load balance. Figure 23(b) plots the CDF of the allocation ratio across all nodes at 00:00, showing that over 90% of nodes are higher than 69.2%.

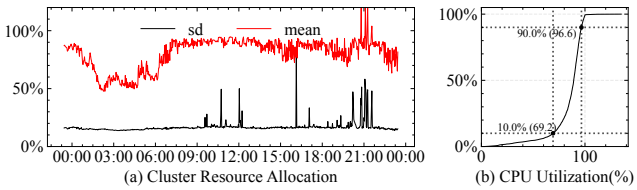


Figure 23: **Overcommitted Resource Allocation.**

7 Failure Analysis

In order to understand the operational situation and identify opportunities for improvement, we record 1.31 million failures over the past six months and classify these failures based on their root cause.

7.1 Failure Distribution

Table 6 summarizes the distribution of these failures. The most apparent is that the majority of recorded failures are user code, or permission issues rather than system-level problems.

Table 6: **Production Failures Distribution.**

Category	Count	%
User SQL / Query Errors	572K	44.0%
Permission / Access	163K	12.5%
Tunnel / Storage I/O	124K	9.5%
UDF / Environment	115K	8.9%
OOM / Resource Exhaustion	101K	7.8%
Driver / Coordinator	98K	7.5%
Shuffle / Compute Engine	78K	6.0%
Optimizer / Rule	47K	3.6%
Other / Unknown	3K	0.2%
Total	**1.31M**	**100%**

User-attributable Failures Dominate. User errors—including SQL mistakes, permission issues, UDF bugs, runtime environment mismatches, and Python worker failures—collectively account for 65.3% of all failures. SQL errors alone constitute 44.0%, making them the single largest category. Common sub-categories include: (1) references to non-existent tables or columns (the most frequent single error, with 296K occurrences); (2) user-initiated job cancellations (40K); (3) type mismatch and cast errors (25K); and (4) unsupported SQL syntax and features (e.g., `UNPIVOT`, `ALTER MATERIALIZED VIEW ... REBUILD`). Permission errors (12.5%) form the second-largest category, dominated by cross-project access restrictions (65K) and insufficient table-level permissions. UDF and environment errors (8.9%) encompass user-defined function loading and execution failures (23K), runtime environment incompatibilities (68K), and Python worker crashes (17K).

OOM Is the Leading System-Level Failure. Out-of-memory errors collectively account for 7.8% of failures. Table 7 breaks down OOM failures by type. Executor-level OOM (container crash or heartbeat timeout due to memory exhaustion) is the dominant sub-category at 47.9%. Sort and shuffle-write OOM (16.9%) and aggregate OOM (9.2%) follow. One benefit of Quark's fine-grained mode is that when a task fails with OOM error, it can be retried in-place with a larger resource footprint, rather than destabilizing a long-lived executor that may host many concurrent tasks. 94.7% of failed tasks succeed upon retry after Quark automatically increases the resource allocation. The remaining 5.3% are dominated by tasks with heavily skewed data partitions, which are inherently difficult to handle via online resource prediction alone. These failed retries also waste CUs, accounting for 35.2% of the total CUs consumed by retried tasks, but are negligible relative to the overall CU consumption of all tasks.

Table 7: **OOM Failure Sub-categories (101K total).**

OOM Type	%
Executor-level (crash / heartbeat timeout)	47.9%
Sort / Shuffle write	16.9%
Other (Java heap, spill disk full, unknown)	16.4%
Hash/Sort Aggregate	9.2%
Gluten/Velox native	9.1%
Join (Broadcast/SMJ/SHJ)	0.2%
Python-related	0.3%

I/O and Data Transfer Failures Are Transient but Frequent. Tunnel and storage-api errors account for 9.5% of failures. The most common sub-categories are quota/capacity limits (29.5% of tunnel errors, dominated by slot quota exceeded with 34K occurrences), schema and partition errors (18.1%), storage API timeouts (19.6%), and data read/write failures from the underlying distributed file system (10.4% and 9.9%). These failures are typically transient—caused by momentary network congestion, service restarts, or load spikes—and are well-served by Quark's built-in retry mechanism. However, the sheer volume highlights the importance of robust I/O error handling in production batch systems operating at scale.

Shuffle Reliability Remains a Concern. Shuffle-related failures (6.0%) include data-loss exceptions when shuffle services experience failures, Celeborn-related errors (connection timeouts, registration failures), and Velox native execution errors. Unlike I/O timeouts, shuffle data loss can cascade: a single lost shuffle partition may cause an entire stage to fail, requiring re-execution of all upstream tasks. Quark's task-level fault isolation limits the blast radius compared to Spark's executor-level model, but shuffle service reliability remains a critical dependency.

7.2 Operational Lessons

Use Secure Containers for Fault and Performance Isolation. Without proper isolation, these failures above (e.g., OOM crashes, transient I/O errors) can cascade across tasks. Morever, batch workloads are both CPU- and I/O-intensive, and their heavy use of kernel resources (e.g., scheduling and mutexes) can interfere with prod. Using secure containers to isolate both fault impact and performance impact is a good practice. We use SKernel [16] as the backbone of Quark, it provides comparable performance to runc (without virtualization-based isolation) [69].

Explore User-Provided Hints for Long-Tail OOM Reduction. Quark employs a history-based query optimizer that predicts memory requirements at submission time, resolving 94.7% of OOM-induced failures upon retry with increased resources. However, the remaining 5.3%—dominated by long-tail tasks with highly skewed data or non-stationary demands—remain difficult to predict purely from history. Prior work [50] on feedback-driven resource estimation shows that achieving stable prediction for such tasks may require many iterations (≈20) of adjustment. User-provided hints (e.g., expected data skew, memory-heavy operators) may help short-circuit this feedback loop where historical signal is insufficient.

Design for Transient I/O Failures at Scale. At the scale of 30K+ daily jobs processing petabytes of data, transient I/O errors are inevitable rather than exceptional. Systems must treat retries as a first-class mechanism: Quark's task-level model, where a failed task is independently retried without affecting other tasks, is well-suited to this environment. By contrast, in Spark's executor-level model, a transient I/O error may invalidate an entire executor's in-memory state.

8 Related Work

Cluster Resource Optimization. Cluster resource optimization focuses on maximizing resource utilization efficiency in clusters, including cluster management [21, 25, 35, 60, 63, 67, 78, 86, 91, 92, 98] and schedulers [14, 23, 33, 42, 47, 93, 107, 108]. Cluster platforms such as Azure [21, 80], Google [76, 87], Alibaba [20, 34, 55, 58] have published traces of their cluster management systems, workload characteristics and optimization strategies. Co-locate low-priority batch workloads with high-priority online services is a common practice to improve overall resource utilization [38, 52, 57, 64, 77, 106]. However, existing work mainly focuses on improving cluster resource utilization without violating SLOs and pays little attention to the *effectiveness* of resource usage because batch workloads are typically treated as "best-effort", and resources from overcommitment are often viewed as a free lunch. In this paper, we identify four types of idle or non-productive costs that significantly undermine effective utilization and propose a lift serverless paradigm for efficient co-location.

Serverless Analytics. In addition to the systems discussed in §2.3, many other works (e.g., PyWren [43], Flint [45],Locus [75], Lambada [66], Cloudburst [84]) also leverage serverless platforms for batch analytics. Existing work mainly focuses on demonstrating the feasibility of serverless for offline analytics, and on improving shuffle operation efficiency on serverless platforms. Notably, Ditto [41] can adjust the degree of parallelism of each stage to optimize job completion time. However, Quark differs from these efforts by introducing a fine-grained job model and targeting production co-location with practical, deployment-oriented optimizations to overcome challenges in §3.1. Quark has been validated for effectiveness in industrial deployments.

Optimizations for Cold Start. We observe encouraging advances in serverless cold-start optimization techniques [1, 4, 15, 27, 28, 37, 48, 51, 62, 68, 71, 82, 90]. E.g., recent research efforts [17, 28, 37, 51, 96, 103] have reduced function cold-start latency to within a 10-milliseconds. These works inspired the fast task provision design in Quark. However, compared with existing serverless platform, Spark is more complex and involves many operations whose state cannot be reused, such as additional plugin loading and establishing connections (e.g., with the Driver and ShuffleManager). The startup latency of a single task in Quark has not yet been optimized to within 10 ms, but it is sufficient for our needs.

9 Conclusion

This paper addresses the inefficiencies of co-location in modern cloud infrastructures, focusing on industrial-scale deployment. Through real-world deployments and in-depth analysis, we identify four types of idle or non-productive costs that significantly undermine effective utilization and leave clusters in a *pretending to be busy* state. This paper presents Quark, applying serverless paradigms to batch workloads for efficient co-location. Results show that Quark increases cluster resource utilization by 37.37%, and has been deployed in large-scale clusters, saving more than 100,000 CPU cores. We sincerely thank our shepherd and the anonymous reviewers, whose reviews and suggestions greatly strengthened our work. This work was supported by Smart Gird-National Science and Technology Major Project (No.2024ZD0803000) and National Natural Science Foundation of China (No.62302300). Correspondence to: Dong Du (Dd_nirvana@sjtu.edu.cn), Yu Chen (yuchen@tsinghua.edu.cn).

References

[1] Alexandru Agache, Marc Brooker, Alexandra Iordache, Anthony Liguori, Rolf Neugebauer, Phil Piwonka, and Diana-Maria Popa. Firecracker: Lightweight Virtualization for Serverless Applications. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 419–434, Santa Clara, CA, February 2020. USENIX Association. URL: https://www.usenix.org/conference/nsdi20/presentation/agache.

[2] Ganesh Ananthanarayanan, Srikanth Kandula, Albert Greenberg, Ion Stoica, Yi Lu, Bikas Saha, and Edward Harris. Reining in the outliers in map-reduce clusters using Mantri. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, OSDI'10, page 265–278, USA, 2010. USENIX Association. URL: https://dl.acm.org/doi/10.5555/1924943.1924962.

[3] Lixiang Ao, Liz Izhikevich, Geoffrey M. Voelker, and George Porter. Sprocket: A Serverless Video Processing Framework. In *Proceedings of the ACM Symposium on Cloud Computing*, SoCC '18, page 263–274, New York, NY, USA, 2018. Association for Computing Machinery. doi:10.1145/3267809.3267815.

[4] Lixiang Ao, George Porter, and Geoffrey M. Voelker. FaaSnap: FaaS made fast using snapshot-based VMs. In *Proceedings of the Seventeenth European Conference on Computer Systems*, EuroSys '22, page 730–746, New York, NY, USA, 2022. Association for Computing Machinery. doi:10.1145/3492321.3524270.

[5] Apache Software Foundation. Apache Flink: Open Source Stream Processing Framework. https://flink.apache.org/. Accessed: 2025-11-07.

[6] Apache Spark Team. CoarseGrainedSchedulerBackend in Spark. https://downloads.apache.org/spark/docs/1.2.0/api/java/org/apache/spark/scheduler/cluster/CoarseGrainedSchedulerBackend.html, 2014. Accessed: 2025-12-11.

[7] Apache Spark Team. SPARK-27495: SPIP: Support stage level resource configuration and scheduling. https://issues.apache.org/jira/browse/SPARK-27495, 2022. Accessed: 2025-11-19.

[8] Apache Spark Team. Cluster Mode Overview. https://spark.apache.org/docs/latest/cluster-overview.html, 2024. Accessed: 2025-12-11.

[9] Apache Spark Team. Running Spark on Kubernetes. https://spark.apache.org/docs/latest/running-on-kubernetes.html, 2024. Accessed: 2025-12-11.

[10] Apache Spark Team. SparkEnv in Spark. https://spark.apache.org/docs/latest/api/java/org/apache/spark/SparkEnv.html, 2024. Accessed: 2025-12-06.

[11] Apache Spark Team. Apache Spark. http://spark.apache.org, 2025. A unified engine for large-scale data analytics. Accessed: 2025-12-11.

[12] Michael Armbrust, Reynold S. Xin, Cheng Lian, Yin Huai, Davies Liu, Joseph K. Bradley, Xiangrui Meng, Tomer Kaftan, Michael J. Franklin, Ali Ghodsi, and Matei Zaharia. Spark SQL: Relational Data Processing in Spark. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, page 1383–1394, New York, NY, USA, 2015. Association for Computing Machinery. doi:10.1145/2723372.2742797.

[13] Noman Bashir, Nan Deng, Krzysztof Rzadca, David Irwin, Sree Kodak, and Rohit Jnagal. Take it to the limit: peak prediction-driven resource overcommitment in datacenters. In *Proceedings of the Sixteenth European Conference on Computer Systems*, EuroSys '21, page 556–573, New York, NY, USA, 2021. Association for Computing Machinery. doi:10.1145/3447786.3456259.

[14] Eric Boutin, Jaliya Ekanayake, Wei Lin, Bing Shi, Jingren Zhou, Zhengping Qian, Ming Wu, and Lidong Zhou. Apollo: Scalable and Coordinated Scheduling for Cloud-Scale Computing. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 285–300, Broomfield, CO, October 2014. USENIX Association. URL: https://www.usenix.org/conference/osdi14/technical-sessions/presentation/boutin.

[15] James Cadden, Thomas Unger, Yara Awad, Han Dong, Orran Krieger, and Jonathan Appavoo. SEUSS: skip redundant paths to make serverless fast. In *Proceedings of the Fifteenth European Conference on Computer Systems*, EuroSys '20, New York, NY, USA, 2020. Association for Computing Machinery. doi:10.1145/3342195.3392698.

[16] Xiaohu Chai, Keyang Hu, Jianfeng Tan, Tiwei Bie, Guotao Tan, Tianyu Zhou, Anqi Shen, Dawei Shen, Xinyao Yang, Xin Chen, Xu Wang, Feng Yu, Zhengyu He, Dong Du, Yubin Xia, Kang Chen, and Yu Chen. SKernel: An Elastic and Efficient Secure Container System at Scale with a Split-Kernel Architecture. In

Proceedings of the 21st European Conference on Computer Systems, EUROSYS '26, page 605–623, New York, NY, USA, 2026. Association for Computing Machinery. doi:10.1145/3767295.3769332.

[17] Xiaohu Chai, Tianyu Zhou, Keyang Hu, Jianfeng Tan, Tiwei Bie, Anqi Shen, Dawei Shen, Qi Xing, Shun Song, Tongkai Yang, Le Gao, Feng Yu, Zhengyu He, Dong Du, Yubin Xia, Kang Chen, and Yu Chen. Fork in the Road: Reflections and Optimizations for Cold Start Latency in Production Serverless Systems. In *19th USENIX Symposium on Operating Systems Design and Implementation (OSDI 25)*, pages 199–218, Boston, MA, July 2025. USENIX Association. URL: https://www.usenix.org/conference/osdi25/presentation/chai-xiaohu.

[18] Guoli Cheng, Shi Ying, and Bingming Wang. Tuning configuration of apache spark on public clouds by combining multi-objective optimization and performance prediction model. *J. Syst. Softw.*, 180:111028, 2021. URL: https://doi.org/10.1016/j.jss.2021.111028, doi:10.1016/J.JSS.2021.111028.

[19] Yue Cheng, Ali Anwar, and Xuejing Duan. Analyzing Alibaba's Co-located Datacenter Workloads. In *2018 IEEE International Conference on Big Data (Big Data)*, pages 292–297, 2018. doi:10.1109/BigData.2018.8622518.

[20] Yue Cheng, Zheng Chai, and Ali Anwar. Characterizing Co-located Datacenter Workloads: An Alibaba Case Study. In *Proceedings of the 9th Asia-Pacific Workshop on Systems*, APSys '18, New York, NY, USA, 2018. Association for Computing Machinery. doi:10.1145/3265723.3265742.

[21] Eli Cortez, Anand Bonde, Alexandre Muzio, Mark Russinovich, Marcus Fontoura, and Ricardo Bianchini. Resource Central: Understanding and Predicting Workloads for Improved Resource Management in Large Cloud Platforms. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, page 153–167, New York, NY, USA, 2017. Association for Computing Machinery. doi:10.1145/3132747.3132772.

[22] Binyang Dai, Qichen Wang, and Ke Yi. SparkSQL+: Next-generation Query Planning over Spark. In *Companion of the 2023 International Conference on Management of Data*, SIGMOD '23, page 115–118, New York, NY, USA, 2023. Association for Computing Machinery. doi:10.1145/3555041.3589715.

[23] Christina Delimitrou and Christos Kozyrakis. Paragon: QoS-aware scheduling for heterogeneous datacenters. *SIGPLAN Not.*, 48(4):77–88, March 2013. doi:10.1145/2499368.2451125.

[24] Christina Delimitrou and Christos Kozyrakis. Paragon: QoS-aware scheduling for heterogeneous datacenters. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '13, page 77–88, New York, NY, USA, 2013. Association for Computing Machinery. doi:10.1145/2451116.2451125.

[25] Christina Delimitrou and Christos Kozyrakis. Quasar: resource-efficient and QoS-aware cluster management. *SIGPLAN Not.*, 49(4):127–144, February 2014. doi:10.1145/2644865.2541941.

[26] Chen Ding, Sicen Li, Kai Lu, Ting Yao, Daohui Wang, Huatao Wu, Jiguang Wan, Zhihu Tan, and Changsheng Xie. DShuffle: DPU-Optimized shuffle framework for large-scale data processing. In *2025 USENIX Annual Technical Conference (USENIX ATC 25)*, pages 1371–1386, Boston, MA, July 2025. USENIX Association. URL: https://www.usenix.org/conference/atc25/presentation/ding.

[27] Dong Du, Qingyuan Liu, Xueqiang Jiang, Yubin Xia, Binyu Zang, and Haibo Chen. Serverless computing on heterogeneous computers. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '22, page 797–813, New York, NY, USA, 2022. Association for Computing Machinery. doi:10.1145/3503222.3507732.

[28] Dong Du, Tianyi Yu, Yubin Xia, Binyu Zang, Guanglu Yan, Chenggang Qin, Qixuan Wu, and Haibo Chen. Catalyzer: Sub-millisecond Startup for Serverless Computing with Initialization-less Booting. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '20, page 467–481, New York, NY, USA, 2020. Association for Computing Machinery. doi:10.1145/3373376.3378512.

[29] Gregory Essertel, Ruby Tahboub, James Decker, Kevin Brown, Kunle Olukotun, and Tiark Rompf. Flare: Optimizing Apache Spark with Native Compilation for Scale-Up Architectures and Medium-Size Data. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 799–815, Carlsbad, CA, October 2018. USENIX Association. URL: https://www.usenix.org/conference/osdi18/presentation/essertel.

[30] Sadjad Fouladi, Francisco Romero, Dan Iter, Qian Li, Shuvo Chatterjee, Christos Kozyrakis, Matei Za-

haria, and Keith Winstein. From Laptop to Lambda: Outsourcing Everyday Jobs to Thousands of Transient Functional Containers. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 475–488, Renton, WA, July 2019. USENIX Association. URL: http://www.usenix.org/conference/atc19/presentation/fouladi.

[31] Sadjad Fouladi, Riad S. Wahby, Brennan Shacklett, Karthikeyan Vasuki Balasubramaniam, William Zeng, Rahul Bhalerao, Anirudh Sivaraman, George Porter, and Keith Winstein. Encoding, Fast and Slow: Low-Latency Video Processing Using Thousands of Tiny Threads. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pages 363–376, Boston, MA, March 2017. USENIX Association. URL: https://www.usenix.org/conference/nsdi17/technical-sessions/presentation/fouladi.

[32] Sukhpal Singh Gill, Xue Ouyang, and Peter Garraghan. Tails in the cloud: a survey and taxonomy of straggler management within large-scale cloud data centres. *J. Supercomput.*, 76(12):10050–10089, December 2020. doi:10.1007/s11227-020-03241-x.

[33] Ionel Gog, Malte Schwarzkopf, Adam Gleave, Robert N. M. Watson, and Steven Hand. Firmament: Fast, Centralized Cluster Scheduling at Scale. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 99–115, Savannah, GA, November 2016. USENIX Association. URL: https://www.usenix.org/conference/osdi16/technical-sessions/presentation/gog.

[34] Jing Guo, Zihao Chang, Sa Wang, Haiyang Ding, Yihui Feng, Liang Mao, and Yungang Bao. Who Limits the Resource Efficiency of My Datacenter: An Analysis of Alibaba Datacenter Traces. In *2019 IEEE/ACM 27th International Symposium on Quality of Service (IWQoS)*, pages 1–10, 2019. doi:10.1145/3326285.3329074.

[35] Benjamin Hindman, Andy Konwinski, Matei Zaharia, Ali Ghodsi, Anthony D. Joseph, Randy Katz, Scott Shenker, and Ion Stoica. Mesos: A Platform for Fine-Grained Resource Sharing in the Data Center. In *8th USENIX Symposium on Networked Systems Design and Implementation (NSDI 11)*, Boston, MA, March 2011. USENIX Association. URL: https://dl.acm.org/doi/10.5555/1972457.1972488.

[36] Ben Holmes, Baltasar Dinis, Lana Honcharuk, Joshua Fried, and Adam Belay. Taming Serverless Cold Starts Through OS Co-Design, 2025. URL: https://arxiv.org/abs/2509.14292, arXiv:2509.14292.

[37] Jialiang Huang, MingXing Zhang, Teng Ma, Zheng Liu, Sixing Lin, Kang Chen, Jinlei Jiang, Xia Liao, Yingdi Shan, Ning Zhang, Mengting Lu, Tao Ma, Haifeng Gong, and YongWei Wu. TrEnv: Transparently Share Serverless Execution Environments Across Different Functions and Nodes. In *Proceedings of the 30th ACM Symposium on Operating Systems Principles (SOSP)*, 2024. doi:10.1145/3694715.3695967.

[38] Pawel Janus and Krzysztof Rzadca. SLO-aware colocation of data center tasks based on instantaneous processor requirements. In *Proceedings of the 2017 Symposium on Cloud Computing*, SoCC '17, page 256–268, New York, NY, USA, 2017. Association for Computing Machinery. doi:10.1145/3127479.3132244.

[39] Zhipeng Jia and Emmett Witchel. Nightcore: efficient and scalable serverless computing for latency-sensitive, interactive microservices. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '21, page 152–166, New York, NY, USA, 2021. Association for Computing Machinery. doi:10.1145/3445814.3446701.

[40] Congfeng Jiang, Yitao Qiu, Weisong Shi, Zhefeng Ge, Jiwei Wang, Shenglei Chen, Christophe Cérin, Zujie Ren, Guoyao Xu, and Jiangbin Lin. Characterizing Co-Located Workloads in Alibaba Cloud Datacenters. *IEEE Transactions on Cloud Computing*, 10(4):2381–2397, 2022. doi:10.1109/TCC.2020.3034500.

[41] Chao Jin, Zili Zhang, Xingyu Xiang, Songyun Zou, Gang Huang, Xuanzhe Liu, and Xin Jin. Ditto: Efficient Serverless Analytics with Elastic Parallelism. In *Proceedings of the ACM SIGCOMM 2023 Conference*, ACM SIGCOMM '23, page 406–419, New York, NY, USA, 2023. Association for Computing Machinery. doi:10.1145/3603269.3604816.

[42] Tatiana Jin, Zhenkun Cai, Boyang Li, Chengguang Zheng, Guanxian Jiang, and James Cheng. Improving resource utilization by timely fine-grained scheduling. In *Proceedings of the Fifteenth European Conference on Computer Systems*, EuroSys '20, New York, NY, USA, 2020. Association for Computing Machinery. doi:10.1145/3342195.3387551.

[43] Eric Jonas, Qifan Pu, Shivaram Venkataraman, Ion Stoica, and Benjamin Recht. Occupy the cloud: distributed computing for the 99%. In *Proceedings of the 2017 Symposium on Cloud Computing*, SoCC '17, page 445–451, New York, NY, USA, 2017. Association for Computing Machinery. doi:10.1145/3127479.3128601.

[44] Anurag Khandelwal, Yupeng Tang, Rachit Agarwal, Aditya Akella, and Ion Stoica. Jiffy: elastic far-memory for stateful serverless analytics. In *Proceedings of the Seventeenth European Conference on Computer Systems*, EuroSys '22, page 697–713, New York, NY, USA, 2022. Association for Computing Machinery. doi:10.1145/3492321.3527539.

[45] Youngbin Kim and Jimmy Lin. Serverless Data Analytics with Flint, 2018. URL: https://arxiv.org/abs/1803.06354, arXiv:1803.06354.

[46] Danny Krizanc and Anton Saarimaki. Bulk Synchronous Parallel: Practical Experience with a Model for Parallel Computing. In *Proceedings of the 1996 Conference on Parallel Architectures and Compilation Techniques*, PACT '96, page 208, USA, 1996. IEEE Computer Society. URL: https://ieeexplore.ieee.org/document/552669.

[47] Neeraj Kumar, Pol Mauri Ruiz, Vijay Menon, Igor Kabiljo, Mayank Pundir, Andrew Newell, Daniel Lee, Liyuan Wang, and Chunqiang Tang. Optimizing resource allocation in hyperscale datacenters: scalability, usability, and experiences. In *Proceedings of the 18th USENIX Conference on Operating Systems Design and Implementation*, OSDI'24, USA, 2024. USENIX Association. URL: https://www.usenix.org/conference/osdi24/presentation/kumar.

[48] Nikita Lazarev, Varun Gohil, James Tsai, Andy Anderson, Bhushan Chitlur, Zhiru Zhang, and Christina Delimitrou. Sabre: Hardware-Accelerated Snapshot Compression for Serverless MicroVMs. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, pages 1–18, Santa Clara, CA, July 2024. USENIX Association. URL: https://www.usenix.org/conference/osdi24/presentation/lazarev.

[49] Tao Li, Yongkun Li, Wenzhe Zhu, Yinlong Xu, and John C. S. Lui. MinFlow: high-performance and cost-efficient data passing for I/O-intensive stateful serverless analytics. In *Proceedings of the 22nd USENIX Conference on File and Storage Technologies*, FAST '24, USA, 2024. USENIX Association. URL: https://www.usenix.org/conference/fast24/presentation/li.

[50] Yang Li, Huaijun Jiang, Yu Shen, Yide Fang, Xiaofeng Yang, Danqing Huang, Xinyi Zhang, Wentao Zhang, Ce Zhang, Peng Chen, and Bin Cui. Towards General and Efficient Online Tuning for Spark. *Proc. VLDB Endow.*, 16(12):3570–3583, August 2023. doi:10.14778/3611540.3611548.

[51] Zijun Li, Jiagan Cheng, Quan Chen, Eryu Guan, Zizheng Bian, Yi Tao, Bin Zha, Qiang Wang, Weidong Han, and Minyi Guo. RunD: A Lightweight Secure Container Runtime for High-density Deployment and High-concurrency Startup in Serverless Computing. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, pages 53–68, Carlsbad, CA, July 2022. USENIX Association. URL: https://www.usenix.org/conference/atc22/presentation/li-zijun-rund.

[52] Yi Liang, Shaokang Zeng, and Lei Wang. Quantifying Resource Contention of Co-located Workloads with the System-level Entropy. *ACM Trans. Archit. Code Optim.*, 20(1), February 2023. doi:10.1145/3563696.

[53] Jinkun Lin, Ziheng Jiang, Zuquan Song, Sida Zhao, Menghan Yu, Zhanghan Wang, Chenyuan Wang, Zuocheng Shi, Xiang Shi, Wei Jia, Zherui Liu, Shuguang Wang, Haibin Lin, Xin Liu, Aurojit Panda, and Jinyang Li. Understanding stragglers in large model training using what-if analysis. In *Proceedings of the 19th USENIX Conference on Operating Systems Design and Implementation*, OSDI '25, USA, 2025. USENIX Association. URL: https://www.usenix.org/conference/osdi25/presentation/lin-jinkun.

[54] Qingyuan Liu, Yanning Yang, Dong Du, Yubin Xia, Ping Zhang, Jia Feng, James R. Larus, and Haibo Chen. Harmonizing Efficiency and Practicability: Optimizing Resource Utilization in Serverless Computing with Jiagu. In *2024 USENIX Annual Technical Conference (USENIX ATC 24)*, pages 1–17, Santa Clara, CA, July 2024. USENIX Association. URL: https://www.usenix.org/conference/atc24/presentation/liu-qingyuan.

[55] Qixiao Liu and Zhibin Yu. The Elasticity and Plasticity in Semi-Containerized Co-locating Cloud Workload: a View from Alibaba Trace. In *Proceedings of the ACM Symposium on Cloud Computing*, SoCC '18, page 347–360, New York, NY, USA, 2018. Association for Computing Machinery. doi:10.1145/3267809.3267830.

[56] Google LLC. Site Reliability Engineering: Service Level Objectives. https://sre.google/sre-book/service-level-objectives/, 2024. Accessed: 2024-01-17.

[57] David Lo, Liqun Cheng, Rama Govindaraju, Parthasarathy Ranganathan, and Christos Kozyrakis. Heracles: improving resource efficiency at scale. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture*,

ISCA '15, page 450–462, New York, NY, USA, 2015. Association for Computing Machinery. doi:10.1145/2749469.2749475.

[58] Chengzhi Lu, Kejiang Ye, Guoyao Xu, Cheng-Zhong Xu, and Tongxin Bai. Imbalance in the cloud: An analysis on Alibaba cluster trace. In *2017 IEEE International Conference on Big Data (Big Data)*, pages 2884–2892, 2017. doi:10.1109/BigData.2017.8258257.

[59] Frank Sifei Luan, Stephanie Wang, Samyukta Yagati, Sean Kim, Kenneth Lien, Isaac Ong, Tony Hong, Sangbin Cho, Eric Liang, and Ion Stoica. Exoshuffle: An Extensible Shuffle Architecture. In *Proceedings of the ACM SIGCOMM 2023 Conference*, ACM SIGCOMM '23, page 564–577, New York, NY, USA, 2023. Association for Computing Machinery. doi:10.1145/3603269.3604848.

[60] Shutian Luo, Huanle Xu, Kejiang Ye, Guoyao Xu, Liping Zhang, Jian He, Guodong Yang, and Chengzhong Xu. Erms: Efficient Resource Management for Shared Microservices with SLA Guarantees. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1*, ASPLOS 2023, page 62–77, New York, NY, USA, 2022. Association for Computing Machinery. doi:10.1145/3567955.3567964.

[61] Chenghao Lyu, Qi Fan, Philippe Guyard, and Yanlei Diao. A Spark Optimizer for Adaptive, Fine-Grained Parameter Tuning. *Proc. VLDB Endow.*, 17(11):3565–3579, July 2024. doi:10.14778/3681954.3682021.

[62] Filipe Manco, Costin Lupu, Florian Schmidt, Jose Mendes, Simon Kuenzer, Sumit Sati, Kenichi Yasukata, Costin Raiciu, and Felipe Huici. My VM is Lighter (and Safer) than Your Container. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, page 218–233, New York, NY, USA, 2017. Association for Computing Machinery. doi:10.1145/3132747.3132763.

[63] Hongzi Mao, Mohammad Alizadeh, Ishai Menache, and Srikanth Kandula. Resource Management with Deep Reinforcement Learning. In *Proceedings of the 15th ACM Workshop on Hot Topics in Networks*, HotNets '16, page 50–56, New York, NY, USA, 2016. Association for Computing Machinery. doi:10.1145/3005745.3005750.

[64] Jason Mars, Lingjia Tang, Robert Hundt, Kevin Skadron, and Mary Lou Soffa. Bubble-Up: increasing utilization in modern warehouse scale computers via sensible co-locations. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-44, page 248–259, New York, NY, USA, 2011. Association for Computing Machinery. doi:10.1145/2155620.2155650.

[65] Abhishek Modi, Kaushik Rajan, Srinivas Thimmaiah, Prakhar Jain, Swinky Mann, Ayushi Agarwal, Ajith Shetty, Shahid K I, Ashit Gosalia, and Partho Sarthi. New query optimization techniques in the Spark engine of Azure synapse. *Proc. VLDB Endow.*, 15(4):936–948, December 2021. doi:10.14778/3503585.3503601.

[66] Ingo Müller, Renato Marroquín, and Gustavo Alonso. Lambada: Interactive Data Analytics on Cold Data Using Serverless Cloud Infrastructure. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, SIGMOD '20, page 115–130, New York, NY, USA, 2020. Association for Computing Machinery. doi:10.1145/3318464.3389758.

[67] Andrew Newell, Dimitrios Skarlatos, Jingyuan Fan, Pavan Kumar, Maxim Khutornenko, Mayank Pundir, Yirui Zhang, Mingjun Zhang, Yuanlai Liu, Linh Le, Brendon Daugherty, Apurva Samudra, Prashasti Baid, James Kneeland, Igor Kabiljo, Dmitry Shchukin, Andre Rodrigues, Scott Michelson, Ben Christensen, Kaushik Veeraraghavan, and Chunqiang Tang. RAS: Continuously Optimized Region-Wide Datacenter Resource Allocation. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, SOSP '21, page 505–520, New York, NY, USA, 2021. Association for Computing Machinery. doi:10.1145/3477132.3483578.

[68] Edward Oakes, Leon Yang, Dennis Zhou, Kevin Houck, Tyler Harter, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. SOCK: Rapid Task Provisioning with Serverless-Optimized Containers. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 57–70, Boston, MA, July 2018. USENIX Association. URL: https://www.usenix.org/conference/atc18/presentation/oakes.

[69] Opencontainers Initiative. runc: A CLI tool for spawning and running containers. https://github.com/opencontainers/runc. Accessed: 2026-01-17.

[70] Kay Ousterhout, Ryan Rasti, Sylvia Ratnasamy, Scott Shenker, and Byung-Gon Chun. Making Sense of Performance in Data Analytics Frameworks. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*, pages 293–307, Oakland, CA, May 2015. USENIX Association. URL: https://www.usenix.org/conference/nsdi15/technical-sessions/presentation/ousterhout.

[71] Xingguo Pang, Yanze Zhang, Liu Liu, Dazhao Cheng, Chengzhong Xu, and Xiaobo Zhou. Expeditious High-Concurrency MicroVM SnapStart in Persistent Memory with an Augmented Hypervisor. In *2024 USENIX Annual Technical Conference (USENIX ATC 24)*, pages 985–998, Santa Clara, CA, July 2024. USENIX Association. URL: https://www.usenix.org/conference/atc24/presentation/pang.

[72] Yeonsu Park, Byungchul Tak, and Wook-Shin Han. QaaD (Query-as-a-Data): Scalable Execution of Massive Number of Small Queries in Spark. *Proc. ACM Manag. Data*, 1(2), June 2023. doi:10.1145/3589279.

[73] Tirthak Patel and Devesh Tiwari. CLITE: Efficient and QoS-Aware Co-Location of Multiple Latency-Critical Jobs for Warehouse Scale Computers. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 193–206, 2020. doi:10.1109/HPCA47549.2020.00025.

[74] Matthew Perron, Raul Castro Fernandez, David DeWitt, and Samuel Madden. Starling: A Scalable Query Engine on Cloud Functions. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, SIGMOD '20, page 131–141, New York, NY, USA, 2020. Association for Computing Machinery. doi:10.1145/3318464.3380609.

[75] Qifan Pu, Shivaram Venkataraman, and Ion Stoica. Shuffling, Fast and Slow: Scalable Analytics on Serverless Infrastructure. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 193–206, Boston, MA, February 2019. USENIX Association. URL: https://www.usenix.org/conference/nsdi19/presentation/pu.

[76] Charles Reiss, John Wilkes, and Joseph Hellerstein. More Google cluster data. http://googleresearch.blogspot.com/2011/11/more-google-cluster-data.html, 2011. Accessed: 2025-12-10.

[77] Krzysztof Rzadca, Pawel Findeisen, Jacek Swiderski, Przemyslaw Zych, Przemyslaw Broniek, Jarek Kusmierek, Pawel Nowak, Beata Strack, Piotr Witusowski, Steven Hand, and John Wilkes. Autopilot: workload autoscaling at Google. In *Proceedings of the Fifteenth European Conference on Computer Systems*, EuroSys '20, New York, NY, USA, 2020. Association for Computing Machinery. doi:10.1145/3342195.3387524.

[78] Malte Schwarzkopf, Andy Konwinski, Michael Abd-El-Malek, and John Wilkes. Omega: flexible, scalable schedulers for large compute clusters. In *Proceedings of the 8th ACM European Conference on Computer Systems*, EuroSys '13, page 351–364, New York, NY, USA, 2013. Association for Computing Machinery. doi:10.1145/2465351.2465386.

[79] Rathijit Sen, Abhishek Roy, Alekh Jindal, Rui Fang, Jeff Zheng, Xiaolei Liu, and Ruiping Li. AutoExecutor: predictive parallelism for spark SQL queries. *Proc. VLDB Endow.*, 14(12):2855–2858, July 2021. doi:10.14778/3476311.3476362.

[80] Mohammad Shahrad, Rodrigo Fonseca, Inigo Goiri, Gohar Chaudhry, Paul Batum, Jason Cooke, Eduardo Laureano, Colby Tresness, Mark Russinovich, and Ricardo Bianchini. Serverless in the Wild: Characterizing and Optimizing the Serverless Workload at a Large Cloud Provider. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pages 205–218. USENIX Association, July 2020. URL: https://www.usenix.org/conference/atc20/presentation/shahrad.

[81] Min Shen, Ye Zhou, and Chandni Singh. Magnet: push-based shuffle service for large-scale data processing. *Proc. VLDB Endow.*, 13(12):3382–3395, August 2020. doi:10.14778/3415478.3415558.

[82] Simon Shillaker and Peter Pietzuch. Faasm: Lightweight Isolation for Efficient Stateful Serverless Computing. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pages 419–433. USENIX Association, July 2020. URL: https://www.usenix.org/conference/atc20/presentation/shillaker.

[83] Fei Song, Khaled Zaouk, Chenghao Lyu, Arnab Sinha, Qi Fan, Yanlei Diao, and Prashant Shenoy. Spark-based Cloud Data Analytics using Multi-Objective Optimization. In *2021 IEEE 37th International Conference on Data Engineering (ICDE)*, pages 396–407, 2021. doi:10.1109/ICDE51399.2021.00041.

[84] Vikram Sreekanti, Chenggang Wu, Xiayue Charles Lin, Johann Schleier-Smith, Joseph E. Gonzalez, Joseph M. Hellerstein, and Alexey Tumanov. Cloudburst: stateful functions-as-a-service. *Proc. VLDB Endow.*, 13(12):2438–2452, July 2020. doi:10.14778/3407790.3407836.

[85] Ariel Szekely, Adam Belay, Robert Morris, and M. Frans Kaashoek. Unifying serverless and microservice workloads with SigmaOS. In *Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles*, SOSP '24, page 385–402, New York, NY, USA, 2024. Association for Computing Machinery. doi:10.1145/3694715.3695947.

[86] Chunqiang Tang, Kenny Yu, Kaushik Veeraraghavan, Jonathan Kaldor, Scott Michelson, Thawan Kooburat, Aravind Anbudurai, Matthew Clark, Kabir Gogia, Long Cheng, Ben Christensen, Alex Gartrell, Maxim Khutornenko, Sachin Kulkarni, Marcin Pawlowski, Tuomas Pelkonen, Andre Rodrigues, Rounak Tibrewal, Vaishnavi Venkatesan, and Peter Zhang. Twine: a unified cluster management system for shared infrastructure. In *Proceedings of the 14th USENIX Conference on Operating Systems Design and Implementation*, OSDI'20, USA, 2020. USENIX Association. URL: https://www.usenix.org/conference/osdi20/presentation/tang.

[87] Muhammad Tirmazi, Adam Barker, Nan Deng, Md E. Haque, Zhijing Gene Qin, Steven Hand, Mor Harchol-Balter, and John Wilkes. Borg: the next generation. In *Proceedings of the Fifteenth European Conference on Computer Systems*, EuroSys '20, New York, NY, USA, 2020. Association for Computing Machinery. doi:10.1145/3342195.3387517.

[88] Transaction Processing Performance Council (TPC). TPC-H Benchmark. https://www.tpc.org/tpch/, 2025. Accessed: 2025-12-05.

[89] Transaction Processing Performance Council. TPC Benchmark DS (TPC-DS) Standard Specification. https://www.tpc.org/tpc_documents_current_versions/current_specifications5.asp, 2023. Accessed: 2025-12-11.

[90] Dmitrii Ustiugov, Plamen Petrov, Marios Kogias, Edouard Bugnion, and Boris Grot. Benchmarking, analysis, and optimization of serverless function snapshots. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '21, page 559–572, New York, NY, USA, 2021. Association for Computing Machinery. doi:10.1145/3445814.3446714.

[91] Vinod Kumar Vavilapalli, Arun C. Murthy, Chris Douglas, Sharad Agarwal, Mahadev Konar, Robert Evans, Thomas Graves, Jason Lowe, Hitesh Shah, Siddharth Seth, Bikas Saha, Carlo Curino, Owen O'Malley, Sanjay Radia, Benjamin Reed, and Eric Baldeschwieler. Apache Hadoop YARN: yet another resource negotiator. In *Proceedings of the 4th Annual Symposium on Cloud Computing*, SOCC '13, New York, NY, USA, 2013. Association for Computing Machinery. doi:10.1145/2523616.2523633.

[92] Abhishek Verma, Luis Pedrosa, Madhukar Korupolu, David Oppenheimer, Eric Tune, and John Wilkes. Large-scale cluster management at Google with Borg. In *Proceedings of the Tenth European Conference on Computer Systems*, EuroSys '15, New York, NY, USA, 2015. Association for Computing Machinery. doi:10.1145/2741948.2741964.

[93] Smita Vijayakumar, Anil Madhavapeddy, and Evangelia Kalyvianaki. Scheduling for Reduced Tail Task Latencies in Highly Utilized Datacenters. In *Proceedings of the 2024 ACM Symposium on Cloud Computing*, SoCC '24, page 302–321, New York, NY, USA, 2024. Association for Computing Machinery. doi:10.1145/3698038.3698522.

[94] Yaohui Wang, Ben Luo, and Yibin Shen. Efficient Memory Overcommitment for I/O Passthrough Enabled VMs via Fine-grained Page Meta-data Management. In *2023 USENIX Annual Technical Conference (USENIX ATC 23)*, pages 769–783, Boston, MA, July 2023. USENIX Association. URL: https://www.usenix.org/conference/atc23/presentation/wang-yaohui.

[95] Michael Wawrzoniak, Rodrigo Bruno, Ana Klimovic, and Gustavo Alonso. Ephemeral Per-query Engines for Serverless Analytics. In *Joint Proceedings of Workshops at the 49th International Conference on Very Large Data Bases (VLDB 2023)*, volume 3462 of *CEUR Workshop Proceedings*, Vancouver, Canada, August 2023. August 28–September 1, 2023. URL: https://ceur-ws.org/Vol-3462/SDA3.pdf.

[96] Xingda Wei, Fangming Lu, Tianxia Wang, Jinyu Gu, Yuhan Yang, Rong Chen, and Haibo Chen. No Provisioned Concurrency: Fast RDMA-codesigned Remote Fork for Serverless Computing. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*, pages 497–517, Boston, MA, July 2023. USENIX Association. URL: https://www.usenix.org/conference/osdi23/presentation/wei-rdma.

[97] Yixin Wu, Xiuqi Huang, Zhongjia Wei, Hang Cheng, Chaohui Xin, Zuzhi Chen, Binbin Chen, Yufei Wu, Hao Wang, Tieying Zhang, Rui Shi, Xiaofeng Gao, Yuming Liang, Pengwei Zhao, and Guihai Chen. Towards Resource Efficiency: Practical Insights into Large-Scale Spark Workloads at ByteDance. *Proc. VLDB Endow.*, 17(12):3759–3771, August 2024. doi:10.14778/3685800.3685804.

[98] Wu Xiang, Yakun Li, Yuquan Ren, Fan Jiang, Chaohui Xin, Varun Gupta, Chao Xiang, Xinyi Song, Meng Liu, Bing Li, Kaiyang Shao, Chen Xu, Wei Shao, Yuqi Fu, Wilson Wang, Cong Xu, Wei Xu, Caixue Lin, Rui Shi, and Yuming Liang. Gödel: Unified Large-Scale Resource Management and Scheduling at ByteDance.

In *Proceedings of the 2023 ACM Symposium on Cloud Computing*, SoCC '23, page 308–323, New York, NY, USA, 2023. Association for Computing Machinery. doi:10.1145/3620678.3624663.

[99] Jinhan Xin, Kai Hwang, and Zhibin Yu. LOCAT: Low-Overhead Online Configuration Auto-Tuning of Spark SQL Applications. In *Proceedings of the 2022 International Conference on Management of Data*, SIGMOD '22, page 674–684, New York, NY, USA, 2022. Association for Computing Machinery. doi:10.1145/3514221.3526157.

[100] Tong Xing, Cong Xiong, Chuan Ye, Qi Wei, Javier Picorel, and Antonio Barbalace. Maximizing VMs' IO Performance on Overcommitted CPUs with Fairness. In *Proceedings of the 2023 ACM Symposium on Cloud Computing*, SoCC '23, page 93–108, New York, NY, USA, 2023. Association for Computing Machinery. doi:10.1145/3620678.3624649.

[101] Ran Xu, Subrata Mitra, Jason Rahman, Peter Bai, Bowen Zhou, Greg Bronevetsky, and Saurabh Bagchi. Pythia: Improving Datacenter Utilization via Precise Contention Prediction for Multiple Co-located Workloads. In *Proceedings of the 19th International Middleware Conference*, Middleware '18, page 146–160, New York, NY, USA, 2018. Association for Computing Machinery. doi:10.1145/3274808.3274820.

[102] Bowen Yu, Guanyu Feng, Huanqi Cao, Xiaohan Li, Zhenbo Sun, Haojie Wang, Xiaowei Zhu, Weimin Zheng, and Wenguang Chen. Chukonu: a fully-featured high-performance big data framework that integrates a native compute engine into Spark. *Proc. VLDB Endow.*, 15(4):872–885, December 2021. doi:10.14778/3503585.3503596.

[103] Hanfei Yu, Rohan Basu Roy, Christian Fontenot, Devesh Tiwari, Jian Li, Hong Zhang, Hao Wang, and Seung-Jong Park. RainbowCake: Mitigating Cold-starts in Serverless with Layer-wise Container Caching and Sharing. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1*, ASPLOS '24, page 335–350, New York, NY, USA, 2024. Association for Computing Machinery. doi:10.1145/3617232.3624871.

[104] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauly, Michael J. Franklin, Scott Shenker, and Ion Stoica. Resilient distributed datasets: A Fault-Tolerant abstraction for In-Memory cluster computing. In *9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*, pages 15–28, San Jose, CA, April 2012. USENIX Association. URL: https://www.usenix.org/conference/nsdi12/technical-sessions/presentation/zaharia.

[105] Hong Zhang, Yupeng Tang, Anurag Khandelwal, Jingrong Chen, and Ion Stoica. Caerus: NIMBLE Task Scheduling for Serverless Analytics. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*, pages 653–669. USENIX Association, April 2021. URL: https://www.usenix.org/conference/nsdi21/presentation/zhang-hong.

[106] Xiao Zhang, Eric Tune, Robert Hagmann, Rohit Jnagal, Vrigo Gokhale, and John Wilkes. CPI2: CPU performance isolation for shared compute clusters. In *Proceedings of the 8th ACM European Conference on Computer Systems*, EuroSys '13, page 379–391, New York, NY, USA, 2013. Association for Computing Machinery. doi:10.1145/2465351.2465388.

[107] Xinchun Zhang, Aqsa Kashaf, Yihan Zou, Wei Zhang, Weibo Liao, Haoxiang Song, Jintao Ye, Yakun Li, Rui Shi, Yong Tian, Wei Feng, Binbin Chen, Zuzhi Chen, Tieying Zhang, and Yongping Tang. ResLake: Towards Minimum Job Latency and Balanced Resource Utilization in Geo-Distributed Job Scheduling. *Proc. VLDB Endow.*, 17(12):3934–3946, August 2024. doi:10.14778/3685800.3685817.

[108] Zhuo Zhang, Chao Li, Yangyu Tao, Renyu Yang, Hong Tang, and Jie Xu. Fuxi: a fault-tolerant resource management and job scheduling system at internet scale. *Proc. VLDB Endow.*, 7(13):1393–1404, August 2014. doi:10.14778/2733004.2733012.

[109] Laiping Zhao, Yanan Yang, Yiming Li, Xian Zhou, and Keqiu Li. Understanding, predicting and scheduling serverless workloads under partial interference. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '21, New York, NY, USA, 2021. Association for Computing Machinery. doi:10.1145/3458817.3476215.

[110] Yiwen Zhu, Rathijit Sen, Brian Kroth, Sergiy Matusevych, Andreas Mueller, Tengfei Huang, Rahul Challapalli, Weihan Tang, Xin He, Mo Liu, Estera Kot, Sule Kahraman, Arshdeep Sekhon, Dario Bernal, Aditya Lakra, Shaily Fozdar, Dhruv Relwani, Rui Fang, Long Tian, Karuna Sagar Krishna, Ashit Gosalia, Carlo Curino, and Subru Krishnan. Rockhopper: A Robust Optimizer for Spark Configuration Tuning in Production Environment. In *Companion of the 2025 International Conference on Management of Data*, SIGMOD/PODS '25, page 743–756, New York, NY, USA, 2025. Association for Computing Machinery. doi:10.1145/3722212.3724451.

Continuation-Centric Computing with Arca

Akshay Srivatsan Yuhan Deng Katherine Mohr Emma Sudo
Sebastian Ingino Francis Chua Keith Winstein
Stanford University

Abstract

This paper presents *continuation-centric computing*, an operating system design paradigm for "serverless"-style workloads: short-lived tasks that spend a substantial fraction of time waiting on dependencies and other services. Under this paradigm, a running function can capture its current *continuation*—a lightweight snapshot of its state—as a distinct function that can be paused, migrated, or copied as needed.

This paper evaluates the continuation-centric computing model using Arca, an operating system providing continuation capture as a core service. Arca supports a process abstraction that is broadly similar to Unix processes in guarantees and isolation, but additionally provides efficient capture and resumption of serializable, portable continuations.

1 Introduction

Many cloud-computing operators offer function-as-a-service (FaaS) products, commonly as part of "serverless computing." These systems allow users to create functions to be invoked in response to external events, such as incoming HTTP requests or scheduled timers. The functions are run in provider-provisioned sandboxes with access to APIs for interacting with external services such as databases and object stores. In contrast to traditional cloud-computing products where a user must request and manage servers to run their functions, serverless providers manage the servers and only charge users based on the runtime of each function invocation. For appropriate workloads, this simplifies the developer experience and gives providers flexibility in scheduling and placing invocations.

To improve server utilization and reduce costs, commercial serverless systems break function invocations into slices as small as a few milliseconds [19] or 128 MiB of RAM [4]. Research has aimed to support even finer granularities in time and space [16,33,44,48], motivated partly by applications that use serverless systems for large-scale, highly parallel jobs, breaking up their workloads into fine-grained tasks with complex inter-task dependencies [20–22,26,28–30,45,55]. This has generally required software developers to break programs up ahead-of-time into subtasks, each a function of some identified input data. Doing so can expose available parallelism and let resource and data needs be matched more closely to the resources allocated and placed at runtime [16,31,43,44].

Isolation primitive	create/destroy (µs)	snapshot/resume (µs)
MicroVM [1]	✘ 742,000	✘ 217,000
Linux process	✘ 540,000	✘ 283,000
WebAssembly [2]	✓ 110	✘ n/a
Arca (our work)	✓ 32.2	✓ 2.55

Table 1: Performance of different isolation primitives (§2.2). Arca's create/destroy and snapshot/resume overheads are several orders of magnitude lower than other isolation primitives. Create/destroy time was measured with 128 parallel processes on a 128-core machine. Full results are in Section 6.

Is it useful to help applications break themselves up into smaller pieces? This paper argues that operating systems could benefit applications by adopting *continuation capture* as a core service. Continuations allow a running program to capture "the rest of this program" as a new program. Capturing a continuation on every I/O operation can transform a monolithic program into a composition of pure functions with delineated I/O. Current operating systems now include ways to capture a program's continuation by snapshotting running processes or VMs [9,13]; however, these methods take hundreds of milliseconds (Table 1). By designing a new OS with continuations as a core primitive, we found it is possible to capture a continuation of a running process in a few *microseconds*. This makes it more feasible to execute programs as a sequence of pure functions with I/O in between.

To explore the continuation-centric computing approach, we designed and implemented Arca, an OS designed for continuation-centric workloads. Arca's kernel provides continuation-capture as a syscall that processes may invoke; developers (or the compiler) do not need to explicitly express the state required by "the rest of the program from point x." Arca processes are broadly similar to Unix-style processes in

that the kernel isolates them via hardware memory protection, and they can do computations and return values. To perform an external side effect, which would traditionally involve system calls like `read` and `write`, an Arca process instead returns an "effect": a description of the desired I/O and a callback function, which may be a continuation of the same process.

Arca is a research prototype with many limitations. We have adapted musl libc to implement part of the POSIX API in terms of Arca effects, but it supports only the most common POSIX calls for file I/O (`open`, `close`, `read`, `write`, ...) and networking (`socket`, `connect`, `accept`, ...). There is no support for mutable shared memory between threads of execution (`mmap`, `shm_open`, `clone`). The most complicated program we've ported to run on Arca is FFmpeg (the video toolkit); most large applications would require considerable porting effort and a fuller libc. And Arca probably isn't very beneficial for long-lived programs with consistent resource demands.

Some of these are engineering limitations that could be overcome with further effort, but others are more fundamental to continuation-centric computing. Many APIs increase the footprint of a process in ways that frustrate clean continuation capture. Berkeley sockets create dependencies on IP addresses and TCP connection state. Shared mutable memory introduces dependencies on the internal state of other processes. A continuation-centric computing system that supports interfaces like these loses some of the benefits, but one that doesn't breaks compatibility.

What is Arca for? We designed Arca to fill a need we found we had ourselves. Many researchers, including us, have explored the use of serverless platforms for large jobs with complex inter-task dependencies: video processing [22], linear algebra [28,45], software compilation and testing [20], theorem proving [55], 3D rendering [21], ML training [26], data analysis [29,31], etc. This literature led to a principle of "I/O-compute separation" [15,30] and several general frameworks for describing and orchestrating the I/O relationships between individual functions to be executed [16, 20, 22, 28, 31, 34]. (Three of these [16,20,22] are from our research group.)

In all of this, what we've found missing is a practical way to *express* everyday programs as a set of pure functions connected by a dependency graph. Consider an ordinary program, perhaps written in Rust or C++, that creates a 1,000-element vector and wants to perform a parallel map-reduce, and, once that's complete, do more single-threaded computation with the result. At the places where the program transitions from 1-way to 1,000-way parallelism and back, Rust and C++ don't provide a tool to capture the state of the program in a way that can be resumed as a new serverless function invocation.

In past work, we and other researchers manually transformed our programs into continuation-passing style (CPS); separating them into smaller programs and tracking the intervening state explicitly. Arca makes this a lot easier, by providing the CPS transform as a core OS service.

Summary of results. For programs that would benefit from capturing their futures in a serializable continuation—generally speaking, programs that mix computation and I/O or that vary their working set over time—Arca's native continuation capture has lower overhead than some prior approaches. In a closed-loop test, Arca captured continuations five orders of magnitude faster than comparable existing systems (Fig. 3). Most of that benefit comes from Arca's ability to produce "zero-copy" continuations of an in-memory process. In tests of throughput, Arca can serve an order of magnitude more requests than existing process-based techniques, and performs comparably to in-process isolation techniques (Table 4).

In an evaluation of continuation-centric computing on a serverless image thumbnailing workload, we found that a continuation-centric approach implemented using Arca is able to achieve a 50–60× speedup over a non-continuation-centric approach. The captured continuation is much smaller than the image to be thumbnailed, and so the ability to send the continuation over the network instead of the image allows Arca to utilize the CPU more effectively (Figure 6).

This paper proceeds as follows: we first discuss the motivation and design goals of continuation-centric computing (§ 2) and the practical developer experience (§ 3). In Section 4, we provide an overview of Arca's design. We describe Arca's implementation in Section 5, and provide an evaluation of its performance and compatibility (§ 6). We then survey related work in Section 7. Finally, we discuss limitations and future work (§§ 8, 9) before concluding (§ 10).

2 Motivation

Continuation-centric computing is motivated by trends in recent work in serverless computing. Besides performance, there are two major selective pressures influencing the evolution of serverless platforms: *compatibility* with existing software and *granularity* of scheduling.

2.1 Compatibility

Serverless systems adopted in industry share a common focus on maintaining compatibility with existing software. Many current serverless providers run off-the-shelf Linux applications via containers or MicroVM; for example, this is the approach taken by AWS Lambda [1,4]. This approach maintains the widest compatibility. Some effort is still required to package an application for serverless execution—a developer must, for example, supply a wrapper that gathers necessary inputs from the provider's datastore, runs the target application, and writes outputs back to the datastore—but for the most part existing software can run unmodified.

However, full compatibility with Linux comes at a cost: shipping a full Linux userspace with every application leads to worse granularity and performance. This has led some

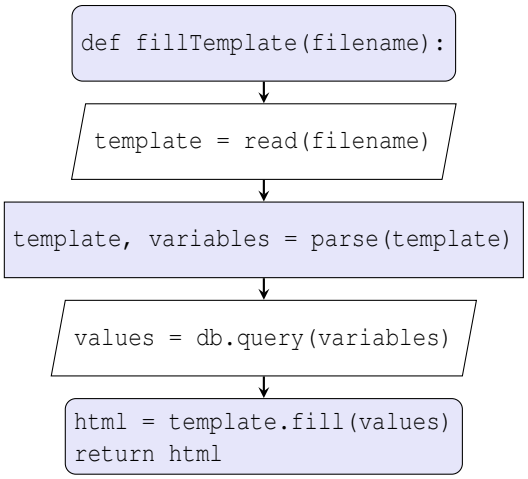


Figure 1: A logical function broken down into a sequence of funclets and I/O operations. Every shaded rectangular flow chart element is a separate funclet, while every unshaded trapezoid is an I/O operation.

providers to innovate on their isolation mechanism; Cloudflare Workers [10] relies on software isolation techniques like WebAssembly [23]. Providers taking this approach often expose traditional operating system primitives, such as file descriptors, so that applications can be ported without many source-level changes; in WebAssembly, this takes the form of the WebAssembly System Interface (WASI) [11]. These platforms are not as widely compatible with existing applications: in the case of WebAssembly, applications must be recompiled for a different architecture and must therefore be configured not to use features that WebAssembly lacks. WASI generally supports the subset of POSIX related to file I/O, but omits features like networking and/or multithreading; applications reliant on these features must use provider APIs to access equivalents instead. However, these platforms have seen adoption despite these incompatibilities, as they enable finer granularity scheduling.

2.2 Granularity

New serverless systems proposed in academia focus primarily on enabling fine-grained scheduling, with the goal of improving end-to-end performance. These works propose various approaches to decompose large monolithic applications into finer-grained functions [16, 31, 43, 44, 48]. While the details vary, all these systems have a notion of a small unit of pure computation (previously referred to as a "proclet") and a larger unit of mixed computation and I/O composed of the small unit (a "logical process"). Across most of these systems, the developer writes the small unit of computation in a traditional language, like C or C++.

The exact method of composing the small unit into the larger unit varies. Dandelion uses an external domain-specific language (DSL) [31], requiring logical processes to be written specifically for Dandelion. Fix uses a custom API exposed through WebAssembly, essentially providing an embedded domain-specific language (eDSL) in the form of a C library [16]. σOS similarly exposes a library interface to its primitives, akin to a reduced version of the Linux system call interface [48]. Likewise, Quicksand exposes its data structures and control flow using an API resembling the C++ standard template library [43].

The premise behind these works is that, when applications can express their data dependencies and resource requirements at a fine granularity, a provider can make better scheduling decisions. Every time the provider learns new information about an application's dependencies, they are able to use this information when scheduling the rest of the application. This same distinction appears in other contexts as well: *xsyncfs* uses this split to achieve the performance of asynchronous I/O while retaining the durability of synchronous I/O [37].

The situations where these systems are most useful share a common aspect: what developers logically think of as "one program" are actually sequences of smaller tasks with varied requirements. Nu [44] notes this by explicitly distinguishing larger "logical processes" from smaller "proclets."

We propose a similar distinction for continuation-centric computing, but with modifications to factor in the I/O-compute separation proposed by Fix [16] and Dandelion [31]. *Logical functions* are high-level programs which a developer would think of as one task; they are composed of I/O and computation. *Funclets* are the small computational steps that make up larger tasks; they run to completion and perform no I/O. Figure 1 shows how a logical function can be broken into funclets and I/O operations.

Existing systems essentially treat the granularity at which developers write functions as the granularity at which to schedule. In continuation-centric computing, these granularities are distinguished, so developers can write code in units natural for them, and the scheduler can schedule at finer granularities. When a logical function captures its continuation, it creates a new funclet that includes its current state. This new funclet is part of the same logical function, and control flow is not interrupted from the developer's perspective.

To a first-order approximation, a continuation is essentially a snapshot of a logical function's execution. In practice, however, capturing a snapshot of a container is incredibly expensive; in the best case it takes on the order of a millisecond, but in many cases is made slower than that by various kernel limitations [25]. Current JIT-based WebAssembly runtimes do not fully isolate the state of each sandbox from the state of the host and therefore do not generally support efficient snapshotting; while it's possible to work around this, doing so interferes with optimizations and impacts runtime performance [56]. In our experiments (Table 1, § 6.1) we observed snapshot/restore systems taking hundreds of milliseconds to capture and restore a snapshot.

2.3 Design Goals

Currently, these two pressures are at odds; the existing systems we have observed tend to specialize in one of compatibility or granularity. We propose Continuation-Centric Computing, exemplified in our serverless system Arca, as a middle ground with these concrete design goals:

Compatibility. We strive to maintain compatibility at a level comparable to a production system. Specifically, we aim to match the compatibility model provided by Cloudflare Workers [10]: programs may have to be recompiled or reconfigured, but can generally run without any internal control-flow changes. We evaluate this compatibility claim by showing a mechanical translation from WebAssembly with WASI, the same interface supported by Cloudflare Workers, to Arca.

Granularity. We strive to provide granularity at a level comparable to existing research systems such as Dandelion, Fix, and Quicksand [16, 31, 43]. As "granularity" is subjective, we evaluate it via its quantitative effect on performance, by comparing function execution latency across the isolation mechanisms used by these systems. We additionally add the restriction that we try to ensure isolation and security equivalent to production systems—once again, comparing to Cloudflare Workers/WebAssembly—as in practice isolation is a necessary property for a system to be adopted.

3 Developer Experience

As "continuations" are a concept many software developers are unfamiliar with, a reasonable concern is that continuation-centric computing will be difficult for developers to use. We introduce our continuation-centric system, Arca, by describing how it compares to more familiar systems from a developer's point of view. We also describe other factors relating to the developer experience, such as ease of debugging.

3.1 Existing WebAssembly Software

Serverless providers such as Cloudflare accept software in the form of WebAssembly binaries [10, 11, 23]. We ensure compatibility with existing WebAssembly binaries, as the restrictions imposed by WebAssembly are stricter than those imposed by Arca. Many existing software toolchains support compiling code to WebAssembly, using the WebAssembly System Interface (WASI) to provide side effects such as file I/O. Among other restrictions, software targeting WebAssembly is limited to a single thread and can only access a sandboxed virtual file system. Despite these limitations, WASI ports of many popular applications, including Python [51] and FFmpeg [49], already exist.

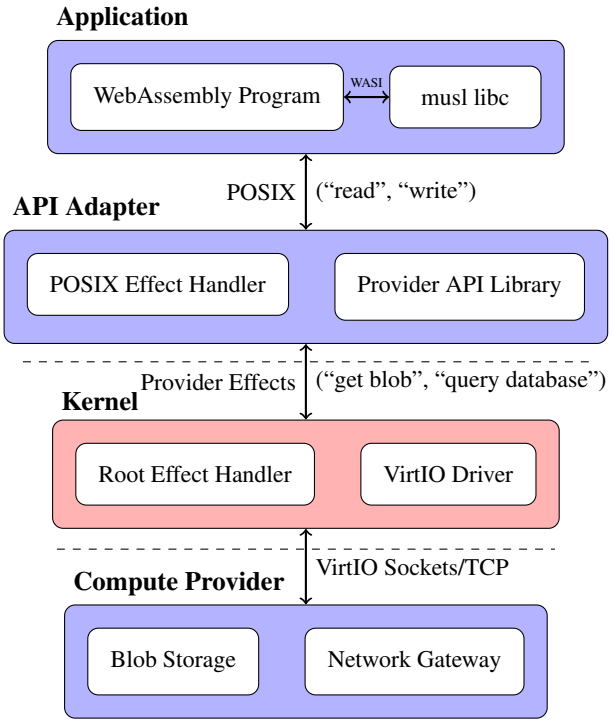


Figure 2: Arca's design allows programs written for traditional APIs, such as WASI, to be run in non-traditional environments with cloud-native APIs. A user provides their code in the form of a WebAssembly program using WASI; WASI I/O operations are translated into POSIX by musl libc, then propagated to the parent function as effects. A second userspace process translates the POSIX API into provider-specific effects. The kernel's root effect handler receives these effects and dispatches them to the outside world using VirtIO; the compute provider then performs the actual I/O. In this diagram, the dashed lines mark the boundaries between userspace and the kernel; since Arca is paravirtualized, the kernel communicates with both its own userspace and the userspace of the host machine.

The process of converting a WebAssembly binary to run on Arca is entirely mechanical; we provide a toolchain built on `wasm2c` [53] that takes an arbitrary WebAssembly/WASI binary, converts it to POSIX C, and links it against a custom implementation of WASI based on musl libc [36]. Section 6.5 describes our qualitative experience developing the WebAssembly compatibility toolchain and porting an existing WebAssembly binary to run on Arca.

The Arca implementation of WASI captures a continuation whenever a WASI I/O function is called. It then exits, returning to its parent a description of the necessary I/O along with the captured continuation—in our parlance, an "effect."

3.2 Effect Handlers

As described so far, however, there is still a missing piece: how does the I/O actually happen? If a WASI program returns an effect requesting a "read", *something* must perform the relevant I/O. In Linux this would be the kernel; in Arca we generalize this to be an "effect handler."

An effect handler is a piece of software that receives an abstract I/O request and a callback function. In the simplest case the provider supports WASI directly; then the kernel acts as the effect handler and performs the I/O itself. However, Arca also supports the case where a provider does not directly support WASI but rather supports some other provider-specific API. In this case, the raw effects produced by the application instead go to an "API Adapter," another userspace process which acts as the parent of the application. The API adapter parses the request, and reissues I/O effects in terms of the provider's native APIs. Figure 2 shows the different layers involved in this case. An API adapter only needs to be written once per cloud provider; the same adapter can run any WASI program against that provider's APIs.

We draw this distinction between the effect handler and the kernel because we have observed that developers tend to prefer familiar POSIX-like interfaces, but serverless systems provide higher-level abstractions. This is true across both the industrial and academic systems we have studied: for example, AWS Lambda uses S3 as long-term object storage rather than a traditional filesystem [4], and Dandelion exposes HTTP as a primitive communication operation [31]. By supporting user-level effect handlers, it's possible to implement a translation between WASI and a provider-native API.

3.3 Existing POSIX Software

Arca offers limited source-level compatibility with existing POSIX software. POSIX defines interfaces at the level of a C standard library (libc), to facilitate running the same application source code on different operating systems. We provide a basic implementation of a POSIX-compatible libc, based on musl [36]; this is the same libc we use for implementing WASI.

From a developer's point of view, they can write a program that looks like ordinary C or POSIX code (as in Listing 1). All the code dealing with continuations is contained within the libc, so the developer does not have to interact with or reason about continuations. Furthermore, the developer does not have to manually split a logical function into its constituent functions. Since a continuation snapshots the "rest of the function," capturing a continuation on every I/O boundary naturally breaks a logical function into funclets. When the continuation is resumed, control returns to the same point in the logical program; this allows the surrounding C code to remain unaware of the new control flow happening within libc.

```c
#include <stdio.h>

int main(int argc, char **argv) {
  FILE *in = fopen("input.txt", "r");
  fseek(in, 0L, SEEK_END);
  long sz = ftell(in);
  fclose(in);

  FILE *out = fopen("output.txt", "w");
  fprintf(out, "%ld\n", sz);
  fclose(out);
}
```

Listing 1: A trivial I/O-heavy function which counts the number of bytes in the file "input.txt", writing the result to "output.txt". Although this is a single logical function (and a single C program), the compute provider executes every line of `main` as a separate funclet (eight in total).

Arca currently supports file I/O and network I/O effects in a style similar to POSIX. Table 2 summarizes some common POSIX library calls and the status of their implementation in Arca. While Arca does support low-level network I/O, applications which use it lose the ability to capture complete continuations and must be resumed on the same machine.

Looking more generally at continuation-centric computing, there are some fundamental limitations on what POSIX APIs can be implemented. File I/O can be emulated, so it can be handled even if the provider's runtime doesn't support it natively. Pipe-based inter-process communication, `fork`, and other similar system calls can also be implemented or emulated with continuations. Network I/O comes with a tradeoff: since socket state cannot be cleanly captured in a continuation, a provider that supports it loses some of the benefits of continuation-centric computing. Many granularity-focused serverless systems explicitly do not support arbitrary network I/O for this reason [16, 31, 48]. Shared mutable memory between processes is not supported in continuation-centric computing at all, similar to WebAssembly, since it would become impossible to capture a continuation of any of the processes mapping a shared memory region independently of the others.

3.4 New Arca Software

While porting existing software to Arca provides continuation-centric computing "for free," it will nevertheless sometimes be necessary to write programs explicitly for Arca. One such case is when writing the aforementioned "API adapters"; these adapters must manipulate continuations directly using the Arca system call API (Table 3). Software which is cloud-native (i.e., new software written expressly to run on cloud platforms) may also use cloud-native APIs rather than POSIX abstractions. Developers who are already writing their applications against cloud-native APIs may be able to take advantage

POSIX Function	Status	Notes
`open`	✓	Translated to provider-native storage effects.
`close`	✓	
`read`	✓	
`write`	✓	
`lseek`	✓	
`socket`	✍	Provider must provide low-level access to networking, which makes continuation capture difficult.
`connect`	✍	
`send`	✍	
`recv`	✍	
`listen`	✍	
`accept`	✍	
`shm_open`	✘	No shared mutable memory support.
`mmap` (`MAP_SHARED`)	✘	

Table 2: Overview of common POSIX library calls and the status of their support on Arca. A status of "✓" means that a library call is always supported; "✍" means that a library call requires explicit provider support due to its impact on continuation capture; and "✘" means that a library call is not supported due to the restrictions of Arca's model. Our prototype implementation supports all the operations described as "✓" or "✍".

of this lower-level access to Arca primitives to achieve higher performance than the compatibility layers can provide.

Currently, such Arca-native code can be written as "free-standing" code in languages suitable for embedded or bare-metal development like C or Rust. So far, we have qualitatively found it easier to work with the raw Arca API in languages that support functional programming paradigms, like Rust; however, we are still exploring ways to effectively wrap the API and ease development of these "bare-metal" Arca programs. So far, we have not prioritized developing ergonomic Arca API wrappers, as we believe that in the short term most developers would prefer to use the compatibility layers to port their existing software, and in the long term developers should use cloud-native APIs like those provided by Fix, Dandelion, σOS, Nu, and Quicksand [16, 31, 43, 44, 48].

3.5 Debugging and Profiling

Since development for Arca is currently similar to embedded development, the debugging experience is also similar. Debugging logical functions is straightforward: a parent function enables single-stepping by changing the CPU flags associated with a child function, and can then inspect its state after every instruction as with `ptrace` under Linux. It can also insert breakpoint instructions by modifying the pages associated with the child function. Basic profiling is also straightforward: a parent function profiles a child function by executing it with a timeout and observing the state after the timeout. More advanced profiling, using performance counters, is possible but not currently implemented.

Debugging compositions of logical functions is more difficult, as the Arca kernel does not maintain any long-lived notion of a "call stack." However, since funclets are pure functions, it is generally possible to abstract logic into independent units to ease debugging.

4 Design

In this section, we describe the design of *Arca*, a continuation-centric operating system. We contrast the design of Arca with the design of traditional Unix-like operating systems such as Linux.

Unix is, at a high level, based on a model of processes communicating. Unix processes communicate using pipes or a shared filesystem; they perform computation in response to communication and communicate the results of their computation. Unix would therefore be classified as a message-oriented operating system [32]. As processes execute, they accumulate substantial implicit state, such as open file descriptors and socket buffers, which are difficult to capture, serialize, or restore elsewhere [25].

In order to enable continuation-centric computing, Arca is based on a model of functions consuming and producing data. Code executes in the form of short-lived funclets whose state consists primarily of their memory and registers, not ambient OS resources. Arca can therefore be classified as a procedure-oriented operating system [32]. Functional code can return requests to do I/O or gain access to more resources, which are interpreted and executed by external effect handlers; the handlers, as in Dandelion [31], then invoke separate callback funclets with the results of the I/O. Each Arca process has a limited amount of local state, forming an atomic unit which can be snapshotted, serialized, or migrated.

The duality and equivalence of message-oriented and procedure-oriented operating systems has been observed and studied. It has been demonstrated that any program written for one can be rewritten for the other as long as the appropriate primitives are present [32]. It is also well-known that the CPS transform is capable of transforming an imperative program into a functional one [7, 42]. We argue that, since serverless computing is a procedure-oriented environment but developers prefer to write message-oriented programs, integrating continuation capture into the operating system is the natural way to unite the two models efficiently.

4.1 Continuations

We have informally described a continuation as "the rest of a logical function." More concretely, a continuation in Arca is the snapshot of the execution state of an address space. This definition is consistent with prior work in distributed systems such as CIEL [34], except applied at the process level rather than the language level. Since Arca is procedure-oriented, this snapshot remains small and tightly-bounded even though the OS does not have insight into the internals of a process.

```
struct Process {
    page_table: PageTable,
    register_file: RegisterFile,
    descriptors: Vec<Value>,
}

enum Value {
    Null,
    Word(u64),
    Blob(Box<[u8]>),
    Tuple(Vec<Value>),
    Page(Page),
    Table(PageTable),
    Funclet(Funclet),
}

struct Funclet {
    defn: Definition,
    args: Vec<Value>,
}

enum Definition {
    Symbolic(Box<Value>),
    Procedural(Process),
}
```

Listing 2: The struct `Process` defines an Arca process, corresponding to Linux's `task_struct`. The other types define the primitive datatypes which Arca processes can manipulate.

The actual in-memory representation of a continuation in Arca is described by the structures in Listing 2. The structure `Process` is similar to Linux's `task_struct`, representing the state of a newly-created or suspended process. The enumeration `Value` gives the core data types which Arca programs have access to; specific abstractions like files and directories are instead replaced with these more generic abstractions. While the most common funclets are defined by a machine-code function definition, it is also possible to define a function that is merely "symbolic"; when it is invoked execution traps to the nearest effect handler instead.

4.2 System Calls

Arca processes provide a procedure-oriented set of system calls instead of the file- and message-oriented system calls available under Unix. In addition to an address space and a register file, every Arca process has a *value descriptor table* which replaces the Unix "file descriptor table." Running funclets use indices into this table to reference these values when making system calls. Instead of referring to files on disk, the entries in the value descriptor table are handles to data in kernel memory. These values include byte arrays (*blobs*), collections (*tuples*), integers, machine pages, page tables, and other Arca functions (*funclets*). Most of the system calls manipulate these values directly—they can be created, read, written, and destroyed. Several system calls instead manipulate the currently running funclet itself; they allow funclets to receive arguments, produce return values, and manipulate their address spaces. A final category of system calls allows a funclet to explicitly capture its continuation. A summary of the available raw system calls is provided in Table 3.

System Call	Description
`create_blob`	Construct a byte array from memory.
`create_tuple`	Construct a collection from other values.
`create_page`	Allocate a page of virtual memory.
`create_table`	Allocate a page table.
`create_fn`	Define a new funclet.
`read`	Read a value into program memory.
`write`	Update a value in kernel memory.
`clone`	Make a copy-on-write clone of a value.
`drop`	Delete a value.
`mmap`	Modify the current address space.
`get_arg`	Get the next argument to this funclet.
`exit`	Return a value from this funclet.
`call`	Call a funclet and get its return value.
`get_cc`	Capture the continuation of this logical function. This system call returns twice, once in the same logical function and once in the funclet described by its continuation, similar to `fork`.
`call_cc`	From the kernel's perspective, call a funclet f with the continuation k of the current logical function as an argument, and return that funclet's return value $f(k)$ as this funclet's return value. From the program's perspective, pause execution of this logical function until the specified funclet f completes.

Table 3: High-level summary of the system calls available within Arca. The first section describes system calls that manipulate values. The second section describes those that manipulate the current funclet. The third describes system calls that capture the logical function's continuation.

The biggest addition over previous procedure-oriented systems is Arca's `call_cc` system call. Since this is an OS primitive, the system call can be embedded within a library. Developers do not have to explicitly reason about, or even be aware of, continuations or effects.

4.3 Security Model

Since serverless providers run untrusted user-provided programs, it is important that programs are sandboxed.

Arca provides a security model similar to Cloudflare Workers, but without the requirement of WebAssembly [10]. Programs within an Arca process are denied access to high-precision timers such as the x86_64 timestamp counter. Since Arca processes have no shared-memory threads with which to emulate a timer and cannot perform I/O without going through

the runtime, programs cannot perform timing side-channel attacks.

The Arca kernel does not share state between different processes, reducing the surface area for attacks and data leaks. It enables standard kernel security CPU features, such as Supervisor Mode Access Prevention and Supervisor Mode Execution Prevention, to further reduce the attack surface. Further, the kernel code uses the Rust type system and memory safety model as an extra check to protect against certain classes of bugs that could leak information or privileges to userspace.

Arca does not give programs the ability to perform side effects directly. This does put some burden on the provider to choose an acceptable level of security—for example, providing arbitrary network access could allow a malicious function to access a high-precision timer over the internet. It is up to the compute provider to choose a list of explicit side effects which they deem acceptable.

4.4 Non-Goals

As a substrate for computation, Arca comes with notable limitations. Most significantly, Arca's model makes it impossible to represent parallel shared-memory threads. This restriction is imposed for both security and performance benefits on general programs, but may limit performance when porting programs that heavily rely on threads for parallelism. Programs which rely on threads only for concurrency, or which rely on processes for parallelism, can still operate as intended. We have experimented with adding support for monitors [24] as a limited form of shared state synchronization; our early results have been promising, but we have not fully fleshed out the design as it pertains to migrations and distributed state.

Arca also probably isn't immediately useful for long-lived programs with consistent resource demands and persistent state (database servers, daemons, etc.). These programs are more suited for microservice-style runtimes than serverless ones, and benefit from a different set of optimizations than those Arca provides.

Despite these limitations of Arca, we envision Arca as a useful substrate for functions that include large numbers of fine-grained funclets with complex inter-dependencies, mixing I/O and computation.

5 Implementation

In this section we describe our implementation of Arca. Our implementation is written in Rust targeting x86_64; however, we do not rely on architecture-specific features so porting to other architectures is dependent only on engineering effort.

5.1 Environment

The Arca kernel currently runs paravirtualized within Linux's Kernel-based Virtual Machine (KVM). A Linux userspace program acts as a virtual machine monitor, servicing the Arca kernel's requests for file and network I/O, and debug logging. The Arca kernel communicates with user-mode Linux programs, including but not limited to the monitor, using a VirtIO [50] Vsock device backed by the Linux kernel. Our kernel uses hypercalls to access the host's TCP stack via the monitor, and uses the Plan 9 Filesystem Protocol (9P) [39] over a Vsock for filesystem access.

5.2 System Calls

Arca's system call interface was previously described in Section 4.2; here, we focus on the kernel implementation thereof.

Values in Arca are conceptually immutable: it is not possible to *modify* a tuple, only to create a new tuple with changes. However, a literal implementation of immutability would require a prohibitive number of copies. We borrow the idea of "functional but in-place" programming from modern functional programming languages like Koka to resolve this issue [41]. System calls are linear by default, and consume their argument unless explicitly copied. System calls *appear* to mutate values; if a value is unique it is executed efficiently as a true mutation, and if a value is not unique it is executed as a less efficient copy-with-modified-contents.

A special case of "functional but in-place" which has proven to be performance-critical is the `call_cc` system call. In almost every circumstance where a continuation is captured, the program intends to use it as a callback. By implementing this operation as a single system call, the kernel avoids unnecessarily making a copy of the continuation. This means the cost of capturing a continuation is constant in this case, regardless of the memory footprint of the function.

5.3 Inter-Process Communication

As a procedure-oriented operating system, Arca does not have traditional inter-process communication constructs like pipes or shared memory. Instead, processes communicate by calling child processes with arguments and returning values to their parents. Arca processes can be thought of as coroutines, which can return values and still be resumed later.

The most frequent case of inter-process communication is in the case of the "API adapters." Instead of running "foreign" software directly, the Arca kernel runs the "native" API adapter within an Arca process and gives it the target function as an argument. The adapter does initial setup, then calls the target function in a second Arca process. The target function is then conceptually nested under the API adapter—when the target function produces an effect, the kernel sends that effect to the API adapter instead of handling it directly. The API adapter either emulates the I/O or translates it into provider-native effects that the kernel knows how to handle; either way, it eventually calls the callback with the result of the I/O.

At an implementation level, this is not inherently different from processes reading from and writing to pipes. In either case, the child process sends a request to its parent, then waits for a response. The API adapter can be thought of as a proxy that translates requests from one protocol to another. Conceptually Arca's approach closely resembles a call gate—the `call` system call allows one Arca process to invoke a function in another address space. Since sending messages on pipes and invoking functions are dual to one another, the underlying implementation remains similar despite the conceptual difference.

6 Evaluation

We test the feasibility of continuation-centric computing by evaluating the need for a new snapshotting technique, the practicality of Arca as an isolation mechanism, and the success of Arca in achieving the expected benefits of continuation-centric computing. We also discuss the process of converting existing code to run under continuation-centric computing.

6.1 Continuation Microbenchmark

The utility of continuation-centric computing is predicated on the assumption that it is possible to efficiently capture the continuation of a machine-code program. Furthermore, the need for continuations to be a core primitive depends on the relative performance of a continuation-centric approach when compared to existing snapshot/restore approaches. To evaluate the feasibility and necessity of continuation-centric computing, we compare Arca to two competitor systems.

This experiment was run on a machine with 2 AMD EPYC 7702 64-Core Processors (for a total of 128 physical cores and 256 hyperthreads) and 160 GiB of RAM, running Ubuntu 24.04.2 LTS (Linux 6.8.0-55-generic). Arca was run within a KVM virtual machine which was allocated 32 GiB of RAM; the other systems were run directly on Linux.

`Arca (snapshot+resume)` evaluates Arca in the case where a continuation is captured but not copied, e.g., when it is being used like a system call. Arca performs a functional-but-in-place optimization to make this case efficient, by keeping the continuation resident in memory. For this experiment, we assume that the effect handler is another Arca process, requiring a full context switch.

`Arca (snapshot+copy+resume)` evaluates Arca in the less-common case where a continuation is captured and copied. This would happen if a serverless runtime decides to swap a continuation to disk, send it over the network to resume on another machine, or is explicitly requested to make a copy of a continuation.

`Linux CRIU` evaluates the Checkpoint/Restore In Userspace (CRIU) system on Linux. This system is able to produce an on-disk snapshot of a process's state for later resumption. For our tests, we configure CRIU to write this snapshot to a tmpfs to avoid writing to disk.

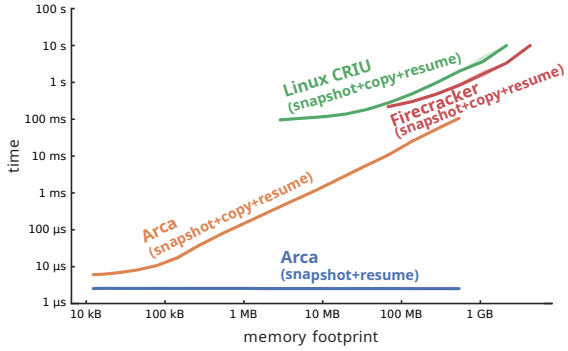


Figure 3: Time to snapshot and resume a process in Arca, Linux, and Firecracker, as process memory size varies. Measurements were averaged across 10 ten-second trials. Arca's ability to elide copies when a snapshot is resumed without migration gives it a significant advantage. Even when making a copy, Arca's elimination of global state lets it take snapshots faster than comparable techniques.

`Firecracker` evaluates the snapshot/resume system of Amazon's Firecracker MicroVM system. This system produces an on-disk snapshot of a MicroVM's state, which can be resumed later on any machine with matching hardware. We configure Firecracker to write the snapshot to a tmpfs.

All the systems run a single-threaded loop, where a parent program repeatedly snapshots and resumes a child program. We ran this loop for one second as a warm-up and then measured the number of iterations completed in the following ten seconds. We varied the memory footprint of the target program across the possible range of sizes. Arca can represent address spaces as small as 12 KiB; a Linux process by default occupies 2.75 MiB of RAM; and Firecracker needs 64 MiB of RAM and 512 KiB of disk to boot Linux. We ran 10 trials of each configuration and report averaged results in Figure 3.

In the case where a continuation is resumed without requiring a copy, `Arca (snapshot+resume)`'s performance is constant at 2.55 µs. In the case where a copy of a continuation is required, `Arca (snapshot+copy+resume)` exhibits performance that linearly scales with the size of the continuation. `Linux CRIU` and `Firecracker` similarly exhibit linear scaling performance; however, both systems entail an order-of-magnitude slowdown relative to `Arca (snapshot+copy+resume)` and have a larger baseline footprint. From these results, we conclude that Arca is feasible for continuation-centric computing as long as continuations are generally captured and resumed without being copied.

6.2 Sandbox Performance

For continuation-centric computing to be practical, it must be possible to use continuation-based sandboxes in places where traditional sandboxes are currently used. We evaluate this by comparing Arca's performance with other isolation systems.

- DKVM is a virtualization-based solution similar to Dandelion's KVM backend [31]. Dandelion supports multiple isolation backends; KVM performs the best of the backends compatible with x86_64. Dandelion's approach to KVM uses an optimization similar to Groundhog [3]: rather than creating and destroying virtual machines for every function invocation, each VM's state is reset in between function invocations.
- FCKVM is a virtualization-based solution similar to Firecracker [1]. FCKVM measures the overhead of creating a new virtual machine for every function invocation, similar to a cold start of Firecracker. To avoid the confounding factor of Linux boot times, we do not run Linux within the virtual machine, and instead run the workload bare-metal as a unikernel.
- PROC is an implementation of sandboxing using Linux processes. For every function invocation, a new process is forked to run the function (which is loaded from a shared object file). The function communicates its results with its parent by writing to a shared memory region. A true sandboxing system would additionally have to move the process to a new namespace.
- SHAREDOBJECT provides a baseline; it provides *no isolation*, and just calls a function from a shared object file. This represents the "ideal" performance if isolation had no overhead.
- WASMTIME uses the WebAssembly engine Wasmtime [2] to isolate WebAssembly functions. We configure Wasmtime to use its pooled allocation strategy to improve performance and parallelism. Functions transfer their inputs and outputs via WebAssembly memories.
- ARCA is our system, as described in the rest of this paper.

6.2.1 Sandbox Creation/Deletion

We first measured the raw sandbox creation/deletion times of the approaches, with a no-op workload. We ran this workload in a closed loop for 60 s, after providing 100 ms of warm-up time, and measured the overall request rate as we varied parallelism. We then calculate the effective time taken per request.

This experiment was run on a machine with 2 AMD EPYC 7702 64-Core Processors (for a total of 128 physical cores and 256 hyperthreads) and 512 GiB of RAM, running Ubuntu 25.04 (Linux 6.14.0-29-generic). Arca was run within a KVM virtual machine which was allocated 4 GiB of RAM; the other systems were run directly on Linux.

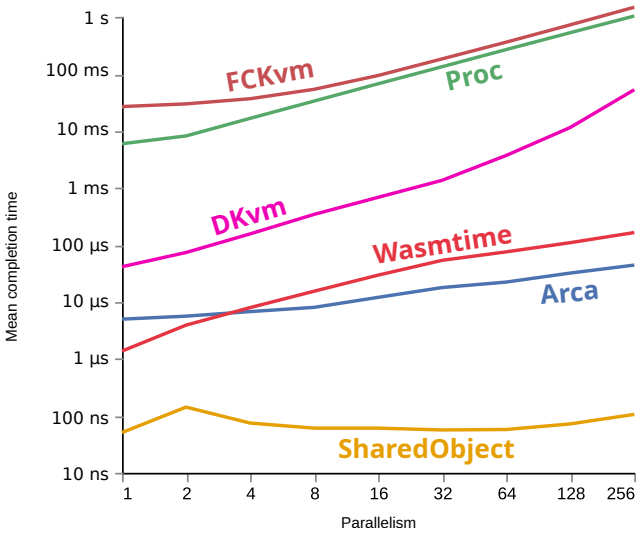


Figure 4: Performance of sandbox creation/deletion on a no-op workload. ARCA and WASMTIME are about an order of magnitude away from the theoretical limit (SHAREDOBJECT). In parallel settings, ARCA outperforms other isolation techniques.

The results of this experiment, averaged over 10 runs, are shown in Figure 4. SHAREDOBJECT provides a theoretical limit of about 100 ns per iteration, regardless of parallelism. In the single-threaded setting, WASMTIME follows at 1.5 µs per iteration, followed by ARCA at 5 µs per iteration. As parallelism increases, ARCA scales better than WASMTIME. At as low as 4-way parallelism, ARCA outperforms WASMTIME, and at 256-way parallelism, ARCA takes 30 µs per iteration (a slowdown of about 6x), while WASMTIME takes 200 µs per iteration (a slowdown of 133x). From our attempts to understand this, we believe that the slowdowns are due to contention on shared state internal to WASMTIME that effectively serializes its execution. DKVM trails about an order of magnitude behind ARCA when single-threaded, and actually scales *inversely* with increased parallelism—we believe this is because of allocator and page table contention within the kernel when changing the memory mappings of the VM. PROC operates on the order of 6 ms and FCKVM on the scale of 30 ms when single-threaded; both of these systems also scale poorly with added parallelism. We note that this microbenchmark is not a realistic measurement of the true scaling behavior under load, as longer-running functions reduce the amount of contention.

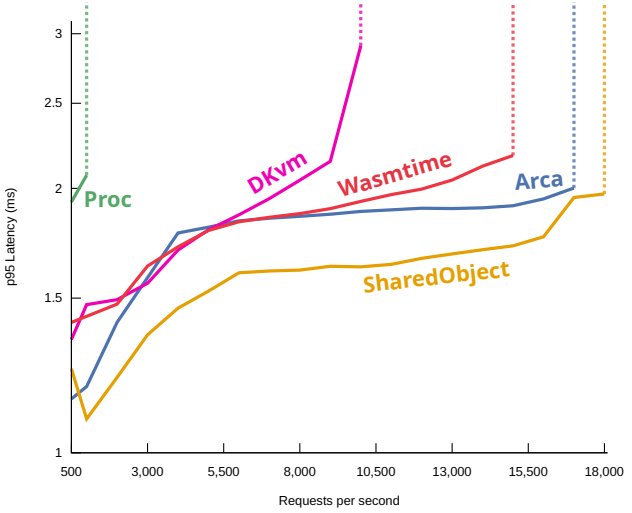


Figure 5: Request latency (95th percentile) of an open-loop 128×128 matrix-multiplication workload at various request rates. Vertical dotted lines indicate the highest request rate at which the system was able to reach a steady state. ARCA can sustain a higher request rate than other isolation methods, performing almost as well as no isolation (SHAREDOBJECT). FCKVM is omitted from this graph; it only reached 200 requests per second at a p95 latency of 42 ms.

6.2.2 Computational Workload

As a more realistic measure of the isolation mechanisms' performance under load, we perform an open-loop experiment in which we run a 128×128 32-bit integer matrix multiplication workload. For this experiment, we ran each system for 120 s, with requests isochronously generated at a range of request rates, and we measured the latency of each request. For every run of each system, we measure the p95 latency in the 10 s windows starting at the 30 s and the 100 s marks. We averaged this change in latency over all five runs of each system to determine how much each system slowed down over the course of 60 s. We declared the system unstable for a particular request rate if the slowdown exceeded 10% of the median latency; this indicates that the system would, if run long enough, exhaust its resources and crash. We only report latencies at request rates where each system remained stable.

This experiment was run on a machine with an AMD Ryzen 9 7950X 16-Core Processor (32 hyperthreads) and 64 GiB of RAM, running Fedora 42 (Linux 6.17.5-200.fc42.x86_64). Arca was run within a KVM virtual machine which was allocated 4 GiB of RAM; the other systems were run directly on Linux. All the systems were run with 32-way parallelism.

The results of this experiment are shown in Figure 5; solid lines indicate p95 latency, averaged across the five runs, and dotted lines indicate the maximum request rate for which the system remained stable. SHAREDOBJECT sets the theoretical maximum request rate for this workload at 18,000 requests per second. ARCA gets close to the theoretical limit, achieving 17,000 requests per second. WASMTIME achieves 15,000 requests per second; at most request rates, ARCA and WASMTIME have comparable response latencies. DKVM achieves 10,000 requests per second on this 32-hyperthread machine.[1] PROC trails behind, achieving 1,000 requests per second. While initially unintuitive that DKVM performs better than PROC, this is due to the implementation detail that DKVM does not actually create a new virtual machine for each request; it only has to reset the state of an already-created virtual machine. FCKVM, which does create a new virtual machine for each request, performs far worse than any of the other systems, sustaining only 200 requests per second with a p95 latency of 42 ms; we omit it from the graph.

Together, these benchmarks confirm that Arca is a viable isolation primitive for serverless computing. It slightly outperforms lightweight isolation through Wasmtime, and performs orders of magnitude better than heavyweight isolation like containers and MicroVMs: 2–3 orders of magnitude in computation-heavy tasks, and 4–5 orders of magnitude in sandbox creation/deletion. It is, however, more limited in the software it supports than the heavyweight isolation primitives.

6.3 Locality

To measure the benefits of continuation-centric computing on a serverless application, we ran an experiment on an image thumbnailing benchmark derived from SeBS [12]. For this experiment, we invoke the image thumbnailing function on one node, where the image data are located on a different node. The function takes a file path, `get`s the data, and then resizes the image. For this experiment, the image thumbnailing function invokes the "provider-native" API directly; we do not use POSIX compatibility helpers.

- CCC represents a continuation-centric computing approach to serverless, and implements `get` as a side effect. The effect handler captures, compresses, and then sends the continuation to the machine where the image data reside. The target machine decompresses and resumes the continuation.
- TRADITIONAL represents a status-quo serverless runtime where functions cannot be migrated once they have begun execution. Instead, `get` requests the image from the remote machine.

Both approaches use Arca as their isolation mechanism, but only CCC uses the continuation-capturing features of Arca.

This experiment was run on AWS `c6a.metal` instances running Ubuntu 24.04.3 LTS (Linux 6.14.0-1017-aws). Both CCC and TRADITIONAL were run within a KVM virtual

[1] This is consistent with the original measurement of 4,700 requests per second on a 16-core machine from the Dandelion paper [31].

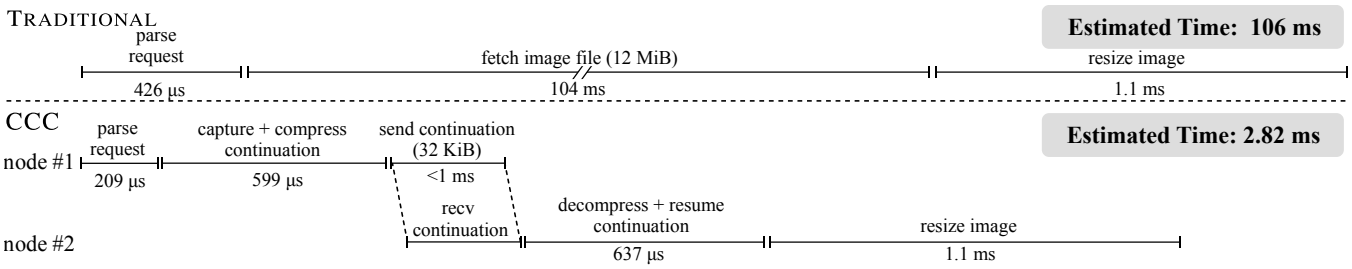


Figure 6: Time breakdown of an image thumbnailing function invoked on a different node than its input image. TRADITIONAL spends most of its runtime—98%—waiting for the image, bottlenecked by the speed of the network. CCC collocates computation and data by capturing and sending the continuation of the function. CCC spends comparatively little of its time—only 9%—doing data transfers, and is bottlenecked by computation time. TRADITIONAL achieves a throughput of 9.42 requests per second, while CCC achieves a throughput of 557.3 requests per second.

machine which was allocated 32 GiB of RAM, with networking support implemented using hypercalls to the Linux-side monitor. Both systems run a single-threaded loop on one node that launches the function locally. In CCC, the other machine receives and resumes continuations. In TRADITIONAL, the other machine serves image data to the first machine. We ran this loop for one second as a warm-up and then measured the number of functions completed in the following 60 seconds; we also measured the breakdown of where time was spent during a single function invocation. The results of the experiment are summarized in Figure 6. In CCC, 61% of a function invocation's runtime is spent resizing the image; on TRADITIONAL only 1% of the function's runtime is spent resizing, while 98% of the time is spent transferring data.

CCC achieves 557.3 requests per second, bottlenecked by decompressing and resuming the continuation and resizing the image on the target machine. In comparison, TRADITIONAL achieves 9.42 requests per second, bottlenecked by the network's speed in fetching the image. While this is an extreme case, these results confirm that continuation-centric computing can provide the benefits of collocating computation and data, similar to Fix [16], without make developers explicitly restructure programs in I/O-compute-separated style.

We also measured the throughput in the less extreme case where half of the functions can be completed with locally available data, while only half of the functions require data transfer to/from the other node. We still observe the benefit of continuation-centric computing in this case: CCC achieves 889.62 requests per second, while TRADITIONAL achieves 18.21 requests per second.

6.4 I/O Performance

We have described an automatic CPS transform based on capturing a continuation on every I/O operation. To measure the effects of this transformation on program performance, we evaluate an I/O-heavy application—a web server—implemented under Arca.

We implement this application two different ways: The first, which we call "pooled," resembles a microservice: many instances of a web server process are spawned in parallel (one per core), each incoming request is handled by the next available process from the pool, and processes return to the pool when they are done handling a request. We compare this approach against version 2.4.66 of the Apache HTTP Server [6] serving a static file from the filesystem.

The second way, which we refer to as "individual," more closely resembles serverless computing: every incoming request is handled by an invocation of a function, which runs to completion and terminates when it is done generating the response. We compare this against version 2.4.66 of the Apache HTTP Server using the Common Gateway Interface to launch a process on every request; our request handler is written in C to avoid the runtime startup penalties caused by managed languages like Python. While the request-per-function model approaches the *ideal* of serverless computing, it is not commonly used in practice due to performance concerns with spawning a process on every incoming request.

In all four cases, we generated load using ApacheBench, the Apache HTTP server benchmarking tool [5]. The four servers all returned the static string "hello, world" along with appropriate HTTP headers. This experiment was run on a machine with an AMD Ryzen 9 7950X 16-Core Processor (32 hyperthreads) and 64 GiB of RAM, running Fedora 43 (Linux 6.19.10-200.fc43.x86_64). Arca was run within a KVM virtual machine which was allocated 16 GiB of RAM; Apache was run directly on Linux. ApacheBench was configured to generate 65,536 requests with 32-way concurrency. For each system, we ran ApacheBench once as a warmup run, then measured request latency across the next 10 runs of 65,536 requests each. ApacheBench computed the CDF of request latencies for each run, and we averaged across the 10 runs. The results of this experiment are shown in Table 4.

These results show that capturing a continuation on every

Mode	System	p50	p90	p95	p99
Pooled	Arca	1.642	1.877	1.970	2.217
	Apache	1.737	1.904	1.959	2.067
Individual	Arca	3.665	4.381	4.677	5.390
	Apache	16.53	17.30	17.59	19.36

Table 4: Performance comparison of a web server implemented in Arca to the Apache HTTP Server; values are given in milliseconds. Both servers were measured under two modes of operation: in "Pooled" mode, the server spawns a thread/process for every core, each of which handles one request at a time; in "Individual" mode, the server spawns a new process for every request.

I/O operation does not noticeably affect the performance of a pooled web server; both Arca and Apache exhibit similar latencies. We believe the higher variance of Arca is due to a combination of performance tuning in Apache and the additional KVM entries and exits incurred by Arca. In a process-per-request web server, Arca outperforms Apache with CGI on Linux by a factor of four. This is likely due to the lower startup overhead of an Arca process relative to a Linux process, as previously measured in Section 6.2.1. With non-trivial workloads, we believe that computation time would dominate the workload—even on this workload, we see that the difference is much smaller than the pure startup time difference measured in Figure 4—so this measurement serves as a lower bound on the latency of each system.

6.5 Compatibility

To verify that continuation-centric computing is capable of handling existing applications, we qualitatively evaluate Arca's compatibility with existing software. We specifically consider the case described in Section 3.1, where we accept an unmodified WebAssembly binary using WASI and convert it into an Arca binary. For the purposes of this evaluation, we started with an off-the-shelf build of FFmpeg for WASI [49], based on a benchmark from SeBS [12]. This binary was compiled by a third party from the sources of FFmpeg version 5.1, using clang 14.0.4 from the official WASI SDK [52]. We first converted this binary to C code using `wasm2c` [53], a WebAssembly to C compiler. We then wrote a WASI shim, resolving WASI imports to equivalent calls into libc. Finally, we linked the converted C code, the WASI shim, and the Arca port of musl to produce a binary. We validated that the program successfully transcodes various audio files.

Developing the WASI shim layer to the level needed to support FFmpeg took one developer about two weeks of part-time effort. Since this work can be reused verbatim in the future, we expect that converting WASI programs to Arca will be fully automated. Providers can accept WASI programs and transparently run them under continuation-centric computing.

We see FFmpeg as a reasonable measure of the complexity of running unmodified WebAssembly applications: it uses a variety of WASI library functions which our conversion toolchain has to support. Since FFmpeg is compute-heavy, however, we don't see it as a difficult case performance-wise.

7 Related Work

This paper's primary contribution is providing continuation-centric computing, a new operating system design paradigm for serverless computing. This paradigm is realized in Arca, an operating system that provides lightweight hardware-isolated processes with efficient continuation capture. This section discusses prior work which relates to Arca and continuation-centric computing.

Low-Level OS Abstractions. Several prior systems provide new kernel abstractions in order to achieve better performance of userspace programs.

Exokernel [17] and Denali [54] provide abstractions that closely resemble the underlying hardware, and use a libOS to provide high-level abstractions. Spice [25] introduces new mechanisms to improve snapshot restoration time in existing OS designs. Similar to these works, Arca introduces new OS-level abstractions that more closely model the underlying serverless abstractions, using them to improve snapshot/restore performance. Lauer and Needham describe the duality between message-oriented operating systems and procedure-oriented operating systems [32]; while these prior systems fall into the former category, Arca falls into the latter.

Low-Latency Isolation. AWS Lambda is built on the Firecracker MicroVM system [1], which provides isolation using hardware virtualization. A lighter-weight approach is to use containers rather than full virtual machines; this is the approach adopted by σOS [48]. Containers share the host's kernel, but provide a new isolated namespace for global state. Previous work has tailored namespace creation for serverless workloads to achieve faster startup [38]. FAASM [46] uses WebAssembly [23] to provide software fault isolation, which outperforms containerized solutions. Arca provides hardware isolation at a level comparable to processes or containers, but by eliminating ambient OS state achieves performance similar to software isolation techniques.

High-Density Isolation. Various systems have proposed OS-backed abstractions to isolate specific components of applications. Shreds [8] provides threads with the ability to create multiple hardware-backed memory protection domains. Orbits [27] allows applications to move auxiliary tasks into separate lightweight process-like structures. These abstractions are primarily message-oriented, and Arca processes can be seen as the procedure-oriented dual to these abstractions.

Granular Scheduling. Nu [44] and Quicksand [43] advocate for resource fungibility: the ability to reassign resources quickly, even across machines. This line of work divides *logical processes* into fine-grained *proclets*. Dandelion and Fix [16, 31] perform a similar transformation, separating programs into I/O and computation. These systems use DSLs and/or eDSLs to capture dependencies. Previous systems like ExCamera and gg [20, 22] break large tasks into highly granular components to achieve burst parallelism with serverless computing. Together, this category of work provides motivation for continuation-centric computing; Arca would be a good fit as a substrate on which to implement them.

Continuations. There is a wide body of work regarding the use of continuations in programming languages [14, 18, 47], and their potential for integration in systems like WebAssembly [40]. Within distributed systems, CIEL [34, 35] explicitly uses continuations to execute programs with data-dependent control flow. These works can be seen as earlier realizations of continuation-centric computing; our contribution is to reify continuations as an operating system concept.

8 Limitations

While continuation-centric computing can improve the performance of serverless-style run-to-completion functional workloads, it is not a good fit for all use cases. We identify several categories of application which would be difficult to run under continuation-centric computing.

Shared-Memory Thread Pools. Arca does not support shared-memory threads, as shared memory increases the footprint of a continuation. An application which internally relies on a thread pool for parallelism would be limited to single-threaded performance in a direct port to Arca. Instead, a developer would need to modify the application to use a serverless provider's primitives for parallelism. We believe this limitation is true of continuation-centric computing in general.

I/O-Heavy Workloads. Applications that heavily mix I/O and computation (e.g., using `io_uring`) would be difficult to port to this model. Additionally, kernel bypass is incompatible with continuation-centric computing unless the hardware's state can be efficiently snapshotted and restored.

Databases and Daemons. Long-lived programs with consistent performance characteristics and large amounts of persistent state are probably a poor fit for "function-as-a-service" platforms, including continuation-centric computing. While Arca can transform such programs into continuation-based funclets, migrating a daemon with a large amount of persistent state will rarely be beneficial, and is generally dominated by the time required to transfer it to disk or over the network.

9 Future Work

Arca serves as a proof-of-concept of continuation-centric computing. A full system will require a runtime that schedules and places functions, akin to Fix, Dandelion, or CIEL [16, 31, 34], along with the other trappings of a serverless (or any) computing environment: access control, UIs, debuggers, metrics and monitoring, resource-allocation policies, billing, etc.

We also believe that continuation-centric computing can be extended to support microservice-style workloads, with long-running daemon processes. A possible step in this direction would be developing a continuation-centric version of a hybrid serverless/microservice system such as σOS [48]; this would require further work to efficiently represent persistent network connections in a continuation-centric style.

As we developed Arca and tested it against alternative systems, we observed that the limiting factor on the performance of sandboxing systems is often memory allocation, especially in highly parallel settings. We believe it would be worthwhile to explore whether kernel-level knowledge of the expected workload can be used to improve allocation performance.

10 Conclusion

This paper presents continuation-centric computing, a new operating system design paradigm, and evaluates it with Arca, a research OS that provides continuation capture as a core service. By designing Arca around this primitive, we found it is possible to capture a continuation of a running process in a few microseconds. This makes it more practical to automate a continuation-passing style transform of everyday programs, running them as sequences of pure funclets with I/O in between. For the right programs, this can yield substantial benefits—but much work remains to build a usable computing system around these abstractions.

Acknowledgments

We thank our shepherd Malte Schwarzkopf and our reviewers for their thoughtful feedback. We are grateful to M. Frans Kaashoek, Robert Morris, Dawson Engler, Ana Klimovic, Timothy Roscoe, Thea Rossman, Matthew Sotoudeh, Zachary Yedidia, Michael Paper, Haibib Kerim, Ariel Szekely, Hannah Gross, Benny Rubin, Ben Holmes, and Nicolaas Kaashoek for helpful conversations. This work was supported in part by NSF grant 2045714, DARPA contract HR001120C0107, a Sloan Research Fellowship, and by Google, Huawei, VMware, Dropbox, Amazon, and Meta Platforms.

Availability

Arca is available under the GNU LGPL v2.1 or later at https://github.com/fix-project/arca.

References

[1] Alexandru Agache, Marc Brooker, Andreea Florescu, Alexandra Iordache, Anthony Liguori, Rolf Neugebauer, Phil Piwonka, and Diana-Maria Popa. Firecracker: Lightweight virtualization for serverless applications. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 419–434, Santa Clara, CA, February 2020. USENIX Association.

[2] Bytecode Alliance. Wasmtime. https://wasmtime.dev/.

[3] Mohamed Alzayat, Jonathan Mace, Peter Druschel, and Deepak Garg. Groundhog: Efficient request isolation in FaaS. In *Proceedings of the Eighteenth European Conference on Computer Systems*, EuroSys '23, page 398–415, New York, NY, USA, 2023. Association for Computing Machinery.

[4] Amazon Web Services. AWS Lambda. https://aws.amazon.com/lambda/.

[5] Apache Software Foundation. ab - Apache HTTP server benchmarking tool. https://httpd.apache.org/docs/2.4/programs/ab.html.

[6] Apache Software Foundation. The Apache HTTP server project. https://httpd.apache.org/.

[7] Andrew W. Appel. SSA is functional programming. *SIGPLAN Not.*, 33(4):17–20, April 1998.

[8] Yaohui Chen, Sebassujeen Reymondjohnson, Zhichuang Sun, and Long Lu. Shreds: Fine-grained execution units with private memory. In *2016 IEEE Symposium on Security and Privacy (SP)*, pages 56–71. IEEE, may 2016.

[9] Christopher Clark, Keir Fraser, Steven Hand, Jacob Gorm Hansen, Eric Jul, Christian Limpach, Ian Pratt, and Andrew Warfield. Live migration of virtual machines. In *2nd Symposium on Networked Systems Design & Implementation (NSDI 05)*, pages 273–286, Boston, MA, May 2005. USENIX Association.

[10] Cloudflare. How Workers works: Cloudflare Workers docs. https://developers.cloudflare.com/workers/reference/how-workers-works/, 2025.

[11] Cloudflare. WebAssembly (Wasm): Cloudflare Workers docs. https://developers.cloudflare.com/workers/runtime-apis/webassembly/, 2025.

[12] Marcin Copik, Grzegorz Kwasniewski, Maciej Besta, Michal Podstawski, and Torsten Hoefler. SeBS: A serverless benchmark suite for function-as-a-service computing. In *Proceedings of the 22nd International Middleware Conference*, Middleware '21, page 64–78, New York, NY, USA, 2021. Association for Computing Machinery.

[13] CRIU Project. CRIU. https://criu.org/Main_Page.

[14] Olivier Danvy and Andrzej Filinski. Abstracting control. In *Proceedings of the 1990 ACM Conference on LISP and Functional Programming*, LFP '90, page 151–160, New York, NY, USA, 1990. Association for Computing Machinery.

[15] Yuhan Deng, Angela Montemayor, Amit Levy, and Keith Winstein. Computation-centric networking. In *Proceedings of the 21st ACM Workshop on Hot Topics in Networks*, HotNets '22, page 167–173, New York, NY, USA, 2022. Association for Computing Machinery.

[16] Yuhan Deng, Akshay Srivatsan, Sebastian Ingino, Francis Chua, Yasmine Mitchell, Matthew Vilaysack, and Keith Winstein. Fix: externalizing network I/O in serverless computing. In *Proceedings of the 21st European Conference on Computer Systems*, EuroSys '26, page 2260–2275, New York, NY, USA, 2026. Association for Computing Machinery.

[17] D. R. Engler, M. F. Kaashoek, and J. O'Toole. Exokernel: an operating system architecture for application-level resource management. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, SOSP '95, page 251–266, New York, NY, USA, 1995. Association for Computing Machinery.

[18] Matthias Felleisen. The theory and practice of first-class prompts. In *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '88, page 180–190, New York, NY, USA, 1988. Association for Computing Machinery.

[19] Aleksandr Filichkin. AWS Lambda battle 2021: performance comparison for all languages (cold and warm start). https://filia-aleks.medium.com/aws-lambda-battle-2021-performance-comparison-for-all-languages-c1b441005fd1, September 2021.

[20] Sadjad Fouladi, Francisco Romero, Dan Iter, Qian Li, Shuvo Chatterjee, Christos Kozyrakis, Matei Zaharia, and Keith Winstein. From laptop to Lambda: Outsourcing everyday jobs to thousands of transient functional containers. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 475–488, Renton, WA, July 2019. USENIX Association.

[21] Sadjad Fouladi, Brennan Shacklett, Fait Poms, Arjun Arora, Alex Ozdemir, Deepti Raghavan, Pat Hanrahan, Kayvon Fatahalian, and Keith Winstein. R2E2: Low-latency path tracing of terabyte-scale scenes using thousands of cloud CPUs. *ACM Trans. Graph.*, 41(4):76:1–76:12, July 2022.

[22] Sadjad Fouladi, Riad S. Wahby, Brennan Shacklett, Karthikeyan Vasuki Balasubramaniam, William Zeng, Rahul Bhalerao, Anirudh Sivaraman, George Porter, and Keith Winstein. Encoding, fast and slow: Low-latency video processing using thousands of tiny threads. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pages 363–376, Boston, MA, March 2017. USENIX Association.

[23] Andreas Haas, Andreas Rossberg, Derek L. Schuff, Ben L. Titzer, Michael Holman, Dan Gohman, Luke Wagner, Alon Zakai, and JF Bastien. Bringing the Web up to speed with WebAssembly. *SIGPLAN Not.*, 52(6):185–200, June 2017.

[24] C. A. R. Hoare. Monitors: an operating system structuring concept. *Commun. ACM*, 17(10):549–557, October 1974.

[25] Ben Holmes, Baltasar Dinis, Lana Honcharuk, Joshua Fried, and Adam Belay. Rethinking process snapshots for near-warm serverless cold starts. In *20th USENIX Symposium on Operating Systems Design and Implementation (OSDI 26)*, Seattle, WA, July 2026. USENIX Association.

[26] Jiawei Jiang, Shaoduo Gan, Yue Liu, Fanlin Wang, Gustavo Alonso, Ana Klimovic, Ankit Singla, Wentao Wu, and Ce Zhang. Towards demystifying serverless machine learning training. In *Proceedings of the 2021 International Conference on Management of Data*, SIGMOD/PODS '21, page 857–871, New York, NY, USA, 2021. Association for Computing Machinery.

[27] Yuzhuo Jing and Peng Huang. Operating system support for safe and efficient auxiliary execution. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 633–648, Carlsbad, CA, July 2022. USENIX Association.

[28] Eric Jonas, Qifan Pu, Shivaram Venkataraman, Ion Stoica, and Benjamin Recht. Occupy the cloud: Distributed computing for the 99%. In *Proceedings of the 2017 Symposium on Cloud Computing*, SoCC '17, page 445–451, New York, NY, USA, 2017. Association for Computing Machinery.

[29] Youngbin Kim and Jimmy Lin. Serverless data analytics with Flint. In *2018 IEEE 11th International Conference on Cloud Computing (CLOUD)*, pages 451–455. IEEE, jul 2018.

[30] Tom Kuchler, Michael Giardino, Timothy Roscoe, and Ana Klimovic. Function as a function. In *Proceedings of the 2023 ACM Symposium on Cloud Computing*, SoCC '23, page 81–92, New York, NY, USA, 2023. Association for Computing Machinery.

[31] Tom Kuchler, Pinghe Li, Yazhuo Zhang, Lazar Cvetković, Boris Goranov, Tobias Stocker, Leon Thomm, Simone Kalbermatter, Tim Notter, Andrea Lattuada, and Ana Klimovic. Unlocking true elasticity for the cloud-native era with Dandelion. In *Proceedings of the ACM SIGOPS 31st Symposium on Operating Systems Principles*, SOSP '25, page 944–961, New York, NY, USA, 2025. Association for Computing Machinery.

[32] Hugh C. Lauer and Roger M. Needham. On the duality of operating system structures. *SIGOPS Oper. Syst. Rev.*, 13(2):3–19, April 1979.

[33] Yilong Li, Seo Jin Park, and John Ousterhout. MilliSort and MilliQuery: Large-scale data-intensive computing in milliseconds. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*, pages 593–611. USENIX Association, April 2021.

[34] Derek G. Murray, Malte Schwarzkopf, Christopher Smowton, Steven Smith, Anil Madhavapeddy, and Steven Hand. CIEL: A universal execution engine for distributed data-flow computing. In *8th USENIX Symposium on Networked Systems Design and Implementation (NSDI 11)*, pages 113–126, Boston, MA, March 2011. USENIX Association.

[35] Derek Gordon Murray. *A distributed execution engine supporting data-dependent control flow*. PhD thesis, University of Cambridge, July 2011.

[36] musl libc. musl libc. https://musl.libc.org/.

[37] Edmund B. Nightingale, Kaushik Veeraraghavan, Peter M. Chen, and Jason Flinn. Rethink the sync. In *7th USENIX Symposium on Operating Systems Design and Implementation (OSDI 06)*, page 1–14, Seattle, WA, November 2006. USENIX Association.

[38] Edward Oakes, Leon Yang, Dennis Zhou, Kevin Houck, Tyler Harter, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. SOCK: Rapid task provisioning with serverless-optimized containers. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 57–70, Boston, MA, July 2018. USENIX Association.

[39] Rob Pike, David L. Presotto, Sean M. Dorward, Bob Flandrena, Ken Thompson, Howard W. Trickey, and Phil Winterbottom. Plan 9 from Bell Labs. *Computing Systems*, 8(3):221–254, Summer 1995.

[40] Donald Pinckney, Arjun Guha, and Yuriy Brun. Wasm/k: delimited continuations for WebAssembly. In *Proceedings of the 16th ACM SIGPLAN International Symposium on Dynamic Languages*, SPLASH '20, page 16–28, New York, NY, USA, 2020. Association for Computing Machinery.

[41] Alex Reinking, Ningning Xie, Leonardo de Moura, and Daan Leijen. Perceus: garbage free reference counting with reuse. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, PLDI '21, page 96–111, New York, NY, USA, 2021. Association for Computing Machinery.

[42] John C. Reynolds. The discoveries of continuations. *LISP and Symbolic Computation*, 6(3–4):233–247, Nov 1993.

[43] Zhenyuan Ruan, Shihang Li, Kaiyan Fan, Seo Jin Park, Marcos K. Aguilera, Adam Belay, and Malte Schwarzkopf. Quicksand: Harnessing stranded datacenter resources with granular computing. In *22nd USENIX Symposium on Networked Systems Design and Implementation (NSDI 25)*, pages 147–165, Philadelphia, PA, April 2025. USENIX Association.

[44] Zhenyuan Ruan, Seo Jin Park, Marcos K. Aguilera, Adam Belay, and Malte Schwarzkopf. Nu: Achieving Microsecond-Scale resource fungibility with logical processes. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 1409–1427, Boston, MA, April 2023. USENIX Association.

[45] Vaishaal Shankar, Karl Krauth, Kailas Vodrahalli, Qifan Pu, Benjamin Recht, Ion Stoica, Jonathan Ragan-Kelley, Eric Jonas, and Shivaram Venkataraman. Serverless linear algebra. In *Proceedings of the 11th ACM Symposium on Cloud Computing*, SoCC '20, pages 281–295, New York, NY, USA, 2020. Association for Computing Machinery.

[46] Simon Shillaker and Peter Pietzuch. Faasm: Lightweight isolation for efficient stateful serverless computing. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pages 419–433. USENIX Association, July 2020.

[47] Dorai Sitaram and Matthias Felleisen. Control delimiters and their hierarchies. *LISP and Symbolic Computation*, 3(1):67–99, Jan 1990.

[48] Ariel Szekely, Adam Belay, Robert Morris, and M. Frans Kaashoek. Unifying serverless and microservice workloads with SigmaOS. In *Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles*, SOSP '24, page 385–402, New York, NY, USA, 2024. Association for Computing Machinery.

[49] Ben Taylor. I built a WASI playground and you can run FFmpeg in it, which is cool. https://runno.dev/articles/ffmpeg/, 2022.

[50] Michael S. Tsirkin and Cornelia Huck. Virtual I/O device (VIRTIO) version 1.3. https://docs.oasis-open.org/virtio/virtio/v1.3/csd01/virtio-v1.3-csd01.html, 2023.

[51] VMware Labs. Python for WASI. https://github.com/vmware-labs/webassembly-language-runtimes/releases/tag/python/3.12.0%2B20231211-040d5a6, 2023.

[52] WebAssembly. WASI SDK. https://github.com/WebAssembly/wasi-sdk.

[53] WebAssembly Binary Toolkit. wasm2c (WebAssembly binary toolkit). https://github.com/WebAssembly/wabt/tree/main/wasm2c, 2018.

[54] Andrew Whitaker, Marianne Shaw, and Steven D. Gribble. Scale and performance in the Denali isolation kernel. *SIGOPS Oper. Syst. Rev.*, 36(SI):195–209, December 2002.

[55] Haoze Wu, Alex Ozdemir, Aleksandar Zeljić, Kyle Julian, Ahmed Irfan, Divya Gopinath, Sadjad Fouladi, Guy Katz, Corina Pasareanu, and Clark Barrett. Parallelization techniques for verifying neural networks. In *Proceedings of the 20th Conference on Formal Methods in Computer-Aided Design – FMCAD 2020*, pages 128–137, Wien, 2020. TU Wien Academic Press.

[56] Alon Zakai. Pause and resume WebAssembly with Binaryen's Asyncify. https://kripken.github.io/blog/wasm/2019/07/16/asyncify.html, July 2019.

ARTIFACT EVALUATED
USENIX
AVAILABLE

Rethinking Process Snapshots for Near-Warm Serverless Cold Starts

Ben Holmes, Baltasar Dinis, Lana Honcharuk, Adam Belay, Joshua Fried†
MIT CSAIL †*University of Pennsylvania*

Abstract

Serverless computing promises effortless resource elasticity, yet this vision remains fundamentally limited by cold-start delays. Despite years of optimization, many function invocations still incur delays of tens to hundreds of milliseconds, often exceeding the function's actual execution time. Existing systems attempt to avoid redundant initialization by snapshotting functions after startup and restoring them on demand, but they still struggle to deliver consistently low-latency startup from disk.

We show that these limitations stem from a lack of OS support for snapshot restoration. Current memory primitives, tailored for incremental process startup, force a trade-off between efficient on-disk layout and cheap virtual address space reconstruction. Furthermore, the kernel lacks a bulk-restore mechanism for process metadata, forcing expensive, fine-grained reconstruction from userspace.

To address this, we introduce the Snapshot Hybrid ELF (SHELF) format and a co-designed kernel primitive, the *spliceVMA*, which decouple a snapshot's physical layout on storage from the process's virtual-memory layout. This enables the efficient overlay of sparse and reordered pages, reliable prefetching, and the elimination of unnecessary I/O, copies, and faults. In parallel, we build an efficient process-metadata restore path that reconstructs process state in bulk rather than using a large number of system calls from userspace.

We implement these mechanisms in Spice, restoring and running functions from disk within 0.6–18ms of warm-invocation latency (compared to 3.6–1197ms in existing systems). For representative Java, Python, and Node.js functions, Spice reduces end-to-end latency by 7.5× and 9.5× on average over state-of-the-art process- and VM-based systems, respectively.

1 Introduction

Serverless computing promises effortless elasticity: developers deploy lightweight functions, while the platform dynamically provisions resources in response to demand [17]. Freed from managing infrastructure, users benefit from fine-grained billing and scalability, while providers can, in theory, achieve high utilization by scheduling workloads just-in-time. In today's commercial platforms (e.g., AWS Lambda [1], Azure Functions [2]), these functions are typically packaged as containers that must be initialized before running user code. This initialization latency, known as a *cold start*, fundamentally limits the vision of effortless elasticity [7, 9, 13, 31, 34, 39, 43].

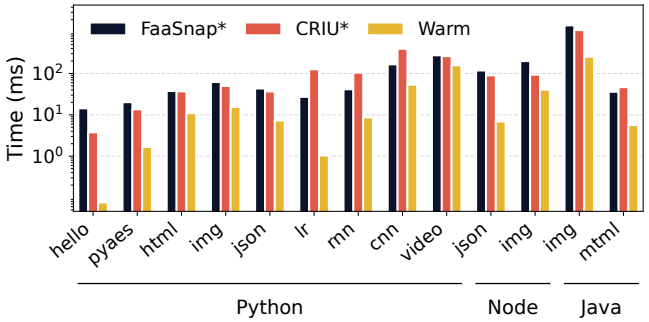


Figure 1: State-of-the-art restore-from-disk systems are still orders of magnitude slower than warm starts for serverless functions. Note the logarithmic y-axis.

Cold start delays stem from multiple sources, including container setup, language runtime initialization, library loading, and function-specific startup logic such as JIT compilation [22, 35, 42]. These steps often add tens to hundreds of milliseconds — frequently longer than the function's actual execution [10] — making them a major source of user-visible latency. Some mitigation strategies rely on *warm-state* approaches, such as keeping containers alive [4, 16] or leveraging *fork-based replication* [6, 10, 19, 43] to clone instances from a resident parent. While effective for scaling *active* workloads, these techniques only work when there is warm state already available in the cluster. However, maintaining warm state for the long tail of serverless functions is economically infeasible. Microsoft reports that 81% of applications are invoked at most once per minute [33], while Ant Financial observes that over 60% of functions experience cold starts more frequently than hot starts due to strict memory constraints [10]. Because rarely invoked functions cannot rely on a resident parent process, cold starts remain a major barrier to responsiveness in serverless systems.

The most effective existing approach to mitigating cold starts is to restore snapshots captured after a function has fully initialized [7, 23, 36, 39]. Over the last decade, work on fast snapshot restoration has focused on techniques such as lazy initialization [13, 44] (which populates process state

Language	Function	Warm Latency (μs)	Snapshot (Working Set)										Description
			VMAs		Delta Intervals		Private Pages		Shared Pages		Zero Pages	Working Set (MiB)	
Java	image	371,067	174	(82)	847	(327)	149430	(10392)	40,949	(1962)	(25119)	(146.37)	Rotate a JPEG
	mtml	6122	134	(36)	420	(69)	7093	(654)	40586	(126)	(2)	(3.1)	Matrix multiplication
NodeJS	image	45,512	241	(149)	446	(246)	26284	(5348)	27323	(1577)	(6677)	(52.7)	Rotate a JPEG
	json	8876	198	(103)	307	(138)	5488	(2060)	23464	(1304)	(275)	(14.2)	JSON (de)serialization
Python	html	11,171	81	(30)	199	(93)	3807	(1645)	3691	(475)	(64)	(8.5)	HTML rendering
	cnn	53,082	1913	(172)	3154	(571)	72801	(10672)	367651	(3461)	(437)	(56.9)	CNN inference
	hello	8	50	(17)	156	(58)	1167	(184)	2405	(198)	(1)	(1.5)	A no-op function
	image	15,540	192	(47)	328	(98)	5169	(1506)	5343	(541)	(11)	(8.0)	Rotate a JPEG
	json	7430	153	(38)	277	(96)	5003	(1633)	3999	(493)	(20)	(8.4)	JSON (de)serialization
	lr	1232	1294	(105)	1742	(166)	25462	(1930)	39852	(578)	(14)	(9.8)	LR inference
	pyaes	1726	49	(16)	157	(50)	1240	(304)	2401	(211)	(1)	(2.0)	AES encryption
	rnn	8571	312	(56)	1439	(164)	52385	(1523)	762571	(1013)	(58)	(10.2)	RNN inference
	video	222,057	393	(76)	702	(172)	7784	(1875)	56421	(1367)	(908)	(16.2)	Grayscale conversion

Table 1: Characterization of memory usage for various serverless functions. Delta intervals counts the number of contiguous sets of pages with modified/private application data. Values in parentheses refer to the working set, as opposed to the whole of the snapshot. For zero pages, only those in the working set are reported.

and memory on demand during execution) and working-set prefetching [7, 39] (which pre-loads only the pages that will be accessed into memory). However, even after applying these techniques, restore latencies are still significantly higher than warm starts. Figure 1 quantifies this gap and shows the end-to-end cold-start invocation times of two state-of-the-art snapshot/restore systems. They are 1.6×–188× slower than a warm start baseline.

In this paper, we show why this gap persists: current OS abstractions are a poor match for fast snapshot restoration in two ways: (1) the OS lacks an object format and interface for reloading memory efficiently, and (2) it provides no mechanism for reconstructing process metadata in bulk. We discuss each in more detail.

First, the OS provides poor support for reloading an application's memory during snapshot restoration. The state that actually needs to be recovered is a sparse, page-granular overlay: a relatively small set of modified pages scattered across large shared and anonymous mappings. To obtain good I/O performance, prior systems [7, 23, 39] reorder these pages on disk in predicted access order rather than virtual address order, producing a snapshot file that is effectively a jumble of pages laid out by predicted access time. However, neither standard binary formats nor the OS's mapping interface can efficiently express this sparse, reordered view of memory, forcing these systems to reconstruct the address space via thousands of tiny non-contiguous mappings or explicit page-wise copies, which scale poorly at restore time.

Second, the OS's lack of an interface to restore non-memory process state in bulk is responsible for large delays at restore time. To restore an application faithfully, a snapshot must recover not only its memory but also its file descriptors, signal handlers, timers, credentials, and other kernel-managed objects. However, today's OS interfaces assume incremental process startup, forcing restorers to recreate kernel objects through many serialized, fine-grained system calls rather than reinstating them in bulk. As a result, process-level tools must replay thousands of operations to rebuild this state (a process we refer to as *syscall replay*), substantially inflating restore latency. VM-level snapshotting sidesteps this replay, but only by capturing and restoring a much larger image, which increases the working set and adds delays due to guest OS processing at resume time.

To address these limitations, we introduce Spice, a system that co-designs an *object format* and *OS interfaces* for fast, elastic process restoration. Spice provides the **Snapshot Hybrid ELF** (SHELF) format, an ELF-inspired format that plays for snapshots the role ELF plays for program startup: it defines a compact contract between bytes on disk and the kernel's in-memory representation of a process. Whereas ELF assumes that a program can be described by a few large contiguous segments, SHELF generalizes this model with a page-granular indirection structure that can encode the sparse, reordered overlays produced by snapshotting. A single SHELF "segment" can thus contain arbitrary holes and page reorderings while still being presented to the kernel as a small number of contiguous virtual ranges.

On the kernel side, Spice introduces **spliceVMAs**, a new mapping mechanism that attaches SHELF's indirection structures directly to contiguous regions of a process's virtual address space and resolves page faults to disparate sources within a single VMA. spliceVMAs allow the kernel to retain a compact set of large mappings while loading pages from whatever on-disk layout SHELF chooses, rather than issuing thousands of tiny non-contiguous `mmap`s or performing page-wise copies. This decouples the snapshot's I/O-optimized representation from the address-space representation used by the kernel. Together, SHELF and spliceVMAs resolve the core tension between on-disk and in-kernel views of a process's memory: snapshots can freely reorder pages for high-throughput restore, while the kernel reconstructs the original address space without per-page mappings or metadata explosion.

For process metadata, Spice demonstrates that snapshots can remain at the process boundary without incurring the high costs of system-call replay. Using a library OS to prototype

bulk restoration of execution state, Spice reconstructs file descriptors, signal dispositions, threads, and other process state from compact serialized descriptions, reducing metadata reconstruction costs by 63–99% compared to replay-based restoration.

Together, these mechanisms show that high performance and memory elasticity need not be at odds. By removing both dominant bottlenecks—metadata replay and inefficient memory reconstruction—Spice reduces cold-restore latency from disk to within 0.6–18ms of a warm invocation (compared to 3.6–1197ms in existing systems), achieving up to orders-of-magnitude improvement over state-of-the-art. These results blur the line between cold storage and warm memory, enabling providers to more aggressively offload idle functions to disk without sacrificing the responsiveness required by user-facing workloads.

This paper makes the following contributions:

- We show experimentally that mismatches in OS interfaces for metadata restoration and memory-layout reconstruction are a central barrier to fast cold starts from disk.
- We introduce Snapshot Hybrid ELF (SHELF) and spliceVMAs, a co-designed snapshot format and kernel mapping abstraction that efficiently reconstruct sparse, reordered process address spaces.
- We present a compact metadata-restoration mechanism that recreates process OS state in bulk without system-call replay.
- We demonstrate that Spice significantly reduces cold-start latency from storage, outperforming state-of-the-art process-based systems by 7.5× and VM-based systems by 9.5× on average.

2 Background and Motivation

Snapshot-based cold-start mitigation requires choosing what state to capture and restore. Existing systems primarily choose one of two snapshot boundaries: the application process, represented by CRIU [3], or the entire virtual machine, represented by REAP [39] and FaaSnap [7]. This boundary determines both how much state must be restored and which operating-system mechanisms are available during restoration.

We first separate this snapshot-boundary choice from the deployment and isolation mechanisms used in serverless platforms. Serverless functions are commonly deployed either as code snippets assembled with a provider-managed runtime or as user-supplied container images. At invocation time, the provider materializes this package as an OS container containing the function process or processes, its language runtime, and its libraries. Depending on the platform's isolation model, this container may run directly using host OS mechanisms such as namespaces and cgroups, or inside a lightweight VM.

Thus, using a VM as the sandbox does not require using the VM as the snapshot boundary. A platform that relies on VMs for isolation can still snapshot the process or processes inside the container and later restore them inside a fresh VM; it need not snapshot the entire guest OS. We therefore treat process-level and VM-level restore as choices of snapshot boundary, rather than as consequences of the platform's sandboxing mechanism.

We evaluate these choices by measuring the end-to-end latency of cold serverless function invocations. To isolate the cost of restoring application state, we exclude non-function-specific container setup such as cgroups and namespaces, which can be prepared ahead of time and for which acceleration techniques are well explored [10, 19, 24, 26, 38].

2.1 The Snapshot Boundary: Processes vs. VMs

One foundational challenge for fast restoration is that modern operating systems lack a native interface to restore a process's kernel-managed state, or metadata. This state includes a wide array of resources: threads, file descriptors, timers, signal handlers, and more. Not only is there no single interface to reinstate this entire collection at once, but even restoring a single resource to its previous condition can require a complex sequence of operations.

Because there is no dedicated restore primitive, systems are forced into one of two suboptimal approaches: either (1) replaying the system calls that originally created the process, or (2) snapshotting and restoring the entire virtual machine it runs within. Both strategies push existing abstractions beyond their intended uses and are responsible for significant cold-start overheads.

Syscall replay. At one extreme, restore begins from an empty process and attempts to reconstruct every piece of kernel state — threads, file descriptors, memory mappings, and more — by reissuing the original system calls that created them. This strategy is expensive: there are often hundreds or thousands of such calls required, as most kernel objects lack dedicated import or restore interfaces to directly set them to their previous state. For example, restoring a single file descriptor may require a sequence of syscalls: `open` to create the file, `lseek` to set its offset, and `dup` to assign the correct descriptor number. Across complete function restores, this replay cost grows with function and runtime complexity. Figure 2 reports the initialization overhead of process-based restore for a range of serverless functions, along with the number of syscalls issued during restoration. Simple functions require hundreds of syscalls, while larger runtime environments require thousands, causing restore latency to rise accordingly. Because most of this state must be reconstructed before execution can resume, syscall replay directly inflates cold-start latency.

The userspace orchestration required for process restore adds further overhead. Rather than asking the kernel to in-

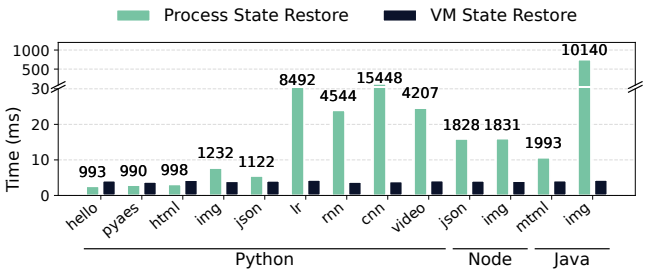


Figure 2: Initialization overheads for VM- and process-based restore; total syscall counts for process restore are shown above the bars. Process restore costs grow with process complexity due to syscall replay. VM restore costs are flat but still non-trivial, reflecting fixed-cost hypervisor setup operations.

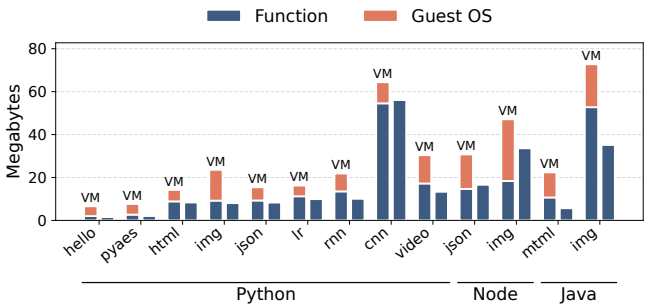


Figure 3: Working set composition for VM vs. process snapshots; the VM snapshot boundary increases the resulting active memory at restore time by 1.2–3.4×.

Python									Node		Java	
hello	pyaes	html	img	json	lr	rnn	cnn	video	json	img	mtml	img
50%	41%	22%	26%	23%	23%	40%	25%	40%	30%	25%	36%	19%

Table 2: Fraction of unmodified file-backed pages that, in a process-level snapshot model, can be reused and shared rather than reloaded from disk; VM snapshots cannot leverage this opportunity.

stantiate a saved process directly, CRIU runs as an ordinary userspace process and morphs its own address space into the restored one. This requires it to avoid collisions between its temporary restore state and the target layout: CRIU stages snapshot data in temporary regions, installs a small restorer binary into otherwise unused virtual address space, transfers control to it, unmaps CRIU's original code and data, and finally remaps the snapshot contents at their original virtual addresses. This elaborate dance avoids even more costly alternatives, such as reconstructing the target from a second process via `ptrace`.

Full-VM snapshots. At the other extreme, systems restore entire virtual machines from snapshots. Because the snapshot includes the guest kernel and its data structures, process metadata is already present when the VM resumes rather than being reconstructed through syscall replay. As shown in Figure 2, restore is reduced to a small, fixed set of hypervisor actions, such as reinitializing vCPU state, reattaching devices, and restoring other host-managed state. These operations complete in just a few milliseconds regardless of the complexity of the application running inside the guest.

The downside is that VM snapshots capture more than the function needs. Capturing the whole VM substantially increases both the stored snapshot image and the memory that must be present at restore time. Figure 3 compares the working-set composition of VM- and process-level snapshots, showing that VM working sets include substantial state from the guest OS at restore time.

Snapshotting at the VM boundary imposes additional memory efficiency costs. In Table 2, we show that 19–50% of the working set consists of pages that are *not function specific*, but are potentially shared with other processes, such as libc, a runtime interpreter, or other common libraries. In a process snapshot, any such pages that are already resident on the machine can be remapped from the page cache without any I/O; in a VM snapshot, these page reuse opportunities are not visible to the hypervisor and therefore it cannot make use of them.

Additionally, we found that after a prolonged pause, the guest kernel immediately experiences a "deferred housekeeping storm" due to the sudden clock jump. When the hypervisor restores the VM's virtual CPU, the guest kernel registers a sudden, massive jump in its system clock representing the time elapsed while paused. This clock jump immediately triggers all pending, timer-based background services and guest kernel housekeeping routines simultaneously, including periodic systemd daemons, guest memory reclamation passes, and critical kernel threads like RCU callback processing. Because these essential tasks are scheduled as soon as the VM becomes active, we found that even boosting the function process to a high-priority real-time scheduling class (`SCHED_FIFO`) offers only partial relief; critical guest kernel threads still preempt and delay the function's execution by up to 10ms, consuming 22–79% of the end-to-end response time.

Ultimately, capturing snapshots at the VM boundary reduces OS metadata restore costs but inflates resource consumption in many dimensions — larger stored snapshot images, more I/O and memory bandwidth consumed, and more duplicated memory on the host. Finally, the execution delays at restore time due to guest kernel interference waste compute and produce user-visible delays.

2.2 Repopulating memory

Even if metadata could be restored instantly, cold-start latency remains dominated by the work required to repopulate a process's address space. Existing snapshot/restore systems ultimately confront the same underlying limitation: the operating system provides only coarse-grained interfaces for constructing virtual address spaces. Interfaces such as `mmap`

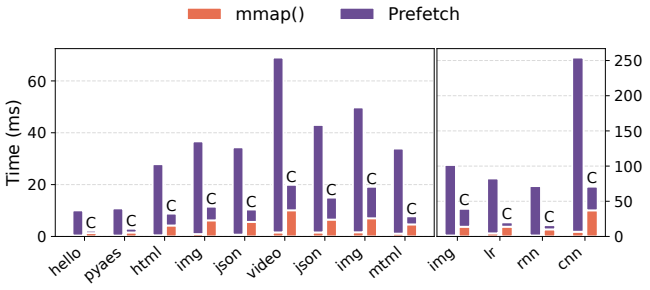


Figure 4: Time spent restoring memory and prefetching the working set under two snapshot file layouts. Storing working-set pages contiguously in the snapshot file (bars marked "C") greatly reduces prefetch time, but increases the cost of reconstructing virtual mappings. Leaving pages at their virtual-address offsets has the opposite trade-off: mapping reconstruction is cheaper, but prefetching is more expensive.

map contiguous file regions into contiguous virtual-memory regions, while restoring a snapshot demands fine-grained, page-level control. This mismatch forces restorers into an unavoidable trade-off between I/O efficiency and memory-management efficiency.

When an application starts, its address space is typically described by a small number of large virtual memory areas (VMAs). Each VMA corresponds to a contiguous region of virtual addresses backed by either a file or anonymous memory. During execution, however, this organization rapidly breaks down. Within a single VMA, page-level states diverge: some pages remain clean and continue to map to the backing file, others become private copy-on-write (CoW) pages due to writes, and still others are never touched. The kernel tracks these differences at page granularity through the page tables. However, address-space management interfaces such as `mmap()` remain segment-oriented: they create and manipulate contiguous virtual-memory regions with uniform backing and protection attributes rather than arbitrary collections of individual pages.

This segment-oriented model is highly effective for process startup. ELF, the standard executable format on Unix-like systems, describes a program image as a small number of large loadable segments. The kernel constructs an address space by mapping these segments and relies on demand paging to populate them lazily as execution proceeds. Because application memory is initially dense and contiguous, this approach requires only a small number of mappings and allows the kernel to rely heavily on demand paging, making ELF loading a highly optimized path in Linux.

A snapshotting system, however, seeks to reconstruct the memory of a running process rather than a freshly initialized one. At this point, the address space is no longer naturally organized as a small number of dense segments. Instead, the pages whose contents must be preserved form a sparse, page-level overlay on top of shared code and data. Moreover, the order in which those pages are accessed after restore is determined by application behavior rather than their virtual addresses. Consequently, the layout that is most efficient for storing and retrieving snapshot data need not mirror the layout of the virtual address space. Instead, arranging pages on disk according to their expected post-restore access order enables efficient prefetching and bulk retrieval through a small number of large sequential operations. Ideally, a snapshot representation would therefore store only pages whose contents cannot be reconstructed from their backing files and arrange those pages according to expected access order rather than virtual address order.

Realizing this design, however, requires a way to map a sparse collection of reordered pages back into their original virtual addresses. The kernel's mapping interface provides no way to directly map a sparse collection of pages, stored in arbitrary order within a file, onto the non-contiguous virtual addresses they originally occupied.

As a result, whenever pages that were adjacent in virtual memory are stored in different locations within the snapshot file, `mmap` requires a separate mapping operation to place each discontinuous region back into its original location. A snapshot for which pages have been clustered or reordered for sequential I/O can therefore expand into thousands of tiny VMAs, each mapping only a single page or a small range of pages. For the workloads in Table 1, we found that such reordering would increase the number of VMAs by up to 32×.

This proliferation of VMAs incurs two costs. First, each VMA creation requires kernel memory allocation and insertion into the process's VMA tree, operations that dominate CPU time during restore and scale poorly with tree size. Second, VMAs consume non-trivial kernel memory (approximately 300 bytes per VMA in Linux), so restoring thousands of page-sized VMAs per function introduces measurable overhead in the kernel's memory footprint.

Figure 4 illustrates this trade-off. Clustering working-set pages into a contiguous on-disk region allows snapshot data to be fetched with a small number of large sequential operations, minimizing both I/O overhead and CPU work in the loader and kernel. However, restoring those pages into their original scattered virtual addresses requires many independent mappings, resulting in a proliferation of VMAs. Conversely, preserving the original address-space layout keeps the number of mappings small, but forces the loader to retrieve snapshot data from many disjoint locations and can leave a long tail of page faults. Because current OS interfaces cannot decouple on-disk layout from virtual-memory layout, existing systems must choose between layout-efficient storage and mapping-efficient restoration.

Existing systems resolve this trade-off in different ways. CRIU [3] preserves the original VMA layout and saves only dirty modified pages in its snapshot, recreating VMAs and

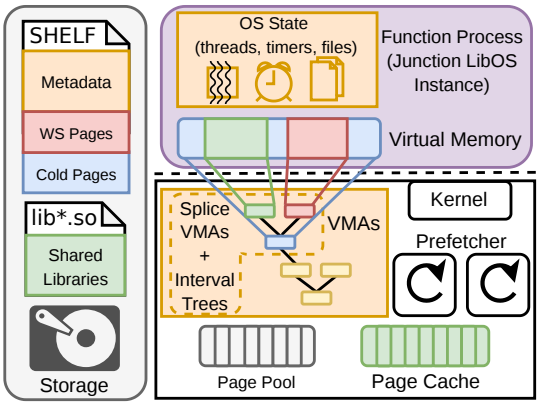


Figure 5: Overview of Spice's architecture.

copying dirty data back into place. This avoids VMA proliferation but requires scattered reads and explicit copying into the target address space. Replayable-Execution [42] reduces copying costs by mapping small modified segments directly from files and relying on demand paging, but doing so across many tiny regions amplifies VMA creation overhead.

REAP, Sabre, and FaaSnap [7, 23, 39] focus on prefetching optimizations and reorder working-set pages into contiguous regions for efficient prefetch. REAP and Sabre issue large sequential reads and then copy each page into its scattered virtual destination; execution cannot resume until this bulk copy completes. FaaSnap overlaps prefetch with execution by asynchronously `mmap`ing reordered segments into the process and warming the page cache. However, because file-backed writable memory in Linux employs CoW semantics, every write in the restored function induces a minor fault and a page allocation; in virtualized environments such as Firecracker, each fault induces a VM exit, adding hundreds of cycles before fault handling even begins.

Across all of these designs, the root cause is the same: the operating system lacks a page-granular mapping abstraction that decouples the layout of data on disk from its placement in virtual memory. Existing memory-management interfaces therefore force restorers to choose between I/O-efficient layouts and mapping-efficient address-space construction. In the next section, we show how Spice breaks this trade-off by extending the segment-based loading model embodied by ELF with support for sparse, reordered page-level overlays. The result retains the efficiency of ELF-style segment loading while enabling page-level reordering and sparsity.

3 Spice Design

We design Spice to address the challenges outlined in Section 2. Spice is a serverless execution engine that minimizes cold-start latency by restoring snapshots from storage at the *process-OS* boundary, rather than restoring an entire VM. This boundary reduces the amount of captured state while avoiding the restore-time overheads of VM-based approaches. As

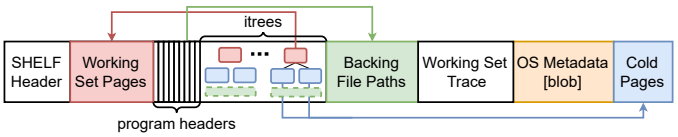


Figure 6: The layout of a SHELF file.

shown in Figure 5, Spice consists of several key components.

First, we introduce a new snapshot object format, Snapshot Hybrid ELF (SHELF), co-designed with a novel kernel memory abstraction: the spliceVMA. Together, these mechanisms enable the kernel to efficiently restore scattered and reordered memory pages directly from a snapshot file. Second, Spice includes a high-performance kernel memory engine that loads SHELF snapshots and performs both snapshot prefetching and page-table-entry (PTE) prefaulting to accelerate restoration.

Finally, user code in Spice executes within the Junction library operating system (LibOS), which serves as our container runtime. We selected Junction because it manages the vast majority of process metadata in userspace, making it well suited for prototyping a metadata restoration interface. We discuss this design choice in greater detail in Section 3.4. The remainder of this section describes each component of Spice in detail.

3.1 The SHELF File Format

We design the Snapshot-Hybrid ELF (SHELF) format, an evolution of the ELF object format that supports efficient loading of an already-running process's memory from disk. Figure 6 shows the layout of a SHELF file. SHELF is intentionally organized to promote access locality. Immediately following the initial header, SHELF stores the process's working-set pages in time-of-access order in a contiguous array. This allows the system to immediately execute an initial high-throughput sequential disk read to prefetch these pages at load time, before performing any additional parsing.

Like ELF, SHELF stores an array of program headers, one describing each VMA that existed in the process's address space at snapshot time. SHELF's program header diverges from ELF's in two key ways. First, each program header can describe backing memory that is primarily sourced from a different file, by referencing a pathname included in the subsequent file paths section. A program header that does not point to another file uses anonymous memory as its main backing source. Second, the program header describes non-contiguous memory regions within the SHELF file that are overlaid on the main backing source. Because these regions vary in size, each header points to a collection of memory intervals stored in a later section using a layout optimized for restoration by spliceVMAs (§3.2). Each interval describes a sub-range of the program header's address space and specifies the offset in the SHELF file containing the corresponding

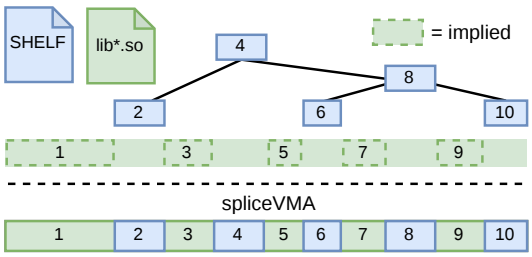


Figure 7: A spliceVMA and its interval tree. The interval tree lets a spliceVMA overlay sparse SHELF-backed ranges onto an existing file-backed (or anonymous) mapping.

data. The pages referenced by an interval reside either in the working-set section at the beginning of the file or in the cold-data section at the end. Any address range not covered by an interval is sourced from the backing memory (e.g., existing file-backed pages or anonymous zero pages).

Next, the SHELF file includes a trace of page accesses to guide prefetching. Each trace entry records the virtual address accessed, a timestamp, and a small amount of metadata about the access, such as whether the trace predicts a write. Our implementation includes records for pages already present in the working-set section, as well as records for accesses to zero pages and pages from other shared files. We discuss this design choice further in §3.3.

Finally, there is a section reserved for serialized OS metadata that is not memory-related. This section's structure is not specified by SHELF, and can be laid out as needed by the metadata restore implementation.

3.2 Splice VMAs & Interval Trees

SHELF's format alone is not sufficient to avoid the restore-time costs identified in §2.2; the kernel must also be able to *consume* the format without devolving into per-page work or VMA explosion. To this end, we introduce the *spliceVMA*, a new kernel VMA type co-designed with SHELF. A spliceVMA represents a mixed-source mapping: a single contiguous virtual address range whose pages may come either from a snapshot file or an underlying file-backed or anonymous mapping.

The design goal for spliceVMAs is to minimize compute work on the restore path. In particular, we want to (1) avoid per-page operations when reconstructing the address space, (2) avoid doing any work at all for pages that are never touched after restore, and yet (3) guarantee that when execution diverges from the predicted working set, any access still sees the exact bytes that were present at snapshot time. The key structure that enables this is the *interval tree* specified in the SHELF file for each program header. Figure 7 shows the representation of a spliceVMA and its interval tree.

For each VMA described in SHELF, the snapshot generator produces an interval tree that compactly encodes the page-granular overlay for that region. On disk, this tree is stored in a layout that can be used directly by the kernel without any deserialization. Concretely, each interval tree is a pre-balanced, cache-friendly B^+ tree stored as a contiguous array of nodes. Each node describes a range of virtual addresses and the corresponding offset in the SHELF file where the data for that range resides. At restore time, the loader maps this array into memory and attaches a pointer to it in the spliceVMA; no rebalancing, allocation, or pointer-fixup is required. On a page fault, the spliceVMA consults its interval tree to determine whether the faulting address falls within a snapshot interval. If so, the fault handler reads the page from the snapshot image at the recorded offset; otherwise, it falls back to the original backing source (the file named in the program header or zero-fill for anonymous memory). This guarantees correctness even when the function's execution path diverges from the original access trace.

Intuitively, the interval tree is a compacted, read-only representation of the application's original page tables for a particular region that enables lazy but correct restoration of that information. The kernel's VMA tree (implemented with maple trees [18]), must support frequent and likely concurrent insertions and deletions, so it is built from dynamically allocated, pointer-linked nodes. Conversely, the spliceVMA interval tree is static: it is constructed offline, never modified at runtime, and always perfectly balanced. This static, array-based layout makes lookups cheaper and more cache-efficient than traversals of the general-purpose VMA map, and it imposes no additional overhead on the restore path beyond wiring in the pointer from the spliceVMA to the already-constructed tree. Because interval trees are built offline and so cheap to query, SHELF can freely introduce arbitrary fractures and reordering of intervals to optimize the on-disk layout for I/O, without inflating the cost of address-space reconstruction. We evaluate the performance of querying an interval tree in §5.

Although the interval tree is immutable, spliceVMA-backed memory remains compatible with ordinary virtual-memory operations such as `munmap`, `mremap`, and `mprotect`. The interval tree describes the contents of the original snapshotted region; it is not the mutable representation of the process's current VMA layout. When an operation splits or moves a spliceVMA, the resulting VMAs continue to reference the same interval tree and record their position relative to the original snapshotted range. Once pages have been restored, subsequent changes are reflected in the process's page tables and VMA metadata, as in normal execution. Thus, runtime memory-management operations mutate the VMAs and PTEs around the interval tree, while the interval tree remains a read-only index for the snapshot contents.

3.3 SHELF Loading & Prefetching

The SHELF loader is a throughput- and latency-optimized kernel engine that ingests a SHELF file and recreates the snap-

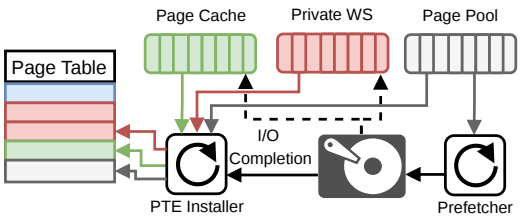


Figure 8: Spice's prefetching architecture.

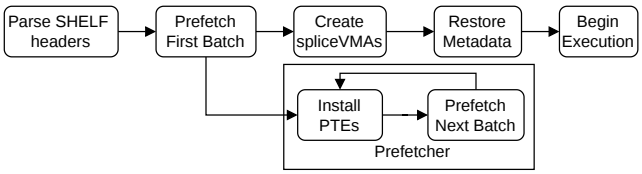


Figure 9: `reexec()` workflow

shotted memory layout. It is exposed through a new system call, `reexec()`, which augments the caller's address space with the contents of a SHELF snapshot. The loader is guided by SHELF's access trace, which records the pages expected to be touched after restore, whether those pages are private snapshot pages from the SHELF file, shared pages from other backing files, or anonymous zero pages, and whether each page was written during profiling. Figure 8 shows the components of the SHELF loader in the kernel, and Figure 9 shows the timeline of restore operations.

Synchronous restore path. `reexec()` is designed to minimize the synchronous time spent inside the system call. Its blocking path performs three steps. First, it initiates reads for the private working set. This is cheap because private working-set pages are packed contiguously in the SHELF file, reducing both file-system block-lookup overhead and the number of I/O submissions needed to fetch them. Internally, these reads are still split into independent I/O submissions so that completions arrive incrementally and can drive page-table installation before the entire private working set has been read.

Second, while the reads are in flight, `reexec()` constructs the process's spliceVMAs in bulk and wires each spliceVMA to its corresponding interval tree. This reconstructs the virtual address layout without per-page mapping operations and without touching pages that may never be accessed. Third, once the virtual address space exists, `reexec()` consults the access trace and installs an initial batch of PTEs for traced pages whose contents are already available, including zero pages, page-cache hits, and private pages whose reads have already completed. It then returns control to the restored process.

Asynchronous prefetching and PTE installation. After `reexec()` returns, kernel prefetch threads continue processing the trace concurrently with function execution. They initiate I/O for shared working-set pages, which come from the original backing files and are often scattered across many files in short intervals. They also install PTEs as page contents become available: for shared pages already resident in the page cache or fetched by the prefetcher, for zero pages that can be mapped immediately, and for private pages as the private-working-set reads complete.

A key design choice is that the loader aggressively installs page-table entries rather than relying on minor faults. In the serverless setting, accesses after restore are scattered across large regions, and minor faults can quickly dominate latency. The prefetcher therefore attempts to prefault traced accesses, including zero pages, so that the restored process rarely incurs even minor faults on its hot path.

Page-placement policy. The loader chooses where page contents should reside based on whether they are reusable and whether they are likely to be written. Shared pages from backing files are populated through the page cache, enabling reuse across functions. Private pages are loaded into newly allocated anonymous memory rather than inserted into the page cache, because they represent snapshot-specific anonymous state and would otherwise compete with broadly reusable file-backed pages for cache space.

This is a policy choice rather than a limitation of SHELF. For functions invoked frequently enough that repeated restores of the same snapshot are common, a future implementation could route read-mostly private pages through the page cache while directing pages likely to be dirtied into anonymous memory to avoid future CoW faults.

Zero pages and allocation policy. The write information in the trace is especially useful for zero pages. For zero pages that are unlikely to be written, Spice installs mappings to the shared CoW zero page. For zero pages predicted to be written, it instead installs private pre-zeroed pages so that the restored function can write without taking a fault.

Restoring a snapshot also puts substantial pressure on physical-page allocation. Both loading private working-set pages and installing writable zero pages require the loader to obtain large numbers of physical pages during a short restore window. Although Linux provides bulk page-allocation interfaces, we found that the per-page overhead still accumulates for snapshots containing tens of thousands of pages, making allocation a visible component of restore latency. To avoid placing this work on the invocation critical path, Spice maintains a reserved pool of preallocated pages that can absorb the allocation bursts caused by restore. The loader draws from this pool when populating private pages and writable zero pages, and falls back to ordinary allocation only when the pool is exhausted.

Together, the `reexec()` fast path, spliceVMA-based mappings, trace-guided prefetching, and proactive PTE installation allow Spice to reconstruct the snapshotted address space with minimal synchronous work, while overlapping I/O and page-table installation with resumed execution.

3.4 Metadata Restore

An effective snapshot/restore mechanism must choose a boundary that can be restored quickly without forcing the system to capture unnecessary state. The process boundary is attractive because it is much lighter weight than a full VM snapshot, while still containing the execution state relevant to the application. In principle, a production implementation could capture and restore this state directly in the Linux kernel by adding restore support for the relevant per-process kernel data structures. In this work, we instead use Junction's LibOS [15] as a practical substrate for prototyping the same process-boundary abstraction.

Junction uses a single-address-space LibOS model: the application and LibOS execute in the same virtual address space, and the LibOS virtualizes the Linux process and system-call interface seen by the application. This means that much of the process metadata that would otherwise be maintained as Linux kernel state is represented by LibOS-managed userspace objects. As a result, our prototype can restore most process metadata by reconstructing these LibOS objects, rather than modifying Linux to serialize and rebuild each corresponding kernel data structure.

Virtual memory remains the important exception. Even with a LibOS, the host kernel ultimately controls the hardware address space, including VMAs, page tables, and page-fault resolution. Spice therefore provides explicit kernel support for virtual-memory restore: the SHELF loader reconstructs the restored address space and installs page-table mappings, while the LibOS reconstructs the remaining process metadata. Open file descriptors are handled lazily, with the LibOS recording enough information to reopen or reconstruct descriptors when they are first used.

This use of a LibOS is primarily methodological. It lets us evaluate what a dedicated process-restore interface can provide without first re-engineering all of Linux's per-process metadata paths. A Linux-native implementation could provide analogous serialization and reconstruction logic inside the kernel. Our prototype instead demonstrates that, once the kernel exposes the right fast path for restoring virtual memory, the rest of process-boundary restore can be implemented efficiently enough to support very fast cold starts.

We considered two possible approaches to capturing the LibOS-managed metadata. One option was to ensure that all metadata objects were located in a dedicated, contiguous memory region that could be snapshotted and restored by bulk copying. The alternative was an object-wise serialization approach, where each process-related LibOS object implements a custom serializer and deserializer that is invoked per object at runtime. We chose the latter for several reasons. First, the LibOS uses C++ features such as dynamic heap allocation, slab allocators, RCU, and smart pointers, which make it impractical to constrain all metadata to a single layout-stable region. Second, not all in-memory state needs to be preserved: custom serializers can omit transient or quiescent structures such as locks, wait queues, and other synchronization artifacts that are guaranteed to be inactive at snapshot time. Third, per-object serializers enable space-efficient encodings tailored to the object type. For example, LibOS pipes are implemented as ring buffers; by serializing only the live portion of each buffer, we avoid copying unused capacity.

This design also circumvents a central limitation faced by userspace restorers such as CRIU. Because CRIU must perform restoration from within an already-running userspace process, it has limited control over the address-space layout of the process being restored. CRIU's restorer must map its own code and data while simultaneously injecting the restored process's memory mappings, and conflicts may require relocating regions at runtime. These relocation steps add CPU overhead. By contrast, our restore path executes from the LibOS's "kernel side," where we control the address space during restoration. This lets us arrange the LibOS and application mappings so that LibOS-internal mappings do not conflict with the restored application layout, avoiding the relocation overhead that CRIU can incur.

Metadata serialization starts from the root task object and emits a sequential, address-independent archive of the reachable LibOS state that must survive restore. Each object's serializer records the fields needed to reconstruct that object and invokes serialization for referenced objects that should also be preserved. Deserialization reverses this process, reading metadata from the SHELF file using a buffered reader to amortize I/O costs and reconstructing the LibOS object graph. In our measurements, metadata restore completes in as little as 0.9ms, compared to 2.6ms for CRIU, and no more than 7.5ms, compared to 749ms for CRIU. Despite the different techniques, we find that the metadata snapshots produced by Spice and CRIU are comparable in size across functions.

To ensure low invocation latency, we also avoid booting a LibOS instance on the restore critical path. Instead, we maintain a pool of freshly initialized LibOS instances that are ready to accept restored tasks. Each instance begins in a cleanly booted state, so restoring consists only of mapping the snapshot's memory image and reconstructing the object graph before execution resumes.

4 Implementation

We implement Spice on Linux 6.5.0 with three main components. First, we modify Junction's library OS [15] to support snapshotting and bulk metadata restore for unmodified Linux binaries running inside its `seccomp`+`chroot` sandbox. Second, we implement `shelftools`, a 7.4 KLoC Rust toolchain that constructs and rewrites SHELF files. Third, we add a 7.1KLoC kernel module that introduces the `reexec()` system call, spliceVMAs, and the SHELF loader and prefetcher. Our implementation is open source at https://github.com/JunctionOS/spice-ae.

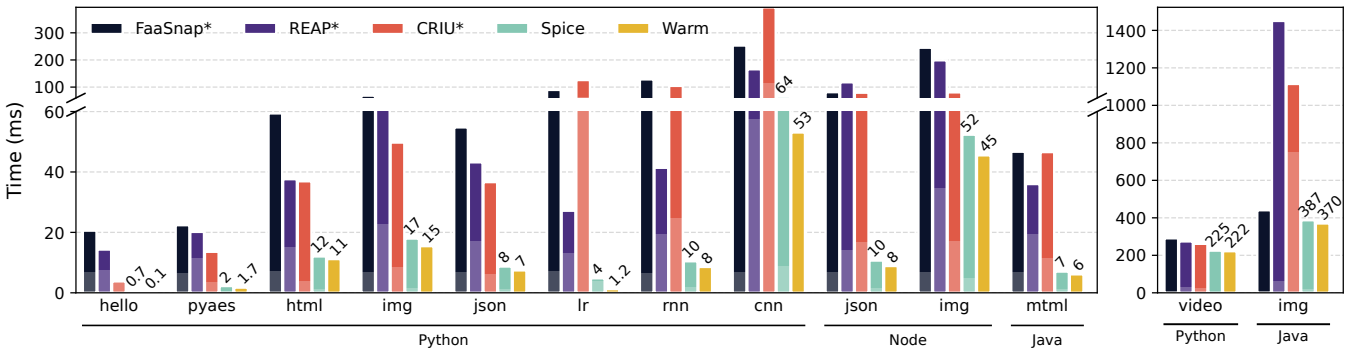


Figure 10: Spice achieves end-to-end cold-start latencies significantly closer to warm invocations than existing systems. The faded lower portion of each bar shows restore latency prior to executing the function.

User-level snapshot pipeline. We add a per-language shim to perform function invocations and prepare for snapshots. A snapshot proceeds in three steps. First, the LibOS pauses all function threads at a safe point and walks its internal object graph starting from the task root, and serializes them using the `cereal` C++ library. Second, the LibOS dumps the contents of each VMA in the process into temporary contiguous segments in a provisional SHELF file, without attempting to remove zero or shared pages. This keeps the in-process snapshot path simple and sequential. Third, `shelftools` runs offline to rewrite this file into the final SHELF layout: it removes unmodified file-backed pages and zero pages, deduplicates identical private ranges across snapshots, builds the balanced interval trees required by spliceVMAs, and emits the per-page access traces that guide prefetching.

Kernel loader and working-set profiling. We implement working-set estimation in the kernel, co-located with the loader, to avoid perturbing access patterns. In profiling runs, the module records the pages that fault after `reexec()`. The next run uses this trace to drive prefetching and prefaulting. We iterate profiling and replay until the trace stabilizes and then ship the resulting SHELF as the production snapshot for that function. SHELF files are agnostic to the working-set estimation mechanism; while we use a simple page-fault-based profiler, other working-set estimators could generate the same trace format.

Snapshot-size reductions. Our language shims cooperate with Spice by running garbage collection and dropping language-level caches when signaled before snapshotting. On the host side, we translate `MADV_FREE` into `MADV_DONTNEED` during snapshot preparation and trim each thread's stack by discarding unused stack regions above the current stack pointer. We found that optimizations like these were able to reduce snapshot sizes by several MB in some cases.

5 Evaluation

We aim to evaluate Spice by answering the following questions:

1. How well does Spice reduce end-to-end cold-start latency in the context of existing snapshot/restore systems (§5.1)?
2. How does each element of Spice's design contribute to its performance (§5.2)?
3. Does Spice's centralized prefetching design scale (§5.3)?
4. How sensitive is Spice to the speed of the device used for snapshot storage (§5.4)?

Experimental Setup All experiments are conducted on a machine with an Intel(R) Xeon(R) Gold 5420+ with 28 cores running at 2.8GHz and 128 GB of RAM. Our storage device is a Crucial T705 NVMe drive with a max sequential read bandwidth of 13,600MB/s and 1400K IOPS over PCIe 5.0. Unless otherwise stated, we run all experiments from this drive and use it to store all SHELF files and shared libraries used during function invocations. We use the commonly evaluated FunctionBench [21] test suite, based directly on the artifact from [39]. We also ported two workloads to Node.js and Java, runtimes that are popular in serverless functions and exhibit more complex behaviors than Python. Table 1 summarizes the functions used to evaluate Spice. For a fair comparison, we do not benchmark function-agnostic container setup costs, such as configuring isolation mechanisms, because this work can be pre-initialized and shifted off the invocation critical path.

5.1 End-to-End Latency

To understand how Spice performs in the context of existing systems, we compare to existing snapshot/restore systems that restore snapshots entirely from storage without relying on any warm state. As discussed in §2, this includes CRIU [3] which snapshots at the process boundary, and both FaaSnap [7] and REAP [39] which operate at the VM boundary. Both FaaSnap

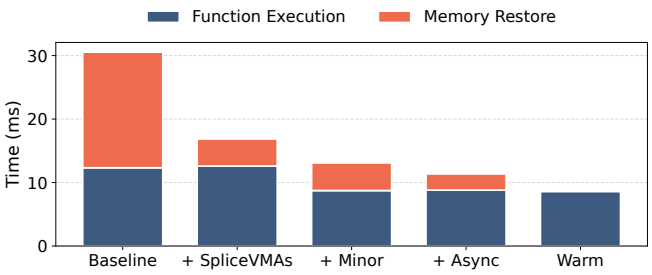


Figure 11: Ablation study of Spice's memory restore optimizations on the RNN serving Python function.

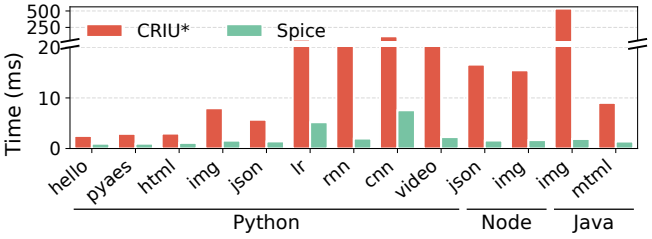


Figure 12: OS metadata restore time comparing CRIU to Spice. Spice lowers latency through a dedicated interface for restoring OS state and compact serialization.

and REAP employ working-set prefetching, while CRIU does not. We made every effort to tune existing systems to their peak performance, and our efforts yielded large performance gains for these systems relative to their out-of-the-box performance; therefore we refer to each of them with an asterisk in their name. In light of our observations regarding scheduler interference in VMs (§2.1), we modify the VM-based systems to use the `SCHED_FIFO` scheduling class for the function's process to minimize interference from other tasks running the guest. By default, CRIU copies every dirty page from the snapshot file back into memory. In-line with [42], we modified CRIU to use lazy mappings instead. While this results in slower execution time due to page faults, it provides a large win on end-to-end latency.

Figure 10 shows the end-to-end function invocation latency when restoring snapshots from storage with an entirely cold page cache. Spice is able to reduce latency significantly in all cases, by 17–96% compared to FaaSnap*, by 18–95% compared to REAP* and 14–96% compared to CRIU*. We show the latency of a warm invocation, which had been previously invoked several times but has cold micro-architectural state (e.g., CPU caches, TLB). This is a good approximation for an invocation on an otherwise busy system that is well utilized by many densely packed functions [4]. Spice is 1.01–6.34× slower than a warm invocation. We find the largest component of the remaining overhead comes from VMA creation, which cannot be parallelized with execution. Spice has the greatest impact on short-running functions, where restore latency can dominate execution time, but it also benefits longer-running functions by avoiding faults through prefetching and proactive PTE installation.

5.2 Microbenchmarks

Memory restore. We evaluate the impact of Spice's memory loading improvements in Figure 11, where we incrementally enable features of the SHELF loader and prefetcher for the RNN workload. In the baseline, the working set is packed contiguously in the snapshot file and prefetched from userspace. This necessitates creating 3212 VMA mappings to place the working-set pages in the proper location in the virtual address space. Adding spliceVMAs and batch creation of VMAs in the kernel dramatically lowers the time spent restoring virtual memory. Installing PTEs to prevent minor faults removes a significant portion of latency by avoiding thousands of kernel crossings during execution. Finally, asynchronous prefetching allows us to begin function execution sooner without incurring faults. With all optimizations, end-to-end function execution is within 2ms (23%) of a warm invocation, compared to 21ms (2.5× warm) added in the baseline case.

Cache State	# Intervals	Interval Trees	Maple Trees
Hot	10	4	41
	100	8	59
	1000	11	97
Cold	10	248	857
	100	278	874
	1000	284	1110

Table 3: Average lookup latency (cycles) for spliceVMA interval trees vs. Linux maple trees.

Metadata restore. Spice adds an interface to Junction for bulk restoring OS metadata, including threads, file descriptor state, signal handlers, and timers. Figure 12 compares the time spent on metadata restore for each workload to CRIU*, which reconstructs this state using thousands of syscalls. In all cases, Spice's metadata restore latency is significantly lower than that of CRIU*, demonstrating that snapshotting at the process boundary is a reasonable alternative to whole-VM snapshots.

Interval tree performance. Interval trees bridge Spice's file format and kernel abstraction, and are accessed on critical prefetching and fault handling paths. We compare their performance with that of the optimized B-Tree implementation used in Linux for VMA lookup (a step that typically precedes an interval tree lookup) in Table 3. Interval trees vastly outperform Linux's maple trees due to their contiguous, cache-friendly layout. In contrast, Linux's maple trees must handle concurrent updates and dynamic resizing.

Prefetcher performance. Figure 13 shows the rate at which Spice's prefetcher threads perform different restore operations. Prefetching shared pages is substantially more ex-

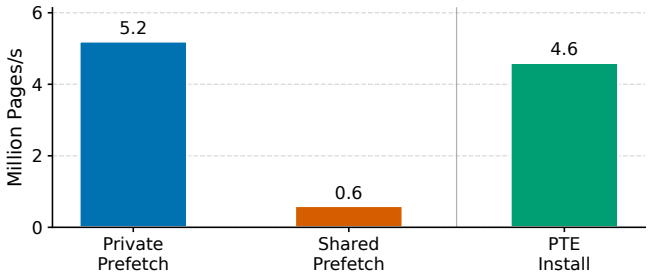


Figure 13: Peak operation rates for Spice prefetcher threads. Contiguous private pages can be prefetched at a much higher rate than scattered shared pages, which require more per-page lookup and I/O work. PTE installation also sustains a high rate, allowing it to keep pace with prefetching.

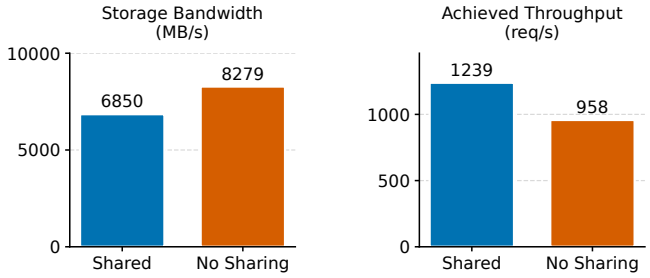


Figure 14: Storage bandwidth consumption and achieved throughput at 25 concurrent restores with and without page cache sharing. Sharing consumes less bandwidth and allows the system to achieve higher throughput.

	hello	pyaes	html	img	json	lr	rnn	cnn	video	node_json	node_img	mtml	java_img
Junction	.076	1.6	10.9	15.4	7.3	1.0	8.6	53.3	156.8	6.9	40.8	5.6	253.1
Linux	.001	1.3	10.2	14.3	5.9	.512	7.9	55.9	160.1	5.8	39.6	5.4	251.1

Table 4: Warm execution latencies (ms) for each function using Junction and Linux. Performance is largely similar between the two regimes.

pensive than prefetching private pages because shared pages are often scattered across many files and short extents, requiring per-page work to locate the backing file and issue the corresponding I/O. In contrast, SHELF stores private working-set pages contiguously, allowing Spice to issue larger, more efficient reads and prefetch at a much higher rate. Although PTE installation proceeds in observed access order rather than virtual-address order, Spice still sustains a high installation rate, allowing PTE installation to keep pace with prefetching.

Impact of sharing. Spice is designed to exploit the page cache to share pages from files (e.g., common libraries) when possible. Sharing in the page cache improves memory utilization, reduces storage bandwidth consumed, and can improve CPU cache efficiency [15]. We evaluate the benefit of this design by showing the storage bandwidth consumed and effective invocation throughput under a heavily concurrent function invocation workload (described in §5.3). We compare Spice with a version that doesn't allow sharing in the page cache, and instead stores the entirety of the snapshot in the SHELF image. Figure 14 shows that page-cache sharing reduces consumed I/O bandwidth by 20% and increases achievable invocation throughput by 30%. Without sharing, the system moves closer to storage saturation, causing prefetching to fall behind execution and exposing the restored functions to more page faults.

Impact of LibOS. Spice uses the Junction LibOS to implement fast OS metadata restore. Although Junction can accelerate kernel-intensive workloads through faster userspace implementations of standard OS services, our workloads are not kernel-intensive and therefore derive little benefit from these optimizations. To isolate the effect of Junction from the improvements shown relative to other systems, we report the warm execution latency of each function running in Junction and in Linux in Table 4.

5.3 Scaling

Spice relies on dedicated kernel threads to handle the asynchronous portions of prefetching and PTE installation for all restores. We evaluate Spice 's ability to scale in Figure 15, where we vary the number of concurrent restores and measure the achieved throughput. We construct this experiment using a mixed workload derived from the Azure public serverless traces [33]: we scale trace invocation durations to the execution time range of our workloads, map each invocation to the closest function in Table 1, and use the resulting distribution as the function mix. Our results show that Spice scales well, even under a mixed workload, achieving 76% of ideal throughput (extrapolated from the throughput at concurrency 1) at 25 concurrent restores.[1]

5.4 Sensitivity to Storage Device Performance

To understand how sensitive Spice is to storage performance, we evaluate it on a slower disk: a Micron 5400 PRO Series SSD rated for 540 MB/s and 95K IOPS. This experiment stresses the prefetcher by increasing I/O latency and reducing available bandwidth, allowing us to measure whether Spice can still hide restore work behind resumed execution when storage becomes a larger bottleneck.

Figure 16 compares performance on the fast and slow disks for three workloads with different execution latencies. We also compare this sensitivity against FaaSnap*. A single read

[1]There are 25 cores available on our machine for applications, after two are dedicated for prefetching and one for Junction's scheduler.

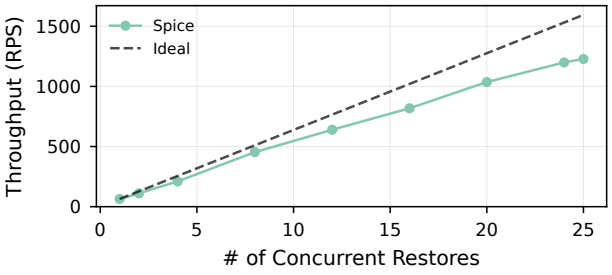


Figure 15: Aggregate throughput of function invocations with Spice as the number of concurrent restores increases, with a mix of all functions shown in Table 1.

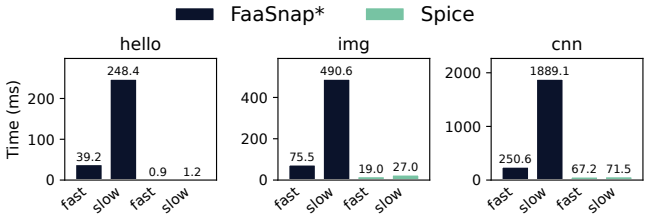


Figure 16: Impact of SSD speed on restore latency. The faster SSD has 2.8× lower read latency and 25× higher throughput than the slower SSD. Spice hides much of the additional latency of slower storage by prefetching asynchronously with execution, so restore latency is minimally affected as long as disk throughput is not the bottleneck.

takes 58 μs on the fast disk and 163 μs on the slow disk. Despite the increase in latency, Spice hides most of the difference by overlapping prefetching with resumed execution. This is possible because the slower disk still provides enough bandwidth to satisfy the demand of a single function restore.

FaaSnap*, by contrast, is much more sensitive to the slower disk. Because it already incurs many major faults during restore, increasing storage latency directly increases the cost of those faults and causes restore latency to grow by more than the proportional increase in single-read latency. We suspect this amplification is caused by the effects discussed in §2.1: delaying function completion gives other processes in the guest more opportunity to wake up, further polluting the working set and increasing interference.

6 Related Work

Working set estimation and prefetching. Working set estimation and prefetching have long been studied in serverless and other contexts [7, 39, 46]. Spice adds a dedicated kernel interface that accelerates prefetching and prefaulting; existing systems could benefit from this interface. Sabre [23] leverages a compression engine to improve effective I/O bandwidth from relatively slower SSDs; this is orthogonal to our approach. Faascale [47] and AFaas [10] both prefault EPT entries to reduce minor fault costs for VMs.

Fork scaling. Recent fork-like approaches [6, 10, 13, 19, 43] achieve low startup latency by cloning warm function instances. These systems are inspired by the kernel's `fork()` mechanism for cloning a process, and extend this concept to cloning containers [10, 13] or leveraging remote memory across RDMA or CXL [6, 19, 43] to create new function instances with demand paging. These techniques are effective for rapid horizontal scaling of functions — as long as a warm parent exists. As discussed in Section 1, the invocation patterns for most functions make retaining warm state for them economically infeasible. Spice targets this set of functions to enable fast cold starts from disk.

State reuse. Some systems aim to reuse partial state within a machine by creating stacks of increasingly specialized snapshot images [9, 10]. Because these stacks enforce a strict lineage of changes, they are effectively limited to sharing a single common base rather than the diverse mixtures of libraries used across functions. Medes [32] deduplicates memory across keep-alive containers so that more functions can remain warm, but still relies on long-lived sandboxes and existing OS restore interfaces. In contrast, Spice preserves large shared file-backed regions through spliceVMAs and focuses on making cold restores fast enough that platforms can rely less on aggressive keep-alive policies. VAS-CRIU [40] enables fast CRIU restores by keeping all pages in memory.

Sandboxing and the snapshot boundary. Existing systems use lightweight VMs [4, 7, 16, 23, 39], processes [22, 36, 40, 42], or containers [25, 32] as both their sandbox and snapshot boundary. As we discussed in Section 2, these approaches suffer from performance challenges. Some systems use alternative approaches for sandboxing. Faasm [34] relies on a Wasm runtime as the isolation mechanism, which offers low startup costs but higher end-to-end execution time compared to native Linux execution due to the cost of SFI [20]. SEUSS [9] implements a unikernel, tailoring its sandbox to executing a specific function. Unlike Spice, SEUSS requires backporting to support additional language runtimes, complicating deployment on existing FaaS platforms. AlloyStack [44] accelerates LibOS initialization via on-demand OS component loading. Spice could use similar techniques to avoid restoring some OS metadata, however, the time savings would be small.

Cold-start mitigation policies. Several systems try to reduce the impact of cold starts by increasing the fraction of requests that hit warm instances. They pre-warm sandboxes, tune keep-alive policies, or reuse containers to keep functions alive longer [5, 12, 16, 25, 27–30, 45]. Spice is complementary: instead of trying to avoid cold starts via longer warm lifetimes, it reduces the cost of the cold path itself, so platforms can rely less on aggressive keep-alive and prewarming. These approaches lower tail latency but consume memory and weaken elasticity, since resources must remain reserved for

functions that may be invoked rarely. Other work optimizes when to snapshot, so that captured state corresponds to a high-performance steady state [9, 48]. Spice benefits from these techniques.

Control-path optimization. Spice focuses on the data path of instance creation: reconstructing memory and process metadata from a snapshot. End-to-end latency also depends on the control path, including request scheduling, placement, and network setup. Dirigent [11], AFaaS [10], SigmaOS [38], and others [26, 37, 41] optimize these orchestration delays with smarter schedulers and resource managers. These efforts are orthogonal to ours. Achieving the lowest possible end-to-end latency requires a holistic approach, combining an efficient control path with the rapid restoration provided by Spice.

7 Discussion

Resource management and keep-alive policies. Spice's restore times in our prototype blur the line between warm and cold starts, potentially altering the economics of serverless resource management. The primary motivation for expensive keep-alive pools (avoiding the high latency of a cold start) is significantly diminished. This enables a new operational model where platforms can practice aggressive reclamation of idle instances to boost utilization, relying on just-in-time instantiation from disk to meet traffic demands without a major latency penalty.

This model also unlocks new strategies for cluster-level optimization. Because Spice leverages the host page cache for sharing, operators can create specialized node pools dedicated to functions with similar software stacks (e.g., a "Python+AI" pool). Co-locating these functions maximizes the page cache hit rate for common runtimes and libraries, which both accelerates restores and increases overall memory density through natural deduplication.

Integration with fork-based approaches. The snapshot/restore model of Spice is complementary to the cloning model of fork-based scaling mechanisms [6, 13, 43], which provide excellent latencies when scaling concurrent execution up from a warm instance. This relationship suggests a new, hybrid architecture for function instantiation. The principles of Spice's page-cache-aware design could be extended across the network to create a distributed page cache. Such a service would maintain a rack- or cluster-wide pool of frequently-accessed memory pages from common libraries and runtimes. During a restore, Spice could then source required pages from this low-latency remote memory fabric in addition to local storage, further reducing instantiation times and improving resource utilization across the cluster.

Limitations and future work. A key next step is extending our approach to virtualized environments. While our current techniques can already improve restore times inside a guest VM by reducing in-guest overheads, further potential can be unlocked with direct host-guest cooperation. This could be achieved with a custom hypercall interface that allows the guest to eagerly request EPT population for a dispersed working set. By enabling the guest to pass its predicted memory layout to the hypervisor in a single, batched operation, this approach would eliminate the thousands of costly VM exits typically caused by individual page faults during memory restoration.

Our prototype assumes that file-backed pages referenced by a SHELF snapshot can be resolved to the same file contents at restore time. We implement this by evaluating functions over a common static container image and recording static path names for file-backed mappings in the SHELF file. This simplifies the prototype, but a production implementation should use a more robust identity for file-backed pages, such as content hashes, image-layer identifiers, or another content-based naming scheme. Such an approach would avoid relying on path stability and would allow the loader to detect mismatches caused by image updates or configuration errors. More generally, SHELF only requires that shared file contents have stable identities across restores; this is compatible with existing approaches [8, 19] that use overlay file systems or image-layer deduplication to share common container contents across functions.

While Spice's `reexec()` mechanism is directly implemented in Linux, we chose to prototype metadata restore in Junction due to the simplicity of capturing C++ data structures in userspace with existing performant serialization libraries. We believe that implementing low-latency metadata restore on top of Linux is feasible but still requires significantly higher engineering effort. We hope that our efforts support the adoption of such a primitive by the kernel community.

8 Conclusion

By demonstrating that cold starts from persistent storage can achieve near-warm latency, this work redefines the fundamental trade-offs in serverless computing. The long-accepted compromise between performance and memory elasticity is not an inherent limitation, but an artifact of operating systems designed for an era before serverless. Our findings suggest that the focus of optimization should shift from userspace heuristics and keep-alive policies to first-class OS support for state restoration.

Spice serves as a blueprint for this new direction. By co-designing the execution engine with the kernel, we unlock a new operational model where functions can be aggressively offloaded to disk to maximize density and efficiency, yet instantiated in milliseconds on demand. This approach opens avenues for future platform architectures built around just-in-time, disk-based instantiation as the default, rather than the exception.

Acknowledgements

We thank our shepherd and anonymous reviewers, and many members of the PDOS group at MIT for their valuable feedback. We also thank CloudLab [14] for providing the testbeds for development and artifact evaluation. This research was supported by CNS-2212099, and by ACE, one of seven centers in JUMP 2.0, a Semiconductor Research Corporation (SRC) program sponsored by DARPA.

A Ablation Study

Table 5 shows the effect of Spice's memory-loading optimizations on the full suite of functions shown in Table 1.

B Artifact Appendix

Abstract

This artifact contains the source code of the Spice prototype, the benchmark harness used in our evaluation, and the source code for systems we compare to in our evaluation.

Scope

The artifact supports validation of the paper's primary performance claims: that Spice substantially reduces serverless cold-start latency relative to CRIU, REAP, and FaaSnap, and approaches the latency of a warm invocation. In particular, the provided experiment scripts reproduce end-to-end latency figure shown in Figure 10.

Contents

The artifact repository is organized as follows:

- `criu/`: a fork of CRIU with added benchmark timestamps and support for restoring an address space using `mmap()` only.
- `faasnap/`: a fork of the FaaSnap artifact, modified to support multiple functions and simplify the invocation path. This component is also used for the REAP experiments.
- `faasnap-kernel/`: the Linux 4.14 guest kernel used in the FaaSnap evaluation.
- `firecracker/`: FaaSnap's modified Firecracker VMM with support for detailed tracing of the guest kernel.
- `functions/`: serverless function workloads and input data, including Java, Node.js, and Python functions.
- `junction/`: the LibOS-based container system used by Spice to generate snapshots and invoke functions from snapshot images.
- `reexec/`: a Linux 6.5.0 kernel module that implements Spice's kernel-space SHELF snapshot loader.
- `shelftools/`: userspace tools for generating, editing, and reading snapshot images.
- `node/`: a modified Node.js runtime with support for freezing the garbage collector.
- `scripts/`: build, benchmark, utility, and plotting scripts used to run the artifact evaluation.

Hosting

The artifact is hosted on GitHub at:

`https://github.com/JunctionOS/spice-ae`

The repository contains several submodules. Evaluators should clone it recursively:

```
git clone https://github.com/JunctionOS/spice-ae
cd spice-ae
git submodule update --init --recursive
```

Requirements

The artifact has been developed and tested on a CloudLab [14] disk image running Ubuntu 24.04 with Linux 6.5.0. Building and running the artifact requires root privileges.

We have tested the artifact on a CloudLab `c6620` node with a Micron 7450 MAX NVMe SSD. The SSD used in the paper is rated for higher throughput/lower latency than the SSD in this CloudLab configuration, so absolute latency numbers may vary. The CloudLab configuration is located at `https://www.cloudlab.us/p/BlinkFaaS/Spice`.

Software and system requirements include:

- Ubuntu 24.04.
- Linux 6.5.0 for the host kernel module used by Spice.
- Root privileges.
- Sufficient disk space for building dependencies and storing snapshot files.
- Approximately 20 GB of disk space for the build, excluding snapshot files.
- A snapshot directory on an SSD-backed mount point, for example `/mnt/unused-nvme1n1/snapshots`.

B.1 Setup

The `README.md` in the GitHub repo contains detailed instructions on setting up and running the artifact.

	Baseline		+ SpliceVMAs		+ Minor		+ Async		Warm
cnn	2.89×	(+101 ms)	1.92×	(+49 ms)	1.38×	(+20 ms)	1.31×	(+17 ms)	53.1 ms
hello	25.83×	(+3 ms)	11.72×	(+1 ms)	7.61×	(+852 μs)	6.34×	(+689 μs)	129.0 μs
html	2.13×	(+13 ms)	1.57×	(+6 ms)	1.18×	(+2 ms)	1.16×	(+2 ms)	11.2 ms
img	1.99×	(+15 ms)	1.49×	(+8 ms)	1.30×	(+5 ms)	1.24×	(+4 ms)	15.4 ms
img (Java)	1.10×	(+36 ms)	1.04×	(+14 ms)	1.06×	(+22 ms)	1.05×	(+17 ms)	373.6 ms
img (Node)	1.84×	(+38 ms)	1.42×	(+19 ms)	1.24×	(+11 ms)	1.21×	(+9 ms)	45.5 ms
json	3.02×	(+15 ms)	1.86×	(+6 ms)	1.38×	(+3 ms)	1.28×	(+2 ms)	7.4 ms
json (Node)	2.87×	(+17 ms)	2.00×	(+9 ms)	1.74×	(+7 ms)	1.47×	(+4 ms)	8.9 ms
lr	22.97×	(+27 ms)	8.77×	(+10 ms)	4.98×	(+5 ms)	4.73×	(+5 ms)	1.2 ms
mtml (Java)	1.98×	(+6 ms)	1.31×	(+2 ms)	1.15×	(+920 μs)	1.15×	(+912 μs)	6.1 ms
pyaes	3.11×	(+4 ms)	1.94×	(+2 ms)	1.47×	(+803 μs)	1.38×	(+659 μs)	1.7 ms
rnn	3.47×	(+21 ms)	2.03×	(+9 ms)	1.52×	(+4 ms)	1.33×	(+3 ms)	8.6 ms
video	1.13×	(+28 ms)	1.05×	(+11 ms)	1.02×	(+5 ms)	1.01×	(+2 ms)	222.1 ms

Table 5: Results from ablation study in §5.2 for all functions, showing the importance of the loader's design elements. Values in each column reflect slowdowns relative to warm.

B.2 Expected Results

The main expected result is an end-to-end latency plot showing that Spice significantly reduces cold-start latency compared with FaaSnap, REAP, and CRIU, while approaching warm invocation performance. The exact numbers may vary with SSD performance.

References

[1] Aws lambda. https://aws.amazon.com/pm/lambda/. Accessed: 2025-08-20.

[2] Azure functions. https://azure.microsoft.com/en-us/products/functions. Accessed: 2025-08-20.

[3] CRIU: Checkpoint/Restore In Userspace. https://criu.org/Main_Page.

[4] Alexandru Agache, Marc Brooker, Alexandra Iordache, Anthony Liguori, Rolf Neugebauer, Phil Piwonka, and Diana-Maria Popa. Firecracker: Lightweight virtualization for serverless applications. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 419–434, Santa Clara, CA, February 2020. USENIX Association.

[5] Siddharth Agarwal, Maria A. Rodriguez, and Rajkumar Buyya. A reinforcement learning approach to reduce serverless function cold start frequency. In *2021 IEEE/ACM 21st International Symposium on Cluster, Cloud and Internet Computing (CCGrid)*, pages 797–803, 2021.

[6] Chloe Alverti, Stratos Psomadakis, Burak Ocalan, Shashwat Jaiswal, Tianyin Xu, and Josep Torrellas. Cxl-fork: Fast remote fork over cxl fabrics. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, ASPLOS '25, page 210–226, New York, NY, USA, 2025. Association for Computing Machinery.

[7] Lixiang Ao, George Porter, and Geoffrey M. Voelker. Faasnap: Faas made fast using snapshot-based vms. In *Proceedings of the Seventeenth European Conference on Computer Systems*, EuroSys '22, page 730–746, New York, NY, USA, 2022. Association for Computing Machinery.

[8] Marc Brooker, Mike Danilov, Chris Greenwood, and Phil Piwonka. On-demand container loading in AWS lambda. In *2023 USENIX Annual Technical Conference (USENIX ATC 23)*, pages 315–328, Boston, MA, July 2023. USENIX Association.

[9] James Cadden, Thomas Unger, Yara Awad, Han Dong, Orran Krieger, and Jonathan Appavoo. SEUSS: skip redundant paths to make serverless fast. In *Proceedings of the Fifteenth European Conference on Computer Systems*, EuroSys '20, New York, NY, USA, 2020. Association for Computing Machinery.

[10] Xiaohu Chai, Tianyu Zhou, Keyang Hu, Jianfeng Tan, Tiwei Bie, Anqi Shen, Dawei Shen, Qi Xing, Shun Song, Tongkai Yang, et al. Fork in the road: Reflections and optimizations for cold start latency in production serverless systems. In *19th USENIX Symposium on Operating Systems Design and Implementation (OSDI 25)*, pages 199–218, 2025.

[11] Lazar Cvetković, François Costa, Mihajlo Djokic, Michal Friedman, and Ana Klimovic. Dirigent: Lightweight serverless orchestration. In *Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles*, SOSP '24, page 369–384, New York, NY, USA, 2024. Association for Computing Machinery.

[12] Nilanjan Daw, Umesh Bellur, and Purushottam Kulkarni. Xanadu: Mitigating cascading cold starts in serverless function chain deployments. In *Proceedings of the 21st International Middleware Conference*, Middleware '20, page 356–370, New York, NY, USA, 2020. Association for Computing Machinery.

[13] Dong Du, Tianyi Yu, Yubin Xia, Binyu Zang, Guanglu Yan, Chenggang Qin, Qixuan Wu, and Haibo Chen. Catalyzer: Sub-millisecond startup for serverless computing with initialization-less booting. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '20, page 467–481, New York, NY, USA, 2020. Association for Computing Machinery.

[14] Dmitry Duplyakin, Robert Ricci, Aleksander Maricq, Gary Wong, Jonathon Duerig, Eric Eide, Leigh Stoller, Mike Hibler, David Johnson, Kirk Webb, Aditya Akella, Kuangching Wang, Glenn Ricart, Larry Landweber, Chip Elliott, Michael Zink, Emmanuel Cecchet, Snigdhaswin Kar, and Prabodh Mishra. The design and operation of CloudLab. In *Proceedings of the USENIX Annual Technical Conference (ATC)*, pages 1–14, July 2019.

[15] Joshua Fried, Gohar Irfan Chaudhry, Enrique Saurez, Esha Choukse, Inigo Goiri, Sameh Elnikety, Rodrigo Fonseca, and Adam Belay. Making Kernel Bypass Practical for the Cloud with Junction. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*, Santa Clara, CA, April 2024. USENIX Association.

[16] Alexander Fuerst and Prateek Sharma. Faascache: keeping serverless computing alive with greedy-dual caching. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '21, page 386–400, New York, NY, USA, 2021. Association for Computing Machinery.

[17] Joseph M. Hellerstein, Jose M. Faleiro, Joseph Gonzalez, Johann Schleier-Smith, Vikram Sreekanti, Alexey Tumanov, and Chenggang Wu. Serverless computing: One step forward, two steps back. In *9th Biennial Conference on Innovative Data Systems Research, CIDR 2019, Asilomar, CA, USA, January 13-16, 2019, Online Proceedings*. www.cidrdb.org, 2019.

[18] Liam R. Howlett. Maple tree. https://docs.kernel.org/core-api/maple_tree.html. Accessed: 2025-12-11.

[19] Jialiang Huang, MingXing Zhang, Teng Ma, Zheng Liu, Sixing Lin, Kang Chen, Jinlei Jiang, Xia Liao, Yingdi Shan, Ning Zhang, Mengting Lu, Tao Ma, Haifeng Gong, and YongWei Wu. Trenv: Transparently share serverless execution environments across different functions and nodes. In *Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles*, SOSP '24, page 421–437, New York, NY, USA, 2024. Association for Computing Machinery.

[20] Abhinav Jangda, Bobby Powers, Emery D. Berger, and Arjun Guha. Not so fast: Analyzing the performance of WebAssembly vs. native code. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 107–120, Renton, WA, July 2019. USENIX Association.

[21] Jeongchul Kim and Kyungyong Lee. Functionbench: A suite of workloads for serverless cloud function service. In *2019 IEEE 12th International Conference on Cloud Computing (CLOUD)*, pages 502–504, July 2019.

[22] Sumer Kohli, Shreyas Kharbanda, Rodrigo Bruno, Joao Carreira, and Pedro Fonseca. Pronghorn: Effective checkpoint orchestration for serverless hot-starts. In *Proceedings of the Nineteenth European Conference on Computer Systems*, EuroSys '24, page 298–316, New York, NY, USA, 2024. Association for Computing Machinery.

[23] Nikita Lazarev, Varun Gohil, James Tsai, Andy Anderson, Bhushan Chitlur, Zhiru Zhang, and Christina Delimitrou. Sabre: Hardware-Accelerated snapshot compression for serverless MicroVMs. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, pages 1–18, Santa Clara, CA, July 2024. USENIX Association.

[24] Zijun Li, Jiagan Cheng, Quan Chen, Eryu Guan, Zizheng Bian, Yi Tao, Bin Zha, Qiang Wang, Weidong Han, and Minyi Guo. RunD: A lightweight secure container runtime for high-density deployment and high-concurrency startup in serverless computing. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, pages 53–68, Carlsbad, CA, July 2022. USENIX Association.

[25] Zijun Li, Linsong Guo, Quan Chen, Jiagan Cheng, Chuhao Xu, Deze Zeng, Zhuo Song, Tao Ma, Yong Yang, Chao Li, and Minyi Guo. Help rather than recycle: Alleviating cold startup in serverless computing through Inter-Function container sharing. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, pages 69–84, Carlsbad, CA, July 2022. USENIX Association.

[26] Zhen Lin, Kao-Feng Hsieh, Yu Sun, Seunghee Shin, and Hui Lu. Flashcube: Fast provisioning of serverless functions with streamlined container runtimes. In *Proceedings of the 11th Workshop on Programming Languages and Operating Systems*, PLOS '21, page 38–45, New York, NY, USA, 2021. Association for Computing Machinery.

[27] David Lion, Adrian Chiu, Hailong Sun, Xin Zhuang, Nikola Grcevski, and Ding Yuan. Don't get caught in the cold, warm-up your JVM: Understand and eliminate JVM warm-up overhead in Data-Parallel systems. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 383–400, Savannah, GA, November 2016. USENIX Association.

[28] Wes Lloyd, Minh Vu, Baojia Zhang, Olaf David, and George Leavesley. Improving application migration to serverless computing platforms: Latency mitigation with keep-alive workloads. In *2018 IEEE/ACM International Conference on Utility and Cloud Computing Companion (UCC Companion)*, pages 195–200, 2018.

[29] Ashraf Mahgoub, Edgardo Barsallo Yi, Karthick Shankar, Sameh Elnikety, Somali Chaterji, and Saurabh Bagchi. ORION and the three rights: Sizing, bundling, and prewarming for serverless DAGs. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 303–320, Carlsbad, CA, July 2022. USENIX Association.

[30] Rohan Basu Roy, Tirthak Patel, and Devesh Tiwari. Icebreaker: warming serverless functions better with heterogeneity. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '22, page 753–767, New York, NY, USA, 2022. Association for Computing Machinery.

[31] Alireza Sahraei, Soteris Demetriou, Amirali Sobhgol, Haoran Zhang, Abhigna Nagaraja, Neeraj Pathak, Girish Joshi, Carla Souza, Bo Huang, Wyatt Cook, Andrii Golovei, Pradeep Venkat, Andrew Mcfague, Dimitrios Skarlatos, Vipul Patel, Ravinder Thind, Ernesto Gonzalez, Yun Jin, and Chunqiang Tang. Xfaas: Hyperscale and low cost serverless functions at meta. In *Proceedings of the 29th Symposium on Operating Systems Principles*, SOSP '23, page 231–246, New York, NY, USA, 2023. Association for Computing Machinery.

[32] Divyanshu Saxena, Tao Ji, Arjun Singhvi, Junaid Khalid, and Aditya Akella. Memory deduplication for serverless computing with medes. In *Proceedings of the Seventeenth European Conference on Computer Systems*, EuroSys '22, page 714–729, New York, NY, USA, 2022. Association for Computing Machinery.

[33] Mohammad Shahrad, Rodrigo Fonseca, Inigo Goiri, Gohar Chaudhry, Paul Batum, Jason Cooke, Eduardo Laureano, Colby Tresness, Mark Russinovich, and Ricardo Bianchini. Serverless in the wild: Characterizing and optimizing the serverless workload at a large cloud provider. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pages 205–218. USENIX Association, July 2020.

[34] Simon Shillaker and Peter Pietzuch. Faasm: lightweight isolation for efficient stateful serverless computing. In *Proceedings of the 2020 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC'20, USA, 2020. USENIX Association.

[35] Wonseok Shin, Wook-Hee Kim, and Changwoo Min. Fireworks: a fast, efficient, and safe serverless framework using VM-level post-JIT snapshot. In *Proceedings of the Seventeenth European Conference on Computer Systems*, EuroSys '22, page 663–677, New York, NY, USA, 2022. Association for Computing Machinery.

[36] Paulo Silva, Daniel Fireman, and Thiago Emmanuel Pereira. Prebaking functions to warm the serverless cold start. In *Proceedings of the 21st International Middleware Conference*, Middleware '20, page 1–13, New York, NY, USA, 2020. Association for Computing Machinery.

[37] Arjun Singhvi, Arjun Balasubramanian, Kevin Houck, Mohammed Danish Shaikh, Shivaram Venkataraman, and Aditya Akella. Atoll: A scalable low-latency serverless platform. In *Proceedings of the ACM Symposium on Cloud Computing*, SoCC '21, page 138–152, New York, NY, USA, 2021. Association for Computing Machinery.

[38] Ariel Szekely, Adam Belay, Robert Morris, and M. Frans Kaashoek. Unifying serverless and microservice workloads with sigmaos. In *Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles*, SOSP '24, page 385–402, New York, NY, USA, 2024. Association for Computing Machinery.

[39] Dmitrii Ustiugov, Plamen Petrov, Marios Kogias, Edouard Bugnion, and Boris Grot. Benchmarking, analysis, and optimization of serverless function snapshots. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '21, page 559–572,

New York, NY, USA, 2021. Association for Computing Machinery.

[40] Ranjan Sarpangala Venkatesh, Till Smejkal, Dejan S. Milojicic, and Ada Gavrilovska. Fast in-memory criu for docker containers. In *Proceedings of the International Symposium on Memory Systems*, MEMSYS '19, page 53–65, New York, NY, USA, 2019. Association for Computing Machinery.

[41] Ao Wang, Shuai Chang, Huangshi Tian, Hongqi Wang, Haoran Yang, Huiba Li, Rui Du, and Yue Cheng. FaaSNet: Scalable and fast provisioning of custom serverless container runtimes at alibaba cloud function compute. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, pages 443–457. USENIX Association, July 2021.

[42] Kai-Ting Amy Wang, Rayson Ho, and Peng Wu. Replayable execution optimized for page sharing for a managed runtime environment. In *Proceedings of the Fourteenth EuroSys Conference 2019*, EuroSys '19, New York, NY, USA, 2019. Association for Computing Machinery.

[43] Xingda Wei, Fangming Lu, Tianxia Wang, Jinyu Gu, Yuhan Yang, Rong Chen, and Haibo Chen. No provisioned concurrency: Fast RDMA-codesigned remote fork for serverless computing. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*, pages 497–517, Boston, MA, July 2023. USENIX Association.

[44] Jianing You, Kang Chen, Laiping Zhao, Yiming Li, Yichi Chen, Yuxuan Du, Yanjie Wang, Luhang Wen, Keyang Hu, and Keqiu Li. Alloystack: A library operating system for serverless workflow applications. In *Proceedings of the Twentieth European Conference on Computer Systems*, pages 921–937, 2025.

[45] Hanfei Yu, Rohan Basu Roy, Christian Fontenot, Devesh Tiwari, Jian Li, Hong Zhang, Hao Wang, and Seung-Jong Park. Rainbowcake: Mitigating cold-starts in serverless with layer-wise container caching and sharing. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1*, ASPLOS '24, page 335–350, New York, NY, USA, 2024. Association for Computing Machinery.

[46] Irene Zhang, Alex Garthwaite, Yury Baskakov, and Kenneth C. Barr. Fast restore of checkpointed memory using working set estimation. In *Proceedings of the 7th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE '11, page 87–98, New York, NY, USA, 2011. Association for Computing Machinery.

[47] Xinmin Zhang, Qiang He, Hao Fan, and Song Wu. Faascale: Scaling microvm vertically for serverless computing with memory elasticity. In *Proceedings of the 2024 ACM Symposium on Cloud Computing*, SoCC '24, page 196–212, New York, NY, USA, 2024. Association for Computing Machinery.

[48] Yifei Zhang, Tianxiao Gu, Xiaolin Zheng, Lei Yu, Wei Kuai, and Sanhong Li. Towards a serverless java runtime. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 1156–1160, 2021.

ARTIFACT EVALUATED
USENIX
AVAILABLE

Distributed Speculative Execution for Resilient Cloud Applications

Tianyu Li
MIT CSAIL

Badrish Chandramouli
Microsoft Research

Philip A. Bernstein
Microsoft Research

Samuel Madden
MIT CSAIL

Abstract

Fault-tolerance is critically important in highly distributed modern cloud applications. Solutions such as Temporal, Azure Durable Functions, and Beldi hide fault-tolerance complexity from developers by automatically persisting execution state and resuming seamlessly after failure. This pattern, often called *durable execution*, usually forces frequent and synchronous persistence, resulting in significant latency overheads. In this paper, we propose *distributed speculative execution* (DSE), a technique for implementing the durable execution abstraction without incurring this penalty. With DSE, developers write code assuming synchronous persistence, and a DSE *runtime* is responsible for transparently eliding persistence and reactively repairing application state on failure. We present libDSE, the first DSE application framework that achieves this vision. To hide speculation from application code, we design a novel programming model centered around message-passing, atomic code blocks, and lightweight threads, and show that it allows developers to build a variety of speculative services, including write-ahead logs, key-value stores, event brokers, and fault-tolerant workflows. Our evaluation shows that libDSE reduces end-to-end latency by up to an order of magnitude for persistence-bound applications compared to current durable execution systems with minimal runtime overhead and complexity.

1 Introduction

Modern cloud applications are more distributed than ever. Companies have long adopted microservice-oriented architectures, splitting their applications into hundreds of loosely coupled, independently deployed distributed components [2, 43]. Recent proposals, such as serverless and granular computing [49, 55], have pushed for even more distribution at a finer granularity. Benefits aside, this makes every modern cloud application inherently distributed and fault tolerance a key challenge, leading to a new class of *Durable Execution* [10] systems (e.g., Azure Durable Functions [8], Temporal [30]). These systems automatically persist application state where necessary and transparently recover from failures to resume execution. Today, they are widely used to orchestrate microservices or serverless workers [12–14, 16, 17, 20, 22, 27, 32, 48, 76, 78]. Despite growing popularity, however, durable execution engines often must perform frequent and synchronous state persistence for correctness, which results in high latency overhead (Section 2). Worse, this *persistence amplification* scales with the degree of distribution — as the application is split into more distributed components, synchronous persistence becomes more frequent and expensive.

In this paper, we propose *distributed speculative execution* (DSE), which decouples the abstraction of durable execution from its physical execution pattern. DSE bypasses synchronous persistence on the failure-free path, allowing services to communicate speculatively. To ensure correctness, the runtime acts as a speculation sandbox, buffering outputs to external systems (e.g., the user, legacy databases) until the underlying state is durable. This dramatically reduces perceived latency on the common path, in exchange for more complex failure recovery. This trade-off is worthwhile as long as the unit of speculation (e.g., an RPC request) is more likely to succeed than to be interrupted by a failure.

Speculation is not a new idea. Prior work in hardware [70], databases [39, 44], and file systems [35, 62, 63] used similar techniques to hide latency. However, applying speculation to modern cloud applications introduces new challenges.

Many previous systems assume a canonical shared state (e.g., main memory, distributed file system). Participants locally speculate on the outcome of operations on that shared state without committing speculative state. In contrast, many modern cloud applications adopt a shared-nothing, message-passing architecture (e.g., microservices, event-driven architecture). This requires speculation across multiple independent, failure-prone entities (i.e., on both sides of a message channel). Consequently, DSE must correctly orchestrate distributed rollbacks across multiple participants when failures occur. Additionally, such rollbacks can interleave and race with client logic unless the client rigorously protects itself by

synchronizing against possible rollbacks at any time, resulting in significant conceptual and engineering complexity.

To solve these challenges, we build libDSE, a speculative programming framework for message-passing applications. For the first challenge, orchestrating distributed rollbacks, libDSE adapts the *Distributed Prefix Recovery* (DPR) protocol [57], a distributed cache recovery scheme, to our message-passing setting. LibDSE automatically instruments messages (e.g., RPC calls) between services to build and maintain a recovery dependency graph. The graph is used to determine when results are safe to expose externally, and to compute a consistent snapshot for every affected component to roll back to after failure. Component services are required to expose hooks, enabling the libDSE runtime to force persistence/rollback when necessary and prevent unbounded growth of the dependency graph.

For the second challenge, synchronizing against possible rollbacks, libDSE provides a speculation-native programming model centered around two intuitive primitives, *actions* and *sthreads*. Actions are user-defined atomic blocks that are guaranteed by libDSE not to overlap with persist/recovery/rollback invocations. Sthreads are stateless and lightweight threads of execution that allow libDSE to guarantee correctness for long-running or asynchronous operations that cannot execute atomically. Advanced developers exercise fine-grained control over speculation behavior through *speculation barriers*, analogous to memory barriers [26], which allow libDSE services to co-exist with non-speculative components by only exposing non-speculative results.

Using libDSE, developers can build speculative message-passing services while relying on the framework to manage the complexities of distributed dependency tracking and concurrent rollback/recovery. We demonstrate the benefits and trade-offs of DSE by building three representative applications: a travel reservation workflow, a high-throughput event processor, and a transaction processing system. We evaluate our libDSE-based implementation against state-of-the-art production and research systems, including Temporal [30], DARQ [56], and Microsoft Orleans [29], showing that it can reduce end-to-end latency by up to an order of magnitude. We also quantify the engineering effort of adopting libDSE and discuss the practical implications of its programming model. In summary, our contributions are:

- We propose a new distributed speculative execution (DSE) scheme that dramatically reduces fault-tolerance overhead in durably executed applications.
- We present the design and implementation of libDSE, the first DSE programming framework for modern shared-nothing message-passing cloud applications.
- We present a detailed evaluation of libDSE on a diverse set of cloud applications, showing up to an order of magnitude latency savings compared to current, non-speculative implementations.

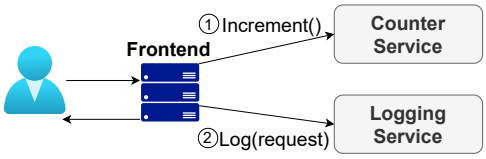


Figure 1: A Simple Running Example

2 Background and Motivation

Running Example. Consider a toy example of an application that consists of a frontend layer, a counter service, and a logging service. Users issue requests to the frontend to increment the counter. Each request and its result are then sent to the logging service for bookkeeping purposes. Failures can result in a number of anomalies even in this simple scenario. Each service can crash and lose data. The frontend server can crash after a successful increment but fail to log the incremented value. Naive attempts to solve this may retry the operation, and cause the services to increment or log too many times. While these anomalies appear innocuous in the toy example, they are highly undesirable in real-world scenarios. If an e-commerce company uses a similar architecture to connect their payment and fulfillment services, such anomalies can lead to double charging or unfulfilled orders. Developers often resort to principled solutions like durable execution to prevent such anomalies.

2.1 Current Solution

Durable execution creates an illusion of uninterrupted, failure-free execution by persisting application state at every step and automatically retrying execution after failure. The goal of durable execution is to ensure that users cannot distinguish between executions of the application with failures, and ones without (except perhaps through performance degradation) [4]. Today's cloud architect might achieve durable execution in our running example by first making each service durable (e.g., by persisting to a database before returning), and then using a resilient workflow engine (e.g., Temporal) to orchestrate the frontend calls.

To protect against frontend failure, the workflow engine automatically persists operation intents [69] and intermediate results, and retries on failure. Each stateful service must be programmed for *idempotency* – retried requests must have no visible effects beyond their first execution, e.g., by attaching a unique ID to each request and de-duplicating on the service side. In our running example, the workflow engine would execute an increment call by first persisting the intent to increment and generate a unique request ID, retry until successful, and persist the result. Under the hood, the workflow system maintains (sometimes implicitly) a directed acyclic graph

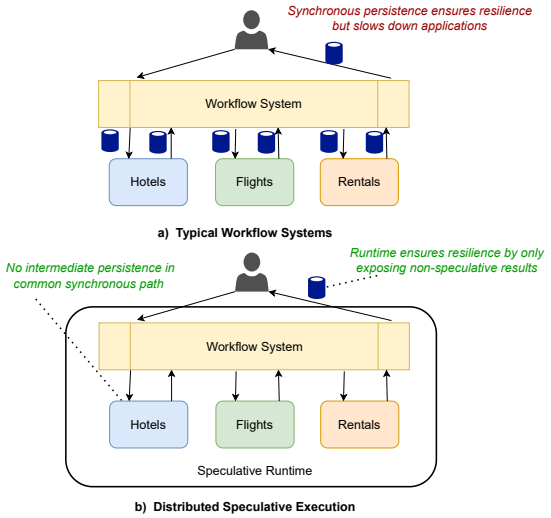


Figure 2: DSE optimizes durable execution by transparently eliding synchronous persistence and waiting in parallel

(DAG) of stateful tasks, and ensures at-least-once execution (which is equivalent to exactly-once if tasks are idempotent).

Most such systems today must synchronously persist each task's output before starting the next task. Otherwise, if tasks are non-deterministic, a replay may yield different outputs and spawn different downstream tasks. The divergent tasks may then conflict and cause anomalies. Consequently, almost all of today's durable execution systems have cumulative latency overheads that scale with the depth of the task graph.

Our Solution. In contrast, DSE forges ahead before operation intents, effects, or intermediate results are durable, greatly reducing latency. This inevitably leads to anomalies upon failures. To maintain the same durable execution guarantees, DSE applications must detect such inconsistencies, repair them, and prevent external users from observing them. In other words, the requirements of correct DSE are:

- *Recoverability.* Inconsistencies are eventually repaired.
- *Failure Transparency.* External entities (e.g., clients) never observe failure-induced inconsistencies.

Fundamentally, this means that DSE systems are rollback-based recovery systems [42]. Any solution must design protocols that correctly track recovery dependencies between participants, determine when state becomes recoverable, and roll back unrecoverable state on failure. On failure, the system determines the extent of failure based on collected dependencies and orchestrates rollback across participants to restore the application to a consistent state. Such schemes support non-determinism, as non-deterministic operations are either restored from persisted state and never replayed, or all their effects are rolled back. The key trade-off is that on failure, DSE systems lose speculative work and must coordinate rollback, which adds complexity.

2.2 System Model and Assumptions

Before we proceed, we discuss the system model and assumptions made for DSE that set it apart from earlier work.

Failure Model. Each component has *fail-restart* semantics [56], meaning it may fail at any point and lose its volatile in-memory state. However, it is guaranteed to restart in bounded time and recover from state that it has explicitly persisted before failure. Multiple incarnations of the same component may temporarily co-exist due to network partitions. We do not consider Byzantine failures.

Network. We assume reliable point-to-point communication channels (e.g., TCP/IP), meaning messages are not lost or corrupted. Network partitions can occur, temporarily preventing services from communicating.

Storage and Persistence. Each component has access to its own private durable storage that persists across failures, such as a local log, remote replicas, or a backend database. However, to enforce loose coupling between components, components cannot directly access each other's storage.

2.3 Targeted Workloads and Boundaries

DSE is broadly applicable to applications dominated by serial persistence overheads. Beyond microservice orchestration, DSE naturally targets:

Event-driven Processing. Many modern cloud systems use event-driven pipelines to process streams of data. While the initial ingestion may be durably logged upfront for reliability, the state of each processing operator in the pipeline must also be fault-tolerant. In conventional systems like Kafka Streams [73], this requires each operator to synchronously persist its intermediate state before passing results to the next stage, incurring a persistence penalty at each step. DSE can optimize this by allowing intermediate states to be managed speculatively, eliminating synchronous I/O between stages.

Distributed Transactions. Many applications require atomic updates across multiple microservices (e.g., orchestrating checkout between inventory, payment, and shipping). This is often achieved using distributed transaction protocols (e.g., two-phase commit [54, 60]), which rely on frequent, synchronous logging to a durable medium. DSE can transparently hide the latency of these synchronous log writes from the critical path without requiring changes to the core protocol logic.

Conversely, DSE's benefits diminish when applied to workloads where persistence is not the primary bottleneck. For *compute-heavy applications*, such as agentic exploration or a video transcoding pipeline, task execution time vastly overshadows persistence latency. In these scenarios, the latency saved by DSE can be negligible to the end-to-end execution time, as compute cost dominates. Similarly, for *shallow, CRUD-heavy applications*, the latency savings are proportionally smaller due to shorter execution chains. However, even for these simple applications, traditional durable execution

engines typically persist execution history (e.g., intent to execute, task start, and task completion) to guarantee deterministic replay. This turns single logical operations into multiple synchronous persistence operations. Consequently, DSE still provides measurable improvements for shallow workloads by bypassing these framework-injected persistence roundtrips, though the absolute savings are less dramatic than in deep workflows.

3 libDSE Design

DSE application developers build speculative services by encapsulating application state into `StateObject`s. Application developers define `StateObject`s by implementing a set of methods to persist or restore application state and using libDSE's primitives to protect state access. It is the responsibility of libDSE to manage the complexity of speculation and failure recovery; it 1) persists, recovers, or rolls back `StateObject`s when necessary by invoking developer-supplied methods, 2) instruments application messages to establish dependencies, drop rolled back messages, or delays speculative messages where necessary for safety, and 3) implements concurrency control primitives to protect application logic against rollbacks. Application developers do not need to implement the distributed recovery protocol logic or reason about speculation beyond specifying atomicity boundaries using libDSE. The rest of this section focuses on these primitives (Table 1). We discuss the details of how libDSE orchestrates speculative execution in Section 4.

3.1 StateObjects and Actions

Users create `StateObject`s by implementing the abstract `StateObject` API shown in Table 1. We show an example implementation of the counter service from our running example using cloud storage files in Figure 3. The framework ensures that `Persist` and `Restore` are not concurrently invoked. For performance, the `Persist` API is asynchronous; users may return from the call once the persistence operation is issued (e.g., flush the log at a specific offset), but before it is completed. While conceptually simple, `Restore` encapsulates two different use cases, one where the `StateObject` has failed and must reload state from persistent storage, and another where the `StateObject` responds to a rollback. For both cases, the example loads persistent state. This is correct for the latter, but potentially slow and inefficient. Developers are therefore encouraged to distinguish between these two cases (e.g., by checking if `value == default` in the example) and apply application-specific optimizations (e.g., by leveraging built-in multi-versioning [34, 65, 67]). If the `StateObject` crashes, the runtime uses `ListVersions` to identify unpruned `Persist` calls. This API easily allows for alternative implementations of persistence and recovery. For

```
1  class CounterStateObject : StateObject {
2    int value;
3
4    void Persist(long ver, byte[] m, Action c) {
5      var content = Marshall(ver, m);
6      // async write to cloud storage
7      Task.Run(() => {
8        WriteFile(GetName(v), content);
9        c();
10     });
11   }
12
13   byte[] Restore(long ver) {
14     content = ReadFile(GetName(ver));
15     var v, metadata = Unmarshall(content);
16     this.value = v;
17     return metadata;
18   }
19   ...
20 }
```

Figure 3: Example `StateObject` Implementation

```
1  class CounterService {
2    CounterStateObject so;
3
4    // Handler for RPC call to increment counter
5    IncrementResponse Increment(
6        IncrementRequest r) {
7      if (!so.StartAction(r.header)) throw;
8      response = new IncrementResponse();
9      response.result =
10         AtomicInc(so.value, r.incrementBy);
11     response.header = so.EndAction();
12     return response;
13   }
14   ...
15 }
```

Figure 4: Example CounterService Implementation

example, one can implement the counter service via either logging and replay or a replicated state machine.

libDSE organizes operations on `StateObject`s as a series of atomic execution units called actions. Each action's effects are either entirely persisted or entirely lost due to recovery or rollback. libDSE achieves this by ensuring actions never interleave with `Persist` or `Restore` operations. However, actions may execute concurrently with each other to allow for parallelism. Messages are consumed as part of actions or produced as a result of actions. libDSE provides users with flexibility on how best to deliver messages between `StateObject`s, relying on users to pass opaque headers rather than mandating a specific messaging implementation.

We present an example in Figure 4 that implements our counter service using gRPC [21]. Each message is instrumented with a `.header` field that holds metadata for libDSE (more on this later). If `StartAction` returns false, the message's sender was rolled back, and the message must be discarded. Otherwise, libDSE protects execution after line 7

StateObject API	**Semantics**
`StartAction(header) → bool`	Start an atomic action, (optionally) receive the given libDSE message header. If returned false, the message originates from rolled back state and should be discarded.
`EndAction() → header`	End an atomic action and (optionally) produce a header that encodes dependencies for any outgoing messages resulting from this action.
`Connect(config)`	Connect this `StateObject` to a libDSE Coordinator using the given configuration. Must be invoked exactly once before performing any other operation.
`Refresh()`	Refresh should be periodically called to perform libDSE background tasks and guarantee progress for the cluster.
`Detach() → sthread`	Temporarily suspend an action and produce a sthread that can be used for speculative operations independent of this `StateObject`.
`Merge(sthread) → bool`	Resume an action by merging the detached sthread back into this `StateObject` and update its dependencies. If false, the state leading to the creation of the sthread was rolled back and the sthread should be discarded.
sthread API	**Semantics**
`Receive(header) → bool`	Attempt to receive the given libDSE message header. If returned false, the message should be discarded. Throws exception if the sthread is rolled back.
`Send() → header`	Produce a header that encodes current dependencies of this sthread.
`Barrier()`	Blocks until all state observed by this sthread becomes non-speculative. Throws exception if the sthread is rolled back.
StateObject Callbacks	**Semantics**
`Persist(v, metadata, onPersist)`	Persist the current state and `metadata` as version `v` and then invoke `onPersist`.
`Restore(v) → metadata`	Recover (or roll back) to the given version and return associated metadata.
`Prune(v)`	Signal that a version (and all that precede it) can be pruned.
`ListVersions() → metadata[]`	List all unpruned versions and associated metadata.

Table 1: Summary of libDSE API

from libDSE-triggered persists and restores. Users still need to guard against multiple concurrent requests, and hence use the thread-safe version of increment in line 10. Finally, it ends the action, obtaining a header to pass back to the caller.

3.2 Handling Asynchrony

Long-running actions can block persistence until finished and halt persistence progress across the cluster. For example, the frontend server in our running example issues RPCs that may complete asynchronously and automatically retry in the background. In some settings, long running actions may even block each other and cause a deadlock. libDSE introduces *sthreads* to prevent this from happening. A sthread is essentially a lightweight thread of execution within a `StateObject` that encodes the speculative state of its parent `StateObject` at the time of sthread creation.

We show an example in Figure 5. Here, the frontend workflow orchestrator calls `Detach` after marking the start of workflow (line 7), but before executing the step (i.e., calling the service) (line 10). After line 8, application code no longer runs in an atomic block, and may go to sleep, retry, or otherwise perform long-running operations without blocking the `StateObject`. Thus, in the event of a rollback, the sthread may still temporarily continue to execute. It is therefore essential that sthreads are treated as *independent, standalone* participants of the system and interact with other participants, including their parent `StateObject`s, exclusively through libDSE-instrumented message passing. For example, on line

```
1  ...
2  WorkflowEngineStateObject so;
3  ...
4  // Handler for RPC call to execute workflow
5  async Task ExecWorkflow(WorkflowRequest r) {
6    if (!so.StartAction(r.header)) throw;
7    so.MarkWorkflowStart(r);
8    var  t = so.Detach();
9    // Request executes outside atomic block
10   cReq = ComposeRequest(r, t.Send());
11   var cRes =
12       await CallCounterServiceAsync(cReq);
13   if (!so.Merge(t)) throw;
14   // Back in an atomic block
15   so.UpdateWorkflow(cRes);
16   ...
17   // Finally, return after barrier to
18   // hide speculation from caller
19   await so.Detach().Barrier();
20   return response;
21 }
```

Figure 5: Example Workflow Implementation with libDSE

12, we obtain the result of an asynchronous action, but cannot yet consume its result; this must happen after the sthread is merged to start a new action (line 13), which logically sends a message from sthread that `StateObject` receives. If the system experiences a rollback while the asynchronous action runs, `StateObject` rejects the message and the sthread can simply terminate. The sthread represents state derived from its parent, which will recover independently and respawn the sthread if necessary.

Finally, sthreads support barriers, which provide fine-grained control over speculative behavior. Similar to a memory fence, a barrier blocks until everything the sthread received becomes non-speculative, thus preventing speculation across the barrier. Only sthreads are allowed to invoke barriers, as they are, by definition, blocking. Barriers are useful when interacting with external entities. For example, in line 19 of Figure 5, by detaching and calling barrier, execution of line 20 will be delayed until the response becomes non-speculative, thereby only sending non-speculative results to users. In many cases, barriers can be automatically inserted (e.g., at the end of an RPC handler processing an external user request), but advanced users may also use barriers to prevent speculative dependencies between parts of their applications. For example, inserting a barrier before line 7 prevents the system from logging workflow starts speculatively.

4 Speculative Protocol

4.1 Inspiration: DPR

Distributed Prefix Recovery (DPR) [57] is a recently proposed technique for causal consistency across sharded cache-stores – storage units spanning volatile memory and durable storage. For example, a shard within a partitioned database with an accompanying Redis caching layer can be considered a (logical) cache-store. Often in such an architecture, the caches are *write-through*: clients read from the cache, but write directly to the backend storage. In contrast, DPR is *write-back*: writes directly update the cache for increased throughput and immediate visibility, and cached entries are asynchronously flushed to storage. This leads to lost writes if caches fail, which may cause application anomalies as readers may have acted on lost updates. DPR presents a lightweight protocol for addressing this problem. First, operation completion and persistence are decoupled using two acknowledgements. Second, DPR clients interact with cache-stores explicitly through *sessions*. Each session is a (linearizable) sequence of operations and DPR guarantees that any surviving state corresponds to session *prefixes*. This ensures that no surviving operation depends on a lost operation when a cache node fails.

Our Insights. We identify two key mismatches between DPR and modern microservices. First, DPR has a rigid client-server formulation. DPR clients are first-class citizens but do not participate in rollbacks; they are responsible for discovering rollbacks and resuming (potentially replaying) operations. This complexity is compounded by possible client failures, as a recovered (amnesiac) client may lack the information needed to handle a rollback. libDSE eliminates this distinction by treating every component as a stateful, message-passing peer, correctly modeling that application control flow (e.g., in a workflow orchestrator) is itself state that must participate in rollbacks. Second, the original DPR relies on a stateful coordinator backed by an external database. Consequently, DPR requires an additional synchronous persist on the failure-free path before writes are recoverable. We improve upon this with a deterministic and stateless coordinator design where the ground truth is the collective persistent state of participants, thereby removing this overhead.

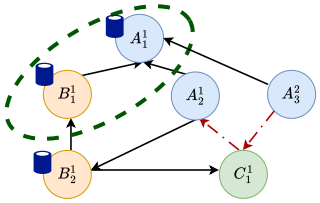


Figure 6: Example Dependency Graph

4.2 Protocol Details

The libDSE protocol centers on an explicit recovery dependency graph that all participants jointly maintain. We first define this graph, then describe the sub-protocols that operate on it: *instrumentation* populates the graph as the application runs, *boundary-finding* identifies regions of the graph that are safely committed and may be externally exposed, and *recovery* uses the graph to coordinate rollback on failure.

Dependency Graph. We show an example dependency graph in Figure 6. Each vertex of the graph is a recoverable point that can be loaded via `Restore`, and is uniquely identified by a combination of a `StateObject` id, a global failure counter (more details on this later), and a local persistence counter. For example, A_2^1 is the recoverable point from `StateObject` A with local persistence counter 2 and global failure counter 1. Vertices start off volatile, but become persistent as the underlying `StateObject` persists their state. In the example, A_1^1 is persistent, but A_2^1 is not yet. Edges of the dependency graph represent recovery dependencies – an edge from u to v iff u recovering without v results in an inconsistency. By definition of consistency [53], such a dependency is established either implicitly by precedence (i.e., u is later than v in the same `StateObject`), or if u received a message originating from state captured by v. Because each recoverable point captures multiple state transitions and messages, the dependency graph may have cycles. Because libDSE tracks dependencies at the granularity of recoverable points, rolling back a recoverable point wipes out all state updates and messages captured, even if only one was poisoned. This false sharing is a deliberate design choice. Per-message dependency tracking would impose substantial bookkeeping overhead on the failure-free path and require multi-round communication on failure to disambiguate causality. Another subtle but important property of this formulation is that recovery dependencies are established at *message consumption*, not at data read or write. This is what allows libDSE to handle indirect, control-flow dependencies uniformly with the data dependencies that prior work

focuses on. Consider a workflow orchestrator that receives a confirmation from service A and, based on the confirmation, conditionally invokes service B. The orchestrator never reads or writes A's state directly — but its decision to branch toward B causally depends on having consumed A's message. By tracking dependencies at message consumption, libDSE correctly classifies the invocation of B as recoverable only if A's confirmation is recoverable, even though the data flow between them is opaque.

Instrumentation Protocol. libDSE tags each message with its originating vertex on the dependency graph (the content of the aforementioned libDSE headers). If a `StateObject` receives a message from v when its current (not yet persisted) recoverable point is u, it adds an edge from u to v in the dependency graph. Each `StateObject` accumulates dependency graph updates and sends them periodically to the libDSE coordinator after persistence.

Boundary Protocol. A *Recoverable Boundary* is a cut of the graph where no future failure will cause vertices within the boundary to roll back. Messages originating from within the boundary are therefore not speculative and safe to expose externally. Recoverable boundaries manifest on the graph as closures, i.e., sets of vertices that 1) are all recoverable and 2) have no edge leading to non-recoverable vertices. For example, in Figure 6, A_1^1 and B_1^1 form a closure, but A_1^1, B_1^1, and B_2^1 do not because B_2^1 depends on C_1^1, which is not yet persistent. The coordinator finds boundaries by periodically searching the graph. However, a naïve implementation is susceptible to *the domino effect* [42], where frequent communications cause a node to always transitively depend on future, uncommitted versions, and there is no way to bound the blast radius of a rollback. To prevent this:

Definition 4.1. (Commit Ordering Rule) A_y^x is allowed to receive a message from B_n^m only if $y \geq n$.

For example, under this scheme, there cannot be an edge (red dotted arrow) from C_1^1 to A_2^1. The libDSE runtime would block an action from starting if the message it consumes has a larger persistence counter. Meanwhile, libDSE transparently issues `Persist`, which advances the local counter (e.g., from C_1^1 to C_2^1); the action can start immediately after the call without waiting for actual persistence. Intuitively, this scheme prevents the domino effect by guaranteeing that for any vertex A_y^x, the set of vertices with persistence number $\leq y$ is a closure that includes A_y^x. As a trade-off, participants that communicate frequently passively synchronize and will tend to issue persists on similar schedules.

Recovery Protocol. Failures lead to vertex loss on the dependency graph, and the system state is inconsistent iff any surviving vertex has an edge into a lost vertex. To recover, the system must roll back vertices until no such violations exist. The primary challenge here is that the system must achieve *consensus* on rollbacks; otherwise, overlapping failures or out-of-sync views of the dependency graph may result in different parts of the system recovering to different closures and creating inconsistency in the process. The global failure counter from before is how we encode consensus – upon failure, a leader assigns a monotonically increasing and unique failure sequence number to the rollback. Each participant of the rollback then recovers to the prescribed state at their own pace. In the example, A_3^2 signals that A has recovered from a failure and is now operating in the post-recovery world. Crucially, while every participant must eventually acknowledge recovery, they do not necessarily actively roll back if they are not impacted. A node like C_1^1 does not depend on any external nodes, and therefore can simply rename itself to C_2^1 without active recovery work. However, during recovery, there is a temporary divergence in the cluster between pre-recovery and post-recovery halves (or more if there are multiple concurrent failures). libDSE disallows communication across failure counters for safety:

Definition 4.2. (Recovery Partition Rule) A_y^x is allowed to receive a message from B_n^m only if $x = m$. If $m < x$, the message is discarded; otherwise, its receipt must be delayed.

In the example, if post-recovery A now receives a message from C_1^1, it again cannot receive the message (red dotted arrow), because otherwise, it would have potentially acted on a message from pre-recovery world that will be rolled back in the post-recovery world, polluting it. Instead, A would reply to C telling it about the failure and the action the cluster took to resolve it, allowing C to reconcile its view of the cluster state.

Supporting sthreads. Unlike `StateObject`s, sthreads are derived state and never appear as vertices in the dependency graph, but they do carry a dependency *set* that is updated as the sthread executes. On receiving a message tagged v, the sthread inserts v into its set. On send, the full set is attached to the header, so a sthread-originated message carries a set of triples rather than a single triple. The set is merged into the receiving `StateObject`'s in-memory vertex, producing the corresponding incoming edges as if the sthread's predecessors had sent directly. At every barrier or merge, the set is cleared, preventing unbounded growth. This is safe because a barrier is a quiescence point: any speculative work the sthread is carrying either commits past the barrier or is abandoned on rollback, and either way is no longer the sthread's dependency to track.

4.3 Coordinator Design

We now describe the design of our coordinator. Unlike DPR's, the libDSE coordinator is backed by a persistent log (either through distributed consensus like Raft [64] or reliable cloud storage) that encodes changes to the cluster state, such as membership changes and recoveries.

Finding Boundaries. LibDSE automatically persists fragments of the dependency graph as part of `StateObject`s us-

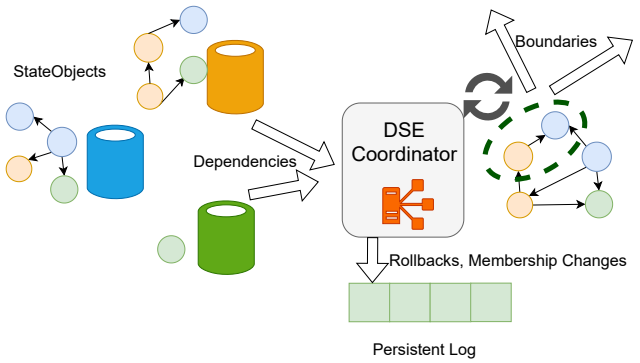


Figure 7: libDSE Coordinator Design

ing the metadata parameter on `Persist`. This establishes the distributed components as the ground truth; the coordinator maintains a (potentially stale) view of the actual graph. Newly persistent versions and in-flight operations may be missing from the coordinator's view. That said, the persistent part of the graph is *immutable*. Future operations may add vertices, but cannot change dependencies in the past. This makes it safe for the coordinator to declare recoverable boundaries on the coordinator's view, because any recoverable boundaries the coordinator finds on its present view must also be recoverable on a later view. Consequently, the coordinator does *not* persist its decisions on the failure-free path – restarted coordinators instead recompute the boundary on a more up-to-date view to find the same (or larger) recoverable boundary. To find recovery boundaries, the coordinator runs a standard graph BFS over its in-memory view of the graph and publishes its results immediately.

Orchestrating Rollback. A rollback is triggered when one of the `StateObject`s fails, and the coordinator computes the extent of data loss by removing any vertices on the dependency graph not reported as persistent by the restarting `StateObject`. Then, it iteratively removes vertices from the graph until no surviving vertex has a dangling edge. As discussed in Section 4.2, on failure, the coordinator must achieve *consensus* between `StateObject`s on what the final state of the graph is in response to the restart. This is achieved by synchronously persisting the failure (by assigning it a failure sequence number) and the rollback actions proposed, before releasing the decision to the rest of the cluster on the log.

Coordinator Recovery. Even though the coordinator is ephemeral, when a coordinator fails, the system must still perform some necessary steps to ensure correctness. On recovery, the new coordinator needs to reconstruct the cluster state, including previous rollback decisions and the list of active participants by replaying the persistent log. It then resends recent rollback decisions and asks for every participant to send locally stored dependency graph segments, which guarantees a view of the dependency graph that is more up-to-date than before the failure. The coordinator cannot answer queries about the current recovery boundary until every participant has responded to ensure a complete view of the graph.

4.4 Correctness Sketch

A full formal proof is out of scope for this paper. We sketch the informal argument here, addressing safety (the protocol never externally exposes inconsistent state), liveness (the protocol always makes progress), and the rollback bound.

Safety. Our libDSE protocol guarantees that external entities do not observe rollback-induced anomalies. Because the system exposes a message externally only after its originating vertex lies within a declared recoverable boundary $\mathcal{B}$. By definition, every out-edge of $u \in \mathcal{B}$ points to another vertex in $\mathcal{B}$, and every vertex in $\mathcal{B}$ has been reported persistent. A future failure can only invalidate vertices not yet persistent, which will not affect any vertex in $\mathcal{B}$. The Recovery Partition Rule additionally prevents participants from acting on or communicating across different pre/post-recovery splits for all failures, so recovery will not introduce additional anomalies.

Liveness. Given liveness of local persistence and recovery, libDSE guarantees to eventually declare every message non-speculative or rolled back. In the failure-free path, this is because 1) local persistence advances independently of each other and does not block on the progress of others (i.e., no deadlock), and 2) the Commit Ordering Rule guarantees finite closure for every recoverable point, which will eventually all become persistence given local liveness. On the failure path, each failure receives a strictly increasing sequence number on the coordinator's persistent log, and by assumption each induced rollback completes in finite time. A message m originating from u is therefore, in finite time, either (i) rolled back, if u lies in the lost set of some failure, or (ii) preserved across the failures u experiences, in which case the failure-free argument applies during any subsequent quiescent interval.

Bounded Rollback. As argued previously, the Commit Ordering Rule ensures $\{v : \text{counter}(v) \leq k\}$ is a closure for any k. Therefore, when a `StateObject` loses state past counter k, the rollback at every other participant is bounded by that participant's own persistence progress past k.

5 Implementation and Discussion

5.1 Putting it Together: libDSE Runtime

The libDSE runtime implements the aforementioned programming model, and connects each `StateObject` to the coordinator. The core libDSE and coordinator implementation consists of approximately 4000 lines of C# code.

gRPC Integrations. libDSE has built-in integration with gRPC [21] and ASP.NET [6]. To create a speculative service, users first write the service API using vanilla gRPC, encapsulate the state with a `StateObject`, and implement the generated gRPC stub using the libDSE API. During runtime, a

libDSE gRPC interceptor [23] automatically injects action boundaries before and after RPC handlers execute, and uses the HTTP headers to transparently pass libDSE headers. The presence of the special header is also used to infer whether the endpoint has speculation enabled (e.g., an external user would believe the speculative service is just another gRPC endpoint and send requests without libDSE headers), and automatically apply barriers to hide speculation. Advanced users can disable the interceptor and manually call libDSE APIs where necessary. Finally, all background logic of libDSE is packaged as an ASP.NET managed service, and users can start an ASP.NET-based speculative service simply by declaring these dependencies and leaving the rest to the ASP.NET framework.

Cluster Orchestrator Integrations. When starting an application, `StateObject`s first `Connect` to the coordinator to report its presence with a predetermined unique ID. If a `StateObject` that has already reported starting does so again, libDSE treats this as indicative of a failure. We observe that this model aligns well with current practices – for example, a user using Kubernetes to manage microservices expects Kubernetes to detect down services and replace them by relaunching in a new container. The relaunched service, upon its `Connect` call, will set off failure handling logic. When `Connect` returns, libDSE has registered the calling `StateObject` as the legitimate incarnation which may start sending and receiving messages. libDSE guarantees that any messages received from or sent to the previous incarnation will eventually be rolled back, but it is up to service implementers to ensure that two incarnations do not simultaneously update persistent state (e.g., with a Persistent Volume Claim in Kubernetes). It is not necessary for a service to explicitly disconnect from a libDSE coordinator, as long as it persists all of its outstanding state before becoming inactive. The coordinator sends periodic heartbeat/boundary update messages to participants, which includes instructions for garbage collecting dependency graphs and outdated recoverable points.

Action Synchronization. To guarantee atomicity of actions, each action executes under a shared lock while `Persist` and `Restore` execute under an exclusive lock. Because actions are far more common than `Persist` and `Restore`, we use biased locking [40, 58, 59, 68] to improve scalability.

5.1.1 Pattern: Speculative Log

A foundational pattern for stateful services is to treat a durable, append-only log as the source of truth, with the service's in-memory state acting as a materialized view over that log. Many systems, including databases, key-value stores, and workflow engines, fall under this model. This model is an excellent fit for libDSE, as it provides a principled way to adapt a wide range of applications to speculative execution by first building a speculative log. The most significant change required from a standard log implementation is attaching custom metadata with `Persist` calls. We achieve this by designing special commit records that hold metadata, and appending them to the end of a log at the beginning of `Persist`. To restore, we scan the log until the requested commit record and truncate the tail. When rolling back, we cache the offsets of speculative commit records in-memory to avoid the scan. On top of the log, service implementers are free to maintain arbitrary in-memory state that can be reconstructed by replaying the log's entries. While this model is powerful, reconstructing in-memory state from replaying can be prohibitively expensive. This is a well-studied problem in prior literature from database systems [60] addressed by periodic checkpointing of an in-memory state and log truncation.

5.1.2 Applying the Speculative Log Pattern

With the speculative log, implementing other complex stateful services is equivalent to maintaining different materialized views over it. This approach demonstrates the versatility of the pattern, allowing us to adapt libDSE to a wide range of common services. We describe three examples below.

Key-Value Store. We implement a speculative key-value store based on the open-source FASTER key-value store [18], which powers production systems such as Orleans [29], Durable Functions [8], and Garnet [24]. FASTER uses FasterLog as its backend and mostly maintains an in-memory hash index over the log to serve key-value workloads. To make FASTER speculative, we only need to make its underlying log speculative and access it using libDSE clients. We reuse most of FASTER's sophisticated storage and checkpointing schemes [36, 65] without modification.

Speculative Workflows. We base our speculative workflow orchestration system on the DARQ system [33, 56], which is itself a wrapper around a log. At a high-level, the DARQ log records the entire history of workflow events and commands, and relies on replay and deduplication to ensure exactly-once execution. Our implementation replaces DARQ's backend log with the speculative version, and triggers replay of the workflow system when the log is rolled back.

Event Broker. We implement a speculative event broker system similar to Kafka [52] and EventHubs [7], with a collection of producers and consumers interacting through disjoint topics. Each topic is fundamentally just a log, which we have replaced with a speculative log. We leave creation, deletion, and management of topics non-speculative, as they are usually not on the critical path.

Transactional Store. Lastly, we implement a transactional store with the standard two-phase commit protocol [60] and strict two-phase locking. Tuples are stored in main memory. We rely on a speculative log as write-ahead log for recoverability and persistence, and libDSE thread primitives for synchronization.

5.2 Limitations and Discussion

Developer Burden. LibDSE is not a transparent, drop-in solution for existing applications. It requires developers to adopt its programming model, encapsulating application state within the `StateObject` abstraction and using actions and sthreads for concurrency control. This may require significant refactoring for legacy applications lacking clear state management boundaries. That said, in our experience, most of these changes align with modern best practices for building correct and scalable microservices. Furthermore, libDSE naturally promotes the creation of highly reusable, foundational components, such as the services we discussed above, which can be implemented once and reused across many applications, amortizing the initial development cost. Adoption can be further simplified with integration with programming languages, but we leave this for future work.

Data Loss. From the external user's perspective, DSE does **not** introduce data loss, as speculative state is withheld from users, and only speculative state is lost on failure. However, our DSE recovery protocol is designed for simplicity and speed during failure, as it avoids complex, multi-round consensus protocols between participants. The trade-off is that this can lead to coarser-grained rollbacks than are strictly necessary, sacrificing speculative throughput (i.e., work loss instead of data loss) to minimize recovery latency. Because the coordinator's view of the dependency graph may be slightly stale, it must conservatively roll back all speculative work that could have depended on a failed component's lost state. We mitigate this in practice by allowing a service to skip a rollback if it can prove locally that it has no dependencies on the speculative state in question. We study the effects of rollback in more detail in Section 6.

Coordination Overhead. Relying on a centralized coordinator creates a potential bottleneck for scalability. libDSE mitigates this risk by decoupling coordination from the execution critical path. `StateObject`s communicate peer-to-peer and do not contact the coordinator during standard message processing. Dependency graph updates are batched and transmitted asynchronously only upon local persistence (e.g., every 10ms), independent of operations, resulting in control traffic that is proportional to the group commit frequency rather than application throughput. Furthermore, the coordinator finds recovery boundaries as a background process. A saturated coordinator may delay the visibility of results to external clients (by advancing the boundary less frequently), but it does not throttle the internal execution throughput of the speculative cluster. We observe that a single coordinator instance is usually enough to handle the metadata traffic of high-throughput workloads without saturation, but coordinator compute can also be parallelized if necessary.

Compatibility with Replicated Systems. The fail-restart semantics of a `StateObject` are a natural fit for services that use single-primary replication (e.g., State Machine Replication or Primary-Backup), where the volatile state exists on the primary node, and recovery is achieved by switching to a backup. However, the model is a poor fit for eventually consistent systems (e.g., Dynamo-style databases [38]). These systems lack a clear, linearizable notion of "restarting" from a single consistent point in time, making them difficult to model as a `StateObject`. That said, persistence is often cheaper in such systems, making speculation less critical and applicable.

Compatibility with External Infrastructure. libDSE assumes that participants in a speculative group are DSE-aware and implement the `StateObject` API. Real applications, however, frequently interact with infrastructure that is not, such as a shared SQL database or an external payment API. We support two strategies for these cases. When the external component is fully external to the speculative group (e.g., a payment API whose side effects cannot be rolled back), services interact with it through the `Barrier()` primitive, which blocks until all upstream speculative dependencies have committed. This forfeits speculation across the barrier but is always safe, and importantly allows libDSE to be adopted incrementally: speculation is enabled only between DSE-capable components, while legacy components continue to participate behind barriers. When the external component is a shared data store that all speculative participants access (e.g., a shared relational database used for coordination among speculative services), the external component can participate in speculation by internally tracking speculative dependencies—an arrangement equivalent to schemes like early lock release [46] on the shared store.

6 Evaluation

Our evaluation studies the following research questions:

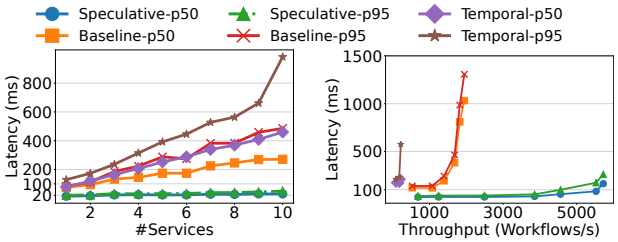
- Does DSE help applications reduce end-to-end latency?
- What overhead does libDSE impose on applications?
- Does libDSE scale to high throughput and/or concurrency?

Experimental Setup. We run all of our experiments on the Azure public cloud. For end-to-end experiments, we build the applications using the popular gRPC + ASP.NET stack and deploy them onto a managed Azure Kubernetes Service (AKS) cluster [28]; all workloads are scheduled onto a pool of 10 Standard_D8s_v3 machines [15] with attached premium locally redundant storage SSDs [9]. Unless otherwise specified, all libDSE services run with a group commit interval of 10ms. For microbenchmarks, we use a pair of D32s_v3 machines [15], each with 32 vCPUs and 128 GB of RAM.

Engineering Effort. Table 2 summarizes the engineering effort required to build the evaluated services. We implement the speculative log by modifying the open-source FasterLog project from Microsoft [19], which involved no modification of the log itself and only ∼200 lines of wrapper logic to expose it as a speculative service. Similarly, porting the

Table 2: Engineering effort for libDSE-based services.

Component	LOC	Base System
Speculative Log	∼200	FasterLog [19]
Key-Value Store	∼400	FASTER KV [18]
Workflow Engine	∼200	DARQ [56]
Event Broker	∼800	DARQ [56]
Transactions	∼2000	From scratch + FasterLog



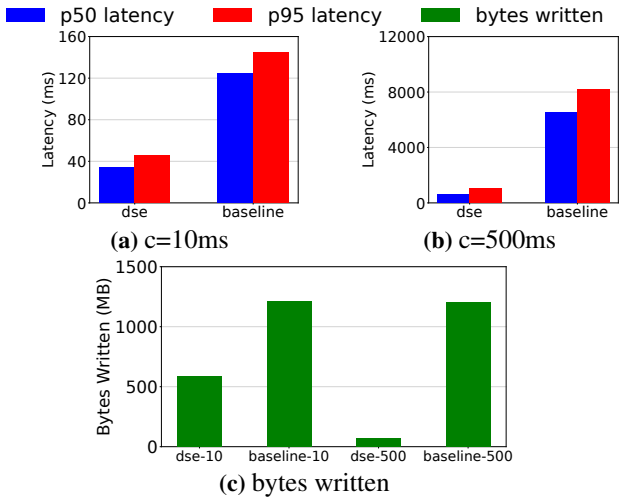
(a) Varying #Services **(b)** Varying Load

Figure 8: TravelReservations

FASTER Key-Value store [18] and the DARQ system (which is an event processing framework built on FasterLog) [56] required minimal code changes (∼400 and ∼200 LOC, respectively), primarily to wrap state objects and implement service-level logic. The Event Broker required slightly more glue code (∼800 LOC) for the gRPC streaming API. The Transactions benchmark is the largest component (∼2000 LOC) because it implements the 2PL + 2PC transaction processing and the TPC-C benchmark [72] as stored procedures from scratch on top of our speculative log, rather than modifying an existing codebase. However, this logic is largely orthogonal to speculative execution, and in our experience, is not further complicated by libDSE. These figures demonstrate that libDSE effectively abstracts the complexity of distributed dependency tracking, allowing developers to adopt speculative execution with minimal engineering overhead.

6.1 End-to-End Benchmarks

TravelReservations. We first assemble a travel reservation system based on DeathStarBench [43], where a workflow engine reserves a series of items, one from each service (e.g., hotel, flight, car-rental) backed by our speculative KV store using sagas [31]. We focus on the *write* portion of the workload. We exclude read operations as they are typically served from local cache in both baselines, exhibiting identical performance; focusing on writes allows us to isolate the overhead and latency benefits of libDSE's persistence mechanism. Note that this does not mean libDSE does not help read-only workloads: many durable execution engines log reads for correctness (e.g., to ensure deterministic replay during recovery), effectively turning reads into writes that libDSE also accelerates. We compare our implementation with the open-source version of Temporal [30], an industry-standard workflow-engine, deployed over an Azure-managed Cassandra cluster and using Azure CosmosDB for application logic. Additionally, we evaluate our system with speculation disabled to isolate the performance gains from speculative execution vs. implementation and tool stack. All benchmarks run for 120 seconds.



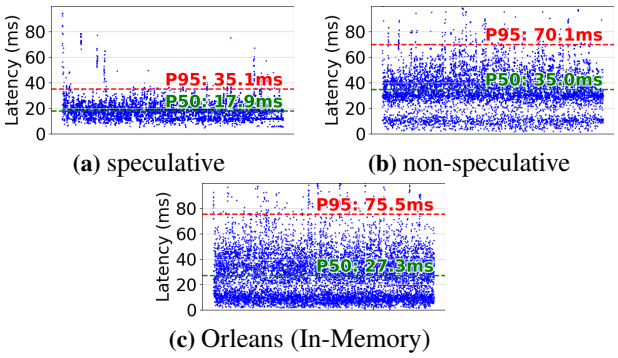
(a) c=10ms **(b)** c=500ms

(c) bytes written

Figure 9: EventProcessing

Figure 8 shows the result of our experiment. The left panel reports average and 95th-percentile latency as the number of participating services increases, using a steady rate of 10 workflows/s to avoid queuing delays. As the service chain grows, baseline latency increases linearly due to cumulative synchronous persistence costs. In contrast, libDSE incurs only minor latency increases due to RPC overhead, effectively bypassing persistence bottlenecks and achieving an order of magnitude lower latency for long workflows. We further demonstrate scalability by fixing the chain length to three services and varying the workflow arrival rate. As shown in the right panel of Figure 8, speculation significantly improves throughput capacity. Traditional workflow engines become CPU-bound under high load because they manage many concurrent workflow instances simultaneously blocked on synchronous persistence, each consuming non-trivial context-management overhead (thread state, stack frames, runtime bookkeeping, etc.). libDSE completes workflows rapidly, sharply reducing the number of in-flight blocked contexts and thus the associated CPU pressure—enabling higher sustained throughput on identical hardware.

Event Processing. We evaluate the search trend alert workload described in the DARQ paper [56] (available in open-source [33]) using our speculative event broker. The workload consists of three processing stages communicating via streaming events. We compare our implementation against the original, non-speculative version of DARQ. Figure 9 presents the results of processing a pre-generated event trace at a steady rate of 50k events/s for 120 seconds, while varying the group commit interval (c). Note that c governs the persistence of *intermediate* stream-processing state. While a larger c implies a wider window, meaning more intermediate results are lost



(a) speculative **(b)** non-speculative

(c) Orleans (In-Memory)

Figure 10: TwoPhaseCommit

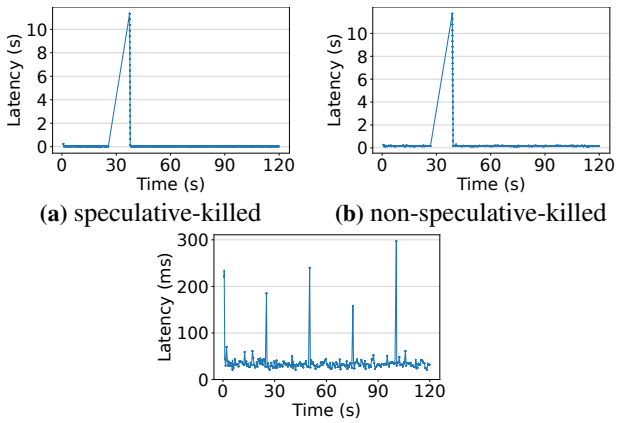
during cluster failures that must be replayed, it does not lead to additional loss of user data, which can be persisted eagerly at ingestion. The first two plots demonstrate that libDSE drastically reduces end-to-end latency compared to the baseline. The bottom graph illustrates the volume of bytes written to storage; libDSE achieves significant reductions in storage bandwidth because it prunes short-lived intermediate results that are generated and consumed entirely within the speculative window, preventing them from ever reaching storage. Consequently, these storage savings are more pronounced with larger group commit intervals, which increase the likelihood that intermediate state is pruned before disk I/O occurs.

Two-Phase Commit. Finally, we evaluate our transactional store. The benchmark cluster consists of four shared-nothing shards, each maintaining its own (speculative) log. We evaluate a modified TPC-C workload [72] consisting of 100% distributed transactions, comparing our implementation against Microsoft Orleans's optimized transactional framework [29, 41]. To ensure fair comparison, we manually place Orleans grains to have the same sharding with our system, and run Orleans purely in-memory to eliminate the throughput limitations of its standard Azure Table Storage backend.

Figure 10 illustrates the commit latency distribution. Non-speculative latencies cluster around multiples of the 10ms group commit interval, as most transactions must wait for the next persistence window. There are a few "lucky" transactions that complete the first round of messages at the end of the last group commit and therefore finish close to 10ms, but most transactions wait longer. In contrast, speculative execution eliminates sequential waiting between phases, allowing the protocol to overlap communication with I/O. The plots show most transactions finishing under 20 ms (essentially one round of group commit + RPC overheads). Consequently, our speculative implementation significantly outperforms both the non-speculative baseline and the in-memory Orleans setup.

6.2 Recovery

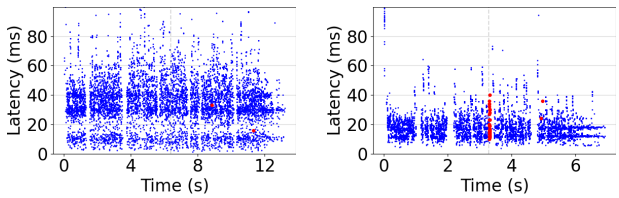
Event Processing. We evaluate realistic recovery scenarios using the EventProcessing workload by terminating a Kubernetes event handling node at the 30-second mark. Kubernetes



(a) speculative-killed **(b)** non-speculative-killed

(c) speculative-simulated

Figure 11: EventProcessing-recovery



(a) non-speculative **(b)** speculative

Figure 12: TwoPhaseCommit-recovery

automatically provisions a replacement container, triggering libDSE's recovery protocol upon startup. During this interval, gRPC requests to the affected node time out and are retried, temporarily halting processing for that partition. Figure 11 shows that recovery takes approximately 10 seconds for both speculative and non-speculative versions, a duration dominated by container orchestration latency rather than protocol overhead. To isolate the overhead of libDSE's rollback mechanism from orchestration delays, we also introduce simulated failures that trigger immediate rollback across all participants without container restarts. This configuration represents the theoretical best-case recovery performance, exposing only the latency induced by speculative state restoration. We inject four independent failures and observe that the system stabilizes rapidly, resulting in latency spikes of 100s of milliseconds.

Two Phase Commit. Similarly, we evaluate recovery in the transaction benchmark using simulated failures, comparing against a failure-free non-speculative baseline. Figure 12 plots the results, where red dots indicate aborted transactions. The primary trade-off of speculative execution is a temporary spike in user-visible aborts during rollback; to ensure safety, our implementation aggressively aborts all in-flight transactions that may depend on invalidated state. However, this effect is minimal in practice. Quantitatively, speculative recovery increases the overall abort rate by only 0.3% compared to the baseline. Because these transactions can be immediately retried, the aggregate impact on system throughput is negligible. Note that the measurement gaps visible in both

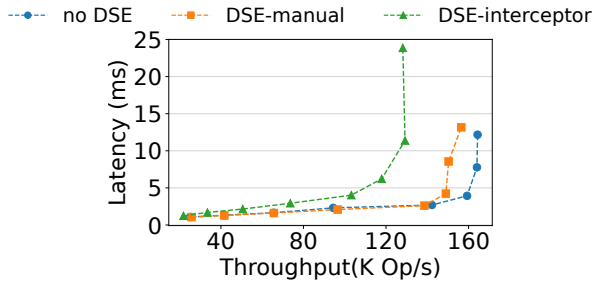
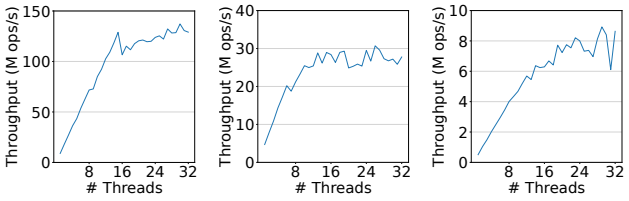


Figure 13: Overhead of libDSE instrumentation



(a) local-action **(b)** send-receive **(c)** detach-merge

Figure 14: Thread scalability of libDSE Primitives

plots correspond to standard server GC pauses. The gaps in the speculative version appear longer because of the different scale on the x-axis (speculative baseline finishes the workload much faster due to higher throughput).

6.3 Microbenchmarks

Finally, we present a series of microbenchmarks to study the overhead of various libDSE mechanisms.

Overhead of libDSE Instrumentation. We first study the latency and throughput overhead of libDSE's message instrumentation protocol on a service. We study three versions of the FASTER-based key-value store from the hotel reservation benchmark, two with libDSE (DSE), and another as a thin RPC wrapper around the vanilla FASTER code (no DSE). To study the effects of the interceptor mechanisms vs. the instrumentation protocol itself, we also build one version that processes libDSE headers in user code (DSE-manual) and another that does so transparently with gRPC interceptors (DSE-interceptor). We vary the request issue rate for each configuration to investigate the resulting latency-throughput trade-off. Figure 13 shows that libDSE with interceptors has slightly higher latency when the system is not saturated and about 25% lower maximum throughput. This is mostly due to the additional work entailed by the gRPC interceptor mechanism (e.g., HTTP header manipulation). In contrast, the libDSE protocol itself causes a negligible increase in latency and a less than 5% reduction in throughput. This shows that the libDSE instrumentation adds only a small overhead to applications.

Thread Scalability of libDSE Primitives. Recall that libDSE users must protect their operations with libDSE threading primitives for correct speculative execution. It is therefore important that libDSE primitives be scalable and not introduce new bottlenecks in an application. To measure this, we design a simple microbenchmark where threads acquire protection under a tight-loop concurrently, reporting throughput as the number of concurrent threads increases. A background thread periodically performs (empty) checkpoints to advance versions. We measure three sets of primitives: local-action, where libDSE users mark blocks of code as atomic with respect to checkpointing and recovery; send-receive, where libDSE users start an action using a (randomly pre-populated) header and write the resulting dependencies to another header at the end of the action; and detach-merge, where libDSE users create a new sthread from the current dependencies and then merge it with the state object. As shown in Figure 14, our implementation of the action primitive (based on [58]) is scalable up to 16 threads with no performance degradation. As expected, send-receive is more expensive than local-action, and detach-merge is more expensive than send-receive, due to the amount of computation involved. However, all three primitives can sustain millions of operations per second on a single server, making them unlikely to be bottlenecks in typical RPC-oriented workloads that only see 100s of thousands of operations per second per node.

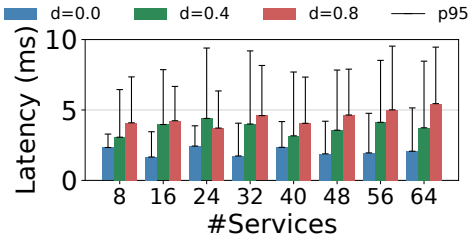


Figure 15: libDSE Coordinator Scalability

Coordinator Performance. Finally, we study the libDSE coordinator, which is responsible for identifying messages that are no longer speculative so their effects can be exposed externally. We design a microbenchmark in which simulated participants submit versions to a real coordinator on a predetermined schedule, and measure the latency between submission and commit. Each version depends on each other participant's current version independently with probability d. Participants run on a Kubernetes cluster, communicate with the coordinator over real intra-datacenter network connections, and refresh their cluster view every 5ms. Figure 15 reports median commit latency (bars) with the 95th-percentile tail as whiskers. A local-commit fast path lets versions with no speculative dependencies commit almost immediately, so the $d = 0.0$ bars sit near 2ms across the entire range and the corresponding p_{95} remains within one 5 ms refresh interval. In the general case, both median and p_{95} scale smoothly from 8 to 64 services: medians stay below a single refresh interval. Graph connectivity has only a modest impact, as in-memory traversal is cheap relative to network and refresh delays.

7 Related Work

Durable Execution and Workflows. Durable Execution, as a term, was first coined by Temporal [10], and is widely accepted in industry [8, 13, 14, 16, 17, 20, 22, 27, 32], although it is similar to earlier concepts of virtual resiliency [4, 45] and reliable workflows [5, 11, 25]. Most of these systems achieve resilience guarantees by modeling the application as a directed acyclic graph (DAG) of tasks and synchronously persisting execution state before starting new steps; external effects of execution are wrapped in special abstractions (e.g., activities in Temporal and DF, impulses in Ambrosia) and required to be idempotent for resilience. ExoFlow [78] is a recent workflow system that allows developers to take advantage of determinism and rollbacks within the workflow for better performance. Developers mark parts of their workflow as deterministic or rollback-capable, and ExoFlow will bypass synchronous persistence where possible, using replay and rollback to recover when necessary. In contrast, libDSE allows true inter-service speculation without relying on determinism.

Cross-Service Transactions. Other systems, such as Olive [69], co-locate service state and orchestration state onto one unified storage layer, providing exactly-once guarantees without requiring idempotency, but also forcing developers to migrate to their storage engine. While Olive used existing database services (e.g., Azure Tables), Boki [48] showed that a shared log abstraction can provide more performance and usability benefits, and Halfmoon [66] further optimizes Boki to reduce the amount of information logged. More recently, SpecLog [35] applies speculation within the shared-log setting itself: it introduces *fix-ante ordering*, which predetermines the global record order via per-shard quotas so that applications can begin processing records before the final global cut is committed. libDSE is complementary in that it speculates across the synchronous persistence of independent message-passing services (i.e., between independent logs) rather than inside the ordering layer of a single log. Other systems support distributed transactions in the unified storage layers [71, 76]. Orleans, of particular note, implemented early lock release [3, 39, 41, 44] to reduce latency. While conceptually similar to DSE, their implementation only applies to Orleans' transactional workers. In contrast, libDSE is designed to work for heterogeneous participants and *per-component private storage*. While some transactional protocols allow distributed transactions across heterogeneous storage engines [1, 51, 77], they are not widely used currently to our knowledge. Finally, DBOS [12, 50] proposes to build all services and runtime layers on top of a distributed transactional DBMS, but would require an even more radical rewrite of existing technology stacks than libDSE.

Other Related Work. Fault-tolerance in microservice-oriented cloud applications can be viewed as a modern instantiation of asynchronous recovery in message-passing systems, which was heavily studied in prior work [42]. Dataflow systems are conceptually similar to workflow systems and compose distributed components, but usually have the benefit of known application semantics (i.e., composed components are operators such as filter and map, rather than blackbox code). Consequently, dataflow systems predominantly use lineage-based techniques to reconstruct application state on failure [61, 74, 75]. libDSE is, at its core, a checkpoint-based recovery scheme, which assumes no such domain-specific knowledge. The DSE protocol bears similarity to classic distributed snapshot algorithms [37], but is redesigned for an online setting where checkpoints happen continuously. The idea of a performant distributed rollback system reliant on logical time is similar to the well-known time warp operating system [47], which employs speculative execution to handle out-of-order simulation actions (synchronization). In contrast, libDSE employs the algorithm to hide persistence latency (durability). Finally, Speculator [62] and xsyncfs [63] similarly use speculative execution to speed up file system operations transparently. However, they assume a shared state model and implement client-centric speculation, whereas libDSE specializes for message-passing scenarios.

8 Conclusion

We presented distributed speculative execution (DSE), a novel approach to reduce persistence overhead in durably executed cloud applications. Through our framework, libDSE, we demonstrate that DSE is both practical and beneficial for performance, and that its complexities can be effectively hidden from most developers through the use of abstractions. Our results show that DSE significantly reduces latency. We envision DSE to be a key technique in building future highly distributed cloud applications.

Acknowledgments

This research was sponsored by the United States Air Force Research Laboratory and the Department of the Air Force Artificial Intelligence Accelerator and was accomplished underCooperative Agreement Number FA8750-19-2-1000. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Department of the Air Force or the U.S. Government. The U.S. Government is authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright notation herein. Additionally, our work has been supported by contributions from Amazon, Google, and Intel as part of the MIT Data Systems and AI Lab (DSAIL). We also thank Frans Kaashoek for his helpful comments and suggestions.

References

[1] Technical standard – distributed transaction processing: the xa specification. https://pubs.opengroup.org/onlinepubs/009680699/toc.pdf, 1991.

[2] Decomposing twitter: Adventures in service-oriented architecture. https://qconnewyork.com/ny2013/node/231.html, 2013. QCon New York 2013.

[3] Orleans transactions for middle-tier stateful applications. https://hpts.ws/papers/2019/PhilBernsteinHPTS2019.pdf, 2019.

[4] Ambrosia: Robust distributed programming made easy and efficient. https://microsoft.github.io/AMBROSIA/, 2024.

[5] Apache airflow. https://airflow.apache.org/, 2024.

[6] Asp.net core. https://dotnet.microsoft.com/en-us/apps/aspnet, 2024.

[7] Azure event hubs: A real-time data streaming platform with native apache kafka support. https://learn.microsoft.com/en-us/azure/event-hubs/event-hubs-about, 2024.

[8] Azure functions overview. https://learn.microsoft.com/en-us/azure/azure-functions/functions-overview, 2024.

[9] Azure storage redundancy. https://learn.microsoft.com/en-us/azure/storage/common/storage-redundancy, 2024.

[10] Building reliable distributed systmes in node.js. https://temporal.io/blog/building-reliable-distributed-systems-in-node, 2024.

[11] Cloud composer. https://cloud.google.com/composer, 2024.

[12] Dbos – transactional serverless platform for typescript. https://www.dbos.dev/, 2024.

[13] Durable execution explained – how conductor delivers resilient systems out of the box. https://orkes.io/blog/durable-execution-explained-how-conductor-delivers-resilient-systems/, 2024.

[14] Durable execution: Justifying the bubble. https://temporal.io/blog/building-reliable-distributed-systems-in-node, 2024.

[15] Dv3 and dsv3-series. https://learn.microsoft.com/en-us/azure/virtual-machines/dv3-dsv3-series, 2024.

[16] Everything wrong with databases and why their complexity is now unnecessary. https://blog.redplanetlabs.com/2024/01/09/everything-wrong-with-databases-and-why-their-complexity-is-now-unnecessary/, 2024.

[17] Fairy tales of workflow orchestration. https://stealthrocket.tech/blog/fairy-tales-of-workflow-orchestration, 2024.

[18] Faster: A fast concurrent persistent key-value store and log. https://microsoft.github.io/FASTER/, 2024.

[19] Fasterlog basics. https://microsoft.github.io/FASTER/docs/fasterlog-basics/, 2024.

[20] Flawless. https://flawless.dev/, 2024.

[21] grpc: A high performance, open source universal rpc framework. https://grpc.io/, 2024.

[22] How convex works. https://stack.convex.dev/how-convex-works, 2024.

[23] Interceptors. https://grpc.io/docs/guides/interceptors/, 2024.

[24] Introducing garnet – an open-source, next-generation, faster cache-store for accelerating applications and services. https://www.microsoft.com/en-us/research/blog/introducing-garnet-an-open-source-next-generation-faster-cache-store-for-accelerating-applications-and-services/, 2024.

[25] Kubeflow. https://www.kubeflow.org/, 2024.

[26] Linux kernel memory barriers. https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/Documentation/memory-barriers.txt, 2024.

[27] Littlehorse: Workflow-driven microservices. https://littlehorse.dev/, 2024.

[28] Managed kubernetes service (aks). https://azure.microsoft.com/en-us/products/kubernetes-service, 2024.

[29] Microsoft orleans. https://learn.microsoft.com/en-us/dotnet/orleans/overview, 2024.

[30] Open source durable execution | temporal technologies. https://temporal.io/, 2024.

[31] Saga distributed transaction pattern. https://learn.microsoft.com/en-us/azure/storage/common/storage-redundancy, 2024.

[32] Why we built restate. https://restate.dev/blog/why-we-built-restate/, 2024.

[33] Darq source code. https://github.com/microsoft/FASTER/tree/research/cs/research, 2025.

[34] H. Berenson, P. Bernstein, J. Gray, J. Melton, E. O'Neil, and P. O'Neil. A critique of ansi sql isolation levels. *SIGMOD Rec.*, 24(2):1–10, may 1995.

[35] S. G. Bhat, T. Hong, X. Luo, J. Hu, A. Ganesan, and R. Alagappan. Low end-to-end latency atop a speculative shared log with fix-ante ordering. In *Proceedings of the 19th USENIX Conference on Operating Systems Design and Implementation*, OSDI '25, USA, 2025. USENIX Association.

[36] B. Chandramouli, G. Prasaad, D. Kossmann, J. Levandoski, J. Hunter, and M. Barnett. Faster: A concurrent key-value store with in-place updates. In *Proceedings of the 2018 International Conference on Management of Data*, SIGMOD '18, page 275–290, New York, NY, USA, 2018. Association for Computing Machinery.

[37] K. M. Chandy and L. Lamport. Distributed snapshots: determining global states of distributed systems. *ACM Trans. Comput. Syst.*, 3(1):63–75, Feb. 1985.

[38] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: amazon's highly available key-value store. In *Proceedings of Twenty-First ACM SIGOPS Symposium on Operating Systems Principles*, SOSP '07, page 205–220, New York, NY, USA, 2007. Association for Computing Machinery.

[39] D. J. DeWitt, R. H. Katz, F. Olken, L. D. Shapiro, M. R. Stonebraker, and D. A. Wood. Implementation techniques for main memory database systems. *SIGMOD Rec.*, 14(2):1–8, jun 1984.

[40] D. Dice and A. Kogan. BRAVO—Biased locking for Reader-Writer locks. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 315–328, Renton, WA, July 2019. USENIX Association.

[41] T. Eldeeb, S. Burckhardt, R. Bond, A. Cidon, J. Yang, and P. A. Bernstein. Cloud actor-oriented database transactions in orleans. *Proc. VLDB Endow.*, 17(12):3720–3730, Aug. 2024.

[42] E. N. M. Elnozahy, L. Alvisi, Y.-M. Wang, and D. B. Johnson. A survey of rollback-recovery protocols in message-passing systems. *ACM Comput. Surv.*, 34(3):375–408, sep 2002.

[43] Y. Gan, Y. Zhang, D. Cheng, A. Shetty, P. Rathi, N. Katarki, A. Bruno, J. Hu, B. Ritchken, B. Jackson, K. Hu, M. Pancholi, Y. He, B. Clancy, C. Colen, F. Wen, C. Leung, S. Wang, L. Zaruvinsky, M. Espinosa, R. Lin, Z. Liu, J. Padilla, and C. Delimitrou. An open-source benchmark suite for microservices and their hardware-software implications for cloud & edge systems. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '19, page 3–18, New York, NY, USA, 2019. Association for Computing Machinery.

[44] D. Gawlick and D. Kinkade. Varieties of concurrency control in ims/vs fast path. *IEEE Database Eng. Bull.*, 8:3–10, 01 1985.

[45] J. Goldstein, A. Abdelhamid, M. Barnett, S. Burckhardt, B. Chandramouli, D. Gehring, N. Lebeck, C. Meiklejohn, U. F. Minhas, R. Newton, R. Ghosh Peshawaria, T. Zaccai, and I. Zhang. A.m.b.r.o.s.i.a: Providing performant virtual resiliency for distributed applications. Technical Report MSR-TR-2018-40, Microsoft, December 2018. This paper describes the main ideas and research behind the open source Ambrosia platform for writing resilient distributed applications.

[46] G. Graefe, M. Lillibridge, H. Kuno, J. Tucek, and A. Veitch. Controlled lock violation. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, SIGMOD '13, page 85–96, New York, NY, USA, 2013. Association for Computing Machinery.

[47] D. Jefferson, B. Beckman, F. Wieland, L. Blume, and M. Diloreto. Time warp operating system. In *Proceedings of the Eleventh ACM Symposium on Operating Systems Principles*, SOSP '87, page 77–93, New York, NY, USA, 1987. Association for Computing Machinery.

[48] Z. Jia and E. Witchel. Boki: Stateful serverless computing with shared logs. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, SOSP '21, page 691–707, New York, NY, USA, 2021. Association for Computing Machinery.

[49] E. Jonas, J. Schleier-Smith, V. Sreekanti, C.-C. Tsai, A. Khandelwal, Q. Pu, V. Shankar, J. Carreira, K. Krauth, N. Yadwadkar, J. E. Gonzalez, R. A. Popa, I. Stoica, and D. A. Patterson. Cloud programming simplified: A berkeley view on serverless computing, 2019.

[50] P. Kraft, Q. Li, K. Kaffes, A. Skiadopoulos, D. Kumar, D. Cho, J. Li, R. Redmond, N. Weckwerth, B. Xia, P. Bailis, M. Cafarella, G. Graefe, J. Kepner, C. Kozyrakis, M. Stonebraker, L. Suresh, X. Yu, and M. Zaharia. Apiary: A dbms-integrated transactional function-as-a-service framework, 2023.

[51] P. Kraft, Q. Li, X. Zhou, P. Bailis, M. Stonebraker, M. Zaharia, and X. Yu. Epoxy: Acid transactions across diverse data stores. *Proc. VLDB Endow.*, 16(11):2742–2754, jul 2023.

[52] J. Kreps, N. Narkhede, J. Rao, et al. Kafka: A distributed messaging system for log processing. In *Proceedings of the NetDB*, volume 11, pages 1–7. Athens, Greece, 2011.

[53] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, jul 1978.

[54] B. W. Lampson. Atomic transactions. In B. W. Lampson, M. Paul, and H. Siegert, editors, *Distributed Systems - Architecture and Implementation, An Advanced Course*, volume 105 of *Lecture Notes in Computer Science*, pages 246–265. Springer, 1980.

[55] C. Lee and J. Ousterhout. Granular computing. In *Proceedings of the Workshop on Hot Topics in Operating Systems*, HotOS '19, page 149–154, New York, NY, USA, 2019. Association for Computing Machinery.

[56] T. Li, B. Chandramouli, S. Burckhardt, and S. Madden. Darq matter binds everything: Performant and composable cloud programming via resilient steps. *Proc. ACM Manag. Data*, 1(2), jun 2023.

[57] T. Li, B. Chandramouli, J. M. Faleiro, S. Madden, and D. Kossmann. Asynchronous prefix recoverability for fast distributed stores. In *Proceedings of the 2021 International Conference on Management of Data*, SIGMOD '21, page 1090–1102, New York, NY, USA, 2021. Association for Computing Machinery.

[58] T. Li, B. Chandramouli, and S. Madden. Performant almost-latch-free data structures using epoch protection. In *Proceedings of the 18th International Workshop on Data Management on New Hardware*, DaMoN '22, New York, NY, USA, 2022. Association for Computing Machinery.

[59] T. Li, B. Chandramouli, and S. Madden. Performant almost-latch-free data structures using epoch protection in more depth. *The VLDB Journal*, 2024.

[60] C. Mohan, B. Lindsay, and R. Obermarck. Transaction management in the r* distributed database management system. *ACM Trans. Database Syst.*, 11(4):378–396, dec 1986.

[61] D. G. Murray, F. McSherry, R. Isaacs, M. Isard, P. Barham, and M. Abadi. Naiad: a timely dataflow system. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, page 439–455, New York, NY, USA, 2013. Association for Computing Machinery.

[62] E. B. Nightingale, P. M. Chen, and J. Flinn. Speculative execution in a distributed file system. *SIGOPS Oper. Syst. Rev.*, 39(5):191–205, oct 2005.

[63] E. B. Nightingale, K. Veeraraghavan, P. M. Chen, and J. Flinn. Rethink the sync. *ACM Trans. Comput. Syst.*, 26(3), Sept. 2008.

[64] D. Ongaro and J. Ousterhout. In search of an understandable consensus algorithm. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)*, pages 305–319, Philadelphia, PA, June 2014. USENIX Association.

[65] G. Prasaad, B. Chandramouli, and D. Kossmann. Concurrent prefix recovery: Performing cpr on a database. In *Proceedings of the 2019 International Conference on Management of Data*, SIGMOD '19, page 687–704, New York, NY, USA, 2019. Association for Computing Machinery.

[66] S. Qi, X. Liu, and X. Jin. Halfmoon: Log-optimal fault-tolerant stateful serverless computing. In *Proceedings of the 29th Symposium on Operating Systems Principles*, SOSP '23, page 314–330, New York, NY, USA, 2023. Association for Computing Machinery.

[67] D. P. Reed. Naming and synchronization in a decentralized computer system. Technical report, USA, 1978.

[68] K. Russell and D. Detlefs. Eliminating synchronization-related atomic operations with biased locking and bulk rebiasing. In *Proceedings of the 21st Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications*, OOPSLA '06, page 263–272, New York, NY, USA, 2006. Association for Computing Machinery.

[69] S. Setty, C. Su, J. R. Lorch, L. Zhou, H. Chen, P. Patel, and J. Ren. Realizing the Fault-Tolerance promise of cloud storage using locks with intent. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 501–516, Savannah, GA, Nov. 2016. USENIX Association.

[70] J. E. Smith. A study of branch prediction strategies. In *Proceedings of the 8th Annual Symposium on Computer Architecture*, ISCA '81, page 135–148, Washington, DC, USA, 1981. IEEE Computer Society Press.

[71] V. Sreekanti, C. Wu, S. Chhatrapati, J. E. Gonzalez, J. M. Hellerstein, and J. M. Faleiro. A fault-tolerance shim for serverless computing. In *Proceedings of the Fifteenth European Conference on Computer Systems*, EuroSys '20, New York, NY, USA, 2020. Association for Computing Machinery.

[72] Transaction Processing Performance Council (TPC). TPC Benchmark™C, Standard Specification (Revision 5.11). Technical Report, February 2010.

[73] G. Wang, L. Chen, A. Dikshit, J. Gustafson, B. Chen, M. J. Sax, J. Roesler, S. Blee-Goldman, B. Cadonna, A. Mehta, V. Madan, and J. Rao. Consistency and completeness: Rethinking distributed stream processing in apache kafka. In *Proceedings of the 2021 International Conference on Management of Data*, SIGMOD '21, page 2602–2613, New York, NY, USA, 2021. Association for Computing Machinery.

[74] S. Wang, J. Liagouris, R. Nishihara, P. Moritz, U. Misra, A. Tumanov, and I. Stoica. Lineage stash: fault tolerance off the critical path. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP '19, page 338–352, New York, NY, USA, 2019. Association for Computing Machinery.

[75] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica. Resilient distributed datasets: a fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, NSDI'12, page 2, USA, 2012. USENIX Association.

[76] H. Zhang, A. Cardoza, P. B. Chen, S. Angel, and V. Liu. Fault-tolerant and transactional stateful serverless workflows. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 1187–1204. USENIX Association, Nov. 2020.

[77] J. Zhang, K. Huang, T. Wang, and K. Lv. Skeena: Efficient and consistent cross-engine transactions. In *Proceedings of the 2022 International Conference on Management of Data*, SIGMOD '22, page 34–48, New York, NY, USA, 2022. Association for Computing Machinery.

[78] S. Zhuang, S. Wang, E. Liang, Y. Cheng, and I. Stoica. ExoFlow: A universal workflow system for Exactly-Once DAGs. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*, pages 269–286, Boston, MA, July 2023. USENIX Association.

A Artifact Appendix

Abstract

The artifact provides libDSE, our distributed speculative execution (DSE) framework, together with the speculative services and end-to-end benchmarks used in the paper. It contains the core libDSE runtime and coordinator (the `StateObject`/sthread programming model, message instrumentation, and the rollback-recovery protocol), the four speculative services built on the speculative-log pattern (a key-value store, a workflow engine, an event broker, and a two-phase-commit transactional store), and the benchmark drivers for the TravelReservations, EventProcessing, and TwoPhaseCommit workloads. It also includes the plotting scripts and the raw experiment results needed to regenerate every figure and table in the evaluation (Section 6).

Scope

The artifact substantiates the paper's central claim that DSE reduces end-to-end latency for persistence-bound cloud applications by up to an order of magnitude relative to non-speculative durable execution, while adding only modest runtime overhead. Concretely, it allows one to reproduce: the latency and scalability results for the travel reservation workflow against Temporal and a non-speculative baseline (Figure 8); the latency and storage-bandwidth results for event processing against DARQ (Figure 9); the commit-latency distributions for distributed transactions against Microsoft Orleans (Figure 10); the recovery behavior under injected and real failures (Figure 11, Figure 12); the instrumentation, threading, and coordinator microbenchmarks (Figure 13, Figure 14, Figure 15); and the engineering-effort summary (Table 2).

Contents

The artifact is organized as follows:

- **libDSE runtime and coordinator** — the core library (approximately 4000 lines of C#) implementing actions, sthreads, speculation barriers, message instrumentation, the dependency-graph protocol, and the stateless coordinator, with built-in gRPC and ASP.NET integration.
- **Speculative services** — the speculative log and the four services layered on it: a key-value store (built on FASTER), a workflow engine (built on DARQ), an event broker, and a 2PL+2PC transactional store.
- **Benchmarks** — drivers and workload generators for TravelReservations (DeathStarBench-style sagas), EventProcessing (DARQ search-trend-alert), and TwoPhaseCommit (a modified TPC-C), plus the baseline deployments (Temporal, DARQ, Orleans).
- **Figure reproduction** — `plots.py` regenerates all figures from the raw measurement files under `data/`.

Hosting

The artifact is publicly available and permanently archived on Zenodo at `doi.org/10.5281/zenodo.20059348` (version v2), which provides a self-contained snapshot of the libDSE implementation, the speculative services and benchmarks, and the experiment data. The implementation builds on the open-source FASTER project [18], hosted on GitHub at `github.com/microsoft/FASTER`. The artifact's `README` documents the build and run procedure.

Requirements

The end-to-end experiments were developed and tested on Microsoft Azure.

- **Cluster.** A managed Azure Kubernetes Service (AKS) cluster with a pool of 10 Standard_D8s_v3 nodes, each with attached premium locally-redundant-storage SSDs.
- **Microbenchmarks.** A pair of Standard_D32s_v3 machines (32 vCPUs, 128 GB RAM each).
- **Software.** The .NET SDK and C# toolchain, gRPC, and ASP.NET for the services; Python 3.12 with `matplotlib` for `plots.py`; and the Azure CLI (`az`) for `download.sh`. Reproducing the baselines additionally requires Temporal (over Azure-managed Cassandra and Azure Cosmos DB), DARQ, and Microsoft Orleans.

Reproducing the exact numbers requires comparable Azure hardware; the framework and microbenchmarks themselves build and run on a single multi-core machine.

TrainMover: An Interruption-Resilient Runtime for ML Training

ChonLam Lao[1,2], Jiaqi Gao[2,*], Jiamin Cao[2], Zhipeng Zhang[2], Pengcheng Zhang[2], Jiangfei Duan[2], Zhilong Zheng[2], Yu Guan[2], Yichi Xu[2], Yong Li[2],Zhengping Qian[2], Aditya Akella[3], Minlan Yu[1], Ennan Zhai[2], Dennis Cai[2], Jingren Zhou[2]

[1]*Harvard University,* [2]*Alibaba Group,* [3]*UT Austin*

Abstract

Large-scale ML training jobs are frequently interrupted by hardware and software anomalies, failures, and management events. Existing solutions like checkpoint-restart or runtime reconfiguration suffer from long downtimes and degraded performance. We present TrainMover, a resilient LLM training runtime that leverages elastic and standby machines to handle interruptions with minimal downtime and zero memory overhead. To achieve these goals, TrainMover introduces three key techniques: two-phase, delta-based communication group setup; communication-free sandboxed warmup; and general standby design that enables failure recovery from any role. Our evaluation shows that TrainMover consistently achieves around 20 seconds of downtime when handling various interruptions at the 1024-GPU scale. TrainMover is projected to reduce wasted GPU hours by 55% compared to the best alternative, saving 1.4 million GPU-hours per week at the 64K-GPU scale.

1 Introduction

Large Language Models (LLMs) have gained significant attention in recent years [6, 8, 31, 35, 36, 45, 46]. Scaling laws continue to guide the design and training of increasingly large models. LLM training jobs are typically deployed on parallel training frameworks (*e.g.,* Megatron-LM [40] and NeMo [22]), require tight coordination across thousands of GPUs, and run for weeks to months. For example, training GPT-3 with 175 billion parameters on 1,024 GPUs required approximately 34 days [30], while Llama 3 with 405 billion parameters was trained over 54 days using up to 16,000 H100 GPUs [17]. Meta and xAI are further pushing the training scale to 100K+ GPUs [41, 54].

This large-scale, long-running, and tightly coupled distributed training frequently encounters *interruptions*, including hardware anomalies, software contentions, network failures, and various management events. First, failures and anomalies become common as scale increases, and even a single component fault can slow down or halt the entire job. Alibaba reports that 60% of large-scale training jobs experience slowness from such issues, causing a 35% increase in average JCT [51], and the Llama 3 training job observed a mean-time-to-failure (MTTF) of only 2.7 hours [17]. Second, because training runs for weeks to months, maintenance tasks (repairs, security upgrades, patches, etc.) cannot be deferred; they often require rebooting servers or switches and thus interrupt training [2, 21]. Third, in shared clusters, operators need to frequently reschedule jobs or rebalance resources when high-priority jobs arrive [12, 53] or when fragmented resources become consolidated [47].

To preserve high ETTR (Effective Training Time Ratio), operators always keep a small pool of backup or elastic GPU machines (e.g., 6% in Alibaba [34], Bytedance [49], Google [42], Meta [17]) to replace affected machines while preserving the original training layout. Large-scale training jobs are highly tailored for peak performance and GPU memory utilization. For example, Meta [17] and Deepseek [9] tailor their model layouts to balance memory, computation, and communication for peak throughput (see §2.2). Therefore, even small layout changes can degrade performance, trigger out-of-memory errors, or leave GPUs idle [14].

Existing solutions for replacing affected machines fall into two broad categories. The first, and the current industry best practice, is the *stop–reschedule–reinitialize* approach [13, 18, 20, 48]: the job is stopped, faulty nodes are replaced with healthy ones from a backup pool, and the training framework re-initializes from the latest checkpoint. For a 16,000-GPU LLM job, interruptions accumulate to over an hour of downtime each day, translating to $86K in wasted costs [4]. The second category is *reconfiguration systems*, such as ReCycle [14], Oobleck [19], and Parcae [12], which can be retargeted to replace a machine by tolerating interruption drop (–1) and adding back healthy machine (+1) elastically without halting the job. However, a fundamental limitation persists across both categories: bringing back a new machine (*joiner*) online after each interruption requires reinitialization, which remains slow and forces the entire job to stall until the joiner resumes, leaving the critical path unchanged (see §2.3).

In this paper, we introduce TrainMover, an **interruption-resilient** LLM training runtime that allows a new joiner to prepare initialization in advance and overlap it with ongoing training, minimizing the interruption impact to the rest. TrainMover is carefully designed to avoid interfering with ongoing training and to incur zero additional GPU memory overhead during handling.

Achieving this is nontrivial. Modern LLM training stacks tightly couple initialization with globally synchronized communication setup, making it impossible to prepare joiners

*Jiaqi Gao is the corresponding author.

in advance without dedicated handling. Decoupling joiner initialization from this global path is essential. TrainMover leverages the two-phase communication-group setup (§5) and sandboxed shadow iterations (§4) to prepare communication and computation state in the background. The general standby design (§6) further enables role-agnostic preparation, allowing recovery regardless of which machine fails.

Executing shadow iteration on sandbox (§4): TrainMover introduces a communication-free sandbox that allows joiners to trigger framework initialization independently by running shadow training iterations in isolation before entering the main training loop, replacing actual communication with pre-recorded tensors. This decouples joiners from existing machines and significantly reduces transitional downtime.

Two-phase delta-based communication group setup (§5): TrainMover extends a two-phase delta-based communication group setup to allow all non–critical-path steps to be performed upfront (the first phase) and overlapped with ongoing training. In the second phase, TrainMover applies the membership changes in a delta manner, minimizing disruption to users and reducing transition downtime.

General standby to handle unexpected failures (§6): TrainMover leverages a general standby that can be immediately promoted to a joiner when a failure occurs, exploiting the high symmetry of distributed training. This symmetry removes the need for any prior role knowledge, allowing a single pre-warmed standby to recover any failed machine. The resulting performance is nearly identical to handling interruptions at a known, specific role.

For expected interruptions, TrainMover prepares the joiners in a sandbox (§4) and two-phase communication setup (§5), allowing the ongoing training job to continue uninterrupted and preserving performance during migration. Once the joiners are ready, TrainMover replaces the affected communication group members and synchronizes the latest model state from the leavers to the joiners before resuming training. For unexpected interruptions, TrainMover uses the general standby design (§6) to pre-warm the standby machine with the same techniques in the background, enabling rapid replacement of the affected machine when a failure occurs.

Our experiments show that TrainMover achieves a consistent downtime of around 20 seconds at the 1,024-GPU scale and is projected to reduce wasted GPU hours by 55% compared to other systems at the 64K-GPU scale. It also enables other use cases, such as resource load balancing at 10-minute intervals, with less than 3% training throughput loss.

2 Background and Motivation

2.1 Interruptions Prevail in LLM Training

LLM training jobs commonly span tens to hundreds of thousands of GPUs and run for weeks to months. Such clusters are expected to operate at peak efficiency to amortize their substantial infrastructure cost. However, sustaining this performance at scale is difficult, as numerous infrastructure and operational events can hinder or slow progress. We refer to these events collectively as **interruptions**, which include:

- **Hardware anomalies** such as GPU failures or down-voltage from overheating or power limits [17]. Aegis [10] reports that GPU servers exhibit failure rates orders of magnitude higher than general cloud nodes, and even within accelerator families, the H100's MTTF is twice that of the A100.
- **Software contentions** such as co-located programs running on the same CPU [51], OS noise, or garbage collection in the Python training framework [27]. Falcon [51] shows that such contentions from background tasks and network anomalies can slow LLM training by 34.59% on average.
- **Network failures** including optical module or switch failures and network congestion. Although network architects provision sufficient redundancy to maintain connectivity [34], Falcon [51] reports that 40–50% of large-scale LLM training jobs still experience network-induced fail-slows, with average slowdowns of 15–35% due to congestion or switch failures.
- **Management events** including driver updates [17], GPU resource balancing, and power or thermal planning [28]. First, bespoke training hardware adopts the latest techniques and therefore requires frequent maintenance, including bug fixes, urgent security patches, and firmware or driver updates [2]; the Llama-3 training team reported at least one such interruption per day [17]. Second, in shared clusters, operators often reassign GPU servers to improve locality and avoid network congestion [17, 34]. Third, to reduce power and thermal fluctuations and better balance resources, operators regularly redistribute workloads across the cluster [1, 17, 28].

Depending on their impact on the training job, these interruptions can be classified into two types: (1) *expected/planned interruptions*, such as scheduled maintenance for which operators are notified in advance, and software contentions or network failures in which the training job is still alive but not performant and waits for the operator to intervene; and (2) *unexpected interruptions*, such as GPU and CPU failures that crash the training framework instantly without notice[1].

2.2 The Goal: Sustaining High ETTR

These interruptions severely impact existing LLM training frameworks (*e.g.*, Megatron-LM [40] and DeepSpeed [39]) that rely on tight synchronization across all GPUs. Since each GPU must exchange intermediate results before every iteration[2], any slowdown propagates to the entire cluster, and a single failure terminates the job [51]. Alibaba [51] reports that slowness can increase large-scale LLM training completion time by up to 90%. ByteDance reports that 42.5% of

[1] This paper does not consider failures resulting from user-level bugs; ensuring application correctness in long-running jobs lies with the user.

[2] We do not handle asynchronous training frameworks because, to the best of our knowledge, no such framework is deployed in production at scale yet.

Restart Stages	Avg. Time	Percentage
Job Stop & Cleanup	0.52 min	7.69%
Job Reschedule	1.5 min	23.08%
Job Initialization	4.45 min	69.23%
• Checkpoint time	1.56 min	35.1%
• NCCL instantiation	1.09 min	24.5%
• Cold warmup	1.80 min	40.4%
Total	6.47 min	100%

Table 1: Average restart time breakdown for 8192-GPU training jobs.

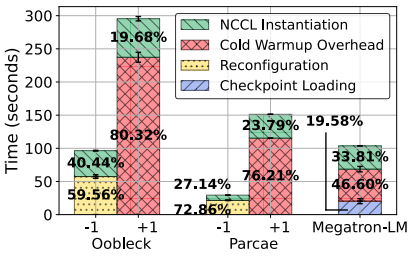


Figure 1: Reconfiguring and booting from scratch are both costly.

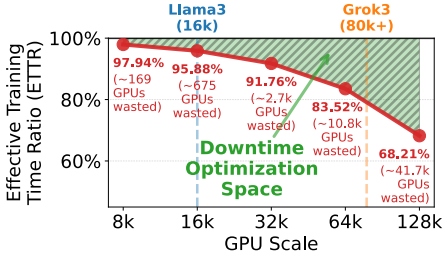


Figure 2: Impact of downtime on ETTR at different scales.

training jobs are at least 10% slower due to stragglers [27]. Meta reports that the ETTR drops to 0.6 when a job spans 8192 GPUs [21]. This wasted time stems from re-training lost progress since the last checkpoint, restart overhead, and checkpoint-saving overhead. Such longer training times translate directly into higher infrastructure costs. For example, based on AWS's public pricing, training with 16K GPUs costs $1.44M per day [4]. An ETTR of 0.6 translates to roughly $0.58M of wasted cost per day. Therefore, when an interruption happens, the highest priority is to restore the training job to its peak performance.

In practice, this goal translates to *quickly replacing interrupted GPUs with healthy ones while maintaining the training layout* (i.e., the parallelism configuration and model partitioning tuned for the job). The training layout is specially optimized from the ground up to achieve peak performance on the specific scale. For example, before training starts, Meta [17] manually fine-tunes the model partitions on each GPU to balance memory and computation load across different pipeline parallelism stages to achieve the highest throughput. Similarly, DeepSeek [9] carefully tunes the GPU SMs dedicated to computation and communication to achieve the optimal balance and trains DeepSeek-V3 at peak performance. Re-distributing the workload to the remaining machines can easily lead to inferior performance, job failures due to unexpected code behavior or out-of-memory errors, or idle GPUs in scenarios where an entire DP group is removed when the distributed optimizer is disabled.

Under such constraints, naively setting up the training cluster to match the job size is undesirable, as a single broken machine can undermine the entire training run. Instead, operators reserve a pool of healthy standby servers in the cluster for on-the-fly replacement to reduce downtime. For example, ByteDance [49] allocates warm-standby pools based on the 99th percentile of historical GPU failure rates. Alibaba's HPN [34] reserves 6% of its GPUs as backup in each segment. Google [42] maintains standby TPU cubes within each SuperPod to support rolling maintenance and recovery. Llama-3 [17] was trained on 16K GPUs within a 24K-GPU cluster. More discussion on the economic cost of standby machines can be found in §9.

2.3 Limitation on Existing Solutions

The industry has deployed automatic systems [10, 49] to reduce interruption downtime. Yet, as training scales grow, recovery in the training stack quickly becomes the bottleneck. We evaluate two representative interruption-handling solutions. Note that, across all experiments, we assume instant fault localization and isolation.

Solution 1: Restart. Restart refers to the standard stop–reschedule–reinitialize mechanism used in production to handle interruptions [10, 17, 21, 51]. When an interruption occurs, the system first *stops* the job by terminating the training framework and cleaning up active processes and connections. It then *reschedules* the cluster by offlining faulty nodes, selecting healthy replacements, and preparing their runtime environments. Finally, the training framework *re-initializes* from the most recent checkpoint before training can resume.

This process introduces substantial overhead. We consulted one of the largest cloud providers, and even with an automated recovery stack similar to that of Llama-3 [17], an 8,192-GPU production job still incurs 6.47 minutes of delay per interruption (Table 1). Excluding infrastructure overhead, framework initialization alone accounts for 4.45 minutes: (1) **Checkpoint loading** (35.1%), where model and optimizer states are restored from remote storage. This cost grows rapidly with model size [48, 50, 51]. (2) **NCCL instantiation** (24.5%), where each parallelism dimension (DP, PP, TP) must form its own NCCL group, requiring synchronized metadata exchange and connection setup. (3) **Cold warm-up** (40.4%), including CUDA context creation, GPU memory allocation, JIT compilation, and data-loader initialization. Since large-scale training is fully synchronized, an interruption affecting only a few nodes [10, 21] still forces *all* machines to restart, significantly amplifying downtime.

Solution 2: Runtime Reconfiguration. Recent works such as Oobleck [19], Parcae [12], and ReCycle [14] aim to avoid global restarts during interruptions by elastically scaling the training cluster up or down without halting the job. These systems adjust training configurations—such as batch sizes or parallelism schemes—to sustain reduced training throughput during failures. When an interruption happens, the system can remove the affected machine (`-1`) and continue training at reduced throughput, then later add a replacement machine once available (`+1`).

However, despite avoiding infrastructure overhead, the core restart *critical path* remains. Figure 1 shows the latency for Oobleck and Parcae when handling interruptions in a 6.7B GPT training job on 32 GPUs. Removing a machine (`-1`)

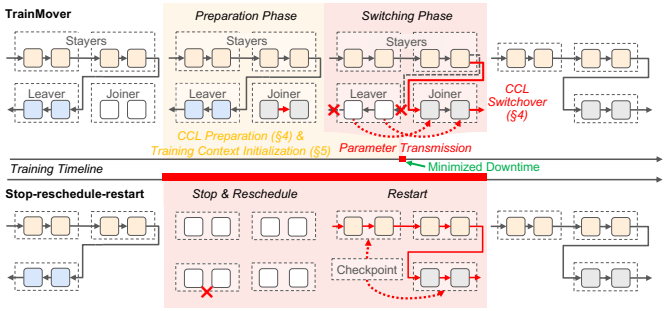


Figure 3: TrainMover Overview

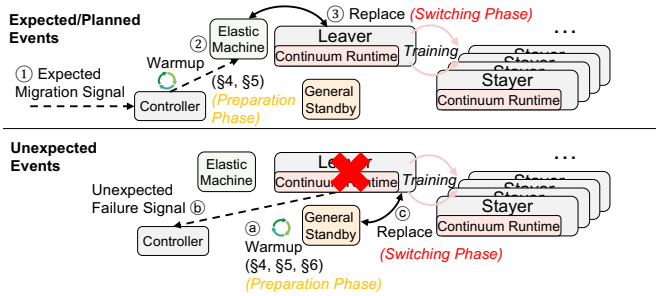


Figure 4: TrainMover Workflow

requires all nodes to redistribute parameters, re-instantiate model components, and reallocate GPU memory to support the new configuration—taking 57 seconds in Oobleck and 21 seconds in Parcae. Adding a machine (+1) is even more expensive: the joiner must initialize NCCL groups and reinitialize software and hardware components, taking over 100 seconds in Oobleck and more than 200 seconds in Parcae. The asymmetry arises because existing machines are already warm—their CUDA kernels are compiled and GPU memory is allocated—while a newly added joiner must complete a full cold warm-up from scratch. Oobleck is built atop ColossalAI [24], while Parcae is built atop DeepSpeed [39], leading to different framework warm-up times. Remarkably, this entire cost exceeds even the job initialization overhead in Megatron-LM at the same scale (rightmost bar, around 100 seconds).

In short, although infrastructure overhead can be reduced, *the heavy initialization required to bring a joiner online remains unavoidable*. While this downtime is tolerable at small scales, growing cluster sizes dramatically amplify its impact: higher interruption frequencies and larger numbers of affected GPUs cause downtime to quickly become the dominant bottleneck, substantially degrading ETTR.

This scaling effect is already evident in practical measurements. Although a 4.45-minute restart time appears modest, Figure 2 shows the resulting ETTR when this measurement is combined with the MTTFs reported by Meta [21] across increasing cluster sizes. As the figure illustrates, effective training throughput collapses at production-scale deployments (64K/128K GPUs) [41, 49, 54], resulting in 16.47% and 31.79% throughput losses, translating to $0.95M and $3.63M of daily monetary losses, respectively.

3 TrainMover Overview

To break through this scalability barrier, we design *TrainMover*, a resilient LLM training runtime for production-scale clusters that sustains high ETTR under frequent interruptions.

TrainMover migrates workloads only from interruption-affected *leavers* to healthy backup *joiners*, leaving the remaining *stayers* unchanged. By strategically leveraging readily available machines and overlapping all preparable work ahead of time, TrainMover moves most job-setup procedures off the critical path before the joiner participates in training.

This design enables a carefully orchestrated migration procedure that achieves 10× lower downtime for both expected and unexpected interruptions, without any additional GPU memory overhead, thereby avoiding the risk of out-of-memory crashes during migration.

As shown in Figure 3, instead of long restarting (bottom), TrainMover (top) splits the migration into two phases: in the *preparation phase*, TrainMover prepares the joiners' training stack in the background without affecting the foreground training job; once the *joiners* are ready and the foreground job finishes the current iteration, the system enters the *switching phase* and synchronizes each *joiner* to the latest training state before taking over the *leaver*. Each joiner takes over the leaver's role, keeping the training layout unchanged. Figure 4 illustrates TrainMover's workflow of handling both expected and unexpected interruptions as described below.

Expected Interruptions. Expected interruptions—such as scheduled maintenance—provide advance notice. When such an event is planned, the controller issues an expected-migration signal ①. TrainMover then selects elastic machines as *joiners* and immediately places them into a background *preparation phase* ②, while *stayers* and *leavers* continue training normally. Once preparation completes, TrainMover triggers a brief *switching phase* ③, during which the joiner receives the latest training state one-to-one from its paired leaver and takes over the leaver's role. This switching phase contributes to the total training pause downtime.

Unexpected Interruptions. Unexpected failures occur without prior notice, so a warmed standby machine must remain in the background *preparation phase* ⓐ. When an unexpected failure is detected ⓑ, the controller initiates recovery: TrainMover immediately promotes the standby machine to a joiner and transitions directly into the *switching phase* ⓒ, where the joiner retrieves the up-to-date state from a checkpoint, and the other machines roll back to the same checkpoint [13, 49, 50][3]. Once the switching phase completes, the *leavers* exit and the *joiners* replace their roles, resuming training seamlessly. Checkpoint loading only occurs as a fallback when no in-memory copy is available, in the switching phase.

TrainMover shifts as much work as possible off the critical

[3] If such a per-iteration checkpointing system is unavailable, other machines must restore their state from remote checkpoint storage.

path into the *preparation phase* while keeping the *switching phase* as lightweight as possible. However, several inherent properties of large-scale training fundamentally limit how much work can be shifted, creating key challenges for minimizing downtime.

(§4) Initialization is implicit and tightly coupled. A joiner must complete many initialization steps before starting training, yet modern frameworks trigger these steps implicitly across different levels (e.g., PyTorch, NCCL, and CUDA) and often require global coordination. These heavy behaviors prevent initialization from being invoked independently; performing them in advance requires untangling the tightly coupled procedures and understanding the implicit steps. TrainMover addresses this with a *communication-free sandbox* (§4): rather than untangling each implicit initialization path, the joiner simply runs a full shadow training iteration in isolation, so all computational initialization is triggered naturally—without blocking ongoing training or requiring assistance from active machines.

(§5) Communication groups assume static membership. Collective communication libraries are designed around fixed group membership. Replacing even a single machine typically requires globally tearing down and rebuilding all communication groups—a slow, synchronized procedure that dominates the interruption-handling critical path. TrainMover addresses this with a *two-phase delta-based CCL setup* (§5) that decouples CCL setup and performs all required setup during the preparation phase, applying only fast, delta connection-level changes when the membership switch actually occurs—with no additional GPU memory overhead.

(§6) Standby resources must cover unexpected failures. Unexpected failures can occur on any machine and in any training role, making advance preparation impossible. Warming up standbys for each role is wasteful at scale, while relying solely on elastic machines acquired after a failure introduces large critical path delays. Standby resources must therefore be role-agnostic and able to recover swiftly when failures occur. TrainMover solves this with a *general standby* design (§6) that exploits the structural symmetry of LLM training ranks: a single standby sequentially warms up for each pipeline role type, making it a universal recovery target without dedicating a separate standby per role.

These three techniques compose directly with the two-phase workflow above. At the start of training, hot standby machines are pre-deployed (§6) ⓐ to guard against unexpected interruptions; each runs the communication-free sandbox (§4) and Phase 1 CCL setup (§5) upfront, so upon a failure it enters the *switching phase* directly. For expected interruptions, upon receiving the migration signal ①, a new machine (no pre-deployed hot standby necessary) can serve as the joiner and undergo the same preparation (§4, §5), overlapping with ongoing training. In both cases, the *switching phase* ③/ⓒ enters CCL Phase 2 to switch only the delta inter-machine connections (§5) and transfer the state from the leavers to the joiners, keeping downtime to seconds.

4 Sandboxed Initialization

The joiner should initialize independently and in advance during the *preparation phase* to avoid global blocking; otherwise, all machines must pause until the joiner completes its setup. Yet, training initialization spans both cross-machine and intra-machine dependencies, making it difficult to identify and trigger initialization gracefully and independently (§4.1).

To address this, we propose a *communication-free sandbox* (§4.2 and §4.3) that allows joiners to execute shadow iterations in isolation to trigger initialization ahead of time. Afterward, the joiner enters the main loop and seamlessly continues training.

4.1 Prolonged and Complex Training Initialization

Training initialization, once overlooked as a one-time cost, has become a significant source of inefficiency as training scales grow and interruptions become more frequent. Each interruption must redo warm-up, wasting GPU time. In the GPT-10B experiment shown in Table 2, excluding NCCL instantiation, a single interruption triggers about 150 seconds of warm-up before reaching stable performance. The first iteration is also much slower—about 6× longer (around 44 seconds in our GPT-10B experiment)—due to just-in-time (JIT) compilation and cascading initialization dependencies—in pipeline-parallel training, for example, each stage depends on the completion of the P2P communication of the previous stage, so initialization proceeds sequentially across stages, further amplifying the startup overhead.

Initialization is complex and spans multiple layers of the training stack. While frameworks like JAX provide an eager mode to explicitly trigger initialization, mainstream training frameworks such as Megatron-LM, DeepSpeed, and NeMo rely on hardware-aware optimizations (e.g., memory-layout specialization and fused kernels) that are only activated when real data arrives (e.g., the first iteration). Furthermore, initialization is intertwined with communication, as exemplified by the pipeline-parallelism case discussed above. Addressing these behaviors would require uncovering and manually invoking many hidden initialization paths or thoroughly examining user training code. We need a mechanism that automatically triggers these complex initialization behaviors without requiring user intervention.

4.2 Shadow Iteration in Communication-free Sandbox

To pre-trigger initialization without any other machines' assistance, we create a *communication-free sandbox* that allows joiners to run a shadow iteration independently before joining the main training loop. This sandbox eliminates the need for case-by-case initialization handling: joiners can execute all implicit and framework-specific initialization routines in one shot, without relying on distributed communication or requiring any assumptions about user code or framework behavior.

This design introduces two key requirements: first, the shadow iteration must execute with valid state values and

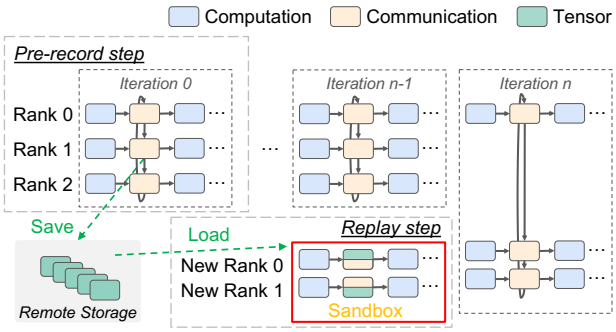


Figure 5: Sandbox initialization workflow

communication results to ensure the program does not crash and that initialization is correctly triggered—using dummy or zero values can easily cause NaNs or assertion errors. Second, communication operations must be handled locally without depending on communication responses from other machines. We address both requirements through valid tensor record–replay, as illustrated in Figure 5.

The sandbox warm-up has two steps: *pre-record* and *replay*. In the pre-record step, the designated recording machines capture one or more valid iterations and store the resulting communication outputs in advance. When a migration signal is received or a standby becomes available, the joiners enter the sandbox, where each joiner runs the replay step—reusing the recorded communication results to trigger its initialization.

Shield the Warmup — Pre-record step. We build a record hooking layer that sits between PyTorch and the CCL to intercept communication calls issued by the training framework. At the start of a new job as shown in Figure 5, designated machines—including Rank 0, Rank 1, and Rank 2—enter the pre-record step to prepare for potential migration. Our hook monitors and records the output tensors of all collective calls (e.g., all-reduce) to persistent storage once the collective completes. They are later replayed in the sandbox to ensure valid training states on the joiner.

This one-time pre-record step runs only during the first training iteration (or a few iterations) at the very beginning. After that, the interception hook is removed, and training proceeds as normal, introducing no additional overhead for the remainder of the training process regardless of the number of interruptions experienced or number of job restarts.

Run the Warmup — Replay step. When a migration signal is received or a standby is available (*the preparation phase*), all joiners (new Rank 0 and new Rank 1) enter the *replay step* and are placed into a sandbox, isolating them from the ongoing training in preparation for triggering initialization before joining the main training process. First, each sandboxed joiner pulls the initial state from the checkpoint. After loading the state, the joiner proceeds to a shadow iteration to trigger initialization. Our sandbox is designed to be *communication-free*, specifically to avoid any interaction with existing training machines. We deliberately prevent any assistance from active participants—such as responding to collective operations or sending state information—to ensure that the joiner triggers initialization entirely on its own.

We attach another hook to the PyTorch communication layer to intercept communication calls made during the replay step. Any communication call that attempts to reach outside the sandbox is intercepted and instead served with previously recorded tensors as responses. Certain operations—such as `barrier` and `send`—are safely bypassed, as invoking these operations does not affect the caller's state.

After the shadow iteration is complete, the joiners (e.g., New Rank 0 and New Rank 1) can replace the leavers (e.g., Rank 0 and Rank 1) at the end of iteration $n-1$ once they have received the up-to-date states in the *switching phase*, enabling a machine transition with no initialization downtime after migration.

Robustness and Correctness. The sandbox warm-up relies on a degree of determinism in the ML framework that modern LLM training frameworks satisfy. By design—fixed kernel schedules, static memory layouts, and deterministic collective ordering are maintained for reproducibility and peak performance. As a result, almost all initialization—including data-loader setup, CUDA context creation, and memory allocation—is correctly triggered during sandbox warm-up. Only in rare cases does a component depend on runtime data or require CUDA graph capture whose trigger may be missing. MoE routing is one such case: token routing is input-dependent and varies across iterations. However, in practice this has minimal impact—MoE frameworks (e.g., Megatron-LM) support preallocating fixed-size expert buffers regardless of the actual routing decision, so the underlying memory layout and communication structure remain static. Any initialization not triggered during sandbox warm-up simply completes during the *switching phase*, introducing only negligible overhead (*ms-level*).

Also, correctness is always guaranteed because the leaver's latest model and optimizer states will eventually overwrite all local states after sandbox warm-up in the *switching phase*.

4.3 Low Overhead Record and Replay

Recording and replaying tensors incur a one-time overhead in both storage and tensor loading time. However, not all communication needs to be recorded or replayed— each recording is essentially *a preparation for communication edges that might be temporarily missing during sandbox warmup*. We can reduce the recording scope by focusing only on what might be needed. For example, since migrations happen only at the machine level, intra-machine communication (where intensive TP traffic is always involved) can run natively inside the sandbox without record. We further eliminate redundant recordings for duplicated training roles, as discussed in §6. The storage overhead can be eventually reduced to under 300 GB in a GPT 5.12T MoE model setting.

During replay, as illustrated in the lower part of Figure 5, only the communication that touches the sandbox boundaries in the training graph relevant to the new joiner needs to be

NCCL Setup Components		
Network bootstrap		2.48s (4.92%)
Topology discovery and computation		9.40s (18.63%)
Connection Establishment	Intra-machine	21.49s (42.59%)
	Inter-machine	17.07s (33.86%)

Table 2: Time breakdown aggregated over all NCCL groups.

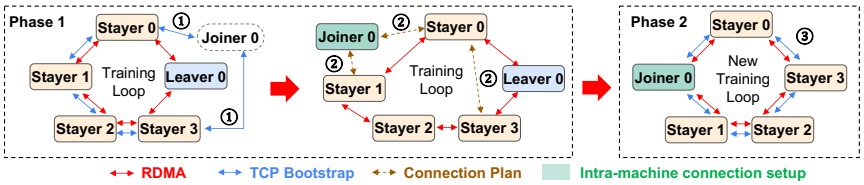


Figure 6: Two-phase CCL Migration workflow

loaded and replayed. When multiple machines are migrated together as a batch (e.g., an entire PP group), the joiners' internal communication can remain real, while only cross-boundary communication is loaded from storage. This boundary-aware record-and-replay design contributes to low I/O overhead.

5 Dynamic and Lightweight Communication group setup

Migrating one machine to another requires updating the membership of the existing CCL groups. However, CCL groups lack flexibility and are costly—they take time to create, and once created, their membership cannot be modified (§5.1). Existing approaches [12, 19] rely on globally destroying and recreating groups, which introduces unacceptable delays. An alternative strategy is to concurrently pre-allocate new groups in the background to avoid blocking, but this incurs additional GPU memory overhead. To solve this downtime–memory dilemma, we introduce a two-phase asynchronous CCL setup (§5.2) that enables dynamic group membership updates with minimal downtime and zero additional GPU memory overhead.

5.1 Today's Slow and Static CCL Setup

Communication setup is *slow*. The slowness stems from three main procedures: (1) *Network bootstrap* — participants form a TCP network through multi-round handshakes and consistency checks, which is expensive at scale due to the need for global coordination [41]; (2) *Topology discovery* — each node gathers local device metadata (e.g., NIC bandwidth) and exchanges it via ring all-gather to compute global topology and determine communication neighbors, incurring high synchronization overhead; and (3) *Connection establishment* — nodes allocate GPU buffers and set up inter- and intra-machine connections (e.g., GPU Direct RDMA, NVLink) based on the computed topology. These setup procedures can still be slow despite high parallelism, due to their inherent synchronization behaviors.

Table 2 shows the CCL setup overhead on an 8-machine cluster with 64×40GB A100 GPUs, training a GPT-10B model with TP=4, PP=2, and DP=8. The setup takes around 50 seconds due to the overhead of establishing and managing communication groups—a challenge that grows with scale. For example, training across 1,000 machines results in a connection count that scales with 1,000 × (# of CCL groups) × (# of channels inside each group). The number of CCL groups depends on the communication dimensions (e.g., DP, PP, TP); even in this modest 3D setup, each host participates in 7 groups. High-bandwidth links (e.g., 400 Gbps) further increase the number of required channels. Although many setup operations can run in parallel, the scalability of these massive numbers of connections is ultimately limited by global synchronization barriers, straggler effects from heterogeneous network latencies, and driver-level contention under high concurrency (e.g., CUDA and IB verbs serialization), causing setup time to grow with cluster size.

Communication setup is *static*. CCL connections are tightly coupled to hardware (e.g., NVLink, GPU Direct RDMA), requiring memory mappings (via CUDA IPC) and optimized connectivity topology computation—all constructed at initialization. These components are specific to the participating ranks and GPUs, so any group member change requires structural updates and may yield suboptimal performance if not carefully handled.

5.2 Two-phase Delta-based CCL Setup

We design a dynamic and lightweight CCL setup that minimizes transition overhead when a joiner migrates into an ongoing training job. Our two-phase design shifts most of the setup to Phase 1, where it is performed in advance and overlapped with training, deferring only minimal connection establishment to Phase 2 to reduce downtime. Additionally, we enable calculating and reestablishing only the delta connections between the old and new topology, avoiding the need to globally reconnect all links.

We demonstrate the migration workflow with four stayers (0–3), one joiner, and one leaver in Figure 6. Migration begins with a signal and role assignment from the controller. In Phase 1, Stayers 0–3 invoke `CCL_prepare_stayers(new_topology)` to incrementally modify their existing CCL group in preparation for the new setup with the joiner. This function also enables topology-aware configurations provided by users as input. Meanwhile, the joiner concurrently calls `CCL_prepare_joiners()`. During this process, the original group's communication remains functional ①. Stayers reuse existing TCP bootstrap connections and initiate the bootstrap process with Joiner 0. Since the additional TCP connections and topology state reside in CPU memory, this setup overlaps with training and incurs no GPU memory overhead. Once connected, the joiner and stayers exchange the necessary topology and state information.

After exchanging topology information, a recalculation of connectivity is required, as the original topology may no longer be optimal after migration ② (e.g., Joiner 0). Once the resilience runtime receives the new topology information, all participants locally compute a *delta topology*, identify the

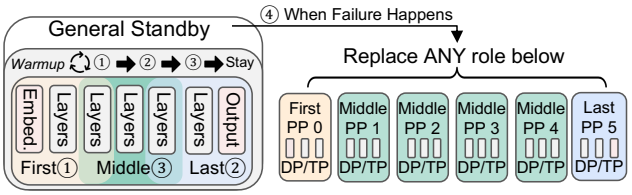


Figure 7: General Standby

minimal set of `channels` required for the updated configuration, and generate a delta reconfiguration plan that determines how to modify the necessary channels. Since all changes occur at the channel level, the design remains compatible with diverse topologies and communication techniques, such as hierarchical connectivity and SHARP [16].

The reconfiguration plan specifies which `channels` need to be updated. For `channels` that do not require replacement, they can be directly inherited and reused. Intra-machine channels (e.g., NVLink) are typically fully inherited, as they provide the fastest and highest-bandwidth paths and usually remain unchanged during migration, which operates at the granularity of entire machines. Meanwhile, the joiner (Joiner 0) begins setting up channels that can be initialized locally (e.g., intra-machine connections). If there are multiple joiners that need to connect with each other, their inter-connections are also established accordingly.

Delta `channel` connections in the reconfiguration plan—typically inter-connections—are intentionally deferred to the second phase. At this point, intra-machine communication is already available through inheritance, while inter-machine connections have not yet been established on the stayer machines—ensuring no additional GPU memory is consumed. Afterward, both the stayer and joiner CCL groups enter the *ready-to-switchout* state. Notably, the stayer and the leaver continue training without interruption during this phase.

Once both the joiners (e.g., Joiner 0) and impacted stayers (e.g., Stayer 0 and Stayer 3) complete Phase 1 and enter the *ready-to-switchout* state, they invoke the second API, `CCL_switchover()`, to switch delta inter-machine connections from stayer–leaver to stayer–joiner. For instance, the connection between Stayer 0 and Leaver 0 is replaced by one between Stayer 0 and Joiner 0; likewise, Stayer 3–Leaver 0 is switched to Stayer 3–Joiner 0. The replacement happens at the RDMA queue pair (QP) level, where it re-establishes the inter-machine peers, transitions the CCL group from *ready-to-switchout* to the normal state, and reclaims the resources tied to the old topology ③. This delta operation is the sole contributor to network downtime. The entire procedure incurs zero memory overhead; a detailed workflow can be found in §8.5.

6 General Standby

Failures can occur unexpectedly at any time. A key challenge in handling such events is that the standby machine does not know in advance which machine it should prepare for, as each machine holds different parallelism roles (e.g., TP and DP IDs). We design a *general standby* mechanism that prepares the standby for *any* rank by leveraging the symmetry of training ranks and the fact that warm-up requires only valid states, independent of role. This design enables both sandboxed initialization (§4) and two-phase CCL warm-up (§5) in unexpected failure cases and significantly reduces the number of standby machines needed.

6.1 Symmetry on Training Ranks

LLM architectures exhibit strong symmetry, as maintaining identical model partitions across all machines helps achieve peak synchronous training performance. Parallel methods such as tensor parallelism (TP), data parallelism (DP), expert parallelism (EP), and other structurally symmetric forms of parallelism maintain these identical partitions, which share the same parameter shapes and optimizer-state layouts; for example, running a DP+TP training job in Megatron-LM produces identical memory utilization across all ranks. Each rank executes the same set of computational operators and CUDA kernels[4], and participates in the same collective communication patterns. This strong, correctness-driven symmetry allows the standby to serve as a universal recovery target.

Pipeline parallelism (PP) is an exception to the model's inherent symmetry: each PP stage contains a subset of layers, while the first and last stages include additional layers such as the `word_embedding_layer` in the *first stage* and the `output_layer` in the *last stage*. Despite these variations, the first and last stages remain similar in structure: the `word_embedding_layer` and `output_layer` share the same shape (e.g., weight tying). The middle stages consist of repeated transformer blocks and are kept symmetric for optimal performance . In total, PP introduces at most three distinct role types—the first stage, the middle stages, and the last stage—yet the overall partitioning remains highly symmetric.

6.2 General Warm-Up Procedure

The symmetric design allows the standby machine to warm up and recover any role in the system. Figure 7 illustrates the warm-up procedure. To prepare it, the standby runs warm-up iterations for initialization (§4). Without pipeline parallelism (PP), a single iteration is sufficient because all machines share identical roles. When PP is enabled, the standby sequentially and independently runs up to three warm-up iterations ①②③, each corresponding to a different role—the *first stage*, *last stage*, and *middle stage*. Running each role during warm-up concretely initializes all role-specific components, such as JIT-compiled fused kernels.

At this stage, the general standby remains in Phase 1 of the CCL setup, where inter-machine connections are not yet established. Its overall GPU memory footprint stays hundreds of megabytes below peak usage (detailed measurements

[4]Although some techniques may introduce different execution orders due to scheduling patterns (e.g., VPP), the underlying set of operators remains the same.

in §8.5). This available headroom easily accommodates the GPU-resident artifacts produced during warm-up across all role types—including JIT-compiled kernels and other compiled GPU artifacts specific to each role's layers—which add at most a few hundred kilobytes per role, never exceeding the original memory budget.

After warm-up, we retain the middle-stage role because it represents the majority of pipeline instances. If a failure occurs on a middle stage, the standby can replace it directly. For failures in the first or last stage, TrainMover updates only the small delta of layers unique to those stages. This delta is negligible because only the memory for the optimizer states and parameters of the additional layers (e.g., embedding or output layers) needs to be allocated. Other artifacts, such as all precompiled kernels, are already initialized. Even at large scale, the standby needs to run only these three role types.

Two-phase CCL preparation (§5) is also enabled by symmetry, because each machine exposes an identical intra-node and inter-node topology. For example, if a CCL group contains two separate TP groups within a machine, the same configuration must exist on all other machines. This eliminates the need for dynamic topology specialization during recovery. The only exception arises with pipeline parallelism (PP): when PP spans multiple machines, the standby must additionally establish a few connections with its neighboring stages after recovery, according to its assigned role.

Leveraging this symmetry, we can prepare a general standby for unexpected failures at the beginning of the training job in parallel, with a deployment time no greater than that of a normal initialization. The general standby achieves downtime nearly identical to that of the switching phase in an expected event, as evaluated in §8.3. A discussion of the cost-effectiveness of the general standby can be found in §9.

The number of standbys needed is small in practice (§8.2): the probability of two machines failing simultaneously is the product of their individual failure rates ($\approx 1/\text{MTTF}$), making concurrent failures extremely unlikely [18, 49, 55]. TrainMover also supports the no-standby case and still recovers via its overlapped recovery path (§7). In practice, 1–2 standbys cover all distinct stage types (first, middle, last) for PP configurations; operators who wish to overprovision can assign dedicated standbys per stage proportion, e.g., with PP degree 8 and a 1:6:1 stage ratio, allocating more standbys to the dominant middle stages.

7 Implementation

TrainMover is implemented in Python and C/C++, comprising two main components: the training-node resilience runtime and the controller. TrainMover comprises about 12K lines of new code: 1.9K LoC of Python for the controller, 7.8K LoC of Python for the Megatron-LM runtime extensions, 1.1K LoC of C++/Python for the PyTorch `c10d` extensions, and 1.5K LoC of C/C++ for the NCCL extensions. The prototype is open-sourced on GitHub [43].

Training Node Resilience Runtime and Controller. The TrainMover runtime builds on Megatron-LM with focused modifications to Megatron-LM, PyTorch, and the CCL layer to support resilient training. In Megatron-LM, it manages interruption signals and preserves CCL ordering during overlapping training and migration. In PyTorch's `c10d`, we enable multiple global CCL groups, support the CCL layer's two-phase initialization API, and add an interception layer that records, replays, or bypasses tensors to facilitate sandbox warm-up. In the CCL layer, we implement the new APIs and demonstrate them in NCCL. The controller coordinates the system by assigning roles, initiating migrations, detecting failures, and distributing topology metadata; each node's runtime maintains a control channel with the controller to keep metadata synchronized and ensure joiners, leavers, and stayers correctly configure their CCL connections.

State Synchronization. Concurrently with CCL switchover, the joiner receives the up-to-date training state before resuming. For expected events, the *leavers* transmit the latest training states—such as model parameters and optimizer states—to their corresponding *joiners*, overwriting the sandbox-warmed state and bringing the joiner fully up to date with the current training progress. For unexpected failures, the *joiners* retrieve the required states based on the redundancy configuration. If a copy is available on other *stayers*—through DP group redundancy or remote CPU memory checkpointing [50]—TrainMover directly queries the states from a *stayer* machine that maintains a copy. Otherwise, in the absence of redundancy—such as with the distributed optimizer [3] or ZeRO optimizer [37, 38]—the *joiners* fetch the data from remote storage checkpoints [12, 48]. The *leaver–joiner* mappings are one-to-one, meaning that state transmission and connection establishment for different *joiners* occur independently and in parallel—the overhead does not grow with cluster size. The transfer is feasible within the ∼20-second recovery window because modern GPU memory is bounded (e.g., 80 GB per A100), and the state is transmitted via high-bandwidth RDMA directly from a neighboring stayer's GPU or CPU memory, achieving transfer rates of hundreds of GB/s, well within the recovery budget.

8 Evaluation

Our experiments show that TrainMover scales effectively, with about 20 seconds of downtime at 1024 GPUs and over 55% less wasted GPU time at 64K GPUs compared to the best baseline (§8.2). We further analyze event downtime (§8.3) and additional use cases (§8.4), where TrainMover sustains 97% training efficiency during periodic 10-minute load rebalancing at the 1024-GPU scale. A detailed breakdown of the designs and the *zero* memory overhead are shown in §8.5.

8.1 Experiment Setup

Testbed and Model Settings. We conduct our experiments on a 1024-GPU testbed and evaluate a range of models—including GPT-5.12T MoE, GPT-175B, GPT-39.1B [25],

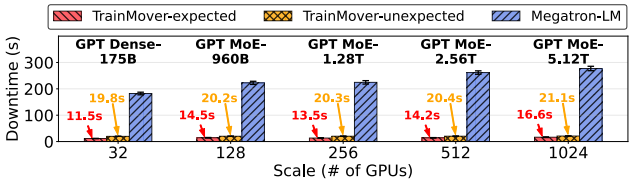


Figure 8: Downtime comparison across different GPU scales.

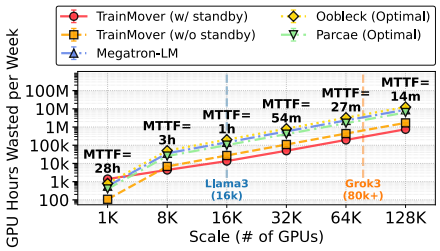


Figure 9: Projected GPU hour wasted in large scale deployment

GPT-20B [50], GPT-Medium [7], GPT-2.7B [7] and others—to cover diverse model sizes at different scales. Large-scale model configurations are summarized in Table 3. Detailed downtime analysis is additionally conducted at the 32-GPU scale to enable fair comparison with Oobleck and Parcae, which cannot scale beyond 32 GPUs (e.g., Oobleck's template construction time becomes prohibitively long at larger scales). For these experiments, models are tested with various TP, DP, and PP configurations including (TP1, PP8, DP3), (TP4, PP8, DP3), and (TP8, PP8, DP3). Default profiles are: GPT-Medium and GPT-2.7B use TP1, PP8, DP3 with a global batch size of 96 and microbatch size of 2; GPT-20B uses TP1, PP8, DP3 with the distributed optimizer, global batch size of 36, and microbatch size of 1; GPT-39.1B uses TP4, PP2, DP3 with the distributed optimizer, global batch size of 36, and microbatch size of 1. The Wikitext dataset [29] is used, following prior work [12, 19].

Baseline. Our primary baselines fall into two categories: the stop-and-restart approach, represented by Megatron-LM, and reconfiguration-based approaches, represented by Oobleck and Parcae. For expected events, since Oobleck and Parcae do not natively support live migration, we perform it by removing a node (-1) and adding a new one (+1). Parcae does not support tensor parallelism, and neither Oobleck nor Parcae supports distributed optimizers [37], as both rely on redundancy within data parallelism. We include various 32-GPU experiments because Oobleck cannot scale due to its prolonged template-generation overhead.

Metrics. We primarily use three metrics in our evaluation. The first is *downtime*, which measures the duration of interruption handling for both expected events and unexpected failures, reflecting the additional system time required to complete training per interruption. The second is *GPU hours wasted per week*, an end-to-end metric that quantifies the total GPU time lost during event handling—essentially overhead expressed as an absolute resource cost, which can be directly translated into training capital cost. The third one is *ETTR*.

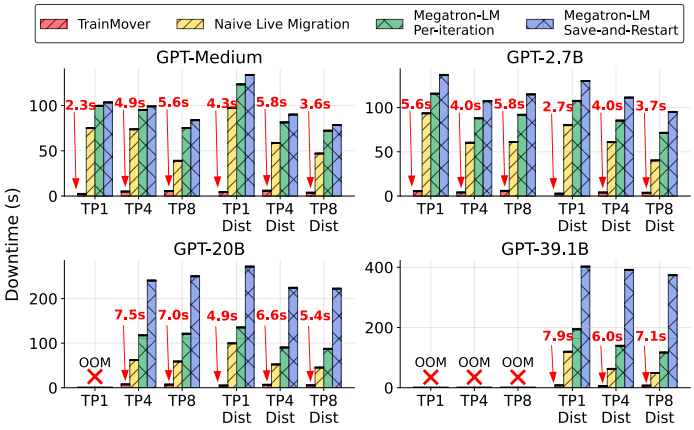


Figure 10: Migration downtime with different models and parallel settings

# GPU	32	128	256	512	1024
Type	Dense	MoE	MoE	MoE	MoE
Model	175B	960B	1.28T	2.56T	5.12T
TP	4	8	8	8	8
PP	8	8	8	8	8
EP	/	2	4	8	16

Table 3: Large-scale GPT model configurations used in experiments.

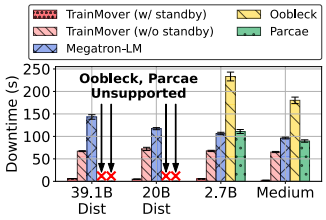


Figure 11: Unexpected failure downtime

8.2 End-to-end Experiments

TrainMover Performance at Scale. We evaluate downtime for both expected and unexpected interruptions on a large-scale testbed with up to 1024 GPUs, demonstrating TrainMover's scalability and low downtime in Figure 8. In TrainMover, migration downtime during expected events consistently remains below 20 seconds, while downtime for unexpected failures is also below 30 seconds even at the 1024-GPU scale. The downtime increases by no more than 10 seconds when scaling from 32 to 1024 GPUs, as TrainMover's design is fundamentally insensitive to job size.

The scalability benefit comes from our delta-based design. For expected events, during the switching phase, TrainMover only updates the connections between the leaver and joiner, while model states are transferred in parallel between corresponding leaver–joiner pairs. All other machines and connections remain unaffected. The small increase in downtime primarily results from the larger number of RDMA connection re-establishments required at higher scales. For unexpected failures, only TrainMover's general standby machine needs to retrieve its up-to-date state from a neighbor [49, 50]. If such a system is not present, the failed machine must recover its state from the remote checkpoint storage. Other procedures remain nearly identical to those for handling expected events, as the general standby machine performs CCL initialization and sandbox initialization in advance.

In contrast, Megatron-LM restarts the entire job during migration, requiring CCL group re-instantiation and other ini-

tialization steps across all machines, taking up to nearly 300 seconds at the 1024-GPU scale. We exclude the job rescheduling and cleanup overhead in Megatron-LM, as these depend heavily on the underlying cluster management system, and this procedure typically incurs minutes-level latency (Table 1). This is consistent with the restart cost breakdown in §2, where framework initialization dominates and scales proportionally with job size.

Projected GPU-Hour Savings at Production Scale. In practice, large-scale training jobs often run on tens of thousands of GPUs. We project GPU-hour waste per week by extrapolating from our 1K-GPU measurements to production scales of up to 128K GPUs [49, 54] (Figure 9), using MTTF values from Meta [21] and a 1:8.9 expected-to-unexpected failure ratio [17]. Each system's downtime is held constant at its measured value: TrainMover uses 1024-GPU measurements, while Oobleck and Parcae use 32-GPU measurements (§8.3) since they cannot scale beyond 32 GPUs. Using 32-GPU measurements gives Oobleck and Parcae an optimistic advantage, as downtime typically grows with scale; for TrainMover, holding the 1024-GPU downtime constant is justified by its delta-based design, which only updates leaver–joiner connections and is scale-insensitive. We also add a 2-minute infrastructure rescheduling cost to all systems from the 8192-GPU measurements (Table 1), which is conservative as this overhead also increases with scale.

Two versions of TrainMover are included for comparison. The first, TrainMover (w/ standby), pre-allocates a standby machine, which is counted as a reserved resource and considered wasted if unused. We use one standby machine as a backup because simultaneous multi-machine failures are rare [18, 49, 55]. The second, TrainMover (w/o standby), uses all GPUs without a dedicated standby. Without a standby machine, the handling downtime for unexpected failures increases, while expected events remain unaffected because their handling does not require standby resources. This configuration represents the case where all GPUs are dedicated to training with no reserved standby, and only TrainMover's recovery path optimizations are applied upon failure. It also characterizes the performance when the pool of reserved standby machines is exhausted.

At the 1K scale in Figure 9, where failures are infrequent, the wasted GPU hours across all systems are similar. In fact, TrainMover (w/ standby) incurs slightly higher waste than Oobleck and Parcae, as the standby GPU remains idle during training. However, as scale increases to production levels—for example, at 8K GPUs where interruptions become more frequent (MTTF = 3h)—TrainMover (w/ standby) achieves the lowest GPU-hour waste, reducing waste by 35.91% over TrainMover (w/o standby) and by 82.77% over Parcae. Beyond 8K GPUs, TrainMover (w/ standby) continues to deliver the highest overall efficiency, as the marginal benefit of adding more training GPUs diminishes while failures become more frequent and restarts take longer. At the 128K scale, TrainMover (w/ standby) further reduces weekly GPU-hour waste by 55% compared to TrainMover w/o standby and by 88% compared to Parcae.

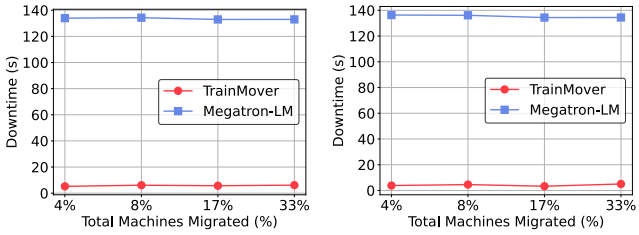
8.3 Downtime Analysis

Fast Recovery from Unexpected Failures. We evaluate TrainMover's downtime under unexpected failures in two settings: with a general standby machine and without one, as shown in Figure 11. Since Oobleck cannot scale to 1024 GPUs because of its prolonged template-generation time, we run other systems at the same 32-GPU scale for a fair comparison using the 32-GPU model settings (§8.1). For the GPT-Medium and GPT-2.7B models without the distributed optimizer enabled, TrainMover —despite having no standby—still outperforms all baselines. This is because TrainMover accelerates the initialization procedure by overlapping CCL instantiation, warm-up, framework initialization, and state transmission even when a general standby is not present. In contrast, Megatron-LM executes all components sequentially after rebooting, resulting in up to 3.48× longer downtime than TrainMover. Parcae and Oobleck, although enabling direct state transfer to reduce checkpoint loading time, still incur high restart overheads from initializing framework components, reinstantiating CCL groups, and performing warm-up. These steps lead to higher downtime than checkpoint-based approaches do.

For the GPT-39.1B and GPT-20B models, where the distributed optimizer (DO) must be enabled to fit the models into GPU memory, Oobleck and Parcae cannot be used because their redundancy assumption is eliminated. TrainMover (no standby) achieves 2.1× and 1.6× shorter downtime than Megatron-LM for the GPT-39.1B and GPT-20B models, respectively. This overlapping advantage becomes more significant as model sizes grow. In the case where a standby machine is available, TrainMover achieves performance nearly identical to live migration, with less than 10 seconds of downtime across all models.

Fast Migration on Expected Events. Migration occurs when an expected event takes place. We present the migration downtime comparison in Figure 10. In addition to the *Megatron-LM per-iteration* checkpoint baseline, which assumes that checkpoint saving is cost-free and can always be overlapped, we include two additional baselines for a comprehensive comparison of expected-event migrations. *Megatron-LM Save-and-Restart* is a more practical baseline. Before shutdown, training stops and waits for the checkpoint to be saved. *Naive live migration* directly transfers states from the leaver to the joiner without using the initialization warm-up design (§4) or the two-phase CCL setup (§5), and without requiring all machines to pull states from remote storage.

Megatron-LM Save-and-Restart performs the worst among all baselines due to significant checkpoint-saving overhead, which increases with model size, reaching approximately 400 seconds for the GPT-39.1B model. *Naive live migration* outperforms Megatron-LM Per-iteration by enabling direct trans-



(a) GPT-20B **(b)** GPT-39.1B

Figure 12: Downtime varying migration scale

fer of model states from migration leaver machines to migration joiner machines. This optimization achieves up to a 1.74× speedup on the GPT-39.1B model with TP4 and DO (distributed optimizer) enabled.

TrainMover delivers the best performance by overlapping CCL setup and warm-up processes with the non-critical path. This design consistently achieves second-level downtime and provides at least a 15× speedup compared to the *Megatron-LM Save-and-Restart* baseline across all models. Both TrainMover and the naive system maintain zero additional memory overhead during migration.

Migrating Multiple Machines at Once. Migrating multiple machines is essential for rapid large-scale operations like rebalancing and maintenance [2]. Figures 12a and 12b show the performance of GPT-20B and GPT-39.1B during migrations of 4% to 33% of the total GPUs.

Both models maintain stable downtime overhead of up to 6.15 seconds across both GPT-20B and GPT-39.1B. This consistency stems from TrainMover's design, where each migration leaver and joiner operates in parallel, performing one-to-one data transfers. This approach ensures constant overhead, preserving efficiency regardless of migration scale.

In contrast, checkpointing systems like Megatron-LM are highly inefficient. Even migrating just 4% of machines forces all machines to reboot and retrieve checkpoints from remote storage, causing 138 seconds of overhead.

8.4 Use Cases

We also present other use cases of TrainMover. Note that none of these use cases require hot standby machines.

Handling Stragglers. We evaluate the use case of handling stragglers, as shown in Figure 13. In this experiment, we inject a straggler by slowing down one GPU by 20% [51] at the 75th iteration of a 100-iteration, 1024-GPU job. Figure 13 reports the training efficiency and visualizes the training process. While this setup represents a specific case where the straggler appears at iteration 75, Figure 14 extends the analysis to all possible iterations (1st–100th), showing the full spectrum of when stragglers occur.

Alongside the previously evaluated *Per-iteration* and *Save-and-Restart* approaches, we include two additional strategies for broader comparison: deferred checkpointing and restart with progress loss. *Save-and-Restart* saves a checkpoint immediately and restarts training when a straggler occurs. *Defer-50/100* continues training and defers restarting

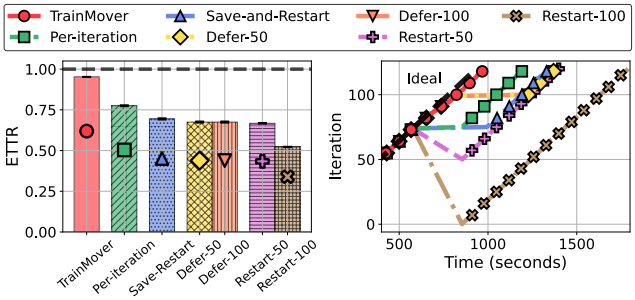


Figure 13: GPT 5.12T MoE model with 20% slowdown starting at the 75th iteration

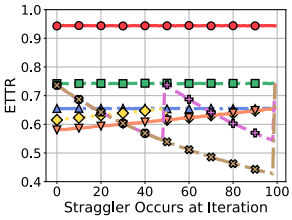


Figure 14: Straggler occurring at different iterations

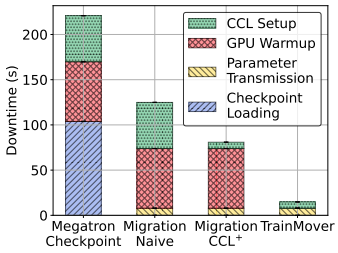


Figure 15: Design Breakdown

until the next scheduled checkpoint (e.g., every 50 or 100 iterations). *Restart-50/100* restarts training immediately, losing progress based on the most recent checkpoint at iteration 50 or 100.

As shown in Figure 13, TrainMover loses only 4.7% of training efficiency—the lowest among all approaches. During the slowdown period, TrainMover continues training while performing migration in parallel, minimizing downtime when switching from the straggler machine to a healthy one.

Defer-type baselines suffer reduced performance until the next checkpoint, restarting only afterward and thus causing delays. Restart-type baselines avoid checkpoint-saving overhead but lose progress proportional to the checkpoint interval, leading to up to 43% lower efficiency than TrainMover.

Figure 14 further illustrates the full performance spectrum across all possible straggler injection points. TrainMover consistently outperforms all baselines, maintaining high efficiency without relying on specific checkpointing frequencies. Remarkably, it even surpasses the ideal per-iteration checkpointing system, improving training efficiency by 22.8% in all cases. Restart-type methods peak immediately after a checkpoint when progress loss is minimal, but their performance deteriorates as the distance from the last checkpoint increases.

Rebalancing / Maintenance. TrainMover handles machine changes swiftly, making it suitable for rebalancing or maintenance that requires frequent GPU or network updates. The upper part of Table 16 compares the training efficiency of TrainMover and Megatron-LM per-iteration checkpointing across scales from 128 to 1024 GPUs. TrainMover consistently maintains an ETTR of above 0.97 across all scales, showing that rebalancing remains effective even as cluster size grows. In contrast, Megatron-LM's ETTR stays low under high rebalancing frequency, dropping to 0.42 at 1024

Scaling up to 1024 GPUs.

System \ Scale	128	256	512	1024
TrainMover	0.98	0.98	0.98	0.97
Megatron-LM	0.50	0.49	0.44	0.42

Breakdown at 32 GPUs.

	Enable Distributed Optimizer					Disable Distributed Optimizer				
		Med.	2.7B	20B	39.1B		Med.	2.7B	20B	39.1B
TrainMover	TP1	0.99	0.99	0.99	0.99	TP1	0.99	0.99	OOM	OOM
	TP4	0.99	0.99	0.99	0.99	TP4	0.99	0.99	0.99	OOM
	TP8	0.99	0.99	0.99	0.99	TP8	0.99	0.99	0.99	OOM
Megatron-LM	TP1	0.79	0.82	0.77	0.68	TP1	0.83	0.81	OOM	OOM
	TP4	0.86	0.86	0.85	0.77	TP4	0.84	0.85	0.92	OOM
	TP8	0.88	0.88	0.85	0.81	TP8	0.87	0.85	0.96	OOM
Oobleck[19]	**Unsupported.**					TP1	0.70	0.61	OOM	OOM
						TP4	0.62	0.49	0.03	OOM
						TP8	0.57	0.49	0.00	OOM
Parcae[12]	**Unsupported.**					TP1	0.85	0.82	OOM	OOM
						Unsupported.				

Figure 16: ETTR across large cluster scales (128–1024 GPUs, top) and detailed performance breakdown at scale 32 (bottom).

GPUs due to the increasing cost of repeated full-job restarts.

We also include comparisons with Parcae and Oobleck at the 32-GPU scale covering various model sizes and training parameters in the lower part of the table. During machine changes, online reconfiguration systems like Oobleck sometimes perform worse than the checkpointing-based Megatron-LM. This is because reconfiguration involves starting new machines, tearing down and reestablishing CCL groups, and performing warm-up procedures; collectively, these steps save little time compared to checkpoint-based systems and perform even worse when the system is not well-engineered. For instance, with the Oobleck 20B TP8 model, reconfiguration delays result in negligible training progress when rebalancing every 10 minutes. Additionally, Oobleck and Parcae lack support for distributed optimizers, and Parcae also does not support tensor parallelism, limiting their scalability for larger models.

TrainMover delivers the best performance among all approaches, maintaining a training efficiency of 0.99 across all models with and without the distributed optimizer. In comparison, the efficiency drops to 0.68 when using Megatron-LM per-iteration checkpointing. This table highlights that even in high-frequency (10-minute) rebalancing scenarios, TrainMover loses no more than 0.1% of performance, making it a robust and efficient solution.

8.5 TrainMover Key Designs

Design Breakdown. We break down our designs and evaluate their incremental impact on system performance in Figure 15 with the GPT-5.12T MoE model. In this setup, Megatron-LM's total downtime is approximately 221 seconds, with checkpoint loading, GPU warmup, and CCL setup times accounting for 47%, 30%, and 23% of the total, respectively.

Migration Naive represents a naive migration system that excludes the two-layer CCL designs and lazy initialization designs. Unlike Megatron-LM, which loads checkpoints from remote storage, it retrieves parameters directly from the leaver to the joiner. This approach eliminates checkpoint loading time, reducing the total downtime by a factor of 1.8×. The

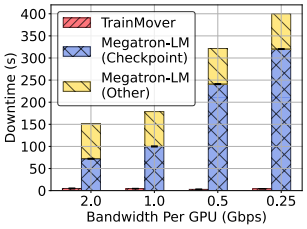


Figure 17: Downtime with different bandwidth (GPT-20B)

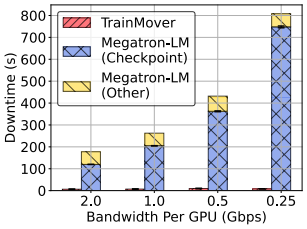


Figure 18: Downtime with different bandwidth per GPU (GPT-39.1B)

benefit becomes more pronounced as model sizes increase.

Migration CCL$^+$ builds on the naive migration baseline by introducing the CCL two-layer designs. This design shortens the critical path, leaving only the second stage of CCL initialization on it. CCL time decreases from 51 seconds to 7 seconds. This means Phase 1 absorbs ∼44 seconds off the critical path, leaving only Phase 2's 7 seconds as downtime—an 86% reduction in CCL-related recovery time.

The full system, *TrainMover*, further integrates the self-detached warmup design, effectively eliminating GPU warmup time. This enables training to resume immediately after migration, reducing downtime to 16 seconds.

TrainMover eliminates Network Bottleneck. TrainMover significantly alleviates network bandwidth bottlenecks during machine changes. The Llama-3 paper [17] reports a storage system bandwidth ranging from 2 TB/s to 7 TB/s while serving approximately 7,500 machines, translating to an effective bandwidth of 0.267 GB/s to 0.933 GB/s per machine. In this experiment, we evaluate checkpointing overhead under bandwidth ranging from 0.25 GB/s to 2 GB/s per GPU, encompassing the bandwidth range reported by Llama-3.

In both GPT-20B (Figure 17) and GPT-39.1B (Figure 18) setups, TrainMover maintains stable overhead of 6–9 seconds, due to parallel one-to-one state transfers between migration leaver and joiner. By contrast, Megatron-LM, which requires all GPUs to pull checkpoints from remote storage, incurs increasing overhead as bandwidth decreases and model size grows. Specifically, checkpoint loading overhead reaches 320 seconds for GPT-20B and 750 seconds for GPT-39.1B at 0.25 GB/s per GPU.

Zero Memory Overhead Workflow. Figure 19 illustrates the network (top) and memory (bottom) usage during a migration in TrainMover. A key step in the process is state transfer. As shown in the network breakdown, the leaver sends traffic to the joiner around the 530th second, resulting in high Rx bandwidth utilization on the joiner. However, model transfer is not without cost. Each leaver–joiner pair requires a new CCL group for state delivery, which consumes GPU memory. To achieve zero memory overhead, TrainMover implements specific optimizations during state transmission:

The migration leaver does not need to continue training after the migration is complete; around the 530th second, it can free up and repurpose the pre-allocated GPU gradient buffer for use as the CCL transmission channel. On the mi-

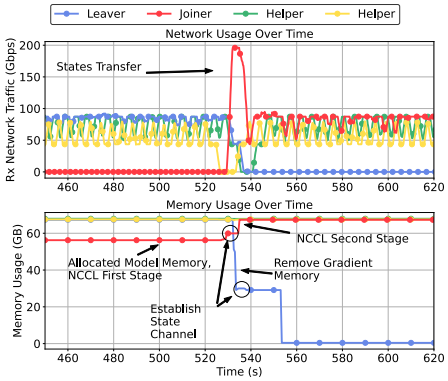


Figure 19: Network traffic and memory usage timeline during Migration

gration joiner, during the first stage of CCL initialization, inter-machine connections are not fully established, creating temporary memory headroom around the 530th second. This allows the state transfer channel to operate without exceeding memory limits. Once the transfer is complete, the channel is immediately destroyed, freeing memory before the second stage of CCL initialization. This process ensures zero memory overhead during migration.

A detailed analysis of CCL design choices is provided in Appendix A.

9 Related Work and Discussion

Economics of Standby. At massive scales, simply adding more GPUs yields marginal returns, whereas leveraging a standby machine can dramatically reduce downtime and provide far more cost-effective gains. For instance, at the 32K-GPU scale in Figure 2, using a single standby machine to shorten recovery from 4.45 minutes to 20 seconds delivers a benefit equivalent to adding roughly 2,400 GPUs (300 machines!) under ideal linear scaling. In other words, the acceleration provided by one or a few standby machines' fast recovery is comparable to provisioning thousands of additional GPUs—making standby resources vastly more economical.

CCL Reconfiguration and Generality. MCCS [52] enables runtime changes to CCL topologies but lacks dynamic membership. HotSpa [15] hides reconfiguration overhead by reusing CCL groups, at the cost of added GPU-memory overhead. NCCLX [41] speeds up CCL initialization by reducing bootstrap costs and adaptively configuring channels. In contrast, TrainMover supports dynamic CCL membership without GPU memory overhead and moves all non-critical initialization off the critical setup path. This two-stage setup is general and applies to systems that frequently reconfigure communication patterns, including reconfiguration frameworks and inference autoschedulers [12, 14, 15, 19, 23, 26, 41, 44, 52, 56].

Scalability. TrainMover scales to very large clusters. Transitional downtime comes from only two steps: inter-connection establishment and state transfer. State transfer is one-to-one—either from the leaver to the joiner during expected failures or from the joiner pulling from a checkpoint during unexpected failures—so its cost is effectively constant and the transfer size is bounded by the GPU size. Inter-connection establishment affects only the leaver's neighbors, which update their channels to the joiner. Neither step requires global coordination, enabling TrainMover to scale efficiently even in large production clusters.

GPU-granularity Migration. TrainMover's implementation supports GPU-granularity migration. The training framework can still benefit from minimal downtime and zero memory overhead. However, it forgoes the intra-machine optimizations introduced in §4.3 and pays extra storage overhead.

Heterogeneity and Warm-up Coverage. Heterogeneous or multimodal models may break symmetry assumptions or introduce non-static training paths. In practice, even if certain layer-specific or data-dependent execution paths are not fully covered, their impact is limited because the vast majority of initialization steps (e.g., data-loader setup, CUDA context creation, memory allocation, and runtime setup) remain shared across execution paths and are already covered by the warm-up procedure. These shared components dominate initialization time and are largely independent of specific layer behaviors. Extending TrainMover to fully support such dynamism remains future work.

Snapshot-based Warm Start. Recent advances such as CUDA checkpoint/restore (e.g., cuda-checkpoint [32] combined with CRIU [33]) provide an emerging mechanism for capturing and restoring GPU runtime state, offering a potential solution for warm-starting initialization—analogous to cold-start mitigation in serverless computing [5, 11]. These techniques enable preserving device memory and CUDA contexts, and are effective when the target role is known. TrainMover targets a different goal: supporting a *general standby* that can recover *any* role without prior knowledge. In large-scale training, different ranks (e.g., pipeline stages) may follow different execution paths. Supporting this with snapshots would require maintaining multiple role-specific snapshots or adapting them at runtime, increasing system complexity. In contrast, TrainMover uses record–replay to enable role-agnostic preparation without pre-defined artifacts.

10 Conclusion

We presented TrainMover, a resilient runtime that handles both expected and unexpected interruptions with minimal downtime and zero memory overhead. TrainMover leverages two-phase communication setup, communication-free sandboxed warmup, and general standby to enable rapid machine replacement. Our evaluation shows that TrainMover achieves a 20-second downtime at the 1K-GPU scale and wastes 55% fewer GPU-hours at 64K GPUs, enabling interruption-tolerant large-scale training.

Acknowledgment

We would like to thank Ion Stoica for insightful discussions on the general standby design.

References

[1] Balance of power: A full-stack approach to power and thermal fluctuations in ML infrastructure. https://cloud.google.com/blog/topics/systems/mitigating-power-and-thermal-fluctuations-in-ml-infrastructure, 2024.

[2] Maintaining large-scale AI capacity at Meta. https://engineering.fb.com/2024/06/12/production-engineering/maintaining-large-scale-ai-capacity-meta/, 2024.

[3] Megatron-LM Github Repository. https://github.com/NVIDIA/Megatron-LM, 2024.

[4] Amazon EC2 Capacity Blocks for ML pricing. https://aws.amazon.com/ec2/capacityblocks/pricing/, 2025.

[5] Lixiang Ao, George Porter, and Geoffrey M. Voelker. Faasnap: Faas made fast using snapshot-based vms. In *Proceedings of the Seventeenth European Conference on Computer Systems*, EuroSys '22, page 730–746, New York, NY, USA, 2022. Association for Computing Machinery.

[6] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, et al. Language models are few-shot learners. In *Proceedings of the 34th International Conference on Neural Information Processing Systems*, NIPS'20, Red Hook, NY, USA, 2020. Curran Associates Inc.

[7] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, et al. Language models are few-shot learners, 2020.

[8] Aakanksha Chowdhery, Sharan Narang, Jacob Devlin, Maarten Bosma, Gaurav Mishra, Adam Roberts, Paul Barham, Hyung Won Chung, Charles Sutton, Sebastian Gehrmann, Parker Schuh, Kensen Shi, Sasha Tsvyashchenko, Joshua Maynez, Abhishek Rao, et al. Palm: Scaling language modeling with pathways, 2022.

[9] DeepSeek-AI, Aixin Liu, Bei Feng, Bing Xue, Bingxuan Wang, Bochao Wu, Chengda Lu, Chenggang Zhao, Chengqi Deng, Chenyu Zhang, Chong Ruan, Damai Dai, Daya Guo, Dejian Yang, Deli Chen, et al. Deepseek-v3 technical report, 2025.

[10] Jianbo Dong, Kun Qian, Pengcheng Zhang, Zhilong Zheng, Liang Chen, Fei Feng, Yikai Zhu, Gang Lu, Zhihui Ren, Xue Li, et al. Evolution of aegis: Fault diagnosis for ai model training cloud service in production (experience track).

[11] Dong Du, Tianyi Yu, Yubin Xia, Binyu Zang, Guanglu Yan, Chenggang Qin, Qixuan Wu, and Haibo Chen. Catalyzer: Sub-millisecond startup for serverless computing with initialization-less booting. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '20, page 467–481, New York, NY, USA, 2020. Association for Computing Machinery.

[12] Jiangfei Duan, Ziang Song, Xupeng Miao, Xiaoli Xi, Dahua Lin, Harry Xu, Minjia Zhang, and Zhihao Jia. Parcae: Proactive, Liveput-Optimized DNN training on preemptible instances. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*, pages 1121–1139, Santa Clara, CA, April 2024. USENIX Association.

[13] Assaf Eisenman, Kiran Kumar Matam, Steven Ingram, Dheevatsa Mudigere, Raghuraman Krishnamoorthi, Krishnakumar Nair, Misha Smelyanskiy, and Murali Annavaram. Check-N-Run: a checkpointing system for training deep learning recommendation models. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, pages 929–943, Renton, WA, April 2022. USENIX Association.

[14] Swapnil Gandhi, Mark Zhao, Athinagoras Skiadopoulos, and Christos Kozyrakis. Recycle: Resilient training of large dnns using pipeline adaptation. In *Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles*, SOSP '24, page 211–228, New York, NY, USA, 2024. Association for Computing Machinery.

[15] Hao Ge, Fangcheng Fu, Haoyang Li, Xuanyu Wang, Sheng Lin, Yujie Wang, Xiaonan Nie, Hailin Zhang, Xupeng Miao, and Bin Cui. Enabling parallelism hot switching for efficient training of large language models. In *Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles*, SOSP '24, page 178–194, New York, NY, USA, 2024. Association for Computing Machinery.

[16] Richard L. Graham, Lion Levi, Devendar Burredy, Gil Bloch, Gilad Shainer, David Cho, George Elias, Daniel Klein, Joshua Ladd, Ophir Maor, Ami Marelli, Valentin Petrov, Evyatar Romlet, Yong Qin, and Ido Zemah. Scalable hierarchical aggregation and reduction protocol (sharp)tm streaming-aggregation hardware design and evaluation. In *High Performance Computing: 35th International Conference, ISC High Performance 2020,*

Frankfurt/Main, Germany, June 22–25, 2020, Proceedings, page 41–59, Berlin, Heidelberg, 2020. Springer-Verlag.

[17] Aaron Grattafiori, Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Alex Vaughan, Amy Yang, Angela Fan, Anirudh Goyal, Anthony Hartshorn, Aobo Yang, et al. The llama 3 herd of models, 2024.

[18] Tanmaey Gupta, Sanjeev Krishnan, Rituraj Kumar, Abhishek Vijeev, Bhargav Gulavani, Nipun Kwatra, Ramachandran Ramjee, and Muthian Sivathanu. Just-in-time checkpointing: Low cost error recovery from deep learning training failures. In *Proceedings of the Nineteenth European Conference on Computer Systems*, EuroSys '24, page 1110–1125, New York, NY, USA, 2024. Association for Computing Machinery.

[19] Insu Jang, Zhenning Yang, Zhen Zhang, Xin Jin, and Mosharaf Chowdhury. Oobleck: Resilient distributed training of large models using pipeline templates. In *Proceedings of the 29th Symposium on Operating Systems Principles*, SOSP '23, page 382–395, New York, NY, USA, 2023. Association for Computing Machinery.

[20] Ziheng Jiang, Haibin Lin, Yinmin Zhong, Qi Huang, Yangrui Chen, Zhi Zhang, Yanghua Peng, Xiang Li, Cong Xie, Shibiao Nong, Yulu Jia, Sun He, Hongmin Chen, Zhihao Bai, Qi Hou, et al. MegaScale: Scaling large language model training to more than 10,000 GPUs. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*, pages 745–760, Santa Clara, CA, April 2024. USENIX Association.

[21] Apostolos Kokolis, Michael Kuchnik, John Hoffman, Adithya Kumar, Parth Malani, Faye Ma, Zachary DeVito, Shubho Sengupta, Kalyan Saladi, and Carole-Jean Wu. Revisiting Reliability in Large-Scale Machine Learning Research Clusters . In *2025 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 1259–1274, Los Alamitos, CA, USA, March 2025. IEEE Computer Society.

[22] Oleksii Kuchaiev, Jason Li, Huyen Nguyen, Oleksii Hrinchuk, Ryan Leary, Boris Ginsburg, Samuel Kriman, Stanislav Beliaev, Vitaly Lavrukhin, Jack Cook, Patrice Castonguay, Mariya Popova, Jocelyn Huang, and Jonathan M. Cohen. Nemo: a toolkit for building ai applications using neural modules, 2019.

[23] Abhishek Vijaya Kumar, Arjun Devraj, Darius Bunandar, and Rachee Singh. A case for server-scale photonic connectivity. In *Proceedings of the 23rd ACM Workshop on Hot Topics in Networks*, HotNets '24, page 290–299, New York, NY, USA, 2024. Association for Computing Machinery.

[24] Shenggui Li, Hongxin Liu, Zhengda Bian, Jiarui Fang, Haichen Huang, Yuliang Liu, Boxiang Wang, and Yang You. Colossal-ai: A unified deep learning system for large-scale parallel training. In *Proceedings of the 52nd International Conference on Parallel Processing*, ICPP '23, page 766–775, New York, NY, USA, 2023. Association for Computing Machinery.

[25] Shengwei Li, Zhiquan Lai, Yanqi Hao, Weijie Liu, Keshi Ge, Xiaoge Deng, Dongsheng Li, and Kai Lu. Automated tensor model parallelism with overlapped communication for efficient foundation model training, 2023.

[26] Zhuohan Li, Lianmin Zheng, Yinmin Zhong, Vincent Liu, Ying Sheng, Xin Jin, Yanping Huang, Zhifeng Chen, Hao Zhang, Joseph E. Gonzalez, and Ion Stoica. AlpaServe: Statistical multiplexing with model parallelism for deep learning serving. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*, pages 663–679, Boston, MA, July 2023. USENIX Association.

[27] Jinkun Lin, Ziheng Jiang, Zuquan Song, Sida Zhao, Menghan Yu, Zhanghan Wang, Chenyuan Wang, Zuocheng Shi, Xiang Shi, Wei Jia, Zherui Liu, Shuguang Wang, Haibin Lin, Xin Liu, Aurojit Panda, et al. Understanding stragglers in large model training using what-if analysis. In *Proceedings of the 19th USENIX Conference on Operating Systems Design and Implementation*, OSDI '25, USA, 2025. USENIX Association.

[28] Qingkai Meng, Hao Zheng, Zhenhui Zhang, ChonLam Lao, Chengyuan Huang, Baojia Li, Ziyuan Zhu, Hao Lu, Weizhen Dang, Zitong Lin, Weifeng Zhang, Lingfeng Liu, Yuanyuan Gong, Chunzhi He, Xiaoyuan Hu, et al. Astral: A datacenter infrastructure for large language model training at scale. In *Proceedings of the ACM SIGCOMM 2025 Conference*, SIGCOMM '25, page 609–625, New York, NY, USA, 2025. Association for Computing Machinery.

[29] Stephen Merity, Caiming Xiong, James Bradbury, and Richard Socher. Pointer sentinel mixture models, 2016.

[30] Deepak Narayanan, Mohammad Shoeybi, Jared Casper, Patrick LeGresley, Mostofa Patwary, Vijay Korthikanti, Dmitri Vainbrand, Prethvi Kashinkunti, Julie Bernauer, Bryan Catanzaro, Amar Phanishayee, and Matei Zaharia. Efficient large-scale language model training on gpu clusters using megatron-lm. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '21, New York, NY, USA, 2021. Association for Computing Machinery.

[31] Maxim Naumov, Dheevatsa Mudigere, Hao-Jun Michael Shi, Jianyu Huang, Narayanan Sundaraman, Jongsoo Park, Xiaodong Wang, Udit Gupta, Carole-Jean Wu, Alisson G Azzolini, et al. Deep learning recommendation model for personalization and recommendation systems. *arXiv preprint arXiv:1906.00091*, 2019.

[32] NVIDIA Corporation. Cuda checkpoint and restore utility. https://github.com/NVIDIA/cuda-checkpoint, 2025. Accessed: 2026-05.

[33] OpenVZ Project. Criu: Checkpoint/restore in userspace. https://criu.org/, 2012. Accessed: 2026-05.

[34] Kun Qian, Yongqing Xi, Jiamin Cao, Jiaqi Gao, Yichi Xu, Yu Guan, Binzhang Fu, Xuemei Shi, Fangbo Zhu, Rui Miao, Chao Wang, Peng Wang, Pengcheng Zhang, Xianlong Zeng, Eddie Ruan, et al. Alibaba hpn: A data center network for large language model training. In *Proceedings of the ACM SIGCOMM 2024 Conference*, ACM SIGCOMM '24, page 691–706, New York, NY, USA, 2024. Association for Computing Machinery.

[35] Alec Radford, Karthik Narasimhan, Tim Salimans, Ilya Sutskever, et al. Improving language understanding by generative pre-training. 2018.

[36] Alec Radford, Jeff Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. Language models are unsupervised multitask learners. 2019.

[37] Samyam Rajbhandari, Jeff Rasley, Olatunji Ruwase, and Yuxiong He. Zero: memory optimizations toward training trillion parameter models. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '20. IEEE Press, 2020.

[38] Samyam Rajbhandari, Olatunji Ruwase, Jeff Rasley, Shaden Smith, and Yuxiong He. Zero-infinity: breaking the gpu memory wall for extreme scale deep learning. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '21, New York, NY, USA, 2021. Association for Computing Machinery.

[39] Jeff Rasley, Samyam Rajbhandari, Olatunji Ruwase, and Yuxiong He. Deepspeed: System optimizations enable training deep learning models with over 100 billion parameters. In *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, KDD '20, page 3505–3506, New York, NY, USA, 2020. Association for Computing Machinery.

[40] Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. Megatron-lm: Training multi-billion parameter language models using model parallelism. *arXiv preprint arXiv:1909.08053*, 2019.

[41] Min Si, Pavan Balaji, Yongzhou Chen, Ching-Hsiang Chu, Adi Gangidi, Saif Hasan, Subodh Iyengar, Dan Johnson, Bingzhe Liu, Regina Ren, Ashmitha Jeevaraj Shetty, Greg Steinbrecher, Yulun Wang, Bruce Wu, Xinfeng Xie, et al. Collective communication for 100k+ gpus, 2025.

[42] Gemini Team, Rohan Anil, Sebastian Borgeaud, Jean-Baptiste Alayrac, Jiahui Yu, Radu Soricut, Johan Schalkwyk, Andrew M. Dai, Anja Hauth, Katie Millican, David Silver, Melvin Johnson, Ioannis Antonoglou, Julian Schrittwieser, Amelia Glaese, et al. Gemini: A family of highly capable multimodal models, 2025.

[43] The Authors. TrainMover prototype source code. https://github.com/alibaba/TrainMover, 2026.

[44] John Thorpe, Pengzhan Zhao, Jonathan Eyolfson, Yifan Qiao, Zhihao Jia, Minjia Zhang, Ravi Netravali, and Guoqing Harry Xu. Bamboo: Making preemptible instances resilient for affordable training of large DNNs. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 497–513, Boston, MA, April 2023. USENIX Association.

[45] Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, et al. Llama: Open and efficient foundation language models. *arXiv preprint arXiv:2302.13971*, 2023.

[46] Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, et al. Llama 2: Open foundation and fine-tuned chat models. *arXiv preprint arXiv:2307.09288*, 2023.

[47] Abhishek Verma, Luis Pedrosa, Madhukar Korupolu, David Oppenheimer, Eric Tune, and John Wilkes. Large-scale cluster management at google with borg. In *Proceedings of the Tenth European Conference on Computer Systems*, EuroSys '15, New York, NY, USA, 2015. Association for Computing Machinery.

[48] Borui Wan, Mingji Han, Yiyao Sheng, Yanghua Peng, Haibin Lin, Mofan Zhang, Zhichao Lai, Menghan Yu, Junda Zhang, Zuquan Song, Xin Liu, and Chuan Wu. Bytecheckpoint: a unified checkpointing system for large foundation model development. In *Proceedings of the 22nd USENIX Symposium on Networked Systems Design and Implementation*, NSDI '25, USA, 2025. USENIX Association.

[49] Borui Wan, Gaohong Liu, Zuquan Song, Jun Wang, Yun Zhang, Guangming Sheng, Shuguang Wang, Houmin Wei, Chenyuan Wang, Weiqiang Lou, Xi Yang, Mofan Zhang, Kaihua Jiang, Cheng Ren, Xiaoyun Zhi, et al. Robust llm training infrastructure at bytedance. In *Proceedings of the ACM SIGOPS 31st Symposium on Operating Systems Principles*, SOSP '25, page 186–203, New York, NY, USA, 2025. Association for Computing Machinery.

[50] Zhuang Wang, Zhen Jia, Shuai Zheng, Zhen Zhang, Xinwei Fu, T. S. Eugene Ng, and Yida Wang. Gemini: Fast failure recovery in distributed training with in-memory checkpoints. In *Proceedings of the 29th Symposium on Operating Systems Principles*, SOSP '23, page 364–381, New York, NY, USA, 2023. Association for Computing Machinery.

[51] Tianyuan Wu, Wei Wang, Yinghao Yu, Siran Yang, Wenchao Wu, Qinkai Duan, Guodong Yang, Jiamang Wang, Lin Qu, and Liping Zhang. Falcon: Pinpointing and mitigating stragglers for large-scale hybrid-parallel training, 2024.

[52] Yongji Wu, Yechen Xu, Jingrong Chen, Zhaodong Wang, Ying Zhang, Matthew Lentz, and Danyang Zhuo. Mccs: A service-based approach to collective communication for multi-tenant cloud. In *Proceedings of the ACM SIGCOMM 2024 Conference*, ACM SIGCOMM '24, page 679–690, New York, NY, USA, 2024. Association for Computing Machinery.

[53] Zhanghao Wu, Wei-Lin Chiang, Ziming Mao, Zongheng Yang, Eric Friedman, Scott Shenker, and Ion Stoica. Can't be late: Optimizing spot instance savings under deadlines. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*, pages 185–203, Santa Clara, CA, April 2024. USENIX Association.

[54] xAI. xai's colossus supercomputer cluster. https://x.ai/colossus/, 2024. Accessed 2024.

[55] Susan Zhang, Stephen Roller, Naman Goyal, Mikel Artetxe, Moya Chen, Shuohui Chen, Christopher Dewan, Mona Diab, Xian Li, Xi Victoria Lin, Todor Mihaylov, Myle Ott, Sam Shleifer, Kurt Shuster, Daniel Simig, et al. Opt: Open pre-trained transformer language models, 2022.

[56] Xiaoyang Zhao, Zhe Zhang, and Chuan Wu. Adapcc: Making collective communication in distributed machine learning adaptive. In *2024 IEEE 44th International Conference on Distributed Computing Systems (ICDCS)*, pages 25–35, 2024.

ARTIFACT EVALUATED
USENIX
AVAILABLE

MOONBRIGHT: A GPU Memory Allocator with Device-Side Page Table Materialization and Deferred TLB Coherence

Yangyu Zhang[1,2] Lei Chen[2] Chunwei Xia[3] Shuaijiang Li[1,2] Shuoming Zhang[1,2]
Zhicheng Li[1,2] *Qianqi Sun*[1,2] *Jiawei Xiao*[1,2] *Ruiyuan Xu*[1,2] *Ao Chen*[1,2]
Guangli Li[1,2] *Xiaobing Feng*[1,2] *Huimin Cui*[1,2] *Chenxi Wang*[1,2] *Jiacheng Zhao*[1,2,†]

[1]*SKLP, Institute of Computing Technology, Chinese Academy of Sciences*
[2]*University of Chinese Academy of Sciences* [3]*University of Leeds*

Abstract

Modern GPU workloads increasingly rely on dynamic and fine-grained memory allocation, yet GPU memory management remains CPU-centric. In current GPU runtimes, allocation metadata updates, page-table construction, translation-state propagation, and TLB shootdowns are largely serialized through the host control path, introducing substantial latency. We present MOONBRIGHT, a GPU memory allocator that enables device-side page-table materialization and deferred TLB coherence on commodity GPUs. MOONBRIGHT keeps validation and allocation metadata on the host, but moves bulk page-table construction to the GPU, turning translation updates into data-parallel device-memory operations. To avoid costly TLB shootdowns on the common path, MOONBRIGHT assigns fresh virtual addresses to new mappings, ensuring that no stale same-address TLB entries can exist. Our evaluation shows that MOONBRIGHT reduces allocation latency, improves LLM inference performance, and mitigates allocator-level external fragmentation across diverse workloads. Unlike application-specific memory managers, MOONBRIGHT requires no GPU hardware modifications and runs on commodity NVIDIA and AMD GPUs with software-only changes. MOONBRIGHT is publicly available at https://github.com/MoonBright-project.

1 Introduction

Memory management has long been a core systems discipline, and decades of CPU hardware-software co-design have produced highly optimized virtual-memory systems and allocator designs, including PTMalloc [15], JEMalloc [14], TCMalloc [20], and others [21, 24, 27, 28].

Modern GPUs now host a diverse spectrum of workloads, spanning large-scale DL models [9, 13, 17, 63], data analytics [34, 42], scientific simulations [2, 3, 55, 69], graph processing [31, 38, 67, 75], and emerging general-purpose computations [54, 60]. GPUs such as NVIDIA B200 [40] have reached the thousand-TFLOPS scale, allowing many CUDA kernels to complete within only a few microseconds. As GPU execution becomes faster and more fine-grained, CPU-driven memory allocation increasingly falls on the critical path [23], turning frequent allocation into a major bottleneck.

Although modern GPUs include hardware MMUs for address translation, GPU memory management remains largely CPU-centric. The host maintains the authoritative page tables, and allocation, deallocation, and remapping requests are routed through the driver, which serializes translation updates and propagates them to the GPU. When mappings are changed or reused, the host is also responsible for enforcing device-side TLB coherence, often through costly TLB shootdowns. Consequently, allocation calls such as `cudaMalloc` incur high control-path latency.

Existing systems mitigate allocation overhead primarily through pool-based allocation. Framework allocators such as TensorFlow's BFC allocator [1] and PyTorch's caching allocator [43], as well as CUDA's stream-ordered allocator, reduce calls into the driver by caching and reusing previously allocated memory. While effective at reducing allocation latency, these designs still rely on reuse within pre-allocated memory regions rather than addressing the CPU-centric control path itself. As a result, they inherit the central weakness of pool-based allocation: fragmentation can accumulate when allocation lifetimes and sizes do not match the allocator's layout and reuse policies.

An alternative is on-demand mapping using low-level memory-management APIs such as CUDA Virtual Memory Management (VMM) [39]. However, this path is blocked by *prohibitive mapping latency*. VMM's mapping overhead scales with the number of pages whose mappings should be established or updated, and serialized host-side driver calls can incur millisecond-scale delays. This makes VMM mapping up to three orders of magnitude slower than microsecond-scale kernel execution. Recent systems such as vAttention [45] and GMLake [16] attempt to *hide* this latency through complex

† Jiacheng Zhao is the corresponding author.

scheduling. However, they do not address the root cause: the mapping latency itself remains high, leaving systems vulnerable to latency spikes in dynamic deep-learning workloads.

The conflict is clear: allocating memory on the fly introduces high latency and hurts performance, while pre-allocated memory pools lead to fragmentation and hurt memory utilization. These derivative problems raise a classic operating-systems question: When a foundational abstraction layer is inadequate, should the user-space application layer continue building workarounds, or is it time to fix the foundation itself? Instead of continuing to patch this foundation with user-space workarounds, we ask a fundamental question: *Can we re-architect GPU virtual memory management to address both allocation latency and fragmentation?*

Our key insight is that GPU memory management should not be treated as a serialized OS control task, but as a data-parallel workload ideally suited for the GPU itself. Although promising, translating this theoretical insight into a practical system requires bridging two fundamental challenges:

Challenge 1: How can the GPU maintain its page table efficiently? Existing GPUs lack privileged instructions for page table manipulation from user space. How can we enable direct GPU control over page tables without hardware modifications?

Our solution. We propose MOONBRIGHT, which leverages both the massive parallelism and ultra-high memory bandwidth of modern GPUs to efficiently construct and update page tables directly on the device. First, by mapping the multi-level page table region into the virtual address space, MOONBRIGHT allows CUDA kernels to natively read and write page-table entries. Second, all modifications are executed by MOONBRIGHT-generated kernels specifically tuned to fully saturate the GPU's multi-TB/s bandwidth, slashing page-table update latencies to mere microseconds.

Challenge 2: How can we minimize TLB flushes while maintaining coherence? CPU-issued TLB shootdown is a fundamental component of current GPU memory management. Commodity GPUs require the CPU to initiate TLB shootdowns, and each flush can impose a global CPU-GPU synchronization point before memory operations proceed. This creates a major performance bottleneck and prevents page-table updates from being integrated cleanly into *stream-ordered execution*.

Our solution. MOONBRIGHT introduces a *Deferred TLB Coherence* protocol to decouple memory allocation from expensive global synchronization. The key mechanism is an *Always-Fresh* virtual address (VA) allocation strategy. By guaranteeing that every new allocation receives a previously unused VA, MOONBRIGHT ensures that new mappings inherently avoid conflicts with stale TLB entries. This invariant allows MOONBRIGHT to safely bypass TLB flushes on the critical allocation path, relegating explicit coherence operations strictly to exceptional cases, such as VA recycling, remapping, or permission modifications.

Method	Fast Alloc.	Stream Order	Flexible Map	Defrag.
cudaMalloc	✗	✗	✗	✗
cudaMallocAsync	✓	✓	✗	✗
CUDA VMM	✗	✗	✓	✓
Userspace Caching Allocator	✓	✓	✗	✗
Ours	✓	✓	✓	✓

Table 1: Comparison of memory primitives. **Fast Alloc.**: Low allocation latency on the critical path; **Stream Order**: Non-blocking, stream-aware allocation semantics; **Flexible Map**: Decoupling of virtual and physical memory for arbitrary mappings; **Defrag.**: External-fragmentation mitigation. MOONBRIGHT uniquely achieves all four properties.

Together, these techniques allow MOONBRIGHT to support GPU-side page-table updates while maintaining the coherence guarantees required by modern GPU runtimes. MOONBRIGHT exports several primitives and composes them into a familiar runtime API surface, including `Malloc`, `MallocAsync`, and VMM-style interfaces, while reducing mapping latency and external fragmentation.

Our evaluation shows that MOONBRIGHT reduces mapping latency by up to three orders of magnitude, and that these low-level improvements translate into end-to-end benefits: MOONBRIGHT improves TTFT by up to 8.2× for LLM inference compared with vAttention, and mitigates external fragmentation across training workloads compared with PyTorch's caching allocator.

This paper makes the following contributions:

- **Bottleneck Analysis.** We identify the *CPU-centric* memory management model and *synchronous TLB flushes* as fundamental architectural bottlenecks, which impose severe synchronization overheads on GPU memory management.
- **GPU-Centric Architecture.** We propose a *GPU-centric* design and, to our knowledge, present the first device-side page-table management mechanism on commodity GPUs, moving critical-path operations off the host.
- **Deferred Coherence Protocol.** We introduce an *Always-Fresh* virtual address allocation strategy. This ensures TLB coherence without frequent TLB shootdowns, removing synchronization cost from the common allocation path.
- **System Implementation.** We implement MOONBRIGHT as a purely software mechanism on commodity NVIDIA and AMD GPUs, requiring no hardware modifications and integrating transparently with CUDA and HIP runtimes.

To facilitate further research in GPU systems, we have open-sourced the implementation of MOONBRIGHT at MoonBright-project.

2 Background and Motivation

2.1 Managing GPU Memory

We use CUDA to illustrate modern GPU memory management, but the same design patterns and limitations also apply to HIP/ROCm. CUDA provides three classes of device-memory management interfaces: the `cudaMalloc`, `cudaMallocAsync`, and the `CUDA Virtual Memory Management (VMM) API`. These interfaces occupy different points in the design space of allocator flexibility, synchronization overhead, and fragmentation.

CUDA VMM API. The Low-Level GPU Virtual Memory Management interface (e.g., `cuMemAddressReserve`, `cuMemCreate`, `cuMemMap`, `cuMemSetAccess`) exposes low-level virtual memory primitives. It decouples virtual address reservation, physical memory creation, mapping, and access control, allowing page-granular management of device memory. In the CUDA VMM path, `cuMemMap` binds physical allocations to reserved virtual ranges, while `cuMemSetAccess` installs access permissions and makes the mapping visible to the device. Current implementations keep this path host-driven and can trigger global CPU-GPU synchronization, resulting in long operation latency [16,45] and preventing stream-ordered execution.

cudaMalloc. The traditional GPU memory allocation API can be seen as a fixed sequence of `VMM` operations. The runtime reserves a contiguous virtual address range, allocates physical memory of the same size, maps the two, and installs access permissions. These calls also impose global synchronization, so allocation and deallocation cannot overlap with kernel execution and often dominate critical-path latency.

cudaMallocAsync. To reduce allocation latency, CUDA provides `cudaMallocAsync` and `cudaFreeAsync`, which operate on a runtime-managed memory pool. Requests are served from cached blocks rather than invoking VMM operations. This achieves low latency but fixes allocation granularity to pool blocks. Because the allocator cannot flexibly remap virtual pages to physical pages, fragmentation accumulates under workloads with diverse or long-lived allocations.

Userspace caching allocators. To mitigate the high mapping latency of `cudaMalloc`, modern frameworks often deploy custom userspace allocators, such as those in PyTorch and TensorFlow. These allocators follow the same design principle as `cudaMallocAsync`: they overprovision a large memory pool using `cudaMalloc` and manage it with best-fit or similar placement policies. As a result, they inherit the same fundamental limitation. Once the pool develops holes, subsequent allocations may be satisfiable only by non-contiguous free regions, yet the allocator lacks the ability to manipulate page tables to eliminate fragmentation. Consequently, memory utilization degrades over time, especially in long-running or dynamically shaped workloads.

Alloc. Size	Book-keeping	Phys. Alloc.	PT Build	TLB Flush	CUDA Total	Ours Total
2 MB	0.8	8	47	3.7	59	**24**
16 MB	1.1	9	422	5.1	437	**27**
128 MB	1.2	10	949	6.3	966	**32**

Table 2: Latency breakdown of `cudaMalloc` for different allocation sizes. All values are in microseconds. Page-table construction dominates the CUDA path; MOONBRIGHT's total latency is shown for comparison.

2.2 Architectural Bottlenecks

The aforementioned limitations are not merely engineering artifacts of specific APIs; they stem from deeper architectural choices in how GPUs manage memory control paths.

2.2.1 Dissecting Allocation Latency.

We find that allocation latency arises fundamentally from the *CPU-centric* page-table model. Historically, the CPU maintains the authoritative page tables, while GPU-resident page tables act as device-side replicas. Consequently, every allocation, deallocation, remapping operation, or permission update is first processed on the CPU, serialized through the vendor driver, and then propagated to GPU-resident translation structures. Once the GPU page table is updated, the host triggers a TLB flush to maintain coherence across the GPU's translation caches.

Table 2 breaks down the latency of a `cudaMalloc` operation on an NVIDIA A100 GPU. By instrumenting the driver and runtime phases, we decompose the end-to-end latency into four components: *Logical bookkeeping* (tracking VA metadata), *Physical allocation* (obtaining device pages), *Page-table construction and transmission* (building CPU-resident PTEs and shipping them via PCIe), and *TLB flush*. As shown in Table 2, allocation latency remains high, reaching approximately 1 ms for a 128 MB allocation.

More importantly, *page-table construction and transmission* overwhelmingly dominate the cost, accounting for 80% to 99% of the total latency across the evaluated sizes. This disproportionate expense is not simply a PCIe bandwidth artifact. Modifying translations requires mutating shared address-space state: validating VA ranges, instantiating missing page-table levels, encoding leaf PTEs, and locking structures against concurrent updates. Crucially, while encoding thousands of independent leaf PTEs is an inherently data-parallel task, the CPU-centric model forces the host driver to construct them *serially* within a sequential execution loop. The observed latency is thus a massive compound penalty of this sequential host-side PTE construction, state propagation across the PCIe bus, and conservative coherence enforcement.

Insight 1: GPU-Resident Translation. This observation mo-

tivates MOONBRIGHT's core architectural pivot: making the device page table authoritative for device translations. Once the host supplies the physical-frame list, PTE materialization becomes an inherently data-parallel operation over independent pages. MOONBRIGHT therefore dispatches GPU kernels to directly update device-resident page tables, leveraging the GPU's massive parallelism and multi-TB/s memory bandwidth to bypass serialized CPU driver loops. As highlighted in Table 2, this design reduces the 128 MB page-table update latency to only 32 μs, shifting page-table materialization from a dominant control-plane bottleneck to a lightweight device-side operation.

2.2.2 Dissecting Memory Fragmentation.

To amortize the severe control-path overhead discussed above, deep learning frameworks universally rely on user-space pooling. Allocators such as TensorFlow's BFC [1] and PyTorch's caching allocator [43] acquire massive memory regions from the driver upfront, serving subsequent requests through user-space suballocation. More recently, NVIDIA introduced stream-ordered allocation via `cudaMallocAsync`, which extends this philosophy by serving requests from CUDA-managed pools without enforcing device-wide synchronization.

However, because these designs treat pooling as their sole defense against mapping latency, they inherit the fundamental weakness of pool-based allocation: they cannot manipulate page tables to consolidate memory. Recent studies [16, 19, 25] confirm that the irregular, dynamic tensor lifetimes in large-scale training and inference inevitably misalign with the pool's layout. This mismatch rapidly accumulates severe external fragmentation, wasting device memory and halting scalability.

Ideally, an allocator would cure fragmentation by using Virtual Memory Management (VMM) APIs to transparently remap physical pages behind contiguous virtual addresses. Yet, under the current CPU-centric architecture, any attempt to dynamically remap pages introduces a catastrophic performance penalty. To ensure coherence, a VMM remapping operation demands global device-wide synchronization: the CPU driver must wait for all active kernels across all streams to finish, drain the device entirely, and only then issue a global TLB flush. Even though the hardware flush itself is relatively fast (e.g., 6.3 μs in Table 2), this mandatory global synchronization forces the GPU to completely empty its execution pipelines. This heavy-handed barrier forcibly degrades modern asynchronous, stream-ordered execution back into blocking, synchronous execution, rendering VMM-based defragmentation practically unusable for latency-sensitive frameworks.

Insight 2: Deferred Coherence via Fresh VAs. One of our observations is that GPU workloads require far fewer TLB coherence events than CPU counterparts. While CPUs frequently mutate existing address spaces (e.g., context switches, permission changes), GPU workloads are highly regular, predominantly allocating *new* memory. Our TLB microbenchmarks (§5.5) show that, when a newly allocated virtual address has no stale same-address translation in the TLB, MOONBRIGHT can safely install the mapping *without* a flush. The first access observes a TLB miss and walks the newly materialized page-table entry.

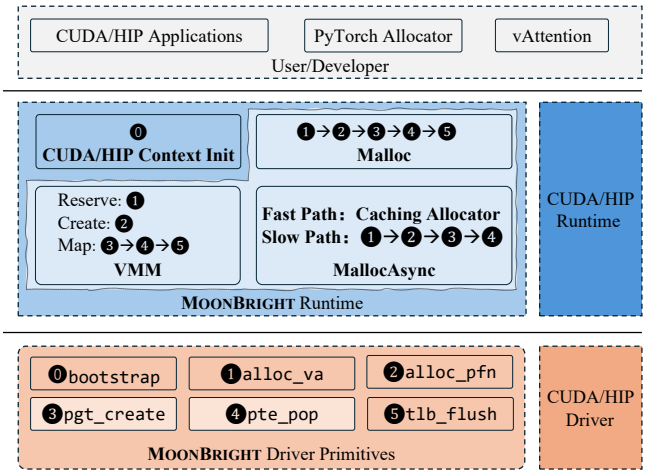


Figure 1: MOONBRIGHT System Overview.

This insight underpins MOONBRIGHT's *Deferred VA Reuse* (Always-Fresh) policy. By ensuring the allocator strictly assigns previously unused virtual address ranges to new allocations, MOONBRIGHT guarantees that new PTEs will not collide with stale TLB entries. This elegantly circumvents the stop-the-world TLB flush, deferring coherence operations only to the rare cases of actual VA recycling. Consequently, MOONBRIGHT can perform asynchronous, low-latency defragmentation in the background without stalling compute streams.

Takeaway. The host-side bottlenecks inherent to the CPU-centric page-table model necessitate a GPU-centric architectural shift. Centralizing page-table construction on the GPU eliminates CPU-mediated serialization latency, and the Always-Fresh virtual address policy eliminates unnecessary TLB flushes. Together, these two pillars form the foundation of MOONBRIGHT, uniquely satisfying the requirements of fast allocation, flexible mapping, and external-fragmentation mitigation (as summarized in Table 1).

3 Design of MOONBRIGHT

MOONBRIGHT introduces two key mechanisms: device-side page-table materialization and deferred TLB coherence. Figure 1 presents the overall architecture of MOONBRIGHT, which adopts a two-layer organization consisting of a driver layer and a runtime layer. The driver layer exposes a set of low-

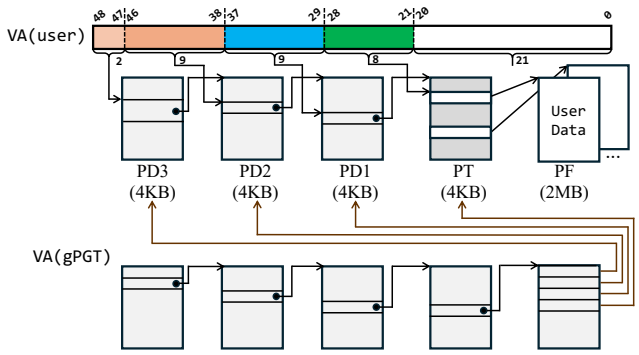


Figure 2: **Bootstrap.** The hardware page table structures backing VA(user) are mapped to a specialized virtual address, VA(gPGT). This pointer allows CUDA kernels to treat the page table as an array.

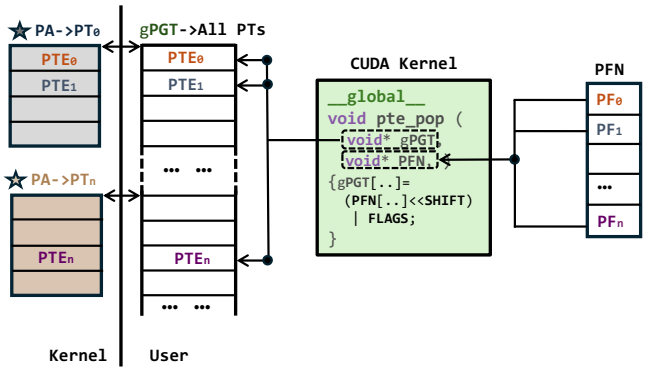


Figure 3: Workflow of the parallel PTE population kernel (`pte_pop`). MOONBRIGHT utilizes massively parallel GPU threads to dynamically synthesize Page Table Entries (PTEs) directly into the global GPU Page Table (*gPGT*).

level memory-management primitives (e.g., page-table manipulation, virtual-address allocation, and fine-grained mapping operations). The runtime layer orchestrates these primitives to implement standard memory allocation interfaces compatible with existing GPU software, from native CUDA/HIP applications to deep-learning frameworks such as PyTorch.

3.1 Composable Low-Level Primitives

MOONBRIGHT fundamentally shifts page table construction to the GPU by defining updates as data-plane workloads. To ensure the host (Control Plane) maintains a consistent view of the device-resident state without costly synchronization, we introduce a *Linearized Page Table Abstraction*. This allows the CPU to deterministically calculate the location of any page table entry (PTE) in GPU memory using purely arithmetic operations, eliminating the need for costly round-trip PCIe communications.

❶ ***Bootstrap: Linearized View Construction.*** Triggered upon CUDA context creation, this primitive initializes the GPU-resident metadata required to manage the hardware's multi-level page table hierarchy ($PD_3 \rightarrow PD_2 \rightarrow PD_1 \rightarrow PT$). As shown in Figure 2, MOONBRIGHT must make the physical pages that hold the hardware page-table hierarchy reachable through a device virtual address before a CUDA kernel can update them. Specifically, MOONBRIGHT obtains the physical addresses of the multi-level page-table pages and bootstrap-maps those pages into a contiguous virtual range, creating the *GPU Page Table* (*gPGT*) view. *Abstraction Provided:* This primitive exposes the multi-level page table as a flattened, linear array ($gPGT[va_idx]$). This abstraction allows both the CPU and GPU to index PTEs using a simple offset calculation ($Base + VPN \times sizeof(PTE)$), enabling fast coordination between host orchestration and device execution.

MOONBRIGHT exposes five *composable primitives* that serve as the building blocks for customized memory allocators:

❶ ***Virtual Address Allocation (alloc_va).*** Managed by a CPU-resident Red-Black tree, this primitive governs the logical layout of the application's virtual address space. It allocates virtual memory ranges (VAs) and maintains high-level metadata (e.g., valid/free status).

❷ ***Physical Page Allocation (alloc_pfn).*** This primitive interfaces with the low-level physical resource manager (e.g., NVIDIA GPU System Processor (GSP) or HIP Buddy Allocator) to acquire available physical pages. It requests physical page frame numbers (PFNs) and ensures the allocated memory remains resident and accessible to the device. Crucially, it captures the *Allocation Layout State* alongside the raw PFNs. This state serves as a coordination protocol for downstream primitives, providing the necessary physical address metadata required for the subsequent parallel mapping operation.

❸ ***Page Table Construction (pgt_create).*** This primitive serves to populate sparse regions of the page table hierarchy. Upon detecting a missing directory level for a target range, it validates the request against the virtual address state maintained by ❶. Subsequently, it allocates the requisite physical backing for the directory (e.g., *PT*) and executes a GPU kernel to install the physical pointers into the global page table array (*gPGT*), ensuring the structural integrity of the mapping tree.

❹ ***Parallel PTE Population (pte_pop).*** This core primitive unifies the transformation of physical addresses into hardware-compliant Page Table Entries (PTEs) and their population into the GPU page table. Unlike standard drivers that perform serialized PTE population from the host CPU, MOONBRIGHT first transfers raw Physical Frame Numbers (PFNs) to a GPU-resident buffer, termed the *GPU PFN Buffer* (*PFN*).

As illustrated in Figure 3, the mapping workflow executes as a single, high-bandwidth parallel kernel that fuses construction and population. At runtime, thousands of GPU threads op-

erate in parallel to execute a two-step process: First, **On-the-fly Construction**: The GPU kernel reads raw page frame numbers (*PFN*) and dynamically synthesizes hardware-compliant PTEs. This is achieved by combining physical addresses with user-specified access permission bits (e.g., Read/Write flags) using efficient bitwise operations. Second, **Population**: The generated PTEs are immediately written into the destination slots of the GPU Page Table (*gPGT*).

By keeping both the source data (*PFN*) and the destination structure (*gPGT*) resident in high-bandwidth device memory, this design transforms memory management into a pure, on-device data movement task. It leverages the GPU's massive parallelism to populate page tables at memory-bandwidth limits, executing substantially faster than CPU-driven approaches. Furthermore, because this stage is implemented via a programmable *GPU Kernel* rather than rigid driver callbacks, MOONBRIGHT allows developers to utilize standard parallel programming models to express diverse mapping paradigms, dynamically routing PFNs to arbitrary virtual addresses.

❺ ***TLB Flush (tlb_flush).*** This primitive is invoked when an update can leave stale translations for the same virtual address, such as unmapping, permission changes, or remapping an existing VA. It triggers a host-initiated hardware TLB shootdown and therefore introduces a GPU-wide synchronization point. Fresh-VA mappings do not use this primitive: subsequent kernels observe the new PTEs through normal stream ordering, and no stale same-VA translation can exist.

3.2 User-Facing Memory API Design

In this subsection, we present how to construct memory allocators based on MOONBRIGHT's primitives.

3.2.1 Reconstructing `Malloc`

To demonstrate the composability of our primitives, we present a high-performance implementation of a standard synchronous memory allocator. Designed as a functional equivalent to `cudaMalloc`, this implementation orchestrates the primitives defined in §3.1 into a cohesive pipeline, addressing the granularity mismatch between user requests and hardware pages via a standard slab layer.

Large Allocation. For requests exceeding the slab threshold, the runtime orchestrates a sequential primitive pipeline: resources are first secured via ❶ and ❷, followed by hierarchy instantiation using ❸. The mapping is then materialized by the ❹ parallel kernel. If the allocation uses a fresh VA, no TLB flush is required. A ❺ flush is appended only when the allocator must recycle an old VA or update an existing mapping.

Small Object Management. Since hardware physical pages are fixed at a coarse granularity (e.g., 2 MB) while user requests might be fine-grained (e.g., 1 KB), we employ a classic slab [8] allocation strategy to manage sub-page objects. The allocator maintains user-space metadata organized by size classes. For requests with smaller objects, it locates available slots within active slabs via bitmap scanning. This operation is performed entirely in user space, avoiding the latency of driver interactions or hardware primitives. Only when a slab is fully exhausted or a request exceeds the slab threshold does the allocator invoke the underlying primitive layer to acquire fresh physical resources.

De-allocation. The de-allocation process (`MemFree`) executes a strictly ordered sequence to ensure memory consistency. First, the runtime enforces a device-side synchronization barrier to ensure all active kernels accessing the target memory have completed. Subsequently, it invokes a specialized unmapping kernel to atomically invalidate the PTEs in the GPU-resident page table by zeroing out the corresponding entries. Once the source mappings are invalidated, a TLB flush is immediately issued to evict any stale translations, thereby enforcing coherence. Only after this sequence is confirmed does the runtime return the physical page frames to the system allocator via the inverse of *alloc_pfn*. To minimize reconstruction overhead during future allocations in the same region, the upper-level page table hierarchy associated with the virtual address range is preserved.

3.2.2 Fragmentation-Mitigating `MallocAsync`

We propose an asynchronous memory allocator that exposes fast, fragmentation-free allocation as a virtual-memory primitive rather than as a framework-specific caching policy.[2] By moving allocation efficiency into the virtual-memory layer, MOONBRIGHT provides a common substrate that can serve existing frameworks directly or simplify their caching allocators, rather than requiring each framework to independently engineer fragmentation mitigation.

Deterministic TLB Coherence via Fresh Virtual Addressing. The central challenge in asynchronous allocation is maintaining TLB coherence without global synchronization caused by TLB flushes. Standard allocators require explicit flushes to ensure both correctness and performance determinism. This is necessitated by the *non-coherent* nature of GPU MMUs: hardware lacks a mechanism to actively propagate page table updates to the TLB, risking the usage of stale or inconsistent entries (e.g., negative caching artifacts [74]) if not flushed.

To eliminate this hazard without blocking execution, we propose an *"Always-Fresh" Virtual Address (VA)* allocation strategy. By ensuring that new allocation occupies a fresh, previously unused VA range within the current coherence epoch, we guarantee the absence of stale same-VA TLB entries; therefore, no flush is needed on this fast path. This strategy is architecturally feasible for two reasons.

[2] Internal fragmentation due to page granularity is a separate policy trade-off and is orthogonal to the virtual-memory mechanism studied in this paper. Unless otherwise stated, fragmentation refers to external fragmentation.

First, MOONBRIGHT provides the high-throughput mapping primitive needed to sustain rapid VA consumption. Second, the required virtual-address space can be sized using a theoretical guideline. We adopt Robson's worst-case allocation analysis [53] as a conservative estimate for provisioning the VA buffer: $M(1+\frac{1}{2}\log_2 n)$, where M is the physical memory size and n is the ratio between memory size and page size. For an 80 GB GPU with 2 MB pages, this estimate is approximately 693 GB. This footprint is well within the hundreds-of-TB virtual address spaces exposed by modern GPUs and incurs only modest overhead for additional page-table structures. Guided by this theoretical estimate, MOONBRIGHT manages VA space as a large buffer with lazy reclamation, ensuring TLB coherence through spatial separation rather than frequent synchronization.

Correctness of Always-Fresh VA. The key invariant behind MOONBRIGHT's TLB-flush avoidance is that a newly installed mapping never reuses a virtual address that may have had a valid same-address translation cached in the GPU TLB. For fresh virtual addresses, there cannot exist a stale same-address valid translation, because the address has not previously been mapped or successfully translated by the device. Therefore, installing a fresh mapping does not require invalidating old translations.

Speculative translation activity does not violate this invariant. Hardware page walkers or translation prefetchers may probe nearby virtual addresses and cache negative translation results, but such entries encode the absence of a valid mapping rather than a *VA-to-physical-frame* binding. They therefore cannot redirect a future access to an old physical frame. Once MOONBRIGHT installs a valid PTE for a fresh VA, the first access either observes the new translation or triggers a page walk that reloads the updated PTE. In this way, MOONBRIGHT replaces a global TLB shootdown with a translation miss, without compromising correctness.

Finally, always-fresh allocation is bounded by the available virtual-address space. MOONBRIGHT treats VA exhaustion as a rare slow-path event. When the current VA epoch approaches exhaustion, MOONBRIGHT waits for outstanding GPU work that may reference the old epoch, performs a global TLB invalidation, and then starts a new epoch in which previously used virtual addresses may be recycled. Thus, VA reuse occurs only after device quiescence and TLB invalidation, preserving the same-address freshness invariant.

Asynchronous Memory Pool Architecture. To provide an allocator with semantics familiar to users of `cudaMallocAsync`, we implement a high-performance memory pool built directly upon our decoupled primitives. The foundation of this architecture is a massive virtual address ring buffer. By maintaining the page table entries (PTEs) in GPU-resident memory, we remove the CPU from the critical path of mapping updates. This design allows the runtime to generate static code with fixed pointers, even as the physical backing memory changes dynamically via the *pte_pop* primitive. Furthermore, this buffer design enables lazy reclamation: when an object is freed, MOONBRIGHT quarantines its VA range instead of handing that VA out immediately. The range becomes reusable only at a recycle epoch, where MOONBRIGHT performs a batched *tlb_flush* before the ring wraps around. This amortizes synchronization across many allocations while keeping the common path flush-free.

Fast Path: Caching Allocator. The first tier of the allocator is designed for the "fast hot path," handling high-frequency allocations. To achieve a low-latency hot path, this tier operates entirely in user space, avoiding the overhead of kernel transitions. Drawing inspiration from the memory management designs of established deep learning frameworks [1, 43], MOONBRIGHT adopts the Best-Fit with Coalescing (BFC) strategy. This approach maintains a user-space memory pool to efficiently satisfy variable-sized requests while actively mitigating fragmentation through immediate block coalescing.

Slow Path: Physical Backing and Orchestration. When the user-space cache is exhausted, or an allocation request exceeds the threshold, the system transitions to the *Slow Path* to acquire fresh backing resources. This process executes a coordinated sequence of primitives: First, the runtime invokes ❶ (`alloc_va`) to reserve a virtual address range from the frontier of the circular buffer and ❷ (`alloc_pfn`) to request physical page frames from the operating system. If the target address range falls within a sparse region of the page table hierarchy, the runtime triggers ❸ (`pgt_create`) to instantiate the necessary page directory structures. Subsequently, the runtime materializes the mapping by launching ❹ (`pte_pop`). This primitive acts as a parallel interpreter, reading mapping directives and populating device-resident page tables at device memory-bandwidth speeds. Since `pte_pop` operates as a standard compute kernel, it follows CUDA stream semantics and can be dependency-tracked together with other GPU work. This allows the page-table population phase to be scheduled asynchronously and, when independent work is available, overlapped with concurrent computation, reducing the exposed latency of mapping construction. Upon *deallocation*, cached blocks are recycled into the Fast Path's free lists rather than being immediately released.

Defragmentation. MOONBRIGHT reduces external fragmentation without data movement. When the allocator lacks a contiguous free block in its pool despite sufficient fragmented physical pages, MOONBRIGHT allocates a fresh contiguous VA range and maps discontiguous physical pages into it. This *logical defragmentation* creates a virtually contiguous buffer from fragmented physical resources solely through parallel page table updates, avoiding the prohibitive overhead of memory compaction and, on the fresh-VA path, TLB flushes.

3.2.3 High-Performance VMM

To provide a familiar programming model while enabling high-performance GPU-centric management, we expose a

driver interface that mirrors the semantics of standard Virtual Memory Management (VMM) APIs. These APIs serve as high-level wrappers around the primitives defined in §3.1, offering granular control over virtual-to-physical mappings while fundamentally redefining the underlying execution mechanism. Consistent with standard protocols like `cuMemMap`, MOONBRIGHT mandates that all operations be aligned to page granularity.

MemAddressReserve. While exporting an interface behaviorally consistent with `cuMemAddressReserve`, this API implements a lightweight booking mechanism via ❶ (alloc_va). It performs logical address reservation by updating the user-space layout tracker, thereby decoupling virtual allocation from physical resource commitment.

MemCreate. This API handles the acquisition of physical backing storage. Corresponding to the primitive ❷(alloc_pfn), it requests a set of physical page frames from the system allocator, ensuring the memory is pinned and accessible to the device. It enforces alignment to the hardware's fixed page granularity (e.g., 2 MB) and retrieves the raw Physical Frame Numbers (PFNs). Instead of keeping these PFNs opaque in the driver, this API exposes them to the runtime's PFN buffer, preparing the data plane for the subsequent, high-bandwidth construction and mapping phases.

MemMap. This API performs the critical binding of physical pages to virtual addresses by orchestrating ❸ (pgt_create) and ❹ (pte_pop). The caller invokes ❺ (tlb_flush) only when the operation can create stale same-VA translations, such as remapping, permission changes, or address reuse. This design addresses the limitations of the split *Map* and *SetAccess* model employed by CUDA and HIP. In existing implementations, the `cuMemSetAccess` call acts as a performance bottleneck by triggering an expensive driver trap. These systems remain constrained by a *CPU-centric control plane*, where access updates require host-side page-table reconstruction, serialized PCIe transfer, and conservative global TLB shootdowns. In contrast, MOONBRIGHT redefines mapping as a high-throughput, data-parallel workload. By offloading these operations to the device and avoiding flushes for fresh mappings, MOONBRIGHT improves mapping latency by orders of magnitude.

De-allocation. To recycle memory resources, MOONBRIGHT provides two complementary primitives: `MemAddressFree` reclaims the virtual address range, while `MemRelease` returns the physical page frames to the system allocator. Crucially, the underlying unmap operation maintains strict TLB coherence, ensuring that stale translations are evicted with the same consistency guarantees as a standard `free` operation.

3.2.4 Stream-Ordered Memory Operations

MemMapAsync. We expose the mapping capability through `MemMapAsync`, which implements page-table updates in a stream-ordered manner. Because `pte_pop` is a normal GPU kernel, mapping tasks are dependency-tracked by the target stream and can overlap with independent compute. For fresh VA ranges, `MemMapAsync` does not require a host-initiated TLB flush. A key advantage of this architecture is the capability to map distinct virtual address ranges to shared physical page frames without data duplication. This mechanism provides zero-copy aliasing of shared physical pages across multiple virtual ranges, which is useful for prefix sharing and beam-search state forking. For operations that overwrite an existing valid mapping, MOONBRIGHT inserts the required coherence point and invokes `tlb_flush`; these same-VA updates are therefore supported, but they are not on the flush-free fast path.

4 Implementation

Our core implementation consists of ≈14.1 KLoC across kernel-space modifications to the open-source NVIDIA [12] and AMD GPU drivers [6], alongside the user-space runtime.

4.1 Adaptation to NVIDIA CUDA GPUs

For NVIDIA CUDA GPUs, our prototype builds on NVIDIA's `open-gpu-kernel-modules` driver, version 560.35.03 [12]. During implementation, we observed two NVIDIA-specific hardware behaviors that affect backend policy choices; we characterize both in §5.4.

First, GSP physical-frame allocation exhibits a size-dependent latency cliff beyond approximately 512 MB. Since MOONBRIGHT does not require physical contiguity, it avoids the cliff by splitting oversized slow-path requests into smaller physical-frame allocation subrequests.

Second, guided by [74], we identify an NVIDIA-specific TLB behavior: aggressive translation prefetching can create *negative caching*, where nearby invalid translations are prefetched and later cause replayable faults. As shown in §5.4, spacing new virtual addresses by the prefetch width avoids this cliff, reducing fault-induced latency by 40.2× at a 32 MB gap. MOONBRIGHT therefore adopts a prefetch-width aligned allocation strategy: new virtual addresses are aligned to 32 MB boundaries so that fresh mappings incur deterministic cold misses rather than costly replayable faults. The 32 MB alignment increases the conservative Robson-style VA budget by 16×, from ≈693 GB to ≈11 TB, still well within the 128 TB user-level VA region exposed by our NVIDIA prototype.

4.2 Adaptation to AMD ROCm GPUs

For AMD GPUs, our prototype builds on AMD's open-source ROCm software stack, version 6.4.0. The AMD backend uses a buddy allocator for physical-frame management. Since we did not observe the large-allocation latency cliff or aggressive

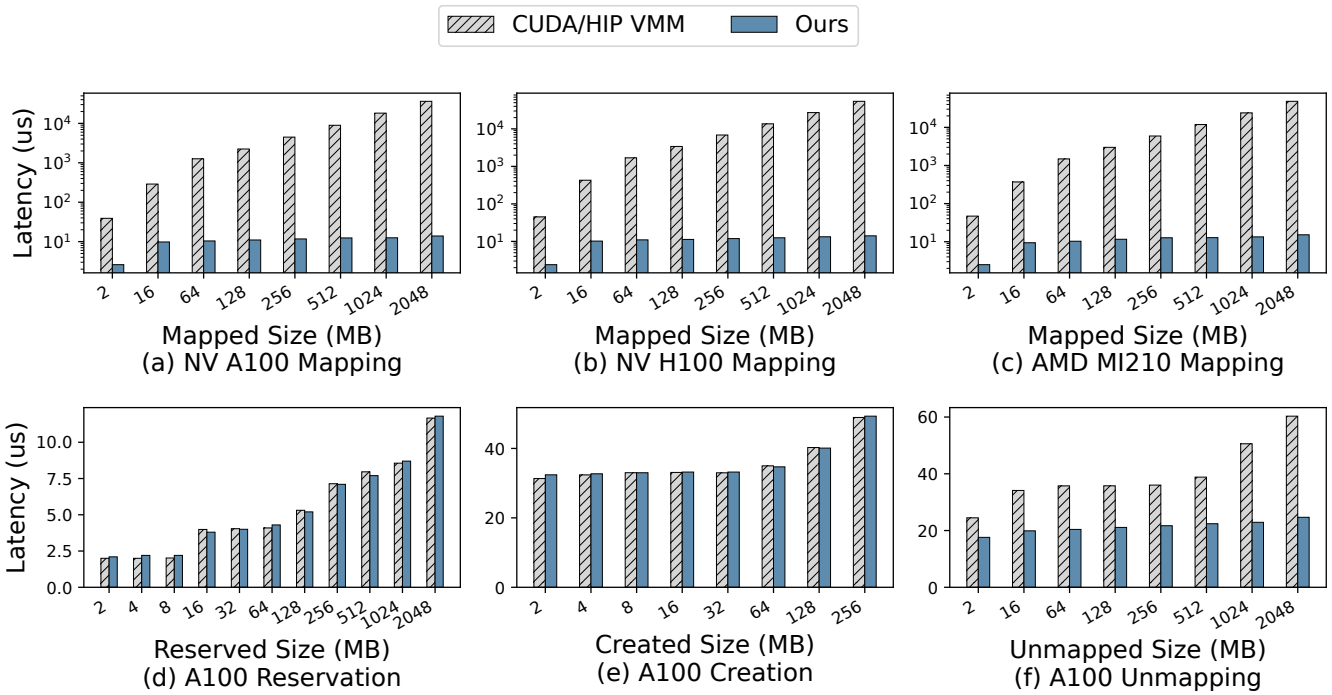


Figure 4: Performance and scalability of low-level virtual-memory APIs across NVIDIA and AMD GPUs.

TLB prefetching seen on NVIDIA GPUs, MOONBRIGHT disables the large-request splitting heuristic and implements the *Always-Fresh VA* strategy without additional virtual-address padding.

5 Evaluation

In this section, we evaluate MOONBRIGHT by answering two research questions:

RQ1: Low-Level Efficiency. Does MOONBRIGHT reduce mapping and allocation overhead across primitives and general GPU workloads while preserving TLB coherence?

RQ2: Application Impact. Do these improvements translate into end-to-end benefits, including lower inference latency, dynamic memory operations, and reduced external fragmentation?

5.1 Experimental Setup

Baselines. We test MOONBRIGHT on a diverse set of GPU architectures to ensure generalizability, including *NVIDIA A100, H100*, and *AMD MI210*. We organize baselines by the functionality and workload targeted by each experiment. For low-level memory primitives, we compare MOONBRIGHT's virtual-memory interface against the vendor CUDA/HIP VMM APIs and runtime allocation primitives such as `cudaMalloc` and `hipMalloc`. For framework-level training workloads, we compare against PyTorch's default caching allocator [43] and GMLake [16], a representative CUDA-VMM-based virtual-memory stitching system for DNN training. For LLM serving workloads, we compare against vAttention [45], which uses VMM-based remapping for KV-cache management, and vLLM [25], which uses software-managed paging.

5.2 Performance of Low-Level APIs

We first evaluate the performance and scalability of MOONBRIGHT's low-level memory-management primitives, which provide functionality comparable to the CUDA/HIP driver VMM APIs.

To ensure a fair comparison against vendor-provided driver-level VMM APIs, we use the same 2 MB page granularity returned as the minimum CUDA VMM granularity on our platform, and vary the mapping range from 2 MB to 2 GB. As shown in Figure 4, MOONBRIGHT consistently outperforms CUDA VMM across all evaluated sizes.

Mapping Cost. For a fresh mapping over a single 2 MB page, MOONBRIGHT reduces mapping latency from ≈ 45 μs with CUDA VMM to 2.6 μs. Here MOONBRIGHT updates only one PTE with one GPU thread, so latency is dominated by fixed device-side mapping overhead, not bulk page-table writes. The gap widens with range size: mapping a fresh 2 GB region takes 36 ms with CUDA VMM, but only 14 μs with MOONBRIGHT, a reduction of over 2,500×.

This improvement comes from a fundamental architectural

shift. As discussed in §3.2.3, CUDA VMM is constrained by a host-mediated control path: page-table construction, state propagation, and conservative synchronization are serialized through the CPU driver and the CPU–GPU interconnect. Even on newer architectures such as H100, page-table updates remain bounded by this off-device control path. In contrast, MOONBRIGHT offloads PTE population to the GPU and treats page-table materialization as a data-parallel device-memory operation, exploiting on-device HBM bandwidth instead of serialized CPU driver loops. Because the mapping operation is expressed as a normal GPU kernel, it also obeys stream ordering: MOONBRIGHT can enqueue the mapping kernel immediately before the computation kernel that consumes the fresh mapping, while independent GPU work in other streams can proceed concurrently.

Initialization Cost. As shown in Figure 4(d) and Figure 4(e), MOONBRIGHT exhibits latency comparable to CUDA VMM for both `reserve` and `create`. This behavior is expected, as these operations remain on the same driver-side critical path: red-black tree manipulation for VA management and GSP invocation for PFN acquisition.

Unmapping Cost. This experiment unmaps each region as a single contiguous range, rather than at a fine-grained 2 MB granularity. As shown in Figure 4(f), under this range-level pattern, MOONBRIGHT consistently outperforms CUDA VMM across region sizes from 2 MB to 2 GB. The absolute latency of CUDA VMM unmapping is much lower than its mapping latency shown in Figure 4(a), because range-level unmapping avoids accumulating fine-grained operation overheads, and invalidating an existing PTE range requires less construction work than installing new physical mappings.

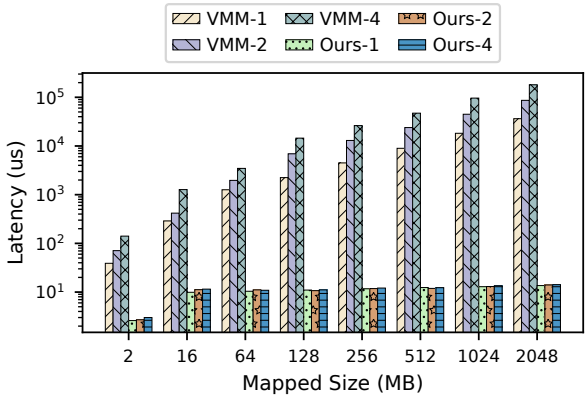
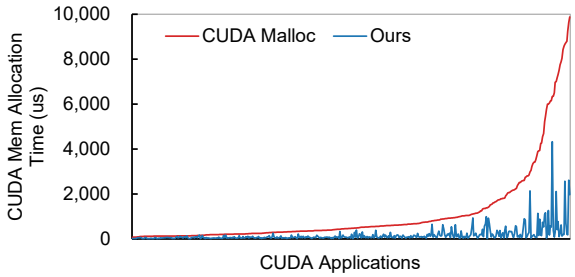
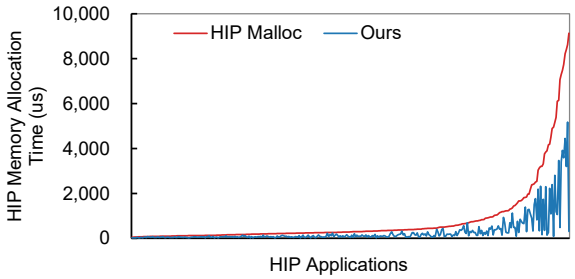


Figure 5: Mapping cost on 1, 2, and 4 A100 GPUs.

Multi-GPU Scalability. We also evaluate the scalability of MOONBRIGHT, which is particularly important for managing memory across multiple GPUs in the era of large language models (LLMs), where VMM is increasingly used to manage and share KV caches [72]. Figure 5 depicts the mapping latency of CUDA VMM and MOONBRIGHT when the number of GPUs ranges from one to four on a single node. When mapping operations are triggered simultaneously across multiple GPUs within a single machine node using CUDA VMM, we observe that the latency grows approximately linearly with the number of devices. For example, the mapping latency grows from 36 ms on one GPU to 86 ms and 180 ms on two and four GPUs, respectively, for a 2 GB mapping. In comparison, MOONBRIGHT remains nearly constant when mapping on multiple GPUs. On a four-GPU setup, MOONBRIGHT is 12,700× faster than CUDA VMM at 2 GB. Because MOONBRIGHT performs mapping operations locally on each GPU in parallel, its latency remains constant regardless of the number of active GPUs. CUDA VMM, in contrast, is severely limited by repeated synchronization operations, such as lock-/unlock in the CUDA driver and the OS kernel, which are required for each page mapping.



(a) On NVIDIA GPU (including CUDA samples and HeCBench).



(b) On AMD GPU (including ROCm samples and HeCBench).

Figure 6: Memory allocation time on different platforms, sorted by baseline allocation latency.

Performance of Allocation We next evaluate a large set of standard benchmarks to quantify the speedup MOONBRIGHT achieves compared to the vendor-provided memory allocators, including `cudaMalloc` and `hipMalloc`. For this large-scale study, we utilize the official *CUDA Samples* [41], *ROCm Samples* [7], and the *HeCBench suite* [22], which collectively cover a diverse range of scientific and compute-intensive kernels for *NVIDIA GPUs* and *AMD GPUs* (via the HIP interface). The combined test suite contains a total of *1,013* distinct test cases, with the HeCBench suite specifically contributing *864* cases, representing a wide variety of allocation patterns

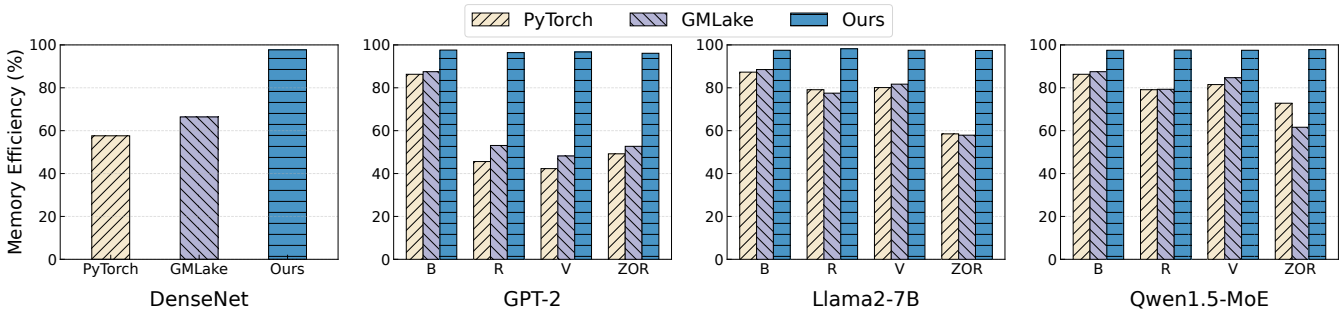


Figure 7: Training memory efficiency on four models across different allocators and optimization combinations: basic (B), recomputation (R), Virtual Pipeline (V), ZeRO (Z), and offload (O).

and memory footprints. This suite provides a broad view of allocation behavior across common GPU workloads.

In each test, we intercept all vendor allocation APIs (e.g., `cudaMalloc`, `hipMalloc`) to measure end-to-end memory-management time. Our objective is to quantify the latency reduction achieved by moving control-plane operations to the data plane. Figure 6 reports absolute memory-management time in μs for MOONBRIGHT and the vendor-provided baseline, with benchmark cases sorted by baseline latency.

As shown in Figure 6, MOONBRIGHT shows consistent latency reductions across NVIDIA and AMD GPUs. On the *NVIDIA GPU* platform, MOONBRIGHT reaches a peak latency reduction of 99.3% and an average reduction of 76.5% across all tested samples. On the *AMD GPU* benchmark suite, MOONBRIGHT shows similar reductions: 98.3% peak and 60.3% on average. We observe a positive correlation between allocation intensity (i.e., size and frequency) and the magnitude of the reduction: benchmarks with heavier memory-management demands benefit more from our approach. These gains come from two effects: MOONBRIGHT performs PTE population on the GPU instead of serializing it through the CPU, and its fresh-VA path avoids unnecessary TLB flushes when no stale same-VA translation can exist.

5.3 Application Impact

5.3.1 Defragmentation for Model Training

We evaluate allocator-level fragmentation in PyTorch-based training stacks and a VMM-based allocator baseline [16, 51, 58] using four NVIDIA A100 GPUs. We use multiple configurations to expose fragmentation under diverse allocation patterns and tensor lifetimes. These stacks combine data, model, and pipeline parallelism [30, 37] with memory-saving techniques such as recomputation (R), virtual pipeline scheduling (V), ZeRO-style distributed optimization (Z), and offload (O) [11, 50, 52]. These techniques reshape tensor lifetimes and allocation sizes, while CUDA tensor allocation remains largely mediated by the framework allocator.

We run DenseNet [44], GPT-2 [49], Llama-2-7B [64], and Qwen1.5-MoE-A2.7B [48], and sweep batch size until out-of-memory. We use *Memory Efficiency*, defined as peak allocated memory over peak reserved memory ($M_{\text{alloc}}/M_{\text{reserved}}$) [16], as a diagnostic metric for allocator-level fragmentation. As shown in Figure 7, MOONBRIGHT improves DenseNet efficiency from 57.6% to 97.7%, and maintains at least 96.1% efficiency on GPT-2, Llama-2-7B, and Qwen1.5-MoE. Under the most dynamic Qwen1.5-MoE Z+O+R setting, PyTorch drops to 72.8%, leaving device memory reserved but unavailable to useful tensor allocations. In contrast, MOONBRIGHT reaches 97.8% by stitching fragmented physical pages into contiguous virtual ranges, effectively reducing external fragmentation. For GMLake, we use a 512 MB granularity as a practical engineering tradeoff between CPU-driven VMM overhead and internal slack [16, 19]; nevertheless, its coarse granularity leaves residual slack under irregular allocations. MOONBRIGHT avoids such slack through fine-grained virtual remapping and preserves end-to-end training performance.

5.3.2 Efficient LLM Inference

We evaluate inference performance on a single NVIDIA A100 GPU across diverse large language model inference scenarios, comparing against vAttention for long-context and prefix-caching scenarios.

Both MOONBRIGHT and vAttention benefit from contiguous, non-paged attention kernels in long-context workloads. As shown in Figure 8, computation increasingly dominates execution at these sequence lengths, reaching 85% at 192K [45]. This leaves limited room for memory-management optimizations; nevertheless, MOONBRIGHT is still up to $\sim$5% faster by eliminating residual host-driven mapping overhead through device-centric page-table updates.

Prefix caching changes this balance. By reusing precomputed KV Cache across requests, it reduces prefill computation and shifts the bottleneck toward memory-management operations that stitch cached blocks into the virtual address space. In this regime, vAttention relies on CPU-driven CUDA VMM and must issue thousands of serialized page mappings, whereas MOONBRIGHT performs these mappings in

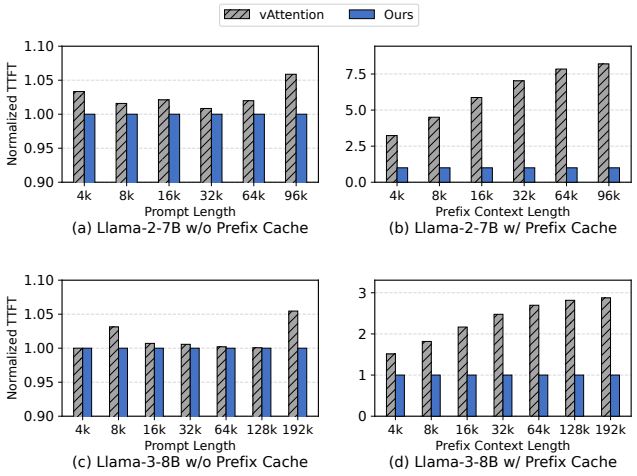


Figure 8: Normalized TTFT for Llama-2-7B and Llama-3-8B under long-context and prefix-caching inference. The first row shows Llama-2-7B; the second row shows Llama-3-8B. For each model, the left panel shows full-prefill long-context requests and the right panel shows prefix-cached requests. Lower is better; all values are normalized to MOONBRIGHT.

microseconds. As a result, MOONBRIGHT achieves up to 8.2× lower TTFT on Llama-2-7B and 2.9× lower TTFT on Llama-3-8B. The difference in gains is primarily explained by the remapping intensity: Llama-3-8B uses grouped-query attention (GQA) [5], which reduces the number of KV heads and shrinks the KV Cache footprint, leaving fewer cached blocks to remap and therefore less CPU-driven VMM overhead for MOONBRIGHT to eliminate.

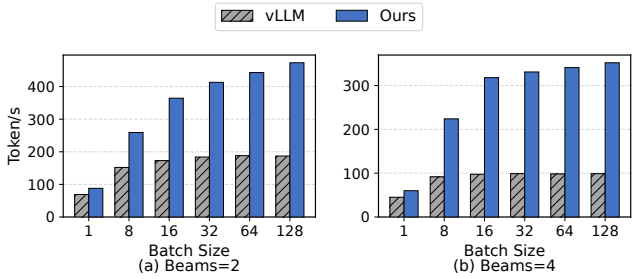


Figure 9: Beam-search inference throughput on Llama-3-8B under beam widths of 2 and 4.

5.3.3 Dynamic Memory Mapping

Finally, we compare Beam Search on MOONBRIGHT against vLLM, a system characterized by step-wise dynamic memory operations governed by data-dependent control flow. In vLLM, each new beam requires a memory fork to update per-sequence states, which is executed on the CPU by deep-copying block metadata. This introduces serialized per-fork overhead, incurring *452* and *780* fork operations per request at beam widths of 2 and 4, respectively. As beam width and batch size increase, this accumulated CPU overhead dominates the runtime. Conversely, MOONBRIGHT offloads fork operations natively to the GPU, aliasing physical pages via lightweight mapping updates. As shown in Figure 9, MOONBRIGHT consistently out-scales vLLM on Llama-3-8B; at batch size 128, MOONBRIGHT achieves 2.5× and 3.6× higher throughput at beam widths of 2 and 4, respectively.

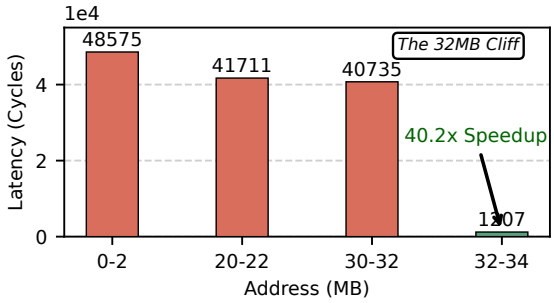


Figure 10: Microbenchmark of zero-flush memory allocation latency. Access latency after a zero-flush fresh mapping is placed at different distances from a previously probed invalid VA. Latency drops after the 32 MB prefetch window, motivating prefetch-width VA padding on NVIDIA.

5.4 Hardware-Guided Policy

MOONBRIGHT's core design, GPU-side page-table materialization and deferred VA reuse, is vendor-agnostic. Platform-specific measurements only tune backend parameters, rather than altering the first principles of the design. In this section, we characterize two NVIDIA-specific effects that inform MOONBRIGHT's backend configuration: TLB prefetching and GSP physical-frame allocation.

TLB prefetching. Figure 10 shows an NVIDIA-specific prefetch cliff. When a fresh VA is placed within the 32 MB prefetch window of a previously probed invalid VA, the first access suffers replayable faults and exceeds 40k cycles. Once the gap reaches 32 MB, latency drops to roughly 1.2k cycles. This observation motivates the 32 MB VA-alignment policy used by MOONBRIGHT on NVIDIA GPUs: new virtual addresses are spaced by the prefetch width so that fresh mappings incur deterministic cold misses rather than replayable faults.

GSP physical-frame allocation. Figure 11 shows another NVIDIA-specific effect: GSP physical-frame allocation remains nearly flat below approximately 512 MB, but increases sharply beyond this point. The undocumented GSP firmware prevents us from identifying the exact cause, but the behavior is consistent with expensive large contiguous-frame allocation. Since MOONBRIGHT does not require physical contiguity, its

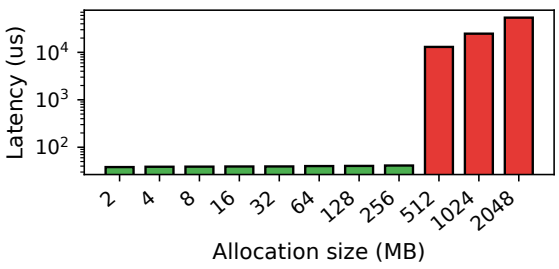


Figure 11: Latency of NVIDIA GSP physical-frame allocation. Allocation latency remains nearly flat below approximately 512 MB and increases sharply beyond this threshold. Green bars are below the observed cliff; red bars are above it.

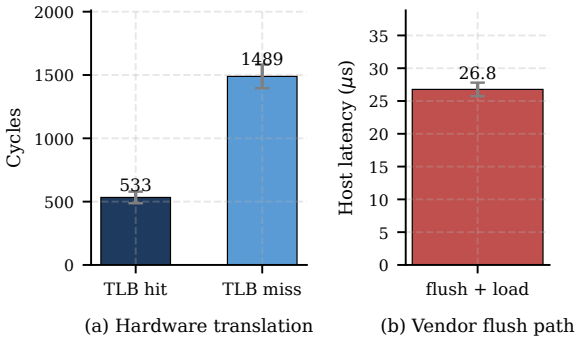


Figure 12: TLB cost decomposition on an NVIDIA A100. **(a)** Device-side cost of a single load under a TLB hit and a forced TLB miss. **(b)** Host-side latency of the conservative vendor flush path.

NVIDIA backend avoids this latency cliff with a simple slow-path heuristic: large physical-frame requests are split into smaller subrequests below the observed threshold. This heuristic only calibrates the backend's physical-frame allocation policy; it does not affect MOONBRIGHT's virtual-memory abstraction or its GPU-side page-table update mechanism.

On platforms where these effects are absent, the corresponding policies can be retuned; for example, our ROCm backend does not use the NVIDIA large-request splitting heuristic.

5.5 Stability of Always-Fresh VA

First-Touch Translation Cost. Always-Fresh VA avoids eager shootdowns for fresh mappings by ensuring that no stale same-address TLB entry can exist. The remaining cost is a first-touch translation by the hardware page walker. Since current GPU counters do not provide a portable way to attribute a specific access to a TLB miss or page walk, we use timing microbenchmarks.

Figure 12(a) shows that modern GPU page walks are fast: on an NVIDIA A100, a pre-warmed TLB hit takes 533 cycles, while a forced miss that walks to a valid PTE takes 1489 cycles, adding only 956 cycles. This miss cost is only one hardware component of the first load. In contrast, Figure 12(b) shows that the conservative CUDA Driver VMM flush path takes 26.8 μs, dominated by host-device synchronization. Thus, Always-Fresh VA replaces a tens-of-microseconds global shootdown on the fresh-allocation path with a small local first-touch translation cost, while deferring global flushes to rare VA-epoch wrap points.

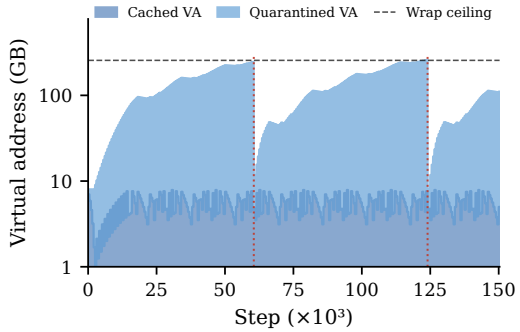


Figure 13: VA footprint under the synthetic stress workload. *Cached VA* denotes reusable VA held by the slow-path block pool, while *Quarantined VA* denotes unmapped VA from evicted segments that awaits batch reclamation at the next epoch boundary.

Quarantine Pressure and Epoch Wrap Behavior. To rigorously evaluate the *Always-Fresh VA* policy under worst-case conditions where allocations are not served from cached blocks, we construct a cache-defeating stress workload. The workload uses non-repeating allocation sizes, extended lifetimes, and high-frequency cadence to defeat cached-block reuse, forcing most requests to reserve fresh VA and advance the frontier at a sustained rate. With a conservative 256 GB wraparound threshold, Figure 13 shows that cached VA remains bounded, while quarantined VA accumulates safely until batch reclamation. At the epoch boundary, MOONBRIGHT performs one batched reclamation with a microsecond-scale `tlb_flush`, as measured in Figure 12. These results show that MOONBRIGHT turns costly per-allocation TLB shootdowns into rare, configurable epoch-boundary events. Furthermore, expanding the epoch size to further reduce flush frequency incurs negligible overhead: flattening a 256 GB VA range requires under 2 MB of page-table memory (using 2 MB pages), and scaling to a 1 TB epoch requires only ∼8 MB.

6 Discussion

This section discusses MOONBRIGHT's scope, limitations, and relationship to existing GPU memory-management mechanisms.

Security and Multi-Tenancy. MOONBRIGHT currently targets trusted and cooperative deployments, such as dedicated LLM inference clusters or single-tenant GPU environments,

where applications and kernels are known to the operator. Under this threat model, compute kernels are assumed to be non-malicious, and page-table update kernels are generated only by the trusted MOONBRIGHT runtime. Extending MOONBRIGHT to untrusted multi-tenant settings raises additional challenges because exposing page-table state to device code creates privileged state that malicious kernels could misuse. MPS does not eliminate this concern, as it targets cooperative multi-process execution rather than protection from malicious device code. MIG provides a stronger hardware partitioning boundary and can help confine cross-instance effects when MOONBRIGHT runs within a single instance. A KASLR-style randomized page-table view can reduce accidental corruption and casual discovery, but is not a security boundary; untrusted deployments still need hidden page-table state, driver ownership checks, capability handles, and sandboxed or verified mapping kernels. We view this as an important open problem for GPU-resident memory management and leave a complete security architecture to future work.

Relation to UVM. NVIDIA Unified Memory (UVM) exposes managed memory through a unified CPU–GPU virtual address space. In demand-paged mode, GPU accesses to non-resident managed pages trigger hardware fault reporting; the driver resolves residency and coherence, updates mappings, and replays the faulting access. MOONBRIGHT is complementary to UVM rather than a replacement. UVM targets transparent host–device sharing through a fault-driven path, whereas MOONBRIGHT targets latency-critical device-resident memory by proactively installing stream-ordered mappings and materializing PTEs on the GPU. Applications can therefore use UVM-backed regions for transparent sharing and MOONBRIGHT-backed regions for low-latency GPU-resident allocation and remapping within the same CUDA context.

7 Related Work

GPU allocators and memory managers. Early GPU allocators such as ScatterAlloc [62] and XMalloc [18] target fine-grained, kernel-initiated allocation for massively parallel programs. They are effective for dynamic parallelism but do not match the dominant memory-management pattern in DL frameworks, where the host/runtime allocates coarse tensors and KV-cache regions. Gallatin [36] improves general-purpose allocation on the device, but it does not address the CPU-driven virtual-memory control plane that dominates page-granular remapping latency. FineMem [66] reduces memory waste in fine-grained disaggregated memory, while Unified Memory simplifies programming by providing a shared address space; prior studies show that UVM can introduce unpredictable transfer and synchronization overheads for latency-sensitive workloads [26, 29]. MOONBRIGHT is complementary: it focuses on making GPU page-table updates low-latency and stream-compatible on commodity devices.

Defragmentation and virtual-memory stitching. Classic defragmentation and compaction techniques move live objects to recover contiguous space [35, 59, 65]. Such approaches are portable but expensive for GPU workloads because they require data movement, pointer rewriting, and synchronization. Model-specific approaches such as memory-efficient DenseNet training [44] hand-tune buffer sharing to reduce activation memory, but do not provide general allocator-level defragmentation. Recent DL systems and allocators address waste through virtual-memory stitching or planning: PyTorch's experimental expandable segments grow VMM-backed segments to reduce allocation slivers [46], GMLake [16] applies CUDA VMM to DNN training, vAttention [45] remaps KV caches, and STAlloc [19] plans training allocations over space and time. These systems demonstrate the value of memory-aware allocation, but they inherit CPU-centric mapping costs or operate above the hardware mapping substrate. MOONBRIGHT targets the root cause by materializing PTEs on the GPU and deferring TLB coherence when mappings use fresh VAs.

LLM serving memory systems. LLM serving systems optimize memory use and scheduling at several layers. ORCA [70], FastServe [68], and Sarathi [4] improve batching, prefill/decode scheduling, or distributed serving. FlexGen [57] and PowerInfer [61] target memory-constrained serving through offloading or hybrid CPU/GPU execution. Mooncake [47], Jenga [73], CacheGen [33], Pensieve [71], and DroidSpeak [32] optimize KV-cache placement, reuse, compression, streaming, or sharing. vLLM [25] introduced PagedAttention for block-based KV-cache management, while S-LoRA [56] and Punica [10] manage serving many LoRA adapters. These systems are effective above the memory-management substrate, but their mechanisms are tied to model/runtime semantics and often require specialized kernels or metadata paths. MOONBRIGHT instead accelerates the hardware MMU mapping path, preserving hardware-level virtual contiguity while supporting dynamic remapping and sharing.

8 Conclusion

GPU memory management remains largely host-driven, leaving device parallelism unused on the allocation critical path. This paper presented MOONBRIGHT, a GPU-centric memory-management substrate that materializes page-table updates on the GPU and avoids unnecessary TLB shootdowns with *Always-Fresh* virtual addressing. By turning translation updates into data-parallel device work, MOONBRIGHT reduces mapping latency by up to three orders of magnitude and improves allocation latency and memory efficiency across training and inference workloads.

Acknowledgment

We would like to extend our most sincere gratitude to our shepherd and reviewers for their feedback and suggestions. This work is supported by the National Key Research and Development Program of China (2024YFB4505603), the National Natural Science Foundation of China (62232015, 62302479).

References

[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. TensorFlow: A system for Large-Scale machine learning. In *12th USENIX symposium on operating systems design and implementation (OSDI 16)*, pages 265–283, 2016.

[2] Mark James Abraham, Teemu Murtola, Roland Schulz, Szilárd Páll, Jeremy C Smith, Berk Hess, and Erik Lindahl. GROMACS: High performance molecular simulations through multi-level parallelism from laptops to supercomputers. *SoftwareX*, 1:19–25, 2015.

[3] Bilge Acun, David J Hardy, Laxmikant V Kale, Keqin Li, James C Phillips, and John E Stone. Scalable molecular dynamics with NAMD on the summit system. *IBM journal of research and development*, 62(6):4–1, 2018.

[4] Amey Agrawal, Nitin Kedia, Ashish Panwar, Jayashree Mohan, Nipun Kwatra, Bhargav Gulavani, Alexey Tumanov, and Ramachandran Ramjee. Taming Throughput-Latency tradeoff in LLM inference with Sarathi-Serve. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, pages 117–134, 2024.

[5] Joshua Ainslie, James Lee-Thorp, Michiel de Jong, Yury Zemlyanskiy, Federico Lebron, and Sumit Sanghai. GQA: Training Generalized Multi-Query Transformer Models from Multi-Head Checkpoints. In Houda Bouamor, Juan Pino, and Kalika Bali, editors, *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pages 4895–4901, Singapore, December 2023. Association for Computational Linguistics.

[6] AMD. ROCm documentation - virtual memory management. https://rocm.docs.amd.com/projects/HIP/en/latest/doxygen/html/group___virtual.html, 2025. Accessed: 2025.12.09.

[7] AMD ROCm Developer Tools. HIP Examples and Samples. https://github.com/ROCm/rocm-examples, 2025. Accessed: 2025.12.09.

[8] Jeff Bonwick. The slab allocator: an object-caching kernel memory allocator. In *Proceedings of the USENIX Summer 1994 Technical Conference on USENIX Summer 1994 Technical Conference - Volume 1*, USTC'94, page 6, USA, 1994. USENIX Association.

[9] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language models are few-shot learners. In *Proceedings of the 34th International Conference on Neural Information Processing Systems*, NIPS '20, Red Hook, NY, USA, 2020. Curran Associates Inc.

[10] Lequn Chen, Zihao Ye, Yongji Wu, Danyang Zhuo, Luis Ceze, and Arvind Krishnamurthy. Punica: Multi-Tenant LoRA Serving. In Phillip B. Gibbons, Gennady Pekhimenko, and Christopher De Sa, editors, *Proceedings of the Seventh Annual Conference on Machine Learning and Systems, MLSys 2024, Santa Clara, CA, USA, May 13-16, 2024*. mlsys.org, 2024.

[11] Tianqi Chen, Bing Xu, Chiyuan Zhang, and Carlos Guestrin. Training deep nets with sublinear memory cost. *arXiv preprint arXiv:1604.06174*, 2016.

[12] NVIDIA Corporation. NVIDIA linux open GPU kernel module source. https://github.com/NVIDIA/open-gpu-kernel-modules, 2025. Accessed: 2025.12.09.

[13] Alexey Dosovitskiy, Lucas Beyer, Alexander Kolesnikov, Dirk Weissenborn, Xiaohua Zhai, Thomas Unterthiner, Mostafa Dehghani, Matthias Minderer, Georg Heigold, Sylvain Gelly, Jakob Uszkoreit, and Neil Houlsby. An image is worth 16x16 words: Transformers for image recognition at scale. In *International Conference on Learning Representations*, 2021.

[14] Jason Evans. A scalable concurrent malloc (3) implementation for freebsd. In *Proc. of the bsdcan conference, ottawa, canada*, 2006.

[15] Wolfram Gloger et al. Dynamic memory allocator implementations in linux system libraries. http://www.malloc.de/papers/malloc-slides, 1997. Accessed: 2025.12.09.

[16] Cong Guo, Rui Zhang, Jiale Xu, Jingwen Leng, Zihan Liu, Ziyu Huang, Minyi Guo, Hao Wu, Shouren Zhao, Junping Zhao, and Ke Zhang. GMLake: Efficient and

Transparent GPU Memory Defragmentation for Large-scale DNN Training with Virtual Memory Stitching. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, ASPLOS '24, page 450–466, New York, NY, USA, 2024. Association for Computing Machinery.

[17] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 770–778, 2016.

[18] Xiaohuang Huang, Christopher I Rodrigues, Stephen Jones, Ian Buck, and Wen-mei Hwu. XMalloc: A scalable lock-free dynamic memory allocator for many-core machines. In *2010 10th IEEE International Conference on Computer and Information Technology*, pages 1134–1139. IEEE, 2010.

[19] Zixiao Huang, Junhao Hu, Hao Lin, Chunyang Zhu, Yueran Tang, Quanlu Zhang, Zhen Guo, Zhenhua Li, Shengen Yan, Zhenhua Zhu, et al. Stalloc: Enhancing memory efficiency in large-scale model training with spatio-temporal planning. *arXiv preprint arXiv:2507.16274*, 2025.

[20] Andrew Hamilton Hunter, Chris Kennelly, Paul Turner, Darryl Gove, Tipp Moseley, and Parthasarathy Ranganathan. Beyond malloc efficiency to fleet efficiency: a hugepage-aware memory allocator. In *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*, pages 257–273, 2021.

[21] Marcus Jägemar. Mallocpool: Improving memory performance through contiguously tlb mapped memory. In *2018 IEEE 23rd International Conference on Emerging Technologies and Factory Automation (ETFA)*, volume 1, pages 1127–1130. IEEE, 2018.

[22] Zheming Jin and Jeffrey S Vetter. A benchmark suite for improving performance portability of the sycl programming model. In *2023 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 325–327. IEEE, 2023.

[23] Vivek Kini and Jake Hemstad. Using the NVIDIA CUDA Stream-Ordered Memory Allocator, Part 2. https://developer.nvidia.com/blog/using-cuda-stream-ordered-memory-allocator-part-2/, July 2021. Accessed: 2025.12.09.

[24] Bradley C Kuszmaul. Supermalloc: a super fast multithreaded malloc for 64-bit machines. *ACM SIGPLAN Notices*, 50(11):41–55, 2015.

[25] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph E. Gonzalez, Hao Zhang, and Ion Stoica. Efficient Memory Management for Large Language Model Serving with PagedAttention. In *Proceedings of the ACM SIGOPS 29th Symposium on Operating Systems Principles*, 2023.

[26] Ruihao Li, Bagus Hanindhito, Sanjana Yadav, Qinzhe Wu, Krishna Kavi, Gayatri Mehta, Neeraja J Yadwadkar, and Lizy K John. Performance implications of pipelining the data transfer in CPU-GPU heterogeneous systems. *ACM Transactions on Architecture and Code Optimization*, 2025.

[27] Ruihao Li, Qinzhe Wu, Krishna Kavi, Gayatri Mehta, Jonathan C Beard, Neeraja J Yadwadkar, and Lizy K John. Speedmalloc: Improving multi-threaded applications via a lightweight core for memory allocation. *arXiv preprint arXiv:2508.20253*, 2025.

[28] Ruihao Li, Qinzhe Wu, Krishna Kavi, Gayatri Mehta, Neeraja J Yadwadkar, and Lizy K John. NextGen-Malloc: Giving memory allocator its own room in the house. In *Proceedings of the 19th Workshop on Hot Topics in Operating Systems*, pages 135–142, 2023.

[29] Ruihao Li, Sanjana Yadav, Qinzhe Wu, Krishna Kavi, Gayatri Mehta, Neeraja J Yadwadkar, and Lizy K John. Performance implications of async memcpy and uvm: a tale of two data transfer modes. In *2023 IEEE International Symposium on Workload Characterization (IISWC)*, pages 115–127. IEEE, 2023.

[30] Shen Li, Yanli Zhao, Rohan Varma, Omkar Salpekar, Pieter Noordhuis, Teng Li, Adam Paszke, Jeff Smith, Brian Vaughan, Pritam Damania, et al. PyTorch Distributed: Experiences on accelerating data parallel training. *arXiv preprint arXiv:2006.15704*, 2020.

[31] Hang Liu and H Howie Huang. Enterprise: Breadth-first graph traversal on GPUs. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–12, 2015.

[32] Yuhan Liu, Yuyang Huang, Jiayi Yao, Shaoting Feng, Zhuohan Gu, Kuntai Du, Hanchen Li, Yihua Cheng, Junchen Jiang, Shan Lu, et al. DroidSpeak: KV cache sharing for cross-LLM communication and multi-LLM serving. *arXiv preprint arXiv:2411.02820*, 2024.

[33] Yuhan Liu, Hanchen Li, Yihua Cheng, Siddhant Ray, Yuyang Huang, Qizheng Zhang, Kuntai Du, Jiayi Yao, Shan Lu, Ganesh Ananthanarayanan, et al. CacheGen: KV cache compression and streaming for fast large language model serving. In *Proceedings of the ACM SIGCOMM 2024 Conference*, pages 38–56, 2024.

[34] Clemens Lutz, Sebastian Breß, Steffen Zeuch, Tilmann Rabl, and Volker Markl. Pump up the volume: Processing large data on GPUs with fast interconnects. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, SIGMOD '20, page 1633–1649, New York, NY, USA, 2020. Association for Computing Machinery.

[35] Simon Marlow, Tim Harris, Roshan P James, and Simon Peyton Jones. Parallel generational-copying garbage collection with a block-structured heap. In *Proceedings of the 7th international symposium on Memory management*, pages 11–20, 2008.

[36] Hunter Mccoy and Prashant Pandey. Gallatin: A general-purpose GPU memory manager. In *Proceedings of the 29th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming*, PPoPP '24, page 364–376, New York, NY, USA, 2024. Association for Computing Machinery.

[37] Deepak Narayanan, Mohammad Shoeybi, Jared Casper, Patrick LeGresley, Mostofa Patwary, Vijay Korthikanti, Dmitri Vainbrand, Prethvi Kashinkunti, Julie Bernauer, Bryan Catanzaro, et al. Efficient large-scale language model training on GPU clusters using Megatron-LM. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–15, 2021.

[38] Amir Hossein Nodehi Sabet, Junqiao Qiu, and Zhijia Zhao. Tigr: Transforming irregular graphs for GPU-friendly graph processing. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 622–636, 2018.

[39] NVIDIA. CUDA Driver API: Virtual Memory Management (VMM) Documentation. https://docs.nvidia.com/cuda/cuda-driver-api/group__CUDA__VA.html, 2025. Accessed: 2025.12.09.

[40] NVIDIA. NVIDIA HGX Platform. https://www.nvidia.com/en-us/data-center/hgx/, 2025. Accessed: 2025.12.09.

[41] NVIDIA Corporation. CUDA Samples. https://github.com/NVIDIA/cuda-samples, 2025. Accessed: 2025.12.09.

[42] NVIDIA Corporation. RAPIDS: Open GPU data science. https://rapids.ai/, 2025. Accessed: 2025.12.09.

[43] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. PyTorch: An imperative style, high-performance deep learning library. *Advances in neural information processing systems*, 32, 2019.

[44] Geoff Pleiss, Danlu Chen, Gao Huang, Tongcheng Li, Laurens van der Maaten, and Kilian Q. Weinberger. Memory-efficient implementation of DenseNets. *arXiv preprint arXiv:1707.06990*, 2017.

[45] Ramya Prabhu, Ajay Nayak, Jayashree Mohan, Ramachandran Ramjee, and Ashish Panwar. vAttention: Dynamic memory management for serving LLMs without PagedAttention. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1*, pages 1133–1150, 2025.

[46] PyTorch Contributors. PyTorch CUDA semantics: memory management and expandable segments. https://docs.pytorch.org/docs/stable/notes/cuda.html#optimizing-memory-usage-with-pytorch-cuda-alloc-conf, 2025. Accessed: 2025.12.09.

[47] Ruoyu Qin, Zheming Li, Weiran He, Jialei Cui, Feng Ren, Mingxing Zhang, Yongwei Wu, Weimin Zheng, and Xinran Xu. Mooncake: Trading more storage for less computation—a KVCache-centric architecture for serving LLM chatbot. In *23rd USENIX Conference on File and Storage Technologies (FAST 25)*, pages 155–170, 2025.

[48] Qwen Team. Qwen1.5-MoE-A2.7B. https://huggingface.co/Qwen/Qwen1.5-MoE-A2.7B, 2024. Accessed: 2025.12.09.

[49] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. Language models are unsupervised multitask learners. *OpenAI Technical Report*, 2019.

[50] Samyam Rajbhandari, Jeff Rasley, Olatunji Ruwase, and Yuxiong He. ZeRO: Memory optimizations toward training trillion parameter models. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–16. IEEE, 2020.

[51] Jeff Rasley, Samyam Rajbhandari, Olatunji Ruwase, and Yuxiong He. DeepSpeed: System optimizations enable training deep learning models with over 100 billion parameters. In *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pages 3505–3506, 2020.

[52] Jie Ren, Samyam Rajbhandari, Reza Yazdani Aminabadi, Olatunji Ruwase, Shuangyan Yang, Minjia Zhang, Dong Li, and Yuxiong He. ZeRO-Offload: Democratizing Billion-Scale model training. In *2021*

USENIX Annual Technical Conference (USENIX ATC 21), pages 551–564, 2021.

[53] J. M. Robson. Bounds for some functions concerning dynamic storage allocation. *J. ACM*, 21(3):491–499, July 1974.

[54] Christopher J Rossbach, Jon Currey, Mark Silberstein, Baishakhi Ray, and Emmett Witchel. PTask: Operating system abstractions to manage GPUs as compute devices. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, pages 233–248, 2011.

[55] Romelia Salomon-Ferrer, Andreas W Gotz, Duncan Poole, Scott Le Grand, and Ross C Walker. Routine microsecond molecular dynamics simulations with AMBER on GPUs. 2. explicit solvent particle mesh ewald. *Journal of chemical theory and computation*, 9(9):3878–3888, 2013.

[56] Ying Sheng, Shiyi Cao, Dacheng Li, Coleman Hooper, Nicholas Lee, Shuo Yang, Christopher Chou, Banghua Zhu, Lianmin Zheng, Kurt Keutzer, Joseph Gonzalez, and Ion Stoica. SLoRA: Scalable Serving of Thousands of LoRA Adapters. In P. Gibbons, G. Pekhimenko, and C. De Sa, editors, *Proceedings of Machine Learning and Systems*, volume 6, pages 296–311, 2024.

[57] Ying Sheng, Lianmin Zheng, Binhang Yuan, Zhuohan Li, Max Ryabinin, Beidi Chen, Percy Liang, Christopher Ré, Ion Stoica, and Ce Zhang. FlexGen: High-throughput generative inference of large language models with a single GPU. In *Proceedings of the 40th International Conference on Machine Learning*, pages 31094–31116, 2023.

[58] Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. Megatron-LM: Training multi-billion parameter language models using model parallelism. *arXiv preprint arXiv:1909.08053*, 2019.

[59] David Siegwart and Martin Hirzel. Improving locality with parallel hierarchical copying gc. In *Proceedings of the 5th international symposium on Memory management*, pages 52–63, 2006.

[60] Mark Silberstein, Bryan Ford, Idit Keidar, and Emmett Witchel. GPUfs: Integrating a file system with GPUs. In *Proceedings of the eighteenth international conference on Architectural Support for Programming Languages and Operating Systems*, pages 485–498, 2013.

[61] Yixin Song, Zeyu Mi, Haotong Xie, and Haibo Chen. PowerInfer: Fast large language model serving with a consumer-grade GPU. In *Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles*, pages 590–606, 2024.

[62] Markus Steinberger, Michael Kenzel, Bernhard Kainz, and Dieter Schmalstieg. Scatteralloc: Massively parallel dynamic memory allocation for the gpu. In *2012 Innovative Parallel Computing (InPar)*, pages 1–10. IEEE, 2012.

[63] Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, et al. Llama: Open and efficient foundation language models. *arXiv preprint arXiv:2302.13971*, 2023.

[64] Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, et al. Llama 2: Open foundation and fine-tuned chat models. *arXiv preprint arXiv:2307.09288*, 2023.

[65] Ronald Veldema and Michael Philippsen. Parallel memory defragmentation on a GPU. In *Proceedings of the 2012 ACM SIGPLAN Workshop on Memory Systems Performance and Correctness*, pages 38–47, 2012.

[66] Xiaoyang Wang, Yongkun Li, Kan Wu, Wenzhe Zhu, Yuqi Li, and Yinlong Xu. FineMem: Breaking the allocation overhead vs. memory waste dilemma in Fine-Grained disaggregated memory management. In *19th USENIX Symposium on Operating Systems Design and Implementation (OSDI 25)*, pages 57–74, 2025.

[67] Yangzihao Wang, Andrew Davidson, Yuechao Pan, Yuduo Wu, Andy Riffel, and John D Owens. Gunrock: A high-performance graph processing library on the GPU. In *Proceedings of the 21st ACM SIGPLAN symposium on principles and practice of parallel programming*, pages 1–12, 2016.

[68] Bingyang Wu, Yinmin Zhong, Zili Zhang, Shengyu Liu, Fangyue Liu, Yuanhang Sun, Gang Huang, Xuanzhe Liu, and Xin Jin. Fast distributed inference serving for large language models. *arXiv preprint arXiv:2305.05920*, 2023.

[69] Rio Yokota and Lorena A Barba. Treecode and fast multipole method for N-body simulation with CUDA. In *GPU Computing Gems Emerald Edition*, pages 113–132. Elsevier, 2011.

[70] Gyeong-In Yu, Joo Seong Jeong, Geon-Woo Kim, Soojeong Kim, and Byung-Gon Chun. Orca: A distributed serving system for Transformer-Based generative models. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 521–538, 2022.

[71] Lingfan Yu, Jinkun Lin, and Jinyang Li. Stateful large language model serving with pensieve. In *Proceedings of the Twentieth European Conference on Computer Systems*, pages 144–158, 2025.

[72] Shan Yu, Jiarong Xing, Yifan Qiao, Mingyuan Ma, Yangmin Li, Yang Wang, Shuo Yang, Zhiqiang Xie, Shiyi Cao, Ke Bao, et al. Prism: Unleashing GPU sharing for cost-efficient multi-LLM serving. *arXiv preprint arXiv:2505.04021*, 2025.

[73] Chen Zhang, Kuntai Du, Shu Liu, Woosuk Kwon, Xiangxi Mo, Yufeng Wang, Xiaoxuan Liu, Kaichao You, Zhuohan Li, Mingsheng Long, et al. Jenga: Effective memory management for serving LLM with heterogeneity. In *Proceedings of the ACM SIGOPS 31st Symposium on Operating Systems Principles*, pages 446–461, 2025.

[74] Zhenkai Zhang, Tyler Allen, Fan Yao, Xing Gao, and Rong Ge. Tunnels for Bootlegging: Fully Reverse-Engineering GPU TLBs for Challenging Isolation Guarantees of NVIDIA MIG. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*, CCS '23, page 960–974, New York, NY, USA, 2023. Association for Computing Machinery.

[75] Jianlong Zhong and Bingsheng He. Medusa: A parallel graph processing system on graphics processors. *ACM SIGMOD Record*, 43(2):35–40, 2014.

ARTIFACT EVALUATED
USENIX
AVAILABLE

Nixie: Efficient, Transparent Temporal Multiplexing for Consumer GPUs

Yechen Xu Yifei Wang Nathanael Ren Yiran Chen Danyang Zhuo

Duke University

Abstract

Consumer machines are increasingly running large ML workloads such as large language models (LLMs), text-to-image generation, and interactive image editing. Unlike datacenter GPUs, consumer GPUs serve single-user, rapidly changing workloads, and each model's working set often nearly fills the GPU memory. As a result, existing sharing mechanisms, such as NVIDIA Unified Virtual Memory (UVM), suffer from severe memory thrashing and consume excessive CPU pinned memory when multiple applications are active.

We design and implement Nixie, a system service that enables efficient and transparent temporal multiplexing on consumer GPUs without requiring any application or driver changes. Nixie coordinates GPU memory allocation and kernel launch behavior to efficiently utilize the CPU-GPU bi-directional bandwidth and CPU pinned memory. A lightweight scheduler in Nixie further improves responsiveness by automatically prioritizing latency-sensitive interactive jobs using MLFQ-inspired techniques. Our evaluations show that, compared to UVM-based approaches, Nixie improves the latency of interactive code-completion tasks co-located with a long-running LLM by up to 3.8× and reduces CPU pinned memory usage by up to 66.8% under the same latency requirement.

1 Introduction

Modern consumer machines increasingly run large and sophisticated machine learning (ML) workloads. Consumers now routinely run large language models (LLMs) [29, 37], text-to-image diffusion models [18], interactive image editing [35], and video processing [5, 32] locally on their desktops equipped with consumer GPUs, such as NVIDIA's RTX 4090 and 5090. This trend is driven by the desire for privacy, interactivity, and reduced hardware costs, as well as by rapid advances in model quantization and optimized inference runtimes [13, 17, 42] that make local ML execution practical.

Despite this wave of local ML execution, today's consumer GPUs are fundamentally designed for a single active application. In contrast to datacenter GPUs, where workloads are typically long-running, batched, and multi-tenant, consumer workloads are rapidly changing, heterogeneous, and strongly user-driven. A single user may frequently switch between very different models and applications: querying an LLM, generating an image, editing a photo, or running a background batch job such as OCR or video processing. These applications interact in two common ways. Some form dependent workflows, such as using an LLM to rewrite a prompt before invoking a diffusion model. Others run independently but compete for responsiveness, such as interactive code completion while an autonomous agent or batched generation job continues in the background. Batch sizes for interactive applications are almost always one, and interactive workloads are expected to provide low-latency responses. These characteristics create a unique resource-management environment that differs substantially from datacenter GPUs.

A central challenge is that *the working set of each ML application often nearly saturates GPU memory*. Models such as modern LLMs or diffusion pipelines require tens of gigabytes of active parameters, activations, and temporary workspace. Users often prefer more accurate and larger models. As a result, when two or more applications run concurrently, the combined working set trivially exceeds GPU memory.

Existing GPU multiplexing mechanisms perform poorly under these conditions. Naive stop-and-restart approaches such as llama-swap [21] introduce substantial delays: each switch requires restarting applications and reloading large models, leading to multi-second or even longer response times. Application-level model swapping (e.g., Ollama [25]) is restricted to a single application and cannot coordinate with other applications. Prior systems such as Prism [39] and Aegaeon [36] require manual integration and primarily target scenarios with many small models sharing a large GPU, which does not reflect the consumer GPU setting where each model's working set nearly fills the GPU memory.

NVIDIA Unified Virtual Memory (UVM) [1] provides a form of transparent GPU multiplexing, but it relies on demand paging and implicitly assumes that the combined working set

fits in GPU memory to sustain good performance. When multiple large applications are active, UVM exhibits severe thrashing: pages repeatedly migrate between GPU and CPU memory, causing drastic throughput degradation and large, unpredictable latency spikes. Systems such as TGS [34] and nvshare [3] mitigate some of UVM's thrashing behavior, but their workload assumptions remain restrictive and do not reflect consumer GPU usage patterns. Moreover, they inherit UVM's fundamental limitations, including excessive use of CPU pinned memory, which significantly limits scalability for large, memory-intensive models.

We design and implement Nixie, a system service that provides efficient and transparent temporal multiplexing on consumer GPUs without requiring any application or driver changes. Nixie interposes on GPU memory allocation and kernel-launch behavior. The key idea behind Nixie is to explicitly control when applications occupy GPU memory and when they yield it, ensuring that a single application's working set fully resides in GPU memory at any given moment. This avoids the pathological thrashing behavior inherent to UVM and maintains predictable performance for each application. For fast context switching, Nixie fully utilizes the bi-directional bandwidth between GPU and CPU.

Nixie also incorporates a lightweight scheduler that automatically identifies latency-sensitive interactive applications and prioritizes them, without requiring any user annotations for application priorities. Nixie uses an MLFQ-like scheduler. It dynamically adapts application priorities by tracking applications' CUDA kernel launch patterns and prioritizes applications that can finish execution within their allocated time window. Lower-priority applications have larger time windows to minimize the context switch frequency and amortize scheduling overhead.

We evaluate Nixie on a consumer-grade NVIDIA RTX 5090 GPU using widely adopted applications (e.g., `llama.cpp` [13], Ollama [25], SGLang [42], ComfyUI [9]) and popular models (e.g., Qwen3 [37], Gemma3 [29], Z-Image [31], Qwen-Image [35]). Compared to application-managed solutions such as Ollama, as well as UVM and the UVM-based GPU multiplexing system nvshare [3], Nixie reduces context-switch overhead by 29.7-82.3% across configurations and cuts CPU pinned memory usage by up to 66.8%. Nixie delivers 1.3-1.6× speedups on diverse multi-application workloads (e.g., using an LLM to generate prompts for image generation). Nixie also improves interactive code-completion latency by 3.1-3.8×, while maintaining competitive throughput on background batch-processing workloads. Our source is available at https://github.com/XOR-op/Nixie.

Our paper makes the following contributions:

- A new system architecture that jointly manages GPU memory usage and kernel dispatch, eliminating UVM-style thrashing without requiring any driver or application changes. The new architecture efficiently uses CPU pinned memory and bi-directional bandwidth between CPU and GPU.
- A lightweight, MLFQ-inspired scheduler for GPU workloads that automatically identifies and prioritizes interactive applications to reduce latency while preserving overall GPU throughput.
- An implementation that supports unmodified, state-of-the-art GPU applications (e.g., `llama.cpp`, SGLang, and ComfyUI) and delivers substantial improvements across diverse consumer GPU workloads.

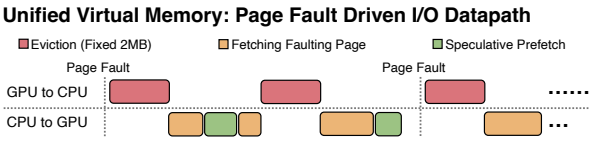


Figure 1: UVM's page-fault-driven migration. UVM can only utilize half of the link bandwidth for full-duplex PCIe channel.

2 Motivation

Here we motivate why the consumer GPU multiplexing problem is a fundamentally different problem from datacenter GPU multiplexing.

2.1 Local ML Workloads on Consumer GPUs

Recent advances in model quantization and optimized inference runtimes have enabled users to run large machine learning (ML) models directly on their personal machines. Applications such as local LLM inference (e.g., `llama.cpp`, SGLang, vLLM), text-to-image generation, and interactive image editing pipelines (e.g., ComfyUI, diffusion-based tools) have become increasingly common. Users are motivated by privacy, offline capability, more fine-tuning opportunities and reduced hardware costs.

Unlike traditional datacenter GPU workloads, a single user may concurrently run or frequently switch between multiple large models: querying an LLM, generating an image, editing a photo, running background AI tasks, among other applications, all on a shared consumer-grade GPU. For interactive applications, the batch size is almost always one. At the same time, modern consumer GPUs (e.g., RTX 4090/5090) offer substantial compute but limited GPU memory relative to model size. Users often run the largest models that can fit in memory to maximize quality. However, this means each individual application may nearly saturate GPU memory.

2.2 Limitations of Existing Multiplexing Approaches on Consumer GPUs

Existing Approach #1: Shut Down an Application and Run the Next Application. A straightforward solution is to

run only one application at a time. For example, the community tool `llama-swap` [21] enables switching by terminating the current application and launching the next. However, unloading and reloading large models introduces multi-second delays, making even moderate task switching impractical for interactive consumer workflows (e.g., prompting an LLM to generate image descriptions that will be used immediately in a diffusion-based text-to-image application).

Existing Approach #2: Application-Specific Approaches. Applications such as Ollama [25] provide model switching by unloading the current model and loading a new one. However, this mechanism is tightly coupled to a single application's internal execution and cannot coordinate with other GPU applications.

Another line of work makes an inference system's GPU memory usage elastic. Prism [39], for example, allows an LLM framework to dynamically shrink its KV cache, freeing GPU memory for other applications. While effective for LLMs, this approach is highly domain-specific: diffusion models, image-editing pipelines, and many other consumer workloads do not use KV caches and therefore cannot benefit from this mechanism. Moreover, these systems require explicit software-level integration into each application or inference framework. As a result, they cannot provide transparent, system-wide multiplexing across the diverse set of applications on consumer GPUs.

Existing Approach #3: NVIDIA Unified Virtual Memory (UVM). UVM is a fully transparent mechanism for handling GPU memory oversubscription. Its basic idea mirrors CPU demand paging: when a GPU kernel accesses a page that is not resident in GPU memory, the hardware raises a page fault interrupt to the driver. The driver's handler then selects pages for eviction, migrates data between GPU and CPU memory, and resumes kernel execution. UVM works well for: (1) a single application whose total memory footprint exceeds GPU capacity but whose working set fits in GPU memory, and (2) multiple applications whose combined working sets fit within GPU memory.

While UVM's fully transparent abstraction is appealing, it has four fundamental limitations in the consumer setting. First, UVM lacks coordinated control over compute and memory management, making it highly susceptible to thrashing. Consider two applications, A and B, running concurrently on a 32 GB GPU and generating one token each in a round-robin fashion. Suppose each model requires 24 GB of parameters; then each forward pass triggers the migration of at least 16 GB of data to and from GPU memory. Even with state-of-the-art PCIe 5.0x16 bandwidth (64 GB/s), this migration takes roughly 250 ms. By comparison, a forward pass without migration typically takes only 20-75 ms. Thus, data movement alone induces a 4-12× slowdown.

Second, UVM utilizes only one direction of the PCIe bandwidth at a time, even though the PCIe channel is full-duplex. Figure 1 shows the IO path between CPU and GPU during page fault handling for UVM. During a page fault, UVM first evicts pages from GPU to CPU memory, and only after eviction completes does it fetch the demanded page and possibly several prefetched pages from CPU to GPU memory. A single fault may trigger multiple evictions followed by multiple fetches, but these transfers never overlap: eviction uses only GPU to CPU bandwidth, while fetching and prefetching use only CPU to GPU bandwidth. As a result, half of the available PCIe bandwidth remains idle.

Third, due to limited visibility into application behavior, UVM relies on a Least Recently Used (LRU)-based eviction policy to select pages during GPU page faults [4]. However, UVM only updates its LRU metadata when page faults occur, which provides a highly incomplete view of an application's true memory access pattern. As a result, pages that will be accessed imminently are often mistaken for cold pages, which leads to unnecessary page eviction and thrashing.

Fourth, host memory becomes a major bottleneck when UVM is used on consumer GPUs. To maximize PCIe transfer performance, UVM allocates CPU pinned memory inside the kernel for DMA operations. When UVM serves as the backing store for multiple GPU applications, every page resident on the GPU must have a corresponding pinned page on the CPU. This creates substantial memory overhead: even data that will never be evicted from the GPU still consumes unswappable, incompressible CPU pinned memory. On consumer machines equipped with modest RAM, this CPU pinned memory footprint competes directly with other applications and can significantly degrade overall system usability.

Several GPU multiplexing systems build on top of UVM to mitigate some of its thrashing issues (e.g., TGS [34] and nvshare [3]). For example, TGS assigns priorities to applications so that low-priority applications run only when high-priority ones are idle. However, because these systems rely on UVM as their underlying GPU memory virtualization mechanism, they inherit UVM's fundamental limitations: page-fault-driven migration, single-directional PCIe transfers, and large pinned-memory consumption.

3 Overview

Our design is guided by three key goals. First, we aim to provide a UVM-like abstraction that supports unmodified CUDA applications, while enabling coordinated control over compute and memory virtualization. Second, we seek to redesign virtual memory management to exploit a hierarchical memory system spanning GPU memory, CPU pinned memory, CPU paged memory, and disk, while ensuring that CPU-GPU data transfers fully utilize the full-duplex PCIe bandwidth. Third, similar to a standard OS kernel scheduler, we aim to optimize latency for interactive applications without sacrificing

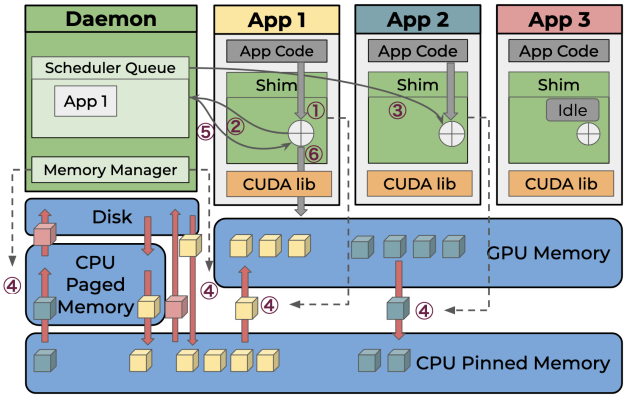


Figure 2: Nixie's architecture overview. Green components belong to Nixie. Cubes with different colors stand for memory blocks from different applications.

throughput for background batch-processing workloads.

Figure 2 presents the architecture of Nixie. The system consists of two components: a lightweight application library, Nixie Shim, and a centralized system service, Nixie Daemon. To provide transparency, Nixie Shim interposes on an application's CUDA Runtime API calls. Nixie Daemon performs global coordination and enforces multiplexing policies, but all kernels continue to execute using the application's own CUDA context to minimize runtime overheads on kernel launches. When App 1 launches a new kernel to execute GPU jobs, it ① checks the execution flag in the Nixie Shim; ② if App 1 cannot execute, Nixie Shim then uses interprocess communication (IPC) to enqueue a schedule request in daemon's scheduler queue. When the scheduler in Nixie Daemon decides App 1 is allowed to run, ③ Nixie Daemon uses IPC to pause the execution of App 2, waiting for its completion of outstanding CUDA kernels. Then ④ Nixie Daemon moves all the memory blocks belonging to App 1 into shared pinned memory, and each Nixie Shim migrates its own memory blocks into or out of GPU memory. After the memory blocks are ready for App 1, ⑤ Nixie Daemon notifies Nixie Shim in App 1 to enable the execution flag; App 1 then ⑥ resumes its execution. When all of App 1's data is ready, App 1 only executes steps ① and ⑥, bypassing IPC with Nixie Daemon.

Nixie Daemon centralizes three decisions: which application may launch kernels, where memory should reside, and the order in which required transfers are issued. Nixie Shim and Nixie Daemon then enforce those decisions with a small set of CUDA-facing mechanisms: intercepting CUDA calls, blocking new kernel launches at safe points, synchronizing outstanding kernels, remapping CUDA VMM allocations, and issuing asynchronous memory copies. The rest of this section describes these mechanisms and the decision logic built on top of them.

Nixie Daemon maintains a chunk-level virtual memory system. Instead of relying on page-fault-driven migration, Nixie manages memory at the granularity of CUDA-allocated chunk (i.e., the units returned by cudaMalloc). Each chunk may reside in one of four locations: (1) GPU memory, (2) CPU pinned memory, (3) CPU paged memory, or (4) disk. Unlike UVM, chunks placed in GPU memory do not require CPU pinned memory. To avoid fragmentation in CPU pinned memory or paged memory, we split a chunk into 2 MB blocks, where a block is the unit of data migration between locations.

Nixie's design follows two key principles. First, Nixie coordinates the control over both compute and memory allocation. When the scheduler selects an application to run, it proactively migrates that application's chunks into GPU memory while simultaneously evicting other applications' chunks to CPU memory or disk. By coupling scheduling with memory placement, Nixie ensures that when an application begins executing, it has both GPU compute and the necessary GPU-resident data available, avoiding demand-paging stalls and eliminating thrashing. Second, Nixie prioritizes interactive workloads automatically. Nixie determines whether an application is interactive by tracking its kernel launch behavior. Borrowing ideas from MLFQ, Nixie adjusts each application's priority and time window dynamically. If an application does not issue a kernel during its assigned time window, Nixie classifies it as interactive, halves its window size, and increases its priority. Conversely, if the application fully uses its assigned time window, Nixie doubles the window size and lowers its priority. At each decision point, the scheduler selects the highest-priority application and grants it GPU access for its current window. This approach allows Nixie to automatically prioritize latency-sensitive applications without any user annotations or application modifications, while preserving throughput for background batch-processing applications.

4 Supporting Unmodified CUDA Applications

Nixie is transparent to CUDA applications, similar to UVM. In UVM, a user replaces all CUDA memory allocations from `cudaMalloc` with `cudaMallocManaged`, and it is easy to make this fully transparent by static rewriting or `LD_PRELOAD`.

Nixie uses `LD_PRELOAD` to hook the CUDA library and interpose several functions. Besides memory allocation and free, Nixie also interposes CUDA kernel/graph launches in order to control compute behavior. Further, Nixie intercepts any APIs that implicitly allocate memory (e.g., `cudaStreamCreate`) or report memory usage (e.g., `cudaMemGetInfo`). Applications sometimes require correct memory usage reporting to function correctly. For example, SGLang reads GPU memory in order to decide how much KV cache memory to be allocated. Nixie Shim provides such interceptions and also communicates with Nixie Daemon via a UNIX domain socket.

To make sure applications can run correctly without being aware of the existence of Nixie, we need to keep an identical running environment before pausing and after resuming, and no invalid GPU operations should be introduced by Nixie.

Maintaining consistent virtual memory addresses. Classic approaches like `cudaMalloc` and `cudaFree` cannot guarantee that virtual memory addresses remain consistent after another call. Applications that use the previous addresses will access memory regions that are no longer valid. Nixie leverages CUDA VMM API [24] to reserve GPU virtual addresses and map these addresses to different physical allocations before and after chunk migrations.

Valid memory access during migration. GPU kernel execution is mostly asynchronous. When migration starts, there can be outstanding kernels still executing, and invalid memory accesses can happen when the accessed pages have been migrated. Before the currently running application is migrated, Nixie blocks any new kernels from launching and performs a CUDA synchronization to complete all the outstanding kernels. This prevents an application from using the migrated chunks.

CUDA graph compatibility. CUDA graph is widely used to reduce kernel launch overhead. When applications are constructing CUDA graph, unexpected API calls can disrupt the graph capture and lead to unexpected behavior. Nixie ensures Nixie Shim does not call any CUDA APIs until the graph capture finishes by interposing on the start and end of capture events.

GPU memory information. Some applications assume they have exclusive access to the entire GPU, and use device-wide statistics to make decisions for tensor placements or buffer allocations. Nixie reports GPU memory usage as the sum of allocations from the process itself, preventing applications from unnecessarily spilling data to CPU.

5 Virtual Memory Management

We first describe the granularity of memory management in Nixie, and then describe how to migrate memory chunks within hierarchical memory.

5.1 Memory Granularity and Hierarchy

Nixie intercepts all GPU memory allocations in applications, and it is thus naturally aware of all memory allocations. Nixie manages memory at the granularity of each allocation and migrates these units as a whole, which we refer to as *chunks*. Each chunk is placed at a specific tier of the memory hierarchy. Some applications (e.g., `llama.cpp`) put all the model weights into one big allocation. To prevent excessively large chunks, Nixie caps each chunk's size at 128 MB. Larger allocations are subdivided accordingly, allowing portions of the same allocation to reside in different memory hierarchy tiers. For example, a 256 MB allocation may be split so that roughly

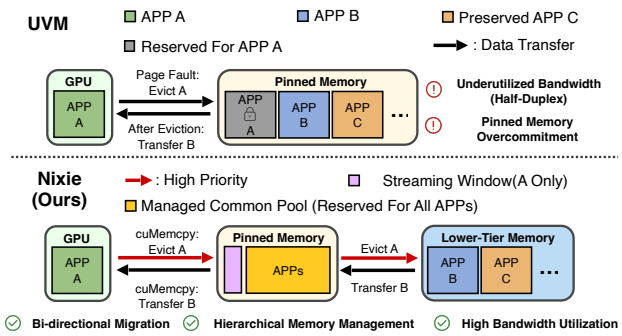


Figure 3: Nixie's memory hierarchy and chunk migration.

half resides in CPU pinned memory while the remainder is placed in CPU paged memory.

Allocations naturally vary in size because tensors (e.g., weights, activations) have diverse shapes. If Nixie managed memory solely at the chunk level, this variability would lead to severe fragmentation. To address this, Nixie introduces blocks for fine-grained control. Each block has a fixed size of 2 MB, which is the smallest physical allocation unit supported by the VMM API. A chunk is composed of one or more blocks (up to 64). Using uniformly sized blocks simplifies placement, migration, and bookkeeping, enabling Nixie to manage memory efficiently while avoiding fragmentation.

Nixie does not treat tier X as a cache backed by tier $X+1$. Instead, each chunk has *exactly one copy* within the entire memory hierarchy and is migrated between tiers as needed. Figure 3 shows Nixie's memory hierarchy. Nixie Daemon is the only component that issues data migration and has all the information needed to decide chunk locations in the memory hierarchy. Nixie Daemon can proactively perform background migrations without involving or interrupting applications. This design avoids keeping a pinned-memory backing copy for every GPU-resident chunk.

5.2 Speeding Up Context Switches

Migration is needed when GPU memory cannot fit the next application that will run, and long migration time directly affects user experience. To shorten migration, Nixie must keep both PCIe directions and the lower-tier memory links busy. Figure 4 illustrates the common case: application B is about to run and its blocks must be fetched into GPU memory, while blocks from the currently resident application A must be evicted.

A straightforward design would issue local transfers between adjacent tiers and rely on back-pressure to create free space. This performs poorly in a multi-tier hierarchy. A fetch from pinned memory to GPU may first require evicting GPU blocks to pinned memory; that eviction may require moving pinned-memory blocks to CPU paged memory; and the CPU paged tier may in turn need to spill blocks to disk. Serializing this chain leaves one or more links idle. It can also create

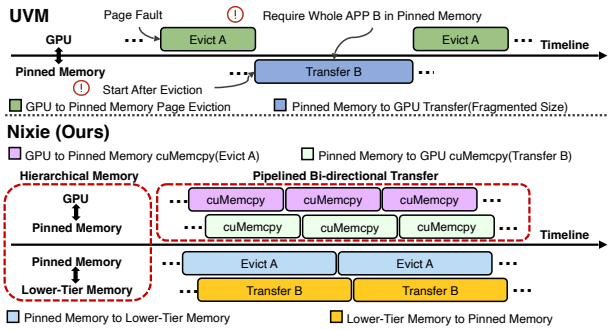


Figure 4: Nixie's communication bandwidth usage (1) between GPU memory and CPU pinned memory, and (2) between CPU pinned memory and lower-tier memory (e.g., CPU paged memory, disk).

avoidable traffic: if B fills the pinned tier while being fetched upward, A's evicted blocks may be forced to move farther down the hierarchy than necessary, or the system may deadlock waiting for space that can only be created by transfers blocked behind the fetch.

Nixie avoids these local decisions by separating *planning* from *orchestration*. The planner decides what moves are needed and where each block ends up; the orchestrator decides when and how to issue the actual data transfers. The planner creates a plan, i.e., a list of block moves, where each entry records the block identifier, current tier, destination tier, transfer direction, and migration order to prioritize blocks that will need further migration. The planner also computes the final state of each tier, so it can determine the minimal migration needed without exceeding any capacity constraints.

For an incoming application B, the planner first identifies all B blocks that are not already resident in GPU memory. It then computes the number of GPU blocks that must be freed and selects that many non-B GPU-resident blocks as eviction candidates. The eviction policy minimizes unnecessary movement: blocks are evicted only when needed to admit B, and each evicted block is placed in the highest lower tier with available non-reserved capacity. Blocks that cannot fit in pinned memory are assigned to CPU paged memory or disk before migration begins: they are transmitted at the beginning of the migration, rather than being pushed down only when one tier is full. When the scheduler already exposes a likely next application, the planner treats its blocks as less attractive eviction candidates, which preserves useful prefetching state. An example is shown in Figure 5.

The migration orchestrator executes the plan as faithfully as possible. It maintains separate queues for upward transfers (toward GPU) and downward transfers (away from GPU), and gives priority to moves that unblock the opposite direction. In the example of Figure 4, the orchestrator reserves a small pinned-memory streaming window for A's evicted blocks. This reservation ensures that GPU-to-pinned eviction can proceed even while B is being staged through pinned memory.

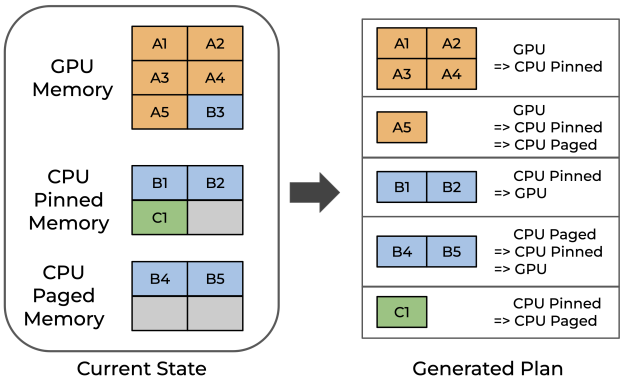


Figure 5: Migration plan given existing chunks' locations.

One issue we observed is that the block transfers may be stragglers with high latency, which comes from OS resource management or underlying CUDA runtime. In this case, to avoid deadlock with the staging buffer, the orchestrator can dynamically adjust the plan by evicting a small number of blocks to unblock the transmission.

For the actual data movement, Nixie Daemon moves incoming blocks into CPU pinned memory using multiple worker threads to saturate DRAM bandwidth. Once B's blocks are present in pinned memory and their GPU destinations are available, Nixie Daemon sends block identifiers and offsets to Nixie Shim via IPC. Nixie Shim then issues asynchronous CUDA copies between the 2 MB pinned-memory blocks and the application's reserved GPU virtual addresses.

6 Scheduling

Similar to traditional CPU scheduling, our goals are: (1) interactive applications should receive higher priority than batch-processing applications, and (2) users should not be required to manually specify application priorities. On CPUs, multi-level feedback queues (MLFQ) are the canonical mechanism to achieve both goals.

GPUs, however, provide neither CPU-style yield/blocking signals nor cheap context switches. The scheduler therefore has to infer when an application has stopped using the GPU and distinguish short, bursty applications from long-running applications without user annotations.

In the following, we describe how Nixie detects applications that are not using GPUs and then infers application priority. Finally, we discuss how to use prefetching to augment an MLFQ-like scheduling system tailored for GPU workloads.

6.1 Idleness Detection

In traditional CPU scheduling, a scheduler is typically invoked when (1) a time window expires or (2) a thread voluntarily relinquishes the CPU, for example by issuing `yield()` or invoking a blocking system call (e.g., I/O). These signals

allow the CPU scheduler to switch to another thread of execution. However, neither signal exists in GPU applications. First, Nixie needs to support applications with long-running services (e.g., SGLang, Ollama), so there is no notion of "task completion." Second, GPU applications do not expose equivalents of yield or blocking system calls to indicate periods of inactivity. As a result, letting applications run until the time window expires, regardless of whether an application is doing useful work, leads to poor GPU utilization and high latency for interactive workloads.

To address this, we design a mechanism to detect when a GPU application becomes idle. A straightforward approach is to observe GPU utilization using device APIs such as NVML. Unfortunately, this has two drawbacks. First, GPU-level APIs are coarse-grained and update slowly. In our experiments, NVML takes roughly 600 ms to report that a workload has become idle. Second, on consumer devices, background tasks (e.g., rendering or video decoding) introduce noise that makes these signals unreliable.

We use a timeout-based mechanism to detect application idleness. CUDA APIs fall into two categories: non-blocking operations (e.g., kernel launches, asynchronous calls) and blocking operations (e.g., synchronization). Nixie treats an application as idle only if the time elapsed since its most recent API return exceeds a fixed threshold. For blocking APIs, Nixie Shim intercepts the call both before and after execution to ensure the application is not mistakenly classified as idle while it is blocked inside the runtime. In our implementation, we use a 100 ms threshold. When the timeout fires, Nixie Shim sends an idleness notification to Nixie Daemon and Nixie Daemon marks the application as inactive. This notification invokes the same scheduling loop used when a time window expires. Idleness helps Nixie Daemon perform potential background migration and adjust scheduling priorities accordingly, as described in the following sections.

6.2 Priority Inference

Classic MLFQ decreases priorities over time and periodically resets all priorities to the highest level. This works well on CPUs, where time windows are short. In Nixie, however, GPU time windows are much longer. Blindly resetting all priorities erases useful history and may unexpectedly preempt interactive workloads.

We introduce a soft priority recovery policy. The scheduler records three timestamps for each application a: (1) the time it became idle i_a, (2) the time of last priority update p_a, and (3) the time of its most recent scheduling request q_a. Ideally, when an application stays idle for a sufficiently long period, it indicates the usage pattern may have changed (e.g., an LLM generating a long response for one prompt now may generate a short response for another prompt), so its priority should be increased to re-estimate the runtime characteristics. However, if the application is already in the scheduler queue pending further execution, promoting it immediately would preempt the currently running application and reduce the effective time window of every application in that priority level. To address this issue, we exclude pending time from consideration. Let R denote the multiplying factor applied to the pending time. If $R \geq 1$, an application could potentially starve at low priority whenever high-priority jobs are continuously running. Therefore, we dynamically adjust R based on the number of applications N at the same priority level, ensuring that $R < 1/N$. The detailed priority inference algorithm is shown in Algorithm 1.

Algorithm 1 Nixie Priority Inference

Require: Application a, pending time factor R, set of multi-level feedback queues $\{Q\}$ and corresponding time allotment $\{T\}$

```
p ← current priority of a
t_a ← accumulated execution time of a in current priority
if t_a > T_p then                          ▷ Standard MLFQ Demotion
    Move a to lower priority queue
    t_a ← 0
else if a is idle then
    i_a ← time since last execution
    p_a ← time since last priority change
    q_a ← time since a enqueued (0 if no new request)
    if p_a > T_p then                      ▷ Prevent priority jitter
        // Compensate idleness and prevent starvation
        if i_a − R · q_a > T_{p−1} + t_a then
            Move a to higher priority queue
        end if
    end if
end if
```

6.3 MLFQ with Prefetching

Nixie uses K queues $(Q_1, Q_2, \ldots, Q_k)$ where the priorities are in descending order from Q_1 to Q_k. The scheduler prioritizes applications with higher priority levels; when multiple applications share the same priority, they are scheduled in round-robin fashion, beginning with the application that has waited longest. Upon reaching a preemption threshold S, the scheduler preempts the current application if another application exists at the same priority level. We set the time allotment for priority demotion to $T = 8$s and the preemption threshold to $S = 4$s for the highest priority queue; each successive lower-priority queue doubles these parameters. The scheduler periodically checks for new scheduling decisions and updates priorities. Upon receiving an idleness notification from the currently running application, the scheduler performs the same operations.

In Nixie's hierarchical memory design, moving data from a lower to a higher tier requires a long time. We leverage information from the scheduler to reduce migration time:

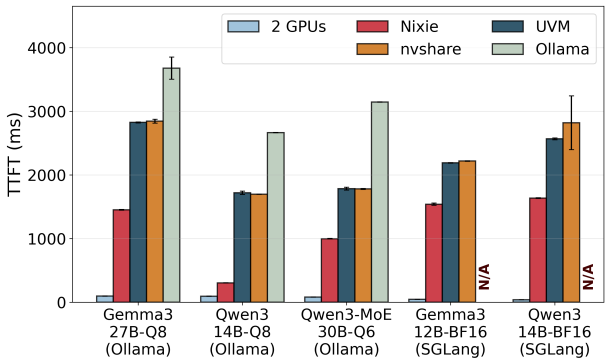


Figure 6: Context switch performance comparison. Error bars represent standard deviations.

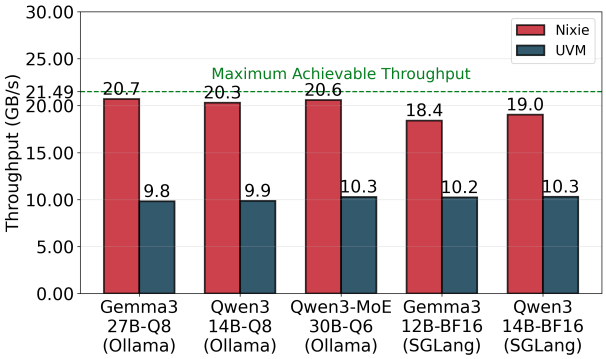


Figure 7: CPU from/to GPU memory copy throughput.

when there is already an application at the head of scheduler queue, it is most likely to run after the current application is scheduled out. Nixie thus prefetches data belonging to the next application to reduce migration time.

7 Evaluation

We have prototyped Nixie using ~10,000 lines of Rust code. Nixie Shim has ~1,900 lines of code, and Nixie Daemon has ~7,400 lines of code, with ~700 lines of shared code. Next, we microbenchmark Nixie 's context-switching latency as well as its compute and memory overheads. We then evaluate a set of representative use cases to demonstrate Nixie 's practical benefits. Following the previous work [6], we report average end-to-end time for these evaluations.

Setup. Our experiments are conducted on a desktop machine equipped with an AMD Ryzen 9 9950X CPU and 96 GB of dual-channel DDR5 memory at 3600 MT/s. The system includes two 32 GB RTX 5090 GPUs, each connected via PCIe 5.0x8. Unless noted, all Nixie, UVM, nvshare, Ollama, and TGS experiments use a single GPU; the second GPU serves only the *2 GPUs* baseline, which assigns one application to each GPU to approximate the zero-context-switch upper bound for multi-application performance. The desktop machine runs Debian 12 with CUDA version 12.9 and NVIDIA driver version 580.95.05. We run ML models using Ollama [25] (version 0.12.11), SGLang [42] (version 0.5.4.post1), `llama.cpp` [13] (version b7027) and ComfyUI [9] (commit eaf68c9). We pick a variety of models from Qwen3 [37], Gemma3 [29], Z-Image [31] and Qwen-Image [35] with different sizes and quantizations. For example, Qwen3-MoE 30B-Q6 means that it is the 30B variant of Qwen3-MoE with 6-bit quantization.

Baselines. We compare Nixie with Ollama [25], UVM [1], TGS [34], and nvshare [3]. The UVM baseline is implemented by hooking only `cudaMalloc`, `cudaFree`, and `cudaMemGetInfo` three APIs. We include Ollama as both an application atop Nixie and as a baseline: Ollama executes ML models and can switch between models natively. We only evaluate TGS using case studies and exclude TGS from microbenchmarks because TGS only supports exactly two applications and requires users to explicitly designate one as high priority and the other as low priority, making it incompatible with our microbenchmark workloads.

7.1 Microbenchmarks

Context Switch Performance. We evaluate context-switch performance by running an application executing a model and then switching to another identical instance of the same application and model on a single GPU. Figure 6 reports the Time-To-First-Token (TTFT) under five settings: *2 GPUs*, Nixie, UVM, nvshare, and Ollama. Here, *2 GPUs* places the two applications on separate GPUs, and thus reflects TTFT assuming the context switch is effectively free.

Overall, Nixie reduces TTFT by 44.0%-82.3% for Ollama cases and 29.7%-36.3% for SGLang cases compared to UVM and nvshare. Ollama uses a simple `mmap`-based strategy to unload and reload models, which results in the worst performance among all solutions tested.

We highlight two interesting observations. First, for MoE models (e.g., Qwen3-MoE), the performance benefit of Nixie for TTFT is smaller because the prefill only activates a subset of experts. Second, Nixie provides larger benefits for Ollama than for SGLang. SGLang aggressively allocates KV-cache memory that remains unused; Nixie proactively offloads all unused KV-cache pages to CPU memory, while UVM moves such pages in a delayed manner.

To understand why Nixie shortens context switch time compared to UVM and nvshare, we measure the bi-directional data-transfer throughput between host and GPU memory. Figure 7 shows the results. For reference, the dotted line marks the maximum achievable bi-directional throughput on our testbed, measured by NVIDIA's `nvbandwidth` [22] tool.

Nixie nearly saturates the available bi-directional bandwidth and achieves around 2× the throughput of UVM. The remaining gap between Nixie and the theoretical maximum primarily comes from memory allocation overheads. Ollama ex-

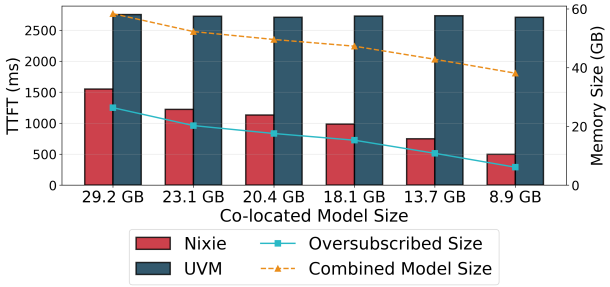
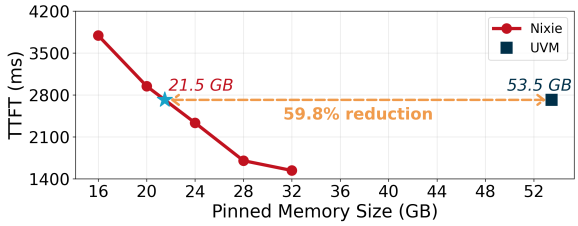
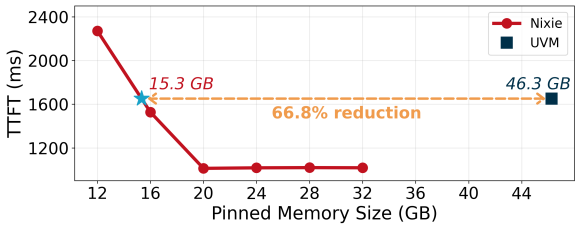


Figure 8: Performance comparison under different amounts of memory oversubscription. Oversubscribed size = Combined model size - GPU memory size.



(a) Gemma3 27B-Q8



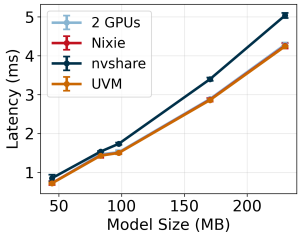
(b) Qwen3-MoE 30B-Q6

Figure 9: Pinned Memory Versus TTFT

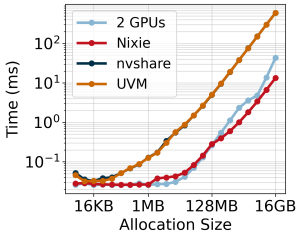
hibits a simple, large size allocation pattern, whereas SGLang has a more complex and non-uniform size pattern. Consequently, Nixie is able to reach higher bi-directional throughput when running Ollama than when running SGLang.

In Figure 8, we analyze the effect on TTFT when co-locating with applications of varying sizes. We use `llama.cpp` as our primary application and Gemma3 27B-Q8 as the model for measuring TTFT. As total GPU memory oversubscription decreases, Nixie achieves improved context switch performance, whereas UVM maintains constant context switch overhead due to its LRU policy, regardless of the extent of GPU memory oversubscription.

Sometimes, users may have insufficient CPU memory when they have to work with other applications. UVM leverages pinned memory for DMA to achieve the best migration performance. We vary the total amount of CPU pinned memory Nixie can use from 16 GB to 32 GB. In Figure 9, we show that Nixie can achieve the same performance as UVM with only 33.2% to 40.2% of the CPU pinned memory used by UVM.



(a) Inference latency



(b) cudaMalloc latency

Figure 10: Nixie's runtime overheads.

Runtime Overheads. During the time an application is scheduled to run on a GPU, Nixie should have minimal performance overheads on kernel launch, even when the kernel itself is small. This is because during the window the application is scheduled to run, the CUDA launch calls directly go through the corresponding CUDA userland driver and do not need to be handled via Nixie Daemon.

Figure 10a displays inference latencies of various small ResNet models ranging from ResNet-18 to ResNet-152 at a batch size of 1. Nixie shows the identical performance with vanilla execution from *2 GPUs* and simple UVM interception. The overhead from nvshare mainly comes from its CPU thread synchronization and GPU synchronization under certain conditions.

Memory Overheads. In Figure 10b, we measure the latency of `cudaMalloc()` with varying allocation sizes. We execute a subsequent `cudaMemset()` to eliminate the effects of lazy initialization. For allocations below 2 MB, Nixie directly uses `cudaMalloc()`, achieving performance comparable to the vanilla implementation. Between 2 MB and 128 MB, Nixie exhibits slightly higher allocation times compared to the vanilla implementation. Because the CUDA runtime is closed-source, we can only hypothesize why Nixie outperforms the vanilla implementation once the allocation size exceeds 128 MB. Our hypothesis is that Nixie allocates memory using multiple contiguous 128 MB physical regions, whereas the vanilla implementation may attempt to allocate a single, maximally contiguous region. UVM and UVM-based nvshare shows worse performance because UVM adopts a first-touch allocation policy, which defers physical memory allocation until the initial access, introducing more overhead.

7.2 Case Studies

We present four representative use cases. Cases #1 and #2 are multi-application execution workflows, where applications interact with each other. Cases #3 and #4 are concurrent workloads, where multiple workloads are independent and have no dependency.

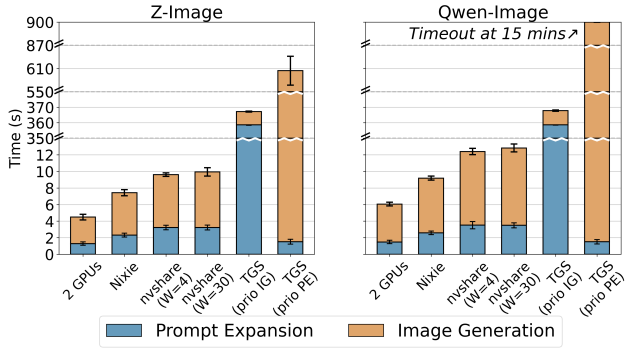


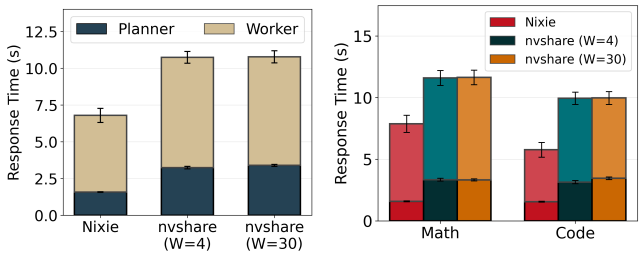
Figure 11: Using Qwen3-MoE 30B-Q6 to extend the text-to-image prompt for better image quality. Error bars represent standard deviations.

Case #1: Prompt-expanded image generation. Prompt expansion [10, 19] is a standard technique for image generation. When the user wants to generate an image given a prompt, an LLM is then used to expand the prompt to make the description of the image more precise and contain more concrete details. This workflow requires cross-application collaboration between an LLM inference application and an image generation application.

We evaluate this workflow with `llama.cpp` and ComfyUI. For prompt expansion, we use Qwen3-MoE 30B-Q6 [37]. For image generation, we select two commonly used models: (1) 6B Z-Image [31] with BF16, and (2) 20B Qwen-Image [35] with FP8. We use the DreamBench++ dataset [26] for our prompts. We compare Nixie against nvshare under two window configurations: W = 4 s, which matches Nixie's preemption threshold of the highest priority, and W = 30 s, which is the default setting of nvshare. Since TGS requires explicit priority assignment, we include two variants: one that prioritizes image generation (prio IG) and one that prioritizes the LLM-based prompt expansion (prio PE).

Figure 11 reports the results. Nixie is 1.4× and 1.3× faster than nvshare (W = 4) and achieves 65.9% and 60.4% of the performance of using two dedicated GPUs for Qwen-Image and Z-Image, respectively. Notably, even though ComfyUI frequently allocates and frees GPU memory during Qwen-Image generation, Nixie maintains higher performance, indicating that the performance of Nixie is robust to memory allocation and deallocation on the data path.

TGS exhibits significantly degraded performance and cannot finish image generation for Qwen-Image within 15 minutes. There are two primary reasons: (1) TGS assumes that the high-priority job must sustain a constant throughput, which does not hold for most consumer-oriented applications. Consequently, the TGS rate limiter fails to function effectively in this scenario. (2) Under memory oversubscription, TGS forces low-priority jobs to access host memory directly via DMA without migration, essentially stalling computation.



(a) Overall Latency (b) Latency by Request Type

Figure 12: Multi-agent orchestration for math and code tasks. Error bars represent standard deviations.

Case #2: Multi-agent systems. Multi-agent systems are an increasingly common way to structure complex LLM applications. Instead of relying on a single monolithic model, a *planner* agent decomposes a user request into subtasks and dispatches each subtask to a specialized *worker* model. This planner-worker orchestration pattern appears in current coding assistants, tool-using agents and many recent multi-LLM workflows, and serves as a canonical multi-agent workload [12, 15, 33].

We instantiate this model-orchestration workload using the open-source multi-agent framework KVCOMM [38]. For routing, we use Qwen3 14B-Q8 to decide the best downstream model. Following model-level benchmarks [5, 29, 30, 37], we select Qwen3-VL 32B-Q6 as the math expert and Qwen3-Coder 30B-Q6 as the coding expert. All models are served through `llama.cpp` on a single GPU; the only difference between configurations is the GPU sharing mechanism. We compare Nixie against nvshare with W=4 and W=30. We issue a mixed stream of math and coding requests (shuffled from GSM8K [8] and HumanEval-FIM [7]) and record end-to-end latency per request, as well as its breakdown into planner and worker components.

Figure 12 shows our results. Across all requests, Nixie is 1.6× faster than both nvshare configurations. The gains are consistent across request types as well: for math and coding tasks, Nixie is 1.5× and 1.7× faster, respectively. These results indicate that Nixie offers strong flexibility across multiple applications, enabling efficient model orchestration on a single shared GPU without sacrificing the interactivity of the multi-agent workflow.

Case #3: Code completion with long-running job. LLM-assisted code completion is a representative latency-sensitive task requiring immediate responsiveness. We expect Nixie to naturally prioritize it over long-running jobs, such as autonomous agent operations or extended video generation tasks with the help of the scheduler. To evaluate scheduler's prioritization behavior, we run code completion with a long-running agentic workload. We use Qwen3-Coder 30B-Q6 for Fill-In-Middle (FIM) completion and Gemma3 27B-Q8 to emulate an LLM-based agent. HumanEval-FIM [7] is used as the in-

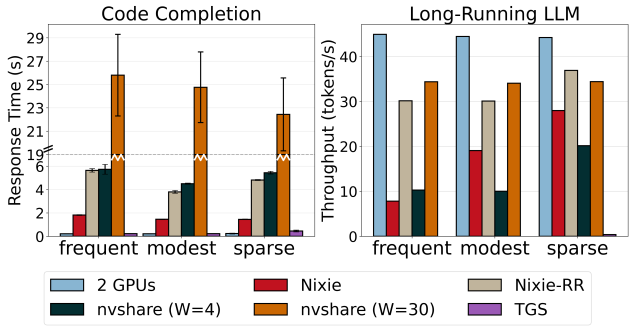


Figure 13: Code completion with long-running generation. Error bars represent standard deviations.

put for code completion. Since the frequency and interval between code completion requests vary depending on individual developers and their tooling implementations, we create three scenarios to represent different usage patterns: frequent (1-second intervals), modest (3-second intervals), and sparse (6-second intervals). Additionally, we run Nixie with a basic round-robin scheduling policy as an ablation to evaluate the effectiveness of our scheduler, noted as Nixie-RR. This variant uses a 4-second time window, consistent with the other experimental settings.

Figure 13 shows the response time of code completion and the throughput of the long-running model. Nixie achieves an average response time of 1.8 s for frequent completion scenario and 1.4 s for modest and sparse scenarios. In contrast, nvshare (W=4) requires 4.5 s to 5.7 s to complete an interactive request depending on the frequency. Nixie is 3.1× to 3.8× faster compared to nvshare (W=4). When nvshare uses the default W=30, the latency exceeds 20 s, rendering LLM code completion impractical for real-world use cases. Notably, without an appropriate scheduling policy, Nixie-RR exhibits long response time for the latency-critical task across different settings.

Regarding the long-running job throughput, Nixie is 90.6% and 39.5% higher than nvshare (W=4) in the modest and sparse scenarios respectively. In the frequent scenario, Nixie throughput is 23.5% lower, which is expected: Nixie prioritizes the interactive jobs and allocates more GPU time for code completion.

These results make the tradeoff explicit: under frequent interactive arrivals, Nixie sacrifices background throughput to keep code-completion latency low; when interactive arrivals are modest or sparse, Nixie achieves both lower interactive latency and higher background throughput than nvshare (W=4). TGS achieves similar performance for code completion task, but fails to allow the long-running model to run. TGS essentially degrades to running only the high priority application.

Case #4: Many Batched Tasks. While Nixie targets interactive workloads, some GPU jobs are sufficiently heavy that they must run for extended periods (e.g., long-document processing, high-volume generation). We therefore evaluate both throughput and fairness in a setting where multiple batched jobs execute concurrently. We also evaluate the fairness of our scheduler design.

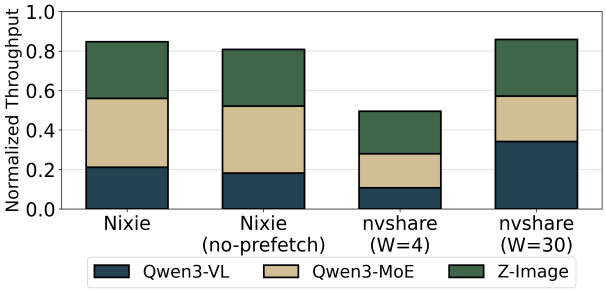


Figure 14: Different Batch Jobs

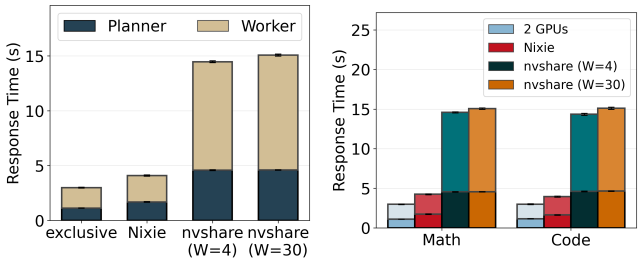
We run three jobs: Qwen3-MoE 30B-Q6, Qwen3-VL 32B-Q6 (long-running LLM request with batch size 1), and Z-Image (batch size 8). We also ablate the effectiveness of our auto-prefetching mechanism in the scheduler. We compare our solution with nvshare configured with W=4 and W=30. All experiments are conducted over a 300-second duration. To account for the heterogeneity of metrics across different applications, we additionally run each application in isolation as a reference, and report normalized throughput relative to its profiled standalone performance.

Figure 14 shows the results. Nixie achieves throughput comparable to nvshare (W=30) achieving 85% of the ideal throughput, even when Nixie employs an adaptive policy to prioritize responsiveness, whereas nvshare utilizes a fixed long time quantum for throughput. Fairness is effectively maintained across all tested configurations. Auto-prefetching yields an additional 5% throughput improvement over the no-prefetch variant, despite context switches already being infrequent in this scenario. In contrast, nvshare (W=4) only realizes 49.5% throughput because of the overhead caused by frequent context switches.

7.3 Performance on Other Hardware

We evaluate Nixie on another server equipped with two AMD EPYC 7352 24-Core Processor CPUs, 1 TB of 16-channel DDR4 memory at 3200 MT/s, and 8x RTX A5000 GPU (24 GB) connected via PCIe 4.0x16. It runs Ubuntu 22.04 with CUDA 12.4 and Nvidia driver version 550.67.

On this platform, we reuse the multi-agent orchestration workload from **Case #2**. Given RTX A5000 has smaller memory, we use Qwen3 14B-Q4 as the planner, Qwen3-VL 32B-Q4 as the math worker, and Qwen3-Coder 30B-Q4 as the coding worker. We add *2 GPUs* variant here given more available GPUs. Figure 15 shows the results. Nixie is 3.4× faster than nvshare in this case and achieves 73% of the performance of *2 GPUs*. The UVM-based solution slows down significantly for this setup. We observed increase in PCIe idleness on this hardware. We suspect the root cause is the older NVIDIA



(a) Overall Latency

(b) Latency by Request Type

Figure 15: Case #2 on RTX A5000 testbed. Error bars represent standard deviations.

driver, and the correspondingly older UVM implementation.

8 Discussion

Generalizability to other operating systems. Nixie leverages CUDA VMM API, which is available on both Linux and Windows. We have prototyped Nixie on Linux, and we believe that it is feasible to port Nixie to Windows, which remains the most widely used operating system for consumers. In contrast, UVM does not support memory oversubscription on Windows [23], and adding such support would require substantial engineering effort in the GPU driver stack on Windows. This distinction makes Nixie a more practical solution for consumer platforms. Furthermore, similar VMM APIs also exist for AMD GPUs [2].

White-box solutions. Nixie is a fully transparent solution that operates without any awareness of tensor semantics. In principle, certain immutable data (e.g., model weights) could be freed directly on the GPU without migrating it back to CPU memory, provided the application retains a separate CPU-side copy. For example, CUDA memory copies inherently leave duplicate data in both CPU and GPU memory. We did not pursue this direction because it would require semantic knowledge of application data structures, compromising transparency. However, incorporating such semantic hints represents an interesting direction for future work.

Policy extensibility. Nixie provides mechanisms for transparent GPU time sharing: orchestrating kernel launches, VMM-based remapping, hierarchical memory placement with migration and migration-scheduling cooperation. In this paper, we deliberately adopt application-agnostic policies that make no assumptions about workload characteristics, prioritizing broad compatibility over peak performance. These policies are intentionally conservative; refining them is a promising direction for future work, for example by learning scheduling decisions from recent execution history or by treating different GPU memory regions (e.g., model weights, activations, KV caches) differently.

Co-locating with small models. Nixie's target workloads are those in which each application utilizes most of the GPU memory, so temporal multiplexing is the only approach. When there is a set of small models, it is possible to perform spatial multiplexing and have different models execute at the same time as existing works on datacenter GPU multiplexing [14, 27, 41]. How to integrate spatial multiplexing into Nixie is a promising future direction.

Security. Nixie Daemon shares the same UID with applications, and applications share the same region of CPU pinned memory for storing migrated GPU data. A malicious process may access the shared data region inside its own address space. However, Linux uses UID for shared memory access control, which means any process under the same UID can access CPU pinned memory, whether the memory is shared by all processes or each process has its own dedicated CPU pinned memory. Since Nixie is for a single-user environment rather than multi-tenancy, this is outside our threat model.

9 Related Work

GPU multiplexing before the era of LLMs. GPU multiplexing has been extensively studied prior to the rise of LLMs. For traditional computer vision workloads, multiplexing allows significant consolidation of hardware by serving many models on shared GPUs (e.g., Nexus [27], Shepherd [41], Clockwork [14]). In these systems, GPU memory stores multiple models simultaneously, and incoming requests are batched and executed across one or more models. Reducing the number of co-located models or lowering batch sizes helps alleviate GPU memory pressure. In this setting, the dominant challenge is maximizing GPU compute efficiency. In contrast, Nixie targets temporal, not spatial, multiplexing: rather than having the GPU run inference for many models, Nixie time-shares it across large, memory-intensive applications whose working sets approach the GPU's full capacity.

GPU multiplexing in the era of LLMs and other large foundation models. LLMs and other large foundation models have fundamentally reshaped the GPU multiplexing landscape. Attention computation is highly memory-intensive, especially with long contexts. Even a single LLM inference often consumes nearly all available GPU memory. As a result, recent research has pivoted toward temporal multiplexing and techniques for handling GPU memory oversubscription.

PipeSwitch [6] introduces a model-aware, layer-based task-switching mechanism, but it requires application modifications. Further, it is less effective as the ratio of migration time to model execution time grows in modern foundation models. ServerlessLLM [11], Aegaeon [36], and Prism [39] address oversubscription by offloading less frequently accessed model components (e.g., entire models or KV caches)

to CPU memory or disk, primarily to meet provider's Service Level Objectives. These systems are designed for datacenter LLM serving, where workloads are controlled by a single provider and can be integrated directly into a specific inference engine. Aegaeon relies on token-granular autoscaling, and Prism relies on on-demand KV-cache allocation. These mechanisms do not apply to many consumer workloads we target, including diffusion, image editing, and video pipelines, which do not expose an LLM KV cache. They also require integration with a particular serving stack, whereas Nixie remains transparent to unmodified applications on a single user's desktop. Conversely, Nixie is not a replacement for these systems in datacenter serving. Nixie assumes whole-application context switches over the CPU-GPU link, with each application's working set nearly saturating the GPU. It does not distinguish model weights from KV cache or reason about prefill/decode phases, so it cannot selectively resize KV cache or reload only the requests about to run. Those application-level optimizations are precisely where Aegaeon and Prism reduce transfers across concurrent LLM models and requests.

Improving UVM. UVM provides a transparent abstraction for GPU memory oversubscription, which is appealing. Systems such as DeepUM [16] optimize UVM performance by incorporating tensor-level prefetching. G10 [40] adds disks to UVM and uses compiler to help manage pipelined data transfer. These solutions require hardware changes or driver changes to GPUs. Further, Nixie is not purely a virtual memory system. It does not depend on page faults and can coordinate compute and memory management at the same time to mitigate memory thrashing.

GPU scheduling. Multiple works discuss multitasking scheduling. XSched [28] presents a framework that allows for preemptive XPU scheduling. Agentix [20] incorporates application dependencies into LLM scheduling. Our scheduling policy is quite simple, and we do not need any application-level information. Our scheduling goal is to prioritize interactive applications.

10 Conclusion

Consumer machines increasingly execute large ML workloads such as LLM inference, text-to-image generation, and interactive image editing. Unlike datacenter environments, consumer GPUs must handle multiple applications concurrently, and each application often nearly saturates GPU memory. We have built Nixie, enabling efficient and fully transparent temporal multiplexing on consumer GPUs without requiring any modifications to applications or GPU drivers. Nixie coordinates GPU memory allocation with kernel-launch behavior to minimize thrashing. Nixie's scheduler further enhances interactivity by automatically identifying and prioritizing latency-sensitive applications. Nixie improves the latency of interactive code-completion workloads by up to 3.8×, while reducing CPU pinned-memory usage by up to 66.8% under equivalent latency constraints.

Acknowledgements

We thank the anonymous OSDI reviewers and shepherd for providing insightful feedback on our work. This work was supported in part by National Science Foundation grants CNS-2112562, CNS-2238665, CNS-2402696, and OAC-2503010, as well as by gifts from Amazon and Meta.

References

[1] CUDA Unified Memory, 2015. Accessed: October 2025.

[2] Advanced Micro Devices, Inc. *Virtual Memory Management*. AMD, 2025. HIP 7.1.52802 Documentation.

[3] Georgios Alexopoulos and Dimitris Mitropoulos. nvshare: Practical GPU Sharing without Memory Size Constraints. In *Proceedings of the 2024 IEEE/ACM 46th International Conference on Software Engineering: Companion Proceedings*, ICSE-Companion '24, pages 16–20, New York, NY, USA, 2024. Association for Computing Machinery.

[4] Tyler Allen and Rong Ge. Demystifying GPU UVM Cost with Deep Runtime and Workload Analysis. In *2021 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 141–150. IEEE, 2021.

[5] Shuai Bai, Yuxuan Cai, Ruizhe Chen, Keqin Chen, Xionghui Chen, Zesen Cheng, Lianghao Deng, Wei Ding, Chang Gao, Chunjiang Ge, Wenbin Ge, Zhifang Guo, Qidong Huang, Jie Huang, Fei Huang, Binyuan Hui, Shutong Jiang, Zhaohai Li, Mingsheng Li, Mei Li, Kaixin Li, Zicheng Lin, Junyang Lin, Xuejing Liu, Jiawei Liu, Chenglong Liu, Yang Liu, Dayiheng Liu, Shixuan Liu, Dunjie Lu, Ruilin Luo, Chenxu Lv, Rui Men, Lingchen Meng, Xuancheng Ren, Xingzhang Ren, Sibo Song, Yuchong Sun, Jun Tang, Jianhong Tu, Jianqiang Wan, Peng Wang, Pengfei Wang, Qiuyue Wang, Yuxuan Wang, Tianbao Xie, Yiheng Xu, Haiyang Xu, Jin Xu, Zhibo Yang, Mingkun Yang, Jianxin Yang, An Yang, Bowen Yu, Fei Zhang, Hang Zhang, Xi Zhang, Bo Zheng, Humen Zhong, Jingren Zhou, Fan Zhou, Jing Zhou, Yuanzhi Zhu, and Ke Zhu. Qwen3-VL Technical Report, 2025.

[6] Zhihao Bai, Zhen Zhang, Yibo Zhu, and Xin Jin. PipeSwitch: Fast Pipelined Context Switching for Deep

Learning Applications. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 499–514, 2020.

[7] Mohammad Bavarian, Heewoo Jun, Nikolas Tezak, John Schulman, Christine McLeavey, Jerry Tworek, and Mark Chen. Efficient Training of Language Models to Fill in the Middle, 2022.

[8] Karl Cobbe, Vineet Kosaraju, Mohammad Bavarian, Mark Chen, Heewoo Jun, Lukasz Kaiser, Matthias Plappert, Jerry Tworek, Jacob Hilton, Reiichiro Nakano, Christopher Hesse, and John Schulman. Training Verifiers to Solve Math Word Problems, 2021.

[9] Comfyanonymous. ComfyUI: The Most Powerful and Modular Diffusion Model GUI, API and Backend with a Graph/Nodes Interface. https://github.com/comfyanonymous/ComfyUI, 2025.

[10] Siddhartha Datta, Alexander Ku, Deepak Ramachandran, and Peter Anderson. Prompt Expansion for Adaptive Text-to-Image Generation. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 3449–3476, Bangkok, Thailand, August 2024. Association for Computational Linguistics.

[11] Yao Fu, Leyang Xue, Yeqi Huang, Andrei-Octavian Brabete, Dmitrii Ustiugov, Yuvraj Patel, and Luo Mai. ServerlessLLM: Low-Latency Serverless Inference for Large Language Models. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, pages 135–153, Santa Clara, CA, July 2024. USENIX Association.

[12] Yingqiang Ge, Wenyue Hua, Kai Mei, Jianchao Ji, Juntao Tan, Shuyuan Xu, Zelong Li, and Yongfeng Zhang. OpenAGI: When LLM Meets Domain Experts. *In Advances in Neural Information Processing Systems (NeurIPS)*, 2023.

[13] Georgi Gerganov. llama.cpp: LLM inference in C/C++. https://github.com/ggerganov/llama.cpp, 2025.

[14] Arpan Gujarati, Reza Karimi, Safya Alzayat, Wei Hao, Antoine Kaufmann, Ymir Vigfusson, and Jonathan Mace. Serving DNNs like Clockwork: Performance Predictability from the Bottom Up. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 443–462. USENIX Association, November 2020.

[15] Sirui Hong, Mingchen Zhuge, Jonathan Chen, Xiawu Zheng, Yuheng Cheng, Jinlin Wang, Ceyao Zhang, Zili Wang, Steven Ka Shing Yau, Zijuan Lin, Liyang Zhou, Chenyu Ran, Lingfeng Xiao, Chenglin Wu, and Jürgen Schmidhuber. MetaGPT: Meta Programming for A Multi-Agent Collaborative Framework. In *The Twelfth International Conference on Learning Representations*, 2024.

[16] Jaehoon Jung, Jinpyo Kim, and Jaejin Lee. DeepUM: Tensor Migration and Prefetching in Unified Memory. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, pages 207–221, 2023.

[17] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph Gonzalez, Hao Zhang, and Ion Stoica. Efficient Memory Management for Large Language Model Serving with PagedAttention. In *Proceedings of the 29th Symposium on Operating Systems Principles*, pages 611–626, 2023.

[18] Black Forest Labs, Stephen Batifol, Andreas Blattmann, Frederic Boesel, Saksham Consul, Cyril Diagne, Tim Dockhorn, Jack English, Zion English, Patrick Esser, Sumith Kulal, Kyle Lacey, Yam Levi, Cheng Li, Dominik Lorenz, Jonas Müller, Dustin Podell, Robin Rombach, Harry Saini, Axel Sauer, and Luke Smith. FLUX.1 Kontext: Flow Matching for In-Context Image Generation and Editing in Latent Space, 2025.

[19] Long Lian, Boyi Li, Adam Yala, and Trevor Darrell. LLM-grounded Diffusion: Enhancing Prompt Understanding of Text-to-Image Diffusion Models with Large Language Models, 2024.

[20] Michael Luo, Xiaoxiang Shi, Colin Cai, Tianjun Zhang, Justin Wong, Yichuan Wang, Chi Wang, Yanping Huang, Zhifeng Chen, Joseph E. Gonzalez, and Ion Stoica. Agentix: An Efficient Serving Engine for LLM Agents as General Programs. In *23rd USENIX Symposium on Networked Systems Design and Implementation (NSDI 26)*, 2026.

[21] mostlygeek. llama-swap: Model swapping for llama.cpp, 2025. Lightweight transparent proxy server for automatic model swapping.

[22] NVIDIA Corporation. nvbandwidth: A tool for bandwidth measurements on NVIDIA GPUs. https://github.com/NVIDIA/nvbandwidth, 2025. Accessed: 2025-11-23.

[23] NVIDIA Corporation. *Unified Memory On Windows, Wsl And Tegra*. NVIDIA Corporation, 2025. CUDA Programming Guide, Version 13.1.

[24] NVIDIA Corporation. *Virtual Memory Management*. NVIDIA Corporation, 2025. CUDA Driver API Documentation, CUDA Toolkit v13.1.0.

[25] Ollama Contributors. Ollama. https://github.com/ollama/ollama, 2025.

[26] Yuang Peng, Yuxin Cui, Haomiao Tang, Zekun Qi, Runpei Dong, Jing Bai, Chunrui Han, Zheng Ge, Xiangyu Zhang, and Shu-Tao Xia. DreamBench++: A Human-Aligned Benchmark for Personalized Image Generation. In *The Thirteenth International Conference on Learning Representations*, 2025.

[27] Haichen Shen, Lequn Chen, Yuchen Jin, Liangyu Zhao, Bingyu Kong, Matthai Philipose, Arvind Krishnamurthy, and Ravi Sundaram. Nexus: a GPU cluster engine for accelerating DNN-based video analysis. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP '19, pages 322–337, New York, NY, USA, 2019. Association for Computing Machinery.

[28] Weihang Shen, Mingcong Han, Jialong Liu, Rong Chen, and Haibo Chen. XSched: Preemptive Scheduling for Diverse XPUs. In *19th USENIX Symposium on Operating Systems Design and Implementation (OSDI 25)*, pages 671–692, 2025.

[29] Gemma Team, Aishwarya Kamath, Johan Ferret, Shreya Pathak, Nino Vieillard, Ramona Merhej, Sarah Perrin, Tatiana Matejovicova, Alexandre Ramé, Morgane Rivière, Louis Rouillard, Thomas Mesnard, Geoffrey Cideron, Jean bastien Grill, Sabela Ramos, Edouard Yvinec, Michelle Casbon, Etienne Pot, Ivo Penchev, Gaël Liu, Francesco Visin, Kathleen Kenealy, Lucas Beyer, Xiaohai Zhai, Anton Tsitsulin, Robert Busa-Fekete, Alex Feng, Noveen Sachdeva, Benjamin Coleman, Yi Gao, Basil Mustafa, Iain Barr, Emilio Parisotto, David Tian, Matan Eyal, Colin Cherry, Jan-Thorsten Peter, Danila Sinopalnikov, Surya Bhupatiraju, Rishabh Agarwal, Mehran Kazemi, Dan Malkin, Ravin Kumar, David Vilar, Idan Brusilovsky, Jiaming Luo, Andreas Steiner, Abe Friesen, Abhanshu Sharma, Abheesht Sharma, Adi Mayrav Gilady, Adrian Goedeckemeyer, Alaa Saade, Alex Feng, Alexander Kolesnikov, Alexei Bendebury, Alvin Abdagic, Amit Vadi, András György, André Susano Pinto, Anil Das, Ankur Bapna, Antoine Miech, Antoine Yang, Antonia Paterson, Ashish Shenoy, Ayan Chakrabarti, Bilal Piot, Bo Wu, Bobak Shahriari, Bryce Petrini, Charlie Chen, Charline Le Lan, Christopher A. Choquette-Choo, CJ Carey, Cormac Brick, Daniel Deutsch, Danielle Eisenbud, Dee Cattle, Derek Cheng, Dimitris Paparas, Divyashree Shivakumar Sreepathihalli, Doug Reid, Dustin Tran, Dustin Zelle, Eric Noland, Erwin Huizenga, Eugene Kharitonov, Frederick Liu, Gagik Amirkhanyan, Glenn Cameron, Hadi Hashemi, Hanna Klimczak-Plucińska, Harman Singh, Harsh Mehta, Harshal Tushar Lehri, Hussein Hazimeh, Ian Ballantyne, Idan Szpektor, Ivan Nardini, Jean Pouget-Abadie, Jetha Chan, Joe Stanton, John Wieting, Jonathan Lai, Jordi Orbay, Joseph Fernandez, Josh Newlan, Ju yeong Ji, Jyotinder Singh, Kat Black, Kathy Yu, Kevin Hui, Kiran Vodrahalli, Klaus Greff, Linhai Qiu, Marcella Valentine, Marina Coelho, Marvin Ritter, Matt Hoffman, Matthew Watson, Mayank Chaturvedi, Michael Moynihan, Min Ma, Nabila Babar, Natasha Noy, Nathan Byrd, Nick Roy, Nikola Momchev, Nilay Chauhan, Noveen Sachdeva, Oskar Bunyan, Pankil Botarda, Paul Caron, Paul Kishan Rubenstein, Phil Culliton, Philipp Schmid, Pier Giuseppe Sessa, Pingmei Xu, Piotr Stanczyk, Pouya Tafti, Rakesh Shivanna, Renjie Wu, Renke Pan, Reza Rokni, Rob Willoughby, Rohith Vallu, Ryan Mullins, Sammy Jerome, Sara Smoot, Sertan Girgin, Shariq Iqbal, Shashir Reddy, Shruti Sheth, Siim Põder, Sijal Bhatnagar, Sindhu Raghuram Panyam, Sivan Eiger, Susan Zhang, Tianqi Liu, Trevor Yacovone, Tyler Liechty, Uday Kalra, Utku Evci, Vedant Misra, Vincent Roseberry, Vlad Feinberg, Vlad Kolesnikov, Woohyun Han, Woosuk Kwon, Xi Chen, Yinlam Chow, Yuvein Zhu, Zichuan Wei, Zoltan Egyed, Victor Cotruta, Minh Giang, Phoebe Kirk, Anand Rao, Kat Black, Nabila Babar, Jessica Lo, Erica Moreira, Luiz Gustavo Martins, Omar Sanseviero, Lucas Gonzalez, Zach Gleicher, Tris Warkentin, Vahab Mirrokni, Evan Senter, Eli Collins, Joelle Barral, Zoubin Ghahramani, Raia Hadsell, Yossi Matias, D. Sculley, Slav Petrov, Noah Fiedel, Noam Shazeer, Oriol Vinyals, Jeff Dean, Demis Hassabis, Koray Kavukcuoglu, Clement Farabet, Elena Buchatskaya, Jean-Baptiste Alayrac, Rohan Anil, Dmitry, Lepikhin, Sebastian Borgeaud, Olivier Bachem, Armand Joulin, Alek Andreev, Cassidy Hardin, Robert Dadashi, and Léonard Hussenot. Gemma 3 Technical Report, 2025.

[30] Qwen Team. Qwen3-Coder-30B-A3B-Instruct. https://huggingface.co/Qwen/Qwen3-Coder-30B-A3B-Instruct, 2025.

[31] Z-Image Team, Huanqia Cai, Sihan Cao, Ruoyi Du, Peng Gao, Steven Hoi, Zhaohui Hou, Shijie Huang, Dengyang Jiang, Xin Jin, Liangchen Li, Zhen Li, Zhong-Yu Li, David Liu, Dongyang Liu, Junhan Shi, Qilong Wu, Feng Yu, Chi Zhang, Shifeng Zhang, and Shilin Zhou. Z-Image: An Efficient Image Generation Foundation Model with Single-Stream Diffusion Transformer, 2025.

[32] Team Wan, Ang Wang, Baole Ai, Bin Wen, Chaojie Mao, Chen-Wei Xie, Di Chen, Feiwu Yu, Haiming Zhao, Jianxiao Yang, Jianyuan Zeng, Jiayu Wang, Jingfeng Zhang, Jingren Zhou, Jinkai Wang, Jixuan Chen, Kai Zhu, Kang Zhao, Keyu Yan, Lianghua Huang, Mengyang Feng, Ningyi Zhang, Pandeng Li, Pingyu Wu, Ruihang Chu, Ruili Feng, Shiwei Zhang, Siyang

Sun, Tao Fang, Tianxing Wang, Tianyi Gui, Tingyu Weng, Tong Shen, Wei Lin, Wei Wang, Wei Wang, Wenmeng Zhou, Wente Wang, Wenting Shen, Wenyuan Yu, Xianzhong Shi, Xiaoming Huang, Xin Xu, Yan Kou, Yangyu Lv, Yifei Li, Yijing Liu, Yiming Wang, Yingya Zhang, Yitong Huang, Yong Li, You Wu, Yu Liu, Yulin Pan, Yun Zheng, Yuntao Hong, Yupeng Shi, Yutong Feng, Zeyinzi Jiang, Zhen Han, Zhi-Fan Wu, and Ziyu Liu. Wan: Open and Advanced Large-Scale Video Generative Models, 2025.

[33] Qian Wang, Tianyu Wang, Zhenheng Tang, Qinbin Li, Nuo Chen, Jingsheng Liang, and Bingsheng He. MegaAgent: A large-scale autonomous LLM-based multi-agent system without predefined SOPs. In *Findings of the Association for Computational Linguistics: ACL 2025*, pages 4998–5036, Vienna, Austria, July 2025. Association for Computational Linguistics.

[34] Bingyang Wu, Zili Zhang, Zhihao Bai, Xuanzhe Liu, and Xin Jin. Transparent GPU Sharing in Container Clouds for Deep Learning Workloads. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 69–85, Boston, MA, April 2023. USENIX Association.

[35] Chenfei Wu, Jiahao Li, Jingren Zhou, Junyang Lin, Kaiyuan Gao, Kun Yan, Sheng ming Yin, Shuai Bai, Xiao Xu, Yilei Chen, Yuxiang Chen, Zecheng Tang, Zekai Zhang, Zhengyi Wang, An Yang, Bowen Yu, Chen Cheng, Dayiheng Liu, Deqing Li, Hang Zhang, Hao Meng, Hu Wei, Jingyuan Ni, Kai Chen, Kuan Cao, Liang Peng, Lin Qu, Minggang Wu, Peng Wang, Shuting Yu, Tingkun Wen, Wensen Feng, Xiaoxiao Xu, Yi Wang, Yichang Zhang, Yongqiang Zhu, Yujia Wu, Yuxuan Cai, and Zenan Liu. Qwen-Image Technical Report, 2025.

[36] Yuxing Xiang, Xue Li, Kun Qian, Yufan Yang, Diwen Zhu, Wenyuan Yu, Ennan Zhai, Xuanzhe Liu, Xin Jin, and Jingren Zhou. Aegaeon: Effective GPU Pooling for Concurrent LLM Serving on the Market. In *Proceedings of the ACM SIGOPS 31st Symposium on Operating Systems Principles*, SOSP '25, pages 1030–1045, New York, NY, USA, 2025. Association for Computing Machinery.

[37] An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengen Huang, Chenxu Lv, Chujie Zheng, Dayiheng Liu, Fan Zhou, Fei Huang, Feng Hu, Hao Ge, Haoran Wei, Huan Lin, Jialong Tang, Jian Yang, Jianhong Tu, Jianwei Zhang, Jianxin Yang, Jiaxi Yang, Jing Zhou, Jingren Zhou, Junyang Lin, Kai Dang, Keqin Bao, Kexin Yang, Le Yu, Lianghao Deng, Mei Li, Mingfeng Xue, Mingze Li, Pei Zhang, Peng Wang, Qin Zhu, Rui Men, Ruize Gao, Shixuan Liu, Shuang Luo, Tianhao Li, Tianyi Tang, Wenbiao Yin, Xingzhang Ren, Xinyu Wang, Xinyu Zhang, Xuancheng Ren, Yang Fan, Yang Su, Yichang Zhang, Yinger Zhang, Yu Wan, Yuqiong Liu, Zekun Wang, Zeyu Cui, Zhenru Zhang, Zhipeng Zhou, and Zihan Qiu. Qwen3 Technical Report, 2025.

[38] Hancheng Ye, Zhengqi Gao, Mingyuan Ma, Qinsi Wang, Yuzhe Fu, Ming-Yu Chung, Yueqian Lin, Zhijian Liu, Jianyi Zhang, Danyang Zhuo, and Yiran Chen. KVCOMM: Online Cross-context KV-cache Communication for Efficient LLM-based Multi-agent Systems. In *The Thirty-Ninth Annual Conference on Neural Information Processing Systems*, 2025.

[39] Shan Yu, Jiarong Xing, Yifan Qiao, Mingyuan Ma, Yangmin Li, Yang Wang, Shuo Yang, Zhiqiang Xie, Shiyi Cao, Ke Bao, Ion Stoica, Harry Xu, and Ying Sheng. Prism: Unleashing GPU Sharing for Cost-Efficient Multi-LLM Serving, 2025.

[40] Haoyang Zhang, Yirui Zhou, Yuqi Xue, Yiqi Liu, and Jian Huang. G10: Enabling an efficient unified gpu memory and storage architecture with smart tensor migrations. In *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 395–410, 2023.

[41] Hong Zhang, Yupeng Tang, Anurag Khandelwal, and Ion Stoica. SHEPHERD: Serving DNNs in the wild. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 787–808, Boston, MA, April 2023. USENIX Association.

[42] Lianmin Zheng, Liangsheng Yin, Zhiqiang Xie, Chuyue Sun, Jeff Huang, Cody Hao Yu, Shiyi Cao, Christos Kozyrakis, Ion Stoica, Joseph E. Gonzalez, Clark Barrett, and Ying Sheng. SGLang: Efficient Execution of Structured Language Model Programs. In *Advances in Neural Information Processing Systems*, 2024.

A Artifact Appendix

The artifact is used to reproduce the evaluation of this paper, and released under Apache-2.0 license. It includes the source code of Nixie, as well as scripts for evaluations and figure creation.

Hosting

We use two GitHub repository to host the artifact. The repository in https://github.com/XOR-op/Nixie hosts the main project source code, and the repository in https://github.com/XOR-op/nixie-eval hosts the code and scripts for evaluations and figure creation.

Requirements

The ML models selected for evaluation are targeted specifically for the 32 GB RTX 5090. If a GPU with a different VRAM capacity is used for evaluation, we suggest using models that fit into the GPU while maximizing VRAM usage, to best reflect our settings.

ARTIFACT EVALUATED
USENIX
AVAILABLE

μShell: A Microkernel-based FPGA Shell Architecture

Jiyang Chen
Technical University of Munich

Anubhav Panda
Technical University of Munich

Harshavardhan Unnibhavi
Technical University of Munich

Atsushi Koshiba
Tokyo University of Science

Pramod Bhatotia
Technical University of Munich

Abstract

FPGAs are widely adopted in cloud environments to meet the growing demand for high-performance, energy-efficient computing, thanks to their unique reconfigurability and programmability. However, the state-of-the-art design of FPGA shells is built for monolithic applications, creating a fundamental architectural mismatch with real-world applications that are highly modular and composable. Modern applications consist of diverse, independent tasks that are often instantiated as standalone hardware modules on FPGAs. The current FPGA shell design statically connects these modules to compose an application-specific, monolithic accelerator on a single virtual FPGA (vFPGA), leading to fundamental challenges such as limited flexibility, poor scalability, resource inefficiency, high scheduling overheads, and programmability issues.

To bridge this research gap, we introduce μShell, a new hardware-OS co-design that addresses these challenges by applying microkernel principles to FPGA acceleration. μShell treats accelerators as a collection of shareable, composable hardware modules, deploying them into distinct vFPGAs and dynamically linking them with a novel inter-process communication (IPC) mechanism. This approach is further supported by capability-enforced isolation and a component-aware task scheduler. Our implementation of a μShell prototype and its evaluation against a state-of-the-art monolithic shell demonstrate that it effectively deploys real-world applications with improved performance, flexibility, and resource efficiency.

1 Introduction

In cloud environments, hardware accelerators such as GPUs [32,34], ASICs [23], and FPGAs [8,31,90] are now widely used for high-performance computing. In particular, FPGAs have emerged as a promising computing platform because of their unique characteristics, e.g., programmability and energy efficiency, allowing us to reconfigure custom hardware logic specialized for compute-intensive tasks. Their versatility has driven FPGA-based acceleration in domains including video and image processing [58, 63], data compression [26, 48], databases [22, 82], security [42, 52], and AI/ML [6, 114, 134].

Modern FPGA systems typically pair a static *shell*, managing I/O and common services, with a dynamic, reconfigurable region for custom user logic [2, 7, 55]. While state-of-the-art shells [27, 64, 71, 76, 103] leverage partial reconfiguration (PR) to divide this dynamic region into multiple virtual FPGAs (vFPGAs), *they are primarily designed for monolithic applications*—single, independent accelerators that occupy an entire vFPGA.

This paradigm clashes with *the modular, composable nature of modern applications* (§ 2.2). More specifically, real-world applications are modular and composable. They are composed of diverse, independent tasks, and exhibit a high degree of correlation (up to 93%) due to the use of shared functions. To maximize throughput, they use direct data paths between hardware modules, avoiding CPU-driven synchronization and improving performance by up to 2.63×. Existing shells enable such communication only by statically wiring all modules into a single monolithic accelerator on a single vFPGA, at the cost of their modularity and composability.

This architectural mismatch imposes significant challenges on modern FPGA platforms. ***First,*** *the monolithic approach leads to limited flexibility*, as any change to a single module requires a costly redevelopment of the entire accelerator. ***Second****, it severely restricts scalability* by limiting the logic size to the capacity of a single vFPGA. ***Third,*** *it results in resource inefficiency* because the monolithic design instantiates all hardware modules for every application, even if they are shareable. ***Fourth,*** *monolithic accelerators incur substantial scheduling overheads* due to the need to fully reconfigure the entire logic when switching between applications. ***Lastly,*** *existing FPGA APIs offer limited support for dynamic I/O configurations* needed to compose modular applications effectively.

To address these research gaps, we aim to answer the following question: *Can we design a new FPGA platform that enables deploying and executing modern applications on FPGAs in a modular and composable manner?*

To build this on modern FPGA architectures, we need a new FPGA management principle that *(i)* enables partial updates and reuses of user logic designs, *(ii)* fully utilizes PR regions (vFPGAs), *(iii)* allows applications to share overlapped modules, *(iv)* avoids unnecessary reconfigurations, and *(v)* facilitates FPGA application development and deployment with high-level APIs. As an end-to-end solution, *microkernels* [4, 68, 72, 81, 89] inspire us to adapt their principles to the FPGA world. Similar to microkernels that manage applications as *a group of processes* connected through *inter-process communication (IPC)*, we propose to manage FPGA accelerators as *a combination of composable and shareable hardware modules* and connect them with *hardware-level IPC*, while *enforcing isolation*.

To this end, this paper introduces **μShell**, a hardware-OS

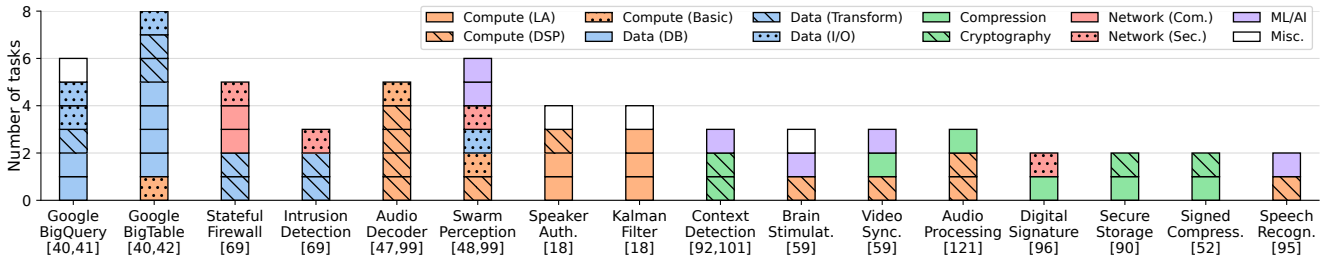


Figure 1: **Modularity analysis** for real-world applications: big data analysis [43–45], networking [79], extended reality [53,115], autonomous vehicles [54,115], wearable devices [20,108,117], cross-domain workloads [67], security [59,105,112], and audio intelligence [111,138]. Classification of their tasks is detailed in Table 1.

Classification	Description & Representative Tasks
Compute (LA)	**Linear algebra**: GEMM, GEMV, LU decomposition.
Compute (DSP)	**Digital signal processing**: FFT/IFFT, rotate order, FIR.
Compute (Basic)	**Basic computations**: Fused Multiply-Add (FMA), etc.
Data (DB)	**Database**: SQL query, hash/sort aggregation/join, etc.
Data (Transform)	**Data transformation**: parsing, filtering, compaction.
Data (I/O)	**I/O tasks**: memory reads/writes, sensing, etc.
Compression	**Data compression/encoding**: GZIP, RLE, Huffman decoding, MPEG encoding/decoding.
Cryptography	**Encryption/decryption**: RSA, AES-GCM, AES-CTR.
Network (Com.)	**Network communication**: TCP/IP, packet switching, connection tracking.
Network (Sec.)	**Network security**: SHA256, IP packet filtering, packet classifier, data descrambling.
ML/AI	**ML and predictive ops**: SVM, Viterbi sensing/decoding, logistic regression, dynamic time warping.

Table 1: Task classifications in Figure 1.

co-design for modular accelerator deployment and execution. Inspired by the microkernel principle, μShell enables *dynamic accelerator composition*, deploying individual hardware modules into distinct vFPGAs and dynamically chaining them together to compose the entire accelerator. To achieve this, μShell adapts microkernel features: *capability-enforced isolation* for secure resource sharing, *component-aware task scheduling* for minimizing reconfiguration overheads, and *IPC* for peer-to-peer data transfers between vFPGAs. μShell provides these features through *a high-level API*, enabling FPGA accelerator composition and deployment in a flexible and dynamic manner. The μShell ecosystem is constructed with three key system components:

(1) **μShell hardware (§4.1).** We design μShell, a microkernel-based shell architecture for dynamic accelerator composition. μShell provides *Capability Enforcement Units (CEUs)* for each vFPGA, which bridges distinct vFPGAs through the IPC mechanism while enforcing capability-based isolation, enabling secure, dynamic accelerator composition.

(2) **μShell OS (§4.2).** We design the μShell OS, privileged software responsible for managing the μShell hardware upon user requests. It provides microkernel features essential for FPGA accelerator composition: capability, address space management, task scheduling, and IPC.

(3) **μShell programming API (§4.3).** We design a high-level API that allows guest applications to compose and deploy FPGA accelerators in a hardware-agnostic manner. The user-space API library transparently communicates with the μShell OS and abstracts complex capability details and hardware architectures from developers.

We implement a μShell prototype on AMD Alveo U280 cards [126]. We build the entire μShell hardware by extending Coyote v2 [103], the state-of-the-art FPGA shell for monolithic accelerators. We also port five real-world applications [59,105,111,112,138] with our μShell prototype. To compose these applications, we leverage pre-built IP cores from FPGA vendors [129] and open-source projects [27,71].

We comprehensively evaluate μShell's performance, scheduling effectiveness, programmability, and FPGA resource overheads using five real-world applications. We highlight that μShell's modular accelerator deployment incurs only 3.3% throughput degradation compared to Coyote v2, the monolithic approach's baseline that achieves optimal performance. In addition, μShell's task scheduler reduces vFPGA reconfigurations by up to 79% compared to Coyote v2, by sharing hardware components across applications.

2 Background and Motivation

2.1 FPGA Shell Architectures

Commercial FPGA products adopted in the cloud [8,31,90] serve as PCIe-connected devices [18,19,126–128], which consist of an FPGA chip and off-chip peripherals for external I/Os such as memory and network ports. The FPGA chip is split into two regions: a static region or *shell*, which is statically configured at system boot and contains I/O modules (e.g., DMA and memory controllers) and common features (e.g., interrupts), and a dynamic region, where custom user logic can be dynamically reconfigured. While vendor-provided shells [2,7,55] often provide a single dynamic region, the state-of-the-art shells in academic studies [27,64,71,76,87,103] use partial reconfiguration (PR) to split it into multiple, isolated *virtual FPGAs (vFPGAs)*, enabling spatial sharing of the FPGA.

Although these state-of-the-art shells promise an efficient use of limited FPGA resources, they assume that applications are ***monolithic***, where user logic is designed as an independent, single accelerator dedicated to its target application and instantiated on a single vFPGA. While this design choice

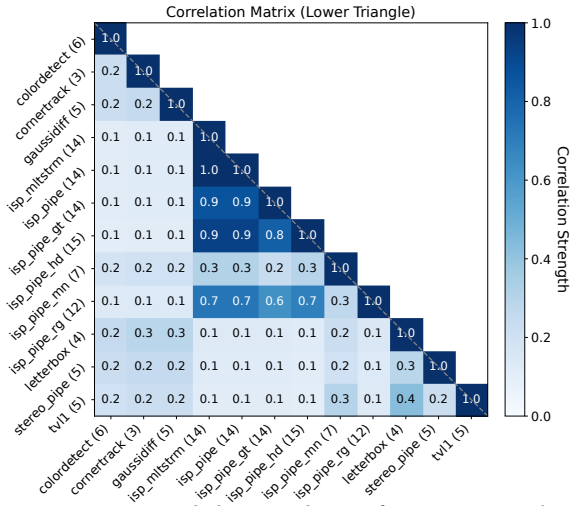


Figure 2: **Composability analysis** of Vitis Vision Library applications [130] based on overlap in library function usage (parentheses indicate call counts).

simplifies an isolation mechanism across vFPGAs, in reality, applications are ***modular*** (§2.2); deploying them on these existing shells poses several challenges (§2.3).

2.2 FPGA Application Requirements

In contrast to the state-of-the-art shells designed for monolithic applications, real-world applications are ***modular and composable***, highlighted by our two empirical studies. First, Figure 1 illustrates the ***modularity*** of real-world applications from prior studies spanning across multiple domains. Each section of bars represents an independent task deployed and executed on any compute device, such as a CPU core, ASIC accelerator, or custom logic on the FPGA. As seen here, modern workloads are not monolithic but composed of *diverse, modular tasks*. Second, Figure 2 shows the ***composability*** study of HLS applications developed with the Vitis Vision library [130], where we analyze the library functions called by each application code and visualize matches of the same function calls between two applications as the correlation strength. We observe that up to 93% of application pairs have shared functions and more than 20% of them have high correlation (more than 0.5), indicating that applications are deployed as a combination of *sharable and composable functions*.

These applications are typically designed and built on FPGAs as a combination of multiple *hardware modules*, each dedicated to a specific task. To maximize acceleration effectiveness, establishing data paths between modules is essential for enabling ***direct communication***, thereby significantly increasing throughput [43, 67, 115, 121]. Figure 3 demonstrates the effectiveness of direct communications, compared to a CPU-driven synchronization, where a host CPU sequentially invokes FPGA accelerators that comprise each application. The results highlight that direct communication achieves up to 3.7× by avoiding unnecessary CPU-FPGA communications.

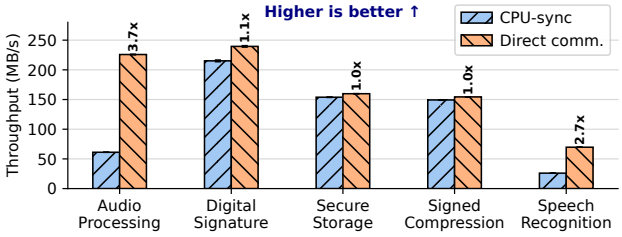


Figure 3: **Direct communication effectiveness** for our target applications [59, 105, 111, 112, 138] (§6).

I/O features	HLS tools [29, 125]	FPGA APIs [41, 46, 66]	FPGA shells [64, 71, 103]	*µ*Shell **(Proposal)**
Direct I/O	✓	✓	✓	✓
I/O switching	-	✓	-	✓
Inter-vFPGA	-	-	✓	✓

Table 2: I/O features offered by FPGA systems/APIs.

2.3 State-of-the-art FPGA Shells

Due to the fundamental mismatch between state-of-the-art shells and real-world applications, existing FPGA platforms deploy applications as *monolithic* accelerators that combine all submodules into a single static accelerator on a single vFPGA, imposing five fundamental challenges. We have a detailed comparison with the state-of-the-art FPGA shells and platforms in §7.

(1) Limited flexibility. First, the monolithic design loses the flexibility of FPGA applications. Real-world applications are composed of multiple tasks, as shown in Figure 1, while they are developed as a single-purpose, static accelerator on the FPGA. Although this design fully optimizes performance, it increases the logic's size and complexity, making it error-prone and hard to debug. Hardware bugs are mostly caused at certain execution stages and intersections between submodules [75]. Moreover, minor changes to a specific submodule (e.g., applying different ML algorithms for speech recognition [111]) affect the entire accelerator design, forcing users to recompile and resynthesize the entire design. It substantially increases development efforts.

(2) Limited scalability. Second, existing shells leveraging partial reconfiguration (PR) limit the scalability of FPGA accelerators. Although multiple vFPGAs can increase the concurrency of FPGA applications, more vFPGAs reduce the available resources. Existing shells, which allow user logic to be deployed to a single vFPGA, limit the deployable logic size to the capacity of that vFPGA. Figure 4 analyzes available FPGA resources per vFPGA on Coyote v2 [103] deployed on the U280 FPGA with different numbers of vFPGAs, highlighting that available resources are severely restricted, i.e., only 10.8% to 12.5% of the U280 capacity when we have 8 vFPGAs.

(3) Resource inefficiency. Third, the monolithic accelerator deployment leads to resource inefficiency. As demonstrated in Figure 2, FPGA applications likely contain the same modules, which can be shared. In a monolithic design, these

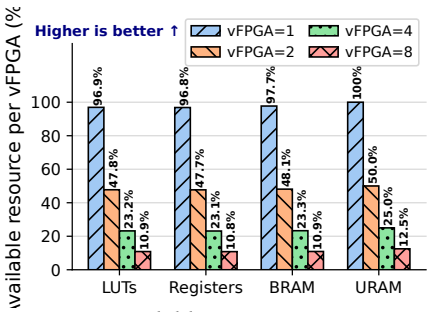


Figure 4: Available FPGA resources per vFPGA.

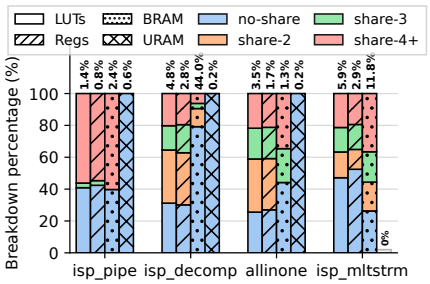


Figure 5: Resource usage analysis. Bar labels show relatives to U280 capacity.

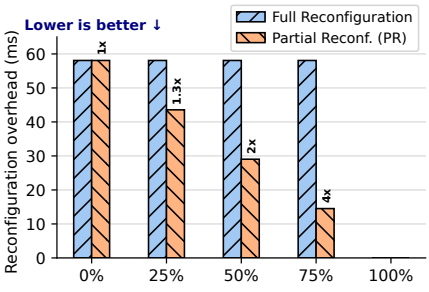


Figure 6: Reconfiguration overhead. The x-axis shows the shared logic ratio.

shareable components are unnecessarily duplicated because each application is deployed as an independent module, leading to wasted FPGA resources. Figure 5 visualizes FPGA resource usage of non-shared and shared modules of four representative applications from the Vitis Vision Library [130] (share-X indicates modules shared by X applications from the library). It highlights that shared components can account for up to 80% of the application's total resource usage.

(4) Scheduling overheads. Fourth, monolithic accelerators induce significant scheduling overheads when switching tasks deployed on the same vFPGA (temporal sharing). This process invokes reconfiguration of the entire accelerator logic, leading to substantial (ms-order) performance penalties [71]. While some shells support accelerator reuse to avoid reconfiguration [71], this feature is only available when the accelerators used by two applications are identical. Figure 6 demonstrates expected task switching (reconfiguration) overheads, comparing two cases: the baseline that reconfigures the entire vFPGA, and the ideal case that only reconfigures non-shared components. The result indicates that we can potentially mitigate the switching overheads if the FPGA platform allows us to reconfigure only an independent portion of the logic.

(5) Limited programmability. Fifth, existing shells and programming APIs are primarily designed for monolithic accelerators and offer limited programmability for dynamic I/O configurations between hardware modules. Table 2 compares existing FPGA systems in terms of I/O functions and their programmability. HLS compiler-based approaches [29, 125] ease the description of inter-module communication, but are limited to a single accelerator design. FPGA APIs [41, 46] support dynamic I/O switching between modules but not direct communication across vFPGAs. Some shells, such as Coyote [71], enable inter-vFPGA communication, while they are statically configured as a function of the shell.

2.4 Our Proposal and Design Goals

To overcome these challenges, we propose μShell, a microkernel-based FPGA shell architecture. Inspired by the microkernel concept [4, 68, 81, 89], μShell enables *the modular deployment and execution of FPGA accelerators*. Unlike the monolithic approach, μShell deploys an application across separate vFPGAs as hardware modules and dynamically chains them to compose the entire application. μShell realizes this by offering microkernels' core system properties: capability management, address space isolation, task management/scheduling, and IPC. We design our μShell architecture to meet the following design goals:

(1) **Capability-enforced isolation.** μShell leverages capabilities [16, 37, 74, 78] to ensure robust isolation across shared and non-shared accelerator modules and memory buffers in a multi-tenant environment.

(2) **Inter-process communication (IPC).** μShell allows applications to establish peer-to-peer communication channels between accelerators deployed on vFPGAs, enabling fast I/O communications while improving flexibility.

(3) **Component-aware task scheduling.** μShell schedules tasks deployed on FPGAs while accounting for hardware modules shared across multiple applications and minimizing the number of reconfigurations.

(4) **API for dynamic accelerator composition.** μShell provides a high-level API that allows developers to describe application dataflow across components and dynamically deploy them in a modular and flexible manner.

The μShell design comprehensively addresses the challenges outlined above. ***First,*** it ensures flexibility, because applications can be updated without involving unchanged components. ***Second,*** it ensures scalability because μShell can chain any number of vFPGAs to construct applications, thereby fully utilizing all available FPGA resources. ***Third,*** it improves resource efficiency by allowing multiple applications to share components deployed on the same FPGA. ***Fourth,*** μShell induces less task switching overhead because only non-shared components need to be reconfigured. ***Fifth,*** μShell provides an easy-to-use API that enables developers to design and deploy their modular, composable applications within the μShell ecosystem.

3 Overview

3.1 System Overview

Figure 7 shows the overview of μShell, comprised of guest applications using the *μShell API*, privileged *μShell OS*, and

the μShell substrate offering multiple vFPGAs. At the core of each vFPGA is a *Capability Enforcement Unit* (CEU), which acts as a hardware gatekeeper, regulating data transfers, connection setup, access control, and memory allocation for the associated accelerator module.

Hardware components. *μShell* (§4.1) is deployed as a privileged component at system boot and provides multiple vFPGAs. Besides common shell components such as a PCIe bridge IP, it offers dedicated CEUs that connect each vFPGA to streaming data paths via the data interconnect. Each CEU enforces capabilities, allowing only authorized data transfers between vFPGAs, while the μShell OS restricts guest application access to prevent tampering with shell components and capabilities.

Software components. *μShell OS* (§4.2) and *μShell API library* (§4.3) are two key software components. The privileged μShell OS offers four microkernel-based features for secure accelerator composition: (1) *capabilities* [50, 123] to restrict access to accelerator components like user logic on vFPGAs and memory buffers; (2) *address space management* to ensure that a group of accelerator components sees the same virtual address space; (3) *task scheduling* to enable temporal and spatial sharing of vFPGAs between applications; (4) *IPC* to enable communication between the components. The μShell API library provides a hardware-agnostic interface for applications to interact with the OS and to deploy and execute accelerators. Since many real-world FPGA applications [51, 57, 93, 97, 107] are composed of modular hardware components and designed as pipelined streaming dataflows, the μShell API aligns with this programming model and does not limit the generality of applications.

System workflow. During initialization, guest applications use μShell APIs to construct dataflow graphs (DFGs) by defining computational tasks, memory buffers, and their data dependencies. These are sent to the μShell OS, which verifies DFG correctness and consistency with capability constraints. Based on programmed logic on vFPGAs and other running tasks, the μShell OS performs task scheduling, task mapping to vFPGA, reconfiguration, memory allocation, and capability initialization. At runtime, the μShell OS enforces capability-based access control and orchestrates data transfers. The system ensures secure resource cleanup through automatic capability reclamation, eliminating manual resource management overhead while maintaining strict security boundaries.

3.2 Technical Challenges and Key Ideas

While the microkernel concept is the key to deploy and execute modular, composable FPGA applications (§2.4), the nature of FPGAs and their toolchains pose several challenges to adapting microkernel principles [37, 68, 81]. We tackle this challenge with a hardware-OS co-design approach. Specifically, we build the μShell ecosystem to address four key design challenges:

#1: Isolation and sharing for accelerator components. In contrast to existing shells for monolithic applications [64, 71, 76, 103], which dedicate each vFPGA to a single application and ensure per-vFPGA isolation, μShell deploys applications as a set of accelerator components (*hardware tasks*) distributed across vFPGAs, similar to microkernel applications comprising multiple services [25]. This design introduces two unique challenges. First, existing shells, including Coyote [71, 104], enforce isolation at the vFPGA granularity, assuming that each vFPGA runs a single monolithic application (detailed in Table 7). As a result, they lack mechanisms to safely compose or share independently deployed hardware components across vFPGAs, since there is no fine-grained control over inter-component communication or memory access. Second, the non-preemptive nature of FPGAs [69] complicates the management of internal state when hardware components are reused across applications, making secure sharing difficult without explicit isolation support.

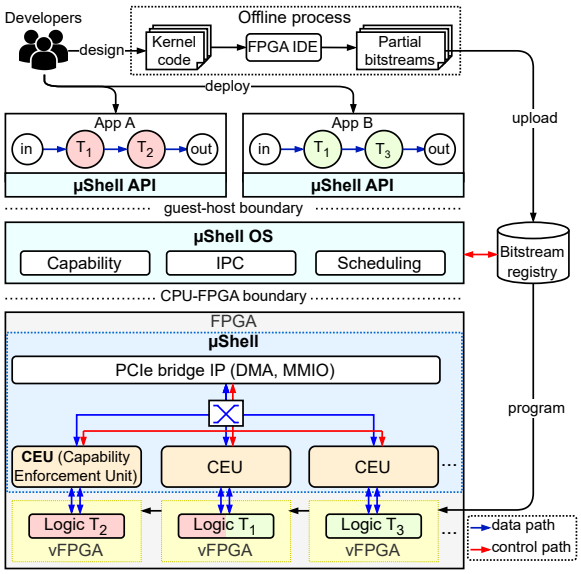


Figure 7: Overview of the μShell system.

Key idea. We design a unique capability-based, hardware-enforced isolation mechanism that restricts IPC channel establishment and memory accesses, achieved by μShell CEUs (§4.1) and OS (§4.2). μShell ensures that internal states of shared hardware components are clearly flushed before switching applications.

#2: Secure IPC establishment across vFPGAs. Constructing secure chains of user logic and memory buffers requires robust resource isolation. μShell adapts the microkernel's IPC mechanism to dynamically establish data paths between accelerator components. To achieve this, we require a reconfigurable I/O switching mechanism that establishes streaming channels between vFPGAs and memory buffers on demand, while maintaining isolation. Such cross-vFPGA communications are restricted in existing shells; they either do not provide any interconnects and rely on static data paths in monolithic designs [5, 64, 80, 113, 119, 135] or require recompiling the entire shell to bridge vFPGAs [71, 79, 104]. As

a result, they cannot dynamically establish communication channels between independently deployed components. Other designs [65, 91] support flexible data movement but lack fine-grained access control, making it difficult to securely connect components across application boundaries.

Key idea. We introduce a fast, flexible IPC mechanism using CEUs and a centralized interconnect (§4.1). μShell OS's IPC module (§4.2) dynamically and securely configures IPC channels by enforcing capabilities with minimal overhead. μShell is the first FPGA shell that functions IPC between components with secure isolation properties.

#3: Hardware task scheduling and deployment. Scheduling and deploying a set of hardware tasks across a limited number of vFPGAs is challenging. While this approach enables component sharing across applications, current PR technologies on FPGAs [12, 56] limit deployment flexibility because PR regions (vFPGAs) are not interchangeable; each vFPGA requires a dedicated bitstream. Moreover, although component sharing potentially improves utilization and reduces reconfiguration overhead (§2.3), it may lower throughput compared to the exclusive use of the entire FPGA, especially when reconfiguration occurs frequently. Prior hardware schedulers either overlook this trade-off [71, 91] or disable PR at the cost of flexibility [64].

Key idea. μShell OS's component-aware task scheduler (§4.2) allows applications to share accelerator components while retaining isolation and throughput. μShell OS also enables flexible hardware task deployment on any system-provided vFPGAs by managing multiple bitstreams.

#4: Programming APIs for dynamic accelerator composition. The lack of FPGA shells that support dynamic accelerator chaining limits software libraries from providing easy ways to pipeline multiple applications. As a result, existing frameworks [41, 46, 76, 131] do not natively support connecting and orchestrating multiple pipelined applications.

Key idea. We present a user-space API library (§4.3) that enables composing dataflows and deploying them without manual communication or resource management.

4 Design

We introduce design details of the core system components: μShell, μShell OS, and μShell API library.

4.1 μShell Hardware

We first introduce the μShell hardware architecture, illustrated in Figure 8. μShell comprises components for common shell functionalities, i.e., a PCIe bridge IP, a PR controller, and a DMA request scheduler, and μShell-specific components for capability enforcement and IPC, i.e., a data interconnect and per-vFPGA CEUs. The PCIe bridge IP provides two host-FPGA communication interfaces: a memory-mapped interface for control paths (e.g., AXI4-Lite [13]) and a

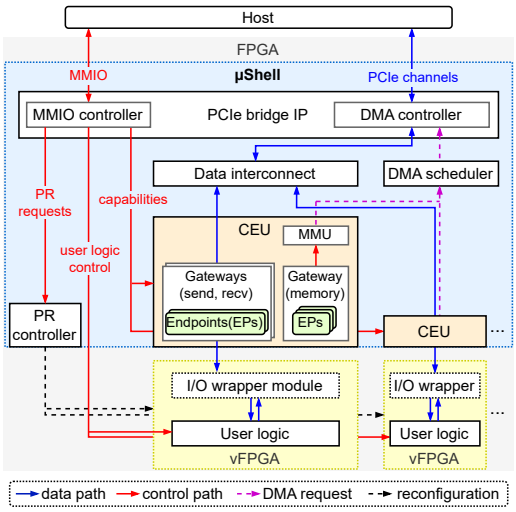


Figure 8: μShell hardware architecture.

streaming interface for data paths (e.g., AXI4-Stream [14]). The PR controller reconfigures a selected vFPGA through the ICAP interface [1] without interrupting other hardware components. The data interconnect provides a configurable peer-to-peer link between vFPGAs and host memory. Through the interconnect, the CEUs establish capability-enforced channels by configuring *endpoints* depending on I/O types (send, receive, or memory). For memory accesses from vFPGAs, the CEUs issue DMA requests, which are handled by the DMA scheduler in a round-robin fashion. For vFPGAs, μShell provides an I/O wrapper module that abstracts multiple input and output ports of user logic.

Capabilities. The μShell framework employs capability-based isolation as its fundamental access control mechanism for accelerator composition on the FPGA. Capabilities are unforgeable tokens that grant specific permissions to access system resources, providing both authentication and authorization in a single primitive [16, 49, 68, 123]. This approach ensures that user logic can only access resources for which they hold valid capabilities, enabling fine-grained access control.

μShell uses capability-based isolation to protect IPC between vFPGAs and memory access. *Object capabilities* authorize access to specific user logic instances and communication endpoints, and *memory capabilities* constrain accesses to approved address ranges and include permissions (read/write). Capabilities are created and managed per application by the μShell OS and enforced at runtime by the CEUs, preventing cross-application interference.

Capability Enforcement Unit (CEU). The CEU enables IPC and capability enforcement and is designed to offer three key functions: handling multiple connections for user logic, safely invalidating unauthorized accesses, and enabling time-sharing of user logic. To achieve this, the CEU includes configurable *gateways* for each communication channel, which act as verification checkpoints for IPC. There are three gateways for different I/O types: *send gateway* to manage

data transfers from the hosted vFPGA, *a receive gateway* to validate incoming data streams, and *a memory gateway* to validate memory accesses and issue DMA requests.

Figure 9 illustrates the CEU architecture. Each gateway consists of *endpoints* and *validators*. Endpoints are registers that store capabilities granted to the associated user logic. Send/receive, and memory endpoints are created from object and memory capabilities, respectively. Creating memory endpoints also involves the MMU registering a page table entry. The gateway establishes communication channels only with authorized vFPGAs or memory spaces whose capabilities are configured as endpoints, and rejects unauthorized requests. Endpoints of these gateways are initialized or updated only once per application execution at the deployment phase. During execution, incoming/outgoing data is seamlessly validated by its gateways without capability updates.

Given that user logic can have multiple input/output ports, a single gateway can configure multiple endpoints to establish multiple channels. The send and receive endpoints use *port IDs* to identify specific ports of user logic. The port IDs are transferred as part of execution requests from applications via the μShell API (§4.3) and enable gateways to internally manage port mappings across different channels.

Validation. The validators detect unauthorized requests and stop them. When sending requests, the send gateway's validator receives a port ID from the user logic and checks whether the associated send endpoint has been created. The gateway then generates *a route ID* containing the port ID, the sender's, and receiver's vFPGA IDs. The route ID is propagated to the data interconnect and the receiver, along with data streams, to guide routing and enable the receiver's capability checks. When receiving requests, the receiver's validator uses the route ID to identify the sender's vFPGA and verify whether the receiver has the associated capability. It then extracts a port ID from the route ID and transfers the data to the associated user-logic port. For memory requests, the validator checks whether the memory capability covers the specified address range.

On failure, the validator drops the request and interrupts μShell OS. It safely suspends the execution without affecting other components: incoming streams are drained to avoid hangs or data leakage, and invalid memory requests are rejected without issuing DMA, ensuring no transaction occurs.

Component sharing. The CEU enables temporal sharing of user logic between applications. While custom IPs can deliver higher performance, standardized reusable modules (e.g., AFI [9]) reduce development cost and improve composability. μShell currently supports sharing of stateless accelerators across applications, as reliably resetting internal hardware state is challenging. For non-shared stateful applications, on-chip memories (e.g., LUTRAMs and BRAMs) can be used to maintain their internal states. μShell also supports off-chip memory such as HBM and DRAM, which are treated similarly to the host memory. These off-chip memory regions are used as data buffers for communication between the host and accelerators, e.g., input/output buffers for data transfer, rather than as persistent, application-private state tied to a specific accelerator instance across invocations. The shell maintains only two types of application-specific contexts: CEU endpoints and MMU page tables, which are cleared upon application switches. Since FPGAs are non-preemptive, μShell waits for the current execution to finish before updating capability mappings of the associated CEUs. It then applies new routing capabilities before switching applications, without saving or restoring the accelerator state. μShell additionally resets the residual state in the accelerators to mitigate potential data leakage. While a full hardware reset of all components is non-trivial, as states can be distributed across many elements (BRAMs, flip-flops, etc.), μShell relies on the fact that HLS provides easy-to-use reset mechanisms to clear internal states (e.g., reset pragma, dedicated reset port [11]).

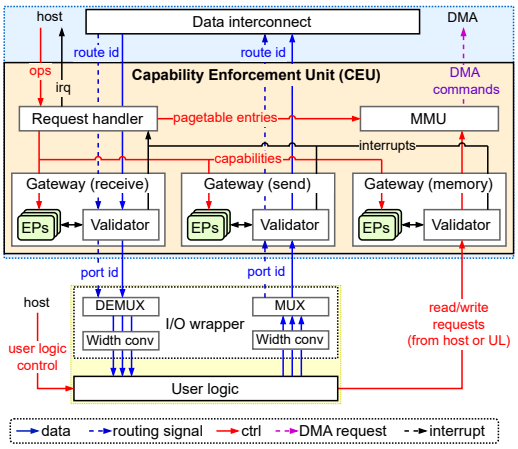


Figure 9: Capability Enforcement Unit (CEU).

Data interconnect. The data interconnect acts as a central switch, routing data streams between vFPGAs and host memory. It maintains static connections to all sources and destinations, while actual routing is dynamically configured via route IDs attached to incoming data streams. The interconnect uses the route ID to identify the correct destination and forwards data and accompanying signals that mark the start and the end of data streams. It provides independent data paths between source and destination pairs and supports multiple concurrent streams without interference. The interconnect uses a round-robin scheduling algorithm for streams that go to or come from the same vFPGA.

I/O wrapper module. The I/O wrapper module abstracts μShell's unified data stream interfaces from user logic. It contains a multiplexer (MUX) and a demultiplexer (DEMUX) to route I/O streams from and to a single shell-side data channel. It includes a width converter to reconcile mismatches between the data widths of the user logic ports and the μShell interface. As a result, user logic can be deployed on μShell without requiring any modification.

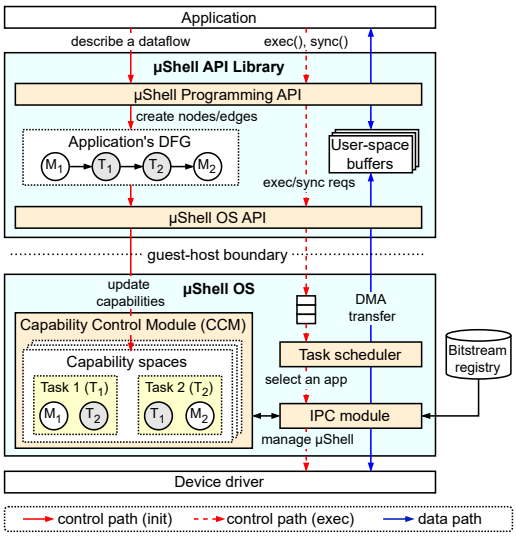


Figure 10: μShell API library and OS architecture.

4.2 μShell OS

We propose μShell OS, a privileged software designed to adapt the microkernel principle to resource management for modular, composable FPGA applications. Unlike traditional microkernels for CPU programs [4, 39, 68, 81], μShell repurposes microkernel features for FPGA applications, accounting for the fundamental limitations of modern FPGA architectures and toolchains.

Figure 10 shows the architecture of the software components. μShell OS consists of three key components: the *capability control module (CCM)* for managing capabilities, the *dataflow task scheduler* for scheduling hardware tasks, and the *IPC module* for deploying and invoking tasks on vFPGAs and enabling IPC. μShell OS provides the microkernel's fundamental features: *capability management*, *address space management*, *task (thread) management*, and *IPC*.

Capability management. The CCM is responsible for maintaining a lifecycle of application capabilities with fine-grained control. Similar to seL4 [68] and M^3 [16], the μShell OS manages all application objects (tasks and memory) using capabilities for robust isolation, while it is tuned for building and executing accelerator chains on the FPGA.

The CCM creates *capability spaces* for individual applications to guarantee multi-tenant isolation. The CCM ensures that capabilities remain unforgeable, as guest applications can only access their own capability spaces via μShell APIs. Unlike existing microkernels that manage capabilities for CPU processes [4, 37, 68], the CCM adapts capabilities to define strict data flows between hardware tasks and memory buffers for secure accelerator composition. The CCM treats a guest CPU application as a *parent* and allows it to manage capabilities for these accelerator components as children or *dataflow nodes* through three primary operations: *create*, *delegate*, and *revoke*.

An application begins by creating object/memory capabilities for dataflow nodes, which allows the application to possess a parent capability with delegation rights and request a new, subordinate capability for nodes through the delegation. Created/delegated capabilities are securely maintained through a tree structure: child capabilities never have more privileges than their parent, preventing privilege escalation. The revocation process verifies every access to capabilities and safely retires capabilities with zero usage count. Once an application is complete, its capabilities are immediately removed.

Delegated capabilities enable nodes to communicate via IPC or memory accesses. For example, as shown in Figure 10, when a task (T_1) sends data to another (T_2), T_1 must have T_2's capability with a write permission, while T_2 owns T_1's capability with a read permission.

Address spaces. μShell OS maintains and exposes virtual memory address spaces not for CPU programs but hardware tasks running on the FPGA. It allows applications to allocate FPGA-accessible memory buffers while enforcing isolation. In μShell, applications do not share page tables; each application maintains its own address space within a dedicated MMU with its TLB, managed by the CCM. The CCM is responsible for memory allocation and mapping, ensuring that all access permissions are maintained per-application and that isolation is enforced across vFPGAs. Since μShell guarantees that each application has exclusive access to its assigned vFPGAs during execution, TLB operations (e.g., flushing) remain localized and do not impact other applications.

Tasks and threads. Instead of CPU tasks and threads, μShell manages *hardware tasks* that execute on specific user logic. Unlike task execution on CPUs, hardware task deployment and execution require FPGA reconfiguration by loading a bitstream into a PR region (vFPGA) to initialize user logic, posing two challenges. First, vFPGAs are not interchangeable and their capacity cannot be changed dynamically due to limitations of existing FPGA toolchains. Current PR technologies [12, 56] require bitstreams dedicated to each vFPGA, which complicates the deployment and switching of user logic across vFPGAs. Second, FPGA reconfiguration is known to be a time-consuming process, which induces performance overheads of hundreds of milliseconds [71].

We address these limitations with two approaches. First, the μShell ecosystem requires developers to provide multiple bitstreams for each pair of user logic and vFPGA, so that the μShell OS can seamlessly select the bitstream associated with the target vFPGA. This constraint follows state-of-the-art PR-enabled FPGA shells [64, 71, 103] (§8). μShell OS maintains multiple bitstreams per user logic in the bitstream registry. Specifically, it classifies bitstreams by assigning unique IDs to user logic and vFPGAs. These IDs are attached to each bitstream depending on its associated logic and vFPGA.

Second, we propose the dataflow task scheduler, which mitigates the reconfiguration overhead by promoting component reuse, i.e., deploying tasks using user logic that has already been programmed on vFPGAs. Algorithm 1 details our scheduling algorithm. The scheduler employs a non-

Algorithm 1: The component-aware scheduler.

```
1  schedule_app(priority_array, vFPGAs)
2  begin
       /* Select apps with highest priority */
3      foreach priority in priority_array do
4          if priority_array[priority] != null then
5              candidate_array ← priority_array[priority];
6              break;
7          priority ← priority - 1;
8      end
       /* Select app with the highest overlap with free vFPGAs */
9      foreach app in candidate_array do
10         app.overlap ← overlap_with_free_vFPGAs(app, vFPGAs);
11     end
12     candidate_app ← max_overlap(candidate_array);
       /* Check if there are enough vFPGAs to schedule the app */
13     if candidate_app.size > free_vFPGAs.size() then
14         return null;
15     foreach priority in priority_array do
16         foreach app in priority_array[priority] do
17             app.time ← app.time + 1;
18         end
           /* Promote long-waited app priority */
19         if app.time > threshhold then
20             app.priority ← app.priority + 1;
21             app.time ← 0;
22         priority ← priority - 1;
23     end
24     return candidate_app
25 end
```

preemptive policy to avoid expensive context switches on FPGAs [17]. It continuously monitors all vFPGAs, tracking programmed user logic and execution status. Any system event (e.g., a request arrival or completion) triggers the scheduling of a new task. The scheduler first considers applications with the highest priority (L3-8). For those with the same priority, it selects an application with the greatest overlap between the requested user logic and the logic programmed on vFPGAs (L9-11). To prevent starvation, an application that has waited longer than a threshold has its priority promoted (L19-22). Once the application is selected, the scheduler forwards the execution request to the IPC module, which is responsible for the μShell configuration and execution invocation. When components have imbalanced execution times that may lead to pipeline stalls, this reflects a common challenge in streaming and pipeline-based accelerator design. It is the developer's responsibility to partition the application such that the processing latency of each stage is balanced to maximize throughput.

Inter-Process Communication (IPC). IPC in a hardware context differs fundamentally from that for CPU processes [68, 81]. CPU-based IPC relies on blocking remote procedure calls (RPC), which require context switches to the OS kernel. In contrast, IPC between hardware tasks is implemented as non-blocking, direct data transfers via streaming protocols (e.g., AXI4-Stream [14]) across multiple hardware modules, i.e., sender/receiver logic and μShell. This hardware-level IPC is faster and more efficient than CPU-based IPC because all components run in parallel and all operations occur without interruption. However, due to the lack of context-switch mechanisms in FPGAs [86, 109, 120], IPC initialization on FPGAs is less flexible than on CPUs. Before launching the entire execution, all hardware components must be instantiated on vFPGAs and IPC channels established between them.

The IPC module of μShell OS is responsible for IPC management and for invoking hardware tasks. The module handles low-level hardware management to trigger the entire dataflow execution on the FPGA, such as vFPGA reconfiguration, CEU and MMU configuration, and DMA invocation. It receives execution requests from the scheduler and reconfigures vFPGAs if necessary. It then refers to the corresponding capability space maintained by the CCM and configures endpoints on the associated CEUs and MMU page tables to establish IPC channels, ensuring all communication between hardware tasks occurs without involving the μShell OS.

High-level API	Description
create_task(*name*)	Create a task node.
create_buffer(*data*)	Create a host data buffer.
dataflow()	Instantiate a dataflow graph.
dataflow::connect(*src, dst*)	Connect two tasks.
dataflow::execute()	Execute the dataflow.

Table 3: μShell API for building host-side control logic.

```
// Create tasks and data buffers as DFG nodes
sha = ushell::create_task("sha");
rsa = ushell::create_task("rsa");
in  = ushell::create_buffer(host_in);
out = ushell::create_buffer(host_out);
// Describe a dataflow
df = ushell::dataflow();
df.connect(in, sha.in);
df.connect(sha.out, rsa.in);
df.connect(rsa.out, out);
// Validate and execute the dataflow
df.execute();
```

Listing 1: An example written by μShell API.

4.3 μShell API and Programming Model

The μShell programming API and library are designed to improve programmability, illustrated in Figure 10. The API lets developers describe only an accelerator's DFG, and the library transparently handles μShell's low-level management, e.g., capability enforcement, via the primitive OS API.

Programming API. Table 3 shows μShell APIs. Applications initialize a dataflow graph (DFG) via the *dataflow()* function. The DFG consists of three components: *tasks*, representing computational nodes on vFPGAs and created with *create_task()*; *buffers*, representing input/output memory and created with *create_buffer()*; and *edges*, representing data dependencies between tasks and buffers and created with *connect()*. Once the complete DFG is constructed, it can be validated and executed using *execute()*. These APIs abstract low-level FPGA management from developers, while the underlying μShell OS transparently orchestrates end-to-end execution.

Example code. Listing 1 presents a code example of a digital signature accelerator composed of two logics: SHA and RSA.

First, the application creates tasks and buffers as dataflow nodes (L2-5). Upon creating these nodes, the μShell library transparently communicates with the CCM and creates their capabilities. Next, the application instantiates a DFG and describes edges between the nodes (L8-11). These edge creations invoke capability delegation, e.g., the input buffer *in* connects to the SHA logic *sha*, delegating the in's memory capability to SHA with a read permission, granting read access. Lastly, it submits the DFG as an execution request (L14). The μShell library then validates the completeness and correctness of the DFG and forwards it to the task scheduler.

5 Implementation

We implement μShell by redesigning and extending Coyote v2 [35] in both hardware and software. We choose Coyote v2 as the base for our implementation because it provides a mature, open-source FPGA shell with a well-defined hardware-software stack on the Xilinx Alveo U280. This allows us to focus on designing and evaluating our proposed mechanisms without requiring significant engineering effort. Our extensions include the AXI4-Stream Switch as the data interconnect (supporting up to 15 vFPGAs), per-vFPGA CEUs, the μShell OS, and the μShell API library.

5.1 μShell Hardware Shell

To support IPC and capability-based isolation, we extend Coyote's hardware shell with two components.

Data interconnect. We use Xilinx's AXI Stream Switch IP [10] to route data between vFPGAs and the host. Since the AXI Stream Switch supports up to 16 master-slave interface pairs, μShell can support up to 15 vFPGAs, with one pair reserved for communication with host memory. The interconnect supports custom-length destination (TDEST) signals for flexible routing configurations and is configured to accommodate varying numbers of vFPGAs in the shell.

Capability Enforcement Unit. We implement one CEU per vFPGA in the static shell region. Each CEU augments Coyote's per-vFPGA MMU with: (1) a memory gateway that stores capabilities as registers (base/bound addresses, access permissions, validity flags) and performs bounds checks on all DMA requests, and (2) send/receive gateways that manage object capabilities using vFPGA IDs for destination lookup and verification. Placing these components in the static region prevents tampering via partial reconfiguration. The CEU also generates signals to notify the host OS of capability violations.

5.2 μShell OS

To support communication between the host application and FPGA driver, we extend Coyote's userspace system software service with μShell OS features for capability management. We implement the dataflow scheduler within this service.

Capability Control Manager (CCM). The CCM manages capabilities through three core operations: allocation, delegation, and revocation. In software, the CCM is implemented as a kernel module that maintains a capability table indexed by capability IDs. When allocating a new capability, it creates an entry with memory bounds and permissions, then programs the corresponding CEU registers via MMIO. For delegation, it creates a new derived capability with restricted permissions and updates both its internal table and the CEU hardware state. Revocation traverses the capability tree to invalidate all derived capabilities before clearing the hardware registers. The CCM tracks process ownership to automatically revoke capabilities when processes terminate, preventing resource leaks.

Task scheduler. We implement μShell's component-aware policy in its dataflow scheduler. The scheduler receives execution requests from host applications and deploys the application logic through the FPGA driver. This design overcomes a limitation in Coyote's server-client model, where per-vFPGA reconfiguration servers have their own local run queues, preventing global visibility into waiting applications.

5.3 μShell API Library

We implement a two-tier C++ library that provides both high-level APIs for FPGA-accelerator composition and low-level APIs for μShell OS control. The high-level API allows users to construct dataflow graphs through node creation and edge connection calls. When users invoke these APIs, the library builds an internal graph representation and automatically determines the required capabilities for each connection. Upon graph execution, the runtime evaluates the dataflow topology and issues corresponding requests to the kernel driver: node creation triggers vFPGA allocation and configuration, while edge creation establishes data paths between components with appropriate capabilities. The primitive OS API provides direct access to these kernel operations for advanced users who need fine-grained control. The library ensures each component receives only the minimum required capabilities based on its connections in the dataflow graph.

6 Evaluation

We evaluate μShell from the following dimensions: performance (§6.1), scheduling (§6.2), deployment overheads (§6.3), programmability (§6.4), and resource overheads (§6.5).

Experimental setup. We perform the experiments on a server equipped with an AMD U280 FPGA [126]. The server has an AMD EPYC 7413 processor operating at 2.65 GHz and running NixOS 23.0 with Linux kernel 6.9. We configure μShell with up to eight vFPGAs, which is equivalent to state-of-the-art shells [64, 103]. We do not support inter-vFPGA communication through HBM [91], but applications can use it as a scratchpad memory to store data for execution. Since other open-source FPGA systems [64, 76] do not support our

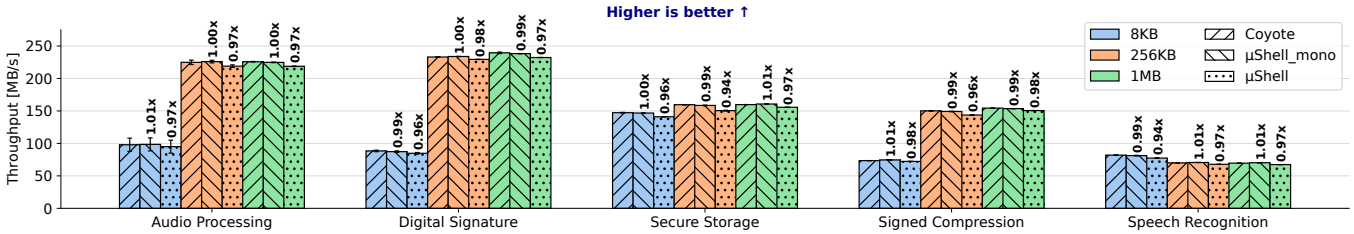


Figure 11: µShell's end-to-end performance against Coyote v2 (*Coyote*) and the monolithic implementation (*µShell_mono*).

Application	Components		
Audio Processing [138]	FFT	Quantize	RLE
Digital Signature [112]	SHA256	RSA	
Secure Storage [105]	RLE	AES-CTR	
Signed Compression [59]	RLE	RSA	
Speech Recognition [111]	FFT	SVM	

Table 4: Applications and their components used in the evaluation: signal processing (FFT, quantization), cryptography (AES-CTR, SHA256, RSA), compression (RLE).

target hardware platform, and porting them would require substantial engineering effort beyond the scope of this work, we use Coyote v2 as the primary baseline for comparison. We note that µShell is not tied to this platform and can be ported to smaller FPGAs that support partial reconfiguration.

Application benchmarks. Our evaluation uses five applications (Table 4) that comprise realistic kernel-level workloads from open-source libraries [129]. While we do not include full end-to-end applications, these benchmarks consist of representative dataflow components commonly found in real-world FPGA workloads (e.g., streaming pipelines and multi-stage processing). This design allows us to effectively demonstrate µShell's capabilities in real-world scenarios.

6.1 Performance

First, we evaluate the end-to-end performance of applications to show the overheads introduced by µShell. We measure the I/O throughput of the five applications listed in Table 4 and run them on Coyote v2 and µShell by varying data sizes (8 KiB, 256 KiB, and 1 MiB), reporting the average across 10 runs.

For µShell, we evaluate two variants: µShell, which allocates vFPGAs per component and composes accelerators from multiple vFPGAs, and µShell_mono, which deploys monolithic accelerators on a single vFPGA, as Coyote v2 does. For all setups, we use host DRAM as data storage and transfer data directly between the memory and the FPGA.

Results. Figure 11 shows the I/O throughput across different data sizes. µShell delivers performance comparable to Coyote v2 for all applications, averaging 3.3% lower throughput. Audio processing and speech recognition applications incur 2.9% and 3.9% overhead, respectively, while digital signature, secure storage, and signed compression incur 2.9%, 4.2%, and 2.8% overhead, respectively. We observe standard deviations below 10.2 MB/s across all experiments, and most measurements exhibit standard deviations under 3.4 MB/s. Results from µShell_mono indicate that monolithic applications perform almost identically to the baseline (within ±1.4%), confirming that the memory gateway adds negligible overhead. The µShell's throughput overheads occur due to additional latency introduced by the CEU's endpoint validation and dynamic routing between vFPGAs. In summary, µShell achieves near-identical throughput to Coyote v2 with minimal overhead.

6.2 Scheduling Improvements

Next, we evaluate µShell's component-aware scheduling policy with Coyote v2's FIFO policy. We measure five metrics: end-to-end latency, reconfiguration count, average response time, tail (95%) response time, and deadline miss rate, using three component-shared applications: digital signature (SHA256, RSA), signed compression (RLE, RSA), and audio processing (FFT, RLE). These metrics are chosen to capture system throughput, reconfiguration overhead, and quality of service for the two scheduling algorithms.

In each experiment, we periodically deploy 8, 12, or 16 application instances every 20ms, which is shorter than the execution time, forcing applications to induce queuing in the scheduler. Each instance is randomly assigned one of three priority levels. Both policies enable component sharing, i.e., if an idle vFPGA has a matching component for a new application, µShell reuses it to avoid reconfiguration.

While Coyote advertises PR support, we encountered technical issues that prevent its activation across multiple vFPGAs, stemming from the complexity of interrupt handling. Instead, we emulate the PR behavior by provisioning all application components in advance. Specifically, we emulate four PR-enabled vFPGAs to ensure that at least two applications can run simultaneously. To do this, we configure two copies of each application component (SHA256, RSA, RLE, and FFT) on the FPGA and use only four of them as 'active' components, while the others remain 'idle'. When an idle component is selected to deploy a new instance, we inject the pre-measured reconfiguration delay, preserving PR's functional behavior.

End-to-end latency. This metric captures the total time to complete all applications, from the arrival of the first instance to the completion of the last. As shown in Figure 12 (a), µShell reduces end-to-end latency by approximately 24-35% across all scenarios, consistently outperforming Coyote v2's FIFO policy. These results highlight µShell's component-aware scheduling's ability to improve overall efficiency.

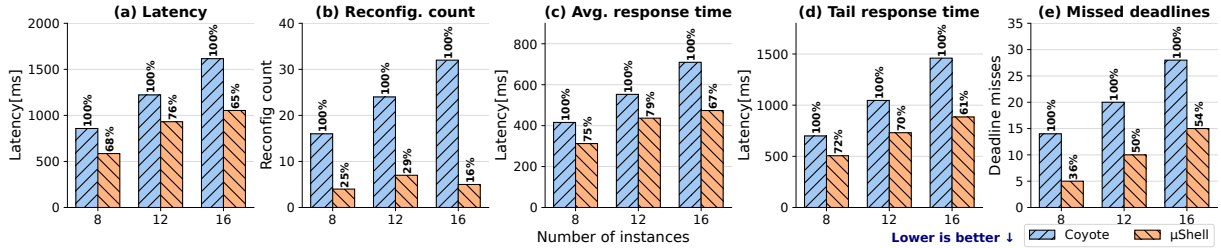


Figure 12: μShell's component-aware scheduling comparing against Coyote's FIFO scheduling algorithm using five metrics: (a) end-to-end latency; (b) reconfiguration count; (c) average response time; (d) tail response time (95%); (e) deadline misses.

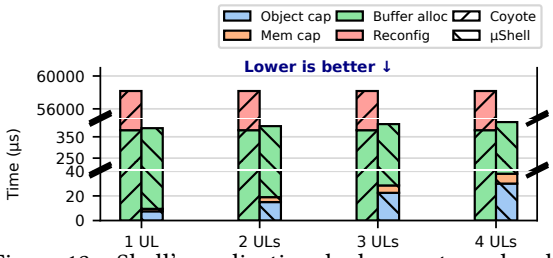


Figure 13: μShell's application deployment overheads.

Applications	SLoC		CC	
	Coyote	μShell	Coyote	μShell
Audio Processing	254	270 (+6.3)	26	14 (-46.2)
Digital Signature	247	251 (+1.6)	24	14 (-41.7)
Secure Storage	302	296 (-2.0)	43	21 (-51.2)
Signed Compression	328	323 (-1.5)	44	32 (-27.3)
Speech Recognition	244	301 (+23.4)	24	18 (-25.0)

Table 5: μShell's programmability, showing Source Lines of Code (SLoC) and Cyclomatic Complexity (CC) of CPU application code. Parentheses show changes from Coyote v2.

Reconfiguration count. To explain μShell's performance gains, we analyze the number of partial reconfiguration operations for both schedulers. Figure 12(b) shows μShell keeps reconfiguration counts stable (5-7), even as application instances grow. In contrast, Coyote v2 requires approximately 3-5× more reconfigurations for the same workload. To conclude, μShell's component-aware scheduling effectively eliminates unnecessary reconfigurations and improves the throughput, especially under heavy queueing.

Average response time. We measure response time, the interval from application instance deployment to completion. Figure 12(c) shows its geometric mean across all instances. As shown here, μShell reduces the response time by 21–33% compared to Coyote, led by fewer vFPGA reconfigurations.

Tail response time. We measure the 95th percentile tail response time to assess the impact on the slowest applications. In Figure 12(d), μShell improves tail response time by 28–39% over Coyote. This gain stems from priority promotion in μShell's component-aware scheduling, which favors long-waiting instances. Thus, μShell improves response time without sacrificing tail latencies.

Deadline miss rate. We measure deadline misses that occur when a component's response time exceeds its deadline under both scheduling policies. Deadlines are assigned per component rather than per application, so a two-component application incurs two deadline misses if both components exceed their individual deadlines due to queuing delays. To ensure that higher-priority applications have shorter deadlines compared to lower-priority applications that can tolerate longer wait times, deadlines are assigned using the formula: $(100\% + x) * (1 + priority)$ of its average response time, where x is a random number between 40 and 80. Figure 12(e) shows the results, highlighting that μShell reduces deadline misses by 46–64% against Coyote v2. The number of missed deadlines increases with more applications because some misses are inevitable when applications arrive at a rate exceeding the vFPGA's execution capability. In summary, μShell's component sharing and priority promotion are effective to mitigate deadline misses.

6.3 Application Deployment Overheads

We next compare the overheads for deploying multi-component applications on μShell and Coyote v2. For μShell, we measure the latency of host-initiated capability updates and memory buffer allocation, assuming an ideal case where all application components are already deployed on the vFPGAs, thus avoiding PR. For Coyote, all user logic (UL) is packaged into a single bitstream, and a monolithic accelerator containing all components is deployed within a single vFPGA, requiring PR and memory buffer allocation. We evaluate four accelerator configurations, each consisting of 1, 2, 3, and 4 ULs, respectively, along with a pair of input/output buffers.

Results. Figure 13 compares deployment overhead between μShell and Coyote v2. For Coyote, the PR time (Reconfig) dominates the overhead, at around 58 ms. Since Coyote v2 does not support component sharing, application deployment always involves a single PR regardless of the number of modules. Consequently, the overhead remains constant for applications with 1–4 ULs. In contrast, μShell avoids PR in this scenario, where all required components already exist on vFPGAs; PR is performed only for missing components. To reuse the shared components, μShell only needs to update memory buffer allocation and capability configuration, with the latter scaling with the number of components. These capability updates (2-3 us per object/memory update) are orders of magnitude faster than PR. Consequently, μShell achieves significantly lower deployment overhead when application components are shareable.

Entity	LUT (%)		Register (%)		BRAM (%)		URAM (%)	
U280 FPGA	1,303,680	(100)	2,607,360	(100)	2,016	(100)	960	(100)
μShell	214,579	(16.5)	330,407	(12.7)	334	(16.6)	0	(0.0)
Coyote	184,074	(14.1)	293,812	(11.3)	280	(13.9)	0	(0.0)
PCIe DMA	70,364	(5.4)	64,758	(2.5)	96	(4.8)	0	(0.0)
Inter 3	10,666	(0.8)	6,628	(0.3)	0	(0.0)	0	(0.0)
Inter 4	13,518	(1.0)	8,986	(0.3)	0	(0.0)	0	(0.0)
Inter 6	35,982	(2.8)	14,273	(0.5)	0	(0.0)	0	(0.0)
Inter 8	65,546	(5.0)	20,001	(0.8)	0	(0.0)	0	(0.0)
MMU	13,004	(1.0)	22,258	(0.8)	48	(2.4)	0	(0.0)
CEU	2,660	(0.2)	4,033	(0.2)	7.5	(0.4)	0	(0.0)
SHA256	1,413	(0.1)	2,982	(0.1)	0	(0.0)	0	(0.0)
RSA	8,365	(0.6)	10,450	(0.4)	0	(0.0)	0	(0.0)
Comp.	1,563	(0.1)	3,239	(0.1)	0	(0.0)	0	(0.0)
AES-CTR	9,355	(0.6)	16,720	(0.7)	129.5	(6.2)	208	(21.7)
FFT	2,082	(0.1)	4,567	(0.1)	4	(0.2)	0	(0.0)
SVM	18,961	(1.5)	16,576	(0.6)	3	(0.1)	0	(0.0)
QUANT	654	(0.0)	917	(0.0)	0	(0.0)	0	(0.0)

Table 6: μShell's resource usage on U280 FPGA. The components in bold are unique to μShell.

6.4 Programmability

Next, we assess the programmability benefits of μShell for host (CPU) code managing FPGA user logic. We compare equivalent host applications using Coyote v2's custom API for monolithic designs versus μShell's programming API. To quantify development effort and complexity, we measure source lines of code (SLOC) and cyclomatic complexity (CC) with *scc* [110].

Results. Table 5 compares SLOC and CC metrics for both approaches. μShell consistently reduces CC by 25.0-51.2% while SLOC changes modestly (-2.0% to +23.4%). The slight SLOC changes result from μShell's API, which adds calls for dynamic accelerator composition, unlike Coyote's static model that manages a single vFPGA with a monolithic accelerator. The CC reduction results from eliminating complex control flow, improving maintainability. In summary, μShell's modular, declarative API enables dynamic composition of FPGA accelerators at a small cost in code size.

6.5 Resource Overheads

Lastly, we measure the resource overhead of μShell using the Vivado IDE, recording the required LUTs, registers, BRAM, and URAM. We configure μShell and Coyote v2 with three vFPGAs and report both total and per-component resource usage. Since the number of vFPGAs affects resource usage, we also measure the resource usage of the data interconnect with 3, 4, 6, and 8 vFPGAs, respectively (Inter 3, 4, 6, 8 in Table 6).

Results. Table 6 shows the resource usage of two shells (μShell and Coyote v2) along with their key components and each accelerator logic for applications. The MMU and CEU rows show resource usage of a single module for one vFPGA, which scales linearly with the number of vFPGAs. We highlight that μShell components (the CEUs and interconnect) consume only 1.4% of LUTs and 0.9% of registers for three vFPGAs.

For scalability, each additional vFPGA requires dedicated CEU and MMU instances, as well as expanded interconnect. While the CEU and MMU's usage scales linearly with the number of vFPGAs, the interconnect's usage scales approximately quadratically, since μShell requires a mesh connection to all vFPGAs so that any vFPGA can establish IPC with any other. Nevertheless, even with eight vFPGAs, the CEUs and interconnect account for 6.6% of the total FPGA resources.

7 Related Work

We first compare μShell with the state-of-the-art shells and FPGA frameworks (Table 7), highlighting our unique contributions across four key dimensions: (1) modularity, (2) composability, (3) communication, and (4) isolation.

FPGA platforms and comparison with μShell. For (1) modularity, ReconOS [83] and Hthreads [98] treat hardware logic as OS threads, yet they target monolithic applications. FOS [119], DrawerPipe [79], Harmonia [80], and Coyote [71, 104] apply modularity to their shells with reusable building blocks. However, application designs remain monolithic, limiting fine-grained reuse and independent updates of hardware modules for FPGA applications.

For (2) composability, most systems support PR to enable spatial or temporal multiplexing, but they do not allow user logic to be flexibly shared or composed across applications. DrawerPipe [79] enables component sharing but is constrained by its static pipeline architecture, which requires the entire shell to be reconfigured. As a result, general-purpose, dynamic composition remains unsupported.

For (3) communication, cross-component communication is constrained by static interconnects (e.g., Coyote's FIFOs and DrawerPipe's linked-list connections) or indirect off-chip paths (e.g., Nyx's [91] memory FIFOs). Rosebud [65] provides dynamically configurable direct I/Os but is specialized for network-processing pipelines.

Lastly, for (4) isolation, existing systems provide either coarse-grained per-application isolation [79], memory isolation via MMUs [113], or both [64,71,91,104,119,135]. However, there are no systems that enforce fine-grained control over inter-component communication distributed across vFPGAs.

In summary, μShell is the first framework to provide the full system properties required for modular FPGA application deployment. By integrating cross-vFPGA IPC control, capability-based isolation, and component sharing support, it enables secure, controlled interaction between independently deployed and shared components on FPGAs.

FPGA OSes. FPGA OSes adopt OS abstractions for custom logic, spanning spatial/temporal sharing, memory management, and scheduling [40, 62, 76, 83, 84, 98, 100, 113, 135]. μShell is the first work to adapt microkernel principles to enable modular, composable FPGA application deployment.

Accelerator chaining and data movement. Prior work has optimized data movement between heterogeneous accelerators [61, 92, 94, 99, 101, 136, 137], often through costly host-mediated point-to-point transfers. Recent efforts enable host-free, composable chains of single-purpose accelerators

Platform	(1) Modularity		(2) Composability		(3) Communication		(4) Isolation		
	Shell	Application	PR	Comp. Sharing	Inter-vFPGA	Direct I/O	App	Component	Memory
ReconOS [5]	Monolithic	Monolithic	✓	✗	✗	✗	✓	✗	✓
BORPH [113]	Monolithic	Monolithic	✓	✗	✗	✗	✗	✗	✓
Feniks [135]	Monolithic	Monolithic	✓	✗	✗	✗	✓	✗	✓
FOS [119]	Modular	Monolithic	✗	✗	✗	✗	✓	✗	✓
AmorphOS [64]	Monolithic	Monolithic	✓	✗	✗	✗	✓	✗	✓
DrawerPipe [79]	Modular	Monolithic	✗	✓	Static	✓	✓	✗	✗
Rosebud [65]	Modular	Modular	✓	✗	Dynamic	✓	✗	✗	✗
Harmonia [80]	Modular	Monolithic	✓	✗	✗	✗	✗	✗	✗
Nyx [91]	Modular	Modular	✓	✗	Dynamic	✗	✓	✗	✓
Coyote [71, 104]	Modular	Monolithic	✓	✗	Static	✓	✓	✗	✓
μShell	Modular	Modular	✓	✓	Dynamic	✓	✓	✓	✓

Table 7: Comprehensive analysis of PR–based and component-based FPGA shells.

(e.g., ASIC) via data restructuring and interconnect-based isolation [15, 16, 43, 121]. μShell's CEUs provide functionality similar to M³'s DTUs [16], but differ fundamentally: CEUs are designed for FPGA streaming architectures, whereas M³ targets NoC-based many-core processors.

FPGA virtualization and isolation. FPGA virtualization supports spatial sharing [21, 24, 95], temporal sharing [73, 106, 122], memory virtualization [30, 47, 83, 124], and context switching [60, 77]. Cloud virtualization systems such as Optimus [84], AvA [132], and others [85, 118] extend guest OSes or hypervisors to isolate FPGA resources across VMs. However, they expose vFPGAs as individual resources, limiting accelerator composition. In contrast, μShell presents all vFPGAs as a unified workspace while preserving multi-tenant isolation.

FPGA programming model. Dataflow programming models describe inter-/intra-accelerator communications [28, 29, 70, 116, 125]. Compiler-based approaches such as HeteroFlow [125], TAPA [29], and SODA [28] are actively studied, while they build static, monolithic accelerators. Runtime libraries support kernel chaining via OpenCL pipes [46] and XRT native API [66], but are also limited to monolithic designs. In contrast, the μShell API enables dynamically chaining accelerators across vFPGAs without compromising isolation.

FPGA task scheduling. FPGA task scheduling balances resource utilization, performance, and isolation [33, 36, 38, 88, 91]. Nimblock [88] provides fine-grained sharing through overlay virtualization, while Nyx [91] virtualizes dataflow execution using virtual FIFOs. However, existing schedulers lack awareness of cross-application component reuse, leading to frequent reconfiguration overhead. μShell's component-aware scheduling prioritizes applications based on user logic overlap, maximizing reuse while minimizing reconfiguration costs.

8 Conclusion and Discussion

We present μShell, a hardware-OS co-design for modular accelerator deployment on FPGAs, delivering three key contributions: (1) a shell architecture for building secure, direct communications between distinct vFPGAs, (2) an OS providing microkernel features to manage FPGA tasks, and (3) a high-level dataflow API for dynamic accelerator composition. We implement and evaluate the μShell prototype on Alveo U280 FPGA, highlighting μShell's performance, flexibility, scheduling effectiveness, programmability, and resource efficiency.

Interchangeability among PR regions. Consistent with modern FPGA shells (e.g., Coyote [71, 103], AmorphOS [64]), μShell generates a separate partial bitstream for each logic-vFPGA pair. For instance, deploying two distinct logic kernels across two vFPGAs requires four unique bitstreams. This limitation arises from current toolchains, which require each kernel to be synthesized for every target vFPGA. Recent studies [96, 133] mitigate these constraints by decoupling compilation from physical placement and flexibly sizing PR regions. μShell can leverage these advances to further enhance deployment flexibility and interchangeability.

vFPGA resource usage. μShell supports variable-size vFPGA partitions through PR. However, due to limitations in current PR toolchains, vFPGA sizes must be manually defined, and bitstreams must be generated for predefined vFPGAs. In the current prototype, we adopt uniform vFPGA partitioning to simplify bitstream management and deployment.

Multi-FPGA deployments. Exploring multi-FPGA deployments (e.g., Catapult [102]) is beyond our current scope and planned as future work. This paper demonstrates the benefits of intra-FPGA chaining for realistic, kernel-level workloads, showing how μShell components efficiently share and compose resources within a single FPGA. For these use cases, efficient chaining of user logic is crucial, as it improves resource utilization and reduces deployment costs, making cloud FPGAs more accessible and economically viable.

Artifact. We release μShell as an open-source project [3].

Acknowledgments. This work was partially supported by an ERC Starting Grant (ID: 101077577) and the Chips Joint Undertaking (JU), European Union (EU) HORIZON-JU-IA, under grant agreement No. 101140087 (SMARTY), the Intel Trustworthy Data Center of the Future (TDCoF), Google Research Grants, and DFG Priority Program SPP-2378 "Resilient Worlds". This work was also supported by JSPS KAKENHI Grant Numbers JP19K24360 and JP20K19776.

References

[1] Amd: Icap interface. https://docs.xilinx.com/r/en-US/pg036_sem/ICAP-Interface. Last accessed: June 23, 2026.

[2] Xrt and vitis™platform overview. https://xilinx.github.io/XRT/master/html/platforms.html. Last accessed: June 23, 2026.

[3] µShell code. https://github.com/TUM-DSE/microShell.

[4] Michael J. Accetta, Robert V. Baron, William J. Bolosky, David B. Golub, Richard F. Rashid, Avadis Tevanian, and Michael Young. Mach: A new kernel foundation for unix development. In *USENIX Summer*, 1986.

[5] Andreas Agne, Markus Happe, Ariane Keller, Enno Lübbers, Bernhard Plattner, Marco Platzner, and Christian Plessl. Reconos: An operating system approach for reconfigurable computing. *IEEE Micro*, 34(1):60–71, 2013.

[6] Morteza Babaee Altman, Wenbin Wan, Amineh Sadat Hosseini, Saber Arabi Nowdeh, and Masoumeh Alizadeh. Machine learning algorithms for fpga implementation in biomedical engineering applications: A review. *Heliyon*, 10(4), Feb 2024.

[7] Amazon. Aws shell interface specification. https://awsdocs-fpga-f2.readthedocs-hosted.com/latest/hdk/docs/AWS-Shell-Interface-Specification.html. Last accessed: June 23, 2026.

[8] Amazon. Amazon ec2 f2 instances. https://aws.amazon.com/ec2/instance-types/f2, 2025. Last accessed: June 23, 2026.

[9] Amazon. Amazon ec2 fpgaimage. https://docs.aws.amazon.com/AWSEC2/latest/APIReference/API_FpgaImage.html, 2025. Last accessed: June 23, 2026.

[10] AMD. Axi4-stream switch – axi4-stream infrastructure ip suite logicore ip product guide (pg085). https://docs.amd.com/r/en-US/pg085-axi4stream-infrastructure/AXI4-Stream-Switch, 2025. Last accessed: June 23, 2026.

[11] AMD. Vitis high-level synthesis user guide (ug1399). https://docs.amd.com/r/en-US/ug1399-vitis-hls/pragma-HLS-reset, 2025. Last accessed: June 23, 2026.

[12] AMD. Vivado design suite user guide: Dynamic function exchange (ug909). https://docs.amd.com/r/en-US/ug909-vivado-partial-reconfiguration/, 2025. Last accessed: June 23, 2026.

[13] Arm. Axi4 and axi4-lite interfaces. https://developer.arm.com/documentation/dui0534/b/Parameter-Descriptions/Interface/AXI4-and-AXI4-Lite-interfaces, 2025. Last accessed: June 23, 2026.

[14] Arm. Axi4 stream interface. https://developer.arm.com/documentation/102482/0000/DMAC-interfaces/AXI4-stream-interface, 2025. Last accessed: June 23, 2026.

[15] Nils Asmussen, Michael Roitzsch, and Hermann Härtig. M³x: Autonomous accelerators via Context-Enabled Fast-Path communication. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 617–632, Renton, WA, July 2019. USENIX Association.

[16] Nils Asmussen, Marcus Völp, Benedikt Nöthen, Hermann Härtig, and Gerhard Fettweis. M3: A hardware/operating-system co-design to tame heterogeneous manycores. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '16, page 189–203, New York, NY, USA, 2016. Association for Computing Machinery.

[17] Sameh Attia and Vaughn Betz. Statereveal: Enabling checkpointing of fpga designs with buried state. In *2020 International Conference on Field-Programmable Technology (ICFPT)*, pages 206–214, 2020.

[18] Bittware. Ia-420f pcie accelerator with intel agilex fpga. https://www.bittware.com/products/ia-420f/, 2025. Last accessed: June 23, 2026.

[19] Bittware. Ia-840f pcie fpga card featuring intel agilex fpga. https://www.bittware.com/products/ia-840f/, 2025. Last accessed: June 23, 2026.

[20] Nathaniel Bleier, Muhammad Husnain Mubarik, Srijan Chakraborty, Shreyas Kishore, and Rakesh Kumar. Rethinking programmable earable processors. In *Proceedings of the 49th Annual International Symposium on Computer Architecture*, ISCA '22, page 454–467, New York, NY, USA, 2022. Association for Computing Machinery.

[21] Stuart Byma, J. Gregory Steffan, Hadi Bannazadeh, Alberto Leon Garcia, and Paul Chow. Fpgas in the cloud: Booting virtualized hardware accelerators with openstack. In *Proceedings of the 2014 IEEE 22Nd International Symposium on Field-Programmable Custom Computing Machines (FCCM '14)*, pages 109–116. IEEE Computer Society, 2014.

[22] Jared Casper and Kunle Olukotun. Hardware acceleration of database operations. In *Proceedings of the 2014 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, FPGA '14, page 151–160, New York, NY, USA, 2014. Association for Computing Machinery.

[23] Stephen Cass. Taking ai to the edge: Google's tpu now comes in a maker-friendly package. *IEEE Spectrum*, 56(5):16–17, 2019.

[24] Fei Chen, Yi Shan, Yu Zhang, Yu Wang, Hubertus Franke, Xiaotao Chang, and Kun Wang. Enabling fpgas in the cloud. In *Proceedings of the 11th ACM Conference on Computing Frontiers*, CF '14, New York, NY, USA, 2014. Association for Computing Machinery.

[25] Haibo Chen, Xie Miao, Ning Jia, Nan Wang, Yu Li, Nian Liu, Yutao Liu, Fei Wang, Qiang Huang, Kun Li, Hongyang Yang, Hui Wang, Jie Yin, Yu Peng, and Fengwei Xu. Microkernel goes general: Performance and compatibility in the HongMeng production microkernel. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, pages 465–485, Santa Clara, CA, July 2024. USENIX Association.

[26] Jianyu Chen, Maurice Daverveldt, and Zaid Al-Ars. Fpga acceleration of zstd compression algorithm. In *2021 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 188–191, 2021.

[27] Jiyang Chen, Harshavardhan Unnibhavi, Atsushi Koshiba, and Pramod Bhatotia. vFPIO: A virtual I/O abstraction for FPGA-accelerated I/O devices. In *2024 USENIX Annual Technical Conference (USENIX ATC 24)*, pages 1167–1184, Santa Clara, CA, July 2024. USENIX Association.

[28] Yuze Chi, Jason Cong, Peng Wei, and Peipei Zhou. Soda: Stencil with optimized dataflow architecture. In *2018 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 1–8, 2018.

[29] Yuze Chi, Licheng Guo, Young-kyu Choi, Jie Wang, and Jason Cong. Extending high-level synthesis for task-parallel programs. In *The 2021 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, FPGA '21, page 225, New York, NY, USA, 2021. Association for Computing Machinery.

[30] Eric S. Chung, James C. Hoe, and Ken Mai. Coram: An in-fabric memory architecture for fpga-based computing. In *Proceedings of the 19th ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA '11)*, pages 97–106. ACM, 2011.

[31] Alibaba Cloud. Alibaba cloud fpga instances. https://www.alibabacloud.com/help/en/doc-detail/108504.html, 2025. Last accessed: June 23, 2026.

[32] Google Cloud. Gpu platforms. https://cloud.google.com/compute/docs/gpus. Last accessed: June 23, 2026.

[33] Roberto Cordone, Francesco Redaelli, Massimo Antonio Redaelli, Marco Domenico Santambrogio, and Donatella Sciuto. Partitioning and scheduling of task graphs on partially dynamically reconfigurable fpgas. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 28(5):662–675, 2009.

[34] NVIDIA Corporation. Nvidia multi-instance gpu. https://www.nvidia.com/en-us/technologies/multi-instance-gpu/. Last accessed: June 23, 2026.

[35] An operating system for fpgas. https://github.com/fpgasystems/Coyote. Last accessed: June 23, 2026.

[36] Enrico A Deiana, Marco Rabozzi, Riccardo Cattaneo, and Marco D Santambrogio. A multiobjective reconfiguration-aware scheduler for fpga-based heterogeneous architectures. In *2015 International Conference on ReConFigurable Computing and FPGAs (ReConFig)*, pages 1–6. IEEE, 2015.

[37] Jack B Dennis and Earl C Van Horn. Programming semantics for multiprogrammed computations. *Communications of the ACM*, 9(3):143–155, 1966.

[38] Ashutosh Dhar, Edward Richter, Mang Yu, Wei Zuo, Xiaohao Wang, Nam Sung Kim, and Deming Chen. Dml: Dynamic partial reconfiguration with scalable task scheduling for multi-applications on fpgas. *IEEE Transactions on Computers*, 71(10):2577–2591, 2022.

[39] D. R. Engler, M. F. Kaashoek, and J. O'Toole. Exokernel: an operating system architecture for application-level resource management. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, SOSP '95, page 251–266, New York, NY, USA, 1995. Association for Computing Machinery.

[40] Kermin Fleming, Hsin-Jung Yang, Michael Adler, and Joel Emer. The leap fpga operating system. In *2014 24th International Conference on Field Programmable Logic and Applications (FPL)*, pages 1–8, 2014.

[41] Intel FPGA. Intel fpga add-on for oneapi base toolkit. https://www.intel.com/content/www/us/en/developer/tools/oneapi/fpga.html, 2025. Last accessed: June 23, 2026.

[42] Dimitra Giantsidi, Julian Pritzi, Felix Gust, Antonios Katsarakis, Atsushi Koshiba, and Pramod Bhatotia. Tnic: A trusted nic architecture: A hardware-network substrate for building high-performance trustworthy distributed systems. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, pages 1282–1301, 2025.

[43] Abraham Gonzalez, Aasheesh Kolli, Samira Khan, Sihang Liu, Vidushi Dadu, Sagar Karandikar, Jichuan Chang, Krste Asanovic, and Parthasarathy Ranganathan. Profiling hyperscale big data processing. In *Proceedings of the 50th Annual International Symposium on Computer Architecture*, ISCA '23, New York, NY, USA, 2023. Association for Computing Machinery.

[44] Google. Bigquery | ai data platform. https://cloud.google.com/bigquery?hl=en, 2025. Last accessed: June 23, 2026.

[45] Google. Bigtable: Fast, flexible nosql. https://cloud.google.com/bigtable?hl=en, 2025. Last accessed: June 23, 2026.

[46] Khronos Group. The opencl specification. https://registry.khronos.org/OpenCL/specs/3.0-unified/html/OpenCL_API.html, 2025. Last accessed: June 23, 2026.

[47] Felix Gust, Shu Anzai, Charalampos Mainas, Atsushi Koshiba, and Pramod Bhatotia. Proteus: Heterogeneous fpga virtualization. In *Proceedings of the 21st European Conference on Computer Systems*, pages 328–349, 2026.

[48] Mahmoud Habboush, Aiman H. El-Maleh, Muhammad E.S. Elrabaa, and Saleh AlSaleh. De-zfp: An fpga implementation of a modified zfp compression/decompression algorithm. *Microprocessors and Microsystems*, 90:104453, 2022.

[49] Matthias Hille, Nils Asmussen, Pramod Bhatotia, and Hermann Härtig. SemperOS: A distributed capability system. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 709–722, Renton, WA, July 2019. USENIX Association.

[50] Matthias Hille, Nils Asmussen, Hermann Härtig, and Pramod Bhatotia. A heterogeneous microkernel os for rack-scale systems. In *Proceedings of the 11th ACM SIGOPS Asia-Pacific Workshop on Systems*, APSys '20, page 50–58, New York, NY, USA, 2020. Association for Computing Machinery.

[51] Hanaa M Hussain, Khaled Benkrid, and Huseyin Seker. Dynamic partial reconfiguration implementation of the svm/knn multi-classifier on fpga for bioinformatics application. In *2015 37th Annual International Conference of the IEEE Engineering in Medicine and Biology Society (EMBC)*, pages 7667–7670. IEEE, 2015.

[52] Siam U. Hussain and Farinaz Koushanfar. Fase: Fpga acceleration of secure function evaluation. In *2019 IEEE 27th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 280–288, 2019.

[53] Muhammad Huzaifa, Rishi Desai, Samuel Grayson, Xutao Jiang, Ying Jing, Jae Lee, Fang Lu, Yihan Pang, Joseph Ravichandran, Finn Sinclair, Boyuan Tian, Hengzhi Yuan, Jeffrey Zhang, and Sarita V. Adve. Illixr: Enabling end-to-end extended reality research. In *2021 IEEE International Symposium on Workload Characterization (IISWC)*, pages 24–38, 2021.

[54] IBM. Mini-era: Simplified version of the main era workload. https://github.com/IBM/mini-era, 2025. Last accessed: June 23, 2026.

[55] Intel. Open fpga stack overview. https://ofs.github.io/ofs-2024.3-1/, 2025. Last accessed: June 23, 2026.

[56] Intel. Quartus® prime pro edition user guide: Partial reconfiguration. https://www.intel.com/content/www/us/en/docs/programmable/683834/25-1/faq.html, 2025. Last accessed: June 23, 2026.

[57] Hasan Irmak, Daniel Ziener, and Nikolaos Alachiotis. Increasing flexibility of fpga-based cnn accelerators with dynamic partial reconfiguration. In *2021 31st International Conference on Field-Programmable Logic and Applications (FPL)*, pages 306–311. IEEE, 2021.

[58] Chuanmin Jia, Xinyu Hang, Shanshe Wang, Yaqiang Wu, Siwei Ma, and Wen Gao. Fpx-nic: An fpga-accelerated 4k ultra-high-definition neural video coding system. *IEEE Transactions on Circuits and Systems for Video Technology*, 32(9):6385–6399, 2022.

[59] M. Johnson, P. Ishwar, V. Prabhakaran, D. Schonberg, and K. Ramchandran. On compressing encrypted data. *IEEE Transactions on Signal Processing*, 52(10):2992–3006, 2004.

[60] Heiko Kalte and Mario Porrmann. Context saving and restoring for multitasking in reconfigurable systems. pages 223–228, 2005.

[61] Sagar Karandikar, Chris Leary, Chris Kennelly, Jerry Zhao, Dinesh Parimi, Borivoje Nikolic, Krste Asanovic, and Parthasarathy Ranganathan. A hardware accelerator for protocol buffers. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on*

Microarchitecture, MICRO '21, page 462–478, New York, NY, USA, 2021. Association for Computing Machinery.

[62] John H. Kelm and Steven S. Lumetta. Hybridos: runtime support for reconfigurable accelerators. In *Proceedings of the 16th International ACM/SIGDA Symposium on Field Programmable Gate Arrays*, FPGA '08, page 212–221, New York, NY, USA, 2008. Association for Computing Machinery.

[63] Alireza Khataei, Gaurav Singh, and Kia Bazargan. Approximate hybrid binary-unary computing with applications in bert language model and image processing. In *Proceedings of the 2023 ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, FPGA '23, page 165–175, New York, NY, USA, 2023. Association for Computing Machinery.

[64] Ahmed Khawaja, Joshua Landgraf, Rohith Prakash, Michael Wei, Eric Schkufza, and Christopher J. Rossbach. Sharing, protection, and compatibility for reconfigurable fabric with AmorphOS. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 107–127, Carlsbad, CA, October 2018. USENIX Association.

[65] Moein Khazraee, Alex Forencich, George C Papen, Alex C Snoeren, and Aaron Schulman. Rosebud: Making fpga-accelerated middlebox development more pleasant. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, pages 586–605, 2023.

[66] Khronos Group. Pcie peer-to-peer (p2p) – xrt master documentation. https://xilinx.github.io/XRT/master/html/p2p.html. Accessed: June 23, 2026.

[67] Joon Kyung Kim, Byung Hoon Ahn, Sean Kinzer, Soroush Ghodrati, Rohan Mahapatra, Brahmendra Yatham, Shu-Ting Wang, Dohee Kim, Parisa Sarikhani, Babak Mahmoudi, Divya Mahajan, Jongse Park, and Hadi Esmaeilzadeh. Yin-yang: Programming abstractions for cross-domain multi-acceleration. *IEEE Micro*, 42(5):89–98, 2022.

[68] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. sel4: formal verification of an os kernel. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, SOSP '09, page 207–220, New York, NY, USA, 2009. Association for Computing Machinery.

[69] Oliver Knodel, Paul R. Genssler, and Rainer G. Spallek. Migration of long-running tasks between reconfigurable resources using virtualization. *SIGARCH Comput. Archit. News*, 44(4):56–61, January 2017.

[70] David Koeplinger, Matthew Feldman, Raghu Prabhakar, Yaqi Zhang, Stefan Hadjis, Ruben Fiszel, Tian Zhao, Luigi Nardi, Ardavan Pedram, Christos Kozyrakis, and Kunle Olukotun. Spatial: A language and compiler for application accelerators. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. ACM, 2018.

[71] Dario Korolija, Timothy Roscoe, and Gustavo Alonso. Do OS abstractions make sense on FPGAs? In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 991–1010. USENIX Association, November 2020.

[72] Atsushi Koshiba, Felix Gust, Julian Pritzi, Anjo Vahldiek-Oberwagner, Nuno Santos, and Pramod Bhatotia. Trusted heterogeneous disaggregated architectures. In *Proceedings of the 14th ACM SIGOPS Asia-Pacific Workshop on Systems*, pages 72–79, 2023.

[73] Atsushi Koshiba, Charalampos Mainas, and Pramod Bhatotia. Funky: Cloud-native fpga virtualization and orchestration. In *Proceedings of the 2025 ACM Symposium on Cloud Computing*, pages 209–224, 2025.

[74] Adam Lackorzynski and Alexander Warg. Taming subsystems: capabilities as universal resource access control in l4. In *Proceedings of the Second Workshop on Isolation and Integration in Embedded Systems*, IIES '09, page 25–30, New York, NY, USA, 2009. Association for Computing Machinery.

[75] Yi-Hsiang Lai, Ecenur Ustun, Shaojie Xiang, Zhenman Fang, Hongbo Rong, and Zhiru Zhang. Programming and synthesis for software-defined fpga acceleration: Status and future prospects. *ACM Trans. Reconfigurable Technol. Syst.*, 14(4), September 2021.

[76] Joshua Landgraf, Matthew Giordano, Esther Yoon, and Christopher J. Rossbach. Reconfigurable virtual memory for fpga-driven i/o. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, ASPLOS 2023, page 556–571, New York, NY, USA, 2023. Association for Computing Machinery.

[77] Trong-Yen Lee, Che-Cheng Hu, Li-Wen Lai, and Chia-Chun Tsai. Hardware context-switch methodology for dynamically partially reconfigurable systems. *J. Inf. Sci. Eng.*, 26:1289–1305, 2010.

[78] Henry M Levy. *Capability-based computer systems*. Digital Press, 2014.

[79] Junnan Li, Zhigang Sun, Jinli Yan, Xiangrui Yang, Yue Jiang, and Wei Quan. Drawerpipe: A reconfigurable pipeline for network processing on fpga-based smartnic. *Electronics*, 9(1):59, 2019.

[80] Luyang Li, Heng Pan, Xinchen Wan, Kai Lv, Zilong Wang, Qian Zhao, Feng Ning, Qingsong Ning, Shideng Zhang, Zhenyu Li, et al. Harmonia: A unified framework for heterogeneous fpga acceleration in the cloud. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, pages 498–514, 2025.

[81] Jochen Liedtke. On micro-kernel construction. *ACM SIGOPS Operating Systems Review*, 29(5):237–250, 1995.

[82] Alec Lu and Zhenman Fang. Sql2fpga: Automatic acceleration of sql query processing on modern cpu-fpga platforms. In *2023 IEEE 31st Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 184–194, 2023.

[83] Enno Lübbers and Marco Platzner. Reconos: Multithreaded programming for reconfigurable computers. *ACM Trans. Embed. Comput. Syst.*, 9(1), October 2009.

[84] Jiacheng Ma, Gefei Zuo, Kevin Loughlin, Xiaohe Cheng, Yanqiang Liu, Abel Mulugeta Eneyew, Zhengwei Qi, and Baris Kasikci. A hypervisor for shared-memory fpga platforms. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '20, page 827–844, New York, NY, USA, 2020. Association for Computing Machinery.

[85] Jiacheng Ma, Gefei Zuo, Kevin Loughlin, Xiaohe Cheng, Yanqiang Liu, Abel Mulugeta Eneyew, Zhengwei Qi, and Baris Kasikci. A hypervisor for shared-memory fpga platforms. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '20, page 827–844, New York, NY, USA, 2020. Association for Computing Machinery.

[86] R. Maestre, F.J. Kurdahi, M. Fernandez, R. Hermida, N. Bagherzadeh, and H. Singh. A framework for reconfigurable computing: task scheduling and context management. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 9(6):858–873, 2001.

[87] Charalampos Mainas, Martin Lambeck, Bruno Scheufler, Laurent Bindschaedler, Atsushi Koshiba, and Pramod Bhatotia. F3: An fpga-accelerated faas framework. In *Proceedings of the 34th International Symposium on High-Performance Parallel and Distributed Computing*, pages 1–16, 2025.

[88] Meghna Mandava, Paul Reckamp, and Deming Chen. Nimblock: Scheduling for fine-grained fpga sharing through virtualization. In *Proceedings of the 50th Annual International Symposium on Computer Architecture*, ISCA '23, New York, NY, USA, 2023. Association for Computing Machinery.

[89] Michael Marty, Marc de Kruijf, Jacob Adriaens, Christopher Alfeld, Sean Bauer, Carlo Contavalli, Michael Dalton, Nandita Dukkipati, William C. Evans, Steve Gribble, Nicholas Kidd, Roman Kononov, Gautam Kumar, Carl Mauer, Emily Musick, Lena Olson, Erik Rubow, Michael Ryan, Kevin Springborn, Paul Turner, Valas Valancius, Xi Wang, and Amin Vahdat. Snap: a microkernel approach to host networking. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP '19, page 399–413, New York, NY, USA, 2019. Association for Computing Machinery.

[90] Microsoft. Np size series - azure virtual machines. https://learn.microsoft.com/en-us/azure/virtual-machines/sizes/fpga-accelerated/np-series, 2025. Last accessed: June 23, 2026.

[91] Panagiotis Miliadis, Dimitris Theodoropoulos, Nectarios Koziris, and Dionisios Pnevmatikatos. Nyx: Virtualizing dataflow execution on shared fpga platforms. In *Proceedings of the 52nd Annual International Symposium on Computer Architecture*, ISCA '25, page 1327–1341, New York, NY, USA, 2025. Association for Computing Machinery.

[92] Derek G. Murray, Jiří Šimša, Ana Klimovic, and Ihor Indyk. tf.data: a machine learning data processing framework. *Proc. VLDB Endow.*, 14(12):2945–2958, jul 2021.

[93] Hau T Ngo, Ryan N Rakvic, Randy P Broussard, and Robert W Ives. An fpga-based design of a modular approach for integral images in a real-time face detection system. In *Mobile Multimedia/Image Processing, Security, and Applications 2009*, volume 7351, pages 83–92. SPIE, 2009.

[94] NVIDIA. Dali, 2024. accessed: 2024-08-01.

[95] Michele Paolino, Sébastien Pinneterre, and Daniel Raho. Fpga virtualization with accelerators overcommitment for network function virtualization. In *2017 International Conference on ReConFigurable Computing and FPGAs (ReConFig)*, pages 1–6. IEEE, 2017.

[96] Dongjoon Park, Yuanlong Xiao, and André DeHon. Fast and flexible fpga development using hierarchical partial reconfiguration. In *2022 International Conference on Field-Programmable Technology (ICFPT)*, pages 1–10. IEEE, 2022.

[97] Aaron Parsons, Donald Backer, Andrew Siemion, Henry Chen, Dan Werthimer, Pierre Droz, Terry Filiba, Jason Manley, Peter McMahon, Arash Parsa, et al. A scalable correlator architecture based on modular fpga hardware, reuseable gateware, and data packetization. *Publications of the Astronomical Society of the Pacific*, 120(873):1207–1221, 2008.

[98] Wesley Peck, Erik Anderson, Jason Agron, Jim Stevens, Fabrice Baijot, and David Andrews. Hthreads: A computational model for reconfigurable devices. In *2006 International Conference on Field Programmable Logic and Applications*, pages 1–4, 2006.

[99] Johan Peltenburg, Jeroen van Straten, Lars Wijtemans, Lars van Leeuwen, Zaid Al-Ars, and Peter Hofstee. Fletcher: A framework to efficiently integrate fpga accelerators with apache arrow. In *2019 29th International Conference on Field Programmable Logic and Applications (FPL)*, pages 270–277, 2019.

[100] Khoa Dang Pham, Kyriakos Paraskevas, Anuj Vaishnav, Andrew Attwood, Malte Vesper, and Dirk Koch. Zucl 2.0: Virtualised memory and communication for zynq ultrascale+ fpgas. In *FSP Workshop 2019; Sixth International Workshop on FPGAs for Software Programmers*, pages 1–9. VDE, 2019.

[101] Arash Pourhabibi, Siddharth Gupta, Hussein Kassir, Mark Sutherland, Zilu Tian, Mario Paulo Drumond, Babak Falsafi, and Christoph Koch. Optimus prime: Accelerating data transformation in servers. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '20, page 1203–1216, New York, NY, USA, 2020. Association for Computing Machinery.

[102] Andrew Putnam, Adrian M. Caulfield, Eric S. Chung, Derek Chiou, Kypros Constantinides, John Demme, Hadi Esmaeilzadeh, Jeremy Fowers, Gopi Prashanth Gopal, Jan Gray, Michael Haselman, Scott Hauck, Stephen Heil, Amir Hormati, Joo-Young Kim, Sitaram Lanka, James Larus, Eric Peterson, Simon Pope, Aaron Smith, Jason Thong, Phillip Yi Xiao, and Doug Burger. A reconfigurable fabric for accelerating large-scale datacenter services. *IEEE Micro*, 35(3):10–22, 2015.

[103] Benjamin Ramhorst, Dario Korolija, Maximilian Jakob Heer, Jonas Dann, Luhao Liu, and Gustavo Alonso. Coyote v2: Raising the level of abstraction for data center fpgas. In *Proceedings of the ACM SIGOPS 31st Symposium on Operating Systems Principles*, SOSP '25, page 639–654, New York, NY, USA, 2025. Association for Computing Machinery.

[104] Benjamin Ramhorst, Dario Korolija, Maximilian Jakob Heer, Jonas Dann, Luhao Liu, and Gustavo Alonso. Coyote v2: Raising the level of abstraction for data center fpgas, 2025.

[105] Swarnalata Bollavarapu Ruchita Sharma. Data security using compression and cryptography techniques. *International Journal of Computer Applications*, 117(14):15–18, May 2015.

[106] Kyle Rupnow, Wenyin Fu, and Katherine Compton. Block, drop or roll(back): Alternative preemption methods for rh multi-tasking. In *FCCM 2009, 17th IEEE Symposium on Field Programmable Custom Computing Machines*, pages 63–70. IEEE, 2009.

[107] Ahmad Sadek, Hassan Mostafa, Amin Nassar, and Yehea Ismail. Towards the implementation of multi-band multi-standard software-defined radio using dynamic partial reconfiguration. *International Journal of Communication Systems*, 30(17):e3342, 2017.

[108] Kartik Sankaran, Minhui Zhu, Xiang Fa Guo, Akkihebbal L. Ananda, Mun Choon Chan, and Li-Shiuan Peh. Using mobile phone barometer for low-power transportation context detection. In *Proceedings of the 12th ACM Conference on Embedded Network Sensor Systems*, SenSys '14, page 191–205, New York, NY, USA, 2014. Association for Computing Machinery.

[109] S.M. Scalera and J.R. Vazquez. The design and implementation of a context switching fpga. In *Proceedings. IEEE Symposium on FPGAs for Custom Computing Machines (Cat. No.98TB100251)*, pages 78–85, 1998.

[110] Sloc cloc and code (scc). https://github.com/boyter/scc. Last accessed: June 23, 2026.

[111] Thapanee Seehapoch and Sartra Wongthanavasu. Speech emotion recognition using support vector machines. In *2013 5th International Conference on Knowledge and Smart Technology (KST)*, pages 86–91, 2013.

[112] Piyush Kumar Shukla, Amer Aljaedi, Piyush Kumar Pareek, Adel R. Alharbi, and Sajjad Shaukat Jamal. Aes based white box cryptography in digital signature verification. *Sensors*, 22(23), 2022.

[113] Hayden Kwok-Hay So and Robert Brodersen. A unified hardware/software runtime environment for fpga-based reconfigurable computers using borph. *ACM Transactions on Embedded Computing Systems (TECS)*, 7(2):1–28, 2008.

[114] Naveen Suda, Vikas Chandra, Ganesh Dasika, Abinash Mohanty, Yufei Ma, Sarma Vrudhula, Jae-sun Seo, and Yu Cao. Throughput-optimized opencl-based fpga accelerator for large-scale convolutional neural

networks. In *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, FPGA '16, page 16–25, New York, NY, USA, 2016. Association for Computing Machinery.

[115] Vignesh Suresh, Bakshree Mishra, Ying Jing, Zeran Zhu, Naiyin Jin, Charles Block, Paolo Mantovani, Davide Giri, Joseph Zuckerman, Luca P. Carloni, and Sarita V. Adve. Mozart: Taming taxes and composing accelerators with shared-memory. In *Proceedings of the 2024 International Conference on Parallel Architectures and Compilation Techniques*, PACT '24, page 183–200, New York, NY, USA, 2024. Association for Computing Machinery.

[116] J. Sérot and F. Berry. High-level dataflow programming for reconfigurable computing. In *2014 International Symposium on Computer Architecture and High Performance Computing Workshop*, pages 72–77, 2014.

[117] Cheng Tan, Manupa Karunaratne, Tulika Mitra, and Li-Shiuan Peh. Stitch: Fusible heterogeneous accelerators enmeshed with many-core architecture for wearables. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, pages 575–587, 2018.

[118] Naif Tarafdar, Thomas Lin, Eric Fukuda, Hadi Bannazadeh, Alberto Leon-Garcia, and Paul Chow. Enabling flexible network fpga clusters in a heterogeneous cloud data center. In *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 237–246. ACM, 2017.

[119] Anuj Vaishnav, Khoa Dang Pham, Joseph Powell, and Dirk Koch. Fos: A modular fpga operating system for dynamic workloads. *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, 13(4):1–28, 2020.

[120] Kizheppatt Vipin and Suhaib A. Fahmy. Fpga dynamic and partial reconfiguration: A survey of architectures, methods, and applications. *ACM Comput. Surv.*, 51(4), July 2018.

[121] Shu-Ting Wang, Hanyang Xu, Amin Mamandipoor, Rohan Mahapatra, Byung Hoon Ahn, Soroush Ghodrati, Krishnan Kailas, Mohammad Alian, and Hadi Esmaeilzadeh. Data motion acceleration: Chaining cross-domain multi accelerators. In *2024 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 1043–1062, 2024.

[122] Wei Wang, Miodrag Bolic, and Jonathan Parri. pvfpga: accessing an fpga-based hardware accelerator in a paravirtualized environment. In *Hardware/Software Codesign and System Synthesis (CODES+ISSS), 2013 International Conference on*, pages 1–9. IEEE, 2013.

[123] Robert N. M. Watson, Jonathan Anderson, Ben Laurie, and Kris Kennaway. A taste of capsicum: Practical capabilities for unix. *Commun. ACM*, 55(3):97–104, mar 2012.

[124] Gabriel Weisz and James C Hoe. Coram++: Supporting data-structure-specific memory interfaces for fpga computing. In *2015 25th International Conference on Field Programmable Logic and Applications (FPL)*, pages 1–8. IEEE, 2015.

[125] Shaojie Xiang, Yi-Hsiang Lai, Yuan Zhou, Hongzheng Chen, Niansong Zhang, Debjit Pal, and Zhiru Zhang. Heteroflow: An accelerator programming model with decoupled data placement for software-defined fpgas. In *Proceedings of the 2022 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, FPGA '22, page 78–88, New York, NY, USA, 2022. Association for Computing Machinery.

[126] AMD Xilinx. Alveo u280 data center accelerator card data sheet. https://docs.amd.com/r/en-US/ds963-u280/Summary, 2025. Last accessed: June 23, 2026.

[127] AMD Xilinx. Alveo v80 compute accelerator. https://www.amd.com/en/products/accelerators/alveo/v80.html, 2025. Last accessed: June 23, 2026.

[128] AMD Xilinx. Amd alveo u50 data center accelerator card. https://www.xilinx.com/products/boards-and-kits/alveo/u50.html, 2025. Last accessed: June 23, 2026.

[129] AMD Xilinx. Vitis accelerated libraries. https://github.com/Xilinx/Vitis_Libraries, 2025. Last accessed: June 23, 2026.

[130] AMD Xilinx. Vitis vision library. https://docs.amd.com/r/en-US/Vitis_Libraries/vision/index.html, 2025. Last accessed: June 23, 2026.

[131] AMD Xilinx. Xrt native apis. https://xilinx.github.io/XRT/master/html/xrt_native_apis.html, 2025. Last accessed: June 23, 2026.

[132] Hangchen Yu, Arthur Michener Peters, Amogh Akshintala, and Christopher J. Rossbach. Ava: Accelerated virtualization of accelerators. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '20, page 807–825, New York, NY, USA, 2020. Association for Computing Machinery.

[133] Yue Zha and Jing Li. Virtualizing fpgas in the cloud. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 845–858, 2020.

[134] Chen Zhang, Peng Li, Guangyu Sun, Yijin Guan, Bingjun Xiao, and Jason Cong. Optimizing fpga-based accelerator design for deep convolutional neural networks. In *Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, FPGA '15, page 161–170, New York, NY, USA, 2015. Association for Computing Machinery.

[135] Jiansong Zhang, Yongqiang Xiong, Ningyi Xu, Ran Shu, Bojie Li, Peng Cheng, Guo Chen, and Thomas Moscibroda. The feniks fpga operating system for cloud computing. In *Proceedings of the 8th Asia-Pacific Workshop on Systems*, pages 1–7, 2017.

[136] Sizhuo Zhang, Hari Angepat, and Derek Chiou. Hgum: Messaging framework for hardware accelerators. In *2017 International Conference on ReConFigurable Computing and FPGAs (ReConFig)*, pages 1–8, 2017.

[137] Mark Zhao, Niket Agarwal, Aarti Basant, Buğra Gedik, Satadru Pan, Mustafa Ozdal, Rakesh Komuravelli, Jerry Pan, Tianshu Bao, Haowei Lu, Sundaram Narayanan, Jack Langman, Kevin Wilfong, Harsha Rastogi, Carole-Jean Wu, Christos Kozyrakis, and Parik Pol. Understanding data storage and ingestion for large-scale deep recommendation model training: industrial product. In *Proceedings of the 49th Annual International Symposium on Computer Architecture*, ISCA '22, page 1042–1057, New York, NY, USA, 2022. Association for Computing Machinery.

[138] Udo Zoelzer. *Digital Audio Signal Processing*. John Wiley & Sons Software, 2008.

A Artifact Appendix

A.1 Abstract

This artifact contains the implementation and evaluation framework used in the USENIX OSDI 2026 paper, "μShell: A Microkernel-based FPGA Shell Architecture" by J. Chen, A. Panda, H. Unnibhavi, A. Koshiba, and P. Bhatotia. μShell is a hardware-OS co-design for modular accelerator deployment and execution. It provides capability-enforced isolation for secure resource sharing, component-aware task scheduling to reduce FPGA reconfiguration overheads, and inter-process communication (IPC) for peer-to-peer data transfers between vFPGAs.

A.2 Scope

The artifact reproduces all experimental results presented in the paper. It includes the hardware and software source code required to build the FPGA shell and host applications, as well as scripts for running the evaluation. To simplify deployment and reduce build times, we also provide precompiled FPGA bitstreams for both the shell and benchmark applications.

A.3 Contents

The artifact is published in our GitHub repository and contains the following items:

- `bitstreams/`: Precompiled FPGA bitstreams.
- `driver/`: Source code for the Linux kernel driver.
- `hw/`: Source code for the μShell shell.
- `examples_hw/`: Source code for benchmark application accelerators.
- `sw/`: Source code for the host-side runtime and infrastructure.
- `examples_sw/`: Source code for benchmark host applications.
- `evaluation/`: Scripts and supporting files for reproducing the evaluation results.

The repository root also contains scripts and configuration files required for FPGA programming and system setup.

A.4 Hosting

All the project source code, including the instructions for evaluating and building the software, is available in the following git repository: https://github.com/TUM-DSE/microShell. Please follow the instructions on the `master` branch to reproduce the results.

A.5 Requirements

We require the following software and hardware configurations to reproduce our experimental results.

A.5.1 Hardware Dependencies

- Machine with AMD EPYC 7413 CPU connected to public network.
- Xilinx (AMD) Alveo U280 FPGA cards.

A.5.2 Software Dependencies

- Operating system: NixOS 25.11 with Linux kernel 6.9.0-rc7.
- Nix: we use the Nix package manager to download all build dependencies for reproducibility. We use nix-shell to provide a consistent runtime environment.
- Python 3.10 or newer for the script that reproduces the evaluation.
- Vivado v2022.1 to compile the FPGA bitstreams and program the FPGA.

A.6 Methodology

- Figure 11 (§ 6.1): Throughput comparison across three deployment setups and varying data sizes, demonstrating the performance overhead of μShell.
- Figure 12 (§ 6.2): Comparison of scheduling metrics between μShell and Coyote, highlighting the benefits of the proposed scheduling algorithm.
- Figure 13 (§ 6.3): Reconfiguration latency comparison between μShell and Coyote for deploying multi-component applications.
- Table 5 (§ 6.4): Comparison of Source Lines of Code (SLoC) and Cyclomatic Complexity (CC) for host applications and user logic, illustrating the programmability improvements of μShell over Coyote.
- Table 6 (§ 6.5): FPGA resource utilization of μShell.

A.6.1 End-to-end performance (§ 6.1)

First, run the following commands to measure the benchmark throughput for different setups:

```
$ cd evaluation/scripts/
$ bash e2e_6.1/run_e2e.sh
 ↪  ~/microShell_base ~/microShell
```

To generate Figure 11, run the following command:

```
$ python3 e2e_6.1/plot_e2e.py
```

A.6.2 Scheduling improvements (§ 6.2)

First program the FPGA with the provided bitstream:

```
$ bash ./program_fpga.sh 6_2_sched
```

Build the software application:

```
$ cd ~/microShell
$ nix-shell shell.nix # this can be
 ↪  skipped if you are already in the nix-shell
$ mkdir build_sched_sw && cd build_sched_sw
$ cmake
 ↪  ../examples_sw/ -DEXAMPLE=scheduler_client
$ make
```

Copy some bash scripts that help to run the application from evaluation/scripts, then run the following commands to perform the experiments:

```
$ cp ../evaluation/scripts/scheduling_6.2/*.sh
 ↪  .
$ bash ./init_csv.sh
$ bash ./run_coyote.sh
$ bash ./run_ushell.sh
```

Copy the generated data back to the evaluation folder and create the plots:

```
$ cp *.csv ../evaluation/data/scheduling_6.2/
$ cd ~/microShell/evaluation/scripts
$ python3 scheduling_6.2/plot_sched.py
```

A.6.3 Application-deployment overheads (§ 6.3)

Open two terminals connected to our machine, in one for the server terminal:

```
$ cd ~\microShell
$ bash ./program_fpga.sh 6_3_pr
$ mkdir build_perf_server_sw/
 ↪  && cd build_perf_server_sw/
$
 ↪  cp
 ↪  ../bitstreams/cyt_top_pr_time_3_0807/config_*/*
 ↪  .
$ cmake ../examples_sw/ -DEXAMPLE=perf_server
$ make
$ sudo ./bin/test
```

In the second terminal for the client:

```
$ cd ~\microShell
$ nix-shell shell.nix # this can be
 ↪  skipped if you are already in the nix-shell
$ mkdir build_perf_client_sw/
 ↪  && cd build_perf_client_sw/
$ cmake ../examples_sw/ -DEXAMPLE=perf_client
$ make
$ sudo ./bin/test -1 true
```

Save the log into a file and copy it into the data folder, kill the server thread, and generate the plots:

```
$ journalctl -n 200 > reconfig_pr.log
$ cp reconfig_pr.log
 ↪  ../evaluation/data/deployment_6.3/
$ ps -aux | grep test
$ sudo kill -9 pid_of_test
$ cd ../evaluation/scripts
$ python3 deployment_6.3/extract_reconfig.py
$ python3
 ↪  deployment_6.3/plot_reconfig_overhead.py
```

A.6.4 Programmability (§ 6.4)

First, install the tools and run the measurements:

```
$ cd ~/microShell
$ nix-shell -p scc jq
$ cd evaluation/scripts/
$
 ↪  bash
 ↪  complexity_6.4/measure_complexity_baseline.sh
 ↪  ~/microShell_base
$
 ↪  bash
 ↪  complexity_6.4/measure_complexity_ushell.sh
 ↪  ~/microShell
```

The two CSV files are located at the path *evaluation/data/complexity_6.4*. To generate the data for the table, run the following commands:

```
$ python3
 ↪  complexity_6.4/extract_complexity.py \

    ↪  --baseline-csv
    ↪  path_to/complexity_baseline_results.csv
    ↪  \
   --ushell-csv
    ↪  path_to/complexity_ushell_results.csv
```

A.6.5 Resource Overheads (§ 6.5)

Run the following commands to extract the resource utilization of each component:

```
$ cd ~/microShell
$ nix-shell shell.nix # this can be
 ↪  skipped if you are already in the nix-shell
$ cd evaluation/scripts
$ python3 resource_usage_6.5/extract_util.py
$ python3
 ↪  resource_usage_6.5/extract_modules.py
$ python3
 ↪  resource_usage_6.5/plot_resource_usage.py
$ python3 scalability_2/plot_scalability.py
 ↪  --baseline ~/microShell_base
```

ARTIFACT EVALUATED
USENIX
AVAILABLE

Virtualizing eBPF with Late-Binding

Jing Zhang[1], *Xiaguannan Song*[2], *Dong Du*[1], *Yubin Xia*[1], *Binyu Zang*[1], *Haibo Chen*[1]

[1]*Shanghai Jiao Tong University*
[2]*Harbin Institute of Technology, Shenzhen*

Abstract

While eBPF has become the de facto standard for kernel customization in cloud-native systems, its design implicitly assumes a single trust domain. Allowing multiple tenants to deploy their own eBPF programs breaks this assumption, making the system both insecure and inefficient. We identify the root cause as eBPF's *static-binding model*, which rigidly couples logical eBPF programs to physical kernel hooks, forcing tenants to contend for shared execution contexts.

We propose ▽BPF, a virtualization layer that shifts to a *late-binding model*. By repurposing physical hooks as generic interposition points and deferring the binding until the event is attributed at runtime, ▽BPF decouples tenant context from the underlying kernel. ▽BPF achieves this via three key mechanisms: (1) a *Sniffer* that accurately attributes interrupt-driven events to tenants, (2) a *Dispatcher* that replaces linear traversal with scalable $O(1)$ program lookup, and (3) a compiler-assisted framework for state isolation. Implemented on Linux 6.12, ▽BPF enables the secure coexistence of multi-tenant workloads. Our evaluation shows that ▽BPF reduces latency by up to 3.9× (lmbench) and improves throughput by 29% (PostgreSQL) compared to native contention.

1 Introduction

In modern cloud and data center systems, the ability to securely customize kernel behavior is critical to ensuring the performance [8,21,27,66], security [14,19,30], and observability [14,38] guarantees required in multi-tenant environments. eBPF has emerged as the de facto standard technology for achieving this goal, giving rise to a new class of powerful infrastructure software. Platform providers such as Meta, Google, and Netflix [53, 57] routinely deploy trusted eBPF programs to implement core services, while projects like Cilium [8] and Falco [14] leverage this technology to deliver efficient network connectivity and security monitoring for all containerized workloads.

At the same time, a new frontier is emerging: enabling tenants to deploy their own eBPF programs. This unlocks fine-grained, application-specific optimizations that platform-level tooling cannot achieve. Recent studies demonstrate the value of tenant-defined eBPF in areas such as storage and I/O [47, 50, 87, 104, 105, 107, 111], scheduling [31, 33, 52, 71, 74, 110], networking [25, 43, 61, 108, 109], concurrency control [88, 89], profiling [54, 90] and AI [97].

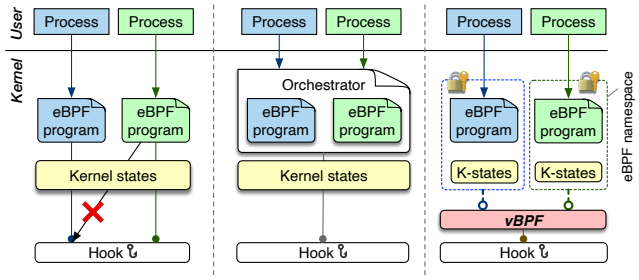
Beyond expanding customization, the use of tenant eBPF is becoming more frequent and more dynamic. Modern cloud workloads run in short-lived containerized environments [29, 53] and increasingly use eBPF for dedicated requirements. This trend is being further accelerated by automation (e.g., via AI agents [10, 17, 40, 51]), which can assist in generating or updating eBPF programs. As a result, tenant eBPF shifts from occasional operator-managed customization toward more frequent runtime adaptation.

However, allowing tenants to load their own programs into a kernel already managed by platform-level eBPF introduces architectural conflicts. The eBPF subsystem is designed under the assumption of a single trust domain, which may break when tenant and platform programs are forced to coexist.

This is not a hypothetical concern. For example, hooks based on `struct_ops` replace global function pointers. Customizing TCP behavior, such as pacing, therefore allows only one implementation to be attached system-wide, forcing a binary choice between the platform policy and a tenant policy. In addition, when multiple tenants use kprobes to trace the same kernel function, they may interfere with one another [54, 69]: a program from one tenant that alters a function's return value can silently corrupt the observation results of others [65, 92], leading to anomalous behavior.

These problems stem from a fundamental architectural flaw: *eBPF's reliance on a static-binding model*. During deployment, eBPF programs are verified, loaded, and bound to physical hooks. Subsequent hook events are broadcast to the fixed attached set, sharing the same execution context. This static coupling between a logical program and physical kernel resources is the root cause of multi-tenancy conflicts. Although the kernel permits multiple attachments to a single hook, it forces them to contend for the same execution context and shared kernel objects, creating unavoidable channels for performance interference and state corruption.

Existing mechanisms are merely workarounds that struggle against the limitations of this static model (Figure 1-a/b). First, *in-program filtering* [54] allows tenants to bypass irrelevant events via conditional checks, but these checks are



(a) Native Static Binding (b) Orchestrator Static Binding (c) Late Binding (Ours)

Figure 1: **eBPF for multi-tenancy.**

neither enforceable nor free, since every program must still be invoked. Second, *cgroup-based dispatch* [89,111] efficiently isolates workloads by scoping programs to specific cgroups, but it relies strictly on process context, leaving interrupt-driven events (e.g., packet processing) completely uncovered. Third, *centralized orchestrators* [7,22,45] can resolve attachment conflicts through global policies, but this rigidity sacrifices eBPF's inherent flexibility and consolidates logic into monolithic binaries that increase the verifier's burden, raising the risk of rejection. Most critically, all these approaches fail to provide state isolation, as tenants continue to contend for shared kernel objects, leaving the system vulnerable to cross-tenant semantic pollution.

Our key insight is that resolving these conflicts requires *decoupling the logical program from the physical hook through a late-binding execution model.* We argue that physical hooks should serve as generic interposition points rather than fixed execution destinations. Much like dynamic dispatch in polymorphic programming languages [1,2,41,94], the binding of an event to its program set and state view should be deferred until the event is attributed at runtime. By intercepting the event first and resolving its semantic context (i.e., the tenant), the system replaces the blind, indiscriminate execution of the static model with a context-aware dispatch.

However, realizing this late-binding model introduces three technical challenges. First, *it requires precise event attribution to identify the target tenant before execution.* This is difficult in interrupt-driven contexts where the execution context is detached from the logical tenant. Second, *it requires high-performance dynamic dispatch that conflicts with the existing kernel's linear organization.* Since current hooks are designed for sequential execution, selective dispatch would degrade to $O(N)$ complexity, whereas critical paths usually demand $O(1)$ latency for better scalability. Third, while late-binding enables state isolation by resolving tenant context, *achieving both high performance and maintainability is difficult.* Naive solutions must either rely on heavyweight hardware switching [54] or enforce invasive software constraints that break existing applications.

Based on our insight, we propose vBPF, a virtualization layer that implements this *late-binding model* (Figure 1-c). vBPF creates eBPF namespaces for each tenant and, upon an event, dynamically resolves the target program and virtual context. To realize this, vBPF introduces three key designs to overcome those challenges.

First, we introduce the vBPF *Sniffer*, which resolves tenant attribution for kernel events in interrupt contexts using a two-phase mechanism. The *context-aware phase* captures the mapping between interrupt-visible resources and namespaces during operations with process context, such as system calls. The *context-free phase* then efficiently determines the targeted tenant from interrupt-driven events, such as XDP packets, by querying this pre-recorded mapping that manages the complex and dynamic lifecycle of these mappings. Rather than attributing all interrupts, we target tenant-bearing resources (e.g., network flows) that cover the key interrupt-driven eBPF use cases in our scope.

Second, to resolve the conflict between linear organization and high-performance dispatch, we introduce the vBPF *Dispatcher*. It redesigns the multi-program architecture by replacing the kernel's existing linear structures with a hierarchical, hash-based index that mirrors the tenant namespace structure. This allows vBPF to locate a tenant's entry point in $O(1)$ time. To efficiently execute the resulting parent-child program chains (e.g., pod-level tracing over container logic), we introduce an optional path flattening: vBPF pre-compiles the entire chain into a single array, trading a small amount of memory for near-zero dispatch latency.

Third, we achieve state isolation without the performance-maintainability trade-off by leveraging a key observation: *eBPF programs interact with shared kernel state exclusively through specific entry points.* So we propose a *compiler-enforced state isolation framework.* At the base, a *static analyzer* audits these helpers, rejecting unsafe global state access by default. To resolve conflicts without changing the tenant code, we provide a *variable library* for kernel developers. This library automatically transforms stateful operations by delegating static variables to tenant-private instances and patching dynamic state accesses. This design ensures strict isolation with minimal runtime cost while preserving the original eBPF programming model.

We implement a prototype of vBPF on Linux kernel 6.12 and evaluate it on a series of microbenchmarks and real-world applications. Our results demonstrate that vBPF effectively resolves multi-tenancy conflicts while delivering superior performance isolation. For example, vBPF enables the coexistence of singleton hooks and allows distinct containers to deploy independent eBPF programs without interference. In multi-tenant scenarios, vBPF reduces latency by up to 3.9× (`lmbench`) and improves application throughput by 29% (`PostgreSQL`) compared to native eBPF, primarily by eliminating redundant execution.

vBPF is available at `https://github.com/vbpf-osdi-2026`.

2 The Case for eBPF Virtualization

2.1 eBPF Explained

eBPF has emerged as the main mechanism for safe kernel programmability in Linux. Its verifier [13] statically enforces strict safety properties before execution, and its JIT compiler [9] translates verified bytecode [5] into efficient native code. The Linux community has widely embraced this model, continuously adding new eBPF features [59] and making kernel customization practical for modern applications.

The architecture of eBPF is illustrated in Figure 2. eBPF programs are attached to predefined *hooks* within the kernel. They retrieve eBPF-specific data via *eBPF maps* and interact with global kernel states through *helper functions* or *kernel functions (kfuncs)*. The following presents three core components of eBPF.

Event-Driven Execution. Developers can write *eBPF programs* using tools like `bcc` [18] and `libbpf` [24], each of which serves as a specific user-defined kernel extension. Numerous *hooks* are distributed across various kernel subsystems, including system call entry/exit, tracepoints, traffic control, and more. eBPF programs are attached to these hooks, which are triggered by *events* like syscall invocations. When execution reaches a hook point, the kernel sequentially invokes the attached eBPF programs.

eBPF Map. The eBPF program is distinct from the map, and these two elements can be linked through system calls, enabling the eBPF program to access the specified map. Conversely, given that the maps are globally shared data structures, the userspace program only needs to get the corresponding `fd` to have access to the map. The eBPF map provides a variety of ways to access the map, such as the file system and the eBPF map ID. *The BPF file system* is a pseudo-file system, and userspace programs can access the map through the file system operation. Furthermore, the eBPF subsystem maintains a global ID table for the map, which enables the user to obtain the map's `fd` directly from the ID.

Global Kernel States and Access Interfaces. Although eBPF was designed to restrict access to arbitrary kernel memory, there are still limited interfaces to interact with specific kernel data structures. For example, eBPF *helper functions* provide safe operations on data structures such as `task_struct` and `sk_buff`, ensuring that arguments and pointers are strictly checked by the verifier. To further enhance accessibility, the eBPF *kernel function (KFunc)* mechanism introduces a whitelist of custom functions through BTF-based metadata [6]. Unlike helper functions, which rely on strict prototypes and stable APIs, KFuncs are verified mostly against BTF types and annotations. As a result, they can expose richer kernel-internal functionality, including functions that inspect or modify kernel state. However, KFuncs do not provide stability or availability guarantees across kernel versions and configurations.

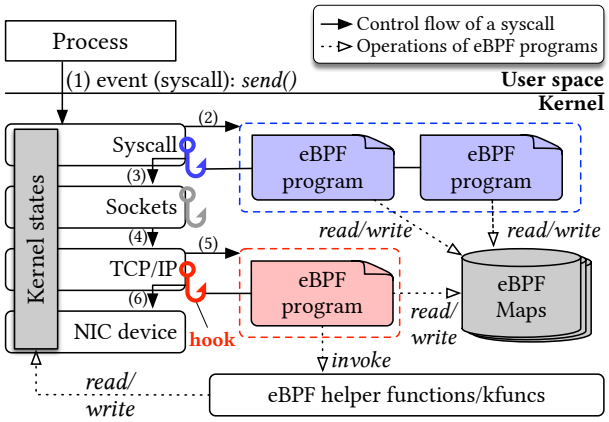


Figure 2: **eBPF in the Linux kernel.**

2.2 From Platform to Tenant eBPF

eBPF has become the standard for programmable cloud infrastructure, enabling providers to enforce policies and manage resources transparently. In this *platform-defined* model, trusted infrastructure components attach eBPF programs to manage all workloads in a shared kernel. For example, CNIs like Cilium [8] and Calico [27] use `XDP` and `TC` hooks for high-performance packet processing and load balancing; security systems like KubeArmor [30] enforce container-aware runtime constraints via LSM hooks; and observability tools like Pixie [19] and NetObserv [26] provide deep visibility into cluster health. These programs are typically managed by the platform administrator or a global eBPF manager [7, 22, 45], and therefore operate within a single trust domain.

However, restricting eBPF to this platform layer leaves a semantic gap: the platform controls kernel resources, but often lacks the application-specific context needed for optimal execution. A *tenant-defined* model closes this gap by allowing tenants to deploy eBPF logic tailored to their own workloads. For example, with the advent of `sched_ext` [31, 33], a latency-sensitive tenant can replace the generic CFS scheduler with a custom FIFO policy to minimize tail latency. Similarly, data-intensive applications (e.g., databases) can optimize page cache replacement policies based on query patterns [50, 104, 111]. Tenants can also deploy tracing programs to debug complex logic such as tracking specific lock contention or transaction latency. This shift transforms the kernel from a static resource manager into a programmable accelerator tailored to each tenant's specific workload.

Beyond enabling richer customization, the tenant-defined model also changes how eBPF programs are deployed and managed. Modern cloud workloads increasingly run in short-lived containerized environments, where requirements may evolve as applications scale, migrate, or change execution phases. Automation can further accelerate this trend (e.g., via AI agents [10, 17, 40, 51]), which can assist in generating or updating eBPF programs. As tenant eBPF shifts from occasional hand-written customization to more frequent runtime

```c
// container-A redirect to egress
SEC("tc/ingress")
int tc_ingress(struct __sk_buff *ctx) {
    if (iph.saddr == 0x9c3ca8c0) {
        iph.saddr = 0x0A0A0A01;
        return bpf_redirect(2, 0);
    }
    return TC_ACT_PIPE;
}

// container-B redirect back
SEC("tc/egress")
int tc_egress(struct __sk_buff *ctx) {
    if (iph.saddr == 0x0A0A0A01)
        return bpf_redirect(2, 1);
    return TC_ACT_PIPE;
}
```

Listing 1: **Infinite forwarding loop caused by functionality conflicts**. *Two tenants in TC redirect the same packets.*

adaptation, multi-tenant platforms must allow independently developed tenant programs, as well as platform-managed programs, to coexist in the same kernel.

2.3 The Challenges for Tenant-Defined eBPF

We observe three limitations of the existing eBPF design for tenant-defined eBPF in cloud-native platforms.

The Singleton Conflict. Certain powerful kernel hooks are designed as singletons, allowing only one eBPF program to govern a hook globally. This is particularly evident in hooks based on `struct_ops`, which function by replacing global function pointers. For example, customizing TCP behavior, such as overriding loss recovery, RTT sampling, or pacing, requires overwriting these global callbacks. So the first successful registration continues to own the singleton resource, and later conflicting registrations fail at load or attach time, preventing tenants from deploying distinct TCP policies. Similarly, the extensible scheduler (i.e., `sched_ext` [31]) operates by replacing the global scheduling class, ruling out the coexistence of multiple scheduling policies. Furthermore, eBPF iterators (e.g., `bpf_iter_task`) allow only a single `seq_ops` implementation to define iteration behavior, preventing multiple tenants from simultaneously customizing how they traverse system resources.

> **Limitation 1:** The singleton design of global eBPF hooks enforces a "winner-takes-all" model.

Functionality Conflicts. The global nature of the eBPF subsystem allows programs from different tenants to destructively interfere with one another. This manifests primarily as *control flow interference*. As illustrated in Listing 1, consider two benign tenants (A and B) attaching redirection logic to interconnected interfaces. Tenant A redirects packets to Interface-2, while Tenant B redirects them back. This creates an *infinite forwarding loop*, where a single packet circulates indefinitely, saturating the CPU and collapsing the network stack. While both programs are valid in isolation, their uncoordinated composition leads to catastrophic failure. Beyond control flow, *data dependency conflicts* occur when programs share mutable context. If a tenant modifies a packet header or overrides a kernel function return value (via kprobe), they silently corrupt the input state for subsequent programs.

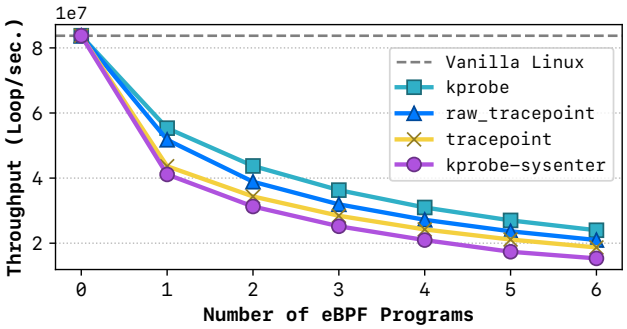


Figure 3: ***Performance interference on UnixBench [96].*** *The Y-axis represents syscall throughput (loops/second). Performance degraded when attaching eBPF programs to the getpid syscall. kprobe-sysenter attaches to* `sys_enter` *and relies on in-program conditional checks to filter for getpid, similar to tracepoint and raw_tracepoint.*

> **Limitation 2:** Shared execution context creates implicit dependencies between tenants, allowing side effects to corrupt correctness.

Performance Degradation. The lack of execution isolation forces a global broadcast model, where unrelated eBPF programs indiscriminately consume CPU cycles on critical paths. We quantified this "noisy neighbor" effect by measuring `getpid` throughput under unrelated eBPF attachments (Figure 3). The results reveal three key findings: (1) *Severe "zero-to-one" penalty*: attaching a single unrelated program causes a sharp throughput drop due to the overhead of enabling the indirect call to the eBPF program. (2) *Consistent runtime cost*: the degradation magnitude remains consistent across different hook types. (3) *Magnified impact on static tracing*: Unlike kprobes which trigger only on specific functions, (raw) tracepoints attach to the global `sys_enter` path, forcing the program to execute on every system call and significantly increasing unnecessary invocations.

> **Limitation 3:** The lack of scope-aware dispatching forces unrelated programs to execute on every event, imposing unavoidable overhead on all tenants.

2.4 Approaches to eBPF Multi-Tenancy

In-Program Filtering. The most rudimentary approach relies on tenants embedding filtering logic directly into their programs [54] (e.g., filtering by `PID`). However, this mechanism is neither mandatory nor enforceable, leaving the platform vulnerable to malicious or buggy tenants who may omit these checks to snoop on neighbors or interfere with system-wide operations. Furthermore, this approach incurs a prohibitive execution tax: *the kernel must invoke every attached program to evaluate these conditions on every event.* This results in unavoidable invocation overhead that scales linearly with the number of programs.

Cgroup-Based Dispatch. Linux cgroups [12] provide a kernel-native mechanism to attach eBPF programs to specific process hierarchies [89, 111]. While effective for cgroup-aware events (e.g., socket operations), this mechanism relies on specialized attachment types that are distinct from the standard hooks (e.g., tracepoints) used by many tools. Crucially, *it fails for interrupt contexts.* Hooks like `XDP` operate at the driver level without access to process information, rendering cgroup identification impossible. Moreover, cgroups only partition the attachment scope, not the underlying hook. Singleton hooks, such as `struct_ops` or specific LSM hooks, cannot be partitioned by cgroups, so only one tenant can attach at a time. Finally, cgroup-based dispatch does not isolate the kernel state observed or mutated by dispatched programs, making it insufficient to handle multi-tenancy.

Orchestrators Provided by Platforms. Platform orchestrators [7, 8, 45] attempt to resolve conflicts by merging multiple tenant programs into a single monolithic executable or a static `tail_call` chain. This centralization shifts the burden to the kernel verifier: *merging logic from multiple tenants increases complexity and instruction counts*, causing the combined program to exceed the verifier's strict limits [42] and potentially resulting in rejection. *Furthermore, dynamic updates incur significant latency.* Inserting or removing a single tenant's program often requires recompiling and atomically replacing the entire monolithic chain within the kernel. *Finally, this approach compromises portability.* Tenants must rewrite standard eBPF programs to fit rigid orchestrator APIs, sacrificing eBPF's inherent flexibility and tool compatibility.

Verification-Based Admission Control. Concurrent with our work, KrakenGuard [91] proposes a trusted user-space eBPF manager that uses symbolic execution to enforce fine-grained policies at load time. It provides policy-based admission control rather than a runtime virtualization layer, rejecting programs that violate policies or conflict with already loaded programs instead of enabling coexistence. KrakenGuard is therefore complementary to ▽BPF. It strengthens safety at load-time, while ▽BPF allows tenant programs to coexist transparently at runtime, including when they use singleton hooks or shared kernel state.

Approaches	Enforced Isolation	State Isolation	Global Scope	Verifier Scalability	Trans-parency
Conditional Checks [54]	✗	✗	✓	✓	✗
Cgroups BPF [89, 111]	✓	✗	✗	✓	✓
Orchestrators [7, 8, 45]	✓	✗	✓	✗	✗
KrakenGuard [91]	✓	▲	✓	▲	✓
▽BPF *(Our work)*	✓	✓	✓	✓	✓

Table 1: ***Existing approaches to eBPF multi-tenancy.***

Summary. As Table 1 shows, no existing mechanism satisfies the requirements for secure, scalable multi-tenancy. They either rely on tenant-provided checks, restrict only certain attachment scopes, compose programs ahead of time, or reject unsafe programs at admission time. Although these approaches can enforce useful restrictions, they lack architectural support needed for general multi-tenancy. The core challenge, therefore, is the absence of a virtualization layer that decouples the tenant's logical view of the kernel from the physical execution environment.

2.5 Threat Model and Assumptions

We target a multi-tenant shared-kernel platform where the host kernel, eBPF verifier, JIT compiler, container runtime, and platform administrator are trusted. Conversely, tenants and their eBPF programs are untrusted. Even if accepted by the verifier, a buggy or malicious tenant program could attempt to observe peer tenants, induce unnecessary execution on unrelated workloads, corrupt shared execution contexts, or manipulate helper-accessible kernel state beyond its authorized scope. Here, we consider the kernel state to include both internal data structures (e.g., `task_struct`) and eBPF-specific structures (e.g., eBPF maps).

The goal of ▽BPF is to confine tenant-defined eBPF execution and state updates to the tenant's scope, while preserving explicitly authorized hierarchical visibility for platform monitoring. ▽BPF does not protect against malicious host administrators, compromised kernel, verifier or JIT bugs, hardware faults, microarchitectural side channels, or resource-exhaustion attacks that should be handled by existing kernel mechanisms.

3 The Late-Binding Architecture

3.1 The Abstraction Gap of Static Binding

The core limitation of eBPF multi-tenancy is its reliance on a *static-binding model.* Currently, tenant programs are bound to physical hooks during deployment. Once a hook is triggered, the kernel dispatches the event to the hook's fixed attached program set, independent of tenant relevance. *This static coupling hardwires tenant code to a specific kernel hook and makes programs operate within a global memory space.* While the kernel permits multiple attachments, it forces tenants

to contend for the same execution context and global states, leading to state corruption and performance interference. Consequently, mechanisms built on today's eBPF model can only mitigate conflicts indirectly, for example through in-program filtering or external coordination of attachments. Because the underlying hooks and kernel state remain shared, these approaches still fall short of the isolation guarantees expected in multi-tenant environments.

3.2 Insight: Decoupling via Late-Binding

We propose that secure multi-tenancy requires a *late-binding architecture*. Rather than statically mapping logical programs to physical resources, ▽BPF introduces a layer of indirection. In this model, physical kernel hooks serve merely as interposition points. The binding of an event to its program set and virtual state view is deferred until the event is attributed at runtime. Upon an event, the ▽BPF runtime dynamically resolves which logical program to execute and which context to map based on the active tenant. By decoupling logical operations from physical realization, late binding transforms eBPF from a shared kernel extension facility into a virtualized runtime, enabling selective execution, precise attribution, and independent state management.

3.3 Design Overview

Inspired by the concept of late binding, we present ▽BPF, a novel eBPF execution model that achieves virtualization through three key techniques.

eBPF Namespaces. To isolate tenants, we introduce a new, hierarchical eBPF namespace bound to processes. A new namespace is essential for orthogonal policy composition, allowing security controls via ▽BPF to be decoupled from resource management like cgroups. This composable design enables fine-grained policies impossible with existing namespaces, such as applying different tracing rules to a sidecar and an application that share the same network. Moreover, its hierarchical structure allows parent namespaces (e.g., the host) to transparently audit child tenants. Following Linux namespace conventions, ▽BPF supports standard operations: creating new namespaces via `clone` [11] and `unshare` [39], and joining existing ones via `setns` [36]. This design enables seamless integration with container runtimes like runc [29] and orchestration platforms like Kubernetes [20].

Execution Context. The OS kernel operates in two distinct states [93]: *process context* and *interrupt context*. Process context handles synchronous user-space transitions (e.g., system calls), where the kernel can reliably identify the initiating tenant by inspecting the `current` task structure. In contrast, interrupt context handles asynchronous events like timer interrupts or packet arrivals that preempt the running task. When an interrupt occurs, the `current` task only represents the process that was interrupted (or the idle task), which is usually unrelated to the event's owner. Consequently, it is unsafe to rely on `current` to identify the target namespace.

▽BPF ***Sniffer.*** The first challenge for late-binding is to identify the tenant namespace accurately, particularly during interrupt-driven events (e.g., `XDP` packet arrival) where process context is absent. The ▽BPF *Sniffer* bridges this semantic gap using a *two-phase attribution mechanism*. During process-context operations (e.g., system calls), the sniffer captures the association between physical resources (e.g., sockets or block I/O requests) and tenant namespaces, storing this mapping in a global registry. When an interrupt fires, the sniffer queries this registry to resolve tenant attribution. This design enables ▽BPF to dynamically determine the tenant attribution for programs and events at runtime.

▽BPF ***Dispatcher.*** Once tenant attribution is resolved, the ▽BPF *Dispatcher* resolves the logical program to execute. Unlike the flat, linear traversal of native eBPF, the dispatcher organizes programs into tenant-scoped domains that mirror the namespace hierarchy. Furthermore, ▽BPF introduces two key optimizations. First, the dispatcher replaces existing linear scans with a hash-based lookup, locating the tenant's entry point in $O(1)$ time regardless of system scale. Second, to support parent-child relationships (e.g., global monitoring), the dispatcher offers a configurable trade-off: a memory-efficient recursive lookup or a path flattening optimization that pre-compiles the execution chain into a single array, trading memory for minimal dispatch latency.

▽BPF ***State Isolation Framework.*** Finally, ▽BPF achieves state isolation through a *compiler-enforced, multi-layered mechanism*. We leverage the observation that eBPF programs interact with shared kernel state exclusively through helper functions. At compile time, ▽BPF deploys a static analyzer to audit these interfaces and reject helper functions that modify global state unless explicitly allowlisted. At runtime, our *variable library* virtualizes state access for eBPF programs: it delegates static variables from global definitions to tenant-private instances and manages kernel structures via fine-grained dynamic patching. This allows eBPF developers to write standard eBPF code without modification, as the ▽BPF automatically redirects "global" writes to shadowed local copies to prevent cross-tenant pollution.

4 Detailed Design

4.1 ▽BPF Sniffer

The initial step of ▽BPF is to identify the correct eBPF namespace for triggered events. While this identification is straightforward in process contexts by inspecting the initiating task, it is non-trivial in interrupt-driven scenarios. As noted in §3, when a packet arrives, the *current* task only represents the interrupted process, which is typically unrelated to the packet's owner. ▽BPF *Sniffer* is designed to resolve this challenge.

Component	API	Description
Sniffer	`insert(writer)`	Extracts a stable key from *writer* and maps it to the current namespace. Called in process context.
	`resolve(reader)`	Looks up the target namespace using a key derived from *reader*. Called in interrupt context.
	`remove(key)`	Deletes the mapping associated with *key* to release resources. Called in process context.
Dispatcher	`get_namespace()`	Identifies the tenant namespace by inspecting the current process context or querying the Sniffer.
	`dispatch(ns)`	Locates the program set for namespace *ns* and invokes the appropriate tenant eBPF programs.
	`attach/detach(ns, prog)`	Registers or unregisters *prog* for namespace *ns*. Updates reference counts to trigger `enable/disable`.
	`enable/disable(ns)`	Lazily initializes the Dispatcher hook on the first attachment, or tears down on the last detach.

Table 2: Core abstractions of ▽BPF *Sniffer* and ▽BPF *Dispatcher*.

Observation. Our design is based on a key observation: *an interrupt event is linked to a resource that was previously set up in a process context.* E.g., a network packet arrives only after a process has established a connection. Thus, instead of relying on the unreliable execution context, we track the resource itself. *Sniffer* records the eBPF namespace for resources (e.g., network IP and port) during their initialization. Later, when an interrupt occurs, the system simply looks up this pre-recorded mapping to identify the target namespace.

Importantly, this resource-based attribution does not assume a strict one-to-one relationship between interrupt event and a task. Multiple interrupt events may resolve to the same namespace (e.g., packets from the same flow share a 5-tuple). Conversely, the resource visible at an interrupt may aggregate work from multiple tasks or namespaces, such as merged `bio/request` objects in the block layer. In such cases, the *Sniffer* records a namespace set as the mapped value and propagates it when requests are split or merged.

General Procedures. As Figure 4 shows, the *Sniffer* implements this logic through a two-phase lifecycle. In the *context-aware phase*, we instrument specific kernel resource allocation paths, such as `bind()` or `connect()` system calls. Since the process context is valid at these points, we capture the resource identifier (e.g., the socket's 5-tuple) and register a `{resource-ns}` entry in the resource-based mapping. In the *context-free phase*, which occurs during interrupts, we extract the identifier from the raw context. We then query mapping to retrieve the associated namespace, enabling precise dispatch without relying on the execution context.

Unified Key Interface. A major challenge is that the two phases see different data structures: the first phase sees high-level kernel objects (e.g., `sock`), while the second sees raw event metadata (e.g., `xdp_md`). To solve this, we *separate the matching mechanism from the data extraction policy (Table 2)*. We define the process-side context as a *writer* and the interrupt-side context as a *reader*. These interfaces serve as adapters that extract the same identity information from their different sources. This *unified key* enables `insert()` and `resolve()` to match resources across the boundary, independent of the data format.

Use Cases. We instantiate ▽BPF *Sniffer* for three main classes of resources. For networking (`TC/XDP`), a `network_sniffer` derives keys from packet headers (e.g., 5-tuples in `sk_buff/xdp_md`) and supports multi-stage matching across connection setup, binding, and acceptance, refining the *{resource-ns}* entry as more information becomes available. For storage, a `request_sniffer` keys on kernel I/O request objects (e.g., bio/request) so that completion interrupts can be resolved back to the namespace that submitted the operation. Finally, we use a `task_sniffer` to handle teardown corner cases: before a task's namespace is destroyed, it records a stable task-namespace mapping, allowing late process-context events in the exit path to identify the correct tenant even after the namespace reference in the task has been reclaimed.

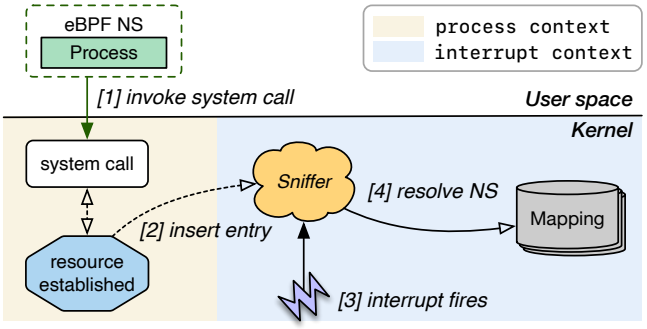


Figure 4: ▽BPF **Sniffer workflow.** *First, a process invokes a system call ([1]), triggering the Sniffer to record the mapping between the established resource and the namespace ([2]). Later, when a interrupt fires ([3]), the Sniffer uses this pre-recorded mapping to resolve the associated namespace ([4]).*

Scope. ▽BPF *Sniffer* targets kernel events whose handling spans both process and interrupt contexts. However, platform-level interrupts such as IPIs, device hotplug, and other hardware management events have no meaningful tenant owner and remain confined to the host namespace. This partition is deliberate: we only pay the cost of *Sniffer* where namespace identification is useful, and fall back to simpler, semantically correct mechanisms elsewhere.

4.2 ▽BPF Dispatcher

Once a tenant's eBPF namespace is identified, ▽BPF should dispatch the event to the correct eBPF programs. However,

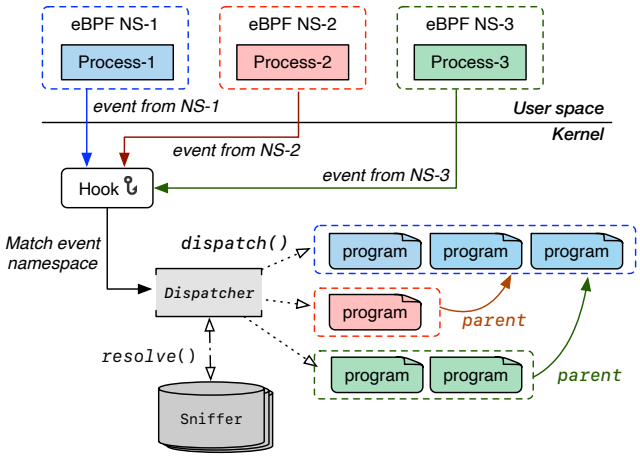


Figure 5: vBPF ***Dispatcher.*** *The Dispatcher sits at each hook, receives the namespace resolved by the Sniffer, and dispatches the event only to the matching tenant programs and eligible parent programs.*

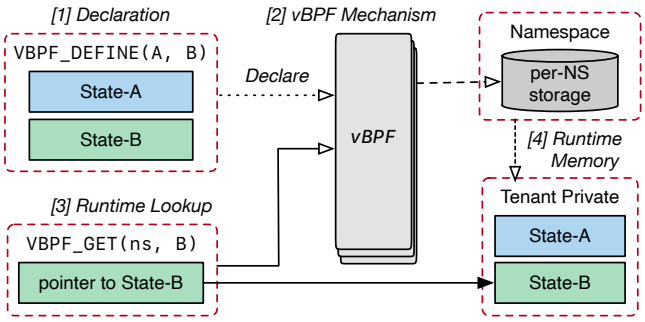


Figure 6: vBPF **tenant-private states.** *Developers declare multiple states using* `VBPF_DEFINE` *([1]), and then* vBPF *automatically maps them into a cohesive per-namespace storage layout ([2]). At runtime,* `VBPF_GET` *efficiently resolves ([3]) the specific state instance from the tenant-private memory ([4]).*

the native eBPF execution model is mismatched with the nature of multi-tenant environments. First, specific attachment points enforce exclusivity. For example, hooks based on `struct_ops` or LSM often accept only a single global program. Second, even for hooks that support multiple programs, the kernel relies on linear data structures (Figure 2). Concretely, kprobes utilize linked lists while tracepoints employ arrays to manage attached programs. A naive implementation might perform filtering at runtime, but this approach degrades performance by sequentially traversing unrelated programs. As shown in Figure 5, vBPF *Dispatcher* redesigns the program organization to address these challenges.

Virtual Hook Multiplexer. To implement the late-binding execution model, vBPF introduces a lightweight multiplexer attached to the physical hook. This component functions as a global indirection layer, similar to a virtual method table in object-oriented languages. Instead of binding tenant programs directly to kernel attachment points, the system installs this multiplexer as the sole occupant of the physical hook. Upon event triggering, the multiplexer acts as an entry point and queries the *Sniffer* to identify the corresponding tenant namespace. It then dynamically routes the event to logical programs registered by that tenant.

Hash-Based Tenant Indexing. Although subsystems like kprobes natively support multi-program attachment, they rely on a linear organization. These programs are then executed in a sequential manner, which is reasonable in today's Linux since all programs need to be executed. However, with vBPF, we need to check each program to ensure that only the targeted programs are executed, bringing a significant runtime overhead. vBPF replaces this linear scan with an O(1) dispatch mechanism based on tenant indexing. The *Dispatcher* uses a hash map, keyed by eBPF namespace, to directly locate a tenant's programs, which are themselves organized into a contiguous array. This design combines an efficient tenant lookup with a cache-friendly intra-tenant program traversal, ensuring that dispatch latency remains constant and isolated from the activities of other tenants.

Flattened Hierarchical Execution. For certain use cases, such as tracing, a hierarchical approach is necessary where parent namespaces should observe events in child namespaces. vBPF supports this via a bottom-up execution order, propagating events from the specific tenant up to the root. This allows platforms to audit, modify, or veto tenant actions, as context modifications are visible up the chain, and parents can override children's return codes. However, recursively walking the namespace tree on the critical path is inefficient. To address this, we introduce a *path flattening* optimization. During the program attachment, vBPF pre-computes the call chain from tenant to root as a static array. At runtime, dispatch becomes a simple iteration over this flattened path. Flattened path consistency is maintained atomically during parent program updates to prevent race conditions.

4.3 The vBPF State Isolation Framework

To deliver a virtualized environment, vBPF must enforce strict boundaries on state modification. The main goal is to ensure that tenants operate on independent state instances. For example, if two tenants attempt to modify the same `sk_buff`, the system must ensure they operate on separate local copies to avoid conflicts. However, achieving this is non-trivial. On the one hand, coarse-grained methods like page-table switching provide strong isolation but incur high overhead and lack the granularity to isolate small kernel objects efficiently. On the other hand, fine-grained methods that manually rewrite kernel logic are efficient but too invasive and hard to maintain across a large, changing codebase.

To address these challenges, vBPF proposes a multi-layered state isolation framework. At its foundation is a static analyzer that prevents access to unverified eBPF ker-

nel interfaces (e.g., eBPF helper functions). Building on this guarantee, a variables library provides high-level, transparent abstractions for managing both simple per-tenant state and complex, shared kernel objects.

The Static Analyzer. The core of vBPF's state isolation is a static analyzer built upon a key observation: *eBPF programs access kernel state exclusively through a bounded set of verifier-visible kernel interfaces, including eBPF helper functions and kfuncs.* This provides a natural checkpoint for enforcing isolation. Our analyzer, integrated into the kernel's compilation toolchain as a compiler frontend plugin, enforces a default-deny policy: *it rejects any eBPF kernel interface that can modify global kernel state unless it is explicitly annotated as safe.* This ensures tenant programs only use interfaces audited for multi-tenant environments, drastically reducing the attack surface for cross-tenant interference.

Analysis Procedure. Our static analyzer's design relies on two compiler attributes. `vbpf_helper` is integrated with the kernel's `BPF_CALL_*` macros to automatically identify the complete API surface of all functions exposed to eBPF. The second, `vbpf_safe`, is a manual annotation applied by kernel developers, serving as a formal assertion of state safety. It then employs a taint analysis to automatically verify the state safety of these helpers. If a side effect is detected, it then requires the `vbpf_safe` annotation as an explicit developer sign-off. To ensure soundness, we define rigorous propagation rules for complex control flows such as static branches, loops, and indirect calls. Any violation of these rules results in a compile-time error, preventing the creation of an insecure vBPF-enabled kernel.

Pointer Analysis. By default, the analyzer treats all argument pointers as tainted (unsafe). However, a naive rejection of all pointers yields excessive false positives. To resolve this, we leverage the eBPF verifier's existing type constraints. Our key observation is that the *verifier strictly defines the argument pointer types and memory regions a pointer can access.* For each analyzed interface, the analyzer reads its verifier-visible prototype (e.g., `bpf_func_proto` for helpers and BTF prototypes for kfuncs) and lifts each pointer argument into a safety category. Pointers that are guaranteed to access tenant-private memory (e.g., the eBPF stack, local maps) or read-only regions (e.g., string constants) are classified as safe. This verifier-assisted pointer classification allows vBPF to accept common interfaces that only write to tenant-owned buffers while still rejecting hidden mutations to shared kernel state.

The vBPF Variables Library. Our static analyzer provides a strong safety foundation, but shifts the burden of state management to developers. To automate this complex and error-prone task, we introduce the vBPF *variables library.* The design is guided by a key insight: *kernel state consists of two distinct categories requiring different management.* The first is *tenant-private state*, which can be safely instantiated per tenant. The second is *shared state*, which cannot be replicated and whose access must be virtualized. Our library provides two distinct mechanisms tailored to these categories: a highly-efficient, declarative API for tenant-private state, and a general state overlay engine for virtualizing shared state. eBPF kernel interfaces can use these mechanisms to redirect accesses to namespace-private instances or to record tenant-specific updates to shared objects.

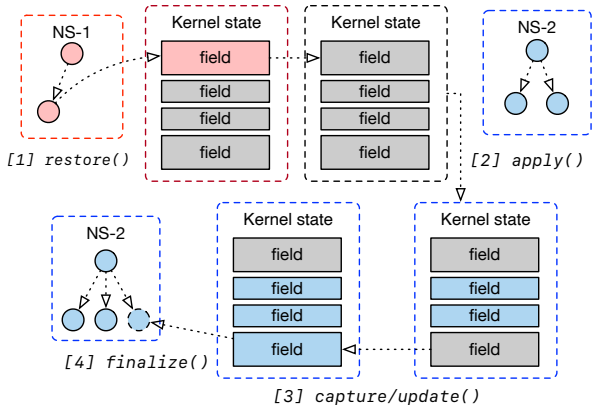


Figure 7: vBPF **semantic state overlay.** *Upon a context switch from NS-1 to NS-2,* vBPF *restores the kernel to a clean state ([1]) before applying NS-2's specific patches ([2]). During eBPF program execution, helper functions capture modifications ([3]), which are finalized and appended to the namespace's patch set ([4]) after the program exits.*

Declarative API. Manually converting a global variable into a per-tenant instance is a complex task. It requires defining a new type, managing memory allocation, and implementing lookup logic, which forces invasive changes across the kernel code. As Figure 6 shows, to minimize this complexity, vBPF introduces a declarative API that abstracts away these details. Instead of writing extensive boilerplate, developers simply use `VBPF_VARS_DEFINE` to declare a variable and `VBPF_FIELD_INIT` to register custom initialization logic. At runtime, accessing the tenant-specific instance requires only a single call to `VBPF_VARS_GET(ns, ...)`. Using this API, vBPF provides per-namespace pointers that allow exclusive access to the states. Fields in the same group are resolved cohesively from a single namespace instance, ensuring auxiliary objects like locks remain consistent with the state they protect. This abstraction handles the storage and mapping automatically, reducing hundreds of lines of code to just a few declarations.

Semantic State Overlay. Virtualizing shared state requires a way to inspect and understand kernel data structures at runtime. A simple memory snapshot is not enough because it cannot handle complex structures like dynamic arrays and does not understand the difference between pointers and data. To solve this, we use the BPF Type Format (BTF), which is embedded within the kernel image. *Although BTF was*

```c
// Precompute the context
struct overlay_ctx ctx;
overlay_ctx_init(&ctx, "sk_buff", skb);
// Hot path: record the patches
struct patch_builder *pb;
pb = overlay_begin(&ctx);
{ skb->priority++; }
overlay_capture(pb, skb);
/* Or use update */
overlay_update(pb, skb, struct sk_buff, tstamp);
overlay_finalize(&ctx, pb);
```

Listing 2: **Primitive usage in helper function.** *The helper initializes an overlay context for the target kernel object, records modifications through either full-object capture or field-level update, and finalizes them into the namespace's patch set.*

originally designed for debugging, we found that its detailed descriptions of data types and memory layouts make it perfect for reflections. As Figure 7 shows, we use BTF to implement a form of reflection that allows vBPF to understand the structure of kernel objects. This enables us to create a semantic overlay: instead of blindly copying memory, our engine calculates the exact size and layout of an object to generate a precise, safe patch that applies tenant-specific changes.

State Overlay in Practice. As presented in Listing 2, the state overlay workflow of vBPF begins by registering an overlay context based on the current object state. When an eBPF helper function executes, it initializes a builder to track the modification process. After the state is modified, the engine generates patches using one of two methods: a `capture` mode that automatically detects changes by comparing snapshots, or an `update` mode that allows manual recording of specific fields. These patches are then committed to the current namespace's patch set via a `finalize` operation. These patches are atomically applied when a tenant context switches in and reverted when they switch out.

5 Implementation

We have implemented a vBPF prototype based on Linux kernel 6.12 and LLVM 20, with 12k LoC for the kernel and 1k for the Clang frontend plugin.

Concurrency. Performance and correctness of vBPF under concurrency are paramount. To achieve this, we employ a hash table protected by Read-Copy-Update (RCU), enabling the query operations, such as *Sniffer* `resolve` and *Dispatcher* `get_namespace`, to proceed in a lock-free manner with minimal synchronization overhead. The insertion and removal operations in the hash table oversee object lifecycle management, with removals processed asynchronously. For scenarios with extremely large key sets where lookup misses are common, we offer an optional bloom filter to preemptively reject lookups for non-existent keys, further reducing cache pollution and lookup latency.

Preallocation and Lazy Allocation. vBPF paradoxically boosts performance using two opposing allocation strategies: eager preallocation and deferred lazy allocation. First, to guarantee the performance of our lock-free data structures, we employ aggressive preallocation. By using the kernel's `kmem_cache` and custom memory pools, we pre-provision objects to prevent runtime allocation failures and latency fluctuation within critical sections. Conversely, to minimize memory footprint and startup latency, a tenant-private state is not created with its namespace but is deferred until its first access. We adapt the *LazyCell* [23] concept for this: a program must first `VBPF_VARS_GET` a variable, which triggers a one-time initialization on the first call, thereby lowering cache pollution and improving data locality at scale.

Pre-Computation for Runtime Efficiency. A core design principle in vBPF is to aggressively shift computational work from hot runtime paths to the less frequent setup phase. We apply this in two critical components. First, for the *Dispatcher*'s hierarchical execution, we implement path flattening. At program load time, we pre-compute the entire call chain from a tenant to the root and cache it in an array. At runtime, this transforms a recursive traversal of the namespace hierarchy into a highly efficient iteration, achieving near-constant-time dispatch. Second, for the *state overlay engine*, we pre-parse BPF Type Format (BTF) metadata at initialization. We cache the layouts of kernel objects into a simple array of field offsets and sizes. This allows the runtime *capture* operation to bypass expensive BTF parsing entirely.

6 Evaluation

Hardware and Software. We use an x86-64 machine with Intel(R) Xeon(R) Gold 6330 CPU @ 2.00GHz, 28x2x2 cores, 512GB DRAM to evaluate microbenchmark and performance breakdown. To deploy our prototype system, we adopted the Linux 6.12 kernel and Ubuntu 24.04 LTS as the base system and built our implementation on top of it. eBPF programs and other components of vBPF are built on LLVM 20 with our customized Clang frontend plugin. In addition, all eBPF programs are jitted in our evaluation. For all benchmarks, we set the machine to run in performance mode and bound dedicated CPU cores for single-thread microbenchmarks.

Benchmarks. We evaluate vBPF on microbenchmarks and real-world applications. The microbenchmarks include `lmbench` [78, 98] and `UNIX Bench` [96]. We use real-world eBPF applications, `sysdig` [35] and `netobserv` [26], as monitor programs and applications like `PostgreSQL` [62] and `Apache` [37] as workload programs, and compare standard eBPF and vBPF in terms of performance degradation relative to vanilla Linux. We run `Postgres` [62] through Phoronix Test Suite and use the official benchmark tool `ab` [4] to eval-

uate Apache. Finally, we use `sched_ext` [31] as a use case to demonstrate the performance improvement of `redis` [28] and `7z` [3] decompression using vBPF.

6.1 Microbenchmarks

We use microbenchmarks to evaluate the runtime characteristics of vBPF. By comparing vanilla Linux, native eBPF, and vBPF, we aim to verify two key properties: (1) the low overhead of the *Sniffer/Dispatcher* mechanism, and (2) the performance isolation benefits for cross-tenant scenarios.

Methodology. We use lmbench [78, 98] to generate system workloads covering syscalls, I/O, process creation, and networking. To simulate a realistic eBPF workload, we run sysdig [35], a comprehensive system visibility tool, to trace system events. We evaluate three configurations: (1) *Native eBPF*, where sysdig is attached globally using the standard kernel mechanism, (2) vBPF *(Co-located)*, which runs lmbench workload and sysdig in the same tenant namespace, (3) vBPF *(Cross-Tenant)*, which runs lmbench workload and sysdig in different tenant namespaces.

Results. Figure 8 shows the latency results of lmbench [78, 98] and Figure 9 shows the bandwidth results. We quantify baseline cost by comparing vBPF (*Co-Located*) against *Native eBPF*. vBPF introduces acceptable overhead: up to 4.81% for syscall, 5.18% for select, 3.39% for process creation and 2.51% for networking. Performance remains comparable to *Native eBPF*, indicating efficient *Sniffer* and *Dispatcher* designs.

In cross-tenant scenarios, vBPF demonstrates significant advantages over *Native eBPF*. While *Native eBPF* imposes overhead on all processes through global hooks, vBPF bypasses execution for unmonitored tenants. For high-frequency operations, vBPF reports 0.258, 0.319, and 0.288 us for `NULL call`, `read`, and `write`, reducing latency by 3.7×, 3.8×, and 3.9× over *Native eBPF*. For network workloads, vBPF reports 17.55, 21.31, and 13.89 us for `RPC/UDP`, `RPC/TCP`, and `TCP/IP`, improving latency by 1.4×, 1.4×, and 1.5× over *Native eBPF*, restoring throughput for non-monitored tenants and approaching *Vanilla Linux* baseline performance.

6.2 Breakdown

We collect the latency of the `tcx` network packets tracing program to analyze the breakdown of the eBPF program execution time. We insert `tcx` to `lo` and use iperf3 to trigger packet transmissions. *Co-Loc(vBPF)* and *Co-Loc(+Sniffer)* both denote that `iperf3` and eBPF programs run in the same eBPF namespace. *+Sniffer* indicates employing a sniffer within the interrupt context, whereas *vBPF* signifies operation in the process context. *X-Tenant(vBPF)* and *X-Tenant(+Sniffer)* follow a similar pattern but with `iperf3` and the eBPF program in different eBPF namespaces.

The results are shown in Figure 10. A basic `tcx` program takes 1135 ns, covering core parsing and statistics function-

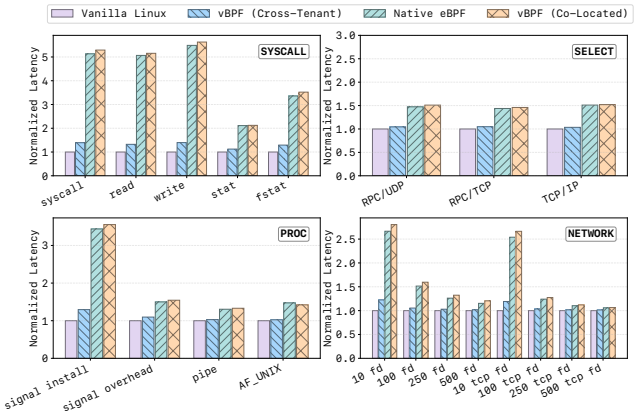


Figure 8: **Latency comparison of lmbench**

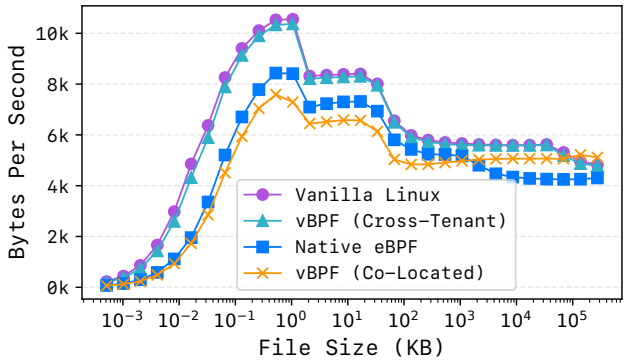


Figure 9: **Bandwidth comparison of file reading**

alities typical of eBPF overhead. In comparison, vBPF adds *sniffer resolve* at 134 to 136 ns, *namespace lookup* at 32 to 33 ns, and *program lookup* at 60 to 74 ns. The largest vBPF overhead accounts for only 2.1% of total eBPF execution, confirming minimal cost in co-located scenarios with substantial gains in cross-tenant settings.

6.3 Scalability Analysis

We further evaluate the scalability of vBPF using the same `tcx` program in §6.2, specifically focusing on the dispatch overhead as tenant density increases. Figure 11a depicts the latency of the *Sniffer* and *Dispatcher* as the number of active eBPF namespaces scales from 10 to 100. The results demonstrate that as the number of namespaces increases, the *sniffer resolve* latency stabilizes at the range of 131.7–136.6 ns and 58.3–60.7 ns for *program lookup*. This confirms our hash-based lookup maintains consistent low-latency performance in high-density, multi-tenant environments.

Figure 11b evaluates the dispatch efficiency when multiple kprobe programs contend for the same kernel function (`sys_read`). We compare vBPF against *Native eBPF* (sequential execution) and *vBPF (Co-located)* (manual conditional

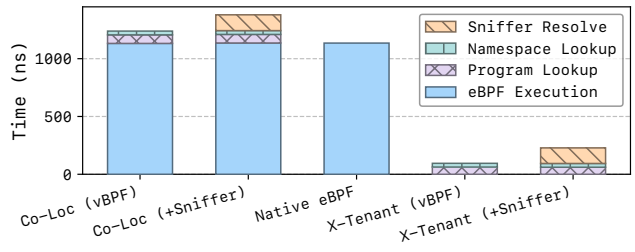
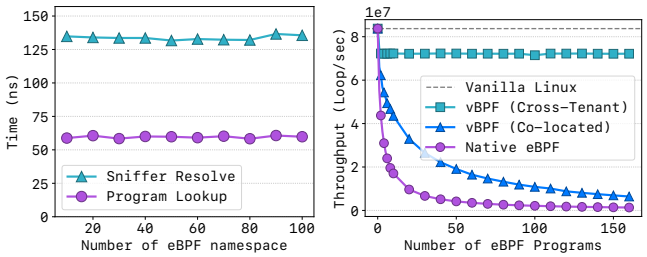


Figure 10: **Breakdown of tcx tracing program**



(a) **Sniffer/Dispatcher latency** (b) **kprobe performance**

Figure 11: **Scalability comparison**

checks). The results highlight a fundamental scalability gap. While both baselines exhibit linear performance degradation ($O(N)$) due to accumulated execution costs, *vBPF (Cross-Tenant)* maintains a flat throughput profile ($O(1)$). By efficiently bypassing unrelated programs, vBPF achieves a speedup of up to 54× compared to *Native eBPF* and 11.4× compared to the manual inner filter approach when inserting 160 kprobe programs, proving its ability to handle high-density multi-tenancy without performance degradation.

6.4 End-to-End Performance under Tracing

We use two real-world workloads and two real-world eBPF tracing applications to evaluate the end-to-end vBPF performance in cross-tenant scenarios.

PostgreSQL. PostgreSQL [62] is an open-source object-relational database system. We evaluate throughput (transactions per second) and latency with different scales, client numbers, and access modes (read-only or read-write). We use `sysdig` [35] as a background eBPF program to monitor system data. As Figure 12 shows, vBPF can offer 29% higher throughput and 23.6% lower latency compared to *Native eBPF*.

Apache. Apache `httpd` [37] is a widely used web server. We analyze the throughput (requests per second) of `httpd` with the `ab` [4] benchmark tool in different concurrent connection settings. Besides, we choose `netobserv-ebpf-agent` [26] as the background network monitor, which utilizes different types of eBPF programs, including kprobe, tracepoint, and traffic control (TC), etc., to collect network data. As Figure 13 shows, compared to the *Native eBPF*, vBPF can achieve up to 2.8× throughput speedup, and is close to *vanilla Linux*.

6.5 Use Cases: sched_ext

In this section, we demonstrate a use case of vBPF. The extensible scheduler class (`sched_ext`) [31] enables customized scheduling policies through eBPF programs. While `sched_ext` can improve performance for specific workloads, it lacks fine-grained isolation beyond cgroup boundaries, which can lead to global interference and performance degradation for unrelated workloads.

To demonstrate this capability, we evaluate two distinct workloads with contrasting characteristics: Redis [28], a key-value database optimized for low-latency operations, and `7z` [3], a file archiver performing CPU-intensive decompression. We employ `scx_central` [33], a scheduler that maximizes time slice allocation, making it particularly well-suited for CPU-bound workloads but potentially detrimental to latency-sensitive applications. We compare three configurations: (1) *Vanilla Linux* using the standard CFS scheduler, (2) *Native eBPF* using `scx_central` to schedule both `redis` and `7z`, and (3) vBPF using `scx_central` exclusively for `7z` while keeping `redis` under the default scheduler, where each scheduler manages a disjoint set of CPU cores.

As Figure 14 shows, *Native eBPF* improves `7z` [3] decompression performance by 10% compared to *Vanilla Linux*, but causes an 18% degradation in `redis` [28] throughput. In contrast, vBPF achieves the same 10% improvement for `7z` while keeping `redis` performance close to vanilla Linux.

6.6 State Overlay Overhead

We evaluate the overhead of the vBPF state overlay using three representative kernel data structures (`sk_buff`, `file` and `task_struct`) alongside a simplified `sk_buff_fake`. We measure the latency of the four critical operations: *capture / update* (calculate or manually create the state overlay), *apply* (switch-in), and *restore* (switch-out). In addition, we compare the optimized implementation against a naive baseline to quantify the benefits of precomputed layout strategy.

Results. In Linux 6.12, `sk_buff` is 232 bytes, `task_struct` is 10,112 bytes, and `file` is 184 bytes. `sk_buff_fake` simplifies `sk_buff` with reduced type complexity. In the test, we modify only one field in these data structures, which is common for a specific helper function. Figure 15 presents the results: the latency of *apply*, *restore*, and *update* remains stable, with the first two in the range of 42–44 ns and the latter at 52–56 ns. The latency of *capture* is influenced by the size of the type. In common cases (e.g., `sk_buff` and `file`), the overhead is 118.8 and 100.9 ns, respectively. In addition, our optimization can reduce the overhead up to 31.4×.

6.7 Memory Overhead

We measure the peak additional memory consumed by vBPF under representative workloads and attribute it to its major

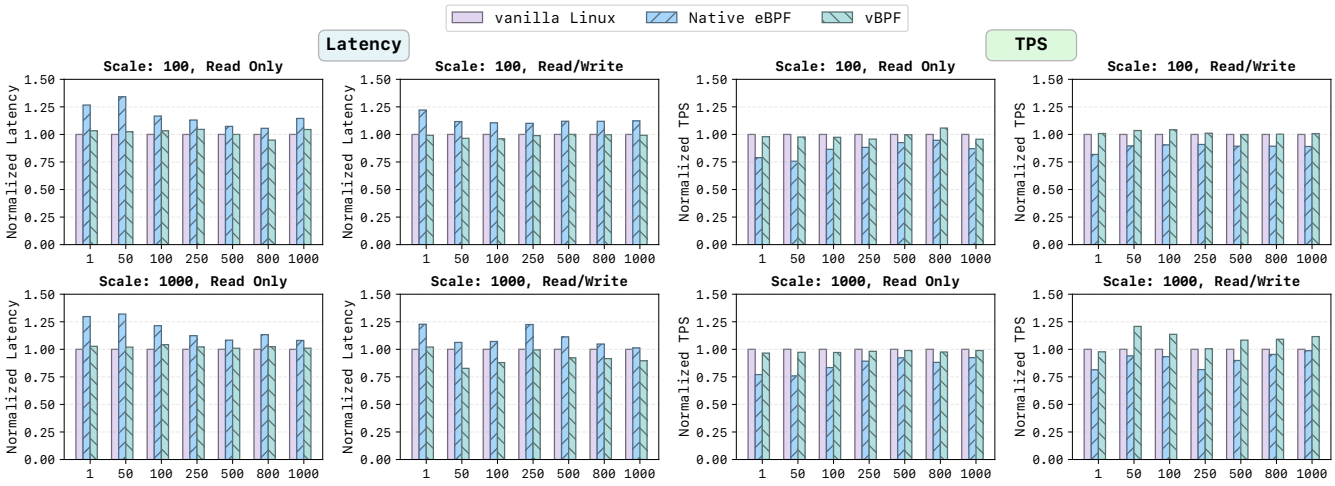


Figure 12: **Performance comparison of PostgreSQL benchmark under different workloads**

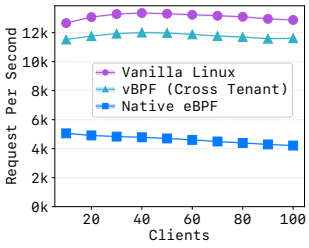


Figure 13: **Apache benchmark**

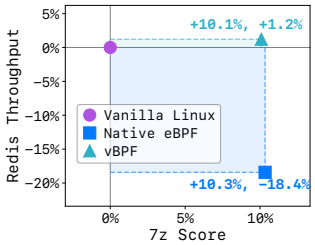


Figure 14: **Scheduler results**

Application	Peak Memory Overhead (KiB)				
	Dispatcher	Sniffer	State	Other	Total
Apache	19.3	39,334.1	0.2	16.4	39,370.0
PostgreSQL	1.8	3.4	0.2	6.2	11.6
`fio`	0.4	38,961.3	0.2	1.6	38,963.5

Table 3: **Peak memory overhead of vBPF by component**

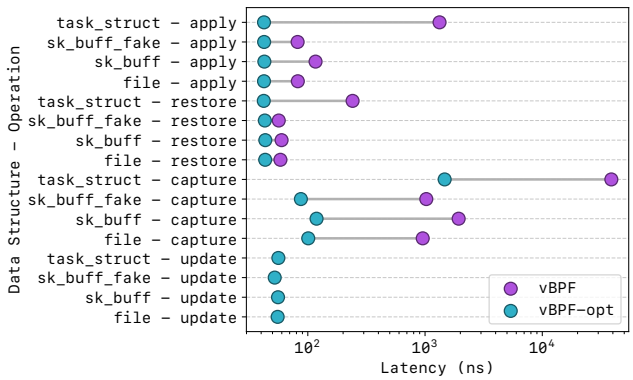


Figure 15: **Latency of vBPF state overlay.** *Capture latency depends on data-structure complexity, while apply and restore remain nearly constant. The optimized version uses precomputed layouts to avoid the runtime analysis.*

components. We use `netobserv`, sysdig, and a simple bio monitor program as the eBPF workload, respectively. Table 3 shows that the fixed metadata remains small: the dispatcher uses at most 19.3 KiB, state bookkeeping uses 0.2 KiB, and other metadata stays below 16.4 KiB. The dominant factor is the *Sniffer* registry, whose footprint depends on the number of live resource-to-namespace mappings. PostgreSQL creates few such mappings and adds only 11.6 KiB in total. In contrast, Apache and `fio` keep many network or I/O resources active at peak, increasing *Sniffer* memory to 39,334.1 KiB and 38,961.3 KiB, respectively.

The state component remains minimal because modifications are rare, and ID-based eBPF state (e.g., eBPF maps use `idr`) adds only constant per-namespace metadata. For workloads that frequently modify shared kernel objects, overlay state grows with the modified objects but remains a demand-driven cost for strict isolation. Thus, vBPF incurs negligible fixed metadata cost, while total memory usage mainly tracks the active *Sniffer* registry and state modifications.

7 Related Work

Kernel Isolation. There is a long line of research works introducing isolation to OS kernels [46, 55, 68, 70, 75, 80, 81, 85, 95]. Some of them [46, 68, 70, 75, 80, 81] use safe languages to extend the type system of the kernel. While others [95] utilize runtime checks to provide a sandbox execution environment. Besides, there are also a lot of works [55, 85] that leverage static verification, which has become a standard method of eBPF nowadays. Additionally, leveraging hardware-based isolation mechanisms [49, 63, 67, 84] has proven to be an effective approach for kernel isolation.

eBPF and Its Applications. eBPF has been used in many scenarios for optimizations. For instance, some works [8, 25, 48, 56, 61, 66, 103, 107, 109] focus on improving the I/O performance. Others are customizing the kernel components with eBPF, including memory management [50, 82, 102], locking [89], scheduling [31–34, 71, 74], file system [72], process management [106], page fault handling [110], etc.

eBPF Enhancements. eBPF enhancements have been a hot topic in recent years. There are a few works [76, 86, 99–101] that improve the verifier. Others are proposed to improve the code generation of eBPF. For example, Merlin [77] is a recent system that utilizes customized IR to optimize generated eBPF bytecodes. Morpheus [79] leverages traffic patterns to optimize datapath binaries, including eBPF bytecodes.

Mitigation of Tracing Overhead. [60] provides a comprehensive survey of tracing, while [83] reduces the unnecessary overhead for fuzzing. As for eBPF, KFuse [72] improves performance by merging multiple eBPF programs without failing the verifier. [54] provides isolated eBPF programs for each process. [69] eliminates the exception of kprobe, and improves the tracing performance. Most of them are orthogonal to vBPF and can be integrated with vBPF.

Multiple Kernel Views. Barrelfish [44] is an early work that splits the kernel into multiple independent components. FACE-CHANGE [64] and MultiK [73] propose per-process kernel view in one kernel. These efforts are too heavy to virtualize the eBPF programs and kernel states.

8 Discussion

Memory Costs. While memory overhead may increase with the number of state overlays over time, we believe that it is acceptable in practice for three reasons. First, most dynamic kernel states like `sk_buff` are short-lived, allowing immediate reclamation of allocated memory. Second, using copy-on-write strategy can make sure that it only stores the specific fields modified rather than the full objects. Third, writes are sparse since they occur solely through helper functions. As most eBPF programs are read-only for inspection, they incur zero additional memory cost.

Maintainability. As the kernel evolves, the number of modifiable states will increase naturally. Although vBPF introduces some maintenance overhead, it mitigates this challenge by leveraging the fact that eBPF programs modify state exclusively through well-defined entry points. Rather than manually rewriting arbitrary kernel code, maintainers need only inspect verifier-visible helper and kfunc boundaries to explicitly mark or virtualize stateful interfaces. Consequently, our static analyzer can seamlessly evolve alongside the kernel, requiring only a one-time annotation effort for each newly introduced function. While our current prototype focuses on helper functions due to their API stability, the underlying isolation mechanism is generic and can be readily extended to kfuncs given their semantic similarity.

State Consistency. vBPF ensures concurrency safety by leveraging synchronization mechanisms like RCU for atomic updates. Our current design assumes exclusive access (e.g., via locking) and does not explicitly account for RCU readers. However, this could be addressed by first reading the base variable and then independently applying per-tenant patches. We agree that virtualizing some kernel states is impractical, but this limitation is intentional. For example, global resources like timers should remain immutable to tenants. Working with our static analyzer, vBPF empowers kernel developers to define strict safety boundaries.

vBPF ***vs. MicroVMs.*** Hardware isolation [15, 16, 58] enforces strict binary isolation. vBPF addresses its limitations in two ways. First, efficiency: MicroVMs incur significant memory and startup overheads, while vBPF provides process-level agility where VM costs are prohibitive. Second, controlled sharing: patterns like sidecars require safe introspection, not total isolation. Unlike MicroVMs, which block visibility, vBPF enables selective virtualization with hierarchical auditing while preventing interference.

Generality of the Sniffer. The *Sniffer* is designed to capture the relationship between synchronous resource allocation and asynchronous event execution. By tracking stable logical resources instead of volatile packet headers, vBPF can handle complex scenarios like network tunnels and connection migration. Furthermore, this mapping mechanism can be offloaded to hardware to eliminate runtime overhead.

9 Conclusion

We present vBPF, a virtualization layer that adopts a late-binding execution model. Guided by the insight of dynamic dispatch in polymorphic languages, vBPF resolves the fundamental architectural conflicts of the static-binding model through a context-aware *Sniffer*, a hierarchical *Dispatcher*, and a state isolation framework. This work opens new directions for building safe, multi-tenant eBPF ecosystems.

Acknowledgments

We thank the anonymous OSDI reviewers and our shepherd for their constructive feedback. We are also grateful to our labmates for many helpful discussions, and to our families for their lasting support and encouragement. This work was supported in part by the Fundamental and Interdisciplinary Disciplines Breakthrough Plan of the Ministry of Education of China (JYB2025XDXM113), the National Natural Science Foundation of China (No. 62472279 and 62302300), and the Advanced Research Center for Agent-Oriented OS (TC20260106010). Corresponding author: Dong Du (dd_nirvana@sjtu.edu.cn).

References

[1] Early and Late Binding - Visual Basic. https://learn.microsoft.com/en-us/dotnet/visual-basic/programming-guide/language-features/early-late-binding/, 2021.

[2] Use early binding and late binding in Automation - Office. https://learn.microsoft.com/en-us/previous-versions/office/troubleshoot/office-developer/binding-type-available-to-automation-clients, 2021.

[3] 7-Zip. https://7-zip.org/, 2025.

[4] Ab - Apache HTTP server benchmarking tool - Apache HTTP Server Version 2.4. https://httpd.apache.org/docs/2.4/programs/ab.html, 2025.

[5] BPF Instruction Set Architecture (ISA) — The Linux Kernel documentation. https://docs.kernel.org/bpf/standardization/instruction-set.html, 2025.

[6] BPF Type Format (BTF) — The Linux Kernel documentation. https://docs.kernel.org/6.11/bpf/btf.html, 2025.

[7] bpfman: An eBPF Manager. https://bpfman.io/, 2025.

[8] Cilium - Cloud Native, eBPF-based Networking, Observability, and Security. https://cilium.io, 2025.

[9] Classic BPF vs eBPF — The Linux Kernel documentation. https://docs.kernel.org/bpf/classic_vs_extended.html, 2025.

[10] Claude Code on the web. https://claude.com/blog/claude-code-on-the-web, 2025.

[11] clone(2) — Linux manual page. https://man7.org/linux/man-pages/man2/clone.2.html, 2025.

[12] Control Groups — The Linux Kernel documentation. https://docs.kernel.org/admin-guide/cgroup-v1/cgroups.html, 2025.

[13] eBPF verifier — The Linux Kernel documentation. https://docs.kernel.org/bpf/verifier.html, 2025.

[14] Falco: Detect security threats in real time. https://falco.org/, 2025.

[15] Firecracker: Secure and fast microVMs for serverless computing. https://firecracker-microvm.github.io, 2025.

[16] Google gVisor: Application Kernel for Containers. https://github.com/google/gvisor, 2025.

[17] Introducing Codex. https://openai.com/index/introducing-codex/, 2025.

[18] Iovisor/bcc. https://github.com/iovisor/bcc, April 2025.

[19] Kubernetes Monitoring, Application Debug Platform | Pixie. https://px.dev/, 2025.

[20] Kubernetes: Production-Grade Container Orchestration. https://kubernetes.io/, 2025.

[21] Kubernetes-sigs/blixt. https://github.com/kubernetes-sigs/blixt, April 2025.

[22] L3AFD: Lightweight eBPF Daemon. https://github.com/l3af-project/l3afd, 2025.

[23] LazyCell in std::cell - Rust. https://doc.rust-lang.org/std/cell/struct.LazyCell.html, 2025.

[24] Libbpf/libbpf. https://github.com/libbpf/libbpf, April 2025.

[25] LKML: KP Singh: [PATCH bpf-next v9 0/8] MAC and Audit policy using eBPF (KRSI). https://lkml.org/lkml/2020/3/28/479, 2025.

[26] Netobserv/netobserv-ebpf-agent. https://github.com/netobserv/netobserv-ebpf-agent, April 2025.

[27] Project Calico. https://www.tigera.io/project-calico/, 2025.

[28] Redis - The Real-time Data Platform. https://redis.io/, 2025.

[29] runc: a CLI tool for spawning and running containers according to the OCI specification. https://github.com/opencontainers/runc, 2025.

[30] Runtime Security Enforcement | KubeArmor. https://kubearmor.io/, 2025.

[31] sched_ext. https://sched-ext.com/, 2025.

[32] Scx/case-studies/scx_layered.md at case-studies · sched-ext/scx. https://github.com/sched-ext/scx/blob/case-studies/case-studies/scx_layered.md, 2025.

[33] Scx/scheds/c at main · sched-ext/scx. https://github.com/sched-ext/scx/tree/main/scheds/c, 2025.

[34] Scx/scheds/c/scx_nest.bpf.c at main · sched-ext/scx. https://github.com/sched-ext/scx/blob/main/scheds/c/scx_nest.bpf.c, 2025.

[35] Security Tools for Containers, Kubernetes, and Cloud (Cloud Native usage report). https://sysdig.com/, 2025.

[36] setns(2) — Linux manual page. https://man7.org/linux/man-pages/man2/setns.2.html, 2025.

[37] The Apache HTTP Server Project. https://httpd.apache.org/, 2025.

[38] Tracee. https://aquasecurity.github.io/tracee/latest/, 2025.

[39] unshare(2) — Linux manual page. https://man7.org/linux/man-pages/man2/unshare.2.html, 2025.

[40] Claude Code overview. https://code.claude.com/docs/en/overview, 2026.

[41] Martin Abadi and Luca Cardelli. *A Theory of Objects*. Springer-Verlag, Berlin, Heidelberg, 1st edition, 1996.

[42] Daroc Alden. Taking BPF programs beyond one-million instructions. https://lwn.net/Articles/1017116/, 2025.

[43] Joshua Bardinelli, Yifan Zhang, Jianchang Su, Linpu Huang, Aidan Parilla, Rachel Jarvi, Sameer G. Kulkarni, and Wei Zhang. hydns: Acceleration of dns through kernel space resolution. In *Proceedings of the ACM SIGCOMM 2024 Workshop on EBPF and Kernel Extensions*, eBPF '24, page 58–64, New York, NY, USA, 2024. Association for Computing Machinery.

[44] Andrew Baumann, Paul Barham, Pierre-Evariste Dagand, Tim Harris, Rebecca Isaacs, Simon Peter, Timothy Roscoe, Adrian Schüpbach, and Akhilesh Singhania. The multikernel: a new os architecture for scalable multicore systems. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, SOSP '09, page 29–44, New York, NY, USA, 2009. Association for Computing Machinery.

[45] Theophilus A. Benson, Prashanth Kannan, Prankur Gupta, Balasubramanian Madhavan, Kumar Saurabh Arora, Jie Meng, Martin Lau, Abhishek Dhamija, Rajiv Krishnamurthy, Srikanth Sundaresan, Neil Spring, and Ying Zhang. Netedit: An orchestration platform for ebpf network functions at scale. In *Proceedings of the ACM SIGCOMM 2024 Conference*, ACM SIGCOMM '24, page 721–734, New York, NY, USA, 2024. Association for Computing Machinery.

[46] B. N. Bershad, S. Savage, P. Pardyak, E. G. Sirer, M. E. Fiuczynski, D. Becker, C. Chambers, and S. Eggers. Extensibility safety and performance in the spin operating system. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, SOSP '95, page 267–283, New York, NY, USA, 1995. Association for Computing Machinery.

[47] Ashish Bijlani and Umakishore Ramachandran. Extension framework for file systems in user space. In *Proceedings of the 2019 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC '19, page 121–134, USA, 2019. USENIX Association.

[48] Marco Spaziani Brunella, Giacomo Belocchi, Marco Bonola, Salvatore Pontarelli, Giuseppe Siracusano, Giuseppe Bianchi, Aniello Cammarano, Alessandro Palumbo, Luca Petrucci, and Roberto Bifulco. hxdp: Efficient software packet processing on fpga nics. *Commun. ACM*, 65(8):92–100, July 2022.

[49] Anton Burtsev, Vikram Narayanan, Yongzhe Huang, Kaiming Huang, Gang Tan, and Trent Jaeger. Evolving operating system kernels towards secure kernel-driver interfaces. In *Proceedings of the 19th Workshop on Hot Topics in Operating Systems*, HOTOS '23, page 166–173, New York, NY, USA, 2023. Association for Computing Machinery.

[50] Xuechun Cao, Shaurya Patel, Soo Yee Lim, Xueyuan Han, and Thomas Pasquier. Fetchbpf: customizable prefetching policies in linux with ebpf. In *Proceedings of the 2024 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC'24, USA, 2024. USENIX Association.

[51] Yinfang Chen, Manish Shetty, Gagan Somashekar, Minghua Ma, Yogesh Simmhan, Jonathan Mace, Chetan Bansal, Rujia Wang, and Saravan Rajmohan. Aiopslab: A holistic framework to evaluate ai agents for enabling autonomous clouds, 2025.

[52] Zhongjie Chen, Qingkai Meng, ChonLam Lao, Yifan Liu, Fengyuan Ren, Minlan Yu, and Yang Zhou. etran: extensible kernel transport with ebpf. In *Proceedings of the 22nd USENIX Symposium on Networked Systems Design and Implementation*, NSDI '25, USA, 2025. USENIX Association.

[53] Jonathan Corbet. BPF at Facebook (and beyond). https://lwn.net/Articles/801871/, 2019.

[54] Milo Craun, Khizar Hussain, Uddhav Gautam, Zhengjie Ji, Tanuj Rao, and Dan Williams. Eliminating ebpf tracing overhead on untraced processes. In *Proceedings of the ACM SIGCOMM 2024 Workshop on EBPF and Kernel Extensions*, eBPF '24, page 16–22,

New York, NY, USA, 2024. Association for Computing Machinery.

[55] Kumar Kartikeya Dwivedi, Rishabh Iyer, and Sanidhya Kashyap. Fast, flexible, and practical kernel extensions. In *Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles*, SOSP '24, page 249–264, New York, NY, USA, 2024. Association for Computing Machinery.

[56] Pekka Enberg, Ashwin Rao, and Sasu Tarkoma. Partition-aware packet steering using xdp and ebpf for improving application-level parallelism. In *Proceedings of the 1st ACM CoNEXT Workshop on Emerging In-Network Computing Paradigms*, ENCP '19, page 27–33, New York, NY, USA, 2019. Association for Computing Machinery.

[57] The Linux Foundation. The state of ebpf. Technical report, The Linux Foundation, January 2024. Licensed under CC BY-ND 4.0.

[58] Joshua Fried, Gohar Irfan Chaudhry, Enrique Saurez, Esha Choukse, Íñigo Goiri, Sameh Elnikety, Rodrigo Fonseca, and Adam Belay. Making kernel bypass practical for the cloud with junction. In *Proceedings of the 21st USENIX Symposium on Networked Systems Design and Implementation*, NSDI'24, USA, 2024. USENIX Association.

[59] Bolaji Gbadamosi, Luigi Leonardi, Tobias Pulls, Toke Høiland-Jørgensen, Simone Ferlin-Reiter, Simo Sorce, and Anna Brunström. The eBPF Runtime in the Linux Kernel, 2024.

[60] Mohamad Gebai and Michel R. Dagenais. Survey and analysis of kernel and userspace tracers on linux: Design, implementation, and overhead. *ACM Comput. Surv.*, 51(2), March 2018.

[61] Yoann Ghigoff, Julien Sopena, Kahina Lazri, Antoine Blin, and Gilles Muller. BMC: Accelerating memcached using safe in-kernel caching and pre-stack processing. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*, pages 487–501. USENIX Association, April 2021.

[62] PostgreSQL Global Development Group. PostgreSQL. https://www.postgresql.org/, November 2025.

[63] Jinyu Gu, Xinyue Wu, Wentai Li, Nian Liu, Zeyu Mi, Yubin Xia, and Haibo Chen. Harmonizing performance and isolation in microkernels with efficient intra-kernel isolation and communication. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pages 401–417. USENIX Association, July 2020.

[64] Zhongshu Gu, Brendan Saltaformaggio, Xiangyu Zhang, and Dongyan Xu. Face-change: Application-driven dynamic kernel view switching in a virtual machine. In *2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, pages 491–502, June 2014.

[65] Yi He, Roland Guo, Yunlong Xing, Xijia Che, Kun Sun, Zhuotao Liu, Ke Xu, and Qi Li. Cross container attacks: the bewildered ebpf on clouds. In *Proceedings of the 32nd USENIX Conference on Security Symposium*, SEC '23, USA, 2023. USENIX Association.

[66] Toke Høiland-Jørgensen, Jesper Dangaard Brouer, Daniel Borkmann, John Fastabend, Tom Herbert, David Ahern, and David Miller. The express data path: fast programmable packet processing in the operating system kernel. In *Proceedings of the 14th International Conference on Emerging Networking EXperiments and Technologies*, CoNEXT '18, page 54–66, New York, NY, USA, 2018. Association for Computing Machinery.

[67] Yongzhe Huang, Vikram Narayanan, David Detweiler, Kaiming Huang, Gang Tan, Trent Jaeger, and Anton Burtsev. KSplit: Automating device driver isolation. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 613–631, Carlsbad, CA, July 2022. USENIX Association.

[68] Galen C. Hunt and James R. Larus. Singularity: rethinking the software stack. *SIGOPS Oper. Syst. Rev.*, 41(2):37–49, April 2007.

[69] Jinghao Jia, Michael V. Le, Salman Ahmed, Dan Williams, Hani Jamjoom, and Tianyin Xu. Fast (trapless) kernel probes everywhere. In *2024 USENIX Annual Technical Conference (USENIX ATC 24)*, pages 379–386, Santa Clara, CA, July 2024. USENIX Association.

[70] Jinghao Jia, Raj Sahu, Adam Oswald, Dan Williams, Michael V. Le, and Tianyin Xu. Kernel extension verification is untenable. In *Proceedings of the 19th Workshop on Hot Topics in Operating Systems*, HOTOS '23, page 150–157, New York, NY, USA, 2023. Association for Computing Machinery.

[71] Kostis Kaffes, Jack Tigar Humphries, David Mazières, and Christos Kozyrakis. Syrup: User-defined scheduling across the stack. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, SOSP '21, page 605–620, New York, NY, USA, 2021. Association for Computing Machinery.

[72] Hsuan-Chi Kuo, Kai-Hsun Chen, Yicheng Lu, Dan Williams, Sibin Mohan, and Tianyin Xu. Verified programs can party: optimizing kernel extensions via

post-verification merging. In *Proceedings of the Seventeenth European Conference on Computer Systems*, EuroSys '22, page 283–299, New York, NY, USA, 2022. Association for Computing Machinery.

[73] Hsuan-Chi Kuo, Akshith Gunasekaran, Yeongjin Jang, Sibin Mohan, Rakesh B. Bobba, David Lie, and Jesse Walker. Multik: A framework for orchestrating multiple specialized kernels, 2019.

[74] Julia Lawall, Himadri Chhaya-Shailesh, Jean-Pierre Lozi, Baptiste Lepers, Willy Zwaenepoel, and Gilles Muller. Os scheduling with nest: keeping tasks close together on warm cores. In *Proceedings of the Seventeenth European Conference on Computer Systems*, EuroSys '22, page 368–383, New York, NY, USA, 2022. Association for Computing Machinery.

[75] Amit Levy, Bradford Campbell, Branden Ghena, Daniel B. Giffin, Pat Pannuto, Prabal Dutta, and Philip Levis. Multiprogramming a 64kb computer safely and efficiently. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, page 234–251, New York, NY, USA, 2017. Association for Computing Machinery.

[76] Dana Lu, Boxuan Tang, Michael Paper, and Marios Kogias. Towards functional verification of ebpf programs. In *Proceedings of the ACM SIGCOMM 2024 Workshop on EBPF and Kernel Extensions*, eBPF '24, page 37–43, New York, NY, USA, 2024. Association for Computing Machinery.

[77] Jinsong Mao, Hailun Ding, Juan Zhai, and Shiqing Ma. Merlin: Multi-tier optimization of ebpf code for performance and compactness. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, ASPLOS '24, page 639–653, New York, NY, USA, 2024. Association for Computing Machinery.

[78] Larry McVoy and Carl Staelin. lmbench: portable tools for performance analysis. In *Proceedings of the 1996 Annual Conference on USENIX Annual Technical Conference*, ATEC '96, page 23, USA, 1996. USENIX Association.

[79] Sebastiano Miano, Alireza Sanaee, Fulvio Risso, Gábor Ré tvári, and Gianni Antichi. Domain specific run time optimization for software data planes. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '22, page 1148–1164, New York, NY, USA, 2022. Association for Computing Machinery.

[80] Samantha Miller, Anirudh Kumar, Tanay Vakharia, Ang Chen, Danyang Zhuo, and Thomas Anderson. Enoki: High velocity linux kernel scheduler development. In *Proceedings of the Nineteenth European Conference on Computer Systems*, EuroSys '24, page 962–980, New York, NY, USA, 2024. Association for Computing Machinery.

[81] Samantha Miller, Kaiyuan Zhang, Mengqi Chen, Ryan Jennings, Ang Chen, Danyang Zhuo, and Thomas Anderson. High velocity kernel file systems with bento. In *19th USENIX Conference on File and Storage Technologies (FAST 21)*, pages 65–79. USENIX Association, February 2021.

[82] Konstantinos Mores, Stratos Psomadakis, and Georgios Goumas. ebpf-mm: Userspace-guided memory management in linux with ebpf, 2024.

[83] Stefan Nagy and Matthew Hicks. Full-speed fuzzing: Reducing fuzzing overhead through coverage-guided tracing. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 787–802, 2019.

[84] Vikram Narayanan, Yongzhe Huang, Gang Tan, Trent Jaeger, and Anton Burtsev. Lightweight kernel isolation with virtualization and vm functions. In *Proceedings of the 16th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE '20, page 157–171, New York, NY, USA, 2020. Association for Computing Machinery.

[85] George C. Necula and Peter Lee. Safe kernel extensions without run-time checking. In *Proceedings of the Second USENIX Symposium on Operating Systems Design and Implementation*, OSDI '96, page 229–243, New York, NY, USA, 1996. Association for Computing Machinery.

[86] Luke Nelson, Jacob Van Geffen, Emina Torlak, and Xi Wang. Specification and verification in the field: applying formal methods to bpf just-in-time compilers in the linux kernel. In *Proceedings of the 14th USENIX Conference on Operating Systems Design and Implementation*, OSDI'20, USA, 2020. USENIX Association.

[87] Chanyoung Park, Minu Chung, and Hyungon Moon. Selective on-device execution of data-dependent read i/os. In *Proceedings of the 23rd USENIX Conference on File and Storage Technologies*, FAST '25, USA, 2025. USENIX Association.

[88] Sujin Park, Irina Calciu, Taesoo Kim, and Sanidhya Kashyap. Contextual concurrency control. In *Proceedings of the Workshop on Hot Topics in Operating Systems*, HotOS '21, page 167–174, New York, NY, USA, 2021. Association for Computing Machinery.

[89] Sujin Park, Diyu Zhou, Yuchen Qian, Irina Calciu, Taesoo Kim, and Sanidhya Kashyap. Application-Informed kernel synchronization primitives. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 667–682, Carlsbad, CA, July 2022. USENIX Association.

[90] Vladimiro Paschali, Andrea Monterubbiano, Francesco Fazzari, Michael Swift, and Salvatore Pontarelli. Inxpect: Lightweight xdp profiling. In *Proceedings of the 3rd Workshop on EBPF and Kernel Extensions*, eBPF '25, page 48–54, New York, NY, USA, 2025. Association for Computing Machinery.

[91] Jainil Patel, Lucas Graeff Buhl-Nielsen, Adrien Ghosn, and Marios Kogias. KRAKENGUARD: Towards Fine-Grained eBPF isolation. In *23rd USENIX Symposium on Networked Systems Design and Implementation (NSDI 26)*, pages 2685–2704, Renton, WA, May 2026. USENIX Association.

[92] pat_h/to/file. DEF CON 29: Bad BPF - Warping reality using eBPF. https://blog.tofile.dev/2021/08/01/bad-bpf.html, 2021.

[93] Rusty Russell. Unreliable Guide To Hacking The Linux Kernel — The Linux Kernel documentation. https://www.kernel.org/doc/html/v6.12/kernel-hacking/hacking.html#user-context, 2024.

[94] Axel-Tobias Schreiner. *Object-Oriented Programming With ANSI-C*. Hanser, 1994.

[95] Margo I. Seltzer, Yasuhiro Endo, Christopher Small, and Keith A. Smith. Dealing with disaster: surviving misbehaved kernel extensions. In *Proceedings of the Second USENIX Symposium on Operating Systems Design and Implementation*, OSDI '96, page 213–227, New York, NY, USA, 1996. Association for Computing Machinery.

[96] Ben Smith. The byte unix benchmarks. *BYTE*, 15(3):273–277, March 1990.

[97] Prabhpreet Singh Sodhi, Georgios Liargkovas, and Kostis Kaffes. Empowering machine-learning assisted kernel decisions with ebpfml. In *Proceedings of the 3rd Workshop on EBPF and Kernel Extensions*, eBPF '25, page 28–30, New York, NY, USA, 2025. Association for Computing Machinery.

[98] Carl Staelin. Lmbench: An extensible micro-benchmark suite. *Software: Practice and Experience*, 35(11):1079–1105, 2005.

[99] Hao Sun and Zhendong Su. Validating the ebpf verifier via state embedding. In *Proceedings of the 18th USENIX Conference on Operating Systems Design and Implementation*, OSDI'24, USA, 2024. USENIX Association.

[100] Hao Sun, Yiru Xu, Jianzhong Liu, Yuheng Shen, Nan Guan, and Yu Jiang. Finding correctness bugs in ebpf verifier with structured and sanitized program. In *Proceedings of the Nineteenth European Conference on Computer Systems*, EuroSys '24, page 689–703, New York, NY, USA, 2024. Association for Computing Machinery.

[101] Harishankar Vishwanathan, Matan Shachnai, Srinivas Narayana, and Santosh Nagarakatte. Sound, precise, and fast abstract interpretation with tristate numbers. In *Proceedings of the 20th IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '22, page 254–265. IEEE Press, 2022.

[102] Zicheng Wang, Yicheng Guang, Yueqi Chen, Zhenpeng Lin, Michael Le, Dang K Le, Dan Williams, Xinyu Xing, Zhongshu Gu, and Hani Jamjoom. Seak: rethinking the design of a secure allocator for os kernel. In *Proceedings of the 33rd USENIX Conference on Security Symposium*, SEC '24, USA, 2024. USENIX Association.

[103] Zhe Yang, Youyou Lu, Xiaojian Liao, Youmin Chen, Junru Li, Siyu He, and Jiwu Shu. λ-io: a unified io stack for computational storage. In *Proceedings of the 21st USENIX Conference on File and Storage Technologies*, FAST'23, USA, 2023. USENIX Association.

[104] Anil Yelam, Kan Wu, Zhiyuan Guo, Suli Yang, Rajath Shashidhara, Wei Xu, Stanko Novakovic, Alex C. Snoeren, and Kimberly Keeton. Pageflex: flexible and efficient user-space delegation of linux paging policies with ebpf. In *Proceedings of the 2025 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC '25, USA, 2025. USENIX Association.

[105] Ioannis Zarkadas, Tal Zussman, Jeremy Carin, Sheng Jiang, Yuhong Zhong, Jonas Pfefferle, Hubertus Franke, Junfeng Yang, Kostis Kaffes, Ryan Stutsman, and Asaf Cidon. Bpf-of: Storage function pushdown over the network, 2023.

[106] Jianwei Zheng, Zhenhua Li, Feng Qian, Wei Liu, Hao Lin, Yunhao Liu, Tianyin Xu, Nan Zhang, Ju Wang, and Cang Zhang. Rethinking process management for interactive mobile systems. In *Proceedings of the 30th Annual International Conference on Mobile Computing and Networking*, ACM MobiCom '24, page 215–229, New York, NY, USA, 2024. Association for Computing Machinery.

[107] Yuhong Zhong, Haoyu Li, Yu Jian Wu, Ioannis Zarkadas, Jeffrey Tao, Evan Mesterhazy, Michael Makris, Junfeng Yang, Amy Tai, Ryan Stutsman, and Asaf Cidon. XRP: In-Kernel storage functions with eBPF. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 375–393, Carlsbad, CA, July 2022. USENIX Association.

[108] Beihao Zhou, Samer Al-Kiswany, and Mina Tahmasbi Arashloo. Toward ebpf-accelerated pub-sub systems. In *Proceedings of the 3rd Workshop on EBPF and Kernel Extensions*, eBPF '25, page 38–44, New York, NY, USA, 2025. Association for Computing Machinery.

[109] Yang Zhou, Zezhou Wang, Sowmya Dharanipragada, and Minlan Yu. Electrode: Accelerating distributed protocols with eBPF. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 1391–1407, Boston, MA, April 2023. USENIX Association.

[110] Tal Zussman, Teng Jiang, and Asaf Cidon. Custom page fault handling with ebpf. In *Proceedings of the ACM SIGCOMM 2024 Workshop on EBPF and Kernel Extensions*, eBPF '24, page 71–73, New York, NY, USA, 2024. Association for Computing Machinery.

[111] Tal Zussman, Ioannis Zarkadas, Jeremy Carin, Andrew Cheng, Hubertus Franke, Jonas Pfefferle, and Asaf Cidon. cache_ext: Customizing the page cache with ebpf. In *Proceedings of the ACM SIGOPS 31st Symposium on Operating Systems Principles*, SOSP '25, page 462–478, New York, NY, USA, 2025. Association for Computing Machinery.

A Artifact Evaluation

This appendix describes the artifact for vBPF and how reviewers can build and exercise it. The artifact entry point is the `.github` repository under the GitHub organization:

https://github.com/vbpf-osdi-2026/

The profile README in that repository gives the full command-by-command guide. The artifact is split into four sibling repositories, which should be checked out under the same parent directory.

Repository	Contents
`llvm-vbpf`	LLVM, Clang, LLD, the vBPF compiler support, and the `VBPFAttribute` Clang plugin.
`linux-vbpf`	Linux 6.12 kernel tree with vBPF support and an evaluation-ready kernel configuration.
`vm`	QEMU/KVM virtual-machine scripts for booting and testing the vBPF kernel.
`ebpf-bootstrap`	Example eBPF programs, helper tools, `bpftool`, and test scripts used to exercise BPF namespaces.

Table 4: vBPF ***artifact repositories.***

A.1 Supported Claims

The artifact is designed to support the main functional claims of the paper. First, it builds the modified LLVM/Clang toolchain and the `VBPFAttribute` plugin used to check vBPF-related annotations during the kernel build. Second, it builds a Linux 6.12 kernel with the vBPF core, Sniffer, benchmark interface, and sample code enabled. Third, it boots that kernel in a QEMU/KVM VM and runs example eBPF programs in ordinary and isolated BPF namespaces. Fourth, it provides scripts to inspect namespace-local programs and maps and to run repeated loaders in one or more BPF namespaces.

The full performance study in §6 used an x86-64 machine with Intel Xeon Gold 6330 CPUs and 512 GB DRAM, Ubuntu 24.04 LTS, Linux 6.12, and an LLVM 20 based toolchain. The artifact VM is intended as a reviewer-friendly environment for build and functionality validation. The benchmark workloads in §6, including `lmbench`, UNIX Bench, PostgreSQL, Apache, `sysdig`, `netobserv-ebpf-agent`, and `sched_ext`, can be run on comparable hardware for performance reproduction.

A.2 Host Requirements

The artifact expects a Linux host with KVM support. On Ubuntu or Debian, the host-side dependencies are standard compiler, kernel-build, and QEMU packages:

```
sudo apt update
sudo apt install -y \
  git cmake ninja-build make clang lld llvm \
  build-essential pkg-config libelf-dev zlib1g-dev \
  libbfd-dev libcap-dev openssl libssl-dev binutils \
  qemu-system-x86 genisoimage bridge-utils
```

The VM launcher uses `uv` and Python 3.12 or newer. Reviewers should install `uv` if it is not already available on the host.

A.3 Checkout

The repositories should be placed under one artifact root so that scripts can refer to them consistently through `AE_ROOT`:

```
mkdir -p ~/vbpf-ae
cd ~/vbpf-ae

git clone https://github.com/vbpf-osdi-2026/llvm-vbpf.git
git clone https://github.com/vbpf-osdi-2026/linux-vbpf.git
git clone https://github.com/vbpf-osdi-2026/vm.git
git clone https://github.com/vbpf-osdi-2026/ebpf-bootstrap.git

export AE_ROOT="$PWD"
```

A.4 Build LLVM and the Static Analyzer

Reviewers first build the modified LLVM tree. This produces the Clang toolchain used by the kernel build and the `VBPFAttribute.so` plugin.

```
cd "$AE_ROOT/llvm-vbpf"

cmake -S llvm -B build -G Ninja \
  -DCMAKE_BUILD_TYPE=Release \
  -DLLVM_ENABLE_PROJECTS="clang;lld" \
  -DLLVM_TARGETS_TO_BUILD="X86;BPF" \
  -DCMAKE_INSTALL_PREFIX="$PWD/install"

cmake --build build
cmake --build build --target VBPFAttribute
sudo cmake --install build

export LLVM_HOME="$AE_ROOT/llvm-vbpf/build"
export PATH="$LLVM_HOME/bin:$PATH"
export LD_LIBRARY_PATH="$LLVM_HOME/lib:$LD_LIBRARY_PATH"
```

The expected sanity checks are that `clang –version` uses the newly built toolchain and that `$LLVM_HOME/lib/VBPFAttribute.so` exists. The LLVM repository also includes a standalone analyzer test at `clang/examples/VBPF/tests/test.c`. It intentionally contains both accepted and rejected helper patterns, so a non-zero exit status is expected when the analyzer reports unsafe global-state writes or unresolved unsafe calls.

A.5 Build the vBPF Kernel

The kernel is built after LLVM because the build loads the `VBPFAttribute` plugin. The provided configuration enables baseline BPF support, BPF JIT, BPF LSM instrumentation, `CONFIG_BPF_VBPF`, `CONFIG_VBPF_SNIFFER`, `CONFIG_VBPF_BENCH`, and the sample vBPF programs.

```
cd "$AE_ROOT/linux-vbpf"

make LLVM=1 -j"$(nproc)" \
  KCFLAGS="-fplugin=$LLVM_HOME/lib/VBPFAttribute.so \
  -Xclang -add-plugin -Xclang vbpf_attrs -ferror-limit=1000"
```

The kernel image used by the VM is:

```
$AE_ROOT/linux-vbpf/arch/x86/boot/bzImage
```

A.6 Prepare and Boot the VM

The `vm` repository prepares an Ubuntu guest image and boots it with the ▽BPF kernel. The first step initializes the image and cloud-init seed:

```
cd "$AE_ROOT/vm"
uv sync
uv run vm.py --prepare
```

The QEMU bridge helper should allow `virbr0`, and the bridge should be available before the first installation boot:

```
sudo mkdir -p /etc/qemu
echo "allow virbr0" | sudo tee /etc/qemu/bridge.conf
sudo chmod 0644 /etc/qemu/bridge.conf

sudo ip link add name virbr0 type bridge
sudo ip addr add 192.168.122.1/24 dev virbr0
sudo ip link set virbr0 up
```

If `virbr0` already exists, reviewers can keep the existing bridge. The first boot installs and initializes the guest:

```
uv run vm.py --install
```

After shutting down the initialized guest, reviewers set `kernel.path` in `$AE_ROOT/vm/config.json` to the absolute path of `linux-vbpf/arch/x86/boot/bzImage`. The VM can then be started with:

```
cd "$AE_ROOT/vm"
uv run vm.py
```

The default guest login is `ubuntu`/`ubuntu`, and the expected SSH address is `192.168.122.10`.

A.7 Build and Run the Examples

After the VM boots the ▽BPF kernel, reviewers copy the examples into the guest, install guest-side build dependencies, and build the loaders:

```
scp -r "$AE_ROOT/ebpf-bootstrap" ubuntu@192.168.122.10:~
ssh ubuntu@192.168.122.10
cd ~/ebpf-bootstrap

sudo apt update
sudo apt install -y \
  clang cmake make pkg-config libelf-dev zlib1g-dev git \
  libbfd-dev libcap-dev llvm-dev openssl libssl-dev \
  ninja-build g++-14

git submodule update --init --recursive
CXX=g++-14 cmake -S . -B build -G Ninja
cmake --build build
```

Before running loaders, reviewers should confirm that the ▽BPF kernel exposes BTF metadata:

```
ls /sys/kernel/btf/vmlinux
```

A basic end-to-end check starts the `kprobe` loader, observes trace output, and triggers events:

```
sudo ./build/kprobe
sudo mount -t debugfs none /sys/kernel/debug
sudo cat /sys/kernel/debug/tracing/trace_pipe
./trigger_kprobe.sh
```

The examples also provide a wrapper test:

```
sudo ./test.sh ./build/kprobe
```

A.8 Namespace Checks

Reviewers can inspect programs and maps in the current BPF namespace with the artifact-built `bpftool`:

```
sudo ./build/bpftool/bpftool prog show
sudo ./build/bpftool/bpftool map show
```

To exercise namespace isolation, the artifact provides an `unshare` helper that starts a loader inside a fresh BPF namespace:

```
sudo ./build/unshare -b ./build/kprobe
sudo ./build/unshare -b ./build/bpftool/bpftool prog show
sudo ./build/unshare -b ./build/bpftool/bpftool map show
```

For repeated loaders in one namespace or across nested BPF namespaces, use:

```
sudo ./multi.sh 4 -- ./build/kprobe
sudo ./ns_multi.sh ./build/kprobe 4 2
```

Successful runs show that eBPF programs and maps can be loaded, inspected, and isolated by BPF namespace under the ▽BPF kernel. These checks exercise the same mechanisms used by the paper's Sniffer and Dispatcher evaluation, while the larger performance experiments can be run with the benchmark workloads described in §6.

DVLA: Dynamic VM Lifetime Aware Scheduling for Drifting Lifetime Distributions and Long-Lived VM Placement Debt (Operational Systems)

Zhengtong Zhang, Zihan Xu, Zhidong Hu, Yanbo Shan, Fei Peng, Suhong Chen,
Kaiyuan Shen, Xiangyun Kong, Handu Ding, Bing He, Binda Ma

Alibaba Cloud Computing

Abstract

Efficient Virtual Machine (VM) scheduling is critical for maximizing resource utilization in cloud computing. However, state-of-the-art lifetime-aware schedulers face two critical issues in real-world deployments. First, their static policies are brittle against the significant *spatial and temporal drifts of VM lifetime distributions*. Second, and more insidiously, their placement strategies inadvertently scatter long-lived VMs, creating a persistent *long-lived VM placement debt*. This debt, compounded by inevitable prediction errors, pins down machines and cripples cluster-wide resource reclamation, and cannot be repaid by online scheduling alone.

To address these challenges, we present *Dynamic VM Lifetime Aware scheduling (DVLA)*, an end-to-end system that synergistically combines online scheduling with offline rectification. DVLA comprises four key components: (1) a Hierarchical Lifetime Prediction Model that delivers multi-horizon predictions to inform both initial placement and offline optimization; (2) a Dynamic Affinity Grouping strategy that adapts to workload distribution drifts in real time; (3) a Debt-Aware Placement Policy (DAPP) that proactively consolidates long-lived VMs to minimize debt creation at the source; and (4) a Placement Debt Rectification Engine (PDRE) that employs strategic live migrations to amortize accumulated debt offline. Extensive trace-driven simulations and a large-scale production deployment at Alibaba Cloud demonstrate that DVLA consistently outperforms state-of-the-art methods, achieving an additional 0.6 percentage points in packing density. This translates to saving thousands of machines in production, delivering substantial cost reductions.

1 Introduction

In large-scale cloud infrastructure, the physical footprint required to host VMs is governed by Packing Density (PD, Eq. 1). Because infrastructure capital and operational costs scale directly with the physical machine count, even a one percentage point (pp) increase in PD can translate into hundreds of millions of dollars in annual savings [8]. Production schedulers at major cloud providers [5, 13, 14, 46] therefore strive to maximize PD by casting VM placement as a large-scale dynamic bin-packing problem, and a leading approach leverages VM lifetime predictions to guide placement and improve utilization [8, 16, 33, 42]. Promising as they are, we find that these lifetime-aware schedulers break down in large-scale, real-world deployments.

Through a systematic analysis of production workloads, we identify two fundamental issues that undermine existing methods. First, their static policies are brittle. Lifetime-aware schedulers rely on fixed, hard-coded lifetime categories, yet VM lifetime distributions exhibit significant *spatial and temporal drift* across clusters and over time, rendering any fixed strategy ineffective. Second, and more insidiously, their placement strategies accumulate what we term *long-lived VM placement debt*—a persistent, cluster-wide inefficiency that directly inflates the number of machines a cluster must maintain to serve the same workload. As we formalize in Section 2.4, a suboptimal long-lived VM packing density (PD_L) translates into L_{debt} additional machines that could otherwise be reclaimed (Eq. 3), a concrete and measurable cost.

This debt arises from a flawed placement philosophy. Strategies such as LAVA [33], which co-locate short-lived VMs with long-lived ones, optimize for single-machine packing at the expense of scattering long-lived VMs across the cluster. Compounded by inevitable lifetime-prediction errors, this dispersion strands many machines for extended periods: our production data show that even a single suboptimally placed long-lived VM dramatically prolongs machine occupancy and reduces recycling frequency (Figure 4a). The result is fragmentation that cripples cluster-wide reclamation and accumulates a persistent debt that online scheduling alone cannot resolve.

To address these challenges, we present *Dynamic VM Lifetime Aware scheduling (DVLA)*, an end-to-end system that synergistically combines online and offline scheduling. DVLA includes four key components. To provide the foundational lifetime estimates, (1) a Hierarchical Lifetime Prediction

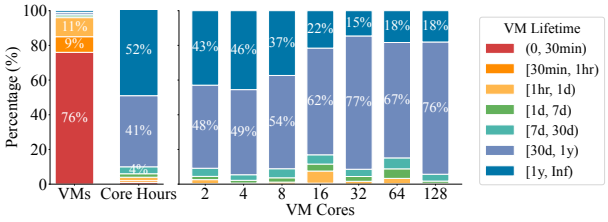
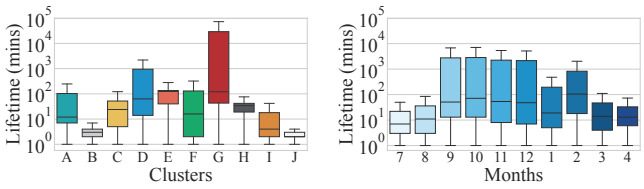


Figure 1: VM lifetime distribution. Left: Request volume vs. core-hours consumption. Right: Lifetime distribution by cores. Long-lived VMs dominate resource footprint.



(a) Cross-cluster heterogeneity. (b) Intra-cluster temporal drift.

Figure 2: The spatial and temporal volatility of VM lifetimes.

Model delivers multi-horizon predictions informing both online placement decisions and offline optimization. To counter distribution drifts, (2) a Dynamic Affinity Grouping strategy with a lightweight change-point detector adapts scheduling policies in real time. To mitigate placement debt at the source, (3) a Debt-Aware Placement Policy (DAPP) guides the online scheduler to proactively consolidate long-lived VMs. Finally, to handle the unavoidable accumulated debt, (4) a Placement Debt Rectification Engine (PDRE) employs strategic live migrations to amortize this debt offline.

In summary, the primary contributions of this paper are:

- We perform the first systematic study that identifies and quantifies the dual challenges of distribution drifts and long-lived VM placement debt, revealing the fundamental limitations of static, online-only schedulers.
- We design and implement DVLA, an end-to-end system that uniquely synergizes a dynamic online strategy to adapt to distribution drifts and prevent debt creation with a targeted offline mechanism to amortize placement debt.
- We demonstrate DVLA's effectiveness through both extensive trace-driven simulations and a large-scale production deployment at Alibaba Cloud, confirming substantial resource savings with manageable overhead.

2 Background

2.1 The Nature of VM Lifetimes in Cloud

Analysis of our production environment, spanning millions of daily allocations, reveals a significant long-tail distribution in VM lifetimes. As shown in the left panel of Figure 1, while short-lived VMs (<1 day) account for 96% of total requests, they contribute less than 2% to total core-hours. In contrast, long-lived VMs (>1 month) dominate resource consumption, representing 93% of core-hours despite comprising only 2.5% of requests. The right panel further breaks down this lifetime distribution by VM cores using data from active production VMs, revealing that long-lived VMs consistently command the majority of resources across all configurations. Notably, this concentration is even more pronounced in smaller VMs (e.g., $\leq$8 cores). These findings underscore that optimizing for long-lived VMs is critical for resource efficiency.

In our infrastructure, a zone comprises multiple clusters. Within a cluster, machines are homogeneous, yet significant heterogeneity exists across clusters within the same zone. Consequently, VM lifetime distributions across clusters are neither uniform nor static. As Figure 2a demonstrates, lifetime profiles exhibit significant spatial heterogeneity across clusters at a single point in time. In some clusters (e.g., B, J), the vast majority of VMs are short-lived, while in others (e.g., E, G), they are predominantly long-running. Furthermore, Figure 2b illustrates substantial temporal drift within individual clusters over months. This spatial and temporal volatility renders static scheduling policies suboptimal, a critical challenge we quantify in Section 3.

2.2 Online and Offline Schedulers

Online Scheduler. The online scheduler handles VM placement when a new VM is being created. Following a common *filter-and-score* paradigm [8, 24, 25, 53], it first filters for candidate machines that meet a VM's resource and constraint requirements. It then scores these candidates to optimize for objectives like packing efficiency and service stability. Strict latency requirements force online decisions to be fast and greedy, causing resource fragmentation over time.

Offline Scheduler. The offline scheduler complements its online counterpart by periodically correcting the fragmentation and suboptimal placements that inevitably accumulate. By performing global analysis and orchestrating live migrations, it consolidates target VMs to reclaim underutilized machines. However, this process is slow and resource-intensive, making it an insufficient remedy on its own for the relentless accumulation of placement errors. This fundamental limitation motivates our synergistic design, which combines proactive online prevention with reactive offline correction.

2.3 VM Lifetime Prediction in Scheduling

Schedulers leverage VM lifetime information to improve resource utilization and accelerate machine reclamation, employing strategies that fall into two main categories. Homogeneous placement, exemplified by LA [8] and NILAS [33],

aims to co-locate VMs with similar predicted lifetimes for synchronized machine reclamation. Heterogeneous placement, adopted by LAVA [33], intentionally co-locates short-lived VMs with long-lived ones to fill resource fragments without further extending the machine's reclamation timeline.

The effective implementation of any lifetime-aware strategy hinges on accurate prediction. This requires two models to suit different operational needs: (1) a low-latency *Initial Lifetime Prediction* at VM creation for the online scheduler, using static request features; and (2) a more accurate *Remaining Lifetime Prediction* for running VMs, which leverages rich runtime data. To manage the inherent uncertainty in lifetime prediction, prior works employ two methodologies: regression and classification. Regression models [31, 33], while predicting exact durations, exhibit high variance in long-tailed environments. They struggle to maintain precision across diverse lifetime ranges, and their sensitivity to extreme values leads to significant numerical deviations, ultimately causing poor co-location and increased resource fragmentation.

In contrast, classification models [8, 10, 16] map lifetimes into discrete categories (e.g., [1d, 7d)) and are more robust. By quantizing lifetime uncertainty into limited buckets, classification mitigates the impact of extreme-value errors. This approach trades overly fine-grained estimates for robustness, ensuring that scheduling decisions remain resilient to the inherent noise and volatility of cloud workloads.

2.4 Metric Definitions

To quantitatively assess the impact of VM placement strategies on cluster-wide resource efficiency, we define three foundational metrics. These metrics enable us to differentiate between overall packing density and the infrastructure overhead incurred by dispersion of long-lived VMs.

Packing Density (PD). A standard metric for overall packing efficiency in [8], defined as the ratio of total allocated cores to the total cores on all non-empty machines, computed as:

$$PD = \frac{\sum_{i=1}^{V} C_{vm,i}}{\sum_{j=1}^{M} C_{pm,j}} \tag{1}$$

where V is the number of VMs, M is the number of non-empty machines, $C_{vm,i}$ is the number of cores for VM i, $C_{pm,j}$ is the number of cores for physical machine j.

Long-Lived VM Packing Density (PD_L). Since long-lived VMs dominate core-hour consumption, we isolate their packing efficiency using PD_L:

$$PD_L = \frac{\sum_{i=1}^{V_{long}} C_{vm,i}}{\sum_{j=1}^{M_{long}} C_{pm,j}} \tag{2}$$

where V_{long} is the number of long-lived VMs, M_{long} is the number of machines hosting at least one long-lived VM. This metric specifically targets the consolidation of high-value, long-lived VMs, making a higher PD_L an indicator of more effective debt reduction and machine reclamation potential.

Long-lived VM Placement Debt (L_{debt}). We quantify the infrastructure overhead induced by long-lived VM dispersion as *Long-lived Placement Debt* (L_{debt}). It represents the number of machines that could be reclaimed if the current PD_L were improved to a target operational density ρ_{target}, which represents the achievable packing limit in production. We define L_{debt} as:

$$L_{debt} \approx M_{long} - \frac{\sum_{i=1}^{V_{long}} C_{vm,i}}{\frac{\sum_{j=1}^{M_{long}} C_{pm,j}}{M_{long}} \cdot \rho_{target}} = M_{long} \cdot \left(1 - \frac{PD_L}{\rho_{target}}\right) \tag{3}$$

L_{debt} bridges the gap between long-lived VM placement quality and machine wastage. A sub-optimal PD_L translates directly into a positive L_{debt}, forcing the cluster to maintain an unnecessarily large M_{long}. While L_{debt} quantifies the theoretical infrastructure overhead at the cluster level, *Stranded Machines* provide an operational measure at the individual machine level. Specifically, a machine is considered stranded when it hosts one or more long-lived VMs, but their core utilization remains below 50%. In the following section, we use Stranded Machines as the measurable manifestation of placement debt to evaluate standalone and synergistic strategies.

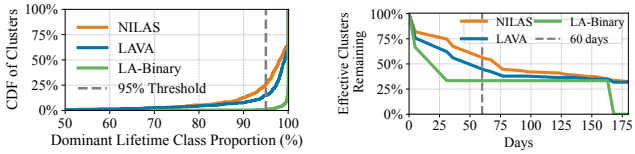
3 Motivation

Achieving high resource efficiency through lifetime-aware scheduling is deceptively complex. State-of-the-art schedulers are undermined by two fundamental challenges in real-world cloud environments: the brittleness of static policies against distribution drifts, and a flawed placement philosophy that creates compounding, long-lived VM placement debt.

3.1 Static Policies are Brittle

Prevailing lifetime-aware schedulers, such as LA-Binary [8], NILAS [33], and LAVA [33], rely on rigid, hard-coded categorization. LA-Binary uses {≤1h,>1h}, NILAS uses a 12-interval grid {0m,...,168h}, and LAVA uses {<1h,1-10h,10-100h,100-1000h}. These methods apply fixed categories uniformly across all clusters, ignoring the significant spatial and temporal drifts in VM lifetime.

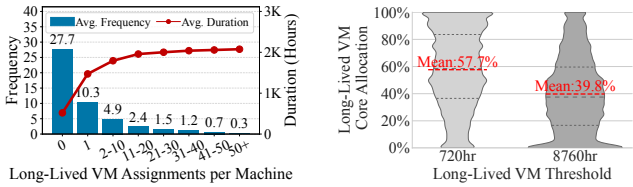
Ineffectiveness Against Spatial Diversity. To evaluate these categorization policies, we define a policy as *applicable* to a cluster if the VM lifetime distribution provides sufficient diversity (i.e., no single lifetime category contains ≥95% of the VMs). When a policy concentrates most of the VMs into a single category, it loses the ability to differentiate workloads, collapsing into random placement. Figure 3a reveals that these policies are inapplicable: for LA-Binary, nearly all clusters fail this diversity criterion; for NILAS and LAVA,



(a) High homogeneity (>95% in one category) renders static policies ineffective across clusters.

(b) Temporal decay showing that policy applicability drops as lifetime distributions drift over time.

Figure 3: Static categorization is ineffective against the spatial diversity and temporal dynamics of real-world workloads.



(a) Machine recycle frequency and occupied duration relative to the number of long-lived VMs.

(b) Per-Machine core allocation of long-lived VMs, revealing suboptimal consolidation.

Figure 4: Long-lived VMs drive machine occupancy and remain poorly consolidated.

85% and 74% of clusters, respectively, exhibit this lack of differentiation.

Decay Under Temporal Drifts. Even if a policy is initially applicable, it inevitably degrades as VM lifetime distributions evolve. Figure 3b tracks the fraction of clusters where policies remain applicable. As VM lifetime distributions drift, the applicability of NILAS and LAVA drops to 55% and 44% within 60 days, while LA-Binary's applicability collapses even faster.

This spatial and temporal volatility demonstrates that static categorizations are fundamentally inadequate for real-world workloads, motivating our DVLA design: a dynamic grouping mechanism that continuously adapts to lifetime distribution drifts within each cluster.

3.2 Long-Lived VM Placement Debt

Beyond the brittleness of static policies, a more insidious problem undermines state-of-the-art schedulers: their placement philosophies inevitably create a compounding, cluster-wide inefficiency we term long-lived VM placement debt. This debt stems directly from a flawed understanding of how to manage the most resource-dominant long-lived VMs.

The Flawed Philosophies of Prior Work. Existing lifetime-aware strategies fall into two camps, both of which inadvertently create placement debt. On one side, schedulers like LA [8] and NILAS [33] champion homogeneous placement, aiming to co-locate VMs with similar lifetimes. However, their design exhibits a narrow definition of long-lived. By focusing their finest-grained categories on VMs lasting up to only three or seven days, they overlook the truly resource-dominant VMs that persist for weeks or even months. Consequently, these schedulers fail to prioritize the aggregation of these critical long-lived VMs, leading to their gradual dispersion and the accumulation of placement debt.

On the other side, schedulers like LAVA [33] advocate for the opposite: heterogeneous placement. They intentionally co-locate short-lived VMs with long-lived ones to maximize single-machine packing density. While this provides local packing benefits, the strategy is counterproductive at the cluster level. It systematically scatters long-lived VMs across the fleet, directly creating placement debt, crippling global resource reclamation efforts.

The Enormous Cost of Placement Debt. Long-lived VMs dictate machine reclamation timelines because they dominate long-term resource consumption (Figure 1). Figure 4a shows that as the number of allocated long-lived VMs increases, machines experience longer occupancy and fewer recycles. The drastic gap between 0 and 1 allocation identifies the first long-lived VM as the primary driver of machine pinning, with diminishing marginal impact thereafter. This confirms that consolidating long-lived VMs effectively limits their footprint, maximizing cluster-wide reclamation.

Despite the necessity for consolidation, Figure 4b exposes a significant inefficiency in current production placement. We analyzed the core allocation of VMs with remaining lifetimes exceeding 720h (1 month) and 8760h (1 year). Their average utilization remains strikingly low at 57.7% and 39.8%, respectively. This reveals that long-lived VMs are rarely densely co-located in practice. Such poor consolidation creates a substantial opportunity for smarter, lifetime-aware strategies to improve cluster-wide resource utilization.

The Inevitable Accumulation of Debt. This placement debt, originating from flawed scheduling philosophies, is further compounded by an unavoidable operational reality: imperfect predictions. Even with a highly accurate model, prediction errors guarantee a continuous stream of misplacements.

For instance, consider a modest 5% error rate of misclassifying a long-lived VM as short-lived. Assuming independent errors, placing just 14 such presumed-short VMs onto a single machine yields an over-50% probability that the machine hosts at least one true long-lived VM, blocking its timely reclamation. When scaled across millions of daily allocations, this compounding effect ensures a relentless increase in the placement debt of the cluster.

3.3 Limitations of Standalone Strategies

Given the persistent accumulation of placement debt, a natural question arises: can this debt be managed by employing either aggressive offline correction or refined online prevention?

We argue that both standalone approaches are fundamentally insufficient, making a synergistic strategy essential.

As illustrated in Figure 5, we evaluate three placement strategies via trace-driven simulations on empty clusters using production workloads. *Online-Only* enforces lifetime affinity during initial placement for long-lived VMs, incorporating a 5% prediction error rate that occasionally disrupts affinity by placing long-lived VMs on non-affine machines. In contrast, *Offline-Only* disregards affinity during initial allocation, relying instead on limited daily offline rebalancing to consolidate long-lived VMs. Finally, *Online+Offline* hybridizes these approaches by combining online lifetime-aware placement with periodic offline consolidation.

Why Offline-Only Correction Fails. Relying solely on offline migration to repay placement debt is a losing battle for two reasons. First, the operational scope is heavily constrained. Our production analysis (Table 1) reveals that 40% of VMs are structurally non-migratable due to hardware dependencies, workload constraints, or policy restrictions. Second, even for the migratable subset, the rate of new debt creation far outpaces feasible correction capacity. As shown in Figure 5, an offline-only strategy, even with more migration budget, fails to curb the growth of debt.

Why Online-Only Prevention is Insufficient. Conversely, a purely proactive, online-only strategy can slow the creation of debt but cannot eliminate it. Inevitable prediction errors will always introduce new misplacements. Furthermore, an online scheduler makes greedy, largely irrevocable decisions and is powerless to fix the debt that has already accumulated from past mistakes. Figure 5 quantifies this limitation: under an online-only policy, the number of misplaced long-lived VMs grows unabated, demonstrating that prevention alone cannot solve a cumulative problem.

This analysis leads to a crucial insight: an effective system must address placement debt on two fronts simultaneously. It requires an online scheduler that minimizes debt creation at the source, coupled with a targeted offline mechanism that strategically amortizes the accumulated debt. This synergistic philosophy is the core motivation for DVLA's design.

Table 1: Breakdown of VM migration constraints.

Constraint Type	Proportion (%)
Migratable	60
Workload Constraints	20
Hardware Dependencies	14
Policy & SLA Restrictions	5
Other	1

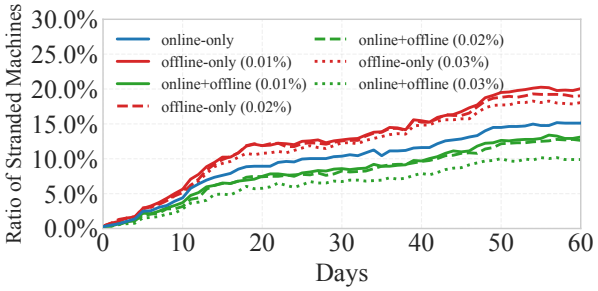


Figure 5: Proportion of stranded machines under three allocation strategies, varying by migration budget (ratio of migrated to total VMs).

4 DVLA Design

Section 3 demonstrated that static lifetime-aware policies are brittle against distribution drifts and inadvertently create long-lived VM placement debt. To overcome these challenges, we designed DVLA, a system that synergistically combines online prevention with offline correction. This section details DVLA's core design components: (1) A Hierarchical Lifetime Prediction Model delivering multi-horizon predictions for both initial placement and offline optimization (Section 4.1); (2) A Dynamic Affinity Grouping strategy that detects and adapts to distribution drifts in real-time (Section 4.2); (3) A Debt-Aware Placement Policy (DAPP) for the online scheduler to proactively consolidate long-lived VMs and prevent debt creation (Section 4.3); (4) A Placement Debt Rectification Engine (PDRE) that employs live migrations to amortize accumulated debt (Section 4.4).

4.1 VM Lifetime Prediction Model

Accurate lifetime prediction is the cornerstone of DVLA. To address the rigidity of static categorization, we implement a two-tiered, extensible prediction framework that decouples lifetime threshold definition from scheduling policy.

Extensible OvR Architecture. We employ a One-vs-Rest (OvR) binary classification design where each lifetime threshold is handled by an independent model. This modularity allows us to refine or extend granularity (e.g., a 15-minute granularity for bursty microservices) by training specific binary classifiers without retraining the entire system. While we utilize the set [0–1hr, 1hr–1d, 1d–7d, 7d–30d, 30d-1y, >1y] in production, these remain environment-specific parameters rather than hard-coded constraints.

Hierarchical Prediction Architecture. The system consists of two specialized models: (1) Initial Lifetime Prediction, which provides low-latency inference at VM creation using static request features; and (2) Remaining Lifetime Prediction, which executes offline to provide more accurate estimates for running VMs by incorporating rich dynamic runtime metrics.

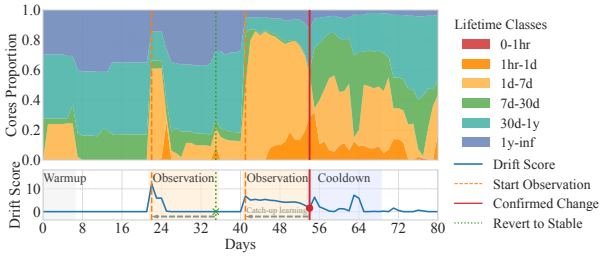


Figure 6: An illustrative dynamic affinity grouping process handles workload changes by first dismissing a transient spike (day 22) as noise, then correctly identifying and adapting to a persistent shift (day 41) by triggering a policy update.

As detailed in Appendix A, our feature importance analysis confirms that while the Initial prediction model captures application semantics, the Remaining prediction model extracts essential insights from survival history, providing a robust signal for our subsequent Dynamic Affinity Grouping mechanism.

4.2 Dynamic Affinity Grouping

As demonstrated in Section 3.1, any static lifetime categorization policy is brittle against the spatial and temporal drifts of real-world workloads. To overcome this, DVLA introduces a dynamic affinity grouping strategy. This mechanism continuously monitors cluster-wide workload to dynamically recalibrate affinity groups, ensuring that scheduling decisions remain aligned with the distribution of VM lifetimes. The entire process is detailed in Algorithm 1. The strategy operates as a robust daily cycle driven by state-machines, distinguishing persistent structural shifts from transient workload noise to prevent policy instability. Figure 6 presents an illustrative example of the mechanism operating on a real-world trace. The key stages of this mechanism are detailed below.

Workload Vectorization and Drift Scoring. Each day, we compute a cores distribution vector for the cluster, representing the proportion of total CPU cores consumed by running VMs in each global lifetime class. We monitor running VMs, not just incoming requests, as this reflects the actual resource landscape the scheduler must manage. This time series of vectors is fed into a lightweight change-point detector. The detector maintains an Exponentially Weighted Moving Average (EWMA) of historical vectors as a dynamic baseline of the cluster's normal state. A daily drift score is calculated as the cosine distance between the current day's vector and this baseline (line 7), flagging significant deviations in the distribution's shape. We use cosine distance as it is effective at capturing changes in the shape of the distribution, independent of the overall core volume.

State-Machine-Based Drift Confirmation. To distinguish persistent shifts from transient noise, a finite state machine (FSM) interprets the stream of drift scores. The FSM transitions between *Warmup, Stable, Observation*, and *Cooldown* states. When the policy is first applied to a new cluster, it enters a *Warmup* phase for an initial period. During this phase, the system exclusively collects data to establish the initial EWMA baseline (lines 9-12), without triggering any alarms. Once initialized, the system transitions to the *Stable* state (line 12), where it continuously updates the baseline with new daily vectors as long as the drift score remains below a predefined alarm threshold θ_{alarm} (lines 13-15).

A high drift score moves the system from *Stable* to *Observation* (lines 15-17), where it collects data for a fixed window W_{obs} (line 19) without updating the baseline. If the drift is confirmed as persistent, a policy update is triggered (lines 23-26). Otherwise, the event is dismissed as noise, and the system performs catch-up learning by retrospectively incorporating the data from the observation window into the baseline, ensuring that it accurately reflects the new normal (line 30).

Adaptive Affinity Group Generation. Upon a confirmed change, a new affinity group $G_{affinity}$ is generated (line 24). The algorithm sorts the global lifetime classes in descending order of their average core consumption during the observation window and greedily selects the top-ranked classes until their cumulative consumption exceeds a 95% threshold. For instance, if classes representing 0-1hr, 1d-7d, and 30d-1y are the most resource-intensive, the new affinity group becomes $\{$0-1hr, 1d-7d, 30d-1y$\}$. After an update, the system enters a *Cooldown* period to prevent oscillatory changes (line 27).

This mechanism counters policy decay by keeping affinity groups aligned with workload dynamics. While Algorithm 1 provides the overall control flow, the detailed formulations for its key functions are consolidated in Appendix B.

4.3 Proactive Debt Mitigation with DAPP

While Dynamic Affinity Grouping (Section 4.2) determines which lifetime categories are significant, it does not specify how to use them to prevent placement debt. To address this, we introduce the Debt-Aware Placement Policy (DAPP), a two-part mechanism designed to proactively mitigate debt at the source. DAPP advances the homogeneous placement principle [8, 33] by first, robustly classifying machines using a core-weighted method that is resilient to existing fragmentation, and second, applying an asymmetric affinity score that heavily penalizes debt-creating placements. We detail these two components below.

Robust Machine Lifetime Classification. To determine a machine's lifetime category, we first need to represent VM lifetimes numerically. Our system predefines a set of fine-grained lifetime classes (e.g., 0-1h, 1h-1d, 1d-7d, etc.), which are then mapped to ascending integer indices (e.g., 1, 2, 3, ...). This mapping provides a standardized numerical representation, L(vm), for a VM's lifetime class, which is essential for

Algorithm 1 Workload Drift Detection and Policy Adaptation

Input: Workload trace $T = \{v_1, v_2, ..., v_N\}$, Config C
Initialize:
  $State \leftarrow$ WARMUP
  $Detector \leftarrow$ DriftDetector(C)
  $t_{obs_start} \leftarrow -1$, $t_{update} \leftarrow -1$
for each day t with workload vector v_t in T **do**
  $score_t \leftarrow Detector$.Score($v_t$)
  switch $State$ **do**
    case WARMUP:
      $Detector$.Learn(v_t)
      if $t \geq C.P_{\text{warmup}}$ **then**
        $State \leftarrow$ STABLE
    case STABLE:
      $Detector$.Learn(v_t)
      if $score_t > C.\theta_{alarm}$ **then**
        $State \leftarrow$ OBSERVATION
        $t_{obs_start} \leftarrow t$
    case OBSERVATION:
      if $t - t_{obs_start} + 1 \geq C.W_{obs}$ **then**
        $V_{obs} \leftarrow$ GetVectorsInWindow(T, t_{obs_start}, t)
        $S_{obs} \leftarrow \{$Detector.Score(v) $\mid v \in V_{obs}\}$
        avg_score $\leftarrow$ Mean(S_{obs})
      if avg_score $\geq C.\theta_{confirm}$ **then**
        $G_{affinity} \leftarrow$ GenerateAffinityGroup(V_{obs}, C)
        UpdateSystemPolicy($G_{affinity}$)
        $t_{update} \leftarrow t$
        $Detector$.CatchUpLearn(V_{obs})
        $State \leftarrow$ COOLDOWN
      else
        $State \leftarrow$ STABLE
        $Detector$.CatchUpLearn(V_{obs})
    case COOLDOWN:
      if $t \geq t_{update} + C.P_{cooldown}$ **then**
        $State \leftarrow$ STABLE
  end switch

the calculations in both Eq. 4 and Algorithm 2.

A common approach in prior work is to use the lifetime of the longest-running VM (a MAX rule) to define a machine's category. However, our analysis revealed this is brittle in production clusters where long-lived VMs are already scattered. In such scenarios, where many machines host at least one long-lived VM (as suggested by Figure 4a), a MAX rule would classify nearly all of them as long-lived, rendering any subsequent affinity scoring ineffective. To address this, we compute a more robust, core-weighted average lifetime index for the machine, $L_{raw}(pm)$, as defined in Eq. 4. This method is less sensitive to the presence of a single long-lived VM and better reflects the machine's overall lifetime character.

$$L_{raw}(pm) = \sum_{i=1}^{n} \left(\frac{C_{vm,i}}{\sum_{j=1}^{n} C_{vm,j}} \times L(vm_i) \right) \tag{4}$$

Algorithm 2 Machine Lifetime Category Calculation

Input: A machine pm; the affinity group $G_{affinity}$
Initialize:
  Calculate $L_{raw}(pm)$ using Eq. (4)
  $min_dist \leftarrow \infty$
  $L_{pm} \leftarrow -1$
Output: The machine's lifetime category L_{pm}
for each lifetime category c in $G_{affinity}$ **do**
  $dist \leftarrow |c - L_{raw}(pm)|$
  if $dist < min_dist$ **then**
    $min_dist \leftarrow dist$
    $L_{pm} \leftarrow c$
  else if $dist = min_dist$ **then**
    $L_{pm} \leftarrow \max(L_{pm}, c)$
return L_{pm}

where vm_i denotes the i-th VM among n VMs running on the machine, $L(vm_i)$ is the numerical index of the lifetime class for vm_i, $C_{vm,i}$ represents the core allocation of vm_i.

This continuous L_{raw} value is then mapped to a discrete category. Crucially, the set of possible categories for a machine is not globally fixed, it is constrained by the cluster's current affinity group $G_{affinity}$, as determined by Dynamic Affinity Grouping (Section 4.2). This ensures a machine's category is always relevant to the prevailing workload. Algorithm 2 performs this mapping by finding the category index c within $G_{affinity}$ that has the minimum distance to L_{raw}. In case of a tie, we conservatively select the larger category index, ensuring a machine on the boundary is treated as the longer-lived one to further mitigate placement debt risk.

Asymmetric Affinity Scoring. With the VM and machine categories determined, DAPP's affinity score calculates their placement compatibility. The score is defined in Eq. 5 and is designed to aggressively penalize debt-creating placements while remaining flexible for others.

$$\text{Score}(vm, pm) = \begin{cases} 1, & \text{if } L_{vm} = L_{pm} \\ \dfrac{K - |L_{vm} - L_{pm}|}{K}, & \text{if } L_{vm} < L_{pm} \\ \dfrac{K - |L_{vm} - L_{pm}|}{K} \times \omega, & \text{if } L_{vm} > L_{pm} \end{cases} \tag{5}$$

where L_{vm} and L_{pm} are the lifetime category indices for the VM and machine. K is the total number of global lifetime classes. The key lies in the adaptive penalty factor ω, which is applied only when placing a longer-lived VM onto a shorter-lived machine ($L_{vm} > L_{pm}$).

The core mechanism is an asymmetric penalty designed to prevent placement debt. It exclusively penalizes placing a longer-lived VM on a shorter-lived machine. The penalty is twofold: a standard one (e.g.,$\omega = 0.8$) for general mismatches, and a harsher one (e.g.,$\omega = 0.2$) for placing a VM with a life-

time over one month. This severe penalty effectively blocks debt-creating placements and forces the consolidation of these critical VMs.

4.4 Amortizing Placement Debt with PDRE

While DAPP proactively minimizes debt creation, prediction errors make some accumulation of placement debt inevitable. To address this residual debt, we designed the Placement Debt Rectification Engine (PDRE), a targeted offline mechanism. Unlike generic defragmenters, PDRE is not focused on overall cluster packing. Instead, it is focused on amortizing the most costly placement debt. PDRE operates in a three-stage process: identifying targets, prioritizing migrations, and executing placements under strict constraints.

Identifying Stranded VMs. PDRE's first task is to locate *stranded VMs*, i.e., long-lived VMs that reside on stranded machines. To do this with high confidence, it leverages the more accurate Remaining Lifetime Prediction model, allowing PDRE to identify VMs with over one month of expected remaining lifetime. Crucially, non-migratable long-lived VMs are excluded from this set.

Migration Prioritization. To maximize the return on a limited migration budget, PDRE prioritizes candidates using a two-tiered strategy. It first targets machines that are easiest to reclaim (i.e., those with the fewest *stranded* VMs). Machines hosting any non-migratable long-lived VM are strictly excluded from consideration. Within this group, it then prioritizes migrating the VMs with the greatest remaining lifetime. This deterministic sorting ensures that each migration delivers the maximum impact towards machine reclamation.

Constrained Placement Execution. The prioritized list of VMs is not migrated blindly. Instead, it is passed to a unified offline scheduler that finds optimal destinations to colocate them with other long-lived VMs. The entire process is governed by production-safe constraints, including migration budgets to limit system disruption and net-positive gain requirements to ensure every action is beneficial. By strategically amortizing placement debt, PDRE acts as the indispensable offline counterpart to our proactive online scheduler, completing DVLA's synergistic approach.

5 System Architecture and Implementation

This section details the production-grade architecture of DVLA, bridging the design principles to a robust implementation. As shown in Figure 7, our system integrates three key components: (1) an online scheduling pipeline for real-time, debt-aware VM allocation; (2) a Placement Debt Rectification Engine (PDRE) for offline debt amortization; and (3) a set of core services for state management and policy adaptation. We will first detail the architecture (Section 5.1), then discuss the key engineering solutions for production deployment (Section 5.2) and operational monitoring (Section 5.3).

5.1 System Architecture

Online Scheduling for Proactive Debt Mitigation. The online scheduler is the first line of defense against placement debt. It handles real-time VM allocation requests with stringent latency constraints ($< 100\,ms$) and is engineered to proactively consolidate long-lived VMs.

Low-Latency Lifetime Prediction. When a scheduling request arrives, the control service injects an initial lifetime prediction. To meet the tight latency budget, we designed a multi-tiered prediction strategy. First, an in-memory prediction cache serves predictions for recurring request features (e.g., user ID, cores), achieving more than 90% hit rate. For a cache miss, the request is sent to a fleet of auto-scaling inference servers. This remote inference is designed to complete within a few tens of milliseconds of latency budget, including network overhead, preventing it from becoming the critical path.

DAPP-based VM Allocation. The allocation process is critically enhanced by DAPP. To compute the score efficiently, the lifetime category of the incoming VM is obtained from the prediction service, while the category of each candidate machine is pre-calculated and retrieved from a low-latency, distributed Machine Tag Cache. This design trades slightly stale machine state for low scoring latency. The state is kept fresh via the Tag Update Service (Section 5.2), which ensures that tag staleness rarely impacts placement quality.

Dynamic Affinity Grouping Engine. To combat the policy decay caused by lifetime distribution drifts (Section 3.1), DVLA incorporates a Dynamic Affinity Grouping Engine. This engine implements the adaptive policy mechanism detailed in Section 4.2 and Algorithm 1.

Orchestration. It operates as a daily scheduled task that analyzes the cores distribution vector of each cluster, using up-to-date data from the Remaining Lifetime Prediction model.

Stateful Drift Detection. The engine feeds this data into the change-point detector, which uses a state machine to distinguish persistent distributional shifts. The engine's state machine for each cluster is persisted to ensure robustness.

Policy Deployment. Upon confirming a persistent drift, the engine generates a new, optimized set of affinity groups for the cluster. This updated policy is then propagated to the online scheduling servers and the Tag Update Service, ensuring that subsequent placement decisions and machine state calculations are based on the most relevant lifetime categorization.

PDRE for Amortizing Accumulated Debt. The Placement Debt Rectification Engine (PDRE) is our offline component designed to amortize accumulated placement debt. It operates as a periodic task, leveraging the high-fidelity Remaining Lifetime Prediction model to identify and execute strategic live migrations. In our production implementation, PDRE

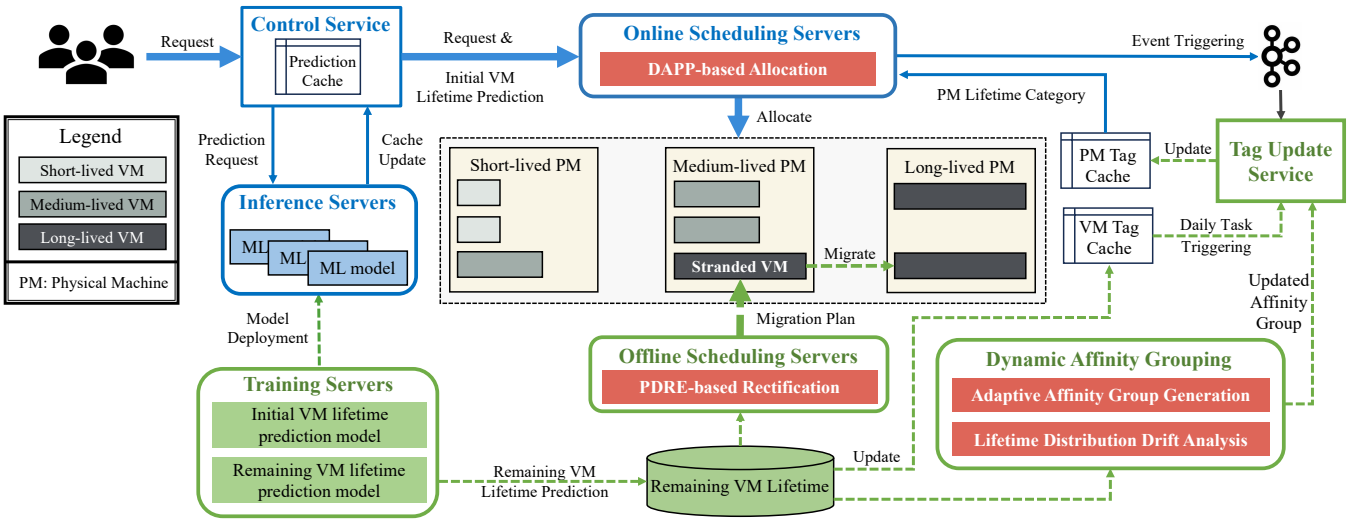


Figure 7: Overview of the online and offline scheduling architecture with DVLA.

is integrated with the cluster's maintenance workflow and focuses on two high-impact scenarios.

Consolidating Stranded Long-Lived VMs. PDRE scans for machines pinned by only a few migratable long-lived VMs. It then generates migration tasks to move these stranded VMs to existing long-lived consolidated machines, maximizing the potential for machine reclamation.

Efficient Machine Evacuation. When a machine is scheduled for maintenance, PDRE provides an optimized evacuation plan. It prioritizes migrating VMs with the longest remaining lifetimes first, while leaving short-lived VMs to terminate naturally. This strategy, similar to LARS [33], minimizes migration volume and accelerates the machine draining process.

Core State Management and Synchronization. The accuracy of DVLA's decisions hinges on up-to-date lifetime state, managed by a dedicated Tag Update Service. This service must provide fresh data without overwhelming the system, a challenge we address with a dual-mechanism design and performance optimizations.

Dual-Mechanism Update for Freshness. To balance accuracy and overhead, the service employs a dual update strategy. Event-triggered updates occur in real-time in response to critical events like VM allocation or release, ensuring immediate state consistency. Complementing this, a daily periodic task recalibrates the lifetime categories of all machines using the latest predictions from the Remaining Lifetime Prediction model. This accounts for the natural aging of VMs and corrects any potential state drift over time.

Selective Updates for Scalability. Naive event-triggered updates would be prohibitively expensive. We implement a selective policy that dramatically reduces update frequency, ensuring the scalability of state management (Section 5.2).

5.2 Addressing Deployment Challenges

To ensure DVLA's robust and efficient operation in a complex production environment, we designed and implemented several mechanisms to address key practical challenges.

ML Service Availability and Fault Tolerance. Online scheduling demands extremely low latency, yet the ML inference service can experience disruptions from network timeouts or node failures. To ensure system resilience, we implement a multi-tiered fallback strategy.

Primary Model. Under normal conditions, the system uses the live inference service for high-accuracy predictions.

Profile Matching. If the service is unavailable or times out, the system falls back to a lightweight profile-matching mechanism. This mechanism leverages a local cache of precomputed profiles, which map common VM request features (e.g., user ID, image type) to lifetime categories. Each profile is a tuple $(\mathbf{F}, c)$ where $\mathbf{F}$ is the feature vector $(f_1, f_2, ..., f_n)$ that are matched against incoming requests and c is the corresponding lifetime category. Our analysis shows that the profile-matching achieves 75% of the primary model's accuracy, which is significantly better than a random or static default, ensuring graceful degradation.

Safe Default. In the rare event that both the primary model and profile matching fail (e.g., for a new customer's first request), the system assigns a default lifetime category (e.g., 1m-1y). This default is conservatively chosen to align with our core principle of debt avoidance. Misclassifying a truly long-lived VM as short-lived creates significant placement debt, whereas the reverse error is far less costly. This *safe default* thus acts as a final guardrail against debt creation during system faults.

To combat concept drift over the long term, we have also established a model retraining pipeline that periodically re-evaluates and updates the classifiers. The frequency and over-

head of this process are detailed in Section 6.4.

Optimizing Machine Category Update Overhead. The machine's lifetime category is critical to DVLA. In a large-scale cluster with thousands of events per second, recomputing this tag upon every VM allocation or release would impose prohibitive overhead, risking system instability.

Our analysis revealed that the vast majority of these events do not actually alter a machine's lifetime category. We therefore designed a selective update policy that triggers recalculation only for three critical events: (1) the last VM on a machine is released; (2) the first VM is allocated to an empty machine; (3) the last long-lived VM (e.g., remaining lifetime more than one month) is migrated off a machine. By implementing this fine-grained strategy, we reduced the frequency of tag updates by an order of magnitude, significantly enhancing system stability and efficiency while maintaining the necessary accuracy of machine lifetime categories.

5.3 Operational Monitoring and Diagnostics

To ensure the long-term health and debuggability of DVLA in production, we have built a comprehensive monitoring dashboard that tracks key metrics across the entire system pipeline. This framework is structured around three stages.

Predictor Health and State Integrity. We assess the quality of the input signals, which includes monitoring the prediction accuracy and service performance of our ML models, tracking not only accuracy but also inference latency and cache hit rates. We also ensure label integrity by tracking the coverage rates of VM and machine lifetime categories.

Policy Dynamics and Enforcement. We evaluate the effectiveness of the core policy execution. This involves tracking the placement affinity achieved by the online scheduler, monitoring the adaptive policy dynamics by logging the frequency of change-point detections and affinity group updates, and measuring the volume and success rate of PDRE.

Consolidation Effectiveness. The ultimate measure of DVLA's success is its ability to reduce placement debt, which we quantify by tracking the packing density of long-lived VMs (formally defined in Section 2.4). This metric directly reflects the end-to-end impact of our DVLA.

This multi-faceted monitoring framework enables operators to perform root-cause analysis. For example, a decline in the consolidation of long-lived VMs can be traced back to poor placement affinity, which might be caused by prediction errors or outdated machine tags, allowing for targeted intervention.

6 Evaluation

6.1 Experimental Setup

Methodology and Workloads. We evaluate DVLA using a high-fidelity, event-driven simulator that precisely models our production scheduling environment. Crucially, the simulator shares its core scheduling logic with our production system, ensuring our findings are directly applicable to real-world scenarios. Our evaluation uses extensive operational traces collected over two months from 23 production clusters. These traces capture a diverse range of real-world workloads, varying in scale (from 200 to 2,200 machines), intensity (from 12k to 800k VM events), and VM lifetime distributions, providing a robust testbed for our system.

Baseline. We compare DVLA against three schedulers: (1) production baseline, our provider's non-lifetime-aware scheduler; (2) LA-Binary [8]; and (3) LAVA [33]. While NILAS [33] is also a relevant lifetime-aware scheduler, the paper reports LAVA's superior performance over NILAS, making LAVA a state-of-the-art baseline for our comparison.

LA-Binary Configuration. We implement the binary Lifetime Alignment algorithm with a 2-hour threshold to distinguish short and long classifications [8]. Machines are dynamically classified using a MAX rule, by the maximum predicted lifetime of resident VMs. Placement prioritizes machines of the same class using Best-Fit. If no suitable machine exists, the VM is placed on any available machine via Best-Fit, or a new machine is allocated.

LAVA Configuration. Following [33], we employ a regression model that leverages VM uptime to dynamically refine remaining lifetime estimates, thereby mitigating initial prediction errors. Both VMs and machines are mapped to four lifetime classes (<1h, 1-10h, 10-100h, 100-1000h). LAVA adopts a recycling strategy: machines exceeding 90% utilization exclusively accept VMs from strictly lower lifetime classes. Machine classifications are dynamically updated upon deadline violations or VM departures.

Prediction Model Fairness. To isolate the impact of scheduling algorithms from prediction quality, all schedulers utilize the same production-grade VM lifetime prediction models. For LAVA, which requires a regression model, we trained a dedicated predictor that achieves 99% precision and 90% recall on a 7-day lifetime threshold, significantly outperforming the 70% recall reported in the original LAVA paper. This ensures we are comparing DVLA against a powerful, state-of-the-art implementation.

Metrics. We assess performance using three key metrics that directly map to our design goals. Beyond the Packing Density and Long-lived VM Packing Density metrics defined in Section 2.4, we also evaluate the VRAR metric.

VM Risk Aggregation Ratio (VRAR). This metric quantifies the trade-off with service stability, defined as the fraction of a customer's VMs co-located on a machine beyond a soft threshold θ (Eq. 6). While our scheduler strictly enforces all explicit anti-affinity constraints, this internal metric helps us monitor fault isolation for customers without such explicit requirements. A modest increase in VRAR is acceptable if

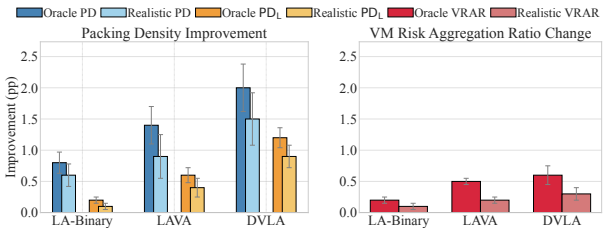


Figure 8: End-to-end performance comparison. DVLA significantly outperforms LA-Binary and LAVA in both overall PD and PD_L, while the VRAR change remains modest.

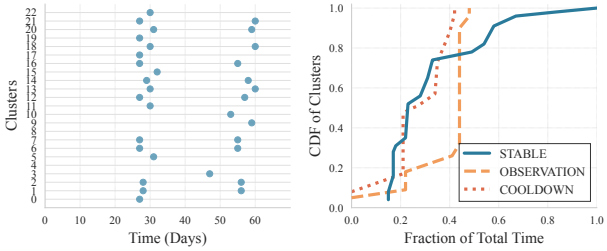


Figure 9: Effectiveness and Stability of Dynamic Policy Adaptation. (Left) Confirmed events (dots) identified via affinity grouping across 23 production clusters over 60 days. (Right) CDF of policy state durations.

accompanied by significant packing gains.

$$VRAR = \frac{\sum_{i=1}^{n} \sum_{u \in U} m_{i,u} \cdot \mathbb{I}(m_{i,u} > \theta)}{\sum_{i=1}^{n} \sum_{u \in U} m_{i,u}} \quad (6)$$

where n is the number of machines, U is the set of all customers, and $m_{i,u}$ represents the number of VMs belonging to customer u on machine i. The parameter θ is the predefined threshold. The indicator function $\mathbb{I}(m_{i,u} > \theta)$ equals 1 if $m_{i,u} > \theta$, and 0 otherwise.

6.2 End-to-End Performance

We evaluate the end-to-end performance of DVLA against the baseline and state-of-the-art schedulers. The results, averaged across 23 production clusters and weighted by size, are presented in Figure 8. We report performance under two settings: a *Realistic* scenario using our production prediction models and an *Oracle* scenario with perfect lifetime predictions to isolate the performance of the scheduling strategy itself.

Packing Density Improvement. Figure 8 demonstrates DVLA's superiority relative to the production baseline. In the Realistic setting, DVLA increases the overall packing density (PD) by 1.5 percentage points (pp), significantly outperforming LAVA (0.9 pp) and LA-Binary (0.6 pp). To understand the source of this improvement, we specifically analyze the packing of resource-dominant long-lived VMs. The results show that DVLA boosts the long-lived VM packing density (PDL) by 0.9 pp, an improvement more than double that of LAVA (0.4 pp) and 9x that of LA-Binary (0.1 pp). This finding confirms that the substantial overall gain is primarily driven by DVLA's effectiveness in consolidating long-lived VMs, directly validating our central hypothesis that tackling placement debt is key to maximizing cluster-level efficiency.

Crucially, DVLA with realistic predictions (1.5 pp) surpasses LAVA's oracle upper bound (1.4 pp), demonstrating that our algorithmic design outweighs the benefit of perfect lifetime information. This lead is consistent across the 23 diverse clusters, where DVLA delivered an additional 0.2 to 4.3 pp in packing density gain over LAVA. Importantly, the average 0.6 pp improvement over this strong baseline translates to saving thousands of machines at cloud scale, highlighting the substantial economic impact of our approach.

Trade-off with Service Stability. As shown in Figure 8, achieving higher density involves a trade-off with risk aggregation. DVLA increases the VRAR by 0.3 pp in the *Realistic* setting. This modest increase, only 0.1 pp higher than LAVA's, remains well within operational tolerance and is a small price to pay for the substantial efficiency gains. This demonstrates that DVLA strikes a favorable balance between maximizing resource savings and maintaining service stability.

The significant improvements observed in simulation motivated its deployment into our production environment. The real-world impact and costs are detailed in Section 6.4.

6.3 Analysis of DVLA's Key Designs

We now dissect DVLA's performance to validate the efficacy of its core design components.

The Efficacy of Dynamic Affinity Grouping. To validate that our dynamic approach overcomes the brittleness of static policies (Section 3.1), we analyzed its adaptation behavior across 23 production clusters over 60 days. Figure 9 confirms that adaptation is both necessary and stable.

The necessity is stark: 91% (21 of 23) of clusters exhibited workload drifts requiring at least one policy update. The CDF in Figure 9 reveals why: clusters spend a median of only 23.3% of time in STABLE, with OBSERVATION and COOLDOWN occupying 43.3% and 33.3%, respectively. This indicates that production workload distributions exhibit continuous drift, with clusters in non-STABLE phases for over 75% of the period.

Crucially, this pervasive drift does not induce instability. The update cadence remains consistent, with a median interval of 28 days and a tight interquartile range of 28–29 days. The substantial OBSERVATION time acts as an effective filter: despite continuous drift detection, most observation windows do not trigger confirmed updates (Figure 9). This demonstrates that our change-point detector effectively distinguishes persistent structural shifts from transient noise, preventing policy

thrashing and validating its robustness for production.

The Impact of Machine Lifetime Classification. As justified in our design (Section 4.3), we opted for a core-weighted average (WAVG) method for machine lifetime classification due to its superior robustness over a simple MAX rule in mature, fragmented clusters. We empirically compared the performance of WAVG against MAX under the Oracle setting. Figure 10 reveals a context-dependent trade-off. In existing populated clusters, the robust WAVG method achieves a 0.5 pp higher PD gain, as it is more resilient to the pinning effect of pre-existing scattered long-lived VMs. Conversely, in empty clusters where placements can be optimized from scratch, the more aggressive MAX rule yields a 0.2 pp higher PD gain by establishing dense long-lived machines early. This insight led to a hybrid, two-phase strategy in our production implementation. DVLA bootstraps empty clusters with the MAX method and automatically transitions to the more robust WAVG method after the first affinity group update, optimizing for both initial packing and long-term stability.

Quantifying Component Contributions via Ablation. To quantify the individual contribution of each component in DVLA's design, we conducted an ablation study. Figure 11 shows a waterfall chart illustrating the drop in PD gain as we sequentially remove key components from the full DVLA. All ablation experiments are conducted under the Oracle prediction setting to isolate the algorithmic contribution from prediction noise. Removing *Dynamic Affinity Grouping* causes the largest single drop (0.34 pp), confirming the high cost of using brittle, static policies. Removing *PDRE* as a whole results in a significant 0.31 pp loss, unequivocally proving the necessity of an offline mechanism to amortize the placement debt that online scheduling alone cannot fix. Within the dual-mechanism of state management, disabling the periodic *Tagging Task* (0.21 pp loss) is more detrimental than disabling reactive *Event-Triggered updates* (0.14 pp loss), highlighting the importance of periodic, global state recalibration over just real-time updates. This study validates our end-to-end design, demonstrating that each component provides a significant, measurable contribution. The synergy between proactive online debt prevention and reactive offline debt correction is what enables DVLA to achieve its superior performance.

Robustness and Parameter Tuning. To ensure DVLA operates robustly in production, we carefully selected its key hyperparameters. For the Dynamic Affinity Grouping mechanism, we performed a comprehensive sensitivity analysis on its three parameters: θ_{alarm}, W_{obs}, and $\theta_{confirm}$. Our analysis, detailed in Appendix C.1, evaluated numerous combinations against a cost function prioritizing system stability over hyper-responsiveness. The results clearly indicated that a conservative strategy (e.g., an observation window of 20 days with high confirmation thresholds) optimally balances adaptability with stability by effectively filtering transient noise. This data-driven approach led to the stable and effective policy

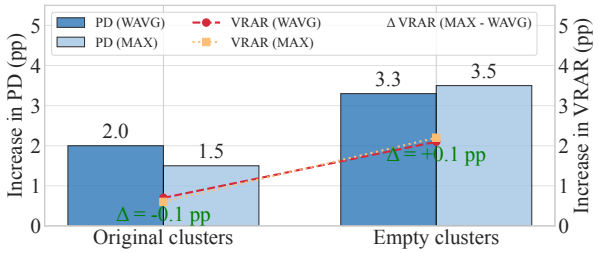


Figure 10: Performance of WAVG and MAX categorization strategies on PD and VRAR in original and empty clusters.

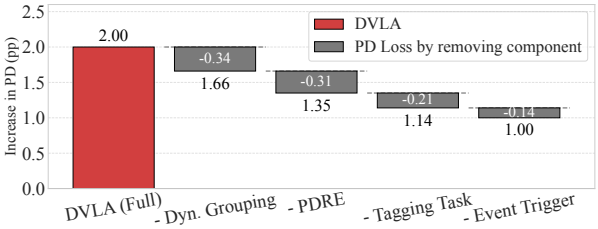


Figure 11: Ablation study on DVLA components. The waterfall chart shows the PD gain loss from removing each component sequentially, quantifying its contribution.

adaptation achieved by our mechanism (Figure 6).

Beyond parameter sensitivity, we also evaluated DVLA's robustness to the accuracy of lifetime predictions, a critical factor for real-world deployment. A detailed sensitivity analysis, presented in Appendix C.2, confirms DVLA's resilience. While its performance naturally scales with prediction accuracy, DVLA still delivers 1.02 pp packing density gain even when the recall for long-lived VMs drops to a low 40%. This demonstrates that DVLA is not brittle and can provide significant benefits even with imperfect prediction models.

6.4 Production Impact and Overhead Analysis

Beyond simulation, we deployed DVLA in Alibaba Cloud for over seven months. This section presents a comprehensive analysis of its real-world performance: (1) operational health and system overhead; (2) sustained end-to-end efficacy in improving packing density; and (3) enhanced robustness against fluctuations in workload demand.

Operational Monitoring. DVLA maintained high fidelity and coverage throughout the deployment. The remaining lifetime prediction model achieved 92% accuracy, with the initial lifetime model at 82%. This high accuracy enabled a near-total lifetime category coverage of 99% for VMs and 98% for machines. Consequently, over 98% of new VMs were placed on machines with a higher or equal lifetime category, effectively preventing long-lived VMs from occupying short-lived machine capacity. To maintain this high affinity, corrective ac-

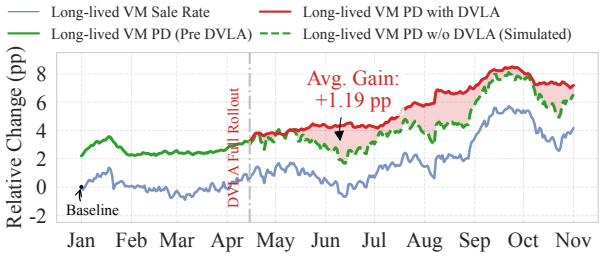


Figure 12: Comparison of long-lived VM packing density with and without DVLA, shown against the sale rate.

tions were opportunistically integrated into daily maintenance tasks, which migrated an average of 810 VMs per day. This process included correcting 39 misplaced long-lived VMs and yielded an average 0.3 pp PD_L increase per task.

System Overhead. The overhead introduced by DVLA is minimal and well-managed. Model training is an offline process, representing a trivial cost for a cloud provider managing hundreds of thousands of machines. The initial lifetime model trains weekly (40 mins on a 16-core, 64 GB instance), and the five remaining lifetime models train daily in parallel (1 hour on five 16-core, 32 GB instances). Since PDRE is piggybacked onto existing maintenance tasks rather than actively triggered, it incurs no extra scheduling overhead. The online scheduling path is also highly efficient, with a 90% cache hit rate for initial lifetime predictions, cache misses result in a negligible 99^{th} percentile inference latency of only 5 ms. The service has maintained 100% availability since its launch.

End-to-End Efficacy. As illustrated in Figure 12, DVLA delivers substantial and sustained improvements. After its full roll-out, the packing density of long-lived VMs consistently outperformed a baseline simulating performance without DVLA. To create this realistic counterfactual, we introduce the long-lived VM sale rate, a metric representing the underlying demand for long-lived resources. We define it as the ratio of total cores allocated by long-lived VMs to the total available core capacity in the cluster. By modeling the historical correlation between this sale rate and packing density from the pre-DVLA period, we project what the PD_L would have likely been without our system. Over the evaluation period, DVLA achieved an average PD_L gain of 1.19 pp against this demand-aware baseline. This significant efficiency gain was achieved with remarkable stability, the VRAR metric saw an average increase of only 0.28 pp, and no user complaints related to performance degradation were reported.

To quantify the impact of DVLA on mitigating placement debt, we analyzed the distribution of *stranded machines*. Figure 13 illustrates a reduction in stranded machines following the full rollout of DVLA, with the average proportion dropping from 25.3% to 21.3%. Notably, the subset of machines hosting only a single long-lived VM decreased from 4.3% to 3.5%. These results confirm that DVLA effectively mitigates

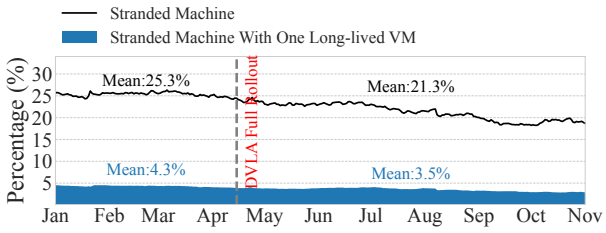


Figure 13: Proportion of stranded machines (out of all non-empty machines) before and after DVLA's full rollout.

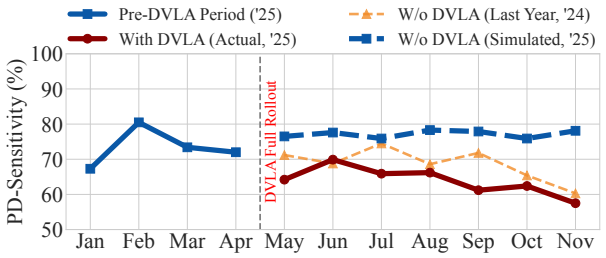


Figure 14: Impact of DVLA on cluster robustness, measured by PD-Sensitivity, where a lower value is better.

the pinning effect of long-lived VMs.

Robustness Against Demand Fluctuation. Beyond static efficiency gains, a critical measure of a scheduler's quality is its robustness to workload demand fluctuations. To quantify this, we propose a novel metric called *Packing Density Sensitivity (PD-Sensitivity)*. It measures how much packing density drops for every one percentage point drop in resource demand (i.e., sale rate). A lower PD-Sensitivity is highly desirable, as it signifies that the cluster's packing efficiency is less perturbed by drops in resource demand.

Formally, for a given time interval defined by a drop in demand, we first define the change in Packing Density (ΔPD) and the change in Sale Rate (ΔSR)

$$\Delta PD = PD_{before} - PD_{after} \tag{7}$$

$$\Delta SR = SR_{before} - SR_{after} \tag{8}$$

The PD-Sensitivity for that event is then the ratio:

$$PD - Sensitivity = \frac{\Delta PD}{\Delta SR}, \; where \; \Delta SR > 0 \tag{9}$$

In a well-consolidated cluster, a drop in demand (e.g., from termination of short-lived VMs) allows the scheduler to reclaim entire machines, preserving high PD on the remaining active machines. In a fragmented cluster, long-lived VMs are scattered, pinning down many machines. A drop in demand here leads to lower utilization across a wide footprint without enabling machine reclamation, causing a sharp decline in overall PD. PD-Sensitivity thus provides a dynamic measure

of the cluster's structural health, directly reflecting the quality of lifetime-aware affinity placement.

Figure 14 plots this metric over time. After DVLA's rollout, the actual PD-Sensitivity (blue line) established a new, more resilient operational baseline, consistently outperforming both the pre-DVLA period and the same period from the previous year (orange dashed line). Critically, the persistent gap between the actual performance and the simulated performance without DVLA baseline (blue dotted line) demonstrates DVLA's direct and continuous causal impact. On average, DVLA reduced PD-Sensitivity by 7.1 pp, showcasing its sustained effectiveness in a dynamic production environment.

7 Related Work

Online Scheduling. Online VM scheduling is a cluster scheduling problem [24, 30, 45, 49, 53]. There are algorithms based on resource allocation [2, 6, 9, 12, 15, 17, 22, 26, 40, 47, 56, 58], bin packing [3, 4, 7, 27, 32, 39, 48], and prediction of future requests [19, 31, 35, 36, 44]. [28] and [34, 43, 55] use SelfTune and LLM to optimize the system. There are also many studies using lifetime to optimize scheduling strategies [21, 51, 52, 54]. According to [16], long-lived VMs consume more than 95% of the total CPU workload, and those with similar lifespans can be grouped. [10] proposed to predict the lifetime and load of VMs in scheduling. [8] designed a VM lifetime-aware scheduling algorithm using a binary classification model. However, due to different payment types and VM lifetime distributions, these algorithms struggle to achieve the best bin-packing effect in our system.

Offline Scheduling. Offline VM rescheduling can improve the scheduling result of online VM allocation [1, 11, 18, 20, 23, 37, 38, 41, 50, 57]. It can use more complex methods to achieve better results than online VM scheduling. [23, 38] discussed the optimal packing rate that can be achieved when offline migration times are less than or more than VMs. [11] used divide-and-conquer idea, graph neural network, and combinatorial optimization solver in VM clustering to improve the customer experience. [50] proposed a VM migration algorithm based on the Markov decision process and the Q-learning algorithm. [18] rescheduled VM with reinforcement learning to achieve an effect close to mixed integer programming at a higher speed. However, no previous work has applied features related to lifetime prediction to VM rescheduling.

VM Lifetime Prediction. In recent years, many works have studied VM lifetime in cloud computing. [16, 42] pointed out that most VMs in public clouds are short-lived, but long-lived VMs take up most of the core hours. [8] considered VM features, customer features, and timing features, and used LightGBM [29] for binary classification prediction. The authors argue that it is difficult to use deep learning models to predict while ensuring millisecond-level scheduling delays.

8 Conclusion

State-of-the-art lifetime-aware schedulers are undermined by two critical, real-world challenges: the brittleness of static policies against drifting lifetime distributions and the insidious accumulation of long-lived VM placement debt. We presented DVLA, an end-to-end system that conquers these issues through a synergistic design combining proactive online prevention with reactive offline correction. Online, DVLA uses dynamic affinity grouping to adapt to workload drifts and a Debt-Aware Placement Policy (DAPP) to prevent debt at the source. Offline, a Placement Debt Rectification Engine (PDRE) employs strategic migrations to amortize the unavoidable residual debt. Our large-scale production deployment at Alibaba Cloud demonstrates DVLA's superiority, achieving an additional 0.6 percentage points higher packing density than state-of-the-art methods. This directly translates to saving thousands of physical machines and significantly enhances cluster robustness.

References

[1] Deafallah Alsadie, Zahir Tari, Eidah J Alzahrani, and Albert Y Zomaya. Life: A predictive approach for vm placement in cloud environments. In *2017 IEEE 16th International Symposium on Network Computing and Applications (NCA)*, pages 1–8. IEEE, 2017.

[2] Pradeep Ambati, Íñigo Goiri, Felipe Frujeri, Alper Gun, Ke Wang, Brian Dolan, Brian Corell, Sekhar Pasupuleti, Thomas Moscibroda, Sameh Elnikety, et al. Providing SLOs for Resource-Harvesting VMs in cloud platforms. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 735–751, 2020.

[3] Nurşen Aydın, İbrahim Muter, and Ş İlker Birbil. Multi-objective temporal bin packing problem: An application in cloud computing. *Computers & Operations Research*, 121:104959, 2020.

[4] Yossi Azar and Danny Vainstein. Tight bounds for clairvoyant dynamic bin packing. *ACM Transactions on Parallel Computing (TOPC)*, 6(3):1–21, 2019.

[5] Microsoft Azure. https://azure.microsoft.com.

[6] Bharathan Balaji, Christopher Kakovitch, and Balakrishnan Narayanaswamy. Fireplace: Placing Firecracker virtual machines with hindsight imitation. *Proceedings of Machine Learning and Systems (MLSys)*, 3:652–663, 2021.

[7] KL Bansal et al. An analytical review of vm allocation and migration policies in cloud computing. In *2023 International Conference on Advancement in Computation & Computer Technologies (InCACCT)*, pages 704–710. IEEE, 2023.

[8] Hugo Barbalho, Patricia Kovaleski, Beibin Li, Luke Marshall, Marco Molinaro, Abhisek Pan, Eli Cortez, Matheus Leao, Harsh Patwari, Zuzu Tang, et al. Virtual machine allocation with lifetime predictions. In *Proceedings of Machine Learning and Systems (MLSys)*, volume 5, 2023.

[9] Shane Bergsma, Timothy Zeyl, Arik Senderovich, and J Christopher Beck. Generating complex, realistic cloud workloads using recurrent neural networks. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles (SOSP)*, pages 376–391, 2021.

[10] Niv Buchbinder, Yaron Fairstein, Konstantina Mellou, Ishai Menache, and Joseph Naor. Online virtual machine allocation with lifetime and load predictions. *ACM SIGMETRICS Performance Evaluation Review*, 49(1):9–10, 2021.

[11] Zuzhi Chen, Fuxin Jiang, Binbin Chen, Yu Li, Yunkai Zhang, Chao Huang, Rui Yang, Fan Jiang, Jianjun Chen, Wu Xiang, et al. Resource allocation with service affinity in large-scale cloud environments. In *2024 IEEE 40th International Conference on Data Engineering (ICDE)*, pages 5280–5293. IEEE, 2024.

[12] Tong Cheng, Hang Dong, Lu Wang, Bo Qiao, Si Qin, Qingwei Lin, Dongmei Zhang, Saravan Rajmohan, and Thomas Moscibroda. Multi-agent reinforcement learning with shared policy for cloud quota management problem. In *Companion Proceedings of the ACM Web Conference 2023*, pages 391–395, 2023.

[13] Alibaba Cloud. https://www.alibabacloud.com.

[14] Google Cloud. https://cloud.google.com.

[15] Maxime C Cohen, Philipp W Keller, Vahab Mirrokni, and Morteza Zadimoghaddam. Overcommitment in cloud services: Bin packing with chance constraints. *Management Science*, 65(7):3255–3271, 2019.

[16] Eli Cortez, Anand Bonde, Alexandre Muzio, Mark Russinovich, Marcus Fontoura, and Ricardo Bianchini. Resource central: Understanding and predicting workloads for improved resource management in large cloud platforms. In *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP)*, pages 153–167, 2017.

[17] Haochuan Cui, Junjie Sheng, Bo Jin, Yiqiu Hu, Li Su, Lei Zhu, Wenli Zhou, and Xiangfeng Wang. Reassigner: A plug-and-play virtual machine scheduling intensifier for heterogeneous requests. In *2022 IEEE International Conference on Big Data (Big Data)*, pages 3726–3734. IEEE, 2022.

[18] Xianzhong Ding, Yunkai Zhang, Binbin Chen, Donghao Ying, Tieying Zhang, Jianjun Chen, Lei Zhang, Alberto Cerpa, and Wan Du. VMR2L: Virtual machines rescheduling using reinforcement learning in data centers. In *Machine Learning for Systems 2023*, 2023.

[19] Hang Dong, Boshi Wang, Bo Qiao, Wenqian Xing, Chuan Luo, Si Qin, Qingwei Lin, Dongmei Zhang, Gurpreet Virdi, and Thomas Moscibroda. Predictive job scheduling under uncertain constraints in cloud computing. In *International Joint Conference on Artificial Intelligence (IJCAI)*, pages 3627–3634, 2021.

[20] Fahimeh Farahnakian, Tapio Pahikkala, Pasi Liljeberg, Juha Plosila, Nguyen Trung Hieu, and Hannu Tenhunen. Energy-aware vm consolidation in cloud data centers using utilization prediction model. *IEEE Transactions on Cloud Computing*, 7(2):524–536, 2016.

[21] Andrew D Ferguson, Peter Bodik, Srikanth Kandula, Eric Boutin, and Rodrigo Fonseca. Jockey: guaranteed job latency in data parallel clusters. In *Proceedings of the 7th ACM european conference on Computer Systems (EuroSys)*, pages 99–112, 2012.

[22] Włodzimierz Funika, Paweł Koperek, and Jacek Kitowski. Automated cloud resources provisioning with the use of the proximal policy optimization. *The Journal of Supercomputing*, 79(6):6674–6704, 2023.

[23] Anupam Gupta, Guru Guruganesh, Amit Kumar, and David Wajc. Fully-dynamic bin packing with limited repacking. *arXiv preprint arXiv:1711.02078*, 2017.

[24] Ori Hadary, Luke Marshall, Ishai Menache, Abhisek Pan, Esaias E Greeff, David Dion, Star Dorminey, Shailesh Joshi, Yang Chen, Mark Russinovich, et al. Protean:VM allocation service at scale. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 845–861, 2020.

[25] Benjamin Hindman, Andy Konwinski, Matei Zaharia, Ali Ghodsi, Anthony D Joseph, Randy Katz, Scott Shenker, and Ion Stoica. Mesos: A platform for {Fine-Grained} resource sharing in the data center. In *8th USENIX Symposium on Networked Systems Design and Implementation (NSDI 11)*, 2011.

[26] Suhas Jayaram Subramanya, Daiyaan Arfeen, Shouxu Lin, Aurick Qiao, Zhihao Jia, and Gregory R Ganger. Sia: Heterogeneity-aware, goodput-optimized ml-cluster scheduling. In *Proceedings of the 29th Symposium on Operating Systems Principles (SOSP)*, pages 642–657, 2023.

[27] Shahin Kamali and Alejandro López-Ortiz. Efficient online strategies for renting servers in the cloud. In *SOFSEM 2015: Theory and Practice of Computer Science:*

41st International Conference on Current Trends in Theory and Practice of Computer Science, Pec pod Sněžkou, Czech Republic, January 24-29, 2015. Proceedings 41, pages 277–288. Springer, 2015.

[28] Ajaykrishna Karthikeyan, Nagarajan Natarajan, Gagan Somashekar, Lei Zhao, Ranjita Bhagwan, Rodrigo Fonseca, Tatiana Racheva, and Yogesh Bansal. SelfTune: Tuning cluster managers. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 1097–1114, 2023.

[29] Guolin Ke, Qi Meng, Thomas Finley, Taifeng Wang, Wei Chen, Weidong Ma, Qiwei Ye, and Tie-Yan Liu. Lightgbm: A highly efficient gradient boosting decision tree. *Advances in Neural Information Processing Systems (NeurIPS)*, 30, 2017.

[30] Xiaodi Ke, Cong Guo, Siqi Ji, Shane Bergsma, Zhenhua Hu, and Lei Guo. Fundy: A scalable and extensible resource manager for cloud resources. In *2021 IEEE 14th International Conference on Cloud Computing (CLOUD)*, pages 540–550. IEEE, 2021.

[31] Haozhe Li, Minghua Ma, Yudong Liu, Si Qin, Bo Qiao, Randolph Yao, Harshwardhan Chaturvedi, Tri Tran, Murali Chintalapati, Saravan Rajmohan, et al. Codec: Cost-effective duration prediction system for deadline scheduling in the cloud. In *2023 IEEE 34th International Symposium on Software Reliability Engineering (ISSRE)*, pages 298–308. IEEE, 2023.

[32] Yusen Li, Xueyan Tang, and Wentong Cai. Dynamic bin packing for on-demand cloud resource allocation. *IEEE Transactions on Parallel and Distributed Systems*, 27(1):157–170, 2015.

[33] Jianheng Ling, Pratik Worah, Yawen Wang, Yunchuan Kong, Chunlei Wang, Clifford Stein, Diwakar Gupta, Jason Behmer, Logan A. Bush, Prakash Ramanan, Rajesh Kumar, Thomas Chestna, Yajing Liu, Ying Liu, Ye Zhao, Kathryn S. McKinley, Meeyoung Park, and Martin Maas. Lava: Lifetime-aware vm allocation with learned distributions and adaptation to mispredictions. In *Proceedings of Machine Learning and Systems*, volume 7. MLSys, 2025.

[34] Tennison Liu, Nicolás Astorga, Nabeel Seedat, and Mihaela van der Schaar. Large language models to enhance bayesian optimization. In *The Twelfth International Conference on Learning Representations (ICLR)*. OpenReview.net, 2024.

[35] Chuan Luo, Bo Qiao, Xin Chen, Pu Zhao, Randolph Yao, Hongyu Zhang, Wei Wu, Andrew Zhou, and Qingwei Lin. Intelligent virtual machine provisioning in cloud computing. In *International Joint Conference on Artificial Intelligence (IJCAI)*, pages 1495–1502, 2020.

[36] Chuan Luo, Bo Qiao, Wenqian Xing, Xin Chen, Pu Zhao, Chao Du, Randolph Yao, Hongyu Zhang, Wei Wu, Shaowei Cai, et al. Correlation-aware heuristic search for intelligent virtual machine provisioning in cloud systems. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 35, pages 12363–12372, 2021.

[37] Suhib Bani Melhem, Anjali Agarwal, Nishith Goel, and Marzia Zaman. Markov prediction model for host load detection and vm placement in live migration. *IEEE Access*, 6:7190–7205, 2017.

[38] Konstantina Mellou, Marco Molinaro, and Rudy Zhou. The power of migrations in dynamic bin packing. *arXiv preprint arXiv:2408.13178*, 2024.

[39] Aniket Murhekar, David Arbour, Tung Mai, and Anup Rao. Dynamic vector bin packing for online resource allocation in the cloud. *arXiv preprint arXiv:2304.08648*, 2023.

[40] Andrew Newell, Dimitrios Skarlatos, Jingyuan Fan, Pavan Kumar, Maxim Khutornenko, Mayank Pundir, Yirui Zhang, Mingjun Zhang, Yuanlai Liu, Linh Le, et al. Ras: continuously optimized region-wide datacenter resource allocation. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles (SOSP)*, pages 505–520, 2021.

[41] Mostafa Noshy, Abdelhameed Ibrahim, and Hesham Arafat Ali. Optimization of live virtual machine migration in cloud computing: A survey and future directions. *Journal of Network and Computer Applications*, 110:1–10, 2018.

[42] Xiaoting Qin, Minghua Ma, Yuheng Zhao, Jue Zhang, Chao Du, Yudong Liu, Anjaly Parayil, Chetan Bansal, Saravan Rajmohan, Íñigo Goiri, et al. How different are the cloud workloads? characterizing large-scale private and public cloud workloads. In *2023 53rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 522–530. IEEE, 2023.

[43] Bernardino Romera-Paredes, Mohammadamin Barekatain, Alexander Novikov, Matej Balog, M Pawan Kumar, Emilien Dupont, Francisco JR Ruiz, Jordan S Ellenberg, Pengming Wang, Omar Fawzi, et al. Mathematical discoveries from program search with large language models. *Nature*, 625(7995):468–475, 2024.

[44] Sultan Mahmud Sajal, Luke Marshall, Beibin Li, Shandan Zhou, Abhisek Pan, Konstantina Mellou, Deepak Narayanan, Timothy Zhu, David Dion, Thomas Moscibroda, et al. Kerveros: Efficient and scalable cloud

admission control. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 227–245, 2023.

[45] Malte Schwarzkopf, Andy Konwinski, Michael Abd-El-Malek, and John Wilkes. Omega: flexible, scalable schedulers for large compute clusters. In *Proceedings of the 8th ACM European Conference on Computer Systems (EuroSys)*, pages 351–364, 2013.

[46] Amazon Web Services. https://aws.amazon.com.

[47] Haiying Shen and Liuhua Chen. Compvm: A complementary vm allocation mechanism for cloud systems. *IEEE/ACM Transactions On Networking*, 26(3):1348–1361, 2018.

[48] Sean R Sinclair, Felipe Vieira Frujeri, Ching-An Cheng, Luke Marshall, Hugo De Oliveira Barbalho, Jingling Li, Jennifer Neville, Ishai Menache, and Adith Swaminathan. Hindsight learning for mdps with exogenous inputs. In *International Conference on Machine Learning (ICML)*, pages 31877–31914. PMLR, 2023.

[49] Chunqiang Tang, Kenny Yu, Kaushik Veeraraghavan, Jonathan Kaldor, Scott Michelson, Thawan Kooburat, Aravind Anbudurai, Matthew Clark, Kabir Gogia, Long Cheng, et al. Twine: A unified cluster management system for shared infrastructure. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 787–803, 2020.

[50] Cong Hung Tran, Thanh Khiet Bui, and Tran Vu Pham. Virtual machine migration policy for multi-tier application in cloud computing based on q-learning algorithm. *Computing*, 104(6):1285–1306, 2022.

[51] Alexey Tumanov, Timothy Zhu, Jun Woo Park, Michael A Kozuch, Mor Harchol-Balter, and Gregory R Ganger. Tetrisched: global rescheduling with adaptive plan-ahead in dynamic heterogeneous clusters. In *Proceedings of the 11th European Conference on Computer Systems (EuroSys)*, pages 1–16, 2016.

[52] Abhishek Verma, Ludmila Cherkasova, and Roy H Campbell. Aria: automatic resource inference and allocation for mapreduce environments. In *Proceedings of the 8th ACM international conference on Autonomic computing*, pages 235–244, 2011.

[53] Abhishek Verma, Luis Pedrosa, Madhukar Korupolu, David Oppenheimer, Eric Tune, and John Wilkes. Large-scale cluster management at google with borg. In *Proceedings of the tenth European Conference on Computer Systems*, pages 1–17, 2015.

[54] Neeraja J Yadwadkar, Ganesh Ananthanarayanan, and Randy Katz. Wrangler: Predictable and faster jobs using fewer resources. In *Proceedings of the ACM Symposium on Cloud Computing (SOCC)*, pages 1–14, 2014.

[55] Chengrun Yang, Xuezhi Wang, Yifeng Lu, Hanxiao Liu, Quoc V Le, Denny Zhou, and Xinyun Chen. Large language models as optimizers. In *The Twelfth International Conference on Learning Representations (ICLR)*, 2024.

[56] Fangkai Yang, Bowen Pang, Jue Zhang, Bo Qiao, Lu Wang, Camille Couturier, Chetan Bansal, Soumya Ram, Si Qin, Zhen Ma, et al. Spot virtual machine eviction prediction in microsoft cloud. In *Companion Proceedings of the Web Conference 2022*, pages 152–156, 2022.

[57] Chen Ying, Baochun Li, Xiaodi Ke, and Lei Guo. Raven: Scheduling virtual machine migration during datacenter upgrades with reinforcement learning. *Mobile Networks and Applications*, 27(1):303–314, 2022.

[58] Zhi-Hui Zhan, Xiao-Fang Liu, Yue-Jiao Gong, Jun Zhang, Henry Shu-Hung Chung, and Yun Li. Cloud computing resource scheduling and a survey of its evolutionary approaches. *ACM Computing Surveys (CSUR)*, 47(4):1–33, 2015.

Appendix

A VM Lifetime Prediction Model

A.1 Initial Lifetime Prediction

This model provides low-latency predictions at VM creation time for the online scheduler, using static features available in the VM request. The 19 features used are listed in Table 2.

Feature Importance Analysis. As shown in Figure 15, feature importance analysis reveals that *image_id* , *user_id*, and *operator* are substantially more predictive than all other static features. This indicates that a VM's lifetime is primarily determined by its intended application (proxied by the image) and the historical usage patterns of its owner (user/operator). A secondary tier of features, including *inst_type_name* and *min_of_day*, also provide significant predictive value.

A.2 Remaining Lifetime Prediction

This model provides higher-accuracy predictions for running VMs, designed for offline optimization tasks and state updates. It augments the 19 static features (Table 2) with dynamic runtime information, historical user statistics, and performance metrics. Key additional features are listed in Table 3.

Table 2: Features for the Initial Lifetime Prediction model.

Feature	Type	Description
min_of_day	Int	Minute of day (0-1439)
day_of_week	Int	Day of week (0-6)
user_id	String	Unique user identifier
biz_id	String	Business unit identifier
channel	String	Request source (API, Console)
operator	String	Operator identity
cpu_cores	Int	Number of CPU cores
mem_size	Int	Memory size (GB)
inst_type_name	String	Full instance type name
inst_family	String	Instance family
sysdisk_cat	String	System disk category
sysdisk_perf	String	System disk performance level
sysdisk_size	Int	System disk size (GB)
region_id	String	Deployment region ID
avail_zone	String	Deployment availability zone
pay_type	String	Payment model
is_spot	Boolean	Is a Spot Instance
image_id	String	VM image ID
use_default_vpc	Boolean	Uses the default VPC

Table 3: Key additional features for the Remaining Lifetime Prediction model.

Feature	Type	Description
survival_hrs	Int	Current runtime (hours)
is_container	Boolean	Is Container Instance
avg_life_1mo	Float	User avg. lifetime (last mo.)
avg_buys_1mo	Float	User avg. daily buys (last mo.)
cpu_p95_1wk	Float	P95 CPU util. (last week)
mem_p95_1wk	Float	P95 memory util. (last week)

Feature Engineering. Beyond the basic features in Table 3, we engineer a comprehensive set of features. (1) *Historical Statistical Features*: we compute statistical aggregates (e.g., avg, std, p75, p90) of user-level metrics like lifetime and *vm_num* over various time windows (e.g., past 0.5, 1, and 2 months); (2) *Offline Performance Metrics*: we collect recent performance profiles, such as the 95th percentile of cpu utilization, mem utilization over the past day or week.

Feature Importance Analysis. As shown in Figure 16, the feature importance for this model shifts significantly. It is dominated by historical user behavior and the VM's current runtime. This confirms that a user's past behavior and a VM's survival history are the strongest predictors of its remaining lifetime, validating our two-model approach where the offline model gains its power from rich, dynamic information.

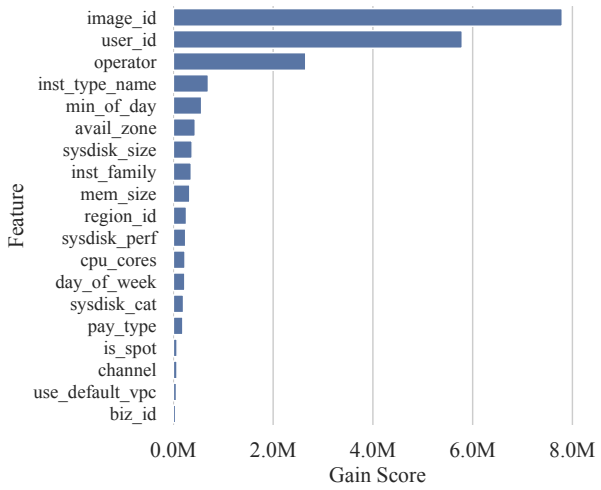


Figure 15: Feature importance for Initial Lifetime Prediction.

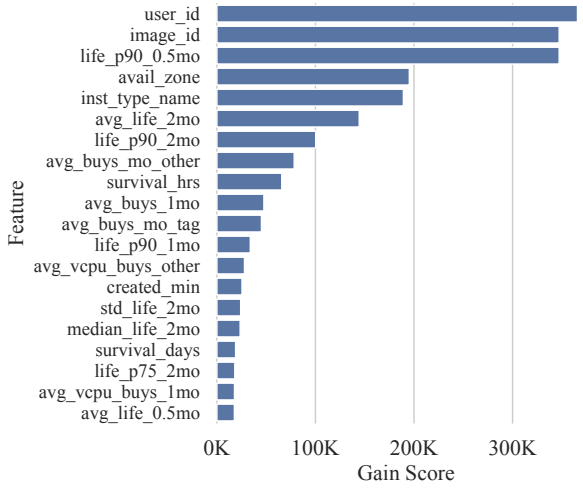


Figure 16: Top 20 Features for Remaining Lifetime Prediction.

B Details of Dynamic Affinity Grouping

This appendix provides the detailed algorithms for the key functions within DVLA's Dynamic Affinity Grouping mechanism, as referenced in the main paper.

B.1 Online Drift Detector

Our change detection is powered by an online detector that maintains Exponentially Weighted Moving Averages (EWMAs) of the workload vector, the drift error, and the error's standard deviation. This enables an adaptive drift scoring mechanism. Algorithm 3 presents the core logic.

Algorithm 3 Online Drift Detector Internals

State Variables:
  `ewma_vector`: EWMA of workload vectors (the baseline).
  `ewma_error`: EWMA of drift errors.
  `ewma_stddev_error`: EWMA of the standard deviation of drift errors.
  α, β: Smoothing factors for vector and error updates.
  t: Internal day counter.
  P_{warmup}: Number of warmup days.

function SCORE(v_t)
  if $t \leq P_{\text{warmup}}$ **or** `ewma_vector` is **null then**
    return 0.0
  $error \leftarrow 1.0 - \text{COSINESIMILARITY}(v_t, \texttt{ewma_vector})$
  $score \leftarrow \max\left(0, \frac{error - \texttt{ewma_error}}{\texttt{ewma_stddev_error} + \varepsilon}\right)$
  return *score*

function LEARN(v_t)
  $t \leftarrow t + 1$
  if `ewma_vector` is **null then**
    `ewma_vector` $\leftarrow v_t$
    return
  $error \leftarrow 1.0 - \text{COSINESIMILARITY}(v_t, \texttt{ewma_vector})$
  if `ewma_error` is **null then**
    `ewma_error` $\leftarrow error$
    `ewma_stddev_error` $\leftarrow 0.0$
  else
    $std_dev \leftarrow |error - \texttt{ewma_error}|$
    `ewma_error` $\leftarrow \beta \cdot error + (1-\beta) \cdot \texttt{ewma_error}$
    `ewma_stddev_error` $\leftarrow \beta \cdot std_dev + (1-\beta) \cdot$ `ewma_stddev_error`
  `ewma_vector` $\leftarrow \alpha \cdot v_t + (1-\alpha) \cdot$ `ewma_vector`

B.2 Adaptive Affinity Group Generation

When a persistent drift is confirmed, this function (Algorithm 4) generates a new, optimized affinity group by identifying the most resource-dominant lifetime classes from the observed workload.

C Sensitivity Analysis

This section analyzes DVLA's sensitivity to its key hyperparameters and the accuracy of its underlying prediction models, providing a principled basis for our production configuration.

C.1 Parameters of Dynamic Affinity Grouping

The Dynamic Affinity Grouping mechanism's key hyperparameters θ_{alarm}, W_{obs}, and $\theta_{confirm}$ must balance responsiveness against stability. We conducted a sensitivity analysis by

Algorithm 4 GenerateAffinityGroup

Input:
  `avg_workload`: Average workload vector.
  `GlobalClasses`: List of all global lifetime classes.
  θ_{coverage}: Cumulative threshold.
  N_{min}: Minimum number of classes to keep.
Output:
  `AffinityGroup`: The new set of lifetime classes.

function GENERATEAFFINITYGROUP(`avg_workload`, `GlobalClasses`, θ_{coverage}, N_{min})
  `Pairs` $\leftarrow$ ZIP(`GlobalClasses`, `avg_workload`)
  `SortedPairs` $\leftarrow$ SORT(`Pairs`, descending = True)
  `AffinityGroup` $\leftarrow \emptyset$
  cum_coverage $\leftarrow 0.0$
  for each $(label, val)$ in `SortedPairs` **do**
    if *cum_coverage* $< \theta_{\text{coverage}}$ **or** $|\texttt{AffinityGroup}| < N_{\text{min}}$ **then**
      `AffinityGroup` $\leftarrow$ `AffinityGroup` $\cup \{label\}$
      cum_coverage $\leftarrow$ *cum_coverage* $+ val$
    else
      break
  return `AffinityGroup`

evaluating parameter combinations against simulated workloads, using a unified cost function that penalizes false positives more heavily than detection latency (70% weight), reflecting our operational priority for system stability.

Figure 17 shows that a conservative strategy yields the lowest cost. Short observation windows proved overly sensitive to transient noise. A moderate window of 20 days, combined with higher alarm and confirmation thresholds, consistently minimized cost by ensuring the system reacts only to significant, persistent changes. This data-driven analysis led us to select the optimal, stability-focused parameter set ($W_{obs} = 20$, $\theta_{alarm} = 2.0$, $\theta_{confirm} = 3.0$) for our production deployment.

C.2 Robustness to Prediction Errors

DVLA's effectiveness is linked to the quality of its VM lifetime predictions. To quantify this relationship, we evaluated its performance under varying synthetic prediction accuracies. Our primary metric is the PD improvement over the production baseline.

The analysis, shown in Figure 18, yields two key insights. First, DVLA's performance scales directly with prediction quality, particularly for long-lived VMs. Under a realistic accuracy model, increasing the recall of long-lived VMs from 40% to 100% boosts the PD gain from 1.02 pp to 1.66 pp. This confirms that correctly identifying these resource-intensive VMs is a primary driver of efficiency.

Second, and more importantly, DVLA demonstrates strong

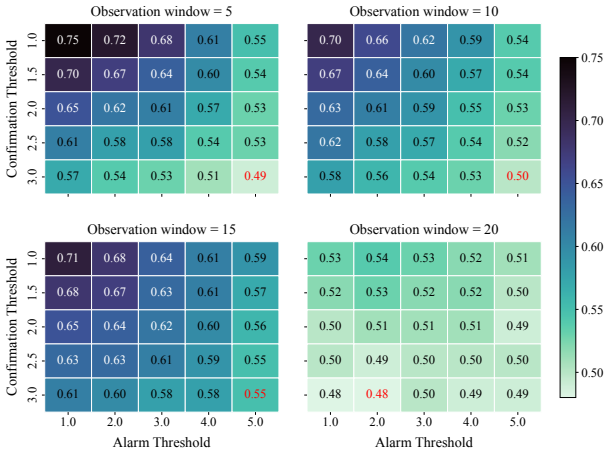


Figure 17: Sensitivity analysis of DVLA's change detection parameters. Each heatmap shows the average cost (lower is better) for a given `observation_window`.

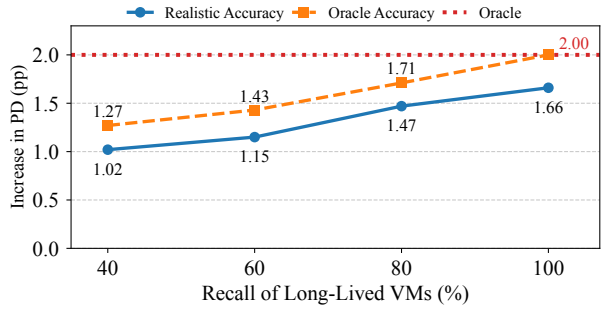


Figure 18: Sensitivity analysis of DVLA's performance with respect to VM lifetime prediction accuracy. The plot shows the Packing Density improvement as a function of the recall rate for long-lived VMs.

robustness to prediction errors. Even with a low 40% recall for long-lived VMs, the system still achieves a significant 1.02 pp PD improvement. This resilience highlights that DVLA's synergistic design can extract substantial value even from noisy signals, making it well-suited for real-world deployment where perfect prediction is unattainable.

PIMS: Fleet-wide Datacenter Maintenance with Minimal Capacity Buffer and Predictable Latency (Operational System)

Benjamin Leonhardi[1], Evangelia Kalyvianaki[2], Yang Wang[1 3], Abdelrahman Adam[1], Agshin Nabiyev[1], Aleks Shirokov[1], Amitav Mohanty[1], Daniil Balenko[1], Elaine Zhao[1], Essam Ewaisha[1], Hongbo Dong[4], Igor Marnat[1], Lev Novikov[1], Min Zeng[1], Steven Shingler[5], Timofey Durakov[1], Wiliam de Abreu Pinho[1], Ben Christensen[1], Mayank Pundir[1], and Kaushik Veeraraghavan[1]

[1] Meta Platforms
[2] University of Cambridge
[3] The Ohio State University
[4] NexGeMM LLC
[5] Independent Researcher

Abstract

Maintenance is a fundamental operation in datacenters to ensure that hardware and software operate correctly, efficiently and use up-to-date versions. We present Meta's maintenance system in production over the last five years that provides continuous support to tens of thousands of services running on our fleet of millions of servers and seamlessly orchestrates this process. To our knowledge, this is the first paper to discuss predictable maintenance at scale.

The key challenge is how to minimize the capacity buffer—servers reserved to absorb capacity loss caused by maintenance—while providing a predictable latency to maintenance operations. This paper presents a series of strategies and techniques we use to accomplish this goal, such as aligning maintenance with fault domains, placing hardware evenly across fault domains, a maintenance contract among participating parties, etc. Indicatively, we observe that these techniques have helped us reduce the size of the capacity buffer by about 15% in one quarter of 2025 and allowed us to perform a fleet-wide deployment under targeted SLOs (e.g., 45 days for a new OS, 90 days for a new firmware).

1 Introduction

This paper presents Predictive Integrated Maintenance at Scale (PIMS), Meta's approach to fleet-wide planned maintenance that includes software upgrades and physical maintenance of devices such as servers, network switches, and power components. Meta operates a hyperscale private cloud of millions of servers running tens of thousands of services and serving billions of users. Orchestrating maintenance at this scale is challenging as it has several conflicting goals.

To prevent service interruption due to loss of servers, Meta reserves additional servers, called a capacity buffer [11,27,31]. If a service loses servers due to maintenance or failures, it can use servers in the capacity buffer to continue its operation. PIMS considers three different types of events that require capacity buffers: a) a loss of servers due to scheduled maintenance; b) a loss of a fault domain (i.e., a group of servers sharing a single point of failure like a switch or a power supply) due to unexpected failures of the shared component; c) a loss of a fault domain due to scheduled physical maintenance of the shared component.

Intuitively, the bigger the buffers allocated to maintenance the more machines can be maintained in parallel and so maintenance can be completed faster. However, considering Meta is operating a hyperscale fleet with millions of servers, these buffers could represent a significant resource usage of the entire fleet. Therefore, the key challenge PIMS faces is **how to minimize capacity buffers while keeping a predictable latency for maintenance operations.**

Minimal capacity buffer. PIMS performs maintenance in each region (i.e., several datacenters close to each other) independently. This enforces us to define capacity buffers per region and so we allocate a capacity buffer at the size of a fault domain within a single region. Below, we explain the reasons for this and describe how a single capacity buffer can be efficiently used to cover for all three types of events described above towards minimizing its size.

First, since servers in a fault domain share either the power supply or the network switch, when the shared device is maintained, all servers in its domain become unavailable. This inevitably drives us to align maintenance with fault domains (i.e., maintain one fault domain at a time), which makes it possible to use a single buffer for events a) and c).

Second, we further use the same buffer to handle events b), i.e., the loss of a fault domain due to a shared device unexpected failure. The risk here is that if a domain fails during the maintenance of another domain, then we are losing two domains and the service may not have enough buffer. However,

we can do this due to the following: for physical maintenance, which cannot be paused easily, due to its low frequency (9% of all maintenance operations), it is very unlikely for a domain failure and a physical maintenance to happen at the same time. Automated maintenance, such as kernel or firmware upgrades, is much more frequent, so the chance of both happening together is non-negligible. To address this problem, our strategy is to pause the automated maintenance quickly when another domain fails and gives back the capacity.

Third, naturally, smaller fault domains lead to a smaller capacity buffer. By applying redundancy to the shared components, we can shrink or even eliminate fault domains, but that would increase the resource usage and the complexity of our infrastructure. Meta takes a middle ground by applying redundancy to the top-level shared components and leave the lower-level components unreplicated. In this way, each of our fault domains contains about 20K servers. However, note that, shrinking fault domains does not automatically lead to smaller capacity buffer. Assuming that a service allocates servers from multiple fault domains, it needs a capacity buffer to account for the loss of the fault domain that holds the greatest number of its servers. Therefore, to minimize the capacity buffer, it's better for a service to allocate servers evenly from different fault domains.

Accomplishing this goal requires both an even distribution of each type of hardware across different fault domains and an algorithm to evenly distribute the machines allocated to a service across different fault domains. While the latter is well studied [1, 12, 21], we are not aware of practical solutions to achieve the former. In Meta, we have introduced a hardware placement algorithm that aims to minimize the uneven distribution of hardware across fault domains via rack movements. We use these outputs as suggestions and as appropriate we regularly relocate racks of machines across different positions. Concretely, the rack relocation moves suggested by the hardware placement algorithm has helped us reduce the size of our buffers by 15% in 2025 Q2 alone, which yields a saving of tens of thousands of servers.

Predictable end-to-end maintenance latency. Maintenance operations can have constraints about conflicts, dependencies, priorities, etc. To handle them, PIMS incorporates a centralized scheduler to schedule maintenance operations, with the following techniques.

First, maintenance latency involves the latency in PIMS to schedule a maintenance operation, the latency in the affected services to switch to the capacity buffer, and the latency of the maintenance operation itself. Achieving a predictable end-to-end latency inevitably requires collaboration among three parties The core of PIMS is a *maintenance contract*, which clearly defines the expectation for each party. Second, to meet its expectation, PIMS incorporates a layered design. Each layer operates with its own service-level objective (SLO) such that collectively all layers contribute to comply with the maintenance contract. Third, PIMS makes a balance between maintenance latency and throughput by combining batching, which executes multiple maintenance operations within a domain, and prioritized scheduling within a batch.

To the best of our knowledge, this is the first work to present fleet-wide datacenter maintenance operations. The contributions of this paper are as follows: 1. We present a holistic approach of planned maintenance covering hardware, software and data placement. The core is a maintenance contract to define the expectations for each participating party. 2. We present methods to minimize capacity buffers, by multiplexing the same buffers across different event types and minimizing their sizes through even hardware spread. 3. We present a maintenance stack to centrally manage shared capacity buffers and to carefully orchestrate fleet-wide predictable rollouts. 4. We present performance results of our system in production over the last five years.

2 Background

2.1 Maintenance operations

Maintenance operations, or maintenances, encapsulate a large and diverse set of required actions on our assets, i.e., all devices including compute and storage servers, network and power components, to keep an up-to-date and secure fleet for our services. We identify four categories of operations, shown in Figure 1, based on how they are initiated and whether they are executed by datacenter (DC) engineers or software workflows. We also distinguish between *planned* and *unplanned* maintenance. The latter is usually corrective. This paper is on planned maintenance shown with green in the figure.

First, planned automated maintenance involves software upgrades managed as rollouts. These operations keep our servers up-to-date and secure and some of these upgrades are required by law. Installation of all types of non application-specific software (e.g., kernels, drivers, etc) and firmware belongs in this category. Note that application-level software maintenance is managed differently, with Conveyor [15], and is not the focus of this paper.

Second, planned physical maintenance refers to periodic operations to support hardware health. For example, some hardware components have to be periodically shut down, taken apart and tested.

Third and fourth unplanned physical and automated operations refer to those executed when an unexpected problem is detected on a server or rack that requires urgent action. Depending on the nature of the problem and its trigger, e.g. automated alert or a ticket from a DC engineer, maintenance operation can be performed either automatically or manually.

For completion, there are also additional important DC maintenance operations which are not the focus of this paper. These include physical operations of large scope (e.g., busway replacement due to overheating); operations resulting

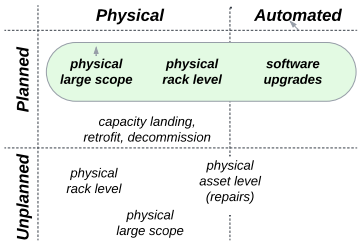


Figure 1: Maintenance operations. The focus of this paper is on planned maintenance operations shown in green.

in changes to fleet composition, at physical or logical levels, such as turning up new hardware and decommissioning old hardware or changing configuration of existing servers to run different kinds of service workloads; and even datacenter outages [32].

To summarize, PIMS is responsible for scheduling the first two categories of planned automated and physical maintenance. The last two categories are more on-demand in nature, relying on a different set of expectations, and are therefore served differently. However, scheduling the first two categories requires PIMS to consider all categories of maintenance coherently, and we will illustrate this in later sections.

To provide context, about 91% of maintenance operations are automated maintenance and 9% are physical maintenance.

2.2 Fault domains

In Meta, a *fault domain* refers to the set of servers connected to the same power supply or network switch. In this case, if the shared device is down, all servers in the domain becomes unavailable. We continue to elaborate on the size of fault domains related to a power supply or a network switch.

Power distributions in DCs are composed of high, medium and low voltage equipment. An equipment with a higher voltage usually supports multiple equipments with a lower voltage, which means the unavailability of the former has a larger impact than the latter. As a result, Meta applies full redundancy to medium and high voltage equipments to avoid to take down too many servers. Natural disasters, like tornadoes or wildefires, can still take down those equipments, but such events are rare, and Meta has incorporated a separate buffer to absorb such loss. As a result, PIMS's capacity buffer does not consider the loss of servers due to such rare major disasters. A Main Switch Board (MSB) is the low voltage component which supports the largest number of servers. It has partial redundancy (there are usually two reserve MSBs per DC; each DC has 36 to 72 MSBs) and this could power around O(10,000) servers. We observed high frequency of MSB faults which led us to the need of a capacity buffer to handle MSB faults.

We designed our network topology so that a switch and its associated servers use the same power supply. In this way, a network fault domain is always within a power fault domain, so that PIMS only needs to consider the power fault domain.

2.3 Maintenance buffer and others

As discussed in Section 1, PIMS uses one buffer, defined as b_{PIMS}, to absorb loss caused by both unexpected domain faults and planned maintenance operations. In other words, PIMS *aligns maintenance domains (MDs) with fault domains.* The requirement to maintain the shared devices, e.g., MSBs and switches, is the primary reason to align maintenance domains with fault domains. In the rest of the paper, we will use MD to refer to a fault or maintenance domain.

Physically, PIMS maintains one b_{PIMS} for each region and it includes servers spanning different MDs of the corresponding region. b_{PIMS} should be able to absorb the loss of any MD. Since there are different types of hardware, which may be distributed unevenly across MDs, the size of b_{PIMS} is typically larger than the size of the largest MD. Concretely, for each server type, its required buffer size is equal to the maximal number of this server type across all MDs. Then the size of b_{PIMS} is equal to the sum of the required buffer size of each server type. For simplicity, we normalize different server types using their power consumptions. The size of b_{PIMS} varies over time. As of May 2026, its size is about 3% of our fleet.

Meta internally maintains multiple kinds of buffers for different purposes. We present a few examples to clarify the scope of b_{PIMS}. First, Meta maintains a buffer to absorb loss caused by random server failures, whose size is about 2% of our fleet. Second, Meta maintains a buffer to absorb loss caused by major disasters like a tornado disabling a whole region. Its size changes over time due to the addition of new regions. Third, as discussed above, each application maintains its own buffer for application-level maintenance. As a result, b_{PIMS} does not need to consider loss caused by these events.

We have been considering the possibility to merge b_{PIMS} and disaster recovery buffer to further reduce buffer size: to accomplish this, we will need to do maintenance region by region. This is possible since services in Meta are designed to be able to tolerate a region failure [32]. However, there are many complexities with HW and services affinity, services expectations, capacity tracking, traffic redirection, reliability expectations, etc, which are under discussion.

3 Design of PIMS

Figure 2 shows an overview of our system on Predictable Integrated Maintenance at Scale (PIMS). There are two independent streams of work, namely maintenance and placement, both targeting all Maintenance Domains (MDs) in a region and across all regions. The placement stream targets to make similar MDs of *even spread* of their hardware assets, services running and data stored. The maintenance stream is responsible to apply all maintenance requests from clients

(i.e., Meta employee who submit maintenance requests), resolve conflicts and adhere to upgrades priorities. It operates on the assumption of even MDs as these are created by the placement stream.

3.1 Maintenance contract

In the maintenance stream we introduce the *maintenance contract* that describes important system guarantees for predictability. The maintenance contract has the following core four expectations among all parties involved including maintenance clients, components of the maintenance stack and services running on machines in our private cloud:

1. All services are expected to tolerate the loss of one MD at any given point in time so that services and data should be spread across different MDs, if necessary.
2. All services have to drain their hosted machines within one MD according to the agreed SLOs, e.g, within three hours. This is to ensure that the maintenance stack can safely schedule maintenance events on clean-state machines for predictability.
3. The buffer (b_{PIMS}) supports a maximum unplanned fault domain of servers powered by a single MSB in the common case. Following the discussion on the selection of MDs in Section 2.2, this ensure that b_{PIMS} occupies the minimum capacity buffer in our fleet that can support our maintenance contract.
4. In the event of an unplanned fault, any software upgrade operations using b_{PIMS} will relinquish capacity within an SLO of 35 minutes so that the buffer can be used to mitigate the capacity loss due to the fault. This is a consequence of multiplexing buffers for maintenance and failures.

This simple but powerful contract allows us to design and build a system with the following characteristics on maintenance, services and failures: It offers a predictable maintenance cycle time for the entire region, reducing the variability of a software rollout duration; where previously some rollouts were very quick and others were very slow. It offers predictable start times for maintenance events and a predictable schedule of operations for DC technicians so they can plan corrective maintenances. Services don't have to be aware of physical maintenances as long as the maintenances are contained within the MD. Exceptional cases can be handled with a regional disaster recovery strategy. Services can choose how to handle the loss of one MD. For example, stateful services can factor that into their state management control plane and quorum-based services can decide to ensure there is no loss of quorum. Service placement can be balanced across MDs. It allows us to support the failure of MSBs on the power path, which is our largest component without full physical redundancy. It enables DC designers to make decisions within the expectation of a maximum supported number of MDs. Finally, we can tolerate physical faults gracefully such that services remain unaware of them. As far as we know, this is the first system to provide these characteristics.

3.2 Scheduling in Maintenance Stream

The input to PIMS includes a list of maintenance requests, either submitted by clients or generated automatically. Each request asks for a software update or physical maintenance on all or a subset of servers in a region. Since preparing for a maintenance can take a long time (i.e., three hours to drain the traffic of an MD) and many types of maintenance requests themselves take shorter (e.g., tens of minutes to upgrade the OS), it is very inefficient to execute one maintenance request after each preparation. Instead, PIMS executes those requests in a batched manner at a single MD at a time: after it drains the traffic for one MD, it executes a batch of maintenance requests in that MD and then moves to the next MD.

We use the railroad metaphor to name the components of this workflow. The *maintenance train* is the overarching approach: a continuous cycle that visits MDs one at a time. A *train cycle* is one full pass through all MDs in a region. A *boxcar* is the atomic unit of maintenance work: a concrete, time-bounded plan for a specific MD, comprising a drain window followed by a work window. In the drain window all services of the selected MD are migrated to b_{PIMS}. In the work window all required upgrades are installed in the MD's assets and at the end of it services are un-drained back from the buffer to this MD.

In such a design, the scheduling algorithm of PIMS needs to answer two questions: 1. Which MD shall PIMS visit next? 2. Which maintenance requests shall PIMS execute in that MD? PIMS employs a two-step approach of first scheduling an MD and then the maintenance operations within the selected MD. However, all scheduling decisions are made in a way to align towards fleet-wide SLOs. This section provides a high-level description as follows. Section 4 provides more details about implementation and corner cases.

Ordering MDs. PIMS selects the next MD for maintenance according to the least recently visited (LRV) order, naturally due to the consideration that we do not want to leave some MDs unmaintained for long. It therefore determines the order within and the duration of the train cycle depending on the number of MDs per region and the time to spend per MD visit. Note that although a train cycle primarily determines the order of the scheduled MDs, in practice we also refer to the duration of a train cycle as the total time needed to maintain all MDs in region. For example, for regions with 72 MDs and MD visits of 12h, we need 36 days to visit each MD once. We also need to account time for revisits in case of failures. Our experiences have shown that an additional 9 days are sufficient to capture our needs. In this case, the SLO

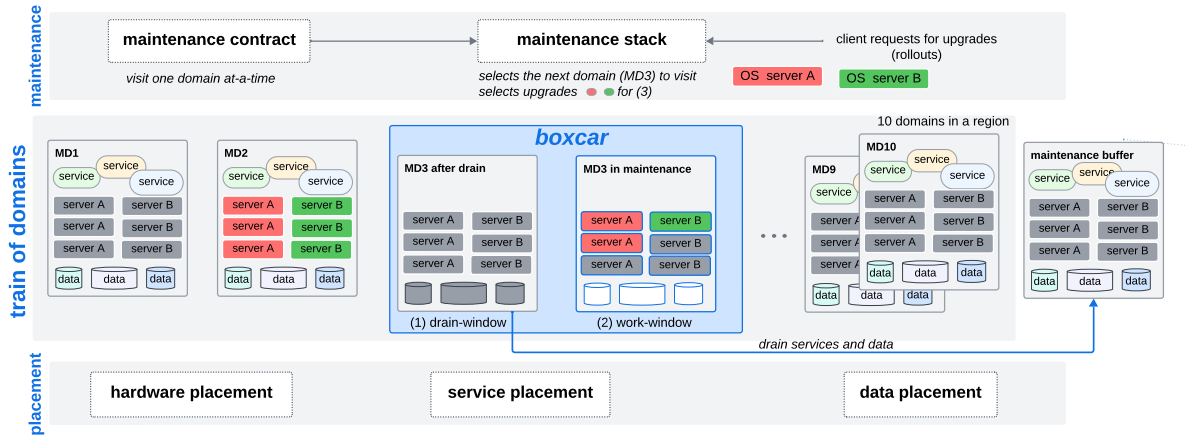


Figure 2: PIMS overview. Figure shows an example of a region of 10 MDs where hardware, service and data are evenly spread across MDs. At any point a single MD is in maintenance leaving 9 MDs to run services. This figure shows a train of 10 MDs and a rollout example in progress of two new OS versions (red and green) to be installed on server types A and B. The figure shows a snapshot in time when MD2 has already been updated and MD3 is being maintained. First, during drain-window, MD3's services have been drained to the maintenance buffer. Then, at the work-window MD3's servers are being updated while MD3's services run from the maintenance buffer.

of this layer is *to guarantee at least one visit for every MD in this region every 45 days*. Over the years we have worked with different train cycles of like 30 or 45 days. Nowadays, we keep a train cycle of 45 days, 90 days, and so on regardless of how machines are located in the region.

There are additional constraints to consider when ordering MDs in a train. For example, there is a constraint on working hours to capture that manual maintenance of small scope, such as of several racks, must be performed within working hours and this is normally scheduled as part of a scheduled MD visit in a train cycle. To address this, we create time slots for up to 90 days with fixed start and end times aligned with the region's timezone which is necessary to enable manual operations alongside automated/planned operations. 90 days allows us to schedule MDs beyond the SLO of 45-days in cases where any time slots cannot be used. Then, we iterate over all slots in a chronological order starting from the earliest until all MDs are scheduled using the LRV order and other constraints. To accommodate preventative physical maintenances we periodically recalculate the order of MDs in the 90 days time slots although we fix the order of MDs for the next 3 working days for stability and predictability.

Note that LRV is not different from round robin scheduling if there is no constraint on when PIMS can maintain an MD. However, the additional constraints for manual maintenance as mentioned above and the occasional emergency maintenance (§4.2.3) may disrupt the round robin order and that's why PIMS adopts LRV.

Planning within an MD. Typically, due to high demand, the time to run all requests for all assets in an MD exceeds the work-window time of its boxcar. Here we describe how PIMS prioritizes the requests in the boxcar plan such that all rollouts will finish according to their SLOs.

tier	description	examples	SLO
gold	important, time-critical	kernel, driver version upgrade	100%, 45 days
silver	important, not urgent	firmware upgrade new OS	95%, 180 days 95%, 90 days
bronze	best-effort	non-critical firmware	n/a

Table 1: Tiers of upgrades. In the first quarter of 2026, about 65% of the rollouts are gold tier, 35% are silver tier, and the number for bronze tier is negligible.

Our goal is to rank all requests and so we define three priority tiers shown in Table 1. The *gold tier* is for time-critical important maintenance workflows like kernel and driver upgrades that need to be scheduled as soon as possible. The *silver tier* is for less time-critical workflows that though still require frequent and deterministic rollout convergence such as firmware version upgrades. The *bronze tier* is for workflows that are not time-sensitive and include deep health checks and other housekeeping. We then fill the boxcar plan with as many maintenance operations as possible starting from the most important ones until we reach the duration of the MD visit for all servers.

A boxcar plan has a fixed duration: e.g., 6h out of which 3h for drain and 3h for effective maintenance work. Our goal is *to allow equitable progress of rollouts based on their importance and their workflow expected duration*. A rollout's expected duration is typically estimated from its historical

data. We split the 3h work window into a set of *budgets* where each budget has a target percent of hosts to represent the minimum set of hosts required to be successfully upgraded with a specific type of upgrade. For example, for gold-tier workflows we set the target budget percent to 100% since they typically take 15 to 30 min in total to execute so all such upgrades can easily fit in a single boxcar. Silver-tier workflows are often slower as they take one to two hours to execute and they are less critical, and so their budgets are proportional to their expected workflow duration. Bronze-tier upgrades are executed on a best-effort once gold- and silver-tier finish execution.

Using budgets, we define SLOs with clients. For example, the SLO with the kernel team is to upgrade every asset in the fleet within 45-days of the rollout creation. The SLO with the firmware team is to upgrade every asset in the fleet within 90-days of the rollout creation (50% budget). We further use the request creation timestamp as a second key of prioritization within the same upgrade type to prioritize new request over older ones. We then create the boxcar plan with scope, duration and priority per request.

3.3 Placement Stream

The goal of the placement stream is to create similar MDs of even spread of their hardware assets, services running, and data stored. As discussed in Section 1, this is fundamental to minimize the capacity buffer b_{PIMS}.

Hardware placement. We formulate the placement problem as a mixed-integer linear programming problem (MILP) to minimize an objective function, which includes the spread of assets across MDs and some other factors. Although our formulations conform to traditional MILP representations that could be solved by known solvers, our problem size makes this approach hard to maintain and with scalability problems [8, 20]. Instead, we use a brute force greedy search algorithm for scalability and low latency reasons. This algorithm works in multiple iterations. In every iteration, it searches for one movement (i.e., move a rack of servers from one location to another) to minimize the objective value, and it repeats this process for multiple iterations until either it cannot find a beneficial movement or a limit is reached. This approach does not guarantee to find the global optimum, but it's fast and can provide a reasonably good plan in our experience. We next describe the objective function and our greedy algorithm in more details. We begin by providing necessary terminology.

A *rack* holds a group of servers of the same type. There are a number of *rack_type*, represented by the type and number of servers in a rack. A *position* is a physical location in a datacenter that can hold a rack. The goal of the placement algorithm is to determine which position should hold which rack. Our placement algorithm considers three types of *resources*: power supply, network bandwidth, and cooling capability. A *scope* is a group of positions that share a resource limit. For example, positions that are connected to the same MSB become a scope; positions that are connected to the same network switch also become a scope. A *period* is a time period (e.g., a week).

Figure 3 presents the detailed definition of our objective function, which is a weighted sum of multiple terms. For most of the terms, we represent decision variables and other inputs as tensors in PyTorch and define the objective terms as certain computations on those tensors. The benefit of this approach is that we can then use PyTorch to compute the gradient of those terms, which represent how the values of those terms would change if we moved a rack into or out of a position. There are a few terms we found challenging to represent as tensors and thus we use a different representation.

Concretely, on the left side of this figure, the layout tensor, which maps a rack_type to a position, is the decision tensor. There are four input tensors: 1. The position_by_scope tensor records which positions each scope includes. Note that a position may be included in multiple scopes. 2. The scope_resource_limit tensor records the resource limit of each scope. 3. The racktype_by_resource tensor records the resource consumption of each rack_type. 4. The resource_scale factor records the max value for each resource across all rack_types, used for normalization purpose.

On the right side, the objective function includes three groups of terms. The first group tries to encourage each scope to have enough unused resource (i.e., "headroom") to accommodate future rack placement. This group considers power of MSB and network bandwidth of switches at this moment. The second group tries to penalize resource over-utilization. Note that we allow oversubscription of power and cooling but want to discourage that. The third group encourages an even distribution of different types of servers across different scopes. The other nodes in the middle represent intermediate tensors that are used to compute the objective terms from the input and decision tensors and how to compute them.

There are two additional terms: the network compatibility score determines how well a rack_type fits with the position's network configuration. There are generally three types: if it's totally incompatible (e.g., a rack_type requires Infiniband but that position does not have Infiniband), the score will be infinite; if it's a perfect match (e.g., a rack_type requires Infiniband and that position has Infiniband), the score will be 0; if it's OK but not preferred (e.g., a rack_type does not need Infiniband and that position has Infiniband), the score will be a positive value. The tapbox score determines the usage of tapbox, a power unit shared between adjacent positions.

Tuning the weight of each term has been a challenge. To this end, we set different values and we rely on feedback from the different clients to adjust accordingly. In practice and over the years, this approach has worked well for our cases.

Our placement algorithm executes iteratively. In each iteration, it enumerates all the possibilities of moving one rack out

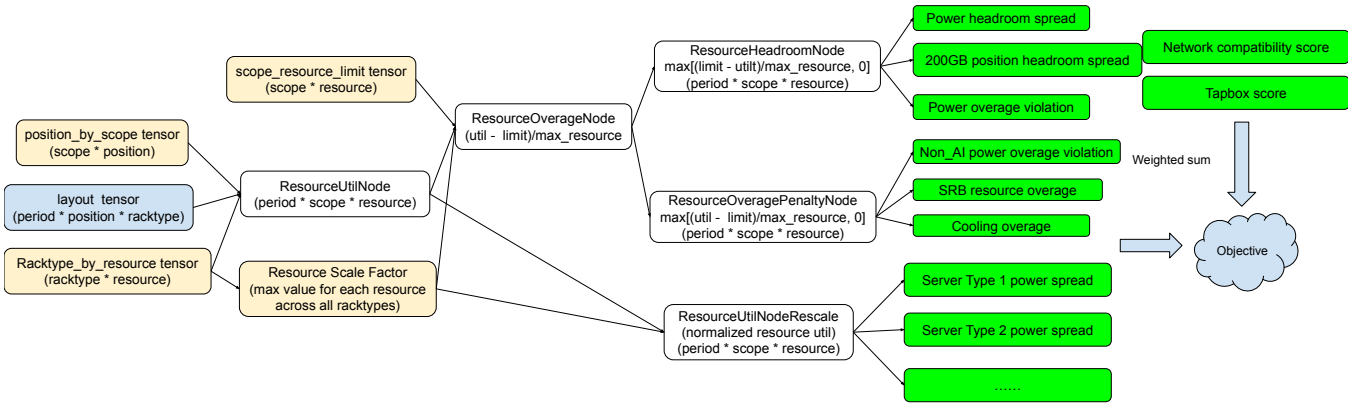


Figure 3: Objective function of hardware placement.

of its current position and into a different position; it adopts the pair of (source position, destination position) which yields the lowest value of the objective function. The algorithm terminates when the benefit is smaller than a threshold.

PIMS runs the placement algorithm monthly to optimize hardware placement. Every run takes about 25 minutes, including data preparation, running the placement algorithm mentioned above, and postprocessing. One run typically performs a few to a few hundred iterations before it hits the termination condition. Each iteration takes about one second. Relocation is executed independently with software maintenance, since the relocation team reserves a small buffer independent of b_{PIMS}. However, it needs to coordinate with hardware maintenance or decommission to avoid conflicts.

Note that reducing the buffer size is only one goal of this algorithm, represented by the third group of its objective terms (i.e., Server Type X power spread). It needs to consider many other factors, as expected in complex production systems.

In order for this whole idea to work efficiently, a fundamental assumption we make here is that different MDs within a region are balanced with similar power supply, network bandwidth, etc. Historically, Meta has incorporated new MD designs over time, but it only applies a new MD design in a new region to avoid unbalanced MDs within a region.

Service and Data Placement. Even when we distribute hardware evenly across the physical infrastructure in a DC, a software engineer could unknowingly deploy all instances of a service in a single MD, not only negating the benefits of balanced spread but also putting service availability under risk. Therefore, we provide placement libraries within our scheduler's capacity management system to ensure that all deployments are spread across MDs. These libraries have been presented in prior papers like Twine [27] and RAS [27] and thus are not the focus of this paper. Additionally, we operate ShardMananger [21], an advanced data management system which may run many shard replicas in a region. It is important to spread shards across all MDs in a region to avoid the case that all replicas of a shard are placed on a given MD, making that MD skewed.

When we move a rack to a new MD due to hardware placement, there is a potential for heterogeneous software configurations to exist within an MD, which may be a problem for certain services. To eliminate such risk, we re-image the servers in the rack to be moved to match those in the destination MD.

3.4 Multiplexing Domain Failure Buffer and Maintenance Buffer

PIMS uses one buffer b_{PIMS}, which has the size of an MD, to absorb loss caused by both domain failures and maintenance operations. To accomplish this, PIMS does not schedule new maintenance operations when an MD fails in the same region. However, the challenging case is that an MD fails *after* a maintenance operation already starts to run in the same region. In this case, we are losing two MDs, which exceed the capacity of b_{PIMS}. This section discusses our analysis and strategy about such risks.

In 2025, Meta has experienced four MD failures in total. They spread across three different regions and the longest aggregated MD failure time in one region is five hours. Considering Meta has tens of regions, many do not experience any MD failures. It should be noted that such failure rate is not going to grow as our fleet grows: PIMS manages maintenance in each region independently. Meta mainly grows its fleet by adding new regions, not by adding machines or MDs within a region. As a result, the MD failure rate per region is not going to grow as our fleet grows. In fact, the failure rate has been dropping over the years due to improved hardware and software reliability.

For physical maintenance, since it accounts for about 9% of the total operations, the chance for it to happen together with a MD failure is low. For a simple analysis, if maintenance happens during X% of the time and MD failure happens during Y% of the time, then they happen together during

$X\% \times Y\%$ of the time, assuming that they are independent. Following this calculation, physical maintenance and MD failure will happen together during $9\% \times \frac{5}{24\times365} = 0.005\%$ of the time, which translates to about 0.45 hours in one year. The actual number should be lower as 5 hours is the longest MD failure time among all regions, and PIMS will not schedule new maintenance operations when an MD failure happens.

For automated maintenance, which accounts for 91% of the total operations, the chance of a concurrent MD failure is not negligible (4.6 hours in one year using the above computation). To address this problem, when a MD fails during the automated maintenance, PIMS pauses maintenance and gives back the capacity back to this affected service as fast as possible. That is why PIMS sets an SLO in its contract that it should be able to return all capacity within 35 minutes, which strike a balance between effectiveness and feasibility. First, empirically, we find the 35-minute delay in returning capacity is generally acceptable in most cases. Second, we find the 35-minute delay is feasible: PIMS can naturally pause maintenance after one upgrade finishes. Many types of upgrades do not take long, but a few types can take significantly longer than average, often requiring multiple rounds of rebooting. For such long upgrades, we require the client who submits the request to split a long upgrade into multiple smaller ones. In addition, PIMS has a monitoring system, which will notify an on-call developer promptly if an upgrade gets stuck for unexpected reasons, and the developer can manually relinquish the capacity. Due to these optimizations, we have not seen a violation of the 35-minute SLO in the last five years.

Despite these strategies, the risk of concurrent failure still exists. In practice, a service may not be running at peak load when this happens, so it may be able to absorb such loss by itself. When the service is not able to absorb the loss, it can still turn to other capacity buffers (e.g., the buffer for major disasters) for temporary help, but this is very rare in our experience: in 2025, for each of the four MD failures mentioned above, we were able to relinquish capacity under 20 minutes and did not need the help of other buffers.

4 Implementation of Maintenance Stack

The overarching goal of PIMS is to implement fleet-wide maintenance in a *scalable* and *predictable* manner. To handle scalability, our system is region-based and operates maintenance decisions in parallel and independently across all regions. To be predictable, we employ a layered design composed of the Intent, Sequencing, Budgeting, Orchestration, and Execution layers. Each layer is expected to complete its goals according to its own SLOs towards the other layers. Collectively the layers handle the complexities of maintenance and provide rollout guarantees. Ultimately, maintenance decisions are performed in a region-based coordinated manner. Next, using Figure 4 we describe the steps taken by each layer to safely execute a maintenance request, referred to as the *maintenance workflow*.

First, maintenance clients (1) submit their requests referred to as *rollouts* to the appropriate rollout configuration systems (1)[1]. Clients include different teams and systems generating maintenance requests. For example, teams like the Kernel Team periodically release new versions of the kernel.

Rollout requests contain different attributes including the request type, such as kernel, firmware version upgrades, and the rollout scope specifying the set of machines to execute the rollout request at, such as an entire region, or machines matching a certain host prefix. Figure 4 shows two example rollout requests. The Rollout Service monitors the status of the rollout progress on a server basis for all rollouts. At any given time we have thousands of rollouts in progress across our fleet.

The Intent layer continuously checks new rollouts and produces a list of *intents* to declare the upgrade that is needed for every machine matching the scope criteria (3). It also maintains a list of all machines in the fleet with the state of their upgrades. The SLO of the Intent layer is to timely gather client rollout requests and to timely update the machine list with upgrade states within a few hours at the latest.

The Sequencing layer selects the next MD to schedule for maintenance and its SLO is to ensure at least one visit per MD in a cycle of a fixed number of days such as 30 or 45 (4). This layer also takes into account additional needs such as time constraints for physical maintenance, revisits due to failed upgrades, and more.

Given the next MD for maintenance, the Budgeting layer populates the list of intents to be executed during the visit of the next selected MD (5). This layer operates on a single MD visit at-a-time. Its SLO is that required upgrades for the assets of this MD are given enough time to complete at the next boxcar according to their tiers as in Table 1.

The Orchestration layer coordinates execution of individual maintenance operations in a way that is safe for the fleet and predictable for the maintenance stack (6). This involves creating a *clean execution environment* for every individual operation - ensuring that service workloads are migrated away (i.e. drained) from affected servers, and ordering execution to avoid interference between operations. The SLOs of this layer is about creating such an environment on time (e.g. 3h) to maintain predictable throughput. This is the only layer in the stack that is shared between planned, corrective, and fleet topology change maintenances, as it requires coherent view of all work in the DC to achieve its objective.

The Execution layer handles the execution of maintenances (7). The SLO of this layer is based on three metrics: success rate of the upgrader, duration of it and host health rate after upgrades, e.g., for kernel it's 99%, O(minutes) and 98% correspondingly. It is also responsible for identifying bad actors - maintenances that do not comply with the above requirements

[1] In this section, all (*) refer to Figure 4.

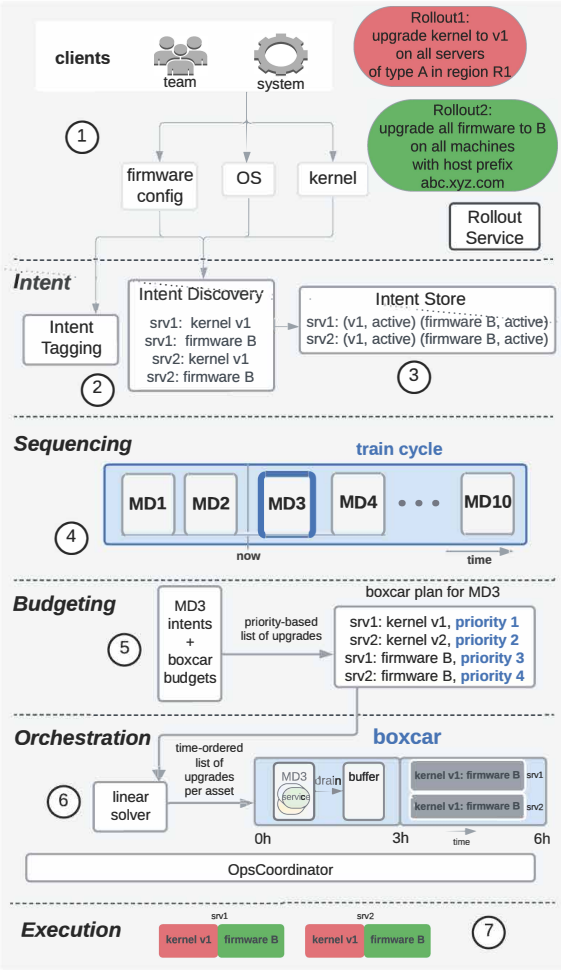


Figure 4: Maintenance stack overview.

- and providing a signal to remove them from the system to maximize utilization of available maintenance time.

Note that our system is not triggered by new rollout requests. Rather we keep two parallel threads of execution. The first keeps an up-to-date list of all upgrades per machine. The second is to pick the next MD for which the requested upgrades for all its assets are to be installed. These two threads run continuously. Despite the two asynchronous threads, our system relies on strict SLOs and performs efficiently and predictable as evaluation shows.

PIMS is backed up by a fault tolerant database to store incoming requests, intents, boxcar plans, etc. As a result, if PIMS process crashes for some reason, we just need to restart it, which will resume based on the information stored in the database.

4.1 Intent Layer

The Intent layer is composed of the *Intent Store*, the *Intent Discovery* and the *Intent Tagging* services to capture clients' rollout requests and to generate upgrade intents per machines.

The Intent Discovery service operates continuously to collect all *intents* from services (2) where an intent denotes the desired maintenance action to be performed on assets. Intents are stored in the Intent Store (3) which keeps the up-to-date state of all clients' intents. When an intent is published at the Intent Store, it is augmented with additional properties including *state*, *priority*, and more. An intent's state can be either *active* or *fulfilled*. All intents are initially active and remain so until fulfilled. An intent priority represents the urgency of an intent and can be used by Budgeting to onboard upgrades to boxcars created in emergency situations. Once an active intent is in the Intent Store, it signifies that its upgrade should be installed. Scheduling and running the upgrade of an active intent is done asynchronously by the layers below described in the next sections.

There are *gradual rollouts* deployed in a stepwise fashion (i.e., after the pre-production testing, production rollouts are deployed first to 5% and then to 25% and 100% of servers of the global scope) to ensure safe and controlled upgrades similar to rolling upgrades [15, 26]. For such upgrades it is challenging to find the specific servers to upgrade to meet rollout scope quotas when we upgrade an MD at-a-time. To this end, the Intent Tagging service aims to find specific assets in the fleet across MDs for gradual rollouts. For this, it greedily assigns assets at the upcoming MDs with the target version to ensure that the rollout will happen in the shortest time possible while ensuring efficiency without compromising safety.

The Intent layer monitors upgrade reattempts so that when an upgrade fails, an intent is kept in the active state until the corresponding upgrade successfully puts the server in its desired state. This also allows us to count the number of upgrade retries and so we can identify problems in cases of many unsuccessful attempts.

4.2 Sequencing Layer

The Sequencing layer determines the order to visit MDs in a train. As described in Section 3.2, it mostly follows the Least Recently Visited (LRV) order. This section discusses more details.

4.2.1 Failed maintenance domain visits

Every MD visit contains upgrades that belong to multiple rollouts and some of them might fail to execute. We need to measure upgrade failures in a boxcar to decide whether the MD should be rescheduled again with the remaining failed and other non-completed upgrades, referred to as a *MD revisit*. For this we define the success rate of a boxcar as the ratio between successful completed upgrades and the total number

of upgrades in the MD visit. If the success rate is below a threshold we schedule the MD to be revisited to retry failed upgrades. We can set this threshold with different values. According to our experience, an 80% threshold works well for our cases, because it achieves a good balance between success of individual rollouts versus the majority of rollouts that are executed in parallel in the same region.

Further, the Sequencing layer reschedules an MD after a failed visit. However, as a revisit requires an additional time slot there could be cases with not enough slots left to meet the 45-days SLO. To minimize this risk, the Sequencing layer monitors MD failures and in cases of subsequent failures of the same MD, it reschedules a visit after a configurable number of days in the future, which is similar to the exponential back-off mechanism. This enables us to continue to other MDs while issues in the problematic MD are being addressed.

4.2.2 Large-scale manual maintenance

In addition to running boxcars with automated upgrades, there is a need to schedule large-scale manual maintenance. Their scope is defined by power and network constraints, which can impact capacity maintained by different trains.

As discussed before, one of the scheduling constraints is that each capacity type under maintenance aggregated should not exceed an MD at-a-time. Because of this, once there is manual maintenance on a specific capacity type, we cannot perform any other maintenances on the scope of the same type. To solve this we place manual maintenances in a separate schedule called *manual train*. Trains are placed in order of priority, with the manual train being placed first so it does not face any conflicts and can schedule a boxcar on any day. When we place other trains, we search for conflicts with already placed boxcars. If any of the constraints are violated, the time slot of the boxcar of the train of less priority is skipped. In this way, automated trains are scheduled around manual trains.

The existence of manual trains could increase the chances of SLO violation, due to manual trains' high priority and restricted scheduling constraints (e.g., a human engineer typically only works during of a workday). Historically, we have alleviated this problem by careful planning. Today, we observe that manual trains have no big impact in PIMS.

4.2.3 Emergency rollouts

There are cases when a specific upgrade needs to be rolled out faster than the usual 45 or 30 days. For instance, assume that a new GPU driver has been rolled out to 50% of machines across MDs in a region when we observe significant performance degradation in the already upgraded machines. Instead of waiting for 45 days to recover the cluster we introduce *emergency rollouts* to roll back to the latest stable version.

There are two different cases: a) requiring upgrades are located in a few MDs; and b) a small number of machines require upgrades spread evenly across the region, for example only when a specific model is affected by buggy software. In the first case we use *emergency trains* that have the highest priority and are configured on demand. By placing emergency trains first, we can achieve automatic conflict resolution without manual intervention. The speed of the rollout by the emergency train is restricted by the number of MDs, drain time, and required work window. In the second case, for some capacity types that depend only on the number of machines available in the region (GPU training capacity), we can make a single visit that covers a few MDs and controls the number of machines on-boarded for maintenance. This approach can reduce the mitigation time from weeks to 1-2 days.

4.3 Budgeting Layer

Following the algorithm described in Section 3.2, The Budgeting layer takes all intents for every asset in an MD from the Intent Store and produces the *boxcar plan* which contains a priority list of upgrades for all assets that can run in the next visit of that MD.

This section reports on issues of safeguarding, fine-grain duration prediction and supercession. First, safeguarding ensures that all systems involved in upgrade operate correctly. To this end, we continuously monitor the reliability of workflows by checking for example whether an intent is successfully applied and the host is in a healthy and production-ready state after the upgrade. Once we detect an unreliable workflow, we stop its execution to investigate the problem. After the problem is fixed we then allow back the intent to boxcars.

Second, accurate workflow duration is important for the boxcar plan and to ensure workflow execution only when there is enough time left in the boxcar. We have developed an ML model to predict workflow duration based on historical data for workflows with high duration variation such as firmware upgrades [10].

Finally, there are cases where some of the workflows in higher priority tiers are applied out-of-the-box by other workflows in the lower priority tiers referred to as *supercession*. For example, OS reinstalling would automatically fetch and install the desired kernel version. To reduce duplication in boxcar plans, this layer further applies a step that detects and deduplicates supercession.

4.4 Orchestration and Execution Layers

The Orchestration layer coordinates the execution of all maintenance operations in the fleet, across categories, in a safe way for service workloads and predictable for maintenance operations. In the context of planned maintenances every train visit should start at a specific time and should finish within its predetermined duration for next MD visit. The OpsCoordinator system in Orchestration is responsible for guaranteeing that

timeline: services drain on time and that a batch of operations in a boxcar can reach one of the terminal states on time.

This layer achieves its objectives by combining fleet-wide safety invariants, service workload drains, time-order and topology awareness, and coordination between maintenances under a single user-facing abstraction. It is guaranteed that once OpsCoordinator makes a decision to start execution, maintenance will operate in a clean execution environment - service workloads are migrated away and no other concurrent maintenances are being executed on the same hardware.

Safety Invariants. In context of planned maintenances OpsCoordinator assesses safety across three dimensions to ensure that the maintenance contract is upheld at all times. First, it ensures that not more than one MD of the same type is taken down for maintenance in a region. Then, it verifies that there's enough buffer capacity to sustain the outage of the chosen MD. And finally, in case of a large sub-region failure event or buildup of organic failures, it will pause planned maintenance execution, and facilitate return of maintained capacity back to production so it can be used to absorb the impact.

Cross maintenance coordination. The main objective at this stage is to order incoming maintenance operations to avoid conflicts such that no two maintenances execute concurrently on the same device, as well as ensure that their execution is not impacted by utility loss. This stage coordinates across different categories of maintenances (software upgrades, unplanned operations, retrofits, etc). In context of planned maintenances this means that execution of software upgrades on devices with ongoing corrective work will be avoided, as well as some ongoing planned maintenances may be cancelled to facilitate cleanup across execution call stack to make room for higher-priority work.

Drain. The main goal at this stage is to gracefully migrate service workloads on all affected servers into buffers. Fully bootstrapping a workload on a new machine can take anywhere between a few seconds (for compute) to a few days (for storage). To avoid disruption all affected service workloads should be migrated to buffer capacity ahead of time. OpsCoordinator initiates and monitors the drain process. Our cloud manager Twine [31] provides a mechanism to ask services to vacate their host machines and also drives workload reallocation. OpsCoordinator is responsible for interfacing with this mechanism on time, and provides the required metadata about the drain. OpsCoordinator further resolves any conflicts due to hardware topologies.

In context of planned maintenances, maintaining predefined schedule of boxcar execution is important for rollout SLOs. This is a secondary goal during this stage, as it favours service availability first. According to the maintenance contract, all service workloads have to be able to migrate within 3h to maintain predictable schedule. OpsCoordinator is responsible for enforcing this expectation by forcefully preempting service workloads if they do not drain on time, apart from the scenarios when a large-scale infrastructure failure is observed.

For stateful services, due to the need to replicate or rebuild data, 3h drain may not be enough to gracefully migrate their workload. Therefore, they also receive an 24h advance notice prior to the maintenance train visit, and can utilize this signal to kick off data replication or rebuild.

Execution. Once the previous stages complete upgrades are ready for execution. OpsCoordinator requires all maintenance executors to implement a standardized service interface that is used to dispatch callbacks, monitor progress of the work, and inject control signals such as preemption. Once the executor reports on completed work, OpsCoordinator handles transparent return of the maintained hardware back to production. For planned maintenances, it is possible that not all operations from the boxcar finish by the end of the predefined work-window. As trains need to follow the schedule from Sequencing to maintain the rollout SLO, PIMS will release those servers that have finished maintenance, keep doing maintenance on those unfinished ones, and move to the next MD. Our statistics show that about 0.5% servers cannot finish maintenance by the end of the window, and we assume that the buffer to absorb random failures can handle such 0.5% capacity loss, since many unfinished maintenances are indeed caused by random failures.

5 Evaluation

In this section we report performance results of PIMS in production to maintain millions of servers in Meta's private cloud and thousands of rollouts. Section 5.1 reports the effectiveness of hardware placement on reducing buffer size. Section 5.2 evaluates predictability. In Section 5.3, we evaluate important SLOs of the maintenance stack. Section 5.4 evaluates the utilization of MDs and discusses related tradeoffs.

5.1 Buffer size reduction

As discussed in this paper, PIMS reduces buffer size in two ways. First, it uses one buffer to absorb the capacity loss caused by server maintenance, MD maintenance, and MD failures. If PIMS did not perform such multiplexing, the required buffer size may be trippled.

Second, PIMS applies the hardware placement algorithm to make MDs within a region balanced, because its buffer needs to accommodate the loss of the largest MD and thus balanced MDs help to reduce buffer size. Meta started to apply the hardware placement algorithm to reduce buffer size since 2022. We quantify its benefit with the size of b_{PMIS}, estimated using the approach described in Section 2.2.

Typically, there are more movement activities following either a significant change to the algorithm or an installation or decommission of many servers, and there are fewer movement activities when the situation gets stable. During the second quarter of 2025, which is a rather active period, we moved a total of 15,716 racks under the suggestion of the hardware

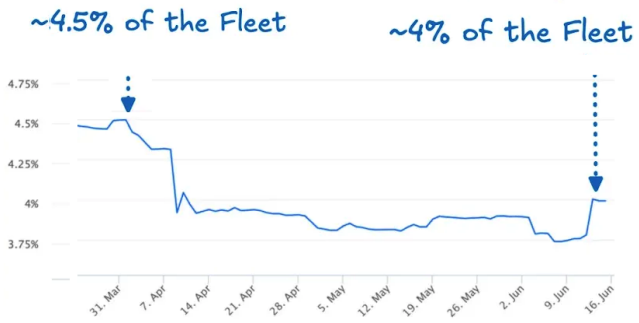


Figure 5: Reduction of buffer size due to hardware placement algorithm in 2025 Q2.

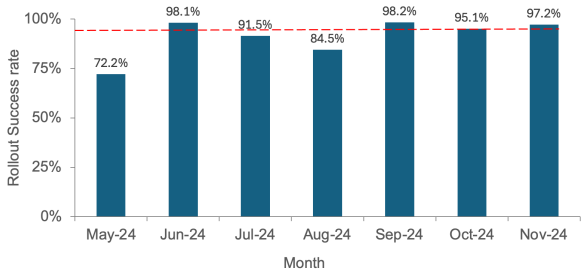


Figure 6: Rollout success rate.

placement algorithm. As shown in Figure 5, such activities have helped Meta to reduce the size of b_{PIMS} from about 4.5% of the fleet capacity to about 4%. Note that since the total fleet capacity was growing during this period, the actual reduction (15%) is larger than $\frac{4.5\%-4\%}{4.5\%}$. This can help us save tens of thousands of servers every year. As of May 2026, the size of b_{PIMS} has been reduced to about 3% of our fleet capacity.

Also note that, since Meta has been running the placement algorithm since 2022, 2025 Q2 already started with a reasonably optimized placement. If we applied this algorithm on a random placement plan, it is likely for this algorithm to achieve more reduction.

The problem of hardware placement is NP hard, so we do not know the optimal buffer size, but we can estimate its lower bound in the following way: if a region has n MDs, then the lower bound of the buffer size is $\frac{1}{n}$ of the region capacity, assuming that MDs are perfectly balanced. Most Meta regions have 36 to 72 MDs, which means that the lower bound is about 1.4% to 2.8%. However, due to different server types, different resources to consider, and different goals to optimize (Figure 3), it is unlikely that we can reach this lower bound. Our speculation is that there might be some room for further reduction, but probably not a significant one.

5.2 Maintenance predictability

Planned software and firmware upgrades. First, we evaluate maintenance predictability for planned software and firmware upgrades. We measure the *rollout success* defined as the ratio of rollouts completed according to their SLOs (see Table 1) on a monthly basis. The rollout success rate is weighted by the number of assets in the rollout scope to align with our goal to get as many assets as possible upgraded to the desired software or firmware version. In six months in 2024, we rolled out more than 300 different versions in Kernel, Firmware and OS categories through 52 upgrade types totaling to an average of 2-3K rollouts globally every month.

Figure 6 illustrates the rollout success rate over six months. Results show 90% or higher rollout success rates for four months. Success rate only drops for May (72%) and Aug (84%) because of failures in upgrade logging caused by a configuration update and a system bug in discovering intents. Once discovered, these issues were immediately addressed and rollout success rate was back again to very high values in the next months. Very high rollout success rates show that PIMS works in a predictable manner according to the SLOs.

To provide context on the importance to consistently meet SLOs consider *OS freshness*: 95% of our millions of servers have their OS provisioned every 45 days. In the past, before the current centralized maintenance system, we needed about 3 years for a fleet-wide OS upgrade. This corresponds to a 23x reduction in rollout time and the current 45-days SLO for OS provision ensures that we can maintain the latest OS version for fleet safety and compatibility.

Planned physical maintenance. Second, we evaluate predictability for planned physical maintenance in two cases. First, *fixed-date* maintenance is scheduled by DC engineers who provide the date and time to run such boxcars in the maintenance stack, which means the date is fixed for PIMS. Second, the *flexi-date* maintenance is scheduled by PIMS, which means the date is flexible for PIMS, and then DC engineers are given the date and time to run these boxcars. Depending on these two types of boxcars, we have two different contracts to ensure that we can start on time, and to limit the wait time.

First, we support the "start-on-time" SLO declared as *95% of both fixed- and flexi-date maintenance should start within 30 minutes of their scheduled time*. Figure 7 shows the percent of both boxcar types in the six months that successfully start within 30 min of their scheduled time. Results show that we are above 95% in most cases. Second, we support the "limit-the-wait-time" SLO as *99% of flexi-date manual maintenance operations should start within 90 days of request creation*. Figure 8 shows the percent of boxcars in the six months that successfully start within 90 days of request creation. Results show that we are compliant in 5 out of the past 6 months with this SLO. The two SLOs above were breached only to support the necessary hardware retrofit of network equipment in a flexible way which accounted for the delivery of parts and resources to fit them.

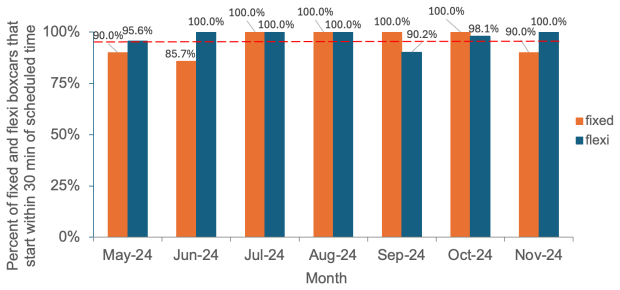


Figure 7: Percent of fixed- and flexi-date maintenances that start within 30 min of their scheduled time.

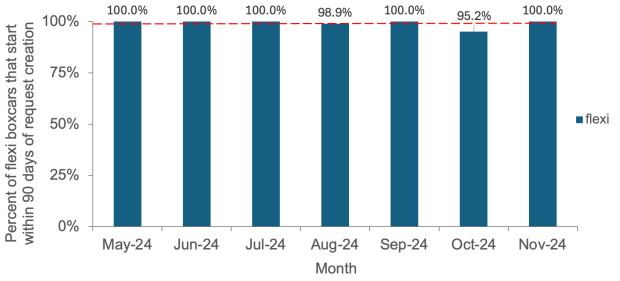


Figure 8: Percent of flexi-date maintenances that start within 90 days of request creation.

5.3 Maintenance layers SLOs

Maintenance predictability depends on how well each layer in the stack meets its own SLOs. Here, we show results on how well the Sequencing and Orchestration layers meet their SLOs in Table 2. The performance of these two layers is instrumental for the maintenance contract to ensure the 45-days train cycle and the total boxcar duration of 6h respectively.

First, Table 2 shows that over the six months, the compliance rate of the 45-days SLO is very high and consistently above 95%. Second, we evaluate the SLO compliance in boxcar for both the drain- and work-window. Table 2 shows that the enforcement rate which is defined as percent of drains that fail to finish in 3h and so enforced by OpsCoordinator is very low at 0.6%. Drain enforcement impact depends on the asset type and service and thus is processed differently; it is either automated or an oncall engineer is notified to handle this.

Third, Table 2 shows the number of breaches of the upgrade success rate, upgrade duration and host failure rate SLOs per million of upgrades in one month for the work-window. Note that SLOs differ across upgrade types. For example, the OsRefresh upgrade has a critical threshold of P80 duration of 50 min per boxcar. A violation of this SLO is a critical breach and requires immediate action, intervention and investigation. Results show that for most upgrade types breaches to these SLOs are rare with the exception of RDMAFirmware upgrades. Although RDMAFirmware has the smallest number of upgrades, it has the highest ratio of breaches likely due to a bug where a different, longer duration upgrade was incorrectly performed under the name of RDMAFirmware.

Sequencing 45 days SLO						
May	June	July	Aug.	Sept.	Oct.	Nov.
99.9%	99.1%	98.6%	98.2%	98.6%	98.9%	98.4%

Orchestration 3h drain SLO			
boxcar drain	P95 duration	P99 duration	enforcement
	77 min	175 min	0.6%

Boxcar 3h execution SLO				
upgrade type	total upgrades per month	no of breaches/1M upgrades in		
		success rate	duration	host failure
RDMAFirmware	46,174	22	282	0
FbossForwardingStack	1,117,445	7	4	1
GenericPWMProvision	260,461	4	0	0
OsMclassicaRefresh	4,579,434	1	0	0
OsRefresh	70,201,856	0	1	0
LemonaidChecker	51,527,866	0	0	0
HostProfilePWM	90,061,861	0	0	0
Firmware	23,945,836	0	0	0

Table 2: Results on meeting maintenance SLOs.

5.4 Maintenance overhead and efficiency

The overall efficiency of our system is reflected at the speed at which we can execute rollouts while maintaining predictability. Rollout speed depends on how fast the train schedule can visit all MDs in a region which depends on the duration of a boxcar visit. We have used different drain-window and work-window durations from 1 to 2 and now to 4 boxcars/day.

To explore whether we can further reduce boxcar duration, we measure the drain- and work-window utilization in one month across all boxcars and regions. The drain-window utilization is 7% on average and 23% for the work-window. This data shows that most drain and upgrades finish early and their assets wait idle until the end of the windows. However, as shown in Table 2 the P95 duration of drains is 77min and the P99 duration is 175min (almost 3h) indicative of a long tail. Similar distributions are observed in the work-window. Further reducing the boxcar duration will increase the enforcement rate and the number of unfinished upgrades which will adversely impact rollout success. Beginning upgrade as soon as a drain completes is possible for automated upgrades, but not possible for those requiring manual effort, since engineer time must be scheduled in advance.

6 History, lesson, and future

In early days, maintenance was performed in isolation by different service teams, which fully managed the allocation and usage of their own buffers. It became challenging with our fleet growth, as we often needed to coordinate multiple maintenances due to resource and reliability reasons. This led to the development of PIMS as a centralized maintenance system in 2019 and the first major deployment in 2021.

As for the major components or ideas of PIMS, we have decided to use a single buffer since the beginning and incorporated the hardware placement algorithm to reduce buffer size in 2022. We created a maintenance contract at the beginning, but it only included the expectations that maintenance

will take down one MD and we will reserve a buffer for that. We added the SLO expectations for different components and teams (i.e., 3 hours for draining and 35 minutes for relinquishing capacity during unplanned faults) in December 2021. Through the years, we have tested different boxcar durations and settled with our current setting.

Our most important lesson is that the maintenance contract is important to the success of PIMS, especially in a hyper-scale environment like Meta. Maintenance is a typical example of tasks that require close collaboration among multiple teams. By clearly defining the responsibility of each team, the contract allows each team to focus on its own goal and achieve the maintenance SLOs with a collective effort.

The recent development of AI clusters, especially the training clusters, has created new challenges to maintenance but it has also simplified maintenance in several other aspects.

1. Training clusters are mostly homogeneous, with the same type of servers and GPUs, so we do not need the hardware placement algorithm to balance MDs.
2. Training is not servicing user traffic, so it can tolerate capacity adjustment, as long as it is planned carefully.
3. Training systems provide fault tolerance capability through mechanisms like checkpoint, so PIMS could interrupt them for maintenance, though the interruption is costly for large-scale training like LLM pre-training [29].
4. The new generations of GPUs allow direct memory access from GPU to GPU across different hosts, which introduces a new kind of network domain. These network domains do not necessarily align with power domains.

In general, smaller maintenance domain means smaller buffer and thus more capacity for training in the common case, but it also means more frequent maintenances and interruptions, which lead to a higher interruption cost. Considering 3 and 4, we currently settle with performing maintenance at the granularity of the new network domains, which are smaller than power domains, for automated upgrades. For physical maintenances that need to take down power domains, we perform them less frequently and take out additional capacity, which is acceptable due to 2. We also try to avoid a physical maintenance during large-scale LLM pre-training. However, due to the fast evolvement of both training software and hardware, our design is subject to changes in the future.

7 Related work

To our best knowledge, this paper is the first to discuss fleet-wide physical maintenance and software upgrades. More broadly, many works discuss the upgrades and maintenance of specific systems, such as ML Supercomputer [34], network devices [2, 3, 14, 23, 25], confidential virtual machines [7], and application-level software [9, 15, 17, 26, 33]. While PIMS shares some of their challenges and design principles, PIMS faces a unique challenge that it has to consider and optimize hardware placement to achieve its goal of minimizing capacity buffer, which leads to our design choices to align maintenance with failure domains and to physically relocate machines to achieve even spread. In addition, as a centralized scheduler, PIMS can schedule many upgrades at the same time, creating opportunities for improving efficiency, while most prior work consider one upgrade at a time.

There are works targeting specific problems during upgrades. For example, Azure's Gandalf focuses on early capturing bad rollouts to prevent widespread outages [22]. Other works focus on classifying upgrades as sources of failures [14, 16, 18, 33]. We also find related work on reducing downtime during upgrade even without migrating running services [5, 6, 13, 19, 26, 28, 30] and tailored approaches to reduce drain time [4]. Finally, Hyrax [24] allows partially-failed servers to be used while waiting for maintenance. These works are largely orthogonal to our work, since our work may apply similar approaches to reduce the chance of failures or reduce down time, though in our experience, these works often work only for specific systems.

8 Conclusions

PIMS achieves small capacity buffer and predictable maintenance at the scale of millions of servers, with a collective effort of clear contract to define the expectation for each participant, optimized hardware placement, SLO-aware prioritized scheduling, and buffer sharing motivated by failure statistics. PIMS has been in production over the last five years in Meta's hyperscale private cloud managing planned maintenance such as kernel and firmware updates. Production data shows that it can significantly reduce capacity buffer size, which yields an annual saving of tens of thousands of servers, and can achieve a reasonable SLO for regular maintenance tasks.

Acknowledgments

This work would not have been possible without the sustained collaboration across many teams at Meta, including Capacity, Core Infra, Datacenter Operations, Network, Production Engineering, Site Operations, and many others. We thank those who have contributed to PIMS: Justin Boseant, Philip Harries, Scott Michelson, David Steere, Ivan Santa Maria Filho, Brian Spaulding, Grigory Turunov, Usman Sharif, Sajal Jain, Yang Xia, Ivan Cikic, Alexander Opeykin, Igor Tufanov, Richa Kala, Victoria Puscas, Maxim Kuthorenko, Andrei Darabanov, Artemiy Kolesnikov, Catherine Sweeney, Aileen McCabe, Konstantin Startsev, Cedric Van Goethem, Sahil Deshpande, Max Khodak, Michal Sedlak, Saranyan A Vigraham, Sudheer Vaddi, Surupa Biswas, and CQ Tang. We also thank all reviewers for their insightful comments.

References

[1] Atul Adya, Daniel Myers, Jon Howell, Jeremy Elson, Colin Meek, Vishesh Khemani, Stefan Fulger, Pan Gu, Lakshminath Bhuvanagiri, Jason Hunter, Roberto Peon, Larry Kai, Alexander Shraer, Arif Merchant, and Kfir Lev-Ari. Slicer: Auto-Sharding for Datacenter Applications. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI'16, page 739–753, USA, 2016. USENIX Association.

[2] Mohammad Al-Fares, Virginia Beauregard, Kevin Grant, Angus Griffith, Jahangir Hasan, Chen Huang, Quan Leng, Jiayao Li, Alexander Lin, Zhuotao Liu, Ahmed Mansy, Bill Martinusen, Nikil Mehta, Jeffrey C. Mogul, Andrew Narver, Anshul Nigham, Melanie Obenberger, Sean Smith, Kurt Steinkraus, Sheng Sun, Edward Thiele, and Amin Vahdat. Change Management in Physical Network Lifecycle Automation. In *2023 USENIX Annual Technical Conference (USENIX ATC 23)*, pages 635–653. USENIX Association, July 2023.

[3] Omid Alipourfard, Jiaqi Gao, Jeremie Koenig, Chris Harshaw, Amin Vahdat, and Minlan Yu. Risk based Planning of Network Changes in Evolving Data Centers. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP '19, page 414–429. Association for Computing Machinery, 2019.

[4] George Amvrosiadis, Angela Demke Brown, and Ashvin Goel. Opportunistic Storage Maintenance. In *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP '15, page 457–473. Association for Computing Machinery, 2015.

[5] Jeff Arnold and M. Frans Kaashoek. Ksplice: Automatic Rebootless Kernel Updates. In *Proceedings of the 4th ACM European Conference on Computer Systems*, EuroSys '09, page 187–198. Association for Computing Machinery, 2009.

[6] Andrew Baumann, Jonathan Appavoo, Robert W Wisniewski, Dilma Da Silva, Orran Krieger, and Gernot Heiser. Reboots Are for Hardware: Challenges and Solutions to Updating an Operating System on the Fly. In *USENIX Annual Technical Conference*, pages 337–350, 2007.

[7] Jiahao Chen, Zeyu Mi, Yubin Xia, Haibing Guan, and Haibo Chen. CPC: Flexible, Secure, and Efficient CVM Maintenance with Confidential Procedure Calls. In *2024 USENIX Annual Technical Conference (USENIX ATC 24)*, pages 1065–1082. USENIX Association, July 2024.

[8] Arnab Choudhury, Yang Wang, Tuomas Pelkonen, Kutta Srinivasan, Abha Jain, Shenghao Lin, Delia David, Siavash Soleimanifard, Michael Chen, Abhishek Yadav, Ritesh Tijoriwala, Denis Samoylov, and Chunqiang Tang. MAST: Global scheduling of ML training across Geo-Distributed datacenters at hyperscale. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, pages 563–580, Santa Clara, CA, July 2024. USENIX Association. URL: https://www.usenix.org/conference/osdi24/presentation/choudhury.

[9] Olivier Crameri, Nikola Knezevic, Dejan Kostic, Ricardo Bianchini, and Willy Zwaenepoel. Staged Deployment in Mirage, an Integrated Software Upgrade Testing and Distribution System. In *Proceedings of Twenty-First ACM SIGOPS Symposium on Operating Systems Principles*, SOSP '07, page 221–236. Association for Computing Machinery, 2007.

[10] Yi Ding, Aijia Gao, Thibaud Ryden, Michal Sedlak, Essam Ewaisha, Igor Marnat, and Henry Hoffmann. Cost-aware Duration Prediction for Software Upgrades in Datacenters. In *Proceedings of the MLSys 2026 Conference*, 2026.

[11] Marius Eriksen, Kaushik Veeraraghavan, Yusuf Abdulghani, Andrew Birchall, Po-Yen Chou, Richard Cornew, Adela Kabiljo, Ranjith Kumar S, Maroo Lieuw, Justin Meza, Scott Michelson, Thomas Rohloff, Hayley Russell, Jeff Qin, and Chunqiang Tang. Global Capacity Management With Flux. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*, pages 589–606, Boston, MA, July 2023. USENIX Association.

[12] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The Google File System. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, SOSP '03, page 29–43, New York, NY, USA, 2003. Association for Computing Machinery.

[13] Cristiano Giuffrida, Anton Kuijsten, and Andrew S. Tanenbaum. Safe and Automatic Live Update for Operating Systems. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '13, page 279–292. Association for Computing Machinery, 2013.

[14] Ramesh Govindan, Ina Minei, Mahesh Kallahalla, Bikash Koley, and Amin Vahdat. Evolve or Die: High-Availability Design Principles Drawn from Googles Network Infrastructure. In *Proceedings of the 2016 ACM SIGCOMM Conference*, SIGCOMM '16, page 58–72. Association for Computing Machinery, 2016.

[15] Boris Grubic, Yang Wang, Tyler Petrochko, Ran Yaniv, Brad Jones, David Callies, Matt Clarke-Lauer, Dan Kel-

ley, Soteris Demetriou, Kenny Yu, and Chunqiang Tang. Conveyor: One-Tool-Fits-All continuous software deployment at meta. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*, pages 325–342, Boston, MA, July 2023. USENIX Association.

[16] Haryadi S. Gunawi, Mingzhe Hao, Riza O. Suminto, Agung Laksono, Anang D. Satria, Jeffry Adityatama, and Kurnia J. Eliazar. Why Does the Cloud Stop Computing? Lessons from Hundreds of Service Outages. In *Proceedings of the Seventh ACM Symposium on Cloud Computing*, SoCC '16, page 1–16. Association for Computing Machinery, 2016.

[17] Hai Huang, Salman Baset, Chunqiang Tang, Ashu Gupta, K N Madhu Sudhan, Fazal Feroze, Rajesh Garg, and Sumithra Ravichandran. Patch Management Automation for Enterprise Cloud. In *2012 IEEE Network Operations and Management Symposium*, pages 691–705, 2012.

[18] Lexiang Huang, Matthew Magnusson, Abishek Bangalore Muralikrishna, Salman Estyak, Rebecca Isaacs, Abutalib Aghayev, Timothy Zhu, and Aleksey Charapko. Metastable Failures in the Wild. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 73–90. USENIX Association, 2022.

[19] Sanidhya Kashyap, Changwoo Min, Byoungyoung Lee, Taesoo Kim, and Pavel Emelyanov. Instant OS Updates via Userspace Checkpoint-and-Restart. In *2016 USENIX Annual Technical Conference (USENIX ATC 16)*, pages 605–619. USENIX Association, June 2016.

[20] Neeraj Kumar, Pol Mauri Ruiz, Vijay Menon, Igor Kabiljo, Mayank Pundir, Andrew Newell, Daniel Lee, Liyuan Wang, and Chunqiang Tang. Optimizing Resource Allocation in Hyperscale Datacenters: Scalability, Usability, and Experiences. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, pages 507–528, Santa Clara, CA, July 2024. USENIX Association.

[21] Sangmin Lee, Zhenhua Guo, Omer Sunercan, Jun Ying, Thawan Kooburat, Suryadeep Biswal, Jun Chen, Kun Huang, Yatpang Cheung, Yiding Zhou, Kaushik Veeraraghavan, Biren Damani, Pol Mauri Ruiz, Vikas Mehta, and Chunqiang Tang. Shard Manager: A Generic Shard Management Framework for Geo-distributed Applications. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, SOSP '21, page 553–569. Association for Computing Machinery, 2021.

[22] Ze Li, Qian Cheng, Ken Hsieh, Yingnong Dang, Peng Huang, Pankaj Singh, Xinsheng Yang, Qingwei Lin, Youjiang Wu, Sebastien Levy, and Murali Chintalapati. Gandalf: An Intelligent, End-To-End Analytics Service for Safe Deployment in Large-Scale Cloud Infrastructure. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 389–402. USENIX Association, February 2020.

[23] Bingzhe Liu, Colin Scott, Mukarram Tariq, Andrew Ferguson, Phillipa Gill, Richard Alimi, Omid Alipourfard, Deepak Arulkannan, Virginia Jean Beauregard, Patrick Conner, P. Brighten Godfrey, Xander Lin, Joon Ong, Mayur Patel, Amr Sabaa, Arjun Singh, Alex Smirnov, Manish Verma, Prerepa V Viswanadham, and Amin Vahdat. CAPA: An Architecture For Operating Cluster Networks With High Availability. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*, pages 1995–2010. USENIX Association, April 2024.

[24] Jialun Lyu, Marisa You, Celine Irvene, Mark Jung, Tyler Narmore, Jacob Shapiro, Luke Marshall, Savyasachi Samal, Ioannis Manousakis, Lisa Hsu, Preetha Subbarayalu, Ashish Raniwala, Brijesh Warrier, Ricardo Bianchini, Bianca Schroeder, and Daniel S. Berger. Hyrax: Fail-in-Place Server Operation in Cloud Platforms. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*, pages 287–304. USENIX Association, July 2023.

[25] Ajay Mahimkar, Carlos Eduardo de Andrade, Rakesh Sinha, and Giritharan Rana. A Composition Framework for Change Management. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*, SIGCOMM '21, page 788–806. Association for Computing Machinery, 2021.

[26] Iulian Neamtiu and Tudor Dumitraş. Cloud Software Upgrades: Challenges and Opportunities. In *2011 International Workshop on the Maintenance and Evolution of Service-Oriented and Cloud-Based Systems*, pages 1–10, 2011.

[27] Andrew Newell, Dimitrios Skarlatos, Jingyuan Fan, Pavan Kumar, Maxim Khutornenko, Mayank Pundir, Yirui Zhang, Mingjun Zhang, Yuanlai Liu, Linh Le, Brendon Daugherty, Apurva Samudra, Prashasti Baid, James Kneeland, Igor Kabiljo, Dmitry Shchukin, Andre Rodrigues, Scott Michelson, Ben Christensen, Kaushik Veeraraghavan, and Chunqiang Tang. RAS: Continuously Optimized Region-Wide Datacenter Resource Allocation. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, SOSP '21, page 505–520. Association for Computing Machinery, 2021.

[28] Shaya Potter and Jason Nieh. Reducing Downtime Due to System Maintenance and Upgrades. In *19th Large*

Installation System Administration Conference (LISA 05). USENIX Association, December 2005.

[29] Omkar Salpekar, Rohan Varma, Kenny Yu, Vladimir Ivanov, Yang Wang, Ahmed Sharif, Min Si, Shawn Xu, Feng Tian, Shengbao Zheng, Tristan Rice, Ankush Garg, Shangfu Peng, Shreyas Siravara, Wenyin Fu, Rodrigo de Castro, Adithya Gangidi, Andrey Obraztsov, Sharan Narang, Sergey Edunov, Maxim Naumov, Chunqiang Tang, and Mathew Oldham. Training LLMs with Fault Tolerant HSDP on 100,000 GPUs, 2026. URL: https://arxiv.org/abs/2602.00277, arXiv:2602.00277.

[30] Maxim Siniavine and Ashvin Goel. Seamless Kernel Updates. In *2013 43rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 1–12, 2013.

[31] Chunqiang Tang, Kenny Yu, Kaushik Veeraraghavan, Jonathan Kaldor, Scott Michelson, Thawan Kooburat, Aravind Anbudurai, Matthew Clark, Kabir Gogia, Long Cheng, Ben Christensen, Alex Gartrell, Maxim Khutornenko, Sachin Kulkarni, Marcin Pawlowski, Tuomas Pelkonen, Andre Rodrigues, Rounak Tibrewal, Vaishnavi Venkatesan, and Peter Zhang. Twine: A Unified Cluster Management System for Shared Infrastructure. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 787–803. USENIX Association, November 2020.

[32] Kaushik Veeraraghavan, Justin Meza, Scott Michelson, Sankaralingam Panneerselvam, Alex Gyori, David Chou, Sonia Margulis, Daniel Obenshain, Shruti Padmanabha, Ashish Shah, Yee Jiun Song, and Tianyin Xu. Maelstrom: Mitigating Datacenter-level Disasters by Draining Interdependent Traffic Safely and Efficiently. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 373–389. USENIX Association, October 2018.

[33] Yongle Zhang, Junwen Yang, Zhuqi Jin, Utsav Sethi, Kirk Rodrigues, Shan Lu, and Ding Yuan. Understanding and Detecting Software Upgrade Failures in Distributed Systems. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, SOSP '21, page 116–131. Association for Computing Machinery, 2021.

[34] Yazhou Zu, Alireza Ghaffarkhah, Hoang-Vu Dang, Brian Towles, Steven Hand, Safeen Huda, Adekunle Bello, Alexander Kolbasov, Arash Rezaei, Dayou Du, Steve Lacy, Hang Wang, Aaron Wisner, Chris Lewis, and Henri Bahini. Resiliency at Scale: Managing Google's TPUv4 Machine Learning Supercomputer. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*, pages 761–774. USENIX Association, April 2024.

Heterogeneity at Hyperscale: Characterization and Scheduling of Large Production AI Clusters at Alibaba (Operational Systems)

Suyi Li[*†], Lingyun Yang[*†‡], Haoxuan Yu[†], Sheng Yao[†], Tianyuan Wu[†], Xiaoxiao Jiang[†], Hanfeng Lu[†], Kangjin Wang[‡], Chenhao Wang[◇], Shenglin Xu[‡], Lun Wang[‡], Qingyang Duan[‡], Shenghao Liang[‡], Xiu Lin[‡], Meng Zhang[‡], Wenchao Wu[‡], Yinghao Yu[‡], Guodong Yang[‡], Liping Zhang[‡], Wei Wang[†]

[†]*HKUST* [‡] *Alibaba Group* [◇] *Fudan University*

{slida, weiwa}@cse.ust.hk, yanglingyun.yly@alibaba-inc.com

Abstract

The rapid scaling of generative AI (GenAI), alongside the continued reliance on classical deep neural networks (DNNs), has pushed production AI infrastructure toward massive, heterogeneous GPU fleets. We present a comprehensive characterization of Alibaba Serverless Infrastructure (ASI), a hyperscale production cluster, based on a six-month trace covering 155,410 GPUs of multiple vendors and generations and jobs from 81 departments, spanning ad-hoc development, training, and online and offline inference. Our central finding is that high GPU demand does not yield high effective utilization: idle GPUs frequently become unallocatable because free capacity is stranded across nodes, lacks matching CPUs, or violates network-locality constraints, and because users reserve ample headroom for production safety. Notably, fractional-GPU fragmentation, a focus of prior work, is now negligible, as GPU sharing is rarely used. We detail deployed solutions that recover this capacity: a practical GPU defragmentation algorithm that cuts the number of nodes with slack resources by 20.2%, and SpotGPU, a preemption-cost-aware scheduling framework that safely harvests idle resources and raises the GPU allocation ratio from 68% to 93%. We further surface open challenges in skewed multi-vendor GPU adoption, bandwidth bottlenecks between heterogeneous GPUs, and interference among colocated workloads. We release the ASI trace, the most comprehensive to date in workload diversity and cluster scale, to support future research.

1 Introduction

Production AI clusters are no longer built for a single class of ML workload. Large generative AI models have driven rapid growth in GPU demand, especially for LLMs and image/video generation [1, 7, 12, 57, 58, 62]. At the same time, classical DNNs and recommendation models remain central to latency-critical business services [5,51,59,66]. These workloads differ in model size, execution time, priority, resource shape, and sensitivity to communication and interference. As a result, modern production clusters must manage large GPU fleets as high-value shared infrastructure for heterogeneous workloads that are *first-class* citizens in production.

This paper studies Alibaba Serverless Infrastructure (ASI), a hyperscale shared production GPU cluster. We analyze a six-month trace from ASI that covers 155,410 GPUs and jobs submitted by 81 internal departments. The trace spans ad-hoc development, training, online inference, and offline inference, and includes LLMs [12,57,58], generative image/video models [1,6,7,25,41], DNNs [5,51], and recommendation models [59,66]. Each job specifies the GPUs, CPUs, memory, and replicas it needs, which the cluster uses for allocation and billing. Unlike prior traces that mainly study homogeneous GPU fleets [16] or dated hardware generations [15,51], our trace captures a recent heterogeneous fleet with multiple GPU generations and vendors (Table 1).

The central lesson from ASI is that high demand does not by itself imply high effective utilization. Our ASI cluster represents a large financial investment, so even a 1% utilization change has substantial operational impact. Yet idle GPUs can remain *unutilizable* because free resources are split across the wrong nodes, lack matching CPU capacity, violate network-locality requirements, or are held as headroom for time-varying production demand. Thus, the main resource-management problem is not simply to fill idle GPUs, but to match heterogeneous workload demands to heterogeneous hardware under topology, priority, and interference constraints. Our study follows this problem from measurement to practice: we first characterize why resources become unusable (*fragmentation*), then describe deployed mechanisms that recover part of the lost capacity, and finally identify remaining challenges that still require new system support.

Diagnosing GPU fragmentation. We first show that GPU fragmentation is a primary reason that idle GPUs become unallocatable in ASI [9,43,52]. Fragmentation arises from several sources: fractional GPU allocations, stranded GPUs left on multi-GPU nodes, insufficient CPU capacity on otherwise available GPU nodes, and network topology constraints. Although prior work has discussed the first three sources [52],

[*]Equal contributions.

our trace shows that fractional GPU fragmentation is now a minor contributor in ASI because GPU sharing is rarely used. Instead, stranded GPUs, CPU bottlenecks, and topology constraints dominate. The topology effect is especially important for large GenAI training and serving jobs, which rely on parallelism strategies [34] and therefore need high-bandwidth placement across multiple GPU nodes.

To mitigate fragmentation, we developed IPC (iterative partitioned consolidation), a practical defragmentation algorithm for hyperscale GPU clusters. IPC uses fast partitioned search and ejection-chain scheduling to generate migration decisions that respect affinity rules and locked tasks. We also introduce an entropy-based metric for topology-aware allocation, which prefers placements concentrated within fewer access switches when possible (§4.1). In replay experiments, IPC makes migration decisions within minutes and reduces the number of nodes with slack resources by 20.2%.

Mitigating resource underutilization. We next show that production reservations create a second source of lost capacity. Users reserve capacity for diurnal traffic, failover redundancy, and peak-event headroom, leaving many GPUs idle during off-peak periods. Our trace shows that the peak amount of idle standby capacity can reach 10,000 GPU hours around midnight. This problem cannot be solved by ordinary best-effort packing alone: production jobs must retain guaranteed availability, while flexible jobs should use the idle capacity only when doing so does not endanger high-priority workloads.

ASI addresses this tension through two priority classes and a deployed preemptive scheduling framework called *SpotGPU*. High-Priority (HP) jobs receive guaranteed resources and can reclaim them from Low-Priority (LP) spot jobs, which run at a discounted rate on flexible capacity. Unlike prior workload characterizations, our trace includes job priority metadata, allowing us to study this policy directly. SpotGPU combines a "*Standby*" mechanism, through which HP users explicitly release temporarily idle resources, with a preemption-cost-aware scheduler that places LP jobs while minimizing wasted work when HP jobs reclaim capacity. Together, these mechanisms harvest idle resources while preserving HP resource availability (§4.2).

Understanding remaining heterogeneity challenges. Even with deployed mechanisms for fragmentation and underutilization, several problems remain open because they stem from deeper heterogeneity in hardware, network topology, and workload behavior.

First, ASI uses GPUs from multiple vendors to reduce supply-chain risk, but users initially adopted XPU-A more slowly than NVIDIA GPUs. Although XPU-A has stronger theoretical specifications than NVIDIA H20, its out-of-the-box GenAI performance was only 80% of H20 because key kernels did not yet match the hardware's execution patterns. Our preliminary XPU-A optimizations for LLM serving improve performance by up to 43% and increase HP demand for XPU-A by 2.5×, but supporting diverse model architectures across heterogeneous accelerators remains an open problem.

Second, hardware heterogeneity interacts poorly with network locality. Systems can use heterogeneous GPUs to match different phases of a workload to different hardware capabilities [14, 18, 33, 39, 61, 65]. In ASI, however, heterogeneous GPUs are often located under different access switches (ASW); in our benchmark, same-ASW placement improves `allreduce` bandwidth by 27% relative to cross-ASW placement. For communication-sensitive workloads such as Prefill-Decode (PD) disaggregation [14, 39, 65], the scheduler must balance hardware specialization against the cost of placing communicating instances further apart.

Third, online inference leaves substantial room for co-location, but current isolation mechanisms do not yet make this opportunity easy to exploit. Online inference has a median GPU SM utilization of only 6%. GenAI serving often exhausts GPU memory while leaving compute resources idle, whereas classical DNN serving tends to underutilize both memory and compute. Although co-locating these workloads could improve efficiency, existing GPU sharing techniques such as static partitioning [36] and time-sharing [2, 53, 56, 60] either partition resources too rigidly or fail to provide the latency isolation needed by online services. Similarly, ASI co-locates CPU-only jobs with GPU jobs to harvest idle CPU cycles, but this can reduce the P90 SM utilization of GPU training workloads by 18%.

We believe the observations from ASI are useful beyond our deployment because they expose resource-management problems that arise when large-scale AI infrastructure combines heterogeneous workloads, accelerators, priorities, and topologies. By releasing the sanitized ASI trace[1] and sharing the mechanisms and open challenges from our production experience, we hope to support follow-up research on more efficient GPU cluster management.

2 Background

ASI cluster. ASI is a shared, multi-tenant GPU cluster that provides infrastructure services to dozens of internal departments whose businesses span e-commerce, advertising, local services, logistics, fintech, and content platforms. These departments rent compute resources from ASI to develop and deploy ML services without operating separate GPU clusters. To ensure fairness in resource allocation, ASI allocates resources *on a quota basis*: a tenant cannot request more resources than its assigned quota. Since its inception in 2022, ASI has supported critical business operations across these departments.

Fig. 1 shows the cluster organization. ASI adopts a fat-tree network [26, 40] in which 8-GPU nodes attach to access

[1] https://github.com/alibaba/clusterdata/tree/master/cluster-trace-gpu-v2026

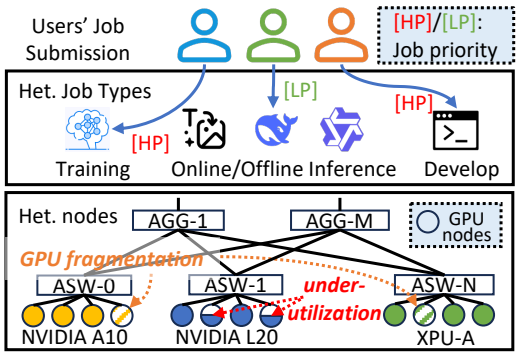


Figure 1: Overview of ASI as a shared GPU infrastructure service. **ASW**: access switch; **AGG**: aggregation switch.

switches (ASW) that are in turn interconnected through aggregation switches (AGG). Each ASW aggregates 32–64 nodes, so a single ASW domain spans roughly 256–512 GPUs, and the GPUs under one ASW are homogeneous. Because the network is not fully non-blocking, a job placed within one ASW communicates at full bandwidth, whereas a job spread across ASWs traverses additional switch layers and loses effective bandwidth. This topology directly shapes placement: communication-sensitive jobs request ASW-local allocations, which in turn constrains how freely the scheduler can pack the cluster. We quantify this effect on fragmentation in §4.1.

Workload scope. ASI must serve two workload classes at once: rapidly growing GenAI models and mature classical DNNs. GenAI models such as LLMs and diffusion-based image/video generation [1, 12, 48, 57, 58] can claim very large allocations; the largest single GenAI job in our trace used over 2,000 high-end GPUs simultaneously and consumed more than 800,000 GPU hours. Classical DNNs remain central to high-throughput production services such as Click-Through Rate (CTR) prediction, recommendation, and optical character recognition (OCR) [51, 59, 66]: they still account for roughly 70% of online-inference GPU hours in our trace (Fig. 3). Co-locating these classes forces ASI to manage workloads with sharply different resource shapes, execution times, priorities, and placement constraints. The trace covers large-scale training and inference in this shared cluster, but *excludes the pre-training of hyperscale foundation models*, which typically runs on dedicated clusters with cutting-edge hardware [47].

Trace. ASI records the metadata needed to study these production resource-management problems, including job types, priorities, model information, hardware types, utilization metrics, and network topology that prior public traces do not expose together. As shown in Table 1, the ASI trace surpasses prior studies by up to two orders of magnitude in both job volume and cluster size, while spanning a far more heterogeneous GPU fleet. These fields connect the workload characterization in §3.1 to the later analyses of fragmentation (§4.1), priority-based resource harvesting (§4.2), and hardware/topology heterogeneity (§5).

Trace	ASI	Acme [16]	PAI [51]	Helios [15]
Year	2025	2023	2020	2020
Duration	6 months	6 months	2 months	6 months
#Jobs	14M	1.09M	1.26M	3.36M
Job priorities	High, Low	-	-	-
Job types	Training, Dev., On-Infer., Off-Infer.	Training, Dev, Evaluation	-	-
Model types	LLM, DM, DNN, Rec	LLM	DNN	DNN
GPU Models[2]	H20, H800, A10, A30 A100, A800, L20 XPU-A, XPU-B XPU-C, XPU-D, XPU-E	A100	T4, P100 V100	1080Ti, V100
#GPUs	155410	4704	6742	6416
#GPU nodes	37707	588	1814	802

Table 1: Comparison between the ASI trace and public GPU cluster traces in prior analysis works. **Dev.**: development; **On-Infer.**: online inference; **Off-Infer.**: offline inference.

3 Workload Characterization

This section characterizes the ASI workload. We first give an overview of the trace—the units of work it records, along with job types, priorities, models, and hardware—and then analyze the temporal and spatial patterns of resource demand and usage.

3.1 Trace Overview

Trace information. The ASI trace records a mix of training, inference, and development jobs running diverse ML models across mainstream frameworks [10, 24, 45, 64]. For each job, it details resource requests, execution durations, scheduling delays, and utilization of GPUs, CPUs, GPU memory, and main memory at job, task, and instance granularity. We further enrich the trace with user-provided application metadata, such as job and model types, and with machine-level telemetry—GPU specifications, node capacities, and time-varying utilization—collected by daemon agents that periodically query the Linux kernel and GPU drivers (e.g., NVML [35]).

Jobs, tasks, and instances. Similar to prior studies [16, 51], users submit jobs as the primary unit of work. Each job consists of one or more tasks that perform distinct computational roles, and each task runs as one or more instances, each encapsulated in a Kubernetes pod (one or more co-scheduled containers). We manage these pods with a customized Kubernetes that lets users specify fine-grained resource requests and limits [21, 22].

Job types. Jobs in ASI span the entire lifecycle of ML model development and deployment, and fall into four broad types:

[2]We use XPU as a desensitized name for non-NVIDIA GPUs.

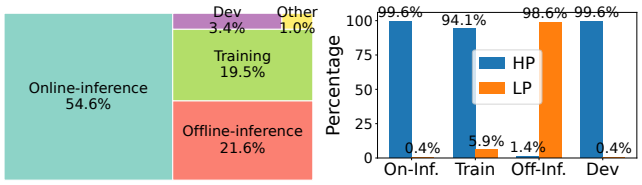


Figure 2: **Left:** Job type distribution. **Right:** Distribution of job priority within each job type.

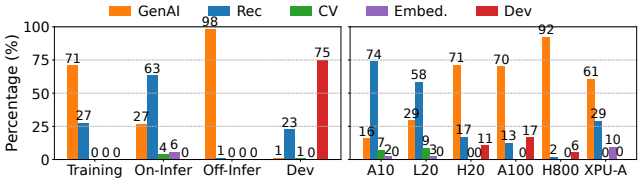


Figure 3: Model type distribution grouped by job types (left) and GPUs (right).

- **Dev** jobs are typically interactive, ad-hoc sessions used for model prototyping and debugging via coding interfaces.
- **Training** jobs execute iterative optimization algorithms to update model parameters. The duration of these jobs varies from thousands of hours when training from scratch to much shorter periods for fine-tuning tasks.
- **Offline-inference** jobs perform batch model inference tasks, such as model evaluation and data generation [50]. These jobs are generally not latency-sensitive.
- **Online-inference** jobs support user-facing or latency-critical model serving applications.

Fig. 2-Left illustrates the distribution of these job types within our cluster. Online-inference constitutes the largest proportion, accounting for over 50% of the workload.

Job priorities. We allow users to specify job priorities, enabling the scheduler to differentiate between workloads. In our trace, we coarsely classify jobs into high-priority (HP) jobs and low-priority (LP) spot jobs for desensitization purposes. HP resources command higher prices but provide guaranteed availability without preemption. Conversely, LP jobs are cheaper but may be preempted by HP jobs.

As shown in Fig. 2-Right, Online-inference and Training jobs are predominantly HP: Online-inference requires high availability for latency-critical services, while training needs long, stable allocations to support iterative optimization. Offline-inference jobs, by contrast, are typically LP, since they are latency-insensitive and well-suited to lower-priority execution that maximizes economic efficiency. Notably, Dev jobs are almost exclusively HP: developers need stable environments and must avoid sudden termination so that intermediate state can be checkpointed.

Various models are deployed. Models executing in ASI can be broadly classified into five categories: *GenAI* (generative AI, including text, image, and video generation), *Rec* (recommendation models, such as CTR prediction [66]), *CV* (computer vision models, e.g., OCR), *Embedding models*, and *Dev* (models under development). We observe several popular model families, including Qwen [57,58], TBStars [1,7], WanX [48], DeepSeek [12], and BERT [5].

Fig. 3-Left shows the model type distribution within each job type. GenAI workloads dominate both Training (71%) and Offline-inference (98%), reflecting heavy demand for large-scale training, model evaluation, and data generation. Online-inference, by contrast, is primarily Rec models (63%), matching the low-latency demands of user-facing services.

Fig. 3-Right breaks down workloads across GPU types. Older or mid-tier GPUs such as the A10 and L20 mostly run classical DNN models, including recommendation and CV models, whereas newer high-end accelerators such as the H800 are dedicated almost entirely to GenAI. This pronounced specialization suggests that users deliberately match hardware capability to workload requirements to control cost.

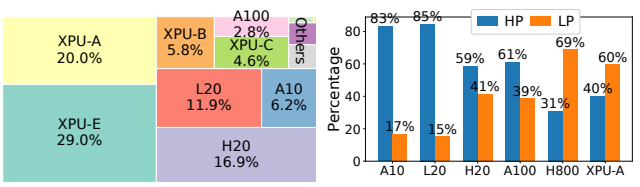


Figure 4: **Left**: Distribution of GPUs. **Right:** Job priority distribution in terms of the GPU hours they run.

Various GPU types to choose from. We offer a heterogeneous set of GPUs (Table 1) and let users request a specific GPU type for their tasks. Strikingly, over 99% of jobs explicitly pin a required GPU model, in sharp contrast to the 6% reported by an earlier study [51].

Fig. 4-Left shows the distribution of GPU types in our inventory, which spans hardware from NVIDIA and other vendors. Fig. 4-Right then shows how HP and LP jobs adopt these types, revealing that job priority correlates with an accelerator's compute capability. Earlier-generation or mid-tier accelerators such as the A10 (83% HP) and L20 (85% HP) are dominated by HP production-critical workloads, partly because they mainly serve mature traditional DNN models that have anchored our businesses for years (see Fig. 3-Right).

Conversely, LP jobs favor newer, more powerful hardware such as the H800, since most of them run GenAI models (Fig. 3-Left), and offline-inference, being latency-insensitive, is cost-effective to run at low priority. Finally, the vast majority of jobs request homogeneous GPUs; fewer than 1% use heterogeneous GPUs.

3.2 Temporal Pattern

Diurnal request and utilization patterns. GPU demand in ASI follows strong diurnal cycles. Fig. 5-Top shows the

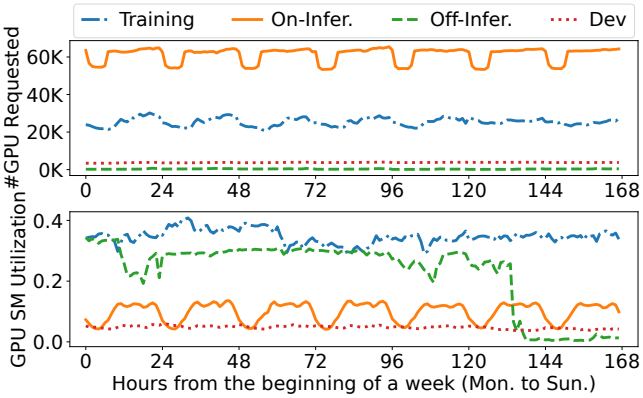


Figure 5: **Top:** number of requested GPUs over one week; **Bottom:** GPU SM utilization over the same week.

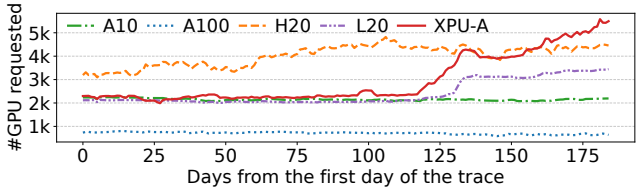


Figure 6: #GPUs requested for different GPU types.

volume of GPUs requested over a representative week: both Training and Online-inference jobs request substantially more GPUs during the day than late at night.

GPU SM utilization, however, tells a more nuanced story (Fig. 5-Bottom). Online-inference utilization stays diurnal because it tracks immediate user demand, whereas Training utilization does not, since long-running iterations keep GPUs busy around the clock. Offline-inference instead exhibits a weekly cycle, with SM utilization dropping sharply over the weekend.

GPU adoption. GPU adoption in ASI has shifted as we diversified our hardware supply. NVIDIA GPUs have been deployed for years and see relatively stable demand. To ease supply-chain constraints and build a more sustainable infrastructure, we recently began sourcing GPUs from other vendors; these non-NVIDIA devices, which we call XPUs, now make up a significant portion of our fleet (see Fig. 4-Left). Users were initially reluctant to adopt XPU-A because its out-of-the-box performance fell short of their expectations. Starting around Day 120, however, XPU-A adoption rises sharply (Fig. 6), following our release of a dedicated optimization suite that markedly improves model performance on XPU-A (§5.1).

Task instance run-time. Fig. 7 shows the CDF of task execution time, grouped by job type and by model type. Dev tasks run the longest, since developers keep long-lived interactive coding sessions open. Excluding Dev, Online-inference and Training tasks have comparable execution times, both longer than Offline-inference tasks. By model type, Rec and CV

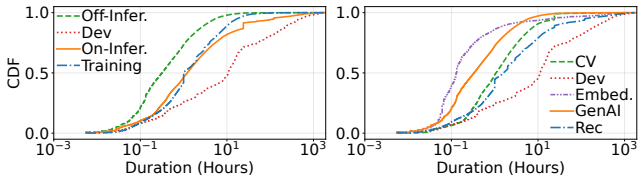


Figure 7: Task execution time grouped by job type (**Left**) and by model type (**Right**).

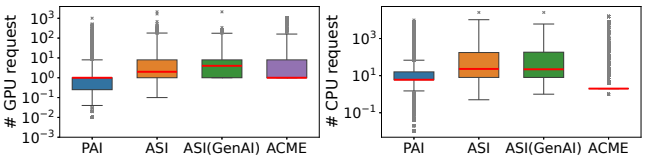


Figure 8: Jobs' #GPU and #CPU requested in different traces.

models accordingly run the longest, as they are dominated by Online-inference workloads, whereas GenAI models, which dominate Offline-inference, run shorter than CV and Rec models.

Overall, the median job execution time in ASI is 5 hours—far longer than the 23 minutes in PAI [51] and the 2 minutes in Acme [16]. We attribute this to ASI's mix of long-running online serving and large-scale training: PAI covers only DNN jobs, while Acme is dominated by short-running evaluation jobs (93%). Scheduling latency, in contrast, is low: the median is just 1 second, and even the P90 for HP tasks is 101 seconds—negligible against their P90 execution time of 24 hours.

3.3 Spatial Pattern

We now turn to spatial patterns—how tasks request and use resources across the cluster. ASI samples telemetry for every running task at 20-second intervals.

GPU and CPU request. Jobs in ASI request markedly more GPUs per job than in older traces. Fig. 8 compares the distribution of requested GPUs and CPUs against PAI [51] and Acme [16] using boxplots; for ASI, we report both the full dataset (ASI) and the GenAI-only subset (ASI(GenAI)). The median and mean GPU request grow from 1 and 2.6 in PAI to 2 and 11.0 in ASI, and to 4 and 11.0 in ASI(GenAI), driven in part by GenAI models that PAI predates. Acme, which targets LLM training, has a median and mean of 1 and 5.0; its low values arise because short-running evaluation jobs dominate its job count yet consume few resources [16], which also explains the outliers in the CPU plot. CPU requests show a similar PAI-to-ASI trend.

GPU allocation. We measure the GPU allocation ratio—the fraction of total GPU capacity claimed by jobs—which reflects how effectively resources are claimed and directly affects revenue. We further differentiate the allocation ratio attributed to HP jobs from that of combined HP and LP jobs

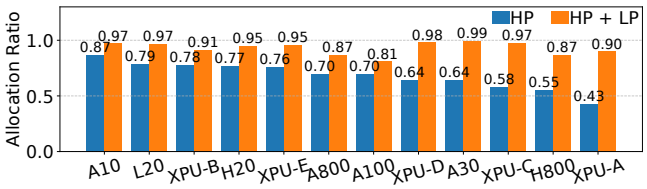


Figure 9: GPU allocation ratio by GPU type. **HP:** HP jobs only. **HP+LP:** HP and LP jobs combined.

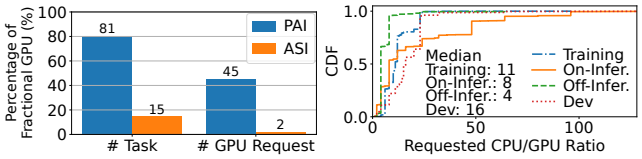


Figure 10: **Left**: Percentage of GPU sharing (fractional GPU) usage. **Right:** Ratio of #CPU/#GPU requested for tasks.

(HP+LP) to illustrate the effectiveness of LP workloads in boosting utilization. As Fig. 9 shows, LP jobs raise the allocation ratio for all GPU types, and especially for high-end GPUs: the cluster average rises from 68% with HP jobs alone to 93% with HP and LP combined. We detail how we enable LP spot jobs in §4.2.

Rare GPU sharing. GPU sharing lets multiple jobs that request fractional GPUs (< 1.0) multiplex a single device, and prior systems treat it as a key efficiency lever [51–53]. In ASI, however, GPU sharing is now rarely used, for reasons we discuss in §4.2. Fig. 10-Left reports both the share of tasks that claim fractional GPUs and the share of total GPUs they request; compared with PAI, both are negligible in ASI.

CPU/GPU ratio. Although ML workloads train and run inference on GPUs, much of their data processing—fetching and sampling, for example—runs on CPUs, which can become a bottleneck [51]. Fig. 10-Right shows the ratio of requested CPUs to requested GPUs per task, the CPU/GPU ratio. Most tasks have moderate ratios, but some are extreme, because ASI's GPU servers span a wide range of CPU/GPU configurations: an 8-GPU H20 server may have 192 CPU cores, whereas a 1-GPU A10 server may have 128. To keep requests feasible, we cap each request's CPU/GPU ratio according to its target server type.

Resource utilization. Fig. 11 shows task-level resource utilization by job type. Dev tasks have the lowest utilization across all metrics, reflecting their ad-hoc nature, while Training tasks have the highest, reflecting their compute intensity. Online-inference and Offline-inference use host memory similarly but differ sharply in CPU and GPU SM utilization, because Online-inference is driven by interactive user requests whereas offline-inference runs as batch jobs. Online-inference also uses less GPU memory than offline-inference, as it mainly runs standard DNN models while offline-inference predominantly runs GenAI models.

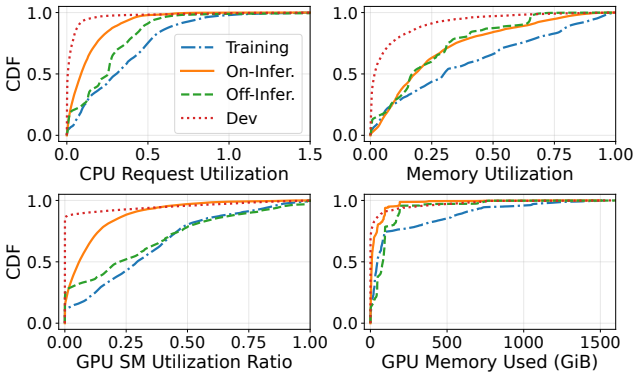


Figure 11: Task-level resource utilization.

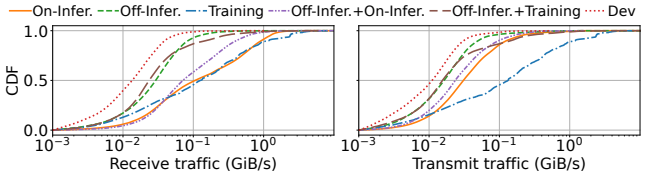


Figure 12: Node-level network utilization.

We next examine node-level network utilization (receive and transmit bandwidth). Because tasks of different job types can share a node, we group nodes by the mix of tasks they host and show the six most common mixes. As Fig. 12 shows, nodes running only training tasks have the highest network utilization, since most ASI training uses GenAI models that rely on parallelism and thus heavy inter-node communication.

3.4 Case Study

We close the characterization with three case studies of online LLM serving in ASI.

PD-disaggregated LLM serving. Adoption of prefill-decode (PD) disaggregation [14, 39, 65] for LLM serving has surged in ASI: the daily average GPU count for these workloads grew 3.7× over the trace period, from 1,041 to 3,841. Yet, contrary to strategies that pair heterogeneous GPUs with the distinct compute profiles of prefill and decode [65], our users request homogeneous GPUs, citing deployment complexity and network degradation when spanning heterogeneous GPU types (see §5.2).

Smaller LLMs are favored. Among one month of online-inference tasks running Qwen-family LLMs [57, 58], smaller models dominate usage. Grouped by parameter count, the *0–30B*, *30–100B*, and *100B+* tiers consume 60.9%, 35.0%, and 4.1% of GPU hours, respectively. Users favor smaller models because they cost less and, after fine-tuning, perform well enough in their target domains.

Comparison between representative LLMs and DNNs. We compare the three most popular models in online-

inference tasks: BERT [5], Qwen [57, 58], and CTR prediction [66]. CTR, BERT, and Qwen request a median of 24, 8, and 4 CPU cores, respectively—consistent with prior reports that recommendation models are more CPU-heavy [51, 59]. As for GPU compute, Qwen's median GPU SM utilization of 26% is 2.7× that of CTR and 9.6× that of BERT.

4 Improving Resource Utilization

In this section, we describe two resource-management challenges observed in the ASI cluster, GPU fragmentation and resource underutilization, and present the production mechanisms we deploy to address them.

4.1 Tackling GPU Fragmentation

GPU fragmentation breaks idle GPU resources into pieces that cannot be allocated as a whole to satisfy a resource request [43, 52], shrinking the effective capacity of the cluster. We first diagnose the causes of fragmentation in ASI and then present two mechanisms that address its dominant causes: *IPC defragmentation*, which consolidates jobs to recover stranded GPUs and relieve CPU bottlenecks, and *topology-aware allocation*, which mitigates the fragmentation induced by network-topology requirements.

Causes of GPU fragmentation. Prior work [52] identifies three common causes of GPU fragmentation:

1) Fractional GPU fragmentation. ML models in production can be small. For example, a ResNet-152 model only has 60 million parameters, which cannot occupy a whole GPU. Therefore, we allow users to request a fractional GPU, e.g., 0.25 GPU, such that multiple small ML models can collocate in a single GPU for resource-efficient GPU sharing [51]. Despite the benefits, fractional GPU can cause intra-GPU fragmentation, as fractional allocations can leave unusable residual capacity on individual GPUs, especially preventing the placement of jobs that require whole GPU cards.

2) Stranded GPU fragmentation. For multi-GPU nodes, GPU fragmentation can appear when a job does not request all GPUs in the node, making the remaining GPU cards stranded. For example, assigning a 2-GPU task to an 8-GPU node results in intra-node fragmentation, leaving the remaining six GPUs as stranded resources that cannot satisfy a subsequent request for a full-node (8-GPU) allocation.

3) Insufficient CPUs. While GenAI models heavily use GPUs, we observe CPUs can be the bottleneck for GPU allocation. This happens when a node can satisfy the GPU requests but fails to meet associated CPU requests, typically due to extreme CPU/GPU ratios (see Fig. 10-Right). Exhausting CPUs in a node can cause GPU fragmentation.

In ASI, fragmentation is rarely caused by fractional GPUs but is instead driven by two bottlenecks: stranded GPUs and insufficient CPUs. Fig. 13 diagnoses this by measuring fragmentation, in unallocatable idle GPUs, under varying resource request configurations. That fractional GPUs barely contribute echoes Fig. 10-Left, where GPU sharing is rare. For requests with high CPU/GPU ratios (e.g., 1G8C and 8G128C), the dominant bottleneck is "Insufficient CPU": GPUs cannot be allocated because CPUs are exhausted. For mid-sized configurations (e.g., 4G32C through 8G64C), "Stranded GPUs" (orange bars) dominate: aggregate GPU capacity exists but is fragmented across nodes, preventing the scheduling of jobs that require contiguous multi-GPU allocations.

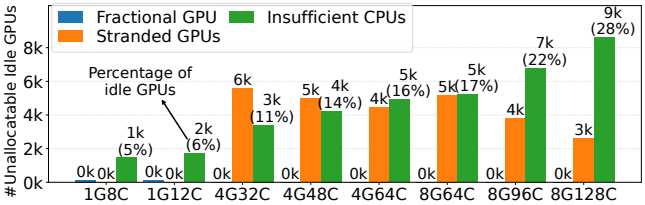


Figure 13: GPU fragmentation with varying resource requests. **8G128C**: request 8 GPUs and 128 CPUs.

Beyond these three known causes, we observe a new cause of GPU fragmentation when users submit a multi-GPU request with network-topology requirements.

4) Network topology. Very large ML models, such as LLMs, often demand aggregate GPU memory exceeding that of a single GPU server, so the standard practice is to partition the model into chunks distributed across multiple GPU nodes [34]. Both training and inference of such models entail intensive cross-node communication. To reduce this communication bottleneck, users often request *topology-aware placement*; for example, requiring all allocated GPUs to reside under the same ASW. As described in §2, the ASI fat-tree network is not fully non-blocking, so a job placed across multiple ASWs traverses additional switch layers and loses effective bandwidth; our benchmark shows that colocating all GPUs under a single ASW improves `allreduce` bandwidth by 27% relative to cross-ASW placement.

However, strictly enforcing such topology requirements introduces a new form of fragmentation: GPUs scattered across different switches become ineligible for a single job, substantially shrinking the effective pool of available resources. To illustrate its severity within ASI, we constructed two large-scale configurations—jobs requesting 128 and 256 GPUs, respectively, each GPU requiring 12 CPUs—and measured how many such requests ASI could fulfill. Fig. 14 reports the impact of these ASW constraints on allocation.

The comparison between the baseline (across ASW) and the ASW-enforced configuration (within ASW) shows that enforcing ASW constraints significantly reduces the number of fulfillable jobs. This reduction is particularly severe when requesting large sets of homogeneous NVIDIA GPUs, where capacity drops to near zero, indicating that strict topological requirements disproportionately impact these high-demand

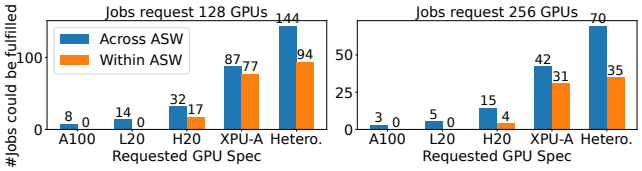


Figure 14: Impact of network topology on GPU allocation.

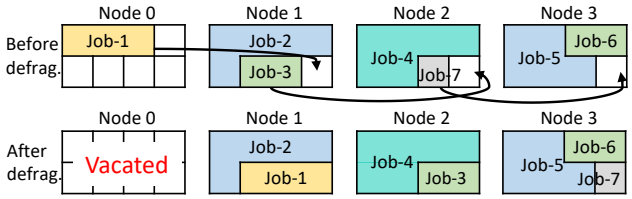


Figure 15: Illustration of the ejection chain in IPC. To evacuate Node 0, Job-1 must first be migrated off the node. Before defragmentation, none of Nodes 1–3 individually has sufficient GPU slots to host Job-1, even though they collectively provide three idle GPU slots. With an ejection chain: Job-3 is moved from Node 1 to Node 2, which first requires shifting Job-7 to Node 3; Job-1 is then moved from Node 0 to Node 1, vacating Node 0 after defragmentation. CPUs are omitted for clarity.

resources. Conversely, for less popular XPU-A, the impact is less pronounced. Furthermore, when heterogeneous GPU allocations are permitted, the cluster retains higher capacity for large-scale requests. Therefore, promoting the use of less popular GPUs alongside heterogeneous GPUs can effectively alleviate the bottlenecks caused by topology requirements.

GPU defragmentation. To recover stranded GPUs and relieve CPU bottlenecks, ASI runs GPU defragmentation as a routine maintenance task that consolidates jobs occupying only a fraction of a server, freeing whole-machine GPU capacity for incoming workloads that demand both multiple GPUs and substantial CPUs.

Challenges. Such consolidation is achieved by migrating running jobs to pack them onto fewer servers, yet performing these migrations in a live production cluster raises three challenges. First, new jobs can be submitted to the cluster at any time, which dynamically alters resource availability and allocation. As a result, the decision-making of GPU job migration must be fast, ensuring that migration and scheduling decisions can be completed within a few minutes, which poses a challenge given the large scale of our cluster. Second, migrating workloads across nodes must satisfy practical cluster-level constraints [28], including adhering to existing affinity and anti-affinity rules [20], which define if tasks can run on a certain node or if a set of tasks can be colocated. Besides, 40% of tasks in ASI are labeled as "locked", indicating that they are business critical and cannot be moved. Third, GPU job migration and defragmentation must remain transparent to users. This means we cannot impose constraints on the underlying machine learning frameworks used by the jobs, nor can these operations introduce any service downtime [29].

IPC Algorithm. We design and deploy IPC (iterative partitioned consolidation), a practical defragmentation algorithm that takes a snapshot of the current node and task states and generates a set of task migration decisions whose objective is to vacate as many nodes as possible. Each of the three challenges above maps to a design choice in IPC: *divide-and-conquer* keeps decision-making fast at cluster scale, *ejection-chain scheduling* respects affinity, anti-affinity, and locked-task constraints during migration, and an *iterative* process drives the number of vacated nodes higher across rounds. To keep migration transparent to users, IPC adopts a make-before-break approach: it brings up a new instance on the target node before terminating the old one, eliminating service downtime.

1) Divide-and-conquer. To generate migration decisions quickly, IPC randomly partitions the cluster's nodes into disjoint groups, each an independent subproblem within which IPC tries to vacate as many nodes as possible. These subproblems run embarrassingly in parallel, accelerating decision-making at cluster scale.

2) Ejection chain scheduling. Within each group, IPC scores every node by priority and heuristics, e.g., the number of pods it hosts, and selects higher-priority (fewer-pod) nodes first as they are easier to vacate; Algorithm 1 gives the pseudocode for a partition. As illustrated in Fig. 15, IPC builds an ejection chain recursively (`Eject`$(\cdot)$ at line 6): if a destination node cannot accommodate a task, it recursively moves tasks off that destination node until all pods are placed or the recursion-depth limit K (line 6) is reached. In practice, we set K to 3.

Algorithm 1: Ejection Chain Scheduling

Input: Node partition $\mathcal{N}$, Max depth K
Output: Set of migration actions $\mathcal{M}$

1 $\mathcal{M} \leftarrow \emptyset$; $\mathcal{L} \leftarrow$ Sort$(\mathcal{N})$ `// sort nodes in the partition`
2
3 **foreach** $n \in \mathcal{L}$ **do**
4 $\quad \mathcal{M}_n \leftarrow \emptyset$; *feasible* $\leftarrow$ **true**
5 $\quad$ **foreach** $\tau \in n.tasks$ **do**
$\quad\quad$ `// Eject() recursively searches a feasible ejection chain C that satisfies the practical constraints and resource requests and is shorter than K`
6 $\quad\quad C \leftarrow$ Eject(τ, K)
7 $\quad\quad$ **if** $C \neq \emptyset$ **then**
8 $\quad\quad\quad \mathcal{M}_n \leftarrow \mathcal{M}_n \cup C$
9 $\quad\quad$ **else**
10 $\quad\quad\quad$ *feasible* $\leftarrow$ **false**; **break**
11 $\quad$ **if** *feasible* **then**
12 $\quad\quad \mathcal{M} \leftarrow \mathcal{M} \cup \mathcal{M}_n$
13 $\quad$ **else**
14 $\quad\quad$ Mark n as non-migratable
15 **return** $\mathcal{M}$

3) Iteration. To vacate as many nodes as possible, the IPC algorithm iteratively runs the process of divide-and-conquer and ejection chain scheduling for several rounds, attempting to vacate nodes in each round. We empirically observe that the marginal benefit becomes small after the third round, so we cap the maximum number of rounds at five.

Effectiveness of IPC. IPC has run stably in our scheduler for years, making a migration decision in under 2 minutes; on a two-month trace replay, it reduces the number of not-fully-occupied nodes by 20.2%. It replaced our earlier reliance on Kubernetes [21] eviction, which redistributes evicted jobs randomly across the cluster. We use a heuristic rather than an optimal plan because optimal defragmentation is impractical online: the optimization is expensive to solve, and an optimal plan may demand large-scale pod migrations that production cannot absorb. Formulating defragmentation as an integer linear program (ILP), a 25-node cluster yields 4,000 binary variables and 800 constraints that Gurobi [13] solves to optimality in roughly 5 minutes on a 16-CPU node, whereas a 100-node cluster takes two days—yet in practice each IPC partition holds 500 nodes.

Topology-aware GPU allocation. To address the topology-induced fragmentation of Fig. 14, ASI deploys an allocation policy that consolidates the GPUs of a large-scale request into as few ASWs as possible. We describe the deployed single-job policy here; its optimal multi-job generalization remains an open challenge (§5), so we report no aggregate benefit number for it. We introduce *entropy* to quantify the distribution of the allocated GPUs of a job. Suppose that the GPUs requested by a job are distributed across n ASWs. Let $g_1, g_2, \ldots, g_n$ denote the number of GPUs in each ASW, and let G denote the total number of GPUs ($G = \sum_{i=1}^{n} g_i$). The GPU entropy of the job is defined as

$$H = -\sum_{i=1}^{n} p_i \log p_i, \text{ where } p_i = \frac{g_i}{G}, \tag{1}$$

where higher entropy indicates a more uniform distribution, whereas lower entropy indicates concentration within a small number of ASWs. Our allocation policy seeks to consolidate the GPUs assigned to the job within as few ASWs as possible; accordingly, lower entropy is preferred.

The process of generating a GPU allocation plan for a large-scale job is shown in Algorithm 2. For each iteration, we decide which ASW to use by iterating over all ASWs, and for each ASW, we calculate the GPU-topology distribution if we allocate as many of its GPUs to the job as possible. Then we select one that yields the minimum entropy, allocate that ASW's idle GPUs to the job, and proceed to the next iteration.

This greedy algorithm is optimal for a single job, but not across multiple jobs, where the scheduling order matters and the problem becomes combinatorial. In practice, we sort jobs by size in descending order and schedule the largest first; cross-ASW migration reuses IPC, with network topology added as a practical constraint. We leave large-scale topology-aware allocation as an open challenge in §5.

Algorithm 2: Topology-aware GPU allocation

Input: Number of requested GPUs G; available ASWs $\mathcal{N}$
Output: Allocation Plan $\mathcal{P}$

$\mathcal{P} \leftarrow \emptyset$; $G_{rem} \leftarrow G$
while $G_{rem} > 0$ ***and*** $\mathcal{N} \neq \emptyset$ **do**
  `// Find ASW s that minimizes entropy if added`
  $s^* \leftarrow$ $\arg\min_{s \in \mathcal{N}} \text{Entropy}(\mathcal{P} \cup \{\min(\text{Cap}(s), G_{rem}) \text{ GPUs of } s\})$
  $n \leftarrow \min(\text{Cap}(s^*), G_{rem})$
  $\mathcal{P} \leftarrow \mathcal{P} \cup \{(s^*, n)\}$; $G_{rem} \leftarrow G_{rem} - n$
  $\mathcal{N} \leftarrow \mathcal{N} \setminus \{s^*\}$
end
return $\mathcal{P}$

4.2 Addressing Resource Underutilization

Our characterization reveals substantial temporal and spatial resource underutilization in the cluster (Fig. 5 and Fig. 11). Discussions with our users trace this primarily to a common practice: users over-provision GPU capacity to absorb diurnal traffic peaks, maintain failover redundancy, and reserve headroom for peak events. Such defensive reservation leaves a sizable fraction of resources idle for long periods, degrading cluster-wide efficiency.

GPU sharing as a strawman solution. A natural way to reclaim underutilized resources is *GPU sharing*: co-locating multiple jobs on one GPU so they time-multiplex it and fill idle cycles [51, 52, 55]. Although ASI supports GPU sharing, adoption remains low (Fig. 10-Left), for two reasons. First, GPUs offer weaker isolation than CPUs, so co-located jobs interfere; even virtualization mechanisms such as MPS (Multi-Process Service) [37] and MIG (Multi-Instance GPU) [36] provide only coarse-grained, inflexible partitioning. Second, emerging GenAI models consume large amounts of GPU memory for parameters and intermediate state (e.g., KV-Cache) [24], making it infeasible to share a single GPU across jobs. Consequently, in ASI fractional GPU requests come almost exclusively from classical CV and recommendation models, which reserve GPUs with MPS enabled to serve small models. GPU sharing therefore reclaims idle capacity only within a single device; to reclaim the much larger pool of reserved-but-idle capacity across jobs, we instead operate at job granularity through SpotGPU.

SpotGPU. SpotGPU is designed to achieve two objectives. First, it reclaims reserved-but-idle GPU capacity and returns it to productive use, raising cluster-level utilization. Second, it offers this reclaimed capacity, together with other idle resources, to users with flexible workloads at a discounted spot rate. Concretely, SpotGPU safely oversubscribes the reserved capacity of HP jobs and allocates the resulting idle capacity to LP spot jobs. Because resources granted to LP jobs are not guaranteed, they are priced lower; when HP jobs reclaim their

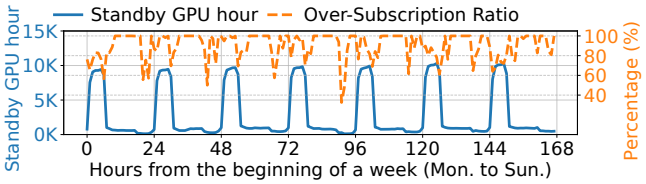


Figure 16: Temporal pattern of GPU hours of "Standby" tasks and the over-subscription ratio of the "Standby" GPU hours.

reserved resources, SpotGPU preempts the running LP jobs in near real-time.

"Standby" HP tasks. To reclaim reserved capacity from HP jobs without disrupting them, SpotGPU offers a "Standby" mechanism and incentivizes participation with financial rebates. Through an API, a user labels an HP task as "Standby" for a chosen window, e.g., by updating the labels of its Kubernetes pods [23], signaling willingness to temporarily release the task's allocated resources to LP spot jobs. Fig. 16 shows that "Standby" GPU hours follow a pronounced diurnal pattern, surging at midnight to as much as 10,000 hours.

A Standby task must reactivate quickly, so SpotGPU does not tear down its container. Instead, it detaches the task's request traffic and replaces the container's main service process with a `sleep` command, releasing the CPUs and GPUs while keeping the container warm; reactivation then skips the costly steps of rescheduling and image pulls. When an HP task must resume, SpotGPU first triggers a graceful eviction of the co-located spot job and then signals the HP task to resume, preserving HP SLOs. The eviction proceeds as follows: (i) the node agent observes the HP pod arriving on its node, (ii) it sends `SIGTERM` to the co-located spot pod, (iii) the in-container graceful shutdown runs, executing user-defined logic, bounded by a 60-second termination period, and (iv) the container exits and the GPU is reclaimed. Steps (i), (ii), and (iv) are platform-side and complete quickly and deterministically, so eviction latency is dominated by the user-defined step (iii). In our deployment, the average eviction time is 13 seconds (P95 of 48 seconds), well within the 60-second window; containers that do not exit within this window are forcibly terminated via `SIGKILL`, affecting fewer than 5% of evictions. To resume the HP task, SpotGPU restores its original command in place of the `sleep`. As shown in Fig. 16, on average, 90% of "Standby" GPU hours are harvested.

Preemptive task scheduling. For newly submitted jobs, ASI uses a preemptive scheduling policy that prefers non-preemptive allocation for stability but, when it must reclaim resources from LP spot jobs, accounts for the resulting preemption cost. Since GPU tasks commonly checkpoint during execution [31,46], a preempted task can resume from its last checkpoint rather than restart. ASI lets users specify a checkpoint interval, and we define the preemption cost of a task as its wasted computation, i.e., the work performed since its last successful checkpoint.

We formulate the preemptive scheduling decision as a mixed-integer linear program (MILP), detailed in Appendix A. Because the MILP is intractable for online scheduling, we instead use an efficient heuristic. The algorithm first attempts non-preemptive allocation for an HP task on available GPUs. If this fails, it switches to preemptive mode and evicts selected LP spot tasks to reclaim their resources for the HP task. New spot tasks, in contrast, are scheduled strictly non-preemptively.

1) Non-preemptive scheduling. For an HP or spot task, the scheduler first filters nodes that meet the task's GPU requirements to form a candidate set. It then ranks the node candidates by three criteria applied in order, each breaking ties left by the previous one:

- *Task packing:* prioritize nodes with the least idle GPU capacity, packing tasks onto fewer nodes to limit fragmentation.
- *Homogeneous co-location*: place HP tasks on nodes hosting other HP tasks, and assign spot tasks to spot-dedicated nodes. This strategy insulates and prioritizes HP tasks, allowing for efficient future scheduling via preempting spot tasks when necessary.
- *Eviction history*: assign a new spot task to the node with the *lowest* historical eviction count so it is less likely to be preempted, and a new HP task to the node with the *highest* eviction count. This strategy concentrates future preemptions on already-volatile nodes and leaves stable nodes for long-running spot tasks, reducing repeated evictions of the same spot tasks.

2) Preemptive scheduling. When a new HP task cannot be placed non-preemptively, the scheduler preempts spot tasks. Because preemption disrupts a spot task's progress, the scheduler chooses victims so as to minimize wasted computation: it identifies the nodes that could host the HP task if their spot tasks were removed, scores each by its preemption cost, and selects the node with the lowest cost.

Concretely, consider an HP task arriving at time τ_{now} that requires G_{hp} GPUs. For a running spot task t using G_t GPUs with its last checkpoint at τ_{ck}, we define its preemption cost as the lost work weighted by resource usage, $C_t = G_t \cdot (\tau_{now} - \tau_{ck})$. To score a node, the scheduler sorts its spot tasks by ascending preemption cost and takes the smallest prefix $\{t_1, \ldots, t_k\}$ such that their combined resources meet the HP task's demand, i.e., $\sum_{j=1}^{k} G_{t_j} \geq G_{hp}$. The node's total preemption cost is hence given by $\sum_{j=1}^{k} C_{t_j}$. Having all nodes scored, the scheduler then places the HP task on the one with the minimum preemption cost and evicts the selected spot tasks there. For heterogeneous GPUs, G_{hp} and G_t become resource vectors whose elements give the required quantity of each GPU type.

	#GPUs per node	Memory per card	HBM BW (TBps)	Inter–GPU BW (GBps)	FP16 Tensor Core Compute Spec.
H20	8	96	4.0	900	148 TFLOPS
XPU-A	8	192	5.2	896	203 TFLOPS

Table 2: Hardware specifications of H20 and XPU-A.

Benefits of SpotGPU. SpotGPU raises the average GPU allocation ratio from 68% to 93% (Fig. 9), meaning a substantially larger fraction of resources is put to productive use, and Fig. 16 shows that users actively release reserved capacity through "Standby", shortening idle periods. To isolate the preemptive scheduler's contribution, we compare it against a baseline that disables co-location and eviction awareness in non-preemptive scheduling and replaces preemptive scheduling with random victim selection. Our scheduler reduces LP task completion time by 24% without affecting HP task performance, a gain that stems from accounting for preemption cost to avoid discarding spot tasks' near-complete work.

5 Open Challenges

The mechanisms in §4 address resource-management problems that we have largely brought under control in production. We now turn to problems that remain open. We first examine the imbalanced adoption of heterogeneous GPUs, an increasingly pressing issue as we diversify beyond a single vendor, together with our preliminary efforts to close the resulting performance gap (§5.1). We then discuss three further challenges in allocation and colocation that we have yet to fully resolve (§5.2).

5.1 Imbalanced Adoption of Heterogeneous GPUs

The adoption gap. Meeting the growing demand for high-end GPUs while avoiding the risks of single-vendor dependency has led us to diversify our hardware. We recently integrated XPU-A from another vendor as a primary alternative, attracted by its large memory capacity and competitive compute. Adoption, however, is heavily skewed: we observe an *imbalanced adoption of heterogeneous GPUs*, in which HP jobs request NVIDIA GPUs and other XPUs far more than XPU-A (Fig. 9). Closing this gap would do more than load balancing. Because most XPU-As can satisfy strict network-topology requirements that NVIDIA GPUs often cannot (Fig. 14), broader XPU-A adoption would also relieve the topology-induced fragmentation of §4.1.

This imbalance stems from performance, not capacity. On paper, XPU-A surpasses the NVIDIA H20, its closest counterpart in FP16 compute, offering larger memory and higher bandwidth (Table 2). In practice, however, it underdelivers on GenAI models: Fig. 17-Left shows XPU-A trailing H20 when serving DeepSeek-R1. Profiling and discussions with

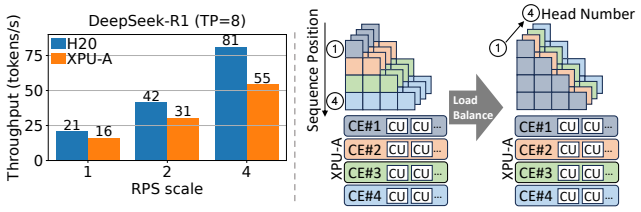


Figure 17: **Left:** Throughput of Deepseek-R1 [12]. **Right:** Optimized prefill. **CE**: Compute Engine. **CU**: Compute Unit.

the XPU team trace this to a single root cause—standard kernel implementations are mismatched to XPU-A's hardware-specific features—which motivates the optimizations below.

Optimizing XPU-A for LLMs. Guided by this diagnosis, we have benchmarked and optimized XPU-A for Qwen-family LLMs on vLLM [24] and RTP-LLM [10], a production-grade inference engine developed by Alibaba. For brevity, we describe two representative optimizations, one for the prefill phase and one for decoding.

Prefill. In the prefill phase, the Flash Attention kernel in XPU-A's official library distributes work poorly under the *causal mask*, the mechanism that lets each token attend only to preceding tokens. The mask imposes a triangular computation in which per-token load grows linearly with a token's position in the sequence (Fig. 17-Right). The existing kernel maps thread blocks to the logical grid along the sequence-length dimension and dispatches them to Compute Engines (CEs) of XPU-A in rigid round-robin order, causing severe load imbalance: the CE handling the tail of the sequence becomes a bottleneck while the others finish early and idle. We instead parallelize CE assignment along the attention-head dimension, which evens out the load across CEs. Simple as it is, this change yields a 1.58× speedup in prefill computation for a 4k-token sequence with 64 heads.

Decoding. During decoding, most serving engines use Paged Attention (PA) kernels [24], which are memory-bound because decoding each token requires reading the entire KV-cache history from GPU memory. To saturate Compute Units (CUs) of an XPU, PA adopts a SplitKV strategy that partitions the KV-cache into chunks processed in parallel across CUs. When we port PA to XPU-A with Triton [38], however, the compiler inverts the thread-block layout, so every thread subgroup redundantly recomputes the full matrix instead of dividing the work, wasting most of the compute. We fix this by injecting custom compiler passes into Triton that enforce the correct parallelism, and have contributed the fix back to the Triton ecosystem.

Effectiveness. We have integrated these and further optimizations into RTP-LLM [10], with two effects. First, user demand for XPU-A has risen sharply: HP jobs request 2.5× more XPU-A GPUs after the optimizations (Fig. 6). Second, the optimizations close the performance gap: on Qwen LLMs

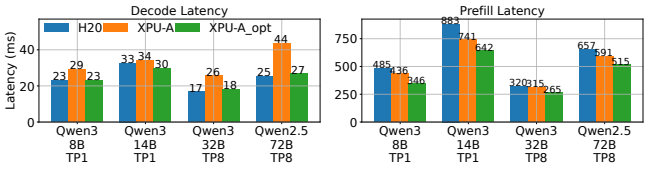


Figure 18: Performance of Qwen LLMs on varying GPUs with tensor parallelism (TP). XPU-A_opt: XPU-A with optimization. **Left:** Decoding performance at batch size=64. **Right:** Prefill performance (batch size=1).

```
# Original FlashAttention import
import flash_attn

# Replace it with our drop-in optimized implementation
import asi.api.flash_attn as flash_attn
```

Figure 19: An example of integrating our optimizations.

of varying sizes [57, 58], optimized XPU-A matches H20 and far outperforms the unoptimized baseline (Fig. 18). Benchmarking the same suite on vLLM [24, 44], optimized XPU-A improves request latency by 33% and 43% at one and two requests per second (RPS) over unoptimized XPU-A, surpassing H20 by 2% and 21%; the larger margin at RPS=2 comes from XPU-A's larger memory, which holds a bigger KV-cache and reduces queuing delay.

Programming interface. To make these optimizations easy to adopt, we expose them through drop-in APIs. In PyTorch, a user enables an optimization by changing a single `import` (Fig. 19); the frontend API preserves compatibility with existing code, and the API layer handles all remaining adaptation transparently.

Beyond LLMs. Our broader goal is to deliver high-performance services across heterogeneous GPUs, so that balanced GPU usage improves resource management and widens the range of viable hardware choices. The optimizations above target LLMs; substantial challenges remain for multimodal models, e.g., Qwen Next [42], and generative image/video models, whose distinct architectures and attention mechanisms demand separate hardware-specific tuning [30].

5.2 Additional Resource-Management Challenges

Topology-aware allocation under heterogeneity. In §4.1, we showed how network-topology constraints limit GPU allocation; these constraints also collide with hardware heterogeneity. In ASI, GPUs under the same ASW are homogeneous, so enforcing ASW-level locality precludes mixing heterogeneous GPUs within a single job—precisely when heterogeneity is becoming a key GenAI optimization [39]. The PD-disaggregation paradigm, for instance, is now widely used in LLM serving and accounts for up to 12,000 GPUs per day in ASI: prefill is compute-bound and favors high-performance GPUs, whereas decoding is memory-bound and needs high bandwidth [65]. Such heterogeneous matching is more cost-effective, but it places the KV-cache transfer between disaggregated nodes on the critical path—decoding cannot begin until the KV cache arrives—making network performance the bottleneck. Cost-efficiency therefore hinges on balancing hardware specialization against topology constraints, a tension that extends beyond LLM serving to any system built on heterogeneous GPUs [8, 18, 33, 49, 61].

Underutilization of online-inference jobs. Online-inference jobs dominate ASI yet run highly inefficiently: aggregated across all such jobs, median SM utilization is only 6% and median GPU-memory utilization only 30%. This low overall utilization masks two distinct workload patterns. GenAI inference is memory-bound, consuming 94% of GPU memory but only 5% of SM, so it exhausts memory while leaving compute idle. Conventional DNN inference, in contrast, is light on both resources, using just 20% of memory and 6% of SM. The two patterns are complementary: a DNN job needs little memory and could run on the compute that a memory-bound GenAI job leaves idle, raising SM utilization without contending for memory. Realizing this colocation in practice, however, is difficult. GPU sharing falls short under latency-sensitive online inference and lacks large-scale support [56]; hardware partitioning such as MIG [36] is too static for dynamic workloads; and GPU time-sharing violates strict latency SLOs because of weak isolation [51, 53].

Interference between colocated CPU and GPU jobs. Like Acme [16], ASI nodes show low CPU utilization (median 31%). We mitigate this by colocating pure-CPU jobs with GPU workloads, which makes 80% of nodes *CPU-job-dominant*, i.e., their CPU-job core usage exceeds that of GPU jobs. This colocation, however, interferes with GPU training: median and P90 SM utilization fall by 10% and 18% on *CPU-job-dominant* nodes relative to *GPU-job-dominant* ones, showing that our naive colocation is suboptimal. Prior approaches that exploit idle CPUs on GPU nodes [4, 27] target specific model architectures (e.g., LoRA adapters) and require high-end CPUs, limiting their applications. Closing this gap calls for a multi-resource scheduler that jointly accounts for GPUs and associated resources such as CPUs, together with new isolation techniques to contain interference.

6 Limitations and Related Work

We close by clarifying the scope of the ASI trace and situating it among prior workload-characterization studies. Our goal is not to cover every AI cluster workload, but to expose the resource-management issues that arise in a large shared production cluster with heterogeneous workloads, priorities, accelerators, and topology constraints.

Limitations. The ASI trace does not cover large-scale

foundation-model pretraining, which typically runs on dedicated clusters with the most advanced GPUs. It also does not distinguish fine-grained subcategories within each job type; for example, training jobs may include both supervised fine-tuning and LoRA fine-tuning, but the trace does not separate them. Finally, although we collect utilization data across CPU, memory, network, and GPU resources, space constraints lead us to focus mainly on GPU workloads and leave CPU-only jobs to future analysis.

Related work. Workload characterization has long informed cluster management, but existing public traces capture only part of the setting faced by modern production AI clusters. Table 1 compares ASI with the closest public GPU-cluster traces, Acme [16] and PAI [51]. Other recent studies focus on narrower slices of the AI infrastructure stack: ServeGen [54] provides granular request-level data for LLM serving, while ByteRobust and Megascale [19, 47] study stability issues in foundation-model training on tens of thousands of NVIDIA GPUs. In contrast, ASI exposes a shared production infrastructure that combines heterogeneous workloads, priorities, GPU types, and topology constraints in one cluster. Earlier DNN workload studies [3, 11, 15, 17, 32, 55, 63] remain useful references, but their reliance on older GPU generations makes them less representative of today's GPU cluster management problems.

7 Conclusion

We have characterized six months of production workloads in ASI, a shared AI cluster with 155,410 GPUs and 81 internal departments. The trace showed that modern AI clusters must manage heterogeneity in both workloads and hardware: classical DNN services, GenAI training, and GenAI serving coexist on GPU fleets spanning multiple generations and vendors. Our key lesson was that high GPU demand did not imply high effective utilization. Idle GPUs often remained unusable because capacity was stranded across nodes, blocked by CPU shortages, constrained by network locality, or reserved as production headroom. We also showed how these observations shaped production mechanisms. The remaining challenges point beyond simple packing: heterogeneous accelerators required software optimization before users adopted them, online inference left substantial compute idle, and CPU/GPU colocation recovered CPU cycles but interfered with GPU training. By releasing the sanitized ASI trace, we hope to support future work on GPU cluster management under coupled workload, hardware, topology, and priority heterogeneity.

Acknowledgments

We thank our shepherd and the anonymous reviewers of OSDI 2026 for their invaluable comments that help improve the quality of this work. We are also grateful to Dongqing Bao, Yujie Deng, Zhiwei Han, Wenhu Hu, and Zhuo Yuan at Alibaba Group for their helpful discussions. This work was supported in part by the HKUST-Alibaba Joint Laboratory on Big Data and AI, RGC CRF Grant (Ref. #C6015-23G), RGC GRF Grant (Ref. #16217124), and NSFC/RGC CRS Grant (Ref. #CRS_HKUST601/24).

References

[1] AIBase. Ali Mama launches Taobao Star video generation model and image-to-video application. https://news.aibase.com/news/14705, 2025.

[2] AliyunContainerService. gpushare-scheduler-extender. https://github.com/AliyunContainerService/gpushare-scheduler-extender, 2025.

[3] Shubham Chaudhary, Ramachandran Ramjee, Muthian Sivathanu, Nipun Kwatra, and Srinidhi Viswanatha. Balancing efficiency and fairness in heterogeneous GPU clusters for deep learning. In *Proc. ACM EuroSys*, 2020.

[4] Hongtao Chen, Weiyu Xie, Boxin Zhang, Jingqi Tang, Jiahao Wang, Jianwei Dong, Shaoyuan Chen, Ziwei Yuan, Chen Lin, Chengyu Qiu, Yuening Zhu, Qingliang Ou, Jiaqi Liao, Xianglin Chen, Zhiyuan Ai, Yongwei Wu, and Mingxing Zhang. KTransformers: Unleashing the full potential of CPU/GPU hybrid inference for MoE models. In *Proc. ACM SOSP*, 2025.

[5] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: Pre-training of deep bidirectional transformers for language understanding. In Jill Burstein, Christy Doran, and Thamar Solorio, editors, *Proc. ACL*, June 2019.

[6] Patrick Esser, Sumith Kulal, Andreas Blattmann, Rahim Entezari, Jonas Müller, Harry Saini, Yam Levi, Dominik Lorenz, Axel Sauer, Frederic Boesel, Dustin Podell, Tim Dockhorn, Zion English, and Robin Rombach. Scaling rectified flow transformers for high-resolution image synthesis. In *Proc. ICML*, 2024.

[7] Hao Fang, Zechao Zhan, Weixin Feng, Ziwei Huang, Xubin Li, and Tiezheng Ge. TBStar-Edit: From image editing pattern shifting to consistency enhancement. *arXiv preprint arXiv:2510.04483*, 2025.

[8] Hao Ge, Fangcheng Fu, Haoyang Li, Xuanyu Wang, Sheng Lin, Yujie Wang, Xiaonan Nie, Hailin Zhang, Xupeng Miao, and Bin Cui. Enabling parallelism hot switching for efficient training of large language models. In *Proc. ACM SOSP*, 2024.

[9] Robert Grandl, Ganesh Ananthanarayanan, Srikanth Kandula, Sriram Rao, and Aditya Akella. Multi-resource packing for cluster schedulers. *Proc. ACM SIGCOMM*, 2014.

[10] Alibaba Group. RTP-LLM. https://github.com/alibaba/rtp-llm, 2025.

[11] Juncheng Gu, Mosharaf Chowdhury, Kang G. Shin, Yibo Zhu, Myeongjae Jeon, Junjie Qian, Hongqiang Liu, and Chuanxiong Guo. Tiresias: A GPU cluster manager for distributed deep learning. In *Proc. USENIX NSDI*, 2019.

[12] Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Peiyi Wang, Qihao Zhu, Runxin Xu, Ruoyu Zhang, Shirong Ma, Xiao Bi, et al. DeepSeek-R1 incentivizes reasoning in LLMs through reinforcement learning. *Nature*, 2025.

[13] Gurobi Optimization, LLC. Gurobi optimizer reference manual. https://www.gurobi.com/documentation/, 2024.

[14] Cunchen Hu, Heyang Huang, Liangliang Xu, Xusheng Chen, Chenxi Wang, Jiang Xu, Shuang Chen, Hao Feng, Sa Wang, Yungang Bao, Ninghui Sun, and Yizhou Shan. ShuffleInfer: Disaggregate LLM inference for mixed downstream workloads. *ACM Trans. Archit. Code Optim.*, 2025.

[15] Qinghao Hu, Peng Sun, Shengen Yan, Yonggang Wen, and Tianwei Zhang. Characterization and prediction of deep learning workloads in large-scale GPU datacenters. In *Proc. ACM/IEEE SC*, 2021.

[16] Qinghao Hu, Zhisheng Ye, Zerui Wang, Guoteng Wang, Meng Zhang, Qiaoling Chen, Peng Sun, Dahua Lin, Xiaolin Wang, Yingwei Luo, Yonggang Wen, and Tianwei Zhang. Characterization of large language model development in the datacenter. In *Proc. USENIX NSDI*, 2024.

[17] Myeongjae Jeon, Shivaram Venkataraman, Amar Phanishayee, Junjie Qian, Wencong Xiao, and Fan Yang. Analysis of large-scale multi-tenant GPU clusters for DNN training workloads. In *Proc. USENIX ATC*, 2019.

[18] Youhe Jiang, Ran Yan, Xiaozhe Yao, Yang Zhou, Beidi Chen, and Binhang Yuan. Hexgen: generative inference of large language model over heterogeneous environment. In *Proc. ICML*, 2024.

[19] Ziheng Jiang, Haibin Lin, Yinmin Zhong, Qi Huang, Yangrui Chen, Zhi Zhang, Yanghua Peng, Xiang Li, Cong Xie, Shibiao Nong, Yulu Jia, Sun He, Hongmin Chen, Zhihao Bai, Qi Hou, Shipeng Yan, Ding Zhou, Yiyao Sheng, Zhuo Jiang, Haohan Xu, Haoran Wei, Zhang Zhang, Pengfei Nie, Leqi Zou, Sida Zhao, Liang Xiang, Zherui Liu, Zhe Li, Xiaoying Jia, Jianxi Ye, Xin Jin, and Xin Liu. MegaScale: Scaling large language model training to more than 10,000 GPUs. In *Proc. USENIX NSDI*, 2024.

[20] Kubernetes. Assigning pods to nodes. https://kubernetes.io/docs/concepts/scheduling-eviction/assign-pod-node/, 2025.

[21] Kubernetes. Kubernetes: Production-grade container orchestration. https://kubernetes.io/, 2025.

[22] Kubernetes. Resource management for pods and containers. https://kubernetes.io/docs/concepts/configuration/manage-resources-containers/, 2025.

[23] Kubernetes Documentation. Labels and selectors. https://kubernetes.io/docs/concepts/overview/working-with-objects/labels/, 2026.

[24] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph Gonzalez, Hao Zhang, and Ion Stoica. Efficient memory management for large language model serving with pagedattention. In *Proc. ACM SOSP*, 2023.

[25] Black Forest Labs. FLUX. https://github.com/black-forest-labs/flux, 2024.

[26] Charles E. Leiserson. Fat-trees: Universal networks for hardware-efficient supercomputing . *IEEE Transactions on Computers*, 1985.

[27] Suyi Li, Hanfeng Lu, Tianyuan Wu, Minchen Yu, Qizhen Weng, Xusheng Chen, Yizhou Shan, Binhang Yuan, and Wei Wang. TOPPINGS: CPU-assisted, rank-aware adapter serving for LLM inference. In *Proc. USENIX ATC*, 2025.

[28] Suyi Li, Luping Wang, Wei Wang, Yinghao Yu, and Bo Li. George: Learning to place long-lived containers in large clusters with operation constraints. In *Proc. ACM SoCC*, 2021.

[29] Suyi Li, Wei Wang, Jun Yang, Guangzhen Chen, and Daohe Lu. Golgi: Performance-aware, resource-efficient function scheduling for serverless computing. In *Proc. ACM SoCC*, 2023.

[30] Suyi Li, Lingyun Yang, Xiaoxiao Jiang, Hanfeng Lu, Dakai An, Zhipeng Di, Weiyi Lu, Jiawei Chen, Kan Liu, Yinghao Yu, Tao Lan, Guodong Yang, Lin Qu, Liping Zhang, and Wei Wang. Katz: Efficient workflow serving for diffusion models with many adapters. In *Proc. USENIX ATC*, 2025.

[31] Xinyu Lian, Sam Ade Jacobs, Lev Kurilenko, Masahiro Tanaka, Stas Bekman, Olatunji Ruwase, and Minjia Zhang. Universal checkpointing: A flexible and efficient distributed checkpointing system for Large-Scale DNN training with reconfigurable parallelism. In *Proc. USENIX ATC*, 2025.

[32] Kshiteej Mahajan, Arjun Balasubramanian, Arjun Singhvi, Shivaram Venkataraman, Aditya Akella, Amar Phanishayee, and Shuchi Chawla. Themis: Fair and efficient GPU cluster scheduling. In *Proc. USENIX NSDI*, 2020.

[33] Yixuan Mei, Yonghao Zhuang, Xupeng Miao, Juncheng Yang, Zhihao Jia, and Rashmi Vinayak. Helix: Serving large language models over heterogeneous GPUs and network via max-flow. In *Proc. ACM ASPLOS*, 2025.

[34] Deepak Narayanan, Mohammad Shoeybi, Jared Casper, Patrick LeGresley, Mostofa Patwary, Vijay Korthikanti, Dmitri Vainbrand, Prethvi Kashinkunti, Julie Bernauer, Bryan Catanzaro, Amar Phanishayee, and Matei Zaharia. Efficient large-scale language model training on GPU clusters using megatron-LM. In *Proc. ACM/IEEE SC*, 2021.

[35] NVIDIA. NVIDIA Management Library (NVML). https://developer.nvidia.com/management-library-nvml, 2025.

[36] NVIDIA. NVIDIA multi-instance GPU. https://www.nvidia.com/en-us/technologies/multi-instance-gpu/, 2025.

[37] NVIDIA. NVIDIA multi-process service. https://docs.nvidia.com/deploy/mps/index.html, 2025.

[38] OpenAI. Triton. https://github.com/triton-lang/triton, 2025.

[39] Pratyush Patel, Esha Choukse, Chaojie Zhang, Aashaka Shah, Íñigo Goiri, Saeed Maleki, and Ricardo Bianchini. Splitwise: Efficient generative LLM inference using phase splitting. In *Proc. ACM/IEEE ISCA*, 2024.

[40] F. Petrini and M. Vanneschi. k-ary n-trees: high performance networks for massively parallel architectures. In *Proc. IPPS*, 1997.

[41] Dustin Podell, Zion English, Kyle Lacey, Andreas Blattmann, Tim Dockhorn, Jonas Müller, Joe Penna, and Robin Rombach. SDXL: Improving latent diffusion models for high-resolution image synthesis. In *Proc. ICLR*, 2024.

[42] Qwen Team. Qwen3-Next revolutionary AI model architecture. https://qwen3-next.com/, 2025.

[43] Abhishek Verma, Madhukar Korupolu, and John Wilkes. Evaluating job packing in warehouse-scale computing. In *Proc. IEEE CLUSTER*, 2014.

[44] vLLM. Benchmark CLI. https://docs.vllm.ai/en/stable/benchmarking/cli/, 2025.

[45] Patrick von Platen, Suraj Patil, Anton Lozhkov, Pedro Cuenca, Nathan Lambert, Kashif Rasul, Mishig Davaadorj, Dhruv Nair, Sayak Paul, William Berman, Yiyi Xu, Steven Liu, and Thomas Wolf. Diffusers: State-of-the-art diffusion models. https://github.com/huggingface/diffusers, 2022.

[46] Borui Wan, Mingji Han, Yiyao Sheng, Yanghua Peng, Haibin Lin, Mofan Zhang, Zhichao Lai, Menghan Yu, Junda Zhang, Zuquan Song, Xin Liu, and Chuan Wu. ByteCheckpoint: A unified checkpointing system for large foundation model development. In *Proc. USENIX NSDI*, 2025.

[47] Borui Wan, Gaohong Liu, Zuquan Song, Jun Wang, Yun Zhang, Guangming Sheng, Shuguang Wang, Houmin Wei, Chenyuan Wang, Weiqiang Lou, Xi Yang, Mofan Zhang, Kaihua Jiang, Cheng Ren, Xiaoyun Zhi, Menghan Yu, Zhe Nan, Zhuolin Zheng, Baoquan Zhong, Qinlong Wang, Huan Yu, Jinxin Chi, Wang Zhang, Yuhan Li, Zixian Du, Sida Zhao, Yongqiang Zhang, Jingzhe Tang, Zherui Liu, Chuan Wu, Yanghua Peng, Haibin Lin, Wencong Xiao, Xin Liu, and Liang Xiang. Robust LLM training infrastructure at ByteDance. In *Proc. ACM SOSP*, 2025.

[48] Team Wan, Ang Wang, Baole Ai, Bin Wen, Chaojie Mao, Chen-Wei Xie, Di Chen, Feiwu Yu, Haiming Zhao, Jianxiao Yang, Jianyuan Zeng, Jiayu Wang, Jingfeng Zhang, Jingren Zhou, Jinkai Wang, Jixuan Chen, Kai Zhu, Kang Zhao, Keyu Yan, Lianghua Huang, Mengyang Feng, Ningyi Zhang, Pandeng Li, Pingyu Wu, Ruihang Chu, Ruili Feng, Shiwei Zhang, Siyang Sun, Tao Fang, Tianxing Wang, Tianyi Gui, Tingyu Weng, Tong Shen, Wei Lin, Wei Wang, Wei Wang, Wenmeng Zhou, Wente Wang, Wenting Shen, Wenyuan Yu, Xianzhong Shi, Xiaoming Huang, Xin Xu, Yan Kou, Yangyu Lv, Yifei Li, Yijing Liu, Yiming Wang, Yingya Zhang, Yitong Huang, Yong Li, You Wu, Yu Liu, Yulin Pan, Yun Zheng, Yuntao Hong, Yupeng Shi, Yutong Feng, Zeyinzi Jiang, Zhen Han, Zhi-Fan Wu, and Ziyu Liu. Wan: Open and advanced large-scale video generative models. *arXiv preprint arXiv:2503.20314*, 2025.

[49] Yujie Wang, Shiju Wang, Shenhan Zhu, Fangcheng Fu, Xinyi Liu, Xuefeng Xiao, Huixia Li, Jiashi Li, Faming Wu, and Bin Cui. FlexSP: Accelerating large language model training via flexible sequence parallelism. In *Proc. ACM ASPLOS*, 2025.

[50] Yuxin Wang, Duanyu Feng, Yongfu Dai, Zhengyu Chen, Jimin Huang, Sophia Ananiadou, Qianqian Xie, and Hao Wang. HARMONIC: Harnessing LLMs for tabular data synthesis and privacy protection. In *Proc. NeurIPS*, 2024.

[51] Qizhen Weng, Wencong Xiao, Yinghao Yu, Wei Wang, Cheng Wang, Jian He, Yong Li, Liping Zhang, Wei Lin, and Yu Ding. MLaaS in the wild: Workload analysis and scheduling in large-scale heterogeneous GPU clusters. In *Proc. USENIX NSDI*, 2022.

[52] Qizhen Weng, Lingyun Yang, Yinghao Yu, Wei Wang, Xiaochuan Tang, Guodong Yang, and Liping Zhang. Beware of fragmentation: Scheduling GPU-sharing workloads with fragmentation gradient descent. In *Proc. USENIX ATC*, 2023.

[53] Bingyang Wu, Zili Zhang, Zhihao Bai, Xuanzhe Liu, and Xin Jin. Transparent GPU sharing in container clouds for deep learning workloads. In *Proc. USENIX NSDI*, 2023.

[54] Yuxing Xiang, Xue Li, Kun Qian, Wenyuan Yu, Ennan Zhai, and Xin Jin. ServeGen: Workload characterization and generation of large language model serving in production. In *Proc. USENIX NSDI*, 2026.

[55] Wencong Xiao, Shiru Ren, Yong Li, Yang Zhang, Pengyang Hou, Zhi Li, Yihui Feng, Wei Lin, and Yangqing Jia. AntMan: Dynamic scaling on GPU clusters for deep learning. In *Proc. USENIX OSDI*, 2020.

[56] Jiarong Xing, Yifan Qiao, Simon Mo, Xingqi Cui, Gur-Eyal Sela, Yang Zhou, Joseph Gonzalez, and Ion Stoica. Towards efficient and practical GPU multitasking in the era of LLM. *arXiv preprint arXiv:2508.08448*, 2025.

[57] An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengen Huang, Chenxu Lv, Chujie Zheng, Dayiheng Liu, Fan Zhou, Fei Huang, Feng Hu, Hao Ge, Haoran Wei, Huan Lin, Jialong Tang, Jian Yang, Jianhong Tu, Jianwei Zhang, Jianxin Yang, Jiaxi Yang, Jing Zhou, Jingren Zhou, Junyang Lin, Kai Dang, Keqin Bao, Kexin Yang, Le Yu, Lianghao Deng, Mei Li, Mingfeng Xue, Mingze Li, Pei Zhang, Peng Wang, Qin Zhu, Rui Men, Ruize Gao, Shixuan Liu, Shuang Luo, Tianhao Li, Tianyi Tang, Wenbiao Yin, Xingzhang Ren, Xinyu Wang, Xinyu Zhang, Xuancheng Ren, Yang Fan, Yang Su, Yichang Zhang, Yinger Zhang, Yu Wan, Yuqiong Liu, Zekun Wang, Zeyu Cui, Zhenru Zhang, Zhipeng Zhou, and Zihan Qiu. Qwen3 technical report. *arXiv preprint arXiv:2505.09388*, 2025.

[58] An Yang, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chengyuan Li, Dayiheng Liu, Fei Huang, Haoran Wei, Huan Lin, Jian Yang, Jianhong Tu, Jianwei Zhang, Jianxin Yang, Jiaxi Yang, Jingren Zhou, Junyang Lin, Kai Dang, Keming Lu, Keqin Bao, Kexin Yang, Le Yu, Mei Li, Mingfeng Xue, Pei Zhang, Qin Zhu, Rui Men, Runji Lin, Tianhao Li, Tianyi Tang, Tingyu Xia, Xingzhang Ren, Xuancheng Ren, Yang Fan, Yang Su, Yichang Zhang, Yu Wan, Yuqiong Liu, Zeyu Cui, Zhenru Zhang, and Zihan Qiu. Qwen2.5 technical report. *arXiv preprint arXiv:2412.15115*, 2025.

[59] Lingyun Yang, Yongchen Wang, Yinghao Yu, Qizhen Weng, Jianbo Dong, Kan Liu, Chi Zhang, Yanyi Zi, Hao Li, Zechao Zhang, Nan Wang, Yu Dong, Menglei Zheng, Lanlan Xi, Xiaowei Lu, Liang Ye, Guodong Yang, Binzhang Fu, Tao Lan, Liping Zhang, Lin Qu, and Wei Wang. GPU-disaggregated serving for deep learning recommendation models at scale. In *Proc. USENIX NSDI*, 2025.

[60] Shan Yu, Jiarong Xing, Yifan Qiao, Mingyuan Ma, Yangmin Li, Yang Wang, Shuo Yang, Zhiqiang Xie, Shiyi Cao, Ke Bao, Ion Stoica, Harry Xu, and Ying Sheng. Prism: Unleashing GPU sharing for cost-efficient multi-LLM serving. *arXiv preprint arXiv:2505.04021*, 2025.

[61] Binhang Yuan, Yongjun He, Jared Quincy Davis, Tianyi Zhang, Tri Dao, Beidi Chen, Percy Liang, Christopher Re, and Ce Zhang. Decentralized training of foundation models in heterogeneous environments. In *Proc. NeurIPS*, 2022.

[62] Susan Zhang, Stephen Roller, Naman Goyal, Mikel Artetxe, Moya Chen, Shuohui Chen, Christopher Dewan, Mona Diab, Xian Li, Xi Victoria Lin, et al. OPT: Open pre-trained transformer language models. *arXiv preprint arXiv:2205.01068*, 2022.

[63] Hanyu Zhao, Zhenhua Han, Zhi Yang, Quanlu Zhang, Fan Yang, Lidong Zhou, Mao Yang, Francis C.M. Lau, Yuqi Wang, Yifan Xiong, and Bin Wang. HiveD: Sharing a GPU cluster for deep learning with guarantees. In *Proc. USENIX OSDI*, 2020.

[64] Lianmin Zheng, Liangsheng Yin, Zhiqiang Xie, Chuyue Sun, Jeff Huang, Cody Hao Yu, Shiyi Cao, Christos Kozyrakis, Ion Stoica, Joseph E. Gonzalez, Clark Barrett, and Ying Sheng. SGLang: Efficient execution of structured language model programs. In *Proc. NeurIPS*, 2024.

[65] Yinmin Zhong, Shengyu Liu, Junda Chen, Jianbo Hu, Yibo Zhu, Xuanzhe Liu, Xin Jin, and Hao Zhang. DistServe: Disaggregating prefill and decoding for goodput-optimized large language model serving. In *Proc. USENIX OSDI*, 2024.

[66] Guorui Zhou, Xiaoqiang Zhu, Chenru Song, Ying Fan, Han Zhu, Xiao Ma, Yanghui Yan, Junqi Jin, Han Li, and Kun Gai. Deep interest network for click-through rate prediction. In *Proc. ACM SIGKDD*, 2018.

A MILP Formulation

Here, we present the MILP formulation of the preemptive scheduling problem described in §4.2.

Consider a set of tasks $\mathcal{T} = \{\tau_1, \tau_2, \cdots, \tau_{|\mathcal{T}|}\}$ submitted to a cluster of M nodes $\mathcal{G} = \{n_1, n_2, \cdots, n_M\}$, where each node n_j possesses S_j GPUs. Each task is defined as a tuple $\tau_i = \langle w_i, g_i, \zeta_i, \psi_i, \iota_i \rangle$, requesting w_i pods with g_i GPUs each. The task type is denoted by $\zeta_i \in \{0,1\}$ (representing LP/spot and HP, respectively), and the task includes D checkpoint milestones $\psi_i = \{c_{i,1}, \cdots, c_{i,D}\}$.

Additionally, let $\iota_i = \{\langle t^s_{i,1}, t^e_{i,1}, f_{i,1} \rangle, \cdots, \langle t^s_{i,E_i}, t^e_{i,E_i}, f_{i,E_i} \rangle\}$ represent the set of E_i runtime logs for task τ_i. Each log corresponds to the k-th execution run, starting at $t^s_{i,k}$, ending at $t^e_{i,k}$, and achieving the $f_{i,k}$-th checkpoint. The scheduler aims to optimize the following dual objectives, formulated as a mixed-integer linear programming problem over a discrete time horizon $t \in \{1, \ldots, T\}$:

$$\min \quad \frac{\sum_{i=1}^{|\mathcal{T}|} \sum_{j=1}^{M} \sum_{t=1}^{T} x_{i,j,t}(E_i - 1)}{\sum_{i=1}^{|\mathcal{T}|} \sum_{j=1}^{M} \sum_{t=1}^{T} x_{i,j,t} E_i} - \alpha \cdot \frac{\sum_{i=1}^{|\mathcal{T}|} \sum_{j=1}^{M} \sum_{t=1}^{T} x_{i,j,t} w_i g_i c_{i,f_{i,E_i}}}{\sum_{j=1}^{M} S_j T} \tag{2a}$$

$$\text{s.t.} \quad \sum_{i=1}^{|\mathcal{T}|} x_{i,j,t} g_i \le S_j, \quad \forall j,t \tag{2b}$$

$$\sum_{j=1}^{M} x_{i,j,t} = w_i, \quad \forall i,k, \forall t \in [t^s_{i,k}, t^e_{i,k}) \tag{2c}$$

$$(E_i - 1)\zeta_i = 0, \quad \forall i \tag{2d}$$

$$\sum_{j=1}^{M} x_{i,j,t^e_{i,k}} = 0, \quad \forall i,k \tag{2e}$$

$$c_{i,f_{i,k}} - c_{i,f_{i,k-1}} \le t^e_{i,k} - t^s_{i,k} < c_{i,f_{i,k}+1} - c_{i,f_{i,k-1}}, \quad \forall i,k \tag{2f}$$

$$x_{i,j,t} \in \mathbb{Z}_{\ge 0}, \quad \forall i,j,t \tag{2g}$$

where $x_{i,j,t}$ is a decision variable representing the number of pods assigned to task τ_i on node n_j at time t, and α is a weighting factor. Constraint (2b) enforces physical GPU limits per node. Constraint (2c) ensures gang-scheduling requirements during each active run. Constraints (2d) and (2e) enforce strict priority, restricting evictions and interruptions to spot tasks only. Constraint (2f) correlates task duration with completed checkpoints; by convention we set $f_{i,0} \triangleq 0$ and $c_{i,0} \triangleq 0$ so that the constraint is well-defined for $k = 1$. Finally, Constraint (2g) restricts the decision variables to non-negative integers.

ARTIFACT EVALUATED
USENIX
AVAILABLE

MIMESYS: Generating Realistic Executable Testing Environments from Resource Usage Traces

Donghyun Kim Zichao Hu Joydeep Biswas Aditya Akella Daehyeok Kim
The University of Texas at Austin

Abstract

Testing applications under realistic resource contention is challenging because production workloads are often inaccessible due to privacy and proprietary concerns. Existing approaches either use simplistic resource stressors that fail to capture temporal dynamics and multi-resource interactions, rely on limited benchmark suites, or require exhaustive per-application profiling. This paper explores an alternative direction: Synthesizing executable workloads from resource usage traces to reproduce realistic colocation scenarios.

We present MIMESYS, a system that transforms time-series resource usage traces into executable workloads that emulate resource contention patterns. MIMESYS represents emulated workloads as compositions of resource stressors and employs a diffusion-based generative model to learn the inverse mapping from traces to stressor compositions. We introduce two key ideas: state-aware conditioning that conditions generation on both target traces and prior system state to capture temporal dependencies, and execution-driven alignment that adapts the model to real application patterns using direct execution feedback without requiring ground-truth labels. Our evaluation shows that MIMESYS achieves up to 5.5× higher trace similarity and reproduces application performance under contention 2.6× more accurately than baselines.

1 Introduction

Access to real-world production applications is essential for many research and engineering tasks. Researchers need realistic workloads to evaluate system designs, architectures, and optimizations. Developers require representative applications to test their software under production-like conditions. Cluster operators need diverse workloads to benchmark hardware configurations and inform capacity planning decisions.

Unfortunately, obtaining access to production applications is often impractical. Privacy regulations prevent organizations from sharing applications that process sensitive user data, while proprietary concerns make companies reluctant to disclose intellectual property. Even when sharing is permitted, applications often have complex dependencies on internal infrastructure that make them difficult to package and distribute.

As a result, the community today relies on imperfect alternatives. Public resource usage traces from production systems [29,54] capture metrics such as CPU utilization, memory bandwidth, cache usage, and I/O throughput, but they provide only aggregate system behavior without executable workloads.

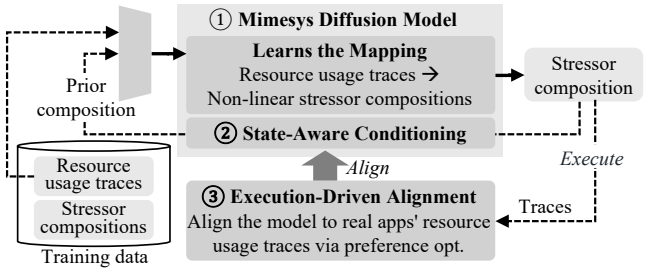


Figure 1: *Overview of* MIMESYS *with its key ideas.*

Benchmark suites [8,47,48] offer reproducibility but are difficult to generalize beyond their limited coverage and lack diversity. Testing directly in production is resource-intensive, difficult to scale, and fails to provide reproducible conditions as workload contention evolves over time. Synthetic workload generation offers a more promising path [44], but existing approaches fall short: resource stressors [3,26] are typically used in simplistic ways that poorly capture temporal variations and multi-resource interactions, while application cloning techniques [27,28] require exhaustive per-application profiling and primarily reproduce coarse-grained statistical patterns rather than fine-grained temporal dynamics.

This paper asks: *Can we reverse-engineer executable workloads from resource usage traces?* Cluster operators routinely collect such traces from production environments, capturing resource usage patterns over time. If we could synthesize executable workloads that reproduce these patterns, we could enable realistic testing and experimentation without requiring access to actual production applications. This would provide controlled, reproducible environments while preserving privacy and protecting proprietary information.

We explore this question in a concrete context: reproducing realistic environments for testing applications under resource contention from co-resident workloads. In production deployments, colocated workloads compete for shared resources such as CPU, memory bandwidth, cache, and I/O, creating contention that can degrade application performance. This contention remains hidden when testing in isolation, leading developers to deploy applications that perform poorly or violate service-level objectives under real deployment conditions. An ideal solution would synthesize workloads at the application-logic level, reproducing not only system-level resource usage but also fine-grained characteristics such as memory access patterns and microarchitectural behaviors.

We present MIMESYS, a system that takes a practical first step toward this vision by transforming resource usage traces into executable workloads that emulate realistic colocation scenarios (Figure 1). While synthesizing application logic from traces remains infeasible, we observe that for reproducing resource contention, emulating resource usage patterns suffices without recovering the underlying application logic. MIMESYS represents emulated workloads as *compositions of parameterized resource stressors* and learns the complex, nonlinear mapping from resource usage traces to stressor compositions. By using stressors as expressive building blocks rather than simple, pre-configured tools, MIMESYS can approximate diverse, realistic resource usage patterns in a practical and scalable manner. We employ a *diffusion model* to learn this mapping because it can capture complex, multimodal distributions over stressor compositions (① in Figure 1).

Realizing this approach entails two challenges. First, stressor compositions exhibit complex temporal and cross-resource dependencies, where a stressor's impact depends on concurrent execution and prior system state. Second, real applications exhibit structured behaviors driven by their computational logic that differ from arbitrary stressor combinations, requiring explicit alignment beyond synthetic training data.

We address these challenges with two key ideas. First, we introduce *state-aware conditioning* (②) that conditions generation not just on target traces but also on prior system state. Unlike conventional diffusion models, our approach incorporates previously generated stressor compositions and their resulting resource usage into the conditioning mechanism, enabling the model to capture temporal dependencies. Second, we devise *execution-driven alignment* (③), a training approach that adapts the model to real application behaviors without requiring ground-truth stressor labels. Inspired by preference optimization [11, 24, 37] in generative models, this approach learns from unlabeled real application traces through direct execution feedback, bridging the gap between synthetic training data and real application traces.

We implement MIMESYS in Python for model training and inference and C++ for the executable generator. The diffusion model uses a U-Net architecture [42] with 8.9M parameters and trains in ≈4 hours on one NVIDIA A100 GPU. The executable generator extends Fleetbench [8] with stress-ng [26] stressors, supporting diverse resource usage patterns across CPU, memory, cache, and I/O. We open-source MIMESYS at https://github.com/ldos-project/mimesys.

Our evaluation shows that MIMESYS generates emulated workloads that closely reproduce multi-resource usage patterns of diverse workload mixes. Using the Dynamic Time Warping distance metric [10], MIMESYS achieves up to 5.5× higher similarity compared to stressor-based baselines. We further demonstrate that MIMESYS accurately emulates application performance under contention across multiple benchmarks. For example, MIMESYS reproduces throughput degradation in database workloads with an average error of 8.3% points, outperforming baselines by 2.6×.

2 Background and Motivation

2.1 Need for Realistic Testing Environments

To evaluate whether an application will perform well in practice, application developers need to test it in realistic environments. An environment can be defined by two components: (1) the hardware configuration (*e.g.,* CPU, memory, storage, network bandwidth) and (2) the collocated workloads that run alongside the target application. For a given hardware configuration, realistic colocated workloads are essential because they share computing resources with the target application, creating resource contention that can affect performance in ways that remain hidden when testing in isolation. This resource contention, arising from competition for CPU, memory bandwidth, cache, and I/O devices, is a defining characteristic of production deployments where applications are colocated with other workloads on shared infrastructure.

The need for such realistic testing environments is particularly pronounced in cloud settings, where providers colocate multiple tenant applications on shared physical hosts to improve resource utilization. This colocation creates resource contention that makes performance unpredictable [39, 52, 57], where colocated workloads unintentionally interfere with one another as noisy neighbors. Application developers need to test their applications against these realistic contention patterns before deployment to ensure their applications will perform acceptably under production conditions. For example, TUNA [19] demonstrates the value of testing under realistic noise by modeling resource contention from noisy neighbors during database configuration tuning, enabling it to identify configurations that remain robust when deployed.

Beyond simply exposing applications to realistic contention, developers also need these testing environments to be reproducible. Reproducibility enables developers to systematically evaluate different application configurations, conduct fair performance comparisons, and validate optimizations under consistent conditions. However, prior studies [18, 21, 51] highlight the difficulty of achieving such reproducibility in shared environments where resource contention varies unpredictably. Without reproducible testing conditions, developers struggle to isolate the effects of their changes from environmental noise, making it difficult to draw reliable conclusions from their experiments.

Difficulties in obtaining realistic colocated workloads. Unfortunately, obtaining realistic colocated workloads is challenging. One straightforward approach would be to use actual workloads from production deployments. However, actual tenant workloads cannot be shared due to privacy and proprietary concerns. Even when developers have access to production environments for testing, they face two critical limitations.

First, it is resource-intensive and difficult to scale. Developers must deploy their applications into production-like infrastructure to experience realistic contention. For instance, TUNA requires actual deployment to measure performance impact from noisy neighbors, demanding substantial infrastructure (10 nodes over 8 hours per run) that becomes impractical when many developers need to test diverse applications.

Second, production environments evolve continuously as workloads change, making it difficult to achieve reproducible testing conditions. The contention patterns observed in production represent specific deployment states that may change over time, preventing developers from conducting consistent experiments across different configurations or optimizations.

2.2 Limitations of Existing Approaches

Given the difficulties in obtaining realistic workloads from production environments, synthetic approaches offer an attractive alternative. Rather than using actual production workloads, these approaches generate controlled workloads or inject artificial interference to mimic resource contention effects. Developers can use synthetic workloads to reproduce stress conditions in a controlled manner while avoiding the infrastructure costs and privacy concerns of real-world testing.

However, existing approaches exhibit limitations in accurately reproducing realistic deployment conditions.

Resource stressors. Resource stressors such as stress-ng [26] and Intel MLC [3] are commonly used to inject resource contention. These tools intentionally stress specific resources (*e.g.,* CPU or memory) for a given duration, enabling researchers to study how systems behave under abnormal conditions [35, 46, 57]. However, stressors are typically used in simple ways: practitioners manually select a few stressors to run continuously at maximum intensity for fixed duration. As shown in §7, this naïve usage poorly captures the realistic conditions operators observe, where applications exhibit temporal variations in resource usage (*e.g.,* periodic bursts, gradual ramps) and complex multi-resource interactions (*e.g.,* memory access patterns that depend on CPU load).

Application benchmarks. Standard benchmarks such as SPEC [47], DCPerf [48], and Fleetbench [8] are carefully designed by domain experts to represent realistic workloads. While benchmarks offer better realism than stressors, they cover only a limited subset of real-world application behaviors. Furthermore, diversifying benchmarks to cover new workload types is challenging: each new benchmark requires manual effort to implement applications that reflect evolving workload characteristics, making it difficult to keep pace with the diversity of modern applications.

Application cloning. Application cloning techniques [27, 28] generate synthetic programs that replicate the performance and resource usage profiles of complex or proprietary applications without exposing their internal logic. They profile target applications across the system stack (*e.g.,* I/O operations, kernel activities) to capture their statistical resource usage properties, then construct synthetic programs that exhibit similar characteristics. However, achieving such fidelity requires exhaustive profiling for each application, making it costly to extend cloning to a diverse set of applications. Furthermore, cloning methods primarily reproduce coarse-grained statistical patterns (*e.g.,* mean or median resource utilization) rather than fine-grained temporal dynamics. This focus on statistical characteristics means cloned applications cannot effectively capture the dynamic, time-varying resource usage that characterizes real workloads.

3 Design Overview

To address the limitations of existing synthetic approaches while avoiding the costs of real-world deployments, we propose MIMESYS, a system that transforms *resource usage traces* into *executable workloads* that run alongside a developer's target application to emulate realistic collocated workloads. Resource usage traces capture system resource consumption over time, recording metrics such as CPU utilization, memory bandwidth, cache usage, and I/O throughput. Our key insight is that while recovering individual application logic from traces is infeasible, we can synthesize executables that reproduce similar time-series resource usage patterns when executed on the same or similar hardware. Unlike application cloning, which targets individual applications and requires exhaustive per-application profiling, MIMESYS synthesizes *environment-level* workloads from aggregate resource usage traces. This enables realistic synthetic test environments without requiring access to actual production workloads.

3.1 Our Approach

MIMESYS is built on two key ideas.

Stressor composition. We represent emulated workloads as *compositions* of parameterized resource stressors. As discussed in §2, resource stressors generate workloads designed to exercise specific system resources (*e.g.,* CPU, memory, cache, I/O). We choose stressors as building blocks because they are already in an executable form and can be mixed and combined to create diverse stress conditions, making them practical for generating many different emulated workloads.

Learning trace-to-stressor composition mapping. However, mapping resource usage traces to stressor compositions is challenging because the relationship between stressor parameters and observed system behavior is complex and nonlinear. Simple approaches, such as linearly scaling stressor intensities and concurrency to match target utilization, fail for most resources. For example, our measurements show that interleaving a memory stressor with sleep periods to reduce its average intensity does not produce proportional memory bandwidth reductions; the actual bandwidth usage differs from the target by more than 3× due to caching effects and memory controller policies. Concurrency further complicates this relationship: running a memory stressor with an 8 MiB

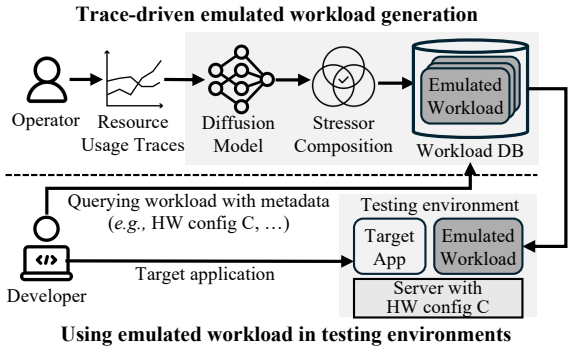


Figure 2: MIMESYS *workflow. Operators use* MIMESYS *to generate emulated workloads from resource usage traces collected in production environments. Developers retrieve emulated workloads matching their target hardware configuration and execute them alongside their applications to test under realistic colocated workloads.*

working set across eight threads increases memory bandwidth usage by 1800× compared to a single thread. These emergent behaviors make it difficult to analytically derive stressor parameters from target resource usage patterns.

To address this complexity, MIMESYS employs a *diffusion-based generative model* [22] that learns the mapping from traces to stressor compositions. Diffusion models can capture complex, multimodal distributions over each stressor's parameters and support conditioning on high-dimensional, long-horizon time-series resource usage traces, enabling trace-driven synthesis of emulated workloads. Our model divides resource usage traces into fixed-duration time windows and generates a stressor composition for each window.

Workflow. Figure 2 illustrates how MIMESYS can be used. Cloud operators (*e.g.,* Azure, AWS, or CloudLab) capture resource usage traces from their production environments and feed them into MIMESYS, which generates a set of emulated workloads as executables. Each workload is labeled with metadata such as hardware configuration and the time period when the traces were collected. The operator can then release these emulated workloads publicly. Application developers retrieve emulated workloads that match the desired deployment characteristics they want to test against. They run these workloads on a hardware environment that is the same or similar to the one described in the metadata, alongside their target application in the testing environment.

This model is analogous to how operators release various forms of traces today (*e.g.,* packet traces [1], flow-level traces [43], microservice call graph traces [29]). By sharing emulated workloads instead of raw traces, operators provide developers with directly executable artifacts that reproduce realistic resource contention while preserving the privacy and proprietary nature of the actual production workloads. Specifically, MIMESYS preserves tenant-sensitive information such as application logic, proprietary algorithms, and business data.

3.2 Challenges

While the diffusion-based stressor composition approach is promising, realizing it entails two main challenges:

C1: Capturing temporal and cross-resource usage dependencies. As mentioned in §3.1, the relationship between stressor compositions and resulting resource usage is inherently nonlinear and context-dependent. A stressor's impact on resources depends not only on its own composition but also on what other stressors are running concurrently and what execution has occurred previously. For example, prior memory accesses can warm the cache hierarchy, drastically reducing subsequent memory bandwidth utilization for the same stressor composition. Similarly, CPU contention from co-running stressors alters thread scheduling patterns, which in turn affects cache miss rates and memory access patterns. To synthesize realistic workloads, the model must learn these system-level dependencies and generate stressor compositions that produce target resource patterns while accounting for temporal execution context.

C2: Aligning with real applications. MIMESYS collects synthetic training data by executing diverse stressor combinations. However, the space of possible stressor combinations is enormous, making it challenging to ensure that the training data covers resource usage patterns exhibited by real applications. Real applications exhibit structured behaviors driven by their computational logic (*e.g.,* periodic memory access in databases, bursty compute phases in scientific workloads) that differ from arbitrary stressor mixes. Although stressor compositions capable of reproducing these patterns may exist, synthetic training data alone provides insufficient supervision for the model to discover them. Without explicit alignment with real application traces, the model will learn only the resource usage distribution of synthetic stressor combinations rather than the patterns exhibited by real applications.

3.3 Key Ideas

To address these challenges, MIMESYS incorporates two key ideas for training the diffusion model.

I1: State-aware conditioning (§4.2). To capture temporal and cross-resource dependencies, we propose a novel adaptation of diffusion models that conditions generation not just on target traces but also on *prior system state*. Unlike typical diffusion models that condition on static inputs (*e.g.,* static text inputs in image generation), our approach makes the model *state-aware* by incorporating prior system state directly into the conditioning mechanism. Rather than generating stressor compositions for all time windows independently, the model generates them sequentially, conditioning each window on: (1) target resource usage traces across multiple metrics, capturing cross-resource dependencies, and (2) a previously generated stressor composition, reflecting execution temporal dependencies. This enables the model to account for state-dependent system effects such as cache warming and resource

contention, generating stressor compositions that adapt to how the system state evolves over time.

I2: Execution-driven alignment (§4.3). Inspired by preference optimization in generative models [11,24,37], we devise *execution-driven alignment*, a training approach that adapts diffusion models to match real application behaviors without requiring ground-truth stressor labels. This approach bridges the distribution gap between synthetic training data and real deployment patterns through direct execution feedback. We treat the diffusion model as a policy and apply reinforcement learning to optimize it based on actual system execution. The model iteratively generates stressor compositions from unlabeled real application traces, executes them on hardware, and receives rewards based on how closely the resulting traces match target patterns. Policy gradient methods then update the model to prefer compositions that better reproduce real application behaviors.

3.4 Training and Inference Pipeline

Model training. For training, operators specify the target resource metrics (*e.g.,* CPU utilization, memory bandwidth, last-level cache (LLC) traffic) and candidate stressors. MIMESYS collects training data by executing diverse stressor combinations using novelty-guided data collection (§4.2.3) and profiling their resource usage traces. The diffusion model learns the mapping from resource usage traces to stressor compositions. To capture temporal and cross-resource dependencies, the model conditions generation on both the target trace and prior stressor compositions (§4.2). MIMESYS then adapts the model to real application traces through execution-driven alignment (§4.3), which optimizes the model based on how closely the generated workloads reproduce target resource patterns when executed.

Emulated workload generation. During inference, operators provide target resource usage traces as input. For each time window, the diffusion model generates a stressor composition conditioned on the target resource usage for that window and the previously generated composition. The workload generator then translates the composition sequence into an executable program that orchestrates the stressors across threads and time windows (§5).

4 Training MIMESYS Model

This section formalizes the emulated workload generation problem (§4.1), presents our diffusion model with state-aware conditioning and training data collection (§4.2), and explains how we adapt the model to real application traces using preference optimization (§4.3).

4.1 Problem Formulation

Consider a workload mix P executed on a system that produces multivariate time-series resource usage traces $o_{1:T} = \{o_1, \ldots, o_T\}$, where each $o_t \in \mathbb{R}^N$ represents N resource metrics (*e.g.,* CPU utilization, memory bandwidth, LLC traffic) measured over a fixed time window of size W seconds. In our setting, we do not have access to the original workload P itself. Our goal is to generate an emulated workload $P' = \{a_1, \ldots, a_T\}$ that, when executed on the same hardware, produces resource usage traces similar to those of P.

Stressor composition representation. We represent each emulated workload as a sequence of stressor compositions, where $a_t \in \mathbb{R}^{M \times K}$ denotes the composition of M stressors across K CPU threads for the t^{th} time window. Each entry $a_t[m,k]$ specifies the fraction of W during which stressor m executes on thread k, with values ranging from 0 to 1. For example, $a_t[m,k] = 0.4$ means stressor m runs for $0.4 \cdot W$ seconds on thread k within the t^{th} window. The duration allocated to all stressors on a single thread sum to at most W. The aggregate intensity of a stressor type across the system is determined by how many threads execute it concurrently.

Objective. We say that an emulated workload P' is δ-trace-equivalent to P under a distance metric $\ominus$ (*e.g.,* mean-squared error, Dynamic Time Warping [10]) if the trace produced by executing P', denoted $o'_{1:T}$, satisfies $\|o_{1:T} \ominus o'_{1:T}\| \leq \delta$. The emulated workload generation problem is to learn a function $f_\theta : o_{1:T} \mapsto P' = \{a_1, \ldots, a_T\}$ such that samples $P' \sim f_\theta(o_{1:T})$ produce traces that are δ-trace-equivalent to the observed traces $o_{1:T}$ with high probability. The objective of MIMESYS training is to learn such a mapping that yields δ-trace-equivalent emulated workloads across diverse resource usage patterns.

4.2 MIMESYS Diffusion Model

Straightforward approaches such as regressing stressor compositions directly from traces or linearly interpolating stressor intensities treat each time step in isolation and assume a deterministic, one-to-one mapping. As discussed in §3.2, these fail due to temporal dependencies, cross-resource interactions, and nonlinear resource behavior.

Our approach. We extend *diffusion* models [11,24] to be *state-aware* by conditioning generation not just on target traces but also on prior system state. We call this technique *state-aware conditioning*. Unlike conventional diffusion models that generate outputs based solely on input conditions, our approach incorporates a prior stressor composition directly into the conditioning mechanism. Concretely, instead of learning a direct mapping $o \mapsto a$, we model the conditional distribution $p(a_t \mid o_t, a_{t-1})$, where o_t represents the target trace for the t^{th} time window and a_{t-1} is a stressor composition generated at the $(t-1)^{th}$ window. This enables sequential generation across time windows, where each window's stressor composition is conditioned on both the target resource usage and the system state affected by the prior composition.

4.2.1 Why Use a Diffusion Model?

Several properties of the workload generation problem guide our choice to use diffusion models. The mapping from traces

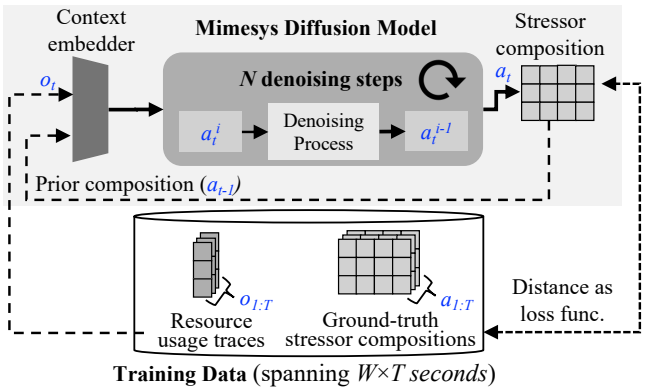


Figure 3: MIMESYS *diffusion modeling. The model is conditioned on input traces and prior stressor compositions to capture temporal and cross-resource dependencies. Each trace and stressor composition corresponds to a time window.*

to stressor compositions is *one-to-many*: many distinct stressor compositions can produce similar resource usage patterns. Diffusion models are well suited to handle such one-to-many mappings through their iterative denoising process [23]. Starting from noise, the model progressively refines samples while exploring multiple feasible stressor compositions before converging to a final output. This iterative refinement naturally captures the complex multimodal conditional distributions that arise from the one-to-many mapping. Furthermore, while traditional search and optimization methods struggle to generalize as the metric space grows (*e.g.,* across dozens of resource metrics), diffusion models efficiently learn the underlying mapping to safely interpolate across workload traces not seen during training.

Importantly, diffusion models support flexible conditioning mechanisms through learned context embeddings [32, 36]. This allows us to realize our state-aware conditioning by incorporating a prior stressor composition directly into the generation process, as we discuss next.

4.2.2 Model Architecture and Training

We now formalize how to extend a diffusion model with state-aware conditioning. The key is to define a conditioning context that captures both the target trace and system state affected by the prior stressor composition, then use this context to guide the iterative denoising process. Figure 3 illustrates our model architecture and training process.

Model interface. To make the diffusion model state-aware, we construct a conditioning context that captures both the target trace and system state. MIMESYS employs a learnable embedder $g(\cdot)$ that encodes the target trace o_t and a prior stressor composition a_{t-1} into a compact context representation: $c = g(o_t, a_{t-1})$. This context embedding summarizes two key components: (1) the target resource usage patterns across multiple metrics, capturing cross-resource dependencies, and (2) how the prior stressor composition has shaped the current system state.

With this conditioning, the model approximates the distribution $\pi_\theta(a_t \mid c)$, where a_t is the stressor composition for the t^{th} time window. Because c incorporates both target traces and the prior stressor composition, the diffusion model generates a_t that accounts for state-dependent effects such as cache warming and resource contention. This formulation enables iterative generation across time windows: after generating a_t, this composition becomes part of the conditioning context c for generating the next window's composition a_{t+1}.

Training objective. The training objective is to align the model's predictions with the ground-truth stressor compositions observed in the training data. At a high level, this corresponds to minimizing the expected discrepancy between predicted and ground-truth stressor compositions:

$$\arg\min_{\theta} \mathbb{E}_{a' \sim \pi_\theta(a|c)} \left[D(a', a^{\text{gt}}) \right],$$

where a^{gt} denotes the ground-truth stressor composition and $D(\cdot,\cdot)$ is a distance metric (*e.g.,* mean-squared error).

In diffusion modeling, we realize this objective by instantiating the general diffusion loss, where the model predicts noise for each denoising step [22]. The clean sample a_t^0 is obtained by denoising N steps from a random noise a_t^N, where a_t^0 corresponds to the ground-truth stressor composition a_t. The training objective aims to minimize the mean-squared error (*i.e.,* distance) between the true noise ε and the predicted noise ε_θ. The resulting training loss for the composition generation is:

$$\mathcal{L} = \mathbb{E}_{a_t^0, \varepsilon, i} \left[\|\varepsilon - \varepsilon_\theta(a_t^i, i, c)\|^2 \right],$$

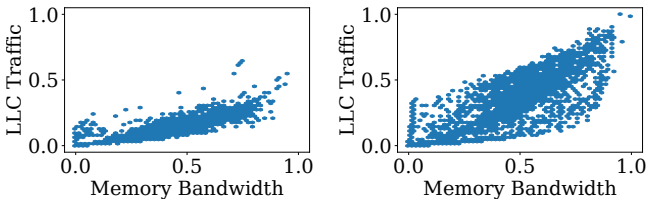
where $\varepsilon_\theta(a_t^i, i, c)$ is the model's noise prediction conditioned on the context embedding c at i^{th} denoising step. By minimizing this loss, the model learns to reverse the noising process: each denoising step refines a stressor composition toward the ground-truth stressor composition paired with the context embedding c. The context embedder $g(\cdot)$ is jointly trained with the diffusion model, so its representations are optimized end-to-end for guiding the denoising process.

During generation, the trained model synthesizes stressor compositions by starting from Gaussian noise and iteratively applying the learned reverse transitions, conditioned on the context embedding (see §5 for details).

4.2.3 Novelty-Guided Training Data Collection

Training our model requires a diverse dataset of traces that spans a wide range of resource-usage patterns across CPU utilization, memory bandwidth, LLC traffic, and disk I/O. Without sufficient coverage of the resource space, the model cannot learn to generate stressor compositions that reproduce diverse or extreme resource behaviors observed in real applications.

Challenge. A straightforward approach is to randomly sample stressor compositions, execute them to collect traces, and construct a dataset of (composition, trace) pairs. However, random sampling produces highly skewed data. As shown in the



(a) *Random sampling* (b) *Novelty-guided collection*

Figure 4: *Joint distribution of memory bandwidth and LLC traffic for 5K training samples. (a) Random sampling produces clustered, skewed coverage. (b) Novelty-guided collection achieves more balanced resource-space coverage.*

left plot of Figure 4, random combinations cluster in narrow regions of the resource space, with many compositions producing similar resource-usage behaviors. This occurs because random composition draws rarely trigger extreme or asymmetric resource patterns (*e.g.,* CPU-intensive but memory-light), causing most samples to collapse into moderate, balanced behaviors. The resulting dataset leaves large portions of the resource space underrepresented, limiting the model's ability to learn mappings for diverse or high-intensity patterns. Because profiling stressor compositions is time-consuming, simply collecting more random samples is impractical.

Our approach. We address this challenge through *novelty-guided training data collection*, which prioritizes profiling stressor compositions likely to produce underexplored resource-usage patterns. The approach collects training data iteratively. At each round, MIMESYS trains a lightweight prediction model (*e.g.,* Random Forest [13] in our implementation) on the current dataset, which consists of all traces collected from previously profiled compositions. This predictor estimates the resource usage that candidate stressor compositions would produce. MIMESYS then assigns each candidate a novelty score based on two complementary factors: (1) how rare its predicted trace is under the current data distribution, measuring underrepresentation, and (2) how uncertain the predictor is about that composition, measured via prediction variance, indicating low-confidence regions where the model needs more data. Compositions with high novelty scores either produce underrepresented behaviors or expose gaps in the predictor's knowledge. MIMESYS selectively profiles the highest-scoring compositions and adds their traces to the training set, iteratively expanding coverage toward underexplored regions.

As shown in the right plot of Figure 4, this approach produces a dataset with substantially better coverage of the joint distribution of memory bandwidth and LLC traffic. The improved diversity enables our model to learn mappings across a more representative range of system behaviors.

4.3 Execution-Driven Alignment

Our diffusion model trained on synthetic stressor combinations learns to map resource-usage traces to stressor compositions across a diverse range of patterns. However, as discussed in §3.2, real applications produce resource-usage patterns shaped by their algorithmic structure that often diverge from those induced by arbitrary synthetic stressor combinations. While the training data may span a broad range of resource-usage values, the temporal structure and cross-resource correlations characteristic of real applications are difficult to capture through synthetic combinations alone. As a result, the pre-trained model may fail to generate stressor sequences that faithfully reproduce real-world application traces, even if it performs well on synthetic test data, as shown in §7.3.

Our approach. To address this gap, we employ *execution-driven alignment*, an approach inspired by preference optimization in large language models [11,24]. The key challenge is that real traces do not come with ground-truth stressor compositions, making standard supervised learning infeasible. Instead of learning from labeled pairs, execution-driven alignment learns from unlabeled real application traces through direct interaction with the system. We treat the diffusion model as a policy π_θ and optimize it to prefer stressor compositions that, when executed, produce traces similar to real applications. The model iteratively generates stressor compositions conditioned on real application traces, executes them on actual hardware to observe their resource usage, and receives rewards based on trace similarity. Policy-gradient methods [30] then update the model weights to maximize these rewards, shifting the distribution toward compositions that better reproduce real application behaviors. This approach bridges the distribution gap between synthetic training data and real deployment patterns without requiring ground-truth labels.

Optimization objective. The goal is to optimize the policy π_θ such that generated stressor compositions, when executed, produce resource-usage traces that closely match those observed in real applications. For each real trace $o \in \mathcal{T}_{\text{real}}$, the policy generates a stressor composition $a' \sim \pi_\theta(a \mid o, a_{\text{prev}})$ conditioned on the target trace o and prior composition a_{prev}, and obtain the corresponding trace o' by profiling. We define a reward function $R(a') = -D(o', o)$ based on the distance between the executed trace o' and the target trace o. For the distance metric D, we use weighted L1 distance, where weights reflect the relative importance of each metric to operators. The optimization objective maximizes the expected reward over the policy distribution:

$$\arg\max_{\theta} \mathbb{E}_{a' \sim \pi_\theta(a|o,a_{\text{prev}})}[R(a')].$$

This objective guides policy updates toward stressor compositions that, under actual execution, produce traces that more closely align with real application behaviors.

Optimization procedure. We realize execution-driven alignment through a reinforcement learning algorithm that updates the policy based on execution-derived rewards. MIMESYS adopts the denoising diffusion policy optimization (DDPO) framework [11, 24], which treats each denoising step in the

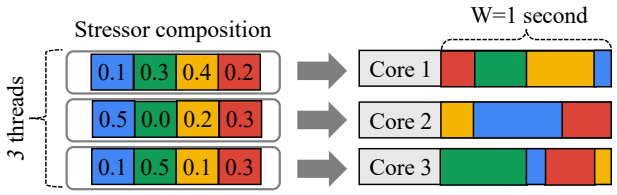


Figure 5: *Converting a stressor composition into an executable workload for a system with three CPU cores. Different colors represent different stressor types. Values specify the fraction of W, allocated to each stressor.*

diffusion model as an action and optimizes the policy via policy-gradient methods [45] to maximize expected rewards.

The adaptation process begins with the pre-trained policy π_θ and a set of real-world traces $\mathcal{T}_{\text{real}}$ provided by the operator. In each iteration, the policy generates a stressor composition $a' \sim \pi_\theta(a \mid o, a_{\text{prev}})$ for a trace o sampled from $\mathcal{T}_{\text{real}}$. The stressor compositions $[a_{\text{prev}}, a']$ are then executed on a system (*i.e.,* server) to collect the resulting trace o' produced by a'. The reward $R(a')$ is computed by measuring the distance (D) between the executed trace o' and the target trace o. The policy gradient method [40, 45] then updates the model parameters to increase the expected reward. This process iterates until convergence or for a fixed number of iterations, producing an adapted model π'_θ that better aligns with real application patterns.

5 Executable Workload Generation

Once the MIMESYS model is trained, operators can use it to generate emulated workloads that reproduce target resource-usage patterns. The generation process operates on time-series resource traces divided into fixed-duration windows, where the window size W is a training hyperparameter determined during model training. For example, given $L_{trace} = 30$ seconds of resource traces and $W = 1$ second, the model produces $T = \frac{L_{trace}}{W} = 30$ time windows. As described in §4.2.2, the model generates a stressor composition a_t for each window t through an iterative diffusion process conditioned on the target trace segment o_t and a prior composition a_{t-1}. This produces a sequence of stressor compositions $a_{1:T}$ that must be translated into an executable workload.

From compositions to executables. Figure 5 illustrates how MIMESYS converts stressor compositions into executables. The left side shows a generated composition a_t represented as a matrix with three rows (one per CPU thread) and four columns (one per stressor type, shown as different colors). Each matrix entry specifies the fraction of the time window allocated to that stressor on that thread. For example, the first row shows $[0.1, 0.3, 0.4, 0.2]$, indicating that thread 1 should execute stressor types 1 through 4 for 10%, 30%, 40%, and 20% of the window duration, respectively.

The executable generator maps each matrix row to a physical CPU core, as shown on the right side of the figure. Within each window (1 second in this example), each core executes its assigned stressors for the specified duration. The colored segments on the right visualize these execution schedules: Core 1 runs the red stressor for 0.2 seconds, the green stressor for 0.3 seconds, and so on. To avoid introducing systematic temporal bias, MIMESYS randomizes the execution order of stressors within each window while preserving their allocated duration. For instance, Core 2 might execute the orange stressor first, then blue, then red, rather than in a fixed order.

MIMESYS generates the final executable as a standalone C++ program that orchestrates stressor execution according to these schedules, as we describe next.

6 Implementation

We implement MIMESYS in approximately 1.8K lines of Python for model training and inference, and 800 lines of C++ for the emulated workload generator. We release MIMESYS at `https://github.com/ldos-project/mimesys`.

Training data collection. For novelty-guided data collection (§4.2), we implement a Random Forest predictor [13] with 100 trees that estimates resource usage from stressor compositions. The novelty score combines prediction uncertainty (measured via variance across trees) and rarity (measured via distance to the nearest training sample in the predicted metric space). We collect training data iteratively, profiling 128 high-novelty compositions per round for 100 rounds, yielding 12K diverse training samples. We profile each data sample multiple times for accurate measurements. Collecting the full training dataset takes 8 hours on 8 machines, though this time varies with the number of samples, window size, and the number of machines used. The profiling infrastructure collects CPU utilization, memory bandwidth, LLC traffic (via hardware performance counters), and disk I/O throughput (via `/proc/diskstats`) at 1000ms granularity using TACC cluster's HPC monitoring tools [49]. The stressor library is fixed during data collection and training; consequently, adding or replacing stressors requires re-profiling and retraining MIMESYS.

Diffusion model. The diffusion model uses a U-Net architecture [42] with 8.9M parameters, where the encoder downsamples the input and the decoder upsamples it to capture how per-thread stressors jointly shape resource usage. For state-aware conditioning (§4.2.2), we employ two MLP-based encoders: a trace encoder that transforms target resource usage vectors into a 256-dimensional embedding via 4 fully-connected layers, and a state encoder that similarly processes prior stressor compositions. These embeddings are concatenated and injected into each U-Net layer to guide the denoising process.

We train the model in PyTorch [31] using the standard noise prediction objective from DDPM [22]. Pretraining runs on one NVIDIA A100 GPU with a learning rate of 1×10^{-4} for 2000 epochs using 25 denoising steps, taking approxi-

mately 2 hours. For execution-driven alignment, we apply DDPO [11] with a learning rate of 3×10^{-6} and the advantage clipping threshold of 3, requiring an additional 2 hours with 8 machines for profiling. We assign equal weights to each resource type in the reward function and use a 1-second time window (W) for both training and inference (see Appendix B for detailed setup). The 1-second window length was selected as a trade-off between data reliability and temporal resolution. Smaller windows resulted in higher variance in collected measurements, whereas larger windows masked fine-grained resource dynamics and increased training data collection time.

From the trained model, generating stressor compositions simply incurs the diffusion model's inference cost. In our setup, it processes roughly 190 time windows per second with a batch size of 128 on an A100 GPU.

Executable generator. We build the executable generator by extending Fleetbench [8] with additional stressors from stress-ng [26], supporting stressor types spanning CPU (integer arithmetic, floating-point operations), memory (sequential/random access with varying working set sizes), cache (LLC-focused access patterns), disk I/O (sequential/random reads and writes), and a `sleep` stressor for idle periods. Each stressor is implemented as a parameterized C++ function. To translate model-generated compositions into executable workloads, we implement a code generation module that produces standalone C++ programs. For each time window, the generator maps stressor parameters to thread-level execution schedules and randomizes stressor execution order within each window to avoid temporal bias.

7 Evaluation

We evaluate MIMESYS to answer four key questions:

- Can MIMESYS reproduce performance degradation caused by noisy neighbor effects (§7.1)?
- How closely do MIMESYS-generated workloads' resource usage patterns match those of real applications (§7.2)?
- How effective are MIMESYS's techniques (§7.3)?
- How sensitive is MIMESYS to input trace variations (§7.4)?

Experimental setup. We evaluate MIMESYS on CloudLab [17] c220g2 machines (Intel Haswell, 20 cores) running Ubuntu 22.04 with Linux kernel 5.15. Input traces consist of per-core CPU utilization, memory bandwidth, LLC traffic, and disk I/O (23 dimensions). We use 14 stressors spanning CPU, memory, cache, and disk I/O resources chosen based on their distinct resource-usage patterns to ensure diversity (see Appendix A for the complete list). To construct this stressor set, we first profiled approximately 600 candidate stressors from Fleetbench and stress-ng, then selected representative stressors based on their resource-usage characteristics to maximize coverage of the resource-usage space while avoiding redundant behaviors.

Category	Workloads	Distributions
Web Serving	DaCapo [12], Renaissance [34]	31%
Big Data	HiBench [2] on Spark [56]	32%
KV Store	YCSB [15] on Redis [5], memcached [4], FASTER [14]	14%
ML Inference	MLPerf Inference [38] (ResNet50 [20], StableDiffusion [41])	11%
Database	TPC-C [6] on Silo [50], TPC-H [7] on PostgreSQL [33]	7%
Graph	GAP Benchmark [9]	5%

Table 1: *Workloads used in the evaluation.*

Workloads. We construct realistic contention scenarios by colocating multiple workloads, each running in a separate VM on Linux/KVM on shared physical machines. We create workload mixes by colocating up to 6 workloads sampled from popular benchmarks (Table 1) according to Azure's workload distribution [16, 53] for realism. To ensure broad coverage of contention scenarios, we balance the mixes by selecting an equal number for each colocation level, controlling the average total load. For each mix, we run workloads for 5 minutes and collect traces across CPU utilization, memory bandwidth, LLC traffic, and disk I/O, totaling 10 hours of diverse workload patterns.

Baselines. We compare MIMESYS against three baselines:
***Search-based*:** For each time window, it selects the stressor composition from the training data with minimum Euclidean distance to the target trace in resource usage space.
***Linear interpolation*:** For each time window, it identifies the 100 nearest training samples in resource usage space and linearly interpolates their stressor compositions to approximate the target trace.
***Single stressor*:** Uses a single resource-targeting stressor with calibrated sleep intervals to match per-core CPU utilization. We use Fleetbench (swissmap) for cache and memory bandwidth, and stress-ng (readahead) for disk I/O.

Metrics. We evaluate fidelity using two metrics. First, *trace similarity* measures how closely synthetic traces match real ones using the Dynamic Time Warping (DTW) distance metric [10], which captures temporal alignment and shape similarity. DTW distances are normalized to [0,1], where 0 indicates identical traces (a value of 0.1 indicates roughly 10% difference). Second, *application performance degradation* compares the impact on colocated applications (throughput drop or latency increase) when running alongside synthetic *vs.* real workloads. We report the %-point difference in degradation.

7.1 Reproducing Noisy Neighbor Effects

To serve as useful test environments, MIMESYS-generated workloads must induce similar performance impacts on colocated applications as real workloads. We measure application performance (throughput or latency) when colocated with MIMESYS-generated workloads *vs.* real workloads. To clearly

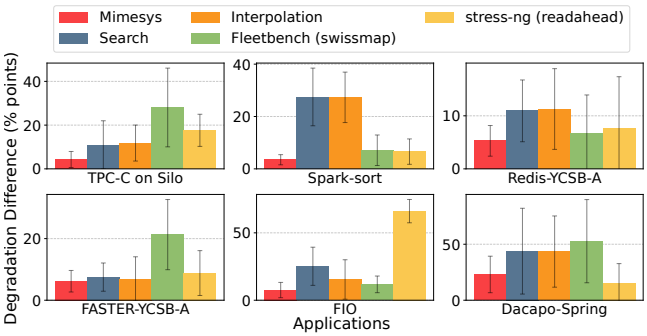


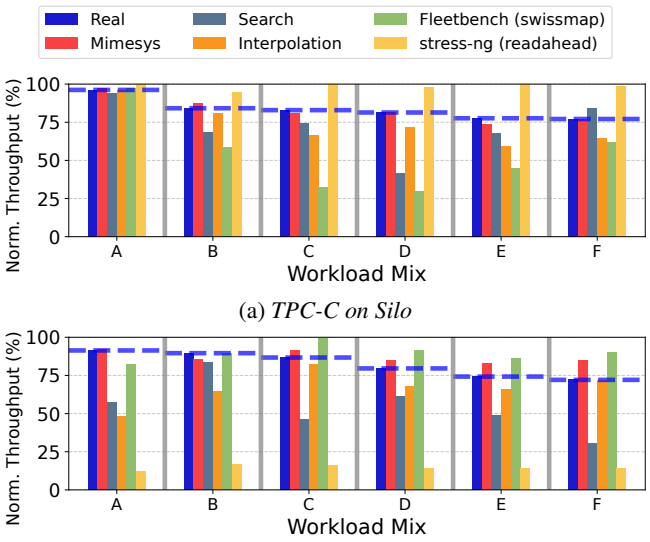
Figure 6: *Differences in application performance degradation when colocated with synthetic workloads generated by* MIMESYS *or baselines.*

expose contention effects, we configure colocated workloads to use more than 50% of available CPU cores, ensuring measurable performance degradation on the target application.

Overall results. We evaluate 6 target applications spanning databases (TPC-C on Silo), key-value stores (YCSB on FASTER and Redis), file I/O (FIO), big data (Spark Sort), and web serving (DaCapo). Figure 6 shows the difference in performance degradation when colocating these applications with MIMESYS-generated *vs.* real workloads.

MIMESYS achieves an average degradation difference of 8.3% points across all applications, compared to 19.4% points for the next best baseline (Interpolation). This shows that MIMESYS captures the resource contention dynamics more accurately than baselines. The baselines perform inconsistently across applications. For example, the Fleetbench baseline, one of single stressor baselines, achieves 11.8% points for FIO but 28.1% points for TPC-C, as it focuses on cache and memory bandwidth stress without capturing cross-resource interactions. For certain applications like DaCapo, MIMESYS exhibits higher degradation differences (15% points), where we use p90 latency as the performance metric. Capturing tail latency effects is more challenging due to their sensitivity to request-level dynamics and data access patterns. For example, for Redis, the throughput degradation predicted by MIMESYS differs by only 5.3% points, whereas the corresponding p99 latency degradation can vary by as much as 9×. This effect arises because, for some cases, small differences in request-level behavior can amplify into large tail-latency changes, making tail latency hard to reproduce with aggregate stressors.

Figure 7 shows a detailed view for TPC-C and FIO. We plot normalized throughput compared to isolated runs, varying the colocated workloads from real workload mixes (blue) and synthetic workloads generated by MIMESYS (red) or the baselines. Different workload mixes (labeled on the x-axis) induce varying contention levels, causing throughput drops from 3% to 23% (blue horizontal lines). MIMESYS-generated workloads closely track the degradation caused by real workloads across all mixes, while baselines show larger



(a) *TPC-C on Silo*

(b) *FIO benchmark*

Figure 7: *Application performance degradation while varying colocated workload mixes for (a) TPC-C on Silo and (b) FIO.*

deviations. In TPC-C (Figure 7a), Mix A (two web servers and one ResNet50 inference) causes a minimal throughput drop of 3%, which most synthetic workloads capture well. However, for mixes B to F with KV and Spark workloads that induce higher contention, MIMESYS maintains close alignment with real workload degradation, while baselines do not consistently track the patterns. For FIO (Figure 7b), running sequential read/write operations alongside colocated workloads causes throughput drops from 5% to 25%. MIMESYS reproduces these drops across all workload mixes, while baselines show deviations in several mixes (*e.g.,* Mix A and D).

Case study: TPC-C on Silo. The ability to reproduce time-varying performance impacts is crucial for evaluating applications under realistic contention. We examine a case where TPC-C on Silo is colocated with two applications (Spark running SVD and a Renaissance web server), each running in separate VMs. Figure 8 shows TPC-C throughput and resource usage over time when colocated with real workloads (blue) *vs.* MIMESYS-generated workloads (red). For the application throughput, we report the difference from the isolated run (no colocation) in percentage to highlight performance degradation due to contention.

As shown in the first plot, when colocated with a real workload mix, TPC-C throughput drops up to 37% after 30 seconds. The resource usage plots below reveal the contention sources: spikes in memory bandwidth, LLC traffic, and CPU utilization (we show per-core utilization for CPU 0 of the web server and CPU 8 of Spark for brevity). Y-axis ranges are normalized by the maximum values from training data.

MIMESYS-generated workloads reproduce this behavior with 4% average throughput deviation and 8% DTW distance in resource usage patterns. Both the timing and magnitude of performance degradation closely match the real

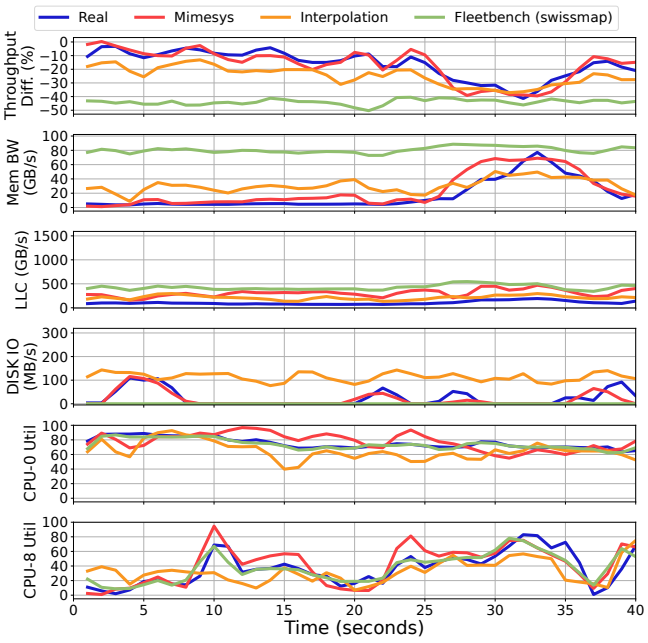


Figure 8: *TPC-C throughput and resource-usage over time when colocated with real (Spark & Renaissance web server workload mix) vs.* MIMESYS*-generated workloads.*

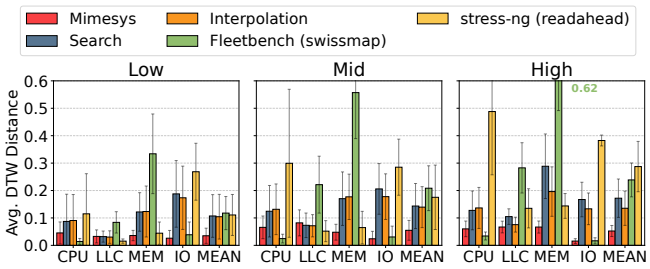


Figure 9: *DTW distances between real and synthetic traces for different metrics and workload mixes. "MEAN" reports the average DTW error over all metrics.*

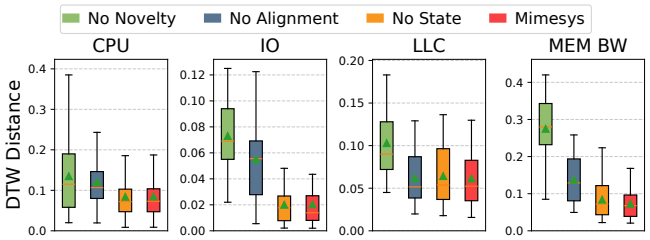


Figure 10: *Effectiveness of the proposed techniques.*

workload scenario. The alignment between memory bandwidth spikes and throughput drops suggests that memory contention from Spark is the primary cause of TPC-C degradation, which MIMESYS successfully captures. While other baselines closely match CPU utilization, they fail to match other resource usage patterns, leading to larger throughput deviations (9% for interpolation and 30% for stressor baselines).

7.2 Trace Similarities

We evaluate how closely MIMESYS-generated workloads' resource usage traces match those of real workloads. We report similarity using DTW distance, which accounts for temporal misalignment and variations in time-series traces. We classify workloads into three load levels based on CPU core allocation: low (0-30%), mid (30-60%), and high (60% and above).

Figure 9 illustrates the overall similarity between the generated and real traces for different metrics (CPU Utilization per core, memory bandwidth, LLC traffic, and disk I/O) and workload mixes (low, mid, and high load). For simplicity, we report the average CPU utilization across all cores instead of reporting per-core utilization separately. The results show that MIMESYS achieves low DTW distances across all metrics, indicating that the generated traces closely match the real ones. In terms of overall performance, MIMESYS outperforms all baseline methods, achieving an average DTW distance improvement up to 5.5× compared to our baselines.

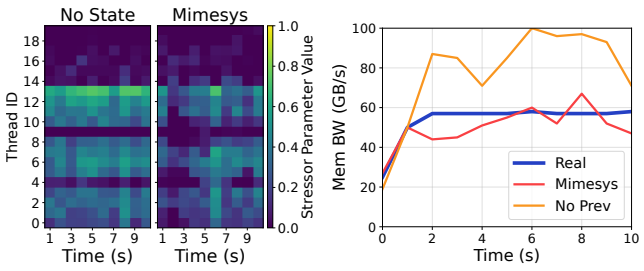
The gaps grow with load level, demonstrating MIMESYS's ability to emulate complex resource patterns. Heuristic baselines (search-based and linear interpolation) fail to capture multi-resource interactions under contention, resulting in high usage deviations. While the memory stressor baseline (Fleetbench) matches CPU utilization via sleep intervals, it fails to capture other resources, yielding higher DTW distances in memory bandwidth and LLC traffic. The IO stressor baseline (stress-ng) performs poorly across all metrics due to its sole focus on disk I/O.

7.3 Effectiveness of MIMESYS Techniques

In this section, we study how MIMESYS's key techniques contribute to its effectiveness. We ablate novelty-guided data collection, state-aware conditioning, and execution-driven alignment to measure their individual impacts on trace similarity between MIMESYS-generated workloads and real ones.

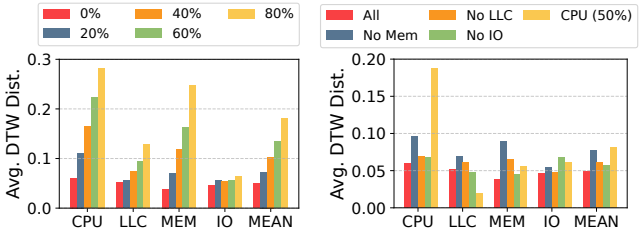
Novelty-guided training data collection. We evaluate the impact of novelty-guided data collection by comparing MIMESYS trained with and without this technique. Without novelty guidance, MIMESYS collects the training data through random sampling of stressor compositions. As shown by the green bars in Figure 10 ("No Novelty"), random sampling yields substantially higher DTW distances. Novelty-guided data collection expands coverage of the resource-usage space (Figure 4), reducing average DTW error by 2.5×.

State-aware conditioning. We evaluate state-aware conditioning by comparing MIMESYS with a variant that does not condition on prior stressor compositions. The yellow bars in Figure 10 ("No State") show DTW distances without state-aware conditioning. While the average difference is small (7% higher), certain cases show larger deviations. Relative to



(a) *Memory stressor intensity* (b) *Memory bandwidth usage*

Figure 11: *Effectiveness of state-aware conditioning.*



(a) *Varying trace dropping rate* (b) *Varying dropped metric*

Figure 12: *Sensitivity to input trace variations.*

the same stressor composition profiled after an idle window, the 90th-percentile change in LLC and memory bandwidth is 135.7% and 95.0%, respectively. State-aware conditioning is beneficial in tail cases where the preceding stressor shifts the observed metric by more than 2×.

In particular, Figure 11 presents a case study where state-aware conditioning substantially improves fidelity. Figure 11a shows how the memory bandwidth stressor is activated across threads over time, where each heatmap cell represents the fraction of time window allocated to this stressor on a particular thread. Without state-aware conditioning (left), the model generates uniformly high activations across most threads with little temporal variation, as shown by the bright green regions. This uniform high intensity creates excessive memory contention that drives bandwidth usage above target levels in subsequent windows. In contrast, with state-aware conditioning (right), the model generates more varied activation patterns, reducing memory-stressor intensity when prior execution has already created high memory load. As shown in Figure 11b, this adaptive behavior produces memory bandwidth traces that align more closely with the target.

Execution-driven alignment. We evaluate the impact of execution-driven alignment by comparing trace similarity before and after this training stage. The blue bars in Figure 10 ("No Alignment") show DTW distances for the model without alignment. Without alignment, DTW errors are 59% higher on average, demonstrating that execution-driven alignment substantially improves fidelity to real application behaviors. The improvement is particularly pronounced for memory bandwidth and I/O metrics. However, for some metrics such as LLC, improvements are less apparent. This likely occurs because the reward function may not fully capture all dimensions of trace similarity, causing the alignment process to prioritize certain metrics over others. While adjusting metric-specific weights can help, exploring more expressive reward formulations remains important future work.

7.4 Sensitivity to Input Traces

We study MIMESYS's sensitivity to variations in input traces through two experiments.

Random trace dropping. We randomly drop portions of input traces during inference to evaluate robustness to incomplete data. Figure 12a shows DTW distances when varying the dropping rate from 0% to 80%. As the dropping rate increases, similarity degrades gradually: 1.4× higher DTW distance at 20% and up to 3.7× at 80%. Missing trace segments challenge the model's ability to infer accurate stressor compositions. Despite expected degradation, MIMESYS maintains reasonable fidelity, demonstrating robustness to incomplete traces.

Metric-specific dropping. We evaluate the impact of dropping specific metric types during inference. Among the 23 input dimensions, we drop one metric type at a time and measure the resulting DTW distances. For CPU utilization, we drop 50% of per-core measurements; for other metrics (memory bandwidth, LLC traffic, disk I/O), we drop the entire metric during inference. Figure 12b shows the results. As expected, dropping specific metrics increases DTW distances for the corresponding metrics. MIMESYS maintains reasonable fidelity across other metrics (less than 1.7× degradation), showing it can infer stressor compositions that approximate overall resource patterns despite missing metrics.

8 Other Related Work

Synthetic trace generation. Prior works have explored synthetic trace generation for system evaluation. NetShare [55], and NetDiffusion [25] generate synthetic network traffic that mimics real-world patterns using machine learning models such as GANs and diffusion models. These approaches produce network packet traces (*e.g.,* packet sequences, flow statistics) as data artifacts for purposes such as privacy-preserving data sharing or network simulation. In contrast, MIMESYS generates executable programs that produce resource usage traces across multiple dimensions (*e.g.,* CPU, memory bandwidth, cache), enabling active experimentation under realistic contention conditions.

Synthetic workload generation. Other works have explored generating synthetic workloads to replicate the behavior of real applications without exposing proprietary code or data. Datamime [27] and Pbench [58] generate synthetic workloads for specific domains by using Bayesian optimization to match application performance characteristics or by combining database queries based on workload statistics. These

approaches focus on reproducing coarse-grained performance metrics for specific application types. MIMESYS differs by generating general-purpose workloads from system-level traces that reproduce fine-grained, multi-resource usage patterns over time across diverse application types.

9 Discussion

Portability across hardware platforms. MIMESYS currently targets a specific hardware platform, as the stressor library and generative model are trained for a known environment. Synthetic workloads generated for one platform may not reproduce the same resource usage patterns on systems with different hardware characteristics. For example, when synthetic workloads generated on a Haswell system are executed on a Skylake platform, we observe 190% higher prediction errors than using synthetic workloads generated by a model trained and deployed on Skylake.

The degradation is primarily caused by architectural differences that change how stressors consume system resources. While the stressors are designed for specific working-set sizes, variations in their per-iteration execution characteristics across architectures result in different LLC utilization and memory bandwidth behaviors. As future work, the generative model could be conditioned on hardware descriptors (*e.g.,* CPU architecture, cache hierarchy, memory topology) to automatically adapt stressor compositions and improve cross-platform generalization.

Other use cases. MIMESYS could be extended to enable operators to evaluate resource management policies under realistic conditions. For example, operators could assess policy changes (*e.g.,* new scheduling algorithms and NUMA balancing strategies). This would require extending MIMESYS to emulate fine-grained resource usage patterns, such as memory access patterns and cache behaviors, that influence how resource policies interact with workloads.

More broadly, realizing these use cases will require richer telemetry than the CPU, LLC, memory bandwidth, and I/O metrics considered in this work. Access to additional signals, such as memory-access locality, NUMA traffic, network activity, and accelerator utilization, together with a richer library of stressors and microbenchmarks that exercise these resources, could enable more faithful workload emulation. Such telemetry and workload primitives would allow MIMESYS to capture resource behaviors beyond those exposed in this work, supporting a broader range of tasks including resource management, and performance debugging.

10 Conclusions

We presented MIMESYS, a system that generates executable workloads from resource usage traces to enable realistic testing environments. MIMESYS represents workloads as stressor compositions and employs a diffusion model to learn the complex mapping from traces to stressor compositions. Two innovations enable this: state-aware conditioning, which captures temporal dependencies by conditioning on a prior stressor composition that affects system state; and execution-driven alignment, which adapts the model to real application behaviors using direct execution feedback. Our evaluation shows that MIMESYS achieves up to 5.5× higher trace similarity than baselines and reproduces application performance under contention 2.6× more accurately. By enabling reproducible testing without proprietary access, MIMESYS opens new possibilities for systems research and practical development.

11 Acknowledgment

We thank our shepherd for their guidance throughout the revision process and the anonymous reviewers for their constructive feedback and insightful suggestions. Their comments helped improve the clarity, quality, and presentation of this work. This material is based upon work supported by the U.S. National Science Foundation (NSF) under Grant Number 2326576.

References

[1] CAIDA. https://catalog.caida.org. Accessed: 2025-12-08.

[2] HiBench Suite. https://github.com/Intel-bigdata/HiBench. Accessed: 2025-12-08.

[3] Intel® Memory Latency Checker (intel® MLC). https://www.intel.com/content/www/us/en/download/736633/intel-memory-latency-checker-intel-mlc.html. Accessed: 2025-12-08.

[4] Memcached, a distributed memory object caching system. http://memcached.org. Accessed: 2025-12-08.

[5] Redis, an in-memory data structure store. http://redis.io. Accessed: 2025-12-08.

[6] TPC benchmark C (TPC-C). http://www.tpc.org/tpcc. Accessed: 2025-12-08.

[7] TPC benchmark H (TPC-H). http://www.tpc.org/tpch. Accessed: 2025-12-08.

[8] Andreas Abel, Yuying Li, Richard O'Grady, Chris Kennelly, and Darryl Gove. A profiling-based benchmark suite for warehouse-scale computers. In *2024 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 325–327, 2024.

[9] Scott Beamer, Krste Asanović, and David Patterson. The gap benchmark suite. *arXiv preprint arXiv:1508.03619*, 2015.

[10] Donald J. Berndt and James Clifford. Using dynamic time warping to find patterns in time series. In *Proceedings of the 3rd International Conference on Knowledge Discovery and Data Mining*, AAAIWS'94, page 359–370. AAAI Press, 1994.

[11] Kevin Black, Michael Janner, Yilun Du, Ilya Kostrikov, and Sergey Levine. Training diffusion models with reinforcement learning. *arXiv preprint arXiv:2305.13301*, 2023.

[12] Stephen M Blackburn, Zixian Cai, Rui Chen, Xi Yang, John Zhang, and John Zigman. Rethinking Java performance analysis. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1, ASPLOS 2025, Rotterdam, Netherlands, 30 March 2025 - 3 April 2025*. ACM, 2025.

[13] Leo Breiman. Random forests. *Mach. Learn.*, 45(1):5–32, October 2001.

[14] Badrish Chandramouli, Guna Prasaad, Donald Kossmann, Justin Levandoski, James Hunter, and Mike Barnett. Faster: A concurrent key-value store with in-place updates. In *Proceedings of the 2018 International Conference on Management of Data*, SIGMOD '18, page 275–290, New York, NY, USA, 2018. Association for Computing Machinery.

[15] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with ycsb. In *Proceedings of the 1st ACM Symposium on Cloud Computing*, SoCC '10, page 143–154, New York, NY, USA, 2010. Association for Computing Machinery.

[16] Eli Cortez, Anand Bonde, Alexandre Muzio, Mark Russinovich, Marcus Fontoura, and Ricardo Bianchini. Resource central: Understanding and predicting workloads for improved resource management in large cloud platforms. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, page 153–167, New York, NY, USA, 2017. Association for Computing Machinery.

[17] Dmitry Duplyakin, Robert Ricci, Aleksander Maricq, Gary Wong, Jonathon Duerig, Eric Eide, Leigh Stoller, Mike Hibler, David Johnson, Kirk Webb, Aditya Akella, Kuangching Wang, Glenn Ricart, Larry Landweber, Chip Elliott, Michael Zink, Emmanuel Cecchet, Snigdhaswin Kar, and Prabodh Mishra. The design and operation of CloudLab. In *Proceedings of the USENIX Annual Technical Conference (ATC)*, pages 1–14, July 2019.

[18] Dmitry Duplyakin, Alexandru Uta, Aleksander Maricq, and Robert Ricci. In datacenter performance, the only constant is change. In *2020 20th IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing (CCGRID)*, pages 370–379. IEEE, 2020.

[19] Johannes Freischuetz, Konstantinos Kanellis, Brian Kroth, and Shivaram Venkataraman. Tuna: Tuning unstable and noisy cloud applications. In *Proceedings of the Twentieth European Conference on Computer Systems*, EuroSys '25, page 954–973, New York, NY, USA, 2025. Association for Computing Machinery.

[20] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

[21] Sören Henning, Adriano Vogel, Esteban Perez-Wohlfeil, Otmar Ertl, and Rick Rabiser. When should i run my application benchmark? studying cloud performance variability for the case of stream processing applications. In *Proceedings of the 33rd ACM International Conference on the Foundations of Software Engineering*, FSE Companion '25, page 400–410, New York, NY, USA, 2025. Association for Computing Machinery.

[22] Jonathan Ho, Ajay Jain, and Pieter Abbeel. Denoising diffusion probabilistic models. *Advances in neural information processing systems*, 33:6840–6851, 2020.

[23] Jonathan Ho and Tim Salimans. Classifier-free diffusion guidance. *arXiv preprint arXiv:2207.12598*, 2022.

[24] Zichao Hu, Chen Tang, Michael Joseph Munje, Yifeng Zhu, Alex Liu, Shuijing Liu, Garrett Warnell, Peter Stone, and Joydeep Biswas. Composablenav: Instruction-following navigation in dynamic environments via composable diffusion. In *9th Annual Conference on Robot Learning*, 2025.

[25] Xi Jiang, Shinan Liu, Aaron Gember-Jacobson, Arjun Nitin Bhagoji, Paul Schmitt, Francesco Bronzino, and Nick Feamster. Netdiffusion: Network data augmentation through protocol-constrained traffic generation. *Proc. ACM Meas. Anal. Comput. Syst.*, 8(1), February 2024.

[26] Colin Ian King. stress-ng. https://github.com/ColinIanKing/stress-ng. Accessed: 2025-12-08.

[27] Hyun Ryong Lee and Daniel Sanchez. Datamime: Generating representative benchmarks by automatically synthesizing datasets. In *Proceedings of the 55th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '22, page 1144–1159. IEEE Press, 2023.

[28] Mingyu Liang, Yu Gan, Yueying Li, Carlos Torres, Abhishek Dhanotia, Mahesh Ketkar, and Christina Delimitrou. Ditto: End-to-end application cloning for networked cloud services. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, ASPLOS 2023, page 222–236, New York, NY, USA, 2023. Association for Computing Machinery.

[29] Shutian Luo, Huanle Xu, Chengzhi Lu, Kejiang Ye, Guoyao Xu, Liping Zhang, Yu Ding, Jian He, and Chengzhong Xu. Characterizing microservice dependency and performance: Alibaba trace analysis. In *Proceedings of the ACM Symposium on Cloud Computing*, pages 412–426, 2021.

[30] Volodymyr Mnih, Adria Puigdomenech Badia, Mehdi Mirza, Alex Graves, Timothy Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. Asynchronous methods for deep reinforcement learning. In Maria Florina Balcan and Kilian Q. Weinberger, editors, *Proceedings of The 33rd International Conference on Machine Learning*, volume 48 of *Proceedings of Machine Learning Research*, pages 1928–1937, New York, New York, USA, 20–22 Jun 2016. PMLR.

[31] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Köpf, Edward Yang, Zach DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. *PyTorch: an imperative style, high-performance deep learning library*. Curran Associates Inc., Red Hook, NY, USA, 2019.

[32] Ethan Perez, Florian Strub, Harm de Vries, Vincent Dumoulin, and Aaron Courville. Film: visual reasoning with a general conditioning layer. In *Proceedings of the Thirty-Second AAAI Conference on Artificial Intelligence and Thirtieth Innovative Applications of Artificial Intelligence Conference and Eighth AAAI Symposium on Educational Advances in Artificial Intelligence*, AAAI'18/IAAI'18/EAAI'18. AAAI Press, 2018.

[33] PostgreSQL Database Management System. https://www.postgresql.org/. Accessed: 2025-12-08.

[34] Aleksandar Prokopec, Andrea Rosà, David Leopoldseder, Gilles Duboscq, Petr Tůma, Martin Studener, Lubomír Bulej, Yudi Zheng, Alex Villazón, Doug Simon, Thomas Würthinger, and Walter Binder. Renaissance: benchmarking suite for parallel applications on the jvm. In *Proceedings of the 40th ACM*

SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2019, page 31–47, New York, NY, USA, 2019. Association for Computing Machinery.

[35] Haoran Qiu, Subho S. Banerjee, Saurabh Jha, Zbigniew T. Kalbarczyk, and Ravishankar K. Iyer. FIRM: An intelligent fine-grained resource management framework for SLO-Oriented microservices. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 805–825. USENIX Association, November 2020.

[36] Alec Radford, Jong Wook Kim, Chris Hallacy, Aditya Ramesh, Gabriel Goh, Sandhini Agarwal, Girish Sastry, Amanda Askell, Pamela Mishkin, Jack Clark, et al. Learning transferable visual models from natural language supervision. In *International conference on machine learning*, pages 8748–8763. PmLR, 2021.

[37] Rafael Rafailov, Archit Sharma, Eric Mitchell, Christopher D Manning, Stefano Ermon, and Chelsea Finn. Direct preference optimization: Your language model is secretly a reward model. *Advances in neural information processing systems*, 36:53728–53741, 2023.

[38] Vijay Janapa Reddi, Christine Cheng, David Kanter, Peter Mattson, Guenther Schmuelling, Carole-Jean Wu, Brian Anderson, Maximilien Breughe, Mark Charlebois, William Chou, Ramesh Chukka, Cody Coleman, Sam Davis, Pan Deng, Greg Diamos, Jared Duke, Dave Fick, J. Scott Gardner, Itay Hubara, Sachin Idgunji, Thomas B. Jablin, Jeff Jiao, Tom St. John, Pankaj Kanwar, David Lee, Jeffery Liao, Anton Lokhmotov, Francisco Massa, Peng Meng, Paulius Micikevicius, Colin Osborne, Gennady Pekhimenko, Arun Tejusve Raghunath Rajan, Dilip Sequeira, Ashish Sirasao, Fei Sun, Hanlin Tang, Michael Thomson, Frank Wei, Ephrem Wu, Lingjie Xu, Koichi Yamada, Bing Yu, George Yuan, Aaron Zhong, Peizhao Zhang, and Yuchen Zhou. Mlperf inference benchmark. In *Proceedings of the ACM/IEEE 47th Annual International Symposium on Computer Architecture*, ISCA '20, page 446–459. IEEE Press, 2020.

[39] Benjamin Reidys, Pantea Zardoshti, Íñigo Goiri, Celine Irvene, Daniel S. Berger, Haoran Ma, Kapil Arya, Eli Cortez, Taylor Stark, Eugene Bak, Mehmet Iyigun, Stanko Novakovic, Lisa Hsu, Karel Trueba, Abhisek Pan, Chetan Bansal, Saravan Rajmohan, Jian Huang, and Ricardo Bianchini. Coach: Exploiting temporal patterns for all-resource oversubscription in cloud platforms. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1*, ASPLOS '25, page 164–181, New York, NY, USA, 2025. Association for Computing Machinery.

[40] Allen Ren, Justin Lidard, Lars Ankile, Anthony Simeonov, Pulkit Agrawal, Anirudha Majumdar, Benjamin Burchfiel, Hongkai Dai, and Max Simchowitz. Diffusion policy policy optimization. In *International Conference on Learning Representations*, volume 2025, pages 77288–77329, 2025.

[41] Robin Rombach, Andreas Blattmann, Dominik Lorenz, Patrick Esser, and Björn Ommer. High-resolution image synthesis with latent diffusion models. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pages 10684–10695, 2022.

[42] Olaf Ronneberger, Philipp Fischer, and Thomas Brox. U-net: Convolutional networks for biomedical image segmentation. In *International Conference on Medical image computing and computer-assisted intervention*, pages 234–241. Springer, 2015.

[43] Arjun Roy, Hongyi Zeng, Jasmeet Bagga, George Porter, and Alex C. Snoeren. Inside the social network's (datacenter) network. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, SIGCOMM '15, page 123–137, New York, NY, USA, 2015. Association for Computing Machinery.

[44] Divyanshu Saxena, Nihal Sharma, Donghyun Kim, Rohit Dwivedula, Jiayi Chen, Chenxi Yang, Sriram Ravula, Zichao Hu, Aditya Akella, Sebastian Angel, Joydeep Biswas, Swarat Chaudhuri, Isil Dillig, Alex Dimakis, P. Brighten Godfrey, Daehyeok Kim, Chris Rossbach, and Gang Wang. On a foundation model for operating systems, 2023.

[45] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017.

[46] Gagan Somashekar, Anurag Dutt, Mainak Adak, Tania Lorido Botran, and Anshul Gandhi. Gamma: Graph neural network-based multi-bottleneck localization for microservices applications. In *Proceedings of the ACM Web Conference 2024*, WWW '24, page 3085–3095, New York, NY, USA, 2024. Association for Computing Machinery.

[47] Standard Performance Evaluation Corporation (SPEC). *SPEC CPU®2017 Benchmark*, 2017. Accessed: 2025-12-08.

[48] Wei Su, Abhishek Dhanotia, Carlos Torres, Jayneel Gandhi, Neha Gholkar, Shobhit Kanaujia, Maxim Naumov, Kalyan Subramanian, Valentin Andrei, Yifan Yuan, and Chunqiang Tang. Dcperf: An open-source, battle-tested performance benchmark suite for datacenter workloads. In *Proceedings of the 52nd Annual International Symposium on Computer Architecture*, ISCA '25, page 1717–1730, New York, NY, USA, 2025. Association for Computing Machinery.

[49] Texas Advanced Computing Center. Texas advanced computing center (tacc). http://www.tacc.utexas.edu, 2025. Accessed: 2025-12-08.

[50] Stephen Tu, Wenting Zheng, Eddie Kohler, Barbara Liskov, and Samuel Madden. Speedy transactions in multicore in-memory databases. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, page 18–32, New York, NY, USA, 2013. Association for Computing Machinery.

[51] Alexandru Uta, Alexandru Custura, Dmitry Duplyakin, Ivo Jimenez, Jan Rellermeyer, Carlos Maltzahn, Robert Ricci, and Alexandru Iosup. Is big data performance reproducible in modern cloud networks? In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 513–527, Santa Clara, CA, February 2020. USENIX Association.

[52] Dana Van Aken, Dongsheng Yang, Sebastien Brillard, Ari Fiorino, Bohan Zhang, Christian Bilien, and Andrew Pavlo. An inquiry into machine learning-based automatic configuration tuning services on real-world database management systems. *Proc. VLDB Endow.*, 14(7):1241–1253, March 2021.

[53] Jaylen Wang, Daniel S. Berger, Fiodar Kazhamiaka, Celine Irvene, Chaojie Zhang, Esha Choukse, Kali Frost, Rodrigo Fonseca, Brijesh Warrier, Chetan Bansal, Jonathan Stern, Ricardo Bianchini, and Akshitha Sriraman. Designing cloud servers for lower carbon. In *2024 ACM/IEEE 51st Annual International Symposium on Computer Architecture (ISCA)*, pages 452–470, 2024.

[54] John Wilkes. Yet more Google compute cluster trace data. Google research blog, April 2020. Posted at https://ai.googleblog.com/2020/04/yet-more-google-compute-cluster-trace.html.

[55] Yucheng Yin, Zinan Lin, Minhao Jin, Giulia Fanti, and Vyas Sekar. Practical gan-based synthetic ip header trace generation using netshare. In *Proceedings of the ACM SIGCOMM 2022 Conference*, SIGCOMM '22, page 458–472, New York, NY, USA, 2022. Association for Computing Machinery.

[56] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J. Franklin, Scott Shenker, and Ion Stoica. Resilient distributed datasets: a fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, NSDI'12, page 2, USA, 2012. USENIX Association.

[57] Yuhong Zhong, Daniel S. Berger, Carl Waldspurger, Ryan Wee, Ishwar Agarwal, Rajat Agarwal, Frank Hady, Karthik Kumar, Mark D. Hill, Mosharaf Chowdhury, and Asaf Cidon. Managing memory tiers with CXL in virtualized environments. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, pages 37–56, Santa Clara, CA, July 2024. USENIX Association.

[58] Yan Zhou, Chunwei Liu, Bhuvan Urgaonkar, Zhengle Wang, Magnus Mueller, Chao Zhang, Songyue Zhang, Pascal Pfeil, Dominik Horn, Zhengchun Liu, Davide Pagano, Tim Kraska, Samuel Madden, and Ju Fan. Pbench: Workload synthesizer with real statistics for cloud analytics benchmarking. *Proc. VLDB Endow.*, 18(11):3883–3895, July 2025.

A List of Stressors Used in MIMESYS

MIMESYS uses 14 stressors spanning CPU, memory, cache, and disk I/O resources, selected to ensure diverse resource usage patterns. We select these stressors by profiling approximately 600 candidates from stress-ng [26] and Fleetbench [8], choosing those with distinct resource usage characteristics. Table 2 lists the selected stressors along with their resource types: Computation (Comp), Memory (Mem), Cache, I/O (IO), and No-op (Sleep). The library column indicates the source of each stressor (either stress-ng or a specific Fleetbench benchmark), while the stressor name specifies the particular function used. For brevity, we omit detailed configuration parameters such as boolean flags or other default settings.

Library	Stressor Name	Type
Compression	ZSTD_decompress_Fleet	Comp
Hashing	Extendcrc32cinternal_Fleet_cold	Mem
Hashing	Computecrc32c_Fleet_cold	Mem
Libc	Memcpy_Fleet_L1	Cache
Simd	SerialDistanceComputation:256	Comp
Simd	SerialDistanceComputation:512	Comp
Swissmap	InsertMiss/density:0	Mem
Swissmap	InsertManyOrdered/density:0	Mem
Swissmap	InsertManyOrdered/density:1	Mem
Swissmap	InsertManyUnordered/density:1	Mem
Stress_ng	Readahead	IO
Stress_ng	Fallocate	IO
Stress_ng	Sendfile	IO
Sleep	-	No-op

Table 2: *Stressors used in* MIMESYS *with their library sources and resource types.*

B Hyperparameters Used for Execution-Driven Alignment

We use denoising diffusion policy optimization (DDPO) [11] to fine-tune the diffusion model during execution-driven alignment (§4.3). DDPO is a reinforcement learning algorithm that adapts the pretrained diffusion model to maximize reward signals based on actual execution feedback. The training process involves several key hyperparameters that affect both training stability and the quality of the aligned model.

We set the learning rate to 3×10^{-6}, which is lower than the pretraining learning rate to ensure stable fine-tuning without destroying the knowledge acquired during pretraining. To normalize rewards and reduce variance during training, we estimate advantages using a running mean baseline. The running mean is computed over a buffer of 512 samples, providing a stable estimate of expected rewards. For training stability, we clip the advantage ratio to 3, which limits the magnitude of policy updates and prevents large, destabilizing changes to the model during alignment.

C Artifact Appendix

C.1 Abstract

This artifact provides the implementation of MIMESYS, including the pipeline for training data collection, diffusion-model training, and execution-driven alignment. It also includes the scripts used to run and profile the workload mixes in the evaluation. The artifact enables users to generate synthetic workloads from resource usage traces and evaluate their fidelity using the profiling methodology described in the paper.

C.2 Scope

The artifact is intended to support the reproducibility of the key results in the paper. Specifically, it allows users to:

- Collect training data from stressor executions.
- Train the diffusion model used by MIMESYS.
- Perform execution-driven alignment.
- Run the evaluation pipeline in the paper.

C.3 Contents

The artifact includes the source code for all major components of MIMESYS, including training data collection, diffusion-model training, and execution-driven alignment. In addition, it provides the scripts, configurations, and profiling infrastructure used to generate, execute, and evaluate the workload mixes presented in the paper.

C.4 Hosting

The artifact is available at `https://github.com/ldos-project/mimesys`. The `artifact-eval` branch contains the version prepared for artifact evaluation.

C.5 Requirements

The artifact is designed to run on Linux systems capable of executing stressors (*e.g.,* Fleetbench and stress-ng) and collecting resource-utilization profiles using TACC's HPC monitoring infrastructure. Training requires NVIDIA GPUs with CUDA 12.x support. Detailed hardware and software dependencies, installation instructions, and environment specifications are provided in the artifact repository.

ARTIFACT EVALUATED
USENIX
AVAILABLE

MERLIN: An Efficient Adaptive Cache Eviction Algorithm via Fine-Grained Characterization

Liujia Li[†], Jinhao Guo[†], Yi Fan[†], Jianyu Wu[†], Zhenlin Wang[§],
Jie Zhang[†], Yuval Tamir[‡], Xiaolin Wang[†], Yingwei Luo[†], Diyu Zhou[†]
[†]*Peking University* [§]*Michigan Tech* [‡]*UCLA*

Abstract

The diverse and complex modern workloads pose a major challenge for cache to remain effective across all scenarios, degrading the performance of critical systems such as web caches. Adaptive cache eviction algorithms promise to address this challenge by observing access patterns and adjusting their behavior accordingly. However, existing ones fail this promise, even underperforming static policies. Our analysis shows that this is because they adapt only to a few typical patterns, incurring poor performance on others. Moreover, they adapt by switching between supposedly complementary algorithms, which turn out to interfere with each other.

We present MERLIN, an efficient adaptive algorithm that robustly handles diverse access patterns while maintaining low overhead and high multicore scalability. The efficiency of MERLIN stems from a principled pattern characterization method that can express a wide spectrum of access patterns rather than a few typical ones. This is achieved by characterizing at the level of individual objects while accounting for both access locality and cache size. Furthermore, MERLIN cleanly decouples responsibilities among its components, with each component performing a single task, thereby eliminating the costly interference between base algorithms. Our evaluation across 11 datasets with 5423 traces shows that MERLIN achieves robust improvements in hit rate over existing algorithms, increasing throughput by 1.4× to 7.8×.

1 Introduction

To improve performance, modern computing systems widely employ software caches, such as in-memory key-value stores in data centers [3,8,9,13,30,58,59,68], edge caches in content delivery networks [3,7,12,16,33,43,52,61,68], and page caches in operating systems [14,15,27,35,42,46,48,78]. The efficiency of such caching systems largely depends on their eviction algorithms, which manage limited cache space to exploit access locality in the workload. The effectiveness of eviction algorithms, in turn, hinges on how well their underlying assumptions match the workload access patterns.

However, prior studies [12, 35, 45, 48, 67, 69, 70, 77] show that modern workloads exhibit diverse and varying access patterns. We confirm the above conclusion through an analysis of 11 real-world datasets (§3.1). To express workload diversity, prior work [48] classifies access patterns into four primitive types: 1) `LFU-friendly` (high-frequency access), 2) `LRU-friendly` (high-recency access), 3) `Churn` (objects accessed periodically), and 4) `Scan` (objects accessed only once). The workload's access pattern is often a varied combination of these primitives, while also shifting rapidly over time.

The complex access patterns are a key challenge to eviction algorithms, as they must remain effective across all patterns. Nevertheless, decades of advancements in the field have mainly focused on static algorithms [8, 13, 17, 27, 35, 40, 71, 72, 78], which use fixed policies to manage cache regardless of access patterns. Despite many elegant proposals, the inherent rigidity of static algorithms limits their ability to deliver the best performance across diverse workloads, as our evaluation shows in §3.

To tame workload diversity, a general system design principle is to introduce *adaptivity*, which has seen great success across multiple domains, such as scheduling, resource allocation [23,24,75] and fuzzing [26,29,66]. In the context of cache eviction, an adaptive algorithm involves two key aspects: 1) *Pattern characterization*, *i.e.*, observing and characterizing workload access patterns; and 2) *Policy adjustment*, dynamically changing its policy to suit the observed patterns. Such adaptivity promises an effective way to treat the diverse access patterns in modern workloads.

However, existing adaptive algorithms [6, 45, 48, 55] fail to fulfill this promise, to a degree that they often underperform their static counterparts across real-world workloads, as we measure in §3.2. Indeed, designing effective adaptive algorithms is challenging. Prior work [71] shows that a simple yet effective static algorithm already performs well on common workloads, while making itself adaptive only offers marginal gains in the rare adversarial workloads. This suggests that the complexity cost of adaptivity may outweigh its benefits. Our work revisits this problem by studying how to design an effective adaptive algorithm with low complexity.

This paper presents MERLIN[1], a cache eviction algorithm that adapts to a broad spectrum of access patterns with low management overhead and high multicore scalability. To design MERLIN, we begin with an in-depth analysis of the typical patterns where existing adaptive algorithms fail (§3.3).

Through the analysis, our key observation is that existing adaptive algorithms are designed to be effective for a few typical patterns, thereby incurring poor performance in many

[1]The Arthurian wizard, who shape-shifts to navigate different scenarios.

other cases. Algorithms like ARC [45] and CAR [6] adapt cache space for accesses dominated by 1) objects accessed only once recently, or 2) objects accessed multiple times recently. As both categories represent short-term accesses, they do not effectively capture long-term access patterns. Cacheus [48] adapts by switching between two complementary base algorithms, each effective for two of the four aforementioned primitive patterns. Thus, Cacheus fails to adapt when accesses deviate from these typical patterns.

To overcome the above limitations, Merlin advances in both key aspects of adaptivity. In *pattern characterization*, state-of-the-art mechanisms classify access sequences into only a few typical patterns, failing to extract the characteristics of many access traces. This is due to a failure to meet the challenge of devising an approach for accurately expressing the full spectrum of access patterns.

To address the above challenge, we identify three key requirements. The characterization must 1) be flexible, *i.e.*, not limited to a small fixed set of access patterns, 2) reflect the degree of locality, the key to caching effectiveness, and 3) account for cache size, since the same access sequence can behave differently under different cache sizes. Prior characterization approaches do not meet all these requirements, explaining their ineffectiveness across diverse patterns.

Merlin's principled characterization approach meets all the above requirements. Merlin characterizes by classifying objects based on two dimensions: *hotness* and *popularity* into four types: hot-popular, hot-rare, cold-popular, and cold-rare. *Hotness* and *popularity* denote short-term and long-term access counts, respectively, capturing both recency and frequency aspects of locality, therefore meeting the second requirement. To meet the third requirement, Merlin sets the total number of hot and popular objects to fit the cache, with an intuition that only the hottest and most popular items are stored. With this object-level classification, Merlin accurately captures a wide range of access patterns using counts of the four object types, satisfying the first requirement.

Regarding *policy adjustment*, prior work uses complementary base algorithms or data structures that turn out to interfere with each other when neither is particularly effective, leading to poor performance (§3.3). Thanks to its principled characterization, Merlin adapts using a simple policy that avoids such interference: prioritize hot-popular objects, then hot or popular objects, and filter out cold-rare objects, as the object type counts reflect the access pattern.

Merlin's design is based on a unified and cohesive architecture that embraces recent advances in multi-queue FIFO-based algorithms [71]. It consists of multiple components with cleanly decoupled responsibilities, each performing a single task. Merlin uses 1) a small FIFO queue to filter cold-rare objects, 2) a core queue and a staging queue, two large FIFO queues to prioritize hot-popular objects, followed by hot or popular objects, 3) a ghost queue [6, 36, 45, 48, 71] to accurately identify hot-rare objects, and 4) a count-min sketch [21, 22, 27, 31] to spot cold-popular objects.

We evaluated Merlin with 16 algorithms on 5423 real-world traces from 11 datasets across various domains, including CDN, KV stores, and VM management/storage caching. Our evaluation shows that Merlin achieves robust improvements in hit rate over prior work, while incurring low management overhead and offering high multicore scalability, improving throughput by 1.4× to 7.8×.

In summary, this paper makes the following contributions:

- **Analysis.** We show that prior adaptive algorithms fail to cover the wide spectrum of access patterns, mainly because of their overly simple pattern characterization.
- **Pattern characterization.** We identify the key requirements for access characterization and contribute a principled approach that expresses diverse patterns.
- **Merlin.** We design Merlin, an adaptive cache eviction algorithm with a unified, cohesive architecture. Merlin delivers robust performance across diverse patterns, while offering low overhead and high scalability.

2 Cache Eviction Algorithms

A cache system's efficiency hinges on its eviction algorithm, since it decides which objects[2] utilize the scarce cache space. This section briefly reviews state-of-the-art cache eviction algorithms, covering both static (§2.1) and dynamic ones (§2.2).

2.1 Static Eviction Algorithms

A static algorithm always applies the same fixed set of rules for eviction, no matter the workload characteristics. Research over the past few decades has primarily focused on static algorithms. Early static algorithms, such as LFU, FIFO [10, 53], and LRU [10, 25], use simple heuristics that perform well only for a narrow set of access patterns. For example, LRU favors accesses with high recency, but performs poorly for objects repeatedly accessed in a round-robin fashion.

Follow-up work aims to be more robust across diverse access patterns by using better ranking metrics [8, 12, 13, 18, 40] and/or combining multiple components (*i.e.*, LRU lists, and FIFO queues) [6, 27, 34–36, 45, 71, 78]. Two notable examples are LIRS [35] and S3-FIFO [71], performing best on the datasets we evaluated (§7). As explained in §5, Merlin inherits the FIFO-based design from S3-FIFO, but generalizes it to: 1) be adaptive, 2) account for both recency and frequency, and 3) use a better characterization of one-hit wonders (§5.4).

2.2 Adaptive Eviction Algorithms

An adaptive eviction algorithm involves: 1) observing and characterizing workload access patterns, and 2) adjusting its policy (behavior) accordingly (by, *e.g.*, automatically tuning internal parameters) to maximize performance.

Adaptive algorithms fall into two categories: 1) size-adaptive and 2) policy-adaptive.

A size-adaptive algorithm partitions cache into multiple

[2] We use the term "object" to collectively refer to various types of data (*e.g.*, pages, blocks, key-value pairs) stored in a cache.

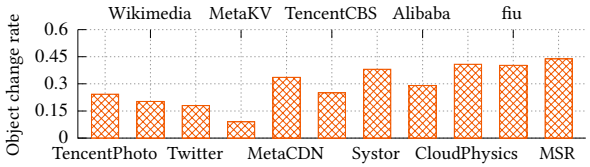


Figure 1: The object change rate of 11 real-world datasets.

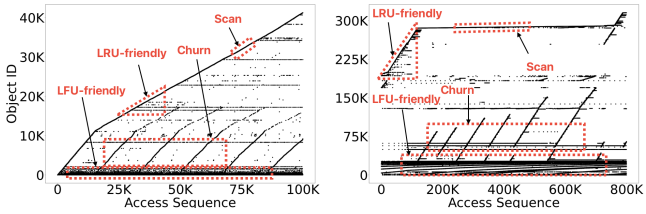


Figure 2: Two examples of real-world access patterns. The *x*-axis is the access sequence number, and the *y*-axis is the object ID. Each point means that the object is accessed at that moment.

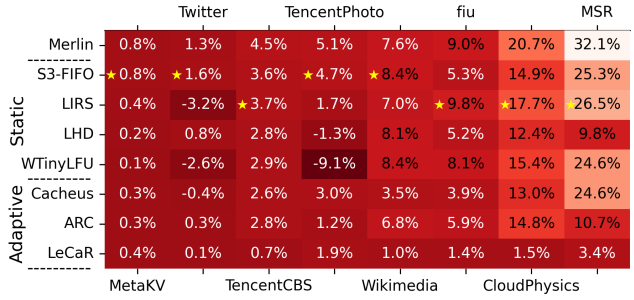


Figure 3: The average hit rate improvement over LRU for 8 representative eviction algorithms on 8 real-world datasets. The cache size is 3% of the working set size. The larger the value, the higher the hit rate, with a star showing the best algorithm (excluding Merlin). Merlin achieves the highest hit rate in 5 datasets, with close to the best performance in 3 datasets.

parts and adapts by adjusting each part's size based on access patterns. The state-of-the-art is ARC [45] and CAR [6].

ARC divides cache into two lists: a one-hit list and a multi-hit list. A new object is first placed in the one-hit list and added to the multi-hit list on a second access. On a cache miss, ARC checks whether it has previously evicted the requested object by looking for it in the eviction history of both lists. If so, ARC expands that list's size and correspondingly reduces the size of the other list. In this way, ARC adapts by dynamically adjusting the cache space partition to favor either 1) objects accessed only once recently (by expanding the one-hit list), or 2) objects accessed multiple times recently (by expanding the multi-hit list). CAR follows a similar design but notably replaces the LRU lists with FIFO queues.

A policy-adaptive algorithm adapts by switching between multiple base eviction algorithms depending on their effectiveness under observed access patterns [55].

Cacheus [48] is state-of-the-art, working by classifying access into four primitive patterns: 1) *LFU-friendly*: a small portion of objects are frequently accessed. 2) *LRU-friendly*: recently accessed objects are likely to be accessed again soon 3) *Churn*: all objects are repeatedly accessed with equal probability, and 4) *Scan*: all objects are accessed only once.

To adapt, Cacheus uses two complementary base algorithms: a *scan-resistant* LRU variant and a *churn-resistant* LFU variant, to cover all four primitive access patterns. Cacheus maintains a weight for each base algorithm. Upon a cache miss, Cacheus selects a base algorithm to perform eviction using probabilities proportional to the weights of the algorithms. Next, Cacheus checks whether it mistakenly evicted the requested object, and, if so, penalizes the responsible base algorithm by reducing its weight. Cacheus uses machine learning to control the rate at which weights change.

3 How Modern Algorithms Perform

This section analyzes the characteristics of modern cache workloads (§3.1), evaluates existing eviction algorithms on them (§3.2), and identifies remaining deficiencies (§3.3)

3.1 Characteristics of Modern Cache Workloads

It is well known that modern cache workloads 1) exhibit diverse access patterns and 2) that their access patterns change over time [48, 49, 56, 57, 70]. We confirm this by analyzing 11 real-world datasets with a total of 5423 traces (Table 2)

Figure 1 summarizes our analysis. As we have not found a prior metric to quantify the diversity and variability of access patterns, we propose a new metric[3], the object change rate, which measures the fraction of change in uniquely accessed objects between consecutive time windows. As shown in the figure, using a time window of 1% of the trace length, all datasets exhibit a high object change rate, up to ~45% (*i.e.*, nearly half of the accessed objects change in each window).

Figure 2 visualizes two examples of diverse and varying access patterns. Following the classification in Cacheus (§2.2), all four patterns appear in two traces with frequent switches.

3.2 Performance of Existing Algorithms

We evaluate 16 state-of-the-art cache eviction algorithms on 11 real-world datasets. Figure 3 shows a subset of the results (full results in §7.2), including the top-performing algorithms in both the static and adaptive categories (§2).

We draw two conclusions. First, for static algorithms, there is no "silver bullet" that consistently outperforms others. S3-FIFO achieves the highest hit rate across 4 datasets (excluding Merlin), in some cases by a significant margin over the runner-up. In the remaining 4 datasets, LIRS is the best.

We analyze a few typical traces where LIRS and S3-FIFO underperform and find that their deficiencies stem from their static nature; a fixed set of rules is a poor match for the diverse and varying access patterns. Introducing adaptiveness enables the two algorithms to perform better on such traces. For example, LIRS may benefit from adjusting its promotion

[3] While this metric is not perfect, it is still useful to quantify accesses.

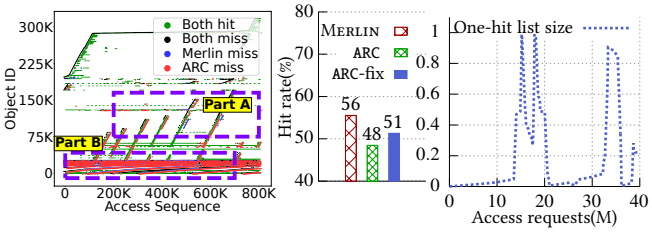


Figure 4: The Alibaba [1] 742 trace. ARC-fix only caches objects in the two-hit list. Assuming the cache is in DRAM with 100ns access time and the backend is SSD with 10μs access time, MERLIN improves performance by 8× over ARC. MERLIN excels by caching both short- and long-term locality objects.

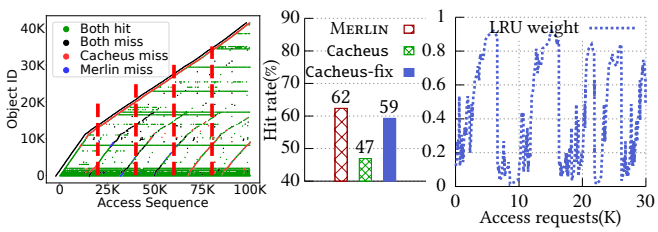


Figure 5: The Alibaba 269 trace. As shown in the left-most figure, given a time window (data between vertical red lines), the access patterns do not fall into any one of the primitive patterns; LRU weight is the activation probability of the LRU variant. Cacheus-fix only uses the LFU variant. MERLIN improves by identifying the high-frequency objects in the churn pattern.

policy or the small FIFO queue's size, and S3-FIFO could improve by adapting its reinsertion policy.

Second, adaptive algorithms consistently underperform the static ones. None of the adaptive algorithms 1) achieves the highest hit rate in any dataset; nor 2) matches the performance of the best static algorithms (*i.e.*, S3-FIFO) and arguably the second-best one (*i.e.*, LIRS) across all datasets.

This result is surprising, as adaptiveness should better match diverse and varying access patterns (§3.1), expectedly. Indeed, we view this as an anomaly in systems research, as adaptiveness succeeds widely in other domains (*e.g.*, scheduling and resource allocation [23,24,75] and fuzzing [26,29,66]), often resulting in significant performance gains over static counterparts. This provides strong motivation for MERLIN, which advances the design of adaptive eviction algorithms.

3.3 Remaining Deficiencies in Adaptive Algorithms

We analyze several representative traces to reveal three limitations of existing adaptive algorithms.

Limitation #1: Failure to adapt to real-world patterns. An inherent limitation of prior mechanisms is that they only adapt to a small subset of access patterns. This is because their design assumes that a given access sequence will always conform to a few fixed patterns. This assumption is explicitly made in Cacheus (§2.2), which uses base algorithms to cover all primitive patterns. The assumption is also made implicitly in ARC and CAR, which model access sequences as dominated only by short-term locality. They thus forgo long-term locality, as we explained next. However, the diverse access patterns in real-world workloads often deviate from these fixed patterns. Therefore, in those cases, existing adaptive algorithms fail. As detailed in §4, the root cause of this limitation is their inaccurate characterization approach.

Figure 4 shows a trace where ARC fails. This trace exhibits a mix of accesses with high long-term locality (Part A) and high short-term locality (Part B). Specifically, Part A contains objects accessed repeatedly but often with long intervals (that exceed the cache size) between two accesses. Such objects are good candidates for caching. However, with ARC, they are often first placed in the one-hit list and then evicted before the second access. Thus, regardless of how ARC partitions the cache between the two lists, it fails to cache these objects, leading to poor performance.

Figure 5 shows a trace where Cacheus fails. In this trace, for most of the time, the access sequences do not fall into any one of the primitive patterns (§2.2); rather, all four of them exist simultaneously. This renders both base algorithms ineffective, and thus, no matter how Cacheus adjusts their weights, it cannot deliver good performance. We further analyze this trace in §5.4.

Limitation #2: Adversarial base algorithms/data structures. Related to the first limitation, another key limitation lies in the adaptiveness approach: if the access patterns do not fall within the assumed ones, due to the nature of their policy- or space-adaptive design, the supposedly complementary base algorithms or data structures end up working against each other, further reducing performance.

For space-adaptive algorithms, like ARC, with a mixed access pattern, accesses to objects in both partitions suffer excessive cache misses. In this case, we find that the algorithm blindly adjusts partition sizes (the right most in Figure 4), leading to thrashing that actually evicts useful objects. Similarly, in such a case, policy-adaptive algorithms like Cacheus, blindly adjust weights (the right most in Figure 5), causing one algorithm to evict useful objects cached by the other. Indeed, as shown in Figures 4 and 5, for these two traces, using a single base data structure (ARC-fix) or algorithm (Cacheus-fix) actually outperforms the full algorithm.

Limitation #3: Poor robustness across cache sizes. Existing adaptive algorithms fail to deliver robust performance across different cache sizes, as shown in Figure 11. Their design does not consider the impact of cache sizes on access patterns (§2.2). This is less inherent, as one can likely enhance these algorithms to overcome it. Nonetheless, MERLIN also overcomes this limitation.

4 Characterizing Workload Access Patterns

We next show how MERLIN addresses the inherent limitations of prior adaptive algorithms (§3.3) through its characterization approach. Later sections present MERLIN's design (§5) and implementation (§6), both enabled by this new way of

	Fine-grained	Flexibility	Short-/long-term Locality	Cache-size Awareness
Cacheus	✗	✗	✓	✗
ARC	✓	✓	✗	✗
Merlin	✓	✓	✓	✓

Table 1: A summary of the characterization approach in state-of-the-art adaptive algorithms. ✗: not considered; ✓: considered.

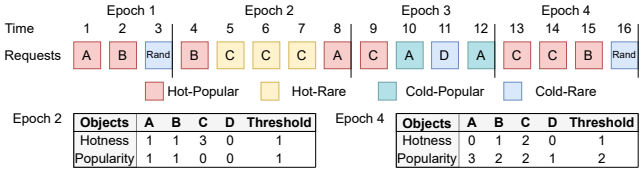


Figure 6: An example of how Merlin characterizes access patterns. Rand denotes an access to an irrelevant object.

characterizing access patterns.

Challenge. Our key observation is that the inherent limitation of prior work (*i.e.*, the inability to adapt to diverse access patterns) lies in their characterization approach. Their mechanism cannot accurately characterize the full spectrum of diverse and complex patterns.

Merlin overcomes this challenge by 1) identifying four key requirements, following general design principles in caching literature, and 2) introducing a new characterization approach that meets all these requirements (Table 1).

Requirements. The first requirement is fine granularity; characterization should not be at the level of access sequences, since a sequence may involve multiple patterns (Figure 5). The second requirement is flexibility, *i.e.*, not assuming a fixed set of access patterns, since such a fixed set is unlikely to match the diversity of real-world access patterns.

The third requirement is that the characterization should capture both short-term access locality (commonly referred to as recency) and long-term access locality (commonly referred to as frequency); as evident in much prior work [8, 13, 36], missing either one leads to high inefficiency.

The final requirement is a cache-size-aware characterization, since the same access sequence may require different handling under different cache sizes [48, 71]. For example, consider a sequence that repeatedly accesses five objects.[4] With a cache size of less than five, an LRU policy does not work, as it thrashes the cache, leading to a zero hit rate. Yet, if the cache size is larger than five, an LRU policy works well.

Object-level characterization. To meet the first requirement, Merlin characterizes at the object-level, the finest granularity possible. For an access sequence, Merlin characterizes by evaluating each accessed object using specific metrics (specifically, hotness and popularity, as detailed next).

Flexible characterization. For the second requirement, Merlin characterizes by using metric distributions (*i.e.*, the number of objects falling into different ranges of the metrics). The continuous nature of metric distributions captures patterns that a fixed set cannot express.

Reflecting short and long-term locality. To meet the third requirement, Merlin first divides an access sequence into *epochs*, each containing as many unique object accesses as the cache size. Merlin measures the short-term locality of an object using *hotness* [5], the number of times an object is accessed in the current epoch. It measures long-term locality through *popularity*, the number of past epochs in which the object was accessed, excluding the current epoch.

Using cache size to decide thresholds. For the last requirement, Merlin accounts for cache size with two steps. First, Merlin introduces a threshold for both hotness and popularity to classify objects. In the hotness dimension, an object is marked as hot or cold based on whether its hotness is above or below the hotness threshold. Likewise, in the popularity dimension, an object is marked as popular or rare by comparing its popularity with the popularity threshold.

Next, Merlin dynamically adjusts the thresholds by considering cache sizes. Suppose that the cache size is S, the hotness (and popularity) threshold is the hotness (and popularity) count of the current S-th hottest (and the most popular) object. The reason is intuitive: the cache should hold the hottest and the most popular objects to maximize hit rate.

Putting it together. With the above design rationale, Merlin's characterization approach is perhaps surprisingly simple. Merlin characterizes workload access pattern by classifying each accessed object into one of the four access types based on its hotness and popularity: hot-popular, hot-rare, cold-popular, and cold-rare. The cache size decides the thresholds for hotness and popularity, and the number of objects in each type reflects the access patterns.

Figure 6 shows an example with the cache size of 3. The access patterns change from *LFU-friendly* at the end of Epoch 2 to *LRU-friendly* at the end of Epoch 4. The change in access type for objects A and C captures this pattern change. Object A is hot-popular at the end of Epoch 2; its hotness is 1, due to an access in Epoch 2, and its popularity is 1, due to that Epoch 1 accesses it. However, at the end of Epoch 4, object A becomes cold-popular with a hotness of 0 (no access in Epoch 4) and a popularity of 3 (accessed in Epochs 1, 2, and 3). Object C changes from hot-rare to hot-popular, due to its frequent accesses in Epochs 3 and 4.

Discussion. In essence, the effectiveness of Merlin's characterization approach lies in meeting all four requirements (Table 1). Among them, the first two are implicitly

[4] For clarity, §4 and §5 assume all objects have a size of one. Our implementation and evaluation consider variable-size objects; see §6

[5] To avoid confusion, we use the terms hotness and popularity rather than the more common, often overloaded terms recency and frequency. For example, ARC uses these terms to distinguish objects accessed once from those accessed multiple times in the recent past. In contrast, we use them to represent short- and long-term locality, respectively.

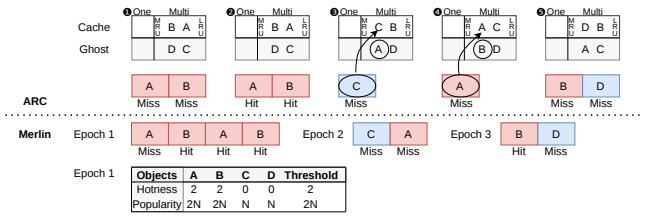


Figure 7: An example of Merlin characterizing access patterns more accurately than ARC. The cache size is 2, with the access sequence repeating ABABCABD. With ARC, the initial cache state in Step ❶ is identical to that in Step 5 (because the access sequence repeats). Thus, accessing A misses and evicts B, and accessing B misses and evicts D. In Step ❸, accessing C misses and evicts A. In Step ❹, accessing A misses and evicts B. In Step ❺, both accesses miss. For Merlin, the table shows the characterization at the end of Epoch 1. In Epoch 1, accessing A misses and evicts D. In Epoch 2, accessing C misses and evicts A, and accessing A misses and evicts C. In Epoch 3, accessing B hits, and accessing D misses and evicts A.

adopted by ARC. We make them explicit here as potentially useful guidelines for future algorithms. The last two are general caching design principles, but prior designs such as ARC and Cacheus do not fully incorporate them.

For the third requirement, there exist other metrics for short- and long-term locality, such as reuse distance for the former and total access count (with a decay factor) for the latter. Merlin uses count-based metrics mainly for their efficiency, as detailed in §5 and §6.

For the fourth requirement, Merlin contributes a new adaptive approach that dynamically adjusts the thresholds based on cache size. Adapting by changing thresholds departs from the prior approach that adapts by 1) changing component sizes (as in ARC) or 2) changing algorithm weights (as in Cacheus). As further explained in §5.1, this new approach is the key for Merlin to avoid the adversarial effect of base algorithms/data structures.

To complement the explanation with real-world traces in Figures 4 and 5, Figure 7 provides an illustrative example of why Merlin characterizes access patterns more accurately than ARC. In this example, as ARC does not consider long-term locality, it cannot distinguish poor-locality objects C and D and good-locality objects A and B. As a result, ARC is effectively reduced to LRU, leading to a 25% hit rate. In contrast, Merlin correctly finds that objects A and B have better locality than objects C and D. Hence, Merlin will keep one of the objects A and B in the cache for the subsequent accesses (object B in this example), leading to a 50% hit rate.

5 Merlin Design

This section presents the design of Merlin, an efficient eviction algorithm that adapts to a broad spectrum of patterns. We present the design overview (§5.1), its components (§5.2) and workflow (§5.3), concluding with examples of how Merlin adapts to various access patterns (§5.4).

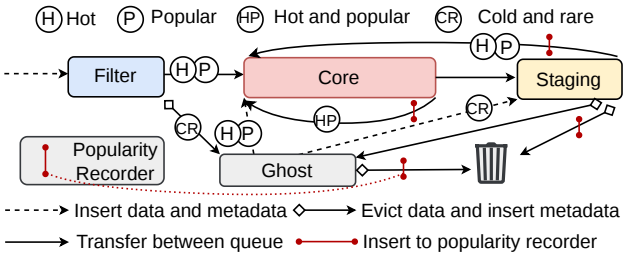


Figure 8: An overview of Merlin's key components and workflow.

5.1 Merlin Design Overview

With its principled characterization method (§4), Merlin's design accomplishes two tasks: 1) track each object's hotness and popularity over time to classify it into one of the four access types; and 2) prioritize keeping hot-popular objects (since they have the best locality), followed by hot-rare and cold-popular objects, while evicting cold-rare objects.

How Merlin adapts. As patterns vary over time, Merlin adapts by dynamically adjusting the hotness and popularity thresholds, which, in turn, changes the number of objects in each access type. As shown in Figure 6, when the pattern shifts from *LRU-friendly* to *LFU-friendly*, Merlin adjusts its thresholds, leading to more popular and fewer hot objects in Epoch 4 than in Epoch 2. In this way, Merlin correctly captures the shift from recency to frequency. Section 5.4 provides more examples of how Merlin adapts.

A cohesive architecture. As dynamic thresholds enable adaptiveness, Merlin's caching architecture forgoes the use of complementary base algorithms/data structures. Merlin thus overcomes the limitation in the adaptiveness approach of prior works (*i.e.*, Limitation #2 in §3.3).

Merlin achieves this cohesive architecture by embracing recent advances in FIFO-queue-based designs, as exemplified by S3-FIFO [71]. Specifically, as S3-FIFO, Merlin uses a small queue to filter out objects with poor locality (*i.e.*, cold-rare objects), and a large queue to store main objects. Moreover, as S3-FIFO and other works [8, 35, 36, 45, 48, 55, 72], Merlin uses a ghost queue to signal and act upon poor evictions.

As detailed next, unlike S3-FIFO, Merlin is adaptive with the following key differences: 1) Merlin admits objects to different caching components based on their access types (rather than fixed access counts); 2) Merlin tracks objects' hotness and popularity, where the latter requires a new component, the popularity recorder, implemented as a count-min sketch (§6); and 3) Merlin adds a staging queue to prioritize hot-popular objects over hot-rare and cold-popular objects.

5.2 Merlin Components

Figure 8 shows Merlin's components and workflow. The main part of Merlin consists of three FIFO queues: a filter queue, a core queue, and a staging queue. In addition, Merlin maintains a FIFO ghost queue and a popularity recorder.

The filter, core, and staging queues store objects and their

metadata, which includes their hotness count and access type. The ghost queue stores only metadata, while the popularity recorder only stores the objects' popularity count.

Task division. The filter queue is a small queue that quickly evicts likely cold-rare objects. The core queue consumes most of the cache space to store hot and/or popular objects. The ghost queue stores metadata for objects evicted by the filter queue, providing a second chance to correctly identify access types. This is necessary since the small filter queue may falsely filter out objects that are hot or popular. The staging queue verifies the object types before the final eviction. The filter, core, ghost, and staging queues also work together to track objects' hotness; these four FIFO queues ensure an invariant that an object is observed for at least one epoch (§4).

The popularity recorder tracks the popularity of objects. Conceptually, it is a two-dimensional counter recording each object's access numbers for past epochs. The actual realization of the popularity recorder is a count-min sketch (§6).

Space allocation. The filter queue, core queue, and the staging queue use 10%, 85%, 5% of the cache space, respectively. The ghost queue stores the metadata of one epoch, while the popularity recorder stores the popularity of all objects accessed in the past 16 epochs. We identify these values through sensitivity analysis (§7.5) and found that they work well across a wide range of real-world workloads.

5.3 Merlin Workflow

Overview. As shown in Figure 8, when an object is accessed for the first time, Merlin places it on the filter queue and starts observing its access type. Upon leaving the filter queue, if its access type is cold-rare, Merlin evicts its data and places its metadata in the ghost queue. Otherwise, Merlin promotes the object to the core queue.

As the core queue is FIFO, the oldest object leaves it when the queue becomes full. If the object is hot-popular, Merlin reinserts it to the core queue, since it has the best locality. Objects of other types are placed in the staging queue, which may lead to eviction.

For an object in the ghost queue, upon access, Merlin must bring it back into the cache. Merlin examines the object's access type and promotes it to the core queue if it is hot or popular. Otherwise, Merlin promotes it to the staging queue. In the latter case, if the staging queue later evicts the object, Merlin places it back in the ghost queue to complete a full epoch of hotness observation.

As discussed earlier, the staging queue stores objects from the core and ghost queues. Upon object eviction, the staging queue verifies its access patterns and reinserts it into the core queue if its access pattern is not cold-rare.

The staging queue thus serves two purposes. For objects from the core queue, the staging queue maximizes the hit rate by retaining objects with good locality, following the spirit of FIFO-reinsertion [20]. For objects from the ghost queue, the staging queue provides yet another chance to verify their access types before eviction.

The popularity recorder tracks, for each object, its access count in the recent epochs. It operates when an object is reinserted into the core queue from 1) the core queue itself, 2) the staging queue, or 3) upon eviction from the ghost queue. On these events, the popularity recorder checks if the object was accessed in the last epoch and, if so, increments its access count for the last epoch.

Finally, with hotness counts tracked in the FIFO queues and popularity counts maintained by the popularity recorder, Merlin updates both thresholds based on their distributions.

Figure 9 shows the detailed algorithm of Merlin.

Cache hits. On a cache hit (L2), Merlin simply updates the hotness of the object (L3) and the hotness distribution (L4).

Cache misses. Upon a cache miss (L5), Merlin first checks if the cache is full, and if so, evicts an object (L12-13). Next, Merlin checks if the object has been previously encountered and resides in the ghost queue (L14). If so, Merlin updates its hotness (L15-16), checks the object's access type (L17), and inserts it into either the core queue (L18) or the staging queue (L20). If not, Merlin treats the object as a new object and inserts it into the filter queue (L23-24).

Object eviction. When the cache is full, Merlin first tries to evict from the filter queue (L28-34). If no object is evicted (*i.e.*, all of them go to the core queue), Merlin evicts from the staging queue (L36). For this, Merlin first reinserts hot-popular objects to the core queue (L44-48) while moving the rest to the staging queue (L50). Upon reinsertion, Merlin updates the object's hotness and popularity for a new epoch (L45-47). Merlin does not directly set hotness to zero, since in practice such a drastic demotion can lead to thrashing. Next, Merlin evicts objects from the staging queue (L51-63). If an evicted object is from the ghost queue, Merlin inserts it back (L57-58). Otherwise, Merlin evicts it completely (L62).

The eviction guarantees to succeed, as objects reinserted to the core queue reduce their hotness and popularity, and thus eventually become cold-rare ones. In practice, we found that the eviction loop rarely iterates more than a few times.

Updating thresholds. Merlin amortizes the overhead of updating the hotness and popularity thresholds (L7-9) by every `ADJUST_INTERVAL` (64 in our implementation) accesses.

Recording objects' popularity. As discussed, Merlin updates popularity upon three events, as shown in L39, L46, L54. Merlin sets the access flag `x.acc` to `True` upon an access (L4, L6) and clears it after updating the popularity (L46, L54).

Deciding thresholds. Figure 10 presents Merlin's remaining algorithm. To decide the thresholds, Merlin maintains the distribution of hotness and popularity counters (*i.e.*, how many objects fall under each specific counter value) (L1-11). Afterward, Merlin scans the distributions from high to low, sums the counts until the total exceeds the cache size, and uses this value as the thresholds (L24–36). This threshold is the minimum hotness or popularity for an object to rank

```
1  def access(x):
2      if x in core or staging or filter: # cache hit
3          oriH = x.hotness, x.hotness = min(oriH+1, MAX_HOTNESS)
4          updateHotDist(oriH, x.hotness), x.acc = True
5      else: # cache miss
6          x.acc = True, insert(x)
7      insertCounter += 1
8      if insertCounter == ADJUST_INTERVAL:
9          adjustThreshold(), insertCounter = 0
10
11 def insert(x):
12     while Merlin is full:
13         evict()
14     if x in ghost: as g
15         oriH = g.hotness, g.hotness = min(oriH+1, MAX_HOTNESS)
16         updateHotDist(oriH, g.hotness)
17         if isHot(g) or isPopular(g):
18             insert x to the head of core
19         else:
20             record x.posInG, insert x to the head of staging
21         remove g from ghost
22     else: #insert new data
23         x.hotness = 0, updateHotDist(None, x.hotness)
24         insert x to the head of filter
25
26 def evict():
27     evicted = False
28     while not evicted and filter exceeds limitF: #evict
29         x ← tail of filter
30         if isHot(x) or isPopular(x):
31             insert x to the head of core
32         else:
33             insert x to the head of ghost
34             evicted = True
35     while not evicted:
36         evicted = evictS()
37     while ghostexceeds limitG: #evict
38         remove tail obj of ghost
39         updateHotDist(obj.hotness, None), recordPop(obj)
40
41 def evictS():
42     while coreexceeds limitC:
43         x ← tail of core
44         if isHot(x) and isPopular(x):
45             if x.acc: #update hotness and popularity for a new epoch
46                 recordPop(x), x.acc = False
47             x.hotness -= 1,updateHotDist(x.hotness+1, x.hotness)
48             insert x to the head of core
49         else:
50             insert x to the head of staging
51     while stagingexceeds limitS: #evict
52         x ← tail of staging
53         if x.acc: #update hotness and popularity for a new epoch
54             recordPop(x), x.acc = False
55         x.hotness -= 1, updateHotDist(x.hotness+1, x.hotness)
56         if isHot(x) or isPopular(x):
57             x.posInG = NULL, insert x to the head of core
58         else: #evict
59             if x.posInG:
60                 insert x to original position in ghost
61             else:
62                 evict x completely
63             return True
```

Figure 9: The core algorithm of Merlin.

among the top S objects, where S is the cache size. Merlin uses the thresholds to classify objects (L38-44).

Progressing epochs. Merlin maintains a global counter, and when the number of inserted objects reaches the cache size (L20), Merlin enters a new epoch (L21-22) by clearing the popularity records of the oldest epoch. This is accurate because the rest of Merlin is FIFO queues, and thus each invocation of `recordpop` entails a new object access.

```
1  def updateHotDist(oriHotness,newHotness):
2      if oriHotness:
3          hotnessDistribution[oriHotness] -= 1
4      if newHotness:
5          hotnessDistribution[newHotness] += 1
6
7  def updatePopDist(oriPopU,newPopU):
8      if oriPopU:
9          popularityDistribution[oriPopU] -= 1
10     if newPopU:
11         popularityDistribution[newPopU] += 1
12
13 def recordPop(x):
14     # sum up the access counts in all epochs for popularity
15     oriPopU = sum(row[x.id] for row in popularityRecoder)
16     popularityRecoder[currentEpoch][x.id]++
17     updatePopDist(oriPopU, oriPopU + 1)
18
19     insertedObjects += 1
20     if insertedObjects reaches CacheSize:
21         insertedObjects = 0, currentEpoch += 1
22         clear popularityRecoder of the oldest epoch
23
24 def adjustThreshold():
25     counter = 0
26     for hotness in reversed(hotnessDistribution):
27         counter += hotness
28         if counter > CacheSize:
29             hotnessThreshold = hotness
30             break
31     counter = 0
32     for popularity in reversed(popularityDistribution):
33         counter += popularity
34         if counter > CacheSize:
35             popularityThreshold = popularity
36             break
37
38 def isHot(x):
39     return x.hotness >= hotnessThreshold
40
41 def isPopular(x):
42     # sum up the access counts in all epochs for popularity
43     PopU = sum(row[x.id] for row in popularityRecoder)
44     return PopU >= popularityThreshold
```

Figure 10: The helper functions for Merlin's algorithm in Figure 9.

5.4 How Merlin Adapts to Access Patterns

With the design in place, this subsection demonstrates how Merlin adapts to diverse access patterns.

The primitive access patterns (§2.2). With an *LRU-friendly* pattern, Merlin tracks hotness and identifies that most objects are hot (hot-popular or hot-rare). The reinsertion behavior of the core and staging queues retains hot objects in the cache, similar to LRU. With an *LFU-friendly* pattern, most objects are popular (hot-popular or cold-popular), and the popularity recorder behaves like LFU by tracking objects' popularity. A *Scan* pattern comprises mostly cold-rare objects, which the filter queue evicts.

The *Churn* pattern is interesting because Merlin behaves differently depending on the cache size. When the cache is large enough to hold all repeatedly accessed objects, they become hot-popular, which Merlin caches to maximize performance. However, if the cache is too small to store all of them, some become cold-popular while the rest remain hot-popular; Merlin caches the latter to prevent thrashing.

Access patterns in prior work. S3-FIFO [71] evicts those objects that are accessed only once, *i.e.*, one-hit wonders. Merlin more accurately characterizes one-hit wonders by

classifying them as either 1) cold-rare objects that are never accessed again, or 2) cold-popular objects that are accessed again after the current epoch. For the former, as in S3-FIFO, Merlin quickly evicts them using the filter queue. For the latter, unlike S3-FIFO, Merlin keeps them due to their good frequency, thereby improving efficiency over S3-FIFO.

LIRS caches objects with low Inter-Reference Recency (IRR), *i.e.*, a small number of requests between consecutive accesses and evicts those with high IRR. However, this policy is suboptimal, since some high IRR objects may still be frequently accessed and therefore good candidates for caching. Merlin identifies these objects as cold-popular and caches them, thereby improving over LIRS.

Complex access patterns. For more complex access patterns, such as those in Figures 4 and 5, Merlin identifies and caches objects with the highest hotness and/or popularity, thereby maximizing performance.

Next, we further explain how Merlin effectively handles the trace in Figure 5. Initially, Merlin sets the hotness threshold to 1, allowing objects hit once (or hit in the ghost-queue) to enter the core queue. This behavior filters out *Scan* (cold) objects. As the workload evolves and more reusable objects appear, Merlin increases the hotness threshold to 2, preventing objects hit once from entering the core queue. This adjustment filters out relatively cold objects, that are part of *LRU-friendly* or *Churn* patterns. Then, the threshold changes from 2 to 5 according to the pattern. Merlin preserves cache capacity for hot objects, thereby improving the hit rate.

In contrast, Cacheus faces a dilemma when workloads contain mixed patterns. It first inserts incoming objects into the cache and later evicts them using a weighted combination of *scan-resistant* LRU and *churn-resistant* LFU policies. For *LRU-friendly* objects, the *churn-resistant* LFU policy makes false eviction decisions, and Cacheus reduces its weight. Then, when the *Churn* pattern emerges, the *scan-resistant* LRU policy performs poorly, causing Cacheus to reduce its weight instead. As a result, Cacheus continuously shifts between policies and cannot adapt effectively to the full mixture of access patterns, resulting in lower hit rates.

6 Realizing Merlin

We implemented Merlin in both CacheLib [11], a development library for caching systems deployed at Facebook, and libCacheSim [2], an efficient cache simulator. We choose these two frameworks because many eviction algorithms (§7) are already implemented on them. We also contributed implementations of ARC and CAR to CacheLib, and an implementation of CAR to libCacheSim.

Realizing the popularity recorder. A challenge Merlin overcomes is minimizing the popularity recorder's space overhead. A naive approach that records object names (*e.g.*, block/page addresses, key values) and popularity counts would incur significant overhead.

Following W-TinyLFU [27], Merlin minimizes such overhead using a count-min sketch [21, 22, 31] with a sliding window. A count-min sketch operates similarly to a counting bloom filter [32, 54] with $\mathcal{H}$ hash functions. It also has $\mathcal{H}$ arrays of counters. Each hash function maps an object to one counter in a row. When an object arrives, the sketch increments all mapped counters. Upon query, the minimum counter value is used as the estimated count. Therefore, a count-min sketch forgoes the need to store an object's name.

In addition, rather than using a count-min sketch for each epoch, Merlin uses a single sketch to reduce space and performance overhead. To approximate access counts in past epochs, Merlin maintains a global counter and increments it when an object arrives. Once the global counter reaches 16 (the number of epochs Merlin tracks) times the cache size, Merlin clears it and halves all counters in the sketch. W-TinyLFU [27] shows this approach is accurate, supported by both theoretical error bounds and evaluation results.

Common optimizations. Merlin applies several common performance and space optimizations in prior work. First, following CacheLib's implementation, Merlin uses a hash table to index all cached objects, minimizing the lookup time.

Second, following [71], Merlin implements the ghost queue with a hash table. For each ghost object, Merlin stores its fingerprint, hotness counter, and a sequence number in the hash table, for a total of 8 bytes. Merlin uses the sequence number to identify if an object is still in the ghost queue, facilitating lazy object deletion. Merlin increments a global sequence number upon each insertion to the ghost queue. Thus, any object with a sequence number less than the global sequence number minus the ghost queue's size has already been evicted from the ghost queue. Thanks to the sequence number, Merlin lazily removes an evicted ghost object from the hash table only upon its access.

Accounting for object sizes. For clarity, the discussion thus far assumes that all objects have a unit size of one. The actual implementation of Merlin handles variable-sized objects. For example, it counts epochs by the total size of unique objects accessed, not their count. It sets the hotness and popularity thresholds for the counts of the hottest and most popular objects whose total size matches the cache size.

Other optimization/implementation. Merlin's current implementation uses a linked list to implement FIFO queues, while one can also implement them with ring buffers. Merlin performs an optimization to avoid deleting the object from the ghost queue upon its promotion to the staging queue (L20 in Figure 9). Therefore, Merlin does not need to record each object's position in the ghost queue. This is possible since a stale object in the ghost queue does not affect the correctness of Merlin; Merlin only checks whether the object is in the ghost queue upon a cache miss (L6) in which case the object must not be in the staging queue.

Tradeoff between cache hit rates and overhead. The count-min sketch and a hash table for the ghost queue may

reduce the hit rates of Merlin. For the former, multiple objects may map to the same counter, and the reset operation is an estimation. For the latter, Merlin may falsely believe that an object is in the ghost queue due to fingerprint collisions. However, our evaluation shows that the impact is negligible.

6.1 Overhead Analysis

This section presents a qualitative analysis of Merlin's overheads with quantitative analysis in §7.3.

Space overhead. The space overhead of Merlin mainly comes from 1) per-object metadata, 2) the ghost queue, and 3) the popularity recorder. Each metadata is 5 bits, consisting of a hotness counter (3 bits), a 1-bit flag recording if the object is from the ghost queue (L14 in §5.3), and a 1-bit access flag (L4). The ghost queue stores the same number of objects as the main cache, and each object consumes an additional 8 bytes. The count-min sketch uses, on average, one byte for each object with a false positive error rate of 1% [21, 22, 31, 32, 54]. Consider a typical object size of 4KB; the space overhead of Merlin is 0.31% of the cache size.

Management overhead. The ultimate goal of a cache is to deliver low latency and high throughput. However, a high hit rate does not guarantee either, since an algorithm may incur high management overhead, *e.g.*, in metadata maintenance.

The management overhead in Merlin is low. Upon a cache access, Merlin searches the object in the hash table. A cache hit only requires updating the hotness counter (L3-4 in §5.3), the access flag, and the hotness map, incurring minimal overhead. On a cache miss, most of the processing time comes from evicting items from the filter or staging queues, involving reinsertion to the core queue. However, as discussed in §5.3, such reinsertion is rare. The rest of the operations (*e.g.*, updates hotness distribution) incur minimal overhead.

Multicore scalability. Multicore scalability is increasingly important for modern caches, which rely on many CPUs to handle the large volumes of requests they receive [6, 8, 71, 72].

Indeed, the design of Merlin is scalable. Following `S3-FIFO` [71] and `SIEVE` [72], Merlin builds on FIFO queues, which are scalable; unlike `LRU` lists, FIFO queues avoid locking the entire queue on every access. The rest of Merlin's design also avoids coarse-grained locks; Merlin uses atomic operations for metadata updates, while the indexing hash table and the count-min sketch deliver good scalability.

Complexity. The complexity of Merlin is modest. While Merlin is more complex than algorithms like `LRU`, `LFU`, and `SIEVE` [72] for it uses multiple components, we argue that Merlin is simpler than machine-learning-based algorithms such as LRB [51] and HALP [52]. In addition, the complexity does not translate to high space or management overhead, as discussed earlier. We found that Merlin's behavior is easy to reason about (§7.6) thanks to its intuitive pattern characterization and clean task division among components.

Self-tuning. Tuning parameters is notoriously difficult for eviction algorithms [6, 45, 48, 55, 71]; Merlin adjusts the hotness and popularity thresholds with complete automation.

Datasets	Type	# Traces	# Requests (million)	# Objects (million)
AlibabaBlock [1, 41, 60]	Block	609	20038	1676
Cloudphysics [57]	Block	106	2115	492
FIU [37]	Block	10	514	19
Systor [38, 39]	Block	6	3531	409
TencentBlock [73, 74]	Block	4611	33593	2172
MSR [5, 47]	Block	14	410	73
Meta CDN [3]	Object	3	231	76
TencentPhoto [76, 77]	Object	2	5649	1038
Wikimedia [4]	Object	3	5667	105
Meta KV [3]	KV	5	13314	371
Twitter [69, 70]	KV	54	253440	26123

Table 2: The datasets evaluated in this work.

6.2 Discussion

Dynamically adjusting component sizes further improves Merlin's adaptiveness. The filter queue size decides how long a new object stays in the cache: a larger queue increases the chance of retaining objects with good locality, thereby improving performance, but it wastes space if the objects' locality is poor. The staging queue reflects a similar tradeoff for previously accessed objects that remain classified as cold-rare in the ghost queue, since they are promoted to the staging queue on access (§5.3).

However, how to adapt size remains an open challenge, as shown by the poor performance of existing size-adaptive algorithms (§3.3). Our view on the poor performance is that they lack a theoretical model to guide the adaptation. As future work, we plan to follow prior performance analysis work (§8) to develop a theoretical model for size adaptation.

Extremely small or large caches may degrade performance for Merlin. For extremely small caches, Merlin cannot rely on short-term access behavior as objects are evicted quickly. With extremely large caches, as a single epoch may span multiple phases with different access patterns, hotness may not accurately reflect the current phase's access pattern.

Despite the above, with the configuration in §7.2, Merlin ranks first in 5 datasets under an extremely large cache (20% of WSS), and ranks first or second in 7 datasets under an extremely small cache (0.3% of WSS). Figures are not shown.

Dynamic memory budgets. Merlin assumes a fixed cache size, but in multi-tenant or elastic environments, cache capacity may change dynamically. Such changes can invalidate the hotness and popularity thresholds, since they depend on cache capacity. It requires Merlin to re-adjust these thresholds, which we left as future work.

7 Evaluation

Our evaluation answers the following questions:

- Does Merlin adapt to various real-world workloads across different cache sizes (§7.2)?
- Does Merlin achieve high throughput and scale with thread counts (§7.3)?

- Is Merlin friendly to flash-based cache systems (§7.4)?
- How sensitive is Merlin to different parameters (§7.5)?
- Why is Merlin robust and effective (§7.6)?

7.1 Evaluation Setup

Environment. Our evaluation platform is a 2-socket machine equipped with a 192-core AMD EPYC 9965 Processor. The system runs Debian 12 and Linux kernel 6.13.8. We disable hyperthreading, turbo boost to obtain stable results.

Workload. We used 11 open-source datasets with 5423 traces to evaluate Merlin, as shown in Table 2. These traces are diverse in both application domains, including key-value, object CDN, and block caches, and time periods, spanning from 2008 to 2023. In total, these datasets contain 338 billion requests, 33 billion objects, and 11 PB of traffic for a total of 1837.2 TB of data. The object size varies from 1 B to 4 GB across workloads. Following prior work [71], we split Tencent CBS [73] and Alibaba [1] into per-tenant traces.

To demonstrate the effectiveness of Merlin on different cache types, we evaluate both scenarios: fixed-size objects and variable-size objects. In total, we ran 38.5 trillion requests, using a million CPU hours.

Baseline. Merlin is compared with 16 eviction algorithms, including 1) classical algorithms: `LRU`, `LFU`, `FIFO` and `2Q` [36], 2) recent static algorithms: `S3-FIFO` [71], `LIRS` [35], `LHD` [8], `W-TinyLFU` [27], GDSF [18], Hyperbolic [13], SIEVE [72], 3) existing adaptive algorithms: `ARC` [45], `CAR` [6], `Cacheus` [48], `LeCaR` [55]. and 4) a machine-learning-based algorithm: GL-Cache [68]. We follow the default configurations of these algorithms. Given the large number of baseline algorithms, presented figures show only the best performers for clarity.

Configuration. We evaluate for three aspects: hit rate, throughput (for management overhead and multicore scalability), and flash-friendliness (for applicability in flash caches). For hit rates, given the thousands of traces, we follow the conventional approach [8, 48, 68, 71, 72] to evaluate with a cache simulator: `libCacheSim` [2]. We evaluate throughput with Cachelib (§6). For flash-friendliness, we evaluate Merlin on an extension of `libCacheSim`.

7.2 Hit rate

Hit rate improvement. Figure 11 shows the hit rate of evaluated algorithms. Following [6, 27, 35, 45, 48], we report the improvement over `LRU` (*i.e.*, $\frac{HR_{algo}-HR_{lru}}{HR_{lru}}$), since hit rate ranges widely given many traces and baselines. Merlin significantly outperforms others. On average, Merlin improves the hit rate over LRU by 10.4% when cache size is 10% of the working set size (WWS). In contrast, `S3-FIFO`, `Cacheus`, and `ARC` achieve 7.1%, 6.8%, and 6.1%, respectively. In terms of robustness, Merlin ranks first in 6 out of 11 datasets. As a comparison, `S3-FIFO` is the best runner-up, ranking first in 3 datasets. Merlin ranks second in another 3 datasets, closely matching the best performance with only 0.5% difference.

Furthermore, as shown in the figure, the performance of Merlin remains robust across different cache sizes; Merlin is always the best in 6 out of 11 datasets and ranks second in another 3 datasets for all cache sizes. The performance of `ARC` and `Cacheus` varies significantly with cache size. (See Cloudphysics, Systor, FIU, and TencentPhoto datasets.)

Comparison to dominant algorithms. To further evaluate robustness, we compare Merlin to a dominant algorithm, defined as the best-performing algorithm for each trace under a given cache size.

Figure 13 shows the results of datasets with more than 100 traces, illustrating the robustness of Merlin. For example, in Tencent CBS dataset with a 10% WSS cache size, Merlin's value at the 1% CDF point (P1) is roughly 88%. This means that, for 99% of traces, Merlin achieves at least 88% of the dominant algorithm's hit rate (note that the dominant algorithm may vary across traces). In contrast, `S3-FIFO`, `Cacheus`, and `ARC` achieve only about 83%, 76%, and 76%, respectively. Among the 4,611 Tencent CBS traces, `Cacheus` and `ARC` have about 138 traces (P3) whose performance falls below 87% of the dominant algorithm, while Merlin achieves 92.5%.

Adversarial workloads for Merlin. To understand the limitations of Merlin, we analyze the traces where Merlin underperforms the best-performing algorithm by more than 5%. These adversarial traces account for 2.9% of all traces. Among them, we identify two representative classes.

The first class is unpredictable workloads, where the past access history does not reflect future reuse behavior. This breaks an inherent assumption of adaptive algorithms, making Merlin, other adaptive algorithms, as well as most static algorithms, underperform.

In particular, we observe three main patterns in this class of objects. First, the trace repeatedly accesses a small set of objects for a long period, but accesses these objects only rarely afterward. Second, the trace accesses a set of objects for a certain amount of time, and then accesses a mostly different set of objects. Third, the trace repeatedly alternates between two phases: (a) a scan phase that touches many objects once, and (b) a phase that repeatedly accesses a small set of hot objects, with the hot set changing in every iteration. In the above cases, Merlin may cache the wrong objects for a certain period of time (due to their hotness or popularity in the past), leading to a lower hit rate.

Another class is workloads that cause high false positives in the count-min sketch (§6), therefore making Merlin misclassify unpopular objects as popular. These workloads account for 1.1% of the total traces.

The small amount of adversarial workloads may potentially include correlated workloads, where objects that are semantically related are accessed together. For example, when a user visits a profile page, it may access a set of objects semantically related to that page, *e.g.*, the user's profile picture, recent posts, and friend list.

Merlin (and many prior algorithms) do not consider the semantics of objects, and thus cannot handle such workloads

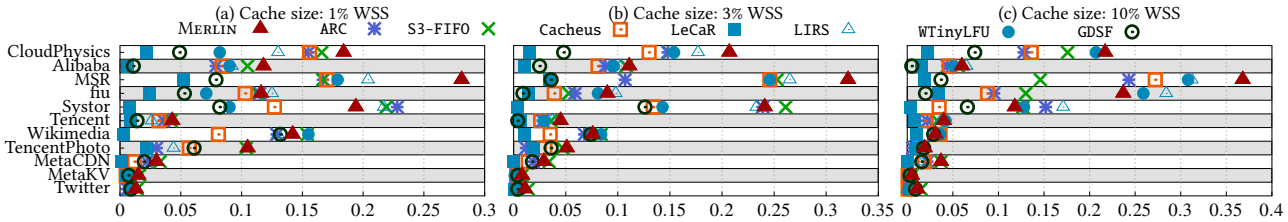


Figure 11: Hit rate of evaluated algorithms. The x-axis shows hit rate improvement over LRU. Only the best-performing baselines are shown.

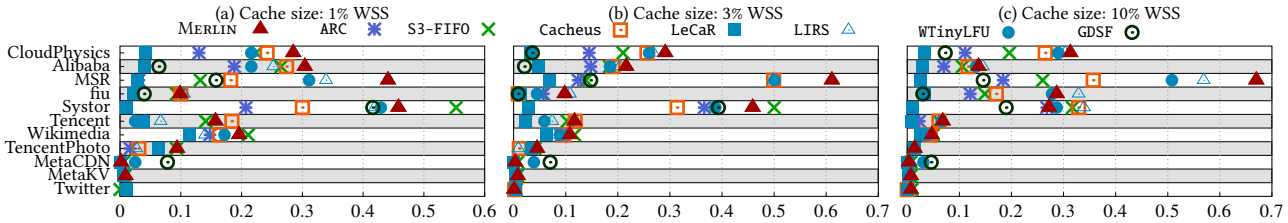


Figure 12: Byte hit rate of evaluated algorithms The x-axis shows improvement over LRU. Only the best-performing baselines are shown.

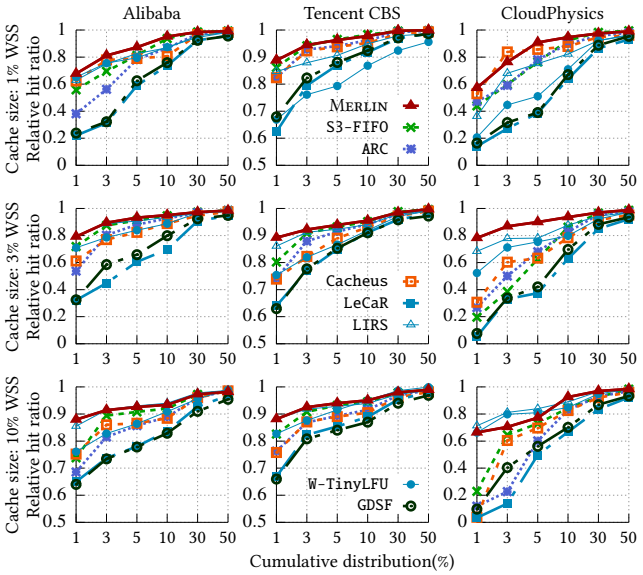


Figure 13: Relative hit rate compared to the dominant algorithm. The y-axis reports the relative hit rate, and the x-axis presents the ascending-sorted cumulative distribution of traces.

well. We did not report the portion of these correlated workloads, as it is difficult to manually identify them without a semantic-aware caching algorithm.

Byte hit rate improvement. The hit rate measures the fraction of requests served from the cache, while another metric is the byte hit rate, *i.e.*, the fraction of requested bytes served from the cache (or $\frac{\sum(ObjSize_{hit})}{\sum(ObjSize_{acc})}$) This metric considers object sizes, and is important for CDN, which cares about bandwidth savings. Figure 12 shows the results, where MERLIN exhibits trends similar to Figure 11.

7.3 Throughput

To evaluate the throughput and management overhead in complex scenarios, we identify typical diverse and shifting access patterns and generate a representative mixed trace, following prior work [8, 71]. We analyze thousands of traces across all datasets and find that most exhibit four primitive access patterns at scale (millions of requests).

We use four distributions to represent four primitive access patterns and randomly shuffle them. They are 1) a Zipfian distribution, 2) a uniform distribution, 3) repeated sequential accesses, and 4) scan accesses. Then we randomly shuffle them to approach the mixed access patterns in real traces, generating a 200M-request trace with 2M objects. We simulate a 10 μs backend latency when there are cache misses.

Multicore scalability. Figure 14 (a) shows the throughput of evaluated algorithms under thread counts from 1 to 32. MERLIN outperforms others significantly, achieving 1.4× to 7.8× higher throughput than others at 32 threads. The reasons are twofold. First, MERLIN achieves a high hit rate, reducing backend accesses. For this reason, MERLIN outperforms S3-FIFO by around 1.4× across different thread counts. Second, MERLIN demonstrates strong scalability due to its multi-FIFO queue design (§6.1), while LRU and ARC suffer from lock contention on the LRU list.

Management overhead. Figure 14 (b) removes backend latency to evaluate management overhead (§6.1). MERLIN incurs low overhead, thanks to the optimizations in §6, and delivers 16% higher throughput than S3-FIFO. Both MERLIN and S3-FIFO achieve 2-3× higher throughput compared to them with backend latency, showing that cache operations are not the bottleneck in real systems. In contrast, LRU and ARC continue to suffer from lock contention.

7.4 Flash-friendliness

Modern datacenters use both DRAM and flash devices for caching [28, 44, 50]. In this setup, a key metric is how much data is written to flash, since excessive writes reduce device lifetime. This subsection evaluates this scenario.

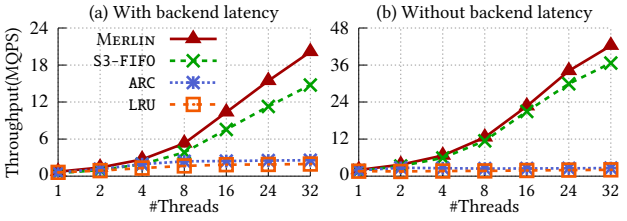


Figure 14: Throughput of the evaluated algorithms.

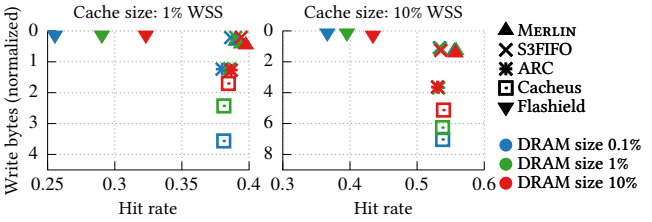


Figure 15: Write bytes (normalized to trace size) and hit rate in Cloudphysics. The upper-right corner is the best.

Setup. MERLIN naturally fits this setup due to its modular design (§5.2): its small filter queue and metadata sit in DRAM, while the staging and core queues reside in flash. We also evaluate several representative baselines. For `Flashield` [28] and `S3-FIFO` [71], we use their original designs. For `ARC`, we set its one-hit list into DRAM and frequency list in flash. For `Cacheus`, we split the SR-LRU list, keeping the SR portion in DRAM and the R portion in flash.

Adaptiveness in flash-friendliness. Figure 15 shows the flash-friendliness of evaluated algorithms, where we vary 1) the cache size from 1% to 10% of the WSS and 2) the DRAM size from 0.1% to 10% of the cache size. MERLIN achieves the highest hit rate with low write bytes. Compared to adaptive algorithms, `ARC` and `Cacheus`, MERLIN reduces write bytes by 70% and improves absolute hit rate by 1–2%. Compared with `S3-FIFO`, MERLIN produces nearly the same write volume but has 1% absolute hit rate improvement. `Flashield` reduces write bytes but at the cost of a substantial drop in hit rate.

MERLIN performs well in DRAM–Flash hierarchical caching due to both its task-division design (§5.2) and its accurate access-pattern characterization (§4). Compared to `Cacheus` and `ARC`, MERLIN's component separation avoids unnecessary flash writes. Compared to `S3-FIFO`, more accurate access-pattern identification further improves the hit rate.

7.5 Sensitivity Analysis

This subsection evaluates the sensitivity of MERLIN's tunable parameters, including 1) the sizes of the filter, staging, and ghost queue; 2) the length of epoch to decide popularity (§5.2); and 3) the size of the count-min sketch (§6).

Queue sizes. Figure 16 shows, across all datasets, how different queue sizes affect MERLIN's hit rates. We find that for the evaluated ranges of the filter (5% - 15%), staging (1% - 10%) queues, and ghost (50% - 200% of the cached objects) queues, the hit rate of MERLIN is stable; only a few traces

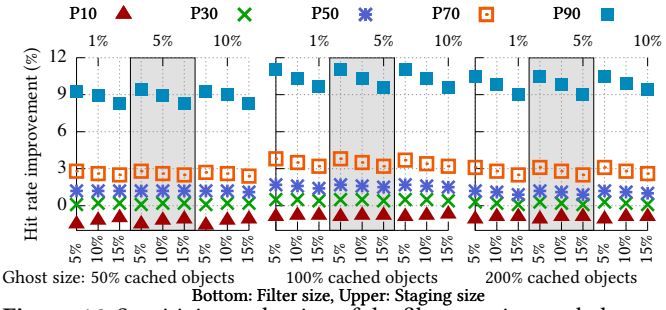


Figure 16: Sensitivity evaluation of the filter, staging, and ghost queue sizes (%). The filter queue ranges 5%, 10%, and 15%; the staging queue is set to 1%, 5%, and 10%; and the ghost queue varies across 50%, 100%, and 200% of the cache size. Cache size is 10% of WSS

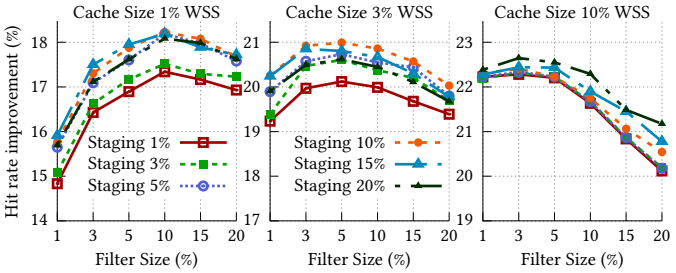


Figure 17: Sensitivity evaluation of the filter and staging queue sizes (%) in Cloudphysics dataset. The queue sizes range from 1% to 20% of the cache size. Cache size ranges from 1% to 10% of WSS.

show more than 1% hit-rate variation. Among these configurations, the 10% filter, 5% staging, and 100% ghost queues achieve the best overall performance.

Sensitivity across different datasets. To further understand the impact of queue sizes, we evaluate 1) a wider range of queue sizes; and 2) the hit rates across different datasets.

Figure 17 shows the results of CloundPhysics, which represents the behavior of most datasets. MERLIN's hit rate is stable in a wide range of filter and staging queue sizes (5%-15%) with all cache sizes. MERLIN only underperforms in two extreme cases: 1) a small filter queue (<5%) under a small cache size (1% of WSS); and 2) a large filter or staging queue (>15%) under a large cache size (10% of WSS).

A few datasets show a slightly different behavior. Specifically, for Meta CDN and Tencent Photo with a 10% WSS cache size, a large filter queue increases the hit-rate improvement over LRU by up to 3%. For Wikimedia with a 1% WSS cache size, a small filter queue, and a large staging queue achieves up to 5% improvement.

The length of recorded epochs and the size of the count-min sketch. We evaluate the sensitivity of MERLIN to the recorded epoch length (Figure 18) and the count-min sketch size (Figure 19) across all datasets. In summary, MERLIN achieves robust performance across a wide range of configurations. For the recorded epoch length ranging from 4 to 128, the hit rate of MERLIN is stable in most traces. For a wide range of count-min sketch sizes, corresponding to false positive rates from 0.1% to 5%, MERLIN achieves a robust

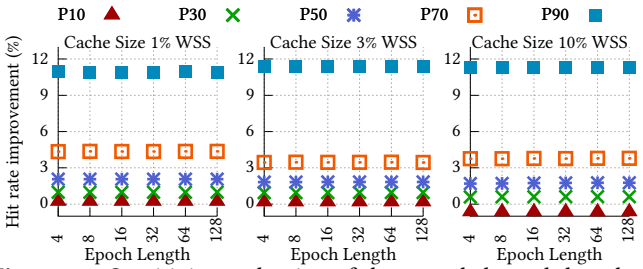


Figure 18: Sensitivity evaluation of the recorded epoch length, which ranges from 4 to 128.

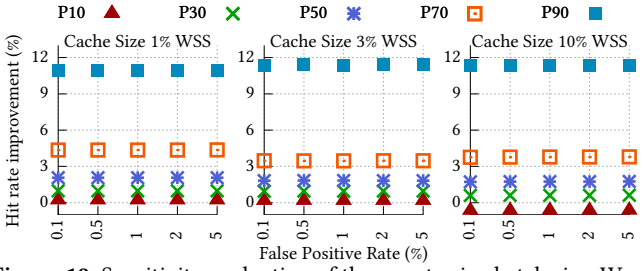


Figure 19: Sensitivity evaluation of the count-min sketch size. We vary the false positive rate (FP) from 0.1% to 5%.

performance, with a few traces benefiting from larger sizes.

In a subset of workloads (8% of the total traces), the recorded epoch length and the count-min sketch cause more than 1% variation in hit rate improvement over LRU (figures not shown). This includes around 25% of CloudPhysics traces, 17% of Alibaba traces, 5% of Tencent CBS traces.

Epoch length. With Merlin, the epoch length is set to the cache size. We note that this is not a tunable parameter, as Merlin uses the properties of FIFO queues to track objects within an epoch (for hotness), as discussed in §5.2.

Therefore, to understand the sensitivity of epoch length, we use the ghost queue size to approximate it. Specifically, a small ghost queue approximates a shorter epoch, as Merlin observes hotness for a smaller amount of time. Vice versa, a large ghost queue approximates a longer epoch. As shown in Figure 16, varying ghost queue sizes has little impact, suggesting that Merlin is robust to different epoch lengths.

7.6 Why is Merlin effective?

To evaluate why Merlin is effective, we measure the precision and average hit count of objects promoted from ghost queues across different adaptive algorithms. The precision is the fraction of promoted objects that are later accessed again, indicating the decision accuracy. The average hit count captures how valuable these promoted objects actually are.

Figure 20 reports precision and average hit counts across Twitter and FIU traces. Merlin performs well on both metrics, demonstrating its accurate identification of hot and/or popular objects. In comparison, ARC and Cacheus show much lower precision and hit counts, demonstrating that they do not make good caching decisions.

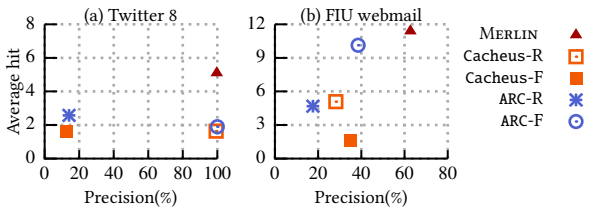


Figure 20: Precision of access pattern identification.

8 Other Related Work

The most important related work (*e.g.*, Cacheus and ARC) has been discussed throughout the paper.

Machine learning based algorithms. LRB [51] and HALP [52] model access sequences and feed object features (*e.g.*, timestamps, gaps, and frequencies) into machine learning predictors to select eviction candidates. These approaches incur high management overhead due to their online training and inference costs.

Performance analysis. HOTL [65], Counter stack [62], and Footprint sampling [63, 64] provide theoretical analysis and approximation methods to predict the hit rate. SHARDS [57], Miniature [56], and Kosmo [49] build small-scale models to estimate cache performance for different configurations. Such work complements Merlin.

Cliff removal. Ideal eviction policies yield convex hit-rate growth as cache size increases, but real-world policies often show non-convex "cliffs." Talus [9] and Cliffhanger [19] mitigate this by segmenting the cache without modifying the underlying algorithm. This work is orthogonal to Merlin.

9 Conclusion

This paper presents Merlin, an adaptive cache eviction algorithm that achieves high hit rates across diverse real-world workloads. Merlin's effectiveness stems from two key designs. First, Merlin proposes a principled fine-grained access characterization method that also accounts for locality and cache size. Specifically, Merlin classifies objects into four types based on hotness and popularity, where the cache size determines the thresholds, and the distribution of types reflects the access patterns. Such a classification allows Merlin to accurately capture a wide spectrum of access patterns beyond a few typical ones. Second, Merlin employs a unified and cohesive architecture that cleanly separates component responsibilities, thereby eliminating the interference seen in prior designs. Evaluations on 11 datasets and 5,423 traces show that Merlin achieves high efficiency, scalability, and low management overhead.

10 Acknowledgment

We thank our shepherd and the anonymous reviewers for their insightful comments. The work is supported by the National Key R&D Program of China under Grant No. 2023YFB4502303, and by the National Science Foundation of China (Nos. 62372011, 92582118, 62032001). Diyu Zhou is the corresponding author.

References

[1] Alibaba block-trace. https://github.com/alibaba/block-traces.

[2] libCacheSim: a high-performance cache simulator. https://github.com/1a1a11a/libCacheSim.

[3] Running cachebench with the trace workload. https://cachelib.org/docs/Cache$_{L}$ibrary$_{U}$ser$_{G}$uides/Cachebench$_{F}$B$_{H}$W$_{e}$val.

[4] Wikimedia: Analytics/data-lake/traffic/caching. https://wikitech.wikimedia.org/wiki/Analytics/Data$_{L}$ake/Traffic/Caching.

[5] Storage Networking Industry Association et al. MSR Cambridge Traces. *http://iotta. snia. org/traces/388*, 2010.

[6] Sorav Bansal, Dharmendra S Modha, et al. CAR: Clock with Adaptive Replacement. In *3rd USENIX Conference on File and Storage Technologies (FAST) (FAST 04)*, San Francisco, CA, March 2004.

[7] Soumya Basu, Aditya Sundarrajan, Javad Ghaderi, Sanjay Shakkottai, and Ramesh Sitaraman. Adaptive TTL-Based Caching for Content Delivery. *IEEE/ACM transactions on networking*, 2018.

[8] Nathan Beckmann, Haoxian Chen, and Asaf Cidon. LHD: Improving Cache Hit Rate by Maximizing Hit Density. In *Proceedings of the 15th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, Renton, WA, March 2018.

[9] Nathan Beckmann and Daniel Sanchez. Talus: A Simple Way to Remove Cliffs in Cache Performance. In *Proceedings of the 21st IEEE Symposium on High Performance Computer Architecture (HPCA)*, San Francisco, CA, February 2015.

[10] Laszlo A. Belady. A Study of Replacement Algorithms for A Virtual-Storage Computer. *IBM Systems journal*, 1966.

[11] Benjamin Berg, Daniel S Berger, Sara McAllister, Isaac Grosof, Sathya Gunasekar, Jimmy Lu, Michael Uhlar, Jim Carrig, Nathan Beckmann, Mor Harchol-Balter, et al. The CacheLib Caching Engine: Design and Experiences at Scale. In *Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Virtual, October 2020.

[12] Daniel S Berger, Ramesh K Sitaraman, and Mor Harchol-Balter. AdaptSize: Orchestrating the Hot Object Memory Cache in A Content Delivery Network. In *Proceedings of the 14th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, Boston, MA, March 2017.

[13] Aaron Blankstein, Siddhartha Sen, and Michael J Freedman. Hyperbolic Caching: Flexible Caching for Web Applications. In *Proceedings of the 2017 USENIX Annual Technical Conference (ATC)*, Santa Clara, CA, July 2017.

[14] Ali R Butt, Chris Gniady, and Y Charlie Hu. The Performance Impact of Kernel Prefetching on Buffer Cache Replacement Algorithms. In *Proceedings of the 2005 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, 2005.

[15] Miguel Castro, Atul Adya, Barbara Liskov, and Andrew C. Myers. HAC: Hybrid Adaptive Caching for Distributed Storage Systems. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles (SOSP)*, Saint-Malo, France, October 1997.

[16] Jiayi Chen, Nihal Sharma, Tarannum Khan, Shu Liu, Brian Chang, Aditya Akella, Sanjay Shakkottai, and Ramesh K Sitaraman. Darwin: Flexible Learning-based CDN Caching. In *Proceedings of the 2023 ACM SIGCOMM*, New York, NY, September 2023.

[17] Zhifeng Chen, Yuanyuan Zhou, and Kai Li. Eviction-based Cache Placement for Storage Caches. In *Proceedings of the 2003 USENIX Annual Technical Conference (ATC)*, San Antonio, TX, June 2003.

[18] Ludmila Cherkasova. *Improving WWW Proxies Performance with Greedy-Dual-Size-Frequency Caching Policy*. Hewlett-Packard Laboratories Palo Alto, CA, USA, 1998.

[19] Asaf Cidon, Assaf Eisenman, Mohammad Alizadeh, and Sachin Katti. Cliffhanger: Scaling Performance Cliffs in Web Memory Caches. In *Proceedings of the 13th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, Santa Clara, CA, March 2016.

[20] Fernando J Corbato. *A Paging Experiment with the Multics System*. Massachusetts Institute of Technology, 1968.

[21] Graham Cormode. Count-Min Sketch. *Encyclopedia of Database Systems. Springer*, 2003.

[22] Graham Cormode and Shan Muthukrishnan. An Improved Data Stream Summary: the Count-Min Sketch and Its Applications. *Journal of Algorithms*, 2005.

[23] Christina Delimitrou and Christos Kozyrakis. Paragon: QoS-aware scheduling for heterogeneous datacenters. In *Proceedings of the 18th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Houston, TX, March 2013.

[24] Christina Delimitrou and Christos Kozyrakis. Quasar: Resource-efficient and QoS-aware Cluster Management. In *Proceedings of the 19th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Salt Lake City, Utah, March 2014.

[25] Peter J Denning. The Working Set Model for Program Behavior. *Communications of the ACM*, 1968.

[26] Yingkai Dong, Xiangtao Meng, Ning Yu, Zheng Li, and Shanqing Guo. Fuzz-Testing Meets LLM-Based Agents: An Automated and Efficient Framework for Jailbreaking Text-to-Image Generation Models . In *Proceedings of the 46th IEEE Symposium on Security and Privacy (Oakland)*, Los Alamitos, CA, May 2025.

[27] Gil Einziger, Roy Friedman, and Ben Manes. TinyLFU: A Highly Efficient Cache Admission Policy. *ACM Transactions on Storage (ToS)*, 2017.

[28] Assaf Eisenman, Asaf Cidon, Evgenya Pergament, Or Haimovich, Ryan Stutsman, Mohammad Alizadeh, and Sachin Katti. Flashield: a Hybrid Key-value Cache that Controls Flash Write Amplification. In *Proceedings of the 16th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, Boston, MA, February 2019.

[29] Shuitao Gan, Chao Zhang, Xiaojun Qin, Xuwen Tu, Kang Li, Zhongyu Pei, and Zuoning Chen. Collafl: Path Sensitive Fuzzing. In *Proceedings of the 39th IEEE Symposium on Security and Privacy (Oakland)*, San Francisco, CA, May 2018.

[30] Yoann Ghigoff, Julien Sopena, Kahina Lazri, Antoine Blin, and Gilles Muller. BMC: Accelerating Memcached Using Safe In-Kernel Caching and Pre-Stack Processing. In *Proceedings of the 18th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, Boston, MA, April 2021.

[31] Xiangyang Gou, Yinda Zhang, Zhoujing Hu, Long He, Ke Wang, Xilai Liu, Tong Yang, Yi Wang, and Bin Cui. A Sketch Framework for Approximate Data Stream Processing in Sliding Windows. *IEEE Transactions on Knowledge and Data Engineering*, 2022.

[32] Deke Guo, Yunhao Liu, Xiangyang Li, and Panlong Yang. False Negative Problem of Counting Bloom Filter. *IEEE transactions on knowledge and data engineering*, 2010.

[33] Xiaojun Guo, Hua Wang, Ke Zhou, Hong Jiang, Yaodong Han, and Guangjie Xing. FLOWS: Balanced MRC Profiling for Heterogeneous Object-Size Cache. In *Proceedings of the 19th European Conference on Computer Systems (EuroSys)*, Athens, Greece, April 2024.

[34] Song Jiang, Feng Chen, and Xiaodong Zhang. CLOCK-Pro: An Effective Improvement of the CLOCK Replacement. In *Proceedings of the 2005 USENIX Annual Technical Conference (ATC)*, Anaheim, CA, April 2005.

[35] Song Jiang and Xiaodong Zhang. LIRS: An Efficient Low Inter-Reference Recency Set Replacement Policy to Improve Buffer Cache Performance. *ACM SIGMETRICS Performance Evaluation Review*, 2002.

[36] Theodore Johnson, Dennis Shasha, et al. 2Q: A Low Overhead High Performance Buffer Management Replacement Algorithm. In *Proceedings of the 20th International Conference on Very Large Data Bases (VLDB)*, Santiago de Chile, Chile, September 1994.

[37] Ricardo Koller and Raju Rangaswami. I/O Deduplication: Utilizing Content Similarity to Improve I/O Performance. *ACM Transactions on Storage (TOS)*, 2010.

[38] Chunghan Lee, Tatsuo Kumano, Tatsuma Matsuki, Hiroshi Endo, Naoto Fukumoto, and Mariko Sugawara. Systor'17 Traces (SNIA IOTTA Trace 5102). *SNIA IOTTA Trace Repository. Storage Networking Industry Association*, 2016.

[39] Chunghan Lee, Tatsuo Kumano, Tatsuma Matsuki, Hiroshi Endo, Naoto Fukumoto, and Mariko Sugawara. Understanding Storage Traffic Characteristics on Enterprise Virtual Desktop Infrastructure. In *Proceedings of the 10th ACM International Systems and Storage Conference*, Haifa, Israel, May 2017.

[40] Donghee Lee, Jongmoo Choi, Jong-Hun Kim, Sam H Noh, Sang Lyul Min, Yookun Cho, and Chong Sang Kim. LRFU: A Spectrum of Policies that Subsumes the Least Recently Used and Least Frequently Used Policies. *IEEE Transactions on Computers*, 2001.

[41] Jinhong Li, Qiuping Wang, Patrick PC Lee, and Chao Shi. An in-Depth Analysis of Cloud Block Storage Workloads in Large-Scale Production. In *2020 IEEE International Symposium on Workload Characterization (IISWC)*, Beijing, China, October 2020.

[42] Chao Lin, Da-Wei Chang, Wei-Jun Chen, Jian-Ting Ke, and Po-Han Huang. Global Clean Page First Replacement and Index-Aware Multi-stream Prefetcher in Hybrid Memory Architecture. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2019.

[43] Ke Liu, Kan Wu, Hua Wang, Ke Zhou, Peng Wang, Ji Zhang, and Cong Li. SLAP: Segmented Reuse-Time-Label Based Admission Policy for Content Delivery Network Caching. *ACM Transactions on Architecture and Code Optimization*, 2024.

[44] Sara McAllister, Benjamin Berg, Julian Tutuncu-Macias, Juncheng Yang, Sathya Gunasekar, Jimmy Lu, Daniel S. Berger, Nathan Beckmann, and Gregory R. Ganger. Kangaroo: Caching Billions of Tiny Objects on Flash. In *Proceedings of the 28th ACM Symposium on Operating Systems Principles (SOSP)*, Koblenz, Germany, October 2021.

[45] Nimrod Megiddo and Dharmendra S Modha. ARC: A Self-Tuning, Low Overhead Replacement Cache. In *2nd USENIX Conference on File and Storage Technologies (FAST) (FAST 03)*, San Francisco, CA, March 2003.

[46] Antoine Murat, Clément Burgelin, Athanasios Xygkis, Igor Zablotchi, Marcos Kawazoe Aguilera, and Rachid Guerraoui. SWARM: Replicating Shared Disaggregated-Memory Data in No Time. In *Proceedings of the 30th ACM Symposium on Operating Systems Principles (SOSP)*, Austin, TX, October 2024.

[47] Dushyanth Narayanan, Austin Donnelly, and Antony Rowstron. Write Off-Loading: Practical Power Management for Enterprise Storage. *ACM Transactions on Storage (TOS)*, 2008.

[48] Liana V Rodriguez, Farzana Yusuf, Steven Lyons, Eysler Paz, Raju Rangaswami, Jason Liu, Ming Zhao, and Giri Narasimhan. Learning Cache Replacement with CACHEUS. In *19th USENIX Conference on File and Storage Technologies (FAST) (FAST 21)*, Virtual, February 2021.

[49] Kia Shakiba, Sari Sultan, and Michael Stumm. Kosmo: Efficient Online Miss Ratio Curve Generation for Eviction Policy Evaluation. In *22nd USENIX Conference on File and Storage Technologies (FAST) (FAST 24)*, Santa Clara, CA, February 2024.

[50] Zhaoyan Shen, Feng Chen, Yichen Jia, and Zili Shao. DIDACache: A Deep Integration of Device and Application for Flash Based Key-Value Caching. In *15th USENIX Conference on File and Storage Technologies (FAST) (FAST 17)*, Santa Clara, CA, February 2017.

[51] Zhenyu Song, Daniel S. Berger, Kai Li, and Wyatt Lloyd. Learning Relaxed Belady for Content Distribution Network Caching . In *Proceedings of the 17th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, Santa Clara, CA, February 2020.

[52] Zhenyu Song, Kevin Chen, Nikhil Sarda, Deniz Altınbüken, Eugene Brevdo, Jimmy Coleman, Xiao Ju, Pawel Jurczyk, Richard Schooler, and Ramki Gummadi. HALP: Heuristic Aided Learned Preference Eviction Policy for YouTube Content Delivery Network. In *Proceedings of the 20th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, Boston, MA, April 2023.

[53] FH Sumner. One-level Storage System. *IEEE Transactions on Computers*, 1962.

[54] Sasu Tarkoma, Christian Esteve Rothenberg, and Eemil Lagerspetz. Theory and Practice of Bloom Filters for Distributed Systems. *IEEE Communications Surveys & Tutorials*, 2011.

[55] Giuseppe Vietri, Liana V Rodriguez, Wendy A Martinez, Steven Lyons, Jason Liu, Raju Rangaswami, Ming Zhao, and Giri Narasimhan. Driving Cache Replacement with ML-based LeCaR. In *10th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 18)*, 2018.

[56] Carl Waldspurger, Trausti Saemundsson, Irfan Ahmad, and Nohhyun Park. Cache Modeling and Optimization Using Miniature Simulations. In *Proceedings of the 2017 USENIX Annual Technical Conference (ATC)*, Santa Clara, CA, July 2017.

[57] Carl A Waldspurger, Nohhyun Park, Alexander Garthwaite, and Irfan Ahmad. Efficient MRC Construction with SHARDS. In *13th USENIX Conference on File and Storage Technologies (FAST) (FAST 15)*, San Francisco, CA, February 2015.

[58] Kefei Wang and Feng Chen. Catalyst: Optimizing Cache Management for Large In-memory Key-value Systems. *Proc. VLDB Endow.*, 2023.

[59] Kefei Wang, Jian Liu, and Feng Chen. Put an Elephant into a Fridge: Optimizing Cache Efficiency for In-Memory Key-Value Stores. *Proceedings of the VLDB Endowment*, 2020.

[60] Qiuping Wang, Jinhong Li, Patrick PC Lee, Tao Ouyang, Chao Shi, and Lilong Huang. Separating Data Via Block Invalidation Time Inference for Write Amplification Reduction in Log-Structured Storage. In *20th USENIX Conference on File and Storage Technologies (FAST) (FAST 22)*, Santa Clara, CA, February 2022.

[61] Yang Wang, Hao Dai, Xinxin Han, Pengfei Wang, Yong Zhang, and Chengzhong Xu. Cost-driven Data Caching in Edge-based Content Delivery Networks. *IEEE Transactions on Mobile Computing*, 2021.

[62] Jake Wires, Stephen Ingram, Zachary Drudi, Nicholas JA Harvey, and Andrew Warfield. Characterizing Storage Workloads with Counter Stacks. In *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Broomfield, Colorado, October 2014.

[63] Xiaoya Xiang, Bin Bao, Tongxin Bai, Chen Ding, and Trishul Chilimbi. All-Window Profiling and Composable Models of Cache Sharing. *ACM SIGPLAN Notices*, 2011.

[64] Xiaoya Xiang, Bin Bao, Chen Ding, and Yaoqing Gao. Linear-Time Modeling of Program Working Set in Shared Cache. In *Proceedings of the 20th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, Galveston, TX, October 2011.

[65] Xiaoya Xiang, Chen Ding, Hao Luo, and Bin Bao. HOTL: A Higher Order Theory of Locality. In *Proceedings of the 18th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Houston, TX, March 2013.

[66] Haoyu Xiao, Ziqi Wei, Jiarun Dai, Bowen Li, Yuan Zhang, and Min Yang. HouseFuzz: Service-Aware Grey-Box Fuzzing for Vulnerability Detection in Linux-Based Firmware. In *Proceedings of the 46th IEEE Symposium on Security and Privacy (Oakland)*, Los Alamitos, CA, May 2025.

[67] Gala Yadgar, MOSHE Gabel, Shehbaz Jaffer, and Bianca Schroeder. SSD-based Workload Characteristics and Their Performance Implications. *ACM Trans. Storage*, 2021.

[68] Juncheng Yang, Ziming Mao, Yao Yue, and KV Rashmi. GL-Cache: Group-Level Learning for Efficient and High-Performance Caching. In *21st USENIX Conference on File and Storage Technologies (FAST) (FAST 23)*, Santa Clara, CA, February 2023.

[69] Juncheng Yang, Yao Yue, and K. V. Rashmi. A Large Scale Analysis of Hundreds of In-Memory Cache Clusters at Twitter. In *Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Virtual, October 2020.

[70] Juncheng Yang, Yao Yue, and KV Rashmi. A Large-Scale Analysis of Hundreds of in-Memory Key-Value Cache Clusters at Twitter. *ACM Transactions on Storage (TOS)*, 2021.

[71] Juncheng Yang, Yazhuo Zhang, Ziyue Qiu, Yao Yue, and Rashmi Vinayak. FIFO Queues Are All You Need for Cache Eviction. In *Proceedings of the 29th ACM Symposium on Operating Systems Principles (SOSP)*, Koblenz, Germany, October 2023.

[72] Yazhuo Zhang, Juncheng Yang, Yao Yue, Ymir Vigfusson, and KV Rashmi. SIEVE is Simpler than LRU: an Efficient Turn-Key Eviction Algorithm for Web Caches. In *Proceedings of the 21st USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, Santa Clara, CA, April 2024.

[73] Yu Zhang, Ping Huang, Ke Zhou, Hua Wang, Jianying Hu, Yongguang Ji, and Bin Cheng. Tencent Block Storage Traces (SNIA IOTTA Trace Set 27917). *SNIA IOTTA Trace Repository. Storage Networking Industry Association*, 2018.

[74] Yu Zhang, Ping Huang, Ke Zhou, Hua Wang, Jianying Hu, Yongguang Ji, and Bin Cheng. OSCA: An Online-Model Based Cache Allocation Scheme in Cloud Block Storage Systems. In *Proceedings of the 2020 USENIX Annual Technical Conference (ATC)*, Virtual, July 2020.

[75] Laiping Zhao, Yanan Yang, Yiming Li, Xian Zhou, and Keqiu Li. Understanding, Predicting and Scheduling Serverless Workloads under Partial Interference. In *Proceedings of the 2021 International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, St. Louis, MO, November 2021.

[76] Ke Zhou, Si Sun, Hua Wang, Ping Huang, Xubin He, Rui Lan, Wenyan Li, Wenji Liu, and Tianming Yang. Tencent Photo Cache Traces (SNIA IOTTA Trace Set 27476). *SNIA IOTTA Trace Repository. Storage Networking Industry Association*, 2016.

[77] Ke Zhou, Si Sun, Hua Wang, Ping Huang, Xubin He, Rui Lan, Wenyan Li, Wenjie Liu, and Tianming Yang. Demystifying Cache Policies For Photo Stores at Scale: A Tencent Case Study. In *Proceedings of the 2018 International Conference on Supercomputing (ICS)*, Beijing, China, June 2018.

[78] Yuanyuan Zhou, James Philbin, and Kai Li. The Multi-Queue Replacement Algorithm for Second Level Buffer Caches. In *Proceedings of the 2001 USENIX Annual Technical Conference (ATC)*, June 2001.

ARTIFACT EVALUATED
USENIX
AVAILABLE

Learning-Augmented Heuristics: Simple, yet Smart, Robust and Interpretable Cache Eviction

Haocheng Xia
Harvard University & UIUC

William Nixon
University of Chicago & Harvard University

Bintang Dwi Marthen
Harvard University & Institut Teknologi Bandung

Pranav Bhandari
Meta

Juncheng Yang
Harvard University

Abstract

Caching is widely used across the system stack to improve performance and efficiency, with eviction algorithms at its core. Existing cache eviction policies fall into two broad categories: static heuristics (e.g., 2Q, S3-FIFO) and smart algorithms (e.g., ARC, LRB). Smart caches can adapt to workloads and have the potential to achieve higher efficiency and robustness than static heuristics. However, we find that existing smart caches suffer from objective mismatches and instability.

We introduce Learning-Augmented Heuristics (LAH), a framework that learns the cache-level parameters of static heuristics. By decoupling the data and control planes, LAH supports simple, high-speed data reads and writes on the data plane, while performing occasional asynchronous learning on the control plane using cache-level features.

We demonstrate the effectiveness of LAH through S4-FIFO, a Smart S3-FIFO cache eviction algorithm. We pre-train a single model on 4,140 production traces and embed it in S4-FIFO to learn optimal cache parameters. On 1,035 evaluation traces, S4-FIFO improves the mean efficiency by 26% compared to S3-FIFO and by 8% compared to 3L-Cache, the best state-of-the-art algorithm. S4-FIFO is also robust—increasing miss ratio over FIFO by 0.8% on the worst trace, whereas 3L-Cache increases FIFO's miss ratio by 8.8%. Finally, S4-FIFO's decisions are also interpretable: a language model can provide a rationale for why a particular configuration was chosen.

1 Introduction

Caching underpins performance across modern systems—from storage stacks [11, 64, 67, 75] and databases [35, 57] to large web services [7,9,12,14,25,27,47,50,54,61,69,72]—by absorbing repeated access and reducing backend load. At the heart of every cache is an eviction algorithm that decides which objects to retain under tight memory budgets. Today's production systems overwhelmingly rely on static heuristics such as LRU, 2Q [35], S3-FIFO [74], and SIEVE [77] because they are simple, fast, and easy to deploy. In parallel, a growing body of work has proposed "smart" caches—adaptive and learned policies such as ARC [46], LeCaR [63], LRB [58], LHD [10], GL-Cache [70], and 3L-Cache [83]—that aim to tailor eviction decisions to each workload.

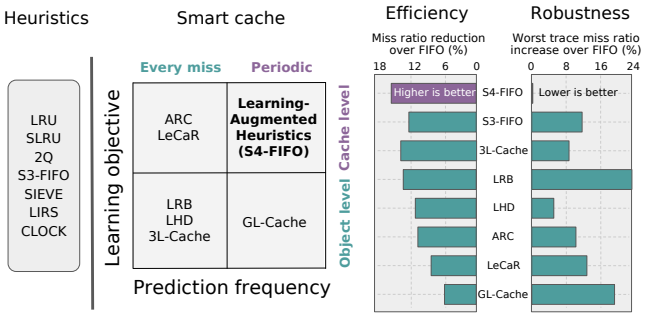


Figure 1: A classification of eviction algorithms (Section 2.2) indicates that periodic cache-level learning is highly effective, yet it has not been explored. We bridge this gap with learning-augmented heuristics, exemplified by a new algorithm, S4-FIFO, which achieves both high efficiency and strong robustness.

In principle, smart caches should dominate hand-crafted heuristics. In practice, very few of them (if any) have been adopted. Many of the smart caches learn at the granularity of individual objects and make predictions on *every miss*, optimizing surrogate objectives such as reuse-distance regression loss or per-object utility scores rather than the cache's true objective, the miss ratio. These object-level, per-miss learning schemes suffer from four recurring problems: (1) **objective mismatch**, where improvements in prediction metrics do not translate into fewer misses; (2) **instability and noise sensitivity**, as these schemes react to highly variable request-level signals; (3) **overheads**, due to per-object metadata, frequent inference on the critical path, and additional runtime complexity; and (4) **limited robustness and interpretability**, as eviction decisions are opaque, difficult to interpret, and may yield miss ratios higher than FIFO on some workloads.

At the same time, recent work shows that static heuristics can achieve surprisingly strong performance. S3-FIFO [74], for example, uses a small FIFO queue, a main FIFO queue,

and a metadata-only ghost queue with simple promotion rules to filter one-hit wonders and retain popular objects. While it is strong on the mean and median trace, our analysis reveals that its static configuration leaves substantial headroom for the tail traces.

We classify smart eviction algorithms along two axes—*learning granularity* (object vs. cache level) and *prediction frequency* (per-miss vs. periodic)—and observe that existing approaches occupy three of the four quadrants (Figure 1). The remaining quadrant, periodic cache-level learning, remains unexplored, despite offering an appealing opportunity: directly learning cache parameters that optimize miss ratio.

This observation motivates a fundamentally different approach to incorporating learning into caching: instead of learning a new, complex eviction policy, we propose learning to configure a simple, expressive heuristic. We introduce **L**earning-**A**ugmented **H**euristics (LAH), a framework that cleanly separates the cache's data and control planes. The data plane implements a deterministic heuristic with a small number of explicit knobs. The control plane runs asynchronously and infrequently, using cache-level features to select a configuration that directly targets miss-ratio reduction. A single model, pre-trained offline on a large corpus of traces, acts as a reusable "foundation model" for caches: it performs zero-shot prediction without per-deployment retraining.

We realize LAH in S4-FIFO, a "Smart S3-FIFO" eviction algorithm that augments S3-FIFO with an additional FIFO region and exposes its internal control knobs for learning. S4-FIFO discretizes the configuration space into a small set of representative parameter combinations and trains a gradient-boosted decision tree [18, 28] to choose among them using a cost-sensitive objective that explicitly anchors robustness to FIFO. Trained once on 4,140 production traces spanning block, key–value, and CDN caches, this model is embedded into the algorithm as lightweight, dependency-free code. When evaluated on 1,035 held-out production traces, S4-FIFO improves mean miss-ratio reduction by 26% over S3-FIFO and by 8% over 3L-Cache, the best prior algorithm, while never exceeding FIFO's miss ratio by more than 0.8% on the worst trace. For comparison, the next-best algorithm, 3L-Cache, increases FIFO's miss ratio by 8.8% in the worst case. Because of the separation of the control and data planes, S4-FIFO achieves throughput on par with other heuristics, such as LRU and 2Q. Because both its inputs and outputs are cache-level quantities with clear semantics, S4-FIFO's decisions are also relatively interpretable: a language model can reason about the factors behind a particular configuration choice.

This paper makes the following contributions:

- We introduce a cache classification and demonstrate that *periodic cache-level* learning is the most effective approach.
- We propose learning-augmented heuristics (LAH), a framework that augments static heuristics with parameters and uses a foundation model to learn them.
- We design and implement S4-FIFO, an example of LAH. We pre-train a gradient boosting tree model for S4-FIFO on 4,140 production traces and provide a dependency-free library for users [1].
- We evaluate S4-FIFO on 1,035 production traces and show that it is both more efficient and more robust than state-of-the-art algorithms, while matching the throughput of heuristics. Moreover, S4-FIFO's decisions are interpretable.

2 Background and Motivations

2.1 Evolution of Cache Eviction Algorithms

A central component of cache performance is the eviction algorithm, deciding which object(s) to remove when space runs out. Over the years, researchers and practitioners have designed a wide spectrum of eviction strategies, from simple heuristics to learning-based policies.

2.1.1 Heuristics-Based Algorithms

Most (if not all) production systems use heuristics-based eviction algorithms today. Traditional heuristics, such as 2Q [35] and LIRS [33], and modern heuristics, such as S3-FIFO [74] and SIEVE [77], operate based on *manually* curated rules that exploit recency and frequency patterns in a workload. These heuristics are valued for their simplicity, high throughput, and ease of implementation. However, they often fall short in efficiency compared to adaptive algorithms.

2.1.2 Smart Cache Eviction Algorithms

Adaptive algorithms. For example, ARC [46] maintains four LRU queues: two for data and two for ghost. It dynamically resizes recency and frequency data queues based on observed hits on the corresponding ghost queues. Similarly, DLIRS [41] extends LIRS by introducing an adaptive partition between low-reuse (HIR) and high-reuse (LIR) regions, which is dynamically resized based on its reuse distance. Although adaptive cache eviction algorithms do not employ machine learning models, we classify them as smart cache eviction algorithms because they can adapt to workloads by observing dynamic features.

Learning-based algorithms. More recent work builds prediction models to guide eviction decisions. Algorithms such as LRB [58], 3L-Cache [83], and GL-Cache [70] use supervised learning at the object level to estimate reuse distance and then evict the object predicted to have the least future value. Other approaches like LHD [10] model eviction using conditional hit-density, while LeCaR [63] adaptively reweights LFU and LRU weights with regret minimization. Although these complex policies may outperform heuristics, deploying them adds practical challenges, mainly: (1) online model training and inference add overheads, reducing cache throughput, and (2) their decision-making logic is opaque, from their features to their outputs. As a result, it is hard to interpret, tune, or debug why they behaved the way they did.

[1] https://libcachesim.com

Table 1: Classifying smart caches based on learning granularity and prediction frequency.

Learning granularity	Prediction frequency	Example	Problems
Object-level	Every miss	LRB [58], LHD [10], 3L-Cache [83]	Objective mismatch, unpredictability, overheads
	Periodic	GL-Cache [70]	Stale prediction
Cache-level	Every miss	ARC [46], LeCaR [63]	Instability, delayed reward
	Periodic	S4-FIFO (this work)	None of the above

Figure 2: Object-level learning has an objective mismatch issue. A smaller loss on object-level metrics, e.g., reuse distance, does not guarantee fewer cache misses.

2.2 Classifying Smart Eviction Algorithms

We classify existing smart cache eviction algorithms along two dimensions: the *granularity of learning* and the *frequency of prediction or adaptation* (Table 1).

Learning granularity. A smart cache eviction algorithm adapts to workloads by making predictions about the workload. These predictions can be either at the object level or the cache level. *Object-level algorithms*, such as 3L-Cache [83], LRB [58], and LHD [10] assign utility score or eviction probability to every individual object in the cache. For example, LRB and 3L-Cache learn the reuse distance for objects in the cache to predict the next access. Note that we also classify GL-Cache [70] as an object-level algorithm because it makes predictions for object groups similar to objects.

In contrast, *cache-level algorithms* learn by optimizing the internal cache parameters rather than evaluating individual objects. A classic example is ARC [46], which dynamically tunes the target sizes of its recency and frequency queues based on workload patterns, without assigning scores to cached objects. Similarly, LeCaR [63] dynamically adjusts the expert weights based on hits on the ghost queues.

Prediction frequency. Besides granularity, the frequency at which an algorithm predicts or adapts is also critical. Most prior algorithms (e.g., LRB, ARC, LeCaR) make a prediction *every miss*. For example, LRB samples 64 objects and predicts their reuse distance at each eviction; ARC updates the queue sizes upon each hit on the ghost queue (cache miss).

GL-Cache is the only algorithm that makes predictions *periodically* by ranking all object groups and evicting the top 10%. It does not trigger another prediction before all the top 10% object groups are evicted.

2.3 Learning Granularity: Object vs Cache

2.3.1 Object-level Learning

Several recent approaches operate at the granularity of individual objects, typically by predicting reuse distance or popularity. While this formulation appears natural, object-level learning faces inherent challenges regarding *objective alignment*, *predictability*, and *robustness and interpretability*.

Objective mismatch. The first challenge with object-level learning is objective mismatch, a phenomenon akin to Goodhart's Law: when a proxy becomes the target, it ceases to be a good measure. Many learned caches optimize for object-level metrics (e.g., minimizing L2 loss on reuse distance) rather than the actual miss ratio objective. For example, Figure 2 shows an example where model A achieves a lower L2 loss than model B, yet it fails to evict the correct object.

Unpredictability. Object-level metrics often cannot be predicted accurately. First, metrics, such as object reuse distance, are not intrinsic to an object and are heavily influenced by the workload, such as burstiness. Furthermore, cache workloads often exhibit a large number of one-hit wonders [74], for which no information (e.g., past N reuse distance, frequency) is available for learning. Some object-level metrics, such as popularity, are intrinsic properties of an object; however, they change quickly over time and cannot be used directly for eviction.

Robustness and interpretability. As object-level learning is driven by the unpredictable metrics described above, it becomes highly sensitive. Variations in workload can produce wrong eviction decisions. This volatility directly translates into poor robustness; models may perform well on average yet fail badly on certain workloads. Moreover, the outputs of object-level learning are fundamentally opaque. The model assigns per-object scores with little to no operational semantics, leaving operators unable to explain when the miss ratio increases. This combination of unpredictable inputs and opaque outputs makes object-level learning risky to deploy at scale.

2.3.2 Cache-level Learning

Cache-level learning avoids the aforementioned problems by tuning a small set of global cache parameters. For example,

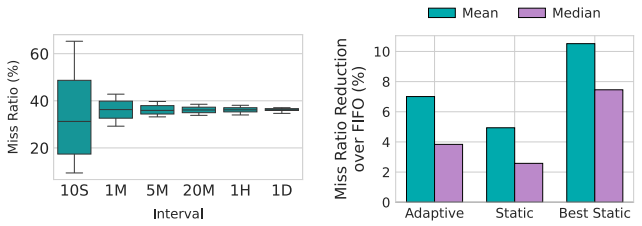


Figure 3: (a) Cache metrics, such as miss ratio, exhibit greater variance at finer time granularity. (b) Choosing the correct parameters can outperform LeCaR's per-miss adaptivity.

ARC [46] maintains two data queues and two ghost queues. Data queues store data, while ghost queues only keep metadata of recently evicted objects. ARC dynamically adjusts the queue size based on ghost hits. Intuitively, the more hits on the ghost queue, the more space is needed for the corresponding data queue. As another example, LeCaR [63] uses two eviction algorithms as experts and randomly chooses one expert to pick an eviction candidate based on its weight. The expert (algorithm) with a larger weight is more likely to be used for eviction. Similar to ARC, LeCaR adjusts experts' weights based on ghost hits using reinforcement learning.

Direct optimization. *Cache-level learning often directly optimizes for the miss ratio*. For example, a ghost hit means that if the data queue is larger, it will be a cache hit, so increasing the data queue size directly reduces misses. Similarly, if an expert has many ghost hits in LeCaR, indicating it is not suitable for the workloads, then reducing its weight would reduce the number of wrong evictions. Direct optimization not only helps reduce the miss ratio but also improves interpretability, enabling operators to reason about specific actions.

Magic parameters. Although cache-level learning is intuitive and avoids the objective-mismatch problem, it often has magic parameters. For example, how much and how often the parameters should be updated. Previous work has shown that the default move *one* slot *per ghost hit* approach in ARC has pathological failure cases [74].

2.4 Prediction Frequency: Per-miss vs Periodic

While the learning objective is critical for the effectiveness of a smart cache, how often a cache makes a prediction or adaptation is equally important.

2.4.1 Per-miss Prediction and Adaptation

Most algorithms make predictions at the request granularity, e.g., every miss. For example, LRB estimates object reuse distance on each cache miss, while ARC and LeCaR update queue sizes or weights on every hit to their ghost queues. Although such per-miss adaptation can appear highly responsive, it suffers from three key limitations: *instability and noise overfitting*, *delayed reward*, and *inference overhead*.

Instability and overfitting to noise. Observations on each request and object might not reflect the overall patterns in the workload. By reacting to every miss, the algorithms will instead "chase" this noise. We find that real-world workloads often exhibit significant variance at fine time granularity, and request-level signals are often very noisy. Figure 3 left shows the miss ratio measured at different time granularities on a CloudPhysics trace. We observe that at smaller time intervals, the miss ratio over time tends to have a very large variance.

We further measured the impact of per-miss adaptivity using 106 traces from the CloudPhysics dataset. Figure 3 right shows that, although per-miss adaptivity improves over using a single static parameter for *all* traces (static), it is significantly worse than choosing the best static parameter for each trace. This suggests that choosing the right parameter is much more important than per-miss adaptivity.

Delayed Reward for Cache-level Learning. Besides overfitting, per-miss learning also suffers from the delayed reward problem. Because a cache miss may happen *long* after an object is evicted, and parameter changes take time to become effective, we find that the change often lags behind the need. This lag makes the feedback signal slow and ambiguous.

Overheads for Object-level Learning. When pairing per-miss prediction with object-level learning, another drawback is the overhead. Performing a model inference on each miss significantly reduces the throughput. Besides, object-level learning also incurs significant per-object metadata storage overhead. For example, LRB adds over 200 bytes of features to each object, which eats into DRAM capacity.

2.4.2 Periodic Prediction

To the best of our knowledge, GL-Cache [70] is the only cache eviction algorithm that uses periodic prediction. Similar to other object-level learned caches, GL-Cache predicts a score for each object group. However, instead of predicting on the critical path, GL-Cache makes one prediction *periodically* and reuses the scores until 10% of the object groups are evicted. However, because the usefulness of objects and groups changes over time, their predictions quickly become stale, and periodic prediction often leads to lower efficiency when used with object-level learning.

Our analysis of existing algorithms reveals a dichotomy in the design space. Object-level algorithms suffer from objective mismatch, unpredictability, and huge overheads. Moreover, per-miss learning algorithms are prone to instability and delayed rewards. Table 1 highlights a distinct gap in the design space—we need smart cache eviction algorithms that *learn and adapt at the cache level, with the learning objective aligned with miss ratio reduction*; meanwhile, we need *periodic learning* to avoid unnecessary overfitting to noise and magic parameters.

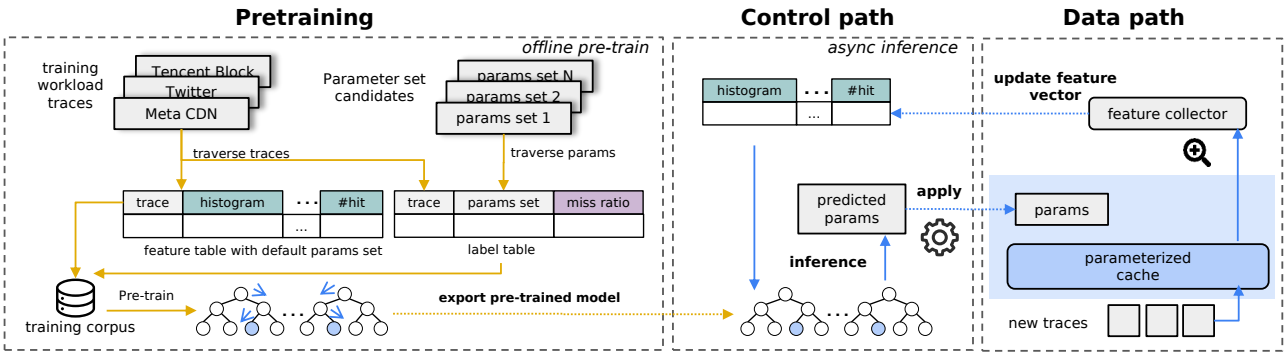


Figure 4: Illustration of learning-augmented heuristics. (1) Offline pretraining (left): A foundation model is trained on a massive corpus of trace features and optimal parameter labels to learn universal caching rules. (2) Online operation (right): The system decouples the critical *data path* from the *control path*. The data path collects lightweight workload-level features, while the control path asynchronously queries the pre-trained model to adapt the cache parameters to the current workload trace.

3 Learning-Augmented Heuristics (LAH)

In this section, we introduce a *learning-to-configure* paradigm that fundamentally rethinks how machine learning interacts with caching systems. Instead of embedding complex learning logic (e.g., neural networks) directly into the critical path, we propose a clean separation of concerns: *the cache data path executes a simple, parameterized heuristic, while the control plane utilizes a pre-trained "foundation model" to choose the heuristic's parameters*, an approach we call Learning-Augmented Heuristics (*LAH*).

Existing learning-based caches (e.g., GL-Cache, LeCaR) often treat learning as an online, instance-specific task and train from scratch for each new or even the same workload. This approach suffers from "catastrophic forgetting" and high retraining costs. In contrast, our approach builds upon the well-established insight that cache access patterns are governed by universal characteristics: phenomena like scanning, looping, and thrashing are structural behaviors that transcend specific datasets [21, 33]. By extracting *content-oblivious* features rather than tracking specific object IDs, we can train a single global model that generalizes across diverse workloads. This effectively builds a foundation model for caching: knowledge learned from thousands of offline traces is preserved and transferred to new, unseen environments. The model does not need to relearn but recognizes the pattern and applies the optimal configurations. This decoupled architecture offers five critical advantages over tightly coupled learned caches.

Simplicity. The data path logic remains simple and deterministic. It requires almost no metadata overhead (unlike LRB or 3L-Cache, which track extensive ghost histories per object) and avoids complex data structures. This "stateless" nature drastically reduces code complexity and bug surface area, making the system easier to verify and deploy in production.

Performance. By offloading the inference to an asynchronous control plane, the critical path for GET and PUT operations remains extremely lightweight, ensuring high throughput and low tail latency.

Efficiency. Simple static heuristics, such as 2Q, cannot achieve high efficiency compared to state-of-the-art algorithms. LAH can automatically optimize for different workloads. During the cache warm-up period, LAH identifies the "golden configurations" and updates its parameters to match the workload's patterns.

Robustness. When evaluating state-of-the-art eviction algorithms on thousands of production workloads, we find that most of them exhibit poor robustness—some traces exhibit a miss ratio significantly higher than a simple baseline such as FIFO. This hinders adoption because users do not know whether these algorithms will work on their workloads. LAH learns the configuration from high-level workload features, which are less susceptible to perturbations and noise. Moreover, LAH selects parameters from a validated set of safe configurations. These enable LAH to achieve robustness.

Interpretability. LAH operates on aggregated metrics such as hit distributions, which align naturally with how operators already think about and understand cache behavior. These high-level signals are then mapped to a small set of heuristic parameters, so each prediction corresponds to a knob with a clear operational meaning and existing tuning intuition. Because both the inputs and outputs live in this shared, cache-centric space, the resulting configurations are much more interpretable than existing learned caches.

While LAH can be designed on top of different parameterized algorithms, we use S3-FIFO [74] as an example because it is simple, performant, and scalable. In the next section, we describe S4-FIFO, a new cache eviction algorithm that is Simple, Scalable, Smart, and only uses Static FIFO queues. S4-FIFO exposes and augments S3-FIFO's internal parameters to our foundation model for learning, transforming a static heuristic into an efficient, robust, and interpretable learning-augmented heuristic.

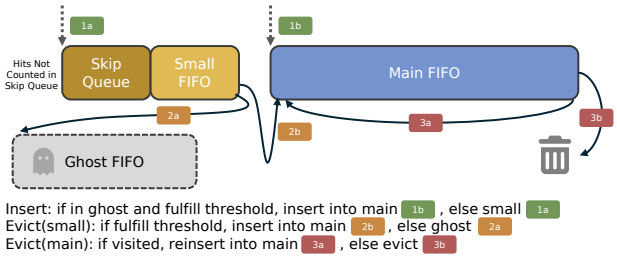


Figure 5: An illustration of the data path in S4-FIFO.

4 S4-FIFO Design and Implementation

In this section, we describe the design of S4-FIFO, detailing the heuristic data path, feature engineering, configuration quantization, and model training.

4.1 Overview

We illustrate the high-level overview as shown in Figure 4.

Heuristic Augmentation. We first augment S3-FIFO with a burst-aware parameter that virtually partitions the small queue into an additional FIFO queue, giving the parameterized heuristic greater flexibility to minimize the miss ratio.

Dataset Preparation. We randomly split the 5,175 traces used in this study into training and test sets to avoid data leakage. Then we perform a grid search to determine the best parameters (labels for training) for each trace. Next, we collect cache and workload-level features by running S4-FIFO using the default parameter on each trace, yielding the samples used to train the model.

Model Pretraining. We use features and labels from the previous step to train a classification model that predicts the best parameters for an unseen workload.

4.2 Augmenting the Parameters

Parameter prediction and request serving. The previous steps only need to be performed once offline. Once the model is trained, it will be used in different settings without retraining. When a cache starts, S4-FIFO collects features online and then asynchronously makes a single prediction. Then S4-FIFO switches from the default parameters to the predicted parameters. Queue resizing is lazy: when the predicted configuration reduces a queue's target size, S4-FIFO does not immediately move or evict objects. Instead, future evictions preferentially come from queues that exceed their new limits until the queue sizes converge. This allows parameter updates without pausing the cache or adding extra work to the request critical path. For simplicity, our evaluation makes a single prediction per trace. We found that substantial shifts requiring a new prediction are uncommon. In practice, S4-FIFO could be made to periodically re-predict its parameters if needed, for example on a weekly basis. We leave the choice of refresh policy and interval outside the scope of this work.

Table 2: Hyperparameters of the S4-FIFO.

Symbol	Physical Meaning	Default	Learning range
ρ_S	Small queue size ratio	0.10	$[0.05, 0.90]$
ρ_G	Ghost queue size ratio	0.90	$[0.90, 6.00]$
κ	Small skip ratio	0.00	$\{0.00, 0.25\}$
τ_S	Small-to-Main promotion threshold	2	$\{1, 2\}$
τ_G	Ghost-to-Main promotion threshold	0	$\{0, 1\}$

S3-FIFO maintains three FIFO queues. A small queue, occupying 10% of the cache, filters out one-hit wonders; a main queue retains popular objects via reinsertion; and a metadata-only ghost queue identifies moderately popular objects that were mistakenly evicted from the small queue. Each cached object is associated with a 2-bit saturated frequency counter that is incremented on every cache hit, and a threshold on this counter governs movement between the queues. S4-FIFO preserves the same three physical FIFO queues, but parameterizes their behavior, as shown in Figure 5. The parameters are shown in Table 2.

Queue size. While S3-FIFO sets the small queue at 10% of the cache, S4-FIFO treats the queue sizes as tunable parameters. This allows the system to select the optimal ratio between the small and protected regions based on workload characteristics. In addition, S4-FIFO also makes the ghost queue size tunable to collect more information about the workload.

Promotion threshold. S3-FIFO moves an object evicted from the small queue to the main queue if the object's frequency is greater than 1. S4-FIFO makes the frequency threshold learnable, allowing control over how strictly the small queue filters new insertions. Similarly, to handle cyclic scan patterns [33, 51], S4-FIFO learns the frequency threshold for promoting an object from the ghost queue to the main queue.

Skip ratio. To mitigate correlated references (bursty "one-hit wonders") [35, 76], S4-FIFO does not increment the frequency counter for objects in the first κ fraction of the small queue. This creates a virtual probationary region inside the small queue, adding an extra filtering stage without introducing a fourth physical FIFO queue.

Parameter discretization. The parameter space of S4-FIFO is continuous and high-dimensional, and its exponentially growing combinations make it impossible to perform a grid search to find the best parameters. Therefore, we discretize the parameter ranges and choose a few values for each based on domain knowledge. Specifically, the small queue size can take up 5%, 10%, 20%, 30%, 50%, 70%, or 90% of cache capacity; the ghost queue can store the same number of objects, $3\times$ or $6\times$ that of the main queue; the small-to-main promotion threshold can be either 1 or 2; the ghost-to-main threshold can be either 0 or 1; and the skip ratio can be either 0 or 0.25.

Table 3: Features used by S4-FIFO.

Features	Dimension	Meaning
Small histogram	20	Hit-position distribution ratio in $\mathcal{S}$
Main histogram	20	Hit-position distribution ratio in $\mathcal{M}$
Ghost histogram	20	Hit-position distribution ratio in $\mathcal{G}$
Small hits ratio	1	Proportion of hits happening in $\mathcal{S}$
Main hits ratio	1	Proportion of hits happening in $\mathcal{M}$
Ghost hits ratio	1	Proportion of hits happening in $\mathcal{G}$
Log cache size	1	log(cache size)
Utility gap	1	Utility divergence between $\mathcal{S}$ and $\mathcal{M}$
Filtering efficiency	1	Relative hit density of $\mathcal{S}$ vs. $\mathcal{M}$
Ghost pressure	1	Intensity of miss-on-eviction events
Tail heaviness	1	Hit concentration at the deep end of $\mathcal{M}$
Decay rate	1	Recency decay slope at the head of $\mathcal{S}$
One-hit ratio	1	Estimated fraction of one-hit wonders
Unique ratio	1	Workload churn rate (unique objects ratio)
Scan intensity	1	Effective scan pressure under capacity limits
Thrashing risk	1	Signal for working set exceeding capacity

4.3 Learning the Parameters

4.3.1 Learning Objective

To enable efficient learning, we formulate the parameter learning as a probabilistic classification task over a finite set of candidate parameter configurations $\mathcal{C}$. We learn a conditional probability distribution $P(y = c_j|\mathbf{x};\phi)$ over these candidates $c_j \in \mathcal{C}$ given the workload features $\mathbf{x}$.

Standard classification objectives (e.g., cross-entropy) treat all misclassifications symmetrically. However, in caching, error costs are highly *asymmetric and pairwise*: mistaking a scan-heavy workload for a recency-friendly one causes catastrophic thrashing, whereas the reverse error often yields only marginal degradation. To capture these varying penalties, we employ a data-driven cost matrix $\mathbf{L} \in \mathbb{R}^{K\times K}$, where K is the number of candidate parameter configurations. Each entry L_{kj} quantifies the pairwise "regret" of selecting configuration c_k when the true optimal is c_j, normalized by the miss ratio of a baseline anchor $\mathcal{M}_{\text{anchor}}$,

$$L_{kj} = \mathbb{E}_{\text{train}}\left[\frac{\text{MR}(c_k) - \text{MR}(c_j)}{\mathcal{M}_{\text{anchor}}}\right].$$

Choice of Anchor. While the anchor can theoretically be any baseline (e.g., LRU, FIFO), we select $\mathcal{M}_{\text{anchor}} = \text{MR}_{\text{FIFO}}$. This choice is motivated by the structural nature of our system: (1) S4-FIFO is fundamentally built upon FIFO queues, therefore FIFO serves as the natural lower bound for robustness, and (2) normalizing by FIFO makes the cost metric invariant to the cacheability of the trace, allowing the model to learn effectively across workloads with vastly different miss ratios.

During inference, we minimize the expected risk,

$$\hat{c} = \arg\min_{c_k\in\mathcal{C}} \sum_j P_\phi(c_j|\mathbf{x}) \cdot L_{kj}.$$

4.3.2 Features

To enable lightweight yet effective learning, we model the cache states and workload characteristics s_t using a compact feature vector (73 dimensions). The list of features used can be found in Table 3, where $\mathcal{S}$, $\mathcal{M}$, and $\mathcal{G}$ are the small queue, the main queue, and the ghost queue, respectively.

Cache metrics and workload characteristics. Cache size is critical for scale-invariant learning: the effectiveness of the small queue depends on its *absolute* capacity to absorb bursts, not just its relative share of the cache. For example, a 10% partition in a 100 GB cache provides ample buffer space, whereas 10% of a 100 MB cache may be insufficient for the same workload. Therefore, we track cache size using the log-transformed capacity. To capture the "shape" of locality beyond simple scalar averages, we maintain lightweight histograms (20 equally-sized bins each) for the small, main, and ghost queues. These histograms capture the distribution of hit position within each queue. For instance, the first bin stores the number of hits in the first 5% of the queue. Intuitively, the histograms enable the cache to learn the queue usage. For example, many ghost hits suggest the small queue might be too small. Besides the histogram, we also include the number of hits per queue and the total number of requests.

Composite features. To avoid complex models, we use feature crossing to capture non-linear relationships between features, following common practice in traditional recommendation systems that rely on simple models [19, 52].

- *Utility gap* measures the utility divergence between the small and main queues and is defined as the absolute hit difference between them divided by total hits, $\frac{H_{\text{main}} - H_{\text{small}}}{H_{totql}}$. A positive gap indicates that the main queue is effectively retaining hotter items than the small queue.
- *Filtering efficiency* quantifies how well the small queue filters one-hit, computed as the ratio of hits in the small queue to those in the main queue, $\frac{H_{\text{small}}}{H_{\text{main}}}$.
- *Ghost pressure* measures the fraction of hits served by the ghost queue, $\frac{H_{\text{ghost}}}{H_{\text{ghost}}+H_{\text{total}}}$. A high ghost pressure suggests that the small queue is undersized or that the promotion threshold is too conservative.
- *Tail heaviness* measures the dependence of the workload on long-term retention and is computed as the sum of hits in the last 10 bins of the main queue's histogram, with larger values indicating that a substantial fraction of hits comes from long-resident objects.
- *Decay rate* measures how quickly items lose utility in the probationary small queue and is computed as the slope between the first two bins of the small queue's histogram, where a steep negative slope indicates that most objects become cold quickly.
- *Unique object ratio* estimates cacheability and is computed as $\frac{\#\text{ insertions to small}}{\#\text{ requests}}$.
- *One-hit ratio* estimates the fraction of one-hit wonders in the workload and is computed as $\rho_{\text{onehit}} = \frac{N_{\text{onehit}}}{N_{\text{unique}}}$, where N_{onehit} is the number of objects evicted from the small queue without reaching the promotion threshold, and N_{unique} is the number of unique objects observed during the window.
- *Scan intensity* and *thrashing risk* are derived by normalizing *unique object ratio* and *one-hit ratio* by the request rate, providing the model with explicit signals to distinguish

between benign scans and destructive thrashing.

4.3.3 Model and learning

Inference Model. We employ a Gradient Boosting Decision Tree ensemble as the predictive model ϕ. It accepts the 73-dimensional feature vector $\mathbf{x}$ and outputs a probability distribution over the 18 classes. We choose GBDT over deep neural networks (DNNs) for two reasons. First, it can be easily packaged into a few portable headers for distribution, allowing users to use the pre-trained model in other programming languages without installing any new libraries. For example, our Cachelib [11] prototype employs the same model we trained using the libCacheSim [1] simulator. Second, GBDT excels at handling heterogeneous, nonlinear features (e.g., histograms mixed with scalar ratios) and needs no complex normalization.

Parameter search space reduction. Although we have discretized each parameter into a few options, the number of possible parameter combinations remains huge, which reduces learning effectiveness. We employ a clustering-based strategy to reduce the search space. We use the offline grid search results to identify the most common parameter combinations and identify 18 representative parameter sets out of 168 candidates via greedy set cover algorithm [55].

4.4 Implementation

We implement S4-FIFO in both libCacheSim for simulation-based evaluation and Meta Cachelib for prototype evaluation and production deployment. Both implementations extend S3-FIFO and employ the same pre-trained model, trained on features and labels collected in the simulator. In each system, a hash table is used for indexing, and linked lists are used to implement the queues.

4.4.1 Model implementation

We use LightGBM for our model implementation in the simulator. Because tree-based models can be compiled into sequences of conditional branches, we export the pre-trained model into simple, *dependency-free* libraries using m2cgen [2], targeting several common programming languages, including C/C++, Go, Rust, Java, and JavaScript. This enables users to use our model directly without relying on LightGBM or other machine learning libraries. For example, our Cachelib implementation uses the C++ header files.

4.4.2 Feature collection

Most of the features are simple counters and straightforward to maintain, so we focus on the histogram features.

Small and Main FIFO queues. We use insertion time (i.e., the insertion index) to calculate the histogram bin for each object as $\lceil \frac{T_{obj}-T_{tail}}{w} \rceil$ where T denotes the insertion time and w is the bin width. For example, suppose the queue has a capacity of 1000, and we divide it into 20 bins, each with a width of 50. If the tail object was inserted at time $T_{tail} = 20$ and the current object at $T_{obj} = 80$, then the object should be incrementing the counter at the $\lceil \frac{80-20}{50} \rceil = 2^{nd}$ bin.

Ghost queue. Computing histogram bin indices in the ghost queue is more involved because a hit in the ghost queue removes the corresponding metadata entry and shifts all subsequent entries forward. As a result, using insertion time alone to determine the bin index yields incorrect positions. To compensate, we maintain an additional *deletion-count* histogram alongside the hit-position histogram for the ghost queue. Concretely, let i be the bin index obtained from the insertion-time-based formula. We sum the deletions recorded in bins 1 through $i-1$, divide this sum by the bin width w to calculate the number of shifted bins s, and then correct the bin index as $i' = i - s$. This adjusted index is used as the hit-position bin in the ghost-queue histogram. Even with the adjustment, the calculated bin is still approximate because we do not consider the deleted objects in bin i. However, the approximation has bounded error and is at most off by 1.

4.4.3 Overhead Analysis

We analyze the runtime complexity of S4-FIFO to demonstrate that it incurs almost no overhead compared to heuristics.

Inference. All cache operations, i.e., read, write (including eviction), are $O(1)$. S4-FIFO adds two components. On the data path, it needs to collect features. However, many of the features it collects, such as the number of requests and hits, are already tracked in most production caches. The primary additional overhead introduced by S4-FIFO is the hit-position distribution histogram. This histogram is maintained as an array of counters, with exactly one counter incremented per access, yielding a strict $O(1)$ time complexity. In Section 5.4, we show that the feature collection has a negligible impact on throughput. In deployments that are very sensitive to computational overhead, the feature collection can use sampled requests. On the control path, S4-FIFO performs inference, which has a time complexity of $O(T \cdot D)$, where T is the number of trees (20 in our model) and D is the maximum depth (9 in our model). Since the inference happens rarely and asynchronously, and each inference call takes less than 2 ms, there is no impact on request serving.

Training. The training cost of S4-FIFO is entirely offline and consists of two stages: label generation and model fitting. Label generation is the dominant cost. For each training trace

and cache-size ratio, we run S4-FIFO over a fixed grid of candidate configurations and select the configuration with the lowest miss ratio as the label. Given N training traces, R cache-size ratios, G candidate configurations, and L requests per trace, label generation costs $O(N \cdot R \cdot G \cdot L)$. In our evaluation, G is a small constant determined by the parameter grid. Moreover, the grid search is embarrassingly parallel across traces, cache sizes, and configurations. Model fitting is substantially cheaper. Once the labels have been generated, we train a small GBDT using one compact feature vector per trace/cache-size pair. The model contains 20 trees with a maximum depth of 9 and predicts over 18 representative parameter sets. Thus, retraining is a one-time offline cost and does not affect the request-serving path. Deployments can directly use the pre-trained model. Retraining is only needed when operators want to incorporate additional training data. Further details can be found in Appendix Section A.1.

Storage. Unlike object-level learning, which requires maintaining feature vectors per object, S4-FIFO uses only 73 global features in total. Beyond the 2-bit counter inherited from S3-FIFO, no additional per-object metadata is needed. Including model parameters, the total storage overhead introduced by learning is on the order of only tens of kilobytes. The dominant storage overhead in S4-FIFO arises from the ghost queue. Each ghost entry requires 8 bytes to record the object ID (or fingerprint) and its insertion timestamp. For a cache holding 1 million objects, the ghost queue can therefore consume several megabytes of memory. Nonetheless, as long as objects are relatively large (e.g., larger than 200 bytes), this overhead remains small compared to the total data volume.

5 Evaluation

In this section, we use S4-FIFO as a case study to answer the following questions:

- Do augmented heuristics expose sufficient headroom for efficiency improvement?
- Can we accurately learn the optimal parameters of augmented heuristics to realize these efficiency gains?
- Do learning-augmented heuristics improve robustness?
- Do learning-augmented heuristics reduce throughput?
- Is a single pre-trained model sufficient across workloads, and why?

5.1 Methodology

Workloads. We evaluate S4-FIFO using 5,175 production traces collected from 14 sources (Table 4). We exclude very small traces with fewer than 100,000 objects, as these typically correspond to low-priority workloads (e.g., development traffic). Notably, many learned caches, such as LRB and 3L-Cache, perform extremely poorly on these short traces, which skews both average efficiency and robustness metrics. In contrast, S4-FIFO is insensitive to trace length because it relies on aggregated cache-level features. Consequently, including all traces would make S4-FIFO's results even stronger.

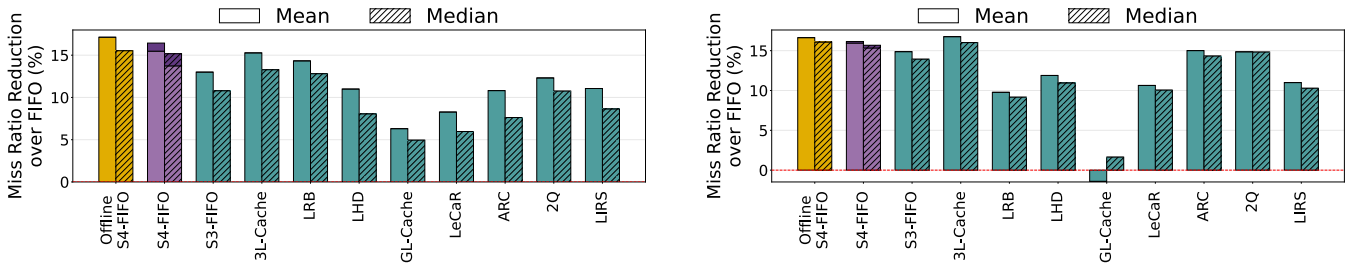
Table 4: Summary of datasets used in our evaluation. We filtered out traces with fewer than 100,000 objects.

Dataset	Approx time	Cache type	# Traces
MSR [48, 49]	2007	Block	14
FIU [37]	2008-11	Block	7
CloudPhysics [64]	2015	Block	104
Systor [39, 40]	2017	Block	6
CDN 1	2018	Object	163
Tencent Photo [81, 82]	2018	Object	2
WikiMedia CDN [66]	2016-19	Object	4
Tencent CBS [78, 79]	2020	Block	3370
Alibaba [4, 43, 65]	2020	Block	552
Twitter [73]	2020	KV	50
CDN 2	2021	Object	890
Meta Storage [15]	2022	Block	5
Meta KV [15]	2022	KV	5
Meta CDN [15]	2023	Object	3

Experiment setup. We randomly split the traces into two sets: 4140 for training and 1035 for evaluation. For each trace in the training set, we run S4-FIFO with the default parameter combinations to collect training samples of (features, label) pairs. For each trace in the test set, we start the cache using S4-FIFO with the default parameter combinations, run for 20% of the trace to collect features, then make one prediction and apply the predicted parameters to the rest of the trace. To ensure a fair comparison with the offline optimum obtained via grid search, we restrict ourselves to one prediction per trace, since the offline optimum also uses a single parameter setting for the entire trace.

We compare S4-FIFO with S3-FIFO [74], LIRS [33], ARC [46], LRB [58], 3L-Cache [83], LHD [10], GL-Cache [70], and LeCaR [63]. Besides, we have also evaluated SIEVE [77], GDSF [20], and TinyLFU [24]; however, their results are worse than the ones in the figure. Due to space limits and for ease of plotting, we omit them in the figure. We evaluate each algorithm at three cache sizes—0.1%, 1%, and 10% of the trace's working-set size—but, for space reasons, we only present results for the smallest (0.1%) and largest (10%) caches. The results for the medium size (1%) consistently fall between these two.

All miss ratio results are from libCacheSim because most state-of-the-art eviction algorithms are not available in open-source caches such as Cachelib. Because our Cachelib implementation uses the same model as the simulator, S4-FIFO shows almost identical miss ratios in the simulator and prototype. We report throughput results by comparing S4-FIFO with highly optimized LRU, 2Q, S3-FIFO, and TinyLFU us-



(a) Large cache (10% of working set)

(b) Small cache (0.1% of working set)

Figure 6: Mean and median miss ratio reduction from FIFO of different algorithms using the large and small cache sizes. S4-FIFO is significantly better than all state-of-the-art algorithms at the large cache size, while on par with the best of all algorithms at the small cache size.

ing CacheBench from Meta. The experiments were performed on a 20-node cluster with 32 cores and 192GB of memory. All experiments are repeated three times, and we report the mean result.

Metrics. We evaluate using the following metrics.
(i) *Efficiency*: Following prior work [74], we report the miss ratio reduction over FIFO, computed as $\frac{MR_{\text{FIFO}} - MR_{\text{algo}}}{MR_{\text{FIFO}}}$. We present the mean reduction in miss ratio across the 1,035 traces in the test set.
(ii) *Robustness*: We report miss ratio reduction for the worst-case workload and for the 10^{th}-percentile workload.
(iii) *Overhead*: We use Cachelib throughput as our primary overhead metric.

5.2 Efficiency

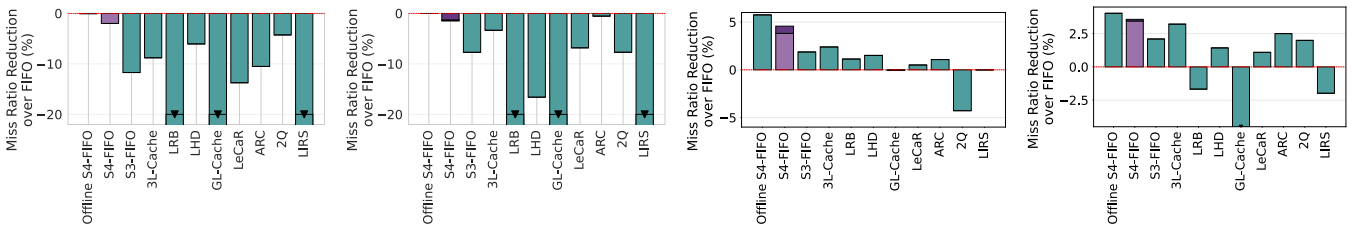
Efficiency is the primary performance metric for cache eviction. Figure 6 reports the mean and median miss-ratio reduction over FIFO for both small and large caches. Since S4-FIFO uses the first 20% of each trace to collect features before making a prediction, we evaluate two variants. In the online variant (v1, light purple), the first 20% of requests are served using the default parameters, and the remaining 80% are served using the predicted parameters. In the retrospective variant (v2, dark purple), the predicted parameters are applied to the entire trace; this isolates the quality of the learned configuration from the cost of the observation window. Across both cache sizes, v2 consistently outperforms v1, indicating that the predicted parameters improve efficiency and that the observation window accounts for the gap between the two variants.

Offline S4-FIFO. Unlike many learned caches, such as 3L-Cache, LRB, and LHD, which use sampling to select eviction candidates and can evict any object in the cache, learning-augmented heuristics like S4-FIFO are constrained by their underlying data structures: objects must pass through at least one queue before they can be evicted. Nevertheless, Figure 6 shows that offline S4-FIFO, which selects parameters via grid search, consistently achieves the highest efficiency. This suggests that the data structures used in S4-FIFO do not prevent it from attaining high efficiency. When comparing S4-FIFO with offline S4-FIFO, we find that the second variant, which runs the full trace using predicted parameters (same as offline S4-FIFO), achieves very similar miss ratio reduction as offline S4-FIFO with at most 0.2% difference in mean and median miss ratio reduction. This demonstrates the effectiveness of learning in S4-FIFO. In Section 5.5, we will show that S4-FIFO learns the best parameters most of the time, and even when it does not, the learned parameters are close to the best.

Comparison with state-of-the-art algorithms. S4-FIFO is built on top of S3-FIFO; compared to S3-FIFO, it achieves a 26% higher miss ratio reduction at the large cache size and an 8% higher reduction at the small cache size. Across all state-of-the-art algorithms, 3L-Cache consistently delivers the best performance. Relative to 3L-Cache, S4-FIFO attains slightly higher or comparable miss ratio reductions. For instance, at the large cache size, S4-FIFO achieves an 8% higher reduction. Compared with ARC, S4-FIFO nearly doubles ARC's improvement over FIFO at the large cache size. Relative to the large-cache setting, S4-FIFO's performance at the small cache size is less impressive: it is slightly worse than 3L-Cache, though still substantially better than the other state-of-the-art algorithms. For instance, S4-FIFO improves LRB's mean reduction ratio from 9.8% to 16.2%. The slight advantage of 3L-Cache at small cache sizes stems from the fact that the cached objects are all highly popular, providing rich signals for object-level learning. However, object-level learning incurs high overhead and low throughput. In our simulator, 3L-Cache is 17.3× slower than S4-FIFO on average (max 274×). In summary, S4-FIFO achieves a high efficiency via its superior learning methods and the pre-trained model.

5.3 Robustness

Historically, work on designing new cache eviction algorithms has focused on reducing miss ratio on a small set of traces or improving average performance. Far less attention has been given to the robustness of these algorithms. Yet robust-



(a) Worst case, large cache **(b)** Worst case, small cache **(c)** 10^{th}-percentile, large cache **(d)** 10^{th}-percentile, small cache

Figure 7: Efficiency on the worst case and 10^{th}-percentile trace. A negative value indicates the algorithm has a miss ratio higher than FIFO. S4-FIFO is more robust than all state-of-the-art algorithms.

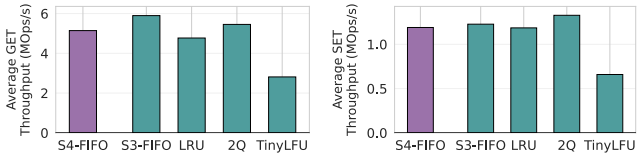


Figure 8: Set and Get throughput evaluated in Cachelib.

ness is crucial, because an algorithm can severely degrade performance when the workload is out of distribution relative to its design assumptions. In this section, we evaluate the robustness of state-of-the-art algorithms using the worst trace and 10^{th}-percentile trace. Note that we do not consider the competitive ratio because the arbitrary workload used in competitive-ratio analysis rarely occurs in the real world, so we focus on the worst-case production trace.

Figure 7a and Figure 7b show the miss ratio change relative to FIFO on the worst trace for each algorithm. We observe that all state-of-the-art eviction algorithms incur a higher miss ratio than FIFO on their worst-case trace. In particular, LRB and LIRS perform substantially worse than the others, increasing FIFO's miss ratio by 20% to 72%. In contrast, S4-FIFO increases FIFO's miss ratio by only 0.8% and 0.2% at large and small cache sizes, respectively. For comparison, the next-best algorithm at the large cache size is 2Q, which raises FIFO's miss ratio by 4.3%.

Although the single worst-case trace may be unrepresentative, Figure 7c and Figure 7d show results on the 10th-percentile trace. On this metric, S4-FIFO widens its lead over state-of-the-art algorithms. While some schemes, such as LRB and LIRS, still exhibit miss ratios higher than FIFO, S4-FIFO reduces FIFO's miss ratio by 4.2% and 3.6% at large and small cache sizes, respectively.

The robustness of S4-FIFO arises from three core design choices. First, S4-FIFO employs static FIFO queues and therefore avoids the pathological behaviors that can arise in adaptive algorithms [74]. Second, S4-FIFO leverages learning to select its parameters, enabling it to sidestep the adversarial workloads that often degrade the performance of fixed, heuristic-based schemes. Third, S4-FIFO's learning objective uses FIFO as an anchor to directly optimize for robustness.

5.4 Throughput

To assess whether S4-FIFO preserves the high throughput of lightweight heuristic policies, we measure its throughput using CacheBench and compare it against highly optimized Cachelib implementations of S3-FIFO, LRU, LRU2Q (a modified 2Q without a ghost queue), and TinyLFU. In this experiment, S4-FIFO collects features continuously for every request throughout the benchmark. This deliberately conservative configuration provides an upper bound on the overhead of feature collection. In practice, periodic or sampled feature collection could further reduce this overhead; exploring these optimizations is left to future work. We use CacheBench's CDN workload and four Graph Cache workloads, run each benchmark with 48 threads, and report the average throughput across all five workloads in Figure 8.

Even under this deliberately heavy configuration, S4-FIFO sustains throughput comparable to conventional heuristics, with only a modest slowdown from continuous feature extraction. Achieving heuristic-level throughput is challenging for typical learned caches, which often require critical-path inference or online training. S4-FIFO avoids these costs by collecting features outside the critical section and performing inference infrequently and asynchronously using an offline-trained model. As a result, its critical path remains lightweight, preserving throughput and simplifying deployment. By contrast, TinyLFU incurs additional overhead because its count-min sketch must be updated within the critical section to ensure correctness, resulting in significant throughput degradation.

5.5 Why does Learning Work?

As a simple algorithm with static queue sizes, S4-FIFO achieves both higher efficiency and robustness compared to state-of-the-art algorithms. This section deep dives into how learning helps S4-FIFO achieve this and why one pre-trained model is sufficient.

Miss ratio over time. Figure 12 illustrates the effect of how learned parameters affect miss ratio over time for a representative trace. The vertical dashed line marks the point at which S4-FIFO switches to the predicted parameters, which shrink the small-queue size to 5% of the cache and skip frequency-counter increments for 25% of the items at the front of the small queue. After this parameter change, S4-FIFO (solid

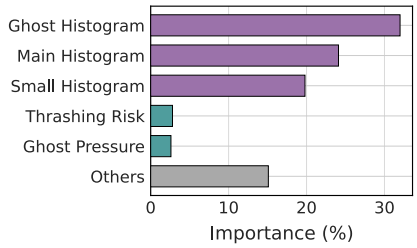


Figure 9: Feature importance by category.

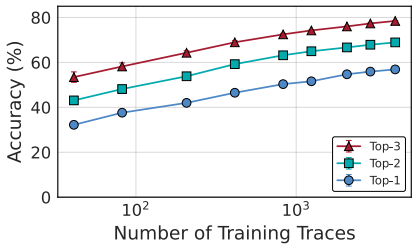


Figure 10: Impact of training data size.

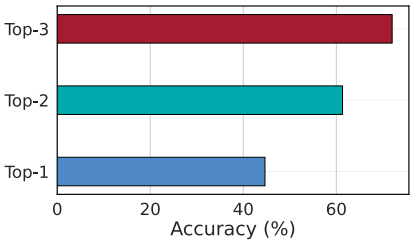
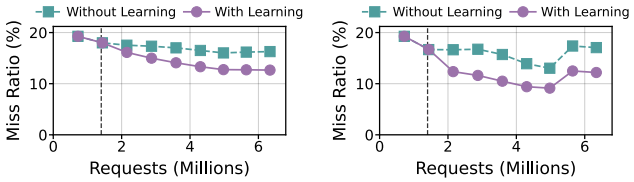
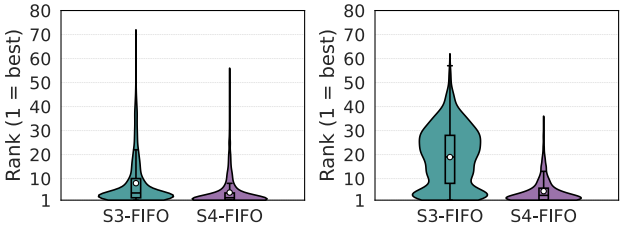


Figure 11: Prediction accuracy on the Twitter dataset on model pretrained on CDN2.



(a) Cumulative miss ratio **(b)** Interval miss ratio

Figure 12: Miss ratio over time where the learned parameters significantly reduce the miss ratio.



(a) Large cache **(b)** Small cache

Figure 13: Rank distribution of the default (S3-FIFO) and predicted parameters (S4-FIFO) from optimal grid search configuration.

line) quickly diverges from the baseline (dashed line): the interval miss ratio drops substantially and remains consistently lower for the remainder of the trace, and the cumulative miss ratio curve steadily widens its gap relative to the baseline. This example shows that although parameters remain static (i.e., not updated per-miss), learning a set of parameters for each workload can deliver a substantial reduction in miss ratio compared to using a fixed set of parameters.

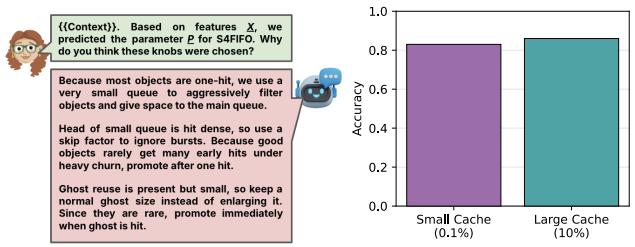
Rank of predicted parameters. Figure 13 compares the grid-search rank distributions of the default S4-FIFO parameters (most are inherited from S3-FIFO) with the learned parameters, where a lower rank indicates better performance (rank 1 is the best). In both small and large cache sizes, the predicted parameters from S4-FIFO are consistently close to the best configuration across traces. In contrast, the default parameters have a substantially wider distribution that extends to much worse ranks, indicating that using the same parameters across traces leaves significant headroom for improvement.

Feature importance. We compute the feature-importance scores from the trained decision tree. As shown in Figure 9, the histograms of all three queues (ghost, main, and small) are the most important. Together, these histogram features account for 75% of the total importance, indicating that the shape of hit distributions within each queue (workload locality) is the strongest signal for predicting good configurations. The remaining composite features, such as thrashing risk (working set size estimate) and ghost pressure (hits found on the ghost), account for 25% of the importance.

Training data size. To understand why a single pre-trained model works, we measure how prediction accuracy changes with the number of training traces. For each training dataset size, we pre-train a new model and evaluate on the same test dataset. We measure top-1, top-2, and top-3 accuracy, which are the fractions of traces in which the predicted parameters are ranked top-1, top-2, or top-3, respectively. Figure 10 shows that all three metrics monotonically increase with more training data. With several thousand traces, top-1 approaches 60%, top-2 approaches 70%, and top-3 approaches 80%. Although performance starts to plateau, increasing the training dataset size further is likely to make the pre-trained model increasingly accurate.

Cross-dataset generalization. The training and test traces we have been using so far are generated using a random split, without considering the dataset (source). This is acceptable because each dataset contains many diverse traces, and traces within a single dataset can be more diverse than those across datasets. To verify whether pre-trained models can generalize across datasets, we train a model on the CDN2 dataset (object cache) and evaluate on the Twitter dataset (key-value cache). Figure 11 shows that the model pre-trained on one dataset can be directly used on a different one, and the knowledge learned is generalizable as long as the training dataset is large and diverse enough.

Generality across eviction algorithms. The principles of LAH can be applied to other policies that expose a small set of configuration knobs. For example, CacheLib's LRU exposes configurable behaviors such as node-promotion frequency, whether metadata is updated on reads, writes, or contention,



(a) Explaining predicted parameters **(b)** Selecting suitable parameters

Figure 14: LLM-based interpretability evaluation. S4-FIFO's cache-level features and semantic parameters allow an LLM to explain predicted configurations and distinguish suitable parameter choices.

and the insertion position of new objects in the LRU queue. 2Q [35] exposes the relative sizes of its hot and cold queues, ARC [46] exposes the balance between its recency and frequency regions, and LeCaR [63] exposes the weights assigned to its eviction experts. Extending LAH to these policies requires policy-specific feature engineering. For example, an LRU-based instantiation could additionally track hit positions in the queue and their reuse distances. These extensions are beyond the scope of this work.

5.6 Is the Learned Decision Interpretable?

S4-FIFO learns from high-level cache features and predicts high-level knobs. Because both the inputs and outputs operate at the cache level, the decisions it produces are far more interpretable than those of prior smart-cache designs. Unlike approaches that make opaque predictions (e.g., ranking objects or estimating reuse densities), S4-FIFO expresses its decisions using knobs that operators already understand, such as queue sizes and promotion thresholds. The features S4-FIFO used characterize workload, capturing burstiness, one-hit ratios, and reuse-distance patterns, making the mapping from features to knobs easier to interpret.

We use LLM-based evaluation as a proxy for semantic interpretability: if an external reasoning model can connect workload-level features to the selected cache knobs, then the learned decisions are likely expressed in a form that operators can reason about. We evaluate this with two language-model-based experiments, shown in Figure 14. First, given workload features and a predicted S4-FIFO configuration, LLMs are able to explain why the selected knobs match the workload behavior. Second, we provide an LLM with the workload features and two configurations with distinctly different miss ratios and ask it to identify the better configuration. Across 100 traces for each cache size, the LLM performs substantially above chance, achieving 83% and 86% accuracy in the small-cache and large-cache setting. While these results do not establish the correctness of the learned policy, they suggest that S4-FIFO's feature and parameter spaces are semantically meaningful enough to support external reasoning, making the selected configurations easier to explain, justify, and debug. We provide further details in Appendix Section A.2.

6 Related Work

Optimizing cache efficiency. Many studies have examined the design of eviction algorithms to improve efficiency. These include static heuristics such as S3-FIFO [74], SIEVE [77], QDLP [71], LIRS [33,80], clock variants [16,32,42], 2Q [35], S4LRU [31], LRFU [22], LFU variants [5, 6, 36], greedy-dual variants [20, 34], MQ [84], Clock2Q+ [76], Hyperbolic [13]; per-miss cache-level adaptive algorithms, such as ARC [46], CAR [8], SEQ [29], EELRU [56], Clock-Pro+ [42], LeCaR [63], CACHEUS [53], TinyLFU [24]; and object-level learning algorithms, such as LRB [58], 3L-Cache [83], LHD [10], PA-Cache [26], RL-Belady [68], Raven [30], HALP [59], Darwin [17]. Compared with these algorithms, to the best of our knowledge, this is the first work to propose learning-augmented heuristics that separate the control and data-serving planes to achieve both high efficiency and performance. S4-FIFO demonstrates this idea by using a pre-trained model to learn the parameters of S3-FIFO.

Machine learning for systems. Prior works integrate ML into systems at three distinct levels of the stack, ranging from replacing internal mechanisms to tuning external knobs. (1) Structural replacement: Works such as learned indexes [38,60] and learned bloom filters [62] aim to *replace* data structures with neural networks. (2) Policy optimization: ML replaces specific heuristics (policies) while keeping the underlying mechanism intact. Examples include RL-based flash admission [75], and language-model-based C++ memory manager [44,45]. (3) Black-box autotuning: Systems like OtterTune [3] treat the database as a black box, optimizing configuration knobs via Bayesian optimization. S4-FIFO occupies a distinct point in this design space: unlike structural replacement, it preserves the mechanism's deterministic safety (the S4-FIFO data path); unlike per-instance policy optimization, it leverages a pre-trained foundation model to generalize to unseen workloads; and unlike black-box autotuning with slow iterative search, it performs zero-shot inference, quickly selecting an optimal configuration after a brief observation window and avoiding costly online exploration.

7 Conclusion

We introduce Learning-Augmented Heuristics (LAH), which keeps the data path simple while using a pre-trained model to configure a few semantic knobs. Our S4-FIFO instantiation extends S3-FIFO with tunable parameters and selects configurations based on cache-level features. Evaluated on 1,035 production traces, S4-FIFO achieves higher average miss-ratio reduction and stronger robustness than state-of-the-art algorithms, while matching the throughput of heuristics. These results indicate that learning to configure expressive heuristics is an effective and interpretable way to bring ML into core caching systems.

Acknowledgments

We thank the anonymous reviewers for their valuable feedback and our shepherd for the constructive suggestions. We also would like to thank the people and organizations that have open-sourced and shared production traces. We thank Cloudlab [23] for providing infrastructure support to run experiments.

References

[1] libcachesim: A high-performance cache simulator. https://github.com/1a1a11a/libCacheSim. Accessed: 2025-12-01.

[2] M2cgen repository. https://github.com/BayesWitnesses/m2cgen. Accessed: 2025-12-10.

[3] Dana Van Aken, Andrew Pavlo, Geoffrey J. Gordon, and Bohan Zhang. Automatic database management system tuning through large-scale machine learning. In Semih Salihoglu, Wenchao Zhou, Rada Chirkova, Jun Yang, and Dan Suciu, editors, *Proceedings of the 2017 ACM International Conference on Management of Data, SIGMOD Conference 2017, Chicago, IL, USA, May 14-19, 2017*, pages 1009–1024. ACM, 2017.

[4] Alibaba Group. Alibaba block traces. https://github.com/alibaba/block-traces, 2020. Accessed: 2025-12-01.

[5] Martin Arlitt, Ludmila Cherkasova, John Dilley, Rich Friedrich, and Tai Jin. Evaluating content management techniques for Web proxy caches. *ACM SIGMETRICS Performance Evaluation Review*, 27(4):3–11, March 2000.

[6] Martin Arlitt, Rich Friedrich, and Tai Jin. Performance Evaluation of Web Proxy Cache Replacement Policies. In *Perform. Eval.*, volume 39, pages 149–164, NLD, February 2000. Elsevier Science Publishers B. V.

[7] Nirav Atre, Justine Sherry, Weina Wang, and Daniel S. Berger. Caching with delayed hits. In Henning Schulzrinne and Vishal Misra, editors, *SIGCOMM '20: Proceedings of the 2020 Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication, Virtual Event, USA, August 10-14, 2020*, pages 495–513. ACM, 2020.

[8] Sorav Bansal and Dharmendra S. Modha. CAR: Clock with Adaptive Replacement. In *3rd USENIX Conference on File and Storage Technologies*, FAST'04, 2004.

[9] Soumya Basu, Aditya Sundarrajan, Javad Ghaderi, Sanjay Shakkottai, and Ramesh K. Sitaraman. Adaptive ttl-based caching for content delivery. In Bruce E. Hajek, Sewoong Oh, Augustin Chaintreau, Leana Golubchik, and Zhi-Li Zhang, editors, *Proceedings of the 2017 ACM SIGMETRICS / International Conference on Measurement and Modeling of Computer Systems, Urbana-Champaign, IL, USA, June 05 - 09, 2017*, pages 45–46. ACM, 2017.

[10] Nathan Beckmann, Haoxian Chen, and Asaf Cidon. LHD: improving cache hit rate by maximizing hit density. In Sujata Banerjee and Srinivasan Seshan, editors, *15th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2018, Renton, WA, USA, April 9-11, 2018*, pages 389–403. USENIX Association, 2018.

[11] Benjamin Berg, Daniel S. Berger, Sara McAllister, Isaac Grosof, Sathya Gunasekar, Jimmy Lu, Michael Uhlar, Jim Carrig, Nathan Beckmann, Mor Harchol-Balter, and Gregory R. Ganger. The CacheLib caching engine: Design and experiences at scale. In *14th USENIX symposium on operating systems design and implementation*, OSDI'20, pages 753–768. USENIX Association, November 2020.

[12] Daniel S. Berger, Ramesh K. Sitaraman, and Mor Harchol-Balter. Adaptsize: Orchestrating the hot object memory cache in a content delivery network. In Aditya Akella and Jon Howell, editors, *14th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2017, Boston, MA, USA, March 27-29, 2017*, pages 483–498. USENIX Association, 2017.

[13] Aaron Blankstein, Siddhartha Sen, and Michael J. Freedman. Hyperbolic caching: Flexible caching for web applications. In *2017 USENIX annual technical conference*, ATC'17, pages 499–511, Santa Clara, CA, July 2017. USENIX Association.

[14] John W. Byers, Jeffrey Considine, Michael Mitzenmacher, and Stanislav Rost. Informed content delivery across adaptive overlay networks. In Matthew Mathis, Peter Steenkiste, Hari Balakrishnan, and Vern Paxson, editors, *Proceedings of the ACM SIGCOMM 2002 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication, August 19-23, 2002, Pittsburgh, PA, USA*, pages 47–60. ACM, 2002.

[15] CacheLib. Evaluating SSD hardware for Facebook workloads: Running cachebench with the trace workload. https://cachelib.org/docs/Cache_Library_User_Guides/Cachebench_FB_HW_eval, n.d. Accessed: 2025-12-01.

[16] Richard W. Carr and John L. Hennessy. WSCLOCK: a simple and effective algorithm for virtual memory management. In *Proceedings of the eighth ACM symposium on Operating systems principles*, SOSP'81, pages 87–95, New York, NY, USA, December 1981. Association for Computing Machinery.

[17] Jiayi Chen, Nihal Sharma, Tarannum Khan, Shu Liu, Brian Chang, Aditya Akella, Sanjay Shakkottai, and Ramesh K Sitaraman. Darwin: Flexible Learning-based CDN Caching. In *Proceedings of the ACM SIGCOMM 2023 Conference*, SIGCOMM '23, pages 981–999, New York, NY, USA, September 2023. Association for Computing Machinery.

[18] Tianqi Chen and Carlos Guestrin. Xgboost: A scalable tree boosting system. In Balaji Krishnapuram, Mohak Shah, Alexander J. Smola, Charu C. Aggarwal, Dou Shen, and Rajeev Rastogi, editors, *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, San Francisco, CA, USA, August 13-17, 2016*, pages 785–794. ACM, 2016.

[19] Heng-Tze Cheng, Levent Koc, Jeremiah Harmsen, Tal Shaked, Tushar Chandra, Hrishi Aradhye, Glen Anderson, Greg Corrado, Wei Chai, Mustafa Ispir, Rohan Anil, Zakaria Haque, Lichan Hong, Vihan Jain, Xiaobing Liu, and Hemal Shah. Wide & deep learning for recommender systems. In Alexandros Karatzoglou, Balázs Hidasi, Domonkos Tikk, Oren Sar Shalom, Haggai Roitman, Bracha Shapira, and Lior Rokach, editors, *Proceedings of the 1st Workshop on Deep Learning for Recommender Systems, DLRS@RecSys 2016, Boston, MA, USA, September 15, 2016*, pages 7–10. ACM, 2016.

[20] Ludmila Cherkasova. *Improving WWW proxies performance with greedy-dual-size-frequency caching policy*. Citeseer, 1998.

[21] Peter J. Denning. The working set model for program behavior. In *Proceedings of the First ACM Symposium on Operating System Principles*, SOSP '67, page 15.1–15.12, New York, NY, USA, 1967. Association for Computing Machinery.

[22] Donghee Lee, Jongmoo Choi, Jong-Hun Kim, S.H. Noh, Sang Lyul Min, Yookun Cho, and Chong Sang Kim. LRFU: a spectrum of policies that subsumes the least recently used and least frequently used policies. *IEEE Transactions on Computers*, 50(12):1352–1361, December 2001.

[23] Dmitry Duplyakin, Robert Ricci, Aleksander Maricq, Gary Wong, Jonathon Duerig, Eric Eide, Leigh Stoller, Mike Hibler, David Johnson, Kirk Webb, Aditya Akella, Kuangching Wang, Glenn Ricart, Larry Landweber, Chip Elliott, Michael Zink, Emmanuel Cecchet, Snigdhaswin Kar, and Prabodh Mishra. The design and operation of CloudLab. In *Proceedings of the USENIX Annual Technical Conference (ATC)*, pages 1–14, July 2019.

[24] Gil Einziger, Roy Friedman, and Ben Manes. Tinylfu: A highly efficient cache admission policy. *ACM Trans. Storage*, 13(4):35:1–35:31, 2017.

[25] Bin Fan, Hyeontaek Lim, David G. Andersen, and Michael Kaminsky. Small cache, big effect: provable load balancing for randomly partitioned cluster services. In Jeffrey S. Chase and Amr El Abbadi, editors, *ACM Symposium on Cloud Computing in conjunction with SOSP 2011, SOCC '11, Cascais, Portugal, October 26-28, 2011*, page 23. ACM, 2011.

[26] Qilin Fan, Xiuhua Li, Jian Li, Qiang He, Kai Wang, and Junhao Wen. PA-Cache: Evolving Learning-Based Popularity-Aware Content Caching in Edge Networks, December 2020.

[27] Qilin Fan, Xiuhua Li, Jian Li, Qiang He, Kai Wang, and Junhao Wen. Pa-cache: Evolving learning-based popularity- aware content caching in edge networks. *IEEE Trans. Netw. Serv. Manag.*, 18(2):1746–1757, 2021.

[28] Jerome H Friedman. Greedy function approximation: a gradient boosting machine. *Annals of statistics*, pages 1189–1232, 2001.

[29] Gideon Glass and Pei Cao. Adaptive page replacement based on memory reference behavior. *ACM SIGMETRICS Performance Evaluation Review*, 25(1):115–126, June 1997.

[30] Xinyue Hu, Eman Ramadan, Wei Ye, Feng Tian, and Zhi-Li Zhang. Raven: belady-guided, predictive (deep) learning for in-memory and content caching. In *Proceedings of the 18th International Conference on emerging Networking EXperiments and Technologies*, CoNEXT'22, pages 72–90, New York, NY, USA, November 2022. Association for Computing Machinery.

[31] Qi Huang, Ken Birman, Robbert van Renesse, Wyatt Lloyd, Sanjeev Kumar, and Harry C. Li. An analysis of Facebook photo caching. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP'13, pages 167–181, New York, NY, USA, November 2013. Association for Computing Machinery.

[32] Song Jiang, Feng Chen, and Xiaodong Zhang. CLOCK-Pro: an effective improvement of the CLOCK replacement. In *Proceedings of the annual conference*

on USENIX Annual Technical Conference, ATC'05, page 35, USA, April 2005. USENIX Association.

[33] Song Jiang and Xiaodong Zhang. LIRS: an efficient low inter-reference recency set replacement policy to improve buffer cache performance. In Richard R. Muntz, Margaret Martonosi, and Edmundo de Souza e Silva, editors, *Proceedings of the International Conference on Measurements and Modeling of Computer Systems, SIGMETRICS 2002, June 15-19, 2002, Marina Del Rey, California, USA*, pages 31–42. ACM, 2002.

[34] Shudong Jin and A. Bestavros. Popularity-aware greedy dual-size Web proxy caching algorithms. In *Proceedings 20th IEEE International Conference on Distributed Computing Systems*, pages 254–261, 2000.

[35] Theodore Johnson and Dennis E. Shasha. 2q: A low overhead high performance buffer management replacement algorithm. In Jorge B. Bocca, Matthias Jarke, and Carlo Zaniolo, editors, *VLDB'94, Proceedings of 20th International Conference on Very Large Data Bases, September 12-15, 1994, Santiago de Chile, Chile*, pages 439–450. Morgan Kaufmann, 1994.

[36] George Karakostas and D Serpanos. Practical LFU implementation for web caching. *Technical Report TR-622-00*, 2000.

[37] Ricardo Koller and Raju Rangaswami. I/O deduplication: Utilizing content similarity to improve I/O performance. *ACM Trans. Storage*, 6(3):13:1–13:26, 2010.

[38] Tim Kraska, Alex Beutel, Ed H. Chi, Jeffrey Dean, and Neoklis Polyzotis. The case for learned index structures. In Gautam Das, Christopher M. Jermaine, and Philip A. Bernstein, editors, *Proceedings of the 2018 International Conference on Management of Data, SIGMOD Conference 2018, Houston, TX, USA, June 10-15, 2018*, pages 489–504. ACM, 2018.

[39] Chunghan Lee, Tatsuo Kumano, Tatsuma Matsuki, Hiroshi Endo, Naoto Fukumoto, and Mariko Sugawara. Systor '17 traces (SNIA IOTTA trace set 4931). In Geoff Kuenning, editor, *SNIA IOTTA Trace Repository*. Storage Networking Industry Association, February 2016.

[40] Chunghan Lee, Tatsuo Kumano, Tatsuma Matsuki, Hiroshi Endo, Naoto Fukumoto, and Mariko Sugawara. Understanding storage traffic characteristics on enterprise virtual desktop infrastructure. In Doron Chen, Peter Desnoyers, and Eyal de Lara, editors, *Proceedings of the 10th ACM International Systems and Storage Conference, SYSTOR 2017, Haifa, Israel, May 22-24, 2017*, pages 13:1–13:11. ACM, 2017.

[41] Cong Li. Dlirs: Improving low inter-reference recency set cache replacement policy with dynamics. In *Proceedings of the 11th ACM International Systems and Storage Conference*, pages 59–64, 2018.

[42] Cong Li. CLOCK-pro+: improving CLOCK-pro cache replacement with utility-driven adaptation. In *Proceedings of the 12th ACM International Conference on Systems and Storage*, SYSTOR'19, pages 1–7, New York, NY, USA, May 2019. Association for Computing Machinery.

[43] Jinhong Li, Qiuping Wang, Patrick P. C. Lee, and Chao Shi. An in-depth analysis of cloud block storage workloads in large-scale production. In *IEEE International Symposium on Workload Characterization, IISWC 2020, Beijing, China, October 27-30, 2020*, pages 37–47. IEEE, 2020.

[44] Martin Maas, David G. Andersen, Michael Isard, Mohammad Mahdi Javanmard, Kathryn S. McKinley, and Colin Raffel. Learning-based memory allocation for C++ server workloads. In James R. Larus, Luis Ceze, and Karin Strauss, editors, *ASPLOS '20: Architectural Support for Programming Languages and Operating Systems, Lausanne, Switzerland, March 16-20, 2020*, pages 541–556. ACM, 2020.

[45] Martin Maas, David G. Andersen, Michael Isard, Mohammad Mahdi Javanmard, Kathryn S. McKinley, and Colin Raffel. Combining machine learning and lifetime-based resource management for memory allocation and beyond. *Commun. ACM*, 67(4):87–96, 2024.

[46] Nimrod Megiddo and Dharmendra S. Modha. ARC: A self-tuning, low overhead replacement cache. In Jeff Chase, editor, *Proceedings of the FAST '03 Conference on File and Storage Technologies, March 31 - April 2, 2003, Cathedral Hill Hotel, San Francisco, California, USA*. USENIX, 2003.

[47] Kianoosh Mokhtarian and Hans-Arno Jacobsen. Caching in video cdns: building strong lines of defense. In Dick C. A. Bulterman, Herbert Bos, Antony I. T. Rowstron, and Peter Druschel, editors, *Ninth Eurosys Conference 2014, EuroSys 2014, Amsterdam, The Netherlands, April 13-16, 2014*, pages 13:1–13:13. ACM, 2014.

[48] Dushyanth Narayanan, Austin Donnelly, and Antony Rowstron. MSR Cambridge traces (SNIA IOTTA trace 386). In Geoff Kuenning, editor, *SNIA IOTTA Trace Repository*. Storage Networking Industry Association, March 2007.

[49] Dushyanth Narayanan, Austin Donnelly, and Antony I. T. Rowstron. Write off-loading: Practical power man-

agement for enterprise storage. *ACM Trans. Storage*, 4(3):10:1–10:23, 2008.

[50] Erik Nygren, Ramesh K. Sitaraman, and Jennifer Sun. The akamai network: a platform for high-performance internet applications. *ACM SIGOPS Oper. Syst. Rev.*, 44(3):2–19, 2010.

[51] Elizabeth J. O'Neil, Patrick E. O'Neil, and Gerhard Weikum. The LRU-K page replacement algorithm for database disk buffering. In Peter Buneman and Sushil Jajodia, editors, *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data, Washington, DC, USA, May 26-28, 1993*, pages 297–306. ACM Press, 1993.

[52] Steffen Rendle. Factorization machines with libfm. *ACM Trans. Intell. Syst. Technol.*, 3(3):57:1–57:22, 2012.

[53] Liana V. Rodriguez, Farzana Yusuf, Steven Lyons, Eysler Paz, Raju Rangaswami, Jason Liu, Ming Zhao, and Giri Narasimhan. Learning Cache Replacement with CACHEUS. In *19th USENIX Conference on File and Storage Technologies*, FAST'21, pages 341–354. USENIX Association, February 2021.

[54] Kyle Schomp, Onkar Bhardwaj, Eymen Kurdoglu, Mashooq Muhaimen, and Ramesh K. Sitaraman. Akamai DNS: providing authoritative answers to the world's queries. In Henning Schulzrinne and Vishal Misra, editors, *SIGCOMM '20: Proceedings of the 2020 Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication, Virtual Event, USA, August 10-14, 2020*, pages 465–478. ACM, 2020.

[55] Peter Slavík. A tight analysis of the greedy algorithm for set cover. *J. Algorithms*, 25(2):237–254, 1997.

[56] Yannis Smaragdakis, Scott Kaplan, and Paul Wilson. EELRU: simple and effective adaptive page replacement. *ACM SIGMETRICS Performance Evaluation Review*, 27(1):122–133, May 1999.

[57] Hyunsub Song, Shean Kim, J. Hyun Kim, Ethan J. H. Park, and Sam H. Noh. First responder: Persistent memory simultaneously as high performance buffer cache and storage. In Irina Calciu and Geoff Kuenning, editors, *Proceedings of the 2021 USENIX Annual Technical Conference, USENIX ATC 2021, July 14-16, 2021*, pages 839–853. USENIX Association, 2021.

[58] Zhenyu Song, Daniel S. Berger, Kai Li, and Wyatt Lloyd. Learning relaxed belady for content distribution network caching. In Ranjita Bhagwan and George Porter, editors, *17th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2020, Santa Clara, CA, USA, February 25-27, 2020*, pages 529–544. USENIX Association, 2020.

[59] Zhenyu Song, Kevin Chen, Nikhil Sarda, Deniz Altinbuken, Eugene Brevdo, Jimmy Coleman, Xiao Ju, Pawel Jurczyk, Richard Schooler, and Ramki Gummadi. HALP: Heuristic Aided Learned Preference Eviction Policy for YouTube Content Delivery Network. In *20th USENIX Symposium on Networked Systems Design and Implementation*, NSDI'23, pages 1149–1163, 2023.

[60] Mihail Stoian, Andreas Kipf, Ryan Marcus, and Tim Kraska. PLEX: towards practical learned indexing. *CoRR*, abs/2108.05117, 2021.

[61] Aditya Sundarrajan, Mingdong Feng, Mangesh Kasbekar, and Ramesh K. Sitaraman. Footprint descriptors: Theory and practice of cache provisioning in a global CDN. In *Proceedings of the 13th International Conference on emerging Networking EXperiments and Technologies, CoNEXT 2017, Incheon, Republic of Korea, December 12 - 15, 2017*, pages 55–67. ACM, 2017.

[62] Kapil Vaidya, Eric Knorr, Michael Mitzenmacher, and Tim Kraska. Partitioned learned bloom filters. In *9th International Conference on Learning Representations, ICLR 2021, Virtual Event, Austria, May 3-7, 2021*. OpenReview.net, 2021.

[63] Giuseppe Vietri, Liana V. Rodriguez, Wendy A. Martinez, Steven Lyons, Jason Liu, Raju Rangaswami, Ming Zhao, and Giri Narasimhan. Driving cache replacement with ml-based lecar. In Ashvin Goel and Nisha Talagala, editors, *10th USENIX Workshop on Hot Topics in Storage and File Systems, HotStorage 2018, Boston, MA, USA, July 9-10, 2018*. USENIX Association, 2018.

[64] Carl A. Waldspurger, Nohhyun Park, Alexander Garthwaite, and Irfan Ahmad. Efficient mrc construction with shards. In *Proceedings of the 13th USENIX Conference on File and Storage Technologies*, FAST'15, page 95–110, USA, 2015. USENIX Association.

[65] Qiuping Wang, Jinhong Li, Patrick P. C. Lee, Tao Ouyang, Chao Shi, and Lilong Huang. Separating data via block invalidation time inference for write amplification reduction in log-structured storage. In Dean Hildebrand and Donald E. Porter, editors, *20th USENIX Conference on File and Storage Technologies, FAST 2022, Santa Clara, CA, USA, February 22-24, 2022*, pages 429–444. USENIX Association, 2022.

[66] Wikimedia Foundation. Analytics/Data Lake/Traffic/Caching. https://wikitech.wikimedia.org/wiki/Analytics/Data_Lake/Traffic/Caching, n.d. Accessed: 2025-12-01.

[67] Kan Wu, Zhihan Guo, Guanzhou Hu, Kaiwei Tu, Ramnatthan Alagappan, Rathijit Sen, Kwanghyun Park, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. The storage hierarchy is not a hierarchy: Optimizing caching on modern storage devices with orthus. In *19th USENIX conference on file and storage technologies*, FAST'21, pages 307–323. USENIX Association, February 2021.

[68] Gang Yan and Jian Li. RL-Bélády: A Unified Learning Framework for Content Caching. In *Proceedings of the 28th ACM International Conference on Multimedia*, MM'20, pages 1009–1017, Seattle WA USA, October 2020. ACM.

[69] Gang Yan and Jian Li. Towards latency awareness for content delivery network caching. In Jiri Schindler and Noa Zilberman, editors, *Proceedings of the 2022 USENIX Annual Technical Conference, USENIX ATC 2022, Carlsbad, CA, USA, July 11-13, 2022*, pages 789–804. USENIX Association, 2022.

[70] Juncheng Yang, Ziming Mao, Yao Yue, and K. V. Rashmi. Gl-cache: Group-level learning for efficient and high-performance caching. In Ashvin Goel and Dalit Naor, editors, *21st USENIX Conference on File and Storage Technologies, FAST 2023, Santa Clara, CA, USA, February 21-23, 2023*, pages 115–134. USENIX Association, 2023.

[71] Juncheng Yang, Ziyue Qiu, Yazhuo Zhang, Yao Yue, and K. V. Rashmi. FIFO can be Better than LRU: the Power of Lazy Promotion and Quick Demotion. In *Proceedings of the 19th Workshop on Hot Topics in Operating Systems*, HOTOS'23, pages 70–79, New York, NY, USA, June 2023. Association for Computing Machinery.

[72] Juncheng Yang, Anirudh Sabnis, Daniel S. Berger, K. V. Rashmi, and Ramesh K. Sitaraman. C2DN: how to harness erasure codes at the edge for efficient content delivery. In Amar Phanishayee and Vyas Sekar, editors, *19th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2022, Renton, WA, USA, April 4-6, 2022*, pages 1159–1177. USENIX Association, 2022.

[73] Juncheng Yang, Yao Yue, and K. V. Rashmi. A large scale analysis of hundreds of in-memory cache clusters at twitter. In *14th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2020, Virtual Event, November 4-6, 2020*, pages 191–208. USENIX Association, 2020.

[74] Juncheng Yang, Yazhuo Zhang, Ziyue Qiu, Yao Yue, and Rashmi Vinayak. Fifo queues are all you need for cache eviction. In *Proceedings of the 29th Symposium on Operating Systems Principles*, SOSP '23, page 130–149, New York, NY, USA, 2023. Association for Computing Machinery.

[75] Tzu-Wei Yang, Seth Pollen, Mustafa Uysal, Arif Merchant, and Homer Wolfmeister. CacheSack: Admission optimization for google datacenter flash caches. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, pages 1021–1036, Carlsbad, CA, July 2022. USENIX Association.

[76] Yiyan Zhai, Bintang Dwi Marthen, Sarath Balivada, Vamsi Sudhakar Bojji, Eric Knauft, Jitender Rohilla, Jiaqi Zuo, Quanxing Liu, Maxime Austruy, Wenguang Wang, and Juncheng Yang. Clock2Q+: A simple and efficient replacement algorithm for metadata cache in VMware vSAN, 2025.

[77] Yazhuo Zhang, Juncheng Yang, Yao Yue, Ymir Vigfusson, and K.V. Rashmi. SIEVE is simpler than LRU: an efficient Turn-Key eviction algorithm for web caches. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*, pages 1229–1246, Santa Clara, CA, April 2024. USENIX Association.

[78] Yu Zhang, Ping Huang, Ke Zhou, Hua Wang, Jianying Hu, Yongguang Ji, and Bin Cheng. Tencent block storage traces (SNIA IOTTA trace 27920). In Geoff Kuenning, editor, *SNIA IOTTA Trace Repository*. Storage Networking Industry Association, September 2018.

[79] Yu Zhang, Ping Huang, Ke Zhou, Hua Wang, Jianying Hu, Yongguang Ji, and Bin Cheng. OSCA: an online-model based cache allocation scheme in cloud block storage systems. In Ada Gavrilovska and Erez Zadok, editors, *Proceedings of the 2020 USENIX Annual Technical Conference, USENIX ATC 2020, July 15-17, 2020*, pages 785–798. USENIX Association, 2020.

[80] Chen Zhong, Xingsheng Zhao, and Song Jiang. LIRS2: an improved LIRS replacement algorithm. In *Proceedings of the 14th ACM International Conference on Systems and Storage*, SYSTOR'21, pages 1–12, Haifa Israel, June 2021. ACM.

[81] Ke Zhou, Si Sun, Hua Wang, Ping Huang, Xubin He, Rui Lan, Wenyan Li, Wenji Liu, and Tianming Yang. Tencent photo cache traces (SNIA IOTTA trace set 27479). In Geoff Kuenning, editor, *SNIA IOTTA Trace Repository*. Storage Networking Industry Association, February 2016.

[82] Ke Zhou, Si Sun, Hua Wang, Ping Huang, Xubin He, Rui Lan, Wenyan Li, Wenjie Liu, and Tianming Yang. Demystifying cache policies for photo stores at scale: A tencent case study. In *Proceedings of the 32nd International Conference on Supercomputing, ICS 2018, Beijing, China, June 12-15, 2018*, pages 284–294. ACM, 2018.

[83] Wenbin Zhou, Zhixiong Niu, Yongqiang Xiong, Juan Fang, and Qian Wang. 3l-cache: Low overhead and precise learning-based eviction policy for caches. In Haryadi S. Gunawi and Vasily Tarasov, editors, *23rd USENIX Conference on File and Storage Technologies, FAST 2025, Santa Clara, CA, February 25-27, 2025*, pages 237–254. USENIX Association, 2025.

[84] Yuanyuan Zhou, James Philbin, and Kai Li. The multi-queue replacement algorithm for second level buffer caches. In *Proceedings of the annual conference on USENIX Annual Technical Conference*, ATC'01, pages 91–104, USA, 2001. USENIX Association.

WriteGuards: Distributed Storage Support for Strongly Consistent Caches

Ziming Mao[†] Atul Adya[‡] Jonathan Ellithorpe[‡]
Rishabh Iyer[†] Matei Zaharia[†] Scott Shenker[†◇] Ion Stoica[†]
[†]*UC Berkeley* [‡]*Databricks* [◇]*ICSI*

Abstract

This paper presents a set of in-process and remote distributed caches for datacenter environments, CLINK and CRINK respectively, that provide linearizable reads entirely from memory without contacting storage. These caches remain loosely coupled to the storage layer and achieve high performance, scale, and availability by cooperating with auto-sharders and by tracking consistency metadata at the granularity of key ranges rather than individual keys. To our knowledge, CLINK is the first distributed linked cache that delivers scalable, linearizable reads from memory while remaining loosely coupled with storage.

At the heart of these caches is a lightweight storage primitive called *WriteGuards* that can be easily added to a distributed store. WriteGuards prevent a subtle race we call the *delayed-writes anomaly* arising during changes in ownership of key ranges. Each write carries a small fencing value tied to the current owner, and the storage system checks this value to reject delayed writes. WriteGuards apply to key ranges instead of individual keys for scalability, add only a conditional check on the write path, and require no coordination on reads.

We implemented our cache designs on TiDB. The in-process cache CLINK cuts tail read latency by three orders of magnitude, and the remote cache CRINK reduces it by $2.2 - 2.4\times$ relative to direct storage access and existing strongly consistent remote caches.

1 Introduction

Many large-scale cloud services in datacenter environments require low-latency access to the latest state on the critical path. For example, a data governance service at Databricks must retrieve current permissions before every request, and a query coordination service must track up-to-date session state to ensure correct SQL execution. These requirements are widespread: major companies such as Dropbox [81], Google [20], and Meta [39] report similar needs. Furthermore, recent work [64] highlights the necessity of linearizable reads in latency-critical domains like financial trading [17, 29] and online ad bidding [96].

While distributed storage systems provide strong consistency, they incur high overheads. Each read involves network round trips, serialization, and transaction processing [31, 44, 79], resulting in multi-millisecond tail latencies [34]. To avoid this cost, large-scale services rely on distributed lookaside caches to serve frequently accessed data from memory. These caches are particularly effective for *serving* workloads, where reads dominate and writes are relatively infrequent [18, 67]. However, these caches generally provide only eventual consistency and cannot guarantee that reads reflect the latest committed state.

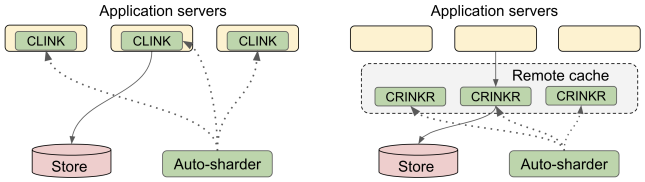


Figure 1: CLINK (in-process) and CRINK-R (remote) architectures.

A distributed write-through cache can provide linearizable reads, but several challenges must be addressed. First, to be usable as a general solution it must remain loosely coupled with storage so each layer can evolve independently. Second, it must scale to hundreds of millions or billions of keys. Third, for high performance and availability, the cache must handle dynamic resharding of keys, e.g. to move keys from unhealthy to healthy pods or for load balancing purposes.

Finally, the most significant challenge, which serves as the primary focus of this paper, is a subtle race condition we term the *delayed-writes problem*. When resharding occurs and a key is reassigned, the previous owner may still have writes in flight to storage. If the new owner reads and caches values before those writes arrive, the late writes can update storage and render the new owner's cache stale. This puts the new owner at risk of serving stale data. The delayed-writes problem is not a hypothetical possibility. Write operations have been observed in production systems [18, 67] to stall for extended periods, with GitHub reporting packets remaining in flight for approximately ninety seconds during an outage [82]. Such slowdowns can cause writes to arrive at the storage layer long after key ownership has changed in the application. Preventing delayed writes from committing after ownership changes is therefore essential for serving linearizable reads.

To address the delayed-writes problem, we introduce *WriteGuards*, a scalable and lightweight fencing mechanism that also enables loose coupling. Storage maintains WriteGuards as soft state: a small map from key ranges to opaque tokens checked on each write. On cache key ownership changes, the new cache pod owner installs WriteGuards over key ranges in storage to block delayed writes from prior owners. Write-

Guards enable decoupling of ownership logic in the cache layer from storage, allowing both systems to evolve independently. Because WriteGuards are simple, efficient, and broadly applicable, we recommend that providers of distributed storage systems expose them as a standard write-fencing primitive.

We leverage WriteGuards to construct in-process and remote caches that serve linearizable reads directly from memory. In these designs, new owners install guards in storage to block delayed writes. Our in-process solution, CLINK (Consistent Linked In-memory Key-value cache, Figure 1), keeps data in the application address space. A read from CLINK is equivalent to a linearizable read from storage, making it a transparent cache over the underlying store. For shared or multi-tenant scenarios, we present a remote strongly consistent cache, CRINK (Consistent Remote In-memory Key-value cache), with two architectures: CRINK-R (Figures 1 and 9) stores versions and values in a remote tier, whereas CRINK-L (Figure 10) stores only versions, allowing it to be paired with any lookaside cache (e.g., Redis). In all our designs, the cache is write-through; all writes flow through it to storage. The write throughput remains bounded by the storage system, ensuring the cache adds no new write bottleneck.

Like other modern cache systems, we address the third challenge of high performance and availability via an auto-sharder [7, 9, 53]. Improving over static placement schemes such as consistent hashing, auto-sharders dynamically reshard key ranges to handle autoscaling, pod failures, rolling upgrades, and load imbalance. They scale by tracking sharding metadata at the range rather than key level and can ensure that all pods agree on the key-to-pod assignment, which CLINK and CRINK rely on to support linearizable cached reads.

Both caches handle hot keys by letting the auto-sharder isolate a key and create replicas, with one acting as primary. Reads for a key can be served from storage by any pod, or directly from the in-memory cache of the pod currently assigned that key. All writes flow through the primary and follow a two-phased invalidation protocol. This could suggest reduced write availability during pod restarts. However, because unplanned failures are rare and the auto-sharder performs proactive resharding during planned restarts, the observed write unavailability remains extremely low (below 0.001%, see §2.3.2). Hence, CLINK and CRINK provide scalable, low latency linearizable reads across replicas while preserving high write availability.

Our approach extends to cross-datacenter environments, where caching offers significant benefits for geo-distributed applications using global stores like Spanner [30] by reducing expensive WAN traffic. However, we limit the scope of this paper to single-datacenter environments.

We implemented WriteGuards in TiDB [44] and built CLINK and CRINK in C++. Using production traces from Meta [5, 15], Twitter [88], and an internal workload from Unity Catalog at Databricks, we show CLINK reduces P_{90} read latency by orders of magnitude relative to accessing storage (from 10.3ms to 4.2μs). For shared data access or multi-tenant deployments, our remote caches CRINK-R/CRINK-L achieve up to $3.6-5.2\times$ lower P_{90} read latency than direct storage access, and $2.2-2.4\times$ lower latency compared to alternative consistent remote caches.

This paper makes three contributions:

- **WriteGuards:** a lightweight storage-level fencing mechanism that operates on dynamic ranges to prevent delayed writes from previous owners after resharding, and which can be easily added to a distributed store.
- **Strongly consistent caches:** CLINK, the first strongly consistent distributed cache that serves scalable, linearizable reads directly from application memory while remaining independent of storage, and its variants CRINK-R and CRINK-L which can be deployed remotely.
- **Implementation & evaluation:** we have integrated WriteGuards in TiDB and demonstrated their use in building performant linked and remote caches.

2 Background

In this section, we introduce the application, storage, caching, and auto-sharding model that motivates our design choices, providing the architectural context needed for the later sections on supporting linearizable cached reads.

2.1 Application and storage model

Many large-scale datacenter services process requests keyed by logical identifiers across billions of keys. Each request issues reads and writes for a specific identifier, such as a *CustomerID*. Underlying systems like DynamoDB [76], FoundationDB [100], and TiKV [44] support these per-key operations. The identifier often prefixes the full storage key (e.g., *CustomerID* in <*CustomerID, WorkspaceID*>); we denote this key as *C*.

2.2 Distributed caching for latency and scale

Remote lookaside caches (Figure 3) like Memcached or Redis offload storage but incur network latencies, serialization, and queuing costs [8]. In contrast, linked in-process caches like CLINK (Figure 1) store objects in native formats, avoiding remote lookups for significantly lower latency [8, 9].

This efficiency is critical for services storing large values (MBs–GBs). Remote caches struggle with partial reads of these values: returning the entire object causes overreads [8], while partial fetching often amplifies one API call into 10–30 network requests. Linked caches eliminate this overhead, turning expensive network operations into cheap local function calls.

2.3 Auto-sharders

To scale caches, modern systems partition keys across pods using auto-sharders [9, 53] rather than using static schemes like consistent hashing. At Databricks, auto-sharding serves more than half of all services and nearly all critical ones. These systems dynamically split, merge, and reassign contiguous key ranges, or *slices*, to handle load imbalance, hot keys, crashes, and to support high performance and availability. Auto-sharders also coordinate with cluster managers for

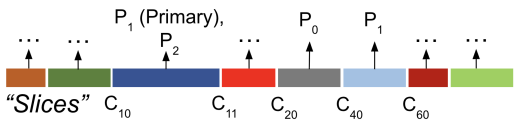


Figure 2: An example assignment of slices to pods. Each slice is assigned to one or more replicas, with one as primary for writes.

planned restarts and migrate state during reassignment [53] to avoid storage reads. Figure 2 illustrates a sample mapping.

Auto-sharding is also integral to storage systems. For example, Bigtable's master and TiDB's Placement Driver place and rebalance *tablets*, which are key ranges with associated records, across storage servers, also called *tablet servers*.

Figure 3 shows how auto-sharders route requests across the cache and storage layers. A request for customer C_{45} is directed to cache pod P_1, which owns slice $[C_{40}, C_{60})$. A cache hit returns immediately. On a miss, the pod issues a lookup to tablet server S_1, which hosts tablet $[C_{30}, C_{50})$, retrieves the value, populates the cache, and responds to the client. Linked caches follow the same routing logic except that the request goes directly to the application pod that owns the slice and stores the cached data locally. The caching layer and the storage layer are loosely coupled and sharded differently based on their respective load.

2.3.1 Features for scalability and strong consistency

Modern auto-sharders provide mechanisms that help services handle skewed workloads, scale under heavy demand, and achieve exclusive ownership over keys.

Hot key detection and replication: Many large-scale systems experience load imbalance driven by hot keys, which attract far more traffic than others in domains such as social media, ML serving, and content platforms [9, 43, 55, 59, 66, 67, 90]. Auto-sharders detect these hotspots and isolate the hot key onto its own slice and pod [9, 53]. When even a dedicated pod cannot meet demand for the hot key, the auto-sharder employs *asymmetric replication*, creating multiple replicas of only the hot key and assigning one to be the primary to handle writes. This approach scales throughput in proportion to key popularity. For example, Figure 2 shows key C_{10} isolated and replicated across P_1 and P_2, with P_1 acting as the primary.

Strong ownership guarantees: Auto-sharders [7, 9, 53]

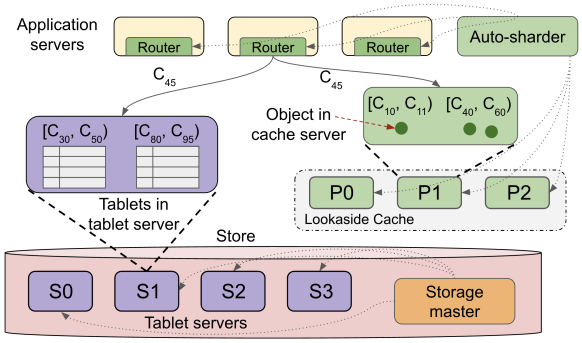


Figure 3: Auto-sharders assign and rebalance key ranges across cache or tablet servers to maintain performance and high availability.

support ensuring that at most one pod holds authoritative ownership of a key at any time. Our auto-sharder generalizes this model by producing assignments that all pods agree on, including which pods act as primaries when replicas exist, a property we call *Assignment Consistency*. To implement this efficiently at scale, the auto-sharder grants leases over key ranges rather than individual keys, amortizing coordination overhead across many keys while preserving consistent ownership semantics. The server library exposes the following API:

```
GetSliceHandle(KeyRange) → SliceHandle
IsStillAssigned(SliceHandle) → Bool
IsPrimary(SliceHandle) → Bool
```

The *GetSliceHandle* API returns a *SliceHandle* for a key range $[lowInclusive, highExclusive)$ where a single key is simply a singleton range. The *IsStillAssigned* API verifies that the *GetSliceHandle*'s range has remained continuously assigned to the pod since the handle was acquired, acting as a lightweight lease check implemented internally by the auto-sharder. When the auto-sharder assigns key K to pod P, the time interval until K is reassigned defines P's *period of ownership*. If a key is replicated, *IsPrimary* says whether the pod is the primary replica for that SliceHandle's range.

2.3.2 Availability impact of restarts and crashes

Strong ownership allows a cache implementation such as CLINK to let the pod that owns a key K serve linearizable reads directly from memory, since all writes on K are guaranteed to route through that pod. Ensuring strong ownership, however, necessarily introduces an availability tradeoff when ownership must move, for example during pod restarts or crashes.

In the common case of planned restarts, auto-sharders integrate with the cluster manager to receive advance notice and proactively migrate affected key ranges before the restart occurs. In systems enforcing strong ownership, this involves transferring leases for those key ranges from the restarting pod to other pods. In the common case, these transfers complete within milliseconds, resulting in minimal availability impact.

In rare cases, however, restarts are unplanned, for example due to a segfault or OOM. In these situations, the old owner's lease must expire before another pod can safely assume

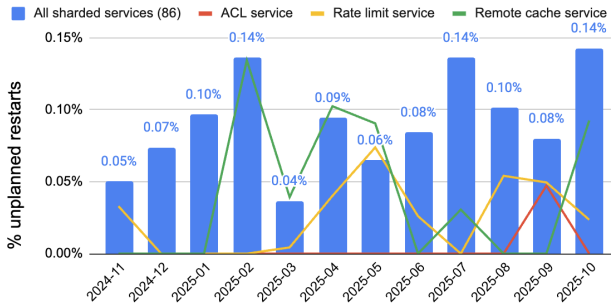


Figure 4: Global % of unplanned restarts across 86 sharded services in production by month. Critical path services for caching, ACLs, and rate limiting generally have few to no crashes in a given month.

ownership, which in practice results in order tens of seconds of unavailability for the key range. Though this may seem disastrous for availability, Figure 4 shows that such unplanned failures in production occur only once in roughly 750 to 2000 restarts, consistent with industry observations that critical services are engineered for robustness [80,84]. In a 100 pod service, this corresponds to roughly one crash every two months. Even if a single pod becomes unavailable for 30 seconds and temporarily affects 1% of keys, overall daily availability still exceeds 99.999%, making the practical impact negligible.

3 Motivation

Large scale services increasingly need both low latency and strong consistency, yet existing storage and caching layers struggle to provide both efficiently.

3.1 Low-latency reads are critical

Low tail latency is essential in serving scenarios [18, 67] as well as for interactive and data-intensive services. In practice, reads often suffer 1–5 ms tails due to queuing and concurrency, despite sub-millisecond network RTTs. Because requests fan out into many key-value RPCs, small delays compound into tail amplification [34, 35, 41, 51], impacting systems like Facebook's web search [18, 67], Google Search [34], Zanzibar [46], Presto [38], Michelangelo [1], and TFX [14]. This forces substantial overprovisioning to meet strict SLOs (e.g., DynamoDB [4], Cosmos DB [3], TAO [2]). Research systems [37, 50, 57] report tens of microseconds, but rely on dedicated hardware and limited multi-tenancy rather than production environments.

3.2 The need for linearizable reads

Many large-scale services require that reads reflect the latest committed value. Systems such as Spanner [31], CosmosDB [3], and Dropbox [81] provide linearizable reads. Such guarantees are also essential in domains like financial trading [19] and ad bidding [72]. Stale reads introduce subtle, hard-to-diagnose anomalies, such as stale permissions in Zanzibar [46], outdated metadata in Presto [38], or inconsistent features in TFX [14]. Similarly, our internal governance and query routing systems require these guarantees to prevent incorrect permission checks and session-dependent execution.

3.3 Storage is slow and costly for reads

Transactional stores (e.g., Spanner [31], CockroachDB [79], TiDB [44]) provide strong semantics but incur protocol overheads and serialization costs. This results in multi-millisecond tail latency and high variability, making them unsuitable for latency-critical paths. Storage is also costly at high request rates, offering limited throughput and requiring significantly more compute than in-memory caches [62]. Consequently, services offload read-heavy workloads: Facebook's Memcache fleet handles massive latency-sensitive volumes [67], and TAO [18] optimizes for social graph lookups. Serving such traffic directly from storage would impose prohibitive operational costs.

3.4 Caches are fast but typically not consistent

Distributed caches reduce latency and cost by serving hot data with sub-millisecond performance and high throughput [18, 67]. However, they provide only eventual consistency: values become stale immediately after a write and are later refreshed via timeout (TTLs), polling, or explicit invalidation. This occurs because remote lookaside caches (e.g., Memcached) sit off the write path, while linked caches see only local writes unless ownership is coordinated across pods. As a result, achieving low latency, linearizable reads, and high availability together remains difficult.

4 The Delayed-Writes Anomaly

To achieve low-latency linearizable reads, we consider a write-through in-process cache, loosely coupled with the storage system, in which applications issue both reads and writes through the cache. An auto-sharder provides ownership over keys (§2.3.1). However, despite this exclusive ownership, a subtle but important anomaly can still occur, causing the pod to serve stale reads:

> **Delayed-write anomaly.** A delayed write for key K is a write that commits in the storage system after a new owner has already assumed ownership of K *in storage*.

Ownership at the caching layer is not enough. A pod must also become the owner in the storage layer before assuming that no other writes can succeed. Otherwise, a write from a previous owner may commit after ownership has changed at the caching layer, and cause the cache to return stale values. Preventing delayed writes is therefore essential for any strongly consistent cache.

4.1 Example: Delayed writes

We now show a resharding scenario in Figure 5 where the delayed-writes problem can occur. The sequence unfolds in the following steps:

1. Pod P_1, which has exclusive ownership over slice $[C_{40}, C_{60})$ issues a write with an OCC check: "Write key C_{45} with version V_2 if its current version is V_1."
2. The auto-sharder reshards $[C_{40}, C_{60})$, assigning $[C_{40}, C_{50})$ to pod P_0 and $[C_{50}, C_{60})$ to P_2 .
3. Pod P_0 reads C_{45} from the database and observes the latest committed version V_1.

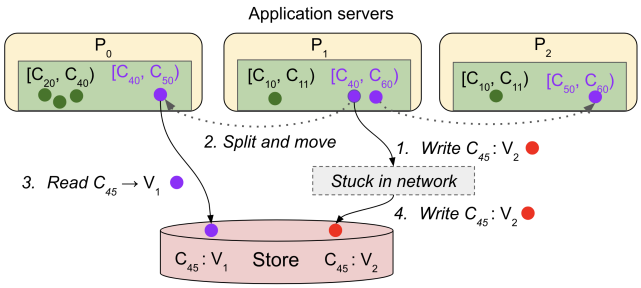


Figure 5: A delayed write of Write(C_{45},V_2) that comes in after P_0 has read version V_1 of C_{45} and cached it in memory.

4. The delayed write from P_1 arrives, passes the version check, and commits version V_2 for C_{45}.

The store now contains V_2, yet P_0 may still cache and serve V_1, violating linearizability. P_0 assumes exclusive ownership of key C_{45} when caching V_1, but this is incorrect because it never established ownership in storage, allowing a delayed write from P_1 to commit. The same issue can occur after a pod restart in a statically sharded cache (e.g., via consistent caching), where a new instance rebuilds its cache but an old in-flight write later commits and leaves the cache stale.

Although seemingly rare, long in-flight delays are well documented. Large systems suffer network anomalies where packets linger for minutes due to corrupted NICs or aggressive TCP retransmission backoffs [12]; GitHub similarly reported packets delayed for ninety seconds [82]. Delays need not be network-related: stop-the-world GC pauses and runtime stalls [49, 77] can last minutes in production workloads [58], and a buffered write may be released only after ownership has moved. In comparison, key reassignment is fast (e.g. 20ms). These real-world anomalies confirm that strongly consistent caches must explicitly guard against delayed writes.

4.2 Existing delayed-write prevention methods

Delayed writes can be prevented, but current mechanisms are slow, inefficient, or unscalable.

4.2.1 Version check on every read

One option is verifying each read against storage before serving, e.g., in Figure 5, the pod will not return C_{45}'s stale value if V_2 is already committed in storage. This ensures correctness but forces a remote version check on every read, adding latency and load which significantly reduces caching benefits [63].

4.2.2 Convert reads into metadata-writes

Alternatively, the first read for each key after ownership transfer can perform a write to advance the version (e.g., P_0 bumps C_{45} to V_1'), ensuring prior delayed writes fail their OCC check. However, this places a write on the critical read path, increasing latency and storage load.

4.2.3 Sequencer-based fencing in Chubby

Another approach to preventing delayed writes uses a lease management system such as Chubby [20] in conjunction with a storage system such as Bigtable [26]. Chubby provides a mechanism called sequencers. At any time, an application holding a lease can request the lease's sequencer, an opaque byte string that captures the state of the lock immediately after acquisition.

When an application performs a Bigtable write, it can attach a sequencer. Bigtable can then use the sequencer to verify that the application still holds the lease or that the sequencer is newer than the most recent one observed by the server. However, this approach requires application servers to acquire leases for every key they cache and requires the storage system to maintain per-key lease or sequencer state. In large multi-tenant deployments, applications may cache tens or hundreds of millions of keys, making this approach prohibitively expensive for the lease service, application servers, and storage backend. Consequently, sequencer-based fencing does not scale well for fine-grained caching systems. In our experience, this usage pattern was uncommon in production and discouraged by Chubby SREs because of the operational overhead it imposed. Additionally, it tightly couples the storage system and lease service by introducing an external dependency on the write path.

4.2.4 Synchronized clocks (e.g., TrueTime)

One possible solution to the delayed writes problem is to leverage synchronized clocks such as TrueTime in Spanner [31]. In this design, the storage system must provide timestamp-based write fencing, rejecting writes whose timestamps exceed a configured threshold T. When ownership of a key changes, the new owner waits for T seconds before serving requests, ensuring that delayed writes from the previous owner can no longer be accepted.

In this approach, choosing T involves a tradeoff. A value that is too small may reject legitimate writes under transient delays, while a value that is too large increases the unavailability period during ownership transitions. Moreover, the approach depends on specialized clock synchronization hardware that is often unavailable in public cloud environments, making it unsuitable for many general-purpose deployments.

4.2.5 Approximate correctness via timeouts

Given the rarity of unplanned pod failures, one might consider coordinating ownership transfer directly with the source pod. Under this approach, a slice is not reassigned until the source pod has drained all outstanding writes and explicitly acknowledges that ownership can be transferred.

While appealing in the common case, this design tightly couples the autosharder to application-specific logic and requires additional mechanisms to ensure liveness when the source pod crashes, becomes unreachable, or fails to respond. One possible approach is to assume bounded write delays and delay ownership transfer until the worst-case write latency has elapsed. Unlike leases, however, which derive their correctness from hardware-based bounds on local clock rates, this approach relies on a bound on write latency as measured across machines. Since no such bound exists in an asynchronous distributed system, the approach cannot provide formal correctness guarantees and instead only decreases the likelihood of stale writes.

5 The WriteGuard Primitive

We now present WriteGuards, which enable a loosely coupled cache to support linearizable cached reads. It does so by allowing the owning pod upon assignment of a new slice to establish ownership of the corresponding key range in storage. Unlike per-key approaches, WriteGuards scale to handle applications with millions to billions of keys efficiently by operating on key ranges. This requires only two storage operations: installing a guard and checking incoming writes against it.

5.1 WriteGuard installation

The storage system exposes a *SetGuard(range, guardValue)* call that associates an opaque guard value (denoted by *WG*) with a key range. If a newly assigned slice spans multiple tablets, the caller must split it into subranges and issue the SetGuard call to the appropriate tablet servers either through the client library or a higher-level layer. Figure 6 shows a case where CLINK splits the range $[C_{40},C_{60})$ into two calls to the relevant two tablets since C_{50} is a splitpoint: SetGuard($[C_{40},C_{50})$, WG_5) and SetGuard($[C_{50},C_{60})$, WG_6). *Splitpoints* or tablet boundaries can be discovered by periodically probing the storage layer using standard APIs available in existing systems [31,44]. An up-to-date view improves efficiency but is not required for correctness: if the tablet layout changes, servers return a status code indicating the change, and the caller can then obtain the new boundaries and retry the SetGuard calls.

Tablet servers maintain an in-memory interval map that associates non-overlapping key ranges with their guard values. This metadata is soft state and not persisted. On startup, a tablet initializes its full key range with an empty guard (lazy initialization is possible, but we assume eager initialization for clarity). When a SetGuard(*R*, *WG*) request arrives, the tablet intersects *R* with its owned ranges and updates the corresponding entries in the interval map, overwriting any existing guards in that region. In Figure 6, the two tablets install WG_5 on $[C_{40},C_{50})$ and WG_6 on $[C_{50},C_{60})$, while $[C_{30},C_{40})$ and $[C_{60},C_{80})$ keep their previous guards. Each tablet then acknowledges the request.

Stale splitpoint maps cause SetGuard calls to hit the wrong tablet server and return a *TabletLayoutChanged* error. CLINK catches this error to refresh the map and retry the call.

5.2 WriteGuards and tablet resharding

Storage-layer reshardings are relatively infrequent and typically triggered only under sustained load over minutes, since splitting or migrating tablets is expensive in systems such as Bigtable, Spanner, and HBase [21, 25, 31]. WriteGuards must be updated correctly during these operations.

When a tablet is split, each resulting tablet inherits the guard values of the original tablet. After a merge, the new tablet keeps the distinct guard values for the subranges contributed by each source tablet. (In practice, systems may need to bound the number of retained guards to avoid excessive metadata, though we did not explore this in our prototype.) For simplicity, our prototype drops guard metadata during tablet migrations rather than transferring it. This does not affect correctness, and we found that retaining guards across migrations offers negligible benefit relative to the complexity of moving guard state between tablet servers.

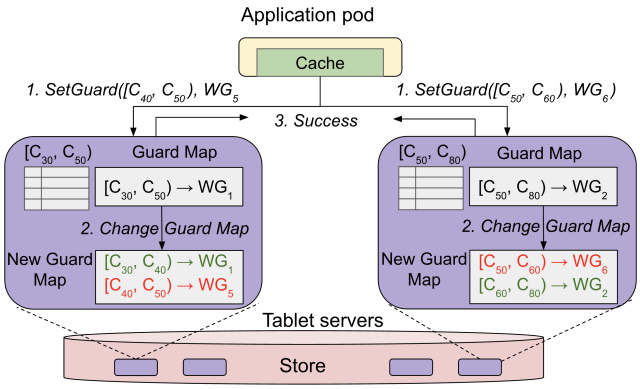


Figure 6: Installing a WriteGuard across two tablets. Upon receiving two slices, $[C_{40},C_{50})$ and $[C_{50},C_{60})$, the application pod generates two guard values (WG_5, WG_6) for each slice, and issues SetGuard to two tablets based on the cached splitpoints of tablets at the tablet servers.

5.3 Checking writes with WriteGuards

The storage system is also extended to validate a write's supplied WriteGuard against the installed WriteGuard for the key's corresponding range. A tablet server accepts writes only if the guards match, thereby rejecting delayed requests from previous owners. For example, in Figure 6, the tablet server would reject a delayed write for C_{45} carrying WG_1 because the WriteGuard in storage has been updated to WG_5. This design also ensures backward compatibility. Applications not using WriteGuards omit guard values, resulting in matching empty guards that trivially pass validation. This allows the storage system to enforce checks uniformly without requiring application changes.

5.4 Example: Preventing delayed writes

We now revisit the example from Section 4.1 to illustrate how WriteGuards prevent delayed writes. In that scenario (Figure 5), pod P_1 issued a write that updated key C_{45} from version V_1 to V_2. The write became delayed in the network while the slice $[C_{40},C_{60})$ was resharded, causing the subrange $[C_{40},C_{50})$ to be reassigned to pod P_0.

Under our approach, P_0 first installs a new WriteGuard, W_{new}, on the tablet server(s) responsible for the reassigned range and then reads key C_{45}, observing version V_1. Later, when the delayed write carrying version V_2 arrives at the tablet server, the write is rejected because its WriteGuard, W_{old}, does not match the currently installed WriteGuard, W_{new}. As a result, the stale write cannot overwrite data owned by the new pod.

6 CLINK: Serving linearizable cached reads

In this section we present CLINK, an auto-sharded cache linked directly into application pods that scales to handle millions to billions of keys. It provides linearizable reads while remaining loosely coupled to storage. CLINK allows reads to access storage from any pod to preserve high availability, but only pods assigned the key may serve reads directly from their local cache. CLINK uses WriteGuards to avoid delayed writes and scales by operating on key ranges rather than individual keys. To ensure linearizable cached reads, CLINK maintains the following invariant:

> **Latest State Invariant (LSI).** A key *K* may be served from the cache only if it equals the latest committed value of *K* in storage.

This invariant makes the cache transparent to applications,

since serving from memory is indistinguishable from serving from storage. A direct consequence is: when a pod issues a write on K, its cached value cannot be returned until that write has fully committed.

Note that we focus solely on LSI and linearizable reads. All other aspects of storage correctness are handled by the storage system's own transactional or consistency model, which operates independently of the mechanisms we introduce. For clarity, we defer optimizations such as buffering repeated reads and rejecting writes during a SetGuard call to §6.6. The remote variants of CLINK is discussed in §7. Detailed steps for various aspects of ownership, reads and writes are presented in Algorithms 1, 2 and 3.

6.1 Types and data structures

Each CLINK pod uses two main types. The first, *GuardHandle*, tracks a WriteGuard, the key range it covers, and the SliceHandle for that range. The second, *OpHandle*, captures the GuardHandle in effect when a request was issued to the store, the operation type (*RD* or *WR*), and a boolean flag *writeOverlap* indicating if any write overlapped with that operation.

```
GuardHandle = (KeyRange, WriteGuardID, SliceHandle)
OpHandle = (opType, writeOverlap, GuardHandle)
```

Each CLINK pod maintains three data structures. The *CacheMap* stores cached values. The *GuardMap* maps WriteGuard ranges to their associated GuardHandles and supports lookups by key or by range. The *OpMap* tracks all pending operations to storage. Together, these structures enable CLINK to serve linearizable reads while being auto-sharded. We write GuardMap[K] to denote a lookup of the GuardHandle for key K.

```
GuardMap : KeyRange → GuardHandle
CacheMap : Key → (GuardHandle, Value)
OpMap : Key → Array[OpHandle]
```

6.2 Preventing delayed writes via WriteGuards

We now describe how WriteGuards are installed on ownership changes via Algorithm 1 and how they prevent delayed writes.

6.2.1 Handling ownership change

Upon receiving a new assignment, CLINK acquires a SliceHandle for each newly assigned slice S, intersects S with the tablet ranges, and generates a unique WriteGuard value (e.g., a UUID) for each intersected sub-interval R_i (lines 7–9). These sub-ranges are precisely the key ranges in the storage layer that must be protected from delayed writes before CLINK can assume ownership of S. CLINK periodically refreshes tablet split points from the store to keep this mapping current. For each R_i, the pod issues a SetGuard call and installs the resulting GuardHandle in the GuardMap if the call succeeds and ownership has remained continuous (lines 11–13) – checking whether the slice S has been assigned to the current pod since the SliceHandle's creation. Figure 8 illustrates this process for a newly assigned slice $[C_{40}, C_{60})$, resulting in two SetGuard calls for the tablet ranges $[C_{40}, C_{50})$ and $[C_{45}, C_{60})$.

If the store reports a tablet-layout change, CLINK refreshes its cached split points and restarts the procedure (lines 14–15). If a SetGuard call times out, it is retried with a fresh unique WriteGuard value (lines 16–17). If ownership has been lost, CLINK aborts creation of the GuardHandle for R_i.

Algorithm 1 Assignment Handling (Red means critical section)

```
 1: procedure HandleNewAssignment(Asn)
 2:   for each new/removed slice X in Asn do
 3:     Clear GuardMap[X]
 4:   for each new slice S in Asn do
 5:     HandleNewSlice(S)
 6: end procedure

 7: procedure HandleNewSlice(S)
 8:   for each range R_i in intersection of S and store splitpoints do
 9:     Obtain SliceHandle SH_i
10:     Create unique WriteGuard WG_i
11:     status ← SetGuard(R_i, WG_i)
12:     if IsStillAssigned(SH_i) then
13:       if status == Success then
14:         GuardMap[R_i] ← GuardHandle(WG_i, R_i, SH_i)
15:       else if status == TabletLayoutChanged then
16:         Refetch store splitpoints and retry from line 8
17:       else                          ▷ SetGuard call had a timeout
18:         Retry from line 10 with a new WriteGuard
19: end procedure
```

6.2.2 WriteGuard ordering

CLINK enforces the following rules on SetGuard calls:

1. **Ownership-contained.** CLINK creates GuardHandles only for WriteGuards whose SetGuard calls both started and finished within a single ownership period.
2. **Non-overlapping.** Within any ownership period and for any key K, CLINK allows at most one outstanding SetGuard call whose range covers K.
3. **Unique-valued.** Every SetGuard call issued by CLINK (including retries) uses a fresh WriteGuard value.

Figure 7 illustrates these rules with four GuardHandles, G_1 through G_4, created across three exclusive ownership periods E_1, E_2, and E_3 established by leveraging the strong ownership

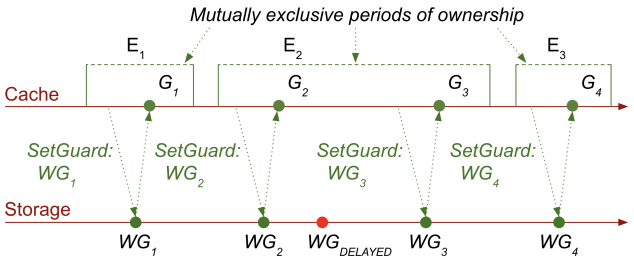


Figure 7: Relevant WriteGuards (green) are ownership-contained and non-overlapping.

guarantees of the auto-sharder (§2.3). Each SetGuard call installs a unique WriteGuard value (WG_1, WG_2, WG_3, WG_4, and the delayed $WG_{DELAYED}$ described later). The GuardHandles are created only when the corresponding SetGuard call both begins and ends within the same ownership interval. In ownership interval E_2, the calls that produce G_2 and G_3 are non-overlapping. In contrast, $WG_{DELAYED}$ comes from a SetGuard call that either timed out or returned so late that it violated the ownership containment rule and thus has no GuardHandle.

We next show how the first two rules induce, for each key, a total order on GuardHandles that aligns with the per key installation order of their WriteGuards in storage. The importance of the last rule is discussed in the next section.

First, because ownership periods are totally ordered by the auto-sharder, the *ownership-contained* rule ensures that any GuardHandle created in a later ownership period follows all GuardHandles from earlier ones. Further, each WriteGuard is installed between the start and end of its SetGuard call, and that call lies entirely within a single ownership period, so the WriteGuards inherit the same total order. In the figure, WG_2 and WG_3 (for G_2 and G_3) follow WG_1 (for G_1).

Second, within a single ownership period, the *non-overlapping* rule ensures that each SetGuard call must finish (or time out) before the next one begins. Again because each WriteGuard is installed at some moment during its SetGuard call, the WriteGuards appear in storage in the exact order of those calls. The corresponding GuardHandles are created in the same order (for example, WG_3 appears after WG_2).

Therefore, we have the following property:

WriteGuard Ordering. All GuardHandles form a total order, and their corresponding WriteGuards in storage follow the same order. We call the WriteGuards associated with these handles *relevant*. All others are *irrelevant*.

6.2.3 Why relevant WriteGuards prevent delayed writes

Relevant WriteGuards prevent delayed writes since once a new owner creates a GuardHandle, its relevant WriteGuard is installed strictly after all relevant WriteGuards from prior owners. Furthermore by the *unique-valued* rule for WriteGuard values (rule 3), all WriteGuards in storage (relevant or not) are unique. Thus any delayed write from a previous owner has a WriteGuard that cannot match the new owner's relevant WriteGuard, so such writes are rejected. This guarantees that all future reads by the new owner are protected.

6.3 Handling writes and reads

We now present the handling of write (Algorithm 2) and read operations (Algorithm 3).

6.3.1 Write operations

Upon a write to K, CLINK rejects the request if the WriteGuard is missing or ownership is lost (lines 2–6). The latter is an optimization, as the write would likely fail anyway due to the new owner's WriteGuard. CLINK then evicts K to preserve LSI, as the pending write renders the storage value unknown. Finally, it flags pending operations as overlapping (lines 8–9) and sets the request's *writeOverlap* bit if another write is in flight (line 10). Figure 8 phase (b) shows the case of a write for C_{43} which overlaps for a read for the same key and thus sets the read operation's *writeOverlap* bit, causing the result of the read to be uncacheable (see §6.3.3).

Algorithm 2 Write Operation (Red means critical section)

```
1:  procedure WRITE(K, V)
2:    if GuardMap[K] is empty then
3:      return error              ▷ Reject: no WriteGuard protection
4:    guardHandle ← GuardMap[K]
5:    if not ISSTILLASSIGNED(guardHandle.SliceHandle) then
6:      return error              ▷ Reject: lost continuous ownership
7:    Evict CacheMap[K]           ▷ Writes invalidate the cache
8:    for each op X in OpMap[K] do
9:      X.writeOverlap ← true     ▷ Prior ops overlap w/ write
10:   writeOverlap ← (OpMap[K] contains write?)
11:   opHandle ← OpHandle(WR, writeOverlap, guardHandle)
12:   Add opHandle to OpMap[K]
13:   status, result ← SENDWRITE(K, V)
14:   if status == Success then
15:     CacheMap[K] ← (guardHandle, result)
16:     if not SATISFIESLSI(opHandle) then
17:       Evict CacheMap[K]  ▷ Unsafe value, LSI not guaranteed
18:     return result
19:   else if status == Timeout || WriteGuardMismatch then
20:     if not ISSTILLASSIGNED(guardHandle.SliceHandle) then
21:       return error            ▷ Reject: lost continuous ownership
22:     Clear GuardMap[K]
23:     Install new GuardHandle
24:     Restart at step 1                ▷ retry with fresh guard
25:   else
26:     return error
27: end procedure
```

CLINK then executes the storage write. Successful results are cached only if they satisfy the LSI (lines 15–17). A timeout (line 19) signals a potential self-inflicted delayed write. To resolve this, or if a delayed WriteGuard error occurs, CLINK clears the GuardHandle, installs a fresh guard, and retries if it still owns K (lines 20–24).

6.3.2 Read operations

Read handling differs from writes because reads may be handled by any pod. When a pod owns K and the key is present in the cache, it may serve the cached value as long as ownership has been continuous since the entry was installed (lines 2–5 of the READ procedure). Figure 8 shows a case where C_{45} was read by a pod from storage and later served from the cache since it has continuous ownership and no writes have invalidated the cached value.

If a continuously owned value was not found, CLINK proceeds by evicting any entry if present (line 6) and reads from storage, setting *writeOverlap* if needed (lines 7–11). The result is returned to the caller and cached if it satisfies LSI.

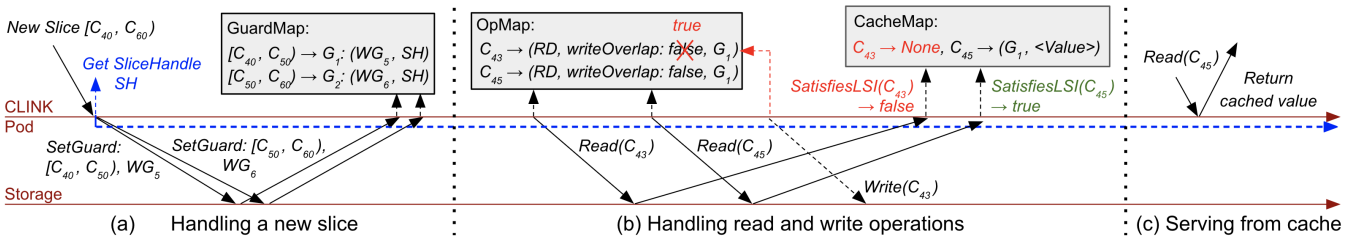


Figure 8: Life of a slice assignment and serving reads and writes in CLINK. Details described in the text.

Algorithm 3 Read Operation (Red means critical section)

```
1:  procedure READ(K)
2:    if CacheMap[K] not empty then
3:      value ← CacheMap[K].value
4:      sliceHandle ← CacheMap[K].GuardHandle.SliceHandle
5:      if ISSTILLASSIGNED(sliceHandle) then return value
6:    Evict CacheMap[K]        ▷ invalidate any stale cached value
7:    writeOverlap ← (OpMap[K] contains write?)
8:    guardHandle ← GuardMap[K] or null
9:    opHandle ← OpHandle(RD, writeOverlap, guardHandle)
10:   Add opHandle to OpMap[K]
11:   result ← SENDREAD(K)
12:   CacheMap[K] ← (guardHandle, result)
13:   if not SATISFIESLSI(opHandle) then
14:     Evict CacheMap[K]      ▷ Unsafe value, LSI not guaranteed
15:   return result
16: end procedure
```

6.3.3 Cacheability

CLINK caches results only if they satisfy LSI. It does this by ensuring no storage mutations occurred during execution. SATISFIESLSI (Algorithm 4) uses the OpHandle to rule out three mutation sources: (1) *previous owners*, by checking for WriteGuard protection (line 3); (2) the *current owner*, by ensuring no writes overlapped (line 4) which is checked with *writeOverlap* and (3) *future owners*, by verifying continuous ownership since WriteGuard installation (line 6). If these conditions hold, the value matches storage and is safe to cache.

Figure 8 phase (b) shows SATISFIESLSI returns false for the read of C_{43} since it was concurrently mutated by the current owner, but SATISFIESLSI returns true for the read of C_{45} (in phase (c)) as it was free of mutations from previous, current, or future owners (was continuously owned).

Algorithm 4 Check for cacheability (Red is critical section)

```
1: procedure SATISFIESLSI(opHandle)
2:   guardHandle ← opHandle.GuardHandle
3:   if guardHandle == null then return false      ▷ prev owner
4:   if opHandle.writeOverlap then return false    ▷ current owner
5:   sliceHandle ← guardHandle.SliceHandle
6:   return ISSTILLASSIGNED(sliceHandle)           ▷ future owner
7: end procedure
```

6.4 Why CLINK serves linearizable reads

In this section, we demonstrate that CLINK guarantees linearizability. Since direct storage reads inherit storage guarantees, we focus on proving that cached reads satisfy LSI. CLINK achieves this by enforcing two invariants:

1. **LSI-at-entry:** The value satisfies LSI upon insertion into the cache.
2. **LSI-at-serve:** The value remains unchanged in storage between insertion and serving.

Algorithms 2 and 3 enforce LSI-at-entry by caching the operation's return value and immediately checking LSI (with `SatisfiesLSI`, Algorithm 4), evicting the entry if the check fails.

To guarantee LSI-at-serve, CLINK employs two mechanisms. First, it evicts cached values on every local write (Algorithm 2, line 7), ruling out mutations by the *current owner*. Second, it verifies continuous ownership before returning a cache hit, ruling out mutations by *future owners* (if the ownership is temporarily transferred to a different owner after the read initiates, the key will not be continuously owned when it returns). Mutations by *previous owners* are already precluded by LSI-at-entry (via WriteGuards). Consequently, any value returned from the cache matches storage, ensuring linearizability.

6.5 Serving reads from multiple pods for scale

To support high-traffic keys, CLINK employs asymmetric replication (§2.3.1). A designated primary handles all writes and maintains the WriteGuard; upon leadership change, a new WriteGuard is installed to reject delayed writes.

Initialization and Ownership: Replicas initialize ranges as *uncacheable*. Once the primary installs an ownership-aligned WriteGuard, it notifies replicas to mark ranges *cacheable* and provides identifiers for the current ownership epoch so they can acquire the correct SliceHandles. Serving reads requires *continuous group ownership*: all pods in the replica set must remain assigned without interruption. Any membership change (notified by the autosharder and detected via SliceHandles) breaks this invariant, forcing replicas to evict cached entries, though reads may proceed directly to storage.

Write Protocol: Writes utilize a two-phase invalidation process. (1) The primary sends invalidation messages; replicas evict the key K and mark it uncacheable. (2) Upon receiving all acknowledgments, the primary writes to storage. (3) The primary sends a finish message (optionally piggybacking the new value), allowing replicas to restore K's cacheability.

Failure Handling: If a write fails, the finish message restores cacheability. Timeouts trigger a new WriteGuard installation and retry (§6.3.1). If membership changes mid-protocol, Assignment Consistency (§2.3.1) ensures replicas detect the loss of group ownership and drop the corresponding ranges and mark the range uncacheable. Because resharding is infrequent, this simple rule is sufficient.

This design significantly increases read throughput and availability. While invalidating replicas adds slight write latency (§9.5), the impact on write availability is negligible due to the rarity of crashes (§2.3.2) and the auto-sharder's rapid removal of unhealthy pods (§2.3.1).

6.6 Potential Optimizations

CLINK admits several future optimizations. First, request gating can buffer repeated reads on the same key so that only one storage fetch is issued. Second, storing value versions in the cache would enable finer cachability decisions. For example, if concurrent writes produce v_8 and v_9 while the cache holds v_7, CLINK currently caches neither; checking versions would allow safely caching v_9. Third, when a write fails, CLINK could install a new WriteGuard for only the affected key instead of the entire key range, though this risks creating many entries in the GuardMap. Finally, tagging reads with their WriteGuard would allow early detection of guard changes in storage (e.g., after a delayed SetGuard call), enabling timely refreshes.

7 CRINK: Strongly consistent remote caches

While CLINK provides linearizable reads at local memory latency, there are cases where a remote cache is the better fit. First, some environments cannot host stateful components because of serverless or stateless runtimes, memory limits, or operational restrictions. Second, service owners generally avoid exposing internal schemas, which means any shared dataset must be served through the owning service's API, forcing that service to scale just to proxy reads. Third, some workloads do not shard cleanly. A single request may touch multiple keys, and forwarding these fanout RPCs through the owning service is undesirable [83].

A remote cache sidesteps these issues. The owning service retains exclusive control of writes and schema evolution while the cache absorbs read traffic, improving latency, increasing read throughput, and reducing demand on storage. A platform team can even operate this tier as shared infrastructure, though the owning service must grant it controlled read/write access.

We present a remote cache CRINK (Consistent Remote In-memory Key-value cache) with two possible architectures, CRINK-R and CRINK-L. Like other remote caches they cannot match CLINK's performance because each access incurs network transfer, serialization overheads, and overreads [8]. They do, however, provide strong consistency, which traditional lookaside caches do not.

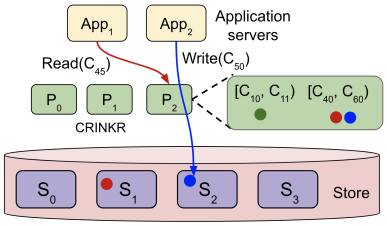


Figure 9: CRINK-R: Remote deployment of CLINK. CRINK-R is a sharded strongly consistent remote write-through cache.

7.1 CRINK-R

CRINK-R (CRINK - Regular) deploys the CLINK architecture as a standalone caching service using a write-through pattern. It retains the auto-sharder and WriteGuard mechanisms to enforce ownership and consistency. Writes flow through CRINK-R to storage, and subsequent reads are served from memory. This setup runs CLINK out-of-process and allows downstream services to share a strongly consistent cache. Figure 9 shows an application writing key C_{50} through CRINK-R and reading C_{45} from the cache.

7.2 CRINK-L

CRINK-L (CRINK - Layered) decouples consistency metadata from value storage (Figure 10). A lightweight *version service* maintains auto-sharded assignment consistency while values reside in a generic lookaside cache like Redis that need not be consistent.

For reads the client queries both the version service and the cache in parallel. Once both fetches complete, if the versions match then it returns the cached value, as with C_{45} in Figure 10. If they differ it falls back to storage and may refresh the cache. For writes the client commits through the version service and pushes the value to the cache asynchronously, as with C_{50} in Figure 10. Any delay in this propagation is harmless because a version mismatch always triggers a storage read. A small client library can hide these details from applications.

CRINK-L improves on the monolithic CRINK-R design by enabling *independent scaling*, *layered consistency*, and *flexible consistency*. Independent scaling lets the value cache (e.g., Redis) replicate for scale while a separate service supplies small consistent versions. Because the approach is layered, it can add linearizability to existing eventually consistent caches. It is flexible since it allows applications to read from the cache for eventual consistency or consult the version service for

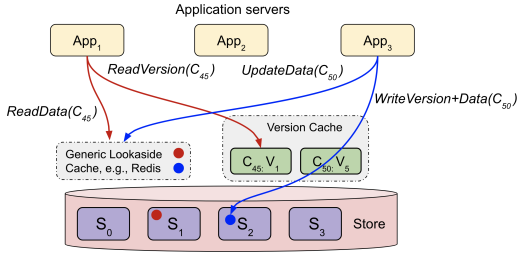


Figure 10: CRINK-L: The version service maintains consistent versions, while values reside in an ordinary lookaside cache.

linearizability. Note that Chrono offers similar capabilities but has several issues including temporary blocking of cached reads after each write (§4.2).

8 System implementation

Auto-sharder with strong ownership: We implemented an auto-sharder with strong ownership, inspired by Centrifuge [7]. When an assignment changes, the previous owner explicitly relinquishes its lease to ensure consistent transitions. The auto-sharder supports asymmetric replication as discussed in §2.3.1. It guarantees that all pods agree on the assignments; it supports the API to check continuous assignment of a key at the application pod without coordination with the auto-sharder. The assigner logic is implemented in about 12000 lines of Scala.

Cache: The cache is implemented in approximately 6000 lines of C++. We implemented the protocols and the data structures described in §6, i.e., GuardMap, CacheMap and OpMap. Storage splitpoints are obtained by periodically probing the underlying storage system (TiDB). We implement CLINK, CRINK-R, and CRINK-L as described above.

Storage: We used TiDB [44] as a representative distributed database. TiKV is the storage engine that TiDB uses. We add a new API (SetGuard) to TiDB (Version 7.5.1). The TiDB placement driver (PD) is responsible for splitting and migrating key ranges that are stored on TiKV. We modified TiDB (↑ 717 lines of Rust for TiKV, ↑ 229 lines of Go for TiDB SQL layer, and ↑ 57 lines of Go for PD) to add the support for WriteGuard installation and checking. WriteGuards are implemented as in-memory state, and are cleared upon storage pod restart or failure without affecting correctness. The entire TiDB modification took roughly 1 person-month, showing that changes are fast and lightweight.

9 Evaluation

Our evaluation addresses four questions:

- **Per-core throughput:** Does CLINK scale per-pod? *Our unoptimized prototype reaches about 1M ops per core.*
- **WriteGuard overhead:** What latency and throughput cost do WriteGuards introduce? *WriteGuards add negligible overhead.*
- **Read performance:** How do CLINK and CRINK compare to other strongly consistent caches? CLINK *reduces read tail latency by three orders of magnitude, and* CRINK *improves tail latency by* 2.2–2.4×.
- **Multi-replica writes:** What is the cost of serving a hot key from multiple pods? *Additional replicas introduce modest coordination overhead on writes.*

9.1 Methodology

Experimental setup All experiments run on production clusters. For storage, we use 9 TiKV pods, 6 TiDB pods, and 3 PD pods. Each TiKV pod is provisioned with 15GB of memory and 30 vCPUs. We use 6 pods for application servers, each is provisioned with 16 vCPUs and 16GB of memory. As is the case in most prior work [10, 36, 37, 47, 54, 60, 67, 68, 85, 91, 98–100], we measure latency from the application server's perspective which is using the cache. The internal cluster runs on cloud networks with single-digit-millisecond latency.

Workloads We evaluate CLINK and CRINK using public Meta [5, 15] and Twitter [88] traces, plus an internal Unity Catalog (UC) production trace. While Meta and Twitter are simple key-value workloads, Unity Catalog reflects a complex serving workload where each request requires 6–8 storage fetches to compose the value. The traces differ significantly in read-write ratios and object sizes: Meta is ≈70% reads with ≈ 10B median values; Twitter varies from read-heavy (82–99%) to write-heavy (31–50%) with 230B values; and Unity Catalog is read-heavy (99.4%) with large ≈ 23KB values. We present performance comparisons across all three traces in §9.4.1. Due to space constraints, the remaining focused evaluations use only Unity Catalog and synthetic workloads.

Baselines We compare our linked (CLINK) and remote (CRINK-R, CRINK-L) caches against several baselines. Chrono [81] (§4.2) uses the 5s attempt timestamp given in [81]. *Version* performs a storage version check on every read, while *Storage* accesses the underlying store directly.

9.2 CLINK throughput

We measured CLINK's throughput as we increased the number of cores. Figure 11a shows that CLINK scales near linearly, sustaining approximately one million read operations per second per core, reaching 22.8 million QPS at 24 cores. Because all reads come directly from memory and require no cross-machine coordination, throughput grows proportionally with available CPUs.

Our throughput results are not intended to represent state-of-the-art cache performance. The prototype focuses on validating the WriteGuard primitive rather than maximizing raw efficiency, so it omits optimizations such as lock-free indexing, advanced eviction strategies, CPU-cache-optimized data layouts, and kernel-bypass or co-designed network stacks [10,24,57,60,61,91]. Even without these techniques, CLINK achieves reasonable CPU performance, and the experiments below show that caches using WriteGuards can provide practical and efficient linearizable reads without the penalties of prior approaches.

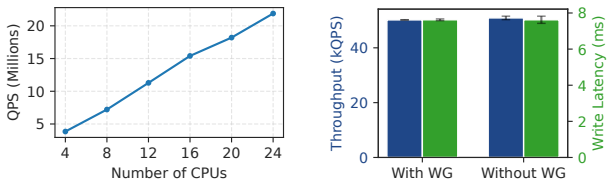


Figure 11: (a) Throughput scalability of CLINK over increasing number of CPU cores (b) Overhead of WriteGuards (WG) on the storage layer, comparing both the throughput (kQPS) and the write latency (ms).

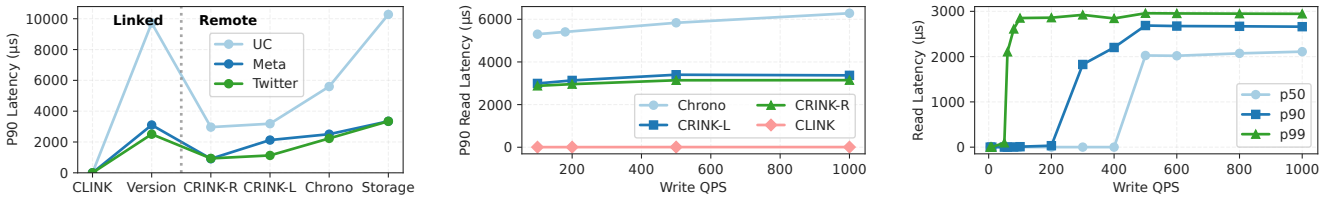


Figure 12: Read latency analysis. (a) P90 read latency comparison across Unity Catalog (UC), Meta, and Twitter traces, comparing both the linked and remote deployments. (b) Performance comparison of CLINK against baselines with increasing write QPS over Unity Catalog traces. We adjust the write QPS by converting reads to writes in the trace. (c) CLINK read latency over a synthetic workload of increasing write QPS on *a single key* to stress test the impact on read latency.

9.3 Overhead of WriteGuards

To quantify the overhead of WriteGuards in the storage layer, we measured throughput and latency under a write-only workload designed to maximally stress the WriteGuard mechanism. Figure 11b presents the throughput and latency results at 75% storage layer CPU utilization. We observe that neither throughput nor latency is adversely affected by WriteGuards, as they introduce only a lightweight conditional check. Furthermore, SetGuard operations occur per *assignment* change (which is infrequent, as shown in Figure 14) rather than per write, thereby imposing minimal overhead on the storage layer.

9.4 Read latency evaluation

We now compare CLINK and CRINK against existing solutions, and analyze how write traffic and resharding affect read latency.

9.4.1 Performance on production traces

Figure 12a summarizes steady-state read latency. We omit P_{50} latency, as it follows a similar trend, and P_{99} latency, which primarily reflects cache misses. Across all systems and percentiles, CLINK demonstrates the lowest latency (P_{90} of 0.5μs–4.2μs), remaining well below remote caches and significantly outperforming Storage (P_{90} of 4.8ms–10.3ms). Version (P_{90} of 3.3 ms-9.0 ms) offers only marginal improvement over Storage, as the latency is dominated by the overhead of accessing storage. By storing values in the application address space and leveraging WriteGuards to bypass coordination on the read path, CLINK achieves latencies orders of magnitude lower than the alternatives.

For remote caching architectures, Figure 12a demonstrates that CRINK-R/CRINK-L (P_{90} of 0.65ms–3.2ms) outperform Chrono (P_{90} of 2.3ms–5.5ms) across all workloads. This performance gap arises because Chrono writes use an attempt timestamp 5s in the future (the default recommended in [81]), preventing reads from being served from the cache for that duration after every write. Our measurements show that Chrono has a lower hit rate due to this; consequently, its P_{90} latency partially reflects storage access times. Chrono also adds additional latency on every read and write path, by requiring a concurrent timestamp check to the remote Chrono service that tracks timestamp bound per-key, adding some variability to read latency.

9.4.2 Sensitivity to write traffic

We next measure read latency sensitivity to write traffic. Figure 12b varies write QPS for the Unity Catalog trace (by converting reads to writes in the trace), comparing CLINK against CRINK-R, CRINK-L, and Chrono. Latency remains stable as in-flight writes increase, except for Chrono, where frequent timestamp updates narrow the cacheable read window. Since writes are distributed and few keys see high write QPS, the overall impact on read latency is minor.

Figure 12c shows an adversarial microbenchmark for CLINK varying concurrent write QPS on a single key. Since CLINK prevents cache population during writes, latencies rise to storage levels (2ms–2.8ms) when hit rates drop below the target percentile. Specifically, P_{99} spikes at 80 write QPS, while P_{90} and P_{50} spike at 200 and 400 QPS. These thresholds significantly exceed observed production rates for write QPS on a single key.

9.4.3 Sensitivity to resharding

Figure 13a shows that even during resharding events (with 0.5% traffic churn), CLINK maintains 100% availability and low latency. This is because key assignment is adjusted quickly (< 20 ms, small enough for retries to succeed), which results in negligible impact on read latency and availability. Figure 14 shows the percentage of load moved per minute for load balancing purposes for three production services, indicating that churn is typically < 0.2%. This means that resharding and the subsequent SetGuard operations triggered for the newly assigned slices have negligible impact on read latency.

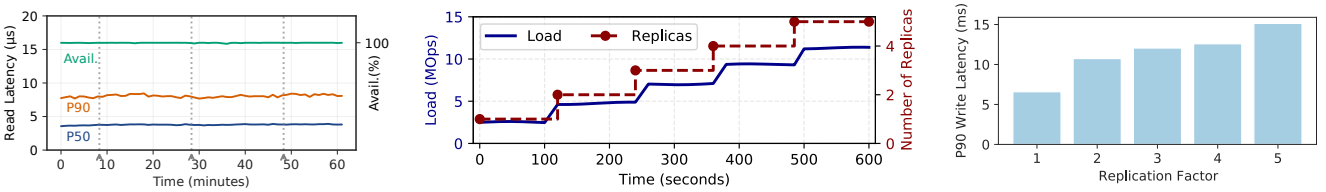


Figure 13: CLINK characteristics. (a) CLINK maintains high availability and low latency during resharding events. (b) the auto-sharder increases replicas as load increases. (c) more replicas increases write tail latency by a small amount.

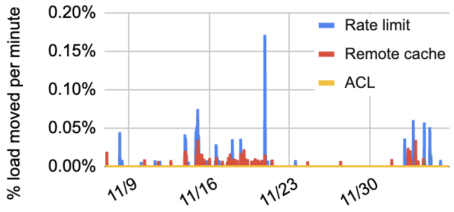


Figure 14: Minimal churn observed in practice. Figure shows the percentage of load moved per minute for load balancing over 30 days for 3 critical services.

Similarly, upon rolling restart, the auto-sharder can proactively migrate the slices to the new pod with state transfer between old and new pod [53], ensuring minimal impact on latency.

Depending on the workload, the storage layer (e.g., TiDB) may also undergo resharding (Figure 15). However, the impact on service availability and write latency is negligible, as WriteGuards are preserved across ownership changes (§6.2.1).

9.5 Asymmetric replication

CLINK manages hot keys by first *isolating* the target key on a single pod, then asymmetrically replicating it across additional pods when load exceeds the capacity of a single pod. Figure 13b demonstrates the auto-sharder dynamically adding replicas as the read QPS increases. Figure 13c shows that write latency increases only modestly with additional replicas due to coordination overhead. We artificially lowered pod capacity for this experiment, as triggering 5 replicas on standard 16-core pods normally requires 80M+ QPS on a single key. Note that though 5 replicas incurs some overhead for writes, in our experience such workloads are dominated almost entirely by reads.

10 Related Work

To the best of our knowledge, loosely coupled caches that provide linearizable reads of state in storage remain largely unexplored. We summarize the most relevant approaches below.

10.1 Strongly consistent caches

The primary systems that support linearizable cached reads are Chrono [81] and Chubby [20]. Chubby introduces tight coupling, and per-key leases limit scalability. Chrono is remote, requires an additional round trip on each operation, and blocks cached reads for a period after every write.

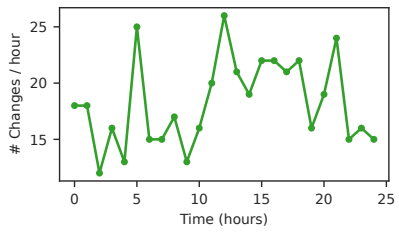


Figure 15: Frequency of resharding events in the TiDB storage layer. The changes capture either the addition or removal of at least one shard. This shows resharding happens over tens of minutes of granularity.

10.1.1 Chrono

Chrono [81] provides linearizable reads from a remote cache by tracking, for each key, an upper bound on the commit timestamps that could have been assigned by the storage system, and then using storage-provided read timestamps to determine whether a cached value is guaranteed to reflect the latest possible write. Chrono assumes an underlying storage system with two properties. First, the storage system must support specifying maximum commit timestamp bounds on writes. Second, it must return read timestamps on reads that respect transaction ordering and that track current time.

More specifically, to perform a write, a client first proposes a new maximum commit timestamp bound for the key to Chrono. If the proposed bound is greater than the current bound, Chrono accepts it and the client issues the write to storage tagged with that bound. The storage system processes the write only if it can assign the write a commit timestamp less than the provided bound.

The cache stores values read from storage together with their corresponding read timestamps. On a cache lookup, clients read both the cached value and the latest timestamp bound for the key from Chrono in parallel. If the cached value's read timestamp is greater than the latest bound returned by Chrono, the cached value is guaranteed to reflect all writes that completed before the read began, and is therefore linearizable. Otherwise, linearizability cannot be guaranteed from the cache and the read must be served by the storage layer.

This design has several drawbacks. First, unlike CLINK, the cache is remote, so all accesses incur remote cache latency. Second, clients must choose write timestamp bounds far enough into the future for writes to succeed, but larger bounds delay when storage reads become cacheable. For example, with the recommended threshold of 5 seconds, a write on a key K prevents cached reads of K for the next 5 seconds; any write rate higher than one every 50 seconds on a key yields a hit rate under 90%. Third, both reads and writes require an extra round trip to Chrono, adding latency and creating an additional availability dependency. Finally, the storage layer must support rejecting writes that exceed a maximum timestamp and must return read timestamps that are guaranteed to be greater than the last committed write. We believe these requirements are more invasive than our WriteGuard approach.

10.1.2 Synchronous invalidation (e.g. Chubby)

Another approach to strong consistency relies on reader leases and synchronous invalidation such as those provided by Chubby [20]. In such systems, a cache pod may serve a cached value only while it holds a lease for the corresponding key or object. Before committing a write, the storage system must synchronously revoke or invalidate all outstanding reader leases to ensure that no cache can subsequently serve stale data.

This approach tightly couples the cache and storage layers because the storage system must track cache ownership and coordinate invalidations across cache instances. It also scales

poorly for distributed caches serving millions to billions of keys, as the storage system must maintain lease state for every cached key. In discussions with Chubby production engineers, we learned that synchronous invalidations increased load on the service because writes were forced to wait for lease recalls and invalidation acknowledgements. Slow, stalled, or crashed clients could delay writes until lease expiration, leading to high write tail latency and reduced throughput. To the best of our knowledge, no modern *general-purpose* scalable storage system exposes distributed reader leases or synchronous invalidation as a client-facing primitive.

Note that CLINK also uses synchronous invalidation when a hot key is replicated across multiple pods (§6.5). However, unlike lease-recall schemes that apply broadly across all keys, this mechanism is used only for the small subset of hot keys that require replication, substantially reducing the amount of coordination required in practice. In addition, invalidations are exchanged directly between application pods, rather than requiring the storage system to participate in lease recall from application servers.

10.2 Storage-layer caches

Many storage systems natively support automatic block or row caching of popular tables and rows for fast lookups [32, 45, 56, 69, 73–75, 86]. While such caches provide linearizable reads, the storage layer quickly becomes a performance bottleneck [88, 92, 93], causing practitioners to explore solutions based on external caches.

10.3 Caches with weak read consistency

Many caches [11, 13, 16, 18, 23, 27, 28, 33, 40, 42, 48, 65, 71, 78, 87, 89, 92, 94] are based on TTLs, providing eventually-consistent and not linearizable reads. Cache invalidation [6, 22, 52, 67, 95, 97] has been explored. However, none of these caches provide linearizable reads. TxCache [70] provides transactional consistency for in process caches by validating cached objects against a central database, but it differs from our approach in several key aspects. First, it does not guarantee that reads return the latest committed value. Second, it is not designed for auto-sharded or dynamically rebalanced systems and therefore does not address correctness under shifting key ownership. Third, it targets a single relational database rather than a distributed, sharded storage layer.

11 Conclusion

This work introduced WriteGuards, a lightweight storage mechanism that prevents delayed writes from previous owners and enables loosely coupled caches to serve linearizable reads from memory. Using this primitive, we presented CLINK, the first in-process datacenter cache that provides linearizable cached reads with low latency, and CRINK-R and CRINK-L, two remote variants that offer the same guarantees for shared and multi-tenant deployments. Our caches are performant: CLINK reduces tail latency of reads by up to three orders of magnitude. CRINK-R and CRINK-L also significantly cut tail latency compared to direct storage access. These results show that linearizable reads at low latency are achievable in large-scale, auto-sharded datacenter services through a simple write-fencing abstraction. We argue that distributed stores should adopt WriteGuards to allow loosely coupled caches such as CLINK and CRINK to serve linearizable reads directly from memory. More broadly, supporting WriteGuards would open the door to a general ecosystem of strongly consistent caches layered on top of these stores.

Acknowledgement

We sincerely thank the anonymous reviewers and our shepherd for their insightful feedback. We also thank Ray Matharu for his assistance in obtaining production workload traces for Unity Catalog, and Ge Gao for his help with TiDB. This work is in part supported by gifts from Accenture, AMD, Anyscale, Broadcom, Cisco, Google, IBM, Intel, Intesa Sanpaolo, Lambda, Lightspeed, Mibura, Microsoft, NVIDIA, Samsung SDS, and SAP.

References

[1] Introducing Michelangelo: Uber's Machine Learning Platform. Uber Engineering Blog, 2017. https://eng.uber.com/michelangelo/.

[2] Scaling the TAO Social Graph System. Meta Engineering Blog, 2022. https://engineering.fb.com.

[3] Consistency Levels in Azure Cosmos DB. Microsoft Azure Documentation, 2023. https://learn.microsoft.com/azure/cosmos-db/consistency-levels.

[4] DynamoDB Performance Guidance. AWS Documentation, 2023. https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/Performance.html.

[5] Evaluating SSD hardware for Facebook workloads. https://cachelib.org/docs/Cache_Library_User_Guides/cachebench-fb-hw-eval, 2024. Accessed: 2024-06-22.

[6] Bahman Abolhassani, John Tadrous, Atilla Eryilmaz, and Edmund Yeh. Fresh caching for dynamic content. In *IEEE INFOCOM 2021-IEEE Conference on Computer Communications*, pages 1–10. IEEE, 2021.

[7] Atul Adya, John Dunagan, and Alec Wolman. Centrifuge: Integrated Lease Management and Partitioning for Cloud Services. In *NSDI*, volume 10, pages 1–16, 2010.

[8] Atul Adya, Robert Grandl, Daniel Myers, and Henry Qin. Fast key-value stores: An idea whose time has come and gone. In *Proceedings of the Workshop on Hot Topics in Operating Systems*, pages 113–119, 2019.

[9] Atul Adya, Daniel Myers, Jon Howell, Jeremy Elson, Colin Meek, Vishesh Khemani, Stefan Fulger, Pan Gu, Lakshminath Bhuvanagiri, Jason Hunter, et al. Slicer:{Auto-Sharding} for datacenter applications. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 739–753, 2016.

[10] Ahmed Alquraan, Sreeharsha Udayashankar, Virendra Marathe, Bernard Wong, and Samer Al-Kiswany. LoLKV: The Logless, Linearizable, RDMA-based Key-Value Storage System. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*, Santa Clara, CA, 2024. USENIX Association.

[11] Microsoft Azure. Azure Cache Pricing - Microsoft Azure, 2025. Accessed: 2025-01-12.

[12] Peter Bailis and Kyle Kingsbury. The Network is Reliable: An Informal Survey of Real-World Communications Failures. *ACM Queue*, 12(7), 2014. Accessed: 2025-03-19.

[13] Soumya Basu, Aditya Sundarrajan, Javad Ghaderi, Sanjay Shakkottai, and Ramesh Sitaraman. Adaptive TTL-based caching for content delivery. *IEEE/ACM transactions on networking*, 26(3):1063–1077, 2018.

[14] David Baylor et al. TFX: A TensorFlow-Based Production-Scale Machine Learning Platform. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, 2019.

[15] Benjamin Berg, Daniel S Berger, Sara McAllister, Isaac Grosof, Sathya Gunasekar, Jimmy Lu, Michael Uhlar, Jim Carrig, Nathan Beckmann, Mor Harchol-Balter, et al. The {CacheLib} caching engine: Design and experiences at scale. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 753–768, 2020.

[16] Daniel S Berger, Philipp Gland, Sahil Singla, and Florin Ciucu. Exact analysis of TTL cache networks. *Performance Evaluation*, 79:2–23, 2014.

[17] Laura Bright and Louiqa Raschid. Using latency-recency profiles for data delivery on the web. In *VLDB'02: Proceedings of the 28th International Conference on Very Large Databases*, pages 550–561. Elsevier, 2002.

[18] Nathan Bronson, Zach Amsden, George Cabrera, Prasad Chakka, Peter Dimov, Hui Ding, Jack Ferris, Anthony Giardullo, Sachin Kulkarni, Harry Li, et al. {TAO}:{Facebook's} distributed data store for the social graph. In *2013 USENIX Annual Technical Conference (USENIX ATC 13)*, pages 49–60, 2013.

[19] Eric Budish, Peter Cramton, and John Shim. The High-Frequency Trading Arms Race. *Quarterly Journal of Economics*, 2015.

[20] Mike Burrows. The Chubby lock service for loosely-coupled distributed systems. In *Proceedings of the 7th symposium on Operating systems design and implementation*, pages 335–350, 2006.

[21] Michael Cafarella, Matteo Interlandi, and et al. HBase: A Distributed, Scalable, Big Data Store. In *IEEE Data Engineering Bulletin*, 2010.

[22] Pei Cao and Chengjie Liu. Maintaining strong cache consistency in the world wide web. *IEEE Transactions on Computers*, 47(4):445–457, 1998.

[23] Damiano Carra, Giovanni Neglia, and Pietro Michiardi. TTL-based cloud caches. In *IEEE INFOCOM 2019-IEEE Conference on Computer Communications*, pages 685–693. IEEE, 2019.

[24] Badrish Chandramouli, Guna Prasaad, Donald Kossmann, Justin Levandoski, James Hunter, and Mike Barnett. Faster: A concurrent key-value store with in-place updates. In *Proceedings of the 2018 International Conference on Management of Data*, pages 275–290, 2018.

[25] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C Hsieh, Deborah A Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E Gruber. Bigtable: A Distributed Storage System for Structured Data. In *OSDI*, volume 6, pages 205–218, 2006.

[26] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C Hsieh, Deborah A Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E Gruber. Bigtable: A distributed storage system for structured data. *ACM Transactions on Computer Systems (TOCS)*, 26(2):1–26, 2008.

[27] Yue Cheng, Aayush Gupta, and Ali R Butt. An in-memory object caching framework with adaptive load balancing. In *Proceedings of the Tenth European Conference on Computer Systems*, pages 1–16, 2015.

[28] Asaf Cidon, Daniel Rushton, Stephen M Rumble, and Ryan Stutsman. Memshare: a dynamic multi-tenant memory key-value cache. *arXiv preprint arXiv:1610.08129*, 2016.

[29] James Cipar. Trading freshness for performance in distributed systems. Technical report, Citeseer, 2014.

[30] James Corbett et al. Spanner: Google's globally-distributed database. *OSDI*, 2012.

[31] James C Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, Jeffrey John Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, et al. Spanner: Google's globally distributed database. *ACM Transactions on Computer Systems (TOCS)*, 31(3):1–22, 2013.

[32] Oracle Corporation. Introducing Oracle True Cache, 2023.

[33] Henggang Cui, James Cipar, Qirong Ho, Jin Kyu Kim, Seunghak Lee, Abhimanu Kumar, Jinliang Wei, Wei Dai, Gregory R Ganger, Phillip B Gibbons, et al. Exploiting bounded staleness to speed up big data analytics. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)*, pages 37–48, 2014.

[34] Jeffrey Dean and Luiz André Barroso. The tail at scale. *Commun. ACM*, 56(2):74–80, February 2013.

[35] Jeffrey Dean and Sanjay Ghemawat. Designs, Lessons and Advice from Building Large Distributed Systems. In *LADIS*, 2009.

[36] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: Amazon's highly available key-value store. *ACM SIGOPS operating systems review*, 41(6):205–220, 2007.

[37] Aleksandar Dragojević et al. FaRM: Fast Remote Memory. In *USENIX NSDI*, 2014.

[38] Facebook Engineering. Presto: Interacting with Petabytes of Data at Facebook. https://engineering.fb.com/2013/11/06/core-infra/presto-interacting-with-petabytes-of-data-at-facebook/, 2013. Accessed: 2025-11-21.

[39] Facebook Engineering. Cache Made Consistent: Improving Cache Efficiency and Data Freshness. *Facebook Engineering Blog*, June 2022.

[40] Nicaise Choungmo Fofack, Philippe Nain, Giovanni Neglia, and Don Towsley. Performance evaluation of hierarchical TTL-based cache networks. *Computer Networks*, 65:212–231, 2014.

[41] Rodrigo Fonseca et al. The Mystery Machine: End-to-End Performance Analysis of Large-Scale Internet Services. In *USENIX OSDI*, 2014.

[42] Alexander Fuerst and Prateek Sharma. FaasCache: keeping serverless computing alive with greedy-dual caching. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 386–400, 2021.

[43] Y. J. Hong and et al. Understanding and Mitigating the Impact of Load Imbalance in Large-Scale Key-Value Stores. In *Proceedings of ACM Symposium on Cloud Computing (SoCC '13)*, pages 1–13, 2013.

[44] Dongxu Huang, Qi Liu, Qiu Cui, Zhuhe Fang, Xiaoyu Ma, Fei Xu, Li Shen, Liu Tang, Yuxing Zhou, Menglong Huang, et al. TiDB: a Raft-based HTAP database. *Proceedings of the VLDB Endowment*, 13(12):3072–3084, 2020.

[45] Murtadha AI Hubail, Ali Alsuliman, Michael Blow, Michael Carey, Dmitry Lychagin, Ian Maxon, and Till Westmann. Couchbase analytics: NoETL for scalable NoSQL data analysis. *Proceedings of the VLDB Endowment*, 12(12):2275–2286, 2019.

[46] Tianyin Jin et al. Zanzibar: Google's Consistent, Global Authorization System. In *USENIX ATC*, 2019.

[47] Xin Jin, Xiaozhou Li, Haoyu Zhang, Robert Soulé, Jeongkeun Lee, Nate Foster, Changhoon Kim, and Ion

Stoica. Netcache: Balancing key-value stores with fast in-network caching. In *Proceedings of the 26th symposium on operating systems principles*, pages 121–136, 2017.

[48] Jaeyeon Jung, Arthur W Berger, and Hari Balakrishnan. Modeling TTL-based Internet caches. In *IEEE INFOCOM 2003. Twenty-second Annual Joint Conference of the IEEE Computer and Communications Societies (IEEE Cat. No. 03CH37428)*, volume 1, pages 417–426. IEEE, 2003.

[49] Flavio P. Junqueira and Benjamin Reed. *ZooKeeper: Distributed Process Coordination*. O'Reilly Media, 2013.

[50] Anuj Kalia et al. eRPC: A Fast, General Purpose, RPC Library for RDMA Networks. In *USENIX NSDI*, 2019.

[51] Suparna Kotra et al. TailBench: A Benchmark Suite and Evaluation Methodology for Latency Critical Applications. In *USENIX ATC*, 2016.

[52] Alexandros Labrinidis and Nick Roussopoulos. Balancing performance and data freshness in web database servers. In *Proceedings 2003 VLDB Conference*, pages 393–404. Elsevier, 2003.

[53] Sangmin Lee, Zhenhua Guo, Omer Sunercan, Jun Ying, Thawan Kooburat, Suryadeep Biswal, Jun Chen, Kun Huang, Yatpang Cheung, Yiding Zhou, et al. Shard manager: A generic shard management framework for geo-distributed applications. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, pages 553–569, 2021.

[54] Bojie Li, Zhenyuan Ruan, Wencong Xiao, Yuanwei Lu, Yongqiang Xiong, Andrew Putnam, Enhong Chen, and Lintao Zhang. Kv-direct: High-performance in-memory key-value store with programmable nic. In *Proceedings of the 26th Symposium on Operating Systems Principles*, pages 137–152, 2017.

[55] Mu Li, David G. Andersen, Jun Woo Park, Alexander J. Smola, Amr Ahmed, Vanja Josifovski, James Long, Eugene J. Shekita, and Bor-Yiing Su. Scaling Distributed Machine Learning with the Parameter Server. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI'14)*, pages 583–598. USENIX Association, 2014.

[56] Shuang Liang, Ke Chen, Song Jiang, and Xiaodong Zhang. Cost-aware caching algorithms for distributed storage servers. In *Distributed Computing: 21st International Symposium, DISC 2007, Lemesos, Cyprus, September 24-26, 2007. Proceedings 21*, pages 373–387. Springer, 2007.

[57] Hyeontaek Lim et al. MICA: A Holistic Architecture for High-Performance Servers. In *USENIX NSDI*, 2014.

[58] Todd Lipcon. Avoiding Full GCs in Apache HBase with MemStore-Local Allocation Buffers: Part 1, February 2011. Available at https://blog.cloudera.com/avoiding-full-gcs-in-apache-hbase-with-memstore-local-allocation-buffers-part-1/.

[59] Zaoxing Liu, Zhihao Bai, Zhenming Liu, Xiaozhou Li, Changhoon Kim, Vladimir Braverman, Xin Jin, and Ion Stoica. DistCache: Provable Load Balancing for Large-Scale Storage Systems with Distributed Caching. In *USENIX Conference on File and Storage Technologies (FAST '19)*, pages 287–301, 2019.

[60] Xuchuan Luo, Jinhong Li, Pengyu Lyu, and Yangfan Zhou. CHIME: A Cache-Efficient and High-Performance Hybrid Index on Disaggregated Memory. In *Proceedings of the 30th Symposium on Operating Systems Principles (SOSP '24)*, pages 110–126, New York, NY, USA, 2024. ACM.

[61] Yandong Mao, Eddie Kohler, and Robert Tappan Morris. Cache craftiness for fast multicore key-value storage. In *Proceedings of the 7th ACM european conference on Computer Systems*, pages 183–196, 2012.

[62] Ziming Mao, Jonathan Ellithorpe, Atul Adya, Rishabh Iyer, Matei Zaharia, Scott Shenker, and Ion Stoica. Rethinking the Cost of Distributed Caches for Datacenter Services. In *Proceedings of the 24th ACM Workshop on Hot Topics in Networks (HotNets '25)*, pages 1–9, 2025.

[63] Ziming Mao, Jonathan Ellithorpe, Atul Adya, Rishabh Iyer, Matei Zaharia, Scott Shenker, and Ion Stoica. Rethinking the Cost of Distributed Caches for Datacenter Services. In *Proceedings of the 24th ACM Workshop on Hot Topics in Networks*, pages 317–325, 2025.

[64] Ziming Mao, Rishabh Iyer, Scott Shenker, and Ion Stoica. Revisiting Cache Freshness for Emerging Real-Time Applications. In *Proceedings of the 23rd ACM Workshop on Hot Topics in Networks*, HotNets '24, page 335–342, New York, NY, USA, 2024. Association for Computing Machinery.

[65] Microsoft. Azure Cosmos DB Integrated Cache - Overview, 2024. Accessed: 2025-04-10.

[66] Muhammad Anis Uddin Nasir, Gianmarco De Francisci Morales, Nicolas Kourtellis, and Marco Serafini. When Two Choices Are Not Enough: Balancing at Scale in Distributed Stream Processing. In *IEEE International Parallel Distributed Processing Symposium (IPDPS '15)*, pages 998–1007, 2015.

[67] Rajesh Nishtala, Hans Fugal, Steven Grimm, Marc Kwiatkowski, Herman Lee, Harry C Li, Ryan McElroy, Mike Paleczny, Daniel Peek, Paul Saab, et al. Scaling memcache at facebook. In *10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*, pages 385–398, 2013.

[68] John Ousterhout, Arjun Gopalan, Ashish Gupta, Ankita Kejriwal, Collin Lee, Behnam Montazeri, Diego Ongaro, Seo Jin Park, Henry Qin, Mendel Rosenblum, et al. The RAMCloud storage system. *ACM Transactions on Computer Systems (TOCS)*, 33(3):1–55, 2015.

[69] Percona. MySQL Caching Methods and Tips, 2023.

[70] Dan R.K. Ports, Irene Zhang, Samuel Madden, and Barbara Liskov. Transactional Consistency and Automatic Management in an Application Data Cache. In *Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation (OSDI '10)*, Vancouver, BC, October 2010. USENIX Association.

[71] Ziyue Qiu, Juncheng Yang, Juncheng Zhang, Cheng Li, Xiaosong Ma, Qi Chen, Mao Yang, and Yinlong Xu. Frozenhot cache: Rethinking cache management for modern hardware. In *Proceedings of the Eighteenth European Conference on Computer Systems*, pages 557–573, 2023.

[72] H. Ramasamy et al. Improving the Performance of Real-Time Bidding Using Edge Compute. In *NSDI*, 2021.

[73] RavenDB. Caching Data: Automatic Database Caching, 2023.

[74] Mohit Saxena, Michael M Swift, and Yiying Zhang. Flashtier: a lightweight, consistent and durable storage cache. In *Proceedings of the 7th ACM european conference on Computer Systems*, pages 267–280, 2012.

[75] Amazon Web Services. Database Caching Strategies Using Redis, 2023.

[76] Swaminathan Sivasubramanian. Amazon dynamoDB: a seamlessly scalable non-relational database service. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, pages 729–730, 2012.

[77] Enis Söztutar. HBase and HDFS: Understanding filesystem usage in HBase. Presented at HBaseCon, June 2013.

[78] Sari Sultan, Kia Shakiba, Albert Lee, Paul Chen, and Michael Stumm. TTLs matter: Efficient cache sizing with TTL-aware miss ratio curves and working set sizes. In *Proceedings of the Nineteenth European Conference on Computer Systems*, pages 387–404, 2024.

[79] Rebecca Taft, Irfan Sharif, Andrei Matei, Nathan VanBenschoten, Jordan Lewis, Tobias Grieger, Kai Niemi, Andy Woods, Anne Birzin, Raphael Poss, et al. Cockroachdb: The resilient geo-distributed sql database. In *Proceedings of the 2020 ACM SIGMOD international conference on management of data*, pages 1493–1509, 2020.

[80] Chunqiang Tang, Thawan Kooburat, Pradeep Venkatachalam, Akshay Chander, Zhe Wen, Aravind Narayanan, Patrick Dowell, and Robert Karl. Twine: A Unified Cluster Management System for Shared Infrastructure. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 787–803, 2020.

[81] Dropbox Tech Team. Meet Chrono: Our Scalable, Consistent Metadata Caching Solution. Dropbox Tech Blog, 2023. Accessed: 2025-03-19.

[82] GitHub Engineering Team. Downtime Last Saturday: What Happened? GitHub Blog, 2024. Accessed: 2025-03-19.

[83] Mike Ulrich. Addressing Cascading Failures. In *Site Reliability Engineering: How Google Runs Production Systems*. O'Reilly Media, Inc., 2016. Chapter 22.

[84] Abhishek Verma, Luis Pedrosa, Madhukar Korupolu, David Oppenheimer, Eric Tune, and John Wilkes. Large-scale cluster management at Google with Borg. In *Proceedings of the 10th European Conference on Computer Systems (EuroSys)*, pages 1–17, 2015.

[85] Xingda Wei, Rong Chen, Haibo Chen, and Binyu Zang. Xstore: Fast rdma-based ordered key-value store using remote learned cache. *ACM Transactions on Storage (TOS)*, 17(3):1–32, 2021.

[86] Wikipedia. List of In-Memory Databases, 2023.

[87] Juncheng Yang, Ziming Mao, Yao Yue, and KV Rashmi. {GL-Cache}: Group-level learning for efficient and high-performance caching. In *21st USENIX Conference on File and Storage Technologies (FAST 23)*, pages 115–134, 2023.

[88] Juncheng Yang, Yao Yue, and KV Rashmi. A large-scale analysis of hundreds of in-memory key-value cache clusters at twitter. *ACM Transactions on Storage (TOS)*, 17(3):1–35, 2021.

[89] Juncheng Yang, Yao Yue, and KV Rashmi. A large-scale analysis of hundreds of in-memory key-value cache clusters at twitter. *ACM Transactions on Storage (TOS)*, 17(3):1–35, 2021.

[90] Juncheng Yang, Yao Yue, Shivaram Venkataraman, Wyatt Lloyd, Michael Kaminsky, and David G. Andersen. A Large-Scale Analysis of Hundreds of In-Memory Key-Value Cache Clusters at Twitter. In *Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2020.

[91] Juncheng Yang, Yao Yue, and Rashmi Vinayak. Segcache: A Memory-Efficient and Scalable In-Memory Key-Value Cache for Small Objects. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*, pages 503–518. USENIX Association, 2021.

[92] Juncheng Yang, Yao Yue, and Rashmi Vinayak. Segcache: a memory-efficient and scalable in-memory key-value cache for small objects. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*, pages 503–518, 2021.

[93] Juncheng Yang, Yazhuo Zhang, Ziyue Qiu, Yao Yue, and Rashmi Vinayak. FIFO queues are all you need for cache eviction. In *Proceedings of the 29th Symposium on Operating Systems Principles*, pages 130–149, 2023.

[94] Juncheng Yang, Yazhuo Zhang, Ziyue Qiu, Yao Yue, and Rashmi Vinayak. FIFO queues are all you need for cache eviction. In *Proceedings of the 29th Symposium on Operating Systems Principles*, pages 130–149, 2023.

[95] Haobo Yu, Lee Breslau, and Scott Shenker. A scalable web cache consistency architecture. *ACM SIGCOMM Computer Communication Review*, 29(4):163–174, 1999.

[96] Shuai Yuan, Jun Wang, and Xiaoxue Zhao. Real-time bidding for online advertising: measurement and analysis. In *Proceedings of the seventh international workshop on data mining for online advertising*, pages 1–8, 2013.

[97] Haoran Zhang, Konstantinos Kallas, Spyros Pavlatos, Rajeev Alur, Sebastian Angel, and Vincent Liu. {MuCache}: A General Framework for Caching in Microservice Graphs. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*, pages 221–238, 2024.

[98] Irene Zhang, Naveen Kr. Sharma, Arvind Krishnamurthy, and Henry M. Levy. Building Consistent Transactions with Inconsistent Replication. In *Proceedings of the 25th ACM Symposium on Operating Systems Principles (SOSP)*, pages 263–279, 2015.

[99] Kai Zhang, Kaibo Wang, Yuan Yuan, Lei Guo, Rubao Li, Xiaodong Zhang, Bingsheng He, Jiayu Hu, and Bei Hua. A distributed in-memory key-value store system on heterogeneous CPU–GPU cluster. *The VLDB Journal*, 26(5):729–750, 2017.

[100] Jingyu Zhou, Meng Xu, Alexander Shraer, Bala Namasivayam, Alex Miller, Evan Tschannen, Steve Atherton, Andrew J Beamon, Rusty Sears, John Leach, et al. Foundationdb: A distributed unbundled transactional key value store. In *Proceedings of the 2021 International Conference on Management of Data*, pages 2653–2666, 2021.

ARTIFACT EVALUATED
USENIX
AVAILABLE

MEGALON: Efficient Data Sharing for Partly Coherent CXL Memory

Jiyu Hu
University of Illinois Urbana-Champaign

Seokjoo Cho*
University of Illinois Urbana-Champaign

Landon Johnson*
University of Illinois Urbana-Champaign

Kiran Hombal
University of Illinois Urbana-Champaign

Shreesha G. Bhat
University of Illinois Urbana-Champaign

Marcos K. Aguilera
NVIDIA

Ramnatthan Alagappan
University of Illinois Urbana-Champaign

Aishwarya Ganesan
University of Illinois Urbana-Champaign

Abstract. CXL allows multiple hosts to share memory. However, the hardware is expected to provide cache coherence only for a small region of CXL memory and it is difficult for hosts to share data in the non-coherent region. Recent work proposes using the small coherent region (SCR) to track coherence metadata and enables coherent and correct sharing of data in the software. We find that this approach suffers from poor performance for large datasets as it cannot fit the metadata in SCR. We propose MEGALON, a new data-sharing approach for CXL that uses a novel split approach, where big but infrequently updated metadata is logically shared via replication, and only small and heavily updated metadata is shared via SCR. This enables MEGALON to support much larger datasets with high performance. MEGALON augments the split approach with novel techniques enabled by a CXL shared log that unlock high performance under many workloads.

1 Introduction

Recent interconnect technologies, like Compute Express Link (CXL) [27, 33, 59, 64], enable *memory sharing* across a group of hosts. This offers an opportunity to build more efficient databases, key-value stores, and object stores that span many hosts and efficiently share data by keeping it in CXL memory and accessing it via CPU loads and stores [10, 18, 19].

Unfortunately, today it is hard for hosts to coherently share data in CXL. This is because, even with the latest CXL versions [18, 19], hardware is expected to provide coherence only for a small part of the CXL memory. Vendors like AMD, Micron, and practitioners alike posit that hardware coherence will be limited to a few hundred MBs of CXL memory, while total CXL memory will span several TBs [10, 26–28, 47, 60]. We call this model the *partly coherent* CXL model, where the CXL memory has a *small coherent region* (SCR) and a *large non-coherent region* (LNR). This coherence model creates a problem for applications: because LNR is not kept cache coherent by the hardware, host processor caches may hold stale values when accessing data, leading to correctness problems.

*Equal contribution.

A natural way to address this problem is to use SCR to provide coherence in software for data in LNR. The actual data objects can be kept in LNR, and their coherence can be ensured in software by tracking a per-object coherence record in SCR. Software coherence is feasible because coherence needs to be tracked only for coarse-granular application objects (e.g., database rows), not cachelines [27]. To enable hosts to locate data objects and coherence records, a sharing approach must also share an index that maps an object identifier to its location in LNR and its coherence record in SCR. This index can itself be shared via SCR. We refer to the coherence records and index together as metadata, and we call the scheme of relying on the hardware-coherent region to share this metadata across hosts to enable CXL data sharing *hardware-coherent metadata-based sharing*, or HCMeta for short.

HCMeta is the approach used by Tigon [26], a recent CXL database. Tigon is a partitioned database that uses CXL to share data accessed in cross-host transactions. By using CXL instead of message passing, Tigon makes cross-host transactions faster than RDMA-based databases [26]. Tigon keeps the shared database tuples in LNR and provides coherence via a per-tuple coherence record in SCR, with each record also pointing to the tuple's location in LNR. Tigon stores an index in SCR that maps a row key to its coherence record.

A downside of the HCMeta approach is that, with a large number of objects, the metadata (which is essential for correctly sharing data) would not fit in SCR. For example, with a large dataset, the metadata would be two orders of magnitude bigger than the expected SCR capacity (§2.2).

Tigon handles this by limiting the number of CXL-resident objects so that their metadata fits in SCR. When this limit is hit, Tigon *unshares* objects, moving them from CXL to local-DRAM partitions to which those objects belong, and reclaims their SCR metadata space for newly shared objects. When the shared working set is small, Tigon can fit the metadata in SCR, yielding good performance. However, with larger datasets, even low fractions of cross-host transactions increases the total amount of shared data over the course of the workload. This causes churn, where objects are repeatedly unshared and

reshared, which degrades performance. For example, with 20% cross-host transactions, increasing the dataset from 2.4M to 24M objects reduces Tigon's throughput by 10× due to churn and data movement between CXL and DRAM. We also implement HCMeta in a key-value store that keeps partitions in CXL (instead of host-local DRAM). Even when avoiding data movement, HCMeta suffers from poor performance due to churn (§2). Overall, enabling CXL data sharing by relying entirely on hardware coherence for metadata leads to repeated unsharing and resharing, and ultimately poor performance.

This paper thus asks a fundamental question: *How can one share a large number of objects in CXL without unsharing some of them when even the metadata is too big to fit in the hardware-coherent memory?* We answer this with MEGALON, a new data-sharing approach for partly coherent CXL.

MEGALON, like HCMeta, shares data objects via some shared metadata. The metadata in MEGALON consists of an index that maps an object identifier to its coherence record and its location within LNR. The coherence record consists of a lock bit and a counter. At a high level, writes to an object are serialized by the lock and writes increment the counter; reads check the counter to catch intervening writes and detect (and flush) stale processor caches at the host.

The key novelty in MEGALON is its ability to share a large number of objects in CXL without being constrained by SCR capacity. Unlike HCMeta, which relies *entirely* on SCR to share *all* metadata, MEGALON adopts a *split* approach. It observes that the index is large (as it contains object identifiers) but is updated infrequently; in contrast, coherence records are tiny (a lock bit and a counter) but are updated on every object write. Based on this asymmetry, MEGALON splits the metadata: it *logically* shares the index by *replicating* it in each host's local DRAM, while relying on SCR only for keeping the small coherence records, maintaining pointers to them from the replicated index. Because each host has significant DRAM capacity, it can easily fit index replicas even for large datasets. Although replication increases local DRAM usage, each index entry is still small compared to a data object. Since the index changes infrequently, keeping the replicas consistent adds minimal overhead. Meanwhile, placing the frequently updated coherence records in SCR keeps sharing efficient. Overall, by storing only the small, frequently updated parts of metadata in SCR and replicating the rest, the split approach dramatically increases the number of objects that can be shared.

While the split approach enables sharing many objects, MEGALON still needs to solve two big challenges. First, how to maintain the consistency of index replicas? Second, because a coherence record is maintained for every object in SCR, there is still a limit on the number of shared objects. How can MEGALON tackle this limit? The answer to both these challenges is a *shared log* in CXL. The shared log is also central to other optimizations in MEGALON. The shared log is an ordered sequence of entries that hosts can append to and read from. To conserve SCR, MEGALON maintains the log entries in LNR and only the shared-log tail pointer in SCR.

The shared log keeps the index replicas across hosts consistent by ordering all operations to the index. To update the index, a host sequences the update by incrementing the tail and appending the update to the log. To read the index, a host synchronizes its replica with the shared log by checking if the tail has advanced and applying any new entries. It then performs the index read, ensuring consistency.

The shared log also enables MEGALON to increase the number of objects that can be shared by allowing it to *dynamically* allocate and deallocate coherence records. MEGALON first observes that coherence records are unnecessary for objects that are only read (read-shared), and so it *dynamically* allocates them only for objects that are both read and written (read-write-shared). Readers perform coherence checks only if a coherence record exists. Thus, the number of read-shared objects is not bounded by SCR capacity. The number of read-write-shared objects, however, remains limited by SCR. When this limit is hit, MEGALON converts some objects to read-shared by deallocating their coherence records and reallocating them to other objects. These dynamic allocations and deallocations are enabled by the shared log which helps update the pointers to the coherence records consistently in the index replicas.

With dynamic coherence records, coherence records can be allocated and deallocated at any time and MEGALON must still correctly ensure coherence. To solve this, MEGALON uses a novel *dual-path coherence* technique, where it uses the shared log to provide coherence when the coherence records themselves alone cannot. In the first path, when the coherence record's allocation status of an object does not change, the coherence record itself ensures coherence. In the second path, when the allocation status changes, the shared log provides the required coherence. Specifically, hosts detect allocation status changes from the (de)allocation events in the shared log and take necessary actions to ensure coherence (e.g., by flushing CPU caches).

Like HCMeta, MEGALON *does* experience churn to deallocate and allocate coherence records. In fact, any sharing approach that maintains metadata in SCR that is linear in the number of the objects in CXL will incur churns. However, MEGALON pushes the point at which churn happens significantly further than HCMeta. It does so by keeping the index out of SCR and allocating coherence records only for read-write-shared objects. Further, a churn in MEGALON is far cheaper. In HCMeta, when SCR runs out, objects must be unshared. To reshare an unshared object, a host must contact and wait for the owner to share the object. In contrast, MEGALON need not unshare an object but only deallocate the coherence record. Further, the shared log enables efficiently resharing an object in read-write mode: any host can allocate a coherence record without synchronously waiting for other hosts.

The shared log also enables many optimizations. First, it allows shrinking coherence-record counters to just a few bits; when the counter wraps around, MEGALON records the event in

the shared log so that hosts can use them to flush caches and avoid stale reads. Second, the shared log allows MEGALON to support local DRAM partitions, letting hosts store exclusively accessed data in local DRAM and keep only shared objects in CXL. Specifically, MEGALON uses the index to point to local-DRAM locations and moves objects between CXL and host-local DRAM by modifying the index via the shared log.

Results. We have built MEGALON in a CXL-based key-value store and file-system page cache. In read-only workloads, MEGALON improves throughput by 15× over HCMeta for large datasets. Under read-write workloads, MEGALON supports up to 12× larger datasets without churns, improving performance by 4×–10×. Even when MEGALON experiences churns, it improves performance by 2.5×–14.9× as it reduces the number of churns and each churn is 8× cheaper. HCMeta's performance drastically drops with larger keys, whereas MEGALON is unaffected by key size. The shared log allows MEGALON to use smaller coherence records, improving performance by 6.3× over the default coherence-record size. It also enables MEGALON to use host-local DRAM, offering high performance in partitioned workloads. Under YCSB, MEGALON performs 3×–14× better than HCMeta. MEGALON can share a larger page cache, improving performance by 1.9×–5.7×.

Contributions. This paper makes four contributions.

- We propose MEGALON, the first sharing approach for partly coherent CXL that can share a large number of objects.
- MEGALON introduces a novel *split* approach for sharing the metadata, logically sharing the index and relying on hardware coherence only for coherence records.
- MEGALON uses many novel techniques centered around a *shared log*. The shared log keeps the index replicas consistent; it also allows MEGALON to dynamically allocate and deallocate coherence records and do so without synchronously waiting for other hosts; it also enables *dual-path coherence*, where coherence is ensured by the shared log when the coherence records themselves alone cannot.
- We experimentally show MEGALON's benefits.

MEGALON is open-sourced at https://github.com/dassl-uiuc/MEGALON-artifact.

2 Background and Motivation

We describe the partly coherent CXL model and explain the drawbacks of the current approach to sharing in this model.

2.1 Partly Coherent CXL Memory

CXL is an emerging interconnect standard built atop PCIe, using which hosts can access a CXL memory device. With CXL 1.1, a single host can use CXL for memory expansion and access the memory in a cache-coherent manner. With recent CXL 3.0 and CXL 3.2 specifications, *multiple hosts* can share memory on an external CXL device, allowing processors in hosts to issue loads and stores directly on the shared memory. Like any other memory, accesses to CXL memory are optimized through several levels of processor caches. In addition to the shared CXL memory, each host has a local memory not accessible by other hosts. Local memory has substantial capacity (a few 100GBs per host [27]), and it has lower latency and higher bandwidth than CXL memory [27, 33, 56, 64].

While CXL 3.0 and later specifications [18, 19] provide cache coherence in hardware, it is expected that this support will be limited to a small region, while the rest of CXL memory is non-coherent without support for speculative prefetching. Indeed, hardware vendors such as AMD, Micron, and Samsung indicate that hardware-based coherence is practical only for a few hundred MBs, while CXL memory can span several TBs [10, 25–28]. The reason for this limit is that the cache coherence mechanisms (snoop filters and back invalidation) do not scale with increasing memory sizes [10, 25, 26].

We call this model the *partly coherent* model, and refer to the small hardware cache-coherent region as SCR and the large non-coherent region as LNR. This model creates a problem for applications. If an application spanning across hosts shares data in the LNR, the processor caches may hold stale values, resulting in correctness problems. For example, in an object store, one host could update an object, but another host that had previously read that object may read an old value from its processor cache, resulting in a stale object being returned.

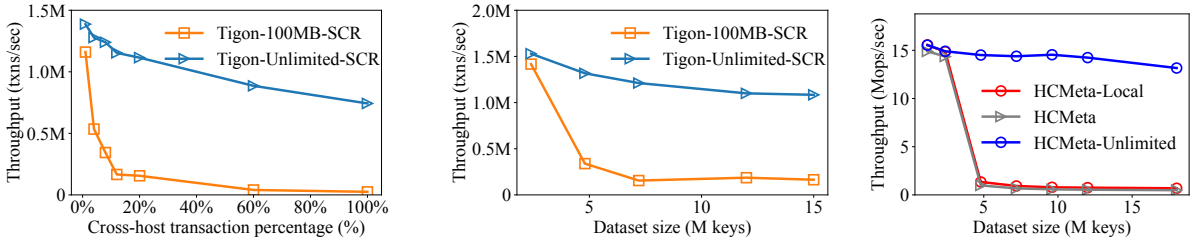
2.2 Sharing Approaches for the Partly Coherent Model

One way to completely sidestep the incoherence problem is to restrict sharing only to the SCR. However, this is impractical given SCR's tiny capacity: applications cannot coherently share more than a few hundred MBs of data simultaneously, which severely limits the utility of shared CXL memory.

Another approach is to allow sharing data in LNR and realize correct and coherent sharing in software by tracking some metadata in the hardware-coherent region. We refer to this general scheme of sharing data in CXL as *hardware-coherent metadata-based sharing* (or HCMeta).

In fact, this is the approach used by Tigon [26]. Tigon is a partitioned database system, where each host has its own partition of the data in the host-local DRAM. A host cannot access the data present in another host's local DRAM. If a transaction at a host accesses a tuple from other partitions (i.e., it is a cross-host transaction), Tigon moves those tuples to CXL memory for sharing across hosts. Thus, cross-host transactions can complete without any network coordination, but with just memory accesses. While Tigon is an ACID compliant database, we focus on its approach to share data in CXL because our goal is to build an efficient data-sharing approach for CXL, and not a specific application like a database.

Tigon realizes correct sharing by keeping the shared tuples in LNR and providing coherence for these tuples via per-tuple coherence information (called HWcc record in Tigon [26, §3.2]) that it stores in SCR. The HWcc record contains a lock and bitmap (one bit per host) that ensures coherence.



(a) Tigon Artifact: Varying Cross-Partition Txns (b) Tigon Artifact: Varying Dataset Size (c) HCMeta in a KV Store

Figure 1: **Drawback of HCMeta** *(a) and (b) show Tigon artifact's performance and (c) shows the performance of HCMeta in a key-value store.*

Specifically, the lock serializes accesses to a tuple. Writers, when updating a tuple, reset the bitmap entries for other hosts. When reading, a reader checks if its bit is set. If so, it can rely on its processor caches to read that tuple; otherwise, its caches are out of date and thus the host flushes the corresponding cachelines and reads the data from CXL and sets its bit in the bitmap. The HWcc record also helps hosts locate the shared tuple in LNR. To allow hosts to locate the HWcc record for a tuple, Tigon maintains an index in SCR, which maps a row key to its HWcc record in SCR. Thus, Tigon, by maintaining metadata (i.e., per-object HWcc record and the index) in SCR, enables correct sharing of data tuples in LNR.

Since systems using the HCMeta approach maintain per-object metadata, with a large number of objects, the metadata would not fit in SCR. For example, in Tigon, with 24-byte keys and 8-byte HWcc records [26, §3.2], metadata would be at least 8GB for a 1TB dataset with 4KB objects, which is roughly two orders of magnitude more than the SCR, which is expected to be a few 100 MBs [10, 27, 28]. HCMeta thus has a limit on the number of objects that can be shared in CXL.

When the SCR limit is hit and no more metadata can be kept in there, HCMeta *unshares* some objects, so that the metadata space can be used for other objects. For example, Tigon moves some objects from CXL to their local partitions in the host-local DRAM. To share an unshared object, a host that wants to access that object contacts the host that owns the partition that the object belongs to. The owner moves the object to CXL and creates a HWcc record and index entry in SCR for it (unsharing other objects in its partition if it needs to make space). After this, other hosts can access the object.

2.3 Drawbacks of HCMeta

When many data objects are shared in a workload, HCMeta incurs a lot of churn to share and unshare objects. Note that the data that needs to be shared at a particular point may be small. For example, in Tigon, only those tuples that are currently being accessed by cross-host transactions need to be present in CXL, which is small [26]. Thus, the metadata to share data at a particular point would fit in SCR. However, over the course of the workload, different data objects need to be shared at different points and the metadata for all of them will not fit in SCR. Thus, some objects need to be unshared so that some other ones can be shared and this happens repeatedly. This churn resulting from unsharing and sharing can significantly degrade performance.

We measure this impact in the publicly available Tigon artifact [58]. Figure 1(a) and (b) show the results. In 1(a), we fix the dataset size to 12M rows. We run the YCSB-based read-only transaction workload from the Tigon paper, where each transaction accesses 10 rows with a skewness factor of 0.7. A transaction can be local (all 10 rows from the local partition) or cross-host (5 from local partition and 5 from remote). We vary the percentage of cross-host transactions and plot Tigon's throughput with an SCR size of 100MB (realistic) or an unlimited SCR (unrealistic). As shown, with few cross-host transactions, both the 100MB-SCR and unlimited-SCR versions perform well. This is because only a small amount of data is shared via CXL and thus there is little churn. However, with a higher percentage of cross-host transactions, there is more data shared over the course of the workload. Thus, the metadata does not fit in the 100MB-SCR, causing churn and reducing its performance compared to an unlimited SCR. Figure 1(b) shows a similar result when varying the dataset size with a fixed cross-host transaction percentage (20%). With small datasets, the metadata fits in SCR and thus Tigon performs well. However, as dataset size increases, the metadata does not fit, causing churns and a steep drop in performance.

Since Tigon is a transactional database, to measure the impact of HCMeta in isolation, we implement the HCMeta scheme in a non-transactional key-value store. We build three variants of this store: (i) where the partition resides on host-local DRAM similar to Tigon (called HCMeta-local), (ii) where partitions reside entirely on CXL, so that only metadata (not data) moves between local DRAM and CXL memory as part of the churn (HCMeta), (iii) a variant of HCMeta with unlimited SCR (HCMeta-Unlimited). Figure 1(c) shows the results for a read-only workload with varying dataset sizes.

First, if the SCR space is unlimited, all metadata fits and thus no churn happens. Thus, the HCMeta-Unlimited variant performs well even as the dataset keeps increasing. However, in HCMeta-local, the metadata does not fit, leading to churn and data movement, which degrades performance. Although HCMeta incurs only metadata movement and performs slightly better than HCMeta-local, as the dataset increases, its performance still drops significantly due to churn.

Summary. Although metadata is critical to enable coherent and correct sharing of CXL data, relying entirely on SCR for sharing metadata leads to churn and poor performance.

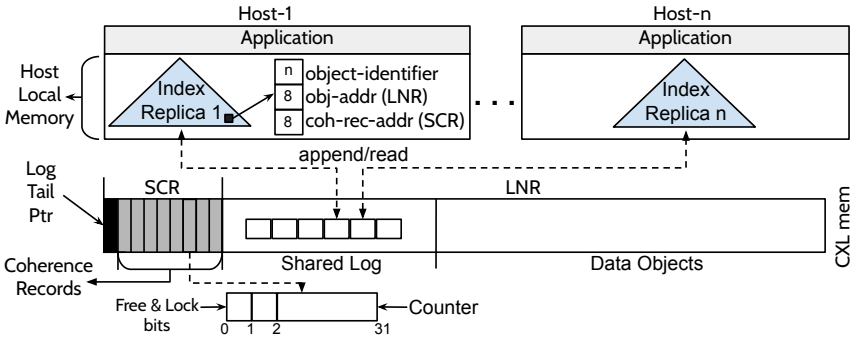


Figure 2: **Split Sharing.** *Each host has a replica of the index in local DRAM; each index entry maps an n-byte object identifier to the object in* LNR *and the object's coherence record in* SCR. *The index replicas are kept consistent via a shared log in CXL (§3.3).*

3 MEGALON Design

MEGALON is a new data-sharing approach for partly coherent CXL memory. It can share many objects without being limited by SCR. We first describe what metadata MEGALON maintains to let hosts coherently share objects and locate them (§3.1). We then explain its novel split approach, which shares only coherence records via SCR and the index via replication (§3.2). Next, we describe how a shared log enables consistency of index replicas (§3.3). We then explain how shared-log-enabled dynamic coherence records allow sharing more objects and how dual-path coherence ensures coherence via either coherence records or the shared log (§3.4). Finally, we discuss how the shared log enables optimizations like compressing coherence records and local-DRAM partitions (§3.5).

3.1 Shared Metadata

Like HCMeta, MEGALON also shares metadata across hosts to enable correct, coherent sharing of data objects. Like HCMeta, MEGALON tracks coherence for coarse-grained (1KB – 4KB) data objects like database rows, key-value pairs, object-store blobs, and file pages (and not cachelines). To ensure coherence of data objects in LNR, MEGALON maintains a *coherence record* per object, that consists of a lock bit, a free bit, and a counter. It also maintains an index that maps each object identifier to its location in LNR and its coherence record.

For coherence of data objects, one must ensure the *single writer or multiple readers (SWMR)* and *data-value* invariants [54, §2.4]. SWMR requires that for each data object, at any time, either a single writer may write it, or some number of readers may read it. The data-value invariant requires that the latest write to an object is visible to subsequent readers.

MEGALON ensures SWMR as follows. When a write to an object starts, the host performing the write sets the lock bit, increments the counter, and performs the write to LNR. It then increments the counter again and clears the lock bit. For a read, the host first notes the counter value as $c1$, reads the data from LNR, then notes the counter value again as $c2$. If $c2$ is odd or $c1$ and $c2$ are not equal, the reader detects an intervening write and retries the read (after flushing the CPU cache lines of the data object). Since writes are serialized by the lock bit and reads cannot succeed with intervening writes, the coherence record ensures SWMR. This mechanism is similar to how sequence locks work [52].

To maintain the data-value invariant, MEGALON keeps a *host-side* counter for each object a host has successfully read. To ensure that reads do not see stale data in CXL memory, upon every write, a host flushes its CPU caches; the host does so after the write to the object but before incrementing the counter the second time. Before reading from LNR, a reader checks whether its host-side counter matches $c1$ (from above). If they match, it proceeds with the read; otherwise, its CPU cache is stale, so it flushes the object's cache lines and then reads. If both the SWMR and the data-value invariant checks pass, the read succeeds. The reader then updates its host-side counter of the object to the post-read counter value $c2$.

A host performing a read or a write first looks up the index to locate the object in LNR and access its coherence record. It then performs the read or write as described above.

3.2 Split Metadata Sharing Approach

The previous subsection explained how the shared metadata across hosts enables MEGALON to coherently share data objects and correctly locate them. We now discuss how MEGALON consistently shares the metadata itself across hosts. Relying purely on SCR limits the number of data objects that can shared in CXL, leading to churn and low performance (§2.3).

To enable sharing a large number of objects, MEGALON proposes a novel *split metadata sharing* approach that is not constrained by SCR. The key insight is to split the metadata into two parts: (1) a small but frequently updated part, which is shared using hardware coherence, and (2) a large but infrequently updated part, which is shared using other means. MEGALON *physically* shares only coherence records via SCR and *logically* shares the index by *replicating* it in each host's local-DRAM. The index maps object identifiers to their data locations in LNR and their coherence-record locations in SCR. Figure 2 shows the split-sharing idea.

The rationale behind the split approach is as follows. First, since the coherence records are small (just lock and free bits and a counter per record), they can be maintained in SCR even for a large number of objects. Further, because they are fre-

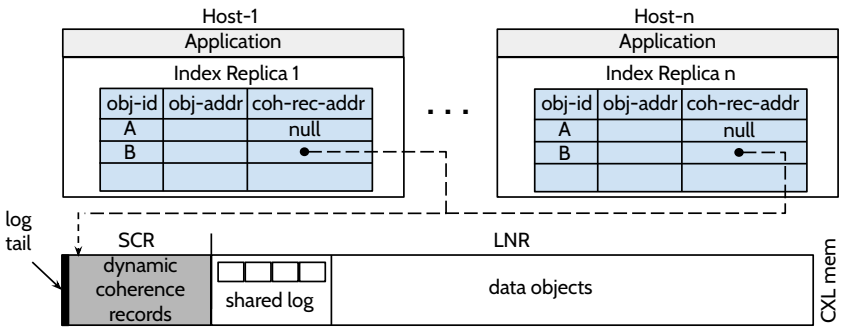


Figure 3: **Dynamic Coherence Records.** *B is read-write shared and thus has a coherence record allocated, while A does not.*

quently updated (on every object write), sharing them via the hardware-provided cache-coherent region is more efficient. Second, the index is large because it contains object identifiers as keys, which can be much larger compared to coherence records. For example, keys can be as large as 50 bytes [14], whereas coherence records are just 4 bytes. Thus, the index cannot be contained in SCR. While SCR is too small (a few 100 MBs) to fit the index, each host locally has considerable amounts of DRAM (a few 100 GBs), which is 1000× bigger than the SCR. This allows the index to fit within every host's local-DRAM even with large datasets. While replication increases local-memory footprint, each index entry is still much smaller compared to the coarse-granular data object. For example, a 50-byte index entry is 80× smaller than 4KB objects. Further, compared to coherence records that are updated on every write, the index is modified less frequently (e.g., when a new object is inserted). Thus, maintaining the consistency of index replicas can be done with little overhead.

Overall, by keeping only coherence records (not the index) in SCR, the split approach significantly increases the number of objects that be shared. Consider a 100MB SCR, 40-byte keys, and 4-byte coherence records. The split approach can share ∼25M objects. The HCMeta approach requires 52 bytes (as it also stores two 4-byte pointers, one to point to coherence record and the other to the object in LNR) in SCR per object. Thus, it can share only 1.9M objects, an order of magnitude less than the split approach. As a result, with large datasets, MEGALON offers much higher performance than HCMeta. Although MEGALON adds local-DRAM memory overhead for replicating the index, it is a small cost to pay for the outsized performance improvements it brings, as we show later (§6.6).

In addition to reducing SCR usage, the replicated index also offers two benefits over keeping the index in SCR. First, it improves latency because index accesses are in local memory. Second, it offers better scalability because each host has its own copy of the index and thus leads to less contention than synchronizing on a shared index in SCR.

Why not logically replicate all metadata? One might consider logically replicating all metadata, including coherence records, so as to not be limited by SCR capacity. Unfortunately, this approach requires synchronizing the replicas for coherence records across hosts, which is expensive because they are frequently updated. The split approach avoids this overhead by physically sharing coherence records via SCR and thus taking advantage of hardware cache coherence for them. We showcase the benefits of the split approach over pure logical replication in our evaluation (§6.10).

Challenges. Although the split approach pushes the limit on how many objects can be shared, MEGALON still faces two major challenges. First, MEGALON must keep the index replicas consistent across hosts. Second, because split sharing requires a coherence record for every object, the total number of shareable objects remains constrained by SCR capacity. The solution to both challenges is a *shared log* in CXL. This shared log also underpins several optimizations in MEGALON.

3.3 Index Consistency atop Shared Log

One way to realize consistency of index replicas across hosts is to use state-machine replication [50] via consensus [31, 37]. However, this incurs significant overhead as it requires software coordination via message passing. MEGALON, instead, takes inspiration from Node Replication [13] and shared logs [6, 7], which consistently replicate data structures using a log. MEGALON adapts this idea to ensure consistency of index replicas via a shared log in CXL that orders all operations on the index.

The log consists of an array of entries and a *log tail*. When a host intends to update the index (e.g., to insert an object), the host performs the update through the shared log. It first reserves a new entry in the log by advancing the log tail via a compare-and-swap operation; it then logs the operation in the new entry. The host then applies all entries up to the tail to its local index replica, after which the update completes. Local reads of index at replicas are guaranteed to be consistent because, before a local read, the host does a *tail check*: it checks if it has applied all entries up to the current tail; if not, it replays all entries up to the current tail to bring the local replica up-to-date. The shared log avoids message passing across hosts to agree on the index. The shared log also eschews synchronous cross-host software coordination, as hosts can independently append to and synchronize with the log.

To minimize SCR use, MEGALON stores log entries in LNR

and keeps only the log tail in SCR (see Figure 2). MEGALON bypasses CPU caches when reading log entries, so coherence is not a concern. CPU caches anyway offer little benefit because log entries are read sequentially. When writing a log entry, hosts flush it to CXL. Hardware guarantees coherence for the log tail (as it is in SCR), ensuring hosts see the changed tail during tail checks. This prompts them to apply new log updates, keeping index replicas consistent across hosts.

3.4 Dynamic Coherence Records, Dual-Path Coherence

The split approach increases the number of objects that can be shared. However, because it maintains a coherence record for every object, the SCR space still limits the the number of shared objects. To address this problem, MEGALON uses the shared log to *dynamically* allocate and deallocate coherence records. It then uses a novel *dual-path coherence* mechanism, where MEGALON leverages the shared log to enforce coherence when coherence records cannot alone do so.

3.4.1 Dynamic Coherence Records

MEGALON first realizes that if a shared object is only read (read-shared), it does not need a coherence record. Thus, MEGALON does not allocate a coherence record for such an object and the object's index entry does not point to a coherence record. For such objects, a host performing a read skips coherence checks. For example, in Figure 3, when reading object *A*, hosts would skip the coherence checks. MEGALON only maintains coherence records for objects that are being both written and read (read-write-shared); a host performing a read for such objects (e.g., object *B* in Figure 3) will locate the coherence record from the replicated index, and perform coherence checks.

Since MEGALON does not spend any memory in SCR for read-shared objects, the number of read-shared objects is not bound by the SCR. However, SCR space still limits how many objects MEGALON can share in the read-write-shared mode. When this limit is hit, because MEGALON does not require coherence records for read-shared objects, MEGALON *dynamically* converts some read-write-shared objects into read-shared objects. Whenever a write occurs to a read-shared object, MEGALON dynamically allocates the coherence record for it. Thus, by dynamically converting between read-write-shared and read-shared modes, MEGALON can avoid unsharing the object when it hits the SCR space limit (unlike HCMeta).

3.4.2 (De)allocating Coherence Records via Shared Log

All hosts must correctly know whether or not an object has a coherence record. MEGALON ensures this by using the shared log to update the coherence-record pointer in the index.

When an object is created, MEGALON does not allocate a coherence record. A host performing a write first checks the index to see if the coherence record is allocated for the object. If not, the host allocates the coherence record in SCR; it uses the free bit (see Figure 2) to find an unoccupied coherence record and clears the free bit. It then updates the pointer in the index to point to the allocated record via the shared log.

If two hosts race to update the pointer in index, only the first one in log order will succeed. Although both hosts can allocate coherence records independently and both can append their request to the log, only the first request can be successfully applied to the replicated index. This is because, all hosts apply the two operations in the same order from the log. Thus, all hosts will know that the pointer is already set in the index by the first request and thus ignore the second request. When this happens at the host that created the second request, it frees the coherence record it allocated. Note that only the first write transitioning an object to read-write sharing needs to assign a new coherence record for the object, requiring log updates. Subsequent writes are efficient; they simply use the coherence record in SCR without log updates.

Deallocations also happen via the log. Such deallocations happen to make space for allocating coherence records for different objects. Any host can deallocate a coherence record by clearing the pointer in the index through the shared log and then deallocating the corresponding coherence record (setting the free bit) in SCR. This can happen either in the synchronous path when a host cannot find space for allocating a new coherence record or in the background as MEGALON proactively tries to keep some free coherence records.

3.4.3 Dual-Path Coherence

So far, we discussed how MEGALON ensures coherence via coherence records. If coherence records were never deallocated, this mechanism would suffice. However, because coherence records are dynamic, they can be deallocated anytime. To solve this problem, MEGALON uses a scheme for coherence with two paths: one provided by the coherence record (discussed so far) and another enabled by the shared log (discussed below). We first explain why coherence records alone are insufficient. We then explain how the shared log helps.

Why coherence records alone are insufficient? Dynamic coherence records do not pose a problem for writes because a coherence record cannot be deallocated when a write is in progress as the writer holds the lock. However, this is not true for reads. When a read starts, a coherence record may exist for the object but may be deallocated before the read finishes. Conversely, a coherence record may not exist at the beginning of the read but may be allocated before the read completes.

One might handle these cases by checking that the index entry is the same before and after the data read. However, this check is insufficient, because reading the index tells only the current allocation status of the coherence record and not the transitions. For example, suppose an object *A* is in read-shared mode (i.e., coherence record does not exist). A host *i* first checks the index and skips the coherence check and performs the read. During the read from LNR by *i*, another host *j* switches *A* to read-write mode, writes to it, and then switches it back to read-shared mode. If host *i* again checks index after the read, it sees that the object does not have a coherence record and incorrectly skips the coherence check.

Such a transition could also happen after a host reads an object and before the host reads the object again. For example, suppose host *i* reads object *A*, then *A* is switched to read-write mode, written to, and then switched to read-shared mode. If host *i* reads *A* again and skips coherence checks because it does not find the coherence record in the index, it incorrectly reads the object from its (stale) CPU caches.

Thus, a host cannot rely only on the current coherence-record state. It must know if there were any allocation status changes for the coherence record since the host began the current read or the host previously read an object.

Shared-log enabled coherence path. One option to realize this is to synchronously notify all hosts of such events. Specifically, a host that changes the coherence-record status of an object notifies and waits for all other hosts to acknowledge before it changes the status. The other hosts can take necessary actions like flushing their caches before they acknowledge. However, this would significantly increase the overhead to switch between read-shared and read-write-shared modes.

MEGALON's shared log enables a better solution, where the host that switches the sharing mode simply records the event in the shared log. Each host independently detects these events when they replay the shared log. This reduces overhead to switch between modes. This mode switch in MEGALON is akin to the churn in HCMeta. However, the cost of the churn in MEGALON is much lower because the log decouples the hosts.

Thus, the shared-log coherence path works as follows. While replaying the log, if a host finds an allocation or deallocation event, it flushes the CPU caches for the affected objects. Reads after this point do not see stale caches. This mechanism also handles cases where the coherence-record allocation status for an object changes while a read is in progress; hosts simply check if there was a coherence-record status change for that object between the first and second coherence checks. If so, the host retries the read after flushing the cache.

3.4.4 Handling Object Deletions

When an object is deleted, its index entry is removed via the shared log and the LNR memory used by the object is reclaimed. Object deletions, however, introduce challenges to coherence that MEGALON must handle. One challenge is that hosts may be caching the old contents of the memory location of the deleted object. Suppose a reader reads object *A* at some location in LNR, storing its contents in its CPU cache. Then suppose *A* is deleted and another object *B* is allocated at the same location by another host. If the first host tries to read *B*, the host will see *A*'s contents (from its cache), which is incorrect. Similarly, consistency can be violated if such deletions happen while the read is in progress. When a write is in progress, MEGALON prevents changes to the index entry of an object because writers set the lock bit. However, because reads are optimistic, an object can be deleted while the read is in progress. So, MEGALON must detect any changes to the index entry of the object while the read was in progress.

MEGALON handles both these challenges via the shared-log-enabled coherence path. Specifically, since object creations and deletions happen via the shared log, hosts can use those events and correctly flush the caches during replay.

3.4.5 Putting It Altogether: MEGALON Reads and Writes

We now describe how the different pieces of MEGALON work together in the read and write of an object.

Read. To read an object *A*, a host synchronizes the index with the shared log, and reads its local index to find *A*'s LNR location. At this point all new log entries have already been applied on the replica, thus flushing potentially stale cachelines of *A*. If *A* is read-write-shared, the host obtains the counter *c* from the coherence record and compares it with the host-side counter *l*. If they are not equal or *c* is odd, the host flushes the cachelines of *A* and retries the read. This is the *pre-read* check. Pre-read check is not needed if *A* is read-shared. The host then accesses the actual data from LNR. The host then performs *post-read* checks: it checks if there were any log updates concerning *A* or if *A* is read-write-shared and the current counter *c'* is different from the previous value of *c*. If either is true, the host flushes cachelines of *A* and retries the read. After a successful read, the host updates *l* to *c'*.

Write. To write an object *A*, a host synchronizes the index and consults the index entry of *A*. If *A* is read-shared, the host allocates a coherence record for *A* by finding a free coherence record and updates the pointer in the index. If the index update fails, it retries the write. It then acquires the lock bit in the coherence record, and checks if there were any log updates since it initially got the index entry; this is because the index might have been updated after acquiring the entry and before acquiring the lock bit. The host then increments the counter in coherence record *c*. This is the *pre-write* step. The host then updates *A* in LNR. It then performs the *post-write* steps. It flushes the relevant cachelines, pushing the update to CXL memory. The host then increments the counter *c* again and sets host-side counter to *l*. It then finally resets the lock bit.

3.5 Shared-Log-Enabled Optimizations

The shared log also enables other optimizations in MEGALON.

3.5.1 Smaller Coherence Records

The counter in the coherence record is 30 bits. If fewer bits are used, then coherence records for more objects can fit in SCR, which increases the number of objects that can be shared in read-write mode. This improves performance by reducing churn. However, with fewer bits, counters may wrap around and threaten coherence: a host could think that its counter in the host-side metadata is the same as the one in the coherence record, but in reality, the counter has wrapped around; in such cases, the host may incorrectly read its stale CPU caches. The shared log provides a way to handle this issue. When the counter wraps around, MEGALON logs the wrap-around event in the shared log. When hosts replay the log, they see such

events and flush their caches, avoiding stale reads.

3.5.2 Supporting Local Partitions

Another optimization that the shared log enables is local DRAM partitions. Partitioned applications store data in host-local DRAMs and move data to CXL only for sharing across hosts (e.g., Tigon). Such applications perform well under workloads where hosts mostly access data from their own partitions. MEGALON can also support local partitions; it uses the index to point to local-DRAM locations and moves objects between CXL and local DRAM of hosts by modifying the index appropriately via the shared log. Thus, MEGALON can also realize high performance for partitioned workloads. MEGALON decides to move an object to local when n consecutive accesses are from the same host (n is configurable). MEGALON performs the movement by making a copy of the object on local DRAM and updating the index to point to the local copy. The correctness of the movement is also guaranteed by the dual-path coherence mechanism. Consistency of local objects are managed by local reader-writer locks. When a host tries to access an object that is moved to a remote host, it sends a message to the remote host. The remote host brings the object back to CXL by performing the data copy and index update.

3.5.3 Supporting Local Data Copies

MEGALON can also consistently maintain local copies of data, where hosts store copies of an object in the host-local DRAM to avoid CXL accesses. MEGALON supports local data copies by having the index of an object in read-shared mode to point to multiple locations for the same object, one in LNR and others in local DRAM of hosts. MEGALON ensures coherence in the presence of data copies via the shared log. Specifically, when a host writes to an object that has multiple copies, it updates the index to point only to the CXL copy and changes the object to read-write-shared mode through the shared log. This logically invalidates the local copies because MEGALON synchronizes the index with the log tail on any host upon a subsequent read. This allows MEGALON to invalidate the local copies asynchronously without waiting for all hosts to acknowledge the invalidation. We call the variant of MEGALON that replicates objects locally MEGALON-DATA-COPY. MEGALON-DATA-COPY currently uses a simple policy to create local copies: it creates a local copy of an object at a host when the number of read accesses for that object from the host crosses a threshold. HCMeta cannot support local data copies because the index in SCR only tracks one location for each object.

4 Implementation

MEGALON is implemented as a C++ library in ~8K LOC. An application runs across hosts, and on each host, the application links against the MEGALON library. To ease application development, MEGALON library implements the *pre-read* and *post-read* checks and *pre-write* and *post-write* steps in read_start, read_end, write_start, and write_end calls. Applications invoke them before and after the actual read or write operation. It also exposes calls to create and delete objects.

The replicated index on each host is a hashmap with object-id as the key that maps to locations of the object in LNR and coherence record in SCR. MEGALON manages the space where the objects are stored as fixed-size slots. The entry of an object in the index points to the slot where the object lives. MEGALON currently takes the slot size as a compile-time parameter. MEGALON can be easily extended to support variable-sized objects at object creation time by keeping the object size as a field in the logically shared metadata. To support objects that can change size during their lifetime, MEGALON could treat them as delete-then-insert. A better approach to support dynamic object sizes with more efficient memory management is an avenue for future work.

To manage index replicas, MEGALON builds on the open-source Node Replication (NR) implementation [48]. NR can replicate any datastructure across NUMA nodes via a shared log, which occupies 32 MB by default; we treat the index hashmap as the NR datastructure. NR stores the shared log as a circular buffer. Updates to the replicated index are flat-combined [24] within a host and appended by a single thread per host to reduce contention on log tail. On each replica, NR protects the index with a reader-writer lock [13]. Before reading the index, a thread replays the log to bring its replica up to date, applying only fully-written entries and ignoring incomplete ones from concurrent writers. After all the replicas have replayed up to a certain log entry, the log head is safely incremented to recycle the log space. NR does not perform checkpointing since the log is used only for replication not recovery. This aligns well with MEGALON's failure model.

We make several modifications to NR. First, we allocate the log on CXL, with only the log tail and head in SCR, and perform non-temporal accesses to log entries. Second, we modify it to support the shared-log enabled coherence path. Specifically, when a replica replays the log, it flushes the CXL memory addresses of the slot contained in the log entries. Further, we add notifications from the log about events that modify the object (i.e., create, delete, coherence-record (de)allocations). For example, when MEGALON starts a read, it subscribes to such changes to the object; upon read end, MEGALON checks if there was an event, if so, retries the read.

The space for coherence records is fixed at initialization. MEGALON runs a background thread periodically to deallocate coherence records to keep the used ratio below a watermark. We use random sampling and recycle coherence records with the lowest counter (indicating objects with least writes).

As described in §3.5.2, MEGALON's policy to move an object to local DRAM is n consecutive accesses to an object. MEGALON does this by keeping the last access host to an object in LNR and keeps a per-object counter locally for tracking consecutive accesses. Keeping the last access host information

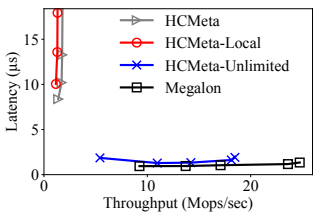
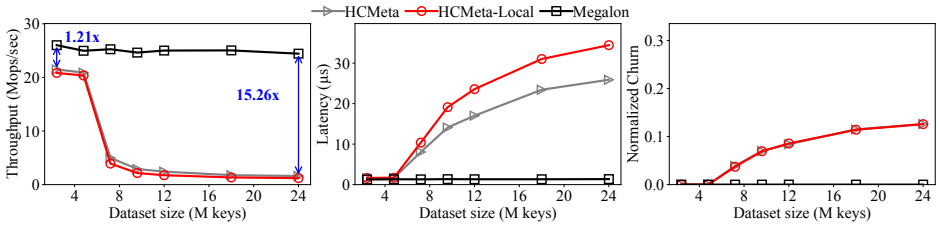


Figure 4: **Read-only Workload**



(a) Throughput (b) Latency (c) Churn

Figure 5: **Read-only performance: Growing Dataset Sizes**

in LNR is acceptable because the policy does not need to be precise. False positives can be limited by setting a reasonable *n*. Also, objects can be always brought back to CXL.

We adopt an open-source shared-memory RPC implementation [16] for general communications between hosts, such as requesting a partitioned object to be moved back to CXL. Underneath the RPC implementation, it uses a small pairwise multi-producer multi-consumer (MPMC) queue. Note that the communication method is orthogonal to the design of MEGALON, thus barely optimized. We can easily reduce the memory footprint of the MPMC queue by moving the queue entries to LNR and only keeping the head and tail in SCR. More extremely, we can even use the shared log for communication, which is already on CXL. We leave these optimizations for future work.

Currently, MEGALON is targeted to a system with 8–16 nodes as MEGALON's local memory footprint grows linearly with the number of nodes. To scale to a larger number of nodes, one could explore techniques such as rendering a partial view of the shared metadata to each node, or sharding the shared metadata by adopting multiple shared logs.

5 Applications

KV Store. The KV store exposes a linearizable put-get interface. Each KV pair is a MEGALON object, with the key serving as the object identifier. Put creates or updates the object via create, or write_start and write_end. Get reads the object and returns the value when read_end finishes without errors.

File-system Page Cache. This application treats each file page ([inode, block-number]) as a 4KB-sized MEGALON object. Here, not all data fits in CXL memory and so the application must evict pages to make space for new pages from disk. The application admits a page into the cache by reading it from disk and then calling MEGALON's create. If two threads admit a page, the second create fails. To evict a page, the application calls delete. The application chooses a victim using a per-object count, which is incremented by application threads upon accesses. Access counts need not be accurate, so the application keeps them in LNR. It runs a clock-like algorithm over the pages. The application writes back dirty pages to disk. The application maintains a dirty bit for each object in SCR, which it sets before it calls write_end. A background thread writes back dirty pages. It calls write_start on the dirty page (preventing other writes), flushes the page, clears the dirty bit, and calls write_end. Before a page is switched to read-shared mode, the application flushes the page to disk and clears the dirty bit, ensuring that the page is clean in read-shared mode.

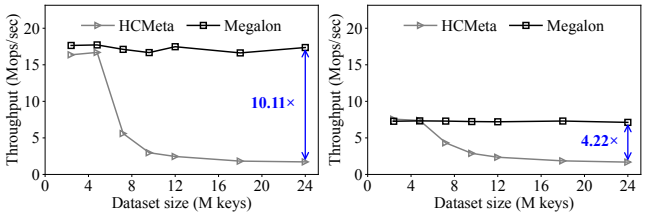
Failure Model. Like Tigon [26, §2], both KV store and page cache assume a fail-stop model. A failure of any host or CXL memory causes the entire system to fail. KV store is in-memory; our current implementation does not implement disk-based checkpointing. Our file-system page cache is disk-based and the file system can recover the state from the disk. Our page cache provides the same guarantees as existing file systems: all fsync-ed writes are guaranteed to be recovered.

6 Evaluation

We ask the following questions in our evaluation:

- How does MEGALON perform in read-only workload? (§6.1)
- How do dataset sizes impact read-only performance? (§6.2)
- How does MEGALON perform in read-write workload? (§6.3)
- How does MEGALON perform in cases where its coherence records do not fit in SCR? (§6.4)
- How does object identifier size affect performance? (§6.5)
- What does MEGALON pay for the higher performance? (§6.6)
- Do smaller coherence records improve performance? (§6.7)
- How do local-DRAM partitions enabled by the shared log help under partitioned workloads? (§6.8)
- How do local data copies enabled by the shared log improve performance? (§6.9)
- How does an approch that logically shares all metadata via replication perform compared to MEGALON's split-sharing approach? (§6.10)
- How does MEGALON perform in YCSB workloads? (§6.11)
- Does MEGALON help the page-cache application? (§6.12)

Setup. CXL 3.0 hardware is not commercially available, so we emulate CXL and hosts on a 4-socket server with Intel Xeon Gold 6418H processors. We designate NUMA node 0 as CXL memory and nodes 1-3 as hosts. Each emulated host has 24 physical cores with hyperthreading disabled and 64GB local DRAM. To approximate CXL latencies, we throttle the uncore frequency of node 0, as in prior work [3]. We distribute threads equally across hosts. We compare against three baselines: HCMeta, HCMeta-local, and HCMeta-Unlimited



(a) 5% write (b) 50% write

Figure 6: **Read-write performance: MEGALON vs. HCMeta**

(described in §2.3). All experiments by default runs on the KV Store application with 1KB objects. Unless stated otherwise, we use 200MB for the SCR (like in Tigon [26]).

6.1 Performance under Read-Only Workload

We first evaluate under a read-only workload with Zipfian access pattern (Zipf parameter 0.99), where most accesses are to some popular objects. We use a dataset of 18M objects. We vary the number of threads and plot throughput and average latency for MEGALON and baselines. Figure 4 shows the result.

As shown, HCMeta and HCMeta-Local do not scale because metadata doesn't fit in SCR at this dataset size, causing excessive churns. HCMeta-Local performs worse than HCMeta due to additional data-copy overhead during churns, but both are ultimately bottlenecked by churn. The performance of HCMeta-Unlimited, which has unlimited SCR, confirms that churn is the bottleneck; with unlimited SCR, it has no churns and achieves high throughput and low latency.

MEGALON's split design keeps the index out of SCR; further, since this is a read-only workload, it does not allocate any coherence records. Thus, MEGALON avoids churn entirely, achieving high performance. MEGALON's throughput exceeds that of HCMeta-Unlimited because accessing the physically shared metadata in HCMeta-Unlimited incurs higher overhead than accessing the local index replicas in MEGALON.

6.2 Read-only Performance with Growing Dataset Sizes

We next examine how dataset size affects performance under read-only workloads. We vary dataset size and measure throughput, latency, and normalized churn (the fraction of operations that incur churn). HCMeta variants churn when they share an unshared object; a churn in HCMeta-local additionally copies the data. In MEGALON, churn occurs when a read-shared object is converted to read-write-shared mode.

As shown in Figure 5, for small datasets, all three systems deliver high throughput and low latency, as seen in the two leftmost points of 5(a) and (b). This is because none of the systems incurs churn, as shown by the two leftmost points in 5(c). MEGALON has a small throughput advantage (about 1.2×) due to lower contention from its locally replicated indexes.

For large datasets, HCMeta and HCMeta-Local suffer from poor throughput and high latency because their churn rises with dataset size. HCMeta-Local has even higher latency due to data-copy overhead. MEGALON is unaffected by dataset size and offers 15× higher throughput because it does not allocate any coherence records in this workload, incurring no churn. Since HCMeta-Local is strictly worse than HCMeta, we omit it from future experiments (except §6.8).

6.3 Performance under Read-Write Workloads

We next compare performance under read-write Zipfian workloads with 5% and 50% writes. In these workloads, MEGALON must convert objects to read-write-shared mode by allocating coherence records. Figure 6 shows the result.

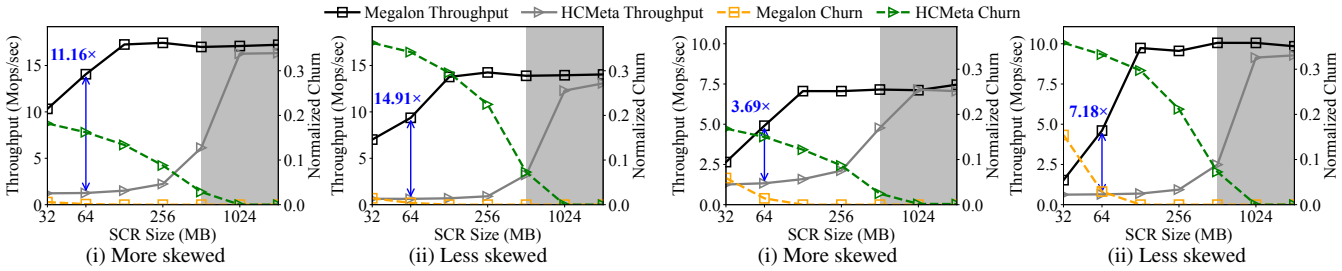
As shown in 6(a), with 5% writes, MEGALON sustains high throughput, while HCMeta's throughput drops significantly due to churn as the dataset grows. MEGALON avoids this drop because its split design uses SCR only for small coherence records, not the index (unlike HCMeta). Thus, even with 24M objects, MEGALON still has ample space left in SCR and incurs no churn, achieving 10× higher throughput than HCMeta. In fact, with a 200MB SCR, MEGALON can hold coherence records for up to 50M objects, which is 12× more than what HCMeta can support without churn. We later examine the case where MEGALON's coherence records no longer fit in SCR (§6.4).

With 50% writes (Figure 6(b)), both systems have lower absolute throughput because more writes cause additional lock contention and cache-line invalidations, reducing performance for both MEGALON and HCMeta. However, MEGALON still offers 4× higher throughput than HCMeta. Overall, even when MEGALON must allocate coherence records in read-write workloads, it still provides significant performance benefit.

6.4 Performance when Coherence Records Do Not Fit

We next examine the case where coherence records don't fit in SCR for MEGALON, causing churn under read-write workloads. Note that read-only workloads incur no churn in MEGALON. Churn frequency is driven by three factors: (1) dataset-to-SCR ratio, (2) write percentage, and (3) access skewness. Thus, we vary the SCR size while fixing the dataset at 18M objects, as well as the workload's write ratio and skewness. We examine two skewness levels (Zipf parameters): 0.99 (more skewed, YCSB default) and 0.7 (less skewed, used in Tigon [26]).

Figure 7(a)(i) shows throughput (left y-axis) and normalized churn (right y-axis) against SCR size for a 5%-write workload with high skew. For unrealistically large SCR sizes greater than 512MB [10, 27, 28] (the grey area), HCMeta performs well because, with such a large SCR, it incurs no churn (shown by the right y-axis). But as SCR becomes realistic (smaller than 512MB), HCMeta's throughput drops sharply. In contrast, MEGALON sustains its throughput until SCR reaches 128MB, which is an 8× smaller footprint than what HCMeta needs for high performance. Below 128MB, coherence records no longer fit, so MEGALON also incurs churn. However, MEGALON incurs far fewer churns because of the low write ratio in this workload. As a result, MEGALON still outperforms HCMeta even when MEGALON experiences churns; for instance, at 64MB SCR, it delivers 11× higher throughput.



(a) 5% write (b) 50% write

Figure 7: **Read-write performance of MEGALON vs. HCMeta with various SCR size**

Figure 7(a)(ii) shows the 5%-write workload with low skew. Because accesses are more uniform in this workload, HCMeta is more likely to access unshared objects and thus exhibits higher normalized churn than in the more skewed case, causing a sharper throughput drop. MEGALON's churn also rises compared to the more skewed workload, but remains far lower than HCMeta's, preserving its performance advantage.

Figure 7(b) presents results for 50% writes. When SCR exceeds 128MB, MEGALON incurs little to no churn; below 128MB, it sees more churn than in the 5%-write case. Thus, MEGALON shows a more pronounced throughput drop, yet still outperforms HCMeta. With low skew, the 50-% write case (7(b)(ii)) is the worst-case workload for MEGALON. However, even in this case, MEGALON continues to deliver much higher performance due to substantially fewer churns.

Churn Overhead. While MEGALON's lower churn frequency leads to higher performance than HCMeta, each churn in MEGALON is also cheaper, further boosting performance. Figure 8 shows MEGALON's lower churn overhead. A churn in MEGALON requires only allocating a coherence record and updating the index. In contrast, a churn in HCMeta occurs when a host accesses an unshared object, which triggers a shared-memory RPC to the host that owns the object. This adds substantial communication and synchronization cost. We also measured the churn overhead of HCMeta in the Tigon artifact [58], and found it be even higher ($\sim$55 μs).

In summary, MEGALON's split design greatly reduces precious SCR usage, allowing much higher throughput with far smaller SCR than HCMeta. Even when coherence records no longer fit, avoiding allocations for read-shared objects lets MEGALON sustain better performance. MEGALON's churns are also cheaper, further contributing to its performance gains.

6.5 Performance vs. Object Key Size

We next evaluate how object-identifier (i.e., key) size influences performance. Using a 4.8M-object dataset, we vary key size from 10 to 90 bytes under a read-only Zipfian workload. Figure 9 shows the throughput. Because HCMeta stores its index in SCR, its SCR footprint grows linearly with key size, leading to more churn and consequently lower throughput for larger keys. In contrast, MEGALON's split design keeps the index out of SCR, so key size does not affect SCR usage. Thus, MEGALON maintains high throughput across all key sizes.

6.6 Local-DRAM Usage and Performance

MEGALON's performance benefits come at the cost of higher local-DRAM usage for replicated indexes. This is justified for two reasons. First, the extra memory for index replicas is modest compared to coarse-granular (e.g., 1KB) objects. Take the 24M dataset point in Figure 5(a); each object has a 24-byte key and 1KB data. MEGALON uses 27.71GB memory, including the replicated indexes, host-side counters, coherence records, the shared log, and the object data. Compared to the 25.76GB footprint of HCMeta, MEGALON requires only 7.6% more memory. Second, the small added memory yields outsized improvements. As shown in 5(a), MEGALON reaches 24.4 MOps/s, achieving 0.88 MOps/s per GB of memory used. HCMeta, however, reaches only 1.6 MOps/s (or 0.062 MOps/s per GB). Thus, for just 7.6% more memory, MEGALON improves absolute performance by 15.25×, achieving 14.19× more throughput per GB of memory spent. With eight hosts, MEGALON's usage would be 33.51GB. Assuming similar performance, MEGALON would still have outsized benefits, offering 12.18× more throughput per GB than HCMeta.

6.7 Performance with Smaller Coherence Records

As shown in Figure 7(b)(ii), with very small SCR (32MB), high write ratios (50%), and less skew, MEGALON's performance drops due to churns. We now examine how making coherence records smaller improves performance for this worst-case workload. MEGALON's shared log enables this optimization: smaller records allow more coherence records to fit in SCR, delaying churn; the log safely handles counter wrap-arounds. We use a dataset of 18M objects and vary the size of coherence records by allocating them fewer bits. Figure 10 shows the result. As shown, throughput increases as we reduce the record size below the 32-bit default. With 8-bit records, throughput matches the maximum seen in Figure 7(b)(ii), as all records now fit in SCR. We observe that reducing below 8 bits hurts performance due to frequent wrap-around log events. Note that we use the default 32-bit records in all experiments; our current implementation does not dynamically pick the optimal size for a workload. However, the shared log provides the fundamental mechanism to explore this optimization.

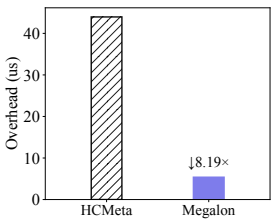


Figure 8: **Overhead of churns**

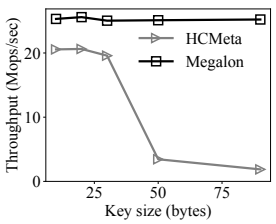


Figure 9: **Key size**

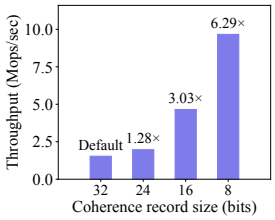


Figure 10: **Smaller coherence records**

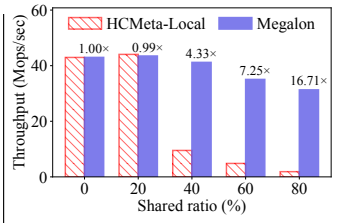


Figure 11: **Throughput vs. shared ratio**

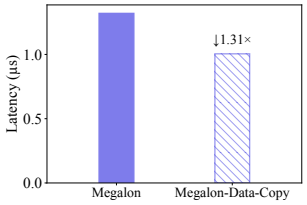


Figure 12: **Local replication**

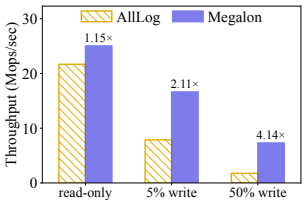


Figure 13: **AllLog variant**

6.8 Performance for Partitioned Workloads

We now evaluate the benefit of the local DRAM partitions enabled by the shared log. We compare MEGALON to HCMeta-Local, which keeps data in host-local DRAM and moves only shared objects in CXL. When hosts mostly access data from their own partitions, HCMeta-Local performs well. Using a read-only Zipfian workload on 18M objects, we vary the shared ratio (i.e., fraction of shared objects). For example, with 20% shared ratio, 20% objects are shared and 80% requests access exclusive data in host-local partitions. As shown in Figure 11, HCMeta-Local performs well at low sharing (0–20%) as it benefits from local DRAM access and little churn due to only small amount of sharing. MEGALON, by enabling local partitions, moves exclusively accessed objects to host-local DRAM and matches HCMeta-Local's performance at low sharing. As sharing increases, HCMeta-Local starts to churn and throughput drops, whereas MEGALON maintains significantly higher throughput by avoiding churns. Overall, by flexibly placing data between local DRAM and CXL, MEGALON matches HCMeta-Local under partitioned workloads and significantly outperforms it as sharing increases.

6.9 Benefit of Replicating Objects Locally

We now evaluate the benefit of replicating data objects locally. We skip comparison to HCMeta because it does not support local data copies as discussed in §3.5.3. We compare MEGALON-DATA-COPY to MEGALON under read-only Zipfian workload on 18M objects. As shown in Figure 12, MEGALON-DATA-COPY achieves 1.3× lower access latency than MEGALON because it avoids CXL accesses by replicating popular objects locally on hosts.

6.10 Performance of AllLog Variant

To showcase the benefit of the split metadata sharing approach, we implement a variant of MEGALON that does not split the metadata into physically shared coherence records and logically replicated indexes. Instead, it replicates all metadata through the shared log. We call this variant, AllLog. In this variant, lock acquisitions and counter increments create log events and are replayed on each metadata replica. We compare MEGALON against AllLog under a Zipfian workload with read-only, 5% writes, and 50% writes over 18M objects. As shown in Figure 13, AllLog performs similar to MEGALON under read-only workload because no locks are acquired or counters are incremented. However, as write ratio increases, AllLog incurs more log events, leading to higher logging overhead and lower throughput than MEGALON. With 50% writes, MEGALON achieves 4.14× higher throughput than AllLog. This demonstrates the benefit of the split approach: by keeping frequently updated metadata (the coherence records) in SCR, MEGALON optimizes performance by avoiding excessive log replays.

6.11 YCSB Macrobenchmark

We next analyze the performance under five YCSB [17] workloads: A (50% w, 50% r), B (5% w, 95% r), C (read-only), D (5% insert, 95% r; read latest), and F (50% read-modify-write, 50% r). The distribution is Zipfian and we use an 18M-object dataset. Figure 14 shows the result. Because HCMeta cannot fit the coherence records in SCR, it experiences churn across all workloads. In read-heavy B and C, MEGALON achieves the largest speedups (9.12×–14.18×). With A's 50% writes, contention reduces throughput of both systems, yielding a smaller benefit for MEGALON (3.93×). F shows the smallest gain (3.18×) because the writes immediately following the reads do not incur churns. D is the only workload where MEGALON requires shared-log updates for insertion. HCMeta incurs higher synchronization overhead because a host must contact the owner if the inserted object does not belong to the requesting host. MEGALON's shared-log-based insertion does not wait for other hosts and performs 4.55× better.

6.12 Page cache application

We finally evaluate the user-level shared page cache built with MEGALON. We run skewed workloads with different mix of reads and writes with a 48GB dataset of 4KB file pages.

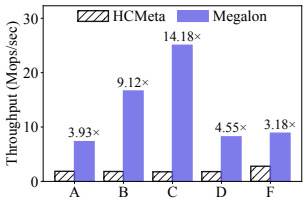


Figure 14: **YCSB workloads**

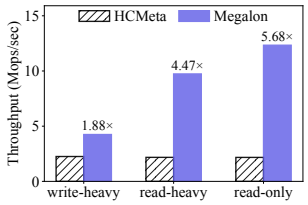


Figure 15: **Page cache**

As shown in Figure 15, HCMeta cannot share all pages as metadata does not fit in SCR; it suffers from churns and low performance. MEGALON is able to share a large page cache without churn, achieving notably higher throughput across all workloads. The throughput is lower than in the KV store due to larger object size. The page-cache application demonstrates that MEGALON can be incorporated into different applications to share large amounts of data in CXL with high performance.

7 Related Work

Disaggregated memory is an old idea [36] that has regained attention with CXL. Prior work on CXL mainly focus on *memory tiering* and *pooling*. Tiering solutions [32, 42, 55, 61–63] seek to hide CXL's higher latency by moving hot data to DIMM-attached memory. Pooling approaches [22, 33] improve memory utilization by apportioning CXL memory to hosts that need it. Other work [23, 33, 38, 56, 57] analyze CXL's raw performance and implications to application performance. None of the above considers CXL memory sharing.

A few recent papers consider sharing CXL memory across hosts [10, 26–28]. They note that cache coherence for all CXL memory is infeasible, motivating an SCR model. Tigon [26] shows how to build a database in this model, by moving objects to be shared from local memory to CXL. To share objects in CXL, Tigon uses hardware-coherent metadata-based sharing (HCMeta). The work in [28] outlines high-level sharing ideas for different CXL versions but provides few technical details. CXLfork [2] proposes remote forks via CXL, where the forked process shares read-only pages with its parent. Unlike our work that enables general data sharing across hosts, CXLfork uses sharing only for deduplicating memory between parent and child, and supports only read sharing.

Systems without any cache coherence have been considered before [5, 15, 30, 44]. However, such systems create significant complexity for applications, and so they have not been widely adopted. Cosh [8] enables transferring bulk data across CPUs, GPUs, and other accelerators that may share memory incoherently. That work has a different goal from ours, namely, bulk data transfers rather than data sharing.

RDMA also enables disaggregated memory, but unlike CXL, it requires processes to issue IO to access disaggregated memory, rather than loads and stores. RDMA-based disaggregated memory has been used to build object and KV stores [20, 21, 29, 35, 40, 45, 53]. However, these systems are unaffected by the lack of cache coherence in CPU caches, since RDMA operations are IO operations. As a consequence, RDMA systems require software-managed caches to avoid remote memory accesses, whereas CXL allows hosts to directly benefit from hardware caches.

Node replication (NR) [13] efficiently replicates data structures across NUMA domains using a log in shared memory, while subsequent work used NR to replicate page tables in the OS [11]. That work was not targeted at systems without full cache coherence. We borrow NR to replicate the index in MEGALON and apply it to our setting with an SCR, by observing that NR works correctly provided that the log tail pointer is in the SCR; even the log can be in incoherent memory provided threads flush or bypass the cache when accessing log entries.

The idea of replicating indexes has appeared in other contexts. DudeTM [39] mirrors durable memory pages in persistent memory to local DRAM, including the index, to speed up transaction execution by avoiding expensive persistent-memory accesses. HydraList [43], similar to NR, replicates the search index across NUMA nodes to avoid remote-memory stalls and cache-coherence traffic. It updates the search index asynchronously by aggregating operations and broadcasting them to all replicas. However, both works use index replication to improve performance, whereas our work uses it to conserve scarce SCR space.

Software-based distributed shared memory (DSM) systems (e.g., [1, 4, 9, 12, 34, 41, 46, 49, 51]) emulate shared memory among hosts connected by a network without any physical shared memory. These systems do not have an SCR to leverage, so they provide coherence entirely in software via message-passing protocols, which are known to be slow.

8 Conclusion

We propose MEGALON, a new data-sharing approach for partly coherent CXL. MEGALON shares the metadata essential for coherent data sharing using a novel split approach that is not limited by hardware-coherent memory capacity. MEGALON uses a CXL shared log to enable dynamic coherence records and provides coherence using the shared log when coherence records cannot. MEGALON improves performance compared to the current sharing approach under many workloads.

Acknowledgments

We thank our shepherd and the OSDI '26 reviewers for their insightful comments. We also thank the other DASSL members for their discussions and feedback. We thank Neil Kaushikkar for his contributions to the initial implementation. This material was supported by funding from NSF grants CNS-2340218 and CNS-2339784, an IIDAI grant, as well as gifts from NetApp and TigerBeetle.

References

[1] Anant Agarwal, Ricardo Bianchini, David Chaiken, Kirk L Johnson, David Kranz, John Kubiatowicz, Beng-Hong Lim, Kenneth Mackenzie, and Donald Yeung. The MIT Alewife machine: Architecture and performance. *ACM SIGARCH Computer Architecture News*, 23(2):2–13, 1995.

[2] Chloe Alverti, Stratos Psomadakis, Burak Ocalan, Shashwat Jaiswal, Tianyin Xu, and Josep Torrellas. CXLfork: Fast remote fork over CXL fabrics. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 210–226, March 2025.

[3] Emmanuel Amaro, Stephanie Wang, Aurojit Panda, and Marcos K Aguilera. Logical memory pools: Flexible and local disaggregated memory. In *Proceedings of the 22nd ACM Workshop on Hot Topics in Networks*, pages 25–32, 2023.

[4] Cristiana Amza, Alan L. Cox, Shandya Dwarkadas, Pete Keleher, Honghui Lu, Ramakrishnan Rajamony, Weimin Yu, and Willy Zwaenepoel. TreadMarks: Shared memory computing on networks of workstations. *IEEE Computer*, 29(2):18–28, February 1996.

[5] ARM922T technical reference manual. https://developer.arm.com/documentation/ddi0184/b/caches--write-buffer--and-physical-address-tag--pa-tag--ram/cache-coherence.

[6] Mahesh Balakrishnan, Dahlia Malkhi, Vijayan Prabhakaran, Ted Wobber, Michael Wei, and John D. Davis. CORFU: A shared log design for flash clusters. In *Proceedings of the 9th Symposium on Networked Systems Design and Implementation (NSDI '12)*, San Jose, CA, April 2012.

[7] Mahesh Balakrishnan, Dahlia Malkhi, Ted Wobber, Ming Wu, Vijayan Prabhakaran, Michael Wei, John D Davis, Sriram Rao, Tao Zou, and Aviad Zuck. Tango: Distributed data structures over a shared log. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP '13)*, Farmington, Pennsylvania, October 2013.

[8] Andrew Baumann, Chris Hawblitzel, Kornilios Kourtis, Tim Harris, and Timothy Roscoe. Cosh: clear OS data sharing in an incoherent world. In *Conference on Timely Results in Operating Systems (TRIOS 2014)*, October 2014.

[9] J. K. Bennett, J. B. Carter, and W. Zwaenepoel. Munin: Distributed shared memory based on type-specific memory coherence. In *Symposium on Principles and Practice of Parallel Programming*, pages 168–176, March 1990.

[10] Daniel S. Berger. Realistic Expectations for CXL Memory Pools. Talk presented at the DIMES 2024 workshop.

[11] Ankit Bhardwaj, Chinmay Kulkarni, Reto Achermann, Irina Calciu, Sanidhya Kashyap, Ryan Stutsman, Amy Tai, and Gerd Zellweger. NrOS: Effective replication and sharing in an operating system. In *USENIX Symposium on Operating Systems Design and Implementation*, pages 295–312, July 2021.

[12] Qingchao Cai, Wentian Guo, Hao Zhang, Divyakant Agrawal, Gang Chen, Beng Chin Ooi, Kian-Lee Tan, Yong Meng Teo, and Sheng Wang. Efficient distributed memory management with RDMA and caching. *Proceedings of the VLDB Endowment*, 11(11):1604–1617, 2018.

[13] Irina Calciu, Siddhartha Sen, Mahesh Balakrishnan, and Marcos K Aguilera. Black-box concurrent data structures for NUMA architectures. *ACM SIGPLAN Notices*, 52(4):207–221, 2017.

[14] Zhichao Cao, Siying Dong, Sagar Vemuri, and David H.C. Du. Characterizing, modeling, and benchmarking RocksDB key-value workloads at Facebook. In *Proceedings of the 18th USENIX Conference on File and Storage Technologies (FAST '20)*, Santa Clara, CA, February 2020.

[15] Nicholas P Carter, Aditya Agrawal, Shekhar Borkar, Romain Cledat, Howard David, Dave Dunning, Joshua Fryman, Ivan Ganev, Roger A Golliver, Rob Knauerhase, et al. Runnemede: An architecture for ubiquitous high-performance computing. In *IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 198–209, 2013.

[16] Jon Chesterfield. HostRPC. https://github.com/JonChesterfield/hostrpc.

[17] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with YCSB. In *Proceedings of the ACM Symposium on Cloud Computing (SOCC '10)*, Indianapolis, IN, June 2010.

[18] Compute Express Link (CXL) Specification, Revision 3.0. https://computeexpresslink.org/wp-content/uploads/2024/02/CXL-3.0-Specification.pdf.

[19] Compute Express Link (CXL) Specification, Revision 3.1. https://computeexpresslink.org/wp-content/uploads/2024/02/CXL-3.1-Specification.pdf.

[20] Aleksandar Dragojević, Dushyanth Narayanan, Miguel Castro, and Orion Hodson. FaRM: Fast remote memory. In *USENIX Symposium on Networked Systems Design and Implementation*, pages 401–414, April 2014.

[21] Aleksandar Dragojević, Dushyanth Narayanan, Ed Nightingale, Matthew Renzelmann, Alex Shamis, Anirudh Badam, and Miguel Castro. No compromises: distributed transactions with consistency, availability, and performance. In *Symposium on Operating Systems Principles*, pages 54–70, October 2015.

[22] Donghyun Gouk, Miryeong Kwon, Hanyeoreum Bae, Sangwon Lee, and Myoungsoo Jung. Memory pooling with CXL. *IEEE Micro*, 43(2):48–57, March 2023.

[23] Donghyun Gouk, Sangwon Lee, Miryeong Kwon, and Myoungsoo Jung. Direct access, high-performance memory disaggregation with DirectCXL. In *USENIX Annual Technical Conference*, July 2022.

[24] Danny Hendler, Itai Incze, Nir Shavit, and Moran Tzafrir. Flat combining and the synchronization-parallelism tradeoff. In *Proceedings of the Twenty-Second Annual ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '10, page 355–364, New York, NY, USA, 2010. Association for Computing Machinery.

[25] Jaewan Hong, Marcos K Aguilera, Emmanuel Amaro, Vincent Liu, Aurojit Panda, and Ion Stoica. The dawn of disaggregation and the coherence conundrum: A call for federated coherence. *arXiv preprint arXiv:2504.16324*, 2025.

[26] Yibo Huang, Haowei Chen, Newton Ni, Vijay Chidambaram, Dixin Tang, Emmett Witchel, Zhiting Zhu, and Zhipeng Jia. Tigon: A distributed database for a CXL pod. In *Proceedings of the 19th USENIX Conference on Operating Systems Design and Implementation (OSDI '25)*, Boston, MA, July 2025.

[27] Yibo Huang, Newton Ni, Vijay Chidambaram, Emmett Witchel, and Dixin Tang. Pasha: An efficient, scalable database architecture for CXL pods. In *Annual Conference on Innovative Data Systems Research (CIDR 2025)*, January 2025.

[28] Sunita Jain, Nagaradhesh Yeleswarapu, Hasan Al Maruf, and Rita Gupta. Memory sharing with CXL: Hardware and software design approaches. *arXiv preprint arXiv:2404.03245*, 2024.

[29] Anuj Kalia, Michael Kaminsky, and David G. Andersen. Using RDMA efficiently for key-value services. In *ACM Special Interest Group on Data Communications*, August 2014.

[30] Wooil Kim, Sanket Tavarageri, P Sadayappan, and Josep Torrellas. Architecting and programming a hardware-incoherent multiprocessor cache hierarchy. In *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 555–565, 2016.

[31] Leslie Lamport. Paxos made simple. *ACM Sigact News*, 32(4):18–25, 2001.

[32] Taehyung Lee, Sumit Kumar Monga, Changwoo Min, and Young Ik Eom. MEMTIS: Efficient memory tiering with dynamic page classification and page size determination. In *Symposium on Operating Systems Principles*, pages 17–34, October 2023.

[33] Huaicheng Li, Daniel S Berger, Lisa Hsu, Daniel Ernst, Pantea Zardoshti, Stanko Novakovic, Monish Shah, Samir Rajadnya, Scott Lee, Ishwar Agarwal, et al. Pond: CXL-based memory pooling systems for cloud platforms. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, pages 574–587, 2023.

[34] Kai Li and Paul Hudak. Memory coherence in shared virtual memory systems. *ACM Transactions on Computer Systems (TOCS)*, 7(4):321–359, 1989.

[35] Pengfei Li, Yu Hua, Pengfei Zuo, Zhangyu Chen, and Jiajie Sheng. A high-performance RDMA-oriented learned key-value store for disaggregated memory systems. *ACM Transactions on Storage*, 19(4):1–30, October 2023.

[36] Kevin Lim, Jichuan Chang, Trevor Mudge, Parthasarathy Ranganathan, Steven K Reinhardt, and Thomas F Wenisch. Disaggregated memory for expansion and sharing in blade servers. In *International Symposium on Computer Architecture*, pages 267–278, June 2009.

[37] Barbara Liskov and James Cowling. Viewstamped replication revisited. Technical Report MIT-CSAIL-TR-2012-021, MIT, July 2012.

[38] Jinshu Liu, Hamid Hadian, Yuyue Wang, Daniel S. Berger, Marie Nguyen, Xun Jian, Sam H. Noh, and Huaicheng Li. Systematic CXL memory characterization and performance analysis at scale. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 1203–1217, March 2025.

[39] Mengxing Liu, Mingxing Zhang, Kang Chen, Xuehai Qian, Yongwei Wu, Weimin Zheng, and Jinglei Ren. DudeTM: Building durable transactions with decoupling for persistent memory. In *Proceedings of the*

Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS '17, page 329–343, New York, NY, USA, 2017. Association for Computing Machinery.

[40] Yi Liu, Minghao Xie, Shouqian Shi, Yuanchao Xu, Heiner Litz, and Chen Qian. Outback: Fast and communication-efficient index for key-value store on disaggregated memory. *Proceedings of the VLDB Endowment*, 18(2):335–348, October 2024.

[41] Haoran Ma, Yifan Qiao, Shi Liu, Shan Yu, Yuanjiang Ni, Qingda Lu, Jiesheng Wu, Yiying Zhang, Miryung Kim, and Harry Xu. DRust: Language-guide distributed shared memory with fine granularity, full transparency, and ultra efficiency. In *USENIX Symposium on Operating Systems Design and Implementation*, July 2024.

[42] Hasan Al Maruf, Hao Wang, Abhishek Dhanotia, Johannes Weiner, Niket Agarwal, Pallab Bhattacharya, Chris Petersen, Mosharaf Chowdhury, Shobhit Kanaujia, and Prakash Chauhan. TPP: Transparent page placement for CXL-enabled tiered-memory. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, page 742–755, 2023.

[43] Ajit Mathew and Changwoo Min. HydraList: a scalable in-memory index using asynchronous updates and partial replication. *Proc. VLDB Endow.*, 13(9):1332–1345, May 2020.

[44] André Maximo, Guilherme Cox, Cristiana Bentes, and Ricardo Farias. Unleashing the power of the Playstation 3 to boost graphics programming. In *Tutorials of the XXII Brazilian Symposium on Computer Graphics and Image Processing*, pages 45–58, 2009.

[45] Christopher Mitchell, Yifeng Geng, and Jinyang Li. Using one-sided RDMA reads to build a fast, CPU-efficient key-value store. In *USENIX Annual Technical Conference*, June 2013.

[46] Jacob Nelson, Brandon Holt, Brandon Myers, Preston Briggs, Luis Ceze, Simon Kahan, and Mark Oskin. Latency-tolerant software distributed shared memory. In *USENIX Annual Technical Conference*, pages 291–305, July 2015.

[47] Newton Ni, Yan Sun, Zhiting Zhu, and Emmett Witchel. Cxlalloc: Safe and efficient memory allocation for a CXL pod. In *Proceedings of the 31st ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, ASPLOS '26, page 528–545, New York, NY, USA, 2026. Association for Computing Machinery.

[48] Node Replication. https://github.com/vmware/node-replication.

[49] Daniel J. Scales, Kourosh Gharachorloo, and Chandramohan A. Thekkath. Shasta: A low overhead, software-only approach for supporting fine-grain shared memory. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 174–185, October 1996.

[50] Fred B. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Comput. Surv.*, 22(4):299–319, December 1990.

[51] Ioannis Schoinas, Babak Falsafi, Alvin R. Lebeck, Steven K. Reinhardt, James R. Larus, and David A. Wood. Fine-grain access control for distributed shared memory. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 297–306, October 1994.

[52] Sequence counters and sequential locks. https://docs.kernel.org/locking/seqlock.html.

[53] Jiacheng Shen, Pengfei Zuo, Xuchuan Luo, Tianyi Yang, Yuxin Su, Yangfan Zhou, and Michael R. Lyu. FUSEE: A fully memory-disaggregated key-value store. In *Proceedings of the 21st USENIX Conference on File and Storage Technologies*, FAST '23, pages 81–98, Berkeley, CA, USA, 2023. USENIX Association.

[54] Daniel Sorin, Mark Hill, and David Wood. *A Primer on Memory Consistency and Cache Coherence*. Morgan & Claypool Publishers, 2011.

[55] Yan Sun, Jongyul Kim, Zeduo Yu, Jiyuan Zhang, Siyuan Chai, Michael Jaemin Kim, Hwayong Nam, Jaehyun Park, Eojin Na, Yifan Yuan, Ren Wang, Jung Ho Ahn, Tianyin Xu, and Nam Sung Kim. M5: Mastering page migration and memory management for CXL-based tiered memory systems. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 604–621, March 2025.

[56] Yan Sun, Yifan Yuan, Zeduo Yu, Reese Kuper, Chihun Song, Jinghan Huang, Houxiang Ji, Siddharth Agarwal, Jiaqi Lou, Ipoom Jeong, Ren Wang, Jung Ho Ahn, Tianyin Xu, and Nam Sung Kim. Demystifying CXL memory with genuine CXL-ready systems and devices. In *IEEE/ACM International Symposium on Microarchitecture*, pages 105–121, December 2023.

[57] Yupeng Tang, Ping Zhou, Wenhui Zhang, Henry Hu, Qirui Yang, Hao Xiang, Tongping Liu, Jiaxin Shan, Ruoyun Huang, Cheng Zhao, Cheng Chen, Hui Zhang, Fei Liu, Shuai Zhang, Xiaoning Ding, and Jianjun Chen. Exploring performance and cost optimization with ASIC-based CXL memory. In *European Conference on Computer Systems*, pages 818–833, April 2024.

[58] Tigon artifact. https://github.com/ut-datasys/tigon, 2025.

[59] Emmett Witchel. Challenges and opportunities for systems using CXL. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, 2024.

[60] Fangnuo Wu, Mingkai Dong, Wenjun Cai, Jingsheng Yan, and Haibo Chen. Guidelines for building indexes on partially cache-coherent CXL shared memory, 2025.

[61] Lingfeng Xiang, Zhen Lin, Weishu Deng, Hui Lu, Jia Rao, Yifan Yuan, and Ren Wang. NOMAD: non-exclusive memory tiering via transactional page migration. In *USENIX Symposium on Operating Systems Design and Implementation*, pages 19–35, July 2024.

[62] Yuhong Zhong, Daniel S. Berger, Carl Waldspurger, Ryan Wee, Ishwar Agarwal, Rajat Agarwal, Frank Hady, Karthik Kumar, Mark D. Hill, Mosharaf Chowdhury, and Asaf Cidon. Managing memory tiers with CXL in virtualized environments. In *USENIX Symposium on Operating Systems Design and Implementation*, pages 37–56, July 2024.

[63] Zhe Zhou, Yiqi Chen, Tao Zhang, Yang Wang, Ran Shu, Shuotao Xu, Peng Cheng, Lei Qu, Yongqiang Xiong, Jie Zhang, and Guangyu Sun. NeoMem: Hardware/software co-design for CXL-native memory tiering. In *IEEE/ACM International Symposium on Microarchitecture*, pages 1518–1531, November 2024.

[64] Zhiting Zhu, Yibo Huang, Yan Sun, Zhipeng Jia, Nam Sung Kim, and Emmett Witchel. Lupin: Tolerating partial failures in a CXL pod. In *Workshop on Disruptive Memory Systems*, pages 41–50, November 2024.

Inference in the Shadows: Taming Memory Bandwidth Contention in Mobile LLM Inference with SERENO

Tong Xin Xinrui Shi Mingkai Dong Zeyu Mi
Institute of Parallel and Distributed Systems, Shanghai Jiao Tong University

Abstract

The proliferation of large language models (LLMs) on mobile devices introduces a new performance challenge: resource contention between compute-intensive inference and latency-sensitive foreground applications. We identify a severe and *asymmetric* interference where concurrent LLM inference substantially degrades foreground applications' quality-of-service (QoS)—increasing the aggregate jank rate (the fraction of frames that appear as visible stutters) by 153%. In contrast, LLM throughput degrades by only 1.01% and 1.64% during prefill and decode stages, respectively. This imbalance arises because the hardware prioritizes NPU memory traffic—originally to guarantee critical media tasks (e.g., video recording)—a privilege that best-effort LLM inference inherits, causing aggressive bandwidth contention.

To address this asymmetric degradation, we present SERENO, a foreground-QoS-friendly LLM inference framework that resolves bandwidth contention between foreground applications and background LLM inference, without hardware modification. SERENO repurposes speculative decoding to introduce fine-grained yield points for preemptible execution, letting the system detect memory contention and dynamically yield bandwidth to the foreground without losing inference progress. Extensive evaluations on commercial smartphones across diverse categories of popular applications demonstrate that SERENO reduces the foreground jank rate by up to 92.6% (58.5% on average) while boosting LLM throughput by up to 67.9% (26.4% on average). Compared with vanilla speculative decoding, SERENO can reduce the foreground jank rate by up to 72.1% while incurring only a 6.2% performance degradation.

1 Introduction

Large language models (LLMs) are increasingly deployed on real-world mobile devices [1,54,58,106]. These models power a growing range of shipping on-device features: notification summarization and smart reply (Apple Intelligence [11]), real-time call translation and chat assistance (Samsung Galaxy AI [77]), and AI-powered on-screen search (Google Gemini Nano [28]) [63]. Beyond these, mobile agents [53,93,95,99] further expand the role of on-device inference, executing tasks such as form filling, calendar management, and cross-app automation. These features are not necessarily continuous services; many are sporadic background bursts triggered by notifications, messages, translation requests, or agent subtasks. They nevertheless overlap with foreground rendering at millisecond timescales, where even short memory bursts can violate frame deadlines.

However, as a resource-intensive background workload, LLM inference on smartphones contends for shared resources and interferes with user-visible tasks (e.g., UI rendering). Our analysis of co-running an LLM with 25 popular mobile applications, one foreground app at a time, reveals that the aggregate foreground *jank rate* surges by 153% and the system's peak CPU and GPU performance regresses to levels comparable to smartphones released up to three years ago—translating into substantially more frequent perceptible stutters, jitters, and input lags. Crucially, we identify a severe *asymmetric interference pattern*: while foreground performance degrades sharply, the degradation of LLM inference throughput is negligible. Specifically, when co-running LLM inference with applications, prefill and decode throughput decrease by only 1.01% and 1.64%, respectively.

To pinpoint the root cause, we profiled hardware subsystems: the degradation is attributable neither to CPU/GPU saturation nor to cache contention, but to a sharp spike in memory access latency during concurrent inference—i.e., severe contention for shared DRAM bandwidth within the mobile SoC's Unified Memory Architecture (UMA).

By further analyzing the software-hardware stack of various mainstream mobile SoCs, we reveal that the asymmetry originates from hardware-level prioritization of the NPU's memory traffic. Crucially, this is not a design oversight, but a consequence of legacy real-time design principle: mobile SoCs have traditionally treated accelerators (e.g., ISP and NPU) as latency-critical media engines and granted them high-priority access to memory. Consequently, best-effort LLM inference inadvertently inherits this privilege, acquiring

the same dominant memory bandwidth allocation as latency-critical media tasks (e.g., video recording). As a result, current system resource arbitration paradoxically prioritizes LLM inference, causing disproportionate harm to foreground tasks.

Preserving user-visible foreground performance necessitates resolving bandwidth contention between interactive foreground applications and background LLM inference. Specifically, it is imperative to enable LLM inference to dynamically yield memory bandwidth to foreground applications.

However, achieving this on commodity devices is impeded by the rigidity of current mobile SoCs, which pose three fundamental challenges: First, *lack of low-overhead bandwidth sensing*. Real-time contention sensing requires high-frequency, precise signals. Most mobile SoCs lack such hardware counters, while software methods incur prohibitive overhead for online deployment. Second, *lack of high-frequency LLM control knobs*. Mobile NPU inference typically executes static computation graphs to maximize efficiency, treating inference as an uninterruptible atomic operation. Third, *lack of efficient bandwidth regulation*. Commodity SoCs lack mechanisms for fine-grained NPU throttling. Coarse-grained regulation methods (e.g., sleeping or frequency scaling) are inefficient and underutilize bandwidth.

Our key insight is that *speculative decoding*—an inference acceleration technique that decomposes LLM decoding into draft and verification phases—provides natural opportunities to introduce yield points, enabling it to be repurposed to regulate bandwidth contention without losing inference progress.

Based on this insight, we propose SERENO, a software-only, foreground-QoS-preserving LLM inference framework that regulates bandwidth contention. First, to enable sensing, we derive proxy signals from the latency of execution blocks during the draft phase of speculative decoding. Observing a strong positive correlation ($R \approx 0.86$) between this proxy signal and physical memory bandwidth usage, we leverage it as an endogenous, near-zero-overhead contention probe. Next, SERENO introduces *Elastic Speculative Decoding* to transform the draft phase of speculative decoding into fine-grained, preemptible control knobs. Finally, we introduce a control system consisting of three bandwidth regulation primitives. It dynamically orchestrates the primitives' combination and intensity based on real-time contention, maximizing preserved progress while releasing bandwidth.

We implement SERENO and conduct extensive evaluations on commercial smartphones across diverse categories of popular applications. SERENO substantially mitigates the foreground QoS degradation caused by bandwidth contention.

Relative to an NPU inference baseline, SERENO reduces the foreground jank rate by up to 92.6% (58.5% on average) while boosting LLM throughput by up to 67.9% (26.4% on average). When compared with vanilla speculative decoding, SERENO can lower the jank rate by up to 72.1% with only a minor 6.2% performance penalty. Furthermore, SERENO outperforms the *llama.cpp* NPU backend, achieving 37.1% lower jank and 82.2% higher throughput.

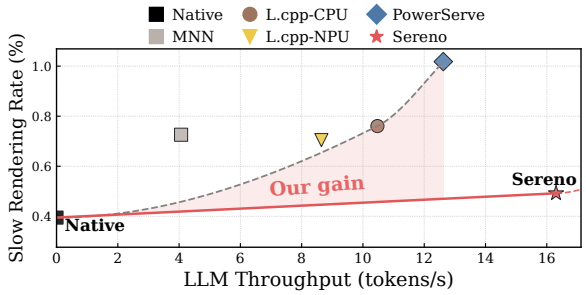


Figure 1: The New Pareto Frontier. Unlike systems that trade off LLM throughput for foreground smoothness, SERENO achieves both. Data are from co-running inference with 25 apps, one at a time.

Our contributions can be summarized as follows:

- We are the first to characterize and attribute the asymmetric performance interference between on-device LLM inference and concurrent foreground workloads.
- We provide systematic root-cause analysis of fundamental defects in mobile SoC unified resource management under LLM workloads.
- We propose a novel repurposing of speculative decoding, transforming it from an acceleration technique into a mechanism that creates fine-grained scheduling windows for QoS management.
- We present SERENO, the first foreground-QoS-aware LLM inference framework. On commercial smartphones, SERENO restores foreground smoothness to near-native levels while improving LLM throughput, achieving a "win-win" for user experience and performance (Figure 1).

2 Background

2.1 LLM Inference and Speculative Decoding

LLM inference comprises two phases: a compute-bound *prefill* phase that processes the prompt to populate KV caches, and a memory-bound *decode* phase. The decode phase generates tokens iteratively; since each step requires reading full model weights to produce a single token, performance is limited by memory bandwidth rather than arithmetic throughput.

To accelerate the decode phase, *speculative decoding* is adopted: a lightweight *draft model* proposes a sequence of candidate tokens, which the full *target model* then validates in parallel via a single large-batch forward (the *draft* and *verify* phases). If verification confirms the draft prefix, multiple tokens are committed at once; otherwise the process rolls back to the last valid token. This amortizes the high memory access cost over multiple tokens, typically decoding faster than autoregressive generation.

2.2 Mobile SoC Architecture and Rendering

Neural Processing Units (NPUs) are designed for compute-intensive tasks like AI, imaging, and sensor processing. Mobile inference is usually offloaded to NPUs [56, 72]. To maxi-

mize throughput, NPUs execute pre-compiled static computation graphs. Once submitted, a graph executes autonomously within the hardware pipeline as an atomic, run-to-completion unit. Critically, modern mobile SoCs adopt *Unified Memory Architecture* (UMA), i.e., CPUs, GPUs, and NPUs share a single DRAM pool and system interconnect. Memory access to unified memory is mediated by hardware priority arbitration.

The Android UI rendering pipeline is latency-bounded. Subsystems like *Choreographer* and *SurfaceFlinger* [10] coordinate CPU input handling, app-level drawing, and GPU composition. If the system skips frames, the user perceives recurring flicker on the screen, referred to as *jank* [7]. Jank encompasses two categories: ❶ *buffer stuffing*, where frames queue up due to backpressure, manifesting as input lag; and ❷ *deadline misses*, where rendering exceeds the Vsync interval, causing visible stutter. We denote the latter as *slow rendering* to isolate visual jitter from latency.

3 Interference and Analysis

3.1 Quantifying Interference

Though mobile LLM inference isolates computation on the NPU, it shares other resources with the rest of the OS. To quantify interference, we profile a OnePlus 13 (Snapdragon 8 Elite) running Llama-3.1-8B (W4A16) [31] using the *PowerServe* [66] alongside each of 25 popular foreground apps individually (selected via previous studies [69, 97] and Play Store rankings). Each ∼30 s run involves continuous swiping, with 1024 prefill tokens and 256 decode tokens.

Metrics. We focus on user-perceived fluidity. Using the *Perfetto* [30] toolchain, we report *jank rate* (percentage of stutter frames [7]) and the stricter *slow rendering rate* (frames missing the Vsync deadline due to execution delays [87]). Notably, slow rendering produces user-visible jitters, and thus is used by Google as a quality metric directly tied to app discoverability on Google Play [8].

Foreground QoS degradation. Figure 2 (left panel) visualizes the impact on user experience, where red bars represent the jank rate normalized to the native baseline (dashed line at 1.0). While the baseline implies no-LLM-inference smooth operation, the red bars for latency-sensitive apps increase sharply under concurrency. For instance, *Discord* reaches nearly 18× the native jank rate. This confirms that background LLM inference substantially degrades QoS across apps, causing large relative regressions even in highly optimized, lightweight apps. The aggregate jank rate of the 25 apps increases by 153%, while the mean per-app normalized jank rate shown in Figure 2 reaches 3.13×, and the slow rendering rate shows the same degradation trend—a stuttering, unresponsive experience.

System peak performance loss. The interference extends to system peak performance, as detailed in Figure 2 (right panel). Here, the grey bars depict benchmark scores normalized to the native baseline. Across both CPU (e.g., GB6 CPU [68]) and GPU (e.g., 3DM Nomad [88]) workloads, the significant drop in bars induces a 37.2%–49.6% loss in peak throughput. Effectively, the Snapdragon 8 Elite performs at the level of hardware from two to three years ago [67, 89], bringing a $1000 flagship down to the performance tier of a $400 device [4, 5].

Asymmetric performance impact. Crucially, the graphs highlight a severe asymmetry. In both panels of Figure 2, the *blue lines* representing LLM prefill and decode speeds stay almost flat near the 1.0 baseline. For example, while foreground performance (red bars) degrades sharply, background prefill/decode throughput degrades by only 1.01% and 1.64% on average. The same trend appears in benchmark scores. This implies a systemic priority inversion where latency-critical UI tasks are starved to feed a best-effort background workload.

3.2 Physical Attribution: Bandwidth

To identify the contended resource, we employ an elimination method using *Simpleperf* [9] and *Snapdragon Profiler* [76].

On the Snapdragon 8 Elite, although matrix computation is offloaded to NPU cores (HMX [71]), the data and control paths remain entangled with the host: high-volume DMA between NPU-internal cache and DRAM traverses the same interconnect used by the CPU and GPU, the CPU retains the inference control plane, and NPU vector units (HVX) interact directly with the cache hierarchy.

Compute and power metrics. Figure 3 plots the normalized performance ratios of key hardware metrics, where a value of 1.0 (dashed line) indicates the contention-free baseline. Under LLM concurrency, we observe that foreground compute metrics—specifically *IPC*, *CPU Util*, and *GPU Util* (green bars)—hover near 1.0. This stability confirms that the CPU and GPU are not saturated by the background task. Furthermore, we rule out thermal throttling, as core frequencies show no abnormal drops.

Cache metrics. Similarly, the blue bars of Figure 3 examine the cache hierarchy. The normalized miss rates for *L2 Cache*, *Last-Level Cache (LLC)*, and *TLB* remain largely unchanged relative to the native baseline. This flat trend indicates that NPU traffic does not significantly pollute the shared caches or thrash TLB entries, ruling out cache contention as the primary bottleneck.

Unified memory bandwidth metrics. In sharp contrast, the red bars of Figure 3 reveal substantial regression in the memory subsystem, with memory latency metrics increasing above the 1.0 baseline. Specifically, *Memory Stall Cycles* increase by 3.8×, and *LLC Miss Latency* (measured in cycles) increases by 3.5×. GPU memory stall rates exhibit a similar 3.1× spike. These metrics directly quantify the back-pressure from the DRAM controller. This data confirms that despite the NPU's compute isolation, its massive DMA traffic saturates the shared Unified Memory Architecture (UMA), blocking CPU/GPU memory requests.

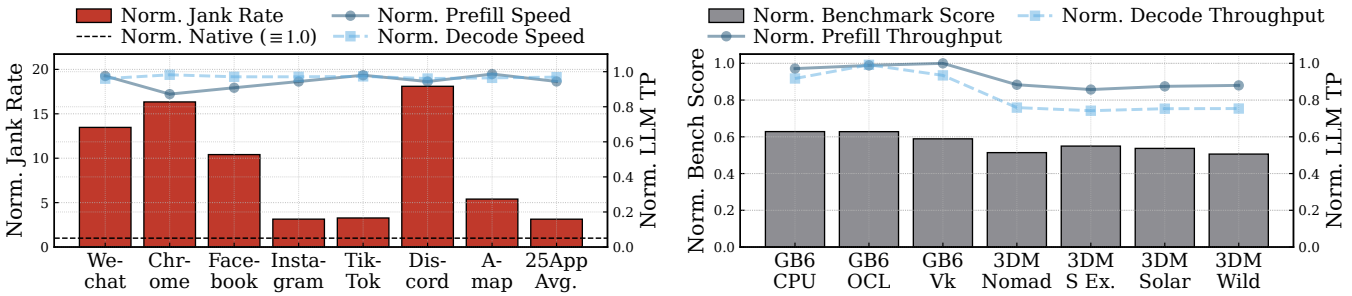


Figure 2: Interference when co-running LLM inference with each popular application. Background LLM inference severely degrades foreground QoS (e.g., jank) and peak system performance. Each application is evaluated individually with the background LLM, not concurrently with the other applications. The interference is asymmetric, as the LLM's own throughput is barely impacted.

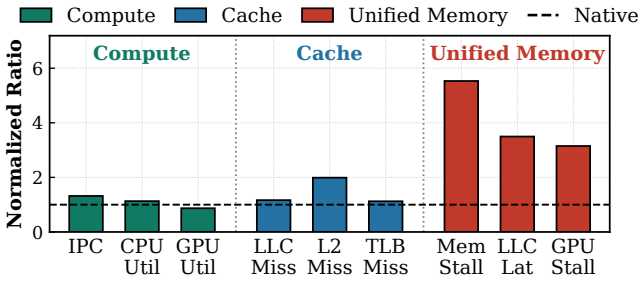


Figure 3: Root cause analysis: averaged metrics from co-running LLM inference with 25 applications, one at a time. Compute and **Cache** metrics remain flat, while **Unified Memory** contention metrics surge 3–4×, pinpointing bandwidth as the culprit. All values are normalized to *Native*, i.e., applications running alone averaged.

3.3 Architectural Root Cause Analysis

Why does the system fail to throttle aggressive NPU traffic? We analyzed the kernel and device tree sources for three flagship SoC families: Snapdragon 8 Elite [65, 75], Dimensity 9400 [57, 64], and Samsung Exynos 2400 [78]. Our analysis shows that the asymmetry arises from deep-rooted mechanism gaps and policy misalignments. As Table 1 shows through representative Snapdragon and Dimensity evidence, NPU workloads remain privileged compared to CPU/GPU workloads; the Exynos source tree exposes analogous vendor-specific BTS policy hooks for NPU traffic.

Missing NPU observability and control. Mobile OSes lack both observability and control handles for the NPU. On Snapdragon 8 Elite, governance frameworks like `memlat` [65] monitor CPU stalls but explicitly exclude NPU traffic, rendering it "invisible" to the OS scheduler. More critically, we find a hardware-level void in bandwidth partitioning. While the CPU cluster supports ARM's *Memory System Resource Partitioning and Monitoring* (MPAM) [12] to enforce QoS, the NPU driver stack lacks these hooks. This implies the NPU operates as an unmanaged bus master: the OS cannot enforce bandwidth quotas on it even if congestion is detected.

NPU-prioritized arbitration policies. Hardware and firmware arbitration structurally bias towards the NPU. Static analysis of the interconnect topology reveals that the NPU issues traffic through a dedicated on-chip lane (`nsp_noc` reaching `SLAVE_EBI1`) that bypasses the congestion of the shared multimedia fabric, and it is admitted into the top arbitration class—priority 0 with urgency forwarding (`prio` = 0, `urg_fwd` = 1), the same elite tier reserved for the latency-critical display—despite carrying merely best-effort LLM traffic. Crucially, this elevated service comes with no countervailing regulation: the CPU/GPU are reined in by MPAM bandwidth partitioning, `memlat` stall accounting, and thermal cooling maps, whereas the NPU is subject to none of the three. The few control loops that could otherwise intervene react far too slowly: while UI rendering demands strict latency (e.g., sub-8.3 ms for 120 Hz), the system's shared bandwidth DVFS governor escalates only after a coarse hysteresis window (∼64 ms), too slow to intercept bursty interference. Most starkly, thermal configurations explicitly exempt the NPU from throttling [64], letting it sustain aggressive bandwidth consumption even when the CPU is throttled. On Exynos 2400, the public kernel exposes Samsung's Bus Traffic Shaper (BTS) machinery and NPU-specific `npu_normal`/`npu_performance` scenarios, and the NPU device tree binds its MIF/INT hardware devices to BTS-controlled units [78]. This supports the same high-level conclusion: accelerator memory policy is configured through vendor-specific firmware/device-tree mechanisms rather than through portable per-application OS controls.

Rationale and workload mismatch. These privileges are not an oversight but a design choice. Consider 4K60 video recording: the image sensor streams massive raw data that the ISP and NPU must process within strict per-frame deadlines (<16 ms), so to prevent buffer overflows and dropped frames, architects established high-priority direct memory paths and disabled thermal throttling for the NPU. This is well matched to media-centric workloads but mismatched for persistent, best-effort AI: an on-device LLM is bandwidth-intensive (like video recording) yet semantically best-effort (like background downloading). Unable to distinguish the semantic intent of NPU instructions, the hardware grants the background LLM

Table 1: Hardware and OS mechanism gaps and policy misalignments. (✗: constrained / regulated; ✓: privileged / unregulated.) Foreground units (CPU/GPU) face strict visibility, latency, and thermal constraints, while NPU workloads remain privileged across SoCs [64, 65, 78].

Category	Dimension	Target SoC	Foreground (CPU/GPU)	Background (NPU)	Code Signature (Source)
Mechanism	**Visibility** *(Monitor)*	Snapdragon 8Elite	✗ **Sensed** (Stall cycles tracked by `memlat`)	✓ **Invisible (Blind Spot)** (No probe nodes in governor)	`qcom,cpulist` excludes NPU (*sun.dtsi*)
	Topology *(Bus Path)*	Snapdragon 8Elite	✗ **Shared Congestion** (Contends on shared `gem_noc`)	✓ **Dedicated High-Priority Path** (Top tier: `prio`=0, `urg_fwd`=1)	`nsp_noc` connects to `SLAVE_EBI1` (*sun.dtsi, sun.c*)
	Partitioning *(MPAM)*	Snapdragon 8Elite	✗ **Granular Task-Level** (cmax, cpbm, prio per PID)	✓ **No MPAM Support** (Not in MSC enum: L2/L3/SLC only)	`enum msc_qcom_id` lacks NPU type (*mpam_msc.h*)
Policy	**Regulation** *(Latency)*	Snapdragon 8Elite	✗ **Needs <8.3 ms** (120 Hz strict deadline)	✓ **BW DVFS @ ∼64 ms** (Too slow for UI bursts)	`zone1_thres_count = 16` (*icc-bwmon.c*)
	Thermal *(Throttling)*	Dimensity 9400	✗ **Aggressive Throttling** (Strict cooling maps applied)	✓ **Unrestricted** (Explicitly bypasses limits)	`THERMAL_APU_UNLIMIT` (*mtk_cooler_setting.h*)

the same high-priority treatment as a foreground camera, so it occupies memory paths designed for millisecond-critical media and delays latency-sensitive UI traffic.

Implication. Because image processing on the NPU is a genuinely critical media task where a single dropped frame is unacceptable, silicon vendors will likely continue to enforce this structural bias. Addressing the contention therefore necessitates a pure-software approach that regulates LLM bandwidth usage without relying on immutable hardware policies. Current vendor toolchains add further constraints: commercial NPU SDKs (e.g., Qualcomm QNN [74]) compile computation graphs into fixed static executables with predetermined batch sizes, and these opaque binaries expose neither per-operator memory behavior nor fine-grained traffic-shaping hooks—even inspection tools like Snapdragon Profiler [76] report only aggregate metrics. These limitations reinforce the need for a *software-level* approach that works within existing vendor ecosystems.

4 Challenges and Opportunities

To bridge this misalignment, our goal is to sense and regulate memory bandwidth, preserving foreground smoothness while the LLM runs in the background.

4.1 Key Challenges

Challenge 1: Lack of Low-Cost Sensing. Real-time visibility into the bandwidth contention is the first prerequisite for effective contention management. However, mobile SoCs typically lack accessible fine-grained hardware counters (e.g., MPAM or high-rate bandwidth PMUs) for the NPU. Software-only probes can infer DRAM latency, but they incur prohibitive CPU overhead and compete for the very bandwidth they seek to measure [23, 82, 90]. Without low-cost, high-frequency sensing, the system stays blind to the millisecond contention.

Challenge 2: Lack of High-Frequency Control Knobs. Effective mitigation requires intervening at a frequency that matches the interference dynamics. Android's SurfaceFlinger typically performs composition (the most bandwidth-intensive phase of rendering) in bursty windows, which we measured at $T_{comp} \approx 2.25$ ms on average. According to the Nyquist-Shannon sampling theorem [79], to effectively intercept these transients without aliasing, the control interval Δt_{ctrl} is strictly constrained by $\Delta t_{ctrl} \leq \frac{T_{comp}}{2} \approx 1.125$ ms. This theoretical limit conflicts with the realities of mobile NPUs. Specifically, modern NPU runtime such as QNN [72] executes computation graphs as monolithic units for maximal efficiency. Even with graph partitioning (i.e., chunking), the smallest runnable unit of an 8B model remains 5–8 ms ($> 4\times$ Δt_{ctrl}), making real-time control mathematically infeasible. Layer-level subdivision of the target model is therefore not sufficient: it creates control points only after long target-model subgraphs and any pause directly stalls committed decoding progress. Recent efforts [26, 34] to relax static-graph constraints offer finer control but at steep costs—reducing prefilling and decoding throughput by over 90% and 50%, respectively.

Challenge 3: Lack of Efficient Regulation. Even with real-time sensing of bandwidth contention and sufficient fine-grained control knobs, we still need a mechanism to yield bandwidth efficiently. Standard mechanisms like *Dynamic Voltage and Frequency Scaling* (DVFS) are ineffective because NPU energy efficiency degrades at low frequencies due to prolonged execution time and accumulated static power leakage [43, 94]. Alternatively, *chunk-and-sleep* strategies inject sleeps between LLM subgraphs, halting inference progress during the sleep interval and recovering foreground QoS only by sacrificing background progress. It is thus challenging to devise an elastic, soft-throttling mechanism that can reduce bandwidth pressure while maintaining forward computation progress.

4.2 Repurposing Speculative Decoding

Of particular interest, we find that *speculative decoding* offers a unique structural alignment with our requirements for solving the challenges described above. Speculative decoding structurally consists of a fine-grained, memory-bound *draft* phase and a heavyweight, compute-dense *verify* phase. The draft phase comprises a sequence of forward passes on a smaller model that naturally break the original LLM execution into smaller windows, thereby providing the missing

fine-grained knobs through draft interruption. The compute-intensive verification phase, in turn, introduces a complementary lower-bandwidth stage, offering opportunities for bandwidth regulation across phases. This is the key distinction from subdividing the target model itself: yielding during the draft phase discards only tentative work, while the target model still verifies already available candidates and preserves committed decoding state.

Based on the above analysis, we derive a key insight: rather than leveraging speculation solely for speed, we can **repurpose** it to serve as a vehicle for memory bandwidth regulation.

Throughput-oriented speculation under contention. However, translating this structural potential into a QoS mechanism is non-trivial because existing frameworks are designed to maximize throughput. Frameworks like Medusa [16] or EAGLE-3 [50] aggressively saturate memory bandwidth to maximize acceptance rates. Our profiling with EAGLE-3 reveals that 94.7% of the decoding time is bandwidth-bound.

Importantly, these algorithms are fragile when operating under constraints. Specifically, our profiling of EAGLE-3 reveals that a naive attempt to yield bandwidth—by enforcing a 50% bandwidth cap during the verification phase—causes decoding throughput to drop by 54% relative to the peak. Moreover, this throttled performance drops 8.4% below the non-speculative baseline. This *negative gain* phenomenon indicates that, without a specialized mechanism, simply throttling speculation can eliminate its benefits entirely.

5 SERENO Design

To enable the seamless coexistence of always-on background intelligence and interactive foreground experiences, we propose SERENO, a pure-software inference framework that resolves foreground–LLM bandwidth contention via a sub-millisecond *Sense-Decide-Act* loop that dynamically regulates the LLM's NPU bandwidth usage.

Design goals. SERENO pursues two objectives to resolve asymmetric foreground performance degradation. ❶ *Foreground Protection:* SERENO should restore foreground smoothness to near-native levels by strictly limiting memory bandwidth during interactive bursts; ❷ *Efficient Yielding:* SERENO should maximize background LLM throughput by avoiding zero-progress yields (e.g., long sleeping) that waste resources.

Design constraints. Despite the challenges we introduce in §4.1, SERENO is designed to be deployable on commodity smartphones, imposing strict boundaries: ❶ *Immutable Hardware:* We cannot modify SoC interconnect priorities, requiring software-only solutions; ❷ *User-Space Constraint:* To ensure compatibility, we avoid relying on vendor-specific kernel or driver modifications; ❸ *Black-Box NPU:* We must accommodate the opaque nature of commercial NPUs, where computation graphs are static and their execution is atomic. Meanwhile, the batch size of each graph must be fixed at export time, which is required by current NPU runtime [70].

System workflow. The control loop, illustrated in Figure 4, begins with the *Contention Sensor* (§5.2), which derives a real-time *Contention Score* from the execution latency of draft subgraphs. This score feeds into the *Controller* (§5.3), which computes the necessary yielding intensity. These decisions drive the *Elastic Speculative Decoding* engine (§5.1): during the draft phase, the controller may trigger an instant *Draft Preemption* to halt the memory bandwidth usage, employing *N-gram Filling* to compensate for incomplete speculative tokens. Then, *Selective Batching* chooses the execution graph with the appropriate batch size from the pre-exported graphs. Finally, the workflow transitions to the verification phase, where *micro-sleeps* might be injected to regulate the remaining bandwidth demand before the next cycle begins.

5.1 Elastic Speculative Decoding

To address the lack of fine-grained control knobs, SERENO introduces *Elastic Speculative Decoding*. We transform the monolithic inference process into a sequence of interruptible, fine-grained units, enabling the system to take actions at sub-millisecond timescales without sacrificing model correctness.

Preemptible draft execution. We leverage the inherent discardability of the speculative draft phase to create high-frequency intervention points. Unlike conventional designs that execute the draft model as a single atomic phase, we refactor it into a chain of fine-grained units.

Using the vendor SDK (e.g., QNN [72]), we compile the draft model at the granularity of individual Transformer layers. We use *subgraph* to denote one statically compiled NPU executable containing a contiguous set of operators, such as one draft Transformer layer or one verification-layer group. For the draft model, this creates execution subgraphs with sub-millisecond latency. At any subgraph boundary, SERENO can trigger an instant preemption: discarding the remaining draft steps and immediately transitioning to the verification phase. This converts the NPU from a run-to-completion black box into a preemptible task that can release memory bandwidth almost instantly upon foreground contention.

Crucially, we retain the static graph: unlike approaches that fall back to inefficient dynamic operators for flexibility [34], our method maintains near-native NPU execution efficiency.

N-gram candidate filling. Draft preemptions inevitably reduce the number of speculated tokens, potentially harming decoding throughput. To mitigate this penalty, SERENO employs a low-cost *N-gram Filling* mechanism to replenish the empty slots in the verification batch. When the neural draft produces fewer candidates than the selected verification batch can hold, SERENO looks up the latest committed token sequence in a local n-gram cache and appends the most likely continuation tokens as additional tentative candidates. These tokens are verified by the target model in the same way as neural draft tokens, so incorrect n-gram candidates are re-

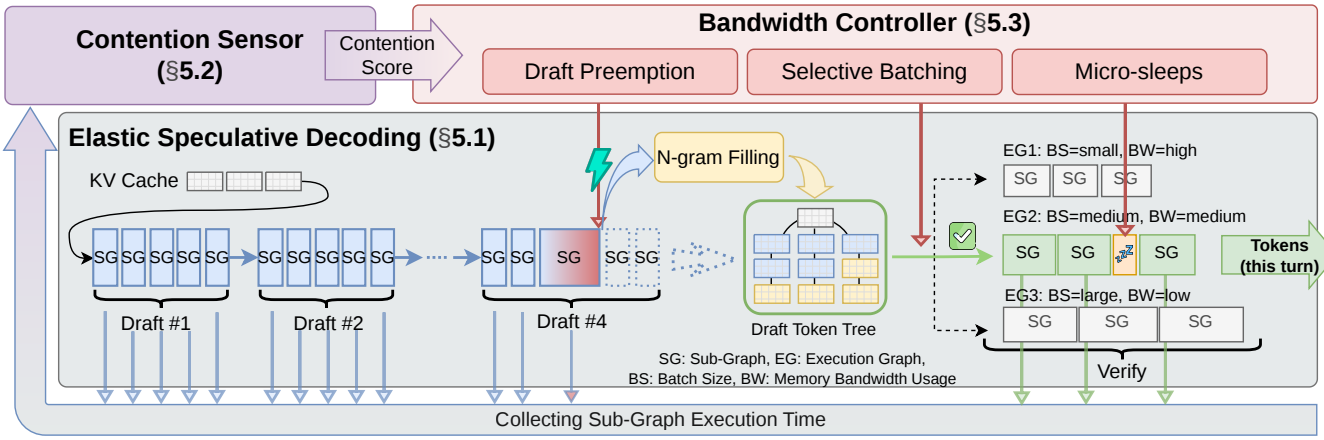


Figure 4: Design overview of SERENO. A Sense-Decide-Act loop to dynamically regulate LLM inference's NPU bandwidth usage.

jected without changing output semantics. We introduce three optimizations to standard N-gram speculation to maximize acceptance rates using only local context: ❶ *Multi-source Construction:* We build the N-gram cache by integrating tokens from three weighted sources: verify outputs (high confidence), prompts (medium), and crucially, rejected draft sequences (low). This recycles valuable local context that is originally discarded. ❷ *Self-Purifying Cache:* We filter candidates via hit-frequency tracking, automatically evicting low-quality sequences to suppress noise. ❸ *Adaptive Length Weighting:* We employ mixed x-gram lengths (e.g., 2-gram, 3-gram, ...), assigning higher weights to longer matches (capturing semantics) and lower weights to short collocations. This prevents noise from short-context matches. These enhancements let SERENO recover speculation candidates even when the neural draft phase is preempted early.

Selective batching. While the draft phase provides preemptible knobs, the verification phase offers various bandwidth control mechanisms. We pre-compile multiple verification graphs with varying batch sizes (e.g., $B = 8, 16, 32$) that share weights but differ in parallelism, resulting in varying bandwidth usage during verification.

During verification, SERENO dynamically selects the appropriate graph based on the contention sensor's output. We characterize each verification graph by its absolute average memory-bandwidth demand while it runs. This is a graph-level bandwidth pressure metric, not a per-candidate cost: normalizing the measured bandwidth of the $B = 1$ verification graph to 1.00, the demand falls to 0.82/0.79/0.74 for $B = 8/16/32$ and further to 0.61/0.38 for $B = 64/128$. Larger verification batches expose more parallel candidate work for roughly one target-model weight traversal, lowering the graph's average bandwidth demand, but they also form longer atomic verification bursts. This does not imply universally higher end-to-end throughput: larger verification batches require more candidate tokens, which may force additional draft work or lower-confidence n-gram candidates before verification can run. The resulting spread in bandwidth levels gives SERENO scheduling room: under heavy memory contention, SERENO switches to lower-bandwidth larger-batch verification graphs and relies on micro-sleeps between verification subgraphs to break up the burst. Under low contention, it prefers smaller graphs to minimize draft waiting and preserve decoding latency [91].

5.2 Endogenous Contention Sensing

Sensing rationale. To sense bandwidth contention, DRAM latency is the primary indicator [44, 45]. As highlighted in §4.1, conventional methods (e.g., pointer chasing) that measure DRAM latency incur prohibitive overhead. However, we observe that the memory-intensive LLM decoding step mimics these latency probes: it generates sequential DMA transfers between NPU SRAM and DRAM that inherently bypass host caches. Crucially, while synthetic probes require multiple rounds to resolve noise, the sheer volume of data processed in a single LLM step ensures high signal robustness against transient system noise. Therefore, instead of introducing external monitors, we leverage the latency of LLM workload itself as an endogenous, zero-cost contention sensor.

Based on this insight, SERENO repurposes the execution latency of fine-grained draft phase subgraphs as a real-time contention proxy. This signal is fully endogenous and nearly zero-overhead: it utilizes timing data captured during the decoding, eliminating the need for hardware counters or intrusive micro-benchmarks. When foreground applications contend for the memory bus, the resulting memory stalls manifest directly as latency spikes in subgraph execution.

Reactive sensing rather than prediction. SERENO uses reactive sensing because foreground behavior is dominated by user input, app-specific rendering paths, and OS composition timing. Predicting these bursts would require app-specific models and would either miss rare interactions or over-throttle inference conservatively. In contrast, the draft-subgraph signal is available at sub-millisecond granularity with no extra memory traffic, so the controller can react within the same timescale as rendering bursts while smoothing noise through

hysteresis and PI control.

Calibration. Specifically, we observe that static graph fusion strategies cause significant latency variance among Transformer blocks (e.g., up to 37% difference in Llama-3.2-1B). This is likely due to non-uniform compiler optimizations, such as different memory alignments and tensor tiling strategies assigned to specific graph nodes. Crucially, despite this spatial variance, the execution latency of any specific subgraph is temporally stable across iterations due to the deterministic nature of NPU static graphs. Therefore, SERENO performs a one-time offline calibration to profile the baseline distribution for *each* unique subgraph in a contention-free environment.

Runtime contention score. At runtime, we compute a normalized *Contention Score* per executed draft subgraph: $CS = (T_{actual} - T_{baseline})/T_{baseline}$, where T represents subgraph execution latency. As demonstrated in §7.3, this software-defined metric exhibits a strong correlation ($R \approx 0.86$) with physical *Memory Stall Cycles per LLC Miss*. This confirms that *CS* serves as a high-fidelity proxy for physical bandwidth contention.

5.3 Bandwidth Controller

To orchestrate the sensing signals and elastic knobs, SERENO implements a closed-loop control system comprising a hierarchical actuation mechanism and a feedback-driven controller. This design minimizes throughput loss while responding to sub-millisecond contention transients.

Actuation primitives. We define three actuation primitives, ordered by their aggressiveness in yielding bandwidth. Level ❶: *Draft Preemption* (instant relief). Since the draft phase executes with Batch=1 (high memory intensity), it saturates bandwidth most aggressively. Preempting the draft phase immediately releases the bus and stops occupying the bandwidth. Level ❷: *Selective Batching* (coarse adjustment). Upon preempting to the verification phase, the controller selects between precompiled graphs (e.g., Batch=16 vs. 32). It prefers smaller batches for low-latency progress under light contention, but switches to larger batches when moderate yielding should lower the verification graph's bandwidth demand. Level ❸: *Micro-Sleeps* (fine-tuning). While batch selection offers discrete control steps, micro-sleeps allow for continuous modulation of the bandwidth duty cycle during the verification phase. The controller injects precise sleep intervals only between verification subgraphs, not inside a running NPU graph. Because each target-model verification subgraph lasts 5–8 ms, a 1–2 ms sleep between subgraphs reduces the average verification bandwidth duty cycle by roughly 10–25%. This primitive complements draft preemption: preemption handles immediate relief during the high-bandwidth draft phase, while micro-sleeps fine-tune residual verification-phase pressure.

Controller. To dynamically orchestrate the actuation primitives, SERENO employs a standard *Proportional-Integral (PI) controller* [36], a classic feedback mechanism that modulates control inputs to minimize the error between a measured state and a desired setpoint. The controller computes the error term $e(t) = CS_{measured} - CS_{target}$, where $CS_{measured}$ is the real-time contention score and CS_{target} is the maximum allowable contention. Formally, the control output $u(t)$ is derived as $u(t) = K_p e(t) + K_i \int_0^t e(\tau) d\tau$. The proportional term (K_p) drives immediate reaction to the current error magnitude, while the integral term (K_i) accumulates historical errors to eliminate steady-state offset [13, 40, 41].

We specifically exclude the *derivative* term due to the stochastic nature of mobile system signals: the execution latency of LLM subgraphs contains minor noise from scheduling jitters and transient DRAM contention, which a derivative term would amplify into erratic control outputs and oscillation.

The controller's output $u(t)$ represents the aggregated *yielding intensity*. We employ a phase-aware mapping strategy to translate this continuous signal into discrete primitives: In the draft phase, we disable micro-sleeps to maximize speculative efficiency when the bus is free. Given the high bandwidth intensity of drafting, we enforce an aggressive preemption threshold: even a low magnitude $u(t)$ triggers immediate draft preemption. This ensures the system yields bandwidth instantaneously at the first sign of contention rather than stalling on speculative work. In the verification phase, $u(t)$ drives graded regulation to maintain progress. The signal is first mapped to selective batching (choosing larger batches for high $u(t)$), and then linearly mapped to micro-sleep durations inserted between subgraphs for fine-grained throttling.

To prevent oscillation from transient noise, we further apply a hysteresis filter to the draft preemption decision, requiring consecutive violation samples before triggering.

Minimum throughput guardrail. To prevent starvation, SERENO incorporates a *Token Bucket* mechanism that overrides the PI controller. The bucket fills at the instant generation rate and drains at a configurable baseline rate. This creates a three-stage gradation: ❶ *Nominal Enforcement:* As long as the bucket contains tokens over the lower bound (e.g., 10% of the maximum capacity), the system runs as normal. ❷ *Adaptive Relaxation:* Upon depletion below the lower bound, the system automatically relaxes the CS_{target} (raising the contention threshold), reducing yielding frequency to help the background task recover throughput. ❸ *Fail-Safe:* Under persistent deficit (i.e., if the bucket remains empty for a sustained duration), SERENO temporarily disables QoS interventions, falling back to standard speculative decoding to accelerate.

While these relaxations theoretically reintroduce some interference, it is a necessary compromise to ensure service continuity. Crucially, our evaluation (§7.2) demonstrates that even with this mechanism active in resource-constrained scenarios like *Heavy Gaming*, SERENO still delivers substantial QoS optimization compared to the uncontrolled baseline.

6 Implementation

Environment & framework. We implemented SERENO on Snapdragon chipsets using the Qualcomm QNN SDK (ver. 2.39) [74], atop the open-source PowerServe [66].

Core modules. SERENO adds ~6.4k lines of C++ code. ❶ *Elastic Speculative Decoding* (~2.8k LoC) manages the interruptible substrate. It carefully manages in-flight states and KV caches during preemptions without polluting committed state, guaranteeing consistency. ❷ *Bandwidth Controller* (~2.7k LoC) refactors the decoding loop for control and contains a PI controller to dynamically orchestrate the actuation primitives. ❸ *Contention Sensor* (~0.9k LoC) injects timer hooks directly into operator paths to derive the real-time contention score.

Automated tuning. For portability, we provide a Ziegler-Nichols [108] calibration script that sweeps the proportional gain under synthetic loads to find the *ultimate gain* (K_u)—the point where the contention signal sustains oscillation [14, 15, 33]—then derives stable coefficients (K_p, K_i) tailored to the SoC from K_u and the oscillation period.

Policy instantiation. We instantiate two policies to address distinct QoS requirements: ❶ *Policy A: Balanced Mode* is our default policy targeting daily multitasking. It sets a moderate contention threshold ($CS_{target} = 0.15$, permitting 15% latency hike) and enables the throughput guardrail (12 tokens/s for filling the bucket) to prevent starvation, balancing the smoothness and throughput. ❷ *Policy B: UI-First Mode* is a strict policy targeting competitive gaming; it enforces a tight threshold ($CS_{target} = 0.07$) and disables the throughput guardrail, prioritizing zero foreground stalls over LLM inference speed.

7 Evaluation

We evaluate SERENO by addressing four questions: ❶ **End-to-End Effectiveness:** How does SERENO perform in terms of foreground QoS and background LLM inference throughput compared to industry SOTA solutions and naive strategies? ❷ **Mechanism Analysis:** Does SERENO mitigate hardware bottlenecks? Do its software-aware signals and scheduling decisions function as intended? ❸ **Ablation Study:** What are the contributions of individual components? ❹ **System Analysis:** What are SERENO's overheads and robustness?

7.1 Experimental Setup

Hardware & Environment. Our primary testbed is the *OnePlus 13* (Snapdragon 8 Elite), alongside a *OnePlus 12* (8 Gen 3) for generality. Both feature a heterogeneous architecture where the CPU, GPU, and Hexagon NPU share 24 GB LPDDR5X unified memory. We measure full-system power with a power meter [17] and ensure no uncontrolled DVFS downclocking during experiments.

Workloads. We deploy *Llama-3.1-8B-Instruct* (W4A16). We use 15 diverse prompts from the GSM8K dataset [21] to cover varying prefill/decode complexities. In §7.5, we further discuss the generalization to other models.

Based on 25 popular applications in §3.1, we add 5 heavy games into the list and categorize these 30 applications by resource intensity from light to heavy: ❶ *Tools* (e.g., Chrome): bursty CPU loads; ❷ *Social* (e.g., WeChat): rich media list scrolling, with mixed CPU/GPU loads; ❸ *Media* (e.g., YouTube): multimedia decoding, with steady CPU/GPU loads; ❹ *Gaming* (e.g., Genshin Impact): sustained CPU/GPU loads and memory pressure.

Automation scripts replay fixed operation sequences: non-game apps simulate scrolls or clicks at ~1 Hz; game loads run complex control loops (e.g., camera rotation, jumps) at ~2 Hz. Each experiment runs one foreground application at a time with the background LLM; the 30 applications form a benchmark suite rather than a concurrent workload.

Protocol. We report the mean of 15 runs (~45 s each, one per prompt), excluding the two largest and smallest values. Continuous decoding serves as the default stress test to expose worst-case overlap with foreground rendering; we further evaluate a sporadic invocation pattern in §7.2 to capture assistant-style usage where bursts are short but still coincide with rendering deadlines.

Baselines. We compare SERENO against three categories of baselines. ❶ **Native:** Foreground only (i.e., the upper bound). ❷ **SOTA Frameworks:** *PowerServe* [66] (QNN-based), *llama.cpp* [27] (CPU/GPU(opencl)/NPU backends), and *MNN* [55]. ❸ **Strategies:** Implemented within *PowerServe* to represent standard approaches, including *Freq-Limit* (static NPU frequency capping to ~50% frequency), *Chunk-Sleep* (fixed 50% duty cycle sleep), and *Speculative* (throughput-oriented speculative decoding schemes like EAGLE-2 [48]). For SERENO and *Speculative*, we use *Llama-3.2-1B-Instruct* (W4A16) as the draft model.

Metrics. For foreground QoS, we extract *jank rate* and *slow rendering rate* from Perfetto FrameTimeline [87]. To evaluate system peak performance, we use Geekbench 6 and 3DMark scores [68, 88]. For background performance, we measure throughput (tokens/s) and latency in prefill and decode phases. Micro-architectural metrics are collected via Snapdragon Profiler [76] and simpleperf [9].

7.2 End-to-End Effectiveness

This section evaluates SERENO in multitasking scenarios, assessing whether it preserves foreground QoS and peak system performance while maintaining competitive background LLM throughput. Unless otherwise stated, we utilize the default *Balanced Policy* (§6).

Comparison with SOTA frameworks. We first compare SERENO against SOTA frameworks (Figure 5 top row). Uncontrolled NPU inference (PowerServe) causes substantial foreground QoS degradation: taking *Social* workloads as an example, the jank rate increases to 22.12% (vs. Native 6.18%, top left panel) while the slow rendering rate also increases to 3.7× the Native baseline (1.08% vs. 0.29%, top

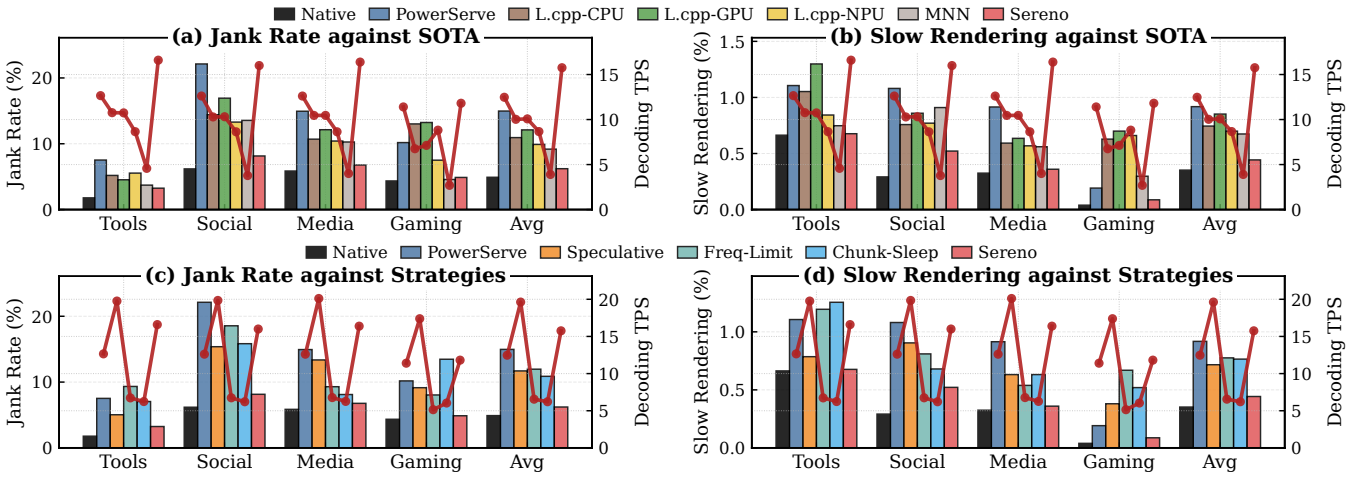


Figure 5: End-to-end effectiveness. SERENO restores foreground QoS to near-native levels while maintaining competitive throughput against SOTA frameworks (a,b) and naive strategies (c,d). Each foreground application is evaluated individually with the background LLM. L.cpp stands for llama.cpp.

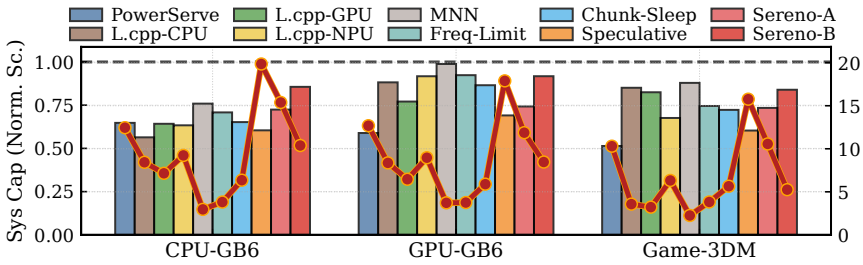


Figure 6: System peak performance recovery. Suffix -A and -B represent two policies in §6.

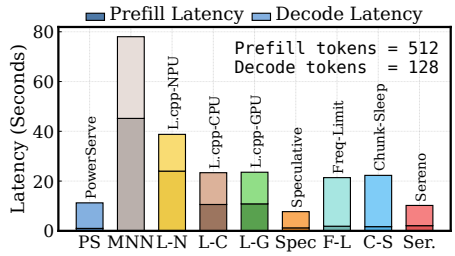


Figure 7: Latency breakdown.

right panel). Other frameworks also fail to resolve this contention: *llama.cpp-GPU* suffers high interference (16.92% jank rate) due to shared memory bottlenecks. *MNN*'s jank rate is 13.55%, and its throughput is severely reduced to 3.78 tokens/s. SERENO substantially mitigates this interference, suppressing the *Social* jank rate to 8.14% (approaching Native's 6.18%)—eliminating most induced stutter—while sustaining 16.00 tokens/s. SERENO delivers up to 92.6% reduction in jank rate and 67.9% increase in throughput in comparison with PowerServe. On average across all categories, SERENO achieves a jank rate of 6.21% (Native is 4.91%) and a slow rendering rate of 0.44% (Native is 0.35%), effectively restoring near-native fluidity. Compared to PowerServe, SERENO reduces the jank rate by 58.5% and improves throughput by 26.4%. Compared to *llama.cpp-NPU*, SERENO reduces the jank rate by 37.1% and improves throughput by 82.2%.

Comparison with scheduling strategies. The scheduling-strategy comparison (Figure 5 bottom row), against aggressive (*Speculative*) and conservative (*Chunk-Sleep*, *Freq-Limit*) baselines, highlights the necessity of SERENO's elastic design. While speculative decoding (*speculative* in the figure) maximizes throughput (19.84 tokens/s), its unregulated bandwidth usage keeps the *Social* jank rate high at 15.38% (far above the Native's 6.18%). SERENO trades a moderate throughput (16.00 tokens/s) to bring the jank rate down to 8.14%, significantly approaching the Native baseline. Under *Reader* application, SERENO achieves a 72.1% jank rate reduction with only a 6.2% throughput degradation. Regarding conservative strategies, *Freq-Limit* proves to be ineffective: by capping NPU frequency to ~50%, it suffers from both low throughput (6.74 tokens/s) and high interference (18.55% jank rate), as it lacks the responsiveness to handle micro-bursts. *Chunk-Sleep* behaves similarly, and the trend holds across categories. Overall, SERENO achieves 2.5× the throughput of these naive strategies while maintaining lower average slow rendering (0.44%), showing that SERENO's sub-millisecond elasticity improves the QoS-throughput balance.

System peak performance. Figure 6 evaluates the recovery of system peak performance using both SERENO policies (the -A and -B suffixes denote the two policies in §6). In *Gaming* scenarios, uncontrolled inference degrades scores to 51% of Native levels. For SERENO, balanced policy recovers this to 73% while maintaining high throughput (10.56 tokens/s), offering a middle ground better than uncontrolled execution. For maximum foreground protection, strict policy further prioritizes other tasks, restoring scores to 84% and GPU scores to 92% (vs. PowerServe's 59%). A critical comparison arises with MNN: while MNN achieves a high benchmark score

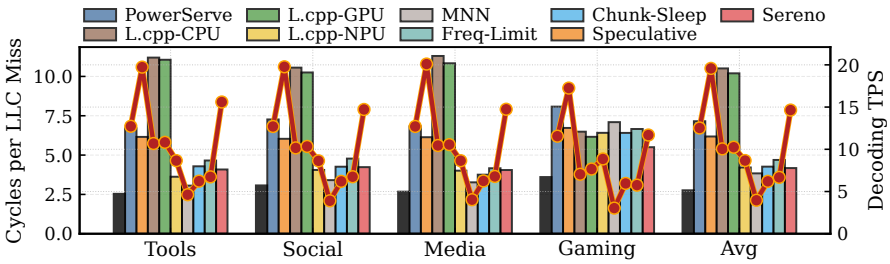


Figure 8: Physical memory stall reduction.

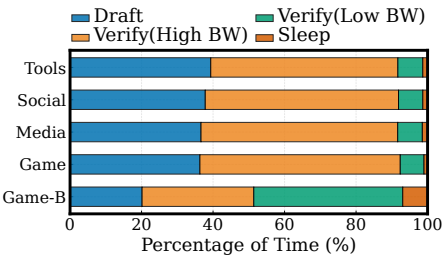


Figure 9: Dynamic mode adaptation.

(88%), it does so by throttling inference to a negligible 2.25 tokens/s. In contrast, SERENO (strict policy) maintains 5.25 tokens/s in the same scenario, indicating true resource elasticity rather than simple pausing.

Latency. Finally, Figure 7 analyzes the request turnaround time (512 prefilling tokens + 128 decoding tokens), averaged over the 30-application benchmark suite with one foreground app running at a time. SERENO's end-to-end latency (10.21 s) is 9% lower than the uncontrolled PowerServe baseline (11.25 s). This result stems from the underlying speculative acceleration: although the prefill phase is slower (2.09 s vs. 0.99 s) due to the smaller batch sizes required for preemptibility, the accelerated decode phase compensates for this delay. Among all systems, SERENO exhibits only slightly higher latency than *Speculative*, as the latter ignores other tasks to solely focus on inference acceleration.

Sporadic workload evaluation. We model SERENO under a sporadic invocation pattern using OnePlus 13. To simulate realistic intermittent usage scenarios, we sweep four invocation schedules, covering short intermittent requests such as notification summarization, smart reply, and short translation [11, 28, 77]. Each schedule holds the invocation period fixed at 50 s and varies only the decode-burst fraction, giving 10/40 s, 20/30 s, 30/20 s, and 40/10 s (burst/idle) schedules, i.e., duty cycles of 20%, 40%, 60%, and 80%. The lightest 10 s-burst/40 s-idle point reflects a single short assistant turn (e.g., a notification summary) per user think-time interval; sweeping toward the 40 s-burst/10 s-idle point progressively stresses the system with denser back-to-back invocations. For the foreground, we deliberately use lightweight, latency-sensitive applications from the categories that realistically co-occur with sporadic assistant turns—*Tools*, *Social*, and *Media* (5 applications in total)—rather than heavy workloads such as gaming. This matches the common case for short background invocations and is also a conservative setting: because such apps already render near the smoothness ceiling, the foreground has little slack to absorb interference, so any residual stutter is directly attributable to the background LLM rather than to foreground load itself. We construct 5 min traces covering six bursts for each system–application pair. Effective throughput is reported as the total committed tokens divided by wall-clock time, including idle intervals; therefore, it captures both useful decoding progress and the idle gaps imposed by sporadic invocation.

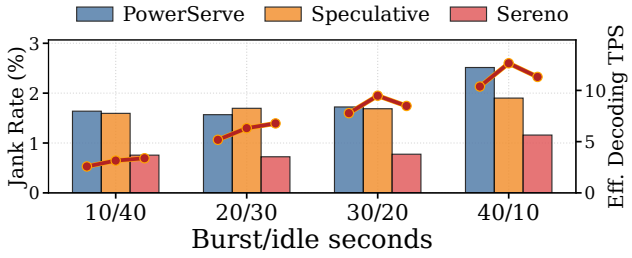


Figure 10: Sporadic workload sensitivity. We vary the decode-burst/idle schedule while keeping the trace length fixed. Effective decoding TPS includes idle intervals in the denominator.

Figure 10 reports the resulting sensitivity sweep. As the scheduled duty cycle increases, effective decoding throughput rises almost linearly for all systems because the LLM is active for a larger fraction of wall-clock time. At the same time, foreground interference remains visible: PowerServe's jank rate rises from 1.64% to 2.51%, and throughput-oriented speculative decoding stays between 1.59% and 1.90% jank across the sweep. SERENO consistently provides the lowest jank rate, ranging from 0.72% to 1.16%, roughly halving PowerServe's jank at each duty cycle. Absolute jank rates here are well below the continuous stress test (Figure 5), as expected from the lighter foreground load and reduced duty cycle; the meaningful signal is thus the consistent relative gap between systems, not the absolute floor. The throughput tradeoff is also bounded. SERENO achieves 3.38 and 6.78 effective tokens/s at 20% and 40% duty cycles, respectively, and remains ahead of PowerServe at the heavier 60% and 80% settings (8.48 and 11.32 effective tokens/s). Thus, increasing the burst frequency stresses all systems, but the relative conclusion is unchanged: intermittent LLM invocations still perturb foreground rendering when they overlap with user interaction, and SERENO's fine-grained yielding preserves a better QoS–throughput balance under both light and heavy sporadic load.

7.3 Mechanism Analysis

We dissect SERENO's internal mechanisms to validate three design principles: reducing memory contention, ensuring signal fidelity, and achieving dynamic elasticity.

Physical bottleneck mitigation. Figure 8 validates that SERENO addresses the root cause: shared memory contention. We quantify this using *Stall Cycles Per LLC Miss (CPLM)*—derived as the ratio of memory stall cycles to LLC misses (calculated as PMU counter `CYCLE_BACKEND_MEM_STALL` / `LLC_MISS`). CPLM serves as a high-fidelity proxy for the effective DRAM latency perceived by the CPU.

Under uncontrolled NPU inference (PowerServe), the average CPLM surges to 7.15 cycles, 2.6× the Native baseline (2.76 cycles), confirming that NPU DMA activity saturates the interconnect. SERENO effectively regulates this traffic, pulling the average CPLM down to 4.17 cycles—a 42% reduction compared to the uncontrolled baseline. Even in the most bandwidth-starved *Gaming* scenario, SERENO outperforms throttling strategies like *Chunk-Sleep* (6.41 cycles), reducing CPLM by a further 14%. This indicates that SERENO's high-frequency knobs and fine-grained yielding are significantly more effective at clearing memory bus congestion than coarse-grained sleep or unoptimized execution.

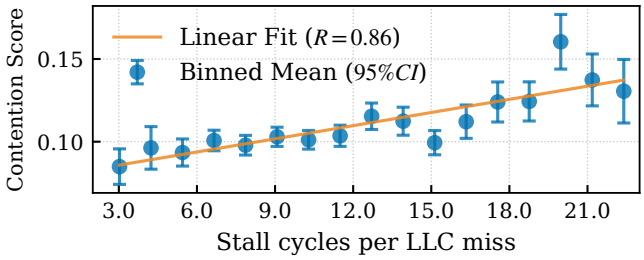


Figure 11: Signal correlation. CI stands for confidence interval.

Signal accuracy. The efficacy of our control loop hinges on the accuracy of SERENO's *Contention Score*. Figure 11 correlates our zero-overhead signal (draft subgraph latency) against CPLM. Across 125,060 samples ranging from light social scrolling to heavy gaming, we observe a strong linear correlation when CPLM ≤ 22 (Pearson $R \approx 0.86$). This correlation allows SERENO to sense congestion at sub-millisecond granularity without software overhead or hardware performance counters.

Controller actuation behavior. Figure 9 shows the dynamic distribution of execution modes. Under the default balanced policy, the distribution remains stable across standard workloads, with *Draft* occupying about 35–40% of the total time. In *Gaming* scenarios, the reduction in *Draft* (to 36.3%) is moderated by the *Minimum Throughput Guarantee*, which limits yielding to preserve baseline inference performance. To validate the mechanism's full adaptive range, we analyze the strict policy (Gaming-B) where this throughput constraint is disabled. Under this configuration, the system exhibits a substantial mode shift: the *Draft* time share drops to 20.1%, while the low-bandwidth, larger-batch verification mode rises to 41.7% (vs. 6.6% in default). This confirms that the controller effectively reconfigures the inference pipeline to reduce memory bus occupancy when policy constraints permit.

Controller effectiveness. Taken together—a stable input signal ($R = 0.86$), the 42% CPLM reduction, and a policy-driven Gaming mode shift that also raises micro-sleep time from 1.0% to 6.9%—these results show that the PI controller is not merely enabling static throttling: it maps measured contention into graded, phase-specific actions, all without a separate hardware mechanism.

Speculative acceptance and rejection. We also inspect the speculative statistics recorded by SERENO's runtime. Across SERENO runs in the continuous stress test with instrumentation, the tentative-token acceptance ratio is 18.9% on average, corresponding to an 81.1% rejection ratio. This token-level ratio is lower than throughput-oriented speculative decoding (30.4% acceptance in the instrumented baseline runs), and we emphasize that this gap is a deliberate design choice of SERENO rather than an unintended side effect. To minimize memory bandwidth occupancy under contention, SERENO intentionally (i) verifies in larger batches that amortize target-model weight traffic over more candidates, and (ii) adopts a more aggressive draft-abort policy that preempts the draft model whenever foreground bandwidth pressure rises. Both decisions trade per-token acceptance for bandwidth headroom; the resulting throughput loss is then compensated by *N-gram Filling*, which recycles high-confidence candidates to keep useful work flowing through the larger verification window. Correctness is unaffected because every candidate, whether produced by the draft model or the n-gram cache, is verified by the target model before commitment. Importantly, the higher rejection rate does not blow up the energy budget: as reported in Figure 13, SERENO reaches 3.52 tokens/J versus 3.78 tokens/J for throughput-oriented speculative decoding, a modest ∼7% efficiency gap. The marginal energy of a rejected candidate is small—the draft model is lightweight and verification is batched, so the extra candidates ride along amortized target-model weight traffic rather than adding proportional compute—and SERENO's race-to-sleep behavior recovers static-power leakage by completing useful work at high-performance operating points before yielding. We therefore attribute the bulk of the efficiency gap to control-plane overhead and forced yielding rather than to wasted draft work.

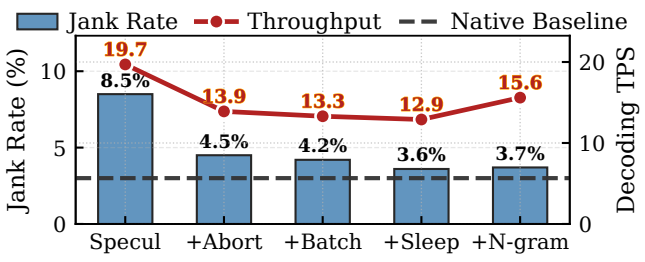


Figure 12: Ablation with dual axes: QoS and throughput.

7.4 Ablation Study

Component ablation. We quantify each component's contribution by enabling them one at a time on four representative apps (Chrome, WeChat, YouTube, and Genshin) covering all categories, as shown in Figure 12.

The baseline, i.e., standard speculative decoding, achieves the highest throughput (19.7 tokens/s) but suffers from a high jank rate of 8.5% (Native is 3.0%) due to unregulated memory bursts. Introducing *Draft Preemption* is the largest single intervention because it targets the bandwidth-dominant draft phase: it reduces the jank rate by 47% (to 4.5%). This confirms that sub-millisecond preemptibility is the primary prerequisite for responsiveness. The remaining components address different phases rather than competing with draft preemption. *Selective Batching* regulates verification by selecting lower-bandwidth verification graphs at a throughput cost, reducing jank from 4.5% to 4.2% while throughput changes from 13.9 to 13.3 tokens/s. *Micro-Sleeps* further handle residual verification bursts, lowering jank to 3.6%. However, this rigorous protection comes at a cost, reducing decoding throughput to 12.9 tokens/s. Crucially, *N-gram Filling* is not a contention-control primitive; it compensates for the throughput loss caused by preemption, recovering throughput to 15.6 tokens/s (+20.9% over the preemption+batch+sleep configuration) without compromising QoS.

This ablation demonstrates that SERENO's components are non-redundant and complementary: the control plane (Draft Preemption/Selective Batching/Micro-Sleeps, §5.3) establishes the safety boundary for foreground tasks, while the algorithmic plane (N-gram Filling, §5.1) maximizes the utilization of the remaining bandwidth budget.

N-gram filling ablation. To isolate the impact of our *N-gram Filling* enhancements, we follow the setup in §7.4 but operate in a contention-free environment. We restrict the neural draft model to 12 tokens and use N-gram Filling to pad the verification batch up to 32 tokens. This forces prediction on distal tokens, where model confidence is typically lower. Under this constraint, a naive baseline yields an acceptance rate of 0.64% and a throughput of 14.42 tokens/s. Integrating *Multi-source Construction* provides the first major improvement—increasing throughput to 16.40 tokens/s (+13.7%)—which confirms the value of recycled drafts. Subsequently, applying *Self-Purifying* and *Adaptive Length Weighting* further improves the acceptance rate to 2.59% and lifts the final throughput to 18.06 tokens/s. Collectively, these optimizations recover 25.2% of the decoding speed lost due to preemptions.

7.5 System Analysis

We evaluate the practical deployability of SERENO by analyzing its energy efficiency, hardware generality, task robustness, and resource overhead.

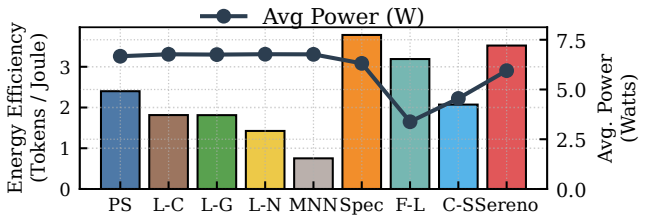


Figure 13: Energy efficiency. Abbreviations follow Figure 7.

Energy efficiency. Mobile inference is strictly energy-constrained. Figure 13 compares system-wide energy efficiency (tokens/J). SERENO achieves 3.52 tokens/J, significantly outperforming CPU-based solutions (1.81 tokens/J) and naive strategies like *Chunk-Sleep* (2.07 tokens/J). Although SERENO's efficiency is slightly lower than *Speculative* (3.78 tokens/J) due to the control overhead and forced yielding, it offers a better trade-off than heuristic throttling. This efficiency gain over *Chunk-Sleep* stems from a race-to-sleep strategy: SERENO utilizes the NPU's high-performance operating points to complete valid work quickly before yielding, while naive throttling prolongs execution time, incurring higher accumulated static power leakage.

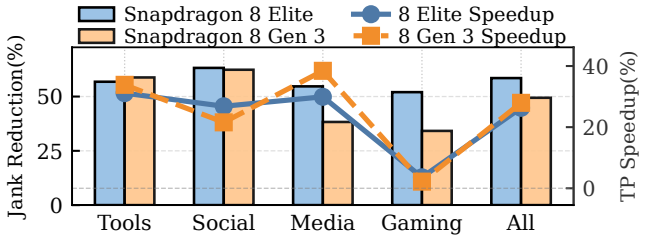


Figure 14: Robustness compared across generations. Jank rate reduction is compared with PowerServe.

Hardware generality. To verify hardware robustness, we replicate experiments on a Snapdragon 8 Gen 3 device (Figure 14). For CPU-dominant workloads like *Tools* and *Social*, SERENO maintains consistent effectiveness on the Gen 3 device, achieving jank rate reductions of 58.79% and 62.34% respectively, similar to the Snapdragon 8 Elite's 56.82% and 59.23% reductions. However, in *Gaming* scenarios, the reduction drops from 52.03% (Elite) to 29.70% (Gen 3). This is because the weaker Gen 3 GPU/CPU makes jank frames often stem from compute saturation rather than bandwidth contention, and SERENO's bandwidth regulation is naturally bounded once the bottleneck shifts to compute—though it remains effective in bandwidth-bound scenarios.

Task and model robustness. LLM speculation acceptance rates vary by task complexity. We use three applications (Chrome, WeChat, YouTube) to evaluate SERENO across three distinct datasets: ShareGPT, GSM8K, and HumanEval, randomly selecting 150 prompts from each. Despite semantic differences, SERENO maintains a stable throughput (15.89–17.16 tokens/s) and consistent jank rate (7.45–8.12%), con-

firming that our endogenous signal detects physical memory pressure regardless of token distribution. We did not evaluate output accuracy, as SERENO is lossless by design.

We standardize on the Llama family as representative of Transformer workloads with invariant memory access patterns. Our 1B-parameter draft model is relatively large for on-device use, establishing a conservative lower bound on control granularity: smaller drafts (e.g., <500M) incur lower per-layer latency, offering finer preemption points and further enhancing SERENO's responsiveness.

System overhead. Compared to the PowerServe baseline, SERENO introduces a marginal CPU utilization increase of ~4.7% (on a single core) due to the bandwidth controller and N-gram Filling. Regarding memory footprint, the system incurs an additional 0.86 GB usage (+19%). This increase is dominated by draft-model weights—an inherent cost of speculative decoding—not SERENO's control logic; the precompiled static graphs specific to SERENO add a negligible <70 MB. Given modern 12 GB+ RAM, this is a justifiable trade-off for the interactivity gains.

8 Related Work & Discussion

Efficient On-Device LLM Inference. Optimization techniques such as quantization [51, 85] and sparsity [6, 80] reduce overheads, while inference frameworks [19, 27, 29, 55, 60, 61, 66, 73, 86, 98, 100] use heterogeneous compute to maximize throughput. These works implicitly assume exclusive resource access and overlook contention in mobile multitasking. SERENO's control plane is orthogonal to these optimizations and can be integrated atop many of these engines.

Speculative Decoding Techniques. Speculative decoding [18, 46] and variants like EAGLE-X [48–50], and others [16, 25, 35, 37, 47, 59, 62, 81, 84, 101, 102] have become standard for on-device inference acceleration. Prior studies treat speculation purely as acceleration; SERENO instead repurposes its algorithmic elasticity into a mechanism for fine-grained, interruptible resource scheduling, turning an acceleration technique into a QoS control primitive. This principle also applies to other methods, such as early exit [22, 52], by steering shallow/deep paths under memory pressure.

Memory Bandwidth Management. Many works notice the importance of memory bandwidth management in system-level QoS [20, 42]. Server-grade systems like LIBRA [107], MT^2 [103], and others [2, 83, 105] rely on hardware enforcement (e.g., RDT [38]/MPAM), while BWLOCK-style systems [2, 3, 104, 105] throttle controllable CPU cores. Mobile NPUs expose neither portable flow control nor scheduler-visible bandwidth accounting, so these mechanisms do not apply. SERENO instead restructures inference via elastic speculative decoding, providing a software-only solution compatible with existing hardware.

Platform scope. Our experimental claims are limited to commercial Snapdragon devices, where we can run foreground/LLM co-execution; source analysis extends the architectural evidence to Dimensity and Exynos SoCs, which expose vendor-specific accelerator traffic or thermal hooks but no portable per-application NPU bandwidth quota. Apple SoCs and unified-memory laptops likely face similar UMA-plus-accelerator contention, though closed toolchains limit mechanism-level claims; discrete-GPU laptops differ because inference and rendering may reside in separate memory. SERENO is thus most relevant to unified-memory smartphones, tablets, AI PCs, AR/VR headsets, and robots.

Model architecture scope. SERENO is evaluated on dense Transformer LLMs, but its requirement is broader: autoregressive inference must expose fine-grained points where tentative work can be stopped without corrupting committed state. The principle applies to MoE models [24, 39] as long as decoding stays bandwidth-bound and speculative work is preemptible, and to SSM-style or hybrid models [32], though Mamba/SSM speculation requires architecture-aware state rollback and verification [92, 96]. What carries over is the control principle, not the exact Transformer-layer actuator.

Overhead optimization. The current prototype uses a 1B draft model, making the reported memory overhead conservative for smaller on-device assistants. Smaller or quantized draft models [51, 85], adaptive speculation depth, and shared KV-cache or shared-prefix representations could further increase the frequency of yielding.

Future hardware trends. We expect contention to remain relevant as mobile NPUs scale: compute throughput and accelerator count grow far faster than package-level DRAM bandwidth and energy, so on UMA devices the gap between how fast accelerators request data and how much shared traffic foreground tasks tolerate keeps widening. Unless future SoCs expose portable per-client NPU bandwidth controls, software-level cooperative yielding remains necessary even as absolute bandwidth improves.

9 Conclusion

We presented SERENO, a software-only framework that resolves the asymmetric interference from on-device LLMs by repurposing speculative decoding into a fine-grained control primitive. SERENO restores foreground smoothness to near-native levels while sustaining competitive LLM throughput.

Acknowledgments

We sincerely thank our shepherd and the anonymous reviewers for their insightful comments. This work is supported in part by the Fundamental and Interdisciplinary Disciplines Breakthrough Plan of the Ministry of Education of China (Grant No. JYB2025XDXM113), and the National Natural Science Foundation of China (Grant Nos. 62132014 and 62372287). Mingkai Dong (mingkaidong@sjtu.edu.cn) and Zeyu Mi (yzmizeyu@sjtu.edu.cn) are the corresponding authors.

References

[1] Helen Sydney Adams. Oppo to make ai phones accessible to 50m smartphone users, 2024.

[2] Homa Aghilinasab, Waqar Ali, Heechul Yun, and Rodolfo Pellizzoni. Dynamic memory bandwidth allocation for real-time gpu-based soc platforms. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 39(11):3348–3360, 2020.

[3] Waqar Ali and Heechul Yun. Protecting real-time gpu kernels on integrated cpu-gpu soc platforms. *arXiv preprint arXiv:1712.08738*, 2017.

[4] AliExpress. OnePlus Ace Pro 10T 5G global rom 12/16gb snapdragon 8+ gen 1 120hz amoled display 150w charge 50mp triple camera smartphone. https://www.aliexpress.us/item/3256808828110227.html, 2025. Accessed: 2025-12-09.

[5] AliExpress. Original Oneplus 13 mobile phone 50.0mp camera 100w charge ip69 waterproof 6000mah battery 6.82" oled 120hz snapdragon 8 elite. https://www.aliexpress.us/item/3256810002639267.html, 2025. Accessed: 2025-12-09.

[6] Keivan Alizadeh, Seyed Iman Mirzadeh, Dmitry Belenko, S Khatamifard, Minsik Cho, Carlo C Del Mundo, Mohammad Rastegari, and Mehrdad Farajtabar. Llm in a flash: Efficient large language model inference with limited memory. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 12562–12584, 2024.

[7] Android Developers. Ui jank detection. https://developer.android.com/studio/profile/jank-detection, 2023. Accessed: 2025-12-07.

[8] Android Developers. Monitor app health with Android Vitals. https://developer.android.com/topic/performance/vitals, 2025. Accessed: 2025-12-07.

[9] Android Developers. Simpleperf command-line tool. https://developer.android.com/ndk/guides/simpleperf, 2025. Accessed: 2025-12-07.

[10] Android Developers. Slow rendering. https://developer.android.com/topic/performance/rendering, 2025. Accessed: 2025-12-07.

[11] Apple Inc. Apple Intelligence: Personal intelligence that puts powerful generative models right at the core of your iPhone, iPad, and Mac. https://www.apple.com/apple-intelligence/, 2024. Accessed: 2025-12-07.

[12] ARM Limited. Memory system resource partitioning and monitoring (MPAM) for A-profile architecture. https://documentation-service.arm.com/static/63fc88d656ea36189d4e77df, 2025. Accessed: 2025-12-07.

[13] Karl Johan Åström and Tore Hägglund. The future of pid control. *Control engineering practice*, 9(11):1163–1175, 2001.

[14] Karl Johan Åström and Tore Hägglund. Revisiting the ziegler–nichols step response method for pid control. *Journal of process control*, 14(6):635–650, 2004.

[15] Rakesh P Borase, DK Maghade, SY Sondkar, and SN Pawar. A review of pid control, tuning methods and applications. *International Journal of Dynamics and Control*, 9(2):818–827, 2021.

[16] Tianle Cai, Yuhong Li, Zhengyang Geng, Hongwu Peng, Jason D. Lee, Deming Chen, and Tri Dao. Medusa: Simple llm inference acceleration framework with multiple decoding heads. In *Proceedings of the 41st International Conference on Machine Learning*, ICML'24. JMLR.org, 2024.

[17] ChargerLAB. ChargerLAB POWER-Z KM003C. https://www.power-z.com/products/chargerlab-power-z-km003c, 2025. Accessed: 2025-12-11.

[18] Charlie Chen, Sebastian Borgeaud, Geoffrey Irving, Jean-Baptiste Lespiau, Laurent Sifre, and John Jumper. Accelerating large language model decoding with speculative sampling, 2023.

[19] Le Chen, Dahu Feng, Erhu Feng, Yingrui Wang, Rong Zhao, Yubin Xia, Pinjie Xu, and Haibo Chen. Characterizing mobile soc for accelerating heterogeneous llm inference. In *Proceedings of the ACM SIGOPS 31st Symposium on Operating Systems Principles*, SOSP '25, page 359–374, New York, NY, USA, 2025. Association for Computing Machinery.

[20] Russell Clapp, Martin Dimitrov, Karthik Kumar, Vish Viswanathan, and Thomas Willhalm. Quantifying the performance impact of memory latency and bandwidth for big data workloads. In *2015 IEEE International Symposium on Workload Characterization*, pages 213–224. IEEE, 2015.

[21] Karl Cobbe, Vineet Kosaraju, Mohammad Bavarian, Mark Chen, Heewoo Jun, Lukasz Kaiser, Matthias Plappert, Jerry Tworek, Jacob Hilton, Reiichiro Nakano, et al. Training verifiers to solve math word problems. *arXiv preprint arXiv:2110.14168*, 2021.

[22] Mostafa Elhoushi, Akshat Shrivastava, Diana Liskovich, Basil Hosmer, Bram Wasti, Liangzhen Lai, Anas Mahmoud, Bilge Acun, Saurabh Agarwal, Ahmed Roman, et al. Layerskip: Enabling early exit inference and self-speculative decoding. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 12622–12642, 2024.

[23] Pouya Esmaili-Dokht, Francesco Sgherzi, Valeria Soldera Girelli, Isaac Boixaderas, Mariana Carmin, Alireza Monemi, Adria Armejach, Estanislao Mercadal, German Llort, Petar Radojković, et al. A mess of memory system benchmarking, simulation and application profiling. In *2024 57th IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 136–152. IEEE, 2024.

[24] William Fedus, Barret Zoph, and Noam Shazeer. Switch transformers: Scaling to trillion parameter models with simple and efficient sparsity. *Journal of Machine Learning Research*, 23(120):1–39, 2022.

[25] Yichao Fu, Peter Bailis, Ion Stoica, and Hao Zhang. Break the sequential dependency of llm inference using lookahead decoding. In *Proceedings of the 41st International Conference on Machine Learning*, ICML'24. JMLR.org, 2024.

[26] Georgi Gerganov et al. The hexagon backend for llama.cpp. https://github.com/ggml-org/llama.cpp/blob/master/docs/backend/hexagon/README.md, 2023. Accessed: 2025-12-11.

[27] Georgi Gerganov et al. llama.cpp: LLM inference in C/C++. https://github.com/ggml-org/llama.cpp, 2023. Release build b3706.

[28] Google. Gemini Nano: The most efficient model for on-device tasks. https://developer.android.com/ai/gemini-nano, 2024. Accessed: 2025-12-07.

[29] Google. LiteRT: High-performance on-device ai runtime. https://ai.google.dev/edge/litert, 2024. Accessed: 2025-12-07.

[30] Google. Perfetto documentation, 2024. Accessed: 2025-12-07.

[31] Aaron Grattafiori, Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, et al. The llama 3 herd of models, 2024.

[32] Albert Gu and Tri Dao. Mamba: Linear-time sequence modeling with selective state spaces, 2023.

[33] Chang C Hang, Karl Johan Åström, and Weng Khuen Ho. Refinements of the ziegler–nichols tuning formula. *IEE Proceedings D (Control Theory and Applications)*, 138(2):111–118, 1991.

[34] Zixu Hao, Jianyu Wei, Tuowei Wang, Minxing Huang, Huiqiang Jiang, Shiqi Jiang, Ting Cao, and Ju Ren. Scaling llm test-time compute with mobile npu on smartphones, 2025.

[35] Zhenyu He, Zexuan Zhong, Tianle Cai, Jason Lee, and Di He. REST: Retrieval-based speculative decoding. In Kevin Duh, Helena Gomez, and Steven Bethard, editors, *Proceedings of the 2024 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume 1: Long Papers)*, pages 1582–1595, Mexico City, Mexico, June 2024. Association for Computational Linguistics.

[36] Joseph L Hellerstein, Yixin Diao, Sujay Parekh, and Dawn M Tilbury. *Feedback control of computing systems*. John Wiley & Sons, 2004.

[37] Coleman Hooper, Sehoon Kim, Hiva Mohammadzadeh, Hasan Genc, Kurt Keutzer, Amir Gholami, and Yakun Sophia Shao. Speed: Speculative pipelined execution for efficient decoding. In *Enhancing LLM Performance: Efficacy, Fine-Tuning, and Inference Techniques*, pages 19–32. Springer, 2025.

[38] Intel Corporation. Intel® resource director technology (Intel® RDT). https://www.intel.com/content/www/us/en/architecture-and-technology/resource-director-technology.html, 2025. Accessed: 2025-12-07.

[39] Albert Q. Jiang, Alexandre Sablayrolles, Antoine Roux, Arthur Mensch, Blanche Savary, Chris Bamford, Devendra Singh Chaplot, Diego de Las Casas, Emma Bou Hanna, Florian Bressand, Gianna Lengyel, Guillaume Bour, Guillaume Lample, Lélio Renard Lavaud, Lucile Saulnier, Marie-Anne Lachaux, Pierre Stock, Sandeep Subramanian, Sophia Yang, Szymon Antoniak, Teven Le Scao, Théophile Gervet, Thibaut Lavril, Thomas Wang, Timothée Lacroix, and William El Sayed. Mixtral of experts, 2024.

[40] Michael A Johnson and Mohammad H Moradi. *PID control*. Springer, 2005.

[41] Carl Knospe. Pid control. *IEEE Control Systems Magazine*, 26(1):30–31, 2006.

[42] Johannes Langguth, Xing Cai, and Mohammed Sourouri. Memory bandwidth contention: Communication vs computation tradeoffs in supercomputers with multicore architectures. In *2018 IEEE 24th International Conference on Parallel and Distributed Systems (ICPADS)*, pages 497–506. IEEE, 2018.

[43] Etienne Le Sueur and Gernot Heiser. Dynamic voltage and frequency scaling: The laws of diminishing returns. In *Proceedings of the 2010 international conference on Power aware computing and systems*, pages 1–8, 2010.

[44] Changhyun Lee, Chunjong Park, Keon Jang, Sue Moon, and Dongsu Han. Accurate latency-based congestion feedback for datacenters. In *2015 USENIX Annual Technical Conference (USENIX ATC 15)*, pages 403–415, Santa Clara, CA, July 2015. USENIX Association.

[45] Changhyun Lee, Chunjong Park, Keon Jang, Sue Moon, and Dongsu Han. Dx: Latency-based congestion control for datacenters. *IEEE/ACM Transactions on Networking*, 25(1):335–348, 2017.

[46] Yaniv Leviathan, Matan Kalman, and Yossi Matias. Fast inference from transformers via speculative decoding. In Andreas Krause, Emma Brunskill, Kyunghyun Cho, Barbara Engelhardt, Sivan Sabato, and Jonathan Scarlett, editors, *Proceedings of the 40th International Conference on Machine Learning*, volume 202 of *Proceedings of Machine Learning Research*, pages 19274–19286. PMLR, 23–29 Jul 2023.

[47] Minghan Li, Xilun Chen, Ari Holtzman, Beidi Chen, Jimmy Lin, Scott Yih, and Victoria Lin. Nearest neighbor speculative decoding for llm generation and attribution. *Advances in Neural Information Processing Systems*, 37:80987–81015, 2024.

[48] Yuhui Li, Fangyun Wei, Chao Zhang, and Hongyang Zhang. Eagle-2: Faster inference of language models with dynamic draft trees, 2024.

[49] Yuhui Li, Fangyun Wei, Chao Zhang, and Hongyang Zhang. Eagle: speculative sampling requires rethinking feature uncertainty. In *Proceedings of the 41st International Conference on Machine Learning*, ICML'24. JMLR.org, 2024.

[50] Yuhui Li, Fangyun Wei, Chao Zhang, and Hongyang Zhang. Eagle-3: Scaling up inference acceleration of large language models via training-time test, 2025.

[51] Ji Lin, Jiaming Tang, Haotian Tang, Shang Yang, Wei-Ming Chen, Wei-Chen Wang, Guangxuan Xiao, Xingyu Dang, Chuang Gan, and Song Han. Awq: Activation-aware weight quantization for on-device llm compression and acceleration. In P. Gibbons, G. Pekhimenko, and C. De Sa, editors, *Proceedings of Machine Learning and Systems*, volume 6, pages 87–100, 2024.

[52] Fangcheng Liu, Yehui Tang, Zhenhua Liu, Yunsheng Ni, Duyu Tang, Kai Han, and Yunhe Wang. Kangaroo: Lossless self-speculative decoding for accelerating LLMs via double early exiting. In *The Thirty-eighth Annual Conference on Neural Information Processing Systems*, 2024.

[53] Xiao Liu, Bo Qin, Dongzhu Liang, Guang Dong, Hanyu Lai, Hanchen Zhang, Hanlin Zhao, Iat Long Iong, Jiadai Sun, Jiaqi Wang, et al. Autoglm: Autonomous foundation agents for guis. *arXiv preprint arXiv:2411.00820*, 2024.

[54] Xudong Lu, Yinghao Chen, Cheng Chen, Hui Tan, Boheng Chen, Yina Xie, Rui Hu, Guanxin Tan, Renshou Wu, Yan Hu, et al. Bluelm-v-3b: Algorithm and system co-design for multimodal large language models on mobile devices. In *Proceedings of the Computer Vision and Pattern Recognition Conference*, pages 4145–4155, 2025.

[55] Chengfei Lv, Chaoyue Niu, Renjie Gu, Xiaotang Jiang, Zhaode Wang, Bin Liu, Ziqi Wu, Qiulin Yao, Congyu Huang, Panos Huang, Tao Huang, Hui Shu, Jinde Song, Bin Zou, Peng Lan, Guohuan Xu, Fei Wu, Shaojie Tang, Fan Wu, and Guihai Chen. Walle: An End-to-End, General-Purpose, and Large-Scale production system for Device-Cloud collaborative machine learning. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 249–265, Carlsbad, CA, July 2022. USENIX Association.

[56] MediaTek Inc. MediaTek AI: Enhancing your life with AI on edge devices. https://i.mediatek.com/ai, 2025. Accessed: 2025-12-07.

[57] MediaTek Inc. Mediatek dimensity 9400. https://www.mediatek.com/products/smartphones/mediatek-dimensity-9400, 2025. Accessed: 2025-12-09.

[58] Sachin Mehta, Mohammad Hossein Sekhavat, Qingqing Cao, Maxwell Horton, Yanzi Jin, Chenfan Sun, Seyed Iman Mirzadeh, Mahyar Najibi, Dmitry Belenko, Peter Zatloukal, and Mohammad Rastegari. OpenELM: An efficient language model family with open training and inference framework. In *Workshop on Efficient Systems for Foundation Models II @ ICML 2024*, 2024.

[59] Xupeng Miao, Gabriele Oliaro, Zhihao Zhang, Xinhao Cheng, Zeyu Wang, Zhengxin Zhang, Rae Ying Yee Wong, Alan Zhu, Lijie Yang, Xiaoxiang Shi, Chunan Shi, Zhuoming Chen, Daiyaan Arfeen, Reyna Abhyankar, and Zhihao Jia. Specinfer: Accelerating large language model serving with tree-based speculative inference and verification. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, ASPLOS '24, page 932–949, New

York, NY, USA, 2024. Association for Computing Machinery.

[60] Microsoft. ONNX Runtime: Cross-platform, high performance ml inference and training accelerator. https://onnxruntime.ai/, 2024. Accessed: 2025-12-07.

[61] MLC team. MLC-LLM, 2023-2025.

[62] Giovanni Monea, Armand Joulin, and Edouard Grave. Pass: Parallel speculative sampling, 2023.

[63] Yutao Mou, Shiyu Huang, Haoyu Kuang, Buzhou Tang, Yu Chen, Jiongchao Jin, Ruoyu Zhang, Ruixuan Huang, Hao Mou, Yi Li, Xuan Luo, Jingcheng Yin, Yang Liu, Yong Peng, Ziyi Ni, Jie Li, and Wei Li. SmartBench: Is your LLM truly a good Chinese smartphone assistant? In *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing (EMNLP 2025)*, 2025. Includes systematic categorization of on-device LLM functionalities: text summarization, Q&A, information extraction, content creation, and notification management.

[64] OnePlusOSS. android_kernel_modules_and_devicetree_oneplus_mt6991. https://github.com/OnePlusOSS/android_kernel_modules_and_devicetree_oneplus_mt6991, 2025. Branch: oneplus/mt6991_v_15.0.2_ace5_ultra; Commit: 76d6e1e5b8deb0bcd00355d7a9e68638a501d8bc; Accessed: 2025-12-07.

[65] OnePlusOSS. android_kernel_modules_and_devicetree_oneplus_sm8750 (repo sync checkout). https://github.com/OnePlusOSS/kernel_manifest, 2025. Branch: oneplus/sm8750; Manifest File: oneplus_13.xml; Kernel Path: kernel_platform/msm-kernel; Commit: a85bac41e21a790e216039cde1d34a6c5d6416d1; Accessed: 2025-12-07.

[66] PowerServe Project. PowerServe: High-speed and easy-use LLM serving framework for local deployment. https://github.com/powerserve-project/PowerServe, 2025. Accessed: 2025-12-07.

[67] Primate Labs Inc. Android benchmark chart. https://browser.geekbench.com/android-benchmarks, 2025. Accessed: 2025-12-09.

[68] Primate Labs Inc. Geekbench. https://www.geekbench.com/, 2025. Accessed: 2025-12-11.

[69] Jiaxing Qiu, Zijie Zhou, Yang Li, Zhenhua Li, Feng Qian, Hao Lin, Di Gao, Haitao Su, Xin Miao, Yunhao Liu, and Tianyin Xu. vsoc: Efficient virtual system-on-chip on heterogeneous hardware. In *Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles*, SOSP '24, page 558–573, New York, NY, USA, 2024. Association for Computing Machinery.

[70] Qualcomm. Heterogeneous Task Processor (HTP) Backend. https://docs.qualcomm.com/doc/80-63442-10/topic/htp_backend.html, 2025. Accessed: 2025-12-07.

[71] Qualcomm Technologies, Inc. Hexagon DSP architecture overview. https://docs.qualcomm.com/doc/80-70017-15SC/topic/architecture.html, 2025. Accessed: 2025-12-09.

[72] Qualcomm Technologies, Inc. Qualcomm AI Engine. https://www.qualcomm.com/processors/ai-engine, 2025. Accessed: 2025-12-07.

[73] Qualcomm Technologies, Inc. Qualcomm AI Hub. https://aihub.qualcomm.com/, 2025. Accessed: 2025-12-07.

[74] Qualcomm Technologies, Inc. Qualcomm Neural Network (QNN) SDK Documentation. https://docs.qualcomm.com/nav/home/index_QNN.html?product=1601111740009302, 2025. Accessed: 2025-12-07.

[75] Qualcomm Technologies, Inc. Snapdragon 8 elite mobile platform. https://www.qualcomm.com/smartphones/products/8-series/snapdragon-8-elite-mobile-platform, 2025. Accessed: 2025-12-09.

[76] Qualcomm Technologies, Inc. Snapdragon profiler. https://www.qualcomm.com/developer/software/snapdragon-profiler, 2025. Accessed: 2025-12-07.

[77] Samsung Electronics. Galaxy AI: The new era of mobile AI. https://www.samsung.com/galaxy-ai/, 2024. Accessed: 2026-05-05.

[78] Samsung Open Source Release Center. Exynos 2400 (s5e9945) kernel source. https://opensource.samsung.com/, 2024. Device tree files include s5e9945-npu.dtsi, s5e9945-bts.dtsi, and s5e9945-thermal.dtsi; accessed: 2026-05-05.

[79] Claude E Shannon. A mathematical theory of communication. *The Bell system technical journal*, 27(3):379–423, 1948.

[80] Yixin Song, Zeyu Mi, Haotong Xie, and Haibo Chen. Powerinfer: Fast large language model serving with a consumer-grade gpu. In *Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles*, SOSP '24, page 590–606, New York, NY, USA, 2024. Association for Computing Machinery.

[81] Hanshi Sun, Zhuoming Chen, Xinyu Yang, Yuandong Tian, and Beidi Chen. Triforce: Lossless acceleration of long sequence generation with hierarchical speculative decoding. In *First Conference on Language Modeling*, 2024.

[82] Shaotong Sun, Yifan Zhu, Xingzhi Ye, and Chen Ding. Measuring data access latency in large cpu caches. In *Proceedings of the International Symposium on Memory Systems*, pages 129–139, 2024.

[83] Hanul Sung, Jeesoo Min, Sujin Ha, and Hyeonsang Eom. Ombm: optimized memory bandwidth management for ensuring qos and high server utilization. *Cluster Computing*, 22(1):161–174, 2019.

[84] Ruslan Svirschevski, Avner May, Zhuoming Chen, Beidi Chen, Zhihao Jia, and Max Ryabinin. Specexec: massively parallel speculative decoding for interactive llm inference on consumer devices. In *Proceedings of the 38th International Conference on Neural Information Processing Systems*, NIPS '24, Red Hook, NY, USA, 2024. Curran Associates Inc.

[85] Fuwen Tan, Royson Lee, Łukasz Dudziak, Shell Xu Hu, Sourav Bhattacharya, Timothy Hospedales, Georgios Tzimiropoulos, and Brais Martinez. Mobilequant: Mobile-friendly quantization for on-device language models, 2024.

[86] Tencent. ncnn: High-performance neural network inference framework optimized for the mobile platform. https://github.com/Tencent/ncnn, 2024. Accessed: 2025-12-07.

[87] The Perfetto Authors. FrameTimeline trace processor module. https://perfetto.dev/docs/data-sources/frametimeline, 2025. Accessed: 2025-12-07.

[88] UL Solutions. 3DMark. https://www.3dmark.com/, 2025. Accessed: 2025-12-11.

[89] UL Solutions. Best smartphones - 3dmark benchmark performance. https://benchmarks.ul.com/compare/best-smartphones, 2025. Accessed: 2025-12-09.

[90] Midhul Vuppalapati and Rachit Agarwal. Tiered memory management: Access latency is the key! In *Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles*, pages 79–94, 2024.

[91] Jikai Wang, Yi Su, Juntao Li, Qingrong Xia, Zi Ye, Xinyu Duan, Zhefeng Wang, and Min Zhang. OPT-tree: Speculative decoding with adaptive draft tree structure. *Transactions of the Association for Computational Linguistics*, 13:188–199, 2025.

[92] Junxiong Wang, Daniele Paliotta, Avner May, Alexander M. Rush, and Tri Dao. The mamba in the llama: Distilling and accelerating hybrid models, 2024.

[93] Junyang Wang, Haiyang Xu, Haitao Jia, Xi Zhang, Ming Yan, Weizhou Shen, Ji Zhang, Fei Huang, and Jitao Sang. Mobile-agent-v2: Mobile device operation assistant with effective navigation via multi-agent collaboration. *Advances in Neural Information Processing Systems*, 37:2686–2710, 2024.

[94] Zibo Wang, Yijia Zhang, Fuchun Wei, Bingqiang Wang, Yanlin Liu, Zhiheng Hu, Jingyi Zhang, Xiaoxin Xu, Jian He, Xiaoliang Wang, et al. Using analytical performance/power model and fine-grained dvfs to enhance ai accelerator energy efficiency. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1*, pages 1118–1132, 2025.

[95] Hao Wen, Yuanchun Li, Guohong Liu, Shanhui Zhao, Tao Yu, Toby Jia-Jun Li, Shiqi Jiang, Yunhao Liu, Yaqin Zhang, and Yunxin Liu. Autodroid: Llm-powered task automation in android. In *Proceedings of the 30th Annual International Conference on Mobile Computing and Networking*, ACM MobiCom '24, page 543–557, New York, NY, USA, 2024. Association for Computing Machinery.

[96] Yangchao Wu, Zongyue Qin, Alex Wong, and Stefano Soatto. Stree: Speculative tree decoding for hybrid state-space models, 2025.

[97] Yuanpei Wu, Dong Du, Chao Xu, Yubin Xia, Ming Fu, Binyu Zang, and Haibo Chen. D-vsync: Decoupled rendering and displaying for smartphone graphics. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1*, ASPLOS '25, page 326–341, New York, NY, USA, 2025. Association for Computing Machinery.

[98] Daliang Xu, Hao Zhang, Liming Yang, Ruiqi Liu, Gang Huang, Mengwei Xu, and Xuanzhe Liu. Fast on-device llm inference with npus. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1*, ASPLOS '25, page 445–462, New York, NY, USA, 2025. Association for Computing Machinery.

[99] Yifan Xu, Xiao Liu, Xinghan Liu, Jiaqi Fu, Hanchen Zhang, Bohao Jing, Shudan Zhang, Yuting Wang, Wenyi Zhao, and Yuxiao Dong. Mobilerl: Online agentic reinforcement learning for mobile gui agents. *arXiv preprint arXiv:2509.18119*, 2025.

[100] Zhenliang Xue, Yixin Song, Zeyu Mi, Xinrui Zheng, Yubin Xia, and Haibo Chen. Powerinfer-2: Fast large language model inference on a smartphone, 2024.

[101] Seongjun Yang, Gibbeum Lee, Jaewoong Cho, Dimitris Papailiopoulos, and Kangwook Lee. Predictive pipelined decoding: A compute-latency trade-off for exact LLM decoding. In *Workshop on Efficient Systems for Foundation Models @ ICML2023*, 2023.

[102] Hanling Yi, Feng Lin, Hongbin Li, Ning Peiyang, Xiaotian Yu, and Rong Xiao. Generation meets verification: Accelerating large language model inference with smart parallel auto-correct decoding. In *Findings of the Association for Computational Linguistics: ACL 2024*, pages 5285–5299, 2024.

[103] Jifei Yi, Benchao Dong, Mingkai Dong, Ruizhe Tong, and Haibo Chen. {MTˆ 2}: Memory bandwidth regulation on hybrid {NVM/DRAM} platforms. In *20th USENIX Conference on File and Storage Technologies (FAST 22)*, pages 199–216, 2022.

[104] Heechul Yun, Waqar Ali, Santosh Gondi, and Siddhartha Biswas. Bwlock: A dynamic memory access control framework for soft real-time applications on multicore platforms. *IEEE Transactions on Computers*, 66(7):1247–1252, 2016.

[105] Heechul Yun, Gang Yao, Rodolfo Pellizzoni, Marco Caccamo, and Lui Sha. Memguard: Memory bandwidth reservation system for efficient performance isolation in multi-core platforms. In *2013 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 55–64, 2013.

[106] Wei Zeng, Xiaozhe Ren, Teng Su, Hui Wang, Yi Liao, Zhiwei Wang, Xin Jiang, ZhenZhang Yang, Kaisheng Wang andx Xiaoda Zhang, Chen Li, Ziyan Gong, Yifan Yao, Xinjing Huang, Jun Wang, Jianfeng Yu, Qi Guo, Yue Yu, Yan Zhang, Jin Wang, Hengtao Tao, Dasen Yan, Zexuan Yi, Fang Peng, Fangqing Jiang, Han Zhang, Lingfeng Deng, Yehong Zhang, Zhe Lin, Chao Zhang, Shaojie Zhang, Mingyue Guo, Shanzhi Gu, Gaojun Fan, Yaowei Wang, Xuefeng Jin, Qun Liu, and Yonghong Tian. Pangu-α: Large-scale autoregressive pretrained chinese language models with auto-parallel computation. *CoRR*, abs/2104.12369, 2021.

[107] Ying Zhang, Jian Chen, Xiaowei Jiang, Qiang Liu, Ian M. Steiner, Andrew J. Herdrich, Kevin Shu, Ripan Das, Long Cui, and Litrin Jiang. Libra: Clearing the cloud through dynamic memory bandwidth management. In *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 815–826, 2021.

[108] John G Ziegler and Nathaniel B Nichols. Optimum settings for automatic controllers. *Transactions of the American society of mechanical engineers*, 64(8):759–765, 1942.

LifeLine: An Object-Page Lifetime Alignment GC Enabling Minimal Memory Copying for Mobile Devices

Jiacheng Huang[1], Yunmo Zhang[1], Qingan Li[2], Junqiao Qiu[1], Chun Jason Xue[3]

[1]*City University of Hong Kong*
[2]*Wuhan University*
[3]*Mohamed bin Zayed University of Artificial Intelligence*

Abstract

LifeLine is a garbage collection (GC) framework for Android that tackles the fundamental object–page lifetime mismatch. Existing copying collectors move objects, while the OS manages pages; mixed lifetimes within a page force excessive object copying instead of efficient page remapping. LifeLine explicitly aligns object lifetimes with physical pages via three components. First, lifetime-based graph partitioning monitors reference updates and partitions the object graph into subgraphs with strong lifetime affinity. Second, lifetime-aligned GC packs these subgraphs into pages so that each page contains almost entirely live or dead objects, enabling effective page-level management. Third, near-zero-copy GC exploits this bimodal per-page liveness by remapping mostly-live pages and copying only the few surviving objects from mostly-dead pages in cooperation with the OS. Implemented in the Android Runtime and evaluated on real smartphones and popular mobile applications, LifeLine significantly cuts GC copy volume by 57.4%, reduces GC time by 22.7% on average, and improves user-visible responsiveness with modest CPU and memory overhead.

1 Introduction

Android is one of the most widely used mobile operating systems, serving billions of users worldwide [9]. It relies on the Android Runtime (ART) to execute applications and uses a copying garbage collection (GC) mechanism to manage dynamic memory [35, 39, 49]. As shown in Figure 1, copying GCs reclaim fragmented memory by physically relocating live objects to a contiguous region. However, such physical data movement is expensive, as it consumes memory bandwidth and often requires suspending application threads to ensure consistency. For example, memory access latency can increase dramatically during GC execution, as shown in Figure 2. As a result, GC activity can hurt overall performance and cause frame drops and stuttering that visibly degrade the user experience [20, 49].

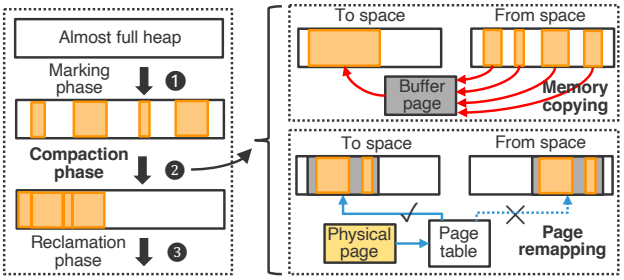


Figure 1: Garbage collection mechanisms in ART. (a) The standard three-phase lifecycle shared by collectors. (b) Compaction strategies utilized by the Concurrent Mark-Compact (CMC) collector: object copying versus OS-assisted page remapping.

To mitigate GC-induced copying overhead, state-of-the-art collectors in ART employ a more efficient memory-movement operation, i.e., page remapping [2, 3]. Supported by the operating system (OS), this operation moves entire pages of memory via modifying page table entries rather than copying the underlying data [31, 52, 61]. However, as illustrated in Figure 1(b), existing collectors fail to fully exploit this opportunity due to a fundamental *object–page lifetime mismatch* problem. Specifically, while the OS manages memory at page granularity, objects targeted by GC are typically much smaller and irregularly sized. Moreover, objects within the same page often have highly diverse lifetimes. As a result, most pages contain a mixture of live and dead objects. Existing collectors are forced to fall back to expensive object-level copying to reclaim space, leaving the OS-assisted zero-copy operation largely underutilized. Generational GC [22, 47, 53] can partially mitigate this issue by separating young objects from mature ones, but it only provides coarse-grained lifetime distinctions. Even within the mature heap, objects on the same page may die at different phases, preventing generational GC from fully utilizing OS-assisted zero-copy mechanisms.

This paper aims to bridge this object-page mismatch by

aligning objects with memory pages according to their lifetimes. When objects in the same page have similar lifetimes, each page is basically either entirely live or entirely dead. Such alignment unifies the basic unit of operation for both GC and the OS: the page. The garbage collector can then reclaim dead objects by freeing whole pages and move surviving objects by remapping the pages that contain them. However, achieving this goal presents three challenges: (1) Lifetime prediction. Object lifetimes in mobile applications are heavily influenced by user behavior and are inherently difficult to predict. (2) Object-page alignment. Packing variable-sized objects into fixed-size pages requires a precise relocation strategy. (3) Low overhead. Mobile devices have limited hardware resources, so the solution must respect tight CPU and memory budgets.

To address these challenges, this paper presents **LifeLine**, a GC framework that solves the object–page mismatch by leveraging the structural properties of the object reference graph. Our design is driven by empirical observations of the object reference graph in Android applications, which show that *reference mutability presents lifetime correlation*. LifeLine leverages graph-related insights to partition the object reference graph into multiple subgraphs based on the mutability of edges. Edges within the same subgraph have similar lifetimes. Specifically, LifeLine comprises three main components: (1) a lifetime-based graph partition method, which identifies subgraphs whose objects have similar lifetimes by relying on mutability information of references within the graph structure. It samples historical field-modification behavior and generates guidance on how to partition the graph. (2) a subgraph–page lifetime alignment GC, which facilitates object–page alignment by mapping entire subgraphs to pages. This GC performs a moving collection over the heap to align subgraphs with pages and explicitly considers subgraph lifetimes when placing objects during GC. (3) a near-zero-copy GC, which exploits page-level primitives after alignment to reclaim and move memory via page operations rather than object copies.

We implement LifeLine in the Android Open Source Project (AOSP) and evaluate it on a Pixel 7 Pro using popular commercial applications. Compared to Android's state-of-the-art production GC, LifeLine reduces total GC copy volume by 57.4% and total GC time by 22.7%, while improving frame rendering smoothness with low memory and CPU overhead. To the best of our knowledge, LifeLine is the first GC that explicitly coordinates the lifetimes of object subgraphs and memory pages. The main contributions are as follows:

- We identify the object–page lifetime mismatch as the primary bottleneck preventing the effective use of OS-assisted compaction in Android runtimes.
- We propose a subgraph-based approach that aligns objects with pages using reference-graph properties. The approach includes a lifetime prediction method based on reference mutability and a GC scheme that places subgraphs on pages according to predicted lifetimes.
- We design and implement LifeLine, a closed-loop system that enables GCs to better exploit OS page-level primitives for minimizing copying overheads and improving mobile application performance.
- We evaluate LifeLine on production Android workloads, and demonstrate a large ratio of GC performance gains.

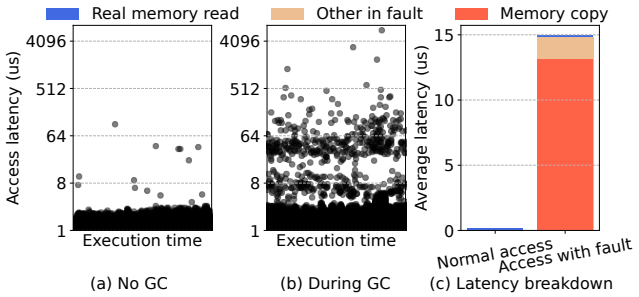


Figure 2: Impact of GC copying on memory access latency. (c) Breakdown of the tail access latency caused by Android's Concurrent Mark-Compact GC, showing that most of the time is spent copying memory to handle page faults.

2 Background

2.1 Garbage Collection in Android

Copying-based GC. Android executes all user applications on the ART [39, 49], which provides automatic memory management via a copying-based garbage collector. Copying GC is widely used in modern managed runtimes, including JVMs [7], .NET [43], and major JavaScript engines [29, 45]. Beyond automatic reclamation, it offers low fragmentation [11, 12], improved cache locality [34, 62], and fast bump-pointer allocation [66, 67]. Consequently, it serves as a fundamental building block in real-world systems.

GC Procedure. Figure 1(a) shows the overall GC process, which is triggered when memory runs low and consists of three phases. ❶ *Marking:* Starting from root objects, the collector traverses references and marks all reachable objects as live. ❷ *Compaction:* Live objects are relocated and packed contiguously. ❸ *Reclamation:* The remaining memory region is reclaimed and returned to the system. Among these phases, compaction typically dominates the performance overhead. During compaction, GC threads and application threads may concurrently modify the same heap regions, so the collector must enforce correctness with additional synchronization. There are three main approaches: (1) *Stop-the-world (STW) pauses* suspend all application threads while objects are moved. While simple to implement, the resulting latency is often unacceptable for interactive devices; therefore, ART re-

stricts STW primarily to root processing. (2) *Baker-style read barriers* [37] insert a small code snippet before each memory read, ensuring application threads only access objects already processed by the GC; Android's Concurrent Copying (CC) [5] GC uses this technique. (3) *Page-level memory protection* [38] uses the OS to protect unprocessed pages and redirects accesses through a fault handler that performs the necessary GC work. This approach underlies Android's Concurrent Mark-Compact (CMC) [1, 26] GC, the default and recommended collector in recent Android releases. However, regardless of the specific approach used, the physical movement of objects during the compaction phase remains a significant source of system overhead.

2.2 OS-assisted Memory Compaction

To move memory efficiently, ART's CMC collector cooperates closely with the OS. With OS assistance, CMC improves memory utilization and mitigates jank during GC [1, 49].

Concurrent Compaction. Conceptually, the collector incorporates two memory spaces: a *from-space* and a *to-space*. At the beginning of the compaction phase, it remaps the memory of all live objects into from-space using an OS remapping system call. It then progressively moves all live objects from from-space to to-space. During this process, CMC relies on the OS to regulate accesses to to-space. When an application thread accesses an unprocessed page, the OS blocks the thread and delivers a `SIGBUS` signal. A registered fault handler then migrates all objects on the faulting page from from-space to to-space. Once the handler completes, the thread resumes and proceeds on the now-processed page.

Strategies for Memory Movement. The key operation in compaction is migrating heap content, for which there are two main strategies, as shown in Figure 1(b):

• *Memory copying:* The straightforward approach is to copy live regions using memory copy operations. However, copying is costly, putting pressure on memory bandwidth, caches, and TLBs [31]. In GC, this cost is amplified [2] because application threads often have to wait for copying to complete.

• *Page remapping:* Recent Linux kernels introduce a zero-copy mechanism through the `userfaultfd` system call [30, 48, 64, 65]. Instead of copying data, the kernel updates Page Table Entries (PTEs) to remap the physical page backing a from-space address to the to-space address [2, 4]. This mechanism is significantly faster than physical copying, potentially reducing movement overhead by over 40% [4]. Within ART, GC threads can invoke `ioctl` with the `UFFDIO_MOVE` command to perform page remapping.

3 Motivation

With OS support, GC has the potential to achieve higher performance than traditional runtime-only approaches [1, 26, 49].

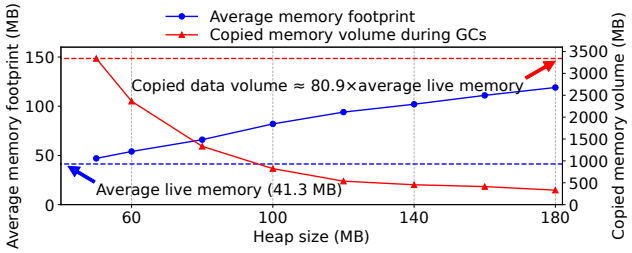


Figure 3: Memory copying overhead (red) and memory usage (blue) under different heap size settings. Smaller heaps reduce memory usage but dramatically increase memory copying.

We now examine how such support impacts the latest Android GC in practice, starting from a classic time–space trade-off.

3.1 Cost-Space Dilemma

Mobile devices are running increasingly rich and memory-intensive applications [10], while hardware scaling has not kept pace. RAM capacity has grown slowly: 8 GB devices still dominate the market, and 16 GB models are limited to a small set of high-end phones [9]. Consequently, GC must navigate a strict time–space trade-off to balance performance and memory usage.

To understand this trade-off in practice, we use Twitter (X) as a case study. We configure the application with a series of fixed heap sizes, run it for one minute under each setting, and measure the additional memory copying induced by GC. For each configuration, we also record the application's average memory footprint and the average live memory traced per collection. Figure 3 shows how average footprint and GC copy volume vary with heap size. With a small heap, the average footprint shrinks, improving memory efficiency, but the volume of copied memory grows rapidly. When the heap approaches its minimum, GC copies 3341 MB per minute, about 80.9× the application's live memory. With larger heaps, the footprint increases roughly linearly, reducing memory efficiency; GC copy volume drops but remains substantial. For example, even with a 140 MB heap, GC still copies about 10× the live memory.

High Cost of Memory Copying. These GC-induced copies impose significant overhead on the entire system, heavily interfering with normal execution [31]. They can directly block memory accesses: as discussed in §2.1, when GC and user threads run concurrently, each user thread must execute an extra read barrier or trigger a page-fault handler to copy the object before accessing it, substantially increasing memory access latency during GC. In addition, the large copy volume lengthens the compaction phase, causing longer GC-induced interruptions of application threads.

To quantify this impact, we again use Twitter (X) and compare memory access latency with and without GC activity. We treat the execution time of the code path that reads fields from

Java objects as the memory access latency, which includes both the actual memory access and GC-related handling. Because the number of accesses is huge, we sample: after every hundred object reads, we record one latency sample and mark whether it occurs during the GC compaction phase. As shown in Figure 2, memory accesses become much slower during GC. The tail latency (10-th parts per million, PPM) increases by about 60×. A closer look at these long-latency cases (Figure 2(c)) shows that, under normal conditions, average access latency is typically below 1 μs, whereas accesses that trigger GC work average about 15 μs, with roughly 87% of the time spent on GC-related copying. Since memory access lies on the critical performance path, blocking it with GC copying directly degrades overall application performance.

Object–Page Lifetime Mismatch Problem. To reduce copying, GC can, in principle, leverage the OS's page-remapping mechanism, as discussed in §2.2. However, current GCs still perform large amounts of copying and rarely benefit from page remapping. The core reason is the *object–page lifetime mismatch*. The OS manages memory at page granularity, whereas GC manages memory at object granularity. Page remapping is only useful when a page is almost entirely filled with live objects; if live and dead objects are heavily mixed, GC cannot reclaim memory simply by remapping that page.

To analyze this mismatch, we examine the liveness of each heap page in mobile applications. During GC tracing, we compute each page's survival ratio as the fraction of live-object size over page size, and then derive the distribution of per-page survival ratios across the heap, as shown in Figure 4 (bars above 100 denote pages whose survival ratio is 100%). A large fraction of pages exhibit medium survival ratios (e.g., 20–80%), where live and dead objects are interleaved, making page remapping ineffective. In contrast, remapping-friendly pages with very high survival ratios close to 100% account for only 16.1% and 13.5% of memory in these applications. Thus, for most of the heap, GC cannot exploit efficient page remapping and must fall back on expensive memory copying.

3.2 Key Idea and Challenges

Bridging Object–Page Lifetimes. To address the object–page lifetime mismatch, this paper aims to enable tighter cooperation between the runtime and the OS. Our key idea is: *by reorganizing the heap layout, we align object lifetimes with page lifetimes.* Concretely, we proactively place objects according to their lifetimes to turn the current roughly uniform, random survival-rate distribution into a *bimodal* one. We seek to reduce pages with medium survival ratios and increase those with very high or very low ratios, so that objects within a page tend to become live or dead together. Once object and page lifetimes are aligned, GC can apply fast page remapping to high-survival pages and perform only limited object copying on low-survival pages.

Challenges. However, realizing this idea in the context of mobile applications raises three major challenges:

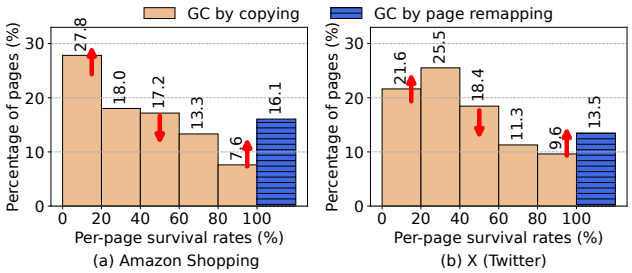


Figure 4: Per-page survival-rate distribution. The rightmost bars correspond to pages with a survival rate of 100%. Our goal is to reduce the number of pages with medium survival rates while increasing those with high or low survival rates.

• *Unpredictable object lifetimes:* In highly interactive mobile applications, object lifetimes are largely driven by unpredictable user behavior and are difficult to predict precisely. The first challenge is to extract and exploit usable lifetime-related information to approximate grouping of objects with similar lifetimes.

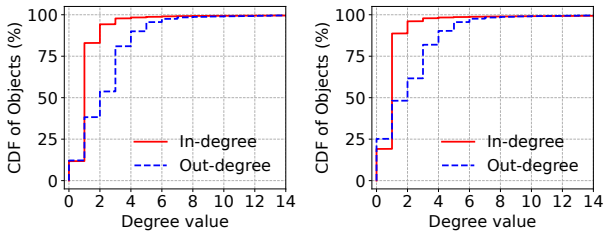
• *Finding structure in a highly dynamic environment:* Android objects are extremely dynamic: allocations and deallocations occur continuously, and object sizes and access patterns vary widely over time. The second challenge is to discover a simple, robust rule to physically organize objects onto pages efficiently and stably in this dynamic setting.

• *Resource constraints on mobile devices:* Our solution must be lightweight: mobile devices have limited CPU and memory, and object counts can reach tens or hundreds of millions. Even storing and updating detailed lifetime information for all objects can be prohibitively expensive. The third challenge is to ensure that, under these constraints, the overhead of our approach remains low enough for deployment on real devices.

4 Preliminary Study and Insights

To address these challenges, we first characterize lifetime-related properties of Android heaps. In this section, we analyze the object reference graph observed during GC tracing and derive three key observations.

Structural Sparsity of Object Reference Graphs. The object reference graph encodes rich lifetime information, as GC determines liveness by tracing this graph. We start by examining the in-degree and out-degree of each object. Using several applications, we instrument ART to record all references traversed during GC tracing and then compute the degree distributions (Figure 5). We find that most objects have an in-degree of exactly one. For instance, in Instagram, about 69% of objects are referenced by a single other object. Thus, the object reference graph is structurally sparse: most objects are pointed to by a single source reference, which largely determines their lifetimes.



(a) Amazon. (b) Instagram.

Figure 5: In-degree and out-degree distributions of objects.

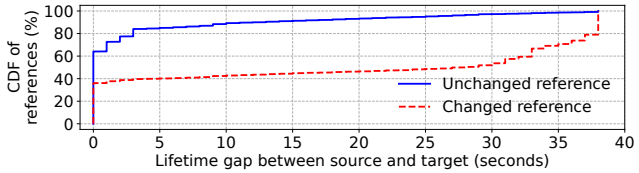


Figure 6: Distribution of lifetime gaps between source and target objects. Unchanged and changed references exhibit different patterns: object pairs connected by unchanged references tend to have small lifetime gaps, whereas those connected by changed references tend to have large gaps.

Insight 1: Strong correlation between object lifetimes and incoming references. For objects with exactly one incoming reference, their lifetime is highly likely to match that of this reference. Hence, the sparsity of the object reference graph allows us to approximate lifetime prediction as edge-lifetime prediction rather than node-lifetime prediction in Android. Edge lifetimes are often easier to model because they are driven by local update operations, whereas object liveness requires global reachability reasoning over the entire graph.

Lifetime Gap between Source and Target Objects. We next study how the lifetime of a reference relates to the lifetimes of its source and target objects. For each reference, we measure the *lifetime gap* between the two endpoints. Concretely, we launch Amazon Shopping, warm it up for 30 seconds, and trigger a GC to capture a snapshot of the current object reference graph. Over the next 38 seconds, we estimate the lifetimes of all objects that die within this window using an approximate reference-counting scheme implemented in the write barrier. During the same interval, we record whether each reference is modified. For every reference, we then compute the lifetime gap between its source and target objects and categorize the reference as *unchanged* or *changed*. Figure 6 shows the resulting distributions. The two patterns differ sharply: for unchanged references, the vast majority of gaps are near zero, indicating that source and target objects tend to have very similar lifetimes. For changed references, many object pairs exhibit large gaps close to the experiment duration: one object dies during the window while the other survives until the end. We conclude that object pairs connected by immutable references tend to co-survive, whereas pairs connected by mutable references are much more likely to have diverging lifetimes.

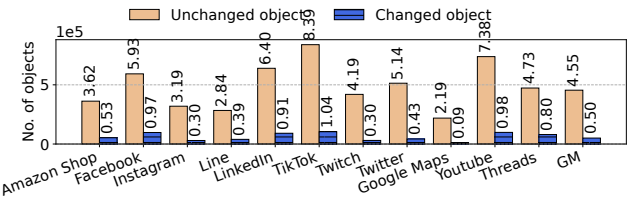


Figure 7: Distribution of unchanged and changed references across applications. GM denotes the geometric mean.

Insight 2: Subgraph-based lifetime estimation. Combining the above observations, we propose a subgraph-based lifetime estimation method. We conceptually cut the object reference graph along mutable edges, yielding subgraphs whose internal edges are all immutable. Objects within such a subgraph tend to survive and die together. We first align each object's lifetime with that of its enclosing subgraph, and then align each subgraph's lifetime with that of a memory page. Rather than predicting exact per-object lifetimes, we focus on lifetime gaps between neighboring objects: if a reference remains unchanged, the connected objects typically live and die together across user sessions. This transforms the complex problem of reasoning about a highly dynamic heap into the simpler problem of identifying and exploiting internally stable subgraphs.

Prevalence of Immutable References. To partition the object reference graph by reference mutability, we must understand how often references change. We record updates to object-reference fields in mobile applications and, over a 30-second window, count how many objects have their reference fields never modified versus modified at least once. As shown in Figure 7, across a set of popular Android applications, the number of objects whose reference fields remain unchanged far exceeds the number of objects whose fields are updated. In all applications, modified objects are much fewer than unmodified ones.

Insight 3: Mutable references are few and cheap to track. Because only a small fraction of references are actually mutated, we need to track only a small subset of fields to infer lifetimes for the entire object reference graph. This alleviates resource constraints on mobile devices: with relatively few cuts, we can partition the whole graph into subgraphs that cluster objects by lifetime.

5 Design and Implementation

5.1 Overview

Design Principles. LifeLine is a GC framework that reshapes the heap so that objects with similar lifetimes are packed onto the same pages, enabling page-granularity GC that fully exploits OS-level page operations. Guided by the empirical

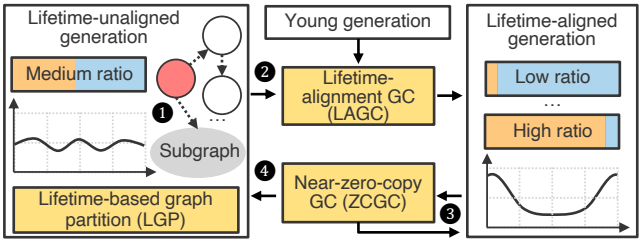


Figure 8: Overview of LifeLine.

insights in §4, it addresses the three challenges in §3.2 as follows: First, instead of predicting exact object lifetimes, LifeLine estimates *lifetime gaps* from reference changes and uses graph partitioning to form subgraphs whose objects have similar lifetimes. Second, it treats each subgraph as a stable intermediary, first aligning objects with subgraphs and then aligning subgraphs with pages. Third, to satisfy mobile resource constraints, LifeLine uses lightweight sampling to approximate reference changes, thereby avoiding expensive fine-grained per-object tracking. Moreover, to address the possibility that predictions may become stale as application behavior changes, LifeLine is designed as a closed-loop optimization framework rather than a one-shot predictor. When an app's workload shifts, the system conservatively falls back to the original object-copying mechanism and initiates a new round of lifetime alignment.

Heap Layout. As illustrated in Figure 8, LifeLine extends ART's generational heap with three logical spaces: (1) *Lifetime-aligned generation:* This is the core new space. Objects that belong to identified lifetime-correlated subgraphs are promoted into this generation. LifeLine strives to make each page in this space either almost fully occupied by objects that share similar lifetimes, making it an ideal target for page remapping or wholesale reclamation. (2) *Lifetime-unaligned generation:* Objects that cannot be reliably placed into any lifetime-correlated subgraph are promoted into this generation. (3) *Young generation:* Newly allocated objects reside in the young generation. Survivors from the young generation are promoted either to the lifetime-aligned generation or to the lifetime-unaligned generation.

Components. LifeLine consists of three components:

• *Lifetime-based graph partition (LGP):* This component partitions the object reference graph into subgraphs with high lifetime affinity based on how references are updated. LifeLine instruments the interpreter and compiler-generated code to intercept operations that update object fields, and samples these updates to identify highly mutable object fields. Using this information, it partitions the graph so that references between objects within each subgraph are mostly stable.

• *Object-page lifetime-alignment GC (LAGC):* The goal of this component is to align subgraphs with memory pages. Concretely, it is a garbage collector that reorganizes objects according to their lifetimes. During tracing, it analyzes the subgraph information produced by LGP. Then, in the compaction phase, it aligns one or more subgraphs with memory pages based on subgraph sizes and their dependency relationships. Each time it runs, it promotes objects into the lifetime-aligned generation.

• *Near-zero-copy GC (ZCGC):* This is a garbage collector that efficiently manages the memory of the lifetime-aligned generation. During the compaction phase, the collector estimates the live ratio of each page. For pages with a high survival ratio, it moves memory via page remapping, thereby avoiding physical data copying. Conversely, for pages with a low survival ratio, it reclaims them by copying objects. Pages with an intermediate survival ratio are rare and are treated similarly to sparse pages. All objects moved by copying are relocated into the lifetime-unaligned generation.

Runtime Workflow. As shown in Figure 8, LifeLine operates as follows: After an application starts, all objects are first allocated in the young generation and are then promoted to the lifetime-unaligned generation. During execution, LifeLine uses LGP to record mutability information for all objects ❶. Afterward, LifeLine uses LAGC to reorganize the heap by grouping objects with similar lifetimes onto the same pages and migrating them into the lifetime-aligned space ❷. In the lifetime-aligned space, LifeLine uses ZCGC to manage memory. For pages with a high survival ratio, ZCGC adopts an efficient page-remapping scheme ❸. For pages with a low or medium survival ratio, ZCGC uses an object-copying scheme to evacuate the page and relocate the surviving objects into the lifetime-unaligned generation ❹. As the application continues to run, new objects keep moving into the unaligned generation, and old objects keep dying in the aligned generation, so ZCGC also monitors these changes. If many objects accumulate in the unaligned generation, it triggers a new round of LAGC to reshape the page distribution and migrate objects into the aligned generation again.

5.2 Lifetime-based Graph Partition

LifeLine proposes a lifetime-based graph partition (LGP) to identify subgraphs of objects that tend to live and die together. As shown in Figure 9, the LGP procedure mainly consists of two steps. First, it modifies the execution engine of the runtime system to capture write operations to objects, and inserts small code snippets into the execution paths of these write operations so that the program executes a write-barrier function. Second, inside this write-barrier function, the captured object reference fields are recorded into a layered Bloom filter. This layered Bloom filter records all captured information about modified fields and uses it as the basis for partitioning the graph into subgraphs.

Write Interposition and Sampling. To record mutability of object fields, LGP inserts a small write barrier at object-field write sites in ART's execution engine. Specifically, the interpreter extends the bytecode handlers for instructions that

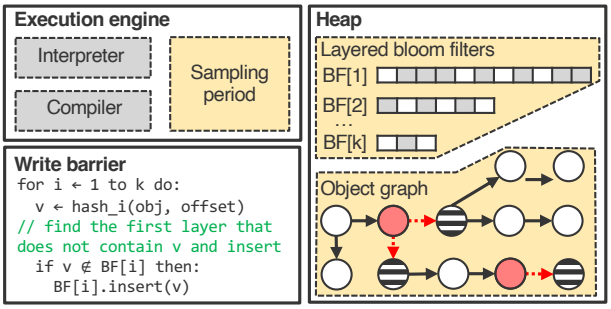


Figure 9: Lifetime-based graph partition.

store references into object fields. The JIT/AOT compiler inlines a few instructions along the object write path. Whenever the application executes a write operation, the operation can be intercepted and the execution flow redirected to the write-barrier function.

However, naively hooking write operations may introduce extra overhead because writes occur frequently, especially inside loops. To manage the overhead incurred by write interposition, LGP employs thread-based periodic sampling. It maintains a counter variable in thread-local storage for each thread and a sampling-period constant. Before calling the write-barrier function, it increments the thread-local counter and checks whether it has reached the period. If it has not reached the period, execution continues without calling the write barrier. Thus, it executes the write barrier for only a subset of write operations, while the remaining writes pay only the cost of a single inexpensive thread-local variable increment. Therefore, the overhead of write interposition remains low even for update-intensive applications.

Tracking Mutations with Layered Bloom Filters. At each sampling period, the program thread invokes a write barrier that records the modified object field and the number of times it has been updated. To maintain this information efficiently, LGP employs a layered Bloom filter [15, 19, 23, 24, 44]. Layered Bloom filters are commonly used for approximate counting in heavy-hitter detection [28, 51]. Here, we use them to maintain approximate write statistics for each object field.

Concretely, we allocate k Bloom filters, $BF[1..k]$, all initially empty. During each sampling operation, the write barrier receives the modified object and the offset of the updated field. The pair $(object, offset)$ uniquely identifies the modified object field. We use this pair as input to compute a hash value and insert the hash into the first Bloom filter that does not yet contain it. In this scheme, the number of layers containing a hash represents the number of sampled mutations. If an object field is present in the i-th Bloom filter, it has been modified at least i times. For example, a field whose hash appears only in $BF[1]$ has been observed once; if it appears in both $BF[1]$ and $BF[2]$, it has been observed at least twice, and so on.

LGP derives a simple mutation summary for any given object field from the layered Bloom filters: fields whose hash either does not appear are treated as mostly immutable owners. Their outgoing references are treated as candidates for lifetime-correlated edges. In contrast, fields whose hashes reach deeper layers (e.g., $BF[3]$ or beyond) are classified as mutable. Their outgoing references are likely to cross lifetime boundaries and thus become subgraph cut edges. This design keeps overhead low, requiring no explicit per-object counters, and amortizes mutation-tracking cost across inexpensive hash computations and bit operations on a small number of filters.

Graph Partition. Based on the field-level mutation information recorded in the Bloom filters, LGP partitions the whole graph into multiple subgraphs by cutting along mutable references. Using different layers of Bloom filters corresponds to applying different mutation thresholds, which in turn leads to different partitioning results. As shown in Figure 10, when a small partition threshold is used (e.g., information from the first Bloom filter), most of the resulting subgraphs are relatively small. In contrast, when a larger partition threshold is applied (i.e., information from higher-layer Bloom filters), the fraction of larger subgraphs increases. Nevertheless, for all thresholds, the distribution of subgraph sizes remains highly skewed: some subgraphs exceed the page size, while others are much smaller. In LifeLine, the partition threshold is set to 3, as this choice makes most subgraphs close to the page size while requiring only three Bloom filters, thus keeping the memory overhead relatively low.

Robustness of Mutability-based Prediction. Reference mutability provides strong robustness because it captures a relatively stable property of the program, rather than attempting to predict the exact moment at which an object becomes unreachable. For example, fields that are repeatedly overwritten often link a long-lived container to a succession of short-lived contents. This pattern is common in mobile applications: fields that remain stable after initialization typically connect objects that are owned and reclaimed together, such as a view hierarchy, a decoded content object, or a per-session data structure. Therefore, LifeLine's lifetime analysis focuses on distinguishing rarely modified stable edges from frequently modified boundary edges, instead of predicting the precise lifetime of each individual object; the latter is inherently difficult and highly dependent on user behavior. When runtime behavior changes, LifeLine's mutability-based lifetime-boundary prediction remains robust because it relies on a relatively stable property of mobile programs.

Although LifeLine's lifetime-prediction method is robust in most cases, significant changes in runtime behavior may still reduce the stability of lifetime alignment and decrease the number of pages that can be optimized through remapping. Such cases include abrupt changes in application behavior, such as navigating to an entirely different page, or substantial short-term fluctuations in user interaction patterns. In the worst case, LifeLine gradually falls back to the behavior of the underlying baseline copying collector, while incurring only bounded additional overhead from extra alignment operations

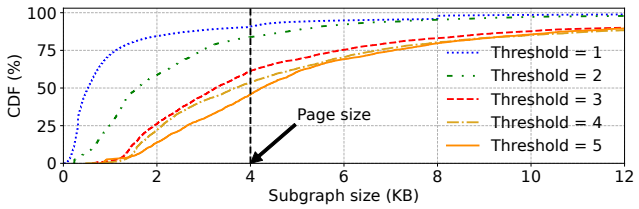


Figure 10: Distribution of subgraph sizes. This distribution is directly influenced by the partition threshold. A smaller threshold yields more small subgraphs, while increasing the threshold raises the proportion of large subgraphs.

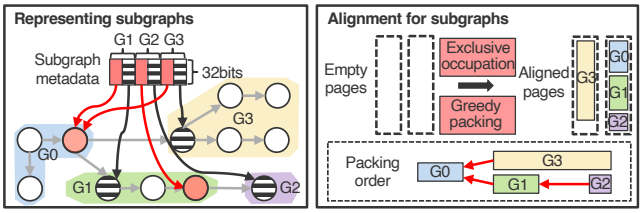


Figure 11: Object–page lifetime alignment GC.

and temporary metadata maintenance.

5.3 Object–Page Lifetime Alignment GC

After partitioning the object reference graph into multiple subgraphs, LifeLine's goal is to reshape the heap so that each page is either almost entirely live or almost entirely dead. To achieve this, LifeLine employs an object–page lifetime alignment GC (LAGC), as shown in Figure 11, which performs a full moving collection over the heap. After this collection, the mature heap exhibits a bimodal distribution of per-page survival ratios. This layout allows subsequent GCs to replace most object copying with inexpensive page remapping.

Representing Subgraphs. To reorganize the heap, LAGC first traces the entire object reference graph, as in conventional GCs, to obtain both liveness information and subgraph metadata. This metadata summarizes the partitioning of the graph according to LGP's records and the relationships among subgraphs. The metadata has three components: (1) It assigns each subgraph a 32-bit identifier, defined as the address of its root object. Using this identifier, LifeLine can locate all objects in the subgraph along the tracing order. (2) It maintains another 32-bit value per subgraph to represent ownership between subgraphs. For example, if a highly mutable edge connects subgraphs G0 and G3, LifeLine designates G0 as the logical owner of G3 and records the host object address of such mutable edges in this ownership field. Using this field, LifeLine can determine the owner of any subgraph, conservatively choosing the first candidate encountered during tracing when multiple owners exist. (3) It stores the total size of each subgraph in bytes, which the collector uses to align subgraphs with pages.

Constructing Metadata during Marking. During the marking phase, LAGC constructs subgraph metadata on the fly. As the GC thread traverses and marks the object reference graph, it checks for each field whether its referent is recorded in the k-th-layer Bloom filter. If so, LifeLine creates a new subgraph entry, uses this object as the subgraph identifier, writes it into the subgraph metadata as a new subgraph, initializes the subgraph size to 0, and pushes the referent onto the tracing stack.

During subsequent traversal, when an object is popped from the stack, the GC thread inspects the current subgraph metadata. If the object represents a subgraph identifier stored in the metadata, it is recognized as a subgraph root, and LifeLine immediately begins counting the size of this subgraph. Due to the depth-first traversal of the GC, all objects scanned thereafter are treated as belonging to this subgraph until another subgraph root is encountered or the stack backtracks past the root's position. For each such object, LifeLine increments the subgraph size and updates the current subgraph metadata.

Finally, LifeLine completes all subgraph metadata setup within this single, mostly concurrent marking pass, with minimal interference to the mutator. The resulting metadata induces an approximate but stable ownership relation over subgraphs. In practice, since ownership is predominantly singular (§4), this relation is sparse rather than dense, keeping the subsequent alignment phase simple and inexpensive.

Align Large Subgraphs: Exclusive Occupation. During compaction, LAGC uses the subgraph metadata to align subgraphs with pages. As shown in Figure 10, subgraph sizes are highly skewed: many are close to or larger than a page, while others are much smaller. LifeLine therefore classifies subgraphs using a size threshold (4 KB by default) and applies different strategies to large and small subgraphs.

For large subgraphs, LifeLine uses an exclusive-allocation policy: each large subgraph occupies one or more pages by itself. When computing target addresses, LAGC proceeds in two steps. (1) It first assigns a page-aligned starting address to the subgraph's root. (2) It then assigns contiguous addresses within those pages to all objects in the subgraph, traversing them in the same DFS order used during tracing. Allocation in the to-space is monotonically increasing. During this traversal, LifeLine accumulates the total size of allocated objects and stops when this total reaches the subgraph size recorded in the metadata or when it encounters the root of another subgraph. This procedure packs all objects in a large subgraph into a contiguous set of pages, so those pages are dedicated to objects with similar lifetimes. It also sidesteps complex binning decisions for very large objects, keeping the alignment algorithm simple and predictable.

Align Small Subgraphs: Greedy Packing. Given the dependency subgraph metadata, LAGC treats each subgraph as an indivisible item and each lifetime-aligned page as a bin, greedily co-locating related subgraphs. The packing algorithm has

two steps. First, ownership-based aggregation. For a small subgraph, LAGC follows ownership edges using the subgraph metadata: it finds the parents on which this subgraph depends, identifies the head of each parent's subgraph, then continues to their owners, and so on. All subgraphs visited during this reverse traversal are tentatively grouped as candidates for the same page, while LAGC tracks their cumulative size. When this total reaches the page-size threshold, the aggregation stops. Second, relocation by guided traversal. Starting from the highest-level owner discovered above, LAGC follows its outgoing references to enumerate all grouped subgraphs. This owner is placed first on the lifetime-aligned page. LAGC then performs a traversal pass similar to that of a normal GC, but with the crucial advantage that it already knows exactly which subgraphs can be merged. As it starts from the top-level owner, it lays out the subgraphs contiguously within the page, so siblings and nearby descendants naturally end up co-located.

This greedy, ownership-aware packing (1) increases the likelihood that objects on the same page have correlated lifetimes, (2) keeps pages tightly packed, since most clusters nearly fill a page, and (3) remains lightweight by maintaining lifetime relationships at subgraph granularity rather than object granularity.

Progressive Alignment across GCs. LifeLine does not require object–page alignment to be completed in a single GC cycle. Large heaps, in particular, may require multiple LAGC runs. As shown in Figure 8, LifeLine maintains two mature generations: one lifetime-aligned and one lifetime-unaligned. Each LAGC compacts only the unaligned generation, incrementally migrating objects into the aligned generation. This staged process helps stabilize the system during alignment.

5.4 Near-Zero-Copy GC

To exploit the bimodal per-page liveness created by LAGC, the near-zero-copy GC (ZCGC) minimizes data movement in the lifetime-aligned mature generation. It runs after the global marking phase of a major collection, when live objects in this generation are already known. As shown in Figure 12, a movement planner cooperates with the OS to decide, on a per-page basis, whether to migrate a page via zero-copy remapping or to compact it by copying surviving objects.

Per-Page Survival Ratio Estimation. Scanning all objects on a page would defeat the goal of fast compaction. Instead, the planner reuses statistics collected during the GC's marking phase. During tracing, the GC accumulates the total size of live objects for each page. The planner then simply divides this live-byte count by the page size to compute the page's survival ratio. Because this accounting piggybacks on the existing marking pass, it yields a robust estimate at essentially negligible additional cost.

Hybrid Page-Remap-or-Copy Strategy. Given the estimated live ratio, the planner chooses between two movement

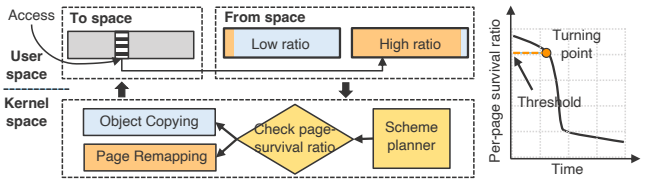


Figure 12: Near-zero-copy GC. It features an efficient hybrid scheme that combines page remapping with object copying.

schemes using two configurable thresholds:

• *Page remapping for dense pages.* If a page is almost entirely live (e.g., a live ratio above a high threshold such as 90%), copying would waste CPU cycles while reclaiming little space. For such pages, the planner uses `userfaultfd` with `UFFDIO_MOVE` to atomically remap the physical page from from-space to to-space. This operation updates only the page-table entries without modifying the underlying data, so the cost per page is effectively constant, while avoiding frequent copying of many small objects.

• *Object copying for sparse and medium-density pages.* If a page has a low live ratio, only a few objects survive. The planner then uses a copy-based scheme: the GC precomputes new locations for live objects on the page, and when an application thread later faults on the corresponding to-space page via `userfaultfd`, the handler copies only these survivors, leaving the rest of the page free for reclamation. Pages whose live ratio falls between the low and high thresholds are also handled by this copy path. Thanks to LAGC, such medium-density pages are rare and do not dominate the cost.

The thresholds correspond to the turning point in Figure 12, where the costs of remapping and copying intersect. Below this point, pages quickly become low-survival because co-located objects tend to die together, making copying cheaper. In our implementation, the thresholds are fixed per deployment but can be tuned using a simple cost model.

Adaptive Re-alignment Trigger. Over time, workload phases may change and the original alignment between subgraphs and pages can drift, causing objects to be repeatedly moved back into the lifetime-unaligned generation while new objects accumulate there. Consequently, the size of the aligned generation S_{align} shrinks and the unaligned generation $S_{unalign}$ grows. ZCGC monitors these sizes and checks:

$$\frac{S_{align0}}{S_{unalign0}} - \frac{S_{align}}{S_{unalign}} > \Delta \times \frac{S_{align0}}{S_{unalign0}} \quad (1)$$

Here, S_{align0} and $S_{unalign0}$ are the aligned and unaligned generation sizes at the first ZCGC after LAGC, and Δ controls when LAGC is re-triggered. When the inequality holds, it indicates significant misalignment, and a new LAGC is invoked.

Fallback and Maintainability. LifeLine preserves the correctness of a conventional tracing collector: reference-mutability information only guides placement, while live-

ness remains determined by normal marking. If phases shift, fields are misclassified, or pages contain mixed-lifetime objects, ZCGC falls back to treating pages as sparse or medium-density and relocates survivors through object copying. Thus, inaccurate prediction may reduce remapping opportunities, but cannot reclaim live objects or expose untraced objects to the mutator. LifeLine also remains maintainable: it adds metadata construction and page-aware placement to ART's GC pipeline, while reusing existing tracing and fault-based relocation mechanisms. Its extra runtime state is temporary and localized to alignment phases, making the design easy to disable, tune, or bypass for workloads that do not benefit from lifetime alignment.

6 Evaluation

We evaluate LifeLine on real hardware and mainstream Android applications, aiming to answer: (1) What is its end-to-end impact on GC work and user-visible performance? (§6.2) (2) What are the underlying causes of these effects? (§6.3) (3) What CPU and memory overheads does it introduce? (§6.4) (4) How sensitive is LifeLine to key parameters? (§6.5)

6.1 Experimental Setup

Comparisons. We compare LifeLine against all state-of-the-art production collectors in the Android runtime: (1) *Concurrent Mark-Compact (CMC)* [1, 26]. This has been the default and recommended collector since Android 13. It relies on a userfaultfd-based fault-and-copy compaction mechanism and, in our configuration, already incorporates OS-level assistance. (2) *Concurrent Copying (CC)* [5]. This was the default collector in earlier Android versions and is still widely deployed. It uses a Baker-style read barrier and performs all compaction via object-level copying. (3) *Semi-space (SS)* [49]. This collector is used during application startup in all Android versions and is a classic stop-the-world copying collector. These three collectors cover the deployed design space of object-copying and OS-assisted GC mechanisms. For LifeLine, we evaluate two LAGC-related configurations: (1) LifeLine (large), which aligns only large subgraphs with memory pages while ignoring small subgraphs; (2) LifeLine (large+small), which aligns both large and small subgraphs. Unless otherwise specified, LifeLine refers to the configuration that aligns both large and small subgraphs. Table 1 lists the key parameters used in LifeLine.

Workloads. Our workload consists of several highly popular Google Play Store applications that represent typical mobile use cases, including Amazon Shopping, Facebook, Instagram, Spotify, LinkedIn, TikTok, Twitch, Twitter (X), Threads, Telegram, Line, YouTube, and Google Maps. For each application, during experiments, we log in to a pre-configured account, wait for the application to finish initialization (10 seconds after launch), connect to a stable Wi-Fi network, and then drive the UI using a repeatable interaction script whose main procedure is to swipe the screen once every 0.4 seconds, covering scrolling and content-loading scenarios. We choose this interaction because it is repeatable across all applications, captures common mobile workloads, and continuously stresses allocation and object reclamation. This methodology also has limitations: it does not cover all application behaviors, such as randomly changing user interactions or short sessions with frequent foreground/background transitions. Nevertheless, the scrolling workflow we use represents the dominant interactive workload in most mobile usage scenarios and therefore provides a practical basis for evaluating performance under common real-world interactions.

Table 1: Summary of parameters used by LifeLine.

Symbol	Component	Semantic	Setting
N_B	LGP	Graph partition threshold	3
M_B	LGP	Sampling period	100
T_G	LAGC	Large subgraph threshold	4 KB
T_M	ZCGC	Page-survival ratio threshold	90%
Δ	ZCGC	Trigger for next alignment GC	30%

Testbed. Experiments run on a Google Pixel 7 Pro smartphone with a 120 Hz display, an octa-core CPU (two 2.85 GHz Cortex-X1, two 2.35 GHz Cortex-A78, and four 1.80 GHz Cortex-A55 cores), a 10-core Mali-G710 MP7 GPU, 12 GB of 3200 MHz LPDDR5, and 128 GB flash. The runtime is Android Open Source Project (AOSP) android-15.0.0_r3. The OS is a Generic Kernel Image (GKI) with the android-gs-pantah-5.10-android15-qpr1 kernel. This kernel does not support page remapping by default; we enable it by porting the corresponding patch [3].

Implementation. We implement LifeLine [1] on top of AOSP and its corresponding GKI kernel. Within AOSP, our modifications are mainly in the ART module, totaling approximately 2.3K lines of C++ code, organized as follows: First, for the GC mechanism, LifeLine introduces two new collector configurations, LAGC and ZCGC, by extending the existing CMC GC (about 1.5K lines). Second, for LGP, we implement the core layered Bloom filter and integrate it into the heap (about 200 lines). LGP also adds write-barrier sampling in both the interpreter and compiler, along with a sampling counter in thread-local storage (about 100 lines). Third, for the heap, we adjust the heap layout by introducing a lifetime-aligned generation while reusing the original bump-pointer space as the lifetime-unaligned generation (about 200 lines). The remaining modifications in ART consist mainly of glue code for configuration, statistics collection, and integration with the existing collector-management infrastructure. In the kernel, we add roughly 100 lines of C code, primarily modifying the `userfaultfd` and page-migration paths.

[1] The source code of LifeLine is available at https://github.com/jiachengh/LifeLine.

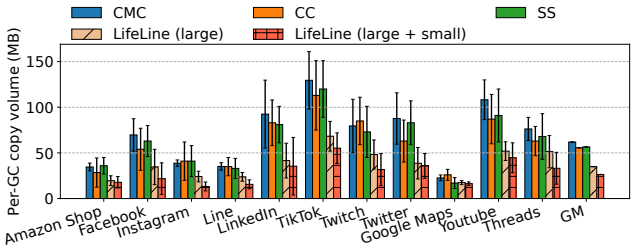
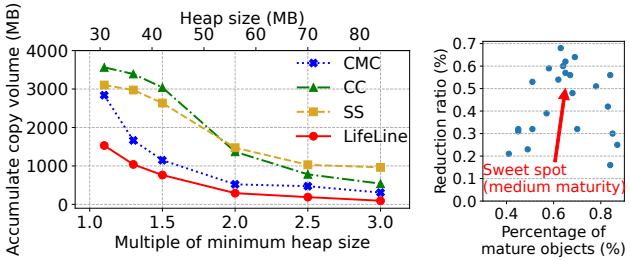


Figure 13: Average per-GC copy volume across applications.



(a) Copy volume across heap sizes. (b) Mature ratio.

Figure 14: Analysis of factors affecting GC copy volume.

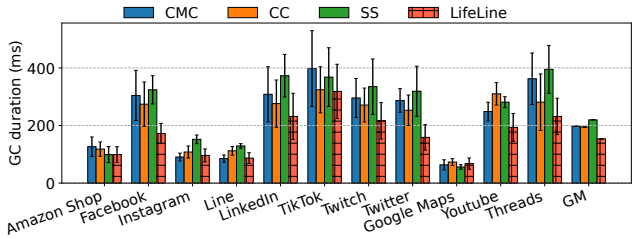
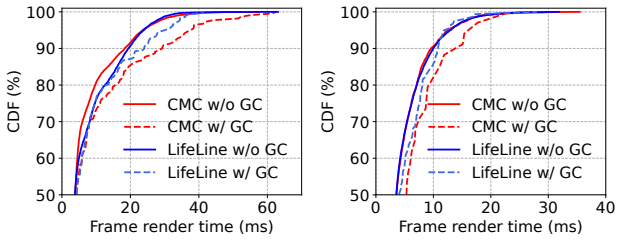


Figure 15: GC duration across applications.



(a) Twitter. (b) Instagram.

Figure 16: Frame render time distributions.

6.2 Overall Performance

Copy Volume. To quantify how replacing object copying with page remapping reduces memory-movement overhead, we measure the GC copy volume per cycle for all collectors. Each application runs a 30 s warm-up phase, after which we record the bytes copied per GC. The warm-up excludes application launch, account restoration, and initial content fetching, and gives all collectors a comparably mature heap before measurement. We fix the scrolling rate across experiments, collect 10 GCs per application, and report the mean and variance. Figure 13 shows the total data copied per GC. LifeLine substantially reduces physical copying: on average, copy volume drops by 57.4%, from 61.9 MB to 26.4 MB. LifeLine replaces many object copies with page remapping and shrinks both the number and size of the remaining copies. Compared to CMC, LifeLine exploits OS-level page remapping more aggressively, whereas CMC still performs object-level copying for most movements. Overall, LifeLine achieves effective compaction with far fewer memory-copy operations.

LifeLine's reduction varies across and within applications, so we next examine what drives its effectiveness. We first study how heap size (normalized to the SS collector's minimum heap size) affects copy volume (Figure 14a). Across heap sizes, LifeLine consistently copies less than the other collectors, and its advantage widens as the heap shrinks, indicating greater benefits under tighter memory constraints.

We then analyze the impact of the mature-object fraction using Amazon Shopping as a case study. Figure 14b correlates LifeLine's reduction in copy volume relative to CMC with the fraction of objects in the mature generation. As the mature fraction increases, LifeLine's benefit first grows, peaking at moderate maturity (e.g., at 65% mature objects, it cuts copy volume by about 60% on average), and then declines once mature objects dominate. Beyond roughly 80% mature objects, CMC can also leverage OS-level page remapping effectively, narrowing LifeLine's relative advantage. Thus, LifeLine's benefit is strongly tied to the proportion of mature objects and is maximized at intermediate mature fractions.

GC Duration. GC duration is critical on mobile devices because longer GCs increase the risk of blocking user interactions. We reuse the methodology above but now measure GC execution time. Figure 15 reports the average duration and variance for each GC method. LifeLine consistently shortens GC duration in most applications. On average, it reduces GC time from 198 ms (CMC) to 153 ms, a 22.7% reduction. The improvement varies with each application's workload. First, the reduction in duration closely tracks the reduction in copy volume: the more copies LifeLine avoids, the more GC time it saves. Second, applications with small heaps (e.g., Google Maps) already have short GCs, so even large percentage reductions translate into smaller absolute gains.

Frame Rendering Performance. For Android applications, frame-rendering smoothness is a primary user-visible metric. Interactive workloads dominated by touch and scrolling are especially sensitive to GC-induced stalls. We use Twitter and Instagram as case studies and measure frame-rendering-time distributions using the `gfxinfo` tool, capping heap size at 80 MB (roughly 2–3× their minimum heap). Figure 16 shows the cumulative distribution function (CDF) of frame latency for Twitter and Instagram, separating intervals with and without GC. Under CMC, GC causes a clear rightward shift in

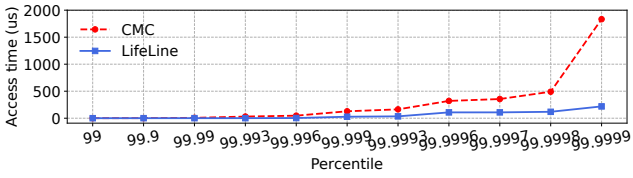
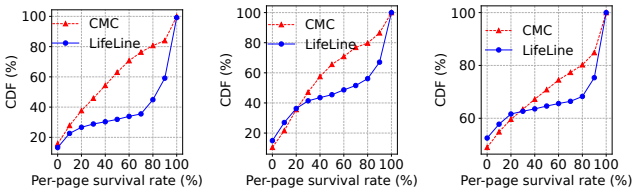


Figure 17: Tail memory access latency during GC.



(a) Amazon. (b) Twitter. (c) Instagram.

Figure 18: Per-page survival-rate distribution.

the tail; for example, the 90th-percentile latency is around 30 ms, leading to noticeable stutters. LifeLine significantly improves tail latency when GC occurs. For Instagram, it reduces the 90th-percentile latency by about 29% relative to CMC. Overall, by reducing memory copying, LifeLine alleviates GC-induced stalls, leading to smoother scrolling and a more responsive user interface.

6.3 Benefit Breakdown

We now decompose LifeLine's overall gains into lower-level effects that explain how it reduces stalls and copying overhead.

Tail Memory Access Latency. As discussed in §3.1, memory copies on the GC critical path can inflate tail memory-access latency. To quantify LifeLine's effect, we use Amazon Shopping as a case study and sample read accesses during compaction. We record every 100th access, treating the time required to read a field from an object as the memory-access latency. Figure 17 shows the tail distribution for LifeLine and CMC. LifeLine substantially shrinks the tail. At the 1-ppm tail, it reduces latency by 85% compared to CMC. This improvement arises because LifeLine avoids large object copies on page faults and instead relies on zero-copy page remapping, eliminating many stalls on the GC critical path.

Accuracy of Object–Page Lifetime Alignment. In §3.1, we identified object–page lifetime mismatch as the main reason existing Android collectors struggle to cooperate with the OS. To assess how well LifeLine addresses this issue, we examine the per-page survival ratios at the next GC after alignment. Figure 18 shows that LifeLine reshapes the per-page survival distribution compared to CMC. Under CMC, survival ratios are roughly uniform, with many pages in the 20–80% range. Under LifeLine, the CDF over 20–80% is relatively flat, with

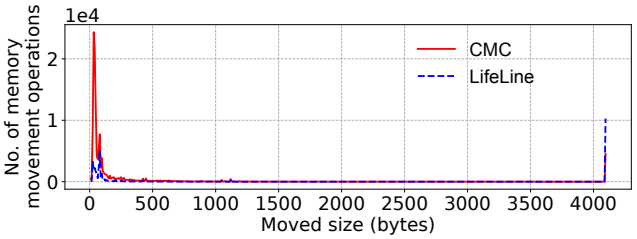


Figure 19: Detailed profile of memory movement.

steep jumps around 10% and 90%, meaning that most pages are either almost empty or almost full. LifeLine thus converts a near-random distribution into a bimodal one, aligning object lifetimes with page lifetimes.

The accuracy of this alignment is also robust to imperfect lifetime prediction. LifeLine does not require exact object lifetimes: it only needs enough correlation to make most pages clearly sparse or dense. When LGP occasionally groups objects whose lifetimes later diverge, the consequence is a medium-survival page rather than an incorrect reclamation decision. ZCGC handles such pages conservatively through the copy path, so inaccurate alignment degrades the opportunity for zero-copy remapping but does not compromise correctness. As objects on poorly aligned pages are copied back to the unaligned generation and newly promoted objects accumulate there, the aligned-to-unaligned generation ratio drops and triggers a new LAGC. Thus, alignment accuracy is self-correcting across GC cycles: transient prediction errors mainly reduce the immediate remapping benefit, while sustained workload changes are absorbed by re-alignment.

Distribution of Memory-Movement Granularity. Given such alignment, LifeLine can perform GC largely at page granularity instead of object granularity. To study this, we measure the size of each memory-movement operation during a single GC for Amazon Shopping. Figure 19 shows that CMC performs many small object moves (e.g., 0–100 bytes), each requiring explicit copying. LifeLine greatly reduces such tiny moves and increases page-sized (4 KB) movements, which are handled efficiently via OS page remapping. Once object and page lifetimes are aligned, LifeLine enables page-granularity GC and tighter GC-OS cooperation.

6.4 Overhead Analysis

Memory Overhead. LifeLine's memory overhead comes primarily from two sources: (1) LGP's layered Bloom filters and (2) LAGC's subgraph metadata. A three-level Bloom-filter design is sufficient in most cases (see §6.5). In our implementation, the first layer uses a 2 MB filter, while the second and third layers use 0.5 MB each, keeping the false-positive rate below 1%. Higher layers store less data and therefore require smaller filters. For each subgraph, LAGC stores two 4-byte pointers and one 4-byte size field, introducing only minor

memory overhead. For example, storing metadata for 5000 subgraphs requires only 59 KB. Overall, LifeLine adds less than 4 MB of auxiliary memory. Moreover, these auxiliary structures are dynamically allocated and used only during lifetime alignment. After alignment, LifeLine switches to near-zero-copy GC and releases these structures. Therefore, its memory overhead is modest relative to the performance gains.

CPU Overhead. Different LifeLine components affect CPU usage in distinct ways. Using System Tracing [6] and Perfetto [8], we profile CPU time per component. ZCGC reduces GC duration, decreasing GC-thread CPU time by 32% compared to CMC. The additional CPU work comes from LGP and LAGC. LGP tracks field mutability by interposing on object-reference writes and consulting the per-thread sampling counter. Because this code executes in the mutator, its cost depends on write frequency; however, the sampling period limits how often the full barrier records a field update. In our measurements, this write-sampling path increases average CPU time by roughly 3.8%. During LAGC, the collector maintains subgraph metadata while marking, identifying subgraph roots and recording ownership and size information. LAGC performs page-aware placement during compaction, assigning large subgraphs to page-aligned regions and packing small, related subgraphs together so that subsequent ZCGC cycles can efficiently remap pages. Metadata maintenance and page-aware placement together add approximately 12.5% CPU time relative to CMC during alignment. Notably, these expensive operations occur only during alignment phases, while steady-state ZCGC leverages the resulting layout and avoids much of CMC's copy work. For example, during a 3-minute Instagram run, LifeLine's average CPU overhead during LAGC is about 2.7% over CMC, which we consider modest given the reductions in GC copying and the improvements in user-visible performance.

6.5 Sensitivity Analysis

Table 1 summarizes the parameters of LifeLine. In this section, we examine how these parameters affect the system. Among them, the number of Bloom-filter levels, N_B, is a critical parameter, as it determines the quality of subgraph generation. We vary N_B from 1 to 5 (see Figure 10). A small N_B (e.g., 1) produces overly fine-grained subgraphs, which in turn multiplies the overhead of LAGC and makes it inefficient. Conversely, a large N_B (e.g., 5) mixes objects with different lifetimes and also introduces unnecessary Bloom-filter sampling and storage overhead, reducing the efficiency of ZCGC. We therefore set $N_B = 3$, which yields subgraphs whose sizes and lifetimes align well with page granularity while incurring relatively low overhead.

We similarly sweep the remaining parameters. If M_B is too small, CPU overhead rises sharply; if it is too large, objects with very different lifetimes are grouped together. For T_G, values below 4 KB create placement gaps for large subgraphs and waste space; since page size is 4 KB, we cap T_G at 4 KB. If T_M is too small, many well-aligned pages are unnecessarily migrated back to the unaligned generation; if it is too large, garbage is reclaimed too late. A 90% threshold works well in practice. Finally, if Δ is too small, LAGC runs too frequently; if it is too large, too many objects remain unaligned and LifeLine's behavior converges toward CMC.

6.6 Discussion on Generalizability

We focus on ART because, to our knowledge, it is the only mainstream open-source runtime whose GC is explicitly co-designed with the OS. Other major runtimes, such as the server-side JVM [7], .NET [43], and JavaScript engines [29,45], currently lack CMC-like OS-cooperative collectors, likely due to cross-platform and engineering constraints. Consequently, we evaluate LifeLine only on Android. LifeLine can, in principle, be integrated into other runtimes to enable stronger GC-OS cooperation and unlock optimizations unavailable to a standalone runtime.

7 Related Work

Garbage Collection. Automatic memory management is a cornerstone of modern managed runtimes [16, 18, 36, 42, 54, 58]. Prior work optimizes GC along multiple dimensions, including pause time [21, 25, 32, 63], throughput [41, 59, 68], fragmentation [11, 13], and memory footprint [32]. LifeLine belongs to this space but focuses on reducing unnecessary GC work through fine-grained lifetime prediction.

Lifetime-based strategies are central to generational collectors [14, 17, 55], which assign objects to a small number of generations based on expected remaining lifetimes. From the OS perspective, however, these schemes are coarse-grained: each generation exposes only a few categories, and pages within a generation may still mix short- and long-lived objects. LifeLine adopts a finer-grained mature-space layout, where each generation corresponds to a single page, aligning object lifetimes with page boundaries. This page-aligned organization improves GC efficiency and, as we demonstrate, reduces jank in Android applications. In this sense, LifeLine can be viewed as a page-level refinement of generational collection for runtimes that already support OS-assisted movement, rather than a competing policy for young-object management. Beyond general-purpose collectors, specialized schemes address disaggregated memory [41, 56, 57], non-volatile memory [40, 50, 60], and large-scale data-processing frameworks [27, 46]. Similarly, LifeLine is designed for mobile runtimes and specifically addresses GC-induced jank in Android deployments.

Cross-layer Memory Management. Recent GC–OS co-design efforts [2, 26] leverage kernel mechanisms [4] to reduce jank, lower energy consumption, and improve perfor-

mance. However, these approaches are fundamentally limited by the mismatch between object lifetimes and page layouts, constraining the benefits of page remapping and reclamation. LifeLine directly addresses this mismatch by producing lifetime-homogeneous pages, enabling more effective GC–OS cooperation. Other systems, such as Marvin [39] and Fleet [33], coordinate GC with the virtual memory subsystem to improve memory efficiency and hot-launch behavior for background applications. This line of work is largely orthogonal to LifeLine. By organizing objects into page-aligned lifetime classes, LifeLine can complement these GC–OS policies, enhancing both foreground GC performance and background-oriented reclamation and compaction.

8 Conclusion

In this paper, we revisit the copying overhead caused by GC in mobile applications and reveal a fundamental lifetime mismatch between objects and memory pages. LifeLine employs a graph-based lifetime alignment mechanism to cluster co-surviving objects onto the same pages, enabling reduced GC overhead and improving overall performance. Although our current prototype targets ART on Android phones, the core idea of aligning object and page lifetimes naturally extends to other managed runtimes and hardware platforms, paving the way for more efficient OS-assisted GC.

Acknowledgments

We thank the anonymous reviewers and our shepherd for their valuable and constructive feedback, which significantly improved this paper. This work was supported in part by the Research Grants Council of Hong Kong (No. 11216925), and by City University of Hong Kong internal and donation fundings (No. 9610598, No. 9220148, and No. 7005991). This work was also supported by OPPO Research Funding.

References

[1] Android 13 is in aosp!, 2022. https://android-developers.googleblog.com/2022/08/android-13-is-in-aosp.html [Accessed: 20-Nov-2025].

[2] Rfc for new feature to move pages from one vma to another without split, 2023. https://lore.kernel.org/linux-mm/CA+EESO4uO84SSnBhArH4HvLNhaUQ5nZKNKXqxRCyjniNVjp0Aw@mail.gmail.com/ [Accessed: 20-Nov-2025].

[3] userfaultfd move option, 2023. https://lwn.net/Articles/952319/ [Accessed: 20-Nov-2025].

[4] userfaultfd: Uffdio_move uabi, 2023. https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=adef440691ba [Accessed: 20-Nov-2025].

[5] Android 8.0 art improvements, 2025. https://source.android.com/docs/core/runtime/improvements/ [Accessed: 20-Nov-2025].

[6] Capture a system trace on a device, 2025. https://developer.android.com/topic/performance/tracing/on-device/ [Accessed: 20-Nov-2025].

[7] The hotspot group, 2025. https://openjdk.org/groups/hotspot/ [Accessed: 20-Nov-2025].

[8] Perfetto - system profiling, app tracing and trace analysis, 2025. https://perfetto.dev/ [Accessed: 20-Nov-2025].

[9] Android phone statistics by best technology, 2026. https://scoop.market.us/android-phones-statistics/ [Accessed: 14-May-2026].

[10] Google play store statistics 2026: Key data every app business should know, 2026. https://www.apptunix.com/blog/google-play-store-statistics/ [Accessed: 01-Jun-2026].

[11] David F. Bacon, Perry Cheng, and V. T. Rajan. Controlling fragmentation and space consumption in the metronome, a real-time garbage collector for java. In Frank Mueller and Ulrich Kremer, editors, *Proceedings of the 2003 Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES'03). San Diego, California, USA, June 11-13, 2003*, pages 81–92. ACM, 2003.

[12] Stephen M. Blackburn, Perry Cheng, and Kathryn S. McKinley. Myths and realities: the performance impact of garbage collection. In Edward G. Coffman Jr., Zhen Liu, and Arif Merchant, editors, *Proceedings of the International Conference on Measurements and Modeling of Computer Systems, SIGMETRICS 2004, June 10-14, 2004, New York, NY, USA*, pages 25–36. ACM, 2004.

[13] Stephen M. Blackburn and Kathryn S. McKinley. Immix: a mark-region garbage collector with space efficiency, fast collection, and mutator performance. In Rajiv Gupta and Saman P. Amarasinghe, editors, *Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation, Tucson, AZ, USA, June 7-13, 2008*, pages 22–32. ACM, 2008.

[14] Stephen M. Blackburn, Sharad Singhai, Matthew Hertz, Kathryn S. McKinley, and J. Eliot B. Moss. Pretenuring for java. In Linda M. Northrop and John M. Vlissides, editors, *Proceedings of the 2001 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications, OOPSLA 2001, Tampa, Florida, USA, October 14-18, 2001*, pages 342–352. ACM, 2001.

[15] Burton H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM*, 13(7):422–426, 1970.

[16] Hans-Juergen Boehm and Mark D. Weiser. Garbage collection in an uncooperative environment. *Softw. Pract. Exp.*, 18(9):807–820, 1988.

[17] Rodrigo Bruno, Luís Picciochi Oliveira, and Paulo Ferreira. NG2C: pretenuring n-generational GC for hotspot big data applications. *CoRR*, abs/1704.03764, 2017.

[18] Cliff Click, Gil Tene, and Michael Wolf. The pauseless GC algorithm. In Michael Hind and Jan Vitek, editors, *Proceedings of the 1st International Conference on Virtual Execution Environments, VEE 2005, Chicago, IL, USA, June 11-12, 2005*, pages 46–56. ACM, 2005.

[19] Saar Cohen and Yossi Matias. Spectral bloom filters. In Alon Y. Halevy, Zachary G. Ives, and AnHai Doan, editors, *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data, San Diego, California, USA, June 9-12, 2003*, pages 241–252. ACM, 2003.

[20] Ulan Degenbaev, Jochen Eisinger, Manfred Ernst, Ross McIlroy, and Hannes Payer. Idle time garbage collection scheduling. In Chandra Krintz and Emery D. Berger, editors, *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2016, Santa Barbara, CA, USA, June 13-17, 2016*, pages 570–583. ACM, 2016.

[21] David Detlefs, Christine H. Flood, Steve Heller, and Tony Printezis. Garbage-first garbage collection. In David F. Bacon and Amer Diwan, editors, *Proceedings of the 4th International Symposium on Memory Management, ISMM 2004, Vancouver, BC, Canada, October 24-25, 2004*, pages 37–48. ACM, 2004.

[22] Tamar Domani, Elliot K. Kolodner, and Erez Petrank. A generational on-the-fly garbage collector for java. In Monica S. Lam, editor, *Proceedings of the 2000 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), Vancouver, Britith Columbia, Canada, June 18-21, 2000*, pages 274–284. ACM, 2000.

[23] Benjamin Van Durme and Ashwin Lall. Probabilistic counting with randomized storage. In Craig Boutilier, editor, *IJCAI 2009, Proceedings of the 21st International Joint Conference on Artificial Intelligence, Pasadena, California, USA, July 11-17, 2009*, pages 1574–1579, 2009.

[24] Domenico Ficara, Stefano Giordano, Gregorio Procissi, and Fabio Vitucci. Multilayer compressed counting bloom filters. In *INFOCOM 2008. 27th IEEE International Conference on Computer Communications, Joint Conference of the IEEE Computer and Communications Societies, 13-18 April 2008, Phoenix, AZ, USA*, pages 311–315. IEEE, 2008.

[25] Christine H Flood, Roman Kennke, Andrew Dinn, Andrew Haley, and Roland Westrelin. Shenandoah: An open-source concurrent compacting garbage collector for openjdk. In *Proceedings of the 13th International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools*, pages 1–9, 2016.

[26] Lokesh Gidra, Hans-J Boehm, and Joel Fernandes. Utilizing the linux userfaultfd system call in a compaction phase of a garbage collection process. 2020.

[27] Lokesh Gidra, Gaël Thomas, Julien Sopena, Marc Shapiro, and Nhan Nguyen. Numagic: a garbage collector for big data on big NUMA machines. In Özcan Özturk, Kemal Ebcioglu, and Sandhya Dwarkadas, editors, *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2015, Istanbul, Turkey, March 14-18, 2015*, pages 661–673. ACM, 2015.

[28] Vaibhav Gogte, William Wang, Stephan Diestelhorst, Aasheesh Kolli, Peter M. Chen, Satish Narayanasamy, and Thomas F. Wenisch. Software wear management for persistent memories. In Arif Merchant and Hakim Weatherspoon, editors, *17th USENIX Conference on File and Storage Technologies, FAST 2019, Boston, MA, February 25-28, 2019*, pages 45–63. USENIX Association, 2019.

[29] Google. V8 JavaScript Engine. https://v8.dev/. Accessed: 20-Nov-2025.

[30] Zhiyuan Guo, Zijian He, and Yiying Zhang. Mira: A program-behavior-guided far memory system. In Jason Flinn, Margo I. Seltzer, Peter Druschel, Antoine Kaufmann, and Jonathan Mace, editors, *Proceedings of the 29th Symposium on Operating Systems Principles, SOSP 2023, Koblenz, Germany, October 23-26, 2023*, pages 692–708. ACM, 2023.

[31] Jingkai He, Yunpeng Dong, Dong Du, Mo Zou, Zhitai Yu, Yuxin Ren, Ning Jia, Yubin Xia, and Haibo Chen. How to copy memory? coordinated asynchronous copy as a first-class OS service. In Youjip Won, Youngjin Kwon, Ding Yuan, and Rebecca Isaacs, editors, *Proceedings of the ACM SIGOPS 31st Symposium on Operating Systems Principles, SOSP 2025, Lotte Hotel World, Seoul, Republic of Korea, October 13-16, 2025*, pages 1062–1081. ACM, 2025.

[32] Matthew Hertz and Emery D. Berger. Quantifying the performance of garbage collection vs. explicit memory management. In Ralph E. Johnson and Richard P. Gabriel, editors, *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2005, October 16-20, 2005, San Diego, CA, USA*, pages 313–326. ACM, 2005.

[33] Jiacheng Huang, Yunmo Zhang, Junqiao Qiu, Yu Liang, Rachata Ausavarungnirun, Qingan Li, and Chun Jason Xue. More apps, faster hot-launch on mobile devices via fore/background-aware gc-swap co-design. In Rajiv Gupta, Nael B. Abu-Ghazaleh, Madan Musuvathi, and Dan Tsafrir, editors, *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3, ASPLOS 2024, La Jolla, CA, USA, 27 April 2024- 1 May 2024*, pages 654–670. ACM, 2024.

[34] Xianglong Huang, Stephen M. Blackburn, Kathryn S. McKinley, J. Eliot B. Moss, Zhenlin Wang, and Perry Cheng. The garbage collection advantage: improving program locality. In John M. Vlissides and Douglas C. Schmidt, editors, *Proceedings of the 19th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2004, October 24-28, 2004, Vancouver, BC, Canada*, pages 69–80. ACM, 2004.

[35] Ahmed Hussein, Mathias Payer, Antony L. Hosking, and Christopher A. Vick. One process to reap them all: Garbage collection as-a-service. In *Proceedings of the 13th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments, VEE 2017, Xi'an, China, April 8-9, 2017*, pages 171–186. ACM, 2017.

[36] Richard E. Jones, Antony L. Hosking, and J. Eliot B. Moss. *The Garbage Collection Handbook: The art of automatic memory management*. Chapman and Hall / CRC Applied Algorithms and Data Structures Series. CRC Press, 2011.

[37] Henry G. Baker Jr. List processing in real time on a serial computer. *Commun. ACM*, 21(4):280–294, 1978.

[38] Haim Kermany and Erez Petrank. The compressor: concurrent, incremental, and parallel compaction. In Michael I. Schwartzbach and Thomas Ball, editors, *Proceedings of the ACM SIGPLAN 2006 Conference on Programming Language Design and Implementation, Ottawa, Ontario, Canada, June 11-14, 2006*, pages 354–363. ACM, 2006.

[39] Niel Lebeck, Arvind Krishnamurthy, Henry M. Levy, and Irene Zhang. End the senseless killing: Improving memory management for mobile operating systems. In Ada Gavrilovska and Erez Zadok, editors, *Proceedings of the 2020 USENIX Annual Technical Conference, USENIX ATC 2020, July 15-17, 2020*, pages 873–887. USENIX Association, 2020.

[40] Anatole Lefort, Yohan Pipereau, Kwabena Amponsem, Pierre Sutra, and Gaël Thomas. J-NVM: off-heap persistent objects in java. In Robbert van Renesse and Nickolai Zeldovich, editors, *SOSP '21: ACM SIGOPS 28th Symposium on Operating Systems Principles, Virtual Event / Koblenz, Germany, October 26-29, 2021*, pages 408–423. ACM, 2021.

[41] Haoran Ma, Shi Liu, Chenxi Wang, Yifan Qiao, Michael D. Bond, Stephen M. Blackburn, Miryung Kim, and Guoqing Harry Xu. Mako: a low-pause, high-throughput evacuating collector for memory-disaggregated datacenters. In Ranjit Jhala and Isil Dillig, editors, *PLDI '22: 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation, San Diego, CA, USA, June 13 - 17, 2022*, pages 92–107. ACM, 2022.

[42] Martin Maas, Krste Asanovic, and John Kubiatowicz. A hardware accelerator for tracing garbage collection. In Murali Annavaram, Timothy Mark Pinkston, and Babak Falsafi, editors, *45th ACM/IEEE Annual International Symposium on Computer Architecture, ISCA 2018, Los Angeles, CA, USA, June 1-6, 2018*, pages 138–151. IEEE Computer Society, 2018.

[43] Microsoft. .NET. https://dotnet.microsoft.com/. Accessed: 20-Nov-2025.

[44] Michael Mitzenmacher. Compressed bloom filters. In Ajay D. Kshemkalyani and Nir Shavit, editors, *Proceedings of the Twentieth Annual ACM Symposium on Principles of Distributed Computing, PODC 2001, Newport, Rhode Island, USA, August 26-29, 2001*, pages 144–150. ACM, 2001.

[45] Mozilla. SpiderMonkey JavaScript Engine. https://spidermonkey.dev/. Accessed: 20-Nov-2025.

[46] Khanh Nguyen, Lu Fang, Guoqing Xu, Brian Demsky, Shan Lu, Sanazsadat Alamian, and Onur Mutlu. Yak:

A high-performance big-data-friendly garbage collector. In Kimberly Keeton and Timothy Roscoe, editors, *12th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2016, Savannah, GA, USA, November 2-4, 2016*, pages 349–365. USENIX Association, 2016.

[47] Tony Printezis and David Detlefs. A generational mostly-concurrent garbage collector. In Craig Chambers and Antony L. Hosking, editors, *ISMM 2000, International Symposium on Memory Management, Minneapolis, Minnesota, USA, October 15-16, 2000 (in conjunction with OOPSLA 2000), Conference Proceedings*, pages 143–154. ACM, 2000.

[48] Amanda Raybuck, Tim Stamler, Wei Zhang, Mattan Erez, and Simon Peter. Hemem: Scalable tiered memory management for big data applications and real NVM. In Robbert van Renesse and Nickolai Zeldovich, editors, *SOSP '21: ACM SIGOPS 28th Symposium on Operating Systems Principles, Virtual Event / Koblenz, Germany, October 26-29, 2021*, pages 392–407. ACM, 2021.

[49] Kunal Sareen, Stephen M. Blackburn, Sara S. Hamouda, and Lokesh Gidra. Memory management on mobile devices. In Michael D. Bond, Jae W. Lee, and Hannes Payer, editors, *Proceedings of the 2024 ACM SIGPLAN International Symposium on Memory Management, ISMM 2024, Copenhagen, Denmark, 25 June 2024*, pages 15–29. ACM, 2024.

[50] Thomas Shull, Jian Huang, and Josep Torrellas. Autopersist: an easy-to-use java NVM framework based on reachability. In Kathryn S. McKinley and Kathleen Fisher, editors, *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2019, Phoenix, AZ, USA, June 22-26, 2019*, pages 316–332. ACM, 2019.

[51] Haoyu Song, Sarang Dharmapurikar, Jonathan S. Turner, and John W. Lockwood. Fast hash table lookup using extended bloom filter: an aid to network processing. In Roch Guérin, Ramesh Govindan, and Greg Minshall, editors, *Proceedings of the ACM SIGCOMM 2005 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications, Philadelphia, Pennsylvania, USA, August 22-26, 2005*, pages 181–192. ACM, 2005.

[52] Timothy Stamler, Deukyeon Hwang, Amanda Raybuck, Wei Zhang, and Simon Peter. zio: Accelerating io-intensive applications with transparent zero-copy IO. In Marcos K. Aguilera and Hakim Weatherspoon, editors, *16th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2022, Carlsbad, CA, USA, July 11-13, 2022*, pages 431–445. USENIX Association, 2022.

[53] Darko Stefanovic, Kathryn S. McKinley, and J. Eliot B. Moss. Age-based garbage collection. In Brent Hailpern, Linda M. Northrop, and A. Michael Berman, editors, *Proceedings of the 1999 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages & Applications, OOPSLA 1999, Denver, Colorado, USA, November 1-5, 1999*, pages 370–381. ACM, 1999.

[54] Kun Suo, Jia Rao, Hong Jiang, and Witawas Srisa-an. Characterizing and optimizing hotspot parallel garbage collection on multicore systems. In Rui Oliveira, Pascal Felber, and Y. Charlie Hu, editors, *Proceedings of the Thirteenth EuroSys Conference, EuroSys 2018, Porto, Portugal, April 23-26, 2018*, pages 35:1–35:15. ACM, 2018.

[55] David M. Ungar. Generation scavenging: A non-disruptive high performance storage reclamation algorithm. In William E. Riddle and Peter B. Henderson, editors, *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments, Pittsburgh, Pennsylvania, USA, April 23-25, 1984*, pages 157–167. ACM, 1984.

[56] Chenxi Wang, Haoran Ma, Shi Liu, Yuanqi Li, Zhenyuan Ruan, Khanh Nguyen, Michael D. Bond, Ravi Netravali, Miryung Kim, and Guoqing Harry Xu. Semeru: A memory-disaggregated managed runtime. In *14th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2020, Virtual Event, November 4-6, 2020*, pages 261–280. USENIX Association, 2020.

[57] Chenxi Wang, Haoran Ma, Shi Liu, Yifan Qiao, Jonathan Eyolfson, Christian Navasca, Shan Lu, and Guoqing Harry Xu. Memliner: Lining up tracing and application for a far-memory-friendly runtime. In Marcos K. Aguilera and Hakim Weatherspoon, editors, *16th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2022, Carlsbad, CA, USA, July 11-13, 2022*, pages 35–53. USENIX Association, 2022.

[58] Kunshan Wang, Stephen M. Blackburn, Peter Zhu, and Matthew Valentine-House. Reworking memory management in cruby: A practitioner report. In Martin Maas, Tim Harris, and Onur Mutlu, editors, *Proceedings of the 2025 ACM SIGPLAN International Symposium on Memory Management, ISMM 2025, Seoul, Republic of Korea, 17 June 2025*, pages 109–121. ACM, 2025.

[59] Mingyu Wu, Liang Mao, Yude Lin, Yifeng Jin, Zhe Li, Hongtao Lyu, Jiawei Tang, Xiaowei Lu, Hao Tang, Denghui Dong, Haibo Chen, and Binyu Zang. Jade: A high-throughput concurrent copying garbage collector. In *Proceedings of the Nineteenth European Conference on Computer Systems, EuroSys 2024, Athens, Greece, April 22-25, 2024*, pages 1160–1174. ACM, 2024.

[60] Mingyu Wu, Ziming Zhao, Haoyu Li, Heting Li, Haibo Chen, Binyu Zang, and Haibing Guan. Espresso: Brewing java for more non-volatility with non-volatile memory. In Xipeng Shen, James Tuck, Ricardo Bianchini, and Vivek Sarkar, editors, *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2018, Williamsburg, VA, USA, March 24-28, 2018*, pages 70–83. ACM, 2018.

[61] Lingfeng Xiang, Zhen Lin, Weishu Deng, Hui Lu, Jia Rao, Yifan Yuan, and Ren Wang. Nomad: Non-exclusive memory tiering via transactional page migration. In Ada Gavrilovska and Douglas B. Terry, editors, *18th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2024, Santa Clara, CA, USA, July 10-12, 2024*, pages 19–35. USENIX Association, 2024.

[62] Albert Mingkun Yang, Erik Österlund, and Tobias Wrigstad. Improving program locality in the GC using hotness. In Alastair F. Donaldson and Emina Torlak, editors, *Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation, PLDI 2020, London, UK, June 15-20, 2020*, pages 301–313. ACM, 2020.

[63] Albert Mingkun Yang and Tobias Wrigstad. Deep dive into zgc: A modern garbage collector in openjdk. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 44(4):1–34, 2022.

[64] Anil Yelam, Stewart Grant, Saarth Deshpande, Nadav Amit, Radhika Niranjan Mysore, Amy Ousterhout, Marcos K. Aguilera, and Alex C. Snoeren. Eden: Developer-friendly application-integrated far memory. In Theophilus A. Benson and Radhika Niranjan Mysore, editors, *22nd USENIX Symposium on Networked Systems Design and Implementation, NSDI 2025, Philadelphia, PA, USA, April 28-30, 2025*, pages 1067–1083. USENIX Association, 2025.

[65] Anil Yelam, Kan Wu, Zhiyuan Guo, Suli Yang, Rajath Shashidhara, Wei Xu, Stanko Novakovic, Alex C. Snoeren, and Kimberly Keeton. Pageflex: Flexible and efficient user-space delegation of linux paging policies with ebpf. In Deniz Altinbüken and Ryan Stutsman, editors, *Proceedings of the 2025 USENIX Annual Technical Conference, USENIX ATC 2025, Boston, MA, USA, July 7-9, 2025*, pages 291–306. USENIX Association, 2025.

[66] Yang Yu, Tianyang Lei, Weihua Zhang, Haibo Chen, and Binyu Zang. Performance analysis and optimization of full garbage collection in memory-hungry environments. In Vishakha Gupta-Cledat, Donald E. Porter, and Vivek Sarkar, editors, *Proceedings of the 12th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments, Atlanta, GA, USA, April 2-3, 2016*, pages 123–130. ACM, 2016.

[67] Wenyu Zhao and Stephen M. Blackburn. Deconstructing the garbage-first collector. In Santosh Nagarakatte, Andrew Baumann, and Baris Kasikci, editors, *VEE '20: 16th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments, virtual event [Lausanne, Switzerland], March 17, 2020*, pages 15–29. ACM, 2020.

[68] Wenyu Zhao, Stephen M. Blackburn, and Kathryn S. McKinley. Low-latency, high-throughput garbage collection. In Ranjit Jhala and Isil Dillig, editors, *PLDI '22: 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation, San Diego, CA, USA, June 13 - 17, 2022*, pages 76–91. ACM, 2022.

ARTIFACT EVALUATED
USENIX
AVAILABLE

Unleash All Cores: Scalable Asymmetry-aware DNN Inference on Mobile CPUs

Qianlong Sang
School of Computer Science
Wuhan University

Puyi He
School of Computer Science
Wuhan University

Huanghuang Liang
School of Computer Science
Wuhan University

Yili Gong
School of Computer Science
Wuhan University

Chuang Hu
IOTSC
University of Macau

Xiaobo Zhou
IOTSC
University of Macau

Dazhao Cheng*
School of Computer Science
Wuhan University

Abstract

Asymmetric Multiprocessing (AMP) CPUs are now central to mobile devices, but exploiting them for efficient Deep Neural Network (DNN) inference remains challenging. Naive scheduling across heterogeneous cores often triggers a performance-collapse paradox: adding LITTLE cores degrades throughput due to workload imbalance. Existing approaches rely on static partitioning, which partially mitigates imbalance but fails to adapt to runtime interference, incurs extra task acquisition overhead, and ignores core–kernel affinities—leaving substantial performance untapped.

We present SANI, a scalable, asymmetry-aware inference framework that unleashes the full potential of AMP architectures. SANI introduces three key mechanisms: (1) an affinity-aware kernel issuer that selects cluster-optimal kernels to exploit core–kernel efficiency from the outset; (2) an adaptive granularity scheduler that dynamically merges or splits tasks, balancing load under runtime interference by mapping smaller tasks to slower cores and larger ones to faster cores; and (3) an on-demand kernel switcher that efficiently transforms kernels during workload migration, preserving affinity across clusters. We implement SANI atop Arm-CL and evaluate it on five mobile SoCs. SANI reduces DNN inference latency by 17.6%–23.7% on average (up to 29.5% on some models) while lowering energy consumption by up to 39% compared to state-of-the-art baselines, scaling efficiently across both symmetric and asymmetric CPU configurations.

1 Introduction

On-device deep neural network (DNN) inference has become a cornerstone of modern mobile applications [21, 51, 56, 61]. While GPUs and NPUs provide high throughput, they are not always available across devices or workloads. In contrast, CPUs remain the most broadly deployable platform for mobile inference: they offer predictable performance when accelerators are absent and enable flexible scheduling for diverse application needs [2, 11, 12, 17, 30, 36, 48].

To meet the dual demands of performance and energy efficiency, modern mobile system-on-chips (SoCs) increasingly adopt AMP CPUs designs. These architectures integrate a mix of *high-performance* **big** cores for compute-intensive tasks and *energy-efficient* **LITTLE** cores for lighter workloads [4, 5, 26, 44]. This heterogeneity provides a powerful substrate for efficient inference—but also introduces new challenges for scheduling and workload balance.

A natural way to exploit AMP CPUs for DNN inference is to partition an operator's output tensor into smaller chunks and distribute them across all available cores for parallel execution. Intuitively, this should improve throughput. In practice, however, the asymmetry between big and LITTLE cores undermines this strategy. Our measurements reveal a **performance-collapse paradox**: adding LITTLE cores can increase end-to-end latency by up to 37%. The root cause is workload imbalance: faster threads on big cores frequently stall at synchronization barriers, waiting for slower LITTLE-core threads to complete. These latencies dominate execution time, erasing the benefits of using additional cores.

Existing mainstream solutions for DNN inference on AMP CPUs fall into three categories, each with inherent limitations. First, symmetry-only execution (e.g., Arm Compute Library (Arm-CL) [6]) confines computations to big cores, sidestepping asymmetry but leaving LITTLE cores idle and wasting available resources. Second, proportional static partitioning (e.g., MNN [25]) divides work according to fixed big-to-LITTLE performance ratios. This strategy reduces imbalance under ideal conditions but is brittle in the presence of runtime interference—ubiquitous on mobile devices—causing significant wait latencies. Third, fine-grained dynamic dispatch, such as AsyMo [49], partitions work into many small tasks guided by cost models. While more adaptive, the fine granularity introduces high task acquisition overhead due to the frequency of dequeue operations.

Beyond scheduling, core–kernel affinity plays a critical but underexplored role. Big and LITTLE cores have distinct mi-

*Corresponding author (dcheng@whu.edu.cn).

croarchitectures that favor different operator implementations (kernels). Existing frameworks typically select a single default kernel per operator and apply it uniformly across clusters, ignoring these affinities. When workloads migrate between clusters, performance often deteriorates. Our measurements show that aligning kernel choice with cluster affinity can yield over 30% performance improvement.

In short, prior methods lack two critical capabilities: (1) runtime control of workload granularity to maintain balance under interference, and (2) mechanisms to exploit core–kernel affinity for efficient execution. These limitations leave significant performance headroom untapped.

To overcome these limitations, we present **SANI**, a ***S***calable ***A***symmetry-aware ***N***eural ***I***nference framework, which is designed to exploit asymmetric mobile CPUs fully. SANI introduces two key innovations. First, an affinity-aware execution strategy that selects the most efficient kernel for each CPU cluster and preserves affinity even during workload migration. Second, an adaptive granularity scheduler that dynamically adjusts task sizes, assigning smaller tasks to slower cores and larger ones to faster cores, thereby sustaining load balance under unpredictable runtime interference.

Firstly, SANI employs a holistic approach to maintain core–kernel affinity throughout the inference pipeline. At initialization, it proactively selects cluster-specific kernels for each operator using a hardware-aware cost model that considers microarchitectural features such as SIMD width and cache hierarchy. This ensures that both big and LITTLE cores are issued their most efficient kernel from the outset, avoiding the one-size-fits-all choices made by existing frameworks. To preserve this efficiency under dynamic execution, SANI performs on-demand kernel transformations, remapping workload indices between the layouts of source and target kernels when workloads migrate across clusters. Unlike prior systems that suffer penalties after migration, this reactive step guarantees that cores continue executing their most-affine kernel, sustaining high performance even in the presence of runtime interference and rescheduling.

Secondly, SANI introduces an adaptive granularity scheduler to combat workload imbalance from core asymmetry and runtime interference. Unlike prior approaches that rely on fixed partitions or excessively fine-grained dispatch, SANI continuously monitors lightweight runtime feedback from each thread to guide scheduling decisions. Based on this feedback, it dynamically adjusts task granularity through two fundamental operations, Merge and Split. Faster threads are assigned larger, merged workloads to reduce task acquisition overhead, while slower threads receive smaller, split workloads to prevent them from becoming bottlenecks. This adaptive mechanism enables SANI to maintain balanced execution even under unpredictable interference from background processes, sustaining throughput while minimizing synchronization stalls and queueing costs.

Together, these mechanisms make SANI inherently scalable across heterogeneous CPU configurations, from symmetric big-core execution to fully asymmetric deployments with LITTLE cores.

In summary, the contributions of this paper are:

- We conduct a measurement study showing that adding LITTLE cores for DNN inference on asymmetric CPUs can paradoxically increase end-to-end latency, and we analyze workload imbalance as the root cause of this performance-collapse phenomenon.
- We design an affinity-aware kernel issuer to proactively select cluster-specific kernels, and an on-demand kernel switcher to transform workloads during migration; together, these components preserve core–kernel affinity and maximize execution efficiency.
- We propose an adaptive granularity scheduler that dynamically adjusts task sizes at runtime using merge and split operations, effectively balancing workloads and mitigating bottlenecks of runtime interference.
- We demonstrate that SANI scales efficiently across various CPU configurations, sustaining performance and energy efficiency on five commercial SoCs.

We implemented SANI atop the Arm-CL. Evaluated on five distinct mobile devices across six representative neural networks, SANI delivers average latency reductions of 17.6%–23.7% and up to 39% lower energy consumption, while demonstrating robust performance and significantly lower wait latency under runtime interference. Notably, SANI is the first inference framework designed to holistically address both the inherent compute asymmetry and runtime interference characteristic of mobile AMP CPUs. This work targets operator-level parallelizable workloads and is not designed for large-scale on-device LLM inference, where NPUs are the more suitable target. The core abstractions are operator-shape rather than model-class specific, and we discuss the carry-over to transformer workloads in § 5.5.

2 Background and Motivation

2.1 DNN Inference on AMP

Modern mobile SoCs increasingly employ AMP architectures that combine heterogeneous CPU clusters on a single die [4,5, 26,44]. A typical big.LITTLE configuration integrates high-performance cores with energy-efficient ones [5], offering a flexible trade-off between performance and power efficiency. These clusters share a unified physical address space and remain cache-coherent through the on-chip interconnect. Each cluster keeps its own private L1 and L2 caches, while the last-level L3 cache and main memory are shared across clusters. As a result, big and LITTLE cores can collaborate on DNN inference without explicit data movement.

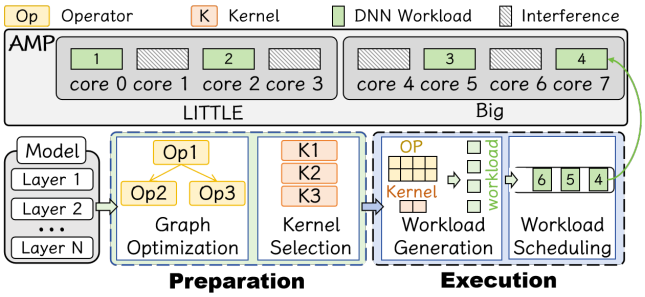


Figure 1: The process of DNN inference on AMP.

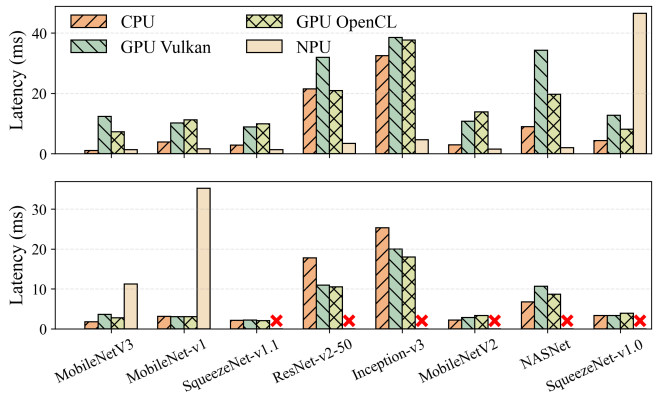


Figure 2: Inference latency comparison across CPU, GPU (Vulkan/OpenCL), and NPU backends on Google Pixel 9 (top) and OnePlus 12 (bottom). Missing bars indicate unsupported models or execution failures.

However, unlike dedicated servers, mobile environments are subject to runtime interference. Foreground applications, background services, and system daemons frequently contend for shared cores [53], leading to volatile compute availability [52, 57]. In this complex runtime environment, DNN inference pipelines translate a network's design into efficient execution flows. As shown in Fig. 1, the process begins with graph optimization, translating high-level layers into mathematical operators, followed by kernel selection, where hardware-tuned kernels are chosen to realize each operator. During execution, operators are partitioned into fine-grained workloads that are placed in task queues and fetched by CPU threads. Specifically, each workload is the schedulable unit handed to one inference thread, consisting of a contiguous sub-region of the operator's output tensor and a pointer to the kernel that executes it. These workloads collaboratively apply the chosen kernel to compute the operator's output tensor.

Why CPU Inference Matters. While mobile GPUs and NPUs are often considered preferred accelerators, CPU-based execution remains indispensable. As shown in Fig. 2, we benchmarked DNNs across CPU, GPU, and NPU backends using MNN [25] on the Google Pixel 9 [19] and OnePlus 12 [39]. Well-optimized CPU inference outperforms mobile GPUs for lightweight models such as MobileNetV3 [23] by 6–11× due to GPU kernel launch overhead and data transfer costs [58]. Moreover, mobile GPUs employ FIFO scheduling, causing inference tasks to compete with rendering and degrade user experience [20]. NPUs, while efficient for supported models, suffer from compatibility limitations: on the Pixel 9, SqueezeNetV1.0 [24] falls back to CPU with 10× slowdown (46.5 ms vs. 4.4 ms), while on the OnePlus 12, most tested models including ResNet-50 [22] and Inception-V3 [47] fail entirely due to vendor-specific operator support gaps. These observations underscore that CPUs are the only universally available and reliable backend for on-device inference, making CPU efficiency essential on AMP architectures.

2.2 Analysis of Existing DNN Inference on AMP

While many DNN inference frameworks exist for homogeneous multi-core systems [37, 54] and heterogeneous SoCs (CPU/GPU/NPU) [1, 9, 15, 27, 35], they face significant performance constraints on AMP architectures. Their limitation stems from an implicit assumption of core symmetry in multi-threaded execution. This prevents the effective use of asymmetric cores and results in suboptimal performance. Therefore, developing frameworks that efficiently utilize all cores is critical for on-device AI. To this end, we perform a measurement study to quantify the performance gap and identify its root causes.

Measurement Setup. We evaluated four convolutional neural networks: MobileNet [23], SqueezeNet, ResNet-50, and Inception-V3, on a Google Pixel 9 equipped with an asymmetric octa-core CPU (1×Cortex-X4, 3×Cortex-A720, 4×Cortex-A520) at their maximum frequencies. To streamline core-type comparisons, Cortex-X4 and Cortex-A720 were categorized as "big" cores throughout the experiments. All measurements were conducted without concurrent foreground or background applications. The baseline implementation leveraged the Arm-CL [6], Arm's official inference framework optimized for mobile architectures. Subsequently, we conducted a systematic analysis of inference latency scaling to thread number increments across heterogeneous core configurations.

Performance Gap and Causes. Fig. 3 illustrates a pronounced performance collapse, revealing a significant discrepancy between the observed *Native* performance and a theoretical *Expected* baseline. We established the *Expected* baseline to serve as a theoretical upper bound on performance. First, we profiled each model to characterize the proportion of its workload that can be parallelized. Then, based on Amdahl's Law [3], we projected the ideal latency scaling. This projection incorporates the fixed performance ratios between big and LITTLE cores, which are specified by chip manufacturers in configuration files [8]. The *Expected* curve, therefore,

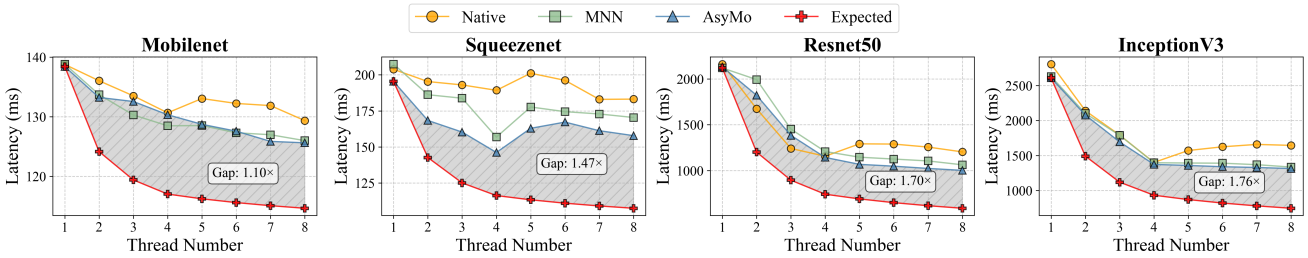
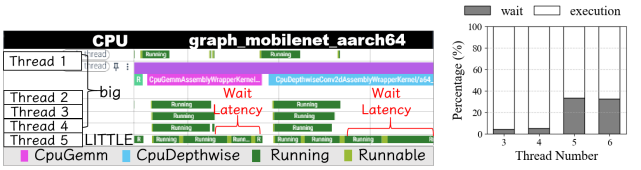
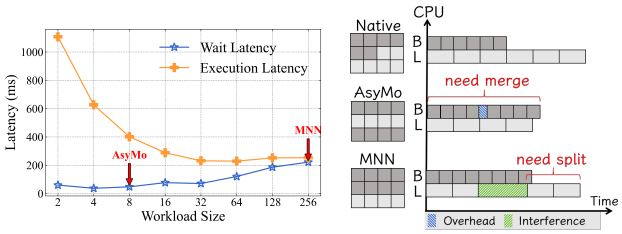


Figure 3: Performance Collapse Paradox: DNN inference throughput degradation with the introduction of LITTLE cores on AMP architectures. The first 4 threads run on different big cores. Others run on different LITTLE cores.



(a) *A real trace of thread execution timeline showing LITTLE cores becoming bottlenecks* (b) *Latency Breakdown*

Figure 4: Bottleneck analysis and latency breakdown for MobileNet inference on Google Pixel 9.



(a) *Impact of workload size on latencies for GemmAssembly kernel.* (b) *An illustration of inefficiency in static workload size.*

Figure 5: Analysis of static workload size inefficiencies on AMP architecture.

represents an ideal scenario that assumes perfect scaling without accounting for real-world factors like scheduling overhead or interference.

Contrary to this theoretical expectation, our analysis reveals a paradox. Integrating LITTLE cores, despite increasing the total number of available threads, markedly increases the overall inference latency.

To pinpoint the cause, we decompose latency into two components in multi-threaded DNN inference: *execution latency*, which measures time spent on computation, and *wait latency*, which quantifies synchronization delays due to imbalanced task distribution and scheduling interference. We employed *Perfetto* [18], the Android's official performance tracing tool, to monitor thread behavior during MobileNet inference. As depicted in Fig. 4(a), on a configuration with 4 big cores and 1 LITTLE core, the thread on the LITTLE core requires substantially more time to complete identical workloads. This arises because the default scheduler in Arm-CL evenly distributes tasks across threads, disregarding core performance disparities and thereby transforming LITTLE cores into bottlenecks. The latency breakdown in Fig. 4(b) confirms this analysis, showing that introducing LITTLE cores increases wait latency from roughly 5% to 30% of the total inference time, as faster big cores remain idle waiting for slower LITTLE cores to complete their workloads.

To address this issue, current approaches can be categorized into three strategies. First, symmetry-only execution as employed by Arm-CL, which defaults to use only homogeneous cores (typically just one core in a common 1+3+4 CPU core set), sacrificing potential computational resources to avoid asymmetry complications. Using only big cores forgoes the energy efficiency that LITTLE cores can deliver for sustained inference, while using only LITTLE cores wastes the throughput that big cores can deliver on the critical path. Second, proportional workload partition that allocates tasks according to computational capability ratios between asymmetric processors, exemplified by MNN [25]. Third, fine-grained partitioning with dynamic scheduling, as in AsyMo [49], which divides computation into smaller tasks and balances load distribution with memory access efficiency.

As shown in Fig. 3, implementing these partitioning strategies in Arm-CL yield speedups of approximately 1.2× to 1.25× for complex models like ResNet-50 and Inception-V3 when using all cores.

Nonetheless, these improvements fall substantially short of theoretical expectations, highlighting a significant performance gap. This gap is particularly critical for on-device inference, which is highly latency-sensitive. Many real-time mobile applications operate under strict per-frame latency budgets (e.g., 8.3 ms for 120 FPS [53]), making even millisecond-scale improvements for DNN models highly significant in closing the gap to the theoretical optimum.

Key Observations. Through the measurement conducted on the same Pixel 9 setup as the earlier experiments, we have the following two key observations:

Observation 1: *Static workload size becomes particularly inefficient in AMP architectures operating under interference-prone systems.* Mobile devices concurrently run numerous system and user applications, creating a dynamic execution

environment where inference threads compete for shared resources and face frequent preemption. Consequently, the effective compute speed of any core can fluctuate unpredictably. Existing frameworks that rely on static workload partitioning are ill-equipped to handle this volatility. Specifically, an operator's computation is partitioned into fixed-size workloads that are dispatched to inference threads, and threads wait at a synchronization barrier once their workloads complete. The workload size here refers to the sub-region of the operator's output tensor that a single workload covers, determined by its extents along multiple dimensions. As shown in Fig. 5(a), their fixed-size workloads create a fundamental trade-off, where larger workloads reduce overhead but increase wait latency under interference, while smaller workloads do the opposite. A larger workload size amplifies wait latency, since the slow thread holding the final workload forces the others to idle at the barrier for its entire duration. A smaller workload size yields more workloads in total, adding task-acquisition overhead to the execution latency. This dilemma is illustrated in Fig. 5(b). Fine-grained partitioning, as used by AsyMo, incurs extra task acquisition overhead as faster cores (e.g., the big core) must fetch tasks multiple times, an issue that could be mitigated by merging workloads. Conversely, coarse-grained proportional partitioning, as used by MNN, leads to severe wait latency when interference slows down a core (e.g., the LITTLE core), a problem that could be solved by splitting its large workload for other cores to share.

Observation 2: *Different core types perform optimally with distinct kernel implementations, creating core-kernel affinity patterns.* The optimal code implementation for an operator is not universal across an AMP CPU, as different core types show distinct performance preferences. To quantify this, we benchmarked representative Convolution and DepthWiseConvolution operators on big and LITTLE cores using various kernel implementations. As detailed in Table 1, the results reveal clear affinities. The best-performing kernel for a GEMM operator on big cores is 9.1% faster than the alternative, while on LITTLE cores, the preference is reversed. This effect is even more pronounced for DepthWise convolutions, where using a suboptimal kernel can degrade performance by over 34%. Current frameworks, however, are blind to this. They select a single kernel for an operator during initialization and use it universally. This static choice inevitably leads to missed optimization opportunities, as one of the two core clusters is forced to execute a non-affine kernel. Moreover, when workloads migrate between clusters at runtime, the affinity mismatch persists, resulting in suboptimal performance and preventing the system from reaching its full potential.

Opportunities. These observations highlight a clear path toward optimization. The core challenge from observation 1 is that any static workload size is inherently suboptimal in dynamic mobile environments. This necessitates an *adaptive granularity scheduler* that can dynamically adjust task sizes to balance execution efficiency and wait latency under interference. Similarly, observation 2 presents a two-fold challenge related to core-kernel affinities. An effective solution must both proactively identify and prepare the optimal kernel for each cluster, which calls for an *affinity-aware kernel issuer*. It must also reactively handle workload migrations at runtime, requiring an *on-demand kernel switcher* to perform efficient, on-the-fly transformations. Together, these three components form the holistic SANI framework, designed to systematically address these challenges and achieve near-optimal performance on AMP architectures.

Table 1: Kernel execution time (ms) across core types. h1: $hybrid_6\times16$, sg: $sgemm_8\times12$, d1: $depthfirst_2\times2$, d2: $depthfirst_3\times3$. B/W: Best/Worst ratio.

Operator	**Convolution**			**DepthWiseConv**		
Kernel / Core	**h1**	**sg**	**B/W**	**d1**	**d2**	**B/W**
big	**2958**	3226	1.09×	92	**79**	1.34×
LITTLE	4509	**4371**	1.03×	**291**	345	1.19×

3 SANI Design

3.1 System Overview

As shown in Fig. 6, SANI's architecture is composed of three synergistic components designed to maximize inference performance on asymmetric multi-core processors.

The workflow begins with the Affinity-Aware Kernel Issuer. This module evaluates operators within a neural network to identify opportunities for a dual-kernel issue, where distinct, cluster-specific kernel implementations are prepared for both big and LITTLE cores based on their hardware affinity. It then standardizes all workloads into a uniform 'block' format and organizes them into separate queues for each processor cluster (e.g., a big-core queue and a LITTLE-core queue). Here, a block is a kernel-agnostic partition unit over the operator's output tensor. Its tile dimensions are set to the least common multiple of the native output sizes of the big- and LITTLE-affine kernels. Blocks are designed as the uniform abstraction above workloads with which the scheduler organizes parallel dispatch. Next, the Adaptive Granularity Scheduler manages the runtime execution. It assigns threads to pull workload blocks from their corresponding cluster's queue. By monitoring the completion time of each block, the scheduler dynamically adjusts task granularity using two core operations: Merge and Split. Merging combines smaller blocks into larger ones to reduce acquisition overhead for faster threads, while splitting breaks down large blocks to prevent slower threads from creating bottlenecks, thus ensuring a balanced load. Finally, the On-Demand Kernel Switcher is triggered when the scheduler decides to migrate a workload between asymmetric CPU clusters to maintain workload balance. If the source and target clusters have different optimal kernels

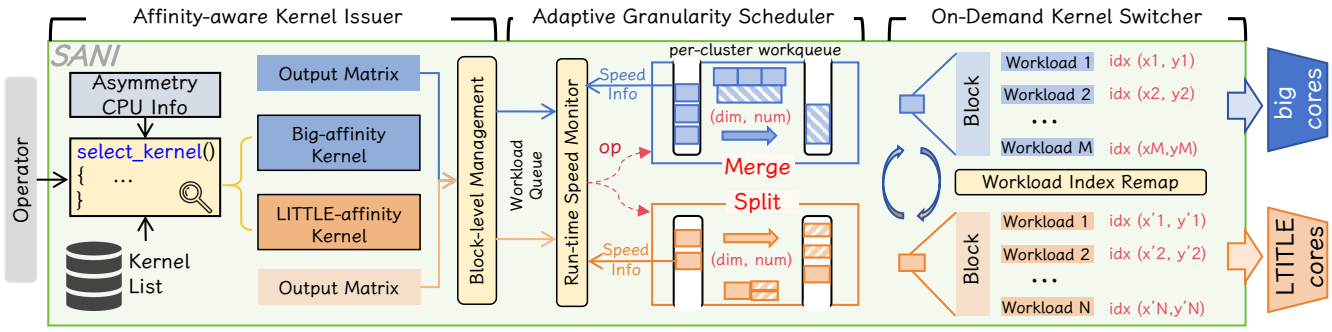


Figure 6: SANI architecture.

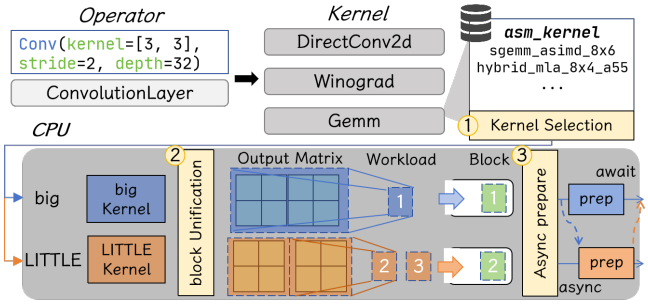


Figure 7: Workflow of affinity-aware kernel issuer.

for that workload, the switcher performs the kernel transformation. It uses a pre-computed map to correctly convert the workload's indices from the source kernel's layout to the destination's. This ensures the new kernel executes on the same computational region, preserving correctness while maximizing performance on the target core. This tightly integrated workflow, from multi-kernel preparation to dynamic scheduling and on-demand transformation.

3.2 Affinity-Aware Kernel Issuer

The affinity-aware kernel issuer prepares operators for efficient heterogeneous execution. Its primary role is to select optimal, cluster-specific kernels and unify their workload representations into a standardized format, all while hiding preparation overhead. As shown in Fig. 7, this process involves three key steps.

Kernel Selection. For each operator O_j in the neural network, the issuer selects the most suitable kernel implementation for a given CPU cluster H by predicting its performance. It leverages a cost model that calculates an affinity score, $A(O_j, K_i, H)$, for each available kernel variant K_i:

$$A(O_j,K_i,H) = \alpha \cdot C(O_j,K_i) + \beta \cdot M(O_j,K_i) + \gamma \cdot I(K_i,H) \quad (1)$$

where $C(O_j,K_i)$ represents the computational cost, which estimates the total number of arithmetic operations based on the operator's tensor shapes and the kernel's logic. $M(O_j,K_i)$ is the memory cost, estimating the total data movement by analyzing the kernel's specific data access pattern for the given operator shapes.

Finally, $I(K_i,H)$ is an instruction compatibility score that quantifies how well the kernel's instruction mix (e.g., specialized SIMD instructions) aligns with the target cluster's micro-architecture. The affinity score is designed to be inversely proportional to execution latency. The cluster-specific weighting factors α, β, and γ are pre-determined by fitting the model to empirical latency measurements using linear regression during an offline profiling stage. The selection process is thus formalized as finding the optimal kernel K_H^* for operator O_j on cluster H:

$$K_H^*(O_j) = \arg\max_{K_i} A(O_j,K_i,H) \quad (2)$$

This process allows the issuer to identify the optimal kernel pair—one for big cores and one for LITTLE cores—for each operator that can benefit from a dual-kernel strategy.

Block Unification. Different kernel implementations often have distinct output shapes and compute orders, which complicates scheduling and workload migration. To address this heterogeneity, we introduce a standardized 'block' abstraction for a consistent workload interface. This unification process involves two key steps. First, to align the computational sequence, we define a unified partitioning and iteration strategy. The operator's total output tensor is partitioned along its height and width dimensions based on the kernel's native tile size, and we enforce a height-first iteration order. This ensures that all kernel variants process workloads in a predictable and consistent sequence. Second, to harmonize the different output shapes, we define the unified block's dimensions using the Least Common Multiple (LCM) of the candidate kernels' output tile dimensions. For example, if a big-core kernel has a 2×4 output tile and a LITTLE-core kernel has a 2×2 tile, the unified block size becomes 2×4. The entire operator's output tensor is then logically partitioned into these blocks. This process establishes a clear mapping between a single block's index and the corresponding workload indices for both the big-core and LITTLE-core kernels, which is critical for subsequent scheduling and migration. For the initial dispatch, these blocks are distributed to the cluster-specific queues, with the

```
struct W {
  int split_dim;
  int start[MAX_DIM];
  int end[MAX_DIM];
  void (* kernel)();
};
W merge(vector<W> w) {
  int dim = w[0].split_dim;
  W w_m = w[0];    // initialize
  w_m.end[dim] = w.back().end[dim];
  return w_m;
}
```

```
vector<W> split(W w, int n, int d) {
  vector<W> w_v;
  W t_w = w;
  int k = (w.end[d] - w.start[d]) / n;
  for (int i = 0; i < n; i++) {
    t_w.start[d] = w.start[d] + i*k;
    t_w.end[d] = w.start[d] + (i+1)*k;
    t_w.split_dim = d;
    w_v.emplace_back(t_w);
  }
  return w_v;
}
```

Figure 8: Merge and Split APIs in SANI.

number of blocks for each queue proportional to the cluster's relative compute capacity. Using cluster-specific queues instead of a single global queue serves two purposes. First, threads on each cluster pull blocks that have been preloaded with the affine kernel and act on data already warm in that cluster's private caches, preserving locality. Second, separating the queues avoids the contention that a single global queue would impose on all inference threads at every dequeue.

Asynchronous Preparation. Preparing distinct kernels for each cluster could introduce significant latency if performed sequentially, as each requires setup tasks like memory allocation and weight pre-transpose. Concretely, preparation covers per-operator memory allocation for the dual-kernel pair, weight pre-transposition into each kernel's native layout, and construction of the index map used by block unification. It runs once at model load and is not repeated during online inference. To mitigate this overhead, SANI employs an asynchronous preparation strategy. The preparation tasks for multiple kernels are submitted to a thread pool concurrently. Since these preparation stages are typically single-threaded, this parallelism effectively utilizes available multi-core resources, hiding the latency of the additional kernel variants. A synchronization barrier ensures all preparations are complete before the online inference phase begins. The overhead introduced by this strategy will be quantified in our overhead analysis (§ 5.4).

3.3 Adaptive Granularity Scheduler

The primary goal of our adaptive granularity scheduler is to balance wait latency and execution latency to maximize overall system performance in AMP environments.

Design Rationale. Observation 1 shows that static workload size cannot fit all execution scenarios, so we propose an adaptive granularity scheduler that adjusts workload size and distribution. Our design philosophy follows the principle: *"With great power comes great responsibility"*. Threads demonstrating higher execution speed should handle larger workloads, as they are less likely to become performance bottlenecks even under system interference. Conversely, slower threads should process smaller workloads to prevent them from becoming system-wide bottlenecks.

Workload Operations. As shown in Fig. 8, we define a `struct W` to represent a workload, encapsulating both the computational region and the kernel to be executed. It contains start and end indices for each dimension, the dimension used for partitioning, and a pointer to its assigned kernel. Based on this structure, we define two fundamental operations: Merge and Split. The Merge operation takes a vector of contiguous workloads and a dimension as input, combining them into a single, larger workload. It creates a new workload that spans from the first workload to the last one along the specified dimension. Conversely, the Split operation takes a workload and divides it into multiple smaller sub-workloads along the dimension. Each resulting sub-workload maintains identical boundaries in all other dimensions. Both operations adjust workload size within a single cluster queue and preserve the kernel pointer carried by each workload.

Algorithm 1: Adaptive Granularity Scheduling

Input: W: Workloads Queue; T: Number of threads;
k: Number of threads to adjust per round;
Output: Completed execution of all workloads

1 Initialize counter $C_i = 1$ for each thread $i \in \{1, 2, ..., T\}$
2 Initialize round counter $r = 0$
3 **while** W *is not empty* **do**
4 $r \leftarrow r + 1$
5 **for** *each thread* $i \in \{1, 2, \ldots, T\}$ *in parallel* **do**
6 **if** $C_i \geq 1$ **then**
7 Fetch C_i workloads $\{w_1, w_2, \ldots, w_{C_i}\}$ from W
8 $w_i \leftarrow \text{Merge}(\{w_1, w_2, \ldots, w_{C_i}\})$
9 **else**
10 Fetch one workload w from W
11 Find the smallest d in `dim` such that $w[d] > T$.
$\{w_1, w_2, \ldots, w_T\} \leftarrow \text{Split}(w, T, d)$
12 Execute w_i with $w_i.kernel$
13 Record completion time t_i for thread i in round r
14 Sort threads by completion time $\{t_1, t_2, \ldots, t_T\}$
15 **for** *the k fastest threads i* **do**
16 $C_i \leftarrow C_i + 1$
17 **for** *the k slowest threads i* **do**
18 $C_i \leftarrow C_i - 1$

Adaptive Scheduling. Algorithm 1 implements our core scheduling strategy. The algorithm operates in rounds, processing workloads until the queue is empty. In each round, after all threads complete their tasks (Lines 6–15), they are sorted by completion time (Line 16). Following our design principle, the k fastest threads are rewarded with larger workloads for the next round by incrementing their counters, enabling them to merge more blocks (Line 8). Conversely, the k slowest threads are penalized by decrementing their counters. If a thread's counter drops below one, it must split its next workload (Lines 10-13). To prevent creating excessively small tasks, the split logic first identifies the smallest dimension whose iteration count is still larger than the number of threads

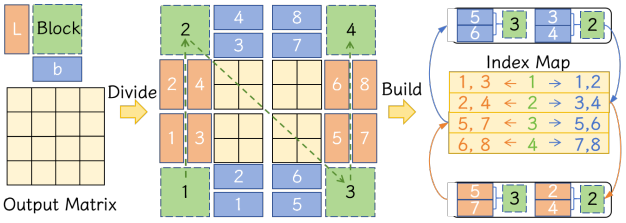


Figure 9: Workflow of on-demand kernel switcher.

before partitioning (Line 11). This adaptive approach gradually aligns task granularity with each thread's true runtime capacity, inherently accounting for both system interference and processor asymmetry.

With N workloads and T threads, each workload is dequeued once, giving a base cost $O(N)$. Moreover, merges across rounds sum to $\Theta(N)$. Splits add only a constant-factor overhead since each event produces at most T sub-workloads. The per-round sort over T completion times is $O(1)$. Therefore, the overall time complexity is $O(N)$.

3.4 On-Demand Kernel Switcher

To maintain workload balance when a cluster's workload queue is depleted, the adaptive granularity scheduler initiates workload migration from another cluster, which creates a challenge for exploiting core-kernel affinity. A core receives a workload prepared for a different architecture, leading to suboptimal performance if executed with a non-affine kernel. The on-demand kernel switcher addresses this by performing an efficient, on-the-fly transformation of the workload.

To ensure transformations are fast and effective, the on-demand kernel switcher incorporates three key optimization strategies. First, to accelerate the index mapping, it pre-builds and stores a direct-mapped lookup table. enabling constant-time lookups at runtime. Second, it employs a threshold-based filter. Workloads smaller than a pre-defined size are exempt from transformation, avoiding overhead where the potential performance gain is negligible. Third, it implements a specialized policy for merged workloads introduced by the scheduler. To handle the larger aggregated workloads efficiently, the switcher uses a columnar transformation strategy. It leverages the start and end indices of the merged block to retrieve the entire range of corresponding target workload indices from the pre-built map in a single operation, which avoids costly individual lookups for each sub-block and is further optimized by the height-first data layout.

When a workload is migrated at runtime, the switcher executes a kernel transform that leverages these optimizations, a process illustrated in Fig. 9. For instance, consider an operator where the selected big-affine kernel has a 1x2 tile size and the LITTLE-affine kernel has a 2x1 tile. Based on the LCM rule, this necessitates a unified 2x2 block for management. Consequently, a 4x4 output tensor for this operator is partitioned into four such logical blocks. The switcher's pre-built index map links each block index to the corresponding workload indices for both kernel types, following a height-first iteration order. Therefore, block index 2 corresponds to workload indices 3 and 4 for the big-affine kernel. Through the map, the switcher finds that this same computational region corresponds to workload indices 2 and 4 for the LITTLE-affine kernel. After passing the size threshold check, the switcher performs a constant-time lookup to retrieve these target indices. Finally, it updates the workload's payload, replacing the kernel pointer with the target cluster's affine kernel and setting the new workload index, which is then passed to the target kernel for execution. This process guarantees that the migrated workload computes the identical output region, preserving correctness while maximizing performance on the new host core.

3.5 Scalability by Design

SANI is designed to scale naturally across heterogeneous CPU configurations, rather than optimizing for a fixed device or core count. The affinity-aware kernel issuer ensures each cluster always executes its most efficient kernel, decoupling operator efficiency from the number or type of cores. The adaptive granularity scheduler rebalances workloads by merging or splitting tasks, which allows the system to maintain balance as concurrency increases or runtime interference fluctuates. Finally, the on-demand kernel switcher preserves affinity under workload migration by transforming kernels at runtime, preventing the penalties that usually arise when workloads shift between clusters.

Together, these mechanisms make SANI inherently scalable: it can seamlessly leverage both big and LITTLE cores across diverse SoCs without hardware-specific retuning. This scalability-by-design principle distinguishes SANI from prior approaches; in evaluation, we demonstrate its ability to sustain efficiency as the number of heterogeneous cores grows.

4 Implementation

We implemented SANI with approximately 11.6k LoC atop Arm-CL (tag v52.3) [6] and plan to publish the code. Our implementation addresses the following key areas:

Scheduler Integration. To maintain backward compatibility, we implement SANI as a custom scheduler that overrides Arm-CL's `schedule_common()` function, allowing it to intercept the execution flow without requiring user-code modifications. Our adaptive granularity scheduler maintains separate queues for each core cluster and implements thread affinity control to bind threads to specific big or LITTLE cores, leveraging standard `pthread` APIs to interact correctly with the Linux scheduler.

Kernel Issue and Transformation. For the affinity-aware kernel issuer, we re-implement Arm-CL's `estimate_cycles()`

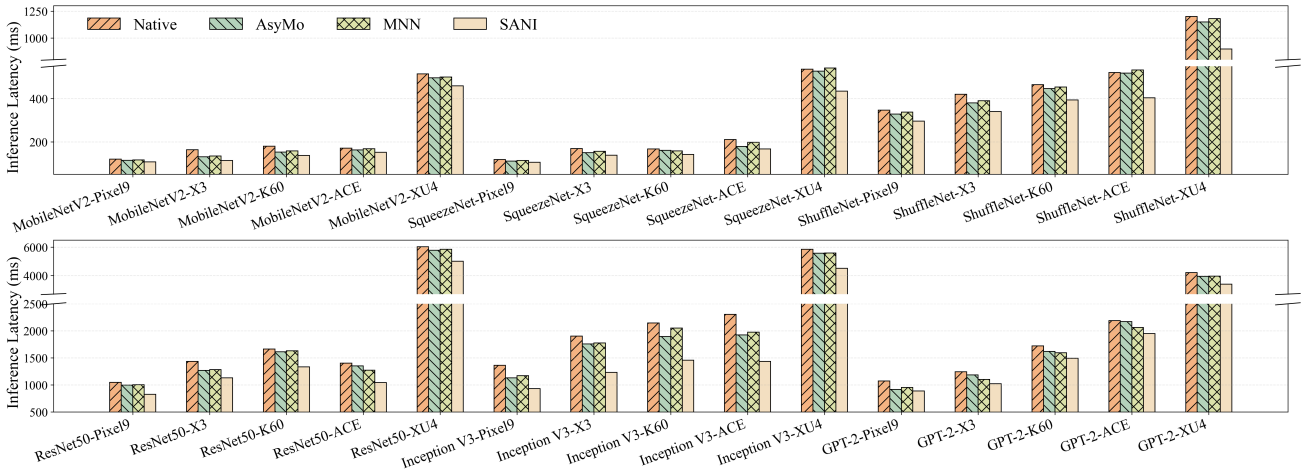


Figure 10: End-to-end latency on Pixel9, X3, K60, ACE, XU4 with *Native*, *AsyMo*, *MNN*, and our work.

Table 2: Device specifications.

Device	Release	SoC	CPU Set	OS
Google Pixel 9	Aug 2024	Tensor G4	1+3+4	Android 14
OPPO Find X3	Mar 2021	SM8350	1+3+4	Android 13
Redmi K60	Dec 2022	SM8475	1+3+4	Android 13
OnePlus Ace 5	May 2025	Mediatek 9400	1+3+4	Android 15
Odroid XU4	Jul 2015	Exynos 5422	4+4	Android 7

function. This allows us to feed the distinct hardware profiles of big and LITTLE cores into our cost model to determine the optimal kernel for each cluster. If the resulting affine kernels are different, SANI initiates a dual-kernel issue and utilizes an asynchronous mechanism to prepare them in parallel via a thread pool. For the on-demand kernel switcher, when a kernel transformation is required, SANI utilizes the pre-computed index map to retrieve the target workload's index with minimal latency, then generates a new workload to compute the same computational region for the new kernel.

5 Evaluation

5.1 Methodology

Testbed. To evaluate the effectiveness and generalizability of SANI, we conduct experiments on four smartphones and one development board, featuring a variety of SoCs from different manufacturers. These devices include the Google Pixel 9 [19], OPPO Find X3 Pro [41], Redmi K60 [55], OnePlus Ace 5 Ultra [40], and the Odroid XU4 development board [38]. Detailed specifications are provided in Table 2. Additionally, the OPPO Find X3 Pro was modified to support the use of Monsoon Power Monitor [34] for precise power consumption measurement.

Workloads. We evaluate SANI with five widely-used DNN models: MobileNetV2 [33], ResNet-50 [22], SqueezeNet [24], ShuffleNet [60], Inception-V3 [47], and one transformer model: GPT-2 [43]. All experiments use pre-trained models in FP32 precision with NEON acceleration enabled. For model compatibility, we deploy GPT-2 using the Arm NN [7] framework with its TensorFlow Lite parser, as Arm-CL lacks support for transformer architectures. Arm NN provides a high-level neural network API built atop Arm-CL while maintaining equivalent computational performance. For vision models, we use standard ImageNet-resolution inputs (224×224) that match common mobile deployment settings. For GPT-2, we use fixed-length 20-token prompts generated randomly, which stress both prefill and decode under a controlled setting.

Baselines. We compare SANI with three state-of-the-art approaches: *Native* (default configuration of Arm-CL [6]), *AsyMo* [49], and *MNN* [25], as detailed in Section 2.2. Our preliminary measurements show that other mainstream frameworks such as TensorFlow Lite [28], ONNX Runtime [13], and PyTorch Mobile [42] exhibit similar inefficiencies on AMP CPUs due to their static, symmetry-assuming workload partitioning. We therefore adopt AsyMo as the primary asymmetry-aware baseline, which has been shown to outperform these frameworks [49]. All implementations are based on Arm-CL, Arm's official open-source framework optimized for Arm architectures. For *AsyMo*, which offers both latency-first and energy-first modes, we implement its latency-first configuration for performance comparison and port its core algorithms to Arm-CL. Similarly, rather than directly comparing with *MNN*, which would introduce unfair comparison due to different framework-specific optimizations, we implement its scheduling strategy within Arm-CL to ensure consistent evaluation.

Metrics. We use three metrics to evaluate performance: 1) *Inference Latency*, measuring end-to-end execution time, 2) *Energy Consumption*, calculated from power measurements during inference, and 3) *Scaling Efficiency*, computed as the speedup relative to the execution on 4 big cores only.

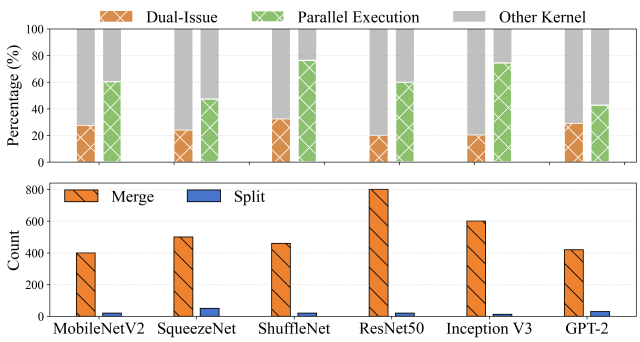


Figure 11: Model analysis: Dual-issue and parallel execution potential (top) and merge-split operation frequency (bottom).

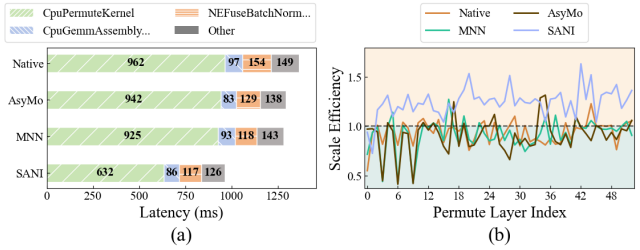


Figure 12: Performance over Inception-V3: (a) Execution time breakdown by kernel types, (b) Scaling efficiency across baselines for Permute kernels.

5.2 Overall Performance

Fig. 10 illustrates the end-to-end inference latency of SANI compared with three baseline approaches across different neural network architectures and mobile devices. SANI consistently outperforms all baselines across all tested models and hardware platforms. On Pixel 9, SANI achieves average latency reductions of 17.6%, 9.8%, and 12.0% compared to Native, AsyMo, and MNN, respectively. The performance gains are even more pronounced on other platforms, with average latency reductions of 23.7%, 14.4%, and 15.0% on the Find X3 Pro; 19.9%, 13.7%, and 15.0% on the Redmi K60; and 21.4%, 15.7%, and 16.7% on the OnePlus Ace. On the Odroid XU4 development board, the improvements are also substantial, with latency reductions of 19.0%, 15.5%, and 16.6% against the same baselines. These results demonstrate that SANI effectively addresses the performance collapse paradox in AMP architectures through its three integrated modules.

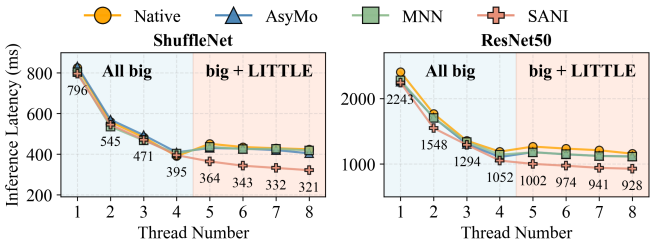
Model Difference. The performance gains vary across different model architectures. Compared to the Native baseline, SANI reduces average latency by 15.7% for MobileNetV2, 17.7% for SqueezeNet, 21.0% for ShuffleNet, 19.3% for ResNet-50, 29.5% for Inception-V3, and 16.1% for GPT-2. To understand this variation, we analyzed the potential for the dual kernel issue and dynamic scheduling. As shown in Fig. 11, models like ShuffleNet and Inception-V3 have a higher percentage of kernels that can be parallelized and benefit from dual-kernel implementations. This allows our affinity-aware kernel issuer to select core-affine kernels from the outset, while the adaptive granularity scheduler optimizes workload distribution at runtime, leading to significant performance gains. Furthermore, we observed that ResNet-50 exhibits the highest frequency of merge operations. This significantly reduces task acquisition overhead, contributing to its substantial performance improvement.

We take Inception-V3, which shows the most significant improvement, as an example for a more detailed analysis. As shown in Fig. 12(a), its performance is dominated by Permute kernels. A key challenge within Inception-V3 is that the Permute Kernel instance has a primary dimension of only 7, which severely limits the degree of parallelism. This results in poor scaling for baselines, as shown in Fig. 12(b). While AsyMo attempts to solve this with multi-dimensional partitioning, but creates excessively fine-grained workloads, which incur significant task acquisition overhead and thus diminish the parallelization benefits. SANI overcomes this with a more balanced approach. The affinity-aware kernel issuer's block unification strategy creates a pool of coarser-grained, manageable blocks. This provides sufficient parallelism to saturate the cores without introducing prohibitive overhead. Then, the adaptive granularity scheduler dynamically adjusts these block-level workloads , achieving a superior balance between wait and execution latency.

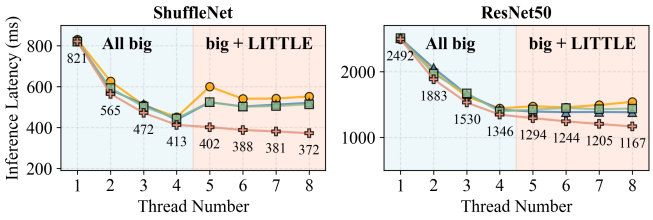
Hardware Difference. The performance variation across hardware platforms reveals a strong correlation between processor asymmetry and SANI's optimization potential. To quantify this, we benchmarked single-threaded performance using the `neon_sgemm` kernel from Arm-CL, revealing asymmetry ratios ranging from 2.6:1 to 3.4:1 across our testbeds. SANI's benefits are most pronounced on devices with higher asymmetry. For instance, it achieves a 23.7% average latency reduction on the Find X3 Pro (3.2:1 ratio) and 19.9% on the Redmi K60 (3.4:1 ratio), compared to a more moderate 17.6% on the Pixel 9 with its lower 2.6:1 ratio. This is because greater architectural asymmetry amplifies the inefficiencies of baseline schedulers, creating more optimization opportunities for SANI. This pattern suggests that SANI's advantages become increasingly valuable as processor asymmetry grows.

Scalability Analysis. Fig. 13 illustrates how SANI and baseline approaches scale across different thread numbers on ShuffleNet and ResNet-50. The results show different behaviors in symmetric configurations using only big cores versus asymmetric configurations incorporating LITTLE cores.

In symmetric scenarios, SANI offers improvements through its adaptive granularity scheduler that adaptively adjusts workload granularity based on runtime execution patterns. On Pixel 9 with ResNet-50 using two big cores, SANI reduces latency by 12.5% versus Native and 9.4% versus



(a) Google Pixel 9



(b) OPPO Find X3 Pro

Figure 13: Performance comparison with increasing thread number on ShuffleNet and ResNet-50.

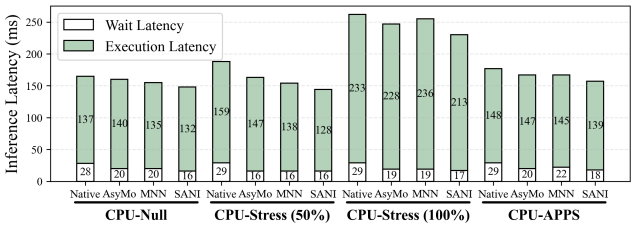


Figure 14: Performance of MobileNetV2 inference under varying system loads on Google Pixel 9.

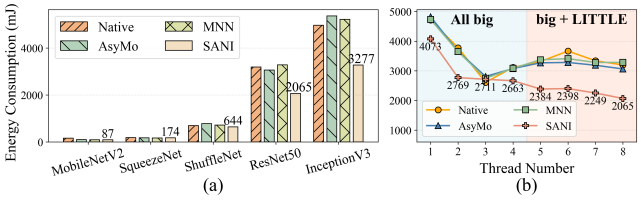


Figure 15: Energy consumption analysis on OPPO Find X3 Pro: (a) Comparison across different models using all cores, (b) Energy scaling behavior of ResNet-50.

AsyMo. This improvement primarily stems from the scheduler's ability to dynamically balance task sizes, optimizing the trade-off between execution efficiency and synchronization overhead rather than relying on fixed partitioning schemes.

The performance difference becomes more pronounced in asymmetric configurations where baseline implementations face the performance collapse paradox. When LITTLE cores are introduced, SANI shows improved performance across all tested models and devices through the coordinated operation of all three components. On Pixel 9 with ShuffleNet using four big cores plus one LITTLE core, SANI reduces latency by 19.3% versus Native and 15.3% versus AsyMo. As more LITTLE cores are added, SANI continues to show good scaling behavior, with performance improving on both devices. For ResNet-50, SANI achieves 19.4% latency reduction versus Native on Find X3 when using all available cores. In comparison, baseline approaches tend to show diminishing returns or performance degradation with additional LITTLE cores. This scaling efficiency highlights the synergy of SANI's modules. Beyond the gains from the adaptive granularity scheduler, SANI's advantage in asymmetric settings is magnified by its two other components. The affinity-aware kernel issuer selects the most efficient kernel for each core type, while the on-demand kernel switcher handles the necessary transformations for migrated workloads at runtime. This comprehensive strategy allows SANI to effectively utilize all cores and avoid the performance collapse paradox.

Dynamic loads. We evaluated SANI's robustness to runtime interference by running MobileNetV2 under four scenarios: idle system ("CPU-Null"), moderate CPU stress ("CPU-Stress (50%)") with 4 threads running `sqrt()` calculations, heavy CPU stress ("CPU-Stress (100%)") with 8 threads, and real application interference from YouTube video playback ("CPU-APPS"). We used a controllable load generator stress [32] to create compute threads. Fig. 14 presents the results decomposed into thread wait and execution latencies.

SANI consistently maintains lower wait times across all scenarios compared to baselines with 20%-40% reduction. This improvement stems from the adaptive granularity scheduler, which continuously adjusts task granularity based on measured thread execution speeds, thereby preventing performance bottlenecks caused by slow or blocked threads. The execution latency remains 7%-10% lower than baselines under interference due to our on-demand kernel switcher, which performs efficient kernel transformations when workloads migrate.

It should be noted that wait latency in moderate stress scenarios decreases slightly compared to idle conditions, as stress threads predominantly occupy big cores, effectively reducing the performance gap between asymmetric cores.

Energy Consumption. Fig. 15(a) demonstrates that SANI reduces energy usage across most models using all cores. For computation-intensive networks, SANI achieves significant improvements: 34.1%-35.3% versus Native, 32.5%-39.0% versus AsyMo, and 37.1%-37.3% versus MNN for ResNet-50 and Inception-V3. Lighter models show moderate gains with MobileNetV2 and ShuffleNet achieving 5.5%-30.5% reductions compared to baselines.

To analyze scaling behavior, Fig. 15(b) shows ResNet-50's energy consumption across thread configurations. Energy consumption decreases as thread number increases due to both reduced inference latency and the higher energy efficiency of

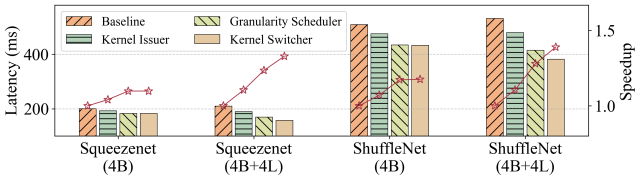


Figure 16: The overall performance breakdown of SANI.

LITTLE cores. Beyond the aggregate savings, these results clarify when the LITTLE cluster is worth using at all. For background or latency-tolerant DNN tasks, LITTLE-only execution remains a valid mode and is the most energy-favorable option. For interactive workloads, big-only execution leaves the LITTLE cores idle and forgoes their lower energy per operation, while naive all-core execution triggers the performance-collapse paradox of § 2.2. SANI's contribution is making the mixed regime both correct and efficient, so that the latency benefit of big cores and the energy benefit of LITTLE cores compound rather than cancel.

5.3 Ablation Study

We conducted experiments to evaluate the contribution of each SANI component to the overall performance improvement. Fig. 16 illustrates the incremental latency reductions achieved by sequentially enabling our three key techniques on SqueezeNet and ShuffleNet using different configurations.

Analysis on Symmetric Cores (4B). In the symmetric setting, adding the affinity-aware kernel issuer alone reduces latency by 4.0% on SqueezeNet and 6.7% on ShuffleNet. Although no dual-kernel selection is performed in this homogeneous environment, this improvement stems from its block unification strategy. Enabling multi-dimensional partitioning creates more parallelizable tasks than the baseline, especially for operators with challenging dimension sizes. Further enabling the adaptive granularity scheduler brings an additional 5.2% and 8.6% improvement, respectively. Even on homogeneous cores, runtime interference from background system processes can cause threads to execute at different speeds, and our scheduler's ability to dynamically adjust workload granularity is crucial for maintaining load balance. As expected, finally adding the on-demand kernel switcher provides negligible benefit (0.0%-0.2%), as no heterogeneous kernel transformations are triggered. These symmetric gains indicate that Arm-CL's single-kernel, fixed-partition pipeline leaves measurable parallelism on the table even when all cores are identical, with block unification recovering parallelism on awkward dimensions and the adaptive scheduler absorbing background interference. The same mechanisms that target AMP asymmetry thus continue to pay off in homogeneous configurations.

Analysis on Asymmetric Cores (4B+4L). In the asymmetric setting, the benefits of each module are magnified. The issuer's contribution increases to 9.5% on SqueezeNet and 9.6% on ShuffleNet, as it now performs dual-kernel issue, selecting the most hardware-affine kernel for both big and LITTLE cores. The scheduler becomes even more critical, delivering further improvements of 10.5% and 13.5% by effectively managing the significant performance disparities between the heterogeneous cores. Finally, the on-demand kernel switcher delivers a substantial 7.1% and 8.0% gain. When workload migration occurs, its ability to transform kernels on the fly prevents cores from executing a non-affine kernel. This study validates that all components are essential for achieving optimal performance, with their synergy being most pronounced in complex and asymmetric environments.

Table 3: Runtime overhead (ms) of SANI operations

Model	Async Prep	Merge & Split	Kernel Trans
MobileNetV2	3.8	2.2	0.08
Squeezenet	3.2	1.1	0.25
Shufflenet	11	2.2	0.18

5.4 Overhead Analysis

We analyze the overhead introduced by SANI in terms of both CPU latency and memory footprint.

CPU Latency. Table 3 quantifies the primary sources of SANI's runtime CPU overhead: the asynchronous preparation of dual kernels, the Merge and Split operations, and on-the-fly kernel transformations. The data shows that these operations are highly efficient. For example, on the complex ShuffleNet model, the cumulative overhead from these operations is well under 3 ms. This constitutes a small fraction of the model's total inference latency, which is typically in the hundreds of milliseconds. As demonstrated in our evaluation, the significant performance improvements delivered by SANI far outweigh this minimal computational overhead.

Memory Footprint. SANI introduces a minor memory overhead. This stems from two main sources: the additional working space required when the affinity-aware kernel issuer prepares a dual-kernel implementation for an operator, and the memory needed to store the pre-computed index maps for the on-demand kernel switcher. Evaluation results show that the peak additional memory footprint across all tested models is less than 2MB. Considering that typical neural network models require 60MB to 100MB of memory for execution, this additional footprint is negligible. That says SANI achieves its performance improvements without imposing a meaningful memory burden on the system. The footprint does not grow linearly with model depth. Index maps are kept per dual-kernel operator-shape pair, so repeated layer patterns such as stacked Conv-BN-ReLU blocks reuse the same map template. The peak therefore tracks the number of distinct operator shapes in the network rather than the raw layer count,

which is why even deep models stay well below the 2 MB ceiling.

5.5 Discussion

Our evaluation targets operator-level parallelizable workloads where mobile CPU execution is the dominant or fallback path, including the five CNN-family models and the GPT-2 transformer case study in Section 5.2. Large-scale on-device LLM inference is not the focus of this work. Our own profiling on a Google Pixel 9 shows that Qwen-1.5-4B sustains only 22.78 prefill tokens/s and 6.10 decode tokens/s on four big cores, far short of an interactive experience. Recent on-device NPU stacks such as llm.npu [56] report over 1000 tokens/s prefilling with substantially better energy per token, making NPUs the more suitable target for production-grade LLM serving on phones. The core abstractions of SANI, dual-kernel issue and adaptive granularity scheduling over a unified block interface, are operator-shape rather than model-class specific. The 16.1% latency reduction already observed on GPT-2 in § 5.2 comes from the same MatMul and GEMM kernels that dominate larger transformers, so we expect the design to carry over whenever CPU-side LLM execution becomes the right trade-off.

6 Related Work

Early on-device frameworks like TensorFlow Lite [28], Arm-CL [6], and NCNN [35] focused on general optimizations such as thread-level parallelism and cache affinity, primarily assuming core symmetry. As AMP architectures became prevalent, MNN [25] introduced a static task allocation strategy based on the compute power of different core types. AsyMo [49] advanced this with cost-model-guided block partitioning for asymmetric scheduling. However, these approaches remain static and fail to adapt to the dynamic mobile environment, where interference can alter a core's performance. In contrast, SANI's adaptive granularity scheduler adjusts workload sizes at runtime to maintain balance.

Workload scheduling on AMPs. While scheduling for DNN workloads on AMPs is an emerging area, general-purpose AMP scheduling has been widely studied [14, 16, 45, 46, 50]. Works like COLAB [59] and WATS [10] use predictive models and work-stealing to improve performance and load balancing. However, these OS-level schedulers are application-agnostic. They operate on coarse-grained OS threads, lacking the ability to partition work at the finer, operator-specific level. Furthermore, they cannot dynamically adjust the granularity of the workload. SANI overcomes these issues by being an application-aware framework that manages fine-grained workloads tailored to DNN operators.

Kernel selection optimizations. Kernel selection is critical for performance, and prior works have focused on making the best static choice. NeoCPU [31] adapts kernel parameters to different CPU architectures, NNV12 [58] selects kernels to optimize cold-start latency, and FlexNN [29] chooses implementations to maximize performance under memory constraints. The fundamental limitation of these approaches is that their selection is static, made once before and after execution. SANI addresses this with a two-part strategy: the affinity-aware kernel issuer prepares the best kernel for each core type, while the on-demand kernel switcher transforms workloads during migration, ensuring cores execute their most affine kernel.

7 Conclusion

In this paper, we explored the performance-collapse paradox in DNN inference on AMP architectures and introduced SANI, a scalable, asymmetry-aware framework. SANI combines three mechanisms—an affinity-aware kernel issuer, an adaptive granularity scheduler, and an on-demand kernel switcher—to preserve efficiency across heterogeneous cores. Evaluation on commercial SoCs shows that SANI scales efficiently, consistently outperforms existing approaches, lowers energy consumption, and sustains robust efficiency under interference across heterogeneous cores.

Looking ahead, we envision extending SANI's scalability principles to future mobile SoCs that integrate CPUs with GPUs and NPUs.

Acknowledgments

We sincerely thank our shepherd and the anonymous OSDI'26 reviewers for their insightful suggestions. This work was supported by the National Key Research and Development Program of China (Grant No. 2023YFE0205700), the National Natural Science Foundation of China (Grant Nos. 62341410 and 62302348), the General Program of the China Postdoctoral Science Foundation (Grant No. 2025M781507), and the Science and Technology Development Fund, Macao S.A.R (FDCT) projects 0078/2023/AMJ, 0056/2025/RIB2, 001/2024/SKL and SKL-IoTSC(UM)/ORP05/2025. Contact author at the University of Macau: Chuang Hu (chuanghu@um.edu.mo).

References

[1] Martin Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu,

and Xiaoqiang Zheng. Tensorflow: A system for large-scale machine learning. In *12th USENIX Symposium on Operating Systems Design and Implementation*, OSDI '16, pages 265–283. USENIX, 2016.

[2] Ahmed Abouelhamayed, Susanne Balle, Deshanand Singh, and Mohamed Abdelfattah. Beyond Inference: Performance Analysis of DNN Server Overheads for Computer Vision. In *Proceedings of the 61st ACM/IEEE Design Automation Conference*, DAC '24, pages 1–6. ACM/IEEE, November 2024.

[3] Gene M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the April 18-20, 1967, Spring Joint Computer Conference*, AFIPS '67, pages 483–485. ACM, April 1967.

[4] Apple. Apple unleashes m1, 2020.

[5] Arm. big.LITTLE: Balancing Power Efficiency and Performance, 2011.

[6] Arm. Compute library, 2017.

[7] Arm. Arm NN, 2018.

[8] Arm. Arm device tree bindings for cpu capacity, 2023.

[9] Baidu. Paddle-Lite, March 2025.

[10] Quan Chen, Yawen Chen, Zhiyi Huang, and Minyi Guo. WATS: Workload-Aware Task Scheduling in Asymmetric Multi-core Architectures. In *2012 IEEE 26th International Parallel and Distributed Processing Symposium*, IPDPS '12, pages 249–260. IEEE, May 2012.

[11] Tianshi Chen, Yunji Chen, Marc Duranton, Qi Guo, Atif Hashmi, Mikko Lipasti, Andrew Nere, Shi Qiu, Michèle Sebag, and Olivier Temam. BenchNN: On the broad potential application scope of hardware neural network accelerators. In *2012 IEEE International Symposium on Workload Characterization*, IISWC '12, pages 36–45. IEEE, November 2012.

[12] Long Cheng, Yan Gu, Qingzhi Liu, Lei Yang, Cheng Liu, and Ying Wang. Advancements in Accelerating Deep Neural Network Inference on AIoT Devices: A Survey. *IEEE Transactions on Sustainable Computing*, 9(6):830–847, November 2024.

[13] ONNX Runtime developers. Onnx runtime. https://onnxruntime.ai/, 2025. Version: 1.22.1.

[14] Hamza Djigal, Linfeng Liu, Jian Luo, and Jia Xu. BUDA: Budget and Deadline Aware Scheduling Algorithm for Task Graphs in Heterogeneous Systems. In *2022 IEEE/ACM 30th International Symposium on Quality of Service*, IWQoS '22, pages 1–10. IEEE/ACM, June 2022.

[15] Facebook. Caffe2, March 2025.

[16] Yu Feng and Yuhao Zhu. PES: proactive event scheduling for responsive and energy-efficient mobile web computing. In *Proceedings of the 46th International Symposium on Computer Architecture*, ISCA '19, pages 66–78. ACM, June 2019.

[17] Zhangxiaowen Gong, Houxiang Ji, Christopher W. Fletcher, Christopher J. Hughes, Sara Baghsorkhi, and Josep Torrellas. SAVE: Sparsity-Aware Vector Engine for Accelerating DNN Training and Inference on CPUs. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '20, pages 796–810. IEEE/ACM, October 2020.

[18] Google. Perfetto, 2018.

[19] Google. Google pixel 9, 2024.

[20] Lixiang Han, Zimu Zhou, and Zhenjiang Li. Pantheon: Preemptible multi-dnn inference on mobile edge gpus. In *Proceedings of the 22nd Annual International Conference on Mobile Systems, Applications and Services*, MOBISYS '24, page 465–478. Association for Computing Machinery, 2024.

[21] Wei Hao, Zixi Wang, Lauren Hong, Lingxiao Li, Nader Karayanni, AnMei Dasbach-Prisk, Chengzhi Mao, Junfeng Yang, and Asaf Cidon. Nazar: Monitoring and Adapting ML Models on Mobile Devices. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '25, pages 746–761. ACM, March 2025.

[22] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *2016 IEEE Conference on Computer Vision and Pattern Recognition*, CVPR '16, pages 770–778. IEEE, 2016.

[23] Andrew G. Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications. April 2017.

[24] Forrest N. Iandola, Song Han, Matthew W. Moskewicz, Khalid Ashraf, William J. Dally, and Kurt Keutzer. SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and <0.5MB model size. November 2016.

[25] Xiaotang Jiang, Huan Wang, Yiliu Chen, Ziqi Wu, Lichuan Wang, Bin Zou, Yafeng Yang, Zongyang Cui, Yu Cai, Tianhang Yu, Chengfei Lyu, and Zhihua Wu. MNN: A Universal and Efficient Inference Engine. In *Proceedings of Machine Learning and Systems*, MLSys '20, pages 1–13, March 2020.

[26] Shubham Kamdar and Neha Kamdar. big. LITTLE Architecture: Heterogeneous Multicore Processing. *International Journal of Computer Applications*, 119:35–38, June 2015.

[27] Seyyed Salar Latifi Oskouei, Hossein Golestani, Matin Hashemi, and Soheil Ghiasi. CNNdroid: GPU-Accelerated Execution of Trained Deep Convolutional Neural Networks on Android. In *Proceedings of the 24th ACM international conference on Multimedia*, MM '16, pages 1201–1205. ACM, October 2016.

[28] Juhyun Lee, Nikolay Chirkov, Ekaterina Ignasheva, Yury Pisarchyk, Mogan Shieh, Fabio Riccardi, Raman Sarokin, Andrei Kulik, and Matthias Grundmann. On-Device Neural Net Inference with Mobile GPUs. July 2019.

[29] Xiangyu Li, Yuanchun Li, Yuanzhe Li, Ting Cao, and Yunxin Liu. FlexNN: Efficient and Adaptive DNN Inference on Memory-Constrained Edge Devices. In *Proceedings of the 30th Annual International Conference on Mobile Computing and Networking*, MobiCom '24, pages 709–723. ACM, May 2024.

[30] Zhuojin Li, Marco Paolieri, and Leana Golubchik. A Benchmark for ML Inference Latency on Mobile Devices. In *Proceedings of the 7th International Workshop on Edge Systems, Analytics and Networking*, EdgeSys '24, pages 31–36. ACM, April 2024.

[31] Yizhi Liu, Yao Wang, Ruofei Yu, Mu Li, Vin Sharma, and Yida Wang. Optimizing cnn model inference on CPUs. In *2019 USENIX Annual Technical Conference*, ATC '19, pages 1025–1040. USENIX, 2019.

[32] m ric. stress-android, July 2024.

[33] Sandler Mark, Howard Andrew, Zhu Menglong, Zhmoginov Andrey, and Liang-Chieh Chen. Mobilenetv2: Inverted residuals and linear bottlenecks. March 2019.

[34] Monsoon. High Voltage Power Monitor, 2025.

[35] Hui Ni and The NCNN contributors. Ncnn, June 2017.

[36] Anant V. Nori, Rahul Bera, Shankar Balachandran, Joydeep Rakshit, Om J. Omer, Avishaii Abuhatzera, Belliappa Kuttanna, and Sreenivas Subramoney. Proximu$: Efficiently Scaling DNN Inference in Multi-core CPUs through Near-Cache Compute. December 2020.

[37] Anant V. Nori, Rahul Bera, Shankar Balachandran, Joydeep Rakshit, Om J. Omer, Avishaii Abuhatzera, Belliappa Kuttanna, and Sreenivas Subramoney. REDUCT: Keep it Close, Keep it Cool! : Efficient Scaling of DNN Inference on Multi-core CPUs with Near-Cache Compute. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture*, ISCA '21, pages 167–180. ACM/IEEE, June 2021.

[38] Odroid. Odroid xu4, 2015.

[39] OnePlus. Oneplus 12, 2025.

[40] OnePlus. Oneplus ace 5 ultra, 2025.

[41] OPPO. OPPO Find X3 series, 2021.

[42] PyTorch Team. Pytorch mobile. https://pytorch.org/mobile/home/, 2025. Version: 0.7.0.

[43] Alec Radford, Jeff Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. Language models are unsupervised multitask learners. *OpenAI Blog*, 2019.

[44] Aravind Srinivasa Raghavan. Heterogeneous Computing for your Demanding Apps, 2020.

[45] Bagher Salami, Hamid Noori, and Mahmoud Naghibzadeh. Fairness-Aware Energy Efficient Scheduling on Heterogeneous Multi-Core Processors. *IEEE Transactions on Computers*, 70(1):72–82, January 2021.

[46] Elham Shamsa, Anil Kanduri, Pasi Liljeberg, and Amir M. Rahmani. Concurrent Application Bias Scheduling for Energy Efficiency of Heterogeneous Multi-Core Platforms. *IEEE Transactions on Computers*, 71(4):743–755, April 2022.

[47] Christian Szegedy, Vincent Vanhoucke, Sergey Ioffe, Jon Shlens, and Zbigniew Wojna. Rethinking the inception architecture for computer vision. In *2016 IEEE Conference on Computer Vision and Pattern Recognition*, CVPR '16, pages 2818–2826. IEEE, 2016.

[48] Xiaohu Tang, Yang Wang, Ting Cao, Li Lyna Zhang, Qi Chen, Deng Cai, Yunxin Liu, and Mao Yang. LUT-NN: Empower Efficient Neural Network Inference with Centroid Learning and Table Lookup. In *Proceedings of the 29th Annual International Conference on Mobile Computing and Networking*, MobiCom '23, pages 1–15. ACM, October 2023.

[49] Manni Wang, Shaohua Ding, Ting Cao, Yunxin Liu, and Fengyuan Xu. AsyMo: scalable and efficient deep-learning inference on asymmetric mobile CPUs. In *Proceedings of the 27th Annual International Conference on Mobile Computing and Networking*, MobiCom '21, pages 215–228. ACM, September 2021.

[50] Siqi Wang, Gayathri Ananthanarayanan, Yifan Zeng, Neeraj Goel, Anuj Pathania, and Tulika Mitra. High-throughput cnn inference on embedded arm big.little multicore processors. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 39(10):2254–2267, September 2020.

[51] Xubin Wang, Zhiqing Tang, Jianxiong Guo, Tianhui Meng, Chenhao Wang, Tian Wang, and Weijia Jia. Empowering Edge Intelligence: A Comprehensive Survey on On-Device AI Models. March 2025.

[52] Carole-Jean Wu, David Brooks, Kevin Chen, Douglas Chen, Sy Choudhury, Marat Dukhan, Kim Hazelwood, Eldad Isaac, Yangqing Jia, Bill Jia, Tommer Leyvand, Hao Lu, Yang Lu, Lin Qiao, Brandon Reagen, Joe Spisak, Fei Sun, Andrew Tulloch, Peter Vajda, Xiaodong Wang, Yanghan Wang, Bram Wasti, Yiming Wu, Ran Xian, Sungjoo Yoo, and Peizhao Zhang. Machine Learning at Facebook: Understanding Inference at the Edge. In *2019 IEEE International Symposium on High Performance Computer Architecture*, HPCA '19, pages 331–344. IEEE, February 2019.

[53] Yuanpei Wu, Dong Du, Chao Xu, Yubin Xia, Ming Fu, Binyu Zang, and Haibo Chen. D-vsync: Decoupled rendering and displaying for smartphone graphics. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '25, page 326–341. ACM, 2025.

[54] Yunyi Xiang, Zheng Wu, Haidong Yao, Xiankui Xiong, and Fan Yang. Aries: A DNN Inference Scheduling Framework for Multi-core Accelerators. In *Proceedings of the 2024 5th International Conference on Computing, Networks and Internet of Things*, CNIOT '24, pages 186–191. ACM, July 2024.

[55] Xiaomi. Redmi k60, 2022.

[56] Daliang Xu, Hao Zhang, Liming Yang, Ruiqi Liu, Gang Huang, Mengwei Xu, and Xuanzhe Liu. Fast On-device LLM Inference with NPUs. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '25, pages 445–462. ACM, March 2025.

[57] Luting Yang, Bingqian Lu, and Shaolei Ren. A note on latency variability of deep neural networks for mobile inference, 2020. arXiv:2003.00138.

[58] Rongjie Yi, Ting Cao, Ao Zhou, Xiao Ma, Shangguang Wang, and Mengwei Xu. Boosting DNN Cold Inference on Edge Devices. In *Proceedings of the 21st Annual International Conference on Mobile Systems, Applications and Services*, MobiSys '23, pages 516–529. ACM, June 2023.

[59] Teng Yu, Pavlos Petoumenos, Vladimir Janjic, Hugh Leather, and John Thomson. Colab: A collaborative multi-factor scheduler for asymmetric multicore processors. In *Proceedings of the 18th ACM/IEEE International Symposium on Code Generation and Optimization*, CGO '20, pages 268–279. ACM/IEEE, February 2020.

[60] Xiangyu Zhang, Xinyu Zhou, Mengxiao Lin, and Jian Sun. ShuffleNet: An Extremely Efficient Convolutional Neural Network for Mobile Devices. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, CVPR '18, pages 6848–6856. IEEE, June 2018.

[61] Yue Zheng, Yuhao Chen, Bin Qian, Xiufang Shi, Yuanchao Shu, and Jiming Chen. A Review on Edge Large Language Models: Design, Execution, and Applications. *ACM Computing Surveys*, 57(8):1–35, February 2025.

Surviving the Impossible Trinity: Revisiting CPU Scheduling Problem on Modern COTS Mobile Devices (Operational Systems)

Jun Xiao[1,*], Qinhui Gu[1,*], Ligeng Chen[1], Lizhi Sun[1], Zicheng Wang[1], Yinggang Guo[1], Lu Liu[1], Hao Wu[2,✉], Borui Li[3,✉]

[1]*Honor Device Co., Ltd.* [2]*Nanjing University* [3]*Southeast University*

Abstract

CPU scheduling performance on mobile devices, especially under user interactions, is hindered by a fundamental semantic gap: the kernel scheduler lacks visibility into the user interaction context, treating latency-critical UI threads and background tasks equally. In this paper, we identify an impossible trinity, i.e., scarce prime cores, cross-process IPC dependencies, and tight latency deadlines, which exacerbate the mobile scheduling problem.

To survive the impossible trinity, we present MUSCHED, a semantic-aware scheduling framework for modern mobile devices that makes interaction capability a first-class scheduling objective. MUSCHED disentangles cross-process dependencies for critical threads along the interaction path and places these threads in a new VIP scheduling class between RT and CFS, allowing interaction-critical tasks to preempt normal background work without compromising system stability. Furthermore, MUSCHED proposes a scheduling policy plug-and-play mechanism that facilitates on-demand policy update in the user space without kernel recompilation for COTS mobile devices. In laboratory evaluations, MUSCHED reduces average application cold-start time by 14.8%. Furthermore, MUSCHED has been deployed on more than 20 million mobile devices since 2024. The deployment results show that MUSCHED reduces real-world startup anomalies by more than 30.7%. This deployment underscores the pivotal role of semantic-aware scheduling in achieving optimal mobile Quality of Experience.

1 Introduction

Mobile devices have evolved into the primary computing platform for billions of users [3], supporting workloads that range from immersive gaming [16] to high-fidelity multimedia processing [19, 24]. Unlike server workloads that prioritize aggregate throughput or tail latency [7, 11, 13, 20], the defining metric for mobile systems is *Quality of Experience* (QoE), which is dictated by the responsiveness and smoothness of user interactions. Whether launching an application or scrolling through a feed, users expect instantaneous feedback. The operating system's CPU scheduler acts as the critical arbiter in meeting these expectations, determining when and where tasks execute on the underlying hardware.

However, *achieving optimal interactivity on modern mobile platforms remains difficult*. Despite decades of optimization, mobile users still experience perceptible jitter, frame drops, and sluggish application launches [21, 25]. The root of this problem lies in a fundamental semantic gap: the kernel scheduler operates on low-level abstractions (threads and timeslices). It lacks high-level visibility into the user's interaction context. It cannot distinguish a critical thread rendering a UI animation from a background thread performing a routine backup, often treating them with "fairness" that is detrimental to interactivity. Designing a scheduler that bridges this gap faces three unique challenges that distinguish mobile systems from desktop or server environments.

1. Resource heterogeneity and constraints. Mobile System-on-Chips (SoCs) operate under strict power budgets and thermal envelopes. Consequently, high-performance execution resources are scarce; a typical modern SoC features a heterogeneous architecture with only one or two "prime" cores alongside several performance and efficiency cores. Unlike server environments where compute can be over-provisioned, the mobile scheduler must make zero-sum decisions to allocate these precious cycles. Existing scheduling heuristics such as ARM energy-aware scheduling (EAS) [23] infer task importance mainly from historical CPU usage, making them too slow and imprecise for short-lived interactive bursts under resource constraints.

2. Complex cross-layer dependency. Mobile applications are not monolithic; they rely on extensive system services. A single user touch event triggers a cascade of execution spanning the application process, the system server (e.g., *InputManager*), and the kernel. In Android, these interactions rely heavily on synchronous Inter-Process Communication

[*] Jun Xiao and Qinhui Gu contributed equally to this work.
[✉] Corresponding authors: Hao Wu and Borui Li.

(IPC) and lock-based synchronization. This creates deep and implicit dependency chains in which a high-priority UI thread may block while waiting for a lower-priority background service. This phenomenon is exacerbated by the scheduler's inability to see this cross-layer dependency. A delay in any link of the dependency chain propagates to the display, stalling the entire interaction.

3. Unforgiving latency deadlines. The window for processing user input is vanishingly small on mobile devices. Modern displays with 120Hz refresh rates demand that a new frame be rendered every 8.3ms. Furthermore, interactive workloads are characterized by extreme "burstiness": a touch event generates a sudden spike of short-lived, latency-critical threads (e.g., layout calculation, render commands) that must be serviced immediately. Neither CFS nor Real-Time (RT) classes can satisfy mobile latency deadlines safely. CFS prioritizes fairness over the immediate "unfair" preemption required by a touch event, leading to deadline misses during 120Hz rendering. Conversely, assigning static RT priorities is unsafe for the complex Android software stack; it cannot adapt to the rapidly changing context, where a thread is critical one moment and background noise the next, risking thermal throttling and system-wide starvation.

These three intertwined challenges are difficult to solve together, forming an *impossible trinity (a.k.a., trilemma)* for existing scheduling frameworks to navigate in this specific mobile landscape. Addressing this trilemma requires breaking the cross-layer fragmentation between application semantics and kernel scheduling, and moving beyond a one-size-fits-all policy toward adaptive scheduling decisions tailored to each application's interaction context.

However, this shift is far from straightforward. On the one hand, the scheduler must *capture the critical path* of an interaction and *disentangle its dependency chain* across threads, locks, and IPC transactions, so that urgency can be propagated end-to-end rather than stopping at a single runnable thread. On the other hand, such adaptivity must remain *practical for commercial off-the-shelf (COTS) devices*: policies must be updated safely, react quickly to changing runtime scenarios, and preserve system stability at production scale.

To solve this, we present MUSCHED, a semantic-aware scheduling framework designed for modern mobile SoCs. The core insight of MUSCHED is that *interaction capability should be a first-class citizen in scheduling*. Hence, we introduce a new "VIP" scheduling class that sits between RT and CFS, ensuring critical interactive tasks preempt normal background work without risking system stability. MUSCHED bridges the semantic gap between low-level kernel scheduler and high-level user interaction through two key mechanisms: *Scenario-Aware Annotation*, which identifies critical threads (e.g., Render, UI) based on runtime hooks; and *Priority Propagation*, which dynamically boosts the priority of any thread (even in other processes) holding a lock or handling an IPC transaction required by a VIP task. This ensures that the entire dependency chain of an interaction executes with high urgency. Moreover, MUSCHED fundamentally rethinks the division of labor between user space and kernel space. Instead of hardcoding policies into the kernel, MUSCHED proposes an eBPF-based scheduling policy plug-and-play mechanism to adapt the scheduling policy on demand in user space to break the one-fits-all problem for COTS mobile devices. This user-space design is practical and deployable, requiring no kernel recompilation to update policies.

We implement and evaluate MUSCHED under both controlled laboratory conditions and on a commercial smartphone powered by a Snapdragon 8 Elite running Android 15, as well as in large-scale production deployments. In laboratory tests across 10 representative apps, MUSCHED reduces average cold-start time by 14.8%. In production, deployed on over *20 million* devices since 2024, MUSCHED reduces cold-start anomalies by 30.7%, animation frame-drop anomalies by 24.9%, and swipe frame-drop anomalies by 35.7%, while maintaining system stability.

This paper makes the following contributions:

- We characterize the scheduling bottlenecks in mobile interactive workloads, identifying priority inversion and cross-process dependency as key sources of latency.
- We design MUSCHED, an extensible scheduler using eBPF that introduces a VIP priority class and a priority propagation mechanism for IPC and lock contention.
- We deploy MUSCHED on over 20 million commercial devices and show that user-space scheduling policies can be safely iterated on without kernel changes and deliver significant improvements across key interactive metrics.

2 Background and Preliminaries

This section details the unique characteristics of mobile interactive workloads, the heterogeneous hardware, and the current Linux scheduling policy used by Android. We highlight why existing mechanisms fail to bridge the semantic gap between high-level user expectations and low-level kernel schedulers.

2.1 Characteristics of Mobile Devices

Mobile Interactive Workloads. Unlike server workloads defined by aggregate throughput, mobile systems are driven by the user's perception of fluidity. Human sensitivity to visual latency is acute; research indicates that users can perceive delays as low as 100ms in discrete interactions (e.g., application launch) and jitter in animations if frame times exhibit high variance [18]. To maintain the illusion of fluidity, modern mobile displays operate at high refresh rates (60Hz-120Hz), imposing a hard deadline for frame rendering (16.6ms or 8.3ms). The Android graphics pipeline enforces this via the

vsync signal. Missing a vsync deadline results in a "jank" (i.e., frame drop), which directly degrades the QoE.

However, achieving this consistency is complicated by the workload's extreme variability. Mobile execution is event-driven and bursty: a period of idling can be instantly interrupted by a touch event, triggering a complex chain of computations. Furthermore, these workloads operate in a volatile environment, where network conditions (e.g., cellular handover) and background activities (e.g., notifications, updates) introduce unpredictable interference.

Heterogeneous SoCs and Power Constraints. Mobile System-on-Chips (SoCs) employ heterogeneous multiprocessing (HMP) architectures, such as ARM big.LITTLE or DynamIQ, to balance performance and battery life. A typical high-end SoC comprises three tiers of cores: (1) *Prime Core:* High frequency and out-of-order execution capacity for single-threaded bursts. (2) *Performance Cores:* Balanced cores for sustained heavy lifting. (3) *Efficiency Cores:* Low-power in-order cores for background tasks. The scheduler's placement decision is critical: placing a UI thread on an efficiency core causes jank, while running background tasks on the prime core wastes power. Moreover, these decisions are constrained by strict thermal envelopes. The system may cap frequencies or disable cores entirely to prevent overheating, thereby dynamically adjusting hardware capacity.

2.2 Status Quo Scheduler in Android

In this paper, we focus on the scheduling mechanism on Android, a widely adopted, open-source mobile operating system. Android relies on the Linux kernel's scheduling framework, augmented with mobile-specific policies. The kernel manages tasks using priority-based scheduling classes.

Priority Hierarchy. Linux maintains a global priority scale ranging from 0 to 139, where lower values denote higher priority. On Android, this priority scale is divided into two primary classes:

1. **Real-Time (RT) Class (0–99):** Threads in this range preempt all non-RT tasks. To avoid starving the system, Android grants RT privileges sparingly, typically reserving them for strictly latency-sensitive pipelines such as audio processing and the display compositor (i.e., `SurfaceFlinger`).

2. **Completely Fair Scheduler (CFS) Class (100–139):** This is the default class for the vast majority of application threads, including the UI thread. Priorities in CFS are mapped to "nice" values ranging from -20 (priority 100) to +19 (priority 139). CFS aims to distribute CPU time fairly among tasks based on their weights (derived from nice values) using a Red-Black tree to track the virtual runtime (`vruntime`).

Energy Aware Scheduling (EAS). Android augments CFS with the EAS policy. EAS uses an energy model to estimate the power impact of assigning a task to a specific core cluster. It relies on task load-tracking mechanisms, primarily Per-Entity Load Tracking (PELT) or Window-Assisted Load Tracking (WALT), to predict future CPU demand using historical execution times.

Android User-Space Policy. To bridge the gap between application semantics and kernel scheduling mechanisms, the Android framework dynamically adjusts the priorities of application threads based on the app's lifecycle state (e.g., Foreground, Background, Top-App). It employs two main knobs. (1) *Cgroups (cpusets):* Android segregates tasks into cgroups such as `top-app` (access to all cores), `foreground` (limited access), and `background` (restricted to efficiency cores). (2) *Nice Values:* The framework adjusts the nice values of threads. For instance, the UI thread of the currently focused app is typically assigned a favorable nice value (e.g., -10), while background threads are penalized (e.g., +10).

2.3 Limitations of the Status Quo

Despite these layered optimizations, the "battle of the schedulers" [5] continues on mobile devices because standard heuristics are fundamentally misaligned with interactive needs due to the following two reasons.

Reactive vs. Proactive. EAS and CFS are reactive; they rely on historical average observations through the PELT or WALT mechanism to ramp up frequency or migrate tasks. Such decisions are made from local signals, such as a task's recent CPU utilization or the current load of a core, rather than from the end-to-end structure of a user interaction. However, a mobile interaction is a short causal chain spanning UI, rendering, system services, Binder calls, and locks. A critical thread may spend most of its time blocked on another task and therefore appear lightweight to utilization-based heuristics, while the remote-side delay is already consuming the same frame or launch deadline. Consequently, boosting frequency or migrating a task after its load has become visible cannot recover the waiting time already accumulated along the interaction path; by the time the scheduler reacts, the end-to-end deadline may have already been missed.

Lack of Cross-Process Visibility. Android applications are heavily componentized, relying on synchronous IPC to communicate with system services, e.g., the Binder mechanism in Android. While Android can adjust the priority of the calling thread, the kernel scheduler is unaware of the semantic dependency chain. Standard priority inheritance mechanisms, such as those for local mutexes, do not automatically extend to complex IPC transactions or to cross-cgroup boundaries. Consequently, a high-priority UI thread often blocks waiting for a service thread that is stuck in a restricted cgroup or running with a low CFS priority. Linux implements priority inheritance for local futex and rt_mutex contention, but this mechanism does not extend to the Binder IPC path or across cgroup boundaries, leaving a common and consequential case

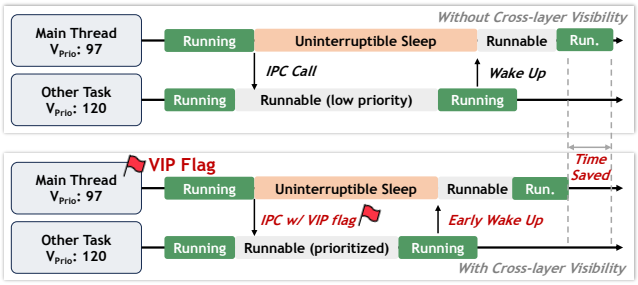


Figure 1: Motivating example: a main thread hands off an IPC request to a lower-priority task. OS scheduler lacks visibility into this dependency, leading to excessive waiting time.

of priority inversion unresolved in the stock kernel.

Figure 1 illustrates a common priority-inversion pattern in mobile interactions. Many latency-sensitive application tasks do not execute in isolation, but depend on cross-process communication with other system components. When the remote-side task of such an IPC path experiences a long scheduling delay, the caller thread is forced to wait as well, even if the caller itself has a much higher priority.

The example in Figure 1 shows this effect using a high-priority main thread (i.e., lower `prio` value) and a lower-priority dependent task (i.e., higher `prio` value). Before optimization, the main thread runs, then enters uninterruptible sleep while waiting for the dependent task to finish. However, the dependent task remains in the runnable state for an extended period before obtaining CPU time. As a result, the main thread cannot be woken up promptly, and its end-to-end waiting time is amplified by the remote-side scheduling delay.

This problem is especially harmful to Android graphics critical paths. For example, when SurfaceFlinger or another graphics-related thread issues a Binder request, a delayed remote-side service thread slows down the return of the IPC, which in turn delays layer composition and increases the likelihood of frame drops and visible jank. After optimization, MUSCHED marks the waiting main thread as VIP and propagates this urgency to the dependent task, thereby allowing the remote-side runnable interval to shrink and the main thread to wake up earlier.

3 System Overview of MUSCHED

Addressing the impossible trinity among scarce resources, cross-process dependencies, and tight deadlines requires the scheduler to understand each application's interaction structure: which threads lie on the critical path of user-visible operations, when these threads should be prioritized, and which cross-process dependencies they rely on. MUSCHED follows a split architecture that separates semantic policy decisions from scheduling implementation, as shown in Figure 2. The user-space layer captures interaction semantics and updates

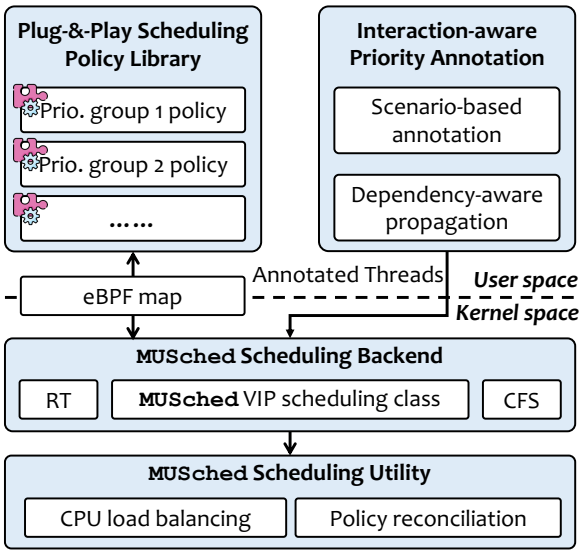


Figure 2: System overview of MUSCHED.

policies, while the kernel-space layer enforces these decisions on the scheduler fast path. This separation allows MUSCHED to adapt policies for different applications and interaction scenarios without recompiling the kernel or shipping a full firmware update.

In user space, a *plug-and-play scheduling policy library* stores scenario-specific policies, such as the task types to promote and the lifetime of each VIP promotion. These policies are loaded, unloaded, and updated through an eBPF-based gateway, which provides a narrow interface between policy management and kernel enforcement. VIP scheduling also relies on *interaction-aware priority annotation*: the user-space controller observes framework-level events, identifies key interactive threads, and marks them as candidates for MUSCHED's VIP scheduling class. Threads that are not annotated continue to follow the original Linux scheduling path. To handle dependencies that are not visible from a single thread alone, MUSCHED further introduces *dependency-aware propagation*, which carries VIP urgency across Binder calls and lock dependencies when a critical thread is blocked by another task.

In kernel space, MUSCHED introduces a VIP scheduling class between the RT and CFS classes. This class is implemented with per-core VIP queues that host threads requiring temporary priority elevation during user-visible interactions. During scheduling, RT tasks retain the highest priority; VIP tasks are served before normal CFS tasks only when no RT task is runnable. To preserve system stability under contention, a *load balancer* prevents VIP and RT tasks from accumulating on a single CPU core, while a *policy reconciliation agent* resolves conflicts among user-space policies and bounds their impact on existing RT and CFS policies.

4 MUSCHED Design

In this section, we introduce the essential building blocks of MUSCHED.

4.1 Kernel-space Scheduling Backend

To support interaction-aware prioritization on COTS mobile devices, MUSCHED introduces a VIP scheduling class between the built-in RT and CFS classes. The actual scheduling policy is built on top of eBPF to facilitate on-demand modification in user space. To keep this VIP scheduling class compatible with existing classes, MUSCHED also proposes a policy reconciliation module and a CPU load-balancing module, which are combined to maximize CPU utilization and task throughput.

Non-intrusive scheduling class. As we noted in Section 2, on Android OS, RT scheduling is used for system-defined critical tasks such as sensor management and system UI rendering, while other user-space non-critical tasks use the CFS strategy for fairness. As Android applications grow more complex, with multi-threaded rendering pipelines and application-specific critical threads, the two existing scheduling classes prove insufficient. The RT class is reserved for system-critical tasks such as audio and SurfaceFlinger; placing application threads there would risk starving these system services. CFS, while safe, weights threads by their historical CPU utilization, which does not reflect interaction criticality: a UI thread that has been idle waiting for a Binder reply appears less CPU-intensive than a background computation thread, and is consequently assigned lower priority. We therefore need *a scheduling class whose effective priority lies above CFS but below RT*, and whose priority decisions are informed by interaction semantics.

Priority conflict reconciliation. To avoid unnecessary CPU resource contention between VIP and RT tasks, we first try to prevent RT and VIP tasks, as well as multiple VIP tasks, from being placed on the same core during CPU selection. When a task is woken up, the scheduler first determines which CPU it should be executed on. At this point, the scheduler attempts to find a suitable CPU using the following strategy. First, it tries to find a CPU core that is idle and places the task in that CPU's local queue. If no CPU is currently idle, it tries to find a CPU whose task queue contains no RT or VIP tasks. If neither condition can be satisfied, the scheduler selects the CPU whose task queue contains no RT tasks and has the fewest VIP tasks.

CPU load balancing. Under heavy-load conditions, it is often unavoidable that multiple VIP and RT tasks are placed in the same CPU's task queue. Given that the completion times of various tasks differ, we further design a CPU load-balancing strategy, depicted in Figure 3, to fully utilize multiple CPU cores.

First, whenever a CPU enters the idle state, load balancing is triggered. The scheduler attempts to search for and pull tasks from other CPUs' task queues based on the following rules. It traverses the task queues of other non-idle CPUs and first tries to select those that contain both RT and VIP tasks. If none are found, it then selects the queue with the largest number of VIP tasks. From the selected queue, it chooses the first VIP task eligible to run on the idle CPU and migrates it to that CPU.

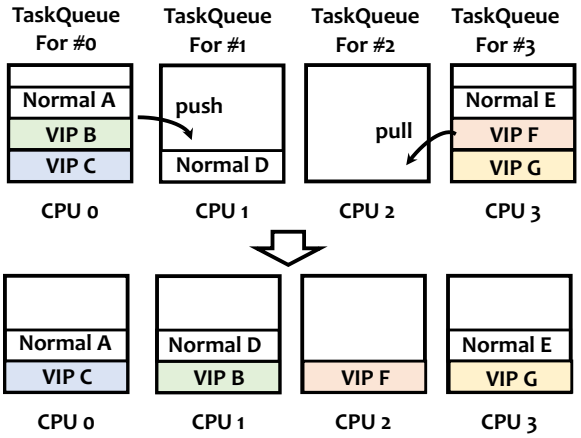


Figure 3: CPU load balancing in MUSCHED that balances both normal and VIP tasks on mobile CPUs.

In addition, on every CPU tick, the scheduler proactively checks the execution status of VIP tasks in the task queue. If it detects a VIP task that has remained runnable for too long, it will also attempt to balance the workload. More specifically, if the currently running task is an RT task and there exists a VIP task that has been runnable for a long time, the scheduler will actively select a new CPU during the tick. A 4 ms threshold is used to determine whether a VIP task has remained runnable for too long. If such a task exists, load balancing is also triggered: the scheduler traverses the CPUs where the task can be executed and selects the target CPU with no RT tasks and minimal VIP tasks. We set the migration threshold to 4 ms, which is approximately half of the 8.3 ms frame budget at 120 Hz. If a VIP task has already remained runnable for more than half a frame interval, further delay on the current CPU materially increases the risk of a missed frame deadline. Conversely, thresholds below 2 ms trigger excessive migrations, because transient run-queue delay of 1-2 ms is common even under moderate load, and the migration cost itself becomes non-negligible. In practice, this parameter is also influenced by hardware capability and product goals. For flagship or performance-oriented devices, we use lower thresholds, whereas we set higher thresholds to reduce frequent migration overhead for thermally constrained or battery-oriented products.

4.2 Plug-and-Play Policies based on eBPF

Different applications and interaction scenarios, such as app start and window switch, require different VIP time limits and dynamic promotion of selected tasks to VIP status. Moreover, these policies must be updated as applications evolve. To implement VIP scheduling policies for each application, traditional approaches that modify the kernel scheduler require intrusive changes to the kernel. This intrusive design is impractical for production-ready mobile devices because modifying the kernel requires frequent reboots to apply the new scheduling policy. MUSCHED adopts a fine-grained scheduling policy for each application, which can be updated when the applications are updated. Moreover, due to the variety of mobile applications, it is necessary to decide the scheduling policies on the fly for unseen applications. Hence, it is not possible for a large-scale mobile device vendor to publish policy patches that frequently require a system reboot.

According to the above analysis, MUSCHED therefore adopts an eBPF-based design because eBPF uniquely combines safe in-kernel programmability, rich observability, and efficient user–kernel communication. Specifically, eBPF facilitates MUSCHED to dynamically update the scheduling policy without a system reboot, which is necessary for application updates and new application installations.

MUSCHED implements the VIP scheduling class on top of Linux `sched_ext`. The kernel-side backend maintains per-core VIP queues, while the user-space scheduler updates scenario-specific policies through BPF maps and dispatch queues.

During scheduling, each CPU first serves RT tasks. If no RT task is runnable, it selects tasks from the local VIP queue before falling back to CFS tasks, ensuring that interaction-critical threads receive bounded priority elevation.

To prevent starvation of certain tasks under high load, tasks in the VIP queue are scheduled in a first-in-first-out (FIFO) manner, and each VIP task in the queue is assigned a 3 ms time slice. For the `cpu.share` allocation in cgroups, MUSCHED by default inherits the kernel's configuration. In addition, to prevent other normal tasks from starving when there are too many VIP tasks, a time limit is imposed on VIP tasks. For each VIP task, if it has used up its time slice but has not exceeded the limit time, it is reinserted into the back of the queue; if it runs out of both the slice time and the limit time, its VIP label is temporarily removed until the next enqueue. Since different types of VIP tasks have different computational requirements, we configure the time limit based on profiling of representative workloads: 20 ms for audio tasks, 10 ms for video tasks, 120 ms for webview tasks, and 20 ms for display tasks. These limit times reflect the computational profile of each interaction class and also serve as the main safeguard against starvation. Audio tasks use a 20 ms budget to match typical buffer-processing windows; video tasks use 10 ms to cover a decode-and-render cycle; WebView tasks use 120 ms because page layout and script execution often span multiple frames; and display tasks use 20 ms to align with composition-critical work. Once a task exhausts its limit, it is *demoted from VIP and returns to CFS* until the next qualifying event, ensuring that priority boosting remains temporary and bounded rather than allowing indefinite VIP service.

Table 1: Annotated scenarios related to user interaction.

Task Type	Examples
Animator	splashworker; wmshell.main; wmshell.anim
UI	Main thread; UI thread
Render	Render thread
WebView	CrRendererMain; Chrome_InProcRe

4.3 Semantic-aware Mobile Scheduling

Another key problem we face is identifying critical tasks currently queued in the CFS scheduler. To address this, we first introduce scenario-aware annotation, which provides guidance for labeling critical tasks based on a predefined list. Furthermore, considering that VIP tasks may depend on the completion of some normal tasks, we design a priority propagation for inter-process interaction. It dynamically propagates VIP tags to the normal task that a VIP task depends on, accelerating its execution and preventing bottlenecks caused by such dependencies.

Scenario-aware Annotation. To identify which threads belong to the critical path of user interactions, MUSCHED combines three complementary sources of information. First, Android already exposes several thread roles that consistently lie on interaction-critical paths across applications, such as the main/UI thread, RenderThread, MotionThread, and Binder worker threads serving user-facing requests. These roles generalize well across applications and therefore form the default annotation candidates in MUSCHED.

Second, we refine this default set through offline profiling. For each target scenario, such as app launch, swipe, or animation, we repeatedly execute the application and collect multiple systrace traces. From these traces, we analyze wake-up relationships and per-thread CPU load to identify additional threads that repeatedly appear on the end-to-end critical path. This step captures application-specific critical threads as a dependency graph, which is not directly exposed as standard Android roles.

Third, we further use jank traces collected from beta users in the field to diagnose missed cases and augment the annotation set. These online traces help us cover rare or highly application-specific behaviors that may not appear in controlled offline runs. Together, these three sources provide broad practical coverage of critical threads while keeping the annotation process lightweight.

These analyses yield the thread categories shown in Table 1.

Some categories, such as render thread and Binder worker pools, generalize across applications, while others require per-application identification of the concrete thread instances. At runtime, MUSCHED identifies these instances through framework hooks placed at SystemUI, Launcher animation transitions, app foreground/background switches, focus changes, and frame-rendering callbacks. When the framework detects that an application has entered a user-interactive state, it tags the corresponding threads with VIP labels so they can be scheduled with elevated priority.

Priority Propagation for Inter-Process Interaction. To optimize the waiting time of VIP tasks for locked resources, we design a VIP lock-waiting queue optimization strategy. In the native Linux mechanism, mutexes are awakened in a FIFO manner, while read–write locks are awakened in batches based on the top waiter type, also following a FIFO strategy. Since both regular tasks and VIP tasks may wait for the same lock resource, VIP tasks may experience long delays. To address this issue, we introduce a new wake-up strategy that allows VIP threads to bypass the strict FIFO sequence, enabling them to be inserted ahead of multiple non-critical threads and be awakened preferentially.

Considering that VIP tasks may depend on the completion of some normal tasks, we further design a call-chain priority propagation mechanism for inter-process interaction. When a VIP thread waits for a lock held by a normal thread, including futex, mutex, and rwsem, the VIP label can be propagated to that normal thread, elevating its priority to accelerate execution and shorten the VIP thread's waiting time. The core idea is that when a thread is blocked while acquiring a lock, it passes the thread ID of the lock-holding thread to the kernel. The kernel identifies whether the blocked thread is a VIP thread; if so, the VIP priority is dynamically propagated to the lock holder to accelerate its execution. For each supported synchronization primitive, we place hooks on the lock-acquisition and release paths to record the current owner thread ID in kernel-visible state. When a waiter blocks, the hook retrieves the recorded owner, checks whether the blocked task is tagged VIP, and, if so, transiently propagates the VIP label to the lock holder. A thread that merely acquires a lock is not proactively marked VIP; propagation is triggered only when a VIP waiter appears and identifies that holder as the blocking dependency. We apply this mechanism to futex, mutex, and rwsem contention, and revoke the inherited label when the owner releases the lock, when the dependency disappears, or when the boost exceeds its bounded lifetime. Once the regular thread releases the lock, it wakes the VIP thread, and the inherited VIP label is removed, restoring it to a normal task and returning to the fair CFS scheduling policy.

Besides, in Android, inter-process communication is also very common. When a task initiates a synchronous Binder call, it enters the sleeping state and waits for the remote process to return the result. If the remote process is under heavy load or its priority is low, the server binder may remain in the Runnable state for a long time, causing the caller task to be blocked. This issue becomes even more severe in scenarios such as layer composition in SurfaceFlinger.

We address this issue by monitoring the start of Binder transactions. When a VIP task communicates with another process through Binder, MUSCHED dynamically propagates the VIP label to the callee to accelerate its execution. Specifically, when a VIP thread initiates a synchronous Binder call, the scheduler immediately locates the corresponding service thread on the remote side and tags it as VIP. After the execution, the dynamically propagated label is cleared, and the task is restored to a normal task.

4.4 Case Study: Cold-start Acceleration

We take a cold-start acceleration scenario to demonstrate MUSCHED's user-space scheduling capability. Cold start is the most demanding application launch state [1]. It occurs when an app starts from scratch, with no existing process in memory. The system must create the app process, initialize the runtime, and build the first Activity before any frame can be shown. This procedure often triggers more than 300 thread creations, large memory allocations from hundreds of MB to several GB, and intensive file and network I/O. During this phase, many launch-critical threads become active, including UI, rendering, animation, web loading, composition, and system service binder threads, creating a short but extremely congested window of resource contention. Launch latency thus becomes a first-order performance objective, and standard startup metrics (e.g., TTID and TTFD) are highly sensitive to scheduling decisions. When cold-start time exceeds about one second, users typically perceive the app as sluggish. Traditional CFS schedulers prioritize fairness across tasks. Under heavy contention, this fairness causes critical-path threads on the launch pipeline to be preempted by less important work. Moreover, the kernel scheduler lacks semantic knowledge of the launch sequence and cannot distinguish threads on the cold-start critical path from best-effort background tasks.

Semantic-aware scheduling of MUSCHED addresses these limitations by injecting launch-specific knowledge into the scheduler. Sampling cold-start critical path offline [17], we annotate launch-critical tasks and add lightweight framework hooks to tag their threads as VIP. VIP threads receive elevated priority and are protected from preemption by non-critical work, directly mitigating CPU contention on the launch critical path. However, critical tasks frequently block on dependencies such as locks and binder IPCs. To handle this, semantic-aware scheduling incorporates priority propagation. When a VIP task waits on a lock, its VIP tag is propagated to the lock holder, temporarily boosting that thread to accelerate lock release and prevent priority inversion. Similarly, when a VIP task issues a synchronous Binder call, the scheduler locates the serving thread in the remote process and dynamically elevates its priority. The boosted priority is revoked once the lock

is released or the IPC completes, restoring normal fairness. By aligning priorities with cold-start semantics, MUSCHED reduces preemption and blocking delays and accelerates the launch pipeline, yielding significantly lower startup latency under load.

5 Implementation Details

Based on the MUSCHED design, we then describe how we facilitate these policies, with a focus on details specific to commodity Android devices.

Hooks and Maps based on sched_ext. sched_ext is an eBPF-based scheduler that allows users to implement scheduling classes using flexible BPF programs, thereby optimizing scheduling policies for specific workloads and scenarios. We implement the kernel scheduling behavior described in Section 4 via the hooks in the `sched_ext_ops` structure, and construct VIP task queues using BPF maps.

The key hooks of MUSCHED include CPU selection, enqueue, dispatch, and polling. `sched_select_cpu` selects a CPU based on energy efficiency when a task is woken up. It iterates over the system's performance cores, first preferring big cores in the idle state; if none are idle, it selects the least loaded big core, ensuring sufficient compute resources for VIP tasks. `sched_enqueue` assigns a task to the VIP queue of the corresponding task CPU, and invokes `scx_bpf_dispatch` to directly dispatch it to the target CPU core's high-priority dispatch queue (PRIO-DSQ). While `sched_dispatch` consumes tasks from the queue by placing them into the local DSQ, i.e., it determines from which DSQ the CPU fetches the next task to run. To achieve the load balancing described in Section 4.1, MUSCHED performs priority-based polling. `scx_bpf_consume(DSQ_ID_VIP)` first attempts to run tasks from the local VIP queue; if the local queue is empty, it then tries to consume tasks from the neighboring core's `DSQ_ID_VIP`, which we refer to as *Steal*.

MUSCHED creates a VIP task queue for each core via the `scx_bpf_create_dsq`. We use a per-CPU array eBPF map, named `cpu_contexts`, to store per-core state, including idle time, current load information, the number of enqueued tasks, CPU halt state, etc. We also use a task storage eBPF map, named `task_contexts`, to store per-task information such as weight, current execution state, compute load, enqueue time, and dequeue time. Together, these two maps maintain the metadata required by the custom scheduler.

Deploying eBPF on Mobile Devices. Deploying eBPF on commodity Android devices faces constraints that differ significantly from those in the upstream kernel community. The Android eBPF compilation and loading pipeline is tightly coupled to specific system libraries and SELinux policies. The system only allows eBPF programs to be loaded during boot by a dedicated process, `bpfloader`, and prohibits loading any new programs after boot completion [2]. Besides, the eBPF libraries provided by Android differ from community-standard libraries in both interfaces and evolution pace. To ensure implementation consistency and portability, we abandon Android-specific eBPF libraries and instead develop with the community-standard eBPF libraries. However, the existing `bpfloader` does not support the `BPF_MAP_TYPE_STRUCT_OPS` type required by sched_ext. To address this limitation, we extend `bpfloader` to recognize and load `struct_ops`-related objects during the boot phase, thereby providing the necessary foundation for subsequently activating the scheduler.

Table 2: Applications used in the laboratory evaluation, selected from top-ranked applications in the application store.

Category	Application	Package Name
Short Video	TikTok	`com.zhiliaoapp.musically`
	Douyin	`com.ss.android.ugc.aweme`
	Kuaishou	`com.smile.gifmaker`
Social Media	Instagram	`com.instagram.android`
	Twitter/X	`com.twitter.android`
Shopping	SHEIN	`com.zzkko`
	Jingdong	`com.jingdong.app.mall`
Video	YouTube	`com.google.android.youtube`
Navigation	Baidu Maps	`com.baidu.BaiduMap`
News	Toutiao	`com.ss.android.article.news`

Enabling sched_ext. sched_ext leverages the eBPF `struct_ops` mechanism, whose enabling method differs from traditional kprobe or tracepoint attachments. The scheduler is enabled by registering and updating a BPF map of type `BPF_MAP_TYPE_STRUCT_OPS`, rather than simply attaching a program to a specific hook. Given the stringent stability requirements of the scheduler, this `struct_ops` map is restricted to a single update, and any modification requires destroying and recreating the map. The map also maintains an enum state with values `INIT`, `INUSE`, `TOBEFREE`, and `READY` to indicate the scheduler state, preventing transient intermediate states from causing system panics. In the Android environment, we do not unload the sched_ext program itself; instead, we indirectly drive the above state transitions via `BPF_F_LINK` [4]. This design enables secure activation and reliable operation of the sched_ext scheduler while adhering to the safety constraint that eBPF programs should be loaded only once during boot.

6 Evaluation

In this section, we first present the laboratory evaluation results of MUSCHED on commodity Android smartphones.

Evaluation Setup. We deploy the proposed approach on the Magic 7 smartphone, which equips the Snapdragon 8 Elite chipset featuring 2 super CPU cores (@4.32 GHz) and 6 performance CPU cores (@3.53 GHz). The software platform

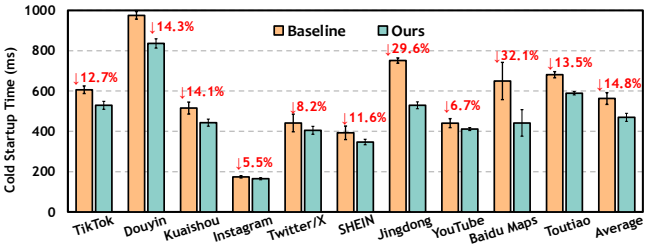


Figure 4: Cold-start time of 10 Applications.

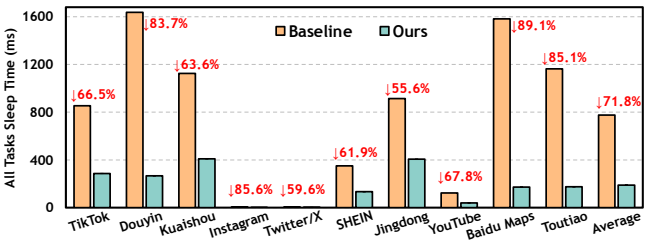


Figure 5: Time in sleep state during cold-start.

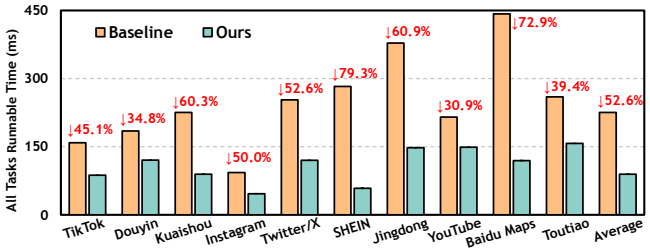


Figure 6: Time in runnable state during cold-start.

Table 3: Measured latency (in ms) of a foreground application with a background picture-in-picture video call with and without MUSCHED.

Foreground App.	Test Scenario	w/o	w/	Diff
Taobao Browsing	Application Cold Start	149.7	135.0	-9.8%
WeChat Moments	Application Cold Start	163.5	126.3	-22.8%
Douyin Live	Scroll 10 Live Room	449.0	386.9	-13.8%
News Browsing	Scroll Top News Article	584.5	480.5	-17.8%

is MagicOS 9, which is based on Android 15 with a Linux kernel version of 6.6.

Benchmarks. We choose 10 representative mobile applications, selected from the top-ranked applications in the application store, as test cases in our laboratory environment, as shown in Table 2. These applications cover commonly used categories such as short video, news, shopping, and navigation, etc. We use the native Android scheduling policy as the baseline, i.e., some predefined tasks are configured as RT task, while others are configured as normal tasks, and evaluate the cold-start performance of 10 applications. Each application is tested 100 times, and the average result is reported. To create reproducible yet realistic contention, we combine application-driven and synthetic background load. We emulate realistic concurrent usage by keeping representative background activities active while measuring the foreground app, e.g., short-video scrolling in the foreground together with navigation, music playback, or other common services in the background. we use tools such as `stress-ng` or `rt-app` to inject periodic per-core pressure. Unless otherwise stated, we fix the display mode, thermal state, battery mode, CPU governor, and cache-clearing procedure across runs so that the measured differences primarily reflect scheduling policy rather than platform drift.

End-to-end Performance. The cold-start performance of 10 applications is shown in Figure 4. Compared with the baseline, we reduce the average cold-start time by 14.8%, while also achieving more stable performance with the standard deviation decreasing by 24.25%. These improvements primarily stem from two aspects. First, we identify critical tasks during application startup and raise their scheduling priority, while preventing competition for CPU resources between VIP tasks and RT tasks. Second, our priority-propagation strategy temporarily boosts the priority of normal tasks that VIP tasks depend on, thereby reducing overall task completion time.

We further measure the total uninterruptible sleep time and waiting for scheduling time, i.e., in runnable state, of all VIP tasks during the startup process for the 10 applications. The results are shown in Figure 5 and 6, respectively. These results indicate that MUSCHED reduces both the resource-waiting time and scheduling-waiting time of critical tasks during application startup. This result is consistent with our analysis of why application startup time decreases.

We further evaluate MUSCHED in edge scenarios where a background picture-in-picture (PiP) video call coexists with foreground user interactions. Table 3 summarizes four representative cases, including application cold start in Taobao and WeChat Moments, live-room interaction in Douyin, and article browsing in a news application. Across these scenarios, MUSCHED consistently reduces foreground response latency by 9.8%–22.8% in all cases, with the largest gain observed in WeChat Moments. Overall, these results suggest that MUSCHED remains effective under mixed foreground/background workloads, while also highlighting that semi-interactive scenarios with concurrent media activity still expose cases that require finer-grained policy reconciliation, as we designed in Section 4.1.

Overhead of MUSCHED. We further evaluate the runtime overhead introduced by the scheduling backend of MUSCHED and its dependency-tracking path. We measure the overhead in two widely adopted scenarios: short-video playback and mobile gaming.

In a steady-state short-video playback scenario, we compare the average latency of two scheduler-critical operations, context switching and pick-next-task selection, before and after enabling MUSCHED. As shown in Table 4, the average context-switch latency remains unchanged at 5 µs, while the

Table 4: Runtime overhead and energy impact of MUSCHED under video and gaming scenarios.

Scenario	Metric	Baseline	Ours
Video	Avg. context-switch latency (µs)	5	5
	Avg. pick next task latency (µs)	2	3
Gaming	Avg. FPS	119.76	119.69
	Worst dropped frames	4	3
	Normalized current (mA)	726.62	718.16

Table 5: End-to-end QoE evaluation in large-scale production deployment. Values report anomaly frequencies per thousand hours of user activity before and after deploying MUSCHED.

Scenario	Anomaly Metrics	# of Anomalies		Improvement
		Baseline	Ours	
Animation	drop $>$ 50ms	27.2	20.4	25.0%
Swipe	drop $>$ 50ms	10.5	6.8	35.7%
Startup	startup $>$ 2s	94.5	65.5	30.7%

average pick next task latency increases only slightly from 2 µs to 3 µs. These results indicate that the IPC and lock tracking hooks and the dependency-propagation logic add little overhead to the scheduler fast path in a representative interactive media workload. We also evaluate a mainstream mobile game at 120 FPS with default graphics settings to assess runtime cost under sustained high-refresh load. As shown in Table 4, the average frame rate is nearly unchanged, the worst dropped-frame count is slightly improved, and the normalized current is comparable. Together, these results suggest that MUSCHED does not introduce non-negligible overhead in practice.

7 Experiences and Lessons Learned

In this section, we introduce the experiences and lessons learned during our development and deployment process of MUSCHED. We also present big data results on large-scale commercial smartphones, covering application cold-start performance, frame drop performance in UI animation, and user swipe scenarios.

7.1 Production Deployment of MUSCHED

Since its deployment in January 2024, the scheduling algorithm designed using MUSCHED has been deployed on over *20 million* mobile devices from Honor. The deployment spans multiple product tiers, including flagship and mid-range device series, and covers both MediaTek and Qualcomm platforms. Targeted scenarios include application cold-start, web browsing, and touch-sliding interactions.

To illustrate how MUSCHED performs in real-world scenarios, we validate it using large-scale user data. We measure anomaly frequency across various real-world scenarios, including application cold start, animation, and swipe, that significantly impact users' QoE. Specifically, for application cold-start, we define a cold-start time that exceeds 2 seconds as an anomaly; for animation and swipe scenarios, we define continuous frame drops lasting more than 50ms as an anomaly. We then measure the anomaly frequency in each scenario before and after deploying our approach. The anomaly counts reported here indicate how many times users encounter such failures per thousand hours of usage. These statistics are first collected on-device and then aggregated across the deployed user population for large-scale analysis.

The results in Table 5 demonstrate that our approach consistently delivers stable and significant performance gains in the real world. As shown in Table 5, we observe clear reductions in anomaly rates across animation, swipe, and startup scenarios. Specifically, animation anomalies decrease by 25.0%, swipe anomalies by 35.7%, and startup anomalies by 30.7%.

These findings indicate that across diverse workloads and complex user-interaction scenarios, our scheduling strategy can effectively identify critical tasks, prevent stalls along the critical path, and maintain smooth interactions even amid heavy system-resource contention. It validates the generalization ability and robustness of our approach in large-scale real-world deployments, enabling continuous improvements to overall user QoE.

7.2 Lessons Learned

Over the course of developing MUSCHED, from initial research prototyping in 2021 to production deployment in January 2024, we have learned several lessons about the practical realities of user space scheduling on commercial mobile devices.

Lessons Learned ①: Practical gains in development efficiency. Adopting a user-space scheduling framework based on Linux sched_ext delivers impactful practical benefits, addressing key challenges in deploying scheduling innovations on commercial mobile devices.

First, user-space design enables dynamic policy iteration without kernel recompilation: scheduling logic can be hot-updated, reducing iteration time from hours to minutes for scenario-specific policies (e.g., gaming, application cold-start). Combined with sched_ext's modularity, this decouples policy logic from core kernel components, mitigating stability risks by confining failures to user space.

Second, user-space scheduling facilitates fine-grained scenario adaptation and controlled experimentation. Dynamic conditional triggering aligns policies with context-aware demands—aggressive CPU provisioning for games, power-efficient scheduling for low-interaction phases, and lifecycle-aware core scaling for application foreground transitions. Its isolated execution also enables safe A/B testing on production

devices, supporting "small-batch" validation without potential global disruption.

Third, the approach lowers developer barriers via high-level abstractions (e.g., BCC, BPFtrace), enabling policy development in C or scripting languages without deep kernel expertise. It integrates seamlessly with Android's system services (e.g., Activity Manager) and leverages widespread kernel support for user-space scheduling frameworks.

***Lessons Learned ②: Future potentials of* MUSCHED.** First, on-demand activation via scenario recognition is non-negotiable for commercial viability. A single user-space scheduling framework cannot optimize all mobile workloads (e.g., interactive apps, background services, optimized games). We should integrate it with a scenario recognizer to dynamically trigger policies in critical contexts (e.g., application launches, UI animations) and to auto-exit during idle/background phases, thereby avoiding unnecessary overhead and ensuring compatibility across diverse use cases.

Second, tight integration with SoC-specific hardware characteristics is essential. Modern mobile SoCs feature heterogeneous cores, dynamic idle states, and CPU affinity constraints. While kernel mechanisms partially address these, user-space schedulers leverage fine-grained hardware telemetry (e.g., per-core utilization, thermal headroom) to make informed core-selection decisions, ensuring consistency across chipset models and unlocking full performance potential.

Third, co-optimization of core selection and frequency scaling is imperative. These two mechanisms are inherently coupled: a high-performance core assignment fails if frequency is throttled, and aggressive scaling wastes power without intelligent core selection. Future user-space frameworks should unify these controls to balance latency and energy efficiency, e.g., triggering frequency boosts only for critical tasks on performance cores. These lessons highlight that the commercial success of user-space scheduling depends on adapting to real-world hardware diversity, workload variability, and intertwined resource management challenges.

Lessons Learned ③: No silver bullets for workloads. Our previous results show that the MUSCHED can significantly improve some interactive Android workloads. However, its performance benefits are limited for highly optimized games with simple, well-understood critical paths.

There are two main reasons. First, mobile games are flagship workloads that showcase the overall device capability. Major SoC and OEM vendors, therefore, invest heavily in this domain. For mainstream devices, the frame rate of popular titles already meets the target cap under most conditions. Second, compared with the heavily interactive workloads discussed earlier, the scheduling patterns of mobile games are relatively simple. The critical threads (e.g., UnityMain, UnityGfxDeviceW) exhibit stable and highly regular switching behavior. Consequently, the primary bottlenecks shift to power and thermal constraints, while SoC/OEM mechanisms such as CPU load estimation, power governors, and power-aware schedulers (e.g., PELT/WALT) dominate performance.

We evaluate a large MOBA game and find that with and without the MUSCHED, the average FPS remains stable, and frame-time variability shows no statistically significant difference. Even after applying optimizations such as core selection and enqueue policies, we observe only a minor decrease in CPU utilization, while the electric current and device shell temperature become slightly worse.

Lessons Learned ④: No free escape with eBPF. In the in-kernel setting, eBPF is deliberately restricted and, in practice, not Turing-complete. For kernel safety, the verifier of eBPF enforces strict constraints: no direct modification of kernel data structures, no unbounded loops, a 512-byte stack limit, and no dynamic memory allocation. The verifier itself does not scale well to complex control-flow paths. When the code contains layered if-else conditions, the program can easily be rejected during verification.

In practice, we agree with these safety constraints, but fast prototyping and iteration often require us to escape the pure eBPF model. Our main strategy is to treat eBPF as a thin control plane and leave complexity back into the kernel (modules) via the kfunc mechanism. With simple annotations, selected kernel functions can be exposed as callable kfuncs to eBPF programs. To support updates to kernel data structures, we implement a memcpy-like BPF kfunc that performs the actual writes in kernel space. For existing complex scheduling algorithms, we encapsulate them as kfuncs rather than reimplementing them. To cope with stack and allocation limits, we store state and large data structures in global variables and BPF maps. To keep verifier complexity manageable, we avoid deep branching and, when necessary, split decision logic into multiple shallow control flows.

8 Related Work

MUSCHED builds on several pioneering works, including application-specific OS scheduling and eBPF-based user-space OS management. In this section, we introduce the works related to MUSCHED.

8.1 OS Scheduling Advancements

OS scheduling has been widely studied in the last few decades. Academia and industry introduce a spectrum of general-purpose policies as well as many domain-specialized schedulers that target latency, throughput, or energy trade-offs. Widely adopted policies such as FIFO and the Linux Completely Fair Scheduler (CFS) provide simple yet effective performance across a wide range of workloads.

Beyond general-purpose schedulers, recent years have seen a proliferation of scheduling work tuned to specific workloads or application patterns. For example, for remote procedure calls (RPCs) in data centers, existing works such as Shinjuku [13], Shenango [20], and eRPC [15] push the envelope on

tail latency and per-core efficiency. Work in this area often involves co-designing CPU scheduling with network transport, as RPCs typically encompass both networking and computing components. Another example is in the serverless computing domain, SFS [8] and Alps [9] focus on the peculiarities of short-lived, latency-sensitive function executions across highly bursty request arrivals. These designs explore techniques including lightweight isolation, fast placement, and resource reclamation to reduce cold-start and queuing delays and to improve utilization in multi-tenant environments.

Mobile systems introduce a different set of constraints where user experience and energy consumption are primary concerns. For example, SmartOS [10] leverages reinforcement learning to obtain the importance of each task on mobile devices and allocates resources according to user preference. Orthrus [22] proposes a framework that co-optimizes the process scheduling and CPU frequency governing on mobile OS for both energy efficiency and quality of service. Targeting the slow UI responsiveness problem, TIHMM [25] reconstructs the process state model on Android with new "hogging" states to reduce the suboptimal resource allocation of the original Android process management.

Compared with the above works, MUSCHED not only focuses on specific scheduling problems during cold start or UI animation but also acts as a fundamental framework to solve the trinity problem of mobile scheduling.

8.2 User-space OS Management with eBPF

Extended Berkeley Packet Filter (eBPF) is a lightweight in-kernel virtual machine that runs sandboxed programs at well-defined OS hook points. Programs interact with the kernel and user space via `maps`, enabling low-overhead telemetry, state sharing, and control. Because eBPF enforces safety and verification, it lets developers iterate on policy logic in user space while applying enforcement close to the kernel. This property makes eBPF a natural substrate for user-space OS management: user processes can define, tune, and deploy policies quickly, and eBPF provides fast, safe kernel mediation and observability. Prior work has explored both eBPF-based management and broader user-space delegation to make OS policies more reconfigurable.

A representative work is ghOst [12], which delegates CPU scheduling decisions to user-space agents. Its key idea is to keep the kernel responsible for safe enforcement and low-level task state management, while allowing developers to implement scheduling policies in user space and update them without redesigning the whole kernel scheduler. This design supports a variety of scheduling models, including centralized, sharded, and preemptive scheduling, and has demonstrated that user-space schedulers can provide both flexibility and low latency for datacenter workloads. Building on this direction, Syrup [14] extends user-space scheduling beyond CPU cores to jointly manage the networking stack and NIC, showing that policy delegation can coordinate multiple OS subsystems.

Beyond CPU scheduling, eBPF-based management has also been explored in the memory-management domain. CacheBPF [26] facilitates the modification of page-eviction policies without changing core kernel code. FetchBPF [6] uses eBPF to implement pluggable memory prefetch strategies. It highlights how introspective kernel probes enable workload-aware prefetching. Results show substantial throughput gains for memory-bound applications.

These prior systems either move selected OS functionality into user space or use eBPF to customize a single subsystem. In contrast, MUSCHED focuses on interaction-aware CPU scheduling for commercial mobile devices. It couples user-space policy adaptation with low-overhead kernel enforcement to support scenario-aware prioritization and cross-process dependency handling without invasive and frequent kernel changes, which is impractical on COTS devices.

9 Conclusion

This paper presents MUSCHED, a semantic-aware scheduling framework for modern mobile SoCs. MUSCHED addresses the fundamental semantic gap between low-level kernel scheduling and high-level user interaction, which becomes especially problematic in the face of the mobile scheduling trilemma: scarce high-performance cores, complex cross-process dependencies, and unforgiving latency deadlines. MUSCHED introduces a new VIP scheduling class between RT and CFS, identifies interaction-critical threads through scenario-aware annotation, propagates priority across Binder and lock dependencies, and supports user-space policy updates without kernel recompilation.

In laboratory experiments, MUSCHED reduces average application cold-start time by 14.8%. In large-scale production deployment on over 20 million commercial devices since 2024, MUSCHED reduces cold-start anomalies by 30.7%, animation anomalies by 24.9%, and swipe anomalies by 35.7%, while preserving system stability. These results demonstrate that semantic-aware, deployable user-space scheduling is a practical way to improve mobile QoE on commercial off-the-shelf devices.

Acknowledgement

We thank the anonymous reviewers for their constructive feedback and valuable suggestions. This work was supported in part by the National Natural Science Foundation of China under Grants 62302096, 62432004, 62302207, and U24B20152; the Natural Science Foundation of Jiangsu Province under Grant BK20230813; the Fundamental Research Funds for the Central Universities under Grants 2026300278 and ZZKT2026A31; and the "111 Center" under Grant No. B26023.

References

[1] App startup time | App quality | Android Developers — developer.android.com. https://developer.android.com/topic/performance/vitals/launch-time. [Accessed 11-12-2025].

[2] Extend the kernel with eBPF | Android Open Source Project — source.android.com. https://source.android.com/docs/core/architecture/kernel/bpf. [Accessed 08-12-2025].

[3] Number of mobile devices worldwide 2020-2025. https://www.statista.com/statistics/245501/multiple-mobile-device-ownership-worldwide. [Accessed 08-12-2025].

[4] Program Type 'BPF_PROG_TYPE_STRUCT_OPS' - eBPF Docs — docs.ebpf.io. https://docs.ebpf.io/linux/program-type/BPF_PROG_TYPE_STRUCT_OPS/. [Accessed 08-12-2025].

[5] Justinien Bouron, Sebastien Chevalley, Baptiste Lepers, Willy Zwaenepoel, Redha Gouicem, Julia Lawall, Gilles Muller, and Julien Sopena. The Battle of the Schedulers: {FreeBSD} {ULE} vs. Linux {CFS}. In *Proc. of USENIX ATC*, pages 85–96, 2018.

[6] Xuechun Cao, Shaurya Patel, Soo Yee Lim, Xueyuan Han, and Thomas Pasquier. {FetchBPF}: Customizable prefetching policies in linux with {eBPF}. In *Proc. of USENIX ATC*, pages 369–378, 2024.

[7] Inho Cho, Ahmed Saeed, Joshua Fried, Seo Jin Park, Mohammad Alizadeh, and Adam Belay. Overload Control for {μs-scale}{RPCs} with Breakwater. In *Proc. of USENIX OSDI*, pages 299–314, 2020.

[8] Yuqi Fu, Li Liu, Haoliang Wang, Yue Cheng, and Songqing Chen. SFS: Smart OS Scheduling for Serverless Functions. In *SC22: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–16, November 2022.

[9] Yuqi Fu, Ruizhe Shi, Haoliang Wang, Songqing Chen, and Yue Cheng. {ALPS}: An Adaptive Learning, Priority {OS} Scheduler for Serverless Functions. In *Proc. of USENIX ATC*, pages 19–36, 2024.

[10] Sepideh Goodarzy, Maziyar Nazari, Richard Han, Eric Keller, and Eric Rozner. Smartos: towards automated learning and user-adaptive resource allocation in operating systems. In *Proc. of ACM APSys*, pages 48–55, 2021.

[11] Mingcong Han, Hanze Zhang, Rong Chen, and Haibo Chen. Microsecond-scale Preemption for Concurrent {GPU-accelerated}{DNN} Inferences. In *Proc. of USENIX OSDI*, pages 539–558, 2022.

[12] Jack Tigar Humphries, Neel Natu, Ashwin Chaugule, Ofir Weisse, Barret Rhoden, Josh Don, Luigi Rizzo, Oleg Rombakh, Paul Turner, and Christos Kozyrakis. ghOSt: Fast & Flexible User-Space Delegation of Linux Scheduling. In *Proc. of ACM SOSP*, pages 588–604, 2021.

[13] Kostis Kaffes, Timothy Chong, Jack Tigar Humphries, Adam Belay, David Mazières, and Christos Kozyrakis. Shinjuku: Preemptive Scheduling for {μsecond-Scale} Tail Latency. In *Proc. of USENIX NSDI*, pages 345–360, 2019.

[14] Kostis Kaffes, Jack Tigar Humphries, David Mazières, and Christos Kozyrakis. Syrup: User-Defined Scheduling Across the Stack. In *Proc. of ACM SOSP*, pages 605–620, 2021.

[15] Anuj Kalia, Michael Kaminsky, and David Andersen. Datacenter {RPCs} can be General and Fast. In *Proc. of USENIX NSDI*, pages 1–16, 2019.

[16] Zhihui Ke, Xiaobo Zhou, Dadong Jiang, Hao Yan, and Tie Qiu. CollabVr: Reprojection-Based Edge-Client Collaborative Rendering for Real-Time High-Quality Mobile Virtual Reality. In *Proc. of IEEE RTSS*, pages 304–316, December 2023.

[17] James E. Kelley Jr and Morgan R. Walker. Critical-path planning and scheduling. In *Papers Presented at the December 1–3, 1959, Eastern Joint IRE–AIEE–ACM Computer Conference*, pages 160–173, 1959.

[18] Devi Klein, Josef Spjut, Ben Boudaoud, and Joohwan Kim. Variable frame timing affects perception of smoothness in first-person gaming. In *2024 IEEE Conference on Games (CoG)*, pages 1–8, 2024.

[19] Yu Liu, Puqi Zhou, Zejun Zhang, Anlan Zhang, Bo Han, Zhenhua Li, and Feng Qian. Muv2: scaling up multi-user mobile volumetric video streaming via content hybridization and sharing. In *Proc. of ACM MobiCom*, pages 327–341, 2024.

[20] Amy Ousterhout, Joshua Fried, Jonathan Behrens, Adam Belay, and Hari Balakrishnan. Shenango: Achieving high {CPU} efficiency for latency-sensitive datacenter workloads. In *Proc. of USENIX NSDI*, pages 361–378, 2019.

[21] Lenin Ravindranath, Jitendra Padhye, Sharad Agarwal, Ratul Mahajan, Ian Obermiller, and Shahin Shayandeh. {AppInsight}: Mobile App Performance Monitoring in

the Wild. In *Proc. of USENIX OSDI*, pages 107–120, 2012.

[22] Qianlong Sang, Jinqi Yan, Rui Xie, Chuang Hu, Kun Suo, and Dazhao Cheng. Qos-aware power management via scheduling and governing co-optimization on mobile devices. *IEEE Transactions on Mobile Computing*, 2024.

[23] The Linux Kernel documentation contributors. Energy aware scheduling. https://www.kernel.org/doc/html/latest/scheduler/sched-energy.html, 2025. Accessed: 2026-06-03.

[24] Yizong Wang, Dong Zhao, Huanhuan Zhang, Teng Gao, Zixuan Guo, Chenghao Huang, and Huadong Ma. Bandwidth-efficient mobile volumetric video streaming by exploiting inter-frame correlation. *IEEE Transactions on Mobile Computing*, 23(10):9410–9423, 2024.

[25] Jianwei Zheng, Zhenhua Li, Feng Qian, Wei Liu, Hao Lin, Yunhao Liu, Tianyin Xu, Nan Zhang, Ju Wang, and Cang Zhang. Rethinking Process Management for Interactive Mobile Systems. In *Proc. of ACM MobiCom*, pages 215–229, New York, NY, USA, May 2024.

[26] Tal Zussman, Ioannis Zarkadas, Jeremy Carin, Andrew Cheng, Hubertus Franke, Jonas Pfefferle, and Asaf Cidon. Cache is king: Smart page eviction with ebpf. *arXiv e-prints*, pages arXiv–2502, 2025.

ARTIFACT EVALUATED
USENIX
AVAILABLE

qTPU: Hybrid Tensor Networks for Quantum-Classical Acceleration

Nathaniel Tornow Emmanouil Giortamis Dennis Sprokholt
Christian B. Mendl Pramod Bhatotia
Technical University of Munich

Abstract

Classical accelerators (GPUs, TPUs) excel at high-performance computing but cannot efficiently represent problems that scale exponentially, such as highly entangled quantum states. Quantum processors (QPUs) address these intractable problems but are constrained by noise, high error rates, and low throughput. This necessitates hybrid quantum-classical execution: computationally complex kernels are offloaded to QPUs, while scalable computation runs on classical accelerators.

Despite this clear need for hybrid systems, we currently lack a concise programming paradigm for expressing and optimizing them. Developers must manually partition and orchestrate applications across quantum and classical hardware through *ad hoc* approaches that produce fragmented, inflexible workflows. This prevents holistic optimization across the quantum-classical interface and limits scalability.

To this end, we introduce the *hybrid tensor network (hTN)* abstraction—a unified representation capturing quantum-classical computation—realized in qTPU, an end-to-end system comprising: the *qTPU programming model* for declarative hybrid computation specification; the *qTPU compiler* for holistic hTN optimization balancing classical cost against quantum error; and the *qTPU runtime* for scalable execution across heterogeneous QPUs and classical accelerators.

Our evaluation demonstrates that qTPU achieves 3–4 orders of magnitude lower classical overhead, up to 7.2× lower quantum error rates, up to 53× faster compilation, and over 20× end-to-end speedups across hybrid ML, circuit knitting, and quantum error mitigation.

1 Introduction

Data centers are increasingly relying on specialized accelerators, such as GPUs, TPUs, and FPGAs, to meet the demands of growing domain-specific compute workloads, *e.g.,* in machine learning and scientific computing [3, 44, 45, 68]. Through recent remarkable engineering efforts, quantum processors (QPUs) are becoming available as new domain-specific accelerators [2, 5, 30, 38]. When combined with classical computing, QPUs could help us address problems in materials science and chemistry that remain intractable on classical computers [40, 53, 66, 72, 78].

QPUs leverage quantum-mechanical properties such as superposition and entanglement to efficiently represent and manipulate exponentially large state spaces that cannot be encoded using classical memory [57]. This capability enables, *e.g.,* the simulation of strongly correlated quantum systems and the modeling of highly entangled or high-dimensional distributions in machine learning, and offers the potential for exponential speedups for certain problems [36, 75, 81].

Although QPUs offer unique advantages for specific classes of problems, they are subject to severe limitations: they are noisy, provide only limited computational resources in memory and availability of devices, and have throughput orders of magnitude lower than classical accelerators [41, 67].

As a result, quantum advantages – where they exist – typically apply only to the complex core of a computational problem, where classical devices fail due to the exponential complexity of representing quantum states [12, 41]. Many important workloads, such as complex machine-learning models or scientific simulations for material sciences, fall into a regime where they are both *(i)* too inefficient or outright infeasible to execute solely on classical accelerators, and *(ii)* too complex and large-scale to run entirely on today's QPU hardware [54, 66].

This motivates a *hybrid* quantum-classical execution paradigm. Instead of running a program entirely on classical hardware or entirely on a QPU, realistic applications distribute work across both: the quantum-inclined components are offloaded to QPUs, while the remaining computation runs on better-suited classical accelerators [12, 34].

To program such hybrid applications, the current approach relies on a rigid host–kernel programming model. Programmers must manually isolate portions of the algorithm that could benefit from quantum or classical acceleration and implement them as separate kernels [8, 59].

Unfortunately, this existing form of quantum–classical hybrid computing is tedious, inflexible, and inefficient to use in cloud environments, suffering from three core limitations:

(1) **Fragmented programming models:** Developing hybrid applications requires reasoning across two distinct programming paradigms and manually implementing separate routines for classical devices and QPUs [19]. This fragmented model forces developers to explicitly manage low-level data movement, control flow, and up-front partitioning decisions, making the development of hybrid applications tedious and prone to inefficiencies.

(2) **Lack of holistic compiler optimizations:** Current programming models require developers to define a fixed

boundary between quantum and classical kernels *a priori* [8]. This rigid separation prevents compilers from reasoning across the quantum–classical interface, hindering holistic, end-to-end optimization. As a result, no dynamic execution decisions are possible; *e.g.,* even if parts of a quantum kernel could be executed more efficiently on a classical accelerator, they cannot be automatically offloaded, locking the application into a potentially suboptimal execution path [88].

(3) **Poor scalability and rigid runtime:** Current hybrid workflows are inefficient and struggle to scale because they are manually orchestrated and executed as separate, sequential steps, lacking a unified program view [33]. This prevents dynamic execution decisions, *e.g.,* even if part of a quantum kernel would run more efficiently on a classical accelerator, the compiler cannot offload it automatically.

We argue that these limitations fundamentally arise from the lack of a concise representation that seamlessly captures both quantum and classical computation, leading to our main research question:

Research question

How can we design a concise and expressive hybrid computing paradigm that unifies quantum and classical computation while enabling efficient execution across heterogeneous accelerators?

Our key idea for addressing this challenge is the **hybrid Tensor Network (hTN)** abstraction, a necessary theoretical foundation to address the fragmentation in hybrid computing. In particular, we establish Tensor Networks (TNs) [61]—a core algebraic formalism for both quantum circuit simulation and accelerated classical computation—as a unifying computational substrate [52, 92]. The hTN extends this formalism by representing the complete hybrid workflow, including quantum components (quantum tensors) and classical components (classical tensors), as a single, connected graph. This unified expression bridges the classical-quantum gap, enabling holistic optimization and treatment of complex algorithms, such as hybrid machine learning models, as a single, coherent computational entity.

To realize hTNs in practice and to answer our posed research question, we present QTPU, an end-to-end systems approach for unified, flexible and scalable hybrid computing. *(1)* Our *QTPU Programming Model* formally defines the primitives built on hTNs, allowing programmers to specify only *what* to compute, *not how*; our *(2) QTPU Compiler* converts hTNs into executable hybrid code, which we holistically optimize for efficient computation across a set of QPUs and classical accelerators; and finally, *(3)* our *QTPU Runtime* dynamically adapts the hybrid computation to scalably execute across the available heterogeneous collection of classical and quantum accelerators.

Our evaluation shows that QTPU achieves substantial improvements across hybrid workloads: up to 53× faster compilation, 7.2× lower quantum error rate, 3,700× smaller generated code, and 10,000× lower classical overhead than state-of-the-art baselines. QTPU further scales to problem sizes that existing hybrid systems cannot handle, while enabling efficient utilization of heterogeneous QPU/GPU resources.

Contributions. We make the following contributions.

- We introduce the hybrid Tensor Network (hTN) abstraction, a unified representation that captures hybrid quantum-classical workflows as a single connected tensor graph, moving beyond existing host-kernel models (§ 3.2).
- We formally introduce the QTPU Programming Model, which realizes our hTN abstraction. We define its core primitives—including the iswitch (for specifying quantum tensors) and hEinsum (for hybrid contraction)—which enable the simple and declarative description of entire hybrid quantum-classical programs (§ 5).
- We introduce compiler transformations that jointly optimize across both modalities by balancing the classical cost of tensor contractions with the quantum execution error, ensuring optimal utilization of available quantum and classical compute paradigms. (§ 6).
- We design a scalable execution approach that distributes and dynamically adapts hybrid programs across a heterogeneous set of QPUs and classical accelerators. The QTPU runtime facilitates this through runtime code generation, tailoring device-specific kernels on demand to maximize parallel execution and efficient resource utilization based on available hardware. (§ 7).

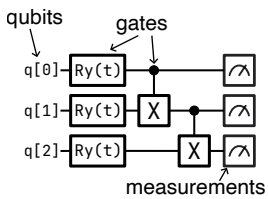
2 Background and Motivation

2.1 Foundations of Quantum Computing

Quantum computing could accelerate tasks in many areas, including quantum physics, materials science, and machine learning [36, 40, 72, 81]. Although current QPUs remain in their early stages, recent experiments have demonstrated behavior that is increasingly difficult to reproduce classically, suggesting potential advantages for specific problem classes [4, 16, 17, 37, 83].

QPUs are typically programmed under the *quantum kernel model*, illustrated in Fig. 1. A quantum kernel is a classical function that produces a *quantum circuit*, which is then compiled and executed on a QPU [59].

The basic computational unit of a quantum circuit is the *qubit*, a two-level quantum system. An n-qubit register corresponds to a complex-valued vector in a 2^n-dimensional Hilbert space, enabling it to represent vectors that would require exponentially many parameters to store classically [57]. A quantum circuit generally *(1)* initializes qubits, *(2)* applies single- and multi-qubit quantum gates, and *(3)* measures a




```
@qkernel
def simple_kernel(t)
    q = alloc_qubits(3)
    for i in range(3):
        ry(q[0], t)
    cx(q[0], q[1])
    cx(q[1], q[2])
    measure(q)

circuit = simple_kernel(0.3)
res = qpu.run(circ, shots=1000)
```

Figure 1: Example of the quantum-kernel model.

subset of the qubits in the computational basis. Measurement collapses each qubit to either 0 or 1 according to a distribution determined by the circuit; repeating the circuit for s *shots* yields bitstrings $b^{(\ell)} \in \{0,1\}^m, \ell = 1,\ldots,s$, over the m measured qubits.

2.2 Quantum-Classical Computing

Both quantum and classical accelerators can efficiently solve domain-specific problems; however, neither provides a one-size-fits-all solution.

On the one hand, modern *classical accelerators* (such as GPUs, TPUs) provide massive data and operation parallelism, large on-device memory, and extremely high throughput and low latency for, *e.g.*, dense numerical workloads. They are reliable, abundantly deployed, and scale to thousands of devices in distributed cloud environments. However, classical accelerators operate on data structures that cannot efficiently represent strongly correlated and entangled quantum systems, whose classical representations scale exponentially compared to their quantum representations [28, 31, 99].

On the other hand, *quantum accelerators* (QPUs) leverage unique quantum-mechanical effects to efficiently represent and manipulate highly correlated quantum states that classical systems cannot handle [41, 57]. This makes them attractive for tasks such as simulating highly correlated systems for material science or quantum chemistry. However, QPUs also have inherent limitations, since qubits are inherently noisy, and quantum operations are prone to errors, resulting in unreliable results at large circuit depths [64, 74]. Achieving fault tolerance requires substantial overhead in both qubit resources and execution time, making logical qubits a scarce resource and significantly limiting QPU throughput and latency.

The need for hybrid acceleration. Classical and quantum accelerators have complementary strengths and limitations, making neither suitable for all tasks. Realistic workloads therefore require a tightly integrated hybrid approach: QPUs are used selectively where quantum capabilities offer an advantage, while all other computation is handled by classical accelerators.

2.3 Blueprint of Hybrid Applications

Hybrid quantum-classical computing leverages the complementary strengths of classical accelerators and QPUs to solve

```
@qkernel # QNN kernel (input x, weights W, output o)
def quantum_layer(x, W, o):
    q = alloc_qubits(N)
    for i in range(N):
        ry(q[i], x[i]) # input encoding
    model(q, W) # apply some QNN-layer
    apply_obs(q, o) # measure specific observable
    measure(q)

@ckernel # a classical GPU/TPU kernel
def classical_layer(Z, V): # linear layer
    return matmul(V, Z) # matrix-matrix multiplication

def forward(X, W, V, O): # hybrid model evaluation
    # Stage 1: construct and run a family of circuits
    circuits = [quantum_layer(x,W,o) for x in X for o in O]
    Z = qpu.run(circuits, shots=1000)
    # Stage 2: perform classical processing
    Y_pred = classical_layer(Z, V)
    return Y_pred

def train(): # model training loop
    X,Y = dataset(); W,V = init_weights(); O = observables()
    for epoch in range(E):
        Y_pred = forward(X, W, V, O)
        update([W, V], loss(Y, Y_pred))
```

Listing 1: Fragmented quantum-classical ML example.

problems that neither can efficiently address on its own.

The *core* of virtually all hybrid quantum-classical algorithms and methods—including quantum simulation [56, 62, 97], circuit knitting [84, 90], error mitigation [50, 89], and quantum-enhanced machine learning (ML) [10, 14, 51]—shares a common computational blueprint, consisting of two tightly coupled stages. For example, Lst. 1 shows a quantum-classical approach to train a machine-learning kernel, implemented via the kernel's `forward`-pass.

Stage 1: Evaluate families of quantum circuits. The workflow begins by executing a *family* of closely related quantum circuits derived from classical inputs or intermediate computation. These circuits typically share a structural backbone but differ in controlled ways, together exposing the necessary information required for the hybrid computation.

In our example (Lst. 1, Line 16-17), we produce and run a family of quantum circuits that encode the individual inputs of our training dataset and measure observables, whose results are later fed into the next layer. This family needs to be explicitly generated and run on a specific QPU to produce all the results required for the next stage.

Stage 2: Perform classical processing. The measurement outcomes from the circuit family are then combined with additional classical inputs and processed using classical computation. Across algorithmic classes, this postprocessing is governed almost entirely by *linear-algebraic* operations, which transform the quantum-generated data into the quantities of interest and feed back into the host-driven control loop that produces the next family of circuit evaluations.

In our example (Lst. 1, Line 19), we apply a simple linear layer—*e.g.*, on a GPU—to the quantum outputs and classical

weights, producing the predicted results for the dataset.

Examples of hybrid applications. Beyond our example of hybrid machine learning, various other applications in quantum computing follow the same structure. In this work, we focus on three representative hybrid applications:

(i) **Hybrid Machine Learning** combines variational quantum circuits with classical layers to model functions or data distributions that cannot be efficiently captured by purely classical or purely quantum models [10], similar to our toy-example of Lst. 1;

(ii) **Scalable hybrid computing** uses circuit knitting techniques [84, 90] to execute circuits that exceed the qubit capacity of QPUs by decomposing a large circuit into families of smaller subcircuits, running them on small QPUs, and classically reconstructing the result; and

(iii) **Quantum Error Mitigation (QEM)** generates and executes families of circuit variations to estimate and suppress noise effects, combining the resulting measurements through classical postprocessing to approximate the noiseless output of the original circuit [26, 49, 89].

We evaluate our approach on these applications in § 8.

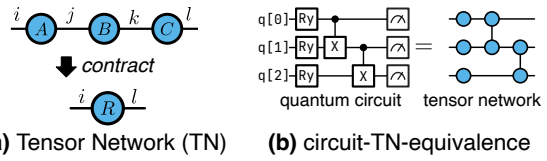
2.4 Limitations of the Current Practices

Using both quantum and classical resources leverages the full benefits of both worlds. Unfortunately, the current approach to writing quantum-classical programs follows a manual, ad-hoc workflow that is error-prone and complex due to several fundamental limitations, which we explain now.

Limitation #1: Fragmented programming model. In current hybrid workflows [8, 59], the quantum and classical parts of a program must be written in separate paradigms, as shown in Lst. 1. The programmer implements one kernel for the QPU and another for the GPU/TPU, and must manually coordinate them in the host code. In the example, the `forward` function builds a family of circuits, sends them to the QPU, waits for results, and then invokes a classical kernel. Thus, even though the programmer specifies *what* the computation should do, they must also express *how* the quantum and classical kernels interact, making hybrid programs unnecessarily complex.

Limitation #2: Inflexible compilation. The programmer must fix the quantum–classical boundary a priori, preventing holistic cross-paradigm optimization because the hybrid computation is not represented as a single unified object. In Lst. 1, the quantum and classical kernels are statically tied to their modalities—QPUs for `quantum_layer` and GPUs/TPUs for `classical_layer`. This rigidity blocks optimization of `forward` as a whole, since it consists of two separately compiled black-box kernels. As a result, tasks cannot be reassigned across paradigms, even when parts of `quantum_layer` could run more efficiently on a classical accelerator.

Limitation #3: Rigid runtime. Current hybrid workflows are manually orchestrated and executed as separate, sequential steps, giving the runtime no unified view of the computation. In Lst. 1, the manual orchestration inside `forward` fixes the quantum and classical stages in order and binds them to their respective devices, preventing dynamic parallelism or resource-aware scheduling. Because the assignment of work is static, the runtime cannot adapt to the availability or load of heterogeneous QPUs and classical accelerators, which severely limits scalability for larger problems.



(a) Tensor Network (TN) **(b)** circuit-TN-equivalence

Figure 2: Tensor Network background (§ 3.1).

Problem statement

We require a solution that unifies quantum-classical applications in a common foundation, thus allowing holistic optimization across both fragments, while adapting its execution to available quantum-classical devices.

3 Hybrid Tensor Networks (hTNs)

We propose *hybrid tensor networks* (hTN), which address the above limitations by unifying quantum and classical computing in a common framework.

3.1 Tensor Networks (TNs)

Tensor networks compactly describe an output tensor as a function of input tensors. TNs emerged in condensed-matter physics as a compact representation of many-body quantum states that would otherwise require exponential resources to describe [60, 91], and have since become the backbone of state-of-the-art classical simulators for quantum circuits [29, 58]. In parallel, the same einsum-based formalism is the standard language for specifying linear-algebraic computations in, *e.g.*, deep-learning frameworks [39, 63].

Intuitively, a tensor generalizes a vector or matrix to an array of arbitrary rank. A tensor network describes how such tensors are combined through *contractions*—pairwise summations over shared indices, of which matrix multiplication is the rank-two special case. Contracting a network composes its input tensors into a single output tensor by summing over all shared indices simultaneously.

Einsum expression. TNs specify pure linear-algebra expressions via the *Einsum* language. Formally, $T_O = \mathrm{einsum}(I_1, \ldots, I_n \to O;\ T_1, \ldots, T_n)$ defines the output tensor T_O entry-wise as

$$T_O[\mathbf{o}] \triangleq \sum_{\mathbf{i} \in (\mathcal{I} \setminus O)} \prod_{k=1}^{n} T_k[\mathbf{i}_{I_k}], \tag{1}$$

where T_i are tensors with index set I_i; O is the output index set, of which $\mathbf{o}$ denotes an assignment; $\mathcal{I} = \bigcup_{k=1}^{n} I_k$ is the

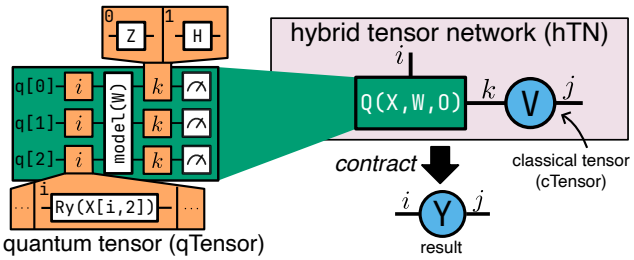


Figure 3: Our hybrid tensor network (hTN) abstraction (§ 3.2).

union of all index sets; and $\mathbf{i}_{I_k}$ is the restriction of the global index assignment $\mathbf{i}$ to the indices of tensor T_k. For example,

$$R = \text{einsum}(ij, jk, kl \to il;\ A, B, C) \qquad (2)$$

specifies the triple matrix product $R = A \cdot B \cdot C$, *i.e.,* $R_{il} = \sum_{j,k} A_{ij} B_{jk} C_{kl}$.

Tensor networks as graphs. An equivalent way to specify a TN is as an undirected graph, whose vertices represent tensors and the edges correspond to shared indices (*i.e.,* contractions). When two tensors are connected by an edge, the corresponding index is contracted (summed over), and dangling edges connected to only a single vertex specify the output indices. Fig. 2 (a) shows an example of such a TN-graph, equivalent to the einsum expression of Eq. 2.

Quantum computations as TNs. The connection between quantum and classical computation becomes explicit when we observe that any quantum circuit can be represented as a TN, as we show by example of Fig. 2 (b). Each k-qubit gate is a unitary linear map, and thus a rank-$2k$ tensor whose indices correspond to its k input and k output wires [52,60]. A quantum circuit then becomes a TN in which adjacent gates share the wires they act on as contracted indices, so that executing the circuit (*i.e.,* applying gates in sequence) is the same operation as contracting the network. Thus, quantum circuits can be expressed using the einsum-based formalism introduced above, enabling a unified representation of both classical and quantum computations.

3.2 Key Idea: Hybrid Tensor Networks (hTNs)

We introduce the **hybrid tensor network (hTN)**, an abstraction that extends the TN formalism to enable unified execution and flexible partitioning of computation across quantum and classical accelerators, yielding a simple and coherent abstraction for hybrid quantum-classical workloads.

Fig. 3 shows an hTN describing the computation of the hybrid `forward` function in Lst. 1. The program executes several circuits over a batch of input data $X \in \mathbb{R}^{|i| \times 3}$ containing $|i| = d$ data points, each represented as a vector of dimension 3, and a set of observables O with $|O| = 2$. The qTensor on the left in Fig. 3 compactly describes this circuit's entire family. The qTensor has two indices, i and k, with sizes $|i| = d, |k| = 2$. These indices are referenced by *index-instructions*, associating each with several operations. For instance, instantiating i=1,k=0 replaces every index-instruction that depends on i with the second operation in its family, and every index-instruction of k with the first operation in its family.

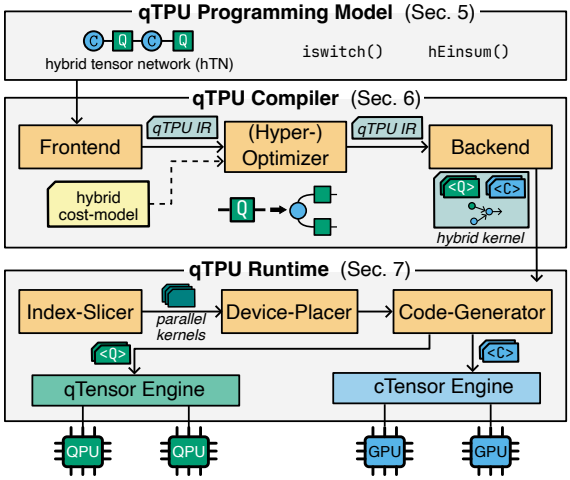


Figure 4: System overview (§ 4).

The hTN on the right in Fig. 3 compactly describes the hybrid program with the qTensor and a cTensor $V \in \mathbb{R}^{|k| \times |j|}$. The index k contracts with the qTensor's output index, while j denotes the dimension of the output vectors $y \in Y$. Contracting this hTN yields the result tensor $Y \in \mathbb{R}^{|i| \times |j|}$, corresponding to the output of the hybrid computation and containing the resulting ML-layer predictions for each input indexed by i.

Semantics. Unlike classical TNs, an hTN first executes its qTensors to its results-cTensor, before contracting the hTN to a tensor with einsum. An hTN, consisting of classical tensors $T_1^c, \ldots, T_n^c$, each with its own index set I_i^c, and qTensors $T_1^q, \ldots, T_m^q$, each with its own index set I_i^q, we contract as follows:

$$T_O = \text{einsum}(I_1^c, \ldots, I_n^c, I_1^q, \ldots, I_m^q \to O; T_1^c, \ldots, T_n^c, [\![T_1^q]\!], \ldots, [\![T_m^q]\!]),$$

where $[\![T^q]\!]$ denotes executing the qTensor for all its indices and transferring the results to a cTensor.

4 qTPU System Overview

To realize our hTN abstraction, we introduce qTPU, an end-to-end system that enables programmable and scalable hybrid computations. Fig. 4 shows an overview of the system.

To express hybrid programs with the hTN abstraction, users program in the **qTPU programming model**, which provides a simple, declarative interface based on the two primitives iswitch and hEinsum, while hiding the complexity of coordinating quantum and classical computation.

To automatically co-optimize a hybrid program, the **qTPU compiler** optimizes it to fully use both quantum and classical computing. The compiler's frontend first parses the program into the unified qTPU IR, after which the optimizer applies a hybrid quantum-classical cost model to identify Pareto-optimal rewrites that reduce quantum errors from noise and classical costs of operations and memory usage. Finally, the backend produces an optimized hybrid kernel comprising

device-agnostic quantum and classical kernels, along with a canonical operation schedule, ready for execution.

Finally, the **qTPU runtime** scalably executes the optimized hybrid program, adapting it to the available resources. To do so, its index slicer first partitions the hTN into smaller, independent, and parallelizable slices. The device placer then maps each concrete quantum and classical computation to an available device to maximize throughput, after which the code generator produces device-optimized kernel code for each computation. The hTN is then executed in a distributed pipeline, where the qTensor engine evaluates all qTensors on the available QPUs, and the cTensor engine contracts the resulting cTensors using the available classical accelerators.

5 qTPU Programming Model

We propose the qTPU programming model, which provides a unified abstraction for hybrid quantum-classical computation, enabling programmers to state hybrid computations *declaratively*, without their low-level execution details – such as manually stitching components into a single workflow.

Our qTPU model exposes two core constructs: (1) *quantum tensors*, created by annotating quantum programs with iswitch statements, and (2) a declarative multi-tensor contraction operator, hEinsum, which composes classical and quantum tensors into a single hybrid computation. These primitives are in addition to the classical tensor (cTensor) primitive, represented by a multi-dimensional array.

5.1 qTensor Primitives

A quantum tensor (qTensor) compactly represents a multi-index family of quantum kernels. It is structured as a quantum program annotated with iswitch operations, each binding a symbolic index to a discrete set of circuits. We consider a simple language t^Q of defining quantum circuits, consisting of gates and our novel iswitch operator:

$$t^Q := \text{gate}(G, \overline{q}) \mid \text{seq}(t^Q, t^Q) \mid \text{iswitch}(k, \overline{G}, \overline{q})$$

Here, G references a quantum gate, while $\overline{G}$ is a vector of gates, and seq refers to a sequence of instructions. $\overline{q}$ is a vector of unique qubit identifiers.

Materialization. Instead of executing the qTensor on a QPU, we could *materialize* it to a classical tensor, given an index set $\mathcal{I}$. U_G is the unitary transformation of gate G.

$$\begin{aligned}
[\![\text{gate}(G,\overline{q})]\!]^{\text{m}}_{\mathcal{I}} &= U_G \\
[\![\text{seq}(t_1^Q, t_2^Q)]\!]^{\text{m}}_{\mathcal{I}} &= [\![t_1^Q]\!]^{\text{m}}_{\mathcal{I}} \cdot [\![t_2^Q]\!]^{\text{m}}_{\mathcal{I}} \\
[\![\text{iswitch}(k,\overline{G},\overline{q})]\!]^{\text{m}}_{\mathcal{I}} &= U_{\overline{G}_{\mathcal{I}_k}}
\end{aligned}$$

5.2 hEinsum: Hybrid Einsum Primitive

We consider another language t^T for tensor computations, embedding quantum computations (t^Q) and our novel hEinsum operator. hEinsum *declaratively* specifies a hybrid TN operation, thus relieving the programmer from managing execution details, while enabling the compiler and runtime to execute it efficiently on QPUs and classical accelerators.

```
1  @quantum_tensor
2  def quantum_layer(X, W, O):
3      q = alloc_qubits(N)
4      for i in range(N):
5          # batch-dimension index
6          iswitch("i", [ry(x[i]) for x in X], q[i])
7      model(q, W)
8      # hidden-dimension index
9      iswitch("k", [apply_obs(o) for o in O], q)
10     measure(q)
11
12 def forward(X, W, V, O): # hybrid computation (declarative)
13     return hEinsum("jk,ik->ij", V, quantum_layer(X, W, O))
```

Listing 2: Unified Quantum-Classical ML example (cf. Lst. 1)

Operationally, hEinsum acts like a standard TN contraction over classical tensors, where qTensors are materialized first to cTensors. To formally state the semantics of our hEinsum term, we define our simple tensor language:

$$t^T := \text{cTensor}(T) \mid \text{qTensor}(t^Q) \mid \text{hEinsum}(\text{expr}; t_1^T, \ldots, t_n^T)$$

We can define the semantics of that language, in particular our hEinsum term, using the mathematical einsum operator (Sec. 3.2). Intuitively, we materialize any qTensor to its classical counterpart, over which we compute a classical einsum.

$$\begin{aligned}
&[\![\text{cTensor}(T)]\!]^{\text{s}}_{\mathcal{I}} &&= T \\
&[\![\text{qTensor}(t^Q)]\!]^{\text{s}}_{\mathcal{I}} &&= [\![t^Q]\!]^{\text{m}}_{\mathcal{I}} \\
&[\![\text{hEinsum}(I_1,\ldots,I_n \to O; T_1,\ldots,T_n)]\!]^{\text{s}}_{\mathcal{I}} &&= \\
&\qquad \text{einsum}(\text{expr}; [\![T_1]\!]^{\text{s}}_{I_1},\ldots,[\![T_n]\!]^{\text{s}}_{I_n}) \text{ where } \textstyle\bigcup_{k=1}^{n} I_k \subseteq \mathcal{I}
\end{aligned}$$

Summary. The qTPU programming model provides a simple, declarative framework for hybrid quantum-classical computation. QTensors introduce indexed quantum kernels via iswitch annotations, while hEinsum lifts classical einsum semantics to the hybrid setting. By deferring kernel specialization and execution until contraction time, the model separates logical specification from concrete realization, permitting the compiler and runtime to optimize placement and scheduling across QPUs and classical accelerators.

5.3 Example Continued

To demonstrate the versatility of the qTPU programming model, we revisit the hybrid ML task from Lst. 1, reimplemented in Lst. 2 (visualized in Fig. 3). The `quantum_layer` function (Line 2) defines a two-dimensional qTensor encoding the required circuit family: index `i` iterates over dataset X to select the input encoding, while `k` selects the measurement observable.

The iswitch primitive instantiates operations based on `i` and `k`. For instance, `i=1,k=0` maps to the second data point and first observable. This allows the programmer to declaratively specify the complete circuit family within the qTensor function, avoiding manual construction of individual circuits.

The `quantum_layer` kernel in Lst. 2 makes the qTensor concrete: each iswitch annotation becomes one index of the qTensor, so the kernel itself specifies the entire iswitch-indexed family. Executing the qTensor on a QPU yields a

cTensor whose entries are the measurement outcomes for each index assignment.

Finally, the hybrid computation in the `forward` function is realized via hEinsum, which defines the contraction over quantum and classical tensors (cf. Fig. 3). In this model, the programmer specifies the computation solely through tensor structure, while details of circuit scheduling, data movement, and coordination remain entirely hidden.

The QTPU programming model serves a dual role: as a user-facing interface for applications such as hybrid ML (Lst. 2), and as a compilation target that upstream tools emit into. For example, our circuit-knitting and error-mitigation pipelines (§ 8.2) lower their output directly into concise hEinsum expressions, obtaining unified optimization and execution without managing hybrid orchestration themselves.

6 QTPU Compiler

The QTPU compiler lowers the high-level, declarative hEinsum-description (§ 5) into an optimized hybrid kernel.

Workflow. Fig. 5 shows the overall workflow of the QTPU compiler. At the frontend, we parse the hEinsum into the intermediate representation (QTPU IR). The optimizer rewrites the QTPU IR to balance quantum and classical costs, leveraging both resource modalities. Finally, the backend emits an optimized hybrid kernel.

6.1 Frontend and QTPU IR

The frontend of the compiler converts a given hEinsum program (§ 5.2) to the unified QTPU IR.

qTensor graphs. We convert each qTensor into a graph $G_{\mathrm{QT}} = (V_{\mathrm{QT}}, E_{\mathrm{QT}})$ mirroring the structure of the underlying quantum circuit's TN (Fig. 2), allowing us to reason about which qTensor parts we can offload to classical operations. Each vertex $v \in V_{\mathrm{QT}}$ is a tuple (op_i, q_x), denoting applying operation op_i on qubit q_x. Edges are defined as (i) a *temporal* edge $\big((\mathrm{op}_i, q_x), (\mathrm{op}_k, q_x)\big) \in E_{\mathrm{QT}}$ if op_i immediately precedes op_k on qubit q_x, and (ii) a *spatial* edge $\big((\mathrm{op}_i, q_x), (\mathrm{op}_i, q_y)\big) \in E_{\mathrm{QT}}$ when op_i acts jointly on qubits q_x and q_y.

Operation graph. The operation graph is a directed acyclic graph capturing the order among computations in the hEinsum. Its initial nodes represent qTensors and cTensors, while its middle nodes are hEinsums. After transformations, we obtain a graph where each operation has two operands, allowing us to map each to an efficient classical implementation.

6.2 Hybrid Cost Model

The QTPU compiler optimizes an hEinsum under a dual-objective cost model that balances classical and quantum resources. It captures the classical contraction cost and the expected error of executing quantum circuits on noisy QPUs.

Classical cost. For a contraction $\mathrm{hEinsum}(I_1, \ldots, I_n \to O; T_1, \ldots, T_n)$, let $\mathcal{I} = I_1 \cup \cdots \cup I_n$, and let $|i|$ denote the size of index i. The FLOP cost of this contraction is $c_x = \prod_{i \in \mathcal{I} \setminus O} |i|$, i.e., the product of the sizes of all summed-out indices. The total classical cost of the hTN is then $C_{\mathrm{classical}} = \sum_{x \in \mathrm{hTN}} c_x$, the sum of the costs of all nodes in the operation graph.

Quantum cost (error). For each qTensor, we estimate the error of its quantum computation from the structure of its underlying circuit. Let $\mathcal{G}(Q)$ be the set of all quantum operations appearing in qTensor Q, and let $\varepsilon(g)$ denote the error probability of operation g due to noise on the QPU's. We define the per-qTensor error score as $E(Q) = 1 - \prod_{g \in \mathcal{G}(Q)} \big(1 - \varepsilon(g)\big)$. This score approximates the probability that at least one error occurs during the execution of Q. The overall quantum error is the maximum error score over all qTensors.

Optimization goal. The compiler seeks a Pareto-optimal front of points in the joint quantum-classical cost space, balancing classical FLOPs and quantum error to produce the most performant hybrid program for the given hardware.

6.3 Optimization Primitives

We optimize hybrid programs using optimization primitives illustrated in Fig. 6.

Tensorization and decomposition. Tensorization captures a linear combination of related hEinsums as a single hEinsum. As shown in Fig. 6(a), any linear combination of hEinsums differing only by small structural changes can be represented as one hEinsum where coefficients c_i are collected into a classical tensor c, and structural variations are encoded by a qTensor with iswitch operations.

When a qTensor contains separable subcircuits acting on disjoint qubit sets, it decomposes into an equivalent hEinsum of smaller qTensors, since subcircuit measurement outcomes are uncorrelated with the full result obtained by multiplying their individual outputs. This replaces one large quantum tensor with several smaller qTensors.

Spatial separation. Gate virtualization partitions circuits by cutting two-qubit gates into a linear combination of single-qubit operations, then recovers the original result by executing all variants and computing their weighted sum [90]. Spatial separation generalizes gate virtualization within our QTPU framework (Fig. 6(b)), replacing the original two-qubit operation with an equivalent hEinsum of one cTensor and one qTensor. When resulting subcircuits operate independently, tensor decomposition factorizes the qTensor into smaller, less noisy components.

Temporal separation. Temporal separation builds on wire cutting, which partitions circuits by cutting qubit wires at specific time steps [84]. The cut decomposes continuous qubit evolution into a linear combination of operations, splitting deep circuits into shallower fragments with reduced accumulated gate errors (Fig. 6 (c)). Applying tensorization and decomposition rewrites large quantum tensors into equivalent hEinsums with multiple shallower, less costly qTensors.

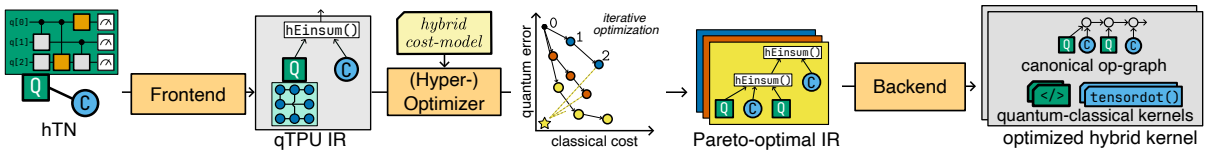


Figure 5: Compiler workflow (§ 6). *The frontend parses hTNs defined with hEinsum to the qTPU IR, the optimizer uses iterative graph partitioning to find Pareto-optimal IR variants, and the backend lowers IR to optimized hybrid quantum-classical kernels.*

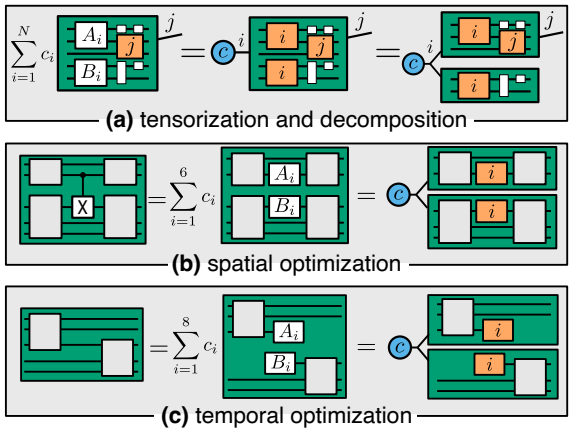


Figure 6: Semantic-preserving qTensor to hEinsum rewrites.

6.4 Optimizer

The optimizer applies the optimization primitives to achieve a Pareto-optimal hybrid cost. It systematically rewrites each qTensor, transforming infeasible quantum workloads into a more balanced hybrid computation.

Optimization workflow. We illustrate the workflow optimizing one qTensor into a hEinsum expression in Fig. 7. We start with the qTensor graph treated as a single partition, and the operation graph containing one qTensor node. At this point, the quantum error is high and the classical cost is zero, as shown by the upper-left point in the graph of Fig. 5.

In each iteration, we partition the qTensor graph into $p \geq 2$ balanced subgraphs to minimize cut edges. These boundary edges dictate the optimization primitive: temporal cuts trigger temporal optimization, and spatial cuts trigger spatial optimization (cf. Fig. 6). As shown in Fig. 7, each rewrite replaces the node with an equivalent subgraph of qTensors and cTensors. We greedily iterate by splitting the partition with the highest quantum cost.

The process continues until each rewrite decreases the distance to the Pareto-optimal point. This produces the cost trajectory shown in Fig. 8: each iteration reduces quantum cost (by shrinking qTensors) while increasing classical cost, thereby moving closer to the optimum. Once an iteration increases the distance to this optimum, we terminate.

Hyperparameter optimization. Because the effectiveness of each iteration depends on the chosen partitioning, we run the optimizer with multiple randomized hyperparameter configurations. These hyperparameters include the target partition count per iteration, p, and the relative weight of partition balance versus minimizing inter-partition edges.

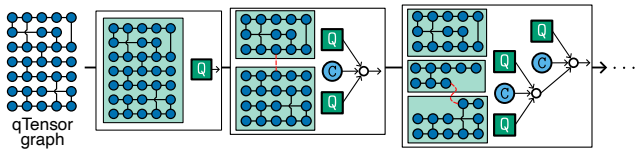


Figure 7: Optimizing a qTensor by iteratively partitioning and recursively building an optimized operation-graph.

Each hyperparameter configuration yields a distinct cost trajectory; from these, we select the Pareto-optimal solution, *i.e.*, the one with final cost closest to the theoretical optimum.

Optimization Constraints. The optimizer is subject to three primary optional constraints: (1) the maximum number of qubits per qTensor in the optimized hEinsum, reflecting the available physical resources of the underlying QPUs; (2) an upper bound on the quantum error accumulated within each qTensor; and (3) a cap on the allowable classical post-processing cost, measured in FLOPs. When such constraints are specified, we first restrict the solution space to Pareto-optimal candidates that satisfy all constraints, and then select the remaining Pareto-optimal solution.

6.5 Backend

Finally, the qTPU compiler lowers the optimized qTPU IR into concrete hybrid kernels.

Contraction scheduling. The operation graph is canonicalized so that every node has exactly two operands, ensuring that each contraction corresponds to a single accelerator operation that can efficiently be implemented on a classical device. We do so while optimizing the contraction order.

Lowering qTensors. Each qTensor across all slices is lowered into a single parametric quantum kernel. For every symbolic index of the qTensor, the compiler introduces a parameter, and each iswitch annotation becomes a corresponding `if-else` block that selects the appropriate gates.

7 qTPU Runtime

The qTPU runtime enables adaptable, scalable, and highly parallel execution of optimized hEinsum programs across a heterogeneous set of quantum and classical accelerators.

Workflow. Fig. 8 illustrates the workflow of the qTPU runtime. The runtime receives the compiler-optimized hybrid kernel, and execution proceeds in two conceptual phases.

In the *adaptation phase* (Fig. 8 (a)), the runtime adapts and lowers the hEinsum expression to the currently available resources. The index slicer and device placer partition

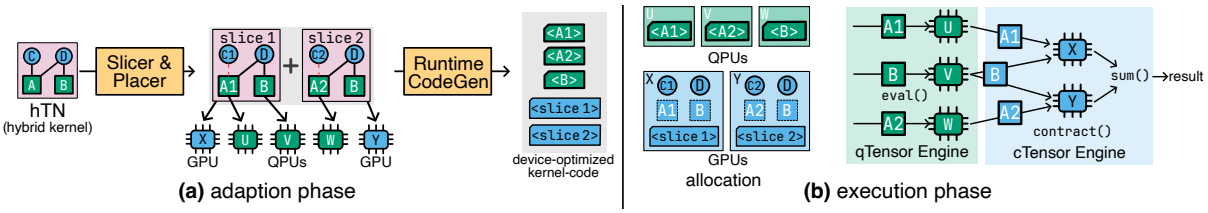


Figure 8: qTPU runtime (§ 7).*(a) Adaptation phase: the slicer and placer partition the hTN into independent slices assigned to QPUs and GPUs, then the code generator compiles device-specific kernels. (b) Execution phase: the qTensor engine evaluates quantum circuits in parallel on QPUs, the cTensor engine contracts results on classical devices to produce the final result.*

the expression into multiple independent slices to expose parallel work and assign each slice's quantum and classical operations to specific QPUs or classical accelerators. After placement, the code generator compiles the mapped quantum and classical operations into device-specific kernel code.

In the *execution phase* (Fig. 8 (b)), the runtime executes the sliced hEinsum across the distributed devices. The runtime first allocates all qTensors and cTensors and loads the generated kernels onto their assigned hardware. Execution proceeds in a map–reduce style: the qTensor engine evaluates all qTensors in parallel on QPUs, the cTensor engine contracts the resulting slices on classical accelerators, and the partial contractions are then combined into the result.

7.1 Index Slicing and Device Placement

During the adaptation phase (Fig. 8 (a)), the slicer and device placer prepare the optimized hEinsum for distributed execution across QPUs and classical accelerators, with the goal being to expose as much parallel quantum work as possible, as QPUs are typically the throughput bottleneck.

Let i be an index with domain $\{0,\dots,|i|-1\}$. Slicing along i simply means creating one hEinsum for each value of i, *i.e.,* a set of sub-expressions $\mathsf{hEinsum}(\text{expr}; T_1,\dots,T_n)\big|_{i=v}$. Each of these subproblems can be evaluated independently, and the original result is recovered by summing their outputs:

$$\mathsf{hEinsum}(\text{expr}; T_1,\dots,T_n) = \sum_{v=0}^{|i|-1} \mathsf{hEinsum}(\text{expr}; T_1,\dots,T_n)\big|_{i=v}.$$

In the example of Fig. 8, we slice an index with $|i| = 2$ at qTensor A, giving us two distinct slices.

To generate sufficient QPU parallelism, the slicer iteratively selects indices adjacent to each qTensor and slices along those that maximize the number of independent circuit instances, continuing until at least as many qTensor instances are produced as there are available QPUs.

This iterative procedure can be written compactly as:

$$\begin{aligned}
&S \leftarrow \{H\};\\
&\textbf{while } \textsc{qInst}(S) < P \textbf{ and } \textsc{adj}(S) \neq \emptyset :\\
&\quad i^* \leftarrow \arg\max_{i \in \textsc{adj}(S)} \textsc{qInst}(\textsc{slice}(S,i));\\
&\quad S \leftarrow \textsc{slice}(S,i^*)
\end{aligned}$$

where P is the target QPU parallelism, $\textsc{adj}(S)$ is the set of indices adjacent to any qTensor in some $H \in S$, $\textsc{qInst}(S) = \sum_{H\in S} \textsc{qInst}(H)$ counts qTensor instances across all slices, and $\textsc{slice}(S,i) = \bigcup_{H\in S} \textsc{slice}(H,i)$ extends per-expression slicing to the slice set. In the example of Fig. 8 with two QPUs, the slicer picks the $|i|$=2 index at qTensor A in a single step, producing two independent slices.

After slicing, the device placer assigns each qTensor instance to a QPU with available capacity, using a load-balanced assignment when the number of instances exceeds the devices. The remaining classical slice contractions are mapped to classical accelerators in a round-robin manner.

7.2 Runtime Code Generation

Following device placement, the runtime code generator compiles the hybrid kernel into device-specific executables: for every qTensor assigned to a QPU, it emits a quantum kernel using the native toolchain, while for classical slices, it compiles the shared operation graph into a single optimized kernel for the target accelerator using a tensor backend [15].

7.3 Tensor Engines

The tensor engines execute the sliced hEinsum across QPUs and classical accelerators.

The **qTensor engine** evaluates all qTensors on available QPUs. For each unique qTensor, it allocates quantum resources and the output buffer for its result cTensor entry, then loads the compiled quantum kernel onto the selected QPU. The compiled kernel is executed for each qTensor instance, returning results one by one into a result-cTensor.

The **cTensor engine** performs classical contractions for each slice. For every slice, it allocates slice-local cTensor buffers on the assigned classical accelerator and loads the compiled contraction kernel from the code generator. The engine waits asynchronously for cTensor inputs from the qTensor engine; once all required inputs arrive, it launches the slice's contraction kernel. Each slice produces a partial result, which the cTensor engine reduces across the sliced indices to yield the final output.

7.4 Fault Tolerance

The runtime handles failures gracefully, since QPUs and classical devices may become unavailable during execution. The sliced hEinsum structure naturally supports fault tolerance, as each slice represents an independent unit of work.

QPU calibrations. Unlike classical accelerators, QPUs require regular calibration to maintain low noise, typically going offline for about an hour daily [1]. The qTPU runtime

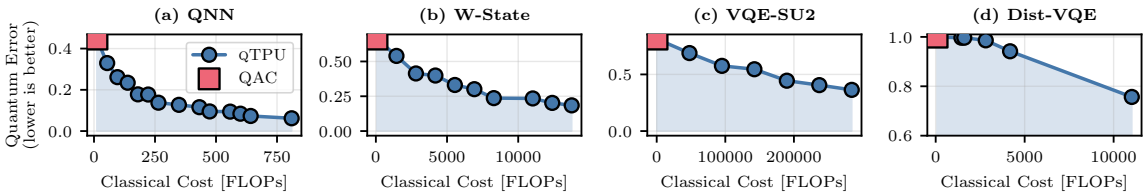


Figure 9: Compiler tradeoff (§ 8.4). *QAC produces a single solution, while QTPU exposes a flexible Pareto frontier.*

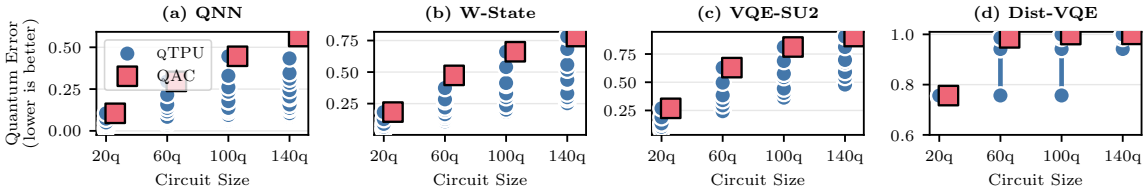


Figure 10: Compiler optimality analysis (§ 8.4). *While QAC yields a single solution with error scaling linearly with circuit size, QTPU maintains consistent error reduction across scale.*

models calibration as scheduled downtime: when a QPU enters calibration, the qTensor engine completes any in-flight circuit (usually seconds) and stops dispatching new qTensor instances to that device. Pending instances are reassigned to remaining QPUs using the same load-balanced policy.

Classical failures. If a classical accelerator fails during cTensor contraction, the cTensor engine re-executes only affected slice contractions. Since each slice depends solely on input cTensors (already in host memory), recovery requires no cross-slice coordination. Uncompleted contractions are reassigned to surviving accelerators.

8 Implementation and Evaluation

8.1 System Implementation

We implement an end-to-end QTPU prototype in Python.

Programming Model. We provide Python constructs for hEinsum and qTensors. To declare a qTensor, users annotate a QISKIT [71] quantum circuit with our custom iswitch operations. A hEinsum operation is then constructed with a einsum expression and corresponding qTensor objects and classical PYTORCH [63] tensors.

Compiler. The compiler uses KAHYPAR 1.3 [79] for graph partitioning at each recursive optimization level and explores the Pareto frontier of classical cost versus quantum error using randomized multi-start optimization. The backend returns a COTENGRA [35] operation graph and generates qTensors as CUDA-Q [18] kernels.

Runtime. The runtime leverages COTENGRA as an interface for contraction optimization and slicing, with PYTORCH for tensor operations. Our code generator JIT-compiles qTensors into optimized QPU-native code using CUDA-Q, and classical operations into GPU kernels using the PYTORCH toolchain.

8.2 Hybrid Application Implementation

To measure QTPU's performance on real-world hybrid applications, we implement three real-world applications in the QTPU programming model, enabling case studies on each (§ 8.6–§ 8.8): (i) for **hybrid ML**, we implement the example from Lst. 2, (ii) for **scalable hybrid computation**, we implement circuit knitting by decomposing a quantum circuit using the state-of-the-art circuit knitting library – QAC [11] and parsing the cut circuit into an equivalent hEinsum, and (iii) for **error mitigation**, we adopt probabilistic error cancellation (PEC), Pauli twirling and Zero-Noise-Extrapolation from the state-of-the-art error-mitigation library MITIQ [49], and convert them into an hEinsum.

	20q	40q	60q	80q	100q	120q	140q
QTPU [s]	0.76	1.05	1.25	1.77	2.01	2.34	2.51
QAC [s]	1.2	10.9	35.0	69.3	91.5	115.0	134.1
Speedup	**1.5**×	**10.3**×	**28.0**×	**39.2**×	**45.5**×	**49.1**×	**53.4**×

Table 1: Compile time scaling for VQE-SU2 circuits. QTPU's compile time is near-constant; QAC scales superlinearly.

8.3 Experimental Methodology

Experimental setup. We use a server with two AMD EPYC 9654 96-Core processors (384 cores with HT), 1.5 TB RAM, and an A40 GPU with 48GB HBM3 memory.

Baselines. We evaluate QTPU against Qiskit-Addon-Cutting (QAC) [11], the current state-of-the-art framework for hybrid quantum-classical execution through circuit decomposition. QAC is inherently hybrid, enabling scalable execution of large quantum circuits by partitioning the circuit into smaller subcircuits, executing them on a QPU, and classically postprocessing to obtain the result of the original circuit. We compare our runtime against cuTensorNet [58], NVIDIA's GPU-accelerated library for circuit simulation, to establish a classical-only baseline. For the error-mitigation case-study, we compare against MITIQ [49], the leading framework for quantum error mitigation. For our hybrid ML case-study, we compare against the BATCH method (Lst. 1), which is commonly used in quantum frameworks [9, 71]. Each baseline is a dedicated, state-of-the-art tool for its respective workload.

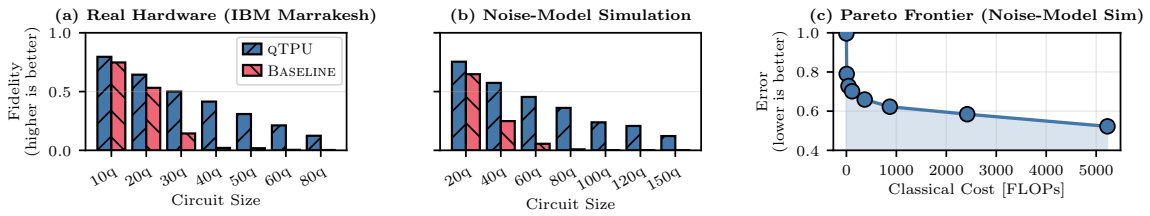


Figure 11: Hardware validation on real QPU (§ 8.4). *(a) Fidelity $\langle P \rangle$ on IBM Marrakesh for $n \in [10, 80]$ (higher is better; noiseless $\langle P \rangle$=+1). (b) Pauli-depolarizing noise-model simulation (ε_{cx}=10^{-2}, ε_{id}=10^{-3}) extending the fidelity sweep to 150 qubits. (c) Pareto frontier traversal at 80 qubits under the same noise model, plotting error (1−$\langle P \rangle$) vs. classical cost.*

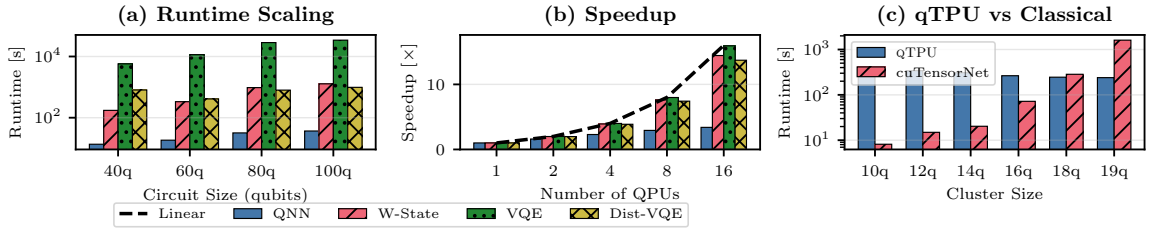


Figure 12: qTPU runtime analysis (§ 8.5). *(a) Total runtime scales with circuit size across benchmarks. (b) Near-linear speedup with multiple QPUs (14.4× at 16 QPUs for W-State) due to embarrassingly parallel qTensor execution. (c) qTPU vs. cuTensorNet on 100q Dist-VQE: qTPU outperforms classical beyond 18-qubit clusters, achieving 6.7× speedup at 19-qubit clusters.*

Benchmarks. We evaluate four quantum circuit families from MQT Bench [73], representative of hybrid quantum-classical applications: **QNN**, a quantum neural network for quantum machine learning [69]; **W-State**, preparing the n-qubit linearly entangled W-state [25]; **VQE-SU2**, a Variational Quantum Eigensolver ansatz with efficient $SU(2)$ rotations for quantum chemistry [66, 70]; and **Dist-VQE**, a distributed VQE variant with clustered connectivity modeling natural circuit partitioning across multiple QPUs.

Metrics. We evaluate qTPU using the following metrics: (1) **Quantum error**: The estimated probability of at least one error occurring during circuit execution (§ 6.2). We assume a uniform error model with 10^{-3} for single-qubit gates and 10^{-2} for two-qubit gates [1]. This uniform model reflects typical gate error rates on current superconducting QPUs and serves as a device-agnostic baseline. (2) **Classical cost**: The number of floating-point operations (FLOPs) required for classical computation (§ 6.2). (3) **Runtime**: Wall-clock execution time in seconds, measured either for classical or quantum or end-to-end hybrid execution. (4) **Code size**: The number of lines of generated CUDA-Q quantum-kernel code. (5) **QPU time**: Estimated quantum hardware execution time, computed by transpiling circuits to IBM Marrakesh using Qiskit's ASAP scheduling with optimization level 3 using 1000 shots [1, 71]. Per-shot execution time is deterministic, fixed by the scheduled gate sequence and device-level reset/readout overhead [71]. We validated this estimator against measured IBM Marrakesh job metadata: it matches monolithic QPU time within ±2% and is slightly conservative for cut variants, making qTPU's reported QPU time an upper bound. (6) **Fidelity**: The retention of a weight-n Clifford observable P under circuit execution, $\langle P \rangle \in [-1, 1]$. A noiseless execution gives $\langle P \rangle = +1$; hardware noise drives it toward 0.

8.4 qTPU Compiler

RQ1 (Compiler trade-off): *Can the compiler trade quantum error for classical computation?* We partition each 100-qubit benchmark into 50-qubit subcircuits using both QAC and qTPU, with qTPU configured to run 50 hyperparameter trials, retaining only Pareto-optimal solutions.

Fig. 9 shows that while QAC produces a single fixed solution, qTPU offers multiple configurations trading classical cost for quantum error. Mid-range solutions yield 1.5–3.4× error reduction, while maximum classical investment achieves 2.2–7.2× improvement across all benchmarks. At equal classical cost, both frameworks provide the same solution-quality.
RQ2 (Compiler scalability): *Does the compiler scale to large qTensors?* We vary the circuit size from 20 to 140 qubits, cutting each circuit into subcircuits of 50% the original size.

Fig. 10 shows qTPU maintains stable error reduction across scales, ranging from 2–26× depending on the benchmark. While QAC error scales linearly with circuit growth, qTPU's Pareto frontiers consistently trade FLOPs for reduced quantum error without diminishing returns.

Table 1 shows the compile time on VQE-SU2: QAC scales superlinearly (1.2s to 134s) due to exhaustive search, while qTPU maintains ~1–3s (up to 53× speedup for 140 qubits). Although graph partitioning, a technique used throughout the compiler, is NP-hard in general, the structure of quantum circuits makes it tractable for KaHyPar 's multi-level hypergraph algorithm [79].

Note that qTPU's cost model (§ 6.2) accepts any user-

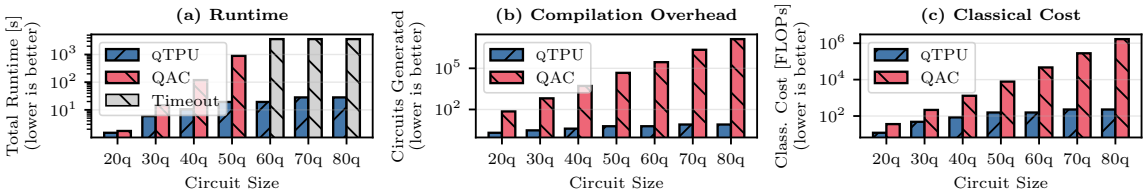


Figure 13: Scalable hybrid computing for QNN on a 10-qubit QPU (§ 8.7). We partition QNN circuits (20–80 qubits) into 10-qubit subcircuits using qTPU and QAC. *(a) qTPU scales to 80 qubits in <30s while QAC times out beyond 50 qubits. (b) qTPU generates 10–42× fewer subcircuits. (c) qTPU's classical postprocessing requires* 10^2–10^3 *FLOPs vs. QAC's* 10^6 *FLOPs.*

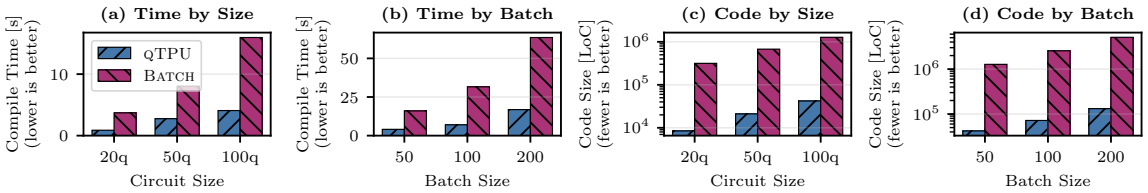


Figure 14: Case study: hybrid machine learning with Batch execution (Lst. 1) vs. qTPU (Lst. 2) (§ 8.6). *(a) Compilation time by circuit size, (b) compilation time by batch size, (c) generated code size by circuit size, (d) generated code size by batch size.*

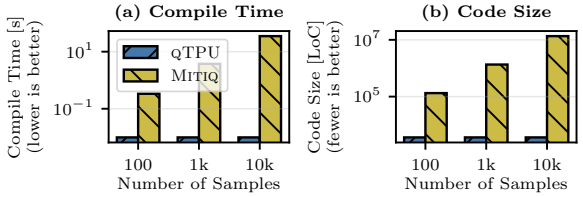


Figure 15: Quantum error mitigation (QEM) vs Mitiq the 100-qubit QNN benchmark (§ 8.8). ***(a)*** *qTPU is 3,500× faster in compilation.* ***(b)*** *qTPU generates 3,700× fewer lines of code.*

supplied error model. The negligible compile time makes recompiling with an up-to-date model of current calibration data practical for obtaining the best possible Pareto front.

RQ3 (Hardware validation): *Does qTPU's error reduction hold on real quantum hardware?* We run QNN circuits from 10 to 80 qubits on IBM Marrakesh with both qTPU and a monolithic Baseline, partitioning each circuit into 10-qubit subcircuits with qTPU.

Fig. 11 (a) shows fidelity on real hardware: Baseline collapses from 0.53 at 20q to 0.003 at 80q, while qTPU retains 0.12 at 80q—a 46× improvement. Fig. 11 (b) extends the sweep to n=150 under a Pauli-depolarizing noise model (ε_{cx}=10^{-2}, ε_{id}=10^{-3}), reproducing the same trend: Baseline reaches the noise floor by 100q, while qTPU maintains fidelity of 0.12.

Pareto frontier traversal. To test whether the compiler's cost model translates into measurable error reduction along the full frontier, we enumerate the Pareto solutions of an 80-qubit QNN circuit and execute one point per partition size. Fig. 11 (c) plots error (1−fidelity) against classical cost: error drops monotonically from 0.997 at the zero-cost solution to 0.522 at the most aggressive optimization—a 0.48-unit reduction, confirming that higher classical investment purchases lower quantum error.

8.5 qTPU Runtime

RQ4 (Runtime analysis): *What is qTPU's runtime performance and where is it spent?* We measure total runtime, including compilation, classical postprocessing, and QPU execution, on a 15-qubit QPU, requiring the compiler to optimize qTensors to a circuit size of 15 qubits or fewer.

Fig. 12(a) shows runtime scaling with circuit size across benchmarks. For 100-qubit circuits, quantum execution dominates at 75-100% of total runtime, while classical contraction accounts for <0.01% and compilation takes 2-3s. Network and I/O overhead is negligible compared to quantum runtime; further, qTPU's compact hTN representation transfers far less circuit data than approaches that enumerate variants.

RQ5 (Scalability): *Can the runtime leverage multiple QPUs?* Fig. 12(b) shows near-linear speedup as QPU count increases from 1 to 16, achieving 14.4× speedup with 16 QPUs (90% efficiency). The slicer produces embarrassingly parallel qTensor instances, thereby minimizing distribution overhead.

RQ6 (Comparison to classical simulation): *How does qTPU perform against pure classical simulation?* We evaluate 100-qubit Dist-VQE circuits with clustered connectivity. qTPU executes on 4 QPUs with 1,000 shots per subcircuit; cuTensorNet performs full simulation. qTPU's reported runtime includes compilation, estimated QPU execution, and classical contraction. cuTensorNet's runtime is total wall-clock simulation time.

Fig. 12 (c) shows that as cluster size increases, qTPU runtime decreases (fewer qTensors and cTensors) while cuTensorNet runtime grows exponentially (larger clusters yield harder contractions). qTPU outperforms classical beyond 18-qubit clusters, achieving 6.7× speedup at 19-qubits.

8.6 Case-Study: Hybrid Machine Learning

We compare manual batching (Lst. 1) against qTPU's unified model (Lst. 2) for evaluating hybrid quantum-classical ML. We use QNN quantum kernels of 20–100 qubits with

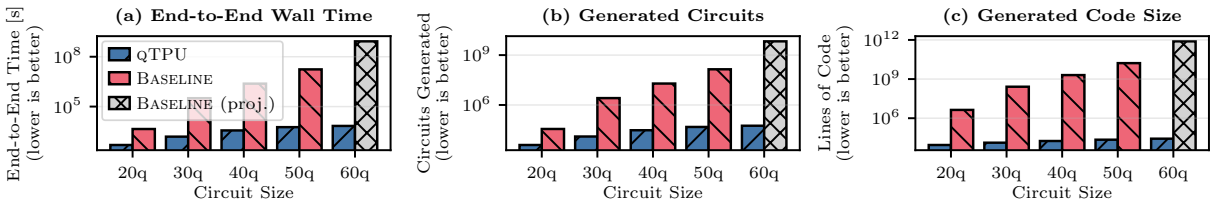


Figure 16: End-to-end application: QNN with circuit cutting, ZNE, and hybrid ML combined in one pipeline (§ 8.9). *(a) At 50q, end-to-end time drops from* $\sim 1.7{\cdot}10^7$*s to* $\sim 5.8{\cdot}10^3$*s* ($\sim 3{\cdot}10^3\times$). *(b)* QTPU *generates* $\sim 3{\cdot}10^3\times$ *fewer subcircuits at 50q (48k vs. 140M). (c)* QTPU *emits* $\sim 7{\cdot}10^5\times$ *less kernel code at 50q (23k vs. 16.7B lines).*

$X \in \mathbb{R}^{b\times h}$, $V \in \mathbb{R}^{h\times f}$, batch sizes $b \in \{50, 100, 200\}$, hidden dimension $h = 20$, and feature dimension $f = 8$. Manual batching (Lst. 1) is the standard practice in QML frameworks [9, 77].

RQ7 (Compilation Scalability): *How does compilation time scale with problem size?* Fig. 14 (a,b) shows compilation scaling with circuit and batch size. QTPU achieves 3.7× average speedup (up to 6×) over batch execution. Critically, QTPU scales sublinearly with batch size, whereas batch execution scales linearly by compiling each circuit independently. At 100 qubits, batch execution grows from 3.1s to 63.5s as batch size increases from 10 to 200, while QTPU grows only from 4.8s to 16.9s, a speedup of up to 4.5×.

RQ8 (Code Generation Overhead): *What is the impact on generated code size?* Fig. 14 (c,d) shows QTPU's code generation efficiency. By compiling the quantum kernel once with parametric control flow rather than instantiating separate circuits, QTPU achieves an average reduction of 33× (up to 48×). At 100 qubits with batch size 200, QTPU generates 132k lines vs. batch execution's 5.1M lines—a 38.7× reduction. This directly translates into faster compilation and a smaller memory footprint, with benefits that scale with batch size.

8.7 Case Study: Scalable Hybrid Computing

In this case study, we implement scalable hybrid computing using circuit knitting on resource-constrained hardware. We run QNN benchmarks on 20-80 qubits, targeting a 10-qubit QPU and enforcing aggressive circuit partitioning.

RQ9 (End-to-end runtime): *What impact does* QTPU *have on the end-to-end runtime with increasing problem size?* Fig. 13 (a) shows total runtime: QAC scales exponentially and times out beyond 40 qubits (20-minute limit), while qTPU completes 80-qubit circuits in under 30 seconds. This dramatic difference stems from QTPU's hTN representation, avoiding redundancies by compactly encoding the entire computation.

RQ10 (Compilation overhead): *How many circuits does* QTPU *generate with increasing problem size?* Fig. 13 (b) reveals the root cause of QAC 's exponential scaling: circuit generation grows combinatorially (12 to 6,480 circuits from 20q to 50q), whereas QTPU produces 10–42× fewer subcircuit instances. The hTN abstraction eliminates this overhead by representing subcircuits declaratively as qTensors rather than enumerating explicit circuit variants.

RQ11 (Postprocessing overhead): *What is the classical cost in FLOPs that* QTPU *incurs with increasing problem size?* Fig. 13 (c) shows the classical postprocessing cost: QTPU maintains modest overhead ($< 10^3$ FLOPs) across all sizes, while QAC grows to 10^6 FLOPs exponentially. The hTN formulation performs only the minimum necessary tensor contractions by structure-aware optimization, whereas QAC's explicit circuit enumeration incurs exponential classical reconstruction overhead proportional to the circuit size.

8.8 Case Study: Quantum Error Mitigation (QEM)

RQ12 (QEM scalability): *What impact does* QTPU *have on the compile time and code size requirements of a hybrid QEM workflow?* We implement three major error mitigation strategies (§ 8.2) using both QTPU and Mitiq on quantum circuits with 200 single-qubit gates. For Mitiq, we vary the sample count from 100 to 10,000 circuits; QTPU represents the full task symbolically as a hEinsum, independent of sample count.

Fig. 15 compares compile-time overhead and code size. While Mitiq explicitly generates each circuit variant, QTPU represents the entire ensemble symbolically, encoding up to $4^{200} \approx 10^{120}$ configurations (for combined PEC+Twirling applied to all gates) as a single qTensor. QTPU maintains constant ~10ms overhead regardless of sample count, while Mitiq scales linearly from ~335ms at 100 samples to 35s at 10,000 samples (~3,550×). Similarly, QTPU generates only ~3.7k lines of code, whereas Mitiq generates 13.5M (~3,700×).

8.9 Case Study: End-to-End Application

In this case study, we combine all three use cases—circuit cutting, ZNE error mitigation, and batched hybrid ML—into one end-to-end workload. We run QNN circuits from 20–60 qubits, partitioning into 10-qubit subcircuits, applying Richardson ZNE at three noise scales, and evaluating over a batch of 20 inputs. QTPU expresses the full pipeline as a single hEinsum; the BASELINE composes QAC, MITIQ, and a manual BATCH loop in a pipeline.

RQ13 (End-to-end runtime): *How does the unified hEinsum compare against pipeline composition?* The BASELINE circuit generation-time exceeded our timeout at 60q. Fig. 16 (a) shows end-to-end wall time. At 50 qubits, time drops from $\sim 1.7{\cdot}10^7$s to $\sim 5.8{\cdot}10^3$s ($\sim 3{\cdot}10^3\times$). QPU execution dominates (>99%). The core speedup from QTPU comes through the fact that it needs to enumerate fewer quantum circuit instantiations due to its compact hTN form.

RQ14 (Generation overhead): *What is the cost of pipeline composition in circuits and code?* Fig. 16 (b,c) shows the generation cost: at 50q the baseline materializes 140M subcircuits vs. 48k for QTPU ($\sim 3{\cdot}10^3\times$ fewer), and emits 16.7B lines of kernel code vs. 23k ($\sim 7{\cdot}10^5\times$ less). Composition multiplies each library's explicit enumeration, while QTPU's qTensors collapse it into a single symbolic expression.

8.10 Developer Productivity

The QTPU programming model reduces the effort of writing hybrid workloads. The hybrid-ML case study (§ 8.6) makes the contrast concrete: the BATCH formulation in Lst. 1 requires a two-stage `forward` pass that enumerates circuits, dispatches them, and threads the results into a classical layer, while the QTPU formulation in Lst. 2 collapses this into a single qTensor with two iswitch indices and one hEinsum. The same primitives carry over to the other studies: scalable hybrid execution (§ 8.7) treats QAC 's cutting recipe as an hEinsum, and error mitigation (§ 8.8) expresses QEM-routines as hTNs rather than separate library calls. Each baseline is scoped to a single workload; composing them requires chaining tools, whereas QTPU expresses any workload, or arbitrary compositions, as one hTN (§ 8.9).

9 Related Work

Hybrid quantum-classical applications. A wide range of quantum applications inherently rely on hybrid programming and execution models. Variational algorithms [27, 66], quantum machine learning [10, 51], quantum error mitigation [13, 20–22, 26, 65, 85, 86, 89, 98], and hybrid runtime systems [33,34,84,88,90] all interleave quantum circuit execution with classical optimization and/or postprocessing.

However, current approaches remain constrained by fragmented programming models, a lack of holistic optimizations, and a lack of resource-adaptable execution. In contrast, QTPU provides a unifying abstraction based on hTNs that encode entire hybrid computations. This enables global optimization, hardware-aware compilation, and flexible execution across QPUs and classical accelerators.

Tensor networks (TNs) in quantum simulation. TN-based quantum circuit simulation represents quantum gates and states as tensors, where simulation is performed through systematic tensor contraction operations [7, 56, 62, 76, 96, 97]. Prior work has used hybrid tensor-network ansätze as problem-specific variational constructions for particular Hamiltonians with small QPUs encoding part of a target wavefunction [80, 95]. QTPU is the first system to formalize hybrid tensor networks as a programming model that can concisely capture arbitrary hybrid workloads. Beyond simulation, TN abstractions also underpin applications such as quantum error analysis [87], which QTPU could naturally accelerate by exploiting its hybrid quantum-classical execution model.

Tensor optimization in classical systems. Deep learning compilers optimize tensor operators through scheduling transformations, operator fusion, and hardware mapping to maximize throughput on classical accelerators [15, 23, 43, 55, 82, 93, 100–102]. Similarly, general tensor compilers focus on efficient contraction and algebraic manipulation of dense and sparse tensors, addressing memory layouts, parallelization, and sparsity exploitation for scientific computing and machine learning workloads [6, 24, 32, 42, 46–48, 94]. QTPU addresses fundamentally different challenges: rather than optimizing dataflow graphs of deterministic, fully materialized tensor operations, QTPU optimizes hybrid tensor *network* contractions that integrate quantum and classical processing, requiring the compiler to determine an optimal trade-off between the two compute paradigms.

10 Conclusion

We introduce the hybrid tensor network (hTN) abstraction, a unified framework that seamlessly integrates quantum and classical computation. hTNs address current hybrid computing limitations—programming fragmentation, static partitioning, and poor scalability—through holistic representation and optimization. Our system, QTPU, realizes hTNs through a declarative programming model, a compiler that balances quantum error with classical cost, and an adaptable runtime for heterogeneous hybrid quantum-classical resources.

Evaluation across hybrid ML, circuit knitting, and error mitigation demonstrates that QTPU achieves up to 53× faster compilation, 7.2× lower quantum error rate, 3,700× code reduction, and 10,000× lower classical overhead than state-of-the-art baselines. Through its concise, flexible, and scalable abstraction for quantum-classical acceleration, QTPU enables automated and efficient heterogeneous resource utilization for next-generation quantum-enhanced applications.

Artifact. QTPU is publicly available at https://github.com/TUM-DSE/qtpu.

Acknowledgments. Supported by the Bavarian State Ministry of Science and the Arts, as part of the Munich Quantum Valley (MQV), grant number 6090181.

References

[1] Ibm quantum. https://www.ibm.com/quantum. Accessed: 2025-12-05.

[2] Quantum error correction below the surface code threshold. *Nature*, 638(8052):920–926, 2025.

[3] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: A system for

Large-Scale machine learning. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 265–283, Savannah, GA, November 2016. USENIX Association.

[4] Rajeev Acharya, Dmitry A. Abanin, Laleh Aghababaie-Beni, Igor Aleiner, Trond I. Andersen, Markus Ansmann, Frank Arute, Kunal Arya, Abraham Asfaw, Nikita Astrakhantsev, Juan Atalaya, Ryan Babbush, Dave Bacon, Brian Ballard, Joseph C. Bardin, Johannes Bausch, Andreas Bengtsson, Alexander Bilmes, Sam Blackwell, Sergio Boixo, Gina Bortoli, Alexandre Bourassa, Jenna Bovaird, Leon Brill, Michael Broughton, David A. Browne, Brett Buchea, Bob B. Buckley, David A. Buell, Tim Burger, Brian Burkett, Nicholas Bushnell, Anthony Cabrera, Juan Campero, Hung-Shen Chang, Yu Chen, Zijun Chen, Ben Chiaro, Desmond Chik, Charina Chou, Jahan Claes, Agnetta Y. Cleland, Josh Cogan, Roberto Collins, Paul Conner, William Courtney, Alexander L. Crook, Ben Curtin, Sayan Das, Alex Davies, Laura De Lorenzo, Dripto M. Debroy, Sean Demura, Michel Devoret, Agustin Di Paolo, Paul Donohoe, Ilya Drozdov, Andrew Dunsworth, Clint Earle, Thomas Edlich, Alec Eickbusch, Aviv Moshe Elbag, Mahmoud Elzouka, Catherine Erickson, Lara Faoro, Edward Farhi, Vinicius S. Ferreira, Leslie Flores Burgos, Ebrahim Forati, Austin G. Fowler, Brooks Foxen, Suhas Ganjam, Gonzalo Garcia, Robert Gasca, Élie Genois, William Giang, Craig Gidney, Dar Gilboa, Raja Gosula, Alejandro Grajales Dau, Dietrich Graumann, Alex Greene, Jonathan A. Gross, Steve Habegger, John Hall, Michael C. Hamilton, Monica Hansen, Matthew P. Harrigan, Sean D. Harrington, Francisco J. H. Heras, Stephen Heslin, Paula Heu, Oscar Higgott, Gordon Hill, Jeremy Hilton, George Holland, Sabrina Hong, Hsin-Yuan Huang, Ashley Huff, William J. Huggins, Lev B. Ioffe, Sergei V. Isakov, Justin Iveland, Evan Jeffrey, Zhang Jiang, Cody Jones, Stephen Jordan, Chaitali Joshi, Pavol Juhas, Dvir Kafri, Hui Kang, Amir H. Karamlou, Kostyantyn Kechedzhi, Julian Kelly, Trupti Khaire, Tanuj Khattar, Mostafa Khezri, Seon Kim, Paul V. Klimov, Andrey R. Klots, Bryce Kobrin, Pushmeet Kohli, Alexander N. Korotkov, Fedor Kostritsa, Robin Kothari, Borislav Kozlovskii, John Mark Kreikebaum, Vladislav D. Kurilovich, Nathan Lacroix, David Landhuis, Tiano Lange-Dei, Brandon W. Langley, Pavel Laptev, Kim-Ming Lau, Loïck Le Guevel, Justin Ledford, Joonho Lee, Kenny Lee, Yuri D. Lensky, Shannon Leon, Brian J. Lester, Wing Yan Li, Yin Li, Alexander T. Lill, Wayne Liu, William P. Livingston, Aditya Locharla, Erik Lucero, Daniel Lundahl, Aaron Lunt, Sid Madhuk, Fionn D. Malone, Ashley Maloney, Salvatore Mandrà, James Manyika, Leigh S. Martin, Orion Martin, Steven Martin, Cameron Maxfield, Jarrod R. McClean, Matt McEwen, Seneca Meeks, Anthony Megrant, Xiao Mi, Kevin C. Miao, Amanda Mieszala, Reza Molavi, Sebastian Molina, Shirin Montazeri, Alexis Morvan, Ramis Movassagh, Wojciech Mruczkiewicz, Ofer Naaman, Matthew Neeley, Charles Neill, Ani Nersisyan, Hartmut Neven, Michael Newman, Jiun How Ng, Anthony Nguyen, Murray Nguyen, Chia-Hung Ni, Murphy Yuezhen Niu, Thomas E. O'Brien, William D. Oliver, Alex Opremcak, Kristoffer Ottosson, Andre Petukhov, Alex Pizzuto, John Platt, Rebecca Potter, Orion Pritchard, Leonid P. Pryadko, Chris Quintana, Ganesh Ramachandran, Matthew J. Reagor, John Redding, David M. Rhodes, Gabrielle Roberts, Eliott Rosenberg, Emma Rosenfeld, Pedram Roushan, Nicholas C. Rubin, Negar Saei, Daniel Sank, Kannan Sankaragomathi, Kevin J. Satzinger, Henry F. Schurkus, Christopher Schuster, Andrew W. Senior, Michael J. Shearn, Aaron Shorter, Noah Shutty, Vladimir Shvarts, Shraddha Singh, Volodymyr Sivak, Jindra Skruzny, Spencer Small, Vadim Smelyanskiy, W. Clarke Smith, Rolando D. Somma, Sofia Springer, George Sterling, Doug Strain, Jordan Suchard, Aaron Szasz, Alex Sztein, Douglas Thor, Alfredo Torres, M. Mert Torunbalci, Abeer Vaishnav, Justin Vargas, Sergey Vdovichev, Guifre Vidal, Benjamin Villalonga, Catherine Vollgraff Heidweiller, Steven Waltman, Shannon X. Wang, Brayden Ware, Kate Weber, Travis Weidel, Theodore White, Kristi Wong, Bryan W. K. Woo, Cheng Xing, Z. Jamie Yao, Ping Yeh, Bicheng Ying, Juhwan Yoo, Noureldin Yosri, Grayson Young, Adam Zalcman, Yaxing Zhang, Ningfeng Zhu, Nicholas Zobrist, Google Quantum AI, and Collaborators. Quantum error correction below the surface code threshold. *Nature*, 638(8052):920–926, 2025.

[5] Frank Arute, Kunal Arya, Ryan Babbush, Dave Bacon, Joseph C Bardin, Rami Barends, Rupak Biswas, Sergio Boixo, Fernando GSL Brandao, David A Buell, et al. Quantum supremacy using a programmable superconducting processor. *Nature*, 574(7779):505–510, 2019.

[6] Manya Bansal, Olivia Hsu, Kunle Olukotun, and Fredrik Kjolstad. Mosaic: An interoperable compiler for tensor algebra. *Proc. ACM Program. Lang.*, 7(PLDI), June 2023.

[7] Aleksandr Berezutskii, Minzhao Liu, Atithi Acharya, Roman Ellerbrock, Johnnie Gray, Reza Haghshenas, Zichang He, Abid Khan, Viacheslav Kuzmin, Dmitry Lyakh, et al. Tensor networks for quantum computing. *Nature Reviews Physics*, pages 1–13, 2025.

[8] Ville Bergholm, Josh Izaac, Maria Schuld, Christian Gogolin, Shahnawaz Ahmed, Vishnu Ajith, M. Sohaib Alam, Guillermo Alonso-Linaje, B. AkashNarayanan,

Ali Asadi, Juan Miguel Arrazola, Utkarsh Azad, Sam Banning, Carsten Blank, Thomas R Bromley, Benjamin A. Cordier, Jack Ceroni, Alain Delgado, Olivia Di Matteo, Amintor Dusko, Tanya Garg, Diego Guala, Anthony Hayes, Ryan Hill, Aroosa Ijaz, Theodor Isacsson, David Ittah, Soran Jahangiri, Prateek Jain, Edward Jiang, Ankit Khandelwal, Korbinian Kottmann, Robert A. Lang, Christina Lee, Thomas Loke, Angus Lowe, Keri McKiernan, Johannes Jakob Meyer, J. A. Montañez-Barrera, Romain Moyard, Zeyue Niu, Lee James O'Riordan, Steven Oud, Ashish Panigrahi, Chae-Yeun Park, Daniel Polatajko, Nicolás Quesada, Chase Roberts, Nahum Sá, Isidor Schoch, Borun Shi, Shuli Shu, Sukin Sim, Arshpreet Singh, Ingrid Strandberg, Jay Soni, Antal Száva, Slimane Thabet, Rodrigo A. Vargas-Hernández, Trevor Vincent, Nicola Vitucci, Maurice Weber, David Wierichs, Roeland Wiersema, Moritz Willmann, Vincent Wong, Shaoming Zhang, and Nathan Killoran. Pennylane: Automatic differentiation of hybrid quantum-classical computations, 2022.

[9] Ville Bergholm, Josh Izaac, Maria Schuld, Christian Gogolin, Shahnawaz Ahmed, Vishnu Ajith, M Sohaib Alam, Guillermo Alonso-Linaje, B AkashNarayanan, Ali Asadi, et al. Pennylane: Automatic differentiation of hybrid quantum-classical computations. *arXiv preprint arXiv:1811.04968*, 2018.

[10] Jacob Biamonte, Peter Wittek, Nicola Pancotti, Patrick Rebentrost, Nathan Wiebe, and Seth Lloyd. Quantum machine learning. *Nature*, 549(7671):195–202, 2017.

[11] Agata M. Brańczyk, Almudena Carrera Vazquez, Daniel J. Egger, Bryce Fuller, Julien Gacon, James R. Garrison, Jennifer R. Glick, Caleb Johnson, Saasha Joshi, Edwin Pednault, C. D. Pemmaraju, Pedro Rivero, Ibrahim Shehzad, and Stefan Woerner. Qiskit addon: circuit cutting. https://github.com/Qiskit/qiskit-addon-cutting, 2024.

[12] Sergey Bravyi, Oliver Dial, Jay M. Gambetta, Darío Gil, and Zaira Nazario. The future of quantum computing with superconducting qubits. *Journal of Applied Physics*, 132(16):160902, 10 2022.

[13] Sergey Bravyi, Sarah Sheldon, Abhinav Kandala, David C. Mckay, and Jay M. Gambetta. Mitigating measurement errors in multiqubit experiments. *Phys. Rev. A*, 103:042605, Apr 2021.

[14] Matthias C Caro, Hsin-Yuan Huang, Marco Cerezo, Kunal Sharma, Andrew Sornborger, Lukasz Cincio, and Patrick J Coles. Generalization in quantum machine learning from few training data. *Nature communications*, 13(1):4919, 2022.

[15] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. TVM: An automated End-to-End optimizing compiler for deep learning. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 578–594, Carlsbad, CA, October 2018. USENIX Association.

[16] Neng-Chun Chiu, Elias C. Trapp, Jinen Guo, Mohamed H. Abobeih, Luke M. Stewart, Simon Hollerith, Pavel L. Stroganov, Marcin Kalinowski, Alexandra A. Geim, Simon J. Evered, Sophie H. Li, Xingjian Lyu, Lisa M. Peters, Dolev Bluvstein, Tout T. Wang, Markus Greiner, Vladan Vuletić, and Mikhail D. Lukin. Continuous operation of a coherent 3,000-qubit system. *Nature*, 646:1075–1080, 2025.

[17] T. A. Cochran, B. Jobst, E. Rosenberg, Y. D. Lensky, G. Gyawali, N. Eassa, M. Will, A. Szasz, D. Abanin, R. Acharya, L. Aghababaie Beni, T. I. Andersen, M. Ansmann, F. Arute, K. Arya, A. Asfaw, J. Atalaya, R. Babbush, B. Ballard, J. C. Bardin, A. Bengtsson, A. Bilmes, A. Bourassa, J. Bovaird, M. Broughton, D. A. Browne, B. Buchea, B. B. Buckley, T. Burger, B. Burkett, N. Bushnell, A. Cabrera, J. Campero, H.-S. Chang, Z. Chen, B. Chiaro, J. Claes, A. Y. Cleland, J. Cogan, R. Collins, P. Conner, W. Courtney, A. L. Crook, B. Curtin, S. Das, S. Demura, L. De Lorenzo, A. Di Paolo, P. Donohoe, I. Drozdov, A. Dunsworth, A. Eickbusch, A. Moshe Elbag, M. Elzouka, C. Erickson, V. S. Ferreira, L. Flores Burgos, E. Forati, A. G. Fowler, B. Foxen, S. Ganjam, R. Gasca, É Genois, W. Giang, D. Gilboa, R. Gosula, A. Grajales Dau, D. Graumann, A. Greene, J. A. Gross, S. Habegger, M. Hansen, M. P. Harrigan, S. D. Harrington, P. Heu, O. Higgott, J. Hilton, H.-Y. Huang, A. Huff, W. Huggins, E. Jeffrey, Z. Jiang, C. Jones, C. Joshi, P. Juhas, D. Kafri, H. Kang, A. H. Karamlou, K. Kechedzhi, T. Khaire, T. Khattar, M. Khezri, S. Kim, P. Klimov, B. Kobrin, A. Korotkov, F. Kostritsa, J. Kreikebaum, V. Kurilovich, D. Landhuis, T. Lange-Dei, B. Langley, K.-M. Lau, J. Ledford, K. Lee, B. Lester, L. Le Guevel, W. Li, A. T. Lill, W. Livingston, A. Locharla, D. Lundahl, A. Lunt, S. Madhuk, A. Maloney, S. Mandrà, L. Martin, O. Martin, C. Maxfield, J. McClean, M. McEwen, S. Meeks, A. Megrant, K. Miao, R. Molavi, S. Molina, S. Montazeri, R. Movassagh, C. Neill, M. Newman, A. Nguyen, M. Nguyen, C.-H. Ni, K. Ottosson, A. Pizzuto, R. Potter, O. Pritchard, C. Quintana, G. Ramachandran, M. Reagor, D. Rhodes, G. Roberts, K. Sankaragomathi, K. Satzinger, H. Schurkus, M. Shearn, A. Shorter, N. Shutty, V. Shvarts, V. Sivak, S. Small, W. C. Smith, S. Springer, G. Sterling, J. Suchard, A. Sztein, D. Thor, M. Torunbalci, A. Vaishnav, J. Vargas, S. Vdovichev,

G. Vidal, C. Vollgraff Heidweiller, S. Waltman, S. X. Wang, B. Ware, T. White, K. Wong, B. W. K. Woo, C. Xing, Z. Jamie Yao, P. Yeh, B. Ying, J. Yoo, N. Yosri, G. Young, A. Zalcman, Y. Zhang, N. Zhu, N. Zobrist, S. Boixo, J. Kelly, E. Lucero, Y. Chen, V. Smelyanskiy, H. Neven, A. Gammon-Smith, F. Pollmann, M. Knap, and P. Roushan. Visualizing dynamics of charges and strings in (2 + 1)D lattice gauge theories. *Nature*, 642:315–320, 2025.

[18] NVIDIA Corporation. Cuda-q documentation. https://nvidia.github.io/cuda-quantum/latest/. Accessed: 2025-12-05.

[19] Andrew Cross, Ali Javadi-Abhari, Thomas Alexander, Niel De Beaudrap, Lev S. Bishop, Steven Heidel, Colm A. Ryan, Prasahnt Sivarajah, John Smolin, Jay M. Gambetta, and Blake R. Johnson. Openqasm 3: A broader and deeper quantum assembly language. *ACM Transactions on Quantum Computing*, 3(3), September 2022.

[20] Siddharth Dangwal, Gokul Subramanian Ravi, Poulami Das, Kaitlin N. Smith, Jonathan M. Baker, and Frederic T. Chong. Varsaw: Application-tailored measurement error mitigation for variational quantum algorithms, 2023.

[21] Poulami Das, Swamit Tannu, Siddharth Dangwal, and Moinuddin Qureshi. Adapt: Mitigating idling errors in qubits via adaptive dynamical decoupling. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '21, page 950–962, New York, NY, USA, 2021. Association for Computing Machinery.

[22] Poulami Das, Swamit Tannu, and Moinuddin Qureshi. Jigsaw: Boosting fidelity of nisq programs via measurement subsetting. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '21, page 937–949, New York, NY, USA, 2021. Association for Computing Machinery.

[23] Chunhua Deng, Fangxuan Sun, Xuehai Qian, Jun Lin, Zhongfeng Wang, and Bo Yuan. Tie: energy-efficient tensor train-based inference engine for deep neural network. In *Proceedings of the 46th International Symposium on Computer Architecture*, ISCA '19, page 264–278, New York, NY, USA, 2019. Association for Computing Machinery.

[24] Adhitha Dias, Kirshanthan Sundararajah, Charitha Saumya, and Milind Kulkarni. Sparselnr: accelerating sparse tensor computations using loop nest restructuring. In *Proceedings of the 36th ACM International Conference on Supercomputing*, ICS '22, New York, NY, USA, 2022. Association for Computing Machinery.

[25] W. Dür, G. Vidal, and J. I. Cirac. Three qubits can be entangled in two inequivalent ways. *Phys. Rev. A*, 62:062314, Nov 2000.

[26] Suguru Endo, Simon C. Benjamin, and Ying Li. Practical quantum error mitigation for near-future applications. *Phys. Rev. X*, 8:031027, Jul 2018.

[27] Edward Farhi, Jeffrey Goldstone, and Sam Gutmann. A quantum approximate optimization algorithm. *arXiv preprint arXiv:1411.4028*, 2014.

[28] Benedikt Fauseweh. Quantum many-body simulations on digital quantum computers: State-of-the-art and future challenges. *Nature Communications*, 15(1):2123, 2024.

[29] Matthew Fishman, Steven R White, and Edwin Miles Stoudenmire. The ITensor software library for tensor network calculations. *SciPost Physics Codebases*, page 004, 2022.

[30] Jay M Gambetta, Jerry M Chow, and Matthias Steffen. Building logical qubits in a superconducting quantum computing system. *npj quantum information*, 3(1):2, 2017.

[31] Hong Gao, Satoshi Imamura, Akihiko Kasagi, and Eiji Yoshida. Distributed implementation of full configuration interaction for one trillion determinants. *Journal of Chemical Theory and Computation*, 20(3):1185–1192, 2024.

[32] Mahdi Ghorbani, Emilien Bauer, Tobias Grosser, and Amir Shaikhha. Compressed and parallelized structured tensor algebra. *Proc. ACM Program. Lang.*, 9(OOPSLA1), April 2025.

[33] Emmanouil Giortamis, Francisco Romão, Nathaniel Tornow, and Pramod Bhatotia. Qos: quantum operating system. In *Proceedings of the 19th USENIX Conference on Operating Systems Design and Implementation*, OSDI '25, USA, 2025. USENIX Association.

[34] Emmanouil Giortamis, Francisco Romao, Nathaniel Tornow, Dmitry Lugovoy, and Pramod Bhatotia. Qonductor: A cloud orchestrator for quantum computing. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '25, page 728–745, New York, NY, USA, 2025. Association for Computing Machinery.

[35] Johnnie Gray and Stefanos Kourtis. Hyper-optimized tensor network contraction. *Quantum*, 5:410, 2021.

[36] Lov K Grover. A fast quantum mechanical algorithm for database search. In *Proceedings of the twenty-eighth annual ACM symposium on Theory of computing*, pages 212–219, 1996.

[37] Flavien Gyger, Maximilian Ammenwerth, Renhao Tao, Hendrik Timme, Stepan Snigirev, Immanuel Bloch, and Johannes Zeiher. Continuous operation of large-scale atom arrays in optical lattices. *Phys. Rev. Res.*, 6:033104, 2024.

[38] David Hanneke, JP Home, John D Jost, Jason M Amini, Dietrich Leibfried, and David J Wineland. Realization of a programmable two-qubit quantum processor. *Nature Physics*, 6(1):13–16, 2010.

[39] Charles R Harris, K Jarrod Millman, Stéfan J van der Walt, Ralf Gommers, Pauli Virtanen, David Cournapeau, Eric Wieser, Julian Taylor, Sebastian Berg, Nathaniel J Smith, et al. Array programming with NumPy. *Nature*, 585(7825):357–362, 2020.

[40] Vojtěch Havlíček, Antonio D Córcoles, Kristan Temme, Aram W Harrow, Abhinav Kandala, Jerry M Chow, and Jay M Gambetta. Supervised learning with quantum-enhanced feature spaces. *Nature*, 567(7747):209–212, 2019.

[41] Torsten Hoefler, Thomas Häner, and Matthias Troyer. Disentangling hype from practicality: On realistically achieving quantum advantage. *Communications of the ACM*, 66(5):82–87, 2023.

[42] Olivia Hsu, Maxwell Strange, Ritvik Sharma, Jaeyeon Won, Kunle Olukotun, Joel S. Emer, Mark A. Horowitz, and Fredrik Kjølstad. The sparse abstract machine. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, ASPLOS 2023, page 710–726, New York, NY, USA, 2023. Association for Computing Machinery.

[43] Zhihao Jia, Oded Padon, James Thomas, Todd Warszawski, Matei Zaharia, and Alex Aiken. Taso: optimizing deep learning computation with automatic generation of graph substitutions. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP '19, page 47–62, New York, NY, USA, 2019. Association for Computing Machinery.

[44] Yimin Jiang, Yibo Zhu, Chang Lan, Bairen Yi, Yong Cui, and Chuanxiong Guo. A unified architecture for accelerating distributed DNN training in heterogeneous GPU/CPU clusters. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 463–479. USENIX Association, November 2020.

[45] Norman P. Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, Rick Boyle, Pierre-luc Cantin, Clifford Chao, Chris Clark, Jeremy Coriell, Mike Daley, Matt Dau, Jeffrey Dean, Ben Gelb, Tara Vazir Ghaemmaghami, Rajendra Gottipati, William Gulland, Robert Hagmann, C. Richard Ho, Doug Hogberg, John Hu, Robert Hundt, Dan Hurt, Julian Ibarz, Aaron Jaffey, Alek Jaworski, Alexander Kaplan, Harshit Khaitan, Daniel Killebrew, Andy Koch, Naveen Kumar, Steve Lacy, James Laudon, James Law, Diemthu Le, Chris Leary, Zhuyuan Liu, Kyle Lucke, Alan Lundin, Gordon MacKean, Adriana Maggiore, Maire Mahony, Kieran Miller, Rahul Nagarajan, Ravi Narayanaswami, Ray Ni, Kathy Nix, Thomas Norrie, Mark Omernick, Narayana Penukonda, Andy Phelps, Jonathan Ross, Matt Ross, Amir Salek, Emad Samadiani, Chris Severn, Gregory Sizikov, Matthew Snelham, Jed Souter, Dan Steinberg, Andy Swing, Mercedes Tan, Gregory Thorson, Bo Tian, Horia Toma, Erick Tuttle, Vijay Vasudevan, Richard Walter, Walter Wang, Eric Wilcox, and Doe Hyun Yoon. In-datacenter performance analysis of a tensor processing unit. *SIGARCH Comput. Archit. News*, 45(2):1–12, June 2017.

[46] Jinsung Kim, Aravind Sukumaran-Rajam, Vineeth Thumma, Sriram Krishnamoorthy, Ajay Panyala, Louis-Noël Pouchet, Atanas Rountev, and P. Sadayappan. A code generator for high-performance tensor contractions on gpus. In *2019 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 85–95, Feb 2019.

[47] Fredrik Kjolstad, Shoaib Kamil, Stephen Chou, David Lugato, and Saman Amarasinghe. The tensor algebra compiler. *Proc. ACM Program. Lang.*, 1(OOPSLA), October 2017.

[48] Scott Kovach, Praneeth Kolichala, Tiancheng Gu, and Fredrik Kjolstad. Indexed streams: A formal intermediate representation for fused contraction programs. *Proc. ACM Program. Lang.*, 7(PLDI), June 2023.

[49] Ryan LaRose, Andrea Mari, Sarah Kaiser, Peter J. Karalekas, Andre A. Alves, Piotr Czarnik, Mohamed El Mandouh, Max H. Gordon, Yousef Hindy, Aaron Robertson, Purva Thakre, Misty Wahl, Danny Samuel, Rahul Mistri, Maxime Tremblay, Nick Gardner, Nathaniel T. Stemen, Nathan Shammah, and William J. Zeng. Mitiq: A software package for error mitigation on noisy quantum computers. *Quantum*, 6:774, Aug 2022.

[50] Ying Li and Simon C. Benjamin. Efficient variational quantum simulator incorporating active error minimization. *Physical Review X*, 7(2), June 2017.

[51] Changzhou Long, Meng Huang, Xiucai Ye, Yasunori Futamura, and Tetsuya Sakurai. Hybrid quantum-classical-quantum convolutional neural networks. *Scientific Reports*, 15(1):31780, 2025.

[52] Igor L. Markov and Yaoyun Shi. Simulating quantum computation by contracting tensor networks. *SIAM Journal on Computing*, 38(3):963–981, 2008.

[53] Sam McArdle, Suguru Endo, Alán Aspuru-Guzik, Simon C. Benjamin, and Xiao Yuan. Quantum computational chemistry. *Rev. Mod. Phys.*, 92:015003, Mar 2020.

[54] Jarrod R McClean, Jonathan Romero, Ryan Babbush, and Alán Aspuru-Guzik. The theory of variational hybrid quantum-classical algorithms. *New Journal of Physics*, 18(2):023023, feb 2016.

[55] Supun Nakandala, Karla Saur, Gyeong-In Yu, Konstantinos Karanasos, Carlo Curino, Markus Weimer, and Matteo Interlandi. A tensor compiler for unified machine learning prediction serving. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 899–917. USENIX Association, November 2020.

[56] Thien Nguyen, Dmitry Lyakh, Eugene Dumitrescu, David Clark, Jeff Larkin, and Alexander McCaskey. Tensor network quantum virtual machine for simulating quantum circuits at exascale. *ACM Transactions on Quantum Computing*, 4(1), October 2022.

[57] Michael A Nielsen and Isaac L Chuang. *Quantum computation and quantum information*. Cambridge university press, 2010.

[58] NVIDIA. *cuTensorNet: A High-Performance Library for Tensor Network Computations*, 2024. Accessed: 2024-10-03.

[59] NVIDIA Corporation. NVIDIA CUDA-Q. https://developer.nvidia.com/cuda-q, 2025. Accessed: 2025-11-26.

[60] Román Orús. A practical introduction to tensor networks: Matrix product states and projected entangled pair states. *Annals of physics*, 349:117–158, 2014.

[61] Román Orús. A practical introduction to tensor networks: Matrix product states and projected entangled pair states. *Annals of Physics*, 349:117–158, 2014.

[62] Feng Pan, Hanfeng Gu, Lvlin Kuang, Bing Liu, and Pan Zhang. Efficient quantum circuit simulation by tensor network methods on modern gpus. *ACM Transactions on Quantum Computing*, 5(4), November 2024.

[63] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. Pytorch: An imperative style, high-performance deep learning library. *Advances in neural information processing systems*, 32, 2019.

[64] Tirthak Patel, Abhay Potharaju, Baolin Li, Rohan Basu Roy, and Devesh Tiwari. Experimental evaluation of nisq quantum computers: Error measurement, characterization, and implications. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–15, Nov 2020.

[65] Tirthak Patel and Devesh Tiwari. Qraft: Reverse your quantum circuit and know the correct program output. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '21, page 443–455, New York, NY, USA, 2021. Association for Computing Machinery.

[66] Alberto Peruzzo, Jarrod McClean, Peter Shadbolt, Man-Hong Yung, Xiao-Qi Zhou, Peter J Love, Alán Aspuru-Guzik, and Jeremy L O'brien. A variational eigenvalue solver on a photonic quantum processor. *Nature communications*, 5(1):4213, 2014.

[67] John Preskill. Quantum Computing in the NISQ era and beyond. *Quantum*, 2:79, August 2018.

[68] Andrew Putnam, Adrian M. Caulfield, Eric S. Chung, Derek Chiou, Kypros Constantinides, John Demme, Hadi Esmaeilzadeh, Jeremy Fowers, Gopi Prashanth Gopal, Jan Gray, Michael Haselman, Scott Hauck, Stephen Heil, Amir Hormati, Joo-Young Kim, Sitaram Lanka, James Larus, Eric Peterson, Simon Pope, Aaron Smith, Jason Thong, Phillip Yi Xiao, and Doug Burger. A reconfigurable fabric for accelerating large-scale datacenter services. In *Proceeding of the 41st Annual International Symposium on Computer Architecuture*, ISCA '14, page 13–24. IEEE Press, 2014.

[69] Qiskit Community. Quantum neural networks. https://qiskit-community.github.io/qiskit-machine-learning/tutorials/01_neural_networks.html, 2024. Accessed: 2025-12-01.

[70] Qiskit Community. Vqe efficient su2 ansatz. https://quantum.cloud.ibm.com/docs/en/api/qiskit/qiskit.circuit.library.EfficientSU2, 2024. Accessed: 2025-12-01.

[71] Qiskit contributors. Qiskit: An open-source framework for quantum computing, 2023.

[72] Google AI Quantum, Collaborators*†, Frank Arute, Kunal Arya, Ryan Babbush, Dave Bacon, Joseph C. Bardin, Rami Barends, Sergio Boixo, Michael Broughton, Bob B. Buckley, David A. Buell, Brian Burkett, Nicholas Bushnell, Yu Chen, Zijun Chen, Benjamin Chiaro, Roberto Collins, William Courtney, Sean Demura, Andrew Dunsworth, Edward Farhi, Austin Fowler, Brooks Foxen, Craig Gidney, Marissa Giustina,

Rob Graff, Steve Habegger, Matthew P. Harrigan, Alan Ho, Sabrina Hong, Trent Huang, William J. Huggins, Lev Ioffe, Sergei V. Isakov, Evan Jeffrey, Zhang Jiang, Cody Jones, Dvir Kafri, Kostyantyn Kechedzhi, Julian Kelly, Seon Kim, Paul V. Klimov, Alexander Korotkov, Fedor Kostritsa, David Landhuis, Pavel Laptev, Mike Lindmark, Erik Lucero, Orion Martin, John M. Martinis, Jarrod R. McClean, Matt McEwen, Anthony Megrant, Xiao Mi, Masoud Mohseni, Wojciech Mruczkiewicz, Josh Mutus, Ofer Naaman, Matthew Neeley, Charles Neill, Hartmut Neven, Murphy Yuezhen Niu, Thomas E. O'Brien, Eric Ostby, Andre Petukhov, Harald Putterman, Chris Quintana, Pedram Roushan, Nicholas C. Rubin, Daniel Sank, Kevin J. Satzinger, Vadim Smelyanskiy, Doug Strain, Kevin J. Sung, Marco Szalay, Tyler Y. Takeshita, Amit Vainsencher, Theodore White, Nathan Wiebe, Z. Jamie Yao, Ping Yeh, and Adam Zalcman. Hartree-fock on a superconducting qubit quantum computer. *Science*, 369(6507):1084–1089, 2020.

[73] Nils Quetschlich, Lukas Burgholzer, and Robert Wille. MQT Bench: Benchmarking Software and Design Automation Tools for Quantum Computing. *Quantum*, 7:1062, 2023. MQT Bench is available at https://mqt-bench.app/.

[74] Gokul Subramanian Ravi, Kaitlin N. Smith, Pranav Gokhale, and Frederic T. Chong. Quantum computing in the cloud: Analyzing job and machine characteristics. In *2021 IEEE International Symposium on Workload Characterization (IISWC)*, pages 39–50, 2021.

[75] Diego Ristè, Marcus P Da Silva, Colm A Ryan, Andrew W Cross, Antonio D Córcoles, John A Smolin, Jay M Gambetta, Jerry M Chow, and Blake R Johnson. Demonstration of quantum advantage in machine learning. *npj Quantum Information*, 3(1):16, 2017.

[76] Niall Robertson, Albert Akhriev, Jiri Vala, and Sergiy Zhuk. Approximate quantum compiling for quantum simulation: A tensor network based approach. *ACM Transactions on Quantum Computing*, 6(3), May 2025.

[77] M. Emre Sahin, Edoardo Altamura, Oscar Wallis, Stephen P. Wood, Anton Dekusar, Declan A. Millar, Takashi Imamichi, Atsushi Matsuo, and Stefano Mensa. Qiskit Machine Learning: an open-source library for quantum machine learning tasks at scale on quantum hardware and classical simulators. *arXiv e-prints*, page arXiv:2505.17756, May 2025.

[78] Raffaele Santagati, Alan Aspuru-Guzik, Ryan Babbush, Matthias Degroote, Leticia González, Elica Kyoseva, Nikolaj Moll, Markus Oppel, Robert M. Parrish, Nicholas C. Rubin, Michael Streif, Christofer S. Tautermann, Horst Weiss, Nathan Wiebe, and Clemens Utschig-Utschig. Drug design on quantum computers. *Nat. Phys.*, 20(4):549–557, 2024.

[79] Sebastian Schlag, Tobias Heuer, Lars Gottesbüren, Yaroslav Akhremtsev, Christian Schulz, and Peter Sanders. High-quality hypergraph partitioning. *ACM Journal of Experimental Algorithmics*, 27:1–39, 2023.

[80] Julian Schuhmacher, Marco Ballarin, Alberto Baiardi, Giuseppe Magnifico, Francesco Tacchino, Simone Montangero, and Ivano Tavernelli. Hybrid tree tensor networks for quantum simulation. *PRX Quantum*, 6:010320, Jan 2025.

[81] Peter W. Shor. Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer. *SIAM Review*, 41(2):303–332, 1999.

[82] Pedro F. Silvestre and Peter Pietzuch. Tempo: Compiled dynamic deep learning with symbolic dependence graphs. In *Proceedings of the ACM SIGOPS 31st Symposium on Operating Systems Principles*, SOSP '25, page 572–588, New York, NY, USA, 2025. Association for Computing Machinery.

[83] Cristian Tabares, Christian Kokail, Peter Zoller, Daniel González-Cuadra, and Alejandro González-Tudela. Programming optical-lattice Fermi-Hubbard quantum simulators. *PRX Quantum*, 6:030356, 2025.

[84] Wei Tang, Teague Tomesh, Martin Suchara, Jeffrey Larson, and Margaret Martonosi. Cutqc: using small quantum computers for large quantum circuit evaluations. In *Proceedings of the 26th ACM International conference on architectural support for programming languages and operating systems*, pages 473–486, 2021.

[85] Swamit S. Tannu and Moinuddin Qureshi. Ensemble of diverse mappings: Improving reliability of quantum computers by orchestrating dissimilar mistakes. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '52, page 253–265, New York, NY, USA, 2019. Association for Computing Machinery.

[86] Swamit S. Tannu and Moinuddin K. Qureshi. Mitigating measurement errors in quantum computers by exploiting state-dependent bias. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '52, page 279–290, New York, NY, USA, 2019. Association for Computing Machinery.

[87] Runzhou Tao, Yunong Shi, Jianan Yao, John Hui, Frederic T Chong, and Ronghui Gu. Gleipnir: Toward practical error analysis for quantum programs. In

Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation, pages 48–64, 2021.

[88] Runzhou Tao, Hongzheng Zhu, Jason Nieh, Jianan Yao, and Ronghui Gu. Quantum virtual machines. In *19th USENIX Symposium on Operating Systems Design and Implementation (OSDI 25)*, pages 411–428, 2025.

[89] Kristan Temme, Sergey Bravyi, and Jay M Gambetta. Error mitigation for short-depth quantum circuits. *Physical review letters*, 119(18):180509, 2017.

[90] Nathaniel Tornow, Emmanouil Giortamis, and Pramod Bhatotia. Qvm: Quantum gate virtualization machine. *Proc. ACM Program. Lang.*, 9(PLDI), June 2025.

[91] Frank Verstraete, Valentin Murg, and J Ignacio Cirac. Matrix product states, projected entangled pair states, and variational renormalization group methods for quantum spin systems. *Advances in Physics*, 57(2):143–224, 2008.

[92] Guifre Vidal. Efficient classical simulation of slightly entangled quantum computations. *Physical Review Letters*, 91(14):147902, October 2003. arXiv:quant-ph/0301063.

[93] Marcel Wagenländer, Guo Li, Bo Zhao, Luo Mai, and Peter Pietzuch. Tenplex: Dynamic parallelism for deep learning using parallelizable tensor collections. In *Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles*, SOSP '24, page 195–210, New York, NY, USA, 2024. Association for Computing Machinery.

[94] Rohan Yadav, Alex Aiken, and Fredrik Kjolstad. Spdistal: compiling distributed sparse tensor computations. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '22. IEEE Press, 2022.

[95] Xiao Yuan, Jinzhao Sun, Junyu Liu, Qi Zhao, and You Zhou. Quantum simulation with hybrid tensor networks. *Physical Review Letters*, 127(4):040501, 2021.

[96] Chen Zhang, Zeyu Song, Haojie Wang, Kaiyuan Rong, and Jidong Zhai. Hyquas: hybrid partitioner based quantum circuit simulation system on gpu. In *Proceedings of the 35th ACM International Conference on Supercomputing*, ICS '21, page 443–454, New York, NY, USA, 2021. Association for Computing Machinery.

[97] Chen Zhang, Haojie Wang, Zixuan Ma, Lei Xie, Zeyu Song, and Jidong Zhai. Uniq: A unified programming model for efficient quantum circuit simulation. In *SC22: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–16, Nov 2022.

[98] Shuaining Zhang, Yao Lu, Kuan Zhang, Wentao Chen, Ying Li, Jing-Ning Zhang, and Kihwan Kim. Error-mitigated quantum gates exceeding physical fidelities in a trapped-ion system. *Nature communications*, 11(1):587, 2020.

[99] Qi Zhao, You Zhou, and Andrew M Childs. Entanglement accelerates quantum simulation. *Nature Physics*, 21(8):1338–1345, 2025.

[100] Lianmin Zheng, Chengfan Jia, Minmin Sun, Zhao Wu, Cody Hao Yu, Ameer Haj-Ali, Yida Wang, Jun Yang, Danyang Zhuo, Koushik Sen, Joseph E. Gonzalez, and Ion Stoica. Ansor: Generating High-Performance tensor programs for deep learning. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 863–879. USENIX Association, November 2020.

[101] Ningxin Zheng, Bin Lin, Quanlu Zhang, Lingxiao Ma, Yuqing Yang, Fan Yang, Yang Wang, Mao Yang, and Lidong Zhou. SparTA: Deep-Learning model sparsity via Tensor-with-Sparsity-Attribute. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 213–232, Carlsbad, CA, July 2022. USENIX Association.

[102] Hongyu Zhu, Ruofan Wu, Yijia Diao, Shanbin Ke, Haoyu Li, Chen Zhang, Jilong Xue, Lingxiao Ma, Yuqing Xia, Wei Cui, Fan Yang, Mao Yang, Lidong Zhou, Asaf Cidon, and Gennady Pekhimenko. ROLLER: Fast and efficient tensor compilation for deep learning. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 233–248, Carlsbad, CA, July 2022. USENIX Association.

A Artifact Appendix

Abstract

The QTPU artifact contains the source code of the QTPU compiler and runtime, the benchmarking scripts used to produce every figure and table in the evaluation (Sec. 8), and the pre-collected benchmark logs from which all plots are regenerated. The artifact ships as a Python package (Python 3.11–3.13) with a uv-managed lock file and an optional GPU-enabled Docker image. All figures and tables can be regenerated from the included logs in seconds; all benchmarks except those running on real IBM Quantum hardware can be re-executed end-to-end on a single workstation.

Scope

The artifact validates the central performance and scalability claims of the paper:

- Pareto-frontier exploration of the compiler vs. the fixed QAC baseline (Fig. 9, Fig. 10, Table 1).

- Hardware validation on IBM Marrakesh and Pauli-noise simulation up to 150 qubits (Fig. 11).
- Runtime scaling and multi-QPU speedup, including comparison against `cuTensorNet` on a 100-qubit Dist-VQE workload (Fig. 12).
- Scalable hybrid computing on a 10-qubit QPU (Fig. 13).
- Hybrid ML: compilation time and code-size reductions vs. batched execution (Fig. 14).
- Quantum error mitigation vs. Mitiq (Fig. 15).
- End-to-end composability (Fig. 16).

The artifact can additionally be used as a general-purpose library for expressing and executing hybrid tensor networks on user-defined circuits.

Contents

The artifact is organized as follows:

- `src/qtpu/` — the core library (compiler, runtime, transforms).
- `evaluation/` — a `run.py` and `plot.py` per evaluation section (`compiler/`, `runtime/`, `hardware/`, `use_cases/{scale,hybrid_ml,error_mitigation,end_to_end}/`).
- `logs/` — pre-collected benchmark results in JSONL. Every line is a self-contained JSON record with parameters and measurements.
- `examples/` — standalone usage examples.
- `REPRODUCE.md` — step-by-step instructions covering every figure and benchmark.
- `Dockerfile`, `pyproject.toml`, `uv.lock` — reproducible environment specifications.

Hosting

The artifact is hosted on GitHub at https://github.com/nathanieltornow/qtpu (branch `main`). The exact version evaluated by the AEC is tagged `osdi26-ae`. An archival snapshot with a citable DOI is mirrored on Zenodo.

Requirements

Software. Python 3.11–3.13 and uv (recommended) or `pip`. The full dependency set is pinned in `uv.lock` and installable via `uv sync`. A GPU-enabled Docker image is provided for `cuTensorNet` comparisons (Fig. 12c).

Hardware. All non-hardware experiments run on a commodity x86_64 Linux or macOS workstation; 16 GB of RAM is sufficient for the log-based plot regeneration, and 64 GB is recommended to re-run the larger simulations from scratch. An NVIDIA GPU with a recent CUDA toolchain is required only to reproduce the `cuTensorNet` comparison in Fig. 12c.

IBM Quantum access. Fig. 11a (IBM Marrakesh) requires an IBM Quantum account with access to the `ibm_marrakesh` backend; the credentials are supplied via the `IBM_TOKEN` and `IBM_CRN` environment variables. The pre-collected hardware logs are included so that Fig. 11a can be regenerated without IBM credentials.

Reproducing the Results

After `uv sync`, every figure can be regenerated from the included logs in seconds, e.g.:

```
uv run python -m evaluation.compiler.plot
uv run python -m evaluation.hardware.plot_qnn
uv run python -m evaluation.runtime.plot
uv run python -m evaluation.use_cases.scale.plot
uv run python -m evaluation.use_cases.hybrid_ml.plot
uv run python -m evaluation.use_cases.error_mitigation.plot
uv run python -m evaluation.use_cases.end_to_end.plot
```

Each section additionally exposes a `run.py` that regenerates the underlying logs from scratch; expected wall-clock times range from ~5 min (scale, error mitigation) to ~30 min (runtime, end-to-end). The Pauli-noise sweep to 150 qubits takes ~20 min and the 80-qubit Pareto sweep ~60 min on a single workstation. Full instructions, including the IBM-hardware path, are documented in `REPRODUCE.md`.

Acumen: A Platform for Encrypted and Accountable Collaborative Editing

Ryan Cottone[1], Darya Kaviani[2], Conor Power[2], Will Giorza[2], Evelyn Koo[2],
Natacha Crooks[2], Raluca Ada Popa[2]
[1]*Stanford University,* [2]*University of California, Berkeley*

Abstract

Modern-day collaborative editing tools must reconcile a prominent tension between user privacy and collaboration: encrypting user data prevents an application server from processing user edits. We present Acumen, a cryptographic system for real-time collaborative applications based on *conflict-free replicated data types* (CRDTs). Acumen is the first system to provide *strong snapshot consistency*, enabling untrusted users to create verifiable document snapshots used to invite new collaborators. Acumen also provides confidentiality, integrity, fork-causal consistency, and ensures that invited users do not learn the previous edit history of the document.

We achieve these properties through the use of cryptographic accumulators and a novel secure garbage collection mechanism. Our evaluation shows that Acumen can support 25 users each simultaneously typing 60 WPM with negligible degradation in latency and availability.

1 Introduction

There has been a recent shift towards real-time, cloud-based collaborative applications such as Notion, Google Docs, and Microsoft Office. These collaborative editors generally follow one of two separate approaches: operational transforms (OTs) [11, 13] or conflict-free replicated data types (CRDTs) [22, 30, 32, 36]. While convenient for users, this shift to the cloud naturally raises privacy concerns.

Although there has been considerable work on end-to-end encrypted file storage systems (e.g. SUNDR [26] and Depot [28]), these solutions are unsuitable to collaborative editing applications where users simultaneously edit the same files and must resolve concurrent user edits. Indeed, it is not immediately obvious how an application provider who cannot observe the contents of its users' behavior can consolidate, manage, and edit their data.

A recent line of *secure collaborative editors* [13, 21, 24, 25] aims to resolve this tension. In this paper, we focus on *decentralized* secure collaborative editors in which users maintain local document state and directly broadcast encrypted edits to each other. This model provides confidentiality against eavesdropping servers, but introduces another attack vector: *user adversaries* may attempt to disrupt the consistency of other users, such that users disagree on the correct edit history. This can be especially problematic if malicious users collude with a malicious network adversary, who can arbitrarily block or reorder network messages to and from users.

Our goal is to construct a secure collaborative editor that ensures confidentiality against unauthorized eavesdroppers and integrity against malicious users. We identify four key requirements that secure decentralized collaborative editors should provide:

1. Confidentiality. Passive eavesdroppers should learn nothing about the document beyond what can be inferred from network traffic. This includes document access patterns (e.g. that User A executed an insert at position k followed by a deletion).

2. Integrity. Honest users should maintain a consistent local document history comprised of authentic edits (e.g. no forking writes). In our decentralized setting, this corresponds to the notion of *fork-causal consistency* [28].

3. Performance. Performance should remain comparable to plaintext editors: users should process operations in real-time, encrypted edits should remain relatively small in size, and user storage should scale linearly with the current document size (rather than the overall edit history length).

4. Secure Dynamic Membership. Users should be able to add new collaborators into existing documents. The new user should not receive data beyond what currently exists in the document (*edit-history privacy*) and should maintain the same integrity guarantees as existing users (*snapshot consistency*).

Secure dynamic membership is particularly difficult to provide in decentralized editors due to the tension between snapshot consistency and edit-history privacy: *how can a new user verify their given snapshot is consistent without access to the corresponding history?*

Existing secure decentralized editors fail to provide all of these guarantees. For instance, the state-of-the-art system Snapdoc [24] leaks user access patterns (violating confidentiality), provides a weak notion of snapshot consistency (violating secure dynamic membership), and has performance scaling with the overall edit history length.

Acumen is the first secure collaborative editor to provide confidentiality, integrity, and secure dynamic membership against malicious users and network adversaries. We also provide real-time performance: Acumen is able to sustain 25 users each typing at 60 WPM, orders of magnitude greater than the state-of-the-art. Acumen's performance and guarantees hinge on two key ideas:

1. Cryptographic accumulators for snapshot consistency & hiding access patterns. At the core of our verifiable snapshot functionality are *cryptographic accumulators* [4, 6, 7]. To verify snapshot consistency, newly-invited users asynchronously compare their untrusted snapshot state against the signed, constant-space accumulators representing existing user states. Our key insight is to expand the use of accumulators to capture both the *CRDT's internal structure* and overall set of document edits. This approach eliminates the variable-length edit data in the state-of-the-art, preventing edit pattern leakage and bandwidth blow-up.

2. Secure garbage collection. We aim for our system to scale with the current document size rather than edit history length. To this end, we utilize garbage collection techniques to prune outdated components of the document data structure without compromising any of Acumen's guarantees. Garbage collection, however, is made more complicated by our snapshot consistency guarantee: each snapshot must have data sufficient to reconstruct every other user's state, such that data can only be deleted once every user has deleted it. We introduce a novel *second-order* asynchronous garbage collection protocol that eliminates this circular dependency.

In summary, this paper makes three contributions:

- We introduce a novel protocol to provide strong snapshot consistency with real-time performance.
- We identify and resolve access-pattern leakage in prior work, simultaneously reducing bandwidth costs by orders of magnitude.
- We introduce a secure garbage collection mechanism, eliminating runaway storage costs and providing competitive performance against standard CRDTs.

2 Background

We first provide the necessary background on collaborative text editors and CRDTs.

2.1 Collaborative Editing Data Structures

Conflict-free Replicated Data Types (CRDTs) [36] are a recent approach to asynchronous, distributed data structures. Acumen employs **operation-based CRDTs** in which users broadcast document edits that other users process locally. Under the guarantee that operations are reliably delivered in causal order, op-based CRDTs satisfy *strong eventual consistency* (users processing concurrent operations in different orders still arrive at the same state).

Treedoc Treedoc [33] is a popular *list CRDT*, often used to implement shared documents represented as character arrays. Treedoc models characters as individual nodes within a tree structure, with the document string derived as the tree's in-order traversal. Each node has an associated *disambiguator*, composed of the User ID and a user-specific counter (precisely the user's version vector counter). The disambiguator forms a unique tuple, which is used for node addressing and as a sorting key for deterministic ordering between sibling nodes. Finally, each Treedoc node contains a `data` field (comprised of a single character) and associated metadata.

Tombstone Nodes In Treedoc, deletion operations mark a corresponding insertion as a *tombstone*. This removes the corresponding node's `data` field, but does not remove the node itself. Tombstone nodes remain within the structure of the overall Treedoc but do not contribute to the final document string. These nodes are generally required to maintain the structure of the tree and are retained until safely garbage collected by our scheme (§5.3). Preserving security with garbage collection is one of the key challenges that Acumen must address.

We choose Treedoc for its useful causal guarantees, potential for fixed-length identifiers, concrete efficiency, and general simplicity. Our protocol could likely be generalized to use any op-based CRDT with tombstoning.

2.2 Fork-Causal Consistency (FCC)

While op-based CRDTs ensure that client states eventually converge given the same operations, they do not provide protection against malicious parties. In Acumen, we enhance CRDTs with fork-causal consistency (FCC) [28]. At a high level, FCC guarantees that honest users agree on the document state at all points in their shared (partially-ordered) edit history.

More precisely, each Acumen document instance implicitly maintains an **execution** E comprised of directed edges between operations and their causal dependencies. Each user u has an associated *local execution* $E_u \subseteq E$ comprised of operations created or processed by u. For our purposes, an execution E (informally) satisfies FCC if three conditions hold:

1. Operations by honest users are totally ordered in E.
2. Honest user executions only see totally-ordered operations from individual users (i.e. do not process forking operations from the same user).
3. All pairs of honest users a,b agree on the document state at each point in their shared history $E_a \cap E_b$.

We formally define FCC in Appendix B. Another key implication of FCC is that users are prohibited from processing operations belonging to conflicting *forks* in the causal graph. This occurs precisely when an adversarial user equivocates two operations o,o' to two distinct subsets of users, effectively branching the edit history. Users who process o will reject operations from those who processed o', and vice versa, forming a permanent fork in the execution graph.

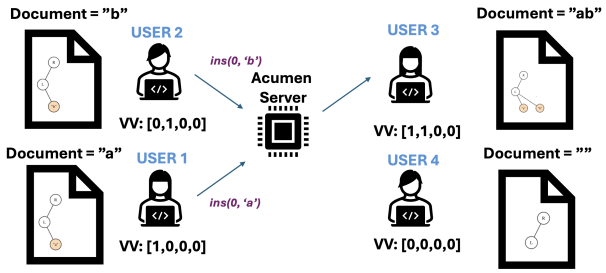


Figure 1: An Acumen setup with four users editing a shared document. User 1 inserts an "a" concurrently with User 2 inserting a "b". User 3 hears about both before User 4.

2.3 Merkle-Tree Accumulator

Cryptographic accumulators [4, 6, 7] enable the efficient representation of unordered sets in constant space. We adopt the *Merkle-tree accumulator* [7] which uses a cryptographic hash function to provide three key properties: constant-space representation, set collision resistance, and update efficiency.

- **Constant-space representation**: The size of acc is independent of the set cardinality $|S|$.
- **Set-collision resistance**: $acc_A = acc_B \iff A = B$ with overwhelming probability.
- **Update efficiency**: There exists a function `update` that can add or remove k elements from acc in time independent of the current set cardinality $|S|$.

Note that the accumulator can store local data to support efficient updates, as long as the final representation is constant size.

We briefly describe the Merkle-tree accumulator construction here. The accumulator state consists of a sparse binary Merkle tree of fixed height h, typically chosen to be the output length of the underlying cryptographic hash. Nodes are stored dynamically in a hash map, with non-existent nodes given a default value. Set elements are accumulated by inserting them into the leaf node determined via cryptographic hash. The root hash of the Merkle tree serves as our constant-space representation.

Following the standard Merkle tree construction, recomputing the root hash after modifying a leaf node takes $O(h)$ time. We can batch k edits in $O(h+k)$ time by using one recomputation pass after editing all k leaves.

3 System Overview

An Acumen deployment consists of n clients and a relay server, which is responsible for relaying all communication between clients as seen in Figure 1. Clients send their updates to the relay, which broadcasts them to the document's collaborators. We note that Acumen can function over peer-to-peer networks—a central server is useful for message availability but has no impact on our protocol guarantees.

3.1 Threat Model

Acumen assumes a threat model in which the relay server is actively malicious (functioning as the “network adversary”). Up to $n-1$ clients are compromised and deviate arbitrarily from the protocol. For example, malicious clients may engage in standard editing activities, such as inserting or deleting characters, but could also attempt more sophisticated attacks, such as sending inconsistent edits for the same document position to different clients.

We assume the presence of a trusted public-key infrastructure and a secure group messaging protocol [8, 9, 37]. Our protocol can be viewed as an additional layer over a secure group messaging platform, focused on enforcing document-level consistency.

Non-goals. Acumen does not offer protection against denial of service (DoS) attacks. That is, the server or ISP may prevent users from receiving or sending updates. Acumen does not defend against timing attacks that leverage knowledge of when a user makes an edit.

3.2 Security Guarantees

We now describe Acumen's security guarantees, which we bucket into three overarching categories: Confidentiality, Integrity, and Secure Dynamic Membership. Formal proofs can be found in our supplemental material. At a high level, *confidentiality* safeguards document contents, access patterns, and edit history from unauthorized access. *Integrity* guarantees a coherent document state across clients, preventing conflicts and maintaining a valid execution history even under adversarial conditions. *Secure Dynamic Membership* ensures that new users can be added to existing documents while maintaining the aforementioned integrity guarantees (snapshot consistency) and without revealing more than the current document state (edit-history privacy).

Confidentiality. Operations do not reveal information about document contents or access patterns to unauthorized parties.

Integrity. Acumen provides *fork-causal consistency* (FCC) [28]. Informally, FCC ensures that honest clients follow a causally consistent subgraph of a global execution graph. Forks can arise when malicious clients issue equivocating updates to different honest users. FCC ensures that honest users with divergent histories detect inconsistencies upon communicating. We give an informal definition of FCC in §2.2, and a formal definition in §B.

Secure Dynamic Membership. Acumen guarantees *strong snapshot consistency* (hereafter just snapshot consistency). Snapshot consistency ensures that new users can verify their document state is the product of a fork-causal consistent execution without requiring any communication with other honest users (formally stated in Definition 1). This definition is notably stronger than the one considered by prior work [24] (hence *strong* snapshot consistency). Informally, prior work allows adversaries to create snapshots that encode any *subset*

	§	Ours	Snapdoc	CRDT
Consistency (FCC)	2.2	✓	✓	X
Edit-Hist. Privacy	3.2	✓	✓	X
Snapshot Consistency	3.2	Strong	Weak	X
Access Patt. Hiding	5.1	✓	X	X
Local Storage	5.3	$O(D)$	$O(E+D)$	$O(D)$
Edit Size (bal.)	5.1	$O(1)$	$O(\log E)$	$O(1)$
Edit Size (WC)	5.1	$O(1)$	$O(E)$	$O(1)$
Snapshot Size (bal.)	5.3	$O(D)$	$O(D+\log E)$	$O(D)$
Snapshot Size (WC)	5.3	$O(D)$	$O(E+D)$	$O(D)$

Table 1: System Feature Comparison. E = edit history length, D = document size, bal./WC = balanced / worst case edit pattern. We consider the number of users to be $O(1)$.

of the correct document state, rather than precise equality as in strong snapshot consistency. We provide a formal comparison in §D.

Acumen also provides *edit-history privacy*: newly invited clients cannot access characters deleted prior to the snapshot creation, unless they collude with another client who had document access before their invitation. For instance, consider adding a collaborator to an existing research paper document — one might prefer that the collaborator not see previous draft versions. Edit-history privacy does **not** hide the access patterns of the document before snapshot creation, as this is inevitably leaked by the structure of the Treedoc object (§2.1) used to represent the shared document. Leaking the Treedoc structure reveals information about the causal order of prior operations, but not their contents.

Edit-history privacy and snapshot consistency are challenging to provide simultaneously. The core challenge in the design of Acumen is verifying a given document snapshot is the product of a fork-causal consistent history without access to the history itself.

4 Strawman Protocol

We now present a starting point for the Acumen protocol, providing only fork-causal consistency and snapshot consistency. In §5, we introduce several novel techniques to enhance the privacy and efficiency of this protocol such that access patterns are hidden, new collaborators cannot access the past edit-history, and user states can be securely garbage-collected.

In our basic rendition of Acumen, after a user creates a document, it can add *collaborators*. Changes to a document or its membership are processed as *operations*. More precisely, we define three types of operations: `Insertion`, `Deletion`, `AddUser`. The former two behave as outlined in §2.1, and we detail the latter in §4.3. Each operation is encrypted and broadcast using the abstracted secure group messaging scheme (§3.1). Users maintain a set called `operations` containing the set of all operations sent or received by a user.

We leverage version vectors [10, 14, 29] to track causal history. Each document collaborator maintains a local version vector as a unique document timestamp. A version vector VV is a mapping of IDs to integers, where $VV[u]$ represents the total number of operations processed from ID u. An operation's version vector is a copy of its sender's, with the sender's index incremented by one to account for the operation's creation.

4.1 History Hash Chains

As outlined in §3.2, we aim to achieve *fork-causal consistency* (FCC). Following prior work [13, 21, 28] we enforce FCC using hash chains over edit histories, referred to here as *history hashes*. Unlike SPORC [13] which utilizes a central server to provide a total order over operations, our environment implies a partially ordered execution graph (e.g. a DAG rather than a single chain).

Acumen defines one hash chain per user, representing the overall DAG as a collection of individual user hash chains. For simplicity, we define an Acumen operation's *causal dependencies* as the most recent operation from each document user, as seen by the operation creator.

Let $HH_{u,n}$ denote the history hash for user u after n operations, $o_{u,n}$ denote user u's n-th operation, and $H(o)$ be the cryptographic hash of an operation (elaborated on in §4.1.1). We define $HH_{u,n} = H(H(o_{u,n}), HH_{u,n-1})$ with $HH_{u,0} = 0$. We refer to HH_u as user u's local history hashes.

4.1.1 Operation Hash Construction

Hashing an operation as a raw bytestring will pose issues for edit-history privacy in our full protocol, due to the resulting hash leaking information about the encoded character data (we elaborate upon this in §5.2). To avoid this, we define the Operation struct to include a random nonce (Table 3). Formally, let `rand` be a λ-bit random value generated by the operation sender (λ is our security parameter), `op.metadata` refer to non-data fields concatenated into one bytestring, and $\mathsf{PRF}: \{0,1\}^\lambda \times \{0,1\}^* \to \{0,1\}^*$ be a secure pseudorandom function [18]. Define $H(\texttt{op})$ as follows:

$$H(\texttt{op}) = H(\texttt{op.metadata}, \mathsf{PRF}(\texttt{op.rand}, \texttt{op.data}))$$

Denote the value of $\mathsf{PRF}(\texttt{op.rand},\texttt{op.data})$ as the *intermediate data hash*. Each operation/node now includes a new `hdata` field consisting of this intermediate hash. This construction is later motivated in our snapshot consistency protocol (§5.2)—looking ahead, we will want to reconstruct $H(\texttt{op})$ using only `op.metadata` and $\mathsf{PRF}(\texttt{op.rand},\texttt{op.data})$ such that `op.data` remains hidden given only these values.

Tombstones Recall that processing a deletion operation marks the targeted operation/node as a *tombstone* (§2.1). More precisely, marking an operation/node as a tombstone sets `data, rand = null` in the associated object. For Treedoc nodes, we also set `tombstone = True`. This has the effect of permanently hiding the original character data of the

Operation Verification

Input: Operation $o = (VV_{update}, HH_{update}, \ldots)$ from v.

Let u be the receiver with version vector VV_u and history hash map HH (recall this is a map over (id, ctr) unlike the map over id like HH_{update}).

1. Verify that we have processed all the causal dependencies of o.

$$\forall j \in VV_{update}, \begin{cases} VV_u[j] \geq VV_{update}[j] & j \neq v \\ VV_u[j] = VV_{update}[j] - 1 & j = v \end{cases}$$

2. Let $k = VV_{update}[v]$ be the operation sender's latest counter. Compute v's next history hash value:

$$HH_{v,k} = H(H(o), HH_{v,k-1})$$

3. Verify that HH_{update} is equal to our local history hashes at each point:

$$\forall j \in HH_{update}, HH_{j,VV_{update}[j]} = HH_{update}[j]$$

Figure 2: Operation Verification

operation/node, which will become relevant for edit-history privacy in our full protocol (§5.2).

4.2 Verifying Operations

Our operation verification subroutine is straightforward and will be used in both the starting-point protocol and full protocol. At a high level, a user accepts an incoming operation o only if the following conditions hold:

- The user has processed o's causal dependencies.
- o's history hash chains are consistent with the user's.

Satisfying these conditions implies the receiver and operation sender agree on the execution graph up to the operation's version vector, i.e. are pairwise fork-causally consistent. A full description of the protocol is given in Figure 2. Once verified, an operation is applied according to its type and associated Treedoc semantics.

4.3 Adding a Collaborator

To add another user as a document collaborator in our basic scheme, an existing user ("snapshot sender") simply sends a copy of the (signed) operation log to the new user and broadcasts an `AddUser` operation specifying the new user ID.

Since the snapshot sender's set of operations encodes a full causal execution graph, the new user can simply replay the entire causal history and apply the operation verification subroutine (Figure 2) at each step to verify fork-causal consistency. In §5, we will update this scheme to only send data proportional to the current document size rather than the edit history size.

Variable	Definition	§
u	User ID	3.1
O	Primary operation set	4
$O^{(2)}$	Secondary operation set	5.3
T	Primary Treedoc obj.	2.1
$T^{(2)}$	Secondary Treedoc obj.	5.3
HH	Current History Hashes	4.1
VV_u	Current Version Vector	4.2
VV_{min}	Minimum Version Vector	5.3.1

Table 2: An Acumen user state with relevant fields. Fields in blue are introduced by our garbage collection changes in §5.3.

Variable	OpType	Definition	§
`u`	*	Sender User ID	3
`VV`	*	Version Vector	4
$\texttt{VV}_{\texttt{min}}$	*	Pairwise Min. V.V.	5.3.1
`HH`	*	History Hashes	4.1
`acc`	*	Operation acc.	5.2.2
`tacc`	*	Treedoc acc.	5.2.2
`hsig`	*	Sig. over above	5.2.2
`optype`	*	OpType enum	4
`data`	Ins.	Character data	4
`rand`	Ins.	λ-bit random	5.2.3
`hdata`	Ins.	PRF(`rand`,`data`)	5.2.3
`t_disamb`	Del.	Target disamb	4
`new_uid`	Add.	New user ID	4.3

Table 3: An Operation object, with operation type-specific fields marked in the OpType column. An Acumen state descriptor is comprised of the first seven fields.

5 The Acumen Protocol

We now present the full Acumen protocol. In this section, we update our protocol to hide user access patterns and efficiently offer edit-history privacy while retaining our confidentiality and consistency guarantees. We then introduce secure garbage collection.

5.1 Hiding Access Patterns

In the Treedoc CRDT (§2.1, [33]), a node's full path to the root node is serialized as part of its representation. Treedoc paths are influenced by the document's access patterns—for instance, an edit trace of N insertions each inserting at the end of the current list results in a linear tree structure, such that the last insertion has a path length of N nodes. On the other hand, a "balanced" edit trace would have path lengths on the order of $\log N$. The sizes of encrypted edits leak information about the document access pattern to network adversaries.

Fixed-Length Paths. To mitigate this issue, we replace the full serialized path with a fixed-length path consisting only of the parent node's disambiguator. This is essentially the

Variable	Definition	§
`disamb`	Disambiguator	2.1
`parent`	Parent node disamb.	2.1
`left/right`	Child node disambs.	2.1
`dir`	Direction (left/right child)	2.1
`tombstone`	Tombstone marker	2.1
`data`	Node data	4.1.1
`rand`	Random nonce	4.1.1
`hdata`	PRF(`rand`,`data`)	4.1.1

Table 4: Updated Treedoc Node

"tree representation" of the original Treedoc paper [33], with the tweak that individual atoms (edits) are also sent in their reduced form. One can view this as the op-based version of the Treedoc CRDT.

This approach has the downside of requiring reliable causal broadcast (RCB)—informally, edits must be processed only after their causal dependencies are processed. Acumen already requires RCB for history-hash verification (§4.2), so this downside is of no extra consequence. A full description of our new Treedoc node struct can be found in Table 4.

Crucially, this approach removes the path-length side channel; timing leakage remains outside our threat model as discussed in Section 3.1. This contrasts with the state-of-the-art system Snapdoc [24], which includes the variable-length paths. While nominally straightforward, this change introduces significant complications when adding secure snapshots and garbage collection. We outline our contributions towards resolving these complications in Sections 5.2 and 5.3.

We note that Snapdoc's construction is incompatible with this change as their snapshot verification explicitly relies on each operation encoding a full Treedoc path from root to leaf.

5.2 Snapshot Consistency

Recall that Acumen provides fork-causal consistency (§2.2, §3.2), which informally guarantees that honest users always see a consistent subset of a global execution. *Snapshot consistency* is the extension of fork-causal consistency (which considers a static, existing group of users) to the setting in which new users are invited to existing documents. For instance, a malicious user may invite a new user and send a malformed copy of the current document. Informally, snapshot consistency states that new user states should be the product of a fork-causal consistent execution, as if an honest user processed the full edit history up to the same point.

We now formalize the notion of (strong) snapshot consistency. Define a function $\mathsf{exec}(S,E)$ that returns the resulting Acumen state (as defined in Table 2) from applying the given (partial) [1] execution E to state S according to the Acumen protocol. Further let $\emptyset_m$ define the "empty state" of user m with no operations applied.

[1] A partial execution is a connected subgraph of some complete execution.

Variable	Definition	§
O_S	Sender's Operation Set	4
T_S	Sender's Treedoc Object	2.1
VV_S	Sender's Version Vectors	4
HH_S	Sender's History Hashes	4.1
A_S	State Descriptors	5.2.2

Table 5: Acumen Snapshot Fields

Definition 1 (Strong Snapshot Consistency)**.** A snapshot S from user s deriving state $S_{snapshot}$ satisfies (strong) *snapshot consistency* if there exists a fork-causal consistent execution E such that $S_{snapshot} = \mathsf{exec}(\emptyset_s, E)$.

Intuitively, a user u instantiating from a snapshot sent by user s should realize the same state as an honest user who processed the "full" history up to the same point. The term $\mathsf{exec}(\emptyset_s, E)$ can be thought of as the result of simulating E as if s was honest (thus deriving the correct snapshot state).

Corollary 1. A snapshot S is consistent if the following two conditions hold:

- There exists some partial execution $E_{missing}$ such that $\mathsf{exec}(S_{honest}, E_{missing}) = S_{snapshot}$ for some honest user state S_{honest}.
- $E_{honest} \cup E_{missing}$ is a fork-causally consistent execution.

Proof Sketch: Intuitively, we show that an honest user derives $S_{snapshot}$ given the operations between their current state and $S_{snapshot}$. Since honest users agree on the state at shared execution nodes, the simulated honest snapshot sender would also derive $S_{snapshot}$ under the same execution. We conclude that the snapshot satisfies Definition 1. □

5.2.1 High-Level Construction

We now outline a high-level construction for the creation and verification of snapshots. Creation is straightforward: a snapshot is simply the copy of the snapshot creator's state with additional metadata (Table 5).

Upon receiving a snapshot, new users will run a *snapshot verification protocol* that outputs an Acumen state if and only if the given snapshot S satisfies Definition 1. Following Corollary 1, our key idea to verify a snapshot will be to reconstruct an honest user state S_u and show that $\mathsf{exec}(S_u, E_{S,u}) = S_{snapshot}$ where $E_{S,u}$ is the partial execution corresponding to $E_{missing}$ for user u. We will then verify that $E_u \cup E_{S,u}$ is fork-causal consistent (where $\mathsf{exec}(\emptyset_u, E_u) = S_u$). Since at least one user included in the snapshot is honest [2], it suffices to verify this condition over all users included in the snapshot.

High-Level Snapshot Verification

[2] This assumption is implicit for snapshot consistency, as documents consisting entirely of malicious users can trivially provide an arbitrary snapshot.

Execute the following for each user u contained in S:

1. Verifiably reconstruct the user state S_u using S.
2. Verify that the history hashes of each S_u are consistent with (i.e. prefixes of) the snapshot history hashes.
3. Identify operations in S that were not processed in S_u. Verify that the partial execution $E_{S,u}$ defined by these operations is well-formed (i.e. satisfies FCC constraints).
4. Apply these missing operations to S_u and verify the result equals $S_{snapshot}$.

We briefly argue that these steps satisfy Corollary 1, and by extension Definition 1. Step 4 ensures that $S_{snapshot} = \mathsf{exec}(S_u, E_{S,u})$ for each user u (and therefore for at least one honest user h). Steps 2-3 ensure that $E_{S,h} \cup E_h$ is a fork-causal consistent execution (if all user history hashes are prefixes of the same history hashes, they must be pairwise consistent).

Achieving Step 1 of the high-level construction while maintaining edit-history privacy is the primary obstacle to efficient snapshot consistency. Our main technical contributions relate to this step, which we now detail in full. Section 5.2.2 will describe the notion of *state descriptors* for verifiably reconstructing user states from untrusted snapshot data. Section 5.2.3 then reconciles our approach with edit-history privacy by introducing *placeholder nodes* and related changes.

5.2.2 State Verification From State Descriptors

Recall that *cryptographic accumulators* [4, 6, 7] (§2.3) represent a set of items in a constant-space value. Given $acc_S = \mathsf{accumulate}(S)$, one can cryptographically verify $S' = S$ for a local set S' by comparing $\mathsf{accumulate}(S') \stackrel{?}{=} acc_S$.

We use accumulators to represent an Acumen state as a constant-space hash value. Define the *operation set accumulator* (denoted *acc*) as the accumulator over (the hashes of) a user's set of operations. Likewise, define the *Treedoc accumulator* (denoted *tacc*) to be the accumulator over a user's set of Treedoc nodes.

Following prior work [24], we now define the notion of a **state descriptor**. User u's state descriptor $A_{S,u}$ is comprised of the first 7 fields from their latest operation (Table 3), including the version vector, history hashes, and operation set and Treedoc accumulators.

$$A_{S,u} = (u, acc_u, tacc_u, VV_u, VV_{u,min}, HH_u, hsig)$$

These fields combine to form a signed description of the user's state at the included version vector, and are included in each Acumen snapshot (Table 5). Using $A_{S,u}$, the snapshot verifier can verify that their reconstruction of S_u using untrusted snapshot data equals user u's true state by comparing the respective accumulators and other metadata.

5.2.3 Edit-History Privacy From Placeholders

Acumen also satisfies *edit-history privacy*, which informally requires the snapshot to not reveal previously-deleted document characters. It is not immediately clear how to reconcile this requirement with state descriptors—how can the verifier reconstruct a user state containing a now-deleted character if the snapshot is not allowed to include said character?

Edit-History Private Operations. This situation occurs when the snapshot creator has processed a deletion, but at least one other user has not. Let u be a user included in some snapshot S. For a given snapshot $S = (O_S, T_S, \ldots)$, define the set of *edit-history private operations with respect to u* as the set of insertions witnessed by u that have an associated deletion *not* yet witnessed by u (equivalently, the set of operations deleted in S but not yet deleted in S_u). Likewise define the set of edit-history private nodes with respect to u as the set of Treedoc tombstones with an associated edit-history private insertion.

Key Insight. Our key idea is that *the original data of edit-history private operations/nodes is irrelevant to the outcome of* exec. Intuitively, the original character data is moot if the operation/node will be deleted in the end state.

Consider an example where the snapshot S contains multiple edit-history private insertions $(o_1, \ldots, o_k)$ with respect to user u. Following the high-level construction, let S'_u be the state reconstructed in Step (1), such that S'_u equals S_u except that $(o_1, \ldots, o_k)$ and their associated nodes are tombstones. Assuming that the execution $E_{S,u}$ from Step (3) contains deletion operations targeting $(o_1, \ldots, o_k)$ (as is the case in correct snapshots), we still have $\mathsf{exec}(S'_u, E_{S,u}) = S_{snapshot}$.

We introduce **placeholder nodes** in lieu of the edit-history private nodes. These nodes will represent a generic character while revealing nothing about their original contents. More precisely, define a placeholder node as a Treedoc node with `data, rand = null` and `tombstone = False` (effectively, a standard Treedoc character node with `data, rand` set to `null`).

Placeholder nodes function identically under Treedoc operations while hiding the original character data. Through this observation, we can relax Step (1) of the high-level construction—instead of verifying full state equality, we need only reconstruct S_u "up to placeholders".

Definition 2. Two states $S'_u = (O'_u, T'_u, \ldots)$ and $S_u = (O_u, T_u, \ldots)$ are *equal up to placeholders* (denoted $S'_u \leq_T S_u$) if their metadata fields are equal and $(O'_u, T'_u) = (O_u, T_u)$ except for a subset of edit-history private operations and associated placeholder nodes in S'_u.

We specify that $S'_u \leq_T S_u$ only if the set of placeholder Treedoc nodes is exactly the set of nodes corresponding to the edit-history private insertions with respect to u. Otherwise, the snapshot creator can violate snapshot consistency by marking nodes as tombstones without a corresponding deletion operation in the execution.

Theorem 1. Let S_u be an honest user state and $S'_u \leq_T S_u$ with edit-history private insertions $O_T = (o_1, \ldots, o_k)$. For all well-formed partial executions E such that there exists a deletion in E targeting each $o \in O_T$, the following equality holds:

$$\mathsf{exec}(S'_u, E) = S_{new} \iff \mathsf{exec}(S_u, E) = S_{new}$$

Input: $S = (O_S, T_S, VV_S, HH_S, A_s)$ (see Table 5)

1. Compute $VV_{S,min}$ as the pairwise minimum over user state descriptor version vectors.
2. Verify that O_S is a well-formed partial execution (e.g. all version vectors are unique, operations from individual users are totally ordered, and operations are within their user state descriptor).
3. If garbage collection enabled, set $(O_{final}, T_{final}) = \mathsf{GarbageCollect}(O_S, T_S, VV_{S,min})$ (where GarbageCollect is the protocol specified in §5.3.2), else $(O_{final}, T_{final}) = (O_S, T_S)$.
4. Verify T_{final} contains no placeholder nodes (i.e. nodes with `tombstone = False` and empty `data,rand` fields).
5. Execute the following for each $u \in VV_S$:
   (a) Let $A_{S,u} = (u, acc_u, tacc_u, VV_u, VV_{u,min}, HH_u, hsig)$ be the state descriptor for u in A_S. Verify $hsig$ with PK_u.
   (b) Compute the user operation set O_u and Treedoc object T_u:
   $$O_u = \{o \in O_S \mid VV_o \leq VV_u\} \text{ and } T_u = \{n \in T_S \mid \text{for } n\text{'s insert op. } o \in O_S, VV_o \leq VV_u\}$$
   (c) Define $O_{T,u}$ to be the set of edit-history private operations with respect to u:
   $$O_{T,u} = \{o \in O_S \mid o \text{ is an insertion} \wedge VV_o \leq VV_u \wedge \exists\, o_d \text{ targeting } o \text{ s.t. } VV_{o_d} \not\leq VV_u\}$$
   (d) For each $o \in O_{T,u}$, set `tombstone = False` in the corresponding node in T_u.
   (e) Set $(O_u, T_u) \leftarrow \mathsf{GarbageCollect}(O_u, T_u, VV_{u,min})$ with input $VV_{u,min}$.
   (f) Verify that $\mathsf{accumulate}(O_u) = acc_u$ and $\mathsf{accumulate}(T_u) = tacc_u$.
   (g) Let $S'_u = (O_u, T_u, VV_u, VV_{u,min})$ be the reconstructed user state. Verify that applying the operations in $O_{missing} = O_S \setminus O_u$ (as defined in Step 5b) to S'_u results in a state $S^{(F)} = (O_S^{(F)}, T_S^{(F)}, VV_S^{(F)}, HH_S^{(F)}, _)$ such that $(O_S^{(F)}, T_S^{(F)}, HH_S^{(F)}) = (O_{final}, T_{final}, HH_S)$.
6. Define primary/secondary objects: $(O, T) = (O_{final}, T_{final})$ and $(O^{(2)}, T^{(2)}) = (O_S, T_S)$

Figure 3: Snapshot Verification Protocol. A full description of the fields contained in a snapshot can be found in Table 5. Text in blue indicates steps added by the garbage collection protocol outlined in §5.3.

Proof Sketch: Deleting a placeholder has the same effect as deleting a live node, and all placeholders are deleted in E. Since the existence of these placeholders are the only difference between S'_u and S_u, the end result of applying E must be equal. □

We conclude that executing the high-level steps using a reconstructed user state equal up to placeholders to the "true" user state still satisfies Definition 1.

Adapting State Descriptors. Recall that our earlier approach exactly reconstructed S_u from untrusted data and compared the resulting accumulators to u's state descriptor. Our new approach derives a state $S'_u \leq_T S_u$, but the accumulators of S'_u will not necessarily equal those of S_u due to the presence of placeholder nodes.

We will resolve this by re-defining the hash of an operation/node such that the hash of a placeholder node equals the hash of its original node. It follows that $\mathsf{accumulate}(O'_u) = \mathsf{accumulate}(O_u)$ and $\mathsf{accumulate}(T'_u) = \mathsf{accumulate}(T_u)$ if $S'_u \leq_T S_u$. Recall that in §4.1.1 we defined $H(\mathtt{op})$ to use an intermediate data hash `op.hdata`. When computing the hash of a tombstone or placeholder operation, we use this intermediate field directly: $H(\mathtt{op}) = H(\mathtt{op.metadata}, \mathtt{op.hdata})$.

We define the hash of a Treedoc node likewise—note that the `tombstone` field will be included in metadata such that placeholder nodes and real tombstones have differing hash values. The resulting placeholder hash value is equal to the original, non-tombstone hash. Likewise, the hash of a tombstone operation is equal to its original hash. We briefly note that this definition does not violate our FCC guarantees from history hashes, as the value of $H(\mathtt{op})$ is originally calculated as a function of the character data.

5.3 Secure Garbage Collection

Without explicit garbage collection, the size of each user state (and of snapshots) will be linear in the edit history length. In practical terms, this results in a blowup in user storage and operation processing time. In this section, we introduce a garbage collection protocol that drastically improves the performance of Acumen, while maintaining our cryptographic and consistency guarantees (§3.2).

Garbage collection is often at odds with CRDT-based collaborative editors because collection is a non-commutative operation. As a result, such operations must often be preceded by a distributed consensus protocol. In this section, we construct *asynchronous* garbage collection for Treedoc by exploiting causal stability and existing reliable causal broadcast (RCB). While previous work [5, 17, 33] provides for mild forms of asynchronous garbage collection (such as pruning obsolete metadata or leaf nodes), our approach provides full garbage collection for Treedoc (i.e. including inner nodes).

We first define the notion of causal stability (§5.3.1), then outline our GC protocol (§5.3.2), show said protocol satisfies strong eventual consistency (§5.3.3), and finally adapt it to snapshot verification (§5.3.4).

5.3.1 Causal Stability

Garbage collection in op-based CRDTs generally relies on *causal stability* [1, 2, 35], the notion that certain operations can be pruned once all users have executed them. An operation is said to be *causally stable* if it has been processed by every user. In the language of version vectors, an operation o is causally stable if $VV_o \leq VV_u$ for each user u. Equivalently, o is causally stable if $VV_o \leq VV_{min}$ where VV_{min} is the component-wise minimum over all user version vectors:

$$VV_{min}[v] = \min\{VV_u[v] : u \text{ is a user ID}\}$$

As part of our garbage collection protocol, users will identify causally stable operations in their local state. However, our asynchronous environment implies no user knows the global VV_{min}, as they may not be fully up-to-date with other user operations. Fortunately, the definition of VV_{min} extends naturally to the local case.

Let $VV_v^{(u)}$ be v's latest version vector from u's perspective (e.g. the version vector of v's latest operation in u's state). Define $VV_{u,min}$ as the component-wise minimum over all $VV_v^{(u)}$ for each user v in user u's state. An operation o is said to be *locally* causally stable if $VV_o \leq VV_{u,min}$. Local causal stability implies (global) causal stability, but the converse is not necessarily true.

Causally Stable Tombstones. As defined in §2.1, deleted Treedoc nodes are marked as *tombstones*. In base Treedoc, tombstone nodes cannot be permanently deleted, as future insertion operations can create nodes referencing a tombstone node as a parent. This occurs precisely when the operation creator has not yet processed the associated deletion operation, such that the tombstone node appears as a live node from their point of view. Thus, a deletion operation being causally stable implies the associated node will never serve as a parent to a new insertion. This property motivates the definition of **causally stable tombstones** (CSTs): tombstone nodes whose associated deletion operation is causally stable. [3]

5.3.2 Garbage Collection Protocol

We now describe our asynchronous garbage collection protocol for Treedoc. The primary challenge to such a protocol is dealing with inner nodes (i.e. non-leaf nodes). While leaf CSTs can be pruned immediately, it is not immediately clear how to prune a CST inner node that has live child nodes without disrupting required consistency guarantees.

The original Treedoc paper [33] approaches this problem by transforming ("flattening") a subtree into a single list of sibling leaf nodes. Their protocol requires running a distributed consensus algorithm to ensure no conflicting insertion can occur. Our key idea is to define an asynchronous `flatten` operation that compresses a *single edge* of the tree, replacing a CST node with its child node(s). By identifying a key invariant relating to causal stability, we ensure that `flatten` cannot conflict with concurrent insertions and therefore does not require a distributed commitment scheme.

Conditions For Garbage Collection. Recall that Treedoc orders sibling nodes according to their disambiguator (as outlined in §2.1). Flattening a node can violate this total order, since its child node(s) have different disambiguators. More concretely, consider two concurrent operations: one flattens a node with parent P and the other inserts a new node under P. Since nodes are inserted into their new child list in sorted order, the `flatten` replacing a list item with child nodes containing a different (possibly out-of-order) disambiguator may change the insertion location of the new node.

In summary, we must not flatten a node N whose parent node P may be the target of a future insertion. We identify a simple sufficient condition to satisfy such a requirement: a Treedoc CST N can be flattened *only if its parent P is also a CST*. Intuitively, if P is a CST all users see it as a tombstone, and will therefore not reference it as a parent to any insertion.

Parametrized Causal Stability. We now define our garbage collection protocol `GarbageCollect`, which takes as input a version vector VV_{gc} to define causal stability (i.e. all operations $\leq VV_{gc}$ are considered causally stable). We will elaborate on this parameter in §5.3.3. For technical reasons, `GarbageCollect` will use $VV_{gc} - 1$, i.e. VV_{gc} with each value decremented by one [4].

For Treedoc nodes, `GarbageCollect` executes as follows: **prune all leaf CSTs, and flatten inner node CSTs whose corresponding parent node is also a CST**. Figure 5 provides an example showcasing both `flatten` and leaf-pruning. Garbage collecting operations is straightforward: prune causally stable deletions and insertions whose associated deletion is causally stable.

5.3.3 Maintaining Strong Eventual Consistency

Recall the definition of strong eventual consistency (SEC) from §2.1: informally, users applying the same execution (up to concurrent ordering) end up with the same state. This property implicitly underlies fork-causal consistency in CRDT-based distributed editors, as it ensures (honest) users agree on the document state at every point in their shared execution regardless of how they locally applied concurrent edits.

However, garbage collection based on causal stability may break SEC—namely, the local causal stability of an operation can differ between two users under the same execution. For example, suppose there exists a two-party document of users

[3] The concept of causally stable tombstones is well-known—we only introduce the term CST for brevity.

[4] This is to ensure that a user's most recent operation is never garbage collected, as it may be referenced for operation verification (Fig 2).

Input: Operation set O, Treedoc object T, causal stability barrier version vector VV_{gc}
Output: Operation set O', Treedoc object T'

Let $VV_{gc}-1$ denote VV_{gc} with each entry decremented by one.

1. Filter the operation set O to generate a new set O' by removing the following items:
   - `Insertion` operations o such that $\exists o_d \in O$, o_d targets o for deletion, and $VV_{o_d} \leq (VV_{gc}-1)$.
   - `Deletion` operations o_d such that $VV_{o_d} \leq (VV_{gc}-1)$.
2. Compute the set of CSTs: $C = \{n \in T \mid n \text{ is a tombstone} \wedge \exists o_d \in O \text{ s.t. } o_d \text{ deleted } n \wedge VV_{o_d} \leq (VV_{gc}-1)\}$
3. Prune all leaf nodes in C. Repeat this step as needed if this generates new leaf CSTs.
4. For each $n \in C$ such that $n.\texttt{parent} \in C$, run the `flatten` subprocedure:
   - Let p be n's parent node, and let $L_C = n.\texttt{left} \| n.\texttt{right}$ be the concatenated child lists of n.
   - Replace n in p's left or right child list with the elements of L_C.
5. Let T' be the Treedoc object created as a result of Steps 3-4. Output (O', T').

Figure 4: Garbage Collection Protocol.

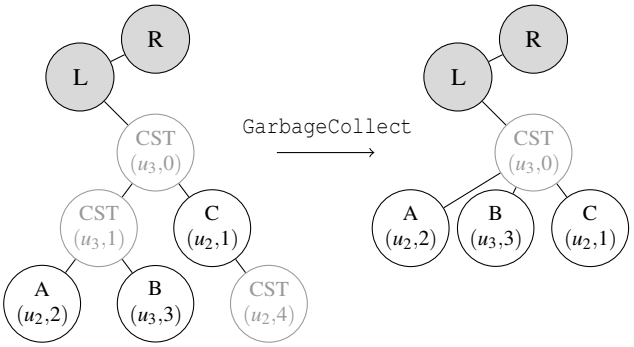


Figure 5: An example instance of garbage collection, with one call to `flatten` and one leaf node pruned.

A, B such that A inserted and deleted a single character (o_i,o_d respectively), with B processing these operations. User A does not see o_d as causally stable and does not prune the associated Treedoc node. However, User B does recognize o_d as causally stable and prunes the associated node.

To ensure strong eventual consistency, we specify `GarbageCollect` such that the set of CSTs is defined according to the *sender's* notion of (local) causal stability. More precisely, when processing an operation o with field $VV_{min,op}$ (the pairwise minimum version vector from Table 3) users run `GarbageCollect` with $VV_{gc} = VV_{min,op}$. When creating a new operation users run `GarbageCollect` with their local VV_{min}.

5.3.4 Adapting Garbage Collection For Snapshots

Recall that the high-level idea for snapshot verification (§5.2.1) is to reconstruct each user state from untrusted snapshot data and check equality via the user's signed state descriptor. This approach is at odds with the idea of permanent garbage collection—in order to retain the ability to create snapshots, a user can only remove operations/nodes that have *already been removed by every other user*. At face value, this is a circular requirement that prevents all garbage collection.

Key Insight. Our insight to avoid this issue is to perform *second-order* garbage collection by duplicating the operation set and Treedoc objects, maintaining a "primary" object (O,T) and secondary object $(O^{(2)},T^{(2)})$. Incoming operations will be applied to both primary and secondary objects. The primary object will be garbage collected as previously outlined, and user state descriptors (§5.2.2) will be computed over the *primary* object. The secondary object will be garbage collected as far as possible while still being able to reconstruct each user state descriptor. More precisely, for each user v with primary object (O_v,T_v), user u's secondary object $(O_u^{(2)},T_u^{(2)})$ must satisfy $(O_v,T_v) = \mathsf{GarbageCollect}(O_u^{(2)},T_u^{(2)},VV_{v,min})$ where $VV_{v,min}$ is defined as in §5.3.1.

Latest Garbage Collection Version Vector Recall that Acumen users run $\mathsf{GarbageCollect}(O,T,VV_{min})$ after processing or creating an operation o (where VV_{min} is taken from o). Let $VV_{v,min}^{(u)}$ denote this field from a user v's latest operation as seen by u. We now define the (local) *pairwise minimum garbage collection version vector*:

$$GCVV_{u,min}[w] = \min\{VV_{k,min}^{(u)}[w] : k \text{ is a user ID}\}$$

We define $GCVV_{u,min}$ as user u's local $GCVV$, computed from the latest state descriptors in their operation set. This definition is essentially a second-order pairwise minimum version vector (§5.3.1), in that $GCVV_{min}$ represents the maximal version vector such that $GCVV_{u,min} \leq VV_{v,min}^{(u)}$ for all users v, as seen from u's state.

Adapting Garbage Collection. As before, the primary operation set/Treedoc is garbage collected with $VV_{gc} = VV_{min}$ where VV_{min} is taken from the processed/created operation o. The secondary object will be garbage collected up to $VV_{gc} = GCVV_{min}$. To reconcile snapshot consistency with garbage collection, we make minor changes to the snapshot verification algorithm (highlighted blue in Figure 3).

Trace Index ≈:	50	250	425	850
Automerge (ins)	<0.01	<0.01	<0.01	<0.01
Automerge (del)	<0.01	<0.01	<0.01	<0.01
Acumen (ins)	0.29	0.32	0.34	0.44
Acumen (del)	0.3	0.3	0.33	0.34
Snapdoc (ins)	5.63	23.81	37.5	193.7
Snapdoc (del)	11.3	22.3	42.9	190.7

Table 6: Local operation creation timings (ms) taken from the $N=5$ user Automerge trace (§6.1).

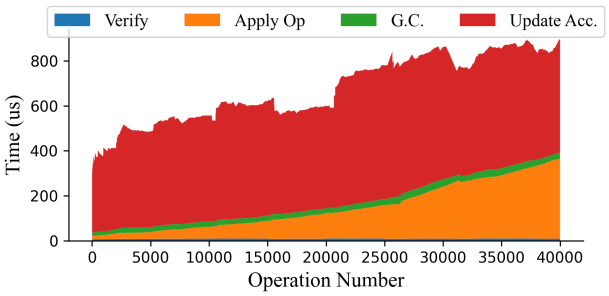


Figure 6: Acumen timings for processing remote operations. Averaged over $N=5$ users alternating to type the first 40,000 edits of the Automerge trace as outlined in §6.1.

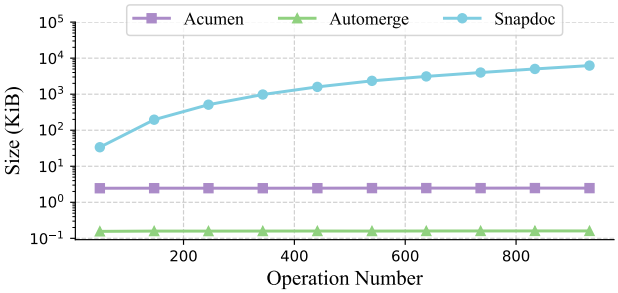


Figure 7: Log-scale insertion update sizes averaged over $N=5$ users typing the first 1,000 edits of the Automerge trace (§6.1).

6 Evaluation

We aim to answer three key questions with our evaluation:

1. How does each component of our protocol affect its ultimate performance?
2. How do user count, edit-history length, and current document size affect snapshot size and verification timing?
3. How does performance scale in real-world concurrent editing scenarios?

6.1 Local Performance

We first evaluate performance over a realistic document workload: the sequential edit trace [15, 16] of the original Automerge research paper [22], containing 182,315 single-character insertion operations and 77,463 single-character deletion operations. We compare against two baselines: Snapdoc [24], the closest prior work in terms of security guarantees, and Automerge [22], a popular open-source CRDT library without security guarantees.

Setup We evaluated the Automerge trace over $N=5$ users in alternating steps. More precisely, the k-th edit is executed by user $k \bmod N$ and the resulting update is immediately processed by each other user. This trace is evaluated on a `c2-standard-16` GCP server (16 vCPU, 64GB memory).

Local Operations Table 6 displays local operation processing times taken at different indices in the Automerge trace (corresponding to well-spaced deletions). Both Acumen and Automerge maintain real-time [5] processing times less than 1ms, whereas Snapdoc insertions and deletions take time proportional to the edit-history length (approaching 200ms after only 850 edits).

Remote Processing We quantify which aspects of Acumen contribute towards processing remote updates in Figure 6. The bulk of Acumen's overhead comes from updating our accumulators (§5.2.2), requiring $\approx 2 \cdot \lambda$ hash function evaluations and hashmap insertions per operation. Aside from periodic memory re-allocation as the hashmap grows in size, this component remains largely constant. Likewise, garbage collection ("G.C.") remains a small fraction of overall processing time, which is to be expected in documents with all users reasonably up-to-date. The bulk of the increase in processing time seen in Figure 6 comes from inserting a new character into the Treedoc and associated list (primarily the latter).

Update Size We plot update sizes under the same trace in Figure 7. As expected, both Acumen and Automerge update sizes remain constant over time. The same cannot be said about Snapdoc, whose update sizes grow linear in the edit-history length. This growth is enough to significantly degrade performance for non-trivial document sizes: after typing $\approx$ 1000 edits ($\approx$ 200 words), a Snapdoc update is already 7 MB in size.

6.2 Snapshot Performance

We next evaluate the impact of three key scaling factors on the load time and size of document snapshots: [6] the number of users in the current document, the size of the current document, and the size of the overall edit history.

Setup We instantiate a document with $|\text{users}| \in [1,10]$ and apply the aforementioned Automerge document trace up to $|\text{ops}| \in \{0,10,50,100,200,400,1000\}$. We then delete 0% or 90% of the remaining characters starting from document end.

We plot the snapshot size and load times for all combinations of $(|\text{users}|,|\text{ops}|)$ in Figure 8. To handle Snapdoc's extreme demand for memory, evaluations are run in the `n2-highmem-48` virtual machine (48 vCPUs, 384 GB memory).

[5] Acumen insert performance will eventually degrade due to inserting into a large list, as seen in Figure 6.

[6] We exclude snapshot save time for space concerns, as it scales almost exactly with current document size across all systems.

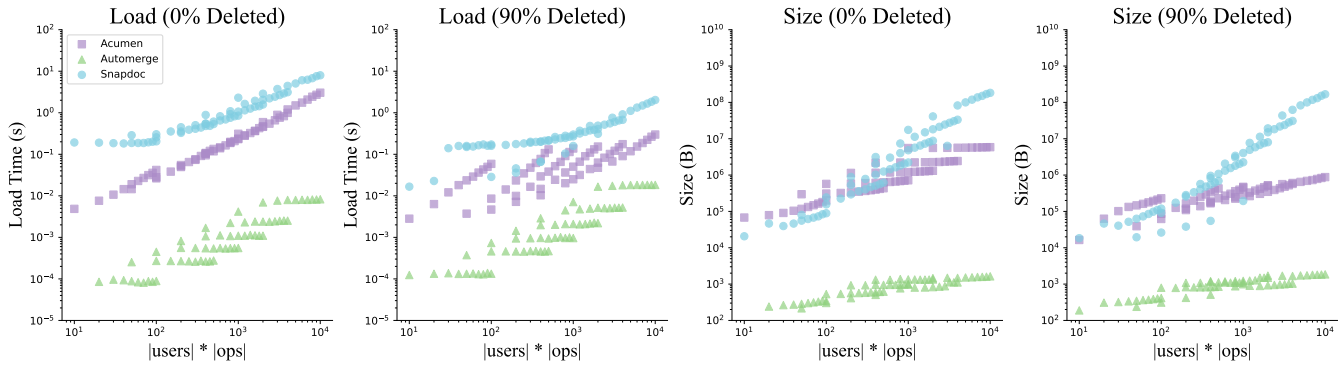


Figure 8: Snapshot Evaluations. The x-axis is composed from the product of combinations over user counts in $[1, 10]$ and operation lengths up to 1000. The first row consists of snapshots created after executing the first k operations of the Automerge trace (§6). The second row applies the same trace but deletes 90% of the document prior to creating the snapshot.

User Scaling The snapshot load performance of Acumen and Snapdoc scale approximately proportional to the number of users (all else constant). This is in contrast to a basic CRDT like Automerge which has negligible scaling on $|\text{users}|$. The impact of extra users is also visible in the snapshot size due to the inclusion of an extra state descriptor (§5.2.2), though the effect is additive and much less pronounced.

Edit-History Scaling As expected, Acumen has better performance than Snapdoc in documents with a large ratio between the edit-history length and current size (captured in the 90% deletion parameter). This is due to our garbage collection protocol: unlike Acumen, the path lengths of Treedoc nodes in Snapdoc snapshots grow proportional to edit-history length (also seen in Figure 7). Even though Snapdoc is able to exclude (most of) the deleted Treedoc nodes from its snapshot, the leftover nodes remain exceptionally large. [7]

Concretely, we see an order of magnitude improvement in load times in 0% deletion environments. In 90% deletion documents with low user count, this increases to nearly two orders of magnitude. Size difference is particularly pronounced in the 90% case: with just 1000 edits in the trace, Acumen already provides *three* orders of magnitude improvement in snapshot size.

6.3 Throughput/Latency Benchmarks

We next evaluate Acumen in full, including network overheads. Each benchmark consists of N users editing a single document, with edits broadcast over a network to individual client machines (`c2-standard-8`) through a central relay server (`c2-standard-30`). Each user repeatedly types and deletes the abstract of this paper ($\approx$ 800 characters). We compare Acumen to a baseline system using the Acumen network framework that sends and receives 200-byte edits, but does not process them locally. [8]

[7] Almost all of Snapdoc's performance degradation is caused by the variable size inherent to their Treedoc atom representation.

[8] We opt to not use Automerge as a baseline due to its performance degrading with edit history length, which prevented the measurement of steady-state ping/throughput numbers.

	# Users:	2	5	10	25
Latency (ms)	Baseline	0.37	0.465	0.51	1.5
	Acumen	1.03	1.17	1.38	2.7
Throughput	Baseline	19672	11400	7983	2240
	Acumen	1340	645	402	270

Table 7: Round-trip latency (5 ops/sec) and maximum throughput (measured as ops/sec processed by a single user).

Results Table 7 contains the results of the aforementioned benchmark. Latency is calculated as the local time difference between sending an update and receiving an acknowledgment packet (sent after the receiver processes the update in question). Direct client-to-server round-trip latency (measured via the `ping` Unix command) is approximately 0.125 ms.

We see that Acumen latency remains in the single-digit millisecond range even for large documents. Adjusted for baseline latency, our protocol adds $<$ 1ms of overhead on average. We also see that Acumen throughput scales inverse with the number of active users, since each user waits for $N-1$ updates before sending their own. Despite this, Acumen maintains throughput into the hundreds of operations per second even with large user counts.

7 Related Work

Untrusted Cloud Storage. SUNDR [26] and Depot [28] are notable secure filesystems that use signed version vectors to securely enforce causal guarantees. SPORC [13] is a central-server based secure collaborative editor utilizing *operational transforms*, but does not consider user adversaries.

(Secure) CRDTs. Many CRDT-based text editors and frameworks have been developed, an incomplete subset of which is as follows: Treedoc [33], Logoot [38], Peritext [27], Automerge [22], YJS [30], and ARDTs [25]. Barbosa et al. [3] developed a model for secure CRDTs for *honest-but-curious* adversaries. Kuessner et al.'s Algebraic Replicated Data Types (ARDTs) [25] utilize symmetric encryption at the

clients with an untrusted server that forwards the edits, but likewise only consider an honest-but-curious threat model.

Malicious-secure CRDTs have generally used hash-graphs [8, 20, 21, 31] and our work is no exception. Snapdoc [24] is the first to implement the CRDT-based snapshot consistency notion (§3). Snapdoc provides only a subset of our guarantees: their solution leaks edit patterns and does not offer the performance and scalability of the Acumen system. Also of note is work by Elvinger et al. [12] outlining a theoretical notion of authenticated snapshots. Though they did not provide an implementation, their approach shares many of our (and Snapdoc's) high-level ideas of state fingerprinting and authenticated logs from collision-resistant hashing.

CRDT Garbage Collection Efficient CRDT garbage collection has been explored in a variety of contexts. The causal stability approach is explored in [5, 17, 34] and particularly for tree-based CRDTs in [23, 33]. Zawirski et al. [39] propose a two-tiered solution involving a "core" set of highly available servers that execute the commitment protocol, then share their result with the remaining nodes.

8 Conclusion

This work introduces Acumen, a secure platform for collaborative editing with real-time performance. Our system is the first to provide confidentiality, integrity, fork-causal consistency, strong snapshot consistency and edit-history privacy in the presence of untrusted clients and servers.

Acknowledgments

We thank our anonymous reviewers and shepherd for their helpful feedback. This work is supported by gifts from Accenture, Algorithmic SuperIntelligence Labs, Amazon, AMD, Anyscale, Broadcom, cmpnd, Google, IBM, Intel, Intesa Sanpaolo, Lambda, Lightspeed, Mirendil, NVIDIA, Samsung SDS, and VESSL.

References

[1] Carlos Baquero, Paulo Sérgio Almeida, and Ali Shoker. Making operation-based crdts operation-based. In *Proceedings of the First Workshop on Principles and Practice of Eventual Consistency*, pages 1–2, 2014.

[2] Carlos Baquero, Paulo Sérgio Almeida, and Ali Shoker. Pure operation-based replicated data types. *arXiv preprint arXiv:1710.04469*, 2017.

[3] Manuel Barbosa, Bernardo Ferreira, João Marques, Bernardo Portela, and Nuno Preguiça. Secure conflict-free replicated data types. In *Proceedings of the 22nd International Conference on Distributed Computing and Networking*, pages 6–15, 2021.

[4] Niko Barić and Birgit Pfitzmann. Collision-free accumulators and fail-stop signature schemes without trees. In *International conference on the theory and applications of cryptographic techniques*, pages 480–494. Springer, 1997.

[5] Jim Bauwens and Elisa Gonzalez Boix. From causality to stability: Understanding and reducing meta-data in crdts. In *Proceedings of the 17th International Conference on Managed Programming Languages and Runtimes*, pages 3–14, 2020.

[6] Josh Benaloh and Michael De Mare. One-way accumulators: A decentralized alternative to digital signatures. In *Workshop on the Theory and Application of Cryptographic Techniques*, pages 274–285. Springer, 1993.

[7] Philippe Camacho, Alejandro Hevia, Marcos Kiwi, and Roberto Opazo. Strong accumulators from collision-resistant hashing. In *Information Security: 11th International Conference, ISC 2008, Taipei, Taiwan, September 15-18, 2008. Proceedings 11*, pages 471–486. Springer, 2008.

[8] The Matrix.org Foundation C.I.C.:. Matrix specification v1.1. technical report. https://spec.matrix.org/v1.1/.

[9] Katriel Cohn-Gordon, Cas Cremers, Benjamin Dowling, Luke Garratt, and Douglas Stebila. A formal security analysis of the signal messaging protocol. *Journal of Cryptology*, 33(4):1914–1983, 2020.

[10] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: Amazon's highly available key-value store. *ACM SIGOPS operating systems review*, 41(6):205–220, 2007.

[11] Clarence A Ellis and Simon J Gibbs. Concurrency control in groupware systems. In *Proceedings of the 1989 ACM SIGMOD international conference on Management of data*, pages 399–407, 1989.

[12] Victorien Elvinger, Gérald Oster, and François Charoy. Prunable authenticated log and authenticable snapshot in distributed collaborative systems. In *2018 IEEE 4th International Conference on Collaboration and Internet Computing (CIC)*, pages 156–165. IEEE, 2018.

[13] Ariel J Feldman, William P Zeller, Michael J Freedman, and Edward W Felten. {SPORC}: Group collaboration using untrusted cloud resources. In *9th USENIX Symposium on Operating Systems Design and Implementation (OSDI 10)*, 2010.

[14] Colin J Fidge. Timestamps in message-passing systems that preserve the partial ordering. 1987.

[15] Joseph Gentle. Editing traces (github repository). https://github.com/josephg/editing-traces, 2023.

[16] Joseph Gentle and Martin Kleppmann. Collaborative text editing with eg-walker: Better, faster, smaller. In *Proceedings of the Twentieth European Conference on Computer Systems*, pages 311–328, 2025.

[17] Richard Andrew Golding. *Weak-consistency group communication and membership*. University of California, Santa Cruz, 1992.

[18] Oded Goldreich. *Foundations of cryptography: volume 2, basic applications*, volume 2. Cambridge university press, 2001.

[19] Victor B. F. Gomes, Martin Kleppmann, Dominic P. Mulligan, and Alastair R. Beresford. Verifying strong eventual consistency in distributed systems. *Proc. ACM Program. Lang.*, 1(OOPSLA):109:1–109:28, 2017.

[20] Kristof Jannes, Bert Lagaisse, and Wouter Joosen. Secure replication for client-centric data stores. In Kaiwen Zhang, Abdelouahed Gherbi, and Paolo Bellavista, editors, *Proceedings of the 3rd International Workshop on Distributed Infrastructure for the Common Good, DICG 2022, Quebec, Quebec City, Canada, 7 November 2022*, pages 31–36. ACM, 2022.

[21] Martin Kleppmann. Making crdts byzantine fault tolerant. In *Proceedings of the 9th Workshop on Principles and Practice of Consistency for Distributed Data*, pages 8–15, 2022.

[22] Martin Kleppmann and Alastair R Beresford. Automerge: Real-time data sync between edge devices. In *1st UK Mobile, Wearable and Ubiquitous Systems Research Symposium (MobiUK 2018). https://mobiuk. org/abstract/S4-P5-Kleppmann-Automerge. pdf*, pages 101–105, 2018.

[23] Martin Kleppmann, Dominic P Mulligan, Victor BF Gomes, and Alastair R Beresford. A highly-available move operation for replicated trees. *IEEE Transactions on Parallel and Distributed Systems*, 33(7):1711–1724, 2021.

[24] Stephan A Kollmann, Martin Kleppmann, and Alastair R Beresford. Snapdoc: Authenticated snapshots with history privacy in peer-to-peer collaborative editing. *Proc. Priv. Enhancing Technol.*, 2019(3):210–232, 2019.

[25] Christian Kuessner, Ragnar Mogk, Anna-Katharina Wickert, and Mira Mezini. Algebraic replicated data types: Programming secure local-first software. In *37th European Conference on Object-Oriented Programming (ECOOP 2023)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2023.

[26] Jinyuan Li, Maxwell Krohn, David Mazières, and Dennis Shasha. Secure untrusted data repository (SUNDR). In *OSDI*. USENIX, 2004.

[27] Geoffrey Litt, Sarah Lim, Martin Kleppmann, and Peter Van Hardenberg. Peritext: A crdt for collaborative rich text editing. *Proceedings of the ACM on Human-Computer Interaction*, 6(CSCW2):1–36, 2022.

[28] Prince Mahajan, Srinath Setty, Sangmin Lee, Allen Clement, Lorenzo Alvisi, Mike Dahlin, and Michael Walfish. Depot: Cloud storage with minimal trust. In *OSDI*. USENIX, 2010.

[29] Friedemann Mattern et al. *Virtual time and global states of distributed systems*. Univ., Department of Computer Science, 1988.

[30] Petru Nicolaescu, Kevin Jahns, Michael Derntl, and Ralf Klamma. Yjs: A framework for near real-time p2p shared editing on arbitrary data types. In *Engineering the Web in the Big Data Era: 15th International Conference, ICWE 2015, Rotterdam, The Netherlands, June 23-26, 2015, Proceedings 15*, pages 675–678. Springer, 2015.

[31] Bernardo Portela, Hugo Pacheco, Pedro Jorge, and Rogério Pontes. General-purpose secure conflict-free replicated data types. In *36th IEEE Computer Security Foundations Symposium, CSF 2023, Dubrovnik, Croatia, July 10-14, 2023*, pages 521–536. IEEE, 2023.

[32] Nuno Preguiça. Conflict-free replicated data types: An overview. *arXiv preprint arXiv:1806.10254*, 2018.

[33] Nuno Preguiça, Joan Manuel Marquès, Marc Shapiro, and Mihai Letia. A commutative replicated data type for cooperative editing. In *2009 29th IEEE International Conference on Distributed Computing Systems*, pages 395–403. IEEE, 2009.

[34] Hyun-Gul Roh, Myeongjae Jeon, Jin-Soo Kim, and Joonwon Lee. Replicated abstract data types: Building blocks for collaborative applications. *Journal of Parallel and Distributed Computing*, 71(3):354–368, 2011.

[35] Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. A comprehensive study of convergent and commutative replicated data types. 2011.

[36] Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. Conflict-free replicated data types. In *Stabilization, Safety, and Security of Distributed Systems: 13th International Symposium, SSS 2011, Grenoble, France, October 10-12, 2011. Proceedings 13*, pages 386–400. Springer, 2011.

[37] Matthew Weidner, Martin Kleppmann, Daniel Hugenroth, and Alastair R. Beresford. Key agreement for decentralized secure group messaging with strong security guarantees. In Yongdae Kim, Jong Kim, Giovanni Vigna, and Elaine Shi, editors, *CCS '21: 2021 ACM SIGSAC Conference on Computer and Communications Security, Virtual Event, Republic of Korea, November 15 - 19, 2021*, pages 2024–2045. ACM, 2021.

[38] Stéphane Weiss, Pascal Urso, and Pascal Molli. Logoot: A scalable optimistic replication algorithm for collaborative editing on p2p networks. In *2009 29th IEEE International Conference on Distributed Computing Systems*, pages 404–412. IEEE, 2009.

[39] Marek Zawirski, Marc Shapiro, and Nuno Preguiça. Asynchronous rebalancing of a replicated tree. In *Conférence Française en Systèmes d'Exploitation (CFSE)*, page 12, 2011.

A Strong Eventual Consistency

Recall the definition of strong eventual consistency [36]:

Definition A.1. (Strong Eventual Consistency, Informal) A CRDT satisfies strong eventual consistency if for all users A,B receiving the same set of updates (possibly in a different order), A's state equals B's state.

It is known that a CRDT satisfies SEC if concurrent operations commute [19]. We now sketch a proof that Acumen with garbage collection satisfies SEC over all well-formed executions.

At a high level, our proof will proceed as follows:

- We first show that garbage collection is commutative with itself: GC at VV_1 before VV_2 results in identical outcomes to GC at VV_2 before VV_1.
- Next, we show that GC is unaffected by insertions and deletions.
- Finally, we conclude that GC satisfies SEC.

Lemma A.1. Let $S_u = (O_u, T_u)$ be an arbitrary Acumen state, and `GarbageCollect` be as defined in Figure 4. The following identity holds:

$$\texttt{GarbageCollect}(\texttt{GarbageCollect}(O_u,T_u,VV_1),VV_2) = \texttt{GarbageCollect}(O_u,T_u,VV_1 \cup VV_2)$$

where $VV_1 \cup VV_2$ denotes the component-wise maximum of VV_1 and VV_2.

Proof Sketch: We first briefly argue that `flatten` is commutative within the same call to GC — that is, we can flatten parent/child nodes in any order. It is easy to see that pruning leaf nodes and operation tombstones commute. The primary case to consider is thus a chain of CSTs $A \to B \to C$, and whether flattening (A,B) preceding (B,C) affects the outcome. It is straightforward to see that this is not the case: both executions collapse into the single node C. We note that this holds even for different-direction children (e.g. $A \to B$, $A \to C$ for left-child B, right-child C), since flattening A moves both children into its current position (e.g. inserts them into the same list in their respective order).

We now argue that this individual commutativity applies to GC operations themselves. Intuitively, the GC protocol identifies a set of eligible operations/nodes as those $\leq VV_{gc} - 1$. The set identified at $VV_1 \cup VV_2$ is equal to the union of the sets at VV_1,VV_2. Since flattening elements in the set can occur in any order, it follows that the GC two sets sequentially is equal to the GC of the union. (Note `flatten` does not impact the CST status of other nodes)

Lemma A.2. (Informal) Insertion and deletion operations commute with garbage collection.

Proof Sketch: Intuitively, an insertion can only target non-CST parent nodes. It follows that insertions do not affect the resulting state of any call to GC, as they modify node state untouched by GC. Deletions follow likewise.

Theorem A.1. Acumen with garbage collection satisfies strong eventual consistency.

Proof Sketch: Immediate from the previous lemmas: garbage collection is commutative and unaffected by concurrent operations, so overall Acumen operations are commutative ($\implies$ SEC).

B Fork-Causal Consistency

We now provide a formal definition and proof sketch of fork-causal consistency, assuming strong snapshot consistency (to be proved in §C). To do so, we construct a *happens-before* global execution graph G that represents the causal relationships between operations. This graph represents a "global view" of the causal execution (restricted to honest users). We then construct individual client execution graphs G_n to represent their "local view" of the causal execution. By comparing G and G_n at every shared node, we prove that our system is fork-causal consistent.

Definition B.1. Let G be a directed acyclic graph. The partial order $\prec_G$ holds if there is a directed edge $a \to b$ or a transitive path from a to b in G.

Definition B.2. For any vertex v in the graph G, define $\mathsf{read}(G,v)$ to be the tuple (O,T) of operation set / Treedoc nodes from the state $\mathsf{exec}(\emptyset,E)$ where $E = \{n \in G \mid n \preceq v\}$ (the subgraph leading into v).

Likewise, define $\mathsf{read}(G_n,v)$ to be the operation set/Treedoc of $\mathsf{exec}(\emptyset_u,E)$, where $E = \{n \in G_n \mid n \preceq v\}$.

Definition B.3. A global execution E is **fork-causal consistent** if there exists a directed acyclic graph G that satisfies the following properties:

FCC0. G contains a vertex for every operation in E sent by an honest user, and also a vertex for every operation that is sent by a malicious user and processed by at least one honest user.

FCC1. The operations of honest users are totally ordered in G, consistent with the actual execution order of the operations at that device.

FCC2. For each honest user n there exists a directed acyclic graph G_n in which there is a vertex for every operation message sent or processed by n. By FCC0, for each vertex v in G_n there is a corresponding vertex m in G. We then require that for each vertex v in G_n, the document state is the same as the document state at the corresponding vertex in G: $\mathsf{read}(G_n,v) = \mathsf{read}(G,v)$.

FCC3. The operations of all users appear totally ordered to honest clients. Formally, if $v \in G$ is a node created by an honest client and $v_1, v_2 \prec v$ are vertices from a user u, then $v_1 \prec v_2$ or $v_2 \prec v_1$.

Definition B.4. We construct G_n (the local execution graph for an honest user n) as follows:

1. Add a vertex for every operation witnessed or sent by n.
2. Add a directed edge $a \to b$ for locally witnessed/sent operations a, b where a is a causal dependency of b. That is, draw an edge from each operation as defined in the version vector, excepting the sender's which draws an edge from the operation with counter $VV[u]-1$. [9].
3. For users joining from snapshots, add the partial execution as given by O_S (i.e. $E_{snapshot}$). Add edges into an `AddUser` node from its causal dependencies present in partial execution.

Definition B.5. We construct G (the global execution graph) as follows:

1. Add a vertex for every operation witnessed or sent by an honest client.
2. Add a directed edge $a \to b$ for locally witnessed/sent operations a,b where a is a causal dependency of b.

   This includes edges from a new user's first message upon joining, which has an edge to the associated `AddUser` node and edges to the same dependencies as the `AddUser` node.

Lemma B.1. G is a DAG.

Proof Sketch: By construction, an edge $a \to b$ exists only if $VV_b > VV_a$ (since operations have edges from their causal dependencies, and the version vector of a given operation dominates the version vectors of its dependencies). This prevents any cycles from appearing on the graph, as the version vector counter is monotonically increasing.

[9] This is precisely the dependencies checked by our operation verification protocol in Fig. 2

Lemma B.2. G satisfies FCC0, FCC1, and FCC3.

Proof Sketch: FCC0 is by definition. FCC1 follows immediately from our definition involving version vectors, as mentioned in Lemma B.1. An edge always exists between a user's k-th node and their $k+1$-th as required.

FCC3 follows much the same as FCC1. An honest user accepts an operation from user u only if its counter is precisely $+1$ from that user's most recent operation counter. Therefore forking operations with identical counters will be ignored, and honest users always see a total order for the operations from any given individual user.

The above reasoning holds for nodes added from a snapshot, as snapshot verification will ensure the well-formedness of the partial execution.

Lemma B.3. Assume that each snapshot node in G satisfies snapshot consistency as in Definition 1 (§5.2). Then G satisfies FCC2.

Proof Sketch: Suppose not, i.e. suppose there exists some global execution E that leads to a FCC2 violation. By snapshot consistency, the state of a snapshot user is precisely equal to that of an honest user processing the same execution up to and including the relevant `AddUser`.

Consider then the same execution over our basic protocol (§4), i.e. where snapshots are created by including the entire edit history. This protocol is clearly fork-causal consistent (for a proof, see Snapdoc [24] "basic protocol").

We claim that the counterexample execution E would also break FCC over the basic protocol. Consider an arbitrary FCC2 violation in E such that $\mathsf{read}(G_n,v) \neq \mathsf{read}(G,v)$ for some v and honest snapshot user n. By strong snapshot consistency, the respective user n in the basic version would have an equal value at v in their $G_{n,basic}$. This would imply $\mathsf{read}(G,v) \neq \mathsf{read}(G_{basic}, v_{basic})$, but this is a contradiction as G is solely determined by E.

C Snapshot Consistency Proof Sketch

We prove that our snapshot verification algorithm given in Figure 3 satisfies Definition 1, assuming that the underlying Acumen system is fork-causal consistent (§B) and that every existing user snapshot satisfied snapshot consistency (effectively inducting over snapshots). Namely, we prove that an arbitrary snapshot S satisfying the verification protocol must satisfy Definition 1.

Our proof will proceed according to the following outline:

1. We first prove that the protocol correctly reconstructs each user state S'_u up to placeholders (§5.2.3).
2. We then prove Theorem 1, which informally states that the outcome of applying the operations between the reconstructed state S'_u and the snapshot S has equal outcome to applying the same execution to the real state S_u (intuitively, that placeholders do not affect the result of `exec` under snapshot conditions).

3. Next, we prove that the result of applying $E_{S,u}$ (the missing operations between S_u and $S_{snapshot}$) to the real user state S_u results in $S_{snapshot}$. This will follow directly from the previous step and the fact that verification ensures $\mathsf{exec}(S'_u, E_{S,u}) = S_{snapshot}$.
4. We then show that $E_u \cup E_{S,u}$ forms a fork-causal consistent execution.
5. Finally, we apply the previous two steps to show S satisfies Corollary 1, and conclude it satisfies Definition 1.

The following statements implicitly assume the snapshot S passed snapshot verification as in Figure 3.

Lemma C.1. Let $S'_u = (O'_u, T'_u)$ be the state derived in Step 5 of the snapshot verification protocol (Fig. 3) for some user $u \in VV_S$. We have $S'_u \leq_T S_u$ (where $\leq_T$ denotes equality up to placeholders as defined in Def. 2).

Proof Sketch: Intuitively, this lemma proves that the reconstruction of the user state during snapshot verification is correct. More precisely, we show that $O'_u = O_u$ and $T'_u = T_u$ except for a subset of placeholders.

Recall that the hash of a placeholder is equivalent to that of the standard operation/node, and that our accumulator construction (§2.3) functions over the hash values of its inputs rather than their direct values. It follows that the accumulator over the placeholder-free set O_u equals that of the placeholder set O'_u.

Any other difference between two sets will violate the set-collision resistance of accumulate, since the difference will appear in the individual hashes. We conclude that $\mathsf{accumulate}(O_u) = \mathsf{accumulate}(O'_u)$ if and only if $O'_u \leq_T O_u$. The same for Treedoc accumulators follows immediately.

Since these accumulators are signed as part of the state descriptor (e.g. Step 5a), we conclude that the overall state $S'_u \leq_T S_u$. □

We now prove Theorem 1. Restated:

Theorem C.1. Let S_u be an honest user state and $S'_u \leq_T S_u$ with edit-history private insertions $O_T = (o_1, \ldots, o_k)$. For all well-formed partial executions E such that there exists a deletion in E targeting each $o \in O_T$, the following equality holds:

$$\mathsf{exec}(S'_u, E) = S_{new} \iff \mathsf{exec}(S_u, E) = S_{new}$$

Proof Sketch: This follows directly from the idempotency of deletion operations (intuitively, deletions are permanent — once marked as a tombstone, an operation/node remains so). Moreover, deleting a placeholder op/node has the same effect as deleting a live op/node.

It follows that in both executions, each edit-history private insertion o and node n_o are marked as (normal) tombstones by the end of E. Since the placeholders in O_T are the only difference in S'_u, S_u, the resulting end states are exactly equal. □

Lemma C.2. Define $E_{S,u}$ as the partial execution derived from $O_{missing,u} = O_S \setminus O_u$ (step 5g). Let E_u be the real execution corresponding to S_u. The following expression holds:

$$\mathsf{exec}(S_u, E_{S,u}) = S_{snapshot}$$

Proof Sketch: Immediate from Lemma C.1, Theorem C.1, and Step 5g.

Lemma C.1 implies that $S'_u \leq_T S_u$ (where S'_u is the reconstructed state from 5g, and S_u is the real user state). Step 5g of the verification protocol ensures that $\mathsf{exec}(S'_u, E_{S,u}) = S_{snapshot}$. Finally, Theorem C.1 states that $\mathsf{exec}(S'_u, E) = S_{new} \implies \mathsf{exec}(S_u, E) = S_{new}$. We conclude the statement. □

Lemma C.3. The execution $E_u \cup E_{S,u}$ is fork-causal consistent.

Proof Sketch: Immediate from Step 2 (verifies the execution from O_S is FCC, implying the subset of O_S induced by u is FCC). □

Finally, recall Corollary 1:

Corollary 1. A snapshot S is consistent if the following two conditions hold:

- There exists some partial execution $E_{missing}$ such that $\mathsf{exec}(S_{honest}, E_{missing}) = S_{snapshot}$ for some honest user state S_{honest}.
- $E_{honest} \cup E_{missing}$ is a fork-causally consistent execution.

Theorem C.2. Snapshots verified by the Acumen snapshot verification algorithm (Figure 3) satisfy Corollary 1 and therefore Definition 1.

Proof Sketch: Lemma C.2 proves the first condition. Lemma C.3 proves the second. □

D Snapdoc's Snapshot Consistency

The full Snapdoc snapshot comprises an operation set O_S and set of state attestations A_S, one per user with at least one operation in O_S. The snapshot verification protocol executes the following high-level steps per user u listed in the snapshot:

1. Using the snapshot operation set O_S and u's version vector VV_u from their state attestation, derive
$$O_S \cap O_u = \{o \in O_S \mid VV_o \leq VV_u\}$$
2. Verify that the derived value for $O_S \cap O_u$ is a subset of O_u using u's state attestation.
3. Verify that the history hashes HH_u are a prefix of the snapshot history hashes (derived from O_S), and conclude that no forks exist between users up to VV_S.

We note that this protocol differs from our high-level approach in that the set of snapshot operations at time VV_u is checked to be a **subset** of O_u, rather than full equality. This is due to the aforementioned edit-history privacy (§5.2) requirement that the sensitive data in operations in $O_S \setminus O_u$ remain hidden. Snapdoc opts to directly withhold these operations from the snapshot.

This difference results in Snapdoc satisfying a weaker form of snapshot consistency, where an attacker can produce a valid snapshot comprised of any subset of the "true" state. For instance, an attacker can issue a snapshot for a document reading "AC" when it should be "ABC". More formally, if S_A is the set of operations in a valid snapshot, then it is possible to create a valid snapshot for any $X \subset S_A$.

Careful readers will notice that any user can locally transform the document with an arbitrary set of operations before creating a snapshot, such that the resulting document is edited to the sender's liking. The resulting snapshot is technically valid, but the key difference to above attack is that the sender edits are *codified as operations* and are visible to the snapshot receiver. This satisfies Definition 1, since these operations are part of a fork-causal consistent execution. This contrasts with the Snapdoc attack, since the deletion of operations is not caused by any operation in the execution. For instance, suppose an honest user issued insertions to create the word "TRAIN". A malicious user could create a snapshot *consisting only of the honest user operations* deriving "RAN", even though this state never appeared in the global causal execution.

Snapdoc requires new users to broadcast a `noop` operation including their own state attestation. The new user will then only accept messages that include the `noop` as a causal dependency. If the user's snapshot is malformed (in a non-trivial way as to pass local verification), they will never process honest user edits, nor will honest users process their updates, as if they were on different causal forks.

This satisfies the overall definition of fork-causal consistency, but is still undesirable since the new user cannot tell that their state is malformed without out-of-band communication from an honest party.

Drs.NAS: Ultra-Efficient Neural Architecture Search for Recommendation Systems

Ruixuan Wang, Xun Jiao
Villanova University

Abstract

Deep learning-based recommendation systems (DRS) have become a dominant workload in hyperscale data centers. However, designing DRS architectures that balance high predictive performance with computational efficiency remains a major challenge due to ever-increasing model complexity and scale. Neural architecture search (NAS) has recently emerged as a promising automated design approach and is now adopted in production by major hyperscalers. Yet, existing NAS methods face two critical limitations: (*i*) prohibitive search costs—often requiring several GPU hours to days—which hinder rapid iteration, and (*ii*) the resulting architectures are typically computation- and memory-intensive, limiting practical deployment. In this paper, we propose Drs.NAS, an *ultra-efficient* NAS framework for DRS. (*i*) **Ultra-efficient search:** We propose a novel metric, *superproxy*, which enables NAS without the costly training and validation required by existing NAS methods. Compared to SOTA NAS search times of $5\sim18$ GPU-hours, Drs.NAS completes the search *within two minutes* on a commodity *CPU*. (*ii*) **Ultra-efficient results:** The models discovered by Drs.NAS drastically reduce resource demands—achieving on average $108.3\times$ and $34.9\times$ smaller model sizes, and $88.8\times$ and $14.7\times$ fewer FLOPs, compared to handcrafted and SOTA NAS results, respectively. Crucially, these gains come *without sacrificing predictive quality*: Drs.NAS delivers on par or even superior predictive performance, surpassing handcrafted and NAS baselines by 0.0123 and 0.0056 in average AUC across three representative benchmarks, respectively.

1 Introduction

Recommendation systems play a crucial role in delivering personalized content across diverse applications, including e-commerce, advertising, social media, and search engines [Zhang et al.(2019), Song et al.(2020), Guo et al.(2017), Song et al.(2019), Zhang et al.(2023)]. With the advent of deep learning, their scale and complexity have grown by more than an order of magnitude since 2017, giving rise to deep learning-based recommendation systems (DRS). Today, DRS workloads have become dominant in hyperscale data centers; for instance, they account for over 70% of all AI inference cycles at Meta's data centers [Gupta et al.(2020)].

While deep learning greatly improves predictive performance by capturing complex multi-modal feature interactions, it also introduces substantial computational demands. These demands strain data center resources and challenge quality-of-service (QoS) requirements, such as tail latency and response time [Berger et al.(2018), Chen et al.(2024)]. Designing efficient architectures for DRS is further complicated by the reliance on manual, domain-expert-driven design, which involves costly and iterative exploration of large architectural design spaces—an approach that does not scale.

To overcome these challenges, neural architecture search (NAS) has emerged as a promising paradigm for automating DRS architecture design and has been adopted in production by several hyperscalers, e.g., Meta [Song et al.(2020), Zhang et al.(2023), Wen et al.(2024)]. NAS effectively explores large design spaces with heterogeneous operators and connection types. For example, AutoCTR [Song et al.(2020)] introduces a hierarchical search space with diverse feature interaction operators, while NASRec [Zhang et al.(2023)] expands the space further by incorporating heterogeneous operations and flexible connection patterns. By leveraging enriched search spaces and automation, NAS methods often achieve state-of-the-art (SOTA) performance across benchmarks, while often reducing model complexity compared to manually designed DRS [Zhang et al.(2023), Song et al.(2020), Zhu et al.(2022)].

Main Challenges: Despite achieving SOTA predictive performance, existing NAS methods for DRS architecture search face two major challenges:

- **Prohibitive search costs:** Current NAS approaches for DRS [Zhang et al.(2023), Song et al.(2020), Zhu et al.(2022)] rely on iterative training and validation cycles to perform the search process. This results in substantial computational overhead, often requiring hours

Table 1: Performance comparison between the SOTA NAS approaches and Drs.NAS over three widely adopted recommendation system benchmarks. **In this paper, the "performance" specifically denotes the model predictive performance**, and Drs.NAS achieves the best or second-best Log Loss (lower is better) and AUC score (higher is better) among all NAS baselines, with significant search cost reduction (from GPU-hours to CPU-minutes) listed in this table and computational overhead reduction of NAS-discovered models as shown in Fig. 4(g) to Fig. 4(l).

	Criteo		Avazu		KDD		Average
	Log Loss↓	AUC↑	Log Loss↓	AUC↑	Log Loss↓	AUC↑	Search Cost
PROFIT [Gao et al.(2021)]	0.4427	0.8095	0.3735	0.7883	-	-	12 hours on GPU
AutoCTR [Song et al.(2020)]	0.4413	0.8104	0.38	0.7791	0.152	0.8011	18 hours on GPU
NAS-CTR [Zhu et al.(2022)]	0.4421	0.8104	0.3754	0.7884	-	-	5.04 hours on GPU
NASRec [Zhang et al.(2023)]	0.4398	0.8116	0.3744	0.7891	0.1491	0.8155	6 hours on GPU
Drs.NAS (proposed)	**0.4404**	**0.8109**	**0.3736**	**0.7905**	**0.1479**	**0.8204**	**1.56 minutes on CPU**

or even days on GPUs. Such costly procedures make NAS time-consuming, resource-intensive, and difficult to scale, thereby hindering the rapid iteration of model architecture design for ever-increasing application tasks.

- **Computation- and memory-heavy architectures:** Existing NAS methods for DRS primarily optimize for predictive performance metrics such as Log Loss or AUC (area-under-curve), while neglecting the computation overhead of the searched architectures, including memory usage (i.e., the number of parameters) and computational cost (i.e., the number of FLOPs).

Our Solution: To overcome these limitations, we propose Drs.NAS, an *ultra-efficient* NAS framework for DRS architecture search. Unlike conventional NAS methods, Drs.NAS avoids the requirement for expensive training and validation of candidate architectures. The key idea is a novel composite metric, *superproxy*, which integrates seven different NAS proxy dimensions as a composite indicator for predictive performance and architectural efficiency. Building on the superproxy, we develop a composite loss function based on the superproxy that allows joint optimization of predictive performance and computational efficiency.

The performance comparison between Drs.NAS and SOTA NAS baselines is indicated in Tab. 1. To the best of our knowledge, Drs.NAS is the first zero-cost proxy-based NAS framework for DRS architecture search that simultaneously delivers three key advantages:

- **Ultra-efficient search:** By leveraging the superproxy, Drs.NAS eliminates the need for costly training and validation during architecture search. As a result, the entire search process completes in *under two minutes* on a commodity *CPU*, compared to $5 \sim 18$ GPU-hours required by SOTA NAS baselines [Gao et al.(2021), Song et al.(2020), Zhu et al.(2022), Zhang et al.(2023)], yielding up to a $692\times$ improvement in search time efficiency.
- **On par or superior predictive performance:** Despite the drastic architecture search efficiency gains, Drs.NAS achieves comparable or better predictive performance. Specifically, Drs.NAS delivers on par or even superior predictive performance and surpasses NAS baselines by 0.0056 in the averaged AUC score.
- **Ultra-efficient architectures:** Drs.NAS enables co-optimization of both predictive performance and computational efficiency. Beyond marginal predictive performance gains, Drs.NAS reduces model size and FLOPs by averages of $34.9\times$ and $14.7\times$, respectively, across three widely adopted recommendation system benchmarks compared to SOTA NAS baselines.

This paper makes the following contributions:

- We propose Drs.NAS, an ultra-efficient NAS framework for DRS. By introducing the superproxy as a composite performance/efficiency indicator, Drs.NAS removes the need for expensive training and validation during search, reducing search time from GPU-hours to CPU-minutes.
- We develop a composite loss function based on superproxies and formulate a multi-objective optimization problem that balances predictive performance and computational efficiency. We solve this optimization problem utilizing a gradient-based search strategy.
- We conduct comprehensive evaluations of both the Drs.NAS framework and the architectures it produces. Results show that Drs.NAS consistently achieves substantial efficiency improvements in computation and memory, while delivering on par or even superior predictive performance compared to handcrafted and NAS baselines on three representative recommendation system benchmarks.

2 Background and Related Work

2.1 Background of DRS Development

DRS are extensively developed and widely adopted in different fields [Guo et al.(2017), Song et al.(2019), Naumov

et al.(2019)]. Specifically, DeepFM [Guo et al.(2017)] integrates factorization machines with neural networks to learn different feature interactions, demonstrating strong performance and efficiency. AutoInt [Song et al.(2019)] leverages the self-attention mechanism and residual connections to capture feature interactions more effectively, and the AutoInt+ [Song et al.(2019)] model further boosts the predictive performance by integrating implicit feature interactions. Moreover, the widely deployed Deep Learning Recommendation Model (DLRM) [Naumov et al.(2019)] employs both dense and sparse features, providing a scalable architecture with high performance.

Despite the mainstream DRS achieving strong performance, these models heavily rely on manual development, requiring substantial domain expertise and interventions. Such manually designed DRS not only increase development cost but also limit the architectural design space exploration, potentially leading to suboptimal performance and inefficient implementation.

2.2 Background of NAS

Weight-Sharing NAS: Weight-sharing NAS approaches [Cai et al.(2019), Zhang et al.(2023), Zhu et al.(2022), Gao et al.(2021)] mitigate the prohibitive computational costs of traditional NAS methods by training a single over-parameterized supernet to represent the entire search space. A search algorithm, such as the evolutionary algorithm (EA), is then employed to select the best-performing subnet [Cai et al.(2019),Zhang et al.(2023)]. Despite being widely applied for DRS architecture search, training the supernet requires substantial computational resources, and each subnet often requires iterative validation and fine-tuning, both of which are time-consuming and computationally expensive.

Hardware-aware NAS: Hardware-aware NAS methods integrate execution latency into the search objective to discover computationally efficient and high-performance architectures on target devices [Wu et al.(2019), Cai et al.(2018), Gao et al.(2025)]. However, most existing approaches depend on latency profiling or auxiliary latency prediction models, which introduce additional measurement complexity and computational overhead. Moreover, conventional hardware-aware NAS methods still require training large supernets, resulting in substantial search costs.

Zero-cost proxy-based NAS: In parallel, to reduce search cost, the zero-cost proxy-based NAS strategy utilizes saliency metrics to estimate the performance of DNN models without any training [Li et al.(2023), Abdelfattah et al.(2021), Jiang et al.(2023)]. These proxies are computed on model weights, activations, or gradients using a single forward and backward propagation and aggregated across all model parameters to approximate predictive performance. While highly efficient, existing zero-cost proxies were primarily developed for CNN and Transformer architectures [Abdelfattah et al.(2021), Li et al.(2023), Mellor et al.(2021), Jiang et al.(2023), Zhou et al.(2024)], and their applicability to DRS architectures, characterized by dense/sparse operations and corresponding feature interactions, remains largely unexplored. In addition, prior methods mainly focus on model accuracy, with limited attention to jointly optimizing predictive performance and computational efficiency [Abdelfattah et al.(2021), Li et al.(2023), Jiang et al.(2023)]. Furthermore, recent studies show that no individual proxy generalizes reliably across tasks and benchmarks [Abdelfattah et al.(2021), Huang et al.(2024), Cortês et al.(2025)], inspiring the ensemble design that aggregates multiple proxies in the NAS methods for more robust performance estimation.

2.3 NAS for DRS

The NAS methods have emerged as a promising strategy for automating the development of DRS architectures, demonstrating strong performance on various recommendation system benchmarks. In particular, PROFIT [Gao et al.(2021)] proposes a progressive differentiable NAS algorithm to discover optimal feature-interaction layers in deep sparse networks, using a low-rank approximated search space. AutoCTR [Song et al.(2020)] utilizes evolutionary algorithms to automatically explore the effective feature interaction architectures in a graph-based search space. NAS-CTR [Zhu et al.(2022)] further integrates architecture optimization with parameter learning via a differentiable NAS method. More recently, NASRec [Zhang et al.(2023)] adopts the weight-sharing NAS paradigm and substantially broadens the search space by incorporating heterogeneous operators such as sigmoid gating [Wang et al.(2021)] and transformer-based modules [Vaswani et al.(2017)], achieving promising performance on multiple benchmarks.

However, prior NAS approaches remain computationally intensive and time-consuming, often requiring substantial GPU resources for architecture search, while the discovered architectures can be resource-intensive and costly to deploy, making these NAS methods impractical for real-world deployment. Moreover, as observed in recent industrial studies [Wen et al.(2024)], NAS for DRS becomes an iterative execution in the recurring engineering cycle to continuously battle the dynamic baselines instead of a one-time investment per task. Therefore, reducing per-task search time and the inference cost of searched architectures is critical to making automated architecture optimization practically deployable.

To overcome these limitations, Drs.NAS introduces a superproxy that comprises seven zero-cost proxies as a performance/efficiency indicator to guide DRS architecture search, which can eliminate the requirement for supernet training or subnet fine-tuning and discover architectures with strong predictive performance and high computational efficiency. While individual zero-cost proxies have been explored in other domains, the strategic composition of the superproxy in NAS

for DRS represents a significant and non-trivial advancement.

3 Proposed Approach: Drs.NAS

3.1 Drs.NAS Overview

Fig. 1 presents the overview of Drs.NAS framework, which is mainly composed of four major phases. In phase 1 (Sec. 3.2), we propose and formulate the superproxy metric, which maps each candidate operator to a seven-dimensional proxy representation. In phase 2 (Sec. 3.3), we construct a graph-based search space built on top of the computed superproxies, where vertices represent candidate layers and edges represent candidate dataflows. This search space eliminates the requirement of expensive training and validation of candidate architectures based on superproxies. In phase 3 (Sec. 3.4), we relax the discrete graph search space, using Gumbel-Softmax, to enable a differentiable NAS process. Finally, in phase 4 (Sec. 3.5), we formulate a multi-objective optimization problem over the relaxed space and develop a composite loss that balances predictive performance and computational efficiency. We utilize a gradient-based method to solve the optimization and discover the optimal model architecture.

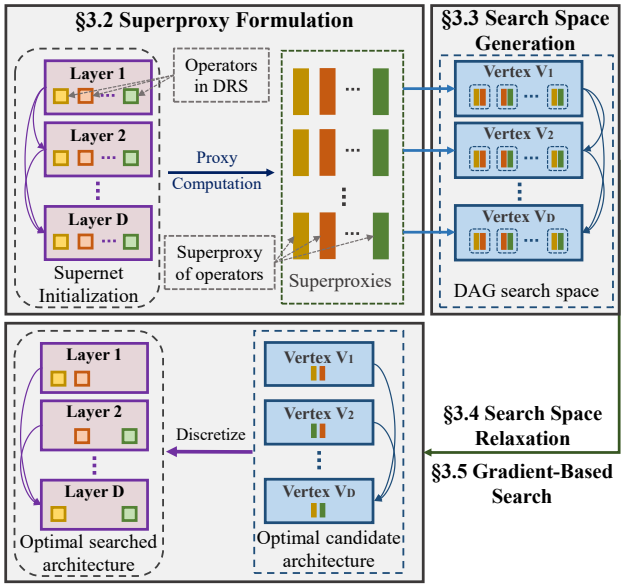


Figure 1: The overview of Drs.NAS framework for the DRS architecture search, consisting of four sequential phases.

3.2 Superproxy Formulation

In Drs.NAS, we focus on two types of zero-cost proxies: the overhead-relevant proxy, which is correlated with the model size and computational overhead, and the overhead-irrelevant proxy, which reflects the model performance and trainability. We initialize a supernet following NASRec [Zhang et al.(2023)], and compute the corresponding proxies. By ensembling these proxies, the superproxy serves as the performance and efficiency indicator, guiding the optimization toward improved predictive performance while regularizing the stability of the gradient-based search process.

3.2.1 Overhead-relevant Proxy

First of all, we consider two standard proxies: the number of parameters (#Params) and FLOPs (#FLOPs). These proxies are generally highly connected with the model size and computational overhead, while the #Params and #FLOPs are also widely utilized proxies in prior NAS methods [Li et al.(2023), Jiang et al.(2023), Lin et al.(2020)] as model performance indicators. The #Params and #FLOPs for weight matrix θ are indicated in Eq. (1). Here, the $Numel(\theta)$ indicates the total number of elements in the weight matrix θ, and X is a subset of randomly selected training samples.

$$S_{param}(\theta) = Numel(\theta);\ S_{flops}(\theta) = FLOPs(\theta; X) \quad (1)$$

We also include the weight norm (w_{norm}) and the gradient norm (g_{norm}) proxies, defined as the Frobenius norm of the weight matrix θ and the corresponding gradients, respectively. These proxies are correlated with the parameter counts and are widely adopted for efficient NAS [Lukasik et al.(2025), Abdelfattah et al.(2021)]. The computation of the w_{norm} and g_{norm} proxies for the weight matrix θ is shown in Eq. (2).

$$S_{w_{norm}}(\theta) = \|\theta\|_F;\ S_{g_{norm}}(\theta) = \left\| \frac{\partial L}{\partial \theta} \right\|_F \quad (2)$$

3.2.2 Overhead-irrelevant Proxy

Additionally, we incorporate the well-established zero-cost proxies that are leveraged to estimate the predictive performance and trainability of DNN models directly from the model initializations [Li et al.(2023), Jiang et al.(2023), Tanaka et al.(2020), Abdelfattah et al.(2021)]. These proxies are not strongly correlated with model size and computational overhead. Specifically, we leverage the zico [Li et al.(2023)], synflow [Tanaka et al.(2020)], and meco [Jiang et al.(2023)] proxies in Drs.NAS.

Zico captures the relationship between the gradient statistics of the weight matrix θ, such as the mean and standard deviation of the corresponding gradient, and the trainability and generalization capacity of the DNN models. The zico proxy computation of θ is demonstrated in Eq. (3).

$$S_{zico}(\theta) = \log\left(\frac{\mathcal{E}[|\partial L/\partial \theta|]}{\sqrt{Var(|\partial L/\partial \theta|)}} \right) \quad (3)$$

Synflow utilizes the Hadamard product of the model weight θ and the corresponding gradients to represent the contribution of each weight matrix θ within the computational flow of

the entire DNN model. The computation of the synflow proxy for θ is denoted as Eq. (4).

$$S_{syn}(\theta) = \frac{\partial L}{\partial \theta} \circ \theta \tag{4}$$

Meco computes the Pearson correlation matrix $P(\cdot)$ of feature maps $\mathcal{F}$ derived from the model weight matrix θ and relates the minimum eigenvalue of the correlation matrix to the model accuracy and trainability. The meco proxy computation for weight matrix θ is shown as Eq. (5), where $\lambda_{\min}(\cdot)$ represents the minimum eigenvalue.

$$S_{meco}(\theta) = \lambda_{\min}(P(\mathcal{F}(X;\theta))) \tag{5}$$

To construct the operator-level superproxy, we first compute the seven different proxies for each weight matrix θ within an operator, ranging from S_{param} to S_{meco}. Then we sum the corresponding proxies across all weight matrices in that operator. For instance, the S_{param} for operator o is computed as $S_{param}(o) = \sum_{\forall \theta \in o} S_{param}(\theta)$. Finally, we concatenate these seven different operator-level proxies into a 7-dimensional tensor, defined as $\mathcal{S}_o \in \mathbb{R}^7$ in Eq. (6), which represents the superproxy of operator o.

$$\mathcal{S}_o = [S_{param}(o), S_{flops}(o), \ldots, S_{zico}(o), S_{meco}(o)] \tag{6}$$

This superproxy integrates information from multiple proxies of the operator to estimate the performance of different candidate architectures and discover the optimal architectures within the search space, eliminating the need for supernet training and candidate architecture validation. Notably, we hypothesize that overhead-relevant proxies can serve as indicators of model capacity and expressive ability in the domain of DRS. Intuitively, we interpret that higher values of overhead-relevant proxies correlate with improved predictive performance, while potentially increasing the computational overhead of candidate architectures. To counterbalance this effect, we further incorporate the overhead-irrelevant proxies, which are weakly correlated with model size but remain informative of trainability and predictive performance, as regularizers within the composite loss function (detailed in Sec. 3.5) to stabilize the optimization process. By integrating these proxies, we enable joint optimization that balances both predictive performance and computational efficiency.

3.3 Search Space Generation

Drs.NAS constructs a directed acyclic graph (DAG) as the search space using the superproxy $\mathcal{S}$ computed from the supernet initialization. Following the design in [Zhang et al.(2023)], in each layer of a candidate architecture, Drs.NAS selects one dense operator, one sparse operator, and the corresponding feature fusion operators from a heterogeneous operator set. The search space additionally permits arbitrary connections between any pair of layers.

In particular, for dense operators, we include the fully connected (FC), sigmoid gating [Chen et al.(2019), Wang et al.(2021)], summation, and dot-product [Cheng et al.(2016), Zhang et al.(2024), Naumov et al.(2019)], while for sparse operators we utilize the Embedded FC and transformer [Song et al.(2019), Vaswani et al.(2017)] operators in Drs.NAS. Furthermore, Drs.NAS incorporates two feature fusion strategies, as adopted in prior work [Zhang et al.(2023)], to enhance the feature interaction between dense and sparse outputs. Drs.NAS leverages both dense-to-sparse and sparse-to-dense feature fusion. The dense-to-sparse feature fusion first transforms the dense tensor through FC and concatenates the resulting output with the sparse tensor, while the sparse-to-dense feature fusion projects the sparse tensor into the dense feature space based on Factorization Machine (FM) [Guo et al.(2017), Rendle(2010)] and sums the resulting output with the dense tensor. Together, these operators cover a wide range of computation modules in modern DRS.

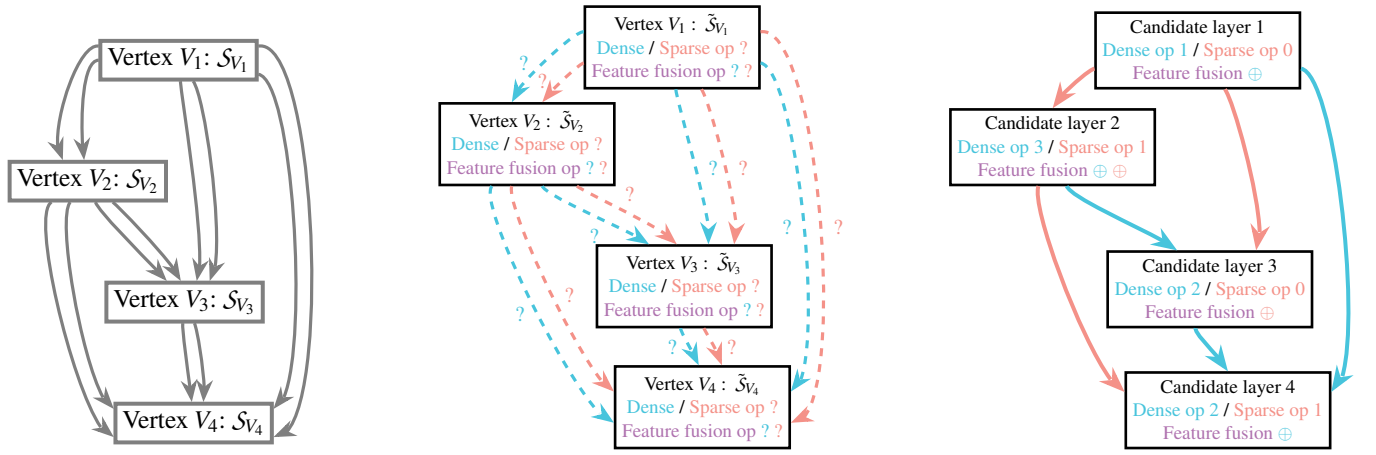
Fig. 2(a) illustrates an example four-layer DAG search space. Formally, the DAG search space is defined as $\mathcal{G} = (\mathcal{V}, \mathcal{E})$. The vertex set $\mathcal{V} = [V_1, \ldots, V_\mathcal{D}]$ contains $\mathcal{D}$ vertices, corresponding to the $\mathcal{D}$ layers in supernet. Each vertex V_i includes p candidate layers at layer i, denoted as $V_i = [v_1, v_2, \ldots, v_p]$, where each candidate layer v_i denotes a specific combination of one dense operator, one sparse operator, and the associated feature fusion.

The superproxy of a candidate layer $\mathcal{S}_{v_i}$ is defined as the element-wise sum of the superproxies of all operators and feature interactions $\mathcal{S}_o$ within that candidate layer: $\mathcal{S}_{v_i} = \sum_{\forall o \in v_i} \mathcal{S}_o$. The superproxy of the vertex $\mathcal{S}_{V_i}$ is then represented as the concatenation of the superproxies of all candidate layers, denoted as $\mathcal{S}_{V_i} = [\mathcal{S}_{v_1}, \mathcal{S}_{v_2}, \ldots, \mathcal{S}_{v_p}]$, where $\mathcal{S}_{V_i} \in \mathbb{R}^{p \times 7}$.

In the meantime, the edge set $\mathcal{E} = [E_1, \ldots, E_\mathcal{D}]$ represents the data flow connections within the supernet, where E_i denotes the set of all candidate incoming edges (in-edges) of the vertex at layer i. The superproxy $\mathcal{S}_e$ for candidate in-edge $e \in E_i$ is computed by the element-wise sum of the superproxy of the source candidate layer $v_s \in V_{j<i}$ and the superproxy of the target candidate layer $v_t \in V_i$ in the DAG search space.

3.4 Search Space Relaxation

Based on the defined DAG search space, Drs.NAS formulates the architecture search problem as candidate architecture sampling over the DAG. The search objective is to discover an optimal candidate architecture by selecting one candidate layer per vertex and a subset of candidate in-edges from the search space. However, directly optimizing over the discrete DAG is non-trivial due to its non-differentiable nature. To address this challenge, Drs.NAS employs continuous relaxation over the DAG search space based on the Gumbel-Softmax [Jang et al.(2016), Maddison et al.(2016)], a strat-



(a) **DAG Search Space** constructed based on superproxies

(b) **Relaxed DAG search space** based on Gumbel-Softmax

(c) **Discretized candidate architecture** from relaxed DAG search space

Figure 2: Illustration of the DAG search space construction, search space relaxation, and sampled candidate architecture in Drs.NAS, utilizing a four-layer architecture as an example

egy widely adopted in differentiable NAS methods [Chang et al.(2019), Xie et al.(2018)], to enable smooth architecture sampling from the discrete DAG.

Formally, let V_i and E_i denote the set of candidate layers and candidate in-edges at layer i. Each candidate layer $v \in V_i$ and candidate in-edge $e \in E_i$ is assigned with a learnable weight parameter, defined as $w_v, w_e \in \mathcal{W}$. The smooth architecture sampling in Drs.NAS includes two components. Firstly, the edge sampling is performed over candidate in-edges E_i iteratively across each layer utilizing Gumbel-Softmax relaxation, as indicated in Eq. (7). Here, the g indicates the Gumbel noise $g \sim Gumbel(0,1)$, and the τ represents the temperature parameter controlling the smoothness of the sampling in the Gumbel-Softmax.

$$\tilde{\mathcal{S}}_E = \sum_{e \in E_i} \frac{\exp\left(\frac{w_e + g_e}{\tau}\right)}{\sum_{e' \in E_i} \exp\left(\frac{w_{e'} + g_{e'}}{\tau}\right)} \mathcal{S}_E \tag{7}$$

Secondly, a similar Gumbel-Softmax relaxation is deployed over candidate layers v_i iteratively across each vertex, and the vertex sampling is performed as Eq. (8). The relaxed search space is illustrated in Fig. 2(b), enabling the joint sampling of the candidate layers and in-edges. Notably, the architecture sampling strategy mainly focuses on the selection of candidate layers and in-edges and ensures valid forward paths without requiring additional graph-level constraints.

$$\tilde{\mathcal{S}}_V = \sum_{v \in V_i} \frac{\exp\left(\frac{w_v + g_v}{\tau}\right)}{\sum_{v' \in V_i} \exp\left(\frac{w_{v'} + g_{v'}}{\tau}\right)} \mathcal{S}_V \tag{8}$$

After the architecture sampling, the candidate architecture refers to $\mathcal{G}_{\mathcal{A}} \subset \mathcal{G}$. The discrete candidate architecture can be obtained through architecture discretization based on the weight parameters $\mathcal{W}$, as illustrated in Fig. 2(c). For vertex discretization, Drs.NAS selects the optimal candidate layer $\tilde{\mathcal{S}}_v$ with the highest candidate layer weight from each vertex V_i, i.e., $argmax_{v \in V_i}(w_v)$. For edge discretization, Drs.NAS sets a fixed threshold d to determine the optimal candidate in-edges and selects the edges with the top d candidate in-edge weights w_e from the candidate in-edges set E_i. Here, we define the threshold $d = 50\%$ to balance the trade-off between model complexity and computational efficiency.

3.5 Gradient-Based Search

To discover the optimal architecture with both predictive performance and computational efficiency, we develop a composite loss function based on the superproxy and define the architecture search loss $\mathcal{L}_{\mathcal{A}}$ that jointly optimizes three objectives, as formulated in Eq. (9). Similar to prior research [Liu et al.(2018), Wu et al.(2019)], we adopt a gradient-based approach to solve this co-optimization problem.

$$\mathcal{L}_{\mathcal{A}} = \lambda_1 * \mathcal{L}_{vertex} + \lambda_2 * \mathcal{L}_{edge} + \lambda_3 * \mathcal{L}_{cost} \tag{9}$$

In Eq. (9), we define the three components in the composite loss function corresponding to the selection of candidate layers, the connection of candidate in-edges, and the computational cost of the candidate architecture, respectively. In particular, $\mathcal{L}_{vertex}$ encourages effective operators in each candidate layer to achieve high performance. $\mathcal{L}_{edge}$ regularizes the data flow of in-edges within the architecture. $\mathcal{L}_{cost}$ controls the computational overhead by penalizing architectures that introduce high computational overhead.

In Drs.NAS, within $\mathcal{L}_{vertex}$ and $\mathcal{L}_{edge}$, we encourage the selection of architectures with high overhead-relevant proxies

to maximize the overall predictive performance, motivated by the insight that overhead-relevant proxies are strongly correlated with predictive performance and trainability [Li et al.(2023)]. However, this strategy may induce increased memory and computational overhead. To counterbalance this effect, we incorporate the overhead-irrelevant proxies as the regularizer to stabilize the optimization process.

Formally, let $\mathcal{V}_{\mathcal{A}}$ and $\mathcal{E}_{\mathcal{A}}$ denote the set of all candidate layers and candidate in-edges within the candidate architecture. $\tilde{\mathcal{S}}^{\mathcal{R}}$ and $\tilde{\mathcal{S}}^{I}$ represent subsets of superproxy $\tilde{\mathcal{S}}$, where $\tilde{\mathcal{S}}^{\mathcal{R}}$ contains the overhead-relevant proxies indexed by $\mathcal{R}$ and $\tilde{\mathcal{S}}^{I}$ includes the overhead-irrelevant proxies indexed by I. The computation of $\mathcal{L}_{vertex}$ is indicated in Eq. (10). In $\mathcal{L}_{vertex}$, we employ the parameter related overhead-relevant proxies $\mathcal{R}_v = \{S_{param}, S_{g_{norm}}, S_{w_{norm}}\}$, while using all the overhead-irrelevant proxies $I_v = \{S_{syn}, S_{zico}, S_{meco}\}$ as regularizer.

$$\mathcal{L}_{vertex} = \sum_{v \in \mathcal{V}_{\mathcal{A}}} \rho_v \left[\tilde{\mathcal{S}}_v^{I_v} - \tilde{\mathcal{S}}_v^{\mathcal{R}_v} \right] \tag{10}$$

Additionally, we compute the $\mathcal{L}_{edge}$ based on the superproxies of the candidate in-edges. Here, we mainly focus on the data flow and gradient flow related proxies in $\mathcal{L}_{edge}$, such as $\mathcal{R}_e = \{S_{g_{norm}}\}$, and overhead-irrelevant proxies $I_e = \{S_{syn}, S_{zico}\}$ as the regularizer in this loss function to optimize the data and gradient propagation within the candidate architecture. The computation of $\mathcal{L}_{edge}$ is denoted as Eq. (11).

$$\mathcal{L}_{edge} = \sum_{e \in \mathcal{E}_{\mathcal{A}}} \rho_e \left[\tilde{\mathcal{S}}_e^{I_e} - \tilde{\mathcal{S}}_e^{\mathcal{R}_e} \right] \tag{11}$$

Last but not least, we define the loss function for the computational overhead, $\mathcal{L}_{cost}$, which integrates both computational and memory usage. We define the $\mathcal{L}_{cost}$ as indicated in Eq. (12), incorporating the #FLOPs and #Params of the candidate architecture. The $degree(\cdot)$ is utilized as a layer-specific normalizer for distinguishing the contribution of different layers. Based on our definition, high #Params encourages the candidate architecture to have better predictive performance and trainability, while high #FLOPs are penalized to promote model computational efficiency. In $\mathcal{L}_{cost}$, we assign equal weight to #FLOPs and #Params, enabling a balance between computational efficiency and predictive performance during the architecture search process.

$$\mathcal{L}_{cost} = \sum_{v \in \mathcal{V}_{\mathcal{A}}} degree(v) \cdot \left[\tilde{\mathcal{S}}_v^{flops} - \tilde{\mathcal{S}}_v^{param} \right] \tag{12}$$

It is worth mentioning that the hyperparameters $\lambda = [\lambda_1, \lambda_2, \lambda_3]$ and $\rho = \{\rho_v, \rho_e\}$ are trainable weighting parameters that dynamically balance the contribution of each component during optimization without manual fine-tuning. To avoid any single loss function from being disregarded, we normalize the learnable parameters λ and ρ via softmax. In Drs.NAS, we utilize a gradient-based optimization strategy to minimize the $L_{\mathcal{A}}$, as illustrated in Algorithm 1, and discover the corresponding optimal architecture.

Algorithm 1 Gradient-based search in Drs.NAS

Input: Given DAG search space $\mathcal{G} = (\mathcal{V}, \mathcal{E})$
Input: Architecture Sampler $Sampler(\mathcal{W}, \rho, \lambda)$
Output: Optimal candidate architecture $\mathcal{G}_{\mathcal{A}} = (\mathcal{V}_{\mathcal{A}}, \mathcal{E}_{\mathcal{A}})$

$\mathcal{L}_{best} \leftarrow \infty$
$\mathcal{W}, \rho, \lambda \leftarrow Initialize(\mathcal{W}, \rho, \lambda)$
for iteration $i = 1$ **to** N **do**
  Sample a candidate architecture ${\mathcal{G}_{\mathcal{A}}}^i$**:**
  ${\mathcal{G}_{\mathcal{A}}}^i \leftarrow Sampler(\mathcal{W}, \rho, \lambda)$
  Compute the composite loss $\mathcal{L}_{\mathcal{A}}$**:**
  $\mathcal{L}_{vertex} \leftarrow \sum_{v \in \mathcal{V}_{\mathcal{A}}} \rho_v \left[\tilde{\mathcal{S}}_v^{I_v} - \tilde{\mathcal{S}}_v^{\mathcal{R}_v} \right]$
  $\mathcal{L}_{edge} \leftarrow \sum_{e \in \mathcal{E}_{\mathcal{A}}} \rho_e \left[\tilde{\mathcal{S}}_e^{I_e} - \tilde{\mathcal{S}}_e^{\mathcal{R}_e} \right]$
  $\mathcal{L}_{cost} \leftarrow \sum_{v \in \mathcal{V}_{\mathcal{A}}} degree(v) \cdot \left[\tilde{\mathcal{S}}_v^{flops} - \tilde{\mathcal{S}}_v^{param} \right]$
  $\mathcal{L}_{\mathcal{A}} \leftarrow \lambda_1 * \mathcal{L}_{vertex} + \lambda_2 * \mathcal{L}_{edge} + \lambda_3 * \mathcal{L}_{cost}$
  if $\mathcal{L}_{\mathcal{A}} < \mathcal{L}_{best}$ **then**
    $\mathcal{L}_{best} \leftarrow \mathcal{L}_{\mathcal{A}}$
    $\mathcal{G}_{\mathcal{A}} \leftarrow {\mathcal{G}_{\mathcal{A}}}^i$
  end if
  Update $Sampler(\mathcal{W}, \rho, \lambda)$ **via gradient descent**
  $\mathcal{W}, \rho, \lambda \leftarrow Adam(\mathcal{W}, \rho, \lambda, \nabla \mathcal{L}_{\mathcal{A}})$
end for
return $\mathcal{G}_{\mathcal{A}}$

4 Experimental Setup

4.1 Evaluation Benchmarks and Metrics

Following prior work [Song et al.(2020), Zhang et al.(2023)], we evaluate Drs.NAS on three representative real-world recommendation system benchmarks, including Criteo [Tien et al.(2014)], Avazu [Wang and Cukierski(2014)], and KDD [Aden and Wang(2012)]. We select these benchmarks for two primary reasons: (1) they are standard public benchmarks for Click-Through Rate (CTR) prediction, widely adopted across both academia and industry research settings; and (2) they are consistently used in prior studies, enabling direct and fair comparisons with existing work. We adopt the standard preprocessing procedures from [Song et al.(2020), Zhang et al.(2023)] to ensure consistency across different models. The statistical information of the three benchmarks is summarized in Tab. 2.

Table 2: Statistics for three representative recommendation system benchmarks, detailing the number of dense and sparse features alongside the total number of samples.

Benchmark	# Dense	# Sparse	# Samples
Criteo	13	26	45,840,617
Avazu	0	23	40,428,967
KDD	3	10	149,639,104

To systematically evaluate the performance and efficiency of Drs.NAS searched architectures, we leverage four widely adopted metrics. For the predictive performance, we evaluate **Log Loss** and **AUC score**. For memory and computational efficiency, we measure the model size **#Params** and computational costs **#FLOPs**. In general, better performance corresponds to lower Log Loss and higher AUC score, while better efficiency is indicated by lower #Params and #FLOPs.

Note that Drs.NAS focuses on neural architecture design, namely searching for effective backbone structures that determine how feature representations are combined and transformed to produce predictions, which is consistent with prior work [Zhang et al.(2023),Zhu et al.(2022)]; the embedding tables are fixed and not part of the search space. In fact, the size of embedding tables is determined by the dataset-specific configurations rather than architectural decisions, because they are used to encode categorical input features. In short, we measure model size and computational cost primarily based on the searched backbone architectures.

4.2 Drs.NAS Implementation

In this study, all experiments are implemented based on PyTorch [Paszke et al.(2019)]. For superproxy computation, we randomly select 2,048 samples (one batch) from the training set and utilize the binary cross-entropy (BCE) loss function to compute the gradient-related proxies, such as $S_{g_{norm}}$ and S_{zico}. Architecture #FLOPs are measured using DeepSpeed [Rasley et al.(2020)] and FvCore [Facebook Research(2022)]. To handle the scale differences among proxies, we apply logarithmic transformations to #Params and #FLOPs and z-score normalization to other proxies in the superproxy. Following previous differentiable NAS studies [Zhu et al.(2022),Liu et al.(2018)], we utilize the Adam optimizer [Kingma and Ba(2014)] with learning rate from {0.02, 0.05, 0.1, 0.2, 0.5}. Due to the ultra-efficient search (minutes on CPU), early stopping is unnecessary, and we run all the architecture search experiments for $N = 10000$ iterations across all benchmarks.

To comprehensively evaluate the scalability of architectures searched by Drs.NAS, we consider different model depths $\mathcal{D} \in \{5,6,7,8,9\}$. In contrast to conventional NAS approaches that typically rely on coarse-grained dimension sets [Song et al.(2020),Zhang et al.(2023),Zhu et al.(2022)], such as sparse integers in {16, 32, 64, 128}, we select hidden dimensions from a fine-grained set of consecutive integers ranging from 4 to 32, resulting in a significantly expanded but more flexible search space. The fine-grained architecture search space allows Drs.NAS to better balance the trade-off between model predictive performance and computational efficiency on different benchmarks.

Furthermore, we conduct the architecture search experiments based on two different search space configurations, following the definition in NASRec [Zhang et al.(2023)]. The "Search space - Small" includes two dense operators (FC and Dot-Product) and one sparse operator (Embedded FC), while the full search space, indicated as "Search space - Full", consists of all four dense operators and two sparse operators mentioned in Sec. 3.3. Both "Search space - Small" and "Search space - Full" search spaces include the dense-to-sparse and sparse-to-dense feature fusion operators.

All the Drs.NAS architecture search processes are conducted on an AMD Ryzen 5975WX CPU. After Drs.NAS produces the optimal DRS architectures, we evaluate these architectures on an NVIDIA A6000 GPU to validate the performance of the resulting models across different benchmark datasets and architecture search configurations.

5 Evaluation

Based on our experimental setup, we investigate the following key research questions (RQs). Specifically, we explore:

- **RQ1**: How much improvement does Drs.NAS provide in the search efficiency and search cost compared with the SOTA NAS baseline approaches [Gao et al.(2021),Song et al.(2020),Zhu et al.(2022),Zhang et al.(2023)]?
- **RQ2**: How much improvement in predictive performance do Drs.NAS searched architectures provide compared with handcrafted [Guo et al.(2017),Song et al.(2019),Naumov et al.(2019)] and NAS baselines across different benchmarks?
- **RQ3**: How much improvement in computational and memory efficiency do Drs.NAS searched architectures provide compared with handcrafted and NAS baselines across different benchmarks?

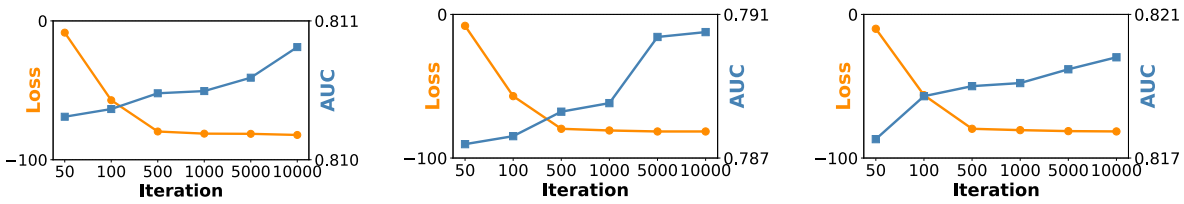
Furthermore, we investigate:

- **RQ4**: How consistent and robust are the Drs.NAS searched architectures under different search space configurations and hyperparameter selections, from both the performance and efficiency perspectives?

5.1 Search Efficiency Evaluation

Answering RQ1: Drs.NAS presents an ultra-efficient search procedure compared with SOTA NAS approaches by providing up to 692× faster NAS process, reducing search cost from GPU-hours to CPU-minutes.

As indicated in Tab. 1, Drs.NAS outperforms all the NAS baselines in the search efficiency. In particular, compared with SOTA NAS approaches including PROFIT [Gao et al.(2021)], AutoCTR [Song et al.(2020)], NAS-CTR [Zhu et al.(2022)], and NASRec [Zhang et al.(2023)], Drs.NAS achieves 461×, 692×, 193× and 230× search time reduction, averaged over three benchmark datasets.



(a) Criteo (b) Avazu (c) KDD

Figure 3: The optimization curve of loss $\mathcal{L}_A$ (●) and the AUC score curve (■) of corresponding candidate architecture during Drs.NAS architecture search process across three benchmarks.

Meanwhile, Drs.NAS requires substantially less computational resources for architecture search. Prior NAS approaches involved the supernet training and fine-tuning procedure of candidate architectures that required significant computation and GPU memory. For instance, NASRec [Zhang et al.(2023)] utilizes over 16 GB of GPU memory for the supernet training and more than 20 GB for the NAS process on the KDD benchmark. In contrast, Drs.NAS performs the NAS process on a commodity CPU only, without GPU acceleration, denoting a dramatic reduction in memory usage and computational cost.

According to our analysis, the improved efficiency of Drs.NAS can be attributed to the superproxy-based performance estimation, which eliminates the requirement for massive training of the supernet and iterative training and validation of candidate architectures [Zhang et al.(2023)]. As a result, Drs.NAS utilizes the superproxies as performance indicators to guide the selection of candidate architectures by minimizing the architecture search loss and significantly reducing the architecture search costs.

Furthermore, we visualize the optimization curve of architecture search loss in Drs.NAS and the corresponding performance of candidate architectures, as illustrated in Fig. 3(a) to Fig. 3(c). Here we select the learning rate of 0.02 and model depth $\mathcal{D} = 9$ in this case study. Six milestone iterations $\{50, 100, 500, 1000, 5000, 10000\}$ are selected to evaluate both the architecture search loss $\mathcal{L}_\mathcal{A}$ and the AUC score of the corresponding candidate architectures. Notably, at early iterations, the AUC scores remain low across different benchmarks, as the initial architectures are randomly sampled from the search space without optimization. As composite loss decreases across optimization steps, the validation AUC of candidate architectures exhibits monotonic improvement. These evaluation results provide potential support for the effectiveness of the gradient-based search in Drs.NAS.

5.2 Model Predictive Performance Evaluation

Answering RQ2: Drs.NAS demonstrates on par or even superior predictive performance compared with the SOTA NAS baselines, across three widely adopted benchmarks.

The performance evaluation of various DRS is presented in Fig. 4(a) to Fig. 4(f). In our experiment, we compare the Log Loss and AUC score of Drs.NAS searched architectures with mainstream handcrafted and SOTA NAS approaches.

In CTR prediction tasks, a 0.001-level improvement in AUC or Log Loss can lead to a substantial impact on the overall performance [Zhu et al.(2022), Song et al.(2020)]. In particular, Drs.NAS searched architectures consistently achieve better predictive performance than the mainstream handcrafted baselines [Guo et al.(2017), Song et al.(2019), Naumov et al.(2019)] and achieve 0.0028, 0.0078, and 0.0046 of averaged Log Loss reduction on the Criteo, Avazu, and KDD benchmark datasets, respectively. Furthermore, across the SOTA NAS approaches [Gao et al.(2021), Song et al.(2020), Zhu et al.(2022), Zhang et al.(2023)], Drs.NAS achieves competitive Log Loss, with 0.0011, 0.0022, and 0.0026 averaged Log Loss reduction on the three benchmarks, respectively.

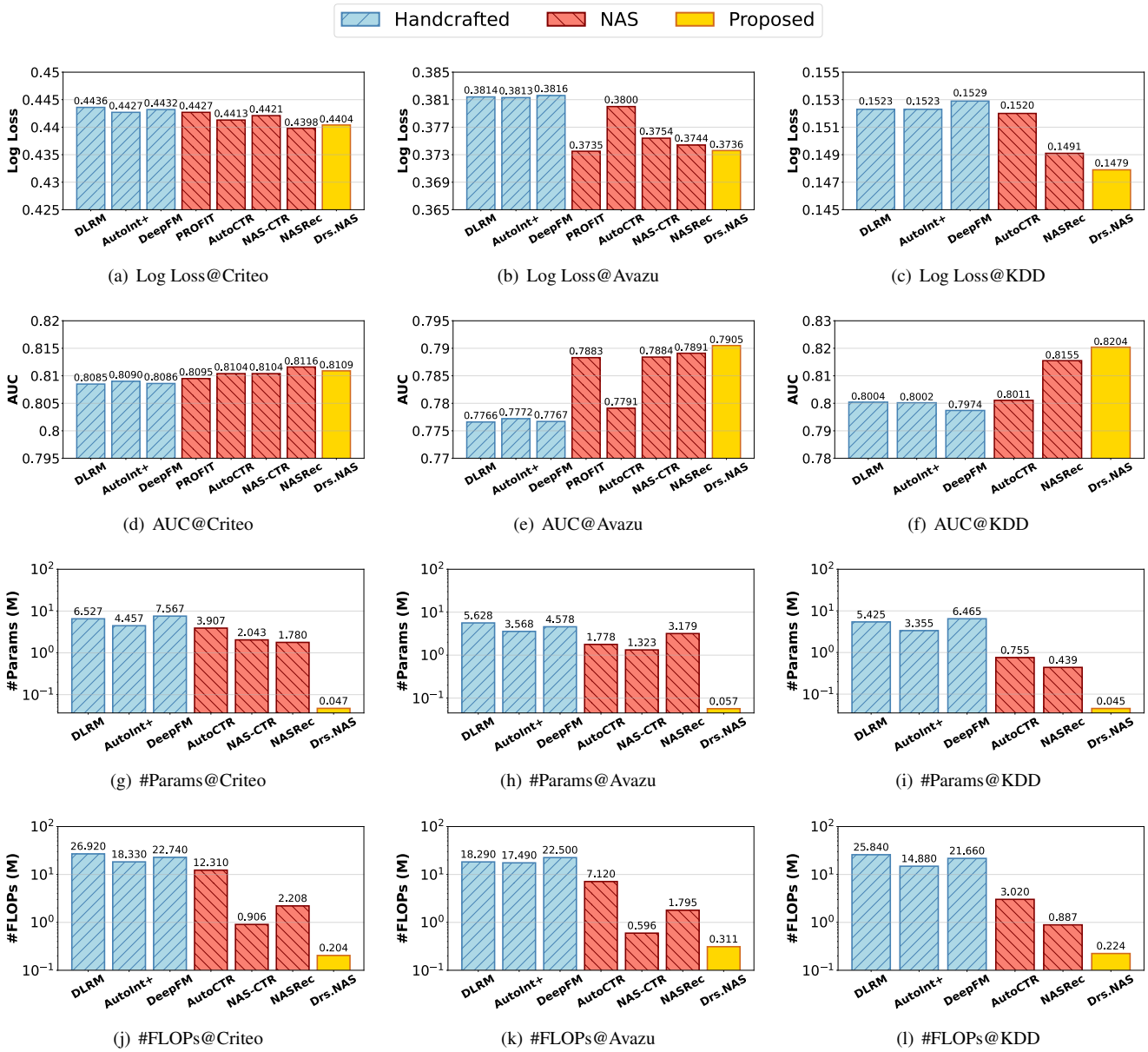
Meanwhile, regarding the AUC score, Drs.NAS achieves better predictive performance than all the mainstream handcrafted baselines by achieving 0.0022, 0.0137, and 0.0211 averaged AUC score improvements on the three benchmark datasets. Furthermore, Drs.NAS searched architectures achieve 0.0004, 0.0043, and 0.0121 of averaged AUC score improvements over the SOTA NAS baselines, on the three benchmarks, respectively. Our experimental results show the effectiveness of Drs.NAS in discovering the high-performing architectures for DRS.

5.3 Model Efficiency Evaluation

Answering RQ3: Drs.NAS searched architectures demonstrate substantial efficiency improvements in both computation and memory compared to SOTA NAS baselines, achieving, on average, 34.9× smaller model size and 14.7× fewer #FLOPs across three benchmarks.

The evaluation results of inference costs are illustrated in Fig. 4(g) to Fig. 4(l). In this section, we compare the #Params and #FLOPs of the optimal architectures with handcrafted and NAS baselines to comprehensively evaluate the memory and computational efficiency of Drs.NAS searched architectures.

Based on the experimental results, Drs.NAS consistently achieves better memory and computational efficiency than the



(a) Log Loss@Criteo (b) Log Loss@Avazu (c) Log Loss@KDD

(d) AUC@Criteo (e) AUC@Avazu (f) AUC@KDD

(g) #Params@Criteo (h) #Params@Avazu (i) #Params@KDD

(j) #FLOPs@Criteo (k) #FLOPs@Avazu (l) #FLOPs@KDD

Figure 4: Evaluation of the optimal architecture searched by Drs.NAS against handcrafted and NAS baselines on three benchmarks, based on Log Loss (↓), AUC score (↑), #Params (↓), and #FLOPs (↓).

baselines across all benchmarks. For the evaluation of model efficiency, we focus primarily on the DRS architectures and exclude the parameters in embedding tables for a fair comparison, following previous works [Song et al.(2020), Zhu et al.(2022), Zhang et al.(2023)]. On average, compared with handcrafted baselines [Guo et al.(2017), Song et al.(2019), Naumov et al.(2019)], Drs.NAS reduces the #Params by 131.6×, 80.5×, and 112.9× on the three benchmarks. Additionally, compared with the SOTA NAS baselines [Gao et al.(2021), Song et al.(2020), Zhu et al.(2022), Zhang et al.(2023)], Drs.NAS achieves 54.8×, 36.7×, and 13.3× averaged #Params reduction across all NAS baselines on Criteo, Avazu, and KDD benchmarks, respectively. In parallel, for computational efficiency, Drs.NAS provides 111.1×, 62.5×, and 92.8× averaged #FLOPs reduction across the handcrafted baselines, and 25.2×, 10.2×, and 8.7× averaged #FLOPs reduction across the NAS baselines, on the three benchmark datasets, respectively.

Our analysis attributes the significant improvement to the cooperation between the fine-grained search space design and the cost-aware optimization. Different from conventional NAS approaches [Song et al.(2020), Zhang et al.(2023)]

that utilize coarse-grained hidden dimension configurations, Drs.NAS employs a low-value range but high-granularity definition based on consecutive integers, enabling an expanded search space with reduced hidden dimension and model size. In the meantime, we incorporate the computational overhead as part of the composite loss function, which allows Drs.NAS to discover optimal architectures with reduced computational overhead.

Table 3: Comparison of **inference time** (μs) for optimal architectures searched by Drs.NAS and SOTA approach NASRec [Zhang et al.(2023)] across three benchmarks and two search space configurations.

Model	Criteo GPU/CPU	Avazu GPU/CPU	KDD GPU/CPU
NASRec - Small	4.96/43.26	3.78/44.15	2.47/53.94
NASRec - Full	8.18/86.36	5.99/69.36	6.19/76.42
Drs.NAS - Small	**4.69/19.72**	**3.44/20.33**	**1.84/18.56**
Drs.NAS - Full	5.21/25.92	3.98/24.91	4.06/33.25

Additionally, we evaluate the inference time of DRS models using Drs.NAS searched architectures based on both GPU and CPU devices, compared with the SOTA NAS baseline [Zhang et al.(2023)]. We select the same model depth, $\mathcal{D} = 7$, for a fair comparison. The inference time evaluation, reported in microseconds (μs), is presented in Tab. 3. We evaluate the inference time per sample, and each measurement is repeated three times to mitigate variance and experimental bias. On average, Drs.NAS searched architectures achieve 60.8% reduction in CPU inference time and 24% inference time reduction on GPU across three benchmarks and two different search space configurations. These results indicate that the decrease in #FLOPs obtained by Drs.NAS can consistently convert to lower inference latency for the DRS deployment on both hardware platforms.

5.4 Ablation Study

Answering RQ4: Drs.NAS demonstrates robustness across diverse architecture search configurations, consistently discovering high-performing and efficient architectures. Different configurations enable distinct design choices and performance-efficiency trade-offs.

To evaluate the configurable components in the Drs.NAS architecture search process, we conduct comprehensive ablation studies to explore the robustness of Drs.NAS, defined by the consistency of the performance and efficiency across different searched architectures. In our ablation study, we select different model depths, learning rates, and search space configurations to systematically investigate the robustness of Drs.NAS searched architectures. Meanwhile, we measure the coefficient of variation (CV) for different performance and efficiency metrics to evaluate the robustness of Drs.NAS.

Takeaway 1: Drs.NAS is robust to model depth and can consistently discover efficient architectures across various model complexities.

We investigate the impact of model depth $\mathcal{D}$. Here we mainly focus on the AUC scores since the #FLOPs and #Params will naturally increase with greater model depth. The evaluation results are illustrated in Fig. 5(a) to Fig. 5(c). We compute the CV of the AUC scores across different model depths. The evaluation results indicate that Drs.NAS searched architectures exhibit the CV of 0.0002, 0.0006, and 0.0004 on various benchmarks, indicating significant robustness in model predictive performance across different model depths.

We can observe that optimal model depth varies across benchmarks. For instance, on the Avazu benchmark, the architecture searched by Drs.NAS achieves the highest averaged AUC at model depth $\mathcal{D} = 9$, and on the Criteo and KDD benchmarks, the optimal depth is $\mathcal{D} = 7$. Notably, even at model depth $\mathcal{D} = 9$, the optimal architecture achieves superior efficiency compared to prior NAS baselines.

Takeaway 2: Lower learning rates during architecture search tend to yield architectures with improved predictive performance.

Meanwhile, we examine the impact of different learning rates within the gradient-based search process. The evaluation results are presented in Fig. 5(d) to Fig. 5(f). Specifically, we observe that the optimal architecture is searched with a lower learning rate, such as 0.02 and 0.05. In particular, we compare the averaged AUC scores of Drs.NAS searched architectures across all three benchmarks, and the lower learning rates (0.02 and 0.05) achieve 0.02%, 0.05%, and 0.03% higher averaged AUC scores than the higher learning rates (0.2 and 0.5).

We evaluate the CV on both #FLOPs and #Params of different optimal architectures, and the inference costs remain stable over varying learning rates. The averaged CVs of #FLOPs and #Params are 0.05 and 0.02 over three benchmarks, which denotes the robustness of Drs.NAS in architecture search. Additionally, we notice that the improvement in performance does not significantly influence the computational or memory efficiency. We hypothesize that higher learning rates may overshoot the sweet spot in the search space, resulting in suboptimal architectures. In contrast, lower learning rates enable a finer search process and discover architectures with improved performance, without introducing extra computational and memory overhead.

Takeaway 3: Different search space configurations induce trade-offs in Drs.NAS searched architectures between predictive performance and computational efficiency.

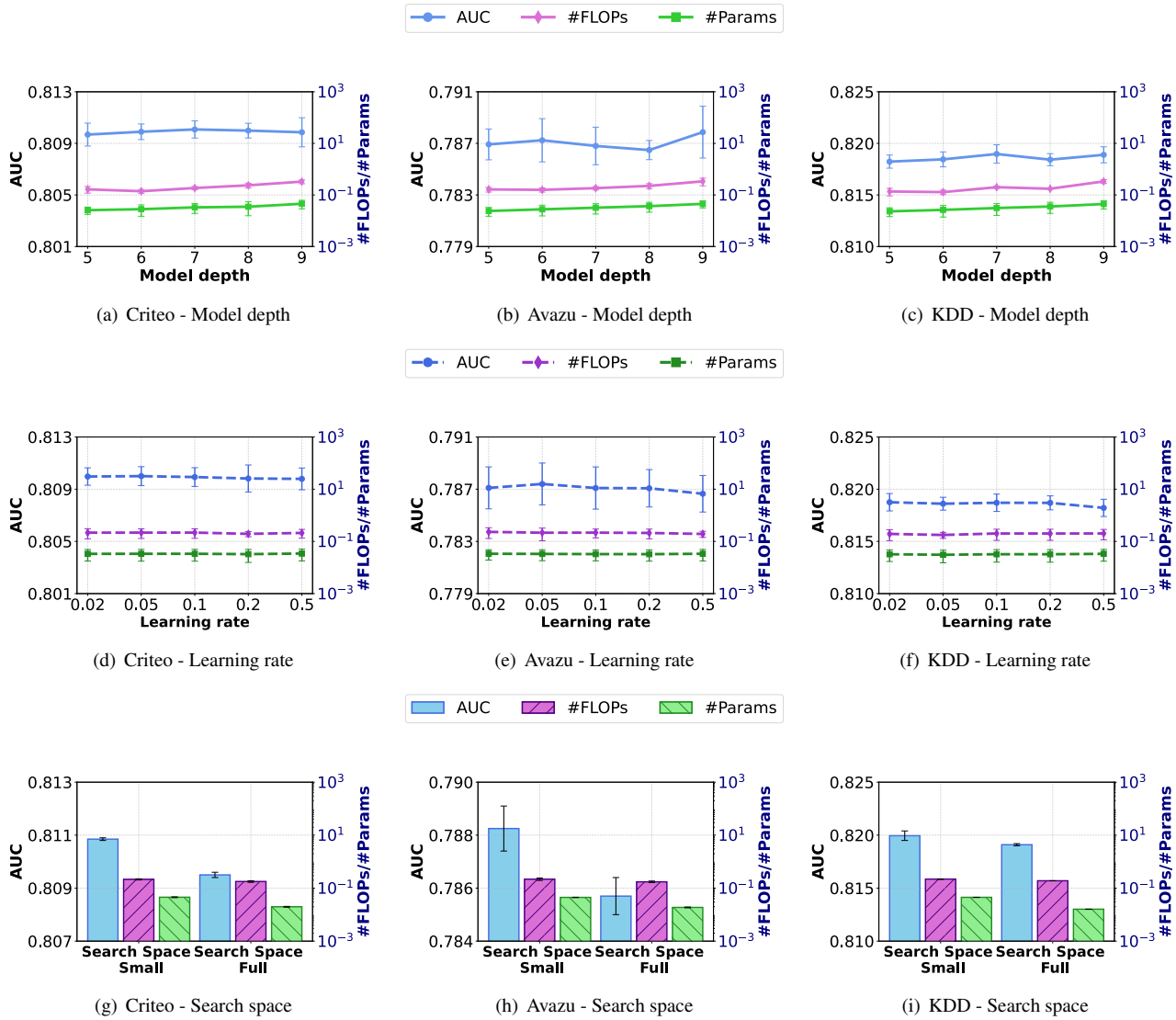
In addition, we evaluate and compare the computational efficiency of Drs.NAS searched architectures based on the "Search space - Small" and "Search space - Full" search space configurations, as discussed in Sec. 4.2. As indicated in Fig. 5(g) to Fig. 5(i), the experimental results show that Drs.NAS searched architectures demonstrate clear trade-offs between performance and efficiency depending on the search

spaces. Specifically, the optimal architectures searched from "Search space - Small" achieve higher AUC scores, and models discovered from "Search space - Full" exhibit reduced #Params and #FLOPs.

In this ablation study, consistent with the NASRec approach, we fix the model depth $\mathcal{D} = 7$ and utilize the lower learning rate in $\{0.02, 0.05\}$. According to our analysis, the architecture searched in "Search space - Small" achieves 0.17%, 0.33%, and 0.1% higher averaged AUC scores, while models from "Search space - Full" search space induce 56%, 58%, and 64% #Params reduction and 17%, 21%, and 14% #FLOPs reduction across the Criteo, Avazu, and KDD benchmarks, respectively. On average, the CVs of AUC scores, #FLOPs, and #Params are 0.0005, 0.05, and 0.02 across different search spaces and varying benchmarks, indicating noticeable robustness of Drs.NAS in architecture search.

We hypothesize that "Search space - Full" includes an expanded operator set, which allows Drs.NAS to bypass the computationally expensive operators, such as FC, leading to more efficient architectures. This high efficiency may trade off the predictive performance, because the overhead-relevant proxies, such as #Params and #FLOPs, are correlated with the model predictive performance [Li et al.(2023), Lin et al.(2020)]. This phenomenon can potentially support the effectiveness of Drs.NAS on balancing the performance and efficiency of candidate architectures.

Furthermore, the searched architectures with higher #Params or #FLOPs, when used as the sole selection crite-



(a) Criteo - Model depth (b) Avazu - Model depth (c) KDD - Model depth

(d) Criteo - Learning rate (e) Avazu - Learning rate (f) KDD - Learning rate

(g) Criteo - Search space (h) Avazu - Search space (i) KDD - Search space

Figure 5: Ablation study on the (1) model depth, (2) learning rate selection, and (3) search space configuration in Drs.NAS, based on averaged AUC score, #FLOPs, and #Params across different benchmarks.

rion, do not consistently achieve better predictive performance across different model depths, learning rates, and search space configurations. This is consistent with prior findings [Huang et al.(2024), Cortês et al.(2025)] that individual proxies, such as #Params and #FLOPs, are insufficient to indicate model performance reliably. More importantly, this observation validates the superproxy rather than relying on a single proxy alone as the performance indicator. In particular, the consistent performance and efficiency of Drs.NAS searched architectures across all configurations indicate that the ensemble design of the superproxy captures more reliable optimization signals than individual proxies.

6 Discussion

The key idea of Drs.NAS is the superproxy, which comprises seven zero-cost proxies and serves as a composite indicator for predictive performance and architectural efficiency. While extensively validated in the DRS context, it is unclear how the strategy scales to different model families and application domains (e.g., LLMs) [Abdelfattah et al.(2021), Yang et al.(2025), Cortês et al.(2025)]. An important direction of our future work is to systematically characterize the boundary conditions under which Drs.NAS remains reliable through evaluation across more diverse search spaces and cross-domain applications, enabling a comprehensive evaluation of the generalizability and robustness of Drs.NAS.

7 Conclusion

In this work, we presented Drs.NAS, an ultra-efficient NAS framework for DRS. Unlike conventional NAS approaches that rely on computationally expensive training and validation during search, Drs.NAS introduces the novel *superproxy* metric, enabling rapid architecture exploration within minutes on CPUs. By integrating superproxy into a composite loss function, Drs.NAS co-optimizes predictive performance and efficiency and produces architectures that are high-performing, computation- and memory-efficient. Extensive evaluations on widely used recommendation benchmarks demonstrate that Drs.NAS achieves drastic efficiency improvements—reducing search costs by nearly three orders of magnitude and lowering model size and FLOPs by averages of 34.9× and 14.7×, respectively, compared with SOTA NAS baselines—while delivering on par or superior predictive performance compared to SOTA NAS and mainstream handcrafted baselines.

References

[Abdelfattah et al.(2021)] Mohamed S Abdelfattah, Abhinav Mehrotra, Łukasz Dudziak, and Nicholas D Lane. 2021. Zero-cost proxies for lightweight NAS. *arXiv preprint arXiv:2101.08134* (2021).

[Aden and Wang(2012)] Aden and Yi Wang. 2012. KDD Cup 2012, Track 2. https://kaggle.com/competitions/kddcup2012-track2. Kaggle.

[Berger et al.(2018)] Daniel S Berger, Benjamin Berg, Timothy Zhu, Siddhartha Sen, and Mor Harchol-Balter. 2018. {RobinHood}: Tail Latency Aware Caching–Dynamic Reallocation from {Cache-Rich} to {Cache-Poor}. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. 195–212.

[Cai et al.(2019)] Han Cai, Chuang Gan, Tianzhe Wang, Zhekai Zhang, and Song Han. 2019. Once-for-all: Train one network and specialize it for efficient deployment. *arXiv preprint arXiv:1908.09791* (2019).

[Cai et al.(2018)] Han Cai, Ligeng Zhu, and Song Han. 2018. Proxylessnas: Direct neural architecture search on target task and hardware. *arXiv preprint arXiv:1812.00332* (2018).

[Chang et al.(2019)] Jianlong Chang et al. 2019. Data: Differentiable architecture approximation. *Advances in Neural Information Processing Systems* 32 (2019).

[Chen et al.(2024)] Hao Chen, Yuanchen Bei, Qijie Shen, Yue Xu, Sheng Zhou, Wenbing Huang, Feiran Huang, Senzhang Wang, and Xiao Huang. 2024. Macro graph neural networks for online billion-scale recommender systems. In *Proceedings of the ACM web conference 2024*. 3598–3608.

[Chen et al.(2019)] Qiwei Chen, Huan Zhao, Wei Li, Pipei Huang, and Wenwu Ou. 2019. Behavior sequence transformer for e-commerce recommendation in alibaba. In *Proceedings of the 1st international workshop on deep learning practice for high-dimensional sparse data*. 1–4.

[Cheng et al.(2016)] Heng-Tze Cheng, Levent Koc, Jeremiah Harmsen, Tal Shaked, Tushar Chandra, Hrishi Aradhye, Glen Anderson, Greg Corrado, Wei Chai, Mustafa Ispir, et al. 2016. Wide & deep learning for recommender systems. In *Proceedings of the 1st workshop on deep learning for recommender systems*. 7–10.

[Cortês et al.(2025)] Gabriel Cortês, Nuno Lourenço, Paolo Romano, and Penousal Machado. 2025. Greenfactory: Ensembling zero-cost proxies to estimate performance of neural networks. *arXiv preprint arXiv:2505.09344* (2025).

[Facebook Research(2022)] Facebook Research. 2022. fvcore. https://github.com/facebookresearch/fvcore. Accessed: 2025-06-28.

[Gao et al.(2021)] Chen Gao, Yinfeng Li, Quanming Yao, Depeng Jin, and Yong Li. 2021. Progressive feature interaction search for deep sparse network. *Advances in Neural Information Processing Systems* 34 (2021), 392–403.

[Gao et al.(2025)] Jianhua Gao, Zeming Liu, Yizhuo Wang, and Weixing Ji. 2025. RaNAS: Resource-aware neural architecture search for edge computing. *ACM Transactions on Architecture and Code Optimization* 22, 1 (2025), 1–18.

[Guo et al.(2017)] Huifeng Guo, Ruiming Tang, Yunming Ye, Zhenguo Li, and Xiuqiang He. 2017. DeepFM: a factorization-machine based neural network for CTR prediction. *arXiv preprint arXiv:1703.04247* (2017).

[Gupta et al.(2020)] Udit Gupta, Carole-Jean Wu, Xiaodong Wang, Maxim Naumov, Brandon Reagen, David Brooks, Bradford Cottel, Kim Hazelwood, Mark Hempstead, Bill Jia, et al. 2020. The architectural implications of facebook's dnn-based personalized recommendation. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 488–501.

[Huang et al.(2024)] Yi-Cheng Huang, Wei-Hua Li, Chih-Han Tsou, Jun-Cheng Chen, and Chu-Song Chen. 2024. UP-NAS: Unified Proxy for Neural Architecture Search. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 1675–1684.

[Jang et al.(2016)] Eric Jang, Shixiang Gu, and Ben Poole. 2016. Categorical reparameterization with gumbel-softmax. *arXiv preprint arXiv:1611.01144* (2016).

[Jiang et al.(2023)] Tangyu Jiang, Haodi Wang, and Rongfang Bie. 2023. Meco: zero-shot NAS with one data and single forward pass via minimum eigenvalue of correlation. *Advances in Neural Information Processing Systems* 36 (2023), 61020–61047.

[Kingma and Ba(2014)] Diederik P Kingma and Jimmy Ba. 2014. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980* (2014).

[Li et al.(2023)] Guihong Li, Yuedong Yang, Kartikeya Bhardwaj, and Radu Marculescu. 2023. Zico: Zero-shot nas via inverse coefficient of variation on gradients. *arXiv preprint arXiv:2301.11300* (2023).

[Lin et al.(2020)] Ji Lin et al. 2020. Mcunet: Tiny deep learning on iot devices. *Advances in neural information processing systems* 33 (2020), 11711–11722.

[Liu et al.(2018)] Hanxiao Liu, Karen Simonyan, and Yiming Yang. 2018. Darts: Differentiable architecture search. *arXiv preprint arXiv:1806.09055* (2018).

[Lukasik et al.(2025)] Jovita Lukasik, Michael Moeller, and Margret Keuper. 2025. An evaluation of zero-cost proxies-from neural architecture performance prediction to model robustness. *International Journal of Computer Vision* 133, 5 (2025), 2635–2652.

[Maddison et al.(2016)] Chris J Maddison, Andriy Mnih, and Yee Whye Teh. 2016. The concrete distribution: A continuous relaxation of discrete random variables. *arXiv preprint arXiv:1611.00712* (2016).

[Mellor et al.(2021)] Joe Mellor, Jack Turner, Amos Storkey, and Elliot J Crowley. 2021. Neural architecture search without training. In *International conference on machine learning*. PMLR, 7588–7598.

[Naumov et al.(2019)] Maxim Naumov, Dheevatsa Mudigere, Hao-Jun Michael Shi, Jianyu Huang, Narayanan Sundaraman, Jongsoo Park, Xiaodong Wang, Udit Gupta, Carole-Jean Wu, Alisson G Azzolini, et al. 2019. Deep learning recommendation model for personalization and recommendation systems. *arXiv preprint arXiv:1906.00091* (2019).

[Paszke et al.(2019)] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. 2019. Pytorch: An imperative style, high-performance deep learning library. *Advances in neural information processing systems* 32 (2019).

[Rasley et al.(2020)] Jeff Rasley, Samyam Rajbhandari, Olatunji Ruwase, and Yuxiong He. 2020. Deepspeed: System optimizations enable training deep learning models with over 100 billion parameters. In *Proceedings of the 26th ACM SIGKDD international conference on knowledge discovery & data mining*. 3505–3506.

[Rendle(2010)] Steffen Rendle. 2010. Factorization machines. In *2010 IEEE International conference on data mining*. IEEE, 995–1000.

[Song et al.(2020)] Qingquan Song et al. 2020. Towards automated neural interaction discovery for click-through rate prediction. In *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*. 945–955.

[Song et al.(2019)] Weiping Song, Chence Shi, Zhiping Xiao, Zhijian Duan, Yewen Xu, Ming Zhang, and Jian Tang. 2019. Autoint: Automatic feature interaction learning via self-attentive neural networks. In *Proceedings of the 28th ACM international conference on information and knowledge management*. 1161–1170.

[Tanaka et al.(2020)] Hidenori Tanaka, Daniel Kunin, Daniel L Yamins, and Surya Ganguli. 2020. Pruning

neural networks without any data by iteratively conserving synaptic flow. *Advances in neural information processing systems* 33 (2020), 6377–6389.

[Tien et al.(2014)] Jean-Baptiste Tien, joycenv, and Olivier Chapelle. 2014. Display Advertising Challenge. https://kaggle.com/competitions/criteo-display-ad-challenge. Kaggle.

[Vaswani et al.(2017)] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. *Advances in neural information processing systems* 30 (2017).

[Wang et al.(2021)] Ruoxi Wang, Rakesh Shivanna, Derek Cheng, Sagar Jain, Dong Lin, Lichan Hong, and Ed Chi. 2021. Dcn v2: Improved deep & cross network and practical lessons for web-scale learning to rank systems. In *Proceedings of the web conference 2021*. 1785–1797.

[Wang and Cukierski(2014)] Steve Wang and Will Cukierski. 2014. Click-Through Rate Prediction. https://kaggle.com/competitions/avazu-ctr-prediction. Kaggle.

[Wen et al.(2024)] Wei Wen, Kuang-Hung Liu, Igor Fedorov, Xin Zhang, Hang Yin, Weiwei Chu, Kaveh Hassani, Mengying Sun, Jiang Liu, Xu Wang, et al. 2024. Rankitect: Ranking architecture search battling world-class engineers at meta scale. In *Companion Proceedings of the ACM Web Conference 2024*. 73–82.

[Wu et al.(2019)] Bichen Wu, Xiaoliang Dai, Peizhao Zhang, Yanghan Wang, Fei Sun, Yiming Wu, Yuandong Tian, Peter Vajda, Yangqing Jia, and Kurt Keutzer. 2019. Fbnet: Hardware-aware efficient convnet design via differentiable neural architecture search. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*. 10734–10742.

[Xie et al.(2018)] Sirui Xie, Hehui Zheng, Chunxiao Liu, and Liang Lin. 2018. SNAS: stochastic neural architecture search. *arXiv preprint arXiv:1812.09926* (2018).

[Yang et al.(2025)] Yeming Yang, Qingling Zhu, Jianping Luo, Ka-Chun Wong, Qiuzhen Lin, and Jianqiang Li. 2025. Trnas: A training-free robust neural architecture search. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*. 2336–2345.

[Zhang et al.(2019)] Shuai Zhang, Lina Yao, Aixin Sun, and Yi Tay. 2019. Deep learning based recommender system: A survey and new perspectives. *ACM computing surveys (CSUR)* 52, 1 (2019), 1–38.

[Zhang et al.(2023)] Tunhou Zhang, Dehua Cheng, Yuchen He, Zhengxing Chen, Xiaoliang Dai, Liang Xiong, Feng Yan, Hai Li, Yiran Chen, and Wei Wen. 2023. NASRec: weight sharing neural architecture search for recommender systems. In *Proceedings of the ACM Web Conference 2023*. 1199–1207.

[Zhang et al.(2024)] Tunhou Zhang, Wei Wen, Igor Fedorov, Xi Liu, Buyun Zhang, Fangqiu Han, Wen-Yen Chen, Yiping Han, Feng Yan, Hai Li, et al. 2024. DistDNAS: Search Efficient Feature Interactions within 2 Hours. In *2024 IEEE International Conference on Big Data (BigData)*. IEEE, 1492–1499.

[Zhou et al.(2024)] Qinqin Zhou, Kekai Sheng, Xiawu Zheng, Ke Li, Yonghong Tian, Jie Chen, and Rongrong Ji. 2024. Training-free transformer architecture search with zero-cost proxy guided evolution. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 46, 10 (2024), 6525–6541.

[Zhu et al.(2022)] Guanghui Zhu, Feng Cheng, Defu Lian, Chunfeng Yuan, and Yihua Huang. 2022. NAS-CTR: efficient neural architecture search for click-through rate prediction. In *Proceedings of the 45th International ACM SIGIR Conference on Research and Development in Information Retrieval*. 332–342.

ARTIFACT EVALUATED
USENIX
AVAILABLE

SMARTTalk: Teaching SMART Logs to Talk to LLMs

Mayur Akewar[+] Dongsheng Luo[+] Sandeep Madireddy[*] Janki Bhimani[+]

[+]*Florida International University*
[*]*Argonne National Laboratory*

Abstract

SMART attributes are the main telemetry for monitoring Solid State Drives (SSDs) and predicting failures in large fleets, but existing methods rely on heavy feature engineering and large supervised pipelines that must be retrained as hardware or workloads change and that compress rich temporal behavior into opaque numeric scores. Large Language Models (LLMs) offer structured reasoning and explanations, yet perform poorly on raw SMART logs because the histories are long and multivariate, their temporal inductive bias is weak, and they hallucinate on numeric inputs. We present **SMARTTalk**, a new systems architecture that introduces a representation layer for device telemetry. SMARTTalk converts each n day SMART window into a sequence of symbolic trend tokens that an LLM can reliably reason over. The system separates numerical trend extraction from language based reasoning through three stages. (1) It slices n-day SMART windows into short temporal patches and encodes them with lightweight Convolutional Neural Network (CNN), then it clusters the resulting embeddings to form compact libraries of attribute-level and cross-attribute temporal patterns; (2) It converts each pattern into concise, human-readable text tokens that are stable across drives and over time; and (3) It feeds these pattern summaries to an LLM with chain-of-thought prompting, augmented with an online pattern memory that detects and incorporates previously unseen behaviors without retraining. By reasoning in natural language, SMARTTalk gives transparent explanations and interactive workflows.

Evaluations on production datacenter SSD traces, across both open-source and proprietary LLMs, show that SMARTTalk delivers roughly 50× higher $F_{0.5}$ than Raw-LLM, about 4× higher than the Heuristic-LLM, and approximately 25% more accurate health classification than existing SMART-based methods, while achieving time-to-failure estimates with bucketed MAE near 10 days. SMARTTalk's natural-language outputs are rated highly by LLM-as-judge, with explanation and recommendation scores around 4.5 out of 5 and perturbation robustness above 80%, making the system operator friendly and ready for the deployment.

1 Introduction

SMART (Self Monitoring, Analysis and Reporting Technology) attributes [2] are the primary telemetry for monitoring the health of Solid State Drives (SSDs) in production datacenters [5, 7, 8, 17, 22–25, 30, 36, 38]. Each drive reports daily counters such as reallocated sectors, pending sectors, and media errors that evolve over time. These long multivariate histories allow operators to identify at-risk drives before failure, enabling proactive replacement and minimizing downtime.

A large body of work converts SMART logs into predictive models. Classical approaches use summary statistics and hand crafted temporal features fed into supervised learners such as random forests and gradient boosted trees [7, 8, 10, 22, 37]. Subsequent work introduces sequence models including autoencoders, LSTMs, and temporal convolutional networks to capture richer temporal behavior [12, 19, 38]. Recent industrial systems such as MVTRF [40] and MSFRD [41] incorporate more elaborate feature engineering, multi view statistics, and mutation based temporal encodings to improve both failure prediction and diagnosis. We discuss the existing body of work in detail in Section 2.4 and 2.5.

Despite these advances, existing methods face three key limitations. First, they rely on heavy feature engineering and large labeled datasets, particularly for rare failures and emerging drive models. Second, models must be retrained to handle firmware, workload, or hardware changes. Third, most methods reduce complex temporal behavior to numeric scores or class labels, providing limited interpretability and actionable guidance for operators.

Large Language Models (LLMs) offer an opportunity to unify prediction, explanation, and interaction. They excel at pattern abstraction, multi step reasoning, and natural language explanation [16, 21, 27–29, 31, 33, 35, 42]. However, directly prompting LLMs with long numeric SMART traces is ineffective. Raw tables obscure temporal structure, overwhelm token budgets, and cause models to hallucinate trends that are not present [18]. This is one of the unavoidable barrier of using LLM that motivates our architecture. As discussed in

Section 2, key signals are local temporal patterns, slow drifts, spikes, bursts, and cross-attribute correlations that standard LLMs do not readily interpret.

Rare failures, evolving patterns, and long multi-attribute histories push classical models toward brittle feature engineering, sequence models toward failure under class imbalance and continual retraining, and LLMs toward hallucinations on raw telemetry, so prior architectures cannot solve this problem. What is needed is a new representation layer rather than another model tweak. These observations motivate our architecture that combines reliable temporal pattern extraction with LLM-based reasoning. Our core insight is to translate an n-day SMART window into a compact, human-readable event language that: (1) Preserves essential temporal structure (local patterns, slow drifts, spikes, bursts). (2) Captures attribute-wise and cross-attribute correlations. (3) Expands naturally over time as new failure modes emerge. (4) Exposes the right "interaction surface" for LLM reasoning, expressive enough for diagnosis, concise enough for token budgets, and structured enough to avoid hallucinated trends.

SMARTTalk is a system that enables LLMs to infer drive health, time to failure, likely causes, and recommended operator actions without heavy feature engineering or large labeled datasets. The design centers on three mechanisms. First, SMARTTalk introduces a temporal patch to pattern to phrase representation: each SMART history is sliced into short patches, embedded, mapped to a learned vocabulary of temporal patterns, and then rendered as concise natural language trend descriptions that LLMs can reliably interpret. Second, SMARTTalk maintains an online pattern memory for novelty tracking, allowing the system to detect patches whose embeddings exceed a learned novelty threshold, accumulate these previously unseen behaviors, recluster them into new temporal patterns, and expand the pattern library as new failure modes appear. This helps SMARTTalk handle rare or evolving failure modes. Third, SMARTTalk performs LLM reasoning over a structured sequence of semantic events; the LLM consumes this event sequence using a small in-context prompt and a Chain of Thought scaffold [34], enabling accurate status inference, time to failure prediction, cause identification, and actionable recommendations.

The design of SMARTTalk is guided by the following research questions (RQ): **(RQ1:)** How can LLMs analyze SMART logs to predict drive health and time-to-failure accurately without heavy feature engineering or large labeled datasets? **(RQ2:)** Does the patch→pattern→phrase representation reliably prevent trend hallucination and capture temporal structure? **(RQ3:)** How does an online pattern-memory improve robustness to rare and emerging failures compared to static encoders? **(RQ4:)** Are SMARTTalk's explanations ("why the drive is at risk") and recommendations ("what to do next") align with operator needs?

We evaluate SMARTTalk on production traces from Alibaba (MB1 and MB2 SSD models). Across five LLM backbones, SMARTTalk raises the average $F_{0.5}$ score for failure status prediction roughly 50× times higher than Raw-LLM, about 4× times higher than the Heuristic-LLM baseline, and approximately 25% improvement over the average of existing SMART-based methods and recent time-series SOTA models (Table 5). For drives that are correctly flagged at risk, the same backbones achieve TTF bucket macro-F_1 around 0.6 with bucketed MAE near 10 days and more than half of failures landing within ±5 days of the predicted bucket (Table 6). SMARTTalk's natural-language outputs are also rated highly by an independent LLM-as-judge, with explanation and recommendation scores around 4.4–4.6 out of 5 and perturbation robustness metrics above 80% (Table 7). Together, these results suggest that SMARTTalk is operator friendly and can substantially reduce the extra work spent deciphering raw SMART logs and triaging spurious alerts, while still providing timely, precise warnings and concrete actions.

2 Understanding SMART Logs

Our goal is not just to predict failures, but to make raw SMART logs *talk* to LLMs in a way that preserves the underlying behavior of the device to support qualitative decision making. Before designing SMARTTalk, we first inspect the data to understand what kinds of temporal patterns appear in SMART attributes and how a human operator would naturally describe them. These observations directly motivate our patching, pattern, and text-generation design.

2.1 SMART Attributes and Their Semantics

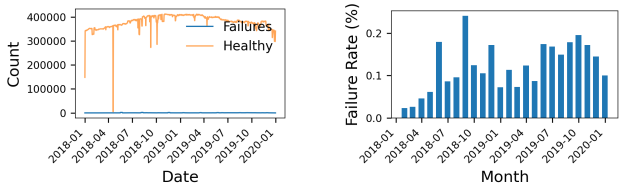
SMART exposes dozens of device–specific counters and indicators [17]. In this work we focus on a subset of 19 *raw* attributes that are both widely used in prior studies and relatively interpretable across models [17]. Table 1 summarizes these fields and the direction in which they are considered "healthy" (e.g., error counts ideally remain low, spare capacity ideally remains high). For a human operator, these attributes naturally suggest qualitative descriptions such as: "this error counter stayed flat for months and then spiked," "this wear indicator is slowly drifting down," or "this attribute is noisy but bounded." SMARTTalk aims to recover exactly such phrases automatically and feed them to an LLM.

The semantic structure of SMART attributes motivates three design choices in SMARTTalk. First, because different attributes reflect health through distinct patterns (e.g., slow wear trends versus sudden error spikes), we require attribute-specific patch encoders rather than a single global model. Second, since operators naturally describe these behaviors linguistically, SMARTTalk must convert temporal shapes into short phrases that mirror operator vocabulary. Third, because attributes interact during many failures, these phrase-level descriptions must later be composed to express cross-attribute events.

Table 1: Selected raw SMART attributes grouped by trend.

Ideal	SMART attributes
Low	r_5: Reallocated Sector Count; r_174: Unexpected Power Loss Count; r_175: Power Loss Protection Failure; r_177: Wear Range Delta; r_181: Program Failed Count; r_182: Erase Failed Count; r_183: SATA Downshift Error Count; r_184: End-to-End Errors; r_187: Reported Uncorrectable Errors; r_197: Current Pending Sector Count; r_199: UltraDMA CRC Error Count.
High	r_173: Wear Leveling Count; r_180: Available Reserved Blocks; r_195: Hardware ECC Recovered; r_233: Media Wearout Indicator.
Monitor	r_9: Power On Hours; r_12: Power Cycle Count; r_241: Number of Blocks Written; r_242: Number of Blocks Read.

2.2 Dataset and High-Level Behavior



(a) Healthy vs. failed counts. (b) Monthly failure rate.

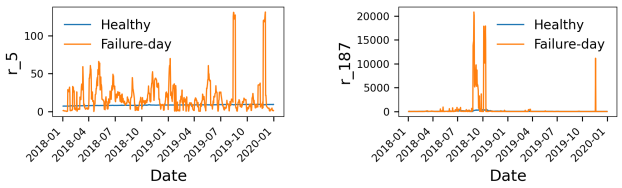
Figure 1: Behavior of the Alibaba SMART dataset.

We use a publicly released dataset of SSD SMART logs from Alibaba Cloud [1, 37]. The dataset spans two years (2018–2019) and covers six SSD models (`MA1`, `MA2`, `MB1`, `MB2`, `MC1`, and `MC2`), each monitored daily with up to 105 SMART attributes. We keep the 19 raw attributes from Table 1 and treat each drive's history as a multivariate daily time series. We focus on the **`MB1`** and **`MB2`** models in our experiments because they have very few missing values for each of the 19 attributes and are also used by prior work for evaluation making it straightforward to compare.

Figure 1 summarize several aggregate trends. Figure 1a shows the total number of healthy and failed drives over time, highlighting the extreme imbalance between the two classes. Figure 1b plots the monthly failure rate, which exhibits a slowly changing baseline with occasional spikes. The long histories and extreme class imbalance observed in the dataset suggest that global features across months or years are ineffective. This motivates our use of short, localized temporal patches that isolate informative behavior. The heterogeneous behavior across SSD models further argues for a learned pattern vocabulary rather than fixed, hand-engineered features. Finally, because failures evolve slowly with occasional bursts, the system must track patterns over time and allow new patterns to emerge, motivating use of online pattern memory.

2.3 Attribute-Level Patterns and "How a Human Would Say It"

To understand what kinds of temporal shapes an LLM should see, we zoom in on individual attributes. Figure 2 shows representative trajectories for two important SMART attributes: Reallocated Sector Count (`r_5`) and Reported Uncorrectable Errors (`r_187`). For each attribute we plot typical traces from days labeled healthy and from days that belong to drives that eventually fail. Two observations are important for designing SMARTTalk:



(a) Reallocated Sector Count. (b) Uncorrectable Errors.

Figure 2: Temporal patterns for two SMART attributes.

Attributes express risk in different ways. `r_5` shows noisy, slowly increasing behaviour with repeated medium-sized spikes, while `r_187` stays flat for long periods and then exhibits sudden, extreme jumps. A human would describe these as "gradual degradation with bursts" versus "quiet then a sharp spike". We want the LLM to see precisely such phrases, not just raw numbers.

Informative structure is local. In both plots, the informative changes happen over tens of days, not over the entire multi-year history. Global features such as a single slope over n days cannot capture whether the last week was stable, noisy, or spiky.

From an LLM's perspective, feeding a long list of daily numbers (e.g., "0, 0, 1, 3, 4, 0, . . . ") is a poor interface: the model must implicitly rediscover all the relevant temporal shapes. Instead, we want to convert each local segment of the time series into a short description that already encodes how an operator would talk about it. These attribute-level observations lead directly to SMARTTalk's patch–pattern–phrase pipeline. The fact that informative structure is local motivates our 5–10 day patch granularity. The diversity of shapes (flat, bursty, drifting) motivates learning a pattern library rather than relying on canned statistical features. The operator-like descriptions motivate generating compact natural language summaries for each patch so that the LLM receives structured, semantically meaningful events instead of raw numbers.

2.4 Challenges for Making SMART "Talk" and How SMARTTalk Responds

These observations reveal three challenges in turning SMART logs into LLM-friendly input:

Multivariate, long histories. Each drive exposes many attributes over years of daily logs, so listing all values overwhelms the LLM and hides the critical behavior.

Attribute and model-specific patterns. As seen in Figures 1 and 2, different attributes and SSD models exhibit distinct

temporal shapes (flat, slowly drifting, bursty, oscillatory). It is challenging to accurately capture the relevance and pattern (as discussed in Section 2.3) of these shapes without requiring in-depth feature engineering for each attribute.

Need for human-like diagnostic thinking. Operators naturally think in terms of phrases like "stable low", "gradual increase", "noisy high", or "repeated spikes in the last 10 days" to clearly understand the cumulative issue to make informed decision. To make the LLM's to be able to clearly understand the issue and reason transparently, we want SMART logs to be expressed in the same style. *SMARTTalk is designed to address above challenges.*

In the next subsection, we place SMARTTalk in the context of existing SMART-based modeling approaches and how they handle the temporal trends.

Table 2: Qualitative comparison of SMART modeling approaches. ✓ = provided; △ = partially or indirectly provided; "–" = not a primary goal. Abbreviations: **TS** = time series aware, **FE** = heavy feature engineering, **LD** = labeled data need, **EP** = evolving patterns, **OF** = operator friendly, **EX** = explanations, **TXT** = text level representation and interaction.

Method	TS	FE	LD	EP	OF	TTF	EX	TXT
RF [5]	✓	–	High	–	△	–	△	–
NN [5]	✓	–	High	–	△	–	△	–
EC [10]	✓	✓	High	–	△	–	△	–
AE [9]	✓	–	Low	△	△	–	△	–
LSTM [19]	✓	–	High	–	△	–	–	–
MVTRF [40]	✓	✓	High	△	△	✓	△	–
MSFRD [41]	✓	✓	High	△	△	–	△	–
SMARTTalk	✓	△	Low	✓	✓	✓	✓	✓

2.5 Existing SMART Models and Why SMARTTalk

Table 2 summarizes the major classes of SMART-based prediction methods. Prior work largely falls into three categories.

Classical supervised models (RF, NN). Early systems treat each SMART snapshot as a feature vector and train models such as Random Forests and feed-forward NNs [5]. These methods need to be trained on large amount of data to predict accurately, operate solely on numerical features, and provide little operator-oriented insight beyond feature importance or binary risk flags.

Unsupervised and sequence models (AE, LSTM). Autoencoders [9] learn low-dimensional representations of SMART sequences and detect anomalies via reconstruction error, reducing labeled data demand but still yielding only numeric anomaly scores and need heavy feature engineering. LSTM-based models [19] capture temporal dynamics more faithfully, yet typically focus on binary failure prediction rather than structured time-to-failure (TTF) reasoning or interpretable explanations. After elaborate training, these models may show good statistical accuracies, hence we use these models as baselines in our evaluation 4.

Feature-engineered temporal systems (MVTRF, MSFRD, EC). Recent state-of-the-art (SOTA) works combine SMART with extensive temporal feature engineering. MVTRF [40] integrates short-term values, long-term histograms, and trend features to jointly predict failure status, type, and TTF, and exposes rule-like decision paths for diagnosis. MSFRD [41] leverages fine-grained telemetry and computes *mutation similarity* between current and historical attribute changes to assign fine-grained failure levels. Alibaba's production ensemble model for SMART logs [10] likewise uses rich temporal features and tree-based ensembles tuned for very low false positives, but, like MVTRF and MSFRD, ultimately exposes only scores and binary labels rather than operator-facing narratives or structured TTF reasoning. These approaches achieve strong accuracy but rely on complex, hand-crafted features and communicate results mainly through scores, labels, and rating levels.

Prior SMART based prediction models across classical supervised methods, unsupervised sequence models, and heavily engineered temporal pipelines share limitations that tuning cannot fix. They depend on fixed feature schemas that miss new failure modes, unstable and costly supervision on rare failures, rigid numeric features that ignore cross attribute correlations and causal structure, and they provide little human interpretable reasoning from SMART trends to failures and operator actions. SMARTTalk differs from these approaches along three main axes: **(1) Representation.** Rather than handcrafted histograms or mutation features, SMARTTalk learns CNN-based patterns over local SMART patches and translates each pattern into a short, human-readable phrase (e.g., "`r_187` stable low", "joint spike in `r_5` and `r_197`"). **(2) Reasoning.** Instead of using a numeric classifier, SMARTTalk feeds these textual trend summaries to an LLM, which, via chain-of-thought prompting, produces joint predictions of device status, TTF buckets, likely causes, and operator actions. **(3) Adaptivity.** SMARTTalk maintains an online pattern memory that detects, clusters, and promotes novel temporal patterns, enabling the system to capture emerging failure modes without retraining and expose them to the LLM in natural language.

In summary, SMARTTalk shifts from designing new numeric predictors to building a representation-and-reasoning pipeline that makes SMART logs *readable* and *explainable* to LLMs, improving operator-facing interpretability while retaining strong predictive performance.

3 SMARTTalk Design

SMARTTalk is a new systems architecture that translates numeric, multivariate telemetry into a stable symbolic trend language that LLMs can reason over. Its novelty is not in using CNNs, clustering, or LLMs, each component is standard, but in how the system composes them into a hybrid sym-

bolic–numeric pipeline that creates a new abstraction layer for telemetry: patch → pattern → phrase. This layer acts as a symbolic intermediate representation that is (i) human-interpretable, (ii) stable across drives and workloads, and (iii) locally adaptive through an online, continual Pattern Memory that recognizes novel failure modes without retraining.

SMARTTalk separates numerical trend extraction from language based reasoning. Offline, it processes historical SMART logs to learn patch encoders and a compact vocabulary of temporal trend patterns, calibrates pattern distance cutoffs, and stores the resulting patterns, cutoffs, and text phrases in a *pattern memory*. Online, a current n day window is sliced into patches, encoded with the frozen encoders, matched against the pattern memory, translated into short textual summaries, and then passed to an LLM that outputs drive status, time to failure, explanations, and recommended actions. This patch → pattern → phrase translation is the core abstraction: it elevates SMART telemetry from dense numeric sequences to interpretable, compositional symbols that support robust LLM reasoning and continual adaptation.

Instead of giving an LLM raw time series, we structure the input to balance detail, abstraction, and explainability. Specifically, SMARTTalk: **(1) Segments windows into patches:** Each $A \times n$ SMART window is split into short per-attribute and cross-attribute patches, capturing both local temporal trends and inter-attribute interactions. **(2) Learns a trend vocabulary:** We build a compact set of *trend patterns* from historical data, e.g., "stable low", "sharp spike", or "joint spike in `r_5` and `r_187`". **(3) Converts windows into textual summaries:** Each window is mapped to a sequence of pattern IDs, rendered as concise, human-like sentences.

The LLM then operates on these summaries using chain-of-thought reasoning, producing health predictions, time-to-failure estimates, explanations, and recommendations.

SMARTTalk consists of two phases: (i) an *offline phase* that learns patch encoders, patterns, and textual labels, and (ii) an *online phase* that maps incoming SMART windows to patterns, constructs textual summaries, and queries the LLM.

In this section, we formalize the problem and explain how SMART logs interface with LLMs through our three-stage pipeline across these two phases.

3.1 Problem Formulation

We consider a fleet of SSDs deployed in a production datacenter. Each drive $u \in \mathcal{U}$ exposes a set of A SMART attributes, which are logged at regular discrete time steps (e.g., daily). Let, $x_{u,a,t} \in \mathbb{R}$ denote the value of attribute $a \in \{1,\ldots,A\}$ for drive u at time t. We assume that for a subset of drives we also observe failure events and their timestamps.

SMART windows. For each drive u and reference time t, we define an n-day SMART window as the $A \times n$ matrix

$$M_{u,t} = \begin{bmatrix} x_{u,1,t-n+1} & \cdots & x_{u,1,t} \\ \vdots & & \vdots \\ x_{u,A,t-n+1} & \cdots & x_{u,A,t} \end{bmatrix} \in \mathbb{R}^{A\times n}.$$

which contains n consecutive time steps of all SMART attributes ending at time t. In our experiments we use $A = 15$ attributes and $n = 30$ days, but the formulation is general.

Status and time-to-failure (TTF) labels. For windows with ground truth, we assign two labels:
1. A *health status* label $y^{\text{status}}_{u,t} \in \mathcal{Y}^{\text{status}}$, which can represent, for example, {healthy, failed}.
2. A *time-to-failure* (TTF) label $y^{\text{ttf}}_{u,t} \in [0,+\infty]$, defined as the time (e.g., in days) from the end of the window t until the first observed failure of drive u. If the drive does not fail within a fixed horizon H, the label can be treated as censored or clipped at H.

Target outputs. Given a new SMART window $M_{u,t}$, **SMARTTalk** aims to produce:
1. A predicted health status $\hat{y}^{\text{status}}_{u,t} \in \mathcal{Y}^{\text{status}}$.
2. A predicted time-to-failure $\hat{y}^{\text{ttf}}_{u,t}$, either as a bucket (e.g., < 7 days, 7–30 days, > 30 days).
3. A natural-language *explanation* $e_{u,t}$ that describes which SMART trends and patterns led to the decision.
4. A natural-language *recommendation* $r_{u,t}$ (e.g., continue monitoring, migrate workload, replace drive) that is actionable for operators.

We therefore seek a mapping $f : \mathbb{R}^{A\times n} \longrightarrow \mathcal{Y}^{\text{status}} \times \mathbb{R}_{\geq 0} \times \mathcal{E} \times \mathcal{R}$, where $\mathcal{E}$ and $\mathcal{R}$ denote the spaces of explanations and recommendations, respectively. Each of these outputs is evaluated and discussed in Sections 4.2, 4.3, and 4.4.

LLM-based reasoning with summaries. *SMARTTalk* decomposes f into two stages:

$$S : \mathbb{R}^{A\times n} \to \mathcal{T}, \qquad F_{\text{LLM}} : \mathcal{T} \to \mathcal{Y}^{\text{status}} \times \mathbb{R}_{\geq 0} \times \mathcal{E} \times \mathcal{R},$$

where, S is a learned summarization function that converts $M_{u,t}$ into a compact textual representation $T_{u,t} \in \mathcal{T}$ describing per-attribute and cross-attribute trend patterns (e.g., "SMART-5 stable, then repeated spikes in the last 10 days"). F_{LLM} is an LLM-based reasoning module that, given $T_{u,t}$ and a chain-of-thought style prompt, infers status, TTF, explanations, and recommendations.

3.2 Offline Pattern Learning

The offline phase of SMARTTalk learns a compact library of patch-level patterns and their associated distance cutoffs from historical SMART logs, as shown in Figure 3, which summarizes the pipeline. Given long sequences of raw SMART attributes, we first form fixed-length windows and slice them into short temporal patches. Small self supervised [6, 11, 15, 20] CNN encoders then map these patches into low dimensional embeddings. We train these encoders with self supervised objectives defined directly on *unlabeled SMART windows*, such as predicting masked days within a

patch from the remaining days or distinguishing a real patch from a time shuffled or attribute permuted version, so that the learned representations capture reliability relevant trends without using failure labels. We cluster these embeddings with k-means to obtain a finite set of attribute and cross-attribute patterns, and finally use failure labels *only* to calibrate distance cutoffs that separate typical patches from out-of-library behavior. The resulting pattern centers, distance cutoffs, and textual phrases initialise the *PatternMemory* used at inference time.

3.2.1 Patch-Based Representation of SMART Windows

Recall from Section 3.1 that an n-day SMART window for drive u ending at time t is represented as $M_{u,t} \in \mathbb{R}^{A\times n}$, where A is the number of SMART attributes. SMARTTalk first converts each window into a set of short views that capture local behaviour while preserving temporal order.

Per-attribute 1D patches. For each attribute $a \in \{1,\dots,A\}$ we consider its length-n time series within the window and split it into $T = n/L$ contiguous blocks of length L (we use $n = 30$ days and $L = 5$ days in our experiments):

$$p^{(a)}_{u,t,\ell} \in \mathbb{R}^L, \quad \ell = 1,\dots,T.$$

Each $p^{(a)}_{u,t,\ell}$ captures a short-term local trend for attribute a over L days and can be mapped back to a concrete time range (e.g., days 0–4, 5–9, ..., 25–29).

Cross-attribute 2D patches. We also form 2D patches that include *all* attributes over the same L-day blocks:

$$P_{u,t,\ell} \in \mathbb{R}^{A\times L}, \quad \ell = 1,\dots,T.$$

Each $P_{u,t,\ell}$ summaries how attributes evolve jointly over the same time segment.

For an $A \times n$ window we therefore obtain $A \cdot T$ 1D patches and T 2D patches. The choice of L trades off temporal resolution and noise averaging: shorter patches isolate rapid changes, whereas longer patches smooth noise but may blur short bursts.

3.2.2 Self-Supervised Patch Encoders

SMARTTalk represents each patch with a low-dimensional embedding using small self-supervised CNNs; failure labels are not required for this stage.

1D attribute encoder. For a 1D patch $p^{(a)}_{u,t,\ell}$ we learn an encoder $f_{\text{attr}} : \mathbb{R}^L \to \mathbb{R}^{d_a}$ implemented as a lightweight 1D CNN:

$$z^{(a)}_{u,t,\ell} = f_{\text{attr}}(p^{(a)}_{u,t,\ell}) \in \mathbb{R}^{d_a}.$$

A decoder g_{attr} reconstructs the original patch; the encoder and decoder are trained jointly via reconstruction loss to ensure that $z^{(a)}_{u,t,\ell}$ preserves the essential local trend.

2D cross-attribute encoder. Similarly, for each cross-attribute patch $P_{u,t,\ell} \in \mathbb{R}^{A\times L}$ we train a 2D CNN encoder $f_{\text{cross}} : \mathbb{R}^{A\times L} \to \mathbb{R}^{d_c}$:

$$z_{u,t,\ell} = f_{\text{cross}}(P_{u,t,\ell}) \in \mathbb{R}^{d_c},$$

with a small decoder g_{cross} trained to reconstruct $P_{u,t,\ell}$.

Self-supervised objective. The encoders are jointly trained on unlabeled historical SMART windows using a combined reconstruction loss $\mathcal{L} = \sum_{u,t,a,\ell} \left\| p^{(a)}_{u,t,\ell} - g_{\text{attr}}(f_{\text{attr}}(p^{(a)}_{u,t,\ell})) \right\|_2^2 + \lambda \sum_{u,t,\ell} \left\| P_{u,t,\ell} - g_{\text{cross}}(f_{\text{cross}}(P_{u,t,\ell})) \right\|_2^2$, where λ balances the two branches. Once converged, we discard the decoders and freeze f_{attr} and f_{cross}.

3.2.3 Pattern Learning and Memory Initialization

A three stage symbolic representation pipeline. SMARTTalk introduces a structured, three stage pipeline that converts raw telemetry into symbolic trends that an LLM can reason about. First, *patch extraction* slices each window into short one dimensional and two dimensional views that capture local temporal behaviour and cross attribute interactions as discussed in Section 3.2.1. Second, *pattern discovery* runs self supervised encoders and clustering on these patches to build a vocabulary of reusable temporal and cross attribute trends as discussed in 3.2.2. Third, *phrase generation*, which we discuss in this section, renders each learned pattern as a concise natural language phrase so that every window is represented as a sequence of symbolic tokens rather than raw numeric traces. This abstraction layer decouples how trends are extracted (numeric and learned) from how they are understood (symbolic and interpreted), which makes the representation modular and robust and allows it to be reused across drives, time periods, and LLM backends.

Pattern learning via k-means. We run k-means on the frozen embeddings to obtain base trend patterns. For attribute-level patterns, we cluster all $\{z^{(a)}_{u,t,\ell}\}$ across drives and windows into K_{attr} clusters with centres $\{c^{\text{attr}}_k\}_{k=1}^{K_{\text{attr}}}$. Similarly, we cluster $\{z_{u,t,\ell}\}$ into K_{cross} centres $\{c^{\text{cross}}_k\}_{k=1}^{K_{\text{cross}}}$. These centres correspond to recurring temporal trends such as "stable near zero", "gradual increase", "late sharp spike", or "joint spike in `r_5` and `r_187`".

Calibrating pattern distance cutoffs with labels. After obtaining the pattern centres, we leverage failure labels to determine distance thresholds that separate typical behaviour from patches that fall outside the learned pattern library. Specifically, for each training patch, we compute its distance d to the nearest pattern centre and record whether it originates from a healthy or failing window. For a candidate cutoff τ, we classify patches with $d > \tau$ as outside the library and compute the corresponding patch-level metrics: the false positive rate $\text{FPR}(\tau)$ on healthy windows and the true positive rate $\text{TPR}(\tau)$ on failing windows. We then sweep τ across a

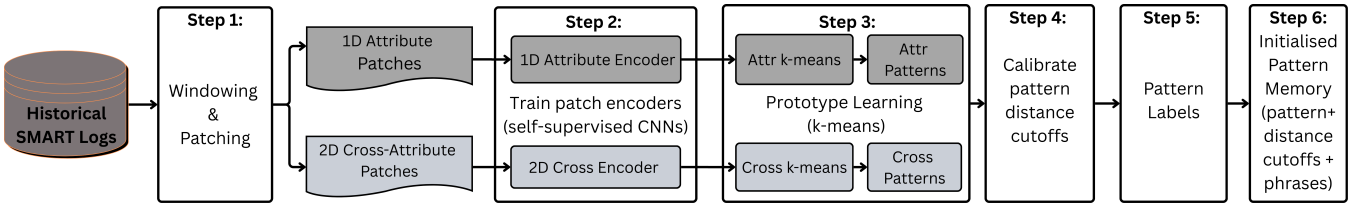


Figure 3: Offline phase of SMARTTalk: from historical SMART logs to self-supervised patch encoders, base patterns, calibrated distance cutoffs, and an initial *PatternMemory*.

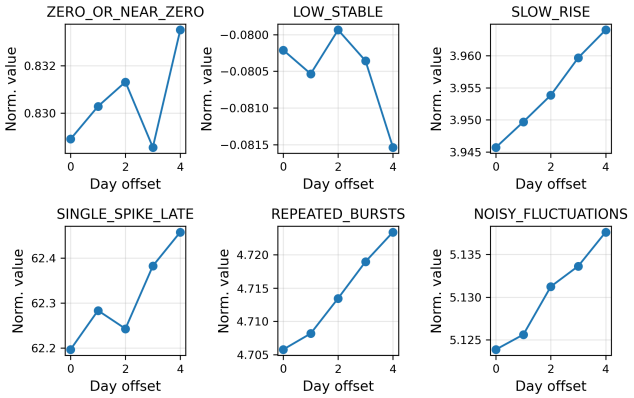


Figure 4: Example attribute-level patterns and their phrases.

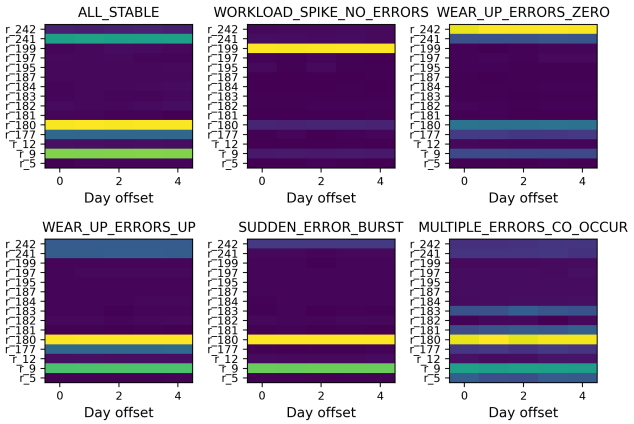


Figure 5: Example cross-attribute patterns and their phrases.

grid of quantiles and select the optimal cutoffs τ_{attr} and τ_{cross} such that $\text{FPR}(\tau) \leq \alpha$ (we use $\alpha = 0.05$) while maximising the F1-score over all patches. These calibrated cutoffs ensure that only a small fraction of patches from healthy drives are mistakenly flagged as anomalous, while a large fraction of patches from failing windows with unusual behaviour are correctly identified.

Learning pattern phrases and initial memory. The final offline step converts numeric patterns into a compact vocabulary of text tokens that the LLM can interpret. Given the attribute and cross-attribute pattern centres along with their calibrated distance cutoffs, phrase learning is formulated as an offline summarisation task over patches that are well represented by each pattern.

Figure 4 illustrates six representative attribute-level patterns and their traces. For each attribute pattern k, we collect all patches whose closest centre is c_k^{attr} and whose distance falls below the cutoff τ_{attr}. We then compute simple temporal statistics from these patches, including mean level, variance, start–end difference, maximum change, number of large excursions, and the position of the largest excursion within the L-day patch. Using rule-based heuristics, we map these statistics to a coarse trend label (e.g., ZERO_OR_NEAR_ZERO, LOW_STABLE, SLOW_RISE, SINGLE_SPIKE_LATE, REPEATED_BURSTS, NOISY_FLUCTUATIONS) and to a short natural-language phrase describing the trend (e.g., "flat near zero", "slowly increasing", "single late spike").

Figure 5 shows examples of these cross-attribute patches. We apply a similar procedure to cross-attribute patterns. For each cross pattern k, we collect patches whose closest centre is c_k^{cross} and whose distance is below τ_{cross}, compute per-attribute statistics over the $A \times L$ patch, and identify which attributes exhibit stable behaviour, gradual wear-out signals, or error bursts. Heuristics then group these patterns into categories such as ALL_STABLE, WORKLOAD_SPIKE_NO_ERRORS, WEAR_UP_ERRORS_ZERO, and assign phrases like "all attributes stable", "workload spike without new errors", or "multiple error counters rise together".

The resulting phrase dictionaries $\mathcal{V}_{\text{attr}}$ and $\mathcal{V}_{\text{cross}}$ associate each pattern index with a discrete trend label and a canonical phrase. These tokens are stable across drives and over time, and they form the vocabulary that will later be presented to the LLM in the online phase. We store all base patterns, calibrated distance cutoffs, and phrase dictionaries in a *PatternMemory* structure that is initialised at the end of the offline phase.

Phrase dictionary selection and generalization. The phrase dictionary is not manually written for individual test examples. It is selected offline from the training split by clustering patch embeddings and then describing each cluster using aggregate temporal statistics such as level, variance, start–end change, spike magnitude, and spike position. This produces coarse phrases such as "flat near zero", "slow rise", "single late spike", or "multiple error counters rise together". The dictionary is expected to generalize when the same SMART attributes preserve similar device semantics, because phrases describe normalized temporal shapes rather than drive identi-

fiers or absolute timestamps. When a new workload or device family produces a patch that does not match the current dictionary, SMARTTalk marks it as out-of-library and logs it for later insertion into *PatternMemory*. The main limitation is that a device with different SMART semantics, new attributes, or failure modes that do not appear as local trends or cross-attribute co-occurrences may require recalibrating the pattern library and phrase dictionary.

3.3 Online Inference

The online phase applies the frozen patch encoders and the *PatternMemory* to each new SMART window and queries an LLM as shown in Figure 6. Given a current n-day window, SMARTTalk maps short patches to pattern phrases using the stored centers and distance cutoffs, builds a natural language summary, and asks the LLM to produce exploratory analysis.

3.3.1 Window Encoding and Pattern Lookup

Given a fresh window $M_{u,t} \in \mathbb{R}^{A\times n}$ for drive u, SMARTTalk first standardizes attributes using the offline mean and variance and performs the same windowing and patching as in Section 3.2.1 to obtain 1D patches $p^{(a)}_{u,t,\ell}$ and 2D patches $P_{u,t,\ell}$ of length L days. Each patch is embedded with the frozen encoders f_{attr} and f_{cross} and compared against the patterns stored in *PatternMemory*.

For each patch we compute its distance to every pattern center and select the closest one. If the distance is below the corresponding distance cutoff τ_{attr} or τ_{cross}, the patch is treated as an instance of that pattern. Otherwise it is marked as an *out-of-library* pattern and its embedding, attribute index, and time range are logged for possible future extension of the pattern library. This yields, for each attribute and for the cross-attribute view, a short sequence of pattern IDs together with flags indicating which patches are out-of-library.

Online pattern-memory growth. Out-of-library patches identified during pattern lookup are appended to a small buffer rather than discarded. At regular intervals, SMARTTalk clusters the buffered embeddings to propose candidate new patterns and reuses the offline calibration procedure to assign each accepted pattern a center, distance cutoff, and phrase before inserting it into *PatternMemory*. This lightweight online update mechanism *allows SMARTTalk to absorb emerging workload regimes and failure signatures without retraining the encoders or rerunning the offline pipeline*, so the pattern library gradually becomes more representative of the field deployment over time.

3.3.2 Summary Construction and LLM Query

SMARTTalk next converts the pattern assignments for window $M_{u,t}$ into a structured textual summary. Pattern IDs are mapped to phrases using the attribute and cross-attribute phrase dictionaries $\mathcal{V}_{\text{attr}}$ and $\mathcal{V}_{\text{cross}}$ in *PatternMemory*. Phrases that repeat over multiple neighboring patches are merged into longer descriptions (e.g., "stable near zero for three segments" or "repeated bursts over the last 10 days"), and out-of-library patches are described explicitly as "unusual" behavior.

The resulting summary $T_{u,t}$ is inserted into a chain-of-thought prompt that instructs the LLM to reason about (i) whether the drive is currently healthy or at risk, (ii) the likely TTF bucket, and (iii) why the recent behavior is benign or concerning, followed by recommended operator actions. The LLM's answer is parsed into a status label, TTF bucket, explanation text, and recommendations, which are logged together with the summary for later analysis.

Role of the LLM. The CNN encoders and pattern memory extract local temporal evidence, but they do not by themselves decide how multiple pieces of evidence should be combined into an operator-facing decision. The LLM bridges this critical gap by composing attribute-level and cross-attribute patterns over the full window, distinguishing workload-driven changes from reliability signals, and mapping evidence to a status and TTF bucket. It also generates concise explanations and recommended actions. In this design, earlier stages provide a structured symbolic interface, while the LLM performs the higher-level reasoning needed for diagnosis, urgency assessment, and operator guidance that raw CNN patterns cannot provide alone.

3.4 Example and Usage Scenarios

Example setup. We select a 30-day window for a single drive and focus on two error-related attributes, `r_5` and `r_187`. Using a patch length of $L = 5$ days, each series is split into six contiguous patches P1–P6. The top panel of Figure 7 shows the raw series; the bottom panel shows the pattern phrases assigned to each 5-day patch.

Pattern assignments. In P1 and P2 (days 0–10), both attributes stay near zero, so `r_5` maps to "stable" and `r_187` to "flat zero". In P3 (days 10–15), `r_5` shifts to "noisy increase" while `r_187` remains "flat zero", indicating the first deviation in `r_5`. In P4 (days 15–20), `r_5` becomes "spike" and `r_187` is still "flat zero", capturing an isolated burst in `r_5`. In P5 and P6 (days 20–30), both attributes are labeled with "spike" patterns, reflecting sustained large spikes in `r_187` together with continued spikes in `r_5`. The cross-attribute patches for P5–P6 map to a pattern such as MULTIPLE_ERRORS_CO_OCCUR, which *PatternMemory* records as frequently appearing shortly before failures on this model family.

Example summary sent to the LLM. For this window, SMARTTalk aggregates the per-patch phrases into a natural-language summary $T_{u,t}$ such as:

```
Over days 0-10, error counters r_5 and r_187 stay near zero
and stable. Between days 10-15, r_5 shows a noisy upward drift
while r_187 remains flat at zero. From days 15-20, r_5 exhibits a
short spike but r_187 is still flat zero. Over days 20-30, both r_5
and r_187 show repeated spikes, and the joint pattern in the last
```

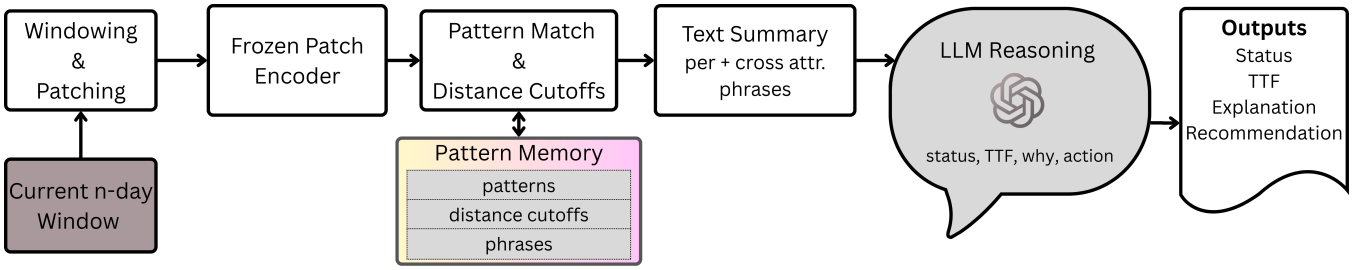


Figure 6: Online phase of SMARTTalk: a current n-day SMART window is mapped to pattern phrases via *PatternMemory* (patterns + distance cutoffs + phrases) and fed to an LLM that returns status, TTF, explanation, and recommended actions.

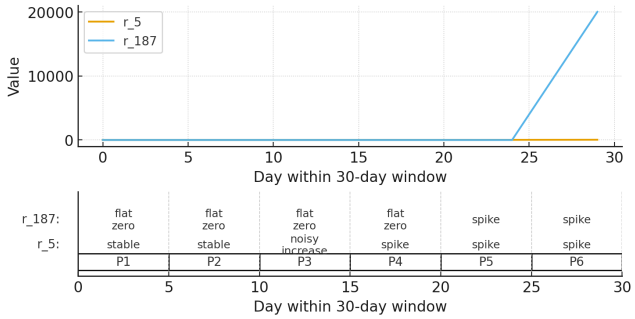


Figure 7: Example of SMARTTalk processing a 30-day window for attributes `r_5` and `r_187`. Top: raw trajectories. Bottom: six 5-day patches (P1–P6) and their attribute-level pattern phrases (e.g., "stable", "noisy increase", "spike").

```
two 5-day segments matches the "MULTIPLE_ERRORS_CO_OCCUR" pattern,
which has historically preceded failures within about one week on
this model family.
```

This summary is inserted into the CoT prompt described in Section 3.3.2. Given this input, the LLM typically reasons that error-related attributes exhibit recent sustained spikes and that the cross-attribute pattern matches one that often occurs shortly before failures; it therefore assigns the drive to a high-risk status with predicted TTF in the "<7 days" bucket and recommends migrating critical workloads and scheduling replacement. When all patches instead map to "stable" or "noisy but bounded" patterns, the same mechanism produces benign assessments.

This example illustrates SMARTTalk in practice. In routine monitoring, error counters and bounded workload changes lead to low concern and continued monitoring. In early warning cases, spikes in reliability-related attributes such as `r_5`, `r_187`, or `r_197` lead to migration, diagnostics, or scheduled replacement depending on urgency. In false alarm triage, a workload spike without error, spare, or wear degradation can be reported as unusual but not immediately dangerous, so the recommendation remains monitoring rather than automatic replacement. In novel pattern handling, out-of-library patches are logged into *PatternMemory*, allowing operators to inspect emerging behaviors and update the pattern library without retraining the pipeline.

4 Evaluation

In this section, we evaluate *SMARTTalk* on real Alibaba SMART logs and study three questions. First, how well does it predict drive status compared with conventional SMART-based baselines and recent state-of-the-art (SOTA) methods? Second, how accurately can it predict time-to-failure (TTF) buckets? Third, how good are the generated explanations and recommendations, both in terms of automatic metrics and their alignment with desired actions?

4.1 Experimental Setup

Dataset and temporal splits. We use the Alibaba SSD dataset as introduced in Section 2.2. For each drive, we construct sliding 30-day windows; labels in the dataset provide the status (failed or healthy) and the time to failure in days. To avoid temporal leakage, windows are partitioned by calendar month: the earliest months are used for training, the subsequent month for validation, and a later month for testing. Table 3 summarises the partitions for three experimental rounds; month indices will be adjusted to match the final datasets.

Table 3: Data partitions for SMARTTalk experiments.

Dataset	Round	Train	Val	Test
MB1, MB2	1	1–22$^{\text{nd}}$	23$^{\text{rd}}$	24$^{\text{th}}$
	2	1–21$^{\text{st}}$	22$^{\text{nd}}$	23$^{\text{rd}}$
	3	1–20$^{\text{th}}$	21$^{\text{st}}$	22$^{\text{nd}}$

SMART windows, patches, and patterns. Unless otherwise stated, we use windows of $n = 30$ days and patch length $L = 5$ days, which yields $T = n/L$ patches per attribute. The 1D and 2D patch encoders described in Section 3 are trained offline using only the training split and then frozen. The number of attribute and cross-attribute patterns is set to $K_{\text{attr}} = 16$ and $K_{\text{cross}} = 8$, and distance cutoffs $(\tau_{\text{attr}}, \tau_{\text{cross}})$ are learned as discussed in Section 3.2.3. Pattern phrases are curated once per dataset and reused across experiments.

Existing solutions. We compare SMARTTalk against wide range of the classical SMART-based methods, recent SOTA, and LLM-based variants. Random forests (**RF**) and feed-forward neural networks (**NN**) follow the feature engineering in [5]. Ensemble Classifier (**EC**): After choosing the most

important features, the data are given to several different classifiers. Their individual predictions are then combined into one final decision about whether the drive is healthy or failing [10]. An autoencoder (**AE**) reconstructs healthy windows and uses reconstruction error as a failure score [9]. An LSTM sequence model (**LSTM**) is trained directly on the SMART time series [19]. The multi-view temporal random forest (**MVTRF**) [40] and the fine-grained telemetry based method (**MSFRD**) [41] are recent SOTA for SMART-based reliability modelling; we treat them as strong reference points. Since there is no published LLM-based solution for this task, we define two LLM baselines. **Raw-LLM** directly verbalises raw SMART values over the window and prompts the LLM without using any patterns. **Heuristic-LLM** converts simple threshold-based rules into textual prompts for the LLM. In contrast, **SMARTTalk** employs the full pipeline: it encodes patches, leverages the pattern memory, generates textual summaries, and applies LLM reasoning to provide diagnostics and predictions.

LLM backbones. We evaluate SMARTTalk with four open-weight, instruction-tuned LLMs and one proprietary LLM as summarized in Table 4. All five LLMs use the same chain-of-thought style prompt and decoding configuration; we vary only the backbone model so that differences in accuracy and explanation quality can be attributed to the underlying LLM rather than prompt design. For the proprietary model, we treat the system as a black box and use only public APIs, without any fine-tuning or access to internals.

Table 4: LLMs used as SMARTTalk backbones.

ID	Model name	Params (B)	Context
$\mathbf{OS}_1$	Llama-3.1-8B-Instruct	8	128K tokens
$\mathbf{OS}_2$	Qwen2.5-14B-Instruct	14.7	131K tokens
$\mathbf{OS}_3$	Phi-4	14	16K tokens
$\mathbf{OS}_4$	Gemma-2-27B-it	27	8K tokens
PROP	GPT-4o	—	128K tokens

Evaluation Metrics. For **status prediction**, we focus on the failure (high-risk) class and treat each window as positive if the drive is truly at risk and negative otherwise. A true positive (TP) is a window where the drive is actually at risk and the model predicts RISK; a false positive (FP) is a healthy window that is incorrectly flagged as RISK, i.e., a false alarm; a true negative (TN) is a healthy window correctly predicted as healthy; and a false negative (FN) is an at-risk window that is incorrectly predicted as healthy, i.e., a missed failure. Because the processed window-level dataset is highly imbalanced, all methods are evaluated on the same fixed imbalanced sampled test set so that confusion-matrix metrics are comparable across methods while keeping LLM inference tractable.

We compute precision as $\text{Prec} = TP/(TP+FP)$ and recall as $\text{Rec} = TP/(TP+FN)$ by aggregating counts over the test windows. We then report the $F_{0.5}$-score, $F_{0.5} = \frac{(1+0.5^2)\,\text{Prec}\,\text{Rec}}{0.5^2\,\text{Prec}+\text{Rec}}$. This is the harmonic mean of precision and recall, where precision is weighted more heavily. In operational settings, SSD failure prediction must limit spurious alarms, so practitioners prioritize precision [37]. Consequently, we adopt the $F_{0.5}$-score, which emphasizes precision, as our main metric for comparing schemes in production-like environments.

To quantify false alarms and missed failures, we report the false positive rate (FPR) and false negative rate (FNR): $\text{FPR} = \frac{FP}{FP+TN}$, $\text{FNR} = \frac{FN}{TP+FN}$. FPR measures healthy windows incorrectly flagged as at risk, while FNR measures missed at-risk windows. Thus, FPR reflects operator burden and FNR reflects failure risk.The results of status prediction are discussed in Section 4.2.

For **TTF prediction** we discretize the time-to-failure (TTF) into coarse urgency buckets (e.g., <7 days, 7–30 days, >30 days) and ask the LLM to choose one bucket per at-risk window. Let t_i denote the true TTF (in days) for window i and $\hat{b}_i$ the bucket predicted by the LLM; each bucket $\hat{b}_i$ is associated with a midpoint $m_{\hat{b}_i}$ (e.g., 3.5 days for <7 days). We compute a macro-averaged TTF F_1 by treating each bucket as a class, computing its per-bucket F_1, and then averaging these scores uniformly across buckets so that short-horizon and long-horizon failures are weighted equally. To quantify how far the predicted urgency is from the true TTF, we report a bucketed mean absolute error $\text{bMAE} = \frac{1}{N}\sum_{i=1}^{N}|t_i - m_{\hat{b}_i}|$, where smaller values indicate that predicted buckets are, on average, closer in days to the true failure times. Finally, we measure the fraction of failure windows where the predicted bucket is time-consistent with the true TTF via $\text{Cov}_{\pm 5} = \frac{1}{N_{\text{fail}}}\sum_{i=1}^{N_{\text{fail}}}\mathbf{1}\big(|t_i - m_{\hat{b}_i}| \le 5\big)$, i.e., the proportion of at-risk windows whose true TTF lies within ± 5 days of the bucket midpoint; higher macro-F_1, lower bMAE, and higher $\text{Cov}_{\pm 5}$ all correspond to better guidance about how urgently a drive needs attention. The results of TTF prediction are discussed in Section 4.3.

For **explanations and recommendations**, operators care about why a drive is at risk and what to do next, but evaluating such explanations is challenging because it is expensive to obtain ground-truth natural language annotations for each window; instead, we use two label-free strategies. First, we evaluate output quality using an external LLM-as-judge [43] (GPT-5.1 Thinking [26]) that scores each explanation and action for correctness and operator usefulness. Second, we run perturbation-based tests that check whether explanations and actions react consistently when we synthetically increase or decrease specific SMART risk signals [14]. The full protocol and results, including the LLM-as-judge prompts and perturbation design, is described in Section 4.4.

4.2 Status Prediction

Table 5 reports status prediction performance of existing solutions (RF, NN, AE, and LSTM), recent SOTA methods (MVTRF and MSFRD), and LLM-based variants on MB1 and MB2. SMARTTalk is evaluated with both open-source

and proprietary backbones. In addition to precision, recall, and $F_{0.5}$, We report FPR and FNR to distinguish false alarms on healthy drives from missed warnings on at-risk drives. This is crucial under highly imbalanced data, where a model may appear conservative by predicting few failures yet still miss most true failures.

Table 5: Status prediction performance on the fixed imbalanced sampled test set. Precision (P), Recall (R), $F_{0.5}$, false positive rate (FPR), and false negative rate (FNR) are reported.

Method	MB1					MB2				
	P↑	R↑	$F_{0.5}$↑	FPR↓	FNR↓	P↑	R↑	$F_{0.5}$↑	FPR↓	FNR↓
RF [5]	0.52	0.35	0.47	0.014	0.65	0.68	0.33	0.56	0.0068	0.67
NN [5]	0.63	0.24	0.48	0.0061	0.76	0.57	0.17	0.39	0.0056	0.83
EC [10]	0.78	0.22	0.52	0.0027	0.78	0.85	0.24	0.56	0.0018	0.76
AE [9]	0.48	0.31	0.43	0.0146	0.69	0.57	0.25	0.45	0.0082	0.75
LSTM [19]	0.53	0.42	0.50	0.0162	0.58	0.60	**0.50**	0.58	0.0145	**0.50**
MVTRF [40]	0.83	**0.53**	0.75	0.0047	**0.47**	0.73	0.27	0.54	0.0043	0.73
MSFRD [41]	0.87	0.49	0.75	0.0032	0.51	0.82	0.33	0.63	0.0031	0.67
Raw-LLM (OS1)	0.00	0.00	0.00	0.0061	1.00	0.00	0.00	0.00	0.0058	1.00
Heuristic-LLM (OS1)	0.36	0.04	0.14	0.0031	0.96	**0.89**	0.07	0.27	0.0004	0.93
Raw-LLM (OS2)	0.00	0.00	0.00	0.0074	1.00	0.00	0.00	0.00	0.0062	1.00
Heuristic-LLM (OS2)	0.26	0.03	0.10	0.0037	0.97	0.85	0.04	0.17	0.0003	0.96
Raw-LLM (OS3)	0.00	0.00	0.00	0.0059	1.00	0.00	0.00	0.00	0.0055	1.00
Heuristic-LLM (OS3)	0.23	0.02	0.07	0.0029	0.98	0.88	0.04	0.17	**0.0002**	0.96
Raw-LLM (OS4)	0.00	0.00	0.00	0.008	1.00	0.00	0.00	0.00	0.0066	1.00
Heuristic-LLM (OS4)	0.51	0.07	0.23	0.0029	0.93	0.78	0.04	0.17	0.0005	0.96
Raw-LLM (PROP)	0.50	0.01	0.05	**0.0004**	0.99	0.55	0.02	0.09	0.0007	0.98
Heuristic-LLM (PROP)	0.43	0.04	0.15	0.0023	0.96	0.83	0.07	0.26	0.0006	0.93
SMARTTalk (OS1)	0.81	0.37	0.65	0.0038	0.63	0.75	0.29	0.57	0.0042	0.71
SMARTTalk (OS2)	0.87	0.41	0.71	0.0027	0.59	0.87	0.34	**0.66**	0.0022	0.66
SMARTTalk (OS3)	**0.90**	0.50	**0.78**	0.0024	0.50	0.81	0.34	0.63	0.0035	0.66
SMARTTalk (OS4)	0.80	0.52	0.72	0.0057	0.48	0.80	0.34	0.63	0.0037	0.66
SMARTTalk (PROP)	0.85	0.45	0.72	0.0035	0.55	0.87	0.30	0.63	0.0019	0.70

The results show that simply prompting an LLM on raw SMART values is not enough for reliable status prediction. Raw-LLM variants almost never predict a drive as failed and label nearly every window as healthy. This behavior produces near-zero precision and recall and leads to an FNR close to one, showing that Raw-LLM misses almost all at-risk windows despite producing few useful warnings. Heuristic-LLM variants perform slightly better by using hand-crafted threshold summaries, and on MB2 they sometimes achieve high precision and low FPR. However, they fire only on the most extreme cases and still miss most true failures, which leads to low recall, high FNR, and poor $F_{0.5}$ scores. These results show that raw numeric prompting or simple hand-written summaries are insufficient because the LLM needs structured temporal abstractions that expose degradation patterns without overwhelming it with noisy variability.

Existing solutions such as RF, NN, AE, LSTM, MVTRF, and MSFRD provide a broad set of numerical baselines because they learn directly from SMART trends and expose different precision, recall, FPR, and FNR tradeoffs. MVTRF and MSFRD achieve strong precision and competitive recall on MB1, but SMARTTalk matches or exceeds their precision-biased $F_{0.5}$ across both SSD models while maintaining low FPR and non-trivial recall. In particular, SMARTTalk achieves the best $F_{0.5}$ on MB1 and the best or comparable $F_{0.5}$ on MB2, while keeping false alarm rates low. This indicates that the patch–pattern–phrase representation gives the LLM discriminative temporal signals similar to carefully engineered numerical features, but in a form that also supports natural-language reasoning, time-to-failure estimation, explanations, and operator actions.

Handling false positives and false negatives. In SMARTTalk, false positives are treated as early warnings, not automatic replacements. The output includes patterns, explanations, concern levels, and recommendations, enabling operators to decide between immediate action and monitoring. False negatives are more critical as they reflect missed risks; SMARTTalk mitigates this through sliding-window evaluation, multi-attribute evidence, and updates to *PatternMemory*, though weak or delayed signals remain a limitation. Unlike numerical predictors that provide only scores, SMARTTalk improves auditability by exposing decision-driving patterns. While Raw-LLM and Heuristic-LLM achieve low FPR by predicting fewer failures, this results in high FNR. SMARTTalk maintains a more practical balance between false alarms and missed failures.

The key takeaway is that converting raw SMART logs into abstract trend patterns turns LLMs from ineffective status predictors into detectors that are competitive with or better than state-of-the-art SMART baselines. SMARTTalk achieves about $50\times$ *higher* $F_{0.5}$ *than Raw-LLM, about* $4\times$ *higher* $F_{0.5}$ *than Heuristic-LLM, and roughly* 25% *better health classification than existing SMART-based methods, while also reporting low false alarm rates and reducing missed failures compared with naive LLM baselines. It further provides richer outputs such as time-to-failure guidance, explanations, and operator recommendations.*

4.3 Time-to-Failure Prediction

We now examine how accurately SMARTTalk predicts time-to-failure (TTF) buckets. Although MVTRF can estimate remaining lifetime, its published setups produce continuous risk or lifetime scores under different horizons and evaluation protocols, which are not directly comparable to our discrete urgency buckets (<7, 7–30, >30 days), we therefore focus our TTF evaluation on LLM-based methods, comparing SMARTTalk across multiple LLMs under the same pattern-based summaries and TTF instruction.

Table 6 reports macro-F_1 across TTF buckets and bucketed mean absolute error (bMAE) for five open-source LLM backbones. Each backbone appears twice: once with Raw-LLM and once with SMARTTalk. For each method the LLM is instructed to choose among the predefined TTF buckets (for example <7 days, 7–30 days, and >30 days); bMAE is computed from the bucket midpoints.

Across backbones, SMARTTalk's TTF performance follows a consistent pattern: weaker open-source models lag behind, while stronger open-source and the proprietary model

Table 6: TTF prediction: We report macro-F1 over TTF buckets (<7, 7–30, >30 days), bucketed MAE (bMAE), and coverage Cov$\pm$5, computed only on windows whose status is correctly predicted as RISK.

LLM	MB1 TTF $F_1 \uparrow$	MB1 bMAE $\downarrow$	MB1 $Cov_{\pm5} \uparrow$	MB2 TTF $F_1 \uparrow$	MB2 bMAE $\downarrow$	MB2 $Cov_{\pm5} \uparrow$
OS_1	0.58	11.0	0.52	0.52	11.8	0.48
OS_2	0.63	9.8	0.57	0.60	10.0	0.56
OS_3	0.70	9.0	0.61	0.58	10.5	0.54
OS_4	0.64	9.5	0.59	0.57	10.7	0.53
PROP	0.65	9.3	0.60	0.59	10.2	0.55

deliver substantially better bucket accuracy, lower timing error, and higher coverage. The best-performing backbone differs slightly between MB1 and MB2, but the gap between the top several models is small, suggesting that once the LLM is reasonably capable, the pattern-based summaries dominate the quality of TTF guidance.

The key takeaway is that SMARTTalk can provide reliable urgency estimates with multiple LLM backbones, achieving TTF bucket macro-F_1 around 0.6 *with bucketed MAE near* 10 *days and more than half of failures landing within* $\pm$5 *days of the predicted bucket, allowing operators to choose an open or proprietary model based on deployment constraints without sacrificing TTF quality.*

4.4 Explanations and Recommendations

LLM-as-judge. We first sampled windows from MB1 and MB2 for failing drives (True Positives predicted by SMARTTalk) and across time-to-failure buckets. For each method, we run the full pipeline and record its predicted status, time-to-failure bucket, free-form explanation, and recommended action for each window. We then use GPT-5.1 Thinking as an *external* LLM-as-judge. The judge is given (i) a short textual summary of the SMART window, (ii) the method's predicted status and time-to-failure, and (iii) the explanation and recommended action, but not the method name. It is prompted to rate, on a 1–5 Likert scale, (a) how *correct and faithful* the explanation is to the status/TTF prediction and the SMART trends, and (b) how *useful and safe* the recommended action would be for an operator. We report the average explanation score (**ExpScore**) and action score (**RecScore**) across windows.

Perturbation-based robustness. Second, we probe how sensitive each method is to controlled changes in the SMART attributes. For a subset of windows, we construct synthetic *perturbations* by modifying one or two reliability-related attributes (e.g., `r_5`, `r_187`, `r_197`) while keeping the others fixed. We consider "risk-up" perturbations (e.g., injecting a monotone increase or spike in an error counter) and "risk-down" perturbations (e.g., clamping an attribute to a healthy baseline). For each original/perturbed pair, we rerun the method and measure:

AttrSens: the fraction of risk-up perturbations for which the explanation explicitly mentions the perturbed attribute (via a fixed attribute-name lexicon), indicating that the method notices the changed signal.

ActDirAcc: the fraction of perturbations for which the recommended action moves in the intuitively correct direction: it becomes *more* conservative (e.g., from "monitor" to "backup and replace") when risk is increased, and does not become more conservative when risk is decreased.

These metrics do not require human reference explanations: we control the perturbations and only check whether each method's explanations and recommendations react in a consistent and sensible way.

Table 7: Quality of explanations and recommendations on MB1+MB2 for SMARTTalk. ExpScore and RecScore are 1–5 ratings (LLM-as-judge). AttrSens and ActDirAcc are perturbation-based robustness metrics (higher is better).

Method	ExpScore (1–5)	RecScore (1–5)	AttrSens ($\uparrow$)	ActDirAcc ($\uparrow$)
OS_1	4.34	4.26	0.79	0.82
OS_2	4.22	4.30	0.84	0.88
OS_3	4.50	4.75	0.87	0.91
OS_4	4.45	4.90	0.90	0.93
PROP	4.50	4.60	0.82	0.90

Table 7 summarises explanation and recommendation quality for SMARTTalk variants. "ExpScore" and "RecScore" are 1–5 scores from GPT-5.1 Thinking (higher is better), while "AttrSens" and "ActDirAcc" are proportions in $[0,1]$. Across the open-source variants, SMARTTalk consistently produces explanations that the judge rates as both accurate and useful, with later-generation backends yielding slightly stronger and more stable judgements than earlier ones. The proprietary configuration delivers explanation and recommendation quality that is broadly comparable to the best open-source variant, indicating that our design does not rely on a single LLM family. The perturbation-based metrics show that the judge is sensitive to attribute-level degradations and usually prefers the original, higher-quality text over perturbed variants, suggesting robust alignment between scores and explanation quality.

The key takeaway is that SMARTTalk can provide high-quality, perturbation-resilient explanations and operator guidance across multiple LLMs, with explanation and recommendation scores around 4.4–4.6 *out of* 5 *from LLM-as-judge and perturbation robustness metrics above* 80%.

4.5 Ablation on Observation Window and Patch Length

We study sensitivity to observation window N and patch length L around the default setting. We fix $L=5$ and vary $N \in 10,20,30,40,50$, and fix $N=30$ and vary $L \in 2,4,5,10,15$. For each setting, we retrain patch encoders and PATTERN-MEMORY on training data and evaluate on a fixed test set.

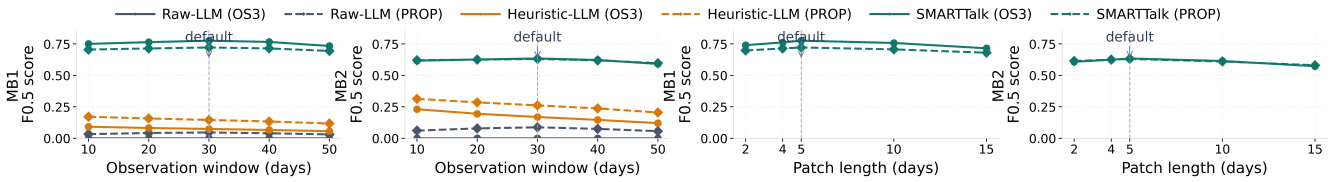


Figure 8: Sensitivity of $F_{0.5}$ to observation window N and patch length L on MB1 and MB2. Top: vary N with $L = 5$; bottom: vary L with $N = 30$. Solid lines show OS3 and dashed lines PROP; the vertical line marks the default setting. SMARTTalk consistently outperforms Raw-LLM and Heuristic-LLM. Performance is non-monotonic in both N and L, with the default near the best $F_{0.5}$ region across both models.

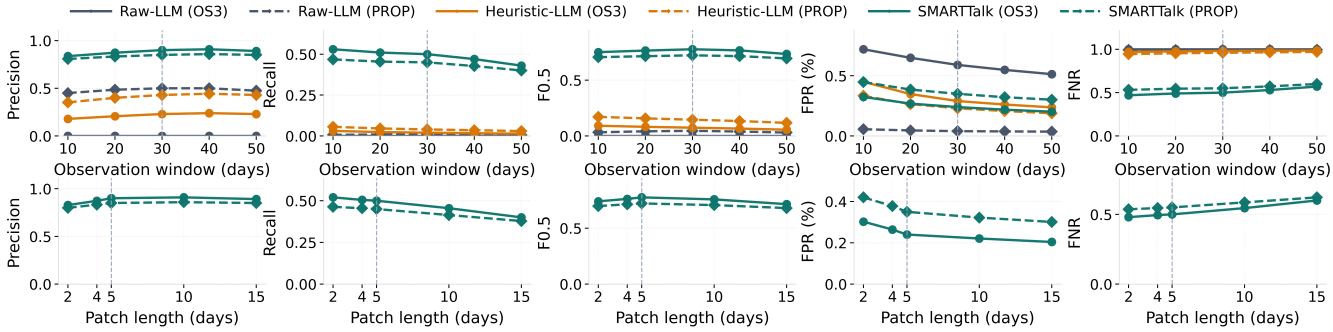


Figure 9: Detailed sensitivity metrics for MB1. Top row varies N with $L = 5$; bottom row varies L with $N = 30$. We report precision, recall, $F_{0.5}$, FPR, and FNR, with the default marked by a dashed line. Increasing N generally improves precision and FPR but may reduce recall due to smoothing of recent failures. Increasing L lowers FPR but raises FNR by smoothing short bursts. The default $(N = 30, L = 5)$ achieves the best precision–recall tradeoff.

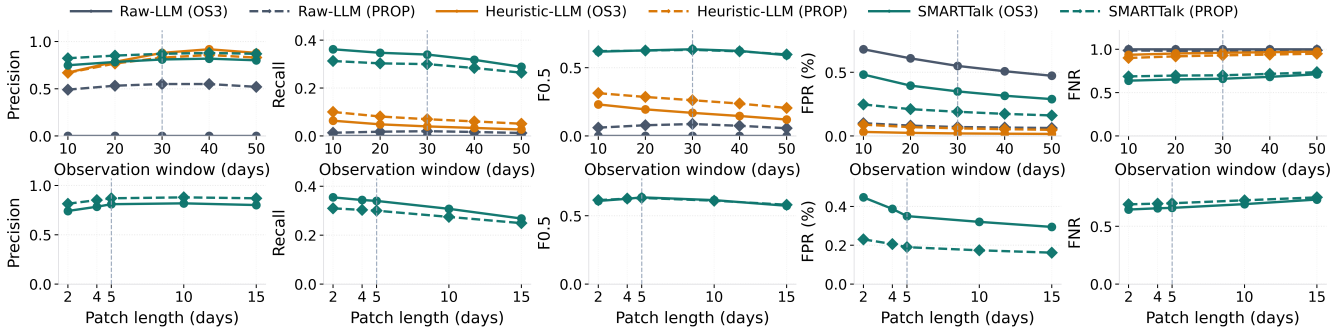


Figure 10: Detailed sensitivity metrics for MB2. Top row varies N with $L = 5$; bottom row varies L with $N = 30$ for SMARTTalk. MB2 exhibits the same pattern as MB1: performance is non-monotonic in N, and larger L reduces FPR but increases FNR. The default $(30, 5)$ remains in a stable high-performing region.

The default $(N = 30, L = 5)$ is highlighted by the vertical dashed line in the figures 8, 9 and 10.

Figure 8 shows that SMARTTalk is stable across a broad range of observation windows. On both MB1 and MB2, very short windows provide less historical context and can miss slow degradation patterns. Performance improves from $N = 10$ to 30 due to better temporal context, but saturates or degrades beyond 30 as older behavior dilutes recent failure signals. Thus, gains saturate around 30 days.

The patch-length study shows a similar tradeoff (Figures 9 and 10). Shorter patches improve temporal resolution and capture rapid spikes but increase noise, while longer patches smooth noise but can obscure failure bursts and reduce recall. The default $L = 5$ balances these tradeoffs, preserving local patterns while limiting noise, and achieves near-optimal $F_{0.5}$ on MB1 and MB2.

Encoding assumptions and information loss. SMARTTalk intentionally uses coarse temporal patches and phrase-level tokens rather than preserving every daily value. This design assumes that many useful SMART failure

signals appear as local trends, spikes, bursts, or cross-attribute co-occurrences in reliability-related fields, while small day-to-day fluctuations are often less useful for operator decisions. The compression can lose some fine-grained numeric detail, but our sensitivity results show this does not dominate performance. Our approach is therefore expected to generalize best when failures manifest through similar SMART trend shapes, but it may require recalibration for devices or telemetry streams where the meaningful signal is not concentrated in local extreme values, trends, or cross-attribute events.

The key takeaway is that SMARTTalk is not dependent on a fragile choice of N or L. Accuracy does not simply improve by increasing the observation window or patch length. Instead, there is a tradeoff: shorter windows and patches can miss longer degradation trends or become noisy, whereas longer windows and patches can smooth away recent failure bursts.

5 Related Work: LLMs for System Logs

Recent work explores how LLMs can assist system operations across many settings. For anomaly detection in scientific and workflow systems, LLMs have been used as supervised models and in-context learners over traces and configuration metadata [21], and more broadly as enablers for autonomic computing and self-managing services [42]. In storage and I/O optimization, ION uses LLMs to guide HPC I/O tuning and configuration [13], while LogLLM applies LLMs directly to system logs to highlight suspicious events and patterns [16]. LLMs have also been explored for root-cause analysis, incident management, and control in cloud environments, where single- and multi-agent designs couple LLM reasoning with log search, dashboards, and code inspection [29, 33]. For scheduling, control, and tuning, LLM-based methods guide policy learning, task coordination, key–value store tuning, and HPC I/O optimization [13, 31, 32, 39].

Within storage management, prior work uses LLMs to manage hybrid SSD tiers by selecting parameters and policies that balance performance and endurance [35]. SSD-specific LLM work has also modeled environmental effects and knowledge-graph-guided operational reasoning for SSDs [3, 4]. These efforts focus on high-level control, environmental what-if analysis, or graph-grounded operational reasoning rather than fine-grained SMART log analysis. In parallel, LogLLM and related systems operate on textual or semi-structured logs for anomaly detection and incident reasoning [16], but they do not target fine-grained SMART attributes or explicit time-to-failure prediction for individual drives.

SMARTTalk differs from prior LLM-for-system-logs work in three ways. First, it targets SSD SMART telemetry with a single pipeline for both health classification and bucketed time-to-failure prediction. Second, it introduces an attribute-aware *patch* $\rightarrow$ *pattern* $\rightarrow$ *phrase* layer that turns raw SMART windows into short, composable trend descriptions. Third, it maintains an adaptive pattern memory so LLM reasoning operates over an evolving library of interpretable SMART behaviors instead of raw logs.

6 Broader Impact

Although *SMARTTalk* focuses on SSD SMART telemetry, the same patch–pattern–LLM pipeline can be applied to a wide range of time-series system logs (e.g., HDD, GPU, CPU, memory, network, and power telemetry), helping operators surface emerging anomalies and interpret them in natural language. On top of SMARTTalk, one can build operator-facing applications such as what-if analysis tools, interactive dashboards, and exploratory diagnostics interfaces that let engineers probe alternative scenarios, inspect pattern-level histories, and compare failure modes across devices and clusters. Deployed as a decision-support layer alongside existing monitors and operator expertise, SMARTTalk encourages transparent, auditable use of LLMs in operations and opens the door to safer, more data-driven maintenance and capacity-planning practices.

7 Conclusion

This work asked whether LLMs can serve as effective reasoning engines for SSD SMART telemetry while preserving temporal structure. For **RQ1**, our results show that SMARTTalk can predict drive health and time-to-failure, without heavy feature engineering or large labeled datasets. For **RQ2**, we find that converting each n-day SMART window into per-attribute and cross-attribute patches, learning trend patterns, and verbalizing them as short natural-language summaries enables LLMs to correctly track temporal behavior while reducing trend hallucinations. For **RQ3**, the online pattern memory improves robustness to rare failures and emerging failure modes compared to a static offline model by detecting, clustering, and promoting novel patterns into new patterns. Finally, for **RQ4**, operator-facing evaluations indicate that SMARTTalk's explanations and its recommendations are aligned with operational needs. Together, these results suggest that pattern-based verbalization plus lightweight LLM reasoning is a promising path for interpretable, adaptive analysis of datacenter SMART logs. The implementation and artifact for SMARTTalk are publicly available at https://github.com/Damrl-lab/SMARTTalk.

Acknowledgments

We thank the reviewers for their constructive feedback and valuable suggestions. This research used resources of the National Energy Research Scientific Computing Center (NERSC), a DOE Office of Science User Facility supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC02-05CH11231, through NERSC award DDR-ERCAP0035598.

References

[1] Alibaba dataset: Ssd smart logs, 2020. [Online]. Available: https://github.com/alibaba-edu/dcbrain/blob/master/ssd_smart_logs/readme.md.

[2] Self-monitoring, analysis and reporting technology (smart), 2023. [Online]. Available: https://en.wikipedia.org/wiki/Self-Monitoring,_Analysis_and_Reporting_Technology.

[3] Mayur Akewar, Sandeep Madireddy, Dongsheng Luo, and Janki Bhimani. KORAL: Knowledge graph guided LLM reasoning for SSD operational analysis. In *Proceedings of the IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2026. [Online]. Available: https://arxiv.org/abs/2602.10246.

[4] Mayur Akewar, Gang Quan, Sandeep Madireddy, and Janki Bhimani. Can LLMs model the environmental impact on SSD? In *Proceedings of the 17th ACM Workshop on Hot Topics in Storage and File Systems (HotStorage '25)*, pages 100–106. ACM, 2025. [Online]. Available: https://doi.org/10.1145/3736548.3737835.

[5] Jacob Alter, Ji Xue, Alma Dimnaku, and Evgenia Smirni. Ssd failures in the field: Symptoms, causes, and prediction models. In *SC '19: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–14, 2019. [Online]. Available: https://doi.org/10.1145/3295500.3356172.

[6] Randall Balestriero, Mark Ibrahim, Vlad Sobal, Ari Morcos, Shashank Shekhar, Tom Goldstein, Florian Bordes, Adrien Bardes, Gregoire Mialon, Yuandong Tian, Avi Schwarzschild, Andrew Gordon Wilson, Jonas Geiping, Quentin Garrido, Pierre Fernandez, Amir Bar, Hamed Pirsiavash, Yann LeCun, and Micah Goldblum. A cookbook of self-supervised learning. In *arXiv preprint arXiv:2304.12210*, 2023. [Online]. Available: https://arxiv.org/abs/2304.12210.

[7] Ajinkya S. Bankar, Shi Sha, Janki Bhimani, Vivek Chaturvedi, and Gang Quan. Thermal aware system-wide reliability optimization for automotive distributed computing applications. *IEEE Transactions on Vehicular Technology*, 71(10), 2022. [Online]. Available: https://doi.org/10.1109/TVT.2022.3185978.

[8] Janki Bhimani, Adnan Maruf, Ningfang Mi, Rajinikanth Pandurangan, and Vijay Balakrishnan. Auto-tuning parameters for emerging multi-stream flash-based storage drives through new i/o pattern generations. *IEEE Transactions on Computers*, 71(2):260–271, 2022. [Online]. Available: https://doi.org/10.1109/TC.2020.3048303.

[9] Chandranil Chakraborttii and Heiner Litz. Improving the accuracy, adaptability, and interpretability of ssd failure prediction models. In *SoCC '20: Proceedings of the 11th ACM Symposium on Cloud Computing*, pages 120–133, 2020. [Online]. Available: https://doi.org/10.1145/3419111.3421300.

[10] Lei Chen, Zongpeng Zhu, Anyu Li, Najmeh Mashhadi, Robert Frickey, Jinhe Ye, and Xin Guo. Ssd drive failure prediction on alibaba data center using machine learning. *IEEE International Memory Workshop (IMW)*, pages 1–4, 2022. [Online]. Available: https://doi.org/10.1109/IMW52921.2022.9779284.

[11] Ting Chen, Simon Kornblith, Mohammad Norouzi, and Geoffrey Hinton. A simple framework for contrastive learning of visual representations. In *Proceedings of the 37th International Conference on Machine Learning*, pages 1597–1607, 2020. [Online]. Available: https://proceedings.mlr.press/v119/chen20j.html.

[12] Florin Cuconasu, Giovanni Trappolini, Federico Siciliano, Simone Filice, Cesare Campagnano, Yoelle Maarek, Nicola Tonellotto, and Fabrizio Silvestri. The power of noise: Redefining retrieval for rag systems. In *SIGIR '24: Proceedings of the 47th International ACM SIGIR Conference on Research and Development in Information Retriev*, 2024. [Online]. Available: https://doi.org/10.1145/3626772.3657834.

[13] Chris Egersdoerfer, Arnav Sareen, Jean Luca Bez, Suren Byna, and Dong Dai. Ion: Navigating the hpc i/o optimization journey using large language models. In *HotStorage '24: Proceedings of the 16th ACM Workshop on Hot Topics in Storage and File Systems*, pages 86–92, 2024. [Online]. Available: https://doi.org/10.1145/3655038.3665950.

[14] Nazanin Fouladgar, Marjan Alirezaie, and Kary Främling. Metrics and evaluations of time series explanations: An application in affect computing. *IEEE Access*, 10:23995–24009, 2022. [Online]. Available: https://doi.org/10.1109/ACCESS.2022.3155115.

[15] Jean-Bastien Grill, Florian Strub, Florent Altché, Corentin Tallec, Pierre H. Richemond, Elena Buchatskaya, Carl Doersch, Bernardo Avila Pires, Zhaohan Daniel Guo, Mohammad Gheshlaghi Azar, Bilal Piot, Koray Kavukcuoglu, Rémi Munos, and Michal Valko. Bootstrap your own latent: A new approach to self-supervised learning. In *Advances in Neural Information Processing Systems 33*, pages 21271–21284, 2020. [Online]. Available: https://proceedings.neurips.cc/paper/2020/hash/

f3ada80d5c4ee70142b17b8192b2958e-Abstract.html.

[16] Wei Guan, Jian Cao, Shiyou Qian, Jianqi Gao, and Chun Ouyang. Logllm: Log-based anomaly detection using large language models. In *arXiv*, 2024. [Online]. Available: https://arxiv.org/abs/2411.08561.

[17] Shujie Han, Patrick P. C. Lee, Fan Xu, Yi Liu, Cheng He, and Jiongzhou Liu. An in-depth study of correlated failures in production ssd-based data centers. In *19th USENIX Conference on File and Storage Technologies (FAST '21)*, 2021. [Online]. Available: https://www.usenix.org/conference/fast21/presentation/han.

[18] Guozhi Hao, Jun Wu, Qianqian Pan, and Rosario Morello. Quantifying the uncertainty of llm hallucination spreading in complex adaptive social networks. In *Nature, Scientific Reports*, 2024. [Online]. Available: https://www.nature.com/articles/s41598-024-66708-4.

[19] Wenwen Hao, Ben Niu, Yin Luo, Kangkang Liu, and Na Liu. Improving accuracy and adaptability of ssd failure prediction in hyper-scale data centers. *SIGMETRICS Performance Evaluation Review*, 49(4):99–104, 2022. [Online]. Available: https://doi.org/10.1145/3543146.3543169.

[20] Kaiming He, Haoqi Fan, Yuxin Wu, Saining Xie, and Ross Girshick. Momentum contrast for unsupervised visual representation learning. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 9729–9738, 2020. [Online]. Available: https://openaccess.thecvf.com/content_CVPR_2020/html/He_Momentum_Contrast_for_Unsupervised_Visual_Representation_Learning_CVPR_2020_paper.html.

[21] Hongwei Jin, George Papadimitriou†, Krishnan Raghavan, Pawel Zuk, Prasanna Balaprakash, Cong Wang, Anirban Mandal, and Ewa Deelman. Large language models for anomaly detection in computational workflows: From supervised fine-tuning to in-context learning. In *SC '24: Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analys*, 2024. [Online]. Available: https://doi.org/10.1109/SC41406.2024.00098.

[22] Qiang Li, Hui Li, and Kai Zhang. A survey of ssd lifecycle prediction. In *2019 IEEE 10th International Conference on Software Engineering and Service Science (ICSESS)*, 2019. [Online]. Available: https://doi.org/10.1109/ICSESS47205.2019.9040759.

[23] Stathis Maneas, Kaveh Mahdaviani, Tim Emami, and Bianca Schroeder. A study of ssd reliability in large scale enterprise storage deployments. In *Proceedings of the 18th USENIX Conference on File and Storage Technologies (FAST '20)*, 2020. [Online]. Available: https://www.usenix.org/conference/fast20/presentation/maneas.

[24] Stathis Maneas, Kaveh Mahdaviani, Tim Emami, and Bianca Schroeder. Operational characteristics of ssds in enterprise storage systems: A large-scale field study. In *Proceedings of the 20th USENIX Conference on File and Storage Technologies (FAST '22)*, 2022. [Online]. Available: https://www.usenix.org/conference/fast22/presentation/maneas.

[25] Justin Meza, Qiang Wu, Sanjeev Kumar, and Onur Mutlu. A large-scale study of flash memory failures in the field. *ACM SIGMETRICS Performance Evaluation Review*, 43(1):177–190, 2015. [Online]. Available: https://doi.org/10.1145/2796314.2745848.

[26] OpenAI. Gpt-5.1 instant and gpt-5.1 thinking system card addendum. 2025. [Online]. Available: https://cdn.openai.com/pdf/4173ec8d-1229-47db-96de-06d87147e07e/5_1_system_card.pdf.

[27] Ravi Ranjan, Utkarsh Grover, Mayur Akewar, Xiaomin Lin, and Agoritsa Polyzou. Catrag: Functor-guided structural debiasing with retrieval augmentation for fair llms. *arXiv*, 2026. [Online]. Available: https://arxiv.org/abs/2603.21524.

[28] Ravi Ranjan, Utkarsh Grover, Xiaomin Lin, and Agoritsa Polyzou. Persa: Reinforcement learning for professor-style personalized feedback with llms. *arXiv*, 2026. [Online]. Available: https://arxiv.org/abs/2605.01123.

[29] Devjeet Roy, Xuchao Zhang, Rashi Bhave, Chetan Bansal, Pedro Las-Casas, Rodrigo Fonseca, and Saravan Rajmohan. Exploring llm-based agents for root cause analysis. In *FSE 2024: Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering*, 2024. [Online]. Available: https://doi.org/10.1145/3663529.3663841.

[30] Bianca Schroeder, Raghav Lagisetty, and Arif Merchant. Flash reliability in production: The expected and the unexpected. In *Proceedings of the 14th Usenix Conference on File and Storage Technologies (FAST '16)*, pages 67–80, 2016. [Online]. Available: https://www.usenix.org/conference/fast16/technical-sessions/presentation/schroeder.

[31] Xuhao Tang, Fagui Liu, Dishi Xu, Jun Jiang, Quan Tang, and Bin Wang. Llm-assisted reinforcement learning: Leveraging lightweight large language model capabilities for efficient task scheduling in multi-cloud environment. In *IEEE Transactions on Consumer Electronics*, 2025. [Online]. Available: https://doi.org/10.1109/TCE.2024.3524612.

[32] Viraj Thakkar, Madhumitha Sukumar, Jiaxin Dai, Kaushiki Singh, and Zhichao Cai. Can modern llms tune and configure lsm-based key-value stores? In *HotStorage '24: Proceedings of the 16th ACM Workshop on Hot Topics in Storage and File Systems*, pages 116–123, 2024. [Online]. Available: https://doi.org/10.1145/3655038.3665954.

[33] Zefan Wang, Zichuan Liu, Yingying Zhang, Aoxiao Zhong, Jihong Wang, Fengbin Yin, Lunting Fan, Lingfei Wu, and Qingsong Wen. Rcagent: Cloud root cause analysis by autonomous agents with tool-augmented large language models. In *arXiv*, 2023. [Online]. Available: https://doi.org/10.1145/3627673.3680016.

[34] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed H. Chi, Quoc V. Le, and Denny Zhou. Chain-of-thought prompting elicits reasoning in large language models. In *Proceedings of the 36th Conference on Neural Information Processing Systems (NeurIPS)*, 2022. [Online]. Available: https://arxiv.org/pdf/2201.11903.

[35] Qian Wei, Yi Li, Zehao Chen, Zhaoyan Shen, Dongxiao Yu, and Bingzhe Li. Managing hybrid solid-state drives using large language models. In *arXiv*, 2025. [Online]. Available: https://arxiv.org/abs/2503.13105.

[36] Erci Xu, Mai Zheng, Feng Qin, Yikang Xu, and Jiesheng Wu. Lessons and actions: What we learned from 10k ssd-related storage system failures. In *Proceedings of the 2019 USENIX Conference on Usenix Annual Technical Conference (USENIX ATC '19)*, pages 961–975, 2019. [Online]. Available: https://www.usenix.org/conference/atc19/presentation/xu.

[37] Fan Xu, Shujie Han, Patrick P. C. Lee, Yi Liu, Cheng He, and Jiongzhou Liu. General feature selection for failure prediction in large-scale ssd deployment. In *2021 51st Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2021. [Online]. Available: https://doi.org/10.1109/DSN48987.2021.00039.

[38] Gala Yadgar, Moshe Gabel, Shehbaz Jaffer, and Bianca Schroeder. Ssd-based workload characteristics and their performance implications. *ACM Transactions on Storage (TOS)*, 17(1):1–26, 2021. [Online]. Available: https://doi.org/10.1145/3423137.

[39] Tingting Yang, Ping Feng, Qixin Guo, Jindi Zhang, Xiufeng Zhang, and Jiahong Ning. Autohma-llm: Efficient task coordination and execution in heterogeneous multi-agent systems using hybrid large language models. In *IEEE Transactions on Cognitive Communications and Networking*, 2025. [Online]. Available: https://doi.org/10.1109/TCCN.2025.3528892.

[40] Yuqi Zhang, Wenwen Hao, Ben Niu, Kangkang Liu, Shuyang Wang, Na Liu, Xing He, Yongwong Gwon, and Chankyu Koh. Multi-view feature-based ssd failure prediction: What, when, and why. In *Proceedings of the 21st USENIX Conference on File and Storage Technologies (FAST '23)*, 2023. [Online]. Available: https://www.usenix.org/conference/atc24/presentation/zhang-yuqi.

[41] Yuqi Zhang, Tianyi Zhang, Wenwen Hao, Shuyang Wang, Na Liu, Xing He, Yang Zhang, Weixin Wang, Yongguang Cheng, Huan Wang, Jie Xu, Feng Wang, Bo Jiang, Yongwong Gwon, Jongsung Na, Zoe Kim, and Geunrok Oh. Msfrd: Mutation similarity based ssd failure rating and diagnosis for complex and volatile production environments. In *Proceedings of the 2024 USENIX Annual Technical Conference*, 2024. [Online]. Available: https://www.usenix.org/system/files/fast23-zhang-yuqi.pdf.

[42] Zhiyang Zhang, Fangkai Yang, Xiaoting Qin, Jue Zhang, Qingwei Lin, Gong Cheng, Dongmei Zhang, Saravan Rajmohan, and Qi Zhang. The vision of autonomic computing: Can llms make it a reality? In *arXiv*, 2024. [Online]. Available: https://doi.org/10.48550/arXiv.2407.14402.

[43] Lianmin Zheng, Wei-Lin Chiang, Ying Sheng, Siyuan Zhuang, Zhanghao Wu, Yonghao Zhuang, Zi Lin, Zhuohan Li, Dacheng Li, Eric P. Xing, Hao Zhang, Joseph E. Gonzalez, and Ion Stoica. Judging LLM-as-a-judge with MT-bench and chatbot arena. *NIPS '23: Proceedings of the 37th International Conference on Neural Information Processing System*, 2023. [Online]. Available: https://dl.acm.org/doi/10.5555/3666122.3668142.

A Additional Paper Details

A.1 Explanations and Recommendations

A.1.1 Perturbation Method

Algorithm 1 summarizes our perturbation-based sanity check for the LLM-as-judge scores. In Step (P-A) we start from a set of windows for which we already have the method's original prediction, explanation, and recommendations, and we obtain baseline `ExpScore` and `RecScore` by querying the judge

prompt once per window. Step (P-B) then applies a library of perturbation operators (e.g., deleting mentions of key error attributes, flipping the trend description, or replacing the explanation with generic boilerplate) to produce systematically degraded or altered variants of the original explanation and recommendations. In Step (P-C) we feed these perturbed variants to the same judge prompt, keeping the SMART summary and predicted status/TTF fixed, and record the new scores. Step (P-D) compares perturbed scores against the baseline for each operator to measure sensitivity, for example how often a quality-degrading perturbation lowers the score. Finally, Step (P-E) aggregates these differences across windows and operators to report summary robustness metrics used in our evaluation.

Algorithm 1 Perturbation-based evaluation of explanations and recommendations

Require: Set of evaluated windows $\mathcal{W}$, each with SMART summary S_w, method prediction (y_w, TTF_w), original explanation E_w, original recommendations R_w; perturbation operators O; judge prompt template P_{judge}

Ensure: For each operator $o \in O$: perturbation metrics over $\mathcal{W}$

1: **(P-A) Collect baseline judge scores:** For each window $w \in \mathcal{W}$, query the LLM-as-judge with $(S_w, y_w, \mathrm{TTF}_w, E_w, R_w)$ using P_{judge} and record baseline scores $\mathrm{ExpScore}_w^{\text{base}}$ and $\mathrm{RecScore}_w^{\text{base}}$.

2: **(P-B) Generate perturbed variants:** For each window w and each operator $o \in O$, apply o to (E_w, R_w) to obtain a perturbed pair (E_w^o, R_w^o). Examples include dropping mentions of key SMART attributes, flipping risk language (e.g., "low concern" $\rightarrow$ "serious concern"), or replacing the text with generic boilerplate.

3: **(P-C) Score perturbed variants with the judge:** For every (w, o) pair, query the same judge prompt with $(S_w, y_w, \mathrm{TTF}_w, E_w^o, R_w^o)$ and record $\mathrm{ExpScore}_w^o$ and $\mathrm{RecScore}_w^o$.

4: **(P-D) Compute sensitivity metrics:** For each operator $o \in O$, compute score differences $\Delta\mathrm{Exp}_w^o = \mathrm{ExpScore}_w^o - \mathrm{ExpScore}_w^{\text{base}}$ and $\Delta\mathrm{Rec}_w^o = \mathrm{RecScore}_w^o - \mathrm{RecScore}_w^{\text{base}}$ across all $w \in \mathcal{W}$, and summarise them via statistics such as mean drop, median drop, and the fraction of windows where $\mathrm{ExpScore}_w^o < \mathrm{ExpScore}_w^{\text{base}}$.

5: **(P-E) Summarise robustness across operators:** Aggregate the per-operator statistics into a table of perturbation metrics, highlighting whether the judge is (i) sensitive to quality-degrading edits and (ii) stable under neutral or minor edits. Use these metrics in our evaluation to validate the reliability of the LLM-as-judge scores.

Algorithm 2 HeuristicSMART-LLM baseline

Require: Raw SMART logs $\mathcal{D}$; window length n; attribute set $\mathcal{F}$; heuristic prompt template P_{heur}

Ensure: For each window: status label, TTF bucket, explanation, recommendations

1: **(H-A) Windowing:** Slice each drive history in $\mathcal{D}$ into n-day windows $M_{u,t} \in \mathbb{R}^{A \times n}$.

2: **(H-B) Per-attribute statistics:** For every window $M_{u,t}$ and attribute $a \in \mathcal{F}$, extract the 30-day vector $v_{u,t,a}$ and compute basic statistics (min, max, mean, first, last, delta, and late/early means).

3: **(H-C) Heuristic trend labelling:** Apply a hand-crafted rule set $\textsc{ClassifyTrend}(v_{u,t,a})$ to assign a coarse phrase $h_{u,t,a}$ (e.g., "mostly zero across 30 days", "sharp late spike") for each attribute.

4: **(H-D) Summary and prompt construction:** Concatenate the phrases $\{h_{u,t,a}\}_{a \in \mathcal{F}}$ into a textual summary $T_{u,t}$ and substitute $T_{u,t}$ into the heuristic LLM prompt template P_{heur} to form the final query $Q_{u,t}$.

5: **(H-E) LLM inference and parsing:** Send $Q_{u,t}$ to the LLM, parse the JSON response into status, TTF bucket, explanation, and recommendations, and store these outputs for later metric computation.

A.2 Heuristic LLM

Algorithm 2 summarizes our heuristic LLM pipeline. In Step (H-A) we first slice the raw SMART logs into 30-day windows so that each example contains the recent history of one drive. Step (H-B) computes simple statistics for every attribute in a window (minimum, maximum, mean, first and last value, delta, and late-versus-early means), which provide the inputs to the hand-crafted trend rules. In Step (H-C) a rule set CLASSIFYTREND maps these statistics to one coarse phrase per attribute (for example "mostly zero across 30 days" or "steady increase with strong late growth"), without any learned prototypes. Step (H-D) concatenates the attribute phrases into a short textual summary and injects it into the Heuristic LLM prompt, producing the final query for the LLM. Finally, Step (H-E) sends this query to the LLM, parses the returned JSON into status, time-to-failure bucket, explanation, and recommendations, and logs these outputs for evaluation.

B Artifact Appendix

B.1 Artifact Overview

We provide the SMARTTalk artifact to support independent inspection and reproduction of the main results in this paper. The artifact contains the implementation of the SMARTTalk pipeline, including SMART log preprocessing, N day window construction, patch extraction, offline pattern learning, phrase dictionary construction, PATTERNMEMORY, LLM based in-

ference, and evaluation scripts. The artifact is publicly available at:

https://github.com/Damrl-lab/SMARTTalk

The repository README gives the most up to date execution commands, data placement instructions, and troubleshooting notes. The appendix summarizes the artifact at a high level so that readers can understand what is included and how it connects to the paper.

B.2 Artifact Contents

The artifact is organized around the main stages of SMARTTalk. It includes: (i) preprocessing and split generation scripts for the Alibaba SMART dataset; (ii) configuration files for MB1 and MB2 experiments; (iii) offline training code for the patch encoders and pattern discovery; (iv) saved PATTERNMEMORY, phrase dictionaries, and cached model outputs; (v) prompt templates for SMARTTalk and the LLM baselines; and (vi) scripts for regenerating paper tables, figures, ablations, and artifact sanity checks.

The repository also includes sample processed data and cached outputs so that reviewers can run a lightweight check without downloading the full raw dataset or invoking live LLM APIs. Full end to end reproduction is also supported when the public Alibaba SMART logs and the required compute resources are available.

B.3 Hardware and Software Requirements

The quick artifact check can be run on a standard CPU machine with a recent Python environment. Rebuilding the full SMARTTalk pipeline from raw SMART logs is more expensive because it requires preprocessing large time series files, training CNN based patch encoders, rebuilding PATTERNMEMORY, and optionally running LLM inference. GPU acceleration is recommended for full offline training and local open model inference, although cached outputs allow the main paper tables to be regenerated without repeating all expensive steps.

The artifact provides environment specifications through the repository documentation, including Python package requirements, a Conda environment file, and Docker support. Live LLM inference requires either local model endpoints or API credentials, but the cached reproduction path does not require private API keys.

B.4 Getting Started

The fastest way to check the artifact is to run the quick reproduction script from the repository root:

```
bash scripts/07_reproduce/reproduce_quick.sh
```

This quick check validates the package structure, loads sample SMART windows, checks cached SMARTTalk artifacts, and regenerates representative evaluation outputs from cached state. It is intended as a smoke test rather than a full rerun of all experiments.

B.5 Reproducing the Main Results

The artifact supports three reproduction modes. First, the quick check uses bundled sample data and cached outputs for a lightweight validation. Second, the cached reproduction path regenerates the main paper tables and figures from saved predictions, pattern memory outputs, phrase dictionaries, and sampled test results:

```
bash scripts/07_reproduce/reproduce_from_cache.sh
```

Third, the full reproduction path reruns preprocessing, offline pattern learning, inference, evaluation, and ablation scripts from the public Alibaba SMART logs:

```
bash scripts/07_reproduce/reproduce_full.sh
```

The full path assumes that the raw Alibaba SMART logs have been downloaded and placed according to the repository documentation. It also assumes sufficient compute resources for CNN training and, when live inference is desired, access to either local LLM serving or external model APIs.

B.6 Expected Outputs

The artifact reproduces the main evaluation components reported in the paper: status prediction results with precision, recall, $F_{0.5}$, FPR, and FNR; time to failure bucket metrics; explanation and recommendation evaluation; sensitivity studies for observation window length N and patch length L; phrase dictionary and PATTERNMEMORY inspection; and selected ablation results. The default paper setting is $N = 30$ days and $L = 5$ days, while the sensitivity scripts evaluate multiple values around this default.

ARTIFACT EVALUATED
USENIX
AVAILABLE

Svalinn: Overload Control in Large-Scale Servers with Multiple Resource Bottlenecks

Bhaskar Pardeshi Peidi Song Ahmed Saeed
Georgia Tech

Abstract

Modern overload controllers treat application binaries as monoliths and react to aggregate performance, a misconception we call *the single-queue fallacy*. Real applications have diverse, data-dependent execution paths that stress different resources. Reacting to overall performance forces the controller to focus on the most bottlenecked resource while leaving others underutilized.

We present Svalinn, a modular overload controller designed to maximize utilization across multiple potential bottlenecks such as CPU, memory bandwidth, and contended locks. Svalinn separates throughput control and latency control. A credit-based admission controller regulates offered load to maximize a user-defined utility function. Per-bottleneck controllers then enforce latency targets using Active Queue Management (AQM) policies. While AQM is straightforward for resources with explicit software queues, managing memory-bandwidth-intensive operations is challenging due to the absence of such queues. To handle this case, we introduce `m_semaphore`, which adaptively limits the number of concurrent memory-bandwidth-intensive requests to achieve high memory-bandwidth utilization using the minimum necessary CPU cores. We integrate Svalinn into four applications and two runtimes and show that it improves goodput by up to 6.51× without compromising latency.

1 Introduction

Modern datacenter applications have strict performance and cost-efficiency requirements. Thus, applications attempt to maximize resource utilization while meeting strict Service Level Objectives (SLOs). Historically, packing has meant maximizing the utilization of CPU, measured as the percentage of active CPU time out of total elapsed time, and memory capacity, measured in bytes. However, as datacenter operators push server utilization to higher levels, other resources such as memory bandwidth, network bandwidth, and storage IOPS have become bottlenecks that limit overall server utilization [2, 21, 39]. In particular, a single application can produce requests with heterogeneous resource requirements (e.g., CPU, network, or memory bandwidth). Such bottlenecks can cause hard-to-debug packet drops [2, 3], receive livelock scenarios where a server is busy processing incoming requests instead of serving already received requests [9, 10, 37], and transaction failures [32]. To that end, overload control systems are employed to balance utilization and latency, maximizing the utilization of *all* resources while avoiding SLO violations.

Despite the heterogeneity in per-request resource requirements, modern admission controllers typically treat applications as a monolith, reacting to overall server performance (e.g., latency) [9, 31, 55, 59] or limiting the maximum number of requests in flight to a certain replica [50, 52, 56]. These approaches implicitly assume that all requests are constrained by the most congested resource on the server. We refer to this assumption as *the single-queue fallacy*. Under this false assumption, once the most congested resource becomes overloaded, aggregate signals such as end-to-end latency or requests in flight immediately reflect that state. As a result, the controller reduces admitted load, leaving uncongested resources stranded. We show in several applications that this behavior can lead to severe underutilization, corresponding to a loss of up to 83% in aggregate machine throughput.

The challenge of balancing the utilization of multiple resources is exacerbated by the fact that the resource requirements of a request can be data-dependent, and may only be precisely known during its execution. For example, a request to a cache server will be CPU-intensive when fetching a small value, memory-bandwidth-intensive when fetching a large value, and lock-intensive when accessing hot, lock-protected data. Thus, typical approaches of resource isolation [7, 19, 25, 41] are insufficient as they operate at the granularity of an application, not a request. Moreover, prioritization-based techniques fail because they assume prior knowledge of the resource requirements of each request. Finally, it is particularly challenging to manage resources that can be accessed implicitly from any part of the code (e.g., memory bandwidth). Intra-application isolation [22] and prioritization [13, 20, 29, 30] techniques conflate congestion of such resources and CPU congestion.

In this paper, we present Svalinn, a novel overload controller that can handle heterogeneous requests with diverse resource requirements. Svalinn can balance the utilization of multiple resources by separating throughput control from latency control. In particular, throughput control is performed by a credit-based admission controller that admits more load as long as a user-defined utility function improves. This utility function may be as simple as aggregate system throughput or as complex as a weighted combination of throughput, loss rate, and resource utilization. Separately, latency control is performed in a distributed fashion at each resource, based on the state of the resource at the time it is accessed. Latency con-

trol is particularly challenging for memory, which is accessed implicitly. To that end, we present the memory semaphore (`m_semaphore`), a lightweight API that limits the number of concurrent threads that can execute memory-bandwidth-intensive code paths. Developers can profile parts of their code and call `m_semaphore.try_wait()` conditionally based on the nature of the request (e.g., before reading a large value from memory). `m_semaphore` adaptively adjusts the number of concurrent threads allowed to consume memory bandwidth based on the observed memory bandwidth utilization using a Multi-Armed Bandit (MAB) algorithm. If a request cannot access a resource, it is queued or dropped, following user-defined Active Queue Management (AQM) policies.

We evaluated Svalinn using synthetic workloads and three real applications: Memcached, RocksDB, and DataFrame. To demonstrate the robustness and generality of Svalinn, we evaluate Svalinn on two different hardware setups. Moreover, we implement it in two runtimes: Shenango's runtime exemplifying a low-latency environment and Go exemplifying a runtime common to modern microservices. We compare Svalinn with SEDA [55], a latency-based overload controller that reacts to the overall performance of an application, and Protego [10], a state-of-the-art overload controller that reacts to a large number of bottlenecks, tailored for lock contention. We observe that Svalinn can improve the goodput of applications by up to 6.51× compared to SEDA without compromising latency. Compared to Protego, Svalinn improves the goodput by up to 6.49× for the same application.

The contributions of this paper are as follows:

- We present Svalinn, a novel overload controller that can balance the utilization of multiple resources by separating throughput control from latency control.
- We present `m_semaphore`, a lightweight API that allows developers to limit the number of concurrent threads accessing memory-bandwidth-intensive code paths.
- We evaluate Svalinn using a synthetic workload as well as by integrating it into three real-world applications: Memcached, RocksDB, and DataFrame.

Svalinn is available as open-source software for Shenango at github.com/GT-ANSR-Lab/svalinn-shenango and for Go at github.com/GT-ANSR-Lab/svalinn-go.

2 Background and Motivation

2.1 Problem Context

Modern applications are typically broken into microservices, each exposing a large number of entrypoints (APIs) [26]. Each of these entrypoints may have different resource requirements. For example, a data processing service may have some APIs that are CPU-intensive, while others are memory-bandwidth-intensive. Both types of requests can be processed by the same binary and share the same pool of resources. However, the resource requirements and usage patterns of each request type are different. Moreover, the same request type can be bottlenecked on a different resource based on the size of its working set or the data it requires.

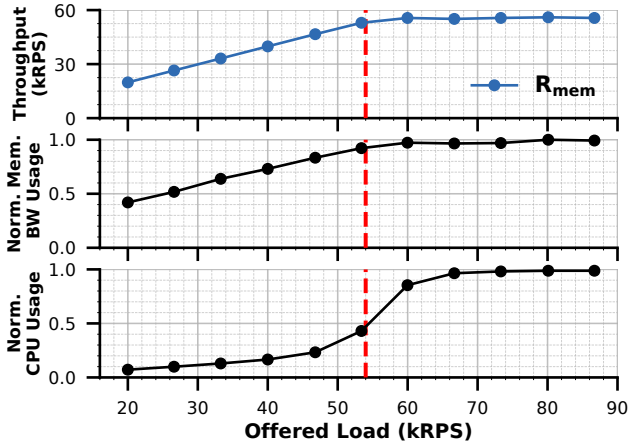


Figure 1: Throughput, normalized memory-bandwidth usage, and normalized CPU usage when the application receives only memory-intensive requests (R_{mem}). The red line marks the load at which throughput and memory bandwidth saturate, and indicates the minimum CPU required to reach peak throughput and memory-bandwidth utilization.

Existing overload controllers typically ignore request heterogeneity and treat all traffic uniformly. They operate under a common assumption, which we call *the single-queue fallacy*, that a single, shared bottleneck queue constrains all requests. Consequently, congestion at this bottleneck triggers a global load reduction, even when underutilized resources could safely serve a subset of incoming requests.

The single-queue fallacy manifests in two distinct ways. First, some controllers react to aggregate server-level signals, such as end-to-end latency [31, 32, 55] or requests-in-flight [50, 52, 56], without distinguishing which specific resource is overloaded. Second, others react exclusively to a fixed resource, such as CPU [9, 59] or storage [21, 35]. Consequently, both approaches prematurely throttle overall machine load when one resource peaks, failing to exploit idle capacity elsewhere. Furthermore, fixed-resource controllers suffer from the additional vulnerability of being entirely blind to congestion in unmonitored resources.

2.2 Modern Multi-Resource Overload Control

To illustrate the impact of the single-queue fallacy, consider a service with two types of requests: R_{cpu} is CPU-intensive and R_{mem} is memory-intensive.[1] For simplicity, each request type in this example performs a completely different set of operations and does not share any common code paths. R_{cpu} requests perform square root computations for 25μs, while the R_{mem} requests perform non-temporal stores on a 32KB buffer,

[1] We will use memory-intensive as a shorthand for memory-bandwidth-intensive for the rest of the paper.

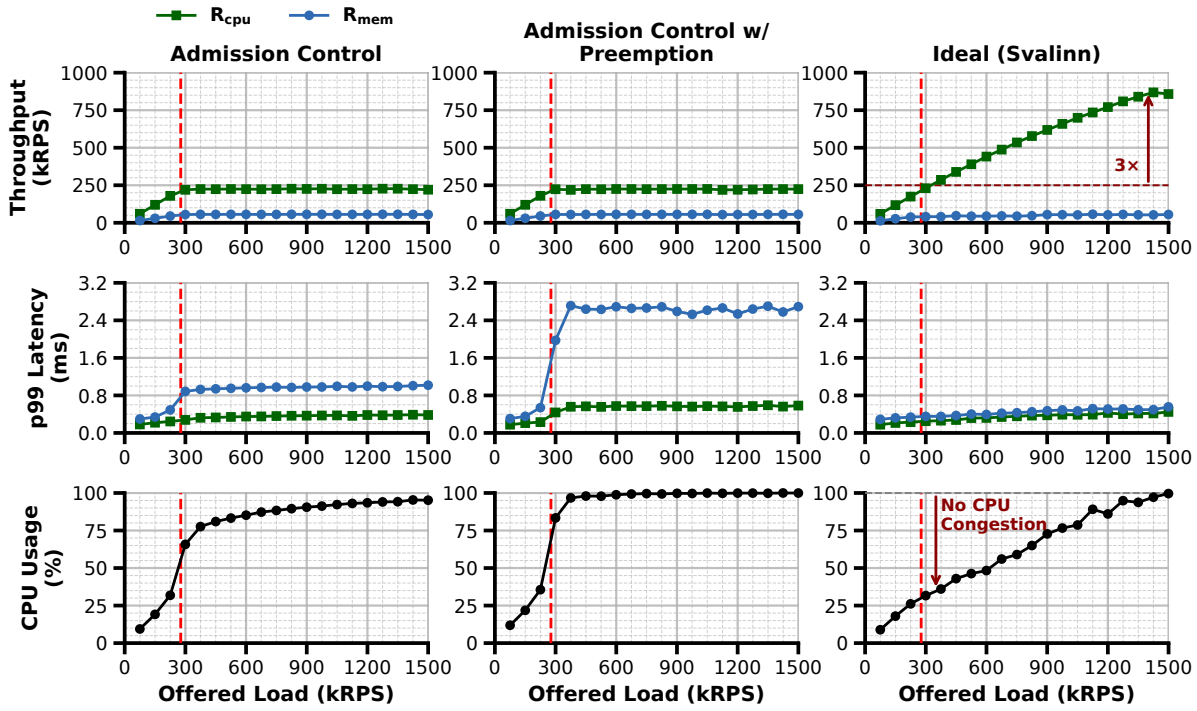


Figure 2: Throughput, tail latency, and CPU usage for an application receiving two types of requests: a CPU-intensive request (R_{cpu}) and a memory-intensive request (R_{mem}). The first column shows performance with an admission controller. The second column shows performance with an admission controller and a preemption-based scheduler. The third column shows the ideal or expected performance of the application. The red line marks the incoming load at which memory bandwidth saturates.

leading to a service time of 100μs. We run this example in experimental SETUPB, discussed in Section 5.1.

We start by profiling the machine capacity for each request type. R_{cpu} requests can achieve a maximum throughput of 1M requests per second (RPS) when a client issues only that type of request, while the R_{mem} requests can achieve a maximum throughput of roughly 55K RPS. R_{mem} requests can saturate the memory bandwidth of the machine while using 50% of the cores. On the other hand, R_{cpu} requests consume a negligible fraction of the memory bandwidth of the machine. Figure 1 shows the performance of the system when clients only issue R_{mem} requests at different offered load values. At low load, CPU utilization and memory bandwidth utilization grow linearly with load. As the system becomes bottlenecked on memory bandwidth, CPU utilization grows superlinearly with load, reflecting the fact that different threads are competing for memory bandwidth, leading to excessive stalling that manifests as high CPU utilization. It is obvious from this experiment that in such scenarios, a replica will seem fully utilized when, in fact, the CPU is underutilized.

Per-replica overload control. Next, we mix the two request types by having clients generate a load that is made of 80% R_{cpu} and 20% R_{mem} requests. We vary the total offered load to the service and measure overall throughput, tail latency, and CPU and memory bandwidth utilization. To keep latency bounded, we rely on a simple admission-control mechanism that limits the number of requests in flight based on the observed overall latency of the service [9, 55, 59]. Results are shown in the first column of Figure 2. We observe that the overall throughput peaks at around 300K RPS.

At first glance, this behavior seems reasonable. In particular, resource usage at peak throughput is high: CPU utilization is between 60-90% and memory bandwidth utilization is near 100%. However, consider that the throughput of R_{cpu} requests plateaus when R_{mem} requests saturate the memory bandwidth of the machine. At that point, we observe that CPU utilization grows superlinearly with the offered load, leading to a situation where the CPU becomes the bottleneck of the system. This behavior is due to the fact that R_{mem} requests consume CPU cycles stalling due to memory bandwidth congestion, which lowers the effective CPU capacity available to serve R_{cpu} requests. As a result, the CPU becomes a bottleneck, although it is effectively underutilized. This suggests that if R_{mem} requests are restricted to the number of cores required to fully utilize memory bandwidth, they do not stall excessively because their aggregate memory bandwidth demand no longer congests memory bandwidth. The remaining cores can then be used by R_{cpu} requests, enabling up to 3× higher R_{cpu} goodput, as shown in the third column of Figure 2.

The challenge of this scenario is exacerbated by the fact that

memory bandwidth is a resource accessed implicitly through LOAD and STORE instructions, not through an explicit queue. Unlike network, disk, and locks, which have their own queues, enabling monitoring and control [3, 10, 21, 32, 38], memory bandwidth is shared implicitly among all requests running on the machine. Access to hardware-level queues is not always feasible and can complicate the design and operation of applications. Thus, simple extensions to existing admission control mechanisms that monitor and control explicit queues are not sufficient. Moreover, memory access is spread throughout the whole application, and high utilization requires simultaneous access by multiple threads within a single application. Thus, precise tracking of memory bandwidth consumption might require tracking the behavior of all the instructions executed by an application, an untenable objective.

Preemption-based scheduling. A natural question is whether effective multiplexing of shared resources can mitigate this problem. To test that hypothesis, we repeat the same experiment but enable preemption. In particular, we use Aspen, a lightweight interrupt-based preemption mechanism [20]. The objective of preemption in this scenario is to improve CPU utilization by preempting R_{mem} requests when they are stalling, replacing them with R_{cpu} requests. This choice comes at the expense of increased latency for R_{mem} requests and added context switching overhead. The results are shown in the second column of Figure 2. Surprisingly, we observe that only the preemption overhead manifests without any significant improvement in throughput.

Preemption does not help because it is based on the assumption that multiplexing the use of resources leads to higher overall utilization. However, in this scenario, we do not only need to multiplex resources; we need to limit the resource usage of a specific type of resource by a specific type of request. In particular, R_{mem} requests in this example saturate the memory bandwidth of the machine using only 24 cores out of the available 45 cores. Thus, any R_{mem} request that is scheduled on the remaining cores will still waste CPU cycles. Moreover, preemption does not differentiate between different types of requests, nor does it react to specific resource usage patterns. As a result, preemption ends up being ineffective as it does not reduce the amount of wasted CPU cycles. Another downside of preemption is that it might be harmful if the service time of R_{cpu} requests is longer than the service time of R_{mem} requests.

Atropos [23], a recent overload system, attempts to address this gap through targeted task cancellation, judiciously aborting requests to maximize resource release. However, this approach relies on the assumption that congestion is driven by a small number of disproportionately heavy requests, a condition that does not hold in the scenario described above. Furthermore, Atropos is limited in its scope: it neither supports memory bandwidth regulation nor provides a framework for proactive admission control.

Inter-application interference vs. intra-replica resource contention. While in this example the two request types can be separated into their own binaries, which can be allocated separate resources and isolated from each other using OS-supported primitives [19, 25, 41, 44], this approach is not always feasible. First, separating request types into different binaries increases the operational complexity of the service, as it requires managing multiple deployments and scaling policies. Second, in many cases, different request types share significant portions of the code path, making it impractical to separate them into different binaries without code duplication. The difference can even lie in their working set size, which would make it impossible to separate them without significant code changes. Finally, separating request types into different binaries can lead to resource fragmentation, as resources allocated to one binary may be underutilized while other binaries are overloaded. Thus, there is a need for an admission control mechanism that can effectively manage the resource usage of different request types within the same binary, avoiding the single-queue fallacy.

2.3 Objectives and Challenges

The single-queue fallacy can lead to significant underutilization of resources in modern data centers, especially as applications become more complex and diverse. To address this issue, we propose Svalinn, an admission control mechanism that aims to maximize the utilization of all resources in a system without affecting latency. Such an ambitious objective requires Svalinn to be extensible, allowing the incremental addition of new resource controllers as new bottlenecks are identified. Another key requirement is that Svalinn functionality should introduce minimal changes to the application logic. Thus, the realization of Svalinn requires overcoming the following challenges:

- **Nondeterministic behavior / unprofilable workloads.** Requests of the same type may have different resource requirements based on their input data or other dynamic factors. This variability makes it difficult to identify the resource requirements of a request before it actually starts to run on the server. Moreover, workload characteristics can change dynamically, requiring the system to adapt to such changes without reconfiguration.
- **Balancing performance objectives.** Maximizing utilization typically corresponds to an improvement in throughput. However, Svalinn should achieve that objective without increasing latency. Moreover, the fact that some requests will need to be executed in order to identify their resource requirements implies that some requests will be dropped after they start executing because of resource congestion. Thus, Svalinn must balance multiple performance objectives, including throughput, latency, and drop rate.
- **Minimal changes to application logic and added overhead.** Tracking and controlling resource usage of different request types within the same binary may require changes to the application logic. This challenge is significant given that memory access is typically performed implicitly through

`LOAD` and `STORE` instructions. Tracking such accesses may require changes to the application code or the use of specialized libraries or frameworks. However, such changes should be minimal to avoid increasing the complexity of the application and hindering adoption. Moreover, the added overhead of tracking and controlling resource usage should be minimal to avoid negating the benefits of improved resource utilization.

- **Scalability.** Modern data centers host services that handle millions of requests per second from hundreds or even thousands of clients. Thus, any admission control mechanism needs to be scalable to handle such high request rates and number of clients without becoming a bottleneck itself.

3 The Design of Svalinn

We design Svalinn as a modular system that separates throughput and latency control. Each of the controllers is designed to be configurable to meet the requirements of the application. The modular design ensures extensibility and minimizes the amount of changes needed to application logic.

Throughput is controlled through a credit-based admission controller. A single credit indicates that the client can send one request to the server. The total number of issued credits is determined based on a user-defined utility function that balances different performance objectives (e.g., throughput, drop rate, or utilization level of a specific resource). This user-defined utility function enables operators to customize application behavior according to their needs; however, the resulting admission decisions may be independent of the status of individual bottlenecks. Consequently, an admitted request can still incur high latency when it competes for a bottlenecked resource.

Latency is minimized by distributed controllers, each independently applying an Active Queue Management (AQM) policy to the queue of a single bottleneck. For example, a latency controller can drop requests at risk of violating SLOs. This modular design allows for extensibility, as latency controllers can be independently designed and incrementally deployed as new bottlenecks are detected.

We focus on memory bandwidth as a bottleneck that is accessed implicitly, building a latency controller for this unique bottleneck. In particular, we present the memory semaphore (`m_semaphore`) to limit the number of concurrent threads performing memory-intensive operations. The `m_semaphore` has two functions. First, it minimizes the number of CPU cycles wasted on stalling by only allowing a limited number of threads to concurrently consume memory bandwidth, maximizing memory bandwidth utilization without wasting CPU cycles. Second, it allows for building a queue of threads awaiting to consume memory bandwidth. That queue enables latency control by applying AQM policies, turning an implicit bottleneck to an explicit bottleneck. Our proposed design is shown in Figure 3.

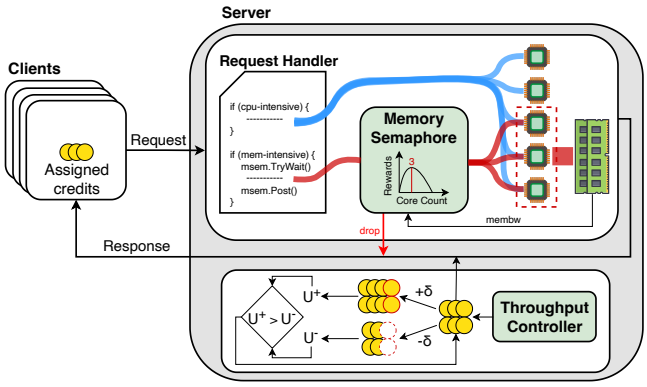


Figure 3: Svalinn system architecture.

3.1 Utility-Based Throughput Control

The throughput controller admits requests to optimize a user-defined utility function that captures different performance metrics (e.g., throughput, drop rate, resource utilization). The controller optimizes the utility function by exploring different levels of admitted load through micro-experiments, adapting to changing workload characteristics over time. Our approach is inspired by innovations in network congestion control, particularly the PCC family of algorithms [14, 15]. Unlike recent proposals that use fixed heuristics to regulate load based on low-level signals (e.g., queueing delay [9] or efficiency [10]), our controller admits general convex utility functions. A server regulates incoming load by controlling the size of a single credit pool (i.e., the maximum number of credits it can issue at a given time). In other words, different request types use the same credits which are issued indiscriminately to clients. When a client wishes to issue a request, it first reports its demand to the server. The server replies with credits, if available, and the client may only send a request if it has sufficient credits. The controller attempts to identify the credit pool size that maximizes the user-defined utility function through micro-experiments.

A micro-experiment is a short test interval during which the controller evaluates the application's performance under a perturbed credit-pool size. In each control cycle, the controller runs two micro-experiments: a credit-increase experiment, denoted by δ^+, and a credit-decrease experiment, denoted by δ^-. During each micro-experiment, the controller either increases or decreases the credit pool by δ credits, monitors the application for $\Delta\, t_{monitor}$ interval, and records relevant performance metrics. Monitored metrics include the number of received requests, the number of successfully sent responses, the number of dropped requests, queueing delay, and the duration of the experiment. The controller calculates utility values $utility_{\delta^+}$ and $utility_{\delta^-}$ for the two micro-experiments. After both micro-experiments finish, the controller increases the pool size if experiments show that $utility_{\delta^+}$ is higher than $utility_{\delta^-}$, or vice versa. This process repeats continuously.

While this micro-experiment framework is conceptually sound, realizing it in practice introduces a key complication: increasing the credit pool size does not guarantee a higher request arrival rate when clients are not backlogged. Because a credit represents only permission to send a request, the actual arrival rate during a δ^+ experiment can inadvertently fall below that of a δ^- experiment. In such cases, if $utility_{\delta^+}$ exceeds $utility_{\delta^-}$, the controller would incorrectly attribute the higher utility to the larger pool size and expand it further. To address this, we leverage the insight that the controller ultimately cares about the request arrival rate, for which the credit pool size is merely a proxy. When the arrival rate during the δ^+ experiment is lower than that of the δ^- experiment, we swap their metrics, computing $utility_{\delta^+}$ using the higher-arrival data and vice versa. This relabeling ensures the controller evaluates the perturbations against actual traffic variations rather than nominal pool adjustments.

Compounding the issue of accurate utility computation is the physical propagation delay between modifying the credit pool and observing its impact on the incoming traffic rate. This propagation delay can be comparable to individual request execution times. For instance, a network RTT of ten microseconds is on par with a key-value store request processing time of a few microseconds. Consequently, the controller must enforce a warmup interval ($\Delta\, t_{warmup}$) immediately following each pool adjustment. This allows the new configuration to stabilize across the network before the controller attributes any subsequent performance metrics to the perturbation. The complete controller algorithm is detailed in Appendix A.1.

A final challenge remains at low to medium loads: clients may hold onto issued credits indefinitely without utilizing them. If the server strictly limits the total issued credits to its exact processing capacity, this credit retention prevents the expected load from fully materializing, leaving the server underutilized. To bridge this gap between nominal credit allocation and actual resource utilization, the controller safely overcommits credits. This ensures the server pipeline remains fully saturated even when a subset of clients retains their credits, mirroring the overcommitment strategies employed by other receiver-driven, credit-based transport and admission control systems [9, 10, 38].

3.2 Making Memory Bandwidth Access Explicit Using `m_semaphore`

Our objective is simple: determine the minimum number of concurrently executing memory-intensive requests required to saturate the available memory bandwidth. While the objective is simple, achieving it in practice is challenging due to the dynamic nature of real workloads and their constituent requests. As demand and request mix evolve over time, the optimal concurrency level changes, necessitating a mechanism that can adjust this level dynamically and be applied selectively to specific requests. To address these challenges, we introduce `m_semaphore`, a semaphore whose capacity is dynamically adjusted based on the available memory bandwidth and the per-request memory bandwidth demand. The `m_semaphore` is used by developers to protect access to memory-intensive sections of their code. Requests that cannot acquire `m_semaphore` are held outside the section, making access to memory bandwidth explicit and moving excess memory-intensive requests into an explicit queue. `m_semaphore` is implemented as a singleton, so all instrumented sections share the same underlying semaphore instance. We discuss how developers identify and instrument such sections in Section 3.4.

Algorithm 1: `m_semaphore` Capacity Controller.

$cores_{current}$: Current `m_semaphore` capacity
$cores_{max}$: Maximum possible `m_semaphore` capacity
$membw_{max}$: Maximum memory bandwidth seen so far
$R_{avg}[cores_{max}]$: Average reward at each core count
τ_{last}: Last time $cores_{current}$ was updated
Parameter α: Weight given to memory bandwidth usage over CPU usage in reward computation
Parameter ω: Weight assigned to the latest reward sample when updating each core's average reward
Parameter ε: Probability of exploration
Parameter $\Delta\, t_{msem}$: Interval at which $cores_{current}$ is recomputed

Function `UpdateCapacity()`:
  `// Run once every control cycle`
  if $\text{now} - \tau_{last} < \Delta\, t_{msem}$ **then**
    return;
  `// Get memory bandwidth usage`
  $membw_{current} \leftarrow$ `GetMemoryBandwidthUsage()`;
  $membw_{max} \leftarrow \text{Max}(membw_{max}, membw_{current})$;
  `// Calculate reward for current capacity`
  $R \leftarrow \alpha \cdot \frac{membw_{current}}{membw_{max}} - (1-\alpha) \cdot \frac{cores_{current}}{cores_{max}}$;
  $R_{avg}[cores_{current}] \leftarrow \omega \cdot R + (1-\omega) \cdot R_{avg}[cores_{current}]$;
  `// Find best performing capacity`
  $cores_{best} \leftarrow \arg\max_{1 \le i \le cores_{max}} R_{avg}[i]$;
  `// Update the capacity`
  $u \sim$ `Uniform`$(0,1)$;
  if $u < \varepsilon$ **then**
    `// Explore`
    $explore_direction \leftarrow$ `RandomChoice`$(-1,+1)$;
    $cores_{current} \leftarrow cores_{best} + explore_direction$;
  else
    `// Exploit`
    $cores_{current} \leftarrow cores_{best}$;
  $\tau_{last} \leftarrow now$;

The semaphore dynamically adjusts its capacity using a Multi-Armed Bandit (MAB) controller (Algorithm 1). The MAB controller treats each possible semaphore capacity as an arm. Once an arm is pulled, the system runs with that configuration for $\Delta\, t_{msem}$ duration, during which we observe the resulting performance. At the end of this control cycle, we compute a reward associated with the chosen capacity and

update the average reward for that arm. Our objective is to identify the minimum semaphore capacity that can saturate memory bandwidth. Consequently, we design a reward function that increases with memory bandwidth utilization but penalizes CPU usage.

$$\alpha \cdot \frac{membw_{current}}{membw_{max}} - (1-\alpha) \cdot \frac{cores_{current}}{cores_{max}}$$

This function encourages configurations that achieve high memory bandwidth utilization with minimal CPU utilization. The parameter $\alpha \in [0,1]$ controls the balance between memory-bandwidth utilization and CPU utilization. We discuss the choice of our reward function and its sensitivity to the parameter α in Appendix B.3 and B.4 respectively.

The controller balances exploration and exploitation using an ε-greedy strategy [6, 48, 51]. In each control cycle, with probability $(1-\varepsilon)$, it selects the capacity with the highest reward, which under our reward design corresponds to high memory bandwidth utilization with minimal core usage. With probability ε, it explores an alternative capacity. Rather than exploring uniformly over all possible capacities, we restrict exploration to capacities adjacent to the current best one. This keeps exploration local, reduces performance noise, and avoids erratic jumps in concurrency.

Continued exploration is important because the workload properties can be non-stationary. For example, a key-value store can experience significant variation in the sizes of values being read, which changes the per-request demand for memory bandwidth, shifting the optimal capacity of the semaphore over time. A controller that only exploits past observations can therefore remain stuck at a capacity that was previously good but is no longer appropriate. We use ε-greedy because it provides this continued exploration explicitly. In contrast, more sophisticated MAB strategies, such as softmax, upper confidence bound, or Thompson sampling, are typically used in stationary settings, and their exploration naturally narrows over time. As a result, under regime changes they may not explore enough to reach the new operating point.

3.3 AQM at Bottleneck Queues

Distributed latency control is achieved by deploying AQM policies at each identified bottleneck. It is possible for a single request to face multiple bottlenecks along its execution path. Therefore, we avoid having a different per-bottleneck AQM policy configuration. Instead, all bottlenecks make AQM decisions using a single per-request property: queueing-delay budget, similar to Protego [10]. Specifically, each request is tagged with an SLO-derived queueing-delay budget when it enters the system. As the request traverses the execution path, latency controllers at subsequent bottlenecks dynamically compute the remaining budget by deducting the time the request spent waiting in each bottleneck's queue. If the remaining budget is less than the bottleneck's instantaneous queueing delay, the request is proactively dropped before being enqueued.

Handling dropped requests. The proposed approach implies that requests may be dropped after being partially executed. When a request is dropped, the server can issue a failure message to the client so that the client can issue the request to another replica of the application. Another potential approach is to have the server spillover the dropped request to another replica directly, similar to load balancing behavior in some sidecars [18]. Moreover, developers have to specify a "cleanup" callback function to be invoked when a request is dropped. This function allows developers to perform any necessary cleanup actions, such as releasing resources or reverting any state changes. Recent work shows that such cleanup support is already common in practice: 76% of 151 popular open-source applications support task cancellation and already provide handlers to clean up request state on abrupt termination [23]. Developers also have the option to label requests as "non-droppable." Such requests are not dropped and treated as high-priority requests by the AQM logic to minimize their queueing delay.

3.4 Svalinn Deployment Guidelines

Identifying bottlenecks. Explicitly accessed bottlenecks can be identified using profilers such as LDB [8] and GigiProfiler [24], which provide function-level latency information and help locate the code paths responsible for bottleneck-induced delay. For memory-intensive bottlenecks, developers can use tools such as Intel VTune [27], HiresPerf [33], or perf [34] to identify memory-intensive functions. These tools also expose the call paths leading to such functions, which helps determine where `m_semaphore` should be placed in the request execution path.

Instrumenting explicit bottlenecks. System designers expose resource contention points to the application through new APIs that serve as enforcement points for latency control. For system- or runtime-level bottlenecks, these APIs can be implemented by interposing the underlying system or runtime calls, as in Protego [10]. For application-internal bottlenecks, these APIs are instrumentation APIs that expose the relevant resource usage needed to estimate queueing delay and perform AQM [22, 23]. In both cases, developers update the application to use these APIs and incorporate custom cleanup logic to handle request drops.

Instrumenting memory-intensive bottlenecks. Developers leverage the runtime-managed singleton `m_semaphore` object to instrument the call path leading to the memory-intensive code region. This explicitly isolates the critical region and exposes a queue for AQM enforcement. In practice, such bottlenecks are rare; for instance, Memcached and RocksDB each require only a single instrumented call path, terminating at the `memcpy` that copies keys into the response buffer. Consequently, while explicit annotation is required, the modification scope remains minimal across our evaluated applications.

4 Implementation

We implement Svalinn in two runtimes: Shenango [41] and Go. Our Shenango implementation builds on the RPC layer originally implemented for Breakwater [9] and Protego [10]. Our Go implementation adds Svalinn to a bare-bones RPC layer. Both RPC layers follow the dispatcher model: a dispatcher thread parses incoming packets into requests and spawns a worker thread for each request. The throughput controller is implemented as an event-driven state machine and runs inline when each worker finishes executing a request. As a result, workers update controller state on the fast path, eliminating the need for dedicated control threads. For each micro-experiment, the controller records the number of received requests, completed responses, and dropped requests, along with queueing delay, energy consumed, memory bandwidth consumed, and experiment duration, and passes these metrics to a user-defined utility function.

`m_semaphore` is implemented as a C++ library with C bindings for Shenango applications, and as a separate Go library for Go-based applications. It exposes three methods: `try_wait()`, `wait_if_uncongested()`, and `post()`. Using `try_wait()` mimics a non-blocking semaphore and thus enforces an AQM policy that maintains a zero-length queue. It returns `true` if admitting the request to the memory-intensive section does not exceed the permitted concurrency, and `false` otherwise. In contrast, `wait_if_uncongested()` allows requests that exceed the permitted concurrency to be enqueued, provided they have sufficient remaining SLO-derived budget, and returns `true` if the request is enqueued and later granted access to the region, and `false` otherwise. By default, all workloads in our evaluation use `try_wait()`. The `UpdateCapacity()` routine is invoked inline with `try_wait()`, `wait_if_uncongested()`, and `post()`, but a capacity update takes effect only once per control cycle. `m_semaphore` reads memory bandwidth counters using Intel PCM [28]. Since fetching memory bandwidth usage can introduce noticeable overhead at small timescales, we move this operation off the request fast path. In Shenango, we modify the IOKernel scheduler, which runs every 5–10 μs, to collect memory bandwidth usage periodically. In Go, we dedicate a logical core to collect performance metrics, including memory bandwidth, once every 5 μs.

The utility-based throughput controller in Shenango extends the existing Breakwater [9, 10] RPC library and adds 1815 LoC. The `m_semaphore` library for Shenango contributes 961 LoC, and the modifications to the Shenango IOKernel needed to expose host memory bandwidth readings add another 631 LoC. Integrating the RPC library into applications required 1462 LoC for DataFrame and 1668 LoC for RocksDB. We reuse Memcached and the synthetic application from Protego, adding 1082 LoC and 812 LoC, respectively, to support heterogeneous request types. Each application required only 4–10 LoC to wrap memory-intensive paths with `m_semaphore` and handle dropped requests.

The Go implementation of the admission-controller-enabled RPC library adds 5696 LoC. Exposing thread queueing delay from the Go runtime required 93 LoC in SysMon. We add a perf goroutine pinned to a logical core that periodically refreshes memory bandwidth usage, thread queueing delay in the Go runtime, and TCP RX buffer delay, which adds 1502 LoC. The Go version of `m_semaphore` adds 454 LoC, and the Go synthetic application adds 961 LoC.

5 Evaluation

Our evaluation answers the following key questions:

1. Does Svalinn deliver high utilization and bounded latency across heterogeneous bottlenecks?
2. Can Svalinn improve application performance even when resources are overprovisioned?
3. Does Svalinn accommodate and optimize different user-defined utility functions?
4. How sensitive is Svalinn to retransmissions, dynamic workloads, parameter changes, and request mixes?
5. Do Svalinn's benefits persist across different machines and runtimes?

5.1 Evaluation Setup

Testbed. We evaluate Svalinn on two testbeds: SETUPA and SETUPB. SETUPA consists of eleven CloudLab [17] xl170 nodes, each equipped with a 10-core (20-logical-core) Intel E5-2640v4 @ 2.4 GHz, 64 GB ECC RAM, and a Mellanox ConnectX-4 25 GbE NIC. The memory bandwidth of each node is approximately 45 GB/s. Nodes are connected via a Mellanox 2410 switch, with an average RTT of 10 μs. All machines run Ubuntu 24.04 with Linux 6.8, and our applications use 18 logical cores per node. SETUPB consists of two machines, each with dual-socket Intel Xeon Gold 6442Y processors (24 local cores per socket @ 2.6 GHz), 128 GB ECC RAM, and Mellanox ConnectX-7 200 Gb/s NICs connected back-to-back. Each machine provides roughly 180 GB/s of memory bandwidth. We use a single socket per machine for the application and observe a 40 μs RTT. Both machines run Ubuntu 22.04 with Linux 6.0, and applications use 45 logical cores per machine. Unless otherwise noted, all applications run on Caladan [19] and are linked with its runtime. For Go-based tests, we run all experiments on SETUPA with Go runtime version 1.25.3. In both setups, one node acts as the server and the remaining nodes act as clients. We use an open-loop load generator based on a Poisson arrival process, with a total of 100 client threads distributed across the client machines. Each thread maintains one connection to the server and generates requests of different types according to the target mix. We disable TurboBoost, frequency scaling, and c-states as done in other works using similar systems [9, 10, 20, 41].

Workloads. We evaluate Svalinn using synthetic workloads and three real applications: Memcached [1], RocksDB [16], and Dataframe [36]. Each application implements up to

three request types: R_{cpu} (a CPU-intensive), R_{mem} (a memory-intensive), and R_{lock} (a mutex-intensive) request.

The synthetic workload represents applications whose requests have different resource requirements but comparable processing times.[2] In particular, an R_{cpu} request performs 7500 square-root computations in a loop. An R_{mem} request issues 25 non-temporal stores over a 32 KB buffer. An R_{lock} acquires a global mutex and executes 5000 square-root computations in a loop. On SETUPA, the average processing times are 40 μs for R_{cpu}, 80 μs for R_{mem}, and 40 μs for R_{lock}.

RocksDB and Memcached are widely used key-value stores in datacenter applications. In both, R_{cpu} requests issue `GET` operations on small values, with sizes drawn uniformly between 5 B and 1000 B, while R_{mem} requests issue `GET` operations on large values, with sizes drawn uniformly between 800 KB and 1200 KB. In Memcached, R_{lock} requests issue `SET` operations on small values using the same size distribution as R_{cpu}. Small values represent the common case of caching text objects, whereas large values represent the less common case of caching media objects. For Memcached, the average processing times are 15 μs for R_{cpu}, 20 μs for R_{lock}, and 275 μs for R_{mem} on SETUPA. For RocksDB, the average processing times are 20 μs for R_{cpu} and 230 μs for R_{mem} on SETUPA, and 40 μs for R_{cpu} and 230 μs for R_{mem} on SETUPB.

DataFrame is a C++ column store. We run an instance operating on a 370K-row EUR–USD FX dataset [11]; its R_{cpu} request computes the maximum of the *Closing* price column, while its R_{mem} request applies a decay-based smoothing operation to the same column and writes the smoothed values back. On SETUPA, the average processing times are 15 μs for R_{cpu} and 750 μs for R_{mem}.

In the Go-based synthetic application, the R_{cpu} request performs 1000 square-root computations in a loop, while the R_{mem} request performs stores to a 1 MB in-memory buffer. On SETUPA, the average processing times are 200 μs for R_{cpu} and 462 μs for R_{mem}.

Baselines: We compare Svalinn against SEDA [55] and Protego [10]. We choose these baselines because SEDA represents the classic approach of using overall server performance to determine how much load to admit.[3] SEDA controls server load by rate-limiting clients, with each client adjusting its sending rate based on the 99th-percentile end-to-end latency it observes. Protego optimizes overall server throughput while performing per-lock AQM. As such, its objective and design are the closest to Svalinn. Protego regulates load at the server through a credit-based mechanism, dynamically adjusting the credit-pool size based on the observed efficiency of request processing. We do not compare against Atropos [23] or pBox [22], as neither provides an admission controller or a technique for handling memory bandwidth contention, making Protego the closest comparison point.

[2]Processing time refers to the request completion time observed by the client at low to medium load.

[3]We observe that other systems that react to overall server performance like Breakwater [9] and Dagor [59] behave similar to SEDA. We omit the result to maintain the clarity of the presentation.

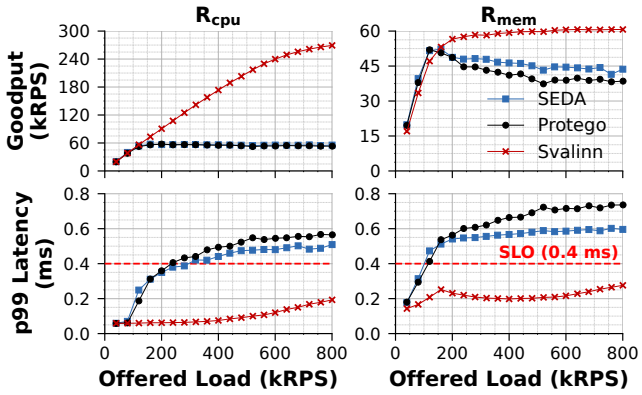


Figure 4: Performance of SEDA, Protego, and Svalinn for synthetic application on SETUPA.

Parameter configuration. We tune each system to operate at its best configuration. Most parameters are a function of the application SLO. We set the SLO to $5 \cdot (\textit{mean service time} + \textit{mean RTT})$. This is consistent with recent work [9, 10, 12, 46]. Since our applications have multiple request types, we use the service time of the longest request type when computing the SLO. In almost all applications, this corresponds to the R_{mem} request.

For SEDA, we set the target delay and timeout according to the application SLO. We further tune adj_i (rate increase factor) and adj_d (rate decrease factor) for each application to achieve high throughput while keeping latency under control. We use the default configuration from [55] for all other parameters.

For Protego, we tune the control-loop period according to the application's service time and configure the per-request queueing-delay budget for each workload. We compute this budget as $\textit{SLO} - \textit{p99 service time} - \textit{p99 RTT}$, using the service time of the longer request type. We use the default configuration from [10] for all other parameters.

For Svalinn, the throughput controller uses a utility function based on overall server throughput. We set δ to 1 for all applications, since this worked well across all workloads. We set $\Delta\, t_{warmup}$ and $\Delta\, t_{monitor}$ based on the application's service time and the RTT between clients and servers. Svalinn uses the same per-request queueing-delay budget as Protego. For `m_semaphore`, we set $\Delta\, t_{msem}$, α, ω, and ε to 500μs, 0.7, 0.8, and 0.3, respectively. These settings are shared across all workloads and setups. As a result, the only Svalinn parameters whose configuration varies across applications are the throughput controller's $\Delta\, t_{warmup}$ and $\Delta\, t_{monitor}$.

Evaluation metrics. For each application, we report goodput and 99th-percentile tail latency. Goodput is the throughput of requests that complete within the configured SLO. We report these metrics separately for each request type. In select experiments, we also report the drop rate, defined as the fraction of

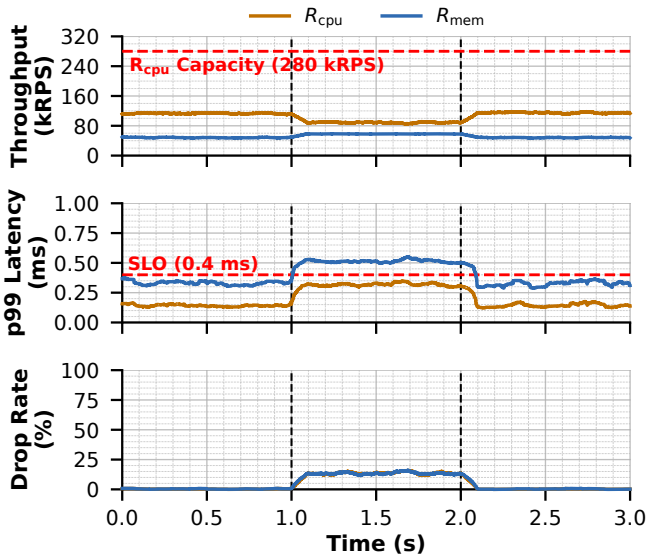


Figure 5: Performance of an overprovisioned server under an R_{mem} burst when using Protego on SETUPA

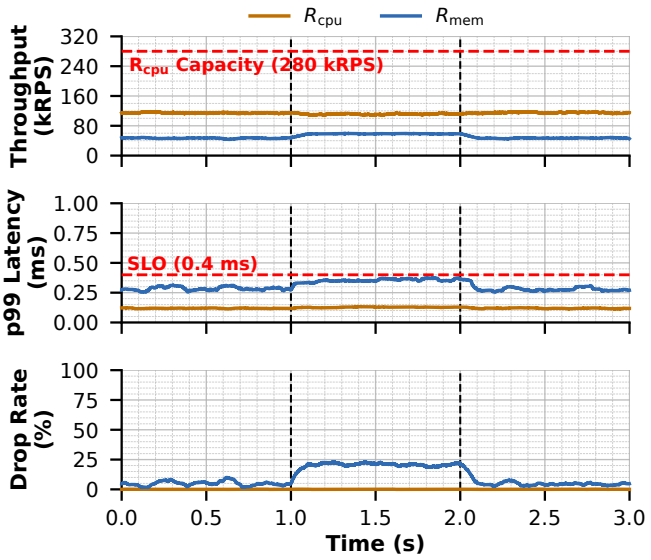


Figure 6: Performance of an overprovisioned server under an R_{mem} burst when using Svalinn on SETUPA

received requests that are dropped.

5.2 End-to-End Performance

We start by studying the performance of Svalinn in a single setup (SETUPA) across multiple applications. We study the robustness of Svalinn across different setups and configurations later.

Synthetic workload. Since the service times of R_{cpu} and R_{mem} are comparable, clients generate a 50/50 mix of R_{cpu}/R_{mem} requests. On SETUPA, six concurrently running R_{mem} requests saturate the available memory bandwidth (Figure 4). SEDA reacts to end-to-end client delay, unaware of the distinct CPU and memory bandwidth bottlenecks at the server. SEDA applies its rate limit based on the state of the most congested resource (memory bandwidth in this scenario), wasting CPU cycles that could have been used to serve R_{cpu} requests (i.e., leads to low throughput). SEDA also suffers from high tail latency that exceeds the SLO due to incast scenarios.

Protego aims to avoid the pitfalls of SEDA by admitting load until all request-execution paths appear fully utilized. In principle, Protego should achieve high throughput for both R_{cpu} and R_{mem} requests. In practice, we observe that R_{cpu} throughput collapses once memory bandwidth saturates, similar to SEDA. This behavior arises because R_{mem} requests stall when memory bandwidth saturates, wasting CPU cycles and eliminating any capacity that could have been used by R_{cpu} requests. Moreover, Protego does not provide any mechanisms to limit the latency of requests bottlenecked by memory bandwidth, leading to tail latency that exceeds the SLO.

Svalinn avoids the pitfalls of both systems. Its `m_semaphore` limits the number of concurrently running R_{mem} requests to the smallest number of cores that saturate memory bandwidth (six on SETUPA). Thus, R_{mem} requests can achieve their maximum throughput while leaving plenty of CPU capacity for R_{cpu} requests. Svalinn's throughput-oriented admission controller then admits additional load until this CPU headroom is fully utilized. As a result, Svalinn improves the goodput of R_{cpu} and R_{mem} requests by up to 5.06× and 1.60× without compromising latency. Svalinn achieves up to 5.48× and 5.95× lower p99 latency for R_{cpu} requests compared to SEDA and Protego, and 2.87× and 3.49× lower latency for R_{mem} requests.

Performance when CPU is overprovisioned. We evaluate how memory bandwidth congestion can undermine an operator's provisioning assumptions, and demonstrate how Svalinn enables the server to fully realize its provisioned CPU capacity. We run the synthetic workload on a SETUPA server with a steady load of 115 kRPS R_{cpu} requests and 50 kRPS R_{mem} requests for 3 seconds. The server can process up to 280 kRPS R_{cpu} and 60 kRPS R_{mem} requests (Figure 4), so the steady demand does not overload either bottleneck. At the 1 second mark, we add a 25 kRPS R_{mem} burst for 1 second, increasing total R_{mem} demand to 75 kRPS and overloading memory bandwidth.

As shown in Figure 5, Protego sustains the steady-state demand prior to the burst. However, during the burst, excess R_{mem} requests saturate the memory bandwidth and stall the CPU. This interference severely diminishes the CPU capacity available for useful R_{cpu} execution, despite the server being nominally overprovisioned for R_{cpu} demand. Consequently, R_{cpu} throughput plummets from 115 kRPS to approximately 90 kRPS (a 21.7% reduction), while tail latencies for both request types spike. Driven by memory bandwidth contention, R_{mem} latency violates its SLO, causing both workloads to

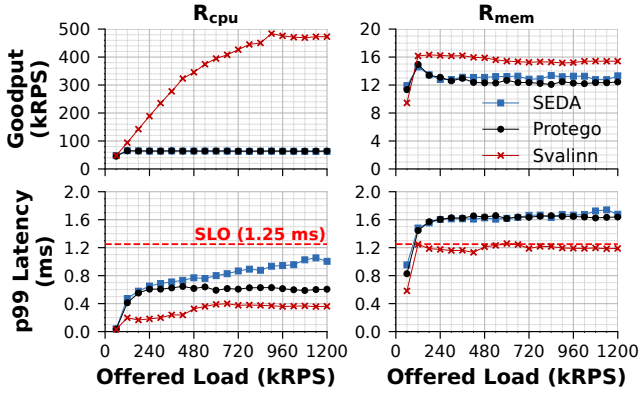


Figure 7: Performance of SEDA, Protego, and Svalinn for RocksDB on SETUPA.

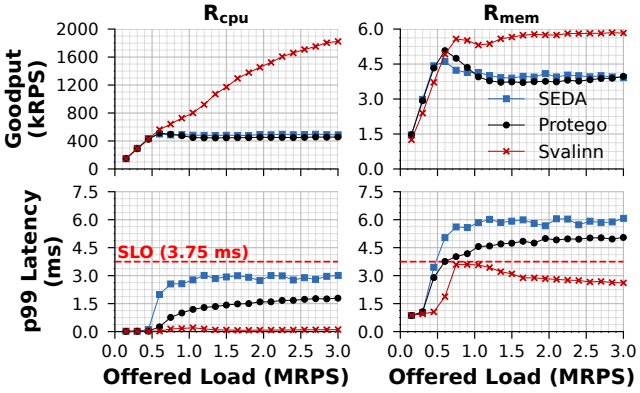


Figure 8: Performance of SEDA, Protego, and Svalinn for Dataframe on SETUPA.

experience a roughly 20% drop rate during the burst. These results demonstrate that a server provisioned adequately for steady-state workloads can still fail under resource contention.

In contrast, Svalinn prevents this hidden capacity degradation by explicitly managing memory bandwidth (Figure 6). Rather than allowing the burst to saturate the CPU, the `m_semaphore` restricts R_{mem} concurrency and drops excess R_{mem} requests before they can occupy CPU cores and waste CPU cycles by stalling. This proactive isolation enables Svalinn to fully sustain the 115 kRPS R_{cpu} demand, maintaining stable R_{cpu} tail latencies and achieving a 0% drop rate throughout the burst. Concurrently, R_{mem} tail latency increases only moderately, remaining well within its SLO bounds. To maintain this uncongested state, Svalinn sheds approximately 25% of R_{mem} requests during the peak of the burst. One note about Svalinn's behavior is that outside of the burst window, while nominal R_{mem} demand remains safely below the server's total memory bandwidth, transient arrival spikes can still briefly saturate the `m_semaphore` capacity, resulting in a baseline R_{mem} drop rate of roughly 5%.

RocksDB. Due to the difference in the service time between the different request types, clients generate an 80/20 mix of R_{cpu} and R_{mem} requests. The results are shown in Figure 7. On SETUPA, 13–14 concurrent R_{mem} requests saturate memory bandwidth. Once this happens, SEDA stops admitting additional load, causing R_{cpu} goodput to plateau together with R_{mem} goodput. SEDA also exceeds the configured SLO because of request incast. Protego admits load until all request paths are saturated but performs no AQM for memory bandwidth, resulting in high tail latencies and no throughput gain for R_{cpu} requests, similar to the synthetic workload.

Svalinn constrains R_{mem} requests to the minimum number of cores needed to saturate memory bandwidth, avoiding memory bandwidth congestion. Moreover, it allows R_{cpu} requests to utilize the remaining CPU. This yields up to 1.26× higher R_{mem} goodput and up to 7.62× higher R_{cpu} goodput than the baselines. Svalinn also reduces R_{cpu} and R_{mem} p99 latency by up to 2.95× and 1.47× relative to the baselines.

DataFrame. Due to the very high variation in request service time, clients issue a 99/1 mix of R_{cpu} and R_{mem} requests. On SETUPA, 7–8 concurrent R_{mem} requests saturate memory bandwidth. Figure 8 shows that SEDA stops admitting additional load as soon as memory bandwidth becomes the bottleneck, while Protego continues admitting load but cannot prevent R_{mem} requests from stalling once memory bandwidth is congested. As a result, both systems limit R_{cpu} goodput and suffer high R_{mem} tail latency, due to request incasts in SEDA and the lack of AQM for R_{mem} requests in Protego.

Svalinn restricts the number of concurrent R_{mem} operations to avoid memory bandwidth congestion and continues admitting additional R_{cpu} operations until CPU capacity is fully utilized. As a result, Svalinn reduces R_{cpu} p99 latency by up to 42.77× relative to SEDA and 21.17× relative to Protego, and reduces R_{mem} p99 latency by 2.33× and 1.93×, respectively. At high load, uncontrolled bandwidth contention causes SEDA and Protego to deliver only 0.67× the R_{mem} goodput of Svalinn. Overall, Svalinn achieves up to 3.99× higher R_{cpu} goodput than both baselines.

We observe similar behavior in Memcached, where clients generate R_{cpu}, R_{mem}, and R_{lock} requests. There, Svalinn improves overall goodput by up to 2.47× relative to the baselines without compromising latency; detailed results are provided in Appendix B.2.

5.3 Microbenchmark

Versatility of the throughput controller. The default utility function used by Svalinn's throughput controller is purely throughput oriented: $Utility_{tput} = \frac{stats.resps_{out}}{stats.duration}$. We compare it against $Utility_{drop}$, which balances throughput and drop rate by enforcing a maximum drop-rate of 10%. Specifically, $Utility_{drop}$ is computed as follows:

$$Utility_{drop} = \begin{cases} \frac{stats.resps_{out} - stats.reqs_{drop}}{stats.duration} & \text{if } \frac{stats.reqs_{drop}}{stats.reqs_{in}} < 0.1 \\ -\frac{stats.reqs_{drop}}{stats.duration} & \text{otherwise} \end{cases}$$

Figure 9 compares the performance of the two utility functions under a synthetic workload consisting of an 80/20 mix of

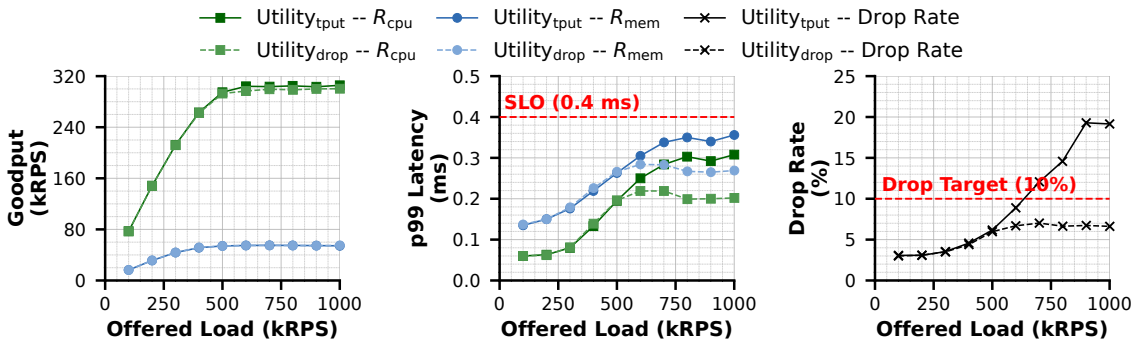


Figure 9: Comparison of throughput-oriented and drop-oriented admission utility functions on SETUPA.

R_{cpu} and R_{mem} requests. Compared to $Utility_{drop}$, $Utility_{tput}$ yields up to 1.03× and 1.01× higher R_{cpu} and R_{mem} goodput, respectively. However, this gain comes at the expense of elevated tail latencies and drop rates reaching 19%. Conversely, $Utility_{drop}$ strictly enforces the 10% drop-rate target, capping actual drops at 7%, while simultaneously reducing R_{cpu} and R_{mem} p99 latencies by up to 1.52× and 1.32×. These results demonstrate that Svalinn 's configurable utility function provides operators with flexible control to effectively balance trade-offs among throughput, latency, and drop rates. The tradeoff becomes sharper at more memory-intensive mixes. For example, with a 50/50 R_{cpu}/R_{mem} mix, memory bandwidth saturates earlier than in the 80/20 case, causing `m_semaphore` to begin dropping requests sooner. As a result, $Utility_{drop}$ favors a lower drop rate over higher throughput, leading to significant throughput degradation while keeping drop rate and latency low. More generally, Svalinn's performance gains vary with the request mix, and we provide a detailed study of this effect in Appendix B.5.

Latency of retransmitted requests. When overload controllers drop requests, clients often retransmit them to an alternate replica. To evaluate whether Svalinn rejects dropped requests quickly enough to prevent inflating end-to-end latency, we execute a synthetic workload (70/30 R_{cpu}/R_{mem} mix) on SETUPA using a two-server configuration. Clients initially direct all traffic to Server 1; if Server 1 drops a request due to overload, the client retransmits it to Server 2 [10, 40]. We drive Server 1 beyond its processing capacity to deliberately induce drops and retransmissions. To accommodate the additional network hop, we scale the workload SLO to twice the single-server baseline defined in Section 5.1.

Figure 10 illustrates the resulting performance. Across all offered loads, the tail latency for both R_{cpu} and R_{mem} requests remains safely within the adjusted SLO, and even tracks below the single-server latency target. This efficiency stems from `m_semaphore` maintaining a zero-length queue for threads attempting to enter the memory-intensive section. Consequently, excess requests are rejected near-instantaneously rather than enduring queuing delays at the congested Server 1, a behavior confirmed by the rapid failure-delivery times highlighted in red in Figure 10. Server 2 then seamlessly absorbs this shed load, as evidenced by the sustained goodput plots.

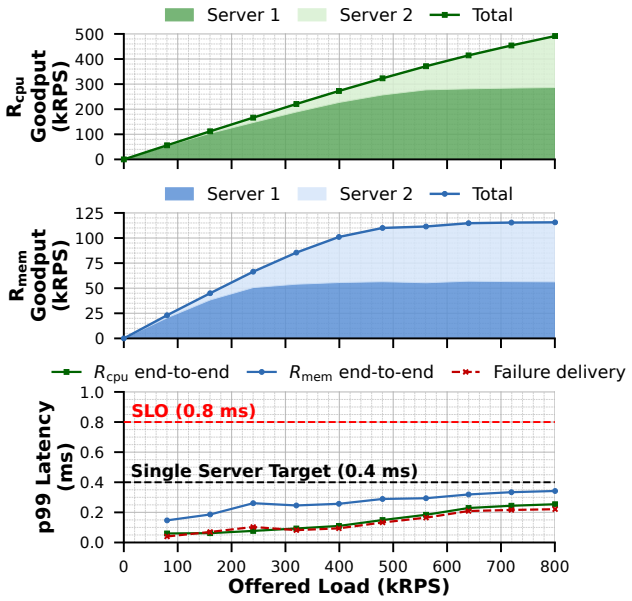


Figure 10: Multi-server performance of Svalinn for synthetic workload with retransmission on SETUPA.

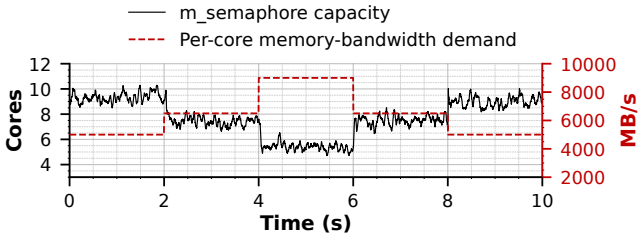


Figure 11: Memory semaphore's ability to adapt to changing per-request memory-bandwidth demand on SETUPA.

Adaptability of `m_semaphore` to workload dynamics. The per-request demand for memory bandwidth can vary, requiring the `m_semaphore` to be able to adapt its capacity quickly. We evaluate this using a synthetic workload with

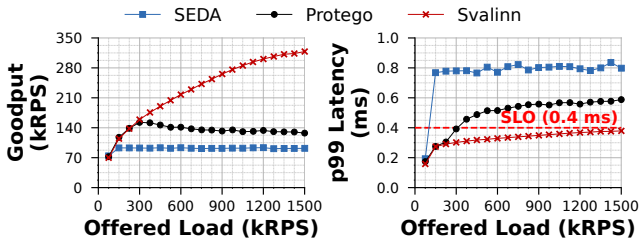


Figure 12: Performance of SEDA, Protego, and Svalinn for synthetic application with multiple bottlenecks on SETUPA.

three R_{mem} request types: a low-demand request performing 25 non-temporal stores on a 3 KB buffer, a medium-demand request performing 500 such stores on a 3 KB buffer, and a high-demand request performing 26 stores on a 128 KB buffer. Their approximate per-core bandwidth demands are 5000, 6500, and 9000 MB/s, respectively. We fix the offered load and switch the request type every two seconds (low to medium to high to medium to low). As shown in Figure 11, `m_semaphore` rapidly converges to the correct capacity after each shift. On average it takes 50 ms for the controller to converge to the new optimal capacity. This adaptability stems from the periodic exploration performed by our ε-greedy MAB algorithm.

Performance under multiple bottlenecks. We evaluate Svalinn's ability to manage multiple concurrent bottlenecks using a synthetic workload that generates a 25/25/50 mix of R_{cpu}/R_{mem}/R_{lock} requests on SETUPA. In this scenario, the global mutex acts as the primary point of contention, causing R_{lock} requests to saturate first at approximately 180 kRPS of offered load (Figure 12). This behavior clearly exposes the single-queue fallacy: SEDA reacts exclusively to this initial bottleneck and halts further admission, leaving the remaining capacity of the R_{cpu} and R_{mem} execution paths entirely unexploited.

Protego mitigates this specific failure mode by applying AQM directly at the global mutex, which enables it to continue admitting traffic beyond the R_{lock} saturation threshold. However, as observed in previous experiments, Protego lacks a mechanism for memory bandwidth latency control. Consequently, once memory bandwidth saturates at roughly 300 kRPS, stalled R_{mem} requests monopolize CPU cores and degrade the capacity available for the R_{cpu} workload.

Svalinn successfully navigates both bottlenecks simultaneously: its mutex AQM isolates R_{lock} latency, while `m_semaphore` bounds R_{mem} concurrency to prevent memory bandwidth congestion from stalling the CPU cores. This joint enforcement allows Svalinn to continuously harvest the remaining CPU capacity for R_{cpu} execution. As a result, Svalinn improves overall application goodput by up to 3.46× and 2.50× compared to SEDA and Protego, respectively, while simultaneously reducing p99 tail latency by up to 2.80× and 1.61×. A detailed per-request breakdown of these metrics is provided in Appendix B.1.

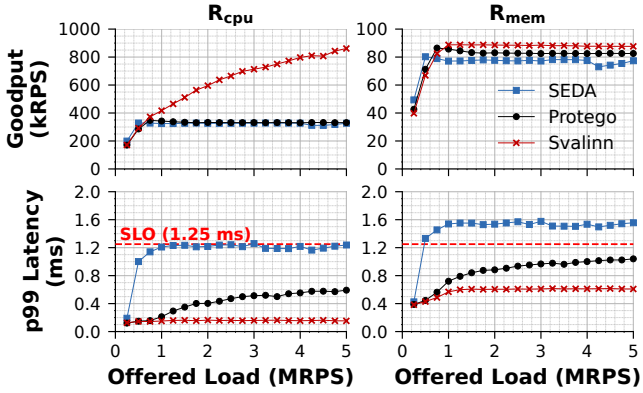


Figure 13: Performance of SEDA, Protego, and Svalinn for RocksDB on SETUPB.

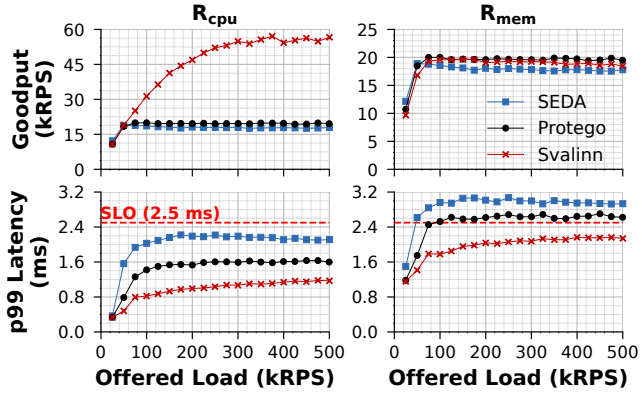


Figure 14: Performance of SEDA, Protego, and Svalinn for synthetic application in Go on SETUPA.

Performance with different machines. We run RocksDB on SETUPB (Figure 13). Svalinn's relative performance persists in this setup. Compared to SEDA, Svalinn reduces tail latency by up to 8.09× for R_{cpu} requests and 2.58× for R_{mem} requests. Compared to Protego, Svalinn reduces tail latency by up to 3.87× for R_{cpu} requests and 1.71× for R_{mem} requests. Svalinn improves the goodput of R_{cpu} requests by 2.65× over both SEDA and Protego without harming R_{mem} requests.

Svalinn in Go. We test a synthetic workload in Go with an 80/20 R_{cpu}/R_{mem} request mix, and observe that the benefits of Svalinn continue to hold across runtimes. As shown in Figure 14, Svalinn achieves up to 3.21× higher R_{cpu} request goodput. Moreover, Svalinn delivers up to 3.25× and 1.85× lower tail latency for R_{cpu} and R_{mem} requests, respectively, than the baselines.

6 Related Work

Receiver-driven admission control. Receiver-driven admission control regulates load at the server using the most accurate view of local capacity. Transport-level schemes such as Homa [38] and SIRD [45] operate on packets to reduce switch congestion. RPC-level systems such as Breakwater [9], CoreSync [42], and Protego [10] control request admission

at the host. These approaches are particularly effective for handling incast. Svalinn follows this receiver-driven model while overcoming the single-queue fallacy in prior work.

Overload control. The importance of the overload problem has led to a wide range of solutions. Some solutions attempt to optimize performance across microservice chains [43, 49, 57, 59]. Other solutions consider the performance of a single host, focusing on a single resource (e.g., CPU [9, 55], locks [10, 23], and storage [21, 35]). Most existing controllers are designed primarily for their narrow set of bottlenecks. Atropos [23] particularly is the most general purpose technique. However, its focus is still on resources that applications can explicitly track (i.e., memory buffers, locks, and queues). Moreover, it does not offer an admission controller, offloading the responsibility of deciding the precise load per replica to a separate system. Svalinn handles implicitly accessed memory bandwidth bottleneck and introduces an admission control scheme that integrates with its per-resource latency controller.

Resource isolation. Another approach to improve resource utilization is to perform isolation between different requests. Typical approaches perform coarse-grained isolation at the binary level [7,19,47,58]. Recently, pBox [22] proposed intra-application isolation, focusing on explicitly visible resources (i.e., it is nontrivial to deal with memory bandwidth contention in pBox). However, pBox does not drop any of the competing requests, leading to potentially high latencies for executing requests. Moreover, it does not provide an admission control technique. Another common technique for dealing with potential intra-application interference or head-of-line blocking is preemption [20, 29, 30]. We demonstrate that such techniques do not address problems solved by Svalinn. Among existing systems, Persephone [13] is closest to Svalinn as it explicitly limits the number of cores allocated to long-running requests. However, Persephone bases this allocation strictly on execution service times, a strategy that can inadvertently assign more cores to memory-intensive requests and thereby exacerbate memory-bandwidth contention. Furthermore, while Persephone relies on the assumption that request types are classified and known a priori, Svalinn requires no such prior knowledge, dynamically adapting to heterogeneous workloads on the fly.

Memory and interconnect congestion. Recently, there has been growing interest in congestion within the host interconnect [2, 3, 39, 53]. Solutions typically operate at the packet level, tackling the host interconnect as an extension of the network path between a client and a server. In particular, HostCC [3] assumes interference between a low-priority memory-intensive application and a high-priority latency-sensitive application, and mitigates it by throttling the memory-intensive application's cores using Intel MBA. Svalinn addresses a different setting: heterogeneous requests within the same application. In our setting, throttling memory-intensive requests with MBA would only cause them to stall longer on the CPU, exacerbating the very inefficiency that Svalinn seeks to eliminate.

Learning-based control. Learning-based controllers are widely used for dynamic optimization, including network congestion control [4, 14, 15], overload control [43, 54], and isolation [47, 58]. Svalinn follows this direction, relying on interpretable algorithms that can accommodate a wide range of user-defined utility functions.

7 Conclusion

This paper demonstrates that traditional, monolithic overload control systems are fundamentally limited by the *single-queue fallacy*. By treating modern server binaries as homogeneous black boxes and reacting solely to aggregate metrics like end-to-end latency, state-of-the-art controllers prematurely throttle inbound load and leave vast swaths of machine capacity stranded. We present Svalinn, a modular overload control architecture that structurally decouples throughput optimization from distributed, per-resource latency management. Through the introduction of the `m_semaphore` abstraction, Svalinn successfully converts the implicit memory bandwidth bottleneck into explicitly managed, observable software entities, optimizing resource execution paths without the need for intrusive application re-architecting. Our evaluation across diverse runtimes and applications proves that Svalinn achieves substantial goodput improvements (up to 6.51×) while strictly preserving latency guarantees.

Beyond its immediate performance gains, Svalinn provides a fundamental architectural blueprint for the future of microsecond-scale infrastructure. As modern hardware trends continue toward massive core parallelism coupled with complex, non-uniform memory and interconnect topologies, the intersection of data-dependent application paths and shared hardware constraints will only intensify. By rejecting the single-queue assumption, Svalinn demonstrates how a resource-agnostic admission controller, guided by a multi-dimensional utility function, can be composed with resource-aware, per-bottleneck latency control. Svalinn establishes an extensible foundation for a new generation of overload control systems capable of harvesting the full potential of next-generation cloud architectures.

Acknowledgments

We thank our shepherd and the anonymous reviewers. We also thank CloudLab [17] for providing us with the infrastructure to run our experiments. This work was funded in part by NSF grants CNS-2212098 and IIS-2542973 and a Google Research Award.

References

[1] Memcached: High-performance, distributed memory object caching system. https://memcached.org/. Accessed: 2025-02-01.

[2] Saksham Agarwal, Rachit Agarwal, Behnam Montazeri, Masoud Moshref, Khaled Elmeleegy, Luigi Rizzo, Marc Asher de Kruijf, Gautam Kumar, Sylvia Ratnasamy, David Culler, and Amin Vahdat. Understanding host interconnect congestion. In *Proceedings of the 21st ACM Workshop on Hot Topics in Networks*, HotNets '22, page 198–204, New York, NY, USA, 2022. Association for Computing Machinery.

[3] Saksham Agarwal, Arvind Krishnamurthy, and Rachit Agarwal. Host congestion control. In *Proceedings of the ACM SIGCOMM 2023 Conference*, ACM SIGCOMM '23, page 275–287, New York, NY, USA, 2023. Association for Computing Machinery.

[4] Venkat Arun and Hari Balakrishnan. Copa: Practical Delay-Based congestion control for the internet. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, pages 329–342, Renton, WA, April 2018. USENIX Association.

[5] Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. Workload analysis of a large-scale key-value store. In *Proceedings of the 12th ACM SIGMETRICS/PERFORMANCE Joint International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '12, page 53–64, New York, NY, USA, 2012. Association for Computing Machinery.

[6] Peter Auer, Nicolò Cesa-Bianchi, and Paul Fischer. Finite-time analysis of the multiarmed bandit problem. *Machine Learning*, 47(2–3):235–256, 2002.

[7] Shuang Chen, Christina Delimitrou, and José F Martínez. Parties: Qos-aware resource partitioning for multiple interactive services. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '19)*, pages 107–120, 2019.

[8] Inho Cho, Seo Jin Park, Ahmed Saeed, Mohammad Alizadeh, and Adam Belay. LDB: An efficient latency profiling tool for multithreaded applications. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*, pages 1497–1510, Santa Clara, CA, April 2024. USENIX Association.

[9] Inho Cho, Ahmed Saeed, Joshua Fried, Seo Jin Park, Mohammad Alizadeh, and Adam Belay. Overload control for µs-scale RPCs with breakwater. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 299–314. USENIX Association, November 2020.

[10] Inho Cho, Ahmed Saeed, Seo Jin Park, Mohammad Alizadeh, and Adam Belay. Protego: Overload control for applications with unpredictable lock contention. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 725–738, Boston, MA, April 2023. USENIX Association.

[11] Siddharth D. Eur/usd 1min dataset, 2023.

[12] Alexandros Daglis, Mark Sutherland, and Babak Falsafi. Rpcvalet: Ni-driven tail-aware balancing of µs-scale rpcs. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '19, page 35–48, New York, NY, USA, 2019. Association for Computing Machinery.

[13] Henri Maxime Demoulin, Joshua Fried, Isaac Pedisich, Marios Kogias, Boon Thau Loo, Linh Thi Xuan Phan, and Irene Zhang. When idling is ideal: Optimizing tail-latency for heavy-tailed datacenter workloads with perséphone. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, SOSP '21, page 621–637, New York, NY, USA, 2021. Association for Computing Machinery.

[14] Mo Dong, Qingxi Li, Doron Zarchy, P. Brighten Godfrey, and Michael Schapira. PCC: Re-architecting congestion control for consistent high performance. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*, pages 395–408, Oakland, CA, May 2015. USENIX Association.

[15] Mo Dong, Tong Meng, Doron Zarchy, Engin Arslan, Yossi Gilad, Brighten Godfrey, and Michael Schapira. PCC vivace: Online-Learning congestion control. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, pages 343–356, Renton, WA, April 2018. USENIX Association.

[16] Siying Dong, Andrew Kryczka, Yanqin Jin, and Michael Stumm. Rocksdb: Evolution of development priorities in a key-value store serving large-scale applications. *ACM Trans. Storage*, 17(4), October 2021.

[17] Dmitry Duplyakin, Robert Ricci, Aleksander Maricq, Gary Wong, Jonathon Duerig, Eric Eide, Leigh Stoller, Mike Hibler, David Johnson, Kirk Webb, Aditya Akella, Kuangching Wang, Glenn Ricart, Larry Landweber, Chip Elliott, Michael Zink, Emmanuel Cecchet, Snigdhaswin Kar, and Prabodh Mishra. The design and operation of CloudLab. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 1–14, Renton, WA, July 2019. USENIX Association.

[18] Envoy Project Authors. Envoy proxy. https://www.envoyproxy.io/, 2025.

[19] Joshua Fried, Zhenyuan Ruan, Amy Ousterhout, and Adam Belay. Caladan: Mitigating interference at microsecond timescales. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 281–297. USENIX Association, November 2020.

[20] Linsong Guo, Danial Zuberi, Tal Garfinkel, and Amy Ousterhout. The benefits and limitations of user interrupts for preemptive userspace scheduling. In *22nd USENIX Symposium on Networked Systems Design and Implementation (NSDI 25)*, pages 1015–1032, 2025.

[21] Mingzhe Hao, Levent Toksoz, Nanqinqin Li, Edward Edberg Halim, Henry Hoffmann, and Haryadi S. Gunawi. LinnOS: Predictability on unpredictable flash storage with a light neural network. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 173–190. USENIX Association, November 2020.

[22] Yigong Hu, Gongqi Huang, and Peng Huang. Pushing performance isolation boundaries into application with pbox. In *Proceedings of the 29th Symposium on Operating Systems Principles (SOSP '23)*, pages 247–263, 2023.

[23] Yigong Hu, Zeyin Zhang, Yicheng Liu, Yile Gu, Shuangyu Lei, Baris Kasikci, and Peng Huang. Mitigating application resource overload with targeted task cancellation. In *Proceedings of the ACM SIGOPS 31st Symposium on Operating Systems Principles (SOSP '25)*, pages 270–285, 2025.

[24] Yigong Hu, Haodong Zheng, Yicheng Liu, Dedong Xie, Youliang Huang, and Baris Kasikci. gigiprofiler: Diagnosing performance issues by uncovering application resource bottlenecks, 2025.

[25] Jack Tigar Humphries, Neel Natu, Ashwin Chaugule, Ofir Weisse, Barret Rhoden, Josh Don, Luigi Rizzo, Oleg Rombakh, Paul Turner, and Christos Kozyrakis. ghost: Fast & flexible user-space delegation of linux scheduling. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, pages 588–604, 2021.

[26] Darby Huye, Yuri Shkuro, and Raja R Sambasivan. Lifting the veil on {Meta's} microservice architecture: Analyses of topology and request workflows. In *2023 USENIX Annual Technical Conference (USENIX ATC 23)*, pages 419–432, 2023.

[27] Intel Corporation. Intel vtune profiler, 2025. Accessed: 2026-05-25.

[28] Intel PCM Project Contributors. Intel ® performance counter monitor (pcm). https://github.com/intel/pcm, 2025.

[29] Rishabh Iyer, Musa Unal, Marios Kogias, and George Candea. Achieving microsecond-scale tail latency efficiently with approximate optimal scheduling. In *Proceedings of the 29th Symposium on Operating Systems Principles*, SOSP '23, page 466–481, New York, NY, USA, 2023. Association for Computing Machinery.

[30] Kostis Kaffes, Timothy Chong, Jack Tigar Humphries, Adam Belay, David Mazières, and Christos Kozyrakis. Shinjuku: Preemptive scheduling for usecond-scale tail latency. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 345–360, Boston, MA, February 2019. USENIX Association.

[31] Gautam Kumar, Nandita Dukkipati, Keon Jang, Hassan M. G. Wassel, Xian Wu, Behnam Montazeri, Yaogong Wang, Kevin Springborn, Christopher Alfeld, Michael Ryan, David Wetherall, and Amin Vahdat. Swift: Delay is simple and effective for congestion control in the datacenter. In *Proceedings of the Annual Conference of the ACM Special Interest Group on Data Communication on the Applications, Technologies, Architectures, and Protocols for Computer Communication*, SIGCOMM '20, page 514–528, New York, NY, USA, 2020. Association for Computing Machinery.

[32] Bradley C. Kuszmaul, Matteo Frigo, Justin Mazzola Paluska, and Alexander (Sasha) Sandler. Everyone loves file: File storage service (FSS) in oracle cloud infrastructure. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 15–32, Renton, WA, July 2019. USENIX Association.

[33] Yizhuo Liang, Ramesh Govindan, and Seo Jin Park. Granular resource demand heterogeneity. In *Proceedings of the 2025 Workshop on Hot Topics in Operating Systems*, HotOS '25, page 187–194, New York, NY, USA, 2025. Association for Computing Machinery.

[34] Linux perf Wiki Contributors. perf: Linux profiling with performance counters, 2025. Accessed: 2026-05-25.

[35] Jaehong Min, Ming Liu, Tapan Chugh, Chenxingyu Zhao, Andrew Wei, In Hwan Doh, and Arvind Krishnamurthy. Gimbal: enabling multi-tenant storage disaggregation on smartnic jbofs. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*, pages 106–122, 2021.

[36] Hossein Moein. Dataframe: A high-performance, modern c++ data analysis library. https://github.com/hosseinmoein/DataFrame. Accessed: 2025-02-01.

[37] Jeffrey C Mogul and KK Ramakrishnan. Eliminating receive livelock in an interrupt-driven kernel. *ACM Transactions on Computer Systems*, 1997.

[38] Behnam Montazeri, Yilong Li, Mohammad Alizadeh, and John Ousterhout. Homa: a receiver-driven low-latency transport protocol using network priorities. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '18, page 221–235, New York, NY, USA, 2018. Association for Computing Machinery.

[39] Rolf Neugebauer, Gianni Antichi, José Fernando Zazo, Yury Audzevich, Sergio López-Buedo, and Andrew W. Moore. Understanding pcie performance for end host networking. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '18, page 327–341, New York, NY, USA, 2018. Association for Computing Machinery.

[40] Rajesh Nishtala, Hans Fugal, Steven Grimm, Marc Kwiatkowski, Herman Lee, Harry C. Li, Ryan McElroy, Mike Paleczny, Daniel Peek, Paul Saab, David Stafford, Tony Tung, and Venkateshwaran Venkataramani. Scaling memcache at facebook. In *10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*, pages 385–398, Lombard, IL, April 2013. USENIX Association.

[41] Amy Ousterhout, Joshua Fried, Jonathan Behrens, Adam Belay, and Hari Balakrishnan. Shenango: Achieving high CPU efficiency for latency-sensitive datacenter workloads. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 361–378, Boston, MA, February 2019. USENIX Association.

[42] Bhaskar Pardeshi, Eric Stuhr, and Ahmed Saeed. Coresync: A protocol for joint core scheduling and overload control of μs-scale tasks. In *2025 IEEE 33rd International Conference on Network Protocols (ICNP)*, pages 1–11. IEEE, 2025.

[43] Jinwoo Park, Jaehyeong Park, Youngmok Jung, Hwijoon Lim, Hyunho Yeo, and Dongsu Han. Topfull: An adaptive top-down overload control for slo-oriented microservices. In *Proceedings of the ACM SIGCOMM 2024 Conference*, ACM SIGCOMM '24, page 876–890, New York, NY, USA, 2024. Association for Computing Machinery.

[44] Yajuan Peng, Shuang Chen, Yi Zhao, and Zhibin Yu. {UFO}: The ultimate {QoS-Aware} core management for virtualized and oversubscribed public clouds. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*, pages 1511–1530, 2024.

[45] Konstantinos Prasopoulos, Ryan Kosta, Edouard Bugnion, and Marios Kogias. SIRD: A Sender-Informed, Receiver-Driven datacenter transport protocol. In *22nd USENIX Symposium on Networked Systems Design and Implementation (NSDI 25)*, pages 451–471, Philadelphia, PA, April 2025. USENIX Association.

[46] George Prekas, Marios Kogias, and Edouard Bugnion. Zygos: Achieving low tail latency for microsecond-scale networked tasks. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, page 325–341, New York, NY, USA, 2017. Association for Computing Machinery.

[47] Haoran Qiu, Subho S. Banerjee, Saurabh Jha, Zbigniew T. Kalbarczyk, and Ravishankar K. Iyer. FIRM: An intelligent fine-grained resource management framework for SLO-Oriented microservices. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 805–825. USENIX Association, November 2020.

[48] Herbert Robbins. Some aspects of the sequential design of experiments. *Bulletin of the American Mathematical Society*, 58(5):527–535, 1952.

[49] Lalith Suresh, Peter Bodik, Ishai Menache, Marco Canini, and Florin Ciucu. Distributed resource management across process boundaries. In *Proceedings of the 2017 Symposium on Cloud Computing (SoCC '17)*, pages 611–623, 2017.

[50] Lalith Suresh, Marco Canini, Stefan Schmid, and Anja Feldmann. C3: Cutting tail latency in cloud data stores via adaptive replica selection. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*, pages 513–527, 2015.

[51] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*. The MIT Press, 2 edition, 2018.

[52] Twitter Engineering. Deterministic aperture: A distributed, load balancing algorithm. https://blog.twitter.com/engineering/en_us/topics/infrastructure/2019/daperture-load-balancer, 2019. Accessed: 2025-12-07.

[53] Midhul Vuppalapati, Saksham Agarwal, Henry Schuh, Baris Kasikci, Arvind Krishnamurthy, and Rachit Agarwal. Understanding the host network. In *Proceedings of the ACM SIGCOMM 2024 Conference*, ACM SIGCOMM '24, page 581–594, New York, NY, USA, 2024. Association for Computing Machinery.

[54] Zibo Wang, Pinghe Li, Chieh-Jan Mike Liang, Feng Wu, and Francis Y. Yan. Autothrottle: A practical Bi-Level approach to resource management for SLO-Targeted microservices. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*, pages 149–165, Santa Clara, CA, April 2024. USENIX Association.

[55] Matt Welsh, David Culler, and Eric Brewer. Seda: an architecture for well-conditioned, scalable internet services. In *Proceedings of the Eighteenth ACM Symposium on Operating Systems Principles*, SOSP '01, page 230–243, New York, NY, USA, 2001. Association for Computing Machinery.

[56] Bartek Wydrowski, Robert Kleinberg, Stephen M Rumble, and Aaron Archer. Load is not what you should balance: Introducing prequal. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*, pages 1285–1299, 2024.

[57] Jiali Xing, Akis Giannoukos, Paul Loh, Shuyue Wang, Justin Qiu, Henri Maxime Demoulin, Konstantinos Kallas, and Benjamin C. Lee. Rajomon: Decentralized and coordinated overload control for Latency-Sensitive microservices. In *22nd USENIX Symposium on Networked Systems Design and Implementation (NSDI 25)*, pages 21–36, Philadelphia, PA, April 2025. USENIX Association.

[58] Yanqi Zhang, Weizhe Hua, Zhuangzhuang Zhou, G Edward Suh, and Christina Delimitrou. Sinan: Ml-based and qos-aware resource management for cloud microservices. In *Proceedings of the 26th ACM international conference on architectural support for programming languages and operating systems (ASPLOS '21)*, pages 167–181, 2021.

[59] Hao Zhou, Ming Chen, Qian Lin, Yong Wang, Xiaobin She, Sifan Liu, Rui Gu, Beng Chin Ooi, and Junfeng Yang. Overload control for scaling wechat microservices. In *Proceedings of the ACM Symposium on Cloud Computing*, SoCC '18, page 149–161, New York, NY, USA, 2018. Association for Computing Machinery.

Algorithm 2: Svalinn's Utility-Based Throughput Controller.

```
1  credits: Current credit pool size
2  Parameter δ: Credit pool change for every
     micro-experiment
3  Parameter Δ t_warmup: Time interval before monitoring the
     micro-experiment
4  Parameter Δ t_monitor: Time interval for which the
     micro-experiment is monitored

5  Function UpdateCredits():
       // Save original credit pool
6      credits_orig ← credits;
       // Admit more load
7      credits ← credits_orig + δ;
8      Wait(Δ t_warmup);
9      StartStatsCollection(stats_δ+);
10     Wait(Δ t_monitor);
11     EndStatsCollection(stats_δ+);
       // Admit less load
12     credits ← credits_orig − δ;
13     Wait(Δ t_warmup);
14     StartStatsCollection(stats_δ−);
15     Wait(Δ t_monitor);
16     EndStatsCollection(stats_δ−);
       // Calculate utilities
17     utility_δ+ ← CalculateUtility(stats_δ+);
18     utility_δ− ← CalculateUtility(stats_δ−);
       // Relabel utilities based on actual load
19     if stats_δ+.reqs_in ≤ stats_δ−.reqs_in then
20         Swap(utility_δ+, utility_δ−);
       // Update the credit pool
21     if utility_δ+ > utility_δ− then
22         credits ← credits_orig + δ;
23     else
24         credits ← credits_orig − δ;
```

A Additional Design Discussion

A.1 Utility-Based Throughput Controller Algorithm

The throughput controller continuously runs the `UpdateCredits()` routine, shown in Algorithm 2, while the application is running. In each iteration, the controller performs two micro-experiments (lines 7–11 and 12–16). During each micro-experiment, the controller either increases or decreases the credit pool by δ. To allow the newly set credit pool value to take effect, the controller waits for $\Delta\, t_{warmup}$ period before attributing observed performance to the new credit pool value. Finally, the controller monitors the application for $\Delta\, t_{monitor}$, and records relevant performance metrics. The controller computes a utility value from these metrics for each micro-experiment (lines 17–18). To handle cases in which perturbing the credit pool does not produce the expected change in incoming request rate, we relabel the micro-experiments using the actually observed incoming request rate (lines 19–20). After both micro-experiments complete, it updates the credit pool in the direction that yields higher utility (lines 21–24).

B Additional Evaluation Results

B.1 Request-level Performance with Multiple Bottlenecks

Figure 15 shows a detailed breakdown of the experiment in Figure 12 on SETUPA. Clients issue three request types: $\mathtt{R}_{\mathrm{cpu}}$, $\mathtt{R}_{\mathrm{mem}}$, and $\mathtt{R}_{\mathrm{lock}}$. As soon as the $\mathtt{R}_{\mathrm{lock}}$ path saturates at around 180 kRPS of offered load, SEDA stops admitting additional load. As a result, the goodput of both $\mathtt{R}_{\mathrm{cpu}}$ and $\mathtt{R}_{\mathrm{mem}}$ also plateaus at that point. SEDA also suffers from high tail latency due to request incasts at the most congested bottleneck, namely the global mutex. In contrast, the $\mathtt{R}_{\mathrm{cpu}}$ and $\mathtt{R}_{\mathrm{mem}}$ paths remain unsaturated and continue to exhibit low tail latency.

Protego continues admitting load past the global-mutex saturation point. It controls $\mathtt{R}_{\mathrm{lock}}$ tail latency by performing AQM at mutex access points and continues increasing load as long as throughput improves. Consequently, its goodput rises until memory bandwidth saturates at around 300 kRPS. Beyond this point, $\mathtt{R}_{\mathrm{mem}}$ requests stall on the CPU, causing CPU congestion and leaving no capacity for $\mathtt{R}_{\mathrm{cpu}}$ requests. As a result, $\mathtt{R}_{\mathrm{cpu}}$ goodput saturates together with $\mathtt{R}_{\mathrm{mem}}$ goodput. The $\mathtt{R}_{\mathrm{lock}}$ goodput declines at higher loads because client bursts can still enqueue many $\mathtt{R}_{\mathrm{lock}}$ requests at the global mutex. Since Protego's AQM uses an instantaneous queueing-delay estimate [10], requests arriving in the same short burst can observe a low estimate and be enqueued together, only to incur high delay later.

Svalinn behaves similarly to Protego until memory bandwidth becomes congested. At that point, `m_semaphore` limits the concurrency of $\mathtt{R}_{\mathrm{mem}}$ requests and prevents the resulting CPU congestion. This allows Svalinn's throughput-driven admission controller to admit more load and use the remaining CPU capacity. As a result, Svalinn achieves up to 8.71× and 4.18× higher $\mathtt{R}_{\mathrm{cpu}}$ goodput than SEDA and Protego, respectively. Throughout the experiment, Svalinn keeps tail latency within the configured SLO, although it is still affected by $\mathtt{R}_{\mathrm{lock}}$ bursts at high load.

B.2 Memcached Performance

Clients generate a modified VAR [5] workload in which 82% of requests are $\mathtt{R}_{\mathrm{lock}}$, 9% are $\mathtt{R}_{\mathrm{cpu}}$, and the remaining 9% are $\mathtt{R}_{\mathrm{mem}}$ on SETUPA. In Memcached, the $\mathtt{R}_{\mathrm{mem}}$ path is the most congested and saturates at around 300 kRPS of offered load. As seen in Figure 16, both SEDA and Protego stop admitting additional load beyond this point, leaving the $\mathtt{R}_{\mathrm{cpu}}$ and $\mathtt{R}_{\mathrm{lock}}$ paths underutilized. SEDA suffers from extremely high $\mathtt{R}_{\mathrm{lock}}$ tail latency because of request incasts, whereas Protego keeps tail latency much better controlled.

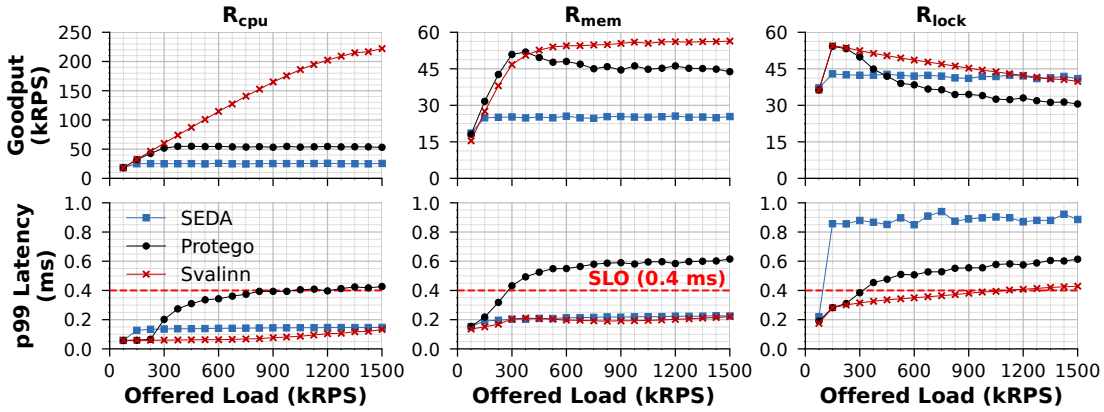


Figure 15: Request-level performance of SEDA, Protego, and Svalinn for synthetic application on SETUPA.

Svalinn addresses the most congested bottleneck, memory bandwidth, using `m_semaphore`. By limiting R_{mem} concurrency, `m_semaphore` preserves CPU capacity for R_{cpu} and R_{lock} requests. As a result, Svalinn achieves up to 2.53× higher R_{cpu} goodput than SEDA and Protego, and up to 2.58× higher R_{lock} goodput than both baselines. Svalinn also achieves up to 13.57×, 1.74×, and 17.32× lower tail latency for R_{cpu}, R_{mem}, and R_{lock} requests, respectively. These gains come from performing AQM at both bottlenecks: memory bandwidth and the global slabs lock on the `SET` path. Tail latencies remain well below the SLO because `m_semaphore` uses an aggressive AQM policy that tolerates no queueing and drops requests as soon as a queue is about to form. This slightly increases drops, but achieves the lowest possible latency.

B.3 Choice of `m_semaphore` Reward Function

In Svalinn, we introduced an MAB-based controller to dynamically update the `m_semaphore` capacity and outlined the reward function that best meets our objective. To design an appropriate reward, we built a simulator that runs the ε-greedy MAB algorithm for our specific problem: finding the minimum number of cores needed to saturate memory bandwidth. We model SETUPB, where 16 cores saturate memory bandwidth, and design reward functions that (i) increase with achieved bandwidth and (ii) penalize using more cores than necessary. We define the normalized memory bandwidth at a given `m_semaphore` capacity as $\hat{m} = \frac{membw_{capacity}}{membw_{max}}$ and the normalized core count as $\hat{c} = \frac{cores_{capacity}}{cores_{max}}$. We evaluate three variants shown below:

$$reward_{simple} = \alpha \cdot \hat{m} - (1-\alpha) \cdot \hat{c}$$

$$reward_{hardgate} = \begin{cases} \alpha \cdot \hat{m}, & \text{if } \hat{m} > 0.98, \\ \alpha \cdot \hat{m} - (1-\alpha) \cdot \hat{c}, & \text{otherwise.} \end{cases}$$

$$reward_{softgate} = \alpha \cdot \hat{m} - Sigmoid_k(\hat{m} - 0.98)(1-\alpha) \cdot \hat{c}$$

We simulate a simple bandwidth model that increases linearly up to 16 cores and plateaus thereafter, running experiments with and without Gaussian noise (3 GB/s). As shown in Figure 17, all three reward functions converge near the optimal core count when the readings are noise-free: $reward_{simple}$ converges exactly to 16, while the other two settle at 15.

With noise, however, $reward_{hardgate}$ and $reward_{softgate}$ converge to core counts much larger than the optimum. Because noise frequently masks whether $\hat{m}$ exceeds 0.98, the penalty condition rarely triggers, allowing high-core configurations to appear equally good or better. Their learned reward curves reflect this by assigning maximal rewards to all core counts beyond the true optimum, preventing stable convergence. In contrast, $reward_{simple}$ consistently converges to the correct core count, and its learned reward curve peaks at the true optimal arm. For this reason, we adopt $reward_{simple}$ in Svalinn.

B.4 Sensitivity to Reward Function Parameter

Our reward function includes a parameter $\alpha \in [0,1]$ that controls the tradeoff between maximizing memory bandwidth and minimizing CPU usage. We evaluate different α values using the same simulator. We disable the memory bandwidth noise in this experiment. As shown in Figure 18, when $\alpha \leq 0.3$, the learned rewards favor core counts below the optimal value. For α between 0.3 and 0.9, the reward peaks exactly at the true optimal core count. At $\alpha = 1.0$, all core counts above the optimal receive the same reward, leading to oscillatory behavior similar to $reward_{hardgate}$ and $reward_{softgate}$.

B.5 Performance with Different Request Mixes

We use the synthetic workload with R_{cpu} and R_{mem} requests on SETUPA. We configure the offered load so that the R_{mem} path is overloaded by 20%, 60%, or 100%. For each overload level, we vary the request mix by increasing the R_{cpu} fraction from 10% to 70%, while keeping the R_{mem} overload level fixed. This lets us evaluate whether Svalinn's benefit holds across different workload mixes, rather than only for one engineered mix.

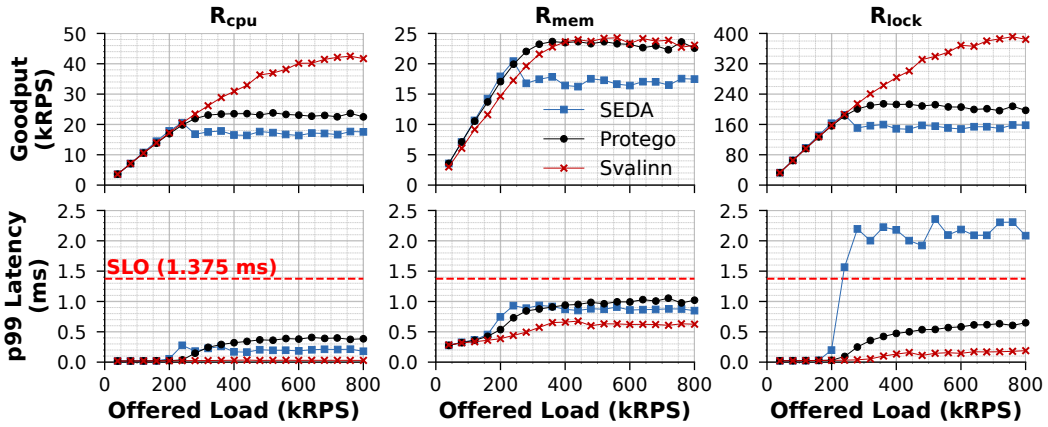


Figure 16: Request-level performance of SEDA, Protego, and Svalinn for Memcached on SETUPA.

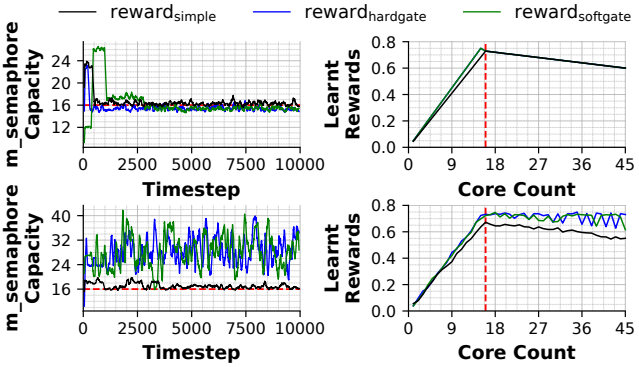


Figure 17: Simulation of ε-greedy MAB controller comparing different reward functions without memory-bandwidth noise (top) and with memory-bandwidth noise (bottom). The red line indicates the optimal core count.

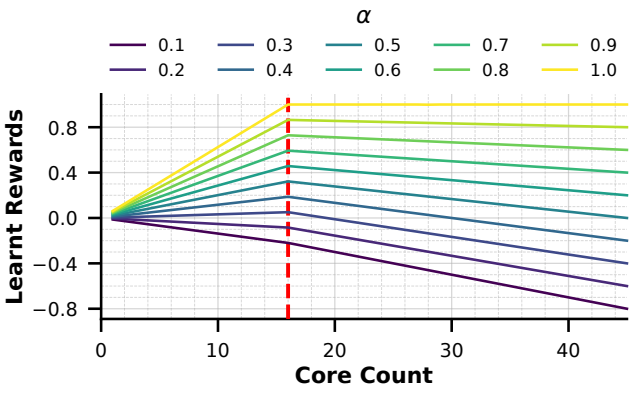


Figure 18: Simulation of ε-greedy MAB controller to analyze the sensitivity to parameter α. The red line shows the optimal core count.

Figure 19 shows the goodput gain of Svalinn over Protego. Svalinn achieves the largest gains when the workload is dominated by R_{mem} requests. In this regime, Protego saturates memory bandwidth early. The stalled R_{mem} requests

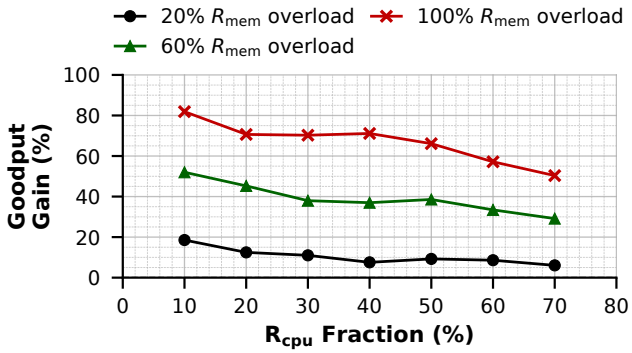


Figure 19: Goodput gain of Svalinn over Protego across different R_{cpu}/R_{mem} request mixes and R_{mem} overload levels on SETUPA.

then occupy CPU cores and leave less capacity for useful R_{cpu} work. Svalinn avoids this by limiting R_{mem} concurrency with `m_semaphore`, preserving CPU capacity for R_{cpu} requests. As the R_{cpu} fraction increases, memory bandwidth becomes congested later, allowing Protego to achieve more R_{cpu} goodput before R_{mem} stalls dominate. Consequently, the gap between Svalinn and Protego narrows at higher R_{cpu} fractions. Overall, Svalinn's goodput gains persist across a range of R_{cpu}/R_{mem} mixes and R_{mem} overload levels.

ARTIFACT EVALUATED
USENIX
AVAILABLE

PeeR: First-Class Scheduling for Latency-Critical eBPF Applications

Jeremy Carin *MIT CSAIL*
Ben Holmes *MIT CSAIL*
Weiyang Wang *MIT CSAIL*
Ankit Bhardwaj *Tufts University*
Manya Ghobadi *MIT CSAIL & Systalyze*

Abstract

We present PeeR, a novel eBPF runtime that makes latency-critical eBPF programs preemptable and schedulable while maintaining low overhead. As eBPF programs grow more complex, they expose a fundamental gap: performance-critical hooks execute in a non-preemptable `softirq` context, which is invisible to the scheduler. These programs bypass resource controls, break isolation, and cause head-of-line blocking, degrading tail latency. PeeR exploits two key properties of eBPF: the verifier enforces clean program state at helper function boundaries, making these sites natural preemption points, and non-trivial programs frequently call helpers, ensuring fine-grained preemption opportunities. Building on these properties, PeeR brings *cooperative preemption* to eBPF: lightweight budget checks, inserted at each helper call, force programs exceeding their budget to yield and resume later on per-CPU kernel threads. To handle a wide variety of workloads, PeeR uses a two-level scheduling model that integrates with `sched_ext`, where the outer level controls aggregate CPU time for eBPF workloads, while an inner micro-scheduler orders individual tasks according to an operator-defined policy. Our evaluation on Redis, Memcached, echo-server, and TPC-C workloads shows that PeeR reduces p99 latency for latency-sensitive requests by 3× to 19.8× over the current eBPF runtime, without starving competing long-running requests.

1 Introduction

eBPF (extended Berkeley Packet Filter) [26] is becoming the de facto standard for building *latency-critical* networking and storage applications inside the Linux kernel. By allowing applications to execute custom logic inside the kernel, eBPF avoids costly context switches and redundant data copies from the critical path, enabling microsecond-scale processing for networking and storage workloads. While kernel-bypass frameworks (e.g., DPDK [47], SPDK [48]) achieve similar performance, they sacrifice kernel services such as isolation, scheduling, and multi-tenancy. eBPF instead delivers high performance while preserving the kernel's safety and resource-sharing model, driving its widespread adoption across industry and academia.

With growing adoption, eBPF programs are becoming increasingly complex. Developers are now offloading complicated application logic into the kernel, including key-value stores [9, 28], distributed transactional systems [58], load balancers [33], DDoS protection [23], transport protocols [6], and storage engines [56]. These programs often run for tens of thousands to millions of instructions, resulting in execution times of hundreds of microseconds.

While eBPF programs have grown dramatically in complexity, their execution model, which was originally designed for tiny packet filters, has remained largely unchanged. On latency-critical I/O paths, each eBPF invocation runs to completion in arrival order and cannot be preempted, so later latency-sensitive invocations wait behind long-running ones. Another important aspect of eBPF's execution model, for latency-critical applications, is that it runs in software interrupt context (`softirq`) [45], allowing quick and low-overhead execution upon I/O events.

The current eBPF execution model creates two fundamental problems for modern workloads. First, because these latency-critical eBPF programs run in `softirq` context, their execution time is accounted to the interrupted userspace process, rather than the eBPF program itself. To make matters worse, eBPF execution begins immediately after an I/O interrupt handler, abruptly preempting any colocated userspace process. This misaccounted and interrupt-driven execution leads to unfair CPU allocations that starve colocated workloads and introduce unpredictable interference. Second, the lack of preemption means that long-running eBPF programs block later invocations, increasing latency for small and latency-critical invocations, defeating the very purpose of in-kernel execution.

We find that this mismatch has severe practical consequences. For a colocated scenario, Redis-KFlex, a Redis-compatible store built on KFlex [9], running under a fair-share scheduler consumes up to 90% of CPU despite receiving only a 50% allocation, starving colocated workloads. This problem is especially concerning in modern datacenter environments, where colocation is the norm due to CPU oversubscription and utilization-driven scheduling policies, leading to unpredictable performance interference. Beyond unfairness, the inability to preempt long-running programs inflates tail latency. For a mixed point-scan workload, Redis-KFlex experiences 7.4× higher p99 latency when short requests are blocked behind long eBPF handlers as compared to a preemptable execution model (§3). Taken together, these effects show that the kernel's existing execution model no longer provides the control over the eBPF fast path that applications and developers expect.

To address these issues, this paper introduces PeeR, a runtime that makes eBPF programs schedulable while preserving their low-overhead execution model. PeeR achieves this by transforming each eBPF invocation into a schedulable *eBPF task*, subject to CPU accounting, preemption, and scheduling policy enforcement. First, PeeR attributes and accounts eBPF execution time, making eBPF visible and accountable to the scheduler. Second, PeeR extends the traditional `softirq` execution model with a hybrid `softirq`-worker thread model. eBPF tasks run in `softirq` context as long as they remain within their execution budget;

once the budget is exceeded, PeeR safely preempts the task using a lightweight preemption mechanism and resumes it on a worker thread. Finally, instead of enforcing a fixed scheduling policy, PeeR integrates with Linux's extensible scheduler interface (`sched_ext`) [49], allowing flexible scheduling policies to control CPU allocations within eBPF and across applications.

Designing PeeR requires addressing three key challenges. The first challenge is enabling *low-overhead preemption* of eBPF tasks without violating their safety and correctness guarantees. A naive approach is to use existing timer-based interrupts to preempt long-running eBPF tasks. However, timer interrupts incur high overheads, negating the performance benefits of eBPF. More importantly, arbitrary interruption risks leaving the eBPF task execution state inconsistent, violating eBPF's safety guarantees. Our key insight is that eBPF's design inherently provides natural preemption points. Because eBPF restricts interaction with kernel state to helper functions, *helper-call boundaries* are safe and well-defined preemption points that PeeR leverages to enable preemption.

To address this, PeeR instruments the JIT compiler to inject budget checks at each helper call site, transforming helpers into lightweight, verifier-safe preemption points (§5.1). When a task exceeds its budget, PeeR forces it to yield execution at the next helper call, avoiding arbitrary mid-execution preemption that breaks eBPF's correctness guarantees. We bring this technique, known as *cooperative preemption*, to the eBPF runtime.

The second challenge is *correct context recovery at resumption*. When a long-running eBPF task exceeds its budget, PeeR must preempt it in `softirq` context and later resume it on a worker thread. Resuming execution in a different context, however, introduces subtle correctness issues. The pointers stored on the eBPF task's stack during `softirq` execution are invalid when the task resumes on a worker thread, as they point to kernel stack memory that is no longer valid in this new context. Such invalid pointers lead to memory-safety violations and incorrect program behavior.

To address this, PeeR leverages the strong type information maintained by the eBPF verifier to automatically patch pointers in the saved context during resumption, ensuring correctness without any changes to the eBPF programs (§5.2). PeeR does this by exporting the verifier's compile-time type information for use at runtime. These two mechanisms guarantee safe preemption and resumption of eBPF tasks.

The third challenge is *efficient integration with the scheduler*. To handle this, PeeR integrates with Linux's extensible scheduler interface, `sched_ext`. PeeR exposes eBPF execution time to `sched_ext` so the kernel correctly tracks CPU consumption and enforces allocation policies. Further, PeeR enables a unified and configurable scheduling framework for both eBPF tasks and userspace processes. PeeR provides two layers of configurability: (1) inter-application CPU allocation, which governs how PeeR schedules its worker threads and other colocated processes, and (2) intra-eBPF task scheduling, which determines the order in which eBPF tasks are executed on each worker thread (§7).

We implement PeeR atop the Linux kernel 6.16, extending the eBPF runtime and scheduler with ≈ 5000 lines of code. We evaluate PeeR with several popular workloads (Redis, Memcached, echo-server, TPC-C [51]). Our evaluation shows that PeeR reduces p99 latency for latency-sensitive requests by 3× to 19.8× over the current eBPF runtime, without starving competing long-running requests within the same application (§8.1). We implement multiple scheduling policies (FIFO, shortest-remaining-processing-time, weighted round-robin) using PeeR's scheduler interface and show that PeeR enforces operator-defined resource-sharing goals across colocated eBPF and userspace applications (§8.2). We quantify PeeR's overhead for a preemption-resumption cycle to be 247 ns, making PeeR suitable for fine-grained scheduling even for microsecond-scale workloads (§8.3). PeeR is, to our knowledge, the first system to make eBPF programs preemptable and schedulable in `softirq` context. We envision PeeR as a foundation for future latency-critical eBPF applications, enabling scheduling models beyond the reach of today's kernel.

PeeR is open source and available at `github.com/hipersys-team/PeeR`.

2 Background

eBPF [26] allows users to execute verified programs at predefined kernel hook points. eBPF use cases fall into two categories: kernel extensibility (e.g., tracepoints and LSM hooks) and latency-critical I/O processing (e.g., XDP and `tc` hooks). This paper focuses on the latter, where eBPF programs run on performance-critical paths.

Users implement eBPF programs in a restricted subset of C; the programs are then compiled into eBPF bytecode and loaded into the kernel. Before execution, the eBPF verifier performs static analysis to ensure memory safety, bounded control flow, and non-blocking behavior. Once verified, a just-in-time (JIT) compiler translates the bytecode into native machine instructions, enabling near-native execution speed.

eBPF enforces a restricted execution model. The bytecode targets a simple abstract machine with no direct access to kernel memory or functions [12]. Programs interact with the kernel only through predefined *helper functions* and kfuncs [11, 39] (referred to as *helper functions* hereafter), ensuring safe, mediated access to kernel services. Together, these mechanisms enable eBPF to extend the kernel, even on performance-critical paths, without compromising system integrity.

Execution model of eBPF on the fast path. For latency-critical eBPF hooks on the I/O-path, the kernel invokes an eBPF program each time an I/O event occurs (e.g., packet ingress or egress). These programs execute inside `softirq` context because their hook points lie on the kernel's interrupt-driven I/O paths. In `softirq`, preemption is disabled: CPU time spent there remains invisible to the scheduler and is effectively charged to the process that was preempted by the interrupt. This avoids context-switch overhead.

This design reflects what we call the *fast-path assumption*:

Application	Static Analysis (largest program)			Execution
	#Ins	#Calls	Instrs. between helper calls	p50/p99
Katran [33]	2,587	88	28±54	0.3/7.6
Cilium [1]	7,622	233	32±33	1.5/22
Memcached (BMC) [28]	289	11	21±27	2.1/37
Redis-KFlex [9]	1,356	95	12±10	0.5/248
DINT [58]	1,373	25	48±42	0.3/1.1
eTran [6]	1,194	14	79±102	0.4/1.4
RocksDB (XRP) [56]	891	17	27±26	4/882

Table 1: Characterization of common eBPF applications. We report static properties of each application's *largest single eBPF program* (one tail-call target plus the bpf-to-bpf subprograms it calls), not the sum over its tail-called suite. #Ins is the instruction count, #Calls the helper and kfunc call sites, and the last static column is the mean±stddev number of instructions between consecutive helper or kfunc calls. Because PeeR yields only at these calls, this gap bounds its preemption granularity. All programs contain loops, and p50/p99 (μs), measured under representative workloads, differ by up to 500×.

work inside `softirq` completes quickly and uniformly, so the impact on the preempted process is negligible. Under this assumption, request processing proceeds in strict first-come, first-served (FCFS) order and continues as long as the CPU has available budget for `softirq` execution. If processing in `softirq` exceeds a threshold, the kernel defers later requests to a special kernel thread called `ksoftirqd`. `ksoftirqd` operates at request granularity: it defers entire batches of pending requests, but does not pause a long-running eBPF task mid-execution. This execution model works well for small tasks.

Growing eBPF complexity. eBPF was initially designed for small, predictable programs suitable for fast-path packet filtering. Over time, its capabilities have expanded significantly [19]. The verifier now admits much larger programs with complex control flow, including bounded loops, BPF-to-BPF function calls, and tail calls, permitting long executions whenever a static bound exists [18, 26]. As a result, the verifier's one-million-instruction limit no longer guarantees short or uniform runtimes. Modern features further decouple static limits from execution time. Programs iterate over large data structures stored in maps or auxiliary memory regions [9], and runtime behavior often depends on workload-specific states, raising questions about whether the fast-path assumption still holds.

3 Measurement Study

We study real-world eBPF applications using static analysis and runtime profiling. Our measurements show that modern programs are complex enough to break the fast-path assumption, causing latency inflation and unfair CPU allocation.

3.1 Quantifying eBPF Application Complexity

We characterize seven recent eBPF applications spanning load balancing, networking, key-value storage, and storage. For each application we statically analyze its compiled BPF objects to extract, per program, the instruction count, the number of helper and kfunc call sites, the presence of loops, and the distribution of instruction intervals between consecutive helper or kfunc calls. We measure one eBPF program at a time, defined as a single tail-call target together with the bpf-to-bpf subprograms it invokes, rather than summing across the tail-called suite; all counts and intervals are derived from the disassembled bytecode. We then deploy each application under representative workloads and measure execution-time distributions using BPF-based tracing. Table 1 summarizes our findings.

A single eBPF program is already large and helper-dense. The largest Cilium program alone spans 7,622 instructions and 233 helper calls, and Katran's balancer runs 2,587 instructions with 88 helper calls, even though each suite compiles into many tail-called programs (33 for Cilium). Programs with loops also exhibit large variance between median and tail latency due to data-dependent execution paths. Redis-KFlex implements a skiplist where point queries complete quickly, but range scans traverse nodes linearly, a 500× difference between p50 and p99. RocksDB (XRP) linearly scans up to 2,000 entries per data block, inflating tail latency. Finally, because PeeR can preempt only at helper and kfunc calls, the gap between consecutive calls bounds its preemption granularity. These gaps can be large, with maximum gaps of 484 instructions for Katran and 286 for eTran, and the longest occur in compute loops with no intervening call: XRP's data-block scan executes a roughly 100-instruction loop body with no helper call up to 2,000 times. Iterator-based loops are an exception, as Redis-KFlex emits a kfunc on each iteration and keeps its maximum gap to 44 instructions.

Our analysis confirms that while the verifier preserves eBPF's safety guarantees, the assumption that programs complete quickly and uniformly no longer holds.

3.2 Breaking Fast-Path Assumptions

To understand the consequences of this mismatched assumption, we construct experiments that illustrate two problems: *latency inflation* within an application and *unfair CPU allocation* across applications.

Intra-application latency inflation. Due to *non-preemptive* and FCFS execution, long-running eBPF tasks block shorter ones, increasing latency for latency-critical operations.

To illustrate this problem, we run eBPF-based key-value stores, a typical use case for eBPF with XDP hooks [9, 28, 58]. For simplicity, we focus on XDP, but these issues generalize to other fast-path hooks. In the first experiment, we run a bimodal point-and-scan workload with 99.5% short (0.2 μs) and 0.5% long requests (200 μs), a mix common in key-value stores. Redis-KFlex handles requests on eight CPU cores generated by an open-loop load generator. More details are in Section 8.

Figure 1 shows the p99-latency of point queries with increasing load. We compare this with a hypothetical eBPF system that supports preemption with a 5 μs time slice (not achievable in current kernels). Even at low load, the current eBPF implementation exhibits high p99 latency due to head-of-line blocking from long scan requests. As load increases, this effect worsens, resulting in up to 7.4× higher p99

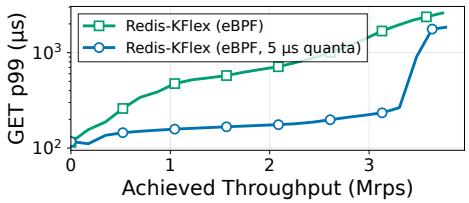


Figure 1: Latency inflation within an application. In eBPF, long-running SCAN operations (200 μs each) block GET queries, increasing GET (0.2 μs each) p99 latency. With 5 μs preemption quanta, point queries p99 latency improves by up to 7.4×.

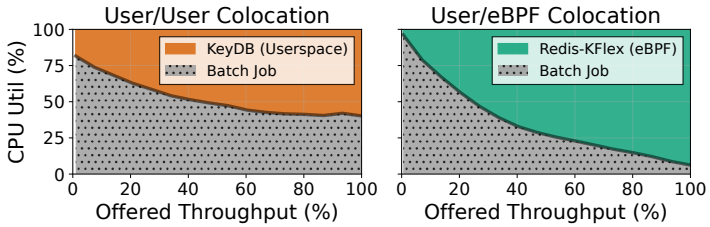


Figure 2: Unfair CPU allocation for colocating eBPF and userspace jobs. CPU utilization breakdown when a colocated job (allocated 50% CPU) shares cores with KeyDB (multithreaded userspace, left) and Redis-KFlex (eBPF, right). The scheduler correctly throttles KeyDB, but Redis-KFlex starves the colocated job because it executes in a context invisible to the scheduler.

latency compared to the preemptable-eBPF implementation.

Offloading to `ksoftirqd` does not resolve the blocking. `ksoftirqd` offers no way to prioritize short tasks. Thus, an eBPF task runs to completion regardless of whether it executes in `softirq` or `ksoftirqd` context, and shorter tasks always wait behind longer ones.

Inter-application unfairness. Across applications, long eBPF execution allows a single workload to monopolize CPU time because the eBPF execution time remains invisible to the scheduler. Moreover, most I/O-intensive eBPF hooks run in interrupt context, so the kernel must execute them immediately upon I/O events. These eBPF workloads monopolize CPU and starve colocated tasks, violating fairness and performance isolation.

We demonstrate this by extending the previous setup to colocate a CPU-bound job on the same eight CPU cores with the key-value store, configuring Linux's scheduler to allocate up to 50% of CPU resources to each job. We compare userspace KeyDB [37] against Redis-KFlex to highlight the differences in scheduling behavior. Figure 2 (left) shows CPU utilization breakdown as we vary the load on KeyDB. At low load, the CPU-bound job uses most of the CPU, as expected. However, as KeyDB load increases, the scheduler allocates CPU time fairly between KeyDB and the CPU-bound job, throttling KeyDB as needed. Figure 2 (right) shows the same experiment but with Redis-KFlex. At low load, the colocated job again gets its fair share of CPU time. However, Redis-KFlex starves the colocated job as load increases, using 90% of the CPU because XDP executes in an interrupt context invisible to the scheduler. Prior systems, such as DINT [58] and eTran [6], have observed similar unfairness, corroborating our findings.

Deferring to `ksoftirqd` does not restore fairness: the kernel charges CPU time to `ksoftirqd` itself rather than the application that installed the hook, leaving the scheduler unable to distinguish between tenants or enforce per-application limits.

3.3 Challenges in Making eBPF Schedulable

The current eBPF runtime lacks mechanisms for schedulability and execution-time attribution. Our goal is to bridge this gap by making eBPF programs schedulable without sacrificing safety or performance. However, making eBPF schedulable is challenging because its execution model differs fundamentally from that of other kernel work. We identify three key challenges:

- **Enabling fine-grained preemption with low overhead.** Performance-critical eBPF programs are invoked at high frequency. Any scheduling mechanism must preserve fast-path performance when preemption is unnecessary, and impose minimal overhead when preemption occurs.
- **Preserving eBPF correctness under preemption.** The eBPF verifier reasons about programs as single, uninterrupted executions and assumes atomic completion. Introducing preemption risks violating assumptions about register state, stack contents, and helper function behavior. A schedulable eBPF must preserve these correctness guarantees without requiring uninterrupted execution.
- **Providing expressive scheduling abstractions.** eBPF currently has no interface to interact with schedulers. A schedulable eBPF requires new abstractions that are expressive enough to support per-task priorities, budgets, and policies.

Addressing these challenges requires rethinking the execution model of eBPF programs on the I/O path. In the next section, we present PeeR, a system that achieves schedulability while preserving eBPF's safety and performance guarantees.

4 PeeR: Preemptable eBPF Execution Runtime

PeeR is a new eBPF runtime that makes latency-critical eBPF programs preemptable and schedulable in `softirq` context, where preemption is otherwise disabled. We implement PeeR for native XDP and describe extending it to other hooks in §6. We build PeeR on a simple but important observation: non-trivial eBPF programs must interact with the kernel through eBPF helper functions and kfuncs, as shown in Table 1. To safely call into kernel code, the verifier already tracks strict invariants at helper boundaries: held locks, borrowed references, and incomplete map operations. PeeR exploits this structure for *cooperative preemption*. Helper boundaries are candidate preemption points where the verifier has precise register and stack types. PeeR enables yielding at a site only when the verifier reports no unsafe resource is live and the enclosing hook has passed the audit in §6.

Figure 3 summarizes PeeR's key components. PeeR adopts

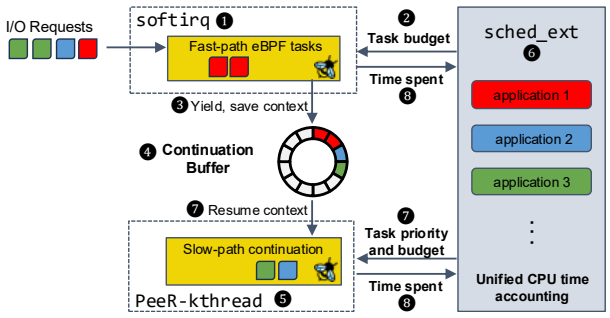


Figure 3: PeeR's design overview. eBPF tasks execute in `softirq` under `sched_ext`-provided budgets. Tasks exceeding budget yield to a continuation buffer and resume on PeeR-kthreads. Both paths report CPU time to `sched_ext` for unified accounting.

a hybrid execution model: short tasks run in `softirq` as a *fast path*, while long-running tasks are deferred to per-CPU PeeR worker kernel threads (PeeR-kthreads) as a *slow path*. On the surface, this model resembles `ksoftirqd`'s threaded model, but PeeR operates at eBPF task level with distinct mechanisms for budget checks, control-flow transfers, and scheduling. In this work, tasks remain on their originating CPU and are not migrated.

Fast path. Each eBPF task enters `softirq` ❶ with a configurable budget ❷, a global preemption quantum set by `sched_ext` (§7.2). Budget checks occur at helper call sites: PeeR augments eBPF's JIT compiler to wrap each helper with a prologue that checks remaining budget (§5.1). This turns helpers into precise, low-overhead preemption points compatible with the verifier. Tasks within budget enter the helper with minimal overhead and run to completion in FCFS order. Tasks that exceed their budget yield ❸: PeeR saves their execution state, consisting of a 512-byte stack, 11 registers, and program context, to a per-CPU array. It then returns control to `softirq` immediately, allowing subsequent I/O to proceed. Upon exiting `softirq`, PeeR flushes yielded tasks into a continuation buffer ❹ and wakes the local PeeR-kthread.

Slow path. Yielded tasks resume on PeeR-kthreads ❺ (§5.2). PeeR instantiates one PeeR-kthread per CPU at boot time. Scheduling occurs at two levels: at the outer level, `sched_ext` schedules PeeR-kthreads as ordinary kernel threads, controlling the aggregate CPU time all eBPF tasks on the slow path receive ❻. At the inner level, a *micro-scheduler* running within each PeeR-kthread orders eBPF tasks ❼. The micro-scheduler pulls continuations from the buffer, restores their state, and resumes them in policy-determined order. Resumed tasks follow the same budget enforcement; if a task yields again, control returns to the micro-scheduler, which requeues it and selects the next task. This two-level model enables operators to implement policies such as priority scheduling or shortest-remaining-processing-time (§7). Both paths report CPU time to `sched_ext` ❽, closing the feedback loop for system-wide scheduling.

5 Cooperative Preemption

Preemption sits at the core of making eBPF schedulable. For eBPF programs on I/O paths that execute millions of times per second, the preemption mechanism must be lightweight while preserving eBPF's correctness guarantees. PeeR therefore avoids traditional interrupt-based preemption, whose timer signals and context switches would be too costly on the fast path, and instead adopts a *cooperative preemption* model.

In classical cooperative scheduling, threads run until they explicitly call `yield()`, giving up the CPU at well-defined points. This avoids frequent interrupts and reduces context-switch overhead, but relies on applications to correctly insert yield points. PeeR retains these benefits while automating the process: rather than requiring programs to call `yield()`, PeeR enforces budgets and checks for preemption at *eBPF helper sites*, i.e., invocations of helper functions that eBPF programs use to interact with the kernel. These sites are natural points for preemption: helpers are fundamental to any non-trivial eBPF program, and the verifier already enforces strict pre- and post-conditions at these call sites (e.g., no held locks, valid pointers, and constrained stack usage).

Figure 4 illustrates a state machine of this model. Each helper site triggers a *preemption check*. If the program remains within budget, execution continues into the helper. If the budget has expired, however, the program starts *yielding*: PeeR invokes a yield handler that saves the current execution state, including preemption site identifier, eBPF stack, and registers (§5.1). The handler then transfers control back to the kernel. The program enters a *preempted* state until `sched_ext` reschedules it (§7), when the program starts *resuming*. PeeR restores the saved state, performs necessary pointer patching, and resumes at the original preemption site (§5.2).

Cooperative preemption via compiler-inserted yield points is well established. Wasmtime [4] inserts epoch checks at function entries and loop headers, and low-latency schedulers approximate preemption for userspace RPC handlers through compiler instrumentation: Concord [34] polls a dispatcher-written cache line, while Compiler Interrupts [2] and TinyQuanta [40] trigger yields from instruction-count-driven probes. Go [7] instead preempts asynchronously with signals, using compiler-emitted metadata to avoid unsafe points. Like these systems, PeeR may yield only where program state is safe; Concord, for example, avoids yielding inside locks or external libraries. PeeR faces analogous but distinct constraints. Rather than choosing yield points heuristically, it reuses the eBPF verifier's helper-boundary invariants (no locks held, valid pointers) and adds eBPF-specific rules that forbid yielding while RCU-protected pointers or other BPF-held resources are live. PeeR adds two mechanisms that userspace runtimes do not require. It rebases `PTR_TO_CTX` and `PTR_TO_STACK` values across a context switch using verifier-derived type information (§5.2), and it resumes through CFI-safe trampolines.

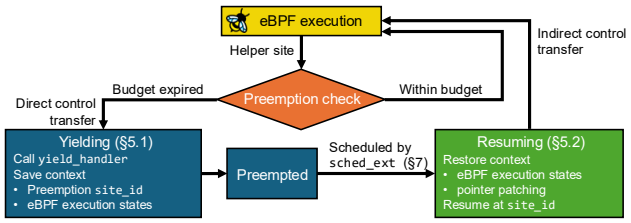


Figure 4: PeeR's state diagram. At each helper site, PeeR checks the task's budget. Tasks within budget continue; tasks over budget yield, saving their context until `sched_ext` reschedules them.

```
1     call preempt_check        ; check if execution budget has expired
2     test eax, eax             ; check if eax != 0 (should preempt)
3     jnz  yield_site_1         ; jump to yield_site_1 (line m) if eax != 0
4
5   helper_site_1:              ; address of helper site 1
6     call bpf_map_lookup_elem  ; call helper function
7     mov  [elem], eax          ; store helper return value
...
m   yield_site_1:               ; atomic state saving for site_1
m+1   mov  r11, 1               ; load site_id (1) to r11
m+2   jmp  yield_handler        ; execute yield handler; store eBPF context
...
```

Figure 5: JIT-augmented assembly for a `bpf_map_lookup_elem()` helper site. The preemption check (orange) tests a global flag and conditionally jumps to the yield stub (blue), which records the site identifier before invoking the yield handler.

5.1 Preemption Path

PeeR augments helper functions to implement preemption at helper sites. Since eBPF is JIT-compiled, PeeR modifies the JIT compiler to emit the necessary code snippets for preemption budget checks and state saving.

Preemption check and yield handler in assembly. Figure 5 shows how PeeR's JIT compiler augments an eBPF helper site, `bpf_map_lookup_elem()`, in pseudo x86 assembly. PeeR assigns a unique `site_id` (line 5 shows site 1), enabling it to identify where a program was preempted for later resumption. During JIT compilation, PeeR inserts a short preemption check prologue (orange highlight, lines 1 to 3) immediately before each helper site. Because helpers are called frequently, this check must be lightweight. Furthermore, it needs to preserve register values to save the execution state to the continuation buffer.

The `preempt_check()` call on line 1 evaluates a per-CPU preemption flag; the scheduler sets this flag at the end of each budget period (details in §7.2). We measure that the prologue costs one to two cycles when the flag is not set (§8.3), adding negligible overhead to fast-path execution. This flag is retrieved into register `eax` (line 2). Since `eax` would be overwritten by the helper call regardless, reusing it is safe; the function also uses the `no_caller_saved_registers` GCC attribute to avoid clobbering BPF argument registers. If the flag indicates the budget has expired (line 3), PeeR jumps to a per-site stub (blue highlight) that stores the `site_id` into x86 register `r11` (line `m+1`), a register not used by eBPF programs. Control then transfers to a shared yield handler (line `m+2`) that saves execution state and returns control to `softirq` or the worker thread.

Execution state capture. When the budget check triggers a yield, PeeR captures all execution state without clobbering live registers or stack contents. The shared handler at line `m+2` immediately saves state into a continuation buffer of type `struct peer_cont` before any other operations. An eBPF program's execution state consists of 11 registers (88 bytes), a 512-byte stack, and a program-specific context. The context type varies by hook; for instance, XDP programs receive `struct xdp_md`, and socket programs receive `struct __sk_buff`. Since the hook type is known at compile time, PeeR can generate type-specific context-saving routines for each program. The handler also stores the `site_id` and a reference to the current task. Because registers and stack are laid out contiguously, the handler copies all 600 bytes with a single `memcpy()`, then saves the context and returns control to the kernel.

Context saving for XDP. In this paper, we implement context saving for XDP, one of the most performance-critical hooks. In Linux's networking stack, the NIC driver polls incoming packets in batches via *NAPI polling*; XDP programs execute immediately upon arrival, before socket processing or routing. The XDP context, `struct xdp_md`, contains pointers to the packet's start and end. Saving this struct alone is insufficient: packets are allocated from the driver's buffer pool and remain valid only within the current NAPI poll cycle.

To extend packet lifetime without copying, PeeR rebinds the packet to an independently owned frame managed via reference counting. This preserves packet data in place and stores minimal metadata in the buffer's headroom. PeeR then returns a new verdict, `XDP_YIELD`, in place of the program's. The driver treats `XDP_YIELD` as a consumed packet and continues its NAPI poll, so the fast path never blocks waiting for the deferred result. The continuation records a pointer to this frame; upon resumption, the worker reconstructs a consistent packet view from the same underlying buffer. After the program completes, PeeR decrements the reference count and returns the buffer to the pool.

5.2 Continuation Path

PeeR wakes the local PeeR-kthread when fast-path execution finishes, or `sched_ext` does so when it schedules the kthread. The micro-scheduler (§7.3) selects the next eBPF program from the continuation buffer; the PeeR-kthread then restores its state and resumes execution from the preempted helper site on the slow path.

Pointer patching. Resuming a preempted eBPF program requires restoring execution state to registers and memory. However, pointer values saved at yield time become invalid when execution resumes on a worker thread: the stack resides at a different address, and the execution context has been remapped. Naively restoring stale pointers causes memory corruption. PeeR has access to the old and new base addresses, but patching is not straightforward: a 64-bit value could be a pointer or an integer, and the kernel retains no type information at runtime.

PeeR extracts type information from the eBPF verifier. In eBPF's calling convention, `r10` serves as the frame pointer (stack base) and `r1` holds the context pointer `ctx` at program entry. The verifier tracks the type of each register

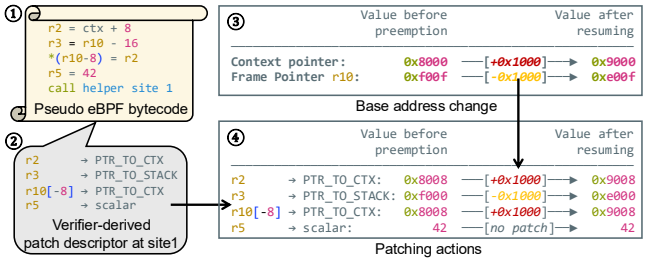


Figure 6: Pointer patching at resumption. ① Pseudo-eBPF bytecode calling helper at site 1. ② Patch descriptor with verifier-derived types at this site. ③ Base address changes between preemption and resumption. ④ PeeR applies pointer rebases according to the patch.

and stack slot at every instruction; two types are relevant for PeeR: `PTR_TO_STACK` for frame-pointer-derived values and `PTR_TO_CTX` for context-derived values. PeeR intercepts this information before JIT compilation, recording which locations hold `PTR_TO_STACK` or `PTR_TO_CTX` values at each helper site in a *patch descriptor*. At resume, PeeR consults the descriptor and rebases each pointer to its new base address.

Figure 6 illustrates this mechanism. ① shows pseudo-eBPF bytecode where `r2` derives from the context pointer, `r3` derives from `r10`, stack slot `r10[-8]` stores `r2`'s value, and `r5` holds a scalar. PeeR leverages the verifier to generate a *patch descriptor* at this site (shown at ②), which stores the type information of variables in registers and on the stack. ③ illustrates that when the program yields and resumes, the context moves from `0x8000` to `0x9000` (`+0x1000`) and the frame pointer from `0xf00f` to `0xe00f` (`-0x1000`). Using the patch descriptor, PeeR rebases pointers according to their types, as shown in ④. It shifts `PTR_TO_CTX` values `r2`, `r10[-8]` by `+0x1000` (`0x8008` → `0x9008`), adjusts `PTR_TO_STACK` value `r3` by `-0x1000` (`0xf000` → `0xe000`), and leaves scalar `r5` unchanged.

One complication arises when multiple control-flow paths converge at a yield site with different types for the same pointer. PeeR uses intersection semantics: a location appears in the patch descriptor only if all paths agree on its type. This is safe because the verifier enforces type consistency for any dereferenced value, hence conflicting pointers are always demoted to scalars and require no patching.

Control-flow transfer under CFI. Finally, the worker must transfer control back to the helper call where the program yielded. This requires an indirect jump, but modern CPUs enforce Control-Flow Integrity (CFI) through Indirect Branch Tracking: indirect jumps must land only on valid entry points marked with special instructions (e.g., `endbr64` on x86), or the CPU triggers a hardware exception. PeeR satisfies CFI by emitting a trampoline for each yield site during JIT compilation. Each trampoline begins with an `endbr64` instruction, marking it as a valid indirect branch target, followed by a relative jump to the corresponding helper call. Trampolines are stored consecutively in the JIT code section, with addresses recorded in an array indexed by `site_id`. At resume, the worker looks up the trampoline for the saved `site_id` and jumps to it. The

Interface	Description
`runtime_event ringbuf`	Per-CPU stream of execution records for accounting
`set_budget(ns)`	Sets the global preemption quantum
`struct peer_cont cont`	Per-task continuation buffer and metadata
`enqueue(cont)`	Handles a newly-preempted eBPF program
`requeue(cont, runtime_ns)`	Handles when an eBPF program is preempted again

Table 2: PeeR's interfaces to schedulers. The top part interacts with the macro-scheduler across threads, while the bottom part implements the micro-scheduler that decides task order within a PeeR-kthread.

`endbr64` satisfies CFI, and the relative jump transfers control to the correct program location. Because PeeR places the yield check immediately before the helper executes, resuming at the helper call runs it exactly once, so a yield never skips or duplicates a helper's side effects.

5.3 Handling RCU Pointer Safety

Many eBPF hooks in `softirq`, such as XDP, execute under an implicit *RCU read-side critical section*. Under *Read-Copy-Update (RCU)*, readers proceed without locks and writers defer freeing data until all pre-existing readers finish. This allows programs to safely dereference pointers returned by helpers like `bpf_map_lookup_elem()` without explicit locking. When `softirq` completes, the critical section ends. This behavior poses a risk for PeeR: if PeeR yields a program holding such pointers, the kernel exits the protected section, risking other programs updating these pointers or even freeing them to cause use-after-free after the task is rescheduled on PeeR-kthreads.

PeeR prevents this by restricting preemption to helper sites where no RCU-protected pointers are live. We again rely on the verifier: its type tracking identifies RCU-protected values, such as map-value pointers (`PTR_TO_MAP_VALUE`) and RCU-tagged kernel pointers (`MEM_RCU`), and its liveness analysis marks a value dead after its last dereference, even if it remains on the stack. PeeR queries this information at each helper site and injects preemption checks only where no RCU values are live. At first glance, this restriction reduces PeeR's preemption opportunities, but in practice it rarely does. The common pattern is to look up an RCU-protected value and dereference it within the same stretch of code, before the next helper call. The pointer is then dead under liveness analysis at the following helper boundary, so that site can still yield. A yield point is forgone only when a program holds an RCU-protected pointer live *across* a helper or kfunc call, which is uncommon. When a program does guard shared state with a BPF spin lock, that region cannot yield: the verifier forbids helper calls while a lock is held, and PeeR, consulting the verifier's lock state (`active_locks`), emits no check at the few data-structure kfuncs still permitted there. Sleepable RCU [41] could lift the restriction entirely by allowing a critical section to span a preemption point; we leave this to future work.

6 Correctness and Hook Safety

PeeR preempts and resumes an eBPF invocation while preserving the safety guarantees of the verifier. Two conditions

are required to guarantee safety: *eBPF-level safety* (§6.1) requires that, at any point where an eBPF program can yield, the program's own state is clean and reconstructable. *Hook-level safety* (§6.2) requires that the kernel path invoking the program can tolerate it being preempted and resumed later. The eBPF verifier guarantees the first requirement. To evaluate the second, we characterize eBPF's common hooks individually and determine whether each is compatible with PeeR.

What PeeR guarantees. PeeR makes a fast-path eBPF invocation *reentrant*: it can yield at a helper boundary, let other work run on the core, and resume from the same site under the same invariants as the fast path, however long it was suspended. Across that yield, PeeR preserves *memory safety* for the three kinds of state an invocation depends on:

1. Its eBPF state, the registers and stack, is saved and restored with each pointer rebased to its new address (§5.2).
2. Its kernel state, any RCU-protected object or acquired reference, stays valid because PeeR never yields while one is live (§5.3).
3. Its hook context, the program's input, stays valid past the original call. For XDP, PeeR retains the packet as an `xdp_frame` (§5.1).

What PeeR does not guarantee. Similar to prior work on preemptive scheduling [34, 35, 40], PeeR allows for interleaving of eBPF program invocations. Normally on the `softirq` fast path, an eBPF invocation runs uninterrupted, so an eBPF application may be written with the assumption that accessing shared state is atomic. Under PeeR, a later invocation can run while a program is preempted and modify that state, so the program might read a map entry, yield, and resume to find a different value. Logic written on the assumption that the entry is stable across the invocation can then incur a race condition. As a result, the program must synchronize explicitly, for example by guarding the access with a spin lock. This is already the case for existing eBPF programs that run in preemptable context, and PeeR extends this requirement to eBPF programs running in a previously non-preemptable context. This interleaving can also change per-core completion order. If a program yields and a later invocation runs on the same core, the later one can complete before the first resumes, so a program that requires per-core completion order must serialize explicitly.

6.1 eBPF-level safety

PeeR establishes eBPF-level safety with two conditions, both derived from the eBPF verifier. First, it yields only at a helper function boundary where the verifier has already proved the call safe and reports no eBPF lock held, so the program's registers, stack, and pointers all carry assigned types. This leaves a clean snapshot that PeeR can later reconstruct. Second, it yields only where the verifier reports no outstanding eBPF resource, such as a live RCU-protected pointer or an acquired reference, since yielding while one is live could let the kernel reclaim it under the suspended program (§5.3). The verifier guarantees both conditions; a site that cannot satisfy them emits no yield point.

6.2 Hook-level safety

The verifier exists only within eBPF context and does not have visibility into the surrounding kernel context. A hook must meet three conditions for compatibility with PeeR:

1. The program's input stays valid after it yields. For XDP, PeeR holds the packet past the original call.
2. The caller does not require the result immediately, so it can continue while PeeR applies the result later.
3. The calling kernel code holds no lock while the program runs.

Resumption runs in process context on an ordinary PeeR-kthread, not inside `softirq`. Deferring eBPF work to a kernel thread has the same properties that already exist when the kernel hands `softirq` processing to `ksoftirqd`. The difference is in granularity; PeeR defers a single eBPF invocation rather than a batch. Before resuming, the PeeR-kthread re-creates the environment the hook runs in, so the continuation executes under the same invariants as the fast path. Below we describe how XDP meets each condition.

Input lifetime. The program's input must stay valid until PeeR applies the result. For XDP, the packet buffer is owned by the driver's page pool and is valid only for the duration of the NAPI poll. The driver can recycle it locklessly only because the program runs to completion within that poll. PeeR must therefore extend the buffer's lifetime past the original call. It retains the packet as an independently owned `xdp_frame` (§5.1), so the driver does not recycle it before the continuation finishes.

Deferred result. The caller must accept the program's verdict after the original call returns rather than during it. PeeR returns `XDP_YIELD` in place of the program's verdict (§5.1); the driver treats it as a consumed packet and continues its NAPI poll, so the fast path does not wait. On resume, PeeR applies the program's real verdict, covering all native single-buffer actions: `XDP_DROP` and `XDP_ABORTED` free the frame, `XDP_PASS` builds an `skb` and injects it, `XDP_TX` transmits via `ndo_xdp_xmit`, and `XDP_REDIRECT` completes through the standard redirect path.

Caller-side invariants. The verifier reasons only about eBPF-visible state and cannot prove that surrounding kernel code holds no lock across the call, so we check this per hook. For XDP, the NAPI receive path invokes the program without holding a driver or stack lock that must be released by the time the verdict is produced, so deferring the verdict leaves no caller-side critical section open. The one piece of caller state a yield could disturb is the XDP redirect bookkeeping, kept in a per-task networking context (`bpf_net_context`). PeeR handles this by completing redirects from the PeeR-kthread's own networking context and flush list rather than the original poll's, similar to `cpumap`.

PeeR does not check a hook at runtime. It is built only for hooks that meet these conditions, and every other hook runs unchanged. Table 3 applies this audit to common hooks, as a design-time classification rather than a runtime test. As §2 explains, PeeR targets latency-critical I/O processing, not eBPF as a whole. These are the packet and storage fast paths such as

Hook	Compatible?	Why
Native XDP [22, 31]	✓	The driver runs the program on an `xdp_buff` and expects a packet verdict. PeeR retains the packet as an `xdp_frame` on yield and applies the verdict on resume, covering all five native actions. Multi-buffer and tail-call programs are rejected at load.
tc / TCX (direct-action) [16]	✓	The hook runs over an skb and returns a TC action. PeeR can apply terminal actions such as drop, stolen, and redirect after a yield, once it retains the skb and the redirect state. Continuing actions re-enter the classifier chain or the network stack, which lies outside the eBPF continuation, so PeeR supports only the terminal actions.
TC action chains (`act_bpf`) [15]	✗	An eBPF program is one action inside a larger action list. After it returns, the caller may run later actions, repeat, or jump to another chain. Reconstructing this requires the full action-list and chain state, which the eBPF continuation does not capture.
XRP storage (NVMe) [56]	✓	The hook runs in NVMe completion on a request-scoped context and either completes the read or resubmits the next I/O. PeeR can retain that request and resumption state across a yield and apply the action later. We do not implement this adapter, but the contract fits the model.
Tracing, LSM, sockets, block [13, 14, 17]	✗	These hooks fall outside PeeR's model for two reasons. PeeR cannot defer the result for tracing or LSM: tracing runs under locks held by the caller, and LSM must return its allow-or-deny decision before the caller continues. Socket and process-context hooks already run preemptably and do not require deferral.

Table 3: Hook compatibility with PeeR's correctness model. A hook is compatible only if the eBPF continuation and the retained hook object suffice to apply the final action without resuming caller-side kernel control flow. PeeR currently implements native XDP.

XDP, `tc`, and XRP. The hooks PeeR does not support, including tracing and LSM, are also not its target: their programs are short and event-driven, not the long request processing that causes head-of-line blocking, so they have little need for preemption. `softirq` packet hooks such as LWT and netfilter share the same skb-verdict structure, and applying PeeR to them is future work.

7 Scheduling eBPF Tasks

The mechanisms in §5 make eBPF programs preemptable, but preemptability alone does not determine *when* to preempt or *which* continuation to resume next. This section describes how PeeR integrates with `sched_ext` [49] to provide schedulability through operator-defined policies.

`sched_ext` is a Linux scheduler interface that allows users to implement arbitrary scheduling policies via a set of callbacks. The core abstraction is the *dispatch queue* (DSQ), into which tasks are enqueued and from which CPUs pull work. `sched_ext` determines scheduling order by controlling how tasks flow through DSQs, enabling policies such as priority scheduling or weighted fair sharing.

7.1 Two-Level Hierarchical Scheduling

Scheduling preempted eBPF programs presents a challenge: on the fast path, programs execute first-come-first-served with bounded budgets, matching existing eBPF semantics to minimize overhead. On the slow path, however, multiple preempted programs accumulate within a single PeeR-kthread, requiring a policy to determine resumption order. `sched_ext` cannot distinguish among them because they share the same PeeR-kthread, which is treated the same as all other kernel threads.

We address this with a two-level hierarchy. At the outer level, `sched_ext` acts as a *macro-scheduler*, making task-level decisions at millisecond granularity: which applications to run, how long they execute, and how CPU time is divided among workloads. Once `sched_ext` schedules a PeeR-kthread onto a CPU, the *micro-scheduler* takes over. The micro-scheduler runs within the PeeR-kthread context at microsecond granularity. It distinguishes among accumulated eBPF tasks and selects the subsequent continuation to resume. This division of responsibility allows `sched_ext` to control how much CPU time eBPF programs receive in aggregate, while the micro-scheduler determines which programs benefit from that time.

7.2 Accounting and Budget Enforcement

Table 2 shows the interface between PeeR and `sched_ext`. The macro-scheduler uses the top two entries to account for eBPF program execution time and enforce the execution budget:

Accounting execution time. PeeR exports a per-CPU `runtime_event` ring buffer. Each record contains the logical owner (e.g., a process or cgroup identifier), the CPU, the execution duration, and whether the task completed or yielded. Each eBPF task execution updates the `runtime_event` buffer. A userspace daemon consumes these events asynchronously and aggregates them into per-application CPU usage. This closes the feedback loop: `sched_ext` observes how much CPU time each eBPF application consumes and adjusts when and how often it schedules PeeR-kthreads accordingly. For example, if an application's eBPF programs have exceeded their fair share, `sched_ext` can deprioritize the PeeR-kthread or delay its next scheduling opportunity. This makes eBPF execution visible to system-wide scheduling policy.

Enforcing execution budgets. The macro-scheduler controls *when* eBPF programs yield through quantized budget enforcement. `set_budget()` configures the preemption quantum, a time interval that defaults to 5 μs in our evaluation. The scheduler advances a per-CPU epoch counter once per quantum. When an eBPF program begins execution, PeeR records the current epoch. At each helper call site, the JIT-inserted preemption check compares the current epoch against the recorded value, and if the epoch has advanced, the program yields.

7.3 Ordering Continuation Tasks

The micro-scheduler controls *which* eBPF task resumes next on a given PeeR-kthread. It uses a simplified version of `sched_ext`'s DSQ abstraction, with priority-based ordering to determine the next continuation task. Operators implement

```c
/* enqueuing new task: set the remaining time to an
 * application-provided runtime estimation.
 */
u64 enqueue(struct peer_cont *cont) {
    return cont->peer_metadata.estimated_work;
}
/* requeueing a task: remove the executed time from
 * the task remaining time.
 */
u64 requeue(struct peer_cont *cont, u64 runtime_ns) {
    /* underflow check omitted in this example */
    return cont->prio - runtime_ns;
}
```

Figure 7: Sample SRPT implementation. The scheduler prioritizes continuations closest to finishing by tracking remaining work.

scheduling policies via the bottom three rows of Table 2 by setting extra fields inside the continuation structure `cont` and two DSQ management callbacks.

The `enqueue()` and `requeue()` callbacks order tasks on the per-kthread DSQ. The first time an eBPF task is preempted, PeeR calls `enqueue()` to put this task on the DSQ to be scheduled. Each subsequent time the same task is preempted within the PeeR-kthread, `requeue()` is called with the time spent executing. These functions return a priority number that determines scheduling order: continuations with a lower priority number are resumed first. When priorities are equal, a sequence number provides FIFO ordering. Without application modification or a custom queueing policy, the micro-scheduler processes task continuations in FIFO order. Beyond that, the `cont` structure allows eBPF programs to provide scheduling hints (such as flow identifiers, weights, or deadlines) that the scheduler uses in its priority calculations.

Sample SRPT policy implementation. The priority-based interface supports a variety of scheduling policies. We demonstrate the Shortest Remaining Processing Time (SRPT) implementation as an example in Figure 7. SRPT prioritizes tasks with the least remaining work, minimizing average completion time. When a task is first preempted, `enqueue()` initializes its priority to an application-provided estimate of remaining work, stored in `cont`. When the same task is preempted again, `requeue()` subtracts the elapsed execution time from the current priority, reflecting progress toward completion. Since tasks with lower priority numbers are scheduled first, tasks closer to completion naturally rise to the front of the queue. Such an estimate is available in many of the workloads PeeR targets, since the program already parses each request to dispatch it and can derive a work hint from the request type, for example a scan's range or a transaction class.

8 Evaluation

This section evaluates PeeR's performance and expressiveness; we implement PeeR in Linux kernel 6.16 with ≈ 5000 lines of C code. We first show how PeeR's preemption mechanism eliminates head-of-line blocking within an application, using a mixed GET/SCAN workload and a TPC-C workload, and use TPC-C to demonstrate expressiveness across multiple scheduling policies (§8.1). We then evaluate inter-application allocation by colocating batch and latency-sensitive jobs under CPU-sharing policies (§8.2), and finally present microbenchmarks of preemption and resumption overhead (§8.3).

System setup. We evaluate PeeR on two servers (application and client), each with a 28-core Intel Xeon Gold 5420+ and 256 GB RAM, connected via Mellanox ConnectX-7 400 Gbps NICs through an Intel Tofino 2 switch. Both run Ubuntu 24.04 with Linux 6.16 modified for PeeR, with hyperthreading, adaptive interrupt coalescing, and IOMMU disabled for low latency.

Our scheduler extends `scx_layered` [30], Meta's production `sched_ext` scheduler, with PeeR's accounting and budget enforcement. We generate requests using `loadgen`, an open-loop generator with Poisson arrivals over UDP and TCP. We use NIC flow steering to evenly distribute load across cores. For Redis-KFlex, we use a lightweight in-eBPF TCP stack that executes in the XDP hook. We set PeeR's preemption budget to 5 μs by default unless otherwise noted.

8.1 Intra-Application Scheduling

PeeR enables preemption within an application and supports operator-defined policies. We first evaluate the preemption mechanism on a bimodal workload, then assess expressiveness across policies on TPC-C [51].

8.1.1 Preemption for Latency-Critical Tasks

We evaluate bimodal Redis and Memcached workloads mixing latency-sensitive GET requests with long-running SCAN queries, similar to the setup in Figure 1. For this workload, we use simple FIFO ordering with preemption enabled; no additional scheduling policy is required to demonstrate the benefits. For Redis, we compare two multi-core key-value stores: the userspace baseline KeyDB [37], a multithreaded fork of Redis (we use it rather than single-threaded stock Redis), and Redis-KFlex, a Redis-compatible store in KFlex [9] that runs entirely in the XDP hook. We also add a cpumap-deferral baseline that tests whether the application can address head-of-line blocking itself, without kernel changes. The workload is 99.5% GET (0.2 μs) and 0.5% SCAN (200 μs). For Memcached, we compare the standard userspace server, BMC [28] (a common pattern where the fast path runs in-kernel but cache misses fall back to userspace), and KFlex (entirely in eBPF), under a 50/50 GET (0.2 μs)/SCAN (30 μs) mix. For both, we measure in-kernel performance with and without PeeR to isolate preemption's impact.

Figure 8 plots achieved throughput against tail latency, with Redis (left) and Memcached (right) columns; top plots show GET p99 latency versus throughput (krps) and bottom plots show the same for SCAN.

For Redis-KFlex (left column), SCAN queries block GETs and inflate tail latency under default eBPF. Enabling PeeR reduces GET p99 latency by 19.8× over default eBPF while achieving 3.77× higher throughput than userspace KeyDB. The bottom plots confirm SCANs are not starved: below 1000 krps, PeeR even improves SCAN latency by reducing blocking among SCANs themselves. PeeR also achieves 10.3% higher throughput than default eBPF, because preempting long

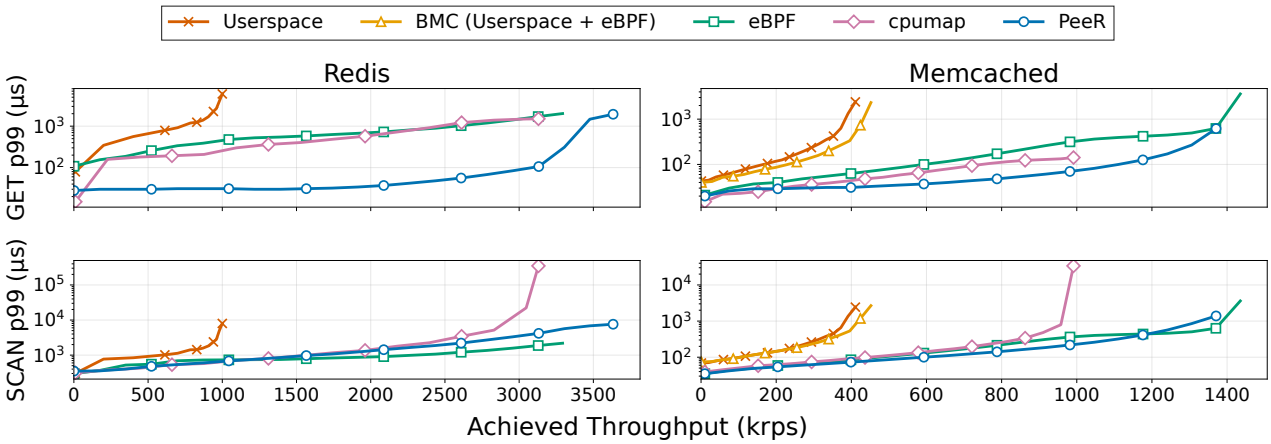


Figure 8: Point/scan workloads for Redis (99.5% GET, 0.5% SCAN) and Memcached (50% GET, 50% SCAN). PeeR retains the performance benefit of in-kernel execution over userspace, while reducing GET p99 latency by preventing long-running scan operations from blocking point queries compared to the default eBPF runtime. Userspace baselines: KeyDB and stock Memcached. The x-axis is total achieved throughput across both query types; curves end at saturation.

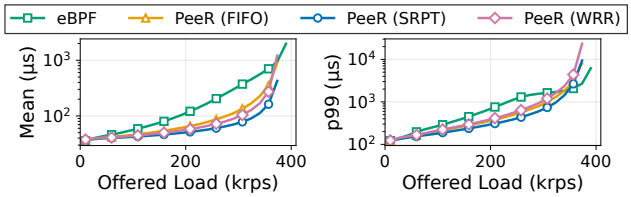


Figure 9: Aggregate TPC-C latency under different scheduling policies. FIFO with preemption outperforms baseline eBPF; SRPT achieves 3× lower mean latency by prioritizing short requests.

SCANs keeps the receive queue drained, whereas under default eBPF the queue fills behind blocked GETs and drops requests.

The application can also chunk long requests itself. The cpumap baseline bounds the work per XDP invocation and defers the rest of a long SCAN to a per-CPU kernel thread without kernel changes, but this only relocates the blocking: a packet batch drains fully before the next, deferred work reaches the thread only at batch boundaries, and the thread runs those SCANs with softirqs disabled, so later GETs wait behind them. At 1.5 Mrps cpumap brings GET p99 to 403 μs (against 581 μs for default eBPF and 31 μs for PeeR), is the first to saturate, at 3.1 Mrps, and near saturation its SCAN p99 climbs to 350 ms against 4 ms for PeeR.

For Memcached (right column), both KFlex configurations (with and without PeeR) achieve 3.46× higher throughput than userspace; BMC matches userspace because cache misses fall back to it. PeeR reduces GET p99 latency by 4.5× over default eBPF. Unlike Redis-KFlex, PeeR's overall throughput is slightly lower than default eBPF: with a 50/50 mix and shorter queries, queue buildup is milder, so preemption overhead is not offset by reduced drops. Still, PeeR isolates GET latency regardless of SCAN load, while the cpumap baseline tracks default eBPF and saturates earlier. At peak, Redis's 3.6 Mrps is 9 Mpps and 12 Gbps, and Memcached's 1.5 Mrps is 3.0 Mpps and 3.1 Gbps.

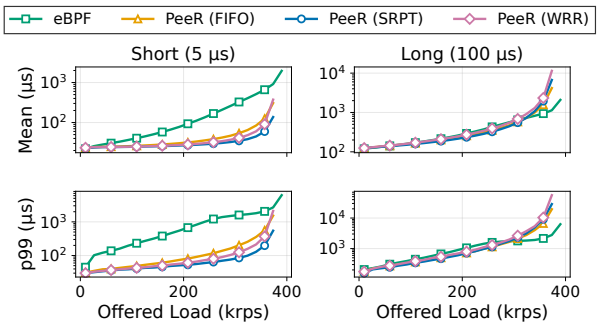


Figure 10: TPC-C latency by request type. Short queries (Payment) benefit from preemption across all loads; long queries (StockLevel) are not impacted until load exceeds 300 krps.

8.1.2 Expressing Scheduling Policies

To demonstrate expressiveness, we run a TPC-C workload and implement multiple scheduling policies in PeeR. We use a synthetic XDP echo server to emulate the compute workload, where each UDP request specifies a spin duration drawn from TPC-C transaction profiles: Payment (5.7 μs, 44%), OrderStatus (6 μs, 4%), NewOrder (20 μs, 44%), Delivery (88 μs, 4%), and StockLevel (100 μs, 4%), taken from prior in-memory database measurements [8]. We implement three policies in PeeR's `sched_ext` component: FIFO, SRPT (shortest-remaining-processing-time), and WRR (weighted round-robin).

Figure 9 compares baseline eBPF (no preemption) against PeeR under each policy, with load (krps) on the x-axis and mean (left) and p99 (right) latency on the y-axis. Even the simplest FIFO scheduler with PeeR outperforms baseline eBPF, but PeeR's expressiveness enables more effective policies: SRPT achieves the best mean and tail latency by prioritizing short requests, cutting mean latency 3× over baseline at 300 krps, while WRR is a practical alternative

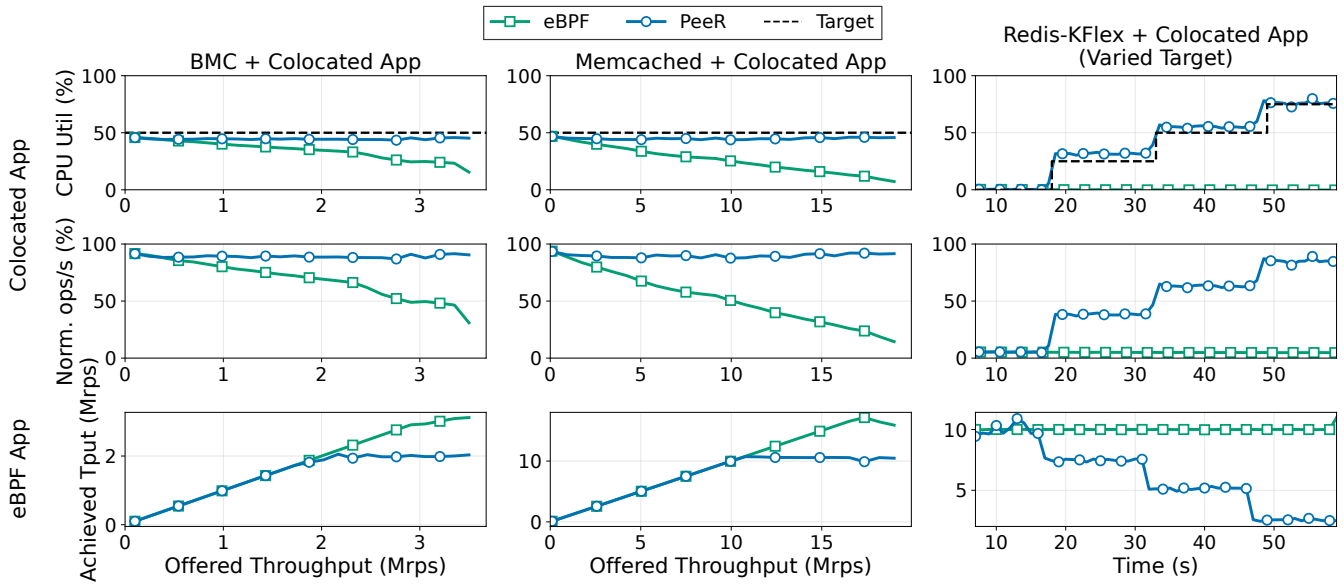


Figure 11: Colocation experiments showing latency-sensitive applications (Memcached, Redis) colocated with batch jobs. PeeR enforces operator-defined CPU allocation among workloads.

when execution time is unknown a priori.

Figure 10 breaks down latency by request type. Short queries (Payment) benefit from preemption across the entire load range under all three policies, while long queries (StockLevel) match baseline until load exceeds 300 krps, so PeeR improves short-request latency without starving long ones. These gains come at a deep-tail cost for long requests, which are delayed when preempted and resumed on a worker. For the Redis point/scan workload, SCAN p99.9 is comparable to the non-preemptive baseline at low load (946 μs vs. 924 μs at 1 Mrps) and rises to 2.3× near saturation (5952 μs vs. 2562 μs at 3.1 Mrps); in TPC-C, StockLevel has 1.5–1.7× higher p99.9 under PeeR, and SRPT concentrates this cost on the single longest class while improving the next-longest (Delivery p99.9 of 1145 μs under SRPT vs. 2042 μs for the non-preemptive baseline). An operator who must protect long requests can raise the budget or give them higher micro-scheduler priority; PeeR's flexibility to match policies to workloads is itself expressiveness beyond today's eBPF.

8.2 Inter-Application Scheduling

We evaluate PeeR's ability to enforce scheduling policies when eBPF applications compete with other userspace workloads for CPU time. We colocate a latency-sensitive eBPF application (BMC, Memcached, and Redis-KFlex serving GET queries) alongside a compute-bound batch job, similar to the setup in Figure 2. A successful scheduler must enforce the configured CPU allocation while preserving throughput and latency for both workloads.

Figure 11 shows the results; each column is an eBPF application, with offered load (left, middle) or time (right) on the x-axis. Rows show the batch job's CPU utilization vs. its target, its normalized throughput, and the eBPF application's achieved throughput. With vanilla eBPF, the batch job starves as load rises: for BMC under a 50/50 fair-share policy, BMC exceeds its allocation and the batch job's throughput drops to 30.9% of its fair share, because eBPF execution is invisible to the scheduler. PeeR fixes this by making eBPF execution time visible and accountable, capping BMC's utilization at its share so the batch job keeps its throughput; the middle column shows the same for Memcached, where the batch job's CPU collapses to 7% without PeeR but holds its fair share with it.

Beyond equal sharing, PeeR enforces weighted allocations. The right column varies Redis-KFlex's batch-job target from 0% to 75% over time: PeeR tracks it within 6% across all configurations, whereas without PeeR the batch job's utilization bears no relation to its target, receiving only what Redis-KFlex leaves behind. PeeR thus makes eBPF workloads first-class scheduling citizens, subject to the same policies as userspace processes.

8.3 Microbenchmarks

PeeR's performance depends on the low overhead enabled by the cooperative preemption mechanism, while the preemption granularity depends on the budget we set. This section studies PeeR's instrumentation overhead, its preemption and resumption cost, and the impact of different preemption budgets.

8.3.1 Overhead Breakdown

We measure preemption and resumption overhead by running an XDP echo server under high load, preempting once per packet. Figure 12 shows the time spent in each phase, averaged over more than one million measurements.

Preemption costs 77 ns: reading the `sched_ext`-set flag incurs an L1 miss (17 ns), and checking it, transferring to the yield handler, and saving state adds 60 ns. Resumption costs

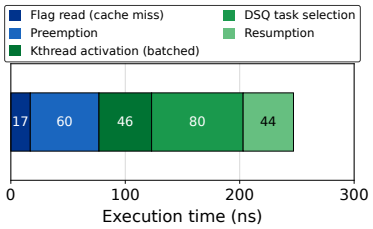


Figure 12: PeeR overhead breakdown for a preemption-resumption cycle. The total 247 ns cost enables fine-grained scheduling while remaining practical for microsecond-scale workloads.

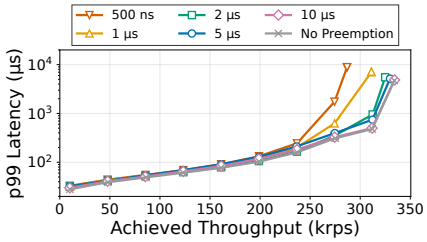


Figure 13: Throughput vs. p99 latency for an XDP echo server under varying preemption budgets. Smaller budgets reduce peak throughput; a budget of 10 μs matches the no-preemption baseline.

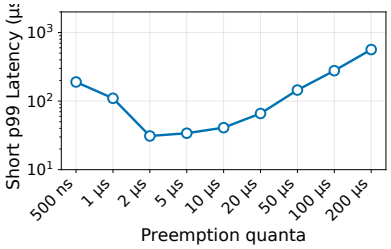


Figure 14: Preemption budget tradeoff for a bimodal workload. Small budgets incur context-switch overhead for the latency-critical short task, while large budgets lead to head-of-line blocking.

170 ns: activating a PeeR-kthread takes 46 ns (amortized across batched wakeups), selecting the next task from the dispatch queue 80 ns, and restoring context 44 ns. A full preemption-resumption cycle thus costs 247 ns, making fine-grained scheduling practical even for microsecond-scale eBPF tasks.

8.3.2 Instrumentation Overhead

The budget check the JIT inserts before each helper call is PeeR's only always-on cost, paid even by a program that never preempts. We measure it with an XDP program that calls `bpf_get_prandom_u32` in a loop, comparing the stock JIT against PeeR's JIT with a budget set so the check executes on every call but never fires. The check adds one cycle per helper call, and two when the scheduler is actively writing the per-CPU budget flag.

8.3.3 Impact of Preemption Budget

While the overhead per preemption-resumption cycle is constant, the overall overhead depends on the preemption frequency, which is controlled by the budget. We evaluate the application-level overhead of PeeR.

Throughput impact. Figure 13 shows how the budget affects throughput for an XDP echo server processing 5 μs requests under uniform load. Smaller budgets check more often, consuming cycles that would process requests: at 500 ns the server saturates at 275 krps, while at 10 μs requests complete without preemption, matching the no-preemption baseline at 330 krps. A 2–5 μs budget sacrifices some throughput for scheduling responsiveness under more complex workloads, which we quantify next.

Latency impact. Figure 14 isolates the budget's effect on tail latency using a bimodal workload: 95% short requests (0.5 μs) and 5% long requests (100 μs) at fixed 20 krps. We sweep budgets from 500 ns to 200 μs and measure p99 latency for short requests. The results show a U-shaped curve. At small budgets (500 ns–1 μs), frequent preemption dominates, inflating latency to 110–190 μs. As the budget increases, overhead drops and latency reaches a minimum of 31 μs at a 2 μs budget. Beyond this, head-of-line blocking takes effect: long requests yield more slowly, thereby delaying short ones. At a 200 μs preemption budget, p99 of the short requests rises to 565 μs.

The U-shape captures a fundamental tradeoff: smaller budgets provide finer scheduling granularity but increase context-switch overhead; larger budgets reduce overhead but allow longer blocking. The optimal budget is workload dependent.

9 Discussion

Existing mechanisms for deferred eBPF execution. Linux provides several mechanisms to defer eBPF work off the fast path, such as eBPF timers [20], workqueues [21], and XDP's `cpumap` redirect [50]. eBPF timers schedule callbacks for later, but they still run in `softirq`, steal time from interrupted applications, and lack hook context or I/O capabilities. Workqueues defer to a sleepable kernel worker, but each callback executes as a single eBPF invocation; the scheduler sees only the worker thread, not individual eBPF tasks. `cpumap` queues packets to a per-CPU kernel thread, but the XDP program still runs to completion per packet and can head-of-line block.

PeeR generalizes this pattern of executing minimal work in an interrupt context, deferring the rest, but with finer-grained control. No existing mechanism combines per-invocation preemption, scheduler visibility, and policy control. None exposes fast-path eBPF execution time to schedulers or supports policy-driven decisions over when and how long each invocation runs. PeeR addresses this gap by treating preemption as a first-class primitive, rather than requiring developers to split programs or accept run-to-completion semantics.

Kernel bypass. Kernel-bypass systems achieve high throughput and low tail latency by removing the kernel from the critical path, but require monopolizing cores and hardware. eBPF offers similar performance while retaining OS-provided isolation, safety, and unified resource management.

Kernel-bypass systems usually assume that scheduling flexibility requires moving to userspace. Systems such as Shinjuku [35] and Shenango [42] implement preemption and core reallocation in userspace because the kernel provides no equivalent. Our work challenges this assumption: eBPF combined with `sched_ext` supports similar policies without sacrificing kernel interposition or dedicating cores to userspace runtimes. Kernel-bypass retains advantages in hardware-specific optimizations, custom allocators, and direct device queue

access, but expanding eBPF capabilities may narrow this gap.

Enabling new eBPF applications. Beyond improving existing workloads, PeeR opens possibilities for applications previously unsuitable for eBPF. File-serving, for example, requires sleepable contexts: programs must yield while waiting for disk I/O rather than blocking the CPU. eBPF's current run-to-completion model cannot express this. PeeR's yield mechanism allows programs to suspend mid-execution during I/O and resume once data is ready, bringing I/O-bound workloads within reach of in-kernel execution.

10 Related Work

PeeR builds on a large body of prior work.

eBPF for extensibility and latency-critical applications. eBPF has been widely adopted for extending kernel functionality safely and efficiently, with applications spanning memory management [55], file systems [5], page caches [60, 61], and networking [6]. Beyond traditional extensibility, eBPF has also been adopted for low-latency, high-performance applications. XDP [31] and BMC [28] leverage eBPF for high-performance packet processing and caching. KFlex [9], DINT [58], and eTran [6] implement full application logic in XDP to bypass the kernel network stack entirely. XRP [56] and Electrode [57] use eBPF for storage and for accelerating distributed systems. However, these systems inherit the non-preemptable execution model of fast-path eBPF hooks, which increases tail latency when operations vary in duration, an issue that DINT and eTran have explicitly noted. PeeR addresses this by enabling cooperative preemption, allowing eBPF programs to handle longer operations without blocking other tasks.

Preemptive scheduling for low latency. A complementary line of work reduces preemption overhead itself to achieve low latency for applications. Shinjuku [35] achieves 5 μs preemption quanta using hardware virtualization, but requires a specialized OS and dedicated dispatcher cores. ZygOS [43] adds work stealing to IX [3]'s run-to-completion dataplane, and Arachne [44] manages user-level threads to rebalance load at microsecond granularity. Shenango [42] achieves similar responsiveness through fast core allocation rather than fine-grained preemption. Compiler-assisted preemption is used more broadly: Wasmtime [4] inserts epoch checks, Go [7] preempts via signals at compiler-identified safe points, and Concord [34], Compiler Interrupts [2], and TinyQuanta [40] approximate preemption with compiler instrumentation that polls a shared cache line or counts instructions. PeeR brings this technique to the eBPF runtime. While Concord is similar in spirit, the eBPF setting imposes different constraints, since yield sites must align with verifier state, hook-object lifetimes, and kernel CFI requirements. User interrupts [29, 38] provide low-overhead userspace preemption but do not extend to kernel code. Recent work also proposes *hybrid* cooperative-preemptive designs that fall back to timer-based preemption when cooperative yields are too sparse [52]. PeeR instead stays purely cooperative: on the eBPF fast path a timer could fire between helper calls, where the verifier leaves no clean state to checkpoint, so PeeR confines yields to helper boundaries and relies on the bounded inter-helper gap (Table 1) to keep them frequent. These systems target userspace applications; PeeR brings similar benefits to unmodified kernel extensions without specialized hardware.

Inter-application performance isolation. Performance isolation is critical in multi-tenant environments. Junction [24] and Caladan [25] isolate application performance through smart resource multiplexing. Static analysis techniques [27] verify safety and performance properties of kernel extensions. These approaches do not address long-running eBPF operations that escape scheduler visibility. PeeR enhances isolation by making eBPF execution time visible to the scheduler and allowing programs to yield, preventing resource monopolization.

User-defined scheduling. Frameworks such as ghOSt [32] and Syrup [36] allow defining custom scheduling policies for threads and network packets in userspace. FlexNet [10] proposes coroutine-based structural extensibility for the network stack, with yield points at protocol-layer boundaries. However, these systems schedule userspace threads and packets, not in-kernel eBPF execution. The `sched_ext` framework [49] enables implementing custom CPU schedulers in eBPF, but it does not schedule eBPF programs themselves. PeeR complements these frameworks by treating existing eBPF programs as schedulable entities that can be preempted and rescheduled without modification.

Extending eBPF to enable new capabilities. Recent work extends eBPF's capabilities. eNetSTL [53] provides STL-like data structures for in-kernel network functions; eGPU [54] executes eBPF on GPUs; and Zussman et al. [59] implement new eBPF hooks to enable custom page-fault handling. These extensions increase both the complexity and execution time of eBPF programs. PeeR complements this trend by providing the scheduling and preemption infrastructure needed as eBPF workloads grow beyond the fast-path assumption. SchedBPF [46] also explores preemptable and schedulable eBPF, with a focus on periodically executed programs and continuous workloads. In contrast, PeeR targets event-driven eBPF hooks in `softirq` context on the critical path. We characterize problems quantitatively in these settings and provide a concrete implementation of preemption and scheduling mechanisms.

11 Conclusion

We presented PeeR, a runtime that makes latency-critical eBPF programs preemptable and schedulable in `softirq` context, where preemption is otherwise disabled, by bringing cooperative preemption to eBPF and integrating with `sched_ext`. Our evaluation shows that PeeR reduces p99 latency by 3× to 19.8× for latency-critical requests in mixed workloads, and enforces resource-sharing policies across colocated eBPF and userspace applications.

Acknowledgments

We would like to thank our anonymous shepherd and OSDI reviewers for their thoughtful feedback. We are grateful to Kumar Kartikeya Dwivedi for his help with the KFlex experiments and his feedback on the paper. We also thank Anton Zabreyko, Gohar Irfan Chaudhry, Om Chabra, Benny Rubin, and the members of the NMS and PDOS groups at MIT for many helpful discussions. The MIT-affiliated authors are supported by NSF grants CAREER-2144766, PPoSS-2217099, CNS-2211382, and CNS-2212099.

References

[1] The Cilium Authors. Cilium: ebpf-based networking, security, and observability. https://cilium.io/.

[2] Nilanjana Basu, Claudio Montanari, and Jakob Eriksson. Frequent background polling on a shared thread, using light-weight compiler interrupts. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, page 1249–1263, New York, NY, USA, 2021. Association for Computing Machinery.

[3] Adam Belay, George Prekas, Mia Primorac, Ana Klimovic, Samuel Grossman, Christos Kozyrakis, and Edouard Bugnion. The ix operating system: Combining low latency, high throughput, and efficiency in a protected dataplane. *ACM Trans. Comput. Syst.*, 34(4), December 2016.

[4] Bytecode Alliance. Wasmtime: Epoch-based interruption. https://docs.wasmtime.dev/api/wasmtime/struct.Config.html#method.epoch_interruption.

[5] Xuechun Cao, Shaurya Patel, Soo Yee Lim, Xueyuan Han, and Thomas Pasquier. FetchBPF: Customizable prefetching policies in linux with eBPF. In *2024 USENIX Annual Technical Conference (USENIX ATC 24)*, pages 369–378, Santa Clara, CA, July 2024. USENIX Association.

[6] Zhongjie Chen, Qingkai Meng, ChonLam Lao, Yifan Liu, Fengyuan Ren, Minlan Yu, and Yang Zhou. eTran: Extensible kernel transport with eBPF. In *22nd USENIX Symposium on Networked Systems Design and Implementation (NSDI 25)*, pages 407–425, Philadelphia, PA, April 2025. USENIX Association.

[7] Austin Clements. Proposal: Non-cooperative goroutine preemption. https://go.googlesource.com/proposal/+/master/design/24543-non-cooperative-preemption.md, 2019. Go design document; signal-based asynchronous preemption shipped in Go 1.14.

[8] Henri Maxime Demoulin, Joshua Fried, Isaac Pedisich, Marios Kogias, Boon Thau Loo, Linh Thi Xuan Phan, and Irene Zhang. When idling is ideal: Optimizing tail-latency for heavy-tailed datacenter workloads with perséphone. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, SOSP '21, page 621–637, New York, NY, USA, 2021. Association for Computing Machinery.

[9] Kumar Kartikeya Dwivedi, Rishabh Iyer, and Sanidhya Kashyap. Fast, flexible, and practical kernel extensions. In *Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles*, SOSP '24, pages 249–264, New York, NY, USA, 2024. Association for Computing Machinery.

[10] Kumar Kartikeya Dwivedi, Rishabh Iyer, and Sanidhya Kashyap. Towards structurally extensible host network stacks. In *Proceedings of the 24th ACM Workshop on Hot Topics in Networks*, HotNets '25, page 263–270, New York, NY, USA, 2025. Association for Computing Machinery.

[11] Helper functions. https://docs.ebpf.io/linux/helper-function/.

[12] BPF Instruction Set Architecture (ISA). https://www.kernel.org/doc/html/v6.16/bpf/standardization/instruction-set.html.

[13] eBPF program type BPF_PROG_TYPE_KPROBE. https://docs.ebpf.io/linux/program-type/BPF_PROG_TYPE_KPROBE/.

[14] eBPF program type BPF_PROG_TYPE_LSM. https://docs.ebpf.io/linux/program-type/BPF_PROG_TYPE_LSM/.

[15] eBPF program type BPF_PROG_TYPE_SCHED_ACT. https://docs.ebpf.io/linux/program-type/BPF_PROG_TYPE_SCHED_ACT/.

[16] eBPF program type BPF_PROG_TYPE_SCHED_CLS (tc/TCX). https://docs.ebpf.io/linux/program-type/BPF_PROG_TYPE_SCHED_CLS/.

[17] eBPF program type BPF_PROG_TYPE_SK_SKB. https://docs.ebpf.io/linux/program-type/BPF_PROG_TYPE_SK_SKB/.

[18] ebpf tail calls. https://docs.ebpf.io/linux/bpf-tail-calls/.

[19] eBPF Timeline. https://docs.ebpf.io/linux/timeline/.

[20] eBPF Timers. https://docs.ebpf.io/linux/concepts/timers/, 2023.

[21] KFunc 'bpf_wq_init'. https://docs.ebpf.io/linux/kfuncs/bpf_wq_init/, 2024.

[22] eBPF program type BPF_PROG_TYPE_XDP. https://docs.ebpf.io/linux/program-type/BPF_PROG_TYPE_XDP/.

[23] Arthur Fabre. L4drop: Xdp ddos mitigations, November 2018.

[24] Joshua Fried, Gohar Irfan Chaudhry, Enrique Saurez, Esha Choukse, Inigo Goiri, Sameh Elnikety, Rodrigo Fonseca, and Adam Belay. Making kernel bypass practical for the cloud with junction. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*, pages 55–73, Santa Clara, CA, April 2024. USENIX Association.

[25] Joshua Fried, Zhenyuan Ruan, Amy Ousterhout, and Adam Belay. Caladan: Mitigating interference at microsecond timescales. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 281–297. USENIX Association, November 2020.

[26] Bolaji Gbadamosi, Luigi Leonardi, Tobias Pulls, Toke Høiland-Jørgensen, Simone Ferlin-Reiter, Simo Sorce, and Anna Brunström. The eBPF Runtime in the Linux Kernel, 2024.

[27] Elazar Gershuni, Nadav Amit, Arie Gurfinkel, Nina Narodytska, Jorge A. Navas, Noam Rinetzky, Leonid Ryzhyk, and Mooly Sagiv. Simple and precise static analysis of untrusted linux kernel extensions. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2019, page 1069–1084, New York, NY, USA, 2019. Association for Computing Machinery.

[28] Yoann Ghigoff, Julien Sopena, Kahina Lazri, Antoine Blin, and Gilles Muller. BMC: Accelerating memcached using safe in-kernel caching and pre-stack processing. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*, pages 487–501. USENIX Association, April 2021.

[29] Linsong Guo, Danial Zuberi, Tal Garfinkel, and Amy Ousterhout. The benefits and limitations of user interrupts for preemptive userspace scheduling. In *Proceedings of the 22nd USENIX Symposium on Networked Systems Design and Implementation*, NSDI '25, USA, 2025. USENIX Association.

[30] Tejun Heo. scx_layered: A case study of using sched_ext for latency sensitive workloads at Meta, November 2023.

[31] Toke Høiland-Jørgensen, Jesper Dangaard Brouer, Daniel Borkmann, John Fastabend, Tom Herbert, David Ahern, and David Miller. The express data path: fast programmable packet processing in the operating system kernel. In *Proceedings of the 14th International Conference on Emerging Networking EXperiments and Technologies*, CoNEXT '18, pages 54–66, New York, NY, USA, 2018. Association for Computing Machinery.

[32] Jack Tigar Humphries, Neel Natu, Ashwin Chaugule, Ofir Weisse, Barret Rhoden, Josh Don, Luigi Rizzo, Oleg Rombakh, Paul Turner, and Christos Kozyrakis. ghost: Fast & flexible user-space delegation of linux scheduling. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, SOSP '21, page 588–604, New York, NY, USA, 2021. Association for Computing Machinery.

[33] Facebook Incubator. Katran: A high performance layer 4 load balancer. https://github.com/facebookincubator/katran.

[34] Rishabh Iyer, Musa Unal, Marios Kogias, and George Candea. Achieving microsecond-scale tail latency efficiently with approximate optimal scheduling. In *Proceedings of the 29th Symposium on Operating Systems Principles*, SOSP '23, page 466–481, New York, NY, USA, 2023. Association for Computing Machinery.

[35] Kostis Kaffes, Timothy Chong, Jack Tigar Humphries, Adam Belay, David Mazières, and Christos Kozyrakis. Shinjuku: Preemptive scheduling for μsecond-scale tail latency. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 345–360, Boston, MA, February 2019. USENIX Association.

[36] Kostis Kaffes, Jack Tigar Humphries, David Mazières, and Christos Kozyrakis. Syrup: User-defined scheduling across the stack. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, SOSP '21, page 605–620, New York, NY, USA, 2021. Association for Computing Machinery.

[37] Keydb: A multithreaded fork of redis. https://github.com/Snapchat/KeyDB.

[38] Yueying Li, Nikita Lazarev, David Koufaty, Tenny Yin, Andy Anderson, Zhiru Zhang, G Edward Suh, Kostis Kaffes, and Christina Delimitrou. Libpreemptible: Enabling fast, adaptive, and hardware-assisted user-space scheduling. In *2024 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 922–936. IEEE, 2024.

[39] Chang Liu, Byungchul Tak, and Long Wang. Understanding performance of ebpf maps. In *Proceedings of the ACM SIGCOMM 2024 Workshop on EBPF and Kernel Extensions*, eBPF '24, pages 9–15, New York, NY, USA, 2024. Association for Computing Machinery.

[40] Zhihong Luo, Sam Son, Dev Bali, Emmanuel Amaro, Amy Ousterhout, Sylvia Ratnasamy, and Scott Shenker. Efficient microsecond-scale blind scheduling with tiny quanta. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, pages 305–319, 2024.

[41] Paul E. McKenney. Sleepable RCU. LWN.net, October 2006. Accessed: 2025-12-11.

[42] Amy Ousterhout, Joshua Fried, Jonathan Behrens, Adam Belay, and Hari Balakrishnan. Shenango: Achieving high CPU efficiency for latency-sensitive datacenter workloads. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 361–378, Boston, MA, February 2019. USENIX Association.

[43] George Prekas, Marios Kogias, and Edouard Bugnion. Zygos: Achieving low tail latency for microsecond-scale networked tasks. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, pages 325–341, New York, NY, USA, 2017. Association for Computing Machinery.

[44] Henry Qin, Qian Li, Jacqueline Speiser, Peter Kraft, and John Ousterhout. Arachne: Core-Aware Thread Management. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 145–160, Carlsbad, CA, October 2018. USENIX Association.

[45] Valentin Rothberg. *Interrupt handling in Linux*. Friedrich-Alexander-Universität Erlangen-Nürnberg (FAU), 2015.

[46] Kavya Shekar and Dan Williams. Schedbpf - scheduling bpf programs. In *Proceedings of the 3rd Workshop on EBPF and Kernel Extensions*, eBPF '25, page 45–47, New York, NY, USA, 2025. Association for Computing Machinery.

[47] The Linux Foundation. DPDK: Data Plane Development Kit. https://www.dpdk.org, 2025. Version 25.03 released March 24, 2025.

[48] The Linux Foundation. SPDK: Storage Performance Development Kit. https://www.spdk.io, 2025.

[49] Extensible Scheduler Class. https://docs.kernel.org/scheduler/sched-ext.html.

[50] The Linux Kernel Documentation Project. BPF_MAP_TYPE_CPUMAP. docs.kernel.org, 2025. Accessed on June 10, 2026.

[51] Transaction Processing Performance Council (TPC). TPC Benchmark C (TPC-C) Standard Specification, Revision 5.11.0. http://www.tpc.org/tpc_documents_current_versions/pdf/tpc-c_v5.11.0.pdf, 2010. [Accessed 2025-12-11].

[52] Yizheng Xie, Di Jin, and Nikos Vasilakis. Towards hybrid cooperative-preemptive scheduling. In *Proceedings of the 13th Workshop on Programming Languages and Operating Systems*, PLOS '25, New York, NY, USA, 2025. Association for Computing Machinery.

[53] Bin Yang, Dian Shen, Junxue Zhang, Hanlin Yang, Lunqi Zhao, Beilun Wang, Guyue Liu, and Kai Chen. enetstl: Towards an in-kernel library for high-performance ebpf-based network functions. In *Proceedings of the Twentieth European Conference on Computer Systems*, EuroSys '25, page 42–58, New York, NY, USA, 2025. Association for Computing Machinery.

[54] Bin Yang, Dian Shen, Junxue Zhang, Hanlin Yang, Lunqi Zhao, Beilun Wang, Guyue Liu, and Kai Chen. enetstl: Towards an in-kernel library for high-performance ebpf-based network functions. In *Proceedings of the Twentieth European Conference on Computer Systems*, EuroSys '25, page 42–58, New York, NY, USA, 2025. Association for Computing Machinery.

[55] Anil Yelam, Kan Wu, Zhiyuan Guo, Suli Yang, Rajath Shashidhara, Wei Xu, Stanko Novaković, Alex C Snoeren, and Kimberly Keeton. {PageFlex}: Flexible and efficient user-space delegation of linux paging policies with {eBPF}. In *2025 USENIX Annual Technical Conference (USENIX ATC 25)*, pages 291–306, 2025.

[56] Yuhong Zhong, Haoyu Li, Yu Jian Wu, Ioannis Zarkadas, Jeffrey Tao, Evan Mesterhazy, Michael Makris, Junfeng Yang, Amy Tai, Ryan Stutsman, and Asaf Cidon. XRP: In-Kernel storage functions with eBPF. In *Symposium on Operating Systems Design and Implementation (OSDI)*, pages 375–393, Carlsbad, CA, July 2022. USENIX Association.

[57] Yang Zhou, Zezhou Wang, Sowmya Dharanipragada, and Minlan Yu. Electrode: Accelerating distributed protocols with eBPF. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 1391–1407, Boston, MA, April 2023. USENIX Association.

[58] Yang Zhou, Xingyu Xiang, Matthew Kiley, Sowmya Dharanipragada, and Minlan Yu. DINT: Fast In-Kernel distributed transactions with eBPF. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*, pages 401–417, Santa Clara, CA, April 2024. USENIX Association.

[59] Tal Zussman, Teng Jiang, and Asaf Cidon. Custom page fault handling with ebpf. In *Proceedings of the ACM SIGCOMM 2024 Workshop on EBPF and Kernel Extensions*, eBPF '24, page 71–73, New York, NY, USA, 2024. Association for Computing Machinery.

[60] Tal Zussman, Ioannis Zarkadas, Jeremy Carin, Andrew Cheng, Hubertus Franke, Jonas Pfefferle, and Asaf Cidon. Cache is King: Smart Page Eviction with eBPF, 2025.

[61] Tal Zussman, Ioannis Zarkadas, Jeremy Carin, Andrew Cheng, Hubertus Franke, Jonas Pfefferle, and Asaf Cidon. cache_ext: Customizing the Page Cache with eBPF. In

Proceedings of the ACM SIGOPS 31st Symposium on Operating Systems Principles, SOSP '25, pages 462–478, New York, NY, USA, 2025. Association for Computing Machinery.

ARTIFACT EVALUATED
USENIX
AVAILABLE

TypeCraft: A Lightweight Data Type Profiler with High Resolution

Zecheng Li*
North Carolina State University

Xu Liu
Google

Namhyung Kim
Google

Blake Jones
Google

Alexey Alexandrov
Google

Jiajia Li
North Carolina State University

Abstract

Improving software efficiency often involves optimizing data locality to reduce memory stalls. However, identifying such optimization opportunities, particularly in complex production software like the Linux kernel, is challenging. Existing profiling tools typically provide metrics such as cache and TLB misses for instructions, loops, functions, or heap allocations, still requiring substantial manual efforts to identify optimization opportunities. To overcome this, we introduce TypeCraft, a lightweight, high-resolution data type profiler, integrated into the Linux `perf` tool, that annotates individual memory access instructions with their associated data types and fields. TypeCraft provides detailed type-centric telemetry such as access counts, CPU cycle costs, cache or TLB misses, which helps identify optimization opportunities around the expensive types. Applying TypeCraft to the Linux kernel, we gain insights that guide us in implementing simple yet effective optimizations. These optimizations, including reordering structure fields and removing pointer chasing patterns, result in significant performance improvements for both benchmarks and real-world workloads.

1 Introduction

Memory access latency has consistently posed a critical performance bottleneck in contemporary data centers. Past research conducted within Google's data centers indicates that a substantial 40-60% of CPU cycles are consumed during data retrieval from memory [37]. This bottleneck has intensified in recent years [57]. To mitigate the lengthy latencies associated with memory accesses, modern CPU processors utilize a cache system for storing frequently accessed data. A common optimization strategy is to improve data locality, which involves staging data into caches and thoroughly accessing it before eviction [24, 31], thereby reducing memory stalls.

The analysis and optimization of data locality have been a subject of extensive research for decades, leading to the development of various profiling tools. Examples include Linux `perf` [67], Intel VTune [21], OProfile [44], SCALENE [9], HPCToolkit [5], Witch [84], and RDX [81], which gather performance metrics from hardware performance monitoring units (PMUs). These tools associate performance metrics, such as cache and TLB misses, with instructions/loops/-functions (aka code-centric view) and data object allocations (aka data-centric view). These tools are able to identify the hotspots in functions and object allocations that are worthy of further investigation. However, they fail to provide a holistic view to connect memory layouts of data types and their access patterns for effective optimization guidance.

We introduce TypeCraft, a new profiler designed to enhance memory inefficiency analysis. TypeCraft annotates performance profiles with data type information, offering a more detailed, higher-dimensional view of performance. TypeCraft aggregates performance costs based on data types and their fields, and then ranks them to prioritize optimization efforts. This capability facilitates two key optimizations on data locality: (1) optimizing memory layouts to minimize the number of cache lines required in hot paths and (2) optimizing access patterns to reduce memory transactions.

TypeCraft is designed to be both lightweight and high-resolution, making it suitable for integration into current data center profilers [2, 6, 33, 70, 71] without adding an online data collection burden. This integration allows TypeCraft to deliver a comprehensive, type-centric profile view of the entire data center. Consequently, this enables effective optimization strategies for common libraries across the data center, moving beyond optimizations tailored only to individual workloads. Furthermore, TypeCraft provides high-resolution analysis by identifying data types for individual memory instructions and concentrating on the type field level. This detailed focus allows TypeCraft to offer specific optimization guidance regarding data type layouts and memory access patterns.

TypeCraft artifact and availability TypeCraft is open source and available as part of Linux `perf` tool [67], which is released together with Linux kernel. Kim [38, 77]

*Work was mostly performed while at Google as student researcher.

```
- unsigned int  clock_update_flags;
- u64           clock;
+ /*The following fields of clock data are frequently
+  *referenced and updated together, and should go
+  *on their own cache line.*/
  u64           clock_task ____cacheline_aligned;
  u64           clock_pelt;
+ u64           clock;
  unsigned long lost_idle_time;
+ unsigned int  clock_update_flags;
  u64           clock_pelt_idle;
  u64           clock_idle;
```

Listing 1: The part of patch for reordering the fields of `rq`.

developed the first version of TypeCraft with integration into Linux `perf` tool. Multiple patch series [39–41, 46, 47] have been upstreamed to enable high accuracy and coverage of TypeCraft for production usage.

1.1 Motivation

The Linux kernel is the de-facto operating system used by most data centers. Prior studies [37] show that the Linux kernel functions account for a significant amount of CPU cycles and cache misses. TypeCraft profiles the Linux kernel v6.17 with the scheduler stressed on a 24-core Intel machine. TypeCraft reports a list of expensive types that account for the majority of CPU cycles and last-level cache (LLC) misses. The structure `rq` is among the top of the list, accounting for more than 1% of kernel CPU cycles to access it in our environment (detailed in Section 7). TypeCraft shows the expensive fields of `rq` and their affinities.

By using this information, we can concentrate on optimizing the layout of `rq` by strategically placing frequently accessed fields (hot fields) with high affinity near each other. For instance, Listing 1 shows a patch that groups the `clock_update_flags` and `clock` fields alongside other clock-related fields because they are often accessed concurrently. The enhanced cacheline utilization resulted in a 5.1% rise in instructions per cycle (IPC) for the microbenchmark stressing `rq`. Section 7.1 elaborates on this optimization.

It is worth noting that existing tools make it challenging to find such opportunities and design optimization strategies because they do not provide performance insights about data types and their fields.

1.2 TypeCraft Overview

Figure 1 overviews TypeCraft. TypeCraft accepts fully optimized binary executables and profiles gathered by Linux `perf` as input. It then uses DWARF debugging information and static binary code analysis to determine types for memory instructions. The final output is the profiles, which are annotated with the resolved types. TypeCraft is specifically engineered to address the following challenges, ensuring its suitability for production use.

- DWARF records are focused on debugging rather than performance analysis, making it challenging to precisely link type information to individual memory access instructions. TypeCraft addresses this challenge in Section 3.
- Aggressive optimizations, such as automatic feedback-directed optimization (AutoFDO) [16], link-time optimization (LTO) [35] and post-link optimizations, e.g., BOLT [64] and Propeller [75], often compromise DWARF quality. This hinders TypeCraft's ability to achieve high accuracy and coverage in type resolution. TypeCraft addresses this challenge in Section 4.
- Introducing a new profiler into a modern data center is complex, requiring careful management of production code and the need to maintain low overhead. TypeCraft addresses this challenge in Section 5.

1.3 Contributions

This paper makes four major contributions as follows:

- We introduce TypeCraft, the first profiler capable of attributing data types and fields to individual memory access instructions.
- We employ novel data-flow analysis to ensure high accuracy and coverage, especially for highly optimized binaries with limited DWARF information.
- We validate TypeCraft using real-world workloads, including the off-the-shelf Linux kernel, and demonstrate its suitability for use in data center environments.
- We utilize TypeCraft to guide several Linux kernel optimizations, resulting in notable performance gains. These patches have been submitted for inclusion upstream.

2 Background

Precise PMU sampling Performance Monitoring Units (PMUs) are an indispensable component in modern CPU processors, which are able to monitor various events (aka, metrics), such as cycles, retired instructions, and cache misses. Existing profilers [5, 21, 67] typically configure PMUs via the kernel driver to sample interesting events to enjoy both lightweight measurement and insightful profiles. Although traditional PMUs are powerful, they suffer from *skid* [17], causing them to fail to accurately attribute metrics to the corresponding program counters (PC). Fortunately, most CPU vendors support precise PMUs in their latest CPU generations. Intel's Precise Event-Based Sampling (PEBS) [82], AMD's Instruction-Based Sampling (IBS) [7], and ARM's Statistical Profiling Extension (SPE) [8, 48] are typical examples. These advanced PMUs deliver precise, unskidded profiles, which allows TypeCraft's insightful per-PC type resolution.

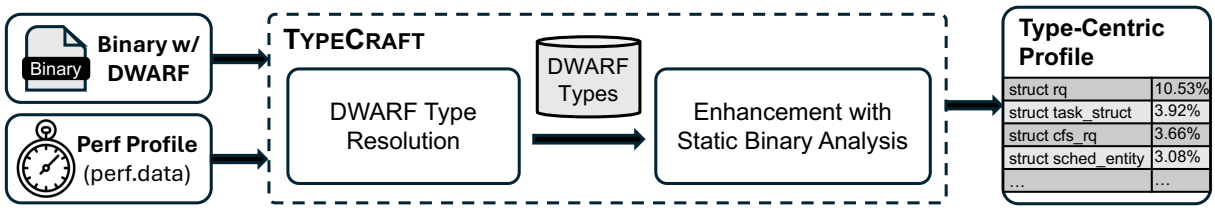


Figure 1: TypeCraft takes as input an unedited, fully optimized binary file that includes DWARF debug data and a profile generated by Linux `perf`. TypeCraft utilizes DWARF and static code analysis to determine the types associated with individual instructions. The final output is a type-annotated profile, which enables metrics to be grouped by specific types and fields.

DWARF debugging information Debugging With Attribute Record Format (DWARF) [19] is the standardized format for embedding debugging metadata within binary files. DWARF provides rich debugging information, such as line mapping from instructions to source code, call frame information, and variable/type names in different lexical scopes. DWARF is widely used in debuggers and profilers to obtain source code insights and determine call paths. It is worth noting that while debuggers (like gdb [78]) can resolve variable names and types—a capability comparable to TypeCraft's—TypeCraft offers three key advantages: (1) TypeCraft maps from program counters to types, while debuggers do not provide such mapping; (2) TypeCraft aims to handle all memory access instructions in the code, unlike debuggers which process a limited subset upon user request; (3) TypeCraft functions on fully optimized binaries, which can have reduced DWARF quality due to various optimizations, whereas debuggers are most effective on unoptimized binaries and require DWARF with high quality.

3 Type Resolution with DWARF

This section shows how TypeCraft leverages the standard DWARF information to resolve the data type for a given memory instruction at a program counter (PC). Although this approach may not achieve high accuracy and coverage, it serves as the foundation of TypeCraft. The enhancement of TypeCraft with static code analysis is described in Section 4.

Figure 2 overviews the workflow of TypeCraft's type resolution based on DWARF. TypeCraft first identifies a series of nested lexical scopes that enclose the input memory instruction. From DWARF, TypeCraft extracts the variables defined and used in these scopes. TypeCraft then decodes the memory instruction for its memory addressing mechanism, which typically includes a base register and a displacement; TypeCraft uses this information to locate the accessed variable at this instruction and resolve its type. Finally, TypeCraft handles conflicts when resolving types in different scopes. TypeCraft uses a heuristic method to produce the deterministic results.

3.1 Identify Lexical Scopes

As shown in Figure 2, DWARF records the lexical scopes and their ranges for a given binary. Such scopes, determined by the PC ranges, include compilation units, subprograms, and (inlined) function frames. For a given PC, TypeCraft is able to identify a series of nested scopes that encloses this PC. TypeCraft limits its analysis to these nested scopes because the input PC can only reference the variables and their types in these scopes. It is worth noting that TypeCraft handles instructions that access global variables by recording global variables from all compilation units.

3.2 Match the Variables and Types

TypeCraft uses the memory addressing formula embedded in the instruction to determine the variable (e.g., local/global variables and function parameters) and type information from all enclosing lexical scopes. The fundamental element for this association is the location descriptions recorded by DWARF. A DWARF location description specifies a variable's data location at a particular PC. This location can be a dedicated register, an offset on the stack, or a complex expression involving memory access, arithmetic, and several registers. We classify the resulting type matches into two categories based on whether the instruction interacts with a value or a pointer.

Interacting with values This scenario arises when an instruction accesses variables by value. The DWARF location description stores the memory address where the variable value is located. For example, a local variable defined as `struct A a` is usually positioned at a specific stack address. This memory address is calculated using the DWARF base-plus-offset operator (e.g., `DW_OP_fbreg` or `DW_OP_bregN` plus an offset). If the instruction's base register aligns with this calculation, and the difference between the instruction's displacement and the DWARF offset is within the size bounds of `struct A`, it signifies that the instruction is directly accessing a field of the variable `a`.

Interacting with pointers This scenario arises when an instruction accesses variables by dereferencing pointers. Specif-

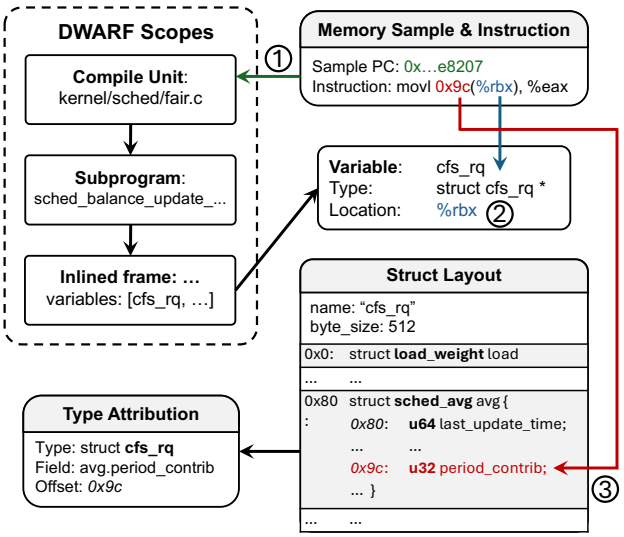


Figure 2: Workflow of type resolution with DWARF. The process involves three steps: ① TYPECRAFT finds candidate scopes by resolving the PC to the nested scope chain. ② TYPECRAFT scans these scopes to link the instruction's base register (`%rbx`) to a variable `cfs_rq` whose location is `%rbx`. ③ TYPECRAFT resolves the offset (`0x9c`) against the struct `cfs_rq` layout to identify the field `avg.period_contrib`.

ically, this happens when the DWARF location description indicates that the variable's value is stored at the location, and the variable itself is a pointer type (e.g., `struct A a*`). If the instruction dereferences `a`'s value, the access is resolved to its pointed-to type, `struct A`. This mechanism primarily appears in two configurations:

- Register location (`DW_OP_regN`): The pointer variable (e.g., `a*`) is stored directly in a register (like `%rbx`). The instruction then performs a memory access operation, such as `mov 0x9c(%rbx), %eax` in Figure 2.
- Base-plus-offset with value tag (`DW_OP_bregN + offset + DW_OP_stack_value`): The variable's value (i.e., the memory address that stores the pointed-to object) is calculated by adding an offset to a register. The presence of the `DW_OP_stack_value` tag explicitly signifies that the result is the variable's value, distinguishing it from an address calculation. This pattern often occurs due to compiler optimizations where a register might point to an internal part of a structure rather than its start.

Handling complex addressing modes Some instructions use more than one register for addressing in x86. In this case, DWARF location descriptions store complex expressions, which make the type matching impossible without a precise symbolic execution engine. To make the system simpler and more robust, we devise a method based on the fact that one register typically acts as the base register while the other acts as an index register. In fully optimized code, DWARF may treat either register as the base register and store it in the location description. Thus, TYPECRAFT attempts to use each register plus the displacement for the type resolution and reports the valid type match.

3.3 Match the Fields inside Types

Once the variable and its fundamental data type are identified, the subsequent step is to precisely locate the specific field being accessed within that type. For composite types (such as structs or classes), fields dictate the internal layout, with their positions defined by either constants or expressions. TYPECRAFT uses the displacement from the instruction to select the member whose offset matches the access point. The output includes the type name, the specific field, and its offset. For primitive types, TYPECRAFT does not further identify fields. If the total size of the data type is available, TYPECRAFT performs a validation check to ensure the access offset remains within the type's memory boundaries. If the size is undetermined (e.g., due to flexible array member [34]), we deduce the field location using other fields whose offsets are known. In cases involving union types, the tool will report the type names and offsets. For bit fields, the tool reports the field that aligns with the byte boundary being accessed.

3.4 Resolve Conflicts

Given the fact that multiple scopes can record the type and field information for a given instruction, we may see conflict analysis results from different scopes. For example, a function passes `a->b` as a parameter to its inlined callee for access. In the caller scope, we are able to resolve the type of `a` and its field `b`, while in the callee scope, we can only obtain the field `b`, as the type information of `a` is not stored in the callee scope. When such a conflict occurs, TYPECRAFT prefers to report the one with richer information. In this example, the type and field resolved from the caller's scope are desired.

It is worth noting that the outer scope does not always produce the desired type and field information. For example, the Linux kernel frequently uses the `container_of` macro to extract the data type for a given field. If this macro is called by a callee, the type information is stored in the callee's scope, rather than the caller's scope. Thus, TYPECRAFT uses the following heuristic method to resolve the conflict and report the deterministic type/field information.

- If TYPECRAFT resolves a composite type in one scope and a primitive type in another scope, TYPECRAFT reports the composite type with a primitive type as its field.
- If TYPECRAFT resolves different types with offsets in different scopes, TYPECRAFT reports the one with the largest offset or within the largest composite type. This typically

happens for nested composite types (i.e., a struct is a field of another struct). This heuristic helps TYPECRAFT report the outermost type with more optimization opportunities.

4 Enhancement with Static Binary Analysis

Relying exclusively on DWARF for type resolution presents two main challenges in maintaining high coverage and accuracy. (1) DWARF cannot handle pointer chasing patterns with a chain of pointer dereferences because of missing variable information. For instance, in the statement `int i = foo->bar->baz`, the DWARF format lacks a record for `bar` to access `baz`. This is because there is no variable corresponding to the intermediate field `bar` explicitly defined in the lexical scopes. This results in the failure of the type resolution for these memory accesses. (2) Another major challenge is that DWARF's accuracy can significantly degrade [45] on heavily optimized code with various optimization passes such as automatic feedback-directed optimization (AutoFDO) [16], link-time optimization (LTO) [35] and post-link optimizations, e.g., BOLT [64] and Propeller [75].

To address these two challenges, TYPECRAFT analyzes the control and data flows of binary code statically to recover missing type information from neighboring instructions. TYPECRAFT combines local analysis on the basic block level and global analysis on the function level. With this effort, TYPECRAFT can significantly improve the coverage and accuracy, which is evaluated in Section 6.

4.1 Abstract Domain

TYPECRAFT structures the static analysis as a *forward data-flow analysis*, a direction chosen to enable the propagation of type data from variable definitions to subsequent memory accesses not covered by DWARF. This simultaneously allows for monitoring any arithmetic modifications to offsets. The execution state at any point in the program is represented by an abstract store σ. This store maps storage locations to abstract values that are sourced from a lattice $\mathcal{L}$, with data types serving as the abstract values used in this analysis.

We partition the domain of storage locations into two categories: *Data Registers (DReg)* and *Stack Frame Locations (SFL)*. While the x86-64 notation is used here, this model is applicable to any ISA. DReg is a subset of general-purpose registers designed to hold a machine word, such as a quadword held by `rax/rdi`. *DReg* excludes stack pointer (`rsp`), instruction pointer (`rip`), and base pointer (`rbp`) when acting as a frame pointer. *SFL* encompasses memory locations situated within the active stack frame. These locations are referenced using offsets relative to the frame base. To ensure coherence between the base and stack pointers, all stack-relative addressing is normalized to offsets originating from the DWARF Canonical Frame Address (CFA) [19]. The mapping for the

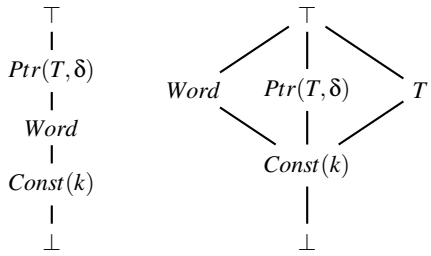


Figure 3: Lattices for DReg (on the left) and SFL (on the right).

abstract store is defined as:

$$\sigma : (DReg \cup SFL) \to \mathcal{L}$$

An update to the store σ is denoted by $\sigma[x \mapsto v]$, which results in a new store identical to σ, except that the location x is mapped to the value v:

$$\sigma[x \mapsto v](y) = v \text{ if } y = x, \text{ else } \sigma(y)$$

The distinction between these domain partitions is based on their unique addressing semantics. Registers within *DReg* typically correspond to a single data object (e.g., a pointer or scalar) over its live range, requiring the tracking of their states for type resolution. In contrast, the registers supporting *SFL* (specifically, `rsp/rbp`) act as base pointers for a memory region containing multiple objects (e.g., local variables and register spills). Consequently, type resolution is achieved by monitoring the state of individual stack locations rather than the `rsp/rbp` registers themselves. Thus, *DReg* and *SFL* possess distinct lattice structures as shown in Figure 3.

***DReg* lattice** The hierarchy of the abstract states for the *DReg* lattice (left in Figure 3) is defined as follows:

- $\bot$ (*Uninitialized*): The location has not been initialized or has become invalid.
- $Const(k)$: The location contains a known integer constant k. This is tracked primarily for resolving pointer offsets.
- *Word*: The location holds a defined machine word (e.g., a 64-bit integer or a memory address on a 64-bit machine), but whether it refers to a memory address is unknown.
- $Ptr(T, \delta)$ (*Pointer to type*): The location is interpreted as a pointer to a composite type T at a byte offset δ from the type's starting point.
- $\top$ (*Conflict*): The location has conflict definitions (e.g., pointers to different types) originating from different paths.

The ordering $Word \sqsubseteq Ptr$ is used to enforce an *optimistic promotion policy* for attributing memory accesses. This policy means that if two control-flow paths suggest that a register

holds a generic *Word* on one path and a specific $Ptr(T)$ on the other, their join results in $Ptr(T)$ instead of *Word*. This is a sound heuristic because TYPECRAFT only resolves types at memory dereference instructions. Consequently, if a path holds a non-pointer value (e.g., a loop counter), it will never execute a dereference and cannot cause a false positive type resolution. This optimistic promotion prevents false positives while significantly decreasing false negatives caused by compiler register reuse and the conservative merging of potentially infeasible control-flow paths.

***SFL* lattice** We use a flatter lattice (right in Figure 3), where *Word*, $Ptr(T,\delta)$, and T are treated as siblings. Note that T appears in this lattice because the object of type T can be directly stored on the stack. Given a stack frame location, we cannot distinguish these states so we treat the join result among them as `Conflict` ($\top$).

4.2 Local Type Analysis

Local type analysis determines the state at the end of each basic block. It tracks the flow of type information including register moves, memory loads, and stack spills at every program counter (PC), beginning from the entry state. DWARF injections are used to source the data-flow analysis and transfer functions are applied for each instruction.

DWARF injection The analysis incorporates DWARF debug data as an external source of information. An injection function, $I_{pc}(\sigma)$, is applied to the current state before processing a given program counter (pc). This DWARF injection serves to restore reliable type information, particularly after instructions like function calls that cause context loss. For example, if DWARF specifies that a variable of a composite type T resides in register r at pc, the state σ is updated to $\sigma[r \mapsto Ptr(T,0)]$. However, injecting DWARF information indiscriminately introduces noise. Therefore, we only apply DWARF injection for variables where the offset δ is 0. The rationale for this constraint is as follows:

- If offset $= 0$, the register points to a variable so we safely set the register to $Ptr(T,0)$, where T is the variable's type.
- If offset $\neq 0$, ambiguity arises as DWARF may provide an inner type but we expect to resolve for an outer one.

Transfer functions for instructions We model the transfer function for each assembly instruction $f_{\text{instr}} : \sigma \to \sigma'$, and Table 1 summarizes these transfer functions in four categories.

- *Data transfer instructions:* The preservation of type information is consistent for values moved between memory and registers. Register-to-register operations replicate the exact state of the source register, and memory loads retrieve the specific field type based on the calculated offset. Stores to the stack are mapped to normalized stack frame locations using DWARF's CFA.
- *Address calculation instructions:* The `lea` instruction is primarily used for pointer arithmetic and creation. When applied to a register base (`lea` $r_d, [r_b + k]$), it modifies the pointer offset δ. If the instruction references the stack, we must determine if the offset falls within the boundary of a known local variable of type T; if so, a pointer is generated to T, adjusted by the relative offset.
- *Arithmetic operations:* Integer arithmetic instructions, such as `add` or `sub`, alter a register's value. When constant arithmetic is applied to a pointer, the offset δ is updated. This principle holds for both immediate values and registers classified as $Const(k)$ in the lattice. However, other arithmetic operations, such as `and`, or non-constant modifications, lead to the invalidation of the destination state. TYPECRAFT adopts a conservative strategy of invalidating the state to prevent the assignment of incorrect types.
- *Function calls:* The call instruction serves as a boundary, invalidating all caller-saved registers by setting them to $\perp$. If DWARF provides the callee's return type, the state of the return register (e.g., `rax` for x86) is updated to this type.

Instruction	**Precondition (σ)**	**Update ($\sigma \to \sigma'$)**
Data transfer instructions		
`mov` r_d, r_s	—	$\sigma' = \sigma[r_d \mapsto \sigma(r_s)]$
`mov` $r_d, [r_b{+}k]$	$\sigma(r_b){=}Ptr(T,\delta)$	Let $f = \text{Field}(T, \delta{+}k)$ $\sigma' = \sigma[r_d \mapsto f]$
`mov` $SFL(x), r_s$	—	$\sigma' = \sigma[SFL(x) \mapsto \sigma(r_s)]$
Address calculation instructions		
`lea` $r_d, [r_b{+}k]$	$\sigma(r_b){=}Ptr(T,\delta)$	$\sigma' = \sigma[r_d \mapsto Ptr(T, \delta{+}k)]$
`lea` $r_d, SFL(x)$	$SFL(x) \in [T_s, T_e]$	$\sigma' = \sigma[r_d \mapsto Ptr(T, x - T_s)]$
Arithmetic operations		
`add/sub` r_d, I	$\sigma(r_d){=}Ptr(T,\delta)$	$\sigma' = \sigma[r_d \mapsto Ptr(T, \delta \pm I)]$
`ARITH` r_d, r_x	$\sigma(r_d){=}Ptr(T,\delta)$	$\sigma' = \sigma[r_d \mapsto \perp]$
Function calls		
`call` *func*	—	$\forall r \in \text{CallerSaved}, \sigma'[r] \leftarrow \perp$ $\sigma'[r_{ret}] \leftarrow \text{RetType}(func)$

Table 1: Transfer functions for different types of instructions. $SFL(x)$ denotes the stack frame location at CFA offset x, k is an integer constant, and I is an immediate or constant register.

Transfer functions for basic blocks The transfer function F_{BB} for a basic block is defined as the composition of instruction transfers and DWARF injections for the instruction sequence $i_1, \ldots, i_n$:

$$F_{BB}(\sigma) = f_{i_n} \circ I_{pc_n} \circ \cdots \circ f_{i_1} \circ I_{pc_1}(\sigma).$$

This composition guarantees that instruction semantics are always interpreted considering the presence of available debug facts. To ensure that the subsequent data-flow analysis

Algorithm 1 The Worklist algorithm used by TYPECRAFT.

```
1:  Input: CFG G = (V, E), DWARF information D
2:  Output: Exit states OUT[b] for all b ∈ V
3:  Initialize: ∀b ∈ V, OUT[b] ← ⊥
4:  I_pc ← BuildFromDWARF(D)  ▷ Build DWARF inject function
5:  Worklist ← {b | Pred(b) = ∅}  ▷ Initialize with entry
6:  while Worklist ≠ ∅ do
7:      b ← Worklist.pop()
8:      σ_in ← ⊔_{p∈Pred(b)} OUT[p]  ▷ Join predecessors
9:      σ_out ← F_BB(σ_in)  ▷ Apply local analysis
10:     if σ_out ≠ OUT[b] then
11:         OUT[b] ← σ_out
12:         Worklist.push(Succ(b))
13:     end if
14: end while
```

terminates, these transfer functions must be monotone with respect to the lattice order $\sqsubseteq$; specifically, if $\sigma_A \sqsubseteq \sigma_B$, then $f(\sigma_A) \sqsubseteq f(\sigma_B)$. Our transfer functions satisfy this monotonicity property for two key operation types:

- *Overwrites* are monotone because the final destination state is set to a fixed abstract value regardless of the source state.
- *Updates* also preserve the lattice order. For example, applying a constant addition to $\bot$ results in $\bot$, and applying a constant addition to $Ptr(T,\delta)$ results in $Ptr(T,\delta')$. Crucially, no operation based on the incoming state can refine a less specific state (like $\top$) into a more specific one (like $Ptr(T,\delta)$).

4.3 Global Type Analysis

To approximate the meet-over-all-paths (MOP) solution [36], a standard worklist algorithm is used, which ensures the analysis spans all basic blocks in a procedure, incorporating control flow merges (joins) and loops during global propagation. Iteration continues until a fixpoint is achieved for the abstract state at the beginning of every block.

Algorithm 1 describes the worklist algorithm for global analysis. We define $OUT[b]$ as the abstract store upon exiting basic block b. The process begins by setting all states to $\bot$ and taking the DWARF injection. It then initializes a worklist with all entry blocks, which addresses disjoint control flow graph (CFG) components often found in optimized binaries. For each block, the entry state is calculated by joining the exit states of all its predecessors (Line 8). Subsequently, the basic block transfer function F_{BB}, derived in Section 4.2, is applied (Line 9). If the resulting output state changes, the successors of the block are then added to the worklist to process in the next iteration.

Join operator ($\sqcup$) The analysis centers on the join operator, which is utilized to reconcile potentially contradictory states inherited from predecessor nodes. The join mechanism, $\sigma_{out} = \sigma_A \sqcup \sigma_B$, applies the lattice hierarchy on a pointwise basis for every location, x. The join operation on two comparable states in the lattice (i.e., when $s_1 \sqsubseteq s_2$) defined in Section 4.1 yields the greater element with respect to $\sqsubseteq$. This behavior is key for several functions: managing initialization by overwriting $\bot$, processing *Word* over *Const* types, and implementing optimistic promotion for registers, which upgrades a generic *Word* to a specific *Ptr* type. We handle the following disagreement raised at the join point.

- *Constant disagreement:* When a location contains different constant values (for instance, $Const(0)$ versus $Const(8)$), the outcome is generalized to the overall *Word* type.
- *Pointer disagreement:* When a location is associated with pointers to different types (such as $Ptr(A,\delta_A)$ versus $Ptr(B,\delta_B)$), the conflict is resolved by checking the relationship between the two types (to be described later). If one type is structurally contained within the other, the state is unified to the composite pointer that encloses it.
- *Other disagreement:* Any other kind of disagreement yields $\top$. For example, merging two pointers, $Ptr(T,\delta_1)$ and $Ptr(T,\delta_2)$, with different offsets results in $\top$, which guarantees that loops involving pointer arithmetic will eventually terminate.

Check relationship between types A problem arises during the merging of two pointer states that refer to the identical memory region but have conflicting type definitions. Consider the scenario where two execution paths converge at block B: Path 1: Register R contains $Ptr(A,0)$; Path 2: Register R contains $Ptr(B,-k)$. A simple approach would result in a Conflict ($\top$) because $A \neq B$. This specific pattern, however, frequently occurs when one path handles a composite type pointer and the other provides a pointer to a distinct member of that type, accompanied by certain offsets. To resolve this, we recursively examine the structure of the larger candidate type to confirm whether the smaller type is a member at offset k. If `struct B` is confirmed to be the type of a member of `struct A` at offset k, the paths are unified as pointing to the same type: $Ptr(A,0) \sqcup Ptr(B,-k) = Ptr(A,0)$. The enclosing type is prioritized because data layout optimizations (such as field reordering) are most effectively applied to the outermost composite type (`struct A`).

4.4 Discussions

Algorithm termination Termination of the analysis is guaranteed because the lattice has finite height, and both the transfer function F_{BB} and the join operator $\sqcup$ are monotonic. Consequently, the state of any location progresses strictly upward from $\bot$ to $Word/Ptr$ until it reaches $\top$, thereby assuring that the worklist algorithm converges to a maximal fixed point that approximates the MOP solution.

Algorithm soundness The analysis utilizes a conservative methodology to ensure the soundness of the algorithm. This approach dictates that if any ambiguity or lack of certainty is encountered during a join operation, the resulting value is explicitly marked as `Conflict`.

5 System Implementation

TypeCraft is implemented as an extension to the Linux `perf` ecosystem [67]. TypeCraft is able to annotate any profile generated by `perf` by sampling precise PMUs. Like `perf`, TypeCraft requires no instrumentation to the target workloads. The type resolution component of TypeCraft is entirely offline, which does not incur any online profiling overhead. TypeCraft has been successfully deployed in a commodity data center and produced unique insights. This section shows the engineering challenges when TypeCraft handles complex production codebases, especially the Linux kernel. We then discuss some limitations of TypeCraft.

5.1 Engineering Challenges

TypeCraft is designed to address various challenges inherent in real-world workloads, specifically those arising from language features and compiler optimizations.

Fragmented data types Scalar Replacement of Aggregates (SROA) [54] is a standard compiler optimization that transforms a composite variable into its constituent, independent scalar fields. This technique improves performance but complicates debugging, as the original composite variable is no longer located at one continuous memory address. TypeCraft addresses this by reconstructing the variable's value from its fragmented locations, which are identified using the `DW_OP_piece` DWARF tag.

KASLR and relocation Kernel Address Space Layout Randomization (KASLR) [26] improves security by randomizing the location of the kernel image at boot time. To overcome the challenges this poses, TypeCraft employs a two-pronged strategy: it leverages the kernel's `kallsyms` and utilizes `perf` to match build-IDs. By combining these methods, TypeCraft successfully computes the difference between runtime and static addresses, allowing it to accurately link program counters sampled at runtime back to their original static DWARF offsets, thus ensuring that address randomization does not interfere with our type resolution.

Per-CPU and segment addressing The Linux kernel frequently uses per-CPU variables accessed via segment registers, such as `mov %gs:0x10, %rax`. TypeCraft resolves the type for this kind of instructions with a segment-aware analysis pass that successfully handles these accesses. The overall process includes: identifying segment-relative addressing patterns; separating structural data, such as fixed-offset stack canaries, from actual data accesses; mapping the offset to the `__per_cpu_offset` array or `this_cpu_off` symbols; and promoting the destination register to a specialized per-CPU pointer type. This approach ensures subsequent accesses are correctly attributed to the specific per-CPU instance of a structure.

Type aliases In production code, typedef is commonly used to create type aliases, which enhances readability. These newly defined types, identified by the DWARF `DW_TAG_typedef` tag, function as symbolic links to specific concrete types but lack their own type layout definitions. To resolve this, TypeCraft is designed to traverse the chain of typedef-defined types until it locates a concrete type, whose information is then used for type resolution.

5.2 System Limitations

First, while TypeCraft generally works for any binary with DWARF, TypeCraft yields the best type resolution for C programs. TypeCraft has a relatively low coverage for Go and C++ programs because of additional language features such as structure embedding in Go and class inheritance in C++. TypeCraft needs to handle extra DWARF information to support these features, which will be our future work. Second, TypeCraft currently works for the x86 ISA. It is straightforward to extend TypeCraft to other architectures, such as ARM, by integrating the necessary ISA decoders. Third, TypeCraft cannot resolve the types that are not recorded in any DWARF scope. This occurs when special pointer encoding [32] and obfuscation [20] are in use. Fortunately, this case is uncommon in practice.

6 Evaluation

We evaluate TypeCraft for the type resolution coverage and overhead. All the experiments are run on a workstation with a 24-core, 48-thread Intel w7-2495X processor with 2.5 GHz frequency, 48KB L1 data cache, 2MB L2 cache, 45MB shared L3 cache, and 128 GB DDR5 memory. We evaluate off-the-shelf Linux 6.17 [80], memcached 1.6.39 [60], Redis 8.4 [69], Git 2.51.1 [29], FFmpeg 8.0 [27], and system software such as Binutils 2.45 [28]. All these workloads are built with `GCC 13.3 -O3 -gdwarf-5`. Moreover, we evaluate CachyOS kernel 6.17 [1], a highly optimized Linux kernel with ThinLTO [35] enabled.

Coverage We evaluate TypeCraft's coverage in two modes. (1) We analyze the entire binary executable and try to resolve types for every memory access instruction. The coverage is computed as the percentage of memory instructions

Workloads	# Instructions	DWARF only		Local analysis		Local+Global analysis		
		Coverage	Time	Coverage	Time	Coverage	Time	Cycle Coverage
Linux 6.17 Ubuntu [80]	4.6M	75.2%	2248s	77.6%	2299s	92.7%	2900s	92.8%
Linux 6.17 CachyOS [1]	4.4M	66.2%	6030s	67.9%	6101s	86.2%	6729s	90.2%
ld [28]	312.9K	63.2%	11s	63.7%	12.4s	80.2%	26.5s	98.9%
ar [28]	227.5K	59.9%	9.2s	60.0%	10.3s	78.3%	20.8s	78.9%
nm [28]	225.6K	59.8%	9.1s	59.9%	10.2s	78.3%	20.5s	99.7%
objdump [28]	348.2K	61.2%	14.3s	61.3%	17s	80.3%	41.8s	85.4%
readelf [28]	189.2K	67.7%	5.1s	67.7%	6.1s	82.3%	18.3s	89.1%
memcached [60]	42.5K	66.1%	1.1s	66.6%	1.3s	77.4%	2.9s	86.5%
redis-benchmark [69]	151.7K	51.6%	5.6s	53.5%	6.7s	68.0%	13.6s	76.0%
redis-server [69]	623.7K	47.9%	27.2s	48.7%	45.2s	67.6%	69.9s	79.1%
git [29]	675.4K	66.2%	10.7s	68.7%	13.4s	86.2%	32.5s	90.3%
FFmpeg [27]	1.46M	40.9%	49.4s	41.0%	59.2s	74.2%	204.2s	40.0%[1]

[1] For the H.264 decoding workload, 96% of the uncovered memory cycles are attributed to functions with hand-written SIMD assembly codes that operate on raw character buffers. The type information is not available in the DWARF debugging information for such codes. However, TYPECRAFT still recovers the dominant decoder structures to guide optimization as decribed in Section 7.4.

Table 2: Coverage and overhead of TYPECRAFT.

with type annotations relative to the total number of memory instructions. (2) We resolve types for memory instructions that have profiling metrics and compute the coverage for these important instructions. We use standard workloads for profiling and measure CPU cycles via Intel PEBS PMUs. Table 2 shows the results, which indicate that TYPECRAFT can yield high coverage, especially for instructions with profiling metrics. While TYPECRAFT yields more than 90% coverage for the Linux kernel, TYPECRAFT achieves 70-80% coverage for userspace workloads. The reduced coverage is caused by userspace workloads utilizing a higher number of unnamed literals and constants. These are stored in the `.rodata` section without DWARF information.

TYPECRAFT offers reliable type profiles as its coverage typically exceeds 80%. For example, TYPECRAFT covers memory instructions with 92% CPU cycles in the Ubuntu Linux kernel. The highly optimized CachyOS also shows 90% coverage. Note that compared to Ubuntu, the DWARF quality of CachyOS is noticeably reduced (66% vs. 75% coverage with DWARF only), but the cycle coverage is comparable thanks to TYPECRAFT's data-flow analysis. We observe a similar coverage in the production kernel fully optimized with AutoFDO [16], ThinLTO [35], and Propeller [75].

Overhead As a unique advantage, TYPECRAFT does not incur any online overhead because the type resolution occurs offline. We measure TYPECRAFT's overhead when resolving types for all memory instructions, which serves as the overhead upper bound. Table 2 shows that TYPECRAFT has a reasonable processing time across workloads. Note that TYPECRAFT's overhead can be amortized by reusing the type resolution for different profiles of the same binary executable.

7 Case Studies

This section demonstrates the efficacy of TYPECRAFT. By employing TYPECRAFT, we successfully pinpointed costly types within the Linux kernel 6.17 and developed practical solutions to address them. Our analysis and subsequent optimizations were derived from performance profiles gathered in a production data center environment. While the specific data center profiles cannot be shared due to confidentiality, we have replicated the observed data center behaviors using carefully configured microbenchmarks and open-source workloads. Crucially, internal measurements performed in Google's data center confirmed substantial performance improvements resulting from these optimizations. The optimization patches are currently awaiting upstream review. Besides the Linux kernel optimization, we also show several case studies on the userspace workloads at the end of this section.

Environment setup We use the same system as described in Section 6 for the case study. To stress on the Linux kernel, we run MySQL Server 8.0.44 [3], a popular, open-source relational database management system. We enable up to 256 servers in a 4-width and 5-depth cgroup hierarchy. We place the 75% MySQL servers in throttled cgroups that utilize 50% machine resources at most, which simulates the batch jobs in data centers. We place the remaining 25% MySQL servers in unlimited cgroups with no resource constrains, which simulates latency sensitive jobs in data centers. We run the Sysbench OLTP read-only benchmark [42] in the unlimited cgroup creating the requests to all the MySQL servers. We run four threads per server. The database size is set to two tables with 10k rows each for each server. We run the experiment for 30 seconds for 10 times to report the average

Data Type	Cycles %	LLC misses %
cfs_rq	7.58%	49.02%
sched_entity	4.73%	6.05%
rq	1.05%	0.35%
cfs_rq *	0.21%	0.86%
sched_entity *	0.16%	2.41%

Table 3: Hot type reported by TYPECRAFT in Linux kernel. The percentages of these metrics are relative to the total metrics consumed in the kernel.

and standard deviation.

Profiles and insights We collect the metrics—CPU cycles, retired instruction counts, and last-level cache (LLC) misses—via TYPECRAFT. The top expensive types in the Linux kernel are structures `rq`, `cfs_rq`, and `sched_entity`. Table 3 shows the percentage of these metrics caused by accessing these types over the total metrics consumed in the kernel. All these types are from the kernel scheduling subsystem, which aligns with the prior study [37] that shows the scheduling subsystem is a major bottleneck in modern data centers.

Optimizations and validation We devise two optimizations (Sections 7.1 and 7.2) for these expensive data types guided by TYPECRAFT. We use microbenchmarks to validate the effectiveness of these optimizations and also quantify the throughput improvement across all MySQL servers (Section 7.3).

7.1 Reordering Fields for Expensive Types

Field reordering [22] is a well-known optimization technique to improve cacheline utilization, which rearranges the fields of a given data type according to the following rules.

- Group the hot fields together. Hot fields are defined as the fields that are frequently accessed.
- Group the fields of high affinity together. Fields of high affinity mean they are frequently accessed together.
- Separate the read- and write-intensive fields as placing them together may result in false sharing.

Insights from TYPECRAFT The fields of data type T are ranked by TYPECRAFT using retired instruction counts to determine their hotness. Specifically, one can choose the top several hot fields within `T` that have the highest instruction counts to access them. For each hot field, TYPECRAFT provides all the monitored PCs that access it. For each PC, TYPECRAFT analyzes all the adjacent memory instructions in the same function to obtain other fields of type `T` accessed. TYPECRAFT treats these fields as of high affinity and weighted by

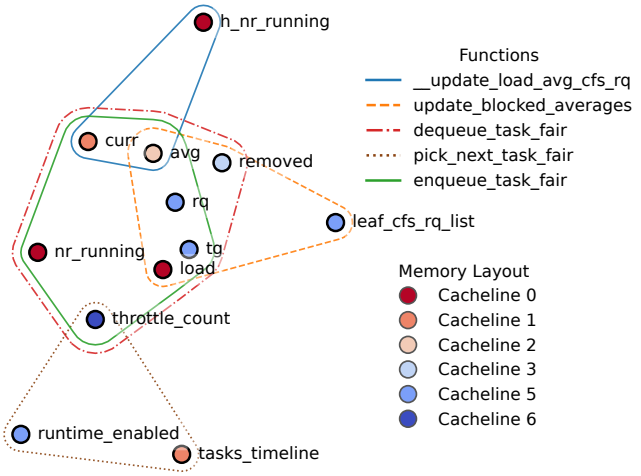


Figure 4: Field affinity of `cfs_rq` reported by TYPECRAFT. This graph only shows the hot fields in `cfs_rq`.

their hotness. By analyzing these instructions, TYPECRAFT is able to determine whether the access is a load or a store. Thus, TYPECRAFT can provide all the information necessary to guide the field reordering optimization.

TYPECRAFT outputs a hypergraph that visualizes the affinity of hot fields for a given data type. Figure 4 shows an example for `cfs_rq`. Each point denotes a field with its name and the color reflects the cacheline number it resides relative to the start of the struct. Each line scope defines a function, meaning the fields are accessed together in this function. From the figure, we observe that field affinity varies in different functions. For example, in the blue line scope, `h_nr_running` affinities with `curr` and `avg` but not in the brown and orange line scopes. Since we optimize a type layout for the whole software rather than individual functions, we define fields of high affinity if they are accessed together in contexts with high instruction counts. For `cfs_rq`, we place the `curr`, `throttle_count`, `rq`, `tg` fields to the same cacheline to improve data locality in hot paths. Although fields such as `leaf_cfs_rq_list` and `tasks_timeline` are also frequently accessed, they are only used in one function. Thus, we place them in a different cacheline from other hot fields.

Performance impact With the guidance of TYPECRAFT, we optimize all `rq` (as shown in Section 1.1), `cfs_rq`, and `sched_entity`. For `rq`, a microbenchmark was utilized. This benchmark generates 960 tasks that execute a loop, sleeping for 1ms per iteration. This resulted in a 5.1% reduction in kernel cycles. To evaluate `cfs_rq`, we configure schbench [4], a kernel scheduling benchmark designed to stress the hot paths associated with these types. Our configuration creates 1024 cgroups and 1024 tasks, showing a 26.4% reduction in kernel LLC misses.

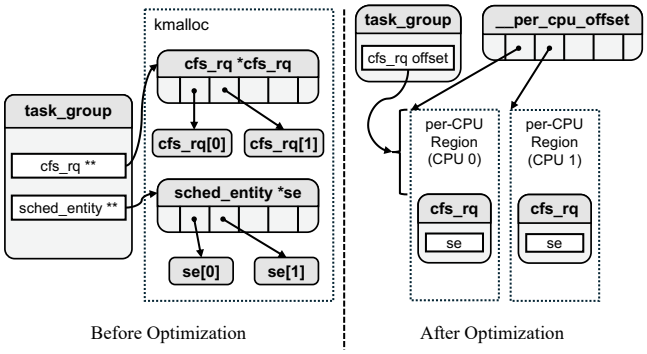


Figure 5: Memory layout before vs. after pointer chasing optimization, which matches the access patterns to avoid expensive pointer chasing.

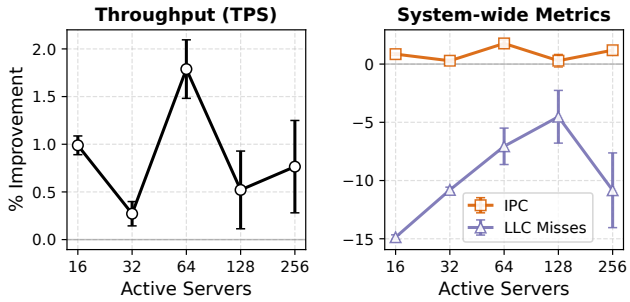


Figure 6: TPS and PMU metric improvement for MySQL. Error bar shows the standard deviation across 10 runs.

7.2 Avoiding Pointer Chasing

Pointer chasing [18] is a well-known access pattern in which a memory load depends on the result of its prior memory load. Due to such dependency, the two loads cannot overlap in the pipeline, resulting in long exposed memory latency.

Insights from TypeCraft TypeCraft pinpoints 0.4% of kernel CPU cycles and 3.3% of kernel LLC misses for accessing `sched_entity*` and `cfs_rq*` pointers themselves, resulting in even more costs to access the objects via these pointers. This suggests that expensive pointer chasing is occurring. Furthermore, the tool provides the associated program counters that are mapped to the source code with a typical access pattern: `cfs_rq->tg->se[cpu]`. There are multiple pointer indirections from `cfs_rq` to access a `sched_entity` object `se[cpu]`, confirming the pointer chasing pattern.

To eliminate pointer chasing, we employ two optimizations: (1) embedding `sched_entity` objects into `cfs_rq` and (2) changing the allocations for `cfs_rq` objects. Figure 5 highlights the memory layout changes with our optimizations. With the new memory layout, accessing `sched_entity` from `cfs_rq` does not incur pointer chasing.

Embedding `sched_entity` As shown in Figure 5, all `sched_entity` objects are allocated randomly in the memory and managed by an array of pointers (i.e., `sched_entity **`) owned in `task_group`. We observe that both `cfs_rq` and `sched_entity` objects are allocated per core and have the same life ranges. Thus, we are able to embed `sched_entity` object as a field in `cfs_rq` with a constant offset. With this optimization, the original pointer chasing pattern can be optimized to avoid pointer indirections, i.e., `cfs_rq + offset`.

Per-CPU allocation for `cfs_rq` objects Similarly, each `task_group` object (i.e., `tg`) stores `cfs_rq` pointers in an array. Hot functions such as `tg_throttle_down` and `tg_unthrottle_up` traverse the list of task groups and repeatedly access a `cfs_rq` object for a given `cpu`. Our optimization is to use the per-CPU allocator for `cfs_rq` objects rather than maintaining them in an array of pointers. When accessing the `cfs_rq` object for a given `cpu`, we use an efficient per-CPU addressing via `__per_cpu_offset`. This optimization enjoys a more efficient addressing strategy and avoids the costly pointer chasing on `struct cfs_rq *`.

Performance impact We reuse the `cfs_rq` microbenchmark described in Section 7.1. Eliminating pointer chasing generally reduces cache misses and saves CPU cycles. However, our optimization on embedding `sched_entity` shows a significant performance regression in microbenchmarks—the LLC misses are increased by more than 50% when accessing `cfs_rq`. The reason is that the optimized `cfs_rq` type is enlarged by embedding a `sched_entity` object. This makes the kernel slab allocator align all `cfs_rq` objects to the 1024-byte boundary. When we traverse different `cfs_rq` objects, we can have power-of-two strided accesses that incur conflict cache misses. Fortunately, our follow-up optimization of per-CPU allocator for `cfs_rq` objects can avoid cache conflicts because the per-CPU allocator does not force a power-of-two overalign for `cfs_rq` objects. The combination of these two optimizations further reduces kernel LLC misses by 33% on our microbenchmark on top of the reordering optimizations. Intel kernel test robot also confirms this improvement, showing 8.8% improvement on the `stress-ng.session.ops_per_sec` benchmark.

7.3 Performance Impact on MySQL

Figure 6 demonstrates the effects of our kernel optimizations on MySQL's throughput (TPS) and microarchitectural metrics. A consistent reduction in LLC misses is observed across all setups, with the most significant reduction, 14.9%, occurring when 16 servers are active. This enhanced data locality directly boosts execution efficiency. Consequently, throughput improves at every concurrency level, reaching a peak TPS increase of 1.8% (along with a 1.8% rise in IPC) when running

on 64 active servers. As the number of active servers increases, so does the scheduling contention, resulting in greater variability in these performance indicators. We also see a throughput improvement of the same magnitude for the Google data center with our optimizations. Note that even a 1% improvement for a data center means saving millions of dollars.

7.4 Analysis of Other Workloads

In addition to the Linux kernel, we evaluate TypeCraft on several prevalent and highly optimized user-space applications. Through profiling FFmpeg, Git, and Binutils using realistic workloads, TypeCraft discovers various prospects for improving data layout. Note that though we obtain nontrivial improvement in memory subsystems on the function level, we do not see obvious end-to-end improvement for FFmpeg and Git. The potential reason is that isolated benchmark runs face much less cache and memory pressure than the datacenter, where contention for the shared L3 cache and memory bandwidth could amplify the benefit of our data layout optimization.

FFmpeg During H.264 decoding in FFmpeg, TypeCraft identifies `H264SliceContext` as a significant consumer of CPU cycles (3.4%). This 35KB structure contains a 2KB set of frequently accessed fields (per-macroblock inner loop) that are separated by `pwt`, a 20KB sub-struct written only once per-slice outer loop and rarely read in the inner loop. By moving the cold fields, such as `pwt` to the end of the structure, we cluster the hot fields into a contiguous memory region. This reorganization prevents hot fields from spanning disjoint cachelines and multiple memory pages, leading to a 4.8% reduction in L1-dcache misses and a 2.5% decrease in dTLB misses.

Git For the `git log --stat` workload, TypeCraft highlights the 112B `xdlclassifier_t` structure. Despite its compact size, its hot fields are split across two cachelines. By consolidating these hot fields into the first cacheline and shifting cold fields to the second, we minimize unnecessary cacheline fetches. This change reduces L1-dcache misses by 24% in the hot `xdl_prepare_env` function and 4% for the overall Git stat execution.

Binutils (nm) When running `nm` on a Linux kernel binary, TypeCraft finds that the 48-byte `asymbol` structure accounts for nearly 10% of cycles. Although it fits in one cacheline, `asymbol` objects are fragmented in memory. This causes pointer-chasing patterns in functions like `qsort` and binary search, resulting in significant cache misses. To address this, we implement structure splitting for `asymbol`. We introduce a dense `slots[]` array containing only the critical sorting keys—`name`, `section`, and `value`—extracted from the original structure. This allows sorting and searching routines to leverage better spatial locality by processing contiguous, packed data. This optimization yields a 32.1% reduction in L1-dcache misses and a 55.4% drop in dTLB misses, ultimately improving end-to-end execution time by 2.7%.

8 Related Work

There are substantial efforts [49, 61, 62, 76, 87, 90, 91] in the security domain to reverse engineer the binary code for types. Typically, these approaches assume DWARF is not available, which is different from TypeCraft. This section only reviews the performance analysis tools that target CPU memory subsystems. We categorize existing approaches into code-centric, data-centric, and type-centric profilers.

8.1 Code-Centric Profilers

Code-centric profilers associate performance metrics with code regions, such as instructions, loops, and functions. Tools such as Linux `perf` [67], HPCToolkit [5], VTune [21], OProfile [44], SCALENE [9], CodeAnalyst [25], and a variety of data center profilers [2, 6, 30, 33, 70, 71] collect PMU events or OS timers with low overhead. Many of these tools are used in production and provide insights on the code level. These tools can identify the hot code that worth optimization but provide limited guidance on optimization.

Some advanced tools provide additional metrics beyond PMU events. CPROF [43], MemSpy [58], SLO [10], DrCCTProf [92], and MACPO [68] collect memory reuse distance [23, 59] to guide the optimization of capacity cache misses. HOTL [86] calculates the memory footprint to complement the reuse distance. RVN [85], DeadSpy [14], RedSpy [83], ZeroSpy [88], and LoadSpy [79] identify redundant memory transactions with value profiling [13]. CacheGrind [74] simulates cache behaviors and identifies cacheline fragmentation [22]. SHERIFF [50] analyzes multithreaded programs to identify false sharing. These tools pinpoint a series of instructions that are involved in the memory inefficiencies and give insightful guidance to optimize the code. These tools instrument memory accesses with compilers or binary engines such as Pin [56], DynamoRIO [11], and Valgrind [73]. Thus, they typically incur high overhead, which are not used in production environment.

Furthermore, some lightweight tools aim to collect these advanced metrics. ThreadSpotter [55, 65], Witch [84], Feather [15], Cheetah [51], and RDX [81] use PMUs, debug registers, or OS page protection to pinpoint redundant memory transactions, identify false sharing, and compute reuse distances. None of these code-centric profilers provides the type information. Actually, TypeCraft can complement these profilers by annotating their assembly profiles.

8.2 Data-Centric Profilers

The data-centric profilers associate metrics with global/static data symbols or heap data allocations. Cache Scope [12], HPCToolkit [52], MemPerf [93], and Oracle Developer Studio [63] are the typical profilers in this category. Advanced analyses based on these tools, such as LWPTool [89], ArrayTool [53], and StructSlim [72], guide array regrouping and structure splitting to reduce cache misses.

The optimization guidance shared between data-centric profilers and TypeCraft centers on memory layout optimization. However, TypeCraft distinguishes itself from data-centric profilers in three key areas. First, TypeCraft offers a global perspective on data types to inform memory layout optimization; while data types can be allocated across various contexts, TypeCraft can aggregate them, whereas data-centric profilers demand significant manual analysis of allocation call sites to determine types. Second, TypeCraft directly examines memory access instructions, eliminating the need to intercept memory allocators as is required by data-centric profilers. Third, TypeCraft provides detailed information at the type field level, in contrast to existing data-centric profilers, which provide information only for the entire allocated data object.

8.3 Type-Centric Profilers

Perhaps DProf [66] is the most relevant work to TypeCraft. DProf leverages PMUs to identify memory inefficiencies in the Linux kernel. DProf offers limited type resolution for dynamically allocated objects. DProf relies on (1) a modified allocator to record the type of each allocation and (2) per-type memory pools that record the types for the allocations from these pools. Moreover, DProf requires manual efforts to resolve some types. TypeCraft differs in three aspects.

- TypeCraft uses DWARF to resolve data types without depending on any customized allocators or manual efforts.
- TypeCraft offers fine-grained type resolutions on the fields, so it can guide the field-level optimization.
- TypeCraft works on an off-the-shelf Linux kernel, while DProf requires a customized kernel and necessary modules.

9 Conclusions and Future Work

This paper introduces TypeCraft, a lightweight, high-resolution data type profiler. TypeCraft utilizes the standard DWARF information embedded in the binary executable to determine the type and field details for memory instructions. To boost the coverage of type resolution in highly optimized binaries, TypeCraft employs a novel data-flow analysis to disseminate type information to memory instructions missing DWARF records due to various optimizations. Our assessment of the latest off-the-shelf Linux kernel indicates that TypeCraft successfully resolves types for over 90% of memory instructions. Moreover, we have employed TypeCraft to direct several optimizations within the Linux kernel, resulting in considerable performance gains in benchmarks and the production data center at Google. All resulting optimization patches are currently awaiting upstream inclusion. TypeCraft will be released with the Linux `perf` tool for public access.

Our future development strategy focuses on three key areas. First, we plan to enhance support for languages beyond C, specifically targeting C++, Go, and Rust by addressing the DWARF extensions necessary for their unique language features. Second, we aim to enable support for the ARM architecture, which is becoming prevalent in modern data centers. Third, we will work on improving DWARF generation for mainstream compilers to further increase the coverage and effectiveness of TypeCraft.

Acknowledgement

We are grateful for Namhyung Kim's contributions to this project. In addition to reviewing all patch series for the upstreaming of TypeCraft into the Linux `perf` tool, Namhyung Kim was responsible for developing the initial version of TypeCraft built upon Linux `perf`. We also thank our shepherd and all reviewers for their valuable comments. This work is partially supported by the National Science Foundation (NSF) under Grants No. CISE-2316201.

References

[1] Optimizing the Kernel with AutoFDO on CachyOS. https://cachyos.org/blog/2411-kernel-autofdo, Nov 2024.

[2] Datadog Profilers. https://docs.datadoghq.com/profiler/, 2025.

[3] MySQL Reference Manual. https://dev.mysql.com/doc, 2025.

[4] schbench: A Linux scheduler benchmark. https://kernel.googlesource.com/pub/scm/linux/kernel/git/mason/schbench/, 2025.

[5] L. Adhianto, S. Banerjee, M. Fagan, M. Krentel, G. Marin, J. Mellor-Crummey, and N. R. Tallent. Hpctoolkit: tools for performance analysis of optimized parallel programs http://hpctoolkit.org. *Concurr. Comput.: Pract. Exper.*, 22(6):685–701, April 2010.

[6] Amazon Web Services, Inc. *Amazon CodeGuru Profiler User Guide*, 2025.

[7] AMD and the Linux Community. *perf-amd-ibs(1) - Support for AMD Instruction-Based Sampling (IBS) with perf tool*. Linux Manual Pages, 2025.

[8] Arm and the Linux Community. *perf-arm-spe - Support for Arm Statistical Profiling Extension within perf tools*. Linux Manual Pages, 2025.

[9] Emery D. Berger, Sam Stern, and Juan Altmayer Pizzorno. Triangulating python performance issues with SCALENE. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI '23)*, pages 51–64. USENIX Association, 2023.

[10] Kristof Beyls and Erik H. D'Hollander. Refactoring for data locality. *Computer*, 42(2):62–71, February 2009.

[11] Derek Bruening, Saman Amarasinghe, et al. DynamoRIO: Dynamic instrumentation tool platform. https://dynamorio.org/, 2024.

[12] Bryan R. Buck and Jeffrey K. Hollingsworth. *Data centric cache measurement using hardware and software instrumentation*. PhD thesis, USA, 2004. AAI3123989.

[13] Brad Calder, Peter Feller, and Alan Eustace. Value profiling and optimization. *Journal of Instruction-Level Parallelism*, 1:94–112, 1999.

[14] Milind Chabbi and John Mellor-Crummey. Deadspy: a tool to pinpoint program inefficiencies. In *Proceedings of the Tenth International Symposium on Code Generation and Optimization*, CGO '12, page 124–134, New York, NY, USA, 2012. Association for Computing Machinery.

[15] Milind Chabbi, Shasha Wen, and Xu Liu. Featherlight on-the-fly false-sharing detection. In *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '18, page 152–167, New York, NY, USA, 2018. Association for Computing Machinery.

[16] Dehao Chen, David Xinliang Li, and Tipp Moseley. Autofdo: automatic feedback-directed optimization for warehouse-scale applications. In *Proceedings of the 2016 International Symposium on Code Generation and Optimization*, CGO '16, page 12–23, New York, NY, USA, 2016. Association for Computing Machinery.

[17] Dehao Chen, Neil Vachharajani, Robert Hundt, Shih-wei Liao, Vinodha Ramasamy, Paul Yuan, Wenguang Chen, and Weimin Zheng. Taming hardware event samples for fdo compilation. In *Proceedings of the 8th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '10, page 42–52, New York, NY, USA, 2010. Association for Computing Machinery.

[18] Jamison Collins, Suleyman Sair, Brad Calder, and Dean M Tullsen. Pointer cache assisted prefetching. In *35th Annual IEEE/ACM International Symposium on Microarchitecture, 2002.(MICRO-35). Proceedings.*, pages 62–73. IEEE, 2002.

[19] DWARF Committee. *DWARF Debugging Information Format Version 5 Standard*. The DWARF Standards Committee, 2017.

[20] Kees Cook. mm: Add slub free list pointer obfuscation. Linux Kernel Mailing List, August 2017. https://lore.kernel.org/all/20170802180609.GA66807@beast/.

[21] Intel Corporation. Intel® vtune™ profiler. https://www.intel.com/content/www/us/en/docs/vtune-profiler/user-guide/, 2025.

[22] Chen Ding and Ken Kennedy. Improving cache performance in dynamic applications through data and computation reorganization at run time. *ACM SIGPLAN Notices*, 34(5):229–241, 1999.

[23] Chen Ding and Yutao Zhong. Predicting whole-program locality through reuse distance analysis. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation*, PLDI '03, page 245–257, New York, NY, USA, 2003. Association for Computing Machinery.

[24] Ulrich Drepper. What every programmer should know about memory. Technical report, Red Hat, Inc., Nov 2007.

[25] Paul Drongowski, Lei Yu, Frank Swehosky, Suravee Suthikulpanit, and Robert Richter. Incorporating instruction-based sampling into amd codeanalyst. In *2010 IEEE International Symposium on Performance Analysis of Systems & Software (ISPASS)*, pages 119–120, 2010.

[26] Jake Edge. Kernel address space layout randomization. LWN.net, October 2013. https://lwn.net/Articles/569635/.

[27] FFmpeg Developers. FFmpeg source code. https://git.ffmpeg.org/ffmpeg.git, 2026.

[28] Free Software Foundation, Inc. GNU Binutils. https://www.gnu.org/software/binutils, 2025.

[29] Git Developers. Git source code. https://github.com/git/git, 2026.

[30] Brendan Gregg. Visualizing system latency. *Communications of the ACM*, 53(7):48–54, 2010.

[31] John L. Hennessy and David A. Patterson. *Computer Architecture, Fifth Edition: A Quantitative Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 5th edition, 2011.

[32] Liam Howlett. The maple tree: Storing ranges and dumping the tree. Oracle Linux Blog, June 2024. https://blogs.oracle.com/linux/post/maple-tree-storing-ranges.

[33] Microsoft Inc. Profiling live Azure web apps with Application Insights. https://learn.microsoft.com/en-us/azure/azure-monitor/app/profiler, 2025.

[34] ISO/IEC. *ISO/IEC 9899:1999(E) – Programming Languages – C*. International Organization for Standardization/International Electrotechnical Commission, 1999. The C99 Standard, which introduced the Flexible Array Member (FAM).

[35] Teresa Johnson, Mehdi Amini, and Xinliang David Li. Thinlto: scalable and incremental lto. In *Proceedings of the 2017 International Symposium on Code Generation and Optimization*, CGO '17, page 111–121. IEEE Press, 2017.

[36] John B. Kam and Jeffrey D. Ullman. Monotone data flow analysis frameworks. *Acta Inf.*, 7(3):305–317, September 1977.

[37] Svilen Kanev, Juan Pablo Darago, Kim Hazelwood, Parthasarathy Ranganathan, Tipp Moseley, Gu-Yeon Wei, and David Brooks. Profiling a warehouse-scale computer. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture*, ISCA '15, page 158–169, New York, NY, USA, 2015. Association for Computing Machinery.

[38] Namhyung Kim. perf tools: Introduce data type profiling, December 2023. https://lore.kernel.org/all/20231213001323.718046-1-namhyung@kernel.org/.

[39] Namhyung Kim. perf annotate-data: Update data-type profiling quality (v2), August 2024. https://lore.kernel.org/all/20240821232628.353177-1-namhyung@kernel.org/.

[40] Namhyung Kim. perf mem: Basic support for data type profiling (v1), July 2024. https://lore.kernel.org/all/20240731235505.710436-1-namhyung@kernel.org/.

[41] Namhyung Kim. perf annotate: Add –code-with-type option, March 2025. https://lore.kernel.org/all/20250310224925.799005-1-namhyung@kernel.org/.

[42] Alexey Kopytov and The Sysbench Community. Sysbench: Scriptable database and system performance benchmark. https://github.com/akopytov/sysbench, 2025.

[43] A.R. Lebeck and D.A. Wood. Cache profiling and the spec benchmarks: a case study. *Computer*, 27(10):15–26, 1994.

[44] John Levon, Philippe Elie, et al. Oprofile: A system profiler for linux. https://oprofile.sourceforge.io, 2023.

[45] Yuanbo Li, Shuo Ding, Qirun Zhang, and Davide Italiano. Debug information validation for optimized code. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2020, page 1052–1065, New York, NY, USA, 2020. Association for Computing Machinery.

[46] Zecheng Li. perf tools: Some improvements on data type profiler, September 2025. https://lore.kernel.org/all/20250917195808.2514277-1-zecheng@google.com/.

[47] Zecheng Li. perf tools: Improvements to data type profiler, March 2026. https://lore.kernel.org/all/20260309175546.916039-1-zli94@ncsu.edu/.

[48] Linaro Limited. Arm statistical profiling extension (spe) overview. Linaro Forge Documentation, 2021.

[49] Zhiqiang Lin, Xiangyu Zhang, and Dongyan Xu. Automatic reverse engineering of data structures from binary execution. In *Proceedings of the 11th Annual Information Security Symposium*, CERIAS '10, West Lafayette, IN, 2010. CERIAS - Purdue University.

[50] Tongping Liu and Emery D. Berger. Sheriff: precise detection and automatic mitigation of false sharing. *SIGPLAN Not.*, 46(10):3–18, October 2011.

[51] Tongping Liu and Xu Liu. Cheetah: Detecting false sharing efficiently and effectively. In *2016 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 1–11, 2016.

[52] Xu Liu and John Mellor-Crummey. A data-centric profiler for parallel programs. In *SC '13: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, pages 1–12, 2013.

[53] Xu Liu, Kamal Sharma, and John Mellor-Crummey. Arraytool: a lightweight profiler to guide array regrouping. In *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation*, PACT '14, page 405–416, New York, NY, USA, 2014. Association for Computing Machinery.

[54] LLVM Project. *LLVM Passes: Scalar Replacement of Aggregates (SROA)*, 2025.

[55] Royd Lüdtke and Chris Gottbrath. Cache-related performance analysis using rogue wave software's threadspotter. In *Tools for High Performance Computing 2012*, pages 75–93. Springer, 2013.

[56] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. *SIGPLAN Not.*, 40(6):190–200, June 2005.

[57] Zhihong Luo, Sam Son, Sylvia Ratnasamy, and Scott Shenker. Harvesting memory-bound CPU stall cycles in software with MSH. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, pages 57–75, Santa Clara, CA, July 2024. USENIX Association.

[58] Margaret Martonosi, Anoop Gupta, and Thomas Anderson. Memspy: analyzing memory system bottlenecks in programs. *SIGMETRICS Perform. Eval. Rev.*, 20(1):1–12, June 1992.

[59] R.L. Mattson, J. Gecsei, D. R. Slutz, and I. L. Traiger. Evaluation techniques for storage hierarchies. *IBM Systems Journal*, 9(2):78–117, 1970.

[60] Memcached project team. Memcached: a distributed memory object caching system. https://memcached.org/, 2025. Accessed on 10 December 2025.

[61] Daniel Mercier, Aziem Chawdhary, and Richard Jones. dynstruct: An automatic reverse engineering tool for structure recovery and memory use analysis. In *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 497–501, 2017.

[62] Matt Noonan, Alexey Loginov, and David Cok. Polymorphic type inference for machine code. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '16, page 27–41, New York, NY, USA, 2016. Association for Computing Machinery.

[63] Oracle Corporation. *Oracle Developer Studio*, 2025.

[64] Maksim Panchenko, Rafael Auler, Bill Nell, and Guilherme Ottoni. Bolt: a practical binary optimizer for data centers and beyond. In *Proceedings of the 2019 IEEE/ACM International Symposium on Code Generation and Optimization*, CGO 2019, page 2–14. IEEE Press, 2019.

[65] ParaTools, Inc. *ParaTools ThreadSpotter User Manual*, 2025.

[66] Aleksey Pesterev, Nickolai Zeldovich, and Robert T. Morris. Locating cache performance bottlenecks using data profiling. In *Proceedings of the 5th European Conference on Computer Systems*, EuroSys '10, page 335–348, New York, NY, USA, 2010. Association for Computing Machinery.

[67] Mauro Petazzoni and the Linux Community. *perf-record - Run a command and record its profile into perf.data*. Linux Manual Pages, 2024.

[68] Ashay Rane and James Browne. Performance optimization of data structures using memory access characterization. In *2011 IEEE International Conference on Cluster Computing*, pages 570–574, 2011.

[69] Redis Community. Redis, In-memory Data Structure Store. https://redis.io, 2025.

[70] Gang Ren, Eric Tune, Tipp Moseley, Yixin Shi, Silvius Rus, and Robert Hundt. Google-wide profiling: A continuous profiling infrastructure for data centers. *IEEE Micro*, 30(4):65–79, 2010.

[71] Jordan Rome. Strobelight: A profiling service built on open source technology. https://engineering.fb.com/2025/01/21/production-engineering/strobelight-a-profiling-service-built-on-open-source-technology/, Jan 2025.

[72] Probir Roy and Xu Liu. Structslim: a lightweight profiler to guide structure splitting. In *Proceedings of the 2016 International Symposium on Code Generation and Optimization*, CGO '16, page 36–46, New York, NY, USA, 2016. Association for Computing Machinery.

[73] Julian Seward and Nicholas Nethercote. Valgrind: A framework for heavyweight dynamic binary instrumentation. *ACM SIGPLAN Notices*, 42(6):94–105, June 2007.

[74] Julian Seward and Nicholas Nethercote. Cachegrind: a cache and branch profiler. Valgrind User Manual, Chapter 5, 2025.

[75] Han Shen, Krzysztof Pszeniczny, Rahman Lavaee, Snehasish Kumar, Sriraman Tallam, and Xinliang David Li. Propeller: A profile guided, relinking optimizer for warehouse-scale applications. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, ASPLOS 2023, page 617–631, New York, NY, USA, 2023. Association for Computing Machinery.

[76] Ian Smith. Binsub: The simple essence of polymorphic type inference for machine code. In *Static Analysis: 31st International Symposium, SAS 2024, Pasadena, CA, USA, October 20–22, 2024, Proceedings*, page 425–450, Berlin, Heidelberg, 2024. Springer-Verlag.

[77] Julian Squires. Data-type profiling for perf. LWN.net, December 2023. https://lwn.net/Articles/955709/.

[78] Richard Stallman, Roland Pesch, Stan Shebs, et al. Debugging with gdb. *Free Software Foundation*, 675, 1988.

[79] Pengfei Su, Shasha Wen, Hailong Yang, Milind Chabbi, and Xu Liu. Redundant loads: a software inefficiency indicator. In *Proceedings of the 41st International Conference on Software Engineering*, ICSE '19, page 982–993. IEEE Press, 2019.

[80] The Linux Kernel Community. Linux Kernel Release 6.17. https://www.kernel.org/, 2025.

[81] Qingsen Wang, Xu Liu, and Milind Chabbi. Featherlight reuse-distance measurement. In *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 440–453, 2019.

[82] Vincent Weaver. Advanced hardware profiling and sampling (pebs, ibs, etc.): Creating a new papi sampling interface. Technical report, University of Maine, 2015.

[83] Shasha Wen, Milind Chabbi, and Xu Liu. Redspy: Exploring value locality in software. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '17, page 47–61, New York, NY, USA, 2017. Association for Computing Machinery.

[84] Shasha Wen, Xu Liu, John Byrne, and Milind Chabbi. Watching for software inefficiencies with witch. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '18, page 332–347, New York, NY, USA, 2018. Association for Computing Machinery.

[85] Shasha Wen, Xu Liu, and Milind Chabbi. Runtime value numbering: A profiling technique to pinpoint redundant computations. In *Proceedings of the 2015 International Conference on Parallel Architecture and Compilation (PACT)*, PACT '15, page 254–265, USA, 2015. IEEE Computer Society.

[86] Xiaoya Xiang, Chen Ding, Hao Luo, and Bin Bao. Hotl: a higher order theory of locality. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '13, page 343–356, New York, NY, USA, 2013. Association for Computing Machinery.

[87] Danning Xie, Zhuo Zhang, Nan Jiang, Xiangzhe Xu, Lin Tan, and Xiangyu Zhang. Resym: Harnessing llms to recover variable and data structure symbols from stripped binaries. In *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security*, CCS '24, page 4554–4568, New York, NY, USA, 2024. Association for Computing Machinery.

[88] Xin You, Hailong Yang, Zhongzhi Luan, Depei Qian, and Xu Liu. Zerospy: exploring software inefficiency with redundant zeros. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '20. IEEE Press, 2020.

[89] Chao Yu, Probir Roy, Yuebin Bai, Hailong Yang, and Xu Liu. Lwptool: A lightweight profiler to guide data layout optimization. *IEEE Transactions on Parallel and Distributed Systems*, 29(11):2489–2502, 2018.

[90] Junyuan Zeng and Zhiqiang Lin. Towards automatic inference of kernel object semantics from binary code. In *Proceedings of the 18th International Symposium on Research in Attacks, Intrusions, and Defenses - Volume 9404*, RAID 2015, page 538–561, Berlin, Heidelberg, 2015. Springer-Verlag.

[91] Zhuo Zhang, Yapeng Ye, Wei You, Guanhong Tao, Wenchuan Lee, Yonghwi Kwon, Yousra Aafer, and Xiangyu Zhang. Osprey: Recovery of variable and data structure via probabilistic analysis for stripped binary. In *2021 IEEE Symposium on Security and Privacy (SP)*, pages 813–832, 2021.

[92] Qidong Zhao, Xu Liu, and Milind Chabbi. Drcctprof: a fine-grained call path profiler for arm-based clusters. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '20. IEEE Press, 2020.

[93] Jin Zhou, Sam Silvestro, Steven (Jiaxun) Tang, Hanmei Yang, Hongyu Liu, Guangming Zeng, Bo Wu, Cong Liu, and Tongping Liu. Memperf: Profiling allocator-induced performance slowdowns. *Proc. ACM Program. Lang.*, 7(OOPSLA2), October 2023.

All Along the Watchtower: Achieving the Trinity of Observability in Cloud with DiTing

Zhenyu Ren[1], Shuzhi Feng[1], Erci Xu[2]*, Changsheng Niu[1], Haoyu Mao[1], Beibei Wang[1], Chong Gao[1], Zhenshan Zhang[1], Xinrui Yu[1], Jiangwei Huang[1], Jiesheng Wu[1], Hong Tang[1]

[1] Alibaba Group [2] Shanghai Jiao Tong University

Abstract

Observability is crucial for diagnosing and troubleshooting cloud systems as they grow in scale and complexity. However, telemetry data are often stored and processed in siloed systems, leading to high latency, redundant data movement, and low operational efficiency. To address these challenges, we present DiTing, an observability framework that unifies the storage and processing of heterogeneous telemetry data, including logs, metrics, and traces. The key idea of DiTing is to harvest underutilized cloud resources for cost-effective processing while relying on centralized storage systems for reliable data persistence and failover. We have deployed DiTing in production at scale, and DiTing can achieve sub-second data ingestion, high-throughput query processing, and up to 65× lower CapEx than existing solutions.

1 Introduction

Modern cloud computing has been advancing in scale (e.g., globally distributed data centers) [7, 11, 2], heterogeneity (e.g., diverse workloads and new hardware) [19, 37, 31], and disaggregation (i.e., compute-to-storage architectures) [45, 29]. As cloud systems evolve, software bugs [30, 42], performance issues [44, 28], and hardware failures [24, 23, 21] have become more prevalent and harder to diagnose.

Observability via telemetry is crucial for diagnosing and troubleshooting cloud systems. Typical telemetry data include traces [38], logs [20], and metrics [33]. Specifically, traces provide detailed information about execution, logs record events that occur during execution, and metrics summarize quantitative measurements of system behavior.

These three types of telemetry are complementary and often used together for diagnosis. For example, consider a site reliability engineer (SRE) investigating a performance issue such as slow I/Os in a cloud virtual disk. The SRE may first be alerted by abnormal metrics. The SRE then examines end-to-end traces (e.g., from the user's VM to the physical disks) to narrow down the problematic component. Finally, after locating a likely culprit (e.g., a misbehaving switch), the SRE can cross-check device logs to determine the root cause and apply a fix.

*Corresponding author: Erci Xu <jostep90@gmail.com>.

The example highlights the need for multiple perspectives on observability, yet it also reveals a bleak reality. In today's cloud environments, SREs often have to use multiple systems with different query languages and manually correlate data sources with custom scripts for a single investigation. This friction stems from the fact that telemetry datasets are typically stored and processed in separate infrastructures, such as Dapper [38] for traces, LogStore [20] for logs, and Prometheus [33] for metrics. Such siloing not only creates operational barriers for SREs but can also degrade performance due to high latency and excessive data transfer.

It is therefore tempting to host and process different types of telemetry data in a single system. Such a *trinity* of logs, traces, and metrics would eliminate much of the manual effort and allow users to query telemetry with a holistic view of an incident. Inspired by this vision, we explored several paths toward unified observability.

We first unified query processing while leaving data in their original storage systems, for example by adopting a data-lake approach (e.g., Iceberg [5] and Delta Lake [17]). While this provides a unified interface without moving the data, it falls short in performance because the different types of telemetry remain scattered, incurring substantial data-transfer overhead. Second, hyperscalers (e.g., Kraken [26], Monarch [15], and Scuba [14]) have built in-memory NewSQL databases to enable fast queries. Although these systems deliver strong performance by keeping processing in memory, they are impractical for us: the total cost of ownership (TCO) is prohibitively high given the volume of logs and traces that would need to be processed and stored in memory. Finally, we attempted to build our observability framework on an open-source OLAP system, ClickHouse, and managed to deploy it on around 600 physical nodes. However, when scaling to serve the entire cloud, CPU and memory quickly became bottlenecks. Simply provisioning more CPU and memory would increase infrastructure CapEx by more than 20%.

Our initial explorations, though unsuccessful, led to several observations. First, when building a centralized system to unify observability, the main bottleneck is not storage capacity but limited CPU and memory for processing. Second, as a cloud vendor, our server fleet is often underutilized, leaving substantial resources available for harvesting.

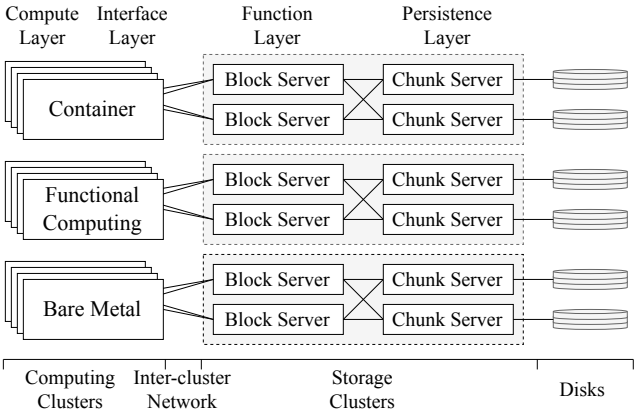


Figure 1: **Overview of our cloud architecture (§2.1).**

However, a fully harvested design is infeasible because end nodes may become unavailable during traffic bursts or failures. Third, telemetry queries exhibit strong temporal locality (i.e., most queries target recently generated data) and spatial locality (i.e., processing queries close to where the data are generated to minimize data transfer).

Based on these lessons, we propose `DiTing`, an observability infrastructure that unifies the storage and processing of heterogeneous telemetry data. `DiTing` is a hybrid query framework that combines AZ-level centralized services with node-level resource harvesting. `DiTing` has three layers: a global layer that serves as the entry point for SREs, an availability-zone (AZ) layer that provides centralized storage and fallback execution, and a node layer that processes queries using harvested resources. On top of this architecture, `DiTing` incorporates a set of optimizations, including multi-level indexing and pre-aggregation.

Our evaluation shows that `DiTing` achieves sub-second data ingestion, 2K QPS using a single CPU core, and 3-65× lower capital expenditure (CapEx) than existing solutions. `DiTing` has been deployed at production scale, serving millions of servers with millions of QPS and storing up to hundreds of PiB of data.

2 Background

2.1 Our Cloud Architecture

Our cloud offers various services such as computing (e.g., virtual machines), storage (e.g., object storage), and SaaS (e.g., MySQL). At a high level, our cloud spans the globe. Each region consists of multiple availability zones (AZs), and each AZ is a separate data center with multiple clusters.

In Figure 1, we highlight the compute-to-storage architecture across clusters. The computing layer consists of physical servers that provide a variety of services, such as containers, serverless functions, and bare-metal instances. The interface layer comprises clients in the compute-node hypervisor which receives users' requests (to remote storage) and

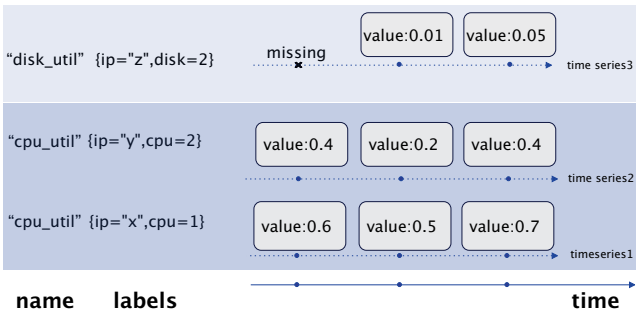


Figure 2: **An example metric record (§2.2).**

packages them into network packets. The function layer consists of servers that process incoming requests and interact with the persistence layer to store or retrieve data. The persistence layer is a distributed storage system. The network between the interface and function layers is inter-cluster, whereas the network between the function and persistence layers is intra-cluster.

2.2 Metrics

Metrics record quantitative measurements of a system, such as health, performance, and behavior. A metric typically includes a name, a set of labels, and one or more values. The name specifies the type of measurement (e.g., CPU utilization), and labels encode contextual information such as location (e.g., CPU #1 on a node with IP address 192.154.123.77). Values form a time series (e.g., 60% at 00:10). Due to their simplicity and broad coverage, metrics often serve as the first-line signal for detecting system anomalies in the field.

Metrics do not consume an exceedingly large amount of storage. As of today, the total compressed size of stored metrics in our cloud is only several PB. However, metrics are often challenging to query. Besides common metrics, each service may export its own attributes, resulting in a high-dimensional space (i.e., many distinct metric types, such as CPU, memory, and NIC utilization). Moreover, users often query across multiple dimensions and over long time ranges (hours to days), which requires scanning and aggregating substantial amounts of data. Therefore, maintaining high queries per second (QPS) at low latency for metric workloads remains challenging.

2.3 Logs

Logging records events across the software, hardware, and network stack. A log entry typically contains a timestamp, a severity level (e.g., INFO or CRITICAL), and a message. In Figure 3, we show an example log entry from a network library. Traditionally, logs are unstructured, but we have recently adopted structured logging (e.g., JSON) to facilitate parsing and querying.

In our cloud, logs are widely used for debugging and trou-

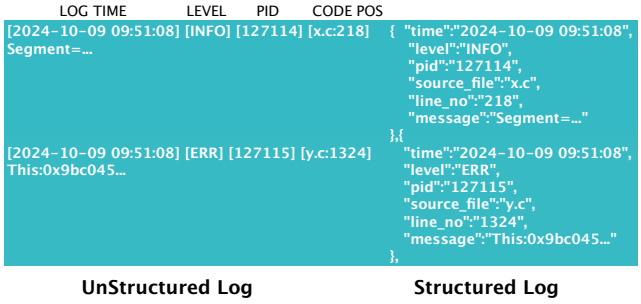


Figure 3: **An example log record (§2.3).**

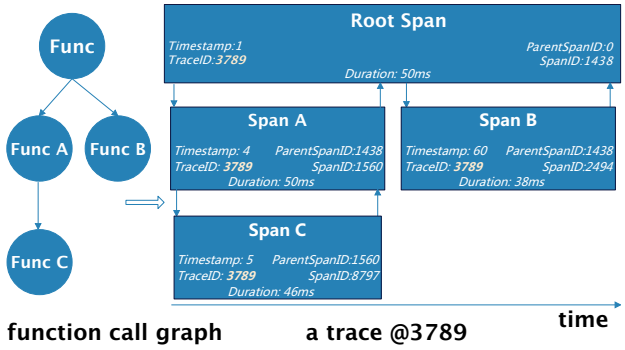


Figure 4: **An example chain of trace records (§2.4).**

bleshooting. In practice, system reliability engineers (SREs) are often first alerted by abnormal metrics and then inspect logs to identify the root cause. Log volume is large: the daily ingestion rate is at the PB scale, and the total stored log data reaches hundreds of PB. Note that most logs are generated by users and thus cannot be discarded.

2.4 Traces

In contrast to point-in-time event records such as logs, tracing provides a fine-grained view of system behavior by recording sequences of events together with timing information. For example, Figure 4 shows a trace of a function call that triggers three sub-procedures. An end-to-end trace consists of multiple spans, each representing a procedure (e.g., Functions A, B, and C).

In our cloud, we primarily use tracing to understand complex call relationships within and across services. Trace data are typically large, with PB-scale daily ingestion, and the total stored volume has reached nearly one hundred PB. To minimize overhead on critical components, we often use static instrumentation (i.e., embedding tracing APIs into source code) to collect traces, making them schema-based. Trace queries usually have low QPS (e.g., tens per second), are range-based (e.g., querying traces within a time window), and focus on short intervals (e.g., up to 30 minutes).

3 Motivation

3.1 A Day in SRE's Life

Maintaining high observability in the cloud is crucial but challenging. Unlike single nodes or small clusters, diagnosing bugs in a geo-distributed service can span multiple layers (e.g., interface and persistence) and involve different clusters (e.g., compute and storage), protocols (e.g., TCP and RDMA), and types of telemetry (e.g., logs, metrics, and traces). Next, we use a recent real-world case for demonstration (with certain details redacted for confidentiality).

Liam is an SRE on our Elastic Block Storage (EBS) team. He receives an alert from the metrics monitoring system indicating a surge of slow requests in a cluster hosting over 4,000 virtual disks. The slowness is worsening and may trigger failures (e.g., I/O hangs), potentially impacting several large customers and thousands of smaller ones.

Liam first double-checks the metrics by logging into the monitoring service (❶ in Figure 5). After confirming severe latency degradation in virtual disks (❷), he scans other metrics (e.g., network throughput and disk IOPS) and finds that cluster-level metrics remain normal, but the top-of-rack switch reports CRC errors (❸). To further narrow down the culprit, he switches to the tracing service to follow the request path along the network topology (❹) and cross-checks logs from each component (❺). Finally, he identifies a BGP down event on a second-layer switch (i.e., PSW), caused by a power failure (❻). He then applies a hotfix to the PSW to prevent escalation (❼).

This incident highlights the complexity of debugging in the cloud. SREs often need to correlate telemetry sources—metrics, logs, and traces—across layers and clusters to confirm symptoms, delineate the blast radius, and identify the root cause. While SREs can develop scripts to assist analysis, these scripts are often ad hoc and hard to reuse across different incidents.

3.2 Why Not Use or Adapt Existing Solutions?

Ideally, SREs would like a unified system that handles both the processing and storage of telemetry data from different sources. With such a system, SREs would no longer need to switch between observability platforms (e.g., Prometheus and Monarch) and correlate results using ad hoc scripts. Moreover, integrating processing with persistence can reduce data-transfer overhead, enabling better query performance and storage efficiency. Next, we discuss why existing solutions are infeasible to port to our cloud. In particular, we describe a failed attempt in which we deployed more than 600 nodes but could not scale further.

Unifying computation only. One intuitive approach is to adopt a data lake solution (e.g., Iceberg [5], Hive [40], Hudi [4], and Doris [3]). This approach keeps different data sources (e.g., logs, traces, and metrics) in their original systems while providing a unified processing interface (e.g., via

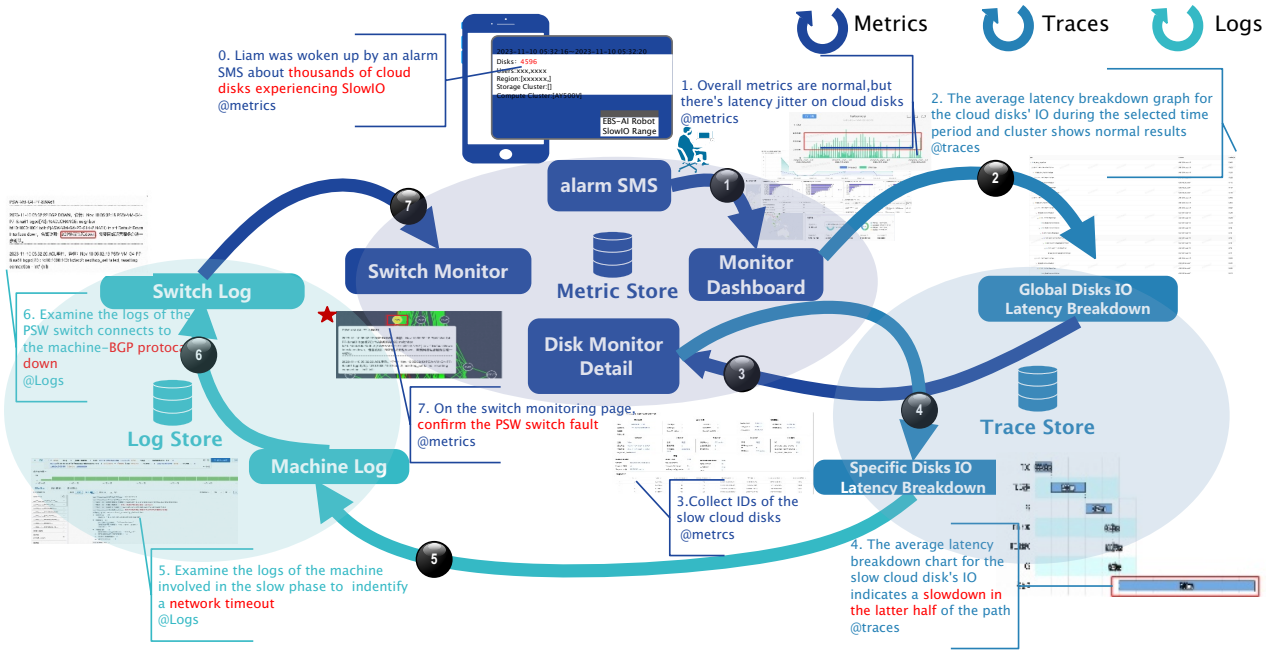


Figure 5: **A Day in SRE's Life.(§3.1).**

Apache Spark). While this is inexpensive to implement because it largely reuses existing ingestion and storage infrastructure, it can lead to high query latency. For example, a typical diagnosis in our cloud may require scanning more than 100 GB of data, which can take tens of seconds (or even minutes) due to network transfers [18]—far slower than the sub-second latency that SREs expect.

Unified storage in memory. For high performance, an alternative is to adopt an in-memory solution, which is common among hyperscalers (e.g., Scuba [14] and Kraken [26] at Meta, and Google's Monarch [15]). These systems are essentially in-memory time-series databases and primarily target metric workloads. On the one hand, they provide fast queries and low ingestion latency (e.g., hundreds of milliseconds in Kraken/Monarch). On the other hand, they are prohibitively expensive at our scale, especially when storing hundreds of PB of telemetry data. Therefore, we do not adopt an in-memory-only design.

ClickHouse: a failed attempt. The above analysis led us to build an OLAP-based solution on top of a set of geo-distributed dedicated clusters, aiming to unify processing and storage with high performance. We chose ClickHouse [35], an open-source OLAP system with strong community support, and started building the service in 2022. At that time, we scaled the deployment to eight clusters across multiple availability zones (AZs), totaling more than 600 nodes (detailed node configuration is listed in §5). The system stored about 8 PB of observability data, including traces and metrics from multiple services.

When attempting to scale ClickHouse further, we encountered several critical challenges. First, ClickHouse bottlenecks when hosting more than one million partitions for a single service [1]; monitoring data from many components across a large fleet can easily exceed this limit. Second, ClickHouse's memory usage grows with the number of columns. Many of our servers produce metric data with thousands of columns, and the per-node memory usage in several clusters reached around 85%, even causing out-of-memory (OOM) events when scanning large amounts of data. These issues indicate that the main constraints are CPU and memory rather than storage capacity. Simply provisioning more CPU and memory is not feasible: our estimates suggest that doing so would account for more than 20% of the cloud infrastructure CapEx.

Alternative-1: a cloud-backed system. One possibility is to use a cloud database (e.g., Snowflake [22] or Amazon Redshift [25]) and adopt a serverless computing model. This disaggregates compute and storage, allowing us to elastically allocate CPU and memory while using object storage as the backend. In principle, this can improve cost efficiency by reducing over-provisioning. However, it introduces a circular dependency (i.e., relying on the cloud to monitor the cloud) and is therefore not feasible in practice.

Alternative-2: harvesting the cloud. Clouds often have abundant idle resources, which makes harvesting appear attractive. In this approach, we push queries to the nodes where observability data are generated and use on-node resources (e.g., CPU and disk) for processing and temporary storage, reducing both data transfer and CapEx. However, end nodes are not always available (e.g., node crashes or in-

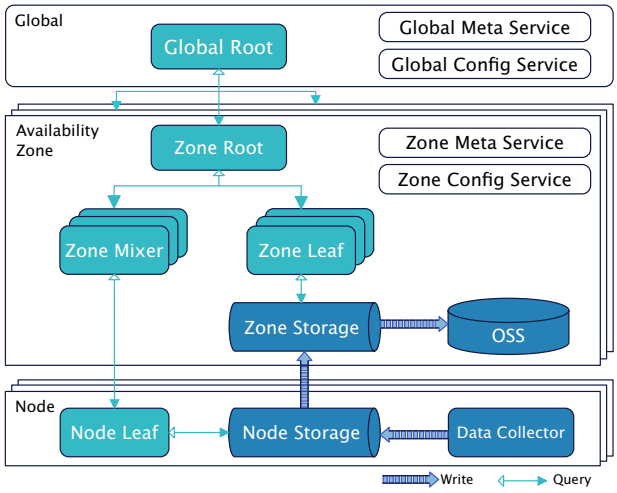


Figure 6: **Overview of `DiTing` (§4.1).**

sufficient resources during traffic bursts). Worse, failures and busy periods are often exactly when SREs need observability the most. Moreover, unlike centralized storage where data can be replicated easily, end nodes are more susceptible to data loss due to disk failures or software-induced corruption.

4 DiTing

Key idea. We now present the key idea behind DiTing, a unified framework for processing and storing telemetry data at cloud scale. Our failed attempt to scale a centralized ClickHouse deployment, together with the limitations of relying solely on idle end-node resources, motivates a hybrid design that combines a centralized system with node-level resource harvesting. We refer to this design as *Central–Node Collaboration* (CNC). After receiving a query, CNC first attempts to push it down to the relevant nodes for in-situ processing. If this fails (e.g., due to node crashes), CNC falls back to the centralized system to execute the query.

Benefits. CNC leverages both spatial and temporal locality in observability workloads. Spatially, it pushes queries to the nodes where the target data are generated. Temporally, SREs most often query recently generated data (e.g., from the last hour or week). CNC therefore keeps recent data on nodes and periodically uploads them to the centralized system for long-term storage. As a result, most queries can be served in situ using harvested resources (e.g., CPU and memory), reducing data transfer overhead and infrastructure CapEx. Meanwhile, the centralized system provides durable storage and serves as a reliable fallback when target nodes are unreachable, improving availability.

4.1 Architecture.

These benefits lead to two design goals: (1) accurate query pushdown and (2) minimizing the use of centralized resources for query processing. To achieve these goals, DiTing adopts a three-layer architecture consisting of the *Global*, *Availability Zone (AZ)*, and *Node* layers, as shown in Figure 6. Note that DiTing serves multiple cloud services (e.g., Elastic Block Service, Object Store Service, and Elastic Compute Service) using the same set of clusters, rather than maintaining service-specific dedicated fleets; this also enables cross-service debugging.

Global. This layer serves as the entry point for queries. Requests that do not specify a target AZ are sent to *Global Root*, which locates the corresponding AZs for execution. To sustain high QPS, Global Root neither processes queries nor stores data; it only forwards requests. Routing relies on the *Global Meta Service*, which maintains mappings (e.g., from a virtual disk ID to the physical nodes and disks that store its data). DiTing also uses the *Global Config Service* to manage configurations such as node-level resource limits and the upload interval for node telemetry to AZ-level centralized storage. To balance load and avoid a single point of failure, we deploy multiple Global Root instances across AZs.

AZ. In this layer, DiTing maintains an AZ-level centralized system, typically consisting of 20–100 dedicated physical machines for query processing and observability-data storage. All nodes share the same configuration. The system usually includes three dedicated zone-root nodes that form a Raft group for high availability. The remaining nodes act as zone-mixer and zone-leaf nodes, which are interchangeable. Upon receiving queries from the global layer, DiTing builds a query tree, partitions the query via zone-root and zone-mixer (i.e., non-leaf) nodes, and pushes sub-queries down to node-level leaf nodes for execution. If pushdown fails (e.g., due to an error or timeout), DiTing falls back to executing the query on an AZ-level leaf node. Periodically, the AZ layer receives observability data uploaded from underlying nodes for long-term persistence. These data are stored on non-root nodes (i.e., zone-mixer and zone-leaf nodes). For higher reliability, DiTing can optionally store an additional copy in the Object Store Service (OSS) within the same AZ. To support different deployment scenarios, the AZ layer manages metadata and configuration through the *AZ Metadata Manager* and the *AZ Configuration Manager*.

Node. Subject to the resource limits enforced by the Global or AZ layer, DiTing harvests idle resources on each node to execute pushed-down queries. Typically, after queries are split and scheduled by the upper layers, sub-queries are pushed down to the node layer for processing. Each node also runs an agent, called the *Data Collector*, which collects observability data including metrics, traces, and logs. DiTing caches recently ingested data, persists it locally on harvested disks, and periodically uploads it to the AZ layer.

4.2 Data Model: co-Log

co-Log format. We now describe DiTing's data format. For metrics, traces, and logs, we use a unified format called

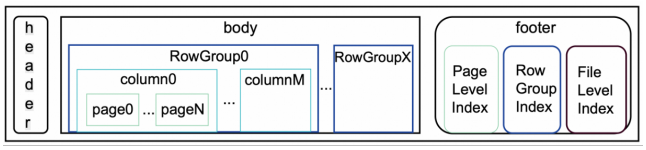


Figure 7: **co-Log data format (§4.2).**

co-Log. To provide a familiar and simple query interface, DiTing supports standard SQL; therefore, telemetry data are organized as relational tables.

- *Metrics.* We adopt a *multi-value* model in which each timestamp is associated with multiple measurements. In a metric table, the first column is the timestamp, and the second column stores labels, i.e., location information (e.g., IP address, hostname, and cluster ID) of the node producing the data. Columns 3 onward store different measurement types (e.g., CPU and memory utilization). A node typically reports 10K–20K metric types. Every 15 seconds, a new row is appended.
- *Traces.* Traces are stored as spans. In a trace table, the first column is the span ID, the second is the timestamp, the third is the parent span ID, the fourth is the trace ID (assigned by the root span), and the last column is the span duration. Each span inserts one row. In Figure 4, there would be four tables corresponding to the root, A, B, and C spans.
- *Logs.* Logs are stored as table entries. Columns include timestamp, location (as in metrics), log severity level, and message body.

Each table is stored as a co-Log file. Similar to Parquet [41] and ORC [6], co-Log uses a PAX (partition-attributes-across) layout: it splits a co-Log file into fixed-size row groups, each containing multiple columns, resulting in a hybrid of row and columnar storage. Each column is further divided into pages to facilitate lookup. To avoid ambiguity in query results [15], we enforce schemas for metrics and traces. For logs, DiTing allows users to specify regular expressions or rules to extract structured fields from unstructured log messages.

We also extend co-Log to store metadata. In DiTing, metadata map telemetry data to their physical locations, enabling accurate query pushdown. co-Log stores metadata in a header and a footer: the header records file size and compression type, and the footer stores indexing information at multiple granularities (file, row group, column, and page).

co-Log features. Beyond standard columnar-storage features, co-Log is tailored to observability workloads. First, SRE queries heavily depend on temporal and spatial attributes. For example, when debugging performance issues, SREs often correlate metrics, traces, and logs within a specific time range and location (see §3.1). Hence, co-Log indexes temporal and spatial fields by default during ingestion.

Second, our dataset exhibits a *wide-table* pattern: while the number of columns can be very large (up to tens of thousands), most queries touch only a small subset (typically around ten columns). Moreover, the hot columns can change over time (e.g., focusing on IOPS and latency during peak events and shifting to failure-related signals during normal periods). To support efficient indexing and adapt to such changes, co-Log stores multi-level indexes (file, row group, column, and page) together in the file footer. For the same reason, schema access is highly skewed and random, with a small set of columns being queried frequently. Unlike Parquet/ORC, we therefore avoid using ProtoBuf [13] or Thrift [39] for metadata serialization; instead, we use a raw format to enable faster access. The larger metadata size is acceptable in our high-speed datacenter networks.

	Feature	P	O	co-Log
1	Mixed row and column storage	✓	✓	✓
2	Multiple compression support	✓	✓	✓
3	Column-based encoding	✓	✓	✓
4	Column-based indexing	✓	✓	✓
5	Temporal and spatial indexing	✗	✗	✓
6	Split data and indexes	✗	✗	✓
7	Random-accessible metadata	✗	✗	✓
8	CRC checking	✓	✗	✓
9	ACID transaction	✗	✓	✗
10	Multi-granularity column compression	✓	✓	✗
11	Complex data types	✓	✓	✗

Table 1: **Comparison between co-Log and parquet/ORC. P refers to parquet and O refers to ORC (§4.2).**

Third, to accommodate frequent hardware changes and software upgrades, co-Log supports schema evolution (e.g., adding new columns) without breaking existing queries. For reliability, we also enforce CRC checking (Table 1, row 8) on both metadata and data to detect silent corruption.

co-Log intentionally omits several features to reduce complexity and overhead. For example, it does not provide strong ACID guarantees (Table 1, row 9), since telemetry workloads can tolerate short-term unavailability and minor inconsistencies. We also do not implement "multi-granularity"column-level compression: given modern high-capacity SSDs, we trade space for query performance. Finally, we do not support complex types (e.g., arrays and maps), which are rarely needed for observability and incur high processing overhead.

4.3 Optimizing Querying

Procedure. We now walk through the query execution flow in DiTing. SREs can either send a query to a global root for cross-AZ lookup or directly to a target AZ. After receiving the query, the zone root in the corresponding AZ first

estimates the load (i.e., how many nodes the query should be pushed to), builds a query tree, and asks zone mixers to identify the specific nodes that hold the target data. DiTing then pushes sub-queries down to these nodes, which execute them using harvested CPU and memory. If pushdown fails (e.g., due to inaccessible nodes or timeouts), DiTing reroutes the query to an AZ-level zone-leaf node for execution. In production, only about 1% of queries fall back to zone-leaf execution. Finally, the zone root aggregates the results and returns them to the user. Next, we highlight key optimizations.

Building the query tree. After receiving a query (either forwarded by the global root or submitted directly by an SRE), DiTing constructs a zone-level query tree consisting of three node types: root, mixer (internal), and leaf. Two key parameters are the tree height and the fan-out. Prior work proposes heuristic policies for choosing these parameters. For example, Monarch [15] uses a fixed height of 3, whereas Kraken [26] dynamically adjusts fan-out to balance parallelism (higher fan-out) against straggler impact (i.e., by lowering fan-out).

A recent study [35] shows that query latency, while largely insensitive to fan-out, scales logarithmically with the number of leaf nodes involved. Based on this insight, DiTing first determines whether a query requires aggregation. If so (e.g., an AZ-level aggregation), and the query needs to be pushed to N nodes, DiTing sets the fan-out to $\min(N, 1000)$. Otherwise (e.g., scan queries), it sets the fan-out to N. This is because, unlike Monarch and Kraken, DiTing must support aggregation-heavy queries that span multiple telemetry types (logs, traces, and metrics) and potentially involve millions of nodes. Such queries can impose substantial aggregation load on mixer nodes; we therefore cap fan-out at 1000.

Query pushdown. Pushing query processing to where the target data reside is a classic approach. Prior systems such as Monarch [15] reuse schema fields (e.g., location) to select target nodes by matching them against query predicates. They also employ Bloom-filter-like indexes (e.g., Monarch's Field Hints Index (FHI) [15]) to speed up target selection.

DiTing does not follow this practice due to precision requirements. If a query is pushed to the wrong nodes (e.g., due to Bloom-filter false positives), the resulting latency penalty and traffic amplification are significant and unacceptable at scale. Instead, DiTing achieves accurate query pushdown by aligning its hierarchy (global, AZ, and node) with the cloud's physical infrastructure (region, AZ, cluster, and node; see §2.1). Each node has an IP address, and its hostname encodes the cluster name. DiTing therefore uses IP and cluster information, together with metadata, to precisely locate target data. For example, hardware components (e.g., NICs and SSDs) have fixed physical locations (e.g., installed on a specific node). For logical entities (e.g., a virtual disk in EBS), DiTing maintains mapping tables (e.g., from a virtual-disk ID to the physical nodes and disks that store its data) and periodically refreshes these mappings from the corresponding services (e.g., EBS block masters).

Query pushdown is not always successful because target nodes may be temporarily unavailable (e.g., due to network partitions or traffic bursts). DiTing therefore provides three fallbacks. First, if a node is deemed unavailable before pushdown (e.g., missing heartbeats), DiTing directly executes the query on AZ-level leaf nodes. Second, if a node becomes unavailable after pushdown, a configurable timeout triggers rerouting to AZ-level leaf nodes. Third, if a node is responsive but overloaded, DiTing switches the node to a fetch-only mode (Figure 6), where the node returns only the latest data and AZ-level leaf nodes complete the computation.

Metadata pushdown. DiTing stores metadata at the global and AZ layers to enable query pushdown. However, node-level execution may still require metadata for joins and filters. For example, a query may join disk traces with disk metadata (e.g., user ID) that would otherwise reside at the global or AZ layer, forcing additional round trips and increasing latency and network overhead.

To avoid this, DiTing slices metadata by node IP and pushes the relevant partitions down to nodes. We first allow users to declare which metadata fields can be associated with nodes. DiTing then combines these metadata with the physical-location table to construct a spatial partitioning scheme, partitions the metadata accordingly, and pushes each partition to the corresponding nodes. This reduces query latency by 33% and lowers node-side CPU utilization from about 45% to 16%, while incurring only a small additional memory overhead (a few MB per node).

Limiting node-level resource usage. DiTing harvests idle node resources for query processing and temporary storage, which requires resource management to avoid interfering with tenant workloads. DiTing therefore enforces user-configurable limits (set at the global and AZ layers) on CPU, memory, and disk usage. To tolerate traffic bursts, users can also specify a short grace period during which resource usage may exceed the thresholds. Severe or sustained limit violations trigger termination of the node-level DiTing agent.

4.4 Optimizing Ingestion

Procedure. All types of observability data—metrics, traces, and logs—are generated on end nodes and formatted as co-Log files. Node-level agents insert metric entries by computing the corresponding statistics every 15 seconds (e.g., average or maximum). Trace and log entries are appended upon function calls and log events. Periodically, node-level agents merge co-Log files and upload them to the AZ layer for persistence. Optionally, the AZ layer can also send a copy of the uploaded data to the object store service for higher reliability.

Node-level in-memory buffer. User access exhibits strong temporal locality: most queries target recently generated

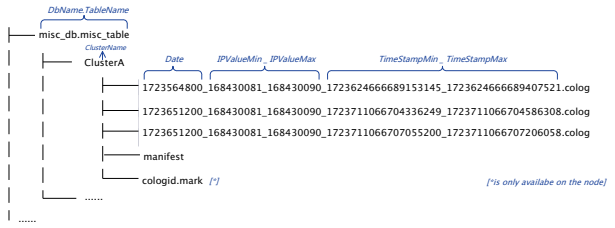


Figure 8: **co-Log file layout (§4.4).**

data. We therefore implement a fixed-size, node-level in-memory buffer to cache the latest data, prioritizing low latency and low cost. AZ-level leaf nodes do not maintain such a buffer because temporal/spatial locality are weaker there.

Surprisingly, for a server that produces 20K distinct metrics (an extreme case), a 900 MiB per-node buffer achieves cache hits for over 99.9% of queries. Even on nodes with fewer idle resources, we find that a 100 MiB buffer is sufficient to reach a hit ratio of around 90%. This effectively provides an in-memory-database experience for most queries.

Pre-aggregation for metrics. Beyond short-term monitoring, SREs often query metrics over long time ranges (e.g., weeks or months) to understand trends. Returning raw 15-second-granularity data would incur high storage and processing overhead while providing limited additional insight. For example, querying a VM's CPU utilization over the past 30 days at 15-second granularity yields 172,800 points, placing a heavy burden on node resources and network traffic. `DiTing` therefore supports pre-aggregation for metrics, computing aggregates such as average, sum, max, and min at coarser granularities (e.g., 1 minute, 5 minutes, and 1 hour). This optimization applies only to metrics.

File merge. In `DiTing`, because end nodes periodically upload co-Log files to the AZ layer (every 1 minute by default), the AZ layer must provide durable storage for a massive number of co-Log files (e.g., billions). This creates a substantial maintenance burden. However, we cannot simply merge all files into large ones, because doing so would erase spatial information (i.e., metadata used to enable accurate query pushdown). `DiTing` therefore employs a two-step merging strategy. First, we merge row groups (Figure 8) from files produced by the same server, and attach the server IP to each row group's metadata. Next, we merge row groups from servers in the same cluster into a single co-Log file, sort the row groups by IP address, and append cluster information and an IP-to-zone map to the file footer. This design also guides the folder structure in Figure 8: the directory structure is flat, and the file layout and naming conventions are explicit and easy to manage.

4.5 Consistency, Integrity, and Availability

We intentionally do not build `DiTing` as a database because supporting transactions (i.e., strong ACID guarantees) is too costly and unnecessary for telemetry workloads. Nevertheless, `DiTing` employs several mechanisms to improve consistency, integrity, and availability.

Consistency. We make several observations. First, most observability data are append-only and produced by a single writer (i.e., the per-node agent). Second, data are replicated by being uploaded from the node layer to the AZ layer and then backed up to the object store. Third, SREs do not directly query nodes: if pushdown fails, `DiTing` reroutes the query to AZ-level leaf nodes. This avoids consistency issues caused by multiple clients reading from and writing to different replicas. Finally, in observability scenarios, results from node-level and AZ-level execution may differ slightly due to freshness. `DiTing` estimates this potential staleness from the query results and notifies users, who can optionally re-run the query against the centralized copy.

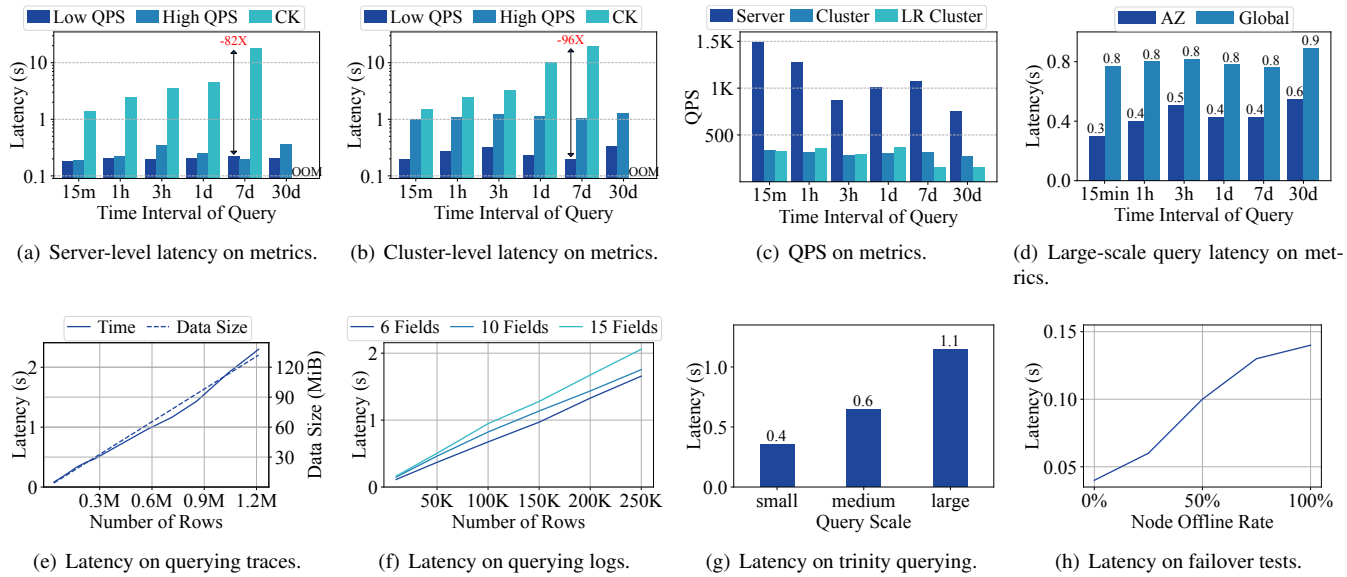
Based on these observations, `DiTing` adopts a simplified Raft-like protocol for each replicated unit spanning node storage, AZ storage, and object-store storage. For example, with 1K nodes, we form 1K independent Raft groups. The key idea is to always treat the node as the leader (i.e., no leader election as in standard Raft): because each node is the sole writer, if it goes down, no new telemetry data will arrive for that group. SREs can still query replicas in the AZ layer or object store. This design substantially simplifies engineering and operations, which is critical at `DiTing`'s scale.

Integrity. `DiTing` uses CRC to verify data integrity from generation, through network transmission, to persistence on disk. We learned two lessons in practice. First, we still need to periodically scrub data and cross-check against the object store to detect corruption. At hundreds of PB (or more), bit flips are inevitable. Second, when computing CRCs over nested data blocks, we leverage the compositional properties of CRC (i.e., combining precomputed CRCs for sub-blocks) to avoid repeated reads and recomputation. This reduces overhead and helps detect corruption that might otherwise be missed due to inconsistent results across repeated reads.

Availability. The two replicas in `DiTing` (node-level and AZ-level) already provide approximately nine nines of availability based on our failure-rate analysis. With an additional backup in the object store, `DiTing` can provide roughly twelve nines. Note that the object store is an independent service rather than a third `DiTing` replica tier. It is also more cost-effective than other storage options (e.g., EBS and cloud databases), which helps meet our CapEx requirements.

5 Evaluation

We organize our evaluation into three parts. First, we evaluate `DiTing` on a microbenchmark, focusing on query latency and queries per second (QPS). Second, we present deployment statistics from `DiTing`'s large-scale production rollout. Finally, we compare capital expenditure (CapEx) against existing solutions.



(a) Server-level latency on metrics. (b) Cluster-level latency on metrics. (c) QPS on metrics. (d) Large-scale query latency on metrics.

(e) Latency on querying traces. (f) Latency on querying logs. (g) Latency on trinity querying. (h) Latency on failover tests.

Figure 9: **The performance of `DiTing` on various circumstances. (§5.1)**

5.1 Microbenchmark

We evaluate `DiTing` using a microbenchmark that issues queries over metrics, logs, traces, a mixture of all three, and metadata. We additionally run failover experiments to assess `DiTing`'s robustness under node unavailability.

Metrics. We construct the metrics microbenchmark as follows. First, we use real-world production metrics from an 80-server cluster in our object store service. For each query, we randomly select 12 fields from a table with more than 1,000 fields. This matches field-access statistics in production: on average, a query touches 11.2 fields, and no more than 1% of all fields are accessed by any single query.

Second, we query metrics over a range of time windows: the most recent 5 minutes, 1 hour, 3 hours, 1 day, 7 days, and 30 days. This reflects common SRE workflows, where recent data are used for real-time monitoring and historical data are used for trend analysis. For long-range queries, `DiTing` serves data pre-aggregated at 5-minute or 1-hour granularity.

Third, we set the benchmark concurrency to 400, simulating 400 SREs continuously issuing queries to the cluster. We compare against ClickHouse, deployed on an 18-node cluster where each node has 96 CPU cores, 256 GB RAM, 12×14 TB QLC SSDs, and 2×1 TB PMEM devices. In contrast, the AZ-level `DiTing` deployment uses three nodes, each with 64 CPU cores, 128 GB RAM, and 12×3.5 TB QLC SSDs.

Figure 9(a) reports query latency and QPS for `DiTing` and a centralized-only variant of `DiTing` when targeting a single server. For 5-minute, 1-hour, and 3-hour windows, metrics are not pre-aggregated. At around 1K QPS, `DiTing` achieves about 0.2 s latency, whereas the centralized-only variant is 7–82× slower. The gap widens for longer time ranges (e.g., 0.2 s vs. 18.1 s) because `DiTing` can pre-aggregate metrics on nodes and thus reduce processing overhead. Somewhat surprisingly, the centralized-only variant may even fail to return results for 30-day queries due to out-of-memory (OOM) errors. Note that we do not enable pre-aggregation in the centralized-only variant because it would introduce additional CPU overhead and write latency.

		15min	1h	3h	1d	7d
Latency (s)	In	1.01	1.09	1.22	1.14	1.05
	Out	9.54	9.68	9.66	9.07	9.71
QPS	In	333	313	279	304	320
	Out	16.0	17.2	17.5	19.3	16.0

Table 2: **Comparison between queries in and out of cache.**

When extending the target from a single IP to an entire cluster, Figure 9(b) shows consistent results, although latency may increase under high QPS due to higher network traffic. Moreover, Figure 9(c) shows that `DiTing` reaches about 1,500 QPS while using only one CPU core in our microbenchmark. Even on low-resource machines with a 300 MB memory limit, `DiTing` can achieve similar QPS in many cases.

Table 2 further compares `DiTing` with and without the in-memory cache under a concurrency of 400. The results highlight the importance of caching recent metric data in memory, since metrics require high-QPS reads. Without caching, leaf nodes cannot sustain such high-rate disk reads, leading to substantial performance degradation and potentially affecting system stability.

We further scale metric queries to the region and global

levels. Specifically, we randomly select 10 fields from the same table but target (i) one region of the object store service (∼20K nodes) and (ii) multiple regions (∼60K nodes). Figure 9(d) shows that even at these scales, latency increases only modestly to about 0.5 s and 0.8 s, respectively. This demonstrates the effectiveness of query pushdown.

Traces. Trace workloads typically have fewer columns but many more rows (with millisecond-level granularity). To evaluate trace queries, we use production trace data from an Elastic Block Service (EBS) server and issue queries that read 10 columns sampled from different column ranges. As shown in Figure 9(e), latency increases linearly with the amount of data read, indicating that `DiTing` scales well for trace workloads.

Logs. Compared to metrics and traces, logs have much larger volume. Due to their unstructured nature, logs are often inspected manually and thus have a lower query rate. Recall that we adopt schema-on-read for logs, converting them to the unified format only at query time; we therefore focus on the throughput of reading and filtering.

We use production log data and apply a 30% selectivity filter, meaning that out of every 100 lines read, 30 lines are returned. We query 6, 10, and 15 fields over log datasets ranging from 10K to 250K rows. Figure 9(f) shows that processing time grows linearly with the amount of data scanned.

Trinity. A key feature of `DiTing` is the ability to query and correlate metrics, traces, and logs in a single request. We construct a trinity microbenchmark that follows production field-access patterns and includes three scales (small, medium, and large):

- **Metric table:** ∼500 columns, one row every 15 s. Small (last 5 min), medium (last 30 min), large (last 1 h).
- **Trace table:** ∼30 columns, ∼2K rows every 15 s. Small (65K rows), medium (210K rows), large (460K rows).
- **Log table:** ∼20 columns, ∼25 rows every 15 s. Small (4.5K rows), medium (5K rows), large (5.5K rows).

Under this setup, each query first retrieves server CPU usage, then joins with a trace table for a cloud component, and finally joins with the component's error logs on the time field. Figure 9(g) shows that latency increases with the amount of queried data; most of the time is spent retrieving trace data due to its larger volume.

Failover. Node-level leaf nodes may be unavailable during failures or traffic bursts. To evaluate robustness, we select a 200-node cluster and vary the fraction of unavailable node-level leaf nodes from 0% to 100% in a single-threaded setup. We query metrics over the most recent 1-hour window. Figure 9(h) shows that latency increases gradually as node unavailability increases.

Metadata. Metadata in `DiTing` map logical entities and fields to physical locations. To evaluate metadata lookup, we query all virtual disks (VDs) subscribed by a major customer.

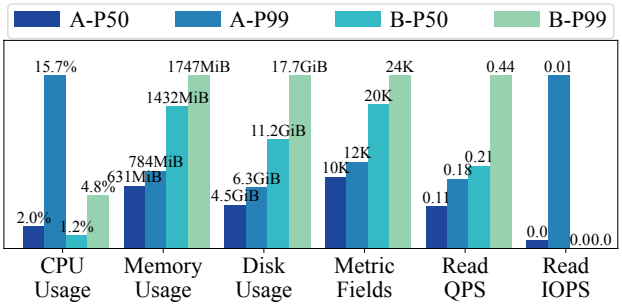
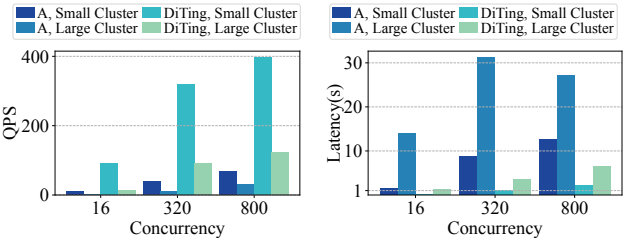


Figure 10: **Deployment statistics of `DiTing`.**



(a) Cluster-level qps on metrics. (b) Cluster-level latency on metrics.

Figure 11: **The performance of `DiTing` on production environments, compared with solution A. (§5.2)**

These VDs (about 430K) are distributed across regions and AZs. We then query the total traffic for these VDs. Without metadata, the query would need to scan the entire dataset (∼1.4 PiB) to locate the disks, taking tens of minutes. With metadata-assisted pushdown, the query completes in 2 s.

5.2 Deployment Statistics

At the time of writing, we have deployed `DiTing` for multiple services (e.g., OSS and EBS) on more than one million nodes across multiple regions worldwide. Next, we report deployment statistics for two services in a single region. Service A emits a typical metrics workload (about 10K metrics per node), whereas Service B emits a larger workload (about 20K metrics per node). The statistics are collected from more than 300 clusters and 36K nodes. The AZ-level monitoring data stored in this region reach about 600 TB. Our AZ-level nodes serve about 1,200 QPS on average, and node-level leaf agents collectively serve about 40K QPS. Figure 10 summarizes node-level overhead (CPU, memory, and disk consumed by `DiTing`) and per-node QPS, reported at the 50th and 99th percentiles.

Overall, the node-level resource overhead of `DiTing` remains low. On average, `DiTing` consumes GB-scale memory per node (about 0.5% of total DRAM), uses fewer than 2% of CPU cores, and requires only 10–20 GB of local disk for short-term persistence. Even for Service B, our implementation occupies only about 0.55% of memory and 7% of the system disk on average. However, when amortized over

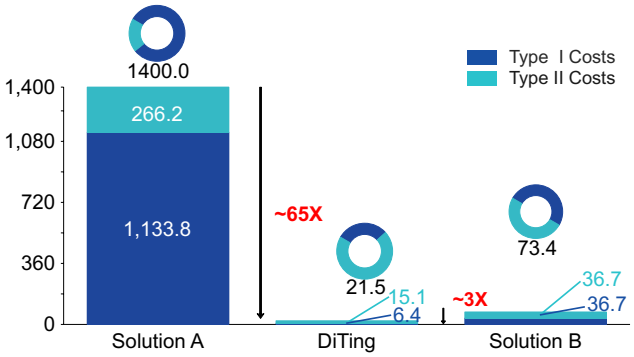


Figure 12: **CapEx comparison of `DiTing`, Solution A, and Solution B.**

a large fleet (36K nodes across Services A and B), the aggregate memory footprint reaches about 34 TB. This large distributed cache is a key reason for `DiTing`'s low query latency, resembling the behavior of a distributed in-memory database. At the global scale, `DiTing` runs on millions of nodes and stores tens to hundreds of PiB of compressed data.

Figure 11 further compares `DiTing` with Solution A in production. Solution A is a classic time-series high-performance DB developed by our company (serving a wide range of users for over a decade). We run experiments on a small deployment (50 nodes) and a large deployment (400 nodes), varying concurrency from 16 to 800. Each experiment queries a single metric over the most recent 1.5 hours. As shown in the figure, `DiTing` improves QPS by 4–9× while reducing latency to 1/10–1/4 of Solution A.

5.3 CapEx Analysis

We compare the CapEx of `DiTing` against two baselines, anonymized as Solution A and Solution B. Solution A corresponds to the system used in the deployment study above, and Solution B is our internal distributed logging service with EB-scale capacity. Both represent state-of-the-art production systems and provide Prometheus-like functionality. We choose internal services as baselines because we can obtain detailed CapEx numbers that have been strictly audited by our finance department. Due to confidentiality constraints, we cannot disclose absolute costs. Instead, we report normalized comparisons to highlight `DiTing`'s CapEx reduction in the field.

We categorize CapEx into two types, following common accounting practice: Type I (*persistence*) and Type II (*ingestion*). Type I captures long-term storage costs. `DiTing` primarily relies on the Object Store Service (OSS), built on top of our internal distributed file system. Both Solution A and Solution B use the same underlying file system directly. Thus, the three systems have similar cost per stored GB. Even so, `DiTing` incurs only about 1/180 and 1/6 of the Type I cost compared to Solutions A and B, respectively. This reduction comes from `DiTing`'s multi-value data model and the co-Log format (see §4.2), which efficiently stores and queries high-cardinality, high-dimensional metrics.

Type II captures ingestion cost and can be decomposed into three factors: (1) *Physical cost*, including hardware depreciation, power, and datacenter (IDC) expenses; (2) *Selling rate*, defined as the fraction of provisioned resources that are actually subscribed/serving; and (3) *Other overheads*, including software licensing and additional infrastructure fees. We summarize this as:

$$\text{TYPE-II COST} = \left(\frac{\text{PHYSICAL COST}}{\text{SELLING RATE}}\right) \times (1 + \text{OTHER}).$$

Note that `DiTing` does not attribute Type II cost to harvested node resources, since those nodes are already paid for by the services. Hence, `DiTing` mainly incurs Type II cost in the centralized (AZ-level) cluster. Overall, `DiTing` reduces Type II cost by about 17.6× and 2.4× compared to Solutions A and B, respectively. In total, `DiTing` achieves about 65× and 3× CapEx reduction relative to Solutions A and B.

6 Lessons Learned

Unifying is not equal to identical handling. In hindsight, "unified" does not mean "handled identically." Although `DiTing` uses one storage format (co-Log) and one processing interface to cover metrics, traces, and logs, each data type still needs its own engineering focus.

For traces, the biggest operational lesson is that the SDK matters as much as the backend. If the instrumentation is unstable or too heavyweight, the system fails exactly when it is needed most: abnormal paths are often rare, time sensitive, and the first to be dropped under pressure. For metrics, high cardinality and long-range queries are the primary cost driver. We repeatedly hit cases where adding a few high-cardinality dimensions (or expanding a monitoring scope) silently multiplied storage footprint and query cost. Our experience suggests the only sustainable approach is to continuously watch growth signals and proactively apply summarization (e.g., pre-aggregation) and retention controls. For logs, we learned to optimize for lossless and fast ingestion first. Heavy preprocessing (parsing, extraction, normalization) is best treated as a best-effort or asynchronous step. Keeping the write path simple makes the system far more robust under bursty incidents.

Stability comes first. `DiTing` relies on a per-node agent for data collection, local persistence, pushdown execution, and metadata handling. This consolidation improves locality and reduces CapEx, but it also enlarges the blast radius. A regression in the agent can affect an entire fleet within hours.

Operationally, the most effective measures were not sophisticated algorithms, but disciplined rollout and guardrails. For example, we implemented canary releases with clear success criteria, staged expansion across clusters/regions, and automated rollback when key SLOs regress. We also found it necessary to bake "self-protection" into the agent

and the control plane—rate limits, circuit breakers, backpressure, and load shedding—because at cloud scale, rare corner cases happen daily, if not hourly. Finally, we intentionally keep the consistency design simple (our Raft-like variant with a fixed leader) so that failure behavior is always predictable and easy to operate.

CapEx! CapEx! CapEx! Our early prototypes focused on making a centralized `DiTing` work for more services and more data types, because that was the fastest way to validate functionality. As deployment scaled, we learned the hard way that a centralized architecture has a steep cost curve for telemetry: ingestion grows continuously, and a design that looks "reasonable per node" can become unsustainable once multiplied by millions of machines.

Two practical lessons stood out. First, cost regressions often come from small, local changes: adding an index, increasing a sampling rate, enabling a new dimension, or extending retention by a few days. Each change is easy to justify in isolation, but the fleet-wide effect can be enormous. Second, the only way to keep costs under control is to make them visible and reviewable. In our deployment process, every major feature must come with an explicit cost model (e.g., added bytes per node per day, incremental write amplification, and extra CPU for ingestion/query), and we routinely run "what-if" analyses during design reviews.

This is also why the final architecture embraces Central–Node Collaboration: we use harvested resources to flatten the steady-state cost curve, while keeping the AZ-level centralized clusters as a bounded, reliable fallback. In practice, treating *cost to serve* as a continuous optimization target—alongside latency and availability—was essential to making `DiTing` viable at cloud scale.

7 Related Work

TSDB. Time-series databases (TSDBs), such as OpenTSDB [12], BlueFlood [32], KairosDB [10], Graphite [8], InfluxDB [9], and Prometheus [33], are designed for storing and querying time-series data and are therefore widely used as building blocks for observability. However, traditional TSDBs primarily optimize for time-partitioned access and often provide limited support for *spatial* attributes (e.g., region/AZ/cluster/node) that are fundamental in cloud telemetry. At large scale, this mismatch can lead to skewed placements and I/O hotspots (e.g., many operators concurrently scanning recent windows for the same AZ or service). Recent in-memory TSDBs, such as Scuba [14], Kraken [26], and Monarch [15], demonstrate that real-time querying can scale to very large datasets with low latency, but their in-memory designs make them expensive to operate at the hundreds-of-PB scale.

OLAP. Online analytical processing (OLAP) systems, such as Druid [43], ClickHouse [35], and Napa [16], provide fast interactive analytics and are widely adopted in practice. Their columnar storage and vectorized execution are a good fit for many observability queries (e.g., scans with filters and aggregations). However, in cloud observability deployments, OLAP systems can become cost-inefficient due to high CPU and memory consumption, especially for wide tables, high-cardinality dimensions, and massive partition counts. Moreover, many existing deployments target PB-scale datasets, whereas cloud-scale observability can accumulate data at the EB scale, making the "scale-up more clusters" approach financially and operationally challenging.

SQL Engines. Modern SQL engines, such as Impala [27], Presto [36], and F1 Query [34], commonly adopt a storage–compute disaggregated architecture and provide full SQL support over large datasets. This model is attractive for flexibility and ecosystem integration. However, complex joins and global aggregations typically require runtime data reshuffling (e.g., repartitioning/shuffles), which can introduce substantial network overhead and sensitivity to stragglers. These characteristics make it difficult to achieve predictable low-latency, high-QPS performance for global observability, where queries frequently target "recent windows" and must remain responsive during incidents.

Cloud-native Data Warehouses. Cloud data warehouses, such as Snowflake [22] and Redshift [25], move the warehouse stack to managed cloud infrastructure to gain elasticity and simplified operations. To mitigate write throughput and query latency challenges, these systems often incorporate intermediate caching/buffering layers and rely on object storage as the durable backend. Nevertheless, they may incur additional data redundancy (e.g., multiple copies for performance) and provide limited flexibility for real-time observability analysis under heavy ingestion. Moreover, using a managed cloud warehouse as the foundation of observability introduces a circular dependency: the cloud service is relied upon to monitor itself, complicating failure diagnosis.

8 Conclusion

The complexity of modern cloud environments calls for a unified observability infrastructure. `DiTing` leverages underutilized resources and a layered architecture to streamline telemetry processing and storage. Beyond improving performance and reducing cost, `DiTing` also enhances reliability and scalability for cloud-scale observability.

Acknowledgments

We sincerely thank the OSDI'25 reviewers for their constructive feedback, which significantly improved both the technical content and the presentation. We also thank the OSDI'26 organizers for their help and support throughout the process. In addition, we are grateful to the EBS, ECS, and OSS teams for their long-term support in deploying and running `DiTing` at scale. Finally, we acknowledge the support of the Alibaba Research Fellow (ARF) program.

References

[1] How many maximum databases, tables, partitions, or parts are recommended in a ClickHouse cluster? https://clickhouse.com/docs/knowledgebase/maximum_number_of_tables_and_databases, 2019.

[2] Alibaba cloud. https://www.alibabacloud.com/, 2024.

[3] Apache Doris. https://hudi.apache.org/, 2024.

[4] Apache Hudi. https://hudi.apache.org/, 2024.

[5] Apache Iceberg. https://iceberg.apache.org/, 2024.

[6] Apache Orc. https://orc.apache.org/, 2024.

[7] Google cloud. https://cloud.google.com/, 2024.

[8] Graphite. https://graphiteapp.org/, 2024.

[9] Influxdata: Influxdb time series data platform. https://www.influxdata.com/, 2024.

[10] Kairosdb. https://kairosdb.github.io/, 2024.

[11] Microsoft azure. https://azure.microsoft.com/, 2024.

[12] Opentsdb. https://opentsdb.net/, 2024.

[13] Protocol Buffers - Google's data interchange format. https://protobuf.dev/, 2024.

[14] Lior Abraham, John Allen, Oleksandr Barykin, Vinayak Borkar, Bhuwan Chopra, Ciprian Gerea, Daniel Merl, Josh Metzler, David Reiss, Subbu Subramanian, et al. Scuba: Diving into data at facebook. Proceedings of the VLDB Endowment, 6(11):1057–1067, 2013.

[15] Colin Adams, Luis Alonso, Benjamin Atkin, John Banning, Sumeer Bhola, Rick Buskens, Ming Chen, Xi Chen, Yoo Chung, Qin Jia, et al. Monarch: Google's planet-scale in-memory time series database. Proceedings of the VLDB Endowment, 13(12):3181–3194, 2020.

[16] Ankur Agiwal, Kevin Lai, Gokul Nath Babu Manoharan, Indrajit Roy, Jagan Sankaranarayanan, Hao Zhang, Tao Zou, Min Chen, Zongchang Chen, Ming Dai, et al. Napa: Powering scalable data warehousing with robust query performance at google. Proceedings of the VLDB Endowment, 14(12):2986–2997, 2021.

[17] Michael Armbrust, Tathagata Das, Liwen Sun, Burak Yavuz, Shixiong Zhu, Mukul Murthy, Joseph Torres, Herman van Hovell, Adrian Ionescu, Alicja Łuszczak, et al. Delta lake: high-performance acid table storage over cloud object stores. Proceedings of the VLDB Endowment, 13(12):3411–3424, 2020.

[18] Haoqiong Bian and Anastasia Ailamaki. Pixels: An efficient column store for cloud data lakes. In 2022 IEEE 38th International Conference on Data Engineering (ICDE), pages 3078–3090. IEEE, 2022.

[19] Matias Bjørling, Abutalib Aghayev, Hans Holmberg, Aravind Ramesh, Damien Le Moal, Gregory R Ganger, and George Amvrosiadis. {ZNS}: Avoiding the block interface tax for flash-based {SSDs}. In 2021 USENIX Annual Technical Conference (USENIX ATC 21), pages 689–703, 2021.

[20] Wei Cao, Xiaojie Feng, Boyuan Liang, Tianyu Zhang, Yusong Gao, Yunyang Zhang, and Feifei Li. Logstore: A cloud-native and multi-tenant log database. In Proceedings of the 2021 International Conference on Management of Data, pages 2464–2476, 2021.

[21] Brian Choi, Randal Burns, and Peng Huang. Understanding and dealing with hard faults in persistent memory systems. In Proceedings of the 16th European Conference on Computer Systems, EuroSys '21, April 2021.

[22] Benoit Dageville, Thierry Cruanes, Marcin Zukowski, Vadim Antonov, Artin Avanes, Jon Bock, Jonathan Claybaugh, Daniel Engovatov, Martin Hentschel, Jiansheng Huang, et al. The snowflake elastic data warehouse. In Proceedings of the 2016 International Conference on Management of Data, pages 215–226, 2016.

[23] Haryadi S Gunawi, Mingzhe Hao, Riza O Suminto, Agung Laksono, Anang D Satria, Jeffry Adityatama, and Kurnia J Eliazar. Why does the cloud stop computing? lessons from hundreds of service outages. In Proceedings of the Seventh ACM Symposium on Cloud Computing, pages 1–16, 2016.

[24] Haryadi S Gunawi, Riza O Suminto, Russell Sears, Casey Golliher, Swaminathan Sundararaman, Xing Lin, Tim Emami, Weiguang Sheng, Nematollah Bidokhti, Caitie McCaffrey, et al. Fail-slow at scale: Evidence of hardware performance faults in large production systems. ACM Transactions on Storage (TOS), 14(3):1–26, 2018.

[25] Anurag Gupta, Deepak Agarwal, Derek Tan, Jakub Kulesza, Rahul Pathak, Stefano Stefani, and Vidhya Srinivasan. Amazon redshift and the case for simpler data warehouses. In Proceedings of the 2015 ACM SIGMOD international conference on management of data, pages 1917–1923, 2015.

[26] Stavros Harizopoulos, Taylor Hopper, Morton Mo, Shyam Sundar Chandrasekaran, Tongguang Chen, Yan Cui, Nandini Ganesh, Gary Helmling, Hieu Pham, and Sebastian Wong. Meta's next-generation realtime monitoring and analytics platform. Proceedings of the VLDB Endowment, 15(12):3522–3534, 2022.

[27] Marcel Kornacker, Alexander Behm, Victor Bittorf, Taras Bobrovytsky, Casey Ching, Alan Choi, Justin Erickson, Martin Grund, Daniel Hecht, Matthew Jacobs, et al. Impala: A modern, open-source sql engine for hadoop. In Cidr, volume 1, page 9. Asilomar, CA, 2015.

[28] Chi Li, Shu Wang, Henry Hoffmann, and Shan Lu. Statically inferring performance properties of software configurations. In Proceedings of the Fifteenth European Conference on Computer Systems, pages 1–16, 2020.

[29] Feifei Li. Modernization of databases in the cloud era: Building databases that run like legos. Proceedings of the VLDB Endowment, 16(12):4140–4151, 2023.

[30] Chang Lou, Peng Huang, and Scott Smith. Understanding, detecting and localizing partial failures in large system software. In Proceedings of the 17th USENIX Symposium on Networked Systems Design and Implementation, NSDI '20, Santa Clara, CA, February 2020. USENIX.

[31] Jiaqi Lou, Xinhao Kong, Jinghan Huang, Wei Bai, Nam Sung Kim, and Danyang Zhuo. Harmonic: Hardware-assisted {RDMA} performance isolation for public clouds. In 21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24), pages 1479–1496, 2024.

[32] Beshr Al Nahas, Antonio Escobar-Molero, Jirka Klaue, Simon Duquennoy, and Olaf Landsiedel. Blueflood: Concurrent transmissions for multi-hop bluetooth 5—modeling and evaluation. ACM Transactions on Internet of Things, 2(4):1–30, 2021.

[33] Björn Rabenstein and Julius Volz. Prometheus: A Next-Generation monitoring system (talk). Dublin, May 2015. USENIX Association.

[34] Bart Samwel, John Cieslewicz, Ben Handy, Jason Govig, Petros Venetis, Chanjun Yang, Keith Peters, Jeff Shute, Daniel Tenedorio, Himani Apte, et al. F1 query: Declarative querying at scale. Proceedings of the VLDB Endowment, 11(12):1835–1848, 2018.

[35] Robert Schulze, Tom Schreiber, Ilya Yatsishin, Ryadh Dahimene, and Alexey Milovidov. Clickhouse-lightning fast analytics for everyone. Proceedings of the VLDB Endowment, 17(12):3731–3744, 2024.

[36] Raghav Sethi, Martin Traverso, Dain Sundstrom, David Phillips, Wenlei Xie, Yutian Sun, Nezih Yegitbasi, Haozhun Jin, Eric Hwang, Nileema Shingte, et al. Presto: Sql on everything. In 2019 IEEE 35th International Conference on Data Engineering (ICDE), pages 1802–1813. IEEE, 2019.

[37] Nimish Shah, Laura Isabel Galindez Olascoaga, Shirui Zhao, Wannes Meert, and Marian Verhelst. Dpu: Dag processing unit for irregular graphs with precision-scalable posit arithmetic in 28 nm. IEEE Journal of Solid-State Circuits, 57(8):2586–2596, 2021.

[38] Benjamin H. Sigelman, Luiz André Barroso, Mike Burrows, Pat Stephenson, Manoj Plakal, Donald Beaver, Saul Jaspan, and Chandan Shanbhag. Dapper, a large-scale distributed systems tracing infrastructure. Technical report, Google, Inc., 2010.

[39] Mark Slee, Aditya Agarwal, and Marc Kwiatkowski. Thrift: Scalable cross-language services implementation. Facebook white paper, 5(8):127, 2007.

[40] Ashish Thusoo, Joydeep Sen Sarma, Namit Jain, Zheng Shao, Prasad Chakka, Suresh Anthony, Hao Liu, Pete Wyckoff, and Raghotham Murthy. Hive: a warehousing solution over a map-reduce framework. Proceedings of the VLDB Endowment, 2(2):1626–1629, 2009.

[41] Deepak Vohra and Deepak Vohra. Apache parquet. Practical Hadoop Ecosystem: A Definitive Guide to Hadoop-Related Frameworks and Tools, pages 325–335, 2016.

[42] Haoze Wu, Jia Pan, and Peng Huang. Efficient exposure of partial failure bugs in distributed systems with inferred abstract states. In Proceedings of the 21st USENIX Symposium on Networked Systems Design and Implementation, NSDI '24, April 2024.

[43] Fangjin Yang, Eric Tschetter, Xavier Léauté, Nelson Ray, Gian Merlino, and Deep Ganguli. Druid: A real-time analytical data store. In Proceedings of the 2014 ACM SIGMOD international conference on Management of data, pages 157–168, 2014.

[44] Junwen Yang, Cong Yan, Chengcheng Wan, Shan Lu, and Alvin Cheung. View-centric performance optimization for database-backed web applications. In 2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE), pages 994–1004. IEEE, 2019.

[45] Weidong Zhang, Erci Xu, Qiuping Wang, Xiaolu Zhang, Yuesheng Gu, Zhenwei Lu, Tao Ouyang, Guanqun Dai, Wenwen Peng, Zhe Xu, et al. What's the story in {EBS} glory: Evolutions and lessons in building cloud block store. In 22nd USENIX Conference on File and Storage Technologies (FAST 24), pages 277–291, 2024.

jwmalloc: A Verified Memory Allocator for Mobile Devices

Jiawei Wang[1], Ming Fu[1,✉], Ruixian Wang[1], Chao Xu[1], Jonas Oberhauser[1,2], Haibo Chen[1,3]

[1]*Huawei Central Software Institute* [2]*Huawei Hilbert Research Center*
[3]*Shanghai Jiao Tong University*

Abstract

Dynamic memory allocators provide memory allocation and release functionality to languages such as C, C++, and Rust. Existing allocators have been primarily optimized for performance and timely memory release. However, emerging mobile workloads also emphasize CPU, energy, and memory usage under soft real-time constraints.

We present jwmalloc, a novel memory allocator built from the ground up to address these mobile-specific challenges. It is based on three key innovations: a *uniform slab size with pooling* that enables immediate cross-size-class reuse of any emptied slab; the *closed sibling tree*, a new data structure for efficiently managing fragments; and a *two-buffer lifetime tracker* that categorizes short- and long-lived objects to guide memory reclamation. For timely response in soft real-time and oversubscription scenarios, all operations are non-blocking. We verified jwmalloc with a bounded model checker under weak memory models. Replacing jemalloc on a flagship smartphone running real-world workloads, jwmalloc reduces whole-system instructions by 10% and cuts allocator-side instruction counts by 3.84×, while lowering CPU power consumption by 5–11%, all at a comparable memory footprint. jwmalloc has been deployed in production on 12 million commercial mobile devices, operating stably for over 30 billion user hours.

1 Introduction

Dynamic memory allocation is a cornerstone of modern operating systems, with a direct and profound impact on overall performance, from warehouse-scale datacenters to personal mobile devices [40–44, 63, 65, 66, 68–71, 76, 86]. For example, recent Google studies report that allocator activity consumes roughly 7% of all CPU cycles in datacenters, noting that *"even a single percent CPU or memory improvement translates to significant savings in server costs"* [53, 87]. On mobile devices, the stakes are even higher: a few gigabytes of DRAM are intensely shared among dozens of applications and system services, making memory a constantly scarce resource [9, 15, 17]. Each component must minimize reserved memory, or the system resorts to aggressive paging and background app termination, visibly degrading user experience [26, 49]. Under this pressure, allocation activity is enormous: allocator instructions account for 8.2% and 12.4% of all executed instructions in real-world mobile workloads on Android and HarmonyOS, respectively (Fig. 1).

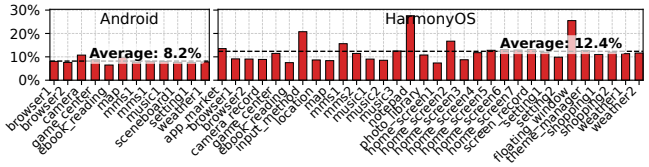


Figure 1: Fraction of CPU instructions spent in dynamic memory allocation with jemalloc under real-world mobile workloads on Android [3] and HarmonyOS [22].

Additionally, mobile workloads have unique challenges, of which we identify four: to avoid reserving new memory, allocations must frequently reuse memory that had been formatted for the prior object size, causing reformatting ❶; foreground and background phases [2], together with bursty user interactions and screen-off idle periods, induce large peak–trough memory swings ❷; short-lived objects [52] causing fragmentation during aggressive memory reclamation ❸; and heavy oversubscription ❹ on a few cores creating potentially high latency with traditional lock-based synchronization [36, 46].

An effective mobile allocator must account for these challenges while achieving the following objectives: (i) *low CPU and energy overhead* to avoid stealing cycles from application logic or triggering thermal throttling [82]; (ii) *bounded memory footprint* to ensure the fluidity of foreground apps and the residency of background processes [23]; (iii) *low tail latency* to prevent visible stuttering on high-refresh-rate displays [39]; and (iv) *correctness guarantees*, ensuring the allocator neither causes crashes nor corrupts memory.

However, despite widespread use of mobile devices [30],

✉ Ming Fu (ming.fu@huawei.com) is the corresponding author.

existing allocator designs do not adequately address these mobile-specific challenges, compromising some of these objectives. Allocators typically employ a two-level architecture: a small-object *slab* frontend [45] and a backend that manages virtual-memory ranges obtained from the OS at page granularity (hereafter, *ranges*). We analyze several allocators, including jemalloc [47], a *performance-oriented* allocator (Sec. 2.2) widely adopted on mobile devices, as well as other mature industrial allocators such as tcmalloc [67] and mimalloc [64], and identify several systematic mismatches.

First, existing frontends mishandle the trade-off between CPU cost and memory footprint for slabs. For example, jemalloc uses heterogeneous slab sizes (i.e., different slab sizes for different *size classes*) to reduce per-slab tail memory waste. However, this incurs high CPU overhead, as the backend must repeatedly subdivide and coalesce ranges to satisfy slabs of varying sizes when demand shifts across size classes (❶).

Second, existing backends contain *intermediate ranges* whose sizes do not match any slab size or large-object size class. These ranges cannot directly satisfy allocation requests but must first be subdivided or coalesced into usably-sized ranges, thereby increasing management and usage costs. In addition, the metadata used to manage ranges is effectively grow-only; under large peak–trough memory swings (❷), it is sized to the historical peak, forcing the allocator to carry substantial dead metadata even at steady state.

Third, existing memory reclamation policies (i.e., returning unused pages to the OS) are not lifetime-aware, leading to suboptimal decisions during aggressive memory reclamation (❸): neighbors of short-lived ranges may be reclaimed too early, missing opportunities to merge with soon-to-be-freed adjacent ranges, incurring extra overhead (e.g., system calls), while neighbors of long-lived ranges may be held too long, leaving memory unreclaimed with little benefit.

Finally, existing allocator data structures are commonly protected by coarse-grained locks, which leads to noticeable latency spikes under heavy oversubscription (❹).

This paper introduces jwmalloc, a new allocator designed from the ground up for mobile devices that addresses these core challenges. Our contributions are as follows:

(i) **Uniform and Pooled Slab Frontend** (Sec. 3.1): All slabs have the same size, allowing immediate reformatting for any frontend size class. We pool empty slabs in the frontend, skipping expensive coalescing of heterogeneous slabs and greatly reducing CPU during frequent slab reformatting (❶).

(ii) **Size-Class-Exact Range Backend** (Sec. 3.2): The set of maintained range sizes in the backend exactly matches the size-class set, eliminating inefficient intermediate ranges. This is realized with a *closed sibling tree*, a generalization of the buddy tree [54,75] in which each node can have more than two children, while maintaining the property that any contiguous subset of siblings can coalesce into a valid size class (hence "closed"). Additionally, we employ a size-friendly range subdivision and coalescing algorithm that tends to preserve large ranges, along with a metadata scheme that grows and shrinks with in-use ranges, keeping metadata overhead low during peak–trough memory swings (❷).

(iii) **Lifetime-Based Reclamation** (Sec. 3.3): jwmalloc employs *lifetime-based reclamation* that allows making good decisions while aggressively reclaiming memory (❸). This is realized with a *two-buffer lifetime tracker* emulating old and young generations in garbage collectors [52]. Short- and long-lived ranges are predicted efficiently without costly per-range timestamps or range scans. Memory neighboring long-lived ranges is reclaimed first, giving short-lived ranges time to be freed and coalesced with their neighbors.

(iv) **Non-Blocking Interface** (Sec. 3.4): jwmalloc's interface is designed to be non-blocking, avoiding high tail latency for operations under heavy thread oversubscription (❹). This is achieved by converting rare but potentially catastrophic blocking into brief, far less harmful memory waste.

(v) **Verification under WMMs** (Sec. 5): We employed bounded formal verification of jwmalloc's functional correctness (e.g., no memory corruption) and general properties (e.g., its memory safety, absence of data races, loop termination) under weak memory models (WMMs) [6,60] using the VSync toolchain [72].

We implemented jwmalloc as a drop-in replacement for state-of-the-art allocators and evaluated it extensively (Sec. 6). On microbenchmarks, jwmalloc improves performance by 74% over jemalloc on average and reduces allocator-side instruction counts by 82%. On a flagship smartphone running real-world workloads, jwmalloc keeps memory usage comparable to that of jemalloc, while reducing overall system instructions by 10% and cutting allocator-side instruction counts by 3.84×. In two representative workloads, it lowers CPU power consumption by roughly 5–11% across different core clusters. Finally, jwmalloc has been deployed in production and is in large-scale commercial use on 12 million mobile devices, including smartphones, tablets, and smartwatches, and has operated stably for more than 30 billion user hours.

2 Background

2.1 Dynamic Memory Allocators

Many modern memory allocators are organized into a *frontend* and a *backend* with distinct responsibilities [47, 64, 67]. The frontend is optimized for fast allocation and deallocation of small *objects*. It typically employs a *slab*-style allocator [45] and maintains per-thread structures to reduce contention in multithreaded workloads. The backend handles large allocations, manages the frontend's memory supply, interacts with the operating system via page-granularity interfaces, and implements caching and reclamation policies for free pages (i.e., returning them to the OS). Allocators maintain a set of *size classes*: predefined standardized allocation

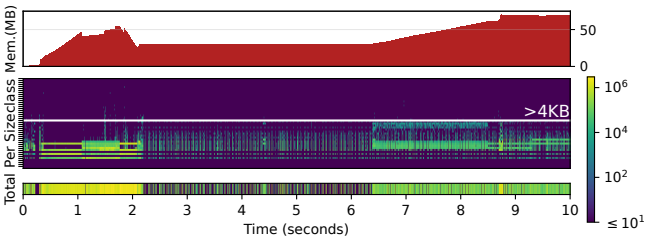


Figure 2: Allocation statistics for the system graphics service of a smartphone over a 10s interval. The first ∼2s correspond to service startup, and the last ∼3s to screen unlock and scrolling. Top: time series of allocator-managed memory in use. Middle: time–size heatmap of per-size-class allocation/free rates. Visually, size classes increase from 8B at the bottom to 16MB at the top; the white horizontal line marks 4KB, and each time slice is 1ms. Bottom: time–size heatmap of total allocation/free rates.

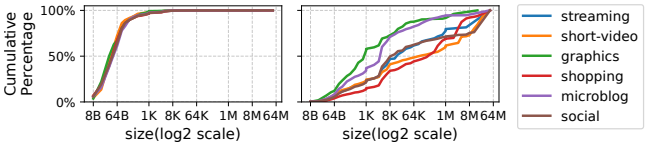


Figure 3: Cumulative distributions by allocation size for services and applications. Left: allocation **operations**. Right: allocated **bytes**.

sizes (e.g., 16B, 32B, 64B, 128B, ...) to which requests are rounded up to reduce fragmentation.

Consider a 100B allocation request. The slab frontend rounds this up to the corresponding size class (e.g., 128B) and serves it from a slab of that class (e.g., an 8KB slab). If no such slab is available, the frontend requests an 8KB range from the backend. If no such free range exists in the backend's cached ranges, the backend instead obtains a larger cached range (e.g., 24KB), carves out 8KB for the slab, and leaves the remaining 16KB available to serve other requests. When all objects in a slab are freed, the slab is returned to the backend as a free range; if its adjacent range (e.g., the 16KB remainder) is also free, the backend coalesces them into a larger range.

2.2 Performance-Oriented Allocators

There is a long-standing trade-off between performance and memory safety in memory allocator design. Performance-oriented allocators such as jemalloc, tcmalloc, and mimalloc prioritize performance over memory safety, making them susceptible to memory-safety issues like use-after-free [7] when the memory user contains bugs. In response, hardened allocators [24, 29, 76] like Scudo and PartitionAlloc emerged, introducing mitigations such as checksums and quarantines. However, these hardening mechanisms come at a significant performance cost; for instance, Bionic's tests show jemalloc

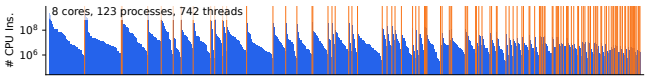


Figure 4: Statistics of processes and threads on a smartphone during the app_market benchmark. Each full-height vertical line (spanning the entire plot) marks a process boundary; within each region, the threads of that process are shown as vertical bars and ordered by executed CPU instruction count in descending order.

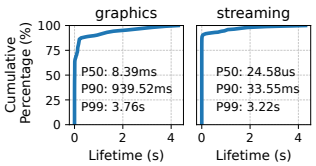


Figure 5: Cumulative percentage of page-granularity allocation lifetimes for the system graphics service and the streaming app.

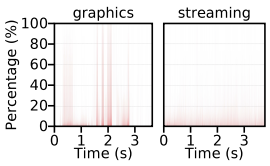


Figure 6: Proportion of cross-thread frees in the system graphics service and the streaming app over a ∼3s interval.

performing up to 4× faster than Scudo [25]. Consequently, the most expensive hardening features, such as Scudo's quarantine, are often disabled by default [29].

This landscape is now being reshaped by emerging technologies that address memory safety at the language and hardware levels. Newly written Android platform code is increasingly written in Rust as the next-generation solution for memory safety [28], and Google reports a 1000× reduction in memory-safety vulnerability density in Android's Rust code compared to C/C++ [38]. In the meantime, ARM's Memory Tagging Extension (MTE) [5] provides hardware-level memory safety checking with only ∼2% performance overhead [37].

Given these observations, as well as our own production-deployment experience (e.g., using large-scale hardware-assisted address sanitizer (HWASAN) deployments [12, 13] to detect memory-safety issues), this paper focuses on the design of a performance-oriented allocator and compares it against other performance-oriented allocators. How to integrate our design with existing memory-safety mechanisms is orthogonal to this work and left out of scope.

2.3 Mobile Allocation Workloads

We profile dynamic allocations across several high-frequency smartphone workloads. Due to space constraints, we show a subset of services and apps; the remaining workloads behave similarly. In general, allocation rates are high in all scenarios, especially during bursty interactions. For example, the bottom panel of Fig. 2 reports that the *system graphics service* (graphics) [10] reaches a peak allocation rate of roughly 3 million ops/s across all threads. Below, we summarize our

detailed observations.

❶ **Frequent memory reformatting.** The per-size-class panel (Fig. 2, middle) shows that activity is dominated by different size classes at different times, while total in-use memory (Fig. 2, top) remains roughly stable. Combined with the size distribution, where allocation counts are strongly skewed toward small sizes (Fig. 3, left) but the fraction of allocated *bytes* is much flatter (Fig. 3, right), this indicates frequent memory reformatting over time.

❷ **Large peak–trough memory swings.** Mobile applications switch between foreground and background phases [2], with bursty user interactions and screen-off idle periods [34]. Consequently, memory usage often exhibits large peak–trough swings. For example, in our measurements of graphics, peak memory usage exceeds 5× the steady-state usage.

❸ **Aggressive Reclamation.** Memory reclamation delays reach 5 to 15 seconds in server loads [19], but are often configured to 1 second or less in mobile [4]. Such short reclamation delays prevent page swapping, but may increase load if ranges are hastily reclaimed before they can be coalesced with neighboring ranges. Only memory near ranges that will anyways not be freed for a long time should be reclaimed immediately. The classical generational hypothesis [80] states that objects that survive for some time are likely to be long-lived. We observe the same in mobile workloads, indicating that a page with old objects is unlikely to be freed in the near future. Fig. 5 reports page-granularity allocation lifetimes. Most pages are freed shortly after allocation: in streaming, 90% of pages become unused within 33.55ms. Beyond this knee point, a small fraction remains allocated for much longer; for example, over 1% of streaming pages live longer than 3.22s.

❹ **High Oversubscription.** Fig. 4 illustrates an app-market installation scenario in which up to 742 threads are time-sliced on an 8-core device. In practice, big.LITTLE asymmetry [59] and power management can further reduce the number of effective cores [46], leading to high thread oversubscription and long delays between timeslices of the same thread. If the thread holds a lock, the delay is propagated to all threads contending on the lock. In existing allocators, this mostly affects *cross-thread frees*, where an object is allocated by one thread and freed by another. The free and further allocations on the same range contend on a lock protecting the range's metadata. Unfortunately, cross-thread frees are frequent in mobile workloads. Fig. 6 reports, for each 1ms time slice, the fraction of cross-thread frees. In the streaming app, most slices remain below 20% cross-thread frees, though short bursts can reach up to 80%. In mobile-typical producer-consumer architectures like graphics [11, 83] and event-driven architectures [48, 79], some slices approach nearly 100% cross-thread deallocation.

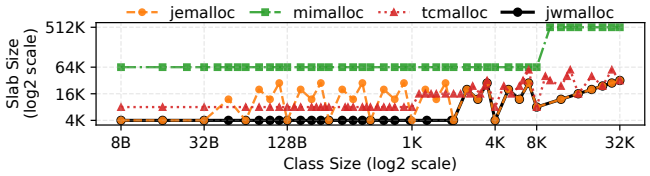


Figure 7: Slab sizes selected for each object size class up to 32KB (default allocator configuration; 4KB OS pages).

3 Our Approaches

3.1 Uniform and Pooled Slab Frontend

Slab-based allocators pack many equal-sized objects into each slab, creating a trade-off: too few objects make in-slab allocations infrequent and reduce performance, while *tail fragmentation* grows due to unused space at the page-aligned slab end. Conversely, any in-use object pins its slab, preventing it from being freed; having many objects per slab amplifies partial-slab persistence, defying reclamation (❸) and thereby inflating both peak and steady-state memory footprints in workloads with frequent memory reformatting and large peak–trough swings (❶ and ❷).

However, this trade-off is often ignored or mishandled by existing allocators. Fig. 7 shows each allocator's *slab size* per size class. mimalloc uses only two large slabs (64KB and 512KB) for simplicity and performance, allowing many objects per slab (e.g., 8192 for an 8B object) and negligible tail fragmentation, but at the cost of a high footprint on mobile workloads (Sec. 6). tcmalloc balances slab sizes to avoid excessive pinning or low performance. jemalloc aggressively reduces footprint by setting a tail-fragmentation threshold and selecting, for each size class, the smallest OS-page-aligned slab size that meets it, yielding heterogeneous slab sizes. This reduces tail waste but increases backend churn when demand shifts across sizes (❶), as more frequent subdivision and coalescing are required, raising CPU overhead (Sec. 6).

To maintain a memory footprint comparable to jemalloc while avoiding its drawbacks, we introduce two changes: (i) *Uniform slab size*: instead of fixing size classes and adjusting slab sizes for each to reduce tail fragmentation as in jemalloc, we fix the slab size and adjust the size classes to keep tail fragmentation below a target cap. In our setting, we use a uniform slab size of one OS page for all size classes of at most half a page. For example, for a 4KB slab containing 128B of metadata, a 512B size class leaves 384B wasted at the slab tail (`(4096-128) mod 512 = 384`), whereas tuning the size class to 560B reduces tail waste to 48B (`(4096-128) mod 560 = 48`). (ii) *Slab pooling*: freed slabs of this uniform size are cached as per-thread plain memory and can later be reinitialized as slabs for any frontend size class. Such pooling is impractical with heterogeneous slab sizes due to memory-footprint constraints.

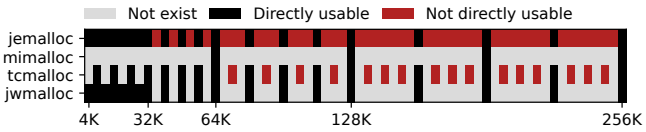


Figure 8: Ranges managed by each allocator. Subdivided ranges are shown in non-gray; directly usable ranges appear in black, intermediate ranges in red.

3.2 Size-Class-Exact Range Backend

3.2.1 Eager Subdivision and On-demand Coalescing

Allocator backends subdivide and coalesce memory ranges to supply slabs and large objects. mimalloc's backend is slab-style: each range is divided into equal-size subranges serving only a single size class. Memory of a range can be reformatted for another size class only once all its subranges are freed. By contrast, tcmalloc and jemalloc manage memory at finer granularity (8KB in tcmalloc; 4KB in jemalloc), allowing subranges to be reformatted independently. For example, a 40KB range can first serve a 16KB object and a 24KB object, and, after the latter is freed, its memory can be immediately reassigned to an 8KB object.

However, their range subdivision method introduces additional CPU overhead. On an allocation demand, they first search for a range of the requested size; if none exists, they select a larger range, split off the required portion, and manage the remainder for further requests. This process produces many intermediate ranges at multiples of the granularity that often do not match any size class (Fig. 8), which increases management and lookup costs. For example, tcmalloc maintains hundreds of size-indexed entries for such ranges. To serve a 224KB request, it probes sizes in ascending order: 224KB, 232KB, 240KB, and so on, leading to higher overhead.

Moreover, they use an eager range coalescing strategy that immediately merges adjacent free ranges, which can be inefficient if the merge produces an intermediate range. For instance, freeing a 224KB range that merges with an adjacent 24KB range yields 248KB. If no size class exists between 224KB and 256KB, the resulting 248KB range cannot be used directly; a subsequent 224KB request will subdivide it again, creating churn and wasting CPU cycles.

To address this, we adopt *size-class-exact ranges* and the following approach: (i) *Eager subdivision*: each range can be subdivided into multiple subranges, one matching the requested size class and the remaining subranges each matching some size class; (ii) *On-demand coalescing*: ranges are coalesced only when the merged size matches a size class. Consequently, the maintained range sizes exactly match the size-class set, greatly reducing these costs.

Table 1: Examples of resolution-R size-class sets for $R \in \{1,2,3\}$ with 8B object alignment. For each R, we list the first eight size classes along with their binary representations.

R	First eight size classes (bytes)							
1	8B 1000_2	16B 10000_2	32B 100000_2	64B 1000000_2	128B $100\ldots0_2$	256B $100\ldots0_2$	512B $100\ldots0_2$	1024B $100\ldots0_2$
2	8B 1000_2	16B 10000_2	24B 11000_2	32B 100000_2	48B 110000_2	64B 1000000_2	96B 1100000_2	128B 10000000_2
3	8B 1000_2	16B 10000_2	24B 11000_2	32B 100000_2	40B 101000_2	48B 110000_2	56B 111000_2	64B 1000000_2

3.2.2 Closed Sibling Tree

The above policy requires a range-management scheme that maintains these size-class-exact ranges. Modern allocators carefully choose size classes to balance internal fragmentation and range reuse: if the size-class set is too sparse, rounding up causes large internal fragmentation (e.g., 9KB to 16KB); if it is too dense, allocations cannot effectively reuse freed ranges from nearby sizes (e.g., a 9KB request rounded up to 12KB cannot satisfy a subsequent 14KB request, but rounding it up to 16KB would work), which is particularly important under frequent memory reformatting (❶). To balance this trade-off, allocators usually use a fixed *resolution-R size-class set*, which bounds the worst-case roundup-induced internal fragmentation rate to at most $1/2^{R-1}$. An allocation request of size n rounds up to the smallest size class $s \geq n$ in which only the R most significant bits of s's binary representation may be non-zero. For example, for $n = 21$ and $R = 3$, 10101_2 is rounded to 11000_2, i.e., $s = 24$. Table 1 shows the first eight size classes for $R \in \{1,2,3\}$ with 8-byte object alignment, along with their binary representations. Most modern allocators [47, 64, 67] use $R = 3$, which corresponds to a worst-case roundup-induced internal fragmentation of 25%.

However, existing range-management schemes cannot maintain a set of range sizes that exactly matches a resolution-R size-class set for arbitrary R. mimalloc's backend manages only two range sizes (64KB and 512KB), while jemalloc's and tcmalloc's backends operate at fixed granularities of 4KB and 8KB, respectively. Classical buddy allocators [54, 75] manage ranges whose sizes are powers of two (2^k), which corresponds to a resolution-1 size-class set. Weighted buddy [78], dual buddy [74], and tertiary buddy [85] extend this by also supporting sizes of the form 3×2^k, and thus can realize a resolution-2 size-class set. Other buddy variants, such as Fibonacci buddy [51, 55], produce range sizes that only partially overlap a given resolution-R size-class set.

We propose the *closed sibling tree*, whose range set can realize any resolution-R size-class set. Each node of the tree represents a range whose size is a size-class size. Subdivision creates two or more child nodes that partition the parent range; each child again has a size-class size. Any contiguous sequence of siblings coalesces into a range whose size is also a size-class size. A formal definition appears in Sec. 4.2.2. In practice, we use the tree to manage a resolution-3 size-class

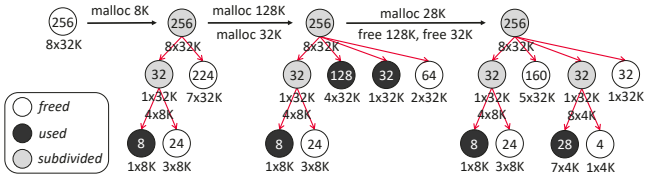


Figure 9: Example of a closed sibling tree on a 256KB range.

set, consistent with modern allocators. We also provide efficient subdivision and coalescing algorithms on this tree. For eager subdivision, we use a size-friendly scheme that tends to preserve large ranges. On-demand coalescing is straightforward: we coalesce adjacent siblings under the same parent.

As an example in Fig. 9, consider a closed sibling tree whose root represents a 256KB range. An 8KB allocation would split it into 8KB and 248KB, but 248KB is not a size-class size. Instead, our algorithm (Sec. 4.2) first carves out a 32KB child, leaving a 224KB child that *is* a size-class size, then subdivides the 32KB child to obtain the requested 8KB, with the remaining 24KB also being a size-class size. Subsequent 128KB and 32KB allocations are served by repeatedly subdividing the 224KB child, eventually leaving a 64KB child; all of these ranges are siblings under the 256KB parent. Later, a 28KB allocation cannot simply split the 64KB child into 28KB and 36KB, because 36KB is not a size-class size. Instead, the algorithm first splits 64KB into two 32KB children and then subdivides one 32KB child into 28KB and 4KB, both of which are size-class sizes. When the 128KB and 32KB children are subsequently freed, these two adjacent siblings coalesce into a single 160KB node, which is a size-class size.

3.2.3 Per-Granule Metadata with Metadata Shifting

Allocator backends face a challenge in balancing metadata lookup efficiency with memory overhead. mimalloc leverages a large range granularity (i.e., 64KB) to employ *per-granularity metadata*, which enables fast address-to-metadata lookups via direct array indexing with minimal memory cost. In contrast, finer-granularity allocators such as jemalloc and tcmalloc cannot afford the overhead of per-granularity metadata and instead rely on *per-range metadata* managed by more complex structures like radix trees. This approach, however, suffers from a major limitation beyond its performance drawbacks: to support multi-threaded performance, metadata records are reused across ranges but are never freed. Consequently, during large peak–trough memory swings (❷), metadata capacity grows with peak usage but fails to shrink at steady state, leading to substantial memory bloat.

To overcome this limitation, we compress the tree's metadata significantly, enabling it to afford fast, per-granularity metadata (12B per 4KB granularity) that is fully reclaimable once its managed ranges are all freed. We adopt a novel *metadata shifting* approach, making the tree entirely pointer-free even with irregular node sizes: the conceptual two-dimensional tree is mapped to a one-dimensional array; internal nodes are encoded into leaf nodes to eliminate parent pointers; and movement between siblings at the leaf level is based on address offsets between siblings, computed from range sizes and tree information (e.g., depth, sibling index) rather than explicit pointers. For example, in Fig. 9, a 256KB range contains 64 metadata nodes (256KB/ 4KB). When the 32KB node shown is freed, its left and right siblings have sibling indices 1 and 6, while the current node's index is 5 (first sibling = 0). Each index corresponds to a distance of 8 nodes (32KB/ 4KB), so the left and right siblings are at offsets of (1 - 5) $\times$ 8 and (6 - 5) $\times$ 8 relative to the current node.

3.3 Lifetime-Based Reclamation

When memory ranges are freed, the allocator caches them temporarily to avoid repeated kernel allocation and deallocation, then eventually releases them to the OS if they are not reused. tcmalloc uses an *asynchronous*, *rate-based reclamation* policy: a background reclamation thread returns cached ranges to the OS at a rate that is tuned either by the user or based on recent allocation activity. However, it lacks timing guarantees for when these ranges will be reclaimed, which is undesirable for mobile workloads: during periods of high allocation and free rates, often coinciding with system-wide high load, tcmalloc may defer reclamation even while other components are competing for memory (Sec. 6.1.2).

mimalloc uses a *synchronous*, *time-based reclamation* policy: each freed range is cached for a fixed interval, after which it is returned to the OS. This policy interacts poorly with the foreground/background behavior [2] of mobile applications. Once an app moves to the background and its allocation and free rates drop sharply, reclamation is rarely triggered, allowing cached ranges to remain resident for a long time (Sec. 6.1.2). Moreover, it requires periodically scanning ranges to check for expiry, wasting CPU cycles.

jemalloc uses an asynchronous, time-based reclamation policy: it divides time into epochs and, at the start of each epoch, computes how much cached memory to reclaim and spreads this work across the epoch. If cached memory exceeds a threshold, it may run extra reclamation before the next epoch. This policy has been battle-tested on mobile devices, but it still has drawbacks. Although it imposes an approximate upper bound on reclamation time, it offers no lower bound. A range may be reclaimed shortly after being freed, even when its adjacent range is about to be freed as well. Waiting slightly longer could allow them to coalesce into a larger range that can satisfy future large requests, potentially avoiding an extra system call if these ranges are eventually returned to the OS.

However, blindly delaying reclamation only increases the memory footprint. Instead, we ask: given a fixed reclamation budget, which ranges should we reclaim first and which should we defer to maximize coalescing? Mobile workloads

follow the generational hypothesis: once a range survives past the knee point, it is likely to remain live much longer, so we prefer to reclaim its adjacent range first. For example, consider five contiguous subranges (1–5). Suppose subranges 1, 3, and 5 are cached, subrange 2 is long-lived, and subrange 4 is short-lived; thus, we prefer to reclaim subrange 1 first.

To this end, we propose a *lifetime-based reclamation* policy. A range starts a countdown once it has no mergeable siblings. If, before the timer expires, one of its siblings is freed and coalesces with it, the countdown is reset; otherwise, the range becomes eligible for reclamation.

A naive implementation of this policy would mirror mimalloc's approach and require scanning all cached ranges, which is expensive. Instead, we implement it using a *two-buffer lifetime tracker*, consisting of an *active buffer* and a *standby buffer*. Freed and coalesced ranges are always placed in the active buffer. Allocations first consult the active buffer; if empty, they consult the standby buffer. A reclamation thread repeatedly performs the following steps: (i) swap the roles of the two buffers, (ii) wait for an interval Δ, and (iii) reclaim all ranges in the standby buffer. This avoids per-range timestamps and full-range scans, significantly reducing CPU load.

Under normal conditions, reclamation is handled asynchronously by the reclamation thread. However, under heavy oversubscription (❹), this thread may not be scheduled promptly, as we observed. To address this, we combine asynchronous and synchronous reclamation: when the number of cached ranges exceeds a watermark, newly freed ranges are immediately reclaimed by the freeing thread, and we trigger an early buffer swapping by waking the reclamation thread, thereby reducing Δ. This mixed reclamation scheme continues until the number of cached ranges drops below the watermark.

3.4 Non-blocking Interface

Allocators need to synchronize between threads accessing the same metadata, for example, during cross-thread frees. In mimalloc, this is synchronized via a concurrent bitmap that controls uniform-sized slabs in the backend; however, it pins ranges to their owning thread, preventing them from migrating to other threads until all subranges have been freed. In contrast, jemalloc and tcmalloc use backend structures shared across threads. These structures are more complex and therefore rely on locks for concurrency control. On heavily oversubscribed workloads (❹), such locks can cause priority inversion [36]: a low-priority thread may hold an allocator lock and be preempted, while a high-priority thread (e.g., the UI rendering thread) waits for the lock, stalls on memory allocation, and causes visible stuttering. Although such events are much rarer than the common case, they are highly disruptive when they do occur.

We therefore design allocator interfaces (e.g., malloc and free) to be non-blocking [50]. Our key observation is that we can turn rare but catastrophic blocking into brief and far less harmful memory waste instead. For each allocation or free operation, the corresponding size class first consults a bounded concurrent bitmap as the primary path. When this bitmap is empty or full, it falls back to a locked, unbounded doubly linked list. If acquiring the lock fails, the operation still does not block: instead, an allocation tries to obtain a larger range from another size class and subdivides it, while a free pushes its range onto a concurrent singly linked list and returns; later operations complete the deferred free.

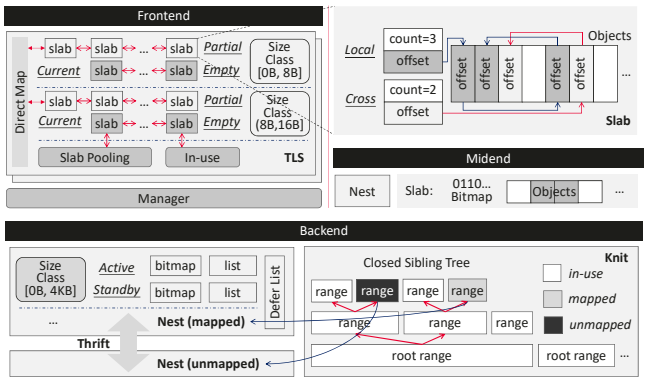


Figure 10: The overall system architecture.

4 Design and Implementation

jwmalloc is organized into three tiers (Fig. 10): frontend, midend, and backend. The frontend is slab-style with a uniform slab size equal to the OS page size (4KB). It maintains *thread-local storage* (TLS) for each thread and a *manager* that creates and destroys TLS. The backend manages virtual memory at OS-page granularity with one instance per process. Allocations smaller than 2KB are served by the frontend (Sec. 4.1). Objects between 2KB and 16KB would incur excessive tail fragmentation if handled by either the frontend or backend (both operate at 4KB granularity), so we introduce a per-process, slab-style midend for these allocations (Sec. 4.3). The backend handles allocations from 16KB up to 4MB and supplies memory for slabs used by both frontend and midend (Sec. 4.2). Allocations larger than 4MB are obtained directly via system calls. These size boundaries are chosen based on observed mobile workloads and are tunable per system configuration.

4.1 Frontend

The TLS manages multiple size classes, each with two doubly linked lists: a *partial list* for slabs with some free objects and an *empty list* for fully free slabs. To accelerate allocation, it uses a *direct map* [64], an array of slab pointers indexed by allocation size (8-byte aligned), each pointing to the *current slab*, i.e., the head of the partial list for that size class. The

TLS also maintains a shared slab pool connecting empty slabs across all size classes. When a size class requires an empty slab, it first checks its empty list and falls back to the shared pool if needed. To bound the pool, a watermark proportional to the total slab count ensures that least-recently-used empty slabs are returned to the backend when exceeded. Finally, the TLS tracks all managed slabs in an *in-use list*, and when a thread is destroyed, its slabs are handed to the reclamation thread (Sec. 4.2) for draining and reuse.

Within each slab, slab metadata resides at the slab head, preceding the objects it manages. Per-object metadata is overlaid on the objects themselves, with freed objects having their memory reinitialized to store the metadata. Cross-thread frees follow mimalloc's design, returning freed objects to the allocating thread. Each slab has an *owner thread* and two *free lists*: a *local list* for objects freed and reused by the owner, and a *cross list* for objects freed by other threads. The owner drains the cross list into the local list either periodically, to check slab reclaimability, or on demand, when allocating from a slab with an empty local list. However, unlike mimalloc's draining, which traverses the entire cross list, we optimize with *batched draining*: each cross list maintains a slab-relative head offset and free-object count (Fig. 10), atomically updatable together. During periodic draining, only the count is consulted, leaving the list intact to avoid expensive singly-linked-list merges; during on-demand draining, both the head and count are moved to the empty local list with a single atomic write.

4.2 Backend

The backend comprises three logical components (Fig. 10): (i) *Nest*, which maintains the search structure used during allocation and deallocation, organizing ranges by size and using fetch() and place() to select suitable ranges for each request (Sec. 4.2.1); (ii) *Knit*, which manages contiguous virtual-memory ranges, providing split() and join() to subdivide and coalesce them (Sec. 4.2.2); and (iii) *Thrift*, which manages cache watermarks and invokes reclaim() to release cached memory when thresholds are exceeded (Sec. 4.2.4).

Figure 11, lines 1–15, illustrates the range fetch() and place() operations, as well as split() and join() during allocation or deallocation. For a fetch(), the backend first attempts to acquire a range (acq_self). If the acquired range exactly matches the requested size, it is returned directly; otherwise, it is subdivided (split) and the remainder is released back (rel). Under eager subdivision (Sec. 3.2), this process may repeat. On a place(), the freed range is released directly if *adjacent* siblings cannot be acquired (acq_sibling); otherwise, it coalesces with the acquired sibling (join) and repeats until no further adjacent siblings can be acquired. Range acquisition and release may be concurrently accessed by multiple threads, requiring concurrency control, which is implemented via ownership transfer [73, 81] (Sec. 4.2.3).

```
1  fetch(sz) {
2    r := acq_self(sz);
3    while (r.size != sz) {
4      r, r' = split(r);
5      rel(r');
6    }
7    return r;
8  }
9
10 place(r) {
11   while(r'=acq_sibling(r)){
12     r = join(r, r');
13   }
14   rel(r);
15 }
16
17 acq_self(sz) {
18   r := nest_acq(sz);
19   if (knit_acq(r))
20     return r;
21   nest_acq_undo(r);
22   retry;
23 }
24
25 acq_sibling(r) {
26   r' := r.sibling;
27   if (!knit_acq(r'))
28     return null;
29   if (nest_acq(r'))
30     return r';
31   knit_rel(r');
32   return null;
33 }
34
35 rel(r) {
36   if (!nest_rel(r)){
37     nest_defer(r);
38   } elif (!knit_rel(r)){
39     nest_rel_undo(r);
40     nest_defer(r);
41   }
42 }
```

Figure 11: Pseudocode of backend operations.

4.2.1 Mapped and Unmapped Ranges

We maintain two Nests: a *mapped* Nest for ranges with physical pages attached, and an *unmapped* Nest for ranges without physical pages. A range allocation request first searches the mapped Nest; if no sufficiently large range is found, it falls back to the unmapped Nest; if that also fails, a 4MB *root range* is requested from the OS. On range free, if synchronous reclamation is not triggered, ranges are first returned to the mapped Nest and later moved to the unmapped Nest by the asynchronous reclamation thread (Sec. 4.2.4) [1]. If synchronous reclamation is triggered, ranges are inserted directly into the unmapped Nest. Before insertion, madvise [20] is called to discard the physical pages of the range.

Each Nest maintains multiple size classes. For each size class, the mapped Nest keeps two buffers of identical structure, an active buffer and a standby buffer, which together support the lifetime-based reclamation policy (Sec. 3.3). The unmapped Nest maintains only an active buffer. Each buffer contains a bounded bitmap and an unbounded linked list for additional ranges. fetch() and place() consult the bitmap first, falling back to the list if necessary. All size classes share a single deferred list to support non-blocking (Sec. 3.4).

4.2.2 Subdivision and Coalescing of Ranges

Knit manages multiple 4MB root ranges, each underlying a closed sibling tree. Range subdivision and coalescing operate on these trees. We formally define the size-class set and the closed sibling tree below. In our setting, a node's weight represents the number of OS pages in the range.

Definition 1 (Resolution-R size-class set)**.** *The resolution-R*

[1] Note that a range in the mapped Nest may not have physical pages if it was originally allocated from the unmapped Nest and then freed into the mapped Nest without triggering paging.

Algorithm 1: MULTI-STEP-SPLIT(X,Y)

Input: a node v of weight X; target weight Y
Precondition: $X \in S(R)$, $Y \in S(R)$, $0 < Y \leq X$
Output: a subtree rooted at v
Postcondition: the subtree is a resolution-R closed sibling tree; some node has weight Y
Begin:
write $X = n_1 \times 2^{m_1}$ with $m_1 \geq 0$ and n_1 the *largest* integer such that $1 \leq n_1 \leq 2^R$
write $Y = n_2 \times 2^{m_2}$ with $m_2 \geq 0$ and n_2 the *smallest* integer such that $1 \leq n_2 \leq 2^R$
if $m_1 \leq m_2$ **then**
  if $X \neq Y$ **then**
    split v into nodes with weights Y and $X - Y$
else
  $Z \leftarrow \lceil Y/2^{m_1} \rceil \times 2^{m_1}$
  split v into nodes with weights Z and $X - Z$
  MULTI-STEP-SPLIT(Z,Y)

size-class set $S(R)$ is defined as

$$S(R) = \{ n \times 2^m \mid m,n \in \mathbb{N},\ 1 \leq n \leq 2^R \}.$$

Definition 2 (Resolution-R closed sibling tree)**.** *A tree T is called a resolution-R closed sibling tree if:*

- *For every node v, its weight $w(v)$ is in $S(R)$.*
- *For any node v with children $c_1, \ldots, c_N$, we have $w(v) = \sum_{i=1}^{N} w(c_i)$, and for all $1 \leq p \leq q \leq N$, $\sum_{i=p}^{q} w(c_i) \in S(R)$.*

Coalescing merges contiguous siblings under the same parent. Once all children of a node have been coalesced, the parent becomes eligible to merge with its own siblings. For subdivision, we introduce the MULTI-STEP-SPLIT algorithm (Alg. 1), which executes the split steps shown in Fig. 11, lines 3–4.

We illustrate MULTI-STEP-SPLIT with an example. Suppose we want to subdivide a 6-page range from a 64-page range under a resolution-3 setting (i.e., $R = 3$, $X = 64$, and $Y = 6$). The max-n decomposition of 64 is 8×2^3, and the min-n decomposition of 6 is 3×2^1. Since $m_1 > m_2$, the recursive case applies: we compute $Z = \lceil 6/8 \rceil \times 8 = 8$, split 64 into two children of weights 8 and 56, and recurse on the child of weight 8. For MULTI-STEP-SPLIT$(8,6)$, we have $8 = 8 \times 2^0$ and $6 = 3 \times 2^1$, so $m_1 \leq m_2$ and $X \neq Y$, which triggers the direct case, splitting 8 into 6 and 2. In this example, the construction obtains the 8-page subrange from the 64-page range in a single step, leaving a contiguous 56-page range available for other large requests. In contrast, a naive binary splitting scheme (e.g., $64 \to 32 \to 16 \to 8$) requires three steps to reach 8, and the intermediate nodes cannot serve the 56-page request.

4.2.3 Ownership Transfer of Ranges

Conceptually, a range allocation transfers ownership of a suitable range from the backend to the requester, while a free operation transfers ownership back to the backend. Within the backend, a range is co-owned by Nest and Knit: acq must obtain ownership from both components, and rel must release it to both. Since these two ownerships cannot be acquired or released atomically, inconsistent states can arise, in which one component succeeds while the other fails. For example, when a free operation attempts to acquire a range's adjacent sibling in Knit to coalesce with it, a concurrent allocation may have already acquired that sibling range from Nest.

To handle this, ownership transfer follows a staged protocol with rollback (Fig. 11). acq_self first locates a suitable range and acquires its Nest ownership (nest_acq), then attempts to acquire its Knit ownership (knit_acq). If the Knit acquisition fails, it rolls back the Nest acquire (nest_acq_undo) and retries a bounded number of times at the same size; if these retries fail, it falls back to a larger range, ensuring a non-blocking path. acq_sibling first acquires the adjacent sibling in Knit, then in Nest; if the Nest acquisition fails, it releases the Knit ownership (knit_rel) and returns null. rel first attempts to release Nest ownership. If this fails, the range is placed on the deferred list (nest_defer) and the operation returns. Otherwise, it releases Knit ownership. If the Knit release fails, it rolls back the Nest release (nest_rel_undo), places the range on the deferred list, and returns. Deferred ranges are later drained either by the asynchronous reclamation thread (Sec. 4.2.4) or synchronously on demand when a nest_acq fails to find a suitable range (e.g., at line 18 when r.size $>$ sz); draining simply re-invokes place() on all deferred ranges.

Each range has three states: inuse (currently in use or on the deferred list), mapped (present in the mapped Nest), and unmapped (present in the unmapped Nest). In Knit, knit_acq for a range r requires that r is not inuse, preventing it from being acquired twice. When releasing a range r to the mapped Nest, Knit must ensure that none of r's adjacent siblings is already mapped; similarly, when releasing to the unmapped Nest, none of them is unmapped. This guarantees that free adjacent siblings can always be coalesced if they are in the same state. The states of all siblings under the same parent are packed into a single atomic variable. If any of these conditions fail, knit_acq or knit_rel returns false; otherwise, it atomically updates r's state via compare-and-swap [50]. In Nest, nest_acq and nest_rel acquire and release ownership in the underlying generic structures, i.e., bitmaps and lists.

4.2.4 Asynchronous Reclamation

The reclamation thread repeatedly performs the following steps: (i) drain slabs handed off from TLS instances whose threads have been destroyed (Sec. 4.1); (ii) traverse the deferred list of each size class in both the mapped and unmapped Nests and call place() on each range to complete deferred re-

leases (Sec. 4.2.3); (iii) swap the ready and standby buffers (Sec. 3.3); (iv) perform a futex wait [8] with a timeout (500ms in our setting), possibly being woken early by synchronous reclamation (Sec. 3.3); and (v) for each size class in the mapped Nest, apply madvise to all ranges in the standby buffer to discard their physical pages, then move those ranges to the corresponding size class in the unmapped Nest.

4.3 Midend

The midend uses a Nest similar to the backend's mapped Nest, but with only the active buffer. It manages *midend slabs*, each containing multiple objects of size 2KB–16KB. An allocation selects a corresponding slab from the Nest; if this turns the slab from partial to full, it is removed from the Nest. On object free, if the object's slab changes from full to partial, it is reinserted into the Nest; if it changes from partial to empty, it is returned to the backend as a free range. Each slab uses a bitmap with atomic bit operations to track its objects and implement these state transitions (Fig. 10). The midend shares metadata with the backend: when a range is initialized as a midend slab, the backend Nest's fields for that range become unused, and the memory is repurposed to hold the midend Nest's metadata, such as list links and the slab's bitmap.

5 Verification under Weak Memory Models

jwmalloc is a complex allocator that lies on the critical path of virtually all system services and applications on the device. Any bug can cause severe failures, such as memory corruption that may only manifest far from the point where it actually occurs. In concurrent settings, especially under weak-memory models, such bugs are particularly difficult to reproduce and diagnose. To mitigate this risk, we employ bounded formal verification to increase confidence in jwmalloc's correctness.

Traditional approaches based on manual, machine-checked proofs (e.g., large Coq developments [84]) are prohibitively labor-intensive for a production allocator of this complexity. More recent frameworks such as RefinedC [77] and Verus [61, 62] reduce the proof burden but reason only under sequential consistency, leaving executions arising from weak-memory behaviors, which are often more error-prone, unchecked. For mobile allocators, this limitation is particularly relevant, as mobile hardware is dominated by architectures like ARM that implement weak memory models.

To balance verification coverage against engineering effort, we adopt the VSync bounded model-checking approach [56–58] that automatically verifies a library (here: jwmalloc) against small multi-threaded test cases under weak-memory models. In this approach, like in testing, the test cases act as functional specifications of the library by means of assertions; unlike testing, *all* weak-memory executions of the library on the test cases are explored, and generic properties such as memory safety, absence of data races, and loop termination, are also verified on each execution. Therefore, verification is complete within the bounds set by the tests, while corner cases not covered by tests are not verified.

Note that the number of executions grows rapidly with the complexity of the test cases, such as the number of threads and library API calls. Since all these executions are explored, the test cases need to be engineered to achieve a good balance between verification time and coverage. We achieve this in the following ways:

First, we fully decouple jwmalloc's components. The frontend, midend, and backend each provide a standard malloc/free-style interface and interact with other components only through these interfaces. During verification, each component is treated as a conventional C library rather than an allocator. For example, the frontend exports *frontend_malloc* and, in production, obtains its slabs via *backend_malloc*; during verification, it uses the system allocator's standard *malloc*, treated by the verifier as a standard library with verification deliberately skipped. This modularity allows verification of components both in isolation and in combination.

Second, we partition the client program into multiple specialized clients, each targeting specific edge cases. For example, the frontend exposes concurrency scenarios such as cross-thread frees, on-demand draining, periodic draining, and thread creation and destruction. We construct multi-threaded clients where each thread invokes the relevant API operations a bounded number of times, triggering these scenarios while keeping verification time manageable.

Finally, we shrink jwmalloc's configuration by scaling down certain constants, for example, reducing the backend bitmap from 64 to 2 entries, without changing the code logic.

With the mechanisms described above, we ensure that verification of each client completes within 10 minutes. This verification uncovered a previously unknown bug during the development of jwmalloc. The bug originates from the backend's closed sibling tree, which employs a metadata-shifting design. For the root node, which has no siblings, this design mistakenly assumes a sibling exists and attempts to access its state along one code path, causing out-of-bounds reads in the data area and resulting in a crash only when the data area contains specific values. Using a client targeting the backend, the model checker can expose this bug in roughly ten seconds and reports an error due to accessing uninitialized memory.

6 Evaluation

We compare jwmalloc against widely-used industrial allocators, including mimalloc [21] v2.1.8, jemalloc [18] v5.3.0, and tcmalloc [33] from gperftools-2.16, across various workloads. All allocators were run with their default configurations and profiling disabled.

Hardware instruction counts. We first analyze hardware instruction and conditional-branch counts for in-slab operations

Table 2: Instruction and branch counts on x86 for in-slab operations. Each cell reports "instructions & branches".

Operation	**jwmalloc**	jemalloc	mimalloc	tcmalloc
alloc	**16 & 1**	27 & 3	17 & 2	36 & 6
local free	**22 & 3**	38 & 4	26 & 4	36 & 5
cross-thread free	**29 & 5**	>100 & >10	72 & 9	>100 & >10

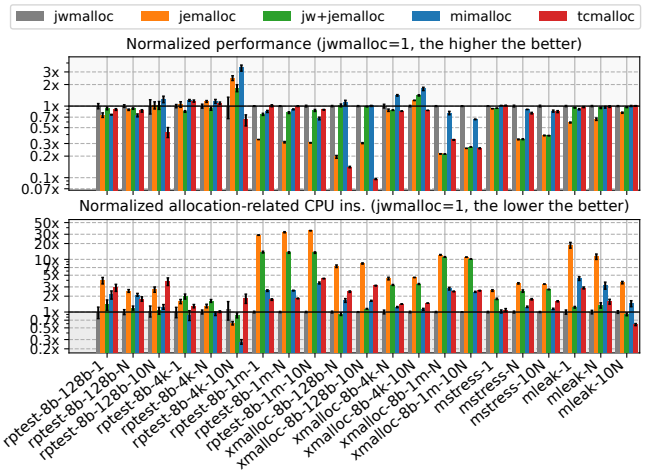


Figure 12: Comparison of performance (top) and CPU instruction counts (bottom).

in each allocator. As shown in Table 2, jwmalloc executes fewer instructions than the other allocators, particularly for cross-thread frees, where it reduces the instruction count by over 70% compared to jemalloc. This low instruction cost contributes to jwmalloc's performance and load advantages.

6.1 Micro-benchmarks

Admittedly, microbenchmarks cannot fully capture the complexity of real-world workloads, but they do reveal how allocators behave under simple synthetic workloads. We select four open-source benchmarks, namely rptest, xmalloc, mstress, and mleak, to exercise four scenarios: predominantly thread-local allocate/free, cross-thread frees, diverse object lifetimes, and frequent thread creation and destruction [27, 31, 32]. To avoid noise from background services on smartphones, we run these benchmarks on a server [16] using 4 cores and different thread counts to emulate mobile behavior. We vary each benchmark's allocation-size range while keeping its default size distribution. For example, rptest-8B-128B-10N denotes rptest with allocation sizes in [8B, 128B] and 10N threads, where N is the number of cores. To better isolate the effect of jwmalloc's frontend, we also pair it with jemalloc's backend; we denote this variant as jw+jemalloc.

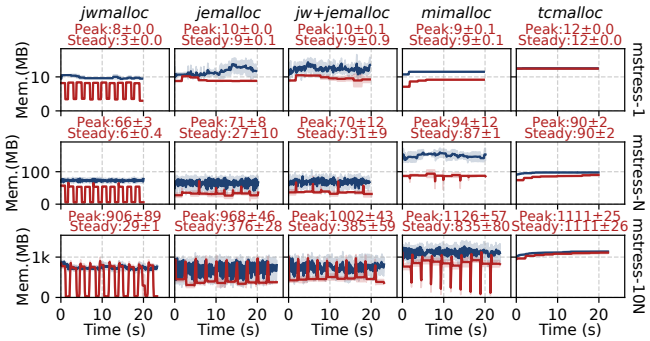


Figure 13: Memory footprint comparison for the original (blue) and sleep-augmented (red) mstress workloads. Shading indicates the standard error.

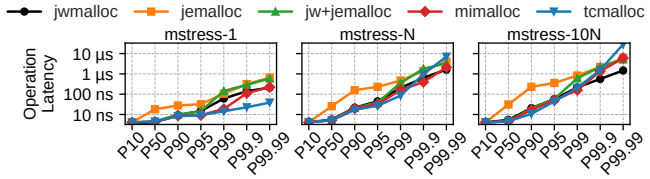


Figure 14: Operation latency comparison (lower is better).

6.1.1 Overall Performance and CPU Overhead

As shown in Fig. 12, even with only the frontend, jwmalloc substantially outperforms jemalloc. For instance, jw+jemalloc is 24% and 4.1× faster than jemalloc on rptest-8B-128B-1 and xmalloc-8B-128B-N, respectively, highlighting the benefits of our frontend. With both frontend and backend enabled, jwmalloc leads the other allocators in nearly all workloads, especially in reducing instruction counts. On average, it improves performance by 74% over jemalloc and reduces allocator-side instructions by about 82%. On rptest-8B-128B-1, jwmalloc outperforms jemalloc, mimalloc, and tcmalloc by 25%, 24%, and 10%, respectively. On rptest-8B-1MB-N, allocator-side instruction counts for jemalloc, mimalloc, and tcmalloc are 32.8×, 2.55×, and 1.81× those of jwmalloc, underscoring the strength of our backend design.

There are a few scenarios where jwmalloc shows modest regressions in performance or load, notably the rptest and xmalloc workloads with the 8B-4KB-10N configuration. One reason is that a nontrivial fraction of the allocation-size distribution lies near 4KB; for these sizes, jwmalloc's unified slab-size design deliberately trades some performance. Another reason is that 10N represents a heavy oversubscription in microbenchmarks; under such workloads, jwmalloc remains non-blocking but incurs additional work, resulting in some performance loss.

6.1.2 Memory Footprint

We use mstress to study allocator memory behavior. In each round, every thread allocates several objects, exchanges some with other threads, and frees a subset, closely resembling mobile workloads. We run both the original configuration and a modified version that inserts a 2-second sleep between rounds to emulate smartphone foreground and background behavior, recording peak and steady-state memory usage for the modified version.

As shown in Fig. 13, under the sleep-augmented workload (the red line), jwmalloc consistently uses less memory than the other allocators across all thread counts. For example, in mstress-10N, jwmalloc's peak and steady-state footprints are 906MB and 29MB, whereas jemalloc's are 968MB and 376MB. In addition, jw+jemalloc and jemalloc exhibit similar peak and steady-state footprints. This demonstrates the effectiveness of our backend for workloads with large peak-trough swings (❷). In contrast, with its default settings, tcmalloc rarely triggers reclamation in this workload. We also observe that mimalloc's reclamation mostly occurs after the sleep period ends rather than shortly after it begins, confirming the delayed effect of synchronous reclamation discussed in Sec. 3.3.

Under the original workload (the blue line), jw+jemalloc exhibits smaller memory fluctuations than jemalloc, especially when there is a larger number of threads, which indicates the effectiveness of slab pooling (Sec. 3.1). In contrast, mimalloc and tcmalloc show higher memory usage, due respectively to a partial inability to reformat memory and a lack of memory reclamation mechanisms.

Taken together across both workloads, we find that under continuous allocation demand (the original workload), jwmalloc exhibits smaller memory fluctuations than jemalloc and jw+jemalloc (the blue line). When demand drops during the inserted sleep, jwmalloc returns more memory to the OS (the red line). This demonstrates the effectiveness of our lifetime-based reclamation policy for workloads requiring aggressive reclamation (❸): it reduces OS-side overhead from repeated allocation and deallocation while maintaining lower peak and steady-state memory footprints.

6.1.3 Operation Latency

We then measure allocator operation latency under the original mstress workload, as shown in Fig. 14. With a single thread, tcmalloc exhibits the lowest latency since it retains memory and rarely requests it from the OS. However, as the thread count increases, particularly under oversubscription, tcmalloc's tail latency rises sharply due to its lock-based design. In contrast, jwmalloc's non-blocking design maintains lower latency than the other allocators. For instance, in mstress-10N, jwmalloc's P99.99 latency is 1.5μs, compared to 5.9μs for the best competing allocator. This demonstrates that jwmalloc ensures low-latency memory operations even under heavy oversubscription (❹).

Table 3: Percentage energy-consumption reduction of jwmalloc relative to jemalloc under selected workloads. CPU_B, CPU_M, and CPU_L denote the big, middle, and little CPU clusters, respectively, and LPDDR denotes DRAM power.

	CPU_B	CPU_M	CPU_L	LPDDR	CPU Ins.
article_reading	7.3 ± 5.2	4.7 ± 0.53	7.2 ± 4.6	2.8 ± 1.1	21.0 ± 3.0
video_playing	11.0 ± 3.9	6.5 ± 0.78	5.8 ± 0.78	2.2 ± 0.41	6.7 ± 3.4

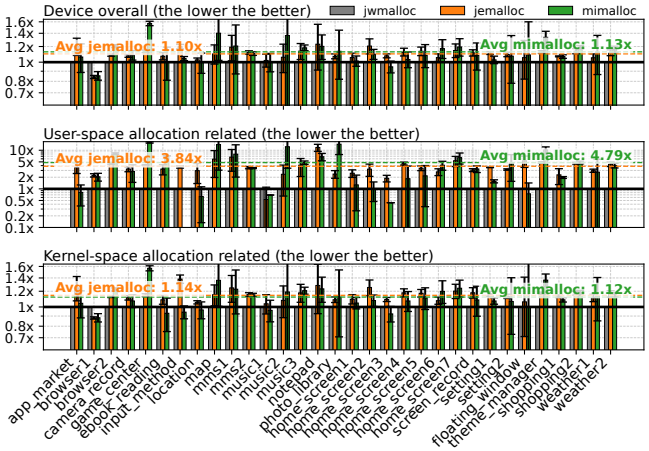


Figure 15: Normalized total CPU instruction counts and user- and kernel-space allocation related instructions.

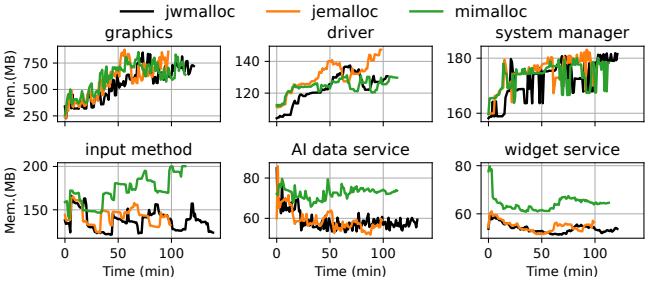


Figure 16: Proportional set size (PSS) over time for the top memory-consuming system services.

A closer comparison with jw+jemalloc further reveals a division of responsibility within jwmalloc's components. The frontend primarily determines non-tail latency, while the backend dominates tail latency. This highlights the effectiveness of the two components in jointly reducing operation latency.

6.2 Real-World Mobile Scenarios

We evaluate jwmalloc in a realistic mobile setting on a flagship smartphone, the Huawei Mate 70 Pro [14], running HarmonyOS 5.1 [22]. We replace the system's default jemalloc with either jwmalloc or mimalloc. We omit tcmalloc, which is implemented in C++ and thus incompatible with the phone's libc and toolchain [1, 35]. The device uses eight ARMv8-A

Benchmark ID	Action
app_market	Open the app market, search for and install a specific app, then uninstall the app.
browser1	Open a browser app, enter a URL, and stay on the page.
browser2	Open a browser app, enter a URL, and scroll the web page.
camera_record	Open the camera app, switch to video mode, and start recording.
game_center	Open the game center app, browse its home screen, then exit the app.
ebook_reading	Open a reader app, open the first book, and turn pages.
input_method	Swipe down from the home screen, tap the search box, and search for a keyword.
location	Open the device-finding app and refresh the location.
map	Open a map app, search for a keyword, and start navigation.
mms1	Open the messages app, enter the message list, view the first message, and scroll back.
mms2	Open the messages app, enter the message list, and scroll.
music1	Open a music app and play a local audio file.
music2	Open a music app and play local music with the screen off.
music3	Open a music app, enter the local song list, and scroll the list.
notepad	Open the notepad app, create a new note, and switch between tabs.
photo_library	Open the photo library, tap “Albums”, tap “Photos”, enter the thumbnail view, and scroll.
home_screen1	Enter an enlarged folder, swipe once, and exit.
home_screen2	Launch the settings app and swipe up to send it to the background.
home_screen3	Continuously launch Settings, Notepad, Messages, Phone, and Calendar; enter the multitasking interface and repeatedly swipe.
home_screen4	Repeatedly increase and decrease the volume.
home_screen5	Repeatedly pull down the control center and then exit it.
home_screen6	Enter the minus-one screen (leftmost panel) and repeatedly swipe.
home_screen7	Enter an enlarged folder, then repeatedly open and exit it.
screen_record	From the home screen, pull down the control center, tap screen recording, record a clip, then stop and return to the home screen.
setting1	Open the settings app and repeatedly swipe.
setting2	Open the settings app, log in to the account, tap “Scan”, select a QR code from Photos, and scan.
floating_window	Open the settings app, swipe up to the floating-window area, and release to create a floating window.
theme_manager	Open the theme app and repeatedly swipe on the home page.
shopping1	Open a shopping app and scroll down to browse products.
shopping2	Open a shopping app, enter the “Me” page, and swipe.
weather1	Open the weather app and send it to the background.
weather2	Open the weather app and swipe left to switch cities.

Figure 17: Detailed description of actions in real-world mobile benchmarks.

CPU cores [6] and 12GB of RAM. We run a suite of common smartphone scenarios for approximately two hours, covering core system interactions (e.g., home-screen scrolling, volume adjustment) and high-frequency app workloads (e.g., map navigation, music playback); the scenarios are detailed in Fig. 17. Each benchmark was executed five times, and we report the mean and standard error.

Figure 15 presents normalized CPU instruction counts from three perspectives: overall device instructions, user-space allocation-related instructions, and kernel-space allocation-related instructions. jwmalloc achieves substantial reductions across all three. On average, jemalloc and mimalloc execute 10% and 13% more overall instructions than jwmalloc, respectively. Broken down by source, in user space, their allocation-related instruction counts are 3.84× and 4.79× those of jwmalloc; in kernel space, they are 14% and 12% higher. These results show that jwmalloc effectively reduces allocation overhead and overall system load.

To further analyze how instruction reduction translates into power savings, we focus on two representative and relatively power-stable benchmark steps: article reading and video playback. For each step, we measure both instruction counts and the power consumption of major hardware components. As shown in Table 3, compared with jemalloc, jwmalloc reduces instruction counts by 7%–21%, lowers CPU-cluster power by 5–11%, and reduces LPDDR power by 2–3%. The CPU power reduction mainly results from shifts to lower voltage-frequency points and the migration of threads from big cores to small cores.

During the benchmark, we sample each process’s proportional set size (PSS), including swapped-out pages, once per minute. We then select the top system services by memory usage and plot their memory footprints over time in Fig. 16. For most services, jwmalloc’s memory usage is comparable to or

lower than that of jemalloc, whereas mimalloc often consumes more memory.

7 Conclusion and Future Work

We presented jwmalloc, a mobile-first memory allocator that significantly improves performance by addressing the mismatch between existing designs and mobile workloads. It has been successfully deployed on over 12 million devices. As future work, we plan to extend its core principles, such as the closed sibling tree, to server environments to enhance datacenter performance.

Acknowledgments

We thank our shepherd and the anonymous reviewers for their insightful comments. We also thank our colleagues and interns at Huawei Fields Lab for their invaluable support in the commercial deployment of jwmalloc.

References

[1] musl libc. https://wiki.musl-libc.org/.

[2] About the background execution sequence. https://developer.apple.com/documentation/uikit/about-the-background-execution-sequence.

[3] Android. https://www.android.com/.

[4] Android jemalloc wrapper. https://android.googlesource.com/platform/bionic/+/refs/heads/main/libc/bionic/jemalloc_wrapper.cpp#89.

[5] Arm memory tagging extension. https://source.android.com/docs/security/test/memory-safety/arm-mte.

[6] ARMv8-A Architecture Reference Manual. https://archive.org/details/armv8-a-reference-manual.

[7] CWE-416: Use After Free. https://cwe.mitre.org/data/definitions/416.html.

[8] futex(2) — linux manual page. https://www.man7.org/linux/man-pages/man2/futex.2.html.

[9] Galaxy S25 Ultra. https://www.samsung.com/us/smartphones/galaxy-s25-ultra.

[10] Graphics. https://source.android.com/docs/core/graphics.

[11] Graphics architecture. https://source.android.com/docs/core/graphics/architecture.

[12] Hardware-assisted addresssanitizer. https://source.android.google.cn/docs/security/test/hwasan.

[13] Hardware-assisted addresssanitizer design. https://clang.llvm.org/docs/HardwareAssistedAddressSanitizerDesign.html.

[14] Huawei Mate 70 Series. https://en.wikipedia.org/wiki/Huawei_Mate_70.

[15] HUAWEI Mate XT. https://consumer.huawei.com/en/phones/mate-xt-ultimate-design.

[16] Intel® Xeon® Platinum 8260 Processor. https://www.intel.com/content/www/us/en/products/sku/192474/intel-xeon-platinum-8260-processor-35-75m-cache-2-40-ghz/specifications.html.

[17] iphone-17. https://www.apple.com/iphone-17.

[18] jemalloc. https://github.com/jemalloc/jemalloc.

[19] Jemalloc Memory Analysis. https://doris.apache.org/docs/3.x/admin-manual/trouble-shooting/memory-management/memory-analysis/jemalloc-memory-analysis#dirty-page-memory-too-large.

[20] madvise(2) — linux manual page. https://man7.org/linux/man-pages/man2/madvise.2.html.

[21] mimalloc. https://github.com/microsoft/mimalloc.

[22] OpenHarmony Project. https://docs.openharmony.cn/pages/v5.1/en/OpenHarmony-Overview.md.

[23] Overview of memory management. https://developer.android.com/topic/performance/memory-overview.

[24] PartitionAlloc Design. https://chromium.googlesource.com/chromium/src/%2B/master/base/allocator/partition_allocator/PartitionAlloc.md.

[25] Performance - ProtonAOSP. https://protonaosp.org/performance.

[26] Reducing terminations in your app. https://developer.apple.com/documentation/uikit/about-the-background-execution-sequence.

[27] rpmalloc-benchmark. https://github.com/mjansson/rpmalloc-benchmark.

[28] Rust in the Android platform. https://security.googleblog.com/2021/04/rust-in-android-platform.html.

[29] Scudo Hardened Allocator. https://llvm.org/docs/ScudoHardenedAllocator.html.

[30] Smartphone Market. https://www.grandviewresearch.com/industry-analysis/smart-phone-market.

[31] Suite for benchmarking malloc implementations. https://github.com/daanx/mimalloc-bench.

[32] SuperMalloc. https://github.com/kuszmaul/SuperMalloc.

[33] tcmalloc. https://github.com/google/tcmalloc.

[34] The activity lifecycle. https://developer.android.com/guide/components/activities/activity-lifecycle.

[35] Third-party open-source software musl. https://github.com/openharmony/third_party_musl.

[36] What really happened on Mars? https://www.cs.unc.edu/~anderson/teach/comp790/papers/mars_pathfinder_long_version.html.

[37] Retrofitting temporal memory safety on c++. https://security.googleblog.com/2022/05/retrofitting-temporal-memory-safety-on-c.html, May 2022.

[38] Rust in Android: move fast and fix things. https://security.googleblog.com/2025/11/rust-in-android-move-fast-fix-things.html, November 2025.

[39] Tomas Akenine-Moller, Eric Haines, and Naty Hoffman. *Real-time rendering*. AK Peters/crc Press, 2019.

[40] Emery D Berger, Kathryn S McKinley, Robert D Blumofe, and Paul R Wilson. Hoard: A scalable memory allocator for multithreaded applications. *ACM Sigplan Notices*, 35(11):117–128, 2000.

[41] Emery D Berger, Benjamin G Zorn, and Kathryn S McKinley. Composing high-performance memory allocators. *ACM SIGPLAN Notices*, 36(5):114–124, 2001.

[42] Emery D Berger, Benjamin G Zorn, and Kathryn S McKinley. Reconsidering custom memory allocation. In *Proceedings of the 17th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 1–12, 2002.

[43] William Blair, William Robertson, and Manuel Egele. Mpkalloc: Efficient heap meta-data integrity through hardware memory protection keys. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 136–155. Springer, 2022.

[44] Jeff Bonwick and Jonathan Adams. Magazines and vmem: Extending the slab allocator to many cpus and arbitrary resources. In *USENIX Annual Technical Conference, General Track*, pages 15–33, 2001.

[45] Jeff Bonwick et al. The slab allocator: An object-caching kernel memory allocator. In *USENIX summer*, volume 16. Boston, MA, USA, 1994.

[46] Rohit Chandra. *Parallel programming in OpenMP*. Morgan kaufmann, 2001.

[47] Jason Evans. A scalable concurrent malloc (3) implementation for freebsd. In *Proc. of the bsdcan conference, ottawa, canada*, 2006.

[48] Umar Farooq, Zhijia Zhao, Manu Sridharan, and Iulian Neamtiu. Livedroid: Identifying and preserving mobile app state in volatile runtime environments. *Proceedings of the ACM on Programming Languages*, 4(OOPSLA):1–30, 2020.

[49] Mel Gorman. *Understanding the Linux virtual memory manager*, volume 352. Prentice Hall Upper Saddle River, 2004.

[50] Maurice Herlihy, Nir Shavit, Victor Luchangco, and Michael Spear. *The art of multiprocessor programming*. Newnes, 2020.

[51] Daniel S Hirschberg. A class of dynamic memory allocation algorithms. *Communications of the ACM*, 16(10):615–618, 1973.

[52] Richard Jones and Rafael Lins. *Garbage collection: algorithms for automatic dynamic memory management*. John Wiley & Sons, Inc., 1996.

[53] Svilen Kanev, Sam Likun Xi, Gu-Yeon Wei, and David Brooks. Mallacc: Accelerating memory allocation. *ACM SIGPLAN Notices*, 52(4):33–45, 2017.

[54] Kenneth C Knowlton. A fast storage allocator. *Communications of the ACM*, 8(10):623–624, 1965.

[55] Donald E Knuth. *The Art of Computer Programming: Fundamental Algorithms, Volume 1*. Addison-Wesley Professional, 1997.

[56] Michalis Kokologiannakis, Azalea Raad, and Viktor Vafeiadis. Effective lock handling in stateless model checking. *Proceedings of the ACM on Programming Languages*, 3(OOPSLA):1–26, 2019.

[57] Michalis Kokologiannakis, Azalea Raad, and Viktor Vafeiadis. Model checking for weakly consistent libraries. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 96–110, 2019.

[58] Michalis Kokologiannakis and Viktor Vafeiadis. Genmc: A model checker for weak memory models. In *International Conference on Computer Aided Verification*, pages 427–440. Springer, 2021.

[59] Rakesh Kumar, Keith I Farkas, Norman P Jouppi, Parthasarathy Ranganathan, and Dean M Tullsen. Single-isa heterogeneous multi-core architectures: The potential for processor power reduction. In *Proceedings. 36th Annual IEEE/ACM International Symposium on Microarchitecture, 2003. MICRO-36.*, pages 81–92. IEEE, 2003.

[60] Ori Lahav, Viktor Vafeiadis, Jeehoon Kang, Chung-Kil Hur, and Derek Dreyer. Repairing sequential consistency in c/c++ 11. *ACM SIGPLAN Notices*, 52(6):618–632, 2017.

[61] Andrea Lattuada, Travis Hance, Jay Bosamiya, Matthias Brun, Chanhee Cho, Hayley LeBlanc, Pranav Srinivasan, Reto Achermann, Tej Chajed, Chris Hawblitzel, et al. Verus: A practical foundation for systems verification. In *Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles*, pages 438–454, 2024.

[62] Andrea Lattuada, Travis Hance, Chanhee Cho, Matthias Brun, Isitha Subasinghe, Yi Zhou, Jon Howell, Bryan Parno, and Chris Hawblitzel. Verus: Verifying rust programs using linear ghost types. *Proceedings of the ACM on Programming Languages*, 7(OOPSLA1):286–315, 2023.

[63] Doug Lea and Wolfram Gloger. A memory allocator, 1996.

[64] Daan Leijen, Benjamin Zorn, and Leonardo De Moura. Mimalloc: Free list sharding in action. In *Asian Symposium on Programming Languages and Systems*, pages 244–265. Springer, 2019.

[65] Ruihao Li, Qinzhe Wu, Krishna Kavi, Gayatri Mehta, Neeraja J Yadwadkar, and Lizy K John. Nextgen-malloc: Giving memory allocator its own room in the house. In *Proceedings of the 19th Workshop on Hot Topics in Operating Systems*, pages 135–142, 2023.

[66] Beichen Liu, Pierre Olivier, and Binoy Ravindran. Slimguard: A secure and memory-efficient heap allocator. In *Proceedings of the 20th International Middleware Conference*, pages 1–13, 2019.

[67] Martin Maas, David G Andersen, Michael Isard, Mohammad Mahdi Javanmard, Kathryn S McKinley, and Colin Raffel. Learning-based memory allocation for c++ server workloads. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 541–556, 2020.

[68] Martin Maas, Chris Kennelly, Khanh Nguyen, Darryl Gove, Kathryn S McKinley, and Paul Turner. Adaptive huge-page subrelease for non-moving memory allocators in warehouse-scale computers. In *Proceedings of the 2021 ACM SIGPLAN International Symposium on Memory Management*, pages 28–38, 2021.

[69] Martin Maas, Chris Kennelly, Khanh Nguyen, Darryl Gove, Kathryn S McKinley, and Paul Turner. Adaptive huge-page subrelease for non-moving memory allocators in warehouse-scale computers. In *Proceedings of the 2021 ACM SIGPLAN International Symposium on Memory Management*, pages 28–38, 2021.

[70] Maged M Michael. Scalable lock-free dynamic memory allocation. In *Proceedings of the ACM SIGPLAN 2004 conference on Programming language design and implementation*, pages 35–46, 2004.

[71] Michael D Moffitt. Minimalloc: A lightweight memory allocator for hardware-accelerated machine learning. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 4*, pages 238–252, 2023.

[72] Jonas Oberhauser, Rafael Lourenco De Lima Chehab, Diogo Behrens, Ming Fu, Antonio Paolillo, Lilith Oberhauser, Koustubha Bhat, Yuzhong Wen, Haibo Chen, Jaeho Kim, et al. Vsync: push-button verification and optimization for synchronization primitives on weak memory models. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 530–545, 2021.

[73] Peter W O'hearn. Resources, concurrency, and local reasoning. *Theoretical computer science*, 375(1-3):271–307, 2007.

[74] Page and Hagins. Improving the performance of buddy systems. *IEEE Transactions on Computers*, 100(5):441–447, 1986.

[75] James L Peterson and Theodore A Norman. Buddy systems. *Communications of the ACM*, 20(6):421–431, 1977.

[76] Antonin Reitz, Aymeric Fromherz, and Jonathan Protzenko. Starmalloc: verifying a modern, hardened memory allocator. *Proceedings of the ACM on Programming Languages*, 8(OOPSLA2):1757–1786, 2024.

[77] Michael Sammler, Rodolphe Lepigre, Robbert Krebbers, Kayvan Memarian, Derek Dreyer, and Deepak Garg. Refinedc: automating the foundational verification of c code with refined ownership types. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, pages 158–174, 2021.

[78] Kenneth K Shen and James L Peterson. A weighted buddy method for dynamic storage allocation. *Communications of the ACM*, 17(10):558–562, 1974.

[79] Ting Su, Lingling Fan, Sen Chen, Yang Liu, Lihua Xu, Geguang Pu, and Zhendong Su. Why my app crashes? understanding and benchmarking framework-specific exceptions of android apps. *IEEE Transactions on Software Engineering*, 48(4):1115–1137, 2020.

[80] David Ungar. Generation scavenging: A non-disruptive high performance storage reclamation algorithm. *ACM Sigplan notices*, 19(5):157–167, 1984.

[81] Viktor Vafeiadis and Matthew Parkinson. A marriage of rely/guarantee and separation logic. In *International Conference on Concurrency Theory*, pages 256–271. Springer, 2007.

[82] Mark Weiser, Brent Welch, Alan Demers, and Scott Shenker. Scheduling for reduced cpu energy. In *Mobile Computing*, pages 449–471. Springer, 1994.

[83] Yuanpei Wu, Dong Du, Chao Xu, Yubin Xia, Ming Fu, Binyu Zang, and Haibo Chen. D-vsync: Decoupled rendering and displaying for smartphone graphics. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1*, pages 326–341, 2025.

[84] Fengwei Xu, Ming Fu, Xinyu Feng, Xiaoran Zhang, Hui Zhang, and Zhaohui Li. A practical verification framework for preemptive os kernels. In *International Conference on Computer Aided Verification*, pages 59–79. Springer, 2016.

[85] Divakar Yadav and AK Sharma. Tertiary buddy system for efficient dynamic memory allocation. In *Conference: Proceeding SEPADS*, volume 10, 2010.

[86] Hanmei Yang, Xin Zhao, Jin Zhou, Wei Wang, Sandip Kundu, Bo Wu, Hui Guan, and Tongping Liu. Numalloc: A faster numa memory allocator. In *Proceedings of the 2023 ACM SIGPLAN International Symposium on Memory Management*, pages 97–110, 2023.

[87] Zhuangzhuang Zhou, Vaibhav Gogte, Nilay Vaish, Chris Kennelly, Patrick Xia, Svilen Kanev, Tipp Moseley, Christina Delimitrou, and Parthasarathy Ranganathan. Characterizing a memory allocator at warehouse scale. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, page 192–206, La Jolla, CA, USA, 2024.

Neuro-Symbolic Proof Generation for Scaling Systems Software Verification

Baoding He[1,2,*] Zenan Li[3,*] Wei Sun[1,2] Yuan Yao[1,2,†] Taolue Chen[4]
Xiaoxing Ma[1,2,†] Zhendong Su[3]

[1]State Key Laboratory of Novel Software Technology, Nanjing University, China
[2]School of Computer Science, Nanjing University, China
[3]Department of Computer Science, ETH Zurich, Switzerland
[4]School of Computing and Mathematical Sciences, Birkbeck, University of London, UK

Abstract

Formal verification via interactive theorem proving is increasingly used to ensure the correctness of critical systems, yet constructing large proof scripts remains highly manual and limits scalability. Advances in large language models (LLMs), especially in mathematical reasoning, make their integration into software verification increasingly promising. This paper introduces a neuro-symbolic proof generation framework designed to automate proof search for system-level verification projects. The framework performs a best-first tree search over proof states, repeatedly querying an LLM for the next candidate proof step. On the neural side, we fine-tune LLMs using datasets of proof state–step pairs; on the symbolic side, we incorporate a range of ITP tools to repair rejected steps, filter and rank proof states, and automatically discharge subgoals when search progress stalls. This synergy enables data-efficient LLM adaptation and semantics-informed pruning of the search space. We implement the framework on a new Isabelle REPL that exposes fine-grained proof states and automation tools, and evaluate it on the FVEL seL4 benchmark and additional Isabelle developments. On seL4, the system proves up to 77.6% of the theorems, substantially surpassing previous LLM-based approaches and standalone Sledgehammer, while solving significantly more multi-step proofs. Results across further Isabelle benchmarks demonstrate strong generalization, indicating a viable path toward scalable automated software verification.

1 Introduction

Formal verification plays a vital role in ensuring the correctness and reliability of software systems, especially in safety- and security-critical domains where failures can be catastrophic [1, 2, 3]. Interactive theorem proving (ITP) stands out for its expressiveness and ability to provide the strongest assurance. With ITPs, landmark projects such as CompCert (a formally verified optimizing C compiler whose correctness is mechanically proven in Rocq [4]) and seL4 (a formally verified OS microkernel whose functional correctness, security properties and reliability guarantees are proven in Isabelle/HOL [5]) have been successfully delivered, which demonstrate that formally verified systems can achieve both practical usability and rigorous correctness.

While ITP offers unparalleled assurance, it also incurs substantial costs, preventing its wide adoption, especially in large-scale, system-level software [6]. Specifically, a typical software verification workflow involves 1) writing precise formal *specifications* to be proved and 2) constructing rigorous *proofs* for these specifications, both of which demand immense human effort. For example, the seL4 verification project required approximately 20 person-years of work, and produced over 100K lines of proof scripts, compared to only about 10K lines of C implementation and merely 3K lines in its simplest abstract specification in Isabelle [5]. Moreover, developing such specifications and proofs necessitates specialized expertise in both theorem proving and the target domain, a combination that is challenging to acquire.

In this work, we focus on the proof generation step in system-level software verification projects. Recently, large language models (LLMs) have demonstrated encouraging mathematical reasoning capabilities, which inspire research into their application for software verification, particularly for proof generation [7, 8, 9, 10]. Unfortunately, both prior studies and our initial experiments show that, even when theorem statements are well formalized and relevant libraries are fully provided in advance, current LLMs still struggle to generate complete, correct proofs. We summarize two challenges as follows.

❶ *Lack of specialized expertise*. The first challenge lies in that software verification often relies on numerous domain-specific lemmas and specialized proof tactics. Although modern LLMs perform well in general reasoning tasks, they are not well-versed in these specialized domains, leading to limited effectiveness when applied to proving corresponding theorems. For example, LLM4FSCQ [7] evaluated models such

*These authors contributed equally to this work.
†Corresponding authors.

as GPT-4o and Gemini 1.5 Flash on FSCQ (a Rocq-based file system verification project [11]) and achieved only 38% proof coverage. Moreover, in the larger seL4 project, which involves extensive lemma libraries and customized tactics, Selene [9] found that GPT-4 succeeded on only 20% of a benchmark theorem set, despite the fact that nearly 40% of them require one-line proofs.

❷ *Paucity of usable data*. The second challenge, which is closely related to the first, is the shortage of high-quality (training) data for software verification. For instance, the seL4 project contains only around 20K theorems and lemmas, and most proofs are written in a procedural style, where much of the reasoning is implicit and encoded in the interaction with the theorem prover. This substantially limits their usefulness in effective LLM training/fine-tuning. Among existing seL4 benchmarks, Selene [9] extracts a small subset of the project, targeting relatively easy theorems with proof lengths of one to five. FVEL [10] is more comprehensive and challenging, as it extracts all theorems to be proved from seL4. Yet even with systematic fine-tuning of Mistral-7B-Instruct [12] and Llama-3-8B-Instruct [13], FVEL reported a success rate below 10% (88 out of 1,077 theorems in the test set).

Approach. In this paper, we propose a proof generation framework that synergistically combines LLMs and ITPs, aiming for automated system-level software verification. Our approach adopts a proof step-based tree-search pipeline that iteratively queries a fine-tuned LLM to predict the next proof step until the target theorem is successfully proved.

To address the first challenge, our framework adopts a neuro-symbolic approach tailored to the nature of software verification. First, to address challenges posed by domain-specific lemmas and tactics, we incorporate a proof-step revision module that repairs or refines LLM-generated steps whenever they fail to compile or make no progress. Second, to efficiently prune unpromising branches, we utilize Nitpick [14] and QuickCheck [15] to "test" each candidate proof state and discard those that admit counterexamples; the remaining candidates are then ranked by the LLM according to their plausibility. Finally, if the tree search still fails to resolve the goal, Sledgehammer [16] acts as a backstop that scans the local library for potentially relevant lemmas and attempts to complete the current proof.

To address the second challenge, we construct our training data by extracting internal proof states from existing theorems. For example, a five-step proof yields five distinct training instances, which are subsequently used for fine-tuning. During inference, our tree-search pipeline leverages this state extraction to facilitate direct interaction between the LLM and the prover, informing step predictions. While we currently train on existing human-authored proofs, this pipeline also enables future "bootstrapping", where new training data could be extracted on-the-fly from successful search paths [17, 18].

Experiments. We evaluate our framework on seL4 [5] proof construction. The results show that, with a fine-tuned language model, our approach achieves a total proof success rate of up to 77.6%, significantly surpassing prior automated proof-generation approaches for seL4. Notably, it excels, compared to the two preceding systems, at successfully handling a larger number of multi-step proofs, a challenge that has not been addressed by prior work. Moreover, our framework substantially alleviates manual proof-engineering effort, reducing expert proof-writing effort by 79.8% on average in an AI–human collaboration setting. Finally, experimental results on benchmarks from Archive of Formal Proofs projects (e.g., X86 Semantics , IEEE Floating Point, and SATSolverVerification), as well as the code verification benchmark Code2Inv (by translating its loop invariants into Isabelle theorems) confirm the strong generalizability of our approach.

Summary of main contributions. (1) We provide a neuro-symbolic framework based on LLM-enabled tree search, tailored for real-world, system-level software verification projects. (2) We implement concrete integration of LLMs and a broad range of symbolic, logic-based tools, substantially enhancing the efficacy of proof-tree search; (3) We curate training datasets and benchmarks, and carry out thorough empirical evaluations. Our work advances a new frontier of AI4Verification, an emerging area focused on applying cutting-edge AI techniques to hardware/software verification, especially to support large-scale proof engineering for safety-critical systems and architectures.

2 Problem

To demonstrate why existing LLMs struggle to synthesize proofs in real-world software verification tasks, we present a representative theorem from seL4. We choose seL4 as it is one of the most widely recognized formally verified systems, with machine-checked proofs of both functional correctness and security properties in Isabelle/HOL.

An example from seL4's verification is demonstrated in Figure 1, where the *theorem* `cap_swap_valid_objs` is drawn from the invariant proofs of the abstract specification of the seL4 microkernel, and similar proof patterns recur throughout the project. To prove this theorem, human experts write down (in Isabelle) three *proof steps*, each comprising one *tactic* (e.g., `simp` and `wp`) and some used *lemmas* (e.g., `cap_swap_def` and `set_cap_valid_objs`; cf. the left panel in Figure 1).

Unlike formal mathematics, theorems in software verification are often formulated as Hoare triples `{P} C {Q}`, where `P` and `Q` are the pre- and post-conditions, and `C` denotes the program being executed. In seL4, the abstract behavior of kernel operations is modeled in a monadic style that yields imperative-like programs; the corresponding Hoare rules capture the semantics of these monadic computations.

Intuitively, the targeted theorem states that if the address space of the seL4 microkernel contains no dangling pointers (`valid_objs`), then swapping the contents of two valid capa-

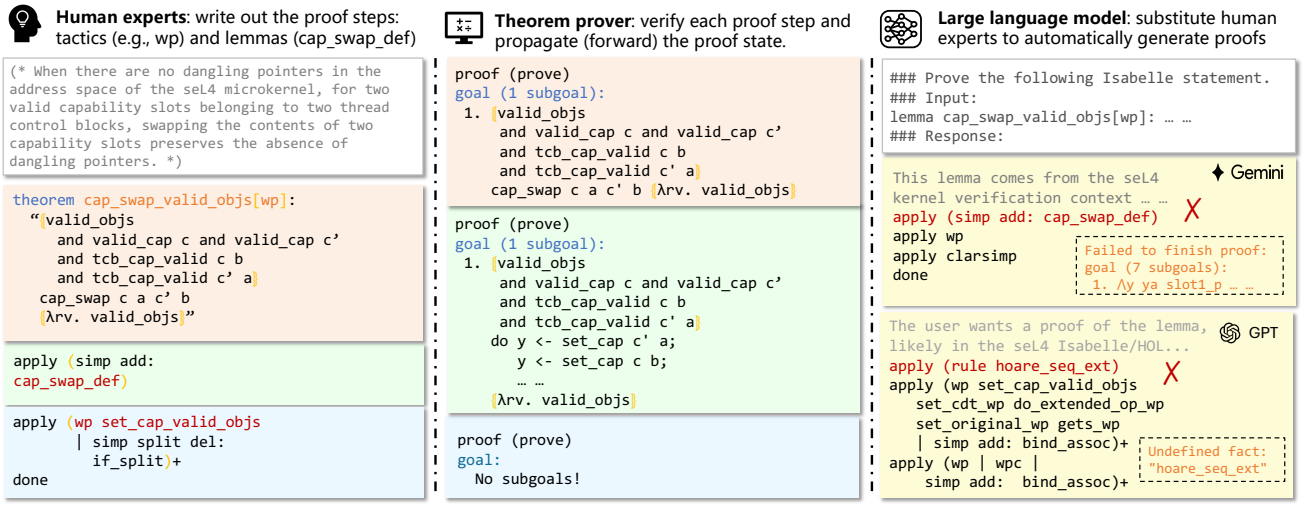


Figure 1: An example theorem and its human-written proof from the seL4 project, shown alongside LLM-generated attempts. The human-written Isabelle proof illustrates the procedural, tactic-driven reasoning style that is common in software verification. We prompted two state-of-the-art LLMs, Gemini 3 Pro and GPT-5.1, to synthesize the same proof under controlled conditions, with web search disabled to avoid any leakage of the original script. Although both models correctly recognized the proof context and invoked some relevant tactics, neither succeeded in producing a valid proof even after roughly one minute of internal reasoning. Their attempts frequently misuse the domain-specific `wp` tactic, or hallucinate nonexistent lemmas.

bility slots `c` and `c'`, which belong to two thread control blocks (`tcb_cap_valid c b` and `tcb_cap_valid c' a`), remains dangling pointers free. Namely, after the operation `cap_swap c a c' b` which exchanges the capabilities stored in these two slots, the resulting program state still satisfies `valid_objs` expressed by the postcondition `λrv. valid_objs` where `rv` denotes the return value of `cap_swap`.

To reduce the burden of manually constructing proofs, developers typically prepare a collection of broadly applicable tactics and lemmas, dedicated to software verification. For example, the `wp` tactic, short for weakest precondition, generates the verification conditions associated with a given Hoare triple. During this process, `wp` systematically applies a suite of previously proved lemmas, each of which encodes a weakest-precondition characterization for a specific construct or monadic computation. After a routine use of `simp` to unfold `cap_swap`, human experts can then easily invoke the pre-defined tactic `wp` together with the lemma `set_cap_valid_objs`.

In this example, the target theorem is discharged by combining the tactics `wp` and `simp`. The simplification step performs splitting, but the usual splitting rule for `if` is disabled to avoid unnecessary case distinction. In the proof script, the symbol `|` indicates that the tactic may apply either branch (or both), while the trailing `+` instructs it to repeat this process until no further progress is possible. This interleaving enables `wp` to generate obligations that `simp` can reduce, while `simp` reshapes the goals to facilitate subsequent `wp` reasoning. In this case, the combined step eliminates the goal entirely, and the proof concludes with `done`.

To assess LLMs in generating this proof automatically, we experiment with two state-of-the-art proprietary models, i.e., Gemini 3 Pro [19] and GPT-5.1 [20]; their results are shown on the right panel of the figure. While both deep-thinking models successfully recognize the seL4 proof context, they are unable to apply the domain-specific tactic `wp` correctly. They also hallucinate lemmas that do not exist in the library (e.g., `hoare_seq_ext`). We further attempted several rounds of iterative repair by feeding back the ITP's error messages, but the regenerated proofs continued to fail verification. An analysis of the models' reasoning traces shows that they frequently misinterpret the functionality and performance characteristics of the `wp` tactic and related lemmas (such as `set_cap_valid_objs`).

There are two potential approaches to addressing this issue. The first is to use retrieval-augmented generation [8, 21]. In this paradigm, given a target theorem, a retrieval module searches for relevant or structurally similar theorems and proofs, which are then supplied to the LLM as contextual guidance. However, this strategy depends on both a high-quality proof corpus and an effective retrieval model: requirements that are difficult to satisfy in our setting. For example, in seL4, when developers introduce a new module (such as SysInit, as shown in our evaluation) together with its accompanying proofs, comparable proofs are often absent from existing libraries. Moreover, even when analogous theorems do exist, neither model-agnostic nor model-dependent retrieval techniques reliably identify them within a large verification codebase [21, 22]. In such cases, weak retrieval can degrade, rather than improve, the quality of generated proofs.

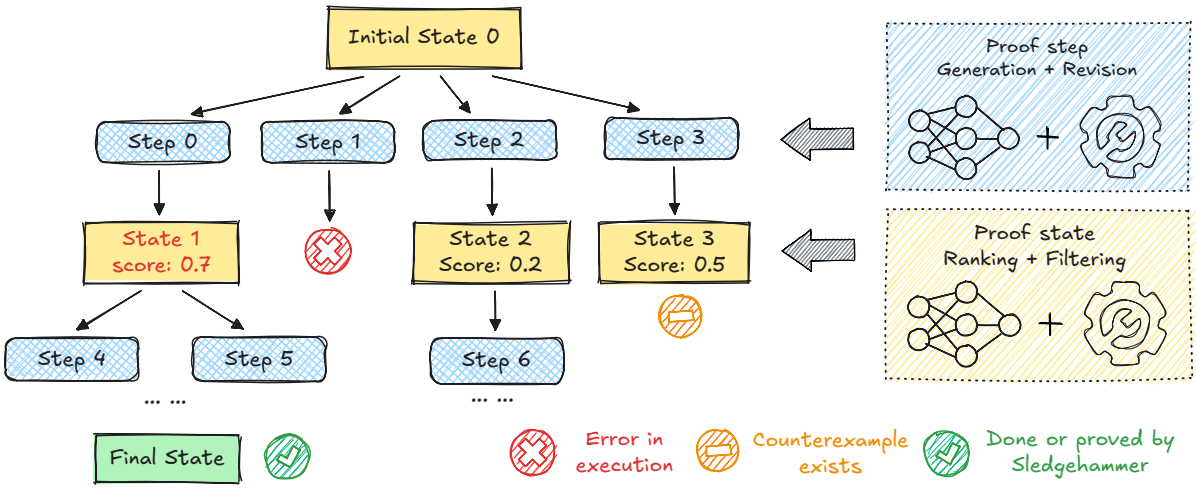


Figure 2: Overview of our neuro-symbolic proof-search framework. Starting from the initial proof state, the system repeatedly generates candidate proof steps using a fine-tuned LLM, followed by rule-based symbolic revision to repair syntactic or semantic issues in the proposed steps. Each accepted step is executed by Isabelle/HOL to derive new proof states, which are then filtered by symbolic tools to prune states containing counterexamples or duplicates. The remaining states are ranked using cumulative LLM log-probabilities, and the highest-scoring ones are selected for further expansion in a best-first search. If the search cannot complete the proof, the framework finally invokes Sledgehammer to attempt to resolve the remaining goals.

Another approach is to fine-tune LLMs on collected proof data. This method is considerably easier to implement and has already demonstrated promising results in formal mathematical proving. However, fine-tuning a model to directly generate complete proofs for software verification remains highly challenging.

First, even in seL4, the available corpus consists of only about 20K theorems, which severely limits the effectiveness of fine-tuning. Second, unlike formal mathematics, most proofs in systems like seL4 are procedural: as our example illustrates, their true semantics reside implicitly in the theorem prover's internal proof states rather than in the proof scripts themselves. As a result, LLMs struggle to acquire the necessary reasoning signal from surface-level proof text alone, making this approach impractical for now.

3 Methodology

We propose a neuro-symbolic approach for proof step search on system-level theorems. The overall procedure is illustrated in Figure 2. Starting from the root node, which represents the initial proof state, the search alternates between proposing new steps and deriving new proof states by executing those steps within the ITP.

To construct the next set of candidate proof steps for the current proof state, we first prompt a fine-tuned LLM to generate potential steps, and then apply a rule-based revision to refine or fix any imprecise suggestions. After executing these steps and obtaining a collection of new proof states, we introduce a hybrid mechanism to identify the most promising ones for further exploration. Specifically, we apply QuickCheck [15] and Nitpick [14], which serve as counterexample generators that help detect false conjectures or invalid intermediate proof states in Isabelle/HOL, to discard any proof state that contains a counterexample. We then use the LLM's prediction score (cumulative perplexity) to rank the remaining states and select the most promising one. Finally, if the search for the proof fails, we invoke an automated tool, i.e., Sledgehammer, to attempt to complete the proof. Putting all the above together, we establish the framework in Algorithm 1.

3.1 Proof Step Generation & Revision

3.1.1 Proof Step Generation

Our approach begins by generating candidate proof steps through prompting a fine-tuned LLM. Given the current proof state, which encapsulates the available hypotheses and the target goal, we query the model to propose the next proof step using a simple instruction template:

```
### Given the following Isabelle proof state,
suggest the next proof step.
### Input:
<current_proof_state>
### Response:
<next_proof_step>
```

The rationale for focusing on the current proof state is

Algorithm 1 Proof Search Framework

Require: initial state s_0, neural generator $\mathcal{G}$, symbolic checker $\mathcal{C}$, max iterations $I_{\max}$

initialize tree T with root containing s_0
for each iteration $i \in [1, I_{\max}]$ **do**
  $n \leftarrow$ highest_score_unexplored(T)
  succStates $\leftarrow []$; failStates $\leftarrow []$
  candidates $\leftarrow \mathcal{G}(n.state)$ ▷ step generation
  for each step $\delta \in$ candidates **do**
    $s \leftarrow$ apply(δ, $n.state$) ▷ ITP execute
    if s is error state **then**
      failStates.add($(s, \delta, \log p_\delta)$)
    else
      succStates.add($(s, \delta, \log p_\delta)$)
  for each $(s, \delta, \log p_\delta)$ in REVISE(failStates) **do**
    $s \leftarrow$ apply(δ, $n.state$)
    succStates.add($(s, \delta, \log p_\delta)$)
  for each $(s, \delta, \log p_\delta)$ in succStates **do**
    if s has no subgoals **then**
      return proof script ▷ proof completed
    if $\mathcal{C}(s)$ detects duplicate or counterexample **then**
      continue ▷ state filtering
    score $\leftarrow$ combine($n.score$, $\log p_\delta$) ▷ scoring
    insert_node(T, $(s, score)$)
return FAIL

twofold. First, proof search can be modeled as a Markov decision process, in which the next step depends solely on the present state; this is also common practice in proof search and is critical for efficiency [23, 24, 25]. Second, the structure and vocabulary of the proof state provide strong cues regarding which tactics are applicable and which premises are likely to be relevant [26, 27]. Additionally, we predict a single step at a time, as one-step prediction empirically yields more reliable results than multi-step prediction.

To fine-tune the model to predict such next steps, we construct a dataset of state–step pairs extracted from existing theorems and their complete proofs. Each proof is decomposed into individual proof steps and replayed in the ITP, allowing us to record the intermediate proof state preceding every step. Using these aligned pairs, we fine-tune the LLM to generate outputs that are not only syntactically correct but also semantically sensible. Finally, since proof search demands efficient generation, we employ relatively small LLMs (e.g., 1.7B or 7B size), which are nevertheless sufficient for our specific tasks.

3.1.2 Proof Step Revision

Writing syntactically and semantically valid proof steps in an ITP (such as Isabelle) is challenging, particularly in system verification tasks that depend on numerous domain-specific

Algorithm 2 Proof-step revision and execution

Require: Failed proof state set failStates

revisedSteps $\leftarrow []$
premiseSet $\leftarrow$ established from trainingCorpora
tacticSet $\leftarrow$ established from trainingCorpora
for each $(s, \delta, \log p_\delta) \in$ failStates **do**
  if s.error is w.r.t. tactic **then** ▷ tactic repair
    $p \leftarrow$ extract referred premises in δ
    revisedSteps.add(combine(tacticSet, p))
  if s.error is w.r.t. premise **then** ▷ premise repair
    $t/p \leftarrow$ extract referred tactic/premises in δ
    $p' \leftarrow$ retrieve similar premises from premiseSet
    revisedSteps.add(combine(t, p'))
return revisedSteps

lemmas and tactics. Consequently, even a well-trained LLM may generate steps that either refer to premises or tactics absent from the current proof context, or cannot be applied to the current goals due to syntactic errors or lack of progress. Such failures substantially degrade the effectiveness and efficiency of the proof search.

Despite their incorrectness, these failed attempts often contain useful information. Continuing our running example, an LLM might identify relevant premises `cap_swap_def` but pair them with an inapplicable tactic `by`; or it may propose a promising tactic `wp` while selecting premises `hoare_seq_ext` that do not precisely fit the context.

To fully exploit these incorrect but reasonable responses from LLMs, we employ a best-effort repair procedure. Whenever the ITP rejects an LLM-generated step, we record it for post-processing and re-evaluate the revised version. After examining all proposed steps for a given proof state, we systematically transform the failed ones in two ways: (1) tactic repair: we extract the premises referenced by misused tactics and recombine them with a curated set of frequently used tactics; and (2) premise repair: we search the proof context for similar premises and substitute them where appropriate. By synthesizing additional candidate steps, these symbolic revisions help reveal prover-accepted steps, thereby strengthening the overall proof search.

For the tactic repair, we first extract all tactics from the training corpus. Given a proof step rejected by the ITP, we attempt to rewrite it by identifying the premises referenced in the failed step and recombining them with the pre-built tactic set. As to the premise repair, which explicitly targets proof steps that fail with an "undefined fact" error, we attempt to correct every LLM-generated undefined premise using an available version from the proof context. Concretely, we compare the undefined fact with all candidate premises using edit distance and select the closest matches (e.g., the top three). The failed step is then revised by substituting the matched premises for the undefined ones.

The procedure of proof-step revision is summarized in Algorithm 2. However, the revision can produce a large number of candidate steps. To alleviate this, we impose constraints to keep the search space manageable. For tactic repair, we limit the tactic set (e.g., to the 12 most frequently used tactics). For premise repair, we pre-select a compact subset of premises from the library. In particular, we use MePo (a heuristic module in Isabelle's Sledgehammer that ranks and selects premises relevant to a given goal) as a lightweight relevance filter to retrieve the 128 facts most related to the current subgoal.

3.2 Proof State Filtering & Ranking

3.2.1 Proof State Filtering

To alleviate state explosion in tree-search approaches, we reorganize the accumulated proof states before further exploration. Particularly, after generating new proof states by executing the newly produced proof steps, we first invoke two complementary symbolic tools, QuickCheck [15] and Nitpick [14], to detect potential counterexamples in the candidate states. QuickCheck performs random property-based testing on executable parts of the proof state, whereas Nitpick translates higher-order formulas into finite relational models and searches for counterexamples using Kodkod, which is a SAT-based first-order relational model finder.

Additionally, we adapt the tool SolveDirect [28] to detect potentially duplicate proof states. Initially, SolveDirect determines whether a newly stated theorem can be solved directly using an existing one. In our setting, we formalize two proof states as theorems and apply SolveDirect in both directions to determine whether they are semantically equivalent.

We use a basic theorem in seL4 as an example to illustrate proof state filtering. Consider the "signed overflow" theorem, whose formal statement is as follows.

```
theorem sofl_test:
  ‹ sint x + sint y = sint (x + y) ⟷
      drop_bit (size x - 1)
          ((x + y XOR x) AND (x + y XOR y)) = 0 ›
  for x y :: ‹ 'a::len word ›
```

The theorem claims that for two signed machine words, the value of their machine-level sum matches the sum of the mathematical values they represent. This property guarantees the correctness of the machine-word definitions on which the low-level kernel specifications depend. To prove this theorem, the LLM proposes a total of 8,445 candidate proof steps, rendering exhaustive exploration prohibitively time-consuming. Nevertheless, our filtering reveals that 44.2% of these steps lead to duplicate proof states, and 52.3% of the remaining states are unprovable due to existing counterexamples. Thus, the state-filtering process effectively eliminates unproductive effort and conserves the search budget, enabling the proof to be found much earlier.

3.2.2 Proof State Ranking

To guide the proof search efficiently, we rank all proof states and always expand the most promising ones first. To score a given proof state, we rely on the log probabilities predicted by the fine-tuned LLM. The LLM is trained to predict the next proof step; its log probability directly reflects the model's confidence in applying that step to the current proof state.

Consider a candidate proof state s_L, obtained by applying a sequence of proof steps $(a_1, \ldots, a_{L-1})$. We define its cumulative log-probability score as

$$\text{score}(s_L) = \frac{\sum_{t=0}^{L-1} \log p(a_t \mid s_t)}{L^{\alpha}},$$

where $s_1, \ldots, s_{L-1}$ are the intermediate states produced by the corresponding steps. The normalization term L^{α} mitigates the inherent bias against longer proof sequences, with the exponent α specifying the degree of mitigation [18], which we set to 1 by default. After computing the score for each proof state, we select the top-k states for further expansion where k is a hyperparameter.

3.3 Hammer Integration

Hammers are tools designed to discharge proof goals in ITPs automatically. In Isabelle, the primary hammer is Sledgehammer [16], which applies ATPs and SMT solvers (e.g., Z3, CVC5, E, SPASS, Vampire, etc) to the current goal and synthesizes corresponding Isabelle proofs. When invoked, it first retrieves a fixed number of potentially relevant premises that may aid in solving the goal. These premises, together with the goal, are translated into a form suitable for backend ATP and SMT solvers. If a backend solver succeeds, Sledgehammer attempts to reconstruct a corresponding Isabelle proof step, and, when this reconstruction succeeds, a complete proof for the current goal is returned to the prover.

Although Sledgehammer's performance is far from satisfactory for our tasks, it plays a mutually beneficial role in the tree search process. In particular, while tree search may fail to solve the original problem within a limited search budget, it often reduces the problem to a simpler form that Sledgehammer can solve, even though it cannot solve the initial goal on its own. This makes it natural to integrate the hammer into our framework. Because invoking the hammer at every search step is prohibitively expensive, we only call it when tree search fails. Specifically, upon search failure, we rank all proof states in the currently expanded tree, select a small subset with the highest scores, and pass them to Sledgehammer for a final attempt at solving the goal.

4 Evaluation

This section presents our experimental results. We use seL4 as a system-level evaluation testbed to demonstrate both the

effectiveness and efficiency of our proposed method. In addition, we introduce four additional repository-level benchmarks to further assess its generalizability. Specifically, our study aims to address the following research questions (**RQ**s):

- **RQ1 – Effectiveness**: Does our proposed method outperform existing techniques in generating proofs for seL4?
- **RQ2 – Efficiency**: How much human effort in proving seL4 can be saved using our proposed method?
- **RQ3 – Generalizability**: Can our method perform well on the verification of other software projects?
- **RQ4 – Ablation**: What is the contribution of each component within our framework?

All evaluations were carried out within a dedicated Docker environment on a Linux server with an AMD EPYC 9654 96-core processor and six high-end GPUs.

4.1 Isabelle REPL

To enable efficient interaction with Isabelle from Python, we implement a new Isabelle REPL (Read-Eval-Print Loop), whose logic builds on scala-isabelle [29] and Py4J [30]. The REPL provides a Gateway Server that exposes Isabelle components to Python clients through a unified API. This design enables programmatic interaction with Isabelle, allowing users to parse Isabelle theories, extract run-time context information, and execute proof steps incrementally.

Compared to the previously developed Isabelle REPL, PISA [31], our REPL supports more recent Isabelle versions and provides extended ML-level capabilities. A wide range of Isabelle's proof-automation tools, including Sledgehammer, Nitpick, QuickCheck, and other integrated methods, can be accessed directly through our interface. It also improves the extraction of information from proof contexts, such as variables and assumptions in proof goals, dependencies of proved theorems, and facts selected by Sledgehammer.

In addition, our REPL provides improved management of the Isabelle process, top-level state, and theories. It allows users to safely impose strict time limits on Isabelle processes, as well as to clone, restore, and switch between different proof states. Theory management is also strengthened: in large verification projects, complex theory dependencies can heavily degrade compilation efficiency. For example, in the CRefine theory of seL4, a single proof step takes more than one minute to process. To address this, we implement a caching mechanism that avoids redundant execution of unrelated theorems and proofs, significantly reducing unnecessary recomputation.

4.2 Experimental Setup

Dataset. We use the FVELER dataset [10], which comprises 29,125 theorems from seL4. The dataset is split into four subsets, i.e., training, validation, test, and test-hard, containing 26,081, 1,115, 1,077, and 852 theorems, respectively. Following the FVEL criteria, the training, validation, and test sets are randomly partitioned. In contrast, the test-hard set consists of theorems drawn from three specific and independent sessions, i.e., SysInit, SysInitExamples, and LibTest, that do not appear in any of the other splits. Note that *sessions* are the way Isabelle organizes theory files, which resemble the relationship between code files and libraries in other programming languages. A theory can import other theories, and a session can depend on other sessions by declaring dependencies in the Isabelle ROOT file. These sessions are typically selected based on the depth of their session-level dependency graphs, and the theorems they contain exhibit more complex dependencies, making them considerably more challenging to prove.

In our experiments, we use the training set as the corpus for fine-tuning LLMs, and adopt validation, test, and test-hard sets to evaluate our proposed framework. From the training set, we further extract all proof state–step pairs, yielding a total of 181,887 pairs. On average, each theorem contributes to about seven pairs, substantially increasing the amount of supervised data available for model training.

Training. We fine-tuned two small models, Qwen3-1.7B [32] and Mistral-7B [12], to predict and score each proof step. We carried out the full-parameter supervised fine-tuning (SFT) using the Llama-Factory framework [33]. We use DeepSpeed ZeRO-2 optimization to enable memory-efficient distributed training. The training used an effective global batch size of 16 and ran 3 epochs using bfloat16 mixed precision. The learning rate was set to 1e-5 and followed a cosine decay schedule with a warmup ratio of 0.1.

Proving. For the LLM inference, we adopt a high temperature of 1.0 and a top-p value of 0.95 to explore the proof space sufficiently. The maximum generation length is fixed at 2,048 tokens. In each search iteration, at most five proof states are selected; for each proof state, the model is prompted to generate 128 candidate proof steps. In our framework, we disable LLMs' internal thinking to improve inference efficiency.

When proof search fails, we select the 16 highest-scoring proof states and invoke Sledgehammer for assistance. We run Sledgehammer with its default premise-selection strategy: it first selects 2,048 most relevant premises for the current proof goal using MeSh, which combines MePo [34] and MaSh [35] to gauge relevance between the proof goal and available facts in the context. With these retrieved results, Sledgehammer then attempts to discharge the current subgoal using the built-in SMT/ATP solvers Z3, CVC5, E, SPASS, and Vampire, with a 60s time limit.

Baseline. We select four neural or symbolic proof-generation methods as baselines. Selene prompts GPT-4o with the target problem and several examples to generate a candidate

Table 1: Proof success rates (%) across validation (Val), test (Test), and test-hard (TsHd) splits. Compared to existing neural and symbolic provers, our framework delivers substantial improvements: Qwen3-1.7B achieves 70.4% overall success, and Mistral-7B reaches 77.6%, exceeding the strongest baseline by 30.0 and 37.3 percentage points, respectively.

Method		Val	Test	TsHd	Total
Neural	Selene	6.1	7.0	3.3	5.6
	FVEL	8.9	9.5	4.5	7.8
Symbolic	Auto	4.9	6.7	6.1	5.9
	Hammer	40.5	39.5	40.9	40.3
Ours	Qwen3	73.6	74.9	61.1	70.4
	Δ	↑33.1	↑35.4	↑20.2	↑30.0
	Mistral	**79.8**	**89.0**	**69.8**	**77.6**
	Δ	**↑39.3**	**↑49.5**	**↑28.9**	**↑37.3**

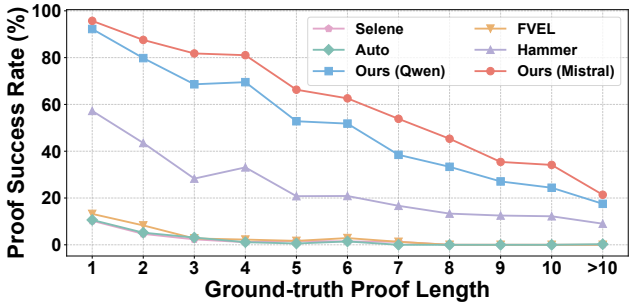


Figure 3: Proof success rate across ground-truth proof lengths. Our approach consistently outperforms existing methods. In particular, the success rate decreases as proofs grow longer, but our approach still maintains non-negligible performance even on proofs exceeding 10 lines.

proof. FVEL follows a similar paradigm but also fine-tunes a Mistral-7B-Instruct model on the training data to produce complete proofs, rather than using in-context learning. We also include two symbolic methods, Auto and Sledgehammer, which operate purely symbolically to prove the target theorem. Here, Auto is a collection of common symbolic automatic proving methods in Isabelle, such as `by simp`, `by auto`, and `by fastforce`. The time limit for all methods is set to 120 minutes, as we observed no noticeable improvement when increasing it further.

4.3 Experimental Results

4.3.1 RQ1: Effectiveness Comparison

Table 1 summarizes the performance of our approach compared to both neural and symbolic baselines. Among neural methods, Selene achieved 156 successful proofs, while FVEL improved this to 219. As for symbolic baselines, Auto proved 164 theorems, whereas Sledgehammer reached 1,124. In contrast, our framework substantially outperforms all these baselines across different splits. Using the fine-tuned Mistral-7B model, our approach produces a total of 2,167 successful proofs, of which 788 are on the validation set, 811 are on the test set, and 568 are on the test-hard set. This accounts for 77.6% of all still valid theorems across the three evaluated sets, 37.3 percentage points higher than the best baseline, Sledgehammer. It is also encouraging that, on a relatively small model, Qwen3-1.7B, our approach manages to complete the proof of 1,965 theorems, which amounts to 70.4% of the total theorems. This demonstrates that in a real environment where new proofs need to be crafted, a deployed small model can already help with the development of verification.

We further evaluate the performance of our automated proving approach in two aspects. First, prior work struggled to generate nontrivial proofs. Since the proofs in seL4 are largely procedural, we use the proof length as a proxy for proving difficulty. We compute the proof success rate across different proof lengths and present the results in Figure 3. The results show that our approach is more capable of proving longer theorems than previous baselines. Although the success rate naturally decreases as proof length increases, it remains around 20% even for the 393 theorems whose proofs exceed 10 lines. We leave it as future work to investigate more effective strategies for constructing longer proofs.

Second, we examine the performance across different proof sessions. We group the seL4 theorems into six major categories w.r.t. the functionality [5, 36, 37]: Base Libraries & Tools (Base), Specifications (Spec), Abstract-Level (A-Level) Properties, Refinement from Abstract-Level to Executable C (A→C), Security/Information Flow Control (IFC), and System Initialization Group (SysInitGroup). Figure 4 presents the results across these categories, demonstrating that our approach delivers superior performance compared with all baselines across all sessions. Notably, under the test-hard set division strategy, SysInitGroup is derived from a completely distinct session and is isolated from the training data, yet our approach still achieves a success rate of 67.6%.

Response to RQ1

Our approach substantially outperforms baselines, achieving a success rate 37.3 percentage points higher than the (symbolic) ATP tool Sledgehammer and 69.8 percentage points higher than the state-of-the-art neural method. It remains reliably effective across different proof difficulty levels and session categories. Further, it demonstrates good generalizability, successfully discharging 67.6% of the theorems from unseen sessions during training.

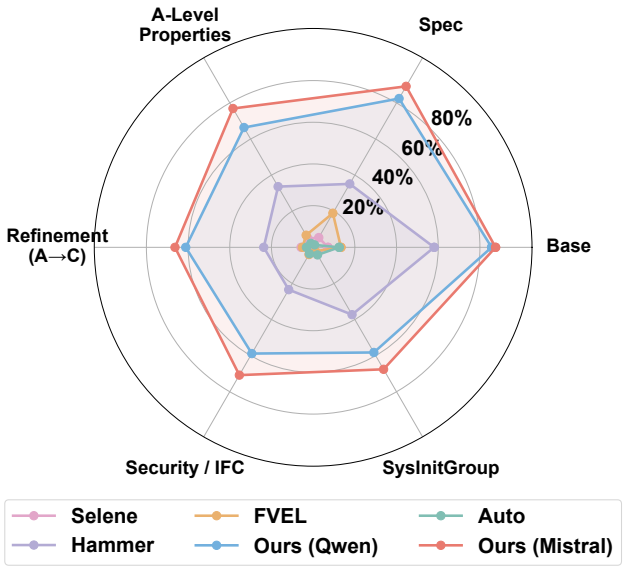


Figure 4: Proof success rates across the six seL4 session categories. Our approach yields stronger performance than all baselines in all sessions. Notably, it achieves high accuracy even on the challenging SysInitGroup category, which is entirely unseen during training.

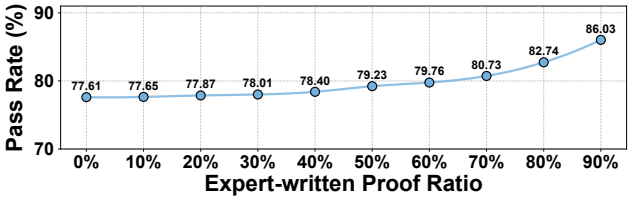


Figure 5: Proof completion rate under varying proportions of provided proof lines. Our method is increasingly effective as more human-written lines are supplied. Overall, the method reduces human effort by 79.8% on average.

4.3.2 RQ2: Proving Efficiency in Saving Human Efforts

To evaluate how effectively our approach reduces manual effort in developing proof scripts, we adopt a standard metric: the coverage of the entire library that can be automatically generated [38]. As summarized in Table 2, our method successfully proves theorems whose corresponding proofs account for 6,235 lines in the seL4 proof corpus, achieving 36.2% coverage across all evaluated theorems. Under this metric, our approach yields more than a one-fold improvement over existing baseline methods.

The coverage metric above credits only fully automated proofs. To also capture partial assistance, we simulate an AI–human collaborative workflow, where our framework completes proofs after several proof lines have been written by human. This experiment depicts a scenario where human experts can initially provide some reliable proof directions, and

Table 2: Coverage results of automatically generated proof lines. Our method attains the highest coverage, proving theorems whose existing proofs amount to over one-third of all proof lines and substantially outperforming existing baselines in reducing manual proof-script development effort.

Method	# of proof lines	% of coverage
Selene	197	1.1
FVEL	271	1.6
Auto	204	1.2
Hammer	2,581	15.0
Ours	**6,235**	**36.2**

then AI attempts to complete the subsequent proof tasks. In practice, for each theorem, we provide a prefix taken from the ground-truth proof (the first σ fraction of expert-written lines from the dataset) and ask our approach to complete the remaining steps, starting from the resulting proof state reached after executing the provided lines.

We then define an effort saving metric η for this experiment. Specifically, if our approach proves a theorem with at least the first σ fraction proofs provided, the effort saving is then $1-\sigma$. For instance, consider a theorem whose ground-truth proof consists of 10 lines: if our framework cannot complete the proof when the first two lines are provided, but succeeds when the first three lines are provided ($\sigma = 0.3$), the effort saving for this theorem is 0.7. We compute the average effort saving under this setting as:

$$\eta = \sum_{i=1}^{k} (p_i - p_{i-1}) \cdot (1 - \sigma_i), \quad p_0 = 0,$$

where $(\sigma_1, \ldots, \sigma_k)$ are the prefix fractions in ascending order, and p_i is the fraction of theorems whose remaining proof can be completed automatically given a σ_i-prefix; theorems unsolved at every evaluated σ_i contribute zero saving.

Figure 5 plots the proof completion rate over the prefix fraction σ. Consistent to the intuition, as the prefix size grows, more theorems become completable. This yields $\eta = 79.8\%$, meaning our approach reduces human proof-writing effort by roughly 80% on average in this setting.

We also record the time consumption of our framework as a reference; the average time to generate a successful proof is 139.1 minutes. However, it is worth noting that this average is skewed upward by a small number of theorems that take extremely long to prove. In our experiment, we find that 58.4% of the theorems are proved within 10 minutes, 73.4% within 30 minutes, and 80.8% within 2 hours. Therefore, the majority of proofs are obtained within minutes to tens of minutes—substantially faster than manual proof development, which often requires hours to days per theorem for complex system properties.

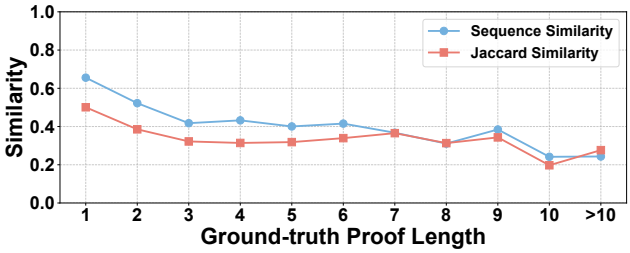


Figure 6: Sequence and Jaccard similarity between generated and ground-truth proofs across proof lengths. Both metrics remain low and decrease as proofs grow longer, suggesting the model synthesizes novel strategies rather than recalling memorized proofs.

Response to RQ2

Our approach alleviates much of the burden of interactive theorem proving, automatically completing proofs that correspond to 36.2% of the proof lines in the evaluated corpus and reducing expert effort by approximately 79.8% in an AI–human collaboration setting. Moreover, it is able to synthesize the majority of proofs within a practical time budget.

4.3.3 RQ3: Generalizability to Other Projects

A major concern with training-based methods is their generalizability, namely, whether the approach is only applicable to a specific domain due to potential data leakage or overfitting.

To investigate possible leakage, given that the seL4 proofs are publicly available and that the pre-trained model may have already incorporated some of the knowledge, we compute the similarity between the ground-truth proofs and the proofs generated by our approach using sequence similarity [39] and Jaccard similarity [40]. As shown in Figure 6, both metrics remain low and, crucially, decrease as proof length grows. If the model were memorizing ground-truth proofs, we would expect similarity to remain high regardless of length; the observed downward trend instead suggests the model is synthesizing novel proof strategies rather than recalling memorized ones. The only exceptions are one- or two-line proofs, where the solution space is inherently small—yet even among these, only 16.7% are identical to the ground truth.

We further explore the generalizability of our approach and the fine-tuned LLM. We select three projects from the Archive of Formal Proofs (AFP*), i.e., X86 Semantics [41], IEEE Floating Point [42], and SATSolverVerification [43], to directly apply proof search on the theorems therein. In addition, we include the code verification benchmark Code2Inv and translate its loop invariants into Isabelle theorems.

*https://www.isa-afp.org/

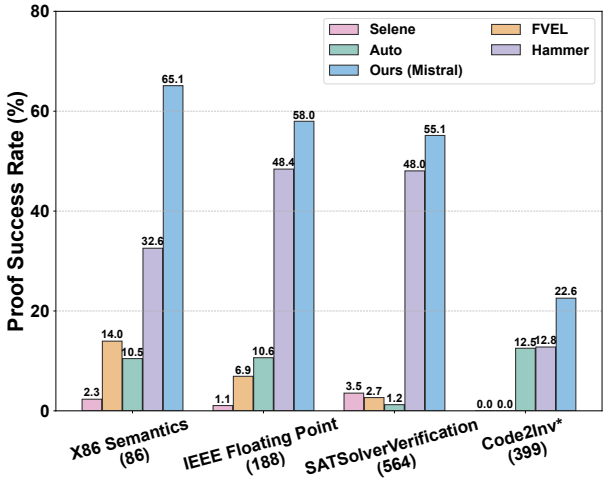


Figure 7: Performance of our proof search framework on additional benchmarks. The x-axis lists the benchmarks, where entries in brackets denote the number of theorems included in each benchmark. Our approach consistently achieves higher proof success rates across all benchmarks, demonstrating strong generalizability and robustness.

The performance of our approach and baselines is shown in Figure 7. Because the X86 semantics benchmark is more relevant to the seL4 project, the fine-tuned LLM performs particularly well on it, yielding a 32.5% higher success rate than the baselines. The other three benchmarks are hammer-friendly because of their SMT-oriented nature. Even in this setting, our framework demonstrates superior effectiveness, achieving an average relative improvement of 36.9%.

Response to RQ3

The knowledge acquired by the fine-tuned LLM, together with our proof search framework, generalizes well across domains. In particular, the approach can be effectively leveraged to assist proof construction in a wide range of formal verification tasks beyond the seL4 setting.

4.3.4 RQ4: Ablation study

Since our approach integrates several components, namely LLMs, hammer, as well as proof-step revision and proof-state filtering mechanisms, we conduct ablation studies to evaluate the contribution of each part of the framework. To assess the impact of our fine-tuned model, we replace it with off-the-shelf LLMs such as GPT-4o or DeepSeek. Due to budget constraints (running tree search costs approximately $4.1 per theorem on average when using the DeepSeek or GPT-4o APIs), we randomly sample 200 theorems from the benchmark and use the DeepSeek API to propose proof steps.

In addition, inspired by Rango [8], we implement a retrieval-augmented generation (RAG) mechanism. The retrieval module fetches both similar proof steps and potentially relevant premises. Similar steps are retrieved from the training set based on the similarity between their associated proof states and the current proof state, where these states are tokenized into sequences of identifiers, and relevance is measured using BM25 [44]. For premises retrieval, we continue to rely on Sledgehammer's MePo heuristic for efficiency. The final prompt for LLMs includes the retrieved steps and premises along with their statements, and we restrict the total lengths of steps and premises to 1024 and 512 tokens, respectively.

The results for all method variations are reported in Table 3. Overall, although adding tree search and RAG steadily improves DeepSeek's performance, these variants still fall significantly short of our full framework. This confirms that the combination of the fine-tuned model, structured search, and hammer integration is essential for achieving high proof success rates, and highlights the importance of each major component in our design.

We also evaluate the impact of the fine-tuning process by adopting a hammer-free tree search setting to focus on the quality of proof steps generated by the model itself. As shown in Table 4, vanilla base models such as Mistral-7B and Qwen3-1.7B struggle with formal proof tasks, yielding negligible success rates even when augmented with tree search. This underscores the necessity of our domain-specific fine-tuning.

To evaluate the impact of the step revision mechanism, we analyze the theorems that become difficult to prove when the revision is disabled. We observe that revision is most beneficial when the LLM fails to generate enough effective proof steps. In particular, for the most challenging 77 theorems where the model initially proposed no valid steps and therefore failed immediately, the revision procedure is highly effective, enabling 24.7% of these theorems to be proved. For a broader set of 220 theorems where the LLM proposed fewer than 5% valid steps, the revision mechanism still provides substantial gains, allowing 11.8% of them to succeed.

Regarding the necessity of proof state filtering, particularly the counterexample detection, our experiments show that incorporating this component allows the system to identify an average of 1.3 counterexamples per evaluated theorem. For the subset of search traces that contain counterexamples, the average number rises to 11.1. These results indicate that counterexample detection is valuable for identifying unprovable states early and preventing unproductive search efforts, although it does not directly increase the overall success rate.

Table 3: Proof success rate (%) on several variants of our framework. The proof success rate increases progressively as tree search, retrieval augmentation, and hammer are incorporated, indicating the necessity of each component. The comparison highlights that our full approach, which replaces the off-the-shelf model with a fine-tuned LLM, achieves substantially stronger performance.

Method Variation	**Rate**
DeepSeek (DpSk) v3.2	5.5
DpSk + Tree Search	12.5
DpSk + RAG + Tree Search	33.0
DpSk + RAG + Tree Search + Hammer	52.0
Ours (Trained LLM+Tree Search+Hammer)	**70.0**

Table 4: Effect of supervised fine-tuning under the same tree-search configuration. Without fine-tuning, the base models solve almost no theorems; in contrast, after supervised fine-tuning, both models reach around 60% proof success.

Method Variation	**Rate**
Mistral-7B + Tree Search	0.0
Qwen3-1.7B + Tree Search	0.1
Mistral-7B SFT + Tree Search	**59.8**
Qwen3-1.7B SFT + Tree Search	**57.1**

Response to RQ4

Each of the symbolic and neural components in our framework plays an essential role, and their synergy substantially strengthens the overall theorem-proving capability for software verification.

5 Further Analysis

Beyond aggregate accuracy, we examine how our proposed framework actually constructs successful proofs and what causes the remaining failures.

5.1 Successful Case Studies

We discuss two successful proofs that exhibit complementary patterns: a short symmetry lemma closed by an LLM-proposed case analysis combined with the symbolic tool Sledgehammer (C1), and a long Hoare-triple proof for an seL4 fault handler built entirely from LLM-generated steps (C2). For each, we show the theorem, the proof our framework discovered, and the search behavior behind it.

C1. A symmetry lemma in CSpace. The seL4 lemma `same_object_as_commute`, from the CSpace invariant de-

velopment, states that the capability-equivalence relation `same_object_as` is symmetric. The seL4 proof reduces the equality to a one-way implication via `subgoal_tac` and closes both directions with a single `auto` call. Our framework takes a different route: it splits on `c'` first and lets Sledgehammer discharge each constructor case independently.

```
lemma same_object_as_commute:
  "same_object_as c' c = same_object_as c c'"
apply (case_tac c')
apply (simp_all add: same_object_as_def split:
cap.splits)
apply blast
apply meson
apply blast
using bits_of_def apply force
apply blast
apply blast
using same_aobject_as_commute by blast
```

What the framework did. At depth 0, the LLM proposed several decompositions. Tree search explored seven of them, including `case_tac c` with various simp-set expansions, before settling on `case_tac c'` followed by `simp_all (split: cap.splits)`, which left a small frontier of constructor cases. Sledgehammer then closed the leaves with single-line `blast`, `meson`, and `force` calls, together with the previously proved `same_aobject_as_commute`.

C2. Invariance under VM-fault handling. The lemma `hv_inv_ex`, from the ARM `ArchSyscall_AI` session, is a top-level Hoare triple for seL4's VM-fault handler: it asserts that an arbitrary precondition P is preserved along the exceptional return path. The seL4 proof is a single composite tactic that relies on dedicated `*_inv` weakest-precondition lemmas:

```
lemma hv_inv_ex:
  "{|P|} handle_vm_fault t vp {|λ_ _. True|}, {|λ_. P|}"
  by (cases vp;
      wpsimp wp: dmo_inv getDFSR_inv getFAR_inv
                 getIFSR_inv getRestartPC_inv)
```

Our framework arrives at a substantially different proof. It does not use any of the `*_inv` lemmas; instead, it rebuilds the argument as a 29-step script of `wp`/`wpsimp` calls, `clarsimp` unfoldings, and four explicit postcondition weakenings via `hoare_post_imp`. Sledgehammer is never invoked: every step is produced by the LLM and validated by the Isabelle kernel. The proof below is abridged.

What the framework did. After the initial `cases vp` split, the proof unfolds along three near-parallel branches, one per fault sub-handler (`getDFSR`, `getFAR`, `getIFSR`). Tree search repeatedly favored `wp`/`wpsimp` steps interleaved with `clarsimp` unfoldings as the symbolic state became exposed. At four points, the LLM proposed intermediate postconditions through `rule_tac P=... + hoare_post_imp`, which were essential to keep the proof state tractable. None of the 29 steps required Sledgehammer.

```
lemma hv_inv_ex:
  "{|P|} handle_vm_fault t vp {|λ_ _. True|}, {|λ_. P|}"
apply (cases vp, simp_all)
apply wp
apply (clarsimp simp: do_machine_op_def getDFSR_def)
apply wp
apply (case_tac rv)
apply clarsimp
apply (rule_tac P="P and (λx. snd (aa,ba) =
       machine_state x)" in hoare_post_imp)
(* ... 20 proof lines omitted ... *)
apply clarsimp
apply (drule in_inv_by_hoareD [OF gets_inv])
by simp
```

Summary of successful cases. In both cases, the framework reaches the theorem through a proof structure that differs from the seL4 script: decomposing the equality on the opposite side in C1, and expanding a one-line composite tactic into a 29-step derivation in C2. These examples show that the system genuinely composes LLM generation, symbolic validation, and Sledgehammer integration on non-trivial seL4 goals, rather than replaying memorized scripts. Further examples are available in our GitHub repository.

5.2 Failure Analysis

We next turn to the failures. Across the three evaluation splits, 692 theorems remain unsolved (validation: 241, test: 199, test-hard: 252). We study them at two different granularities. At the step level, our *tactic-error* analysis draws on the 7.77M individual tactic attempts logged for the 241 failed theorems on the validation split, where each rejected step records the underlying Isabelle kernel error. At the run level, our *search-outcome* analysis draws on the 451 failed runs from the test and test-hard splits, whose logs record the maximum search depth reached before the budget was exhausted.

Step-level error categories. Of the 7.77M tactic attempts on the validation failures, 30.1% advance or duplicate the proof state, while 69.8% raise a kernel error. Table 5 groups these errors into 12 categories, two of which dominate. The first is *lemma hallucination*, which combines the `undefined-fact`, `undefined-method`, and `undefined-constant` categories and accounts for 61.1% of all errors. The second is *shape mismatch*, captured by `tactic-inapplicable`, which contributes another 17.9%. Together, these two patterns explain 79% of rejected tactics: the model most often cites premises absent from the active context, and otherwise produces well-formed tactics that do not fit the current goal shape. The remaining categories, including schematic-variable mistakes, syntax errors, type errors, tactic timeouts, and unfinished `by` closures, each contribute under 3%.

Search-level outcomes. Table 6 shows how far search gets before exhausting its budget. Only 3.7% of failures stall by depth 10. Most failures therefore make real multi-step progress (≥ 11 depth in 96% of cases) but run out of the 128-

Table 5: Step-level error breakdown over all tactic attempts on the validation failures (241 theorems, 7.77M attempts). Undefined-fact alone accounts for nearly two-thirds of all errors: the LLM repeatedly cites premises that do not exist in the active context.

Error category	Count	%	Error category	Count	%
Undefined fact	3,263,494	60.1	Bad method args.	65,430	1.2
Tactic inapplicable	976,398	17.9	Inner syntax error	50,033	0.9
Multi-line / other	423,982	7.8	Undefined method	47,163	0.8
Unknown variable	161,842	2.9	Proof unfinished	37,100	0.6
Outer syntax error	153,037	2.8	THM exc.	26,750	0.4
Type unification	110,717	2.0	Undefined constant	8,852	0.1
Tactic timeout	102,099	1.8			
			All errors	5,427,419	100

Table 6: Search-depth distribution for failures on the test and test-hard splits (the validation log format does not record depth markers). Only 3.7% of failures stall before depth 10; most failures make substantial multi-step progress and run out of budget later in the search.

Max search depth reached	Theorems	%
0 (no progress past initial state)	11	2.4
1–10 (early stall)	6	1.3
11–30 (medium-depth stall)	109	24.2
31–60 (deep stall)	181	40.1
>60 (very deep)	145	32.0
Total	451	100

attempt budget later in the search. On the validation split, where QuickCheck signals are logged, 23 of 241 failed theorems (9.5%) contain at least one search node with a genuine counterexample. These cases suggest that the extracted obligation is unprovable in the active context, typically because an essential assumption is dropped when proof states are sliced from the original seL4 scripts to form the benchmark; we treat them as benchmark-side issues rather than framework failures. Sledgehammer was invoked on all 241 validation failures that still had a residual goal but closed none of them, indicating that the remaining goals lie outside the reach of off-the-shelf SMT/superposition.

Summary of failure modes. The two analyses converge on the same bottlenecks. At the step level, wasted budget is dominated by missing premises ($\sim$61% of errors) and goal-shape mismatches ($\sim$18%). At the run level, most failed runs do not stall early; they reach substantial depth and then exhaust the budget on residual goals that Sledgehammer cannot close either. We therefore see two clear directions for future work: better premise grounding to reduce lemma hallucination, and longer-horizon planning to prevent late-search collapse on deep proofs.

6 Related Work

6.1 Software System Verification via ITP

Formal methods have successfully verified critical software systems including operating system kernels, hypervisors, file systems and compilers. Interactive theorem proving (ITP) provides strong formal assurance by allowing engineers to encode specifications and build proofs in ITP systems such as Rocq or Isabelle/HOL. Notable examples include CertiKOS [45], which employs a compositional refinement framework to verify a layered OS kernel; CompCert [4], an optimizing C compiler whose translation passes are all verified to preserve semantics; FSCQ [11], a crash-safe file system proven correct under all failure scenarios using Crash Hoare Logic; SeKVM hypervisor [46], a reduced KVM core with enforced confidentiality and integrity of guest memory; seL4 [5], an OS microkernel with machine-checked proofs of functional correctness and security properties. These efforts demonstrate that ITP enables high-assurance proofs for critical systems, albeit at a significant engineering cost.

Another line of work seeks to reduce this cost by automating more of the verification pipeline itself. AutoCorres [47] synthesizes high-level Isabelle specifications from C to ease low-level proof writing within the seL4 verification effort. More recently, Spoq [48] automates the derivation of layered specifications and refinement proofs from unmodified C code, reducing manual Coq effort by around 70% on the SeKVM hypervisor [46]. Spoq 2 [49] extends this direction to security properties, reducing them to inductive invariants over a transition system that Z3 can discharge automatically, and reports up to 80% reduction in manual effort across four real-world systems.

6.2 Automated Theorem Proving

Automated theorem proving (ATP) has long served as a benchmark for machine reasoning. Classical ATP systems rely on symbolic methods, such as resolution-based first-order provers (e.g., Vampire [50], E [51]) and SMT solvers (e.g., Z3 [52], CVC5 [53]) with built-in theories. These solvers are also integrated into interactive theorem provers via hammers (e.g., Sledgehammer [16] in Isabelle), enabling automated discharge of proof goals [54]. Related to ATP, several works (e.g., Ironclad Apps [55] and Verve [56]) leverage verification-aware languages (e.g., Dafny [57] and Boogie [58]) to prove memory and control-flow safety of system components, including kernels and device drivers.

In recent years, machine learning has been increasingly applied to ATP [59]. Early efforts focused on premise selection and tactic prediction using classical learning methods (e.g., Holophrasm [60], TacticToe [26], Proverbot9001 [61], Diva [62]), showing improved success rates in Metamath, HOL4, and Coq. More recently, large language models

(LLMs) have been used to generate complete proofs or sketches [63, 64], achieving near-superhuman results on mathematical problems [65]. Notably, AlphaProof [66] combines Monte Carlo tree search with reinforcement learning to solve IMO-level competition problems in Lean, representing the state of the art in search-based mathematical theorem proving.

Closer to our setting, prior efforts have attempted to bring such techniques to software verification, including Baldur [67], PALM [68], Guided Proof Search [69], and Goedel-Code-Prover [70]. However, these approaches remain limited to textbook-scale problems and do not address the challenges of real-world system-level verification projects such as seL4.

Complementary to whole-proof generation, the proof-step prediction paradigm harnesses LLMs to propose the next tactic or lemma, guided by proof-state feedback. Systems such as GPT-f [71], LISA [31], HTPS [72] and LeanDojo [21] demonstrate the potential of this paradigm across Metamath, Isabelle and Lean, but again primarily on isolated mathematical problems. Our method fills this gap by integrating a fine-tuned LLM for proof search with symbolic techniques, namely proof-step revision and proof-state filtering, making large-scale verification both effective and efficient.

Hybrid approaches such as Thor [73], ProofAug [74], Strat2Rocq [75], and LIPS [76] couple LLM-generated reasoning with ATPs or hammers. However, hammer-only proofs grow brittle on long obligations and place additional trust in the external solver. In contrast, our framework invokes ATP only as a final-stage closer for residual subgoals, keeping the bulk of the proof inside the kernel-checked workflow.

7 Limitations and Future Work

Despite strong empirical performance, our framework has several limitations that suggest directions for future research.

Computational cost. The tree-search procedure remains expensive, as each iteration invokes Isabelle for proof-step execution and state checking. Future work includes state caching, parallel expansion, and cost-aware search policies to improve throughput.

Performance degradation for longer proofs. While our method successfully constructs longer proofs than prior methods, its performance indeed degrades as proof length increases. The current tree-search procedure adopts one-step prediction and bounded search, making it challenging to uncover deeper reasoning chains. Future improvements could involve hierarchical planning, subgoal decomposition, or reinforcement learning from bootstrapped trajectories.

Rigid rule-driven step revision. The step revision modules rely on heuristic tactic-premise recombination and lightweight premise retrieval, which are imperfect and may introduce misleading candidates. Future improvements may learn premise relevance directly from the proof state or integrate stronger semantic filters.

Leveraging frontier models. As frontier LLMs advance in reasoning, an important direction is whether agentic workflows—where a large model autonomously invokes tools, inspects proof states, and iterates—can complement or replace parts of our pipeline. Our use of fine-tuned small models is motivated by lower cost and higher inference throughput, both critical for generating and scoring large candidate pools. A promising hybrid would delegate high-level planning and subgoal decomposition to a frontier model, directly addressing the degradation on longer proofs, while relying on lightweight models for intensive step generation and ranking.

8 Conclusion

In this paper, we present a neuro-symbolic proof-generation framework that automates proof search for large-scale software verification projects in interactive theorem provers. Instead of attempting whole-proof synthesis, our approach performs a best-first tree search over fine-grained proof states, repeatedly querying a lightweight, fine-tuned LLM for the next proof step. This paradigm not only substantially improves data efficiency, but also enables a seamless integration of symbolic reasoning tools with LLM-based step generation and scoring, forming an effective and efficient pipeline for proof-step construction as well as proof-state filtering and ranking. The experiments on the seL4 verification corpus show that our approach significantly outperforms prior baselines. Notably, our method succeeds on a considerably higher proportion of multi-step, nontrivial theorems, addressing a long-standing limitation of LLM-based provers. Additional experiments on AFP projects and translated program-verification benchmarks further showcase strong cross-project generalizability.

Our work provides evidence that proof-step–centric neuro-symbolic search is a practical path toward scalable, automated software verification at the systems level. While challenges remain, our results represent promising progress toward AI-enabled, rigorous engineering methods for real-world computer systems.

Availability. The tool, experimental data, and source code are available at the seL4-proof-search GitHub repository. Additionally, a prebuilt Docker image containing the seL4 verification environment and all required dependencies is available at Docker Hub.

Acknowledgments. We thank the anonymous reviewers and our shepherd for their valuable and constructive feedback. This work is supported by the Fundamental and Interdisciplinary Disciplines Breakthrough Plan of the Ministry of Education of China (No. JYB2025XDXM118), the National Natural Science Foundation of China (Grant #62025202), and the Frontier Technologies R&D Program of Jiangsu (BF2024059). T. Chen is partially supported by overseas grants from the State Key Laboratory of Novel Software Technology, Nanjing University (KFKT2023A04, KFKT2025A05).

References

[1] Edmund M. Clarke and Jeannette M. Wing. Formal methods: state of the art and future directions. *ACM Computing Surveys*, 28(4):626–643, 1996.

[2] Bruno Blanchet, Patrick Cousot, Radhia Cousot, Jérôme Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. A static analyzer for large safety-critical software. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 196–207, 2003.

[3] James Woodcock, Peter Gorm Larsen, Juan Bicarregui, and John Fitzgerald. Formal methods: Practice and experience. *ACM Computing Surveys*, 41(4):1–36, 2009.

[4] Xavier Leroy, Sandrine Blazy, Daniel Kästner, Bernhard Schommer, Markus Pister, and Christian Ferdinand. Compcert-a formally verified optimizing compiler. In *ERTS 2016: Embedded Real Time Software and Systems, 8th European Congress*, 2016.

[5] Gerwin Klein, June Andronick, Kevin Elphinstone, Toby Murray, Thomas Sewell, Rafal Kolanski, and Gernot Heiser. Comprehensive formal verification of an os microkernel. *ACM Transactions on Computer Systems (TOCS)*, 32(1):1–70, 2014.

[6] Talia Ringer, Karl Palmskog, Ilya Sergey, Milos Gligoric, Zachary Tatlock, et al. Qed at large: A survey of engineering of formally verified software. *Foundations and Trends® in Programming Languages*, 5(2-3):102–281, 2019.

[7] Jianxing Qin, Alexander Du, Danfeng Zhang, Matthew Lentz, and Danyang Zhuo. Can large language models verify system software? a case study using fscq as a benchmark. In *Proceedings of the 2025 Workshop on Hot Topics in Operating Systems*, pages 34–41, 2025.

[8] Kyle Thompson, Nuno Saavedra, Pedro Carrott, Kevin Fisher, Alex Sanchez-Stern, Yuriy Brun, João F Ferreira, Sorin Lerner, and Emily First. Rango: Adaptive retrieval-augmented proving for automated software verification. In *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*, pages 347–359, 2025.

[9] Lichen Zhang, Shuai Lu, and Nan Duan. Selene: Pioneering automated proof in software verification. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (ACL)*, pages 1776–1789, 2024.

[10] Xiaohan Lin, Qingxing Cao, Yinya Huang, Haiming Wang, Jianqiao Lu, Zhengying Liu, Linqi Song, and Xiaodan Liang. Fvel: Interactive formal verification environment with large language models via theorem proving. *Advances in Neural Information Processing Systems*, 37:54932–54946, 2024.

[11] Haogang Chen, Daniel Ziegler, Tej Chajed, Adam Chlipala, M Frans Kaashoek, and Nickolai Zeldovich. Using crash hoare logic for certifying the fscq file system. In *Proceedings of the 25th Symposium on Operating Systems Principles*, pages 18–37, 2015.

[12] Albert Q. Jiang, Alexandre Sablayrolles, Arthur Mensch, Chris Bamford, Devendra Singh Chaplot, Diego de las Casas, Florian Bressand, Gianna Lengyel, Guillaume Lample, Lucile Saulnier, Lélio Renard Lavaud, Marie-Anne Lachaux, Pierre Stock, Teven Le Scao, Thibaut Lavril, Thomas Wang, Timothée Lacroix, and William El Sayed. Mistral 7b. *arXiv preprint arXiv:2310.06825*, 2023.

[13] Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Amy Yang, Angela Fan, et al. The llama 3 herd of models. *arXiv e-prints*, pages arXiv–2407, 2024.

[14] Jasmin Christian Blanchette and Tobias Nipkow. Nitpick: A counterexample generator for higher-order logic based on a relational model finder. In *International conference on interactive theorem proving*, pages 131–146. Springer, 2010.

[15] Koen Claessen and John Hughes. Quickcheck: a lightweight tool for random testing of haskell programs. In *Proceedings of the fifth ACM SIGPLAN international conference on Functional programming*, pages 268–279, 2000.

[16] Sascha Böhme and Tobias Nipkow. Sledgehammer: judgement day. In *International Joint Conference on Automated Reasoning*, pages 107–121. Springer, 2010.

[17] Zijian Wu, Suozhi Huang, Zhejian Zhou, Huaiyuan Ying, Zheng Yuan, Wenwei Zhang, Dahua Lin, and Kai Chen. Internlm2. 5-stepprover: Advancing automated theorem proving via critic-guided search. In *2nd AI for Math Workshop@ ICML 2025*, 2025.

[18] Ran Xin, Chenguang Xi, Jie Yang, Feng Chen, Hang Wu, Xia Xiao, Yifan Sun, Shen Zheng, and Ming Ding. Bfs-prover: Scalable best-first tree search for llm-based automatic theorem proving. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 32588–32599, 2025.

[19] Google DeepMind. Gemini: A family of highly capable multimodal models. https://deepmind.google/technologies/gemini/, 2024. Accessed model variant: Gemini 3 Pro.

[20] OpenAI. GPT-5.1: A smarter, more conversational ChatGPT, 2025. Accessed model variants: GPT-5.1 Instant and GPT-5.1 Thinking.

[21] Kaiyu Yang, Aidan Swope, Alex Gu, Rahul Chalamala, Peiyang Song, Shixing Yu, Saad Godil, Ryan J Prenger, and Animashree Anandkumar. Leandojo: Theorem proving with retrieval-augmented language models. *Advances in Neural Information Processing Systems*, 36:21573–21612, 2023.

[22] Jasmin Christian Blanchette, Cezary Kaliszyk, Lawrence C Paulson, and Josef Urban. Hammering towards qed. *Journal of Formalized Reasoning*, 9(1):101–148, 2016.

[23] Minchao Wu, Michael Norrish, Christian Walder, and Amir Dezfouli. Tacticzero: Learning to prove theorems from scratch with deep reinforcement learning. *Advances in Neural Information Processing Systems*, 34:9330–9342, 2021.

[24] Suozhi Huang, Peiyang Song, Robert Joseph George, and Anima Anandkumar. Leanprogress: Guiding search for neural theorem proving via proof progress prediction. In *ICLR 2025 Workshop: VerifAI: AI Verification in the Wild*, 2025.

[25] Yuhuai Wu, Albert Jiang, Roger Grosse, and Jimmy Ba. Neural theorem proving on inequality problems. *Artificial Intelligence and Theorem Proving (AITP)*, 2020.

[26] Thibault Gauthier, Cezary Kaliszyk, Josef Urban, Ramana Kumar, and Michael Norrish. Tactictoe: learning to prove with tactics. *Journal of Automated Reasoning*, 65(2):257–286, 2021.

[27] Bartosz Piotrowski, Ramon Fernández Mir, and Edward Ayers. Machine-learned premise selection for lean. In *International Conference on Automated Reasoning with Analytic Tableaux and Related Methods*, pages 175–186. Springer, 2023.

[28] Timothy Bourke and Gerwin Klein. Solve_direct: A tool for isabelle/hol to check whether a newly stated theorem can be solved directly by an existing theorem. https://isabelle.in.tum.de/library/HOL/HOL/ISABELLE_HOME/src/Tools/solve_direct.ML.html, 2025. File: src/Tools/solve_direct.ML in Isabelle/HOL library.

[29] Dominique Unruh. scala-isabelle: A scala library for interacting with isabelle. https://github.com/dominique-unruh/scala-isabelle, 2025. Accessed 2025-11-18.

[30] N. M. Nishant and G. S. Mamatha. Using py4j for java-python communication. *International Journal of Advanced Research in Computer and Communication Engineering (IJARCCE)*, 2023.

[31] Albert Qiaochu Jiang, Wenda Li, Jesse Michael Han, and Yuhuai Wu. Lisa: Language models of isabelle proofs. In *6th Conference on Artificial Intelligence and Theorem Proving*, pages 378–392, 2021.

[32] An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengen Huang, Chenxu Lv, et al. Qwen3 technical report. *arXiv preprint arXiv:2505.09388*, 2025.

[33] Yaowei Zheng, Richong Zhang, Junhao Zhang, Yanhan Ye, Zheyan Luo, Zhangchi Feng, and Yongqiang Ma. Llamafactory: Unified efficient fine-tuning of 100+ language models. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 3: System Demonstrations)*, Bangkok, Thailand, 2024. Association for Computational Linguistics.

[34] Jia Meng and Lawrence C Paulson. Lightweight relevance filtering for machine-generated resolution problems. *Journal of Applied Logic*, 7(1):41–57, 2009.

[35] Daniel Kühlwein, Jasmin Christian Blanchette, Cezary Kaliszyk, and Josef Urban. Mash: machine learning for sledgehammer. In *International Conference on Interactive Theorem Proving*, pages 35–50. Springer, 2013.

[36] Thomas Sewell, Simon Winwood, Peter Gammie, Toby Murray, June Andronick, and Gerwin Klein. seL4 enforces integrity. In *Proceedings of the Second International Conference on Interactive Theorem Proving*, ITP '11, pages 325–340, Berlin, Heidelberg, August 2011. Springer-Verlag.

[37] Toby Murray, Daniel Matichuk, Matthew Brassil, Peter Gammie, Timothy Bourke, Sean Seefried, Corey Lewis, Xin Gao, and Gerwin Klein. seL4: From general purpose to a proof of information flow enforcement. In *IEEE Symposium on Security and Privacy*, pages 415–429, San Francisco, CA, May 2013. IEEE.

[38] Daniel Matichuk, Toby Murray, June Andronick, Ross Jeffery, Gerwin Klein, and Mark Staples. Empirical study towards a leading indicator for cost of formal software verification. In *Proceedings of the 37th International Conference on Software Engineering - Volume 1*, ICSE '15, page 722–732. IEEE Press, 2015.

[39] Vladimir I Levenshtein et al. Binary codes capable of correcting deletions, insertions, and reversals. In *Soviet physics doklady*, volume 10, pages 707–710. Soviet Union, 1966.

[40] Paul Jaccard. Étude comparative de la distribution florale dans une portion des alpes et des jura. *Bull Soc Vaudoise Sci Nat*, 37:547–579, 1901.

[41] Freek Verbeek, Abhijith Bharadwaj, Joshua Bockenek, Ian Roessle, Timmy Weerwag, and Binoy Ravindran. X86 instruction semantics and basic block symbolic execution. *Archive of Formal Proofs*, October 2021. https://isa-afp.org/entries/X86_Semantics.html, Formal proof development.

[42] Lei Yu. A formal model of ieee floating point arithmetic. *Archive of Formal Proofs*, July 2013. https://isa-afp.org/entries/IEEE_Floating_Point.html, Formal proof development.

[43] Filip Marić. Formal verification of modern sat solvers. *Archive of Formal Proofs*, July 2008. https://isa-afp.org/entries/SATSolverVerification.html, Formal proof development.

[44] Stephen Robertson, Hugo Zaragoza, et al. The probabilistic relevance framework: Bm25 and beyond. *Foundations and Trends® in Information Retrieval*, 3(4):333–389, 2009.

[45] Ronghui Gu, Zhong Shao, Hao Chen, Xiongnan Newman Wu, Jieung Kim, Vilhelm Sjöberg, and David Costanzo. {CertiKOS}: An extensible architecture for building certified concurrent {OS} kernels. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 653–669, 2016.

[46] Shih-Wei Li, Xupeng Li, Ronghui Gu, Jason Nieh, and John Zhuang Hui. Formally verified memory protection for a commodity multiprocessor hypervisor. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 3953–3970. USENIX Association, August 2021.

[47] David Greenaway, Japheth Lim, June Andronick, and Gerwin Klein. Don't sweat the small stuff: formal verification of c code without the pain. *ACM SIGPLAN Notices*, 49(6):429–439, 2014.

[48] Xupeng Li, Xuheng Li, Wei Qiang, Ronghui Gu, and Jason Nieh. Spoq: Scaling {Machine-Checkable} systems verification in coq. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*, pages 851–869, 2023.

[49] Ganxiang Yang, Wei Qiang, Yi Rong, Xuheng Li, Fanqi Yu, Jason Nieh, and Ronghui Gu. Highly automated verification of security properties for unmodified system software. In *Proceedings of the 31st ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, pages 912–928, 2026.

[50] Alexandre Riazanov and Andrei Voronkov. The design and implementation of vampire. *AI communications*, 15(2-3):91–110, 2002.

[51] Stephan Schulz, Simon Cruanes, and Petar Vukmirović. Faster, higher, stronger: E 2.3. In *International Conference on Automated Deduction*, pages 495–507. Springer, 2019.

[52] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer, 2008.

[53] Haniel Barbosa, Clark Barrett, Martin Brain, Gereon Kremer, Hanna Lachnitt, Makai Mann, Abdalrhman Mohamed, Mudathir Mohamed, Aina Niemetz, Andres Nötzli, et al. cvc5: A versatile and industrial-strength smt solver. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 415–442. Springer, 2022.

[54] Łukasz Czajka and Cezary Kaliszyk. Hammer for coq: Automation for dependent type theory. *Journal of automated reasoning*, 61(1):423–453, 2018.

[55] Chris Hawblitzel, Jon Howell, Jacob R Lorch, Arjun Narayan, Bryan Parno, Danfeng Zhang, and Brian Zill. Ironclad apps:{End-to-End} security via automated {Full-System} verification. In *11th USENIX symposium on operating systems design and implementation (OSDI 14)*, pages 165–181, 2014.

[56] Jean Yang and Chris Hawblitzel. Safe to the last instruction: automated verification of a type-safe operating system. *Communications of the ACM*, 54(12):123–131, 2011.

[57] K Rustan M Leino. Dafny: An automatic program verifier for functional correctness. In *International conference on logic for programming artificial intelligence and reasoning*, pages 348–370. Springer, 2010.

[58] Mike Barnett, Bor-Yuh Evan Chang, Robert DeLine, Bart Jacobs, and K Rustan M Leino. Boogie: A modular reusable verifier for object-oriented programs. In *International Symposium on Formal Methods for Components and Objects*, pages 364–387. Springer, 2005.

[59] Zhaoyu Li, Jialiang Sun, Logan Murphy, Qidong Su, Zenan Li, Xian Zhang, Kaiyu Yang, and Xujie Si. A survey on deep learning for theorem proving. In *Proceedings of the First Conference on Language Modeling*, 2024.

[60] Daniel Whalen. Holophrasm: a neural automated theorem prover for higher-order logic. *arXiv preprint arXiv:1608.02644*, 2016.

[61] Alex Sanchez-Stern, Yousef Alhessi, Lawrence Saul, and Sorin Lerner. Generating correctness proofs with neural networks. In *Proceedings of the 4th ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*, pages 1–10, 2020.

[62] Emily First and Yuriy Brun. Diversity-driven automated formal verification. In *Proceedings of the 44th International Conference on Software Engineering*, pages 749–761, 2022.

[63] Albert Q Jiang, Sean Welleck, Jin Peng Zhou, Timothee Lacroix, Jiacheng Liu, Wenda Li, Mateja Jamnik, Guillaume Lample, and Yuhuai Wu. Draft, sketch, and prove: Guiding formal theorem provers with informal proofs. In *The Eleventh International Conference on Learning Representations (ICLR)*, 2023.

[64] Chenrui Cao, Liangcheng Song, Zenan Li, Xinyi Le, Xian Zhang, HUI XUE, and Fan Yang. Reviving dsp for advanced theorem proving in the era of reasoning models. In *The Thirty-ninth Annual Conference on Neural Information Processing Systems*, 2025.

[65] Trieu H Trinh, Yuhuai Wu, Quoc V Le, He He, and Thang Luong. Solving olympiad geometry without human demonstrations. *Nature*, 625(7995):476–482, 2024.

[66] Thomas Hubert, Rishi Mehta, Laurent Sartran, Miklós Z Horváth, Goran Žužić, Eric Wieser, Aja Huang, Julian Schrittwieser, Yannick Schroecker, Hussain Masoom, et al. Olympiad-level formal mathematical reasoning with reinforcement learning. *Nature*, pages 1–3, 2025.

[67] Emily First, Markus N Rabe, Talia Ringer, and Yuriy Brun. Baldur: Whole-proof generation and repair with large language models. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 1229–1241, 2023.

[68] Minghai Lu, Benjamin Delaware, and Tianyi Zhang. Proof automation with large language models. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*, pages 1509–1520, 2024.

[69] Tarun Prasad and Nada Amin. Guided proof search using large language models and lemma extraction in coq. In *ICLR 2025 Workshop: VerifAI: AI Verification in the Wild*, 2024.

[70] Zenan Li, Ziran Yang, Haoyu Zhao, Andrew Zhao, Shange Tang, Kaiyu Yang, Aarti Gupta, Zhendong Su, Chi Jin, et al. Goedel-code-prover: Hierarchical proof search for open state-of-the-art code verification. *arXiv preprint arXiv:2603.19329*, 2026.

[71] Stanislas Polu and Ilya Sutskever. Generative language modeling for automated theorem proving. *arXiv preprint arXiv:2009.03393*, 2020.

[72] Guillaume Lample, Timothee Lacroix, Marie-Anne Lachaux, Aurelien Rodriguez, Amaury Hayat, Thibaut Lavril, Gabriel Ebner, and Xavier Martinet. Hypertree proof search for neural theorem proving. *Advances in neural information processing systems*, 35:26337–26349, 2022.

[73] Albert Qiaochu Jiang, Wenda Li, Szymon Tworkowski, Konrad Czechowski, Tomasz Odrzygóźdź, Piotr Miłoś, Yuhuai Wu, and Mateja Jamnik. Thor: Wielding hammers to integrate language models and automated theorem provers. *Advances in Neural Information Processing Systems*, 35:8360–8373, 2022.

[74] Haoxiong Liu, Jiacheng Sun, Zhenguo Li, and Andrew C Yao. Proofaug: Efficient neural theorem proving via fine-grained proof structure analysis. In *International Conference on Machine Learning*, pages 39568–39586. PMLR, 2025.

[75] Jian Fang, Yican Sun, and Yingfei Xiong. Proof strategy extraction from llms for enhancing symbolic provers. *arXiv preprint arXiv:2510.10131*, 2025.

[76] Zenan Li, Zhaoyu Li, Wen Tang, Xian Zhang, Yuan Yao, Xujie Si, Fan Yang, Kaiyu Yang, and Xiaoxing Ma. Proving olympiad inequalities by synergizing llms and symbolic reasoning. In *The Thirteenth International Conference on Learning Representations*, 2025.

ARTIFACT EVALUATED
USENIX
AVAILABLE

Spain: Succinct proofs for numerical computations

Zachary DeStefano, Noah Golub, Zile Huang, Julius Zhang, Sam Frank, Michael Walfish

Courant Institute, NYU

Abstract. In a succinct proof protocol, a *verifier* gets assurance that an untrusted *prover* executed an agreed computation, without requiring the verifier to re-execute the computation itself. In little more than a decade, this area has undergone a remarkable transformation from theory to implemented systems. This activity is extremely exciting. But there is a catch. To apply succinct proofs, one needs to translate one's computation to a set of equations, or *constraints*. The required translation has so far completely blocked systematic support for *numerical* computations, namely those for which the bulk of the computation uses approximations of real numbers. This paper fills that void with the design, implementation, and evaluation of a system called Spain. The starting insight of Spain is that since numerical computations inherently have approximation error, the constraint formalism should likewise allow for *approximate* satisfiability. Based on this insight, Spain introduces a new proof protocol and new ways to translate computations to constraints. Spain's implementation improves over natural baselines by multiple orders of magnitude.

1 Introduction

A *succinct proof protocol* provides execution integrity: one party (the *verifier*) gets assurance that another party (the *prover*) executed an agreed computation, without requiring the verifier to redo the computation or trust the prover. A variant of this setup is zero-knowledge, which additionally hides sensitive inputs from the verifier. Note that execution integrity is orthogonal to program verification. Program verification is about ensuring that a given program meets a specification; here, the question is whether the alleged outputs of a program, verified or otherwise, truly came from running that program.

Succinct proofs have undergone a remarkable transformation. These constructs were known to theorists, based on foundational results in interactive proofs, zero-knowledge proofs, arguments, and probabilistically checkable proofs (PCPs) in the 1980s and 1990s [13, 14, 16, 17, 34, 65, 80, 89, 91, 120]. At the time, they were considered to be wildly impractical. However, over the past 15 years, these constructs have been refined and implemented (see [130, 140] for surveys). They have even been deployed [69, 126], mostly in cryptocurrency [54, 55, 72, 73, 84, 88, 103]. For example, in a traditional blockchain, every node re-executes all transactions to check the validity of state transitions; with succinct proofs, by contrast, a prover generates a proof once and records it on the blockchain, and other nodes need only verify the proof [133].

Despite all of the progress, there is an inescapable awkwardness in applying succinct proofs. One must *arithmetize* one's computation, which means translating it to equations, or *constraints*; roughly speaking, a solution to the constraints corresponds to a valid program trace. Unfortunately, this process does not match how software developers write programs. Control flow, for example, is unnatural to represent as constraints. Making matters worse, the constraints are expressed over a finite field (such as the integers mod a given prime q), yet finite fields have no notion of negative numbers or even order relationships. Consequently, a simple program operation like conditionally branching based on a comparison between two numbers has a very verbose expression in constraints (§2).

Still, researchers and practitioners manage to get around the semantic gap for certain small-scale computations (like validating blockchain transactions). The verifier is truly fast, and the prover's overhead is "only" $10^6\times$ relative to executing the computation natively (§7, §8).

However, the semantic gap becomes a yawning chasm for *numerical* computations: those for which the bulk of the computation uses fixed-point or floating-point approximations of real numbers. The core issue is that numerical computations don't map naturally to equations over finite fields, not even with the kinds of gymnastics used for non-numerical computations. Not to mention, researchers have no hope of competing with the decades of investment in optimizing FPUs and GPUs.

Yet, numerical computations were one of the primary motivations for the invention of computers, and continue to be of intense interest. In the modern era, LLM training and inference, physics simulations, cyberphysical systems, and far more all compute over approximations of real numbers. Succinct proofs applied to numerical computations would thus allow such computations to run on untrusted infrastructure. For example, a user of an LLM could get a proof that the produced tokens truly came from running inference on a given model. Or someone could run a fluid simulation and then prove to others that the simulation ran as specified, without anyone having to redo the computation.

The purpose of this paper is to produce a system that applies succinct proofs to numerical computations, with three goals:

1. *The system should be general-purpose.* This means that the core proving machinery should not be specialized to the computation itself; in particular, if the computation changes, the core protocol should not have to change, nor should the pencil-and-paper mathematics that undergird the correctness of the succinct proof protocol.
2. *The verifier should be less expensive computationally than native execution.* Otherwise, the verifier could simply run the computation itself. Note that in zero-knowledge setups, this requirement does not arise, as the verifier

is not expected to be able to run the computation itself. However, zero-knowledge is a non-goal for us.

3. *The prover's overhead is no more than three orders of magnitude versus natively executing the computation.* While in ordinary computing contexts, overhead of 1000× would be preposterous, in this research area, the state of the art, including in deployed systems, is $10^5\times$ or $10^6\times$.

Prior works encoding numerical computations in proof systems do one of two things (§8). Some works give standalone encodings of specific numerical operations, embedded in general-purpose proof systems; these arguably meet goal 1 but at the cost of goal 3 [9, 50, 51, 59, 82, 115, 124]. Other works build end-to-end special-purpose protocols for specific numerical computations, sacrificing goal 1 – and in most cases goal 3, too [19, 63, 77, 78, 86, 127, 128, 151]. In fact, only one work that we are aware of has achieved goal 3 [105], but at severe compromise of goals 1 and 2.

This paper describes a system, *Spain*, that achieves all three goals in certain regimes. Spain makes several contributions:

A succinct proof framework for approximate rational arithmetic (§3). Spain's idea here is to reflect existing numerical notions of accuracy by using constraints but having the proving machinery establish that each constraint holds *approximately*.

A new proving protocol (§4). To actually prove that the constraints hold approximately, Spain introduces a new proof protocol. It is built on and inspired by prior work, specifically Spartan [116], Zaratan [42], and DARK [40].

Highly compact arithmetizations (§5). Rather than paying a number of constraints proportional to the number of bits in numerical operations, Spain translates individual numerical operations to *single constraints*; for example, division and square root become one constraint each. This in turn enables highly efficient comparisons, range checks, and piecewise functions. Spain also uses division to create efficient translations of transcendental operations, such as e^x. These encodings could be of independent interest.

Rigorous proof (Apps. B, E, F). We prove correctness of Spain's proof protocol and its arithmetizations.

Implementation of Spain (§6, Appx. G). We have implemented Spain. It includes several *front-ends* that translate numerical computations to constraints and a common *back-end* that implements the Spain prover and verifier.

Experimental evaluation (§7). We evaluate Spain and baseline systems on various applications, including linear programming problems; machine learning primitives and an end-to-end application (GPT-2); fluid simulations; and geospatial calculations. We find that Spain has the fastest (general-purpose) numerical prover in the literature. Additionally, Spain's verifier is the first that we are aware of to beat native execution for reasonably-sized numerical computations.

Like all succinct proof systems, Spain must pay high overheads. However, in exhibiting a new kind of arithmetization, Spain has substantially expanded what is possible.

2 Background

2.1 Numerical computations

People want to perform computations that involve real-numbered arithmetic. However, doing so exactly is often intractable. Instead, numerical programs are implemented with *approximate arithmetic*, which comes in two flavors: fixed-point and floating-point. Both have some rounding, or *error*. *Numerical analysis* studies how to ensure that the final result is meaningful.

Fixed-point arithmetic encodes a subset of the rational numbers (denoted $\mathbb{Q}$), specifically those expressible as an integer divided by a fixed denominator (normally a power of two). Real numbers outside of fixed-point are mapped to nearby fixed-point numbers via rounding, prior to entering fixed-point operations. Operations obey absolute error bounds. For any two fixed-point numbers x and y and primitive operation op (for example, $+$, $-$, $\times$, or $\div$), the fixed-point operation op_δ is defined (ignoring overflow) as:

$$(x \ \text{op}_\delta \ y) = (x \ \text{op} \ y) + \widehat{\delta_\pm},$$

where $|\widehat{\delta_\pm}| \leq \delta$ for some fixed δ that depends on the operation and the fixed-point format.

With floating-point representations and arithmetic, operations obey relative error bounds. Then with op as above, x, y as floating-point numbers, and op_δ now denoting the floating-point operation:

$$(x \ \text{op}_\delta \ y) = (x \ \text{op} \ y) \cdot (1 + \widehat{\delta_\pm}).$$

Where $|\widehat{\delta_\pm}| \leq \delta$. Even simple operations introduce errors. For example, when using single-precision IEEE 754 [2] floating point, $(2 \div_\delta 5) \approx 0.400000006$ rather than exactly 0.4.

2.2 Succinct and probabilistic proofs

A *succinct proof* (or *probabilistic proof*) is a cryptographic protocol between a *prover* and a *verifier* (typically thought of as probabilistic algorithms) about a *statement* that the prover wants to persuade the verifier of [64, 130]. We will in this work consider statements of the form: "(in, out) is a valid input-output pair for some procedure F." That is, the succinct proof is aimed at establishing that the alleged *out* is truly the output when F is run on *in*.

Astonishingly, the data flowing from prover to verifier, and the work required by the verifier, is (at least in principle) much smaller than the work to execute F. Yet, a verifier is unlikely to be fooled by a false claim from the prover.

Succinct proof implementations typically have a *front-end* and a *back-end*. The front-end compiles F into a set of *constraints*. The back-end is the proving and verifying algorithms. We elaborate on both below.

Front-end: Arithmetization and R1CS. Various proof back-ends require statements to be compiled, by the front-end, to a *rank-one constraint system* (R1CS), which we often

refer to as *constraints*. An R1CS *structure* is a system of m equations and n variables over a finite field (typically $\mathbb{F}_q$, the integers mod a prime q) with a subset of the variables designated as *in* and a subset designated as *out* [118]. These equations are represented by three matrices A, B, C, each of dimension $m \times n$. An R1CS *instance* for a given structure is (A, B, C, in, out); we say that an instance is *satisfiable* if there exists some *witness* w such that, for $z := (in, out, 1, w)$, we have: $(A \cdot z) \circ (B \cdot z) - (C \cdot z) = \vec{0}$, with $\circ$ denoting entry-wise multiplication. Unpacking the algebra, each constraint $i \in \{1, \ldots, m\}$ restricts any *assignment*, that is any valuation of the variables $z = (z_1, \ldots, z_n)$, as follows: $(A_{i,1}z_1 + \cdots + A_{i,n}z_n) \cdot (B_{i,1}z_1 + \cdots + B_{i,n}z_n) - (C_{i,1}z_1 + \cdots + C_{i,n}z_n) = 0$.

As a simple example, take the computation

function $F(z_1, z_2, z_3)$
  $z_4 \leftarrow (z_1^2 - 4) \cdot (z_2 + 3z_3) - 2$
  return z_4

Here *in* is (z_1, z_2, z_3) and *out* is z_4. The first step in translation to R1CS is to unroll the computation so that each line of code is written as a product of linear combinations of variables, minus another linear combination of variables:

function $F(z_1, z_2, z_3)$
  $z_6 \leftarrow z_1 \cdot z_1 - 4$ // z_6 is a new var, part of the witness
  $z_4 \leftarrow z_6 \cdot (z_2 + 3z_3) - 2$
  return z_4

Next, these lines are translated to constraints, each of which becomes a row in the A, B, C matrices. These constraints are:

$$(1 \cdot z_1) \cdot (1 \cdot z_1) - (1 \cdot z_6 + 4 \cdot 1) = 0$$
$$(1 \cdot z_6) \cdot (1 \cdot z_2 + 3 \cdot z_3) - (1 \cdot z_4 + 2 \cdot 1) = 0.$$

In matrix form, the constraints are:

$$A = \begin{array}{c} \begin{matrix} z_1 & z_2 & z_3 & z_4 & z_5 & z_6 \end{matrix} \\ \left[\begin{array}{ccc|c|c|c} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{array}\right] \end{array}$$

$$B = \left[\begin{array}{ccc|c|c|c} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 3 & 0 & 0 & 0 \end{array}\right]$$

$$C = \begin{array}{c} \left[\begin{array}{ccc|c|c|c} 0 & 0 & 0 & 0 & 4 & 1 \\ 0 & 0 & 0 & 1 & 2 & 0 \end{array}\right] \\ \begin{matrix} \multicolumn{3}{c}{in} & out & 1 & w \end{matrix} \end{array}$$

Now, consider this valuation of the *in* variables: z_1=3, z_2=2, z_3=−1 (these quantities are in $\mathbb{F}_q$, so −1 is shorthand for $q - 1$), and the corresponding *out* valuation: $z_4 = -7$ (shorthand for $q - 7$). In this case, the full assignment is $z = [3, 2, -1, -7, 1, 5]^\intercal$. When subsequently giving example constraints, we drop the explicit "1·", add the "C piece" to both sides, and use semantically appropriate variables to refer to elements of z.

One can translate an entire unrolled computation to a set of constraints, in the sense that some z satisfies the constraints only if z respects the original semantics of the computation. This translation is called *arithmetization* [7, 17, 18, 24, 35, 36, 49, 75, 81, 96, 97, 115, 118, 138, 146, 147]. It is typically accompanied by a *witness generator*, which either executes the computation and thereby obtains the assignment to z [15, 24, 26, 76, 81, 102], or else solves the constraints using annotations provided by the translation process [35, 75, 96, 100, 114, 115].

Modern proof systems also translate computations into *lookup tables* [31, 33, 41, 57, 71, 119, 119], often in combination with constraints. However, lookup tables outperform constraints only when the table is small (for example, 2^8 elements) or has algebraic structure [119]. Numerical operations do not generally fit into either of these categories.

Back-end: Proving R1CS satisfiability. The job of the back-end is, given an R1CS structure and instance (A, B, C, in, out), to prove that a satisfying assignment exists. The back-end provides the following properties, which we state informally.

- *Back-end soundness:* If there does not exist a w such that $z := (in, out, 1, w)$ makes $(A \cdot z) \circ (B \cdot z) - (C \cdot z) = 0$, then the probability (over the verifier's random choices) that the verifier accepts is negligible.
- *Back-end completeness:* If a prover has access to w such that $z := (in, out, 1, w)$ makes $(A \cdot z) \circ (B \cdot z) - (C \cdot z) = 0$, then the prover can make the verifier accept with probability 1, over the prover's and verifier's random choices.

For our work, the relevant strand of back-end protocols is based on the *sum-check* primitive [89]. We defer justifying this choice until we have established the necessary context (§4.1, §4.2.2). The sum-check primitive is an interactive proof [16, 65] by which the prover persuades the verifier that the sum of a given multi-variate polynomial's evaluations over all Boolean combinations that its variables can take is a given value. (Appendix A reviews this primitive.) Succinct proofs that use the sum-check primitive typically do so multiple times within the same protocol; for example, GKR [66, 67] invokes the sum-check primitive once per layer in a layered circuit while Spartan [116] invokes the primitive twice (loosely speaking, one invocation is over the rows of the R1CS instance and one is over the columns). We call proofs built on the sum-check primitive *sum-check protocols* [30, 42, 44–46, 48, 60, 66–68, 116, 117, 119, 123, 129, 135–137, 139, 143, 147, 148].

Modern works in this strand [131] rely on *polynomial commitment*. The idea is that w is encoded as a polynomial by the prover; a small *commitment* (far smaller than w) is sent to the verifier; this commitment binds the prover to that polynomial; and one or more invocations of the sum-check primitive prove that an additional polynomial (formed from the prover's committed polynomial, from *in* and *out*, and from the R1CS structure itself) has a certain property that is equivalent to $z = (in, out, 1, w)$ satisfying the instance.

The costs of the back-end are driven by the *number of constraints*, m and the *number of witness elements*, $|w|$. Specifically, the prover's costs are roughly $O(m + |w|)$. The verifier's costs vary based on the protocol. Our work follows Spartan [92, 116] (without its SPARK module), where the verifier

has an $O(m)$ *fixed cost* that is specific to the structure (the constraints); this cost amortizes over synchronized instances, with each instance adding cost that is $O(\log(|w|))$. There is also an $O(\log(|w|))$ *setup cost*, reusable across all future invocations, including different R1CS structures. See elsewhere [38, 116, 117, 119] for techniques that asymptotically lower the verifier's work at the prover's expense.

To quantify, running the prover on a single machine is generally considered unreasonable when m and $|w|$ are above 2^{29} (unless one possesses terabytes of RAM and hundreds of cores) [108], owing to memory bottlenecks. There are techniques for scaling across a fleet [87, 108, 142, 144], and reducing memory requirements at the cost of increased prover time [20, 93, 99], but neither approach mitigates the sheer amount of work required to prove large computations.

Arithmetization poses a semantic challenge. A salient cost is translating from ordinary computations to R1CS; this appears in the values of m and $|w|$. Indeed, there is a massive semantic gap in this research area: ordinary computations do not map to equations over finite fields in a natural way. To highlight the challenge, consider the division of two positive k-bit integers. Checking this computation, $z \leftarrow x/y$, in R1CS involves two steps. First, the prover supplies the purported quotient z and remainder r and the following constraint is added: $y \cdot z = x - r$, where r is a witness variable. Second, constraints are required to assert that $x \geq (z+1)$ and $y \geq (r+1)$.

But finite fields have no native notion of order, so a typical arithmetization of assert$(x \geq (z+1))$, where x and z are k-bit numbers, introduces witness variables $b_0, \ldots, b_{k-1}$ representing the bits of $x - (z+1)$. Then, for all i, constraints are added to enforce that each variable b_i is a bit: $b_i \cdot (1 - b_i) = 0$. Finally, a constraint is added to enforce the relationship among x, z, and the bits: $\sum_{i=0}^{k-1} b_i 2^i = x - z - 1$. This set of constraints is satisfiable only if $x - z - 1$ is itself a k-bit number. This assert alone requires k additional variables and $k+1$ constraints.

This blowup is not isolated to division. Researchers in succinct proofs often perform contortions to represent computations. While there are, for example, more succinct encodings of certain kinds of conditional operations [115], arbitrary low-level operations of the kind that would ordinarily be accelerated in hardware are not well-handled. In particular, representing a single numerical operation in R1CS typically involves explicitly expressing the digital logic of the operation. This results in a number of auxiliary variables and constraints that is a multiple of the *number of bits of the numbers being represented* in each operation [9, 50, 51, 82, 124] (§7, §8), rather than the *number of machine instructions that would be required in an ordinary computing context*.

3 Problem statement and overview of Spain

Problem statement. A prover and a verifier agree on a single numerical computation, F. The verifier sends input *in*, the prover executes F on that input, and claims that the output is *out*. The prover then wants to persuade the verifier that *out* indeed could have been produced by $F(in)$, given the kinds of approximation that are inherent in fixed- or floating-point numerical computation. The verifier should spend fewer computational resources than if it ran $F(in)$ itself; otherwise, the verifier could disregard *out*, and simply use the result of executing F on *in*. We will sometimes work in an amortized setting where multiple instances of F are outsourced on $in_1, \ldots, in_L$ different inputs, getting different outputs $out_1, \ldots, out_L$.

The core conflict. On the one hand, existing proof machinery ensures that an execution proceeded *exactly* as it was supposed to. On the other hand, in the numerical context "supposed to" means "certain *error*". To resolve this conflict between exactness and error, any protocol must reject truly wrong executions, namely those that include any operations with more than δ error (§2.1). At the same time, the protocol must not be too much of a scold: if all operations do obey the δ error bound, then the given execution should be accepted.

As noted in Section 2.2, some works handle this tension by representing the digital logic of the approximate computation in the arithmetization (the R1CS instance). A related approach is for the prover to embed in the R1CS instance variables that capture each operation's error, thereby making the error incurred by each operation part of the statement to be validated [59]. Both approaches bring expense.

Overview of Spain. In contrast to existing work, Spain's arithmetizations are agnostic to the error in each operation, yet the prover is forced to adhere to the bounds. Spain combines several new techniques, in the front-end and back-end. Figure 1 depicts Spain, contrasting it with the most natural baseline.

Front-end (§5): Spain introduces a new kind of arithmetization; it translates F to a set of constraints over the *rational numbers* $\mathbb{Q}$ (rather than a finite field $\mathbb{F}_q$) in way that satisfying all constraints up to some ϵ corresponds to error up to δ in each operation. Each constraint $i \in \{1, \ldots, m\}$ enforces: $\left|\left(A_{i,1}z_1 + \cdots + A_{i,n}z_n\right) \cdot \left(B_{i,1}z_1 + \cdots + B_{i,n}z_n\right) - \left(C_{i,1}z_1 + \cdots + C_{i,n}z_n\right)\right| \leq \epsilon$. We call such constraints *approximate constraints*; compare to the constraints in §2.2 that enforce equality, which we call *traditional constraints*.

Approximate constraints are dramatically more concise than the alternative; in fact, each low-level operation in F generally corresponds to one or a small number of constraints. The intuition is that constraints in Spain are freed from operating on the bits of operands, being expressed over $\mathbb{Q}$, and are freed from explicitly encoding approximation error, as "$\leq \epsilon$" encodes slackness directly. The trade-off is that a user of Spain must perform certain analyses not needed when working with traditional constraints. This requirement is a manifestation of the conflict stated earlier: the user must show that satisfying the constraints means the prover adhered to "$\leq \delta$" error, and also show that if the prover does execute the operation with bounded error, then it *can* satisfy the approximate constraints.

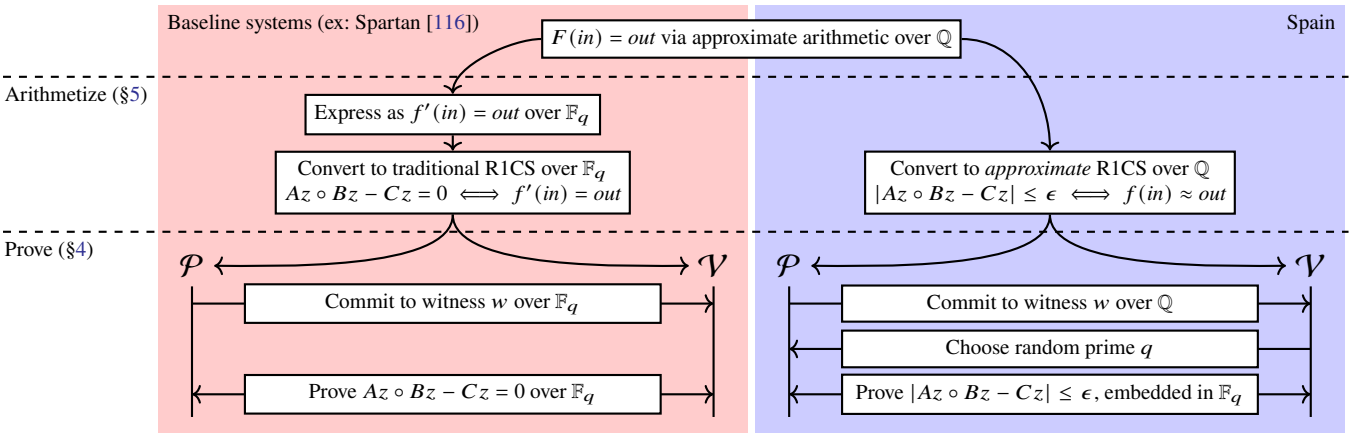


Figure 1: High-level comparison between existing proof systems (left) and Spain (right). Given an initial claim, the protocol is divided into two phases: arithmetization and proving. Similar steps are vertically aligned to facilitate comparison.

Back-end (§4): Spain's back-end enables a prover to establish that a given assignment (to a given set of constraints) meets the ϵ bounds. We call such an assignment ϵ-*accurate*.

Compared to prior work, Spain's back-end has three interlocked aspects. First, Spain maps the constraints over $\mathbb{Q}$ to a finite field, which is the kind of domain typically used in sum-check protocols (§2.2). However, preserving the semantics of the constraints in the finite field requires care; a key mechanism is that the verifier chooses the finite field (via prime q) only after the prover commits to the assignment.

Second, Spain establishes a statement that implies ϵ-accuracy. Specifically, Spain modifies Spartan [116] to prove a statement about the sum of each constraint's squared error, for a given assignment; this sum will be notated as $\|E_{\mathbb{X},z}\|_2^2$. As we argue later, by proving that $\|E_{\mathbb{X},z}\|_2^2$ is small, the protocol guarantees that the assignment obeyed the required ϵ bounds. Third, Spain uses a polynomial commitment protocol (§2.2) that is geared to integers, but makes several technical adjustments, as naive use of this primitive in our context would be prohibitive.

4 Spain's approximation-friendly back-end

This section presents the core statement that Spain's back-end proves (§4.1) and the techniques that it uses to do so (§4.2–§4.3). First, though, we must define what it means for a back-end to be correct in Spain's context. We do so by modifying the existing back-end notions of soundness and completeness (§2.2). The modified properties are as follows, with respect to an R1CS instance $\mathbb{X} = (A, B, C, in, out)$:

- *Back-end ϵ-soundness.* If there does not exist a w such that $z := (in, out, 1, w)$ is ϵ-accurate for $\mathbb{X}$, then the probability κ (over the verifier's random choices) that the verifier accepts is negligible. κ is known as *soundness error*.
- *Back-end ϵ_C-completeness.* A prover with access to w such that $z := (in, out, 1, w)$ is ϵ_C-accurate for $\mathbb{X}$ can make the verifier accept with probability 1, over the prover's and verifier's random choices in the protocol.

These definitions embed two different levels of ϵ-accuracy; at the end of Section 4.1 below, we will see that $\epsilon_C < \epsilon$. To preview the practical consequence, an honest prover will have to execute with greater precision (for completeness) than what the protocol guarantees to the verifier (via soundness).

4.1 What claim about error should be proved?

Spain's back-end must establish the ϵ-accuracy of an assignment z. Before highlighting the challenges, we give a compact notation for ϵ-accuracy. Define the *error vector* $E_{\mathbb{X},z}$ for an R1CS instance $\mathbb{X} = (A, B, C, in, out)$ and an assignment $z = (in, out, 1, w)$ as:

$$E_{\mathbb{X},z} := (A \cdot z) \circ (B \cdot z) - (C \cdot z).$$

This error vector has m components. Its ith component is $(A_i \cdot z) \cdot (B_i \cdot z) - (C_i \cdot z)$, where A_i, B_i, and C_i are the ith rows of A, B, and C. Notice that if the component with maximum absolute value is upper-bounded by ϵ, that is equivalent to saying that the assignment z for the instance $\mathbb{X}$ is ϵ-accurate (§3). Meanwhile, the maximum absolute value over all components of $E_{\mathbb{X},z}$ is, by definition, the ℓ^∞ norm of $E_{\mathbb{X},z}$, denoted $\|E_{\mathbb{X},z}\|_\infty$. Thus, ϵ-accuracy can be written:

$$\|E_{\mathbb{X},z}\|_\infty \leq \epsilon.$$

One challenge is that we don't know how to apply proving machinery to this inequality directly. Sum-check protocols (§2.2), for example, work over polynomial expressions. Although Spartan [116] shows how to cast R1CS satisfiability as suitable polynomial expressions, the ℓ^∞ norm cannot be turned into polynomial expressions, because of the absolute value and maximum operations.

Instead, our insight here is that *the square of the ℓ^2 norm* is compatible with sum-check protocols, and this quantity upper-bounds the square of the ℓ^∞ norm. Specifically, for a vector v,

the square of the ℓ^2 norm, $\|v\|_2^2$, is $\sum_{i=1}^{m} v_i^2$. Also, the definitions of the norms imply that for any vector v, $\|v\|_\infty^2 \leq \|v\|_2^2$. So, if the prover could persuade the verifier that

$$\|E_{\mathbb{X},z}\|_2^2 = J, \tag{1}$$

for some J, and if the verifier then checked that $J \leq \epsilon^2$, that would suffice to establish that $\|E_{\mathbb{X},z}\|_\infty \leq \epsilon$.

However, there is another difficulty. While $\|E_{\mathbb{X},z}\|_2^2 \leq \epsilon^2$ implies $\|E_{\mathbb{X},z}\|_\infty \leq \epsilon$, the converse is not true. Consequently, an assignment z could well satisfy $\|E_{\mathbb{X},z}\|_\infty \leq \epsilon$, but Equation (1) would not hold for any $J \leq \epsilon^2$. In a full protocol, an honest prover would then have no way to convince the verifier.

To address this, the execution by the prover has to be more accurate than ϵ. Specifically, while Spain must meet Back-end ϵ-soundness, it must meet Back-end ϵ_C-completeness, where $\epsilon_C = \epsilon/\sqrt{m}$. (The gap between ϵ and ϵ_C can be narrowed; §D.3.) Under these conditions, there does exist $J \leq \epsilon^2$ with $\|E_{\mathbb{X},z}\|_2^2 = J$. To see this, notice that the definitions of the norms imply that for any vector v, $\|v\|_2^2 \leq m \cdot \|v\|_\infty^2$. Consequently, requiring that the prover produces an ϵ_C-accurate z yields $\|E_{\mathbb{X},z}\|_\infty \leq \epsilon_C = \epsilon/\sqrt{m}$ and thus:

$$\|E_{\mathbb{X},z}\|_2^2 \leq m \cdot \|E_{\mathbb{X},z}\|_\infty^2 \leq m \cdot \epsilon_C^2 \leq \epsilon^2.$$

4.2 Main protocol

Spain's back-end aims to establish Equation (1). Spain adapts three existing tools, specifically *Spartan* [116], which is a sum-check protocol (§2.2) that targets R1CS instances, and forms the core of Spain's back-end; *DARK* [40], which is a polynomial commitment protocol (§2.2) with the ability to bind the prover to a polynomial over $\mathbb{Q}$; and *Zaratan* [42], which is a framework for proving traditional R1CS satisfiability over the integers, as opposed to over a finite field.

One of the configurations explicitly considered by Zaratan is combining Spartan and DARK, and we will do likewise. Specifically, whereas Spartan assumes that an R1CS structure is defined over a finite field (typically $\mathbb{F}_q$; §2.2), Zaratan observes that q can be chosen at run-time by the verifier, after the prover commits to an encoded version of w via DARK.

Spain's full protocol is given in Appendix B. Figure 2 provides an overview. This protocol embeds three major changes compared to prior work, as outlined below.

4.2.1 Proving statements over $\mathbb{Q}$

Constraints in Spain are expressed over the rational numbers, $\mathbb{Q}$. However, the sum-check primitive is typically applied over a finite field (§1, §2.2). Accordingly, Spain maps Equation (1) into a corresponding claim over a finite field, namely $\mathbb{F}_q$. But, this mapping is fraught; to highlight the challenge, we introduce some mathematical language.

Define $\mathbb{Q}^{(q)}$ as $\{(a, b) \in \mathbb{Q} \mid q \nmid b\}$. This set is the rational numbers without multiples of q in the denominator.[1] Spain

[1]Mathematically, this set is known as a localization [141], and could be written $(\mathbb{Z} \setminus q\mathbb{Z})^{-1}\mathbb{Z}$, or $\mathbb{Z}_{(q)}$.

1. The prover runs an agreed-upon computation F, thereby producing w.
2. Using DARK, the prover encodes w as a polynomial, $\widetilde{w}$; *commits* (§2.2) to that polynomial; and sends this commitment to the verifier, effectively binding itself to the value of $E_{\mathbb{X},z}$.
3. The prover computes and sends J to the verifier; J is purportedly $\|E_{\mathbb{X},z}\|_2^2$.
4. The verifier checks that J is less than ϵ^2.
5. The verifier sends a prime q to the prover randomly chosen from a set of large primes.
6. The prover and verifier map all rationals to elements of $\mathbb{F}_q$ and apply the sum-check protocol to the claim $\rho_q(\|E_{\mathbb{X},z}\|_2^2) = \rho_q(J)$, reducing it to a claim about $\widetilde{w}$.
7. The verifier checks consistency between the claim in Step 6 and the commitment to $\widetilde{w}$ in Step 2.

Figure 2: Spain's back-end protocol (simplified).

maps these numbers into a finite field, via:

$$\rho_q : \mathbb{Q}^{(q)} \longrightarrow \mathbb{F}_q$$
$$(a, b) \longmapsto (a \bmod q)(b \bmod q)^{-1},$$

and the Spain prover and verifier apply a sum-check protocol to the claim:

$$\rho_q(\|E_{\mathbb{X},z}\|_2^2) = \rho_q(J). \tag{2}$$

However, for this approach to make sense, there are two requirements. First, arithmetic must stay in $\mathbb{Q}^{(q)}$, otherwise ρ_q is undefined. Second, unequal quantities in $\mathbb{Q}^{(q)}$ should stay unequal after the mapping. That is, if $\|E_{\mathbb{X},z}\|_2^2 \neq J$, then $\rho_q(\|E_{\mathbb{X},z}\|_2^2) = \rho_q(J)$ should not occur, except with negligible probability (over the verifier's choice of q).

Spain achieves both requirements through a combination of convention, DARK, and the verifier's checks. The starting convention is that all rational numbers in the A, B, C matrices that define an R1CS instance $\mathbb{X}$ are presumed to have a denominator D that is a power of two; thus q does not divide D. Furthermore, the assignment z is supposed to follow this convention as well. DARK partially preserves the convention, by ensuring that all committed rationals (Step 2) have the same denominator but not necessarily that it is exactly D. Surprisingly, this partial preservation does not give the prover room to cheat without detection (§B). For simplicity, our description below assumes that DARK preserves the convention fully.

One consequence is that arithmetic begins in $\mathbb{Q}^{(q)}$, and stays there. Thus, ρ_q is defined on all intermediate values and the result. To see this, notice that the convention ensures that the entries of $A \cdot z$, $B \cdot z$, and $C \cdot z$ have denominator D^2; the entries of $(A \cdot z) \circ (B \cdot z)$ have denominator D^4, and the values of $((A_i \cdot z) \cdot (B_i \cdot z) - (C_i \cdot z))^2$, and hence $\|E_{\mathbb{X},z}\|_2^2$, have denominator D^8. The verifier completes the enforcement by requiring J to have denominator D^8.

Another consequence is that this approach avoids collisions. If $\|E_{\mathbb{X},z}\|_2^2$ and J are not equal, then given that they have the same denominator, $\rho_q(\|E_{\mathbb{X},z}\|_2^2) = \rho_q(J)$ only if the numerator N of $\|E_{\mathbb{X},z}\|_2^2 - J$ is a multiple of q. Analysis shows that by limiting the numerators of the entries of A, B, C, and z, N can be kept small enough that the probability of a randomly chosen large prime, like q, dividing N is low (Appx. B.5).

Both Spain and Zaratan [42] map a claim from a larger domain (in their case $\mathbb{Z}$, in Spain's case $\mathbb{Q}$) to some $\mathbb{F}_q$. However, the introduction of a denominator in Spain significantly complicates the analysis of collisions.

The bottom line is that Spain ensures that if the prover sends a $J \neq \|E_{\mathbb{X},z}\|_2^2$ (Step 3, Figure 2), then with overwhelming probability over the choice of q in Step 5, $\rho_q(\|E_{\mathbb{X},z}\|_2^2) \neq \rho_q(J)$. Consequently, if the prover did lie in Step 3, it would be stuck trying to prove a false statement in Step 6.

4.2.2 Proving a new type of claim

To prove that $\rho_q(\|E_{\mathbb{X},z}\|_2^2) = \rho_q(J)$, Spain observes that an adaptation of Spartan can directly prove statements about the squared ℓ^2 norm of the errors in an R1CS instance.

To explain how Spain both borrows and diverges from Spartan, we must present some further mathematical detail. One can view $A, B, C \in \mathbb{F}_q^{m \times n}$, and $z \in \mathbb{F}_q^n$ as *functions*. Specifically, letting $s = \lceil \log m \rceil$ and $k = \lceil \log n \rceil$, we view the matrices as functions $\{0,1\}^s \times \{0,1\}^k \to \mathbb{F}_q$. For example, $A(r, c)$ is the entry of A at the row given by the binary representation of r and the column given by the binary representation of c. We do the same for z, viewing it as a function $\{0,1\}^k \to \mathbb{F}_q$.

An important concept is a polynomial *extension* of a discrete function. The polynomial extension of a function agrees with that function everywhere the function is defined, but the polynomial is defined over a larger domain. A polynomial extension can be thought of as an *encoding* of that function. As a simple example, imagine defining a function f at points 0 and 1, and drawing a line (a first-degree polynomial) through $f(0)$ and $f(1)$. The entire line – its description, or any two points on it – encodes the values of $f(0)$ and $f(1)$. We denote *multilinear extensions* with tildes. For example, for a function $f(\cdot)$ defined over $x \in \{0,1\}^s$, the multilinear extension of f is $\tilde{f} : \mathbb{F}_q^s \to \mathbb{F}_q$, which is a polynomial that equals f on $\{0,1\}^s$, with degree no more than 1 in each variable.

Now, let M stand in for A, B, or C, and let $\widetilde{g_M}(t) := \sum_{c \in \{0,1\}^k} \widetilde{M}(t, c) \cdot \widetilde{z}(c)$, where $\widetilde{M}$ and $\widetilde{z}$ are the multilinear extensions of M and z, respectively (when mapped to $\mathbb{F}_q$). Also, let $Eq(\cdot,\cdot)$ return 1 if its two arguments, viewed as s-bit strings, are the same and 0 otherwise; denote the multilinear extension of Eq as $\widetilde{Eq}$.

In Spartan (and Zaratan), the prover aims to persuade the verifier of the following, where τ is chosen randomly [116, §4]:

$$\sum_{r \in \{0,1\}^s} (\widetilde{g_A}(r) \cdot \widetilde{g_B}(r) - \widetilde{g_C}(r)) \cdot \widetilde{Eq}(\tau, r) = 0. \qquad (3)$$

In Spain, by contrast, the prover aims to persuade the verifier (in Step 6) that the following holds:

$$\sum_{r \in \{0,1\}^s} (\widetilde{g_A}(r) \cdot \widetilde{g_B}(r) - \widetilde{g_C}(r))^2 = \rho_q(J). \qquad (4)$$

Notice from the definition of extension that the left-hand side above sums the squared error for each constraint, given assignment z, when the constraints and z are mapped to $\mathbb{F}_q$. That sum is $\rho_q(\|E_{\mathbb{X},z}\|_2^2)$. Thus, the protocol is indeed establishing Equation (2). After this step, Spain follows Spartan.

4.2.3 Accelerating and adapting DARK

Spain makes two sets of modifications to DARK. First, whereas DARK normally works over a group of order unknown to both prover and verifier, Spain exploits interactivity to have the verifier generate a group for which it alone knows the order. Appendix C details this technique.

Second, DARK as originally proposed had an error: it provided only a weak binding property, namely it ensured only that the committed-to polynomial had *rational* coefficients [29]. This was problematic for Zaratan and other uses of DARK (for example, in combination with Spartan), as those uses require a polynomial with *integer* coefficients. Thus, Zaratan (and other protocols) layer techniques [29, 42] atop DARK, which have extra costs. However, in our context, the original (weaker) form of DARK suffices. Appendix B gives details.

4.3 Support for batching and "just in time" R1CS

Spain supports two variants of R1CS. The first is a *SIMD-R1CS instance* [134], which is a combination of L R1CS instances with the same structure (that is, multiple executions of the same program on different inputs). The second is an *I-R1CS instance* [97], where the prover and verifier interact to construct the full instance. Given a partial R1CS instance (one with some variables missing), the prover and verifier take turns filling it in. In round i, the prover commits to w_i, and the verifier sends additional in_i. The final R1CS instance has assignment $z := (in_1, \ldots, in_R, out, 1, w_1, \ldots, w_R)$.

Spain extends I-R1CS in a natural way: in Spain, the verifier may also supply *constraints*. Thus, the prover and verifier start with a partial structure, and as the prover commits to w_i in round i, the verifier submits in_i or new constraints. Further details of Spain's support for R1CS variants are in Appendix D.

5 Encoding numerical operations in R1CS

This section describes how, by taking advantage of the backend claim of ϵ-accuracy (§4), Spain produces dramatically more concise arithmetizations versus the traditional approach.

Using approximate constraints (§3) requires a new kind of correspondence between a numerical function F and an

R1CS structure that allegedly arithmetizes F. Roughly speaking, (a) any assignment that is ϵ-accurate (§3) for the constraints should correspond to an execution of F with operations bounded by δ (§2.1), and (b) for every execution of F with operations bounded by δ_{wg} (where δ_{wg} is a constant smaller than δ) there should be an ϵ_C-accurate assignment. The need for separate δ and δ_{wg} reflects the same asymmetry that leads to $\epsilon_C < \epsilon$ (§4). Appendix E formalizes (a) and (b) as *translation fidelity*, and proves that the constraints described in this section meet these properties.

Note that translation fidelity concerns the error in *each* operation, not the accumulated error in the output of a computation. This is analogous to how the IEEE floating-point specification [2] bounds the error in each operation and relies on numerical analysis to derive bounds on the accumulated error.

Below we present arithmetizations using approximate constraints, notating them with $\approx_\epsilon$. For example, $z_1 \cdot z_2 \approx_\epsilon z_3$ enforces $|z_1 \cdot z_2 - z_3| \leq \epsilon$.

Division and square root. Recall that checking $z \leftarrow x/y$ traditionally requires dozens of constraints (§2.2). Spain uses *one* constraint: $y \cdot z \approx_\epsilon x$.

In traditional constraints, the square root operation ($y \leftarrow \sqrt{x}$) is encoded as $y \cdot y = x - r$ plus dozens of constraints to bound r in terms of x and y. Spain again requires only one constraint: $y \cdot y \approx_\epsilon x$. This takes advantage of the fact that all real square roots have arbitrarily close rational approximations. If one wants a specifically positive or negative root, one applies the square root operation again! For example, suppose we want y to be the negative square root of x. Then we use an additional constraint: $t \cdot t \approx_\epsilon -y$, exploiting the fact that $-y$ must be non-negative to have a real square root.

Note that this encoding of the square root operation can be satisfied when x is a small negative number, in contrast to usual numerical computations, where Not a Number (NaN) would arise. If this is problematic, the constraints and/or numerical analysis should ensure that the function's input is strictly non-negative.

The power of approximate square roots. In traditional arithmetization, bottlenecks include range checks, branching, max, and min. As we show next, Spain arithmetizes approximate versions of these operations orders of magnitude more efficiently, using square roots.

assert ($\mathbf{x} \geq \mathbf{y}$). Recall from Section 2.2 that this operation required k additional variables and $k+1$ constraints. Spain requires only 1 constraint and 1 additional variable, regardless of bit width: $t \cdot t \approx_\epsilon x - y$, for some witness variable t. This is not a strict translation of assert but instead an approximate version that allows x to be ϵ smaller than y but no smaller.

$\mathbf{b} \leftarrow (\mathbf{x} \geq \mathbf{y})$. Spain requires only 3 constraints and 2 auxiliary variables: $b \cdot (1-b) \approx_\epsilon 0$, $(2b-1) \cdot (x-y) \approx_\epsilon t$, and $s \cdot s \approx_\epsilon t$. The first constraint ensures that b is Boolean. The second ensures that t is non-negative only when b correctly reflects the comparison. The third ensures that t is indeed non-negative. As with square root and assert, this comparison is an approximate version; Appendix E.2 delves into the semantics.

max. Consider $z \leftarrow \max(v)$ for an array $v = [v_1, \ldots, v_L]$. Spain's approach here is inspired by Distiller [76], which encodes a *checker* for min, rather than embedding logic to identify the minimum. However, Distiller requires $L \cdot (k+3)+1$ constraints and $L \cdot (k+2)$ witness variables, where k is the bit-widths of the elements in the array. Spain's are far more concise because it pays so little for comparison operations:

- for all $i \in \{1, \ldots, L\}$: $t_i \cdot t_i \approx_\epsilon z - v_i$
- for all $i \in \{1, \ldots, L\}$: $b_i \cdot (1 - b_i) \approx_\epsilon 0$
- for all $i \in \{1, \ldots, L\}$: $b_i \cdot (z - v_i) \approx_\epsilon 0$
- $\sum_{i=1}^{L} b_i \approx_\epsilon 1$

The first set ensures that z is greater than or equal to all elements in the array. The next two sets ensure that b_i is Boolean and that b_i can be 1 only if $v_i = z$ (to support multiple equal maxima, b_i is allowed to be 0 even when $v_i = z$). The last ensures that exactly one of the b_i is 1. This requires $3 \cdot L + 1$ constraints and $2 \cdot L$ auxiliary variables.

Piecewise functions, sorting, and beyond. The encoding of comparisons naturally leads to efficient translations of piecewise functions; ReLU will demonstrate this below. Similarly, the previous best solutions for sorting or for string manipulation paid relative to the number of bits in their inputs due to comparisons, while Spain's approach more closely matches the number of instructions in a typical CPU execution.

ReLU is a primitive in machine learning contexts. The operation is $z \leftarrow \max(0, x)$. This function can be represented in constraints as $s \cdot s \approx_\epsilon z - x$, $t \cdot t \approx_\epsilon z$, and $(z - x) \cdot z \approx_\epsilon 0$. The first two ensure that $z \geq \max(0, x)$. The last one ensures that $z = 0$ or $z = x$.

Transcendental functions. Arithmetizing transcendental functions, such as e^x or $\tanh(x)$, over some interval traditionally uses *polynomial approximation* [50, 51, 78, 82, 105, 124, 128, 151]. Specifically, for a function $f(x)$ to be approximated, one identifies a polynomial $P(x)$ such that $|P(x) - f(x)| \leq \delta$ for x in the given interval. The motivation is that add and multiply, which are the operations that polynomials require, are the easiest to represent in traditional constraints.

However, there is an additional issue (besides the overhead from traditional enforcement of numerical operations in constraints): polynomials require many terms to converge. Drawing on its support for division, Spain uses *rational approximations*: a ratio of polynomials $P(x)/Q(x)$ such that $|P(x)/Q(x) - f(x)| \leq \delta$ for x in the interval. In Spain, rational approximations cost nearly nothing extra over polynomial approximations, but rational approximations converge faster. To illustrate, compare the degree-4 Taylor series for e^x:

$$g(x) = 1 + x + \frac{x^2}{2} + \frac{x^3}{6} + \frac{x^4}{24}$$

to the degree-$[4/4]$ Padé approximant [98]:

$$h(x) = \frac{1 + \frac{x}{2} + \frac{x^2}{10} + \frac{x^3}{120} + \frac{x^4}{1680}}{1 - \frac{x}{2} + \frac{x^2}{10} - \frac{x^3}{120} + \frac{x^4}{1680}}.$$

Note that the number of constraints for the two approaches is the same (for translating $h(x)$, Spain memoizes the powers of x). One can compare accuracy (using a computer algebra system or a large piece of paper), and consider the size of the intervals for which $|g(x) - e^x| \leq \delta$ and $|h(x) - e^x| \leq \delta$ respectively. In this example, take $\delta = 0.01$. The Taylor Series (g) has error at most 0.01 over the interval $[-1.07, 1.00]$ and larger errors outside this interval. The Padé approximant (h) has error at most 0.01 over $[-8.11, 2.84]$. In fact, the degree $[2/2]$ Padé approximant has error at most 0.01 over $[-2.18, 1.16]$: better than the degree-4 Taylor series.

Better approximations (more accuracy, wider interval) for the same number of terms also translates to fewer variables and hence fewer constraints to achieve the same accuracy. Going beyond exponentials, the technique of rational approximation works for a large class of functions.

Concurrent work [124] makes a similar observation; they find that for highly precise approximations, arithmetizations (of high-degree polynomials) can be even more expensive than traditional arithmetizations of division. These authors then incorporate Padé approximations to existing proof pipelines, obtaining a small speedup. Their paper contains many intriguing examples of approximation using this technique. All of these examples can be used in Spain directly, only with lower cost, because division is so inexpensive in Spain.

6 Implementation of Spain

6.1 Back-end implementation

Spain's back-end implements the protocol described in Section 4.2. It also supports SIMD-R1CS and I-R1CS (§4.3).

Spain implements fixed-size large integer arithmetic with inline limb storage; this avoids the performance cost of existing large integer libraries' [3–5] heap allocation of limb arrays. This functionality is used to compute and check J (Steps 3 and 4 in Fig. 2). Spain also implements 64- and 128-bit finite field arithmetic, with support for fast type conversion from large integers (Step 6); Spain does not use existing finite field libraries [11] because they optimize at compile time for a specific field, whereas in Spain, the finite field is determined only at run time. Finally, Spain implements fixed-size RSA group arithmetic, to do fast modular arithmetic for moduli not known at compile time (Steps 2 and 7).

Spain's back-end is implemented in 12,135 lines of Rust, including the new libraries.

6.2 Front-end implementation

We implemented three front-ends for Spain.

Gadget front-end. This front-end provides a library of gadgets [113, 150] (pairs of functions where the first generates constraints and the second generates witness values satisfying them), including the primitives in Section 5. The framework is modular and extensible.

ONNX front-end. ONNX [95, 132] is a specification for machine learning models. In ordinary use, one describes a model in ONNX's format (a collection of nodes), and then a highly optimized run-time executes the model on the available hardware. Spain's ONNX front-end translates from the ONNX format to approximate constraints. For the accompanying witness generator (§2.2), we had hoped to reuse ONNX-compatible run-times and tools; however, existing run-times neither produce satisfying assignments to auxiliary variables of the kind that arise in arithmetization (§5) nor run with the high-precision floating-point types that Spain uses (§6.3).

Spain solves this challenge with two sub-components. The first is a *model translator* that converts an ONNX model to a verbose version of the model, whose nodes produce output that includes the auxiliary information. In the process, the model translator rewrites transcendental functions to use rational approximations (§5), using the textbook implementation of the Remez algorithm [104]. The translator also modifies the operations in the model to specify higher-precision data types (for example, double- or quadruple-precision floating point instead of single-precision floating point). The second sub-component is a from-scratch ONNX executor that supports these high-precision data types. This executor implements each ONNX operator that appears in Spain's benchmarks (§7).

Linear programming front-end. This front-end is specialized to linear programming problems for comparison with prior work (§7), specifically Otti [9].

Spain's front-ends are implemented in, respectively, 2898 lines of Rust; 4747 lines of Python and 2025 lines of Rust; and 1524 lines of Rust.

6.3 Numerical considerations

A user of Spain begins with a particular computation F and desired error δ. Note that the choice of δ is exogenous to Spain, and should be driven by numerical considerations (§2.1). The user must then determine: (a) ϵ, to inform the verifier's check of J (§4), (b) δ_{wg}, the precision at which the prover must run F (§5, Appx. E), and (c) D, the denominator used in the arithmetization of F (§4.2.1, Appx. B). The general recipe for doing so is as follows. First, translate F to R1CS. Next, perform numerical analysis on each operation in the translation and compose the analyses to obtain an upper bound on ϵ in terms of δ (Appx. E.2). This ϵ, along with the number of constraints, m, upper-bounds ϵ_C (§4). Finally, returning to the numerical analysis, ϵ_C enforces an upper bound on δ_{wg} and on $1/D$, and thus a lower bound on D (Appx. E.2).

One could perform this last step analytically; essentially one would determine the maximum magnitude of values in the computation over all possible inputs, and then choose small-

enough δ_{wg} and $1/D$ to ensure each constraint has error upper-bounded by ϵ_C. For convenience, we instead guess (by running the witness generator at an estimated high precision) and check (by observing that it produces a witness for which $\|E_{\mathbb{X},z}\|_2^2 \leq \epsilon$). This approach is not dangerous: choosing δ_{wg} and $1/D$ too large – meaning underestimating the required precision – cannot affect soundness (which holds regardless of the prover's behavior). Furthermore, if there were such an underestimate, the prover could recalibrate and run witness generation at higher precision for all subsequent proof generations; witness generation is not the primary bottleneck for the prover (§7.2). For the same reason, accidentally overestimating precision does not unduly increase costs.

In Section 7, we provide values of δ, ϵ, ϵ_C, and D for the benchmarks in our experiments. Here, we work through their derivation for one example, a fluid simulation with $\delta = 2^{-32}$.

By analyzing the translation to R1CS (Appx. E), we get that $\epsilon = \delta/4$ suffices: an ϵ-accurate assignment to the constraints ensures absolute error of $\delta = 2^{-32}$. Supposing that $m \leq 2^{26}$, the prover needs a witness with error $\epsilon_C \leq \epsilon/\sqrt{2^{26}} = 2^{-47}$ (§4.1). For witness generation, this example uses double-precision floating-point arithmetic, which has *relative* error about 2^{-53}. In this application, the magnitude of intermediate values is sufficiently small that this relative error keeps the absolute error of each operation (δ_{wg}) below 2^{-47}. Note that the next choice up, quadruple-precision floating-point arithmetic, has relative error 2^{-112}, and would suffice if the intermediate values were larger.

Furthermore, most constraints are satisfied with error far smaller than ϵ_C, so the "closeness" of δ_{wg} and the floating-point relative error is not a concern in practice, and one can even run witness generation in a way where *some* constraints violate ϵ_C provided their totality satisfies $\|E_{\mathbb{X},z}\|_2^2 \leq \epsilon$. To counterbalance δ_{wg} being close to ϵ_C, this example chooses $1/D$ to be significantly smaller than ϵ_C: 2^{-70}.

Spain's back-end scales these floating-point values and converts them to large fixed-width integers to commit via DARK and to compute J exactly (Steps 2 and 3, Figure 2). For these steps, Spain combines 256-, 512-, and 786-bit integer arithmetic. Spain then maps these integers to field elements to execute the sum-check protocol steps (Step 6, Figure 2).

7 Experimental evaluation

We aim to answer the following questions experimentally:

1. How does Spain perform compared to the best proof systems for numerical computations?
2. What are the contributions to Spain's costs?
3. In what regimes does Spain meet the performance goals stated in Section 1?

Baselines. We compare Spain to five baselines, listed below. Two are end-to-end general-purpose systems, two are synthetic general-purpose systems with back-ends very similar to Spain's (for isolating the effect of Spain's front-end), and one is a special-purpose system.

Otti [9]: This is a general-purpose proof system in which fixed-point numerical computations are translated to finite field operations (§2.2). Otti aims at linear programming and semi-definite programming. Otti's back-end is a non-interactive and zero-knowledge variant of Spartan [116] so Otti offers qualitative properties that Spain does not (§7.4). Our experiments run Otti's released code [10].

ZKLP [51]: This general-purpose proof system encodes IEEE 754 floating-point logic in I-R1CS (§4.3, §D.2). ZKLP's back-end is the gnark [32] implementation of Groth16 [70], which (like Otti's back-end) is non-interactive and zero-knowledge. Our experiments run ZKLP's released code [52].

Otti-FE: This baseline uses Otti's front-end, meaning the same constraints as in Otti, and couples it with Spain's back-end implementation (including DARK), adjusted to prove a statement in the form of Equation (3) (§4.2.2). This setup results in a small fraction of Otti's constraints not being satisfied, because they rely on inverses over Otti's original finite field, rather than the one that Spain uses for sum-check, which is smaller. To be pessimistic to Spain, we filter out these constraints in our measurements.

ZKLP-FE: This baseline uses the same back-end as Otti-FE. For the front-end, we adopt a synthetic approach because ZKLP's front-end is not compatible with Spain's back-end, and making it so would have been a significant engineering effort. To apply this baseline to a benchmark, we count the operation types in the benchmark; for each operation type, we borrow constraint counts from ZKLP, and use the total number of constraints to construct an R1CS structure. Although this structure is semantically meaningless, its *size* (which is what drives performance) reflects the expected number of constraints that would be produced by using ZKLP's arithmetization of single-precision floating-point computations.

zkGPT [105]: This baseline is a special-purpose proof system for GPT-2 inference (described below). We choose it because it is, to our knowledge, the fastest application-specific prover for any numerical computation in the literature; however, the proof system itself mirrors the structure of GPT-2 inference, so it cannot be adapted to other computations.

Benchmarks. Our benchmarks are in four families:

Linear Programming. This family includes several linear programming instances from the Netlib library [61] that are used by Otti [9] (specifically, adlittle, afiro, sc105, scagr7, and scsd8).

Machine Learning. This family includes common machine learning primitives (Softmax, LayerNorm, and GELU) as well as the inference phase of GPT-2 [106].

Softmax maps $x = [x_1, \ldots, x_L]$ to $y = [y_1, \ldots, y_L]$ as follows: $y_i \leftarrow e^{x_i}/(\sum_{j=1}^{L} e^{x_j})$, for $i \in \{1, \ldots, L\}$. Spain arithmetizes this computation with Remez approximations of $e^{x_1}, \ldots, e^{x_L}$ (§5). This microbenchmark runs on each row of a 32×32 tensor.

F	δ	ϵ	ϵ_C	D
Linear Programming	2^{-32}	2^{-32}	$2^{-38.26}$	2^{50}
Machine Learning	2^{-20}	2^{-40}	$2^{-46.75}$	2^{75}
Fluid Simulation	2^{-32}	2^{-34}	$2^{-44.27}$	2^{70}
Geolocation	2^{-13}	2^{-26}	$2^{-35.10}$	2^{70}

Figure 3: Numerical parameters for benchmark families. ϵ_C varies within a family as it depends on the number of constraints m (§4.1) and batch size L (§4.3, §D.1) for the particular instance(s). We report the minimum ϵ_C across the family. δ_{wg} is omitted because all witness generation is performed using double- or quadruple-precision floating point, which has relative rather than absolute error guarantees (§6.3). Specifically, the Linear Programming, Fluid Simulation, and Geolocation families use double-precision, and the Machine Learning family uses quadruple-precision.

LayerNorm involves computing the mean and variance of a vector and then normalizing the vector. Specifically, $x = [x_1, \ldots, x_L]$ is transformed to $y = [y_1, \ldots, y_L]$ via $y_i \leftarrow \frac{x_i - \mu}{\sqrt{v+\delta}}$, where μ and v are the mean and variance of the inputs x, and δ provides numerical stability. Spain arithmetizes this benchmark with square root, division, and multiplication operations. We run this microbenchmark on a 32×768 tensor.

GELU is a popular activation function used in GPTs, defined as $x \cdot \Phi(x)$, where $\Phi(x)$ is the Gaussian CDF. Spain arithmetizes this with a piecewise linear approximation (§5). We run this microbenchmark on a 32×3072 tensor.

GPT-2 [106] is an open source LLM that uses several machine learning primitives, including Softmax, LayerNorm, GELU, and matrix multiplication (MatMul). For MatMul, Spain includes an optimization not described earlier: Spain translates to a *checker* that uses the Freivalds algorithm [56]. This requires an approximate version of Freivalds [74] together with Spain's extension to I-R1CS (§4.3, Appx. D.2). This benchmark runs the inference phase of GPT-2, and is parameterized by a length of input tokens (*seq*), and a number of iterations, *passes*.

Fluid Simulation. This family is a Stable-Fluids-style [125] incompressible fluid simulation that approximately solves the 2D Navier-Stokes equations over square grids. Spain arithmetizes this benchmark with multiplication, max, and min operations. We benchmark 8×8 and 16×16 grids, each simulated for 10 timesteps, denoted *small* and *large* respectively. Even at such small sizes, these benchmarks are computationally intensive; simulations of this form often run on GPUs in practice.

Geolocation. This family includes the geospatial computation in ZKLP [51], which checks consistency between spherical coordinates (latitude, longitude) and coordinates in a hierarchical Discrete Global Grid System [110] called Uber H3 [37]. This benchmark required simulating modular arithmetic, by arithmetizing the $\lfloor \cdot \rfloor$ operation.

Figure 3 gives the numerical parameters for each family.

Metrics, measurement, and method. We report the proving time, verification time, constraint counts (m, from §2.2), and proof size of Spain and baseline systems on the benchmarks above. We also report the cost of native execution. In addition, we measure the memory usage of Spain's prover, and fixed and setup costs for the Spain verifier.

We run all experiments in single-threaded configurations, on CPUs. We measure time based on the wall clock, with a harness that runs and times all phases of Spain. The exception is witness generation. That is timed differently depending on the benchmark. For the Linear Programming benchmarks, we time a custom Rust program that solves the linear program and uses the solution to generate the witness values. For ONNX benchmarks, we time the execution of the transformed ONNX model on Spain's high-precision ONNX executor (§6.2). For Fluid Simulation, similar to the Linear Programming benchmarks, we time a custom Rust program that generates witness values.

We report the average of at least 5 runs for each experiment. The standard deviations are within 11% of the means.

Testbed. For all experiments, we run the verifier on a 2.1 GHz 64-Core AMD Opteron 6272 with 256 GB RAM, running Red Hat Enterprise Linux 9.8. To reflect the motivation for succinct proofs, we run the prover on a more powerful machine: a 64-Core Intel Xeon Platnium 8592+ with 3 TB RAM, running Red Hat Enterprise Linux 9.6. When running Otti and ZKLP on our hardware, we use the artifacts provided by the authors.

7.1 Spain versus baselines

To compare Spain to the baselines, we run the aforementioned benchmarks. For ZKLP-FE run on GPT-2, the constraint counts are too large to run the back-end. Instead, we estimate prover and verifier time, as well as proof size, by executing ZKLP-FE on synthetic instances of various sizes and extrapolating times linearly ($R^2 > 0.99$ for both).

Figure 4 depicts the results. Spain shows dramatic improvement in number of constraints (m). This owes to the techniques in Section 5; for example, with the Linear Programming benchmarks, the baseline needs verbose constraints to represent comparisons ($\geq$, $\leq$, and so on) whereas Spain does not.

The lowered m for Spain yields prover speedups of 8–2700×, bringing the prover's overhead down in some cases to our goal of 3 orders of magnitude (§1). In fact, notice that Spain's back-end is *worse* than the baselines on a per-constraint basis; for example, Otti outperforms Otti-FE (which has a Spain-like backend) when running with substantially the same constraints. Yet, the reduction in m is more than enough to compensate. Indeed, the only baseline that Spain's prover does not exceed is zkGPT. But zkGPT does not apply beyond that one benchmark, and it has qualitative disadvantages, described later (§7.3).

Similarly, Spain's verifier improves on all baselines except zkGPT (see above) and ZKLP. ZKLP's back-end is specifically aimed at verifier performance, but it has high setup cost (§7.3, §7.4). Spain's improvement over all other baselines

			Constraints (m)			Prover time				Verifier time			Proof size		
Benchmark	**Baseline**	**Native time**	**Baseline**	**Spain**	**Ratio**	**Baseline**	**Spain**	**Speedup**	**Spain/Native**	**Baseline**	**Spain**	**Speedup**	**Baseline**	**Spain**	**Ratio**
adlittle	Otti	2.5 ms	181,000	292	620×	930 ms	62 ms	15×	69×	480 ms	17 ms	29×	29 KB	9.3 KB	3.1×
adlittle	Otti-FE	2.5 ms	142,000	292	490×	7.8 s	62 ms	130×	69×	190 ms	17 ms	11×	16 KB	9.3 KB	1.8×
afiro	Otti	420 μs	36,800	111	330×	270 ms	33 ms	8.2×	180×	160 ms	14 ms	11×	20 KB	7.4 KB	2.7×
afiro	Otti-FE	420 μs	29,100	111	260×	2.1 s	33 ms	62×	180×	59 ms	14 ms	4.3×	15 KB	7.4 KB	2.0×
sc105	Otti	3.1 ms	113,000	372	300×	490 ms	62 ms	8.0×	64×	320 ms	17 ms	19×	21 KB	9.3 KB	2.2×
sc105	Otti-FE	3.1 ms	80,000	372	220×	3.9 s	62 ms	63×	64×	110 ms	17 ms	6.6×	16 KB	9.3 KB	1.7×
scagr7	Otti	6.2 ms	229,000	455	500×	900 ms	62 ms	14×	34×	540 ms	17 ms	31×	29 KB	9.3 KB	3.1×
scagr7	Otti-FE	6.2 ms	159,000	455	350×	7.6 s	62 ms	120×	34×	200 ms	17 ms	12×	16 KB	9.3 KB	1.8×
scsd8	Otti	300 ms	3.42e6	5,900	580×	9.9 s	320 ms	31×	4.0×	8.9 s	30 ms	290×	81 KB	13 KB	6.2×
scsd8	Otti-FE	300 ms	3.42e6	5,900	580×	120 s	320 ms	380×	4.0×	4.0 s	30 ms	130×	20 KB	13 KB	1.5×
Softmax	ZKLP-FE	41 μs	1.31e6	7,230	180×	15 s	200 ms	74×	48,000×	1.5 s	49 ms	31×	22 KB	15 KB	1.4×
LayerNorm	ZKLP-FE	230 μs	1.01e7	98,400	100×	120 s	1.3 s	92×	52,000×	12 s	160 ms	73×	24 KB	19 KB	1.3×
GELU	ZKLP-FE	1.7 ms	4.71e7	1.47e6	32×	470 s	19 s	25×	120,000×	49 s	1.6 s	31×	26 KB	22 KB	1.2×
GPT-2 (*seq*=2, *passes*=1)	ZKLP-FE	79 ms	1.85e10	1.63e6	11,000×	*37 h*	50 s	2,700×	1,500×	*4.7 h*	22 s	770×	*860 KB*	1.5 MB	0.56×
GPT-2 (*seq*=32, *passes*=1)	ZKLP-FE	440 ms	2.26e10	4.19e7	540×	*45 h*	750 s	210×	11,000×	*5.7 h*	78 s	260×	*860 KB*	1.6 MB	0.52×
GPT-2 (*seq*=32, *passes*=1)	zkGPT	440 ms	—	4.19e7	—	64 s	750 s	0.085×	11,000×	5.8 s	78 s	0.075×	88 KB	1.6 MB	0.053×
GPT-2 (*seq*=32, *passes*=2)	zkGPT	870 ms	—	4.19e7	—	130 s	1,600 s	0.081×	12,000×	12 s	78 s	0.15×	180 KB	3.2 MB	0.053×
GPT-2 (*seq*=32, *passes*=4)	zkGPT	1.7 s	—	4.19e7	—	260 s	3,200 s	0.081×	12,000×	23 s	78 s	0.30×	350 KB	6.4 MB	0.054×
GPT-2 (*seq*=32, *passes*=8)	zkGPT	3.5 s	—	4.19e7	—	510 s	1.8 h	0.081×	12,000×	46 s	78 s	0.60×	710 KB	13 MB	0.054×
GPT-2 (*seq*=32, *passes*=16)	zkGPT	7.0 s	—	4.19e7	—	1,000 s	3.3 h	0.086×	11,000×	93 s	79 s	1.2×	1.4 MB	26 MB	0.054×
Fluid Sim. (small)	ZKLP-FE	3.3 ms	2.32e7	291,000	80×	240 s	4.4 s	54×	5,500×	25 s	440 ms	58×	25 KB	21 KB	1.2×
Fluid Sim. (large)	ZKLP-FE	13 ms	1.26e8	1.53e6	83×	920 s	17 s	53×	4,800×	140 s	1.9 s	71×	27 KB	23 KB	1.2×
Geoloc. (1 Location)	ZKLP	790 ns	25,500	596	43×	660 ms	76 ms	8.7×	96,000×	8.6 ms	36 ms	0.24×	0.38 KB	13 KB	0.030×
Geoloc. (512 Locations)	ZKLP	410 μs	1.02e7	596	17,000×	220 s	4.8 s	46×	12,000×	9.7 ms	50 ms	0.19×	0.38 KB	20 KB	0.019×

Figure 4: Constraint counts, native execution time, prover time, verifier time, proof sizes, and overhead of proving (ratio of prover:native) for baselines and benchmarks. "Native time" refers to the verifier's hardware while the Spain/native ratio is measured on the prover's hardware. Times, proof sizes, and ratios are displayed with two significant figures. Constraints are displayed with three significant figures. The italicized numbers for ZKLP-FE mean that the times and proof sizes are extrapolated (see text); the back-end could not run at that scale. zkGPT does not support a batched configuration, so we scale zkGPT's measurements by *passes* where appropriate. Spain improves over baselines in constraint counts (from 32× to 4 orders of magnitude). With two exceptions, Spain improves over baselines in prover and verifier time, by at least an order of magnitude. The first exception is zkGPT, which outperforms Spain on all metrics, except for verifier time when *passes* = 16, where Spain's verifier is slightly better. The second exception is ZKLP, whose verifier outperforms Spain's verifier by 4–5×.

stems from substantial reduction in the number of auxiliary variables in the arithmetizations, making the witness w correspondingly smaller. Meanwhile, recall that one contribution to the verifier's costs is a component logarithmic in $|w|$ (§2.2).

7.2 Costs of Spain

We examine the relative costs of the protocol phases, for the prover and verifier. Figures 5 and 6 depict the results. For most benchmarks, DARK is the dominant cost. For GPT-2, witness generation (compute w) has a larger share of prover time, due almost entirely to the difference in complexity between executing matrix multiplication (which must happen to generate the witness) and checking matrix multiplication with the Freivalds algorithm (which is what the constraints enforce, as described in the GPT-2 benchmark earlier).

The prover's primary resource bottleneck is memory. For example, with GPT-2, our largest benchmark, the prover requires ≈41 GB for *seq*=32, *passes*=1, and ≈267 GB for *seq*=32, *passes*=16. The verifier's bottleneck for small computations (small m) is DARK evaluation. The bottleneck for larger computations is evaluating $\widetilde{A}$, $\widetilde{B}$, and $\widetilde{C}$ at a random point.

7.3 Breaking even and batching

Recall that to justify outsourcing on performance grounds, the verifier must be cheaper than native execution (§1). For the largest linear program, scsd8, Figure 4 shows that Spain's verifier *does* meet this condition. However, for most of the other benchmarks, this condition is not met.

In these cases, Spain must amortize fixed costs (§2.2) via *batching*. This is the SIMD-R1CS model (§4.3, Appx. D.1): the same computation repeated on different input/output pairs. This structure is natural for many numerical computations. For example, when running an LLM, people don't just want one token; they want a whole paragraph (or more...).

Spain's verifier indeed benefits from batching; notice that for the rows in Figure 4 featuring GPT-2 with various *passes*, the verifier's work grows less than linearly in batch size. Indeed, Figure 6 shows that the fixed costs – namely fixed work in DARK (§4.2.3, Appx. C) and evaluating $\widetilde{A}$, $\widetilde{B}$, $\widetilde{C}$ at a random point (Step 6, Figure 2) – are the dominant contribution when verifying single instances, and thus amortization is beneficial. These results agree with the theory. Specifically, Spain's fixed-cost is $O(m)$ for the entire batch (§2.2). Meanwhile, Spain's variable costs comprise a component logarithmic in the total witness size across all instances in the batch and a component linear in the number of variables in *in* and *out* across all instances. Given this cost profile, there is usually a batch size for which, in principle, Spain's verifier breaks even once the batch size crosses a threshold. We say *usually* because, for very small computations with very large *in* and/or *out*, the

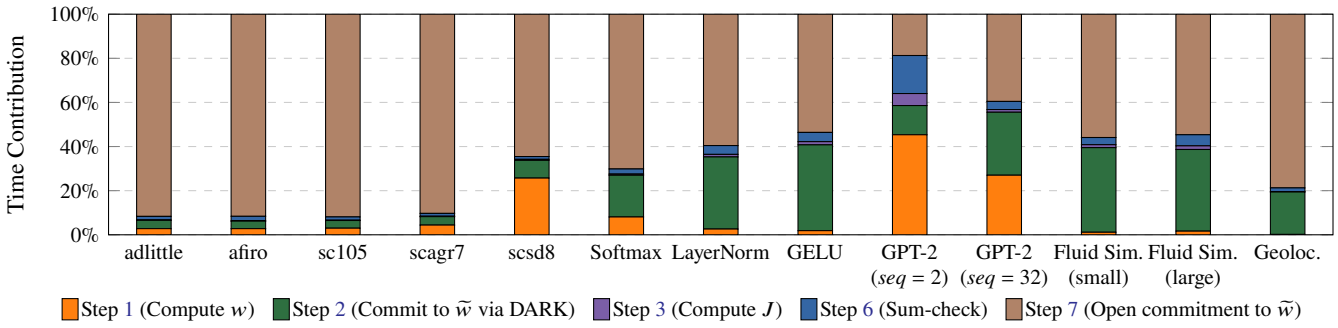


Figure 5: Prover phase breakdown for Spain. Steps refer to those in Figure 2. Steps 4 and 5 are omitted as they incur no cost for the prover.

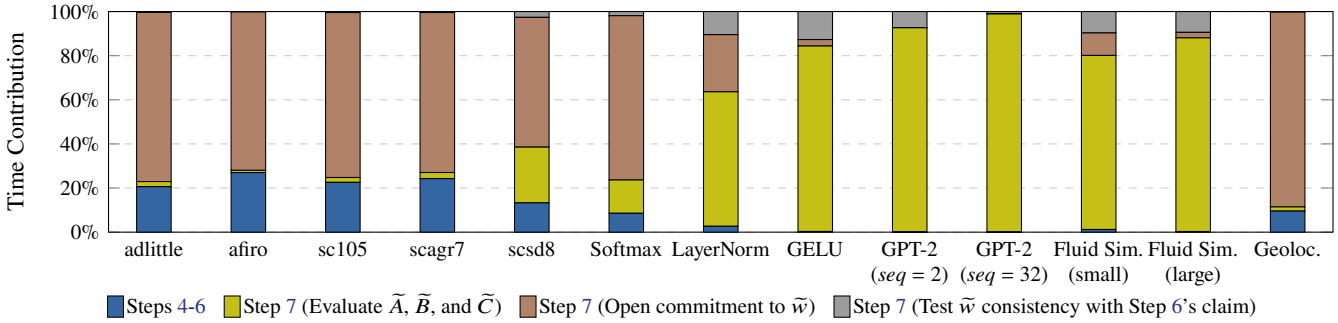


Figure 6: Verifier phase breakdown for Spain. Steps refer to those in Figure 2. Steps 1-3 are omitted as they incur no cost for the verifier. The majority of the verifier's time is spent in Step 7, which decomposes into three sub-steps: evaluating $\widetilde{A}$, $\widetilde{B}$, $\widetilde{C}$; opening the commitment to $\widetilde{w}$; and checking consistency between the claim in Step 6 about $\widetilde{w}$ and the opening of $\widetilde{w}$, using *in* and *out*.

variable costs alone may exceed the cost of native execution.

In contrast to Spain, zkGPT does not amortize with *passes*; as shown in Figure 4 (the GPT-2 rows), zkGPT's verifier is more expensive than Spain's verifier at *seq*=32, *passes*=16. For ZKLP, the comparison is equivocal. On the one hand, ZKLP's fixed costs (which are not included in Figure 4) amortize over all future uses of an R1CS structure, not merely a synchronized batch as in Spain; on the other hand, ZKLP's fixed costs are dramatically higher (over 50 thousand times larger than the verifier's work on a single instance [112]).

A loose end is *setup costs*, as distinct from fixed costs (§2.2); these are not depicted. However, on our benchmarks, Spain's setup costs range from negligible to 5× the verifier's work to check a single instance, so this cost quickly amortizes.

7.4 Summary and discussion

We believe that Spain meets the goal of being general-purpose (§1): it easily handled an array of natural benchmarks, and we did not have to discard any benchmarks because Spain lacked sufficient expressiveness.

At its core, Spain bakes approximation into the constraints themselves, eliminating the need to enforce numerical semantics explicitly, which in turn improves on the number of constraints (m) in an arithmetization, by 32× to 17,000× (§7.1).

This reduction lowers the prover overhead versus natural baselines by 1–3 orders of magnitude, down to 3–5 orders of magnitude versus native execution (§7.1). Spain's prover isn't the fastest in the literature; that honor belongs to zkGPT [105], but as we have noted, zkGPT is specialized to a specific machine learning model architecture, and its verifier neither breaks even nor benefits from amortization (§7.3).

Spain's verifier beats native execution for some reasonably-sized numerical problems (§7.3), and is the only system we experimented with to do so. Given their cost profiles (§7.2), Spain's verifier and prover would directly benefit from improvements in integer polynomial commitment protocols or rational ones [6, 121]. There are additional avenues for improving the verifier's costs. Most saliently, the verifier could shift some work to the prover using SPARK [116] or one of its descendants [38, 117, 119].

A structural disadvantage of Spain is that, for the time being, its new arithmetizations *require* its back-end. Traditional constraints are more portable; they can be combined with back-ends that provide different properties. Otti's back-end, for example, provides zero-knowledge and non-interactivity. ZKLP's back-end does too, and it inherits from Groth16 [70] an extremely fast verifier (albeit with high fixed costs; §7.3). If Spain's arithmetizations could be ported to alternative back-ends (§9), Spain could derive end-to-end speedups and enhanced properties.

8 Other related work

We have described Spain's intellectual antecedents throughout. Here we focus on the intersection of proof systems and numerical computations; we divide the work into two strands.

General-purpose (§1) proof systems. The first attempt to represent numerical operations in succinct proofs was Ginger [115], which supported fixed-point and primitive floating-point operations, by mapping them to operations in a finite field $\mathbb{F}_q$. Ginger used static analysis, bit-wise decompositions of values, and a large field (large q) to ensure a unique representation of each rational encountered during execution.

Subsequent works applied variations on this encoding to specific numerical problems, for example Otti [9] to linear programs, zkQMC [50] to Monte Carlo simulations, and KL24 [82] to transcendental functions. All of these works target general-purpose back-ends [8, 21–24, 46, 47, 49, 58, 62, 70, 85, 90, 100, 116, 117] and express computations as traditional constraints over a finite field.

A few recent works have explored alternative approaches, focusing on arithmetizing floating-point arithmetic. The works in this vein range between strict IEEE 754 compliance with support for all rounding modes and exceptions [51] (see §7) and more relaxed semantics [59]. All such works translate numerical operations using more constraints than Spain; however, they offer relative error guarantees that Spain does not.

Recently, interest has renewed in a class of proof systems known as *zkVMs* [15, 25, 26, 53, 83, 145]. These proof systems allow for proving the correct execution of any programs written in a given instruction set, typically RISC-V [107]. Although RISC-V includes numerical operations via floating-point instructions in an extension, we do not know of any zkVMs that currently support this extension.

Like Spain, some works have back-ends that work over domains other than finite fields. Chen et al. [45] arithmetize numerical computations as arithmetic circuits (addition and multiplication gates) over Galois rings, by encoding a rounding operation as a sequence of additions and multiplications within the ring. They then run GKR [66, 67] (§2.2) over the Galois ring. Bitan et al. [27] observe that some sum-check protocols can be modified so that the protocol messages themselves become approximate. Although we have yet to do a detailed analysis, we estimate that the costs of these approaches would be in between Spain's and the approaches derived from Ginger.

Finally, concurrent with this work, others have refined Zaratan [42] (discussed in §4.2), which is geared to integer computations. These works design polynomial commitment protocols over rational numbers that are significantly faster than DARK [6, 121].[2] While these works apply their faster polynomial commitments to general-purpose proof systems for exact integer and rational arithmetic, rather than approximate computations, we observe that the polynomial commitments themselves are compatible with Spain. We anticipate that replacing DARK with the fastest of them will improve the most expensive components of Spain by at least an order of magnitude (§7.2).

Special-purpose proof systems. Here we consider systems designed around specific applications. These systems rely on the observation that checkers for some numerical computations can be directly represented as sum-check protocols (§2.2).

The first in this line of work is Thaler's protocol for proving matrix multiplication [129]. Some works use Thaler's protocol as a primitive in proof systems for machine learning [19, 63, 77, 78, 86, 105, 127, 128, 151] (see the survey of Peng et. al [101] for a taxonomy). These protocols typically quantize numerical values, and alternate between checking exact operations (such as matrix multiplications, convolutions, and activation functions) and rounding to return to a quantized domain. From our testing, the most efficient of these systems is zkGPT [105], which was discussed and evaluated (§7).

[2] An older work in this family, Zinc [60], relies on results from another paper [28]. However, under scrutiny, it turned out that the latter paper's performance claims are about a protocol without any provable guarantees.

9 Conclusion

Spain meets the goal of proving numerical computations in a general-purpose way, while in some regimes meeting the goals (§1) of a prover with 3 orders of magnitude overhead and a verifier less expensive than native (§7.4).

Spain also introduces a new form of arithmetization, approximate constraints (§5). We believe this is of independent interest, as efficient arithmetizations are an object of intense study among those interested in succinct proofs. In principle, approximate constraints are usable in conjunction with the two dominant forms of arithmetization: traditional constraints and lookup tables (§E.4, §2.2).

A natural area for future work is adapting other proof back-ends (both those based on sum-check protocols and those *not* based on sum-check protocols) to handle approximate constraints. Other future work includes reducing the cost of polynomial commitment in Spain, slashing the cost of Spain's verifier by shifting work to the prover (§7.4), making Spain zero-knowledge and non-interactive, supporting relative error in constraints, and adding numerical support to zkVMs (§8) [15, 53, 83, 145]. Many of these are separate research efforts, which we hope our work here has motivated.

Acknowledgments

We are grateful to our shepherd, Andi Quinn, who went above and beyond with close readings that fundamentally improved the presentation of this work. In a similar vein, this paper was substantially improved by the thoughtful suggestions of the anonymous OSDI reviewers. Andrew Blumberg gave useful comments. All errors and problems are the fault of the authors. This project used computing resources at the Courant Institute and NYU's HPC organization, and was partially supported by gifts from Google and Stellar.

References

[1] Carmichael function. https://en.wikipedia.org/wiki/Carmichael_function. Wikipedia, Wikimedia foundation.

[2] IEEE standard for floating-point arithmetic. *IEEE Std 754-2019 (Revision of IEEE 754-2008)*, pages 1–84, 2019.

[3] FLINT: Fast Library for Number Theory. http://flintlib.org, 2025.

[4] GNU MP: The GNU Multiple Precision Arithmetic Library. https://gmplib.org, 2025.

[5] MPFR: The GNU Multiple Precision Floating-Point Reliably Library. https://www.mpfr.org, 2025.

[6] Alexander Abdugafarov, Albert Garreta, Amit Kumar, Michał Osadnik, Psi Vesely, Ilia Vlasov, and Kai Zhe Zheng. Zinc+: SNARKs for polynomial rings. Cryptology ePrint Archive, Paper 2026/855, 2026.

[7] Miguel Ambrona, Anne-Laure Schmitt, Raphael R. Toledo, and Danny Willems. New optimization techniques for PlonK's arithmetization. Cryptology ePrint Archive, Paper 2022/462, 2022.

[8] Scott Ames, Carmit Hazay, Yuval Ishai, and Muthuramakrishnan Venkitasubramaniam. Ligero: Lightweight sublinear arguments without a trusted setup. In *ACM Conference on Computer and Communications Security (CCS)*, 2017.

[9] Sebastian Angel, Andrew J. Blumberg, Eleftherios Ioannidis, and Jess Woods. Efficient representation of numerical optimization problems for SNARKs. In *SECURITY*, 2022.

[10] Sebastian Angel, Andrew J. Blumberg, Eleftherios Ioannidis, and Jess Woods. Otti: A zkSNARK compiler, solver, prover and verifier for optimization problems. https://github.com/eniac/otti, 2022.

[11] arkworks contributors. arkworks zkSNARK ecosystem. https://arkworks.rs, 2022.

[12] Sanjeev Arora and Boaz Barak. *Computational Complexity: A modern approach*. Cambridge University Press, 2009.

[13] Sanjeev Arora, Carsten Lund, Rajeev Motwani, Madhu Sudan, and Mario Szegedy. Proof verification and the hardness of approximation problems. *Journal of the ACM*, 45(3):501–555, May 1998.

[14] Sanjeev Arora and Shmuel Safra. Probabilistic checking of proofs: a new characterization of NP. *Journal of the ACM*, 45(1):70–122, January 1998.

[15] Arasu Arun, Srinath Setty, and Justin Thaler. Jolt: SNARKs for virtual machines via lookups. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques (EUROCRYPT)*, 2024.

[16] László Babai. Trading group theory for randomness. In *ACM Symposium on the Theory of Computing (STOC)*, May 1985.

[17] László Babai, Lance Fortnow, Leonid A Levin, and Mario Szegedy. Checking Computations in Polylogarithmic Time. In *ACM STOC*, 1991.

[18] László Babai, Lance Fortnow, and Carsten Lund. Nondeterministic exponential time has two-prover interactive protocols. *computational complexity*, 1(1):3–40, Mar 1991.

[19] David Balbás, Dario Fiore, Maria Vasco, Damien Robissout, and Claudio Soriente. Modular sumcheck proofs with applications to machine learning and image processing. In *ACM Conference on Computer and Communications Security (CCS)*, 2023.

[20] Anubhav Baweja, Pratyush Mishra, Tushar Mopuri, Karan Newatia, and Steve Wang. Scribe: Low-memory SNARKs via read-write streaming. Cryptology ePrint Archive, Paper 2024/1970, 2024.

[21] E. Ben-Sasson, A. Chiesa, E. Tromer, and M. Virza. Scalable zero knowledge via cycles of elliptic curves. In *IACR International Cryptology Conference (CRYPTO)*, August 2014.

[22] Eli Ben-Sasson, Iddo Bentov, Yinon Horesh, and Michael Riabzev. Fast Reed-Solomon Interactive Oracle Proofs of Proximity. In *ICALP*, 2018.

[23] Eli Ben-Sasson, Iddo Bentov, Yinon Horesh, and Michael Riabzev. Scalable, transparent, and post-quantum secure computational integrity. Cryptology ePrint Archive, Report 2018/046, 2018.

[24] Eli Ben-Sasson, Alessandro Chiesa, Daniel Genkin, Eran Tromer, and Madars Virza. SNARKs for C: Verifying program executions succinctly and in zero knowledge. In *IACR International Cryptology Conference (CRYPTO)*, August 2013.

[25] Eli Ben-Sasson, Alessandro Chiesa, Daniel Genkin, Eran Tromer, and Madars Virza. TinyRAM architecture specification, v0.991, 2013.

[26] Eli Ben-Sasson, Alessandro Chiesa, Eran Tromer, and Madars Virza. Succinct non-interactive zero knowledge for a von Neumann architecture. In *USENIX Security*, 2014.

[27] Dor Bitan, Zachary DeStefano, Shafi Goldwasser, Yuval Ishai, Yael Tauman Kalai, and Justin Thaler. Sum-check protocol for approximate computations. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques (EUROCRYPT)*, 2026.

[28] Alexander R. Block, Zhiyong Fang, Jonathan Katz, Justin Thaler, Hendrik Waldner, and Yupeng Zhang. Field-Agnostic SNARKs from Expand-Accumulate Codes. In *IACR International Cryptology Conference (CRYPTO)*, 2024.

[29] Alexander R. Block, Justin Holmgren, Alon Rosen, Ron D. Rothblum, and Pratik Soni. Time- and Space-Efficient Arguments from Groups of Unknown Order. In *IACR International Cryptology Conference*

(CRYPTO), 2021.

[30] Andrew J. Blumberg, Justin Thaler, Victor Vu, and Michael Walfish. Verifiable computation using multiple provers. Cryptology ePrint Archive, Paper 2014/846, 2014.

[31] Jonathan Bootle, Andrea Cerulli, Jens Groth, Sune Jakobsen, and Mary Maller. Arya: Nearly Linear-Time Zero-Knowledge Proofs for Correct Program Execution. In *ASIACRYPT*, 2018.

[32] Gautam Botrel, Thomas Piellard, Youssef El Housni, Ivo Kubjas, and Arya Tabaie. Consensys/gnark: v0.11.0, September 2024.

[33] Sean Bowe, Jack Grigg, and Daira Hopwood. Recursive proof composition without a trusted setup. Cryptology ePrint Archive, Paper 2019/1021, 2019.

[34] G. Brassard, D. Chaum, and C. Crépeau. Minimum disclosure proofs of knowledge. *Journal of Computer and System Sciences*, 37(2):156–189, October 1988.

[35] B. Braun, A. J. Feldman, Z. Ren, S. Setty, A. J. Blumberg, and M. Walfish. Verifying computations with state. In *ACM Symposium on Operating Systems Principles (SOSP)*, November 2013.

[36] Benjamin Braun. Compiling computations to constraints for verified computation. UT Austin Honors thesis HR-12-10, December 2012.

[37] Isaac Brodsky. H3: Uber's hexagonal hierarchical spatial index. Uber Blog, June 2018.

[38] Benedikt Bünz, Jessica Chen, Zachary DeStefano, and Binyi Chen. Almost linear-time permutation check. Cryptology ePrint Archive, Paper 2025/1850, 2025.

[39] Benedikt Bünz and Ben Fisch. Multilinear Schwartz-Zippel mod N and lattice-based succinct arguments. In *Theory of Cryptography Conference (TCC)*, 2023.

[40] Benedikt Bünz, Ben Fisch, and Alan Szepieniec. Transparent SNARKs from DARK compilers. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques (EUROCRYPT)*, 2020.

[41] Matteo Campanelli, Dario Fiore, and Rosario Gennaro. Natively compatible super-efficient lookup arguments and how to apply them: Natively compatible super-efficient lookup arguments. *J. Cryptol.*, 38(1), January 2025.

[42] Matteo Campanelli and Mathias Hall-Andersen. Fully-succinct arguments over the integers from first principles. Cryptology ePrint Archive, Paper 2024/1548, 2024.

[43] R. D. Carmichael. Note on a new number theory function. *Bulletin of the American Mathematical Society*, 16(5):232–238, 1910.

[44] Binyi Chen, Benedikt Bünz, Dan Boneh, and Zhenfei Zhang. HyperPlonk: Plonk with Linear-Time Prover and High-Degree Custom Gates. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques (EUROCRYPT)*, 2023.

[45] Shuo Chen, Jung Hee Cheon, Dongwoo Kim, and Daejun Park. Interactive proofs for rounding arithmetic. *IEEE Access*, 10, 2022.

[46] Alessandro Chiesa, Yuncong Hu, Mary Maller, Pratyush Mishra, Noah Vesely, and Nicholas Ward. Marlin: Preprocessing zkSNARKs with universal and updatable SRS. In *IACR International Cryptology Conference (CRYPTO)*, 2020.

[47] Alessandro Chiesa, Dev Ojha, and Nicholas Spooner. Fractal: Post-quantum and transparent recursive proofs from holography. In *IACR International Cryptology Conference (CRYPTO)*, 2020.

[48] Graham Cormode, Michael Mitzenmacher, and Justin Thaler. Practical verified computation with streaming interactive proofs. In *Innovations in Theoretical Computer Science (ITCS)*, pages 90–112, January 2012.

[49] Craig Costello, Cédric Fournet, Jon Howell, Markulf Kohlweiss, Benjamin Kreuter, Michael Naehrig, Bryan Parno, and Samee Zahur. Geppetto: Versatile verifiable computation. In *IEEE Symposium on Security and Privacy*, May 2015.

[50] Zachary DeStefano, Dani Barrack, and Michael Dixon. zkQMC: Zero-knowledge proofs for (some) probabilistic computations using quasi-randomness. Cryptology ePrint Archive, Paper 2022/1007, 2022.

[51] Jens Ernstberger, Chengru Zhang, Luca Ciprian, Philipp Jovanovic, and Sebastian Steinhorst. Zero-knowledge location privacy via accurate floating-point SNARKs. In *IEEE Symposium on Security and Privacy*, 2025.

[52] Jens Ernstberger, Chengru Zhang, Luca Ciprian, Philipp Jovanovic, and Sebastian Steinhorst. Zero-knowledge location privacy via accurate floating-point SNARKs. https://github.com/tumberger/zk-Location, 2025.

[53] Ethereum Foundation. Ethereum foundation zkVM list. https://ethproofs.org/zkvms, 2025.

[54] Filecoin Foundation. Filecoin: A decentralized, efficient, and robust foundation for humanity's information. https://filecoin.io.

[55] Mina Foundation. Mina protocol. https://minaprotocol.com, 2026.

[56] R. Freivalds. Probabilistic machines can use less running time. In *Proceedings of the IFIP Congress*, pages 839–842, 1977.

[57] Ariel Gabizon and Zachary J. Williamson. plookup: A simplified polynomial protocol for lookup tables. Cryptology ePrint Archive, Report 2020/315, 2020.

[58] Ariel Gabizon, Zachary J. Williamson, and Oana Ciobotaru. PLONK: Permutations over Lagrange-bases for Oecumenical Noninteractive arguments of Knowledge. Cryptology ePrint Archive, Report 2019/953, 2019.

[59] Sanjam Garg, Abhishek Jain, Zhengzhong Jin, and Yinuo Zhang. Succinct zero knowledge for floating

point computations. In *ACM Conference on Computer and Communications Security (CCS)*, 2022.

[60] Albert Garreta, Hendrik Waldner, Katerina Hristova, and Luca Dall'Ava. Zinc: Succinct arguments with small arithmetization overheads from IOPs of proximity to the integers. Cryptology ePrint Archive, Paper 2025/316, 2025.

[61] David Gay. netlib. https://portal.ampl.com/~dmg/netlib/lp/data/, 2013.

[62] Rosario Gennaro, Craig Gentry, Bryan Parno, and Mariana Raykova. Quadratic span programs and succinct NIZKs without PCPs. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques (EUROCRYPT)*, May 2013.

[63] Zahra Ghodsi, Tianyu Gu, and Siddharth Garg. SafetyNets: verifiable execution of deep neural networks on an untrusted cloud. International Conference on Neural Information Processing Systems (NIPS), 2017.

[64] Oded Goldreich. Probabilistic proof systems – a primer. *Foundations and Trends in Theoretical Computer Science*, 3(1), 2008.

[65] S. Goldwasser, S. Micali, and C. Rackoff. The knowledge complexity of interactive proof systems. *SIAM Journal on Computing*, 18(1):186–208, 1989.

[66] Shafi Goldwasser, Yael Tauman Kalai, and Guy N. Rothblum. Delegating computation: Interactive proofs for muggles. In *ACM Symposium on the Theory of Computing (STOC)*, May 2008.

[67] Shafi Goldwasser, Yael Tauman Kalai, and Guy N Rothblum. Delegating computation: interactive proofs for muggles. *J. ACM*, 62(4), 2015.

[68] Alexander Golovnev, Jonathan Lee, Srinath Setty, Justin Thaler, and Riad S. Wahby. Brakedown: Linear-time and field-agnostic SNARKs for R1CS. In *CRYPTO*, 2023.

[69] Google. Longfellow ZK. https://github.com/google/longfellow-zk.

[70] Jens Groth. On the size of pairing-based non-interactive arguments. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques (EUROCRYPT)*, 2016.

[71] Ulrich Habock. Multivariate lookups based on logarithmic derivatives. Cryptology ePrint Archive, Paper 2022/1530, 2022.

[72] Daira-Emma Hopwood, Sean Bowe, Taylor Hornby, and Nathan Wilcox. Zcash protocol specification. https://zips.z.cash/protocol/protocol.pdf, 2025.

[73] StarkWare Industries. Starkware. https://starkware.co.

[74] Hao Ji, Michael Mascagni, and Yaohang Li. Gaussian variant of freivalds' algorithm for efficient and reliable matrix product verification. *CoRR*, abs/1705.10449, 2017.

[75] Kunming Jiang, Fraser Brown, and Riad S. Wahby. CoBBL: Dynamic Constraint Generation for SNARKs. In *IEEE Symposium on Security and Privacy*, 2025.

[76] Kunming Jiang, Devora Chait-Roth, Zachary DeStefano, Michael Walfish, and Thomas Wies. Less is more: refinement proofs for probabilistic proofs. In *IEEE Symposium on Security and Privacy*, 2023.

[77] Chanyang Ju, Hyeonbum Lee, Heewon Chung, Jae Hong Seo, and Sungwook Kim. Efficient sum-check protocol for convolution. *IEEE Access*, 9:164047–164059, 2021.

[78] Daniel Kang, Tri Dao, and Matei Zaharia. ZKML: An optimizing system for ML inference in zero-knowledge proofs. *Proceedings of the ACM SIGMOD Conference*, 2024.

[79] Aniket Kate, Gregory M. Zaverucha, and Ian Goldberg. Constant-size commitments to polynomials and their applications. In *ASIACRYPT*, 2010.

[80] J. Kilian. A note on efficient zero-knowledge proofs and arguments (extended abstract). In *ACM Symposium on the Theory of Computing (STOC)*, pages 723–732, May 1992.

[81] Ahmed Kosba, Charalampos Papamanthou, and Elaine Shi. xJsnark: a framework for efficient verifiable computation. In *IEEE Symposium on Security and Privacy*, 2018.

[82] Kaarel August Kurik and Peeter Laud. Novel approximations of elementary functions in zero-knowledge proofs. Cryptology ePrint Archive, Paper 2024/859, 2024.

[83] Succinct Labs. SP1 zkVM. https://github.com/succinctlabs/sp1, 2025.

[84] Ryan Lavin, Xuekai Liu, Hardhik Mohanty, Logan Norman, Giovanni Zaarour, and Bhaskar Krishnamachari. A survey on the applications of zero-knowledge proofs, 2026.

[85] Jonathan Lee. Dory: Efficient, transparent arguments for generalised inner products and polynomial commitments. In *IACR Theory of Cryptography Conference (TCC)*, 2021.

[86] Seunghwa Lee, Hankyung Ko, Jihye Kim, and Hyunok Oh. vCNN: Verifiable convolutional neural network based on zk-SNARKs. *IEEE Transactions on Dependable and Secure Computing*, 21(4):4254–4270, 2024.

[87] Tianyi Liu, Tiancheng Xie, Jiaheng Zhang, Dawn Song, and Yupeng Zhang. Pianist: Scalable zkRollups via fully distributed zero-knowledge proofs. In *IEEE Symposium on Security and Privacy*, pages 1777–1793, Los Alamitos, CA, USA, May 2024. IEEE Computer Society.

[88] Loopring Foundation. Loopring: Ethereum's First zkRollup Layer 2. https://loopring.org.

[89] Carsten Lund, Lance Fortnow, Howard J. Karloff, and Noam Nisan. Algebraic methods for interactive proof systems. *Journal of the ACM*, 39(4):859–868, 1992.

[90] Mary Maller, Sean Bowe, Markulf Kohlweiss, and Sarah Meiklejohn. Sonic: Zero-knowledge SNARKs from linear-size universal and updatable structured reference strings. In *ACM Conference on Computer and Communications Security (CCS)*, 2019.

[91] Silvio Micali. Computationally sound proofs. *SIAM Journal on Computing*, 30(4):1253–1298, 2000.

[92] Microsoft. Spartan: High-speed zero-knowledge SNARKs without trusted setup. https://github.com/microsoft/Spartan, 2025.

[93] Vineet Nair, Justin Thaler, and Michael Zhu. Proving CPU executions in small space. Cryptology ePrint Archive, Paper 2025/611, 2025.

[94] Ihyun Nam. A survey of multivariate polynomial commitment schemes. arXiv:2306.11383, 2023.

[95] ONNX Contributors. Open Neural Network Exchange Intermediate Representation (ONNX IR) Specification. https://github.com/onnx/onnx/blob/main/docs/IR.md, 2026.

[96] Alex Ozdemir, Fraser Brown, and Riad Wahby. CirC: Compiler infrastructure for proof systems, software verification, and more. In *IEEE Symposium on Security and Privacy*, 2022.

[97] Alex Ozdemir, Evan Laufer, and Dan Boneh. Volatile and persistent memory for zksnarks via algebraic interactive proofs. In *IEEE Symposium on Security and Privacy*, 2025.

[98] H. Padé. Sur la représentation approchée d'une fonction par des fractions rationnelles. *Annales scientifiques de l'École Normale Supérieure*, 3e série, 9:3–93, 1892.

[99] Christodoulos Pappas and Dimitrios Papadopoulos. HOBBIT: space-efficient zkSNARK with optimal prover time. In *USENIX Security*, 2025.

[100] Bryan Parno, Craig Gentry, Jon Howell, and Mariana Raykova. Pinocchio: Nearly practical verifiable computation. In *IEEE Symposium on Security and Privacy*, May 2013.

[101] Zhizhi Peng, Chonghe Zhao, Taotao Wang, Guofu Liao, Zibin Lin, Yifeng Liu, Bin Cao, Long Shi, Qing Yang, and Shengli Zhang. A survey of zero-knowledge proof based verifiable machine learning. *Artificial Intelligence Review*, Apr 2026.

[102] Pepper Project. Pequin: An end-to-end toolchain for verifiable computation, SNARKs, and probabilistic proofs. https://github.com/pepper-project/pequin, 2018.

[103] Polygon Labs UI. Polygon. https://polygon.technology.

[104] William H. Press, Brian P. Flannery, Saul A. Teukolsky, and William T. Vetterling. *Numerical Recipes in C: The Art of Scientific Computing*. Cambridge University Press, 2 edition, October 1992.

[105] Wenjie Qu, Yijun Sun, Xuanming Liu, Tao Lu, Yanpei Guo, Kai Chen, and Jiaheng Zhang. zkGPT: an efficient non-interactive zero-knowledge proof framework for LLM inference. In *USENIX Security*, 2025.

[106] Alec Radford, Jeff Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. Language models are unsupervised multitask learners. https://cdn.openai.com/better-language-models/language_models_are_unsupervised_multitask_learners.pdf, 2019.

[107] RISC-V International. RISC-V: Ratified Specification. https://riscv.org/specifications/ratified/, 2025.

[108] Michael Rosenberg, Tushar Mopuri, Hossein Hafezi, Ian Miers, and Pratyush Mishra. Hekaton: Horizontally-scalable zkSNARKs via proof aggregation. In *ACM Conference on Computer and Communications Security (CCS)*, 2024.

[109] J. Barkley Rosser and Lowell Schoenfeld. Approximate formulas for some functions of prime numbers. *Illinois Journal of Mathematics*, 6:64–94, 1962.

[110] Kevin Sahr, Denis White, and Jon A. Kimerling. Geodesic discrete global grid systems. *Cartography and Geographic Information Science*, 30:121–134, 2003.

[111] J. T. Schwartz. Fast probabilistic algorithms for verification of polynomial identities. *J. ACM*, 27(4):701–717, October 1980.

[112] SCIPR Lab. libsnark ppzkSNARK empirical performance README. https://github.com/scipr-lab/libsnark/blob/master/libsnark/zk_proof_systems/ppzksnark/README.md, 2017. GitHub repository documentation.

[113] SCIPR Lab and contributors. libsnark: a C++ library for zksnark proofs. https://github.com/scipr-lab/libsnark, 2026.

[114] S. Setty, B. Braun, V. Vu, A. J. Blumberg, B. Parno, and M. Walfish. Resolving the conflict between generality and plausibility in verified computation. In *European Conference on Computer Systems (EuroSys)*, pages 71–84, April 2013.

[115] S. Setty, V. Vu, N. Panpalia, B. Braun, A. J. Blumberg, and M. Walfish. Taking proof-based verified computation a few steps closer to practicality. In *USENIX Security*, August 2012.

[116] Srinath Setty. Spartan: Efficient and general-purpose zkSNARKs without trusted setup. In *CRYPTO*, 2020.

[117] Srinath Setty and Jonathan Lee. Quarks: Quadruple-efficient transparent zkSNARKs. Cryptology ePrint Archive, Paper 2020/1275, 2020.

[118] Srinath Setty, Justin Thaler, and Riad Wahby. Customizable constraint systems for succinct arguments. Cryptology ePrint Archive, Paper 2023/552, 2023.

[119] Srinath Setty, Justin Thaler, and Riad Wahby. Unlocking the Lookup Singularity with Lasso. In *Annual International Conference on the Theory and Applications of*

Cryptographic Techniques (EUROCRYPT), 2024.

[120] Adi Shamir. IP = PSPACE. *Journal of the ACM*, 39(4):869–877, October 1992.

[121] Alireza Shirzad, Sriram Sridhar, Dimitrios Papadopoulos, and Charalampos Papamanthou. Relaxed modular PCS from arbitrary PCS and applications to SNARKs for integers. Cryptology ePrint Archive, Paper 2026/347, 2026.

[122] Michael Sipser. *Introduction to the Theory of Computation*. Cengage Learning, 3 edition, 2013.

[123] Eduardo Soria-Vazquez. Doubly efficient interactive proofs over infinite and non-commutative rings. In *IACR Theory of Cryptography Conference (TCC)*, 2022.

[124] Sriram Sridhar, Shravan Srinivasan, Dimitrios Papadopoulos, and Charalampos Papamanthou. Efficiently provable approximations for non-polynomial functions. In *USENIX Security*, 2026.

[125] Jos Stam. Stable fluids. In *SIGGRAPH*, 1999.

[126] Alan Stapelberg. Opening up 'Zero-Knowledge Proof' technology to promote privacy in age assurance. https://blog.google/technology/safety-security/opening-up-zero-knowledge-proof-technology-to-promote-privacy-in-age-assurance/, July 2025.

[127] Haochen Sun, Tonghe Bai, Jason Li, and Hongyang Zhang. zkDL: Efficient zero-knowledge proofs of deep learning training. *IEEE Transactions on Information Forensics and Security*, 2024.

[128] Haochen Sun, Jason Li, and Hongyang Zhang. zkLLM: Zero knowledge proofs for large language models. In *ACM Conference on Computer and Communications Security (CCS)*, 2024.

[129] Justin Thaler. Time-optimal interactive proofs for circuit evaluation. In *IACR International Cryptology Conference (CRYPTO)*, August 2013.

[130] Justin Thaler. Proofs, Arguments, and Zero-Knowledge. http://people.cs.georgetown.edu/jthaler/ProofsArgsAndZK.html, 2023.

[131] Justin Thaler. Sum-check is all you need: An opinionated survey on fast provers in SNARK design. Cryptology ePrint Archive, Paper 2025/2041, 2025.

[132] The Linux Foundation. ONNX: Open Neural Network Exchange. https://onnx.ai, 2019.

[133] Louis Tremblay Thibault, Tom Sarry, and Abdelhakim Senhaji Hafid. Blockchain scaling using rollups: A comprehensive survey. *IEEE Access*, 10:93039–93054, 2022.

[134] Ioanna Tzialla, Abhiram Kothapalli, Bryan Parno, and Srinath Setty. Transparency dictionaries with succinct proofs of correct operation. In *Network and Distributed System Security Symposium (NDSS)*, 2022. https://eprint.iacr.org/2021/1263.

[135] V. Vu, S. Setty, A. J. Blumberg, and M. Walfish. A hybrid architecture for interactive verifiable computation. In *IEEE Symposium on Security and Privacy*, May 2013.

[136] Riad S. Wahby, Max Howald, Siddharth Garg, abhi shelat, and Michael Walfish. Verifiable ASICs. In *IEEE Symposium on Security and Privacy*, 2016.

[137] Riad S. Wahby, Ye Ji, Andrew J. Blumberg, abhi shelat, Justin Thaler, Michael Walfish, and Thomas Wies. Full accounting for verifiable outsourcing. In *ACM Conference on Computer and Communications Security (CCS)*, 2017.

[138] Riad S. Wahby, Srinath Setty, Max Howald, Zuocheng Ren, Andrew J. Blumberg, and Michael Walfish. Efficient RAM and control flow in verifiable outsourced computation. In *Network and Distributed System Security Symposium (NDSS)*, 2015.

[139] Riad S. Wahby, Ioanna Tzialla, abhi shelat, Justin Thaler, and Michael Walfish. Doubly-efficient zkSNARKs without trusted setup. In *IEEE Symposium on Security and Privacy*, 2018.

[140] Michael Walfish and Andrew J. Blumberg. Verifying computations without reexecuting them: from theoretical possibility to near practicality. *Communications of the ACM (CACM)*, 58(2), 2015.

[141] Wikimedia Foundation Wikipedia. Localization (commutative algebra). https://en.wikipedia.org/wiki/Localization_(commutative_algebra).

[142] Howard Wu, Wenting Zheng, Alessandro Chiesa, Raluca Ada Popa, and Ion Stoica. DIZK: a distributed zero knowledge proof system. In *USENIX Security*, 2018.

[143] Tiacheng Xie, Jiaheng Zhang, Yupeng Zhang, Charalampos Papamanthou, and Dawn Song. Libra: Succinct Zero-Knowledge Proofs with Optimal Prover Computation. In *IACR International Cryptology Conference (CRYPTO)*, 2019.

[144] Tiancheng Xie, Jiaheng Zhang, Zerui Cheng, Fan Zhang, Yupeng Zhang, Yongzheng Jia, Dan Boneh, and Dawn Song. zkBridge: Trustless cross-chain bridges made practical. In *ACM Conference on Computer and Communications Security (CCS)*, 2022.

[145] RISC Zero. RISC0 zkVM. https://github.com/risc0/risc0, 2025.

[146] Collin Zhang, Zachary DeStefano, Arasu Arun, Joseph Bonneau, Paul Grubbs, and Michael Walfish. Zombie: Middleboxes that don't snoop. In *Symposium on Networked Systems Design and Implementation (NSDI)*, 2024.

[147] Yupeng Zhang, Daniel Genkin, Jonathan Katz, Dimitrios Papadopoulos, and Charalampos Papamanthou. vSQL: Verifying arbitrary SQL queries over dynamic outsourced databases. In *IEEE Symposium on Security and Privacy*, 2017.

[148] Yupeng Zhang, Daniel Genkin, Jonathan Katz, Dimitrios Papadopoulos, and Charalampos Papamanthou.

A zero-knowledge version of vSQL. Cryptology ePrint Archive, Paper 2017/1146, 2017.

[149] Richard Zippel. Probabilistic algorithms for sparse polynomials. In Edward W. Ng, editor, *Symbolic and Algebraic Computation*, pages 216–226. Springer Berlin Heidelberg, 1979.

[150] zkcrypto. bellman: zk-SNARK library, 2026.

[151] Zkonduit. Easy Zero-Knowledge Inference. https://github.com/zkonduit/ezkl, 2024.

A Sum-check primitive

Given a prover, $\mathcal{P}$, and a verifier, $\mathcal{V}$, the sum-check primitive of Lund et al. [89] is an interactive protocol for proving that a claimed value H is equal to the sum of evaluations of a v-variate polynomial G over the Boolean hypercube: $\{0,1\}^v$. This primitive is depicted in Figure 7.

Spain relies on the following lemma concerning the soundness and completeness of this primitive [122, Chapter 10.4] [12, Chapter 8.5] [130, Chapter 4.1].

Lemma 1. *The primitive in Figure 7 is an interactive proof (the term is defined in the references above) for the claim that* $H = \sum_{b\in\{0,1\}^v} G(b)$. *If the initial claim is false,* $\mathcal{V}$ *rejects with probability at least* $1 - sd/q$. *If the initial claim is true,* $\mathcal{P}$ *can make* $\mathcal{V}$ *accept with probability* 1.

Setup: $\mathcal{P}$ and $\mathcal{V}$ agree on a field $\mathbb{F}_q$ and a degree d multi-variate polynomial G over $\mathbb{F}_q$ with v variables.
Online phase: If at any point $\mathcal{V}$'s check fails, it rejects.

1. $\mathcal{P}$ sends $\mathcal{V}$ a value H, claimed to equal
$$\sum_{b\in\{0,1\}^v} G(b)$$
and a univariate polynomial $g_1(x)$, claimed to equal
$$\sum_{(b_2,\ldots,b_s)\in\{0,1\}^{v-1}} G(x, b_2, \ldots, b_s).$$
2. $\mathcal{V}$ checks that $H = g_1(0) + g_1(1)$.
3. $\mathcal{V}$ sends $\mathcal{P}$ a random $r_1 \in \mathbb{F}_q$.
4. For $k = 2, \ldots, v$:
   (a) $\mathcal{P}$ sends $\mathcal{V}$ a univariate polynomial $g_k(x)$, claimed to equal
   $$\sum_{(b_{k+1},\ldots,b_s)\in\{0,1\}^{v-k}} G(r_1, \ldots, r_{k-1}, x, b_{k+1}, \ldots, b_s).$$
   (b) $\mathcal{V}$ checks that $g_{k-1}(r_{k-1}) = g_k(0) + g_k(1)$.
   (c) $\mathcal{V}$ sends $\mathcal{P}$ a random $r_k \in \mathbb{F}_q$.
5. Finally, $\mathcal{V}$ checks that $g_s(r_s) = G(r_1, \ldots, r_s)$.

If all checks pass, $\mathcal{V}$ accepts.

Figure 7: The sum-check primitive of Lund et al. [89].

B Spain's full back-end protocol and analysis

Here we provide background on polynomial commitment schemes and define the properties required by Spain (§B.1); provide Spain's full back-end protocol (§B.2) and its correctness proofs (§B.3); quantify the guarantees based on the specific parameters in our implementation and evaluation (§B.4); and finally prove some supporting lemmas (§B.5).

For clarity in exposition, when we mention a polynomial commitment scheme, we are referring to a polynomial commitment scheme for multilinear polynomials. Additionally, when we use the term "coefficients" for a v-variate multilinear polynomial, these refer to the coefficients in the multilinear Lagrange basis [130, §3.5] with the interpolating set being the v-variate Boolean hypercube: $\{0,1\}^v$. These are exactly the *evaluations* of the polynomial over the Boolean hypercube.

B.1 Spain-compatible polynomial commitment schemes

A multilinear polynomial commitment scheme [79, 94] *PC* is a tuple of protocols (*Setup*, *Commit*, *Open*, *Verify*), defined as follows.

- *Setup*: Given security parameter λ and a bound on the number of variables in polynomials to be committed to, produce public parameters *pp*.
- *Commit*: Given public parameters *pp* and a multilinear polynomial $\tilde{f}$, produce a commitment $com_{\tilde{f}}$.
- *Open*: Given public parameters *pp*, a multilinear polynomial $\tilde{f}$, and a point x, produce a result y and a proof π_y that $y = \tilde{f}(x)$. Together, the result and proof are called an *opening* of $com_{\tilde{f}}$ at x.
- *Verify*: Given public parameters *pp*, a commitment $com_{\tilde{f}}$, a point x, and an alleged opening (y, π_y), output "accept" if π_y is a valid proof that $y = \tilde{f}(x)$ and "reject" otherwise.

A polynomial commitment scheme is *binding* if it is intractable for a prover to produce two distinct openings (y, π_y) and $(y', \pi_{y'})$ for the same commitment $com_{\tilde{f}}$ such that the verifier accepts both openings. For Spain, intractable means that a polynomial-time adversary cannot succeed with probability greater than κ_{PC}.

Spain requires, as a building block, a binding polynomial commitment scheme, *PC*, that satisfies an additional property, which we call *Weak Integer Binding*. Weak Integer Binding is parameterized by 2 constants, $N_{\max}$ and $D_{\max}$. Weak Integer Binding has two sub-properties, given below. To state the sub-properties we define *Integer-limited*: a v-variate multilinear polynomial $\tilde{f}$ is called *Integer-limited* if all of its coefficients are integers with magnitude at most $N_{\max}$.

- *Weak Integer Binding, sub-property 1.* It is impossible for a prover to produce a commitment $com_{\tilde{f}}$ unless for some positive integer D' and multilinear polynomial $\tilde{g}$, $\tilde{f} = \tilde{g}/D'$, where $D' \leq D_{\max}$ and $\tilde{g}$ is Integer-limited. (Here $\tilde{g}/D'$

means a polynomial with the same coefficients as $\tilde{g}$ but with an extra D' factor in each coefficient's denominator.)

- *Weak Integer Binding, sub-property 2.* Given an honest (polynomial-time) prover, for any Integer-limited polynomial $\tilde{f}$, invoking *Commit* causes the prover to produce a valid commitment, and for any point x, invoking *Open* causes the prover to produce a corresponding opening that causes *Verify* to accept

Weak Integer Binding is similar to, but weaker than, the notion of a *Relaxed Mod-PCS* introduced by Zinc [60]. A Relaxed Mod-PCS imposes an additional requirement, *extractability* (essentially, that the prover must "know" $\tilde{f}$ of the right form in order to produce $com_{\tilde{f}}$).

Also, note that Weak Integer Binding is an idealized property in that sub-property 1 states that it is *impossible*, rather than computationally *intractable*, for a prover to produce a commitment $com_{\tilde{f}}$ that does not satisfy the stated condition. When instantiating a multilinear commitment primitive (§B.4), we will handle probabilistic deviation from the ideal.

B.2 Spain's back-end protocol

Spain's full protocol is in Figure 8. This protocol is parameterized by a constant ϵ; an R1CS instance, $\mathbb{X}$, containing rationals with unreduced denominator D and unreduced numerators with magnitude at most $N_{\max}$; a polynomial commitment scheme PC satisfying Weak Integer Binding with parameters $N_{\max}$ and $D_{\max}$ (§B.1); and a large set of primes, P. The R1CS structure underlying $\mathbb{X}$ has m constraints and n variables. Recall from Section 4.2.2 that s and k are defined to be $\lceil \log_2 m \rceil$ and $\lceil \log_2 n \rceil$ respectively.

Note that Weak Integer Binding means an honest prover commits to a polynomial $\tilde{f}$ with *integer* coefficients, yet Spain has been described (Fig. 2, §4) as committing to a polynomial $\widetilde{w}$ with *rational* coefficients. In fact, in Step 1 (Fig. 8), Spain's prover obtains $\tilde{f}$ by multiplying $\widetilde{w}$ by D. Meanwhile the elements of w are expected to have unreduced denominator D (§4.2.1) and hence the same holds for the coefficients of $\widetilde{w}$. Thus, $\tilde{f}$ indeed has integer coefficients.

B.3 Correctness of Spain's back-end protocol

In this section, we prove the correctness, meaning completeness and soundness (§4), of Spain's back-end, specifically the protocol in Figure 8. The figure depicts, and the proof is about, an idealized version of Spain in which the polynomial commitment scheme meets the properties in Appendix B.1. In addition to quantifying parameters, Appendix B.4 instantiates the polynomial commitment scheme with a concrete protocol, namely DARK [40], and handles the deviation from the ideal.

En route to the proof, we define $U_{m,n,N_{\max},D,D_{\max}}$ (written as U for brevity) to be

$$U := m \cdot n^2 \cdot N_{\max}^4 \cdot D_{\max}^4 \cdot (n \cdot N_{\max}^2 + D^2)^2 + D^8 \cdot D_{\max}^4.$$

Additionally, we use $\min P$ to denote the smallest prime in P, and $|P|$ to denote the number of primes in P.

The definitions of completeness and soundness (§4) reference ϵ-accuracy, which was described informally in Section 3. Below, we state ϵ-accuracy more formally. Let M_i denote row i of matrix M.

Definition (ϵ-accuracy)**.** *An assignment z for an R1CS structure $\mathbb{S} = (A, B, C)$ is* ϵ-accurate *if*

$$|(A_i \cdot z) \cdot (B_i \cdot z) - (C_i \cdot z)| \leq \epsilon, \text{ for all } i \in \{1, \ldots, m\}.$$

We are now ready to state the central correctness theorem for Spain.

Theorem 1. *The protocol in Figure 8 satisfies Back-end $(\epsilon/\sqrt{m})$-completeness. The protocol also satisfies Back-end ϵ-soundness, with soundness error κ upper-bounded by*

$$\frac{\lfloor \log_{\min P} D_{max} \rfloor + \lfloor \log_{\min P} U \rfloor}{|P|} + \frac{4s + 2 + 2k}{\min P} + \kappa_{PC}.$$

Proof. We use two facts about ρ_q, both proved in Appendix B.5.

Lemma 2 (ρ_q ill-defined)**.** *Let $\mathbb{X}$, z, and $\widetilde{w}$ be as in Figure 8, and let q be sampled uniformly from P. Then*

$$\Pr_{q \sim P}\left[\rho_q \text{ is undefined on } \|E_{\mathbb{X},z}\|_2^2\right] \leq \frac{\lfloor \log_{\min P} D_{max} \rfloor}{|P|}.$$

Lemma 3 (ρ_q collision)**.** *Let $\mathbb{X}$, z, $\widetilde{w}$, and J be as in Figure 8 with $J \neq \|E_{\mathbb{X},z}\|_2^2$, and let q be sampled uniformly from P. Conditioned on ρ_q being defined for $\|E_{\mathbb{X},z}\|_2^2$, we have*

$$\Pr_{q \sim P}\left[\rho_q(J) = \rho_q(\|E_{\mathbb{X},z}\|_2^2)\right] \leq \frac{\lfloor \log_{\min P} U \rfloor}{|P|}.$$

Back-end $(\epsilon/\sqrt{m})$-completeness. If $\mathcal{P}$ has an assignment z that is $(\epsilon/\sqrt{m})$-accurate for the instance $\mathbb{X}$, by the argument of Section 4.1, $\|E_{\mathbb{X},z}\|_2^2 \leq \epsilon^2$. Thus if $\mathcal{P}$ sends messages prescribed by the protocol, every check by $\mathcal{V}$ passes, and $\mathcal{V}$ accepts with probability 1.

Back-end ϵ-soundness. Suppose that there does not exist an ϵ-accurate assignment for $\mathbb{X}$. We bound the probability that $\mathcal{V}$ accepts.

First, an adversarial prover need not use *Commit* in Step 1 to produce $com_{\tilde{f}}$ nor use *Open* in Step 8 to produce an alleged opening (y, π_y) of $com_{\tilde{f}}$ at $\hat{\beta}$. To handle all adversarial prover behavior involving the polynomial commitment, we define $\mathsf{E}_{\mathsf{bind}}$ to capture the event that $\mathcal{V}$ accepts the opening in Step 8 but this opening is inconsistent with the commitment provided in Step 1. By inconsistent, we mean that $y \neq \tilde{f}(\hat{\beta})$. By the binding property of the polynomial commitment scheme,

$$\Pr[\mathsf{E}_{\mathsf{bind}}] \leq \kappa_{\mathrm{PC}}.$$

We now consider the case where $\mathsf{E}_{\mathsf{bind}}$ does not occur, so the binding property of the polynomial commitment scheme fixes a unique $\tilde{f}$ and thus a unique w (and unique z).

Setup: Prover $\mathcal{P}$ and Verifier $\mathcal{V}$ agree on the parameters of the protocol, which are as follows:

- a small constant $\epsilon \ll 1$
- an instance, $\mathbb{X} = (A, B, C, in, out)$, consisting of rationals with unreduced denominator D (a power of 2).
- a polynomial commitment scheme PC satisfying Weak Integer Binding with parameters $N_{\max}$ and $D_{\max}$.
- a large set of primes, P, from which $\mathcal{V}$ will sample in the protocol.

Preprocessing: $\mathcal{P}$ and $\mathcal{V}$ performs the setup required by PC.
Online phase: If at any point $\mathcal{V}$'s check fails, it rejects.

1. Using PC, $\mathcal{P}$ commits to a multilinear polynomial $\tilde{f}$ whose coefficients are purportedly integers. $\mathcal{P}$ sends this commitment to $\mathcal{V}$.
   - Define $\widetilde{w} := \tilde{f}/D$. Define w as the coefficients of $\widetilde{w}$, and define $z := (in, out, 1, w)$. If $\mathcal{P}$ is honest, then w is chosen to make z an ϵ-accurate assignment to $\mathbb{X}$.
2. $\mathcal{P}$ computes $\|E_{\mathbb{X},z}\|_2^2$ (over $\mathbb{Q}$), and sends $S := \|E_{\mathbb{X},z}\|_2^2 \cdot D^8$ to $\mathcal{V}$.
3. $\mathcal{V}$ checks that S is an integer, computes $J := S/D^8$, and checks that $0 \le J \le \epsilon^2$.
4. $\mathcal{V}$ samples a random prime $q \sim P$, sends it to $\mathcal{P}$, and $\mathcal{P}$ and $\mathcal{V}$ map all quantities in $\mathbb{Q}$ into $\mathbb{F}_q$ via ρ_q.
5. $\mathcal{V}$ and $\mathcal{P}$ run a degree-4 sum-check primitive on
$$\rho_q(J) = \sum_{r \in \{0,1\}^s} (\widetilde{g_A}(r) \cdot \widetilde{g_B}(r) - \widetilde{g_C}(r))^2,$$
where $\widetilde{g_A}$, $\widetilde{g_B}$, and $\widetilde{g_C}$ are defined as in Section 4.2.2.
The result of this primitive is three purported claims over $\mathbb{F}_q$:
$$v_a = \widetilde{g_A}(\alpha), \quad v_b = \widetilde{g_B}(\alpha), \quad \text{and} \quad v_c = \widetilde{g_C}(\alpha)$$
for a random $\alpha \in \mathbb{F}_q^s$.
6. $\mathcal{V}$ samples a random $\gamma \in \mathbb{F}_q$, sends γ to $\mathcal{P}$, and computes
$$v \leftarrow v_a + v_b \cdot \gamma + v_c \cdot \gamma^2.$$
7. $\mathcal{P}$ and $\mathcal{V}$ run a degree-2 sum-check primitive on
$$v = \sum_{c \in \{0,1\}^k} (\widetilde{A}(\alpha, c) + \widetilde{B}(\alpha, c) \cdot \gamma + \widetilde{C}(\alpha, c) \cdot \gamma^2) \cdot \widetilde{z}(c).$$
This results in the following four purported claims over $\mathbb{F}_q$:
$$u_a = \widetilde{A}(\alpha, \beta), \quad u_b = \widetilde{B}(\alpha, \beta), \quad u_c = \widetilde{C}(\alpha, \beta), \quad \text{and} \quad u_z = \widetilde{z}(\beta)$$
for a random $\beta \in \mathbb{F}_q^k$.
8. $\mathcal{P}$ maps the point β into $\mathbb{Z}$ as $\hat{\beta}$, opens the commitment to $\tilde{f}$ at $\hat{\beta}$, and sends the result y (and accompanying proof π_y) to $\mathcal{V}$.
9. $\mathcal{V}$ uses π_y to check that the result y is a valid opening; if it is, $y = \tilde{f}(\hat{\beta})$ over the rationals. $\mathcal{V}$ computes $\widetilde{in}(\beta)$ and $\widetilde{out}(\beta)$ (over $\mathbb{F}_q$), and combines them with $y/D = \tilde{f}(\hat{\beta})/D = \widetilde{w}(\hat{\beta})$ (mapped to $\mathbb{F}_q$ via ρ_q) to produce $\tilde{z}(\beta)$. $\mathcal{V}$ then checks the purported claims from the end of Step 7.

If all checks pass, $\mathcal{V}$ accepts.

Figure 8: Spain's back-end protocol for proving that an R1CS instance has an ϵ-accurate assignment.

Because $\mathbb{X}$ has no ϵ-accurate assignment, $\|E_{\mathbb{X},z}\|_\infty > \epsilon$ and thus $\|E_{\mathbb{X},z}\|_2^2 > \epsilon^2$. However, in Step 3, $\mathcal{V}$ accepts only if $J \le \epsilon^2$. Consequently, $J \ne \|E_{\mathbb{X},z}\|_2^2$.

For $\mathcal{V}$ to accept the final check in Step 9, it must be the case that, in some step of the protocol, a false claim is transformed into a true claim. We define one bad event per step that captures this transformation.

- E_{ρ_q}: the event that, in Step 4, $J \ne \|E_{\mathbb{X},z}\|_2^2$, but either ρ_q is undefined on $\|E_{\mathbb{X},z}\|_2^2$ or $\rho_q(J) = \rho_q(\|E_{\mathbb{X},z}\|_2^2)$.
- $\mathsf{E}_{\mathsf{sc1}}$: the event that, in Step 5, $\rho_q(J) \ne \rho_q(\|E_{\mathbb{X},z}\|_2^2)$, but after the first sum-check primitive, the claims $v_a = \widetilde{g_A}(\alpha)$, $v_b = \widetilde{g_B}(\alpha)$, and $v_c = \widetilde{g_C}(\alpha)$ are all true.
- $\mathsf{E}_{\mathsf{agg}}$: the event that, in Step 6, $v_a \ne \widetilde{g_A}(\alpha)$, $v_b \ne \widetilde{g_B}(\alpha)$, or $v_c \ne \widetilde{g_C}(\alpha)$, but the aggregated claim $v = v_a + v_b \cdot \gamma + v_c \cdot \gamma^2$ is true.

- $\mathsf{E}_{\mathsf{sc2}}$: the event that, in Step 7,

$$v \neq \sum_{c \in \{0,1\}^k} (\widetilde{A}(\alpha, c) + \widetilde{B}(\alpha, c) \cdot \gamma + \widetilde{C}(\alpha, c) \cdot \gamma^2) \cdot \widetilde{z}(c),$$

but after the second sum-check primitive, the claims $u_a = \widetilde{A}(\alpha, \beta)$, $u_b = \widetilde{B}(\alpha, \beta)$, $u_c = \widetilde{C}(\alpha, \beta)$, and $u_z = \widetilde{z}(\beta)$ are all true.

If none of these occur, then ρ_q is defined on $\|E_{\mathbb{X},z}\|_2^2$, $\rho_q(J) \neq \rho_q(\|E_{\mathbb{X},z}\|_2^2)$, and each step of the protocol transforms a false claim into a false claim. As a result, in the final step, at least one of the following must be false:

$$u_a = \widetilde{A}(\alpha, \beta)$$
$$u_b = \widetilde{B}(\alpha, \beta)$$
$$u_c = \widetilde{C}(\alpha, \beta)$$
$$u_z = \widetilde{z}(\beta)$$

Either the verifier's explicit evaluation of $\widetilde{A}$, $\widetilde{B}$, or $\widetilde{C}$ at (α, β) will expose a false claim, or the opening of the commitment to $\widetilde{w}$ will expose a false claim about $\widetilde{z}(\beta)$. Hence $\mathcal{V}$ rejects if none of the bad events occur.

By a union bound, the probability that $\mathcal{V}$ accepts is at most the sum of the probabilities of these bad events, plus $\Pr[\mathsf{E}_{\mathsf{bind}}]$. We now bound the probability of each bad event.

For E_{ρ_q}, by Lemma 2, the probability that ρ_q is undefined on $\|E_{\mathbb{X},z}\|_2^2$ is at most

$$\frac{\lfloor \log_{\min P} D_{\max} \rfloor}{|P|}.$$

By Lemma 3, the probability that $\rho_q(J) = \rho_q(\|E_{\mathbb{X},z}\|_2^2)$, conditioned on $J \neq \|E_{\mathbb{X},z}\|_2^2$ and ρ_q being defined on $\|E_{\mathbb{X},z}\|_2^2$, is at most

$$\frac{\lfloor \log_{\min P} U \rfloor}{|P|}.$$

Thus, by a union bound,

$$\Pr[\mathsf{E}_{\rho_q}] \leq \frac{\lfloor \log_{\min P} D_{\max} \rfloor + \lfloor \log_{\min P} U \rfloor}{|P|}.$$

For $\mathsf{E}_{\mathsf{sc1}}$, this event involves an invocation of the sum-check primitive on a degree-4 polynomial with s variables where the claimed sum is false but the final polynomial evaluation claims are all true. By Lemma 1 and $\min P \leq q$,

$$\Pr[\mathsf{E}_{\mathsf{sc1}}] \leq 4s/q \leq 4s/\min P.$$

For $\mathsf{E}_{\mathsf{agg}}$, this event treats three claims as a degree 2 polynomial and evaluates this polynomial at a random point γ. If at least one constituent claim is false, the resulting polynomial is non-zero, and thus, by the Factor Theorem (a univariate case of the Schwartz-Zippel lemma [111, 149]) and $\min P \leq q$,

$$\Pr[\mathsf{E}_{\mathsf{agg}}] \leq 2/q \leq 2/\min P.$$

For $\mathsf{E}_{\mathsf{sc2}}$, this event involves an invocation of the sum-check primitive on a degree-2 polynomial with k variables where the claimed sum is false but the final polynomial evaluation claims are all true. By Lemma 1 and $\min P \leq q$,

$$\Pr[\mathsf{E}_{\mathsf{sc2}}] \leq 2k/q \leq 2k/\min P.$$

By a union bound over all bad events, we get Back-end ϵ-soundness holds with error at most

$$\frac{\lfloor \log_{\min P} D_{\max} \rfloor + \lfloor \log_{\min P} U \rfloor}{|P|} + \frac{4s + 2 + 2k}{\min P} + \kappa_{\mathrm{PC}}.$$

□

B.4 Quantifying the guarantees of Spain's back-end

The implementation described in Sections 6 and 7 uses specific parameters. The following corollary characterizes the soundness and completeness guarantees of Spain's back-end when instantiated with DARK as the polynomial commitment scheme, and in a regime that covers the aforementioned parameters.

Corollary 1. *When Figure 8 is instantiated with the following parameters:*

- $m \leq 2^{32}$, $n \leq 2^{32}$;
- $D \leq 2^{96}$;
- *PC is the DARK multilinear polynomial commitment scheme [40] with* $N_{max} \leq 2^{96}$; *and*
- P *is the set of* 128*-bit primes,*

the resulting protocol has Back-end $(\epsilon/\sqrt{m})$*-completeness and Back-end* ϵ*-soundness, with soundness error* $\kappa \leq 2^{-40}$.

Proof. Back-end $(\epsilon/\sqrt{m})$*-completeness.* DARK satisfies sub-property 2 of Weak Integer Binding, so the completeness result of Theorem 1 applies directly to this corollary.

Back-end ϵ*-soundness.* For soundness, DARK only satisfies sub-property 1 of Weak Integer Binding *with high probability* (explicitly bounded further below), so we have to modify the soundness analysis of Theorem 1.

Suppose that there does not exist an ϵ-accurate assignment for $\mathbb{X}$. We bound the probability that $\mathcal{V}$ accepts. Define the event $\mathsf{E}_{\mathsf{weak}}$ to be the event that, over randomness internal to DARK's *Verify* function, $\mathcal{V}$ accepts an alleged opening of a commitment $com_{\tilde{f}}$ at $\hat{\beta}$ where $\tilde{f}$ does not satisfy the conditions detailed in sub-property 1 of Weak Integer Binding.

Conditioned on $\mathsf{E}_{\mathsf{weak}}$ not occurring, the verifier's acceptance probability is upper-bounded by the soundness error stated in Theorem 1. Thus, the overall probability that $\mathcal{V}$ accepts is at most

$$\frac{\lfloor \log_{\min P} D_{\max} \rfloor + \lfloor \log_{\min P} U \rfloor}{|P|} + \frac{4s + 2 + 2k}{\min P} + \kappa_{\mathrm{PC}} + \Pr[\mathsf{E}_{\mathsf{weak}}].$$

It remains to fill in the parameters and bound $\Pr[\mathsf{E}_{\mathsf{weak}}]$.

First, we quantify DARK-related parameters. We explicitly set $N_{\max} \leq 2^{96}$ and derive κ_{PC}, $D_{\max}$, and $\Pr[\mathsf{E}_{\text{weak}}]$ using a combination of the analysis in the corrected DARK paper [40] and a subsequent work [39]. In the corrected DARK paper [40], it is proved that κ_{PC} for the DARK polynomial commitment scheme is at most $(3\log|w|)/2^{\lambda}$, where λ is a parameter internal to DARK that introduces a tradeoff between κ_{PC} and $D_{\max}$. By choosing $\lambda = 50$, we get that $\kappa_{\text{PC}} \leq (3\log|w|)/2^{\lambda} = 3 \cdot 32/2^{50} < 2^{-43}$. From the analysis and scripts provided in the subsequent work [39], we get that for $D_{\max} \leq 2^{1422}$, $\Pr[\mathsf{E}_{\text{weak}}] \leq \kappa_{\text{PC}}$. Note that the analysis below is not very sensitive to the magnitude of $D_{\max}$ or $N_{\max}$; however, it is very sensitive to κ_{PC}, and λ is chosen with that in mind.

Next, we need to determine the size of P, the set of 128-bit primes. The number of 128-bit primes is exactly $\pi(2^{128}) - \pi(2^{127})$, where $\pi(x)$ is the prime counting function. By the following standard bound [109]:

$$\pi(2x) - \pi(x) \geq 3x/(5 \cdot \ln x) \quad \forall x \geq 21,$$

we have that

$$|P| \geq \frac{3 \cdot 2^{127}}{5 \cdot 127 \cdot \ln 2} > \frac{3 \cdot 2^{127}}{5 \cdot 128 \cdot \ln 2} = \left(\frac{3}{5 \cdot \ln 2}\right) \cdot 2^{120} > 2^{119}.$$

Additionally, $\min P > 2^{127}$ by definition.

The remainder of the proof proceeds by computing

$$\frac{\lfloor \log_{\min P} D_{\max} \rfloor + \lfloor \log_{\min P} U \rfloor}{|P|} \quad \text{and} \quad \frac{4s + 2 + 2k}{\min P}$$

using the parameters above. This calculation is tedious but mechanical. By definition:

$$\begin{aligned} U &\leq m \cdot n^2 \cdot N_{\max}^4 \cdot D_{\max}^4 \cdot (n \cdot N_{\max}^2 + D^2)^2 + D^8 \cdot D_{\max}^4 \\ &\leq 2^{32+2\cdot 32+4\cdot 96+4\cdot 1422} \cdot (2^{32+2\cdot 96} + 2^{2\cdot 96})^2 + 2^{8\cdot 96+4\cdot 1422} \\ &\leq 2^{32+64+384+5688} \cdot (2^{32+192} + 2^{192})^2 + 2^{768+5688} \\ &\leq 2^{6168} \cdot (2^{224} + 2^{192})^2 + 2^{6456} \\ &< 2^{6168} \cdot 2^{450} + 2^{6456} \\ &< 2^{6619}. \end{aligned}$$

It follows that

$$\frac{\lfloor \log_{\min P} D_{\max} \rfloor + \lfloor \log_{\min P} U \rfloor}{|P|} < \frac{\lfloor \frac{1422}{127} \rfloor + \lfloor \frac{6619}{127} \rfloor}{2^{119}} = \frac{63}{2^{119}}.$$

Additionally, we have the following bound on the second term:

$$\frac{4s + 2 + 2k}{\min P} < \frac{4 \cdot 32 + 2 + 2 \cdot 32}{2^{127}} = \frac{194}{2^{127}}.$$

By summing these two terms, $\kappa_{\text{PC}} < 2^{-43}$, and $\Pr[\mathsf{E}_{\text{weak}}] \leq \kappa_{\text{PC}}$, we get that the probability that the verifier accepts a non-ϵ-accurate assignment is less than 2^{-40}, as claimed. □

B.5 Deferred proofs

Here we prove Lemma 2 and Lemma 3. We start with two counting lemmas that they use.

Lemma 4. *Given an integer x of magnitude at most M and a set P of primes,*

$$\Pr_{q \sim P}[x \equiv 0 \pmod{q}] \leq \frac{\lfloor \log_{\min P} M \rfloor}{|P|}.$$

Proof. For x to be divisible by a prime q, q must be one of the prime factors of x. Suppose by contradiction that x had more than $\lfloor \log_{\min P} M \rfloor$ prime factors from P (counting multiplicity). Then x would have magnitude at least $(\min P)^{\lfloor \log_{\min P} M \rfloor + 1} > M$, a contradiction. Thus, x has at most $\lfloor \log_{\min P} M \rfloor$ prime factors from P, and the probability that a random prime from P divides x is at most $\lfloor \log_{\min P} M \rfloor / |P|$. □

Lemma 5. *Consider instance $\mathbb{X}$, multilinear polynomial $\tilde{f}$, and vector z, as defined in the protocol in Figure 8. Let D' be the unreduced denominator shared by the coefficients of $\tilde{f}$. Then, the denominator of $\|E_{\mathbb{X},z}\|_2^2$ is exactly $D^8 \cdot D'^4$. The numerator of $J - \|E_{\mathbb{X},z}\|_2^2$ is at most U, where U is as defined in Section B.3, and the denominator of this difference has the same denominator as $\|E_{\mathbb{X},z}\|_2^2$.*

Proof. The proof proceeds by bookkeeping: tracking exact denominators and an upper bound on the magnitude of intermediate numerators. The accounting pessimistically assumes that no reduction occurs when performing arithmetic operations. For example, when adding two rationals (N_1, D_1) and (N_2, D_2) where $\gcd(D_1, D_2) = 1$, the numerator of the result is at most $|N_1| \cdot D_2 + |N_2| \cdot D_1$ in magnitude and the denominator is exactly $D_1 \cdot D_2$.

By definition (§4.1),

$$\|E_{\mathbb{X},z}\|_2^2 = \sum_{r=1}^{m} \left(\sum_{c=1}^{n} A_{r,c} z_c \cdot \sum_{c=1}^{n} B_{r,c} z_c - \sum_{c=1}^{n} C_{r,c} z_c \right)^2.$$

Note that these summations are over m and n summands respectively as opposed to 2^s and 2^k summands as described in Section 4.2.2. This is because we are working with A, B, C as matrices, rather than their multilinear extensions.

Now, recall that the entries of A, B, C, *in*, and *out* all have unreduced denominator D; recall also that the numerators of the entries of A, B, C, *in*, and *out* have magnitude at most $N_{\max}$ (§B.2).

z is a concatenation of *in*, *out*, and w (the definition of w is in Figure 8). To bound the components of w, Weak Integer Binding's sub-property 1 (§B.1) implies that, for the coefficients of $\tilde{f}$, their unreduced denominator, D', satisfies $D' \leq D_{\max}$ and their numerators have magnitude at most $N_{\max}$. By definition of $\widetilde{w}$, its coefficients – that is, the components of w – share an unreduced denominator of $D \cdot D'$, and each has

an unreduced numerator of magnitude at most $N_{\max}$. From this, it follows that z, when written with a common unreduced denominator of $D \cdot D'$, has numerators of magnitude at most $N_{\max} \cdot D'$.

For any matrix M, $M_{r,c} z_c$ is a rational with an unreduced numerator of magnitude at most $N_{\max}^2 \cdot D'$ and an unreduced denominator exactly $D^2 \cdot D'$.

Thus, $\sum_{c=1}^{n} M_{r,c} z_c$ is a rational with a numerator of magnitude at most $n \cdot N_{\max}^2 \cdot D'$ and an unreduced denominator exactly $D^2 \cdot D'$.

It follows that $\sum_{c=1}^{n} A_{r,c} z_c \cdot \sum_{c=1}^{n} B_{r,c} z_c$ is a rational with a numerator of magnitude at most $n^2 \cdot N_{\max}^4 \cdot D'^2$ and an unreduced denominator of magnitude exactly $(D^2 \cdot D')^2 = D^4 \cdot D'^2$.

When subtracting $\sum_{c=1}^{n} C_{r,c} z_c$, a rational of the form $(N_1, D^2 \cdot D')$, from $\sum_{c=1}^{n} A_{r,c} z_c \cdot \sum_{c=1}^{n} B_{r,c} z_c$, a rational of the form $(N_2, D^4 \cdot D'^2)$, one needs to multiply the numerator and denominator of the first rational by $D^2 \cdot D'$ to get a common denominator (in the worst case). Thus, after performing the scaling and subtraction, the resulting rational has a numerator of magnitude at most

$$n^2 \cdot N_{\max}^4 \cdot D'^2 + n \cdot N_{\max}^2 \cdot D^2 \cdot D'^2$$

and an unreduced denominator of magnitude exactly $D^4 \cdot D'^2$.

Squaring this rational to get the contribution of row r to $\|E_{\mathbb{X},z}\|_2^2$ results in a rational with a numerator of magnitude at most

$$(n^2 \cdot N_{\max}^4 \cdot D'^2 + n \cdot N_{\max}^2 \cdot D^2 \cdot D'^2)^2$$

and an unreduced denominator of magnitude exactly $D^8 \cdot D'^4$.

Summing over m rows to get $\|E_{\mathbb{X},z}\|_2^2$ does not change the denominator so results in a rational with denominator $D^8 \cdot D'^4$ (as claimed) and a numerator of magnitude at most

$$m \cdot (n^2 \cdot N_{\max}^4 \cdot D'^2 + n \cdot N_{\max}^2 \cdot D^2 \cdot D'^2)^2$$

which simplifies to

$$m \cdot n^2 \cdot N_{\max}^4 \cdot D'^4 \cdot (n \cdot N_{\max}^2 + D^2)^2.$$

To bound the numerator and denominator of $J - \|E_{\mathbb{X},z}\|_2^2$, we first need to characterize the numerator and denominator of J. To do this, we consider $\mathcal{V}$'s checks in Step 3. To pass these checks, J must be a rational of the form (S, D^8) where S is a positive integer bounded above by D^8 (since $0 \le J \le \epsilon^2 \le 1$). To perform the subtraction $J - \|E_{\mathbb{X},z}\|_2^2$, one needs to multiply the numerator and denominator of J by D'^4 to get a common denominator (in the worst case). The resulting rational $J - \|E_{\mathbb{X},z}\|_2^2$ has a numerator of magnitude at most

$$m \cdot n^2 \cdot N_{\max}^4 \cdot D'^4 \cdot (n \cdot N_{\max}^2 + D^2)^2 + S \cdot D'^4$$

and an unreduced denominator of magnitude exactly $D^8 \cdot D'^4$, as claimed.

Given that $D' \le D_{\max}$ and $S \le D^8$, the numerator is upper-bounded by

$$m \cdot n^2 \cdot N_{\max}^4 \cdot D_{\max}^4 \cdot (n \cdot N_{\max}^2 + D^2)^2 + D^8 \cdot D_{\max}^4.$$

This is exactly U, as claimed. □

With the prime-divisibility bound (Lemma 4) and the numerator bound (Lemma 5) in hand, the two ρ_q facts quickly follow.

Proof of Lemma 2 (ρ_q ill-defined). ρ_q is undefined on $\|E_{\mathbb{X},z}\|_2^2$ exactly when q divides the denominator of $\|E_{\mathbb{X},z}\|_2^2$. By Lemma 5, the denominator is a product of powers of D and D' (where $D' \le D_{\max}$). By construction, q does not divide D; thus q divides the denominator of $\|E_{\mathbb{X},z}\|_2^2$ if and only if q divides D'. Applying Lemma 4 with $M = D_{\max}$ gives the bound. □

Proof of Lemma 3 (ρ_q collision). $\rho_q(J) = \rho_q(\|E_{X,z}\|_2^2)$ if and only if q divides the numerator of $J - \|E_{X,z}\|_2^2$, which, by Lemma 5, has magnitude at most U. Applying Lemma 4 with $M = U$ gives the desired bound. □

C DARK with verifier-known group order

This appendix details the technique mentioned in Section 4.2.3.

We consider an RSA group $\mathbb{G}$ with generator g and modulus $M = p_1 \cdot p_2$. The verifier knows p_1 and p_2, while the prover only knows g and M but not its factorization.

DARK requires, as a primitive, a way for the prover to convince the verifier that $g^z = g^x \cdot g^{c \cdot y}$ given g^x, g^y, and g^z and a constant c known to both parties. The problem is computing $(g^y)^c$; the verifier cannot efficiently compute an exponentiation this large, because it requires $\lceil \log c \rceil$ multiplications, which is potentially on the order of a million or a billion (the combined bit-length of all witness elements). DARK solves this with a proof of exponentiation protocol that is burdensome for the prover. In Spain, the verifier borrows an optimization from RSA signing algorithms when the signer knows the factorization of M.

The key observation is that knowledge of the factorization of M allows the verifier to compute $\lambda(M)$, where λ is Carmichael's function [1, 43]. Then the verifier can compute $c' = c \bmod \lambda(M)$, a value on the order of M rather than c itself, and compute $(g^y)^{c'}$ instead of $(g^y)^c$.

The correctness of this substitution is as follows. g^y is a group element and thus is relatively prime to M, so by the definition of Carmichael's function, we have $(g^y)^{\lambda(M)} \equiv 1 \pmod{M}$, and thus $(g^y)^{c'} \equiv (g^y)^c \pmod{M}$.

This substitution is efficient because $\log c' \approx \log M$, which is at worst a few thousand rather than a few billion.

At a high level, replacing the proof of exponentiation in DARK with an explicit exponentiation, independent of how the verifier computes, does not change the security of DARK.

Block et al. [29, Lemma 6.4] formally prove this in the context of a verifier that does not know the factorization of M and opts to compute $(g^y)^c$ directly. Their proof applies to our setting as well, as the verifier's computation of $(g^y)^{c'}$ is functionally equivalent to computing $(g^y)^c$ directly.

D Protocol extensions

Spain can be adapted and extended in various ways.

D.1 SIMD-R1CS

Per Section 4.3, Spain can be adapted to support common variations on R1CS, namely SIMD-R1CS and I-R1CS.

In SIMD-R1CS (§4.3), the prover has L instances, each with its own public input and witness, but sharing the same R1CS structure (A, B, C). That is, there are vectors $z_0, \ldots, z_{L-1}$, corresponding to instances $\mathbb{X}_0, \ldots, \mathbb{X}_{L-1}$, where all instances are the same R1CS structure. To support SIMD-R1CS, define $z(c, j)$ similarly to $z(c)$. Here j selects the particular instance and c selects the variable within that instance (similar to how c functioned earlier). Given the multilinear extension $\widetilde{z}(t_1, t_2)$, define

$$\widetilde{h_M}(t_1, t_2) := \sum_{c \in \{0,1\}^k} \widetilde{M}(t_1, c) \cdot \widetilde{z}(c, t_2),$$

Then there are two options for proving correctness across all L instances.

The first option is for the prover to make a single claim J about the sum of squared errors across all L instances. Then, the verifier can confirm that $J \leq \epsilon^2$ and the two parties can apply the sum-check protocol to prove that

$$\rho_q(J) = \sum_{j \in \{0,1\}^\ell} \sum_{r \in \{0,1\}^s} \left(\widetilde{h_A}(r, j) \cdot \widetilde{h_B}(r, j) - \widetilde{h_C}(r, j) \right)^2$$

where $\ell = \lceil \log_2 L \rceil$. This provides Back-end ϵ-soundness and Back-end $(\epsilon / \sqrt{L \cdot m})$-completeness. Notice that completeness degrades with L here.

The second option is to have the prover make $L - 1$ separate claims, where for $j = \{0, \ldots, L-1\}$, each claim J_j is about $\|E_{\mathbb{X}_j, z_j}\|_2^2$ for the respective instance $\mathbb{X}_j$. The verifier then checks that each J_j is at most ϵ^2. Then, the prover and verifier interpret $\rho_q(J) = [\rho_q(J_0), \ldots, \rho_q(J_{L-1})]$ as a function (from index bits to value) to get its multilinear extension: $\widetilde{J}(t)$. Finally, the verifier chooses a random $\tau \in \mathbb{F}_q^\ell$ and uses polynomial identity testing (similar to Spartan [116]) to prove that $\widetilde{J}(\tau)$ is equal to

$$\sum_{j \in \{0,1\}^\ell} \widetilde{Eq}(\tau, j) \sum_{r \in \{0,1\}^s} \left(\widetilde{h_A}(r, j) \cdot \widetilde{h_B}(r, j) - \widetilde{h_C}(r, j) \right)^2.$$

This provides Back-end ϵ-soundness and Back-end $(\epsilon / \sqrt{m})$-completeness. Compared to the first option, completeness does not degrade with L here; however, the expression is degree 5 in j rather than degree 4, which mildly increases prover and verifier time.

D.2 I-R1CS

For I-R1CS (§4.3), the modification to Spain is simply: at the end of the protocol, rather than opening a single commitment, the prover opens all commitments $\widetilde{w}_1, \ldots, \widetilde{w}_R$.

Recall that Spain introduces an additional optimization: Spain allows for the prover and verifier to collaboratively introduce not only new values for the *assignment* but also new *constraints on the witness* in each round. This allows the verifier to encode random challenges in constraints, or to partially specify constraints that are "filled-in" by the prover. These techniques reduce the cost of performing dot products between the witness and random vectors, which is the core operation in checkers for matrix multiplication [56, 74]

D.3 Different norms for error measurement

Spain can be generalized to have a smaller gap between soundness and completeness versus the one presented in the paper, namely $\epsilon_C = \epsilon / \sqrt{m}$. Doing so lowers the precision required of the prover's witness generation (see §6.3 for the interplay between "precision of witness generation" and ϵ_C) but makes the underlying back-end protocol more costly.

Specifically, by using the ℓ^{2t} norm rather than the ℓ^2 norm of $E_{\mathbb{X},z}$, for integer $t > 1$, Spain will continue to have Back-end ϵ-soundness but now it will have Back-end $(\epsilon / m^{1/(2t)})$-completeness. That is because the following chain of inequalities holds:

$$\|E_{\mathbb{X},z}\|_\infty^{2t} \leq \|E_{\mathbb{X},z}\|_{2t}^{2t} \leq m \cdot \|E_{\mathbb{X},z}\|_\infty^{2t}$$

for any integer $t > 1$. To adapt Spain to use the ℓ^{2t} norm, the prover and verifier start with the following equation in place of Equation (4) in Section 4.2.2 (Step 5 in Figure 8):

$$\rho_q(J) = \sum_{r \in \{0,1\}^s} \left(\widetilde{g_A}(r) \cdot \widetilde{g_B}(r) - \widetilde{g_C}(r) \right)^{2t}.$$

This is a degree-$(4t)$ polynomial in r, so as t increases, so does prover time, verifier time, and communication.

E Translation fidelity

This appendix defines translation fidelity and covers some technical details of this concept. The bulk of the appendix analyzes the translation fidelity of the arithmetizations presented in Section 5.

E.1 Preliminaries

Definitions. Given an R1CS structure $\mathbb{S} = (A, B, C)$, recall that all entries in A, B, C are presumed to have a given denominator D, and likewise with the assignment z (§4.2.1, Appx. B). We call such numbers *η-multiples*, where $\eta = 1/D$.

Let $F_\delta(x)$ denote the possible outputs of function F on input x when its individual operations have error bounded by δ.

Recall the distinction between δ and δ_{wg} (§3). δ is part of the underlying protocol's soundness guarantee: if any operation in an execution has error greater than δ, the verifier is supposed to reject with high probability. On the other hand, δ_{wg} connects to the protocol's *completeness* guarantee: we want to arrange the design and implementation so that if the prover limits the per-operation error to δ_{wg}, then it *can* satisfy all constraints with suitable error, and thereby cause the verifier to accept.

The reason for the asymmetry is technical. It relates to the definitions of soundness and completeness in the front-end (given in the next paragraph), and to the fact that ϵ and ϵ_C are different, which itself stems from the fact that $\|E_{\mathbb{X},z}\|_\infty \leq \epsilon$ does not imply $\|E_{\mathbb{X},z}\|_2^2 \leq \epsilon^2$ (§4.1).

Now, consider a purported translation of F to an R1CS structure $\mathbb{S} = (A, B, C)$. Translation fidelity is two properties:

- The translation is *ϵ-tf-sound* if: for all ϵ-accurate assignments $(in, out, 1, w)$ to $\mathbb{S}$, $out \in F_\delta(in)$.
- The translation is *ϵ_C-tf-complete* if: for all (in, out) such that $out \in F_{\delta_{wg}}(in)$ and (in, out) contains only η-multiples, there exists an accompanying w such that $(in, out, 1, w)$ is an ϵ_C-accurate assignment to $\mathbb{S}$.

It is helpful to keep the following ordering in mind:

$\delta \geq \epsilon$ (will be established in the examples below)
$> \epsilon_C$ (§4.1)
$\geq \delta_{wg}$ (will be established in the examples below)

Per-operation vs per-function translation fidelity. The definitions of tf-soundness and tf-completeness cover translation of an entire *function*. However, the arithmetizations that we present in the next section will be at the level of *individual operations*. For these lower-level translations to be relevant, Spain needs a way to combine the translations of individual operations. The needed concept is *composition* (of operations or functions), and specifically how translations combine under composition. The description below delves into detail; the high-level point is that focusing on the translations of individual operations is justified.

Given two functions F and G, with corresponding R1CS instances $\mathbb{X}_F = (A_F, B_F, C_F, in_F, out_F)$ and $\mathbb{X}_G = (A_G, B_G, C_G, in_G, out_G)$, the corresponding instance for $G \circ F$, $\mathbb{X}_{G\circ F}$, is (A, B, C, in, out) where $in = in_F$, $out = out_G$, and A, B, C contain the constraints of A_F, B_F, C_F and A_G, B_G, C_G, relabeled so that the output variables of $\mathbb{X}_F$ and the input variables of $\mathbb{X}_G$ refer to the same set of variables, which we call *med*; the variables in *med* are not part of either *in* or *out*. The lemma below states that translation fidelity is preserved by the composition of functions.

Lemma 6. *If F and $\mathbb{X}_F$ satisfy ϵ-tf-soundness and ϵ_C-tf-completeness, and if G and $\mathbb{X}_G$ satisfy ϵ'-tf-soundness and ϵ'_C-tf-completeness, then $G \circ F$ and $\mathbb{X}_{G\circ F}$ satisfy* $\min\{\epsilon, \epsilon'\}$*-tf-soundness and* $\max\{\epsilon_C, \epsilon'_C\}$*-tf-completeness.*

Proof. We first establish $\min\{\epsilon, \epsilon'\}$-tf-soundness, where $\epsilon^* := \min\{\epsilon, \epsilon'\}$. The constraints of $\mathbb{X}_{G\circ F}$ are those of $\mathbb{X}_F$ together with those of $\mathbb{X}_G$, sharing only the variables *med*. An ϵ^*-accurate assignment to the composite therefore restricts to an ϵ^*-accurate assignment to each part; as $\epsilon^* \leq \epsilon$ and $\epsilon^* \leq \epsilon'$, these are in particular ϵ- and ϵ'-accurate. Then ϵ-tf-soundness of F gives $med \in F_\delta(in)$, and ϵ'-tf-soundness of G gives $out \in G_\delta(med)$, so $out \in (G \circ F)_\delta(in)$.

For tf-completeness, set $\epsilon^* := \max\{\epsilon_C, \epsilon'_C\}$. Consider (in, out) where *in* and *out* contain η-multiples and $out \in (G \circ F)_{\delta_{wg}}(in)$. By definition of the composite, there must be *med*, consisting of η-multiples, with $med \in F_{\delta_{wg}}(in)$ and $out \in G_{\delta_{wg}}(med)$. Applying ϵ_C-tf-completeness of F, there exists w_F, where $(in, med, 1, w_F)$ is an ϵ_C-accurate assignment to (A_F, B_F, C_F). Likewise, there exists w_G, where $(med, out, 1, w_G)$ is an ϵ'_C-accurate assignment to (A_G, B_G, C_G). Combining w_F, w_G, and *med* into w, the assignment $(in, out, 1, w)$ satisfies each constraint of $\mathbb{X}_{G\circ F}$ with error upper-bounded by ϵ_C or ϵ'_C, hence upper-bounded by ϵ^*. □

E.2 Analysis of arithmetizations

The ϵ and ϵ_C described for the translations below depend on the magnitudes of the inputs to these subfunctions. This is not problematic because Spain enforces a strict bound on the magnitude of rationals in the proof protocol (§B). By considering this maximum magnitude, one derives absolute bounds on ϵ and ϵ_C.

$\mathbf{z \leftarrow x/y}$. Division is a special case where the error bound is inversely proportional to the magnitude of y, and thus one must place restrictions on the domain of y or perform static analysis to ensure y is not too close to 0 when using this operation. Here we will assume that $|y| \geq M$ for some known $M > 0$.

First, we establish ϵ-tf-soundness. Suppose we have an ϵ-accurate assignment to $y \cdot z \approx_\epsilon x$. That means $|y \cdot z - x| \leq \epsilon$. Rearranging, we have that $|z - (x/y)| \leq \epsilon/|y|$. This implies that z is within $\epsilon/|y|$ of x/y, and thus $|z - (x/y)| \leq \epsilon/M$ (using the assumed lower bound on $|y|$). Letting $\delta = \epsilon/M$ completes the proof of ϵ-tf-soundness.

Second, for ϵ_C-tf-completeness, consider all (x, y, z), restricted to η-multiples, such that $z = (x/y) + \widehat{\delta_\pm}$ where $|\widehat{\delta_\pm}| \leq \delta_{wg}$. The constraint has error at most

$$|y \cdot z - x| = |y \cdot ((x/y) + \widehat{\delta_\pm}) - x| = |y \cdot \widehat{\delta_\pm}| \leq \delta_{wg}|y|.$$

This establishes ϵ_C-tf-completeness provided that $\epsilon_C \geq \delta_{wg}|y|$ for all inputs y.

$\mathbf{y} \leftarrow \sqrt{\mathbf{x}}$. First, we establish ϵ-tf-soundness. Suppose we have an ϵ-accurate assignment to $y \cdot y \approx_\epsilon x$. That means $|y^2 - x| \leq \epsilon$. This can be rewritten as $|y - \sqrt{x}| \cdot |y + \sqrt{x}| \leq \epsilon$, where $\sqrt{x}$ is a true square root of x, and assume WLOG that $\sqrt{x}$ is the positive root. This implies that either $|y - \sqrt{x}|$ or $|y + \sqrt{x}|$ is less than or equal to $\sqrt{\epsilon}$. This, in turn, implies that y is within $\sqrt{\epsilon}$ of a true square root of x. Letting $\delta = \sqrt{\epsilon}$ finishes the proof for ϵ-tf-soundness.

Second, for ϵ_C-tf-completeness, consider all (x, y), restricted to η-multiples, such that $y = \sqrt{x} + \widehat{\delta_\pm}$ where $\widehat{\delta_\pm}$ is a real number with $|\widehat{\delta_\pm}| \leq \delta_{wg}$. The constraint $y \cdot y \approx_\epsilon x$ is satisfied with error at most

$$\left|(\sqrt{x} + \widehat{\delta_\pm})^2 - x\right| \leq 2\delta_{wg}|\sqrt{x}| + \delta_{wg}^2.$$

This establishes ϵ_C-tf-completeness provided that $\epsilon_C \geq 2\delta_{wg}|\sqrt{x}| + \delta_{wg}^2$ for all inputs x.

assert $(\mathbf{x} \geq \mathbf{y})$. Recall that this is an assert of the form "greater-than-or-approximately equal" where x and y are allowed (but not required) to be "approximately-equal" when they are within δ of each other.

First, we establish ϵ-tf-soundness. Suppose we have an ϵ-accurate assignment to $t \cdot t \approx_\epsilon x - y$. That means $|t^2 - (x - y)| \leq \epsilon$. By definition, $t^2 \geq 0$. This implies that $x - y \geq -\epsilon$. Letting $\delta = \epsilon$ finishes the proof for ϵ-tf-soundness.

Second, for ϵ_C-tf-completeness, consider all x, y, restricted to η-multiples, such that $x \geq y$. Let t be the nearest η-multiple to $\sqrt{x - y}$. Thus, $t = \sqrt{x - y} + \widehat{\eta_\pm}$ where $|\widehat{\eta_\pm}| \leq \eta/2$. Then the constraint $t \cdot t \approx_\epsilon x - y$ is satisfied with error at most

$$|t^2 - (x - y)| = |2\widehat{\eta_\pm}\sqrt{x - y} + \widehat{\eta_\pm}^2| \leq \eta\sqrt{x - y} + \eta^2/4.$$

This establishes ϵ_C-tf-completeness provided that $\epsilon_C \geq \eta\sqrt{x - y} + \eta^2/4$.

$\mathbf{b} \leftarrow (\mathbf{x} \geq \mathbf{y})$. First, we establish ϵ-tf-soundness for $\epsilon < 1/8$. Suppose we have an ϵ-accurate assignment to the constraints: $b \cdot (1 - b) \approx_\epsilon 0$, $(2b - 1) \cdot (x - y) \approx_\epsilon t$, and $s \cdot s \approx_\epsilon t$. Using the same reasoning as in proving ϵ-tf-soundness for the assert statement, the third constraint implies $t \geq -\epsilon$.

The first constraint implies $|b \cdot (1 - b)| \leq \epsilon$. By our bound on ϵ and the quadratic formula applied to $-\epsilon \leq b \cdot (1 - b) \leq \epsilon$, we have

$$b \in \left[\frac{1 - \sqrt{1 + 4\epsilon}}{2}, \frac{1 - \sqrt{1 - 4\epsilon}}{2}\right]$$

or

$$b \in \left[\frac{1 + \sqrt{1 - 4\epsilon}}{2}, \frac{1 + \sqrt{1 + 4\epsilon}}{2}\right].$$

Given that $\sqrt{1 - 4\epsilon} \geq 1 - 4\epsilon$ for $\epsilon < 1/4$, and $\sqrt{1 + 4\epsilon} \leq 1 + 4\epsilon$ for $\epsilon < 1/4$, this implies that $b \in [-2\epsilon, 2\epsilon] \cup [1 - 2\epsilon, 1 + 2\epsilon]$, namely that b is within 2ϵ of either 0 or 1.

The second constraint requires the most involved analysis. An ϵ-accurate assignment implies $|(2b - 1) \cdot (x - y) - t| \leq \epsilon$. Given that $t \geq -\epsilon$, this implies $(2b - 1) \cdot (x - y) \geq -2\epsilon$. Let $b' := 2b - 1$. From the bounds on b from before

$$b' \in [-1 - 4\epsilon, -1 + 4\epsilon] \cup [1 - 4\epsilon, 1 + 4\epsilon].$$

We also have $b' \cdot (x - y) \geq -2\epsilon$. We consider two cases, depending on which interval b' falls into.

Case 1: $b' \in [-1 - 4\epsilon, -1 + 4\epsilon]$. In this case, we have

$$x - y \leq \frac{2\epsilon}{1 - 4\epsilon}.$$

Given that $\epsilon \leq 1/8$, we have

$$\frac{2\epsilon}{1 - 4\epsilon} \leq \frac{2\epsilon}{1/2} = 4\epsilon.$$

Case 2: $b' \in [1 - 4\epsilon, 1 + 4\epsilon]$. In this case, we have

$$x - y \geq \frac{-2\epsilon}{1 - 4\epsilon}.$$

Given that $\epsilon \leq 1/8$, we have

$$\frac{-2\epsilon}{1 - 4\epsilon} \geq \frac{-2\epsilon}{1/2} = -4\epsilon.$$

Summarizing the two cases: $b \in [-2\epsilon, 2\epsilon]$ (which is case 1) implies $x - y \leq 4\epsilon$, while $b \in [1 - 2\epsilon, 1 + 2\epsilon]$ (which is case 2) implies $x - y \geq -4\epsilon$, or $y - x \leq 4\epsilon$.

Taking the contrapositive of both

$$x - y > 4\epsilon \implies b \in [1 - 2\epsilon, 1 + 2\epsilon]$$
$$y - x > 4\epsilon \implies b \in [-2\epsilon, 2\epsilon].$$

Finally, if $|x - y| \leq 4\epsilon$, no further conclusion can be drawn about b, except that it must remain in the intervals derived above. This means it must be within 2ϵ of either 0 or 1.

By taking $\delta = 4\epsilon$, the relationships among x, y, and b obey the intended semantics of the operation, and thus ϵ-tf-soundness holds.

Second, we establish ϵ_C-tf-completeness, provided that $\delta_{wg} \leq 1/2$. Consider all (x, y, b), restricted to η-multiples, such that $b = 1 + \widehat{\delta_\pm}$ if $x \geq y$ and $b = \widehat{\delta_\pm}$ otherwise, where $\widehat{\delta_\pm}$ is a real number with $|\widehat{\delta_\pm}| \leq \delta_{wg}$.

For the first constraint there are two cases to consider.

Case 1: suppose $x \geq y$ and thus $b = 1 + \widehat{\delta_\pm}$. The constraint is satisfied with error at most

$$|b \cdot (1 - b)| = |(1 + \widehat{\delta_\pm}) \cdot (-\widehat{\delta_\pm})| \leq \delta_{wg} + \delta_{wg}^2.$$

Case 2: suppose $x < y$ and thus $b = \widehat{\delta_\pm}$. The upper bound on error is the same:

$$|b \cdot (1 - b)| = |\widehat{\delta_\pm} \cdot (1 - \widehat{\delta_\pm})| \leq \delta_{wg} + \delta_{wg}^2.$$

For the second constraint, take $t = (2b - 1) \cdot (x - y)$. The constraint is then satisfied with zero error.

For the third constraint, take s to be the nearest η-multiple to $\sqrt{t}$. Thus, $s = \sqrt{t} + \widehat{\eta_\pm}$ where $|\widehat{\eta_\pm}| \leq \eta/2$. Such a square root exists because $t \geq 0$. This follows because $(2b-1)$ and $(x-y)$ have the same sign, as long as $\delta_{wg} \leq 1/2$. Now, we show $t \leq 2 \cdot |x-y|$, as follows. If $x \geq y$, then $b = 1 + \widehat{\delta_\pm}$ and thus

$$t = (2b-1) \cdot (x-y) = (1 + 2\widehat{\delta_\pm}) \cdot |x-y| \leq 2 \cdot |x-y|.$$

If $x < y$, then $b = \widehat{\delta_\pm}$ and thus

$$t = (2b-1) \cdot (x-y) = (1 - 2\widehat{\delta_\pm}) \cdot |x-y| \leq 2 \cdot |x-y|.$$

Given this, the third constraint is satisfied with error at most

$$\begin{aligned} |s^2 - t| &= |(\sqrt{t} + \widehat{\eta_\pm})^2 - t| \\ &= |2\widehat{\eta_\pm}\sqrt{t} + \widehat{\eta_\pm}^2| \\ &\leq \eta\sqrt{t} + \eta^2/4 \\ &\leq \eta\sqrt{2 \cdot |x-y|} + \eta^2/4. \end{aligned}$$

This establishes ϵ_C-tf-completeness provided that $\epsilon_C \geq \max\{\delta_{wg} + \delta_{wg}^2, \eta\sqrt{2 \cdot |x-y|} + \eta^2/4\}$.

$\mathbf{z} \leftarrow \max\{\mathbf{v_1}, \ldots, \mathbf{v_L}\}$. Recall the constraints from Section 5:

- for all $i \in \{1, \ldots, L\}$: $t_i \cdot t_i \approx_\epsilon z - v_i$
- for all $i \in \{1, \ldots, L\}$: $b_i \cdot (1 - b_i) \approx_\epsilon 0$
- for all $i \in \{1, \ldots, L\}$: $b_i \cdot (z - v_i) \approx_\epsilon 0$
- $\sum_{i=1}^{L} b_i \approx_\epsilon 1$

First, we establish ϵ-tf-soundness, for all $\epsilon \leq 1/4$ when $\epsilon \cdot (2L+1) < 1$. Suppose we have an ϵ-accurate assignment to the constraints. For the first set of constraints, we can reuse the results from the assert operation to establish that $z \geq v_i - \epsilon$ for all i. From the second set of constraints, we can use the same reasoning as in $b \leftarrow (x \geq y)$ to establish that each b_i is within 2ϵ of either 0 or 1. Given that $\epsilon \cdot (2L+1) < 1$, the last constraint and the second set together imply that there exists some index j such that b_j is within 2ϵ of 1. This follows by contradiction, suppose this was not the case, then each b_i would be within 2ϵ of 0, and thus $\sum_{i=1}^{L} b_i \leq 2L \cdot \epsilon < 1 - \epsilon$, leaving the final constraint unsatisfied by more than ϵ.

This leaves the third set of constraints to analyze. Let j be the index for which $b_j \geq 1 - 2\epsilon$. Since the assignment is ϵ-accurate, $|b_j \cdot (z - v_j)| \leq \epsilon$. Thus:

$$|z - v_j| \leq \frac{\epsilon}{b_j} \leq \frac{\epsilon}{1-2\epsilon}.$$

In particular:

$$z \leq v_j + \frac{\epsilon}{1-2\epsilon} \leq \max_i v_i + \frac{\epsilon}{1-2\epsilon}.$$

We already have $z \geq v_i - \epsilon$ for all i, which implies

$$z \geq \max_i v_i - \epsilon,$$

and since $\epsilon < \frac{\epsilon}{1-2\epsilon}$ also

$$z \geq \max_i v_i - \frac{\epsilon}{1-2\epsilon}.$$

Thus, $|z - \max_i v_i| \leq \frac{\epsilon}{1-2\epsilon}$. Given that $\epsilon \leq 1/4$, $\frac{\epsilon}{1-2\epsilon} \leq \frac{\epsilon}{1/2} = 2\epsilon$. Setting $\delta = 2\epsilon$ completes the proof of ϵ-tf-soundness.

Second, we establish ϵ_C-tf-completeness. Consider all $(v_1, \ldots, v_L, z)$, restricted to η-multiples, such that $z = \max_i v_i + \widehat{\delta_\pm}$ for some $\widehat{\delta_\pm}$ where $|\widehat{\delta_\pm}| \leq \delta_{wg}$.

Take $b_j = 1$ for the index j corresponding to the maximum value, and set all other b_i to 0.

For the first set of constraints, we use that $z - v_i \geq -\delta_{wg}$. This follows from the fact that $z = \max_i v_i + \widehat{\delta_\pm}$ and thus $z - v_i = \max_i(v_i) + \widehat{\delta_\pm} - v_i \geq \widehat{\delta_\pm} \geq -\delta_{wg}$. Now there are two cases to consider.

Case 1: suppose $z - v_i \geq 0$. In this case, set t_i to be the nearest η-multiple to $\sqrt{z - v_i}$. Thus, $t_i = \sqrt{z - v_i} + \widehat{\eta_{\pm i}}$, where $|\widehat{\eta_{\pm i}}| \leq \eta/2$. These constraints are satisfied with error at most

$$\begin{aligned} |t_i^2 - (z - v_i)| &= |(\sqrt{z - v_i} + \widehat{\eta_{\pm i}})^2 - (z - v_i)| \\ &= |2\widehat{\eta_{\pm i}}\sqrt{z - v_i} + \widehat{\eta_{\pm i}}^2| \\ &\leq \eta\sqrt{\max_i(v_i) - \min_i(v_i) + \delta_{wg}} + \eta^2/4. \end{aligned}$$

Case 2: suppose $z - v_i < 0$. In this case, take $t_i = 0$. These constraints are satisfied with error at most

$$|t_i^2 - (z - v_i)| = |z - v_i| \leq \delta_{wg}.$$

The second and fourth set of constraints are satisfied with zero error, by the choice of b_i above.

The third set of constraints breaks into two cases. When $b_i = 0$, the ith constraint in the set is satisfied with zero error. When $b_i = 1$ (that is, when $i = j$), constraint j in the set is satisfied with error at most δ_{wg}, as $b_j \cdot (z - v_j) = z - v_j = \widehat{\delta_\pm} \leq \delta_{wg}$.

This establishes ϵ_C-tf-completeness provided that

$$\epsilon_C \geq \max\left\{\delta_{wg}, \eta\sqrt{\max_i(v_i) - \min_i(v_i) + \delta_{wg}} + \eta^2/4\right\}.$$

ReLU. First, we establish ϵ-tf-soundness for $\epsilon \leq 1$. Suppose we have an ϵ-accurate assignment to the constraints: $s \cdot s \approx_\epsilon z - x$, $t \cdot t \approx_\epsilon z$, and $(z - x) \cdot z \approx_\epsilon 0$ From the first and second constraints, we have that $z \geq -\epsilon + \max(0, x)$. The third constraint is where things diverge from prior analyses. An ϵ-accurate assignment implies $|(z - x) \cdot z| \leq \epsilon$. This can be decomposed as $|z - x| \cdot |z| \leq \epsilon$. For the product on the left to be at most ϵ, then at least one of the two factors must be at most $\sqrt{\epsilon}$. Thus z is within $\sqrt{\epsilon}$ of either 0 or x.

Now there are two cases to consider.

Case 1: suppose $x \geq 0$. Here z is at least $x - \epsilon$ by the first two constraints, and at most $x + \sqrt{\epsilon}$ by the third constraint. thus, as required, z is within $\sqrt{\epsilon}$ of x.

Case 2: suppose $x < 0$. Here z is at least $-\epsilon$ by the first two constraints, and at most $\sqrt{\epsilon}$ by the third constraint. Thus, as required, z is within $\sqrt{\epsilon}$ of 0.

By combining these cases, we get that z is within $\sqrt{\epsilon}$ of $\max(0, x)$. Setting $\delta = \sqrt{\epsilon}$ completes the proof.

Second, we establish ϵ_C-tf-completeness. Consider all (x, z), restricted to η-multiples, such that $z = \max(0, x) + \widehat{\delta_\pm}$ where $\widehat{\delta_\pm}$ is a real number with $|\widehat{\delta_\pm}| \leq \delta_{wg}$. We consider each constraint individually and take the maximum error across all constraints.

Constraint 1: $s \cdot s \approx_\epsilon z - x$. This constraint has error at most

$$|s^2 + x - (\max(0, x) + \widehat{\delta_\pm})| = |s^2 + \min(x, 0) - \widehat{\delta_\pm}|. = |s^2 + x'|,$$

where $x' := \min(x, 0) - \widehat{\delta_\pm}$. x' is an η-multiple $\leq \delta_{wg}$. There are two cases to consider.

Case 1: suppose $x' > 0$. In this case, setting $s = 0$ satisfies this constraint with error at most δ_{wg}.

Case 2: suppose $x' \leq 0$. In this case, set s to the closest η-multiple to $\sqrt{-x'}$. Thus, $s = \sqrt{-x'} + \widehat{\eta_\pm}$, for some $\widehat{\eta_\pm}$ where $|\widehat{\eta_\pm}| \leq \eta/2$. Then this constraint has error at most

$$\begin{aligned} |(\sqrt{-x'} + \widehat{\eta_\pm})^2 + x'| &= |2\widehat{\eta_\pm}\sqrt{-x'} + \widehat{\eta_\pm}^2| \\ &\leq \eta\sqrt{-x'} + \eta^2/4 \\ &\leq \eta\sqrt{|x| + \delta_{wg}} + \eta^2/4. \end{aligned}$$

Thus the first constraint is satisfied with error at most

$$\max\left\{\delta_{wg}, \eta\sqrt{|x| + \delta_{wg}} + \eta^2/4\right\}.$$

Constraint 2: $t \cdot t \approx_\epsilon z$. This constraint has error at most

$$|t^2 - (\max(0, x) + \widehat{\delta_\pm})|.$$

The analysis is similar to the one for the first constraint. The cases here are $z \leq 0$ (in which case let $t = 0$) and $z > 0$ (in which case let t be the closest η-multiple to $\sqrt{z}$). These settings satisfy this constraint with error at most

$$\max\left\{\delta_{wg}, \eta\sqrt{|x| + \delta_{wg}} + \eta^2/4\right\}.$$

Constraint 3: $(z - x) \cdot z \approx_\epsilon 0$. This constraint is satisfied with error at most

$$\left|(\max(0, x) + \widehat{\delta_\pm} - x) \cdot (\max(0, x) + \widehat{\delta_\pm})\right| \leq \delta_{wg}|x| + \delta_{wg}^2.$$

Combining the above, we have that all constraints are satisfied with error at most

$$\max\left\{\delta_{wg}, \eta\sqrt{|x| + \delta_{wg}} + \eta^2/4, \delta_{wg}|x| + \delta_{wg}^2\right\}.$$

This establishes ϵ_C-tf-completeness provided that

$$\epsilon_C \geq \max\left\{\delta_{wg}, \eta\sqrt{|x| + \delta_{wg}} + \eta^2/4, \delta_{wg}|x| + \delta_{wg}^2\right\}.$$

E.3 On choosing between alternative arithmetizations

When using traditional constraints, the fewer constraints in an arithmetization the better. However, with approximate constraints, this is not always the case. Take ReLU as an example. The ReLU arithmetization in Section E.2 is 3 constraints and satisfies ϵ-tf-soundness with $\delta = \sqrt{\epsilon}$. By contrast, if the following 5 constraints are used (derived from the max primitive):

- $s \cdot s \approx_\epsilon z - x$
- $t \cdot t \approx_\epsilon z$
- $b \cdot (1 - b) \approx_\epsilon 0$
- $b \cdot (z - x) \approx_\epsilon 0$
- $(1 - b) \cdot z \approx_\epsilon 0$

then ϵ-tf-soundness is satisfied with $\delta = 2\epsilon$.

Despite using almost twice as many constraints, the second arithmetization produces a significantly tighter relationship between ϵ and δ. This in turn enables a larger ϵ_C and δ_{wg} (which means lower precision is needed in witness generation). In settings where high-precision witness generation is a bottleneck, the second arithmetization may be preferable, in spite of using a larger number of constraints.

E.4 Hybrid arithmetizations

Spain exclusively handles approximate constraints. Here we sketch a short exploration of a hybrid setting where there are two structures, $\mathbb{S}_\epsilon$ and $\mathbb{S}_{\text{exact}}$, that enforce approximate and traditional constraints respectively on the same assignment z.

By appending to the definition of ϵ-accuracy the requirement that the traditional constraints (those in $\mathbb{S}_{\text{exact}}$) are satisfied with zero error, the notion of translation fidelity can be extended to this hybrid setting. To demonstrate, we step through the analysis of a simple operation, $b \leftarrow_{\text{exact}} (x \geq y)$, where b is exactly 0 or 1 and the comparison is approximate. Consider a hybrid arithmetization of this primitive that has two traditional constraints: $b \cdot (1 - b) = 0$ and $(2b - 1) \cdot (x - y) = t$, and one approximate constraint: $s \cdot s \approx_\epsilon t$.

$\mathbf{b} \leftarrow_{\text{exact}} (\mathbf{x} \geq \mathbf{y})$. First, we establish ϵ-tf-soundness. Suppose we have an ϵ-accurate assignment to the constraints.

From the first constraint, we have that b is either 0 or 1.

From the second constraint, we have that

$$\begin{aligned} b = 0 &\implies t = y - x \\ b = 1 &\implies t = x - y. \end{aligned}$$

Using the same reasoning as in proving ϵ-tf-soundness for the assert statement, the third constraint (the approximate one) implies $t \geq -\epsilon$.

By combining the above, we have that

$$\begin{aligned} x - y > \epsilon &\implies b = 1 \\ y - x > \epsilon &\implies b = 0. \end{aligned}$$

In the case where $|x - y| \leq \epsilon$, b can be either 0 or 1, which is consistent with the semantics of the approximate comparison.

By taking $\delta = \epsilon$, we have that b is consistent with the semantics of the approximate comparison, and thus ϵ-tf-soundness is established.

Second, we establish ϵ_C-tf-completeness. Consider all (x, y, b) restricted to η-multiples such that $b = 1$ if $x \geq y$ and $b = 0$ otherwise. Take t to be exactly $(2b - 1) \cdot (x - y)$ and s to be the nearest η-multiple to $\sqrt{t}$. Thus $s = \sqrt{t} + \hat{\eta}$ for some $\hat{\eta}$ where $|\hat{\eta}| \leq \eta/2$.

The traditional constraints are satisfied with zero error by the choice of t and the definition of b.

Because b is either 0 or 1, $t = (2b - 1) \cdot (x - y) = |x - y|$. Given this, the third constraint is satisfied with error at most

$$\begin{aligned} |s^2 - t| &= |(\sqrt{t} + \widehat{\eta_\pm})^2 - t| \\ &= |2\widehat{\eta_\pm}\sqrt{t} + \widehat{\eta_\pm}^2| \\ &\leq \eta\sqrt{t} + \eta^2/4 \\ &\leq \eta\sqrt{|x - y|} + \eta^2/4. \end{aligned}$$

This establishes ϵ_C-tf-completeness provided that $\epsilon_C \geq \eta\sqrt{|x - y|} + \eta^2/4$.

Note that both the tf-soundness and tf-completeness results here are stronger than those presented in the prior analysis of $b \leftarrow (x \geq y)$ where b is approximately Boolean and no traditional constraints are used.

Also, when exclusively using approximate constraints, it is extremely complex to confine b to be strictly Boolean, while with traditional constraints, doing so is straightforward. This example indicates the potential benefits of hybrid arithmetization, the full exploration of which we leave to future work.

F End-to-end correctness

We derive end-to-end soundness and completeness for Spain in the general case and for the parameters in our implementation. We do so by combining back-end correctness (Theorem 1, §B.3) and translation fidelity (§E.1). The statements below rely on δ and δ_{wg}, which are discussed in Appendix E.1.

Corollary 2. *Let F be a computation and (A, B, C) be an R1CS structure with m constraints and n variables. If (A, B, C) is an ϵ-*tf-sound *and $(\epsilon/\sqrt{m})$-*tf-complete *translation of F, then the protocol in Figure 8 satisfies the following two guarantees:*

1. *If out $\notin F_\delta(in)$, then the verifier accepts with probability at most*

$$\frac{\lfloor \log_{\min P} D_{max} \rfloor + \lfloor \log_{\min P} U \rfloor}{|P|} + \frac{4s + 2 + 2k}{\min P} + \kappa_{PC}.$$

2. *If out $\in F_{\delta'}(in)$, then the prover can always make an honest verifier accept.*

This corollary follows by straightforward combination of Theorem 1 with the definitions of tf-soundness and tf-completeness in Appendix E.1.

Corollary 3. *Let F be a computation from the list of benchmarks in Section 7, and (A, B, C) be an R1CS structure with $m \leq 2^{32}$ constraints and $n \leq 2^{32}$ variables. If (A, B, C) is an ϵ-*tf-sound *and $(\epsilon/\sqrt{m})$-*tf-complete *translation of F, then the protocol in Figure 8 with the parameters in Corollary 1 satisfies the following two guarantees:*

1. *If out $\notin F_\delta(in)$, then the verifier accepts with probability at most 2^{-40}.*
2. *If out $\in F_{\delta'}(in)$, then the prover can always make an honest verifier accept.*

The proof of this corollary follows mechanically from Corollaries 1 and 2.

G Artifact Appendix

Abstract

The purpose of this artifact is to disseminate our implementation of Spain, let others to prove execution of numerical computations using Spain, and enable reproduction of the paper's experimental results.

Scope

The artifact comprises the implementation of Spain described in Section 6 as well as the experimental results and figures in Section 7.

Contents

The artifact includes implementations of Spain's front-ends and back-end (§6), implementations of the ZKLP-FE and Otti-FE baselines, scripts for reproducing experimental results, and ONNX files used for arithmetization.

The artifact is documented with a README that includes further detail and instructions for running numerical computations through Spain.

Hosting

The artifact is publicly available at `https://doi.org/10.5281/zenodo.20090527`.

Requirements

Docker is required to run this artifact. Memory requirements to reproduce results for benchmarks on the prover are as follows:

- For GPT-2, *seq*=2: at least 10 GB RAM
- For GPT-2, *seq*=32: at least 40 GB RAM
- For GPT-2, the largest *passes* with which we experiment, namely *passes*= 16: at least 270 GB RAM
- For ZKLP-FE: at least 92 GB RAM
- For all other benchmarks: experimental results can be reproduced with under 5 GB of RAM for the prover.

ARTIFACT EVALUATED
USENIX
AVAILABLE

RT: Regular Types for the Streaming Shell

Zekai Li*
Brown University

Lukas Lazarek*
Brown University

Evangelos Lamprou
Brown University

George Kapetanakis
Brown University

Konstantinos Mamouras
Rice University

Nikos Vasilakis
Brown University

Abstract

This paper presents an overlay type system, RT, for statically checking streaming shell programs or fragments *before* their execution. RT's *regular types* offer expressiveness appropriate for capturing a command's standard input and output streams, support computationally tractable and efficient type checking, and provide an interface encoded as regular expressions—*i.e.*, annotations and error messages familiar to developers versed in the Unix environment. RT's extensions around type polymorphism, finite-state transductions, environment concretization, and syntactic primitives offer additional expressiveness and improved precision. Applying RT to hundreds of programs from various sources including StackOverflow, GitHub, and prior literature indicates efficient type checking (0.02*s* on average), effectiveness at discovering serious bugs (91% accuracy), and key benefits from RT's extensions (up to 83% reduction in false negatives).

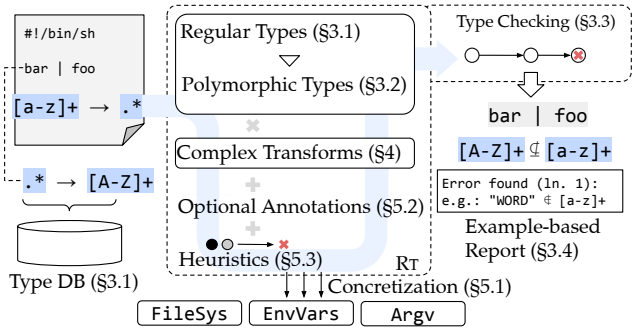


Fig. 1: RT overview. A shell program is parsed into RT, which assigns types to commands from a database of type declarations (§3.1). It models command behavior with simple regular types (§3.1), extends expressiveness through polymorphism (§3.2) and custom regular operations (§4), and detects composition errors with counterexamples (§3.4). Optional extensions—concretization (§5.1), annotations (§5.2), and heuristics (§5.3)—boost precision.

1 Introduction

The type systems of modern programming languages offer significant benefits [28, 37, 39, 54]: fast pre-execution checks for computations that may take days; detection of misbehaviors by a program's developer, not its user; improved error messages, including counter-examples; whole-program optimization opportunities, such as dead-code removal; and elimination of entire classes of bugs.

Unfortunately, these benefits are absent from the Unix shell—a pervasive environment used for a variety of tasks, ranging from system administration to data processing [24, 33, 36, 50, 55, 64]. Shell programs often combine commands developed in a variety of languages, interact in an untyped bytestream fashion, and are composed using the shell's intricate composition primitives. Coupled with the myriad of command flags and options, the implicit assumptions about the structure of certain file contents (*e.g.*, `/etc/passwd`), and the ease by which stream contents are manipulated, these characteristics routinely lead to errors that with current practices can only be discovered *after* the execution of shell programs. Such errors, arising when programming-in-the-large and in-the-small, can have devastating effects—at best crashing the execution of a long-running task and at worst silently, irreversibly corrupting entire filesystems [23, 51, 61, 62].

This paper proposes a new type system and associated implementation, RT, for statically checking shell programs before their execution (Fig. 1). RT's novel *regular types* offer expressiveness appropriate for capturing the structure of a command's input, output, and other streams, support computationally tractable and efficient type checking, and provide an interface encoded as regular expressions—*i.e.*, annotations and error messages familiar to developers versed in the Unix environment. With regular types, RT allows expressing, propagating, and checking constraints over the values of interprocess communication streams and the commands operating on them. RT targets shell fragments and scripts whose cor-

*Equal contribution.

```
1 find . |                          # Find files                    ε                    →   \.(/.+)?
2 grep -E 'book[0-9]+\.txt' |       # Filter by name               .*                   →   .*book[0-9]+\.txt.*
3 xargs cat |                       # Concat files      [^␣\t]+|".+"|'.+'                →   .*
4 tr -cs A-Za-z '\n' |              # Split words                   .*                   →   [A-Za-z]*
5 tr '[:lower:]' '[:upper:]' |      # Normalize                [A-Za-z]*                →   [A-Z]*
6 grep -fw dict.txt |               # Filter words                                       ...
7 sort | uniq -c | sort -rn         # Freq sort                ␣*[-+]?[0-9]+.*          →   ␣*[-+]?[0-9]+.*
```

Fig. 2: Left: A shell program with subtle composition mistakes. Right: Regular types for each line in the left script; the last corresponds to `sort -rn`. These are not the only types that could accurately summarize each invocation's behavior—in general, there may be infinitely many possible types for any given invocation, all of which may be correct (if not precise); see §3.

rectness depends on the line-level structure of streams passed between commands. This focus enables RT to catch bugs such as path-shape and argument-splitting mistakes, incompatible record formats between stages, text field mismatches, filters that eliminate output unintentionally, and malformed streams supplied to dangerous side-effectful commands.

These benefits of regular types come with (deliberately) constrained expressiveness: not all command behaviors are directly representable. With carefully designed extensions that maintain regularity, RT demonstrates that regular types are sufficiently expressive to capture many interesting properties of command behavior. It also reasons about many commands via useful approximations and regular subsets of their domains, including handling commands manipulating data such as CSV or JSONL. RT's regular-type reasoning allows it to identify bugs in computations that directly lead to dangerous side effects, such as pipelines computing path names to delete. By identifying bugs in such input-output manipulations, RT can prevent a wide array of bugs relating to dangerous side-effects without reasoning about those effects directly.

RT's type system (§3) and checking algorithm (§3.3) are extended with polymorphic types (§3.2) aimed at improving precision while maintaining tractability. Additional extensions such as compositional regular-type operators and finite-state transducers further improve expressiveness and accuracy (§4). The RT implementation is parameterized over a database of type declarations for every supported command invocation, akin to a typed standard-library in other languages. RT can optionally be configured to leverage developer annotations, additional heuristics, and environment concretization to improve its results (§5).

Evaluated on hundreds of shell programs, including dozens of real-world bugs collected from GitHub and StackOverflow, RT achieves 91% accuracy (864 out of 954) at classifying programs as correct or buggy. It uncovers 87 bugs ignored by state-of-the-art systems, including newly discovered bugs on production shell programs previously considered correct. RT's extensions offer valuable benefits in several cases—for example, finite-state transducers drop RT's false positive rate from 20% to 4%. And it is fast: RT typechecks 954 programs in 20s, averaging 0.02s per program.

The paper starts by exemplifying the kinds of composition problems often found at the streaming core of the shell and how RT exposes them ahead of time (§2). Sections 3–5 highlight RT's key contributions:

- *Regular types*: A type system targeting stream contents and commands operating on them, along with an efficient type checking algorithm that automatically detects composition errors in shell programs (§3).
- *Compositional regular-type operators*: An extended set of regular-type operations for common input-output transformations implemented efficiently via automata constructions (§4).
- *Extensions and optimizations*: Extensions and optimizations improving RT's precision—including a concretization subsystem for probing the current runtime context, a lightweight annotation language for expressing additional constraints, and heuristics for subtle composition errors (§5).

The paper continues with RT's evaluation (§6), discussion of related work (§7), and conclusion (§8). Appendix A contains examples of RT's optional syntax sugar, and Appendix B summarizes the open-source RT artifact:

https://github.com/atlas-brown/rt

2 Motivating Example

Consider the shell script in Fig. 2, a modern equivalent to McIlroy's classic one-liner [6]. It counts frequencies of correctly spelled words across all book files anywhere within the file tree rooted at the user's current directory. This example illustrates RT's target setting: a shell fragment whose behavior depends on whether each stage receives and emits lines in the shape expected by downstream.

Unfortunately, it does not work right: it produces an error saying that some files do not exist, and nothing on standard out. While ShellCheck—a state-of-the-practice linter [22]—reports no warnings, RT uncovers several issues causing the undesired behavior. Applied to `spell.sh`, its output begins:

```
$ rt spell.sh

  Error (ln. 2):
  > grep -E 'book[0-9]+\.txt' | xargs cat ...
        grep -E > (\.(/.+)?)&(.*book[0-9]+\.txt.*)
  maybe incompatible w/
   xargs cat > [^ \t]+|".+"|'.+'
  Counter-example: "./ book0.txt" ...
```

RT reports that the output of `grep` may be incompatible with `xargs cat`. To make the issue concrete for the developer, RT gives an incompatibility witness: `"./ book0.txt"` is a line that the upstream command may produce and cause the downstream command to misbehave because `xargs` splits the path at the space.

Regular types: RT reasons about composition mistakes by specifying the contents of communication channels with regular types. Regular *stream* types describe the language of the *lines* in the stream—*e.g.*, `find .` outputs paths relative to the current directory, one per line, so its output type is `\.(/.+)?`. Regular *command* types describe the languages of input-output streams—*e.g.*, `find .` :: $\varepsilon \rightarrow$ `\.(/.+)?`.

To reason about Fig. 2's pipeline, RT leverages the corresponding command types listed on the right of Fig. 2. By checking that the output language of every stage is included in the expected input of the following, RT reasons that the output language of `grep` (ln. 2) is not included in the expected input of `xargs cat` (ln. 3):

$$\texttt{.*book[0-9]+\.txt.*} \not\subseteq \texttt{[^␣\t]+|".+"|'.+'}$$

This is because filenames with spaces (*e.g.*, `my book1.txt`) are problematic inputs for `xargs cat`, as they are split into multiple arguments leading them to be misinterpreted as separate files (as in `cat my book1.txt`). The fix: `grep`'s pattern should be stricter to eliminate unexpected paths (`grep -E '^\./book[0-9]+\.txt$'`).

Configurability: Unfortunately, RT cannot reason precisely about the output stream type of `grep -fw dict.txt` without the contents of `dict.txt`. Absent that information, RT can only say that the patterns could be anything, therefore the invocation's output could be any subset of its input type (`[A-Z]*`)—not enough information to determine a problem.

However, RT can perform *environment-specific concretization* by reading the contents of `dict.txt` and computing a highly precise type for the command capturing every word in the file. The intersection of the input type of the `grep -fw dict.txt` invocation (`[A-Z]*`) with the disjunction of every word in the file simplifies to the empty type (*i.e.*, the patterns can never match any input lines), thereby identifying another error.

With that information from RT, the problem and corresponding fix are apparent. The arguments of line 5's `tr` are swapped: it should normalize words to lowercase, not upper. The fixed line 5 is: `tr '[:upper:]' '[:lower:]'`.

Extensions: Fig. 2 omits simple regular types for `sort` and `uniq` on purpose. They pose a problem: what are their appropriate types? Consider `sort`, which only reorders lines, and therefore by the definition of stream types, accepts any type and produces output of the same type. Hence, the most precise input type for `sort` would be whatever `grep -fw dict.txt` produces; similarly for its output.

The trouble with `sort` reveals a general concern, shared by `grep`, `uniq`, and others: the most precise type for a particular use of command depends on its context. To enable succinctly expressing types with this property, RT supports two extensions to regular types: *polymorphic command types* and *domain-specific type transformation operations*. With these extensions, RT's type database can provide equivalent or more precise versions of all of the types in Fig. 2 as well as (for the final program): `[a-z]*` $\rightarrow$ `[a-z]*` for `sort`, and `[a-z]*` $\rightarrow$ `␣*[0-9]+ [a-z]*` for `uniq -c`.

Results: RT identifies all composition mistakes in 0.02s. Once corrected, the program triggers no RT warnings.

3 Core Regular Types

RT summarizes stream contents using regular stream types, and command input/output behavior using regular command types. This section details the nature of these types, and how RT uses them to automatically identify composition mistakes.

A regular stream type is a regular language, written as a regular expression, that describes the set of all possible contents of each line in a stream, mirroring the line-oriented nature of most UNIX commands. The regular stream type is therefore line-oriented: it can precisely capture regular structures such as the formats of `/etc/passwd` entries and IPv6 addresses, and can approximate line-delimited formats such as CSV and text search results. For example, a regular stream type describing the output of `find .` is `\.(/.+)?`, capturing that every line of output is a relative path—*i.e.*, either `.` itself, or `./` followed by any non-empty string.

At its core, RT describes commands by combining two stream types: an input and an output stream type. Intuitively, a regular command type describes a transformation from a language of input to the language of output of the command. For example, the regular command type `.*` $\rightarrow$ `[0-9]+` describes how `wc -l` transforms the input language (anything) to the output language (decimal digits).

3.1 Regular Types and Commands

RT summarizes byte-stream contents using stream types—regular languages defined via extended regular expressions (EREs) [63] (*T* defined in Fig. 3). These include standard ERE features such as concatenation, alternation (`|`), repetition (`*`, `+`, `?`), grouping with parentheses, character classes (e.g., `[A-Z]`,

$$
\begin{array}{rcl}
T & ::= & \textit{literal} \mid \textit{character class} \mid \texttt{.} \mid \texttt{\^{}} \mid \texttt{\$} \\
 & \mid & TT \mid T\texttt{*} \mid T\texttt{+} \mid T\texttt{?} \mid \texttt{(}T\texttt{)} \mid T\texttt{|}T \\
 & \mid & T\texttt{\{}n\texttt{\}} \mid T\texttt{\{}n\texttt{,\}} \mid T\texttt{\{}n\texttt{,}m\texttt{\}} \\
 & \mid & \texttt{!}T \mid T\texttt{\&}T \\
n & \in & \mathbb{N} \\
 & & \\
C & ::= & T \to T
\end{array}
$$

Fig. 3: Base regular types accepted by RT. The grammar is a superset of POSIX ERE, with the addition of negation (`!`) and intersection (`&`). Regular stream types T are closed under regular languages. Command types C pair an input and output stream type.

`[:digit:]`), anchors (`^`, `$`), and bounded quantifiers (e.g., `a{2,5}`).

RT does not support lookaround assertions (see, e.g., [12,40,47]). It also does not support non-regular features such as backreferences (found in PCRE [3]) so that language inclusion remains decidable [20]. To improve expressiveness while retaining regularity, RT adds two operators: intersection (`&`) and negation (`!`). For instance, `[a-z]+ & err.*` matches all lowercase words with an "err" prefix, and `!(abc)` matches any string not matching `abc`. All supported operations are closed under regular languages.

RT defines a regular stream type as *sound* if it accepts all possible stream contents, and *precise* if it accepts only those that can actually occur. For example, if the possible lines in a stream are exactly `foo` and `bar`, then the regular stream type `foo` is unsound because it misses `bar`; `.*` is sound but not precise because it also accepts lines such as `baz`; and `foo|bar` is precise.

Command types: RT summarizes a command's input/output behavior using command types (C defined in Fig. 3), which pair stream types for each stream. A command type is *sound* if its input type is a subset of the command's safe domain—the set of inputs guaranteed not to cause crashes and misbehaviors—and its output type is a superset of the command's possible outputs. A command type is *precise* if for every input type satisfying the input constraints, the output type is exactly the language of possible outputs.

Although many UNIX commands do not correspond directly to regular-type transformations, most can be approximated with sufficient accuracy. For instance, `grep`'s output is the intersection of its input with a search-pattern. In contrast, commands such as `sort` exceed the expressiveness of regular types, but can still be usefully approximated—*e.g.*, by assigning identical input-output types that abstract over line contents, as the set of lines remains unchanged.

Commands such as `tee`, `paste`, or `comm` involve multiple streams; RT assigns each a separate type. RT also instantiates a type to the error streams of commands. For simplicity, this paper discusses commands in terms of a single input and single output stream unless otherwise noted.

$$
\begin{array}{rcl}
C & ::= & \ldots \mid \forall\, \alpha \subseteq T\ .\ \alpha \to T \\
T & ::= & \ldots \mid \alpha
\end{array}
$$

Fig. 4: Polymorphic regular type extensions. Polymorphic regular types have an input type constraint, and an input type variable α that may be used anywhere in the range type. This grammar extends the regular types of Fig. 3.

The type database: RT is parameterized over a database of command type declarations for supported invocations, much like how the type system of a typical programming language requires a manually-crafted base type environment declaring the types of primitive runtime functions. In general, RT's type database could be a literal mapping from invocations to command types, or arbitrary functions that produce a command type given an invocation.

RT strikes a middle-ground, with manually-written configuration files that define (1) how to parse relevant invocations and (2) how to construct a command type given an invocation (useful for example for `grep`, whose output type can be constructed from its argument regular expression). The higher-order command constructor `xargs` is not typed individually: an invocation such as `xargs cat` is handled as a single command. For commands not found in the database, RT defaults to the command type that is trivially correct for all command invocations: `.* → .*`. In total, RT's starting database covers 71/106 GNU coreutils (*Cf.*§6).

RT's type database is available for systems and users to inspect and modify the types of individual commands during development and use. It is integrated into the UNIX environment through the `rti` command. To inspect the type of a specific invocation, a user runs:

```
$ rti wc -l
wc -l :: .*  →  [0-9]+
```

Conversely, to add or update the regular type of a command in the type database, a user—or a system synthesizing regular types—runs:

```
$ rti word-stem :: '[A-Za-z]+' → '[A-Za-z]+'
```

3.2 Polymorphic Regular Types

A finite type database that maps each command invocation to a simple type struggles with commands whose most precise type depends on the input type. For example, `cat` copies its input stream, so its precise output type should be exactly the input type. Enumerating every possible `cat` type is impossible, while using a broad type such as `.* → .*` is sound but not precise. RT overcomes this challenge by introducing *polymorphic* command types. In particular, command types may be parameterized over *type variables*, which stand in for arbitrary regular types and may appear as the input type and anywhere within the output type. Fig. 4 defines the syntax

Algorithm 1: RT type checking algorithm. RT iterates over pipeline stages (commands), checking if the output of each stage is compatible with the following stage's input type. An example application is shown in Fig. 5. The algorithm generalizes naturally to directed acyclic command graphs by processing commands in topological order.

```
1  Function CheckPipeline(c_1|c_2|...|c_n, in):
2      input ← in;
3      for i ← 1 to n do
4          type ← GetTypeAnnotation(c_i);
5          if not CheckInput(type, input) then
6              ReportMismatch(type, input);
7          input ← InstantiateOutput(type, input);
8  Function CheckInput(type, input):
9      if type is simple then
10         return IsSubset(input, type.input);
11     else
           // Polymorphic type annotation
12         return IsSubset(input, type.constraint);
13 Function InstantiateOutput(type, input):
14     if type is simple then
15         return type.output;
16     else
17         return type.output[type.var ↦ input];
```

as an extension of simple command and stream types C and T; as a notational convenience, the input bound type can be omitted (equivalent to a bound of `.*`). The polymorphic type of `cat`, for example, can be written $\forall\alpha\,.\,\alpha \to \alpha$—capturing the fact that the output type of `cat` is the same as its input.

While some commands, such as `cat`, impose no expectations on their input, many commands with polymorphic types do: they are polymorphic over a constrained set of regular types (analogous to bounded polymorphism [11]). For instance, `sort -n` copies its input to its output (abstracting across all lines), but its numeric sorting expects each input line to begin with decimal digits. Polymorphic command types can express constraints over polymorphic variables, such as the following type for `sort -n`: $\forall\alpha \subseteq$ `[␣\t]*[-+]?[0-9]+.*` $\,.\,\alpha \to \alpha$.

Polymorphic command types do not introduce any challenges for type checking. RT reasons about polymorphic command types after instantiating them into simple types, so that type compatibility checks are always over simple stream types. For a fixed command invocation and input type, RT uses the unique most-precise instantiation. For example, when the input type is `[0-9]+`, RT instantiates the polymorphic type of `cat`, $\forall\alpha\,.\,\alpha \to \alpha$, by substituting `[0-9]+` for α, yielding the simple type `[0-9]+` $\to$ `[0-9]+`.

3.3 Type Checking with Regular Types

Alg. 1 outlines the core of RT's type checking algorithm. The algorithm is shown for pipelines, but generalizes naturally to directed acyclic command graphs by processing commands in topological order. At a high level, for pipelines, RT iterates over stages (commands), checking if the output of each stage is compatible with the following stage's input type.

In detail, the main `CheckPipeline` procedure accepts the pipeline of commands $c_1|c_2|...|c_n$ and an input stream type for the pipeline *in*. It iterates over each stage (command), keeping track of the intermediate stream types between each stage in the *input* local variable. The initial value of *input* is *in*, the assumed input to the pipeline; absent any specific information, *in* could be either the type that only accepts the empty string ε—assuming the pipeline receives no input—or `.*`—assuming it could receive any input. The choice between these two assumptions is easily configurable alongside other aspects of RT (*Cf.*§5).

For each stage of the pipeline, the algorithm performs four steps. First, it retrieves the command type for the stage's command invocation (c_i) with `GetTypeAnnotation`. Then, it checks if the current intermediate stream type is a valid input according to the input stream type of the command with `CheckInput`; if not, it calls `ReportMismatch` to report an error (see §3.4). Otherwise, the algorithm computes the next intermediate stream type based on the command type and its input using `InstantiateOutput`, and then proceeds with the rest of the pipeline.

The `CheckInput` and `InstantiateOutput` helpers check type compatibility and compute output types respectively. `CheckInput` consists of two cases: (1) if the command type is not polymorphic, then it checks if the input is a subset of the command type's input stream type, *i.e.*, regular language inclusion; (2) if the command type is polymorphic, then it checks if the input is a subset of the constraint bound on the command type's polymorphic input type. Similarly, `InstantiateOutput` mirrors the same structure with two cases: (1) if the command type is simple, then it returns the literal output stream type of the command; (2) if the command type is polymorphic, then it *instantiates* the output of the command type to obtain a simple stream type. Instantiating the polymorphic type means syntactically substituting the type variable with the actual input type. For example, for a polymorphic command type $\forall\alpha.\alpha \to$ `hi␣`α, and input type `[a-z]+`, the substitution in the output type results in the concrete type `hi␣[a-z]+`.

3.4 Error message generation

If the type checking algorithm detects type incompatibility between a stream and the input type of a command consuming it, it generates error messages to help users understand and fix the composition error. The error message includes

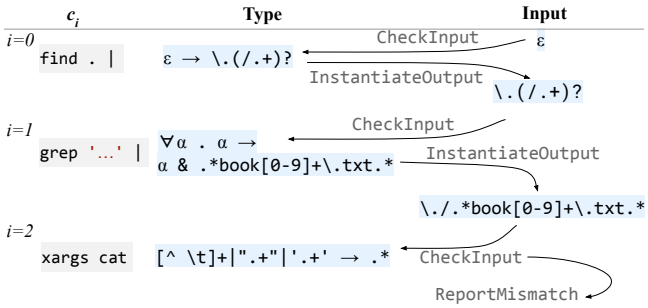


Fig. 5: Type-checking example. Applying Alg. 1 to Fig. 2's pipeline, step by step. Columns (left to right) are the command in the pipeline, the type produced by `GetTypeAnnotation`, and the intermediate stream type assigned to *input* after each iteration. Rows (top to bottom) show iterations 1–3 of the loop in `CheckPipeline`.

a counterexample witness x that demonstrates the type incompatibility, where x belongs to the input type but not to the expected type (for simple annotations) or the constraint type (for polymorphic annotations). Equivalently, RT looks for a string that L_1 accepts and L_2 rejects. By the set-theoretic equivalence $x \in L_1 \wedge x \notin L_2 \leftrightarrow x \in L_1 \backslash L_2$, RT finds such a witness from the automaton for $L_1 \backslash L_2$.

4 Complex Transformations

While simple and polymorphic types suffice to precisely model many commands—and approximate others—some approximated commands perform fine-grained manipulations that are (1) critical for reasoning about program correctness and (2) amenable to precise modeling. To capture such behaviors, RT leverages finite-state transducers to express complex regular-type transformations corresponding to common command semantics.

RT exposes these transformations as new primitive operations that extend regular types beyond standard operations such as union, intersection, and negation. These operations enable precise modeling of commands such as `tr`, `cut`, and `grep -o` whose behaviors are regular but not expressible using standard regular operations alone.

For instance, `grep '[A-Z]'` can be precisely typed using intersection and polymorphism, but `tr A-Z a-z` cannot—its transformation, while regular, lies outside the expressive power of standard operations. Naive approximations using only standard operations can be unsound: modeling `tr -s ' '` as $\forall\alpha.\alpha \to \alpha$ `&!(.*␣␣.*)` underapproximates its output. With input `(␣b␣)*`, this results in `(␣b␣)?`, whereas the actual output is `(␣(b␣)+)?`. Unsound command types can lead to false negatives, since downstream checks then reason about an output type that omits streams the command can actually produce. Imprecise command types can lead to false positives, since downstream checks must also account for streams that the command will never produce.

$$
\begin{array}{rcl}
T & ::= & \ldots \\
 & | & \texttt{reverse}(T) \\
 & | & \texttt{translate-match}(T, T, s, \texttt{global=}b) \\
 & | & \texttt{line-extract}(T, T) \\
 & | & \texttt{translate-chars}(T, C, C, \texttt{invert=}b, \texttt{squeeze=}b) \\
 & | & \texttt{field-select}(T, C, N, \texttt{invert=}b) \\
s & ::= & \textit{string} \mid \backslash n \mid s\,s \\
b & ::= & \texttt{true} \mid \texttt{false} \\
C & ::= & c \mid c-c \mid C,C \mid \textit{character class} \\
c & ::= & \textit{literal} \\
N & ::= & n \mid n-n \mid n- \mid N,N \\
n & \in & \mathbb{N}
\end{array}
$$

Fig. 6: The full RT types. The full syntax of regular stream types, extending regular expressions with five new primitive operations (extending Fig. 3 and Fig. 4).

4.1 New Regular-Type Operations

To improve precision while maintaining soundness, RT supports concisely expressing regular types for these kinds of transformations via a novel set of regular-type operations that, alongside standard operations, make up a domain-specific language of regular types. Fig. 6 shows the now-extended language of regular types.

The semantics of each operation is that of a type-level transformation of regular languages. The `reverse(`T_i`)` operator transforms language T_i into the reverse language, as recognized by the reversed DFA. The `translate-match(`T_i`, `T_p`, `s`, global=`b`)` operator transforms input type T_i into a type where all leftmost-longest[1] occurrences of T_p are replaced with s (if b is true, otherwise only the first)—and s may include group references (not to be confused with backreferences in patterns). The `line-extract(`T_i`, `T_p`)` operator transforms T_i into the sublanguage which also matches T_p. The `translate-chars(`T_i`, `C_f`, `C_t`, invert=`b_v`, squeeze=`b_s`)` operator transforms T_i into a type where all characters in C_f are replaced with the same-position character in C_t (where positions beyond the end of C_t translate to the last character of C_t, as in `tr`); if b_v is true, the set of characters to be replaced is the complement of C_f; if b_s is true, the result type also compresses repeated occurrences of each individual character in C_t to a single occurrence. The `field-select(`T_i`, `c`, `N`, invert=`b`)` operator transforms T_i into the type of just fields at indices N (or all other fields, if b is true), separated by c; fields that are absent come out empty in the transformed type.

The `translate-chars` and `field-select` operations are specializations of the more general `translate-match` construct; these specializations correspond to commonly-used commands and admit more precise and efficient modeling than the fully general `translate-match`.

[1] This corresponds to the POSIX disambiguation policy for regular expressions, which differs from the leftmost-greedy policy used by PCRE [41].

Tab. 1: Soundness and precision of operations. Each row describes a regular type operation, whether RT can compute the output type soundly and precisely, and an example command that is described with the operation. The last column describes the type of FST used to compute the output type, which can be a functional non-deterministic FST (FN-FST), a deterministic FST (DFST), or a non-deterministic FST (NFST).

Operation	Instance	*Sound?*	*Precise?*	Example (`Command :: Output Type`, for input type T)	FST Type
`translate-match`	String to string	✓	✓	`sed 's/old/new/' :: translate-match(`T`,old,new)`	FN-FST
	Anch. regex	✓	✓	`sed 's/^a.a*b/new/' :: translate-match(`T`,^a.a*b,new)`	FN-FST
	General case	✓	-	`sed 's/a.*/b/g' :: translate-match(`T`,a.*,b,true)`	NFST
`line-extract`	Cap. string	✓	✓	`grep -o 'pattern' :: line-extract(`T`,pattern)`	FN-FST
	Cap. anch. regex	✓	✓	`rg -o '^a.a*b' :: line-extract(`T`,^a.a*b)`	NFST
	Cap. regex	✓	-	`awk '/a.a*b/' :: line-extract(`T`,a.a*b)`	NFST
`field-select`	General case	✓	✓	`cut -d ' ' -f 2 :: field-select(`T`&(.*␣.*),␣,2)|`T`&[^␣]*`	DFST
`translate-chars`	General case	✓	✓	`tr a-z A-Z :: translate-chars(`T`,a-z,A-Z)`	DFST

To illustrate the operations concretely, Tab. 1 lists example command invocations and corresponding command types demonstrating each operation. For instance, `sed` uses `translate-match`, and `tr` uses `translate-chars`.

The behavior of `cut` has three cases: (1) if the input line lacks the delimiter entirely, it outputs the whole line, (2) if the input line includes the delimiter but fewer fields than those requested, it outputs the empty string for the missing fields, and (3) otherwise it outputs the requested fields separated by the delimiter and in ascending order of index. To capture this complex behavior, `cut`'s type performs field selection according to `field-select`'s stricter semantics for the sublanguage of inputs that contains the delimiter, and then unions with the sublanguage of inputs that do not contain the delimiter.

While some common commands map directly to one operation, others compose multiple. For example, the type of `awk '{print $2}'` in RT's database is:

```
∀α.α →
  field-select(
    translate-chars(
      translate-match(α,"^[␣\t]+", ""),
    "␣,\t", "␣", squeeze=true),
  "␣", 2)
```

4.2 Finite-State Transducers

RT implements the regular type operations shown in Fig. 6 by constructing finite-state transducers (FSTs) and composing them with the input automata representing regular types. An FST is a finite-state machine that not only recognizes input strings but also emits output strings along transitions [48]. This allows RT to define type-level string transformations that preserve regularity: the output of an FST applied to a regular language is itself regular.

Each new regular type operation that RT supports corresponds to its own FST construction. Once constructed, the FST is composed with the input automaton (corresponding to the input stream regular type) to produce a new automaton representing the transformed language (the output regular type). These constructions do not complicate type checking: they essentially contribute one new step to `InstantiateOutput` in Alg. 1, which computes the result of any operations; hence, type checks remain in terms of simple regular languages.

To more concretely understand how RT constructs the FST for a given regular type operation, consider the `translate-chars` operation with `squeeze=true`, which transforms an input type `A` by collapsing consecutive occurrences of characters in the translated-to set into a single instance. At a high level, RT computes the `translate-chars` operation over regular types by constructing the operation's FST for the particular arguments, computing its product with the input language automata, and constructing a resulting NFA. Fig. 7 illustrates this process in detail for the input type `(␣b␣)*`, with space as the character being squeezed. Starting from the DFA of the input (a) and the FST representing the `translate-chars` operation (b), RT computes their product (c) and derives an NFA over the FST's outputs (d), yielding the transformed type.

Different operations require different classes of FSTs depending on their complexity and the structure of their pattern-matching logic. Tab. 1 summarizes the operations supported by RT, including whether their output types are computed soundly and precisely, and the class of FST used. Deterministic FSTs (DFSTs) are transducers with exactly one possible transition per input symbol in each state. They are used for transformations that are positionally local and unambiguous. Functional non-deterministic FSTs (FN-FSTs) allow multiple accepting paths for a given input, but all such paths produce the same output. This enables more expressive transformations while preserving precision. General non-deterministic FSTs (NFSTs) may produce multiple outputs for the same input and are used for the most general forms of pattern-based transformations. These transducers remain sound but may sacrifice precision—the result of computing the corresponding operation may be a larger language than necessary.

Tab. 2: RT heuristics. Each row describes a heuristic that RT uses to identify likely mistakes in command compositions, its description, and an example command composition that triggers the heuristic.

Name	Description	Example
Empty output	Command has empty output type	`cat data.txt | tr a b | grep pattern`
Ignored input	Command that expects no input	`cat data.txt | grep -rHn "word1" $list`
Useless composition	Stage does nothing	`echo "http://www.google.com" | cut -f3`
Lexicographic number sort	Sort numbers with non-numeric `sort`	`cat data.txt | sort | uniq -c | sort -r`

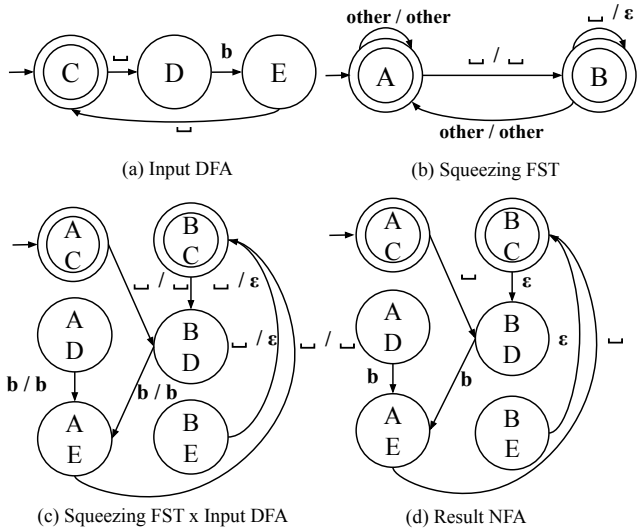


Fig. 7: FST construction example. RT constructs the FST `translate-chars((␣b␣)*, ␣, ␣, squeeze=true)` by constructing the DFA for the regular type `(␣b␣)*` (a) and then taking its product with the `translate-chars` FST (b). Taking the output of each transition of the resulting FST (c) produces the output NFA corresponding to the transformation of the input type (d).

Precision of transformation: Not all operations can be implemented perfectly precisely. Some instantiations of the operations are not regular transformations: consider the invocation `sed -E "s/(.*)/\1\1/"`, corresponding to `translate-match(A, (.*), \1\1)`, which duplicates input strings. The image of input type `.*` under this duplication operation is not a regular language [31]. On the other hand, other instantiations do perform regular transformations, but the transformations do not admit precise modeling. RT supports sound approximations in all such cases, even while supporting precise modeling for simple common cases; Tab. 1 breaks down which cases RT can and cannot support precisely.

RT precisely computes the output type for operations `field-select` and `translate-chars`. They are implemented using deterministic FSTs, which contain no ambiguous transitions. However, `line-extract` and `translate-match` operations are challenging to implement because determining exact match positions often requires nondeterministic guesses. Specific cases of `translate-match`—string matching and anchored patterns—are addressed with special transitions in the FST that recover from failed partial matches, which results in functional non-deterministic FSTs. Similarly, for the precise cases of `line-extract`, such as capturing an anchored regex pattern, RT relies on non-deterministic FSTs. In the general cases of `line-extract` and `translate-match`, however, RT constructs non-deterministic FSTs that over-approximate the actual output type. These non-deterministic structures ensure soundness by covering all possible outputs but sacrifice precision. For instance, RT computes `translate-match("abab", "a.*b", c, global=true)` as `c?c`, an over-approximation of the precise output `c`. If a downstream command expects strictly `c`, this can lead to false positives.

5 Configurability and Concretization

RT optionally performs environment-specific concretization, inferring input types based on the contents of input files at the time of analysis, to provide more precise information (§5.1). Two types of annotations provide a way for developers to describe the specific shape of unknown inputs, or assert expectations about the type of desired outputs (§5.2).

Also configurably, RT can warn about composition mistakes that do not strictly violate any command's input constraints, but rather behave in ways that are almost always undesirable. For instance, a common bug is a pipeline that never produces output due to incorrect command composition. Using regular type information, RT can warn developers about such mistakes ahead-of-time based on a set of heuristics about typical input and output types (§5.3).

5.1 Environment-based Concretization

RT can be configured to leverage information from the environment during analysis to obtain information about actual inputs to reason precisely about program behavior within the local system context. For instance, by analyzing the content of an input file, RT can infer a regular type capturing the current contents, and then use that type to be able to notify developers that a program will certainly be problematic if executed using that particular file. This can help developers stay ahead of typical mistakes that slow down development, such as using `cut` to select fields but forgetting to specify the field delimiter.

RT leverages concretization as part of a secondary specialization pass when type checking programs. After performing normal type checking (*i.e.*, making no assumptions refining the range of possible values) and reporting any general issues with a program, RT then attempts concretization to offer finer-grained information. Specifically, if there are concrete inputs that are accessible at analysis-time on the local machine, including files or environment variables, RT reads the input and uses the entire contents verbatim as the input type in the same way as assumption annotations.

5.2 Type Annotations

Type annotations let developers configure how RT checks command compositions, enabling more precise validation. While some compositions may seem unsafe for arbitrary inputs, they can be valid for specific input formats. For example, using `grep` to filter out `NA` lines before passing data to a command expecting numbers is safe when the input contains only numbers or `NA`. By annotating such expectations, developers help RT suppress irrelevant warnings and provide more accurate feedback.

Annotations also help developers use RT to catch bugs beyond unsafe compositions. For instance, a program's output may need to meet specific correctness properties—*e.g.*, not containing newlines before being sent over the network. With output annotations, developers can encode such expectations, allowing RT to verify them.

Developers can provide annotations using comments in a script's source, such as the following:

```
# @assume "cat $1" --> "[0-9]\t[A-Za-z]"
cat $1 | tr A-Z a-z | cut -f 2 | ...
```

In general, annotations appear as comments that begin with `@` and a keyword identifying the kind of annotation. Corresponding to the different kinds of information developers can offer, there are five distinct kinds of annotations. The first two are assumptions that RT should use while reasoning instead of computing the corresponding information: `assume` assumes the type of an invocation's output, and `input` assumes the type of input to the entire program. The remaining three are assertions that RT checks in addition to confirming input-output type compatibility: `assert` checks that the computed output type of an invocation is a subset of the annotated type, `expect` checks that the input received by an invocation is a subset of the annotated type, and `output` checks that the program's output is a subset of the annotated type.

To help developers write annotations when necessary, RT supports human-friendly definitions for a variety of types that desugar behind the scenes. These definitions capture common patterns such as ip-addresses, numbers in a floating-point decimal format, urls, and more (appendix A lists the full set of definitions RT supports, which is also extendable).

RT incorporates annotations into the type-checking process at different points for each kind of annotation. For assumption annotations, RT simply replaces the corresponding type with the assumed one at the relevant stage of the pipeline. For assertions, RT first computes the actual regular type at the relevant stage, and then checks if that actual type is included in the asserted type—if so, it proceeds with the computed type, and otherwise raises an error reporting the failure.

5.3 Heuristics

RT includes several heuristics identifying compositions that do not strictly violate any command's input constraints, and hence are well-typed, but correspond to likely mistakes. That is, the circumstances identified by the heuristics are not necessarily problematic, but they capture common mistakes and undesirable behavior.

Tab. 2 lists the four heuristics RT considers, each of which describes a criterion for rejecting a program. The first heuristic is command invocations in pipelines that have an empty output type—therefore are guaranteed to have no output. The second is providing input to commands that expect none, such as `echo`. The third is pipeline stages that are guaranteed to have no effect on the pipeline contents. This heuristic applies only to filter-like commands such as `grep`, selector-like commands such as `cut`, and transformer-like commands such as `tr`. The fourth is using the `sort` command in lexicographic mode (without `-n`) on numeric inputs. Critically, the heuristics describe properties of compositions that depend on input and output type information. Hence, the heuristics capture common mistake patterns at a more semantic level than syntactic matching, which is not robust to syntactic variation.

6 Evaluation

We evaluate RT on hundreds of real-world programs, investigating the following questions: **Q1**) What is the effectiveness of RT? and **Q2**) What is the analysis performance of RT?

Overview of Results: RT's overall accuracy across all programs is 91%. Fig. 8 breaks this down into accuracy on buggy programs (72%) and correct ones (96%)—and when programs include extra type annotations to inform RT, its accuracy rises to 94% and 98% respectively. In comparison to state-of-the-art systems, RT is 52% more accurate than LADDERTYPES and ShellCheck on buggy programs. Furthermore, RT provides confidence information indicating whether approximations may lead it to warn about a program, and for programs with definite confidence RT exhibits 100% accuracy (and 86% accuracy for possible-bug reports). RT analyzes every program in the set in under 1s (avg. 0.02s).

Tab. 3: Benchmark summary. Correct (#C) and buggy (#B) columns show the number of programs that are correct or include one or more composition mistakes, respectively.

Set	#C	#B	Description
GitHub	57	57	GitHub bugfix commits
StackOverflow	–	11	StackOverflow
LadderTypes	4	8	Example programs [60]
Koala	478	3	Koala benchmarks [38]
Intercode	188	17	NL synthesis tasks [66]
LLM	–	120	LLM-generated [52]
Handwritten	3	8	Microbenchmarks
Total	730	224	

6.1 Benchmark Suite

We use a comprehensive suite of both correct and buggy programs to answer these research questions. Since no established suite meeting both of these criteria already exists, we have collected and curated a new set of benchmarks.

Tab. 3 summarizes our benchmark suite, consisting of 7 different evaluation sets drawn from a variety of sources. The sets in rows three through five are drawn directly from the literature and community. The first is examples of both buggy and correct programs from Sippel and Schirmeier's pipeline typing system, LadderTypes [60] (*Cf.*§7); second is all scripts in the Koala benchmark suite [38]; and third is the full set of programs from the Intercode evaluation suite for natural-language-to-code-synthesis systems [66]. These three sets come with explicit and implicit labels; some of the LadderTypes programs are provided as buggy examples, and all of the other programs are intended to be correct. However, in the process of collecting these sets, we identified two other bugs in the LadderTypes programs, three in the Koala benchmarks, and 17 in the Intercode suite, and therefore labeled these particular programs buggy.

The remaining benchmark sets focus on buggy programs from four sources: (1) GitHub scripts, extracted from bugfix commits; (2) StackOverflow bugs identified via manual search and inspection; (3) LLM-generated programs, prompted to produce common shell mistakes; (4) Handwritten tests targeting specific composition patterns.

We collected the GitHub programs using a partially automated process. First, we selected the top twenty repositories with shell code on GitHub (according to number of stars) using the GitHub query API. For each such repository, we filtered the history of commits (total 47080) to those which (1) modify a file ending with `.sh|.bash|.zsh` and (2) the modified line contains the pipe character `|`. For every such commit (total 1028), we prompt gpt-4o-mini to identify which commits fix a bug relating to the contents of a pipeline. We manually inspected the resulting 328 commits to check their relevance, discarding commits that do not fix bugs, or for which the bug is unrelated to pipeline contents—ending up with 57 relevant commits. Each such commit provides two programs in our suite: the pre-commit script, which contains a bug (included in the buggy set), and the post-commit script, which fixes the bug (included in the correct set).

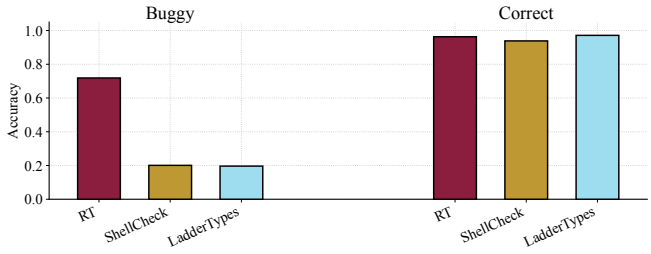


Fig. 8: Accuracy results. The bars depict the accuracy of RT, ShellCheck, and LADDERTYPES on buggy and correct programs.

We collected the StackOverflow scripts manually, using both web search (with phrases such as "shell", "bash", and "pipeline"), and the StackExchange API, filtering for questions containing the keywords "error", "bug", "pipeline", or "unexpected" in the core site and Unix & Linux Stack Exchange sub-domain. We manually reviewed the results and included only buggy programs, as these posts rarely provided corrected versions.

6.2 Q1: Effectiveness

To answer Q1, we calculate RT's accuracy in identifying composition mistakes across all benchmark sets. Accuracy is defined as the proportion of correct categorizations RT makes relative to the whole population; a system that always flags buggy programs and never correct ones is 100% accurate. We evaluate RT configured with its included type database—which covers 71/106 of the GNU coreutil commands and 86% of the command invocations in the GitHub set (*Cf.*§3.1).

Big picture: At a high level, RT has an overall accuracy of 91% (90 false positive/negatives), and identifies 87 bugs that state-of-the-art systems fail to detect. Fig. 8 breaks this down into accuracy on buggy programs (72%) and correct ones (96%)—and for programs with extra type annotations, its accuracy rises to 94% and 98% respectively. Breaking RT's judgments down into definite and possibly-buggy warning levels, RT's definite warnings are 100% accurate, while its possibly-buggy warnings are 86%—the drop in accuracy reflecting the use of approximations and heuristics in its reasoning. In a comparison across RT, ShellCheck, and LADDERTYPES, RT significantly outperforms both baseline systems on buggy programs (by 52 percentage points) while maintaining comparable accuracy on correct programs.

Detailed Analysis: Tab. 4 breaks down RT's accuracy across correct and buggy benchmark subsets, reporting true positives and negatives based on whether RT correctly signals warnings for buggy programs or refrains from warning on correct ones.

Tab. 4: Breakdown of configurations. Each column represents a different configuration of RT. The leftmost is RT with all features enabled, and following columns disable one feature: without heuristics, without FSTs, and without concretization. Rows indicate RT's true result under each configuration on correct and buggy scripts.

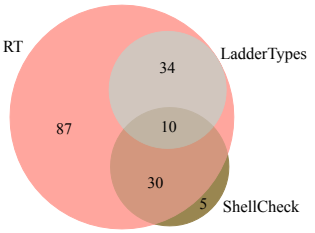
		RT	w/o heus	w/o FSTs	w/o con
w/o Ann.	Correct	703	714	583	702
	Buggy	161	94	176	161
	Acc	91%	85%	80%	90%
w/ Ann.	Correct	716	724	594	708
	Buggy	210	142	200	210
	Acc	97%	91%	83%	96%

The leftmost column reports results for the full configuration of RT. The remaining columns list results for a version of RT with one or more features disabled, thereby quantifying the benefits offered by each feature in isolation or combination. Overall, RT has 91% accuracy in the evaluation suite, with 27 false positives on correct scripts and 63 false negatives on incorrect ones. With annotations, RT has 97% accuracy on the programs, with 14 false positives on correct scripts and 14 false negatives on incorrect ones.

Sources of inaccuracy: To better understand RT's inaccurate results, Tab. 5a breaks down their causes for the maximal configuration without annotations. RT produces 11 false positives due to programs that violate heuristics, but are actually correct (*Cf.*§5.3). RT reports 16 false positives due to over-approximations in type inference. For instance, the file referenced by `${input}` in the pipeline `cat ${input} | sort -n` is unknown ahead of time. Consequently, RT conservatively infers the output of `cat` as `.*`, which is sound but imprecise. RT produces 14 false negatives for programs that are well-typed, but whose behavior is incorrect according to the program's original source (*i.e.*, the bug is not of a nature that RT could detect). RT produces 49 false negatives due to the absence of expected output specifications. Without user-provided assertion annotations to define the required format, the programs are well-typed.

Tab. 5b shows the false results for the full configuration of RT evaluated on the benchmark set with annotations. RT eliminates all false negatives caused by the absence of expected output specifications with annotations. However, RT still produces 6 false positives related to over-approximations. These are invocations that RT models soundly but imprecisely due to inherent complexity of the commands—such as complex uses of `awk` and `sed`. Properly supporting all possible invocations of these commands demands a specialized language analysis (in the case of `awk`); `sed` invocations with complex capture groups, however, pose fundamental challenges (§4.2).

Comparison with state-of-the-art systems: We compare RT's accuracy to that of LADDERTYPES and ShellCheck. Like RT, LADDERTYPES classifies programs as buggy or correct. However, LADDERTYPES by default comes with a smaller set of type declarations than RT. To enable fair comparison we manually add reasonable versions of all RT's simple type declarations and a fallback type declaration mirroring RT's. ShellCheck differs from RT and LADDERTYPES in that it emits a wide range of warnings—many irrelevant. To bring it into parity for comparison, we consider a program buggy under ShellCheck if it produces any warning related to pipelines or output stream contents,[2] and correct otherwise.



The Euler diagram on the right compares the effectiveness of these systems, showing the number of buggy programs each identifies. Overlaps indicate agreement across systems. RT is evaluated without annotations, as neither ShellCheck nor LADDERTYPES benefits from developer-supplied information. RT uniquely detects 87 bugs, compared to 5 unique to ShellCheck and 0 to LADDERTYPES. Notably, 58 bugs remain undetected by all three systems in the absence of type annotations.

Digging into the nature of the buggy programs that ShellCheck identifies but RT does not reveals that all are generic warnings that are not relevant for the particular bug at hand. While the error code ShellCheck issues is potentially relevant for composition mistakes, and hence conservatively counted in the evaluation metrics, manual inspection reveals that in every case, the bug is unrelated to ShellCheck's warning. For example, in several of the cases ShellCheck warns about irrelevant quoting in an unrelated pipeline stage.

6.3 Q2: Analysis Performance

Experimental setup: All evaluations were performed on a single machine running Ubuntu 20.04, equipped with an AMD Ryzen 7 4800H processor (2.9 GHz, 8-core) and 16 GB of RAM, using OpenJDK 17.0.2, Python 3.10, and the automaton package dk.brics.automaton (version 1.12-4).

Results: RT analyzes each program in 0.020s on average (0.009–0.903s), which is comparable to ShellCheck's 0.018s average (0.010–0.038s). In contrast, LADDERTYPES averages 3.081s per program (0.234–14.861s), roughly two orders of magnitude slower than RT; this is primarily due to an inefficient type lookup using an external shell script to search for command invocations in its type database.

[2] Ignored (irrelevant) warnings: SC2012,SC2046,SC2086,SC2018,SC2019, SC2002,SC2006,SC2009,SC2035,SC2060,SC2061,SC2062,SC2063,SC2126, SC2154,SC2185,SC2196,SC2225.

Tab. 5: Breakdown of false results. The tables break down the false results for the full configuration of RT evaluated on the benchmark set with and without annotations.

(a) Without annotations

Category	Group	#	Description	Example
False Positive	Bad heuristics	11	Correct scripts flagged by heuristic	`du -ah /workspace | sort -rh`
	Over-approximation	16	Imprecise command type	`cat ${input} | sort -n`
False Negative	Unexpressible semantics	14	Well-typed, but *e.g.*, unsorted	`... | sort | cut -d',' -f10 | uniq`
	Wrong output format	49	Expected output format is not provided	`echo "$INDEX" | grep '^ D '`

(b) With annotations

Category	Group	#	Description	Example
False Positive	Bad heuristics	8	Correct scripts flagged by heuristic	`du -ah /workspace | sort -rh`
	Over-approximation	6	Imprecise command type	`... | sed -E 's/(.*)/\1\1/'`
False Negative	Unexpressible semantics	14	Well-typed, but *e.g.*, unsorted	`... | sort | cut -d',' -f10 | uniq`

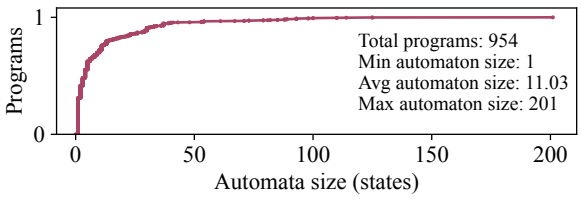


Fig. 9: Automata size across evaluated programs. The CDF plot shows the fraction of evaluated programs where the largest automaton representing a type has at most the number of states on the x-axis. The average automaton size across all programs is 11.03 states, minimum is 1, and maximum is 201.

RT's overall performance is dependent on the complexity of the types it reasons about, one proxy for which is the size of minimal automata representing the types. Fig. 9 shows the cumulative distribution of automata sizes RT constructs for the types in the evaluation suite, where each recorded automaton size corresponds to the largest DFA within a given program. Most programs contain minimal DFAs with fewer than 10 states. However, several programs include large DFAs; this occurs primarily when input or variable annotations involve lengthy regular expressions, resulting in complex DFAs even without additional transformation. Indeed, all but one of the programs taking more than 0.2s to analyze are annotated with relatively complex, large input types. The longest time is for a 5-stage pipeline at 0.90s, which has an input annotation requiring a minimal DFA with 105 states.

7 Related Work

Static analyses for the shell: Other kinds of static analyses have been proposed for the shell. ABash [44] analyzes scripts for expansion and word-splitting-related bugs, with a particular emphasis on code-injection style vulnerabilities. The CoLiS shell [34] includes a static analyzer that identifies dangerous or undesirable filesystem actions. Prior work [56] statically infers filesystem pre-conditions for file manipulation scripts. These analyses target different kinds of bugs and script behaviors than RT.

Semantic models of the shell: Semantic models of the shell provide critical underpinnings for understanding and reasoning about the semantics of shell scripts ahead of execution. Smoosh [26] defines the first mechanized formal semantics for the POSIX shell. The CoLiS language includes a formally verified interpreter [35], thereby defining the semantics of the (similar, but not POSIX shell) language. Such models inform how static analyses like RT should reason about shell programs, but are not analyses themselves.

Regular expressions for stream processing: Prior works [4, 42] have considered the use of unambiguous regular expressions to describe the hierarchical decomposition of streams into finite windows for computing quantitative queries. The restriction of unambiguity is imposed to ensure that streaming queries have a uniquely determined output. In these works, regular expressions are not used to express correctness properties or data invariants for streams.

Types for POSIX shell pipelines: LadderTypes [60] models pipeline contents using types familiar from statically typed programming languages (*e.g.*, `Int` or `Sequence[Byte]`), encoding hierarchies of types (dubbed ladders) connected by a "represented as" relationship. Hence, it is a fundamentally different type system design, with a stricter notion of compatibility: two ladder types are compatible only if the full interpretation of a downstream consumer is included in the representation of the upstream producer's output.

Regular languages for static string reasoning: Regular languages provide a well-known formalism for statically

reasoning about string-like values and their formats. Prior works [7,32] extend XML-oriented type systems with Kleene-algebraic operators. Boomerang uses regular languages to refine the basic `string` type into precise format types [9]. Prior work also uses automata or regular-language abstractions to analyze string expressions in Java [14], jQuery [53] and dynamic languages [5]. String solvers and decision procedures further reason about path feasibility for programs with rich string operations [13]. Prior work [56] also uses an automata-based representation of file paths, and leverages FSTs to manipulate them. RT differs from these analyses in (1) domain—POSIX shell streams—and (2) defining regular languages as the primitive type of its type system.

Shell replacements: Both the literature and open-source community abound with shell-like scripting languages. These include traditional-looking shells with features drawn from typical general-purpose languages [15, 19]; libraries and embeddings in general purpose languages which maintain the essential design of byte-level communication streams [1, 8, 16, 21, 23, 25, 27, 57, 65]; languages that introduce higher level structure in at least some process communication [2, 29, 46, 58]; languages that also feature a type system that can catch some composition mistakes [30, 43, 59]; and languages that offer security monitoring in shell programming [49]. Unlike this direction of work, RT is a static analysis tool for the POSIX shell rather than a new language.

Linters for the shell and shell-wrapping systems: Several shell linting tools have been developed to analyze shell scripts. Checkbashisms [10] mainly focuses on identifying the usage of shell syntax that is exclusive to the bash shell environment. PSScriptAnalyzer [45] specializes in detecting common syntax issues and code smells in PowerShell scripts, while ShellCheck [22] does a similar job for various popular shells such as Bash and zsh. SecureCode [17] extracts shell code embedded in DevOps orchestration frameworks like Ansible in order to analyze them with ShellCheck. DRIVE [67] similarly extracts shell code from Dockerfiles and applies rule-based patterns to identify code-smells and possible bugs. These tools rely on syntax-based pattern matching, which limits their analysis to syntactic issues. In contrast, RT models stream contents, enabling semantic analysis beyond predefined syntactic rules.

Automated program repair for the shell: NoFAQ is a rule-based command-repair tool [18]. It matches the given command and error message with bugfix examples in a database and suggests the corresponding fixes. RT does not aim to suggest fixes, and identifies bugs semantically rather than with rules/patterns over syntax or error messages.

8 Conclusion

RT demonstrates that regular types provide an effective and efficient basis for statically detecting input-output composition mistakes in the UNIX shell. Regular types capture significant structure in stream contents while remaining faithful to the byte-oriented nature of shell communication, leverage the rich literature around regular languages and automata, and lean into the familiarity of regular expressions that are already ubiquitous in UNIX environments.

Availability

RT's implementation, alongside all benchmarks in this paper, is available as MIT-licensed open-source software:

https://github.com/atlas-brown/rt

Acknowledgements

We are thankful to the anonymous OSDI reviewers, Brown CS2952R (Fall'24) participants, Michael Greenberg, and the Brown Systems group for their input. This material is based upon research supported by NSF awards CCF-2525351, CCF-2340479, CNS-2247687, and CNS-2312346, DARPA contract no. HR001124C0486, an Amazon Research Award (Fall 2024), a Google ML-and-Systems Junior Faculty Award, a seed grant from Brown University's Data Science Institute, and a BrownCS Faculty Innovation Award.

References

[1] Eshell: The emacs shell. https://www.gnu.org/software/emacs/manual/html_node/eshell/, 2025. Accessed: 2025-03-22.

[2] Nushell: A new type of shell. https://www.nushell.sh/, 2025. Accessed: 2025-03-22.

[3] PCRE - Perl compatible regular expressions. PCRE.org, https://www.pcre.org/, 2025.

[4] Rajeev Alur, Dana Fisman, and Mukund Raghothaman. Regular programming for quantitative properties of data streams. In Peter Thiemann, editor, *Proceedings of the 25th European Symposium on Programming (ESOP '16)*, volume 9632 of *Lecture Notes in Computer Science*, pages 15–40, Berlin, Heidelberg, 2016. Springer.

[5] Vincenzo Arceri and Isabella Mastroeni. Static program analysis for string manipulation languages. *Electronic Proceedings in Theoretical Computer Science*, 299:19–33, August 2019.

[6] Jon Bentley, Don Knuth, and Doug McIlroy. Programming pearls: a literate program. *Commun. ACM*, 29(6):471–483, June 1986.

[7] Véronique Benzaken, Giuseppe Castagna, and Alain Frisch. CDuce: an XML-centric general-purpose language. *SIGPLAN Not.*, 38(9):51–63, August 2003.

[8] Joel Berger, Jaap Karssenberg, and R.L. Zwart. Zoidberg: A modular perl shell. https://metacpan.org/pod/Zoidberg, 2025. Accessed: 2025-03-22.

[9] Aaron Bohannon, J. Nathan Foster, Benjamin C. Pierce, Alexandre Pilkiewicz, and Alan Schmitt. Boomerang: resourceful lenses for string data. In *Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '08, pages 407–419, New York, NY, USA, 2008. Association for Computing Machinery.

[10] R Braakman, J Rodin, J Gilbey, and M Hobley. Checkbashisms. https://sourceforge.net/projects/checkbaskisms/, 2015. Accessed: 2024-11-26.

[11] Luca Cardelli and Peter Wegner. On understanding types, data abstraction, and polymorphism. *ACM Computing Surveys (CSUR)*, 17(4):471–523, 1985.

[12] Agnishom Chattopadhyay, Angela W. Li, and Konstantinos Mamouras. Verified and efficient matching of regular expressions with lookaround. In *Proceedings of the 14th ACM SIGPLAN International Conference on Certified Programs and Proofs*, CPP '25, pages 198–213, New York, NY, USA, 2025. ACM.

[13] Taolue Chen, Matthew Hague, Anthony W. Lin, Philipp Rümmer, and Zhilin Wu. Decision procedures for path feasibility of string-manipulating programs with complex operations. *Proc. ACM Program. Lang.*, 3(POPL), January 2019.

[14] Aske Simon Christensen, Anders Møller, and Michael I Schwartzbach. Precise analysis of string expressions. In *International Static Analysis Symposium*, pages 1–18. Springer, 2003.

[15] Andy Chu. Oils: Our upgrade path from bash to a better language and runtime. https://oils.pub/, 2025. Accessed: 2025-03-22.

[16] David Crawshaw. Neugram: scripting language integrated with go. https://github.com/neugram/ng, 2025. Accessed: 2025-03-22.

[17] Ting Dai, Alexei Karve, Grzegorz Koper, and Sai Zeng. Automatically detecting risky scripts in infrastructure code. In *Proceedings of the 11th ACM Symposium on Cloud Computing*, SoCC '20, pages 358–371, New York, NY, USA, 2020. Association for Computing Machinery.

[18] Loris D'Antoni, Rishabh Singh, and Michael Vaughn. NoFAQ: synthesizing command repairs from examples. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2017, pages 582–592, New York, NY, USA, 2017. Association for Computing Machinery.

[19] Elvish Developers. Elvish shell. https://elv.sh/, 2025. Accessed: 2025-03-22.

[20] Mark Jason Dominus. Perl regular expression matching is np-hard. https://perl.plover.com/NPC/. Accessed: 2025-04-15.

[21] Dundalek. Closh: Bash-like shell based on clojure. https://github.com/dundalek/closh, 2025. Accessed: 2025-03-22.

[22] Vidar Holen et al. Shellcheck: A shell script static analysis tool. https://www.shellcheck.net/, 2012. Accessed: 2024-10-14.

[23] Tomer Filiba. Plumbum: Shell combinators and more. https://plumbum.readthedocs.io/en/latest/, 2025. Accessed: 2025-03-22.

[24] Inc. GitHub. The state of open source software. https://octoverse.github.com/#top-languages-over-the-years, 2024. Accessed: 2024-11-01.

[25] Gabriel Gonzalez. Turtle: Shell programming, haskell style. https://hackage.haskell.org/package/turtle, 2025. Accessed: 2025-03-22.

[26] Michael Greenberg and Austin J. Blatt. Executable formal semantics for the POSIX shell. *PACMPL*, 4(POPL):43:1–43:30, December 2019.

[27] Li Haoyi. Ammonite: Scala scripting. https://ammonite.io/, 2025. Accessed: 2025-03-22.

[28] Robert Harper. *Practical Foundations for Programming Languages*. Cambridge University Press, 2 edition, 2016.

[29] William Gallard Hatch and Matthew Flatt. Rash: from reckless interactions to reliable programs. *ACM SIGPLAN Notices*, 53(9):28–39, 2018.

[30] Alec Heller and Jesse A. Tov. Caml-shcaml: an ocaml library for unix shell programming. In *Proceedings of the 2008 ACM SIGPLAN Workshop on ML*, ML '08, pages 79–90, New York, NY, USA, 2008. Association for Computing Machinery.

[31] John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Pearson Education, 3rd edition, 2006.

[32] Haruo Hosoya, Jérôme Vouillon, and Benjamin C. Pierce. Regular expression types for xml. In *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming*, ICFP '00, page 11–22, New York, NY, USA, 2000. Association for Computing Machinery.

[33] Jeroen Janssens. *Data science at the command line*. O'Reilly Media, Sebastopol, CA, October 2014.

[34] Nicolas Jeannerod. *Verification of Shell Scripts Performing File Hierarchy Transformations*. PhD thesis, University of Paris, 2021.

[35] Nicolas Jeannerod, Claude Marché, and Ralf Treinen. A Formally Verified Interpreter for a Shell-like Programming Language. In *9th Working Conference on Verified Software: Theories, Tools, and Experiments*, volume 10712, Heidelberg, Germany, July 2017.

[36] Igibek Koishybayev, Aleksandr Nahapetyan, Raima Zachariah, Siddharth Muralee, Bradley Reaves, Alexandros Kapravelos, and Aravind Machiry. Characterizing the security of github {CI} workflows. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 2747–2763, 2022.

[37] Shriram Krishnamurthi. *Programming Languages: Application and Interpretation*. 3 edition, 2003.

[38] Evangelos Lamprou, Ethan Williams, Georgios Kaoukis, Zhuoxuan Zhang, Michael Greenberg, Konstantinos Kallas, Lukas Lazarek, and Nikos Vasilakis. The Koala benchmarks for the shell: Characterization and implications. In *Proceedings of the 2025 USENIX Annual Technical Conference (USENIX*

ATC '25), pages 449–64, Boston, MA, July 2025. USENIX Association.

[39] Lukas Lazarek, Seong-Heon Jung, Evangelos Lamprou, Zekai Li, Anirudh Narsipur, Eric Zhao, Michael Greenberg, Konstantinos Kallas, Konstantinos Mamouras, and Nikos Vasilakis. From ahead-of- to just-in-time and back again: Static analysis for Unix shell programs. In *2025 Workshop on Hot Topics in Operating Systems*, HotOS '25, pages 88–95, New York, NY, USA, 2025. Association for Computing Machinery.

[40] Konstantinos Mamouras and Agnishom Chattopadhyay. Efficient matching of regular expressions with lookaround assertions. *Proceedings of the ACM on Programming Languages*, 8(POPL):Article 92, 31 pages, 2024.

[41] Konstantinos Mamouras, Alexis Le Glaunec, Wu Angela Li, and Agnishom Chattopadhyay. Static analysis for checking the disambiguation robustness of regular expressions. *Proceedings of the ACM on Programming Languages*, 8(PLDI):Article 231, 25 pages, 2024.

[42] Konstantinos Mamouras, Mukund Raghothaman, Rajeev Alur, Zachary G. Ives, and Sanjeev Khanna. StreamQRE: Modular specification and efficient evaluation of quantitative queries over streaming data. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '17, pages 693–708, New York, NY, USA, 2017. ACM.

[43] Kouji Matsui. A proposal for an interactive shell based on a typed lambda calculus. https://arxiv.org/abs/2104.03678, 2021.

[44] Karl Mazurak and Steve Zdancewic. ABash: Finding bugs in bash scripts. In *PLAS*, pages 105–114, San Diego California USA, June 2007. ACM.

[45] Microsoft. Psscriptanalyzer. https://github.com/PowerShell/PSScriptAnalyzer, 2020. Accessed: 2024-11-26.

[46] Microsoft. What is powershell? https://learn.microsoft.com/en-us/powershell/scripting/overview, 2025. Accessed: 2025-03-22.

[47] Takayuki Miyazaki and Yasuhiko Minamide. Derivatives of regular expressions with lookahead. *Journal of Information Processing*, 27:422–430, 2019.

[48] Mehryar Mohri. Finite-state transducers in language and speech processing. *Computational Linguistics*, 23(2):269–311, 1997.

[49] Scott Moore, Christos Dimoulas, Dan King, and Stephen Chong. {SHILL}: A secure shell scripting language. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 183–199, 2014.

[50] Jason Morris, Chris McCubbin, and Raymond Page. *Hands-On Data Science with the Command Line*. Packt Publishing, Birmingham, England, January 2019.

[51] Shaun Nichols. Scary code of the week: Steam cleans linux pcs. https://www.theregister.com/2015/01/17/scary_code_of_the_week_steam_cleans_linux_pcs/, January 2015. Accessed: 2025-03-06.

[52] OpenAI. Gpt-4o system card. https://arxiv.org/abs/2410.21276, 2024.

[53] Changhee Park, Hyeonseung Im, and Sukyoung Ryu. Precise and scalable static analysis of jquery using a regular expression domain. In *Proceedings of the 12th Symposium on Dynamic Languages*, DLS 2016, pages 25–36, New York, NY, USA, 2016. Association for Computing Machinery.

[54] Benjamin C. Pierce. *Types and Programming Languages*. The MIT Press, 1st edition, 2002.

[55] Deepti Raghavan, Sadjad Fouladi, Philip Levis, and Matei Zaharia. POSH: A Data-Aware shell. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pages 617–631. USENIX Association, July 2020.

[56] Rodney Rodriguez and Xiaoyin Wang. Understanding execution environment of file-manipulation scripts by extracting pre-conditions. In *2021 IEEE/ACM 29th International Conference on Program Comprehension (ICPC)*, pages 406–410. IEEE, 2021.

[57] Anthony Scopatz. Xonsh: Python-powered shell. https://xon.sh/, 2025. Accessed: 2025-03-22.

[58] Olin Shivers. Scsh manual 0.6.7. https://scsh.net/docu/html/man.html, 2006.

[59] Jon Shultis. A functional shell. In *Proceedings of the 1983 ACM SIGPLAN Symposium on Programming Language Issues in Software Systems*, SIGPLAN '83, pages 202–211, New York, NY, USA, 1983. Association for Computing Machinery.

[60] Michael Sippel and Horst Schirmeier. Process composition with typed unix pipes. In *Proceedings of the 12th Workshop on Programming Languages and Operating Systems*, pages 34–40, 2023.

[61] Slashdot. itunes 2.0 installer deletes hard drives. https://apple.slashdot.org/story/01/11/04/0412209/itunes-20-installer-deletes-hard-drives, 2001. Accessed: 2025-03-06.

[62] Valve Software. Moved /.local/share/steam. ran steam. it deleted everything on system owned by user. https://github.com/ValveSoftware/steam-for-linux/issues/3671, 2023. Accessed: 2023-10-04.

[63] The Open Group. Regular expressions – posix basic regular expressions. https://pubs.opengroup.org/onlinepubs/9699919799/basedefs/V1_chap09.html, 2018. Accessed: April 15, 2025.

[64] Nikos Vasilakis, Konstantinos Kallas, Konstantinos Mamouras, Achilles Benetopoulos, and Lazar Cvetković. PaSh: Light-touch data-parallel shell processing. In *Proceedings of the Sixteenth European Conference on Computer Systems*, EuroSys '21, page 49–66, New York, NY, USA, 2021. Association for Computing Machinery.

[65] Greg Weber, Petr Rockai, Andreas Abel, John Wiegley, Michael Snoyman, and psibi. Shelly: Shell-like (systems) programming in haskell. https://hackage.haskell.org/package/shelly, 2025. Accessed: 2025-03-22.

[66] John Yang, Akshara Prabhakar, Karthik Narasimhan, and Shunyu Yao. Intercode: Standardizing and benchmarking interactive coding with execution feedback, 2023.

[67] Yu Zhou, Weilin Zhan, Zi Li, Tingting Han, Taolue Chen, and Harald Gall. Drive: Dockerfile rule mining and violation detection. *ACM Trans. Softw. Eng. Methodol.*, 33(2), 2023.

A Syntactic Sugar Definitions

Syntactic Sugar	Regular Type
{ipv4}	((25[0-5]|2[0-4][0-9]|1?[0-9][0-9]?)\.){3}(25[0-5]|2[0-4][0-9]|1?[0-9][0-9]?)
{integer}	[-+]?[0-9]+
{float}	[-+]?([0-9]+\.?[0-9]*|[0-9]*\.[0-9]+)
{hex}	0x[0-9a-fA-F]+
{percentage}	(100(\.0+)?|[0-9]{1,2}(\.[0-9]+)?)%
{currency_usd}	\$[ ]?[0-9]+(\.[0-9]+)?
{mac_address}	([0-9a-fA-F]{2}[:-]){5}([0-9a-fA-F]{2})
{email_address}	[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,}
{url}	https?:\/\/[a-zA-Z0-9.-]+\.[a-zA-Z]{2,}[^]*
{iso_date}	[0-9]{4}-(0[1-9]|1[0-2])-(0[1-9]|[12][0-9]|3[01])
{year}	[12][0-9]{3}
{month_name}	[Jj]an(uary)?|[Ff]eb(ruary)?|[Mm]ar(ch)?|[Aa]pr(il)?|[Mm]ay|[Jj]un(e)?|...
{day_of_week}	[Mm]on(day)?|[Tt]ue(sday)?|[Ww]ed(nesday)?|[Tt]hu(rsday)?|[Ff]ri(day)?|...
{time_24h}	([01]?[0-9]|2[0-3]):[0-5][0-9](:[0-5][0-9])?
{time_12h}	([1-9]|1[0-2])(:[0-5][0-9](:[0-5][0-9])?)?[ ]?[AaPp][Mm]
{uuid}	[0-9a-fA-F]{8}-[0-9a-fA-F]{4}-[0-9a-fA-F]{4}-[0-9a-fA-F]{4}-[0-9a-fA-F]{12}
{hex_color}	#([0-9a-fA-F]{3}|[0-9a-fA-F]{6})
{semantic_version}	(0|[1-9][0-9]*)\.(0|[1-9][0-9]*)\.(0|[1-9][0-9]*)
{abs_path}	(/[^/]+)+/?|/
{file_extension}	\.[a-zA-Z0-9]+
{hidden_file}	\.[a-zA-Z0-9_.-]+
{perms}	[-d]([r-][w-][x-]){3}
{html_tag}	<[a-z]+([^<>]+)*(>.*<\/[a-z]+>|[ ]+\/>)
{domain_name}	([a-zA-Z0-9]([a-zA-Z0-9-]{0,61}[a-zA-Z0-9])?\.)+[a-zA-Z]{2,}
{subdomain}	([a-zA-Z0-9-]+\.)+
{url_slug}	[a-z0-9]+(-[a-z0-9]+)*
{ipv6}	([0-9a-fA-F]{1,4}:){7}[0-9a-fA-F]{1,4}|([0-9a-fA-F]{1,4}:){1,7}:|::1|::
{us_phone}	(\+?1[-.]?)?\(?[0-9]{3}\)?[-.]?[0-9]{3}[-.]?[0-9]{4}
{base64}	([A-Za-z0-9+/]{4})*([A-Za-z0-9+/]{2}==|[A-Za-z0-9+/]{3}=)?
{md5}	[a-fA-F0-9]{32}
{sha1}	[a-fA-F0-9]{40}
{sha256}	[a-fA-F0-9]{64}
{subnet_mask}	(255\.){3}(0|128|192|224|240|248|252|254|255)
{username_unix}	[a-z_][a-z0-9_-]*

B Artifact Appendix

Abstract

The artifact for RT packages the prototype checker, command specifications, documentation, tests, benchmark inputs, and reproduction scripts used for the evaluation in §6. It is organized to support three tasks: inspecting how the implementation realizes the regular-type analysis described in the paper, running functional checks for the core analyzer, and regenerating the principal evaluation artifacts, including the accuracy summaries, ablation and false-result breakdowns, bug-detection comparison, timing table, and automata-size plots.

Scope

The artifact supports the paper's design and evaluation claims. In particular, it covers: (1) the RT implementation of regular stream and command types from §3, the type-checking procedure in §3.3, and the regular-language operators and FST machinery from §4; (2) the configurable mechanisms in §5, including environment concretization, annotations, and heuristics; (3) the motivating example and the seven benchmark sets discussed in §2 and §6; and (4) the automation for reproducing the main tables and figures reported in §6.

Contents

The artifact includes the following components:

- the RT checker implementation and command type database;
- support code for regular-language operations, finite-state transducers, annotations, concretization, and heuristics;
- documentation, including the top-level README and artifact-evaluation instructions;
- functional tests and smoke-test examples, including the motivating example from §2;
- the benchmark suites used in the paper's evaluation and the configuration needed to run baseline comparisons against ShellCheck and LADDERTYPES; and
- scripts for checking the artifact, reproducing the full evaluation, and generating the resulting tables and plots.

Hosting

The artifact is publicly available as a GitHub repository at `https://github.com/atlas-brown/rt`.